Elementos de Cálculo Numérico / Cálculo Numérico (Primer cuatrimestre 2020)

El problema de valores de contorno que analizamos en la clase práctica (04/05) es el de la ecuación de transporte:

$$\begin{cases} u_t(x,t) + a u_x(x,t) = 0 \qquad a \in \mathbb{R}, a > 0, x \in (0, \mathcal{X}], t \in (0, T] \\ u(0,t) = g(t) \\ u(x,0) = f(x) \end{cases}$$

Discretizamos el intervalo $[0, \mathcal{X}]$ en N intervalos de medida h y el intervalo $[0, T]$ en M intervalos de medida k. Vimos que el siguiente esquema resulta consistente, estable y convergente si $a\nu \leq 1$, siendo $\nu = \frac{k}{h}$:

$$u_i^{j+1} = a\nu u_{i-1}^j + (1 - a\nu) u_i^j$$

Teniendo en cuenta que:

$$i = 1 \qquad u_1^{j+1} = a\nu u_0^j + (1 - a\nu) u_1^j = a\nu g(t_j) + (1 - a\nu) u_1^j$$

$$i = 2, \ldots, N \qquad u_i^{j+1} = a\nu u_{i-1}^j + (1 - a\nu) u_i^j$$

Podemos escribir la representación matricial como $u^{j+1} = Au^j + g^j$ donde:

- A es la matriz que tiene $1 - a\nu$ en la diagonal y $a\nu$ en la subdiagonal.
- g^j es un vector cuya primera coordenada es $a\nu g(t_j)$ y tiene cero en las demás.

En el siguiente archivo de Python se resuelve el siguiente problema de valores de contorno:

$$\begin{cases} u_t(x,t) + u_x(x,t) = 0 \qquad x \in (0, 2], t \in (0, 1] \\ u(0,t) = 0 \\ u(x,0) = e^{-10(4x-1)^2} \end{cases}$$

Además, el script muestra una animación de cómo evoluciona u en el tiempo y la compara con la solución exacta para este problema: $u(x,t) = e^{-10(4(x-t)-1)^2}$

Observación: para asegurar que el método sea estable y convergente, pedimos $a\frac{k}{h} \leq 1$, luego $k \leq \frac{h}{a}$. En términos de N y M, para garantizar estabilidad, tenemos que:

$$k \leq \frac{h}{a} \Leftrightarrow \frac{T}{M} \leq \frac{\mathcal{X}}{aN} \Leftrightarrow M \geq \frac{aTN}{\mathcal{X}}$$

