03-05-Transformations Eigenvectors and Eigenvalues

September 2, 2018

1 Transformations, Eigenvectors, and Eigenvalues

Matrices and vectors are used together to manipulate spatial dimensions. This has a lot of applications, including the mathematical generation of 3D computer graphics, geometric modeling, and the training and optimization of machine learning algorithms. We're not going to cover the subject exhaustively here; but we'll focus on a few key concepts that are useful to know when you plan to work with machine learning.

1.1 Linear Transformations

You can manipulate a vector by multiplying it with a matrix. The matrix acts a function that operates on an input vector to produce a vector output. Specifically, matrix multiplications of vectors are *linear transformations* that transform the input vector into the output vector.

For example, consider this matrix ***A*** and vector ***v***:

$$A = \begin{bmatrix}2 & 3\\5 & 2\end{bmatrix} \quad \vec{v} = \begin{bmatrix}1\\2\end{bmatrix}$$

We can define a transformation ***T*** like this:

$$T(\vec{v}) = A\vec{v}$$

To perform this transformation, we simply calculate the dot product by applying the *RC* rule; multiplying each row of the matrix by the single column of the vector:

$$\begin{bmatrix}2 & 3\\5 & 2\end{bmatrix} \cdot \begin{bmatrix}1\\2\end{bmatrix} = \begin{bmatrix}8\\9\end{bmatrix}$$

Here's the calculation in Python:

```
In [1]: import numpy as np

        v = np.array([1,2])
        A = np.array([[2,3],
                      [5,2]])

        t = A@v
        print (t)
```

```
[8 9]
```

In this case, both the input vector and the output vector have 2 components - in other words, the transformation takes a 2-dimensional vector and produces a new 2-dimensional vector; which we can indicate like this:

$$T : \mathbb{R}^2 \to \mathbb{R}^2$$

Note that the output vector may have a different number of dimensions from the input vector; so the matrix function might transform the vector from one space to another - or in notation, ℝn -> ℝm.

For example, let's redefine matrix ***A***, while retaining our original definition of vector ***v***:

$$A = \begin{bmatrix} 2 & 3 \\ 5 & 2 \\ 1 & 1 \end{bmatrix} \quad \vec{v} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Now if we once again define ***T*** like this:

$$T(\vec{v}) = A\vec{v}$$

We apply the transformation like this:

$$\begin{bmatrix} 2 & 3 \\ 5 & 2 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 8 \\ 9 \\ 3 \end{bmatrix}$$

So now, our transformation transforms the vector from 2-dimensional space to 3-dimensional space:

$$T : \mathbb{R}^2 \to \mathbb{R}^3$$

Here it is in Python:

```
In [2]: import numpy as np
        v = np.array([1,2])
        A = np.array([[2,3],
                      [5,2],
                      [1,1]])

        t = A@v
        print (t)
```

```
[8 9 3]
```

```
In [3]: import numpy as np
        v = np.array([1,2])
        A = np.array([[1,2],
                      [2,1]])
        t = A@v
        print (t)
```

```
[5 4]
```

1.2 Transformations of Magnitude and Amplitude

When you multiply a vector by a matrix, you transform it in at least one of the following two ways: * Scale the length (*magnitude*) of the matrix to make it longer or shorter * Change the direction (*amplitude*) of the matrix

For example consider the following matrix and vector:

$$A = \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix} \quad \vec{v} = \begin{bmatrix}1\\0\end{bmatrix}$$

As before, we transform the vector $\boldsymbol{v}$ by multiplying it with the matrix $\boldsymbol{A}$:

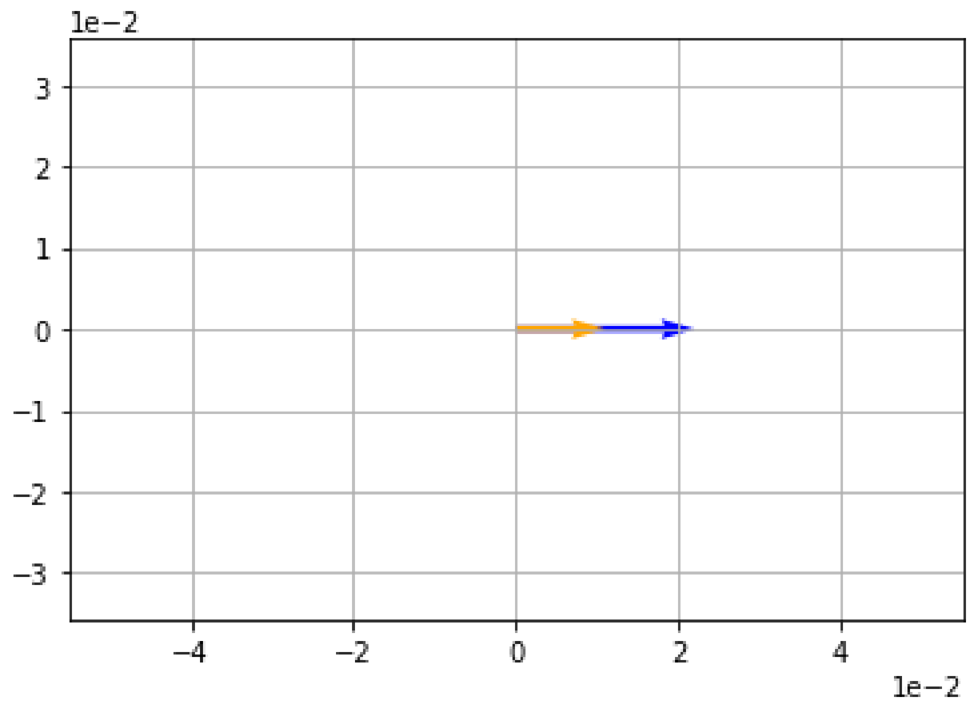
$$\begin{bmatrix}2 & 0\\0 & 2\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix} \tag{1}$$

In this case, the resulting vector has changed in length (*magnitude*), but has not changed its direction (*amplitude*).

Let's visualize that in Python:

```
In [4]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        v = np.array([1,0])
        A = np.array([[2,0],
                      [0,2]])

        t = A@v
        print (t)

        # Plot v and t
        vecs = np.array([t,v])
        origin = [0], [0]
        plt.axis('equal')
        plt.grid()
        plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
        plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
        plt.show()
```

```
[2 0]
```



The original vector *v* is shown in orange, and the transformed vector *t* is shown in blue - note that *t* has the same direction (*amplitude*) as *v* but a greater length (*magnitude*).

