



3 Independencia lineal, base ortonormal, combinación lineal.

Objetivo.

Revisar e ilustrar los conceptos de independencia lineal y base ortonormal para \mathbb{R}^2 usando la biblioteca `numpy`.

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#)

Trabajo realizado con el apoyo del Programa UNAM-DGAPA-PAPIME PE101922

```
# Importamos las bibliotecas requeridas
import numpy as np
import ipywidgets as widgets
import macti.visual as mvis
from macti.evaluation import *
```

```
quizz = Quizz('03', 'notebooks', 'local')
```

3.1 Independencia lineal. [🔗](#)

Los vectores $\vec{x}_1, \vec{x}_2, \dots, \vec{x}_n$ son **linealmente independientes** si de la ecuación:

$$\sum_{i=1}^n \alpha_i \vec{x}_i = 0 \quad (1)$$

se deduce que $\alpha_i = 0$, para toda i . Si por lo menos una de las α_i es distinta de cero, entonces los vectores son **linealmente dependientes**.

3.2 Ejemplo 1.

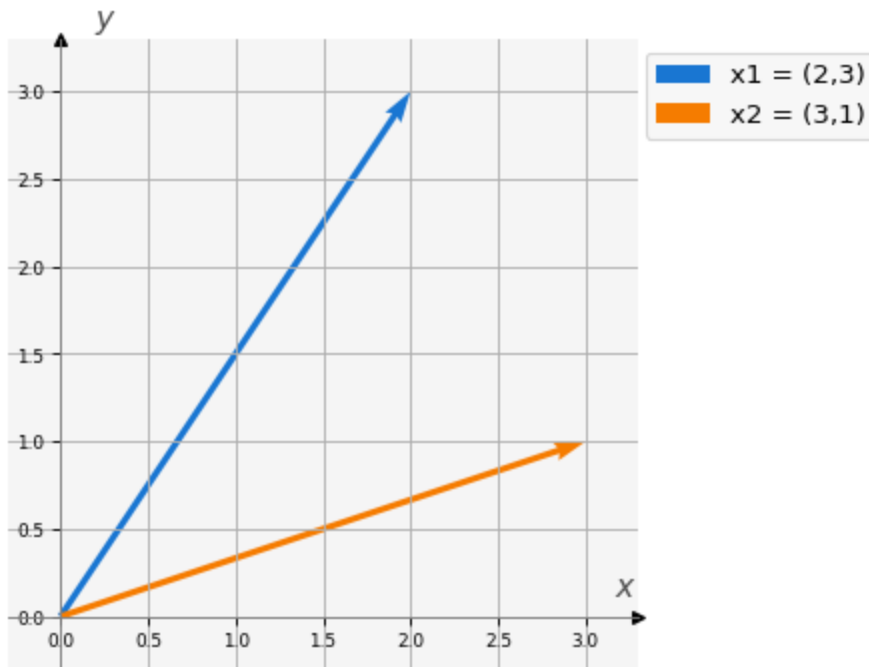
Definimos dos vectores, $\vec{x}_1 = (2, 3)$ y $\vec{x}_2 = (3, 1)$ en \mathbb{R}^2 usando `numpy` como sigue:

```
x1 = np.array([2, 3])
x2 = np.array([3, 1])

# Imprimimos los vectores
print('x1 = {}'.format(x1))
print('x2 = {}'.format(x2))
```

```
# Visualizamos los vectores.
v = mvis.Plotter() # Definición de un objeto para crear figuras.
v.set_coordsys(1) # Definición del sistema de coordenadas.
v.plot_vectors(1, [x1, x2], ['x1 = (2,3)', 'x2 = (3,1)'], ofx=-0.1) # Graficación
v.grid() # Muestra la rejilla del sistema de coordenadas.
```

```
x1 = [2 3]
x2 = [3 1]
```



Observa que los vectores **no son paralelos**, esto es equivalente a que los vectores sean **linealmente independientes**.

Ahora, de acuerdo con la definición (1) tenemos que $\alpha_1 \vec{x}_1 + \alpha_2 \vec{x}_2 = 0$ solo se cumple cuando $\alpha_1 = \alpha_2 = 0$. La siguiente celda de código, genera un interactivo en donde se muestra lo anterior de manera gráfica para $\alpha_1, \alpha_2 \in [-2, 2]$. Ejecuta la celda y posteriormente mueve el valor de las α 's.

```
def dependencial_lineal(x1, x2,  $\alpha$ 1,  $\alpha$ 2):
    print('x1 = {} \t x2 = {}'.format(x1, x2))
    print('alpha1 * x1 + alpha2 * x2 = {}'.format( $\alpha$ 1 * x1 +  $\alpha$ 2 * x2))
    # Visualizamos los vectores.
    v = mvis.Plotter() # Definición de un objeto para crear figuras.
    v.set_coordsys(1) # Definición del sistema de coordenadas.
    v.plot_vectors_sum(1, [ $\alpha$ 1 * x1,  $\alpha$ 2 * x2], ['alpha1 * x1', 'alpha2 * x2'], ofx=-0.1) #
    v.grid() # Muestra la rejilla del sistema de coordenadas.
    return

widgets.interactive(dependencial_lineal,
                    x1 = widgets.fixed(x1),
                    x2 = widgets.fixed(x2),
                     $\alpha$ 1 = widgets.FloatSlider(min=-2.0, max=2.0, step=0.5, value=1
                     $\alpha$ 2 = widgets.FloatSlider(min=-2.0, max=2.0, step=0.5, value=1
```

3.3 Base ortonormal

En el espacio euclidiano \mathbb{R}^n , los vectores $\vec{e}_1 = (1, 0, \dots, 0)$, $\vec{e}_2 = (0, 1, \dots, 0)$, \dots , $\vec{e}_n = (0, 0, \dots, n)$, son linealmente independientes y representan una **base ortonormal**. Además, cualquier vector $\vec{z} = (z_1, z_2, \dots, z_n) \in \mathbb{R}^n$ se puede representar como

$$\vec{z} = \sum_{i=1}^n z_i \vec{e}_i$$

3.4 Ejemplo 2.

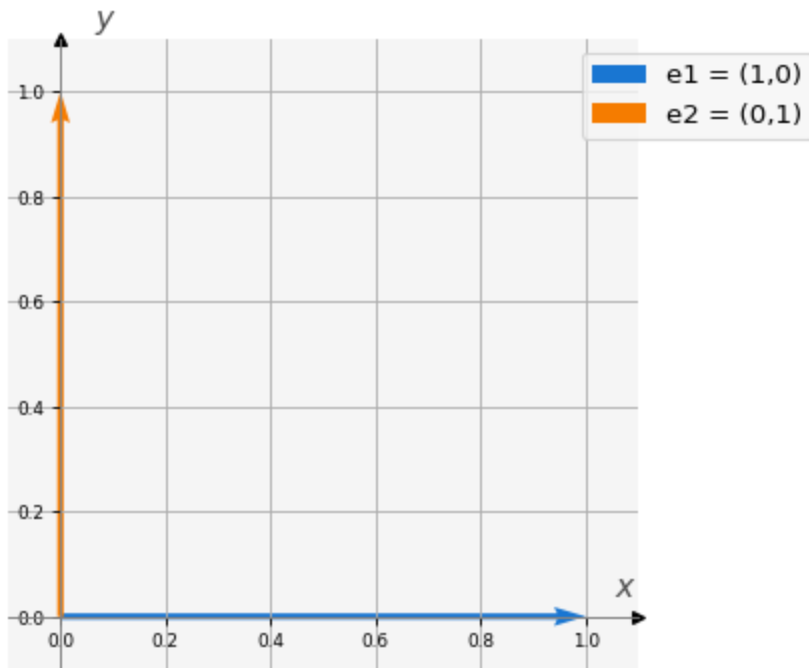
Definimos los vectores: $\vec{e}_1 = (1, 0)$ y $\vec{e}_2 = (0, 1)$ como sigue:

```
e1 = np.array([1, 0])
e2 = np.array([0, 1])

# Imprimimos los vectores
print('e1 = {}'.format(e1))
print('e2 = {}'.format(e2))

# Visualizamos los vectores.
v = mvis.Plotter() # Definición de un objeto para crear figuras.
v.set_coordsys(1)  # Definición del sistema de coordenadas.
v.plot_vectors(1, [e1, e2], ['e1 = (1,0)', 'e2 = (0,1)'], ofx=-0.2) # Graficación
v.grid()           # Muestra la rejilla del sistema de coordenadas.
```

```
e1 = [1 0]
e2 = [0 1]
```



Observa que los vectores **son ortogonales** y de tamaño unitario, por lo que representan una base ortonormal de \mathbb{R}^2 .

Con esta base podemos representar cualquier vector de \mathbb{R}^2 , particularmente $\vec{x}_1 = (2, 3)$ y $\vec{x}_2 = (3, 1)$ del **Ejemplo 1**.

Construcción del vector \vec{x}_1 :

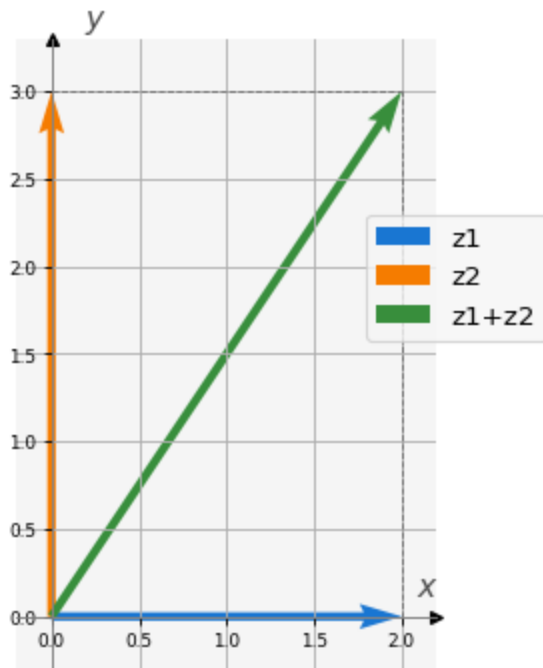
```
# Construcción de la combinación lineal
z1 = x1[0] * e1
z2 = x1[1] * e2

# Resultado final
z = z1 + z2

# Imprimimos el vector
print('x1 = {}'.format(z))

# Visualizamos los vectores.
v = mvis.Plotter() # Definición de un objeto para crear figuras.
v.set_coordsys(1)  # Definición del sistema de coordenadas.
v.plot_vectors_sum(1, [z1, z2], ['z1', 'z2'], ofx=-0.2, w=0.02) # Graficación de 1
v.grid() # Muestra la rejilla del sistema de coordenadas.
```

$x1 = [2 \ 3]$



3.5 Ejercicio 1.

Construye \vec{x}_2 usando la base ortonormal definida en el ejemplo 2 como se hizo para \vec{x}_1 .

```
# Construcción de la combinación lineal
# z1 = ...
# z2 = ...

# Resultado final
# z = z1 + z2

### BEGIN SOLUTION
# Construcción de la combinación lineal
z1 = x2[0] * e1
z2 = x2[1] * e2

# Resultado final
z = z1 + z2

file_answer = FileAnswer()
file_answer.write('1', z, 'z es incorrecta: revisa la construcción de la combinación lineal')
### END SOLUTION

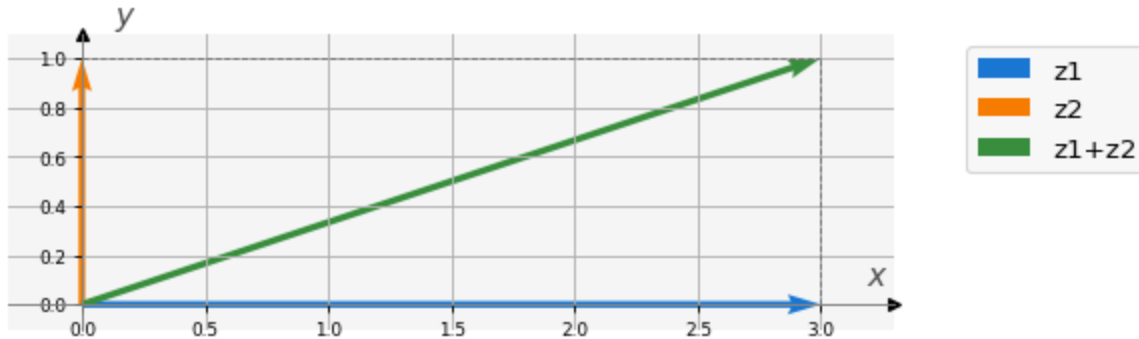
# Imprimimos el vector
print('x2 = {}'.format(z))

# Visualizamos los vectores.
```

```
v = mvis.Plotter() # Definición de un objeto para crear figuras.
v.set_coordsys(1) # Definición del sistema de coordenadas.
v.plot_vectors_sum(1, [z1, z2], ['z1', 'z2'], ofx=-0.2, w=0.0075) # Graficación de
v.grid() # Muestra la rejilla del sistema de coordenadas.
```

Creando el directorio `:/home/jovyan/macti/notebooks/.ans/Algebra_Lineal_01/`
 Respuestas y retroalimentación almacenadas.

`x2 = [3 1]`



```
quizz.eval_numeric('1', z)
```

 1 | Tu resultado es correcto.

3.6 Ejercicio 2.

Usando `numpy` define la base ortonormal $\{\vec{e}_1, \vec{e}_2, \vec{e}_3, \vec{e}_4\} \in \mathbb{R}^4$ y con ella construye el vector $\vec{y} = (1.5, 1.0, 2.3, -1.0)$. Imprime la base ortonormal y el resultado de construir el vector \vec{y} .

```
### BEGIN SOLUTION
e1 = np.array([1, 0, 0, 0])
e2 = np.array([0, 1, 0, 0])
e3 = np.array([0, 0, 1, 0])
e4 = np.array([0, 0, 0, 1])

# Imprimimos los vectores
print('e1 = {}'.format(e1))
print('e2 = {}'.format(e2))
print('e3 = {}'.format(e3))
print('e4 = {}'.format(e4))

y1 = 1.5 * e1
y2 = 1.0 * e2
```

```

y3 = 2.3 * e3
y4 = -1.0 * e4

print(' y = {}'.format(y1 + y2 + y3 + y4))

file_answer.write('2', e1, 'e1 es incorrecto: revisa la construcción del vector.'
file_answer.write('3', e2, 'e2 es incorrecto: revisa la construcción del vector.'
file_answer.write('4', e3, 'e3 es incorrecto: revisa la construcción del vector.'
file_answer.write('5', e4, 'e4 es incorrecto: revisa la construcción del vector.'
### END SOLUTION

```

```
e1 = [1 0 0 0]
```

```
e2 = [0 1 0 0]
```

```
e3 = [0 0 1 0]
```

```
e4 = [0 0 0 1]
```

```
y = [ 1.5  1.   2.3 -1. ]
```

El directorio `:/home/jovyan/macti/notebooks/.ans/Algebra_Lineal_01/` ya existe

Respuestas y retroalimentación almacenadas.

```

quizz.eval_numeric('2', e1)
quizz.eval_numeric('3', e2)
quizz.eval_numeric('4', e3)
quizz.eval_numeric('5', e4)

```

 2 | Tu resultado es correcto.

 3 | Tu resultado es correcto.


 4 | Tu resultado es correcto.

 5 | Tu resultado es correcto.

6 Sistemas de ecuaciones lineales: introducción

Objetivo general - Plantear y resolver un problema en términos de la solución de un sistema de ecuaciones lineales.

Objetivos particulares - Entender como plantear un problema en términos de un sistema de ecuaciones lineales. - Usar funciones de la biblioteca `numpy` para resolver el problema. - Comparar varios métodos para la solución de problemas más complejos.

[MACTI NOTES](#) by Luis Miguel de la Cruz Salas is licensed under [CC BY-NC-SA 4.0](#) 

Trabajo realizado con el apoyo del Programa UNAM-DGAPA-PAPIME PE101922

Planes de telefonía móvil.

Dos compañías de telefonía compiten por ganar clientes. En la tabla que sigue se muestra el costo de la renta y el costo por Megabyte (MB) de datos de cada compañía.

	Renta mensual	Costo por MB
Compañía A	200	0.10
Compañía B	20	0.30

¿Cómo podríamos decidir cuál de estas compañías conviene contratar?

Modelo matemático - Observamos en la tabla anterior que la compañía A tiene un precio fijo de 200 pesos mensuales que es 10 veces mayor al precio que cobra la compañía B (20 pesos). - Por otro lado, la compañía B cobra 0.30 pesos por cada MB, que es 3 veces mayor al precio por MB de la compañía A. - El precio final mensual de cada compañía depende básicamente de cuantos MB se usen.

Podemos escribir la forma en que cambia el precio de cada compañía en función de los MB usados:

\$

$$\begin{aligned} P_A &= 0.10x + 200 \\ P_B &= 0.30x + 20 \end{aligned} \quad (1)$$

\$

donde x representa el número de MB usados durante un mes.

6.0.1 Ejercicio 1. Gráfica de rectas.

En el código siguiente complete las fórmulas para cada compañía de acuerdo con las ecuaciones dadas en (1) y posteriormente ejecute el código para obtener una gráfica de cómo cambia el precio en función de los MB utilizados.


```
# Importación de las bibliotecas numpy y matplotlib
import numpy as np
import matplotlib.pyplot as plt
import sys, macti.visual

from macti.evaluation import *
```

```
quizz = Quizz('06', 'notebooks', 'local')
```

Fórmulas a implementar: \$

$$P_A = 0.10x + 200$$

$$P_B = 0.30x + 20$$

\$

```
# Megabytes desde 0 hasta 1500 (1.5 GB) en pasos de 10.
x = np.linspace(0,1500,10)

# Fórmulas de cada compañía
# PA = ...
# PB = ...
#
#### BEGIN SOLUTION
PA = 0.10 * x + 200
PB = 0.30 * x + 20

file_answer = FileAnswer()
file_answer.write('1', PA, 'Checa la fórmula para PA')
file_answer.write('2', PB, 'Checa la fórmula para PB')
#### END SOLUTION

print('PA = {}'.format(PA))
print('Pb = {}'.format(PB))
```

El directorio `:/home/jovyan/macti_notes/notebooks/.ans/SMM/` ya existe
Respuestas y retroalimentación almacenadas.

```
PA = [200.          216.66666667 233.33333333 250.          266.66666667
      283.33333333 300.          316.66666667 333.33333333 350.          ]
Pb = [ 20.   70.  120.  170.  220.  270.  320.  370.  420.  470.]
```

```
quizz.eval_numeric('1', PA)
```

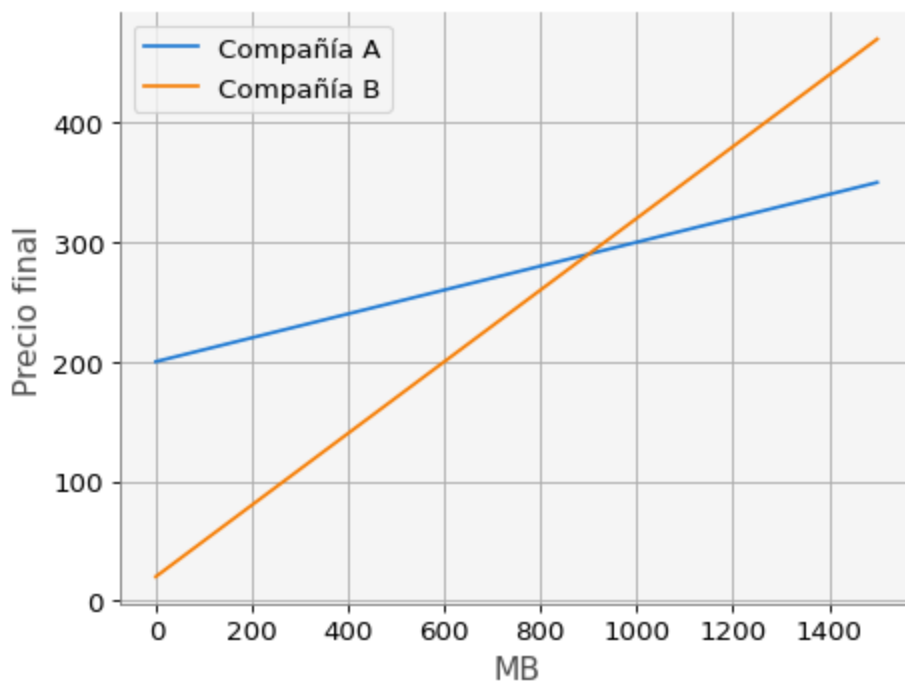
1 | Tu resultado es correcto.

```
quizz.eval_numeric('2', PB)
```

2 | Tu resultado es correcto.

```
# Gráfica de ambos casos
plt.plot(x, PA, label = 'Compañía A')
plt.plot(x, PB, label = 'Compañía B')

# Decoración de la gráfica
plt.xlabel('MB')
plt.ylabel('Precio final')
plt.legend()
plt.grid()
plt.show()
```



¿Qué observamos en la figura anterior?

Para decidir cuál de los dos compañías elegir, debemos saber cuantos MB gastamos al mes. En la figura se ve que al principio, con pocos MB usados conviene contratar a la compañía B. Pero después, si hacemos uso intenso de nuestras redes sociales, el consumo de MB aumenta y como consecuencia el precio de la compañía A es más barato.

¿Será posible determinar con precisión el punto de cruce de las rectas?

Sistema de ecuaciones lineales.

Las ecuaciones (1) tienen la forma típica de una recta: $y = mx + b$

Para la compañía A tenemos que $m = 0.10$ y $b = 200$, mientras que para la compañía B tenemos $m = 0.35$ y $b = 20$, entonces escribimos:

$$\begin{aligned} y &= 0.10x + 200 \\ y &= 0.35x + 20 \end{aligned}$$

Ahora, es posible escribir las ecuaciones de las líneas rectas en forma de un sistema de ecuaciones lineales como sigue:

$$\begin{bmatrix} 0.10 & -1 \\ 0.35 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -200 \\ -20 \end{bmatrix} \quad (2)$$

¿Puede verificar que el sistema (2) es correcto?

Si resolvemos el sistema (2) entonces será posible conocer de manera precisa el cruce de las rectas.

6.0.2 Ejercicio 2. Solución del sistema lineal.

1. En el siguiente código, complete los datos de la matriz **A** y el vector **b** de acuerdo con el sistema (2).

```
# Definimos la matriz A y el vector b
# A = np.array([[ ], [ ]])
# B = np.array([ [ ] ])
#
#### BEGIN SOLUTION
A = np.array([[0.10, -1.], [0.30, -1.] ])
b = np.array([[-200.0, -20.0]])

file_answer.write('3', A, 'Checa los elementos de la matriz A')
file_answer.write('4', b, 'Checa los elementos del vector b')
#### END SOLUTION

print("Matriz A : \n", A)
print("Vector b : \n", b)
```

El directorio `:/home/jovyan/macti_notes/notebooks/.ans/SMM/` ya existe
Respuestas y retroalimentación almacenadas.

Matriz A :

```
[[ 0.1 -1. ]
 [ 0.3 -1. ]]
```

Vector b :

```
[[ -200.  -20. ]]
```

```
quizz.eval_numeric('3', A)
```

3 | Tu resultado es correcto.

```
quizz.eval_numeric('4', b)
```

4 | Tu resultado es correcto.

2. Investigue como usar la función [numpy.linalg.solve\(\)](#) para resolver el sistema de ecuaciones. Resuelva el sistema y guarde la solución en el vector `xsol`.

```
# Resolvemos el sistema de ecuaciones lineal
# xsol = np.linalg.solve( ... )
#
#### BEGIN SOLUTION
xsol = np.linalg.solve(A,b.T)

file_answer.write('5', xsol, 'Verifica que usaste correctamente la función np.linalg.solve()')
#### END SOLUTION

print("Solución del sistema: \n", xsol)
```

El directorio `:/home/jovyan/macti_notes/notebooks/.ans/SMM/` ya existe
Respuestas y retroalimentación almacenadas.

Solución del sistema:

```
[[900.]
 [290.]]
```

```
quizz.eval_numeric('5', xsol)
```

5 | Tu resultado es correcto.

```
# Dot product
# rhs = np.dot( ... )
#
#### BEGIN SOLUTION
rhs = np.dot(A, xsol)

file_answer.write('6', rhs, 'Checa que la representación de cada número sea la correcta')
file_answer.to_file('06')
#### END SOLUTION

print(rhs)
```

El directorio `:/home/jovyan/macti_notes/notebooks/.ans/SMM/` ya existe
Respuestas y retroalimentación almacenadas.

```
[[ -200.]
 [ -20.]]
```

```
quizz.eval_numeric('6', rhs)
```

6 | Tu resultado es correcto.

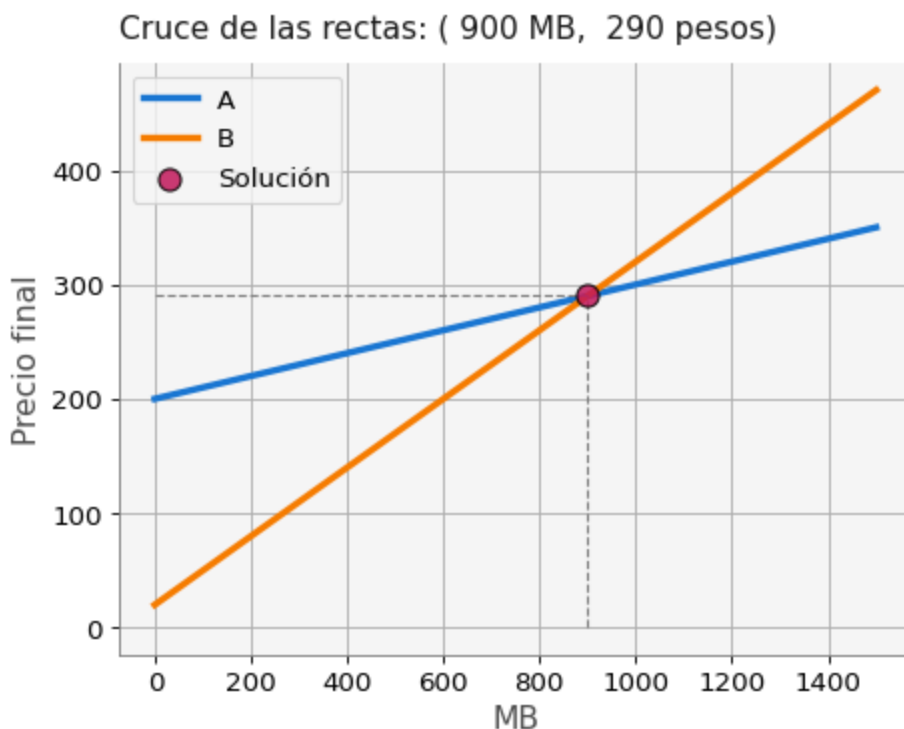
Si todo se hizo correctamente, el siguiente código debe graficar las rectas de las dos compañías y en el punto donde se cruzan

```
# Gráfica de las líneas de cada compañía
plt.plot(x, PA, lw=3, label = 'A')
plt.plot(x, PB, lw=3, label = 'B')

# Punto de cruce de las líneas rectas
plt.scatter(xsol[0], xsol[1], fc = 'C3', ec = 'k', s = 100, alpha=0.85, zorder=5,

# Decoración de la gráfica
plt.xlabel('MB')
plt.ylabel('Precio final')
plt.title('Cruce de las rectas: ({:4.0f} MB, {:4.0f} pesos)'.format(xsol[0][0], x
plt.vlines(xsol[0][0], 0, xsol[1][0], ls='--', lw=1.0, color='gray')
plt.hlines(xsol[1][0], 0, xsol[0][0], ls='--', lw=1.0, color='gray')

plt.grid(True)
plt.legend()
plt.show()
```



7 Métodos iterativos para la solución de sistemas de ecuaciones lineales

Objetivo.

