Positional Tracking Performance of the Extended Kalman Filter and the Particle Filter algorithms.

Mark McKelvy

CMPS 523: Computational Basis of Intelligence

Abstract – In this paper I implement a simulation testbed and agent for testing two Bayes filter algorithms, the Extended Kalman Filter (EKF) and the Particle Filter (PF). After implementing supplementary algorithms needed for motion noise, landmark detection and the like, the two algorithms are put through similar testing scenarios and the results are reported and compared. The results obtained for the test cases involving the EKF proved to be satisfactory, while the PF seemed to come up short. As it is commonly accepted that the PF is a more robust algorithm than the EKF, the author assumes that implementation errors have led to incorrect and disappointing results.

1. Introduction

The purpose of this paper is to explore two specialized Bayes filter algorithms by implementing an agent/testbed simulator that uses them. Specifically of interest for implementation are the ***EKF_localization_known_correspondences*** (an Extended Kalman Filter) and the ***MCL*** (Monte Carlo Localization, a Particle Filter) Bayes filter algorithms listed in [1].

A Bayes filter algorithm is a general algorithm used for calculating beliefs. It does this by using measurement and control data to estimate the state of a system. Bayes filter algorithms are applicable to many robotic systems, and particularly robot localization.

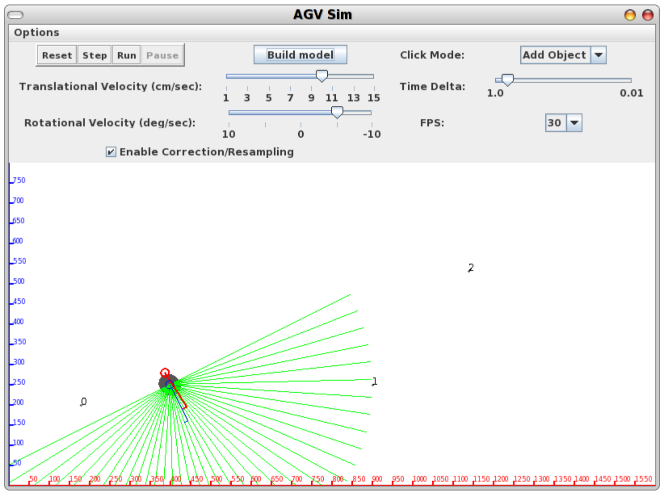
The way a Bayes filter algorithm works is by using recursion, calculates the belief of the state of the robot at time *t* based on time *t-1*. To do this, it calculates a new belief based on the previous belief, the most recent control action, and the most recent measurement of the world. The algorithm is broken into two steps – a prediction step and a correction (or measurement update) step. The basic Bayes algorithm requires integration in closed form, which can only be done for very simple problems. The algorithm also requires the initial belief $p(x_0)$, the measurement probability $p(z_t \mid x_t)$, and the state transition probability $p(x_t \mid u_t, x_{t-1})$. Lastly, a Bayes filter algorithm makes a *complete state assumption*, also known as the *Markov assumption*, which is the assumption that knowledge of any other states whether past or present, give us no more knowledge about the current state of the system than the immediately previous state in time. This assumption allows simplifications in algorithm implementation at the cost of reduced accuracy.

In this paper, two tractable implementations of the Bayes filter algorithm are used. The first, which will be described immediately following, is known as the Extended Kalman Filter algorithm, while the second, which will be discussed shortly, is known as the Particle Filter algorithm.

The Extended Kalman Filter (EKF) algorithm is a recursive state estimator in the family of Gaussian filters, and is a tractable implementation of the Bayes filter for continuous spaces. The EKF represents the belief state as a multivariate normal distribution with a mean μ and a covariance Σ, where Σ is symmetric and positive-semidefinite. It is important to note here that since the belief state is represented as a multivariate normal distribution (a Gaussian), it only solves the position tracking problem since the belief is a unimodal distribution – the robot is believed to only be in one possible location. The EKF differs from the Kalman Filter in that it relaxes an assumption that state transition probabilities (and measurement probabilities) are linear. An example where state transition probabilities are non-linear would be a robot traveling along a circle. The EKF relaxes this assumption by approximating the non-linear state transitions (and measurement probabilities) using Taylor expansion, which produces linear approximations to the non-linear probabilities. What this means is that the non-linear function is approximated by a linear function at a specific point by taking the first derivative of the non-linear function. The linearization makes the algorithm more efficient at the expense of reduced accuracy as we move away from the point the derivative was taken at. Since state transition probabilities and measurement probabilities are functions of more than one variable, we must take partial derivatives with respect to each variable and evaluate at a specific point, producing a matrix of derivatives known as the *Jacobian*. The Jacobian is then uses to linearize the motion and measurement probabilities so that the algorithm may proceed.

The Particle Filter (PF) algorithm is also a recursive state estimator and a tractable implementation of the Bayes filter, but is nonparametric. This means that it approximates the set of beliefs by a finite number of parameters, also known as *particles*. Unlike the EKF which represents beliefs by the parametric Gaussian distribution, the PF's nonparametric estimations are approximate but more flexible in that they can model nonlinear transformations of random variables without the need for linear approximation (which introduces error). In the PF, each *particle* represents a hypothesis as to the state of the system at time *t*. For most PF implementations, a large number of particles (denoted *M*) is used – commonly at least 1,000. Most PF implementations work essentially by following three steps. First, each particle is updated by a motion model with added noise. Second, each particle is assigned a weight (*importance factor*) through a process of taking a measurement and calculating the likelihood of a particle detecting a measurement compared to what actually happened (*measurement update*). This typically has the effect that particles closer to the actual location of the robot have higher weights, while outliers have lower weights. The third step is a process known as *resampling*, which creates a new particle set of the same size (*M*) that is a subset of the previous set and has particles picked according to the their respective weights. This has the effect that particles with higher weights have a higher likelihood of being selected, though the process is random and it may not occur that way. The weights are stripped from the particle set and the process is repeated.

The simulation software designed for this project was built using the Java programming language where at the core was roughly a Model-View-Controller framework. The agent (believed robot state) and the testbed (actual robot state) had very minimal interaction, to the effect that essentially the agent gave the testbed a control action to execute and the testbed reported back a measurement if there was one. Using the simulation software, we should be able to determine which algorithm (if any) is better than the other, and in which circumstances, by collecting and analyzing the data produced by the different scenarios.



Click Mode: Add Object

FPS: 30

[✓] Enable Correction/Resampling

Figure 1: The ***AGVsim*** *main GUI interface showing the robot's believed (blue) and actual (red) pose, the laser range finder scans (green) and objects (black dots). The main interface allows, among other things, changing of the translational and rotational velocity as well as the time step.*

2. Methods

This section is subdivided into two subsections, the first of which identifies test cases goal is to, on a point-by-point basis, identify the various aspects of each algorithm and compare differences (or similarities) of each. Though some points may not be comparable, they may be included as lagniappe.

2.1 Default Parameters and Values

In most scenarios, unless stated otherwise, default values of parameters will be used. The parameters, values, and descriptions are listed in the following paragraphs.

2.1.1 Control Actions

The default values for control actions, commonly denoted $u_t = (\; v\,\omega \;)^T$ are as follows: translational velocity $v=10\,\frac{cm}{sec}$; rotational velocity $\omega=-5\,\frac{deg}{sec}$.

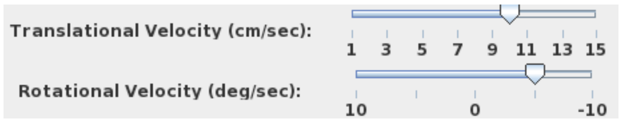

Translational Velocity (cm/sec):
1 3 5 7 9 11 13 15
Rotational Velocity (deg/sec):
10 0 -10

Figure 2: Velocity controls.

2.1.2 Initial Pose

Both the believed and actual bot pose, commonly denoted $x_t = (\; x\,y\,\theta \;)^T$ are as follows, on a grid 1600cm x 800cm: $x=180\text{cm}$; $y=200\text{cm}$; $\theta=\frac{\pi}{2}$.

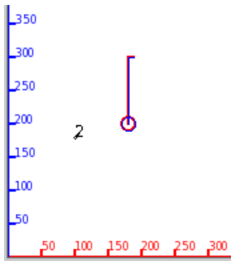


Figure 3: Initial pose of the actual (red) and believed (blue) robot.

2.1.3 Initial Objects

Initially there will 3 objects (*landmarks*) within the simulation testbed, commonly denoted $m_i = (x\;y\;s)$ where *s* is the signature, as follows: $m_0=\begin{pmatrix}600 & 700 & 0\end{pmatrix}$; $m_1=\begin{pmatrix}700 & 200 & 1\end{pmatrix}$; $m_2=\begin{pmatrix}100 & 180 & 2\end{pmatrix}$.

