Localization

All self-driving cars go through the same series of steps to safely navigate through the world.

You've been working on the first step: **localization**. Before cars can safely navigate, they first use sensors and other collected data to best estimate where they are in the world.

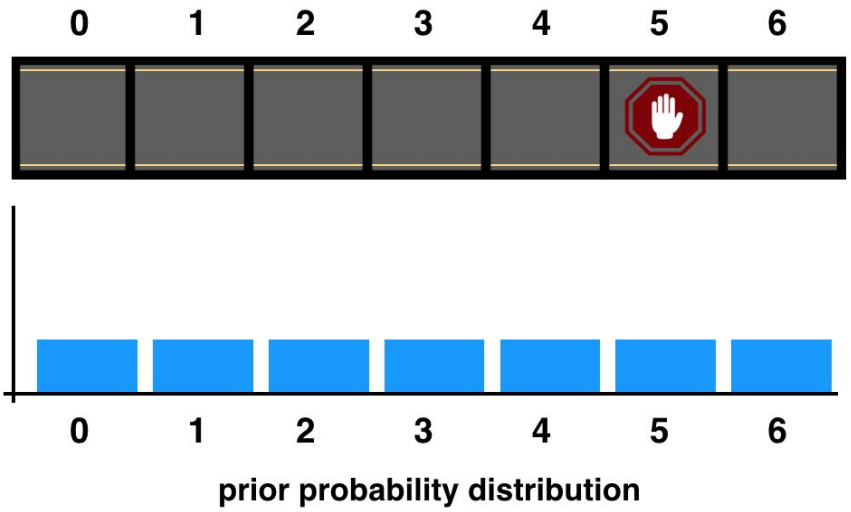
Kalman Filter

Let's review the steps that a Kalman filter takes to localize a car.

1. Initial Prediction

First, we start with an initial prediction of our car's location and a probability distribution that describes our uncertainty about that prediction.

Below is a 1D example, we know that our car is on this one lane road, but we don't know its exact location.

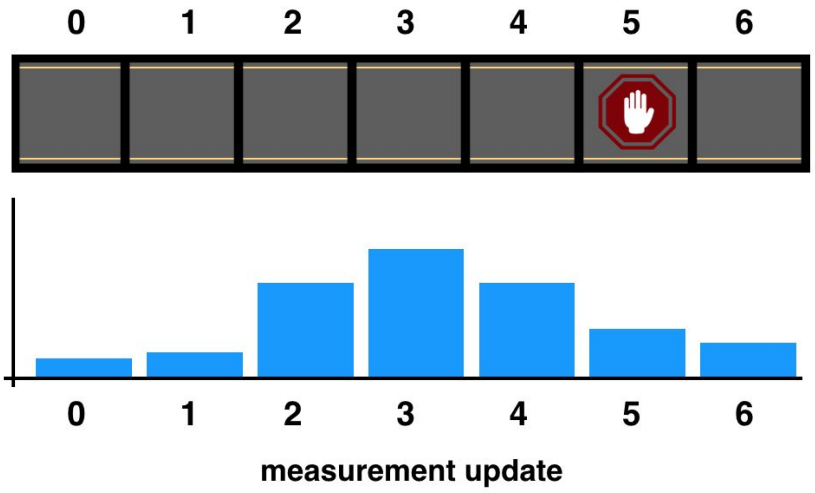


A one lane one and an initial, uniform probability distribution.

2. Measurement Update

We then sense the world around the car. This is called the measurement update step, in which we gather more information about the car's surroundings and refine our location prediction.

Say, we measure that we are about two grid cells in front of the stop sign; our measurement isn't perfect, but we have a much better idea of our car's location.

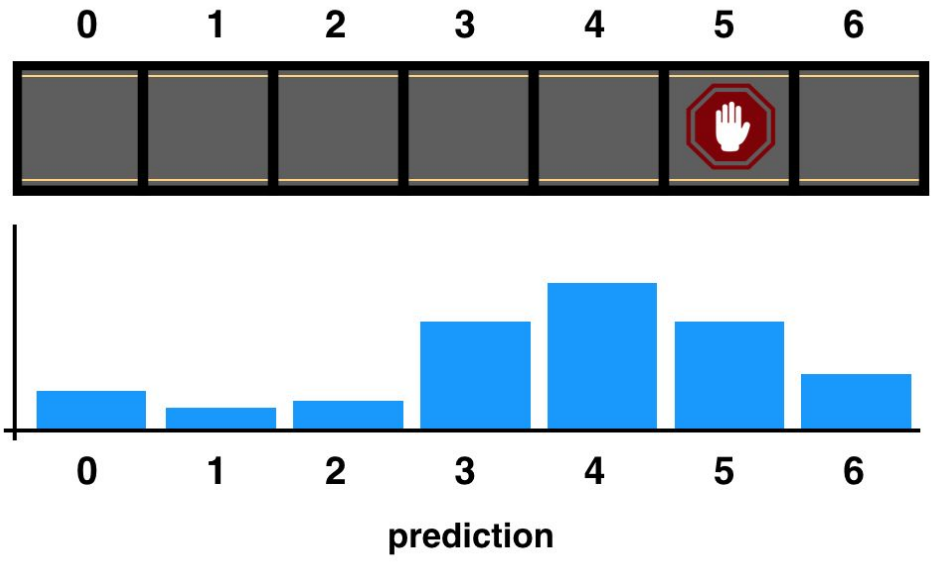


Measurement update step.

3. Prediction (or Time Update)

The next step is moving. Also called the time update or prediction step; we predict where the car will move, based on the knowledge we have about its velocity and current position. And we shift our probability distribution to reflect this movement.

