CS 8803 Project 2 Report (Team 44)

Thomas France
tfrance@gatech.edu

Eric Thompson
ethompson44@gatech.edu

Mian Murad Ahmed
mian.ahmad@gatech.edu

Osman Din
osman.din@gatech.edu

Abstract

Kalman filters, EKF, k-Nearest Neighbors, Random Forests and particle filters: The team implemented several different algorithms and approaches for tackling the prediction problem for the micro robot. Different motion models were explored, including linear, circular and Brownian motion. The best results were achieved with particle filters and a simple motion model.

1 Introduction

HEXBUG Nano is a micro-robot that propels forward and explores its environment by using the physics of its vibration. A solution for predicting the motion of the robot, therefore, must tackle several different challenges which the robot faces (as demonstrated in the videos):

- Motion without collision
- Collision with box boundaries and the candle
- Flipping over
- Generating all predictions within a minute

1.1 Approaches

In a nutshell, we implemented simple linear prediction, filtering using the Kalman filter, prediction with EKF, regression, and particle filters. The following is the complete list of the approaches the team explored:

- Particle filters
- Kalman filter, Extended Kalman Filter (EKF) with k-nearest neighbors (kNN) and EKF with Brownian motion model
- Regression using kNN and random forests
- Generating random slices of data to avoid overfitting
- Incorporating a collision model

We performed image processing and provide our custom set of test points as input to the particle filters. The image preprocessing step was straight-forward: based on background subtraction, thresholding, and a morphological open operation, take the center of mass of the remaining pixels. The key to the success was the conversion to grayscale. The Hexbug has a noticeable green area on its back, the only noticeable green in the image. So the grayscale values were calculated as `green - red - blue`. This resulted in the same patch of the Hexbug standing out nicely in all the images.

Machine learning algorithms such as kNN, random forests, neural networks and SVM were explored initially. The only approaches that showed promise (especially given the expertise of the team and limited time) were kNN and random forests. We coded the solution for kNN with and without localization.

The motivation behind the machine learning algorithms is that these algorithms can handle the environment (like the boundary and the candle) simultaneously and variations in the robot's motion. The programmer does not have to worry about box boundaries or candles nor does she have to account for a collision model.

2 Temporal models

2.1 Particle filtering

The particle filter approach models many possible paths of the Hexbug at the same time, then averages the position of the particles to get its current estimate. A particle filter algorithm is a good method for a problem like this. Because the generated particles modeling the Hexbug (like the Hexbug itself) are constrained to a rectangular box, they tend to bounce around over time, covering more and more of the box. This represents the increasing uncertainty of our estimate the further we get from our last measurement. The average of the particles therefore tends toward the center of the box. While this average does not correspond to the most likely position, it does reduce the worst-case error.

The Hexbug runs at a fairly steady top speed of about 8 pixels/second. It slows down when it runs into walls, then accelerates back toward its top speed. We model this acceleration using the following formula:

```
new_speed = old_speed + (max_speed - old_speed) * speed_recovery_factor
```

Note that this will also decelerate the particle back toward its top speed if noise added to the speed causes it to go faster.

Collisions with the wall or candle cause the Hexbug (and the particle) to change direction. A head-on collision results in little change in direction, as does a glancing blow. However, a collision at 45 degrees results in a significant angle change. We use a simplified, 2-mode collision model. For collisions with an angle to the surface > 60 degrees, the formula is:

```
new_angle = (90 - old_angle) * 2 + 90
```

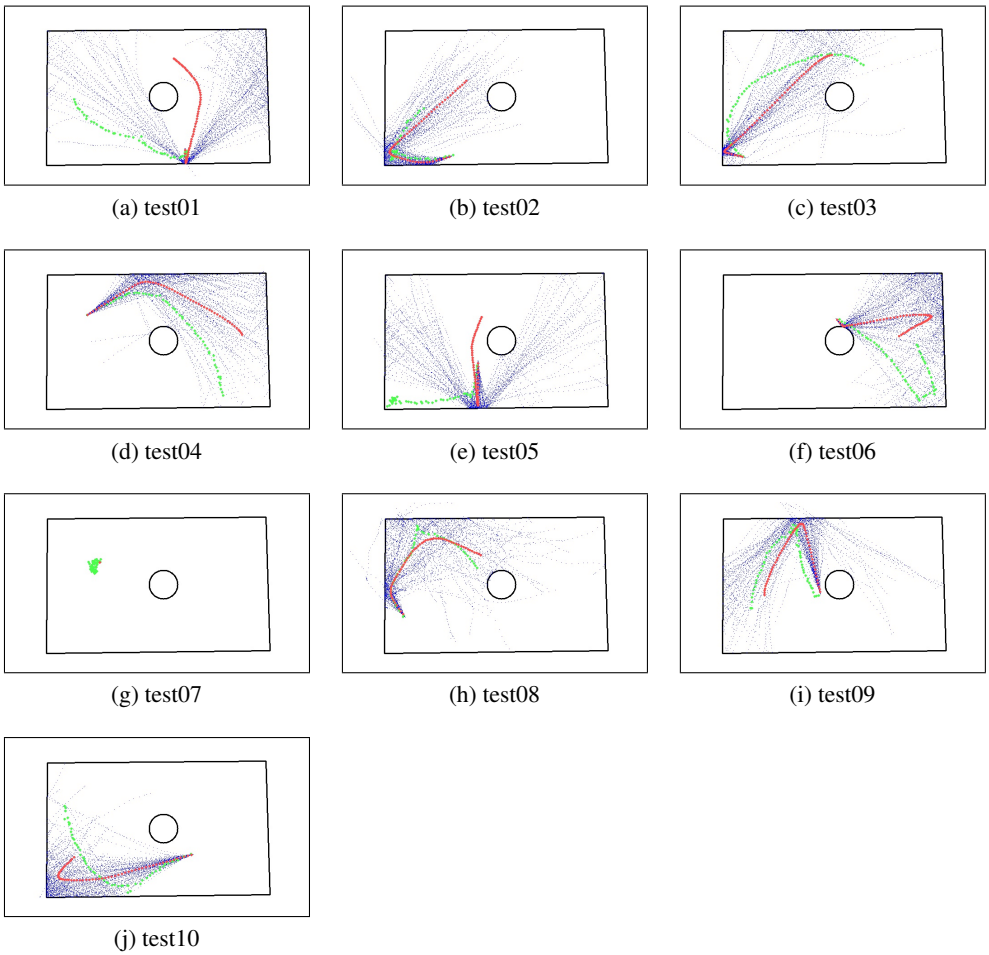
This results in the particle still pointing toward the wall, but not quite so head-on. For collisions at an angle < 60 degrees, the particle is mirrored off the surface.

```
new_angle = 360 - old_angle
```

