# A Note on Notation

In the previous video (and in the next) you saw the following equation:

x' = x + \dot{x}*x*′=*x*+*x*˙

Translated into plain speech, this says

the x position **after** motion (x'*x*′) is equal to the x position **before** motion (x*x*) plus the velocity in the x direction (\dot{x}*x*˙).

If you read through that statement, you might notice that it doesn't quite make sense because it doesn't take into account the **duration** of motion. If I drive for 10 seconds I go farther than if I only drive for 1 second!

In the previous video we are assuming that the duration of motion (typically called \Delta tΔ*t*) is equal to 1 second. The "complete" version of the equation above would be

x' = x + \dot{x} \Delta t*x*′=*x*+*x*˙Δ*t*

| **Symbol** | **Meaning** |
| --- | --- |
| x*x* | x position **before** motion |
| x'*x*′ | x position **after** motion |
| \dot{x}*x*˙ | velocity in x direction |
| \Delta tΔ*t* | duration of motion "delta t" |

The assignment

x\_prime = x + x\_dot

assumes that the time interval delta\_t is equal to 1. A more general formula is

x\_prime = x + x\_dot\***delta\_t**

NEXT

# Kalman Filter Equations Fx versus Bu

Consider this specific Kalman filter equation: x' = Fx + Bu*x*′=*Fx*+*Bu*.

This equation is the move function that updates your beliefs in between sensor measurements. Fx models motion based on velocity, acceleration, angular velocity, etc of the object you are tracking.

B is called the control matrix and u is the control vector. Bu measures extra forces on the object you are tracking. An example would be if a robot was receiving direct commands to move in a specific direction, and you knew what those commands were and when they occurred. Like if you told your robot to move backwards 10 feet, you could model this with the Bu term.

When you take the self-driving car engineer nanodegree, you'll use Kalman filters to track objects that are moving around your vehicle like other cars, pedestrians, bicyclists, etc. In those cases, you would ignore Bu*Bu* because you do not have control over the movement of other objects. The Kalman filter equation becomes x' = Fx*x*′=*Fx*.

# The Rest of the Lesson

The rest of this lesson will introduce you to **vectors**, **matrices** and the **operations** associated with them. Most of the instruction will be in the form of code demonstrations.

Depending on your background you may need to adjust your pace through the rest of this lesson. If you have a mathematical background then there's a good chance you'll move very quickly through this content. If you haven't worked with math in a while you may need to go more slowly.

A lot of people have bad memories with math and trying to learn something like linear algebra can sometime bring those memories back. If you start to feel stressed just step back, take a breath and remember that math is just another thing to learn. You can do it!

If you get stuck, don't hesitate to ask for help in the Study Groups or Knowledge. You've got classmates and staff who will be thrilled to help you learn!

# Representing State with Matrices

### The State Vector

You just learned how to represent a self-driving car's state using a motion model. It turns out that matrices provide a very convenient and compact form for representing a vehicle's state.

Let's go back to the constant velocity motion model:

distance = velocity \times time*distance*=*velocity*×*time*

The vehicle's state is represented by the two variables distance and velocity. If you were going to store these two variables in Python, you'd probably use a list like this:

state = [distance, velocity]

That Python code looks a lot like a mathematical concept called a vector. A vector is essentially a list where each element in the list contains some information.

You could imagine that a state vector could have even more information. In a two-dimensional world, the state could have a distance\_x*distancex*​, distance\_y*distancey*​, velocity\_x*velocityx*​ and a velocity\_y*velocityy*​.

In Python, the list would look like this:

state = [distance\_x, distance\_y, velocity\_x, velocity\_y]

Or an even more complex model could include information about the turning angle of the vehicle and the turning rate:

state = [distance\_x, distance\_y, velocity\_x, velocity\_y, angle, angle\_rate]

### State Vector in a One-Dimensional World

For now, consider the one-dimensional model with just distance and velocity.

state = [distance, velocity]

How did you calculate the distance and velocity of the vehicle over time when the velocity was constant? There were two different equations.

\begin{cases} distance = velocity \times time \\ velocity = velocity \end{cases}{*distance*=*velocity*×*timevelocity*=*velocity*​

Now, think about these equations in terms of state. For convenience, you can represent distance as x*x*, velocity as v*v*, and time as t*t*.

##### Initial State

When the vehicle first starts moving, you can consider that t = t\_0*t*=*t*0​, x = x\_0*x*=*x*0​ and v = v\_0*v*=*v*0​. So the state vector is state\_0 = [x\_0, v\_0]*state*0​=[*x*0​,*v*0​].

##### First Time Step

What about after a certain amount of time has passed and now you are at a time t\_1*t*1​?

At t\_1*t*1​, the state vector is:

state\_1 = [x\_1, v\_1]*state*1​=[*x*1​,*v*1​]

According to the model formulas,

x\_1 = x\_0 + v\_0 \times (t\_1 - t\_0)*x*1​=*x*0​+*v*0​×(*t*1​−*t*0​)

and since velocity is constant:

v\_1 = v\_0*v*1​=*v*0​.

##### Second Time Step

Then after the next time step t\_2*t*2​, the state vector is: state\_2 = [x\_2, v\_2]*state*2​=[*x*2​,*v*2​]

where

x\_2 = x\_1 + v\_0 \times (t\_2 - t\_1)*x*2​=*x*1​+*v*0​×(*t*2​−*t*1​)

and since velocity is constant v\_2 = v\_0*v*2​=*v*0​.

### A Better Way

The math so far is not too hard, right? You have a distance equation and a velocity equation. You plug in the previous state into each equation, the time lapse, and you get the new velocity and the new distance.

But imagine what will happen as your self-driving car model gets more complex. What happens when you have to take into account an x-direction, y-direction, x and y velocities, and steering angle and angular velocity? Or what about an even more complex model like a drone or helicopter that also has a z-direction?

Instead of updating your equations one by one, you can actually use vectors and matrices to do all of the calculations in just one step.

