Práctica Dirigida 5

Análisis y Modelamiento Numérico I

Alumno:

- Chowdhury Gomez, Junal Johir 20200092K

13 de noviembre de 2024

Ejercicio 2

Enunciado

Programe el algoritmo de Gram-Schmidt y el algoritmo modificado de Gram-Schmidt y pruébelos para ver cuál es mejor. La primera prueba podría comprender una matriz de 20×10 con elementos aleatorios uniformemente distribuidos en el intervalo $[0, 1]$. La segunda prueba podría comprender una matriz de 20×10 con elementos generados por una función elemental, como por ejemplo

$$a_{ij} = \left(\frac{2i - 2j}{19}\right)^{j-1}$$

En cada caso, genera a partir de (A) una matriz (B) cuyas columnas deben ser ortogonales. Luego, examina $(B^T B)$ para ver cuán próxima está a la matriz identidad.

Solución:

Código en Python

```
import numpy as np
def gram_schmidt(A):
    """Gram-Schmidt"""
    (m, n) = A.shape
    Q = np.zeros((m, n))
    R = np.zeros((n, n))
    for j in range(n):
        v = A[:, j]
        for i in range(j):
            R[i, j] = np.dot(Q[:, i], A[:, j])
            v = v - R[i, j] * Q[:, i]
        R[j, j] = np.linalg.norm(v)
        Q[:, j] = v / R[j, j]
    return Q, R

def gram_schmidt_modificado(A):
    """Gram-Schmidt Modificado"""
    (m, n) = A.shape
    Q = np.zeros((m, n))
    R = np.zeros((n, n))
    V = A.copy()
    for i in range(n):
        R[i, i] = np.linalg.norm(V[:, i])
        Q[:, i] = V[:, i] / R[i, i]
        for j in range(i + 1, n):
            R[i, j] = np.dot(Q[:, i], V[:, j])
            V[:, j] = V[:, j] - R[i, j] * Q[:, i]
    return Q, R

# Test 1: Matrizz Random  20x10
A1 = np.random.uniform(0, 1, (20, 10))
Q1_Sin_Modificar, R1_Sin_Modificar = gram_schmidt(A1)
Q1_Modificado, R1_Modificado = gram_schmidt_modificado(A1)

# Test 2: Matriz con la ffuncion dada
A2 = np.array([[(2*i - 2*j) / 19 for j in range(1, 11)] for i in range(1, 21)])
Q2_Sin_Modificar, R2_Sin_Modificar = gram_schmidt(A2)
Q2_Modificado, R2_Modificado = gram_schmidt_modificado(A2)

# Funcion que compara resultados con mmatriz identidad
def comparar_con_identidad(Q):
    identidad_aproximada = np.dot(Q.T, Q)
    identidad = np.eye(Q.shape[1])
    return np.linalg.norm(identidad_aproximada - identidad)

# Comparacion
error_sin_modificar_1 = comparar_con_identidad(Q1_Sin_Modificar)
error_modificado_1 = comparar_con_identidad(Q1_Modificado)
error_sin_modificar_2 = comparar_con_identidad(Q2_Sin_Modificar)
error_modificado_2 = comparar_con_identidad(Q2_Modificado)
print(f"Error para Gram-Schmidt Sin modificar con una matriz random:
```

```
{error_sin_modificar_1}")
print(f"Error para Gram-Schmidt Modificado con una matriz random:
{error_modificado_1}")
print(f"Error para Gram-Schmidt Sin modificar con una funcion matriz:
{error_sin_modificar_2}")
print(f"Error para Gram-Schmidt Modificado con una funcion matriz:
{error_modificado_2}")
```

Salida del código

```
Error para Gram-Schmidt Sin modificar con una matriz random: 2.0820275844552317e-15
Error para Gram-Schmidt Modificado con una matriz random: 1.1114271526571848e-15
Error para Gram-Schmidt Sin modificar con una funcion matriz: 8.485066310265424
Error para Gram-Schmidt Modificado con una funcion matriz: 1.8621038345155194
```

Observación:

- La matriz A1, contiene valores aleatorios, los cuales pueden variar en la compilación.
- Se observa que los errores de Gram-Schmidt Modificado y sin modificar para la matriz con valores aleatorios tienden a cero, mientras que los errores para la matriz con la funcion dada tiene un margen de error considerado.

