Contents

Chapter 1

Introduction to Linear Algebra

Welcome to the world of linear algebra, a branch of mathematics that relies on vectors, matrices, and linear transformations. You are familiar with most of these concepts, so in this workbook you'll see how you can use them together to solve problems.

Let's review what you know.

- **Vectors**. In workbook 5, you saw how vectors can represent forces and then how to add and multiply them to figure out such things as rocket engine force and direction.
- **Matrices**. In workbook 8 you learned to use spreadsheets to solve problems numerically, like how to figure out the number of barrels a cooper has to produce to achieve a certain take-home pay. Spreadsheets are essentially matrices—a row by column structure that contains values.
- **Linear transformations**. When you studied congruence in workbook 4, you were introduced to a few linear transformations such as translation and reflection.

1.1 What's With the Linear?

You might be thinking, "Hey, haven't I've been doing algebra already?"

You have. You've come a long way in your problem solving journey. You've used algebra to solve simple equations like $7x + 10 = 24$ and quadratic equations like $4x^2 + 9x + 31 = 0$. What distinguishes linear algebra is that linear combinations are at its heart. Any equation with a power greater than 1, like a quadratic, is nonlinear.

We'll first take a look at linear combinations

1.2 Linear Combinations

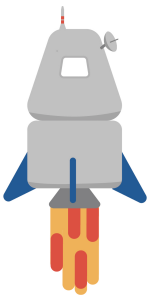
You won't see any **sin**, **cos**, or **tan** operations in this section. Linear operations do NOT use trigonometric functions. Those are all nonlinear. A linear combination preserves addition and scalar multiplication. You will see that linear combinations allow you to solve many types of problems in science and engineering. Before we get deep into the numbers, let's take a look at a few linear operations you can perform on images. This will give you an intuition for the underlying math. Then we'll take a look at some numbers.

1.3 Image Operations

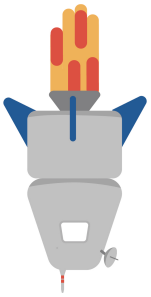
The simplest image, a bitmap, can be represented by a two-dimensional matrix of values, either 0, for black, or 1, for white. Grayscale images are also represented by a two-dimensional matrix of values, but the values typically range from 0 to 255. Zero is black, 255 is white, the values in between represent shades of gray.

Color images are more complex. The simplest color image is a three-dimensional matrix of values. You can think of it as three 2D matrices, one to represent red values (R), another for green values (G), and the third for blue values (B). The combination of R, G, and B determines the color you see.

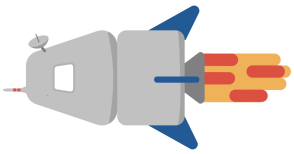
Working with images, means working with millions of pixels. Fortunately modern techniques make this a snap. Let's look at some common operations on an image of a rocket.



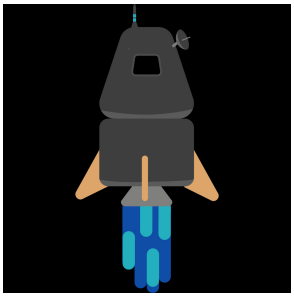
Flipping is a linear operation.



Next, the image is rotated 90 degrees. This rotation is linear, but if you want to rotate it at an angle that isn't a multiple of 90, you would need trigonometry. You'd be treading into nonlinear territory. But that happens in the field of linear algebra. You'll learn about nonlinear extensions later that use trigonometric functions and imaginary numbers.



Inversion is an interesting linear operation that involves redefining the red, green and blue values such that the new value is 1.0 minus the old value. The resulting black background gives the impression the rocket is in deep space, don't you think?



It is possible to redefine the red, green, and blue values in many ways. Visit the NASA website and search for false color images. NASA and other scientists redefine colors to communicate such things as the amount of vegetation or water in an area, the temperatures of the sun's surface, and so on. Photographers often do this for artistic effect. For example, the image on the left was taken with an infrared camera. (But not the thermal infrared you've seen. This is the infrared that's emitted by living plants.) The image is further processed to swap channels. For example, the matrix representing red might be

swapped with the matrix representing blue. The image on the right shows the image after swapping color values. All these swapping operations are linear.

1.4 The Numbers Behind Some Image Operations

You'll see a few matrices in this section. Let's first see how a spreadsheet can be represented as a matrix. Recall the barrel making shop example. This is part of that spreadsheet.

	A	B	C	D
1	**Barrels Produced** (per month)	115	120	125
2	**Materials cost** (per barrel)	$45.00	$45.00	$45.00
3	**Sale price** (per barrel)	$100.00	$100.00	$100.00
4	**Rent** (per month)	$2,000.00	$2,000.00	$2,000.00
5	**Pretax Earnings** (per month)	$4,325.00	$4,600.00	$4,875.00
6	**Taxes** (per month)	$865.00	$920.00	$975.00
7	**Take home pay** (per month)	$3,460.00	$3,680.00	$3,900.00

Represented as a matrix, it looks like the following. Note the differences. A matrix contains only values, no labels. This matrix uses floating point values, hence the inclusion of decimal points.

