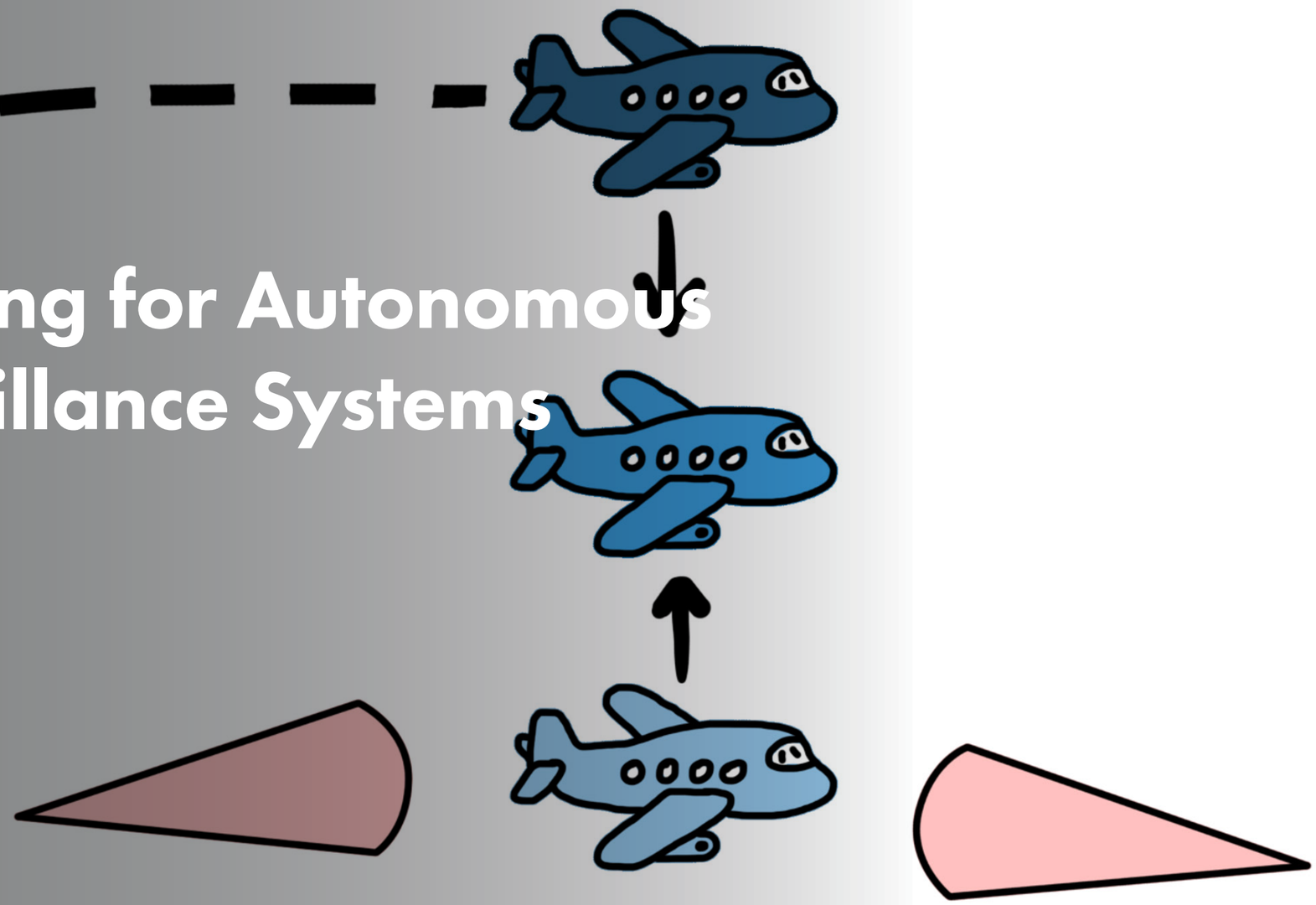
Multi-Object Tracking for Autonomous Systems and Surveillance Systems



MathWorks®

Preface

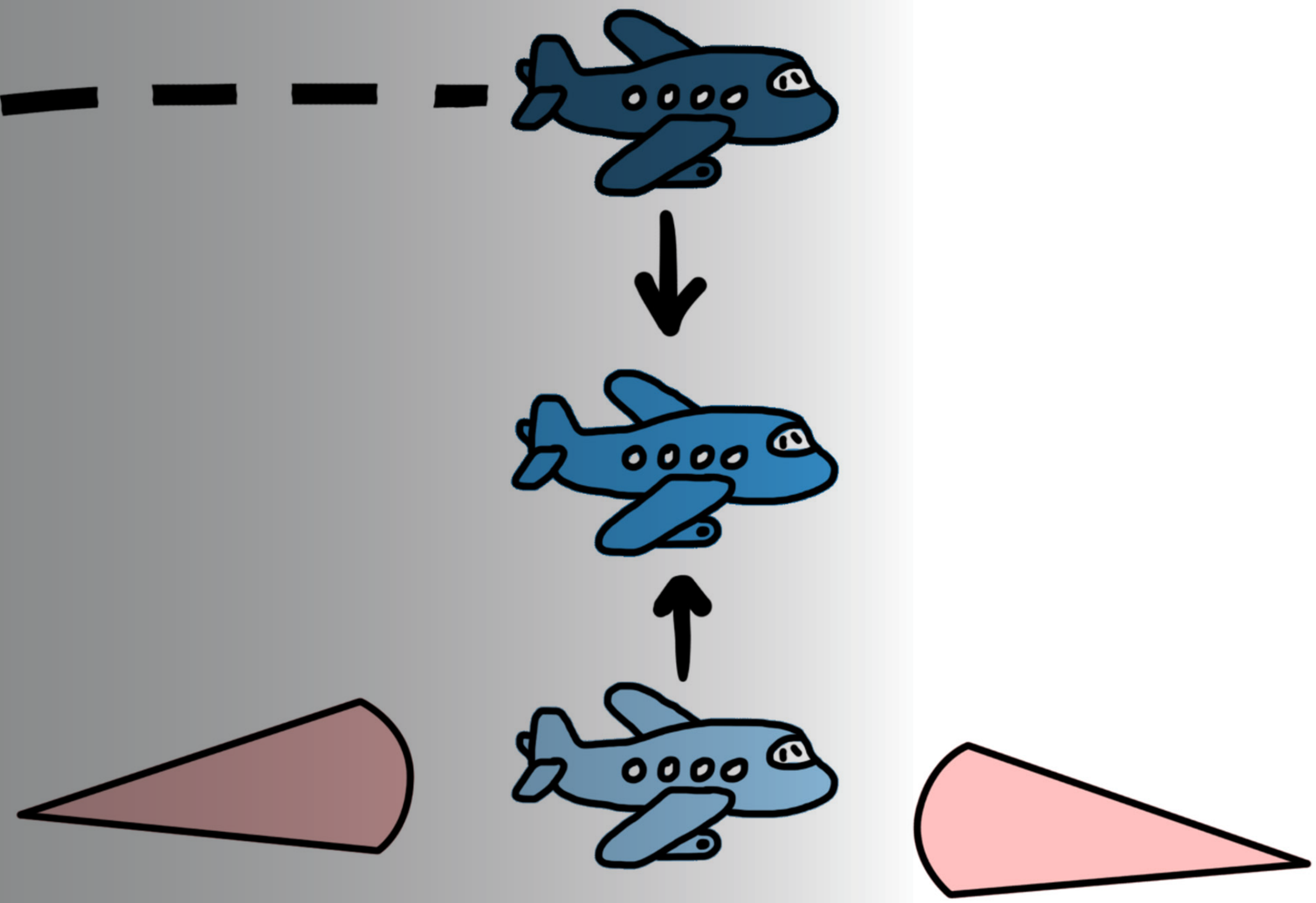
This ebook provides an overview of multi-object tracking. It covers the fundamental ideas without diving too deep into the mathematics. Hopefully, this approach makes learning the mathematics and implementing your own algorithms easier.

Topics

1. **Introduction:** Understanding tracking filters, measurement noise, prediction errors, and process noise
2. **Single-object tracking:** Using a tracker to determine position and motion of a remote object
3. **Multi-object tracking:** Overcoming the challenges of tracking several objects at once

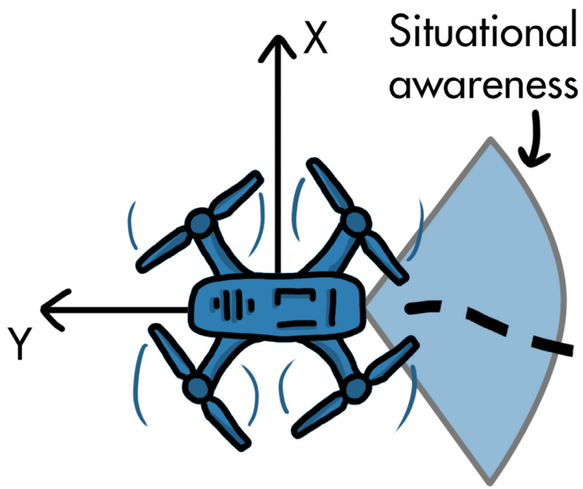
By the end you should have a solid understanding of how multi-object trackers contribute to the success of autonomous systems and surveillance systems.

Part 1: Introduction

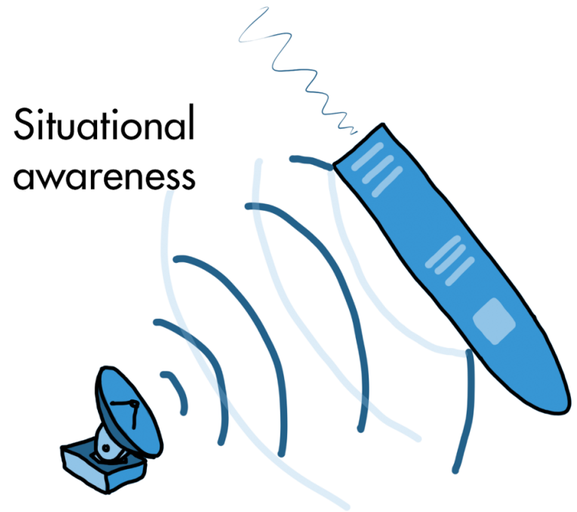


Multi-Object Tracking

Perception is a critical component of both autonomous systems and surveillance systems. Multi-object tracking and sensor fusion are at the heart of perception systems. The goal of an autonomous system is to operate within an environment without human interaction. This means that the system needs to be able to maintain situational awareness just as a surveillance system does.



Autonomous System



Surveillance System

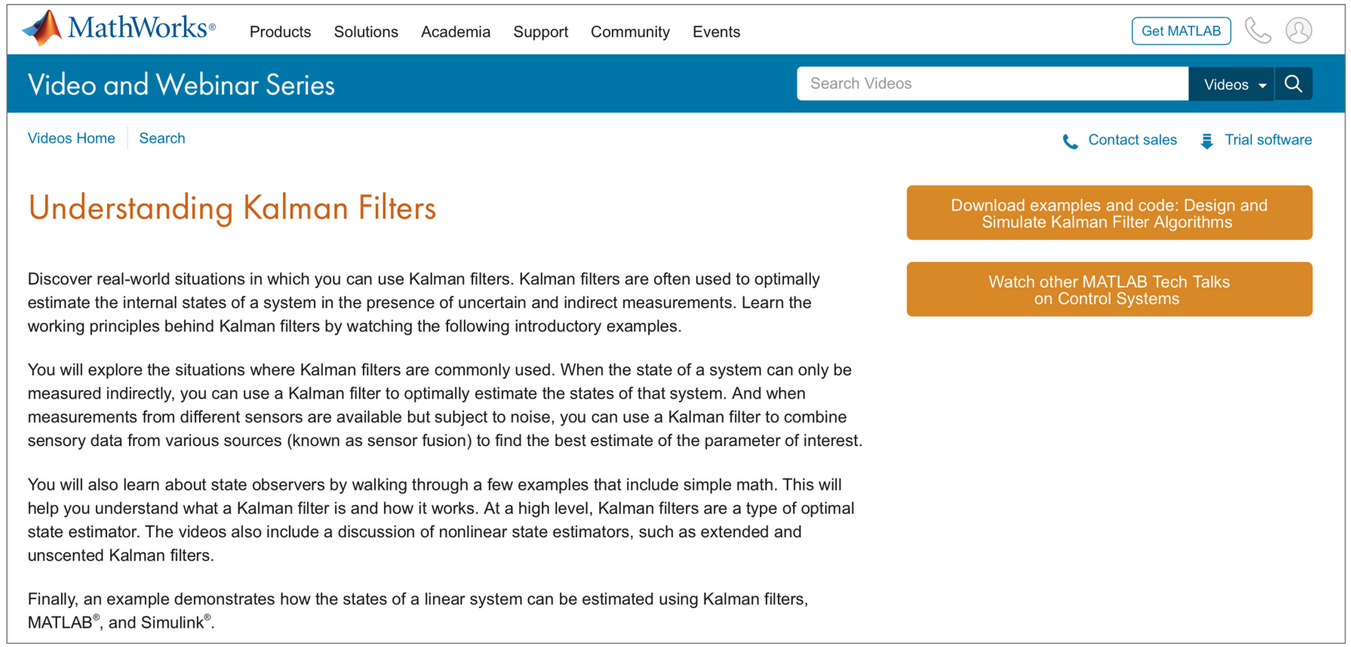
The Many Algorithms of Multi-Object Tracking

At the core of multi-object tracking is the ability to estimate the motion of each object separately.

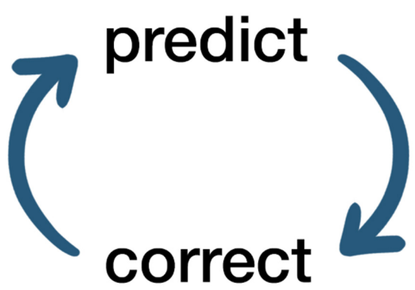
The basis for this estimation is the use of an estimation filter. Other types of estimation filters are used in tracking, and we will cover some of them, but first let's understand the most fundamental and simple filter: the Kalman filter.

We're going to approach the Kalman filter at a high level and provide some insight into how the filter is able to estimate state by combining measurements and models.

If you'd like a more in-depth explanation of the filter, check out the *Understanding Kalman Filters* video series.



The Kalman Filter



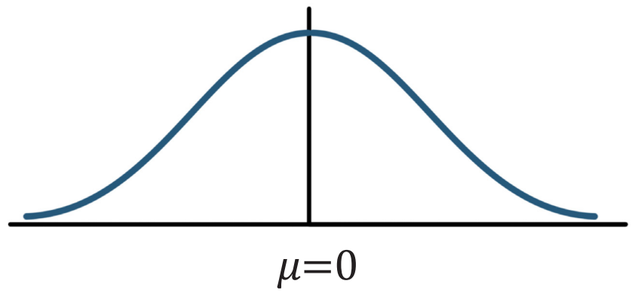
A Kalman filter is part of a class of estimation filters that use a two-step process to estimate state: prediction and correction.

If a system can be described with a linear model and the probability distribution of the process noise and measurement noise is Gaussian, then a linear Kalman filter will produce the optimal state estimation.

Linear system model

$$\dot{x}=Ax+Bu$$
$$y=Cx+Du$$

Gaussian noise distribution



In practice, though, most real systems are not truly linear and noise might not have a perfect Gaussian probability distribution, which means the result of a linear Kalman filter is rarely absolutely optimal for real systems. However, they still work very well for many real-life problems, and understanding linear Kalman filters can help you to better understand their nonlinear counterparts such as the extended Kalman filter, the unscented Kalman filter, and the particle filter.

How do we know state?

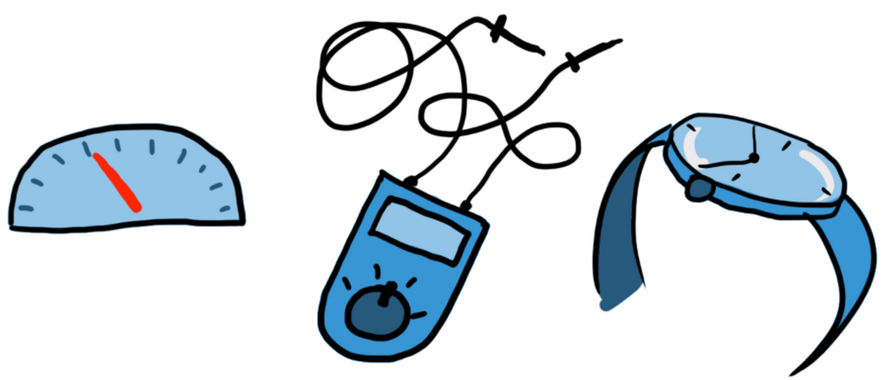
Let’s walk through a simple math approach to understanding the linear Kalman filter.

Think about how we as humans estimate the state of something. For example, how do we know the velocity of a vehicle, the voltage of an electrical circuit, or the time of day?

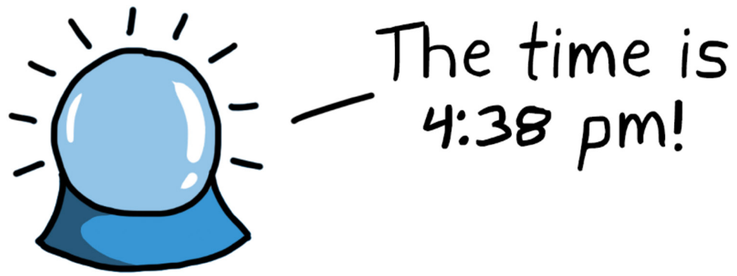
The obvious answer is that we measure it directly with a sensor like we do with a speedometer, a voltmeter, or a clock. Similarly, we may derive the state from measuring other quantities. This would be the case if we estimated velocity using the difference between two successive *position* measurements and the time difference in which they were taken.

But, as humans, we have other knowledge that we use to challenge the validity of our measurements. If your watch says it’s 4:37 p.m. and about a minute later it says 6:15 p.m., you wouldn’t trust that result. We have a general understanding of the passage of time and how the universe should behave, and this extra knowledge allows us to make a prediction of how the state of a system should change over some time horizon.

