Redes Neuronales (2025)

Ayudas e indicaciones para la Guía de ejercicios N°3 (Trabajo en progreso)*

Índice

1. Ejercicio 1: Importando librerías **3**
- 1.1. Explicación de las Importaciones Principales . . . 3
- 1.2. `import numpy as np` . . . 3
- 1.3. `import scipy as sp` . . . 3
- 1.4. `import scipy.linalg as linalg` . . . 3
- 1.5. `import matplotlib.pyplot as plt` . . . 3

2. Ejercicio 2: Inversa de una matriz **4**
- 2.1. Calcular la inversa con `linalg.inv()` . . . 4
- 2.2. Verificar que $A^{-1}A \approx \mathbb{I}$ . . . 4
- 2.3. Verificar que $AA^{-1} \approx \mathbb{I}$ . . . 5

3. Ejercicio 3: Sistema de ecuaciones lineales **5**
- 3.1. Resolver el sistema con `linalg.solve()` . . . 5
- 3.2. Verificar que $Ax = b$ . . . 6

4. Ejercicio 4: Cuadrados Mínimos con `SciPy` **6**
- 4.1. Encontrar los coeficientes del polinomio . . . 6
- 4.2. Evaluar la bondad del ajuste con un gráfico . . . 7

5. Ejercicio 5: Pseudo-inversa de Moore-Penrose **8**
- 5.1. 1) Crear una matriz aleatoria . . . 8
- 5.2. 2) Calcular la pseudo-inversa con `linalg.pinv()` . . . 9
- 5.3. 3) Dimensiones de la pseudo-inversa A^+ . . . 9
- 5.4. 4) Verificar la propiedad $AA^+A \approx A$ . . . 9
- 5.5. 5) Verificar la propiedad $A^+AA^+ \approx A^+$ . . . 9

6. Ejercicio 5: Pseudo-inversa de Moore-Penrose (solución alternativa) **10**

7. Ejercicio 6: Producto de Kronecker **10**
- 7.1. Caso 1: Matriz y Vector Columna . . . 11
- 7.2. Caso 2: Matriz y Vector Fila . . . 12

*Reportar errores a: tristan.osan@unc.edu.ar

8. Ejercicio 7: Autovalores y Autovectores **12**
8.1. Calcular solo los autovalores con `linalg.eigvals()` 12
8.2. Calcular autovalores y autovectores a derecha con `linalg.eig()` 13
8.3. Verificar la ecuación a derecha ($Av = \lambda v$) 13
8.4. Calcular autovalores y autovectores a izquierda 14
8.5. Verificar la ecuación a izquierda ($uA = \mu u$) 14

9. Ejercicio 8: Singular Value Decomposition (SVD) **15**
9.1. Crear una matriz compleja aleatoria 15
9.2. Calcular la SVD e inspeccionar las dimensiones 16
9.3. Construir la matriz diagonal D 16
9.4. Verificar la Ecuación de Reconstrucción 16

Algebra lineal con `SciPy`

1. Ejercicio 1: Importando librerías

1.1. Explicación de las Importaciones Principales

En Python, la sentencia `import` se utiliza para cargar bibliotecas (o módulos) externas que nos dan acceso a nuevas funciones. A menudo se les asigna un **alias** (un apodo más corto) con la palabra clave `as` para que el código sea más limpio y rápido de escribir.

1.2. `import numpy as np`

```
import numpy as np
```

Esta línea importa la biblioteca **NumPy**, que es el pilar de la computación numérica y científica en Python. Su principal característica es el objeto `array`, una estructura de datos muy eficiente para trabajar con vectores y matrices. El alias `np` es la convención estándar utilizada por toda la comunidad de desarrolladores.

1.3. `import scipy as sp`

```
import scipy as sp
```

Importa la biblioteca **SciPy**, que está construida sobre NumPy y provee una vasta colección de algoritmos para matemáticas, ciencia e ingeniería. Mientras NumPy se enfoca en la estructura del `array`, SciPy la utiliza para realizar operaciones complejas de optimización, álgebra lineal, procesamiento de señales, etc.