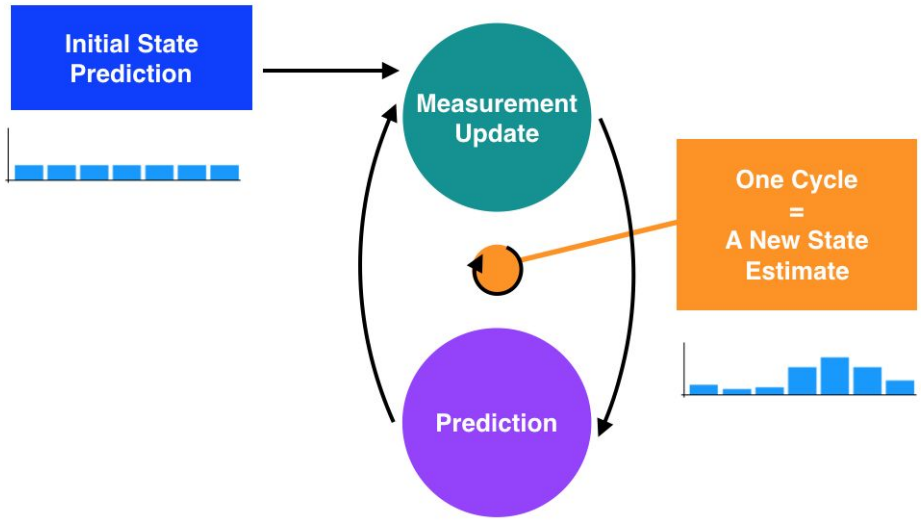
In the next example, we shift our probability distribution to reflect a one cell movement to the right.



Prediction step.

4. Repeat

Then, finally, we've formed a new estimate for the position of the car! The Kalman Filter simply repeats the sense and move (measurement and prediction) steps to localize the car as it's moving!



Kalman Filter steps.

The Takeaway

The beauty of Kalman filters is that they combine somewhat inaccurate sensor measurements with somewhat inaccurate predictions of motion to get a filtered location estimate **that is better than any estimates that come from *only* sensor readings or *only* knowledge about movement.**

What is State?

When you localize a car, you're interested in only the car's position and it's movement.

This is often called the **state** of the car.

- The state of any system is a set of values that we care about.

In the cases we've been working with, the state of the car includes the car's current **position, x,** and its **velocity, v**.

In code this looks something like this:

```
x = 4
```

```
vel = 1
```

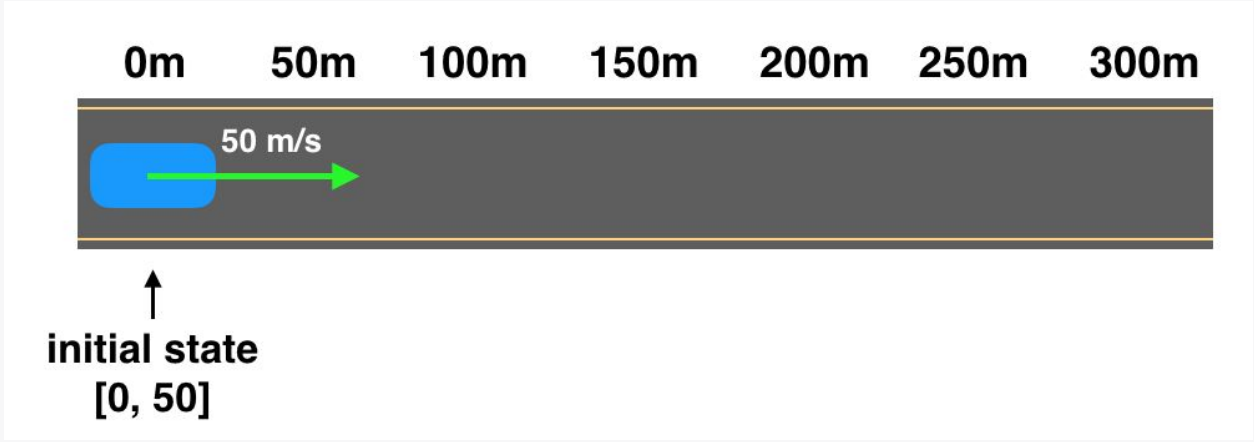
```
state = [x, vel]
```

Predicting Future States

The state gives us most of the information we need to form predictions about a car's future location. And in this lesson, we'll see how to represent state and how it changes over time.

For example, say that our world is a one-lane road, and we know that the current position of our car is at the start of this road, at the 0m mark. We also know the car's velocity: it's moving forward at a rate of 50m/s. These values are it's *initial state*.

```
state = [0, 50]
```



The estimate of the initial state of the car.

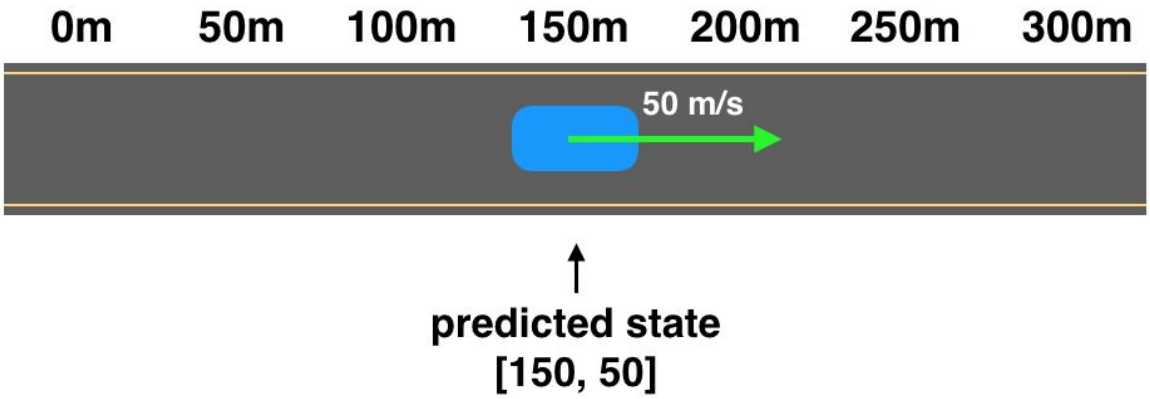
Predicting State

Let's look at the last example.

The initial state of the car is at the 0m position,and it's moving forward at a velocity of 50 m/s. Let's assume that our car keeps moving forward at a constant rate.

Every second it moves 50m.

So, after three seconds, it will have reached the **150m mark** and its velocity will not have changed (that's what a constant velocity means)!



Predicted state after 3 seconds have elapsed.