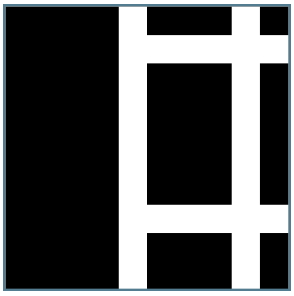
$$\begin{bmatrix} 115. & 120. & 125. \\ 45.0 & 45.0 & 45.0 \\ 100.0 & 100.0 & 100.0 \\ 2000. & 2000. & 2000. \\ 4325. & 4600. & 4875. \\ 865. & 920. & 975. \\ 3460. & 3680. & 3900. \end{bmatrix}$$

A matrix that represents an image contains only pixel values, whereas the barrel making shop matrix represents seven kinds of variables: barrels produced, materials cost, sales price, rent, pretax earnings, taxes, and take home pay.

The simplest image to create is a bitmap because that requires a matrix of zeros and ones. This is a matrix for a 10 pixel by 10 pixel image. Why the decimal points when this is obviously a matrix of integers? It turns out that when you use tools like python to process matrices, you must be conscious of data types. I already know that most of the python methods I use for image operations expect floating points. A few expect integer types, but you'll see how to handle type conversion later, in the section on python.

$$\begin{bmatrix}
0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 1. & 0 \\
0. & 0. & 0. & 0. & 1. & 1. & 1. & 1. & 1. & 1 \\
0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 1. & 0 \\
0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 1. & 0 \\
0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 1. & 0 \\
0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 1. & 0 \\
0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 1. & 0 \\
0. & 0. & 0. & 0. & 1. & 1. & 1. & 1. & 1. & 1 \\
0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 1. & 0 \\
0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 1. & 0
\end{bmatrix}$$

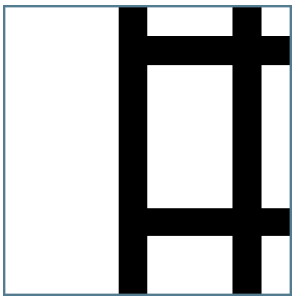
When converted to an image, it is very tiny. This is an enlarged version so you can see the pattern.



We can create an inverse of this image by changing all the values in the matrix so that 0 becomes 1 and 1 becomes 0. (Technically this is not the way you'd invert a matrix, as you'll see in the python section. For this example, you will get the same visual result, but when you start inverting matrices programmatically, you will learn the formal definition.)

$$\begin{bmatrix}
1. & 1. & 1. & 1. & 0. & 1. & 1. & 1. & 0. & 1 \\
1. & 1. & 1. & 1. & 0. & 0. & 0. & 0. & 0. & 0 \\
1. & 1. & 1. & 1. & 0. & 1. & 1. & 1. & 0. & 1 \\
1. & 1. & 1. & 1. & 0. & 1. & 1. & 1. & 0. & 1 \\
1. & 1. & 1. & 1. & 0. & 1. & 1. & 1. & 0. & 1 \\
1. & 1. & 1. & 1. & 0. & 1. & 1. & 1. & 0. & 1 \\
1. & 1. & 1. & 1. & 0. & 1. & 1. & 1. & 0. & 1 \\
1. & 1. & 1. & 1. & 0. & 0. & 0. & 0. & 0. & 0 \\
1. & 1. & 1. & 1. & 0. & 1. & 1. & 1. & 0. & 1 \\
1. & 1. & 1. & 1. & 0. & 1. & 1. & 1. & 0. & 1
\end{bmatrix}$$

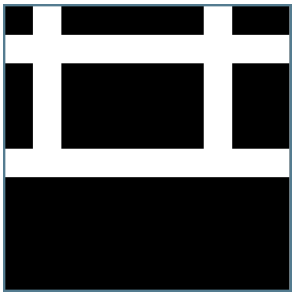
When converted to an image and enlarged, it looks like this:



Rotating the original matrix by 90 degrees gives this:

$$\begin{bmatrix}
0. & 1. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \\
1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. \\
0. & 1. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \\
0. & 1. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \\
0. & 1. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \\
1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0.
\end{bmatrix}$$

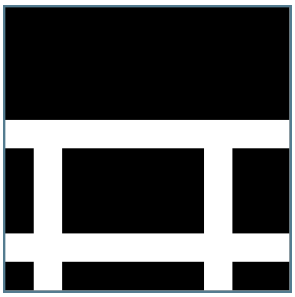
This is the resulting enlarged image:



You'll transpose many matrices in the upcoming pages. It requires swapping rows for columns.

$$\begin{bmatrix} 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 0. & 1. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \\ 0. & 1. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \\ 0. & 1. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \\ 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 0. & 1. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \end{bmatrix}$$

The resulting image looks like this:

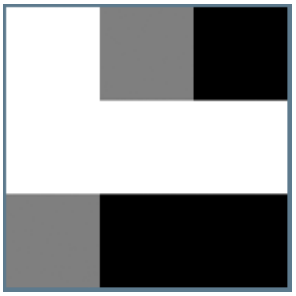


What about adding images? That fits the definition of a linear combination. Recall that grayscale images have values from 0 to 255. To make things simple, let's define two matrices with values ranging from 0.0 to 1.0. When we want a grayscale image, it's easy to multiply the matrix by 255.