1.4. `import scipy.linalg as linalg`

```
import scipy.linalg as linalg
```

Esta es una forma más específica y recomendada de importar. En lugar de cargar toda la biblioteca SciPy, aquí solo importamos el submódulo `linalg` (de **Lin**ear **Alg**ebra). Este módulo contiene funciones especializadas en álgebra lineal, como el cálculo de determinantes, inversas o autovalores de una matriz.

1.5. `import matplotlib.pyplot as plt`

```
import matplotlib.pyplot as plt
```

Importa el módulo `pyplot` de la biblioteca **Matplotlib**, la herramienta por excelencia para la creación de gráficos y visualizaciones de datos en Python. El módulo `pyplot` ofrece una interfaz sencilla para crear figuras, ejes y todo tipo de gráficos (líneas, barras, histogramas, etc.). El alias `plt` es la convención universal para este módulo.

2. Ejercicio 2: Inversa de una matriz

2.1. Calcular la inversa con `linalg.inv()`

La función `linalg.inv()` del módulo `scipy.linalg` calcula la inversa de una matriz cuadrada. La inversa A^{-1} de una matriz A es aquella que al multiplicarla por A da como resultado la matriz identidad $\mathbb{I}$.

Listing 1: Definición e inversión de la matriz A

```
import numpy as np
from scipy import linalg

# 1) Definimos la matriz A
A = np.array([[1, 2],
              [3, 4]])

# Calculamos su inversa
A_inv = linalg.inv(A)

print("Matriz A:\n", A)
print("\nInversa de A (A_inv):\n", A_inv)

# --- RESULTADO ESPERADO ---
# Matriz A:
#  [[1 2]
#   [3 4]]
#
# Inversa de A (A_inv):
#  [[-2.   1. ]
#   [ 1.5 -0.5]]
```

2.2. Verificar que $A^{-1}A \approx \mathbb{I}$

Para confirmar que hemos calculado la inversa correctamente, podemos multiplicarla por la matriz original. El resultado debe ser (aproximadamente) la matriz identidad. Usamos `np.dot()` para realizar la multiplicación de matrices.

```
# 2) Chequeamos que A_inv * A es la identidad
identidad_check1 = np.dot(A_inv, A)

print("Resultado de A_inv * A:\n", identidad_check1)

# --- RESULTADO ESPERADO ---
# Resultado de A_inv * A:
#  [[1. 0.]
#   [0. 1.]]
```

Explicación: El resultado es la matriz identidad de 2x2. Usamos el símbolo de aproximación ($\approx$) porque en cálculos con números de punto flotante (floats) más complejos, los resultados podrían no ser exactamente 0 o 1, sino números muy cercanos como `1.11e-16` en lugar de 0, por ejemplo.

2.3. Verificar que $AA^{-1} \approx \mathbb{I}$

La propiedad conmutativa también debe cumplirse, es decir, A multiplicada por su inversa A^{-1} también debe resultar en la matriz identidad.

```
# 3) Chequeamos que A * A_inv es la identidad
identidad_check2 = np.dot(A, A_inv)

print("Resultado de A * A_inv:\n", identidad_check2)

# --- RESULTADO ESPERADO ---
# Resultado de A * A_inv:
#  [[1. 0.]
#   [0. 1.]]
```

Como se esperaba, el resultado es nuevamente la matriz identidad, lo que confirma que `linalg.inv()` calculó la inversa de forma correcta.

3. Ejercicio 3: Sistema de ecuaciones lineales

3.1. Resolver el sistema con `linalg.solve()`

Para un sistema de ecuaciones lineales representado en su forma matricial $Ax = b$, donde A es una matriz de coeficientes y b es un vector de resultados, la función `linalg.solve(A, b)` encuentra el vector de incógnitas x.