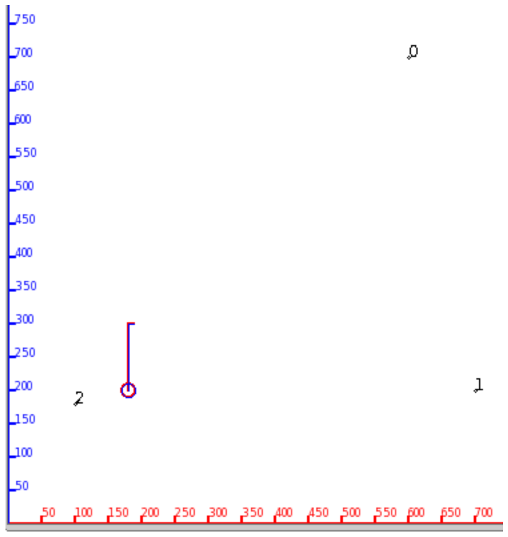


Figure 4: Locations of the default placed objects.

2.1.4 Testbed Sensor

There is a single laser range finder sensor with a field of view of 180 degrees (number of sensors and field of view is not configurable). The angular resolution will be set at 5 degrees. The maximum range will be set at 500cm. The beam display will be on. The beam model values for testbed sensor noise $(z_{hit}, z_{short}, z_{max}, z_{rand}, \sigma_{hit}, \lambda_{short})$ will all be "disabled" - set to 0 to have no effect.

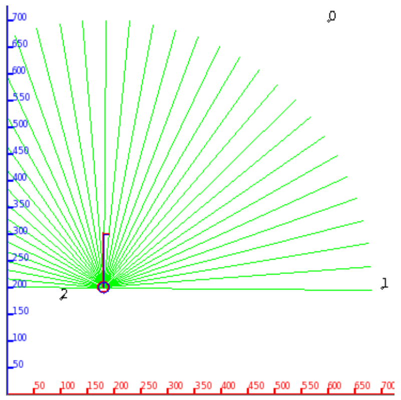
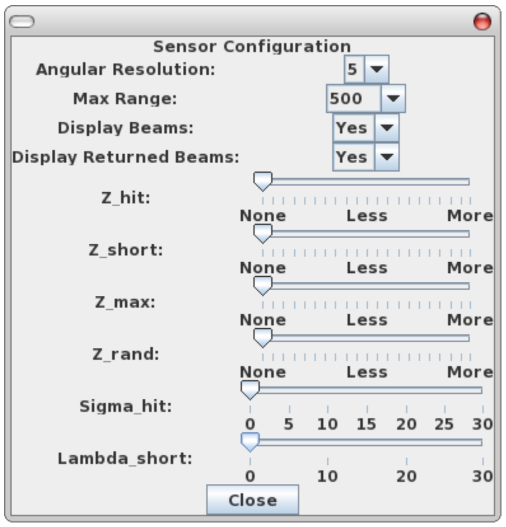


Figure 5: Visualization of a sensor scan.



Sensor Configuration

Angular Resolution: 5

Max Range: 500

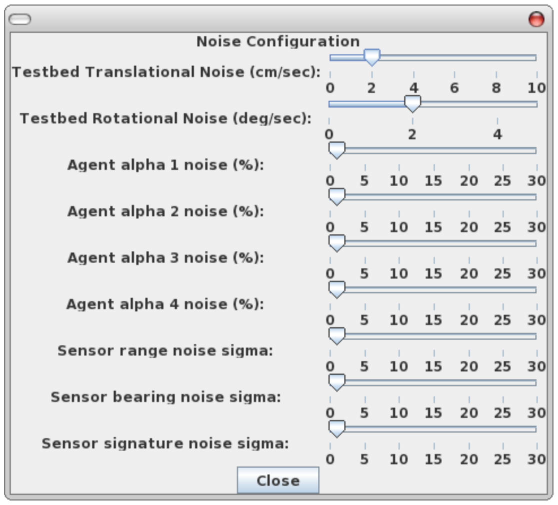
Display Beams: Yes

Display Returned Beams: Yes

Figure 6: The sensor configuration dialog.

2.1.5 Testbed Motion Noise, Agent Motion Noise, and Agent Sensor Noise

The testbed translational noise σ_v and the rotational noise σ_ω will both be normally distributed random variables with an initial standard deviations of $\sigma_v = 2\frac{cm}{sec}; \sigma_\omega = 2\frac{deg}{sec}$. The motion noise, commonly denoted $(\alpha_1, \ldots, \alpha_4)$ in the case of EKF, or $(\alpha_1, \ldots, \alpha_6)$ in the case of PF, will all have values corresponding to 1%. The agent sensor noise standard deviations $(\sigma_r \quad \sigma_\phi \quad \sigma_s)^T$ will be set to $(1\,1\,1)^T$ respectively.

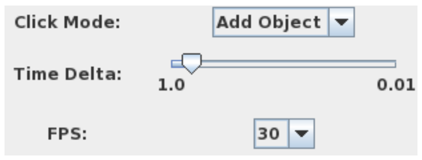


Noise Configuration

Figure 7: The noise configuration dialog.

2.1.6 Miscellaneous

The time step for the simulation will be set at $\Delta t = 0.1$. The simulation frame rate will be set at 30 frames per second. The default click mode will be to add objects to the testbed. The default number of particles (for PF), will be chosen to be 1000.


Click Mode: Add Object

Time Delta: 1.0 … 0.01

FPS: 30

Figure 8: Miscellaneous controls.

2.2 EKF Implementation & Testing

The particular flavor of the EKF implemented for this project is the ***EKF_localization_known_correspondences*** found in Chapter 7 of [1]. This EKF is used for feature-based maps where there are point-based landmarks of (theoretically) zero size (though in the implementation I've used a size of 1). A motion model for the agent is used that incorporates translational and rotational noise estimations. A measurement model (for detecting landmark objects) is used that is based on the Pythagorean theorem with added measurement noise and can identify landmarks and assign signatures to those landmarks via a correspondence variable. In this algorithm, Jacobians are first computed which are needed for the linearized motion model. Next the motion update step is performed using the method mentioned above, which will calculate a new pose belief and uncertainty covariance matrix. The correction step follows next for the case that there are objects that were detected in the testbed. In short, for each object detected, an *innovation* vector is calculated and updated to the believed pose after factoring in Kalman bias based on sensor uncertainty as well as pose uncertainty. Lastly, the new belief pose and covariance matrix are saved for the next iteration of the algorithm, at which point a control action and sense command are given to the testbed.

2.2.1 Positional tracking with default values

In this test case, the simulation will be run for approximately 220 seconds. The robot's default pose will be used, as well as all of the default values for noise , objects, and sensor configuration. The EKF will be used and the positional error as well as orientation error will be recorded.

2.2.2 Positional tracking with no objects

In this test case, the simulation will be run for approximately 240 seconds, the amount of time for the robot to complete two loops with the rotational velocity modified as $\omega=-3\frac{deg}{sec}$. Additionally, all landmark objects have been removed from the simulation testbed. All of parameter values and configurations will remain the same. The EKF will be used and the purpose of this test is to see the prediction step in action without the correction step.

2.2.3 Positional tracking, no objects and slow rotational velocity

In this test case, the simulation will be run for approximately 100 seconds with the rotational velocity modified as $\omega=1\frac{deg}{sec}$ and the initial orientation of the robot's believed and actual poses set to $\theta=0$. Additionally, all landmark objects have been removed from the simulation testbed. All of parameter values and configurations will remain the same. The EKF will be used and the purpose of this test is to see the prediction step in action without the correction step.

2.2.4 Positional tracking, no objects and slow rotational velocity, large Δt

This test is the same as 2.2.3, with the exception that $\Delta t=1.0$ instead of $\Delta t=0.1$.

2.2.5 Positional tracking, two objects and slow rotational velocity

In this test case, two objects are placed at $m_0=(800 \quad 200 \quad 0)$ and $m_1=(1200 \quad 750 \quad 1)$. Additionally, the translational speed has been modified as $v=15\frac{cm}{sec}$ and the rotational velocity has been modified as $\omega=1\frac{deg}{sec}$. The initial orientation of the robot's believed and actual poses are set to $\theta=0$.

2.2.6 Positional tracking with sensor noise

In this test case we will model sensor noise in the agent and create sensor noise in the testbed with $(\sigma_r \quad \sigma_\phi \quad \sigma_s)^T$ set to $(30 \quad 10 \quad 10)^T$ and $(z_{hit}, z_{short}, z_{max}, z_{rand}, \sigma_{hit}, \lambda_{short})=(0.5793, 0.5793, 0.0, 0.5793, 30.0, 10.0)$. The robot will make two loops and there is no testbed control noise. All other parameters will remain the same.

2.2.7 Positional tracking with control noise

In this test case we will model control noise in the agent and create control noise in the testbed. All of parameters will be at their default values. The testbed noise will be $\sigma_v = 4\frac{cm}{sec}; \sigma_\omega = 2\frac{deg}{sec}$. The agent noise parameters will be $(\alpha_1 = .30 \quad \alpha_2 = .10 \quad \alpha_3 = .10 \quad \alpha_4 = .20)$. The robot will perform two loops.

