
TASK

# Discrete Maths

Visit our website

## INTRODUCTION

It's imperative that you understand enough maths to grapple with some of the more advanced concepts in this course. Let's get started!

If calculus is the engine of machine learning, linear algebra is its body. We use vectors and matrices to represent our various models. We will begin by taking a look at the basics and use linear algebra to extend our basic example of linear regression.

## LINEAR ALGEBRA

Let's start with a simpler question: what is algebra? If you think back to your high school days, algebra is just replacing numbers with letters. If `x + 3 = 8`, then what is `x`? Well, according to what we learn in algebra, it should be `5`. We do this kind of stuff every day in our own programming: we define variables to hold values and use that to achieve our goals.

Linear algebra extends regular algebra by making `x` not just a single number, but rather a whole bunch of numbers. This is what we call **matrices** and **vectors**.

### Vectors

A vector is a list of numbers. In programming terms, you can think of a vector as an array of values. This is just a way of representing data with multiple values. You have used this many times without even realising it!

For the purposes of this task, we will use matrix notation to represent our vectors:

$$\mathbf{V} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}.$$

There are two different aspects to look at with this vector: the number of elements, and which way it goes. In this case, there are three elements, so we call this a **3-vector**. Because it's going down, like a column, we call it a **column vector**. Let's look at the column vector's sibling, the **row vector**:

$$\mathbf{V} = \begin{pmatrix} v_1 & v_2 & v_3 \end{pmatrix}.$$

How do we represent vectors in Python? A useful library for scientific programming is Numerical Python (NumPy).

## NumPy

**NumPy** is a numerical computation library that is very commonly used in the field of data science. The code in this library has been **vectorized**. This is just a fancy way of saying that, while the code runs on the CPU, it is highly optimised. For example, running `np.sum()` on an array of values is much faster than making your own function to sum the values up. It also helps you to convert maths equations to code much more easily.

Let's have a look at how we represent vectors with NumPy.

```
import numpy as np

# Create a vector as a row

vector_row = np.array([1, 2, 3])

# Create a vector as a column

vector_column = np.array([[1],

                          [2],

                          [3]])

print('Vector row \n', vector_row)

print('Vector column \n', vector_column)
```

Output:

```
Vector row

 [1 2 3]

Vector column

 [[1]

 [2]

 [3]]
```

And that's all you need to know about vectors for now! See, maths really is easy!

## Matrices

What is a matrix? According to the movie *The Matrix*, it's something that hacks real life and gives you superpowers. Sadly, this is nothing like how matrices work in maths.

A matrix is a set of values written up to represent something. You can think of it as an Excel spreadsheet, with rows and columns. Or, framing it as a programming construct, you can think of it as a multi-dimensional array. It is typically written in the form of:

$$\mathbf{X} = \begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \\ x_{3,1} & x_{3,2} \end{pmatrix}.$$

Now, this is a lot to unpack. Firstly, the standard notation for matrices in equations is a capital bold letter. In this case, **X** is our matrix name. You can see that **X** is made up of a bunch of different values $x_{i,j}$. In this notation, $i$ denotes which row we are in, and $j$ denotes which column we are in.

When looking at matrices, it is important to know the shape of the matrix. We write this as $R \times C$, where $R$ is the number of rows, and $C$ is the number of columns. In this case, we have a $3 \times 2$ matrix. When dealing with matrix operators, knowing this is incredibly important.

**One important note**: not all matrices have only two dimensions. In fact, you can have as many dimensions or as few dimensions as possible. In some (highly complex) mathematics, there have been matrices existing with an infinite number of dimensions. In the context of machine learning, a multi-dimensional array is referred to as a **tensor**. Tensors are particularly important in deep learning, where they are used to represent and manipulate data in neural networks. A matrix with one dimension is usually called a **vector**. You can even have a matrix with no dimensions. This is just a single number. In linear algebra, we call this a **scalar**.

Let's create some matrices with NumPy.

```
import numpy as np

# Create a scalar (0-dimensional array)

scalar = np.array(5)

# Create a vector (1-d array)

vector = np.array([1, -2, 6])    # row vector by default

# Create a matrix 2x3 (2-d array)

matrix = np.array([[1, 2, 3], [4, 5, 6]])

print('Scalar \n', scalar)

print('Vector \n', vector)

print('Matrix \n', matrix)
```

Output:

```
Scalar

 5

Vector

 [ 1 -2  6]

Matrix

 [[1 2 3]

 [4 5 6]]
```

### Vectors vs matrices

Is an apple a fruit? Yes, it absolutely is. Now, is a fruit an apple? Well, not necessarily. It is, by definition, some type of fruit, but not necessarily an apple, as not all fruits are apples.