Let's call this matrix f.

$$\begin{bmatrix} 1.0 & 0.5 & 0.0 \\ 1.0 & 1.0 & 1.0 \\ 0.5 & 0.0 & 0.0 \end{bmatrix}$$

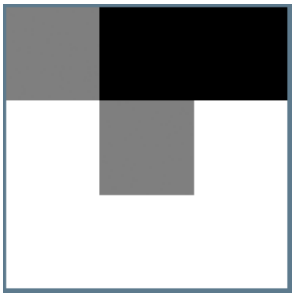
When multiplied by 255 and converted to a grayscale image:



Let's call this matrix g:

$$\begin{bmatrix} 0.5 & 0.0 & 0.0 \\ 1.0 & 0.5 & 1.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix}$$

When multiplied by 255 and converted to a grayscale image:



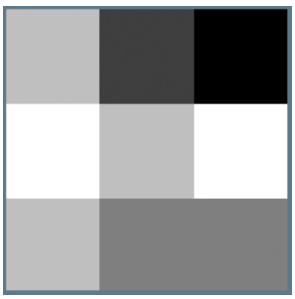
When we add f and g we get k:

$$\begin{bmatrix} 1.5 & 0.5 & 0.0 \\ 2.0 & 1.5 & 2.0 \\ 1.5 & 1.0 & 1.0 \end{bmatrix}$$

But the values in k exceed the range of 0.0 to 1.0, so we normalize by dividing the matrix by 1.0

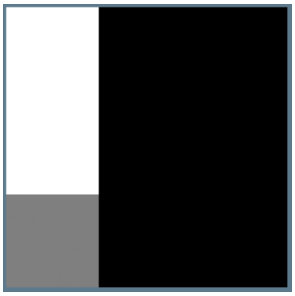
$$\begin{bmatrix} 0.75 & 0.25 & 0.00 \\ 1.00 & 0.75 & 1.00 \\ 0.75 & 0.50 & 0.50 \end{bmatrix}$$

When multiplied by 255 and converted to grayscale, we get:



Let's go back to the first small grayscale image:

If you want to keep the pattern in the first column, you could multiply the matrix by a vector, [1.0, 0.0, 0.0]. The 1.0 will keep the values in the first column, but the 0.0 will knock out the other values because 0.0 times anything equals 0.0.



What do you think will happen if you use the vector [0.0, 1.0, 0.0] or [0.0, 0.0, 1.0]? You'll get a chance later to use python to perform image operations.

All the operations we performed on these images satisfy the requirement for linear combinations: preserving addition and scalar multiplication.

1.5 Applications of Linear Algebra

So far you've seen how linear operations on matrices can process images by:

- multipling a matrix using a scalar (e.g., normalize, change the range)

- adding one matrix to another to get a composite image
- multipling two matrices to perform a transform (e.g., flipping)
- mulitpliing a matrix with a vector (isolating a channel)

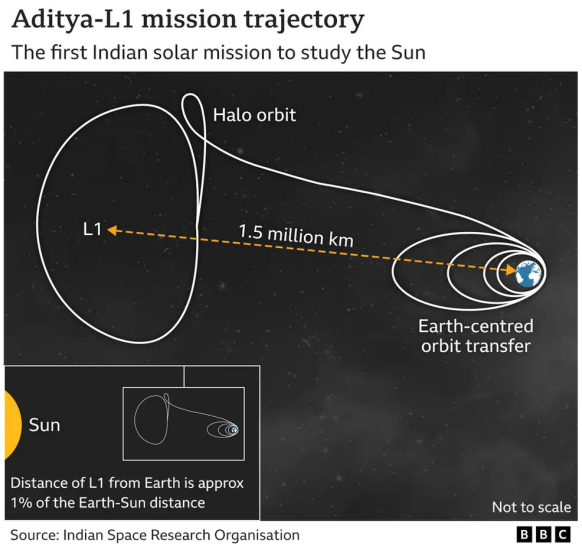
Many areas in engineering and science rely on the matrix operations defines by linear algebra. Besides image processing, linear algebra is used for:

- Computer Graphics. When you play a video game or watch the latest CG animation, matrix operations transform objects in the scene to make them appear as if moving, getting closer, and so on. You can represent the vertices of objects as vectors, and then apply a transformation matrix.
- Data Analysis. We live in an era in which it's easy to collect so much data that it's difficult to make sense of the data by just looking at it. You can represent the data in matrix form and then find a solution vector. For example, scientists use this technique to figure out the effectiveness of drug treatments on disease.
- Economics. Take a look at financial section of any news organization and you'll see headlines such as "Economic Data Points to Faster Growth" or "Is the Inflation Battle Won?" Economists can use systems of linear equations to represent economic indicators, such as consumer consumption, government spending, investment rate, and gross national product. By using various methods that you'll learn about later, they can get a good idea of the state of the economy.
- Engineering. Engineers couldn't do without linear algebra. For example, the orbital dynamics of space travel relies on it. Engineers must predict and calculate the the motion of planetary bodies, satellites, and spacecraft. By solving systems of linear equations engineers can make sure that a spacecraft travels to its destination without running into a satellite or space rock.