### Updating State with Matrix Algebra

If you look back at the equation that updates the distance, you'll notice that distance depends on the previous distance, the initial velocity, and how much time has elapsed since the distance formula was updated. You end up with a generic function:

x\_{t+1} = x\_{t} + v\_0 \times (t\_{t+1} - t\_{t})*xt*+1​=*xt*​+*v*0​×(*tt*+1​−*tt*​)

You can also write t\_{t+1} - t\_{t}*tt*+1​−*tt*​ as:

\Delta tΔ*t*

For a constant velocity model, the generic velocity equation becomes: v\_{t+1} = v\_{t}*vt*+1​=*vt*​

How could you combine the x and v equations into one matrix algebra expression? The matrix algebra would look like this:

\begin {bmatrix} x\_{t+1} \\ v\_{t+1} \end {bmatrix} = \begin {bmatrix} 1& \Delta t \\ 0 & 1 \end{bmatrix} \times \begin {bmatrix} x\_{t} \\ v\_{t} \end{bmatrix}[*xt*+1​*vt*+1​​]=[10​Δ*t*1​]×[*xt*​*vt*​​]

Don't worry if you're not sure what this expression means or how to multiply these matrices. You will learn how in this lesson.

### Notation

The matrix algebra equation you just saw is actually one part of the Kalman filter update equation.

Traditionally, the matrix operation:

\begin {bmatrix} x\_{t+1} \\ v\_{t+1} \end {bmatrix} = \begin {bmatrix} 1& \Delta t \\ 0 & 1 \end{bmatrix} \times \begin {bmatrix} x\_{t} \\ v\_{t} \end{bmatrix}[*xt*+1​*vt*+1​​]=[10​Δ*t*1​]×[*xt*​*vt*​​]

is represented by this notation for Kalman filters: \mathbf{\hat{x}\_{k|k-1}} = \mathbf{F} \mathbf{\hat{x}\_{k-1|k-1}}**x**^**k∣k**−**1**​=**Fx**^**k**−**1∣k**−**1**​

where \mathbf{\hat{x}}**x**^ is the state vector and \mathbf{F}**F** is the matrix

\begin {bmatrix} 1& \Delta t \\ 0 & 1 \end{bmatrix}[10​Δ*t*1​]

\mathbf{k - 1}**k**−**1** is the previous step and \mathbf{k }**k** is the current step.

You will see in the next part of the lesson why the notation contains \mathbf{k-1|k-1}**k**−**1∣k**−**1** and \mathbf{k|k-1}**k∣k**−**1** .

This notation can get a little bit confusing. For example, what is the difference between x*x* and \mathbf{\hat{x}}**x**^?

The regular x*x* would usually represent distance along the x-axis; on the other hand, the bold \mathbf{\hat{x}}**x**^ indicates a vector. In the one-dimensional case being discussed here, the \mathbf{\hat{x}}**x**^ vector contains two variables: distance along the x-axis and velocity; hence \bold{\hat{x}} = \begin{bmatrix} x \\ v \end{bmatrix}**x**^=[*xv*​].

Why is there a capitalized bold \mathbf{F}**F** instead of \mathbf{f}**f**? The capitalized, bold \mathbf{F}**F** tells you that this variable is a matrix.

# Kalman Equation Reference

We're just including this here in case you want to refer back to the Kalman Filter equations at any time. Feel free to move along :)

### Variable Definitions

\mathbf{\hat{x}}**x**^ - state vector

\mathbf{F}**F** - state transition matrix

\mathbf{P}**P** - error covariance matrix

\mathbf{Q}**Q** - process noise covariance matrix

\mathbf{R}**R** - measurement noise covariance matrix

\mathbf{S}**S** - intermediate matrix for calculating Kalman gain

\mathbf{H}**H** - observation matrix

\mathbf{K}**K** - Kalman gain

\mathbf{\tilde{y}}**y**~​ - difference between predicted state and measured state

\mathbf{z}**z** - measurement vector (lidar data or radar data, etc.)

\mathbf{I}**I** - Identity matrix

**Prediction Step Equations**

PREDICT STATE VECTOR AND ERROR COVARIANCE MATRIX

\mathbf{\hat{x}\_{k|k-1}} = \mathbf{F\_{k}} \mathbf{\hat{x}\_{k-1|k-1}}**x**^**k∣k**−**1**​=**Fk**​**x**^**k**−**1∣k**−**1**​

\mathbf{P\_{k|k-1}} = \mathbf{F\_{k}} \mathbf{P\_{k-1|k-1}} \mathbf{F\_{k}^T} + \mathbf{Q\_{k}}**Pk∣k**−**1**​=**Fk**​**Pk**−**1∣k**−**1**​**FkT**​+**Qk**​

**Update Step Equations**

KALMAN GAIN

\mathbf{S\_{k}} = \mathbf{H\_{k}} \mathbf{P\_{k|k-1}} \mathbf{H\_{k}^T} + \mathbf{R\_{k}}**Sk**​=**Hk**​**Pk∣k**−**1**​**HkT**​+**Rk**​

\mathbf{K\_{k}} = \mathbf{P\_{k|k-1}} \mathbf{H\_{k}^T} \mathbf{S\_{k}}^{-1}**Kk**​=**Pk∣k**−**1**​**HkT**​**Sk**​−1

UPDATE STATE VECTOR AND ERROR COVARIANCE MATRIX

\mathbf{\tilde{y\_{k}}} = \mathbf{z\_{k}} - \mathbf{H\_{k}} \mathbf{\hat{x}\_{k|k-1}}**yk**​~​=**zk**​−**Hk**​**x**^**k∣k**−**1**​

\mathbf{\hat{x}\_{k|k}} = \mathbf{\hat{x}\_{k|k-1}} +\mathbf{ K\_{k}} \mathbf{\tilde{y\_{k}}}**x**^**k∣k**​=**x**^**k∣k**−**1**​+**Kk**​**yk**​~​

\mathbf{P\_{k|k}} = (\mathbf{I} - \mathbf{ K\_{k}} \mathbf{H\_{k}}) \mathbf{P\_{k|k-1}}**Pk∣k**​=(**I**−**Kk**​**Hk**​)**Pk∣k**−**1**​

# What is a vector? Physics versus Computer Programming

You might have learned at some point that a vector is a measurement or quantity that has both a **magnitude** and a **direction**. Examples might be distance along a y-axis or velocity towards the north-west.