Measure



Predict

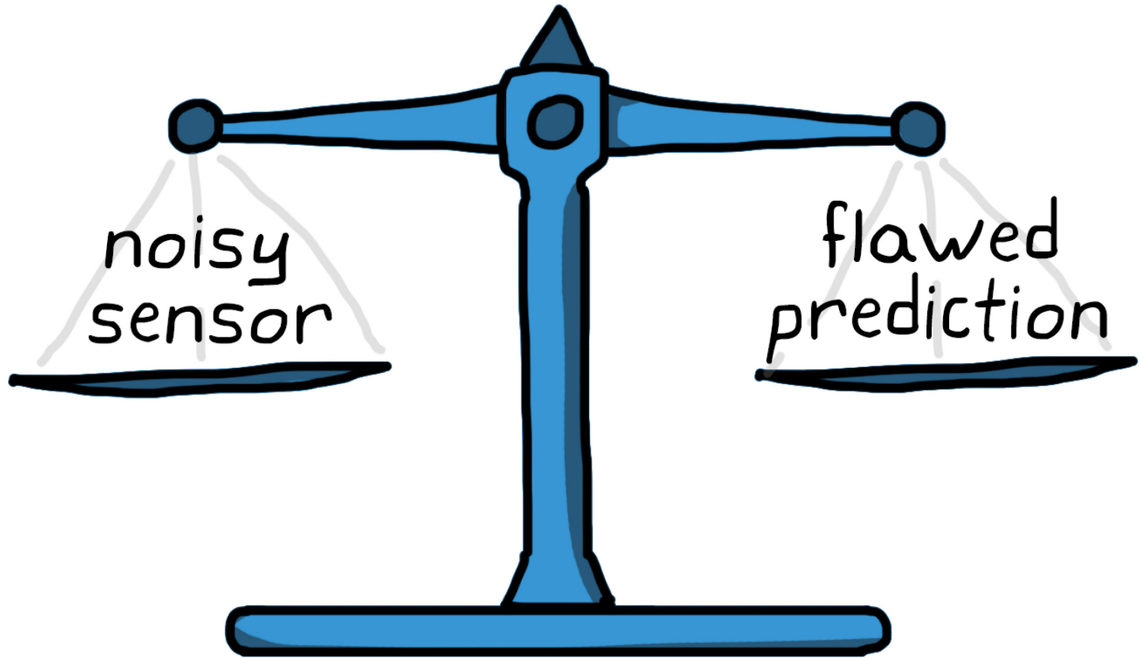


Combining a Prediction and a Measurement

Our ability to predict isn't perfect; it's subject to error. This error comes from us having the wrong mental model of the system behavior and from not knowing and accounting for every external input that can influence the system state. Therefore, it's not wise to base the state of the system solely on our prediction of what it should be. In fact, the further we predict into the future, the less certain we are in the answer. If you thought an hour had passed but according to your watch it's only been 54 minutes, then you are probably more inclined to believe your watch and discount your prediction.

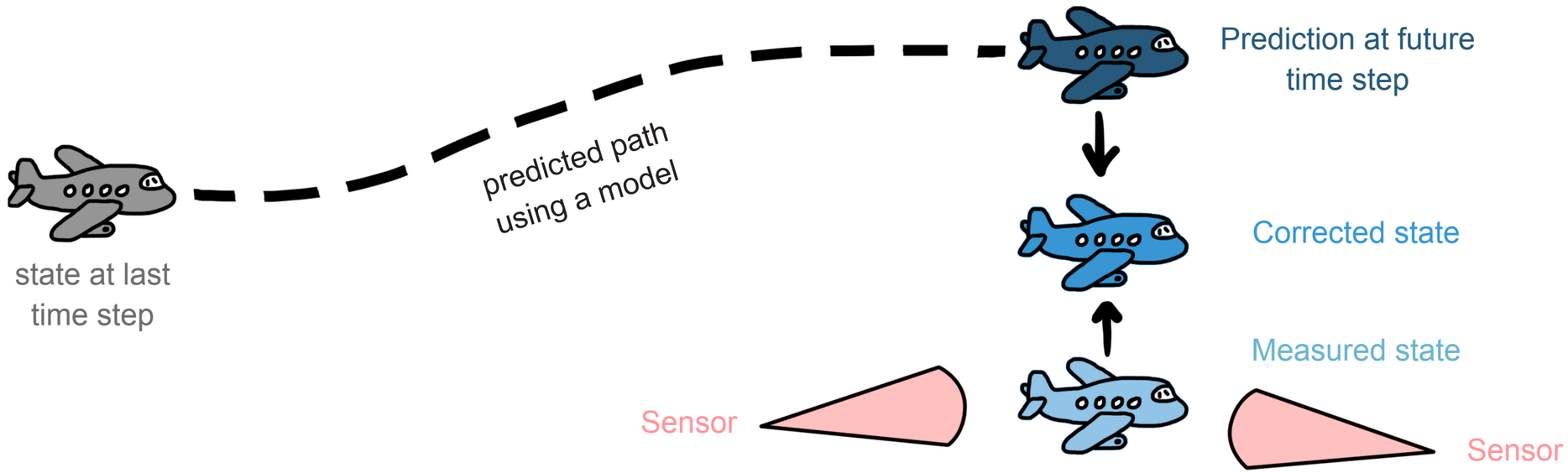
This is exactly the thought process behind how we typically balance the results from a noisy sensor and a flawed prediction to get a more accurate and more reliable estimate of system state. The further we predict into the future, the less confidence we have in the prediction and the more we trust the measurement. Predictions over shorter periods, however, are treated with more confidence and can be used to challenge a measurement from a sensor.

This is what a Kalman filter does.



How Estimation Filters Work

Kalman filters and other estimation filters can estimate the future state of a system because we give them a mathematical model. Using this model, the filter propagates the state forward each time step. At some point, the true state is measured with a noisy sensor. Now, we have a predicted state and a measured state, which are likely different. The question is, which one is correct? Well, since they both have uncertainty, the question really is, how can we combine the two based on their relative uncertainties?



A Kalman filter determines how much trust, or weight, to apply to both the prediction and the measurement so that the corrected state is placed exactly at the optimal location between the two. This balancing act hinges on a mathematical representation of uncertainty, which we get in the form of covariance.

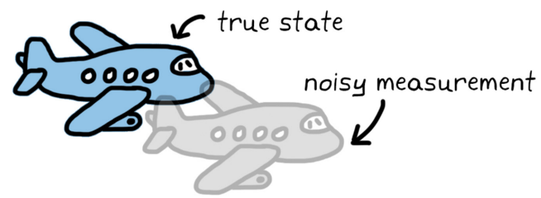
Measurement Noise

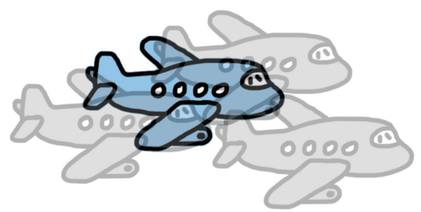
The measurement noise covariance matrix, R, captures the expected uncertainty that you have with the sensor measurements. The uncertainty in a measurement is pretty easy to understand. There is a true state that we want to know the value of, but unfortunately we have to measure it with a noisy sensor.

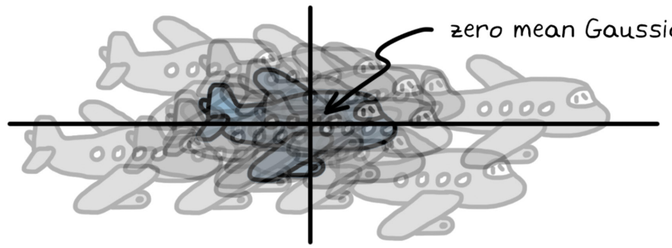
If we measured the state several times, we would see different results from the sensor due to the random noise.

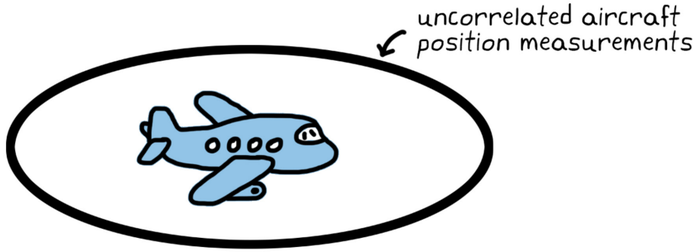
If the noise had zero mean, then the group of measurements would all be centered around the true state. If the noise was Gaussian, then the measurements would follow a normal probability distribution where the width of the distribution, or the amount of noise in the sensor, is the variance.

For example, in radar, the measurement noise will vary between range and cross-range measurements. As the range from the radar increases, the cross-range measurement noise increases as well. Understanding the accuracy of a sensor is best understood through a combination of modeling and field testing.



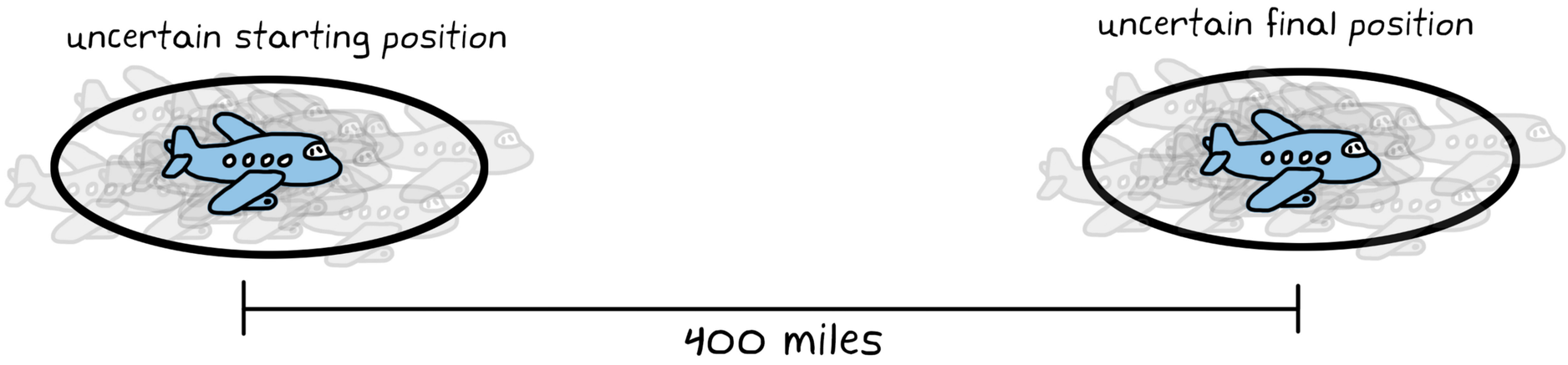






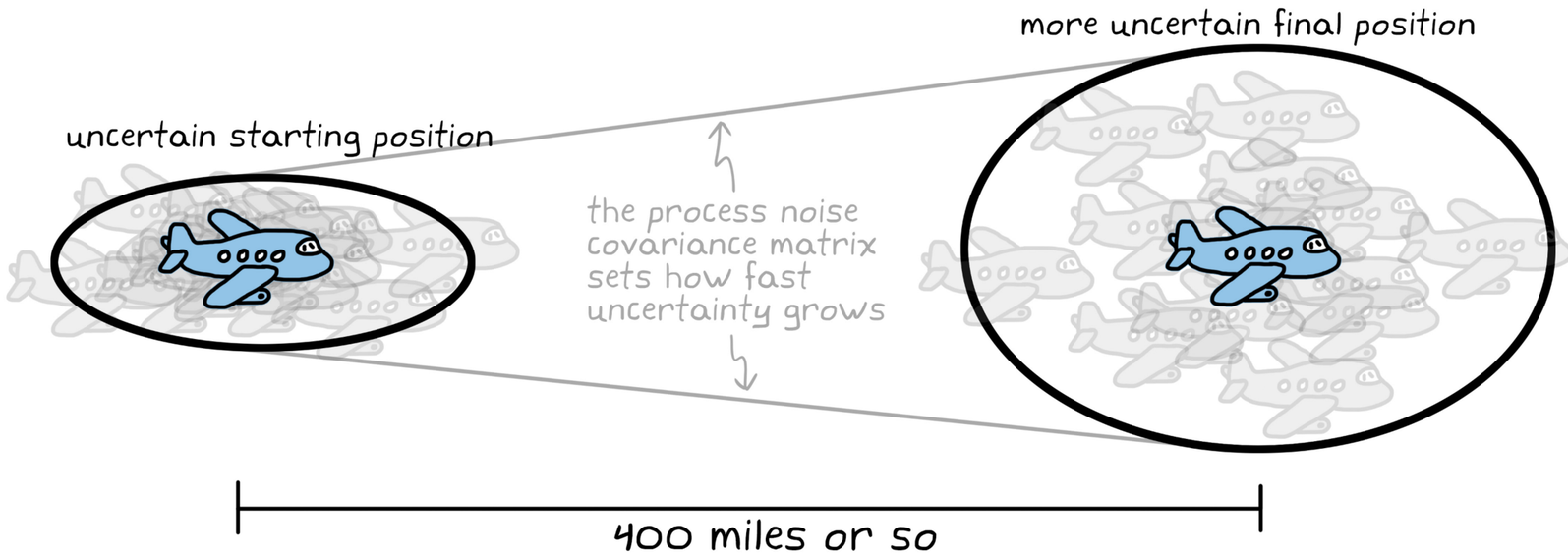
Prediction Error

Let’s look at the uncertainty in predicting the state. Remember, to predict we start at an initial state and then use a mathematical model to propagate that state into the future. There is uncertainty in this initial state before we even start the prediction process. This is captured in the prediction error covariance matrix. This means that even with a perfect model, this starting uncertainty will never go away and the final prediction will also have uncertainty associated with it. Imagine an airplane that is traveling exactly 400 mph and you are asked to predict where it will be in one hour. You know that it will be 400 miles further than the starting position, but if you were uncertain of the starting position, then you’d have the same amount of uncertainty in your answer.



Process Noise

If the model isn't perfect (which it never is!) the act of predicting causes additional uncertainty. The further into the future we have to predict, the more uncertain it becomes and so the prediction error covariance grows over time. We specify how the uncertainty grows with the process noise covariance matrix, Q. This matrix captures the uncertainty that comes from model discrepancies and unknown inputs into the system.

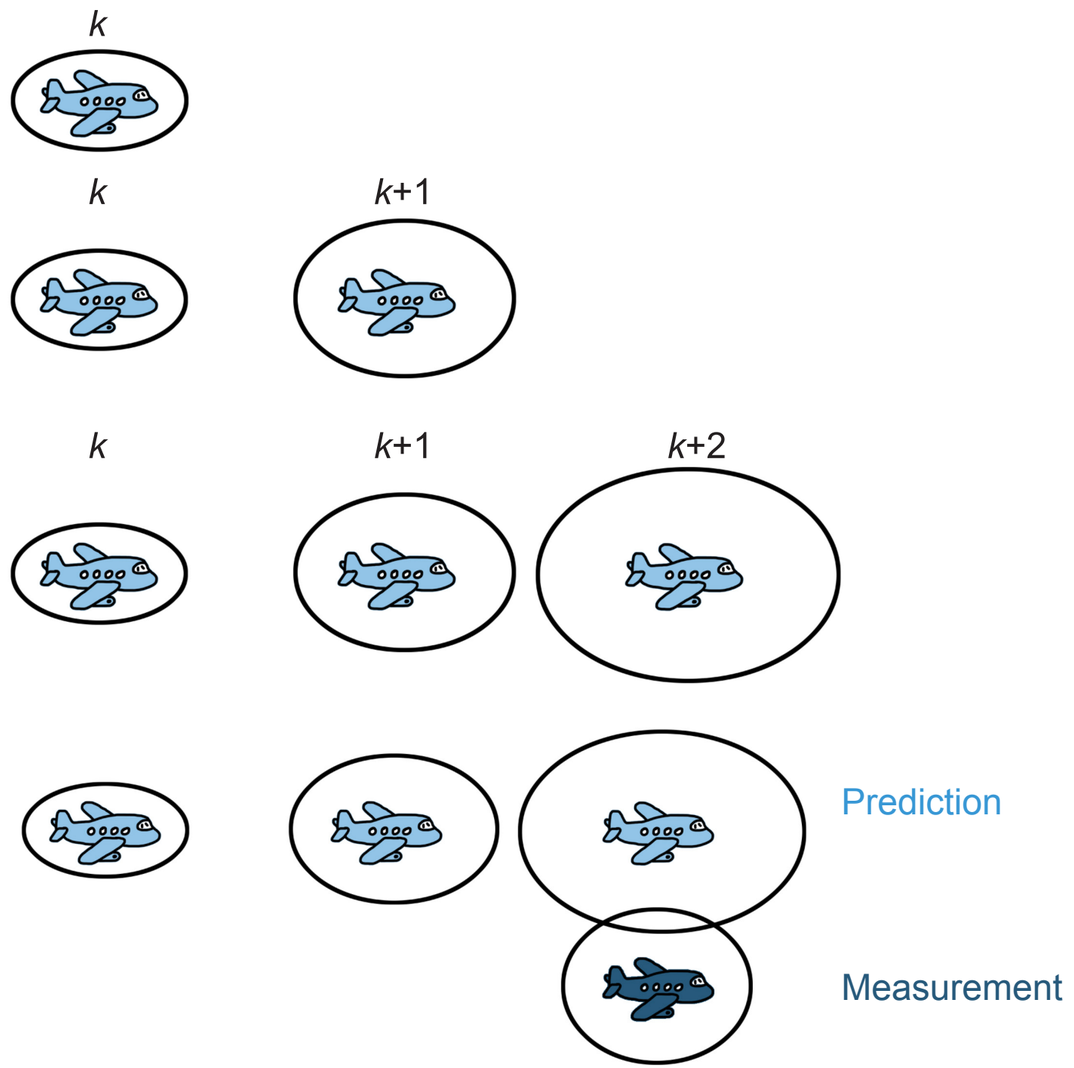


If the model is linear (or has been linearized prior to running it), a Gaussian probability distribution maintains its Gaussian shape throughout the prediction. Therefore, the final prediction uncertainty is still Gaussian in nature. This is why Gaussian distributions are so important for Kalman filters!

Managing Uncertainty

We can now frame all of these uncertainties into a single coherent workflow.

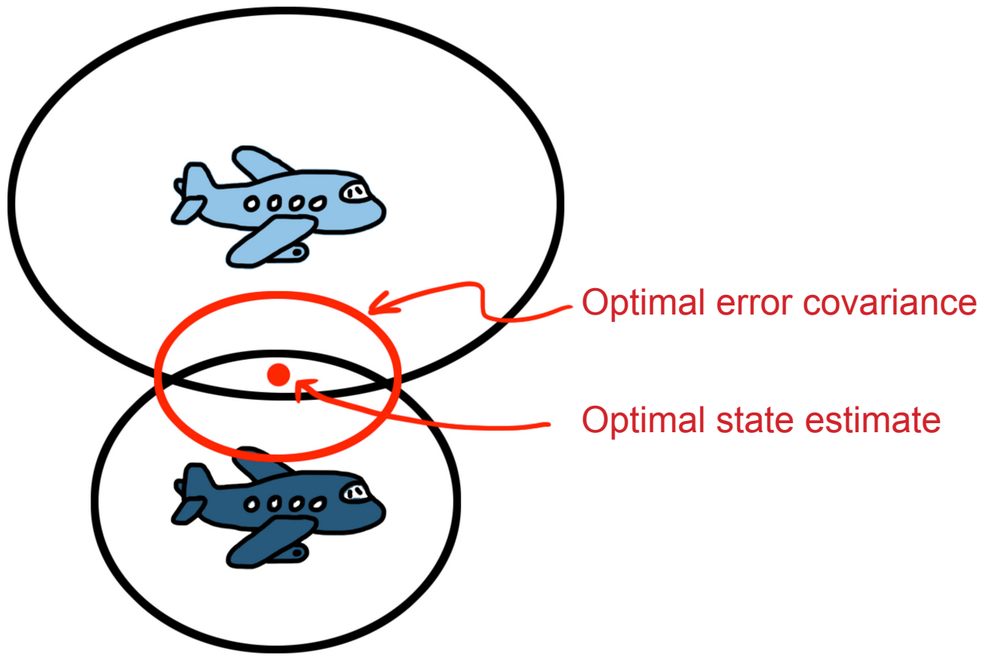
1. Start with an initial state and its associated error covariance.
2. Propagate the state and error covariance into the future with a model of the system. The error covariance grows based on the specified process noise covariance.
3. Continue propagating the prediction each time step. The error covariance will continue to grow, but it will maintain its Gaussian shape.
4. When a measurement is available, it will have its own error with a Gaussian distribution that is specified by the measurement noise covariance.