The particle filter is using our own input point set with much less noise than the project's original input dataset, therefore it was not necessary to start particle filtering until the end of the measured data, starting from the last measured position, heading in the last measured direction at the last measured speed. At each iteration we perform the following steps:

1. Adjust the speed toward the top speed
2. Add Gaussian noise to the speed ($\mu = 0$ and σ= `speed noise std dev`)
3. Add Gaussian noise to the direction ($\mu = 0$ and σ=`angle noise std dev`)
4. Calculate the next particle position based on the speed and direction
5. Check for collisions with each of the four walls, and update direction and speed accordingly
6. Check for collisions with the candle, and update direction and speed accordingly
7. Calculate the average of the particle positions and use this as our next estimated point

A consideration when implementing particle filters is how many particles to use. More particles provide more accurate results, but take more time to process. We found that 100 particles gave pretty



(a) test01 (b) test02 (c) test03

(d) test04 (e) test05 (f) test06

(g) test07 (h) test08 (i) test09

(j) test10

Figure 1: Particle filter approach prediction visualization. The images were created with 100 particles for clarity (the actual implementation uses 5000 particles).

good results, with an average score around 550, under two seconds for the entire set of tests. However, some scores would be below 540, and some would be above 600. By increasing the particle count to 5000, we consistently get a score in the 535-540 range, at about 12 seconds for the entire test set. Since 12 seconds fell well within the allotted time limit, we went with 5000 particles.

Table 1 describes the algorithm's four tunable parameters whose values were determined using the Twiddle algorithm. Figure 1 on Page 3 shows the visualization for the predictions.

In addition to the particle filter approach, our solution also includes a special case for a tipped-over Hexbug. A tipped-over Hexbug travels little, so over the short term the best strategy is to stay in the same place. Over the longer term it is possible for the Hexbug to correct itself, after which there is no way to determine which way it will be heading. If we had to predict further into the future, the tipped-over state could become a state of each particle, with particles exiting the tipped over state with a certain probability in each iteration.

Table 1: Parameters for the particle filter approach

`angle noise std dev`	5.0
`speed noise std dev`	2.695
`speed recovery factor`	0.2028
`max speed`	8.38

This would cause the predicted location to start moving as particles exited the tipped over state, and would result in the estimated position drifting toward the center of the box. However, we don't have many cases of the bug flipping over in the test data, so it would be difficult to come up with a good probability estimate.

2.2 Kalman filter

We started exploring Hexbug motion using a standard Kalman filter (c.f. kalman.py), as described in the course. The filter had proven rather effective in robot localization experiments, so we set one to work, tracking the Hexbug's erratic paths.

This simple model just takes the past `n` measurements to compute the next positions of the robot with respect to its previous motion. This approach was used with Kalman filter. The following equation describes the model [1].

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} v\Delta t cos\theta_t \\ v\Delta t sin\theta_t \\ w_t \Delta t \end{pmatrix}$$

Our first filter used 4 parameters to track position and distance in both x and y dimensions. From the output, it is apparent that model predictions quickly diverge from known measurements, and, after approximately 100 steps, begin to converge (see Figure 2(a)). Hexbug motions are in green, the predictions are in red. Once the model converges, position estimations tend to be confined to a small region of the box (in Figure 2(a)), based on test01.txt, the upper right hand quadrant of the box). To overcome the tendency of the model to converge quickly, we incorporated additional noise into the covariance matrix P using the equation `P = F * P * F.transpose() + Ex`, where Ex adds additional variance to each term at each time step. This idea of adding noise to prevent premature convergence is based on discussion of tracking with Kalman filtering was found in a tutorial [3]. Tuning the model in this way resulted in significantly greater tracking accuracy, as can be seen in Figure 2(b) (based on test01.txt).

Next we attempted to apply this model to predict 60 steps beyond the given test data. The approach used the output from the previous time step as input to the next estimation. From Figure 2(c), it is apparent that such a simplistic solution does not yield good results. The model predicts a straight line path with constant heading and velocity.