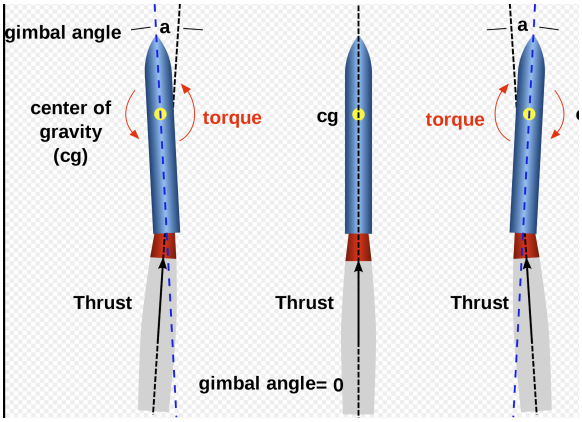
1.6 Let's Observe the Sun!

India recently sent the Aditya spacecraft on a mission to study the Sun. Without a thorough understanding of linear algebra (among other things), the engineers would not have accomplished the amazing feat of getting Aditya in a stable orbit around a Lagrange point. In previous chapters you learned about gravity and its effects.

A Lagrange point is a point in space between two bodies (e.g. Earth and Sun) where there is gravitational equilibrium. With the right trajectory, a spacecraft will orbit around a Lagrange point in a stable position that doesn't require much energy to maintain. That's called a Halo orbit. Because that there are no fueling stations in space, a Halo orbit will allow Aditya to maintain position for about 5 years. Pretty good mileage!



Aditya's engineers had to calculate a looping maneuver that would precisely inject the Aditya spacecraft into the Halo orbit. They determined the angles and burn times for the thrust engine. If they were wrong in one direction, the spacecraft would fly off to the sun. The other direction would send the spacecraft back in the direction of Earth. Their success is due to a solid understanding of vectors and linear algebra.



1.7 Images in Python

One of the wonderful things about python is the availability of libraries for specialized computation. The Python Imaging Library, PIL, is what you'll use to create images, read existing images from disk, and perform operation on images. To create and manipulate arrays, you will use NumPy.

Create a file called `image_creation.py` and enter this code:

```
# Import necessary modules
import numpy as np
import PIL
from PIL import Image
from PIL import ImageOps

# Create a 10 by 10 pixel bitmap Image.
# Using a decimal point ensure python see the values as floating point numbers
# Some image operations assume floats

bitmapArray = np.array([
[0., 0., 0., 0., 1., 0., 0., 0., 1., 0.],
[0., 0., 0., 0., 1., 1., 1., 1., 1., 1.],
[0., 0., 0., 0., 1., 0., 0., 0., 1., 0.],
[0., 0., 0., 0., 1., 0., 0., 0., 1., 0.],
[0., 0., 0., 0., 1., 0., 0., 0., 1., 0.],
[0., 0., 0., 0., 1., 0., 0., 0., 1., 0.],
[0., 0., 0., 0., 1., 0., 0., 0., 1., 0.],
[0., 0., 0., 0., 1., 1., 1., 1., 1., 1.],
[0., 0., 0., 0., 1., 0., 0., 0., 1., 0.],
[0., 0., 0., 0., 1., 0., 0., 0., 1., 0.]])

# Image.fromarray assumes a range of 0 to 255, so scale by 255

myImage = Image.fromarray(bitmapArray*255)
myImage.show()

# A window opens with an image so tiny you might think nothing is there
# Zoom in to see the pattern

# Transpose the array, create an image, and then show it.
# Note that you operate on the original array (not the image)
# Remember to zoom in to see the pattern

myImageTransposed = bitmapArray.transpose()
myImageTransposed.show()

# Invert the array. You'll use the NumPy invert method.
# The invert method assumes integer values. You need to convert the data type
# Numpy has a method for that

intBitmapArray = np.asarray(bitmapArray,dtype="int")
invertedArray = np.invert(intBitmapArray)

# Take a look at the array
```

```
invertedArray

# The values range from -2 to -1. Image values are positive.
# You need to change the range so the values are from 0 to 255
# Further you need to change back to floating point values because
# the PIL method requires them

invertedArray = (invertedArray + 2)*1.0
invertedImage = Image.fromarray(255*invertedArray)
invertedImage.show()

# Zoom in on the image and compare the pattern with the original
```

1.8 Exercise

Create a python program that creates matrix f and matrix g from the previous section, and then performs all the operations shown in that section. If you are not sure how to accomplish something, consult the online documentation for the PIL and NumPy python libraries.