Combining a Prediction and a Measurement

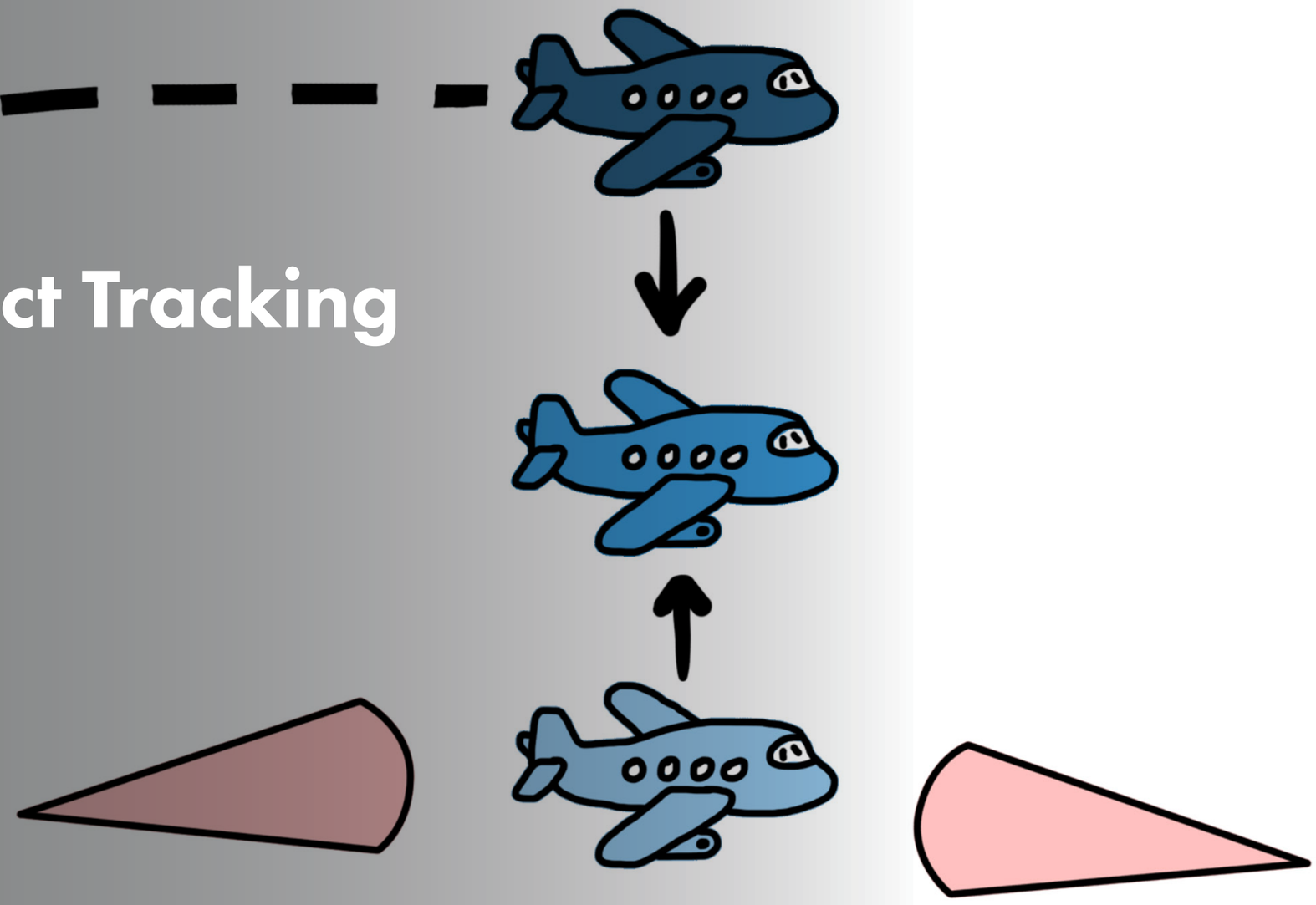
At this point, we have a predicted state with an uncertainty described by the prediction error covariance and we have a measured state with an uncertainty described by the measurement noise covariance. Now, we're back to our question of how to optimally combine these two estimates. Here's where the magic of the Kalman filter happens. To get the optimal corrected state, all we need to do is combine the two Gaussian distributions together.

The combination of two Gaussian distributions is a third Gaussian distribution! The resulting distribution represents the optimal corrected state (the mean of the distribution) and the uncertainty in the result (the covariance of the distribution). Lower covariance means you have more confidence in the estimated state.



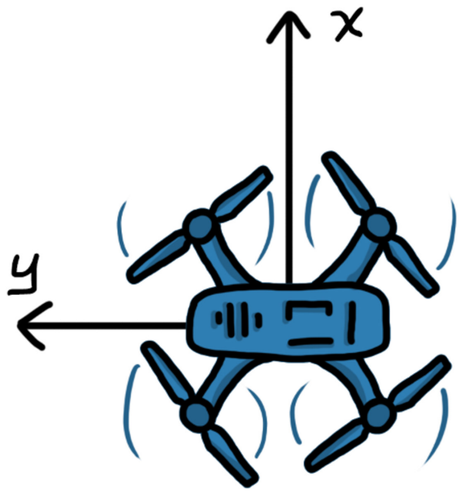
So, now we can think of a Kalman filter as an algorithm that just combines a prediction and its error distribution with a measurement and its error distribution.

Part 2: Single-Object Tracking

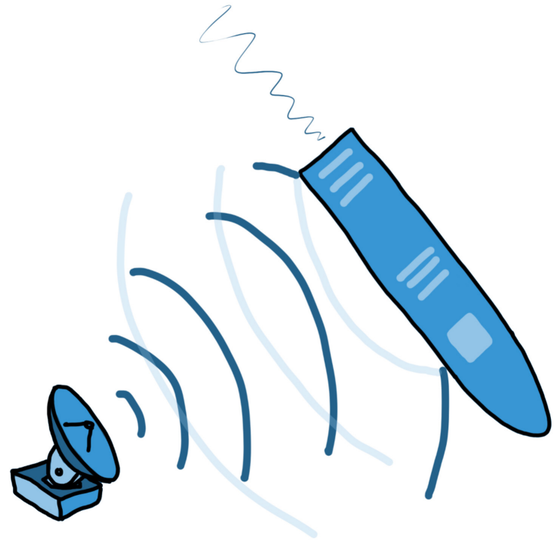


Applying This Approach to Tracking

This brings us to single-object tracking. Figuring out where another object is isn't all that different from figuring out where you are. We're simply trying to determine state, such as position or velocity, by fusing the results from sensors and models. The part that makes tracking harder is that we usually have to do it with less information.



What is the state of my system?



What is the state of a remote object?

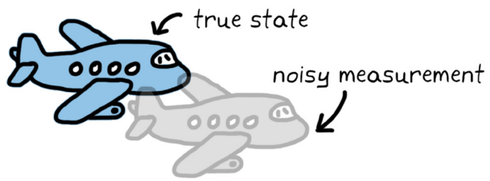
Next, we have to look at how the two steps of an estimation filter, prediction and correction, get more challenging when applied to a tracked object.

Measuring a Remote Object

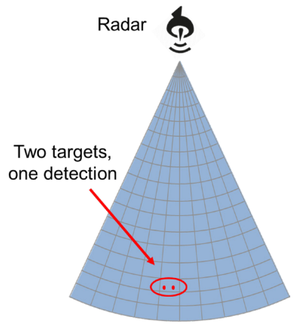
Let's start with the differences in the correction step. With tracking, measurements need to come from sensors such as a radar station or a camera vision system. The types of sensors don't fundamentally change the nature of the correction step. The idea is that we want to make a measurement of the system state in a way that could be used to correct the uncertain predicted state. What generates that measurement isn't as important as the quality of the measurement and the update rate of the sensor.

There are many challenges in obtaining remote measurements. Some of the most common categories are:

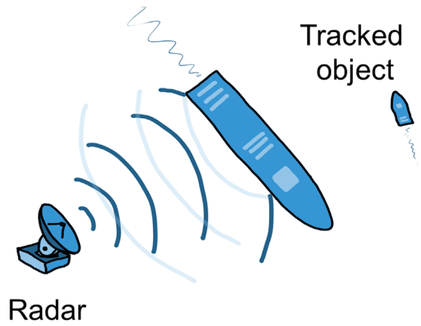
- Inaccurate position measurements of detected objects
- Inaccurate estimates of the number of detected objects (e.g., two objects sensed as one because the sensor was unable to distinguish between them)
- Missed detections due to occlusion and other scenario conditions and sensor limitations
- False positives (due to clutter)



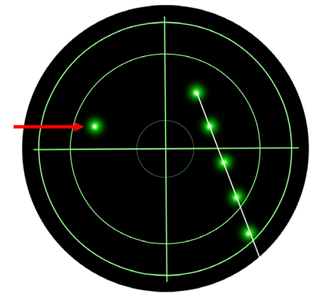
Inaccurate Measurements



Ambiguous Measurements



Target Occlusion

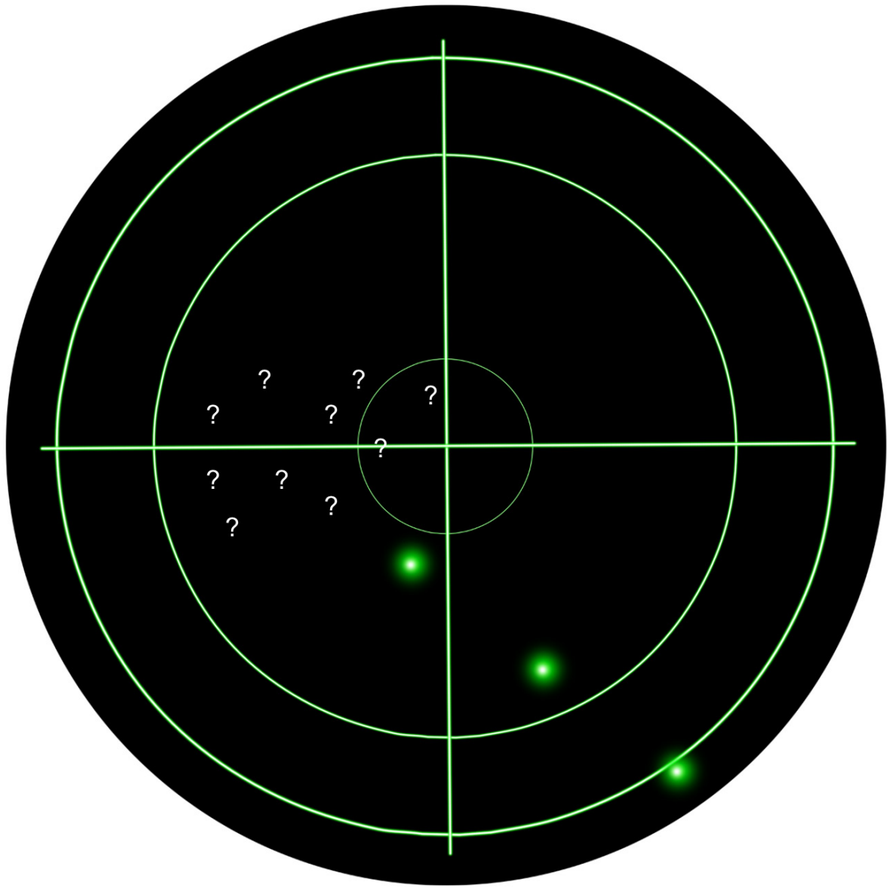


False Alarms

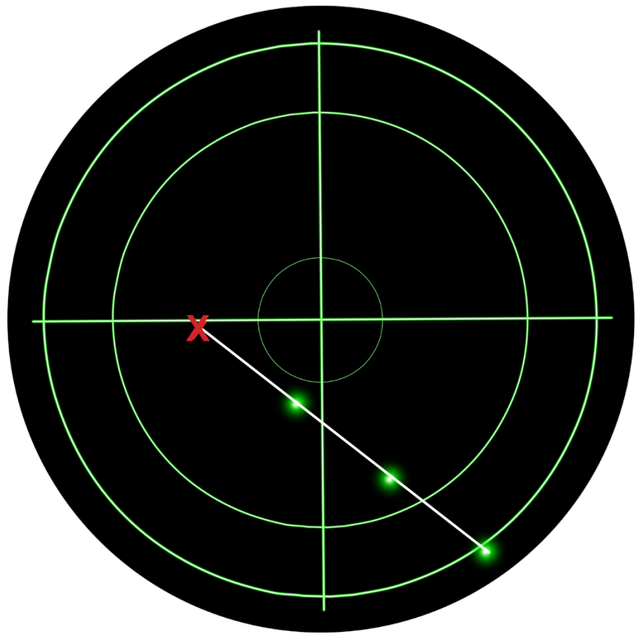
Predicting the Future State of a Tracked Object

It’s challenging to predict the future state of an object that you don’t have control over.

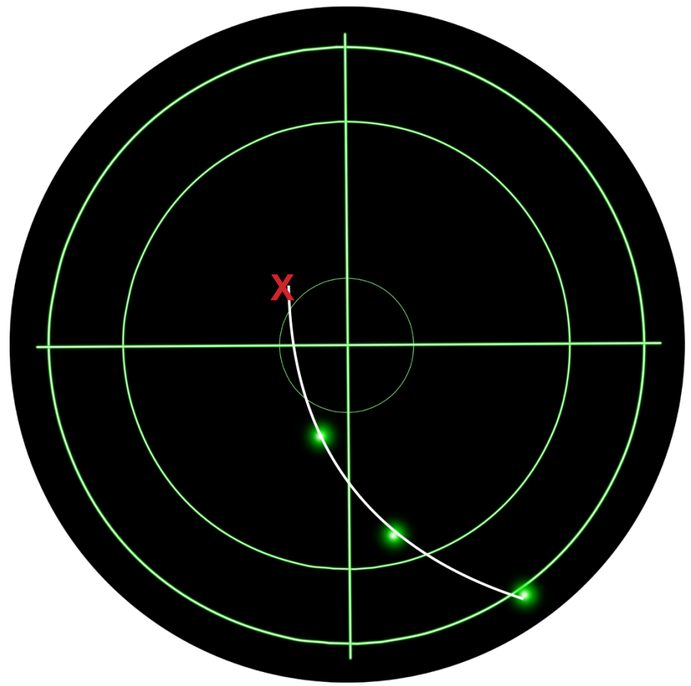
Let’s demonstrate the prediction problem with a thought exercise. Imagine an airplane that entered a radar station from the lower right, and every few seconds the aircraft location is updated. This is all of the information that you, or an estimation filter, have access to, and you want to predict where the airplane will be at the next detection. You are acting as the prediction step in the Kalman filter. Do you have a guess, and how confident are you in that guess?



Predicting the Future State of a Tracked Object



It’s probably around the red X, right? It’s been moving at a pretty consistent velocity before this so it makes sense that it’ll continue on this trajectory.



Now, what if the last few measurements looked like on the right instead? You’d probably assume that the airplane was currently turning and you’d have more confidence in a prediction that continued that trend. And if the measurements after this continued along a straight line, we might assume the object stopped turning and began to move at a constant velocity again. So, we tend to look at some window of past behavior to predict what the future behavior will be. How could we code this kind of intuition into a filter?

The Anatomy of Prediction

Consider that motion comes from three things:

- The dynamics and kinematics of the system that carries the current state forward. The airplane already has some velocity, and it would continue to move forward in a fairly predictable manner based on the physics of the plane traveling through the air.
- The commanded and known inputs into the system that add or remove energy and change the state. This would be things like adjusting the engines or control surfaces. If the pilot rotates the control wheel to the right, you could predict that the state of the plane also moves to the right.
- The inputs into the system that are unknown or random from the environment. This includes things like wind gusts and air density changes.

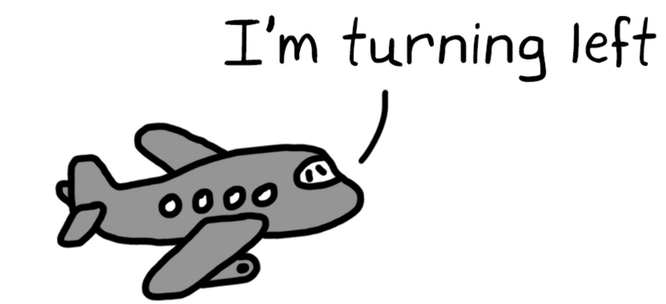
We need to take into account these three things when predicting a future state. The first and third bullet points are pretty much the same whether you're tracking an object or determining the state of your own system; however, the second bullet point is much harder for a tracked object. To understand why, we have to talk about cooperative and uncooperative objects.

Cooperative vs. Uncooperative Objects

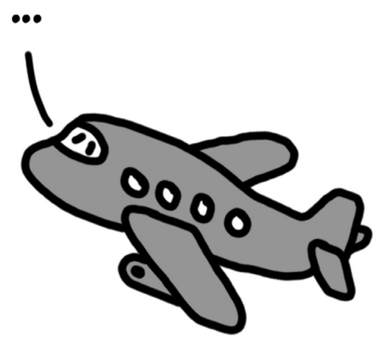
If you were the one flying the plane, and you knew that you didn't command any adjustments to the airplane—no control inputs—then you could expect, with reasonable certainty, that the plane would maintain its current speed and direction.

A cooperative object will also share this type of information with the tracking filter. Therefore, if you were tracking a cooperative airplane the filter would still have access to the control inputs and the prediction would be better off for it. In this way, tracking a cooperative aircraft is pretty similar to determining the state of an aircraft that you're flying yourself.

Uncooperative objects, however, don't share their control inputs and so we have to find a different way to handle the state changes that come from these additional unknown inputs.



Cooperative



Uncooperative

Addressing the Prediction Problem

The mathematical model of the system takes into account the dynamics (state transition matrix, F) and how the control inputs contribute (B matrix). The unknown inputs are accounted for by setting the process noise (not shown in the equation).

Dynamics

Unknown control input matrix

$$\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_k + w_k \qquad w_k \sim N(0, Q_k)$$

Known control input matrix

Process noise (Q_k) = model uncertainty

We now know that for an uncooperative tracked object, the control inputs, u_k, are unknown as well. Therefore, we've increased the number of unknowns in our system and necessarily should have less confidence in our prediction. The airplane could turn or slow down or speed up; we just don't know. We can account for this increase in unknowns by increasing the process noise. This would cause the Kalman filter to trust the prediction less and, in turn, trust the correction measurement more. This makes sense, right? If we have a hard time predicting where the airplane will be, why not just believe the radar measurement when we get one and basically ignore most of the useless prediction?