Describir e implementar los algoritmos de Jacobi, Gauss-Seidel y SOR para la solución de sistemas de ecuaciones lineales.

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under

[Attribution-ShareAlike 4.0 International](#) 

Trabajo realizado con el apoyo del Programa UNAM-DGAPA-PAPIME PE101922

```
import numpy as np
import ipywidgets as widgets
import macti.visual as mvis
```

8 Cruce de dos rectas.

Las siguientes dos rectas se cruzan en algún punto.

$$\begin{aligned} 3x + 2y &= 2 \\ 2x + 6y &= -8 \end{aligned}$$

Las ecuaciones de las rectas se pueden escribir como:

$$\begin{aligned} \frac{3}{2}x + y &= 1 \\ \frac{2}{6}x + y &= -\frac{8}{6} \end{aligned} \implies \begin{aligned} y &= m_1x + b_1 \\ y &= m_2x + b_2 \end{aligned} \text{ donde } \begin{aligned} m_1 &= -\frac{3}{2} & b_1 &= 1 \\ m_2 &= -\frac{2}{6} & b_2 &= -\frac{8}{6} \end{aligned}$$

Ahora realizaremos la gráfica de las rectas:

8.1 Definición y gráfica de las rectas

8.2 Ejercicio 1.

En la siguiente celda se define el dominio x para las líneas rectas, los parámetros para construir la línea recta 1 y su construcción. De la misma manera define los parámetros y construye la recta 2. Si todo lo hiciste correctamente, la celda de graficación mostrará las gráficas de las líneas rectas.

```
from macti.evaluation import FileAnswer, Quizz
#file_anser = FileAnswer()
#quizz = Quizz()
```

```
# Dominio
x = np.linspace(-3,6,10)

# Línea recta 1
m1 = -3/2
b1 = 1
y1 = m1 * x + b1

# Línea recta 2
# m2 = ...
# b2 = ...
# y2 = ...

#### BEGIN SOLUTION
m2 = -2/6
b2 = -8/6
y2 = m2 * x + b2

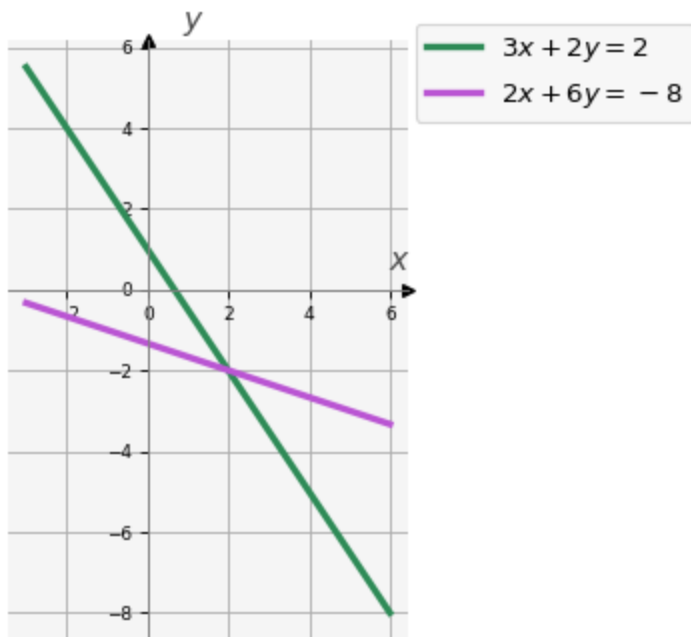
#file_answer('1', m2, 'm2 incorrecta revisa el valor del parámetro.')
#file_answer('2', b2, 'b2 incorrecta revisa el valor del parámetro.')
#file_answer('3', y2, 'y2 no está definida correctamente.')
#### END SOLUTION
```

```
#quizz.eval_numeric('1', m2)
#quizz.eval_numeric('2', b2)
#quizz.eval_numeric('3', y2)
```

Gráfica de las líneas rectas.

```
v = mvis.Plotter(1,1,[dict(aspect='equal')],title='Cruce de rectas')
v.set_coordsys(1)
v.plot(1, x, y1, lw = 3, c = 'seagreen', label = '$3x+2y=2$') # Línea recta 1
v.plot(1, x, y2, lw = 3, c = 'mediumorchid',label = '$2x+6y=-8$') # Línea recta 2
v.legend(ncol = 1, frameon=True, loc='best', bbox_to_anchor=(1.75, 1.05))
v.grid()
v.show()
```


Cruce de rectas



8.3 Sistemas lineales.

Las ecuaciones de las rectas se pueden escribir en forma de un sistema lineal:

$$\begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 2 \\ -8 \end{bmatrix} \quad (1)$$

Podemos calcular el cruce de las rectas resolviendo el sistema lineal:

8.4 Ejemplo 1.

Definir el sistema lineal y resolverlo. Posteriormente graficar las rectas y el punto solución.

El sistema lineal se puede resolver directamente con la función `np.linalg.solve()` como sigue:

```
A = np.array([[3, 2],[2,6]] )
b = np.array([2,-8])
print("Matriz A : \n",A)
print("Vector b : \n", b)

sol = np.linalg.solve(A,b[0]) # Función del módulo linalg para resolver el sistema
print("Solución del sistema: ", sol)
```

Matriz A :
[[3 2]

```
[2 6]]
```

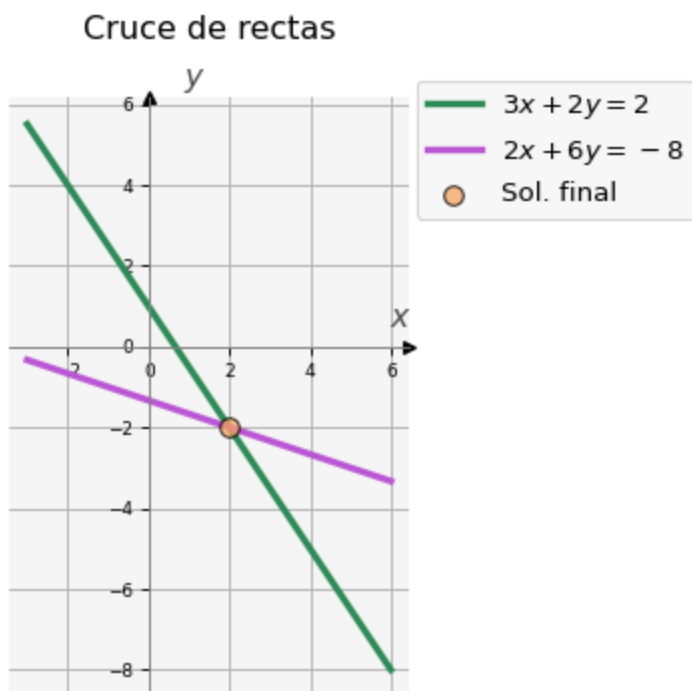
Vector b :

```
[[ 2 -8]]
```

Solución del sistema: [2. -2.]

Gráfica de las líneas rectas y el punto de cruce (solución).

```
v = mvis.Plotter(1,1,[dict(aspect='equal')],title='Cruce de rectas')
v.set_coordsys(1)
v.plot(1, x, y1, lw = 3, c = 'seagreen', label = '$3x+2y=2$') # Línea recta 1
v.plot(1, x, y2, lw = 3, c = 'mediumorchid', label = '$2x+6y=-8$') # Línea recta
v.scatter(1, sol[0], sol[1], fc='sandybrown', ec='k', s = 75, alpha=0.75, zorder=
v.legend(ncol = 1, frameon=True, loc='best', bbox_to_anchor=(1.75, 1.05))
v.grid()
v.show()
```



En general, un sistema de ecuaciones lineales de $n \times n$ se escribe como sigue:

$$\begin{array}{ccccccc}
 a_{11}x_1 & + & a_{12}x_2 & + \cdots + & a_{1n}x_n & = & b_1 \\
 a_{21}x_1 & + & a_{22}x_2 & + \cdots + & a_{2n}x_n & = & b_2 \\
 \vdots & & \vdots & & \vdots & & \vdots \\
 a_{i1}x_1 & + & a_{i2}x_2 & + \cdots + & a_{in}x_n & = & b_i \\
 \vdots & & \vdots & & \vdots & & \vdots \\
 a_{n1}x_1 & + & a_{n2}x_2 & + \cdots + & a_{nn}x_n & = & b_n
 \end{array}$$

Es posible usar diferentes métodos para resolver este tipo de sistemas. Veamos tres de ellos.

9 Método de Jacobi

- En este método, de la primera ecuación se despeja x_1 ; de la segunda ecuación se despeja x_2 ; y a sí sucesivamente, de tal manera que obtenemos:

$$\begin{aligned} x_1 &= (b_1 - (a_{12}x_2 + \cdots + a_{1n}x_n))/a_{11} \\ x_2 &= (b_2 - (a_{21}x_1 + \cdots + a_{2n}x_n))/a_{22} \\ &\vdots \\ x_i &= (b_i - (a_{i1}x_1 + \cdots + a_{in}x_n))/a_{ii} \\ &\vdots \\ x_n &= (b_n - (a_{n1}x_1 + \cdots + a_{nn-1}x_{n-1}))/a_{nn} \end{aligned}$$

- Suponemos ahora que tenemos una solución inicial aproximada $\mathbf{x}^0 = [x_1^0, \dots, x_n^0]$. Usando esta solución inicial, es posible hacer una nueva aproximación para obtener $\mathbf{x}^1 = [x_1^1, \dots, x_n^1]$ como sigue:

$$\begin{aligned} x_1^1 &= (b_1 - (a_{12}x_2^0 + \cdots + a_{1n}x_n^0))/a_{11} \\ x_2^1 &= (b_2 - (a_{21}x_1^0 + \cdots + a_{2n}x_n^0))/a_{22} \\ &\vdots \\ x_i^1 &= (b_i - (a_{i1}x_1^0 + \cdots + a_{in}x_n^0))/a_{ii} \\ &\vdots \\ x_n^1 &= (b_n - (a_{n1}x_1^0 + \cdots + a_{nn-1}x_{n-1}^0))/a_{nn} \end{aligned}$$

- En general para $i = 1, \dots, n$ y $k = 1, 2, \dots$ tenemos:

$$x_i^k = \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} x_j^{k-1} \right)$$

- En términos de matrices, la **iteración de Jacobi** se escribe:

$$\mathbf{x}^k = -\mathbf{D}^{-1}\mathbf{B}\mathbf{x}^{k-1} + \mathbf{D}^{-1}\mathbf{b}$$

donde \mathbf{D} es la matriz diagonal y $\mathbf{B} = \mathbf{A} - \mathbf{D}$.

- El cálculo de cada componente x_i^k es independiente de las otras componentes, por lo que este método se conoce también como de *desplazamientos simultáneos*.

9.1 Algoritmo Jacobi.

En general, podemos definir el siguiente algoritmo para el método de Jacobi.

$$x_i^k = \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} x_j^{k-1} \right)$$

Algoritmo 1 Jacobi

```

1: INPUT:  $\mathbf{x}_{\text{ini}}$ ,  $A$ ,  $b$ ,  $\text{tol}$ ,  $\text{kmax}$ .
2:  $\mathbf{xold} = \mathbf{xini}$  // Solución inicial
3:  $k = 0$ 
4: while error > tol and  $k < \text{kmax}$  do
5:   for  $i := 1, \dots, N$  do
6:     // En el siguiente ciclo nos saltamos  $i$ 
7:     for  $j := 1, \dots, i-1, i+1, \dots, N$  do
8:        $\mathbf{xnew}_i \leftarrow \mathbf{xnew}_i + A_{i,j} * \mathbf{xold}_j$  // Se usa la aprox. anterior
9:     end for
10:     $\mathbf{xnew}_i \leftarrow (b_i - \mathbf{xnew}_i) / A_{i,i}$ 
11:    error =  $\|\mathbf{xnew} - \mathbf{xold}\|_2$ 
12:  end for
13:   $k \leftarrow k + 1$ 
14: end while
15: OUTPUT:  $\mathbf{x}_{\text{new}}$  // Solución final aproximada
  
```

Observa que en este algoritmo hay un ciclo **while** el cual termina cuando el error es menor o igual que una tolerancia **tol** o se ha alcanzado un número máximo de iteraciones **kmax**. En la línea **11** se calcula el error, que en términos matemáticos se define como $\text{error} = \|\mathbf{x}^k - \mathbf{x}\|$ donde \mathbf{x}^k es la aproximación de la iteración k -ésima y \mathbf{x} es la solución exacta. En muchas ocasiones no se tiene acceso a la solución exacta por lo que se compara con la solución de la iteración anterior, es decir $\text{error} = \|\mathbf{x}^k - \mathbf{x}^{k-1}\|$. En los ejemplos que siguen si tenemos la solución exacta, por lo que haremos la comparación con ella.

9.2 Implementación.

```

def jacobi(A,b,tol,kmax,xi, yi):
    N = len(b[0])
    xnew = np.zeros(N)
    xold = np.zeros(N)
    x = np.array([2, -2]) # Solución exacta

    # Solución inicial
    xold[0] = xi
    xold[1] = yi

    xs = [xi]
    ys = [yi]

    e = 10
    error = []

    k = 0
    print('{:^2} {:^10} {:^12} {:^12}'.format(' i ', 'Error', 'x0', 'x1'))
  
```

```

while(e > tol and k < kmax) :
    for i in range(0,N): # se puede hacer en paralelo
        xnew[i] = 0
        for j in range(0,i):
            xnew[i] += A[i,j] * xold[j]
        for j in range(i+1,N):
            xnew[i] += A[i,j] * xold[j]
        xnew[i] = (b[0,i] - xnew[i]) / A[i,i]

    # Almacenamos la solución actual
    xs.append(xnew[0])
    ys.append(xnew[1])

    e = np.linalg.norm(xnew-x, 2) # Cálculo del error
    error.append(e)
    k += 1
    xold[:] = xnew[:]
    print('{:2d} {:10.9f} ({:10.9f}, {:10.9f})'.format(k, e, xnew[0], xnew[1]))
    return xnew, np.array(xs), np.array(ys), error, k

```

9.3 Ejemplo 3. Aplicación del método de Jacobi.

Haciendo uso de la función `jacobi` definida en la celda anterior, aproxima la solución del sistema de ecuaciones (1). Utiliza la solución inicial $(x_i, y_i) = (-2, 2)$, una tolerancia `tol` = 1×10^{-5} y `kmax` = 50 iteraciones.

```

# Solución inicial
(xi, yi) = (-2, 2)
tol = 1e-5
kmax = 50

# Ejecución del método de Jacobi
solJ, xs, ys, eJ, itJ = jacobi(A, b, tol, kmax, xi, yi)

```

i	Error	x0	x1
1	2.981423970	(-0.666666667, -0.666666667)	
2	1.257078722	(1.111111111, -1.111111111)	
3	0.662538660	(1.407407407, -1.703703704)	
4	0.279350827	(1.802469136, -1.802469136)	
5	0.147230813	(1.868312757, -1.934156379)	
6	0.062077962	(1.956104252, -1.956104252)	
7	0.032717959	(1.970736168, -1.985368084)	
8	0.013795103	(1.990245389, -1.990245389)	
9	0.007270657	(1.993496926, -1.996748463)	
10	0.003065578	(1.997832309, -1.997832309)	
11	0.001615702	(1.998554873, -1.999277436)	

```

12 0.000681240 (1.999518291, -1.999518291)
13 0.000359045 (1.999678861, -1.999839430)
14 0.000151387 (1.999892954, -1.999892954)
15 0.000079788 (1.999928636, -1.999964318)
16 0.000033641 (1.999976212, -1.999976212)
17 0.000017731 (1.999984141, -1.999992071)
18 0.000007476 (1.999994714, -1.999994714)

```

Observa que la función `jacobi()` regresa 5 valores: * `solJ` la solución obtenida, * `xs` y `ys` componentes de las soluciones aproximadas en cada paso, * `eJ` el error con respecto a la solución exacta e * `itJ` el número de iteraciones realizadas.

A continuación graficamos como es que la solución se va aproximando con este método.

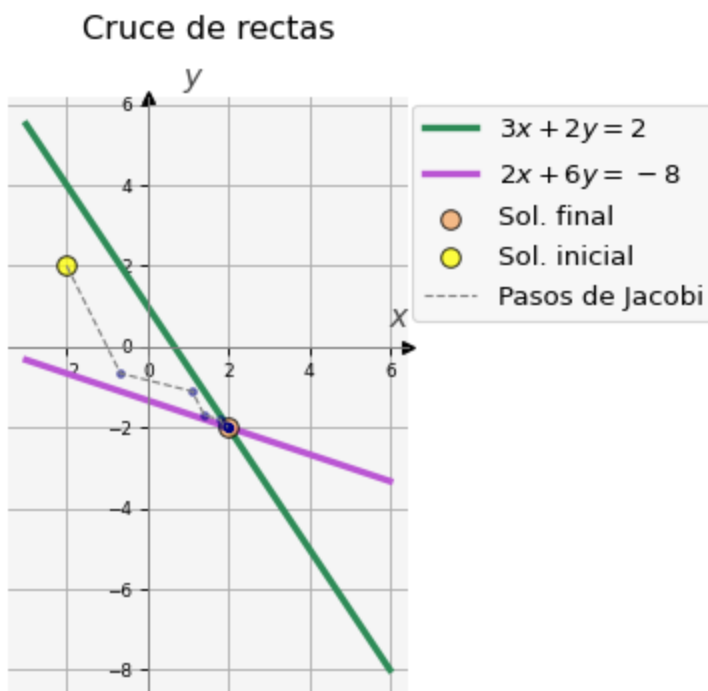
```

v = mvis.Plotter(1,1,[dict(aspect='equal')],title='Cruce de rectas')
v.set_coordsys(1)
v.plot(1, x, y1, lw = 3, c = 'seagreen', label = '$3x+2y=2$') # Línea recta 1
v.plot(1, x, y2, lw = 3, c = 'mediumorchid', label = '$2x+6y=-8$') # Línea recta
v.scatter(1, sol[0], sol[1], fc='sandybrown', ec='k', s = 75, alpha=0.75, zorder=

# Graficamos los pasos
v.scatter(1, xs[0], ys[0], fc='yellow', ec='k', s = 75, alpha=0.75, zorder=8, lat
v.scatter(1, xs[1:], ys[1:], c='navy', s = 10, alpha=0.5, zorder=8)
v.plot(1, xs, ys, c='grey', ls = '--', lw=1.0, zorder=8, label='Pasos de Jacobi')

v.legend(ncol = 1, frameon=True, loc='best', bbox_to_anchor=(1.80, 1.01))
v.grid()
v.show()

```



9.4 Cálculo del error

- Definimos $e_i^k = x_i^k - x_i$ como la diferencia entre la i -ésima componente de la solución exacta y la i -ésima componente de la k -ésima iteración, de tal manera que $\mathbf{e} = [e_1, \dots, e_n]^T$ es el vector error.
- Aplicando una vez la iteración de Jacobi para x_i y x_i^{k+1} podemos escribir la diferencia como sigue:

$$\begin{aligned} |e_i^{k+1}| &= |x_i^{k+1} - x_i| \\ |e_i^{k+1}| &= \left| \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} x_j^k \right) - \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} x_j \right) \right| \\ |e_i^{k+1}| &= \left| - \sum_{j \neq i} \frac{a_{i,j}}{a_{i,i}} (x_j^k - x_j) \right| \\ |e_i^{k+1}| &= \left| - \sum_{j \neq i} \frac{a_{i,j}}{a_{i,i}} e_j^k \right| \leq \sum_{j \neq i} \left| \frac{a_{i,j}}{a_{i,i}} \right| \|\mathbf{e}^k\|_\infty, \quad \forall i, k. \end{aligned}$$

- En particular:

$$\max_{1 \leq i \leq n} (|e_i^{k+1}|) = \|\mathbf{e}^{k+1}\|_\infty \leq \sum_{j \neq i} \left| \frac{a_{i,j}}{a_{i,i}} \right| \|\mathbf{e}^k\|_\infty$$

- Definimos $K = \max_{1 \leq i \leq n} \sum_{j \neq i} \left| \frac{a_{i,j}}{a_{i,i}} \right|$ entonces:

$$\begin{aligned} \|\mathbf{e}^{k+1}\|_\infty &\leq K \|\mathbf{e}^k\|_\infty \leq K (K \|\mathbf{e}^{k-1}\|_\infty) \leq \dots \leq K^k \|\mathbf{e}^1\|_\infty \\ \|\mathbf{e}^{k+1}\|_\infty &\leq K^k \|\mathbf{e}^1\|_\infty \end{aligned}$$

- Si $K < 1$ entonces $\mathbf{e}^k \rightarrow 0$ cuando $k \rightarrow \infty$
- La condición $K < 1$ implica:

$$\sum_{j \neq i} |a_{i,j}| < |a_{i,i}|, \forall i$$

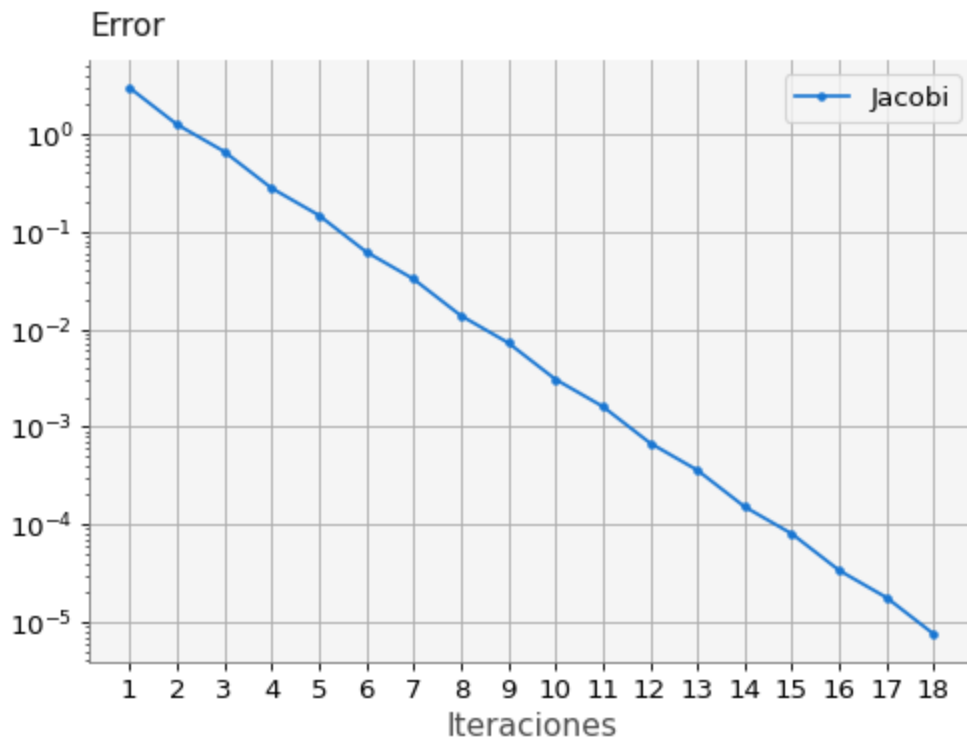
A continuación graficamos el error que se va obteniendo en cada paso del método:

```
# Lista con el número de las iteraciones
l_itJ = list(range(1,itJ+1))

# Parámetros para los ejes
a_p = dict(yscale='log', xlabel='Iteraciones', xticks = l_itJ)

# Gráfica del error
v = mvis.Plotter(1,1,[a_p])
v.axes(1).set_title('Error', loc='left')
```

```
v.plot(1, l_itJ, eJ, marker='.', label='Jacobi') # Error eJ
v.legend()
v.grid()
```



10 Método de Gauss-Seidel

- La principal diferencia con el método de Jacobi es que las ecuaciones se analizan en un orden determinado.
- Por ejemplo, si realizamos el cálculo en orden ascendente y ya hemos evaluado x_1 y x_2 , para evaluar x_3 haríamos lo siguiente:}

$$\begin{aligned}\underline{x}_1^1 &= (b_1 - (a_{12}x_2^0 + a_{13}x_3^0 + \cdots + a_{1n}x_n^0)) / a_{11} \\ \underline{x}_2^1 &= (b_2 - (a_{21}\underline{x}_1^1 + a_{23}x_3^0 + \cdots + a_{2n}x_n^0)) / a_{22} \\ x_3 &= (b_3 - (a_{31}\underline{x}_1^1 + a_{32}\underline{x}_2^1 + \cdots + a_{3n}x_n^0)) / a_{33}\end{aligned}$$

- En general la fórmula del método es como sigue: $x_i^k = (b_i - \sum_{j < i} a_{ij}x_j^k - \sum_{j > i} a_{ij}x_j^{k-1}) / a_{ii}$
- Este algoritmo es serial dado que cada componente depende de que las componentes previas se hayan calculado (*desplazamientos sucesivos*).
- El valor de la nueva iteración \mathbf{x}^k depende del orden en que se examinan las componentes. Si se cambia el orden, el valor de \mathbf{x}^k cambia.