2.3 PF Implementation & Testing

The type of PF implemented for this project is the ***MCL*** (Monte Carlo Localization) which is found in Chapter 8 of [1]. The easy-to-implement PF-based algorithm represents belief by particles, which by default for this project are distributed by a very small standard deviation of the initial actual robot pose. To accomplish a scenario involving global localization, the source code must be slightly modified, and since the EKF does not handle global localization, it does not seem applicable to the goal of this project (or perhaps it doesn't work correctly in my implementation – you decide). In this algorithm, the new particle set is set to uninitialized, but the same number of particles (*M*) as the current particle set. The algorithm then iterates through the particle set to do the following things. First, a sample motion model is applied to each particle which is based on the current control action and the previous pose of the particle, incorporating translational and rotational noise estimations. Next, a measurement model (for detecting landmark objects) is used that is based on the Pythagorean theorem with added measurement noise and can identify landmarks and assign signatures to those landmarks via a correspondence variable. The way the measurement model is used is to calculate a probability of detecting an object (for a given particle) and that value is used as a weight (*importance factor*) for the particle. Lastly in this iteration through particles, the current particle is added (along with the weight) to the new particle set. Next, a new iteration loop (*resampling* step) through the particles samples with replacement the particles according to their weights (a low variance sampler is used in the implementation). At this point, the weights are discarded and the current iteration of the MCL is complete.

2.3.1 Positional tracking with default values

In this test case, the simulation will be run for approximately 220 seconds. The robot's default pose will be used, as well as all of the default values for noise , objects, and sensor configuration. The PF will be used with 1000 particles and the positional error as well as orientation error will be recorded.

2.3.2 Positional tracking with no objects

In this test case, the simulation will be run for approximately 240 seconds, the amount of time for the robot to complete two loops with the rotational velocity modified as $\omega = -3\frac{deg}{sec}$. Additionally, all landmark objects have been removed from the simulation testbed. All of parameter values and configurations will remain the same. The PF will be used and the purpose of this test is to see the prediction step in action without the correction step. There are 1000 particles being used.

2.3.3 Positional tracking, no objects and slow rotational velocity

In this test case, the simulation will be run for approximately 100 seconds with the rotational velocity modified as $\omega = 1\frac{deg}{sec}$ and the initial orientation of the robot's believed and actual poses set to $\theta = 0$. Additionally, all landmark objects have been removed from the simulation testbed. All of parameter values and configurations will remain the same. The PF will be used and the purpose of this test is to see the prediction step in action without the resampling step.

2.3.4 Positional tracking, no objects and slow rotational velocity, large Δt

This test is the same as 2.3.3, with the exception that $\Delta t = 1.0$ instead of $\Delta t = 0.1$.

2.3.5 Positional tracking, two objects and slow rotational velocity

In this test case, two objects are placed at $m_0 = (800 \quad 200 \quad 0)$ and $m_1 = (1200 \quad 750 \quad 1)$. Additionally, the translational speed has been modified as $v = 15\frac{cm}{sec}$ and the rotational velocity has been modified as $\omega = 1\frac{deg}{sec}$. The initial orientation of the robot's believed and actual poses are set to $\theta = 0$.

2.3.6 Positional tracking with sensor noise

In this test case we will model sensor noise in the agent and create sensor noise in the testbed with $(\sigma_r \quad \sigma_\phi \quad \sigma_s)^T$ set to $(30 \quad 10 \quad 10)^T$ and $(z_{hit}, z_{short}, z_{max}, z_{rand}, \sigma_{hit}, \lambda_{short}) = (0.5793, 0.5793, 0.0, 0.5793, 30.0, 10.0)$. The robot will make two loops and there is no testbed control noise. All other parameters will remain the same.

2.3.7 Positional tracking with control noise

In this test case we will model control noise in the agent and create control noise in the testbed. All of the other parameters will be at their default values. The testbed noise will be $\sigma_v = 4\frac{cm}{sec}; \sigma_\omega = 2\frac{deg}{sec}$. The agent noise parameters will be $(\alpha_1 = .30 \quad \alpha_2 = .10 \quad \alpha_3 = .10 \quad \alpha_4 = .20 \quad \alpha_5 = .10 \quad \alpha_6 = .10)$. The robot will perform two loops.

2.3.8 Positional tracking with control noise

This test case is the same as 2.3.7, with the following change in the agent noise parameters: $(\alpha_1 = .30 \quad \alpha_2 = .10 \quad \alpha_3 = .10 \quad \alpha_4 = .20 \quad \alpha_5 = 0.0 \quad \alpha_6 = 0.0)$. The robot will perform two loops.

3. Results

In the following two subsections, the individual results of the scenarios in the previous section are presented, while in the next section comparisons and conclusions are made.

3.1 EKF Results and Analysis

The following subsections present the data collected for the tests in subsection 2.2.

3.1.1 Positional tracking, default values

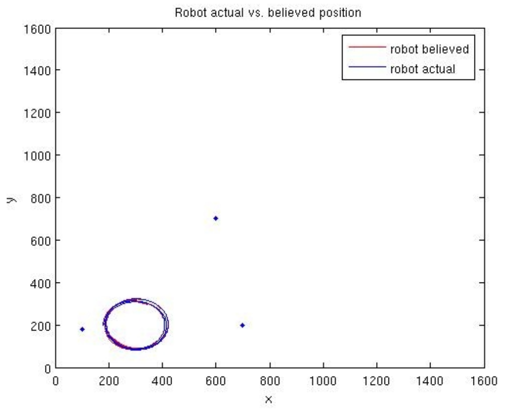


Figure 9: The robot's actual vs. believed path.

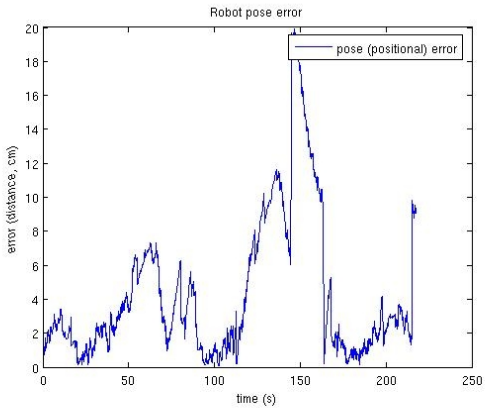


Figure 10: The error in positional tracking.

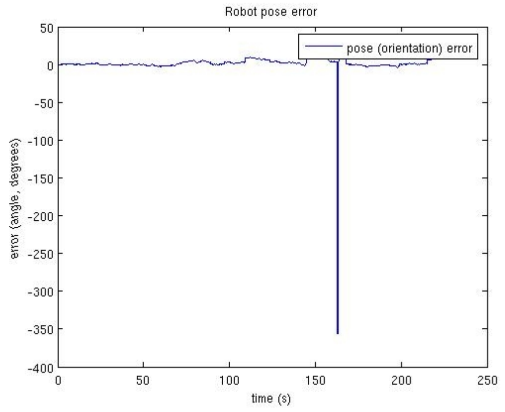


Figure 11: The error is orientational tracking (degrees).

The robot ran for approximately 220 simulated seconds. During that time, the EKF algorithm was used to track the pose of the robot with added noise. As can be seen from *Figure 9*, the tracked position was quite accurate, where the positional error (*Figure 10*) was not greater than 20cm. The orientational error was also quite low, except for one spike.

3.1.2 Positional tracking with no objects

As can be seen in *Figure 12*, the pose prediction capability follows quite closely to that of the actual path. Due to noise in the control actions, the positional error fluctuates over time (*Figure 14*), gradually getting worse, but occasionally getting better. In *Figure 13*, the uncertainty can be seen as an ellipse surrounding the mean pose.

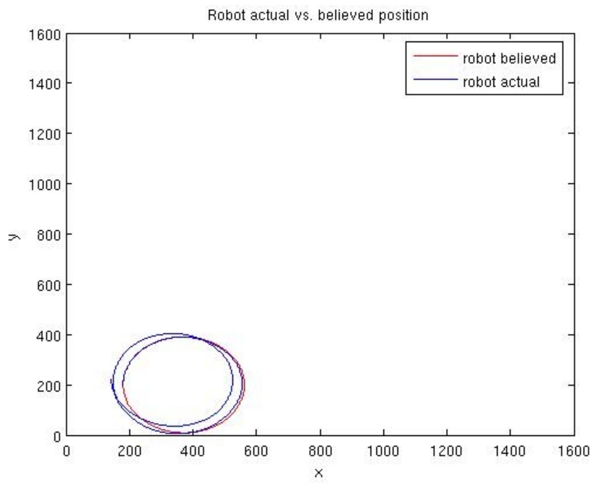


Figure 12: Paths of believed vs. actual robot.