We then altered the model parameters from 4 to 3 dimensions, employing one parameter for heading and two for position in the x and y dimensions, similar to what we have explored in class exercises. The intention behind the change was to better model the variability in the Hexbug's motion, especially its heading. For this approach, updating heading and velocity was based on keeping track of average turn and position change (distance) across time steps. While the tracking was good, the predictions fared no better than the first approach (see Figure 2(d)).

At this point, we attempted to improve predictions using a standard trigonometric model. Having collected information about average turn and velocity in the first phase, we could easily generate likely paths, assuming constant velocity and heading. Clearly, that is an ideal assumption, as there are numerous sources of noise in the Hexbug motion and its environment, and it is also possible that the Hexbug accelerates under certain conditions. The first tests of predictions based on trigonometric functions and our model assumptions revealed that we had to incorporate boundary collisions into the model (see Figure 2(e)). We determined the box boundaries based on average minimum and maximum values in the x and y dimensions. We used averages rather than absolute values on the assumption of measurement noise. The results were much improved (see Figure 2(f)). At this point we graded our predictions by modeling on the first 1740 measurements, and reserving the final 60 as the actual against which we would make predictions. With this model and its assumptions, we achieved an L2 score of 1203.

To improve this score, we needed to rethink our assumptions about Hexbug motion, namely, the assumption of constant turn angle and velocity, as well as the billiard ball-like interactions with the box boundaries. We included a variety of parameters to capture model bias in the turn rate, velocity, and collision behavior. We noticed, for example, that upon collision the Hexbug's angle of deflection is rarely equal to the angle of incidence. In fact, the exit angle is often much flatter (less acute) than its entry angle, and we incorporated a parameter, `alpha`, to tune this. Furthermore, it appears that there