10.1 Algoritmo Gauss-Seidel.

En general, podemos definir el siguiente algoritmo para el método de Gauss-Seidel.

$$x_i^k = \frac{1}{a_{i,i}} \left(b_i - \sum_{j < i} a_{i,j} x_j^k - \sum_{j > i} a_{i,j} x_j^{k-1} \right)$$

Algoritmo 2 Gauss Seidel

```

1: INPUT:  $\mathbf{x}_{\text{ini}}$ ,  $A$ ,  $b$ ,  $\text{tol}$ ,  $\text{kmax}$ .
2:  $\mathbf{xold} = \mathbf{xini}$  // Solución inicial
3:  $k = 0$ 
4: while error > tol and  $k < \text{kmax}$  do
5:   for  $i := 1, \dots, N$  do
6:     for  $j := 1, \dots, i - 1$  do
7:        $\mathbf{xnew}_i \leftarrow \mathbf{xnew}_i + A_{i,j} * \mathbf{xnew}_j$  // Se usa la aprox. nueva
8:     end for
9:     for  $j := i + 1, \dots, N$  do
10:       $\mathbf{xnew}_i \leftarrow \mathbf{xnew}_i + A_{i,j} * \mathbf{xold}_j$  // Se usa la aprox. anterior
11:    end for
12:     $\mathbf{xnew}_i \leftarrow (b_i - \mathbf{xnew}_i) / A_{i,i}$ 
13:    error =  $\|\mathbf{xnew} - \mathbf{xold}\|_2$ 
14:  end for
15:   $k \leftarrow k + 1$ 
16: end while
17: OUTPUT:  $\mathbf{x}_{\text{new}}$  // Solución final aproximada

```

Se aplican los mismo comentarios que para el algoritmo de Jacobi.

10.2 Implementación.

```

def gauss_seidel(A,b,tol,kmax,xi,yi):
    N = len(b[0])
    xnew = np.zeros(N)
    xold = np.zeros(N)
    x = np.array([2, -2]) # Solución exacta

    # Solución inicial
    xold[0] = xi
    xold[1] = yi

    xs = [xi]
    ys = [yi]

    e = 10
    error = []

    k = 0
    print('{:^2} {:^10} {:^12} {:^12}'.format(' i ', 'Error', 'x0', 'x1'))

```

```

while(e > tol and k < kmax) :
    for i in range(0,N): # se puede hacer en paralelo
        xnew[i] = 0
        for j in range(0,i):
            xnew[i] += A[i,j] * xnew[j]
        for j in range(i+1,N):
            xnew[i] += A[i,j] * xold[j]
        xnew[i] = (b[0,i] - xnew[i]) / A[i,i]

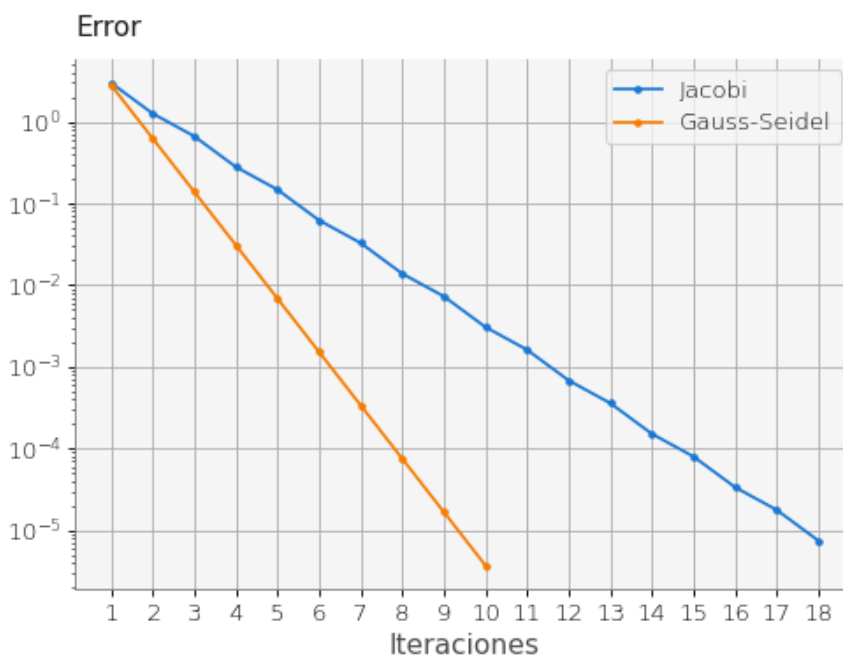
    # Almacenamos la solución actual
    xs.append(xnew[0])
    ys.append(xnew[1])

    e = np.linalg.norm(xnew-x,2) # Cálculo del error
    error.append(e)
    k += 1
    xold[:] = xnew[:]
    print('{:2d} {:10.9f} ({:10.9f}, {:10.9f})'.format(k, e, xnew[0], xnew[1])
    return xnew, np.array(xs), np.array(ys), error, k

```

10.3 Ejercicio 2.

Haciendo uso de la función `gauss_seidel()` definida en la celda anterior, aproxima la solución del sistema de ecuaciones del Ejemplo 1. Utiliza la solución inicial $(x_i, y_i) = (-2, 2)$, una tolerancia `tol` = 1×10^{-5} y `kmax` = 50 iteraciones. Utiliza las variables `solG`, `xs`, `ys`, `eG` e `itG` para almacenar la salida de la función `gauss_seidel()`. Posteriormente grafica las rectas y cómo se va calculando la solución con este método (puedes usar el mismo código que en el caso de Jacobi). Grafica también los errores para el método de Jacobi y para el de Gauss-Seidel, deberías obtener una imagen como la siguiente:



Cálculo de la solución con Gauss-Seidel

```
# Solución inicial
# xi, yi =
# tol =
# kmax =

# Método de Gauss-Seidel
# ...

#### BEGIN SOLUTION
# Solución inicial
xi, yi = -2, 2
tol = 1e-5
kmax = 50

# Método de Gauss-Seidel
solG, xs, ys, eG, itG = gauss_seidel(A, b, tol, kmax, xi, yi)

#file_answer.write('4', solG, 'solG es incorrecta: revisa la llamada y ejecución
#file_answer.write('5', eG[-1], 'eG[-1] es incorrecto: revisa la llamada y ejecu
#file_answer.write('6', itG, 'itG es incorrecto: revisa la llamada y ejecución c

#### END SOLUTION
```

i	Error	x0	x1
1	2.810913476	(-0.666666667,	-1.111111111)
2	0.624647439	(1.407407407,	-1.802469136)
3	0.138810542	(1.868312757,	-1.956104252)
4	0.030846787	(1.970736168,	-1.990245389)
5	0.006854842	(1.993496926,	-1.997832309)
6	0.001523298	(1.998554873,	-1.999518291)
7	0.000338511	(1.999678861,	-1.999892954)
8	0.000075225	(1.999928636,	-1.999976212)
9	0.000016717	(1.999984141,	-1.999994714)
10	0.000003715	(1.999996476,	-1.999998825)

```
#quizz.eval_numeric('4', solG)
#quizz.eval_numeric('5', eG[-1])
#quizz.eval_numeric('6', itG)
```

Gráfica de las rectas, la solución y los pasos realizados

```
# Puedes usar el mismo código que en el caso anterior.

#### BEGIN SOLUTION
v = mvis.Plotter(1,1,[dict(aspect='equal')],title='Cruce de rectas')
v.set_coordsys(1)
v.plot(1, x, y1, lw = 3, c = 'seagreen', label = '$3x+2y=2$') # Línea recta 1
```

```

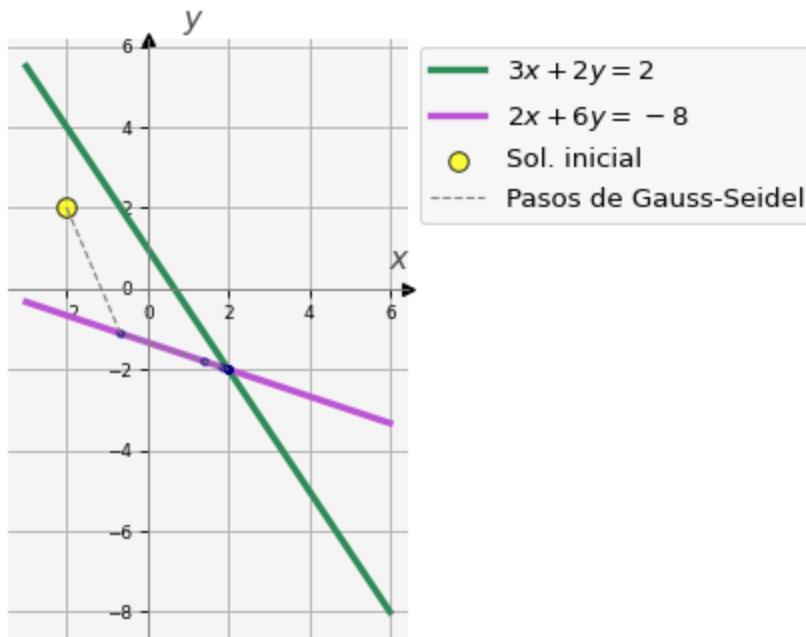
v.plot(1, x, y2, lw = 3, c = 'mediumorchid', label = '$2x+6y=-8$') # Línea recta

# Graficamos los pasos
v.scatter(1, xs[0], ys[0], fc='yellow', ec='k', s = 75, alpha=0.75, zorder=8, label='')
v.scatter(1, xs[1:], ys[1:], c='navy', s = 10, alpha=0.5, zorder=8)
v.plot(1, xs, ys, c='grey', ls = '--', lw=1.0, zorder=8, label='Pasos de Gauss-Seidel')

v.legend(ncol = 1, frameon=True, loc='best', bbox_to_anchor=(2.05, 1.01))
v.grid()
v.show()
#### END SOLUTION

```

Cruce de rectas



Graficación de los errores de Jacobi y Gauss-Seidel

```

# Utiliza el código del caso anterior adaptado para que pueda graficar ambos errores

#### BEGIN SOLUTION
# Lista con el número de las iteraciones máxima
it_max = max(itJ, itG)+1
l_it_max = list(range(1,it_max))

# Listas con el número de las iteraciones para cada algoritmo
l_itJ = list(range(1,itJ+1))
l_itG = list(range(1,itG+1))

# Parámetros para los ejes
a_p = dict(yscale='log', xlabel='Iteraciones', xticks = l_it_max)

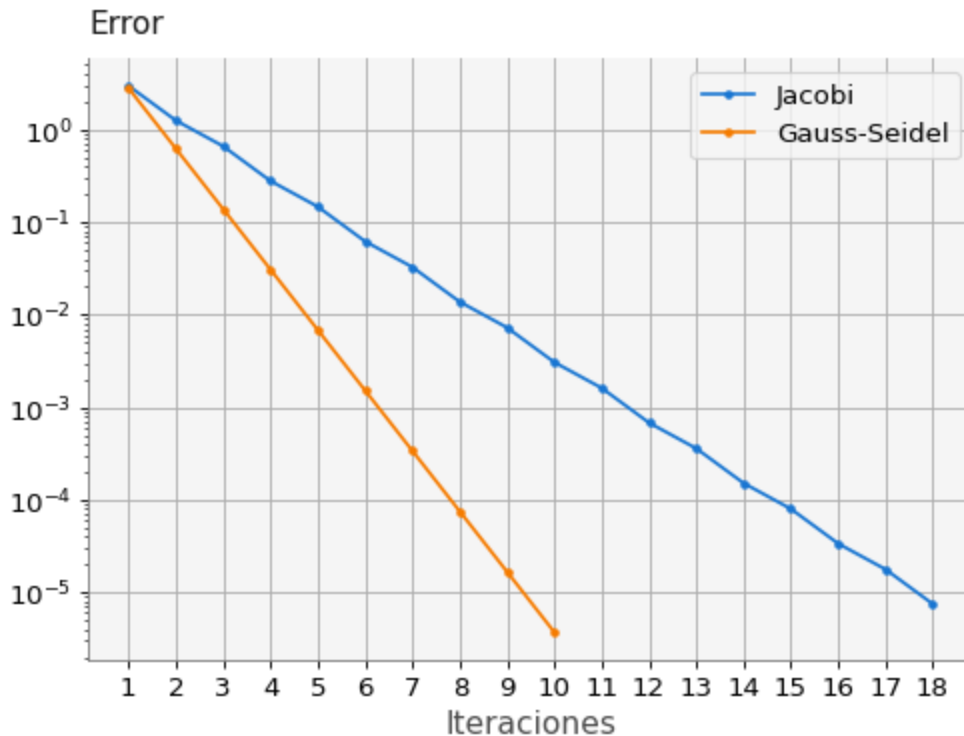
# Gráficas del error
v = mvis.Plotter(1,1,[a_p])

```

```

v.axes(1).set_title('Error', loc='left')
v.plot(1, l_itJ, eJ, marker='.', label='Jacobi')
v.plot(1, l_itG, eG, marker='.', label='Gauss-Seidel')
v.legend()
v.grid()
#### END SOLUTION

```



11 Método de Sobrerrelajación sucesiva (*Successive Overrelaxation, SOR*)

- Se obtiene aplicando una extrapolación a la iteración de Gauss-Seidel.
- Esta extrapolación es un promedio pesado entre la iteración actual y la anterior:

$$x_i^k = \omega \bar{x}_i^k + (1 - \omega)x_i^{k-1}$$

donde \bar{x} denota una iteración de Gauss-Seidel y ω es el factor de extrapolación.

- En términos de matrices tenemos: $A^k = (-)^{-1} + (1 -)^{k-1}$
- $(-)^{-1}$
- Elegir la ω óptima no es simple, aunque se sabe que si ω está fuera del intervalo $(0, 2)$ el método falla.

11.1 Implementación 3.

```

def sor(A,b,tol,kmax,w,xi,yi):
    N = len(b[0])
    xnew = np.zeros(N)
    xold = np.zeros(N)
    x = np.array([2, -2]) # Solución exacta

    # Solución inicial
    xold[0] = xi
    xold[1] = yi

    xs = [xi]
    ys = [yi]

    e = 10
    error = []

    k = 0
    while(e > tol and k < kmax) :
        for i in range(0,N): # se puede hacer en paralelo
            sigma = 0
            for j in range(0,i):
                sigma += A[i,j] * xnew[j]
            for j in range(i+1,N):
                sigma += A[i,j] * xold[j]
            sigma = (b[0,i] - sigma) / A[i,i]
            xnew[i] = xold[i] + w * (sigma - xold[i])

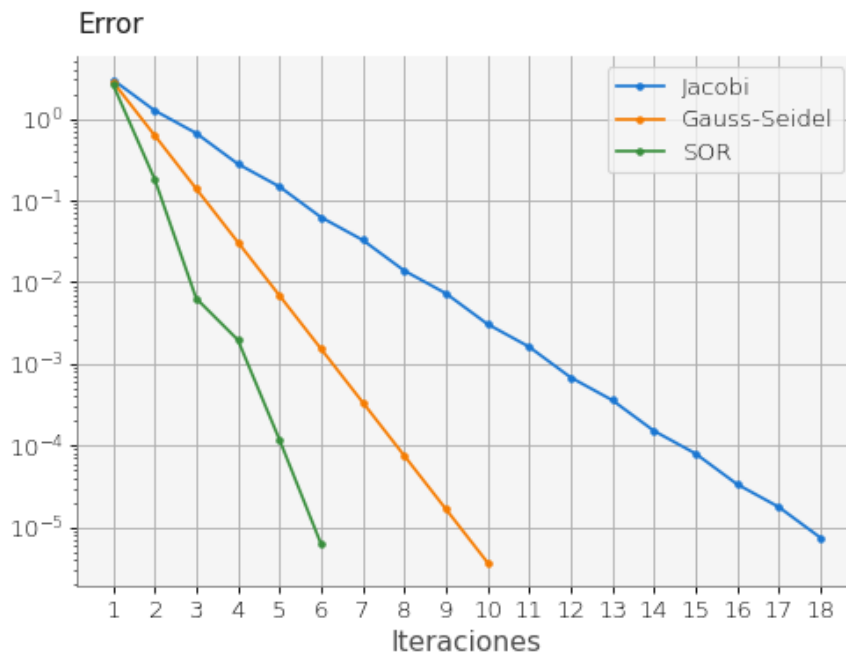
        # Almacenamos la solución actual
        xs.append(xnew[0])
        ys.append(xnew[1])

        e = np.linalg.norm(xnew-x, 2) # Cálculo del error
        error.append(e)
        k += 1
        xold[:] = xnew[:]
        print('{:2d} {:10.9f} ({:10.9f}, {:10.9f})'.format(k, e, xnew[0], xnew[1]))
    return xnew, np.array(xs), np.array(ys), error, k

```

11.2 Ejercicio 3.

Haciendo uso de la función `sor()` definida en la celda anterior, aproxima la solución del sistema de ecuaciones del Ejercicio 1. Utiliza la solución inicial $(x_i, y_i) = (-2, 2)$, una tolerancia $tol = 1 \times 10^{-5}$ y $kmax = 50$ iteraciones. Elige el valor de $\omega = 1.09$. Utiliza las variables `solsOR`, `xs`, `ys`, `eSOR` e `itSOR` para almacenar la salida de la función `gauss_seidel()`. Posteriormente grafica las rectas y cómo se va calculando la solución con este método (puedes usar el mismo código que en el caso de Jacobi). Grafica también los errores para los tres métodos (Jacobi, Gauss-Seidel y SOR).



Cálculo de la solución con SOR

```
# Solución inicial
# xi, yi =
# tol =
# kmax =

# Método de SOR, probar con w = 1.09, 1.8, 1.99, 2.0
# w = ...
# ...

#### BEGIN SOLUTION
# Solución inicial
xi, yi = -2, 2
tol = 1e-5
kmax = 50

# Método de SOR, probar con w = 1.09, 1.8, 1.99, 2.0
w = 1.09
solSOR, xs, ys, eSOR, itSOR = sor(A, b, tol, kmax, w, xi, yi)

#file_answer.write('7', solSOR, 'solSOR es incorrecta: revisa la llamada y ejecu
#file_answer.write('8', eSOR[-1], 'eSOR[-1] es incorrecto: revisa la llamada y ej
#file_answer.write('9', itSOR, 'itSOR es incorrecto: revisa la llamada y ejecu

#### END SOLUTION
```

```
1 2.608651498 (-0.546666667, -1.434711111)
2 0.182203110 (1.818423407, -1.984903171)
3 0.006309667 (2.005371531, -2.003310371)
```

```

4 0.001963366 (2.001922098, -2.000400429)
5 0.000118187 (2.000117990, -2.000006831)
6 0.000006254 (1.999994345, -1.999997330)

```

```

#quizz.eval_numeric('7', solSOR)
#quizz.eval_numeric('8', eSOR[-1])
#quizz.eval_numeric('9', itSOR)

```

Gráfica de las rectas, la solución y los pasos realizados

Puedes usar el mismo código que en el caso anterior.

```

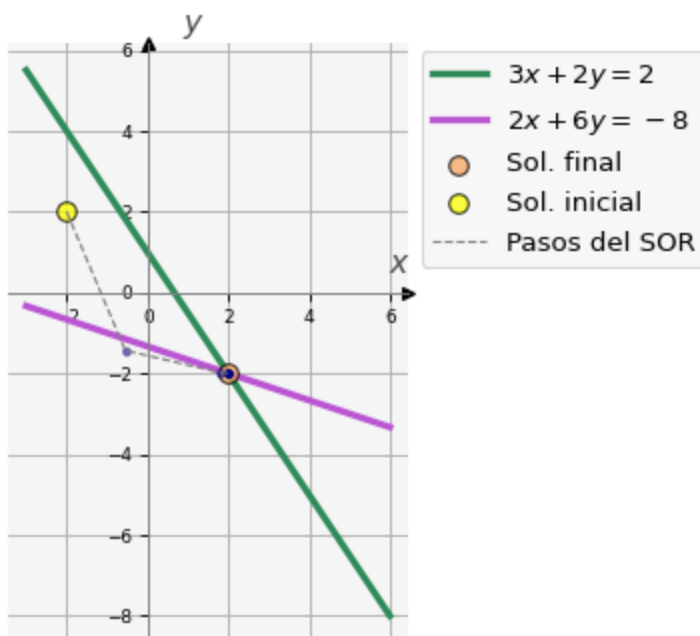
### BEGIN SOLUTION
v = mvis.Plotter(1,1,[dict(aspect='equal')],title='Cruce de rectas')
v.set_coordsys(1)
v.plot(1, x, y1, lw = 3, c = 'seagreen', label = '$3x+2y=2$') # Línea recta 1
v.plot(1, x, y2, lw = 3, c = 'mediumorchid', label = '$2x+6y=-8$') # Línea recta
v.scatter(1, sol[0], sol[1], fc='sandybrown', ec='k', s = 75, alpha=0.75, zorder=

# Graficamos los pasos
v.scatter(1, xs[0], ys[0], fc='yellow', ec='k', s = 75, alpha=0.75, zorder=8, lat
v.scatter(1, xs[1:], ys[1:], c='navy', s = 10, alpha=0.5, zorder=8)
v.plot(1, xs, ys, c='grey', ls = '--', lw=1.0, zorder=8, label='Pasos del SOR')

v.legend(ncol = 1, frameon=True, loc='best', bbox_to_anchor=(1.78, 1.01))
v.grid()
v.show()
### END SOLUTION

```

Cruce de rectas



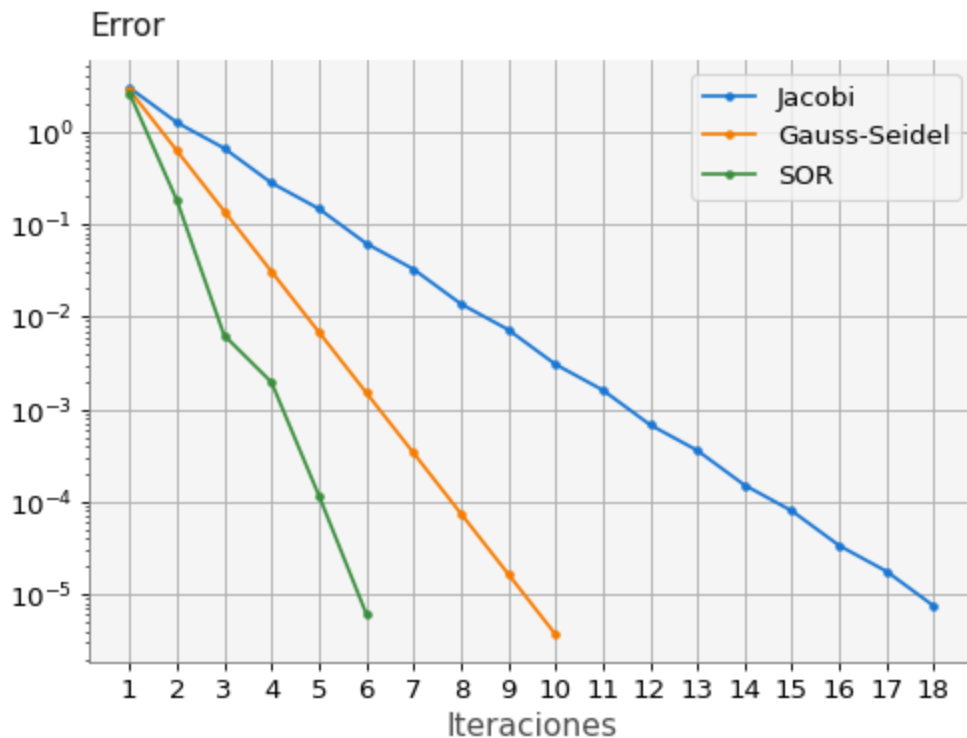

```
# Utiliza el código del caso anterior adaptado para que pueda graficar los tres e

#### BEGIN SOLUTION
# Lista con el número de las iteraciones máxima
it_max = max(itJ, itG, itSOR)+1
l_it_max = list(range(1,it_max))

# Listas con el número de las iteraciones para cada algoritmo
l_itJ = list(range(1,itJ+1))
l_itG = list(range(1,itG+1))
l_itSOR = list(range(1,itSOR+1))

# Parámetros para los ejes
a_p = dict(yscale='log', xlabel='Iteraciones', xticks = l_it_max)

# Gráficas del error
v = mvis.Plotter(1,1,[a_p])
v.axes(1).set_title('Error', loc='left')
v.plot(1, l_itJ, eJ, marker='.', label='Jacobi')
v.plot(1, l_itG, eG, marker='.', label='Gauss-Seidel')
v.plot(1, l_itSOR, eSOR, marker='.', label='SOR')
v.legend()
v.grid()
#### END SOLUTION
```



11.3 Ejercicio 4.

Almacena los errores de los tres métodos en los archivos: `errorJacobi.npy`, `errorGaussSeidel.npy` y `errorSOR.npy` usando la función `np.save()`, checa la documentación [aquí](#).

Prueba que tu código funciona usando:

```
print('Error Jacobi = \n{}\n'.format(np.load('errorJacobi.npy')))
print('Error Gauss-Seidel = \n{}\n'.format(np.load('errorGaussSeidel.npy')))
print('Error SOR = \n{}\n'.format(np.load('errorSOR.npy')))
```

La salida debería ser:

```
Error Jacobi =
[2.98142397e+00 1.25707872e+00 ...]
```

```
Error Gauss-Seidel =
[2.81091348e+00 6.24647439e-01 ...]
```

```
Error SOR =
[2.60865150e+00 1.82203110e-01 ...]
```

```
# np.save( ... )
#

#### BEGIN SOLUTION
np.save('errorJacobi.npy', eJ)
np.save('errorGaussSeidel.npy', eG)
np.save('errorSOR.npy', eSOR)
#### END SOLUTION
```

```
print('Error Jacobi = \n{}\n'.format(np.load('errorJacobi.npy')))
print('Error Gauss-Seidel = \n{}\n'.format(np.load('errorGaussSeidel.npy')))
print('Error SOR = \n{}\n'.format(np.load('errorSOR.npy')))
```

```
Error Jacobi =
[2.98142397e+00 1.25707872e+00 6.62538660e-01 2.79350827e-01
 1.47230813e-01 6.20779616e-02 3.27179585e-02 1.37951026e-02
 7.27065745e-03 3.06557835e-03 1.61570166e-03 6.81239633e-04
 3.59044812e-04 1.51386585e-04 7.97877361e-05 3.36414634e-05
 1.77306080e-05 7.47588075e-06]
```

```
Error Gauss-Seidel =
[2.81091348e+00 6.24647439e-01 1.38810542e-01 3.08467871e-02
 6.85484158e-03 1.52329813e-03 3.38510695e-04 7.52245990e-05
 1.67165775e-05 3.71479501e-06]
```

Error SOR =

[2.60865150e+00 1.82203110e-01 6.30966741e-03 1.96336589e-03
1.18187146e-04 6.25365681e-06]



8 Descenso del gradiente y Gradiente Conjugado.

Objetivo.