The same concept applies to vectors and matrices. A vector is a type of matrix with one dimension. That means that all vectors are matrices. However, not all matrices are vectors, as a matrix with more than one dimension is not a vector.

What's important to remember is that everything that applies to matrices applies to vectors, but not necessarily the other way around.

## LINEAR OPERATIONS

Okay, so now we know the data we are working with, let's look at some things we can do with this. A quick note on these operations: some operations require compatibility between matrices.

### Addition and subtraction

Let's start with the easy one. How do we add or subtract two matrices from one another?

**Compatibility:** Both matrices need to have the same shape. If Matrix **A** is $3 \times 2$, then Matrix **B** must also be $3 \times 2$.

Keeping this in mind, let's define our Matrices **A** and **B**:

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

$$\mathbf{B} = \begin{matrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \end{matrix}$$

Now, let's say we want to create Matrix **C**, which is **A** + **B**. This new matrix will have the same shape as the first two, and $c_{i,j} = a_{i,j} + b_{i,j}$ for all indices. Therefore, we get:

$$\mathbf{A} + \mathbf{B} = \begin{pmatrix} (a_{1,1} + b_{1,1}) & (a_{1,2} + b_{1,2}) \\ (a_{2,1} + b_{2,1}) & (a_{2,2} + b_{2,2}) \\ (a_{3,1} + b_{3,1}) & (a_{3,2} + b_{3,2}) \end{pmatrix}.$$

The same principle applies for subtraction.

Let's look at an example of addition and subtraction using NumPy.

```
import numpy as np

# A and B are matrices with the same shape

A = np.array([[6, 12, 5], [3, 0, 14]])

B = np.array([[9, -2, 4], [4, -1, -5]])

# Addition: A + B

C = A + B

print("C = A + B \n", C)

# Subtraction: A - B

D = A - B

print("D = A - B \n", D)
```

Output:

```
C = A + B

 [[15 10  9]

 [ 7 -1  9]]

D = A - B

 [[-3 14  1]

 [-1  1 19]]
```

### Scalar multiplication

If you remember above, a scalar is just a single number. In the world of linear algebra, we denote a scalar variable as a simple $x$ (lowercase, not bold). So scalar multiplication, in English terms, is just multiplying a matrix with a single number. Let's say we want to multiply our Matrix **A** with $x$. We write this as:

$$x\mathbf{A}$$

See why the notation is important? Keeping to the notation makes it easy to know which symbol refers to our scalar variable, and matrix in the expression $x\mathbf{A}$.

**Compatibilities:** No compatibilities are required.

To multiply **A** by a scalar, you just need to multiply each element in **A** with the scalar. This makes it:

$$x\mathbf{A} = \begin{pmatrix} x(a_{1,1}) & x(a_{1,2}) \\ x(a_{2,1}) & x(a_{2,2}) \\ x(a_{3,1}) & x(a_{3,2}) \end{pmatrix}.$$

Let's use NumPy to see scalar multiplication in action.

```
import numpy as np

# Matrix A

A = np.array([[6, 12, 5], [3, 0, 14]])

# Print matrix A

print("A: \n", A)

# Multiply the matrix A by the scaler 2

print("2*A: \n", 2*A)
```

Output:

```
A [[ 6 12  5]

 [ 3  0 14]]

2*A:

 [[12 24 10]

 [ 6  0 28]]
```

And this is all there is to it!

## Dot product

The **dot product**, also known as the **inner product**, is a way to multiply two vectors or matrices to produce a scalar or another matrix. For two vectors, the dot product is the sum of the products of their corresponding components. In the case of matrices, the dot product involves multiplying rows from one matrix by columns from another, resulting in a new matrix.

**Compatibilities:** For two matrices **A** and **B** to be compatible with the dot product, you need to look at their shapes. If Matrix **A** has a shape $x \times y$, then Matrix **B** *must* have a shape $y \times z$. The dot product will then have a shape $x \times z$.

This compatibility simply means that Matrix **B** needs to have a number of rows equal to the number of columns in Matrix **A**. When making the dot product, you will see why.