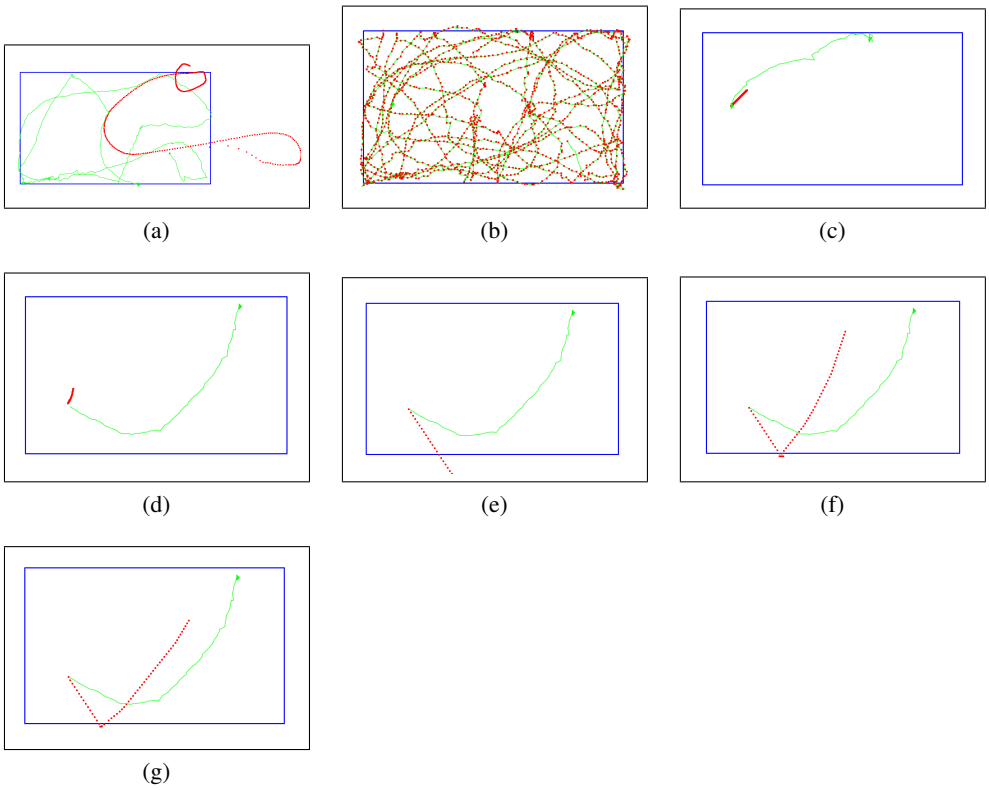


Figure 2: Kalman filter prediction visualization

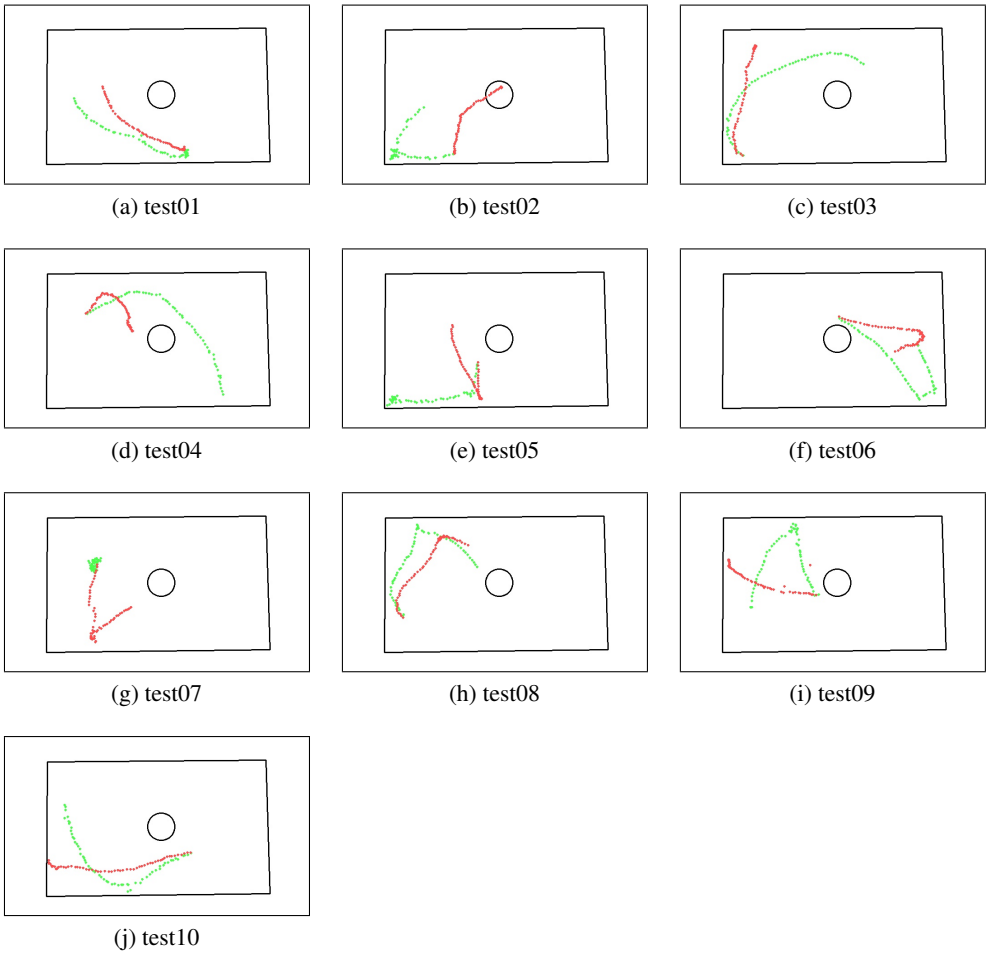
is a bias in the Hexbug's heading, with a tendency to list to the left. Again, we added a parameter, `beta`, to tune this bias. Finally, we noticed that the Hexbug's velocity is affected after a collision, with a tendency to remain in place for several time steps (especially in corners) before heading off on a new trajectory. We modeled this behavior with a third parameter, `gamma`, that allowed us to dynamically reduce the bug's velocity upon impact and restoring it to the initial rate in subsequent time steps. This model also works in the corners, where repeated impacts keeps the velocity low for a greater number of time steps. Tuning these parameters allowed us to reduce our L2 score to 960.

Having improved model performance with additional parameters, we needed to confirm our findings against further test sets in order to determine if we were overfitting the data. We created numerous and often overlapping slices of the training data (trainingdata.txt) to compare with our initial findings. On repeated sets of 20 slices, for example, we saw L2 scores rise to about 1050, which suggests that we were indeed over-tuning the model against the original test data. In the light of this finding, adding additional parameters to our model to further fine tune behavior seemed counterproductive. In the end, the trigonometric model of prediction, combined with a simple Kalman filter, yielded results no better than an L2 score of approximately 1000.

2.3 Extended Kalman filter

Our velocity based model immitated the runaway robot motion, where the robot can be thought of moving approximately in a circle (perhaps with the candle acting as the center). The wooden box corners were chosen as the landmarks for measurements. The motion model is given by the following equation [2]

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} -rsin\theta + rsin(\theta + \hat{w}\Delta t) \\ rcos\theta - rcos(\theta + \hat{w}\Delta t) \\ \hat{w}\Delta t \end{pmatrix}$$



(a) test01 (b) test02 (c) test03

(d) test04 (e) test05 (f) test06

(g) test07 (h) test08 (i) test09

(j) test10

Figure 3: EKF (with kNN) prediction visualization

where

$$r = \frac{\hat{v}}{\hat{w}}$$

The measurement model is given as [2]:

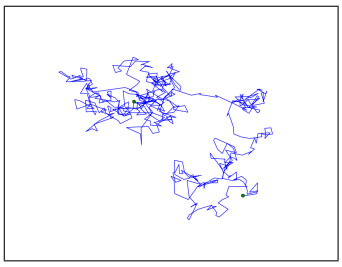
$$\begin{pmatrix} r_t^i \\ \phi_t^i \\ s_t^i \end{pmatrix} = \begin{pmatrix} \sqrt{(m-x)^2 + (m-y)^2} \\ atan2(m_{j,y} - y_t, m_{j,x} - x_t) - \theta \\ m_{j,s} \end{pmatrix}$$

The idea for using EKF and kNN was to use past predictions to inform the filter. The motivation for using EKF was to account for non-linearities in the robot's motion. EKF works well, after all, where the nonlinear motion function is smooth or diverges from linearity at a slow rate. EKF with kNN generated an average score of 750 (even without accounting for easy cases, like the robot flipping over or staying too long at a corner), but the run time was much worse compared to particle filters or pure kNN, possibly due to EKF implementation issues. Hence, this approach was not paid attention to in the team's final push.

2.4 Brownian motion

The initial motivation for trying the Brownian motion model with EKF was to simulate the randomness of Hexbug's movement.

Figure 4: Brownian motion



The model did not prove helpful as the motion of Hexbug turned out to be less random than originally expected. The motion and measurement models are as follows [4].