Describir e implementar los métodos de descenso del gradiente y de gradiente conjugado para la solución de sistemas de ecuaciones lineales.

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under

[Attribution-ShareAlike 4.0 International](#)

Trabajo realizado con el apoyo del Programa UNAM-DGAPA-PAPIME PE101922

```
import numpy as np
import ipywidgets as widgets
import macti.visual as mvis
import macti.matem as mmat
```

La siguiente función será usada para graficar algunos resultados.

```
def grafica(x, y1, y2, sol = [], xs = [], ys = [], vA = [], xg = [], yg = [], z =
    """
    Esta función grafica las líneas rectas, la solución, los pasos y los eigenvec
    """
    v = mvis.Plotter(1,1,[dict(aspect='equal')],title='Cruce de rectas')
    v.set_coordsys(1)

    # Graficamos las líneas rectas
    v.plot(1, x, y1, lw = 3, c = 'seagreen', label = '$3x+2y=2$') # Línea recta 1
    v.plot(1, x, y2, lw = 3, c = 'mediumorchid', label = '$2x+6y=-8$') # Línea re

    if len(sol):
        # Graficamos la solución
        v.scatter(1, sol[0], sol[1], fc='sandybrown', ec='k', s = 75, alpha=0.75,

    if len(xs) and len(ys):
        # Graficamos los pasos
        v.scatter(1, xs[0], ys[0], fc='yellow', ec='k', s = 75, alpha=0.75, zorde
        v.scatter(1, xs[1:], ys[1:], c='navy', s = 10, alpha=0.5, zorder=8)
        v.plot(1, xs, ys, c='grey', ls = '--', lw=1.0, zorder=8, label='Pasos del

    if len(vA):
        # Graficamos los eigenvectores
        v.quiver(1, [sol[0], sol[0]], [sol[1], sol[1]], vA[0], vA[1], scale=10, z

    if len(xg) and len(yg) and len(z):
        v.contour(1, xg, yg, z, levels = 25, cmap='twilight', linewidths=1.0, zor
```

```
v.legend(ncol = 1, frameon=True, loc='best', bbox_to_anchor=(1.90, 1.02))
v.grid()
v.show()
```

8.1 Ejemplo 1. Cruce de líneas rectas.

Las siguientes dos rectas se cruzan en algún punto.

$$\begin{aligned} 3x + 2y &= 2 \\ 2x + 6y &= -8 \end{aligned}$$

Las ecuaciones de las rectas se pueden escribir como:

$$\begin{aligned} \frac{3}{2}x + y &= 1 \\ \frac{2}{6}x + y &= -\frac{8}{6} \end{aligned} \implies \begin{aligned} y &= m_1x + b_1 \\ y &= m_2x + b_2 \end{aligned} \text{ donde } \begin{aligned} m_1 &= -\frac{3}{2} & b_1 &= 1 \\ m_2 &= -\frac{2}{6} & b_2 &= -\frac{8}{6} \end{aligned}$$

Las ecuaciones de las rectas se pueden escribir en forma de un sistema lineal:

$$\begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 2 \\ -8 \end{bmatrix} \quad (1)$$

Podemos calcular el cruce de las rectas resolviendo el sistema lineal:

```
# Dominio
x = np.linspace(-3,6,10)

# Línea recta 1
m1 = -3/2
b1 = 1
y1 = m1 * x + b1

# Línea recta 2
m2 = -2/6
b2 = -8/6
y2 = m2 * x + b2

# Definimos el sistema de ecuaciones lineales
A = np.array([[3, 2],[2,6]] )
b = np.array([2,-8])
print("Matriz A : \n",A)
print("Vector b : \n", b)

# Resolvemos el sistema
sol = np.linalg.solve(A,b)
print("Solución del sistema: ", sol)
```

```
# Usamos la función grafica() para mostrar las rectas y la solución
grafica(x, y1, y2, sol)
```

Matriz A :

```
[[3 2]
```

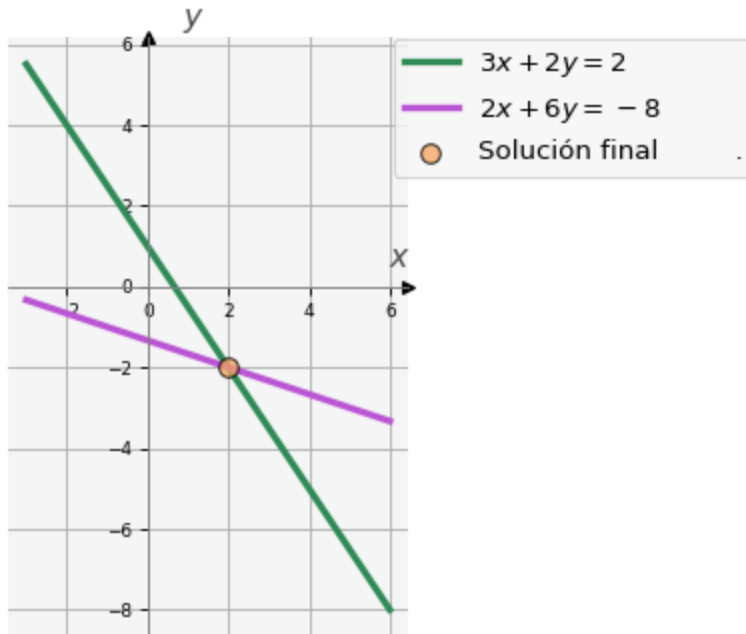
```
[2 6]]
```

Vector b :

```
[ 2 -8]
```

Solución del sistema: [2. -2.]

Cruce de rectas



En general, un sistema de ecuaciones de $n \times n$ se escribe como sigue:

$$\begin{array}{ccccccc}
 a_{11}x_1 & + & a_{12}x_2 & + \cdots + & a_{1n}x_n & = & b_1 \\
 a_{21}x_1 & + & a_{22}x_2 & + \cdots + & a_{2n}x_n & = & b_2 \\
 \vdots & & \vdots & & \vdots & & \vdots \\
 a_{i1}x_1 & + & a_{i2}x_2 & + \cdots + & a_{in}x_n & = & b_i \\
 \vdots & & \vdots & & \vdots & & \vdots \\
 a_{n1}x_1 & + & a_{n2}x_2 & + \cdots + & a_{nn}x_n & = & b_n
 \end{array}$$

Es posible usar métodos más eficientes que el de Jacobi, Gauss-Seidel y SOR para resolver este tipo de sistemas. A continuación veremos los métodos del descenso del gradiente y método de gradiente conjugado.

9 Métodos del subespacio de Krylov

Una excelente referencia para comenzar con estos métodos es la siguiente:

Shewchuk, J. R. (1994). [An Introduction to the Conjugate Gradient Method Without the Agonizing Pain](#). Carnegie-Mellon University. Department of Computer Science.

9.1 Cálculo de eigenvectores

Los eigenvalores y eigenvectores de una matriz son herramientas muy útiles para entender ciertos comportamientos. Una descripción la puedes ver en la notebook [05_Matrices_Normas_Eigen.ipynb](#). Los eigenvalores y eigenvectores se pueden calcular como sigue:

```
# Usando la función np.linalg.eig()
np.linalg.eig(A) # w: eigenvalues, v: eigenvectors
```

```
EigResult(eigenvalues=array([2., 7.]), eigenvectors=array([[ -0.89442719, -0.4472136 ],
[ 0.4472136 , -0.89442719]]))
```

La función `eigen_land()` de la biblioteca `macti` utiliza la función `np.linalg.eig()` para ofrecer una salida más entendible:

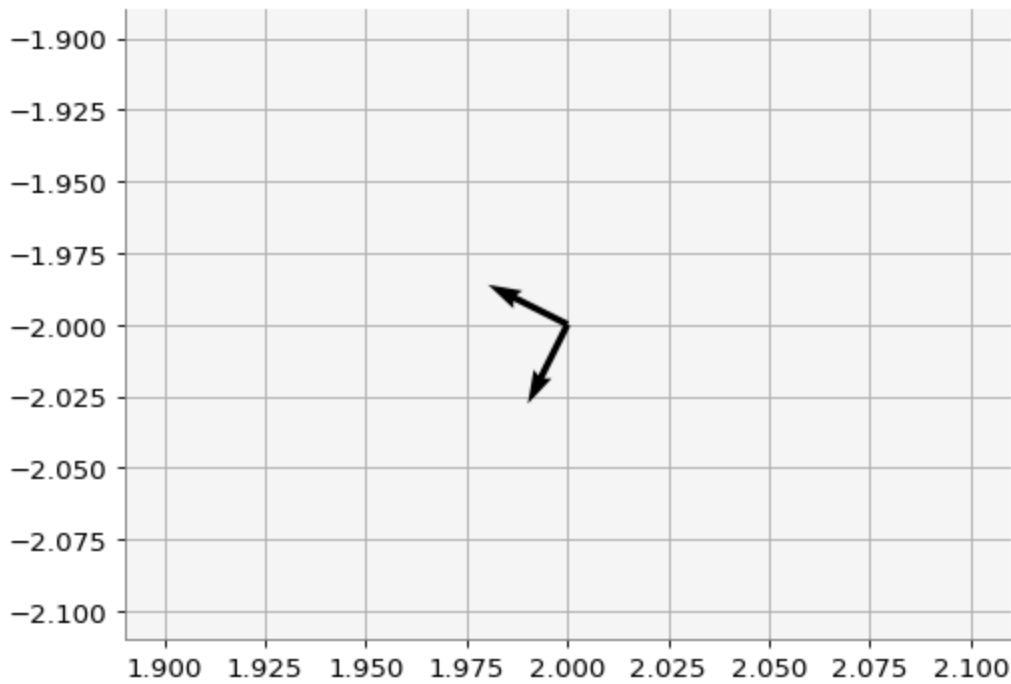
```
# Usando la función eigen_land() de macti
wA, vA = mmat.eigen_land(A)
```

```
eigenvalores = [2. 7.]
eigenvectores:
[ -0.89442719  0.4472136 ]
[ -0.4472136 -0.89442719]
ángulo entre eigenvectores = 90.0
```

Los eigenvectores se pueden visualizar, cuando la matriz es de 2×2 :

```
# Graficamos los eigenvectores
xv = np.array([[sol[0], sol[0]],
               [sol[1], sol[1]]])

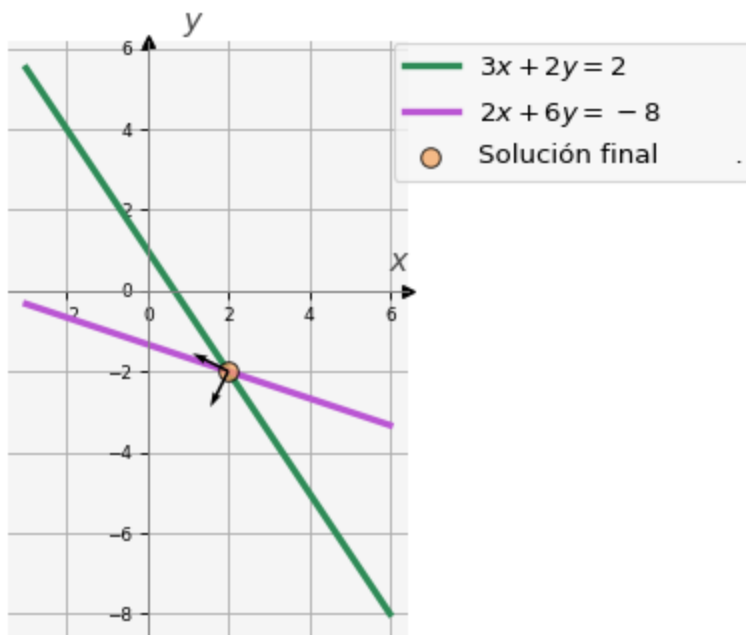
v = mvis.Plotter()
v.quiver(1, xv[0], xv[1], vA[0], vA[1], scale=10, zorder=6)
v.grid()
v.show()
```



Ahora usamos la función `grafica()` definida al principio de esta notebook para ver los eigenvectores y las líneas rectas:

```
# Usamos la función grafica() para ver los eigenvectores
grafica(x,y1,y2,sol,vA=vA)
```

Cruce de rectas



9.2 Forma cuadrática

La forma cuadrática de un sistema de ecuaciones lineales, permite transformar el problema $A\mathbf{x} = \mathbf{b}$ en un problema de minimización.

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A \mathbf{x} - \mathbf{x}^T \mathbf{b} + \mathbf{c}$$

$$A = \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 2 \\ -8 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

$$f'(\mathbf{x}) = \frac{1}{2}A^T \mathbf{x} + \frac{1}{2}A \mathbf{x} - \mathbf{b}$$

- Cuando A es simétrica: $f'(\mathbf{x}) = A \mathbf{x} - \mathbf{b}$
- Entonces un punto crítico de $f(\mathbf{x})$ se obtiene cuando $f'(\mathbf{x}) = 0$, es decir cuando $A\mathbf{x} = \mathbf{b}$

Calculemos la forma cuadrática para nuestro ejemplo:

```
# Función cuadrática
f = lambda A,b,c,x: 0.5 * x @ A @ x.T - x @ b.T + c

# Tamaño de la malla para graficar
size_grid = 30
xg, yg = np.meshgrid(np.linspace(-3,6,size_grid),
                     np.linspace(-8,6,size_grid))

# Arreglo para almacenar los valores de la función cuadrática
z = np.zeros((size_grid, size_grid))

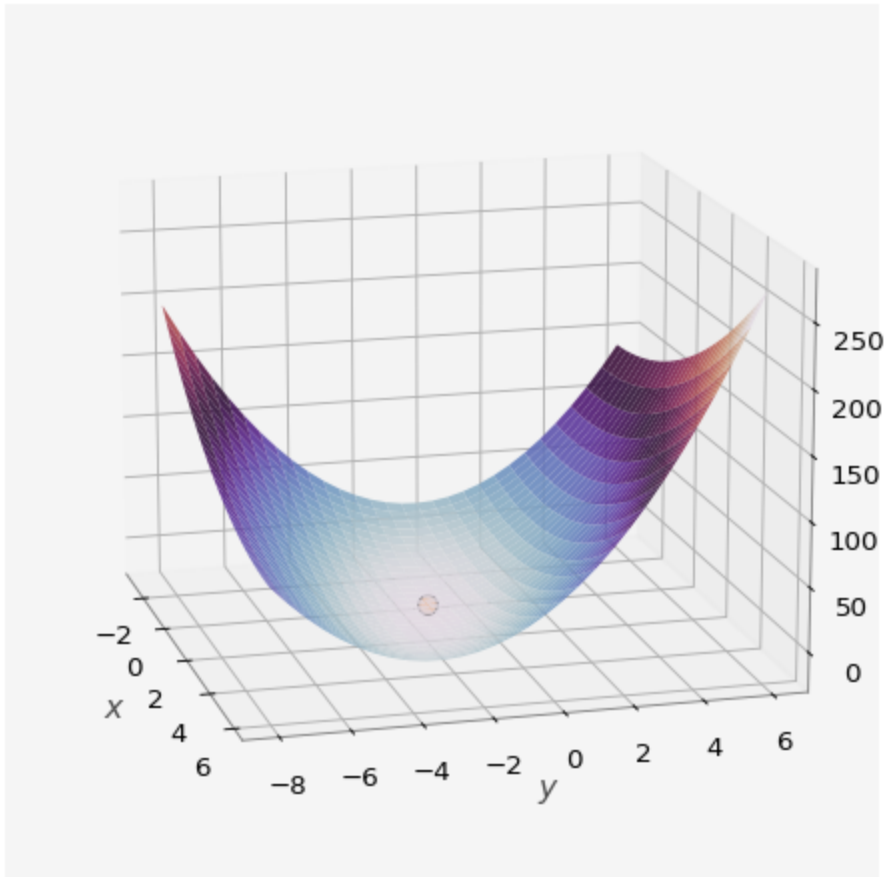
# Cálculo
for i in range(size_grid):
    for j in range(size_grid):
        xc = np.array([xg[i,j], yg[i,j]])
        z[i,j] = f(A,b,0,xc)
```

/tmp/ipykernel_217/3515168418.py:16: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)

```
z[i,j] = f(A,b,0,xc)
```

Graficamos la forma cuadrática, almacenada en z , y la solución. Esta última debe estar en el mínimo de $f(\mathbf{x})$.

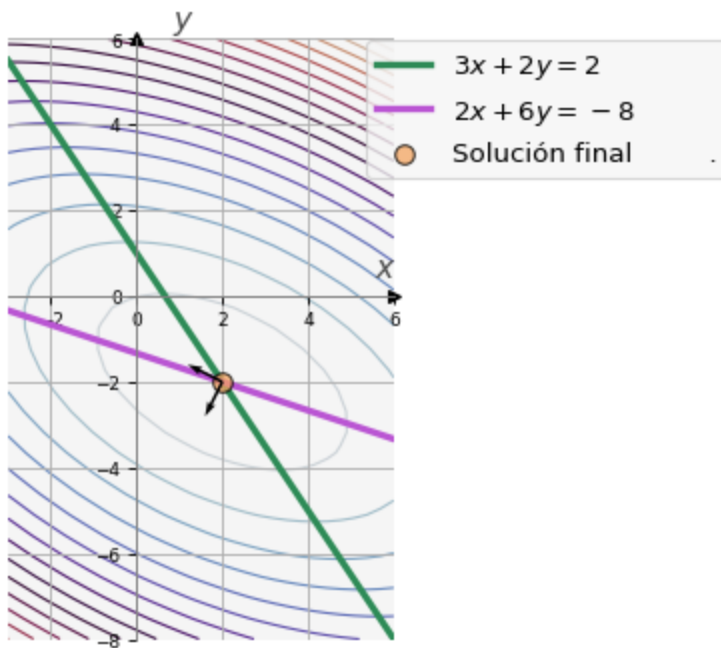
```
axis_par = [dict(projection='3d', aspect='auto', xlabel = '$x$', ylabel = '$y$',
v = mvis.Plotter(1,1, axis_par, dict(figsize=(8,6)))
v.plot_surface(1, xg, yg, z, cmap='twilight', alpha=0.90) # f(x)
v.scatter(1, sol[0], sol[1], fc='sandybrown', ec='k', s = 75, zorder=5, label='Sol')
v.axes(1).view_init(elev = 15, azimuth = -15)
```



Observamos un paraboloide cuyo mínimo es la solución del sistema. Esto es más claro si graficamos los contornos de $f(\mathbf{x})$:

```
grafica(x, y1, y2, sol, vA = vA, xg = xg, yg = yg, z = z)
```

Cruce de rectas



9.3 Algoritmo de descenso por el gradiente.

Este algoritmo utiliza la dirección del gradiente, en sentido negativo, para encontrar el mínimo y la solución del sistema:

\$

```

Input :  $\mathbf{x}_0, tol$ 
 $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$ 
 $k = 0$ 
WHILE( $\mathbf{r}_k > tol$ )
     $\mathbf{r}_k \leftarrow \mathbf{b} - A\mathbf{x}_k$ 
     $\alpha_k \leftarrow \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{r}_k^T A \mathbf{r}_k}$ 
     $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{r}_k$ 
     $k \leftarrow k + 1$ 
ENDWHILE

```

\$

9.4 Implementación.

```

def steepest(A,b,xi, yi,tol,kmax):
    # Solución inicial en forma de vector
    x = np.array([xi, yi])

    # Arreglos para almacenar los pasos.
    xs, ys = [xi], [yi]

    # Solución exacta
    xe = np.array([2, -2])

    r = b.T - A @ x
    res = np.linalg.norm(r, 2)
    res_list = []
    error = []
    k = 0
    while(res > tol and k < kmax):
        alpha = r.T @ r / (r.T @ A @ r)
        x = x + r * alpha
        xs.append(x[0])
        ys.append(x[1])
        r = b.T - A @ x

    # Resido
    res = np.linalg.norm(r, 2)

```

```

res_list.append(res)

# Error
e = np.linalg.norm(np.array([x[0], x[1]]) - xe, 2)
error.append(e)

k += 1
print('{:2d} {:10.9f} ({:10.9f}, {:10.9f})'.format(k, e, x[0], x[1]))
return x, np.array(xs), np.array(ys), error, res_list, k

```

9.5 Ejercicio 1.

Haciendo uso de la función `steepest()` definida en la celda anterior, aproxima la solución del sistema de ecuaciones del Ejemplo 1. Utiliza la solución inicial $(x_i, y_i) = (-2, 2)$, una tolerancia $tol = 1 \times 10^{-5}$ y $kmax = 50$ iteraciones. Utiliza las variables `solGrad`, `xs`, `ys`, `eGrad`, `rGrad` e `itGrad` para almacenar la salida de la función `steepest()`. Posteriormente grafica las rectas y cómo se va calculando la solución con este método. Utiliza la función `grafica()`. Grafica también el error y el residuo.

```

# Solución inicial (debe darse como un arreglo tipo columna)
# (xi, yi) = ...

# Método Steepest descend
# ...

#### BEGIN SOLUTION
# Solución inicial
(xi, yi) = (-2., 2.)
tol = 1e-5
kmax = 50

# Método Steepest descend
solGrad, xs, ys, eGrad, rGrad, itGrad = steepest(A, b, xi, yi, tol, kmax)

#file_answer.write('1', solGrad, 'solGrad es incorrecta: revisa la llamada y ejec
#file_answer.write('2', eGrad[-1], 'eGrad[-1] es incorrecto: revisa la llamada y
#file_answer.write('3', rGrad[-1], 'rGrad[-1] es incorrecto: revisa la llamada y
#file_answer.write('4', itGrad, 'itGrad es incorrecto: revisa la llamada y ejecu

#### END SOLUTION

```

```

1 3.261835423 (-1.180722892, -1.277108434)
2 1.717502736 (0.785542169, -0.785542169)
3 0.990340394 (1.034286544, -1.780519669)
4 0.521458662 (1.631273044, -1.631273044)
5 0.300681662 (1.706795433, -1.933362598)
6 0.158322389 (1.888049165, -1.888049165)

```

```

7 0.091291300 (1.910978854, -1.979767921)
8 0.048068966 (1.966010108, -1.966010108)
9 0.027717358 (1.972971893, -1.993857248)
10 0.014594433 (1.989680177, -1.989680177)
11 0.008415391 (1.991793876, -1.998134972)
12 0.004431081 (1.996866753, -1.996866753)
13 0.002555034 (1.997508502, -1.999433750)
14 0.001345340 (1.999048701, -1.999048701)
15 0.000775745 (1.999243545, -1.999828078)
16 0.000408465 (1.999711172, -1.999711172)
17 0.000235528 (1.999770329, -1.999947802)
18 0.000124016 (1.999912308, -1.999912308)
19 0.000071510 (1.999930269, -1.999984152)
20 0.000037653 (1.999973375, -1.999973375)
21 0.000021711 (1.999978829, -1.999995188)
22 0.000011432 (1.999991916, -1.999991916)
23 0.000006592 (1.999993572, -1.999998539)
24 0.000003471 (1.999997546, -1.999997546)
25 0.000002001 (1.999998048, -1.999999556)

```

```

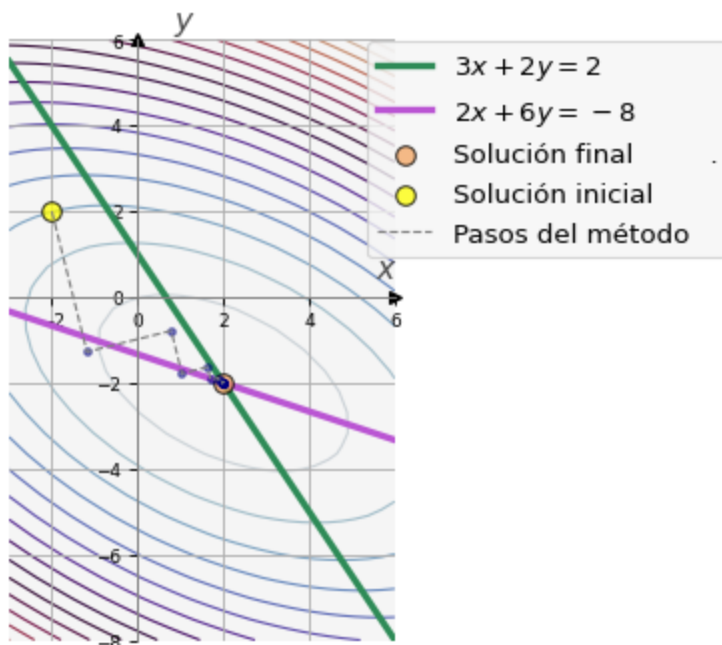
#quizz.eval_numeric('1', solGrad)
#quizz.eval_numeric('2', eGrad[-1])
#quizz.eval_numeric('3', rGrad[-1])
#quizz.eval_numeric('4', itGrad)

```

Gráfica de las rectas, la solución y los pasos realizados

```
grafica(x, y1, y2, sol, xs, ys, xg = xg, yg = yg, z = z)
```

Cruce de rectas

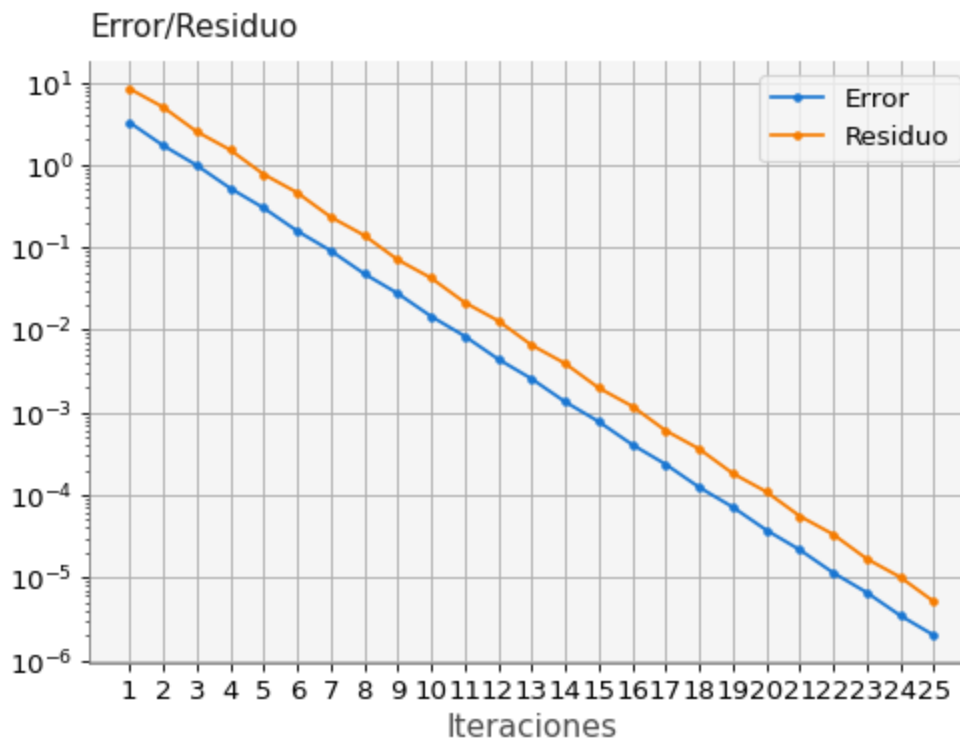


Grafica del error y el residuo.

```
# Lista con el número de las iteraciones
l_itGrad = list(range(1,itGrad+1))

# Parámetros para los ejes
a_p = dict(yscale='log', xlabel='Iteraciones', xticks = l_itGrad)

# Gráfica del error
v = mvis.Plotter(1,1,[a_p])
v.axes(1).set_title('Error/Residuo', loc='left')
v.plot(1, l_itGrad, eGrad, marker='.', label='Error')
v.plot(1, l_itGrad, rGrad, marker='.', label='Residuo')
v.legend()
v.grid()
```



9.6 Algoritmo de Gradiente Conjugado

Este algoritmo mejora al descenso del gradiente tomando direcciones conjugadas para evitar repetir un paso en una misma dirección.

