Instituto Tecnológico de Buenos Aires

93.54 - Métodos Numéricos

Parcial Integrador

Vekselman, Alan 59378

Profesores
Fierens, Pablo Ignacio
Grisales Campeón, Juan Pablo

Fecha de entrega: 28/05/2021

Índice

1. Ejercicio 1

En este ejercicio, se proponía implementar un algoritmo para resolver un sistema de ecuaciones lineales por el método iterativo de Jacobi, de la forma:

$$A \cdot \vec{x} = \vec{b} \tag{1}$$

El código implementado fue el siguiente:

```
import numpy as np

def jacobi(coef: np.array, y: np.array, tol : [int, float]):

    x_k = np.zeros(coef.shape[0])
    diag = coef.diagonal()

    tot, done = 0, False
    while not done:
        tot += 1
        sums_ = coef.dot(x_k.reshape(-1, 1)).reshape(-1) - diag * x_k
        x_temp = (y - sums_) / diag
        if np.all(np.abs(x_k - x_temp) <= tol): done = True
        x_k = x_temp

    return x_k, tot
```

En esencia, se empieza con un vector de ceros, y se va iterando por las ecuaciones y sobreescribiendo el valor actual de las variables. A medida que aumentan las iteraciones, disminuirá el error con respecto al resultado real del problema.

En caso de que en alguna iteración la diferencia entre el valor anterior y el actual sea menor a *tol* para todas las variables, se guarda ese valor y se sale de la función.

1.1. Test y convergencia

El código de test fue el siguiente:

```
legajo = 59378

rng = np.random.default_rng(legajo)
M = 5
A = rng.random((M,M))
b = rng.random((M))

for k in range(M):
    A[k,k] = (-1)**k * (A[k].sum()+k)

jacobi(A,b,legajo*1e-14)
```

```
(array([ 0.11525734,  0.01394051,  0.12640672, -0.14451684,  0.02226767]), 17)
```

Se puede observar que el algoritmo requirió 17 iteraciones, pero convergió a un resultado correcto (se verificó con $NumPy$). Esto se puede explicar planteando la condición de convergencia: el módulo de todos los autovalores de $I - B \cdot A$ debe ser menro a 1, donde B es la matriz inversa de la matriz diagonal con la misma diagonal que A.

Entonces, se tendrá:

```
B = np.diag(1 / np.diagonal(A))

mat = np.eye(*A.shape) - B.dot(A)

eigvals = np.linalg.eig(mat)[0]

print(np.abs(eigvals))
print(np.all(np.abs(eigvals) < 1))
```

```
[0.28491303 0.28491303 0.09368752 0.09368752 0.15222975]
True
```

Como se puede ver, todos los autovalores tienen módulo menor a 1, con lo cual el algoritmo de Jacobi convergerá.

2. Ejercicio 2

En este inciso, se buscó implementar un algoritmo que resuelva un sistema de ecuaciones diferenciales con el método de Taylor de orden 3. El algoritmo implementado fue el siguiente:

```
from scipy.special import factorial

def taylor3(dx, ddx, dddx, x_0, t_0, t_f, N):
    delta_t = (t_f - t_0) / (N - 1)

    x_k = np.zeros((N, len(x_0)))
    x_k[0] = x_0
    t_k = t_0 + delta_t * np.arange(N)

    for i in range(N - 1):
        derivs = np.asarray([
            [dx(t_k[i], x_k[i]),
             ddx(t_k[i], x_k[i]),
             dddx(t_k[i], x_k[i])]
        ]).T.reshape(-1, 3)

        expos = np.arange(1, 4)
        x_k[i + 1] = x_k[i] + derivs.dot(delta_t ** expos / factorial(expos))

    return t_k, x_k
```

El algoritmo iterará N - 1 veces (porque el valor de x_0 ya está fijado), implementando el algoritmo de Taylor de orden 3. Esto quiere decir que se hará:

$$x_{k+1} = x_k + \sum_{n=1}^{3} \left(\frac{f^{(n)}(t_k, x_k)}{n!} \cdot \Delta t^n \right) \tag{2}$$

2.1. Test

Para verificarlo, se usó el siguiente ejemplo:

```python
def dx(t, x): return x[0] + np.sin(t)
def ddx(t, x): return dx(t, x) + np.cos(t)
def dddx(t, x): return ddx(t, x) - np.sin(t)

x_0 = [-1/2]
t_0 = 0
t_f = 2
N = 100

t, x = taylor3(dx, ddx, dddx, x_0, t_0, t_f, N)

K = (x_0[0] + np.cos(t_0)/2 + np.sin(t_0)/2) * np.exp(-t_0)
real = K * np.exp(t) - np.sin(t) / 2 - np.cos(t) / 2

print(f'Error máximo: {np.abs(x.reshape(-1) - real).max().round(9)}')
```

```
Error máximo: 1.326e-06
```

Se puede observar que dio un error máximo bajo, con lo cual el algoritmo está dando un resultado correcto. Para verlo de forma más gráfica, se decidió agregar una imagen superpuesta de la respuesta real y la obtenida con Taylor.

