Soluciones a los ejercicios de Condicionales y bucles en Python (con NumPy)

Ejercicio: Método de Jacobi

Queremos adaptar el código dado para el método de Jacobi a una función de Python cuya cabecera sea

```
import numpy as np

def jacobi(N, A, b, x, d):
```

donde:

- `N` es el número máximo de iteraciones.
- `A` es la matriz del sistema (array 2D de NumPy).
- `b` es el vector de términos independientes (array 1D de NumPy).
- `x` es el valor inicial (array 1D de NumPy).
- `d` es el parámetro de control de la tolerancia.

La idea es:

1. Repetir el proceso de Jacobi como máximo `N` veces.
2. En cada iteración, calcular una nueva aproximación `x_nueva` usando la fórmula

$$x_n(i) = \frac{1}{a_{ii}} \left(-\sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_{n-1}(j) + b_i \right).$$

3. Calcular la norma de la diferencia entre `x_nueva` y `x`:

$$\|x_n - x_{n-1}\| = \|x\_\text{nueva} - x\| = \texttt{np.linalg.norm(x\_nueva - x)}.$$

4. Si esta norma es menor que `d`, consideramos que hemos alcanzado una aproximación suficiente y usamos `break` para salir del bucle.

5. Actualizar `x` con la nueva aproximación y seguir iterando mientras no se haya alcanzado la tolerancia y no se superen las `N` iteraciones.

Una posible implementación completa es:

```
import numpy as np

def jacobi(N, A, b, x, d):
    """
    Método de Jacobi para resolver A x = b.

    Parámetros
    ----------
    N : int
        Número máximo de iteraciones.
    A : np.ndarray
        Matriz del sistema (n x n).
    b : np.ndarray
        Vector de términos independientes (n,).
    x : np.ndarray
        Aproximación inicial (n,).
    d : float
        Tolerancia para la norma de x_n - x_{n-1}.

    Devuelve
    --------
    x : np.ndarray
        Aproximación final a la solución.
    k : int
        Número de iteraciones realizadas.
    """

    n = x.shape[0]

    for k in range(N):
```

```
        # Guardamos la aproximación anterior
        x_anterior = x.copy()

        # Nueva aproximación x_n
        x_nueva = np.zeros_like(x)

        for i in range(n):
            cuant = 0.0
            for j in range(n):
                cuant = cuant + A[i, j] * x_anterior[j]

            x_nueva[i] = (-cuant + A[i, i] * x_anterior[i] + b[i]) / A[i, i]

        # Comprobamos la norma de la diferencia
        diferencia = np.linalg.norm(x_nueva - x_anterior)

        # Actualizamos x
        x = x_nueva

        # Criterio de parada
        if diferencia < d:
            break

    return x, k + 1
```

Obsérvese que:

- Se usa `np.zeros_like(x)` para crear un vector de ceros con el mismo tamaño que `x`.
- La expresión `np.linalg.norm(x_nueva - x_anterior)` calcula la norma euclídea de la diferencia.
- La identación (sangrado) de los bloques `for`, `if`, etc., es esencial para que el programa funcione correctamente.