Chapter 2

Vectors and Matrices

THIS CHAPTER IS UNDER MAJOR CONSTRUCTION. IGNORE IT FOR NOW

You've come a long way in your problem solving journey. You've used algebra to solve simple equations like $7x + 10 = 24$ and quadratic equations like $4x^2 + 9x + 31 = 0$. Algebra works with scalars, regular numbers that represent values. You also learned about vectors, quantities that represent both a magnitude and direction. Recall the parachute jumper in Workbook 5. To figure out the force, and in turn how fast the parachutist is traveling, you needed to know the direction and speed of the wind, the direction and speed of the plane, the wind resistance, and gravity. A scalar can represent speed, but to take direction into account, you need to use vectors. That's where linear algebra comes in.

Linear algebra is a specialized form of algebra that represents and manipulate linearly related variables. It uses vectors and matrices to represent variables. A vector is a one-dimensional array of numbers. A matrix is a multidimensional array of numbers. In this chapter, a matrix will be two dimensions. You can think of a matrix as a collection of vectors.

2.1 Vector-Matrix Multiplication

Let's take a look at the general form of vector-matrix multiplication. Given a matrix A of size $m \times n$ and a vector x of size $n \times 1$, the product Ax is a new vector of size $m \times 1$.

You compute the i-th component of the product vector Av by taking the dot product of the i-th row of A and the vector v:

$$(Av)_i = \sum_{j=1}^{n} a_{i,j} x_j$$

where $a_{i,j}$ is the element in the i-th row and j-th column of A, and v_j is the j-th element of v.

This is the general form of a matrix and a vector, written to show the specific components of each:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \dots & a_{2,n} \\ \dots & & & & \\ a_{m,1} & a_{m,2} & a_{m,3} & \dots & a_{m,n} \end{bmatrix}$$

$$v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \dots \\ v_m \end{bmatrix}$$

$$Av = \begin{bmatrix} v_1 * a_{1,1} + v_2 * a_{1,2} + v_3 * a_{1,3} + \dots + v_m * a_{1,n} \\ v_1 * a_{2,1} + v_2 * a_{2,2} + v_3 * a_{2,3} + \dots + v_m * a_{2,n} \\ \dots \\ v_1 * a_{m,1} + v_2 * a_{m,2} + v_3 * a_{m,3} + \dots + v_m * a_{m,n} \end{bmatrix}$$

Let's look at a specific example.

$$A = \begin{bmatrix} 2 & 4 & 6 \\ 3 & 5 & 7 \\ 1 & 2 & 3 \\ 8 & 6 & 2 \end{bmatrix}$$

$$v = \begin{bmatrix} -2 \\ 1 \\ 3 \end{bmatrix}$$

Solution:

$$= \begin{bmatrix} -2*2+1*4+3*6 \\ -2*3+1*5+3*7 \\ -2*1+1*2+3*3 \\ -2*8+1*6+3*2 \end{bmatrix}$$

$$= \begin{bmatrix} 18 \\ 20 \\ 9 \\ -4 \end{bmatrix}$$

$$= (18, 20, 9, -4)$$

Exercise 1 Vector Matrix Multiplication

Multiply the array A with the vector v. Compute this by hand, and make sure to show your computations.

Working Space

$$A = \begin{bmatrix} 1 & -2 & 3 & 5 \\ -4 & 2 & 7 & 1 \\ 3 & 3 & -9 & 1 \end{bmatrix}$$

$$v = \begin{bmatrix} 2 \\ 2 \\ 6 \\ -1 \end{bmatrix}$$

Answer on Page 35

2.1.1 Vector-Matrix Multiplication in Python

Most real-world problems use very large matrices where it becomes impractical to perform calculations by hand. As long as you understand how matrix-vector multiplication is done,

you'll be equipped to use a computing language, like Python, to do the calculations for you.

Create a file called `vectors_matrices.py` and enter this code:

```
// import the python module that supports matrices
import numpy as np

// create an array
a = np.array([[5, 1, 3, -2],
              [1, -1, 8, 4],
              [6, 2, 1, 3]])

// create a vector
b = np.array([1, 2, 3, -8])

//calculate the dot product
print(a.dot(b))
```

When you run it, you should see:

```
[16, 6, 8]
```

2.2 Where to Learn More

Watch this video from Khan Academy about matrix-vector products: https://rb.gy/frga5

Chapter 3

Vector Spans and Independence

A vector span is the collection of vectors obtained by scaling and combining the original set of vectors in all possible proportions. Formally, if the set $S = \{v_1, v_2, ..., v_n\}$ contains vectors from a vector space V, then the span of S is given by:

$$\text{Span}(S) = \{a_1v_1 + a_2v_2 + ... + a_nv_n : a_1, a_2, ..., a_n \in \mathbb{R}\} \tag{3.1}$$

This means that any vector in the $\text{Span}(S)$ can be written as a linear combination of the vectors in S.