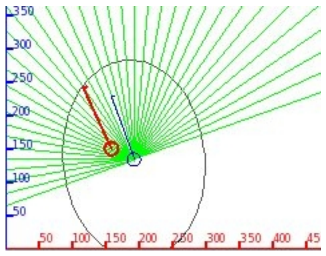


Figure 13: Visualization showing uncertainty in position.

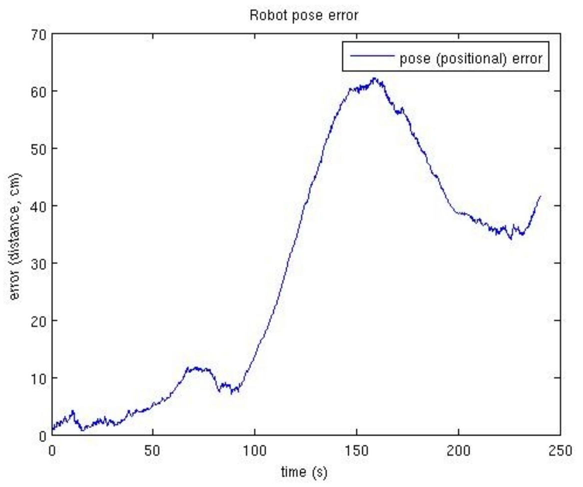


Figure 14: Graph of the positional error over time.

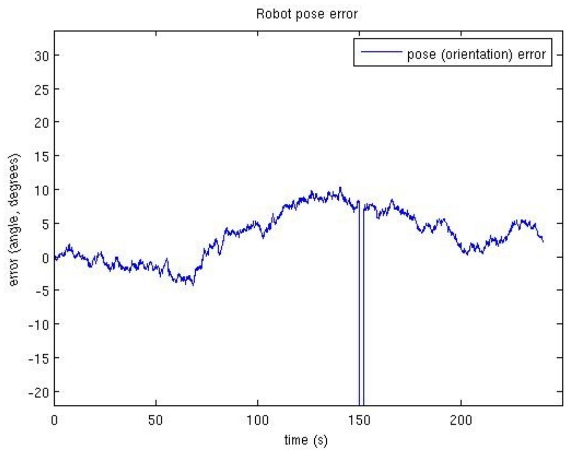


Figure 15: Graph of the orientational error over time.

3.1.3 Positional tracking, no objects and slow rotational velocity

Though similar to the previous case, this case shows that without correction the positional accuracy over time gradually degrades (*Figure 16*). The orientational accuracy stays surprisingly high even without correction from landmark objects, likely do to good noise estimation (*Figure 17*).

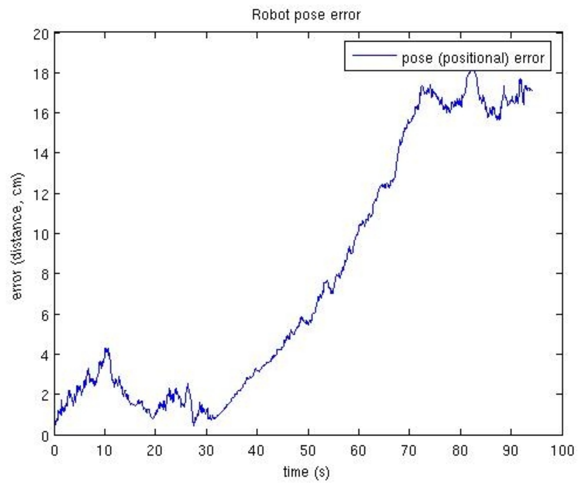


Figure 16: Graph of the positional error over time, getting gradually worse due to no correction.

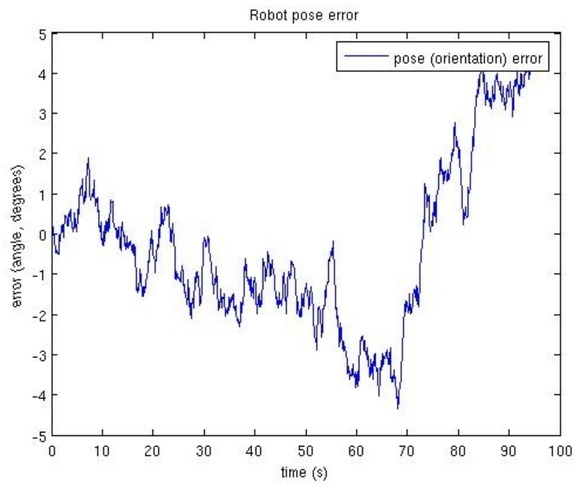


Figure 17: Graph of the rotational error over time, which stays quite low, within +/- 5 degrees, even without correction.

3.1.4 Positional tracking, no objects and slow rotational velocity, large Δt

The results of this test case serve to highlight the fact that with a smaller time step (Δt), greater accuracy can be achieved. In this example, Δt was changed from the default value of *0.1* to a value of *1.0*. In *Figure 18*, after just a short amount of time (about 100 seconds) the paths of the believed vs. actual location start to differ significantly. This can be verified by *Figure 19* as there is an almost linear increase in the positional error. Orientational error (*Figure 20*) doesn't suffer as much as positional error in this particular scenario.

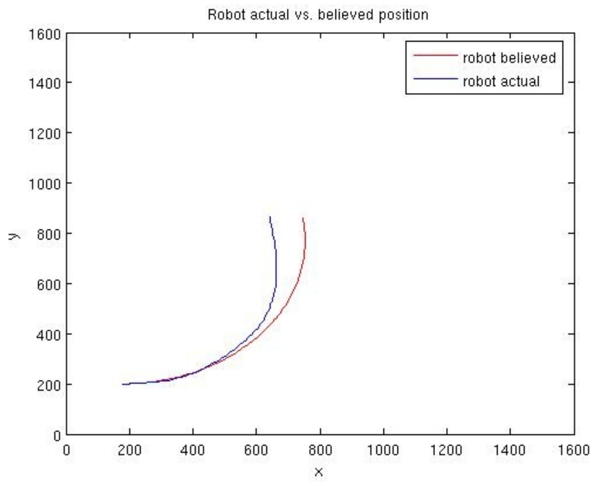


Figure 18: Paths of believed vs. actual robot with $\Delta t = 1.0$.

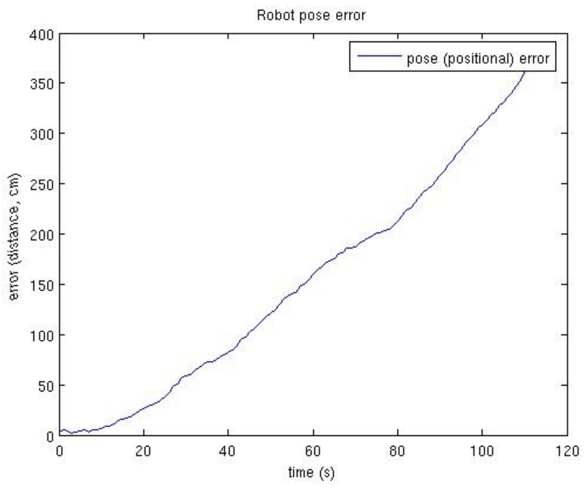


Figure 19: A large time step in this case creates an almost linear increase in positional error.

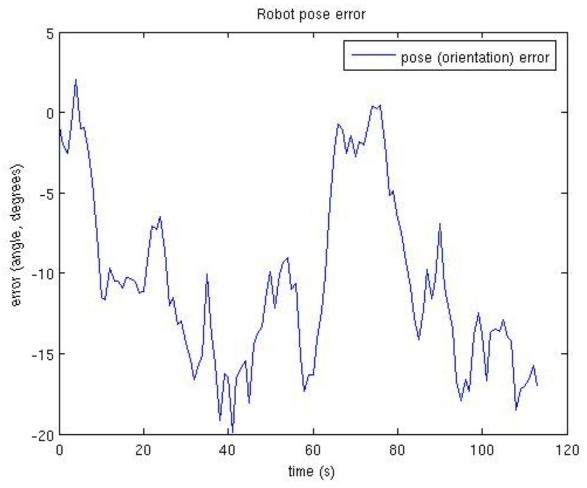


Figure 20: The large time step does have as great of an effect on the orientational error, at least for this small amount of time and this scenario.

3.1.5 Positional tracking, two objects and slow rotational velocity

This is where we can see the power of the EKF to integrate corrections in each iteration. Initially the positional error increases over time as in the previous examples, but once the robot starts sensing landmarks we notice the effects of the correction step. After a short time, we see that the corrections start to bring the positional error down and back into line with the actual position of the robot (*Figure 21*). A similar effect is seen with the orientational error (*Figure 22*).