\$

Input : $A, \mathbf{b}, \mathbf{x}_0, k_{max}, tol$
 $\mathbf{d}_0 = \mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
 $k = 0$
 While($\|\mathbf{r}\| > tol$ AND $k < k_{max}$)
 $\alpha_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{d}_k^T A \mathbf{d}_k}$
 $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$
 $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k A \mathbf{d}_k$
 $\beta_{k+1} = \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}$
 $\mathbf{d}_{k+1} = \mathbf{r}_{k+1} + \beta_{k+1} \mathbf{d}_k$
 $k = k + 1$
 End While

\$

9.6.1 Implementación.

```

def conjugateGradient(A,b,xi, yi, tol,kmax):
    # Solución inicial en forma de vector
    x = np.array([xi, yi])

    # Arreglos para almacenar los pasos.
    xs, ys = [xi], [yi]

    # Solución exacta
    xe = np.array([2, -2])

    r = b.T - A @ x
    d = r
    rk_norm = r.T @ r
    res = np.linalg.norm(rk_norm)
    res_list = []
    error = []

    k = 0
    while(res > tol and k < kmax):
        alpha = float(rk_norm) / float(d.T @ A @ d)
        x = x + alpha * d
        xs.append(x[0])
        ys.append(x[1])
        r = r - alpha * A @ d

        # Residuo
        res = np.linalg.norm(r, 2)
        res_list.append(res)
  
```

```

# Error
e = np.linalg.norm(np.array([x[0], x[1]]) - xe, 2)
error.append(e)

rk_old = rk_norm
rk_norm = r.T @ r
beta = float(rk_norm) / float(rk_old)
d = r + beta * d
k += 1
print('{:2d} {:10.9f} ({:10.9f}, {:10.9f})'.format(k, e, x[0], x[1]))
return x, np.array(xs), np.array(ys), error, res_list, k

```

9.7 Ejercicio 2.

Haciendo uso de la función `conjugateGradient()` definida en la celda anterior, aproxima la solución del sistema de ecuaciones del Ejemplo 1. Utiliza la solución inicial $(x_i, y_i) = (-2, 2)$, una tolerancia `tol = 1×10^{-5}` y `kmax = 50` iteraciones. Utiliza las variables `solCGM`, `xs`, `ys`, `eCGM`, `rCGM` e `itCGM` para almacenar la salida de la función `conjugateGradient()`. Posteriormente grafica las rectas y cómo se va calculando la solución con este método. Utiliza la función `grafica()`. Grafica también el error y el residuo.

```

# Solución inicial (debe darse como un arreglo tipo columna)
# (xi, yi) = ...

# Método CGM
# ...

### BEGIN SOLUTION
# Solución inicial
(xi, yi) = (-2., 2.)
tol = 1e-5
kmax = 50

# Método CGM
solCGM, xs, ys, eCGM, rCGM, itCGM = conjugateGradient(A, b, xi, yi, tol, kmax)

#file_answer.write('5', solCGM, 'solCGM es incorrecta: revisa la llamada y ejecuc
#file_answer.write('6', eCGM[-1], 'eCGM[-1] es incorrecto: revisa la llamada y ej
#file_answer.write('7', rCGM[-1], 'rCGM[-1] es incorrecto: revisa la llamada y ej
#file_answer.write('8', itCGM, 'itCGM es incorrecto: revisa la llamada y ejecució

### END SOLUTION

```

```

1 3.261835423 (-1.180722892, -1.277108434)
2 0.000000000 (2.000000000, -2.000000000)

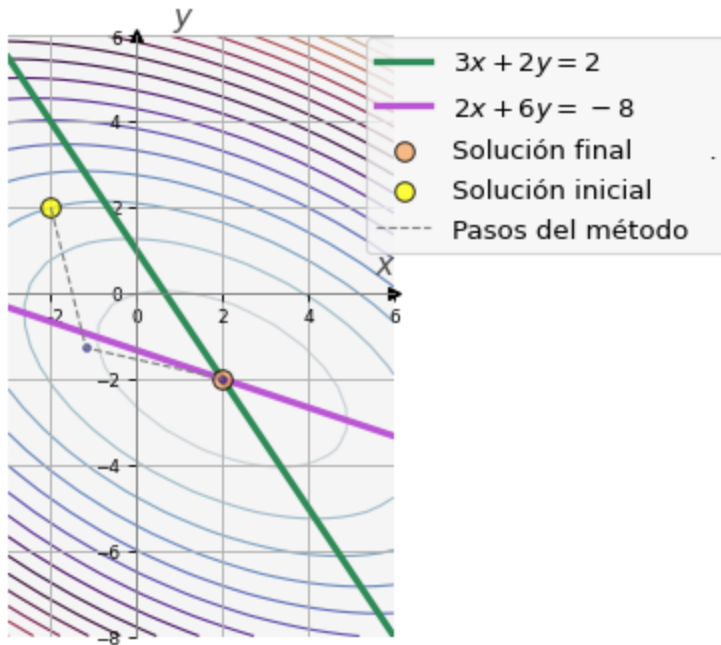
```



```
#quizz.eval_numeric('5', solCGM)
#quizz.eval_numeric('6', eCGM[-1])
#quizz.eval_numeric('7', rCGM[-1])
#quizz.eval_numeric('8', itCGM)
```

```
grafica(x, y1, y2, sol, xs, ys, xg = xg, yg = yg, z = z)
```

Cruce de rectas



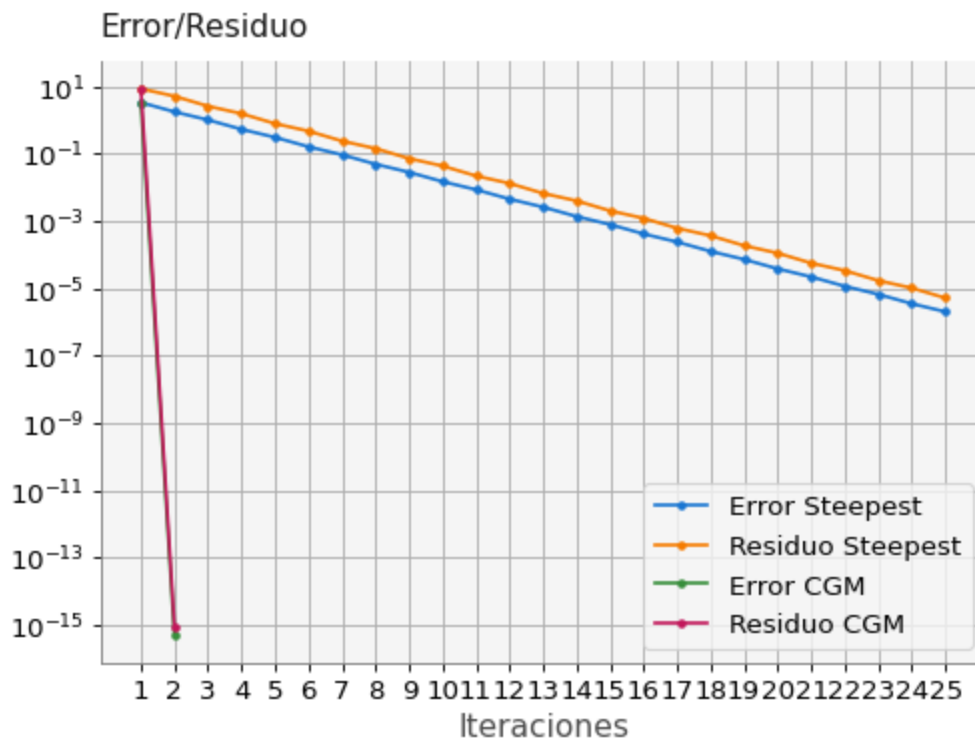
```
# Lista con el número de las iteraciones
l_itGrad = list(range(1, itGrad+1))
l_itCGM = list(range(1, itCGM+1))

# Parámetros para los ejes
a_p = dict(yscale='log', xlabel='Iteraciones', xticks = l_itGrad)

# Gráfica del error
v = mvis.Plotter(1,1,[a_p])
v.axes(1).set_title('Error/Residuo', loc='left')

v.plot(1, l_itGrad, eGrad, marker='.', label='Error Steepest')
v.plot(1, l_itGrad, rGrad, marker='.', label='Residuo Steepest')
v.plot(1, l_itCGM, eCGM, marker='.', label='Error CGM')
v.plot(1, l_itCGM, rCGM, marker='.', label='Residuo CGM')

v.legend()
v.grid()
```



9.8 Ejercicio 3.

Carga los archivos `errorJacobi.npy`, `errorGaussSeidel.npy` y `errorSOR.npy` en las variables `eJ`, `eG` y `eSOR` respectivamente (utiliza la función `np.load()`). Posteriormente grafica los errores de los 5 métodos: Jacobi, Gauss-Seidel, SOR, Steepest Descend, CGM. ¿Cuál de todos estos métodos usarías?

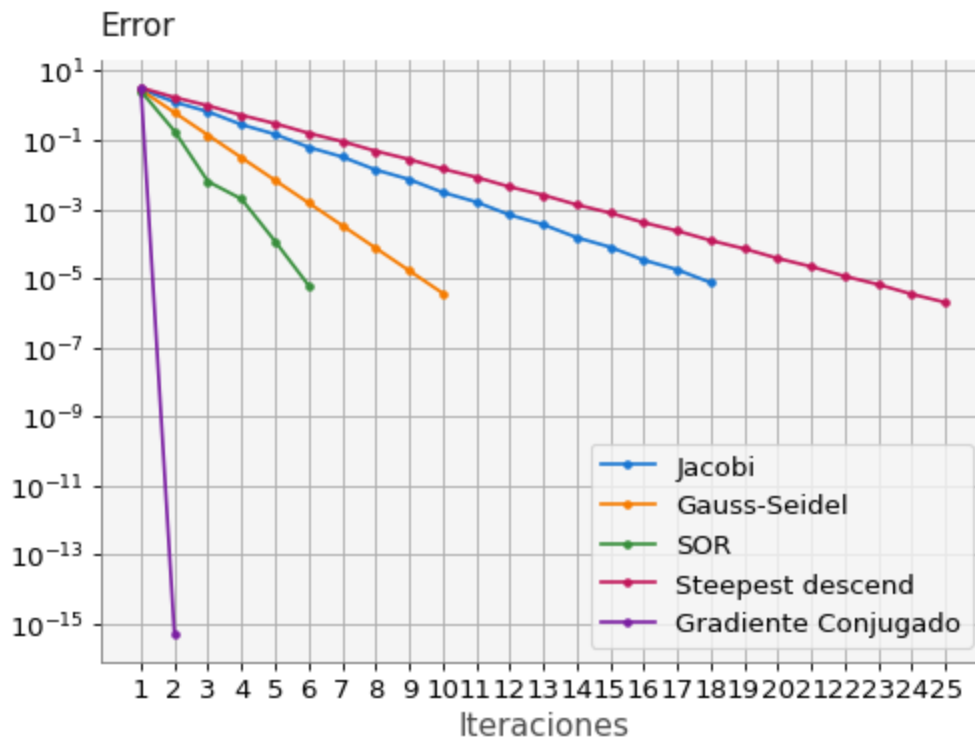
```
eJ = np.load('errorJacobi.npy')
eG = np.load('errorGaussSeidel.npy')
eSOR = np.load('errorSOR.npy')

# Lista con el número de las iteraciones
l_itJ = list(range(1, len(eJ)+1))
l_itG = list(range(1, len(eG)+1))
l_itSOR = list(range(1, len(eSOR)+1))
l_itGrad = list(range(1, itGrad+1))
l_itCGM = list(range(1, itCGM+1))

# Parámetros para los ejes
a_p = dict(yscale='log', xlabel='Iteraciones', xticks = l_itGrad)

# Gráfica del error
v = mvis.Plotter(1,1,[a_p])
v.axes(1).set_title('Error', loc='left')
v.plot(1, l_itJ, eJ, marker='.', label='Jacobi')
v.plot(1, l_itG, eG, marker='.', label='Gauss-Seidel')
```


```
v.plot(1, l_itSOR, eSOR, marker='.', label='SOR')  
v.plot(1, l_itGrad, eGrad, marker='.', label='Steepest descend')  
v.plot(1, l_itCGM, eCGM, marker='.', label='Gradiente Conjugado')  
  
v.legend()  
v.grid()
```



4 Normas vectoriales.

Objetivo.

Revisar e ilustrar los conceptos de normas vectoriales usando la biblioteca `numpy`.

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

Trabajo realizado con el apoyo del Programa UNAM-DGAPA-PAPIME PE101922

```
# Importamos las bibliotecas requeridas
import numpy as np
import ipywidgets as widgets
import macti.visual as mvis
```

5 Definición.

Una función $|| \cdot ||$ de vectores se denomina norma vectorial si para cualesquiera dos vectores \vec{x} y \vec{y} de \mathbb{R}^n se satisfacen los siguiente axiomas:

1. $||\vec{x}|| \geq 0$
2. $||\vec{x}|| = 0 \iff \vec{x} = 0$
3. $||a\vec{x}|| = |a| ||\vec{x}||$
4. $||\vec{x} + \vec{y}|| \leq ||\vec{x}|| + ||\vec{y}||$ (desigualdad triangular)

5.1 Tipos de normas.

Norma 1 $\rightarrow ||\vec{x}||_1 = \sum_{i=1}^n |x_i|$

Norma 2 (Euclideana) $\rightarrow ||\vec{x}||_2 = (\sum_{i=1}^n |x_i|^2)^{1/2} = \langle \vec{x}, \vec{x} \rangle^{1/2} = (\vec{x}^T \cdot \vec{x})^{1/2}$

Norma Infinito $\rightarrow ||\vec{x}||_\infty = \max_{i \leq 1 \leq n} |x_i|$

5.2 Ejemplo 1.

Para los vectores $\vec{x} = (2, 3, -4, 5)$ y $\vec{y} = (3.0, -1.45, 8.5, 2.1)$ en \mathbb{R}^4 probar que se cumplen las propiedades de la norma para los tres tipos de norma antes definidos.

Primero definimos los vectores

```
x = np.array([2, 3, -4, 5])
y = np.array([3.0, -1.45, 8.5, 2.1])

# Imprimimos los vectores
print('x = {}'.format(x))
print('y = {}'.format(y))
```

```
x = [ 2  3 -4  5]
y = [ 3.  -1.45  8.5  2.1 ]
```

Podemos calcular los diferentes tipos de norma para estos vectores usando la función `np.linalg.norm()`:

```
x_n1 = np.linalg.norm(x, 1)
y_n1 = np.linalg.norm(y, 1)
print('\nNorma 1 \n |x|1 = {} \n |y|1 = {}'.format(x_n1, y_n1))

x_n2 = np.linalg.norm(x, 2)
y_n2 = np.linalg.norm(y, 2)
print('\nNorma 2 \n |x|2 = {} \n |y|2 = {}'.format(x_n2, y_n2))

x_nI = np.linalg.norm(x, np.infty)
y_nI = np.linalg.norm(y, np.infty)
print('\nNorma infinito \n |x|∞ = {} \n |y|∞ = {}'.format(x_nI, y_nI))
```

Norma 1

```
|x|1 = 14.0
|y|1 = 15.049999999999999
```

Norma 2

```
|x|2 = 7.3484692283495345
|y|2 = 9.368164174479437
```

Norma infinito

```
|x|∞ = 5.0
|y|∞ = 8.5
```

Propiedad 1: Del resultado observamos que en todos los casos la norma es mayor que 0.

Propiedad 2: En ningún caso la norma es igual a 0, pues tanto \vec{x} como \vec{y} son diferentes de cero. La norma solo será igual a 0 si el vector es idénticamente 0:

```
z = np.zeros(4)
print('z = {}'.format(z))

z_n1 = np.linalg.norm(z, 1)
print('\nNorma 1 \n |z|1 = {}'.format(z_n1))

z_n2 = np.linalg.norm(z, 2)
print('\nNorma 2 \n |z|2 = {}'.format(z_n2))

z_nI = np.linalg.norm(z, np.infty)
print('\nNorma infinito \n |z|∞ = {}'.format(z_nI))
```

```
z = [0. 0. 0. 0.]
```

Norma 1

```
|z|1 = 0.0
```

Norma 2

```
|z|2 = 0.0
```

Norma infinito

```
|z|∞ = 0.0
```

Propiedad 3: definimos un escalar $a = -3.5$ entonces:

```

a = -3.5
print('a = {}, \t x = {}'.format(a, x))

a_x_n1 = np.linalg.norm(a * x, 1)
a_x_n2 = np.linalg.norm(a * x, 2)
a_x_nI = np.linalg.norm(a * x, np.infty)

print('\n ||a x||1 = {} \n |a| ||x||1 = {}'.format(a_x_n1, np.abs(a) * x_n1))
print('\n ||a x||2 = {} \n |a| ||x||2 = {}'.format(a_x_n2, np.abs(a) * x_n2))
print('\n ||a x||∞ = {} \n |a| ||x||∞ = {}'.format(a_x_nI, np.abs(a) * x_nI))

```

a = -3.5, x = [2 3 -4 5]

||a x||₁ = 49.0
|a| ||x||₁ = 49.0

||a x||₂ = 25.71964229922337
|a| ||x||₂ = 25.71964229922337

||a x||_∞ = 17.5
|a| ||x||_∞ = 17.5

Propiedad 4:

```

x_p_y_n1 = np.linalg.norm(x+y, 1)
print('\nNorma 1:')
print(' ||x + y||1 = {}'.format(x_p_y_n1))
print(' ||x||1 + ||y||1 = {}'.format(x_n1 + y_n1))
print(' ¿ ||x + y||1 ≤ ||x||1 + ||y||1 ? : {}'.format(x_p_y_n1 <= x_n1 + y_n1))

```

Norma 1:

||x + y||₁ = 18.15
||x||₁ + ||y||₁ = 29.049999999999997
¿ ||x + y||₁ ≤ ||x||₁ + ||y||₁ ? : True

5.3 Ejercicio 1.

Verifica que la propiedad 4 se cumple para las normas 2 e infinito para los vectores \vec{x} y \vec{y} antes definidos.

El resultado debería ser:

Norma 2:

||x + y||₂ = 9.90265116016918
||x||₂ + ||y||₂ = 16.716633402828972
¿ ||x + y||₂ ≤ ||x||₂ + ||y||₂ ? : True

Norma Infinito:

||x + y||_∞ = 7.1
||x||_∞ + ||y||_∞ = 13.5
¿ ||x + y||_∞ ≤ ||x||_∞ + ||y||_∞ ? : True

```

### BEGIN SOLUTION
x_p_y_n2 = np.linalg.norm(x+y, 2)
print('\nNorma 2:')
print(' ||x + y||2 = {}'.format(x_p_y_n2))
print(' ||x||2 + ||y||2 = {}'.format(x_n2 + y_n2))
print(' ¿ ||x + y||2 ≤ ||x||2 + ||y||2 ? : {}'.format(x_p_y_n2 <= x_n2 + y_n2))

x_p_y_nI = np.linalg.norm(x+y, np.infty)
print('\nNorma Infinito:')
print(' ||x + y||∞ = {}'.format(x_p_y_nI))
print(' ||x||∞ + ||y||∞ = {}'.format(x_nI + y_nI))
print(' ¿ ||x + y||∞ ≤ ||x||∞ + ||y||∞ ? : {}'.format(x_p_y_nI <= x_nI + y_nI))
### END SOLUTION

```

Norma 2:

```

||x + y||2 = 9.90265116016918
||x||2 + ||y||2 = 16.716633402828972
¿ ||x + y||2 ≤ ||x||2 + ||y||2 ? : True

```

Norma Infinito:

```

||x + y||∞ = 7.1
||x||∞ + ||y||∞ = 13.5
¿ ||x + y||∞ ≤ ||x||∞ + ||y||∞ ? : True

```

5.4 Desigualdad de Holder.

Para cualesquiera dos vectores \vec{x}, \vec{y} se cumple:

$$|\vec{x}^T \cdot \vec{y}| \leq \|\vec{x}\|_p \|\vec{y}\|_q, \text{ donde } p > 1, q > 1 \text{ y } \frac{1}{p} + \frac{1}{q} = 1$$

(Cuando $p = q = 2$ se obtiene la desigualdad de Schwarz)

5.5 Ejercicio 2.

Verifica que se cumple la desigualdad de Schwarz para los vectores \vec{x} y \vec{y} antes definidos.

El resultado debería ser:

Desigualdad de Holder

```

|<x, y>| = 21.85
||x||p * ||y||q = 68.84166616228866
¿ |<x, y>| ≤ ||x||p * ||y||q ? : True

```

```

### BEGIN SOLUTION
print('\nDesigualdad de Holder')
x_dot_y = np.abs(np.dot(x, y))

```

```
print(' |<x, y>| = {}'.format(x_dot_y))
print(' ||x|| * ||y|| = {}'.format(x_n2 * y_n2))
print(' ¿ |<x, y>| ≤ ||x|| * ||y|| ? : {}'.format(x_dot_y <= x_n2 * y_n2))
### END SOLUTION
```

Desigualdad de Holder

```
|<x, y>| = 21.85
||x|| * ||y|| = 68.84166616228866
¿ |<x, y>| ≤ ||x|| * ||y|| ? : True
```

5.6 Equivalencia de normas.

En un espacio \mathbb{R}^n de dimensión finita, cualquiera dos normas arbitrarias son equivalentes:

$$\begin{aligned} \|\vec{x}\|_2 &\leq \|\vec{x}\|_1 \leq \sqrt{n} \|\vec{x}\|_2 \\ \|\vec{x}\|_\infty &\leq \|\vec{x}\|_2 \leq \sqrt{n} \|\vec{x}\|_\infty \\ \|\vec{x}\|_\infty &\leq \|\vec{x}\|_1 \leq n \|\vec{x}\|_\infty \end{aligned}$$

```
print('\nEquivalencia entre norma 1 y norma 2\n')
print('||x||_2 = {}, ||x||_1 = {}, sqrt(4) * ||x||_2 = {}'.format(x_n2, x_n1, np.sqrt(4) * x_n2))
print('¿ ||x||_2 ≤ ||x||_1 ≤ sqrt(4) * ||x||_2 ? : {}'.format(x_n2 <= x_n1 <= np.sqrt(4) * x_n2))
```

Equivalencia entre norma 1 y norma 2

```
||x||_2 = 7.3484692283495345, ||x||_1 = 14.0, sqrt(4) * ||x||_2 = 14.696938456699069
¿ ||x||_2 ≤ ||x||_1 ≤ sqrt(4) * ||x||_2 ? : True
```

5.7 Ejercicio 3.

Verificar la equivalencia entre las normas $\|\cdot\|_\infty$ y $\|\cdot\|_2$ y entre $\|\cdot\|_\infty$ y $\|\cdot\|_1$ para los vectores \vec{x} y \vec{y} antes definidos.

El resultado debería ser:

Equivalencia entre norma infinito y norma 2

```
||x||_∞ = 5.0, ||x||_2 = 7.3484692283495345, sqrt(n) * ||x||_∞ = 10.0
¿ ||x||_∞ ≤ ||x||_2 ≤ sqrt(n) * ||x||_∞ ? : True
```

Equivalencia entre norma infinito y norma 1

```
||x||_∞ = 5.0, ||x||_1 = 14.0, n * ||x||_∞ = 20.0
¿ ||x||_∞ ≤ ||x||_1 ≤ n * ||x||_∞ ? : True
```

```
### BEGIN SOLUTION
n = 4
```



```

print('\nEquivalencia entre norma infinito y norma 2\n')
print('||x||∞ = {}, ||x||2 = {}, √n * ||x||∞ = {}'.format(x_nI, x_n2, np.sqrt(n) * x_nI))
print('¿ ||x||∞ ≤ ||x||2 ≤ √n * ||x||∞ ? : {}'.format(x_nI <= x_n2 <= np.sqrt(n) * x_nI))

print('\nEquivalencia entre norma infinito y norma 1\n')
print('||x||∞ = {}, ||x||1 = {}, n * ||x||∞ = {}'.format(x_nI, x_n1, n * x_nI))
print('¿ ||x||∞ ≤ ||x||1 ≤ n * ||x||∞ ? : {}'.format(x_nI <= x_n1 <= n * x_nI))
### END SOLUTION

```

Equivalencia entre norma infinito y norma 2

$\|x\|_{\infty} = 5.0$, $\|x\|_2 = 7.3484692283495345$, $\sqrt{n} * \|x\|_{\infty} = 10.0$

¿ $\|x\|_{\infty} \leq \|x\|_2 \leq \sqrt{n} * \|x\|_{\infty}$? : True

Equivalencia entre norma infinito y norma 1

$\|x\|_{\infty} = 5.0$, $\|x\|_1 = 14.0$, $n * \|x\|_{\infty} = 20.0$

¿ $\|x\|_{\infty} \leq \|x\|_1 \leq n * \|x\|_{\infty}$? : True



5 Matrices, normas y eigenvalores/eigenvectores.

Objetivo.