Its new, predicted state will be at the position 150m, and with the velocity still equal to 50m/s.

```
predicted_state = [150, 50]
```

Motion Model

This is a reasonable prediction, and we made it using:

1. The initial state of the car and
2. An assumption that the car is moving at a constant velocity.

This assumption is based on the physics equation:

```
distance_traveled = velocity * time
```

This equation is also referred to as a **motion model**. And there are many ways to model motion!

This motion model assumes *constant* velocity.

In our example, we were moving at a constant velocity of 50m/s for three seconds.

And we formed our new position estimate with the distance equation: `150m = 50m/sec*3sec`.

The Takeaway

In order to predict where a car will be at a future point in time, you rely on a motion model.

Uncertainty

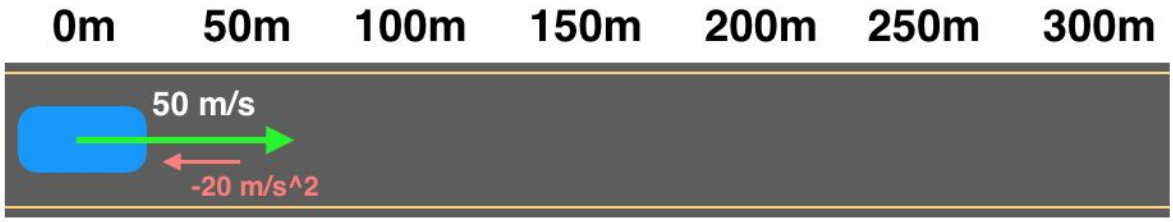
It's important to note, that no motion model is perfect; it's a challenge to account for outside factors like wind or elevation, or even things like tire slippage, and so on.

But these models are still very important for localization.

More Complex Motion

Now, what if I gave you a more complex motion example?

And I told you that our car starts at the same point, at the 0m mark, and it's moving 50m/s forward, but it's *also* slowing down at a rate of 20m/s^2. This means it's acceleration = -20m/s^2.



Car moving at 50m/s and slowing down over time.

Acceleration

So, if the car has a -20 m/s^2 acceleration, this means that:

- If the car starts at a speed of 50m/s
- At the next second, it will be going 50-20 or 30m/s and,
- At the *next* second it will be going 30-20 or 10m/s.

This slowing down is also *continuous*, which means it happens gradually over time.

New Model, New State

For the next two quizzes, I want you to keep in mind this question: **Where will the car be after 3 seconds?**

I also want to ask you:

- What variables do you need to solve this problem? In other words, what values should be included in the state? And...
- What motion model should we use to solve this problem?

Where do you think the car will be after three seconds have elapsed? And what will it's velocity be?

x=60m, vel=-10m/s

Kinematics

Kinematics is the study of the motion of objects. Motion models are also referred to as kinematic equations, and these equations give you all the information you need to be able to predict the motion of a car.

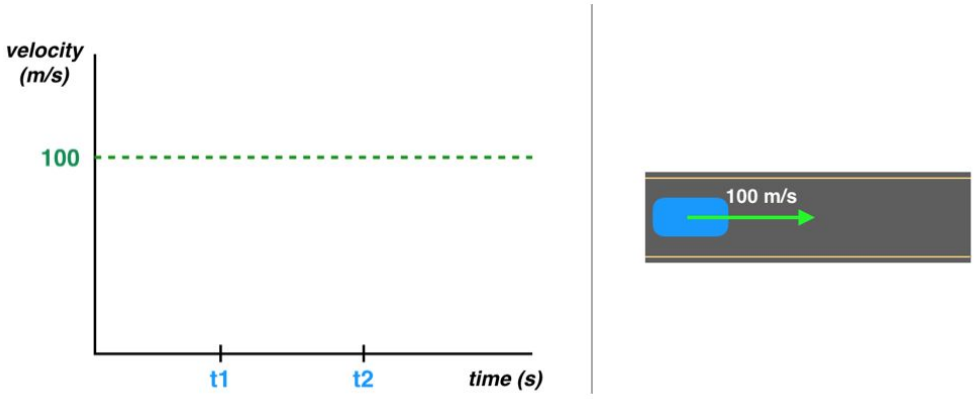
Let's derive some of the most common motion models!

Constant Velocity

The constant velocity model assumes that a car moves at a constant speed. This is the simplest model for car movement.

Example

Say that our car is moving 100m/s, and we want to figure out how much it has moved from one point in time, t1, to another, t2. This is represented by the graph below.



(Left) Graph of car velocity, (Right) a car going 100m/s on a road

Displacement

How much the car has moved is called the **displacement** and we already know how to calculate this!

We know, for example, that if the difference between t2 and t1 is one second, then we'll have moved `100m/sec*1sec = 100m`. If the difference between t2 and t1 is two seconds, then we'll have moved `100m/sec*2sec = 200m`.

The displacement is always = `100m/sec*(t2-t1)`.

Motion Model

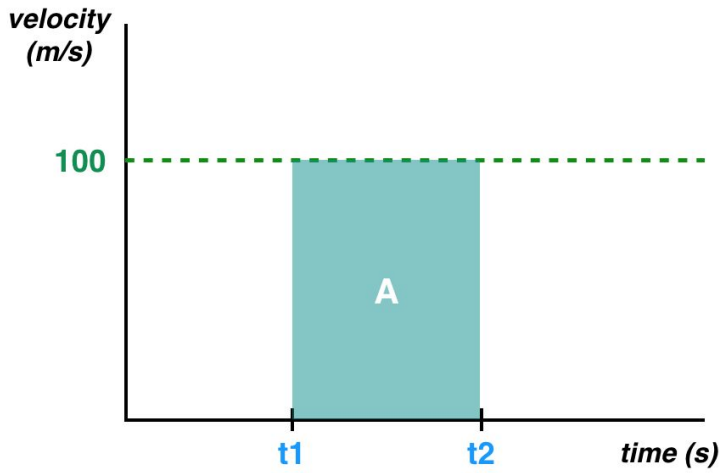
Generally, for constant velocity, the motion model for displacement is:

```
displacement = velocity*dt
```

Where `dt` is calculus notation for *"difference in time."*

Area Under the Line

Going back to our graph, displacement can also be thought of as the area under the line and within the given time interval.