Let's update our example of defining **A** to something like:

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \\ a_{4,1} & a_{4,2} \end{pmatrix}.$$

But now, to add a twist, our **B** is:

$$\mathbf{B} = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{pmatrix}$$

Let's first see if they are compatible. Matrix **A** is of shape $4 \times 2$ and Matrix **B** is of shape $2 \times 3$. Does **B** have the same number of rows as **A** has columns? Yes. Therefore, they are compatible. We also know that the resulting matrix will be of shape $4 \times 3$.

So, now we need to compute the matrix. Let's say we are creating a Matrix **C** such that $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, where $\cdot$ is our operator for the dot product. Then,

$$\begin{aligned} c_{1,1} &= (a_{1,1} \cdot b_{1,1}) + (a_{1,2} \cdot b_{2,1}) \\ c_{1,2} &= (a_{1,1} \cdot b_{1,2}) + (a_{1,2} \cdot b_{2,2}) \\ &\dots \end{aligned}$$

and so on.

This can be a lot to wrap our heads around, so just think of it this way: the element of **C** at position `1,1` is each element of **A** in **row 1** multiplied with its corresponding

element in **B** at **column 1**. Now you can see why it's important that **B** must have the same number of rows as **A** has columns: this means that there will always be something to multiply together. For a full definition of the dot product:

$$\mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} (a_{1,1}b_{1,1} + a_{1,2}b_{2,1}) & (a_{1,1}b_{1,2} + a_{1,2}b_{2,2}) & (a_{1,1}b_{1,3} + a_{1,2}b_{2,3}) \\ (a_{2,1}b_{1,1} + a_{2,2}b_{2,1}) & (a_{2,1}b_{1,2} + a_{2,2}b_{2,2}) & (a_{2,1}b_{1,3} + a_{2,2}b_{2,3}) \\ (a_{3,1}b_{1,1} + a_{3,2}b_{2,1}) & (a_{3,1}b_{1,2} + a_{3,2}b_{2,2}) & (a_{3,1}b_{1,3} + a_{3,2}b_{2,3}) \\ (a_{4,1}b_{1,1} + a_{4,2}b_{2,1}) & (a_{4,1}b_{1,2} + a_{4,2}b_{2,2}) & (a_{4,1}b_{1,3} + a_{4,2}b_{2,3}) \end{pmatrix}$$

Easy, right? Takes a bit of getting used to, but you will get to understand it more when we use it in practice.

At least you don't need to do the calculations, as NumPy does it for you.

```
import numpy as np

# Define matrices G and H

G = np.array([[10, 3], [-1, 4], [5, 12], [2, 6]])

H = np.array([[2, 9, 0], [-2, 4, 1]])

print(f"The shape of G is {G.shape}")

print(f"The shape of H is {H.shape}")

# Compute dot product of G and H

dot_prod_G_H = np.dot(G, H)

print("Dot Product of matrices G and H: \n", dot_prod_G_H)

print(f"The shape of this new matrix is {dot_prod_G_H.shape}")
```

Output:

```
The shape of G is (4, 2)
The shape of H is (2, 3)
Dot Product of matrices G and H:

 [[ 14 102   3]

 [-10   7   4]

 [-14  93  12]

 [ -8  42   6]]

The shape of this new matrix is (4, 3)
```

**Please note:** The dot product is not mutable. This means that $\mathbf{A} \cdot \mathbf{B}$ is **not** the same as $\mathbf{B} \cdot \mathbf{A}$.

For some more ease, let's look at the dot product of two vectors. From earlier, you should remember that a vector is a one-dimensional matrix. Let's define two vectors, **P** and **Q**:

$$\mathbf{P} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix}.$$

$$\mathbf{Q} = \begin{pmatrix} q_1 & q_2 & q_3 & q_4 \end{pmatrix}.$$

Here, you'll notice that **P** is a 3-vector and **Q** is a 4-vector. A key difference is that P is a column vector, while **Q** is a row vector. Since vectors can be represented as matrices (with one dimension), we can write P as a $3 \times 1$ matrix and **Q** as a $1 \times 4$ matrix. These matrices are compatible for matrix multiplication because the number of columns in **P** matches the number of rows in **Q**. As a result, their dot product will yield a matrix with the shape $3 \times 4$, which represents the product of the row from **P** and the column from **Q**. Their dot product is:

$$\mathbf{P} \cdot \mathbf{Q} = \begin{pmatrix} p_1 q_1 & p_1 q_2 & p_1 q_3 & p_1 q_4 \\ p_2 q_1 & p_2 q_2 & p_2 q_3 & p_2 q_4 \\ p_3 q_1 & p_3 q_2 & p_3 q_3 & p_3 q_4 \end{pmatrix}.$$

