Linear Transformations

This concept of adding two vectors with fixed direction, but scaling them to get different combined vectors, is hugely important. This combined vector, except in cases of linear dependence, can point in any direction and have any length we choose. This sets up an intuition for linear transformations where we use a vector to transform another vector in a function-like manner.

Basis Vectors

Imagine we have two simple vectors $\hat{i}$ and $\hat{j}$ ("i-hat" and "j-hat"). These are known as *basis vectors*, which are used to describe transformations on other vectors. They typically have a length of 1 and point in perpendicular positive directions as visualized in Figure 4-13.

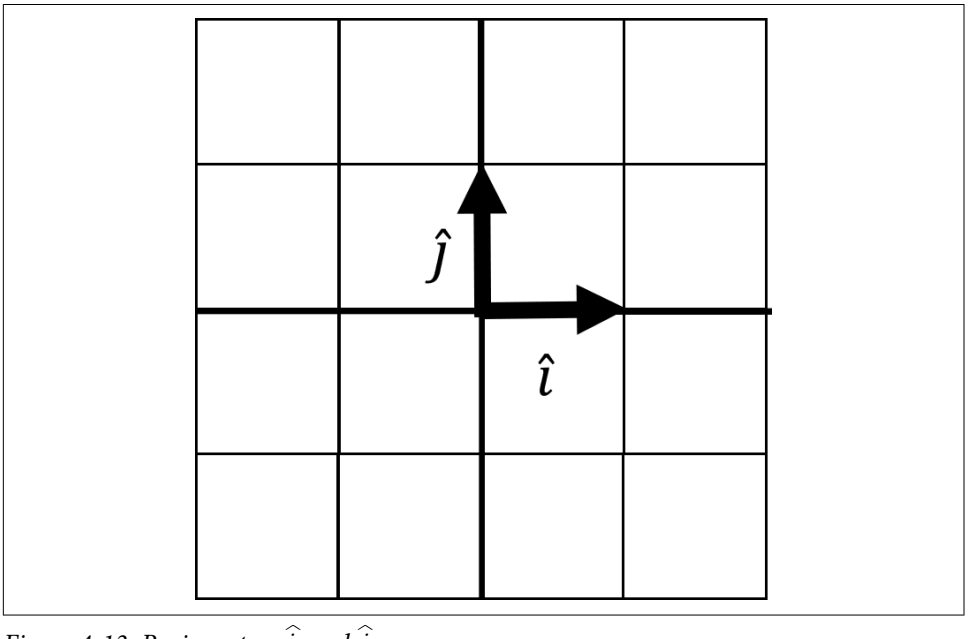

Figure 4-13. Basis vectors $\hat{i}$ and $\hat{j}$

Think of the basis vectors as building blocks to build or transform any vector. Our basis vector is expressed in a 2 × 2 matrix, where the first column is $\hat{i}$ and the second column is $\hat{j}$:

$$\hat{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\hat{j} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\text{basis} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

A *matrix* is a collection of vectors (such as $\hat{i}$, $\hat{j}$) that can have multiple rows and columns and is a convenient way to package data. We can use $\hat{i}$ and $\hat{j}$ to create any vector we want by scaling and adding them. Let's start with each having a length of 1 and showing the resulting vector $\vec{v}$ in Figure 4-14.

