Laboratorio 5: métodos implícitos

¡Ya estamos en el quinto laboratorio! El curso partió revisando conceptos básicos de discretización de derivadas, para pasar a ecuaciones diferenciales ordinarias y después a ecuaciones diferenciales parciales. Eso si, en todos los laboratorios hemos usado la misma forma de discretizar el tiempo, usando diferencias atrasadas, y eso cambiará hoy. Algo de esto vimos en el laboratorio 3 (sobre ecuaciones diferenciales ordinarias), pero hoy profundizaremos aún más.

Teoría

Método explícito

Usemos la ecuación de difusión en una dimensión para guiar la discusión:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}.$$

¿Por qué usamos T y α? La ecuación de difusión modela la conducción de calor, donde T es la temperatura y $\alpha = \kappa/\rho c_p$ el coeficiente de difusividad térmica.

En el laboratorio 4 discretizamos la ecuación de difusión con diferencias atrasadas en tiempo y centradas en el espacio, lo que nos dió:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} - \alpha \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{(\Delta x)^2} = 0.$$

La gracia de usar diferencia atrasada es que para obtener el valor de T_i^{n+1} usamos solamente variables del paso de tiempo anterior, que es conocido. Pongamos lo que no conocemos al lado izquierdo y lo conocido al lado derecho:

$$T_i^{n+1} = T_i^n + \frac{\alpha \Delta t}{(\Delta x)^2}\left(T_{i+1}^n - 2T_i^n + T_{i-1}^n\right),$$

así, llegamos a una ecuación explícita, fácil de resolver. La diferencia atrasada en el tiempo es equivalente a decir que estamos usando el método de Euler explícito que vimos en el laboratorio 3.

Una representación gráfica de esto es:

Vimos en el laboratorio 4 que esta discretización tiene una restricción de estabilidad muy estricta, que era:

$$\frac{\alpha \Delta t}{(\Delta x)^2} < \frac{1}{2}$$

y nos obligaba a usar pasos de tiempo ridículamente pequeños.

Método implícito

El laboratorio 3 esbozó una solución a esto. Vimos que la integración con Euler usando los valores del tiempo $n+1$ en la función $f(t_{n+1}, y(t_{n+1}))$ era incondicionalmente estable. Esto es equivalente a usar diferencias adelantadas en el tiempo, lo que nos deja:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} - \alpha \frac{T_{i+1}^{n+1} - 2T_i^{n+1} + T_{i-1}^{n+1}}{(\Delta x)^2} = 0.$$

Hasta ahora, todo bien. No es más que usar Euler implícito en el tiempo.

Pongamos lo que no conocemos a la izquierda y lo que conocemos a la derecha:

$$-T_{i+1}^{n+1} + \left(2 + \frac{(\Delta x)^2}{\alpha \Delta t}\right) T_i^{n+1} - T_{i-1}^{n+1} = \frac{(\Delta x)^2}{\alpha \Delta t} T_i^n$$

y aquí se hace evidente que tenemos un problema: una ecuación, tres incógnitas. Gráficamente, esto se puede visualizar así:

Digamos que tenemos una malla con N puntos en el espacio que va desde $i=0$ hasta $i=N-1$, con una condición de borde de Dirichlet a la izquierda (T_0^n conocido para todo n) y una de Neumann a la derecha ($\partial T_{N-1}^n/\partial x$ conocido para todo n). T_0^{n+1} es conocido, pero ¿cómo se ve la ecuación para T_1^{n+1}?

$$-T_0^{n+1} + \left(2 + \frac{\Delta x^2}{\alpha \Delta t}\right) T_1^{n+1} - T_2^{n+1} = T_1^n \frac{\Delta x^2}{\alpha \Delta t}$$

¿y para T_2^{n+1}?

$$-T_1^{n+1} + \left(2 + \frac{\Delta x^2}{\alpha \Delta t}\right) T_2^{n+1} - T_3^{n+1} = T_2^n \frac{\Delta x^2}{\alpha \Delta t}$$

¿y T_3^{n+1}?

$$-T_2^{n+1} + \left(2 + \frac{\Delta x^2}{\alpha \Delta t}\right) T_3^{n+1} - T_4^{n+1} = T_3^n \frac{\Delta x^2}{\alpha \Delta t}$$

ven el patrón, ¿cierto? Se repite T_2^{n+1} en los tres casos: si aplicamos la ecuación para cada punto de la malla, tendremos el mismo número de ecuaciones e incógnitas.