Ejercicio 6

Enunciado

Encuentre una función exponencial ($y = ae^{bx}$) que ajuste

x	0,4	1,2	1,6	2	2,3
y	800	975	1950	2900	3600

Solución:

Código en Python

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import linregress

# Datos proporcionados
x = np.array([0.4, 0.8, 1.2, 1.6, 2.0, 2.3])
y = np.array([800, 975, 1500, 1950, 2900, 3600])

# Transformar y usando logaritmo natural
log_y = np.log(y)

# Realizar regresion lineal
slope, intercept, r_value, p_value, std_err = linregress(x, log_y)

# Convertir coeficientes a la forma exponencial
a = np.exp(intercept)
b = slope

print("ln(y) = ln(a) + bx")
print("a: ",a)
print("b: ",b)

# Generar valores de x para la curva ajustada
x_fit = np.linspace(min(x), max(x), 100)
y_fit = a * np.exp(b * x_fit)

# Graficar datos originales y curva ajustada
plt.scatter(x, y, label='Datos originales', color='red')
plt.plot(x_fit, y_fit, label='Curva ajustada', color='blue')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('Ajuste de la funcion exponencial')
plt.grid(True)
plt.show()
```

Salida del código

```
ln(y) = ln(a) + bx
a:  546.5909394331758
b:  0.8186512283093649
```

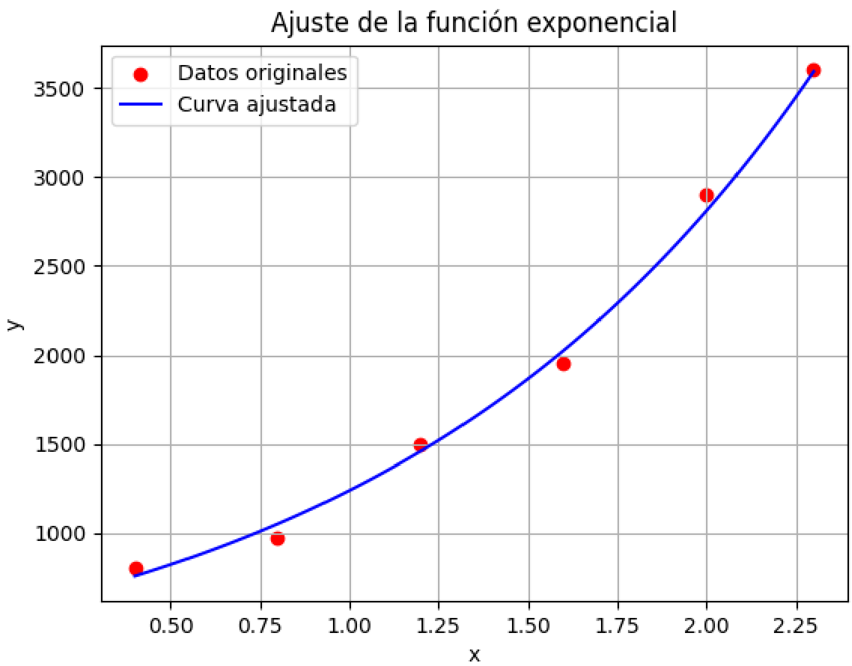

Figura 1: Ajuste de la funcion exponencial.

Observación:

- Observamos en la grafica que la funcion exponencial ($y = 546{,}59e^{0{,}8186x}$) se ajusta adecuadamente.

Ejercicio 8

Enunciado

Encuentre una función $k = \frac{ac^2}{b+c^2}$ que ajuste los siguientes datos:

c	k
0.5	1.1
0.8	2.4
1.5	5.3
2.5	7.6
4.0	8.9

Solución:

Código en Python

```
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# Datos proporcionados
c = np.array([0.5, 0.8, 1.5, 2.5, 4])
k = np.array([1.1, 2.4, 5.3, 7.6, 8.9])

# Definir la funcion de ajuste
def func(c, a, b):
    return (a * c**2) / (b + c**2)

# Usar curve_fit para encontrar los valores optimos de a y b
params, covariance = curve_fit(func, c, k, p0=[1, 1])
a, b = params
print("Para el ajuste: (a * c^2) / (b + c^2)")
print("a: ",a)
print("b: ",b)
# Generar valores de c para la curva ajustada
c_fit = np.linspace(min(c), max(c), 100)
k_fit = func(c_fit, a, b)

