Matias Davila
Métodos Numéricos

UNIVERSITAT Carlemany

Ejercicio 1

Pregunta 1(a): Implementación de la Factorización LU

La factorización LU descompone una matriz *A* en dos matrices: una matriz triangular inferior que llamamos *L* y luego una matriz triangular superior llamada *U*. Este método es muy útil y frecuentemente usado para resolver sistemas de ecuaciones lineales. Esto se hace al transformar la matriz *A* en *U* mediante operaciones elementales, mientras que se registra o se guarda, cada paso de estas operaciones en *L*.

A = np.array([[1, -2, 3, 2],

[2, 5, -1, -3],

[2, -3, 0, 1],

[-1, 0, 3, 1]])

```python
import numpy as np

def factorizacion_LU(A):
    n = len(A)
    L = np.zeros_like(A)
    U = np.zeros_like(A)

    for j in range(n):
        L[j][j] = 1
        for i in range(j + 1):
            suma1 = sum(U[k][j] * L[i][k] for k in range(i))
            U[i][j] = A[i][j] - suma1
        for i in range(j, n):
            suma2 = sum(U[k][j] * L[i][k] for k in range(j))
            L[i][j] = (A[i][j] - suma2) / U[j][j]

    return L, U
```

Resultado:

```
Matriz L:
[[ 1.          0.          0.          0.        ]
 [ 2.          1.          0.          0.        ]
 [ 2.          0.11111111  1.          0.        ]
 [-1.         -0.22222222 -0.85106383  1.        ]]

Matriz U:
[[ 1.         -2.          3.          2.        ]
 [ 0.          9.         -7.         -7.        ]
 [ 0.          0.         -5.22222222 -2.22222222]
 [ 0.          0.          0.         -0.44680851]]
```

Matias Davila
Métodos Numéricos

UNIVERSITAT Carlemany

1(b): Implementación de la Rutina para la Matriz Inversa

Para calcular la matriz inversa A^{-1} de matriz no singular *A,* como se nos pide en la segunda parte del ejercicio, lo que haremos será resolver sistemas de ecuaciones lineales $A\mathbf{c}_i=\mathbf{e}_i$ para cada columna $\mathbf{c}_i$ de la matriz inversa, donde $\mathbf{e}_i$ es el vector canónico correspondiente. Utilizando la factorización LU obtenida en el apartado anterior, cada sistema se resuelve de manera más eficiente.

```python
def resolver_Ly_b(L, b):
    y = np.zeros_like(b)
    for i in range(len(b)):
        y[i] = b[i] - np.dot(L[i, :i], y[:i])
    return y

def resolver_Ux_y(U, y):
    x = np.zeros_like(y)
    for i in range(len(y) - 1, -1, -1):
        x[i] = (y[i] - np.dot(U[i, i+1:], x[i+1:])) / U[i, i]
    return x

def matriz_inversa(A):
    n = len(A)
    A_inv = np.zeros_like(A, dtype=float)
    L, U = factorizacion_LU(A)

    for i in range(n):
        e = np.zeros(n)
        e[i] = 1
        y = resolver_Ly_b(L, e)
        A_inv[:,i] = resolver_Ux_y(U, y)

    return A_inv
```

Resultado:

```
Matriz Inversa A^-1:
[[ 4.28571429e-01  1.42857143e-01 -4.76190476e-02 -3.80952381e-01]
 [ 1.00000000e+00 -4.93432455e-17 -1.00000000e+00 -1.00000000e+00]
 [-5.71428571e-01  1.42857143e-01  6.19047619e-01  9.52380952e-01]
 [ 2.14285714e+00 -2.85714286e-01 -1.90476190e+00 -2.23809524e+00]]
```

