**Overview of the Kalman Filter Algorithm Map**

Graphical user interface, diagram

Description automatically generated

For your reference: a map of the Kalman Filter algorithm! Keep an eye out, because we'll add a little bit more detail to this later.

Imagine you are in a car equipped with sensors on the outside. The car sensors can detect objects moving around: for example, the sensors might detect a pedestrian, as described in the video, or even a bicycle. For variety, let's step through the Kalman Filter algorithm using the bicycle example.

The Kalman Filter algorithm will go through the following steps:

* **first measurement** - the filter will receive initial measurements of the bicycle's position relative to the car. These measurements will come from a radar or lidar sensor.
* **initialize state and covariance matrices** - the filter will initialize the bicycle's position based on the first measurement.
* then the car will receive another sensor measurement after a time period \Delta{t}Δ*t*.
* **predict** - the algorithm will predict where the bicycle will be after time \Delta{t}Δ*t*. One basic way to predict the bicycle location after \Delta{t}Δ*t* is to assume the bicycle's velocity is constant; thus the bicycle will have moved velocity \* \Delta{t}Δ*t*. In the extended Kalman filter lesson, we will assume the velocity is constant.
* **update** - the filter compares the "predicted" location with what the sensor measurement says. The predicted location and the measured location are combined to give an updated location. The Kalman filter will put more weight on either the predicted location or the measured location depending on the uncertainty of each value.
* then the car will receive another sensor measurement after a time period \Delta{t}Δ*t*. The algorithm then does another **predict** and **update** step.

**Definition of Variables**

* x*x* is the mean state vector. For an extended Kalman filter, the mean state vector contains information about the object's position and velocity that you are tracking. It is called the "mean" state vector because position and velocity are represented by a gaussian distribution with mean x*x*.
* P*P* is the state covariance matrix, which contains information about the uncertainty of the object's position and velocity. You can think of it as containing standard deviations.
* k represents time steps. So x\_k*xk*​ refers to the object's position and velocity vector at time k.
* The notation k+1|k*k*+1∣*k* refers to the prediction step. At time k+1*k*+1, you receive a sensor measurement. Before taking into account the sensor measurement to update your belief about the object's position and velocity, you predict where you think the object will be at time k+1*k*+1. You can predict the position of the object at k+1*k*+1 based on its position and velocity at time k*k*. Hence x\_{k+1|k}*xk*+1∣*k*​ means that you have predicted where the object will be at k+1*k*+1 but have not yet taken the sensor measurement into account.
* x\_{k+1}*xk*+1​ means that you have now predicted where the object will be at time k+1*k*+1 and then used the sensor measurement to update the object's position and velocity.

Graphical user interface

Description automatically generated

Graphical user interface, text, application, email, Teams

Description automatically generated

### Additional Info about the Last Quiz

Diagram

Description automatically generated

Because we have already run a prediction-update iteration with the first sensor at time k+3, the output of the second prediction at time k+3 will actually be identical to the output from the update step with the first sensor. So, in theory, you could skip the second prediction step and just run a prediction, update, update iteration.

But you'll learn more about that later. First, a bit of math.

### Kalman Filter Intuition

The Kalman equation contains many variables, so here is a high level overview to get some intuition about what the Kalman filter is doing.

##### Prediction

Let's say we know an object's current position and velocity , which we keep in the x variable. Now one second has passed. We can predict where the object will be one second later because we knew the object position and velocity one second ago; we'll just assume the object kept going at the same velocity.

The x' = Fx + \nu*x*′=*Fx*+*ν* equation does these prediction calculations for us.

But maybe the object didn't maintain the exact same velocity. Maybe the object changed direction, accelerated or decelerated. So when we predict the position one second later, our uncertainty increases. P' = FPF^T + Q*P*′=*FPFT*+*Q* represents this increase in uncertainty.

Process noise refers to the uncertainty in the prediction step. We assume the object travels at a constant velocity, but in reality, the object might accelerate or decelerate. The notation \nu \sim N(0, Q)*ν*∼*N*(0,*Q*) defines the process noise as a gaussian distribution with mean zero and covariance Q.