But in computer programming, when we say "vector" we are just referring a **list of values**.

These two ways of thinking about vectors are actually deeply related, but for this Nanodegree we're going to look at vectors from a computer science point of view.

### Vectors, Motion Models and Kalman Filters in Self-Driving Cars

In a physics class, you might have one vector for position and then a separate vector for velocity. But in computer programming, a vector is really just a list of values.

When using the Kalman filter equations, the bold, lower-case variable \mathbf{x}**x** represents a vector in the computer programming sense of the word. The \mathbf{x}**x** vector holds the values that represent your motion model such as position and velocity.

In a basic motion model, the vector \mathbf{x}**x** will contain information about position and linear velocity like: \mathbf{x} = [x, y, v\_x, v\_y]**x**=[*x*,*y*,*vx*​,*vy*​]. In a physics class, these might be represented with two different vectors: a position vector and a velocity vector.

A more complex motion model might take into account yaw rate, which considers a rotational angle and angular velocity around the center of the vehicle like \mathbf{x} = [x, y, v\_x, v\_y, \psi, \dot{\psi}]**x**=[*x*,*y*,*vx*​,*vy*​,*ψ*,*ψ*˙​].

So in order to use the Kalman filter equations and execute object tracking, you have to be familiar with vectors and how to write programs with them.

# Vectors in Python

In Python, you can represent a vector with lists. So a vector like \begin{bmatrix}5, 9, 10, 2, 20\end{bmatrix}[5,9,10,2,20​] could be represented with

myvector = [5, 9, 10, 2, 20]

#### Vector Indexing

If you wanted to access values inside the vector, you would use indexing. The first value, which in this case is 5, would be accessed by

myvector[0]

The second value:

myvector[1]

And so on.

#### Assigning Values to Vectors

If you wanted to change a value in the vector, you could use this syntax:

myvector[3] = 19

which would change the fourth element from 20 to 19.

To add a value to the end of a vector, you can use the .append() method:

myvector = [5, 9, 2, 20]

myvector.append(18)

and the resulting vector would be [5, 9, 2, 20, 18].

### Vector Math in Python

There are a few vector operations you'll want to become familiar with in order to use the Kalman filter equations:

* vector addition
* scalar multiplication
* the dot product

#### Vector Addition

To add two vectors together, you sum them element by element. For example,

\mathbf{v\_1} = [a, b, c, d]**v1**​=[*a*,*b*,*c*,*d*]

\mathbf{v\_2} = [w, x, y, z]**v2**​=[*w*,*x*,*y*,*z*]

The sum of these two vectors would then be:

\mathbf{v\_1} + \mathbf{v\_2} = [a + w, b + x, c + y, d + z]**v1**​+**v2**​=[*a*+*w*,*b*+*x*,*c*+*y*,*d*+*z*]

Say that you know the current state of your vehicle \mathbf{x} = [x, y, v\_x, v\_y]**x**=[*x*,*y*,*vx*​,*vy*​]

If you knew the change in the position and velocity of your vehicle, you could use vector addition to find the new state vector. You will code this in the vector coding exercises.

##### Vector addition Python Code

How might you execute vector addition using Python? In general, for loops are very useful for accessing values inside of a Python list:

**for** i **in** range(len(v1)):

v1[i]

would give you access to v1, one element at a time. So you could access each value of v1 and v2, sum them together, and then append each sum to a new vector like:

*# initialize an empty vector*

vsum = []

*# use for loops to take the sum of a value and then append to vsum*

**for** ....

.....

vsum.append(value)

We are not showing you the full answer because you will be figuring this out in a coding exercise.

### Scalar Multiplication

Scalar multiplication involves multiplying each element in a vector by a constant.

Here is a concrete example.

If \mathbf{v} = [a, b, c, d]**v**=[*a*,*b*,*c*,*d*], then 5\mathbf{v} = [5a, 5b, 5c, 5d]5**v**=[5*a*,5*b*,5*c*,5*d*]

You will also implement scalar multiplication in the coding exercises. Think about how you can use a for loop and the append method to code scalar multiplication.

A use case for scalar multiplication could be changing between units of measurement. For example, if your state were measured in meters, you could use scalar multiplication to convert to feet. Get ready to code this in the vector coding exercises!

#### Dot Product

The dot product of two vectors is very important for matrix multiplication. If

\mathbf{v\_1} = [a, b, c, d]**v1**​=[*a*,*b*,*c*,*d*]

\mathbf{v\_2} = [w, x, y, z]**v2**​=[*w*,*x*,*y*,*z*],

then the dot product would be

\mathbf{v\_1 \cdot v\_2} = aw+ bx + cy + dz**v1**​⋅**v2**​=*aw*+*bx*+*cy*+*dz*

This is another place where a for loop would be useful. You will figure out how to code the dot product in the exercises in the next section.

You'll see how to apply the dot product and why it is useful in the coding exercises.

### Summary

Here is a summary of what you will need to know about Python lists to complete the coding exercises:

* assigning a vector - myvector = [5, 9, 10 2, 20]
* accessing a value - myvector[3]
* changing a value - myvector[3] = 15
* appending a value to the end of the list - myvector.append(6)
* and finally, using for loops to access and manipulate vectors:

**for** i **in** range(len(myvector)):

myvector[i]

Check out this [**link**](https://docs.python.org/3/tutorial/datastructures.html) to learn about other methods that come with Python lists. You might find them useful although you will not need them for the following exercises.

Next, you will practice coding vectors in Python.

NEXT

### Vector versus Matrix

Vectors are one part of the Kalman filter equations. But you also need to be able to use matrices.

You've seen how Python can represent a vector in a list. A vector can be thought of like a simple grid with one row and a column for each element. If you thought of a vector like a grid, the vector \begin{bmatrix}17, 25, 6, 2\end{bmatrix}[17,25,6,2​] would be represented like this:

A picture containing clock

Description automatically generated

But you could also call this vector a matrix. This four element vector is a one by four matrix or 1x4. The one represents the number of rows and the four represents the number of columns.