The area under the line, A, is equal to the displacement!

So, in addition to our motion model, we can also say that the displacement is equal to the area under the line!

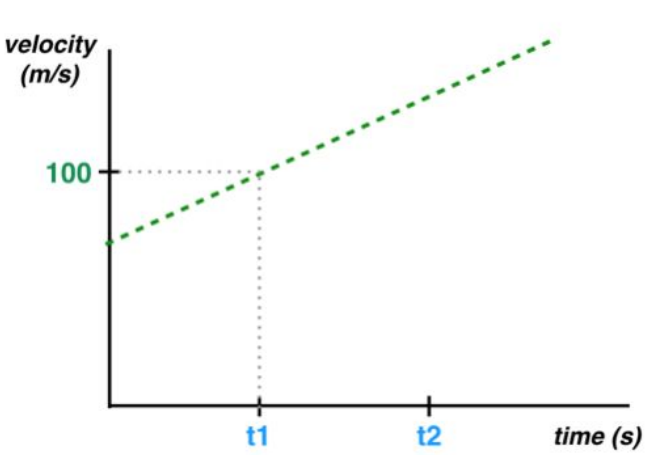
```
displacement = A
```

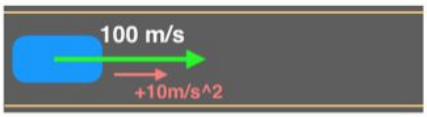
Constant Acceleration

The constant acceleration model is a little different; it assumes that our car is constantly accelerating; its velocity is changing at a constant rate.

Let's say our car has a velocity of 100m/s at time t1 and is accelerating at a rate of

10m/s^2





Changing Velocity

For this motion model, we know that the velocity is constantly changing, and increasing +10m/s each second. This can be represented by this kinematic equation (where dv is the change in velocity):

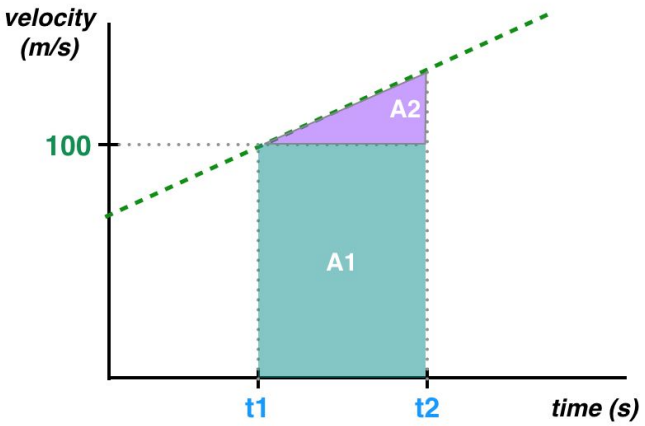
```
dv = acceleration*dt
```

At any given time, this can also be written as the current velocity is the initial velocity + the change in velocity over some time (dv):

```
v = initial_velocity + acceleration*dt
```

Displacement

Displacement can be calculated by finding the area under the line in between t1 and t2, similar to our constant velocity equation but a slightly different shape.



Area under the line, A1 and A2

This area can be calculated by breaking this area into two distinct shapes; a simple rectangle, A1, and a triangle, A2.

A1 is the same area as in the constant velocity model.

```
A1 = initial_velocity*dt
```

In other words, `A1 = 100m/s*(t2-t1)`.

A2 is a little trickier to calculate, but remember that the area of a triangle is `0.5*width*height`.

The width, we know, is our change in time (t2-t1) or `dt`.

And the height is the change in velocity over that time! From our earlier equation for velocity, we know that this value, dv, is equal to: `acceleration*(t2-t1)` or `acceleration*dt`

Now that we have the width and height of the triangle, we can calculate A2. Note that`**` is a Python operator for an exponent, so `**2` is equivalent to `^2` in mathematics or squaring a value.

```
A2 = 0.5*acceleration*dt**2
```

Motion Model

This means that our total displacement, A1+A2 ,can be represented by the equation:

```
displacement = initial_velocity*dt + 0.5*acceleration*dt**2
```

We also know that our velocity over time changes according to the equation:

```
dv = acceleration*dt
```

And these two equations, together, make up our motion model for constant acceleration.

Different Motion Models

Constant Velocity

In the first movement example, you saw that if we assumed our car was moving at a constant speed, 50 m/s, we came up with one prediction for it's new state: at the 150 m mark, with no change in velocity.

```
# Constant velocity case


# initial variables

x = 0

velocity = 50

initial_state = [x, velocity]


# predicted state (after three seconds)

# this state has a new value for x, but the same velocity as in the initial state

dt = 3

new_x = x + velocity*dt

predicted_state = [new_x, velocity]  # predicted_state = [150, 50]
```

For this constant velocity model, we had:

- initial state = `[0, 50]`
- predicted state (after 3 seconds) = `[150, 50]`

Constant Acceleration

But in the second case, we said that the car was slowing down at a rate of 20 m/s^2 and, after 3 seconds had elapsed, we ended up with a different estimate for its state.

To solve this localization problem, we had to use a different motion model and **we had to include a new value in our state: the acceleration of the car**.

The motion model was for constant acceleration:

- `distance = velocity*dt + 0.5*acceleration*dt^2` and
- `velocity = acceleration*dt`

The state includes acceleration in this model and looks like this: `[x, velocity, acc]`.

```
# Constant acceleration, changing velocity


# initial variables

x = 0

velocity = 50

acc = -20


initial_state = [x, velocity, acc]


# predicted state after three seconds have elapsed

# this state has a new value for x, and a new value for velocity (but the acceleration stays the same)

dt = 3


new_x = x + velocity*dt + 0.5*acc*dt**2

new_vel = velocity + acc*dt


predicted_state = [new_x, new_vel, acc]  # predicted_state = [60, -10, -20]
```

For this constant *acceleration* model, we had:

- initial state = `[0, 50, -20]`
- predicted state (after 3 seconds) = `[60, -10, -20]`

As you can see, our state and our state estimates vary based on the motion model we used and how we assumed the car was moving!