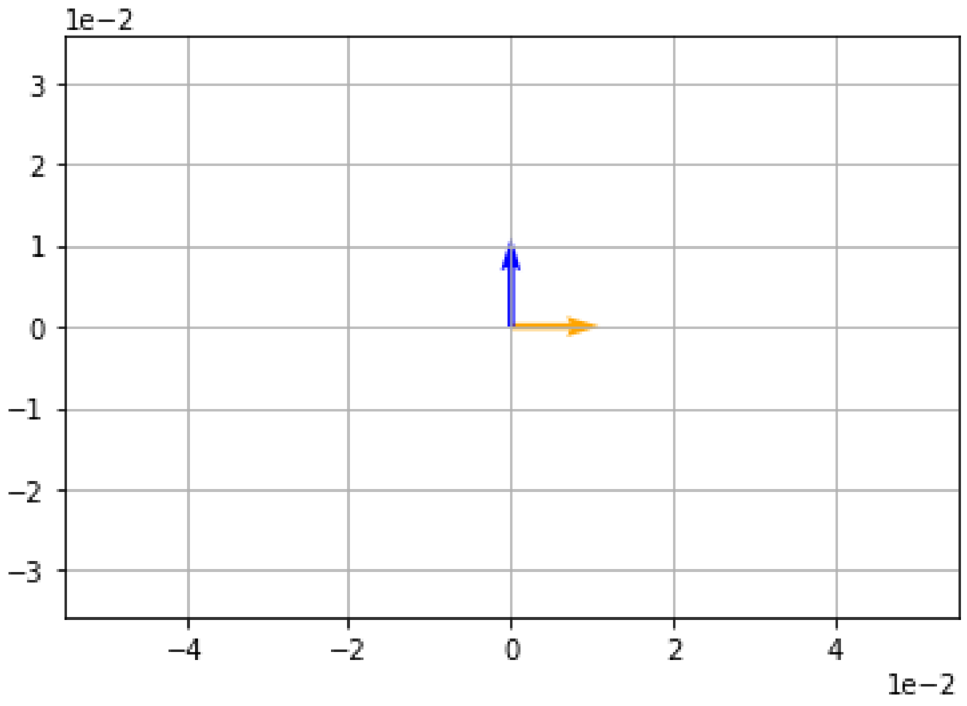
Now let's use a different matrix to transform the vector *v*:

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2}$$

This time, the resulting vector has been changed to a different amplitude, but has the same magnitude.

```
In [5]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        v = np.array([1,0])
        A = np.array([[0,-1],
                      [1,0]])

        t = A@v
        print (t)

        # Plot v and t
        vecs = np.array([v,t])
        origin = [0], [0]
        plt.axis('equal')
        plt.grid()
        plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
        plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'blue'], scale=10)
        plt.show()
```

```
[0 1]
```



Now let's see change the matrix one more time:

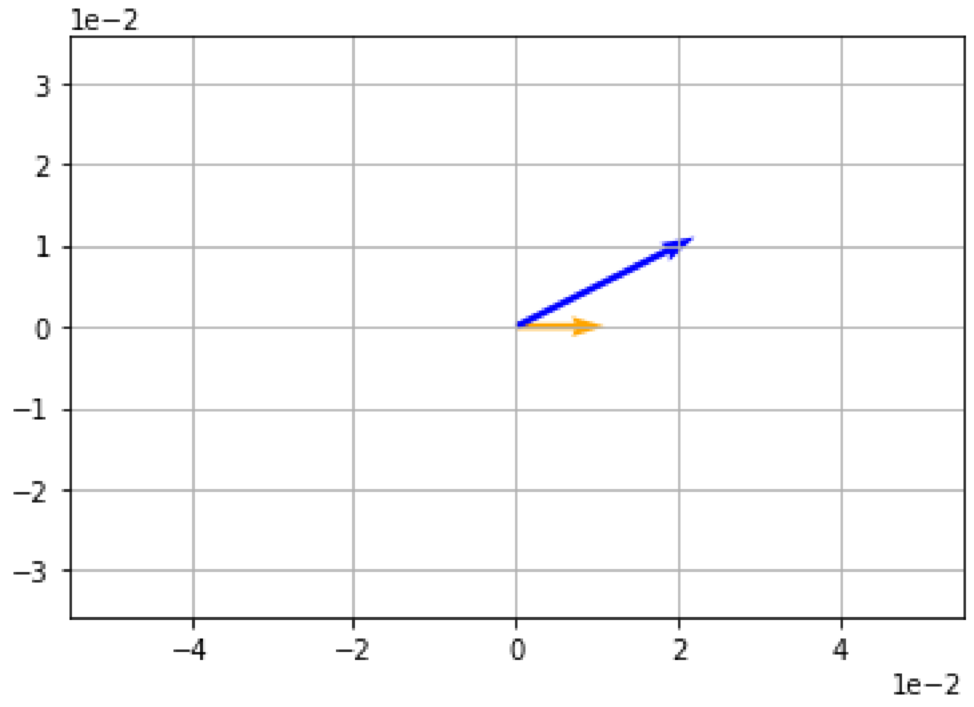
$$\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \tag{3}$$

Now our resulting vector has been transformed to a new amplitude *and* magnitude - the transformation has affected both direction and scale.

```
In [6]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        v = np.array([1,0])
        A = np.array([[2,1],
                      [1,2]])

        t = A@v
        print (t)
```

```
# Plot v and t
vecs = np.array([v,t])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'blue'], scale=10)
plt.show()
```

```
[2 1]
```



1.2.1 Afine Transformations

An Afine transformation multiplies a vector by a matrix and adds an offset vector, sometimes referred to as *bias*; like this:

$$T(\vec{v}) = A\vec{v} + \vec{b}$$

For example:

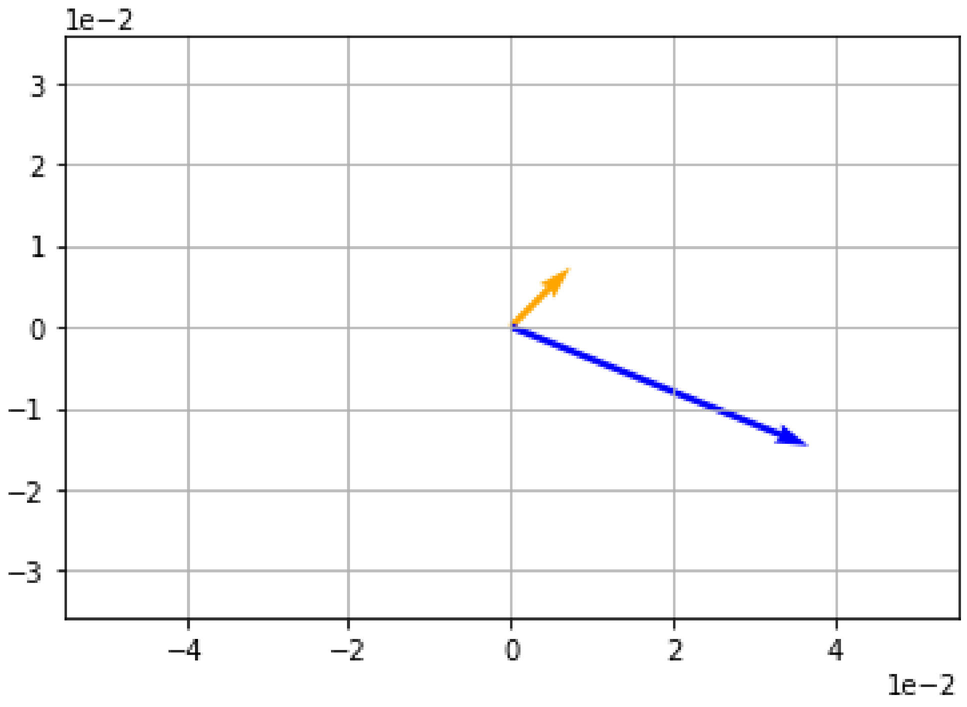
$$\begin{bmatrix} 5 & 2 \\ 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -2 \\ -6 \end{bmatrix} = \begin{bmatrix} 5 \\ -2 \end{bmatrix} \tag{4}$$

This kind of transformation is actually the basis of linear regression, which is a core foundation for machine learning. The matrix defines the *features*, the first vector is the *coefficients*, and the bias vector is the *intercept*.

here's an example of an Afine transformation in Python:

```
In [7]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        v = np.array([1,1])
        A = np.array([[5,2],
                      [3,1]])
        b = np.array([-2,-6])

        t = A@v + b
        print (t)

        # Plot v and t
        vecs = np.array([v,t])
        origin = [0], [0]
        plt.axis('equal')
        plt.grid()
        plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
        plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'blue'], scale=15)
        plt.show()
```

```
[ 5 -2]
```



1.3 Eigenvectors and Eigenvalues

So we can see that when you transform a vector using a matrix, we change its direction, length, or both. When the transformation only affects scale (in other words, the output vector has a different magnitude but the same amplitude as the input vector), the matrix multiplication for the transformation is the equivalent operation as some scalar multiplication of the vector.