Matias Davila
Métodos Numéricos

UNIVERSITAT Carlemany

1(c): Implementación del Pivoteo Maximal por Columnas

Para mejorar la estabilidad numérica del método de Gauss, usaremos el método de pivoteo maximal por columnas. Este se trata de seleccional el mayor elemento absoluto en cada columna como un pivote mientas se hace la eliminación de Gauss, con esto estaríamos logrando reducir el error de redondeo en los cálculos.

```python
import numpy as np

def pivoteo_maximal(A, b):
    n = len(A)
    for i in range(n):
        # Encontrar el máximo elemento en la columna actual
        max_index = np.argmax(abs(A[i:n, i])) + i

        # Intercambiar filas en A y b
        A[[i, max_index]] = A[[max_index, i]]
        b[[i, max_index]] = b[[max_index, i]]

        # Verificar si el elemento diagonal es cero
        if A[i, i] == 0:
            raise ValueError("La matriz es singular o casi singular.")

        # Eliminación de Gauss
        for j in range(i+1, n):
            factor = A[j, i] / A[i, i]
            A[j, i:] -= factor * A[i, i:]
            b[j] -= factor * b[i]

    # Resolución hacia atrás
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        x[i] = (b[i] - np.dot(A[i, i+1:], x[i+1:])) / A[i, i]
    return x

# Ejemplo de uso
A = np.array([[1, -2, 3, 2],
              [2, 5, -1, -3],
              [2, -3, 0, 1],
              [-1, 0, 3, 1]], dtype=float)
b = np.array([1, 2, 3, 4], dtype=float)  # Debes definir el vector b según tu problema

x = pivoteo_maximal(A, b)
print("Solución del sistema:", x)
```

✓ 0.4s

```
Solución del sistema: [ -0.95238095  -6.          5.38095238 -13.0952381 ]
```

Matias Davila
Métodos Numéricos

UNIVERSITAT Carlemany

Ejercicio 2

Parte (a): Algoritmos de Jacobi y Gauss-Seidel para Matrices Tridiagonales

Para matrices tridiagonales, podemos adaptar los algoritmos de Jacobi y Gauss-Seidel con esto lograríamos que sean más eficientes, para esto estaremos aprovechando la estructura de la matriz. La mayoría de las operaciones involucra solo tres elementos por fila, lo que reduce la complejidad computacional.

```python
import numpy as np
def jacobi_tridiagonal(A, b, x0=None, tol=1e-10, max_iter=1000):
    n = len(A)
    x = np.zeros(n) if x0 is None else x0
    x_new = np.copy(x)

    for _ in range(max_iter):
        for i in range(n):
            suma = 0
            if i > 0:
                suma += A[i, i-1] * x[i-1]
            if i < n-1:
                suma += A[i, i+1] * x[i+1]

            x_new[i] = (b[i] - suma) / A[i, i]

        if np.linalg.norm(x_new - x, np.inf) < tol:
            return x_new

        x[:] = x_new

    raise ValueError("El método de Jacobi no converge en el número máximo de iteraciones")
def gauss_seidel_tridiagonal(A, b, x0=None, tol=1e-10, max_iter=1000):
    n = len(A)
    x = np.zeros(n) if x0 is None else x0

    for _ in range(max_iter):
        x_old = np.copy(x)
        for i in range(n):
            suma = 0
            if i > 0:
                suma += A[i, i-1] * x[i-1]
            if i < n-1:
                suma += A[i, i+1] * x_old[i+1]

            x[i] = (b[i] - suma) / A[i, i]

        if np.linalg.norm(x - x_old, np.inf) < tol:
            return x

    raise ValueError("El método de Gauss-Seidel no converge en el número máximo de iteraciones")
```

```python
# Definición de A y b para los ejemplos
A = np.array([[2, -1, 0, 0],
              [-1, 2, -1, 0],
              [0, -1, 2, -1],
              [0, 0, -1, 2]], dtype=float)
b = np.array([1, 0, 0, 1], dtype=float)
# Ejecución de los métodos
solucion_jacobi = jacobi_tridiagonal(A, b)
solucion_gauss_seidel = gauss_seidel_tridiagonal(A, b)

print("Solución Jacobi:", solucion_jacobi)
print("Solución Gauss-Seidel:", solucion_gauss_seidel)
```

✓ 0.1s

```
Solución Jacobi: [1. 1. 1. 1.]
Solución Gauss-Seidel: [1. 1. 1. 1.]
```