What if you rotated the boxes around so that they looked like this? \begin{bmatrix}17 \\ 25 \\ 6 \\ 2 \end{bmatrix}⎣⎢⎢⎡​172562​⎦⎥⎥⎤​

A picture containing clock, building, drawing, window

Description automatically generated

### Two Vectors

What happens if you take a vector and duplicate the vector like this?

\begin{bmatrix} 17 &17 \\ 25 & 25 \\ 6 & 6 \\ 2 & 2 \end{bmatrix}⎣⎢⎢⎡​172562​172562​⎦⎥⎥⎤​

A close up of a clock

Description automatically generated

Now, you have 4 rows and 2 columns for a 4x2 matrix. What about duplicating the horizontal vector? \begin{bmatrix} 17 & 25 & 6 & 2 \\ 17 & 25 & 6 & 2 \end{bmatrix}[1717​2525​66​22​]

A picture containing clock, hanging, room

Description automatically generated

### Representing a Matrix in Python

A matrix is thus a two-dimensional grid with m rows and n columns. Take a look at this matrix, which is a little larger than the examples so far.

\begin{bmatrix} 17 & 25 & 6 & 2 & 16 \\ 6 & 1 & 8 & 4 & 22 \\ 17 & 8 & 54 & 15 & 65 \\ 11 & 25 & 68 & 9 & 2 \end{bmatrix}⎣⎢⎢⎡​1761711​251825​685468​24159​1622652​⎦⎥⎥⎤​

A picture containing outdoor, clock, dark, light

Description automatically generated

This matrix is 4x5; the matrix has 4 rows and 5 columns.

How would you represent a matrix like this in Python? Start with the top row \begin{bmatrix}17, 25, 6, 2, 16\end{bmatrix}[17,25,6,2,16​]. When looking at the top row alone, it looks like a vector. And in Python, you were using lists to represent vectors.

When representing matrices in Python, you can think of each row as a vector:

first\_row = [17, 25, 6, 2, 16]

What about the second row?

second\_row = [6, 1, 8, 4, 22]

And so on:

third\_row = [17, 8, 54, 15, 65]

fourth\_row = [11, 25, 68, 9, 2]

#### Representing All Rows with One Variable

You are representing each row with its own variable. The next step is to represent the entire matrix with only one variable.

If you list all of the rows one after another, you get a list of lists:

matrix = [first\_row, second\_row, third\_row, fourth\_row]

The "matrix-ness" is more explicit if you write it like this:

matrix = [

first\_row,

second\_row,

third\_row,

fourth\_row

]

Replacing the variables with the vectors gives you a big list of lists:

matrix = [[17, 25, 6, 2, 16],

[6, 1, 8, 4, 22],

[17, 8, 54, 15, 65],

[11, 25, 68, 9, 2]]

or, slightly cleaner:

matrix = [

[17, 25, 6, 2, 16],

[6, 1, 8, 4, 22],

[17, 8, 54, 15, 65],

[11, 25, 68, 9, 2]

]

# Matrix Addition

In order to use the Kalman Filter equations, you will need to do matrix addition.

For example, the equation for calculating the error covariance matrix after the prediction step includes matrix addition:

\mathbf{P\_{k|k-1}} = \mathbf{F\_{k}} \mathbf{P\_{k-1|k-1}} \mathbf{F\_{k}^T} + \mathbf{Q\_{k}}**Pk∣k**−**1**​=**Fk**​**Pk**−**1∣k**−**1**​**FkT**​+**Qk**​

or, using our simplified notation:

\mathbf{P}' = \mathbf{F} \mathbf{P} \mathbf{F^T} + \mathbf{Q}**P**′=**FPFT**+**Q**

### Matrix Addition General Formula

As a reminder, this is the general formula for carrying out matrix addition:

\mathbf{A} + \mathbf{B} = \begin{bmatrix} a\_{11} & a\_{12} & \ldots & a\_{1n}\\ a\_{21} & a\_{22} & \ldots &a\_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a\_{m1} & a\_{m2} & \ldots & a\_{mn} \end{bmatrix} + \begin{bmatrix} b\_{11} & b\_{12} & \ldots & b\_{1n}\\ b\_{21} & b\_{22} & \ldots &b\_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b\_{m1} & b\_{m2} & \ldots & b\_{mn} \end{bmatrix}**A**+**B**=⎣⎢⎢⎡​*a*11​*a*21​⋮*am*1​​*a*12​*a*22​⋮*am*2​​……⋱…​*a*1*n*​*a*2*n*​⋮*amn*​​⎦⎥⎥⎤​+⎣⎢⎢⎡​*b*11​*b*21​⋮*bm*1​​*b*12​*b*22​⋮*bm*2​​……⋱…​*b*1*n*​*b*2*n*​⋮*bmn*​​⎦⎥⎥⎤​

= \begin{bmatrix} a\_{11}+b\_{11} & a\_{12}+b\_{12} & \ldots & a\_{1n}+b\_{1n} \\ a\_{21}+b\_{21} & a\_{22}+b\_{22} & \ldots & a\_{2n}+b\_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a\_{m1}+b\_{m1} & a\_{m2}+b\_{m2} & \ldots & a\_{mn}+b\_{mn} \end{bmatrix}=⎣⎢⎢⎡​*a*11​+*b*11​*a*21​+*b*21​⋮*am*1​+*bm*1​​*a*12​+*b*12​*a*22​+*b*22​⋮*am*2​+*bm*2​​……⋱…​*a*1*n*​+*b*1*n*​*a*2*n*​+*b*2*n*​⋮*amn*​+*bmn*​​⎦⎥⎥⎤​

The first element of matrix A is added to the first element of matrix B. The second element gets added to the second element, etc.