For example, earlier we examined the following transformation that dot-mulitplies a vector by a matrix:

$$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

You can achieve the same result by mulitplying the vector by the scalar value **2**:

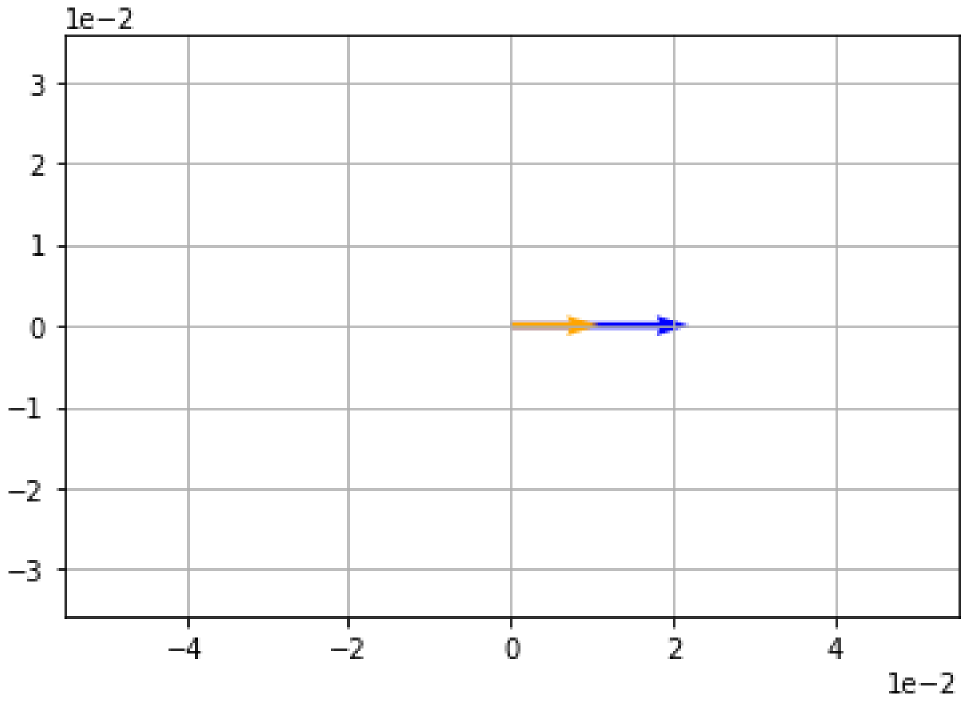
$$2 \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

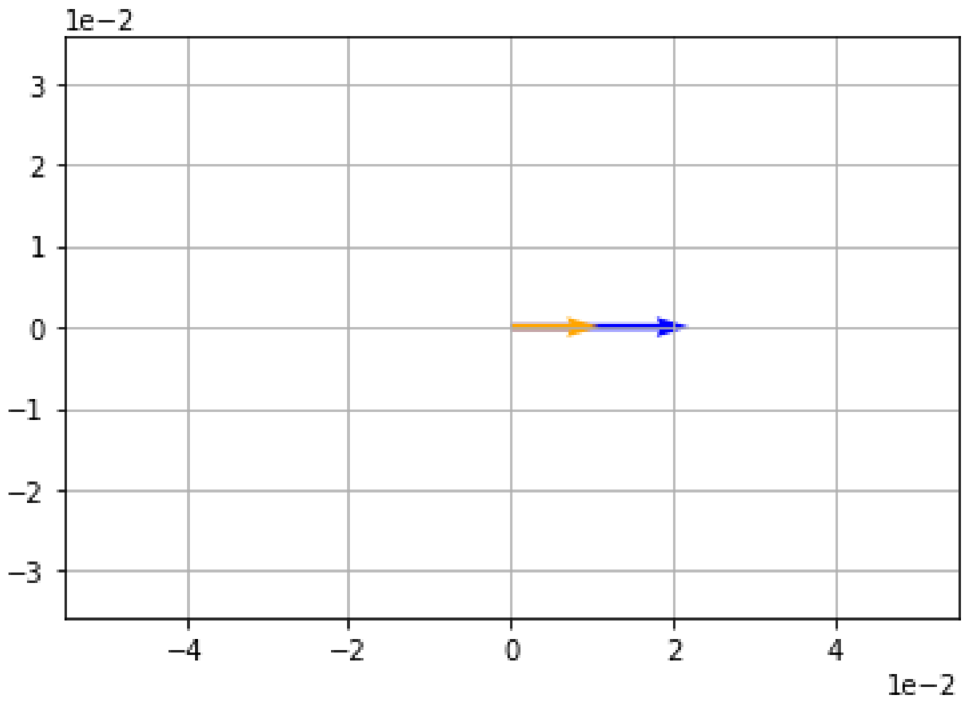
The following python performs both of these calculation and shows the results, which are identical.

```
In [8]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        v = np.array([1,0])
        A = np.array([[2,0],
                      [0,2]])


        t1 = A@v
        print (t1)
        t2 = 2*v
        print (t2)

        fig = plt.figure()
        a=fig.add_subplot(1,1,1)
        # Plot v and t1
        vecs = np.array([t1,v])
        origin = [0], [0]
        plt.axis('equal')
        plt.grid()
        plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
        plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
        plt.show()
        a=fig.add_subplot(1,2,1)
        # Plot v and t2
        vecs = np.array([t2,v])
        origin = [0], [0]
        plt.axis('equal')
        plt.grid()
        plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
```

```
    plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
    plt.show()
```

```
[2 0]
[2 0]
```





In cases like these, where a matrix transformation is the equivelent of a scalar-vector multiplication, the scalar-vector pairs that correspond to the matrix are known respectively as eigenvalues and eigenvectors. We generally indicate eigenvalues using the Greek letter lambda (), and the formula that defines eigenvalues and eigenvectors with respect to a transformation is:

$$T(\vec{v}) = \lambda\vec{v}$$

Where the vector ***v*** is an eigenvector and the value is an eigenvalue for transformation ***T***.

When the transformation ***T*** is represented as a matrix multiplication, as in this case where the transformation is represented by matrix ***A***:

$$T(\vec{v}) = A\vec{v} = \lambda\vec{v}$$

Then ***v*** is an eigenvector and is an eigenvalue of ***A***.

A matrix can have multiple eigenvector-eigenvalue pairs, and you can calculate them manually. However, it's generally easier to use a tool or programming language. For example, in Python you can use the ***linalg.eig*** function, which returns an array of eigenvalues and a matrix of the corresponding eigenvectors for the specified matrix.