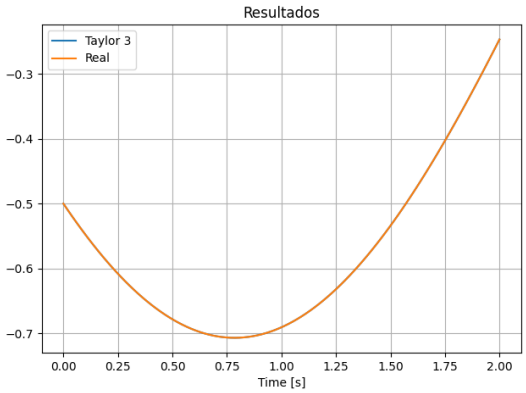

Figura 1: Solución

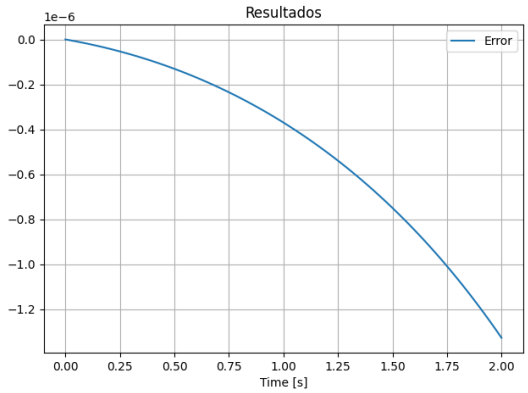

Figura 2: Error

Se puede ver que la solución tiene errores imperceptibles en esta escala, y que, al graficar estos errores, se encuentra que son del orden obtenido previamente, 10^{-6} en todo el rango.

2.2. Resolución de sistema de EDO

Finalmente, se busca resolver:

$$\frac{d^2y}{dt^2} = -y + e^{-t}, (t > -5) \tag{3}$$

Con:

$$y(-5) = \frac{legajo}{1000} = 59.378$$

$$\frac{dy(-5)}{dt} = 1$$

2.2.1. Garantía del error

Para resolverlo con un error menor a 10^{-6}, primero se hizo lo siguiente:

```
n = 3
maxerr = 1e-6

def dx(t, _x):
    k, x = _x
    return [-x + np.exp(-t), k]

def ddx(t, _x):
    k, x = _x
    return [-k - np.exp(-t), -x + np.exp(-t)]

def dddx(t, _x):
    k, x = _x
    return [x, -k - np.exp(-t)]

t_0, t_f = -5, 5
x_0 = [1, legajo / 1000]

N_from_delta = lambda delta, t_0, t_f: int(np.round((t_f - t_0)/delta + 1))

delta_t1 = .1

_, x1 = taylor3(dx, ddx, dddx, x_0, t_0, t_f, N_from_delta(delta_t1, t_0, t_f))
_, x2 = taylor3(dx, ddx, dddx, x_0, t_0, t_f, N_from_delta(delta_t1 / 2, t_0, t_f))

c = np.max(np.abs(x1[:, 1] - x2[::2, 1]) / (delta_t1**n * (1 - 1/2**n)))

delta_t = .85 * (maxerr / c)**(1/n)
print('Delta_t:', delta_t.round(5))
```

```
Delta_t: 0.0029
```

Utilizando la fórmula conocida del error en un método de orden n, se eligió un Δt_x arbitrario que convergiera y se resolvió al problema con Δt_x y $\frac{\Delta t_x}{2}$, obteniendo dos soluciones distintas. Luego, se buscó la constante c, que será:

$$c = max\left(\frac{|x_k^1 - x_{2k}^2|}{\Delta t_x^n \cdot (1 - 2^{-n})}\right) \tag{4}$$

Los exponentes 1 y 2 representan a la resolución Δt y $\frac{\Delta t}{2}$, respectivamente.

Con esto se obtiene la constante $c = 25.2415$. Luego se definirá el verdadero Δt como:

$$\Delta t = P \cdot \left(\frac{E_{max}}{c}\right)^{\frac{1}{n}} \tag{5}$$

Donde E_{max} es el máximo error permitido y P es un porcentaje de tolerancia, que en este caso se tomó como 0.85 (85 %). Y así se obtuvo el valor de 0.0029 de arriba.