In NumPy it looks like this.

```
import numpy as np

# Define vectors P and Q
P = np.array([[2],

              [6],

              [0]])

Q = np.array([[-2, 0, 6, 3]])

# Compute dot product of P (3x1) and (1x4)
dot_prod_P_Q = np.dot(P, Q)

print('Dot Product of vectors P and Q: \n', dot_prod_P_Q)
```

Output:

```
Dot Product of vectors P and Q

[[ -4   0  12   6]

 [-12   0  36  18]

 [  0   0   0   0]]
```

### Hadamard product

The Hadamard product, also known as the element-wise product or Schur product, is a binary operation performed on matrices. It involves multiplying the corresponding elements of two matrices together to produce a new matrix of the same size. It is referred to as an element-wise product of two matrices, meaning that each element in the resulting matrix is the product of the corresponding elements in the two original matrices. Unlike the dot product, the Hadamard product does not require any specific conditions for the number of rows or columns, except that the two matrices must have the same dimensions.

**Compatibility:** For two matrices **A** and **B** to be compatible with the Hadamard product they need to be of the same size.

Consider the following two matrices **A** and **B**:

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

The Hadamard product A∘B is calculated by multiplying each corresponding element:

$$A \circ B = \begin{bmatrix} 1 \times 5 & 2 \times 6 \\ 3 \times 7 & 4 \times 8 \end{bmatrix} = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$$

The Hadamard product is often used in Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) within neural networks. It plays a crucial role in element-wise operations between matrices, which is key to how these models process and retain information over time, especially when handling sequential data.

### Transpose

Transpose is an operator that works on a single matrix. The general formula for transpose is easy to define. But first, let's get some facts straight about transpose:

**Properties:** If Matrix **A** is composed with a shape of $x \times y$, its transpose, denoted $\mathbf{A^T}$, will have dimensions $y \times x$.

This means you just flip it over. Mathematically, we say that the element $a'_{i,j}$ in $\mathbf{A^T}$ is just defined as:

$$a'_{i,j} = a_{j,i}.$$

Let's try an example with actual values:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

That means that:

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

```python
import numpy as np
# Matrix A
A = np.array([[6, 12, 5], [3, 0, 14]])
# Transpose matrix A
A_transpose = np.transpose(A)
print("A:\n", A)
print("A_transpose:\n",A_transpose)
```

Output:

```
A:
 [[ 6 12  5]
 [ 3  0 14]]
A_transpose:
 [[ 6  3]
 [12  0]
 [ 5 14]]
```

The transpose of any row vector is a column vector, and the transpose of any column vector is a row vector.

When working with NumPy, a vector can be represented as a one-dimensional (1D) array. However, a 1D array doesn't have a clear distinction between a row vector and a column vector because it lacks a second dimension. If you try to transpose this vector using **`np.transpose(vector)`** or **`vector.T`**, the result will still be a 1D array. The transpose operation has no effect because NumPy doesn't know if you want to treat this 1D array as a row or column vector.

To clearly define whether the vector is a row or a column, you can reshape it explicitly. Once the vector is reshaped, transposing will have a clear and expected effect.

```python
import numpy as np

vector = np.array([1, 2, 3])

print("Shape of vector:", vector.shape)

# If you try to transpose this vector using .transpose before reshaping, the result will still be a 1D array:

vector_transposed = np.transpose(vector)

print("Vector Transposed:", vector_transposed)

print("Shape of vector_transposed:", vector_transposed.shape)

# To clearly define whether the vector is a row or a column, you have to reshape it, and then transpose.

# Reshaping and transposing:

# Row Vector: Reshaping

row_vector = np.array([1, 2, 3]).reshape(1, -1)  # Shape becomes (1, 3)

print("Row Vector:", row_vector)

print("Shape of row_vector:", row_vector.shape)

# Row Vector: Transposing

row_vector_transposed = np.transpose(row_vector)

print("Row Vector Transposed:", row_vector_transposed)

print("Shape of row_vector_transposed:", row_vector_transposed.shape)

# Column Vector: Reshaping

column_vector = np.array([1, 2, 3]).reshape(-1, 1)  # Shape becomes (3, 1)

print("Column Vector:", column_vector)

print("Shape of column_vector:", column_vector.shape)

# Column Vector: Transposing

column_vector_transposed = np.transpose(column_vector)

print("Column Vector Transposed:", column_vector_transposed)

print("Shape of column_vector_transposed:", column_vector_transposed.shape)
```