Condiciones de borde

Como en los puntos en $i=0$ e $i=N-1$ se enfuerzan las condiciones de borde, no es necesario aplicar la ecuación en ellos. ¿Qué hacemos entonces?

Condición de borde de Dirichlet

El caso $i=0$ es una condición de borde de Dirichlet, por lo tanto, el término T_0^{n+1} en la ecuación para T_1^{n+1} es conocido y puede pasar al lado derecho de la ecuación:

$$-\left(2 + \frac{\Delta x^2}{\alpha \Delta t}\right) T_1^{n+1} + T_2^{n+1} = -T_1^n \frac{\Delta x^2}{\alpha \Delta t} - T_0^{n+1}$$

Condición de borde de Neumann

En $i=N-1$ hay una condición de borde de Neumann y es algo más difícil. Este punto aparece en la ecuación para $i=N-2$, que es:

$$-T_{N-3}^{n+1} + \left(2 + \frac{\Delta x^2}{\alpha \Delta t}\right) T_{N-2}^{n+1} - T_{N-1}^{n+1} = T_{N-2}^n \frac{\Delta x^2}{\alpha \Delta t}$$

Digamos que la condición de borde es que hay una entrada de calor en $x=L$:

$$\left.\frac{\partial T}{\partial x}\right|_{x=L} = q$$

La forma más fácil de tratar con la condición de borde de Neumann es discretizar con diferencias finitas:

$$\frac{T_{N-1}^{n+1} - T_{N-2}^{n+1}}{\Delta x} = q.$$

y reemplazar en la ecuación anterior:

$$-T_{N-3}^{n+1} + \left(1 + \frac{\Delta x^2}{\alpha \Delta t}\right) T_{N-2}^{n+1} = T_{N-2}^n \frac{\Delta x^2}{\alpha \Delta t} + \Delta x q$$

¿Cuál es el problema? Hasta ahora, la discretización en el espacio era de segundo orden. Con esto, estamos agregando un error de primer orden. Existen maneras más sofisticadas de enforzar la condición de borde de Neumann sin perder el orden de convergencia, pero eso es materia para un curso más avanzado.

Generación de la matriz

Si aplicamos la ecuación discretizada en cada punto, llegamos a un sistema de ecuaciones del tipo

$$\begin{bmatrix} \left(2+\frac{1}{\sigma}\right) & -1 & 0 & \cdots & & 0 \\ -1 & \left(2+\frac{1}{\sigma}\right) & -1 & 0 & \cdots & 0 \\ 0 & & \ddots & & & \vdots \\ \vdots & & & \left(2+\frac{1}{\sigma}\right) & & \\ 0 & \cdots & & -1 & & \left(1+\frac{1}{\sigma}\right) \end{bmatrix} \cdot \begin{bmatrix} T_1^{n+1} \\ T_2^{n+1} \\ \vdots \\ \\ T_{N-2}^{n+1} \end{bmatrix} = \begin{bmatrix} T_1^n \frac{1}{\sigma} \\ T_2^n \frac{1}{\sigma} \\ \vdots \\ \\ T_{N-2}^n \frac{1}{\sigma} \end{bmatrix} + \begin{bmatrix} T_0^{n+1} \\ 0 \\ \vdots \\ 0 \\ q\Delta x \end{bmatrix}$$

donde $\sigma = \frac{\alpha \Delta t}{\Delta x^2}$. Fíjense que las condiciones de borde solamente afectan el lado derecho del sistema lineal en los puntos $i=1$ (primera fila) e $i=N-2$ (última fila).