### Matrix Addition Concrete Example

Here is a concrete example of matrix addition:

\mathbf{A} + \mathbf{B} = \begin{bmatrix} 17&25&6&2&16\\ 6&1&97&4&22\\ 80&8&54&15&65\\ 11&25&68&9&2 \end{bmatrix} + \begin{bmatrix} 3&14&1&7&42\\ 32&11&2&4&18\\ 19&81&4&8&5\\ 27&2&3&6&7 \end{bmatrix} = \begin{bmatrix} 20&39&7&9&58\\ 38&12&99&8&40\\ 99&89&58&23&70\\ 38&27&71&15&9 \end{bmatrix}**A**+**B**=⎣⎢⎢⎡​1768011​251825​6975468​24159​1622652​⎦⎥⎥⎤​+⎣⎢⎢⎡​3321927​1411812​1243​7486​421857​⎦⎥⎥⎤​=⎣⎢⎢⎡​20389938​39128927​7995871​982315​5840709​⎦⎥⎥⎤​

The graphic below shows how to calculate the sum. You take an element in matrix A and then add the matching element in matrix B that has the same position.

A screen shot of a computer keyboard

Description automatically generated

The top left element in matrix A is 17 and the top left element in matrix B is 3. The sum is 20, so the top left element of the resulting matrix is 20.

Here is another example: The second row third column A value is 97. The second row third column B value is 2. The sum is 99, so the second row third column resulting matrix has the value 99.

### Characteristics of Matrix Addition

You will notice an important characteristic about matrix addition: the size of matrix A and matrix B need to be the same; in other words, they need the same number of rows and the same number of columns. If you go back to the Kalman Filter prediction equation shown at the top of the page, \mathbf{P\_{k|k-1}} = \mathbf{F\_{k}} \mathbf{P\_{k-1|k-1}} \mathbf{F\_{k}^T} + \mathbf{Q\_{k}}**Pk∣k**−**1**​=**Fk**​**Pk**−**1∣k**−**1**​**FkT**​+**Qk**​,

this means that the matrix \mathbf{Q\_{k}}**Qk**​ must be the same size as the matrix that results from multiplying \mathbf{F\_{k}} \mathbf{P\_{k-1|k-1}} \mathbf{F\_{k}^T}**Fk**​**Pk**−**1∣k**−**1**​**FkT**​.

Furthermore, the sum of two matrices will have the same size as well. So \mathbf{P\_{k|k-1}} \text{as well as } \mathbf{Q\_{k}}**Pk∣k**−**1**​as well as **Qk**​ must have the same number of rows and columns.

### Matrix Subtraction

To subtract two matrices, the same rules apply. To find \mathbf{A} - \mathbf{B}**A**−**B**, you would subtract an element from B from its corresponding element in A.

### Coding Matrix Addition

Matrix addition involves adding elements from the same position. So for S = matrix A + matrix B, you would need to do operations like these:

S[0][0] = A[0][0] + B[0][0]

S[0][1] = A[0][1] + B[0][1]

S[0][2] = A[0][2] + B[0][2]

S[0][3] = A[0][3] + B[0][3]

.... etc

S[1][0] = A[1][0] + B[1][0]

S[1][1] = A[1][1] + B[1][1]

...etc.

However, this code isn't very efficient for a number of reasons. You would have to write a line of code for every element in the matrix. If a matrix had 5 rows and 4 columns, you would need to write 20 lines of code. Another problem is that you won't always know beforehand how large your matrices will be. Your code might need to accommodate summing 5x4 matrices. But then your might also need to sum 10x2 matrices and any number of other configurations.

This is a perfect place to use nested for loops.

### Using For Loops to Code Matrix Addition

In the previous exercises, you wrote code to do scalar multiplication. You also wrote a function that prints out a matrix.

The code for matrix addition is very similar. So try to do it on your own in the IPython notebook exercises on the next page.

# Matrix Multiplication

The Kalman filter equations have many matrix multiplication operations. Actually, every equation involves a matrix multiplication operation.

Matrix multiplication is different than matrix addition or subtraction. In matrix addition, you took an element from the first matrix, found the matching element in the second matrix, and outputted the sum.

You can multiply matching elements in a matrix as well, but that is called element-wise multiplication. Matrix multiplication is a different operation. And matrix multiplication is trickier to code.

Multiplication of Matrix \mathbf{A}**A** with matrix \mathbf{B}**B** is only possible if the number of columns in \mathbf{A}**A** is equal to the number of rows in \mathbf{B}**B**. So if matrix \mathbf{A}**A** is m \times n*m*×*n*, then \mathbf{B}**B** needs to be n \times p*n*×*p*. The values for m*m* and p*p* can be any positive integer.

The result of \mathbf{A} \times \mathbf{B}**A**×**B** is a matrix of size m \times p*m*×*p*.

### Formal Definition of Matrix Multiplication

Here is the formal equation for multiplying two matrices together:

(\mathbf{AB})\_{ij} = \sum\_{k=1}^n a\_{ik}b\_{kj}(**AB**)*ij*​=∑*k*=1*n*​*aik*​*bkj*​

All this is saying is that to find the element (i,j) in the resulting matrix, you need to

* take row i in matrix A, column j in matrix B
* do element-wise multiplication on the i-row A vector and j-column B vector
* sum the resulting elements

But you have already done element-wise multiplication and then summed the resulting elements of two vectors. That was the definition of the dot product in the vectors part of the lesson! Think of a matrix row as a vector and a matrix column also as a vector; you already wrote code for calculating the dot product.

So you are already part way done with coding matrix multiplication!

Let's go through a concrete example to see how matrix multiplication works. The two matrices being multiplied are:

\begin{bmatrix} 17 & 25 & 6 & 2 \\ 6 & 1 & 97 & 4 \\ 80 & 8 & 54 & 15 \end{bmatrix} \times \begin{bmatrix} 3 & 14 & 1 & 7 & 42 & 5 \\ 32 & 11 & 2 & 4 & 18 & 17 \\ 19 & 81 & 4 & 8 & 5 & 10 \\ 27 & 2 & 3 & 6 & 7 & 3 \end{bmatrix} = \begin{bmatrix} 1019 & 1003 & 97 & 279 & 1208 & 576 \\ 2001 & 7960 & 408 & 846 & 783 & 1029 \\ 1927 & 5612 & 357 & 1114 & 3879 & 1121 \end{bmatrix}⎣⎡​17680​2518​69754​2415​⎦⎤​×⎣⎢⎢⎡​3321927​1411812​1243​7486​421857​517103​⎦⎥⎥⎤​=⎣⎡​101920011927​100379605612​97408357​2798461114​12087833879​57610291121​⎦⎤​

Note that matrix \mathbf{A}**A** has three rows and four columns (3 x 4) and matrix \mathbf{B}**B** has four rows and six columns (4 x 6). So the output is a 3 x 6 matrix.