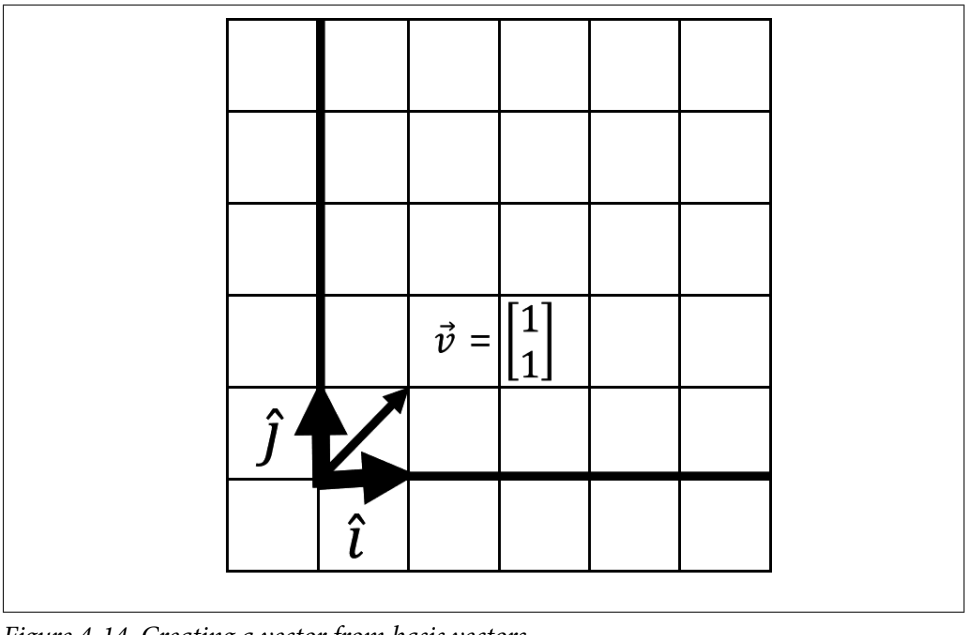

Figure 4-14. Creating a vector from basis vectors

I want vector $\vec{v}$ to land at [3, 2]. What happens to $\vec{v}$ if we stretch $\hat{i}$ by a factor of 3 and $\hat{j}$ by a factor of 2? First we scale them individually as shown here:

$$3\hat{i} = 3\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$

$$2\hat{j} = 2\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

If we stretched space in these two directions, what does this do to $\vec{v}$? Well, it is going to stretch with $\hat{i}$ and $\hat{j}$. This is known as a *linear transformation*, where we transform a vector with stretching, squishing, sheering, or rotating by tracking basis vector movements. In this case (Figure 4-15), scaling $\hat{i}$ and $\hat{j}$ has stretched space along with our vector $\vec{v}$.

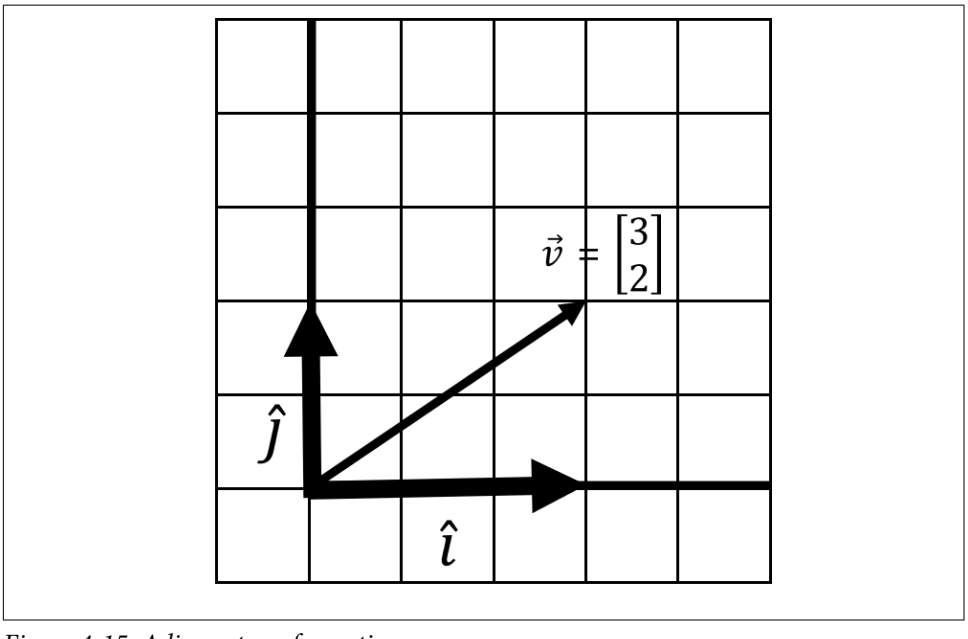

Figure 4-15. A linear transformation

But where does $\vec{v}$ land? It is easy to see where it lands here, which is [3, 2]. Recall that vector $\vec{v}$ is composed of adding $\hat{i}$ and $\hat{j}$. So we simply take the stretched $\hat{i}$ and $\hat{j}$ and add them together to see where vector $\vec{v}$ has landed:

$$\vec{v}_{new} = \begin{bmatrix} 3 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

Generally, with linear transformations, there are four movements you can achieve, as shown in Figure 4-16.

