Spring 5-18-2012

The Interacting Multiple Models Algorithm with State-Dependent Value Assignment

Rastin Rastgoufard
University of New Orleans, rrastgou@uno.edu

The Interacting Multiple Models Algorithm with State-Dependent Value Assignment

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Engineering
Electrical

by

Rastin Rastgoufard

B.Sc. Tulane University, 2008

May, 2012

Acknowledgements

I had a lot of fun in the three and a half years of my master's degree. For that, I would like to thank *all* of the professors of UNO's Department of Electrical Engineering and the students who were with me.

I would like to make special mention of the three professors in the Information Systems Lab: Dr. Jilkov, Dr. Chen, and my advisor, Dr. Li. Your continuous instruction and encouragement shaped my coursework, provided the path to my master's thesis, and radically expanded the boundaries of my mind.

Also, I would like to thank my parents. Mom, Dad, you two take really good care of me.

Contents

List of Figures **iv**

Abstract **v**

Introduction **1**
Toy Problem . . . 1
Goals . . . 2
Brief Overview of Existing Literature . . . 5
Brief Overview of Proposed Methods . . . 7

Kalman Filter **8**
Algorithm . . . 8
Small Visual Example . . . 9

Interacting Multiple Models **11**
Step 1, Mix . . . 11
Step 2, Run Individual Filters . . . 12
Step 3, Remix . . . 14
Prediction . . . 14
Small Visual Example . . . 15

State-Dependent Value Assignment **18**
A State's Value . . . 18
SD Model Probabilities . . . 19
SD Transition Probabilities . . . 20
Small Visual Example . . . 21

Experiment and Results **23**
Experimental Design . . . 23
Results . . . 26

Discussion **28**
State-Dependent Performance . . . 29
State-Dependent Comparison . . . 30
The Value Function . . . 32

Conclusion **34**

References **36**

Appendices **37**
Simulation Results . . . 37
Code Listing . . . 49

Vita **83**

List of Figures

#		Page
1	Vehicle passes beside two obstacles.	4
2	Vehicle enters an obstacle.	4
3	One entire player-controlled trajectory.	4
4	Another entire trajectory.	4
5	KF, One Step	10
6	KF, One Step, Two Models	10
7	A first step using the IMM algorithm.	17
8	A second step using the IMM algorithm.	17
9	IMM's prediction for the third step.	17
10	The value function $s(x)$, two different β.	19
11	SD TPM Example	22
12	Mode sequences over two time steps.	24
13	**Performance Summaries**	28
14	Truth and Predictions.	31
15	SD cases with good behavior.	31
16	SD MPs sometimes better than SD TPM.	33
17	Inaccurate TPM.	33
18	2011.10.17.11.36.22, 0.08 to 74.15	37
19	2011.10.17.11.36.22, 10.00 to 15.00	38
20	2011.10.17.11.36.22, 10.00 to 50.00	39
21	2011.10.17.11.36.22, 15.00 to 20.00	40
22	2011.10.17.11.36.22, 25.00 to 30.00	41
23	2011.10.17.11.36.22, 30.00 to 35.00	42
24	2011.10.17.11.36.22, 30.00 to 60.00	43
25	2011.10.17.11.36.22, 35.00 to 40.00	44
26	2011.10.17.11.36.22, 45.00 to 50.00	45
27	2011.10.17.11.36.22, 50.00 to 55.00	46
28	2011.10.17.11.36.22, 55.00 to 60.00	47
29	2011.10.17.11.36.22, 65.00 to 70.00	48

Abstract

The value of a state is a measure of its worth, so that, for example, waypoints have high value and regions inside of obstacles have very small value. We propose two methods of incorporating world information as state-dependent modifications to the interacting multiple models (IMM) algorithm, and then we use a game's player-controlled trajectories as ground truths to compare the normal IMM algorithm to versions with our proposed modifications. The two methods involve modifying the model probabilities in the update step and modifying the transition probability matrix in the mixing step based on the assigned values of different target states. The state-dependent value assignment modifications are shown experimentally to perform better than the normal IMM algorithm in both estimating the target's current state and predicting the target's next state.

Keywords: IMM, state-dependent, constraints, penalty function, waypoints, obstacles

Introduction

Toy Problem

We created a "game" that allows the player to generate ground truth trajectories. The user controls a vehicle, in real time, through a two-dimensional world which contains several obstacles, circular regions of varying radii. The vehicle is not supposed to enter the obstacle regions as it navigates around. Consider Figures 1 to 4 for visual reference during the following explanation.

The vehicle itself is shown in blue. The **open blue circle** shows the location of the vehicle, and the **blue dot** shows the direction that the vehicle is facing. The player controls both the direction of the vehicle, using the left and right arrow keys, and the forward speed of the vehicle, using the space-bar key. In the game, the vehicle's rotation and forward speed are not coupled, meaning that the vehicle can rotate while stopped. The maximum turn rate is constant regardless of the vehicle's forward speed.

The **black dots** indicate the boundary of an obstacle. In Figure 1, two obstacles are visible. One has a large radius and is located up and to the left of the vehicle. The other is small and is centered at (2,2). In the same figure, the **small red X** shows where the center of the (2,2) obstacle is located.

The playing world contains a total of four obstacles. There is one **small red dot** for each of the obstacles. The location of each red dot is an indication of where each corresponding obstacle's boundary is located with respect to the vehicle. In Figure 1, there are two red dots immediately to the left of the vehicle that indicate the vehicle is close to two obstacle boundaries. There is another red dot located up and to the right of the vehicle that indicates there is a slightly distant obstacle in that direction. There is a final red dot below and to the left of the vehicle that indicates there is an obstacle very far from the vehicle in that direction.

Figures 1 to 4 were created after the player controlled the vehicle for approximately 75 seconds. The **black line** shows a sliding window of the vehicle's trajectory. The **green circles** show snapshots of the vehicle's trajectory taken every $T = 0.35$ seconds. Neither of these was visible to the player before the run was completed, as both would have required knowledge of the future.

The player can drive the vehicle anywhere in the toy world. The obstacles do not have hard boundaries, and the only penalty for entering an obstacle is the appearance of a **heavy red X** in place of an obstacle's red dot. Figure 2 shows the vehicle inside of an obstacle. The position of the heavy red X with respect to the vehicle shows the direction of the nearest exit point from the obstacle. If this toy world were a game that distributed points, then the player would always choose to exit toward the heavy red X in order to minimize the amount of points lost for being in an obstacle.

Goals

Three main goals form the basis for this thesis. It is important to use a real world target, to incorporate the obstacle information into a tracking algorithm, and to track (both estimate and predict) the motion of that target.

1. Real World Target

   The vehicle that is controlled by the player behaves like a “real world” target. The target has a very wide array of possible maneuvers, and the player can make decisions of how to move the target in real time. This is in contrast with an algorithmically determined target that is often used in computer simulations. This goal is important because a real world target’s behavior cannot be neatly captured in a small set of models and as such creates a realistic and challenging tracking problem.

2. Obstacle Information

   The presence of the obstacles changes where the vehicle is allowed to travel. Figure 3 and Figure 4 show entire player-controlled trajectories. It is quite obvious that specific areas of the world were avoided due to the obstacles. Knowledge of the obstacles should improve the performance of tracking algorithms. In this thesis, we incorporate the obstacle information into the interacting multiple models (IMM) algorithm.

3. Estimation and Prediction

   There are at least two different ways to evaluate the performance of a tracking algorithm. One involves estimating the state of a target using all currently available data points. Another involves predicting the state of the target at a future time using all currently available data points. Varying the measurement noise level has the effect of focusing on either one or the other. When there is very little measurement noise, the estimation error is negligible and the focus shifts toward prediction. When there is larger measurement noise, the estimation performance becomes the focal point.

The goal is not necessarily to find the best estimator or predictor for the toy problem; the goal is to show that even with very crude assumptions, embedding the world information into the tracking algorithms yields better results than not incorporating it.

The first point, using a real world target, is a fundamental underlying assumption. The second point, incorporating the obstacle information, is addressed mathematically in the section titled **State-Dependent Value Assignment**. The third point, evaluating the performances of the proposals, is covered in the **Experiment and Results** and **Discussion** sections.

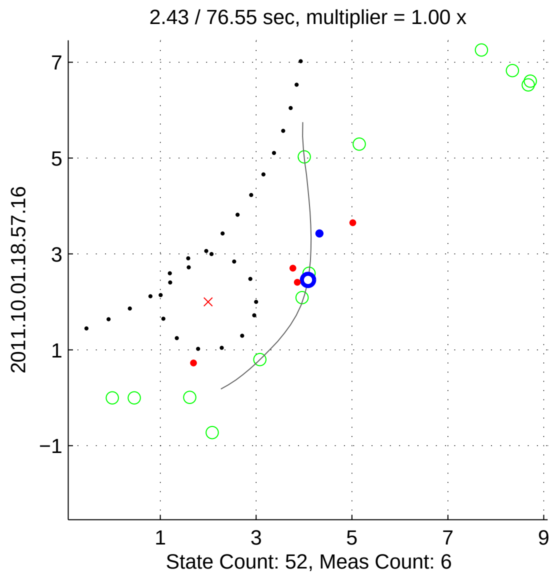


Figure 1: Vehicle passes beside two obstacles.

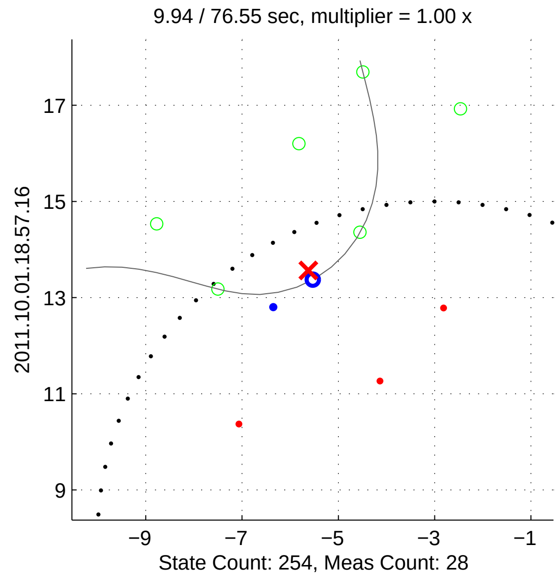


Figure 2: Vehicle enters an obstacle.

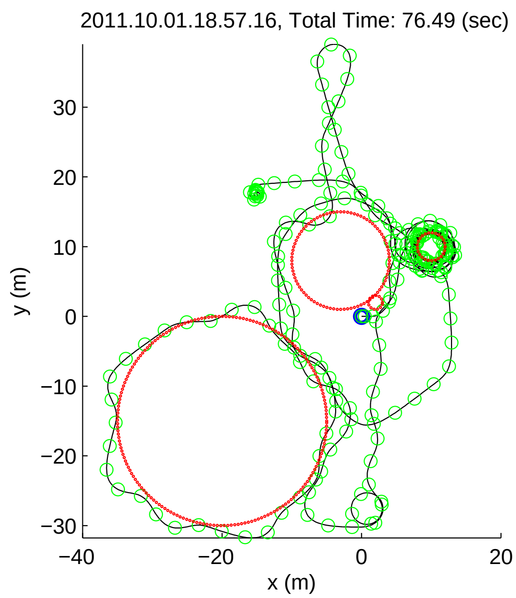


Figure 3: One entire player-controlled trajectory.

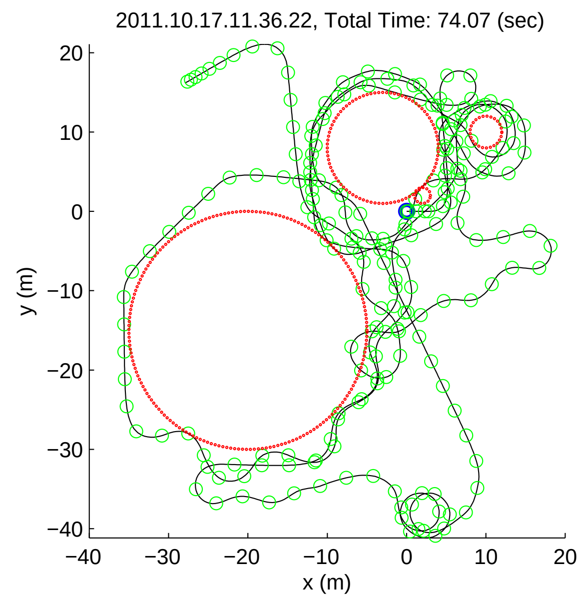


Figure 4: Another entire trajectory.

Brief Overview of Existing Literature

The **Toy Problem** would be a typical target tracking with noisy measurements problem, but the presence of obstacles makes it relatively unique.

Target tracking problems are often modeled as "hybrid systems" [1] in which the target's state is continuous, but the target moves according to only one of a finite number of modes or models at any time. This modeling is applicable to the **Toy Problem**. A very popular algorithm to solve this hybrid estimation problem is the Interacting Multiple Models (IMM) algorithm which runs several Kalman filters [2] in parallel and merges their results depending on measurements. The IMM algorithm is popular because it is very cost-efficient [3–5], meaning it performs relatively well and is computationally inexpensive to calculate.

An application of the IMM algorithm is demonstrated in [6] in which a vehicle is driving along a highway. Two conditions are of interest – maintaining a lane or changing lanes. There is a motion model and associated "directional" process noise that corresponds to maintaining a lane, and there is a different motion model with a different type of process noise associated with the lane change maneuver. [6] shows that the IMM algorithm tracks the vehicle well under both conditions and quickly determines when lane changes happen. A complex behavior is captured neatly by two models.

The **Toy Problem** is a simple problem that is well suited to the normal IMM algorithm – with some assumptions on the behavior of the target, the system can be modeled using only five modes of operation. (Refer to **Experiment and Results**.) However, problems often require many more modes of operation to characterize a target's range of motion. The IMM algorithm's performance suffers when there are too many motion models that overlap and compete [7]. As a result, researchers developed variable-structure multiple model (VSMM) algorithms that perform better than the normal, fixed-structure IMM algorithm [7–11].

While the **Toy Problem** is not a very complicated problem that requires variable-structure algorithms, those algorithms are of interest in this problem because of the fact that the set of models can be adapted based on the target's current state. For example, [7] describes a problem in which the acceleration of the target cannot change rapidly. An overarching set of models is designed to cover all of the possible target accelerations, but at any time the VSMM algorithm uses the set of accelerations that are "near" the target's current acceleration. As the target's acceleration changes, the VSMM algorithm chooses different sets of models accordingly.

Some researchers have implemented VSMM algorithms with model selection or switching rules that are based not only on the target's internal state but also on the properties of the world around the target. For example, [12] and [13] limit the available modes of operation based on the presence of roads and whether or not the target is on a road. [12] describes a general ground target tracking problem where a target might be navigating in an unconstrained environment (off-road), it might be near a road, or it might be constrained to be on a piece-wise linear road. Furthermore, roads might have junctions, in which case the target can choose from different branches. Each condition, including motion at a junction, is captured by a different set of models, and there is a very intricate method of selecting which models are applicable. [13] expands the on-road condition and describes how to incorporate the actual curvature of road segments as constraints.

In existing VSMM methods, the state-dependent information is captured by the strategic selection and omission of models. The work in this thesis, in [14, 15], and in [16] consider how to embed state-dependent information even deeper into the IMM algorithm. These methods could complement the VSMM methods and would operate after the set of models is selected. All of these methods modify the modes' transition probability matrix and the modes' likelihoods based on the target's state.

[16] considers a problem in which a target can choose to stop randomly as an evasive maneuver. The authors make the argument that real world targets cannot instantaneously cease their motion, and thus the probability that the target will stop is small when its speed is large. The transition probability matrix governing the switching of motion models depends on the speed of the target.

Guard conditions [14, 15] use a transition probability matrix that depends on the proximity to a waypoint. An example is given in [14] where an airplane should turn toward a new destination when it arrives at a waypoint. There are two guard conditions to model this desired behavior. The first is to switch from constant velocity to coordinated turn when the plane's position is near the waypoint. The second is to switch away from coordinated turn back to constant velocity when the plane's heading is near a specific angle.

The work in this thesis is most similar to [14]. A very large difference is that their work is derived theoretically when the guard conditions are of specific forms. The method described here does not have theoretical support but is instead slightly more flexible.

Brief Overview of Proposed Methods

There are two locations in the IMM algorithm that allow the world information to be incorporated. The first is in the model probabilities (MPs) update/remix step that uses the likelihoods of each mode. The second is in the transition probability matrix (TPM) that tells the probability of transitioning from one mode of operation to another mode of operation.

Just before the end of one cycle of the IMM algorithm there is one estimated state for each of the models in the algorithm. Each of these states is assigned a value, and the values then modify the weights of their respective models during the final update/remix step of the algorithm. This process is described in **SD Model Probabilities**. Those same estimated states will interact in the mixing step to obtain the next cycle's initialization. Before that happens, each estimated state is propagated by all of the modes of motion to determine "what-if" predicted states. The number of these predicted states is equal to the number of elements in the transition probability matrix. The assigned values of these predicted states characterize the values of the transitions, and thus they are used to modify the transition probability matrix, as described in **SD Transition Probabilities**.

The value assignment is problem specific and up to the designer in the same way as the choice of models. The section titled **A State's Value** shows how value assignment is defined in this thesis. Value assignment has an effect that is similar to penalty functions in constrained optimization problems – that is, it can convert a tracking problem with constraints into an unconstrained one.

The layout of this thesis follows the order of building blocks. First, the Kalman filter is described, followed by the IMM algorithm which is based on the Kalman filter. Then comes state-dependent value assignment, the main novelty of this thesis, which modifies parts of the IMM algorithm. Finally the experiment that implements the original problem and tests the performance of the value system is followed by an analysis and discussion of the results.

Kalman Filter

Algorithm

Consider the following dynamics and measurement model. x_k is the state of the system at time k. A_k is the dynamics matrix that advances the state from time k to time $k+1$. $w_k \sim \mathcal{N}(0, Q_k)$ is the process noise. z_k is the measurement at time k. H_k is the sensing matrix. $v_k \sim \mathcal{N}(0, R_k)$ is the sensor noise.

$$x_{k+1} = A_k x_k + w_k \tag{1}$$

$$z_{k+1} = H_{k+1} x_{k+1} + v_{k+1} \tag{2}$$

Beginning with a state estimate at time k, the Kalman filter first predicts the state at time $k+1$ then updates the prediction when the measurement at time $k+1$ arrives. The result is an estimate of the state at time $k+1$.

$$\text{Given} : \hat{x}_{k|k}, \hat{P}_{k|k}, A_k, Q_k$$

$$\hat{x}_{k+1|k} = A_k \hat{x}_{k|k} \tag{3}$$

$$\hat{P}_{k+1|k} = A_k \hat{P}_{k|k} A_k' + Q_k \tag{4}$$

The variables in Equations (3) and (4) with time index $k+1|k$ are predictions. When the $k+1$th measurement arrives, the Kalman filter first computes the filter gain K_{k+1}.

$$\text{Given} : \hat{x}_{k+1|k}, \hat{P}_{k+1|k}, z_{k+1}, H_{k+1}, R_{k+1}$$

$$y_{k+1} = z_{k+1} - H_{k+1} \hat{x}_{k+1|k} \tag{5}$$

$$S_{k+1} = H_{k+1} \hat{P}_{k+1|k} H_{k+1}' + R_{k+1} \tag{6}$$

$$K_{k+1} = \hat{P}_{k+1|k} H_{k+1}' S_{k+1}^{-1} \tag{7}$$

After the filter gain has been computed, the Kalman filter updates the state and uncertainty estimates.

$$\hat{x}_{k+1|k+1} = \hat{x}_{k+1|k} + K_{k+1} y_{k+1} \tag{8}$$

$$\hat{P}_{k+1|k+1} = \hat{P}_{k+1|k} - K_{k+1} S_{k+1} K_{k+1}' \tag{9}$$

The final result of the Kalman filter is the state estimate $\hat{x}_{k+1|k+1}$ along with an estimate of its uncertainty $\hat{P}_{k+1|k+1}$.

Small Visual Example

Figure 5 shows one step of the Kalman filter. Figure 6 shows one step of the Kalman filter for two different dynamics models.

The state variable contains two quantities: a linear displacement x and a linear velocity $\dot{x}$. The measurement z contains only displacement information; it does not directly measure the velocity. The velocity part of the state is not shown in the two figures so that both the state and the measurement can be displayed in the same space.

Figure 5 shows a black line that has its peak centered at $\hat{x}_0$. The width of the curve is a measure of the uncertainty of the initialization, $\hat{P}_0$. The dynamics model A_0 helps to predict the state, $\hat{x}_{1|0}$, and uncertainty, $\hat{P}_{1|0}$, at the next time step. This pair is depicted by the green curve.

A new measurement, z_1, arrives at time $k = 1$. The measurement has a noise level, and thus the value and uncertainty of the measurement are shown as a (red) curve instead of a single point. The actual value of the measurement is the center of the curve. Note that there is a disparity between the green curve, the predicted state, and the red curve, the current measurement. The Kalman filter's role is to weigh and appropriately combine the two.

The uncertainties of the predicted state and the measurement are weighed against each other in order to obtain the filter's gain. The gain is a measure of how much "correction" the predicted state requires now that the newest measurement has arrived. The updated state, $\hat{x}_{1|1}$, has been "corrected" by the filter and is shown as the orange curve. Just like the other variables, this state has its value at the center of the curve, and the width of the curve is a measure of its uncertainty.

Part of Figure 6 contains the same content as Figure 5. The other part is a repeat of the previous example, except the dynamics matrix A_0 is different. For the same measurement, there are two different updated state estimates. The **Interacting Multiple Models** algorithm is a way to combine together the two estimates.

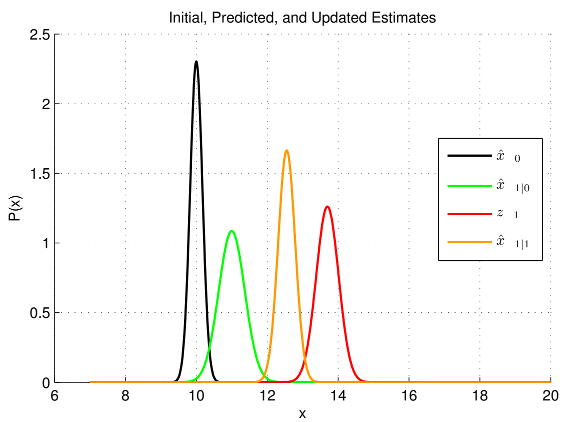


Figure 5: One step of the Kalman filtering algorithm. The initial estimate is $\hat{x}_0$, and its uncertainty $\hat{P}_0$ is depicted by the Gaussian curve centered at $\hat{x}_0$ (shown in black). The green curve is $\hat{x}_{1|0}$, the filter's prediction of the next state. The red curve is the measurement z_1. The orange curve is the updated estimate $\hat{x}_{1|1}$.

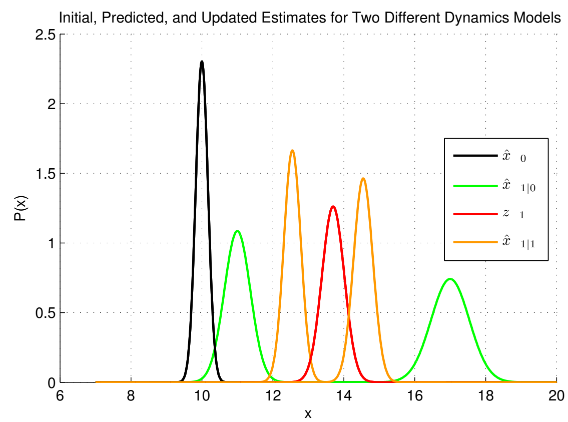


Figure 6: One step of the Kalman filtering algorithm, but showing two different dynamics models. The colors are the same as in Figure 5. There is only one measurement z_1, but since there are two dynamics models, there are two sets of $\hat{x}_{1|0}$ and $\hat{x}_{1|1}$. Both models are initialized at the same point $\hat{x}_0$.

Interacting Multiple Models

The Interacting Multiple Models (IMM) algorithm runs several Kalman filters in parallel. The individual filters are initialized using a mixture of results from the previous step's filters. The output of the IMM algorithm, the overall state estimate, is also a mixture of the individual filters' estimates.

The IMM algorithm requires three items. The first is a set of Kalman filters, one for each of M models or modes of operation. The second is a probability vector, μ_k, that contains the set of probabilities that the ith model is in effect at the current time step, k. The third is a transition probability matrix (TPM) that tells how probable it is to jump from model i at time k to model j at time $k+1$.

The IMM algorithm itself consists of three main steps.

1. Mix the model probabilities $\mu_{i,k}$ based on the TPM, and initialize several $\hat{x}^0_{j,k+1}$, $\hat{P}^0_{j,k+1}$ based on the mixed probabilities.

2. Run M separate Kalman filters starting on each $\hat{x}^0_{j,k+1}$, $\hat{P}^0_{j,k+1}$ to obtain $\hat{x}_{j,k+1}$, $\hat{P}_{j,k+1}$.

3. Mix the estimates $\hat{x}_{j,k+1}$, $\hat{P}_{j,k+1}$ based on the model probabilities and the likelihoods of obtaining the innovations $y_{j,k+1}$.

Step 1, Mix

For this step, the IMM algorithm requires three sets of components. It requires a vector of model probabilities at the current time k. In addition, it requires a transition probability matrix for the current time. Finally, it requires the M individual filters' estimates at time k. All of these are required in order to begin the IMM algorithm for time $k+1$. Note that this mixing step occurs before z_{k+1} has arrived.

Let the column vector μ_k denote the probabilities of the M models such that the ith entry is the probability that model i is in effect at time k. Then, let T_k be the TPM that tells the probabilities of transitioning from model i to model j at time k. The elements of the TPM are $T_{ji,k} = P(m_{j,k+1}|m_{i,k})$. Note that each column of T_k must sum to one, and that

$$\mu^p_{k+1} = T_k \mu_k \tag{10}$$

where μ^p_{k+1} is the vector of predicted mode probabilities at time $k+1$ given only T_k and μ_k.

The mixing step begins by calculating the probabilities $\mu_{ij,k} = P(m_{i,k}|m_{j,k+1})$.

$$\mu_{ij,k} = T_{ji,k}\mu_{i,k}/\mu^p_{j,k+1} \tag{11}$$

Each $\mu_{ij,k}$ is the probability that mode i was in operation at time k given that mode j is in operation at time $k+1$. Note that z_{k+1} has not yet arrived, so μ^p_{k+1} is a prediction.