Matias Davila
Métodos Numéricos

UNIVERSITAT Carlemany

Parte (b): Resolución del Problema de la Viga

Para la resolución de este apartad, trabajaremos utilizando el método de las diferencias finitas, con esto logramos un aproximado, tomaremos las características dadas de la viga, puedes construir la matriz A_h y el vector R_h, y luego aplicar los algoritmos de Jacobi o Gauss-Seidel para encontrar la solución aproximada.

```python
import numpy as np

def gauss_seidel(A, b, x0=None, tol=1e-10, max_iter=5000):
    n = len(A)
    x = np.zeros(n) if x0 is None else x0.copy()

    for _ in range(max_iter):
        x_old = x.copy()
        for i in range(n):
            if A[i, i] == 0:
                raise ValueError("Elemento cero en la diagonal, el método no puede proceder.")

            x[i] = (b[i] - np.dot(A[i, :i], x[:i]) - np.dot(A[i, i+1:], x_old[i+1:])) / A[i, i]

        if np.linalg.norm(x - x_old, np.inf) < tol:
            return x

    raise ValueError("El método de Gauss-Seidel no converge en el número máximo de iteraciones")

# Propiedades de la viga
l = 120  # longitud en pulgadas
q = 100  # carga en lb/ft
E = 3e7  # módulo de elasticidad en lb/in^2
S = 1000 # tensión en lb
I = 625  # momento de inercia en in^4
h = 0.1  # tamaño del paso en pulgadas

# Número de puntos en la discretización
n = int(l / h) - 1

# Construcción de Ah
diagonal = np.full(n, -2 * S * h**2 / E / I)
off_diagonal = np.ones(n - 1)
Ah = np.diag(diagonal) + np.diag(off_diagonal, k=1) + np.diag(off_diagonal, k=-1)

# Construcción de Rh
x = np.linspace(h, l - h, n)
Rh = q * x * (x - l) * h**2 / (2 * E * I)

# Resolución del sistema usando Gauss-Seidel
x0 = np.zeros(n)  # Valor inicial para Gauss-Seidel
solucion = gauss_seidel(Ah, Rh, x0)
print("Solución aproximada:", solucion)
```

Matias Davila
Métodos Numéricos

Parte (c): Comparación con la Solución Exacta

Finalmente podremos generar una comparación de la solución aproximada con la solución exacta dada y con esto debemos calcular el error global máximo. Para esto, evaluamos ambas soluciones en una serie de puntos y grafica la diferencia entre ellas.

```python
import numpy as np
import matplotlib.pyplot as plt
# Función para la solución exacta
def solucion_exacta(x, c1, c2, a, b, c, l):
    return c1 * np.exp(a * x) + c2 * np.exp(-a * x) + b * (x - l) * x + c
# Parámetros de la viga y de la solución exacta
l = 120  # longitud en pulgadas
q = 100  # carga en lb/ft
E = 3e7  # módulo de elasticidad en lb/in^2
S = 1000 # tensión en lb
I = 625  # momento de inercia en in^4
h = 0.1  # tamaño del paso en pulgadas
c1 = 7.7042537e4
c2 = 7.9207462e4
a = 2.3094010e-4
b = -4.1666666e-3
c = -1.56249999e5

# Generar la matriz Ah y el vector Rh
n = int(l / h) - 1
diagonal = np.full(n, -2 * S * h**2 / E / I)
off_diagonal = np.ones(n - 1)
Ah = np.diag(diagonal) + np.diag(off_diagonal, k=1) + np.diag(off_diagonal, k=-1)
x_aprox = np.linspace(h, l - h, n)
Rh = q * x_aprox * (x_aprox - l) * h**2 / (2 * E * I)
# Calcula la solución aproximada
solucion = gauss_seidel(Ah, Rh)  # Asegúrate de que la función gauss_seidel esté definida
# Evaluar la solución exacta
x = np.linspace(0, l, num=1200)
solucion_ex = solucion_exacta(x, c1, c2, a, b, c, l)
# Interpolar la solución aproximada para los mismos puntos x
solucion_aprox = np.interp(x, x_aprox, solucion)
# Cálculo del error
error = solucion_ex - solucion_aprox
error_maximo = np.max(np.abs(error))
# Graficar las soluciones y el error
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(x, solucion_ex, label="Solución Exacta")
plt.plot(x, solucion_aprox, label="Solución Aproximada", linestyle='--')
plt.title("Comparación de Soluciones")
plt.xlabel("x")
plt.ylabel("ω(x)")
plt.legend()
plt.subplot(2, 1, 2)
plt.plot(x, error)
```