Here's an example that returns the eigenvalue and eigenvector pairs for the following matrix:

$$A = \begin{bmatrix}2 & 0\\0 & 3\end{bmatrix}$$

```
In [9]: import numpy as np
        A = np.array([[2,0],
                      [0,3]])
        eVals, eVecs = np.linalg.eig(A)
        print(eVals)
        print(eVecs)
```

```
[2. 3.]
[[1. 0.]
 [0. 1.]]
```

So there are two eigenvalue-eigenvector pairs for this matrix, as shown here:

$$\lambda_1 = 2, \vec{v}_1 = \begin{bmatrix}1\\0\end{bmatrix} \qquad \lambda_2 = 3, \vec{v}_2 = \begin{bmatrix}0\\1\end{bmatrix}$$

Let's verify that multiplying each eigenvalue-eigenvector pair corresponds to the dot-product of the eigenvector and the matrix. Here's the first pair:

$$2 \times \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix} \quad and \quad \begin{bmatrix}2 & 0\\0 & 3\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix}$$

So far so good. Now let's check the second pair:

$$3 \times \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\3\end{bmatrix} \quad and \quad \begin{bmatrix}2 & 0\\0 & 3\end{bmatrix} \cdot \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\3\end{bmatrix}$$

So our eigenvalue-eigenvector scalar multiplications do indeed correspond to our matrix-eigenvector dot-product transformations.

Here's the equivalent code in Python, using the ***eVals*** and ***eVecs*** variables you generated in the previous code cell:

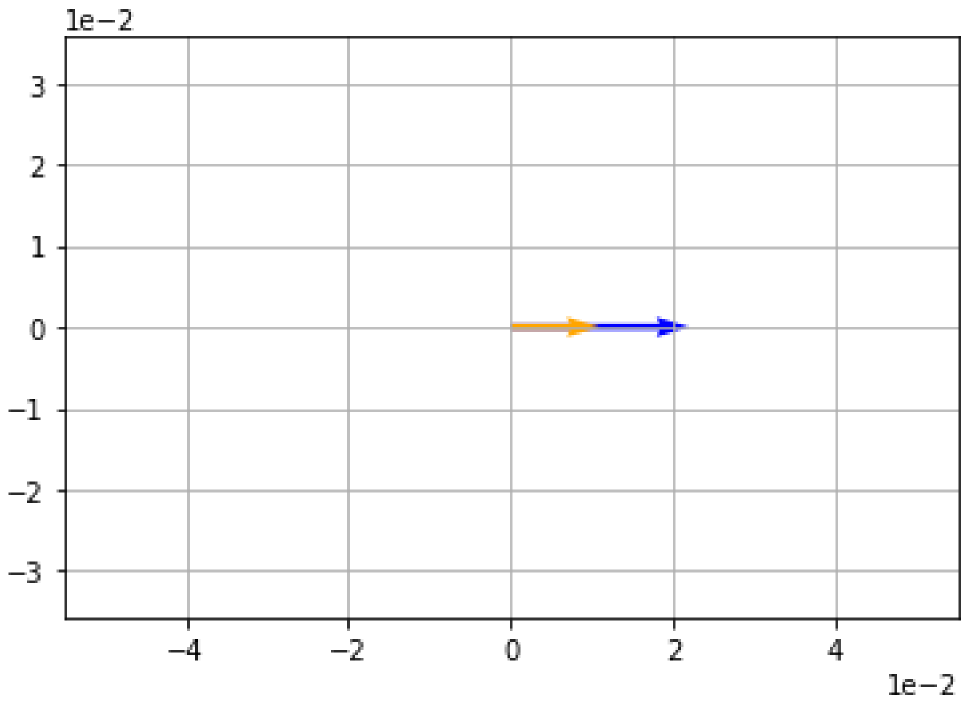
```
In [10]: vec1 = eVecs[:,0]
         lam1 = eVals[0]

         print('Matrix A:')
         print(A)
         print('-------')

         print('lam1: ' + str(lam1))
         print ('v1: ' + str(vec1))
         print ('Av1: ' + str(A@vec1))
         print ('lam1 x v1: ' + str(lam1*vec1))

         print('-------')

         vec2 = eVecs[:,1]
         lam2 = eVals[1]

         print('lam2: ' + str(lam2))
         print ('v2: ' + str(vec2))
         print ('Av2: ' + str(A@vec2))
         print ('lam2 x v2: ' + str(lam2*vec2))
```

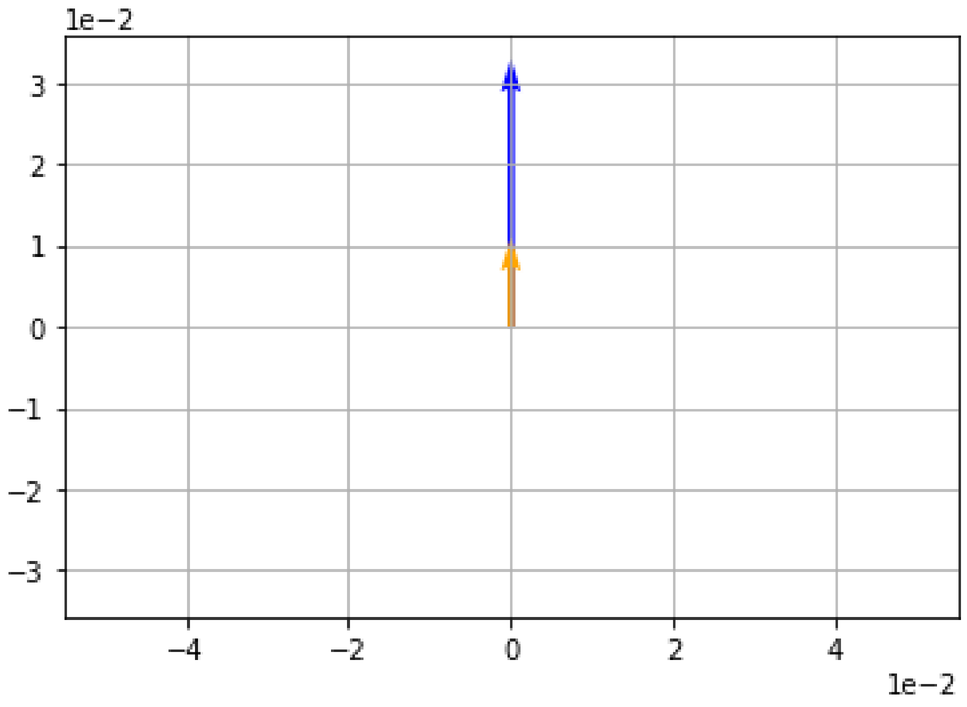
```
Matrix A:
[[2 0]
 [0 3]]
-------
lam1: 2.0
v1: [1. 0.]
Av1: [2. 0.]
lam1 x v1: [2. 0.]
-------
lam2: 3.0
v2: [0. 1.]
Av2: [0. 3.]
lam2 x v2: [0. 3.]
```

You can use the following code to visualize these transformations:

```
In [11]: t1 = lam1*vec1
         print (t1)
         t2 = lam2*vec2
         print (t2)

         fig = plt.figure()
         a=fig.add_subplot(1,1,1)
         # Plot v and t1
         vecs = np.array([t1,vec1])
         origin = [0], [0]
         plt.axis('equal')
         plt.grid()
         plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
         plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
         plt.show()
         a=fig.add_subplot(1,2,1)
         # Plot v and t2
         vecs = np.array([t2,vec2])
         origin = [0], [0]
         plt.axis('equal')
         plt.grid()
         plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
         plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
         plt.show()
```

```
[2. 0.]
[0. 3.]
```





Similarly, earlier we examined the following matrix transformation:

$$\begin{bmatrix}2 & 0\\0 & 2\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix}$$

And we saw that you can achieve the same result by mulitplying the vector by the scalar value 2:

$$2 \times \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix}$$

This works because the scalar value 2 and the vector (1,0) are an eigenvalue-eigenvector pair for this matrix.

Let's use Python to determine the eigenvalue-eigenvector pairs for this matrix:

```
In [12]: import numpy as np
         A = np.array([[2,0],
                       [0,2]])
         eVals, eVecs = np.linalg.eig(A)
         print(eVals)
         print(eVecs)
```

```
[2. 2.]
[[1. 0.]
 [0. 1.]]
```

So once again, there are two eigenvalue-eigenvector pairs for this matrix, as shown here:

$$\lambda_1 = 2, \vec{v_1} = \begin{bmatrix}1\\0\end{bmatrix} \qquad \lambda_2 = 2, \vec{v_2} = \begin{bmatrix}0\\1\end{bmatrix}$$

Let's verify that multiplying each eigenvalue-eigenvector pair corresponds to the dot-product of the eigenvector and the matrix. Here's the first pair:

$$2 \times \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix} \quad \textit{and} \quad \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix}$$

Well, we already knew that. Now let's check the second pair:

$$2 \times \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\2\end{bmatrix} \quad \textit{and} \quad \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix} \cdot \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\2\end{bmatrix}$$

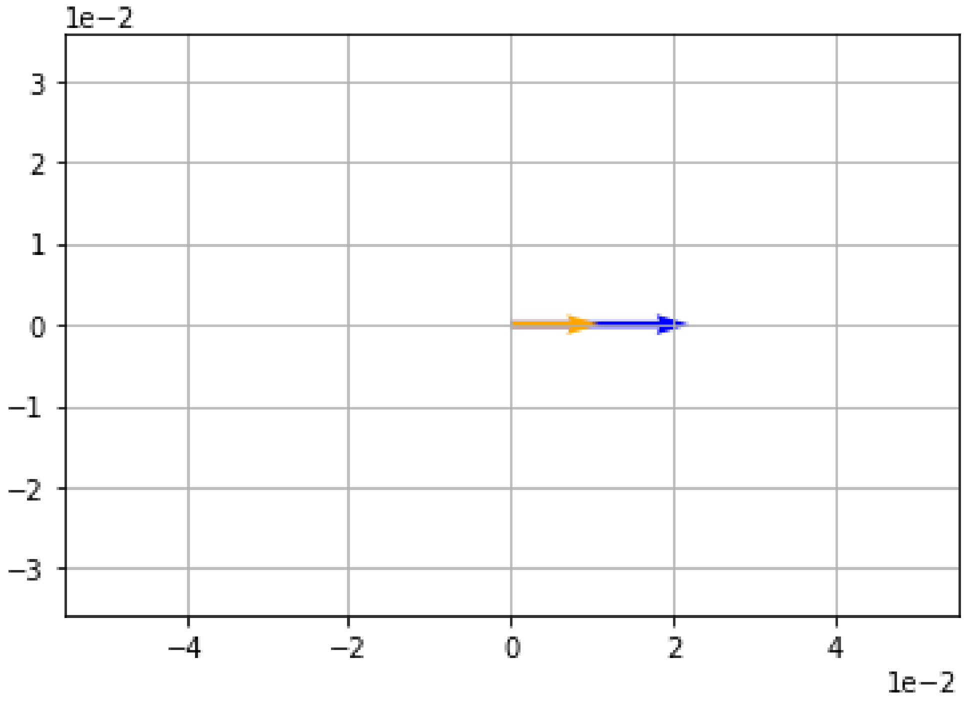
Now let's use Pythonto verify and plot these transformations:

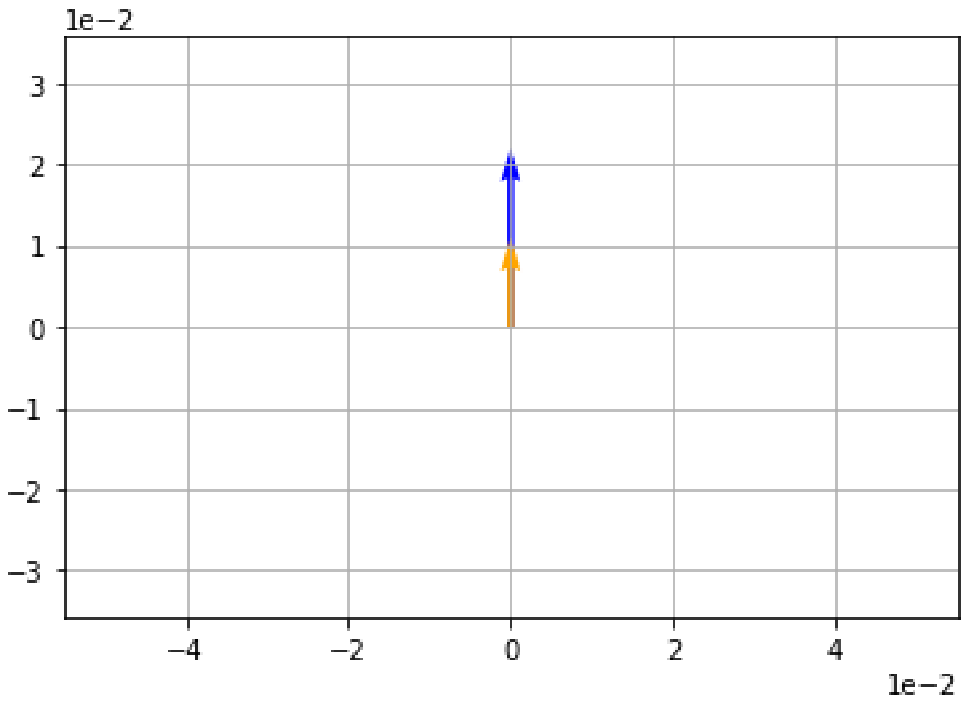
```
In [13]: vec1 = eVecs[:,0]
         lam1 = eVals[0]

         print('Matrix A:')
         print(A)
         print('-------')

         print('lam1: ' + str(lam1))
```

```
        print ('v1: ' + str(vec1))
        print ('Av1: ' + str(A@vec1))
        print ('lam1 x v1: ' + str(lam1*vec1))

        print('-------')

        vec2 = eVecs[:,1]
        lam2 = eVals[1]

        print('lam2: ' + str(lam2))
        print ('v2: ' + str(vec2))
        print ('Av2: ' + str(A@vec2))
        print ('lam2 x v2: ' + str(lam2*vec2))


        # Plot the resulting vectors
        t1 = lam1*vec1
        t2 = lam2*vec2

        fig = plt.figure()
        a=fig.add_subplot(1,1,1)
        # Plot v and t1
        vecs = np.array([t1,vec1])
        origin = [0], [0]
        plt.axis('equal')
        plt.grid()
        plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
        plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
        plt.show()
        a=fig.add_subplot(1,2,1)
        # Plot v and t2
        vecs = np.array([t2,vec2])
        origin = [0], [0]
        plt.axis('equal')
        plt.grid()
        plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
        plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
        plt.show()
```

```
Matrix A:
[[2 0]
 [0 2]]
-------
lam1: 2.0
v1: [1. 0.]
Av1: [2. 0.]
lam1 x v1: [2. 0.]
-------
```

```
lam2: 2.0
v2: [0. 1.]
Av2: [0. 2.]
lam2 x v2: [0. 2.]
```