2.2.2. Resultados

Con este resultado, se resolvió al problema tanto con Δt como con $\frac{\Delta t}{2}$. La solución obtenida fue la siguiente:

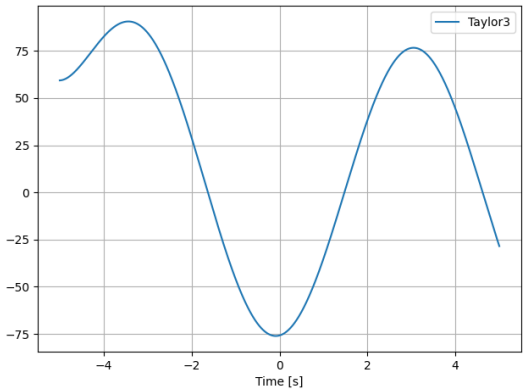

Figura 3: Solución

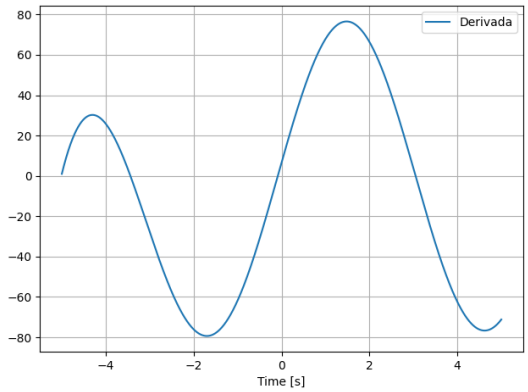

Figura 4: Derivada

Se puede ver que las condiciones iniciales se cumplen. Luego, para estimar el error, se utilizó la fórmula:

$$Error = max\left(\frac{|x^2_{2k} - x^1_k|}{1 - 2^{-n}}\right) \tag{6}$$

```
t, _x = taylor3(dx, ddx, dddx, x_0, t_0, t_f, N_from_delta(delta_t, t_0, t_f))
k, x = _x.T

_, _x2 = taylor3(dx, ddx, dddx, x_0, t_0, t_f, N_from_delta(delta_t/2, t_0, t_f))
k2, x2 = _x2.T

real_err = np.max(np.abs(x2[:x.size*2:2] - x) / (1 - 1/2**n))
derv_err = np.max(np.abs(k2[:x.size*2:2] - k) / (1 - 1/2**n))

print('x error:', real_err.round(10))
print('Deriv error:', derv_err.round(10))
```

```
x error: 6.11e-07
Deriv error: 7.331e-07
```

Como se puede ver, ambos son menores al máximo error pedido.

3. Ejercicio 3

En este ejercicio, se debe implementar un algoritmo para resolver un problema de cuadrados mínimos por el método de Cholesky. En particular, este problema será de sólo dos variables, y tiene como restricción el no usar ciclos for ni $while$.

El algoritmo implementado fue el siguiente:

```python
def cuadmin(A : np.array, b : np.array) -> np.array:
    h, w = A.shape
    A_T_A = A.T.dot(A)

    G = np.zeros((w, w))
    G[0, 0] = np.sqrt(A_T_A[0, 0])
    G[1, 0] = A_T_A[0, 1] / G[0, 0]
    G[1, 1] = np.sqrt(A_T_A[1, 1] - G[1, 0]**2)

    B = A.T.dot(b)
    res1 = np.zeros(B.shape)

    res1[0] = B[0] / G[0, 0]
    res1[1] = (B[1] - G[1, 0] * res1[0]) / G[1, 1]

    res2 = np.zeros(res1.shape)
    G2 = G.T
    res2[1] = res1[1] / G2[1, 1]
    res2[0] = (res1[0] - G2[0, 1] * res2[1]) / G2[0, 0]

    return res2, np.linalg.norm(A.dot(res2) - b)
```

En esencia, lo que se hizo fue escribir al sistema $A \cdot \vec{x} = \vec{b}$ de la forma $A^T \cdot A \cdot \vec{x} = \vec{b}$. Haciendo esto, luego se puede encontrar una matriz G triangular inferior tal que tal que $G \cdot G^T = A^T \cdot A = A\_T\_A$.

De esta manera, se tendrá que $A\_T\_A_{00} = G_{00}^2$, con lo cual obtenemos el valor de G_{00}. Luego, $A\_T\_A_{01} = G_{00} \cdot G_{10}$, de donde podemos despejar G_{10}. Finalmente, $A\_T\_A_{11} = G_{10}^2 + G_{11}^2$, de donde podemos despejar G_{11}. Nótese que no se usó $A\_T\_A_{10}$, porque la matriz es simétrica.

Luego de tener G, se procede a resolver al sistema en dos pasos:

$$G \cdot \vec{w} = \vec{B} \tag{7}$$

$$G^T \cdot \vec{x} = \vec{w} \tag{8}$$

Donde B $= A^T \cdot \vec{b}$. Como estos sistemas son triangulares (inferior y superior), son sencillos de resolver, y se implementó con una sustitución hacia atrás y hacia adelante *hardcodeadas* para 2 variables.

3.1. Test

Para probarlo, se utilizó el siguiente código:

```python
rng = np.random.default_rng(legajo)
M = 1000
A = rng.random((M,2))
b = legajo*A[:,0]+legajo/2*A[:,1]+rng.random((M))

res, err = cuadmin(A, b)

print('Resultado:', res.reshape(-1))
print('Norma del error:', err.round(4))
```

```
Resultado: [59378.45444798 29689.41918225]
Norma del error: 10.6138
```

Se verificó la solución con *SciPy* y la respuesta obtenida es correcta, tanto para los resultados como para la norma del error.