Where did the number 1019 come from in row one, column one? According to the formula for matrix multiplication, it came from element-wise multiplication of the first row of matrix \mathbf{A}**A** with the first column of matrix \mathbf{B}**B**.

Thus you calculate the dot product of row one of A and column one of B. The dot product involves element-wise multiplication and then summing the results:

\begin{bmatrix} 17 & 25 & 6 & 2 \end{bmatrix} \cdot \begin{bmatrix} 3 & 32 & 19 & 27 \end{bmatrix}[17​25​6​2​]⋅[3​32​19​27​]

= 17 \times 3 + 25 \times 32 + 6 \times 19 + 2 \times 27=17×3+25×32+6×19+2×27 = 1019=1019

See the illustration below to see what is happening in more detail:

A picture containing crossword, dark, clock, light

Description automatically generated

How would you calculate the value in row one, column two?

You can see in the visualization below how to calculate the value in row one, column two.

A picture containing crossword, dark, clock, light

Description automatically generated

Jumping ahead, here is how you would calculate row two, column four. You would need the second row of matrix \mathbf{A}**A** and the fourth column of \mathbf{B}**B**.

A screen shot of a keyboard

Description automatically generated

Continuing with the process, you end up with \begin{bmatrix} 1019 & 1003 & 97 & 279 & 1208 & 576 \\ 2001 & 7960 & 408 & 846 & 783 & 1029 \\ 1927 & 5612 & 357 & 1114 & 3879 & 1121 \end{bmatrix}⎣⎡​101920011927​100379605612​97408357​2798461114​12087833879​57610291121​⎦⎤​

Now it's your turn to code matrix multiplication in Python.

**Coding Matrix Multiplication**

Writing code for matrix multiplication can be quite tricky. So you are going to build the solution in pieces. The good news is that coding matrix multiplication is a big part of this module's final project.

Think about what matrix multiplication involves. You will multiply two matrices with sizes

* m x n
* n x p

and output a matrix of size m x p.

In previous exercises, you already iterated through matrices using nested for loops. How could you use a nested for loop to calculate each element in the m x p matrix? And then once you have grabbed the necessary row in A and the necessary column in B, how will you combine the values to get the right answer?

The next page has an Ipython notebook where you can write your code.

# Transpose of a Matrix

There were a few Kalman filter equations that required the transpose of a matrix. You can identify these matrices because they have a T superscript. For example the transpose of matrix \mathbf{A}**A** is written \mathbf{A^T}**AT**

These were the three equations that contained the transpose of a matrix:

\mathbf{P\_{k|k-1}} = \mathbf{F\_{k}} \mathbf{P\_{k-1|k-1}} \mathbf{F\_{k}^T} + \mathbf{Q\_{k}}**Pk∣k**−**1**​=**Fk**​**Pk**−**1∣k**−**1**​**FkT**​+**Qk**​

\mathbf{S\_{k}} = \mathbf{H\_{k}} \mathbf{P\_{k|k-1}} \mathbf{H\_{k}^T} + \mathbf{R\_{k}}**Sk**​=**Hk**​**Pk∣k**−**1**​**HkT**​+**Rk**​

\mathbf{K\_{k}} = \mathbf{P\_{k|k-1}} \mathbf{H\_{k}^T} \mathbf{S\_{k}}^{-1}**Kk**​=**Pk∣k**−**1**​**HkT**​**Sk**​−1

### What exactly is the transpose?

You can think of the tranpose as switching rows and columns. The matrix rows become the columns or alternatively you can consider the columns become the rows.

Here is an example. If you start with this matrix,

\begin{bmatrix} 3 & 25 & 9 & 2 & 4 \\ 7 & 15 & 6 & 92 & 17 \\31 & 18 & 0 & 11 & 8 \end{bmatrix}⎣⎡​3731​251518​960​29211​4178​⎦⎤​

the transpose would be

\begin{bmatrix} 3 & 7 & 31 \\ 25 & 15 & 18 \\ 9 & 6 & 0 \\ 2 & 92 & 11 \\ 4 & 17 & 8 \end{bmatrix}⎣⎢⎢⎢⎢⎡​325924​71569217​31180118​⎦⎥⎥⎥⎥⎤​

The original matrix was size 3x5. The transpose is 5x3.

To get a better understanding of what is happening, this image is color coded to match values from the original matrix and the transpose of the matrix. If you think of switching the rows and making them into columns, the matrix operation looks like this:

A close up of a screen

Description automatically generated

But you could also think of transposing the columns into rows:

A screen shot of a computer

Description automatically generated

Mathematically, you are switching around the i and j values for every element in the matrix. For example, the element in the 3rd row, 4th column is 11. For the transpose of the matrix, 11 is now in the 4th row, 3rd column. The formal mathematical definition of the transpose of a matrix is

[\mathbf{A^T}]\_{ij} = [\mathbf{A}]\_{ji}[**AT**]*ij*​=[**A**]*ji*​

### Motivation for Calculating the Transpose

In order to use the Kalman filter equations, you need to calculate the transpose of both the \mathbf{F} \text{ and } \mathbf{H}**F** and **H** matrices.

