

```
In [1]: # we will need matplotlib for plotting
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
```

Linear algebra: (Chapter 2 of MML Book by Deisenroth et al)

1. Linear algebra is the study of vectors and certain rules to manipulate vectors.

Equation of a 2D line

There are two equations of lines

1. $y = mx + c$
2. $ax + by + c = 0$

What are the advantages or disadvantages of both of those?

A 2D line is the set of all points (x, y) that satisfy the an equation $ax + by + c = 0$ for given a, b, c . For the same line there can be multiple valid parameters a, b, c . Either a or b can be zero. But both a and b cannot be zero at the same time.

Ideally the equation of line must be written in the set notation:

$$\mathcal{L}(a, b, c) = \{(x, y) : ax + by + c = 0, x \in \mathbb{R}, y \in \mathbb{R}\}$$

This equation is read as: the line \mathcal{L} defined by the paramters a, b, c is the set of all points (x, y) such that it the x, y satisfy the equation $ax + by + c = 0$ and x and y are in the set of all real numbers.

Implicit form

Line is a type of curve. Other curves can be parabola, circle etc. Every curve can be represented in implicit form like we did for the line above. In implicit form, the points are *constrained* by one or more equations to lie on the curve. The equations define a test whether the point lies on the curve or not.

$$\mathcal{C}(p_1, p_2, \dots, p_n) = \{(x, y) : f(x, y; p_1, p_2, \dots, p_n) = 0, x \in \mathbb{R}, y \in \mathbb{R}\}$$

This equation is read as the curve \mathcal{C} defined by the parameters p_1, p_2, \dots, p_n is the set of all points (x, y) such that it the x, y satisfy the equation $f(x, y; p_1, p_2, \dots, p_n) = 0$ and x and y are in the set of all real numbers.

Take circle with center (x_0, y_0) and radius r , as an example. The implicit form is:

$$\mathcal{C}(x_0, y_0, r) = \{(x, y) : (x - x_0)^2 + (y - y_0)^2 - r^2 = 0, x \in \mathbb{R}, y \in \mathbb{R}\}$$

Parametric form of 2D Line

A parameteric form defines how the points on the curve are generated from a free parameters. For a line:

$$\mathcal{L}(d_x, d_y, x_0, y_0) = \{(d_x r + x_0, d_y r + y_0) : r \in \mathbb{R}\}$$

This equation is read as: the line \mathcal{L} defined by the parameters d_x, d_y, x_0, y_0 is the set of all points $(d_x r + x_0, d_y r + y_0)$ such that r is any real number.

Take circle with center (x_0, y_0) and radius r , as an example. The parameteric form is:

$$\mathcal{C}(x_0, y_0, r) = \{(r \cos(\theta) + x_0, r \sin(\theta) + y_0) : \theta \in [0, 2\pi)\}$$

In general, the parameteric form of a curve depend on some free parameters and the points are defined as functions of the free parameters.

Linear system of equations and Matrices

$$4x_1 + 4x_2 = 5 \quad (1)$$

$$2x_1 - 4x_2 = 1 \quad (2)$$

 No description has been provided for this image

The equation

$$\underbrace{\begin{bmatrix} 4 & 4 \\ 2 & -4 \end{bmatrix}}_{A \in \mathbb{R}^{2 \times 2}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}_{\mathbf{x} \in \mathbb{R}^2} = \underbrace{\begin{bmatrix} 5 \\ 1 \end{bmatrix}}_{\mathbf{b} \in \mathbb{R}^2}, \quad (3)$$

can be written in matrix form as,

$$A\mathbf{x} = \mathbf{b}$$

In general a system of m equations with n unknowns,

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \quad (4)$$

$$\vdots \quad (5)$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \quad (6)$$

can be written in matrix form as

$$\underbrace{\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}}_{A \in \mathbb{R}^{m \times n}} \underbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}}_{\mathbf{x} \in \mathbb{R}^n} = \underbrace{\begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}}_{\mathbf{b} \in \mathbb{R}^m}. \quad (7)$$

Or

$$A\mathbf{x} = \mathbf{b}$$

Vectors

We will denote vectors with bold font notations instead of the usually \vec{x} notation. The arrow notation vectors are sometimes called geometric vectors. We will make no such distinction. The set of all real numbers will be denoted as \mathbb{R} . The set of all real 2D vectors is written as \mathbb{R}^2 . When we write $\mathbf{x} \in \mathbb{R}^2$, it means that \mathbf{x} is in the set of real 2D vectors, hence a 2D real vector. We will write $\|\mathbf{x}\|$ for the magnitude of the vector, and $\mathbf{x} \cdot \mathbf{y}$ for dot product between two vector \mathbf{x} and \mathbf{y} .

n-D vector

A n-D vector is also written as $\mathbf{x} \in \mathbb{R}^n$ and the vector has $\mathbf{x} = [x_1; \dots; x_n]$ n real components. Every vector has a magnitude $\|\mathbf{x}\|$ and a direction $\hat{\mathbf{x}}$. The magnitude and direction are given by:

$$\|\mathbf{x}\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \in \mathbb{R}$$

$$\hat{\mathbf{x}} = \frac{1}{\|\mathbf{x}\|} \mathbf{x} \in \mathbb{R}^n$$

The direction vector $\hat{\mathbf{x}}$ is a unit vector because its magnitude is one i.e. $\|\hat{\mathbf{x}}\| = 1$.

```
In [27]: x = np.array([2, 3, 4, 5, 9, 10])
          xmag = np.linalg.norm(x)
```

```
xunit = x / xmag

# For many many x vectors you can do this:
xs = np.array([[2, 3, 4, 5, 9, 10],
               [-3, 2, -5, 4, -10, 9]]) # 2 x 6
xmags = np.linalg.norm(xs, axis=-1, keepdims=True) # 2 x 1
xunits = xs / xmags # 2 x 6
print(xunit, xunits)
```

