

4101553 Métodos Numéricos aplicados a la Ingeniería Civil

Departamento de Ingeniería Civil
Universidad Nacional de Colombia
Sede Manizales

Docente: Juan Nicolás Ramírez Giraldo (jnramirezg@unal.edu.co)
(<mailto:jnramirezg@unal.edu.co>)

"Cum cogitaveris quot te antecedant, respice quot sequantur" (Séneca)

Repositorio de la asignatura (https://github.com/jnramirezg/metodos_numericos_ingenieria_civil/)

Unidad 1: Sistemas de ecuaciones lineales

Lo más básico de Python

```
In [1]: # Los comentarios se ponen con el símbolo numeral.  
''' También se pueden poner entre tres comillas sin afectar el código'''
```

```
Out[1]: ' También se pueden poner entre tres comillas sin afectar el código'
```

```
In [2]: 4+3 # Suma
```

```
Out[2]: 7
```

```
In [3]: 94-100 # Resta
```

```
Out[3]: -6
```

```
In [4]: 6*4 # Multiplicación
```

```
Out[4]: 24
```

```
In [5]: 7/8 # División
```

```
Out[5]: 0.875
```

```
In [6]: 9//4 # División entera
```

```
Out[6]: 2
```

```
In [7]: 9%4 # Residuo de la división
```

Out[7]: 1

In [8]: `2**10` # *Potencia, que en Matlab sería 2^10*

Out[8]: 1024

In [9]: `a = -10` # *Asignación de variables*

In [10]: `print(a)` # *Impresión dentro del código.*

-10

In [11]: `a` # *Impresión en consola.*

Out[11]: -10

In [12]: `a == -10` # *Condición de igualdad.*

Out[12]: True

In [13]: `a != -10` # *Condición de desigualdad.*

Out[13]: False

In [14]: `a == -10 or a != -10` # *Operador lógico 'o'.*

Out[14]: True

In [15]: `a == -10 and a != -10` # *Operador lógico 'y'*

Out[15]: False

In [16]: # *Condicional básico*
`if a == 100:`
 `print('Correcto')`
`else:`
 `print('Incorrecto')`

Incorrecto

In [17]: # *Ciclo for*
`for i in range(5):`
 `print(i)`

0
1
2
3
4

```
In [18]: # Ciclo while
contador = 0
while contador < 5:
    print(contador)
    contador += 1 # Que es equivalente a (contador = contador + 1),
                  # es decir, una reasignación recurrente de variable.
```

0
1
2
3
4

Matrices, modificación de matrices y operaciones

Un sistema de ecuaciones algebraicas lineales tiene la forma general:

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2$$

.

.

.

$$a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n$$

siendo a coeficientes constantes, b las constantes, n el número de ecuaciones y x las incógnitas.

Sistemáticamente se usa la notación matricial para facilitar su resolución, de la siguiente manera:

$$\underline{\underline{A}} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

donde cada elemento a_{ij} representa un elemento de la matriz (los coeficientes constantes).

Nota: De aquí en adelante se usará la siguiente nomenclatura.

M : matrices

V : vectores filas y vectores columnas

E : constantes y escalares

Las incógnitas x_j se representan en el vector columna:

$$\underline{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Y las constantes b_i con el vector columna:

$$\underline{B} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Que en síntesis es: $A \cdot X = B$

Formas de definir matrices

1. Lista de listas
2. Arreglos en numpy
3. Matrices en sympy

1. Lista de listas

Conceptos previos de listas

```
In [19]: h = 0 # Se define una variable cualquiera.
```

```
In [20]: mi_lista = [1, 'flor', 1.5, h] # lista con cualquier tipo de elemento.
```

```
In [21]: mi_lista
```

```
Out[21]: [1, 'flor', 1.5, 0]
```

```
In [22]: mi_lista_2 = [mi_lista, 100, 'flor'] # Una lista con una lista dentro.
```

```
In [23]: mi_lista_2
```

```
Out[23]: [[1, 'flor', 1.5, 0], 100, 'flor']
```

Ahora, definiendo matrices como lista de listas.

```
In [24]: matriz = [[5, 1, 2], [7, 3, 7], [4, 7, 8]] # Matriz como lista de listas.
```

```
In [25]: matriz
```

```
Out[25]: [[5, 1, 2], [7, 3, 7], [4, 7, 8]]
```

```
In [26]: # Una forma gráfica más cómoda.  
A = [  
    [5, 1, 2],  
    [7, 3, 7],  
    [4, 7, 8],  
    ]
```

```
In [27]: print(A)  
  
[[5, 1, 2], [7, 3, 7], [4, 7, 8]]
```

```
In [28]: # También se puede usar simplemente  
A
```

```
Out[28]: [[5, 1, 2], [7, 3, 7], [4, 7, 8]]
```

```
In [29]: # Tipo de dato  
type(A)
```

```
Out[29]: list
```

```
In [30]: # Tamaño de la matriz  
m = len(A)      # Número de filas de la matriz, o tamaño de la lista A.
```

```
In [31]: m
```

```
Out[31]: 3
```

```
In [32]: A[0]
```

```
Out[32]: [5, 1, 2]
```

```
In [33]: n = len(A[0]) # Número de columnas de la matriz, o tamaño de la primera l
```

```
In [34]: m
```

```
Out[34]: 3
```

```
In [35]: print('El número de filas de A es:')  
print(m)
```

```
El número de filas de A es:  
3
```

```
In [36]: print('El número de columnas de A es:')  
print(n)
```

```
El número de columnas de A es:  
3
```