The next part of this mixing step is to create M initializations for the individual Kalman filters. From the previous iteration, the ith Kalman filter had a state estimate $\hat{x}_{i,k|k}$ and uncertainty estimate $\hat{P}_{i,k|k}$. These estimates are mixed together to form new $\hat{x}^0_{j,k+1}$ and $\hat{P}^0_{j,k+1}$.

$$\hat{x}^0_{j,k+1} = \sum_{i=1}^{M} \hat{x}_{i,k|k}\mu_{ij,k} \tag{12}$$

$$\hat{P}^0_{j,k+1} = \sum_{i=1}^{M} \hat{P}_{i,k|k}\mu_{ij,k} + X_{j,k} \tag{13}$$

The term after the second summation is $X_{j,k}$, the so called "spread of the means" [5]. Define $d_{ij,k}$ and use it to compute $X_{j,k}$.

$$d_{ij,k} = \hat{x}^0_{i,k+1} - \hat{x}^0_{j,k+1}$$

$$X_{j,k} = \sum_{i=1}^{M} d_{ij,k}d'_{ij,k}\mu_{ij,k}$$

The final results of this mixing step are the filter initializations $\hat{x}^0_{j,k+1}$, $\hat{P}^0_{j,k+1}$ and the predicted model probabilities μ^p_{k+1}. The sets of $\mu_{ij,k}$ and $X_{j,k}$ are not used outside of this first mixing step.

Step 2, Run Individual Filters

The first step of the IMM algorithm resulted in $\hat{x}^0_{j,k+1}$ and $\hat{P}^0_{j,k+1}$. This step combines those quantities with z_{k+1} to obtain $\hat{x}_{j,k+1}$ and $\hat{P}_{j,k+1}$. This step also produces $\lambda_{j,k+1}$, the likelihood of the measurement z_{k+1} given the jth model is in effect. These likelihoods will be combined with μ^p_{k+1} in the next step of the IMM algorithm to find updated model probabilities μ_{k+1}.

Each of the M models has its own dynamics and measurement equations similar to Equations (1) and (2). The dynamics, sensing, process noise, and measurement noise matrices have model dependencies in addition to time dependencies. However, the same measurement z_{k+1} is used by all M models. Equations (14) and (15) show the dynamics and measurement equations from the point of view of the jth model.

$$x_{j,k+1} = A_{j,k}x_{j,k} + w_k \tag{14}$$

$$z_{k+1} = H_{j,k+1}x_{j,k+1} + v_{k+1} \tag{15}$$

The process noise is $w_k \sim \mathcal{N}(0, Q_{j,k})$. The measurement noise is $v_{k+1} \sim \mathcal{N}(0, R_{j,k+1})$.

This second step of the IMM algorithm is to apply the M Kalman filters. Each filter uses the same measurement, but each filter begins at a unique $\hat{x}^0_{j,k+1}$ and $\hat{P}^0_{j,k+1}$. The Kalman filter algorithm is detailed in Equations (3) to (9). Equations (3) and (4) are replaced by the following equations.

$$\hat{x}_{j,k+1|k} = A_{j,k}\hat{x}^0_{j,k+1} \tag{16}$$

$$\hat{P}_{j,k+1|k} = A_{j,k}\hat{P}^0_{j,k+1}A'_{j,k} + Q_{j,k} \tag{17}$$

The remainder continues as normal with the modification that every variable (except for z_{k+1}) has a model j dependence. That is, the algorithm uses or calculates the following values.

$$\hat{x}_{j,k+1|k}, \hat{P}_{j,k+1|k}$$

$$z_{k+1}, H_{j,k+1}, R_{j,k+1}$$

$$y_{j,k+1}, S_{j,k+1}$$

$$K_{j,k+1}$$

$$\hat{x}_{j,k+1|k+1}$$

$$\hat{P}_{j,k+1|k+1}$$

As soon as $y_{j,k+1}$ and $S_{j,k+1}$ are obtained, the IMM algorithm calculates the likelihood of $y_{j,k+1}$ using the covariance matrix $S_{j,k+1}$. The likelihood is called $\lambda_{j,k+1}$.

$$\begin{aligned} \lambda_{j,k+1} &= \mathcal{N}(y_{j,k+1} \mid 0, S_{j,k+1}) \\ &= \frac{1}{\sqrt{\det 2\pi S_{j,k+1}}} \exp\left(-\frac{1}{2}y'_{j,k+1}S^{-1}_{j,k+1}y_{j,k+1}\right) \end{aligned} \tag{18}$$

The combination of the updated state and uncertainty estimates, $\hat{x}_{j,k+1|k+1}$ and $\hat{P}_{j,k+1|k+1}$, with the state likelihoods, $\lambda_{j,k+1}$, is the end result of this step.

Step 3, Remix

The third and final step of the IMM algorithm combines the predicted mode probabilities μ^p_{k+1}, the measurement likelihoods $\lambda_{j,k+1}$, and the individual state estimates $\hat{x}_{j,k+1|k+1}$ and $\hat{P}_{j,k+1|k+1}$.

The updated mode probabilities are μ_{k+1} and take into account all of the models' likelihoods. The jth element of μ_{k+1} is $\mu_{j,k+1}$.

$$\mu_{j,k+1} = \frac{\mu^p_{j,k+1}\lambda_{j,k+1}}{\sum_{i=1}^{M} \mu^p_{i,k+1}\lambda_{i,k+1}} \tag{19}$$

The denominator of Equation (19) is a normalizing factor and is the same for all j.

The overall state estimate given by the IMM algorithm is a weighted combination of the individual filters' estimates. The final state and uncertainty estimates $\hat{x}_{k+1|k+1}$ and $\hat{P}_{k+1|k+1}$ do not have model dependencies.

$$\hat{x}_{k+1|k+1} = \sum_{j=1}^{M} \mu_{j,k+1}\hat{x}_{j,k+1|k+1} \tag{20}$$

$$\hat{P}_{k+1|k+1} = \sum_{j=1}^{M} \mu_{j,k+1}\hat{P}_{j,k+1|k+1} + X_{k+1} \tag{21}$$

The term after the second summation is another "spread of the means." As before in Equation (13), define $d_{j,k+1}$ and use it to calculate X_{k+1}.

$$d_{j,k+1} = \hat{x}_{k+1|k+1} - \hat{x}_{j,k+1|k+1}$$

$$X_{k+1} = \sum_{j=1}^{M} d_{j,k+1}d'_{j,k+1}\mu_{j,k+1}$$

Prediction

The IMM algorithm can be used to obtain state and uncertainty predictions $\hat{x}_{k+1|k}$ and $\hat{P}_{k+1|k}$. The previously-described second and third steps of the IMM algorithm are modified slightly in order to obtain those predictions.

The first step of the algorithm, mixing, obtains μ^P_{k+1} along with $\hat{x}^0_{j,k+1}$ and $\hat{P}^0_{j,k+1}$. The second step begins as normal so that Equations (16) and (17) give individual filter predictions $\hat{x}_{j,k+1|k}$ and $\hat{P}_{j,k+1|k}$. The remainder of the second step is omitted as there is no new measurement.

The main difference in the third step is that all $\lambda_{j,k+1} = 1$ are equal. Therefore, $\mu_{j,k+1} = \mu^P_{j,k+1}$ in Equations (19) to (21). The variable $\hat{x}_{j,k+1|k+1}$ is replaced by $\hat{x}_{j,k+1|k}$ and $\hat{P}_{j,k+1|k+1}$ is replaced by $\hat{P}_{j,k+1|k}$ inside Equations (20) and (21). Instead of obtaining $\hat{x}_{k+1|k+1}$ and $\hat{P}_{k+1|k+1}$, the result is $\hat{x}_{k+1|k}$ and $\hat{P}_{k+1|k}$.

The result of the three steps, after all modifications have been made, is $\hat{x}_{k+1|k}$ and $\hat{P}_{k+1|k}$. These two are the IMM algorithm's one-step prediction.

Small Visual Example

Consider a simple problem that uses an IMM filter with $M = 5$ models. Suppose that initially all models' probabilities are equal.

$$\mu_{j,0} = 1/M$$

Suppose also that the transition probability matrix is constant over time and tends to favor remaining in the current mode. That is, the diagonal elements of the TPM are much larger than the off-diagonal elements.

The initial state is $\hat{x}_0$, shown in Equation (22). $\hat{P}_0$ is very small to indicate that the initialization is accurate. Figure 7 shows the beginning of this example.

$$\hat{x}_0 = [10 \text{ (m)}, 10 \text{ (m)}, 0 \text{ (m/s)}, 10 \text{ (m/s)}]' \tag{22}$$

The five green circles of Figure 7 show the endpoints of the five models' trajectories. These endpoints are the position parts of the individual models' predicted state estimates $\hat{x}_{j,1|0}$.

The red circle is the measurement z_1. The blue crosses represent the individual filters' updated state estimates $\hat{x}_{j,1|1}$. Note that each of the crosses is on a line that connects a green circle $\hat{x}_{j,1|0}$ to the red circle z_1.

The blue circle is the IMM algorithm's final state estimate $\hat{x}_{1|1}$. It is a linear combination of the individual filters' estimates. Because all of the model probabilities are equal initially, the only factor that controls the mixing weights is the likelihood of each model given the measurement. The measurement fits the two left-turn models much better than the straight or right-turn models. Between the two left-turn models, the measurement is more likely to have come from the shallower turn. Thus, the blue circle is closest to the individual state estimate of the shallow-left-turn model.

Figure 8 shows the continuation of the example. Another measurement, z_2, is shown as a red circle. It is connected to the measurement z_1 using a thin red line. There are five blue crosses (two of them are almost stacked) that correspond to the five individual filters' updated state estimates $\hat{x}_{j,2|2}$. The blue circle near the new measurement is the IMM algorithm's updated state estimate $\hat{x}_{2|2}$. This new state estimate is connected to the previous step's estimate using a thin blue line.

Figure 9 shows a one-step prediction using the IMM algorithm. The example continues from before, and now there is no new measurement z_3. The blue crosses, which now represent the individual models' predicted states $\hat{x}_{j,3|2}$, fan out and are not reined in by a measurement. The blue square shows the IMM algorithm's predicted state $\hat{x}_{3|2}$. It is connected to the previous state estimate using a dashed blue line.

Note that the predicted state is very close to the shallow-left-turn model's estimate. This is because of the TPM which favors the continuation of the motion. In going from time $k = 1$ to $k = 2$, the IMM algorithm estimated that the target performed a shallow-left-turn, and thus the algorithm predicts that the same turn will continue when transitioning from time $k = 2$ to $k = 3$.

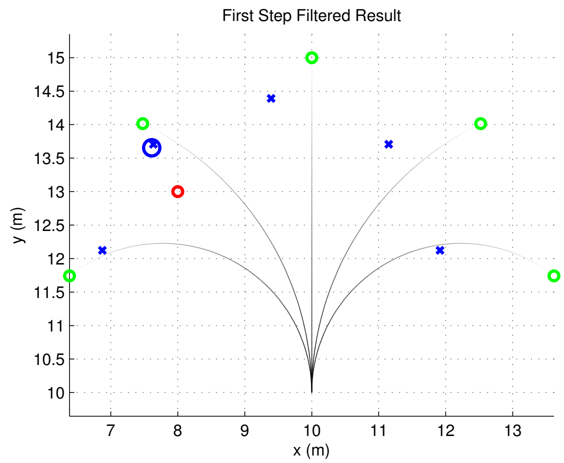


Figure 7: A first step using the IMM algorithm.

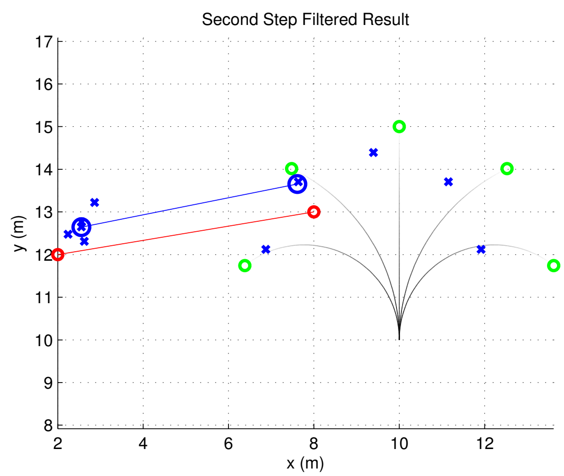


Figure 8: A second step using the IMM algorithm.

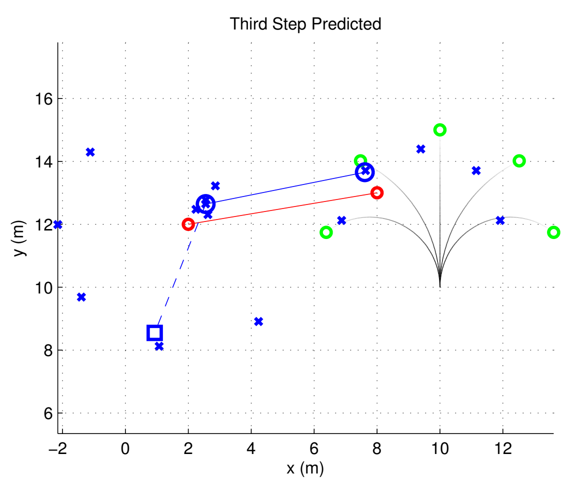


Figure 9: IMM's prediction for the third step.

State-Dependent Value Assignment

To every possible state that the target can take needs to be assigned a penalty or benefit value. In our **Toy Problem**, for example, we could define a simple mapping such that every location inside of an obstacle is assigned a zero and every other location is assigned a one. The state-to-value mapping will be used to modify Equation (19) in **SD Model Probabilities** and Equations (10) and (11) in **SD Transition Probabilities**.

A State's Value

The state-to-value mapping described in this section is only one of many possible mappings for our very specific toy problem. Every problem will have many mappings, and the design of the state-to-value mapping is something that must be carefully considered.

In our toy problem, there are two factors that give hints towards the design of a state-to-value mapping. The first is that the obstacles are circles with known radii. The second is that in the toy world, a player is "allowed" to drive the vehicle inside an obstacle, but such a maneuver is discouraged. (See Figure 2.) These two factors can be handled by a sigmoid function, as shown in Equation (23).

Assume there are N circular obstacles, each with radius r_i and with center (x_i, y_i), where $i \in [1..N]$. Then, consider only the position part, (x, y), of a state $\boldsymbol{x}$. The distance between the state $\boldsymbol{x}$ and the ith obstacle is $d_i(\boldsymbol{x})$.

$$d_i(\boldsymbol{x}) = \sqrt{(x - x_i)^2 + (y - y_i)^2}$$

If $d_i(\boldsymbol{x}) > r_i$, then the state $\boldsymbol{x}$ is outside of the ith obstacle.

Define the function $s(\boldsymbol{x}, i)$ as follows.

$$s(\boldsymbol{x}, i) = \frac{1}{1 + \exp\Big(-\beta(d_i(\boldsymbol{x}) - r_i)\Big)} \tag{23}$$

The function $s(\boldsymbol{x}, i)$ has a sigmoidal shape. A state that is outside of the ith obstacle will have $d_i > r_i$ and $s(\boldsymbol{x}, i)$ will be approximately one. A state that is inside the ith obstacle will have $d_i < r_i$ and $s(\boldsymbol{x}, i)$ will be approximately zero. The parameter β controls the steepness of the transition between the outside region and the inside region. One minor but nice property of this sigmoidal shape is that the gradient always points away from the center of an obstacle. Figure 10 shows the values of all states (x, y) with respect to an obstacle centered at $(x_i, y_i) = (0, 0)$ with radius $r_i = 2$.

The function $s(\boldsymbol{x}, i)$ is the value of the state $\boldsymbol{x}$ with respect to the single obstacle i. In order to find the overall value of the state $\boldsymbol{x}$ with respect to the world, define $s(\boldsymbol{x})$ as the minimum of all N of the functions $s(\boldsymbol{x}, i)$.

$$s(\boldsymbol{x}) = \min_i s(\boldsymbol{x}, i) \tag{24}$$

SD Model Probabilities

The function $s(x)$ gives the value of every state x. This information can be incorporated into the model probabilities' update step, Equation (19), with the assumption that an intelligent target will want to maneuver toward high-valued states.

The IMM algorithm has M modes, each of which runs a separate Kalman filter. Suppose the jth mode's state estimate at time $k+1$ is $\hat{x}_{j,k+1}$. The IMM algorithm would calculate the mode probabilities and then mix together the M estimates weighted by those probabilities in order to obtain an overall state estimate. The procedure does not change when the states' value information is incorporated. However, the mode probabilities are updated as follows.

$$\mu^*_{j,k+1} = \frac{\mu^p_{j,k+1} \lambda_{j,k+1} s_{j,k+1}}{\sum_{i=1}^{M} \mu^p_{i,k+1} \lambda_{i,k+1} s_{i,k+1}} \tag{25}$$

For notational convenience, let $s_{j,k+1} = s(\hat{x}_{j,k+1})$.

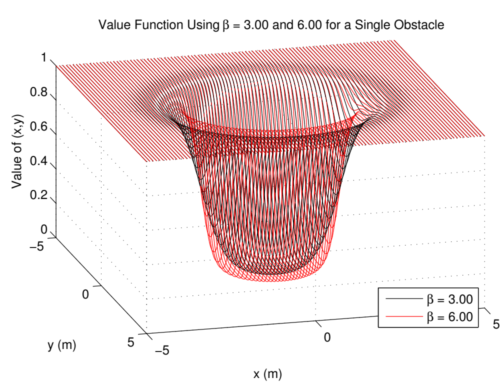


Figure 10: The function $s(\boldsymbol{x}, i)$, shown in Equation (23), for an obstalce with $r = 2$ and two values of β.

SD Transition Probabilities

The function $s(x)$ gives a value to every state x. The states' value information can be embedded into the transition probability matrix under the assumption that an intelligent target will want to maneuver toward high-valued states.

The first step of each iteration of the IMM algorithm is a mixing step in which mode probabilities are predicted based on the system's transition probability matrix. At the end of the previous step, the IMM algorithm had M state estimates $\hat{x}_{i,k|k}$ that were updated by using the measurement z_k. To incorporate the world information, the the IMM algorithm will use the transition probability matrix T_k^* to predict the model probabilities at time $k+1$, where T_k^* is a modification of the original T_k shown in Equation (10).

The mixing step in the IMM algorithm mixes together mode estimates. It considers the possibility that mode i was in effect previously when mode j is in effect currently. Embedding the states' value information into the transition probability matrix relies on those "what-if" state estimates. Define the state $\hat{x}_{j,k+1|i,k}$ as follows.

$$\hat{x}_{j,k+1|i,k} = A_{j,k}\hat{x}_{i,k|k} \tag{26}$$

The state $\hat{x}_{j,k+1|i,k}$ is a prediction of the state of the target that would arise if mode i were in effect at time k but mode j is used to propagate the state to time $k+1$.

There are M^2 states $\hat{x}_{j,k+1|i,k}$, one for each possible transition. The value of each state is $s_{ji,k}$.

$$s_{ji,k} = s(\hat{x}_{j,k+1|i,k}) \tag{27}$$

These states' values are merged together with the original transition probability matrix T_k to obtain T_k^*. T_k is the matrix of elements $[T_{ji,k}]$. Suppose that $[s_{ji,k}]$ is a matrix that contains all of the values of $s_{ji,k}$.

$$T_k^* = \text{col\_norm}\big(\ [s_{ji,k}]\ .^*\ [T_{ji,k}]\ \big) \tag{28}$$

The "dot-star" $.^*$ operation means element by element multiplication and in this case results in a matrix. Each of the transition probability matrix's *columns* must sum to one, thus the col_norm function is applied to the resulting matrix.

The matrix T_k^* is obtained before Equations (10) and (11). Those two equations are modified to use T_k^* in place of T_k. The remainder of the IMM algorithm is unchanged.

Small Visual Example

Figure 11 is a small visual example that shows a change in the estimated state due to the presence of an obstacle. Figure 11a shows the initial location and orientation of the target, specified by the **blue circle** and **blue dot**, as well as the first two measurements that will arrive, shown as **red circles**. There is no ground truth for this case, as the purpose of this example is not to track the target. Actual tracking will be examined in more detail in **Experiment and Results**.

Figure 11b shows five individual modes' estimates after the first measurement has arrived. The **blue squares** correspond to the normal IMM algorithm's estimates while the **green X's** correspond to SD TPM, the IMM algorithm with world information incorporated into the transition probability matrix. The normal TPM is fixed and has large diagonal elements, while the SD TPM modifies the normal TPM at every time step. The initial location of the target and the first measurement are not close to any obstacles, and thus the normal and SD TPM modes' estimates match exactly. Figure 11c shows that the overall normal and SD TPM state estimates also coincide. The normal estimate is shown as a **heavy blue square**, and the SD TPM estimate is shown as a **heavy green X**.

Figure 11d takes place during the second time step. The blue squares and green X's have the same meaning as before, except now they are calculated using the second measurement. Notice that the normal estimates and the SD TPM estimates are different now due to the proximity of an obstacle. Figure 11e shows the overall normal estimate as a heavy blue square and the overall SD TPM estimate as a heavy green X. The SD TPM estimate lies just outside the obstacle.

The measurement noise is assumed to be relatively small in this example, and thus the IMM algorithm's estimates hug the measurements. Incorporating the world information makes a difference despite the small measurement noise.

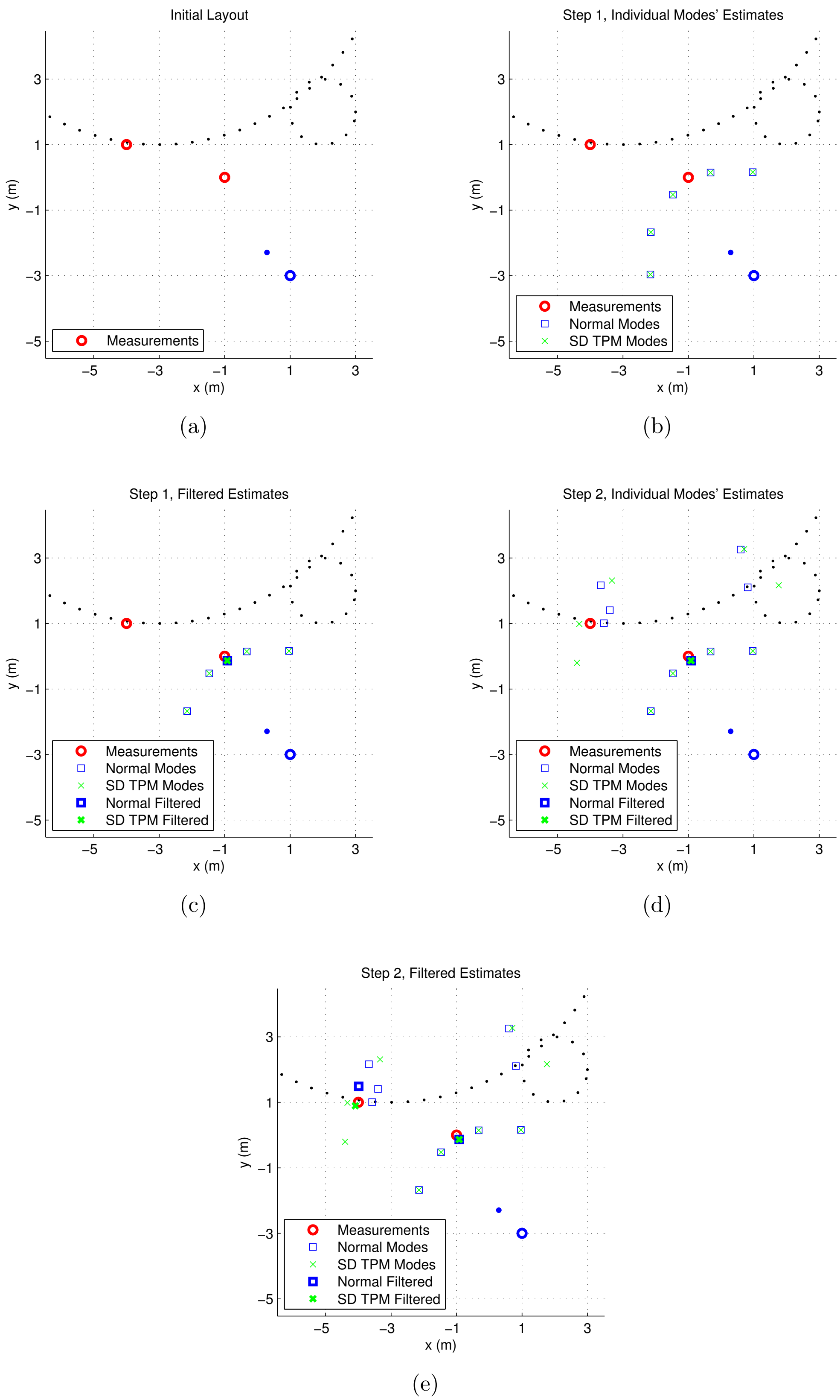


Figure 11: Refer to **Small Visual Example** on Page 21 for explanations.

Experiment and Results

The section titled **Toy Problem** describes how ground truth trajectories are created. Samples come from the position part of the ground truth every $T = 0.35$ seconds, and these samples are used as the basis of the tracking experiment. The purpose of the experiment is to compare the performance of the IMM algorithm under the normal case with no state-dependent features to the IMM algorithm with state-dependent modifications.

Experimental Design

The first step of designing the experiment is to design the IMM algorithm's model set. This model set should capture the possible maneuvers that the target can take while simultaneously being as simple as possible. In order to simplify the design, we chose sections of ground truth in which the speed of the target is constant, thus avoiding the need to model linear accelerations. This simplification would be an acceptable assumption in a real game, because generally a skilled player would maneuver without slowing down in order to gain the maximum number of points.

We chose to track the position and the velocity of the target. The state variable is $\boldsymbol{x}$. The variables x and y represent coordinates.

$$\boldsymbol{x} = [x, y, \dot{x}, \dot{y}]' \tag{29}$$

The model set that we chose consists of five constant turn models with varying turn rates. Two left-turn models have $\omega_1 = 1.42 \times 2\pi$ rad/s and $\omega_2 = 0.71 \times 2\pi$ rad/s. Two right-turn models have $\omega_4 = -\omega_2$ and $\omega_5 = -\omega_1$. A final model has $\omega_3 = 0$, which corresponds to going straight. The dynamics matrix of a constant turn model, adapted from [5], is as follows.

$$A(\omega, T) = \begin{pmatrix} 1 & 0 & \dfrac{\sin(\omega T)}{\omega} & \dfrac{\cos(\omega T) - 1}{\omega} \\ 0 & 1 & \dfrac{1 - \cos(\omega T)}{\omega} & \dfrac{\sin(\omega T)}{\omega} \\ 0 & 0 & \cos(\omega T) & -\sin(\omega T) \\ 0 & 0 & \sin(\omega T) & \cos(\omega T) \end{pmatrix} \tag{30}$$

Figure 12 shows the possible combinations of the five models after two time steps. The green circles represent the end position, and each black line shows the trajectory that the target would have taken to get to an endpoint. Even though the mode sequences $[\omega_2, \omega_2]$ and $[\omega_1, \omega_5]$ have the same endpoint, the resulting orientations are different.

The second step of the design of the experiment is to design various filters. We implemented and tested four variations of the IMM algorithm, all of which use the same model set. The first two variations have a constant transition probability matrix with large diagonal elements. The first variation is the standard IMM algorithm with no state-dependent features. This case is called "Normal." The second algorithm is the IMM algorithm with the world information embedded into the model probabilities, as described in the section **SD Model Probabilities**. This case is called "SD MPs."

The third and fourth variations have a new transition probability matrix at every time step. The third algorithm is the IMM algorithm with the world information embedded into the transition probability matrix, as described in the section **SD Transition Probabilities**. This case is called "SD TPM." The fourth variation is the IMM algorithm with world information embedded into both the model probabilities and the transition probability matrix. This case is called "SD Both."

