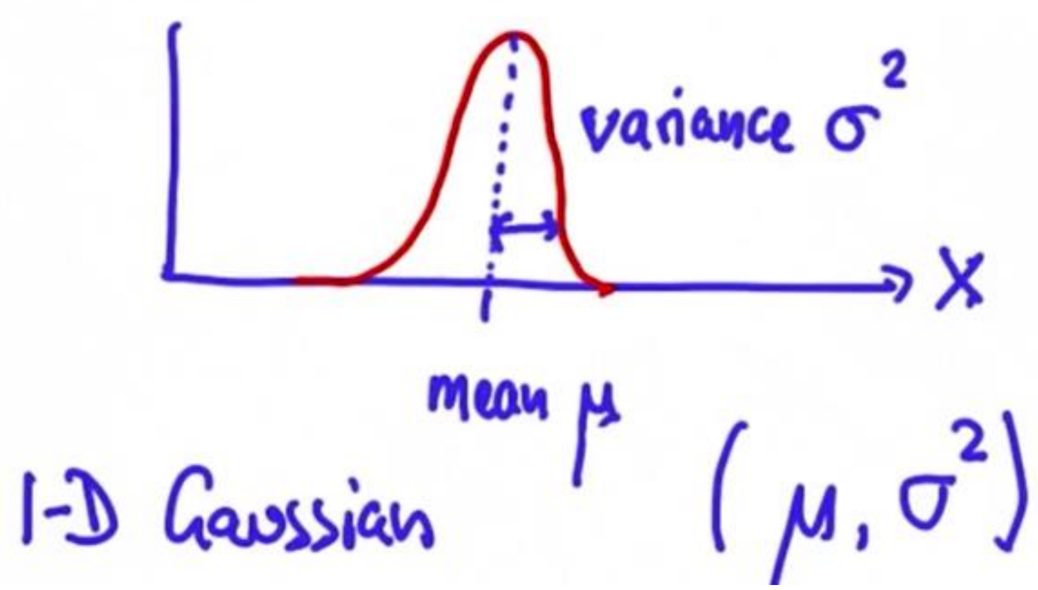
In Kalman filters the distribution is given by what we called a Gaussian and a Gaussian is a continuous function over the space of locations in the area underneath sums up to one.

A Gaussian is characterized by two parameters:

- The mean often abbreviated with the Greek letter Mu
- The width of the Gaussian often called Variance.



The formula of the Gaussian is presented below:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}}$$

To track objects, the Kalman Filter represents our distributions by Gaussians and iterates on two main cycles. The key concepts from these cycles referenced in the below:

The first cycle is the Measurement Update.

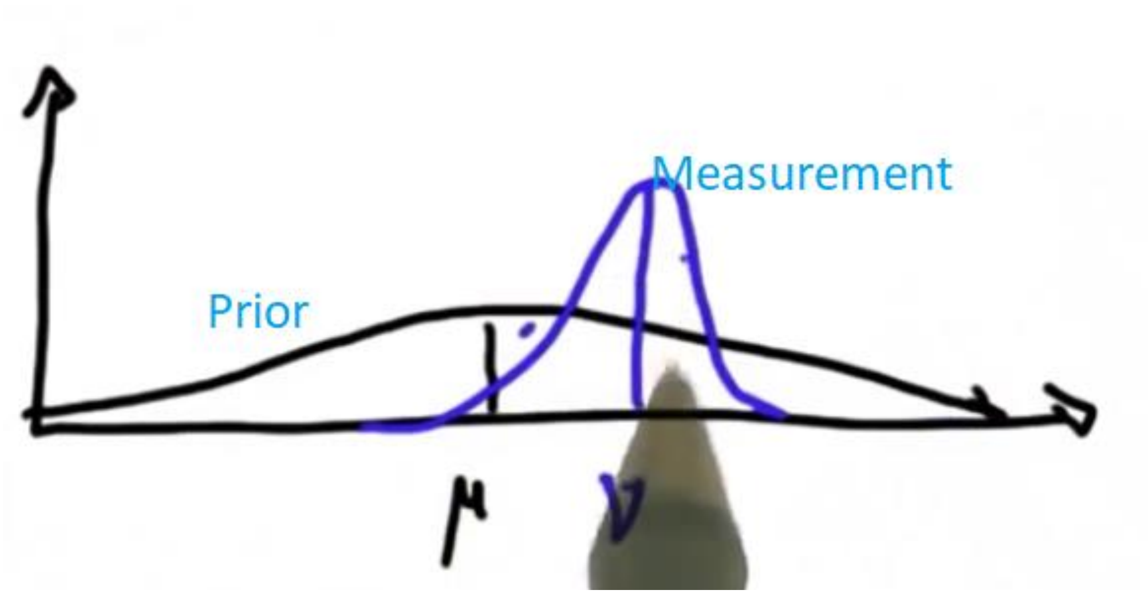
- Requires a product
- Uses Bayes rule.

The second cycle is the Motion Update (Prediction).

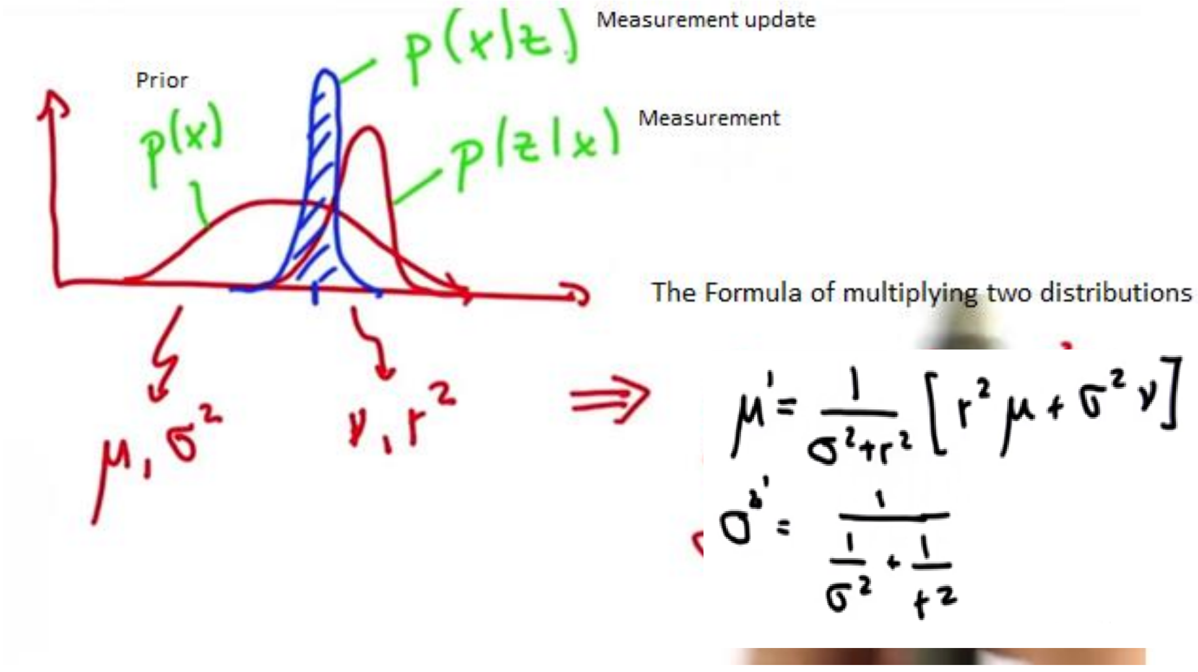
- Involves a convolution
- Uses total probability.

The first cycle is the Measurement Update:

I am going to talk about the measurement cycle using Gaussians: Suppose we are localizing another vehicle which has a distribution (called prior) that looks like as follow (black line) now we get a measurement that tell us something about localization of the vehicle (blue line) .

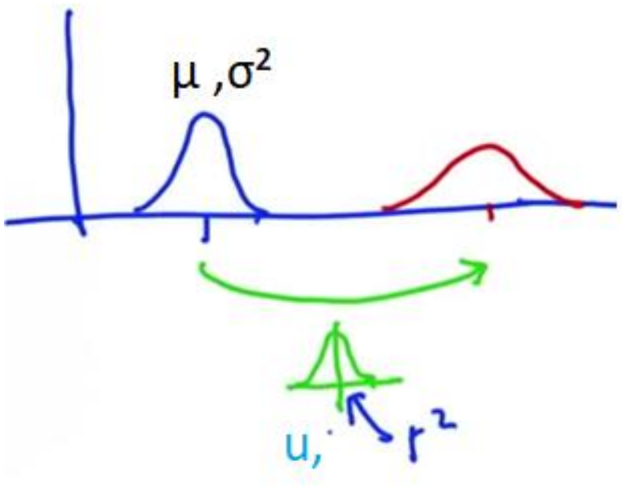


By multiplying two Gaussians (prior and Measurement), the new mean and covariance of the subsequent Gaussian (measurement update) has a smaller covariance then the measurement and prior distribution (Notice: wide covariance means we are uncertain about localization and smaller covariance means we are more certain about localization)


$p(x|z)$
Measurement update
Prior
$p(x)$
$p(z|x)$
Measurement
The Formula of multiplying two distributions
μ, σ^2
ν, r^2
$\Rightarrow$
$\mu' = \frac{1}{\sigma^2 + r^2}[r^2\mu + \sigma^2\nu]$
$\sigma^{2'} = \frac{1}{\frac{1}{\sigma^2} + \frac{1}{r^2}}$

The second cycle is the Motion Update (Prediction):

Suppose the robot lives in the life like below (blue line) and want to move to the right side with a certain distance, the motion(green line) has itself its own set of uncertainty r2 (because the motion tends to lost information), which adds to the uncertainty of the current uncertainty σ2 and leads to a new Gaussian with high uncertainty σ2 prime (red line):



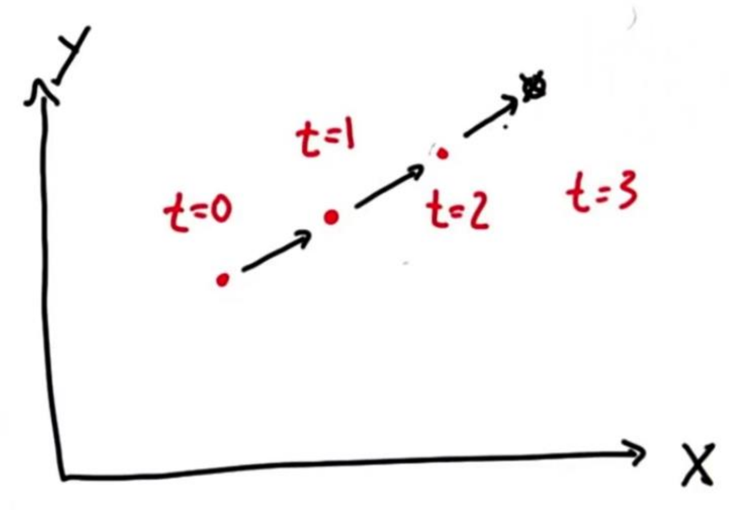
This leads to increased uncertainty over the initial uncertainty and the maths for this is really easy:

$$\mu' \leftarrow \mu + u$$

$$\sigma^{2\prime} \leftarrow \sigma^2 + r^2.$$

That was a full kalman filter for 1D case but in the reality, we have many dimensions:

Suppose we have 2D state (x and y position) in our case, you might have a car that uses a Radar to detect the location of other vehicles over time (the sensor itself only sees positions and it never sees the actual velocity), what the 2D kalman filter affords you is something amazing.