In [37]: A

Out[37]: [[5, 1, 2], [7, 3, 7], [4, 7, 8]]

```
In [38]: # Llamado de un elemento
a00 = A[0][0]
```

```
In [39]: # Impresión de resultados.
print('El elemento de la fila 1 y la columna 1 es:')
print(a00)
```

El elemento de la fila 1 y la columna 1 es:
5

```
In [40]: # Llamado de un elemento
a22 = A[2][2]
a12 = A[1][2]

# Impresión de resultados.
print('\nEl elemento de la fila 3 y la columna 3 es:')
print(a22)
print('\nEl elemento de la fila 2 y la columna 3 es::')
print(a12)
```

El elemento de la fila 3 y la columna 3 es:
8

El elemento de la fila 2 y la columna 3 es::
7

In [41]: A

Out[41]: [[5, 1, 2], [7, 3, 7], [4, 7, 8]]

```
In [42]: # Llamado de filas
f0 = A[0] # Se llama la primera fila. En Matlab es con 1, acá es con 0.
f2 = A[2] # Se llama la tercera fila.
```

In [43]: f2

Out[43]: [4, 7, 8]

```
In [44]: # Impresión de resultados.
print('\nLa primera fila es:') # El comando \n es usado dejar un espacio
print(f0)
print('\nLa tercera fila es:')
print(f2)
```

```
La primera fila es:  
[5, 1, 2]
```

```
La primera columna es:
```

```
In [45]: # Llamado de columnas  
c0 = []  
for i in range(len(A)):  
    c0 += [A[i][0]]
```

```
In [46]: print('\nLa primera columna es:')  
print(c0)
```

```
La primera columna es:  
[5, 7, 4]
```

2. Arreglo en numpy

```
In [47]: # Mediante el módulo numpy con arrays (arreglos).  
# Importar la librería  
import numpy as np
```

```
In [48]: # Definición explícita.  
B = np.array([  
    [3, 1, 9],  
    [2, 4, 1],  
    ])
```

```
In [49]: # Impresión de resultados  
print('\n La matriz B es:')  
print(B)
```

```
La matriz B es:  
[[3 1 9]  
 [2 4 1]]
```

```
In [50]: # Definición directa con una variable previamente guardada.  
C = [  
    [3, 1, 9],  
    [2, 4, 1],  
    ]  
  
D = np.array(C)
```

```
In [51]: D
```

```
Out[51]: array([[3, 1, 9],  
               [2, 4, 1]])
```

```
In [52]: # Tipo de dato
         type(B)
```

```
Out[52]: numpy.ndarray
```

```
In [53]: # Tamaño de la matriz
         dim_B = B.shape
         print(dim_B) # Imprime el tamaño del arreglo B.

         (2, 3)
```

```
In [54]: # Tipo de dato
         type(dim_B)
```

```
Out[54]: tuple
```

```
In [55]: # Definiendo número de filas y columnas.
         m = dim_B[0] # Filas
         n = dim_B[1] # Columnas
```

```
In [56]: # Impresión de resultados
         print('El número de filas es:')
         print(m)
         print('El número de columnas es:')
         print(n)
```

```
El número de filas es:
2
El número de columnas es:
3
```

```
In [57]: A
```

```
Out[57]: [[5, 1, 2], [7, 3, 7], [4, 7, 8]]
```

```
In [58]: type(A)
```

```
Out[58]: list
```

```
In [59]: A.shape # Genera error.
```

```
-----
-----
AttributeError                                Traceback (most recent call
last)
<ipython-input-59-a9c3e3880230> in <module>
----> 1 A.shape # Genera error.

AttributeError: 'list' object has no attribute 'shape'
```


Otras formas de definir arreglos:

```
In [60]: # Arreglo de unos  
E = np.ones((4, 5))
```

```
In [61]: print(E)
```

```
[[1. 1. 1. 1. 1.]  
 [1. 1. 1. 1. 1.]  
 [1. 1. 1. 1. 1.]  
 [1. 1. 1. 1. 1.]]
```

```
In [62]: type(E)
```

```
Out[62]: numpy.ndarray
```

```
In [63]: E[0, 0]
```

```
Out[63]: 1.0
```

```
In [64]: print("\nCon elementos de tipo:")  
print(type(E[0][0]))
```

```
Con elementos de tipo:  
<class 'numpy.float64'>
```

```
In [65]: # Arreglo de ceros  
F = np.zeros((3, 2))
```

```
In [66]: F
```

```
Out[66]: array([[0., 0.],  
               [0., 0.],  
               [0., 0.]])
```

```
In [67]: print("\nCon elementos de tipo:")  
print(type(F[0][0]))
```

```
Con elementos de tipo:  
<class 'numpy.float64'>
```

```
In [68]: # Matriz vacía rellena con los valores residuales de la memoria.  
G = np.empty((2, 3))
```

```
In [69]: print(G)  
print("\nCon elementos de tipo:")  
print(type(G[0][0]))
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

Con elementos de tipo:

```
In [70]: # De forma opcional se puede definir el tipo de dato que tendrá el arreglo
H = np.zeros((3, 3), dtype=int) # dtype tiene muchísimas posibilidades.
```

```
In [71]: H
```

```
Out[71]: array([[0, 0, 0],
                [0, 0, 0],
                [0, 0, 0]])
```