All three of the state-dependent variations use the value function $s(\boldsymbol{x})$ described in Equation (24). The function $s(\boldsymbol{x})$ knows the locations and sizes of the obstacles in the toy world.

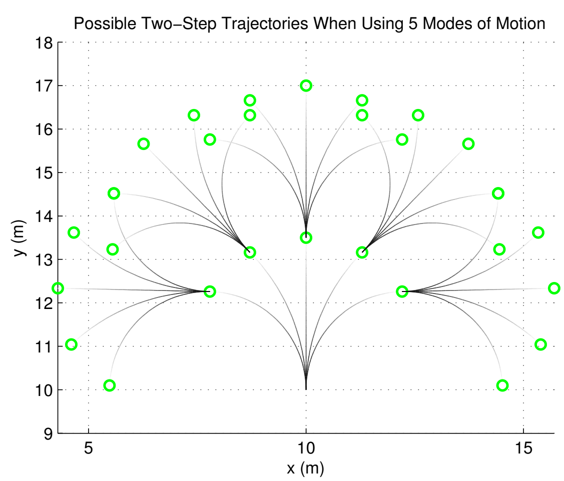


Figure 12: Possible sequences over two time steps. The initial state is $[x, y, \dot{x}, \dot{y}]' = [10, 10, 0, 10]'$. Each time step is $T = 0.35$ seconds long. The green circles indicate a possible position (x, y) at each time step.

The final step of the design of the experiment is to prepare true trajectory samples with corresponding noisy measurements. Figures 18 to 29 show the snippets of true trajectories that were chosen. Each green circle in those figures represents the truth at a particular time step. Those serve as the reference samples and are corrupted by noise of varying degree to obtain the measurements.

The measurement model used in the IMM algorithms matches the mechanism by which the measurements are created. The measurement model, shown in Equation (31), uses position-only measurements and has additive Gaussian noise v_{k+1}.

$$
\begin{aligned}
z_{k+1} &= H_{k+1}x_{k+1} + v_{k+1} \\
H_{k+1} &= H = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}
\end{aligned} \tag{31}
$$

The noise v_{k+1} has a parameter σ_z^2, as shown in Equation (32).

$$
\begin{aligned}
v_{k+1} &\sim \mathcal{N}(0, R_{k+1}) \\
R_{k+1} &= R = \begin{pmatrix} \sigma_z^2 & 0 \\ 0 & \sigma_z^2 \end{pmatrix}
\end{aligned} \tag{32}
$$

The experiment runs all of the filters for each value of σ_z^2 in order to see if a varying noise level affects the filters in different ways. For example, Figure 18 has within it Figure 18a to Figure 18g, one set of results for each noise level.

Results

Four variations of the IMM algorithm try to estimate and predict the motion of the target. There are several ground truth trajectories, one for each of Figures 18 to 29.

Suppose one of the ground truth trajectories has $N + 1$ true samples, samples that are not corrupted by noise. The first sample is used as the initialization of the algorithms. (The first sample is the sample, green circle, that is "behind" the blue circle.) The remaining N samples are corrupted by additive Gaussian noise with a specific σ_z^2.

The measurements z_1 to z_N have corresponding true states x_1 to x_N. The filters use those measurements to obtain state estimates $\hat{x}_{1|1}$, $\hat{x}_{2|2}$, ..., $\hat{x}_{N|N}$. The difference between the truth and the estimate is $\tilde{x}_{k|k}$.

$$\tilde{x}_{k|k} = x_k - \hat{x}_{k|k} \tag{33}$$

The filters also use those measurements to obtain state predictions $\hat{x}_{1|0}$, $\hat{x}_{2|1}$, ..., $\hat{x}_{N|N-1}$. The difference between the truth and the prediction is $\tilde{x}_{k|k-1}$.

$$\tilde{x}_{k|k-1} = x_k - \hat{x}_{k|k-1} \tag{34}$$

The estimation error, e_e, is defined as the average distance between the true position and the estimate of the position.

$$e_e = \frac{1}{N} \sum_{k=1}^{N} \sqrt{(H\hat{x}_{k|k})'(H\hat{x}_{k|k})} \tag{35}$$

The prediction error, e_p, is defined as the average distance between the true position and the prediction of the position.

$$e_p = \frac{1}{N} \sum_{k=1}^{N} \sqrt{(H\hat{x}_{k|k-1})'(H\hat{x}_{k|k-1})} \tag{36}$$

Note that both e_e and e_p are scalars.

There is one value of e_e per trial for each of the four algorithms in the experiment, plus there is one value of e_e for the measurements themselves for each trial. Similarly, there is one value of e_p per trial for each of the four algorithms. Conversely, there is no e_p for the measurements. Figures 18a to 29g show the average results, averages of e_e and e_p called $\bar{e}_e$ and $\bar{e}_p$, over 300 trials for each value of σ_z^2.

The value of $\bar{e}_e$ that corresponds to the measurements is related to the noise level σ_z^2. Since the variable $\tilde{x}_{k|k}$ is a zero-mean Gaussian random variable with covariance matrix $R = \text{diag}(\sigma_z^2, \sigma_z^2)$, the magnitude of $\tilde{x}_{k|k}$ is a Rayleigh distributed random variable with mean $\mu = \bar{e}_e$.

$$\mu = \sqrt{\frac{\pi}{2}\sigma_z^2} \tag{37}$$

This relationship gives a relative scale to the estimation and prediction errors.

In Figures 18a to 29g, the measurements' estimation error is always bolded because it is the baseline reference, and there is at most one other value in each column that has been bolded. The other bolded value corresponds to the filter that had the lowest error, on average, for a specific level of σ_z^2.

Discussion

Figures 18a to 29g show the results of the experiment described in **Experiment and Results**. There are *twelve* ground truth cases, each of which contains cases for *seven* different measurement noise levels. Thus, the experiment contains a total of *eighty-four* parameter combinations. One purpose of this section is to try to identify patterns in the results, and to that end two major questions need to be addressed.

1. Does knowledge of the environment actually make a difference in tracking performance?

2. Is there a difference between implementing the world information in the model probabilities as opposed to the transition probability matrix?

Figure 13 provides some counts that might help to answer those questions.

	Normal	SD TPM	SD MPs	SD Both
Normal	84	4	2	3
SD TPM	73	84	70	4
SD MPs	52	7	84	5
SD Both	74	48	72	84

(a) Estimation

	Normal	SD TPM	SD MPs	SD Both
Normal	84	1	0	1
SD TPM	76	84	21	10
SD MPs	77	56	84	41
SD Both	76	67	36	84

(b) Prediction

Figure 13: Performance Summaries. The entry in the ith row and jth column shows how many times algorithm i was better than algorithm j. Refer to **State-Dependent Performance** for details of how the values are obtained.

State-Dependent Performance

The experiment contains four variations of the IMM algorithm. In addition to calling them "Normal," "SD TPM," "SD MPs," and "SD Both," we can refer to them as the first, second, third, or fourth algorithm, respectively. This is the order in which the algorithms were implemented as well as the order in which the results are displayed. Figure 13 counts how many times algorithm i was better than algorithm j over the *eighty-four* cases. Consider Figure 18a, reproduced here for convenience.

	Estimation	Prediction
Measurements	**0.08873**	-
Normal	0.08827	1.57677
SD TPM	0.08811	1.43514
SD MPs	0.08826	**1.40860**
SD Both	**0.08811**	1.41925

The Estimation column shows the values of $\bar{e}_{e,i}$, and the Prediction column shows the values of $\bar{e}_{p,i}$ for the case of 18a. (Add an algorithm subscript i to Equations (35) and (36).) For every pair $i, j \in [1..4]^2$, add one to the i, jth entry of Figure 13a if $\bar{e}_{e,i}$ is less than $\bar{e}_{e,j}$. Also add one to the i, jth entry if i is equal to j. Note that $\bar{e}_{e,i}$ is never less than itself. This process is performed over all eighty-four cases to obtain Figure 13a. The process is repeated again using $\bar{e}_{p,i}$ to obtain Figure 13b.

Figure 13 shows that the three state-dependent variations of the IMM algorithm very frequently perform better, on average, than the "Normal" algorithm both for estimation and for prediction. This can be seen in two ways. In the first column of Figures 13a and 13b, the entries that correspond to the state-dependent algorithms are very high, with the exception of "SD MPs" in the Estimation case, meaning that the state-dependent variations frequently perform better than the "Normal" case. A similar conclusion comes from the first row of each of the two figures; the "Normal" case is almost never better than the state-dependent cases. In Estimation, "SD MPs" is better than "Normal" 52 times, "Normal" is better than "SD MPs" 2 times, and the remaining 30 times the two were equal (within five decimal places). Seven of the equivalent performances come from Figure 25 because there are no obstacles.

It is important to note that Figure 13 does not give an indication of how much better the state-dependent algorithms performed. It merely counts how many times the state-dependent algorithms performed better by any amount. In Figure 18a, for example, all of the estimation errors $\bar{e}_{e,i}$ are the same up to the thousandths place. When the measurement noise σ_z^2 is very low, there is not much room for improvement. Even in 18g, the highest σ_z^2 of Figure 18, the estimation performance of the state-dependent algorithms is not much better than that of the "Normal" algorithm.

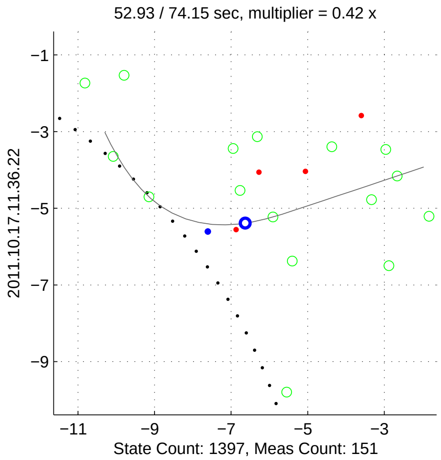
Examining the actual numbers in Figures 18a to 29g shows that the state-dependent predictors perform better than the "Normal" predictor by a qualitatively discernible amount. Further evidence is given by Figure 14 and Figure 15 in which the state-dependent algorithms predict correctly that the target will maneuver to avoid the obstacles.

State-Dependent Comparison

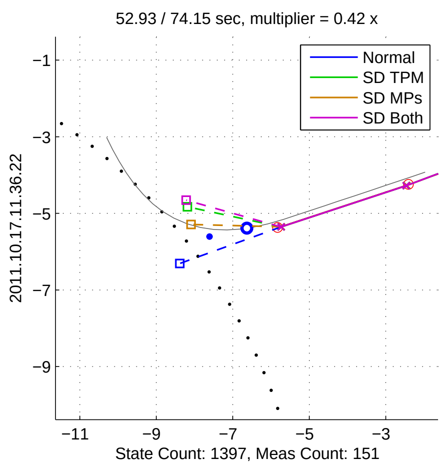
According to Figure 13a and Figure 13b, there are differences between using "SD MPs," "SD TPM," and "SD Both." The effects of the differences can be seen by carefully examining Figure 13, but the causes of the differences are not clear yet. The relative differences between the variations depend as much on the choice of the state-to-value mapping as they do on the methods of incorporating that information.

Figure 13a shows that "SD TPM" performs better than "SD MPs" a majority of the time in estimation. Further, it shows that "SD Both" is at least as good as, if not better than, "SD TPM" in the majority of cases. Conversely, Figure 13b shows that "SD MPs" often predicts better than "SD TPM." It also shows that "SD MPs" is roughly equivalent in performance to "SD Both." The combination algorithm, "SD Both," can take both the good parts and the bad parts of the individual variations "SD TPM" and "SD MPs," thus the challenge becomes to differentiate between "SD TPM" and "SD MPs."

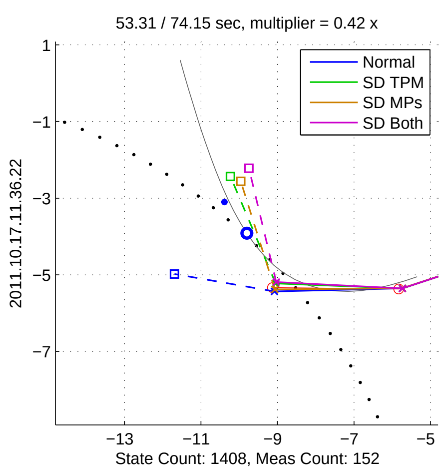
It seems that "SD TPM" is not as good as "SD MPs" for prediction, but it is better for estimation. Generally, the IMM algorithm at the $k+1$th time step has the ability to lessen the strength of the kth measurement because of the mixing step, as shown in Equations (11) to (13). The "SD TPM" algorithm has even more of that power because of the fact that the transition probability matrix can be very heavily modified, as in Equation (28), before the mixing step takes place. I believe this causes the predictions of the "SD TPM" algorithm to be more wild than the predictions of the "SD MPs" algorithm, evidenced by Figure 16 and Figure 17. Even though the predictions of "SD TPM" are wild, they appear to be good mixing candidates once a new measurement arrives to tame them.



(a) Window of ground truth around $k = 151$.

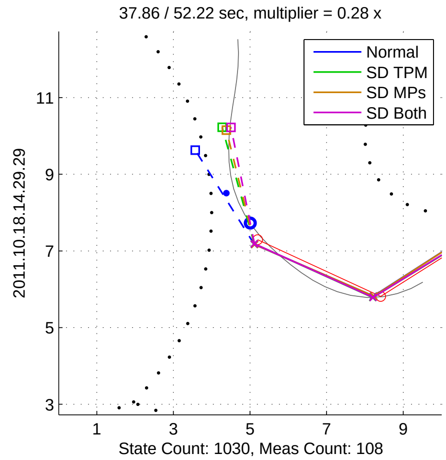


(b) Four predicted states $\hat{x}_{152|151}$.

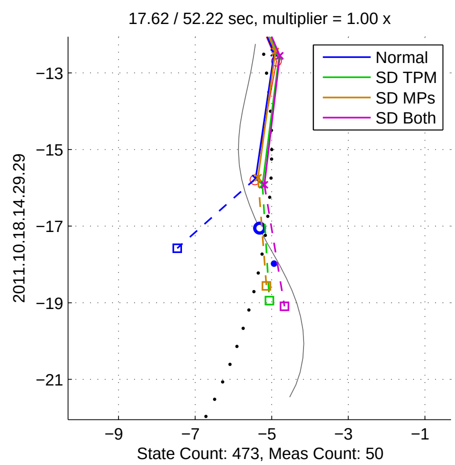


(c) Four predicted states $\hat{x}_{153|152}$.

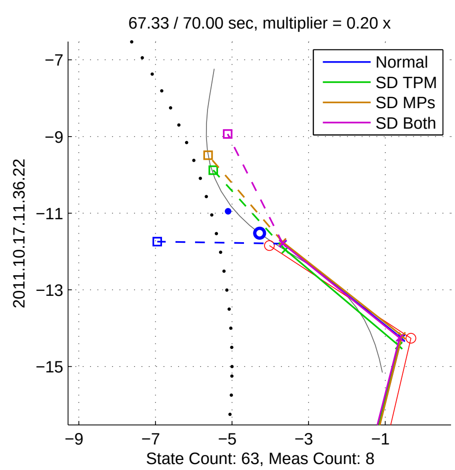
Figure 14: Truth and Predictions.



(a)



(b)



(c)

Figure 15: SD cases with good behavior.

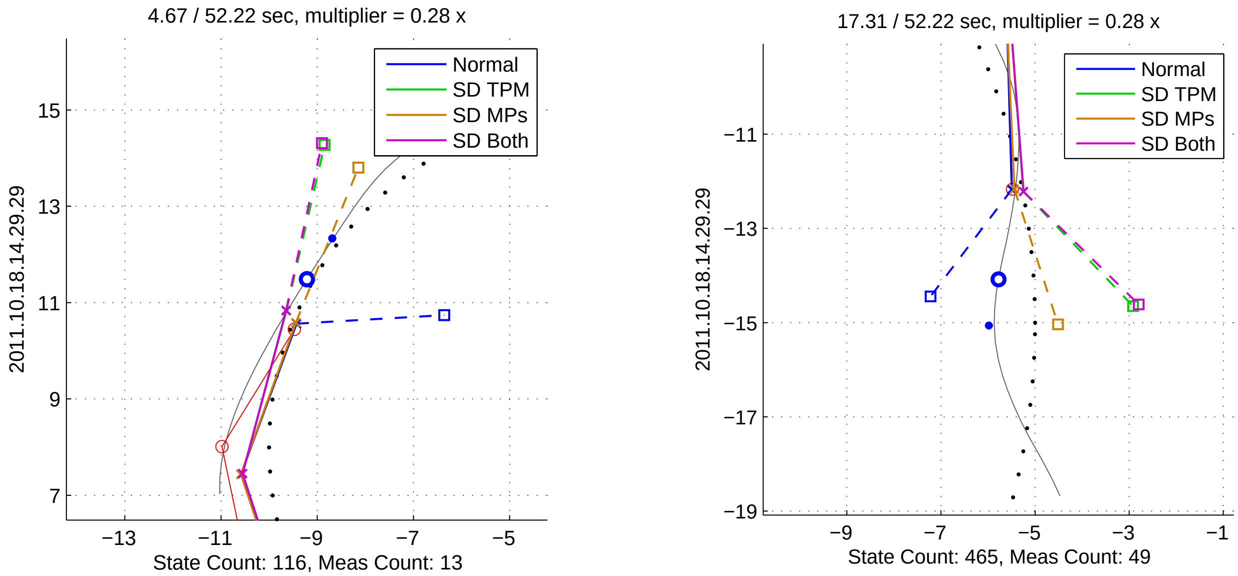
In particular cases, the aforementioned general tendencies do not occur. In the case of Figure 19, “SD TPM” is often better than “SD MPs” for both estimation and prediction. Alternatively, Figure 23 shows that the best estimator depends on the noise level. It is not clear whether these differences are inherent properties of the world information embedding methods or if they are more due to the specific choice of state-to-value mapping.

The Value Function

There are certain valid maneuvers in this specific world that the specific choice of value function deems as very improbable. Figure 17 is an example of a case in which the target continues to maneuver inside of an obstacle’s boundary. It is clear that the state-dependent algorithms behave correctly according to the choice of value function but incorrectly with respect to the actual rules governing the toy problem. This causes large estimation and prediction errors for all of the state-dependent algorithms.

The inaccuracy of the value function also might explain why the performance rankings of the algorithms vary so much in Figure 23. Determining whether the performance variations are due to the value function or to the method by which the world information is incorporated requires more research.

Regardless, a better value function would improve the tracking performance for this specific player in this specific toy problem. For example, Figure 16a shows that the “SD MPs” prediction stays close to the boundary of the obstacle while the “SD TPM” and “SD Both” predictions push away from the edge. A slightly modified value function could give more value to the area that hugs the boundaries, because this specific player (the author) has a tendency to follow the walls while playing.



(a) (b)

Figure 16: The combination of the value function, the transition probability matrix, and the methods of embedding the world information sometimes allows SD MPs to perform better than SD TPM.

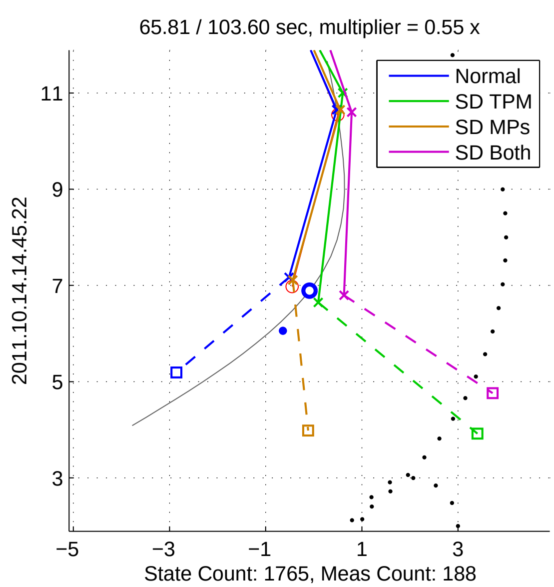


Figure 17: This case shows that the value function $s(\boldsymbol{x})$ from Equations (23) and (24) is not entirely accurate for the toy problem, because, though unlikely, the target can maneuver inside an obstacle region.

Conclusion

All three of the **Goals** have been accomplished. The target is controlled by a human player in real time in order to generate the ground truth trajectories for the simulations. The player generally avoids certain obstacle regions in the world, and this world information is incorporated into the IMM algorithm. Knowledge of the world information allows state-dependent variations of the IMM algorithm to estimate and predict the motion of the target better than the normal version.

There are three state-dependent variations of the IMM algorithm. State-dependent model probabilities, "**SD MPs**," embeds the world information into the final remixing step of the IMM algorithm. State-dependent transition probability matrix, "**SD TPM**," incorporates the world information into the transition probability matrix before the first mixing step of the IMM algorithm. A combination, "SD Both," modifies both the model probabilities and the transition probability matrix.

Based on the set of ground truth trajectories and the simulation results, "SD TPM" seems to perform **better** than "SD MPs" for estimation but **worse** for prediction. The combination, "SD Both," often performs at least as well as the other two. All three of the variations almost always perform **better** than the "Normal" algorithm. The estimation performance increase is not always significant, but the prediction performance is almost always qualitatively better.

The value function presented in **A State's Value** is relatively simple and does not accurately capture the states' values from the point of view of the specific player in this specific toy world. A better value function certainly would improve the tracking performance of the state-dependent IMM algorithms. Regardless, the simple value function is enough to improve the performances of the state-dependent algorithms when compared to the normal algorithm.

The value function must be tailored to a specific problem, but the method of incorporating the world information into the IMM algorithm is generally applicable. Specific performance details depend on both the problem and the choice of value function.

References

[1] Y. Bar-Shalom, X. R. Li, and K. C. Chang, 1990.
Nonstationary noise identification with the interacting multiple model algorithm. *Proceedings of the 5th IEEE International Symposium on Intelligent Control*, vol. 1, pp. 585–589

[2] R. E. Kalman, 1960.
A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, vol. 82(Series D): pp. 35–45

[3] X. R. Li and Y. Bar-Shalom, 1992.
A recursive hybrid system approach to noise identification. *First IEEE Conference on Control Applications, 1992.*, vol. 2, pp. 847–852

[4] X. R. Li and Y. Bar-Shalom, 1993.
Performance prediction of the interacting multiple model algorithm. *IEEE Transactions on Aerospace and Electronic Systems*, vol. 29(3): pp. 755–771

[5] X. R. Li and Y. Bar-Shalom, 1993.
Design of an interacting multiple model algorithm for air traffic control tracking. *IEEE Transactions on Control Systems Technology*, vol. 1(3): pp. 186–194

[6] R. Toledo-Moreo and M. A. Zamora-Izquierdo, 2009.
Imm-based lane-change prediction in highways with low-cost gps/ins. *IEEE Transactions on Intelligent Transportation Systems*, vol. 10(1): pp. 180–185

[7] X. R. Li and Y. Bar-Shalom, 1996.
Multiple-model estimation with variable structure. *IEEE Transactions on Automatic Control*, vol. 41(4): pp. 478–493

[8] X. R. Li, Y. Zhang, and X. Zhi, 1997.
Multiple-model estimation with variable structure: model-group switching algorithm. *Proceedings of the 36th IEEE Conference on Decision and Control, 1997.*, vol. 4, pp. 3114–3119

[9] X. R. Li, X. Zwi, and Y. Zwang, 1999.
Multiple-model estimation with variable structure. iii. model-group switching algorithm. *IEEE Transactions on Aerospace and Electronic Systems*, vol. 35(1): pp. 225–241

[10] X. R. Li, 2000.
Multiple-model estimation with variable structure. ii. model-set adaptation. *IEEE Transactions on Automatic Control*, vol. 45(11): pp. 2047–2060

[11] X. Wang, S. Challa, R. Evans, and X. R. Li, 2003.
Minimal submodel-set algorithm for maneuvering target tracking. *IEEE Transactions on Aerospace and Electronic Systems*, vol. 39(4): pp. 1218–1231

[12] T. Kirubarajan, Y. Bar-Shalom, K. R. Pattipati, and I. Kadar, 2000.
Ground target tracking with variable structure imm estimator. *IEEE Transactions on Aerospace and Electronic Systems*, vol. 36(1): pp. 26–46

[13] M. Zhang, S. Knedlik, and O. Loffeld, 2008.
An adaptive road-constrained imm estimator for ground target tracking in gsm networks. *2008 11th International Conference on Information Fusion*, pp. 1–8

[14] I. Hwang and C. E. Seah, 2006.
An estimation algorithm for stochastic linear hybrid systems with continuous-state-dependent mode transitions. *2006 45th IEEE Conference on Decision and Control*, pp. 131–136

[15] C. E. Seah and I. Hwang, 2009.
State estimation for stochastic linear hybrid systems with continuous-state-dependent transitions: An imm approach. *IEEE Transactions on Aerospace and Electronic Systems*, vol. 45(1): pp. 376–392

[16] S. Zhang and Y. Bar-Shalom, 2011.
Tracking move-stop-move targets with state-dependent mode transition probabilities. *IEEE Transactions on Aerospace and Electronic Systems*, vol. 47(3): pp. 2037–2054

[17] D. J. Kershaw, 1999.
Issues in single target tracking with state dependent detection probability. *Proceedings of IDC 99 on Information, Decision and Control*, pp. 99–104

[18] V. P. Jilkov and X. R. Li, 2004.
Online bayesian estimation of transition probabilities for markovian jump systems. *IEEE Transactions on Signal Processing*, vol. 52(6): pp. 1620–1630

[19] Z. Ding and H. Leung, 2008.
Evaluation of two imm-based algorithms in real radar tracking environments. *2008 Canadian Conference on Electrical and Computer Engineering*, pp. 1569–1574

[20] X. R. Li and V. P. Jilkov, 2010.
Survey of maneuvering target tracking. part ii: Motion models of ballistic and space targets. *IEEE Transactions on Aerospace and Electronic Systems*, vol. 46(1): pp. 96–119

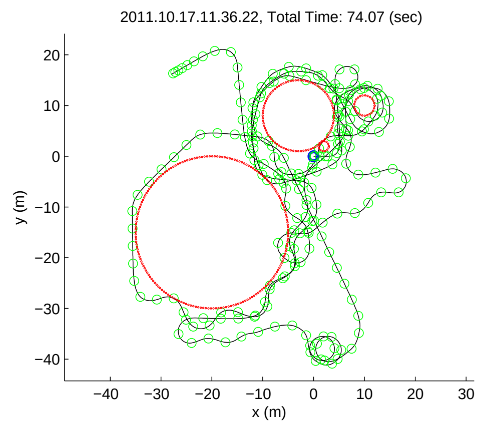
[21] Y. Liu and X. R. Li, 2011.
Sequential multiple-model detection of target maneuver termination. *2011 Proceedings of the 14th International Conference on Information Fusion (FUSION)*, pp. 1–8

[22] D. Gruyer and E. Pollard, 2011.
Credibilistic imm likelihood updating applied to outdoor vehicle robust ego-localization. *2011 Proceedings of the 14th International Conference on Information Fusion (FUSION)*, pp. 1–8

[23] D. Gu, 2011.
A game theory approach to target tracking in sensor networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 41(1): pp. 2–13

Appendices

Simulation Results



	Estimation	Prediction
Measurements	**0.08873**	-
Normal	0.08827	1.57677
SD TPM	0.08811	1.43514
SD MPs	0.08826	**1.40860**
SD Both	**0.08811**	1.41925

(a) Averages of 300 Runs, $\sigma_z^2 = 0.005$

	Estimation	Prediction
Measurements	**0.12572**	-
Normal	0.12439	1.61050
SD TPM	0.12407	1.46384
SD MPs	0.12438	**1.44233**
SD Both	**0.12407**	1.44316

(b) Averages of 300 Runs, $\sigma_z^2 = 0.010$

	Estimation	Prediction
Measurements	**0.19813**	-
Normal	0.19462	1.69597
SD TPM	0.19352	1.53771
SD MPs	0.19451	1.52323
SD Both	**0.19347**	**1.51042**

(c) Averages of 300 Runs, $\sigma_z^2 = 0.025$

	Estimation	Prediction
Measurements	**0.27937**	-
Normal	0.27229	1.78459
SD TPM	0.27028	1.62484
SD MPs	0.27196	1.61392
SD Both	**0.27012**	**1.59280**

(d) Averages of 300 Runs, $\sigma_z^2 = 0.050$

	Estimation	Prediction
Measurements	**0.39631**	-
Normal	0.38525	1.93268
SD TPM	0.38058	1.77883
SD MPs	0.38448	1.76810
SD Both	**0.38006**	**1.74156**

(e) Averages of 300 Runs, $\sigma_z^2 = 0.100$

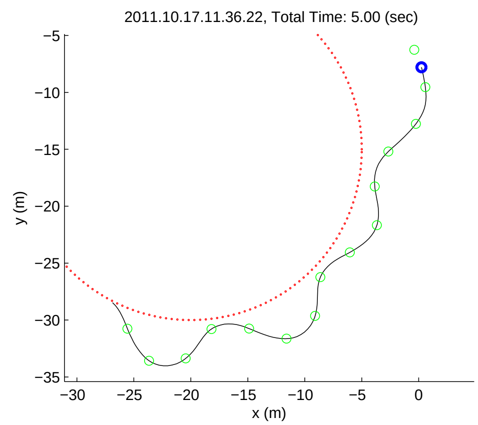
	Estimation	Prediction
Measurements	**0.62607**	-
Normal	0.60578	2.20458
SD TPM	0.59379	2.07286
SD MPs	0.60353	2.04680
SD Both	**0.59263**	**2.02831**

(f) Averages of 300 Runs, $\sigma_z^2 = 0.250$

	Estimation	Prediction
Measurements	**0.88759**	-
Normal	0.85571	2.51311
SD TPM	0.83711	2.39872
SD MPs	0.84918	**2.34708**
SD Both	**0.83372**	2.34748

(g) Averages of 300 Runs, $\sigma_z^2 = 0.500$

Figure 18: c_2011.10.17.11.36.22_0.08_74.15 with 7 different measurement noise variances.