Let's take a look at one more, slightly more complex example. Here's our matrix:

$$\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

Let's get the eigenvalue and eigenvector pairs:

```
In [14]: import numpy as np

         A = np.array([[2,1],
                       [1,2]])

         eVals, eVecs = np.linalg.eig(A)
         print(eVals)
         print(eVecs)
```

```
[3. 1.]
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

This time the eigenvalue-eigenvector pairs are:

$$\lambda_1 = 3, \vec{v}_1 = \begin{bmatrix} 0.70710678 \\ 0.70710678 \end{bmatrix} \qquad \lambda_2 = 1, \vec{v}_2 = \begin{bmatrix} -0.70710678 \\ 0.70710678 \end{bmatrix}$$

So let's check the first pair:

$$3 \times \begin{bmatrix}0.70710678\\0.70710678\end{bmatrix} = \begin{bmatrix}2.12132034\\2.12132034\end{bmatrix} \quad \textit{and} \quad \begin{bmatrix}2 & 1\\0 & 2\end{bmatrix} \cdot \begin{bmatrix}0.70710678\\0.70710678\end{bmatrix} = \begin{bmatrix}2.12132034\\2.12132034\end{bmatrix}$$

Now let's check the second pair:

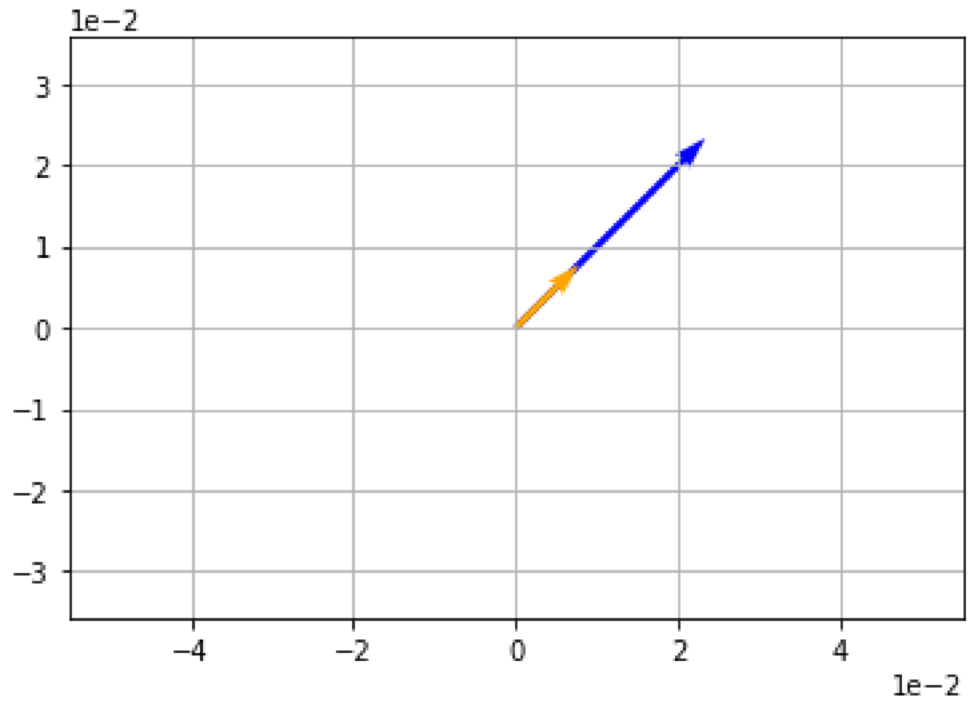
$$1 \times \begin{bmatrix}-0.70710678\\0.70710678\end{bmatrix} = \begin{bmatrix}-0.70710678\\0.70710678\end{bmatrix} \quad \textit{and} \quad \begin{bmatrix}2 & 1\\1 & 2\end{bmatrix} \cdot \begin{bmatrix}-0.70710678\\0.70710678\end{bmatrix} = \begin{bmatrix}-0.70710678\\0.70710678\end{bmatrix}$$

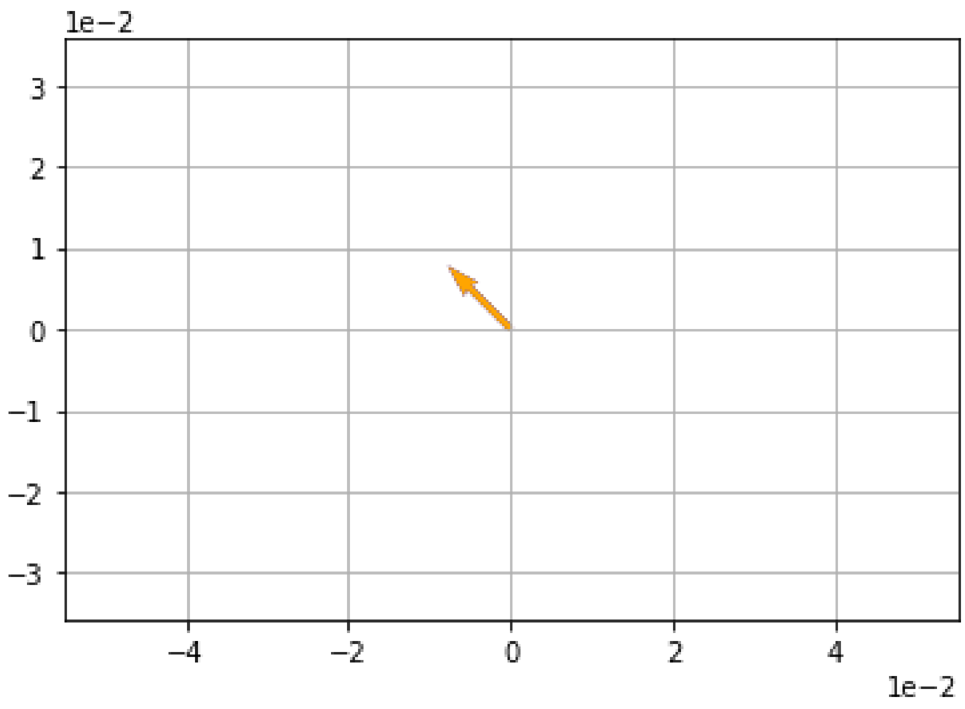
With more complex examples like this, it's generally easier to do it with Python:

```
In [15]: vec1 = eVecs[:,0]
         lam1 = eVals[0]

         print('Matrix A:')
         print(A)
         print('-------')

         print('lam1: ' + str(lam1))
         print ('v1: ' + str(vec1))
         print ('Av1: ' + str(A@vec1))
         print ('lam1 x v1: ' + str(lam1*vec1))

         print('-------')

         vec2 = eVecs[:,1]
         lam2 = eVals[1]

         print('lam2: ' + str(lam2))
         print ('v2: ' + str(vec2))
         print ('Av2: ' + str(A@vec2))
         print ('lam2 x v2: ' + str(lam2*vec2))


         # Plot the results
         t1 = lam1*vec1
         t2 = lam2*vec2

         fig = plt.figure()
         a=fig.add_subplot(1,1,1)
         # Plot v and t1
         vecs = np.array([t1,vec1])
         origin = [0], [0]
         plt.axis('equal')
         plt.grid()
         plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
         plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
         plt.show()
         a=fig.add_subplot(1,2,1)
```

```
# Plot v and t2
vecs = np.array([t2,vec2])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
plt.show()
```

