

3 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()
```

3.1 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:

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

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]
```

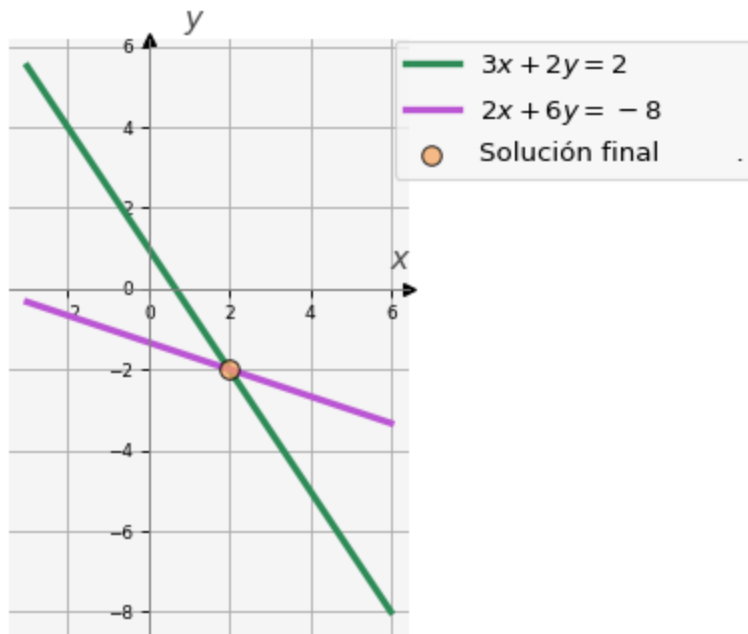
```
[2 6]]
```

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.

4 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.

4.1 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:

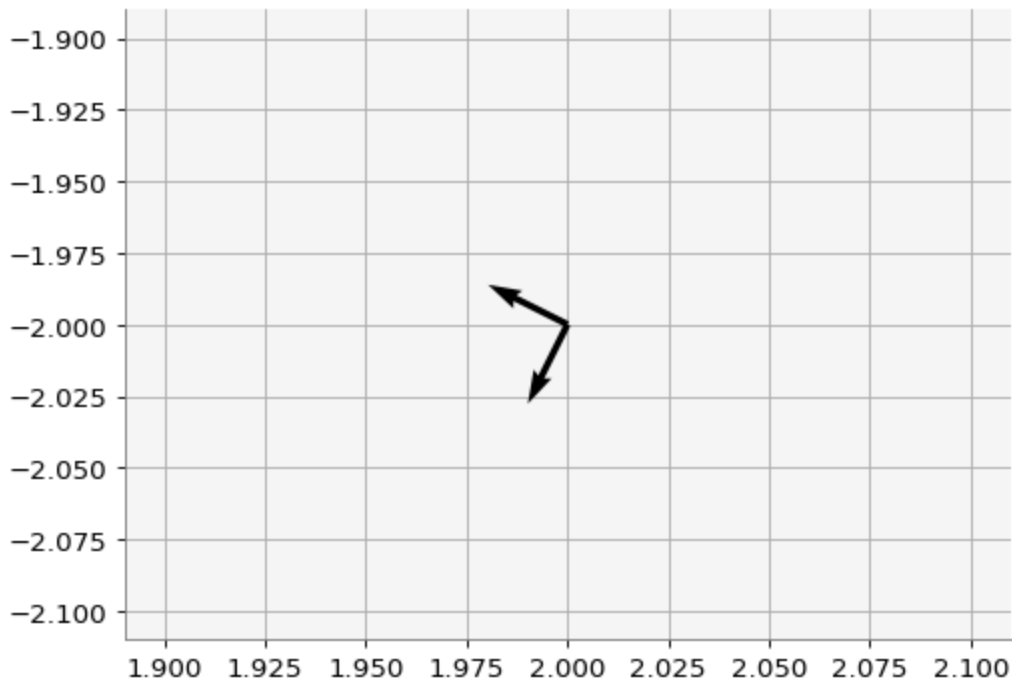
```
# Usando la función eigen_land() de macti
wA, vA = mmat.eigen_land(A)
```

```
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]]])

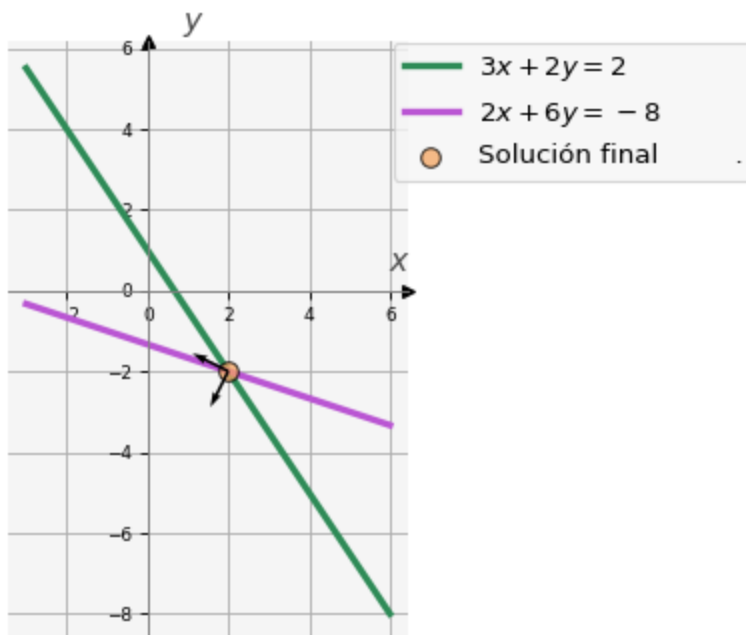
v = mvis.Plotter()
v.quiver(1, xv[0], xv[1], vA[0], vA[1], scale=10, zorder=6)
v.grid()
v.show()
```



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



4.2 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))

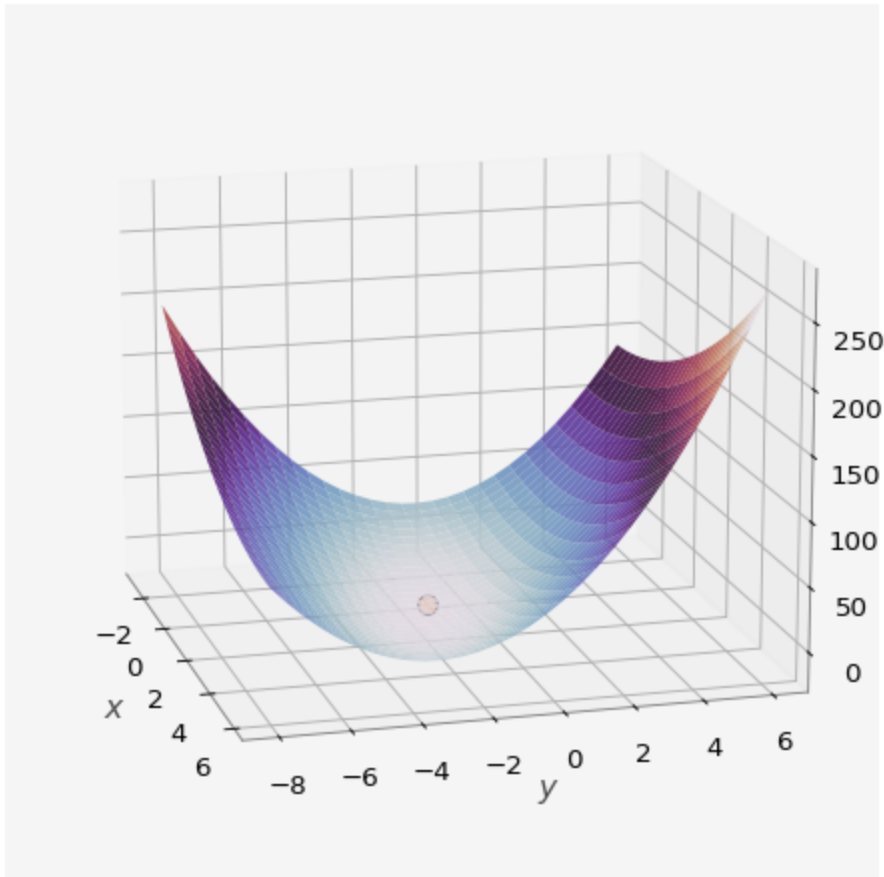
# Cálculo
for i in range(size_grid):
    for j in range(size_grid):
        xc = np.array([xg[i,j], yg[i,j]])
        z[i,j] = f(A,b,0,xc)
```

/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 $f(\mathbf{x})$.

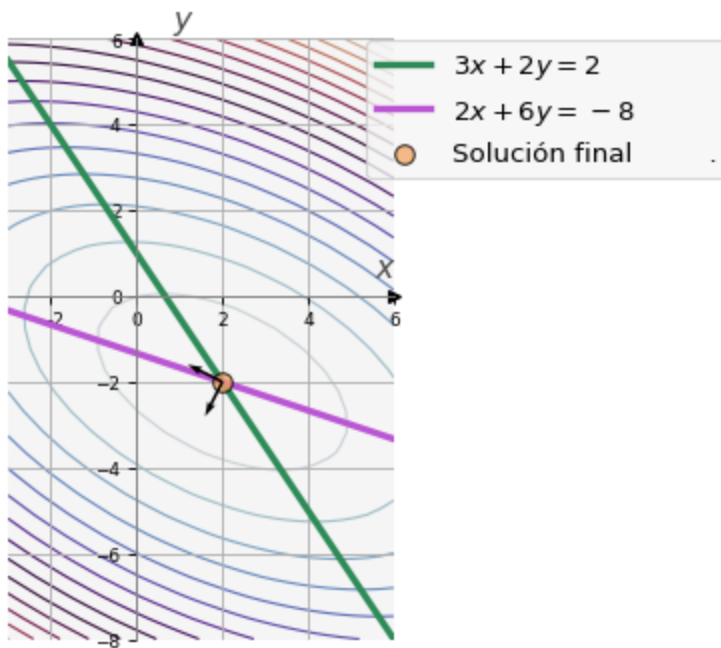
```
axis_par = [dict(projection='3d', aspect='auto', xlabel = '$x$', ylabel = '$y$',
v = mvis.Plotter(1,1, axis_par, dict(figsize=(8,6)))
v.plot_surface(1, xg, yg, z, cmap='twilight', alpha=0.90) # f(x)
v.scatter(1, sol[0], sol[1], fc='sandybrown', ec='k', s = 75, zorder=5, label='Sol')
v.axes(1).view_init(elev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



4.3 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

```

\$

4.4 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

```

4.5 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 ejecu
#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)

```

```

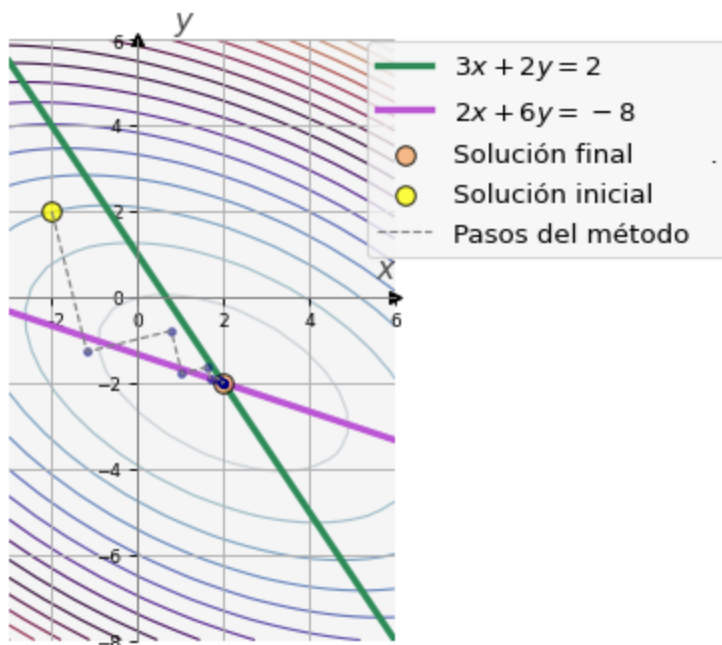
#quizz.eval_numeric('1', solGrad)
#quizz.eval_numeric('2', eGrad[-1])
#quizz.eval_numeric('3', rGrad[-1])
#quizz.eval_numeric('4', itGrad)

```

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

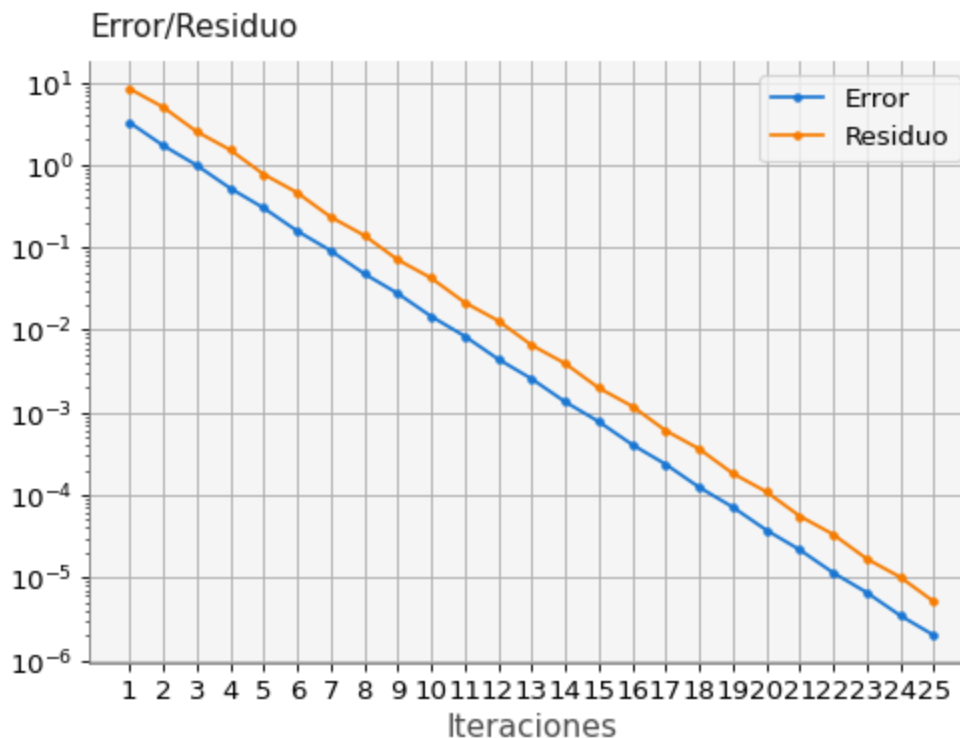


Grafica del error y el residuo.

```
# 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()
```



4.6 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

\$

4.6.1 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

```

4.7 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

```

```

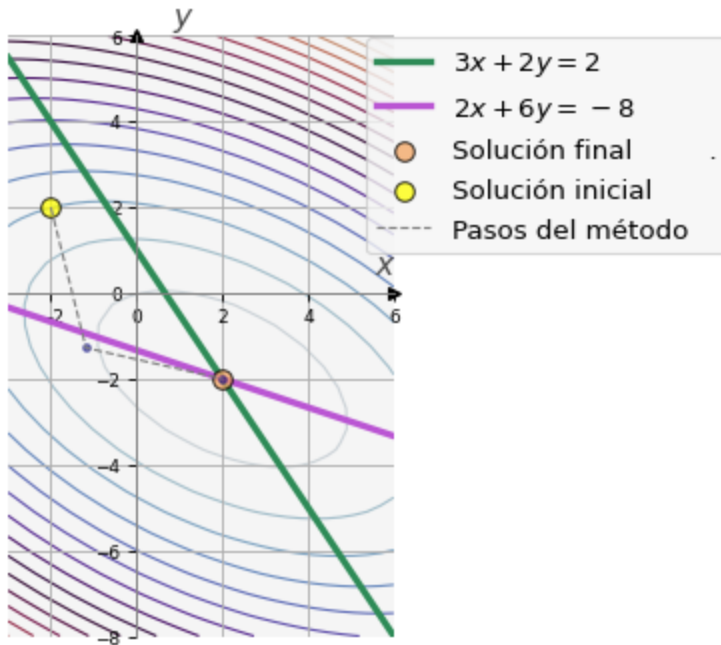
1 3.261835423 (-1.180722892, -1.277108434)
2 0.000000000 (2.000000000, -2.000000000)

```

```
#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



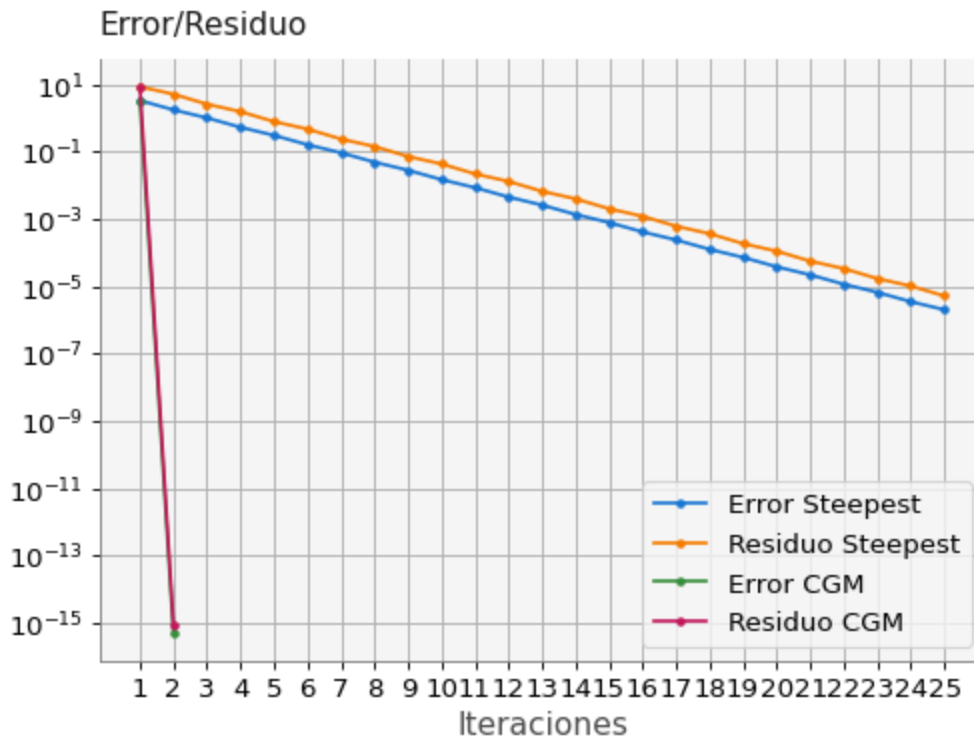
```
# Lista con el número de las iteraciones
l_itGrad = list(range(1,itGrad+1))
l_itCGM = list(range(1,itCGM+1))

# 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()
```



4.8 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)

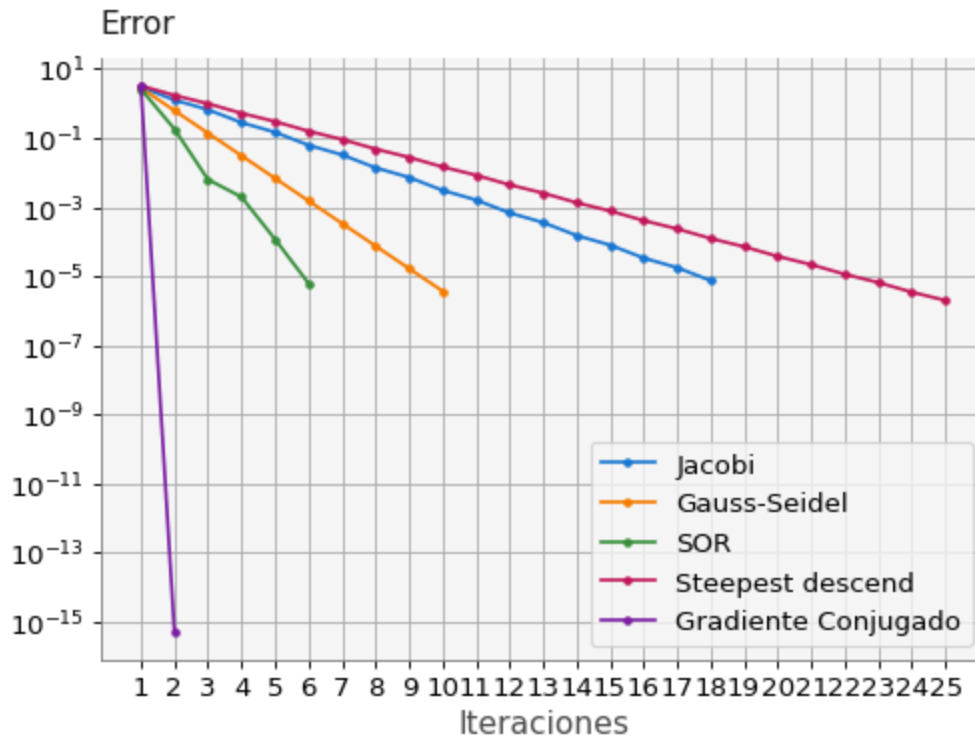
# Gráfica del error
v = mvis.Plotter(1,1,[a_p])
v.axes(1).set_title('Error', loc='left')
v.plot(1, l_itJ, eJ, marker='.', label='Jacobi')
v.plot(1, l_itG, eG, marker='.', label='Gauss-Seidel')
```

```

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 Conducción de Calor estacionaria en 2D.