At time t=0 you observe the object of interest to be at the coordinate t=0, one time step later you see over here (t=1) and so on. A Kalman filter allows you to figure out what the velocity of the object is and uses the velocity estimate to make a good prediction about the future location of an object (the velocity is inferred from seeing multiple positions).

To explain a kalman filter for many dimension I have to explain high dimensional Gaussian, which often called Multivariate Gaussian:

The mean is now a vector for each of the dimensions:

$$\mu = \begin{pmatrix} \mu_0 \\ \vdots \\ \mu_D \end{pmatrix}$$

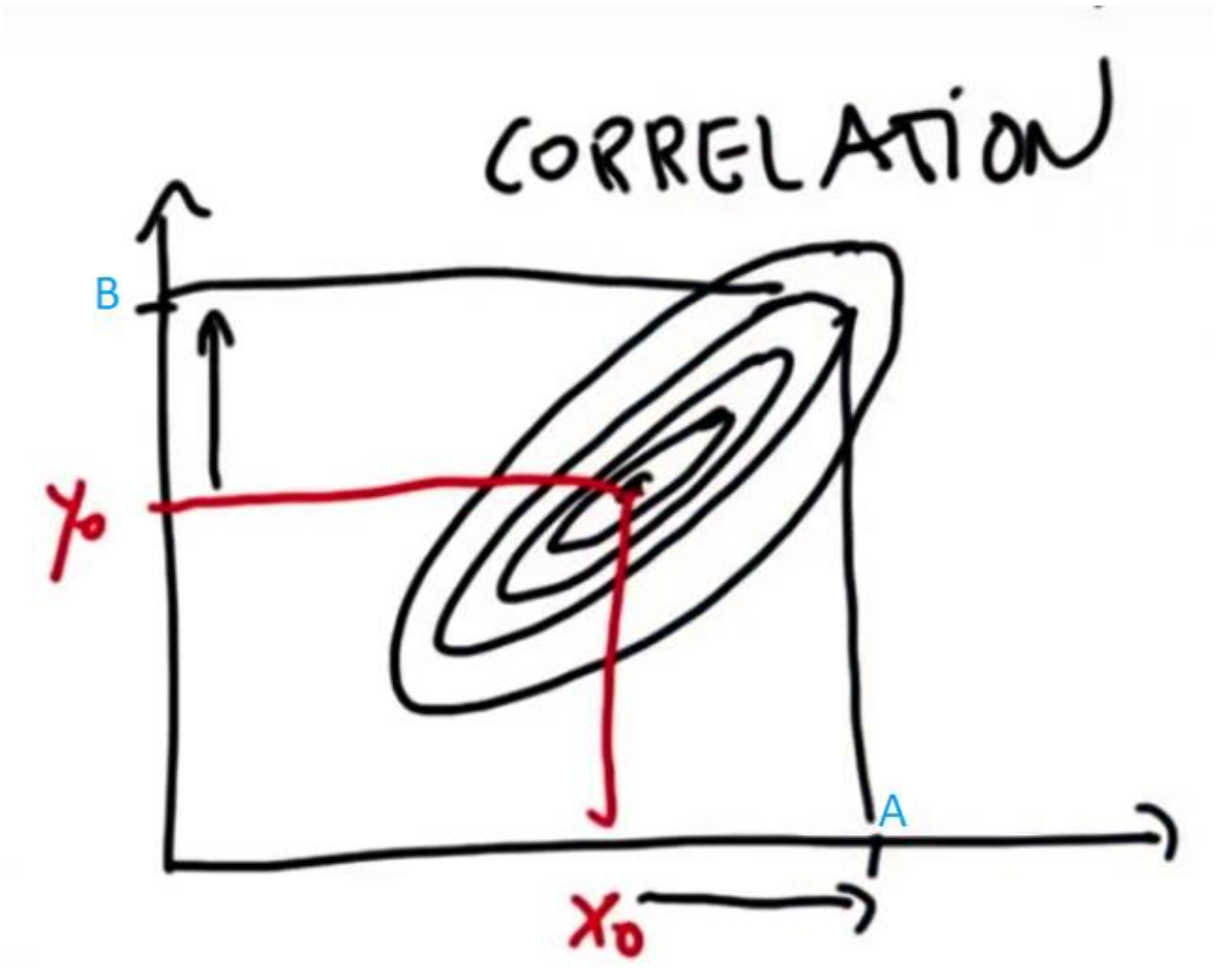
The variance is replace by what called a covariance and it is a matrix with D rows and D columns (if dimensionality of the estimate is D)

$$\Sigma = \begin{pmatrix} \cdot & \cdots & \cdot \\ \vdots & & \\ \cdot & & \end{pmatrix}$$

The formula is:

$$(2\pi)^{-D/2} \, |\Sigma|^{-\frac{1}{2}} \exp -\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu$$

For two-dimensional space, a two dimensional Gaussian is defined over the space (it is possible to draw the contour lines of a Gaussian), the mean of the Gaussian is x and y pair and the covariance defined over the spread of the Gaussian. When the Gaussian is tilted as showed the uncertainty of x and y is correlated, which mean if I get information about x (point A) that make me believe that y probably sits at coordinate B.

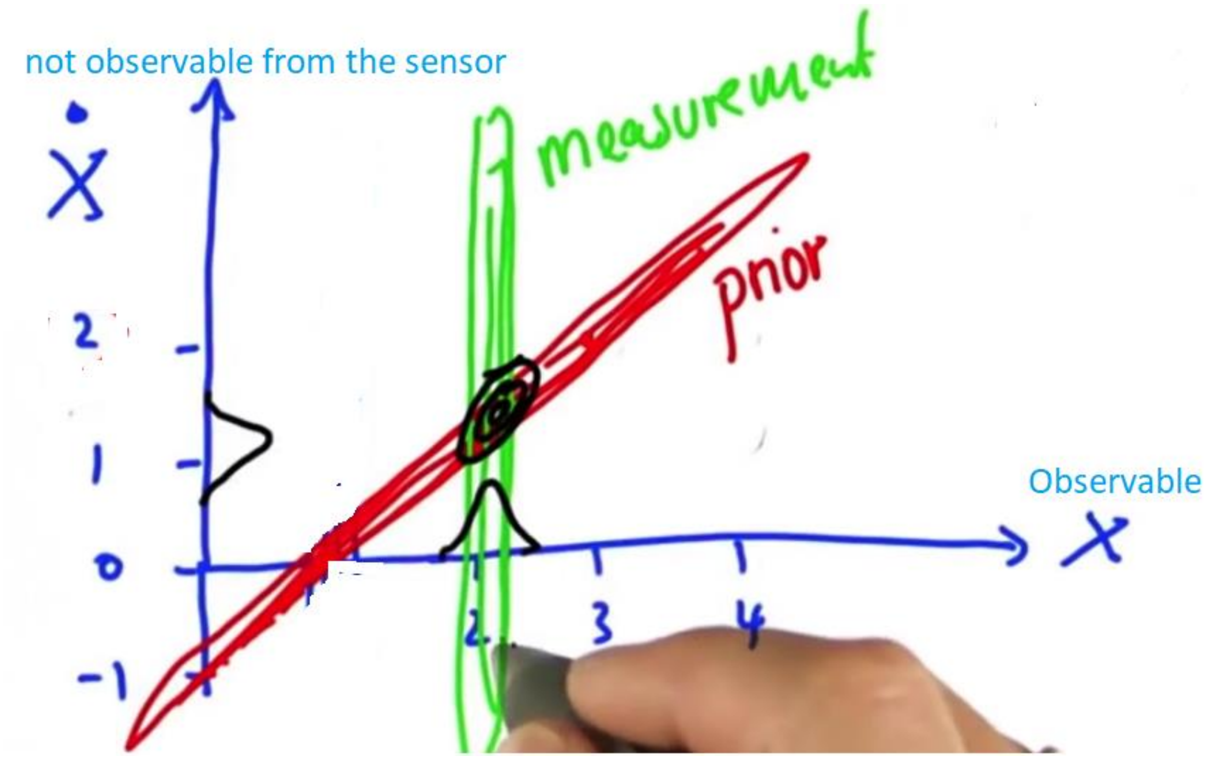


Another Example that can explain a Kalman filter is presented below:

Suppose we have two dimensions (one for the location, which is observable from the sensor and denoted by x, and one for the velocity, which is not observable from the sensor and denoted by x dot)

- you have a correlation Gaussian called prior (we know our locations is correlated to the velocity, much faster I move, the further is the location)
- you got a new measurement about the location (measurement) but you know nothing about the velocity)

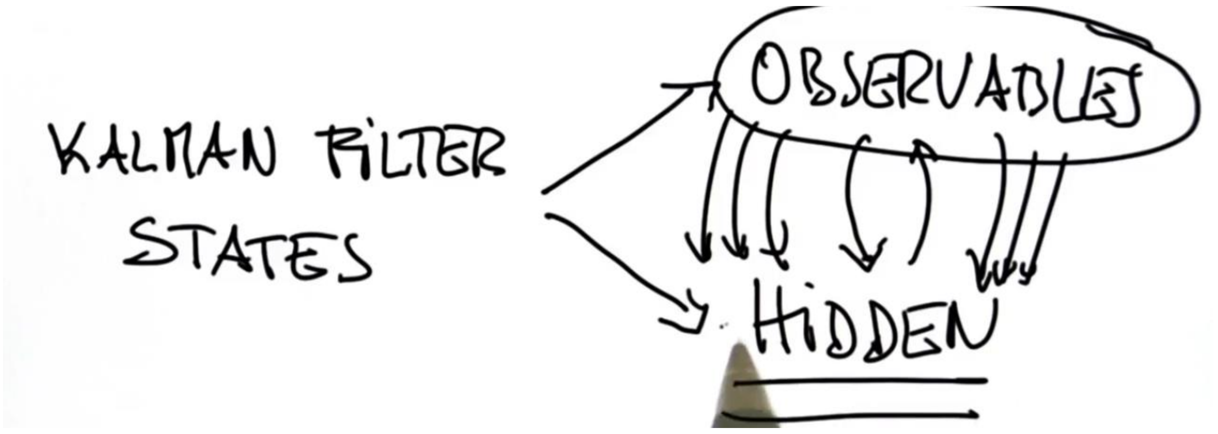
By multiplying the measurement and prior Gaussians, you get a Gaussian (black line) that sits on the middle and has a good estimate what your velocity is and where your location is.



Big Lesson:

The Variables of a Kalman Filter

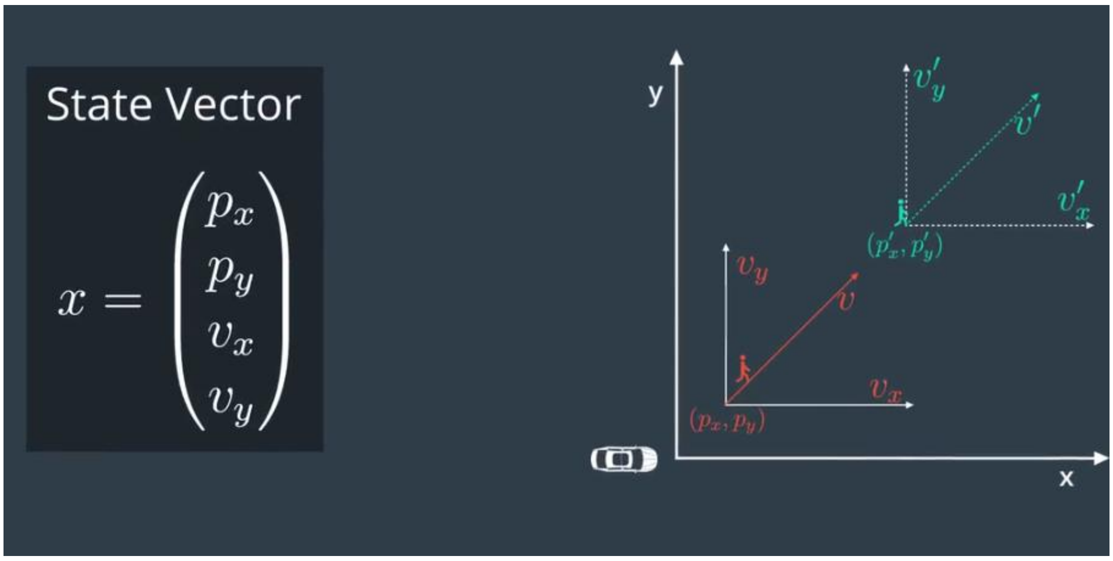
- Often called State because they reflect the state of the physical world like position and velocity.
- They separate into two subsets:
    - Observable (like the position)
    - Hidden which can never directly observed (in our example velocity) and because these two thing interact (observable variables give us information about hidden information) we can estimate or inference what these hidden variables are.



Extended Kalman Filter:

It is extended in the sense that it will be capable of handling more complex motion model and measurement models and consists of an endless loop of prediction and update state.

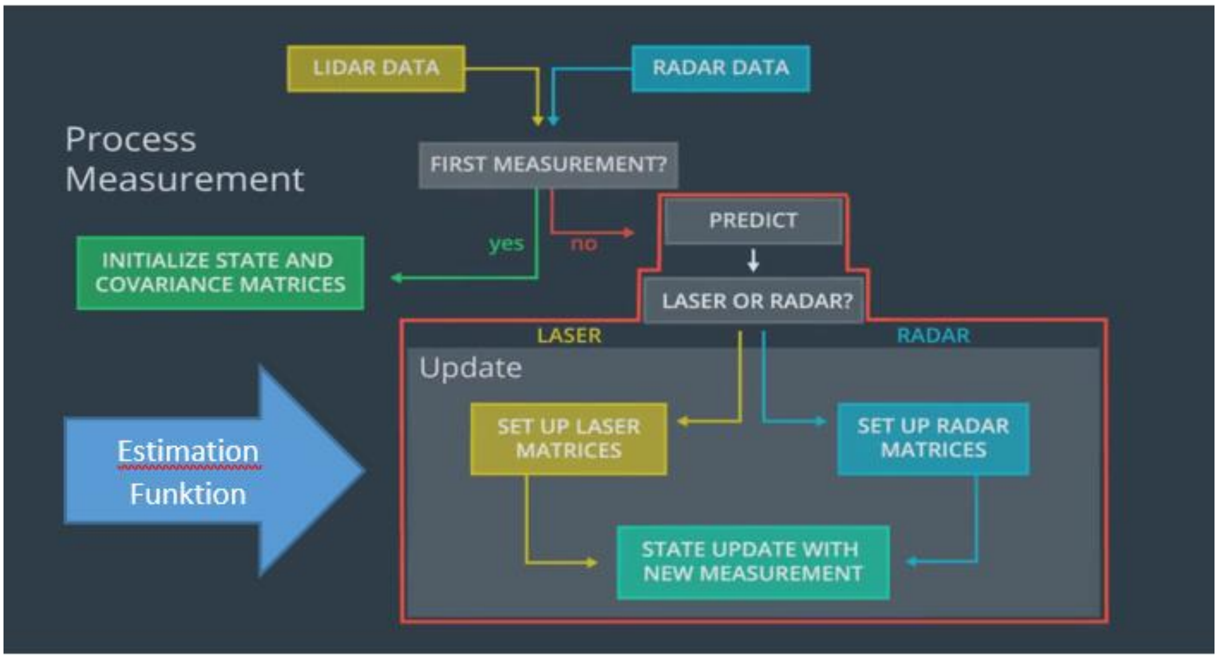
Imagine you are in a car equipped with sensors on the outside. The car sensors can detect objects moving around: for example, the sensors might detect a pedestrian. Let's step through the Kalman Filter algorithm using the pedestrian example. We have two sensors(a Lidar and a Radar) and the information provided be these two sensors is used to estimate the state of a moving pedestrian and this state is presented by a 4D state vector(a x,y position and a x,y velocity).



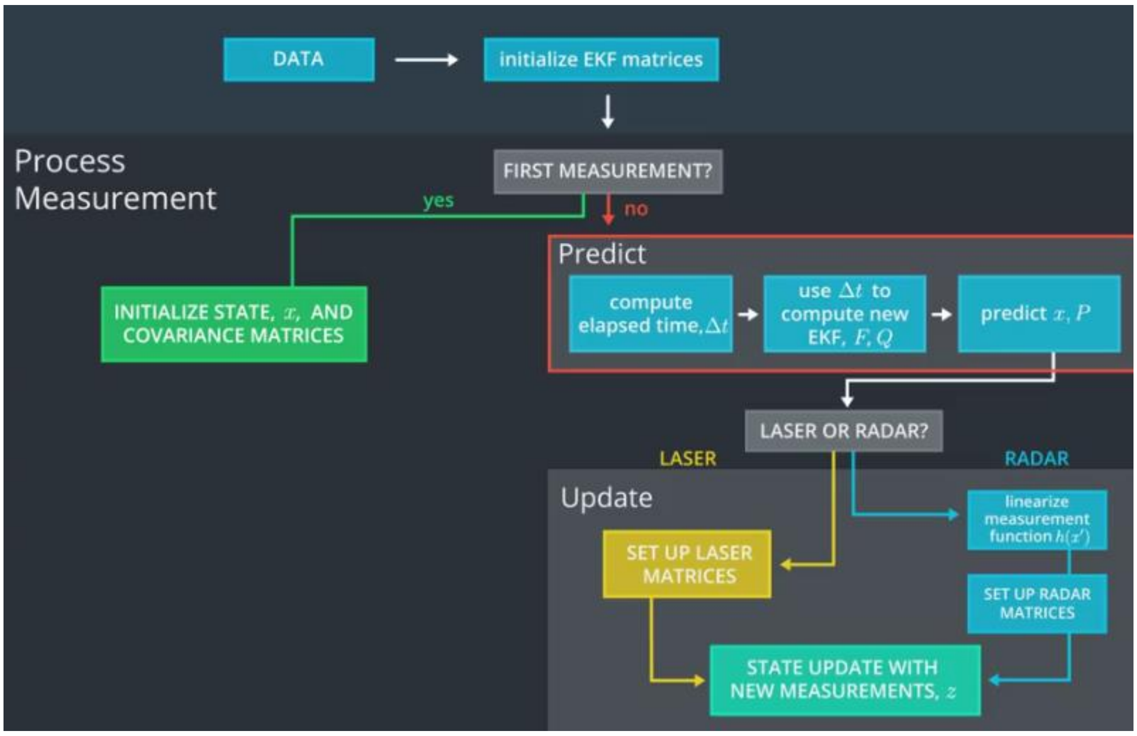
- x is the mean state vector. For an extended Kalman filter, the mean state vector contains information about the object's position and velocity that you are tracking. It is called the "mean" state vector because position and velocity are represented by a Gaussian distribution with mean x.

- P is the state covariance matrix, which contains information about the uncertainty of the object's position and velocity.

Based on the below diagram, the filter (**first measurement**) will receive initial measurements of the pedestrian’s position relative to the car. These measurements will come from a radar or lidar sensor and the filter will initialize the pedestrian’s position based on the first measurement (**initialize state and covariance matrices**). At this point, the car will receive another sensor measurement after a time period Δt and then we perform two steps, state prediction and measurement update (Each time we receive a new measurement from a given sensor, the estimation function is trigged).



In the prediction state (for Radar and Laser Measurements):


DATA
initialize EKF matrices
Process Measurement
FIRST MEASUREMENT?
yes
no
INITIALIZE STATE, x, AND COVARIANCE MATRICES
Predict
compute elapsed time, Δt
use Δt to compute new EKF, F, Q
predict x, P
LASER OR RADAR?
LASER
RADAR
Update
linearize measurement function $h(x')$
SET UP LASER MATRICES
SET UP RADAR MATRICES
STATE UPDATE WITH NEW MEASUREMENTS, z

- We predict the pedestrian’s state by taking to account the elapsed time between the current and the pervious observations, because in the reality the time elapsed between two consecutive observations might vary and is not constant.
    - We can use the timestamp values to compute the elapsed time between two consecutive observations and additionally we divide the result by 10^6 to transform it from microseconds to seconds.

```
float dt = (measurement_pack.timestamp_ - previous_timestamp_) / 1000000.0;
```

- Linear Motion Model, we can predict the next state with help of the old positions plus displacement times Δt and plus noise, which has the mean zero.

$$\begin{cases} p'_x = p_x + v_x \Delta t + \boxed{\nu_{px}} \\ p'_y = p_y + v_y \Delta t + \boxed{\nu_{py}} \quad \text{zero mean} \\ v'_x = v_x + \boxed{\nu_{vx}} \\ v'_y = v_y + \boxed{\nu_{vy}} \quad \text{zero mean} \end{cases}$$

- F: is a transition matrix that, when multiplied with x, predicts where the object will be after time Δt.

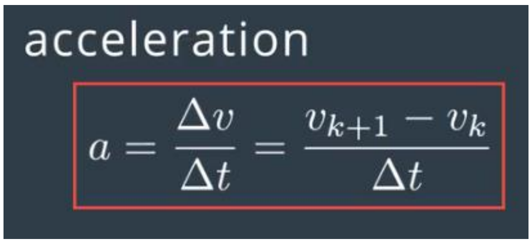
$$F = \begin{pmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- ν: We assume the object travels at a constant velocity, but in reality, the object might accelerate or decelerate. The notation ν~N(0,Q) defines the process noise as a Gaussian distribution with mean zero and covariance Q. (The mean = 0 is saying that the mean noise is zero and the uncertainty shows up in the Q matrix as acceleration noise).

- Q: 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, which explained in the section (ν). The process noise depends on two things: the elapsed time and the uncertainty of acceleration. We can model the process noise by considering both of these factors.

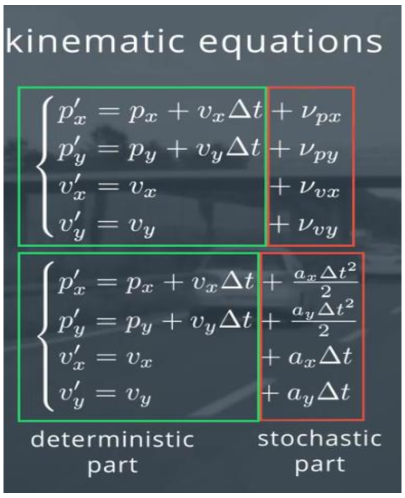
  First I am going to show how the acceleration is expressed by the kinematic equation then I use that information to drive the process covariance Q (ν~N(0,Q)).

Say we have two consecutive observation of the same pedestrian with initial and final velocities then we can drive from the kinematic formula the change in the velocity, in other word including acceleration:


acceleration

$$\boxed{a = \frac{\Delta v}{\Delta t} = \frac{v_{k+1} - v_k}{\Delta t}}$$

Since the acceleration is unknown, we can add it to the last component (stochastic part) and acceleration is a random vector with zero mean and standard deviation sigma ax and sigma ay.

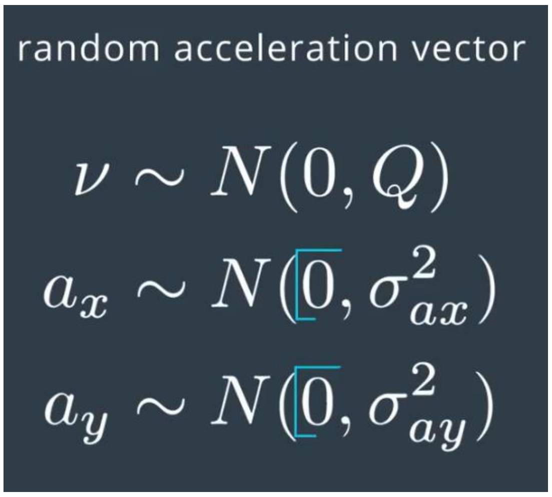

kinematic equations

$$\begin{cases} p'_x = p_x + v_x \Delta t + \nu_{px} \\ p'_y = p_y + v_y \Delta t + \nu_{py} \\ v'_x = v_x + \nu_{vx} \\ v'_y = v_y + \nu_{vy} \end{cases}$$

$$\begin{cases} p'_x = p_x + v_x \Delta t + \frac{a_x \Delta t^2}{2} \\ p'_y = p_y + v_y \Delta t + \frac{a_y \Delta t^2}{2} \\ v'_x = v_x + a_x \Delta t \\ v'_y = v_y + a_y \Delta t \end{cases}$$

deterministic part

stochastic part


random acceleration vector

$$\nu \sim N(0, Q)$$

$$a_x \sim N(0, \sigma_{ax}^2)$$

$$a_y \sim N(0, \sigma_{ay}^2)$$

The below presented vector can be decomposed into two components:

- A four by two matrix G which doesn't contain random components
- A two by one matrix contains random acceleration components.

$$\nu = \begin{pmatrix} \frac{a_x \Delta t^2}{2} \\ \frac{a_y \Delta t^2}{2} \\ a_x \Delta t \\ a_y \Delta t \end{pmatrix} = \begin{pmatrix} \frac{\Delta t^2}{2} & 0 \\ 0 & \frac{\Delta t^2}{2} \\ \Delta t & 0 \\ 0 & \Delta t \end{pmatrix} \begin{pmatrix} a_x \\ a_y \end{pmatrix} = Ga$$

Based on our noise vector we can define now the new covariance matrix Q. The covariance matrix is defined as the expectation value of the noise vector ν times the noise vector ν^T:

$$Q = E[\nu\nu^T] = E[Gaa^T G^T]$$

As G does not contain random variables, we can put it outside the expectation calculation.

$$Q = GE[aa^T]G^T = G \begin{pmatrix} \sigma_{ax}^2 & \sigma_{axy} \\ \sigma_{axy} & \sigma_{ay}^2 \end{pmatrix} G^T = GQ_\nu G^T$$

This leaves us with three statistical moments:

- the expectation of ax times ax, which is the variance of ax squared: σ_{ax}^2.
- the expectation of ay times ay, which is the variance of ay squared: σ_{ay}^2.
- and the expectation of ax times ay, which is the covariance of ax and ay: σ_{axy}.

a_x and a_y are assumed uncorrelated noise processes. This means that the covariance σ_{axy} in Q_ν is zero:

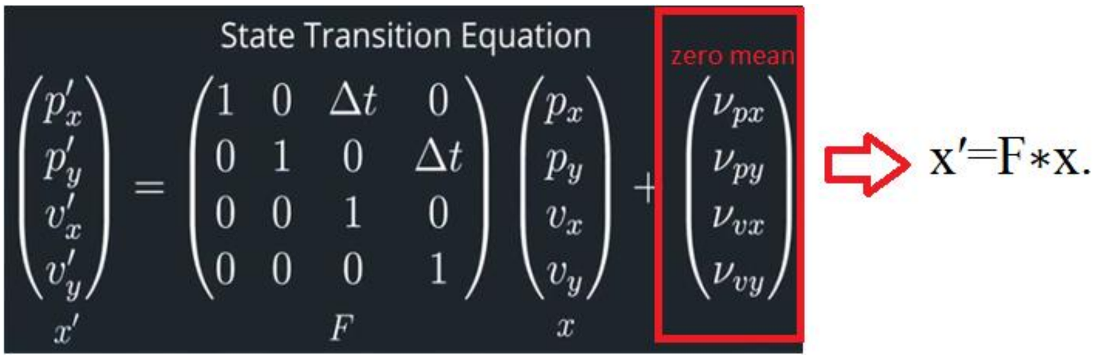
$$Q_\nu = \begin{pmatrix} \sigma_{ax}^2 & \sigma_{axy} \\ \sigma_{axy} & \sigma_{ay}^2 \end{pmatrix} = \begin{pmatrix} \sigma_{ax}^2 & 0 \\ 0 & \sigma_{ay}^2 \end{pmatrix}$$

After combining everything in one matrix, we obtain our 4 by 4 Q matrix:

$$Q = GQ_\nu G^T = \begin{pmatrix} \frac{\Delta t^4}{4}\sigma_{ax}^2 & 0 & \frac{\Delta t^3}{2}\sigma_{ax}^2 & 0 \\ 0 & \frac{\Delta t^4}{4}\sigma_{ay}^2 & 0 & \frac{\Delta t^3}{2}\sigma_{ay}^2 \\ \frac{\Delta t^3}{2}\sigma_{ax}^2 & 0 & \Delta t^2\sigma_{ax}^2 & 0 \\ 0 & \frac{\Delta t^3}{2}\sigma_{ay}^2 & 0 & \Delta t^2\sigma_{ay}^2 \end{pmatrix}$$

- B: is a matrix called the control input matrix and u is the control vector. 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 and represent acceleration as a random noise with mean ν.

- x′: The x′=Fx+Bu+ν equation does these prediction calculations for us but then Bu was crossed out leaving x′=Fx+ν. The noise mean = 0 is saying that the mean noise is zero. The equation then becomes x′=F∗x.



$$\begin{pmatrix} p'_x \\ p'_y \\ v'_x \\ v'_y \end{pmatrix} = \begin{pmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ v_x \\ v_y \end{pmatrix} + \begin{pmatrix} \nu_{px} \\ \nu_{py} \\ \nu_{vx} \\ \nu_{vy} \end{pmatrix}$$

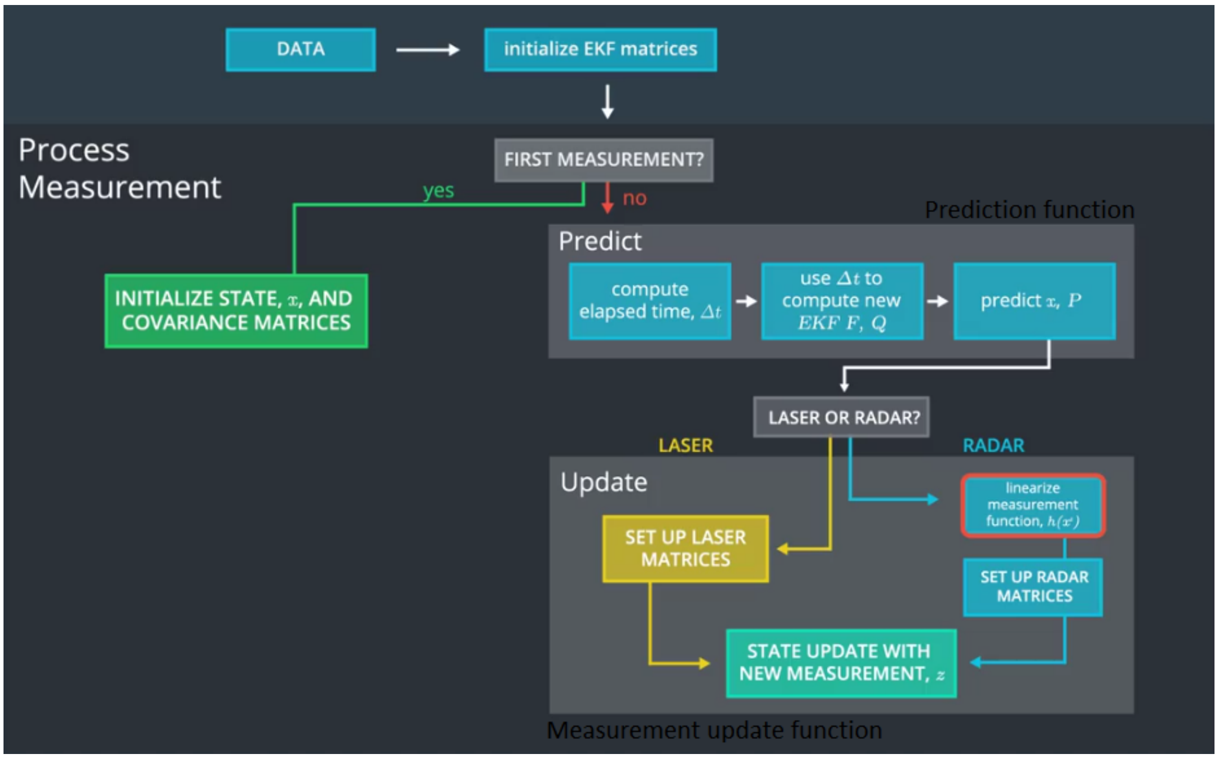
$$x' \qquad F \qquad x$$

$\Rightarrow$ x′=F∗x.

- $P' = FPF^T + Q$ represents this increase in uncertainty.

In The measurement update

Uses new observations to correct our belief about the state of the predictions and depends on sensor type (Lidar and Radar). If a laser sensor generates the current measurement, we just apply a standard kalman filter to update the pedestrian state and why radar measurements involves a non-linear measurement function when we receive a radar measurement, we use the extended kalman filter to measurement update.


DATA
initialize EKF matrices
Process Measurement
FIRST MEASUREMENT?
yes
no
INITIALIZE STATE, x, AND COVARIANCE MATRICES
Prediction function
Predict
compute elapsed time, Δt
use Δt to compute new EKF F, Q
predict x, P
LASER OR RADAR?
LASER
RADAR
Update
linearize measurement function, $h(x')$
SET UP LASER MATRICES
SET UP RADAR MATRICES
STATE UPDATE WITH NEW MEASUREMENT, z
Measurement update function

Notice: the state of the pedestrian's position and velocity is updated asynchronously each time the measurement received regardless of the source sensor.

- **Update Step for Laser Measurement:**
    - H (state transition matrix) 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 (px, py) 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.

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

    - w: represents sensor measurement noise. Measurement noise refers to uncertainty in sensor measurements. The notation 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.

    - z=H∗x+w for the update step (Measurement Function which is a linear function). 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. Then the lidar measurement position and our belief about the object's position can be compared.

$$z = \begin{pmatrix} p_x \\ p_y \end{pmatrix}$$

    - R, which represents the uncertainty in our sensor measurements. The dimensions of the R matrix is square and each side of its matrix is the same length as the number of measurements parameters z. The matrix R represents the uncertainty in the position measurements we receive from the laser sensor and generally, the parameters for the random noise measurement matrix will be provided by the sensor manufacturer.

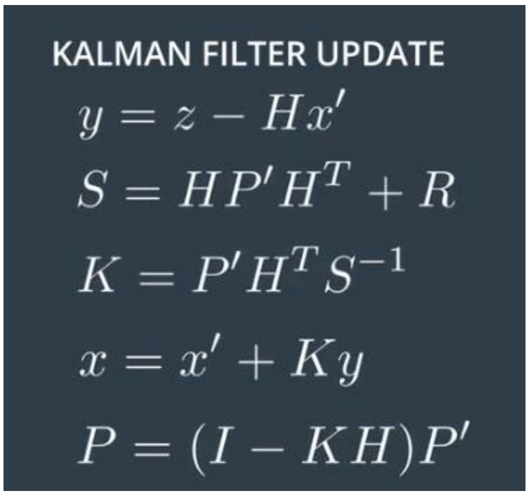
$$R = E[\omega\omega^T] = \begin{pmatrix} \sigma_{px}^2 & 0 \\ 0 & \sigma_{py}^2 \end{pmatrix}$$

- y=z−Hx′: Now we get some sensor information (z) 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′.

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

- I is identity matrix.

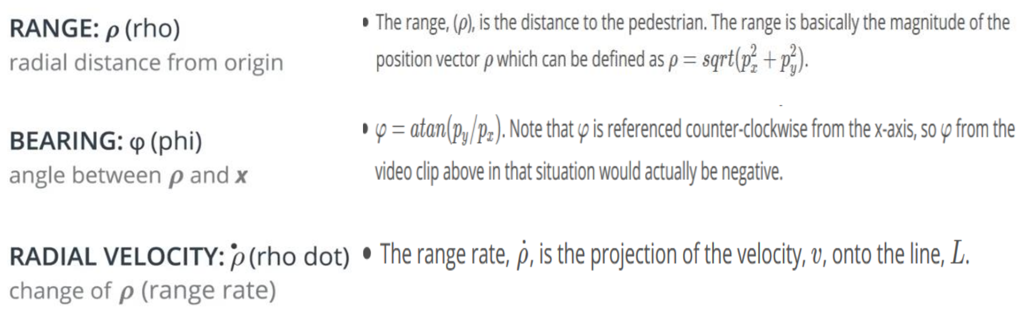
- The Kalman Filter update Formula for Laser Measurements:



KALMAN FILTER UPDATE

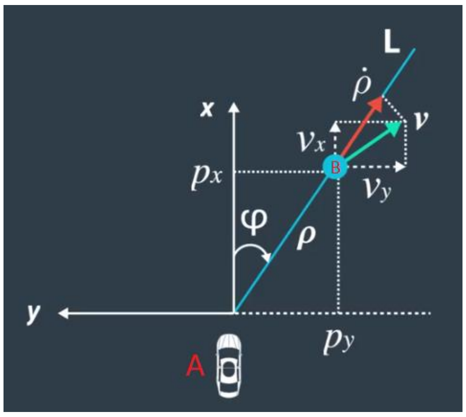
$$y = z - Hx'$$
$$S = HP'H^T + R$$
$$K = P'H^TS^{-1}$$
$$x = x' + Ky$$
$$P = (I - KH)P'$$

- **Update Step for Radar Measurements:**
Radar sees the world differently, as you can see in the photo our vehicle is at A coordinate and the pedestrian at B coordinate. The x-axis is always points in the vehicle’s direction of movement and the y-axis is always points to the left. Instead of 2D, the Radar can directly measure:



RANGE: ρ (rho) radial distance from origin	• The range, (ρ), is the distance to the pedestrian. The range is basically the magnitude of the position vector ρ which can be defined as $\rho = sqrt(p_x^2 + p_y^2)$.
BEARING: φ (phi) angle between ρ and x	• $\varphi = atan(p_y/p_x)$. Note that φ is referenced counter-clockwise from the x-axis, so φ from the video clip above in that situation would actually be negative.
RADIAL VELOCITY: $\dot{\rho}$ (rho dot) change of ρ (range rate)	• The range rate, $\dot{\rho}$, is the projection of the velocity, v, onto the line, L.

Using Doppler effects the Radar can directly measure the radial velocity of a moving object and the radial velocity is the component (ρ˙) of the velocity moving towards or away from the sensor.



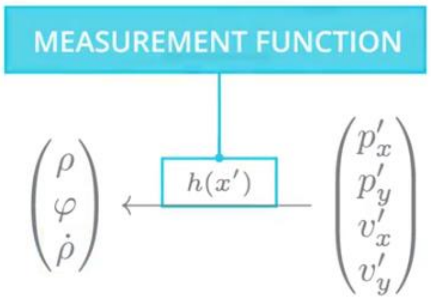
- z: Measurement vector is 3-measurement vector component:

$$z = \begin{pmatrix} \rho \\ \varphi \\ \dot{\rho} \end{pmatrix}$$

- R: Radar measurement Covariance becomes a 3by 3 diagonal matrix (considering that the tree measurement vector component are not cross-correlated).

$$R = \begin{pmatrix} \sigma_\rho^2 & 0 & 0 \\ 0 & \sigma_\varphi^2 & 0 \\ 0 & 0 & \sigma_{\dot{\rho}}^2 \end{pmatrix}$$

- y=z−h(x′): Our state (x′) from the prediction step still has four parameter as the same and as before said the measurement vector has 3 parameters, in order to calculate y=z−h(x′) we need a measurement function that maps the predicted state(x′) into measurement space:


MEASUREMENT FUNCTION
$\begin{pmatrix} \rho \\ \varphi \\ \dot{\rho} \end{pmatrix}$
$h(x')$
$\begin{pmatrix} p'_x \\ p'_y \\ v'_x \\ v'_y \end{pmatrix}$

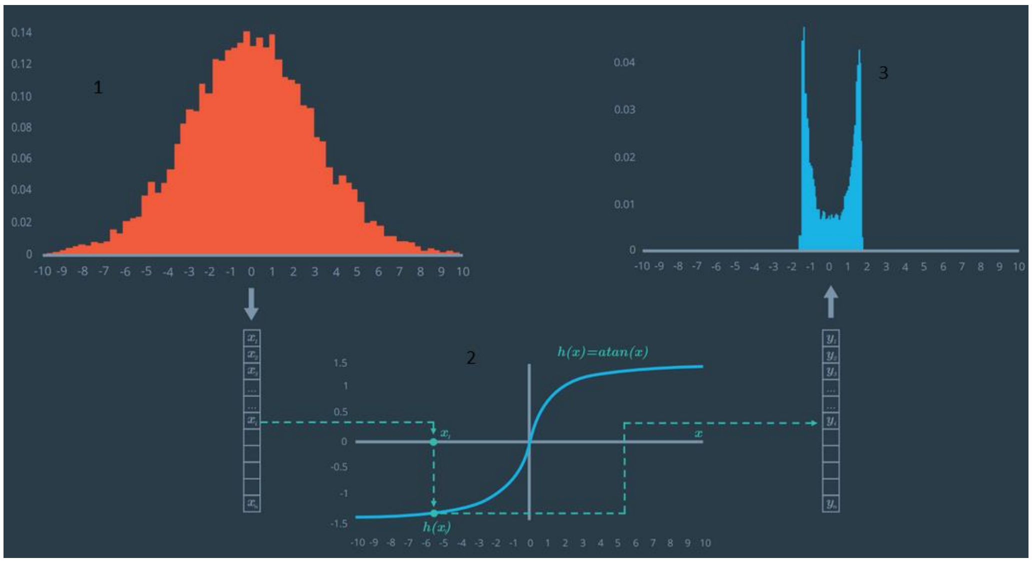
The h is nonlinear function that specifies how the predicted position and speed get mapped to the polar coordinates of range, bearing and range rate and h function is presented below:

$$h(x') = \begin{pmatrix} \rho \\ \phi \\ \dot{\rho} \end{pmatrix} = \begin{pmatrix} \sqrt{p'^2_x + p'^2_y} \\ \arctan(p'_y / p'_x) \\ \frac{p'_x v'_x + p'_y v'_y}{\sqrt{p'^2_x + p'^2_y}} \end{pmatrix}$$

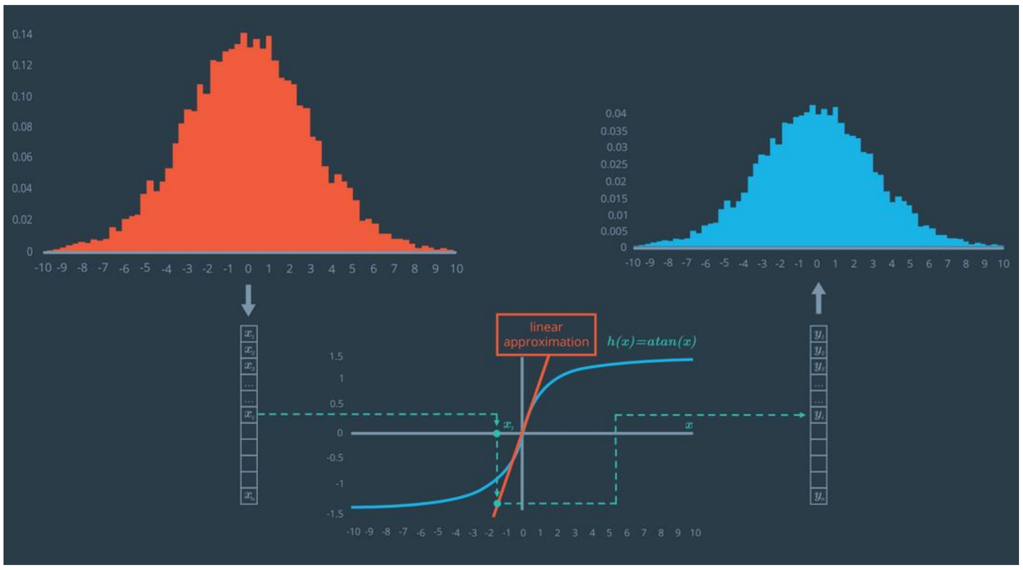
Notice: One other important point when calculating y with radar sensor data: the second value in the polar coordinate vector is the angle ϕ. You will need to make sure to normalize ϕ in the y vector so that its angle is between –π and π.

Hj : After applying a nonlinear measurement function, we have 3-measurement vector component (z) which cannot be used for the Kalman Filter equations to update the predicted state (with new measurements) because we are not working with the Gaussian distribution after applying a nonlinear measurement.

To understand the problem I used a Gaussian (1) from 10,000 random values in a normal distribution with a mean of 0 then applied (2) a nonlinear function (arctan) to transform each value as you can see in 3 the resulting distribution is not any more a Gaussian distribution.



To solve this problem, we have to approximate (linearize) our measurement function to a linear function, which is the key idea of the extended Kalman filter. I repeated the test by using a linear approximation. Notice how the blue graph, the output, remains a Gaussian after applying a first order Taylor expansion.



The extended kalman filter uses a method called First Order Tylor Expansion

$$f(x) \approx f(\mu) + \frac{\partial f(\mu)}{\partial x}(x - \mu)$$

What we do is we first evaluate the nonlinear function h at the mean location (μ), which is the best estimate of our predicted distribution then we extrapolate a line with slope around μ and this slope is given by the first derivative of h.

$$h(x) \approx h(\mu) + \frac{\partial h(\mu)}{\partial x}(x - \mu)$$

Now that you have seen how to do a Taylor series expansion with a one-dimensional equation, we will need to look at the Taylor series expansion for multi-dimensional equations. We will need to use a multi-dimensional Taylor series expansion to make a linear approximation of the h function. Here is a general formula for the multi-dimensional Taylor series expansion:

$$T(x) = f(a) + (x-a)^T Df(a) + \frac{1}{2!}(x-a)^T D^2 f(a)(x-a) + \ldots$$

Where Df(a) is called the Jacobian matrix and D^2f(a) is called the Hessian matrix. They represent first order and second order derivatives of multi-dimensional equations. A full Taylor series expansion would include higher order terms as well for the third order derivatives, fourth order derivatives, and so on. Notice the similarities between the multi-dimensional Taylor series expansion and the one-dimensional Taylor series expansion:

$$T(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \ldots$$

To derive a linear approximation for the h function, we will only keep the expansion up to the Jacobian matrix Df(a). We will ignore the Hessian matrix $D^2f(a)$ and other higher order terms. Assuming (x−a) is small, $(x-a)^2$ or the multi-dimensional equivalent $(x-a)^T$ (x−a) will be even smaller; the extended Kalman filter we'll be using assumes that higher order terms beyond the Jacobian are negligible.
The derivative of h(x) with the respect to x is called Jacobian matrix and is going to be a matrix containing all the partial derivatives, we know, the measurement function describes tree component (Range, Bearing, Range Rate) and my state is a vector with four components (px,py,vx,vy), in that case the Jacobian matric is going to be a matrix with 3 rows and 4 columns.

Jacobian Matrix

$$H_j = \begin{bmatrix} \frac{\partial \rho}{\partial p_x} & \frac{\partial \rho}{\partial p_y} & \frac{\partial \rho}{\partial v_x} & \frac{\partial \rho}{\partial v_y} \\ \frac{\partial \varphi}{\partial p_x} & \frac{\partial \varphi}{\partial p_y} & \frac{\partial \varphi}{\partial v_x} & \frac{\partial \varphi}{\partial v_y} \\ \frac{\partial \dot{\rho}}{\partial p_x} & \frac{\partial \dot{\rho}}{\partial p_y} & \frac{\partial \dot{\rho}}{\partial v_x} & \frac{\partial \dot{\rho}}{\partial v_y} \end{bmatrix}$$

$$z = \begin{pmatrix} \rho \\ \varphi \\ \dot{\rho} \end{pmatrix} \begin{matrix} \leftarrow \text{Range} \\ \leftarrow \text{Bearing} \\ \leftarrow \text{Range rate} \end{matrix}$$

$$x = \begin{pmatrix} p_x \\ p_y \\ v_x \\ v_y \end{pmatrix} \begin{matrix} > \text{Position} \\ > \text{Velocity} \end{matrix}$$

After calculating all the partial derivatives, our resulted Jacobian, Hj is:

$$H_j = \begin{bmatrix} \frac{p_x}{\sqrt{p_x^2+p_y^2}} & \frac{p_y}{\sqrt{p_x^2+p_y^2}} & 0 & 0 \\ -\frac{p_y}{p_x^2+p_y^2} & \frac{p_x}{p_x^2+p_y^2} & 0 & 0 \\ \frac{p_y(v_xp_y-v_yp_x)}{(p_x^2+p_y^2)^{3/2}} & \frac{p_x(v_yp_x-v_xp_y)}{(p_x^2+p_y^2)^{3/2}} & \frac{p_x}{\sqrt{p_x^2+p_y^2}} & \frac{p_y}{\sqrt{p_x^2+p_y^2}} \end{bmatrix}$$

Notice: because the linearization points change, we have to recompute the Jacobian matrix at every point in the time.

- For radar measurement update, Hj is used to calculate S, K and P.

Measurement update

$$y = z - h(x')$$

$$S = H_j P' H_j^T + R$$

$$K = P' H_j^T S^{-1}$$

$$x = x' + Ky$$

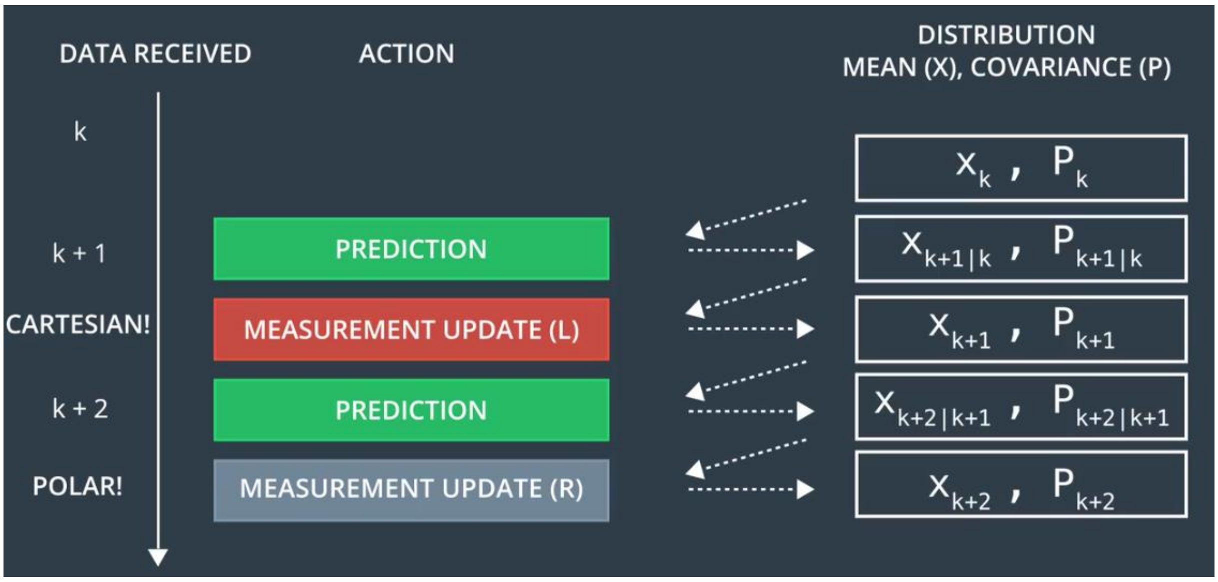
$$P = (I - K H_j) P'$$

Workflow of the extended kalman filter works:

Here the pedestrian's state at time K is a distribution with mean (X) and covariance (P) and at time k+1 we just received the laser measurement (Cartesian coordinate System) and the first thing to do is to make a prediction about where we think the pedestrian would be at time k+1.

The second thing is to do called measurement update where we combine the pedestrian predicted state with the new laser measurement and what we now have is a more accurate beliefs about the pedestrian's position at time k+1, this is what we called posterior.

Now we received the radar measurement (Polar coordinate system) at time k+2 and we make again a prediction and then combine the prediction with the radar measurement (the prediction for the radar is the same as in the laser case but what change is the measurement update step because the radar sees the world differently).



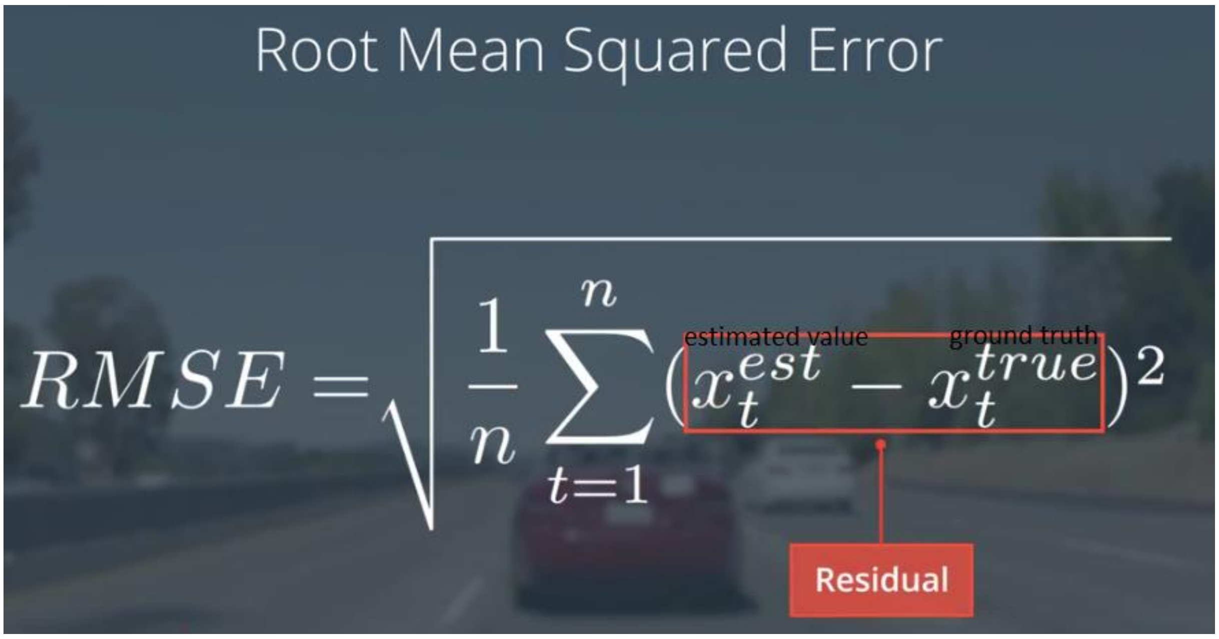
Notice: Because of two different worlds (Cartesian and Polar) we have to use two different measurement steps. If both the radar and laser measurements arrive at the same time, it uses one of the sensors then another sensor to prediction and measurement step.

Evaluating Kalman Filter Performanc:

Once we have implemented our tracking Kalman filter algorithm, we might want to check its performance in terms how far the estimated result is from the true result, there are many evaluation metrics, but the most common one is what called Root Mean Squared Error.
In the context of the tracking it is an accuracy metric used to measure deviation of the estimated state of the true state.
At a given processing step, we need two values:

- The estimated values which is a vector with position and velocity components
- The real values that are usually known as ground truth.

The difference between ground truths and estimated values called residual, this residual are squared and averaged, and finally the square root gives us the error metric. The lower is the error the higher is the estimation accuracy.

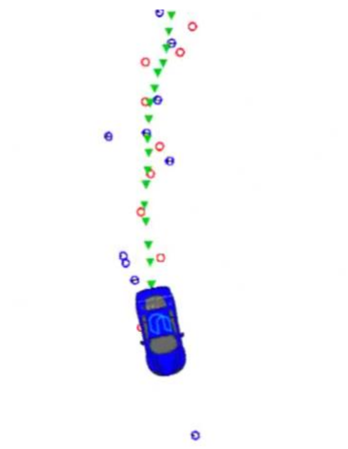


Root Mean Squared Error

$$RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(\boxed{x_t^{est} - x_t^{true}})^2}$$

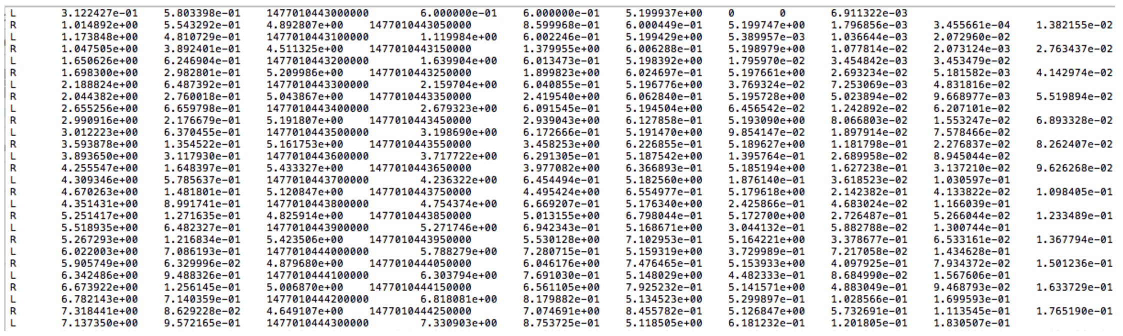
Implementation of the Extended Kalman Filter for Sensor Fusion:

Now that we have learned how the extended Kalman filter works, in this section we are going to implement the extended Kalman filter in C++. We are providing simulated lidar and radar measurements detecting a bicycle that travels around your vehicle. You will use a Kalman filter, lidar measurements and radar measurements to track the bicycle's position and velocity like below:



- ### Explanation of the Data File

Lidar measurements are red circles, radar measurements are blue circles with an arrow pointing in the direction of the observed angle, and estimation markers are green triangles. The simulator will be using this data file, and feed main.cpp values from it one line at a time.



```
L   3.122427e-01   5.803398e-01   1477010443000000   6.000000e-01   6.000000e-01   5.199937e+00   0              0              6.911322e-03
R   1.014892e+00   5.543292e-01   4.892807e+00   1477010443050000   8.599968e-01   6.000449e-01   5.199747e+00   1.796856e-03   3.455661e-04   1.382155e-02
L   1.173848e+00   4.810729e-01   1477010443100000   1.119984e+00   6.002246e-01   5.199429e+00   5.389957e-03   1.036644e-03   2.072960e-02
R   1.047505e+00   3.892401e-01   4.511325e+00   1477010443150000   1.379955e+00   6.006288e-01   5.198979e+00   1.077814e-02   2.073124e-03   2.763437e-02
L   1.650626e+00   6.246904e-01   1477010443200000   1.639904e+00   6.013473e-01   5.198392e+00   1.795970e-02   3.454842e-03   3.453479e-02
R   1.698300e+00   2.982801e-01   5.209986e+00   1477010443250000   1.899823e+00   6.024697e-01   5.197661e+00   2.693234e-02   5.181582e-03   4.142974e-02
L   2.188824e+00   6.487392e-01   1477010443300000   2.159704e+00   6.040855e-01   5.196776e+00   3.769324e-02   7.253069e-03   4.831816e-02
R   2.044382e+00   2.760018e-01   5.043867e+00   1477010443350000   2.419540e+00   6.062840e-01   5.195728e+00   5.023894e-02   9.668977e-03   5.519894e-02
L   2.655256e+00   6.659798e-01   1477010443400000   2.679323e+00   6.091545e-01   5.194504e+00   6.456542e-02   1.242892e-02   6.207101e-02
R   2.990916e+00   2.176679e-01   5.191807e+00   1477010443450000   2.939043e+00   6.127858e-01   5.193090e+00   8.066803e-02   1.553247e-02   6.893328e-02
L   3.012223e+00   6.370455e-01   1477010443500000   3.198690e+00   6.172666e-01   5.191470e+00   9.854147e-02   1.897914e-02   7.578466e-02
R   3.593878e+00   1.354522e-01   5.161753e+00   1477010443550000   3.458253e+00   6.226855e-01   5.189627e+00   1.181798e-01   2.276837e-02   8.262407e-02
L   3.893650e+00   3.117930e-01   1477010443600000   3.717722e+00   6.291305e-01   5.187542e+00   1.395764e-01   2.689958e-02   8.945044e-02
R   4.255547e+00   1.648397e-01   5.433327e+00   1477010443650000   3.977082e+00   6.366893e-01   5.185194e+00   1.627238e-01   3.137210e-02   9.626268e-02
L   4.309346e+00   5.785637e-01   1477010443700000   4.236322e+00   6.454494e-01   5.182560e+00   1.876140e-01   3.618523e-02   1.030597e-01
R   4.670263e+00   1.481801e-01   5.120847e+00   1477010443750000   4.495424e+00   6.554977e-01   5.179618e+00   2.142382e-01   4.133822e-02   1.098405e-01
L   4.351431e+00   8.991741e-01   1477010443800000   4.754374e+00   6.669207e-01   5.176340e+00   2.425866e-01   4.683024e-02   1.166039e-01
R   5.251417e+00   1.271635e-01   4.825914e+00   1477010443850000   5.013155e+00   6.798044e-01   5.172700e+00   2.726487e-01   5.266044e-02   1.233489e-01
L   5.518935e+00   6.482327e-01   1477010443900000   5.271746e+00   6.942343e-01   5.168671e+00   3.044132e-01   5.882788e-02   1.300744e-01
R   5.267293e+00   1.216834e-01   5.423506e+00   1477010443950000   5.530128e+00   7.102953e-01   5.164221e+00   3.378677e-01   6.533161e-02   1.367794e-01
L   6.022003e+00   7.086193e-01   1477010444000000   5.788279e+00   7.280715e-01   5.159319e+00   3.729989e-01   7.217058e-02   1.434628e-01
R   5.905749e+00   6.329996e-02   4.879680e+00   1477010444050000   6.046176e+00   7.476465e-01   5.153933e+00   4.097925e-01   7.934372e-02   1.501236e-01
L   6.342486e+00   9.488326e-01   1477010444100000   6.303794e+00   7.691030e-01   5.148029e+00   4.482333e-01   8.684990e-02   1.567606e-01
R   6.673922e+00   1.256145e-01   5.006870e+00   1477010444150000   6.561105e+00   7.925232e-01   5.141571e+00   4.883049e-01   9.468793e-02   1.633729e-01
L   6.782143e+00   7.140359e-01   1477010444200000   6.818081e+00   8.179882e-01   5.134523e+00   5.299897e-01   1.028566e-01   1.699593e-01
R   7.318441e+00   8.629228e-02   4.649107e+00   1477010444250000   7.074691e+00   8.455782e-01   5.126847e+00   5.732691e-01   1.113545e-01   1.765190e-01
L   7.137350e+00   9.572165e-01   1477010444300000   7.330903e+00   8.753725e-01   5.118505e+00   6.181232e-01   1.201805e-01   1.830507e-01
```

- Each row represents a sensor measurement where the first column tells you if the measurement comes from radar (R) or lidar (L).

- For a row containing radar data, the columns are: sensor_type, rho_measured, phi_measured, rhodot_measured, timestamp, x_groundtruth, y_groundtruth, vx_groundtruth, vy_groundtruth, yaw_groundtruth, yawrate_groundtruth.

- For a row containing lidar data, the columns are: sensor_type, x_measured, y_measured, timestamp, x_groundtruth, y_groundtruth, vx_groundtruth, vy_groundtruth, yaw_groundtruth, yawrate_groundtruth.

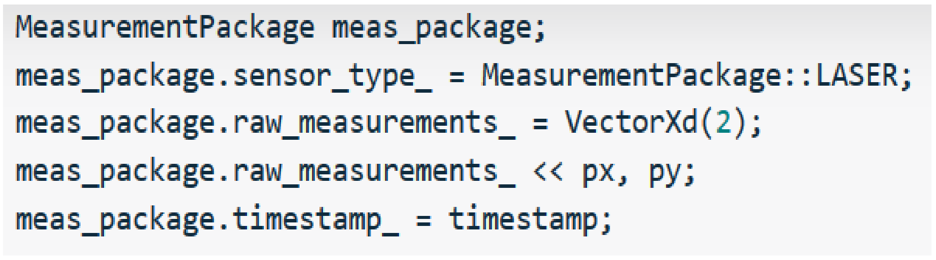
- Whereas radar has three measurements (rho, phi, rhodot), lidar has two measurements (x, y).

- You will use the measurement values and timestamp in your Kalman filter algorithm. Ground truth, which represents the actual path the bicycle took, is for calculating **root mean squared error.**

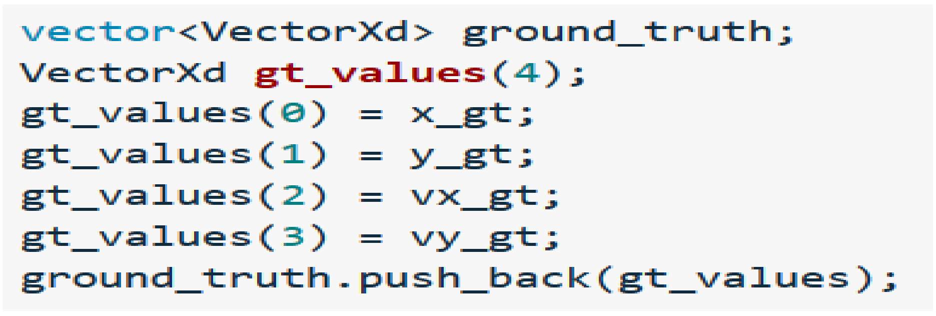
- **Reading in the Data**

  - It provided code that will read in and parse the data files for you. This code is in the main.cpp file. The main.cpp file creates instances of a **MeasurementPackage**.

  - If you look inside 'main.cpp', you will see code like:



```
MeasurementPackage meas_package;
meas_package.sensor_type_ = MeasurementPackage::LASER;
meas_package.raw_measurements_ = VectorXd(2);
meas_package.raw_measurements_ << px, py;
meas_package.timestamp_ = timestamp;
```



```
vector<VectorXd> ground_truth;
VectorXd gt_values(4);
gt_values(0) = x_gt;
gt_values(1) = y_gt;
gt_values(2) = vx_gt;
gt_values(3) = vy_gt;
ground_truth.push_back(gt_values);
```

  - The ground truth [px, py,vx,vy] for each line in the data file gets pushed onto ground_truth so RMSE can be calculated later from tools.cpp.

- **File Structure**

To review what we learned in the extended Kalman filter lectures, let's discuss the three main steps for programming a Kalman filter:

  - **initializing** Kalman filter variables
  - **predicting** where our object is going to be after a time step Δt
  - **updating** where our object is based on sensor measurements

Then the prediction and update steps repeat themselves in a loop. To measure how well our Kalman filter performs, we will then calculate root mean squared error comparing the Kalman filter results with the provided ground truth. These three steps (initialize, predict, update) plus calculating RMSE encapsulate the entire extended Kalman filter project.

Files in the Github src Folder, the files you need to work with are in the src folder of the github repository:

- main.cpp - communicates with the Simulator receiving data measurements, calls a function to run the Kalman filter, calls a function to calculate RMSE
- FusionEKF.cpp - initializes the filter, calls the predict function, calls the update function
- kalman_filter.cpp- defines the predict function, the update function for lidar, and the update function for radar
- tools.cpp- function to calculate RMSE and the Jacobian matrix

- **How the Files Relate to Each Other:**

Here is a brief overview of what happens when you run the code files:

- Main.cpp reads in the data and sends a sensor measurement to FusionEKF.cpp

- FusionEKF.cpp takes the sensor data, initializes variables, and updates variables. The Kalman filter equations are not in this file. FusionEKF.cpp has a variable called ekf_, which is an instance of a KalmanFilter class. The ekf_ will hold the matrix and vector values. You will also use the ekf_ instance to call the predict and update equations.

- The KalmanFilter class is defined in kalman_filter.cpp (contains functions for the prediction and update steps) and kalman_filter.h.

- **Main.cpp**

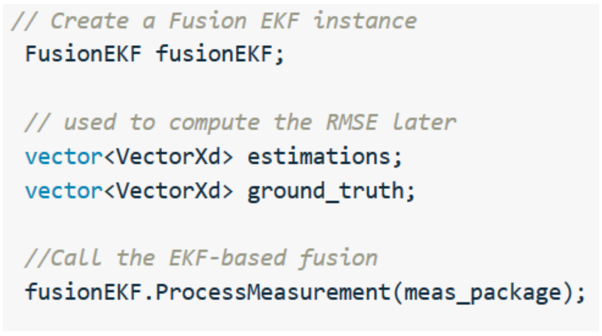
We already discussed how `main.cpp` reads in the sensor data. Recall that `main.cpp` reads in the sensor data line by line from the client and stores the data into a measurement object that it passes to the Kalman filter for processing. Also a ground truth list and an estimation list are used for tracking RMSE.

- main.cpp is made up of several functions within main(), these all handle the uWebsocketIO communication between the simulator and itself and all the main code loops in h.onMessage(), to have access to initial variables that we created at the beginning of main(), we pass pointers as arguments into the header of h.onMessage().

```
h.onMessage([&fusionEKF,&tools,&estimations,&ground_truth]
            (uWS::WebSocket<uWS::SERVER> ws, char *data, size_t length,
             uWS::OpCode opCode)
```

- The rest of the arguments in `h.onMessage` are used to set up the server.
- The below code is:

        - creating an instance of the `FusionEKF` class
        - Receiving the measurement data calling the `ProcessMeasurement()` function. `ProcessMeasurement()` is responsible for the initialization of the Kalman filter as well as calling the prediction and update steps of the Kalman filter. The `ProcessMeasurement()` function is built in `FusionEKF.cpp`



```
// Create a Fusion EKF instance
 FusionEKF fusionEKF;

 // used to compute the RMSE later
 vector<VectorXd> estimations;
 vector<VectorXd> ground_truth;

 //Call the EKF-based fusion
 fusionEKF.ProcessMeasurement(meas_package);
```

    - Finally, The rest of `main.cpp` will output the following results to the simulator:
        - estimation position
        - Calculated RMSE (RMSE function is implemented in the `tools.cpp` file.)

- **FusionEKF.cpp**

  Every time main.cpp calls fusionEKF.ProcessMeasurement(measurement_pack_list[k]),the code in FusionEKF.cpp will run. - If this is the first measurement, the Kalman filter will try to initialize the object's location with the sensor measurement.

    - Variables and matrices (x, F, H_laser, H_jacobian, P, etc.) will be Initialized.
    - The Kalman filter position vector with the first sensor measurements will be initialized.
    - The F and Q matrices prior to the prediction step based on the elapsed time between measurements will be calculated.
    - The update step for either the lidar or the radar sensor measurement will be called.

- **Important Dependencies**

    - cmake >= 3.5
        - All OSes: click here for installation instructions
    - make >= 4.1
        - Linux: make is installed by default on most Linux distros
        - Mac: install Xcode command line tools to get make
        - Windows: Click here for installation instructions
    - gcc/g++ >= 5.4
        - Linux: gcc / g++ is installed by default on most Linux distros

    - Mac: same deal as make - install Xcode command line tools
    - Windows: recommend using MinGW

- **Build Instructions**
    - First, clone this repo.
    - This project can be used with a Simulator, which can be downloaded here
    - In order to use the simulator, we need to install uWebSocketIO so that it can communicate with the C++ program--The simulator is the client, and the C++ program is the web server. To install it download uWebSocketIO then:
        - `chmod +x install-mac.sh`
        - `./install-mac.sh`
    - Once the install for uWebSocketIO is complete, the main program can be built--with the completed code--and run by doing the following from the project top directory.
        - Make the build directory: `mkdir build && cd build`
        - Compile: `cmake .. && make`
        - Run it: `./ExtendedKF path/to/input.txt`
            - e.g: `./ExtendedKF ../data/sample-laser-radar-measurement-data-1.txt`