Vector spans have practical applications in a number of fields. Computer graphics and physics are two of them. For example, in space travel, knowing the vector span is essential to calculating a slingshot maneuver that will give spacecraft a gravity boost from a planet. For this, you'd need to know the gravity vector of the planet relative to the sun and the velocity vectors that characterize the spacecraft. Engineers would use this information to figure out the trajectory angle that would allow the spacecraft to achieve a particular velocity in the desired direction.

3.1 Vector Independence

A set of vectors $S = \{v_1 v_2 ... v_n\}$ is linearly independent if the only solution to the equation $a_1 v_1 + a_2 v_2 + ... + a_n v_n = 0$.

is $a_1 = a_2 = ... = a_n = 0$. This means that no vector in the set can be written as a linear combination of the other vectors.

If there exists a nontrivial solution (i.e., a solution where some $a_i \neq 0$), then the vectors are said to be linearly dependent. This means that at least one vector in the set can be written as a linear combination of the other vectors.

The concept of vector independence is fundamental to the study of vector spaces, bases, and rank. You'll learn more about these concepts in future modules.

3.1.1 Dependent Vectors

Let's start by looking at two vectors.

$$v_1 = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$
$$v_2 = \begin{bmatrix} -14 \\ -28 \end{bmatrix}$$

These two vectors are dependent because $v_2 = -7 * v_2$. This is an obvious example but let's show it mathematically. If linearly independent, the two vectors must satisfy:

$$v_1 a_1 + v_1 a_2 = 0$$
$$v_2 a_1 + v_2 a_2 = 0$$

which is:

$$2a_1 - 14a_2 = 0$$
$$4a_1 - 28a_2 = 0$$

To solve, multiply the top equation by -2 and add it to the bottom:

$$2a_1 - 14a_2 = 0$$
$$0 + 0 = 0$$

The bottom equation drops out. Now olve for a_1 in the remaining equation:

$$a_1 = -7a_2$$

As you can see, one vector is a multiple of another.

$$a_1 \neq a_2 \neq 0$$

3.1.2 Independent Vectors

Let's see if these two vectors are independent.

$$v_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$v_2 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

To be independent,the two vectors must satisfy:

$$v_1 a_1 + v_1 a_2 = 0$$

$$v_2 a_1 + v_2 a_2 = 0$$

which is:

$$\begin{bmatrix} a_1 + 0 * a_2 \\ 0 * a_1 + a_2 \end{bmatrix}$$

So: $a_1 = a_2 = 0$ These vectors are not only independent, but they are orthogonal (perpendicular) to one another. You'll learn more about orthogonality later.

Here is an example whose solution isn't as obvious. You can solve using Gaussian elmination.

$$v_1 = [21]$$

$$v_2 = [1 - 6]$$

Rewrite as a system of equations:

$$a_1 * 2 + a_2 * 1 = 0$$

$$a_1 * 1 + a_2 * (-6) = 0$$

First swap the equations to that the the top equation has a coefficient of 1 for a_1:

$$a_1 - 6a_2 = 0$$

$$2a_1 + a_2 = 0$$

Next multiply row 1 by -2 and add it to row 2:

$$a_1 - 6a_2 = 0$$

$$0 - 11a_2 = 0$$

Multiply row 2 by 1 divided by 11.

$$a_1 - 6a_2 = 0$$

$$0 + a_2 = 0$$

Back substitute a_2 solution into the first equation:

$$a_1 = 0$$

$$a_2 = 0$$

Therefore

$$a_1 = a_2 = 0$$

and the two vectors are linearly independent.

Exercise 2 Vector Independence

Are these vectors independent?

Working Space

$$[214]$$

$$[2 - 12]$$

$$[01 - 2]$$

Show your work.

Answer on Page 35

3.2 Checking for Linear Independence Using Python

One way to use python to check for linear independence is to use the linalg.solve() function to solve the system of equations. You need to create an array that contains the coefficients of the variable and a vector that contains the values on the right-side of each

equation. So far, you've either been given equations that equal 0 or you've manipulated each equation to be equal to 0.

Let's first see how to use python to solve the equations in the previous exercise. If the equations are linearly independent, then $a_1 = a_2 = a_3 = 0$

Create a file called linear_independence.Python and enter this code:

```
import numpy as np

A = np.array([[2, 2, 0],
              [1, -1, 1],
              [4, 2, -2]])
b = np.array([0, 0, 0])
c = np.linalg.solve(A,b)
print(c)
```

Your should get this result, which shows the equations are linearly independent.

```
[ 0., -0.,  0.]
```

But what happens if the equations are not independent? Let's make the first two equations dependent by making equation 1 two times equation 2. Enter this code into your file:

```
import numpy as np

D = np.array([[2, -2, 2],
              [1, -1, 1],
              [4, 2, -2]])
e = np.array([0, 0, 0])
f = np.linalg.solve(D,f)
print(f)
```

You should get many lines indicating an error. Among the spew, you should see:

```
raise LinAlgError("Singular matrix")
```

So while the linalg.solve() function is quite useful for solving a system of independent linear equations, raising an error is not the most elegant way to figure out if the equations are dependent. That's where the concept of a determinant comes in. You'll learn about that in the next section. But for now, let's use the linalg.solve() function to find a solution for a set of equations known to be linearly independent.

$$4x_1 + 3x_2 - 5x_3 = 2$$

$$-2x_1 - 4x_2 - 5x_3 = 5$$

$$8x_2 + 8x_3 = -3$$

You will create a matrix that contains all the coeffients and a vector that contains the values on the right-side of the equations.