Listing 2: Resolución de un sistema de ecuaciones lineales

```
import numpy as np
from scipy import linalg

# 1) Definimos la matriz A y el vector b
A = np.array([[3, 2, 0],
              [1, -1, 0],
              [0, 5, 1]])

b = np.array([2, 4, 1])

# Resolvemos para x
x = linalg.solve(A, b)

print("Matriz A:\n", A)
print("\nVector b:\n", b)
print("\nSolucion x:\n", x)

# --- RESULTADO ESPERADO ---
# Matriz A:
#  [[ 3  2  0]
#   [ 1 -1  0]
#   [ 0  5  1]]
#
# Vector b:
#  [2 4 1]
#
```

```
# Solucion x:
#  [ 2. -2. 11.]
```

3.2. Verificar que $Ax = b$

Para comprobar que la solución es correcta, realizamos la multiplicación de la matriz original A por el vector solución x. El resultado de esta operación debe ser el vector original b.

```
# 2) Chequeamos el resultado calculando A * x
resultado = np.dot(A, x)

print("Resultado de A * x:\n", resultado)

# Comprobamos programaticamente si el resultado es igual a b
# Se usa np.allclose() para comparar floats de forma segura
print("\nLa solucion es correcta?:", np.allclose(resultado, b))


# --- RESULTADO ESPERADO ---
# Resultado de A * x:
#  [2. 4. 1.]
#
# La solucion es correcta?: True
```

Explicación: El producto de la matriz de coeficientes A y el vector solución x nos devuelve el vector de resultados b original, lo que confirma que el sistema se resolvió correctamente.

4. Ejercicio 4: Cuadrados Mínimos con `SciPy`

El método de cuadrados mínimos es una técnica estándar para encontrar la "mejor" curva que se ajusta a un conjunto de puntos de datos. La idea es minimizar la suma de las distancias verticales al cuadrado entre cada punto de datos y el punto correspondiente en la curva ajustada. La función `linalg.lstsq()` de SciPy es la herramienta ideal para este propósito.

4.1. Encontrar los coeficientes del polinomio

Para ajustar el polinomio $p(x) = c_0 + c_1x + c_2x^2$ a los datos, primero debemos construir la matriz X (conocida como matriz de Vandermonde) y luego usar `linalg.lstsq()` para resolver el sistema y encontrar los coeficientes c_0, c_1, c_2.

Listing 3: Cálculo de coeficientes por cuadrados mínimos

```
import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt

# Datos del problema
x_data = np.array([1.0, 2.5, 3.5, 4.0, 5.0, 7.0, 8.5])
y_data = np.array([0.3, 1.1, 1.5, 2.0, 3.2, 6.6, 8.6])
```

```
# 1) Construimos la matriz X donde cada columna es una potencia de x
# Columna 0: x^0 (unos), Columna 1: x^1 (x), Columna 2: x^2
X = np.column_stack([x_data**0, x_data**1, x_data**2])

# Usamos linalg.lstsq para encontrar el vector de coeficientes 'c'
# La funcion devuelve varias cosas, pero solo nos interesa la primera:
    la solucion
c = linalg.lstsq(X, y_data)[0]

print("Matriz X (primeras 5 filas):\n", X[:5])
print("\nVector de coeficientes c [c0, c1, c2]:\n", c)

# --- RESULTADO ESPERADO ---
# Matriz X (primeras 5 filas):
#  [[ 1.     1.     1.   ]
#   [ 1.     2.5    6.25 ]
#   [ 1.     3.5   12.25 ]
#   [ 1.     4.    16.   ]
#   [ 1.     5.    25.   ]]
#
# Vector de coeficientes c [c0, c1, c2]:
# [0.0578403  0.07701453 0.11262261]
```

Explicación: El vector `c` contiene los coeficientes que buscábamos. El polinomio que mejor se ajusta a los datos es aproximadamente $p(x) = 0{,}058 - 0{,}077x + 0{,}1136x^2$.

4.2. Evaluar la bondad del ajuste con un gráfico

Una forma visual de comprobar si el ajuste es bueno es graficar los puntos de datos originales junto con la curva del polinomio que hemos encontrado.