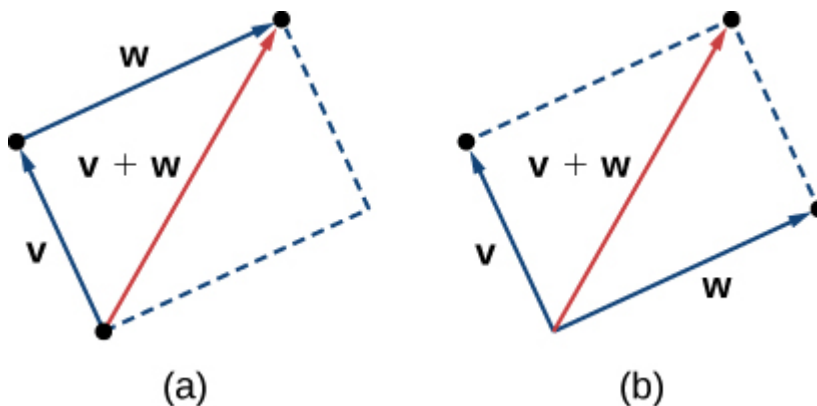
```
[0.13046561 0.19569842 0.26093123 0.32616404 0.58709527 0.65232807] [[ 0.13046561 0.19569842 0.26093123 0.32616404 0.58709527 0.65232807]
 [-0.19569842 0.13046561 -0.32616404 0.26093123 -0.65232807 0.58709527]]
```

Vector addition

Vector addition is element-wise addition

$$\mathbf{v} + \mathbf{w} = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} + \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} v_1 + w_1 \\ \vdots \\ v_n + w_n \end{bmatrix}$$

Geometrically the resulting vector can be obtained by triangle law or the parallelogram law.



Reference: [1]

```
In [31]: v = np.arange(0, 10)
w = np.arange(10, 20)
v+w
```

```
Out[31]: array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28])
```

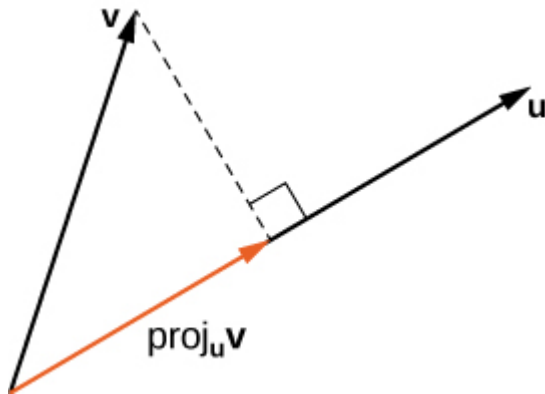
Dot product of vectors

Dot product of two vectors is a scalar given by sum of element-wise product.

$$\mathbf{v} \cdot \mathbf{u} = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix} = v_1 u_1 + v_2 u_2 + \cdots + v_n u_n$$

Geometrically, dot product is closely related to the projection. Projection of vector \mathbf{v} on \mathbf{u} is the dot product of \mathbf{v} with the direction of \mathbf{u}

$$\text{proj}_{\mathbf{u}} \mathbf{v} = \mathbf{v} \cdot \hat{\mathbf{u}}$$



Dot product of vector with itself gives the square of the magnitude $\mathbf{v} \cdot \mathbf{v} = \|\mathbf{v}\|^2$.

Reference: [2]

```
In [36]: v = np.arange(10)
         u = np.arange(10, 20)
         v @ u # v . u
```

Out[36]: 735

```
In [37]: v @ u / np.linalg.norm(u)
```

Out[37]: 15.723948706640511

```
In [39]: np.allclose(v @ v , np.linalg.norm(v)**2)
```

Out[39]: True

Matrices

Matrices are a group of vectors. A matrix can be obtained by vertical stacking of row (horizontal) vectors or horizontal stacking of column (vertical) vectors. It is common to represent all vectors as column vectors unless specified otherwise, so we consider a matrix \mathbf{V} as a horizontal concatenation of column vectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$. Let each vector be m -dimensional $\mathbf{v}_i \in \mathbb{R}^m$.

$$\mathbf{V} = [\mathbf{v}_1 \quad \mathbf{v}_2 \quad \dots \quad \mathbf{v}_n] = \begin{bmatrix} \mathbf{v}_1[1] & \mathbf{v}_2[1] & \dots & \mathbf{v}_n[1] \\ \mathbf{v}_1[2] & \mathbf{v}_2[2] & \dots & \mathbf{v}_n[2] \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{v}_1[m] & \mathbf{v}_2[m] & \dots & \mathbf{v}_n[m] \end{bmatrix}$$

Such a matrix is said to be a $m \times n$ matrix. It is also written as $\mathbf{V} \in \mathbb{R}^{m \times n}$.

```
In [41]: v1 = np.arange(5)
v2 = np.arange(5, 10)
v3 = np.arange(10, 15)
V = np.hstack((v1[:, None], v2[:, None], v3[:, None]))
V
```

```
Out[41]: array([[ 0,  5, 10],
 [ 1,  6, 11],
 [ 2,  7, 12],
 [ 3,  8, 13],
 [ 4,  9, 14]])
```

Linear functions

What are linear functions?

Definition of Linear combination (2.11 in MML book)

Given m vectors $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m \in \mathbb{R}^n$, a linear combination of vectors is defined as, $\mathbf{v} := \alpha_1 \mathbf{x}_1 + \alpha_2 \mathbf{x}_2 + \dots + \alpha_n \mathbf{x}_m$ for every any scalar $\alpha_1, \alpha_2, \dots, \alpha_n \in \mathbb{R}$

Definition of Linear function

A function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is linear in \mathbf{x} when

1. $f(\alpha \mathbf{x}) = \alpha f(\mathbf{x})$ for all $\alpha \in \mathbb{R}$ and $\mathbf{x} \in \mathbb{R}^n$
2. $f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y})$ for all $\alpha, \beta \in \mathbb{R}$ and $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

The above two properties can be combined in a single test

$$f(\alpha \mathbf{x} + \beta \mathbf{y}) = \alpha f(\mathbf{x}) + \beta f(\mathbf{y})$$

Examples of linear functions

1. $f([x_1, x_2]) = 3x_1 + 2x_2$
2. $f([x_1, x_2, x_3]) = 4(3x_1 + 2x_2) + x_3$
3. $f([x_1, x_2, x_3], [w_1, w_2, w_3]) = w_1 x_1 + w_2 x_2 + w_3 x_3$

$$4. f([x_1, x_2]) = \exp(\log(3x_1 + 3x_2))$$

Homework 3. Problem 1: Show that the above functions are linear.