```
In [72]: print("\nCon elementos de tipo:")
print(type(H[0][0]))
```

Con elementos de tipo:
<class 'numpy.int32'>

```
In [73]: # Matriz identidad para matrices cuadradas
np.identity(5)
```

```
Out[73]: array([[1., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0.],
                [0., 0., 1., 0., 0.],
                [0., 0., 0., 1., 0.],
                [0., 0., 0., 0., 1.]])
```

```
In [74]: # Matriz identidad en un caso más general mxn
np.eye(3, 4)
```

```
Out[74]: array([[1., 0., 0., 0.],
                [0., 1., 0., 0.],
                [0., 0., 1., 0.]])
```

```
In [75]: sp.zeros(3,2)
```

```
-----
NameError                                Traceback (most recent call
last)
<ipython-input-75-d896e439781d> in <module>
----> 1 sp.zeros(3,2)

NameError: name 'sp' is not defined
```

Llamado de elementos, filas y columnas.

```
In [76]: B # Previamente fue definida la matriz B.
```

```
Out[76]: array([[3, 1, 9],
               [2, 4, 1]])
```

```
In [77]: # Llamado de un elemento
b12 = B[1][2] # En la numeración natural sería el elemento de la fila 2 y
print(f'El elemento b12 es: {b12}') # Nótese el uso de la f y de {}.
```

El elemento b12 es: 1

```
In [78]: B[0][0]
```

```
Out[78]: 3
```

```
In [79]: B[0, 0]
```

```
Out[79]: 3
```

```
In [80]: # Alternativamente, se usa así:
b12 = B[1, 2]

print(f'El elemento b12 es: {b12}')
```

El elemento b12 es: 1

```
In [81]: B
```

```
Out[81]: array([[3, 1, 9],
               [2, 4, 1]])
```

```
In [82]: B[0]
```

```
Out[82]: array([3, 1, 9])
```

```
In [83]: # Llamado de una fila
f1 = B[0] # Se llama la PRIMERA fila.

print(f'La segunda fila es: {f1}')
```

La segunda fila es: [3 1 9]

```
In [84]: f2 = B[2] # Se llama la tercera fila. Como no existe: genera error.
```


IndexError

Traceback (most recent call

last)

In [85]: B[0]

Out[85]: array([3, 1, 9])

In [86]: B[:, 1]

Out[86]: array([1, 4])

In [87]: B

Out[87]: array([[3, 1, 9],
[2, 4, 1]])

In [88]: *# Llamado de columnas*
c1 = B[:, 1]

print(f'La segunda columna es: {c1}')

La segunda columna es: [1 4]

In [89]: *# Se define una matriz C, nuevamente con*
C = np.array([
[0, 1, 2, 3],
[4, 5, 6, 7],
[8, 9, 10, 11],
[12, 13, 14, 15], *# Nótese que esta ',' del final no genera*
])

print(f'La matriz C es: \n{C}')

La matriz C es:
[[0 1 2 3]
[4 5 6 7]
[8 9 10 11]
[12 13 14 15]]

Llamado de más de una fila o de una columna:

In [90]: C[1:]

Out[90]: array([[4, 5, 6, 7],
[8, 9, 10, 11],
[12, 13, 14, 15]])

In [91]: C[:2]

Out[91]:

```
array([[0, 1, 2, 3],
```

```
In [92]: C
```

```
Out[92]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])
```

```
In [93]: C[:, :2]
```

```
Out[93]: array([[ 0,  1],
                [ 4,  5],
                [ 8,  9],
                [12, 13]])
```

```
In [94]: C[1:, :2]
```

```
Out[94]: array([[ 4,  5],
                [ 8,  9],
                [12, 13]])
```

```
In [95]: C
```

```
Out[95]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])
```

```
In [96]: C[1:3, 1:3]
```

```
Out[96]: array([[ 5,  6],
                [ 9, 10]])
```

3. Matrices en sympy

```
In [97]: # Mediante el módulo sympy con "Array" y con "Matrix".
# Importar la librería
import sympy as sp
```

```
In [98]: J = sp.Array([[ 8,  1,  9], [-1,  4, -2], [ 2, -4,  1],])
```

```
In [99]: J
```

```
Out[99]: 
$$\begin{bmatrix} 8 & 1 & 9 \\ -1 & 4 & -2 \\ 2 & -4 & 1 \end{bmatrix}$$

```

```
In [100]: # Definición con 'Array', no se recomienda usar pues es inmutable.

J = sp.Array([
```

```

        [ 8,  1,  9],
        [-1,  4, -2],
        [ 2, -4,  1],
        ])
```

```

In [101]: # Impresión de resultados
print('La matriz J es:')
J # Acá no se pone el comando 'print' para que genere un 'Out', y en sym
```

La matriz J es:

```

Out[101]: 
$$\begin{bmatrix} 8 & 1 & 9 \\ -1 & 4 & -2 \\ 2 & -4 & 1 \end{bmatrix}$$

```

```

In [102]: print(J)

[[8, 1, 9], [-1, 4, -2], [2, -4, 1]]
```

```

In [103]: print(sp.prettv(J))
```

```


$$\begin{bmatrix} 8 & 1 & 9 \\ -1 & 4 & -2 \\ 2 & -4 & 1 \end{bmatrix}$$

```

```

In [104]: # Tipo de dato
type(J)
```

```

Out[104]: sympy.tensor.array.dense_ndim_array.ImmutableDenseNDimArray
```

```

In [105]: # Definición con "Matrix"
K = sp.Matrix([
        [-1,  1,  9],
        [ 4,  4, -2],
        [ 2,  0,  8],
        ])
```