Objetivo General - Resolver numérica y computacionalmente la ecuación de conducción de calor estacionaria en dos dimensiones usando un método implícito.

Objetivos particulares - Definir los parámetros físicos y numéricos. - Definir la malla del dominio. - Definir la temperatura inicial junto con sus condiciones de frontera y graficarla sobre la malla. - Definir el sistema lineal y resolverlo. - Graficar la solución.

[HeCompA - 02_cond_calor](#) 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

4.1 Introducción.

Jean-Baptiste Joseph Fourier fue un matemático y físico francés que ejerció una fuerte influencia en la ciencia a través de su trabajo *Théorie analytique de la chaleur*. En este trabajo mostró que es posible analizar la conducción de calor en cuerpos sólidos en términos de series matemáticas infinitas, las cuales ahora llevan su nombre: *Series de Fourier*. Fourier comenzó su trabajo en 1807, en Grenoble, y lo completó en París en 1822. Su trabajo le permitió expresar la conducción de calor en objetos bidimensionales (hojas muy delgadas de algún material) en términos de una ecuación diferencial:

$$\frac{\partial T}{\partial t} = \kappa \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) + S$$

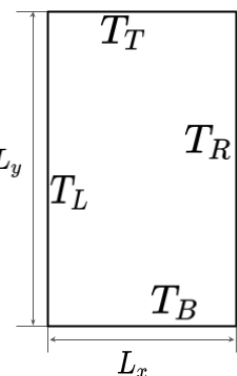
donde u representa la temperatura en un instante de tiempo t y en un punto (x, y) del plano Cartesiano, κ es la conductividad del material y S una fuente de calor.

4.2 Conducción estacionaria en 2D.

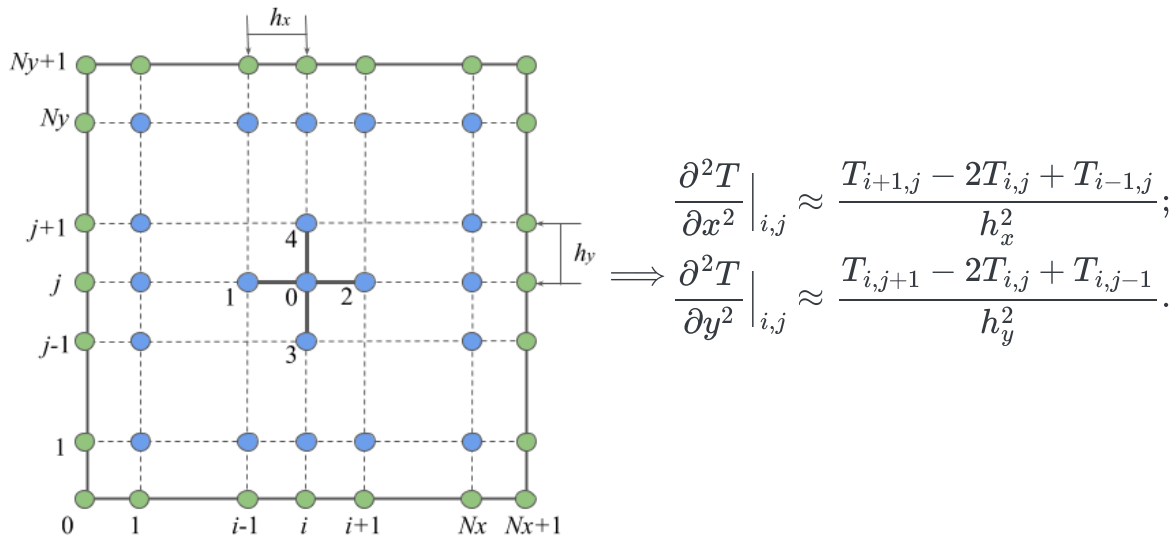
Cuando el problema es estacionario, es decir no hay cambios en el tiempo, y el dominio de estudio es una placa en dos dimensiones, como la que se muestra en la figura, podemos escribir el problema como sigue:

$$-\kappa \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) = S \quad (1)$$

Podemos aplicar condiciones de frontera son de tipo Dirichlet o Neumann en las paredes de la placa. En la figura se distingue T_L , T_R , T_T y T_B que corresponden a las temperaturas dadas en las paredes izquierda (LEFT), derecha (RIGHT), arriba (TOP) y abajo (BOTTOM), respectivamente.



A la ecuación (1) le podemos aplicar el método de diferencias finitas:



de tal manera que obtendríamos un sistema de ecuaciones lineales como el siguiente:

$$\begin{bmatrix} -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \dots \\ 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \dots \\ 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \dots \\ 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & \dots \\ 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & \dots \\ 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & \dots \\ 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & \dots \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & \dots \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & \dots \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{4,1} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \\ u_{4,2} \\ u_{1,3} \\ u_{2,3} \\ u_{3,3} \\ \vdots \end{bmatrix} = \begin{bmatrix} f_{1,1} \\ f_{2,1} \\ f_{3,1} \\ f_{4,1} \\ f_{1,2} \\ f_{2,2} \\ f_{3,2} \\ f_{4,2} \\ f_{1,3} \\ f_{2,3} \\ f_{3,3} \\ \vdots \end{bmatrix}$$

En general un sistema de ecuaciones lineales puede contener n ecuaciones con n incógnitas y se ve como sigue:

$$A \cdot \mathbf{x} = \mathbf{b} \Rightarrow \begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0n} \\ a_{10} & a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}$$

El sistema se puede resolver usando diferentes tipos de métodos.

```
import numpy as np
import matplotlib.pyplot as plt
import macti.visual as mvis
```

4.3 Parámetros físicos y numéricos

```
# Tamaño del dominio
Lx = 1.0
Ly = 1.0
k = 1.0
# Número de nodos en cada eje
Nx = 4
```

```

Ny = 4

# Número total de nodos en cada eje incluyendo las fronteras
NxT = Nx + 2
NyT = Ny + 2

# Número total de nodos
NT = NxT * NyT

# Número total de incógnitas
N = Nx * Ny

# Tamaño de la malla en cada dirección
hx = Lx / (Nx+1)
hy = Ly / (Ny+1)

# Coordenadas de la malla
xn = np.linspace(0,Lx,NxT)
yn = np.linspace(0,Ly,NyT)

# Generación de una rejilla
xg, yg = np.meshgrid(xn, yn, indexing='ij')

```

```

print('Total de nodos en x = {}, en y = {}'.format(NxT, NyT))
print('Total de incógnitas = {}'.format(N))
print('Coordenadas en x : {}'.format(xn))
print('Coordenadas en y : {}'.format(yn))
print('hx = {}, hy = {}'.format(hx, hy))

```

```

Total de nodos en x = 6, en y = 6
Total de incógnitas = 16
Coordenadas en x : [0.  0.2 0.4 0.6 0.8 1. ]
Coordenadas en y : [0.  0.2 0.4 0.6 0.8 1. ]
hx = 0.2, hy = 0.2

```

4.3.1 Graficación de la malla del dominio

```

from mpl_toolkits.axes_grid1 import make_axes_locatable
def set_axes(ax):
    """
    Configura la razón de aspecto, quita las marcas de los ejes y el marco.

    Parameters
    -----
    ax: axis
    Ejes que se van a configurar.
    """
    ax.set_aspect('equal')
    ax.set_xticks([])

```

```

ax.set_yticks([])
ax.spines['bottom'].set_visible(False)
ax.spines['left'].set_visible(False)

def plot_mesh(ax, xg, yg):
    """
    Dibuja la malla del dominio.

    Paramters
    -----
    ax: axis
    Son los ejes donde se dibujará la malla.

    xn: np.array
    Coordenadas en x de la malla.

    yn: np.array
    Coordenadas en y de la malla.
    """
    set_axes(ax)

    xn = xg[:,0]
    yn = yg[0,:]

    for xi in xn:
        ax.vlines(xi, ymin=yn[0], ymax=yn[-1], lw=0.5, color='darkgray')

    for yi in yn:
        ax.hlines(yi, xmin=xn[0], xmax=xn[-1], lw=0.5, color='darkgray')

    ax.scatter(xg,yg, marker='.', color='darkgray')

def plot_frame(ax, xn, yn, lw = 0.5, color = 'k'):
    """
    Dibuja el recuadro de la malla.

    Paramters
    -----
    ax: axis
    Son los ejes donde se dibujará la malla.

    xn: np.array
    Coordenadas en x de la malla.

    yn: np.array
    Coordenadas en y de la malla.
    """
    set_axes(ax)

    # Dibujamos dos líneas verticales
    ax.vlines(xn[0], ymin=yn[0], ymax=yn[-1], lw = lw, color=color)

```

```
ax.vlines(xn[-1], ymin=yn[0], ymax=yn[-1], lw = lw, color=color)
```

```
# Dibujamos dos líneas horizontales
```

```
ax.hlines(yn[0], xmin=xn[0], xmax=xn[-1], lw = lw, color=color)
```

```
ax.hlines(yn[-1], xmin=xn[0], xmax=xn[-1], lw = lw, color=color)
```

```
def set_canvas(ax, Lx, Ly):
```

```
    """
```

```
    Configura un lienzo para hacer las gráficas más estéticas.
```

```
    Parameters
```

```
    -----
```

```
    ax: axis
```

```
    Son los ejes que se van a configurar.
```

```
    Lx: float
```

```
    Tamaño del dominio en dirección x.
```

```
    Ly: float
```

```
    Tamaño del dominio en dirección y.
```

```
    Returns
```

```
    -----
```

```
    cax: axis
```

```
    Eje donde se dibuja el mapa de color.
```

```
    """
```

```
    set_axes(ax)
```

```
    lmax = max(Lx,Ly)
```

```
    offx = lmax * 0.01
```

```
    offy = lmax * 0.01
```

```
    ax.set_xlim(-offx, Lx+offx)
```

```
    ax.set_ylim(-offy, Ly+offy)
```

```
    ax.grid(False)
```

```
    ax.set_aspect('equal')
```

```
    divider = make_axes_locatable(ax)
```

```
    cax = divider.append_axes("right", "5%", pad="3%")
```

```
    cax.set_xticks([])
```

```
    cax.set_yticks([])
```

```
    cax.spines['bottom'].set_visible(False)
```

```
    cax.spines['left'].set_visible(False)
```

```
    return cax
```

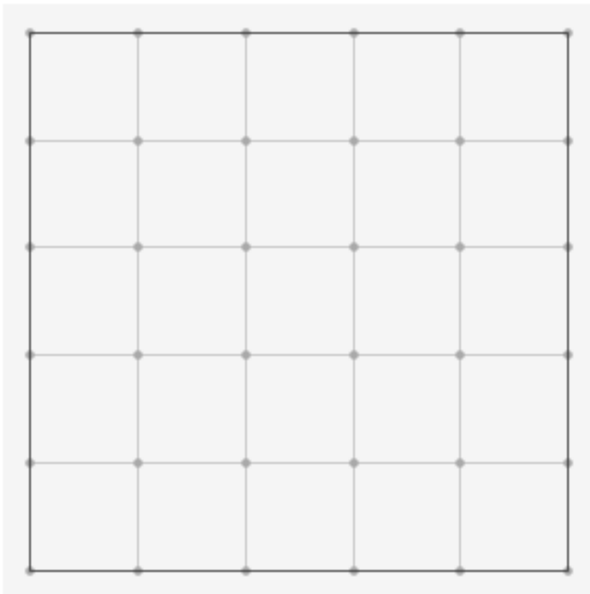
```
fig = plt.figure()
```

```
ax = plt.gca()
```

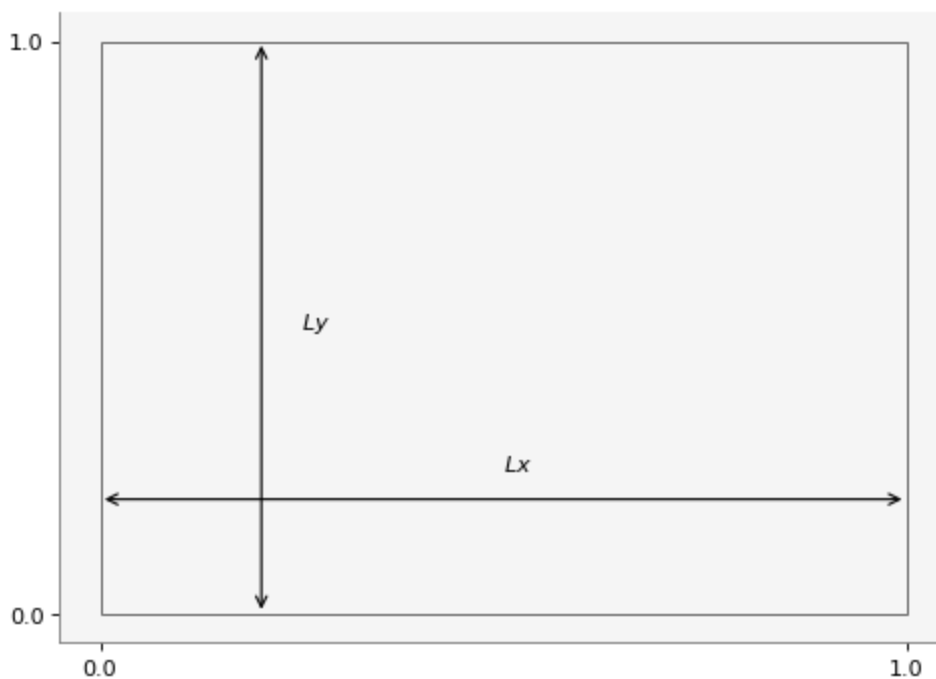
```
# Ejecutamos la función plot_mesh(...)
```

```
plot_mesh(ax, xg, yg)
```

```
# Dibujamos el recuadro con la función plot_fame(...)
plot_frame(ax, xn, yn)
```

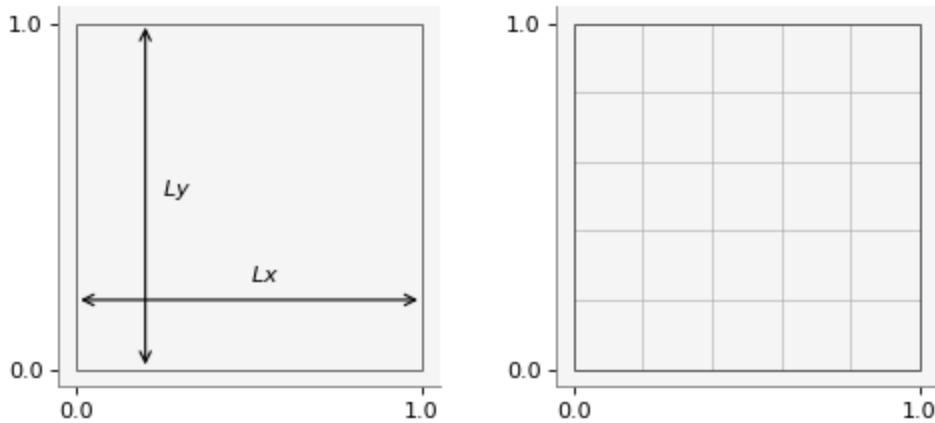


```
vis = mvis.Plotter(1,1)
vis.draw_domain(1, xg, yg)
```



```
vis = mvis.Plotter(1,2,[dict(aspect='equal'), dict(aspect='equal')])
vis.draw_domain(1, xg, yg)
```

```
vis.plot_mesh2D(2, xg, yg)
vis.plot_frame(2, xg, yg)
```



4.4 Campo de temperaturas y sus condiciones de frontera

```
# Definición de un campo escalar en cada punto de la malla
T = np.zeros((NxT, NyT))

# Condiciones de frontera
TB = 1.0
TT = -1.0

T[0, :] = 0.0 # LEFT
T[-1, :] = 0.0 # RIGHT
T[:, 0] = TB # BOTTOM
T[:, -1] = TT # TOP

print('Campo escalar T ({}):\n {}'.format(T.shape, T))
```

```
Campo escalar T ((6, 6)):
[[ 1.  0.  0.  0.  0. -1.]
 [ 1.  0.  0.  0.  0. -1.]
 [ 1.  0.  0.  0.  0. -1.]
 [ 1.  0.  0.  0.  0. -1.]
 [ 1.  0.  0.  0.  0. -1.]
 [ 1.  0.  0.  0.  0. -1.]]
```

4.4.1 Graficación del campo escalar sobre la malla

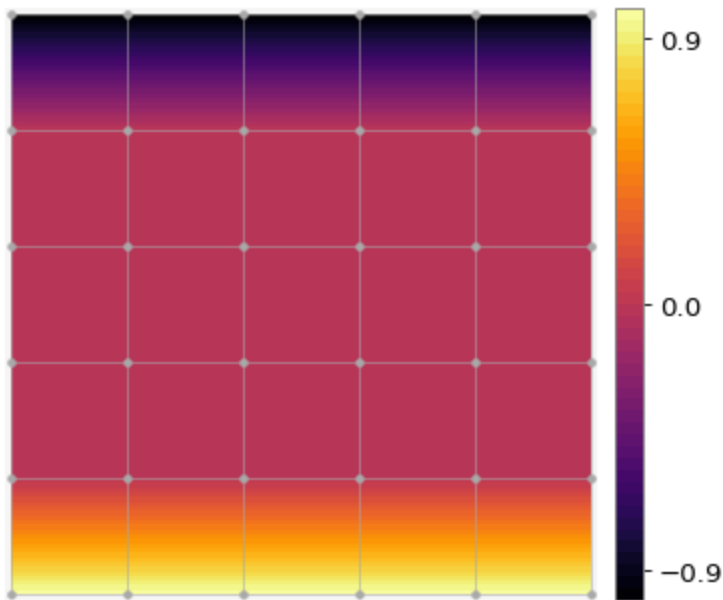
```
fig = plt.figure()
ax = plt.gca()
cax = set_canvas(ax, Lx, Ly)

c = ax.contourf(xg, yg, T, levels=50, cmap='inferno')
```

```

plot_mesh(ax, xg, yg)
fig.colorbar(c, cax=cax, ticks=[-0.9, 0.0, 0.9])
plt.show()

```

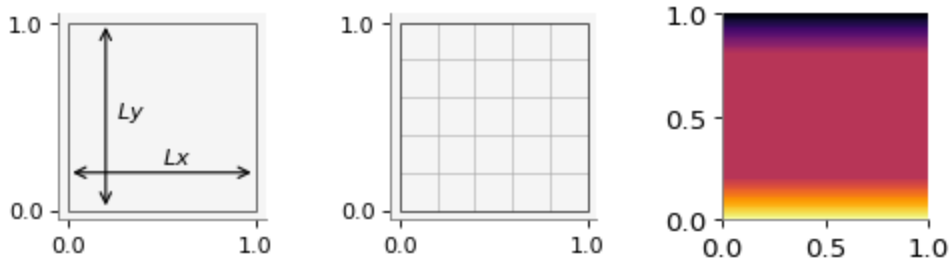


```

vis = mvis.Plotter(1,3,[dict(aspect='equal'), dict(aspect='equal'), dict(aspect='

vis.draw_domain(1, xg, yg)
vis.plot_mesh2D(2, xg, yg)
vis.plot_frame(2, xg, yg)
vis.contourf(3, xg, yg, T, levels=50, cmap='inferno')
vis.show()