Examples of non-linear functions

1. $f([x_1, x_2]) = x_1^2$
2. $f([x_1, x_2]) = x_1^3$
3. $f([x_1, x_2]) = \sin(x_1)$
4. $f([x_1, x_2]) = \exp(x_1)$
5. $f([x_1, x_2]) = \log(x_1)$

Homework 3. Problem 2: Show that the above functions are non-linear.

```
In [42]: def f(x):
          return 4*(3*x[0] + 2*x[1]) + x[2]
x = np.random.rand(3)
y = np.random.rand(3)
alpha = np.random.rand()
beta = np.random.rand()
np.allclose(f(alpha * x + beta * y) ,
            alpha * f(x) + beta * f(y))
```

Out[42]: True

Properties of Matrices

Transpose of a Matrix

Transpose of a matrix \mathbf{V} (denoted as \mathbf{V}^\top) is an operation that swaps rows with columns and columns with rows. For example, the transpose of the above matrix will make it a vertical concatenation of row vectors.

$$\mathbf{V}^\top = \begin{bmatrix} \mathbf{v}_1^\top \\ \mathbf{v}_2^\top \\ \vdots \\ \mathbf{v}_n^\top \end{bmatrix} = \begin{bmatrix} \mathbf{v}_1[1] & \mathbf{v}_1[2] & \dots & \mathbf{v}_n[m] \\ \mathbf{v}_2[1] & \mathbf{v}_2[2] & \dots & \mathbf{v}_n[m] \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{v}_n[1] & \mathbf{v}_n[2] & \dots & \mathbf{v}_n[m] \end{bmatrix}$$

If $\mathbf{V} \in \mathbb{R}^{m \times n}$, then $\mathbf{V}^\top \in \mathbb{R}^{n \times m}$. The two dimensions get swapped.

```
In [44]: V = np.arange(12).reshape(3, 4)
          V, V.T
```

```
Out[44]: (array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]]),
          array([[ 0,  4,  8],
                 [ 1,  5,  9],
                 [ 2,  6, 10],
                 [ 3,  7, 11]]))
```

Tranpose of a column vector

All vectors are also matrices. By convention, all vectors are considered column matrices and hence called column vectors. A n -D vector $\mathbf{v} = [v_1; \dots; v_n] \in \mathbb{R}^n$ is by convention considered a $n \times 1$ column matrix i.e. $\mathbf{v} \in \mathbb{R}^{n \times 1}$.

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \in \mathbb{R}^{n \times 1}$$

The transpose of a column vector is a row vector

$$\mathbf{v}^T = [v_1 \quad v_2 \quad \dots \quad v_n] \in \mathbb{R}^{1 \times n}$$

Row vectors are always denoted with a tranpose of their corresponding column vector.

```
In [45]: v = np.arange(5).reshape(5, 1)
          v.shape, v.T.shape
```

```
Out[45]: ((5, 1), (1, 5))
```

Matrix addition

You can add two matrices element-wise.

$$\underbrace{\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}}_{A \in \mathbb{R}^{m \times n}} + \underbrace{\begin{bmatrix} b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{bmatrix}}_{B \in \mathbb{R}^{m \times n}} := \underbrace{\begin{bmatrix} a_{11} + b_{11} & \dots & a_{1n} + b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & \dots & a_{mn} + b_{mn} \end{bmatrix}}_{A+B \in \mathbb{R}^{m \times n}}$$

The matrix addition is commutative $A + B = B + A$

```
In [46]: A = np.random.rand(3, 4)
          B = np.random.rand(3, 4)
          np.allclose(A + B, B + A)
```


Out[46]: True

Matrix-scalar product

You can multiply matrices with a scalar (a real number)

$$\alpha \underbrace{\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}}_{A \in \mathbb{R}^{m \times n}} := \underbrace{\begin{bmatrix} \alpha a_{11} & \dots & \alpha a_{1n} \\ \vdots & \ddots & \vdots \\ \alpha a_{m1} & \dots & \alpha a_{mn} \end{bmatrix}}_{\alpha A \in \mathbb{R}^{m \times n}} \quad (9)$$

Matrix scalar product is commutative

$$\alpha A = A \alpha$$

```
In [48]: alpha = np.random.rand()
A = np.random.rand(3, 4)
np.allclose(alpha * A, A * alpha)
```

Out[48]: True

Matrix-vector product

For those who know dot product, matrix-vector product is best defined as a collection of dot products. Define a matrix $\mathbf{V} \in \mathbb{R}^{m \times n}$ as a vertical concatenation of m n -dimensional row-vectors $\mathbf{v}_1^\top, \dots, \mathbf{v}_m^\top$

$$\mathbf{V} = \begin{bmatrix} \mathbf{v}_1^\top \\ \mathbf{v}_2^\top \\ \vdots \\ \mathbf{v}_m^\top \end{bmatrix}$$

The matrix $\mathbf{V} \in \mathbb{R}^{m \times n}$ can be multiplied by a n -dimensional column vector $\mathbf{u} \in \mathbb{R}^n$ with the product defined as the vector-wise dot product vertically concatenated to result in another column vector:

$$\mathbf{V}\mathbf{u} = \begin{bmatrix} \mathbf{v}_1 \cdot \mathbf{u} \\ \mathbf{v}_2 \cdot \mathbf{u} \\ \vdots \\ \mathbf{v}_m \cdot \mathbf{u} \end{bmatrix} \in \mathbb{R}^m$$