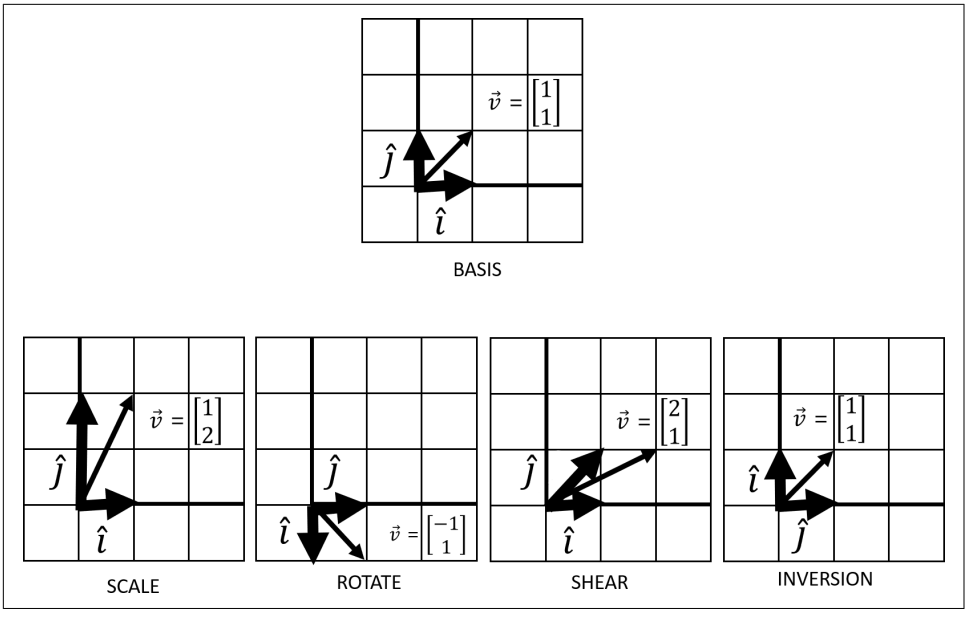

Figure 4-16. Four movements can be achieved with linear transformations

These four linear transformations are a central part of linear algebra. Scaling a vector will stretch or squeeze it. Rotations will turn the vector space, and inversions will flip the vector space so that $\widehat{i}$ and $\widehat{j}$ swap respective places.

It is important to note that you cannot have transformations that are nonlinear, resulting in curvy or squiggly transformations that no longer respect a straight line. This is why we call it linear algebra, not nonlinear algebra!

Matrix Vector Multiplication

This brings us to our next big idea in linear algebra. This concept of tracking where $\widehat{i}$ and $\widehat{j}$ land after a transformation is important because it allows us not just to create vectors but also to transform existing vectors. If you want true linear algebra enlightenment, think why creating vectors and transforming vectors are actually the same thing. It's all a matter of relativity given your basis vectors being a starting point before and after a transformation.

The formula to transform a vector $\overrightarrow{v}$ given basis vectors $\widehat{i}$ and $\widehat{j}$ packaged as a matrix is:

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

$\hat{i}$ is the first column [*a, c*] and $\hat{j}$ is the column [*b, d*]. We package both of these basis vectors as a matrix, which again is a collection of vectors expressed as a grid of numbers in two or more dimensions. This transformation of a vector by applying basis vectors is known as *matrix vector multiplication*. This may seem contrived at first, but this formula is a shortcut for scaling and adding $\hat{i}$ and $\hat{j}$ just like we did earlier adding two vectors, and applying the transformation to any vector $\vec{v}$.