Dynamics

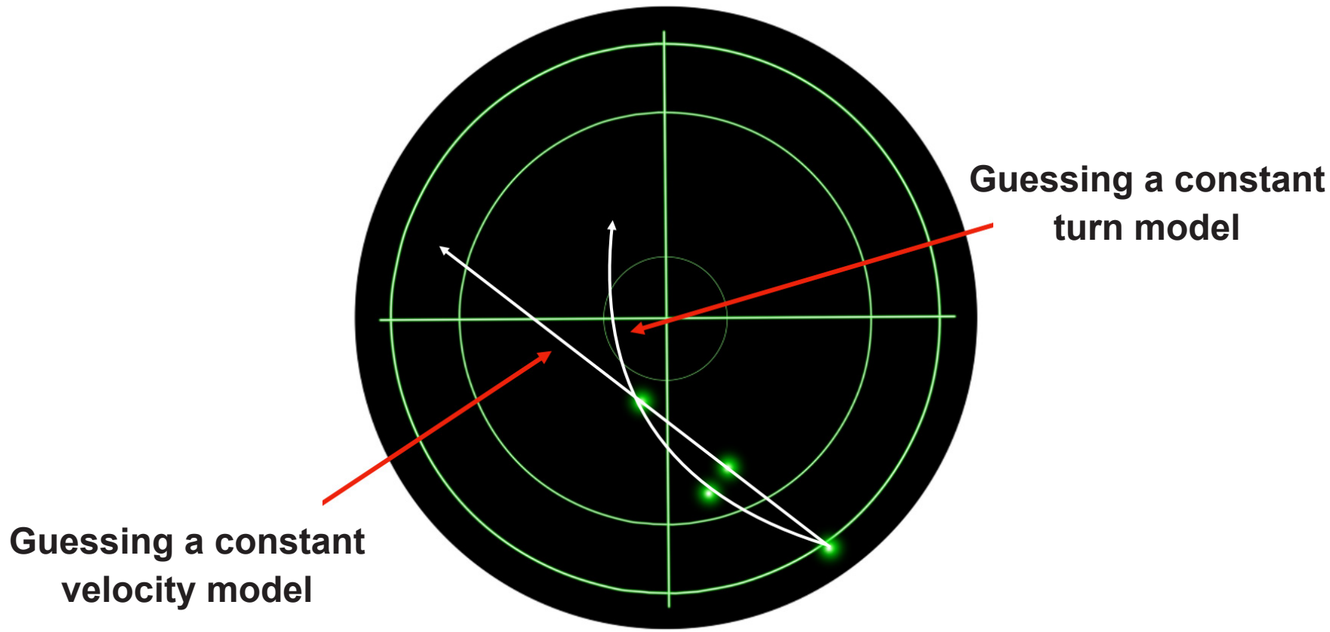
$$\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + w_k \qquad w_k \sim N(0, Q_k)$$

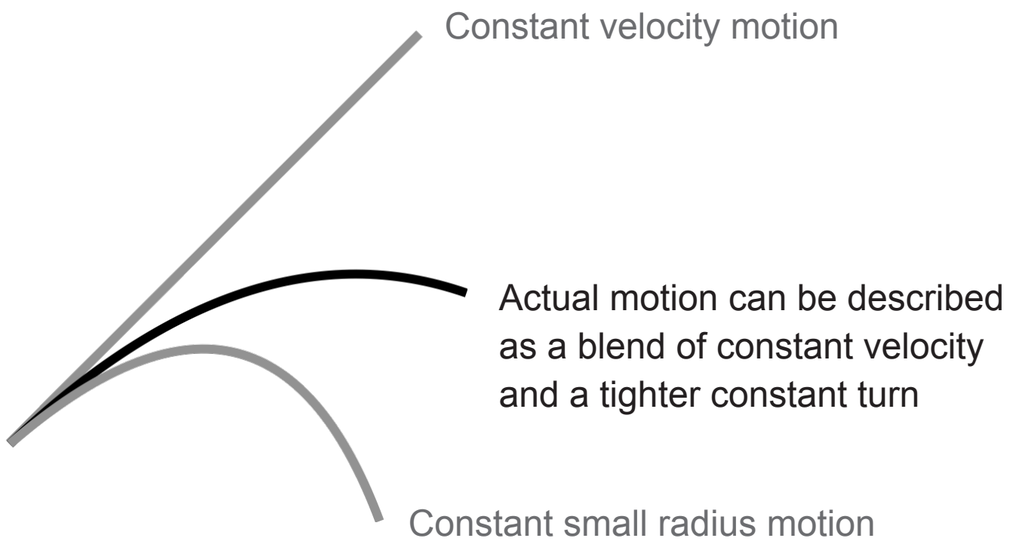
Process noise (Q_k) = model uncertainty + control inputs

Degrading the prediction diminishes the usefulness of the estimation filter, and in some situations we don't have to do that because we can guess how the object is maneuvering. We can use this guess to reduce some of the uncertainty that came from not knowing the control inputs.

We Already Did This

Guessing the maneuver is essentially what we did when we tried to predict where an airplane would be at the next radar update. We mentally changed the internal model in our head to switch between a constant velocity and a constant turn depending on what the previous measurements indicated. So, as long as we understand all of the different types of motion an object can perform, then we have a complete collection of mental models that we can jump between.





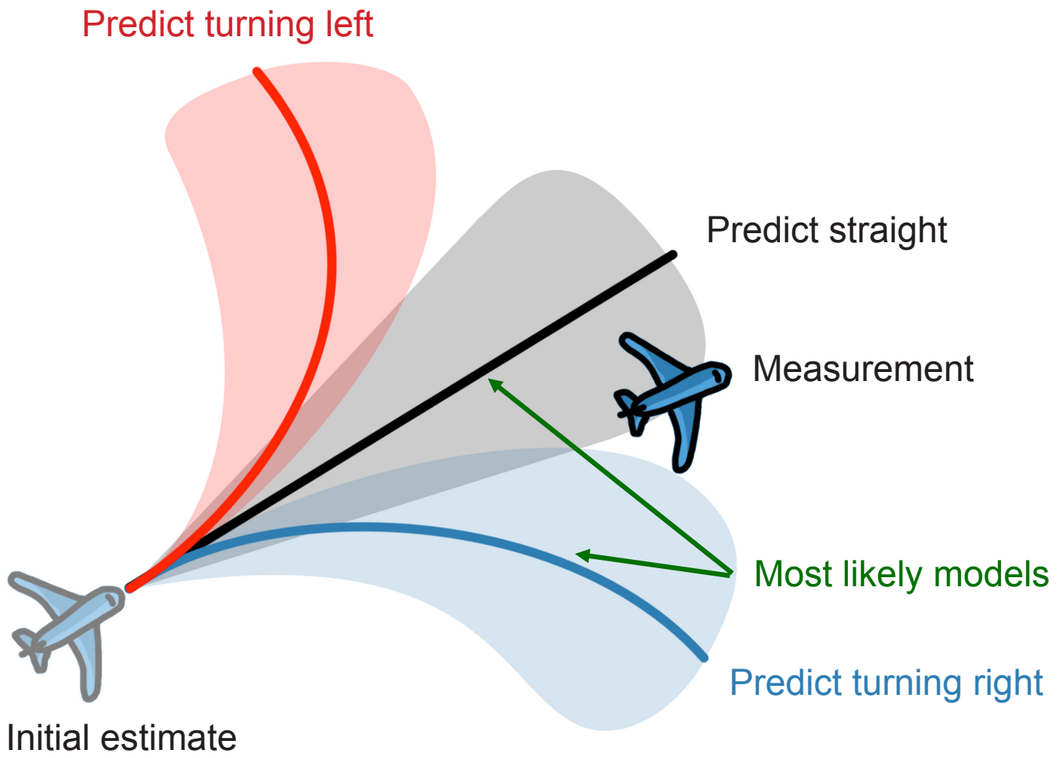
Some motions, however, can be described as the combination of two different motions. Something like a large radius turn could be the combination of a smaller radius turn and moving in a straight line. In this way, we don't need a model for every single possible motion, just enough models that can be used as the building blocks for every expected motion.

This is how a human might approach the problem, but how do we give our filter this capability?

Additional Motion Models

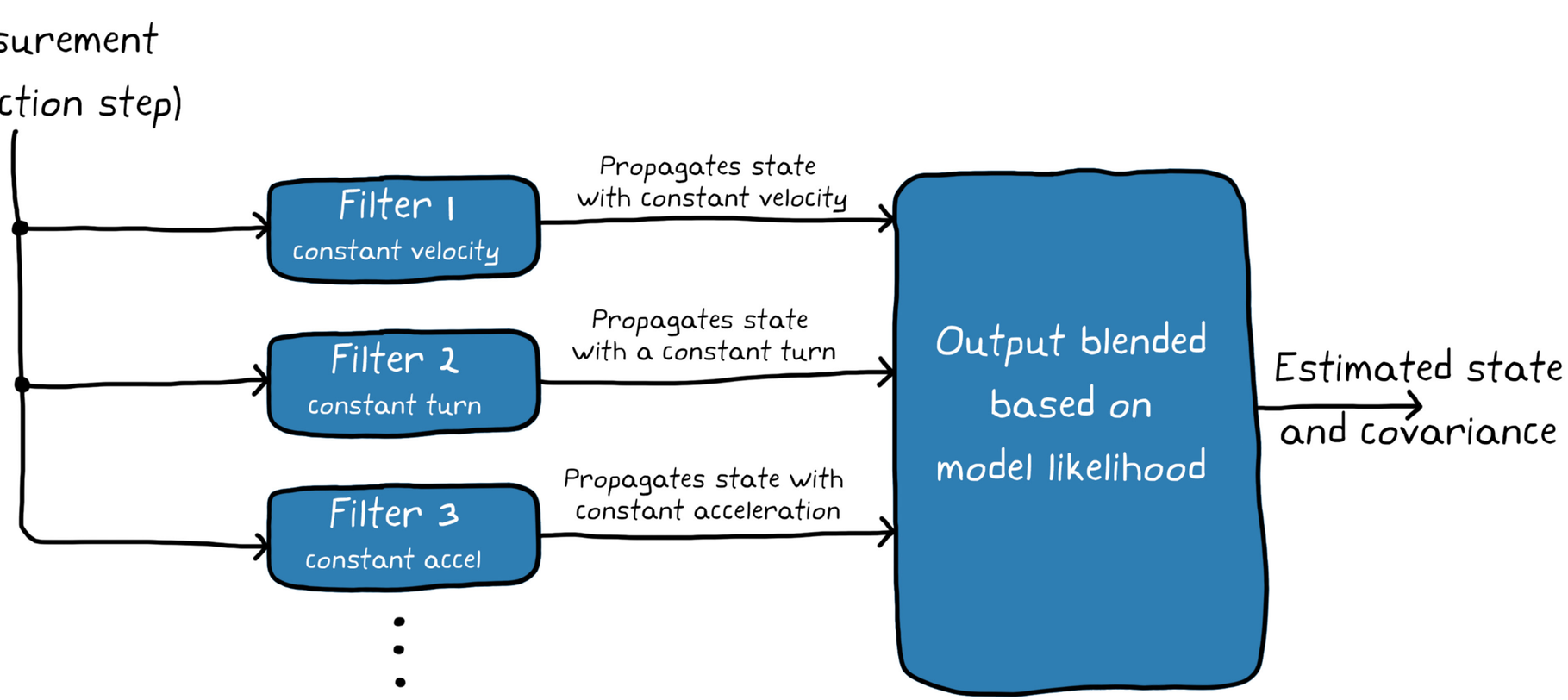
The answer is to run more than one prediction model at a time. Basically, we can think of this as running several simultaneous Kalman filters, each with a different prediction model and process noise. The idea is to have one model for each type of building block motion for the tracked object. These would be models such as constant velocity, constant acceleration, or constant turning.

Each model predicts where the object will be if it follows that particular motion. Then, when a measurement is available, it is compared with each prediction individually. From this, claims can be made as to which model most likely represents the true motion, and we can place more trust in that model for the next prediction cycle. Of course, the filter doesn't have to trust one model fully. The result could be a blend of the models, where the weights are based on the likelihood that that model is correct.



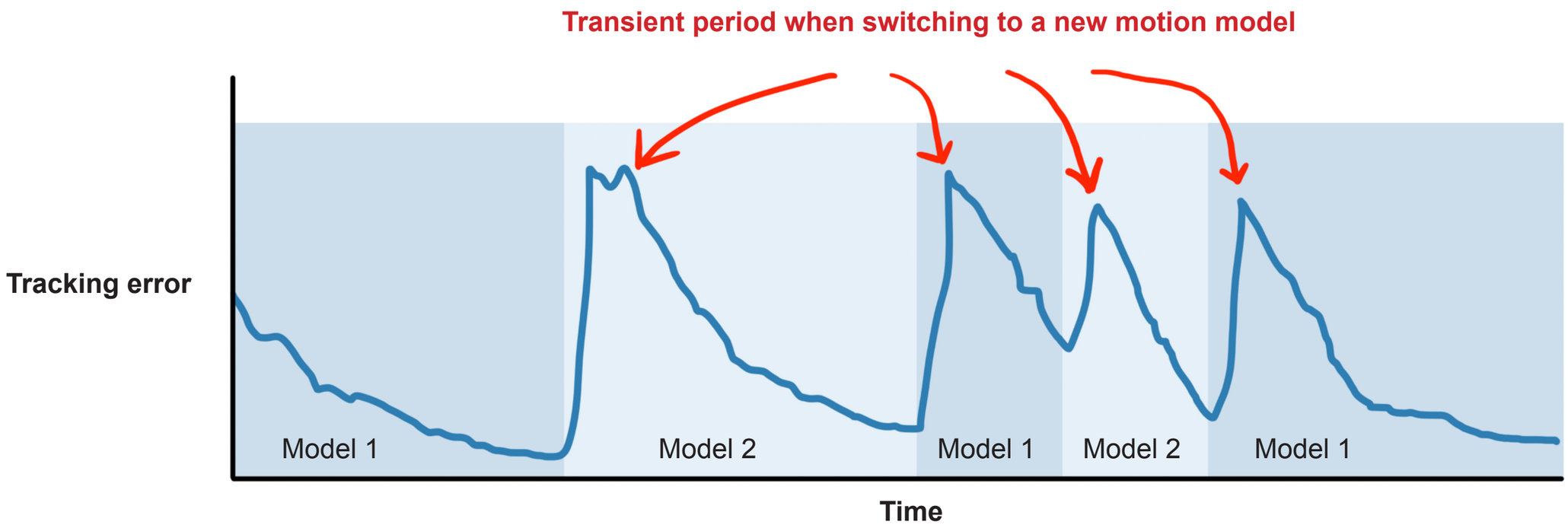
Blending the Models

In this block diagram, each filter operates independently of the others to propagate the system state forward. The outputs of the set of filters are blended to generate the estimated state of the system based on each model's likelihood that it's capturing the correct motion. When a measurement is available, that measurement is used not only to correct the state in every filter but also to adjust the model likelihoods.



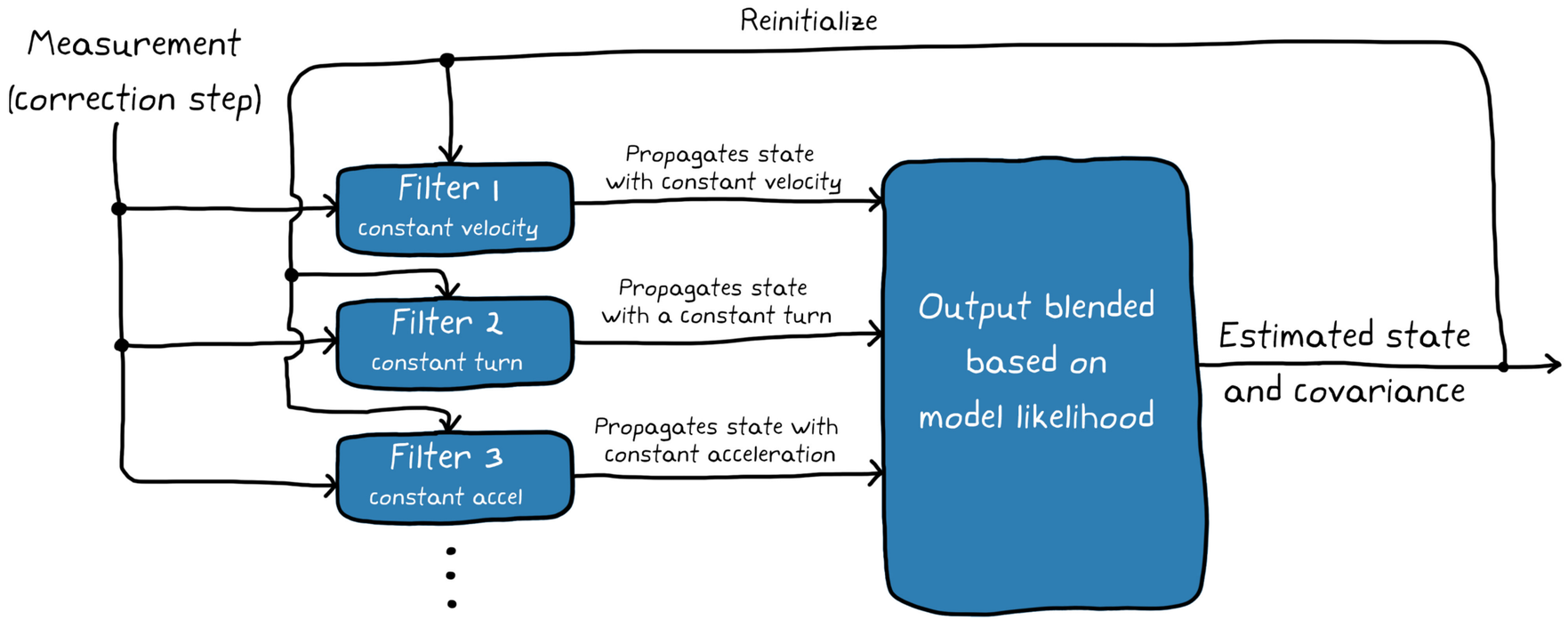
The Transition Problem

This is the general idea behind multiple model algorithms, but there is still one more step to get to interacting multiple models. The problem we have with the current way the filters are set up is that each one is operating on its own, isolated from the others. This means that for a model that doesn't represent the true motion, it's going to be maintaining its own bad estimate of the system state and state covariance. Then, when the object changes motion, and there is a transition to a new model, the filter is going to take some time to converge again. Furthermore, if the underlying filter uses a nonlinear model, it may never converge. We are running the risk that every time there is a transition to a new motion, the transient period will be longer than necessary while the filter tries to catch up.