When $m = 1$, then $\mathbf{V} = \mathbf{v}_1^\top$ and the matrix product is $\mathbf{v}_1^\top \mathbf{u} = \mathbf{v}_1 \cdot \mathbf{u}$. Dot product between two vectors \mathbf{v} and \mathbf{u} is also written as $\mathbf{v}^\top \mathbf{u}$. Going forward we

will prefer $\mathbf{v}^\top \mathbf{u}$ notation for dot product instead of $\mathbf{v} \cdot \mathbf{u}$.

```
In [49]: V = np.random.rand(3, 4)
u = np.random.rand(4)
V @ u
```

```
Out[49]: array([0.63202228, 0.66688924, 1.10880745])
```

Matrix-matrix product

Matrix-matrix product between two matrices $\mathbf{V} \in \mathbb{R}^{m \times n}$ and $\mathbf{U} \in \mathbb{R}^{n \times p}$ can be defined in terms of matrix-vector product by writing \mathbf{U} as a horizontal concatenation of p n -dimensional column vectors $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_p$.

$$\begin{aligned}\mathbf{V}\mathbf{U} &= [\mathbf{V}\mathbf{u}_1 \quad \mathbf{V}\mathbf{u}_2 \quad \dots \quad \mathbf{V}\mathbf{u}_p] \in \mathbb{R}^{m \times p} \\ &= \begin{bmatrix} \mathbf{v}_1^\top \mathbf{u}_1 & \mathbf{v}_1^\top \mathbf{u}_2 & \dots & \mathbf{v}_1^\top \mathbf{u}_p \\ \mathbf{v}_2^\top \mathbf{u}_1 & \mathbf{v}_2^\top \mathbf{u}_2 & \dots & \mathbf{v}_2^\top \mathbf{u}_p \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{v}_m^\top \mathbf{u}_1 & \mathbf{v}_m^\top \mathbf{u}_2 & \dots & \mathbf{v}_m^\top \mathbf{u}_p \end{bmatrix}\end{aligned}$$

The result is a horizontal concatenation of matrix-vector products, where the left matrix \mathbf{V} gets multiplied with each column vector of right matrix \mathbf{U} .

Another interpretation is that the matrix-matrix product are all possible dot products between left matrices' row vectors with right matrices' column vectors.

Matrix-matrix product or short matrix products do not commute i.e $\mathbf{V}\mathbf{U} \neq \mathbf{U}\mathbf{V}$ in general.

Matrix-matrix product does follow distributive property $A(B + C) = AB + AC$.

```
In [50]: V = np.random.rand(3, 4)
U = np.random.rand(4, 5)
V @ U
```

```
Out[50]: array([[0.31447524, 0.73716028, 1.28026682, 0.99156111, 1.24665231],
                [0.21518341, 0.61803498, 0.57341321, 0.47848747, 0.44250388],
                [0.37677031, 0.32739954, 0.98070004, 0.65674803, 0.77542503]])
```

Identity matrix

$$\mathbf{I}_n = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

Square matrix

A square matrix is a matrix with number of rows equal to the number of columns.

Inverse of a square matrix

A matrix \mathbf{V}^{-1} is called the inverse of a square matrix \mathbf{V} if $\mathbf{V}^{-1}\mathbf{V} = \mathbf{V}^{-1} = \mathbf{I}_n$. The inverse of a square matrix exists only when it is singular i.e the determinant of the matrix is non-zero $\det(\mathbf{V}) \neq 0$.

```
In [51]: I = np.eye(10)
I
```

```
Out[51]: array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
                [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

Linearity of matrix multiplication

Consider a function $f_A : \mathbb{R}^n \rightarrow \mathbb{R}^m$ defined as

$$f_A(\mathbf{x}) := A\mathbf{x}$$

.

Show that the function f_A is linear.

Proof

The function $f_A(\mathbf{x})$ is linear if for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and $\alpha, \beta \in \mathbb{R}$,

$$f_A(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha f_A(\mathbf{x}) + \beta f_A(\mathbf{y})$$

$$A(\alpha\mathbf{x} + \beta\mathbf{y}) = A(\alpha\mathbf{x}) + A(\beta\mathbf{y})$$

Theorem

All linear functions of the form $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ can be written as matrix multiplications of the form $f(\mathbf{x}) = F\mathbf{x}$.

Proof:

Consider any given linear function $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ which is linear. Because of linearity it satisfies $f(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha f(\mathbf{x}) + \beta f(\mathbf{y})$.

Consider special vectors, called the standard basis vectors, that have only one element as 1 and the rest are 0.

$$\mathbf{e}_1 := [1, 0, \dots, 0]^\top \in \mathbb{R}^n$$

$$\mathbf{e}_2 := [0, 1, \dots, 0]^\top \in \mathbb{R}^n$$

$$\mathbf{e}_n := [0, 0, \dots, 1]^\top \in \mathbb{R}^n$$

Any vector $\mathbf{x} = [x_1, x_2, \dots, x_n]^\top \in \mathbb{R}^n$ can be written as a linear combination of standard basis vectors,

$$\mathbf{x} = x_1\mathbf{e}_1 + x_2\mathbf{e}_2 + \dots + x_n\mathbf{e}_n$$

Hence the function evaluation at \mathbf{x} , can also be written as the linear combination of function evaluations at standard basis vectors,

$$f(\mathbf{x}) = f(x_1\mathbf{e}_1 + x_2\mathbf{e}_2 + \dots + x_n\mathbf{e}_n)$$

$$f(\mathbf{x}) = x_1f(\mathbf{e}_1) + x_2f(\mathbf{e}_2) + \dots + x_nf(\mathbf{e}_n)$$

Note that $f(\mathbf{e}_1)$, $f(\mathbf{e}_2)$, are constants with respect to \mathbf{x} . So you can arrange them into a constant matrix, independent of \mathbf{x} ,

$$f(\mathbf{x}) = \underbrace{[f(\mathbf{e}_1) \quad f(\mathbf{e}_2) \quad \dots \quad f(\mathbf{e}_n)]}_F \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}}_{\mathbf{x}}$$

$$f(\mathbf{x}) = F\mathbf{x}$$

Theorem

Show that composition $f(g(x))$ of two linear functions, f and g is also a linear function.

Proof is left as Homework 3: Problem 3

Rank of a matrix

Using vectors for 2D line notation