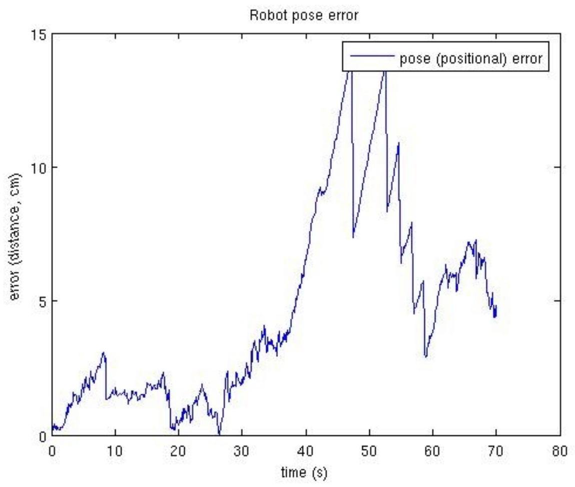


Figure 21: Positional accuracy of the robot increases normally, and the starts to decrease after several iterations of corrections integrated into the estimation.

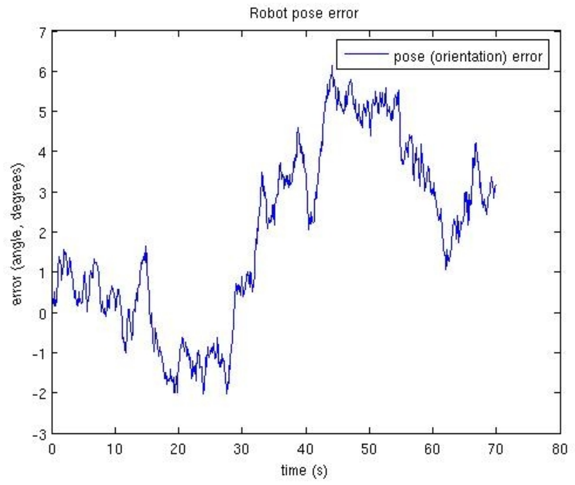


Figure 22: Orientational accuracy seems to oscillate as the EKF attempts to bring the believed pose back into line with the actual pose after measurement updates.

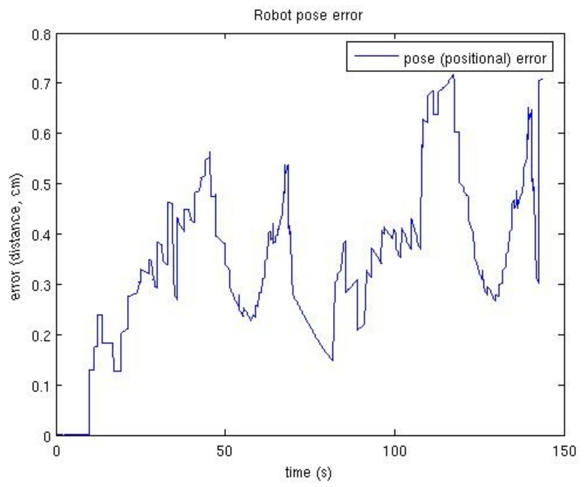


Figure 23: The EKF performs exceptionally well for positional error once error estimation parameters are taken into account for sensor noise.

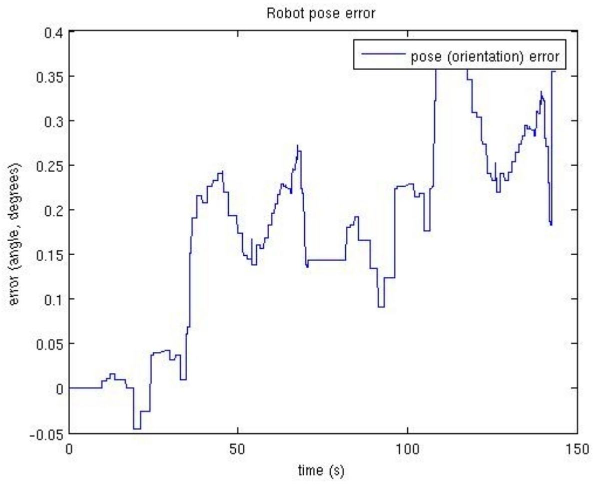


Figure 24: Again, the EKF works exceptionally well even for orientational error by compensating for sensor noise.

3.1.6 Positional tracking with sensor noise

The results of this test are particularly interesting. In this scenario, the default values were used for all parameters except for the sensor noise. As a reminder, the scenario modeled sensor noise in the agent and created sensor noise in the testbed with $\begin{pmatrix}\sigma_r & \sigma_\phi & \sigma_s\end{pmatrix}^T$ set to $\begin{pmatrix}30 & 10 & 10\end{pmatrix}^T$ and $(z_{hit}, z_{short}, z_{max}, z_{rand}, \sigma_{hit}, \lambda_{short}) = (0.5793, 0.5793, 0.0, 0.5793, 30.0, 10.0)$. For the test time of 150 seconds, the positional error never goes above 1cm (*Figure 23*), and likewise the orientational error stays below half a degree (*Figure 24*).

3.1.7 Positional tracking with control noise

In this test case the control noise is modeled in the agent and control noise is created in the testbed. All of parameters are at their default values, save for testbed noise which is $\sigma_v = 4\frac{cm}{sec}; \sigma_\omega = 2\frac{deg}{sec}$ and the agent motion noise parameters $\begin{pmatrix}\alpha_1 = .30 & \alpha_2 = .10 & \alpha_3 = .10 & \alpha_4 = .20\end{pmatrix}$. The robot performed two loops, for a total time of about 150 seconds.

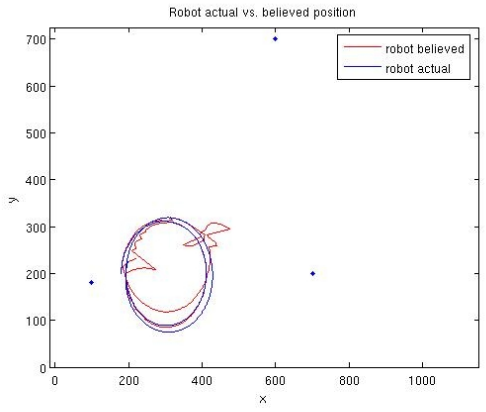


Figure 25: The believed position of the robot experiences a few hiccups as the actual robot loops around in circles. This could be due to too high of noise alphas.

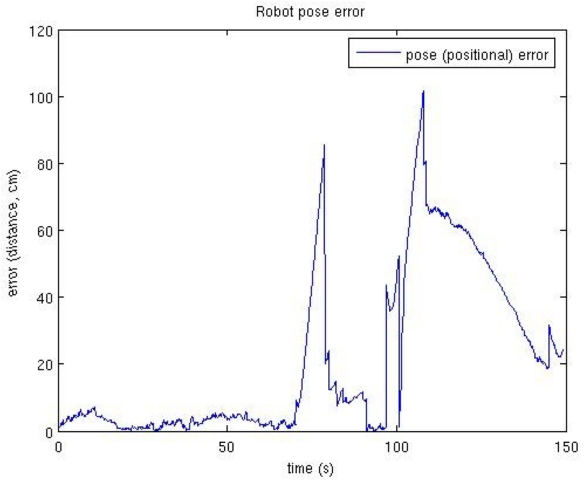


Figure 26: The positional error has some interestingly high spikes before correcting quite well. It would be nice if the spikes weren't so severe though, and it is likely a result of the noise parameters chosen.

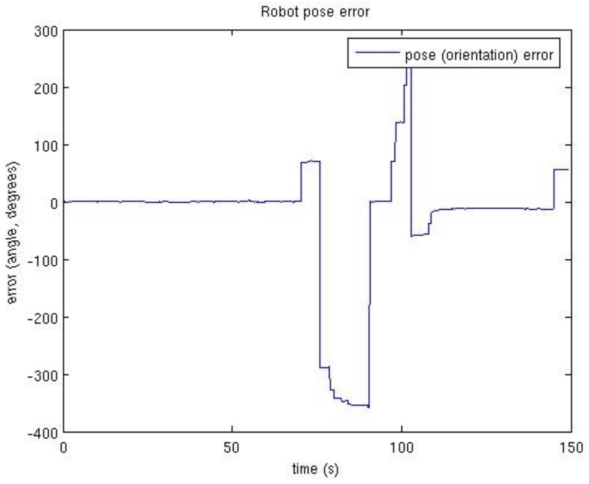


Figure 27: The orientational error experiences similar spikes as the positional error does in the previous figure. Why?

It is strange that the EKF positional tracking experiences such huge jumps in position, as seen in *Figure 25*. This is likely due to two factors: the detection/correction of landmark objects and the noise compensation values chosen (the motion noise alphas). The EKF does a reasonably good job at tracking back after the huge spikes, however.

3.2 PF Results and Analysis

The following subsections present the data collected for the tests in subsection 2.3.

3.2.1 Positional tracking, default values

The results of this test case are significant compared to the similar test with the EKF. As seen in *Figure 28* and *Figure 29*, the positional accuracy was quite worse than in *Figure 9* and *Figure 10*, respectively. The orientational error is roughly the same with similar spikes (*Figure 30*).

