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 will 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, as well as 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, such as 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 been doing algebra already?"

You have! You have come a long way in your problem solving journey. You have used algebra to solve simple equations like $7x + 10 = 24$ and quadratic equations like $4x^2 + 9x + 31 = 0$. What distinguishes linear algebra is the focus on linear combinations. Any equation with a power greater than 1, such as a quadratic, is nonlinear.

We will 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 operations 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. After that, 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. 0 is black, 255 is white, and 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.
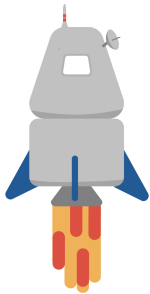
Flipping is a linear operation.
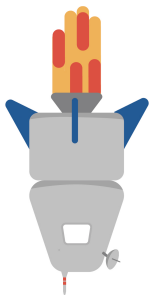
Next, the image is rotated 90 degrees. This rotation is linear, but if you want to rotate it at an angle that isn't a multiple of 90, you would need trigonometry. This would be treading into nonlinear territory, but that happens in the field of linear algebra. You will learn about nonlinear extensions later, which use trigonometric functions and imaginary numbers.
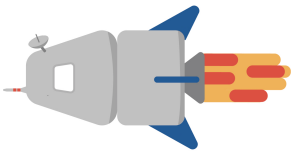
Inversion is an interesting linear operation that involves redefining the red, green, and blue values, such that the new value is 1.0 minus the old value. The resulting black background gives the impression the rocket is in deep space, don't you think?
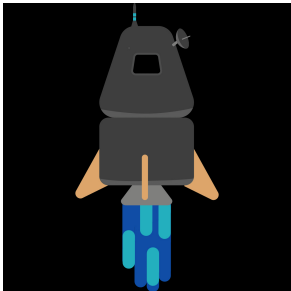
It is possible to redefine the red, green, and blue values in many ways. Visit the NASA website and search for false color images. NASA and other scientists redefine colors to communicate such things as the amount of vegetation or water in an area, the temperatures of the sun's surface, and so on. Photographers often do this for artistic effect. For example, the image on the left was taken with an infrared camera. (This infrared is not the same as thermal kind you have likely seen before. This is the infrared that is 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 will see a few matrices in this section. Let's first look at 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 use 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. Most of the Python methods we 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.

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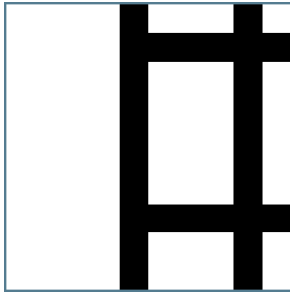\end{bmatrix}
$$

When converted to an image, it is very tiny. This is an enlarged version, so you can see the pattern.
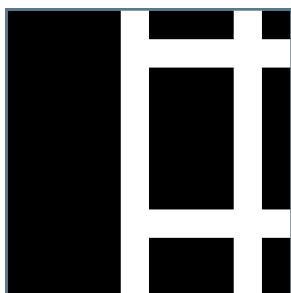
We can create an inverse of this image by changing all the values in the matrix so that 0 becomes 1 and 1 becomes 0. (Technically this is not the way you would invert a matrix, as you will see in the Python section. For this example, you will get the same visual result, but when you start inverting matrices programmatically, you will learn the formal definition.)

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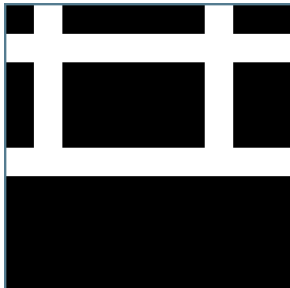1. & 1. & 1. & 1. & 0. & 1. & 1. & 1. & 0. & 1
\end{bmatrix}
$$

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

Rotating the original matrix by 90 degrees gives this:

$$\begin{bmatrix} 0. & 1. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \\ 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 0. & 1. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \\ 0. & 1. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \\ 0. & 1. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \\ 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \end{bmatrix}$$
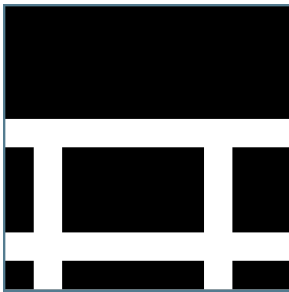
This is the resulting enlarged image:

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 is easy to multiply the matrix by 255.

Let's call this matrix f.

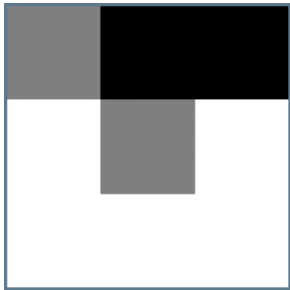$$\begin{bmatrix} 1.0 & 0.5 & 0.0 \\ 1.0 & 1.0 & 1.0 \\ 0.5 & 0.0 & 0.0 \end{bmatrix}$$

When multiplied by 255 and converted to a grayscale image:
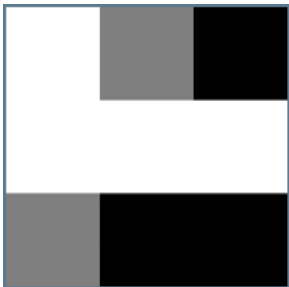
Let's call this matrix g:

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

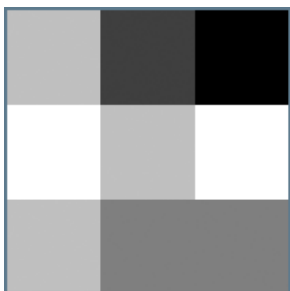When multiplied by 255 and converted to a grayscale image:

When we add f and g we get k:

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

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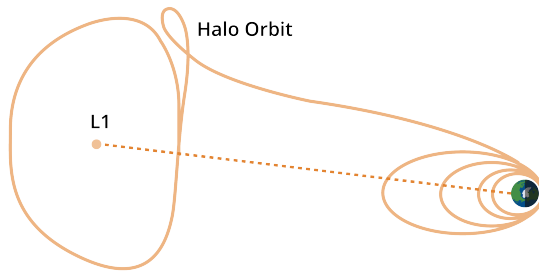
Aditya's engineers had to calculate a looping maneuver that would precisely inject the Aditya spacecraft into the Halo orbit. They determined the angles and burn times for the thrust engine. If they were wrong in one direction, the spacecraft would fly off to the sun. The other direction would send the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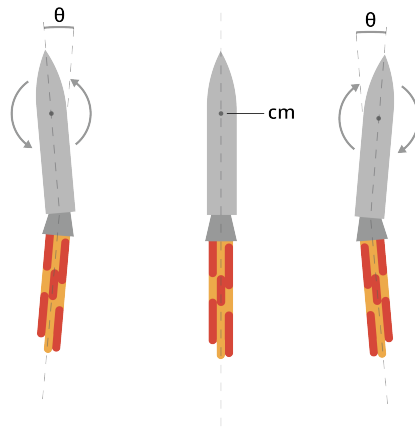
1.7   Images in Python

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
```

```python
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.

This is a draft chapter from the Kontinua Project. Please see our website (https://kontinua. org/) for more details.

Answers to Exercises