```
In [14]: def stylizeax(ax, limits):
    """Set ax style"""
    minx, maxx, miny, maxy = limits
    # x-axis, y=0
    ax.annotate("",
                xy=(minx, 0),
                xytext=(maxx, 0),
                arrowprops=dict(arrowstyle="<->"),
                color='k')
    ax.text(maxx, 0, "x")
    # y-axis, x=0
    ax.annotate("",
                xy=(0, miny),
                xytext=(0, maxy),
                arrowprops=dict(arrowstyle="<->"),
                color='k')
    ax.text(0, maxy, "y")

    ax.grid(True, which='both') # show the grid
    ax.set_aspect('equal') # set aspect ratio of the grid to 1:1

def points_on_line(a, b, c, Npts=6, scale=10):
    """Generate points on the line ax + by + c = 0"""
    # ax + by + c = 0
    # In parametric form with free parameter r
    # (x, y) = (-b*r + x0, a*r + y0)
    # where
    # x0 = - a*c / (a*a + b*b)
    # y0 = - b*c / (a*a + b*b)
    # (x0, y0) is the point on the line closest to the origin
    uniformgrid = [i/Npts for i in range(-scale*Npts//2, scale*Npts//2, scale)]
    x0 = -a*c/(a*a + b*b)
    y0 = -b*c/(a*a + b*b)
    x = [-b*r + x0 for r in uniformgrid]
    y = [ a*r + y0 for r in uniformgrid]
    return x, y
```

Matplotlib

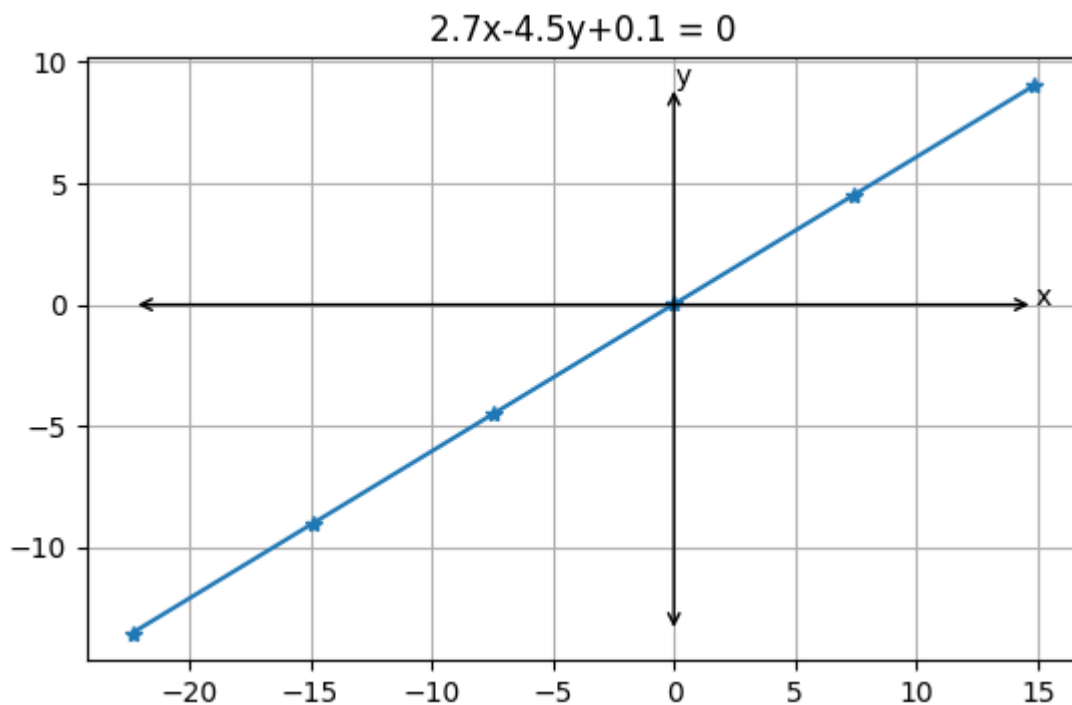
```
In [11]: # Plot a line  $ax + by + c = 0$ 
# a, b, c = 2.5, -1, -5 # pick numbers by hand

# pick a, b, c at random
import random
scale = 10
a, b, c = [scale*(random.random()-0.5) for _ in range(3)] # random number fr