So in effect, a matrix really is a transformation expressed as basis vectors.

To execute this transformation in Python using NumPy, we will need to declare our basis vectors as a matrix and then apply it to vector $\vec{v}$ using the `dot()` operator (Example 4-7). The `dot()` operator will perform this scaling and addition between our matrix and vector as we just described. This is known as the *dot product*, and we will explore it throughout this chapter.

Example 4-7. Matrix vector multiplication in NumPy

```
from numpy import array

# compose basis matrix with i-hat and j-hat
basis = array(
    [[3, 0],
     [0, 2]]
 )

# declare vector v
v = array([1,1])

# create new vector
# by transforming v with dot product
new_v = basis.dot(v)

print(new_v) # [3, 2]
```

When thinking in terms of basis vectors, I prefer to break out the basis vectors and then compose them together into a matrix. Just note you will need to *transpose*, or swap the columns and rows. This is because NumPy's `array()` function will do the opposite orientation we want, populating each vector as a row rather than a column. Transposition in NumPy is demonstrated in Example 4-8.

Example 4-8. Separating the basis vectors and applying them as a transformation

```
from numpy import array

# Declare i-hat and j-hat
i_hat = array([2, 0])
j_hat = array([0, 3])

# compose basis matrix using i-hat and j-hat
# also need to transpose rows into columns
basis = array([i_hat, j_hat]).transpose()

# declare vector v
v = array([1,1])

# create new vector
# by transforming v with dot product
new_v = basis.dot(v)

print(new_v) # [2, 3]
```

Here's another example. Let's start with vector $\vec{v}$ being [2, 1] and $\hat{i}$ and $\hat{j}$ start at [1, 0] and [0, 1], respectively. We then transform $\hat{i}$ and $\hat{j}$ to [2, 0] and [0, 3]. What happens to vector $\vec{v}$? Working this out mathematically by hand using our formula, we get this:

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} (2)(2) + (0)(1) \\ (2)(0) + (3)(1) \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$$

Example 4-9 shows this solution in Python.

Example 4-9. Transforming a vector using NumPy

```
from numpy import array

# Declare i-hat and j-hat
i_hat = array([2, 0])
j_hat = array([0, 3])

# compose basis matrix using i-hat and j-hat
# also need to transpose rows into columns
basis = array([i_hat, j_hat]).transpose()
```

```
# declare vector v 0
v = array([2,1])

# create new vector
# by transforming v with dot product
new_v = basis.dot(v)

print(new_v) # [4, 3]
```

The vector $\vec{v}$ now lands at [4, 3]. Figure 4-17 shows what this transformation looks like.

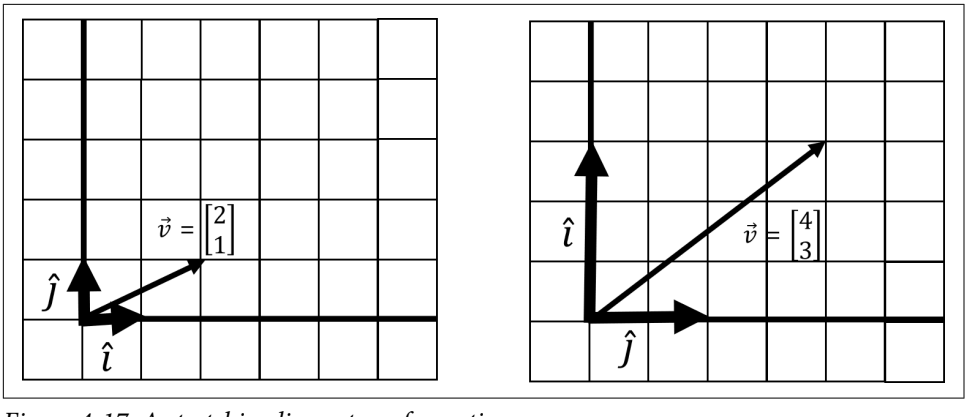

Figure 4-17. A stretching linear transformation

Here is an example that jumps things up a notch. Let's take vector $\vec{v}$ of value [2, 1]. $\hat{i}$ and $\hat{j}$ start at [1, 0] and [0, 1], but then are transformed and land at [2, 3] and [2, -1]. What happens to $\vec{v}$? Let's look in Figure 4-18 and Example 4-10.