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \Delta x_{k+1} \\ \Delta y_{k+1} \end{bmatrix} = \begin{bmatrix} exp(-0.25(x_k + 1.5\Delta x_k)) \\ exp(-0.25(y_k + 1.5\Delta y_k)) \\ exp(-0.25\Delta x_k) \\ exp(-0.25\Delta y_k) \end{bmatrix} + w_k$$

$$\begin{bmatrix} xm_k \\ ym_k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \\ \Delta x_k \\ \Delta y_k \end{bmatrix} + v_k$$

The model could still be useful for cases where the robot stays in a place for an extended period of time. This possibility was not explored due to time constraints.

3 Machine Learning

3.1 k-Nearest Neighbors

We started with coding the class `Hexbug` (in hexbug.py), which trained on features such as the last point, speed (distance from the last point) and the angle from the last point. The `Hexbug` class sets up separate regressors for x and y coordinates. (This means that to estimate the next x, the algorithm would take n last x coordinate values based on parameters we pass but it would take the speed (distance) and angle based on <x, y> coordinates. The regressor for y value would operate similarly.) The algorithms normalize distances and angles. (We see the results deteriorate if we normalized the x or y points.)

We have programmed the Python class in such a manner that the methodology chosen, i.e., the last point, speed, angle, number of last points (e.g., 2, 3, 4), machine learning algorithm (either kNN or random forests) and the parameters of the algorithms (number of neighbors in kNN and number of estimators for random forests) can be changed. Other parameters of machine learning algorithm were not thought to be important to change. The only parameter that we would have explored given more time would have been weights metrics in kNN. We used "uniform" setting as the value but could have explored "distance". We ran both algorithms, varying the parameters, and the results are presented in the next section.

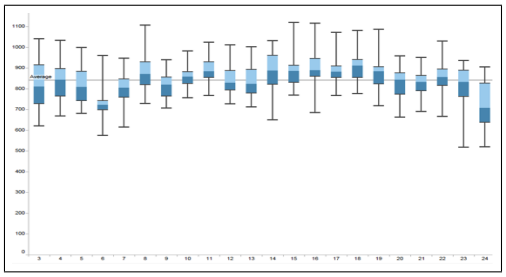
The lowest error value for kNN algorithm was found to be 512 at 7 neighbors and 23 last points.

This result shows that the lowest error values are obtained with 6 to 7 last points or between 20 to 24. To experiment we ran the algorithm with up to 60 points to see whether the performance would improve further. Intuitively, it makes sense as the algorithm knows about more previous points it should perform better.

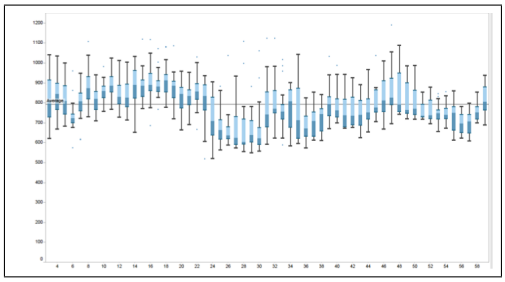
We found that it overall performs the best between 24 to 30 previous points. This means that for KNN algorithm trained on the data of the previous 1 second would produce the best results. Please see Fig. 5(a) for a visual overview.

The heat map for the kNN (below) confirmed our intuition and previous results. The best results (darker region) are around between 24 to 36 last points (about 1 second) of the previous data. Please see Fig. 5(b) for a summary.

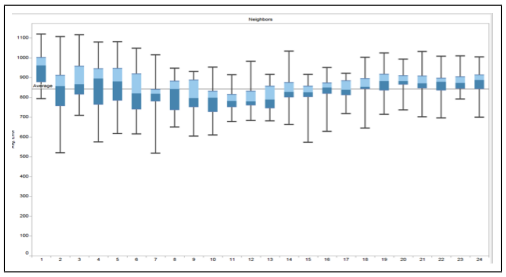
The algorithm is time efficient and took 9 to 17 seconds to execute. The number of features has more impact than the number of neighbors (but it is only 8 seconds, as stated earlier). Increasing neighbors from 1 to 24 only resulted in 2 - 3 seconds increase. Please see Fig. 5(c).



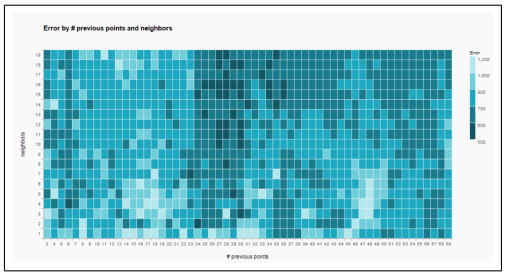
(a) KNN average error based on grade.py as compared to how many last points were used.