```
Matrix A:
[[2 1]
 [1 2]]
-------
lam1: 3.0
v1: [0.70710678 0.70710678]
Av1: [2.12132034 2.12132034]
lam1 x v1: [2.12132034 2.12132034]
-------
lam2: 1.0
v2: [-0.70710678  0.70710678]
Av2: [-0.70710678  0.70710678]
lam2 x v2: [-0.70710678  0.70710678]
```





1.4 Eigendecomposition

So we've learned a little about eigenvalues and eigenvectors; but you may be wondering what use they are. Well, one use for them is to help decompose transformation matrices.

Recall that previously we found that a matrix transformation of a vector changes its magnitude, amplitude, or both. Without getting too technical about it, we need to remember that vectors can exist in any spatial orientation, or *basis*; and the same transformation can be applied in different *bases*.

We can decompose a matrix using the following formula:

$$A = Q\Lambda Q^{-1}$$

Where $\boldsymbol{A}$ is a trasformation that can be applied to a vector in its current base, $\boldsymbol{Q}$ is a matrix of eigenvectors that defines a change of basis, and is a matrix with eigenvalues on the diagonal that defines the same linear transformation as $\boldsymbol{A}$ in the base defined by $\boldsymbol{Q}$.

Let's look at these in some more detail. Consider this matrix:

$$A = \begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix}$$

$\boldsymbol{Q}$ is a matrix in which each column is an eigenvector of $\boldsymbol{A}$; which as we've seen previously, we can calculate using Python:

```
In [16]: import numpy as np
```

```
         A = np.array([[3,2],
                       [1,0]])

         l, Q = np.linalg.eig(A)
         print(Q)
```

```
[[ 0.96276969 -0.48963374]
 [ 0.27032301  0.87192821]]
```

So for matrix ***A***, ***Q*** is the following matrix:

$$Q = \begin{bmatrix} 0.96276969 & -0.48963374 \\ 0.27032301 & 0.87192821 \end{bmatrix}$$

is a matrix that contains the eigenvalues for ***A*** on the diagonal, with zeros in all other elements; so for a 2x2 matrix, will look like this:

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

In our Python code, we've already used the ***linalg.eig*** function to return the array of eigenvalues for ***A*** into the variable ***l***, so now we just need to format that as a matrix:

```
In [17]: L = np.diag(l)
         print (L)
```

```
[[ 3.56155281  0.        ]
 [ 0.         -0.56155281]]
```

So is the following matrix:

$$\Lambda = \begin{bmatrix} 3.56155281 & 0 \\ 0 & -0.56155281 \end{bmatrix}$$

Now we just need to find ***Q-1***, which is the inverse of ***Q***:

```
In [18]: Qinv = np.linalg.inv(Q)
         print(Qinv)
```

```
[[ 0.89720673  0.50382896]
 [-0.27816009  0.99068183]]
```

The inverse of ***Q*** then, is:

$$Q^{-1} = \begin{bmatrix} 0.89720673 & 0.50382896 \\ -0.27816009 & 0.99068183 \end{bmatrix}$$

So what does that mean? Well, it means that we can decompose the transformation of *any* vector multiplied by matrix ***A*** into the separate operations ***QQ-1***:

$$A\vec{v} = Q\Lambda Q^{-1}\vec{v}$$

To prove this, let's take vector $\boldsymbol{v}$:

$$\vec{v} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

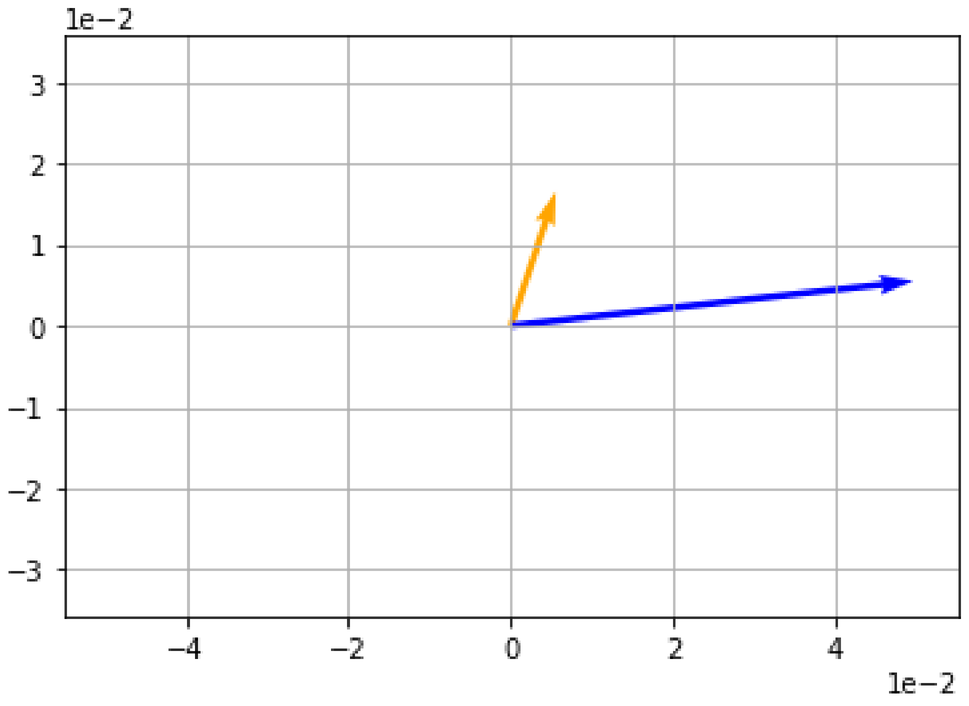
Our matrix transformation using $\boldsymbol{A}$ is:

$$\begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

So let's show the results of that using Python:

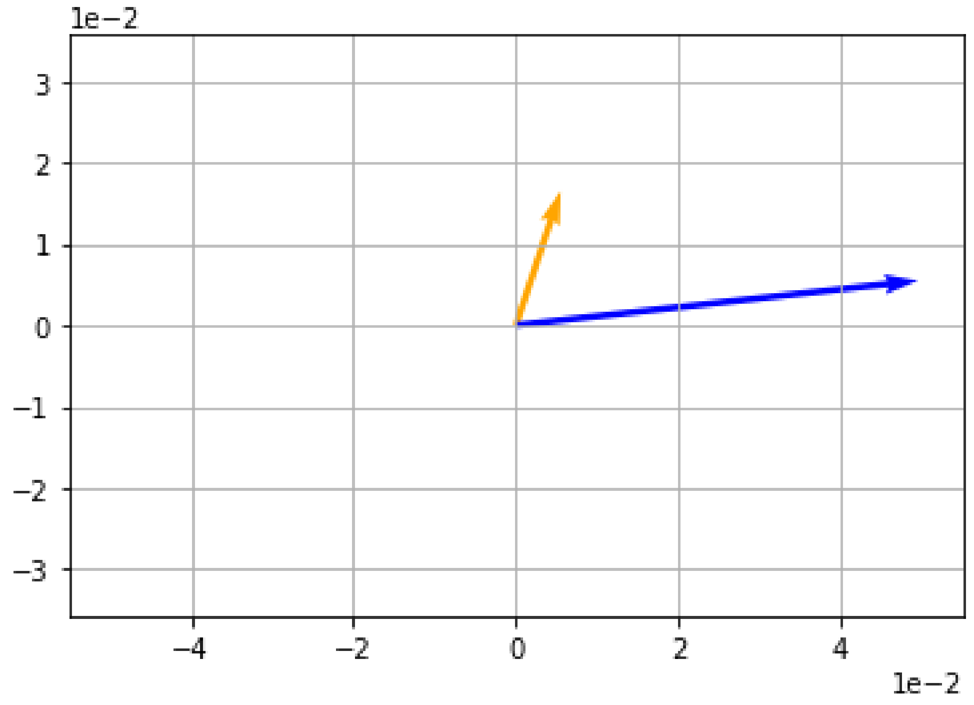
```
In [19]: v = np.array([1,3])
         t = A@v

         print(t)