```

In [106]: # Impresión de resultados
print('La matriz K es:')
K
```

La matriz K es:

```

Out[106]: 
$$\begin{bmatrix} -1 & 1 & 9 \\ 4 & 4 & -2 \\ 2 & 0 & 8 \end{bmatrix}$$

```

```

In [107]: # Tipo de dato
```

```
type(K)
```

```
Out[107]: sympy.matrices.dense.MutableDenseMatrix
```

```
In [108]: K.shape
```

```
Out[108]: (3, 3)
```

```
In [109]: # Tamaño de la matriz
dim_K = K.shape # Nótese que es el mismo comando en numpy y en sympy
print(dim_K)    # Imprime el tamaño del arreglo K

(3, 3)
```

```
In [110]: # Tipo de dato
type(dim_K)
```

```
Out[110]: tuple
```

Formas específicas de arreglos:

Se recomienda revisar en detalle la documentación de sympy en: <https://docs.sympy.org/latest/tutorial/matrices.html> (<https://docs.sympy.org/latest/tutorial/matrices.html>)

Las siguientes matrices tienen la forma de 'Matrix', es decir, son mutables:

```
In [111]: # Matriz de ceros.
sp.zeros(2)
```

```
Out[111]: 
$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

```

```
In [112]: sp.zeros(3, 2)
```

```
Out[112]: 
$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

```

```
In [113]: # Matriz de unos.
sp.ones(3)
```

```
Out[113]: 
$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

```

```
In [114]: # Matriz identidad
sp.eye(3)
```

```
Out[114]: 
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```

```
In [115]: # Matriz diagonal
sp.diag(1, 2, 3, 4)
```

```
Out[115]: 
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

```

Llamado de elementos, filas y columnas.

```
In [116]: K # Previamente fue definida la matriz K.
```

```
Out[116]: 
$$\begin{bmatrix} -1 & 1 & 9 \\ 4 & 4 & -2 \\ 2 & 0 & 8 \end{bmatrix}$$

```

```
In [117]: # Teniendo en cuenta la longitud de matriz, es decir, la cantidad de elementos
len(K) # Nótese que aquí sí se puede usar el comando "len()" a diferencia de numpy
```

```
Out[117]: 9
```

```
In [118]: # Se puede llamar a cualquiera de los elementos de la matriz en un ordenado
K[5]
```

```
Out[118]: -2
```

```
In [119]: K[1, 1] # Llamado del último elemento, que es lo mismo que llamar K[8]
```

```
Out[119]: 4
```

```
In [120]: # Pero el llamado básico se hace de la misma manera que numpy
K[1, 1] # No funciona poner K[1][1], genera error.
```

```
Out[120]: 4
```

```
In [121]: K[0, 0]
```

```
Out[121]: -1
```



```
In [122]: K
```

```
Out[122]: 
$$\begin{bmatrix} -1 & 1 & 9 \\ 4 & 4 & -2 \\ 2 & 0 & 8 \end{bmatrix}$$

```

```
In [123]: # Llamado de filas  
K.row(1) # Llamado de la segunda fila
```

```
Out[123]: 
$$\begin{bmatrix} 4 & 4 & -2 \end{bmatrix}$$

```

```
In [124]: # Alternativamente  
K[1, :] # También se puede llamar así, siempre y cuando se use coma intermedia
```

```
Out[124]: 
$$\begin{bmatrix} 4 & 4 & -2 \end{bmatrix}$$

```

```
In [125]: # Llamado de columnas  
K.col(2)
```

```
Out[125]: 
$$\begin{bmatrix} 9 \\ -2 \\ 8 \end{bmatrix}$$

```

```
In [126]: # Alternativamente  
K[:, 2]
```

```
Out[126]: 
$$\begin{bmatrix} 9 \\ -2 \\ 8 \end{bmatrix}$$

```

```
In [127]: # Se pueden hacer estos llamados de manera similar a numpy  
K[1:, 1:]
```

```
Out[127]: 
$$\begin{bmatrix} 4 & -2 \\ 0 & 8 \end{bmatrix}$$

```

Modificación de elementos

Cambiar algún elemento de la matriz

1. En una lista de listas

```
In [128]: A # Previamente fue definida la matriz A.
```

```
Out[128]: 
$$\begin{bmatrix} [5, 1, 2], [7, 3, 7], [4, 7, 8] \end{bmatrix}$$

```

```
In [129]: # Por ejemplo, el elemento de la fila 3 y la columna 1
A[2][0]
```

```
Out[129]: 4
```

```
In [130]: # Se puede modificar directamente
A[2][0] = 0
```

```
In [131]: A # Matriz A modificada
```

```
Out[131]: [[5, 1, 2], [7, 3, 7], [0, 7, 8]]
```

2. En un array de numpy

```
In [132]: C # Previamente fue definida la matriz C.
```

```
Out[132]: array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11],
                 [12, 13, 14, 15]])
```

```
In [133]: C[2, 1]
```

```
Out[133]: 9
```

```
In [134]: # Modificando el elemento C[2, 1]
C[2, 1] = -15
```

```
In [135]: C # Matriz C modificada
```

```
Out[135]: array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8, -15, 10, 11],
                 [12, 13, 14, 15]])
```

```
In [136]: # Así modifico toda la fila 2 simultáneamente
C[2]
```

```
Out[136]: array([ 8, -15, 10, 11])
```

```
In [137]: C[2] = 0
```

```
In [138]: C
```

```
Out[138]: array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 0,  0,  0,  0],
                 [12, 13, 14, 15]])
```

```
In [139]: # Por ejemplo, así se modifican todos los elementos.  
C[:, :] = 0
```

```
In [140]: C
```

```
Out[140]: array([[0, 0, 0, 0],  
                [0, 0, 0, 0],  
                [0, 0, 0, 0],  
                [0, 0, 0, 0]])
```