Enter this code into your file.

```
G = np.array([[4, 3, -5],
              [-2, -4, 5],
              [8, 8, 0]])
h = np.array([2, 5, -3])

j = np.linalg.solve(G, h)
print(j)
```

You should get this answer:

```
[2.20833333, -2.58333333, -0.18333333]
```

3.3 Determinants

The determinant of a matrix is a scalar value that can be calculated for a square matrix. If a matrix has linearly dependent rows or columns, the determinant is 0. One way to figure out if a set of equations are independent is to calculate the determinant.

***REVISE TO INCORPORATE THIS CONCEPT: determinant tells you what happens to the volume of a shape when it goes through a linear transformation

For a matrix that is 2 by 2, the calculation is easy:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

the determinant is:

$$\det(A) = (a * d) - (b * c)$$

For a larger matrix, finding the determinant is more complex and requires breaking down the matrix into smaller matrices until you reach te 2x2 form. The process is called expansion by minors. For our purposes, we simply want to first check to see if a matrix contains linearly independent rows and columns before using our Python code to solve. Modify your code so that is uses the $np.linalg.det()$ function. If the determinat is not zero, then you can call the $np.linalg.solve()$ function. Your code should look like this:

```
if (np.linalg.det(D) != 0):
    j = np.linalg.solve(D,e)
    print(j)
else:
     print("Rows and columns are not independent.")
```

3.4 Where to Learn More

Watch this video on *Linear Combinations and Vector Spans from Khan Academy*: http://rb.gy/g1snk

If you curious about the *Expansion of Minors,* see: https://mathworld.wolfram.com/DeterminantExpansionbyMinors.html

Chapter 4

Matrices

You've already had experience with matrices earlier in this module and also when you"'ve used spreadsheets. In this chapter you'll learn the types of matrices and get an introduction to some of the special matrices used for various calculations.

As you know, a matrix is a rectangular array of numbers arranged in rows and columns. The individual numbers in the matrix are called elements or entries. Matrices can be described by their dimensions. For example, a matrix with 2 rows and 3 columns is a 2 by 3 matrix.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

More generally, a matrix with m rows and n columns is referred to as an $m \times n$ matrix or simply an m-by-n matrix, and m and n are its dimensions.

The general form of a 2×3 matrix A is:

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

4.1 Types of Matrices

Matrices can be described by their shape:

- **Row Matrix:** has only one row.
- **Column Matrix:** has only one column.
- **Square Matrix:** has the same number of rows and columns.
- **Rectangular Matrix:** has an unequal number of rows and columns.

They can also be described by their unique numerical properties. Special matrices tht come in handy for certains types of computations. These are a few of the most common special matrices.

- **Zero Matrix:** contains only entries that are zero.
- **Identity Matrix:** sometimes called the unit matrix, is a square matrix whose diagonal entries are 1 and all other entries are 0.
- **Symmetric Matrix:** a square matrix that equals its transpose.
- **Diagonal Matrix:** has nonzero elements on the main diagonal, but all other elements are zero
- **Triangular Matrix:** This is a special square matrix that can be upper triangular or lower triangular. If upper, the main diagonal and all entries above it are nonzero while the lower entries are all zero. If lower, the main diagonal and all the entries below it are nonzero while the upper entries are all zero.

4.1.1 Symmetric Matrices

If you want to find out if a square matrix is symmetric, you need to transpose it. If the transpose is equal to the original matrix, then the matrix is symmetric.

To transpose a matrix, flip it over its diagonal so that the rows and columns are switched, like this:

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

After transposing:

$$A^T = \begin{bmatrix} a_{1,1} & a_{2,1} & a_{3,1} \\ a_{1,2} & a_{2,2} & a_{3,2} \\ a_{1,3} & a_{2,3} & a_{3,3} \end{bmatrix}$$

Note that A^T means the transpose of A.

Let's see how this works for the following square matrix, A.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}$$

Switch the rows and columns to get the transpose:

$$A^T = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}$$

Notice that $A = A^T$, so the matrix is symmetric.

What about matrix B?

$$B = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 7 & 8 & 9 \end{bmatrix}$$

Switch the rows and columns to get the transpose:

$$B = \begin{bmatrix} 1 & 2 & 7 \\ 2 & 4 & 8 \\ 3 & 5 & 9 \end{bmatrix}$$

Note that $B \neq B^T$. So B is not symmetrical.

Exercise 3 Matrix Transposition

Find the transpose of this matrix. Is it symmetric?