Listing 4: Graficando el ajuste del polinomio

```
# 2) Graficamos los resultados

# Creamos un rango suave de valores de x para dibujar la curva
x_fit = np.linspace(x_data.min(), x_data.max(), 100)

# Calculamos los valores 'y' del polinomio para cada x_fit
y_fit = c[0] + c[1] * x_fit + c[2] * x_fit**2

# Graficamos los puntos de datos originales como circulos ('o')
plt.plot(x_data, y_data, 'o', label='Datos Originales')

# Graficamos el polinomio ajustado como una linea continua ('-')
plt.plot(x_fit, y_fit, '-', label='Ajuste Polinomial')

# Agregamos etiquetas y leyenda para mayor claridad
plt.xlabel("x")
plt.ylabel("y")
plt.title("Ajuste de Polinomio por Cuadrados Mínimos")
plt.legend()
plt.grid(True)
plt.show()
```

Resultado Esperado: Al ejecutar este código, se generará una ventana con un gráfico. El gráfico mostrará los puntos de datos discretos en azul y una curva parabólica en naranja que pasa a través de ellos, demostrando un buen ajuste a la tendencia general de los datos.

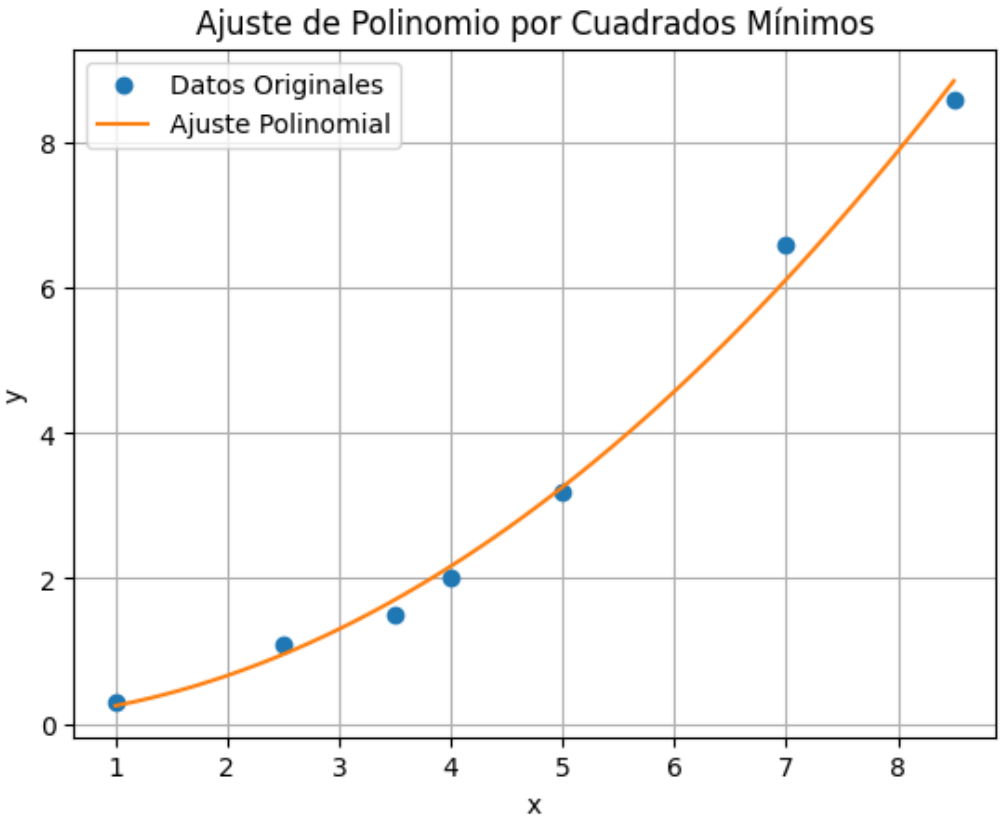

Figura 1: Ajuste de un polinomio de grado 2 a los datos mediante el método de cuadrados mínimos.

5. Ejercicio 5: Pseudo-inversa de Moore-Penrose

La pseudo-inversa es una generalización de la inversa de una matriz para matrices que no son cuadradas o no son invertibles. Es una herramienta muy importante en el álgebra lineal numérica, especialmente para resolver sistemas de ecuaciones lineales por cuadrados mínimos que no tienen una solución única.

5.1. 1) Crear una matriz aleatoria

Primero, creamos una matriz A de 9×6 con entradas aleatorias muestreadas de una distribución normal estándar (media 0, varianza 1) utilizando la función `np.random.randn()`.