In [ ]:

```
# Importamos numpy y matplotlib
import numpy as np
import matplotlib.pyplot as plt
# Importamos time, es un módulo que ya viene con Python, así que no hay que instalarlo
import time


# Esta función aplica el esquema que vimos en clase:
# t0: es el tiempo inicial
# tf: es el tiempo final
# x0: inicio del intervalo de posiciones
# xf: final del intervalo de posiciones
# a: constante del problema (para que funcione bien, debe valer que a > 0)
# g : función de valor de contorno u(0,t) = g(t)
# f : función de valor de contorno u(x, 0) = f(x)
# N: cantidad de puntos en los que queremos discretizar [x0, xf]
# M: cantidad de puntos en los que queremos discretizar [t0, tf]
def upwind(t0, tf, x0, xf, a, g, f, N, M):

    # Discretizamos [x0, xf] y calculamos h
    x = np.linspace(x0, xf, N)
    h = (xf - x0) / N

    # Discretizamos [t0, tf] y calculamos k
    t = np.linspace(t0, tf, M)
    k = (tf - t0) / M

    # Calculamos nu
    nu = k / h

    # Escribimos la matriz A con 1-a*nu en la diagonal y a*nu en la subdiagonal
    A = np.zeros((N, N))
    np.fill_diagonal(A, 1 - a * nu)
    np.fill_diagonal(A[1:N, :N - 1], a * nu)

    # Inicializamos una matriz que guarda las aproximaciones numericas. Esta matriz
    # tiene M filas y N columnas: en el lugar (j,i) tendremos la aproximación
    # numérica de u(x_i, t_j)
    sol_numerica = np.empty((M, N))

    # Escribimos u^0: en la coordenada i, este vector tiene el valor de f(x_i) = u(x,
 0) .
    # Lo agregamos como primera fila de la matriz de aproximaciones:
    u = np.array([f(X) for X in x])
    sol_numerica[0, :] = u

    # Realizamos las iteraciones del modelo
    for j in range(1, M):
        # Calculamos el vector g^j   (recordar que t_j = j*k + t0)
        G = np.array([g(j * k + t0)] + [0] * (N - 1))
        # Calculamos los valores del siguiente paso temporal y los guardamos en la vari
able u
        u = A @ u + G
        # Agregamos el resultado de esta iteracion en la matriz de aproximaciones numer
icas
        sol_numerica[j, :] = u

    # Devolvemos la matriz de resultados y las discretizacion en x
    return sol_numerica, x
```

```
def ecuacion_de_transporte():

    # Definimos la funcion f(x)   (corresponde al valor de contorno u(x,0) = f(x))
    def f(x):
        return np.exp(-10 * ((4 * x - 1) ** 2))

    # Definimos la funcion g(t)    (corresponde al valor de contrno u(0, t) = g(t))
    def g(t):
        return 0

    # Definimos la solucion exacta de la ecuacion
    def u(x, t):
        return f(x - a * t)

    # Introducimos los datos de nuestro problema
    x0 = 0    # Inicio del intervalo espacial
    xf = 2    # Fin del intervalo espacial
    t0 = 0    # Tiempo inicial
    tf = 1    # Tiempo final

    # Estos valores cumplen con la condicion que garantiza estabilidad y convergencia
    N = 200
    M = 100

    # Introducimos el valor de la constante de la ecuacion y calculamos h y k
    a = 1
    h = (xf - x0) / N
    k = (tf - t0) / M

    # Utilizamos el método para obtener la matriz con las aproximaciones
    # numéricas y la discretizacion en x
    sol_numerica, xd = upwind(t0, tf, x0, xf, a, g, f, N, M)

    # Creamos un matriz donde guardamos el valor exacto de u(x_i, t_j) en su fila j y c
olumna i
    sol_exacta = np.empty((M, N))
    for j in range(M):
        for i in range(N):
            sol_exacta[j, i] = u(h * i + x0, j * k + t0)

    # Aqui comienza la parte en la que elaboramos el gráfico animado

    # Abrimos una ventana sin ninguna figura o gráfico
    plt.show()

    # Creamos una nueva figura y pedimos que sólo nos muestre los
    # valores del eje x en [0, 1.5] y los valores del eje y en [0,2]
    axes = plt.gca()
    axes.set_xlim(0, 1.5)
    axes.set_ylim(0, 2)

    # Agregamos a la figura el gráfico de la aproximacion numerica a
    # tiempo 0, que es la primera fila de la matriz "sol_numerica"
    grafico_solnum, = axes.plot(xd, sol_numerica[0, :], 'r')

    # Agregamos a la figura el gráfico de la solucion exacta a tiempo 0,
    # que es la primera fila de la matriz "sol_exacta"
    grafico_solex, = axes.plot(xd, sol_exacta[0, :], 'b')
```

```
    # Ahora elaboraremos los "cuadros" de la animación,
    # es decir, vamos a graficar para cada t_j
    for j in range(M):

        # Actualizamos el gráfico de la solucion numerica
        # En el eje x mantenemos los mismo datos
        grafico_solnum.set_xdata(xd)
        # Actualizamos los datos del eje y: usamos la fila j de sol_num, que correspond
e a t_j
        grafico_solnum.set_ydata(sol_numerica[j, :])

        # Actualizamos el gráfico de la solucion exacta de manera análoga
        grafico_solex.set_xdata(xd)
        grafico_solex.set_ydata(sol_exacta[j, :])

        # Pedimos que dibuje la figura y los gráficos en la ventana que abrimos al prin
cipio
        plt.draw()

        # Pedimos que realice una breve pausa entre este cuadro y el siguiente
        plt.pause(1e-17)
        time.sleep(0.1)

    # Incluimos esto para que no se cierre la ventana con la animación cuando termine
    plt.show()


ecuacion_de_transporte()
```