How Many State Variables?

In fact, how many variables our state requires, depends on what motion model we are using.

For a constant velocity model, `x` and `velocity` will suffice.

But for a constant acceleration model, you'll also need our acceleration:`acc`.

But these are all just models.

The Takeaway

For our state, we always choose **the smallest representation** (the smallest number of variables) that will work for our model.

Lesson Outline

The one unifying theme in this lesson is representing and predicting state, but there are two threads that we'll use to explore this idea.

1. Object-Oriented Programming

On the programming side, we'll use something called Object-Oriented Programming as a way to represent state in code. We'll use `variables` to represent state values and we'll create `functions` to change those values.

2. Linear Algebra

On the mathematical side we'll use `vectors` and `matrices` to keep track of state and change it.

We'll learn all the necessary math notation and code, as we learn more about predicting the state of a car!

Always Moving

Self-driving cars constantly monitor their state. So, movement and localization have to occur in parallel.

If we use a Kalman filter for localization, this means that as a car moves, the Kalman filter has to keep coming up with new state estimates. This way, the car *always* has an idea of where it is.

Always Predicting State

In the code below, you are given a `predict_state` function that takes in a current state and a change in time, `dt`, and returns the new state estimate (based on a constant velocity model).

It will be up to you to use this function repeatedly to find the predicted_state at 5 different points in time:

- the initial state
- the predicted state after 2 seconds have elapsed
- the predicted state after 3 *more* seconds have elapsed
- the predicted state after 1 *more* second has elapsed
- the predicted state after 4 *more* seconds have elapsed

To first three states have been given to you in code.

```
#---- predict state function --#

def predict_state(state, dt):

    # Assumes a valid state had been passed in

    # Assumes a constant velocity model

    x = state[0]

    new_x = x+state[1]*dt


    # Create and return the new, predicted state

    predicted_state = [new_x, state[1]]

    return predicted_state


# predict_state takes in a state and a change in time, dt

# So, a call might look like: new_state = predict_state(old_state, 2)


# The car starts at position = 0, going 60 m/s

# The initial state:

initial_state = [10, 60]
```

```
# After 2 seconds:
state_est1 = predict_state(initial_state, 2)


# 3 more seconds after the first estimated state
state_est2 = predict_state(state_est1, 3)


## Use the predict_state function
## and the above variables to calculate the following states


## 1 more second after the second state estimate
state_est3 = predict_state(state_est2, 1)


## 4 more seconds after the third estimated state
state_est4 = predict_state(state_est3, 4)
```

Creating a Car Object

To create a Car, which was named `carla` in the example. I have to:

1. Import our car file and define a car's initial state variable, and
2. Call `car.Car()`; a special function that initializes a Car object, and pass in the initial state variables.

The state is defined by a position: [y, x] and a velocity, which has vertical and horizontal components: [vy, vx]. And lastly, we had to pass in a world, which is just a 2D array.

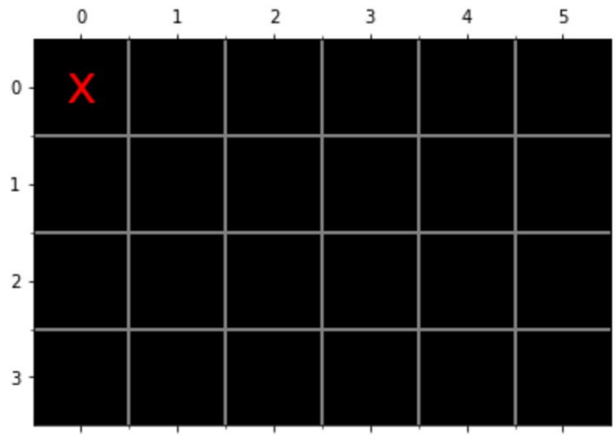
Imports and Defining initial variables

```
# Import statements
import numpy
import car


# Declare initial variables
```

```
# Create a 2D world of 0's
height = 4
width = 6
world = np.zeros((height, width))

# Define the initial car state
initial_position = [0, 0] # [y, x] (top-left corner)
velocity = [0, 1] # [vy, vx] (moving to the right)
```

Creating and Visualizing a Car!

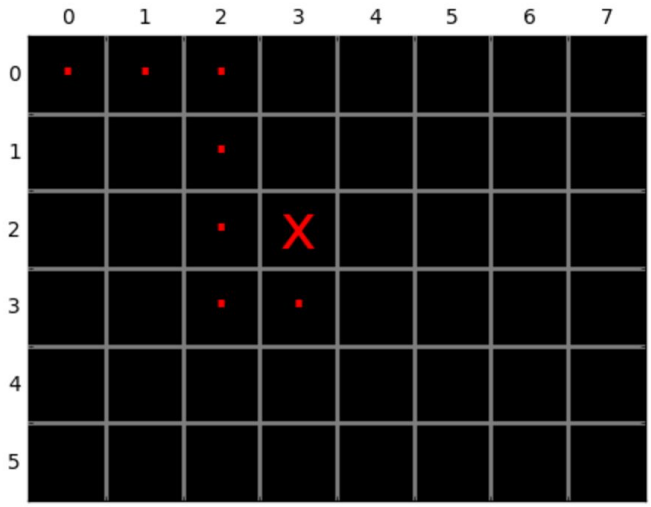
```
# Create a car object with these initial params
carla = car.Car(initial_position, velocity, world)

# Display the world
carla.display_world()
```



Carla's initial state at position [0,0]

Car movement

Carla can also move in the direction of the velocity and turnleft with the functions: `move()` and `turn_left()`.



Car path after some movement.

Overloading

Now that we've seen how to create a class and talked about some of the fundamental functions and variables that classes contain, let's look at something new!

The double underscore `__x__`

You've seen a couple of examples of functions that have a double underscore, like:

`__init__`

`__repr__`

These are **special functions** that are used by Python in a specific way.

We typically don't call these functions directly, as we do with ones like `move()` and `turn_left().

Instead, **Python calls them *automatically*** based on our use of keywords and operators.

For example, `__init__` is called when we create a new object and `__repr__` is called when we tell Python to print the string representation of a specific object!