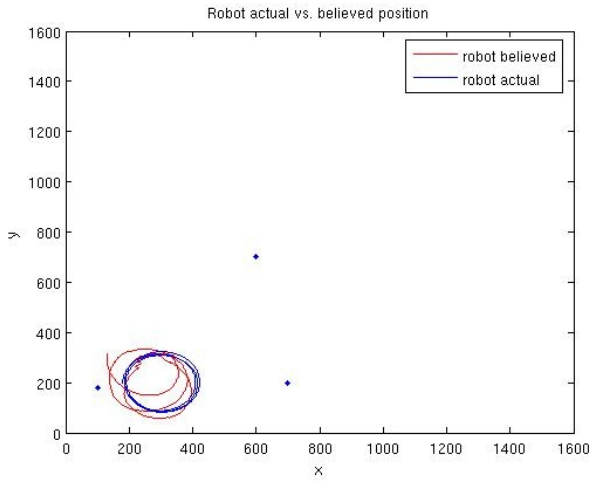


Figure 28: The robot's believed vs. actual path is less accurate than in the similar scenario where the EKF is used.

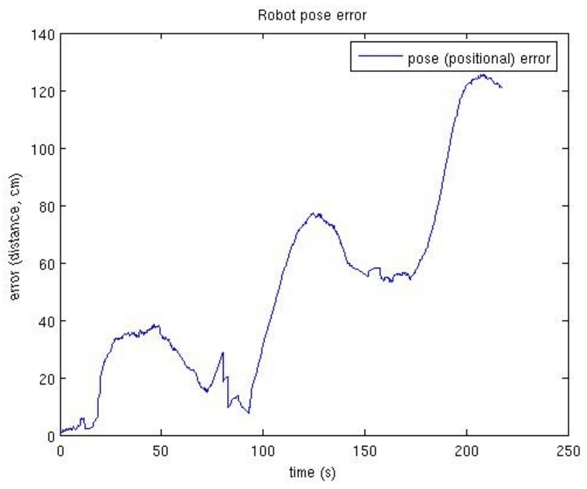


Figure 29: The positional accuracy degrades quicker in this scenario than in the similar scenario where the EKF is used.

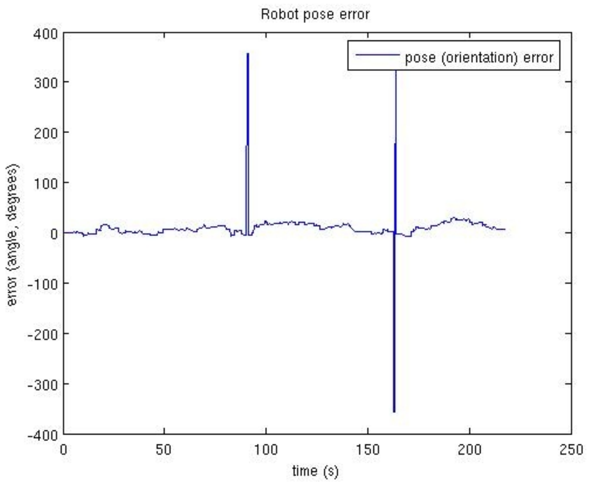


Figure 30: The orientational accuracy is comparable to that of Figure 11.

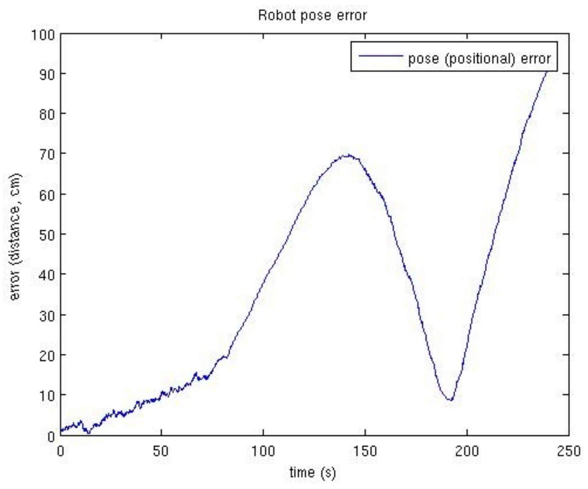


Figure 31: The positional error using the PF looks virtually identical to the scenario used for Figure 14.

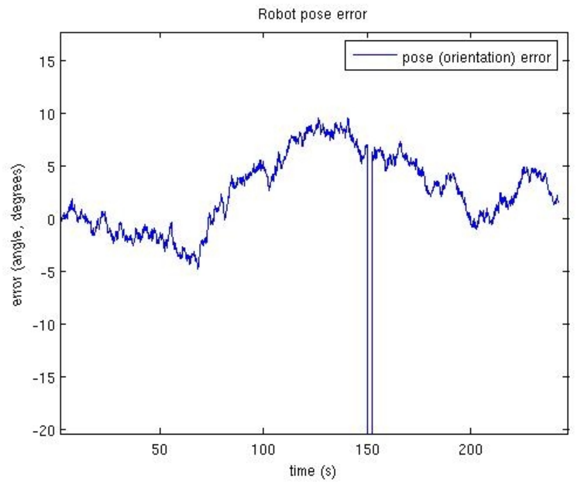


Figure 32: The orientational error using the PF looks virtually identical to the scenario used for Figure 15.

3.2.2 Positional tracking with no objects

The results of this test case are practically on par with the results of section 3.1.2. In fact, the graphs of *Figure 31* and *Figure 32* are difficult to tell apart from the corresponding figures using the EKF, *Figure 14* and *Figure 15* at first glance. This scenario covered the basic prediction without integrating the detection and correction of landmark objects from the map.

3.2.3 Positional tracking, no objects and slow rotational velocity

For this scenario, the PF positional error was about 50% more than for the scenario involving the EKF (*Figure 33*). However, the orientational error was only marginally better than in the corresponding EKF case, by no more than about ½ of a degree (*Figure 34*).

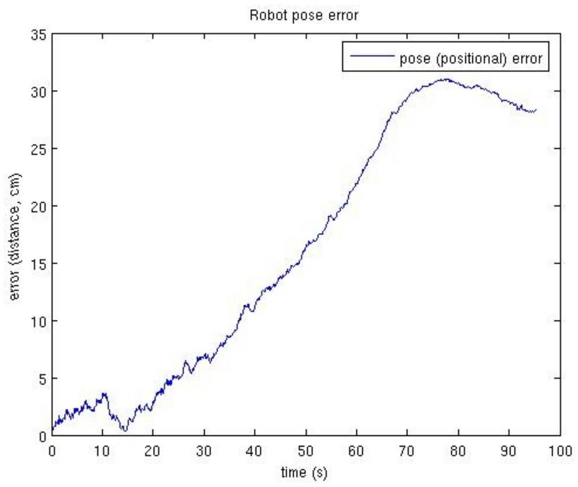


Figure 33: Positional error is slightly worse than the scenario for Figure 16.

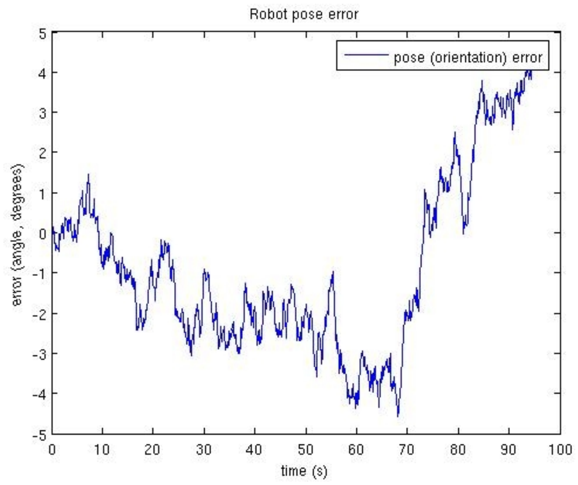


Figure 34: Orientational error is similar to that of Figure 17, the magnitude of the peaks is only slightly less than in the EKF case.

3.2.4 Positional tracking, no objects and slow rotational velocity, large Δt

As seen in the results of section 3.1.4, once a Δt of 1.0 is used, the positional error increased nearly linearly. The actual position graph vs. the believed position graph (*Figure 35*) actually looks significantly worse than in *Figure 18*, which was already pretty bad. Once again, the positional error increased nearly linearly, while the orientational error was not as severely affected. The lesson to take away from these results (and the results of section 3.1.4) is that it is probably better to use a small time step in order to minimize errors.