# Generate some sample points on a line
x, y = points_on_line(a, b, c, scale=scale)

# Plot the points
fig, ax = plt.subplots()
stylizeax(ax, (min(x), max(x), min(y), max(y)))
ax.plot(x, y, '*-') # the line
ax.set_title(f'{a:.1f}x{b:+.1f}y{c:+.1f} = 0') # print the equation
```

Out[11]: Text(0.5, 1.0, '2.7x-4.5y+0.1 = 0')



We started from 2D linear models, but we want to work with N-D models where N can be even in thousands or millions. It makes sense to simplify the notation by using vector notation.

Recall that the implicit equation for a line is

$$\mathcal{L}(a, b, c) = \{(x, y) : ax + by + c = 0, x \in \mathbb{R}, y \in \mathbb{R}\}$$

We will represent a 2D point (x, y) by a 2D vector $\mathbf{x} = [x; y]$ and the parameters (a, b) with weight vector $\mathbf{w} = [a; b]$. Let's compute the dot product between the two newly defined vectors :

$$\mathbf{w} \cdot \mathbf{x} = ax + by$$

The equation of the line under new notation in full its full glory is

$$\mathcal{L}(\mathbf{w}, c) = \{\mathbf{x} : \mathbf{w} \cdot \mathbf{x} + c = 0, \mathbf{x} \in \mathbb{R}^2\}$$

Unique line notation

Same line $ax + by + c = 0$ can be represented by multiple equations for the same form. This representation of line is not unique. For example, equations $5x + 2y + 10 = 0$ and $10x + 4y + 20 = 0$ represent the same line. In general, for any real number $\alpha \neq 0$ all equations $\alpha ax + \alpha by + \alpha c = 0$ represent the same line. One can choose an arbitrary non-zero α for making the equation unique.

In vector notation, all non-zero $\alpha \mathbf{w} \cdot \mathbf{x} + \alpha c = 0$ represent the same line. One good candidate for α is $\alpha = \frac{1}{\|\mathbf{w}\|}$, because this changes $\alpha \mathbf{w}$ to $\hat{\mathbf{w}}$ a unit vector.

$$\mathcal{L}(\hat{\mathbf{w}}, w_0) = \{\mathbf{x} : \hat{\mathbf{w}} \cdot \mathbf{x} + w_0 = 0, \mathbf{x} \in \mathbb{R}^2\}$$

where $w_0 = \frac{c}{\|\mathbf{w}\|}$.

Geometric interpretation



No description has been provided for this image

The new equation of the line has a convenient geometric interpretation. Recall that $\hat{\mathbf{w}} \cdot \mathbf{x}$ is the projection of \mathbf{x} on $\hat{\mathbf{w}}$. In other words, the equation $\text{proj}_{\hat{\mathbf{w}}} \mathbf{x} + w_0 = 0$ constrains all vectors on the line to have a constant projection $\text{proj}_{\hat{\mathbf{w}}} \mathbf{x} = -w_0$.

This means that vector $\hat{\mathbf{w}}$ is perpendicular to the line and cuts the line a distance $|w_0|$ from the origin.

```
In [4]: import numpy as np # a vector algebra library

a = np.array([0, 1, 2, 3]) # a vector
print("a=", a)
b = np.array([4, 5, 6, 7]) # another vector
print("b=", b)
C = np.array([[0, 1, 2, 3],
              [4, 5, 6, 7]]) # A matrix
```

```
print("C=", C)
D = np.zeros((2, 4)) # a 2x4 matrix of zeros
print("D=", D)
E = np.random.rand(2,5) # Random 2x5 matrix of numbers between 0 and 1
print("E=", E)
```

```
a= [0 1 2 3]
b= [4 5 6 7]
C= [[0 1 2 3]
     [4 5 6 7]]
D= [[0. 0. 0. 0.]
     [0. 0. 0. 0.]]
E= [[0.23192693 0.44345236 0.21142332 0.27449444 0.41147911]
     [0.4699159  0.47266503 0.04658258 0.13593016 0.14999181]]
```

```
In [5]: print("a*0.1 = ", a * 0.1) # element-wise multiplication
print("C*0.2 = ", C * 0.2) # element-wise multiplication
print("a*b = ", a * b) # element-wise multiplication (Note: different from
print("a*b*0.2 = ", a * b * 0.2) # element-wise multiplication
print("C @ a = ", C @ a) # matrix-vector product
print("C.T = ", C.T) # matrix transpose
print("C.T @ D = ", C.T @ D) # matrix-matrix product
print("a * C = ", a * C) # so called broadcasting; numpy specific
```

```
a*0.1 = [0.  0.1 0.2 0.3]
C*0.2 = [[0.  0.2 0.4 0.6]
          [0.8 1.  1.2 1.4]]
a*b = [ 0  5 12 21]
a*b*0.2 = [0.  1.  2.4 4.2]
C @ a = [14 38]
C.T = [[0 4]
        [1 5]
        [2 6]
        [3 7]]
C.T @ D = [[0. 0. 0. 0.]
            [0. 0. 0. 0.]
            [0. 0. 0. 0.]
            [0. 0. 0. 0.]]
a * C = [[ 0  1  4  9]
          [ 0  5 12 21]]
```

Numpy: General Broadcasting Rules

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimension and works its way left. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1.

Otherwise a `ValueError` is raised

Ref: <https://numpy.org/doc/stable/user/basics.broadcasting.html>

In the following example, both the A and B arrays have axes with length one that are expanded to a larger size during the broadcast operation:

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):    7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

```
In [6]: A = np.random.rand(8, 1, 6, 1)
        B = np.random.rand(7, 1, 5)
        (A * B).shape # Returns the shape of the multi dimensional array
```

```
Out[6]: (8, 7, 6, 5)
```

Here are some more examples:

```
A      (2d array):  5 x 4
B      (1d array):    1
Result (2d array):  ?
```

```
A      (2d array):  5 x 4
B      (1d array):    4
Result (2d array):  ?
```

```
A      (3d array): 15 x 3 x 5
B      (3d array): 15 x 1 x 5
Result (3d array):
```

```
A      (3d array): 15 x 3 x 5
B      (2d array):   3 x 5
Result (3d array):  ?
```

```
A      (3d array): 15 x 3 x 5
B      (2d array):   3 x 1
Result (3d array):  ?
```

```
In [16]: def points_on_line(hatw, w0, Npts=6, scale=10):
        """ Generate some sample points on a line """
        assert hatw.shape == (2,) # only works for 2D
        perp_hatw = np.array([-hatw[1], hatw[0]])# vector perpendicular to hatw
        uniformgrid = np.linspace(-scale//2, scale//2, Npts)
        return perp_hatw * uniformgrid[:, None] - w0*hatw
```