No se confundan: hay N puntos de discretización, pero dos de ellos son condiciones de borde, por lo tanto, la dimensión de la matriz es $N-2 \times N-2$.

Para resolver la ecuación numéricamente con un método implícito tenemos que encontrar la solución de un sistema lineal en cada paso de tiempo, donde la matriz es siempre la misma, pero el lado derecho de la ecuación va cambiando. Esto es mucho más lento que simplemente calcular T_i^{n+1} en el caso explícito, pero nos permite usar pasos de tiempo muchísimo mayores ¿qué es mejor? depende del caso.

Ejemplo

Tenemos una barra de grafito $(\alpha = 1.22 \cdot 10^{-3} \mathrm{m}^2/\mathrm{s})$ de $1\mathrm{m}$ de largo. Inicialmente está a $0^\circ\mathrm{C}$, y al tiempo $t=0$ subimos la temperatura al lado izquierdo a $100^\circ\mathrm{C}$. Al lado derecho esta aislado por lo que no fluye calor $(q=0)$ ¿Cómo evoluciona la temperatura en la barra?

Con una malla de $N=51$ puntos, grafiquen la temperatura en la barra luego de 100 pasos de tiempo.

Primero, resolveremos este problema con un método explícito, luego ustedes lo harán con el método implícito.

Euler explícito

Para empezar, importemos las librerías necesarias y definamos la malla

In [1]:

```
import numpy as np
import numpy
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.size'] = 12

L = 1.
nt = 100
nx = 51
alpha = 1.22e-3

dx = L/(nx-1)

x  = numpy.linspace(0, 1, nx)
Ti = numpy.zeros(nx)
Ti[0] = 100
```

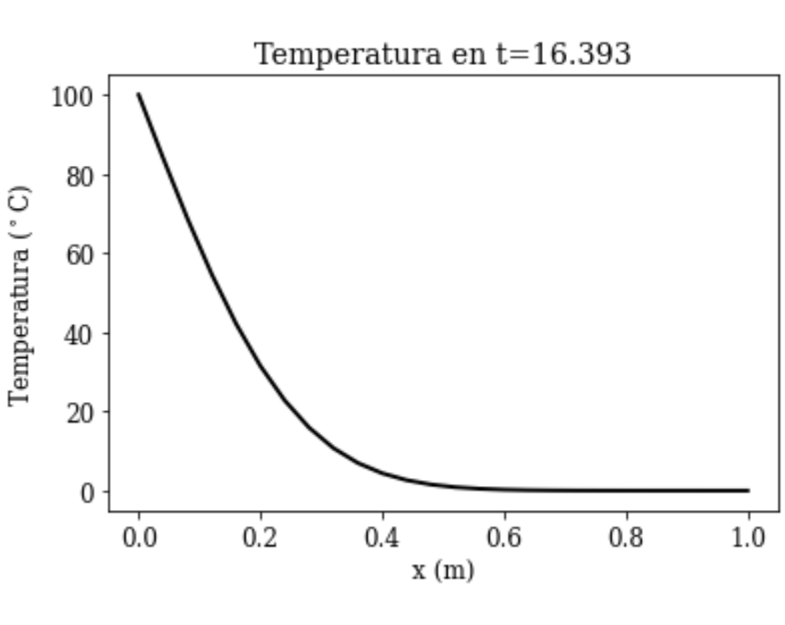
Generen una función que resuelva el problema usando un método explícito de primer orden en el tiempo y segundo orden en el espacio (ver laboratorio 4). Sabemos que el método es inestable para $\sigma = \Delta t\alpha/(\Delta x)^2 > 0.5$. Para evitar problemas, vamos a definir Δt en función de σ.