# Graficar datos originales y curva ajustada
plt.scatter(c, k, label='Datos originales', color='red')
plt.plot(c_fit, k_fit, label='Curva ajustada', color='blue')
plt.xlabel('c')
plt.ylabel('k')
plt.legend()
plt.title('Ajuste de la funcion $k = \\frac{ac^2}{b + c^2}$')
plt.grid(True)
plt.show()
```

Salida del código

```
Para el ajuste: (a * c^2) / (b + c^2)
a:  10.03574863392488
b:  2.0178928592262455
```

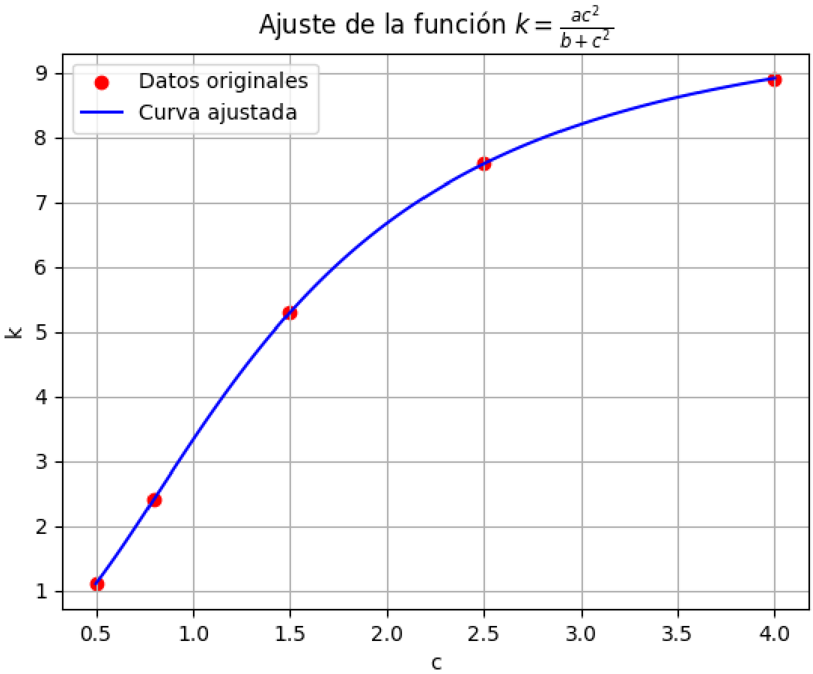

Figura 2: Ajuste de la funcion $k = \frac{ac^2}{b+c^2}$.

Observación:

- Observamos en la grafica que la funcion exponencial $k = \frac{10{,}0357c^2}{2{,}01789+c^2}$ se ajusta adecuadamente.

Ejercicio 13

Enunciado

Considere $A_{n\times n}$ una matriz tridiagonal con $a_{ii} = 2$, $a_{ij} = -1$ para $|i-j| < 2$ y B un vector columna tal que $b_n = 1$, $b_i = 0$ para $i < n$. Compruebe que $x_i = -\frac{n}{4} + \frac{i}{2}$ es la solución exacta del sistema $A\mathbf{x} = \mathbf{b}$. Resuelva el problema para $n = 20$, con el método SOR y el método de gradiente conjugado, compare el número de iteraciones realizadas.

Solución:

Primero, definimos la matriz A y el vector $\mathbf{b}$:

$$A = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 & 0 \\ -1 & 2 & -1 & \cdots & 0 & 0 \\ 0 & -1 & 2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 2 & -1 \\ 0 & 0 & 0 & \cdots & -1 & 2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

Queremos verificar que $\mathbf{x} = \left(-\frac{n}{4} + \frac{i}{2}\right)$ es la solución exacta. Entonces,

$$x_i = -\frac{n}{4} + \frac{i}{2}, \quad i = 1, 2, \ldots, n$$

Calculemos $A\mathbf{x}$:

$$(A\mathbf{x})_i = \begin{cases} 2x_1 - x_2 & \text{si } i = 1 \\ -x_{i-1} + 2x_i - x_{i+1} & \text{si } 2 \leq i \leq n-1 \\ -x_{n-1} + 2x_n & \text{si } i = n \end{cases}$$

Para $i = 1$:

$$2x_1 - x_2 = 2\left(-\frac{n}{4} + \frac{1}{2}\right) - \left(-\frac{n}{4} + 1\right) = -\frac{n}{2} + 1 + \frac{n}{4} - 1 = -\frac{n}{4}$$

Para $2 \leq i \leq n-1$:

$$-x_{i-1} + 2x_i - x_{i+1} = -\left(-\frac{n}{4} + \frac{i-1}{2}\right) + 2\left(-\frac{n}{4} + \frac{i}{2}\right) - \left(-\frac{n}{4} + \frac{i+1}{2}\right) = 0$$

Para $i = n$:

$$-x_{n-1} + 2x_n = -\left(-\frac{n}{4} + \frac{n-1}{2}\right) + 2\left(-\frac{n}{4} + \frac{n}{2}\right) = 1$$

Esto verifica que $A\mathbf{x} = \mathbf{b}$ y, por lo tanto, $\mathbf{x}$ es la solución exacta.

Resolución del problema para $n = 20$

Para resolver el sistema utilizando los métodos SOR y de gradiente conjugado, utilizamos algoritmos iterativos y comparamos el número de iteraciones necesarias.

Método SOR

El método SOR (Successive Over-Relaxation) es una extensión del método de Gauss-Seidel con un factor de relajación ω. Para el sistema $A\mathbf{x} = \mathbf{b}$, el método SOR se define como:

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right)$$

La elección de ω es crucial para la convergencia. Generalmente, $1 < \omega < 2$ se elige para acelerar la convergencia.

Método de gradiente conjugado

El método de gradiente conjugado es un método iterativo para resolver sistemas lineales simétricos y definidos positivos. Se define como:

1. Inicializar $\mathbf{x}^{(0)}$, $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$, $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$ 2. Para $k = 0, 1, 2, \ldots$ hasta la convergencia:

$$\alpha^{(k)} = \frac{\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)}}{\mathbf{p}^{(k)} \cdot A\mathbf{p}^{(k)}}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}$$

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha^{(k)}A\mathbf{p}^{(k)}$$

$$\beta^{(k)} = \frac{\mathbf{r}^{(k+1)} \cdot \mathbf{r}^{(k+1)}}{\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)}}$$

$$\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta^{(k)}\mathbf{p}^{(k)}$$

Comparación del número de iteraciones

Implementaremos ambos métodos y compararemos el número de iteraciones necesarias para alcanzar una tolerancia dada.

Implementación en Python