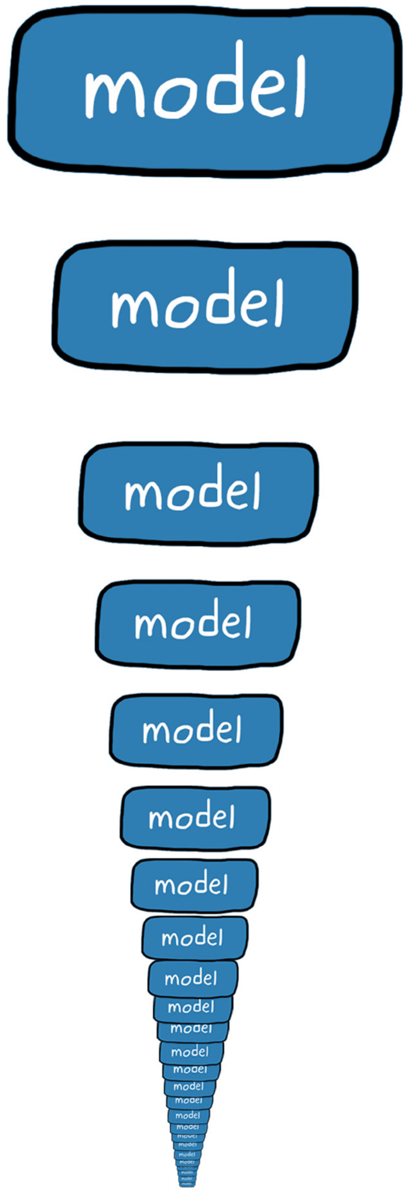


Interacting Multiple Models (IMMs)

To fix this, we allow the models to interact. After a measurement, the overall filter gets an updated state and state covariance based on the blending of the most likely models. At that point, every filter is reinitialized with a mixed estimate of state and covariance based on their probability of “switching to” or “mixing with” each other. This is constantly correcting each individual filter to reduce its own residual error, even when it doesn't represent the true motion of the object. In this way, an IMM filter can switch to an individual model without having to wait for it to converge first.



More Models!



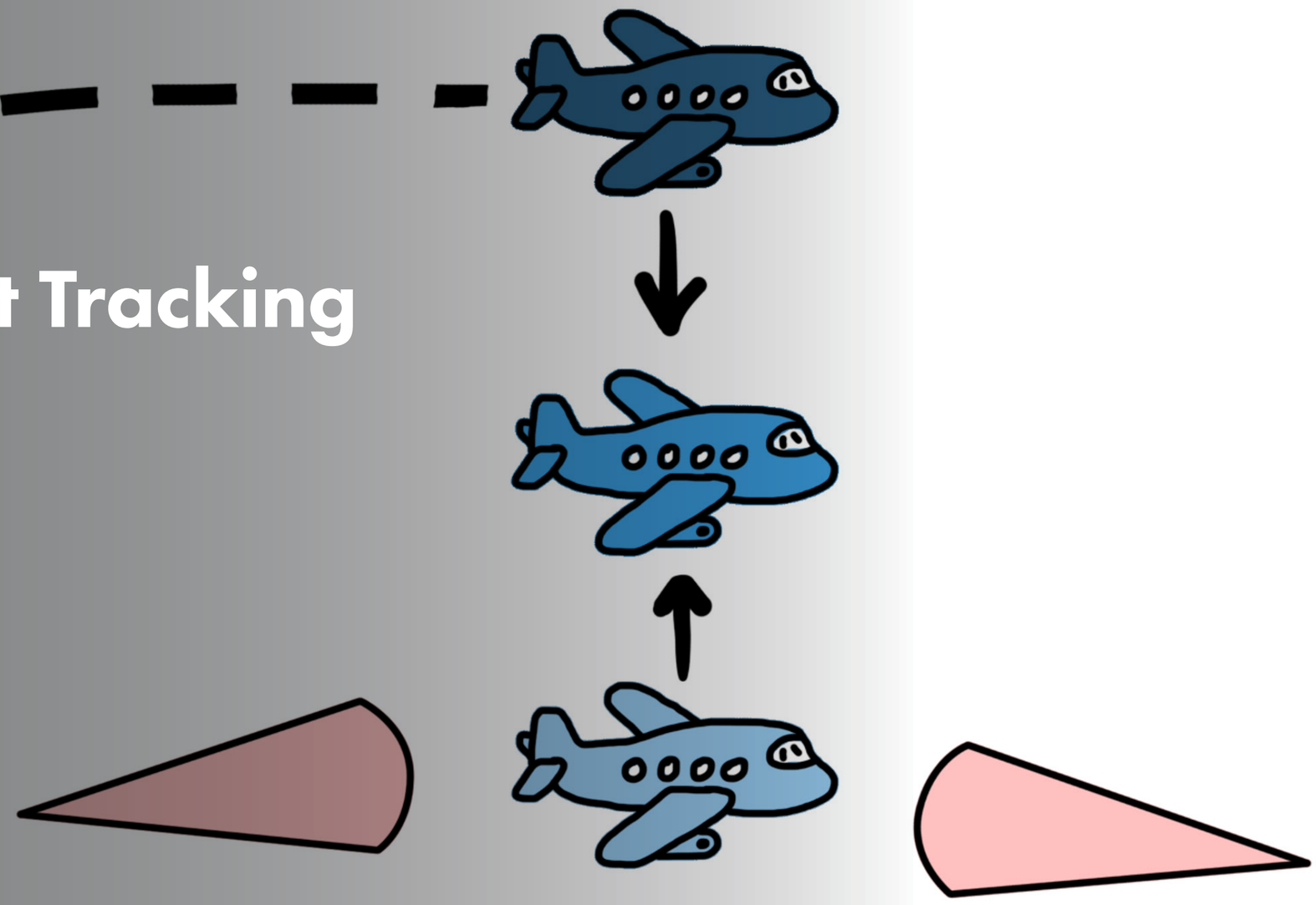
Models, all the way down!

You might be tempted to just run an IMM with dozens of models, something that could cover every possible motion scenario, right? Well, the problem with this is that for every model you run, you pay a price—namely, the computational cost of running a pile of predictions. And if it's a high-speed, real-time tracking situation, you may have only milliseconds to run the full filter. In addition, there is also the pain of having to set up all of these filters and get the process noise right.

Let's say computational speed is not a problem; you only care about accuracy. Well, having too many models can hurt accuracy too. It increases the number of transitions between models, and it's harder to determine when that transition should take place if there are a lot of models that represent very similar motions. Both of these contribute to a less optimal estimation.

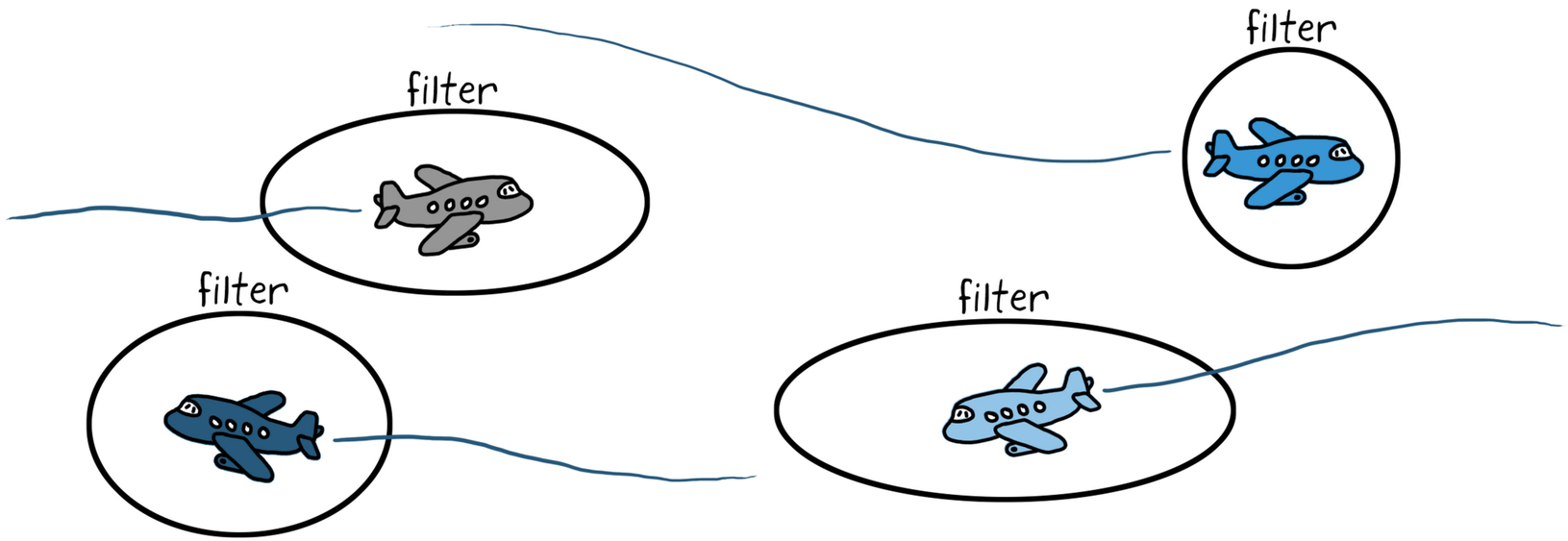
So, unfortunately, you still have to approach this filter in a smart way and try to find the smallest set of models that can adequately predict the possible motions for the object that you're tracking. Practically speaking, this tends to be less than 10 models, usually around just three or four.

Part 3: Multi-Object Tracking



Multiple-Object Tracking

What we've covered so far has addressed tracking a single object; however, we can apply that knowledge and build on it to track multiple objects at the same time. Multi-object tracking is important for many applications including autonomous systems and surveillance systems. At first glance, it doesn't seem like tracking multiple objects is that much harder than tracking a single object. For example, can't we just take the tracking algorithm like the IMM from the last section and apply one to each object?

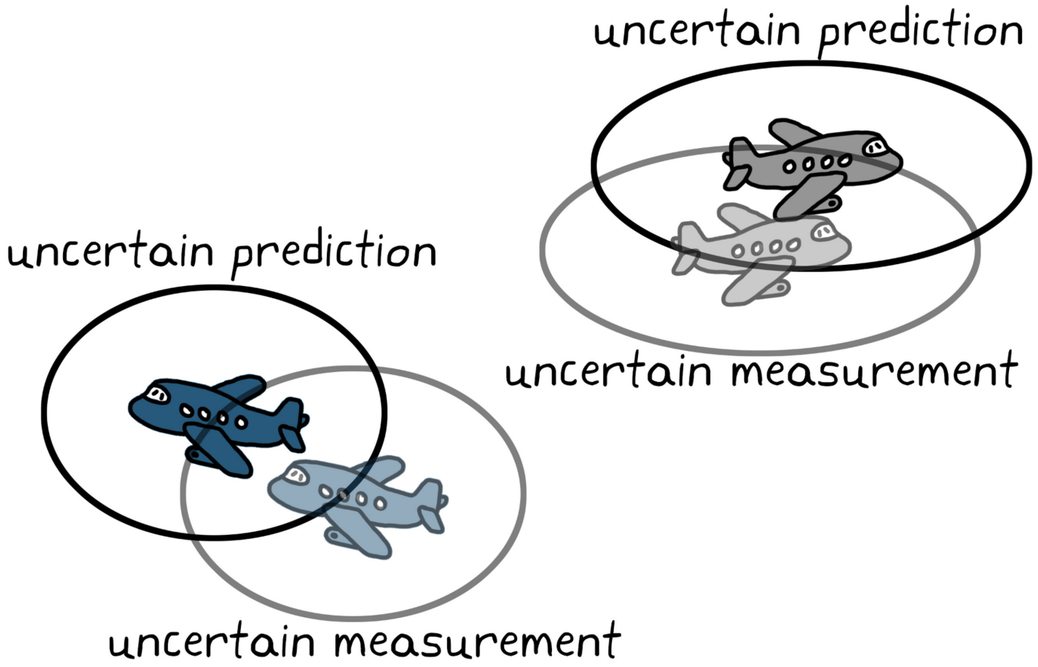


Yes, but unfortunately that's not enough. We need to consider some additional things with multi-object tracking, and that's what we're going to talk about in this section.

The Difficulty of Multi-Object Tracking

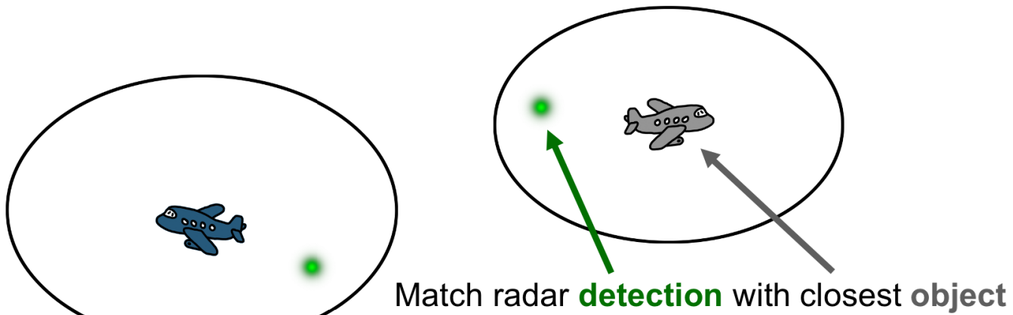
Most of the difficultly in multi-object tracking comes down to dealing with uncertainty. We have uncertainty in our predictions of the paths that the objects are taking, and we have a set of uncertain measurements or observations of the objects.

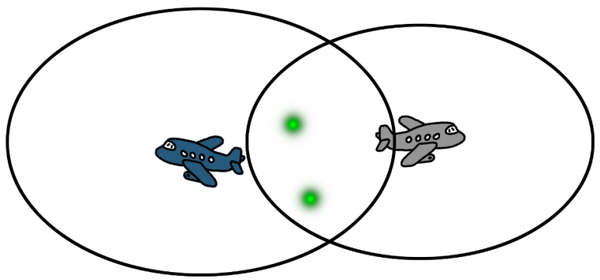
In the last section, we looked at tracking an airplane with a radar station by predicting where the airplane would be in the future and then correcting it with a noisy radar measurement. If we expand this to multiple airplanes, we have an imperfect prediction for each of them that we need to correct with their corresponding uncertain measurement. This can be difficult in practice when the tracked objects are clustered near each other or are unpredictably appearing and disappearing within the tracked area.



Data Association

This brings us to our first problem. We don't want to correct the prediction of one object using the measurement from a different object. But if there's no identifying information that comes with the detection, such as an airplane tail number or some other unique signature, how do we know which object we've just detected? For example, if all of the airplanes we are tracking have similar radar cross-sections, and this is all of the information we have access to, how can we match an arbitrary detection with the appropriate tracked object?





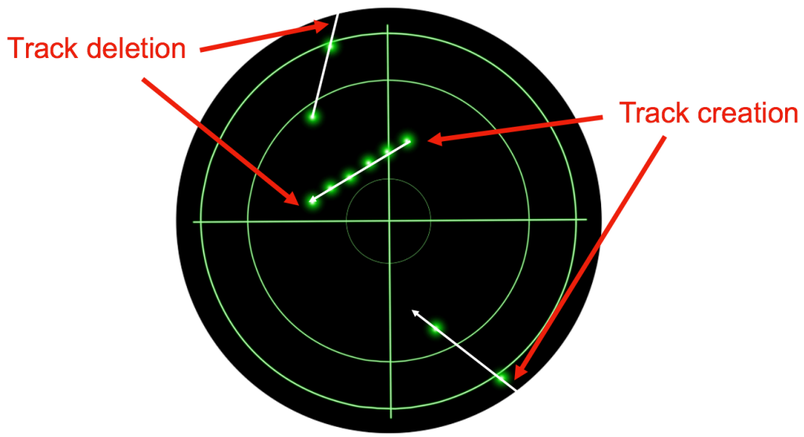
We have a data association problem

If the objects are sparsely distributed and the observations are relatively reliable, it would be a simple matter to claim that an observation is of the predicted object that it's closest to. In this case, we would just assign the measurement to the nearest object and run the estimation filter for it like we would when tracking a single object.

The tricky part, however, comes when predicted objects are close enough to each other or our uncertainty is great enough that a measurement could be of more than one object. Now we have some figuring out to do. This is the data association problem. We have to associate the detections with the right objects.

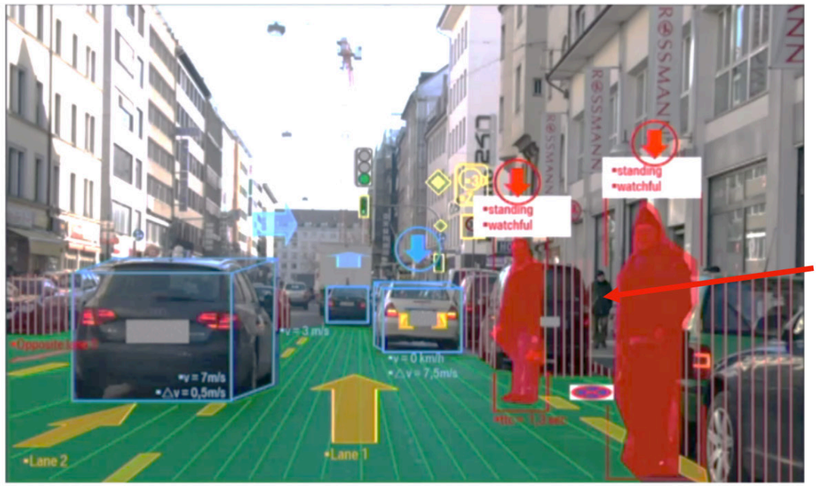
Track Maintenance

The other problem that we need to consider is that the number of objects being tracked is not fixed. Sometimes tracks need to be created or removed based on what we observe. This is the track maintenance problem.



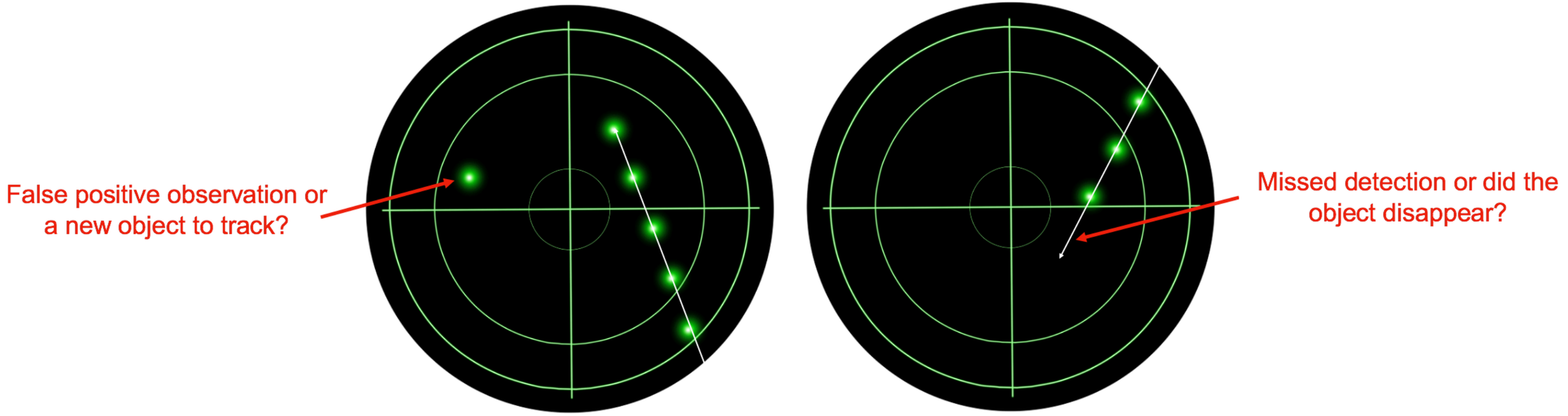
We may add a new track when an airplane flies into the radar range, and similarly we may delete a track when one flies out. But track creation and deletion doesn't just happen along the edge of the field of view of the sensors: objects may appear and disappear anywhere. For example, an airplane may take off or land within the radar range.

Or, as another example, if a self-driving car is tracking pedestrians as it drives down a busy street, it has to be aware of new people and track their walking direction as they come into view in front of the car and it no longer has to care about the pedestrians that it has already passed.



Uncertainty Complicates Track Maintenance

A basic way to approach track maintenance is to add a track whenever there is a detection that doesn't match an existing object and to delete a track if an existing object is not detected. It's unfortunately not this straightforward, because remote sensors make their observations in a probabilistic manner. Sometimes sensors have false positive measurements: they detect something that isn't actually there. And, sometimes, sensors fail a few times in a row to detect an object that is actually there. So, we need to be careful not to create tracks prematurely, which clutters our view of what's actually there, or to delete tracks prematurely.



Overview of the Approach

These are the two questions that we want to answer in this section. When tracking multiple objects:

1. What are some ways that we can approach data association?
2. What are some ways that we can address track maintenance?

To answer these questions, we're going to walk through each element of the multi-object tracking flow chart.

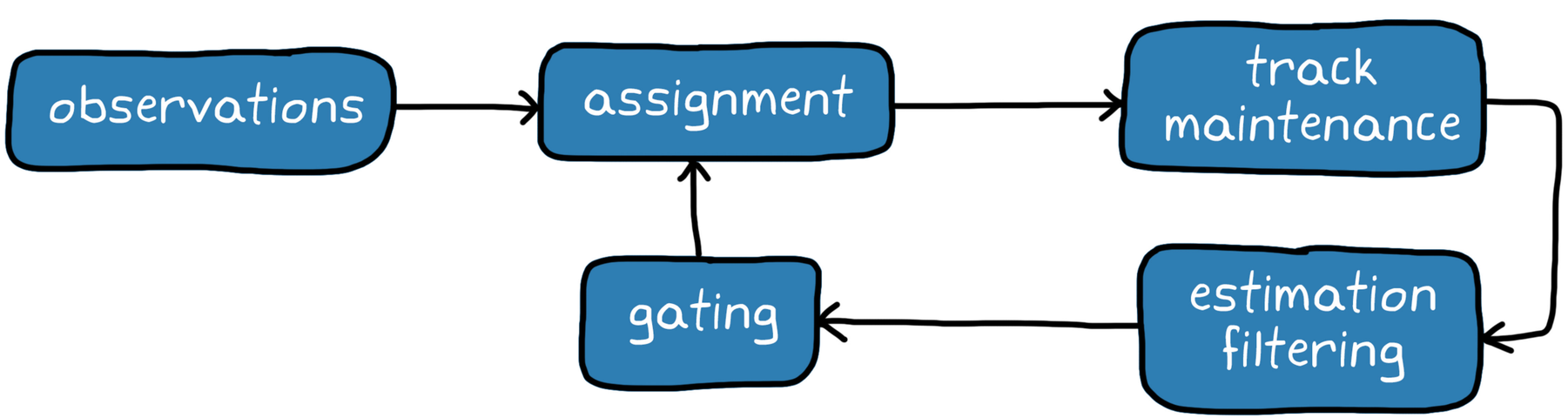
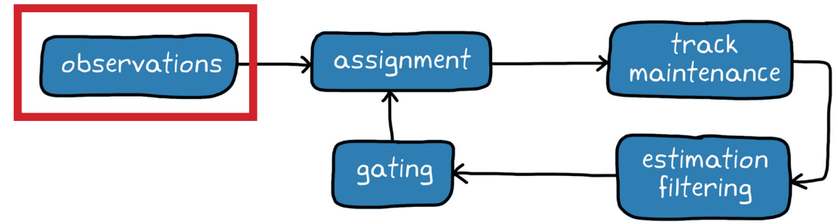


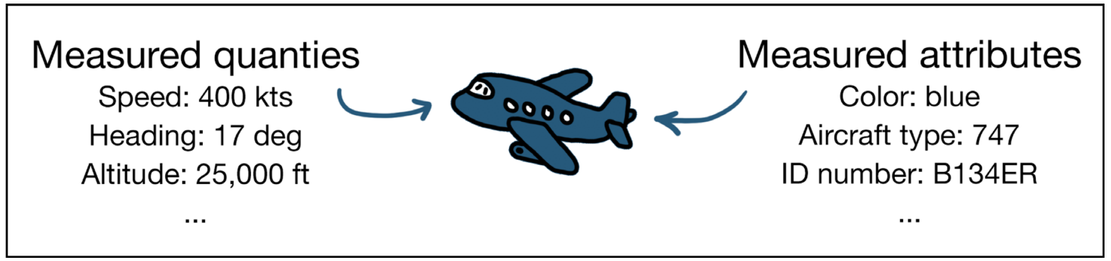
Figure adapted from Design and Analysis of Modern Tracking Systems by Samuel Blackman and Robert Popoli (Artech House Radar Library).

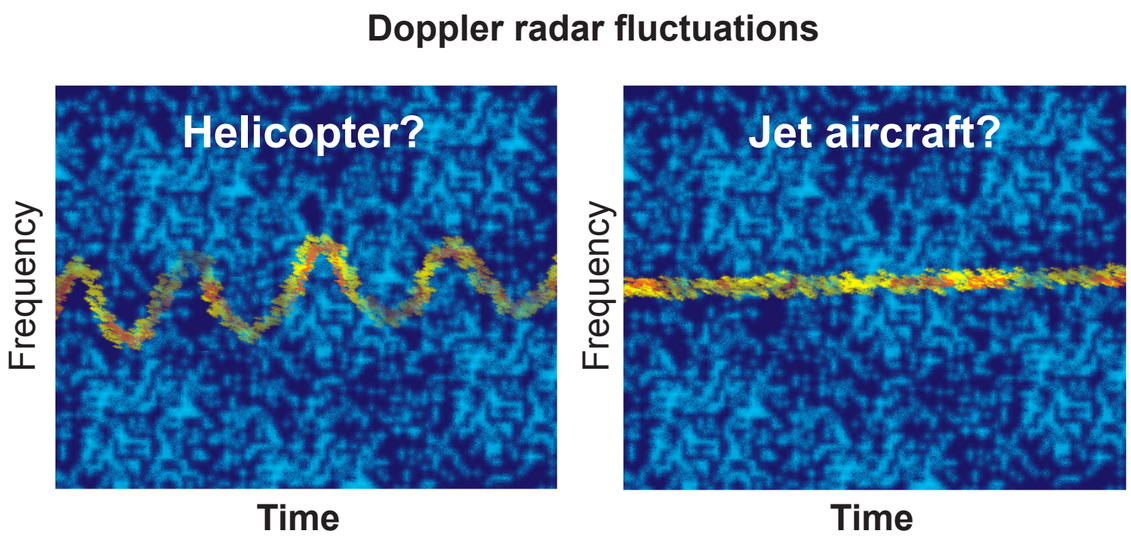
It's important to note that while your particular tracking algorithm might not result in a functional structure that looks exactly like this, each of these elements will almost certainly be present in your approach somewhere.

Observations



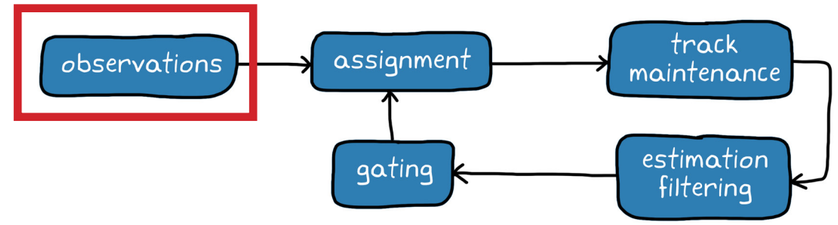
An observation, also known as a detection, occurs when the sensor measures an object. The observation may contain measured quantities like range, range rate, or elevation—values that represent the kinematic nature of the object. But observations could also contain measured attributes such as target type, ID number, and object shape. For example, you might get the tail number of an aircraft or some other identifying information with a radar measurement. Receiving a tail number is pretty unambiguous for data association: just match the observation ID with the track ID.



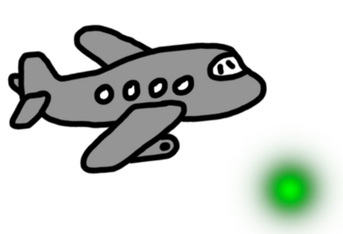
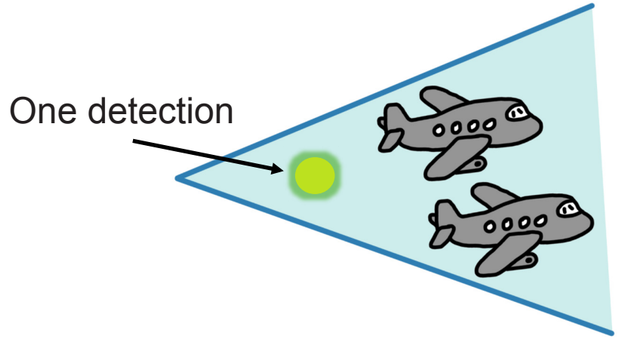


However, other types of attributes require some interpretation. For example, you might collect micro-Doppler radar fluctuations and from that be able to determine the type of aircraft you're tracking, and the uncertainty in the observation depends on the degree of separation between the two objects. A helicopter might look completely different from a jet aircraft because of the large rotating propellers, whereas a bird and a model airplane might have similar Doppler signatures.

Types of Observations

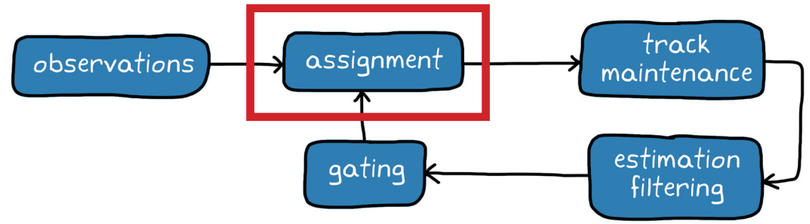


Other things to consider with observations is that if the tracked object is a point target, the sensor would report at most one detection. So, we have to associate one detection with one object. But if the target object is large and the sensor has sufficient resolution, there may be more than one detection per target and we need to consider this when determining how we're going to handle associating this data. Also, if the resolution of the sensor is low, two objects may exist within a single detection. In this case, both objects have been observed, so we don't want to stop tracking either of them, but they show up as only a single detection. Our track deletion algorithm will have to handle these situations.

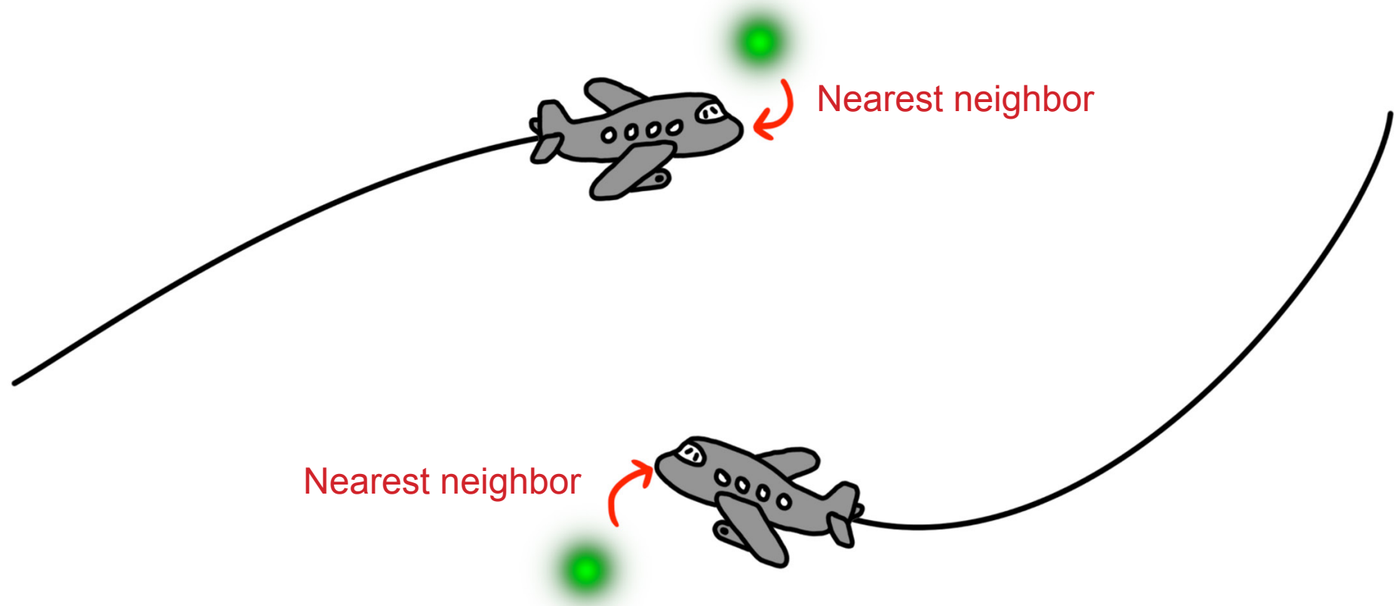
Point targets	**Extended objects**	**Merged detection**
		
One-to-one	Many-to-one	One-to-many

For the rest of this section, we're only going to talk about point targets—the case where we expect one detection for each tracked object.

Assignment

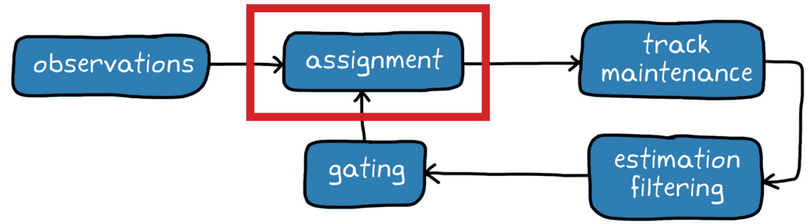


Assignment is the process of matching an observation to a tracked object (a track). Possibly the simplest assignment algorithm to think about is the global nearest neighbor (GNN). This algorithm simply assigns a track to the nearest observation, accounting for all the observations and tracks.



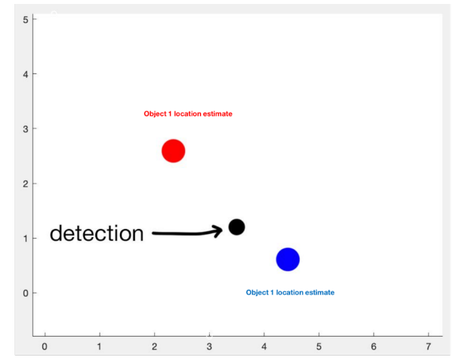
The interesting thing here is that it’s not necessarily the nearest Euclidean or geometric distance but the nearest probabilistic distance. One such probabilistic metric is the Mahalanobis distance. Let’s look at why something like the Mahalanobis distance is a better indication of how "near" an observation is to a track.

Probabilistic Distance

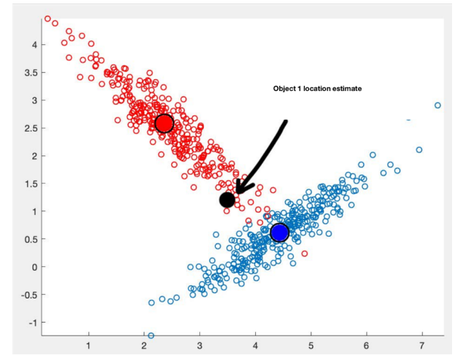


With a probability distribution, like we have with both our predictions and our measurements, the lowest Euclidean distance doesn't always indicate that a prediction and a measurement are the best fit. This is because we have more confidence in our predictions and measurements in the directions with lower standard deviations.

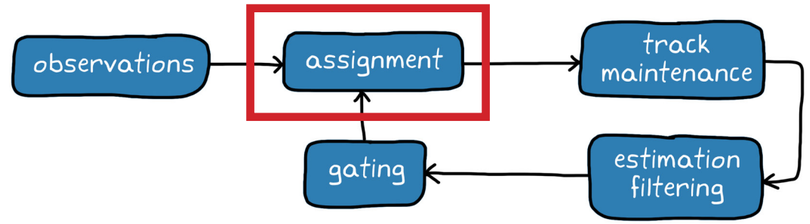
For example, in this simple image, we have a prediction for the location of two objects and a single detection that lies between them. If we used Euclidean distance, we'd assume that the detection is of Object 2 since it's closer



But if we look at the probability distributions of the two predictions, we can see that it's more probable that the detection is of Object 1. This is what the Mahalanobis distance does for us. It's the distance normalized by the standard deviation.

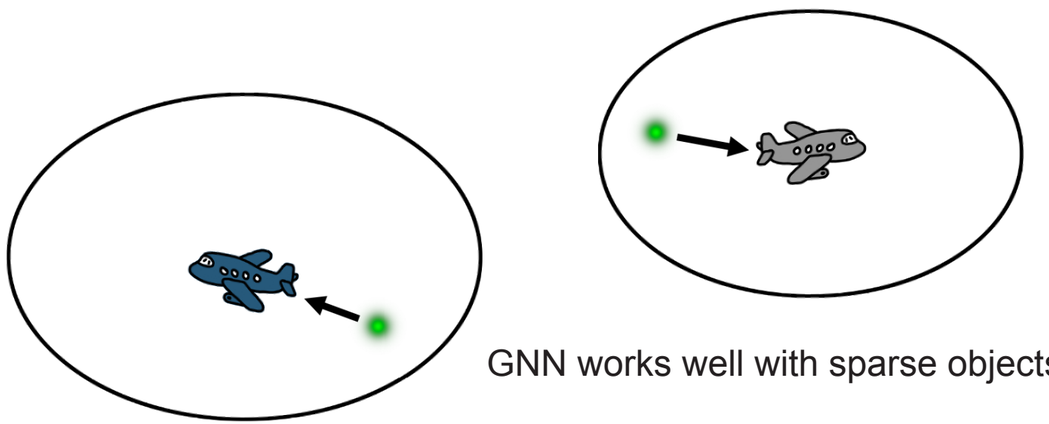


Global Nearest Neighbor

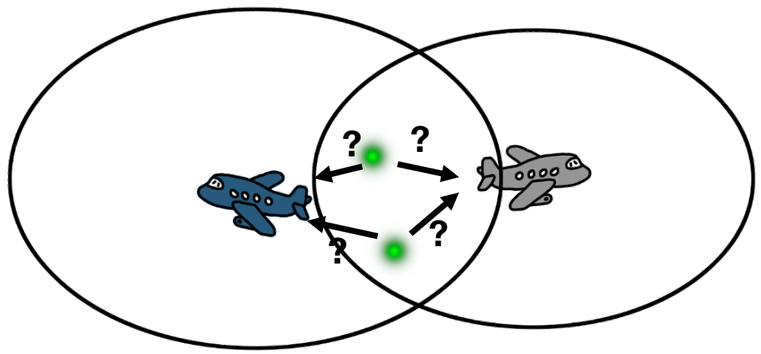


It's easy to see why an algorithm like the GNN works well when the tracked objects are sparse relative to the uncertainty of the prediction and observation.

For clustered objects, however, we can no longer be certain that the nearest probabilistic distance is the right assignment. In fact, we often can't figure out any way to perfectly match one observation to one track in these situations. Therefore, we need to investigate other types of assignment algorithms.

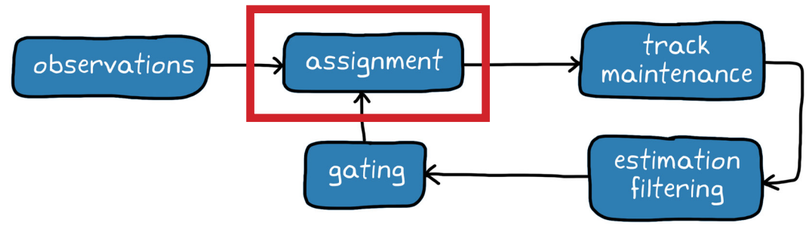


GNN works well with sparse objects.

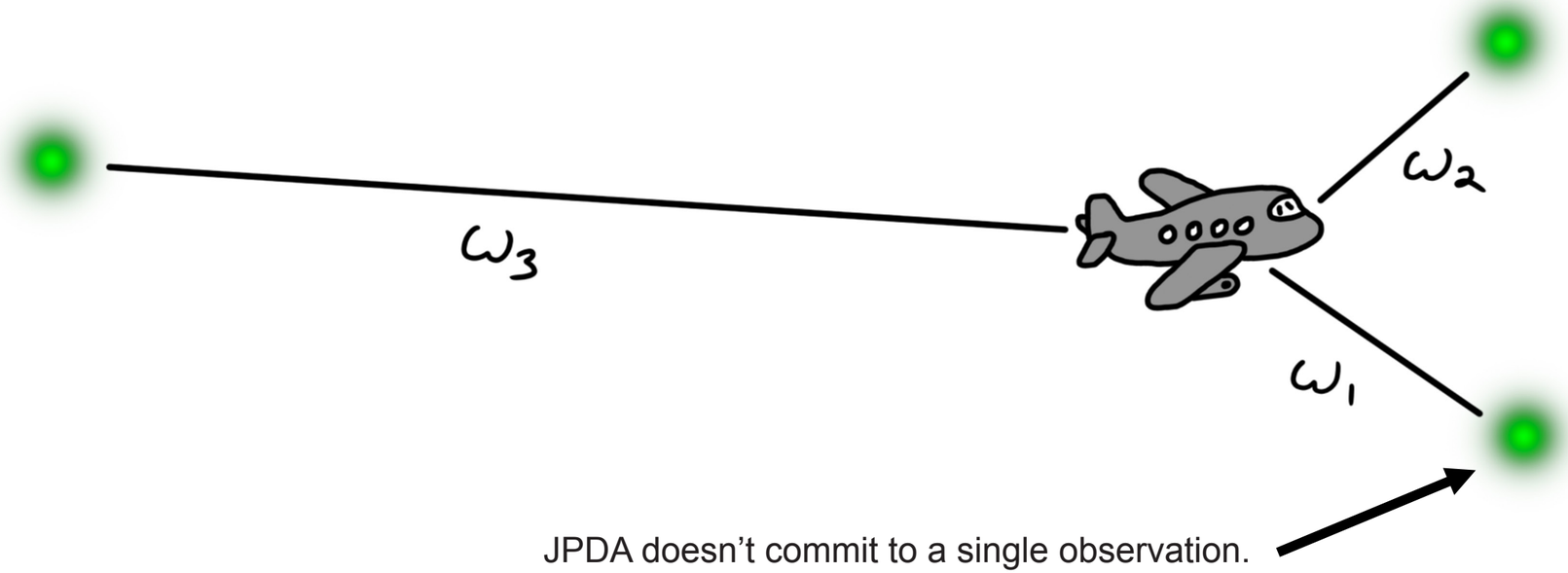


GNN doesn't work well for clustered objects.

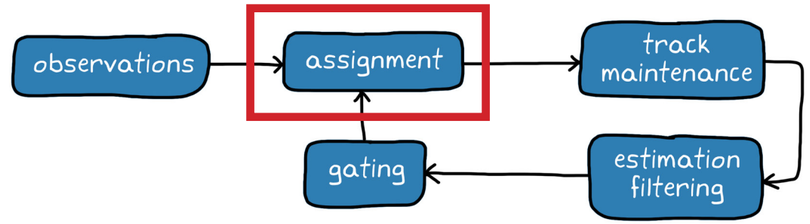
Joint Probabilistic Data Association



We saw how GNN works well for sparsely distributed objects, but for clustered objects the joint probabilistic data association algorithm (JPDA) will be better. The JPDA doesn't make a hard assignment between one observation and one track. Instead, it makes a weighted combination all of the neighboring observations, with more probable observations being weighted more heavily than further ones. This is an improvement over GNN because if there are two observations that could be the object, the JPDA won't fully commit to one, possibly the wrong one. So, if the tracked objects are clustered near each other, and the observations are all clustered near them too, this algorithm can handle that by blending a few of them together rather than jumping around between wrong and right detections.



Other Assignment Algorithms



There are more assignment algorithms than just these two, and you can create your own based on your tracking situation. However, the bottom line with all of these algorithms is that you are trying to figure out how best to associate an observation with a track.

Help Center

Search Support

CONTENTS

« Documentation Home

« Sensor Fusion and Tracking Toolbox

Get Started with Sensor Fusion and Tracking Toolbox

Applications

Orientation, Position, and Coordinate Systems

Trajectory and Scenario Generation

Sensor Models

Inertial Sensor Fusion

Estimation Filters

Multi-Object Trackers

Visualization and Analytics

Documentation | Examples | Functions | Blocks | Apps | Videos | Answers

Topics

Introduction to Multiple Target Tracking
Introduction to assignment-based multiple target trackers

Introduction to Assignment Methods in Tracking Systems
Introduce 2-D and S-D assignment problems in tracking systems

Introduction to Track-To-Track Fusion
Track-To-Track Fusion Architecture Using Track Fuser

Multiple Extended Object Tracking
Introduction to methods and examples of multiple extended object tracking in the toolbox.

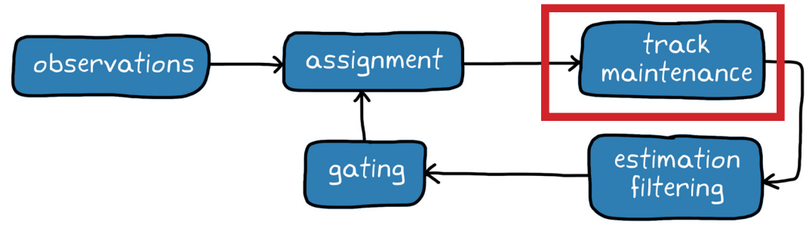
Convert Detections to objectDetection Format
These examples show how to convert actual detections in the native format of the sensor into `objectDetection` objects.

Introduction to Using the Global Nearest Neighbor Tracker
This example shows how to configure and use the global nearest neighbor (GNN) tracker.

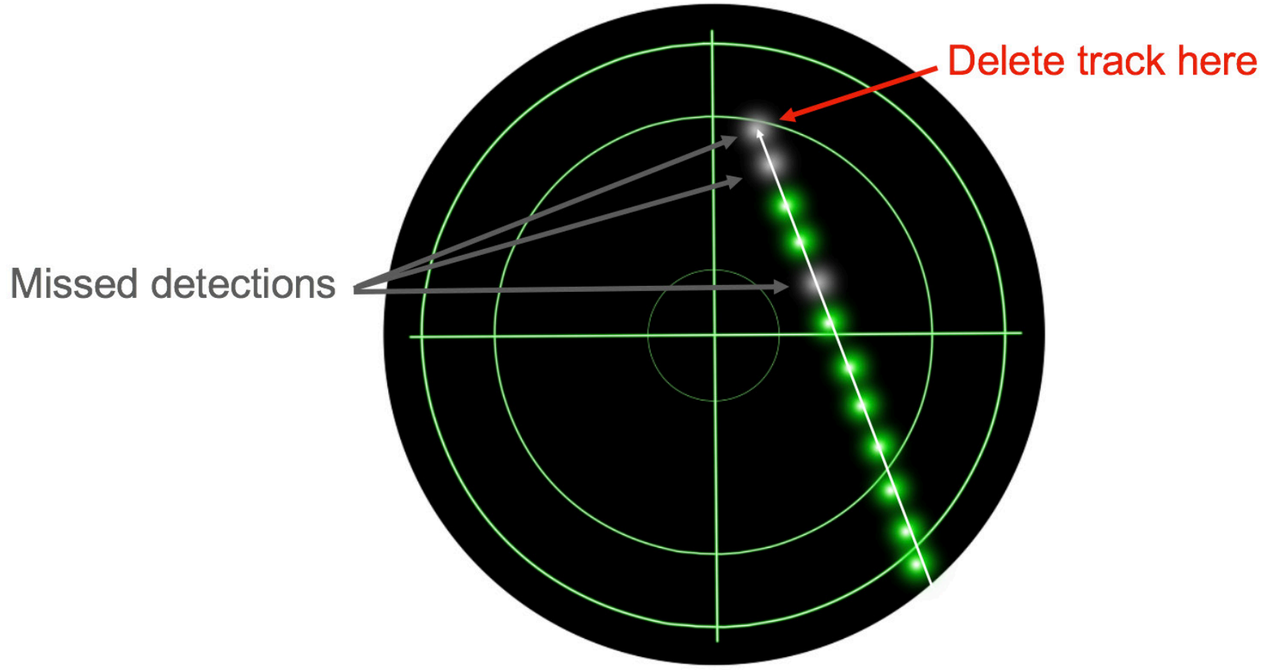
Introduction to Track Logic
This example shows how to define and use confirmation and deletion logic that are based on history or score.

If you'd like to learn more about assignment algorithms, check out the *Sensor Fusion and Tracking Toolbox documentation.*

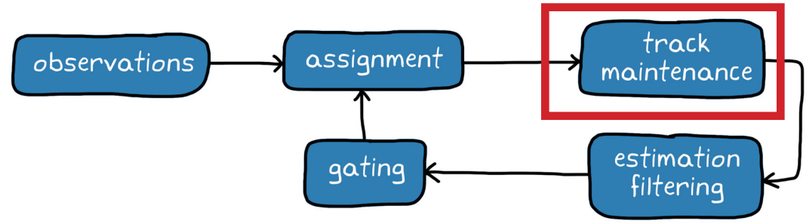
Deleting a Track



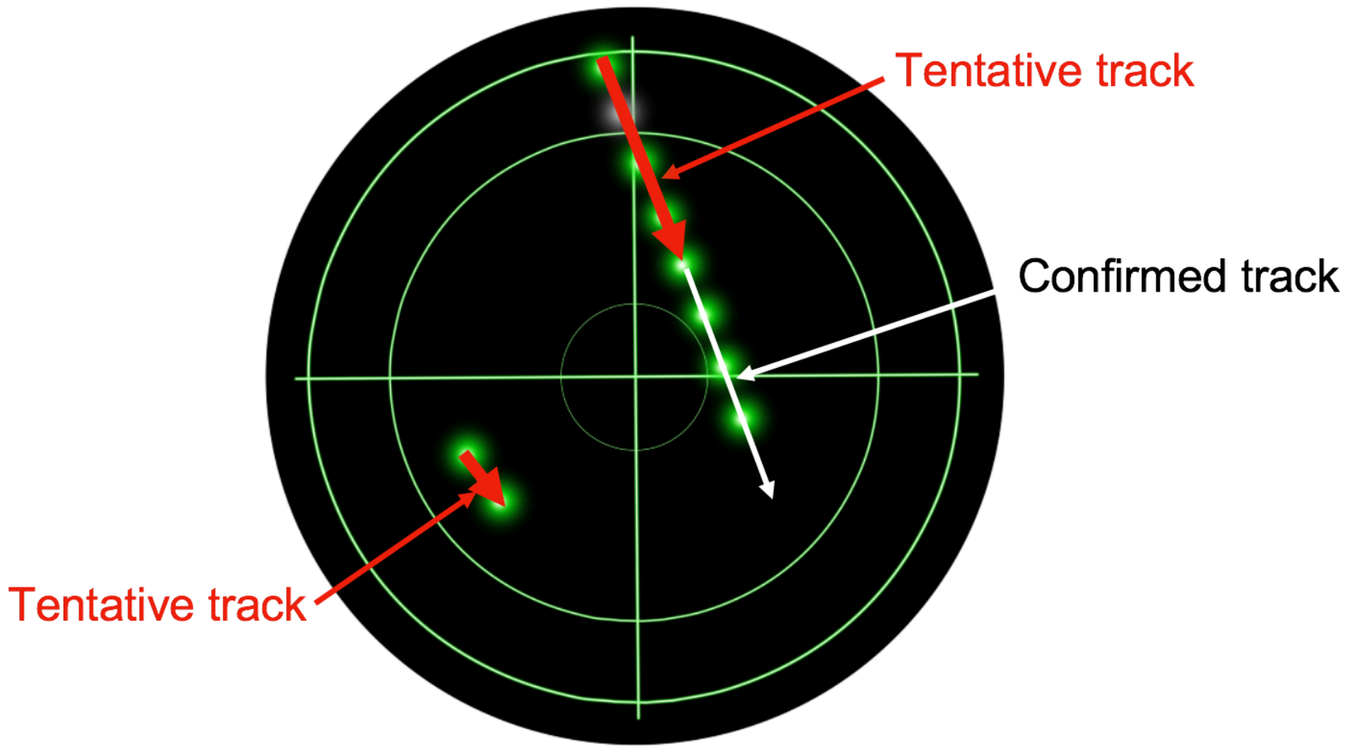
Not all observations get assigned, and not all tracks have observations. This is where track maintenance algorithms are used to delete and create tracks. As we said before, we have to be careful here so we don't do anything prematurely. Let's start with one way to delete a track in a conservative way. Rather than saying an object is gone as soon as we miss a single observation, we could delete a track only if it has not been assigned to a detection at least M times during the last N updates. In this case, M and N are parameters that you can tune to your situation. So, you might decide to delete a track if it hasn't been detected at least three times in the last five updates.



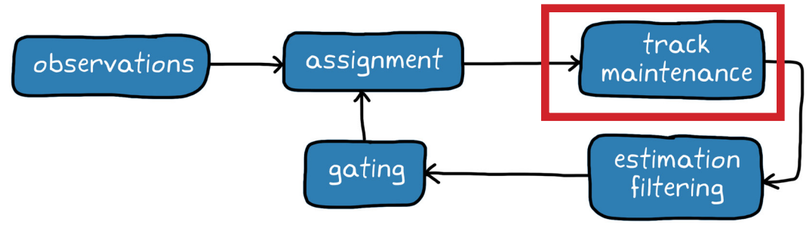
Creating a Track



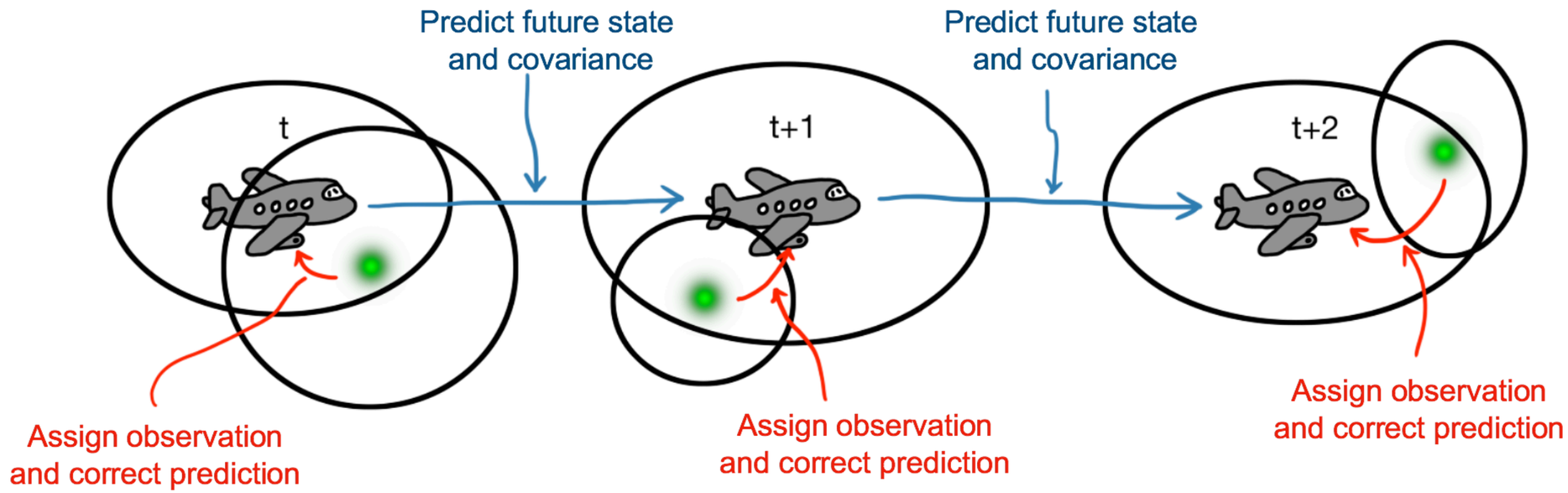
Creating a track is a little trickier because you don't know if a single un-assigned observation is a new object right away, but you still have to pay attention to it so you can figure out over time if it's worth tracking. One way to handle this is to create a tentative track—one that you maintain but you don't treat like a real object. Once the tentative track has been detected M times in the last N updates, you move the track to confirmed. You can remove a tentative track with the same logic as removing a confirmed track. So in this way, you may have dozens of tentative tracks that you are maintaining due to false positive measurements, but are deleted before they ever become confirmed.



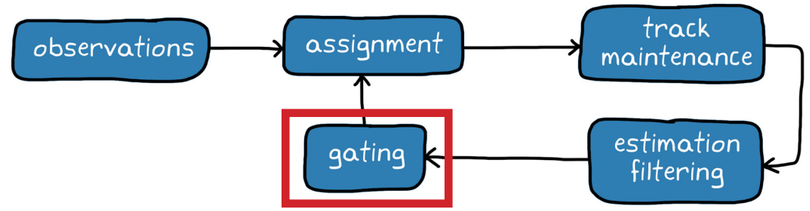
Running the Estimation Filters



With the tracks created and removed and with the observations assigned, we can run a set of estimation filters—one filter for each tracked object. This part is identical to single-object tracking where we had choices like the interacting multiple model filter or the single model Kalman filter. The predicted state of each tracked object that is assigned an observation (both tentative and confirmed objects) gets updated with its respective observation. After that, the whole process starts anew. We get more observations, they are assigned to tracks, tracks are created and deleted, and the filters run again.