	Estimation	Prediction
Measurements	**0.08744**	-
Normal	0.08739	2.07265
SD TPM	0.08735	1.73294
SD MPs	0.08739	1.70343
SD Both	**0.08735**	**1.60858**

(a) Averages of 300 Runs, $\sigma_z^2 = 0.005$

	Estimation	Prediction
Measurements	**0.12314**	-
Normal	0.12298	2.10733
SD TPM	0.12269	1.75524
SD MPs	0.12298	1.74120
SD Both	**0.12269**	**1.63091**

(b) Averages of 300 Runs, $\sigma_z^2 = 0.010$

	Estimation	Prediction
Measurements	**0.19788**	-
Normal	0.19664	2.18962
SD TPM	0.19556	1.82160
SD MPs	0.19664	1.85959
SD Both	**0.19556**	**1.68555**

(c) Averages of 300 Runs, $\sigma_z^2 = 0.025$

	Estimation	Prediction
Measurements	**0.28012**	-
Normal	0.27856	2.25738
SD TPM	0.27500	1.90231
SD MPs	0.27856	1.96382
SD Both	**0.27500**	**1.76260**

(d) Averages of 300 Runs, $\sigma_z^2 = 0.050$

	Estimation	Prediction
Measurements	**0.39782**	-
Normal	0.38691	2.35647
SD TPM	0.37949	2.03984
SD MPs	0.38689	2.04847
SD Both	**0.37949**	**1.88700**

(e) Averages of 300 Runs, $\sigma_z^2 = 0.100$

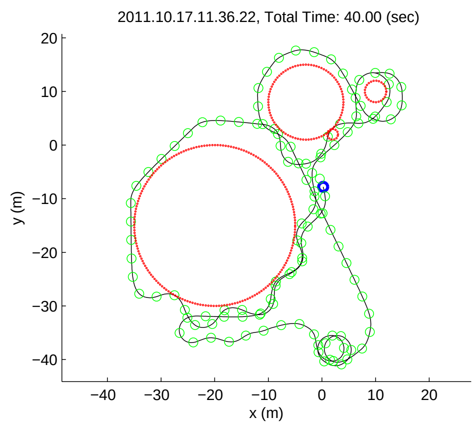
	Estimation	Prediction
Measurements	**0.63475**	-
Normal	0.60169	2.63397
SD TPM	0.58653	2.31658
SD MPs	0.59944	2.24449
SD Both	**0.58583**	**2.13093**

(f) Averages of 300 Runs, $\sigma_z^2 = 0.250$

	Estimation	Prediction
Measurements	**0.87034**	-
Normal	0.79279	2.89796
SD TPM	0.78164	2.63250
SD MPs	0.78783	2.47478
SD Both	**0.77894**	**2.42925**

(g) Averages of 300 Runs, $\sigma_z^2 = 0.500$

Figure 19: c_2011.10.17.11.36.22_10.00_15.00 with 7 different measurement noise variances.



	Estimation	Prediction
Measurements	**0.08888**	-
Normal	0.08805	1.48972
SD TPM	**0.08791**	1.32325
SD MPs	0.08805	1.29526
SD Both	0.08791	**1.29497**

(a) Averages of 300 Runs, $\sigma_z^2 = 0.005$

	Estimation	Prediction
Measurements	**0.12499**	-
Normal	0.12294	1.53558
SD TPM	**0.12252**	1.36822
SD MPs	0.12294	1.34206
SD Both	0.12252	**1.33378**

(b) Averages of 300 Runs, $\sigma_z^2 = 0.010$

	Estimation	Prediction
Measurements	**0.19851**	-
Normal	0.19204	1.63938
SD TPM	0.19102	1.45671
SD MPs	0.19199	1.44554
SD Both	**0.19100**	**1.41300**

(c) Averages of 300 Runs, $\sigma_z^2 = 0.025$

	Estimation	Prediction
Measurements	**0.27902**	-
Normal	0.26530	1.74427
SD TPM	0.26295	1.55800
SD MPs	0.26518	1.54498
SD Both	**0.26288**	**1.50963**

(d) Averages of 300 Runs, $\sigma_z^2 = 0.050$

	Estimation	Prediction
Measurements	**0.39611**	-
Normal	0.37121	1.90289
SD TPM	0.36718	1.72361
SD MPs	0.37091	1.70141
SD Both	**0.36696**	**1.66478**

(e) Averages of 300 Runs, $\sigma_z^2 = 0.100$

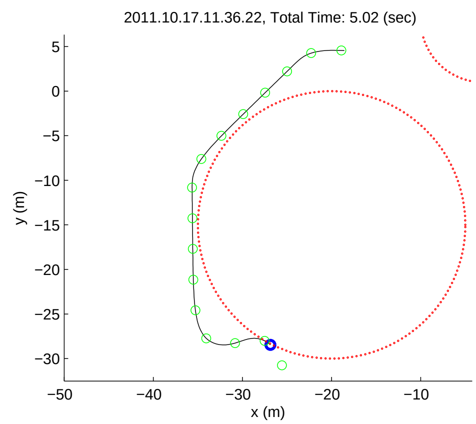
	Estimation	Prediction
Measurements	**0.62645**	-
Normal	0.57556	2.18277
SD TPM	0.56655	2.02787
SD MPs	0.57412	1.97888
SD Both	**0.56559**	**1.95632**

(f) Averages of 300 Runs, $\sigma_z^2 = 0.250$

	Estimation	Prediction
Measurements	**0.87879**	-
Normal	0.80174	2.47478
SD TPM	0.78328	2.33213
SD MPs	0.79624	2.26771
SD Both	**0.78080**	**2.25386**

(g) Averages of 300 Runs, $\sigma_z^2 = 0.500$

Figure 20: c_2011.10.17.11.36.22_10.00_50.00 with 7 different measurement noise variances.



	Estimation	Prediction
Measurements	**0.08849**	-
Normal	0.08764	1.23475
SD TPM	**0.08749**	**1.06268**
SD MPs	0.08765	1.07112
SD Both	0.08750	1.10986

(a) Averages of 300 Runs, $\sigma_z^2 = 0.005$

	Estimation	Prediction
Measurements	**0.12340**	-
Normal	0.12076	1.34194
SD TPM	**0.12065**	1.18893
SD MPs	0.12078	**1.12733**
SD Both	0.12069	1.22002

(b) Averages of 300 Runs, $\sigma_z^2 = 0.010$

	Estimation	Prediction
Measurements	**0.19802**	-
Normal	0.18926	1.50145
SD TPM	**0.18898**	1.36808
SD MPs	0.18916	**1.22674**
SD Both	0.18902	1.36886

(c) Averages of 300 Runs, $\sigma_z^2 = 0.025$

	Estimation	Prediction
Measurements	**0.28049**	-
Normal	0.26167	1.63642
SD TPM	**0.26120**	1.52253
SD MPs	0.26161	**1.32396**
SD Both	0.26124	1.49567

(d) Averages of 300 Runs, $\sigma_z^2 = 0.050$

	Estimation	Prediction
Measurements	**0.39677**	-
Normal	0.35939	1.76913
SD TPM	0.35802	1.65545
SD MPs	0.35760	**1.44934**
SD Both	**0.35739**	1.61093

(e) Averages of 300 Runs, $\sigma_z^2 = 0.100$

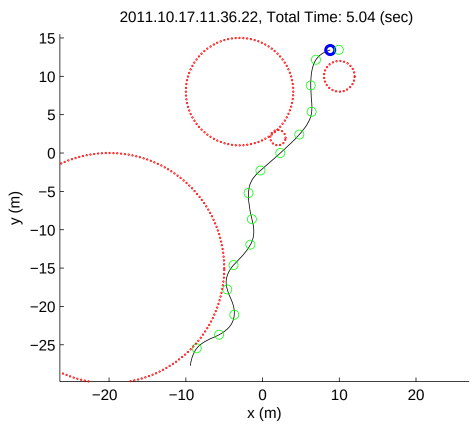
	Estimation	Prediction
Measurements	**0.63116**	-
Normal	0.56572	2.07629
SD TPM	0.55080	1.95732
SD MPs	0.55461	**1.74294**
SD Both	**0.54610**	1.87388

(f) Averages of 300 Runs, $\sigma_z^2 = 0.250$

	Estimation	Prediction
Measurements	**0.88564**	-
Normal	0.80242	2.45280
SD TPM	0.76064	2.25532
SD MPs	0.76922	**2.08731**
SD Both	**0.74513**	2.14933

(g) Averages of 300 Runs, $\sigma_z^2 = 0.500$

Figure 21: c_2011.10.17.11.36.22_15.00_20.00 with 7 different measurement noise variances.



	Estimation	Prediction
Measurements	**0.08832**	-
Normal	0.08762	1.85386
SD TPM	0.08728	**1.46961**
SD MPs	0.08762	1.56943
SD Both	**0.08728**	1.50644

(a) Averages of 300 Runs, $\sigma_z^2 = 0.005$

	Estimation	Prediction
Measurements	**0.12690**	-
Normal	0.12493	1.88246
SD TPM	0.12429	**1.51373**
SD MPs	0.12493	1.61263
SD Both	**0.12429**	1.53954

(b) Averages of 300 Runs, $\sigma_z^2 = 0.010$

	Estimation	Prediction
Measurements	**0.20066**	-
Normal	0.19623	1.97187
SD TPM	0.19312	**1.62654**
SD MPs	0.19622	1.71504
SD Both	**0.19310**	1.63932

(c) Averages of 300 Runs, $\sigma_z^2 = 0.025$

	Estimation	Prediction
Measurements	**0.27659**	-
Normal	0.26681	2.05942
SD TPM	0.26155	1.70887
SD MPs	0.26674	1.79647
SD Both	**0.26145**	**1.70155**

(d) Averages of 300 Runs, $\sigma_z^2 = 0.050$

	Estimation	Prediction
Measurements	**0.39701**	-
Normal	0.38039	2.12497
SD TPM	0.37047	1.79007
SD MPs	0.38000	1.91415
SD Both	**0.37001**	**1.78327**

(e) Averages of 300 Runs, $\sigma_z^2 = 0.100$

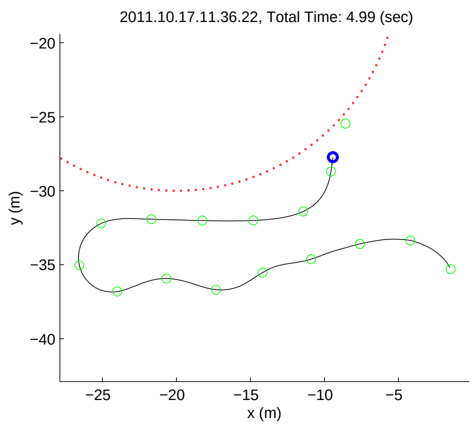
	Estimation	Prediction
Measurements	**0.61771**	-
Normal	0.59017	2.29826
SD TPM	0.56932	2.01657
SD MPs	0.58752	2.11496
SD Both	**0.56744**	**1.98977**

(f) Averages of 300 Runs, $\sigma_z^2 = 0.250$

	Estimation	Prediction
Measurements	**0.87141**	-
Normal	0.80584	2.43772
SD TPM	0.77543	2.20806
SD MPs	0.79386	2.24909
SD Both	**0.76850**	**2.17802**

(g) Averages of 300 Runs, $\sigma_z^2 = 0.500$

Figure 22: c_2011.10.17.11.36.22_25.00_30.00 with 7 different measurement noise variances.



	Estimation	Prediction
Measurements	**0.08851**	-
Normal	0.08737	1.65819
SD TPM	**0.08731**	1.66728
SD MPs	0.08737	**1.65689**
SD Both	0.08731	1.66666

(a) Averages of 300 Runs, $\sigma_z^2 = 0.005$

	Estimation	Prediction
Measurements	**0.12669**	-
Normal	0.12389	1.64009
SD TPM	0.12367	1.63928
SD MPs	0.12389	**1.63647**
SD Both	**0.12367**	1.63799

(b) Averages of 300 Runs, $\sigma_z^2 = 0.010$

	Estimation	Prediction
Measurements	**0.20017**	-
Normal	0.19042	1.64195
SD TPM	0.18997	1.64026
SD MPs	0.19042	**1.62033**
SD Both	**0.18997**	1.63426

(c) Averages of 300 Runs, $\sigma_z^2 = 0.025$

	Estimation	Prediction
Measurements	**0.28095**	-
Normal	0.25995	1.69875
SD TPM	0.25983	1.69305
SD MPs	0.25995	**1.65060**
SD Both	**0.25983**	1.67878

(d) Averages of 300 Runs, $\sigma_z^2 = 0.050$

	Estimation	Prediction
Measurements	**0.39790**	-
Normal	**0.36128**	1.84582
SD TPM	0.36306	1.83547
SD MPs	0.36128	**1.75973**
SD Both	0.36306	1.81455

(e) Averages of 300 Runs, $\sigma_z^2 = 0.100$

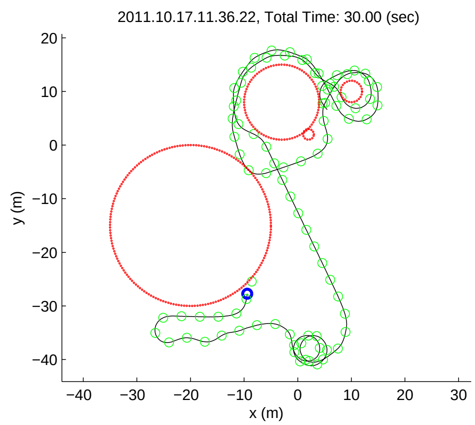
	Estimation	Prediction
Measurements	**0.63932**	-
Normal	0.58275	2.19681
SD TPM	0.58377	2.13788
SD MPs	**0.58273**	**2.06564**
SD Both	0.58376	2.09806

(f) Averages of 300 Runs, $\sigma_z^2 = 0.250$

	Estimation	Prediction
Measurements	**0.88006**	-
Normal	0.81951	2.59175
SD TPM	0.81118	2.48689
SD MPs	0.81864	**2.43675**
SD Both	**0.81110**	2.43763

(g) Averages of 300 Runs, $\sigma_z^2 = 0.500$

Figure 23: c_2011.10.17.11.36.22_30.00_35.00 with 7 different measurement noise variances.



	Estimation	Prediction
Measurements	**0.08869**	-
Normal	0.08776	1.31083
SD TPM	0.08768	1.24740
SD MPs	0.08776	**1.21168**
SD Both	**0.08767**	1.23594

(a) Averages of 300 Runs, $\sigma_z^2 = 0.005$

	Estimation	Prediction
Measurements	**0.12595**	-
Normal	0.12345	1.34199
SD TPM	0.12317	1.27149
SD MPs	0.12344	**1.24630**
SD Both	**0.12316**	1.26343

(b) Averages of 300 Runs, $\sigma_z^2 = 0.010$

	Estimation	Prediction
Measurements	**0.19769**	-
Normal	0.19057	1.40890
SD TPM	0.18932	1.33011
SD MPs	0.19049	**1.30924**
SD Both	**0.18925**	1.32240

(c) Averages of 300 Runs, $\sigma_z^2 = 0.025$

	Estimation	Prediction
Measurements	**0.28163**	-
Normal	0.26659	1.50339
SD TPM	0.26457	1.42209
SD MPs	0.26638	**1.40522**
SD Both	**0.26442**	1.41222

(d) Averages of 300 Runs, $\sigma_z^2 = 0.050$

	Estimation	Prediction
Measurements	**0.39323**	-
Normal	0.36480	1.65082
SD TPM	0.36067	1.57223
SD MPs	0.36436	**1.55515**
SD Both	**0.36034**	1.55580

(e) Averages of 300 Runs, $\sigma_z^2 = 0.100$

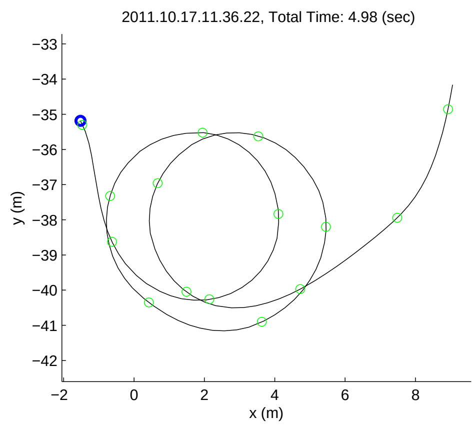
	Estimation	Prediction
Measurements	**0.63004**	-
Normal	0.57278	1.97299
SD TPM	0.56488	1.89882
SD MPs	0.57157	**1.86502**
SD Both	**0.56391**	1.87572

(f) Averages of 300 Runs, $\sigma_z^2 = 0.250$

	Estimation	Prediction
Measurements	**0.87976**	-
Normal	0.79246	2.28963
SD TPM	0.77400	2.21799
SD MPs	0.78808	**2.17406**
SD Both	**0.77168**	2.18650

(g) Averages of 300 Runs, $\sigma_z^2 = 0.500$

Figure 24: c_2011.10.17.11.36.22_30.00_60.00 with 7 different measurement noise variances.



	Estimation	Prediction
Measurements	**0.08758**	-
Normal	0.08771	1.15123
SD TPM	0.08771	1.15123
SD MPs	0.08771	1.15123
SD Both	0.08771	1.15123

(a) Averages of 300 Runs, $\sigma_z^2 = 0.005$

	Estimation	Prediction
Measurements	**0.12552**	-
Normal	0.12650	1.23311
SD TPM	0.12650	1.23311
SD MPs	0.12650	1.23311
SD Both	0.12650	1.23311

(b) Averages of 300 Runs, $\sigma_z^2 = 0.010$

	Estimation	Prediction
Measurements	**0.19685**	-
Normal	0.19676	1.38674
SD TPM	0.19676	1.38674
SD MPs	0.19676	1.38674
SD Both	0.19676	1.38674

(c) Averages of 300 Runs, $\sigma_z^2 = 0.025$

	Estimation	Prediction
Measurements	**0.27814**	-
Normal	0.27650	1.55360
SD TPM	0.27650	1.55360
SD MPs	0.27650	1.55360
SD Both	0.27650	1.55360

(d) Averages of 300 Runs, $\sigma_z^2 = 0.050$

	Estimation	Prediction
Measurements	**0.39460**	-
Normal	0.38853	1.74246
SD TPM	0.38853	1.74246
SD MPs	0.38853	1.74246
SD Both	0.38853	1.74246

(e) Averages of 300 Runs, $\sigma_z^2 = 0.100$

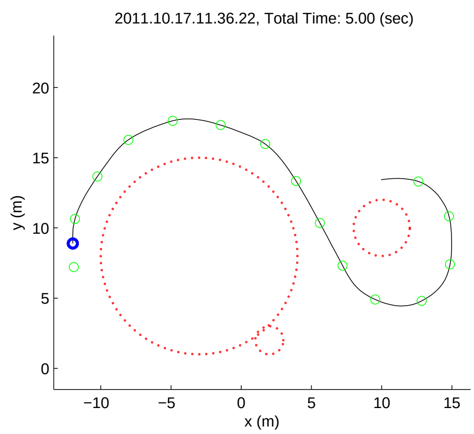
	Estimation	Prediction
Measurements	**0.62953**	-
Normal	0.59048	2.05139
SD TPM	0.59048	2.05139
SD MPs	0.59048	2.05139
SD Both	0.59048	2.05139

(f) Averages of 300 Runs, $\sigma_z^2 = 0.250$

	Estimation	Prediction
Measurements	**0.88222**	-
Normal	0.81850	2.39579
SD TPM	0.81850	2.39579
SD MPs	0.81850	2.39579
SD Both	0.81850	2.39579

(g) Averages of 300 Runs, $\sigma_z^2 = 0.500$

Figure 25: c_2011.10.17.11.36.22_35.00_40.00 with 7 different measurement noise variances.



	Estimation	Prediction
Measurements	**0.08827**	-
Normal	0.08728	1.49989
SD TPM	0.08729	1.38310
SD MPs	**0.08728**	**1.29906**
SD Both	0.08729	1.34491

(a) Averages of 300 Runs, $\sigma_z^2 = 0.005$

	Estimation	Prediction
Measurements	**0.12497**	-
Normal	0.12206	1.47425
SD TPM	0.12203	1.39912
SD MPs	0.12206	**1.30289**
SD Both	**0.12203**	1.36488

(b) Averages of 300 Runs, $\sigma_z^2 = 0.010$

	Estimation	Prediction
Measurements	**0.20052**	-
Normal	0.19358	1.49401
SD TPM	0.19255	1.43729
SD MPs	0.19358	**1.34143**
SD Both	**0.19255**	1.40934

(c) Averages of 300 Runs, $\sigma_z^2 = 0.025$

	Estimation	Prediction
Measurements	**0.27952**	-
Normal	0.26470	1.57489
SD TPM	0.26324	1.50482
SD MPs	0.26470	**1.42404**
SD Both	**0.26324**	1.47528

(d) Averages of 300 Runs, $\sigma_z^2 = 0.050$

	Estimation	Prediction
Measurements	**0.39712**	-
Normal	0.37484	1.70409
SD TPM	**0.36866**	1.60063
SD MPs	0.37484	**1.54782**
SD Both	0.36866	1.56644

(e) Averages of 300 Runs, $\sigma_z^2 = 0.100$

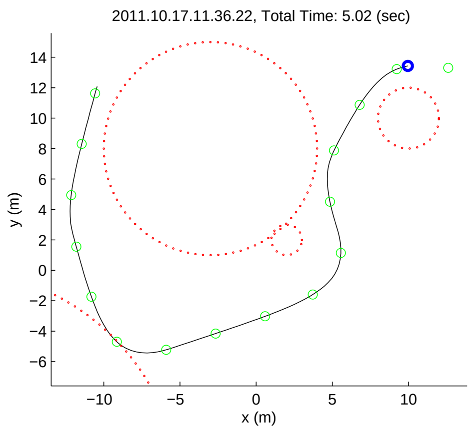
	Estimation	Prediction
Measurements	**0.62400**	-
Normal	0.58950	2.01470
SD TPM	0.56593	1.87280
SD MPs	0.58948	1.85982
SD Both	**0.56593**	**1.83504**

(f) Averages of 300 Runs, $\sigma_z^2 = 0.250$

	Estimation	Prediction
Measurements	**0.87100**	-
Normal	0.82380	2.37903
SD TPM	0.76880	2.20230
SD MPs	0.82175	2.20897
SD Both	**0.76820**	**2.14846**

(g) Averages of 300 Runs, $\sigma_z^2 = 0.500$

Figure 26: c_2011.10.17.11.36.22_45.00_50.00 with 7 different measurement noise variances.



	Estimation	Prediction
Measurements	**0.08885**	-
Normal	0.08778	1.32295
SD TPM	0.08767	1.17741
SD MPs	0.08777	**1.15395**
SD Both	**0.08767**	1.20843

(a) Averages of 300 Runs, $\sigma_z^2 = 0.005$

	Estimation	Prediction
Measurements	**0.12548**	-
Normal	0.12265	1.37474
SD TPM	0.12212	1.23630
SD MPs	0.12258	**1.21309**
SD Both	**0.12208**	1.26504

(b) Averages of 300 Runs, $\sigma_z^2 = 0.010$

	Estimation	Prediction
Measurements	**0.20072**	-
Normal	0.19383	1.49982
SD TPM	0.19196	**1.33593**
SD MPs	0.19344	1.33861
SD Both	**0.19178**	1.34841

(c) Averages of 300 Runs, $\sigma_z^2 = 0.025$

	Estimation	Prediction
Measurements	**0.28384**	-
Normal	0.27022	1.67710
SD TPM	0.26572	1.46309
SD MPs	0.26905	1.49313
SD Both	**0.26533**	**1.45582**

(d) Averages of 300 Runs, $\sigma_z^2 = 0.050$

	Estimation	Prediction
Measurements	**0.39541**	-
Normal	0.37429	1.90147
SD TPM	0.36784	1.68302
SD MPs	0.37241	1.72860
SD Both	**0.36698**	**1.64954**

(e) Averages of 300 Runs, $\sigma_z^2 = 0.100$

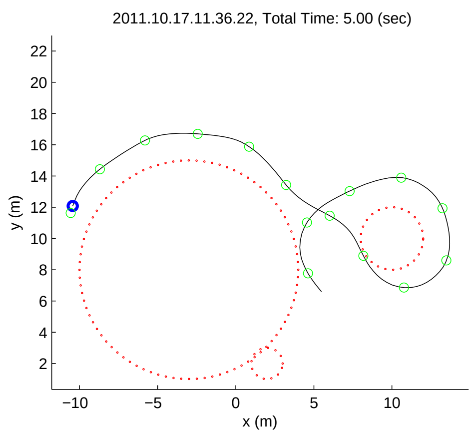
	Estimation	Prediction
Measurements	**0.61672**	-
Normal	0.56860	2.29375
SD TPM	0.56049	2.10909
SD MPs	0.56546	2.11662
SD Both	**0.55948**	**2.04034**

(f) Averages of 300 Runs, $\sigma_z^2 = 0.250$

	Estimation	Prediction
Measurements	**0.88121**	-
Normal	0.80489	2.64224
SD TPM	0.79037	2.47960
SD MPs	0.79332	2.42738
SD Both	**0.78590**	**2.38769**

(g) Averages of 300 Runs, $\sigma_z^2 = 0.500$

Figure 27: c_2011.10.17.11.36.22_50.00_55.00 with 7 different measurement noise variances.