Output:

```
Shape of vector: (3,)
Vector Transposed: [1 2 3]
Shape of vector_transposed: (3,)
Row Vector: [[1 2 3]]
Shape of row_vector: (1, 3)
Row Vector Transposed: [[1]
 [2]
 [3]]
Shape of row_vector_transposed: (3, 1)
Column Vector: [[1]
 [2]
 [3]]
Shape of column_vector: (3, 1)
Column Vector Transposed: [[1 2 3]]
Shape of column_vector_transposed: (1, 3)
```

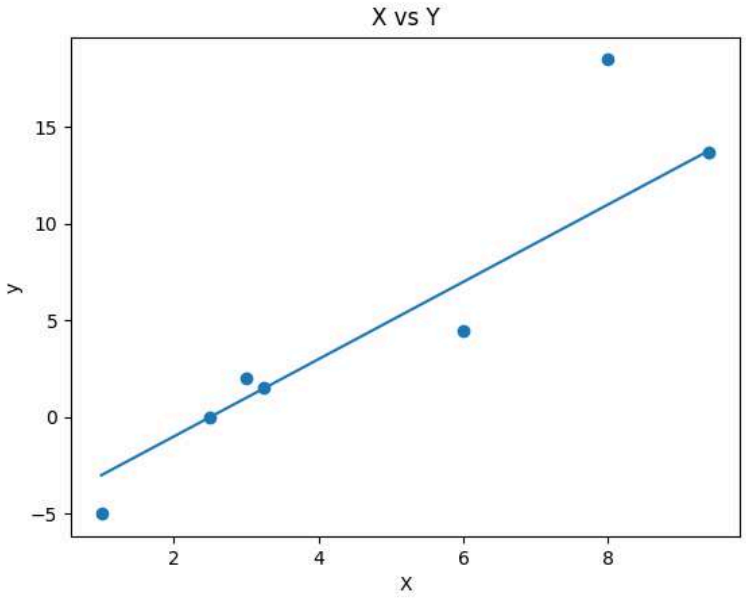
## MULTIPLE LINEAR REGRESSION

The equation for a straight line is an important aspect of linear regression, as it allows us to model the relationship between multiple variables, make predictions, and extract valuable insights from their collective impact.

Commonly represented as $f(x) = mx + c$ or $y = mx + c$, the equation for a straight line estimates the optimal values of the slope and y-intercept to best fit the data points.

In this equation, $f(x)$ (or $y$) represents the dependent variable we aim to predict, $x$ represents the independent variable, $m$ denotes the slope illustrating the change in the dependent variable for each unit change in the independent variable, and $c$ represents the y-intercept.

We would probably want to make a line through the data that looks something like:




The y-intercept (where the line intercepts the y-axis) looks like it is roughly at `y=−5`. So, now we have two choices: we could calculate, derive and create an update rule for each and every parameter, or we could do this for a single parameter vector (containing all parameters) using linear algebra. Keep in mind, this line could get more complicated with more parameters. A typical machine learning algorithm can sometimes have millions of parameters: that's a lot of maths!

This is where linear algebra comes in handy. We just need to define a way of representing this data using vectors and matrices, and then we can extend this to any size of vector and matrix. If you look at our equation for a straight line, you might notice something similar: this is the equation for the **dot product**! Don't believe it? Watch this:

$$f(x) = \begin{pmatrix} m & c \end{pmatrix} \cdot \begin{pmatrix} x \\ 1 \end{pmatrix}.$$

Because the first matrix is of shape $1 \times 2$ and the second matrix is of shape $2 \times 1$, this will result in a matrix of shape $1 \times 1$ – that is just a single number! And what is this single number? Well, it is $(mx) + (1c)$, which is our y-value.

Crazy, right? This is why linear algebra is so useful in machine learning. However, we can do one better: we can calculate all values of $y$ using a single calculation! Let's change our definition a bit: we have a matrix **X** of all our $x$ values:

$$\mathbf{X} = \begin{pmatrix} x_1 & x_2 & x_3 \\ 1 & 1 & 1 \end{pmatrix}.$$

Don't worry about the **1**'s at the bottom: you will see where they factor in for now. Let's define our **parameter matrix P** as the matrix containing our $m$ and $c$ values as a row vector:

$$\mathbf{P} = \begin{pmatrix} m & c \end{pmatrix}.$$

We can obtain our **Y** values using:

$$\mathbf{Y} = \mathbf{P} \cdot \mathbf{X}.$$

Let's take a closer look at this equation:

$$\mathbf{Y} = \begin{pmatrix} m & c \end{pmatrix} \times \begin{pmatrix} x_1 & x_2 & x_3 \\ 1 & 1 & 1 \end{pmatrix}.$$