Para enforzar las condiciones de borde, utilicen la aproximación de primer orden $\frac{T_{N-1} - T_{N-2}}{\Delta x} = 0$.

In [2]:

```
### ALUMNO

def euler_explicito(T,nt,nx,dt,dx,alpha):
    for n in range(1,nt):
        Tn = T.copy()
        for i in range(nx-1):
            if i == 0:
                T[i] = 100
            else:
                T[i] = Tn[i]+dt*(alpha/(dx**2)*(Tn[i+1]-2*Tn[i]+Tn[i-1]))
        T[-1] = T[-2]
    return T


###
sigma = 0.5
dt    = sigma*dx**2/alpha

print ('El paso de tiempo es %1.4f'%dt)

T = Ti.copy()
T = euler_explicito(T, nt,nx, dt, dx, alpha)
plt.plot(x, T, color='k', lw=2)
plt.title('Temperatura en t=%1.3f'%(nt*dt))
plt.ylabel(r'Temperatura ($^\circ$C)')
plt.xlabel('x (m)');
```

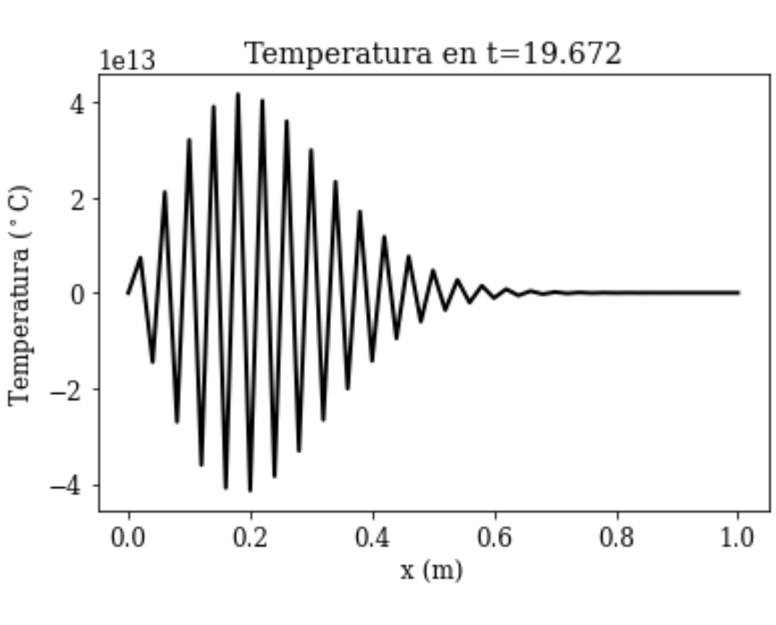
```
El paso de tiempo es 0.1639
```



In [3]:

```
sigma = 0.6
dt    = sigma*dx**2/alpha

print ('El paso de tiempo es %1.4f'%dt)

T = Ti.copy()
T = euler_explicito(T, nt,nx, dt, dx, alpha)
plt.plot(x, T, color='k', lw=2)
plt.title('Temperatura en t=%1.3f'%(nt*dt))
plt.ylabel(r'Temperatura ($^\circ$C)')
plt.xlabel('x (m)');
```

```
El paso de tiempo es 0.1967
```



Claramente el método se vuelve inestable para ese valor de σ

Prueben con $\sigma = 0.6$. Ya saben lo que va a pasar ¿cierto?

Euler implícito

Vamos a resolver el mismo problema usando Euler implícito. En este caso, tenemos que resolver un sistema lineal en cada paso de tiempo ¿Cómo podemos resolver el sistema lineal?

Aquí es donde usar lenguajes como Python se hace muy útil: existen librerías que resuelven sistemas lineales `Ax=b` dada una matriz `A` y un vector `b`. La más conocida es `scipy`, pero no es la única. Por ejemplo `scipy.linalg.solve(A, b)` encuentra el vector `x` para `Ax=b`.