##### Update

Now we get some sensor information that tells where the object is relative to the car. First we compare where we think we are with what the sensor data tells us y = z - Hx'*y*=*z*−*Hx*′.

The K*K* matrix, often called the Kalman filter gain, combines the uncertainty of where we think we are P'*P*′ with the uncertainty of our sensor measurement R*R*. If our sensor measurements are very uncertain (R is high relative to P'), then the Kalman filter will give more weight to where we think we are: x'*x*′. If where we think we are is uncertain (P' is high relative to R), the Kalman filter will put more weight on the sensor measurement: z*z*.

Measurement noise refers to uncertainty in sensor measurements. The notation \omega \sim N(0, R)*ω*∼*N*(0,*R*) defines the measurement noise as a gaussian distribution with mean zero and covariance R. Measurement noise comes from uncertainty in sensor measurements.

### A Note About the State Transition Function: Bu

If you go back to the video, you'll notice that the state transition function was first given as x' = Fx + Bu + \nu*x*′=*Fx*+*Bu*+*ν*.

But then Bu*Bu* was crossed out leaving x' = Fx + \nu*x*′=*Fx*+*ν*.

B*B* is a matrix called the control input matrix and u*u* is the control vector.

As an example, let's say we were tracking a car and we knew for certain how much the car's motor was going to accelerate or decelerate over time; in other words, we had an equation to model the exact amount of acceleration at any given moment. Bu*Bu* would represent the updated position of the car due to the internal force of the motor. We would use \nu*ν* to represent any random noise that we could not precisely predict like if the car slipped on the road or a strong wind moved the car.

For the Kalman filter lessons, we will assume that there is no way to measure or know the exact acceleration of a tracked object. For example, if we were in an autonomous vehicle tracking a bicycle, pedestrian or another car, we would not be able to model the internal forces of the other object; hence, we do not know for certain what the other object's acceleration is. Instead, we will set Bu = 0*Bu*=0 and represent acceleration as a random noise with mean \nu*ν*.

### Kalman Filter Equations in C++

Now, let's do a quick refresher of the Kalman Filter for a simple 1D motion case. Let's say that your goal is to track a pedestrian with state x*x* that is described by a position and velocity.

x = \begin{pmatrix} p \\ v \end{pmatrix}*x*=(*pv*​)

##### Prediction Step

When designing the Kalman filter, we have to define the two linear functions: the state transition function and the measurement function. The state transition function is

x' = F\*x + noise*x*′=*F*∗*x*+*noise*,

where,

F = \begin{pmatrix} 1 & \Delta t \\ 0 & 1 \end{pmatrix}*F*=(10​Δ*t*1​)

and x'*x*′ is where we predict the object to be after time \Delta tΔ*t*.

F*F* is a matrix that, when multiplied with x*x*, predicts where the object will be after time \Delta tΔ*t*.

By using the linear motion model with a constant velocity, the new location, p'*p*′ is calculated as

p' = p + v \* \Delta t*p*′=*p*+*v*∗Δ*t*,

where p*p* is the old location and v*v*, the velocity, will be the same as the new velocity (v' = v*v*′=*v*) because the velocity is constant.

We can express this in a matrix form as follows:

\begin{pmatrix} p' \\ v' \end{pmatrix} = \begin{pmatrix}1 & \Delta t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} p \\ v \end{pmatrix}(*p*′*v*′​)=(10​Δ*t*1​)(*pv*​)

Remember we are representing the object location and velocity as gaussian distributions with mean x*x*. When working with the equation x' = F\*x + noise*x*′=*F*∗*x*+*noise*\*, we are calculating the mean value of the state vector. The noise is also represented by a gaussian distribution but with mean zero; hence, noise = 0 is saying that the mean noise is zero. The equation then becomes x' = F\*x*x*′=*F*∗*x*

But the noise does have uncertainty. The uncertainty shows up in the Q*Q* matrix as acceleration noise.

##### Update Step

For the update step, we use the measurement function to map the state vector into the measurement space of the sensor. To give a concrete example, lidar only measures an object's position. But the extended Kalman filter models an object's position and velocity. So multiplying by the measurement function H matrix will drop the velocity information from the state vector x*x*. Then the lidar measurement position and our belief about the object's position can be compared.

z = H\*x + w*z*=*H*∗*x*+*w*

where w*w* represents sensor measurement noise.