Revisar e ilustrar los conceptos de matrices, sus normas y eigenvalores/eigenvectores usando la biblioteca [numpy](#).

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#)

Trabajo realizado con el apoyo del Programa UNAM-DGAPA-PAPIME PE101922

```
# Importamos las bibliotecas requeridas
import numpy as np
import sympy
import ipywidgets as widgets
import macti.visual as mvis
import macti.matem as mmat
```

Sea $A = a_{ij}$ una matriz de $n \times n$, donde n indica la dimensión de la matriz (n renglones por n columnas). Los números a_{ij} son los elementos de la matriz, y $i, j = 1, \dots, n$. La matriz $A^T = a_{ji}$ es la matriz transpuesta.

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \quad A^T = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{pmatrix}$$

Definamos una matriz usando [numpy](#):

```
A = np.array([[2, 3, 5],
              [1, -4, 8],
              [8, 6, 3]])
A
```

```
array([[ 2,  3,  5],
       [ 1, -4,  8],
       [ 8,  6,  3]])
```

5.1 Matriz transpuesta

La matriz $A^T = a_{ji}$ es la matriz transpuesta.

```
AT = A.T
AT
```

```
array([[ 2,  1,  8],
       [ 3, -4,  6],
```

```
[ 5,  8,  3]])
```

5.2 Matriz identidad

La matriz identidad I es aquella donde todas sus entradas son cero excepto en la diagonal donde sus entradas son 1.

```
I = np.eye(3)
I
```

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

5.3 Matriz inversa

La matriz inversa de A se denota por A^{-1} y es tal que $A^{-1}A = I$.

```
Ainv = np.linalg.inv(A)
Ainv
```

```
array([[ -0.23715415,  0.08300395,  0.17391304],
       [ 0.24110672, -0.13438735, -0.04347826],
       [ 0.15019763,  0.04743083, -0.04347826]])
```

```
# Comprobar que Ainv es la inversa de A
np.dot(A, Ainv)
```

```
array([[1.00000000e+00, 0.00000000e+00, 2.77555756e-17],
       [2.22044605e-16, 1.00000000e+00, 0.00000000e+00],
       [1.11022302e-16, 2.77555756e-17, 1.00000000e+00]])
```

5.4 Matriz diagonal

Una matriz $A = a_{ij}$ se llama diagonal si $a_{ij} = 0, \forall i \neq j$ y se denota por $A = \text{diag } a_{ii}$.

```
A
```

```
array([[ 2,  3,  5],
       [ 1, -4,  8],
       [ 8,  6,  3]])
```

```
np.diagonal(A)
```

```
array([ 2, -4,  3])
```

```
# Diagonales inferiores
np.diagonal(A,1)
```

```
array([3, 8])
```

```
# Diagonales superiores
np.diagonal(A,-1)
```

```
array([1, 6])
```

5.5 Matriz triangular superior e inferior

Una matriz $A = a_{ij}$ se llama triangular superior si $a_{ij} = 0, \forall i > j$ y triangular inferior si $a_{ij} = 0, \forall i < j$.

```
# Matriz triangular superior
np.triu(A)
```

```
array([[ 2,  3,  5],
       [ 0, -4,  8],
       [ 0,  0,  3]])
```

```
# Matriz triangular inferior
np.tril(A)
```

```
array([[ 2,  0,  0],
       [ 1, -4,  0],
       [ 8,  6,  3]])
```

5.6 Matrices simétricas

Una matriz A es simétrica si $A^T = A$ y antisimétrica si $A^T = -A$.

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

```
print('Matriz A = \n{} \n\nMatriz B = \n{}'.format(A,B))
```

```
Matriz A =
[[ 2  3  5]
 [ 1 -4  8]
 [ 8  6  3]]
```

```
Matriz B =
[[ 2  3  5]
 [ 3 -4  8]
 [ 5  8  3]]
```

```
# Función para checar si una matriz es simétrica
isSymmetric = lambda mat: np.array_equal(mat, mat.T)
```

```
isSymmetric(B)
```

True

```
isSymmetric(A)
```

False

5.7 Matriz ortogonal

Una matriz A es ortogonal si $A^T A = I$, o equivalentemente $A^T = A^{-1}$.

La [matriz rotación](#) en 2D es una matriz ortogonal y se define como sigue:

$$R(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

```
theta = sympy.symbols('theta')

# Matriz rotación
R = sympy.Matrix([[sympy.cos(theta), -sympy.sin(theta)],
                  [sympy.sin(theta), sympy.cos(theta)]])

R
```

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Verifiquemos que cumple con las propiedades de una matriz ortogonal.

```
R.T
```

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$R * R.T$$

$$\begin{bmatrix} \sin^2(\theta) + \cos^2(\theta) & 0 \\ 0 & \sin^2(\theta) + \cos^2(\theta) \end{bmatrix}$$

$$\text{sympy.simplify}(R * R.T)$$

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

Esta matriz rota un vector por un cierto número de grados, veamos:

```

angulo = 90 # ángulo de rotación

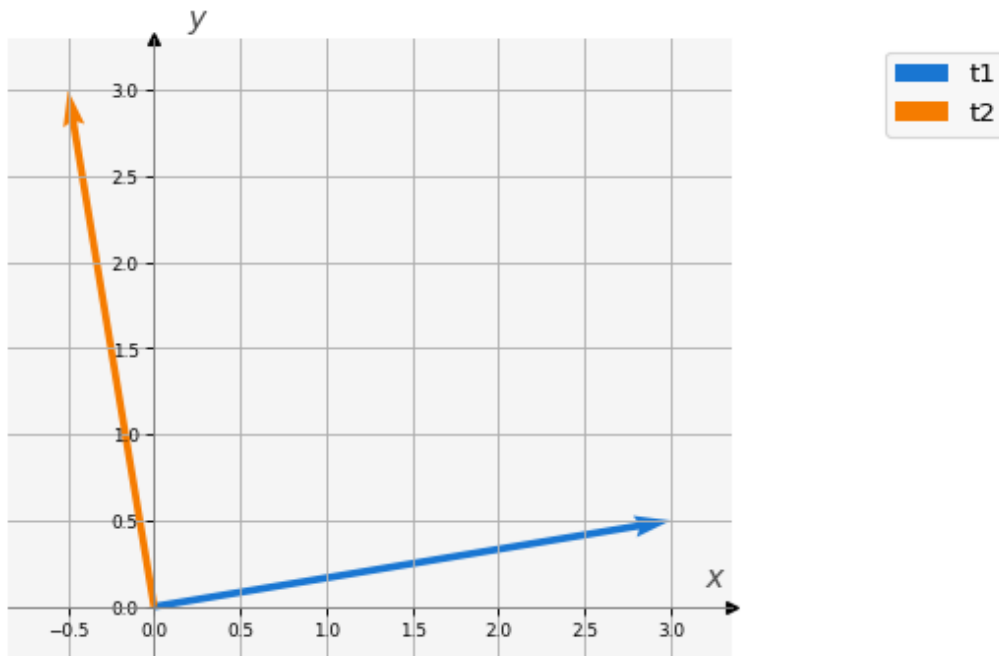
# Vector a rotar
t1 = sympy.Matrix([3, 0.5])

# Rotación usando la matriz R
t2 = R.subs('theta', angulo * np.pi / 180).evalf(14) * t1

# Transformación a arreglos de numpy
nt1 = np.array(t1, dtype=float).reshape(2,)
nt2 = np.array(t2, dtype=float).reshape(2,)

# Visualizamos los vectores.
v = mvis.Plotter() # Definición de un objeto para crear figuras.
v.set_coordsys(1) # Definición del sistema de coordenadas.
v.plot_vectors(1, [nt1, nt2], ['t1', 't2'], ofx=-0.1) # Graficación de los vectores.
v.grid() # Muestra la rejilla del sistema de coordenadas.

```



Cada par de renglones o de columnas de una matriz ortogonal, son ortogonales entre sí. Además la longitud de cada columna o renglón es igual a 1.

```
# Definimos una matriz ortogonal
C = np.array([[1/3, 2/3, -2/3],
              [-2/3, 2/3, 1/3],
              [2/3, 1/3, 2/3]])
```

```
# Verificamos que es ortogonal
np.dot(C, C.T)
```

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

```
# Verificamos ortogonalidad entre renglones
np.dot(C[0], C[1])
```

```
0.0
```

```
# Verificamos ortogonalidad entre columnas
np.dot(C[:,0], C[:,1])
```

```
0.0
```

```
# Verificamos la norma de los renglones
np.linalg.norm(C[2])
```

1.0

```
# Verificamos la norma de las columnas
np.linalg.norm(C[2])
```

1.0

5.8 Matriz transpuesta conjugada

La matriz A^* representa a la matriz A transpuesta y conjugada. La matriz $A^* = \bar{a}_{ji}$ se llama también la adjunta de A .

```
# Creación de una matriz con valores complejos
real = np.arange(1,10).reshape(3,3)
imag = np.arange(1,10).reshape(3,3)
C = real + imag *1.0j
C
```

```
array([[1.+1.j, 2.+2.j, 3.+3.j],
       [4.+4.j, 5.+5.j, 6.+6.j],
       [7.+7.j, 8.+8.j, 9.+9.j]])
```

```
# Transpuesta conjugada
C.conj().T
```

```
array([[1.-1.j, 4.-4.j, 7.-7.j],
       [2.-2.j, 5.-5.j, 8.-8.j],
       [3.-3.j, 6.-6.j, 9.-9.j]])
```

5.9 Matriz definida positiva

Una matriz A se denomina **positiva definida** si $\langle A\vec{x}, \vec{x} \rangle = \vec{x}^T A \vec{x} > 0$ para cualquier vector no nulo \vec{x} de \mathbb{R}^n .

La matriz se llama **positiva semidefinida** si $\vec{x}^T A \vec{x} \geq 0$ para cualquier vector \vec{x} de \mathbb{R}^n .

Recordemos que:

$$\vec{x}^T A \vec{x} = \sum_{i=1}^n \sum_{j=1}^n a_{ij} x_i x_j$$

5.10 Ejemplo 1.

Las siguientes dos rectas se cruzan en algún punto.

$$\begin{aligned} 3x + 2y &= 2 \\ 2x + 6y &= -8 \end{aligned}$$

En términos de un sistema lineal, las dos ecuaciones anteriores se escriben como sigue:

$$\begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ -8 \end{bmatrix} \quad (1)$$

Podemos calcular $\vec{x}^T A \vec{x}$ para este ejemplo como sigue:

```
# Usaremos sympy.
# Primero definimos los símbolos
x, y = sympy.symbols('x y')

# Construimos el vector de incógnitas
X = sympy.Matrix([x, y])
print(X)

# Construimos la matriz
A = sympy.Matrix([[3.0, 2.0], [2.0, 6.0]])
print(A)
```

```
Matrix([[x], [y]])
Matrix([[3.000000000000000, 2.000000000000000], [2.000000000000000, 6.000000000000000]])
```

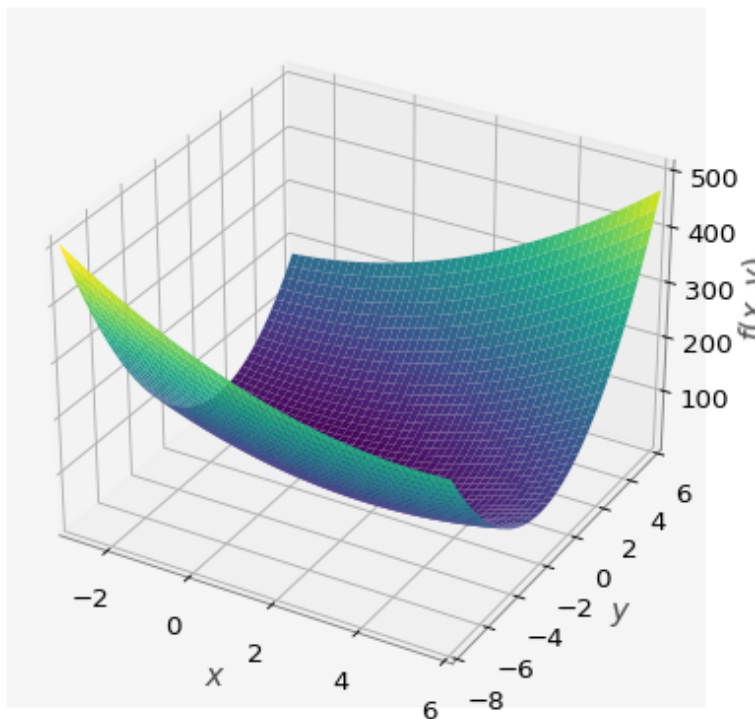
```
# Calculamos xT * A * x
pos_def = X.T @ A @ X
pos_def
```

```
[x (3.0x + 2.0y) + y (2.0x + 6.0y)]
```

```
# Simplificamos
f = sympy.simplify(pos_def)
f
```

```
[3.0x2 + 4.0xy + 6.0y2]
```

```
# Graficamos
sympy.plotting.plot3d(f[0], (x, -3, 6), (y, -8, 6))
```



Observa que se obtiene una función cuadrática cuya gráfica es un paraboloide orientado hacia arriba. Esta es una característica de las matrices definidas positivas.

5.11 Ejercicio 1.

Determinar si en el siguiente sistema de ecuaciones se tiene una matriz definida positiva:

$$\begin{aligned} y &= 0.10x + 200 \\ y &= 0.30x + 20 \end{aligned}$$

Sistema lineal.

$$\begin{bmatrix} 0.10 & -1 \\ 0.30 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -200 \\ -20 \end{bmatrix} \quad (2)$$

Guarda tu respuesta en la variable `respuesta = 'SI'` si la matriz es definida positiva o `respuesta = 'NO'` en caso contrario.

Hint: Utilizar el mismo código del ejemplo 1 y modificarlo de acuerdo al ejercicio planteado. Observa cómo sale la gráfica y responde la pregunta. Para un mejor resultado, utiliza valores muy grandes y muy chicos en los rangos de x y y al momento de graficar (> 2000).

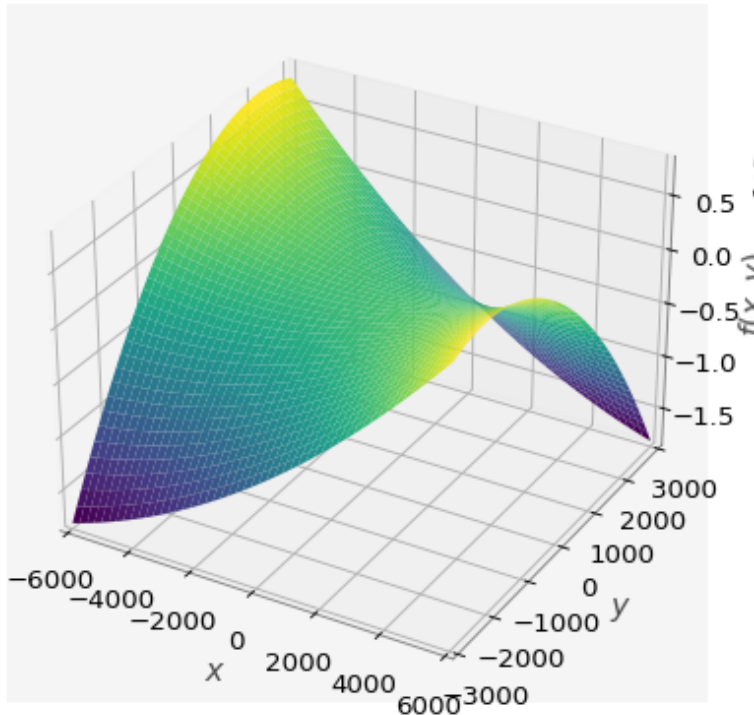
```
# B = sympy.Matrix(...)
# pos_def_B = ...
# fB = ...
```

```
# sympy.plotting.plot3D( ...)

### BEGIN SOLUTION
B = sympy.Matrix([[0.10, -1.0], [0.30, -1.0]])

pos_indef_B = X.T @ B @ X
fB = sympy.simplify(pos_indef_B)
sympy.plotting.plot3d(fB[0], (x, -6000, 6000), (y, -3000, 3000))
### END SOLUTION

# respuesta = ...
```



5.12 Eigenvalores y Eigenvectores

Si A es una matriz cuadrada, entonces definimos el número λ (real o complejo) como **autovalor** (**valor propio** o **eigenvalor**) de A si $A\vec{u} = \lambda\vec{u}$, o equivalentemente si $\det(A - \lambda I) = 0$. El vector \vec{u} se llama autovector (vector propio o eigenvector) de A . El conjunto de todos los autovalores de la matriz A se denomina espectro de A y se denota como $\rho(A)$.

```
# Convertimos la matriz A a un arreglo de numpy
A = np.array(A, dtype=float)
A
```

```
array([[3., 2.],
       [2., 6.]])
```

Los eigenvalores y eigenvectores se pueden calcular usando la función `np.linalg.eig()` de `numpy` como sigue:

```
np.linalg.eig(A) # w: eigenvalues, v: eigenvectors
```

```
EigResult(eigenvalues=array([2., 7.]), eigenvectors=array([[ -0.89442719, -0.4472136 ],
 [ 0.4472136 , -0.89442719]]))
```

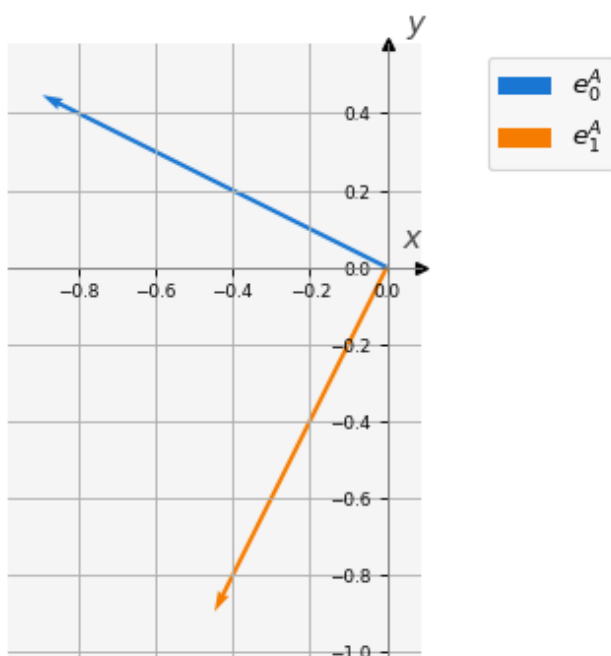
También podemos usar la función `macti.matem.eigen_land()` para obtener mayor información de los eigenvalores y eigenvectores como sigue:

```
wA, vA = mmat.eigen_land(A)
```

```
eigenvalores = [2. 7.]
eigenvectores:
[-0.89442719  0.4472136 ]
[-0.4472136 -0.89442719]
ángulo entre eigenvectores = 90.0
```

Podemos graficar los eigenvectores:

```
v = mvis.Plotter()
v.set_coordsys()
v.plot_vectors(1, [vA[:,0], vA[:,1]], ['$e_0^A$', '$e_1^A$'])
v.grid()
```



Observa que en este caso los eigenvectores son ortogonales.

La relación

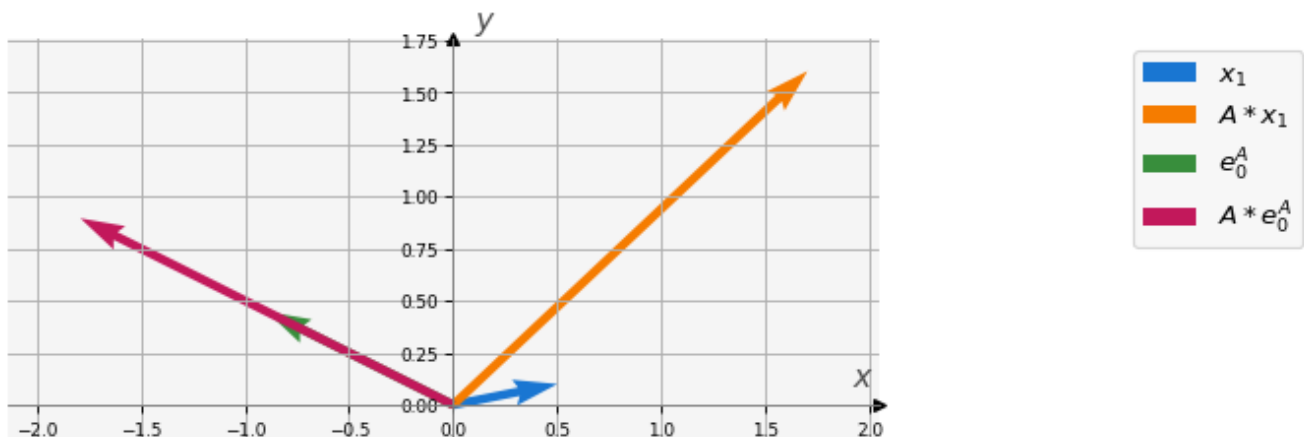
$$A\vec{u} = \lambda\vec{u}$$

indica básicamente que al aplicar la matriz A a un eigenvector \vec{u} , el resultado es el mismo vector escalado $\lambda\vec{u}$, es decir no lo rota. Cualquier otro vector, que no sea un múltiplo de los eigenvectores, será rotado. Veamos esto en el siguiente código:

```
# definimos un vector
x1 = np.array([0.5,0.1])

# Aplicamos la matriz A
r1 = A @ x1

# Ahora graficamos
v = mvis.Plotter()
v.set_coordsys()
v.plot_vectors(1, [x1, r1, vA[:,0], A @ vA[:,0]],
                ['$x_1$', '$A * x_1$', '$e_0^A$', '$A * e_0^A$'])
v.grid()
```



Observamos que el eigenvector e_0^A no rota cuando se le aplica A , pero el vector \vec{x}_1 si es rotado un cierto ángulo cuando le aplicamos la matriz A .

5.13 Normas Matriciales.

La norma de una matriz A es un número real positivo denotado por $\|A\|$. Dadas cualesquiera dos matrices A y B se cumplen los siguiente axiomas. 1. $\|A\| \geq 0$. 2. $\|A\| = 0 \iff A = 0$. 3. $\|aA\| = |a|\|A\|$ para cualquier número real a . 4. $\|A + B\| \leq \|A\| + \|B\|$ (desigualdad triangular). 5. $\|AB\| \leq \|A\|\|B\|$ (compatibilidad).

Definimos la siguiente matriz

$$M = \begin{bmatrix} -3 & 2 \\ 1 & -5 \end{bmatrix}$$

```
M = np.array([[-3, 2], [1, -5]])
M
```

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

5.13.1 Norma 1.

Consiste en sumar los valores absolutos de los elementos de cada **columna** y luego calcular la suma máxima:

$$\|A\|_1 = \max_{1 \leq j \leq n} \left(\sum_{i=1}^n |a_{ij}| \right)$$

```
np.linalg.norm(M, 1)
```

```
7.0
```

5.13.2 Norma ∞ .

Consiste en sumar los valores absolutos de los elementos de cada **renglón** y luego calcular la suma máxima:

$$\|A\|_\infty = \max_{1 \leq i \leq n} \left(\sum_{j=1}^n |a_{ij}| \right)$$

```
np.linalg.norm(M, np.infty)
```

```
6.0
```

5.13.3 Norma de Frobenius

$$\|A\|_F = \left(\sum_{i=1}^n \sum_{j=1}^n |a_{ij}|^2 \right)^{1/2}$$

```
np.linalg.norm(M, 'fro')
```

```
6.244997998398398
```

5.14 Ejemplo 2.

Verificar que se cumplen los 5 axiomas de las normas matriciales para la Norma 1 usando la matriz M .

Propiedad 1. $\|M\|_1 \geq 0$

```
M_n1 = np.linalg.norm(M,1 )
print('M =\n {}'.format(M))
print('||M||_1 = {}'.format(M_n1))
```

```
M =
[[-3  2]
 [ 1 -5]]
||M||_1 = 7.0
```

Propiedad 2.

```
ZERO = np.array([[0.0, 0.0], [0.0, 0.0]])
ZERO_n1 = np.linalg.norm(ZERO,1)
print('ZERO = \n{}'.format(ZERO))
print('||ZERO||_1 = {}'.format(ZERO_n1))
```

```
ZERO =
[[0. 0.]
 [0. 0.]]
||ZERO||_1 = 0.0
```

Propiedad 3.

```
a = -3.5
a_M_n1 = np.linalg.norm(a * M, 1)
print('||M||_1 = {}, \t a = {}'.format(M_n1, a))
print('\n ||a * M||_1 = {} \n |a| * ||M||_1 = {}'.format(a_M_n1, np.abs(a) * M_n1))
```

```
||M||_1 = 7.0,      a = -3.5
```

```
||a * M||_1 = 24.5
|a| * ||M||_1 = 24.5
```

Propiedad 4.

```
N = np.arange(4).reshape(2,2)

M_p_N_n1= np.linalg.norm(M + N, 1)

N_n1 = np.linalg.norm(N, 1)

print('\nNorma 1:')
print(' ||M + N||_1 = {}'.format(M_p_N_n1))
```

```
print(' ||M||_1 + ||N||_1 = {}'.format(M_n1 + N_n1))
print(' ¿ ||M + N||_1 ≤ ||M||_1 + ||N||_1 ? : {}'.format(M_p_N_n1 <= M_n1 + N_n1))
```

Norma 1:

```
||M + N||_1 = 6.0
||M||_1 + ||N||_1 = 11.0
¿ ||M + N||_1 ≤ ||M||_1 + ||N||_1 ? : True
```

Propiedad 5.

```
M_x_N_n1= np.linalg.norm(M * N, 1)

print('\nNorma 1:')
print(' ||M * N||_1 = {}'.format(M_x_N_n1))
print(' ||M||_1 * ||N||_1 = {}'.format(M_n1 * N_n1))
print(' ¿ ||M * N||_1 ≤ ||M||_1 * ||N||_1 ? : {}'.format(M_x_N_n1 <= M_n1 * N_n1))
```

Norma 1:

```
||M * N||_1 = 17.0
||M||_1 * ||N||_1 = 28.0
¿ ||M * N||_1 ≤ ||M||_1 * ||N||_1 ? : True
```

5.15 Ejercicio 2.

Verificar se cumplen los axiomas de las normas para $\|\cdot\|_F$ usando la matriz M .