Generemos una función que haga el paso de tiempo de Euler implícito:

In [4]:

```
from scipy.linalg import solve
def euler_implicito(T, nx, nt, sigma):
    A = generarMatriz(nx,sigma)
    for t in range(nt):
        Tn = T.copy()
        b = lado_derecho(Tn, sigma)
        T_interior = solve(A,b)
        T[1:-1] = T_interior
        T[0]  = 100   # Dirichlet BC
        T[-1] = T[-2] # Neumann BC

    return T
```

Fíjense que el resultado del sistema lineal es `T_interior`. Éste es un arreglo con las temperaturas al interior del dominio, sin considerar los bordes. Por lo tanto, `T_interior` tendrá $N-2$ elementos, mientras que `T` dentrá N elementos.

Es el turno de ustedes

Como ven en la función `euler_implicito`, la matriz `A` es un dato de entrada, y debemos generar el lado derecho de la ecuación en cada paso de tiempo. La tarea de este laboratorio es generar una función que genere la matriz `A` y otra función llamada `lado_derecho` que calcule el lado derecho de la ecuación en cada paso de tiempo.

Primero prueben con $\sigma = 0.5$, pero luego cambien a $\sigma = 2$ ¿Explota la solución?

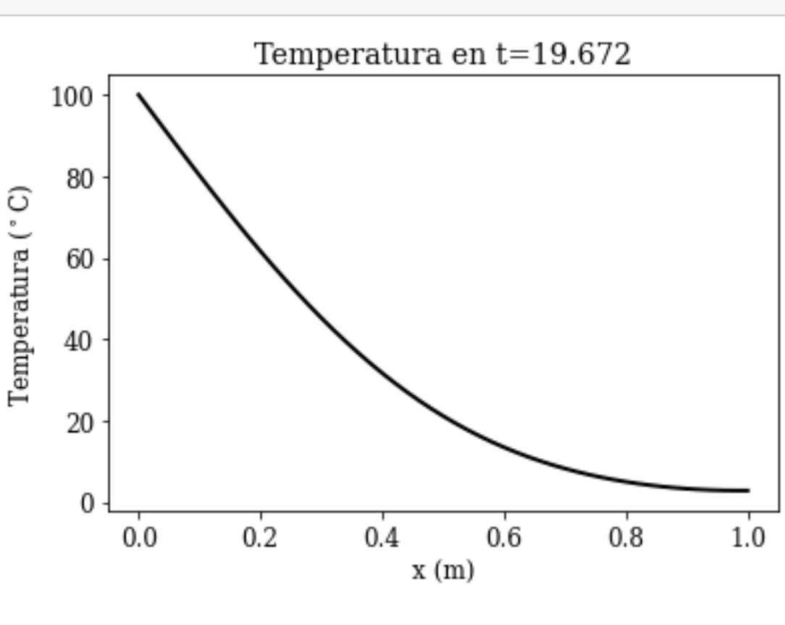
Ayuda: Python nos ofrece un sinnúmero de opciones para generar la matriz, pero una forma fácil es generar una matriz llena de zeros de tamaño $N-2 \times N-2$ con

```
nx = len(x)
numpy.zeros((nx-2,nx-2))
```

y modificar las posiciones de la matriz que correspondan.

In [5]:

```
###ALUMNO
def generarMatriz(nx, sigma):
    n2 = nx-2
    A = np.zeros((n2,n2))
    for i in range(n2):
        A[i,i] = 2+1/sigma
        if i!=0:
            A[i,i-1] = -1
        if i!= n2-1:
            A[i,i+1] = -1
    A[-1,-1] = 1+1/sigma
    return A

def lado_derecho(T, sigma):
    b = T[1:-1].copy()/sigma
    b[0] += 100
    return b

sigma = 2
A = generarMatriz(nx,sigma)
T = Ti.copy()
T = euler_implicito(T, nx,nt, sigma)
plt.plot(x, T, color='k', lw=2)
plt.title('Temperatura en t=%1.3f'%(nt*dt))
plt.ylabel(r'Temperatura ($^\circ$C)')
plt.xlabel('x (m)');
#generarMatriz(10,1)
```