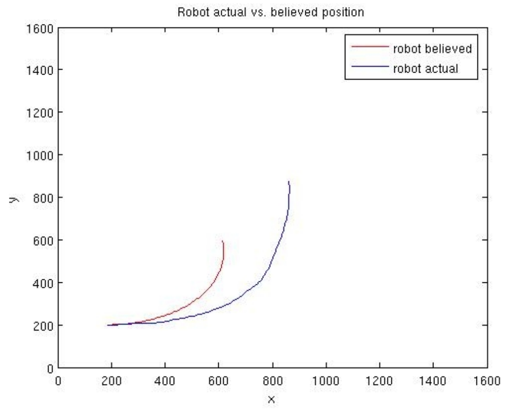


Figure 35: The actual vs. believed position start to diverge quite rapidly once a large Δt value is used (here it is 1.0)

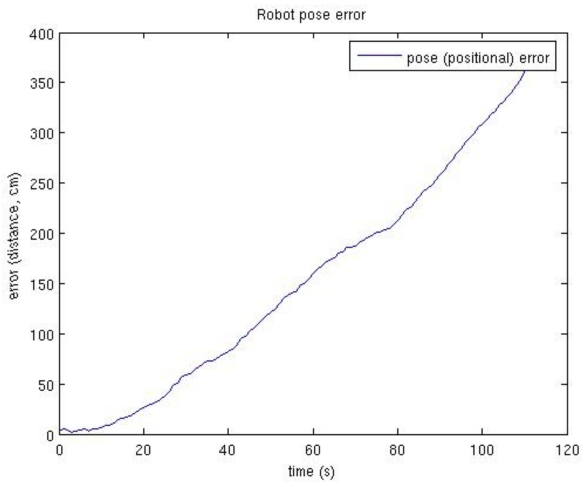


Figure 36: The positional error increases nearly linearly due to a large time step of 1.0.

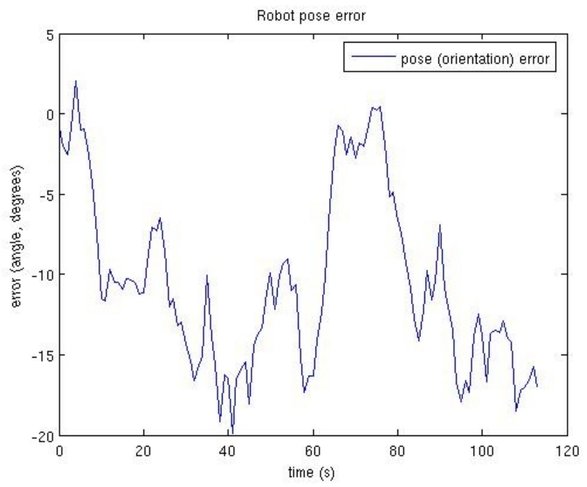


Figure 37: The orientational error is hardly distinguishable from Figure 20, and suffers similarly even with the PF algorithm.

3.2.5 Positional tracking, two objects and slow rotational velocity

In this test case, from the figures presented below seem to indicate similar (though slightly worse) performance to the scenario for the EKF. However, not shown are the several runs of the scenario that did not fare as well. In a rough estimate, 1 out of 5 cases ended up with the algorithm choosing the wrong particle set at which point the positional error increased rapidly. This is likely due to implementation errors and not a fault of the algorithm itself.

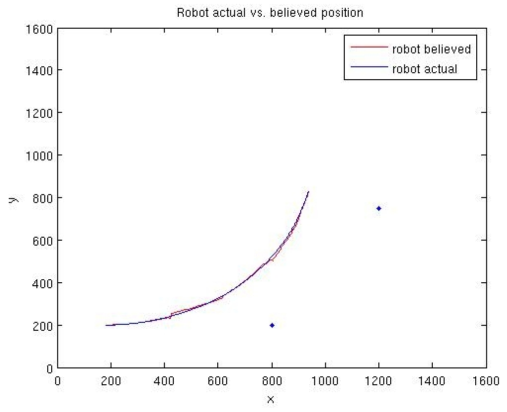


Figure 38: The PF does a decent job of following the robot's actual position by integrating the measurement updates from sensed landmarks and resampling the particle set according to assigned weights.

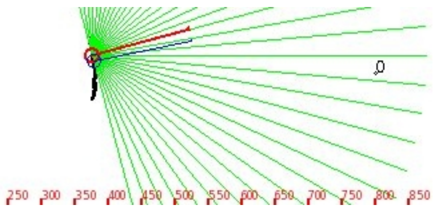


Figure 39: Visualization of the particles used in pose estimation by the PF algorithm. The particles spread out according to a motion model with added noise.

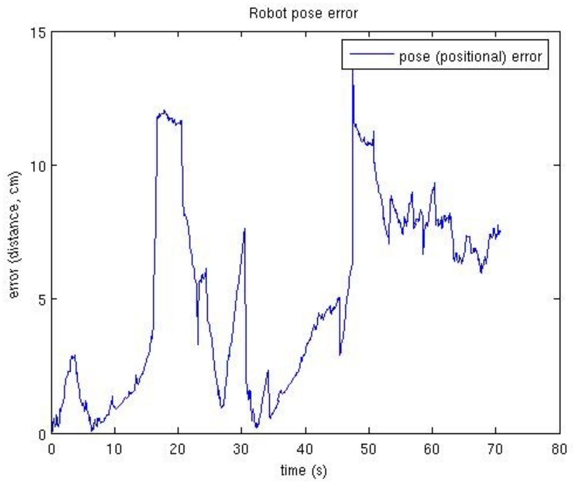


Figure 40: The positional accuracy in the best cases was slightly worse than in the scenario using the EKF, though this may be an implementation error.

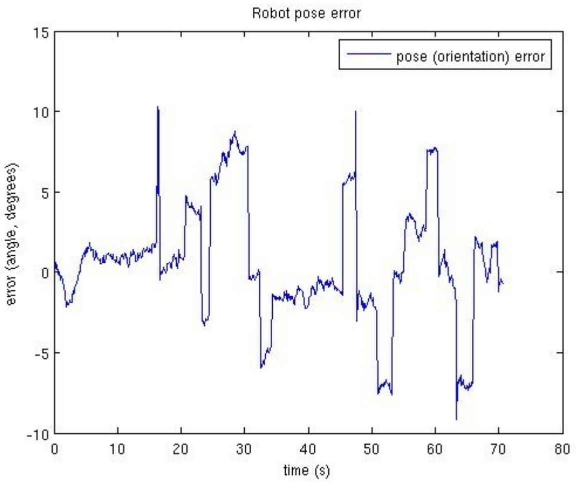


Figure 41: The orientational accuracy is not too bad if you exclude the large number of spikes.

3.2.6 Positional tracking with sensor noise

The results presented in this scenario are quite disappointing compared to the same scenario using the EKF in section 3.1.6. There are two possible explanations. Either the PF is simply inferior to the EKF in this particular scenario (unlikely) or there are implementation errors that are definitely manifesting in the scenario's results. Whatever the case, the results of *Figure 42* and *Figure 43* are not worth discussing further without a deeper look into the algorithm and implementation.

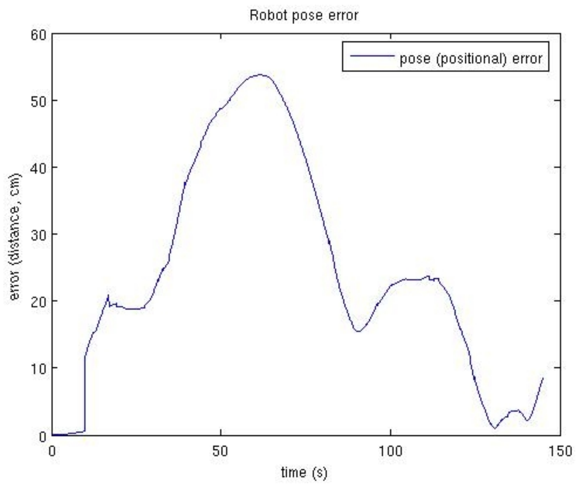


Figure 42: An interesting curve for the positional error, but disappointing once it is compared against the EKF of Figure 23.

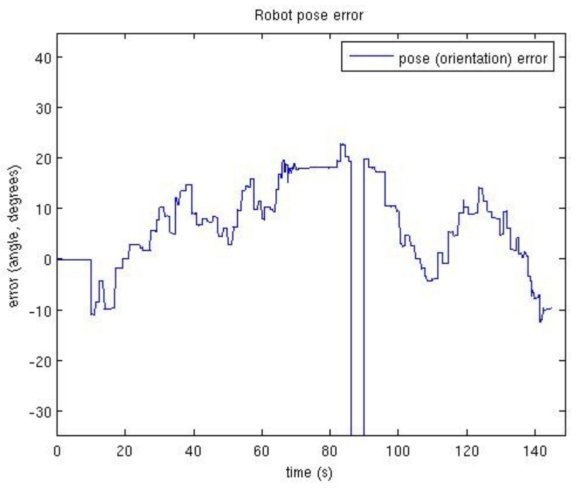


Figure 43: Of course there is a large amount of orientational error associated with the scenario, which is almost expected given the results shown in Figure 42.