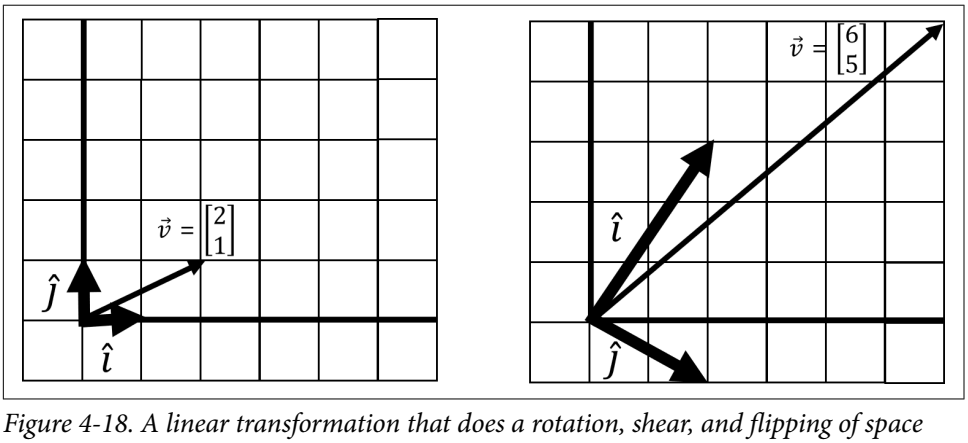

Figure 4-18. A linear transformation that does a rotation, shear, and flipping of space

Example 4-10. A more complicated transformation

```
from numpy import array

# Declare i-hat and j-hat
i_hat = array([2, 3])
j_hat = array([2, -1])

# compose basis matrix using i-hat and j-hat
# also need to transpose rows into columns
basis = array([i_hat, j_hat]).transpose()

# declare vector v 0
v = array([2,1])

# create new vector
# by transforming v with dot product
new_v = basis.dot(v)

print(new_v) # [6, 5]
```

A lot has happened here. Not only did we scale $\hat{i}$ and $\hat{j}$ and elongate vector $\vec{v}$. We actually sheared, rotated, and flipped space, too. You know space was flipped when $\hat{i}$ and $\hat{j}$ change places in their clockwise orientation, and we will learn how to detect this with determinants later in this chapter.

Basis Vectors in 3D and Beyond

You might be wondering how we think of vector transformations in three dimensions or more. The concept of basis vectors extends quite nicely. If I have a three-dimensional vector space, then I have basis vectors $\hat{i}$, $\hat{j}$, and $\hat{k}$. I just keep adding more letters from the alphabet for each new dimension (Figure 4-19).

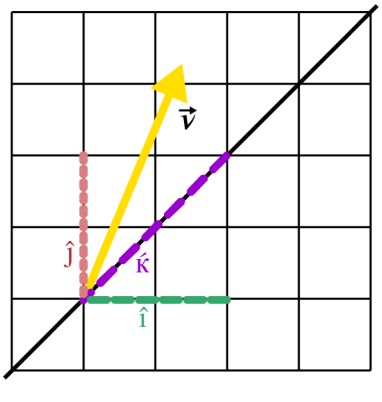

Figure 4-19. 3D basis vector

Something else that is worth pointing out is some linear transformations can shift a vector space into fewer or more dimensions, and that is exactly what nonsquare matrices will do (where number of rows and columns are not equal). In the interest of time we cannot explore this. But 3Blue1Brown explains and animates this concept beautifully.

Matrix Multiplication

We learned how to multiply a vector and a matrix, but what exactly does multiplying two matrices accomplish? Think of *matrix multiplication* as applying multiple transformations to a vector space. Each transformation is like a function, where we apply the innermost first and then apply each subsequent transformation outward.