Si les resultó el programa (¡esperamos!), verán que pueden usar σ (y por ende Δt) muchísimo mayores. Esto es conveniente ya que pueden llegar mucho más lejos en el tiempo, con menos pasos. Por supuesto, existe un límite a esto: al aumentar Δt la aproximación se hace cada vez menos exacta, y puede llegar al límite de que no capte cosas que están pasando físicamente a una escala de tiempo menor que Δt.

Convergencia espacial

El problema que resolvimos recién es suficientemente simple que tiene una solución analítica. Ésta es:

$$T(x,t) = 100 - \sum_{n=1}^{\infty} \frac{400}{(2n-1)\pi} \sin\left(\frac{(2n-1)\pi}{2L} x\right) \exp\left[-\alpha\left(\frac{(2n-1)\pi}{2L}\right)^2 t\right]$$

y la siguiente función la calcula en los puntos `x` al tiempo `t`, usando `n_max` términos (en vez de ∞):

In [6]:

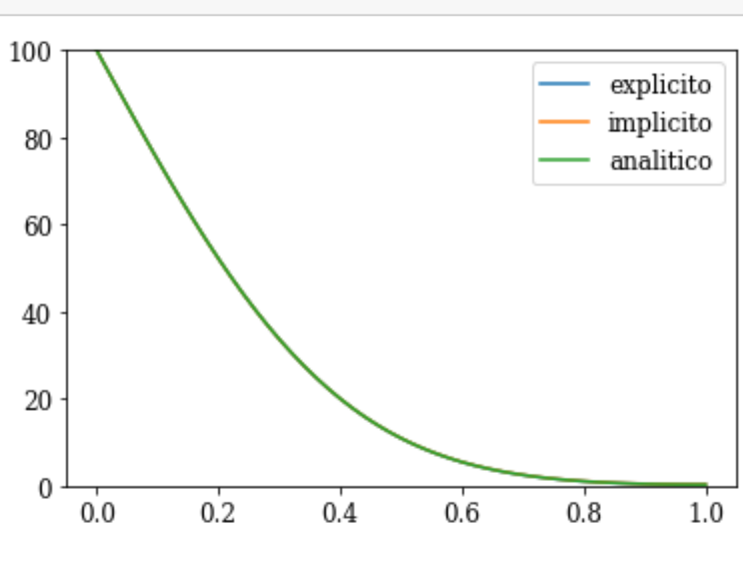
```
from math import pi
def T_analytical(x, t, alpha, L, n_max=100):

    T = 100
    for n in range(1,n_max+1):
        k = (2*n-1)*pi/(2*L)

        summation = 400/((2*n-1)*pi) * numpy.sin(k*x) * numpy.exp(-alpha*k*k*t)
        T -= summation

    return T
```

Usando $\sigma = 1.22$, calculen el error L2 (laboratorio 2) de aproximación al tiempo $t=40$ con mallas de $N = 26, 51, 101, 201$, y grafiquen error versus Δx usando *log-log* ¿Cuál es la convergencia espacial del método? ¿Es lo que esperaban? *Nota*: fíjense que necesitan diferente número de pasos de tiempo en cada caso.

Repitan los cálculos al tiempo $t=400$. ¿Pasa algo con la convergencia? ¿Por qué? *Nota*: ¿Cuál es el orden de aproximación de la condición de contorno en el lado derecho?