So for lidar, the measurement function looks like this:

z = p'*z*=*p*′.

It also can be represented in a matrix form:

z=\begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} p' \\ v' \end{pmatrix}*z*=(1​0​)(*p*′*v*′​).

As we already know, the general algorithm is composed of a prediction step where I predict the new state and covariance, P*P*.

And we also have a measurement update (or also called many times a correction step) where we use the latest measurements to update our estimate and our uncertainty.

Here is a download link to the [Eigen Library](https://d17h27t6h515a5.cloudfront.net/topher/2017/March/58b7604e_eigen/eigen.zip) that is being used throughout the programming assignments. Further details regarding this library can be found [here](http://eigen.tuxfamily.org/).

Note: In the classroom editor we are calling just Dense instead of Eigen/Dense as seen in videos. This is because the Eigen library had to have its folder structure reformatted to work with the programming quiz editor. If you run the code on your own computer you would still use Eigen/Dense.

**Notes for using the Eigen Library:**

You can create a vertical vector of two elements with a command like this:

VectorXd my\_vector(2);

You can use the so called comma initializer to set all the coefficients to some values:

my\_vector << 10, 20;

and you can use the cout command to print out the vector:

cout << my\_vector << endl;

The matrices can be created in the same way. For example, This is an initialization of a 2 by 2 matrix with the values 1, 2, 3, and 4:

MatrixXd **my\_matrix**(2,2);

my\_matrix << 1, 2,

3, 4;

You can use the same comma initializer or you can set each matrix value explicitly. For example, that's how we can change the matrix elements in the second row:

my\_matrix(1,0) = 11; *//second row, first column*

my\_matrix(1,1) = 12; *//second row, second column*

Also, you can compute the transpose of a matrix with the following command:

MatrixXd my\_matrix\_t = my\_matrix.transpose();

And here is how you can get the matrix inverse:

MatrixXd my\_matrix\_i = my\_matrix.inverse();

For multiplying the matrix m with the vector b you can write this in one line as let’s say matrix c equals m times v:

MatrixXd another\_matrix;

another\_matrix = my\_matrix\*my\_vector;

Note that in the quiz below, in the filter() function, we actually do the measurement and then the prediction in the loop. Over time, the order of these doesn't have a huge impact, since it is just a cycle from one to the other. Here, the first thing you need is a measurement because otherwise there is no location information or even information that the object exists unless a sensor picked it up. So, you initialize location values with the measurement.

### The State Transition Matrix

A picture containing calendar

Description automatically generatedAs a reminder, the above equation is x' = Fx + noise*x*′=*Fx*+*noise*

Motion noise and process noise refer to the same case: uncertainty in the object's position when predicting location. The model assumes velocity is constant between time intervals, but in reality we know that an object's velocity can change due to acceleration. The model includes this uncertainty via the process noise.

Measurement noise refers to uncertainty in sensor measurements, which will be discussed in more detail later.

Graphical user interface, text, application, email

Description automatically generated

From the examples I’ve just showed you we can clearly see that the process noise depends on both: the elapsed time and the uncertainty of acceleration. So, how can we model the process noise by considering both of these factors? Keep going to find out :)

### Calculating Acceleration Noise Parameters

Before we discuss the derivation of the process covariance matrix Q, you might be curious about how to choose values for \sigma\_{ax\_{}}^2*σax*​2​ and \sigma\_{ay}^2*σay*2​.

For the extended Kalman filter project, you will be given values for both.

### Process Covariance Matrix Q - Intuition

As a reminder, here are the state covariance matrix update equation and the equation for Q.