```

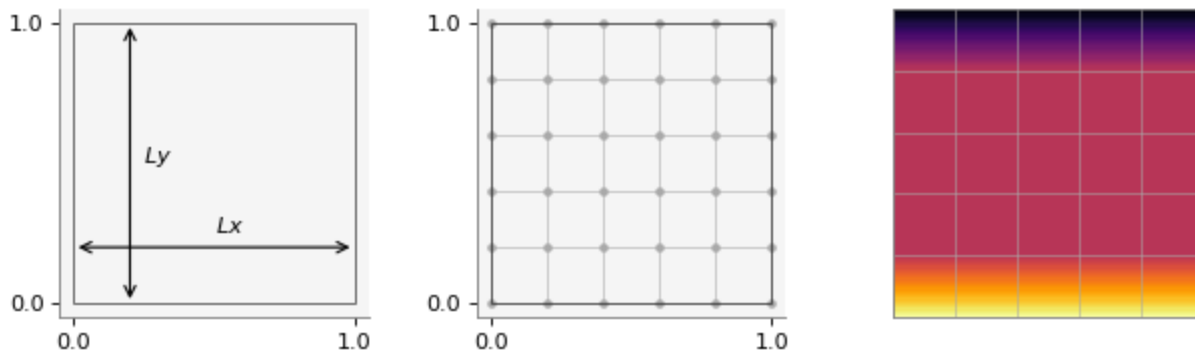


```

vis = mvis.Plotter(1,3,[dict(aspect='equal'), dict(aspect='equal'), dict(aspect='
dict(figsize=(8,16)))

vis.draw_domain(1, xg, yg)
vis.plot_mesh2D(2, xg, yg, nodeson=True)
vis.plot_frame(2, xg, yg)
vis.contourf(3, xg, yg, T, levels=50, cmap='inferno')
vis.plot_mesh2D(3,xg, yg)
vis.show()

```

4.5 Flujo de calor

Fourier también estableció una ley para el flujo de calor que se escribe como:

$$\vec{q} = -\kappa \nabla u = -\kappa \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y} \right)$$

```
def heat_flux(T, hx, hy):
    NxT, NyT = T.shape
    qx = np.zeros(T.shape)
    qy = qx.copy()

    for i in range(1,NxT-1):
        for j in range(1,NyT-1):
            qx[i,j] = -k * (T[i+1,j] - T[i-1,j]) / 2 * hx
            qy[i,j] = -k * (T[i,j+1] - T[i,j-1]) / 2 * hy
    return qx, qy
```

```
qx, qy = heat_flux(T, hx, hy)
```

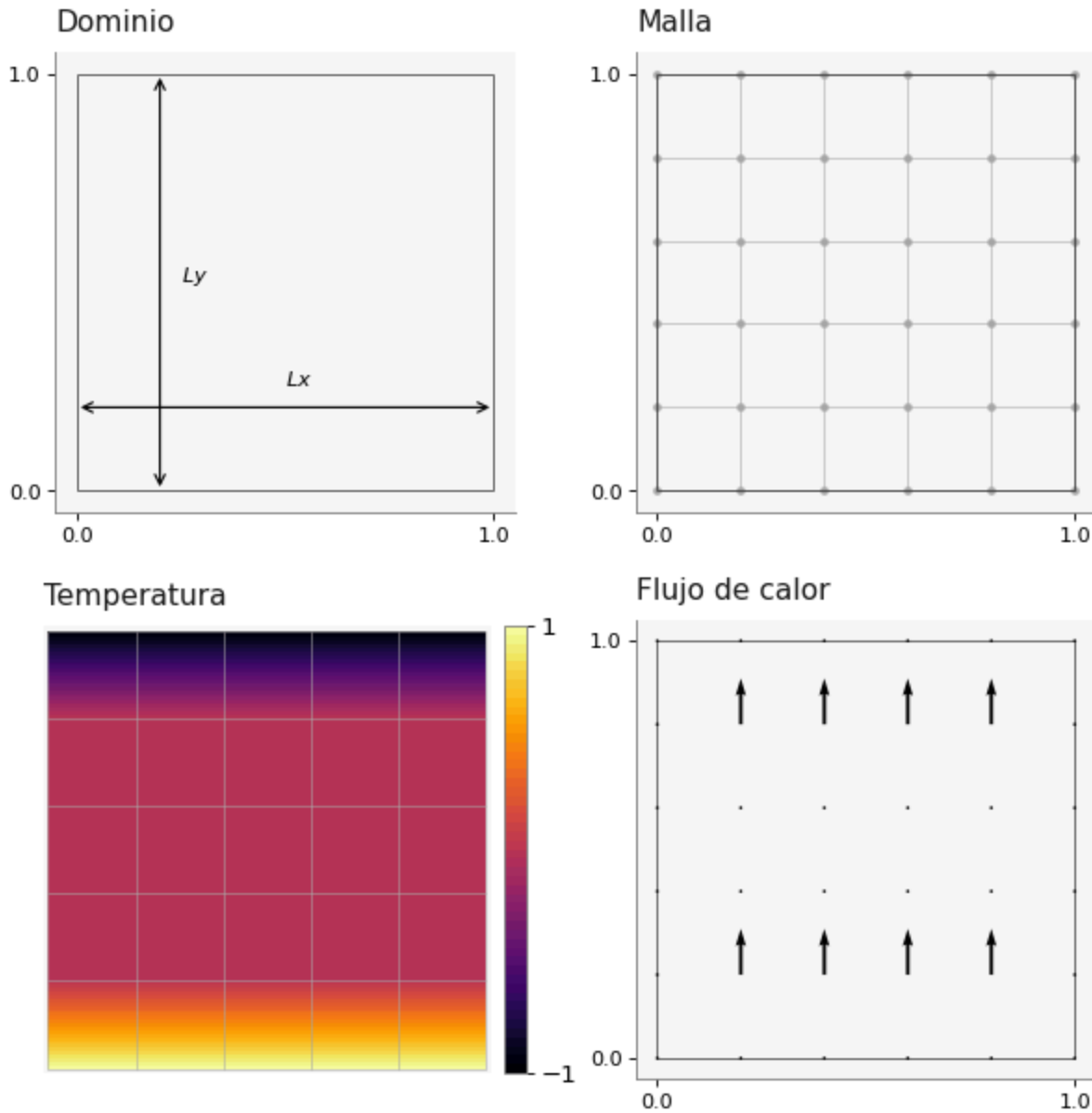
```
ax1 = dict(aspect='equal', title='Dominio')
ax2 = dict(aspect='equal', title='Malla')
ax3 = dict(aspect='equal', title='Temperatura')
ax4 = dict(aspect='equal', title='Flujo de calor')

vis = mvis.Plotter(2,2,[ax1, ax2, ax3, ax4],
                  dict(figsize=(8,8)))

vis.draw_domain(1, xg, yg)
vis.plot_mesh2D(2, xg, yg, nodeson=True)
vis.plot_frame(2, xg, yg)

cax3 = vis.set_canvas(3,Lx,Ly)
c = vis.contourf(3, xg, yg, T, levels=50, cmap='inferno')
vis.fig.colorbar(c, cax=cax3, ticks = [T.min(), T.max()], shrink=0.5, orientatio
vis.plot_mesh2D(3, xg, yg)
```

```
vis.plot_frame(4, xg, yg)
vis.quiver(4, xg, yg, qx, qy, scale=1)
vis.show()
```



4.6 Sistema lineal

```
import FDM
# La matriz del sistema. Usamos la función predefinida buildMatrix2D()
A = FDM.buildMatrix2D(Nx,Ny,-4)
A
```

```
array([[ -4.,  1.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.],
       [ 1., -4.,  1.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.]])
```

```

    0., 0., 0.],
[ 0., 1., -4., 1., 0., 0., 1., 0., 0., 0., 0., 0., 0.,
  0., 0., 0.],
[ 0., 0., 1., -4., 0., 0., 0., 1., 0., 0., 0., 0., 0.,
  0., 0., 0.],
[ 1., 0., 0., 0., -4., 1., 0., 0., 1., 0., 0., 0., 0.,
  0., 0., 0.],
[ 0., 1., 0., 0., 1., -4., 1., 0., 0., 1., 0., 0., 0.,
  0., 0., 0.],
[ 0., 0., 1., 0., 0., 1., -4., 1., 0., 0., 1., 0., 0.,
  0., 0., 0.],
[ 0., 0., 0., 1., 0., 0., 1., -4., 0., 0., 0., 1., 0.,
  0., 0., 0.],
[ 0., 0., 0., 0., 1., 0., 0., 0., -4., 1., 0., 0., 1.,
  0., 0., 0.],
[ 0., 0., 0., 0., 0., 1., 0., 0., 1., -4., 1., 0., 0.,
  1., 0., 0.],
[ 0., 0., 0., 0., 0., 0., 1., 0., 0., 1., -4., 1., 0.,
  0., 1., 0.],
[ 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 1., -4., 0.,
  0., 0., 1.],
[ 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., -4.,
  1., 0., 0.],
[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 1.,
  -4., 1., 0.],
[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
  1., -4., 1.],
[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.,
  0., 1., -4.]]))

```

```

# RHS
b = np.zeros((Nx,Ny))
b[:, 0] -= TB # BOTTOM
b[:, -1] -= TT # TOP
b

```

```

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

```

4.7 Solución del sistema

Revisamos el formato del vector b

```
b.shape
```

(4, 4)

El vector debe ser de una sola dimensión:

```
b.flatten()
```

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

```
# Calculamos la solución.
T_temp = np.linalg.solve(A, b.flatten())
T_temp
```

```
array([ 0.40909091,  0.11363636, -0.11363636, -0.40909091,  0.52272727,
        0.15909091, -0.15909091, -0.52272727,  0.52272727,  0.15909091,
       -0.15909091, -0.52272727,  0.40909091,  0.11363636, -0.11363636,
       -0.40909091])
```

```
T_temp.shape
```

(16,)

Colocamos la solución en el campo escalar T de manera adecuada

```
T[1:-1,1:-1] = T_temp.reshape(Nx,Ny)
T
```

```
array([[ 1.,  0.,  0.,  0.,  0.,
        -1.,  ],
       [ 1.,  0.40909091,  0.11363636, -0.11363636, -0.40909091,
        -1.,  ],
       [ 1.,  0.52272727,  0.15909091, -0.15909091, -0.52272727,
        -1.,  ],
       [ 1.,  0.52272727,  0.15909091, -0.15909091, -0.52272727,
        -1.,  ],
       [ 1.,  0.40909091,  0.11363636, -0.11363636, -0.40909091,
        -1.,  ],
       [ 1.,  0.,  0.,  0.,  0.,
        -1.,  ]])
```

```
qx, qy = heat_flux(T, hx, hy)
```

4.7.1 Gráfica de la solución

```

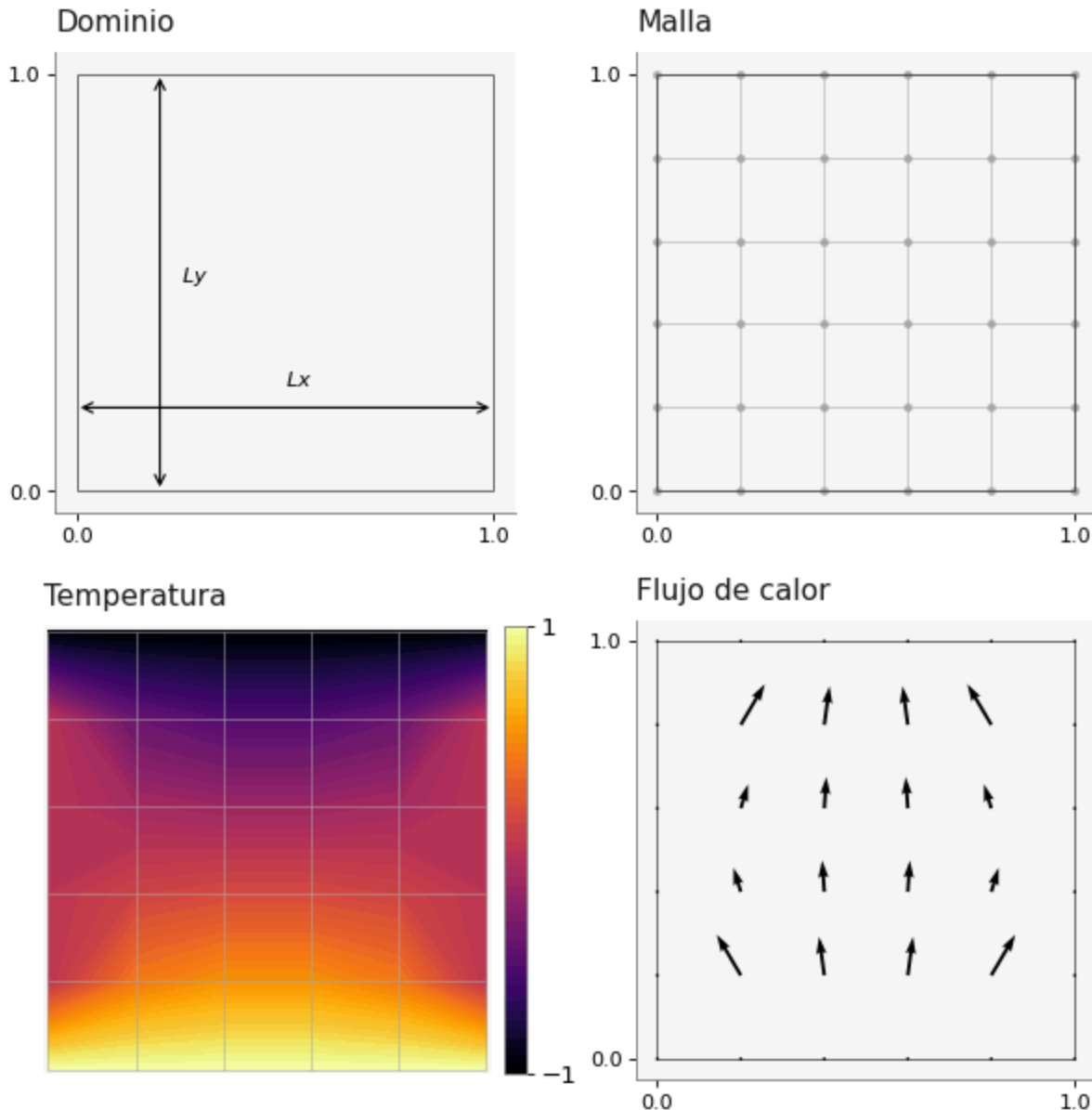
vis = mvis.Plotter(2,2,[ax1, ax2, ax3, ax4],
                  dict(figsize=(8,8)))

vis.draw_domain(1, xg, yg)
vis.plot_mesh2D(2, xg, yg, nodeson=True)
vis.plot_frame(2, xg, yg)

cax3 = vis.set_canvas(3,Lx,Ly)
c = vis.contourf(3, xg, yg, T, levels=50, cmap='inferno')
vis.fig.colorbar(c, cax=cax3, ticks = [T.min(), T.max()], shrink=0.5, orientation='vertical')
vis.plot_mesh2D(3, xg, yg)

vis.plot_frame(4, xg, yg)
vis.quiver(4, xg, yg, qx, qy, scale=1)
vis.show()

```



4.7.2 Interactivo

```
def heat_cond(Lx, Ly, Nx, Ny):
    # Número total de nodos en cada eje incluyendo las fronteras
    NxT = Nx + 2
    NyT = Ny + 2

    # Número total de nodos
    NT = NxT * NyT

    # Número total de incógnitas
    N = Nx * Ny

    # Tamaño de la malla en cada dirección
    hx = Lx / (Nx+1)
    hy = Ly / (Ny+1)

    # Coordenadas de la malla
    xn = np.linspace(0, Lx, NxT)
    yn = np.linspace(0, Ly, NyT)

    # Generación de una rejilla
    xg, yg = np.meshgrid(xn, yn, indexing='ij')

    # Definición de un campo escalar en cada punto de la malla
    T = np.zeros((NxT, NyT))

    # Condiciones de frontera
    TB = 1.0
    TT = -1.0

    T[0, :] = 0.0 # LEFT
    T[-1, :] = 0.0 # RIGHT
    T[:, 0] = TB # BOTTOM
    T[:, -1] = TT # TOP

    # La matriz del sistema. Usamos la función predefinida buildMatrix2D()
    A = FDM.buildMatrix2D(Nx, Ny, -4)

    # RHS
    b = np.zeros((Nx, Ny))
    b[:, 0] -= TB # BOTTOM
    b[:, -1] -= TT # TOP

    # Calculamos la solución.
    T[1:-1, 1:-1] = np.linalg.solve(A, b.flatten()).reshape(Nx, Ny)

    # Calculamos el flujo de calor
    qx, qy = heat_flux(T, hx, hy)
```

```
ax1 = dict(aspect='equal', title='Dominio')
ax2 = dict(aspect='equal', title='Malla')
ax3 = dict(aspect='equal', title='Temperatura')
ax4 = dict(aspect='equal', title='Flujo de calor')

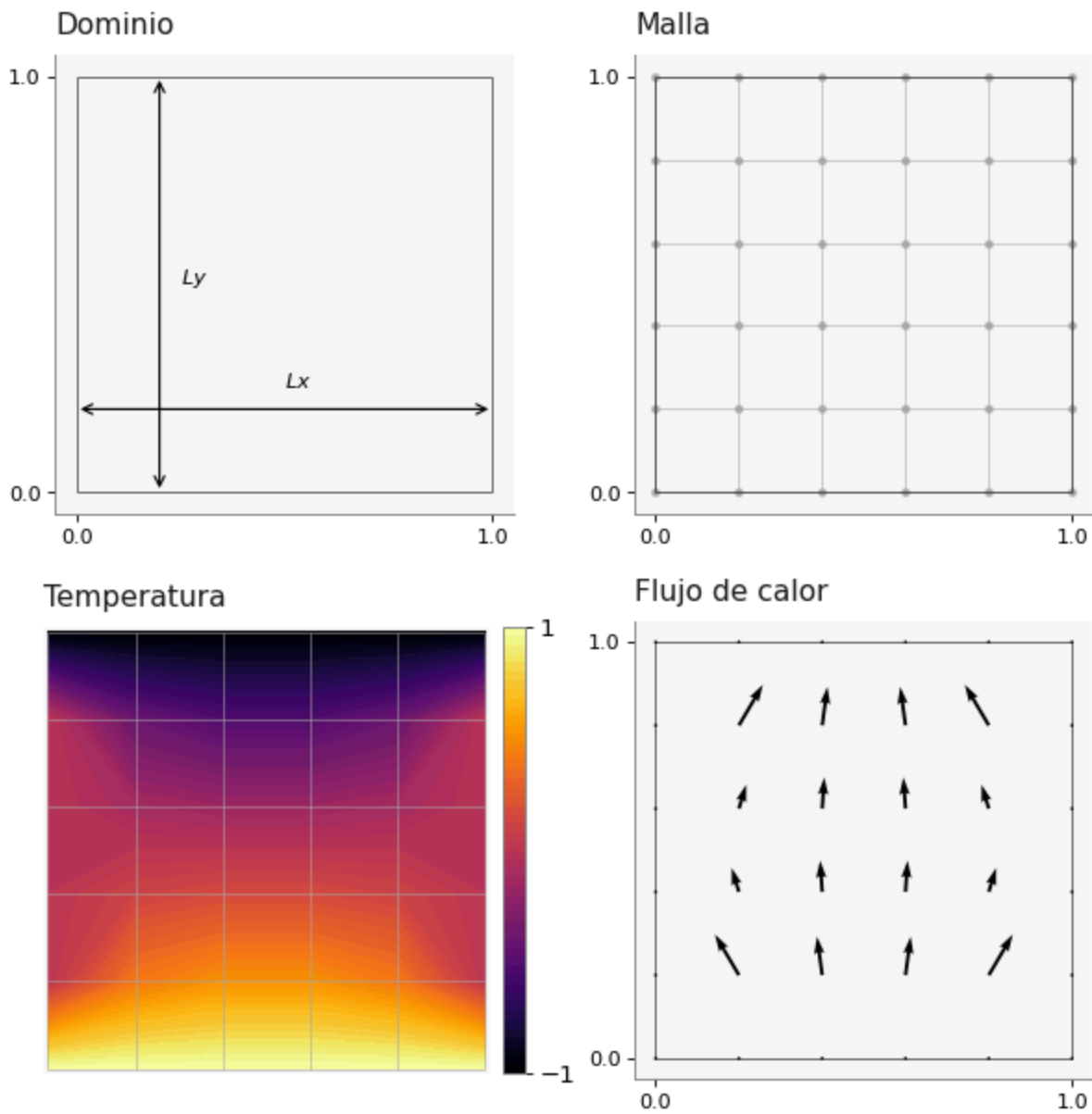
vis = mvis.Plotter(2,2,[ax1, ax2, ax3, ax4],
                  dict(figsize=(8,8)))

vis.draw_domain(1, xg, yg)
vis.plot_mesh2D(2, xg, yg, nodeson=True)
vis.plot_frame(2, xg, yg)

cax3 = vis.set_canvas(3,Lx,Ly)
c = vis.contourf(3, xg, yg, T, levels=50, cmap='inferno')
vis.fig.colorbar(c, cax=cax3, ticks = [T.min(), T.max()], shrink=0.5, orienta
vis.plot_mesh2D(3, xg, yg)

vis.plot_frame(4, xg, yg)
vis.quiver(4, xg, yg, qx, qy, scale=1)
vis.show()
```

```
heat_cond(Lx=1, Ly=1, Nx=4, Ny=4)
```



```
import ipywidgets as widgets
```

```
widgets.interact(heat_cond, Lx = (1,3,1), Ly = (1,3,1), Nx = (4, 8, 1), Ny = (4,
```

```
<function __main__.heat_cond(Lx, Ly, Nx, Ny)>
```




5 Conducción de calor No estacionaria.

Objetivo.