Another example: `__add__`

All of these special functions have their names written between double underscores `__`, and there are many of these types of functions! To see the full list of these functions, check out the Python documentation.

For example, we can define what happens when we add two car objects together using a `+` symbol by defining the `__add__` function.

```
def __add__(self, other):
    # Create an empty list
    added_state = []

    # Add the states together, element-wise
    for i in range(self.state):
        added_value = self.state[i] + other.state[i]
        added_state.append(added_value)

    return added_state
```

The above version, adds together the state variables! Or.. you may choose to just print out that adding cars is an invalid operation, as below.

```
def __add__(self, other):
    # Print an error message and return the unchanged, first state
    print('Adding two cars is an invalid operation!')
    return self.state
```

Operator Overloading

When we define these functions in our class, this is called **operator overloading**.

And, in this case, overloading just means: *giving more than one meaning to* a standard operator like addition.

Operator overloading can be a powerful tool, and you'll not only see it pop up again and again in classes, but it is useful for writing classes that are intuitive and simple to use. So, keep this in mind as you continue learning, and **let's get some practice with overloading operators!**

State Vector

So far, you've seen that the state of a car contains multiple values which we've been putting into a Python list.

However, these values are *typically* contained in one data type: a **state vector**.

A vector is similar to a list in Python, in that it contains multiple values, but they are mathematically very different.

What is a Vector?

A vector is a kind of matrix, which for now you can think of as a grid of numbers, and matrices are a math data type, similar to something like an integer or a float.

And similar to an integer or a float, we can multiply matrices, scale them, add two of them together and so on!

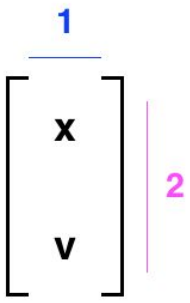
This is called **linear algebra**, and linear algebra will come up again and again in self-driving car systems. It's in everything from:

- car movement calculations
- deep learning applications, and
- image and video analysis

In this section, we'll see how state vectors and matrices can be used together to help us efficiently predict a new state, and localize a car.

State Vector

A state vector is a **column of values** whose dimensions are 1 in width and M in height. This vector should contain all the values that we are interested in, and for predicting 1D movement, we are interested in the position, x, and the velocity, v.



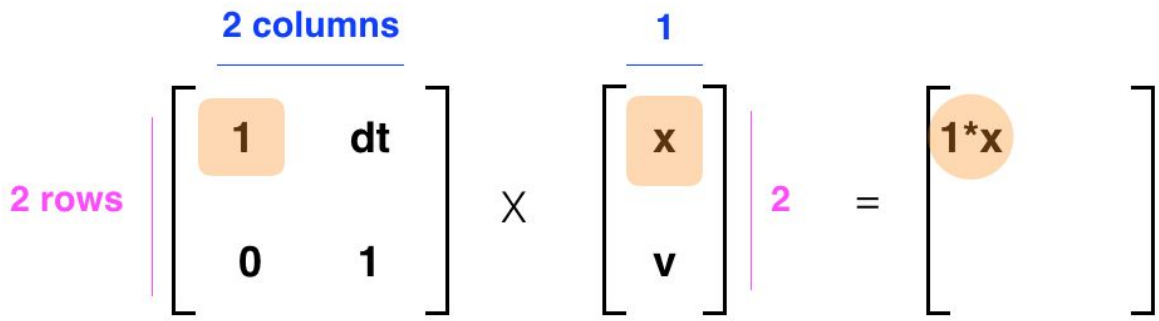
An example of a 2x1 state vector that contains variables: x and v.

Efficiently predicting state

With a state vector, we can predict a new state in just one **matrix multiplication** step.

Matrix multiplication

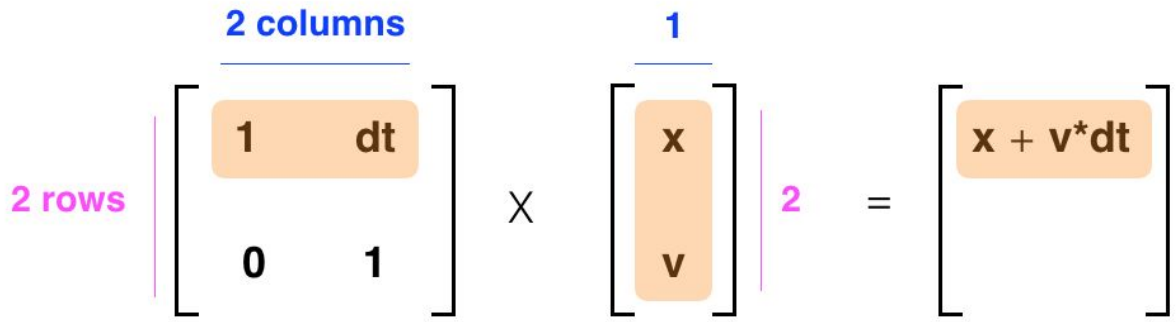
Matrix multiplication multiplies two grids of numbers; multiplying the rows in the first matrix, by the columns in the second. One step in this process is pictured, below.



The start of matrix multiplication for these 2x2 and 2x1 matrices.

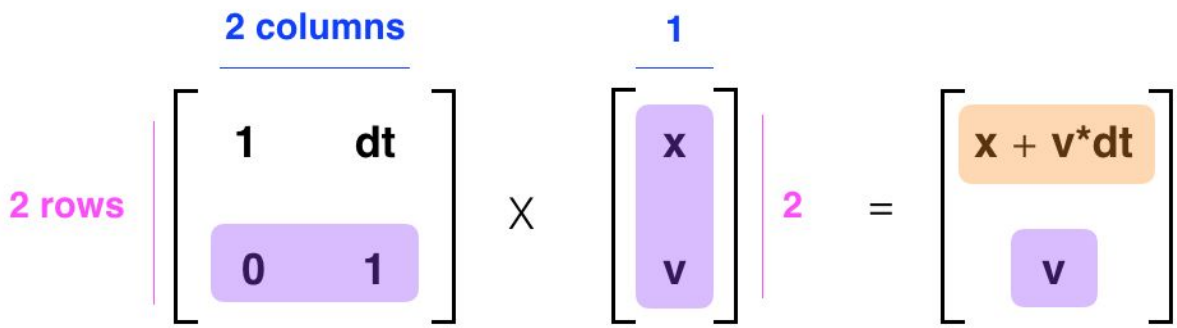
Summing step

Once a whole row and column have been multiplied, matrix multiplication sums those values to form a single, new value in a resulting matrix.