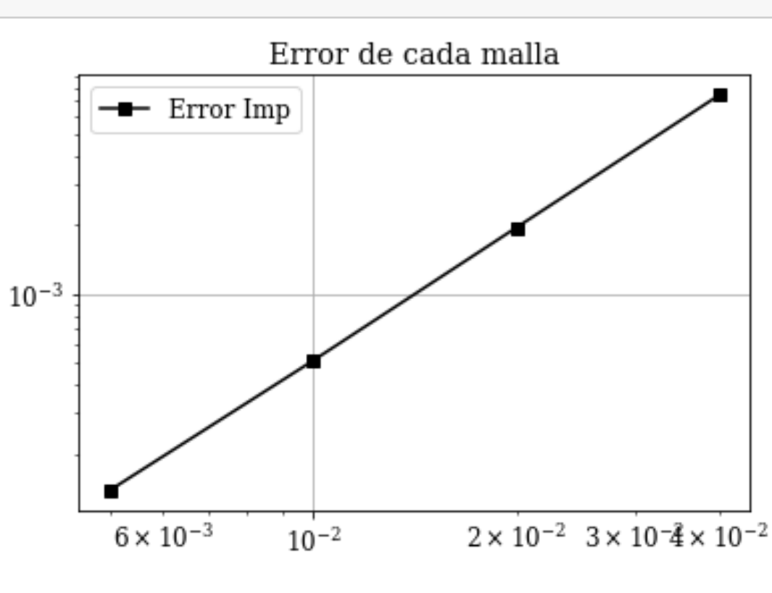
In [7]:

```
### ALUMNO
#Estudie la convergencia con las mallas indicadas
###

def graficar():
    #x = np.linspace(0,1,nx)
    L = 1.
    #nt = 100
    nx = 51
    t = 40
    alpha = 1.22e-3
    sigma = 0.4

    dx = L/(nx-1)
    x  = numpy.linspace(0, 1, nx)

    dt = sigma*dx**2/alpha
    nt = int(t/dt)
    Ti = numpy.zeros(nx)
    Ti[0] = 100

    t = nt*dt
    T_expl = euler_explicito(Ti, nt, nx,dt, dx, alpha)
    plt.plot(x,T_expl, label="explicito")

    Ti = numpy.zeros(nx)
    Ti[0] = 100

    T_impl = euler_implicito(Ti, nx,nt, sigma)
    plt.plot(x,T_impl, label="implicito")
    x = np.linspace(0,1,100)
    T_anal = T_analytical(x, t, alpha, L, n_max=100)
    plt.plot(x,T_anal, label="analitico")
    plt.ylim([0,100])
    plt.legend()

graficar()
```



In [8]:

```
def L2_error(f_ap,f_an):
    df_an_app = (f_ap-f_an)**2
    f_an2 = f_an**2
    return np.sqrt(np.sum(df_an_app)/np.sum(f_an2))


L_nodos = np.array([26,51,101,201])
delta_dx = 1/(np.array(L_nodos)-1)
error = numpy.zeros(len(L_nodos))

for k in range(len(L_nodos)):
    L = 1.
    nx = L_nodos[k]
    t = 40
    alpha = 1.22e-3
    sigma = 1.22

    dx = L/(nx-1)
    x  = numpy.linspace(0, 1, nx)

    dt = sigma*dx**2/alpha
    nt = int(t/dt)
    t = nt*dt
    Ti = numpy.zeros(nx)
    Ti[0] = 100
    T_impl = euler_implicito(Ti, nx,nt, sigma)
    T_anal = T_analytical(x, t, alpha, L, n_max=100)

    error[k] = L2_error(T_impl,T_anal)

plt.loglog(delta_dx,error,label="Error Imp",c='k', ls='-', marker='s')
plt.title("Error de cada malla")
plt.legend(loc='best',prop={'size':12})
plt.grid()
```



In [9]:

```
err = error[-4]/error[-1]
dx = delta_dx[-4]/delta_dx[-1]
print(dx,err)
```