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Diagram

Description automatically generated

**Variable Definitions**

To reinforce was what discussed in the video, here is an explanation of what each variable represents:

* **z** is the measurement vector. For a lidar sensor, the z*z* vector contains the position-x*position*−*x* and position-y*position*−*y* measurements.
* **H** is the matrix that projects your belief about the object's current state into the measurement space of the sensor. For lidar, this is a fancy way of saying that we discard velocity information from the state variable since the lidar sensor only measures position: The state vector x*x* contains information about [p\_x, p\_y, v\_x, v\_y][*px*​,*py*​,*vx*​,*vy*​] whereas the z*z* vector will only contain [px, py][*px*,*py*]. Multiplying Hx allows us to compare x, our belief, with z, the sensor measurement.
* What does the prime notation in the x*x* vector represent? The prime notation like p\_x'*px*′​ means you have already done the prediction step but have not done the measurement step yet. In other words, the object was at p\_x*px*​. After time \Delta{t}Δ*t*, you calculate where you believe the object will be based on the motion model and get p\_x'*px*′​.

**H Matrix Quiz**

Find the right H*H* matrix to project from a 4D state to a 2D observation space, as follows:

Text, letter

Description automatically generated

Text

Description automatically generated

Text, letter

Description automatically generated

A picture containing diagram

Description automatically generated

Input Data Link:

If you would like to run this coding exercise on your own computer, here is the [input data](https://d17h27t6h515a5.cloudfront.net/topher/2017/March/58bd257a_ekf-data/ekf-data.zip) that is being used.

### More Info on Timestamps

Graphical user interface, text, application, email

Description automatically generated

Time keeps on tickin', tickin', tickin'...

Timestamps are often used for logging a sequence of events, so that we know exactly, for example, in our case when the measurements were generated.

We can use the timestamp values to compute the elapsed time between two consecutive observations as:

**float** **delta\_t** = ( timestamp(k+1) - timestamp(k) ) / 1000000.0.

Additionally we divide the result by 10^6 to transform it from microseconds to seconds.

**float** dt = (measurement\_pack.timestamp\_ - previous\_timestamp\_) / 1000000.0;

NEXT

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Graphical user interface, application

Description automatically generated

A picture containing chart

Description automatically generated

Follow the arrows from top left to bottom to top right: (1) A Gaussian from 10,000 random values in a normal distribution with a mean of 0. (2) Using a nonlinear function, arctan, to transform each value. (3) The resulting distribution.

Chart, histogram

Description automatically generated

This one looks much better! Notice how the blue graph, the output, remains a Gaussian after applying a first order Taylor expansion.

Graphical user interface, text, application, email

Description automatically generated

A screenshot of a computer

Description automatically generated

The orange line represents the first order Taylor expansion of arctan(x). What is it?

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Text, letter

Description automatically generated

A picture containing text

Description automatically generated

Text, letter

Description automatically generated

A close up of text on a white background

Description automatically generated

**Extended Kalman Filter Equations**

Although the mathematical proof is somewhat complex, it turns out that the Kalman filter equations and extended Kalman filter equations are very similar. The main differences are:

* the F*F* matrix will be replaced by F\_j*Fj*​ when calculating P'*P*′.
* the H*H* matrix in the Kalman filter will be replaced by the Jacobian matrix H\_j*Hj*​ when calculating S*S*, K*K*, and P*P*.
* to calculate x'*x*′, the prediction update function, f*f*, is used instead of the F*F* matrix.
* to calculate y*y*, the h*h* function is used instead of the H*H* matrix.

For this project, however, we do not need to use the f*f* function or F\_j*Fj*​. If we had been using a non-linear model in the prediction step, we would need to replace the F*F* matrix with its Jacobian, F\_j*Fj*​. However, we are using a linear model for the prediction step. So, for the prediction step, we can still use the regular Kalman filter equations and the F*F* matrix rather than the extended Kalman filter equations.

The measurement update for lidar will also use the regular Kalman filter equations, since lidar uses linear equations. Only the measurement update for the radar sensor will use the extended Kalman filter equations.

**One important point to reiterate is that the equation**y = z - Hx'*y*=*z*−*Hx*′**for the Kalman filter does not become**y = z - H\_jx*y*=*z*−*Hj*​*x***for the extended Kalman filter. Instead, for extended Kalman filters, we'll use the h function directly to map predicted locations**x'*x*′**from Cartesian to polar coordinates.**

A picture containing graphical user interface, text, application

Description automatically generated

The comparison for reference.

Text, letter

Description automatically generated

If f and h are linear functions, then the Extended Kalman Filter generates exactly the same result as the standard Kalman Filter. Actually, if f and h are linear then the Extended Kalman Filter F\_j turns into f and H\_j turns into h. All that's left is the same ol' standard Kalman Filter!