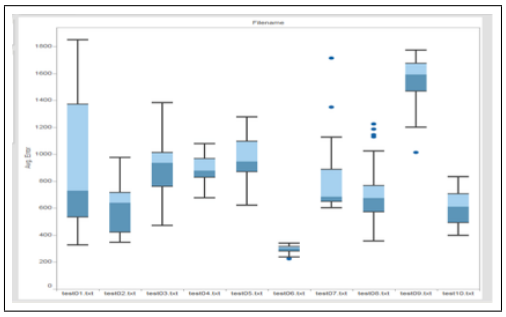
(b) Average error versus neighbors



(c) KNN average error as compared to neighbors



(d) KNN heat map for performance



(e)

Figure 5: kNN statistics

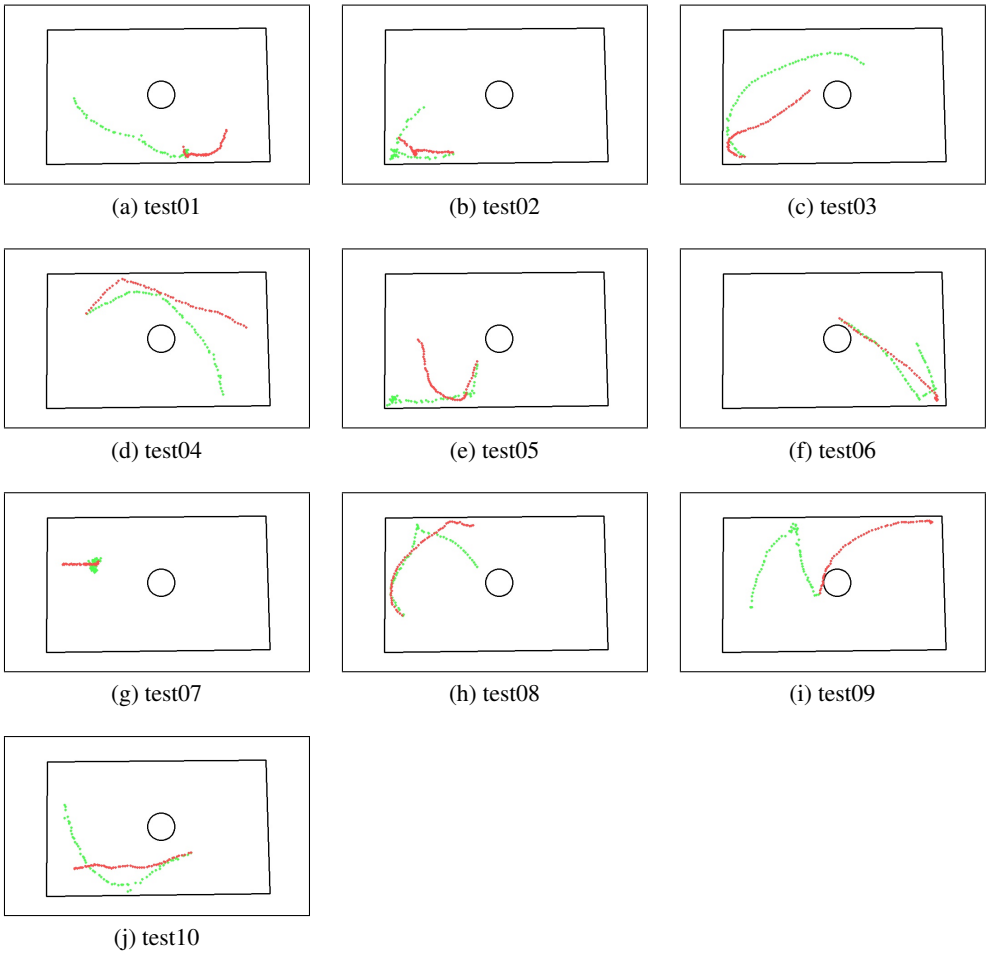
Test files 1, 3, 7 and 9 have consistently high scores and in test file 7 the robot has rolled over so it is difficult for machine learning to work. However, files 3 and 9 have a sudden change in direction or unexpected sliding along the wall respectively. One solution could be to even use test file data to train the machine learning algorithms. However, we did not invest time in it as its benefits were not clear.

One idea was whether number of features made the errors smaller and we investigated that. However, as you can see there is not much difference in the results as the number of last points used as features increases.

These challenges are not insurmountable. One could improve performance by:

- Choosing new features like robot flipping over
- Training on more data using test files

The trajectory predictions are given on the following page.



(a) test01 (b) test02 (c) test03

(d) test04 (e) test05 (f) test06

(g) test07 (h) test08 (i) test09

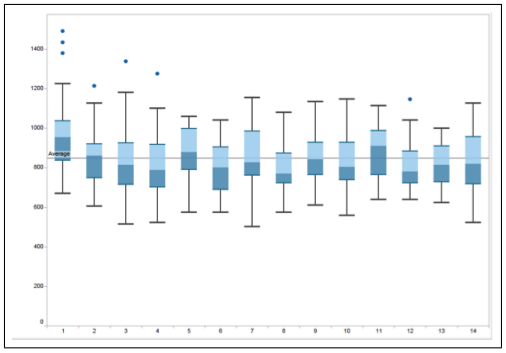
(j) test10

Figure 6: kNN prediction visualization

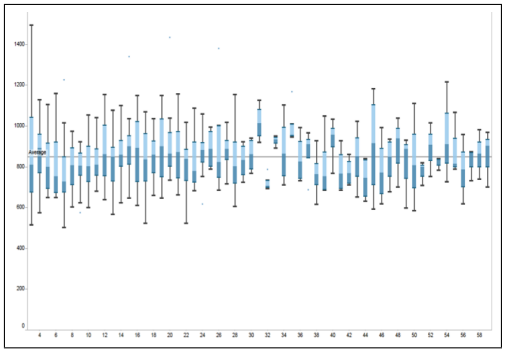
The predication accuracy visualizations for the kNN based approach are shown in Figure 6.