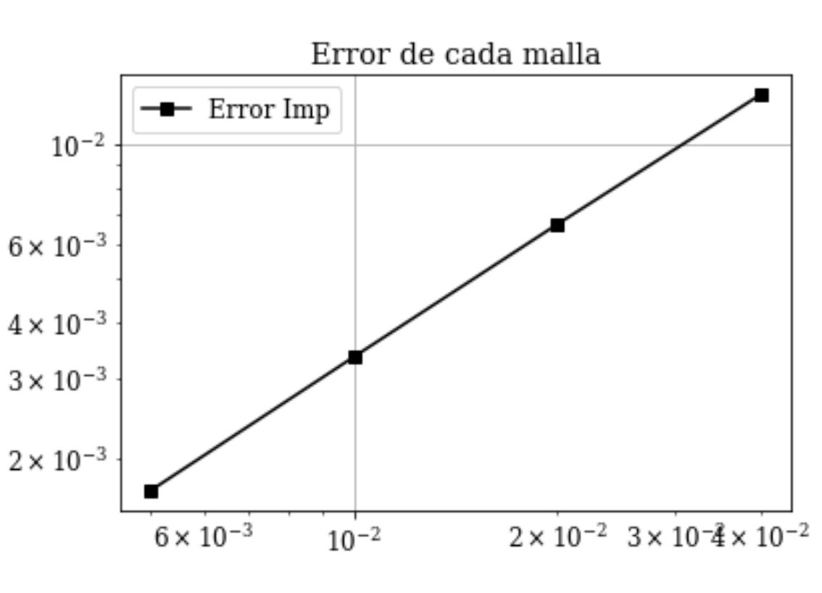
```
8.0 53.94366340169201
```

El método parece ser de orden 2. Yo pensé que tenía que ser de orden 1, dada la discretización para la condición de Neumann que se usó. Yo creo que el método quedó de orden 2 ya que como la temperatura es casi uniforme en el extremo derecho para este tiempo, la diferencia de T entre los nodos es casi nula, por lo que un diferente stencil daría el mismo resultado. Por esta razón, para este tiempo, la discretización de orden 1 parece no afectar el método. Se podría en términos físicos decir que la perturbación en la temperatura todavía no llega al lado derecho, por ende, este no tiene efecto.

In [10]:

```
for k in range(len(L_nodos)):
    L = 1.
    nx = L_nodos[k]
    t = 400
    alpha = 1.22e-3
    sigma = 1.22

    dx = L/(nx-1)
    x  = numpy.linspace(0, 1, nx)

    dt = sigma*dx**2/alpha
    nt = int(t/dt)
    t = nt*dt
    Ti = numpy.zeros(nx)
    Ti[0] = 100
    T_impl = euler_implicito(Ti, nx,nt, sigma)
    T_anal = T_analytical(x, t, alpha, L, n_max=100)

    print(k)
    error[k] = L2_error(T_impl,T_anal)

plt.loglog(delta_dx,error,label="Error Imp",c='k', ls='-', marker='s')
plt.title("Error de cada malla")
plt.legend(loc='best',prop={'size':12})
plt.grid()
```

```
0
1
2
3
```



In [11]:

```
err = error[-4]/error[-1]
dx = delta_dx[-4]/delta_dx[-1]
print(dx,err)
```

```
8.0 7.640901008134011
```

Se nota claramente que el método ahora es de orden 1. Esto coincide con lo que se dijo anteriormente. Ahora el lado derecho es importante ya que llegó la perturbación de temperatura. Dado esto, el método se condiciona completamente por su discretización de orden 1.

Para los interesados

La convergencia temporal debiese ser de primer orden, ya que estamos usando Euler. ¿Se acuerdan de Crank-Nicolson en el laboratorio 3? Ese método es de segundo orden. Si hacen la misma derivación que hicimos acá, veran que con modificaciones simples a la matriz y el lado derecho de la ecuación, ¡pueden implementar un esquema de Crank-Nicolson!

In [ ]:

In [ ]:

In [ ]:

In [ ]: