**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.