3. En una matriz de sympy

```
In [141]: K # Previamente fue definida la matriz K.
```

```
Out[141]: 
$$\begin{bmatrix} -1 & 1 & 9 \\ 4 & 4 & -2 \\ 2 & 0 & 8 \end{bmatrix}$$

```

```
In [142]: type(K)
```

```
Out[142]: sympy.matrices.dense.MutableDenseMatrix
```

```
In [143]: K[2,2]
```

```
Out[143]: 8
```

```
In [144]: K[2, 2] = -10
```

```
In [145]: K
```

```
Out[145]: 
$$\begin{bmatrix} -1 & 1 & 9 \\ 4 & 4 & -2 \\ 2 & 0 & -10 \end{bmatrix}$$

```

```
In [146]: # Modificando el elemento K[1, 1]  
K[1, 1] = 0
```

```
In [147]: K # Matriz K modificada
```

```
Out[147]: 
$$\begin{bmatrix} -1 & 1 & 9 \\ 4 & 0 & -2 \\ 2 & 0 & -10 \end{bmatrix}$$

```

```
In [148]: K[8]
```

```
Out[148]: -10
```

```
In [149]: # Pilas que así se modifica el elemento 2 del recorrido fila a fila.  
K[2] = 1
```

```
In [150]: K
```

```
Out[150]: 
$$\begin{bmatrix} -1 & 1 & 1 \\ 4 & 0 & -2 \\ 2 & 0 & -10 \end{bmatrix}$$

```

```
In [151]: # Si se quiere modificar la columna 1.  
K.col(1)
```

```
Out[151]: 
$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

```

```
In [152]: # Se genera error.  
K.col(1) = -100
```

```
File "<ipython-input-152-56f824f47dfd>", line 2  
    K.col(1) = -100  
      ^  
SyntaxError: cannot assign to function call
```

```
In [153]: # La forma alternativa de llamar la columna 1.  
K[:, 1]
```

```
Out[153]: 
$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

```

```
In [154]: # Si se trata de modificar  
K[:,1] = -100
```

```

-----
ValueError                                Traceback (most recent call
last)
<ipython-input-154-deddd104c22b5> in <module>
      1 # Si se trata de modificar
----> 2 K[:,1] = -100

~\anaconda3\lib\site-packages\sympy\matrices\dense.py in __setitem__(self, key, value)
      356         [2, 2, 4, 2]])
      357         """
--> 358         nx = self._setitem(key, value)

```

```

In [155]: # Una forma correcta es:
K[:,1] = (-100, -100, -100)

```

```

In [156]: K

```

```

Out[156]: 
$$\begin{bmatrix} -1 & -100 & 1 \\ 4 & -100 & -2 \\ 2 & -100 & -10 \end{bmatrix}$$


```

Operaciones con matrices y vectores

1. Suma
2. Multiplicación
3. Inversión
4. Matriz transpuesta
5. Producto punto y producto cruz

Las operaciones básicas presentan algunas dificultades en una lista de lista, por lo que, claramente es mejor realizar las operaciones con los módulos numpy y sympy.

1. Suma

La suma de matrices solo es posible en matrices del mismo tamaño $m \times n$.

La suma de las matrices $\underline{\underline{A}}$ y $\underline{\underline{B}}$ es denotada por: $\underline{\underline{A}} + \underline{\underline{B}}$

Usando numpy

```

In [157]: # Usando el módulo numpy se crean las matrices M y N.
M = np.array([
            [ 2, 7],
            [-1, 3],
            ])

N = np.array([

```

```
[-3, 4],
[ 9, -2],
])
```

In [158]: M

```
Out[158]: array([[ 2,  7],
                [-1,  3]])
```

In [159]: N

```
Out[159]: array([[ -3,  4],
                [ 9, -2]])
```

In [160]: M+N *# La suma de las matrices M y N.*

```
Out[160]: array([[ -1, 11],
                [ 8,  1]])
```

¿Qué pasa si las matrices a sumar son diferente tamaño?

In [161]: B *# Llamando la matriz B, previmante definida*

```
Out[161]: array([[3, 1, 9],
                [2, 4, 1]])
```

In [162]: B+N *# Lo que ocurre si se suman matrices de diferente tamaño.*

```
-----
-----
ValueError                                Traceback (most recent call
last)
<ipython-input-162-fd3fd26b354f> in <module>
----> 1 B+N # Lo que ocurre si se suman matrices de diferente tamaño.

ValueError: operands could not be broadcast together with shapes (2,3)
(2,2)
```

Usando sympy

In [163]: *# Se crean algunas variables simbólicas con sympy.*
a, b, c, d = sp.symbols("a b c d")

In [164]: *# Usando el módulo sympy se crean las matrices O y P.*
O = sp.Matrix([
 [a, c],
 [b, d],
])

P = sp.Matrix([
 [-a, b],

```
[ c, d],
    ])
```

In [165]: O

Out[165]: $\begin{bmatrix} a & c \\ b & d \end{bmatrix}$

In [166]: P

Out[166]: $\begin{bmatrix} -a & b \\ c & d \end{bmatrix}$

In [167]: O-P # La resta de las matrices O y P.