```python
import numpy as np

# Parametros
n = 20
tol = 1e-10
max_iter = 10000

# Matriz tridiagonal A
A = 2 * np.eye(n) - np.eye(n, k=1) - np.eye(n, k=-1)

# Vector b
b = np.zeros(n)
b[-1] = 1

# Solucion exacta
x_exact = -n / 4 + np.arange(1, n + 1) / 2

# Metodo SOR
def sor(A, b, omega, tol, max_iter):
    n = len(b)
    x = np.zeros(n)
    for k in range(max_iter):
        x_old = np.copy(x)
        for i in range(n):
            sigma = sum(A[i, j] * x[j] for j in range(n) if j != i)
            x[i] = (1 - omega) * x[i] + (omega / A[i, i]) * (b[i] - sigma)
        if np.linalg.norm(x - x_old) < tol:
            return x, k + 1
    return x, max_iter

# Metodo de gradiente conjugado
def gradiente_conjugado(A, b, tol, max_iter):
    x = np.zeros(len(b))
    r = b - A.dot(x)
    p = r.copy()
    rsold = np.dot(r, r)
    for k in range(max_iter):
        Ap = A.dot(p)
        alpha = rsold / np.dot(p, Ap)
        x = x + alpha * p
        r = r - alpha * Ap
        rsnew = np.dot(r, r)
        if np.sqrt(rsnew) < tol:
            return x, k + 1
        p = r + (rsnew / rsold) * p
```

```python
        rsold = rsnew
    return x, max_iter

# Ejecutar metodos
omega = 1.5
x_sor, iter_sor = sor(A, b, omega, tol, max_iter)
x_cg, iter_cg = gradiente_conjugado(A, b, tol, max_iter)

# Comparacion de iteraciones
print(f"SOR iteraciones: {iter_sor}")
print(f"Gradiente conjugado iteraciones: {iter_cg}")
```

Salida del código