Here is how we apply a rotation and then a shear to any vector $\vec{v}$ with value [*x*, *y*]:

$$\begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

We can actually consolidate these two transformations by using this formula, applying one transformation onto the last. You multiply and add each row from the first matrix to each respective column of the second matrix, in an "over-and-down! over-and-down!" pattern:

$$\begin{bmatrix}a & b\\c & d\end{bmatrix}\begin{bmatrix}e & f\\g & h\end{bmatrix} = \begin{bmatrix}ae+bg & af+bh\\ce+dy & cf+dh\end{bmatrix}$$

So we can actually consolidate these two separate transformations (rotation and shear) into a single transformation:

$$\begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

$$= \begin{bmatrix}(1)(0)+(1)(1) & (-1)(1)+(1)(0)\\(0)(0)+(1)(1) & (0)(-1)+(1)(0)\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

$$= \begin{bmatrix}1 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

To execute this in Python using NumPy, you can combine the two matrices simply using the `matmul()` or `@` operator (Example 4-11). We will then turn around and use this consolidated tranformation and apply it to a vector [1, 2].

Example 4-11. Combining two transformations

```
from numpy import array

# Transformation 1
i_hat1 = array([0, 1])
j_hat1 = array([-1, 0])
transform1 = array([i_hat1, j_hat1]).transpose()

# Transformation 2
i_hat2 = array([1, 0])
j_hat2 = array([1, 1])
transform2 = array([i_hat2, j_hat2]).transpose()

# Combine Transformations
combined = transform2 @ transform1

# Test
print("COMBINED MATRIX:\n {}".format(combined))

v = array([1, 2])
print(combined.dot(v))  # [-1, 1]
```

Using `dot()` Versus `matmul()` and `@`

In general, you want to prefer `matmul()` and its shorthand `@` to combine matrices rather than the `dot()` operator in NumPy. The former generally has a preferable policy for higher-dimensional matrices and how the elements are broadcasted.

If you like diving into these kinds of implementation details, this StackOverflow question is a good place to start.

Note that we also could have applied each transformation individually to vector $\vec{v}$ and still have gotten the same result. If you replace the last line with these three lines applying each transformation, you will still get [-1, 1] on that new vector:

```
rotated = transform1.dot(v)
sheered = transform2.dot(rotated)
print(sheered) # [-1, 1]
```

Note that the order you apply each transformation matters! If we apply `transformation1` on `transformation2`, we get a different result of [-2, 3] as calculated in Example 4-12. So matrix dot products are not commutative, meaning you cannot flip the order and expect the same result!

Example 4-12. Applying the transformations in reverse

```
from numpy import array

# Transformation 1
i_hat1 = array([0, 1])
j_hat1 = array([-1, 0])
transform1 = array([i_hat1, j_hat1]).transpose()

# Transformation 2
i_hat2 = array([1, 0])
j_hat2 = array([1, 1])
transform2 = array([i_hat2, j_hat2]).transpose()

# Combine Transformations, apply sheer first and then rotation
combined = transform1 @ transform2

# Test
print("COMBINED MATRIX:\n {}".format(combined))

v = array([1, 2])
print(combined.dot(v)) # [-2, 3]
```

Think of each transformation as a function, and we apply them from the innermost to outermost just like nested function calls.

Linear Transformations in Practice

You might be wondering what all these linear transformations and matrices have to do with data science and machine learning. The answer is everything! From importing data to numerical operations with linear regression, logistic regression, and neural networks, linear transformations are the heart of mathematically manipulated data.

In practice, however, you will rarely take the time to geometrically visualize your data as vector spaces and linear transformations. You will be dealing with too many dimensions to do this productively. But it is good to be mindful of of the geometric interpretation just to understand what these contrived-looking numerical operations do! Otherwise, you are just memorizing numerical operation patterns without any context. It also makes new linear algebra concepts like determinants more obvious.

Determinants

When we perform linear transformations, we sometimes "expand" or "squish" space and the degree this happens can be helpful. Take a sampled area from the vector space in Figure 4-20: what happens to it after we scale $\hat{i}$ and $\hat{j}$?