         # Plot v and t
         vecs = np.array([v,t])
         origin = [0], [0]
         plt.axis('equal')
         plt.grid()
         plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
         plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'b'], scale=20)
         plt.show()
```

```
[9 1]
```



And now, let's do the same thing using the ***QQ-1*** sequence of operations:

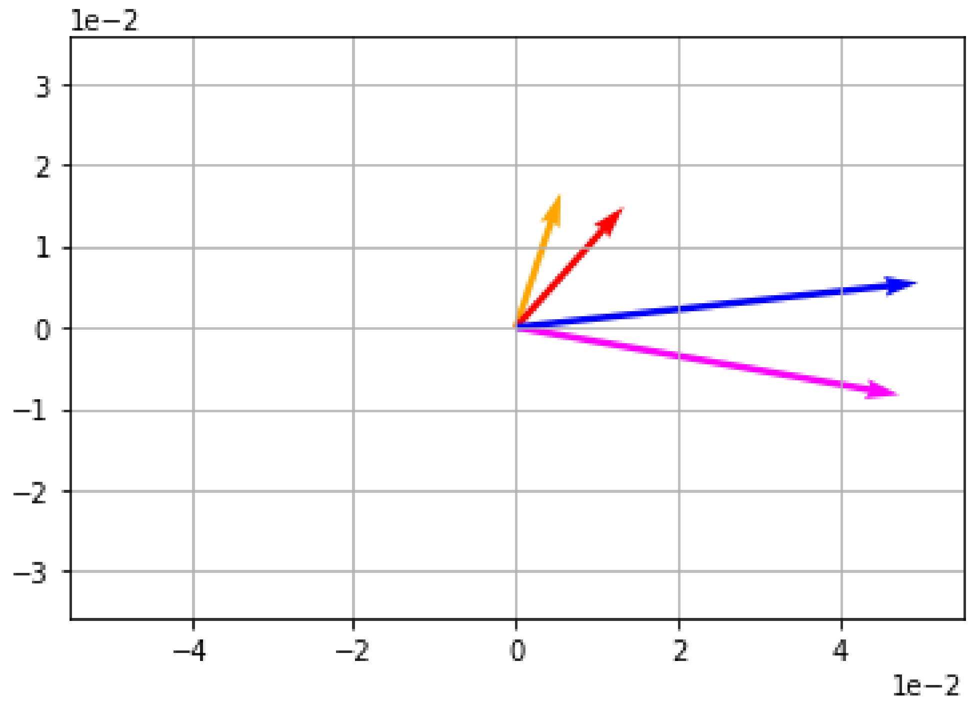
```
In [20]: import math
         import numpy as np
         import matplotlib.pyplot as plt
         %matplotlib inline

         t = (Q@(L@(Qinv)))@v

         # Plot v and t
         vecs = np.array([v,t])
         origin = [0], [0]
         plt.axis('equal')
         plt.grid()
         plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
         plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'b'], scale=20)
         plt.show()
```



So ***A*** and ***QQ-1*** are equivalent.

If we view the intermediary stages of the decomposed transformation, you can see the transformation using ***A*** in the original base for v (orange to blue) and the transformation using in the change of basis decribed by ***Q*** (red to magenta):

```
In [21]: import numpy as np
         import matplotlib.pyplot as plt
         %matplotlib inline

         t1 = Qinv@v
         t2 = L@t1
         t3 = Q@t2

         # Plot the transformations
         vecs = np.array([v,t1, t2, t3])
         origin = [0], [0]
         plt.axis('equal')
         plt.grid()
         plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
         plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'red', 'magenta', 'blue'],
         plt.show()
```



So from this visualization, it should be apparent that the transformation ***Av*** can be performed by changing the basis for *v* using ***Q*** (from orange to red in the above plot) applying the equivalent linear transformation in that base using (red to magenta), and switching back to the original base using ***Q-1*** (magenta to blue).

1.5 Rank of a Matrix

The **rank** of a square matrix is the number of non-zero eigenvalues of the matrix. A **full rank** matrix has the same number of non-zero eigenvalues as the dimension of the matrix. A **rank-**

deficient matrix has fewer non-zero eigenvalues as dimensions. The inverse of a rank deficient matrix is singular and so does not exist (this is why in a previous notebook we noted that some matrices have no inverse).

Consider the following matrix $\boldsymbol{A}$:

$$A = \begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix}$$

Let's find its eigenvalues ():

```
In [22]: import numpy as np
         A = np.array([[1,2],
                       [4,3]])
         l, Q = np.linalg.eig(A)
         L = np.diag(l)
         print(L)
```

```
[[-1.  0.]
 [ 0.  5.]]
```

$$\Lambda = \begin{bmatrix} -1 & 0 \\ 0 & 5 \end{bmatrix}$$

This matrix has full rank. The dimensions of the matrix is 2. There are two non-zero eigenvalues.

Now consider this matrix:

$$B = \begin{bmatrix} 3 & -3 & 6 \\ 2 & -2 & 4 \\ 1 & -1 & 2 \end{bmatrix}$$

Note that the second and third columns are just scalar multiples of the first column.

Let's examine it's eigenvalues:

```
In [23]: B = np.array([[3,-3,6],
                       [2,-2,4],
                       [1,-1,2]])
         lb, Qb = np.linalg.eig(B)
         Lb = np.diag(lb)
         print(Lb)
```

```
[[3.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 5.23364153e-16 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]]
```

$$\Lambda = \begin{bmatrix} 3 & 0 & 0 \\ 0 & -6 \times 10^{-17} & 0 \\ 0 & 0 & 3.6 \times 10^{-16} \end{bmatrix}$$

Note that matrix has only 1 non-zero eigenvalue. The other two eigenvalues are so extremely small as to be effectively zero. This is an example of a rank-deficient matrix; and as such, it has no inverse.

1.6 Inverse of a Square Full Rank Matrix

You can calculate the inverse of a square full rank matrix by using the following formula:

$$A^{-1} = Q\Lambda^{-1}Q^{-1}$$

Let's apply this to matrix ***A***:

$$A = \begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix}$$

Let's find the matrices for ***Q***, ***-1***, and ***Q-1***:

```
In [25]: import numpy as np
         A = np.array([[1,2],
                       [4,3]])

         l, Q = np.linalg.eig(A)
         L = np.diag(l)
         print(Q)
         Linv = np.linalg.inv(L)
         Qinv = np.linalg.inv(Q)
         print(Linv)
         print(Qinv)
```

```
[[-0.70710678 -0.4472136 ]
 [ 0.70710678 -0.89442719]]
[[-1.  -0. ]
 [ 0.   0.2]]
[[-0.94280904  0.47140452]
 [-0.74535599 -0.74535599]]
```

So:

$$A^{-1} = \begin{bmatrix} -0.70710678 & -0.4472136 \\ 0.70710678 & -0.89442719 \end{bmatrix} \cdot \begin{bmatrix} -1 & -0 \\ 0 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} -0.94280904 & 0.47140452 \\ -0.74535599 & -0.74535599 \end{bmatrix}$$

Let's calculate that in Python:

```
In [26]: Ainv = (Q@(Linv@(Qinv)))
         print(Ainv)
```

```
[[-0.6  0.4]
 [ 0.8 -0.2]]
```

That gives us the result:

$$A^{-1} = \begin{bmatrix} -0.6 & 0.4 \\ 0.8 & -0.2 \end{bmatrix}$$

We can apply the ***np.linalg.inv*** function directly to ***A*** to verify this:

```
In [27]: print(np.linalg.inv(A))

[[-0.6  0.4]
 [ 0.8 -0.2]]
```