Out[167]: $\begin{bmatrix} 2a & -b+c \\ b-c & 0 \end{bmatrix}$

2. Multiplicación

El producto de una matriz A de tamaño $a \times b$ con una matriz B de tamaño $c \times d$ solo es posible si $b = c$ y, en donde la matriz resultante tendrá un tamaño $a \times d$.

La multiplicación de las matrices $\underline{\underline{A}}$ y $\underline{\underline{B}}$ es denotada por: $\underline{\underline{A}} * \underline{\underline{B}}$

Usando numpy

```
In [168]: # Usando el módulo numpy se crean las matrices Q y R.
Q = np.array([
            [3, -2],
            ])

R = np.array([
            [ 3],
            [-1],
            ])
```

In [169]: O

Out[169]: array([[3, -2]])

In [170]: R

Out[170]: array([[3],
 [-1]])

In [171]: np.matmul(Q, R) # La operación Q*R hace otra cosa completamente diferente

Out[171]: array([[11]])

```
In [172]: Q@R # Otra forma más sencilla de hacer la misma operación.
```

```
Out[172]: array([[11]])
```

```
In [173]: Q*R # No sigue las reglas matriciales, pues debería dar de 1x1
```

```
Out[173]: array([[ 9, -6],  
                [-3,  2]])
```

Los arreglos en numpy permiten realizar sobre ellos las operaciones básicas **elemento a elemento**, lo cual no responde a las reglas de las operaciones con matrices.

```
In [174]: N
```

```
Out[174]: array([[ -3,  4],  
                [ 9, -2]])
```

```
In [175]: # No sigue reglas matriciales, pero funciona.  
N + 100 # Es equivalente a la suma de una matriz de unos multiplicada po
```

```
Out[175]: array([[ 97, 104],  
                [109,  98]])
```

```
In [176]: N * 3 # Es equivalente a N+N+N, es la multiplicación de una matriz por u
```

```
Out[176]: array([[ -9, 12],  
                [27, -6]])
```

```
In [177]: N@N # Para multiplicar una matriz por sí misma.
```

```
Out[177]: array([[ 45, -20],  
                [-45,  40]])
```

```
In [178]: # No sigue las reglas de las operaciones con matrices.  
N**2 # Matricialmente no es lo mismo que NxN. Porque lo que hace es elev
```

```
Out[178]: array([[ 9, 16],  
                [81,  4]], dtype=int32)
```

```
In [179]: (N+2)**2/8 # No sigue las reglas de matrices.
```

```
Out[179]: array([[ 0.125,  4.5  ],  
                [15.125,  0.   ]])
```

```
In [180]: N
```

```
Out[180]: array([[ -3,  4],  
                [ 9, -2]])
```



```
In [181]: 1/N # Cuidado, que esta operación no es la matriz inversa.
```

```
Out[181]: array([[ -0.33333333,  0.25      ],
                 [ 0.11111111, -0.5      ]])
```

```
In [182]: # ¿Qué ocurre si se eleva a potencias negativas.
N**-1 # Se parece a la inversa, pero NO ES LA INVERSA
```

```
-----
-----
ValueError                                Traceback (most recent call
last)
<ipython-input-182-6917764aacf7> in <module>
      1 # ¿Qué ocurre si se eleva a potencias negativas.
----> 2 N**-1 # Se parece a la inversa, pero NO ES LA INVERSA

ValueError: Integers to negative integer powers are not allowed.
```

Usando sympy

```
In [183]: # Usando el módulo sympy se crea la matriz A y el vector X.

# Se crean algunas variables simbólicas adicionales.
x, y = sp.symbols("x y")

A = sp.Matrix([
                [ a, b],
                [ c, d],
                ])

X = sp.Matrix([
                [x],
                [y],
                ])
```

```
In [184]: A
```

```
Out[184]: 
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

```

```
In [185]: X
```

```
Out[185]: 
$$\begin{bmatrix} x \\ y \end{bmatrix}$$

```

```
In [186]: A*X # En sympy sí se puede hacer esta operación así. operacion matricial
```

```
Out[186]: 
$$\begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

```

Algunas consideraciones sobre las operaciones:

```
In [187]: A # Llamando la matriz previamente definida.
```

```
Out[187]: 
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

```

```
In [188]: # Nótese que aquí sí se respetan las reglas de las operaciones entre matr.  
A+10
```

```
-----  
-----  
TypeError                                Traceback (most recent call  
last)  
<ipython-input-188-5324d02d0726> in <module>  
      1 # Nótese que aquí sí se respetan las reglas de las operaciones  
entre matrices.  
----> 2 A+10  
  
~\anaconda3\lib\site-packages\sympy\core\decorators.py in binary_op_wr  
apper(self, other)  
    134             if f is not None:  
    135                 return f(self)  
--> 136             return func(self, other)  
    137             return binary_op_wrapper  
    138             return priority_decorator  
  
~\anaconda3\lib\site-packages\sympy\matrices\common.py in __add__(sel  
f, other)  
    2694             return MatrixArithmetic._eval_add(self, other)  
    2695  
-> 2696             raise TypeError('cannot add %s and %s' % (type(self),  
type(other)))  
    2697  
    2698     @call_highest_priority('__rtruediv__')  
  
TypeError: cannot add <class 'sympy.matrices.dense.MutableDenseMatrix'  
> and <class 'int'>
```

```
In [189]: # Las matrices se pueden multiplicar por escalares.  
O*15
```

```
Out[189]: 
$$\begin{bmatrix} 15a & 15c \\ 15b & 15d \end{bmatrix}$$

```

```
In [190]: # Esta operación sí representa la multiplicación de 0 con 0.
0**2
```

```
Out[190]: 
$$\begin{bmatrix} a^2 + bc & ac + cd \\ ab + bd & bc + d^2 \end{bmatrix}$$

```

3. Inversión de matrices

Solo es posible invertir matrices cuadradas no singulares, es decir, que su determinante sea diferente de 0.