Summation step: x + v*dt

Then it moves on to the next row, and this process repeats.



Completed matrix multiplication!

You can see that this creates a new 2x1 vector, with two values in it that may look familiar! These are just our equations for our **constant velocity motion model**. So, matrix multiplication let's us create a new, predicted state vector in just one multiplication step!

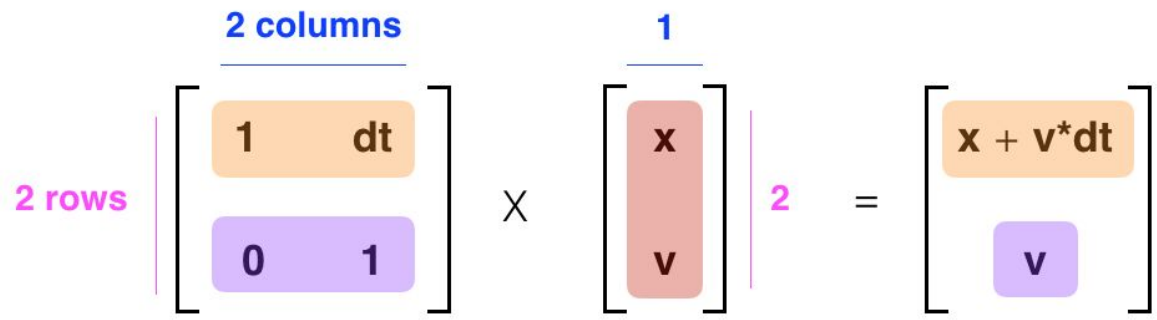
In fact, this is such a common way to predict a new state, that the 2x2 matrix on the left is often called the **state transformation matrix**.

Matrix Multiplication

Now that you've seen how Matrix multiplication is used in state transformation, let's go through some concrete examples, and test your knowledge!

Remember that multiplying two matrices involves a couple steps:

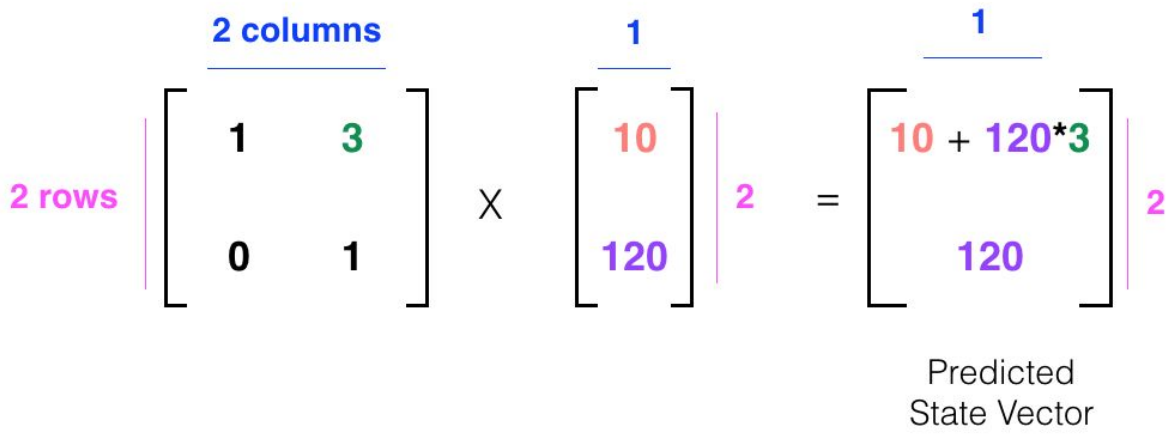
1. It multiplies each row in the first matrix with the columns in the second, element-wise (that means the number of columns in the first matrix and rows in the second **must be equal**).
2. It sums up those multiplied values to form a new value in a new matrix at a specific location.
    - **ex.** If we multiply the *first row* of the first matrix by the *first column* of the second, this new, summed value belongs in the *first row and first column* of the resulting matrix.
    - **ex.** If we multiply the *second row* of the first matrix by the *first column* of the second, this new, summed value belongs in the *second* row and first column of the resulting matrix.
3. Matrix multiplication takes practice, so take a look at this page for more examples!



State transformation by matrix multiplication.

Let's consider the case where the position of a self-driving car is at: `x = 10` , and velocity,`v = 120`.

Where would you predict that the car with be in 3 seconds, using matrix multiplication? Well, we'd simply plug in the numbers for x, v, and dt in the transformation equation:

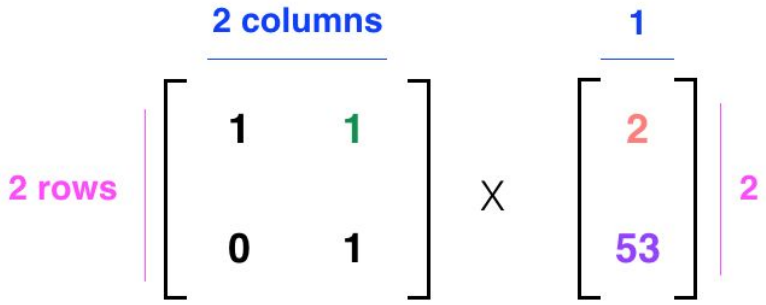


New, predicted state vector.

Predicted State Vector

We'd get a new, predicted state vector with x = 10+120*3. and a constant velocity.