**P** is a $1 \times 2$ matrix, and **X** is a $2 \times 3$ matrix. This means that **Y** will end up being a $1 \times 3$ matrix. This is perfect, because there are three $x$ values, which means that there should be three $y$ values. When we apply the formula for dot product, we get:

$$\begin{pmatrix} m & c \end{pmatrix} \times \begin{pmatrix} x_1 & x_2 & x_3 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} (mx_1 + c) & (mx_2 + c) & (mx_3 + c) \end{pmatrix}$$

Wow, that is a powerful notation! You can represent so much using so little. This means that if **X** is a $2 \times n$ matrix, you will get $y$ values in a shape of $1 \times n$, which means that you can have as many $x$ values as you want!

A quick note on the **1**'s in the X array: this is sometimes just called the **bias**. This is because there is a *y*-intercept that isn't multiplied with any $x$ values.

# Instructions

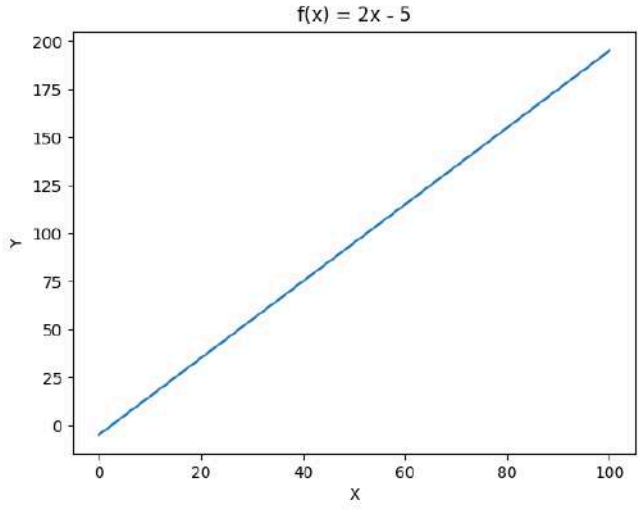
You will need NumPy to tackle this task. To become familiar with NumPy, please explore the code examples above and **numpy_example.ipynb**.

## Practical task

Open up **linear_algebra.py** in your folder. You will see code that generates a range of **X** values and gives you **P**, the parameter vector. Using what you have learned in linear algebra to:

1. In the given Python file, plot the graph of $f(x) = \mathbf{P} \cdot \mathbf{X}$.

   Your graph should look something like this:




**Important:** Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

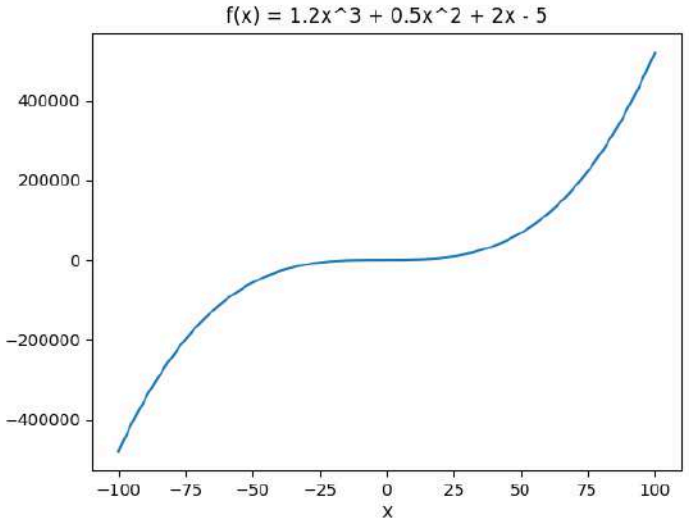
## Challenge

Use this opportunity to extend yourself by completing an optional challenge activity.

The powerful thing about this form of expression is its extensibility. For example, we can move from a linear equation to a **quadratic equation** or even a **cubic equation**! Open up **challenge_task.py**; you should see some similar code to that for previous practical task. The main difference is that the X values now range from -100 to 100, and the parameter vector now has four values.

1. In the given Python file, plot the graph of $f(x) = \mathbf{P} \cdot \mathbf{X}$. If all went well in the first task, this shouldn't warrant any changes to your code.

   Your resultant graph should look something like this:



## Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

## REFERENCES

IEEE. (1993). *IEEE Standards Collection: Software Engineering*. IEEE Standard 610.12-1990.