La inversa de la matriz $\underline{\underline{A}}$ es denotada por $\underline{\underline{A}}^{-1}$.

El determinante de la matriz $\underline{\underline{A}}$ se puede denotar como $\det(\underline{\underline{A}})$ o como $|\underline{\underline{A}}|$.

Usando numpy

```
In [191]: # Usando el módulo numpy se crea la matriz S.
S = np.array([[ 1, -2],
               [-3, 5]])
```

```
In [192]: S # Se imprime S
```

```
Out[192]: array([[ 1, -2],
                 [-3,  5]])
```

```
In [193]: # Se usa el submódulo np.linalg
np.linalg.det(S) # Se determina el determinante de la matriz S.
```

```
Out[193]: -1.0000000000000004
```

```
In [194]: # Como el determinante es diferente de 0, la matriz es inversible.
np.linalg.inv(S) # Inversa de la matriz S.
```

```
Out[194]: array([[ -5., -2.],
                 [-3., -1.]])
```

¿Qué ocurre si se trata de sacar la inversa de una matriz singular?

```
In [195]: W = np.array([[ 1, -2],
                         [-2,  4]])
```

```
In [196]: np.linalg.det(W)
```

```
Out[196]: 0.0
```

```
In [197]: np.linalg.inv(W)
```

```
-----  
-----  
LinAlgError                                Traceback (most recent call  
last)  
<ipython-input-197-003d22724806> in <module>  
----> 1 np.linalg.inv(W)  
  
<__array_function__ internals> in inv(*args, **kwargs)  
  
~\anaconda3\lib\site-packages\numpy\linalg\linalg.py in inv(a)  
    543     signature = 'D->D' if isComplexType(t) else 'd->d'  
    544     extobj = get_linalg_error_extobj(_raise_linalgerror_singul  
ar)  
--> 545     ainv = _umath_linalg.inv(a, signature=signature, extobj=ex  
tobj)  
    546     return wrap(ainv.astype(result_t, copy=False))  
    547  
  
~\anaconda3\lib\site-packages\numpy\linalg\linalg.py in _raise_linalg  
error_singular(err, flag)  
    86  
    87 def _raise_linalgerror_singular(err, flag):  
--> 88     raise LinAlgError("Singular matrix")  
    89  
    90 def _raise_linalgerror_nonposdef(err, flag):  
  
LinAlgError: Singular matrix
```

Usando sympy

```
In [198]: # Usando la matriz A previamente creada con variables simbólicas.  
A
```

```
Out[198]: 
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

```

```
In [199]: # Cálculo del determinante.  
A.det()
```

```
Out[199]:  $ad - bc$ 
```

```
In [200]: A.inv() # Inversa de la matriz A. Aquí la utilidad del módulo sympy para
```

```
Out[200]: 
$$\begin{bmatrix} \frac{d}{ad-bc} & -\frac{b}{ad-bc} \\ -\frac{c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}$$

```

```
In [201]: A**-1 # Esta operación sí es la inversa de la matriz, al contrario de lo
```

```
Out[201]: 
$$\begin{bmatrix} \frac{d}{ad-bc} & -\frac{b}{ad-bc} \\ -\frac{c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}$$

```

4. Matriz transpuesta

Se denota a la transpuesta de la matriz $\underline{\underline{A}}$ como $\underline{\underline{A}}^T$.

¿Cómo se transpone la matriz $\underline{\underline{A}}$? (ejercicio de clase con lista de listas)

```
In [202]: A = [[ 0, 1, 2, 3, 4],  
               [ 5, 6, 7, 8, 9],  
               [10, 11, 12, 13, 14]]
```

```
In [203]: # Solución
```

Usando numpy

```
In [204]: B # Previamente fue definida la matriz B.
```

```
Out[204]: array([[3, 1, 9],  
                [2, 4, 1]])
```

```
In [205]: B.T # Matriz transpuesta.
```

```
Out[205]: array([[3, 2],  
                [1, 4],  
                [9, 1]])
```

Usando sympy

```
In [206]: K # Previamente fue definida la matriz K.
```

```
Out[206]: 
$$\begin{bmatrix} -1 & -100 & 1 \\ 4 & -100 & -2 \\ 2 & -100 & -10 \end{bmatrix}$$

```

```
In [207]: K.T # Matriz transpuesta.
```

```
Out[207]: 
$$\begin{bmatrix} -1 & 4 & 2 \\ -100 & -100 & -100 \\ 1 & -2 & -10 \end{bmatrix}$$

```

5. Producto punto y producto cruz

El producto punto o producto escalar de dos vectores está definido como:

$$\underline{A} \cdot \underline{B} = (a_1, a_2, \dots, a_n) \cdot (b_1, b_2, \dots, b_n) = a_1 b_1 + a_2 b_2 + \dots + a_n b_n = \sum a_i b_i$$

Alternativamente, se puede definir como:

$$\underline{A} \cdot \underline{B} = ||\underline{A}|| \cdot ||\underline{B}|| \cos(\theta), \text{ donde } \theta \text{ es el ángulo formado entre los vectores.}$$

Nota: De aquí en adelante un vector entre dos líneas a lado y lado representará la magnitud o norma: $||\underline{A}||$.