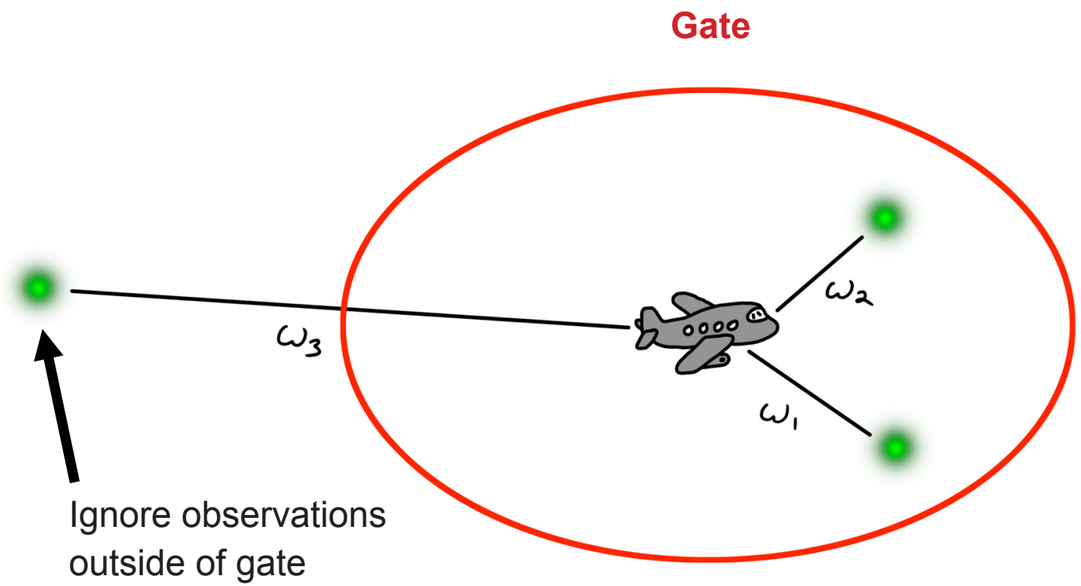
Gating



It would be a computational challenge to look at every observation and consider how likely it is to be assigned to every track. Therefore, we may choose to ignore observations outside of a defined region for each track. This is called *gating*, and it's a screening mechanism that determines which detections are valid candidates to look at for assignment and which should be ignored.

For example, with JPDA, an observation that is far away from the tracked object would statistically contribute very little to the overall solution, so why spend the computational resources to calculate this minuscule amount? If you're tracking dozens or hundreds of objects, this could be extremely wasteful. By ignoring observations outside of a specific region—outside of this gate—we can speed up the assignment process.

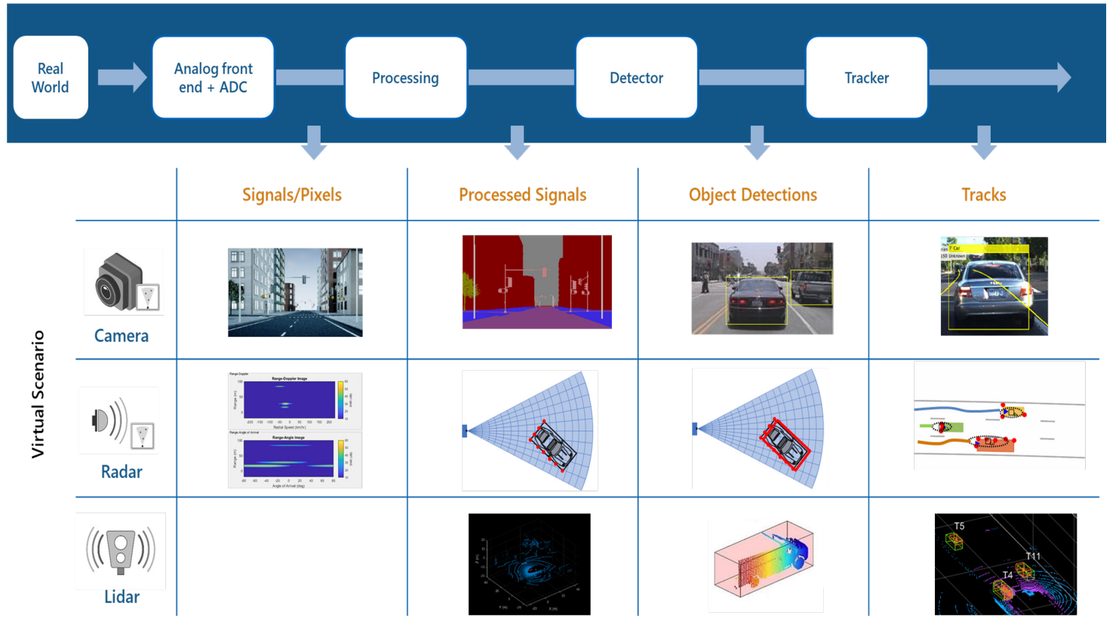
In this way, gating impacts the assignment algorithms so that they consider only the observations that are worth looking at.

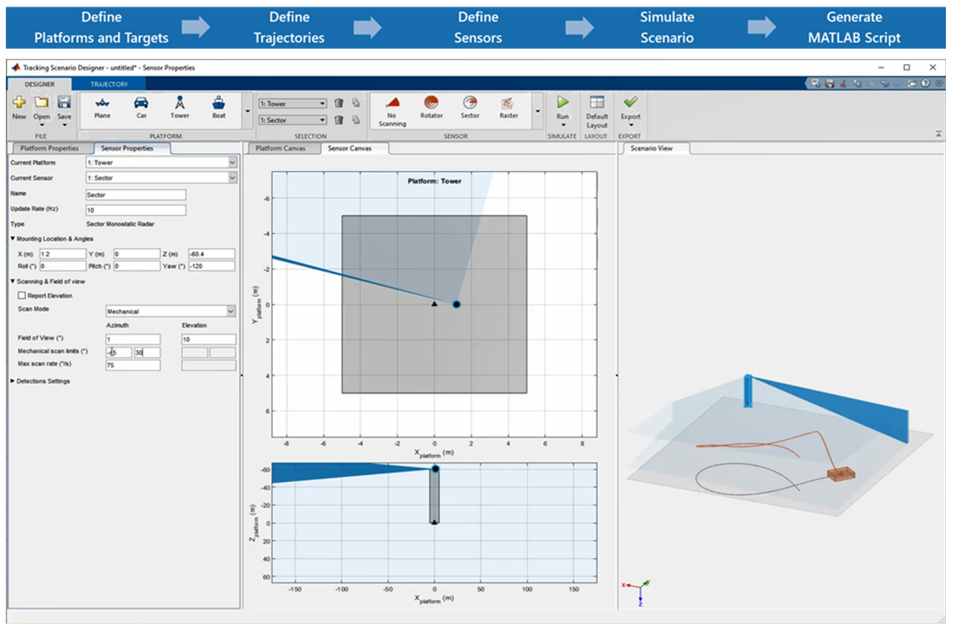


Evaluating Tracking Performance

Before you start your own multi-object tracking system design, it is essential to devise a method to evaluate performance against ground-truth scenarios. You can build these types of scenarios in *MATLAB®* and *Sensor Fusion and Tracking Toolbox™*.

You can use sensor models to augment data from real sensors and build scenarios to focus testing on the edge cases that stress your algorithm performance.





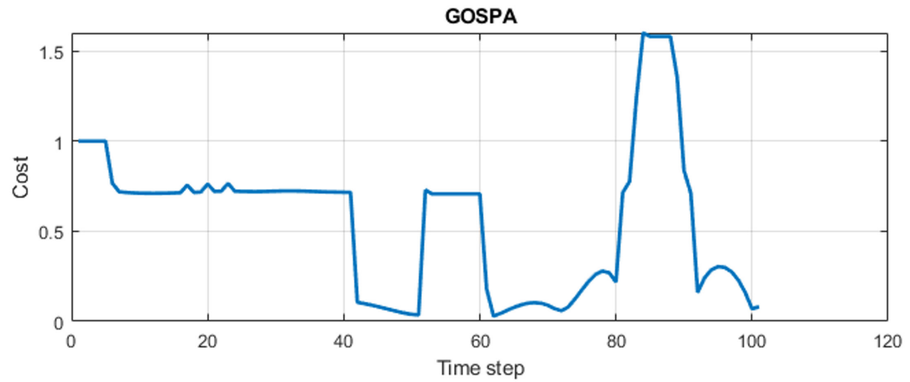
Synthesizing sensor data and developing tracking scenarios for system verification.

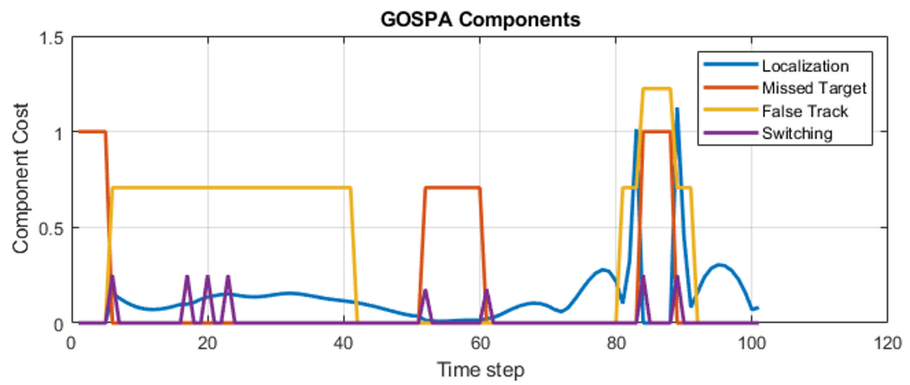
Evaluating Tracking Performance *continued*

With ground-truth scenarios available, the analysis of track metrics, including positional accuracy and the quality of data association, is possible. The accuracy of the assignment and metrics are evaluated by comparing the state of the tracking results with the ground truth.

Integrated metrics like the optimal subpattern association (OSPA) and the generalized optimal subpattern association (GOSPA) are frequently used by the autonomous systems community as a measure of tracker performance.

The main benefit of using integrated metrics is that they support automated testing, which is crucial in running the many thousands of test cases needed for verification and validation of autonomous systems. The integrated metrics provide a single score that can easily be compared with test criteria to provide a pass/fail result. In addition, if configured correctly, both OSPA and GOSPA can be broken into their components to enable quick investigation of failures.



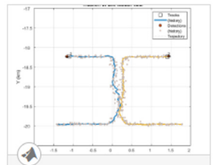


Just the Beginning

As you can see, multi-object tracking, along with sensor fusion, is at the heart of the perception component of autonomous and surveillance systems. This helps makes it an interesting and rewarding topic to learn.

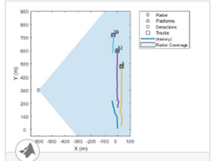
Explore examples to get started.

Multi-Object Trackers



Tracking Closely Spaced Targets Under Ambiguity

Track objects when the association of sensor detections to tracks is ambiguous. In this example, you use a single-hypothesis tracker, a

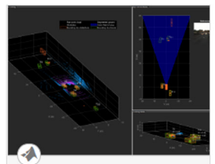


Tuning a Multi-Object Tracker

Tune and run a tracker to track multiple objects in the scene. The example explains and demonstrates the importance of key properties of

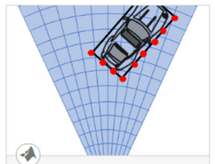
Tracking for Autonomous Systems

Track extended objects and fuse tracks from multiple tracking sources



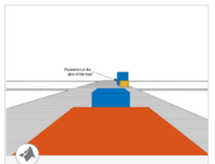
Track Vehicles Using Lidar: From Point Cloud to Track List

Track vehicles using measurements from a lidar sensor mounted on top of an ego vehicle. Lidar sensors report measurements as a point



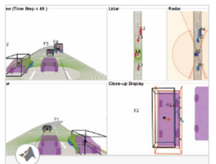
Extended Object Tracking and Performance Metrics Evaluation

Track extended objects. Extended objects are objects whose dimensions span multiple sensor resolution cells. As a result, the



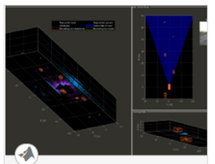
Track-to-Track Fusion for Automotive Safety Applications

Fuse tracks from two vehicles in order to provide a more comprehensive estimate of the environment that can be seen by



Track-Level Fusion of Radar and Lidar Data

Generate an object-level track list from measurements of a radar and a lidar sensor and further fuse them using a track-level fusion scheme.

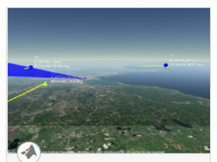


Track Vehicles Using Lidar Data in Simulink

Track vehicles using measurements from a lidar sensor mounted on top of an ego vehicle. Due to high resolution capabilities of the lidar

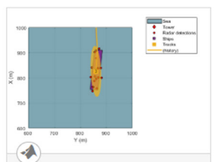
Tracking for Surveillance Systems

Track targets in surveillance region using active and passive sensor detections



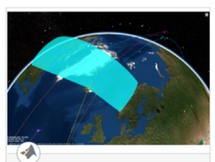
Air Traffic Control

Generate an air traffic control scenario, simulate radar detections from an airport surveillance radar (ASR), and configure a global



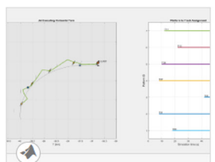
Marine Surveillance Using a PHD Tracker

Generate a marine scenario, simulate radar detections from a marine surveillance radar, and configure a multi-target Probability



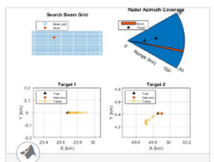
Track Space Debris Using a Keplerian Motion Model

Model earth-centric trajectories using custom motion models within trackingScenario, how to configure a monostatic radar sensor to generate



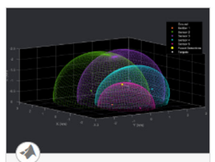
Multiplatform Radar Detection Fusion

Fuse radar detections from a multiplatform radar network. The network includes two airborne and one ground-based long-range radar



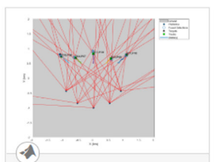
Search and Track Scheduling for Multifunction Phased Arr…

Simulate a multifunction phased array radar system. A multifunction radar can perform jobs that usually require multiple traditional radars.



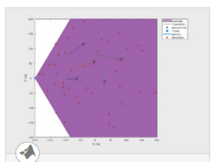
Tracking Using Bistatic Range Detections

Simulate bistatic range-only detections using four sensor-emitter pairs. In addition, this example demonstrates how to localize and



Tracking Using Distributed Synchronous Passive Sensors

Illustrates the tracking of objects using measurements from spatially-distributed and synchronous passive sensors. In the Passive Ranging



Track Point Targets in Dense Clutter Using GM-PHD Tracker

Track points targets in dense clutter using a Gaussian mixture probability hypothesis density (GM-PHD) tracker using a constant velocity

Glossary

Clutter: Undesirable detections typically returned from environmental conditions in a sensor's field of regard.

Detection/observation: Observed or measured quantities reported from a sensor. These may contain measured kinematic quantities (e.g., range, line of sight, and range-rate, and for extended objects shape and orientation) and measured attributes (e.g., target type, identification number), in addition to the time the measurements are obtained.

False alarm: A detection that the sensor reports where a true object does not exist. False alarms introduce additional possible assignments and therefore increase the complexity of data assignment.

Gating: Screening mechanism used to determine which detections are valid candidates to update existing tracks. Gates help to reduce unnecessary computations in track-to-detection assignment.

Sensor resolution: Determines the sensor's ability to distinguish between detections from two targets.

Track confirmation: The use of new detections to confirmation that a new track is estimating a real object. This is done by a track logic; for example, M-out-of-N logic.

Track deletion: Deletion of a track if it is not updated within some reasonable time.

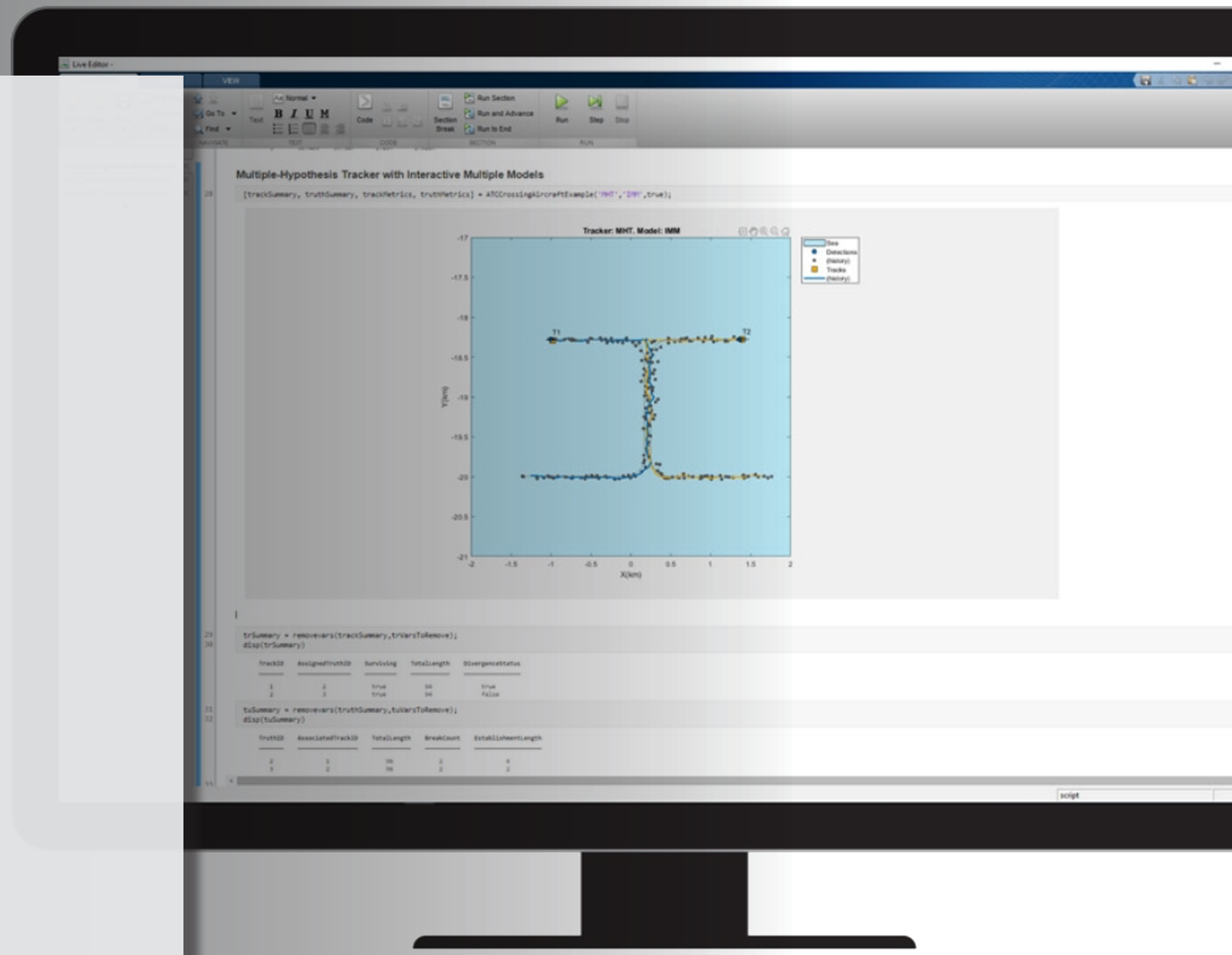
Track initiation: Creation of a new track. When a detection is not assigned to an existing track, a new track might need to be created. Usually, a track is initiated as tentative, indicating that the tracker cannot ascertain whether it estimates a real object or a false alarm.

Track logic: Algorithm used by the tracker to define the rules of confirmation and deletion of tracks.

Track maintenance: A multi-object tracker function that includes track initiation, confirmation, and deletion.

Learn More

Sensor Fusion and Tracking Toolbox – Product Overview
Understanding Sensor Fusion and Tracking – Video Series
Multi-Object Trackers – Documentation



MathWorks®