	Estimation	Prediction
Measurements	**0.08889**	-
Normal	0.08861	1.64942
SD TPM	0.08820	1.49927
SD MPs	0.08861	**1.42104**
SD Both	**0.08819**	1.44640

(a) Averages of 300 Runs, $\sigma_z^2 = 0.005$

	Estimation	Prediction
Measurements	**0.12481**	-
Normal	0.12337	1.65670
SD TPM	0.12283	1.51766
SD MPs	0.12337	**1.42706**
SD Both	**0.12280**	1.47558

(b) Averages of 300 Runs, $\sigma_z^2 = 0.010$

	Estimation	Prediction
Measurements	**0.19969**	-
Normal	0.19572	1.70275
SD TPM	0.19392	1.58207
SD MPs	0.19558	**1.51274**
SD Both	**0.19381**	1.55887

(c) Averages of 300 Runs, $\sigma_z^2 = 0.025$

	Estimation	Prediction
Measurements	**0.28296**	-
Normal	0.27422	1.82739
SD TPM	0.27013	1.72671
SD MPs	0.27377	**1.65479**
SD Both	**0.26966**	1.70958

(d) Averages of 300 Runs, $\sigma_z^2 = 0.050$

	Estimation	Prediction
Measurements	**0.39628**	-
Normal	0.37824	1.92932
SD TPM	0.37042	1.82628
SD MPs	0.37741	**1.78812**
SD Both	**0.36925**	1.81670

(e) Averages of 300 Runs, $\sigma_z^2 = 0.100$

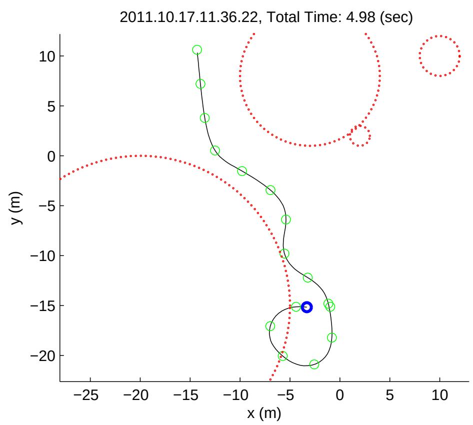
	Estimation	Prediction
Measurements	**0.62418**	-
Normal	0.58699	2.14521
SD TPM	0.56136	2.05061
SD MPs	0.58423	**1.98220**
SD Both	**0.55846**	2.03939

(f) Averages of 300 Runs, $\sigma_z^2 = 0.250$

	Estimation	Prediction
Measurements	**0.89960**	-
Normal	0.83539	2.42474
SD TPM	0.79075	2.38229
SD MPs	0.81862	**2.27333**
SD Both	**0.78386**	2.36767

(g) Averages of 300 Runs, $\sigma_z^2 = 0.500$

Figure 28: c_2011.10.17.11.36.22_55.00_60.00 with 7 different measurement noise variances.



	Estimation	Prediction
Measurements	**0.08953**	-
Normal	0.08832	2.10609
SD TPM	**0.08822**	1.76903
SD MPs	0.08832	**1.73640**
SD Both	0.08822	1.75705

(a) Averages of 300 Runs, $\sigma_z^2 = 0.005$

	Estimation	Prediction
Measurements	**0.12488**	-
Normal	0.12184	2.06820
SD TPM	**0.12160**	1.73264
SD MPs	0.12184	**1.71589**
SD Both	0.12160	1.72111

(b) Averages of 300 Runs, $\sigma_z^2 = 0.010$

	Estimation	Prediction
Measurements	**0.19870**	-
Normal	0.18782	2.02633
SD TPM	0.18781	1.68175
SD MPs	**0.18777**	1.70432
SD Both	0.18780	**1.67814**

(c) Averages of 300 Runs, $\sigma_z^2 = 0.025$

	Estimation	Prediction
Measurements	**0.27979**	-
Normal	0.25982	2.02578
SD TPM	0.25983	**1.71123**
SD MPs	**0.25954**	1.72944
SD Both	0.25975	1.71643

(d) Averages of 300 Runs, $\sigma_z^2 = 0.050$

	Estimation	Prediction
Measurements	**0.39797**	-
Normal	0.36408	2.08411
SD TPM	0.36105	1.81593
SD MPs	0.36247	1.82540
SD Both	**0.36027**	**1.81364**

(e) Averages of 300 Runs, $\sigma_z^2 = 0.100$

	Estimation	Prediction
Measurements	**0.63031**	-
Normal	0.58074	2.33641
SD TPM	0.56693	2.07605
SD MPs	0.57429	2.11834
SD Both	**0.56450**	**2.06558**

(f) Averages of 300 Runs, $\sigma_z^2 = 0.250$

	Estimation	Prediction
Measurements	**0.89369**	-
Normal	0.81435	2.63839
SD TPM	0.80130	2.38905
SD MPs	0.80126	2.44028
SD Both	**0.79624**	**2.35231**

(g) Averages of 300 Runs, $\sigma_z^2 = 0.500$

Figure 29: c_2011.10.17.11.36.22_65.00_70.00 with 7 different measurement noise variances.

Code Listing

callbacks/comparing_reset_button.m 51
callbacks/comparing_save_button.m 51
callbacks/comparing_save_function.m 51
callbacks/comparing_window_button.m 51
callbacks/comparing_window_function.m . . . 51
callbacks/comparing_wsave_button.m 52
callbacks/viewing_close_button.m 52
callbacks/viewing_open_button.m 52
callbacks/viewing_run_button.m 52
callbacks/viewing_save_button.m 52
closing/close_fig.m 52
closing/close_measurement_savefile.m 52
closing/close_state_savefile.m 52
closing/driving_close.m 53
closing/viewing_close.m 53
closing/write_last_run.m 53
comparing.m 53
comparing_db.m 53
driving.m 53
init/init_comparing.m 54
init/init_fig.m 55
init/init_filter.m 55
init/init_immo.m 56
init/init_listeners.m 56
init/init_measurements.m 56
init/init_obstacles.m 56
init/init_passenger.m 57
init/init_state_savefile.m 57
init/init_target.m 57
init/init_time.m 57
init/init_truth.m 57
init/init_truth_meas.m 58
init/init_truth_state.m 58
init/init_truths.m 58
init/init_viewing.m 58
init/init_viewing_plot.m 59
math/compute_errors.m 59
math/evaluate_state.m 59
math/evaluate_trans.m 60
math/filter_both.m 60
math/filter_normal.m 60
math/filter_sdmp.m 60
math/filter_sdtpm.m 61
math/filtering_test.m 61
math/gauss.m 61
math/get_P.m 61
math/get_Q.m 61
math/get_R.m 61
math/get_Rhalf.m 61
math/get_distance.m 61
math/get_turn_matrix.m 62
math/get_ws.m 62
math/get_z.m 62
math/imm_filter.m 62
math/imm_kf.m 62
math/imm_kf_predict.m 62
math/imm_mix.m 63
math/imm_predict.m 63
math/imm_remix.m 63
math/imm_set_R.m 63
math/imm_set_z.m 63
math/imm_update.m 63
math/immobj_test.m 64
math/init_tpm.m 64
math/kalman_filter.m 64
math/kobj_test.m 64
math/make_tpm.m 65
math/report_results.m 65
math/structure_test.m 65
math/update_mp.m 65
math/update_tpm.m 65
passengering.m 65
tests/adjacent_count_test.m 66
tests/distance_test.m 66
tests/evaluate_state_test.m 66
tests/evaluate_state_test_f.m 66
tests/gen_figures.m 68
tests/imm_test.m 68
tests/imm_test2.m 69
tests/kf_test.m 70
tests/tpm_test.m 71
tests/turn_test.m 71
tests/z_test.m 71
update/check_screenshot.m 72
update/draw_axes.m 72
update/draw_driver.m 72
update/draw_filter.m 72
update/draw_measurement.m 73
update/draw_objects.m 73
update/draw_obstacles.m 73
update/draw_passenger.m 73
update/draw_target.m 73
update/draw_tracks.m 74
update/draw_view.m 74
update/save_measurement.m 74
update/save_state.m 75

update/update_camera_counts.m 75
update/update_camera_time.m 75
update/update_camera_view.m 76
update/update_filter.m 76
update/update_filter_meas.m 77
update/update_key_changes.m 77
update/update_key_statuses.m 77
update/update_passenger.m 78
update/update_target.m 78
update/update_viewing_plot.m 78
utility/addseconds.m 79
utility/append_db.m 79
utility/display_keys.m 79
utility/enable_all.m 79
utility/find_adjacent_count.m 79
utility/get_datestr.m 80
utility/get_running.m 80
utility/key_changed.m 80
utility/key_down.m 80
utility/key_downed.m 80
utility/key_tracker_down.m 80
utility/make_results_table.m 81
utility/open_last_run.m 82
utility/print_progress.m 82
utility/query_filter.m 82
utility/save_fig.m 82
utility/set_addedpath.m 82
utility/set_running.m 82
viewing.m 82

callbacks/comparing_reset_button.m

```
function comparing_reset_button( hObject, edata )

fig = get( hObject, 'Parent' );
handles = guidata( fig );
case_time = handles.case_time;

write_last_run( case_time );
update_viewing_plot( fig );
```

callbacks/comparing_save_button.m

```

function comparing_save_button( hObject, edata )

fig = get( hObject, 'Parent' );
comparing_window_function( fig, 1 );
comparing_save_function( fig );
```

callbacks/comparing_save_function.m

```
function comparing_save_function( fig )

resdir = 'results/';

handles = guidata( fig );
case_time = handles.case_time;

nruns_def = get( handles.ph_nruns, 'Title' );
nruns = get( handles.edit_nruns, 'String' );
nruns = sscanf( nruns, '%d' );

NF = 1;
zes = zeros( [nruns, 1] );
xes = zeros( [nruns, NF] );
zps = zeros( [nruns, NF] );

enable_all( fig, 'button', 'off' );
enable_all( fig, 'edit', 'off' );
for( i = 1:nruns )
  it = sprintf( '%d', i );
  set( handles.edit_nruns, 'String', it );
  itt = sprintf( '%d/%d', i, nruns );
  set( handles.ph_nruns, 'Title', itt );
  drawnow;

  init_filter( fig );
  handles = guidata( fig );
%  num_filters is not known until init_filter
%  is run once.  Reallocate memory once that
%  happens.
  if( i == 1 )
    NF = handles.num_filters;
    zes = zeros( [nruns, 1] );
    xes = zeros( [nruns, NF] );
    zps = zeros( [nruns, NF] );
  end
  zes(i) = handles.filter_ze;
  xes(i,:) = handles.filter_xe(:)';
  xps(i,:) = handles.filter_xp(:)';
end

%  zrhalf is not known either unless it is
%  specified in the last_run.csv file.
s = sprintf( '%s_%s_%4.2f_%4.2f', ...
  'c', ...
  case_time, ...
  handles.T_simulation_min, ...
  handles.T_simulation_max );

fig2 = figure('visible', 'off');
ax = copyobj( handles.axes_main, fig2 );
set( ax, ...
  'units', 'normalized', ...
  'position', [0.1 0.1 0.8 0.8] );
save_fig( fig2, ...
  sprintf( '%s%s.eps', resdir, s ) );
close( fig2 );

robj = struct( ...
  'namebase', s, ...
  'NF', NF, ...
  'zes', zes, ...
  'xes', xes, ...
  'xps', xps, ...
  'zrhalf', handles.zrhalf, ...
  'names', {handles.filters_names}, ...
  'resdir', resdir );
make_results_table( robj );

enable_all( fig, 'button', 'on' );
enable_all( fig, 'edit', 'on' );
set( handles.ph_nruns, 'Title', nruns_def );
drawnow;
```

callbacks/comparing_window_button.m

```
function comparing_window_button( hObject, edata )

fig = get( hObject, 'Parent' );
comparing_window_function( fig );
```

callbacks/comparing_window_function.m

```
function comparing_window_function(fig,varargin)

save = 0;
if( size( varargin, 2 ) == 1 )
  disp( 'Saving Parameters to DB File' );
  save = varargin{1};
end

handles = guidata( fig );
case_time = handles.case_time;

tmin = get( handles.edit_tmin, 'String' );
tmin = sscanf( tmin, '%f' );

tmax = get( handles.edit_tmax, 'String' );
tmax = sscanf( tmax, '%f' );

```

```
nruns = get( handles.edit_nruns, 'String' );

%  In the db, we want to repeat each case for
%  multiple sensor noise values.  Thus,
%  do not append the sensor noise to the
%  database file.
s = sprintf( '%s,%4.2f,%4.2f,%s', ...
  case_time, tmin, tmax, nruns );

if( save )
  append_db( s );
end

%  In the last_run file, we need the sensor
%  noise (or we could use the default) in
%  order to have the filtering program
%  recognize the non-default value.
zrhalf = '';
if( isfield( handles, 'zrhalf' ) )
  zrhalf = sprintf( ',%4.3f', handles.zrhalf );
end
s = sprintf( '%s%s', s, zrhalf );
write_last_run( s );

update_viewing_plot( fig );
```

callbacks/comparing_wsave_button.m

```
function comparing_wsave_button( hObject, edata )

fig = get( hObject, 'Parent' );
comparing_window_function( fig, 1 );
```

callbacks/viewing_close_button.m

```
function viewing_close_button( hObject, edata )

fig = get( hObject, 'Parent' );
close( fig );
```

callbacks/viewing_open_button.m

```
function viewing_open_button( hObject, edata )

fig = get( hObject, 'Parent' );
handles = guidata( fig );
case_time = handles.case_time;
s = ['states/', case_time, '.csv'];
[filename,path] = uigetfile( '*.csv', ...
  'Choose a Ground Truth Set', s );

filename = filename( 1:(end-4) );
if( ~isempty( filename ) )
  write_last_run( filename );
end

update_viewing_plot( fig );
```

callbacks/viewing_run_button.m

```
function viewing_run_button( hObject, edata )

fig = get( hObject, 'Parent' );
close( fig );
passengering;
```

callbacks/viewing_save_button.m

```

function viewing_save_button( hObject, edata )

fig = get( hObject, 'Parent' );
handles = guidata( fig );
case_time = handles.case_time;
s = ['saves/viewing_', case_time, '.eps'];

fig2 = figure('visible', 'off');
ax = copyobj( handles.axes_main, fig2 );
set( ax, ...
  'units', 'normalized', ...
  'position', [0.1 0.1 0.8 0.8] );
axis( ax, 'tight' );
axis( ax, 'equal' );
save_fig( fig2, s );
close( fig2 );
```

closing/close_fig.m

```
function close_fig( fig )

handles = guidata( fig );

write_last_run( handles.case_time );

fprintf( '%s\n', handles.case_time );
fprintf( '\n%3.3f seconds have elapsed.\n', ...
  handles.T_since_start );
fprintf( '%d points saved.\n\n', handles.count );
fprintf( 'Average FPS was %.2f\n\n', ...
  handles.count/handles.T_since_start );

delete( fig );
```

closing/close_measurement_savefile.m

```
function close_measurement_savefile( fig )

handles = guidata( fig );
fclose( handles.measurement_savefile );
```

closing/close_state_savefile.m

```
function close_state_savefile( fig )

handles = guidata( fig );
fclose( handles.state_savefile );
```

closing/driving_close.m

```matlab

function driving_close( fig )

set_running( fig, false );
```

closing/viewing_close.m

```matlab
function viewing_close( fig )

handles = guidata( fig );
rmpath( handles.addedpath );

delete( fig );
```

closing/write_last_run.m

```matlab
function write_last_run( case_time )

fout = fopen( 'last_run.csv', 'w+' );
fprintf( fout, '%s\n', case_time );
fclose( fout );
```

comparing.m

```matlab
close all
clear all
clc

addedpath = genpath( '.' );
addpath( addedpath );

fig = figure();
set_addedpath( fig, addedpath );
set( fig, 'CloseRequestFcn', ...
  'viewing_close( fig )' );
init_comparing( fig );
update_viewing_plot( fig );
```

comparing_db.m

```matlab
clear all
clc

addedpath = genpath( '.' );
addpath( addedpath );

fin = fopen( 'runs_db.csv' );
Aorig = textscan( fin, '%s' );
Aorig = Aorig{1};

Atotal = {};
v = [.005, .01, .025, .05, .1, .25, .5];

for( i = 1:numel( Aorig ) )
  A = Aorig{i};
  sf = strfind( A, ',' );
  if( numel( sf ) == 3 )
    for( j = 1:numel( v ) )
      An = sprintf( '%s,%4.3f', A, v(j) );
      Atotal{end+1} = An;
    end
  else
    Atotal{end+1} = A;
  end
end

for( i = 1:numel( Atotal ) )
  A = Atotal{i};
  params = A;

  nstar = strfind( A, '*' );
  if( numel(nstar) == 0 )
    fprintf( ...
      '\nParameters: %s\n', ...
      params );
    write_last_run( params );

    fig = figure();
    init_comparing( fig );
    update_viewing_plot( fig );

    comparing_save_function( fig );
    close( fig );
  end
end

rmpath( addedpath );
```

driving.m

```matlab
clear all
close all
clc

addedpath = genpath( '.' );
addpath( addedpath );

fig = figure();

init_fig( fig );
init_listeners( fig );
init_target( fig );
init_obstacles( fig, 'config/rocks.csv' );

state_savefile = init_state_savefile( fig );
measurement_savefile = init_measurements( fig );
init_time( fig );

while( get_running( fig ) )

  update_key_changes( fig );
  update_target( fig );
  if( key_down( fig, {'escape'} ) )
    set_running( fig, false );
  end

  clf;
  draw_driver( fig );
  drawnow;

```

```
  save_state( fig );
  save_measurement( fig );
end

close_state_savefile( fig );
close_measurement_savefile( fig );
close_fig( fig );

rmpath( addedpath );
```

init/init_comparing.m

```
function init_comparing( fig )

set( fig, 'Name', 'Select Ground Truth Case' );
handles = guidata( fig );

if( ~isfield( handles, 'nruns' ) )
  handles.nruns = '2';
end

% Set up the basic controls section
handles.axes_main = axes( 'Parent', fig, ...
  'Position', [.1, .3, .53, .6] );
handles.edit = uicontrol( 'Parent', fig, ...
  'Style', 'edit', ...
  'Units', 'Normalized', ...
  'Position', [.13, .12, .5, .07] );
handles.button_open = uicontrol( ...
  'Parent', fig, ...
  'Style', 'pushbutton', ...
  'String', 'Change Track', ...
  'Units', 'Normalized', ...
  'Position', [.67, .12, .2, .07] );
handles.button_run = uicontrol( ...
  'Parent', fig, ...
  'Style', 'pushbutton', ...
  'String', 'Passengering', ...
  'Units', 'Normalized', ...
  'Position', [.67, .03, .2, .07] );
handles.button_close = uicontrol( ...
  'Parent', fig, ...
  'Style', 'pushbutton', ...
  'String', 'Close', ...
  'Units', 'Normalized', ...
  'Position', [.13, .03, .2, .07] );

% % Set up the time control section
handles.button_window = uicontrol( ...
  'Parent', fig, ...
  'Style', 'pushbutton', ...
  'String', 'Set', ...
  'Units', 'Normalized', ...
  'Position', [.67, .58, .09, .07] );
handles.button_wsave = uicontrol( ...
  'Parent', fig, ...
  'Style', 'pushbutton', ...
  'String', 'Save', ...
  'Units', 'Normalized', ...
  'Position', [.78, .58, .09, .07] );
handles.button_reset = uicontrol( ...
  'Parent', fig, ...
  'Style', 'pushbutton', ...
  'String', 'Reset window', ...
  'Units', 'Normalized', ...
  'Position', [.43, .03, .2, .07] );

handles.ph_tmin = uipanel( 'Parent', fig, ...
  'Title', 'Tmin (sec)', ...
  'Units', 'Normalized', ...
  'Position', [.67, .8, .2, .11] );
handles.edit_tmin = uicontrol( ...
  'Parent', handles.ph_tmin, ...
  'Style', 'edit', ...
  'Units', 'Normalized', ...
  'Position', [.1, .1, .8, .8] );
handles.ph_tmax = uipanel( 'Parent', fig, ...
  'Title', 'Tmax (sec)', ...
  'Units', 'Normalized', ...
  'Position', [.67, .67, .2, .11] );
handles.edit_tmax = uicontrol( ...
  'Parent', handles.ph_tmax, ...
  'Style', 'edit', ...
  'Units', 'Normalized', ...
  'Position', [.1, .1, .8, .8] );

% % Set up the monte carlo run controls
handles.button_save = uicontrol( ...
  'Parent', fig, ...
  'Style', 'pushbutton', ...
  'String', 'Make Runs', ...
  'Units', 'Normalized', ...
  'Position', [.67, .3, .2, .07] );
handles.ph_nruns = uipanel( 'Parent', fig, ...
  'Title', 'N. of Runs', ...
  'Units', 'Normalized', ...
  'Position', [.67, .39, .2, .11] );
handles.edit_nruns = uicontrol( ...
  'Parent', handles.ph_nruns, ...
  'Style', 'edit', ...
  'String', handles.nruns, ...
  'Units', 'Normalized', ...
  'Position', [.1, .1, .8, .8] );

set( handles.button_open, 'CallBack', ...
  @viewing_open_button );
set( handles.button_run, 'CallBack', ...
  @viewing_run_button );
set( handles.button_close, 'CallBack', ...
  @viewing_close_button );

set( handles.button_window, 'CallBack', ...
  @comparing_window_button );
set( handles.button_wsave, 'CallBack', ...
  @comparing_wsave_button );
set( handles.button_reset, 'CallBack', ...
  @comparing_reset_button );
set( handles.button_save, 'CallBack', ...
  @comparing_save_button );

guidata( fig, handles );
```

init/init_fig.m

```
function init_fig( fig )

handles = guidata( fig );

set_running( fig, true );
set( fig, 'CloseRequestFcn', ...
  'driving_close( fig )' );

s = get_datestr();
handles.case_time = s;

s = 'Driving';

set( fig, 'Name', s );
set( fig, 'MenuBar', 'none' );

handles.viewwidth0 = 5;
handles.viewwidth = handles.viewwidth0;
handles.offset_view = false;
handles.offset_amount = 0;

set( fig, 'Units', 'Normalized' );
set( fig, 'Position', [0, 0, .4, .6] );
movegui( 'northeast' );

guidata( fig, handles );
```

init/init_filter.m

```
function init_filter( fig )

handles = guidata( fig );

%  One function for each filter that is running.
funs = {   @filter_normal, ...
           @filter_sdtpm, ...
           @filter_sdmp, ...
           @filter_both };
names = {  'Normal', ...
           'SD TPM', ...
           'SD MPs', ...
           'SD Both' };
colors = { [0,0,1], ...
           [0,.8,0], ...
           [.8,.5,0], ...
           [.8,0,.8] };
colnames={ 'Blue', ...
           'Green', ...
           'Orange', ...
           'Purple' };

NF = numel( funs );
handles.filters_funs = funs;
handles.filters_names = names;
handles.filters_colors = colors;
handles.filters_color_names = colnames;
handles.num_filters = NF;

%  Initialize at the current time k
zcount = handles.count_meas;
handles.filter_init_count = zcount;

trus = handles.truth_meas;
handles.filter_t0 = trus{1}(zcount);

if( ~isfield( handles, 'zrhalf' ) )
  handles.zrhalf = 0.1;
  guidata( fig, handles );
  handles = guidata( fig );
end

%  The first measurement comes from time k+1
if( zcount < handles.count_meas_max )
  zcount = zcount + 1;
end

%  Initialize imm object immo.
immo = init_immo( fig );
x = immo.x;
P = immo.P;
mus = immo.mus;
tpm0 = immo.tpm;

%  robj = struct( ...
%    'x', x, ...
%    'P', P, ...
%    'mus', mus, ...
%    'tpm', tpm0 );

robj = struct( 'x', x );

handles.filter_x0 = x;
handles.filter_P0 = P;
handles.filter_tpm0 = tpm0;

%  set of "true measurements"
%  these will be corrupted by noise to obtain
%  actual measurements.
zs = [trus{2}(zcount:end), ...
      trus{3}(zcount:end)];
nmeas = length( zs );
handles.filter_zs = zs;

%  results of filter i, time k
handles.filters_e = repmat( robj, NF, nmeas );
handles.filters_p = repmat( robj, NF, nmeas );

%  set of current time's imm objects.
handles.filters_immos = repmat( immo, NF, 1 );

if( isfield( handles, 'filter_zn' ) )
  handles = rmfield( handles, 'filter_zn' );
end

handles.filter_running = true;
guidata( fig, handles );

update_filter( fig );
compute_errors( fig );
```

init/init_immo.m

```
function immo = init_immo( fig )

handles = guidata( fig );

nummodes = 5;
Tsamp = handles.samp_time;

%  The turnrate is seconds / turn
turnrate = 1.42;

zcount = handles.count_meas;

trus = handles.truth_meas;
x = [trus{2}(zcount);
     trus{3}(zcount);
     trus{4}(zcount);
     trus{5}(zcount)];
P = get_P();
mu = ones([nummodes, 1]) / (nummodes);

if( zcount < handles.count_meas_max )
  zcount = zcount + 1;
end

z = [trus{2}(zcount);
     trus{3}(zcount)];

kobj = struct( ...
  'x', x, ...
  'P', P, ...
  'A', get_turn_matrix( 0, Tsamp ), ...
  'Q', get_Q(), ...
  'z', z, ...
  'H', [1, 0, 0, 0; 0, 1, 0, 0], ...
  'R', get_R() );

if( isfield( handles, 'zrhalf' ) )
  kobj.R = get_R( handles.zrhalf );
end

tpm0 = init_tpm( nummodes, nummodes^2 );

immo = struct( ...
  'modes', repmat( kobj, nummodes, 1 ), ...
  'tpm', tpm0, ...
  'mus', mu, ...
  'x', x, ...
  'P', P );

ws = get_ws( nummodes, turnrate );
for( i = 1:(nummodes) )
  immo.modes(i).A = ...
    get_turn_matrix( ws(i), Tsamp );
end
```