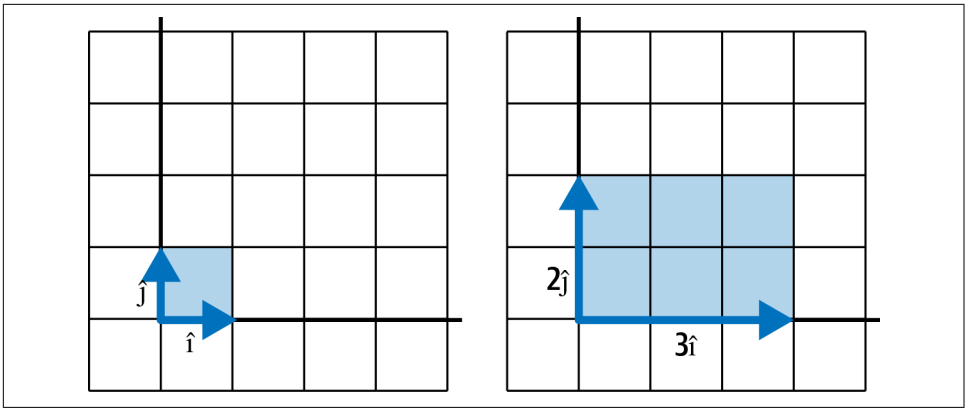

Figure 4-20. A determinant measures how a linear transformation scales an area

Note it increases in area by a factor of 6.0, and this factor is known as a *determinant*. Determinants describe how much a sampled area in a vector space changes in scale with linear transformations, and this can provide helpful information about the transformation.

Example 4-13 shows how to calculate this determinant in Python.

Example 4-13. Calculating a determinant

```
from numpy.linalg import det
from numpy import array

i_hat = array([3, 0])
j_hat = array([0, 2])

basis = array([i_hat, j_hat]).transpose()

determinant = det(basis)

print(determinant) # prints 6.0
```

Simple shears and rotations should not affect the determinant, as the area will not change. Figure 4-21 and Example 4-14 shows a simple shear and the determinant remains a factor 1.0, showing it is unchanged.

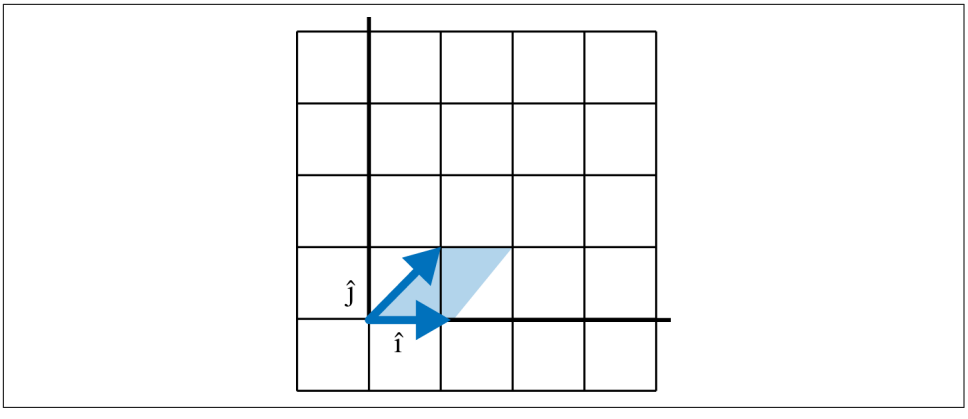

Figure 4-21. A simple shear does not change the determinant

Example 4-14. A determinant for a shear

```
from numpy.linalg import det
from numpy import array

i_hat = array([1, 0])
j_hat = array([1, 1])

basis = array([i_hat, j_hat]).transpose()

determinant = det(basis)

print(determinant) # prints 1.0
```

But scaling will increase or decrease the determinant, as that will increase/decrease the sampled area. When the orientation flips ($\hat{i}$, $\hat{j}$ swap clockwise positions), then the determinant will be negative. Figure 4-22 and Example 4-15 illustrate a determinant showing a transformation that not only scaled but also flipped the orientation of the vector space.

