Ejercicios Avanzados: Python con NumPy y Matemáticas

Maestría en Investigación en Ciencia de Datos — FCFM, BUAP

Ejercicio 1: Método de Newton-Raphson Vectorizado

El método de Newton-Raphson aproxima raíces de una función f(x) iterando:

$$x_{n+1} = x_n - f(x_n) / f'(x_n)$$

Implementa una versión **completamente vectorizada** (sin ciclos sobre puntos iniciales) de este método que:

a) Reciba una función f, su derivada df, un arreglo de puntos iniciales x0, una tolerancia tol y un máximo de iteraciones max_iter.

b) Itere simultáneamente sobre todos los puntos iniciales usando operaciones NumPy.

c) Detenga la iteración de cada punto individualmente cuando |f(x)| < tol, usando np.where() para actualizar solo los puntos que aún no convergen.

d) Aplica el método para encontrar todas las raíces reales de:

$$f(x) = x^5 - 5x^3 + 4x \text{ en el intervalo } [-3, 3]$$

e) Grafica f(x) y señala las raíces encontradas con marcadores.

```
x0 = np.linspace(-3, 3, 50)  # 50 puntos iniciales
f  = lambda x: x**5 - 5*x**3 + 4*x
df = lambda x: 5*x**4 - 15*x**2 + 4
```

Funciones sugeridas: np.where(), np.abs(), np.unique() con atol para agrupar raíces, np.isclose()

■ Reto extra: reporta cuántas iteraciones necesitó cada raíz para converger.

Ejercicio 2: Descomposición QR y Proceso de Gram-Schmidt

Toda matriz A de rango completo puede descomponerse como A = QR, donde Q es ortogonal ($Q^TQ = I$) y R es triangular superior. Implementa desde cero el **proceso de Gram-Schmidt modificado** para obtener esta descomposición:

a) Para cada columna a_j de A, calcula el vector ortogonal:

$$v_j = a_j - \Sigma_{i<j} (q_i \cdot a_j) q_i$$

b) Normaliza: $q_j = v_j / \|v_j\|$

c) Construye R como $R_{ij} = q_i \cdot a_j$

d) Verifica tu implementación comparando con np.linalg.qr() usando np.allclose().

e) Verifica que $Q^TQ = I$ y que A = QR se cumplen con precisión numérica.

f) Usa tu descomposición QR para resolver el sistema Ax = b sin usar np.linalg.solve().

```
np.random.seed(0)
A = np.random.randint(1, 10, (4, 4)).astype(float)
b = np.array([1., 2., 3., 4.])
```

Funciones sugeridas: np.dot(), np.linalg.norm(), np.zeros_like(), np.linalg.qr()

■ Reto extra: implementa la sustitución hacia atrás para resolver $Rx = Q^T b$ sin np.linalg.solve().

Ejercicio 3: Ecuación de Calor 1D con Diferencias Finitas

La ecuación de calor en una dimensión describe cómo evoluciona la temperatura u(x,t) a lo largo del tiempo:

$$\partial u/\partial t = \alpha \cdot \partial^2 u/\partial x^2$$

Usando el **esquema explícito de diferencias finitas** (método de Euler hacia adelante), la actualización en cada paso de tiempo es:

$$u_i^{n+1} = u_i^n + r\cdot(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

donde $r = \alpha\cdot\Delta t/(\Delta x)^2$. El esquema es estable si $r \leq 0.5$.

Implementa la simulación entera usando operaciones vectorizadas de NumPy (sin ciclos sobre i). La función debe:

a) Discretizar el dominio $x \in [0, 1]$ con nx puntos y el tiempo $t \in [0, T]$ con nt pasos.

b) Usar condición inicial: $u(x, 0) = \sin(\pi x)$

c) Condiciones de frontera fijas: $u(0, t) = u(1, t) = 0$

d) Graficar la solución u(x,t) en varios instantes de tiempo en la misma figura.

e) Comparar con la solución analítica: $u(x,t) = e^{-\pi^2\alpha t} \cdot \sin(\pi x)$

```
alpha = 0.01   # difusividad térmica
nx = 50        # puntos espaciales
T  = 2.0       # tiempo total
dt = 0.001     # paso de tiempo
```

Funciones sugeridas: np.linspace(), np.sin(), np.exp(), slicing u[1:-1], u[2:], u[:-2]

■ Reto extra: repite con r > 0.5 y documenta qué ocurre numéricamente.