Una matriz entre dos líneas representa un determinante: $|\underline{A}|$.

Pregunta de clase

¿Qué significa que el $\underline{A} \cdot \underline{B} = 0$?

Pregunta de clase

¿Qué significa que el $\underline{A} \cdot \underline{B} = ||\underline{A}|| \cdot ||\underline{B}||$?

El producto cruz o producto vectorial de dos vectores está definido en \mathbb{R}^3 como $\underline{A} \times \underline{B}$, en donde:

$$\underline{A} = (a_x, a_y, a_z)$$

$$\underline{B} = (b_x, b_y, b_z)$$

es decir,

$$\underline{A} \times \underline{B} = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

```
In [208]: # Usando sympy se obtiene

ax, ay, az, bx, by, bz = sp.symbols("a_x a_y a_z b_x b_y b_z")
ig, jg, kg = sp.symbols("\hat{i} \hat{j} \hat{k}")

AxB = sp.Matrix([
                    [ig, jg, kg],
                    [ax, ay, az],
                    [bx, by, bz]
                ]).det()

sp.collect(AxB, [ig, jg, kg])
```

```
Out[208]: \hat{i} (a_y b_z - a_z b_y) + \hat{j} (-a_x b_z + a_z b_x) + \hat{k} (a_x b_y - a_y b_x)
```

De esta manera, $\underline{A} \times \underline{B} = \underline{C}$, donde:

$$\underline{C} = (c_x, c_y, c_z),$$

$$c_x = a_y \cdot b_z - a_z \cdot b_y$$

$$c_y = -a_x \cdot b_z + a_z \cdot b_x$$

Además, es útil la definición:

$$||\underline{A} \times \underline{B}|| = ||\underline{A}|| \ ||\underline{B}|| \ |\sin(\theta)|$$

Pregunta de clase

¿Qué significa que $\underline{A} \times \underline{B} = \underline{0}$

Usando numpy

```
In [209]: # Se crean los vectores u1 y u2.
u1 = np.array([7, -4, -1])
u2 = np.array([3, -5, 2])
```

```
In [210]: u1 # Se visualiza el vector u1.
```

```
Out[210]: array([ 7, -4, -1])
```

```
In [211]: u2 # Se visualiza el vector u2.
```

```
Out[211]: array([ 3, -5, 2])
```

```
In [212]: # El producto punto de u1 y u2.
np.dot(u1, u2)
```

```
Out[212]: 39
```

Usando sympy

```
In [213]: # Se crean algunas variables simbólicas
a1, a2, a3, b1, b2, b3 = sp.symbols("a_1 a_2 a_3 b_1 b_2 b_3")

# Se crean los vectores v1 y v2 con las variables simbólicas disponibles.
v1 = sp.Matrix([a1, a2, a3])
v2 = sp.Matrix([b1, b2, b3])
```

```
In [214]: v1 # Se visualiza el vector v1.
```

```
Out[214]: 
$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

```

```
In [215]: v2 # Se visualiza el vector v2.
```

```
Out[215]: 
$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

```

```
In [216]: # El producto punto de v1 y v2.  
v1.dot(v2) # Es lo mismo que v2.dot(v1)
```

```
Out[216]:  $a_1 b_1 + a_2 b_2 + a_3 b_3$ 
```

```
In [217]: # El producto cruz de v1 y v2.  
v3 = v1.cross(v2)  
  
v3 # Se imprime v3.
```

```
Out[217]: 
$$\begin{bmatrix} a_2 b_3 - a_3 b_2 \\ -a_1 b_3 + a_3 b_1 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

```

```
In [218]: # El producto punto de v1 y v3.  
v1.dot(v3) # Debería dar 0.
```

```
Out[218]:  $a_1 (a_2 b_3 - a_3 b_2) + a_2 (-a_1 b_3 + a_3 b_1) + a_3 (a_1 b_2 - a_2 b_1)$ 
```

```
In [219]: sp.simplify(v1.dot(v3)) # Se usa el comando "simplify".
```

```
Out[219]: 0
```

```
In [220]: # El producto punto de v2 y v2.  
sp.simplify(v2.dot(v3))
```

```
Out[220]: 0
```

Resumen de conocimientos

- Lo más básico de Python
- Formas de definir matrices
- Modificación de elementos
- Operaciones con matrices y vectores

Resumen de comandos usados

- +
- -
- *
- /

- //
- %
- **
- +=
- ==
- !=
- print()
- type()
- len()
- for __ in __ :
- np.array()
- __.shape
- np.ones()
- np.zeros()
- np.empty()
- dtype
- np.identity()
- np.eye()
- sp.Array()
- sp.Matrix()
- sp.zeros()
- sp.ones()
- sp.eye()
- sp.diag()
- __, __ = sp.symbols('__ __')
- np.linalg.det()
- np.linalg.inv()
- __.det()
- __.inv()
- __.T
- sp.collect()
- np.dot(__, __)
- __.dot(__)
- __.cross(__)