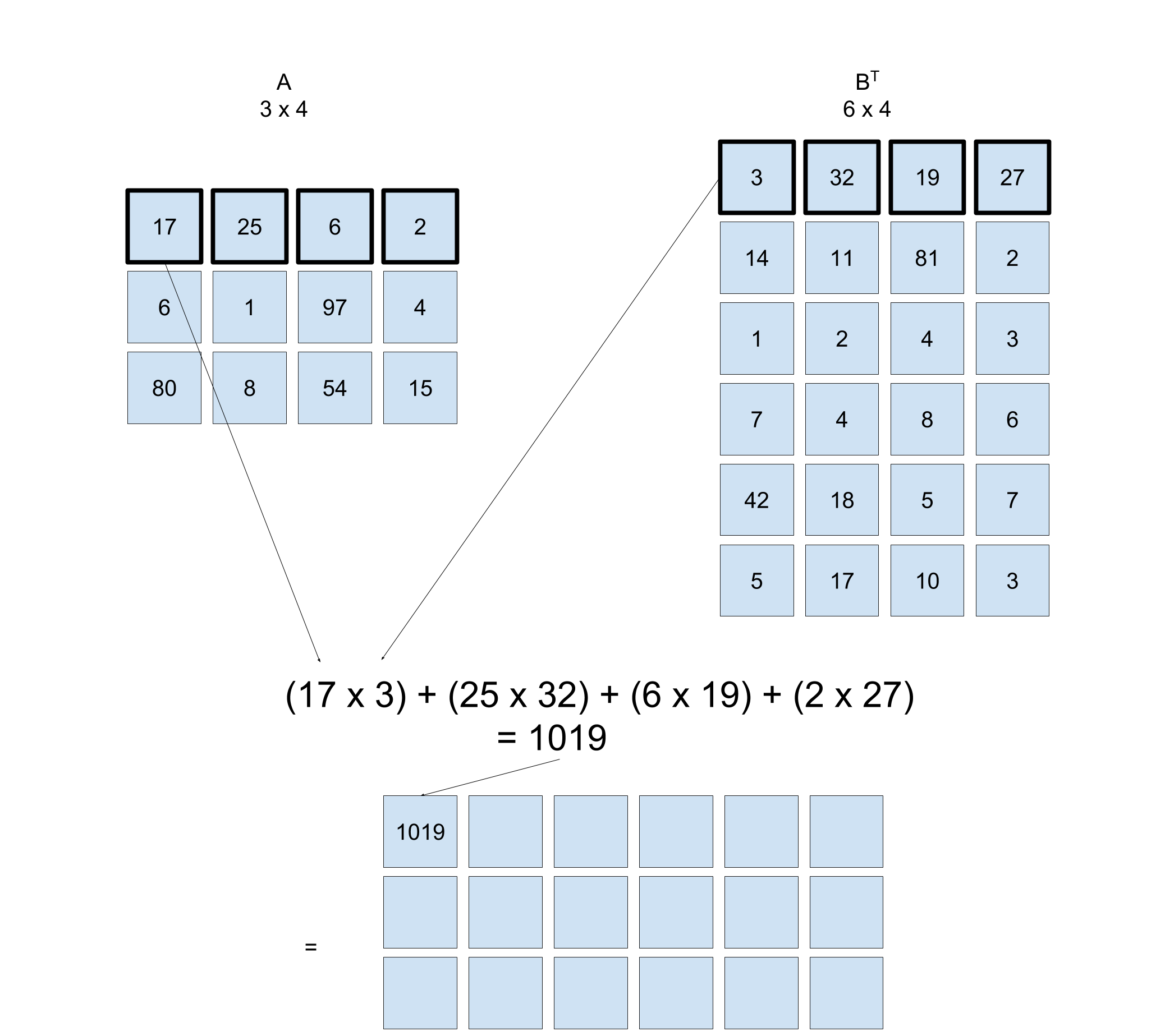
But there is also another place where you could use a matrix transposition: when carrying out matrix multiplication. In the previous exercises for matrix multiplication, you wrote a function that returned a column of a matrix. You needed matrix columns in order to find the dot product of a row from matrix \mathbf{A}**A** and a column from matrix \mathbf{B}**B**.

Here is a reminder of what that looked like:

A picture containing crossword, dark, clock, light

Description automatically generated

But what happens if you take the transpose of matrix \mathbf{B}**B**? All of the columns in \mathbf{B}**B** become rows. Your matrix multiplication function then involves finding the dot product between rows of \mathbf{A}**A** and rows of \mathbf{B^T}**BT**:



Dot product between rows of A and transpose of matrix B

You are not calculating the product of AB^T*ABT*. Instead, you are taking advantage of matrix transposition to make matrix multiplication easier to code.

In the previous coding exercises, the get\_column function you built to change a matrix column into a horizontal vector was essentially a transpose.

In the coding exercises for this part of the lesson, you will not only write a transpose function but also write a new multiplication function that takes advantage of the matrix transpose.

### Coding the Transpose of a Matrix

This is a similar problem to what you have already seen; you'll need to use nested for loops. But what exactly will this nested for loop look like?

You could iterate through the matrix like you've done already with the rows in the outer loop and the columns in the inner loop:

**for** i **in** range(len(matrixA)):

**for** j **in** range(len(matrixA[0])):

print(matrixA[i][j])

The transpose of the matrix would need to store each i, j element inside a new matrix with position j, i. But how are you going to populate this new matrix? You would probably first need to create an n x m list within a list and populate this nested list with empty values. That sounds complicated.

Is there a more efficient way to code matrix transposition? Think about how you could start iterating through the columns in the outside for loop and the rows on the inside for loop.

Go to the next part of the lesson to write your code.

# The Identity Matrix

The identity matrix is a special matrix in linear algebra that shows up in quite a few applications. For the purposes of this lesson, gaining insight into the identity matrix will help you understand matrix inversion. The identity matrix is represented by the symbol \mathbf{I}**I**.

\mathbf{I}**I** is an n x n square matrix with 1 across the main diagonal and 0 for all other elements.

For a 1x1 matrix, the identity matrix looks like this:

\begin{bmatrix} 1 \end{bmatrix}[1​]

A 2x2 identity matrix looks like this:

\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}[10​01​]

The 3x3 identity matrix is the following:

\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}⎣⎡​100​010​001​⎦⎤​

The 4x4 identity matrix is:

\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}⎣⎢⎢⎡​1000​0100​0010​0001​⎦⎥⎥⎤​

and so on.

# Identity Matrix is like the Number One

In scalar multiplication, the number one has a special property: 1\times a = a1×*a*=*a*.

Likewise, a\times 1 = a*a*×1=*a*.

It turns out the Identity matrix has the same property: \mathbf{AI} = \mathbf{IA} = \mathbf{A}**AI**=**IA**=**A**. And although the identity matrix is always square, matrix \mathbf{A}**A** does not have to be square.

Here is an example of multiplying \mathbf{AI}**AI**

A screen shot of a keyboard

Description automatically generated

What about the other case of multiplying \mathbf{IA}**IA**? You'll need to take into account the dimensions of the I and A matrix so that the multiplication works.

How do you figure out the size of \mathbf{I}**I**?

The output matrix has to be 3x4 just like \mathbf{A}**A**. So a 3x3 matrix multiplied by a 3x4 matrix will give a 3x4 matrix.

A picture containing crossword, electronics, dark, keyboard

Description automatically generated

# Initializing an Identity Matrix with Code

In the coding exercise, you will write a function that receives a size and outputs the identity matrix for that size.

Think about how you might go about coding this starting from an empty Python list. The ones will always be at indicies

* [0][0]
* [1][1]
* [2][2]
* [3][3]
* etc.

Everywhere else in the matrix will be zero. So you will need to not only use nested for loops but also if else statements.

# Matrix Inverse

There is one more matrix operation that you will need in order to use the Kalman Filter equations: the inverse of a matrix.

Specifically when calculating the Kalman filter gain matrix \mathbf{K}**K**, you will need to take the inverse of the \mathbf{S}**S** matrix. The superscript ^{-1}−1 represents the inverse of a matrix. Here is a reminder of the Kalman Filter gain equations where you can see the need for the inverse of S.

\mathbf{S\_{k}} = \mathbf{H\_{k}} \mathbf{P\_{k|k-1}} \mathbf{H\_{k}^T} + \mathbf{R\_{k}}**Sk**​=**Hk**​**Pk∣k**−**1**​**HkT**​+**Rk**​

\mathbf{K\_{k}} = \mathbf{P\_{k|k-1}} \mathbf{H\_{k}^T} \mathbf{S\_{k}}^{-1}**Kk**​=**Pk∣k**−**1**​**HkT**​**Sk**​−1

### Formal Definition of Inverse of a Matrix

As mentioned, the inverse of a matrix \mathbf{A}**A** would be denoted by \mathbf{A^{-1}}**A**−**1**.

Formally, if matrix \mathbf{A}**A** has an inverse, then

\mathbf{A} \times \mathbf{A^{-1}}**A**×**A**−**1** = \mathbf{A^{-1}} \times \mathbf{A}**A**−**1**×**A** = \mathbf{I}**I**

where \mathbf{I}**I** is an identity matrix.

Only square matrices, or in other words matrices with the same number of columns as rows, have inverses. You can see that this must be true based on the definition of the inverse and the identity matrix. The identity matrix is always a square matrix, so

if \mathbf{A}**A** is m x n, then \mathbf{A^{-1}}**A**−**1** has to be n x m to get a square identity matrix of m x m.

Multiplying \mathbf{A^{-1}} \mathbf{A}**A**−**1A** gives (n x m)(n x m) = n x m, which is not a square matrix unless n = m.

So in order for a matrix to have an inverse, the matrix must be square. At the same time, not all square matrices have inverses.

### Relationship between a Scalar Inverse and a Matrix Inverse

In scalar math, the inverse of a number x*x* is 1/x1/*x*.

If you multiply a scalar by its inverse, you get 1:

x \times \frac{1}{x} = 1*x*×*x*1​=1

In linear algebra, the inverse of a matrix is analogous to the scalar inverse:

\mathbf{A} \times \mathbf{A^{-1}} = \mathbf{I}**A**×**A**−**1**=**I**

As you saw in the previous part of the lesson, the identity matrix has a similar role to the number 1.

### Calculating the Inverse of a Matrix

For the purposes of this class, you will learn to calculate the inverse of a 1x1 matrix and a 2x2 matrix.

For matrices with more dimensions, the [**calculations become more complicated**](https://en.wikipedia.org/wiki/Invertible_matrix). Both Python and C++ have libraries that can calculate the inverse of a matrix such as the [**NumPy Library**](http://www.numpy.org/) and the [**Eigen Library**](http://eigen.tuxfamily.org/index.php?title=Main_Page).

#### Inverse of a 1 x 1 matrix

For a 1 \times 11×1 matrix with a single element with value a*a*, the inverse is simply \frac{1}{a}*a*1​.

So the inverse of

\begin{bmatrix} 25 \end{bmatrix}[25​] is

\begin{bmatrix} \frac{1}{25} \end{bmatrix}[251​​].

#### Inverse of a 2 x 2 matrix

Say you have a matrix

\begin{bmatrix} a & b \\ c & d \end{bmatrix}[*ac*​*bd*​]

The inverse of this 2 x 2 matrix is

\frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}*ad*−*bc*1​[*d*−*c*​−*ba*​]

And you can see that if ad = bc*ad*=*bc*, then the matrix does not have an inverse.

##### Another formula for 2x2 inverse matrix

Here is a more formal formula for the 2x2 inverse matrix.

\mathbf{A}^{-1} = \frac{1}{\text{det }\mathbf{A}} \left[\left(\text{tr } \mathbf{A}\right) \mathbf{I} - \mathbf{A}\right]**A**−1=det **A**1​[(tr **A**)**I**−**A**]

where

\text{det }\mathbf{A}det **A** is called the determinant of a matrix. For a 2x2 matrix, the determinant is ad - bc.

\text{tr } \mathbf{A}tr **A** is called the trace of a matrix. The trace is the sum across the major diagonal, which in this case would be a + d.

If you multiply everything, you end up with the same equation already presented, namely:

\mathbf{A^{-1}} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}**A**−**1**=*ad*−*bc*1​[*d*−*c*​−*ba*​]

### Inverse of a 3 x 3 or larger matrix

Calculating the inverse of a larger matrix involves relatively complex matrix algebra theory. In this course, you will only need to calculate the inverse of a 2 x 2 matrix.

### Coding the Inverse of a Matrix

You are going to write a function that calculates the inverse of a matrix. Remember that you will need to check the size of the matrix because a 1x1 matrix and a 2x2 matrix have different formulas.