Ejercicio 4: Análisis de Componentes Principales (PCA) desde Cero

El PCA transforma un conjunto de datos a un nuevo sistema de coordenadas que maximiza la varianza. Implementa PCA **desde cero usando NumPy** (sin sklearn) siguiendo estos pasos:

a) Centra los datos restando la media de cada variable (columna).

b) Calcula la matriz de covarianza: $\Sigma = (1/n)\ X^T X$

c) Calcula los valores y vectores propios de Σ usando np.linalg.eigh().

d) Ordena los vectores propios de mayor a menor varianza explicada.

e) Proyecta los datos sobre los k primeros componentes principales.

f) Calcula el porcentaje de varianza explicada por cada componente.

g) Grafica un scatter plot de los datos proyectados en las 2 primeras componentes, coloreando por clase.

```
# Dataset: Iris (cargar con sklearn solo para obtener los datos)
from sklearn.datasets import load_iris
```

```
data = load_iris()
X = data.data          # matriz 150 x 4
y = data.target        # etiquetas de clase

# Tu implementación de PCA debe ir aquí (sin usar sklearn.decomposition)
```

Funciones sugeridas: np.mean(), np.linalg.eigh(), np.argsort(), np.dot(), plt.scatter()

■ Reto extra: verifica que tu PCA produce los mismos componentes que sklearn.decomposition.PCA.

Ejercicio 5: Transformada Discreta de Fourier y Filtrado de Señales

La Transformada Discreta de Fourier (DFT) descompone una señal en sus frecuencias componentes:

$$X[k] = \Sigma_{n=0}^{N-1} x[n] \cdot e^{(-j \cdot 2\pi \cdot k \cdot n/N)}$$

Implementa el siguiente pipeline completo de análisis y filtrado de señales:

a) Genera una señal compuesta por tres frecuencias más ruido gaussiano:

```
fs = 1000          # frecuencia de muestreo (Hz)
t  = np.linspace(0, 1, fs, endpoint=False)
senal = (3.0 * np.sin(2*np.pi*50*t) +    # componente a 50 Hz
         1.5 * np.sin(2*np.pi*120*t) +   # componente a 120 Hz
         0.8 * np.sin(2*np.pi*300*t))    # componente a 300 Hz
np.random.seed(7)
senal_ruidosa = senal + 1.2 * np.random.randn(len(t))
```

b) Calcula la DFT de la señal ruidosa usando np.fft.fft().

c) Grafica el espectro de frecuencias (magnitud vs frecuencia en Hz) e identifica visualmente los picos en 50, 120 y 300 Hz.

d) Implementa un **filtro pasa-bajas** en el dominio de frecuencia: anula todos los coeficientes de Fourier cuya frecuencia supere un umbral f_corte = 200 Hz.

e) Reconstruye la señal filtrada con np.fft.ifft() y grafícala junto a la señal original.

f) Calcula la Relación Señal-Ruido (SNR) antes y después del filtrado:

$$SNR = 10 \cdot \log\blacksquare\blacksquare(Var(señal\_limpia) / Var(ruido))$$

Funciones sugeridas: np.fft.fft(), np.fft.ifft(), np.fft.fftfreq(), np.abs(), np.var(), np.log10()

■ Reto extra: implementa también un filtro pasa-bandas que conserve solo la frecuencia de 120 Hz ± 10 Hz.

Nota: La funcionalidad del código vale 70% y la calidad del código 30%.