init/init_listeners.m

```
function init_listeners( fig )

handles = guidata( fig );
handles.running = true;

%  tmin specifies the response time for
%  switching between key pressed and key
%  released
handles.tmin = 0.05;

%  handles.keys is a list of all of the
%  keys that we wish to monitor
handles.keys = {'a', 's', 'd', 'f', 'space', ...
                'uparrow', 'downarrow', ...
                'leftarrow', 'rightarrow', ...
                'escape', 't', 'p', ...
                'i', 'z', 'q', ...
                'pageup', 'pagedown', 'home'};
numkeys = numel( handles.keys );
handles.numkeys = numkeys;

c = clock;
handles.key_downtimes = repmat( {c}, 1, numkeys );
handles.key_uptimes = repmat( {c}, 1, numkeys );
handles.key_statuses = zeros( 1, numkeys );
handles.key_changes = zeros( 1, numkeys );

guidata( fig, handles );

set( fig, 'keypressfcn', @(obj,evt) ...
          key_tracker_down( evt, true, fig ) );
set( fig, 'keyreleasefcn', @(obj,evt) ...
          key_tracker_down( evt, false, fig ) );
```

init/init_measurements.m

```
function savefile = init_measurement( fig )

handles = guidata( fig );

%  s = datestr( now, 'yyyy.mm.dd.HH.MM.SS' );
s = handles.case_time;
s = ['measurements/', s, '.csv'];
savefile = fopen( s, 'w' );

handles.measurement_string = s;
handles.measurement_savefile = savefile;

%  Get one measurement every T seconds.
handles.T_measurement = 0.35;

guidata( fig, handles );
```

init/init_obstacles.m

```
function init_obstacles( fig, filename )

handles = guidata( fig );

fid = fopen( filename, 'r' );
f = textscan( fid, '%f,%f,%f' );
fclose( fid );

x = f{1};
```

```
y = f{2};
r = f{3};

handles.obstacles_x = x;
handles.obstacles_y = y;
handles.obstacles_r = r;

xo = [];
yo = [];

for( i = 1:numel( r ) )
  thetaskip = 1 / ( 2 * ceil( r(i) ) );
  thetas = [0:thetaskip:(2*pi)]';

  xs = cos(thetas) * r(i);
  ys = sin(thetas) * r(i);

  xs = xs + ones(size(xs))*x(i);
  ys = ys + ones(size(ys))*y(i);

  xo = [xo; xs];
  yo = [yo; ys];
end

handles.obstacle_marks_x = xo;
handles.obstacle_marks_y = yo;
guidata( fig, handles );
```

init/init_passenger.m

```

function init_passenger( fig )

open_last_run( fig );

handles = guidata( fig );
fprintf('%s\n\n', handles.case_time );

s = 'Passengering';
set( fig, 'Name', s );

handles.filter_running = false;

guidata( fig, handles );
```

init/init_state_savefile.m

```
function savefile = init_savefile( fig )

handles = guidata( fig );

%  s = datestr( now, 'yyyy.mm.dd.HH.MM.SS' );
s = handles.case_time;
s = ['states/', s, '.csv'];
savefile = fopen( s, 'w' );

handles.state_string = s;
handles.state_savefile = savefile;

guidata( fig, handles );
```

init/init_target.m

```
function init_target( fig )

handles = guidata(fig);

handles.target_x = 0;
handles.target_y = 0;
handles.target_v = 0;
handles.target_dv = 7.5;

handles.meas_last_x = handles.target_x;
handles.meas_last_y = handles.target_y;

handles.target_theta = 0;
handles.target_dtheta = 0;
handles.target_ddtheta = pi*8;

handles.target_cv = .75;
handles.target_cdtheta = pi*1.5;

guidata( fig, handles );
```

init/init_time.m

```
function init_time( fig )

handles = guidata( fig );

handles.T_start = clock;
handles.T_since_start = 0;
handles.T_last_draw = clock;

handles.T_last_measurement = clock;

handles.count = 0;

if( ~isfield( handles, 'T_simulation' ) )
  handles.T_simulation = 0;
end
if( ~isfield( handles, 'T_simulation_min' ) )
  handles.T_simulation_min = 0;
end
handles.paused = true;
handles.T_simulation_mult = 0;

handles.T_window = .5;

guidata( fig, handles );
```

init/init_truth.m

```
function truth = init_truth( fn, fmt, varargin )

fin = fopen( fn, 'r' );
truth = textscan( fin, fmt, 'Delimiter', ',' );

t = truth{1};

l = 1;
h = length( t );
```

```
if( size( varargin, 2 ) == 1 )
  tlow = varargin{1};
  l = find_adjacent_count( tlow, l, t );
  tl = t(l);
end
if( size( varargin, 2 ) == 2 )
  tlow = varargin{1};
  thigh = varargin{2};
  l = find_adjacent_count( tlow, l, t );
  h = find_adjacent_count( thigh, h, t );
  tl = t(l);
  th = t(h);
end

for( i = 1:numel(truth) )
  truth{i} = truth{i}(l:h);
end
```

init/init_truth_meas.m

```
function init_truth_meas( fig, varargin )

handles = guidata( fig );

fname = ['measurements/', ...
  handles.case_time, '.csv'];
fmt = '%n%n%n%n%n%n';

truth_meas = init_truth( ...
  fname, fmt, varargin{:} );

handles.truth_meas = truth_meas;
handles.count_meas = 1;
handles.count_meas_max = length(truth_meas{1});

handles.samp_time = truth_meas{1}(2) - ...
                    truth_meas{1}(1);

guidata( fig, handles );
```

init/init_truth_state.m

```

function init_truth_state( fig, varargin )

handles = guidata( fig );

fname = ['states/', ...
  handles.case_time, '.csv'];
fmt = '%n%n%n%n%n%n';

truth_state = init_truth( ...
  fname, fmt, varargin{:} );

handles.truth_state = truth_state;
handles.count_state = 1;
handles.count_state_max = length(truth_state{1});

guidata( fig, handles );
```

init/init_truths.m

```
function init_truths( fig )

han = guidata( fig );
tl = isfield( han, 'T_simulation_min' );
th = isfield( han, 'T_simulation_max' );

if( tl )
  tlow = getfield( han, 'T_simulation_min' );
  if( th )
    thigh = getfield( han, 'T_simulation_max' );
    init_truth_state( fig, tlow, thigh );
    init_truth_meas( fig, tlow, thigh );
  else
    init_truth_state( fig, tlow );
    init_truth_meas( fig, tlow );
  end
else
  init_truth_state( fig );
  init_truth_meas( fig );
end

handles = guidata( fig );
T1 = min( handles.truth_state{1} );
T2 = max( handles.truth_state{1} );
if( ~tl ) handles.T_simulation_min = T1; end
if( ~th ) handles.T_simulation_max = T2; end

fprintf( 'Total Time: %.3f (sec)\n', ...
  handles.T_simulation_max - ...
  handles.T_simulation_min );
fprintf( 'Start Time: %.3f (sec)\n', ...
  handles.T_simulation_min );
fprintf( '  End Time: %.3f (sec)\n', ...
  handles.T_simulation_max );
fprintf( 'Total Samples: %d\n', ...
  handles.count_meas_max );
fprintf( 'Sampling Time: %.3f (sec)\n\n', ...
  handles.samp_time );

guidata( fig, handles );
```

init/init_viewing.m

```
function init_viewing( fig )

set( fig, 'Name', 'Select Ground Truth Case' );
handles = guidata( fig );

handles.axes_main = axes( 'Parent', fig, ...
  'Position', [.1, .3, .8, .6] );
handles.edit = uicontrol( 'Parent', fig, ...
  'Style', 'edit', ...
  'Units', 'Normalized', ...
  'Position', [.13, .12, .5, .07] );
handles.button_open = uicontrol( ...
  'Parent', fig, ...
  'Style', 'pushbutton', ...
  'String', 'Change Track', ...
  'Units', 'Normalized', ...
  'Position', [.67, .12, .2, .07] );
```

```
handles.button_run = uicontrol( ...
  'Parent', fig, ...
  'Style', 'pushbutton', ...
  'String', 'Passengering', ...
  'Units', 'Normalized', ...
  'Position', [.67, .03, .2, .07] );
handles.button_save = uicontrol( ...
  'Parent', fig, ...
  'Style', 'pushbutton', ...
  'String', 'Save Fig', ...
  'Units', 'Normalized', ...
  'Position', [.43, .03, .2, .07] );
handles.button_close = uicontrol( ...
  'Parent', fig, ...
  'Style', 'pushbutton', ...
  'String', 'Close', ...
  'Units', 'Normalized', ...
  'Position', [.13, .03, .2, .07] );

set( handles.button_open, 'CallBack', ...
  @viewing_open_button );
set( handles.button_run, 'CallBack', ...
  @viewing_run_button );
set( handles.button_close, 'CallBack', ...
  @viewing_close_button );
set( handles.button_save, 'CallBack', ...
  @viewing_save_button );

guidata( fig, handles );
```

init/init_viewing_plot.m

```
function init_viewing_plot( fig )

handles = guidata( fig );

cla( handles.axes_main, 'reset' );
axes( handles.axes_main );
hold on;
axis equal;

case_time = handles.case_time;
var_state = handles.truth_state;
var_meas = handles.truth_meas;
ttime = var_state{1}(end) - var_state{1}(1);

s = sprintf( '%s, Total Time: %.2f (sec)', ...
  case_time, ttime );
title( s );

plot( var_state{2}, var_state{3}, 'k-' );
plot( var_state{2}(1), var_state{3}(1), 'bo', ...
  'LineWidth', 2 );
plot( var_meas{2}, var_meas{3}, 'go' );
xlabel( 'x (m)' );
ylabel( 'y (m)' );
axis manual;

scale = .05;
xw = get( gca, 'XLim' );
w = ( xw(2) - xw(1) ) * scale;
xw = xw + [-w, w];
set( gca, 'XLim', xw );

xw = get( gca, 'YLim' );
w = ( xw(2) - xw(1) ) * scale;
xw = xw + [-w, w];
set( gca, 'YLim', xw );

init_obstacles( fig, 'config/rocks.csv' );
handles = guidata( fig );
plot( handles.obstacle_marks_x, ...
      handles.obstacle_marks_y, ...
      'o', 'Color', 1*[1,0,0], ...
      'MarkerSize', 1 );
```

math/compute_errors.m

```
function compute_errors( fig )

handles = guidata( fig );

xt = handles.filter_zs(2:end,:);
zn = handles.filter_zn(2:end,:);
nmeas = length( xt );

ze = 0;
for( k = 1:nmeas )
  d = xt(k,:)' - zn(k,:)';
  ze = ze + sqrt(d'*d);
end
handles.filter_ze = ze / nmeas;

NF = handles.num_filters;
handles.filter_xe = zeros( [NF, 1] );
handles.filter_xp = zeros( [NF, 1] );

for( i = 1:NF )
  xs = cat( 2, handles.filters_e(i,2:end).x )';
  xp = cat( 2, handles.filters_p(i,1:end-1).x)';

  xs = xs(:,1:2);
  xp = xp(:,1:2);
  xse = 0;
  xpe = 0;
  for( k = 1:nmeas )
    xe = xt(k,:)' - xs(k,:)';
    xse = xse + sqrt( xe' * xe );

    xe = xt(k,:)' - xp(k,:)';
    xpe = xpe + sqrt( xe' * xe );
  end
  handles.filter_xe(i) = xse / nmeas;
  handles.filter_xp(i) = xpe / nmeas;
end

guidata( fig, handles );
```

math/evaluate_state.m

```
% %
% This function takes an immi object as its
```

```
% input. It also takes in the centers of
% the obstacles (cs) and the radii of the
% obstacles (rs).
%
% It outputs the value of each of the
% imm object's modes' states.
% %
function values = evaluate_state( immi, cs, rs )

beta = 12;

nummodes = length( immi.mus );
values = ones( size( immi.mus ) );

for( i = 1:nummodes )
  x = immi.modes(i).x(1:2);
  v1 = 1;
  for( k = 1:numel(rs) )
    c = cs(k,:)';
    v2 = get_distance( x, c, rs(k), beta );
    v1 = min( v1, v2 );
  end
  values(i) = v1;
end
```

math/evaluate_trans.m

```
% %
% This function takes an immi object as its
% input. It also takes in the centers of
% the obstacles (cs) and the radii of the
% obstacles (rs).
%
% It outputs the value of each of the
% imm object's modes' states' transitions.
% %
function values = evaluate_trans( immi, cs, rs )

beta = 6;

nummodes = length( immi.mus );
values = ones( nummodes );

for( i = 1:nummodes )
  x0 = immi.modes(i).x;
  for( j = 1:nummodes )
    A = immi.modes(j).A;
    x = A * x0;
    x = x(1:2);
    v1 = 1;
    for( k = 1:numel(rs) )
      c = cs(k,:)';
      v2 = get_distance( x, c, rs(k), beta );
      v1 = min( v1, v2 );
    end
    values(j,i) = v1;
  end
end
```

math/filter_both.m

```
function immo = filter_both(fig,immi,varargin)

immo = immi;

if( size( varargin, 2 ) == 1 )
  z = varargin{1};
  immo = imm_set_z( immi, z );

  immo = update_tpm( fig, immo );
  immo = imm_mix( immo );
  [immo, likes] = imm_kf( immo );

  likes = likes .* update_mp( fig, immo );
  [x, P, immo] = imm_remix( immo, likes );

  immo.x = x;
  immo.P = P;
else
  immo = update_tpm( fig, immo );
  immo = imm_mix( immo );
  immo = imm_kf_predict( immo );

  likes = update_mp( fig, immo );
  [x, P, immo] = imm_remix( immo, likes );

  immo.x = x;
  immo.P = P;
end

```

math/filter_normal.m

```
function immo = filter_normal(fig,immi,varargin)

immo = immi;

if( size( varargin, 2 ) == 1 )
  z = varargin{1};
  immo = imm_set_z( immi, z );
  [x,P,immo] = imm_filter( immo );
  immo.x = x;
  immo.P = P;
else
  [x,P,immo] = imm_predict( immi );
  immo.x = x;
  immo.P = P;
end
```

math/filter_sdmp.m

```
function immo = filter_sdmp(fig,immi,varargin)

immo = immi;

if( size( varargin, 2 ) == 1 )
  z = varargin{1};
  immo = imm_set_z( immi, z );

  immo = imm_mix( immo );
```

```
  [immo, likes] = imm_kf( immo );

  likes = likes .* update_mp( fig, immo );
  [x, P, immo] = imm_remix( immo, likes );

  immo.x = x;
  immo.P = P;
else
  immo = imm_mix( immo );
  immo = imm_kf_predict( immo );

  likes = update_mp( fig, immo );
  [x, P, immo] = imm_remix( immo, likes );

  immo.x = x;
  immo.P = P;
end
```

math/filter_sdtpm.m

```
function immo = filter_sdtpm(fig,immi,varargin)

immo = immi;

if( size( varargin, 2 ) == 1 )
  z = varargin{1};
  immo = imm_set_z( immi, z );
  immo = update_tpm( fig, immo );
  [x,P,immo] = imm_filter( immo );
  immo.x = x;
  immo.P = P;
else
  immo = update_tpm( fig, immi );
  [x,P,immo] = imm_predict( immo );
  immo.x = x;
  immo.P = P;
end
```

math/filtering_test.m

```
%  Nothing here anymore...
```

math/gauss.m

```
function p = gauss( x, mu, C )

const = 1 / sqrt( det( 2*pi*C ) );
p = const*exp(-.5* (x-mu)' * minv(C) * (x-mu) );
```

math/get_P.m

```
function P = get_P()

P = diag( [0.03, 0.03, 0.003, 0.003] );
```

math/get_Q.m

```
function Q = get_Q( varargin )

v = 0.1;
if( size( varargin,2 ) == 1 )
  v = varargin{1};
end

%  Q = diag( [1,1,1,1] * .001 );
Q = diag( [1,1,.5,.5] * v );
```

math/get_R.m

```
function R = get_R( varargin )

v = 0.1;
if( size(varargin,2) == 1 )
  v = varargin{1};
end

R = get_Rhalf(v) * get_Rhalf(v)';
```

math/get_Rhalf.m

```
function R = get_Rhalf( varargin )

v = 0.1;
if( size(varargin,2) == 1 )
  v = varargin{1};
end

R = sqrt(v) * eye( 2 );
%  R = sqrt(1) * eye( 2 );
```

math/get_distance.m

```
function v = get_distance( x, c, r, beta )

minval = 0.01;
scale = 1 - minval;

z = x - c;
d = sqrt( z' * z );

t = d - r;
v = 1 / ( 1 + exp( -beta*t ) );

v = v * scale + minval;
```

math/get_turn_matrix.m

```
%  %
%  This function gives the constant turn
%  motion model's matrix for a given
%  rotational speed w and time duration T.
%
%  The matrix produced assumes that the state
%  vector is x = [ x, y, xd, yd ].
%
%  NOTENOTENOTE: w > 0 results in a left turn
%                w < 0 results in a right turn
%  %
function A = get_turn_matrix( w, T )

%  w is the rotational speed (radians/sec)
%  T is the duration of the turn (sec)

A = eye( 4 );

if( abs(w) > .0001 )
A(1,:) = [1, 0, sin(w*T)/w, (cos(w*T)-1)/w];
A(2,:) = [0, 1, (1-cos(w*T))/w, sin(w*T)/w];
A(3,:) = [0, 0, cos(w*T), -sin(w*T)];
A(4,:) = [0, 0, sin(w*T), cos(w*T)];
else
A = [1, 0, T, 0;
     0, 1, 0, T;
     0, 0, 1, 0;
     0, 0, 0, 1];
end
```

math/get_ws.m

```
%  %
%  The turnrate should be specified as
%  seconds per turn.
%
%  The output is radians per second.
%  %
function ws = get_ws( nummodes, turnrate )

ws = 1:nummodes;
ws = ws - ceil( nummodes/2 );
ws = 2 * ws / (nummodes - 1);
ws = ws * 2 * pi;
ws = ws( end:-1:1 );

ws = ws / turnrate;
```

math/get_z.m

```
function z = get_z( zt, varargin )

v = 0.1;
if( size(varargin,2) == 1 )
  v = varargin{1};
end

z = get_Rhalf(v) * randn( size( zt ) );
z = z + zt;
```

math/imm_filter.m

```
%  %
%  Following Table 1 of 1993_design
%
%  This function takes an imm object as its
%  input and returns a new imm object as its
%  output along with the state estimate xkp1
%  and uncertainty estimate Pkp1.
%  %
function [xkp1, Pkp1, immo] = imm_filter( immi )

%  mix the modes together.  Create separate
%  x0j and P0j for each mode.
immo = imm_mix( immi );

%  Filter each of the modes separately
[immo, likes] = imm_kf( immo );

%  use the mixed result to find output estimates
[xkp1, Pkp1, immo] = imm_remix( immo, likes );
```

math/imm_kf.m

```
function [immo, likes] = imm_kf( immi )

likes = ones( size( immi.mus ) );
immo = immi;

%  find the j independent filtered results
%  along with the likelihoods of each result
for( j = 1:numel(immo.mus) )
  [x,P,likes(j)] = kalman_filter(immo.modes(j));
  immo.modes(j).x = x;
  immo.modes(j).P = P;
end
```

math/imm_kf_predict.m

```
function immo = imm_kf_predict( immi )

immo = immi;

for( i = 1:numel( immo.mus ) )
  A = immo.modes(i).A;
  Q = immo.modes(i).Q;
  x = immo.modes(i).x;
  P = immo.modes(i).P;
  immo.modes(i).x = A * x;
  immo.modes(i).P = A * P * A' + Q;
end
```

math/imm_mix.m

```
function immo = imm_mix( immi )

immo = immi;

%  predicted mode probabilities follow the
%  transition probability matrix
muje = immo.tpm * immo.mus;

%  mixing probabilities are the probabilities that
%  we were in mode i given that we are now in
%  mode j.
muij = zeros( size( immo.tpm ) );
for( i = 1:numel(immo.mus) )
  for( j = 1:numel(immo.mus) )
    muij(i,j) = immi.tpm(j,i)*immi.mus(i);
    muij(i,j) = muij(i,j) / muje(j);
  end
end

for( j = 1:numel(immo.mus) )

%    mixed initial state x0j and
%    mixed initial P0j
  immo.modes(j).x = zeros(size(immo.modes(j).x));
  immo.modes(j).P = zeros(size(immo.modes(j).P));

  for( i = 1:numel(immo.mus) )
    immo.modes(j).x = immo.modes(j).x + ...
      immi.modes(i).x * muij(i,j);
    immo.modes(j).P = immo.modes(j).P + ...
      immi.modes(i).P * muij(i,j);
  end

%    "spread of the means"
  X = zeros( size( immo.modes(j).P ) );
  Xdiff = zeros( size( immo.modes(j).x ) );
  for( i = 1:numel(immo.mus) )
    Xdiff = immi.modes(i).x - immo.modes(j).x;
    X = X + Xdiff*Xdiff'*muij(i,j);
  end

  immo.modes(j).P = immo.modes(j).P + X;
end

immo.mus = muje;
```

math/imm_predict.m

```
function [x1,P1,immo] = imm_predict( immi )

immo = imm_mix( immi );
immo = imm_kf_predict( immo );
[x1, P1, immo] = imm_remix( immo );
```

math/imm_remix.m

```
function [x1,P1,immo] = imm_remix(immi,varargin)

x1 = zeros(size(immi.modes(1).x));
P1 = zeros(size(immi.modes(1).P));
immo = immi;

likes = ones( size( immi.mus ) );
if( nargin == 2 )
  likes = varargin{1};
elseif( nargin > 2 )
  error( 'Too many inputs?' );
end

%  predicted mode probability is assigned
%  during imm_mix, and it is simply
%  immi.tpm * immi.mus
muje = immo.mus;

%  Combine the results from each of the filters
%  to form the overall estimate of state and
%  uncertainty.
for( j = 1:numel(immo.mus) )
  immo.mus(j) = muje(j)*likes(j) / (muje'*likes);
  x1 = x1 + immo.mus(j) * immo.modes(j).x;
  P1 = P1 + immo.mus(j) * immo.modes(j).P;
end

%  Another spread of the means
X = zeros( size( P1 ) );
for( j = 1:numel(immo.mus) )
  Xdiff = immo.modes(j).x - x1;
  X = X + Xdiff * Xdiff' * immo.mus(j);
end
P1 = P1 + X;
```

math/imm_set_R.m

```
function immo = imm_set_R( immi, R )

immo = immi;
for( i = 1:numel( immo.mus ) )
  immo.modes(i).R = R;
end
```

math/imm_set_z.m

```
function immo = imm_set_z( immi, z )

immo = immi;
for( i = 1:numel( immo.mus ) )
  immo.modes(i).z = z;
end
```

math/imm_update.m

```
%  This function has been replaced by
%  imm_kf and imm_remix.
```

math/immobj_test.m

```
% %
% This function tests to see if an interacting
% multiple models object (immobj) has a valid
% structure.
%
% Note: the modes of the immobj are stored
% in an array.  Each of the modes must be
% a valid kalman filter object (kobj).
% %
function immobj_test( immo )

if( ~isfield( immo, 'modes' ) )
  error( 'Mode set not found.' );
elseif( ~isfield( immo, 'tpm' ) )
  error( 'Transition probability matrix not found.' );
elseif( ~isfield( immo, 'mus' ) )
  error( 'Mode probabilities not found.' );
end

for( i = 1:numel( immo.mus ) )
  kobj_test( immo.modes(i) );
end

if( length( immo.mus ) ~= numel( immo.modes ) )
  error( ['Mode probabilities and ', ...
          'number of modes do not match.'] );
end
if( length( immo.mus )^2 ~= numel( immo.tpm ) )
  error( ['Mode probabilities and ', ...
          'transition probability matrix sizes.'] );
end
```

math/init_tpm.m

```
function tpm = init_tpm( n, const )

%  const specifies our preference for staying
%  in the current mode.
%  const = 0 implies that we have no mode
%  preference.
%  const = inf implies we never change modes

%  const = 1;
%  const = n;
%  const = n^2;

tpm = ones( n ) + const*eye( n );
tpm = make_tpm( tpm );
```

math/kalman_filter.m

```
% %
% This function takes a specially designed
% kobj as its input.
%
% kobj must be a struct that has the following
% fields:
% - x, current state estimate
% - P, current state uncertainty
% - A, the dynamics matrix
% - Q, covariance of process noise
% - z, next measurement
% - H, sensing matrix
% - R, sensor noise
%
% The output is an updated x and P reflecting
% the most recent data point z, given in the
% input.
%
% The equations governing the problem are
% as follows:
%
% xkp1 = A xk + w
% z = H xkp1 + v
%
% w ~ (0, Q)
% v ~ (0, R)
%
% The output will be xkp1, which is a fusion
% between xkp1e and the new measurement z.
%
% Also, the likelihood of y = z - xkp1e is
% reported in the variable l.
% %

function [xkp1, Pkp1, l] = kalman_filter( kobj )

% If kobj is incomplete, then the following
% function will throw errors.
kobj_test( kobj );

x = kobj.x;
P = kobj.P;
A = kobj.A;
Q = kobj.Q;
z = kobj.z;
H = kobj.H;
R = kobj.R;

I = eye( size( P ) );

xkp1e = A*x;
Pkp1e = A*P*A' + Q;

y = z - H * xkp1e;
S = H * Pkp1e * H' + R;
l = gauss( y, zeros(size(z)), S );
K = Pkp1e * H' * minv(S);
xkp1 = xkp1e + K*y;
Pkp1 = Pkp1e - K * S * K';

end
```

math/kobj_test.m

```
% %
% This function ensures that a kobj, short
% for a kalman filter object, has all of the
% necessary parameters.
%
```

```
% xkp1 = A xk + w
% z = H xkp1 + v
%
% w ~ (0, Q)
% v ~ (0, R)
% %
function kobj_test( objin )

if( ~isfield( objin, 'x' ) )
  error( 'Current state estimate x not found.' );
elseif( ~isfield( objin, 'P' ) )
  error( 'Current uncertainty P not found.' );
elseif( ~isfield( objin, 'A' ) )
  error( 'Dynamics matrix A not found.' );
elseif( ~isfield( objin, 'Q' ) )
  error( 'Process noise covariance Q not found.' );
elseif( ~isfield( objin, 'z' ) )
  error( 'Next measurement z not found.' );
elseif( ~isfield( objin, 'H' ) )
  error( 'Sensing matrix H not found.' );
elseif( ~isfield( objin, 'R' ) )
  error( 'Sensor noise covariance R not found.' );
end
```

math/make_tpm.m

```
function tpmo = make_tpm( tpm )

tpmo = tpm;
s = size( tpmo );

% iterate over columns.  sum tpmo(:,i) == 1
for( i = 1:s(2) )
  colsum = sum( tpmo(:,i) );
  tpmo(:,i) = tpmo(:,i) / colsum;
end
```

math/report_results.m

```
function report_results( fig )

% compute_errors( fig );
handles = guidata( fig );

fprintf( 'Measurement Error: %.5f (m)\n', ...
  handles.filter_ze );

NF = handles.num_filters;
fprintf( 'There are %d filters.\n', NF );
fprintf( '%10s%10s%10s  Name\n', ...
         'Error', 'P.Error' ,'Color' );

for( i = 1:NF )
  fprintf( '%10.5f%10.5f%10s  %s\n', ...
    handles.filter_xe(i), ...
    handles.filter_xp(i), ...
    handles.filters_color_names{i}, ...
    handles.filters_names{i} );
end

fprintf( '\n' );
```