Propiedad 1.

El resultado debería ser:

```
M =
[[-3  2]
 [ 1 -5]]
||M||_F = 6.244997998398398
```

```
### BEGIN SOLUTION
M_nF = np.linalg.norm(M, 'fro')
print('M =\n {}'.format(M))
print('||M||_F = {}'.format(M_nF))
### END SOLUTION
```

```
M =
[[-3  2]
```



```
[ 1 -5]]
```

```
||M||F = 6.244997998398398
```

Propiedad 2.

El resultado debería ser:

```
ZERO =
```

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

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

```
||ZERO||F = 0.0
```

```
#### BEGIN SOLUTION
ZERO_nF = np.linalg.norm(ZERO, 'fro')
print('ZERO = \n{}'.format(ZERO))
print('||ZERO||F = {}'.format(ZERO_nF))
#### END SOLUTION
```

```
ZERO =
```

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

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

```
||ZERO||F = 0.0
```

Propiedad 3.

El resultado debería ser:

```
||M||F = 6.244997998398398,    a = -3.5
```

```
||a * M||F = 21.857492994394395
```

```
|a| * ||M||F = 21.857492994394395
```

```
#### BEGIN SOLUTION
a = -3.5
a_M_nF = np.linalg.norm(a * M, 'fro')
print('||M||F = {}, \t a = {}'.format(M_nF, a))
print('\n ||a * M||F = {} \n |a| * ||M||F = {}'.format(a_M_nF, np.abs(a) * M_nF))
#### END SOLUTION
```

```
||M||F = 6.244997998398398,    a = -3.5
```

```
||a * M||F = 21.857492994394395
```

```
|a| * ||M||F = 21.857492994394395
```

Propiedad 4.

El resultado debería ser:

Norma de Frobenius:

```
||M + N||F = 5.5677643628300215
||M||F + ||N||F = 9.98665538517234
¿ ||M + N||F ≤ ||M||F + ||N||F ? : True
```

```
### BEGIN SOLUTION
N = np.arange(4).reshape(2,2)

M_p_N_nF= np.linalg.norm(M + N, 'fro')

N_nF = np.linalg.norm(N, 'fro')

print('\nNorma de Frobenius:')
print(' ||M + N||F = {}'.format(M_p_N_nF))
print(' ||M||F + ||N||F = {}'.format(M_nF + N_nF))
print(' ¿ ||M + N||F ≤ ||M||F + ||N||F ? : {}'.format(M_p_N_nF <= M_nF + N_nF))
### END SOLUTION
```

Norma de Frobenius:

```
||M + N||F = 5.5677643628300215
||M||F + ||N||F = 9.98665538517234
¿ ||M + N||F ≤ ||M||F + ||N||F ? : True
```

Propiedad 5.

El resultado debería ser:

Norma de Frobenius:

```
||M * N||F = 15.264337522473747
||M||F * ||N||F = 23.366642891095847
¿ ||M * N||F ≤ ||M||F * ||N||F ? : True
```

```
### BEGIN SOLUTION
M_x_N_nF= np.linalg.norm(M * N, 'fro')

print('\nNorma de Frobenius:')
print(' ||M * N||F = {}'.format(M_x_N_nF))
print(' ||M||F * ||N||F = {}'.format(M_nF * N_nF))
print(' ¿ ||M * N||F ≤ ||M||F * ||N||F ? : {}'.format(M_x_N_nF <= M_nF * N_nF))
### END SOLUTION
```

Norma de Frobenius:

```

||M * N||F = 15.264337522473747
||M||F * ||N||F = 23.366642891095847
¿ ||M * N||F ≤ ||M||F * ||N||F ? : True

```

5.15.1 Número de condición

El número de condición de una matriz A se define como

$$\kappa(A) = \|A\| \|A^{-1}\|$$

Este número siempre es más grande o igual a 1. Además nos da información acerca de que tan bien o mal está definido un problema que depende de la matriz en cuestión. Entre más grande sea este número es más difícil de resolver el problema.

```

A = np.array([[3., 2.],[2., 6.]])
print(A)
# Calculamos el número de condición usando funciones de numpy
kA_F = np.linalg.norm(A, 'fro') * np.linalg.norm(np.linalg.inv(A), 'fro')
print('κ(A) = {}'.format(kA_F))

```

```

[[3. 2.]
 [2. 6.]]
κ(A) = 3.7857142857142847

```

```

# Existe una función para calcular el número de condición directamente
kA_F = np.linalg.cond(A, 'fro')
print('κ(A) = {}'.format(kA_F))

```

```

κ(A) = 3.7857142857142847

```

```

# Matriz con un número de condición más grande
B = np.array([[0.10, -1],[0.30, -1]])
kB_F = np.linalg.cond(B, 'fro')
print(B)
print('κ(B) = {}'.format(kB_F))

```

```

[[ 0.1 -1. ]
 [ 0.3 -1. ]]
κ(B) = 10.5

```

```

# Matriz mal condicionada
C = np.array([[0.10, -1000],[0.30, -1]])
kC_F = np.linalg.cond(C, 'fro')
print(C)
print('κ(C) = {}'.format(kC_F))

```

```

[[ 1.e-01 -1.e+03]
 [ 3.e-01 -1.e+00]]
κ(C) = 3334.448482827609

```

5.16 Ejercicio 3.

Calcula el número de condición para las matrices A , B y C usando las normas 1 y 2. Utiliza la función `print()` de tal manera que obtengas una salida similar a la siguiente:

Número de condición con la norma 1:

$\kappa(A) = \dots$

$\kappa(B) = \dots$

$\kappa(C) = \dots$

Número de condición con la norma 2:

$\kappa(A) = \dots$

$\kappa(B) = \dots$

$\kappa(C) = \dots$

```
# Con la norma 1
# kA_1 = ...
# ...
# print('Número ...')
# print('κ(A) = {}, ...')

# Con la norma 2
# ...

#### BEGIN SOLUTION
# Usando la norma 1
kA_1 = np.linalg.cond(A, 1)
kB_1 = np.linalg.cond(B, 1)
kC_1 = np.linalg.cond(C, 1)
print('Número de condición con la norma 1:')
print(' κ(A) = {} \n κ(B) = {} \n κ(C) = {}'.format(kA_1, kB_1, kC_1))

# Usando la norma 2
kA_2 = np.linalg.cond(A, 2)
kB_2 = np.linalg.cond(B, 2)
kC_2 = np.linalg.cond(C, 2)
print('Número de condición con la norma 2:')
print(' κ(A) = {} \n κ(B) = {} \n κ(C) = {}'.format(kA_2, kB_2, kC_2))
#### END SOLUTION
```

Número de condición con la norma 1:

$\kappa(A) = 4.571428571428571$

$\kappa(B) = 13.0$

$\kappa(C) = 3338.113037679226$

Número de condición con la norma 2:

$\kappa(A) = 3.499999999999999$

$$\kappa(B) = 10.40388203202208$$

$$\kappa(C) = 3334.4481829279107$$



2 Producto escalar.

Objetivo. Revisar e ilustrar las propiedades del producto escalar en \mathbb{R}^n , para $n \geq 2$ usando la biblioteca `numpy`.

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under

[Attribution-ShareAlike 4.0 International](#)

Trabajo realizado con el apoyo del Programa UNAM-DGAPA-PAPIME PE101922

```
# Importamos las bibliotecas requeridas
import numpy as np
import macti.visual as mvis
```

2.1 Definición y propiedades.

Producto escalar es una operación algebraica que toma dos vectores y retorna un escalar. También se conoce como producto interno o producto punto. Su definición matemática es la siguiente:

$$\langle \vec{x}, \vec{y} \rangle = \vec{y}^T \cdot \vec{x} = \sum_{i=1}^n x_i y_i \quad (1)$$

Propiedades: 1. $\langle \vec{x}, \vec{y} \rangle = 0$ si y solo si \vec{x} y \vec{y} son ortogonales. 2. $\langle \vec{x}, \vec{x} \rangle \geq 0$, además $\langle \vec{x}, \vec{x} \rangle = 0$ si y solo si $\vec{x} = 0$ 3. $\langle \alpha \vec{x}, \vec{y} \rangle = \alpha \langle \vec{x}, \vec{y} \rangle$ 4. $\langle \vec{x} + \vec{y}, \vec{z} \rangle = \langle \vec{x}, \vec{z} \rangle + \langle \vec{y}, \vec{z} \rangle$ 5. $\langle \vec{x}, \vec{y} \rangle = \langle \vec{y}, \vec{x} \rangle$ 6. Desigualdad de Schwarz : $|\langle \vec{x}, \vec{y} \rangle| \leq \|\vec{x}\| \|\vec{y}\|$

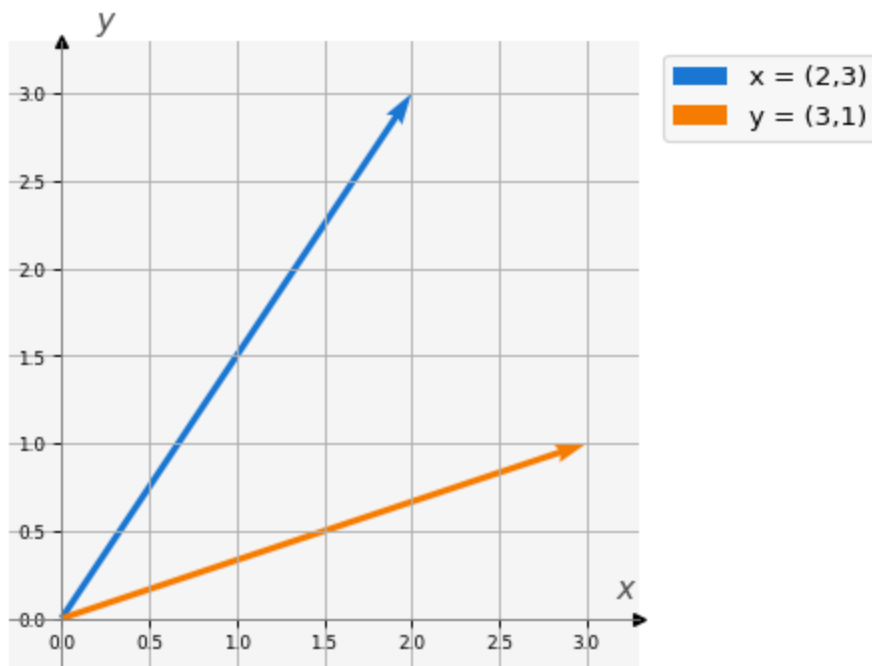
En lo que sigue realizaremos ejemplos en \mathbb{R}^2 de las propiedades antes descritas usando vectores (arreglos) contruidos con la biblioteca `numpy`.

```
# Definimos dos vectores en R^2 usando numpy
x = np.array([2, 3])
y = np.array([3, 1])

# Imprimimos los vectores
print('x = {}'.format(x))
print('y = {}'.format(y))
```

```
x = [2 3]
y = [3 1]
```

```
# Visualizamos los vectores.
v = mvis.Plotter() # Definición de un objeto para crear figuras.
v.set_coordsys(1) # Definición del sistema de coordenadas.
v.plot_vectors(1, [x, y], ['x = (2,3)', 'y = (3,1)'], ofx=-0.1) # Graficación de
v.grid() # Muestra la rejilla del sistema de coordenadas.
```



2.2 Implementación.

En Python es posible implementar el producto escalar de varias maneras, a continuación presentamos algunas de ellas.

2.2.1 Usando el ciclo `for`.

Es posible hacer una implementación del producto escalar usando ciclos `for`. De acuerdo con la definición $\langle \vec{x}, \vec{y} \rangle = \sum_{i=1}^n x_i y_i$ una implementación es como sigue:

```
dot_prod = 0.0

for i in range(len(x)):
    dot_prod += x[i] * y[i]

print('for cycle → <x, y> = {:.2f}'.format(dot_prod))
```

for cycle → <x, y> = 9.00

2.2.2 Usando la función `numpy.dot()`.

Esta función implementa un producto generalizado entre matrices cuyos elementos pueden ser flotantes o números complejos. Cuando se usa con arreglos de flotantes se obtiene el producto escalar. Usando esta función el ejemplo anterior se implementa cómo sigue:

```
dot_prod = np.dot(x,y)
print('np.dot → <x, y> = {:.2f}'.format(dot_prod))
```

`np.dot` → $\langle x, y \rangle = 9.00$

2.2.3 Usando la función `np.inner()`.

Esta función implementa el producto interno entre dos arreglos. Usando esta función el ejemplo anterior se implementa cómo sigue:

```
dot_prod = np.inner(x,y)
print('np.inner → <x, y> = {:.2f}'.format(dot_prod))
```

`np.inner` → $\langle x, y \rangle = 9.00$

2.2.4 Usando el operador `@`.

El operador `@`, disponible desde la versión Python 3.5, se puede usar para realizar la multiplicación de matrices convencional. Cuando se usa con arreglos de 1D se obtiene el producto escalar.

```
dot_prod = x @ y
print('Operador @ → <x, y> = {:.2f}'.format(dot_prod))
```

Operador `@` → $\langle x, y \rangle = 9.00$

Lo conveniente es usar el operador `@` o alguna de las funciones de biblioteca que ya están implementadas, como `dot()` o `inner()` y evitar la implementación usando el ciclo `for`. La razón es que la biblioteca [Linear algebra](#), cuando es posible utiliza la biblioteca [BLAS](#) optimizada.

En lo que sigue usaremos el operador `@` para calcular el producto escalar y probar las propiedades descritas al principio.

2.3 Propiedad 1: Ortogonalidad.

```
# Definimos otro vector en R^2
z = np.array([-3, 2])
```

```
# Calculamos el producto escalar entre los vectores x, y, z
print('<x, y> = {:.2f}'.format(x @ y))
print('<x, z> = {:.2f}'.format(x @ z))
print('<z, y> = {:.2f}'.format(z @ y))
```

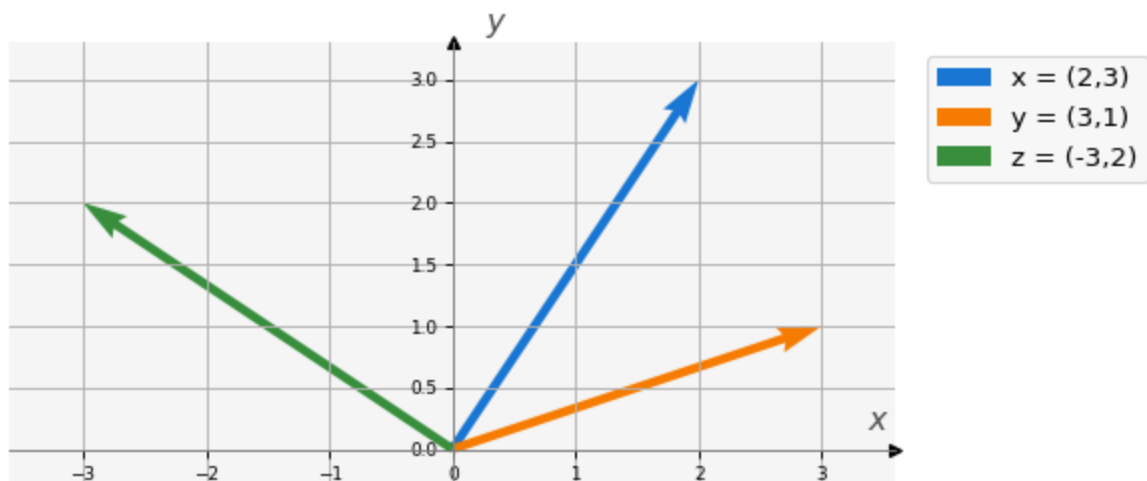
$\langle x, y \rangle = 9.00$

$\langle x, z \rangle = 0.00$

$\langle z, y \rangle = -7.00$

Como se puede observar, solo el producto $\langle \vec{x}, \vec{z} \rangle = 0$, lo cual significa que son ortogonales. Veamos los vectores gráficamente:


```
v = mvis.Plotter() # Definición de un objeto para crear figuras.
v.set_coordsys(1) # Definición del sistema de coordenadas.
v.plot_vectors(1, [x, y, z], ['x = (2,3)', 'y = (3,1)', 'z = (-3,2)'], ofx=-0.2) #
v.grid() # Muestra la rejilla del sistema de coordenadas.
```



La función `calc_angle(a, b)`, definida en la siguiente celda, calcula el ángulo entre los vectores \vec{a} y \vec{b} utilizando la siguiente fórmula

$$\cos(\alpha) = \frac{\langle \vec{a}, \vec{b} \rangle}{\|\vec{a}\| \|\vec{b}\|} \implies \alpha = \arccos \left(\frac{\langle \vec{a}, \vec{b} \rangle}{\|\vec{a}\| \|\vec{b}\|} \right)$$

Se usan las funciones `np.linalg.norm()` que calcula la norma de un vector, `np.arccos()` que es la función inversa del coseno y la constante `np.pi` que proporciona el valor de π .

```
def calc_angle(a, b):
    return np.arccos(a @ b / (np.linalg.norm(a) * np.linalg.norm(b))) * 180 / np.
```

```
# Calculamos el ángulo entre los vectores x, y, z
print('Ángulo entre x y y : {}'.format(calc_angle(x, y)))
print('Ángulo entre x y z : {}'.format(calc_angle(x, z)))
print('Ángulo entre z y y : {}'.format(calc_angle(z, y)))
```

Ángulo entre x y y : 37.8749836510982

Ángulo entre x y z : 90.0

Ángulo entre z y y : 127.8749836510982

Observamos que efectivamente el ángulo entre \vec{x} y \vec{z} es de 90° .

2.4 Propiedad 2. $\langle \vec{x}, \vec{x} \rangle \geq 0$

Verificamos que se cumple para los vectores \vec{x} , \vec{y} y \vec{z} :

```
print('<x, x> = {:.2f}'.format(x @ x))
print('<y, y> = {:.2f}'.format(y @ y))
print('<z, z> = {:.2f}'.format(z @ z))
```

<x, x> = 13.00

<y, y> = 10.00

<z, z> = 13.00

2.5 Propiedad 3. Multiplicación por un escalar.

$$\langle \alpha \vec{x}, \vec{y} \rangle = \alpha \langle \vec{x}, \vec{y} \rangle$$

```
# Definimos un escalar
alpha = 1.5

print('<alpha * x, y> = {}'.format((alpha * x) @ y))
print('alpha * <x, y> = {}'.format(alpha * x @ y))
print('¿ <alpha * x, y> == alpha * <x, y> ? : {}'.format(np.isclose((alpha * x) @ y, alpha * x @ y)))
```

<alpha * x, y> = 13.5

alpha * <x, y> = 13.5

¿ <alpha * x, y> == alpha * <x, y> ? : True

2.6 Propiedad 4. Asociatividad.

$$\langle \vec{x} + \vec{y}, \vec{z} \rangle = \langle \vec{x}, \vec{z} \rangle + \langle \vec{y}, \vec{z} \rangle$$

```
print('    <x + y, z> = {}'.format((x + y) @ z))
print('<x, z> + <y, z> = {}'.format(x @ z + y @ z))
print('¿ <x + y, z> == <x, z> + <y, z>? : {}'.format(np.isclose((x + y) @ z, x @ z + y @ z)))
```

<x + y, z> = -7

<x, z> + <y, z> = -7

¿ <x + y, z> == <x, z> + <y, z>? : True

2.7 Propiedad 5. Conmutatividad.

$\langle \vec{x}, \vec{y} \rangle = \langle \vec{y}, \vec{x} \rangle$

```
print('<x, y> = {}'.format(x @ y))
print('<y, x> = {}'.format(y @ x))
print('¿ <x, y> == <y, x> ? : {}'.format(np.isclose(x @ y, y @ x)))
```

```
<x, y> = 9
<y, x> = 9
¿ <x, y> == <y, x> ? : True
```

2.8 Propiedad 6. Desigualdad de Schwarz.

$$||\langle \vec{x}, \vec{y} \rangle|| \leq ||\vec{x}|| ||\vec{y}||$$

```
print('||<x, y>|| = {}'.format(np.abs(z @ y)))
print('||x|| ||y|| = {}'.format(np.linalg.norm(z) * np.linalg.norm(y)))
print('¿ ||<x, y>|| ≤ ||x|| ||y||? : {}'.format( np.abs(z @ y) <= np.linalg.norm(z)
```

```
||<x, y>|| = 7
||x|| ||y|| = 11.40175425099138
¿ ||<x, y>|| ≤ ||x|| ||y||? : True
```

2.9 Ejercicio 1.

Definimos los siguientes vectores $\vec{x} = (3.5, 0, -3.5, 0)$, $\vec{y} = (1.5, 1.0, 2.3, -1.0)$ y $\vec{z} = (1.0, 1.0, 1.0, 1.0)$ en \mathbb{R}^4 y $\alpha = 0.5$ un escalar. Verifica que se cumplen las propiedades 1 a 6.

Hint. Define los vectores \vec{x} , \vec{y} y \vec{z} usando **numpy** y posteriormente copia los códigos utilizados en el ejemplo de \mathbb{R}^2 para cada propiedad.

Observación. En este caso no es posible realizar gráficas.

Definición de los vectores.

Deberías obtener un resultado como el siguiente al imprimir los tres vectores:

```
x = [ 3.5  0.  -3.5  0. ]
y = [ 1.5  1.   2.3 -1. ]
z = [1.  1.  1.  1.]
```

```
### Definición de los vectores en R^4 con numpy
### BEGIN SOLUTION
x = np.array([3.5, 0, -3.5, 0])
y = np.array([1.5, 1.0, 2.3, -1.0])
z = np.array([1.0, 1.0, 1.0, 1.0])

print('x = {}'.format(x))
```

```
print('y = {}'.format(y))
print('z = {}'.format(z))
### END SOLUTION
```

```
x = [ 3.5  0.  -3.5  0. ]
y = [ 1.5  1.   2.3 -1. ]
z = [1.  1.  1.  1.]
```

Propiedad 1.

El resultado debería ser:

```
<x, y> = -2.80
<x, z> =  0.00
<z, y> =  3.80
```

```
# Calculamos el producto escalar entre los vectores x, y, z
### BEGIN SOLUTION
print('<x, y> = {:>5.2f}'.format(x @ y))
print('<x, z> = {:>5.2f}'.format(x @ z))
print('<z, y> = {:>5.2f}'.format(z @ y))
### END SOLUTION
```

```
<x, y> = -2.80
<x, z> =  0.00
<z, y> =  3.80
```

Propiedad 2.

El resultado debería ser:

```
<x, x> = 24.50
<y, y> =  9.54
<z, z> =  4.00
```

```
### BEGIN SOLUTION
print('<x, x> = {:>5.2f}'.format(x @ x))
print('<y, y> = {:>5.2f}'.format(y @ y))
print('<z, z> = {:>5.2f}'.format(z @ z))
### END SOLUTION
```

```
<x, x> = 24.50
<y, y> =  9.54
<z, z> =  4.00
```

Propiedad 3.

El resultado debería ser:

```
<α * x, y> = -1.3999999999999997
α * <x, y> = -1.3999999999999997
¿ <α * x, y> == α * <x,y> ? : True
```

```
# Definimos un escalar
#### BEGIN SOLUTION
α = 0.5

print('<α * x, y> = {}'.format((α * x) @ y))
print('α * <x, y> = {}'.format(α * x @ y))
print('¿ <α * x, y> == α * <x,y> ? : {}'.format(np.isclose((α * x) @ y, α * x @
#### END SOLUTION
```

```
<α * x, y> = -1.3999999999999995
α * <x, y> = -1.3999999999999995
¿ <α * x, y> == α * <x,y> ? : True
```

Propiedad 4.

El resultado debería ser:

```
<x + y, z> = 3.8
<x, z> + <y, z> = 3.8
¿ <x + y, z> == <x, z> + <y, z>? : True
```

```
#### BEGIN SOLUTION
print('      <x + y, z> = {}'.format((x + y) @ z))
print('<x, z> + <y, z> = {}'.format(x @ z + y @ z))
print('¿ <x + y, z> == <x, z> + <y, z>? : {}'.format(np.isclose((x + y) @ z, x @
#### END SOLUTION
```

```
<x + y, z> = 3.8
<x, z> + <y, z> = 3.8
¿ <x + y, z> == <x, z> + <y, z>? : True
```

Propiedad 5.

El resultado debería ser:

```
<x, y> = -2.7999999999999994
<y, x> = -2.7999999999999994
¿ <x, y> == <y, x> ? : True
```

```
### BEGIN SOLUTION
print('<x, y> = {}'.format(x @ y))
print('<y, x> = {}'.format(y @ x))
print('¿ <x, y> == <y, x> ? : {}'.format(np.isclose(x @ y, y @ x)))
### END SOLUTION
```

```
<x, y> = -2.7999999999999999
<y, x> = -2.7999999999999999
¿ <x, y> == <y, x> ? : True
```

Propiedad 6.

El resultado debería ser:

```
||<x, y>|| = 3.8
||x|| ||y|| = 6.1773780845922
¿ ||<x, y>|| ≤ ||x|| ||y|| ? : True
```

```
### BEGIN SOLUTION
print('||<x, y>|| = {}'.format(np.abs(z @ y)))
print('||x|| ||y|| = {}'.format(np.linalg.norm(z) * np.linalg.norm(y)))
print('¿ ||<x, y>|| ≤ ||x|| ||y|| ? : {}'.format(np.abs(z @ y) <= np.linalg.norm(z) * np.linalg.norm(y)))
### END SOLUTION
```

```
||<x, y>|| = 3.8
||x|| ||y|| = 6.1773780845922
¿ ||<x, y>|| ≤ ||x|| ||y|| ? : True
```

1 Espacio vectorial \mathbb{R}^2 .

Objetivo. Revisar e ilustrar las propiedades del espacio vectorial \mathbb{R}^2 usando la biblioteca `numpy`.