Listing 5: Creación de una matriz aleatoria no cuadrada

```
import numpy as np
from scipy import linalg

# 1) Creamos la matriz A de 9x6
A = np.random.randn(9, 6)
```

```
print(f"La forma de la matriz A es: {A.shape}")

# --- RESULTADO ESPERADO ---
# La forma de la matriz A es: (9, 6)
```

5.2. 2) Calcular la pseudo-inversa con `linalg.pinv()`

La función `linalg.pinv()` calcula la pseudo-inversa de Moore-Penrose de una matriz.

```
# 2) Calculamos la pseudo-inversa de A
A_plus = linalg.pinv(A)


# 3) Dimensiones de la pseudo-inversa A+
print(f"La forma de la pseudo-inversa A+ es: {A_plus.shape}")

# --- RESULTADO ESPERADO ---
# La forma de la pseudo-inversa A+ es: (6, 9)
```

5.3. 3) Dimensiones de la pseudo-inversa A^+

Como se puede observar en el resultado anterior, si una matriz A tiene dimensiones $n \times m$, su pseudo-inversa A^+ tendrá las dimensiones transpuestas, es decir, $m \times n$. En nuestro caso, A es de 9×6, por lo que A^+ es de 6×9.

5.4. 4) Verificar la propiedad $AA^+A \approx A$

Una de las propiedades que definen a la pseudo-inversa es que $AA^+A = A$. Usaremos la multiplicación de matrices (con el operador `@`) y la función `np.allclose()` para verificar esta igualdad de forma numérica.

```
# 4) Verificamos la primera propiedad. El operador @ denota
   multiplicacion de matrices.
A_check_1 = A @ A_plus @ A

# np.allclose() compara dos arrays elemento a elemento viendo si son
   muy cercanos
son_cercanos_1 = np.allclose(A, A_check_1)

print(f"Se cumple que A * A+ * A es aproximadamente A?: {
   son_cercanos_1}")

# --- RESULTADO ESPERADO ---
# Se cumple que A * A+ * A es aproximadamente A?: True
```

5.5. 5) Verificar la propiedad $A^+AA^+ \approx A^+$

La segunda propiedad fundamental es que $A^+AA^+ = A^+$. La verificamos de la misma manera.

```python
# 5) Verificamos la segunda propiedad
A_plus_check = A_plus @ A @ A_plus

son_cercanos_2 = np.allclose(A_plus, A_plus_check)

print(f"Se cumple que A+ * A * A+ es aproximadamente A+?: {
    son_cercanos_2}")

# --- RESULTADO ESPERADO ---
# Se cumple que A+ * A * A+ es aproximadamente A+?: True
```

6. Ejercicio 5: Pseudo-inversa de Moore-Penrose (solución alternativa)

```python
# 5.1)
import numpy as np

# Dimensiones
n, m = 9, 6

# Crear la matriz A
A = np.random.randn(n, m)

print("Matriz A:")
print(A)

# 5.2) Calcular la pseudo-inversa de Moore-Penrose de A
A_plus = np.linalg.pinv(A)

print("Pseudo-inversa de A (A+):")
print(A_plus)

# 5.3) Dimensiones de A+
dim_A_plus = A_plus.shape
print(f"Dimensiones de A+: {dim_A_plus}")

# 5.4) Verificar que A*(A+)*A = A
check_A = np.allclose(np.dot(A, np.dot(A_plus, A)), A)
print(f"A*(A+)*A = A?: {check_A}")

# 5.5) Verificar que (A+)*A*(A+) = A+
check_A_plus = np.allclose(np.dot(A_plus, np.dot(A, A_plus)), A_plus)
print(f"(A+)*A*(A+) = A+?: {check_A_plus}")
```

7. Ejercicio 6: Producto de Kronecker

El producto de Kronecker, denotado por $A \otimes B$, es una operación entre dos matrices de tamaños arbitrarios que resulta en una matriz de bloques más grande. Esencialmente,

consiste en multiplicar cada elemento de la primera matriz por la segunda matriz completa.

7.1. Caso 1: Matriz y Vector Columna

Calculamos el producto de Kronecker entre una matriz A de 2×2 y una matriz B (vector columna) de 3×1.