math/structure_test.m

```
% See tests/kf_test.m
```

math/update_mp.m

```
function likes = update_mp( fig, immi )

handles = guidata( fig );

cs = [handles.obstacles_x, handles.obstacles_y];
rs = handles.obstacles_r;

likes = evaluate_state( immi, cs, rs );
```

math/update_tpm.m

```
function immo = update_tpm( fig, immi )

handles = guidata( fig );
tpm0 = handles.filter_tpm0;
cs = [handles.obstacles_x, handles.obstacles_y];
rs = handles.obstacles_r;

values = evaluate_trans( immi, cs, rs );

tpmnew = make_tpm( values .* tpm0 );

immo = immi;
immo.tpm = tpmnew;
```

passengering.m

```

clear all
close all
clc

addedpath = genpath( '.' );
addpath( addedpath );

fig = figure();

init_fig( fig );
init_passenger( fig );
init_truths( fig );

init_listeners( fig );
init_obstacles( fig, 'config/rocks.csv' );

init_time( fig );

while( get_running( fig ) )
  update_key_changes( fig );
  if( key_down( fig, {'escape'} ) )
    set_running( fig, false );
  end

  clf;
  update_passenger( fig );
  draw_passenger( fig );
```

```
  drawnow;
  check_screenshot( fig );
end

delete( fig );
rmpath( addedpath );
```

tests/adjacent_count_test.m

```
close all
clear all
clc

added_path = genpath( '../utility/' );
addpath( added_path );

ts = .5:.5:10;
cn = 4;

T = [7, 7.003, 1, 1.02, .98];
for( i = 1:length(T) )
  disp( find_adjacent_count( T(i), cn, ts ) )
end
% Should be 14, 14, 2, 2, 1

rmpath( added_path );
```

tests/distance_test.m

```
close all
clear all
clc

added_path = genpath( '../' );
addpath( added_path );

beta = [3, 6];

% Pretend there is an obstacle with center
% at c and radius r.
c = [0;0];
r = 2;

x = -5:.1:5;
[xx,yy] = meshgrid( x, x );

% Evaluate the value of the distance function
% over a range of locations.
zz1 = zeros( size( xx ) );
zz2 = zz1;
for( i = 1:numel(zz1) )
  p = [xx(i); yy(i)];
  zz1(i) = get_distance( p, c, r, beta(1) );
  zz2(i) = get_distance( p, c, r, beta(2) );
end

% Make the figure
fig = figure();
hold on;

% Plot the beta(i) distance values
p1 = plot3( xx, yy, zz1, 'k' );
p2 = plot3( xx, yy, zz2, 'r' );
xlabel( 'x (m)' );
ylabel( 'y (m)' );
zlabel( 'Value of (x,y)' );
view( [15, -30] );

grid on;

% Title information
s = sprintf( ...
['Value Function', ...
 ' Using \\beta = %0.2f and', ...
 ' %0.2f for a Single Obstacle'], ...
 beta(1), beta(2) );
title( s );

% Legend information
h = legend( [p1(1), p2(1)], ...
  sprintf( '\\beta = %0.2f', beta(1) ), ...
  sprintf( '\\beta = %0.2f', beta(2) ), ...
  'Location', 'SouthEast' );

% Save file information
name = '../saves/distance_function.eps';
save_fig( fig, name );
close( fig );

rmpath( added_path );
```

tests/evaluate_state_test.m

```

close all
clear all
clc

added_path = genpath( '../' );
addpath( added_path );

v0 = 10;
theta0 = 3*pi/4;
x = [1; -3; v0; theta0];

z1 = [-1;0];
z2 = [-4;1];
z = [z1'; z2'];

name = '../saves/case1';
evaluate_state_test_f( x, z, name );

rmpath( added_path );
```

tests/evaluate_state_test_f.m

```
% %
% Input variables
% x is [x, y, v, theta]
% z is [x1, y1; x2, y2]
% %
function evaluate_state_test_f(x, z, name)
```

```
legend_location = 'SouthWest';

nummodes = 5;
Tsamp = 0.35;

%  The turnrate is seconds / turn
turnrate = 1.42;

fig = figure();
hold on;
init_fig( fig );
init_obstacles( fig, 'config/rocks.csv' );
handles = guidata( fig );
handles.filter_running = true;
guidata( fig, handles );

draw_obstacles( fig );

v0 = x(3);
theta0 = x(4);

x0 = [x(1); x(2); ...
      v0*cos(theta0); v0*sin(theta0)];

z1 = z(1,:)';
z2 = z(2,:)';

disp( 'The two green circles are z1 and z2.' );
pmeas = plot( z1(1), z1(2), 'ro', ...
              'LineWidth', 2 );
plot( z2(1), z2(2), 'ro', 'LineWidth', 2 );

handles.target_x = x0(1);
handles.target_y = x0(2);
handles.target_v = v0;

tpm0 = init_tpm( nummodes, nummodes^2 );
handles.filter_tpm0 = tpm0;

handles.target_theta = theta0;
handles.filter_running = true;
handles.offset_view = true;
handles.offset_amount = 0.35;

guidata( fig, handles );
draw_target( fig );

%  Set up an initial kalman filter object
kobj = struct( ...
  'x', x0, ...
  'P', diag( [0.03, 0.03, 0.003, 0.003] ), ...
  'A', get_turn_matrix( 0, 0.5 ), ...
  'Q', diag( [.1, .1, .1, .1] ), ...
  'z', z1, ...
  'H', [1, 0, 0, 0; 0, 1, 0, 0], ...
  'R', diag([.3, .3]) );

%  Create an imm object that repeats the
%  kalman filter object nummodes times.
immo = struct( ...
  'modes', repmat( kobj, nummodes, 1 ), ...
  'tpm', tpm0, ...
  'mus', ones([nummodes, 1]) / (nummodes) );

%  set the turn radii for each of the modes
%  of the imm object
ws = get_ws( nummodes, turnrate );

%  immo.modes(i).A is the dynamics matrix that
%  advances the state by Tsamp ( = 0.5 ) sec.
for( i = 1:(nummodes) )
  immo.modes(i).A = ...
    get_turn_matrix( ws(i), Tsamp );
end

draw_axes( fig );
%  disp( ['Press a key to show ',...
%         'the initial range of motions.'] );
legend( pmeas, 'Measurements', ...
  'Location', legend_location );
xlabel( 'x (m)' );
ylabel( 'y (m)' );
title( 'Initial Layout' );
nameout = sprintf( '%s_1_meas.eps', name );
save_fig( fig, nameout );
%  pause;

%  %
%  Begin two steps of filtering.
%  First, starting at x0, make a prediction
%  using the normal tpm (contained in immol)
%  and using the state-dependent tpm (in
%  immot).
%
%  Display the individual filters' results
%  (one for each of nummodes).
%  %
immol = immo;
immot = update_tpm( fig, immo );
[x1pl, P1pl, immo1plegs] = imm_predict( immol );
[x1pt, P1pt, immo1pthes] = imm_predict( immot );
for( i = 1:nummodes )
  plegsmode = plot( immo1plegs.modes(i).x(1), ...
              immo1plegs.modes(i).x(2), 'bs' );
  pthesmode = plot( immo1pthes.modes(i).x(1), ...
              immo1pthes.modes(i).x(2), 'gx' );
end

draw_axes( fig );
%  disp( ['Press a key to show ',...
%         'the first filtered result.'] );
title( 'Step 1, Individual Modes'' Estimates' );
legend( [pmeas, plegsmode, pthesmode], ...
  'Measurements', ...
  'Normal Modes', ...
  'SD TPM Modes', ...
  'Location', legend_location );
nameout = sprintf( '%s_2_modes.eps', name );
save_fig( fig, nameout );
%  pause;

%  %
%  Next, filter using z1 and x0.
```

```
% %
[x1fl, P1fl, immo1flegs] = imm_filter( immol );
[x1ft, P1ft, immo1fthes] = imm_filter( immot );
plegsfilt = plot( x1fl(1), x1fl(2), ...
                  'bs', 'LineWidth', 2 );
pthesfilt = plot( x1ft(1), x1ft(2), ...
                  'gx', 'LineWidth', 2 );

draw_axes( fig );
% disp( ['Press a key to show ',...
%         'the second set of motions.'] );
nameout = sprintf( '%s_3_filt.eps', name );
title( 'Step 1, Filtered Estimates' );
legend( [pmeas, plegsmode, pthesmode, ...
         plegsfilt, pthesfilt], ...
  'Measurements', ...
  'Normal Modes', ...
  'SD TPM Modes', ...
  'Normal Filtered', ...
  'SD TPM Filtered', ...
  'Location', legend_location );
save_fig( fig, nameout );
% pause;

% %
% From here, we predict again.  Make sure
% to update immo1fthes based on the new
% state.
% %
[x2pl, P2pl, immo2plegs] = ...
  imm_predict( immo1flegs );

immo1fthes = update_tpm( fig, immo1fthes );
[x2pt, P2pt, immo2pthes] = ...
  imm_predict( immo1fthes );

% %
% Show the individual filters' results to see
% where the possible motions are.
% %
for( i = 1:nummodes )
  plot( immo2plegs.modes(i).x(1), ...
        immo2plegs.modes(i).x(2), 'bs' );
  plot( immo2pthes.modes(i).x(1), ...
        immo2pthes.modes(i).x(2), 'gx' );
end

draw_axes( fig );
% disp( ['Press a key to show ',...
%         'the second filtered result.'] );
nameout = sprintf( '%s_4_modes.eps', name );
title( 'Step 2, Individual Modes'' Estimates' );
save_fig( fig, nameout );
% pause;

% %
% Set the new z and filter both the tpm0 case
% and the state-dependent tpm case.
%
% Display the filtered results.
% %
immo2l = imm_set_z( immo1flegs, z2 );
immo2t = imm_set_z( immo1fthes, z2 );
[x2fl, P2fl, immo2flegs] = imm_filter( immo2l );
[x2ft, P2ft, immo2fthes] = imm_filter( immo2t );
plot( x2fl(1), x2fl(2), 'bs', 'LineWidth', 2 );
plot( x2ft(1), x2ft(2), 'gx', 'LineWidth', 2 );

draw_axes( fig );
% disp( 'Press spacebar again to quit.' );
title( 'Step 2, Filtered Estimates' );
nameout = sprintf( '%s_5_filt.eps', name );
save_fig( fig, nameout );
% pause;
delete( fig );

end
```

tests/gen_figures.m

```
close all
clear all
clc

kf_test
evaluate_state_test
imm_test
imm_test2
distance_test

close all
clear all
clc
```

tests/imm_test.m

```
close all
clear all
clc

added_path = genpath( '../' );
addpath( added_path );

nummodes = 5;
T = 0.01;
NSec = 0.35;
% NSec = 0.5;
turnrate = 1.42;

kobj = struct( ...
  'x', [10; 10; 0; 10], ...
  'P', diag( [0.03, 0.03, 0.003, 0.003] ), ...
  'A', get_turn_matrix( 0, 0.5 ), ...
  'Q', diag( [.1, .1, .1, .1] ), ...
  'z', [10.5; 10.5], ...
  'H', [1, 0, 0, 0; 0, 1, 0, 0], ...
  'R', [.3, .3] );

immo = struct( ...
  'modes', repmat( kobj, nummodes, 1 ), ...
  'tpm', ones(nummodes) / (nummodes), ...
  'mus', ones([nummodes, 1]) / (nummodes) );

```

```
ws = get_ws( nummodes, turnrate );

for( i = 1:(nummodes) )
  immo.modes(i).A = ...
    get_turn_matrix( ws(i), T );
end

immobj_test( immo );

fig = figure();
hold on;
axis equal;
grid on;
for( i = 1:length(immo.mus) )
  xo = immo.modes(i).x;
  for( k = 1:round(NSec/T) )
    alpha = k / round(NSec/T);
    xn = immo.modes(i).A * xo;
    plot( [xo(1), xn(1)], [xo(2), xn(2)], ...
        'Color', alpha*[1,1,1] );
    xo = xn;
  end
  plot( xo(1), xo(2), 'go', ...
    'LineWidth', 1.5 );
  xoo = xo;
  for( j = 1:length(immo.mus ) )
    xo = xoo;
    for( k = 1:round(NSec/T) )
      alpha = k / round(NSec/T);
      xn = immo.modes(j).A * xo;
      plot( [xo(1), xn(1)], [xo(2), xn(2)], ...
        'Color', alpha*[1,1,1] );
      xo = xn;
    end
    plot( xo(1), xo(2), 'go', ...
      'LineWidth', 1.5 );
  end
end

s = 'Possible Two-Step Trajectories When Using';
s = [s, ' %d Modes of Motion'];
title( sprintf( s, nummodes ) );
xlabel( 'x (m)' );
ylabel( 'y (m)' );

name = '../saves/two_step.eps';
save_fig( fig, name );
close( fig );

rmpath( added_path );
```

tests/imm_test2.m

```

close all
clear all
clc

added_path = genpath( '../' );
addpath( added_path );

nummodes = 5;
Tdraw = 0.01;
Tsamp = 0.5;
NSec = 0.5;

zold = [8; 13];
znew = [2; 12];

x0 = [10; 10; 0; 10];

turnrate = 1.4;

Adraws = {};

%  Set up an initial kalman filter object
kobj = struct( ...
  'x', x0, ...
  'P', diag( [0.03, 0.03, 0.003, 0.003] ), ...
  'A', get_turn_matrix( 0, 0.5 ), ...
  'Q', diag( [.1, .1, .1, .1] ), ...
  'z', zold, ...
  'H', [1, 0, 0, 0; 0, 1, 0, 0], ...
  'R', diag([.3, .3]) );

%  Create an imm object that repeats the
%  kalman filter object nummodes times.
immo = struct( ...
  'modes', repmat( kobj, nummodes, 1 ), ...
  'tpm', init_tpm( nummodes, nummodes^2 ), ...
  'mus', ones([nummodes, 1]) / (nummodes) );

%  set the turn radii for each of the modes
%  of the imm object
ws = get_ws( nummodes, turnrate );

%  immo.modes(i).A is the dynamics matrix that
%  advances the state by Tsamp ( = 0.5 ) sec.
%  Adraws advances the state by Tdraw << Tsamp.
for( i = 1:(nummodes) )
  Adraws{i} = ...
    get_turn_matrix( ws(i), Tdraw );
  immo.modes(i).A = ...
    get_turn_matrix( ws(i), Tsamp );
end

immobj_test( immo );

%  Plot the five possible motion trajectories
%  along with the imm filtered result
fig = figure();
hold on;
axis equal;
grid on;
for( i = 1:length(immo.mus) )
  xo = immo.modes(i).x;
  for( k = 1:round(NSec/Tdraw) )
    alpha = k / round(NSec/Tdraw);
    xn = Adraws{i} * xo;
    plot( [xo(1), xn(1)], [xo(2), xn(2)], ...
        'Color', alpha*[1,1,1] );
    xo = xn;
  end
```

```
  plot( xo(1), xo(2), 'go', ...
    'LineWidth', 2 );
end

[xkp1, Pkp1, immo2] = imm_filter( immo );
plot( kobj.z(1), kobj.z(2), 'ro', ...
  'LineWidth', 2 );

for( i = 1:length(immo2.mus) )
  x = immo2.modes(i).x;
  plot( x(1), x(2), 'bx', ...
    'LineWidth', 2 );
end

plot( xkp1(1), xkp1(2), 'bo', ...
  'LineWidth', 2, 'MarkerSize', 10 );

title( 'First Step Filtered Result' );
xlabel( 'x (m)' );
ylabel( 'y (m)' );

axis tight;
axis equal;

name = '../saves/one_step_filtered.eps';
save_fig( fig, name );

%  Load in a new measurement and plot the
%  five models' states along with the mixed
%  imm result.
for( i = 1:nummodes )
  immo2.modes(i).z = znew;
end

plot( znew(1), znew(2), 'ro', 'LineWidth', 2 );
[xkp2, Pkp2, immo3] = imm_filter( immo2 );
for( i = 1:length(immo3.mus) )
  x = immo3.modes(i).x;
  plot( x(1), x(2), 'bx', ...
    'LineWidth', 2 );
end
plot( xkp2(1), xkp2(2), 'bo', ...
  'LineWidth', 2, 'MarkerSize', 10 );
plot( [xkp1(1), xkp2(1)], ...
      [xkp1(2), xkp2(2)], 'b' );
plot( [zold(1), znew(1)], ...
      [zold(2), znew(2)], 'r' );
axis tight;
axis equal;
title( 'Second Step Filtered Result' );
name = '../saves/two_step_filtered.eps';
save_fig( fig, name );

%  After the second time step, predict the
%  state at the next time step.
[xkp3, Pkp3, immo3p] = imm_predict( immo3 );
for( j = 1:numel(immo3p.mus) )
  x = immo3p.modes(j).x;
  plot( x(1), x(2), 'bx', ...
    'LineWidth', 2 );
end
plot( xkp3(1), xkp3(2),'bs', ...
  'LineWidth', 2, 'MarkerSize', 10 );
plot( [xkp2(1), xkp3(1)], ...
      [xkp2(2), xkp3(2)], 'b--' );
axis tight;
axis equal;
title( 'Third Step Predicted' );
name = '../saves/three_step_predicted.eps';
save_fig( fig, name );
close( fig );

rmpath( added_path );
```

tests/kf_test.m

```
close all
clear all
clc

addedpath = genpath( '../' );
addpath( addedpath );

kobj = struct( ...
  'x', [10; 1], ...
  'P', [.03, 0; 0, .005], ...
  'A', [1, 1; 0, 1], ...
  'Q', diag( [.1; .1] ), ...
  'z', [13.7], ...
  'H', [1, 0], ...
  'R', [.1] );
kobj1 = kobj;
kobj2 = kobj;
kobj2.A = [1, 7; 0, 1];
kobj2.Q = diag( [.1; .1] / 7 );

[kobj1.x, kobj1.P, l] = kalman_filter( kobj1 );
[kobj2.x, kobj2.P, l] = kalman_filter( kobj2 );

xstep = .01;
xrange = 7:xstep:20;
yrange = 0:xstep:2;

%  %
%  Begin fig1
%  %

kobjs = {kobj1, kobj2};

fig = figure;
hold on;
grid on;

titleinit = ...
  'Initial, Predicted, and Updated Estimates';
titlestring = ...
  { titleinit, ...
   [titleinit, ...
    ' for Two Different Dynamics Models'] };


for( modelnumber = 1:2 )

```

```
kobjn = kobjs{modelnumber};

pk = zeros( size( xrange ) );
pkp1 = zeros( size( xrange ) );
pz = zeros( size( xrange ) );

xkp1e = kobjn.A * kobj.x;
Pkp1e = kobjn.A * kobj.P * kobjn.A' + kobjn.Q;
pkp1e = zeros( size( xrange ) );

for( i = 1:numel(xrange) )
  pk(i) = gauss( xrange(i), kobj.x(1), ...
                            kobj.P(1,1) );
  pkp1e(i) = gauss( xrange(i), xkp1e(1), ...
                            Pkp1e(1,1) );
  pkp1(i) = gauss( xrange(i), kobjn.x(1), ...
                            kobjn.P(1,1) );
  pz(i) = gauss( xrange(i), kobjn.z, kobjn.R );
end

gauss( kobjn.z, kobjn.x(1), kobjn.P(1,1) );

plot( xrange, pk, 'k', 'LineWidth', 1.5 );
plot( xrange, pkp1e, 'g', 'LineWidth', 1.5 );
plot( xrange, pz, 'r', 'LineWidth', 1.5 );
plot( xrange, pkp1, ...
  'Color', [1, .6, 0], 'LineWidth', 1.5 );

h = legend( '$\hat{x}_{~~0}$', ...
  '$\hat{x}_{~~1|0}$', ...
  '$z_{~~1}$', ...
  '$\hat{x}_{~~1|1}$', ...
  'Location', 'best' );
set( h, 'Interpreter', 'Latex' );

xlabel( 'x' );
ylabel( 'P(x)' );
title( titlestring{modelnumber} );

name = sprintf( '../saves/kf_%d.eps', ...
                modelnumber );
save_fig( fig, name );
end

close( fig );
rmpath( addedpath );
```

tests/tpm_test.m

```
close all
clear all
clc

added_path = genpath( '../math/' );
addpath( added_path );

A = [ 2, 1, 1;
      0, 0, 0;
      0, 0, 0 ];

make_tpm( A )

A(3,1) = 3;

make_tpm( A )

rmpath( added_path );
```

tests/turn_test.m

```
close all
clear all
clc

added_path = genpath( '../math/' );
addpath( added_path );

N = 20;

x = [12; 15; 0; 10];
xs = zeros( [4, N] );
xs(:,1) = x;

A = get_turn_matrix( -2*pi/1.53, 0.5 );
a = 1/10;

fig = figure;
hold on;

for( i = 1:N )
if( i > 1 )
xs(:,i) = A*xs(:,i-1);
plot( xs(1,i), xs(2,i), 'ko' );
else
plot( xs(1,1), xs(2,1), 'go', 'LineWidth', 2 );
end
plot( xs(1,i)+a*[0,xs(3,i)], ...
      xs(2,i)+a*[0,xs(4,i)], 'r' );
end

plot( xs(1,:), xs(2,:), 'Color', [1,1,1]*.8 );

grid on;
axis equal;

rmpath( added_path );
```

tests/z_test.m

```
close all
clear all
clc

added_path = genpath( '../math/' );
addpath( added_path );

N = 2000;
z = [3; 7];

zs = zeros( [N, numel(z)] );
for( i = 1:N )
  zs(i,:) = get_z( z )';
```

```
end

plot( zs(:,1), zs(:,2), 'b.' );
axis equal;

disp( mean( zs ) );
disp( cov( zs ) );

rmpath( added_path );
```

update/check_screenshot.m

```
function check_screenshot( fig )

handles = guidata( fig );

key_p = key_downed( fig, {'p'} );

if( handles.paused && key_p )
  if( handles.filter_running )
    l = legend( handles.filters_ph, ...
                handles.filters_names );
  end
  s = get_datestr;
%    disp( 'Saving Screenshot' );
  name = ['saves/', s, '.eps'];
  save_fig( fig, name );
%    disp( 'Done!' );
end
```

update/draw_axes.m

```
function draw_axes( fig )

handles = guidata( fig );

x = handles.target_x;
y = handles.target_y;

if( handles.offset_view )
  v = handles.target_v;
  t = handles.target_theta;
  alpha = handles.offset_amount;
  x = v * cos(t) * alpha + x;
  y = v * sin(t) * alpha + y;
end

viewwidth = handles.viewwidth;

xl = x - viewwidth;
xr = x + viewwidth;
yd = y - viewwidth;
yu = y + viewwidth;

xticks = round(xl):round(xr);
xticks = xticks( find( mod( xticks, 2 ) ) );

yticks = round(yd):round(yu);
yticks = yticks( find( mod( yticks, 2 ) ) );

axis equal;
axis( [xl, xr, yd, yu] );
grid on;
set( gca, 'xtick', xticks );
set( gca, 'ytick', yticks );
```

update/draw_driver.m

```
function draw_driver( fig )

hold on;
draw_measurement( fig );
draw_target( fig );
draw_view( fig );
```

update/draw_filter.m

```
function draw_filter( fig )

handles = guidata( fig );
if( handles.filter_running )

  lw = 1;

  nmeas = length( handles.filter_zs );
  x0count = handles.filter_init_count;

  offset = - x0count;

  zlow = handles.count_meas_low + offset;
  zcount = handles.count_meas + offset;
  zhigh = handles.count_meas_high + offset;

  zlow = max( 1, zlow );
  zcount = max( 1, zcount );
  zhigh = max( 1, zhigh );

  zlow = min( nmeas, zlow );
  zcount = min( nmeas, zcount );
  zhigh = min( nmeas, zhigh );

  if( zlow <= 1 )
    x0 = handles.filter_x0;
    plot( x0(1), x0(2), 'ro', 'MarkerSize', 12 );
  end

%  Suppose zlow = 10, zcount = 11, zhigh = 13
%  zlow:zhigh = 10:13.
%  zlt = 1
%  zct = 2 = zcount - zlow + 1
%  zht = 4 = zhigh - zlow + 1

  zlt = 1;
  zct = zcount - zlow + 1;
  zht = zhigh - zlow + 1;

  zn = handles.filter_zn(zlow:zhigh,:);
  plot( zn(:,1), zn(:,2), 'ro' );
  plot( zn(:,1), zn(:,2), 'r' );

  ph = [];