3.2.7 Positional tracking with control noise

In this test case control noise was modeled in the agent and control noise was added in the test bed. As a reminder, the following values were used: testbed motion noise $\sigma_v = 4\frac{cm}{sec}; \sigma_\omega = 2\frac{deg}{sec}$; agent motion noise parameters $(\alpha_1 = .30 \quad \alpha_2 = .10 \quad \alpha_3 = .10 \quad \alpha_4 = .20 \quad \alpha_5 = .10 \quad \alpha_6 = .10)$. The robot performed two loops, and the PF did not do well in this case. As is noticeable in figured *44-47*, it is highly lightly a bug is rearing its ugly head – leading to large amounts of positional and orientational errors.

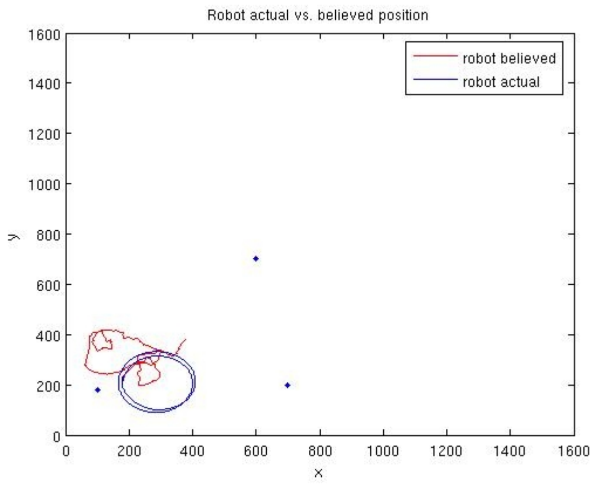


Figure 44: What happened here? The believed robot position is way out of line. Could it be too large of values for the motion orientation noise parameters α_5 *and* α_6 *?*

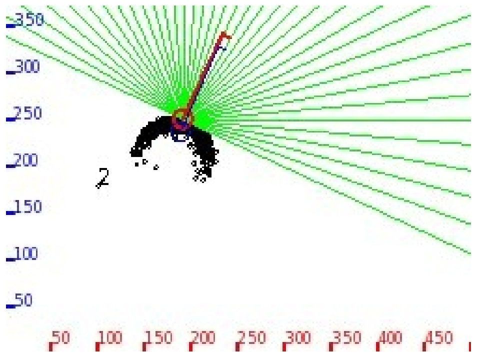


Figure 45: Due to large values for the noise parameters, the particles spread out quite rapidly.

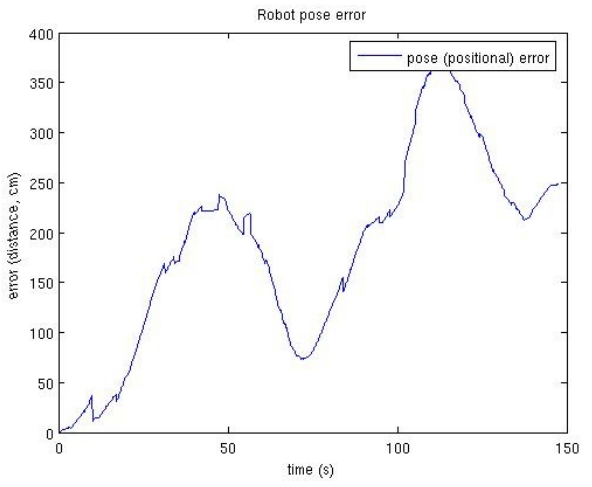


Figure 46: The magnitude of the positional error is very poor and hints strongly that there are implementation errors.

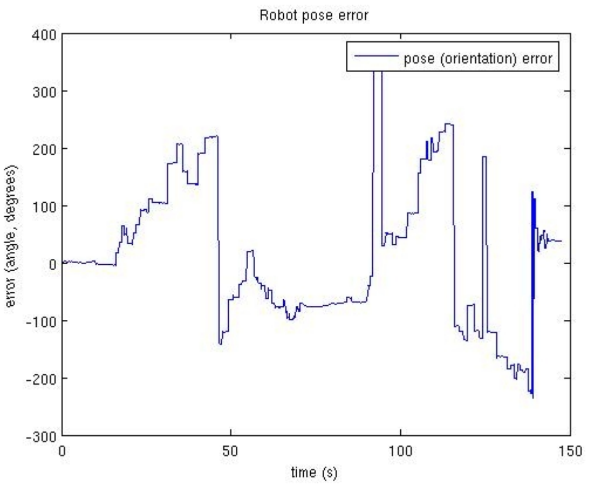


Figure 47: Similar results to those of Figure 46.

3.2.8 Positional tracking with control noise

This test case is quite similar to the the one performed for section 3.2.7, with the exception that it was performed to see what the effect of the α_5 and α_6 parameters were on the magnitude of positional and rotational error seen for the previous subsection's results. Here $\alpha_5 = 0$ and $\alpha_6 = 0$, rather than 10% for both as is the case with section 3.2.7. In *Figure 48*, for the first 100 seconds the positional error is significantly reduced from *Figure 46*. Towards the end of the simulation, however, the positional error is not much better than *Figure 46*. *Figure 49* shows similar improvement over *Figure 47*, while *Figure 50*'s path is still undesirable even though it is improved over the previous section's.

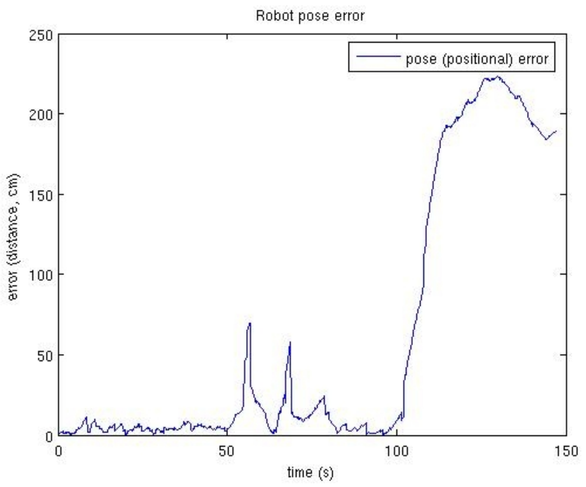


Figure 48: The PF does a good job for the first 100 seconds, and then positional error spikes rapidly and doesn't recover by the end of the simulation run of two loops.

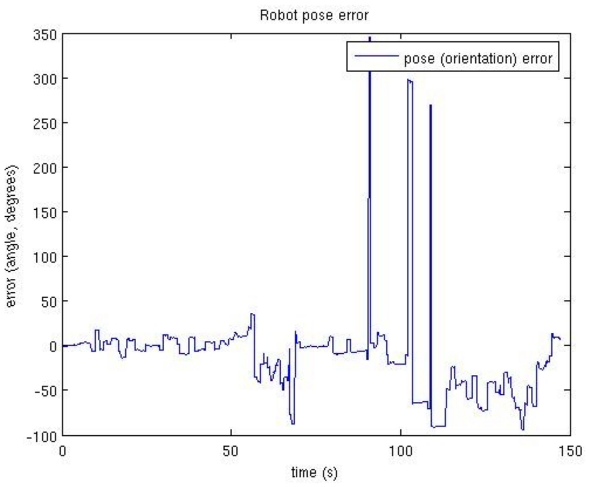


Figure 49: The orientational error drops significantly once the associated parameters in the motion noise are set to 0.

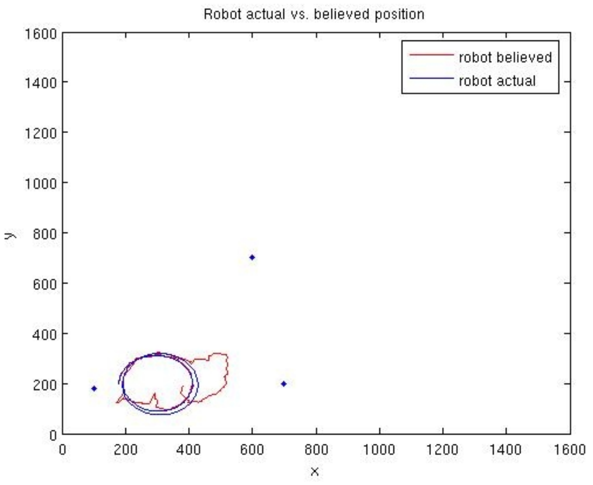


Figure 50: The robot's believed position follows quite closely for the first loop before veering off significantly during the second loop.

4. Conclusion

Summarizing the comments of the previous section, it is clear that there are some issues with the PF results. Overall I was fairly impressed with the performance of the EKF, especially once noise was introduced. The EKF was able to reduce its error over time using the measurement updates. The same cannot be said for the implementation of the PF. Though the results of the first few test cases of the PF seems to nearly mirror those of the EKF, it became clearer that either the PF was lacking in the same scenarios or that there were implementation errors. It is likely that with more effort that implementation errors would be uncovered and better results would be seen with the PF algorithm.

5. Bibliography

1. Thrun, S., Burgard, W., Fox, D., "Probabilistic Robotics," *MIT Press*, 2005