Resolver la ecuación de calor no estacionaria en 2D usando un método explícito.

[HeCompA - 02_cond_calor](#) 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

6 Conducción de calor

Jean-Baptiste Joseph Fourier fue un matemático y físico francés que ejerció una fuerte influencia en la ciencia a través de su trabajo *Théorie analytique de la chaleur*. En este trabajo mostró que es posible analizar la conducción de calor en cuerpos sólidos en términos de series matemáticas infinitas, las cuales ahora llevan su nombre: *Séries de Fourier*. Fourier comenzó su trabajo en 1807, en Grenoble, y lo completó en París en 1822. Su trabajo le permitió expresar la conducción de calor en objetos bidimensionales (hojas muy delgadas de algún material) en términos de una ecuación diferencial:

$$\frac{\partial u}{\partial t} = \kappa \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

donde u representa la temperatura en un instante de tiempo t y en un punto (x, y) del plano Cartesiano y κ es la conductividad del material.

La solución a la ecuación anterior se puede aproximar usando el método de diferencias y una fórmula explícita de dicha solución es la siguiente:

$$u_{i,j}^{n+1} = u_{i,j}^n + \frac{h_t \kappa}{h^2} (u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n)$$

donde: -

$u_{i,j} = u(x_i, y_j)$, $u_{i+1,j} = u(x_{i+1}, y_j)$, $u_{i-1,j} = u(x_{i-1}, y_j)$, $u_{i,j+1} = u(x_i, y_{j+1})$, $u_{i,j-1} = u(x_i, y_{j-1})$

. - El superíndice indica el instante de tiempo, entonces el instante actual es $n = t$ y el instante siguiente es

$n + 1 = t + h_t$, con h_t el paso de tiempo. - En este ejemplo $h_x = h_y$.

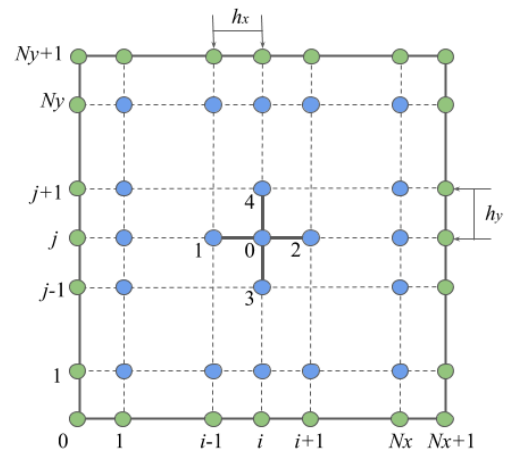
Usando esta aproximación, vamos a realizar una ejemplo de conducción de calor, pero para ello necesitamos conocer las herramientas de [numpy](#) y de [matplotlib](#).

6.1 Ejercicio 1.

Calculemos la transferencia de calor por conducción en una placa cuadrada unitaria usando el método de diferencias finitas. El problema se describe de la siguiente manera:

$$\frac{\partial u}{\partial t} = \kappa \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

| | |
|----------------------|-------------------|
| $u(x, y, t = 0) = 0$ | Condición inicial |
| $u(0, y, t) = 20$ | Condiciones |
| $u(1, y, t) = 5$ | de |
| $u(x, 0, t) = 50$ | frontera |
| $u(x, 1, t) = 8$ | |



1. Definir los parámetros físicos y numéricos del problema:

```
import numpy as np
```

```
# Parámetros físicos
k = 1.0 # Conductividad
Lx = 1.0 # Longitud del dominio en dirección x
Ly = 1.0 # Longitud del dominio en dirección y

# Parámetros numéricos
Nx = 9 # Número de incógnitas en dirección x
Ny = 9 # Número de incógnitas en dirección y
h = Lx / (Nx+1) # Espaciamiento entre los puntos de la rejilla
ht = 0.0001 # Paso de tiempo
N = (Nx + 2)* (Ny + 2) # Número total de puntos en la rejilla
```

2. Definir la rejilla donde se hará el cálculo (malla):

```
x = np.linspace(0,Lx,Nx+2) # Arreglo con las coordenadas en x
y = np.linspace(0,Ly,Ny+2) # Arreglo con las coordenadas en y
print(x)
print(y)
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

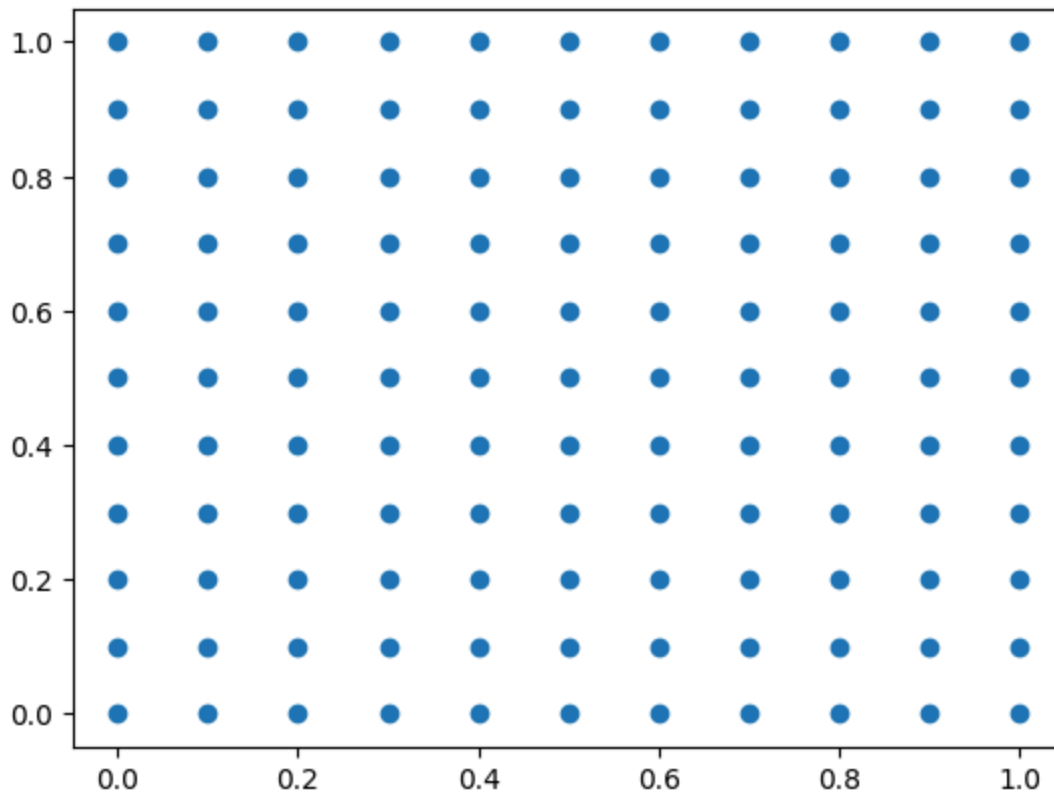
```
xg, yg = np.meshgrid(x,y) # Creamos la rejilla para usarla en Matplotlib
print(xg)
print(yg)
```

```
[[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
 [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
 [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
 [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
 [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
 [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
 [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]]
[[0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
 [0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2]
 [0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3]
 [0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4]
 [0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5]
 [0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6]
 [0.7 0.7 0.7 0.7 0.7 0.7 0.7 0.7 0.7 0.7 0.7]
 [0.8 0.8 0.8 0.8 0.8 0.8 0.8 0.8 0.8 0.8 0.8]
 [0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]]
```

```
import matplotlib.pyplot as plt
```

```
plt.scatter(xg, yg) # Graficamos la rejilla
```



3. Definir las condiciones iniciales y de frontera:

| | |
|----------------------|-------------------------------|
| $u(x, y, t = 0) = 0$ | Condición inicial |
| $u(0, y, t) = 20$ | Condiciones de frontera |
| $u(1, y, t) = 5$ | |
| $u(x, 0, t) = 50$ | |
| $u(x, 1, t) = 8$ | |

```

u = np.zeros((Nx+2, Ny+2))
#u = np.zeros(N).reshape(Nx+2, Ny+2) # Arreglo para almacenar la aproximación
print(u)
u[0,:] = 20 # Pared izquierda
u[Nx+1,:] = 5 # Pared derecha
u[:,0] = 50 # Pared inferior
u[:,Ny+1] = 8 # Pared superior
print(u)

```

```

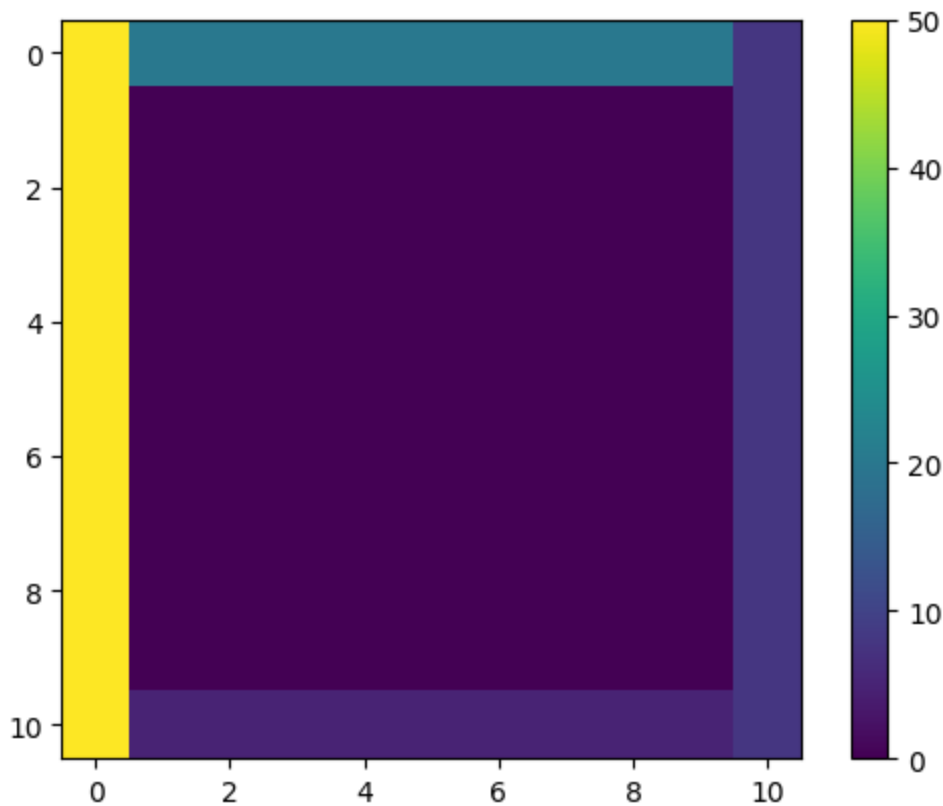
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
[[50. 20. 20. 20. 20. 20. 20. 20. 20. 20. 8.]
 [50.  0.  0.  0.  0.  0.  0.  0.  0.  0. 8.]
 [50.  0.  0.  0.  0.  0.  0.  0.  0.  0. 8.]
 [50.  0.  0.  0.  0.  0.  0.  0.  0.  0. 8.]
 [50.  0.  0.  0.  0.  0.  0.  0.  0.  0. 8.]
 [50.  0.  0.  0.  0.  0.  0.  0.  0.  0. 8.]
 [50.  0.  0.  0.  0.  0.  0.  0.  0.  0. 8.]
 [50.  0.  0.  0.  0.  0.  0.  0.  0.  0. 8.]
 [50.  0.  0.  0.  0.  0.  0.  0.  0.  0. 8.]
 [50.  0.  0.  0.  0.  0.  0.  0.  0.  0. 8.]
 [50.  5.  5.  5.  5.  5.  5.  5.  5.  5. 8.]]

```

```

f = plt.imshow(u)
plt.colorbar(f)

```



4. Implementar el algoritmo de solución:

$$u_{i,j}^{n+1} = u_{i,j}^n + \frac{h_t \kappa}{h^2} (u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n)$$

```

ht = 0.001
r = k * ht / h**2
u_new = u.copy()
tolerancia = 9.0e-1 #1.0e-3
error = 1.0
error_lista = []
while(error > tolerancia):
    for i in range(1,Nx+1):
        for j in range(1,Ny+1):
            u_new[i,j] = u[i,j] + r * (u[i+1,j] + u[i-1,j] + u[i,j+1] + u[i,j-1])
        error = np.linalg.norm(u_new - u)
        error_lista.append(error)
    # print(error)
    u[:] = u_new[:]

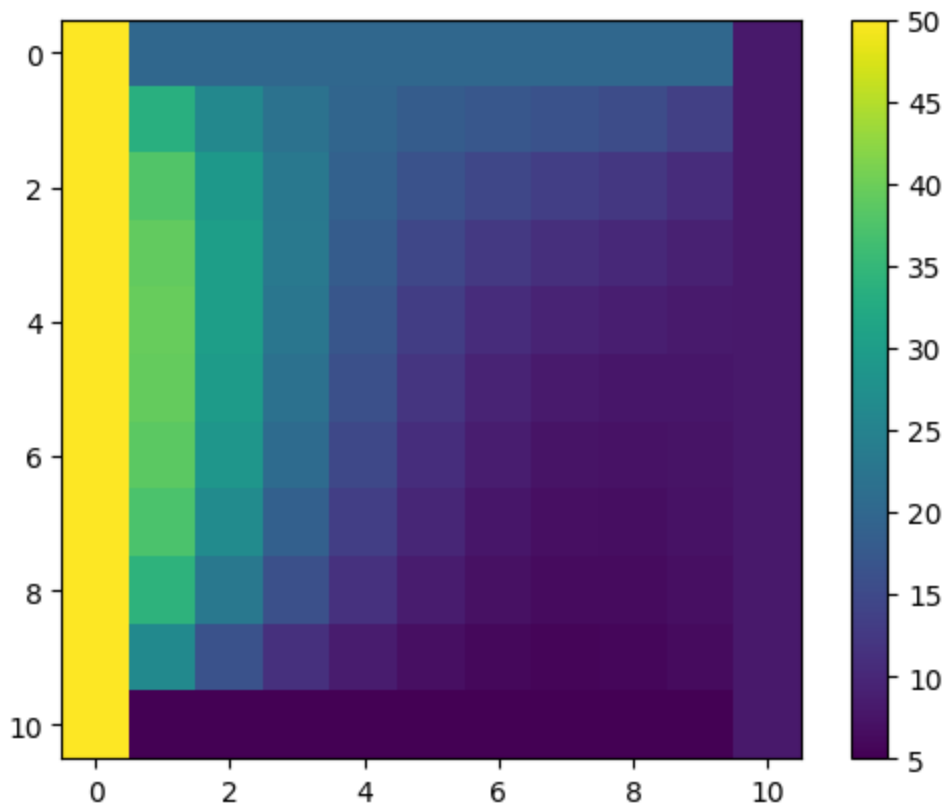
print(error_lista)

f = plt.imshow(u)
plt.colorbar(f)

```

[17.2629661414254, 13.602554907075358, 11.121464292079526, 9.370340343338656,
8.089431314598079, 7.121431307338295, 6.368158288285477, 5.766710029025608,

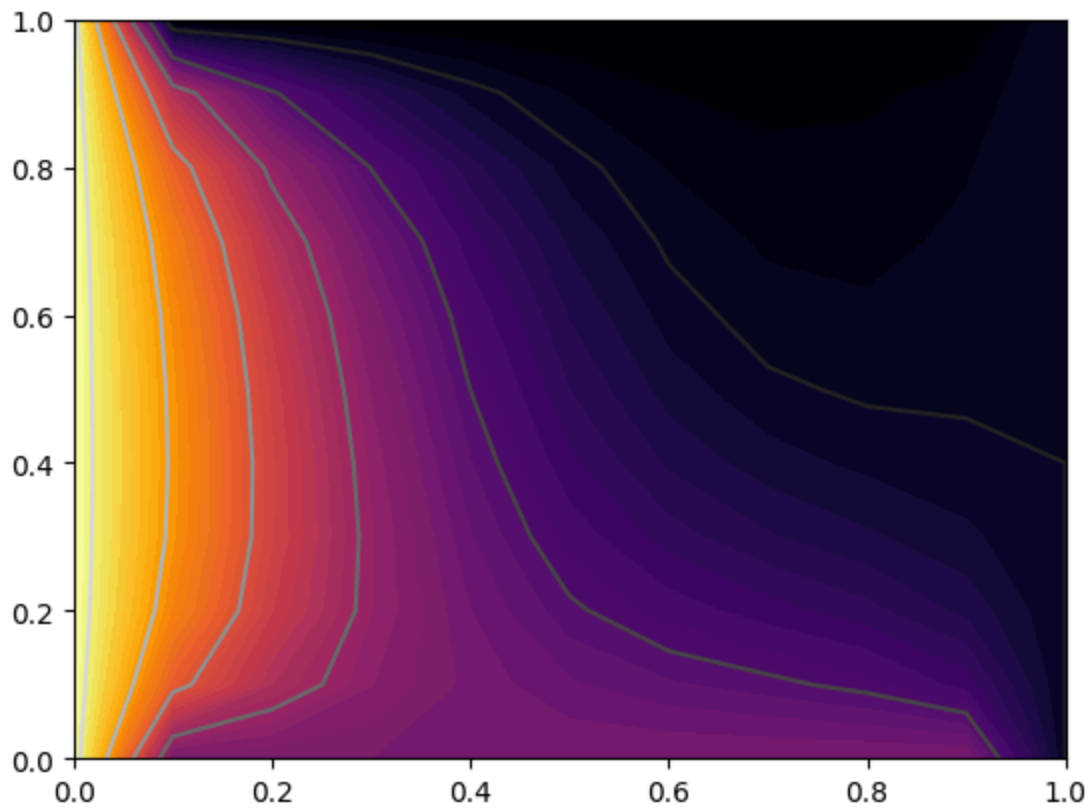
5.275727580531657, 4.867291496014421, 4.522054027452314, 4.226261372943586,
 3.9698959855402296, 3.745493517737013, 3.547373613271126, 3.3711295389631717,
 3.21328287724892, 3.0710454523767496, 2.9421521475712167, 2.824741357872151,
 2.7172679513259683, 2.6184387521297463, 2.5271638648303663, 2.4425193146763595,
 2.363717902820954, 2.290086125094306, 2.2210456430727876, 2.1560982312045556,
 2.094813422134956, 2.0368182790286324, 1.9817888683696612, 1.9294431092766995,
 1.8795347490871392, 1.8318482687809046, 1.7861945617665091, 1.7424072597326865,
 1.7003396024699928, 1.6598617667041742, 1.620858583384764, 1.5832275844672883,
 1.5468773296753073, 1.5117259715044917, 1.4777000231834065, 1.4447332996949114,
 1.4127660064863383, 1.3817439543093082, 1.3516178818527333, 1.322342870562428,
 1.293877838356635, 1.2661851009139355, 1.239229990882043, 1.2129805267782563,
 1.1874071245620865, 1.1624823458905373, 1.1381806779428072, 1.114478340447452,
 1.0913531161802335, 1.0687842017418048, 1.0467520758851223, 1.025238383054624,
 1.0042258301337272, 0.9836980946818261, 0.9636397431848914, 0.9440361580507419,
 0.9248734722567947, 0.9061385107087614, 0.8878187374976269]



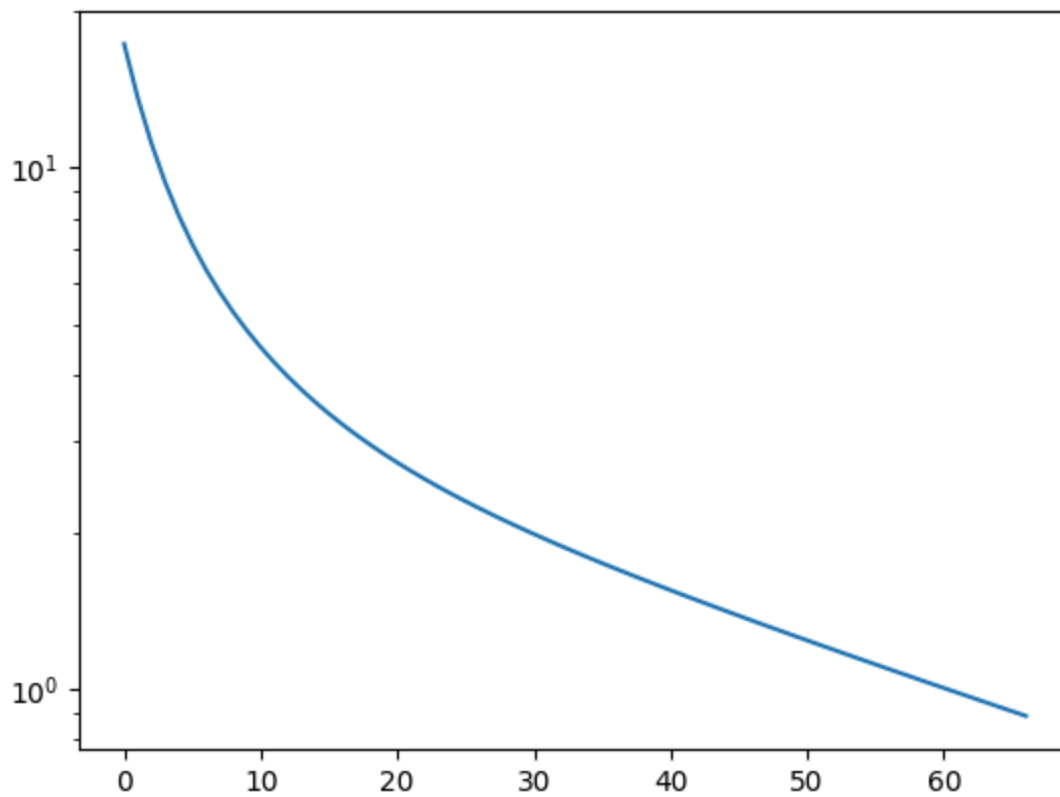
6.2 Ejercicio 2.

Realiza los siguiente gráficos de la solución anterior: 1. Contornos llenos (`contourf`) y líneas de contorno negras sobrepuestas (`contour`). 2. Almacena el error en cada iteración y gráficalo en semi-log. 3. Realiza las dos gráficas anteriores en un solo renglón.