```
In [17]: # Plot a line ax + by + c = 0
        scale = 10
        # a, b, c = [scale*(random.random()-0.5) for _ in range(3)] # random number
        # abc = scale*(np.random.rand(3)-0.5) # random number from -10 to 10
        abc = [3, 2, -6] # pick your favorite line
        w = abc[:2]
        hatw = w / np.linalg.norm(w) # What does np.linalg.norm do?
        w0 = abc[2] / np.linalg.norm(w)

        # Generate some sample points on a line
```

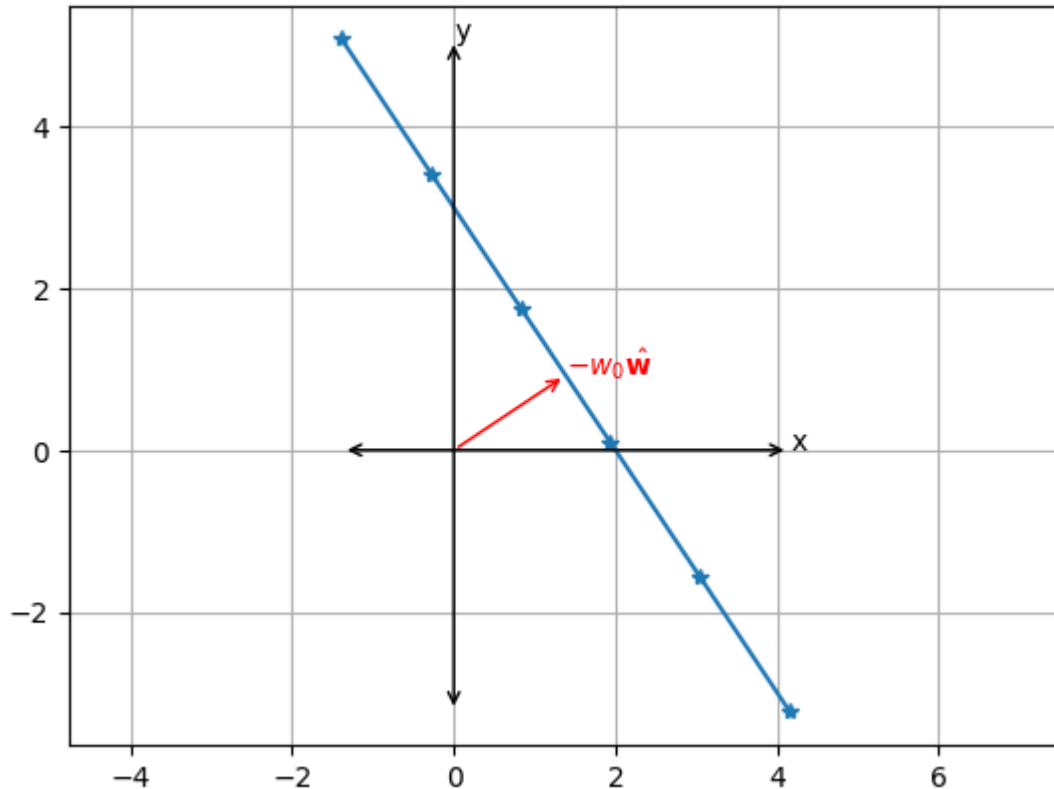
```

x = points_on_line(hatw, w0, Npts=6, scale=scale) # Npts x 2 array

# Plot the points
fig, ax = plt.subplots()
ax.axis('equal')
stylizeax(ax, (x[:, 0].min(), x[:, 0].max(), x[:, 1].min(), x[:, 1].max()))
ax.plot(x[:, 0], x[:, 1], '*-') # the line
pt0 = -w0*hatw
ax.annotate("", xytext=(0, 0), xy=(pt0[0], pt0[1]),
            arrowprops=dict(arrowstyle="->", color='r'))
ax.text(pt0[0], pt0[1], r"$-w_0\hat{w}$", color='r')

```

Out[17]: Text(1.3846153846153848, 0.9230769230769231, '\$-w_0\hat{w}\$')



Linear regression: review

Let's take the simple linear regression example from STS332 textbook (uploaded on brightspace; page 300; Table 6-1).

"As an illustration, consider the data in Table 6-1. In this table, y is the salt concentration (milligrams/liter) found in surface streams in a particular watershed and x is the percentage of the watershed area consisting of paved roads."

```

In [ ]: %%writefile saltconcentration.tsv
#Observation    SaltConcentration    RoadwayArea
1              3.8          0.19

```

2	5.9	0.15
3	14.1	0.57
4	10.4	0.4
5	14.6	0.7
6	14.5	0.67
7	15.1	0.63
8	11.9	0.47
9	15.5	0.75
10	9.3	0.6
11	15.6	0.78
12	20.8	0.81
13	14.6	0.78
14	16.6	0.69
15	25.6	1.3
16	20.9	1.05
17	29.9	1.52
18	19.6	1.06
19	31.3	1.74
20	32.7	1.62

```
In [ ]: # numpy can import text files separated by separator like tab or comma
salt_concentration_data = np.loadtxt("saltconcentration.tsv")
salt_concentration_data
```