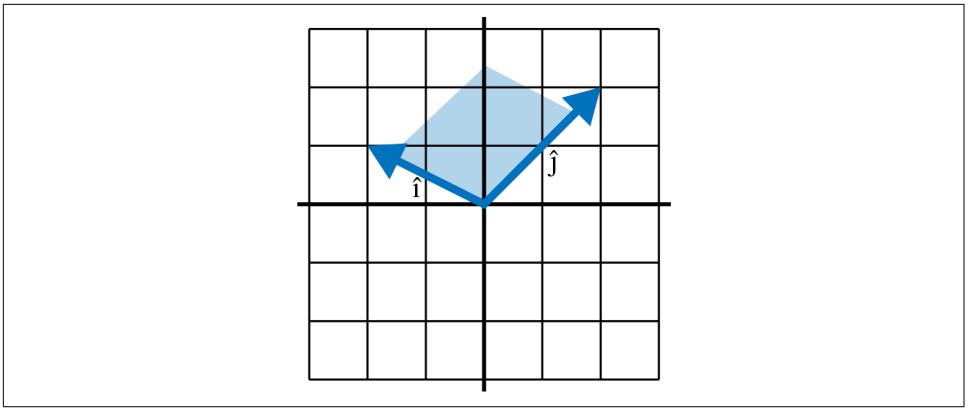

Figure 4-22. A determinant on a flipped space is negative

Example 4-15. A negative determinant

```
from numpy.linalg import det
from numpy import array

i_hat = array([-2, 1])
j_hat = array([1, 2])

basis = array([i_hat, j_hat]).transpose()

determinant = det(basis)

print(determinant) # prints -5.0
```

Because this determinant is negative, we quickly see that the orientation has flipped. But by far the most critical piece of information the determinant tells you is whether the transformation is linearly dependent. If you have a determinant of 0, that means all of the space has been squished into a lesser dimension.

In Figure 4-23 we see two linearly dependent transformations, where a 2D space is compressed into one dimension and a 3D space is compressed into two dimensions. The area and volume respectively in both cases are 0!

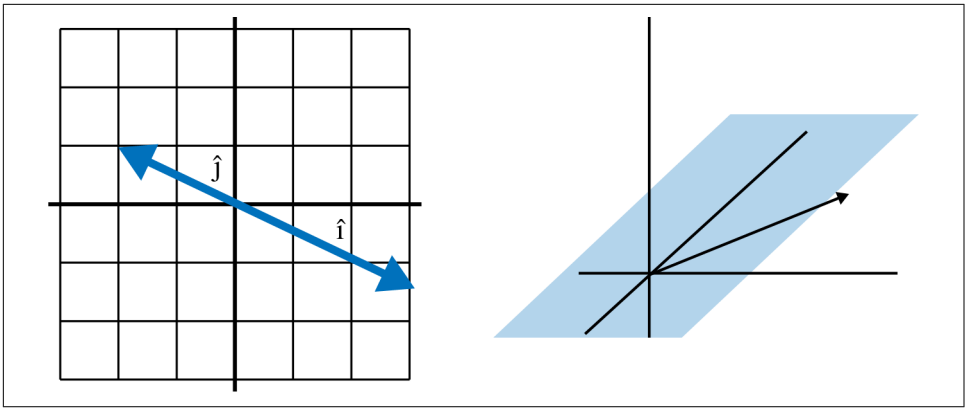

Figure 4-23. Linear dependence in 2D and 3D

Example 4-16 shows the code for the preceding 2D example squishing an entire 2D space into a single one-dimensional number line.

Example 4-16. A determinant of zero

```
from numpy.linalg import det
from numpy import array

i_hat = array([-2, 1])
j_hat = array([3, -1.5])

basis = array([i_hat, j_hat]).transpose()

determinant = det(basis)

print(determinant) # prints 0.0
```

So testing for a 0 determinant is highly helpful to determine if a transformation has linear dependence. When you encounter this you will likely find a difficult or unsolvable problem on your hands.