The algorithm performs well in most cases, but for files such as test09, it fails to acquire the correct trajectory altogether, possibly due to a lack of training examples or due to sudden or unprecented change in trajectory.

3.2 Random forests

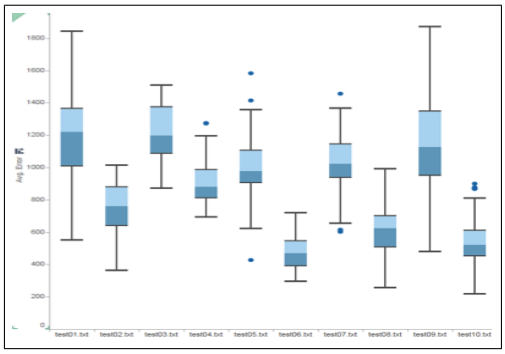


(a) Random Forest results for different number of n samples from 1 - 14



(b) Random Forest results with the different number of last point 3 to 60 (last two seconds)

(c) random forest heat map



(d)

Figure 7: Random forest statistics

The random forest algorithm was experimented with as it is widely used to solve many machine learning problems. This algorithm produced worse results than kNN. The algorithm is also less time efficient than KNN especially when the number of samples (i.e., trees) is increased. For number of samples `n=1`, even for 60 points it takes 70 seconds, but as n increases to 14, only for 3 previous points it took 70 seconds. The worse is at `n = 14`, which took about 7 seconds to execute. Looking at the data we concluded that n = 3-4 and previous points = 4-6 should give us the least error. (This is indicated by the dark region at the lower left corner of the heat map.)

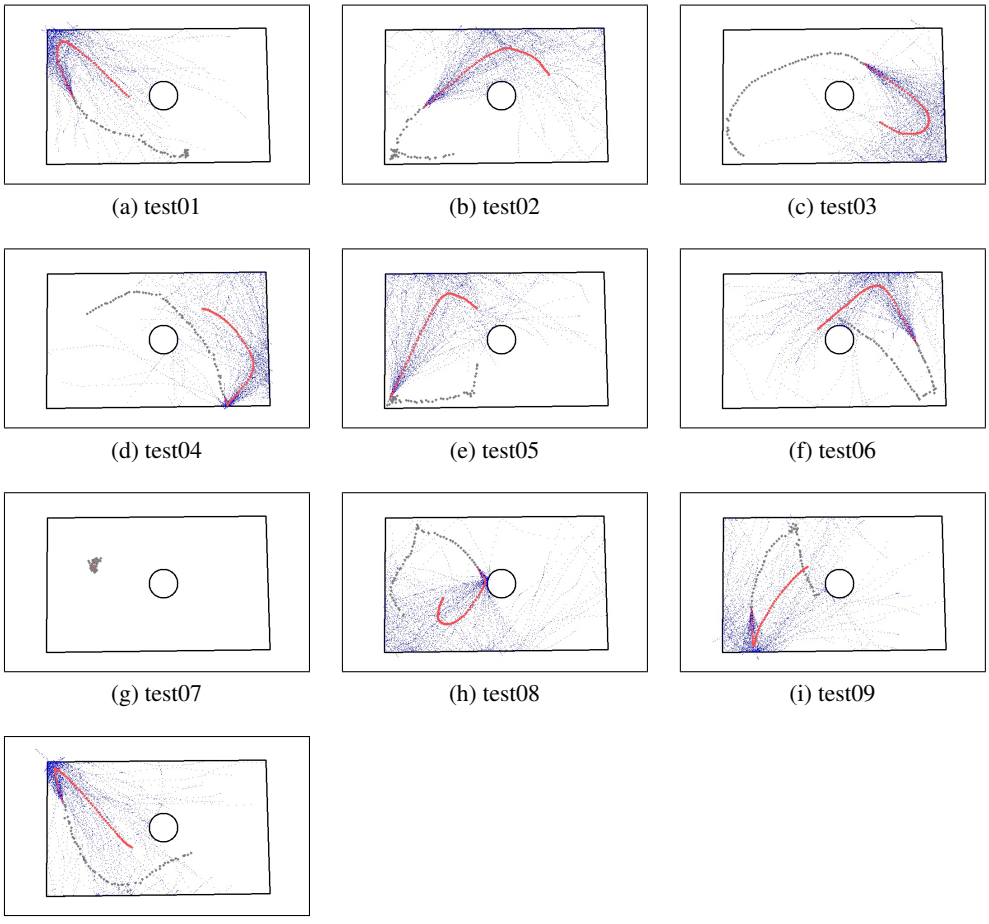
The average error of the each file confirmed the challenges that we faced in kNN. Times for roll-over cases, sudden direction change and slide along the boundary made the performance of those files consistently worse.

Give more time, we would have:

- Added different types of features to tackle roll-over cases, sudden direction change, etc.
- Explored machine learning approaches like neural networks
- Trained the algorithms using more data
- Tried different parameters of algorithms

4 Results

Particle filters proved to be the most effective. The trajectory completion diagrams are given in Fig. 8 below. The diagrams depict how the prediction (in red) would complete the last set of measurements (in black). The predictions seem to be logical and natural in most cases. Note, for example, how the predictions correctly model the bounce off the box boundary.



(a) test01 (b) test02 (c) test03

(d) test04 (e) test05 (f) test06

(g) test07 (h) test08 (i) test09

(j) test10

Figure 8: Particle filter trajectory completion

Table 2 lists the error for each approach for the robot's motion, from step 1741 to 1800. Clearly, particle filters scored the best, although kNN also proved to be effective.