```
plt.contour(xg, yg, u, cmap='gray', levels=5)
plt.contourf(xg, yg, u, levels=50, cmap='inferno')
```



```
plt.plot(error_lista)  
plt.yscale('log')
```



```

fig = plt.figure()

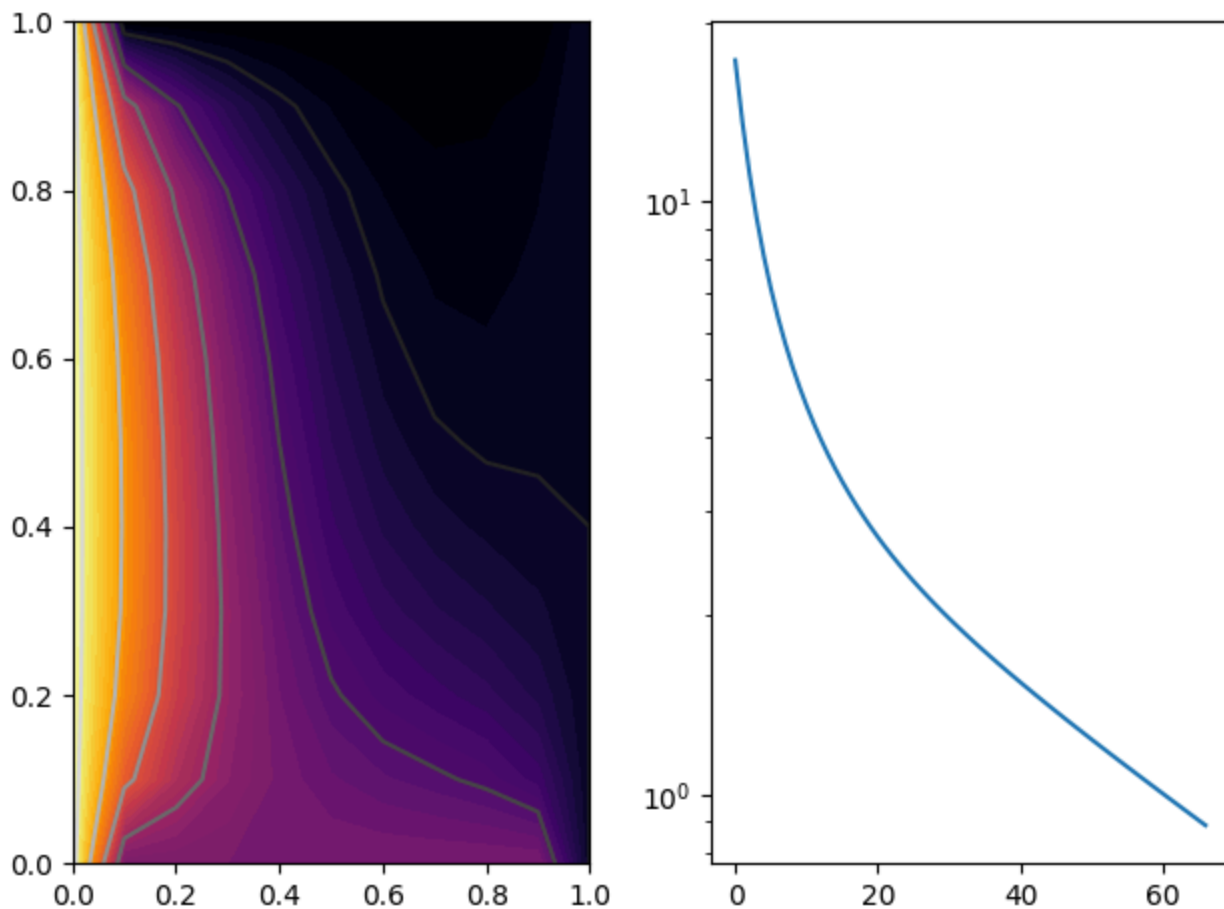
ax1 = fig.add_subplot(1, 2, 1)
ax2 = fig.add_subplot(1, 2, 2)

ax1.contour(xg, yg, u, cmap='gray', levels=5)
ax1.contourf(xg, yg, u, levels=50, cmap='inferno')

ax2.plot(error_lista)
ax2.set_yscale('log')

plt.tight_layout()

```



6.3 Flujo de calor

Fourier también estableció una ley para el flujo de calor que se escribe como:

$$\vec{q} = -\kappa \nabla u = -\kappa \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y} \right)$$

6.4 Ejercicio 3.

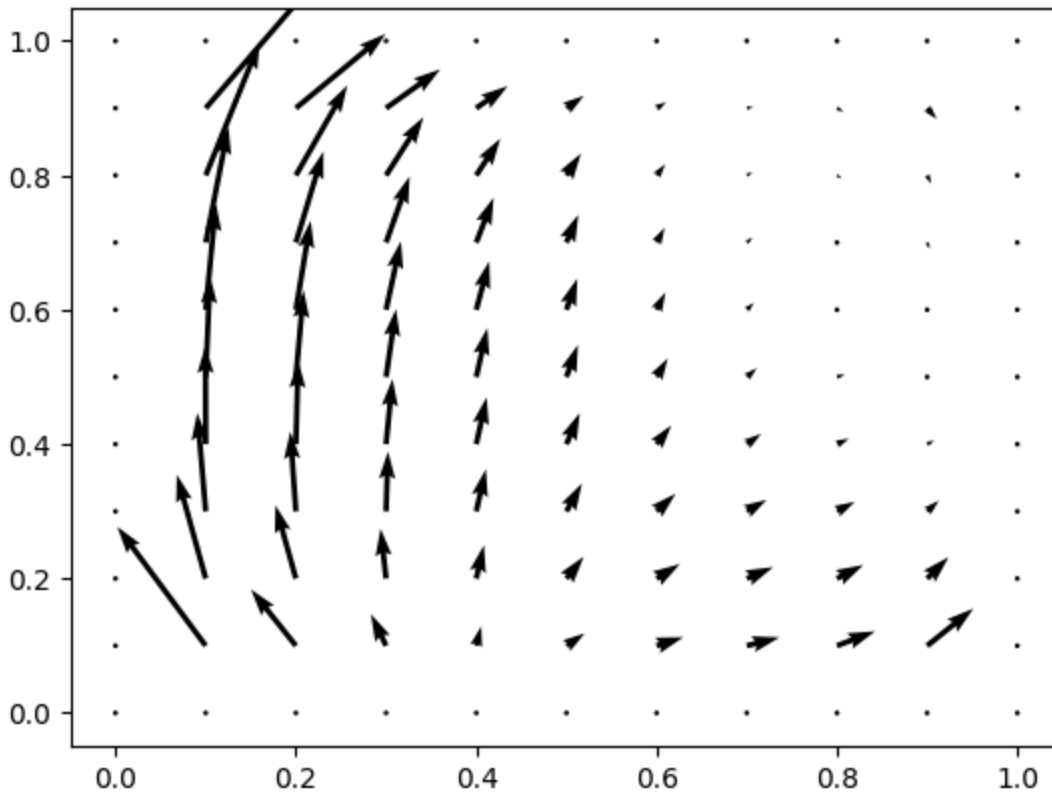
Usando la información calculada de la temperatura (almacenada en el arreglo `u`), vamos a calcular el flujo de calor usando la siguiente fórmula en diferencias:

$$\vec{q}_{i,j} = (qx_{i,j}, qy_{i,j}) = -\frac{\kappa}{2h}(u_{i+1,j} - u_{i-1,j}, u_{i,j+1} - u_{i,j-1})$$

```
qx = np.zeros((Nx+2, Ny+2))
qy = qx.copy()

s = k / 2*h
for i in range(1,Nx+1):
    for j in range(1,Ny+1):
        qx[i,j] = -s * (u[i+1,j] - u[i-1,j])
        qy[i,j] = -s * (u[i,j+1] - u[i,j-1])

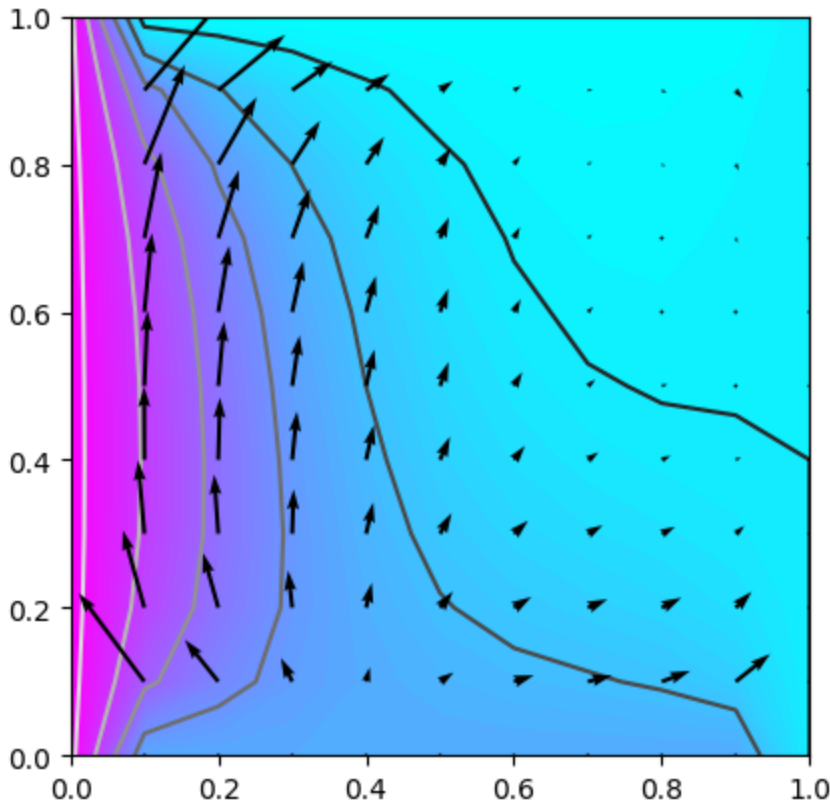
plt.quiver(xg, yg, qx, qy, scale=10, zorder=10)
```



6.5 Ejercicio 4.

Grafica el campo vectorial del flujo de calor, junto con los contornos de la temperatura (`contourf` y `contour`). Haz que tu gráfica se vea con razón de aspecto correcta de 1 por 1.

```
plt.contour(xg, yg, u, cmap='gray', levels=5)
plt.contourf(xg, yg, u, levels=50, cmap='cool', zorder=1)
plt.quiver(xg, yg, qx, qy, scale=10, zorder=10)
ax = plt.gca()
ax.set_aspect('equal')
```



7 Seguimiento de partículas

Si soltamos una partícula en un flujo, dicha partícula seguirá la dirección del flujo y delinearé una trayectoria como se muestra en la siguiente figura. Para calcular los puntos de la trayectoria debemos resolver una ecuación como la siguiente:

$$\frac{\partial \vec{x}}{\partial t} = \vec{v} \quad \text{con} \quad \vec{x}(t=0) = \vec{x}_o$$

donde $\vec{x} = (x, y)$ representa la posición de la partícula y $\vec{v} = (v_x, v_y)$ su velocidad. El método más sencillo para encontrar las posiciones de la partícula es conocido como de *Euler hacia adelante* y se escribe como:

$$\vec{x}_i^{n+1} = \vec{x}_i^n + h_t * \vec{v}_i^n$$

donde \vec{x}_i^n representa la posición de la partícula i en el instante n , h_t es el paso de tiempo y \vec{v}_i^n es la velocidad en la partícula i en el instante n .

7.1 Ejercicio 5.

Calcular y graficar las trayectorias de varias partículas usando el campo vectorial generado por el flujo de calor del ejemplo 2.

Escribimos la fórmula de *Euler hacia adelante* en componentes como sigue:

$$\begin{aligned}x_i^{n+1} &= x_i^n + h_t * vx_i^n \\y_i^{n+1} &= y_i^n + h_t * vy_i^n\end{aligned}$$

1. Definimos un punto inicial de forma aleatoria en el cuadrado unitario:

```
xo = 0.2 #np.random.rand(1)
yo = 0.5 #np.random.rand(1)
print(xo)
print(yo)
```

0.2

0.5

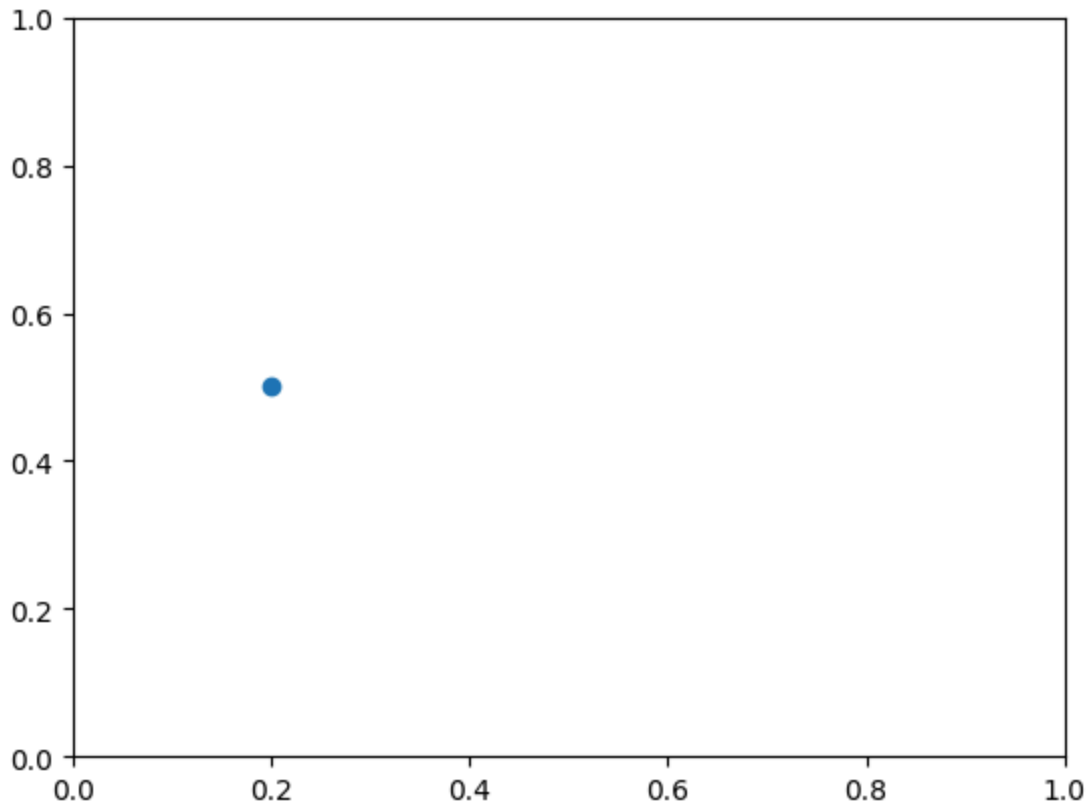
2. Definimos arreglos para almacenar las coordenadas de la trayectoria:

```
Pasos = 10
xp = np.zeros(Pasos)
yp = np.zeros(Pasos)
xp[0] = xo
yp[0] = yo
print(xp)
print(yp)
```

```
[0.2 0.  0.  0.  0.  0.  0.  0.  0.  0. ]
```

```
[0.5 0.  0.  0.  0.  0.  0.  0.  0.  0. ]
```

```
plt.plot(xp[0], yp[0], 'o-')
plt.xlim(0,1)
plt.ylim(0,1)
```



3. Implementamos el método de Euler hacia adelante:

```
# Interpolación de la velocidad
def interpolaVel(qx, qy, xpi, ypi, h):
    # localizamos la partícula dentro de la rejilla:
    li = int(xpi/h)
    lj = int(ypi/h)
    return (qx[li,lj], qy[li,lj])
```

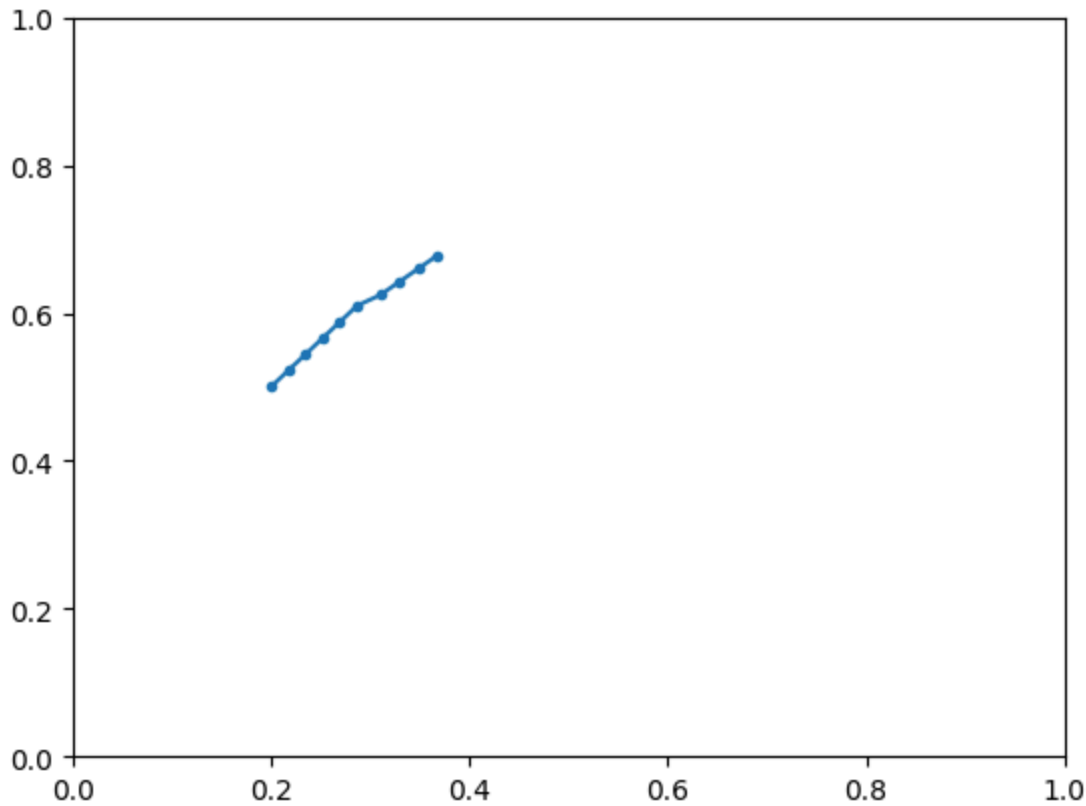
```
ht = 0.1
for n in range(1,Pasos):
    vx, vy = interpolaVel(qx, qy, xp[n-1], yp[n-1], h)
    xp[n] = xp[n-1] + ht * vx
    yp[n] = yp[n-1] + ht * vy
```