Listing 6: Producto de Kronecker con un vector columna

```
import numpy as np
from scipy import linalg

# 1) Definimos las matrices A y B
A = np.array([[1, 2],
              [3, 4]])

B = np.array([[1],
              [1],
              [1]])

# Calculamos el producto de Kronecker
kron_product_1 = np.kron(A, B)

print("Matriz A (shape 2x2):\n", A)
print("\nMatriz B (shape 3x1):\n", B)
print("\nProducto de Kronecker A_k_B (caso 1):\n", kron_product_1)
print(f"\nLa forma de A prodkron B es: {kron_product_1.shape}")

# --- RESULTADO ESPERADO ---
# Matriz A (shape 2x2):
#  [[1 2]
#   [3 4]]
#
# Matriz B (shape 3x1):
#  [[1]
#   [1]
#   [1]]
#
# Producto de Kronecker A prodkron B (caso 1):
#  [[1 2]
#   [1 2]
#   [1 2]
#   [3 4]
#   [3 4]
#   [3 4]]
#
# La forma de A prodkron B es: (6, 2)
```

Dimensiones de $A \otimes B$: Si las dimensiones de A son $n \times m$ y las de B son $r \times s$, las dimensiones del producto de Kronecker son $(nr) \times (ms)$. En este caso, A es 2×2 y B es 3×1, por lo que las dimensiones del resultado son $(2 \times 3) \times (2 \times 1)$, es decir, 6×2.

7.2. Caso 2: Matriz y Vector Fila

Ahora repetimos el cálculo, pero con una matriz B (vector fila) de 1×3.

Listing 7: Producto de Kronecker con un vector fila

```
# 2) Definimos la nueva matriz B
B_row = np.array([[1, 1, 1]])

# Calculamos el producto de Kronecker
kron_product_2 = np.kron(A, B_row)

print("Matriz A (shape 2x2):\n", A)
print("\nMatriz B (shape 1x3):\n", B_row)
print("\nProducto de Kronecker A_k_B (caso 2):\n", kron_product_2)
print(f"\nLa forma de A_k_B es: {kron_product_2.shape}")

# --- RESULTADO ESPERADO ---
# Matriz A (shape 2x2):
#  [[1 2]
#   [3 4]]
#
# Matriz B (shape 1x3):
#  [[1 1 1]]
#
# Producto de Kronecker A_k_B (caso 2):
#  [[1 1 1 2 2 2]
#   [3 3 3 4 4 4]]
#
# La forma de A_k_B es: (2, 6)
```

Dimensiones de $A \otimes B$: En este otro caso, A es 2×2 y B es 1×3, por lo que las dimensiones del resultado son $(2 \times 1) \times (2 \times 3)$, es decir, 2×6.

8. Ejercicio 7: Autovalores y Autovectores

Los autovalores y autovectores son propiedades fundamentales de las matrices cuadradas que revelan información profunda sobre su comportamiento como transformaciones lineales. Un autovector de una matriz es un vector que, al ser multiplicado por la matriz, no cambia su dirección, solo es escalado por un factor. Ese factor de escala es el autovalor correspondiente.

8.1. Calcular solo los autovalores con `linalg.eigvals()`

Si únicamente necesitamos los autovalores de una matriz, la función `linalg.eigvals()` es más eficiente que calcular también los autovectores.

Listing 8: Cálculo de autovalores

```
import numpy as np
from scipy import linalg

# 1) Definimos la matriz A
A = np.array([[0, -1],
```

```
               [1,  0]])

# Calculamos solo sus autovalores
eigenvalues = linalg.eigvals(A)

print("Matriz A:\n", A)
print("\nAutovalores de A:\n", eigenvalues)

# --- RESULTADO ESPERADO ---
# Matriz A:
#  [[ 0 -1]
#   [ 1  0]]
#
# Autovalores de A:
#  [0.+1.j 0.-1.j]
```

Explicación: La matriz A representa una rotación de 90 grados. Es interesante notar que no tiene autovalores reales, ya que ningún vector (excepto el nulo) mantiene su dirección después de rotar 90 grados. Sus autovalores son los números complejos i y $-i$.