```

```
  for( i = 1:handles.num_filters )
    xs = cat( 2, ...
           handles.filters_e(i,zlow:zhigh).x )';
    plot( xs(:,1), xs(:,2), 'x', ...
           'LineWidth', lw, ...
           'Color', handles.filters_colors{i} );
    ph(i) = plot( xs(:,1), xs(:,2), ...
           'LineWidth', lw, ...
           'Color', handles.filters_colors{i} );

    xp = cat( 2, ...
           handles.filters_p(i,zlow:zhigh).x )';
    plot( xp(zct,1), xp(zct,2), 's', ...
           'LineWidth', lw, ...
           'Color', handles.filters_colors{i} );
    plot( [xs(zct,1), xp(zct,1)], ...
           [xs(zct,2), xp(zct,2)], '--', ...
           'LineWidth', lw, ...
           'Color', handles.filters_colors{i} );
  end

  handles.filters_ph = ph;
end

guidata( fig, handles );
```

update/draw_measurement.m

```
function draw_measurement( fig )

handles = guidata( fig );

x = handles.meas_last_x;
y = handles.meas_last_y;

te = etime( clock, handles.T_last_measurement );
T = handles.T_measurement;
alpha = exp( - 3 * te / T );

plot( x, y, 'o', 'Color', [.25, 1, .25], ...
      'MarkerSize', 16*alpha, ...
      'LineWidth', 2 );
```

update/draw_objects.m

```
function draw_objects( fig )

hold on;
draw_measurement( fig );
draw_target( fig );
draw_view( fig );
```

update/draw_obstacles.m

```
function draw_obstacles( fig )

handles = guidata( fig );

draw_obstacle_centers = true;
if( isfield( handles, 'filter_running' ) )
  if( handles.filter_running )
    draw_obstacle_centers = false;
  end
end

if( draw_obstacle_centers )
  x_obs = handles.obstacles_x;
  y_obs = handles.obstacles_y;
  plot( x_obs, y_obs, 'rx' );
end

x_marks = handles.obstacle_marks_x;
y_marks = handles.obstacle_marks_y;
plot( x_marks, y_marks, 'k.' );
```

update/draw_passenger.m

```
function draw_passenger( fig )

hold on;

draw_obstacles( fig );
draw_tracks( fig );
draw_target( fig );

draw_filter( fig );

draw_axes( fig );
```

update/draw_target.m

```
function draw_target( fig )

handles = guidata( fig );

x = handles.target_x;
y = handles.target_y;
theta = handles.target_theta;

A = 1;

xt = x + A*cos(theta);
yt = y + A*sin(theta);

plot( x, y, 'bo', 'LineWidth', 2, ...
      'MarkerFaceColor', [1,1,1] );
plot( xt, yt, 'b.', 'MarkerSize', 12 );

xs = handles.obstacles_x;
ys = handles.obstacles_y;
rs = handles.obstacles_r;

alpha = .2;

draw_obstacle_dists = true;
if( isfield( handles, 'filter_running' ) )
  if( handles.filter_running )
    draw_obstacle_dists = false;
  end
end

if( draw_obstacle_dists )
```

```
  for( i = 1:numel(rs) )
    dx = xs(i) - x;
    dy = ys(i) - y;
    d = sqrt(dx^2 + dy^2) - rs(i);
    theta = atan2( dy, dx );
    xp = x + d*alpha*cos(theta);
    yp = y + d*alpha*sin(theta);

    if( d < 0 )
      plot( xp, yp, 'rx', 'LineWidth', 2, ...
        'MarkerSize', 12 );
    else
      plot( xp, yp, 'r.', 'MarkerSize', 10 );
    end
  end
end
```

update/draw_tracks.m

```
function draw_tracks( fig )

handles = guidata( fig );

xm = handles.truth_meas{2};
ym = handles.truth_meas{3};
ml = handles.count_meas_low;
mh = handles.count_meas_high;

xs = handles.truth_state{2};
ys = handles.truth_state{3};
sl = handles.count_state_low;
sh = handles.count_state_high;

if( ~handles.filter_running )
  plot( xm, ym, 'go' );
end
plot( xs(sl:sh),ys(sl:sh), 'Color', .4*[1,1,1] );
```

update/draw_view.m

```
function draw_view( fig )

handles = guidata( fig );

s = ['Spacebar Accelerates.  ', ...
  'Press the Escape Key to Exit.'];
xlabel( s );
angle = handles.target_theta * 180 / pi;
s = sprintf( 'Velocity: %3.2f m/s at %.0f deg', ...
  handles.target_v, angle );
title( s );

draw_obstacles( fig );
draw_axes( fig );
```

update/save_measurement.m

```
% %
% T_start
% ...
% T_last_measurement = tlastm
% target_T_prev = tprev
% T_last_measurement + T_measurement = tnowm
% T_last_draw = tlastd
% %
function save_measurement( fig )

handles = guidata( fig );

tlastm = handles.T_last_measurement;
tprev = handles.target_T_prev;
tlastd = handles.T_last_draw;

tenum = etime( tlastd, tlastm );

if( tenum > handles.T_measurement )

  tnowm = addseconds( tlastm, ...
                      handles.T_measurement );

  alpha = etime( tnowm, tprev ) / ...
          etime( tlastd, tprev );

%    s = ['Time: %0.3f, tprev: %0.3f,', ...
%         ' tnowm: %0.3f, tlastd: %0.3f, ', ...
%         'alphanum: %0.3f, alphaden: %0.3f\n'];
%    fprintf( s, ...
%      etime( clock, handles.T_start ), ...
%      etime( tprev, handles.T_start ), ...
%      etime( tnowm, handles.T_start ), ...
%      etime( tlastd, handles.T_start ), ...
%      etime( tnowm, tprev), ...
%      etime( tlastd, tprev) );

  x1 = handles.target_x_prev;
  y1 = handles.target_y_prev;
  v1 = handles.target_v_prev;
  theta1 = handles.target_theta_prev;

  xd1 = v1*cos(theta1);
  yd1 = v1*sin(theta1);

  x2 = handles.target_x;
  y2 = handles.target_y;
  v2 = handles.target_v;
  theta2 = handles.target_theta;

  xd2 = v2*cos(theta2);
  yd2 = v2*sin(theta2);

  x = (1-alpha)*x1 + alpha*x2;
  y = (1-alpha)*y1 + alpha*y2;
  xd = (1-alpha)*xd1 + alpha*xd2;
  yd = (1-alpha)*yd1 + alpha*yd2;

  fprintf( handles.measurement_savefile, ...
    '%03.3f,%f,%f,%f,%f\n', ...
    etime( tnowm, handles.T_start ), ...
    x, ...
    y, ...
    xd, ...
    yd );

```

```
  handles.T_last_measurement = tnowm;
  handles.meas_last_x = handles.target_x;
  handles.meas_last_y = handles.target_y;
end

guidata( fig, handles );
```

update/save_state.m

```
function save_state( fig )

handles = guidata( fig );

te = etime( handles.T_last_draw, ...
            handles.T_start );
handles.T_since_start = te;

handles.count = handles.count + 1;

fprintf( handles.state_savefile, ...
  '%03.3f,%f,%f,%f,%f,%f\n', ...
  te, ...
  handles.target_x, ...
  handles.target_y, ...
  handles.target_v, ...
  handles.target_theta, ...
  handles.target_dtheta );

guidata( fig, handles );
```

update/update_camera_counts.m

```
function update_camera_counts( fig )

handles = guidata( fig );
xwindow = handles.T_window;
zwindow = handles.samp_time / 2;

%  %
%  calculate counters
%  %

zcount = find_adjacent_count( ...
  handles.T_simulation, ...
  handles.count_meas, ...
  handles.truth_meas{1} );
zcount_low = find_adjacent_count( ...
  handles.T_simulation - 3*zwindow, ...
  handles.count_meas, ...
  handles.truth_meas{1} );
zcount_high = find_adjacent_count( ...
  handles.T_simulation + 1*zwindow, ...
  handles.count_meas, ...
  handles.truth_meas{1} );

xcount = find_adjacent_count( ...
  handles.T_simulation, ...
  handles.count_state, ...
  handles.truth_state{1} );
xcount_low = find_adjacent_count( ...
  handles.T_simulation - xwindow, ...
  handles.count_state, ...
  handles.truth_state{1} );
xcount_high = find_adjacent_count( ...
  handles.T_simulation + xwindow, ...
  handles.count_state, ...
  handles.truth_state{1} );

%  %
%  save counters
%  %

handles.count_meas = zcount;
handles.count_state = xcount;

handles.count_meas_low = zcount_low;
handles.count_meas_high = zcount_high;

handles.count_state_low = xcount_low;
handles.count_state_high = xcount_high;

guidata( fig, handles );
```

update/update_camera_time.m

```
function update_camera_time( fig )

handles = guidata( fig );

left = key_down( fig, {'leftarrow'} );
right = key_down( fig, {'rightarrow'} );
up = key_down( fig, {'uparrow'} );
down = key_down( fig, {'downarrow'} );
space = key_downed( fig, {'space'} );
key_t = key_downed( fig, {'t'} );
key_s = key_downed( fig, {'s'} );

key_f = key_downed( fig, {'f'} );
key_i = key_downed( fig, {'i'} );
key_z = key_downed( fig, {'z'} );

pgup = key_down( fig, {'pageup'} );
pgdn = key_down( fig, {'pagedown'} );
home = key_downed( fig, {'home'} );

T_last = handles.T_last_draw;
T = etime( clock, T_last );

handles.viewwidth = 5*T * (pgdn - pgup) + ...
  handles.viewwidth;

if( home )
  handles.viewwidth = handles.viewwidth0;
end

handles.T_simulation_mult = ...
  T * (up - down) + ...
  handles.T_simulation_mult;

Tmult = exp( handles.T_simulation_mult );

if( space > 0 )
```

```matlab
  handles.paused = ~handles.paused;
end

if( ~handles.paused )
handles.T_since_start = T + ...
  handles.T_since_start;

handles.T_simulation = handles.T_simulation + ...
  T * Tmult;
else
handles.T_simulation = ...
  handles.T_simulation + ...
  T * ( right - left ) * Tmult;

  if( key_i )
    guidata( fig, handles );
    draw_passenger( fig );
    xlabel( 'Filtering!' );
    drawnow;
    init_filter( fig );
    handles = guidata( fig );
    report_results( fig );
  end

  if( key_z )
    update_filter( fig );
    handles = guidata( fig );
  end
end

if( handles.T_simulation < ...
    handles.T_simulation_min )
      handles.T_simulation = ...
      handles.T_simulation_min;
elseif( handles.T_simulation > ...
        handles.T_simulation_max )
  handles.T_simulation = ...
    handles.T_simulation_max;
  handles.paused = true;
end

if( key_t )
  if( ~handles.filter_running )
    handles.T_simulation = ...
      handles.T_simulation_min;
  else
    handles.T_simulation = ...
      handles.filter_t0;
  end
end

if( key_s )
  handles.T_simulation_mult = 0;
end

if( key_f )
  if( isfield( handles, 'filter_x0' ) )
    handles.filter_running = ...
      ~handles.filter_running;
  end
end

s = sprintf( ...
  ['%2.2f / %2.2f sec,', ...
  ' multiplier = %2.2f x'], ...
  handles.T_simulation, ...
  handles.T_simulation_max, ...
  Tmult );
title( s );

handles.T_last_draw = clock;
guidata( fig, handles );
```

update/update_camera_view.m

```matlab
function update_camera_view( fig )

handles = guidata( fig );

xcount = handles.count_state;
zcount = handles.count_meas;

sfilter = '';
s = sprintf( ...
  'State Count: %d, Meas Count: %d%s', ...
  xcount, zcount, sfilter );
xlabel( s );
ylabel( handles.case_time );

if( xcount < handles.count_state_max )
t = handles.truth_state{1}(xcount:xcount+1);
x = handles.truth_state{2}(xcount:xcount+1);
y = handles.truth_state{3}(xcount:xcount+1);
v = handles.truth_state{4}(xcount:xcount+1);
theta = handles.truth_state{5}(xcount:xcount+1);

alpha = ( handles.T_simulation - t(1) ) / ...
        ( t(2) - t(1) );

handles.target_x = (1-alpha)*x(1) + alpha*x(2);
handles.target_y = (1-alpha)*y(1) + alpha*y(2);
handles.target_v = (1-alpha)*v(1) + alpha*v(2);
handles.target_theta = theta(1);
else

handles.target_x = ...
  handles.truth_state{2}(end);
handles.target_y = ...
  handles.truth_state{3}(end);
handles.target_v = ...
  handles.truth_state{4}(end);
handles.target_theta = ...
  handles.truth_state{5}(end);

end

guidata( fig, handles );
```

update/update_filter.m

```matlab
function update_filter( fig )

update_filter_meas( fig );
```

```

handles = guidata( fig );

if( handles.filter_running )

zn = handles.filter_zn;
nmeas = length( zn );
s1 = '';
tstart = clock;
for( i = 1:nmeas )
  s2 = sprintf( '%.3f', i / nmeas );
  print_progress( s1, s2 );
  s1 = s2;

  z = zn(i,:)';
  for( f = 1:handles.num_filters )
    % get the imm object
    immo = handles.filters_immos(f);

    % filter the imm object
    immo = handles.filters_funs{f}( fig,immo,z );

    % save the immm object after filtering
    handles.filters_immos(f) = immo;

    % save the imm object's parameters
    handles.filters_e(f,i).x = immo.x;
    % handles.filters_e(f,i).mus = immo.mus;
    % handles.filters_e(f,i).P = immo.P;
    % handles.filters_e(f,i).tpm = immo.tpm;

    % predict the imm object
    immo = handles.filters_funs{f}( fig, immo );

    % save the predicted result's parameters
    handles.filters_p(f,i).x = immo.x;
    % handles.filters_p(f,i).mus = immo.mus;
    % handles.filters_p(f,i).P = immo.P;
    % handles.filters_p(f,i).tpm = immo.tpm;
  end
end
tend = clock;
%   s2 = sprintf( ...
%      'Filtered %d Measurements\n', nmeas );
  s2 = '';
  print_progress( s1, s2 );
%   delta_t = etime( tend, tstart );
%   s = '';
%   NF = handles.num_filters;
%   if( NF > 1 ) s = 's'; end
%   fprintf( ['using %d filter%s\n', ...
%               'in %.3f Seconds.\n', ...
%               '(%.3f Seconds/Measurement)\n'], ...
%      NF, s, delta_t, delta_t/nmeas );
end

% fprintf( '\n' );

guidata( fig, handles );
```

update/update_filter_meas.m

```
function update_filter_meas( fig )

handles = guidata( fig );

if( handles.filter_running )

zcount = handles.count_meas - ...
         handles.filter_init_count + 1;

if( zcount < 1 )
  zcount = 1;
end

zs = handles.filter_zs;
nmeas = length( zs );

zn = zs;
if( isfield( handles, 'filter_zn' ) )
  zn = handles.filter_zn;
end
zstrun = zs( zcount:end, : );

v = 0.1;
if( isfield( handles, 'zrhalf' ) )
  v = handles.zrhalf;
end
zntrun = get_z( zstrun', v )';

zn( zcount:end, : ) = zntrun;
handles.filter_zn = zn;

end

guidata( fig, handles );
```

update/update_key_changes.m

```
function update_key_changes( fig )

handles = guidata( fig );
old_statuses = handles.key_statuses;

update_key_statuses( fig );
handles = guidata( fig );
new_statuses = handles.key_statuses;

handles.key_changes = new_statuses - old_statuses;
guidata( fig, handles );
```

update/update_key_statuses.m

```

function update_key_statuses( fig )

handles = guidata( fig );
tmin = handles.tmin;
tnow = clock;

for( i = 1:numel( handles.keys ) )
```

```
  tu = handles.key_uptimes{i};
  td = handles.key_downtimes{i};

  if( etime( td, tu ) > 0 )
    handles.key_statuses(i) = 1;
  elseif( etime( tnow, tu ) > tmin )
    if( etime( tu, td ) > 0 )
      handles.key_statuses(i) = 0;
    end
  end
end

guidata( fig, handles );
```

update/update_passenger.m

```
function update_passenger( fig )

update_camera_time( fig );
update_camera_counts( fig );
update_camera_view( fig );
```

update/update_target.m

```
function update_target( fig )

handles = guidata( fig );

left = key_down( fig, {'leftarrow'} );
right = key_down( fig, {'rightarrow'} );
up = key_down( fig, {'uparrow'} );
%  down = key_down( fig, {'downarrow'} );
down = 0;
space = key_down( fig, {'space'} );

T_last = handles.T_last_draw;
T = etime( clock, T_last );

x = handles.target_x;
y = handles.target_y;
v = handles.target_v;
dv = handles.target_dv;

theta = handles.target_theta;
dtheta = handles.target_dtheta;
ddtheta = handles.target_ddtheta;

handles.target_x_prev = x;
handles.target_y_prev = y;
handles.target_v_prev = v;
handles.target_theta_prev = theta;
handles.target_T_prev = T_last;

cv = handles.target_cv;
cdtheta = handles.target_cdtheta;

%  Acceleration forces
if( space )
  v = v + dv*T;
end

if( left == right )
elseif( left )
  dtheta = dtheta + ddtheta*T;
elseif( right )
  dtheta = dtheta - ddtheta*T;
end

if( up == down )
elseif( down )
  v = v - dv*T;
elseif( up )
%     v = v + dv*T;
end

%  Friction forces
v = v - cv*T*v;
dtheta = dtheta - cdtheta*T*dtheta;

%  Updates
theta = theta + dtheta*T;
x = x + v*cos(theta)*T;
y = y + v*sin(theta)*T;

handles.target_x = x;
handles.target_y = y;
handles.target_v = v;

if( theta > pi )
  theta = theta - 2*pi;
elseif( theta < -pi )
  theta = theta + 2*pi;
end
handles.target_theta = theta;
handles.target_dtheta = dtheta;

handles.T_last_draw = addseconds( T_last, T );

guidata( fig, handles );
```

update/update_viewing_plot.m

```
function update_viewing_plot( fig )

open_last_run( fig );

handles = guidata( fig );
set( handles.edit, 'String', handles.case_time );
guidata( fig, handles );

init_truths( fig );
init_viewing_plot( fig );

handles = guidata( fig );
if( isfield( handles, 'edit_tmin' ) )
  s = sprintf('%4.2f', handles.T_simulation_min);
  set( handles.edit_tmin, 'String', s );
end
if( isfield( handles, 'edit_tmax' ) )
  s = sprintf('%4.2f', handles.T_simulation_max);
  set( handles.edit_tmax, 'String', s );
end
```

```
if( isfield( handles, 'nruns' ) )
  s = handles.nruns;
  if( isfield( handles, 'edit_nruns' ) )
    set( handles.edit_nruns, 'String', s );
  end
end
```

utility/addseconds.m

```
% %
% v = addseconds( t, T )
% t is a date vector
% T is the time in seconds to add
% v is the resulting date vector
% %
function v = addseconds( t, T )

tnum = datenum( t );
vnum = addtodate( tnum, floor(1000*T), ...
        'millisecond' );
v = datevec( vnum );
```

utility/append_db.m

```
function append_db( s )

fname = 'runs_db.csv';
A = {''};

if( exist( fname, 'file' ) )
  fin = fopen( fname );
  A = textscan( fin, '%s' );
  A = A{1};
  fclose( fin );
end

A = unique( {A{:}, s} );
fout = fopen( fname, 'w+' );
for( i = 1:numel( A ) )
  fprintf( fout, '%s\n', A{i} );
end
fclose( fout );
```

utility/display_keys.m

```
function s = display_keys( fig )

handles = guidata( fig );

s = '';
for( i = 1:numel( handles.keys ) )
  s = sprintf( '%s  %s is %d\n', s, ...
              handles.keys{i}, ...
              handles.key_statuses(i) );
end
```

utility/enable_all.m

```
function enable_all( fig, b, state )

handles = guidata( fig );

f = fieldnames( handles );
for( i = 1:numel( f ) )
  if( strfind( f{i}, b ) )
    set( getfield( handles, f{i} ), ...
      'Enable', state );
  end
end
```

utility/find_adjacent_count.m

```
% %
% T is the current time.
%
% cin is a good guess of which index of
% ts is closest to Tnow.
%
% ts is a vector of times.
%
% The output is cout.
% cout has the property that
% ts( cout ) < Tnow < ts( cout + 1 )
% %

function cout = find_adjacent_count( T, cin, ts )

if( cin < 1 )
  cin = 1;
elseif( cin > numel( ts ) )
  cin = numel( ts )
end

t = ts(cin);

cmax = length( ts );
tmax = ts(cmax);

tn = t;
cn = cin;

% threshold = .0001;
% if( abs(T-t) < threshold )
%   T = T + 3*threshold;
%   disp( T );
% end
% disp( T );

if( T > tn )
% %
% Check if there is a next data point, and
% check to see if the current time has
% passed the current data point.
% %
  if( T > tmax )
    cin = cmax;
  else
    while( tn < T )
```

```
      if( cn < cmax )
        cn = cn + 1;
        tn = ts(cn);
      end
      if( T < tn )
        cin = cn;
        if( cin > 1 )
          cin = cin - 1;
        end
      elseif( T == tn )
        cin = cn;
      end
    end
  end
else
% %
% Check if there is a previous data point, and
% check to see if the current time has
% preceded the current data point.
% %
  if( T < ts(1) )
    cin = 1;
  else
    while( tn > T && cn > 1 )
      if( cin > 1 )
        cn = cn - 1;
        tn = ts(cn);
      end
      if( T >= tn )
        cin = cn;
      end
    end
  end
end

cout = cin;
```

utility/get_datestr.m

```
function s = get_datestr()

s = datestr( now, 'yyyy.mm.dd.HH.MM.SS' );
```

utility/get_running.m

```
function running = get_running( fig )

handles = guidata( fig );
running = handles.running;
```

utility/key_changed.m

```
function changed = key_changed( fig, keys )

numkeys = numel( keys );
handles = guidata( fig );

changed = zeros( size( keys ) );

for( i = 1:handles.numkeys )
  for( k = 1:numkeys )
    if( strcmp( handles.keys{i}, keys{k} ) )
      changed(k) = handles.key_changes(i);
    end
  end
end

```

utility/key_down.m

```
function down = key_down( fig, keys )

numkeys = numel( keys );
handles = guidata( fig );

down = zeros( size( keys ) );

for( i = 1:handles.numkeys )
  for( k = 1:numkeys )
    if( strcmp( handles.keys{i}, keys{k} ) )
      down(k) = handles.key_statuses(i);
    end
  end
end
```

utility/key_downed.m

```
function k = key_downed( fig, s )

kd = key_down( fig, s );
kc = key_changed( fig, s );

k = kd && kc;
```

utility/key_tracker_down.m

```
function key_tracker_down( evt, down, fig )

handles = guidata( fig );
key_name = evt.Key;
% disp( key_name );

for( i = 1:handles.numkeys )
  if strcmp( key_name, handles.keys{i} )
    if( down )
      handles.key_downtimes{i} = clock;
    else
      handles.key_uptimes{i} = clock;
    end
  end
end

guidata( fig, handles );
```

utility/make_results_table.m

```
function make_results_table( robj )

resdir = robj.resdir;

zes = robj.zes;
xes = robj.xes;
xps = robj.xps;
zrhalf = robj.zrhalf;

nruns =  length( zes );
NF = robj.NF;

namebase = robj.namebase;
namebasez = sprintf( '%s_%4.3f', ...
  namebase, zrhalf );

foutname = sprintf( '%s%s_mf.tex', ...
  resdir, namebasez );
figname = [namebase, '.pdf'];
tabname = [namebasez, '_tab.tex'];
tabnamef = [resdir, tabname];
nrunsname = [resdir, namebasez, '_nruns.tex'];

fout = fopen( foutname, 'w' );
fprintf( fout, ...
  '%s\n  %s\n  %s{%s}\n  %s\n%s\n', ...
  '\begin{minipage}{.45\textwidth}', ...
  '\begin{center}', ...
  '\includegraphics[width=.8\textwidth]', ...
  figname, ...
  '%\end{center}', ...
  '%\end{minipage}' );

fprintf( fout, '\\\\ \\medskip\n' );

fprintf( fout, ...
  '%s\n  %s\n  %s', ...
  '%\begin{minipage}{.45\textwidth}', ...
  '%\begin{center}', ...
  '\begin{tabular}{' );
%  for( i = 1:(NF+2) )
%    fprintf( fout, ' c' );
%  end
fprintf( fout, ' c c c' );
fprintf( fout, '%s\n%s', ...
  ' }', '    \toprule' );

names = { '\midrule Measurements', ...
  robj.names{:} };

s = {};
for( row = 1:(NF+2) )
for( col = 1:3 )
  if( col == 1 )
    if( row >= 2 )
      s{row,col} = names{row-1};
    end
  elseif( col == 2 )
    mxes = mean( xes );
    nxes = sum( mxes == min( mxes ) );
    mxps = mean( xps );
    nxps = sum( mxps == min( mxps ) );
    if( row == 1 )
      s{row,col} = sprintf( '%s', ...
        'Estimation' );
    elseif( row == 2 )
      s{row,col} = sprintf( '%s%7.5f%s', ...
        '{\bf ', mean( zes ), '}' );
    else
      s1 = '';
      s2 = '';
      x = mean( xes( :,row-2 ) );
      if( nxes == 1 )
      if( x == min( [mean( xes ), mean(zes)] ) )
        s1 = '{\bf ';
        s2 = '}';
      end
      end
      s{row,col} = sprintf( '%s%7.5f%s', ...
        s1, x, s2 );
    end
  else
    if( row == 1 )
      s{row,col} = sprintf( '%s', ...
        'Prediction' );
    elseif( row == 2 )
      s{row,col} = '-';
    else
      s1 = '';
      s2 = '';
      x = mean( xps( :,row-2 ) );
      if( nxps == 1 )
      if( x == min( mean( xps ) ) )
        s1 = '{\bf ';
        s2 = '}';
      end
      end
      s{row,col} = sprintf( '%s%7.5f%s', ...
        s1, x, s2 );
    end
  end
end
end

tfout = fopen( tabnamef, 'w' );
nrow = size(s,1);
ncol = size(s,2);
for( row = 1:nrow )
  for( col = 1:ncol )
    es = ' & ';
    if( col == ncol ) es = ' \\'; end
    fprintf( tfout, '%s%s', s{row,col}, es );
  end
  fprintf( tfout, '\n' );
end
fclose( tfout );

fprintf( fout, '\n    \\input{%s}\n', tabname );

tfout = fopen( nrunsname, 'w' );
fprintf( tfout, '%d', nruns );
fclose( tfout );
```

```
s = '\captionof{table}{';
s = sprintf( '%s Calculated Using %d Runs}', ...
  s, nruns );
fprintf( fout, ...
  '  %s\n  %s\n  %s\n  %s\n%s\n%s', ...
  '  \bottomrule', ...
  '\end{tabular}', ...
  s, ...
  '\end{center}', ...
  '\bigskip', ...
  '\end{minipage}' );


fclose( fout );
```

utility/open_last_run.m

```
function open_last_run( fig )

handles = guidata( fig );

oldfields = { ...
  'T_simulation_max', ...
  'T_simulation_min', ...
  'nruns', ...
  'zrhalf' };
for( i = 1:numel( oldfields ) )
  if( isfield( handles, oldfields{i} ) )
    handles = rmfield( handles, oldfields{i} );
  end
end

d = ',';
last_run = fopen( 'last_run.csv', 'r' );
scanned = fscanf( last_run, '%s' );
case_time = scanned;

sf = strfind( scanned, d );
if( numel( sf ) >= 1 )
  case_time = scanned( 1:(sf(1)-1) );
  scanned = scanned( (sf(1)+1):end );

  A = textscan( scanned, '%f', 'Delimiter', d );
  handles.T_simulation_min = A{1}(1);
  if( numel( sf ) >= 2 )
    handles.T_simulation_max = A{1}(2);
  end
  if( numel( sf ) >= 3 )
    handles.nruns = sprintf( '%d', A{1}(3) );
  end
  if( numel( sf ) >= 4 )
    handles.zrhalf = A{1}(4);
  end
end

handles.case_time = case_time;

guidata( fig, handles );
```

utility/print_progress.m

```
function print_progress( s1, s2 )

n = numel( s1 );
for( i = 1:n )
  fprintf( '\b' );
end

fprintf( '%s', s2 );
```

utility/query_filter.m

```
%  Nothing here anymore...
```

utility/save_fig.m

```
function save_fig( fig, name )

disp( name );

set( fig, ...
  'PaperUnits', 'inches', ...
  'PaperPosition', [0 0 6 4], ...
  'PaperPositionMode', 'manual' );

saveas( fig, name, 'epsc' );
system( ['epstopdf ', name] );
system( ['rm ', name] );
```

utility/set_addedpath.m

```
function set_addedpath( fig, addedpath )

handles = guidata( fig );
handles.addedpath = addedpath;
guidata( fig, handles );
```

utility/set_running.m

```
function set_running( fig, val )

handles = guidata( fig );
handles.running = val;
guidata( fig, handles );
```

viewing.m

```
close all
clear all
clc

addedpath = genpath( '.' );
addpath( addedpath );

fig = figure();
set_addedpath( fig, addedpath );
set( fig, 'CloseRequestFcn', ...
  'viewing_close( fig )' );
init_viewing( fig );
update_viewing_plot( fig );
```

Vita

Rastin Rastgoufard has lived in New Orleans his whole life. He spent more time obtaining his Master's degree (2012) than he did his undergraduate degree, but that was largely due to the fact that he finished his Bachelor's degree (2008) at Tulane University in only three years. He learned a lot and very much enjoyed his time in the graduate program of the University of New Orleans. He spends his free time by dancing, playing music, playing video games, programming, or studying engineering.