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under

[Attribution-ShareAlike 4.0 International](#) 

Trabajo realizado con el apoyo del Programa UNAM-DGAPA-PAPIME PE101922

1.1 Vectores en \mathbb{R}^2 .

Usando la biblioteca `numpy` podemos definir vectores en varias dimensiones. Para ejemplificar definiremos vectores en \mathbb{R}^2 y haremos algunas operaciones en este espacio vectorial.

```
# Importamos las bibliotecas requeridas
import numpy as np
import macti.visual as mvis
```

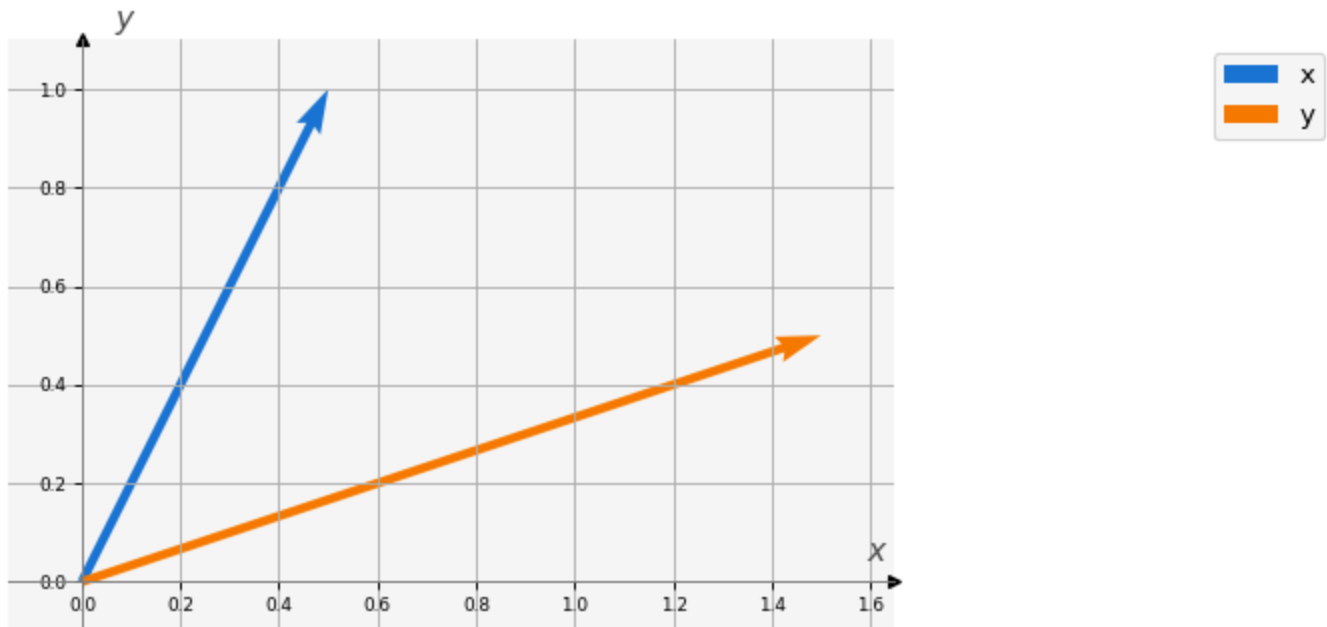
```
# Definimos dos vectores en  $\mathbb{R}^2$  (arreglos 1D de numpy de longitud 2).
x = np.array([0.5, 1.0])
y = np.array([1.5, 0.5])

print('x = {}'.format(x))
print('y = {}'.format(y))
```

```
x = [0.5 1. ]
y = [1.5 0.5]
```

La biblioteca `macti.visual` permite graficar los vectores en \mathbb{R}^2 como sigue:

```
v = mvis.Plotter() # Definición de un objeto para crear figuras.
v.set_coordsys(1) # Definición del sistema de coordenadas.
v.plot_vectors(1, [x, y], ['x', 'y']) # Graficación de los vectores 'x' y 'y'.
v.grid() # Muestra la rejilla del sistema de coordenadas.
```



1.2 Propiedades de un espacio vectorial.

1. $\vec{x} + \vec{y} \in \mathbb{R}^n$.
2. $\vec{x} + \vec{y} = \vec{y} + \vec{x}$.
3. $\vec{x} + (\vec{y} + \vec{z}) = (\vec{x} + \vec{y}) + \vec{z}$.
4. Existe el vector neutro $\vec{0} \in \mathbb{R}^n$ tal que $\vec{x} + \vec{0} = \vec{x}$.
5. Para cada $\vec{x} \in \mathbb{R}^n$ existe el opuesto $-\vec{x}$ tal que $\vec{x} + (-\vec{x}) = \vec{0}$.
6. $\alpha \vec{x} \in \mathbb{R}^n$.
7. $\alpha(\vec{x} + \vec{y}) = \alpha \vec{x} + \alpha \vec{y}$.
8. $(\alpha + \beta)\vec{x} = \alpha \vec{x} + \beta \vec{x}$.
9. $\alpha(\beta \vec{x}) = (\alpha\beta)\vec{x}$.
10. Existe el elemento neutro $1 \in \mathbb{R}$ tal que $1\vec{x} = \vec{x}$.

1.2.1 Propiedad 1: la suma es una operación interna.

$$\vec{x} + \vec{y} \in \mathbb{R}^n.$$

```
# Suma de dos vectores
z = x + y

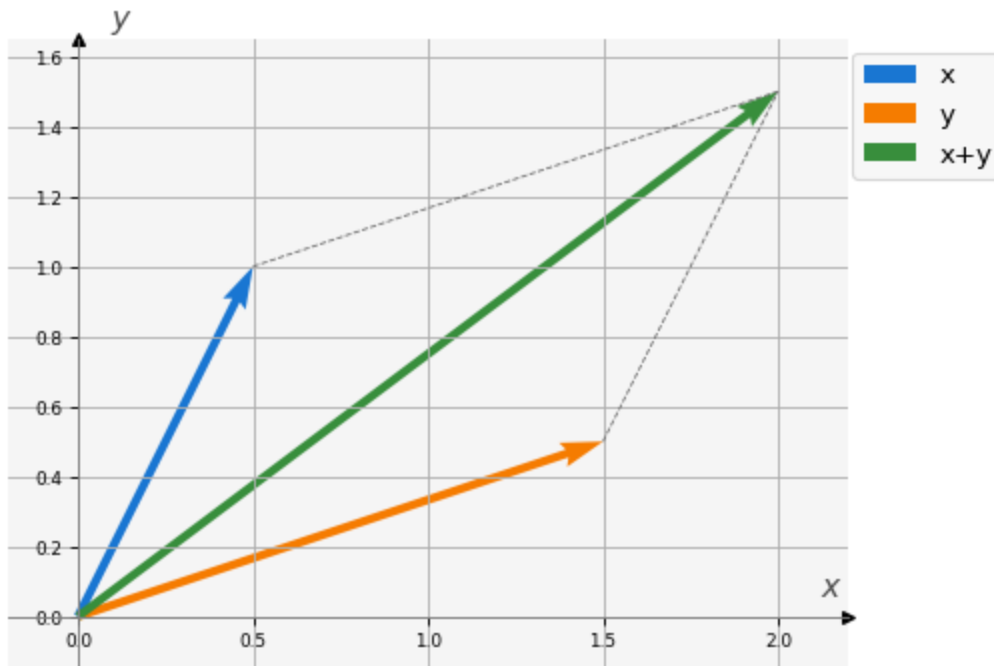
print('x = {}'.format(x))
print('y = {}'.format(y))
print('z = x + y = {}'.format(z))

# Graficamos los vectores y su suma
v = mvis.Plotter()
v.set_coordsys()
```



```
v.plot_vectors_sum(1, [x, y], ['x', 'y'], ofx=-0.3) # Grafica los vectores y el r
v.grid()
```

```
x = [0.5 1. ]
y = [1.5 0.5]
z = x + y = [2.  1.5]
```



Observa que la función `plot_vectors_sum()` muestra los vectores originales y la suma de ellos. El nuevo vector \vec{z} está en \mathbb{R}^2 .

1.2.2 Propiedad 2: la suma es conmutativa.

$$\vec{x} + \vec{y} = \vec{y} + \vec{x}.$$

```
xpy = x + y
ypx = y + x
print(' x + y = {} \t y + x = {}'.format(xpy, ypx))
print('\n ¿ x + y == y + x ? : {}'.format(np.isclose(xpy, ypx)))
```

```
x + y = [2.  1.5]   y + x = [2.  1.5]
```

```
¿ x + y == y + x ? : [ True  True]
```

Observa que la operación suma `+` se realiza componente a componente.

La función `np.isclose()` compara cada componente de los arreglos y determina que tan “parecidas” son hasta una tolerancia absoluta de 10^{-8} . Esta forma de comparación es la más conveniente cuando se comparan números reales (punto flotante, *floating point*).

1.2.3 Propiedad 3: la suma es asociativa.

$$\vec{x} + (\vec{y} + \vec{z}) = (\vec{x} + \vec{y}) + \vec{z}.$$

```
print(' x = {} \t y = {} \t z = {}'.format(x, y, z))
print(' x + (y + z)= {} \t (x + y) + z = {}'.format(x + (y + z), (x + y) + z))
print('\n ¿ x + (y + z) == (x + y) + z : {}'.format(np.isclose(x + (y + z), (x +
```

```
x = [0.5 1. ]   y = [1.5 0.5]   z = [2.  1.5]
x + (y + z)= [4. 3.]   (x + y) + z = [4. 3.]
```

```
¿ x + (y + z) == (x + y) + z : [ True  True]
```

1.2.4 Propiedad 4: elemento neutro de la suma.

Existe el vector neutro $\vec{0} \in \mathbb{R}^n$ tal que $\vec{x} + \vec{0} = \vec{x}$.

```
# Definimos el vector neutro.
cero = np.zeros(2)

print(' x = {} \t cero = {}'.format(x, cero))
print(' x + cero = {}'.format(x + cero))
```

```
x = [0.5 1. ]   cero = [0. 0.]
x + cero = [0.5 1. ]
```

1.2.5 Propiedad 5: elemento inverso en la suma.

Para cada $\vec{x} \in \mathbb{R}^n$ existe el inverso $-\vec{x}$ tal que $\vec{x} + (-\vec{x}) = \vec{0}$.

```
print(' x = {} \t -x = {}'.format(x, -x))
print(' x + (-x) = {}'.format(x + (-x)))
```

```
x = [0.5 1. ]   -x = [-0.5 -1. ]
x + (-x) = [0. 0.]
```

1.2.6 Propiedad 6: la multiplicación de un vector por un escalar produce un vector.

$$\alpha \vec{x} \in \mathbb{R}^n.$$

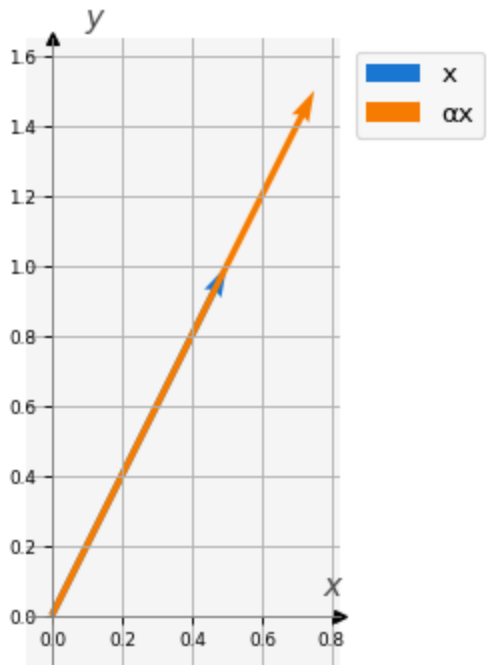
```
# Definimos un escalar
α = 1.5

# Realizamos la multiplicación de x por el escalar
αx = α * x

# Mostramos el resultado
print(' α = {} \t x = {} \n α * x = {} \n '.format(α, x, αx))

# Graficamos el vector original y el resultado.
v = mvis.Plotter()
v.set_coordsys()
v.plot_vectors(1, [x, αx], ['x', 'αx'], w=0.020)
v.grid()
```

```
α = 1.5      x = [0.5 1. ]
α * x = [0.75 1.5 ]
```



Observa que el vector original $\vec{x} = (0.5, 1.0)$, representado por la flecha azul, es más pequeño que el vector resultante $\alpha\vec{x} = (0.75, 1.5)$, representado por la flecha naranja. α multiplica a cada componente del vector \vec{x} .

Cuando multiplicamos un vector por un escalar, puede ocurrir que su longitud se agrande, se reduzca y/o que cambie de sentido.

Intenta modificar el valor de α a 0.5 y luego a -0.5 , observa que sucede.

1.2.7 Propiedad 7: distributividad I.

$$\alpha(\vec{x} + \vec{y}) = \alpha\vec{x} + \alpha\vec{y}$$

```
print(' α = {}'.format(α))
print(' x = {} \t y = {}'.format(x, y))
print(' α * (x + y) = {}'.format(α * (x + y)))
print(' α * x + α * y = {}'.format(α * x + α * y))
print(' ¿ α * (x + y) == α * x + α * y ? : {}'.format(np.isclose(α * (x + y), α * x + α * y)))
```

```
α = 1.5
x = [0.5 1. ]   y = [1.5 0.5]
α * (x + y) = [3.   2.25]
α * x + α * y = [3.   2.25]
¿ α * (x + y) == α * x + α * y ? : [ True  True]
```

1.2.8 Propiedad 8: distributividad II.

$$(\alpha + \beta)\vec{x} = \alpha\vec{x} + \beta\vec{x}$$

```
# Definimos dos escalares
α = 2.0
β = 1.5

print(' α = {} \t β = {} \t α + β = {}'.format(α, β, α + β))
print(' x = {}'.format(x))
print(' (α + β) * x = {}'.format((α + β) * x))
print(' α * x + β * x = {}'.format(α * x + β * x))
print(' ¿ (α + β) * x == α * x + β * x ? : {}'.format(np.isclose((α + β) * x, α * x + β * x)))
```

```
α = 2.0    β = 1.5    α + β = 3.5
x = [0.5 1. ]
(α + β) * x = [1.75 3.5 ]
α * x + β * x = [1.75 3.5 ]
¿ (α + β) * x == α * x + β * x ? : [ True  True]
```

1.2.9 Propiedad 9. asociatividad.

$$\alpha(\beta\vec{x}) = (\alpha\beta)\vec{x}.$$

```
# Definimos dos escalares
α = 2.0
β = 1.5

print(' α = {} \t β = {} \t x = {}'.format(α, β, x))
print(' α * (β * x) = {}'.format(α * (β * x)))
print(' (α * β) * x = {}'.format((α * β) * x))
print(' ¿ α * (β * x) == (α * β) * x ? : {}'.format(np.isclose(α * (β * x), (α * β) * x)))
```

```
α = 2.0    β = 1.5    x = [0.5 1. ]
α * (β * x) = [1.5 3. ]
(α * β) * x = [1.5 3. ]
¿ α * (β * x) == (α * β) * x ? : [ True  True]
```

1.2.10 Propiedad 10.

Existe el elemento neutro $\mathbf{1} \in \mathbb{R}$ tal que $\mathbf{1}\vec{x} = \vec{x}$.

```
 $\alpha$  = 1.0
```

```
print('  $\alpha$  = {} \t x = {}'.format( $\alpha$ , x))
print('  $\alpha * x$  = {}'.format( $\alpha * x$ ))
print(' ¿  $\alpha * x == x$  ? : {}'.format(np.isclose( $\alpha * x$ , x)))
```

```
 $\alpha$  = 1.0      x = [0.5 1. ]
 $\alpha * x$  = [0.5 1. ]
¿  $\alpha * x == x$  ? : [ True  True]
```

1.3 Ejercicio 1.

Definimos los siguientes vectores $\vec{x} = (1.2, 3.4, 5.2, -6.7)$ y $\vec{y} = (4.4, -2.3, 5.3, 8.9)$ ambos en \mathbb{R}^4 .

Usando $\alpha = 0.5$ y $\beta = 3.5$, verifica que se cumplen las propiedades 1 a 10 para \vec{x} y \vec{y} .

Hint. Define los vectores \vec{x} y \vec{y} usando `numpy` y posteriormente copia los códigos utilizados en el ejemplo de \mathbb{R}^2 para cada propiedad. En algunos casos debes ajustar el código para este ejercicio.

Observación. En este caso no es posible realizar gráficas.

```
### Definición de los vectores en  $\mathbb{R}^4$  con numpy
x = np.array([1.2, 3.4, 5.2, -6.7])
y = np.array([4.4, -2.3, 5.3, 8.9])

print('x = {}'.format(x))
print('y = {}'.format(y))
```

```
x = [ 1.2  3.4  5.2 -6.7]
y = [ 4.4 -2.3  5.3  8.9]
```

Propiedad 1.

El resultado debería ser:

```
x = [ 1.2  3.4  5.2 -6.7]
y = [ 4.4 -2.3  5.3  8.9]
z = x + y = [ 5.6  1.1 10.5  2.2]
```

```

#### Propiedad 1.

#### BEGIN SOLUTION
z = x + y

print('x = {}'.format(x))
print('y = {}'.format(y))
print('z = x + y = {}'.format(z))
#### END SOLUTION

```

```

x = [ 1.2  3.4  5.2 -6.7]
y = [ 4.4 -2.3  5.3  8.9]
z = x + y = [ 5.6  1.1 10.5  2.2]

```

Propiedad 2.

El resultado debería ser:

```

x + y = [ 5.6  1.1 10.5  2.2]   y + x = [ 5.6  1.1 10.5  2.2]

¿ x + y == y + x ? : [ True  True  True  True]

```

```

#### Propiedad 2.

#### BEGIN SOLUTION
xpy = x + y
ypx = y + x
print(' x + y = {} \t y + x = {}'.format(xpy, ypx))
print('\n ¿ x + y == y + x ? : {}'.format(xpy == ypx))
#### END SOLUTION

```

```

x + y = [ 5.6  1.1 10.5  2.2]   y + x = [ 5.6  1.1 10.5  2.2]

¿ x + y == y + x ? : [ True  True  True  True]

```

Propiedad 3.

El resultado debería ser:

```

x = [ 1.2  3.4  5.2 -6.7]   y = [ 4.4 -2.3  5.3  8.9]   z = [ 5.6  1.1 10.5  2.2]
x + (y + z) = [11.2  2.2 21.  4.4]   (x + y) + z = [11.2  2.2 21.  4.4]

¿ x + (y + z) == (x + y) + z : [ True  True  True  True]

```

```
#### Propiedad 3.
```

```
#### BEGIN SOLUTION
```

```
print(' x = {} \t y = {} \t z = {}'.format(x, y, z))
print(' x + (y + z)= {} \t (x + y) + z = {}'.format(x + (y + z), (x + y) + z))
print('\n ¿ x + (y + z) == (x + y) + z : {}'.format(np.isclose(x + (y + z), (x +
#### END SOLUTION
```

```
x = [ 1.2  3.4  5.2 -6.7]   y = [ 4.4 -2.3  5.3  8.9]   z = [ 5.6  1.1 10.5  2.2]
x + (y + z)= [11.2  2.2 21.  4.4]       (x + y) + z = [11.2  2.2 21.  4.4]
```

```
¿ x + (y + z) == (x + y) + z : [ True  True  True  True]
```

Propiedad 4.

El resultado debería ser:

```
x = [ 1.2  3.4  5.2 -6.7]   cero = [0. 0. 0. 0.]
x + cero = [ 1.2  3.4  5.2 -6.7]
```

```
#### Propiedad 4.
```

```
#### BEGIN SOLUTION
```

```
cero = np.zeros(4)
print(' x = {} \t cero = {}'.format(x, cero))
print(' x + cero = {}'.format(x + cero))
#### END SOLUTION
```

```
x = [ 1.2  3.4  5.2 -6.7]   cero = [0. 0. 0. 0.]
x + cero = [ 1.2  3.4  5.2 -6.7]
```

Propiedad 5.

El resultado debería ser:

```
x = [ 1.2  3.4  5.2 -6.7]   -x = [-1.2 -3.4 -5.2  6.7]
x + (-x) = [0. 0. 0. 0.]
```

```
#### Propiedad 5.
```

```
#### BEGIN SOLUTION
```

```
print(' x = {} \t -x = {}'.format(x, -x))
print(' x + (-x) = {}'.format(x + (-x)))
#### END SOLUTION
```

```
x = [ 1.2  3.4  5.2 -6.7]   -x = [-1.2 -3.4 -5.2  6.7]
x + (-x) = [0.  0.  0.  0.]
```

Propiedad 6.

El resultado debería ser:

```
 $\alpha = 1.5$       x = [ 1.2  3.4  5.2 -6.7]
 $\alpha * x = [ 1.8    5.1    7.8 -10.05]$ 
```

```
### Propiedad 6.
```

```
### BEGIN SOLUTION
```

```
 $\alpha = 1.5$ 
```

```
 $\alpha x = \alpha * x$ 
```

```
print('  $\alpha = {}$  \t x = {} \n  $\alpha * x = {}$  \n '.format( $\alpha$ , x,  $\alpha x$ ))
```

```
### END SOLUTION
```

```
 $\alpha = 1.5$       x = [ 1.2  3.4  5.2 -6.7]
 $\alpha * x = [ 1.8    5.1    7.8 -10.05]$ 
```

Propiedad 7.

El resultado debería ser:

```
 $\alpha = 1.5$ 
x = [ 1.2  3.4  5.2 -6.7]   y = [ 4.4 -2.3  5.3  8.9]
 $\alpha * (x + y) = [ 8.4    1.65 15.75  3.3 ]$ 
 $\alpha * x + \alpha * y = [ 8.4    1.65 15.75  3.3 ]$ 
¿  $\alpha * (x + y) == \alpha * x + \alpha * y$  ? : [ True  True  True  True]
```

```
### Propiedad 7.
```

```
### BEGIN SOLUTION
```

```
print('  $\alpha = {}$ '.format( $\alpha$ ))
```

```
print(' x = {} \t y = {}'.format(x, y))
```

```
print('  $\alpha * (x + y) = {}$ '.format( $\alpha * (x + y)$ ))
```

```
print('  $\alpha * x + \alpha * y = {}$ '.format( $\alpha * x + \alpha * y$ ))
```

```
print(' ¿  $\alpha * (x + y) == \alpha * x + \alpha * y$  ? : {}'.format(np.isclose( $\alpha * (x + y)$ ,  $\alpha * x + \alpha * y$ ))))
```

```
### END SOLUTION
```

```
 $\alpha = 1.5$ 
x = [ 1.2  3.4  5.2 -6.7]   y = [ 4.4 -2.3  5.3  8.9]
 $\alpha * (x + y) = [ 8.4    1.65 15.75  3.3 ]$ 
```



```

α * x + α * y = [ 8.4   1.65 15.75  3.3 ]
¿ α * (x + y) == α * x + α * y ? : [ True  True  True  True]

```

Propiedad 8.

El resultado debería ser:

```

α = 2.0    β = 1.5    α + β = 3.5
x = [ 1.2  3.4  5.2 -6.7]
(α + β) * x = [ 4.2   11.9   18.2  -23.45]
α * x + β * x = [ 4.2   11.9   18.2  -23.45]
¿ (α + β) * x == α * x + β * x ? : [ True  True  True  True]

```

```

#### Propiedad 8.

```

```

#### BEGIN SOLUTION

```

```

α = 2.0

```

```

β = 1.5

```

```

print(' α = {} \t β = {} \t α + β = {}'.format(α, β, α + β))

```

```

print(' x = {}'.format(x))

```

```

print(' (α + β) * x = {}'.format((α + β) * x))

```

```

print(' α * x + β * x = {}'.format(α * x + β * x))

```

```

print(' ¿ (α + β) * x == α * x + β * x ? : {}'.format(np.isclose((α + β) * x, α * x + β * x)))

```

```

#### END SOLUTION

```

```

α = 2.0    β = 1.5    α + β = 3.5
x = [ 1.2  3.4  5.2 -6.7]
(α + β) * x = [ 4.2   11.9   18.2  -23.45]
α * x + β * x = [ 4.2   11.9   18.2  -23.45]
¿ (α + β) * x == α * x + β * x ? : [ True  True  True  True]

```

Propiedad 9.

El resultado debería ser:

```

α = 2.0    β = 1.5    x = [ 1.2  3.4  5.2 -6.7]
α * (β * x) = [ 3.6  10.2  15.6 -20.1]
(α * β) * x = [ 3.6  10.2  15.6 -20.1]
¿ α * (β * x) == (α * β) * x ? : [ True  True  True  True]

```

```

#### Propiedad 9.

```

```

#### BEGIN SOLUTION

```

```

α = 2.0

```

```

β = 1.5

```

```

print(' α = {} \t β = {} \t x = {}'.format(α, β, x))
print(' α * (β * x) = {}'.format(α * (β * x)))
print(' (α * β) * x = {}'.format((α * β) * x))
print(' ¿ α * (β * x) == (α * β) * x ? : {}'.format(np.isclose(α * (β * x), (α *
### END SOLUTION

```

```

α = 2.0      β = 1.5      x = [ 1.2  3.4  5.2 -6.7]
α * (β * x) = [  3.6  10.2  15.6 -20.1]
(α * β) * x = [  3.6  10.2  15.6 -20.1]
¿ α * (β * x) == (α * β) * x ? : [ True  True  True  True]

```

Propiedad 10.

El resultado debería ser:

```

α = 1.0      x = [ 1.2  3.4  5.2 -6.7]
α * x = [ 1.2  3.4  5.2 -6.7]
¿ α * x == x ? : [ True  True  True  True]

```

```

### Propiedad 10.

### BEGIN SOLUTION
α = 1.0

print(' α = {} \t x = {}'.format(α, x))
print(' α * x = {}'.format(α * x))
print(' ¿ α * x == x ? : {}'.format(np.isclose(α * x, x)))
### END SOLUTION

```

```

α = 1.0      x = [ 1.2  3.4  5.2 -6.7]
α * x = [ 1.2  3.4  5.2 -6.7]
¿ α * x == x ? : [ True  True  True  True]

```

1.4 Ejercicio 2.

Definimos dos vectores $\vec{x} = (x_1, x_2)$ y $\vec{y} = (y_1, y_2)$ ambos en \mathbb{R}^2 y $\alpha \in \mathbb{R}$.

Al ejecutar la siguiente celda, se presenta un simulador interactivo en el cual se implementa la operación conocida como SAXPY que se define como: $\alpha\vec{x} + \vec{y}$.

Modifica el valor de α y de las componentes de los vectores \vec{x} y \vec{y} , y observa lo que sucede cuando:

- $\alpha = 1.0, \alpha > 1.0, \alpha < 1.0, \alpha = 0.0, \alpha < 0.0$.
- $\vec{y} = (0, 0)$.

NOTA: para ejecutar el simulador haz clic en el botón de *play* ►

```
%run ./vecsapace.py
```