```
print(xp)
print(yp)
```

```
[0.2      0.21738397 0.23476794 0.25215191 0.26953588 0.28691984
 0.31035226 0.32940321 0.34845415 0.36750509]
[0.5      0.52197449 0.54394898 0.56592346 0.58789795 0.60987244
 0.62441928 0.64213907 0.65985885 0.67757864]
```

```
plt.plot(xp, yp, '.-')
plt.xlim(0,1)
```

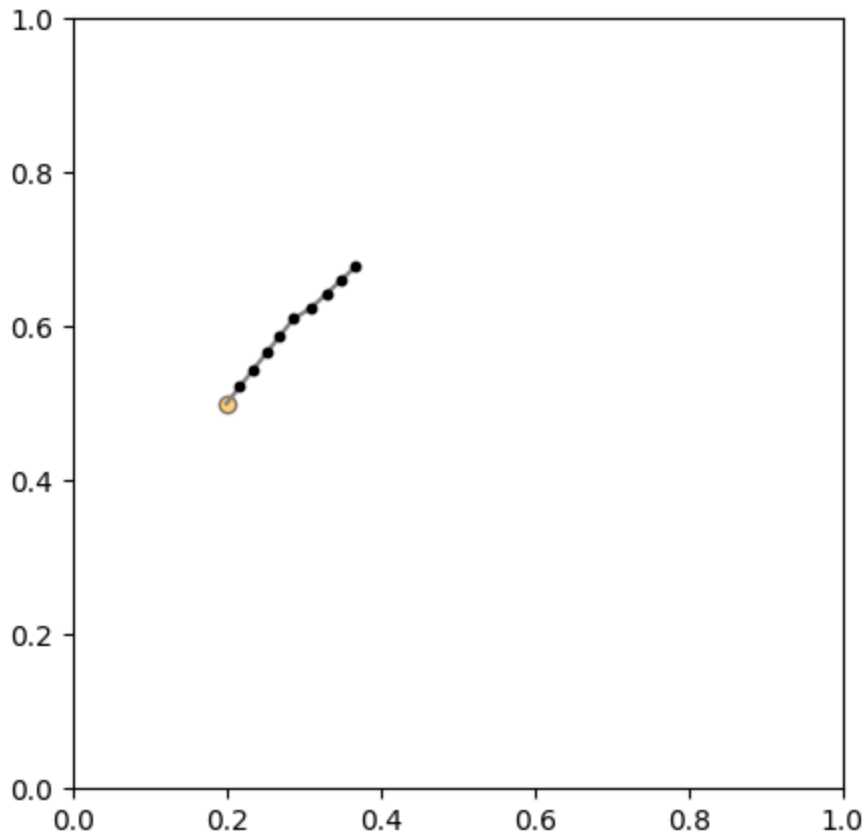
```
plt.ylim(0,1)
```



7.2 Ejercicio 6.

Dibuja la trayectoria de la siguiente manera. - El primer punto color naranja transparente y contorno negro. - Las posiciones siguientes de color negro sobre puestas sobre la trayectoria. - La trayectoria de color gris. - Verifica que la trayectoria no se salga del cuadrado unitario.

```
plt.figure(figsize=(5,5))
plt.scatter(xp[0], yp[0], c='orange', edgecolor='k', alpha=0.5)
plt.plot(xp, yp, c='gray')
plt.scatter(xp[1:], yp[1:], c='k', s=10, zorder=5)
plt.xlim(0,1)
plt.ylim(0,1)
ax = plt.gca()
ax.set_aspect('equal')
plt.savefig('trayectoria1.pdf')
```



7.3 Ejercicio 7.

Dibuja varias trayectorias que inicien en sitios diferentes.

7.4 Ejercicio 8.

Implementa una interpolación bilineal para calcular la velocidad.



8 Flujo Monofásico en 2D con SciPy

Objetivo general - El alumno resolverá la ecuación de Laplace en dos dimensiones, la cual representa el flujo monofásico incompresible en estado estable, con propiedades del fluido y del medio constantes. Mediante la solución de esta ecuación, se conocerán los tipos de condiciones de frontera y los tipos de solvers que se aplican en la solución de problemas relacionados con la Simulación Matemática de Yacimientos (SMY).

Objetivos particulares - Conocer los diferentes tipos de “solvers” para los sistemas de ecuaciones lineales. - Identificar la dirección del flujo de fluidos de acuerdo a las isobaras.

8.1 Contenido

- [1 - Implementación de SciPy](#)
 - [1.1 -](#)
 - [Ejercicio 1](#)

9 1 Implementación de SciPy

La biblioteca de scipy permite utilizar una basta cantidad de algoritmos, solvers y funciones científicas. El uso de éstas logra resolver el problema científico en mucho menos tiempo. Como alternativa al algoritmo de Thomas, se plantea el uso de los siguientes solvers: - gmres - bicgstab - spsolve

```
import numpy as np
import matplotlib.pyplot as plt
import ipywidgets as widgets
import funciones_personalizadas as fp

from scipy.sparse import lil_matrix
from scipy.sparse.linalg import spsolve
from scipy.sparse.linalg import gmres
from scipy.sparse.linalg import bicgstab
```

9.0.1 Funciones auxiliares del jupyter anterior

- ecuaciones_discretizadas
- condiciones_de_frontera_dirichlet

```
def ecuaciones_discretizadas(lx, ly, nx, ny, hx, hy):
    """
    Esta funcion genera los coeficientes de la ecuacion de flujo monofasico discr
    condiciones de frontera.
    Parametros
    -----
    lx, ly : entero o flotante.
             Longitud en el eje x e y.
```

```

nx, ny : int, int.
    Nodos en de la malla rectangular.
Retorna
-----
AP,AE,AW,AN,AS,B : ndarray.
    Arreglos en 2D para generar una matriz pentadiagonal.
"""

# se definen los arreglos en dos dimensiones
AP = np.zeros((nx,ny))
AE = np.zeros((nx,ny))
AW = np.zeros((nx,ny))
AN = np.zeros((nx,ny))
AS = np.zeros((nx,ny))
B = np.zeros((nx,ny))

for j in range (1,ny-1):
    for i in range (1,nx-1):
        AP[i][j]=2.0/hx**2.0+2.0/hy**2.0
        AE[i][j]=1.0/hx**2.0
        AW[i][j]=1.0/hx**2.0
        AN[i][j]=1.0/hy**2.0
        AS[i][j]=1.0/hy**2.0
        B[i][j]=0.0

return AP,AE,AW,AN,AS,B

```

```

def condiciones_de_frontera_dirichlet(P1, P2, P3, P4, AP, AW, AE, AN, AS, B, nx,
    """
    Esta función modifica los coeficientes y asigna las condiciones de frotera de
    """

    for i in range (1,nx-1):
        AP[i][0]=1.0
        AW[i][0]=0.0
        AE[i][0]=0.0      #Frontera sur
        AN[i][0]=0.0
        B[i][0]=P2

        AP[i][ny-1]=1.0
        AW[i][ny-1]=0.0
        AE[i][ny-1]=0.0    #Frontera norte
        AS[i][ny-1]=0.0
        B[i][ny-1]=P4

    for j in range (0,ny):
        AP[0][j]=1.0
        AE[0][j]=0.0      #Frontera Oeste
        AN[0][j]=0.0
        AS[0][j]=0.0
        B[0][j]=P1

```



```

AP[nx-1][j]=1.0
AW[nx-1][j]=0.0
AN[nx-1][j]=0.0 #Frontera Este
AS[nx-1][j]=0.0
B[nx-1][j]=P3

```

9.0.2 Uso de SciPy

```

def flujo_monofasico_2d_scipy(nx, ny, lx, ly):
    """
    Esta función resuelve el problema de flujo monofásico en 2D con ayuda de la 1
    """

    """
    |---P4(NORTE)---|
    |                 |
    P1(OESTE)         P3(ESTE)
    |                 |
    |---P2(SUR)----|
    """

    P1, P2, P3, P4 = 1000.0, 500.0, 500.0, 500.0
    Press= np.zeros((nx,ny))

    #Generación de malla para graficar
    x = np.linspace(0, lx, nx)
    y = np.linspace(0, ly, ny)
    malla_x, malla_y = np.meshgrid(x, y)

    # se calcula el espaciamiento entre nodos
    hx = lx/(nx-1)
    hy = ly/(ny-1)

    # Llamar a la función para obtener los coeficientes
    AP, AW, AE, AN, AS, B = ecuaciones_discretizadas(lx, ly, nx, ny, hx, hy)

    #Aplicar fronteras de primera clase (Dirichlet)
    condiciones_de_frontera_dirichlet(P1, P2, P3, P4, AP, AW, AE, AN, AS, B, nx,

    n = nx*ny
    A = np.zeros((n,n))
    Press1 = np.zeros(n)
    B1 = np.zeros(n)

    for j in range (0,ny):
        for i in range (0,nx):
            k=i+nx*j

```

```

A[k][k] = AP[i][j]
B1[k] = B[i][j]

if j < ny-1: A[k][k+nx] = -AN[i][j]
if i < nx-1: A[k][k+1] = -AE[i][j]
if i > 0: A[k][k-1] = -AW[i][j]
if j > 0: A[k][k-nx] = -AS[i][j]

A1 = lil_matrix(A)
A1 = A1.tocsr()

Press1 = gmres(A1, B1, tol=1.0E-07, restart = 2000) #scipy.sparse.linalg.gmr
#Press1 = bicgstab(A1, B1, tol=1.0E-05) #scipy.sparse.linalg.gmres(A, b, x0=
#Press1 = spsolve(A1, B1) #scipy.sparse.linalg.gmres(A, b, x0=None, tol=1e-0

for j in range (0,ny):
    for i in range (0,nx):
        k=i+j*nx
        Press[i][j]=Press1[0][k] #Cuando se utilizan los algoritmos del su
        #Press[i][j]=Press1[k] #Cuando se utiliza spsolve (descomposición

# graficar
fp.graficar_isobaras_presion_y_campo_velocidad(nx, ny, hx, hy, malla_x, malla

```

```

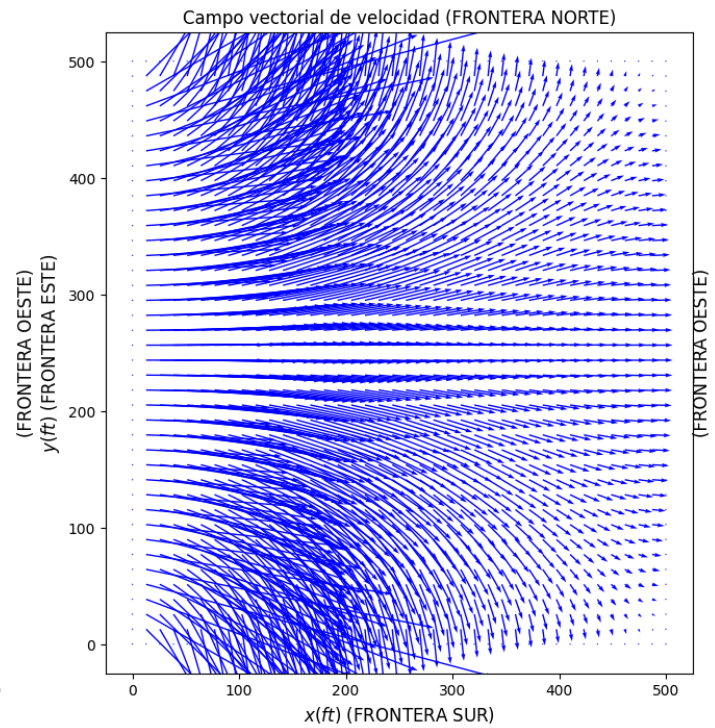
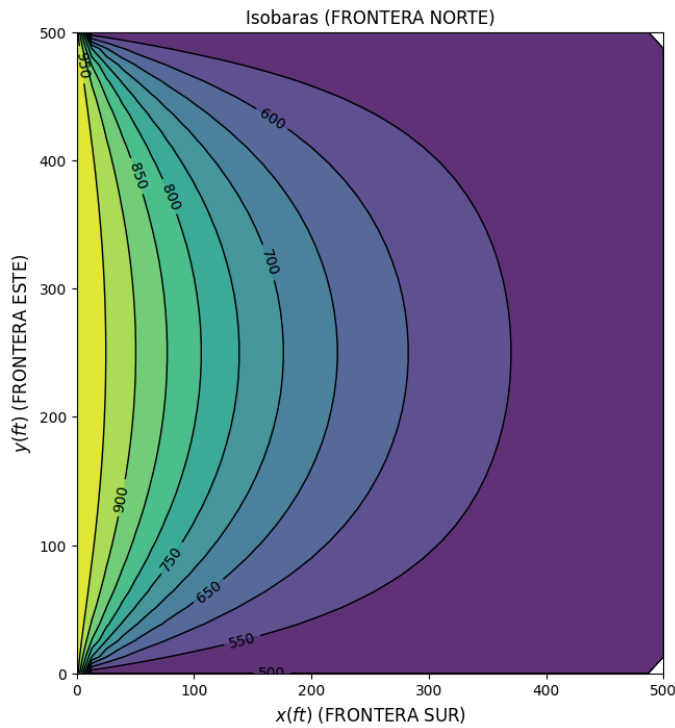
nx, ny = 40, 40
lx, ly = 500, 500

```

```

%%time
flujo_monofasico_2d_scipy(nx, ny, lx, ly)

```



CPU times: user 871 ms, sys: 512 ms, total: 1.38 s

Wall time: 859 ms

9.1 Ejercicio 3 - Uso de SciPy

En el script `widgets.interact(flujo_monofasico_2d_scipy_animacion, presion_cambiante = widgets.Play(min=600, max=1000))`:

1.- Cambia el rango de presiones desde 750 psi a 1,400 psi.

En el script `flujo_monofasico_2d_scipy_animacion(presion_cambiante)`:

2.- Asigna la variable --> `presion_cambiante` a `P3` y el resto de presiones con el valor de 600 psi.

3.- Comenta las líneas de código pertenecientes al solver `gmres` --> `Press1 = gmres(A1, B1, tol=1.0E-07, restart = 2000)` y --> `Press[i][j]=Press1[0][k]`

4.- Descomenta las líneas de código pertenecientes a `spsolve` --> `Press1 = spsolve(A1, B1)` y --> `Press[i][j]=Press1[k]`

5.- Ejecuta las celdas de código modificadas y visualiza los resultados

7 Flujo Monofásico en 2D con Algoritmo de Thomas

Objetivo general - El alumno resolverá la ecuación de Laplace en dos dimensiones, la cual representa el flujo monofásico incompresible en estado estable, con propiedades del fluido y del medio constantes. Mediante la solución de esta ecuación, se conocerán los tipos de condiciones de frontera y los tipos de solvers que se aplican en la solución de problemas relacionados con la Simulación Matemática de Yacimientos (SMY).

Objetivos particulares - Conocer e identificar los tipos de condiciones a la frontera utilizadas en SMY. - Identificar la dirección del flujo de fluidos de acuerdo a las isobaras.

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

7.1 Contenido

- [1 - Resolución de un problema de ingeniería mediante matemática computacional](#)
- [2 - Condiciones de frontera](#)
 - [2.1 - Primera clase \(Neumann\)](#)
 - [2.2 - Segunda clase \(Dirichlet\)](#)
 - [Ejercicio 1.](#)
- [3 - Resolución del problema para flujo monofasico en 2D](#)
 - [3.1 - Algoritmo de Thomas.](#)
 - [Ejercicio 2](#)

1. Resolución de un problema de ingeniería mediante matemática computacional Se realiza la siguiente metodología: 1. Definir un modelo físico conceptual 2. Definir el modelo matemático 3. Definir el modelo numérico 4. Definir el modelo computacional (algoritmo de solución)

7.1.1 1.1 Modelo físico conceptual

Se inyecta un fluido a presión constante (**1,000 psi**) y se produce a presión constante (**500 psi**), en un medio poroso bidimensional homogéneo (de longitud $L_x = 1,000 \text{ ft}$ y $L_y = 500 \text{ ft}$ con permeabilidad y porosidad constantes), así mismo las propiedades del fluido no dependen de la presión, adicionalmente el medio poroso se encuentra saturado completamente de este fluido, por lo que se desea conocer el gradiente de presión en estado permanente.

1. No se consideran los efectos de la fuerza de gravedad
2. No hay fuentes ni sumideros
3. El fluido es incompresible y la viscosidad constante
4. El medio está inicialmente saturado del fluido
5. Ya se ha llegado al estado estable o permanente
6. Las permeabilidades en dirección x y y son iguales e invariantes en el espacio y tiempo

Partiendo de la ecuacion general de balance de materia para flujo monofásico (de cualquier fluido α) en medios porosos se tiene:

$$\frac{\partial \phi \rho_\alpha}{\partial t} - \nabla \left(\rho_\alpha \frac{k}{\mu} (\nabla \rho_\alpha - \rho_\alpha \delta \nabla z) \right) = q_\alpha$$

7.1.2 1.2 y 1.3 Modelo matemático y numérico

Tomando en cuenta las consideraciones del modelo físico se llega a:

$$\frac{\partial^2 \rho_\alpha}{\partial x^2} + \frac{\partial^2 \rho_\alpha}{\partial y^2} = 0$$

Discretizando la ecuación anterior en una malla bidimensional se llega a:

$$AP_{i,j}p_i = AE_{i,j}p_{i+1,j} + AW_{i,j}p_{i-1,j} + AN_{i,j}p_{i,j+1} + AS_{i,j}p_{i,j-1} + B_{i,j}$$

$$i = 1, 2, 3, 4, \dots, nx - 2$$

$$j = 1, 2, 3, 4, \dots, ny - 2$$

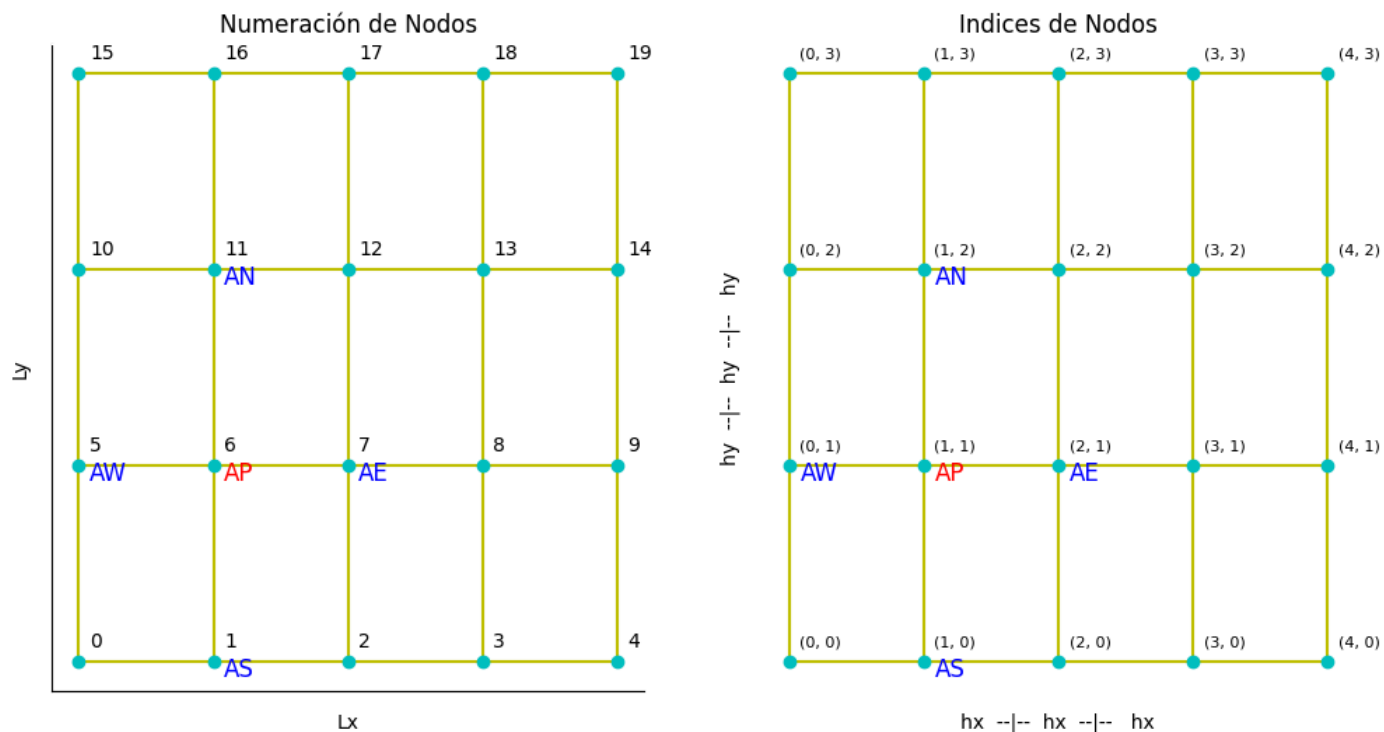
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import ipywidgets as widgets
import funciones_personalizadas as fp
```

```
from matplotlib import animation, cm
from IPython.display import HTML
```

```
# Considere una malla rectangular de 1,000 ft x 500 ft, en el eje x e y, respecti
```

```
nodos_x, nodos_y = 5, 4
longitud_x, longitud_y = 1000, 500
nx_a_discretizar, ny_a_discretizar = 1,1
```

```
fp.discretizacion_en_malla_rectangular(longitud_x, longitud_y, nodos_y, nodos_x,
```