8.2. Calcular autovalores y autovectores a derecha con `linalg.eig()`

La función `linalg.eig()` devuelve una tupla con los autovalores y los autovectores correspondientes. Por defecto, calcula los autovectores a derecha.

```
# 2) Calculamos autovalores y autovectores a derecha
lambdas_r, right_vectors = linalg.eig(A)

print("Autovalores (lambda):\n", lambdas_r)
print("\nAutovectores a derecha (v):\n", right_vectors)

# --- RESULTADO ESPERADO ---
# Autovalores (lambda):
#  [0.+1.j 0.-1.j]
#
# Autovectores a derecha (v):
#  [[0.70710678+0.j         0.70710678-0.j        ]
#   [0.         -0.70710678j 0.         +0.70710678j]]
```

Explicación: El resultado `right_vectors` es una matriz donde cada **columna** es un autovector. La columna i corresponde al autovalor en la posición i del array `lambdas_r`.

8.3. Verificar la ecuación a derecha ($Av = \lambda v$)

Ahora comprobamos que la definición se cumple para el primer par autovalor-autovector.

```
# 3) Verificamos para el primer par (columna 0)
lambda_1 = lambdas_r[0]
v_1 = right_vectors[:, 0] # Primera columna

# Calculamos el lado izquierdo de la ecuacion: A @ v
lado_izquierdo = A @ v_1

# Calculamos el lado derecho: lambda * v
```

```
lado_derecho = lambda_1 * v_1

print("Lado izquierdo (A @ v1):\n", lado_izquierdo)
print("\nLado derecho (lambda1 * v1):\n", lado_derecho)
print("\nSon aproximadamente iguales?:", np.allclose(lado_izquierdo,
    lado_derecho))

# --- RESULTADO ESPERADO ---
# Lado izquierdo (A @ v1):
#  [0.         +0.70710678j 0.70710678+0.j        ]
#
# Lado derecho (lambda1 * v1):
#  [0.         +0.70710678j 0.70710678+0.j        ]
#
# Son aproximadamente iguales?: True
```

8.4. Calcular autovalores y autovectores a izquierda

Para calcular los autovectores a izquierda, pasamos el argumento `left=True` a la función `linalg.eig()`.

```
# 4) Calculamos autovalores y autovectores a izquierda
lambdas_l, left_vectors = linalg.eig(A, left=True, right=False)

print("Autovalores (lambda):\n", lambdas_l)
print("\nAutovectores a izquierda (u):\n", left_vectors)

# --- RESULTADO ESPERADO ---
# Autovalores (lambda):
#  [0.+1.j 0.-1.j]
#
# Autovectores a izquierda (u):
#  [[0.         -0.70710678j 0.70710678+0.j        ]
#   [0.         +0.70710678j 0.70710678-0.j        ]]
```

Explicación: En este caso, la matriz de autovectores a izquierda contiene cada autovector como una **fila**. La fila i corresponde al autovalor en la posición i.

8.5. Verificar la ecuación a izquierda ($uA = \mu u$)

Finalmente, comprobamos que se cumple la definición para el primer par de autovalor y autovector a izquierda.

```
# 5) Verificamos para el primer par (fila 0)
mu_1 = lambdas_l[0]
u_1 = left_vectors[0, :] # Primera fila

# Calculamos el lado izquierdo de la ecuacion: u @ A
lado_izquierdo_u = u_1 @ A

# Calculamos el lado derecho: mu * u
lado_derecho_u = mu_1 * u_1
```

```
print("Lado izquierdo (u1 @ A):\n", lado_izquierdo_u)
print("\nLado derecho (mu1 * u1):\n", lado_derecho_u)
print("\nSon aproximadamente iguales?:", np.allclose(lado_izquierdo_u,
    lado_derecho_u))

# --- RESULTADO ESPERADO ---
# Lado izquierdo (u1 @ A):
#  [0.70710678+0.j         0.         +0.70710678j]
#
# Lado derecho (mu1 * u1):
#  [0.70710678+0.j         0.         +0.70710678j]
#
# Son aproximadamente iguales?: True
```

9. Ejercicio 8: Singular Value Decomposition (SVD)

La Descomposición en Valores Singulares (SVD, por sus siglas en inglés) es una de las factorizaciones de matrices más importantes del álgebra lineal. Permite descomponer cualquier matriz A en el producto de tres matrices con propiedades muy especiales: una matriz unitaria L, una matriz diagonal D con valores no negativos, y la transpuesta conjugada de otra matriz unitaria, $R^{\dagger}$. La SVD tiene innumerables aplicaciones, desde la compresión de imágenes hasta el procesamiento de lenguaje natural.