```
SOR iteraciones: 305
Gradiente conjugado iteraciones: 20
```

Ejercicio 19

Enunciado

Las edades de tres hermanos tales que el quíntuplo de la edad del primero, más el cuádruplo de la edad del segundo, más el triple de la edad del tercero, es igual a 60. El cuádruplo de la edad del primero, más el triple de la edad del segundo, más el quíntuplo de la edad del tercero, es igual a 50. Y el triple de la edad del primero, más el quíntuplo de la edad del segundo, más el cuádruplo de la edad del tercero, es igual a 46. Determine la edad de los tres hermanos usando el programa desarrollado del gradiente conjugado.

Solución:

Para resolver este sistema de ecuaciones lineales, definimos x_1, x_2 y x_3 como las edades del primer, segundo y tercer hermano respectivamente. El sistema de ecuaciones se puede escribir como:

$$\begin{aligned} 5x_1 + 4x_2 + 3x_3 &= 60 \\ 4x_1 + 3x_2 + 5x_3 &= 50 \\ 3x_1 + 5x_2 + 4x_3 &= 46 \end{aligned}$$

Utilizando el método del gradiente conjugado para resolver el sistema, obtenemos las edades de los hermanos:

Código en Python

```python
import numpy as np
def gradiente_conjugado(A, b, tol=1e-10, max_iteraciones=1000):
    x = np.zeros_like(b)
    r = b - np.dot(A, x)
    p = r
    rsold = np.dot(r.T, r)
    for iteration in range(max_iteraciones):
        Ap = np.dot(A, p)
        alfa = rsold / np.dot(p.T, Ap)
        x = x + alfa * p
        r = r - alfa * Ap
        rsnew = np.dot(r.T, r)
        if np.sqrt(rsnew) < tol:
            return x, iteration + 1
        p = r + (rsnew / rsold) * p
        rsold = rsnew
    return x, max_iteraciones
# Matriz A y vector B del sistema de ecuaciones
A = np.array([[5, 4, 3],
              [4, 3, 5],
              [3, 5, 4]])
B = np.array([60, 50, 46])
# Resolver el sistema usando el metodo de Gradiente Conjugado
X, iterations = gradiente_conjugado(A, B)
for i in range(len(X)):
    print(f'x{i+1} = {X[i]}')
```

Salida del código

```
x1 = 9.000000000000004
x2 = 3.0000000000000036
x3 = 1.0000000000000042
```

Observación:

- El primer hermano tiene 9 años.
- El segundo hermano tiene 3 años.

- El tercer hermano tiene 1 años.

Ejercicio 32a

Enunciado

Determine la solución por mínimos cuadrados de los siguientes sistemas lineales:

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 &= 1 \\ 2x_1 + x_2 - x_3 + 3x_4 &= 0 \\ x_1 - 2x_2 + x_3 + x_4 &= -1 \end{aligned}$$

Solución:

Podemos escribir el sistema en forma matricial:

$$A\mathbf{x} = \mathbf{b} \quad \text{donde} \quad A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & 3 \\ 1 & -2 & 1 & 1 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$$

La solución por mínimos cuadrados se obtiene resolviendo la ecuación normal:

$$A^T A\mathbf{x} = A^T\mathbf{b}$$

Primero, calculamos $A^T A$ y $A^T\mathbf{b}$:

$$A^T = \begin{pmatrix} 1 & 2 & 1 \\ 1 & 1 & -2 \\ 1 & -1 & 1 \\ 1 & 3 & 1 \end{pmatrix}$$

$$A^T A = \begin{pmatrix} 6 & 3 & 0 & 7 \\ 3 & 6 & -4 & 3 \\ 0 & -4 & 3 & 0 \\ 7 & 3 & 0 & 11 \end{pmatrix}$$

$$A^T\mathbf{b} = \begin{pmatrix} 1 & 2 & 1 \\ 1 & 1 & -2 \\ 1 & -1 & 1 \\ 1 & 3 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 3 \\ 0 \\ 0 \end{pmatrix}$$

Por lo tanto, el sistema a resolver es:

$$\begin{pmatrix} 6 & 3 & 0 & 7 \\ 3 & 6 & -4 & 3 \\ 0 & -4 & 3 & 0 \\ 7 & 3 & 0 & 11 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 3 \\ 0 \\ 0 \end{pmatrix}$$

Observación:

- Resolviendo este sistema de ecuaciones, obtenemos las soluciones:

$$x_1 = 0, \quad x_2 = \frac{1}{3}, \quad x_3 = \frac{1}{3}, \quad x_4 = -\frac{1}{3}$$

- Por lo tanto, la solución por mínimos cuadrados es:

$$\mathbf{x} = \begin{pmatrix} 0 \\ \frac{1}{3} \\ \frac{1}{3} \\ -\frac{1}{3} \end{pmatrix}$$