Table 2: Algorithms accuracy comparison

Approach					
Algorithm	Optimal Parameter	Best Score (LMS)	Mean	σ	Avg. Time (s)
Particle filter	5000 particles	537.37	560.94	337.14	20
kNN	7 neighbors, 23 prior points	518	791.3[1]	110.5[1]	8-17
Random Forests	7 samples, 7 previous points	500	848.7[1]	152.6[1]	180-420
kNN with EKF	5 neighbors, 4 features	747	749.13		120
Kalman filtering		950	966.82	303.13	60

[1] Results across the full parameter spectrum experimented with (e.g., k=3 to 25 for kNN).

Table 3: Accuracy comparison for each test file

File	Particles	Kalman	kNN	EKF-kNN
test01	1011.10	1394.12	1371	311.22
test02	487.15	706.50	604	1297.06
test03	680.31	772.61	1013	816.38
test04	513.86	776.80	902	1055.32
test05	1177.03	1357.82	855	1012.16
test06	808.06	1430.99	292	649.35
test07	140.23	793.76	992	944.26
test08	130.16	770.32	602	278.87
test09	368.83	1162.68	1603	780.48
test10	292.62	619.25	493	399.90

Table 3 lists relative errors pertaining to each test file.

5 Discussion

The particle filter approach appears to give the best results. It was not the best on every test case, but it was the best in a majority of cases, and, more importantly, it was never the worst approach. The particle approach appears to have two advantages over other algorithms. First, although each particle is dumb, the combination of different particle paths reveals effects that are impossible to see when considering just a single, most likely path. For example, consider the case where the Hexbug is traveling in a straight line parallel to a wall to its left. Its most likely path may be to travel straight, with lower likelihood of turning to the left or right. However, if it turns to the left it will hit the wall, bounce off, and end up heading to the right. So even though the most likely single path is straight, the most likely position is to the right. Second, the further into the future we try to predict, the more uncertain the actual position will be. The more uncertain the position, the more we should favor a guess near the center of the box. Not because the center of the box is more likely than any other position, but because the center of the box minimizes the worst-case error. (If you wanted to minimize the L2 error of your position estimate without ever measuring the Hexbug's position, you should always guess the center of the box.) Because of the way the particles spread out over time, the average naturally tends to drift toward the center of the box. These two effects appear to have been enough for particle filters to outperform the other approaches we considered.

6 Improvements

Given more time, we would have explored some ideas further:

- **Switching Kalman filter** Using a switching Kalman filter, where multiple Kalman filters run in parallel, each using a different model of the system (collision, straight motion, flipping over), and then using a weighted sum of predictions [1].
- **UKF** Using an Unscented Kalman filter (UKF) to gauge performance against EKF.
- **More training** Resolving some limitations in our machine learning approach by adding more features or increasing the training data. The challenges included Hexbug rollover, sliding against the boundary and sudden direction change. (The results for all other cases were consistent, though. Only four test files that face these challenges exhibited high errors.)
- **Particle filter with an advanced motion model** The implemented particle filter uses very simple models for linear motion, collisions and noise. It would be interesting to make each particle move based on an EKF or kNN. (This seems like a promising approach, although the run time constraint would likely prove challenging.)

7 Conclusion

Each member of the team explored different algorithms for predicting the robot's motion, and the most optimal approach was chosen in the end. The best results were achieved with the particle filter approach, as particle filters capture the possibilities inherent in a robot's future motion quite well. The machine learning algorithms also produced competitive scores, but particle filters gave more consistent results, ran more efficiently, and did not require training data.

Our experience with this project suggests that blending different but complimentary approaches would generate an even more effective predictor. We hope to explore this opportunity in future endeavors.

References

[1] Stuart J. Russell and Peter Norvig. 2003. Artificial Intelligence: A Modern Approach (2 ed.). Pearson Education.

[2] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. 2005. Probabilistic Robotics (Intelligent Robotics and Autonomous Agents). The MIT Press.

[3] http://studentdavestutorials.weebly.com/object-tracking-2d-kalman-filter.html, Retrieved July 30, 2016.

[4] Kalman filter for vision tracking, Kalman filter for vision tracking, Retrieved July 21, 2016.

8 Appendix

8.1 Program

This section details the artifacts in the codebase and execution instructions.

8.1.1 Usage

The program can simply be run as:

```
python grading.py finalproject
```

It must be noted that the inputs folder contains custom centroid data.

8.1.2 Optional Requirements

The software requirements pertain to the approaches that were not used in the final program. Since these scripts are optional and provided only as a reference point, installation instructions for these libraries are not given. The optional scripts may require installation of:

- scikit 0.17 (optional)
- filterpy (optional)
- OpenCV (optional)

8.1.3 Contents

The folder `extra-solutions` contains additional programs not used in our final implementation.

The submission contents are as follows:

Table 4: Contents

File		
Name	Path	Description
finalproject.py	.	Implementation (particle filters)
hexbuy.py	extra-solutions	Machine learning approaches
machineLearning csv files	extra-solutions	Machine learning analysis
customimageprocessing	extra-solutions	Image pre-processing
kalman.py	extra-solutions	A Kalman implementation
datamaker.py	extra-solutions	Random data slicer
matrix.py	extra-solutions	Utility program
inputs	.	Custom input files