In our case we have a linear motion model, but a nonlinear measurement model when we use radar observations. So, we have to compute the Jacobian only for the measurement function.

## Additional Resources on Sensor Fusion and Object Detection & Tracking

Nice work reaching the end of the sensor fusion content! While you still have the project left to do here, we're also providing some additional resources and recent research on the topic that you can come back to if you have time later on.

Reading research papers is a great way to get exposure to the latest and greatest in the field, as well as expand your learning. However, just like the project ahead, it's often best to learn by doing - if you find a paper that really excites you, try to implement it (or even something better) yourself!

##### Optional Reading

All of these are completely optional reading - you could spend days reading through the entirety of these! We suggest moving onto the project first so you have Kalman Filters fresh on your mind, before coming back to check these out.

We've categorized these papers to hopefully help you narrow down which ones might be of interest, as well as highlighted a couple key reads by category by including their Abstract section, which summarizes the paper. We've also included some additional papers you might consider as well if you want to delve even deeper.

### Tracking Multiple Objects and Sensor Fusion

The below papers and resources concern tracking multiple objects, using Kalman Filters as well as other techniques!

[No Blind Spots: Full-Surround Multi-Object Tracking for Autonomous Vehicles using Cameras & LiDARs](https://arxiv.org/pdf/1802.08755.pdf) by A. Rangesh and M. Trivedi

***Abstract:****Online multi-object tracking (MOT) is extremely important for high-level spatial reasoning and path planning for autonomous and highly-automated vehicles. In this paper, we present a modular framework for tracking multiple objects (vehicles), capable of accepting object proposals from different sensor modalities (vision and range) and a variable number of sensors, to produce continuous object tracks. [...] We demonstrate that our framework is well-suited to track objects through entire maneuvers around the ego-vehicle, some of which take more than a few minutes to complete. We also leverage the modularity of our approach by comparing the effects of including/excluding different sensors, changing the total number of sensors, and the quality of object proposals on the final tracking result.*

[Multiple Sensor Fusion and Classification for Moving Object Detection and Tracking](https://hal.archives-ouvertes.fr/hal-01241846/document) by R.O. Chavez-Garcia and O. Aycard

***Abstract:****[...] We believe that by including the objects classification from multiple sensors detections as a key component of the object’s representation and the perception process, we can improve the perceived model of the environment. First, we define a composite object representation to include class information in the core object’s description. Second, we propose a complete perception fusion architecture based on the Evidential framework to solve the Detection and Tracking of Moving Objects (DATMO) problem by integrating the composite representation and uncertainty management. Finally, we integrate our fusion approach in a real-time application inside a vehicle demonstrator from the interactIVe IP European project which includes three main sensors: radar, lidar and camera. [...]*

### Stereo cameras

The below papers cover various methods of using stereo camera set-ups for object detection and tracking.

[Robust 3-D Motion Tracking from Stereo Images: A Model-less Method](http://www.cse.cuhk.edu.hk/~khwong/J2008_IEEE_TIM_Stereo%20Kalman%20.pdf) by Y.K. Yu, et. al.

***Abstract:****Traditional vision-based 3-D motion estimation algorithms require given or calculated 3-D models while the motion is being tracked. We propose a high-speed extended Kalman filter-based approach that recovers camera position and orientation from stereo image sequences without prior knowledge as well as the procedure for the reconstruction of 3-D structures. [...] The proposed method has been applied to recover the motion from stereo image sequences taken by a robot and a hand-held stereo rig. The results are accurate compared to the ground truths. It is shown in the experiment that our algorithm is not susceptible to outlying point features with the application of a validation gate.*

[Vehicle Tracking and Motion Estimation Based on Stereo Vision Sequences](http://hss.ulb.uni-bonn.de/2010/2356/2356.pdf) by A. Barth (long read)