Nodo discretizado:

```

nodos_x, nodos_y = 5, 4
nodo_a_discretizar = 1

```

```
fp.generar_matriz(nodos_x, nodos_y, nodo_a_discretizar)
```

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | AP | AE | 0 | 0 | 0 | AN | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | AW | AP | AE | 0 | 0 | 0 | AN | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | AW | AP | AE | 0 | 0 | 0 | AN | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | AW | AP | AE | 0 | 0 | 0 | AN | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | AW | AP | 0 | 0 | 0 | 0 | AN | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | AS | 0 | 0 | 0 | 0 | AP | AE | 0 | 0 | 0 | AN | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | AS | 0 | 0 | 0 | AW | AP | AE | 0 | 0 | 0 | AN | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | AS | 0 | 0 | 0 | AW | AP | AE | 0 | 0 | 0 | AN | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | AS | 0 | 0 | 0 | AW | AP | AE | 0 | 0 | 0 | AN | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | AS | 0 | 0 | 0 | AW | AP | 0 | 0 | 0 | 0 | AN | 0 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | AS | 0 | 0 | 0 | 0 | AP | AE | 0 | 0 | 0 | AN | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | AS | 0 | 0 | 0 | AW | AP | AE | 0 | 0 | 0 | AN | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | AS | 0 | 0 | 0 | AW | AP | AE | 0 | 0 | 0 | AN | 0 | 0 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | AS | 0 | 0 | 0 | AW | AP | AE | 0 | 0 | 0 | AN | 0 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | AS | 0 | 0 | 0 | AW | AP | 0 | 0 | 0 | 0 | AN |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | AS | 0 | 0 | 0 | 0 | AP | AE | 0 | 0 | 0 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | AS | 0 | 0 | 0 | AW | AP | AE | 0 | 0 |
| 17 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | AS | 0 | 0 | 0 | AW | AP | AE | 0 |
| 18 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | AS | 0 | 0 | 0 | AW | AP | AE |
| 19 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | AS | 0 | 0 | 0 | AW | AP |

7.1.3 Discretización de las ecuaciones

Aplicando diferencias finitas centradas

$$\frac{\partial^2 \rho_\alpha}{\partial x^2} + \frac{\partial^2 \rho_\alpha}{\partial y^2} = \frac{P_{i+1,j-1} - 2P_{i,j} + P_{i-1,j}}{hx^2} + \frac{P_{i,j+1} - 2P_{i,j} + P_{i,j-1}}{hy^2} = 0$$

Despejando $P_{i,j}$ (donde P es el valor incognita)

$$P_{i,j} \left(\frac{2}{hx^2} + \frac{2}{hy^2} \right) = \frac{P_{i+1,j}}{hx^2} + \frac{P_{i-1,j}}{hx^2} + \frac{P_{i,j+1}}{hy^2} + \frac{P_{i,j-1}}{hy^2}$$

La ecuación anterior se puede reescribir como:

$$APP_{i,j} = AEP_{i+1,j} + AWP_{i-1,j} + ANP_{i,j+1} + ASP_{i,j-1} + B_{i,j}$$

Donde :

$$AP = \left(\frac{2}{hx^2} + \frac{2}{hy^2} \right), AE = \frac{1}{hx^2}, AW = \frac{1}{hx^2}, AN = \frac{1}{hy^2}, AS = \frac{1}{hy^2},$$

```
def ecuaciones_discretizadas(lx, ly, nx, ny, hx, hy):
```

```
    """
```

```
    Esta funcion genera los coeficientes de la ecuacion de flujo monofasico discr
    condiciones de frontera.
```

```
    Parametros
```

```
    -----
```

```
    lx, ly : entero o flotante.
```

```
        Longitud en el eje x e y.
```

```
    nx, ny : int, int.
```

```
        Nodos en de la malla rectangular.
```

```
    Retorna
```

```
    -----
```

```
    AP,AE,AW,AN,AS,B : ndarray.
```

```
        Arreglos en 2D para generar una matriz pentadiagonal.
```

```
    """
```

```
    # se definen los arreglos en dos dimensiones
```

```
    AP = np.zeros((nx,ny))
```

```
    AE = np.zeros((nx,ny))
```

```

AW = np.zeros((nx,ny))
AN = np.zeros((nx,ny))
AS = np.zeros((nx,ny))
B = np.zeros((nx,ny))

for j in range(1,ny-1):
    for i in range(1,nx-1):
        AP[i][j]=2.0/hx**2.0+2.0/hy**2.0
        AE[i][j]=1.0/hx**2.0
        AW[i][j]=1.0/hx**2.0
        AN[i][j]=1.0/hy**2.0
        AS[i][j]=1.0/hy**2.0
        B[i][j]=0.0

return AP,AE,AW,AN,AS,B

```

2. Codiciones de frontera De acuerdo con la SEG Wiki, las condiciones de frontera se aplican en el término integral superficial de la ecuación integral de un campo y consiste en valores prescritos de la solución y de las derivadas de la solución en la frontera. Cuando los valores de campo se especifican las condiciones se denominan **condiciones de Dirichlet**. Cuando se especifican las derivadas las condiciones se llaman **condiciones de Neumann**.

2.1 Primera clase o Dirichlet Esta condición indica una funcion conocida que varia en el tiempo o un escalar dado (invariante en el tiempo) en las fronteras, es decir:

$$p_1 = f(t, y) \quad o \quad p_1 = cte_1 = 1,000psi$$

$$p_2 = f(t, x) \quad o \quad p_1 = cte_2 = 500psi$$

$$p_3 = f(t, y) \quad o \quad p_1 = cte_3 = 500psi$$

$$p_4 = f(t, x) \quad o \quad p_1 = cte_4 = 500psi$$

Implementación: 1. Identificar las fronteras y los puntos de la ecuacion discreta donde se implementará la condición 2. Llevar a cabo las sustituciones pertinentes en cada ecuación discreta, para este caso:

$$AP_{i,j}P_i = AEP_{i+1,j} + AWP_{i-1,j} + ANP_{i,j+1} + ASP_{i,j-1} + B_{i,j}$$

Considerando una malla con **nx = 5x** y **ny = 4**, se tiene la siguiente figura: 

7.1.3.1 Para la frontera SUR:

Tomando en cuenta la figura anterior, se puede apreciar que no existe relacion con la presión $Sur(i, j - 1)$, por lo tanto, la ecuación se reduce a:

$$AP_{i,j}P_i = AEP_{i+1,j} + AWP_{i-1,j} + ANP_{i,j+1} + B_{i,j}$$

En este caso el valor de presión está definido, por lo que el sistema de ecuaciones de reduce a:

$$AP_{i,j}P_i = B_i$$

$$para \quad i = 0, 1, 2, 3, 4 \quad j = 0$$

$$\text{donde } AP_{i,j} = 1 \quad y \quad B_{i,j} = Cte_2$$

Todos los demás coeficientes se asignan a cero

Para el resto de las fronteras se utiliza la misma lógica

```
def condiciones_de_frontera_dirichlet(P1, P2, P3, P4, AP, AW, AE, AN, AS, B, nx,
    """
    Esta función modifica los coeficientes y asigna las condiciones de frontera de
    """
    for i in range(1,nx-1):
        AP[i][0]=1.0
        AW[i][0]=0.0
        AE[i][0]=0.0      #Frontera sur
        AN[i][0]=0.0
        B[i][0]=P2

        AP[i][ny-1]=1.0
        AW[i][ny-1]=0.0
        AE[i][ny-1]=0.0    #Frontera norte
        AS[i][ny-1]=0.0
        B[i][ny-1]=P4

    for j in range(0,ny):
        AP[0][j]=1.0
        AE[0][j]=0.0      #Frontera Oeste
        AN[0][j]=0.0
        AS[0][j]=0.0
        B[0][j]=P1

        AP[nx-1][j]=1.0
        AW[nx-1][j]=0.0
        AN[nx-1][j]=0.0    #Frontera Este
        AS[nx-1][j]=0.0
        B[nx-1][j]=P3
```

2.2 Fronteras de segunda clase Neumann (No flujo o frontera cerrada) Éstas especifican la derivada normal a la frontera para ser cero o constante.

$$p_1 = f(t, y) \quad o \quad p_1 = cte_1 = 1,000psi$$

$$\frac{dp}{dy}\bigg|_2 = f(t, x) \quad o \quad \frac{dp}{dy}\bigg|_2 = 0.0$$

$$p_3 = f(t, y) \quad o \quad p_1 = cte_3 = 500psi$$

$$\frac{dp}{dy}\bigg|_4 = f(t, x) \quad o \quad \frac{dp}{dy}\bigg|_4 = 0.0$$

Discretización:

Frontera Sur:

$$\left. \frac{dp}{dy} \right|_2 = 0.0$$

Sustituyendo una diferencia adelantada:

$$\frac{P_{i,j+1} - P_{i,j}}{hy} = 0$$

$$P_{i,j+1} = P_{i,j}$$

Frontera Norte:

$$\left. \frac{dp}{dy} \right|_2 = 0.0$$

Sustituyendo una diferencia atrasada:

$$\frac{P_{i,j} - P_{i,j-1}}{hy} = 0$$

$$P_{i,j} = P_{i,j-1}$$

```
def condiciones_de_frontera_neumann(x, P1, P2, P3, P4, AP, AW, AE, AN, AS, B, nx,
    """
    Esta función modifica los coeficientes de la matriz y asigna las condiciones
    """

    for i in range(1, nx-1):
        AP[i][0]=1.0
        AW[i][0]=0.0
        AE[i][0]=0.0      #Frontera sur
        AN[i][0]=1.0
        B[i][0]=0

        AP[i][ny-1]=1.0
        AW[i][ny-1]=0.0
        AE[i][ny-1]=0.0    #Frontera norte
        AS[i][ny-1]=1.0
        B[i][ny-1]=0.0

    for j in range(0,ny):
        AP[0][j]=1.0
        AE[0][j]=0.0      #Frontera Oeste
        AN[0][j]=0.0
        AS[0][j]=0.0
        B[0][j]=P1

        AP[nx-1][j]=1.0
```

```

AW[nx-1][j]=0.0
AN[nx-1][j]=0.0 #Frontera Este
AS[nx-1][j]=0.0
B[nx-1][j]=P3

```

3. Resolución del problema para flujo monofásico en 2D

Los pasos para resolver el problema de flujo monofásico en 2D se presentan en el siguiente algoritmo: - Definir las variables del problema (nodos, longitudes, etc) - Implementar las ecuaciones discretizadas - Aplicar las condiciones de frontera - **Resolver la matriz con algún solver**

A continuación se mostraran una comparativa entre el algoritmo de Thomas y las alternativas de SciPy.

3.1 Solución mediante el Algoritmo de Thomas

```

# variables del problema
nx, ny = 10, 10
lx, ly = 500, 500

#Generación de malla para graficar
x = np.linspace(0, lx, nx)
y = np.linspace(0, ly, ny)
malla_x, malla_y = np.meshgrid(x, y)

```

```

def flujo_monofasico_2d_thomas(i):
    """
    Esta funcion aplica el algoritmo de Thomas para resolver el problema de flujo
    """

    """
    |---P4(NORTE)---|
    |                 |
    P1(OESTE)         P3(ESTE)
    |                 |
    |----P2(SUR)----|

    donde P es presión (psi)
    """
    P1 = 700+(i*10)
    P2, P3, P4 = 500.0, 500.0, 500.0
    Press = np.zeros((nx,ny))

    # se calcula el espaciamiento entre nodos
    hx = lx/(nx-1)
    hy = ly/(ny-1)

    # Llamar a la función para obtener los coeficientes
    AP, AW, AE, AN, AS, B = ecuaciones_discretizadas(lx, ly, nx, ny, hx, hy)

    """

```

Ejercicio 1 y Ejercicio 2

Modificar -----.

v

"""

#Aplicar condiciones de frontera

condiciones_de_frontera_dirichlet(P1, P2, P3, P4, AP, AW, AE, AN, AS, B, nx,

Resolver mediante Thomas

iteramax, eps = 3000, 1.0E-04

fp.algoritmo_thomas_2D(nx, ny, AP, AE, AW, AN, AS, B, Press, iteramax, eps)

u = np.zeros((nx,ny))

v = np.zeros((nx,ny))

for j in range (1,ny-1):

for i in range (1,nx-1):

u[i][j]=-(Press[i+1][j]-Press[i-1][j])/(2*hx) #Discretizamos por DFC

v[i][j]=-(Press[i][j+1]-Press[i][j-1])/(2*hy)

return Press,u,v

Press,u,v = flujo_monofasico_2d_thomas(1)

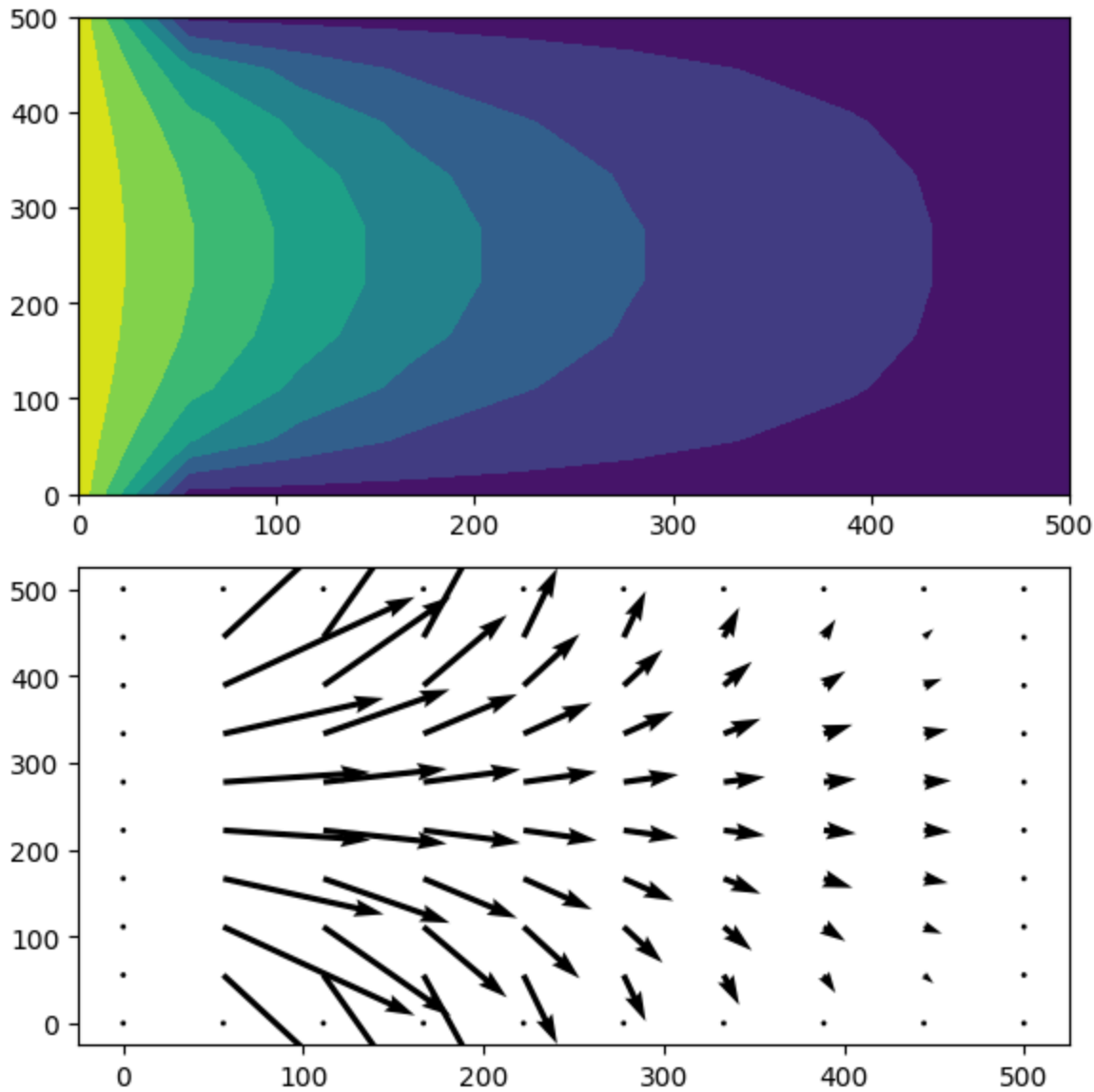
fig, ax = plt.subplots(2,1, figsize=(6,6))

ax[0].contourf(malla_x, malla_y, Press.T, cmap='viridis')

ax[1].quiver(malla_x, malla_y, u.T,v.T)

plt.tight_layout()

plt.show()



7.2 Ejercicio 1 - uso de condiciones de frontera tipo Neumann

En el script `flujo_monofasico_2d_thomas(i)`:

- 1.- Asigna 1,000 psi a la variable P3 y el resto de presiones en 500 psi
- 2.- Sustituye la función que ejecuta las condiciones de frontera de primera clase (Dirichlet) e implementa las condiciones de segunda clase (Neumann). =>
`condiciones_de_frontera_neumann(x, P1, P2, P3, P4, AP, AW, AE, AN, AS, B, nx, ny)`
- 3.- Ejecuta las celdas de código modificadas y visualiza los resultados

7.3

7.4 Ejercicio 2 - condiciones de frontera mixtas (Neumann + Dirichlet)

En el script **condiciones_de_frontera_mixtas(x, P1, P2, P3, P4, AP, AW, AE, AN, AS, B, nx, ny)** (se ubica al final del ejercicio):

- 1.- Copia el contenido de la función => `condiciones_de_frontera_neumann(x, P1, P2, P3, P4, AP, AW, AE, AN, AS, B, nx, ny)`
- 2.- En la frontera Norte asigna el código que aplica las condiciones de primera clase (Dirichlet)

En el script **flujo_monofasico_2d_thomas(lx, ly, nx, ny)**:

- 3.- Sustituye la función `condiciones_de_frontera_neumann(x, P1, P2, P3, P4, AP, AW, AE, AN, AS, B, nx, ny)` por => `condiciones_de_frontera_mixtas(x, P1, P2, P3, P4, AP, AW, AE, AN, AS, B, nx, ny)`
- 4.- Modifica los nodos de la malla a 40 en nx y 40 en ny
- 4.- Ejecuta las celdas de código modificadas y visualiza los resultados


7.4.1 Notas:

En el ejercicio 2, al aumentar a 40 el número de nodos de la malla, el tiempo de cómputo aumenta significativamente. Para disminuir este tiempo se propone el uso de SciPy...