9.1. Crear una matriz compleja aleatoria

Para este ejercicio, crearemos una matriz A de 9×6 con entradas complejas. La parte real y la parte imaginaria de cada elemento se generarán a partir de una distribución normal estándar.

Listing 9: Creación de una matriz compleja

```
import numpy as np
from scipy import linalg

# 1) Definimos las dimensiones n y m
n, m = 9, 6

# Creamos la parte real y la parte imaginaria con numeros aleatorios
# de una distribución normal estándar (media 0 y varianza 1).

real_part = np.random.randn(n, m)
imag_part = np.random.randn(n, m)

# Combinamos para crear la matriz compleja A. '1j' es la unidad
   imaginaria.
A = real_part + 1j * imag_part

print(f"La forma de la matriz A es: {A.shape}")
print(f"El tipo de dato de A es: {A.dtype}")

# --- RESULTADO ESPERADO ---
# La forma de la matriz A es: (9, 6)
```

```
# El tipo de dato de A es: complex128
```

9.2. Calcular la SVD e inspeccionar las dimensiones

Usamos `linalg.svd()` para descomponer A. La función devuelve la matriz unitaria de llegada L (llamada "U" en la salida de SciPy), el vector de valores singulares s, y la matriz unitaria de salida conjugada y transpuesta $R^\dagger$ (llamada "Vh" en SciPy).

```
# 2) Calculamos la SVD de A
# Usamos la nomenclatura de SciPy: U, s, Vh (corresponden a L, s,
   R_conjugado)
L, s, Rc = linalg.svd(A)

print(f"Forma de L (matriz unitaria de llegada): {L.shape}")
print(f"Forma de s (vector de valores singulares): {s.shape}")
print(f"Forma de Rc (matriz R conjugada transpuesta): {Rc.shape}")

# --- RESULTADO ESPERADO ---
# Forma de L (matriz unitaria de llegada): (9, 9)
# Forma de s (vector de valores singulares): (6,)
# Forma de Rc (matriz R conjugada transpuesta): (6, 6)
```

Explicación: Como se esperaba, L es una matriz cuadrada de $n \times n$, $R^\dagger$ ('Rc') es una matriz cuadrada de $m \times m$, y s es un vector con $k = \min(n, m) = 6$ valores singulares.

9.3. Construir la matriz diagonal D

La función `svd` devuelve los valores singulares como un vector 1D. Para reconstruir la matriz original, debemos colocar estos valores en la diagonal de una matriz D de las mismas dimensiones que A ($n \times m$).

```
# 3) Creamos la matriz diagonal D de forma n x m
D = np.zeros((n, m), dtype=complex)

# Poblamos la diagonal principal de D con los valores singulares de s
# np.diag(s) crea una matriz cuadrada 6x6, la insertamos en D
D[:m, :m] = np.diag(s)

print(f"La forma de la matriz diagonal D es: {D.shape}")
# Para visualizarlo, imprimimos D (mostrará los sigmas en la diagonal)
print("\nMatriz D (primeras 6 filas):\n", D[:6])

# --- RESULTADO ESPERADO ---
# La forma de la matriz diagonal D es: (9, 6)
```

9.4. Verificar la Ecuación de Reconstrucción

Finalmente, comprobamos que la descomposición es correcta verificando si $LDR^\dagger$ reconstruye la matriz original A.

```
# 4) Reconstruimos A usando las matrices de la SVD
# A_recons = L @ D @ Rc
```

```
A_recons = np.dot(L, np.dot(D, Rc))

# Verificamos que la matriz reconstruida es numéricamente cercana a la
    original
son_iguales = np.allclose(A, A_recons)

print(f"Se cumple que A es aproximadamente igual a L*D*R_conjugada?: {
    son_iguales}")

# --- RESULTADO ESPERADO ---
# Se cumple que A es aproximadamente igual a L*D*R_conjugada?: True
```