***Abstract:****In this dissertation, a novel approach for estimating trajectories of road vehicles such as cars, vans, or motorbikes, based on stereo image sequences is presented. Moving objects are detected and reliably tracked in real-time from within a moving car. [...] The focus of this contribution is on oncoming traffic, while most existing work in the literature addresses tracking the lead vehicle. The overall approach is generic and scalable to a variety of traffic scenes including inner city, country road, and highway scenarios. [...] The key idea is to derive these parameters from a set of tracked 3D points on the object’s surface, which are registered to a time-consistent object coordinate system, by means of an extended Kalman filter. Combining the rigid 3D point cloud model with the dynamic model of a vehicle is one main contribution of this thesis. [...] The experimental results show the proposed system is able to accurately estimate the object pose and motion parameters in a variety of challenging situations, including night scenes, quick turn maneuvers, and partial occlusions.*

### Deep Learning-based approaches

The below papers include various deep learning-based approaches to 3D object detection and tracking.

[Fast and Furious: Real Time End-to-End 3D Detection, Tracking and Motion Forecasting with a Single Convolutional Net](http://openaccess.thecvf.com/content_cvpr_2018/papers/Luo_Fast_and_Furious_CVPR_2018_paper.pdf) by W. Luo, et. al.

***Abstract:****In this paper we propose a novel deep neural network that is able to jointly reason about 3D detection, tracking and motion forecasting given data captured by a 3D sensor. By jointly reasoning about these tasks, our holistic approach is more robust to occlusion as well as sparse data at range. Our approach performs 3D convolutions across space and time over a bird’s eye view representation of the 3D world, which is very efficient in terms of both memory and computation. Our experiments on a new very large scale dataset captured in several north american cities, show that we can outperform the state-of-the-art by a large margin. Importantly, by sharing computation we can perform all tasks in as little as 30 ms.*

[VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection](https://arxiv.org/abs/1711.06396) by Y. Zhou and O. Tuzel

***Abstract:****Accurate detection of objects in 3D point clouds is a central problem in many applications, such as autonomous navigation, housekeeping robots, and augmented/virtual reality. To interface a highly sparse LiDAR point cloud with a region proposal network (RPN), most existing efforts have focused on hand-crafted feature representations, for example, a bird's eye view projection. In this work, we remove the need of manual feature engineering for 3D point clouds and propose VoxelNet, a generic 3D detection network that unifies feature extraction and bounding box prediction into a single stage, end-to-end trainable deep network. [...] Experiments on the KITTI car detection benchmark show that VoxelNet outperforms the state-of-the-art LiDAR based 3D detection methods by a large margin. Furthermore, our network learns an effective discriminative representation of objects with various geometries, leading to encouraging results in 3D detection of pedestrians and cyclists, based on only LiDAR.*

### Other papers on Tracking Multiple Objects and Sensor Fusion

The below papers and resources concern tracking multiple objects, using Kalman Filters as well as other techniques! We have not included the abstracts here for brevity, but you should check those out first to see which of these you want to take a look at.

[*Multiple Object Tracking using Kalman Filter and Optical Flow*](http://www.ejaet.com/PDF/2-2/EJAET-2-2-34-39.pdf)*by S. Shantaiya,*et. al.

[*Kalman Filter Based Multiple Objects Detection-Tracking Algorithm Robust to Occlusion*](https://pdfs.semanticscholar.org/f5a2/bf3df3126d2923a617b977ec2b4e1c829a08.pdf)*by J-M Jeong,*et. al.

[*Tracking Multiple Moving Objects Using Unscented Kalman Filtering Techniques*](https://arxiv.org/pdf/1802.01235.pdf)*by X. Chen,*et. al.

[*LIDAR-based 3D Object Perception*](https://velodynelidar.com/lidar/hdlpressroom/pdf/Articles/LIDAR-based%203D%20Object%20Perception.pdf)*by M. Himmelsbach,*et. al

[*Fast multiple objects detection and tracking fusing color camera and 3D LIDAR for intelligent vehicles*](https://www.researchgate.net/publication/309503024_Fast_multiple_objects_detection_and_tracking_fusing_color_camera_and_3D_LIDAR_for_intelligent_vehicles)*by S. Hwang,*et. al.

[*3D-LIDAR Multi Object Tracking for Autonomous Driving*](https://repository.tudelft.nl/islandora/object/uuid%3Af536b829-42ae-41d5-968d-13bbaa4ec736)*by A.S. Rachman (long read)*