1 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:

\$

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

\$

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

1.0.1 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)
```

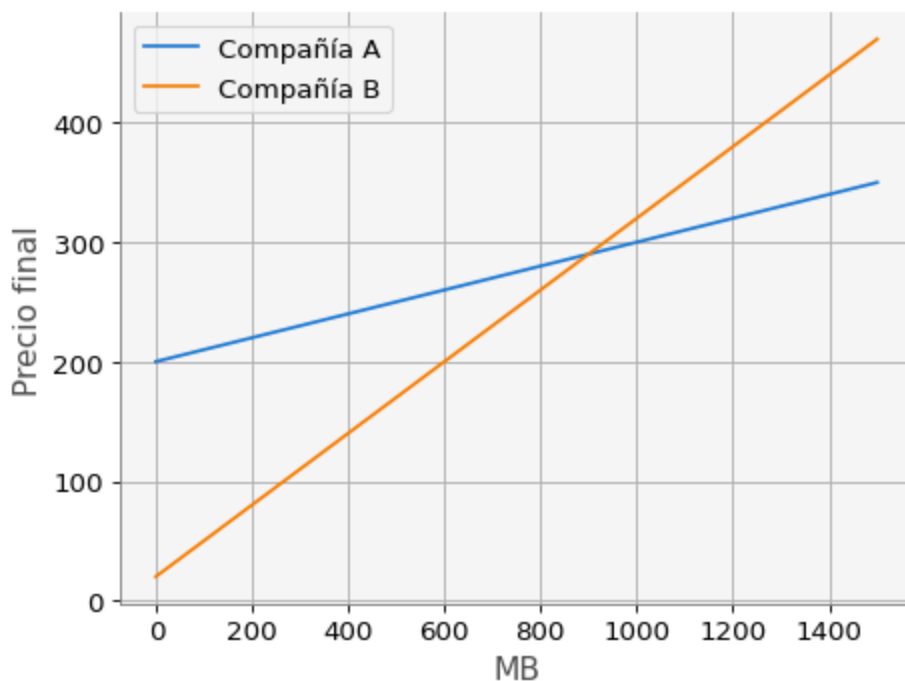
1 | Tu resultado es correcto.

```
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.

1.0.2 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.

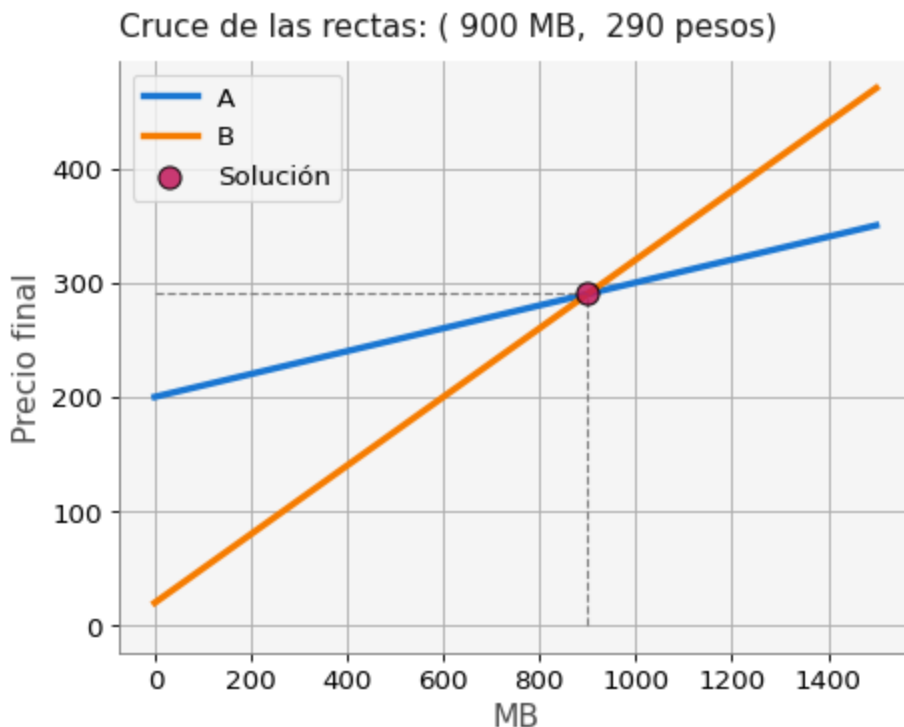
Si todo se hizo correctamente, el siguiente código debe graficar las rectas de las dos compañías y en el punto donde se cruzan

```
# 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()
```





2 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
```

3 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:

3.1 Definición y gráfica de las rectas

3.2 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

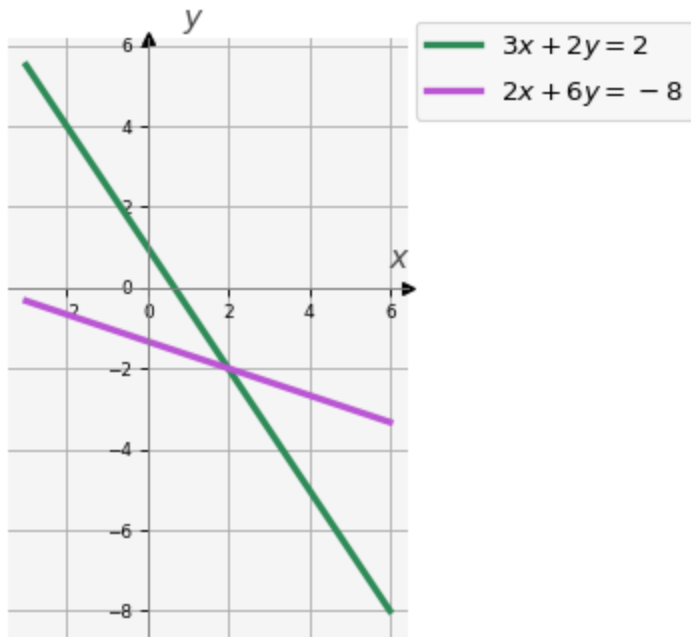
#file_answer('1', m2, 'm2 incorrecta revisa el valor del parámetro.')
#file_answer('2', b2, 'b2 incorrecta revisa el valor del parámetro.')
#file_answer('3', y2, 'y2 no está definida correctamente.')
#### END SOLUTION
```

```
#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



3.3 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:

3.4 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]]
```

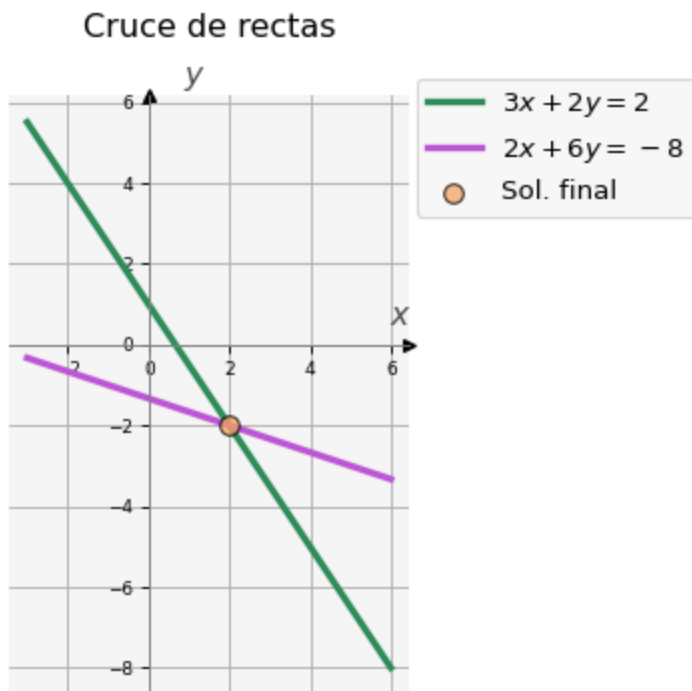
Vector b :

```
[[ 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.

4 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*.

4.1 Algoritmo Jacobi.

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



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 $error = \|\mathbf{x}^k - \mathbf{x}\|$ 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 $error = ||\mathbf{x}^k - \mathbf{x}^{k-1}||$. En los ejemplos que siguen si tenemos la solución exacta, por lo que haremos la comparación con ella.

4.2 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
```

4.3 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=

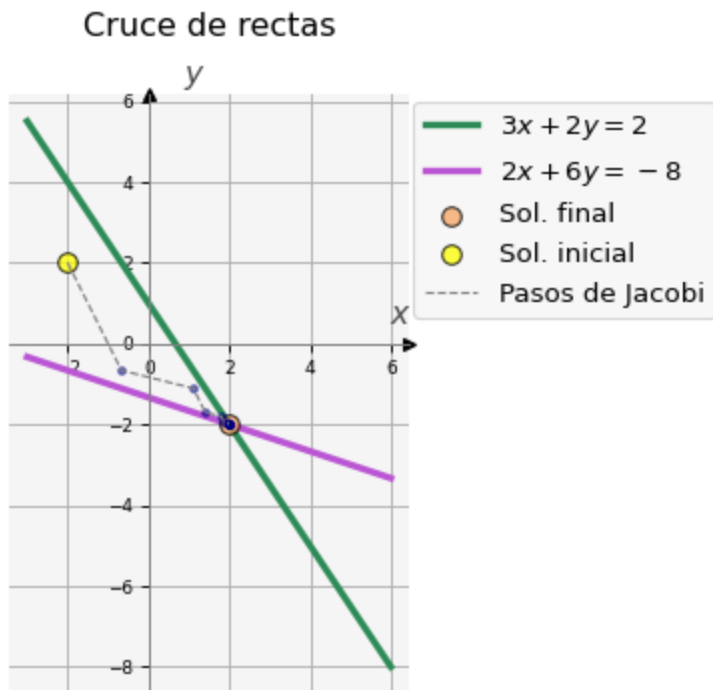
# Graficamos los pasos
```

```

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()

```



4.4 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:

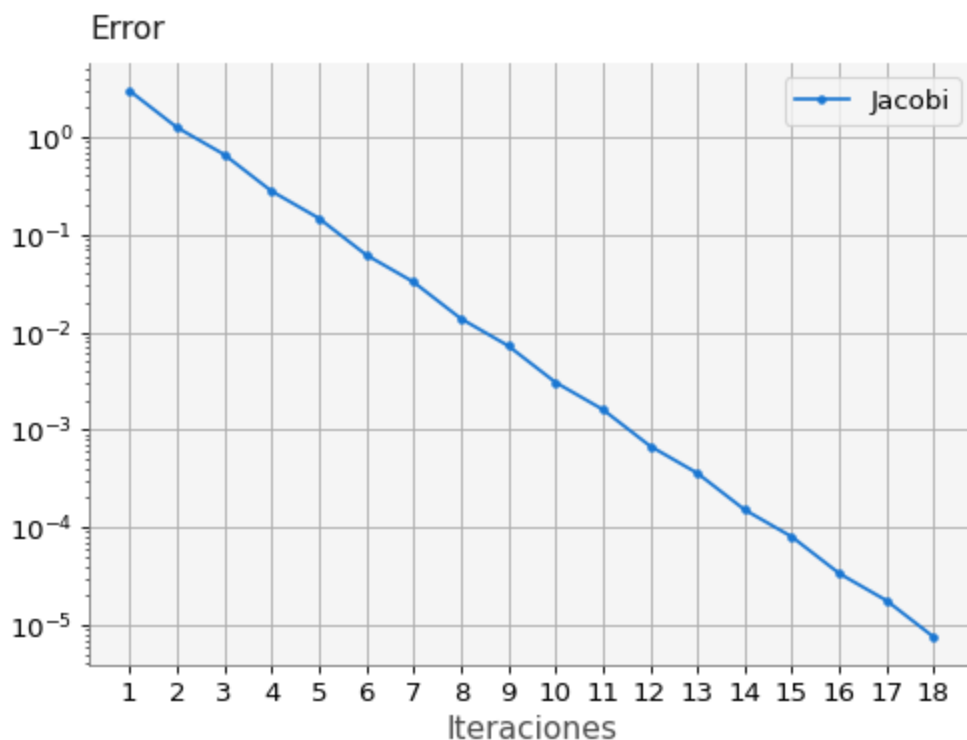
$$\sum_{j \neq i} |a_{i,j}| < |a_{i,i}|, \forall i$$

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()
```



5 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: $x_i^k = (b_i - \sum_{j < i} a_{ij} x_j^k - \sum_{j > i} a_{ij} x_j^{k-1}) / a_{ii}$
- $\sum_{j > i} a_{ij} x_j^{k-1}$
- 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.

5.1 Algoritmo Gauss-Seidel.

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



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

5.2 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
```


5.3 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

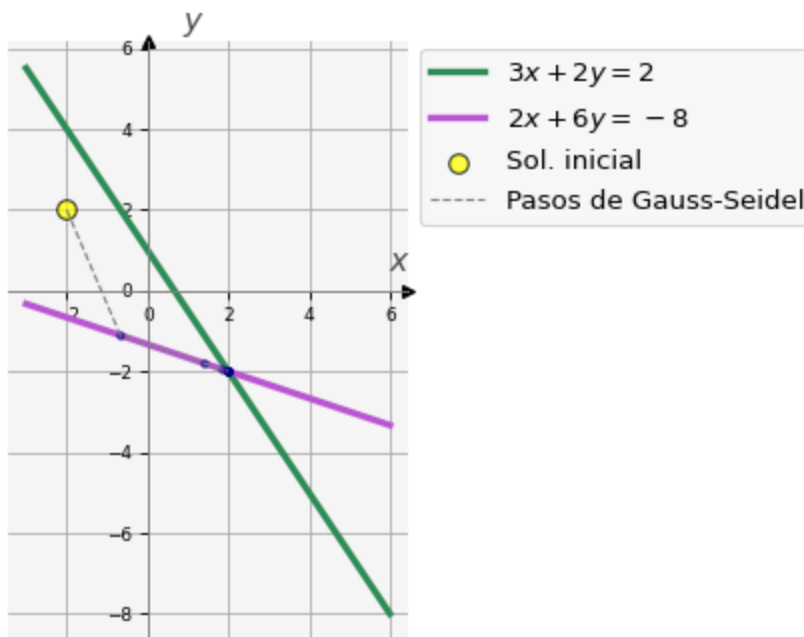
Puedes usar el mismo código que en el caso anterior.

```
#### 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, 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 Gauss-Se

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

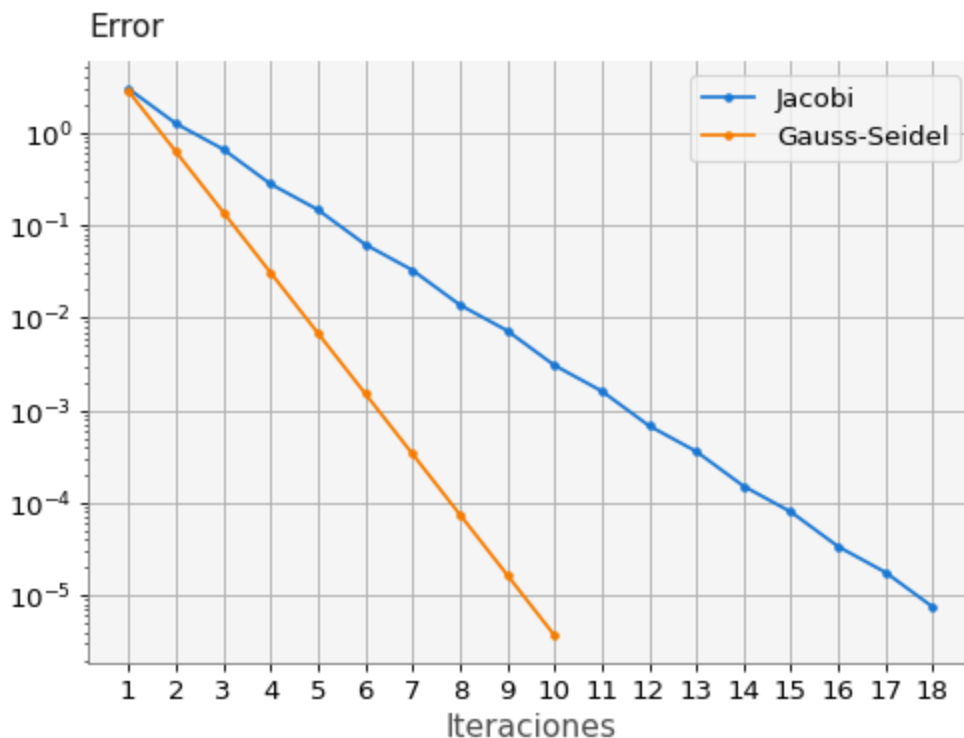
```
# Utiliza el código del caso anterior adaptado para que pueda graficar ambos errores

#### 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
```



6 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: $x^k = (-)^{-1} \{ + (1 -) \}^{k-1}$
- $(-)^{-1}$
- Elegir la ω óptima no es simple, aunque se sabe que si ω está fuera del intervalo $(0, 2)$ el método falla.

6.1 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

```

6.2 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 $\text{tol} = 1 \times 10^{-5}$ y $\text{kmax} = 50$ 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

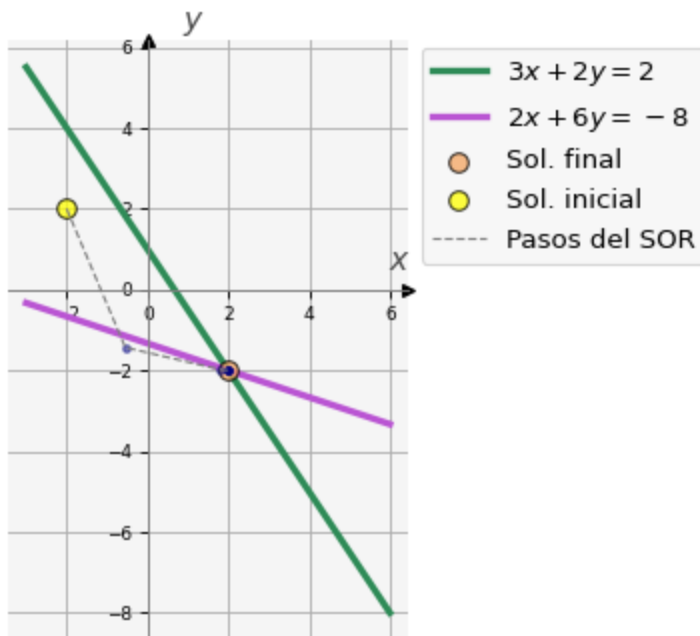
Puedes usar el mismo código que en el caso anterior.

```
#### 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



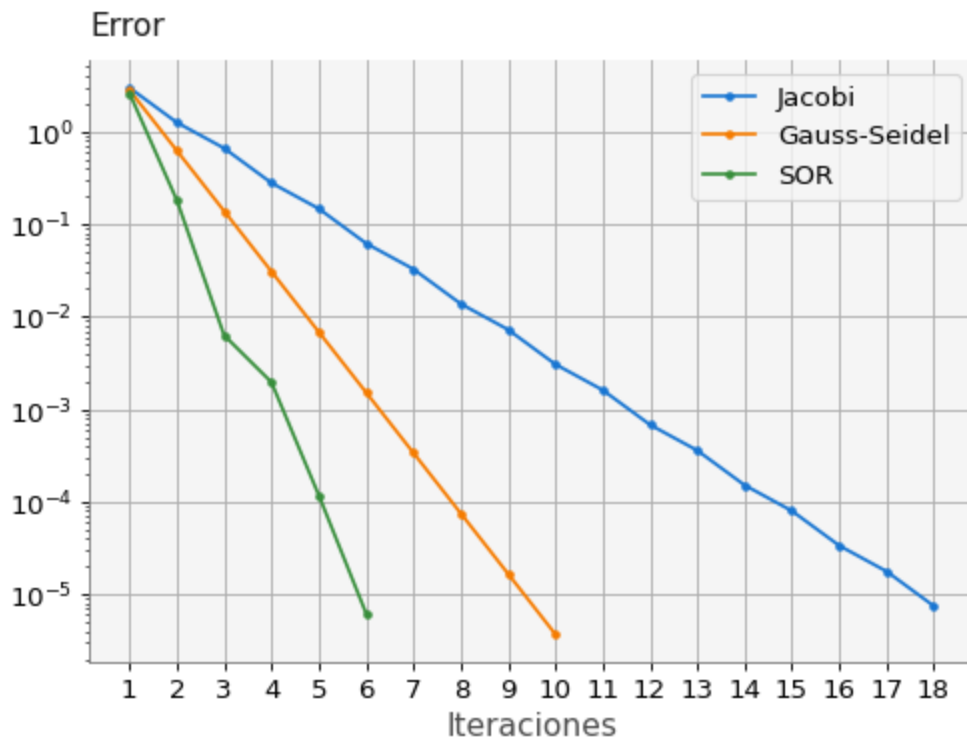
```
# Utiliza el código del caso anterior adaptado para que pueda graficar los tres e

#### 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
```



6.3 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]



6 Conducción de calor en 2D: algoritmos de solución de sistemas de ecuaciones.

Objetivo.

Comparar mediante un interactivo varios métodos de solución de sistemas de ecuaciones lineales.

[MACTI-Analisis_Numerico_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

Al ejecutar la siguiente celda obtendrás un interactivo en donde podrás seleccionar el método de solución y el tamaño de la malla ($M \times N$) para resolver numéricamente, por diferencias finitas, un problema de conducción de calor.

Analiza con cuidado los valores más óptimos para encontrar una buena solución.

NOTA. Para ejecutar el interactivo debes hacer clic en el botón de play (a la derecha).

```
%run "./zHeatCondSolvers.py"
```