Working Space

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

Answer on Page 36

4.1.2 Creating Matrices in Python

****REVISE TO INCLUDE: np.diag() function for working with diagonal matrices.

Create a file called `matrices_creation.py` and enter this code:

```
// import the python module that supports matrices
import numpy as np
// Use the function np.array to define a matrix that
// contains specific values that you supply.
A = np.array([[ 5, 1, 3],
              [ 1, -1, 8],
              [ 6, 2, 1]])
// The transpose function returns
A.transpose()
```

When you run it, you should see:

```
array([[ 5, 1, 6],
       [ 1, -1, 2],
       [ 3, 8, 1]])
```

As you can see, $A \neq A^T$ so A is not symmetric. Try another:

```
// create a matrix, B
B = np.array([[ 5, 1, 6],
              [ 1, -1, 2],
              [ 6, 2, 1]])
B.transpose()
```

When you run it, you should see:

```
array([[ 5, 1, 6],
       [ 1, -1, 2],
       [ 6, 2, 1]])
```

B is symmetric. You can actually transpose any matrix using this function. But a matrix cannot be symmetric unless it is square.

Try this code to see what happens when you transpose a rectangular matrix.

```
// create a matrix, B
J = np.array([[ 5, 1, 3, 0],
              [ 1, -1, 8, 11],
              [ 6, 2, 1,-7]])
J.transpose()
```

Note that transposing a rectangular matrix changes its dimension from 3 by 4 to 4 by 3. You should see a transposed matrix, but it's not symmetric.

```
array([[ 5,  1,  6],
       [ 1, -1,  2],
       [ 3,  8,  1],
       [ 0, 11, -7]])
```

4.1.3 Creating Special Matrices in Python

Use the same file to add this code for creating a zero matrix.

```
// create an 8 by 10 Zero matrix.
C = np.zeros((8, 10))
C
```

When you run it, you should see:

```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

Add the following code to create an 8 by 8 Identity matrix.

```
// create an 8 by 8 Identity matrix
D = np.eye(8)
D
```

When you run it, you should see:

```
array([[1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0.],
```

```
       [0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1.]])
```

Appendix A

Answers to Exercises

Answer to Exercise 1 (on page 19)

$$Av = (1137 - 43)$$

Answer to Exercise 2 (on page 24)

Rewrite as a system of equations:

$$\begin{aligned} 2 * a_1 + 2 * a_2 + 0 * a_3 &= 0 \\ 1 * a_1 - 1 * a_2 + 1 * a_3 &= 0 \\ 4 * a_1 + 2 * a_2 - 2 * a_3 &= 0 \end{aligned}$$

Simplify

$$\begin{aligned} 2a_1 + 2 * a_2 &= 0 \\ a_1 - a_2 + a_3 &= 0 \\ 4a_1 + 2a_2 - 2a_3 &= 0 \end{aligned}$$

Swap row 2 and 1:

$$\begin{aligned} a_1 - a_2 + a_3 &= 0 \\ 2a_1 + 2 * a_2 &= 0 \\ 4a_1 + 2a_2 - 2a_3 &= 0 \end{aligned}$$

Multiply row 1 by -2 and add to row 2:

$$\begin{aligned} a_1 - a_2 + a_3 &= 0 \\ 0 + 3 * a_2 - 2a_3 &= 0 \\ 4a_1 + 2a_2 - 2a_3 &= 0 \end{aligned}$$

Multiply row 1 by -4 and add to row 3:

$$\begin{aligned} a_1 - a_2 + a_3 &= 0 \\ 0 + 3 * a_2 - 2a_3 &= 0 \\ 0 + 6a_2 - 6a_3 &= 0 \end{aligned}$$

Multiply row 2 by -4 and add to row 3:

$$\begin{aligned} a_1 - a_2 + a_3 &= 0 \\ 0 + 3 * a_2 - 2a_3 &= 0 \\ 0 + 0 - 2a_3 &= 0 \end{aligned}$$

Multiply row 3 by -1 and add to row 2:

$$\begin{aligned} a_1 - a_2 + a_3 &= 0 \\ 0 + 3 * a_2 + 0 &= 0 \\ 0 + 0 - 2a_3 &= 0 \end{aligned}$$

Divide row 3 by -2 and row 2 by $\frac{1}{3}$:

$$\begin{aligned} a_1 - a_2 + a_3 &= 0 \\ 0 + a_2 + 0 &= 0 \\ 0 + 0 + a_3 &= 0 \end{aligned}$$

Backsubstitute a_2 and a_3 into row 1:

$$\begin{aligned} a_1 + 0 + 0 &= 0 \\ 0 + a_2 + 0 &= 0 \\ 0 + 0 + a_3 &= 0 \end{aligned}$$

Therefore

$$a_1 = a_2 = a_3 = 0$$

.

Answer to Exercise 3 (on page 31)

$$A = A^t = \begin{bmatrix} 3 & -2 & 4 \\ -2 & 6 & 2 \\ 4 & 2 & 3 \end{bmatrix}$$