```
In [ ]: # Plot the points
fig, ax = plt.subplots()
# Scatter plot using matplotlib
ax.scatter(salt_concentration_data[:, 2], salt_concentration_data[:, 1])
ax.set_xlabel(r"Roadway area %")
ax.set_ylabel(r"Salt concentration (mg/L)")
```

Least squares regression

The problem of linear regression is to find a line that "best fits" the given data. That is we want all the points $\{(x_1, y_1), \dots, (x_n, y_n)\}$ to satisfy the equation of the line $y = mx + c$. Since we know that there exists no such line, so we will try to make $y \approx mx + c$, by minimizing some error/distance/cost/loss function between y and $mx + c$ for every point (x_i, y_i) in the dataset. The simplest error function that results in nice answers is squared distance:

$$e(x_i, y_i) = (y_i - (mx_i + c))^2$$

Then we can minimize the total error to find the line:

$$m^*, c^* = \arg \min_{m, c} \sum_{i=1}^n e(x_i, y_i)$$

Geometrically, this error minimization corresponds to minimizing the stubs in the following figure:

Vectorization of Least square regression

Recall that the magnitude of a vector $\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$ has a similar form to the error function. This suggests that we can define an error vector with the signed error for each data point as it's elements

$$\mathbf{e} = \begin{bmatrix} y_1 - (mx_1 + c) \\ y_2 - (mx_2 + c) \\ \vdots \\ y_n - (mx_n + c) \end{bmatrix}$$

Minimizing the total error is same as minimizing the square of error vector magnitude

$$m^*, c^* = \arg \min_{m, c} \|\mathbf{e}\|^2$$

While we are at it let us define $\mathbf{x} = [x_1; \dots; x_n]$ to denote the vector of all x coordinates of the dataset and $\mathbf{y} = [y_1; \dots; y_n]$ to denote y coordinates. Then the error vector is:

$$\mathbf{e} = \mathbf{y} - (\mathbf{x}m + \mathbf{1}_n c)$$

where $\mathbf{1}_n$ is a n-D vector of all ones. Finally, we vectorize parameters of the line $\mathbf{m} = [m; c]$. We will also need to horizontally concatenate \mathbf{x} and $\mathbf{1}_n$. Let's call the result $\mathbf{X} = [\mathbf{x}, \mathbf{1}_n] \in \mathbb{R}^{n \times 2}$. Now, the error vector looks like this:

$$\mathbf{e} = \mathbf{y} - \mathbf{Xm}$$

Expanding the error magnitude:

$$\begin{aligned} \|\mathbf{e}\|^2 &= (\mathbf{y} - \mathbf{Xm})^\top (\mathbf{y} - \mathbf{Xm}) \\ &= \mathbf{y}^\top \mathbf{y} + \mathbf{m}^\top \mathbf{X}^\top \mathbf{Xm} - 2\mathbf{y}^\top \mathbf{Xm} \end{aligned}$$

Our minimization problem in vectorized form is:

$$\mathbf{m}^* = \arg \min_{\mathbf{m}} \mathbf{y}^\top \mathbf{y} + \mathbf{m}^\top \mathbf{X}^\top \mathbf{Xm} - 2\mathbf{y}^\top \mathbf{Xm}$$

This is a quadratic equation in \mathbf{m} that can be minimized by equating the derivate to zero.

Two rules of vector derivatives

There are two conventions in vector derivatives:

1. Gradient convention
2. Jacobian convention

Gradient convention

Under gradient convention the derivative of scalar-valued vector function $f(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$ is defined as vertical stacking of element-wise derivatives

$$\frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix} \in \mathbb{R}^n$$

Jacobian convention

Under gradient convention the derivative of scalar-valued vector function $f(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$ is defined as horizontal stacking of element-wise derivatives

$$\frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1} \quad \dots \quad \frac{\partial f(\mathbf{x})}{\partial x_n} \right] \in \mathbb{R}^{1 \times n}$$

For a vector-value vector function $\mathbf{f}(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}^m$, Jacobian of $\mathbf{f}(\mathbf{x})$ is the vertical concatenation of gradients transposed, resulting in $m \times n$ matrix

$$\mathbf{J}_{\mathbf{x}}(\mathbf{f}(\mathbf{x})) = \frac{\partial}{\partial \mathbf{x}} \mathbf{f}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}} \\ \dots \\ \frac{\partial f_m(\mathbf{x})}{\partial \mathbf{x}} \end{bmatrix}$$

We will use Jacobian convention in this course, because it works nicely with chain rule.

Derivative of a linear function

All scalar-valued linear functions of \mathbf{x} can be written in the form $f(\mathbf{x}) = \mathbf{b}^T \mathbf{x}$.

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{c}^T \mathbf{x} = \mathbf{c}^T \quad (10)$$

Derivative of a quadratic function

All scalar-valued homogeneous quadratic functions of \mathbf{x} can be written in the form $f(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x}$.

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{x}^T \mathbf{A} \mathbf{x} = \mathbf{x}^T (\mathbf{A} + \mathbf{A}^T) \quad (11)$$

Back to Least square regression

$$\mathbf{0}^T = \frac{\partial}{\partial \mathbf{m}} (\mathbf{y}^T \mathbf{y} + \mathbf{m}^T \mathbf{X}^T \mathbf{X} \mathbf{m} - 2 \mathbf{y}^T \mathbf{X} \mathbf{m}) \quad (12)$$

$$= 2 \mathbf{m}^{*T} \mathbf{X}^T \mathbf{X} - 2 \mathbf{y}^T \mathbf{X} \quad (13)$$

This gives us the solution

$$\mathbf{m}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

The symbol \mathbf{V}^{-1} is called inverse of matrix \mathbf{V} .

The term $(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T$ is also called the pseudo-inverse of a matrix \mathbf{X} , denoted as \mathbf{X}^\dagger .

```
In [ ]: n = salt_concentration_data.shape[0]
        bfx = salt_concentration_data[:, 2:3]
        bfy = salt_concentration_data[:, 1]
        bfX = np.hstack((bfx, np.ones((bfx.shape[0], 1))))
        bfX
```

```
In [ ]: bfm = np.linalg.inv(bfX.T @ bfX) @ bfX.T @ bfy
        print(bfm)
        bfm, *_ = np.linalg.lstsq(bfX, bfy, rcond=None)
        print(bfm)
```

```
In [ ]: m = bfm.flatten()[0]
        c = bfm.flatten()[1]

        # Plot the points
        fig, ax = plt.subplots()
        ax.scatter(salt_concentration_data[:, 2], salt_concentration_data[:, 1])
        ax.set_xlabel(r"Roadway area $\\%$")
        ax.set_ylabel(r"Salt concentration (mg/L)")
```

```
x = salt_concentration_data[:, 2]
y = m * x + c
# Plot the points
ax.plot(x, y, 'r-') # the line
```