`x = 370` and `v = 120`

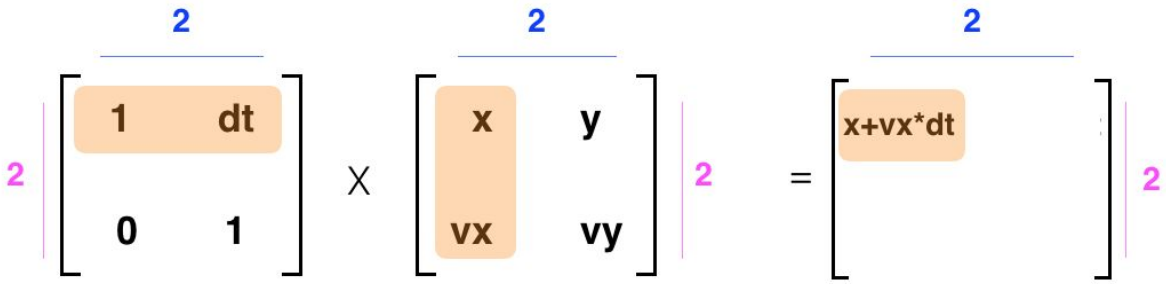


dt = 1, x = 2, and v = 53

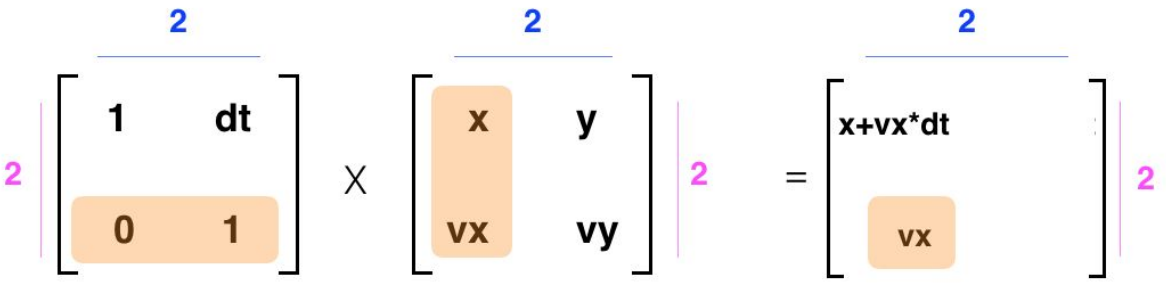
Matrix Multiplication

Let's walk through that last quiz example, step-by-step.

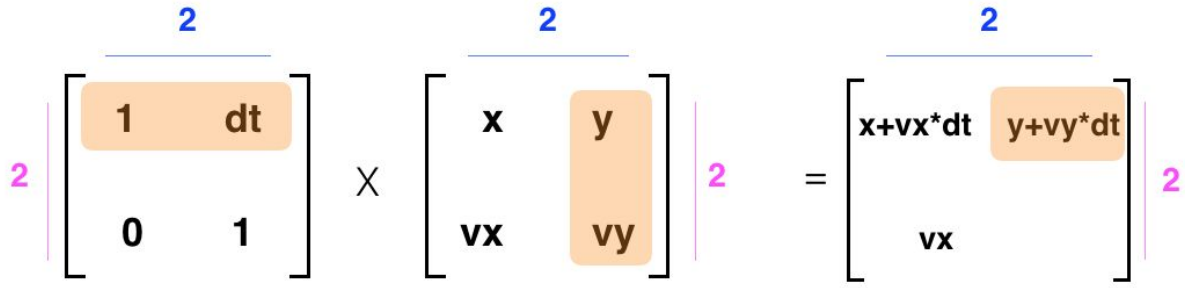
- Multiply the first row by the first column and sum.



- Then, the second row, by the first column.



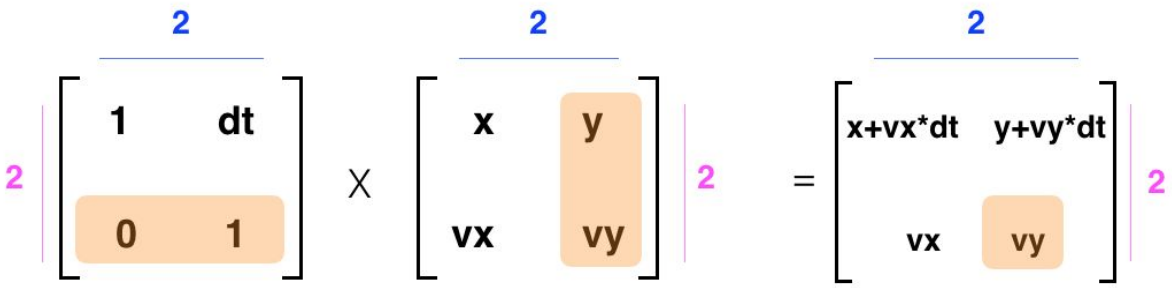
- Then *back* to the first row, this time, multiplied by the second column.



And, finally the last step:

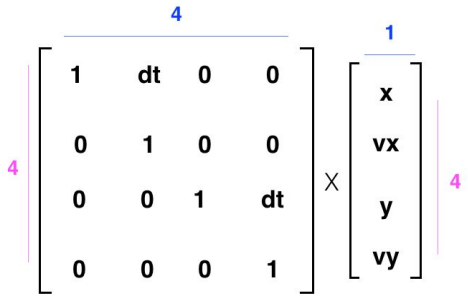
- The last row multiplied by the last column.

To get our complete, resulting matrix!



Constant velocity

This kind of multiplication can be really useful, if x and y are not dependent on one another. That is, there is a separate and constant x-velocity and y-velocity component. For real-world, curved and continuous motion, we still use a state vector that is one column, so that we can handle any x-y dependencies. So, you'll often see state vector and transformation matrices that look like the following.



State vector equivalent

These extra spaces in the matrix allow for more detailed motion models and can account for a x and y dependence on one another (just think of the case of circular motion). So, **state vectors are always column vectors**.

State Transformation and Linear Algebra

Now you've seen that we can transform the *state* of a car using matrix multiplication.

This kind of linear algebra can be used to update multiple state variables in just one line of code! And this becomes really useful when you're working with big datasets and variables that represent our 3 dimensional world.

Next, you'll see how linear algebra is used to create a *two* dimensional kalman filter.

In the process, you'll learn more about matrix operations and notation, and you'll be one step closer to creating an algorithm that is really used to localize self-driving cars. **Great job getting this far!**