Ayudantía
SCT - Semana - 05

Repaso para Certamen 1

1. Se tiene la siguiente función igual a una sumatoria infinita:

$$S(x) = \sum_{k=1}^{\infty} \frac{\cos(kx)}{k}$$

Esta función no se puede evaluar en un múltiplo de 2π, pues cada uno de los $\cos(kx)$ internos se evaluaría a 1, obteniendo la famosa serie armónica que diverge a infinito: $S(2m\pi) = \sum_{k=1}^{\infty} \frac{1}{k} = +\infty$.

Nos interesa seguir analizando esta función, pero no podemos sumar infinitos términos en un computador. En su lugar, podemos definir una variante $S_n(x)$ que considera solo los primeros n términos:

$$S_n(x) = \sum_{k=1}^{n} \frac{\cos(kx)}{k}$$

a) Construya un algoritmo que, dado un entero j, realice j iteraciones del **método de Newton** para aproximar un **punto crítico** de la función $S_n(x)$, es decir, un valor r tal que $S_n'(r) = 0$.

> Un error común en esta pregunta es construir la siguiente iteración de Newton:
>
> $$x_{i+1} = x_i - \frac{S_n(x)}{S_n'(x)}$$
>
> pues esta iteración busca aproximar una raíz de $S_n(x)$. Sin embargo, la pregunta no busca una raíz de $S_n(x)$, sino un **punto crítico**, que equivale a buscar una **raíz de su derivada** $S_n'(x)$.
> La iteración correcta sería la siguiente:
>
> $$x_{i+1} = x_i - \frac{S_n'(x)}{S_n''(x)}$$
>
> donde las derivadas $S_n'(x)$ y $S_n''(x)$ están dadas por:
>
> $$S_n'(x) = -\sum_{k=1}^{n} \sin(kx)$$
> $$S_n''(x) = -\sum_{k=1}^{n} k\cos(kx)$$
>
> Entonces, el algoritmo a construir debe ser:
>
> - **Elegir un *initial guess* x_0.** No es tan sencillo llegar y elegir $x_0 = 0$ (o, en general, $x_0 = 2m\pi$), pues a medida que la cantidad n de términos aumenta, el valor de $S_n(0)$ (o $S_n(2m\pi)$) diverge. **Una mejor opción es $x_0 = 1$.**
> - **Realizar la siguiente iteración:**
>
> $$x_{i+1} = x_i - \frac{\sum_{k=1}^{n} \sin(kx)}{\sum_{k=1}^{n} k\cos(kx)}$$
>
> j veces en una máquina de precisión doble.

b) Implemente el algoritmo anterior en Python, mediante una función que reciba dos parámetros: un entero n indicando la cantidad de términos en la sumatoria $S_n(x)$, y un entero j indicando la cantidad de iteraciones a realizar del método de Newton. La función que implemente no solo debe retornar el punto crítico r, sino también una estimación de la **tasa lineal de convergencia** S y la **tasa cuadrática de convergencia** M calculadas sobre las j iteraciones del método de Newton.

Use la librería NumPy y sus capacidades de vectorización donde sea adecuado. En particular, puede usar:

- `np.arange(start, stop)` para generar un arreglo `[start, start+1, start+2, ..., stop-2, stop-1]`;

- `np.sum` para calcular la suma de todos los valores de un arreglo: `np.sum([1, 2, 4]) = 7`; y
- `np.sin` y `np.cos` para calcular el seno/coseno de un valor o de un arreglo de valores. En este último caso, un ejemplo es `np.sin([0, 1, 2]) = [0.0, 0.84147098, 0.90929743]`.

La función debe llevar la siguiente firma:

```python
'''
input:
n : (int64) Upper limit of the sum S_n(x).
j : (int64) Number of iterations to be used in Newton's method.
output:
r : (double) Root obtained by Newton's method.
S : (double) Estimated linear rate of convergence.
M : (double) Estimated quadratic rate of convergence.
'''
def find_critical_point_of_Sn(n, j):
    # Your code goes here
    return r, S, M
```

```python
def find_critical_point_of_Sn(n, j):
    x0 = 1

    # np.arange(n) = [0, 1, 2, ..., n-1]
    # np.arange(start, stop) = [start, start+1, start+2, ..., stop-1]
    k = np.arange(1, n+1)  # [1, 2, 3, ..., n]

    # k*x = [1, 2, 3, ..., n] * x
    #     = [x, 2x, 3x, ..., nx]
    #
    # np.sin(k*x) = [sin(x), sin(2x), sin(3x), ..., sin(nx)]
    #
    # np.sum(np.sin(k*x)) = sin(x) + sin(2x) + sin(3x) + ... + sin(nx)
    f = lambda x: -np.sum(np.sin(k*x))

    # np.cos(k*x) = [cos(x), cos(2x), cos(3x), ..., cos(nx)]
    #
    # k * np.cos(k*x) = [1, 2, 3, ..., n] * [cos(x), cos(2x), cos(3x), ..., cos(nx)]
    #                 = [cos(x), 2cos(2x), 3cos(3x), ..., ncos(nx)]
    #
    # np.sum(k * np.cos(k*x)) = cos(x) + 2cos(2x) + 3cos(3x) + ... + ncos(nx)
    fp = lambda x: -np.sum(k * np.cos(k*x))

    # Newton
    xi = x0
    xim1 = 0  # placeholder para xi anterior
    xim2 = 0  # placeholder para xi anterior al anterior
    for _ in range(j):
        xim2 = xim1
        xim1 = xi
        xi = xi - f(xi) / fp(xi)

    r = xi

    ei = abs(xi - xim1)  # error actual
    eim1 = abs(xim1 - xim2)  # error anterior
    S = ei / eim1
    M = ei / (eim1 * eim1)

    return r, S, M
```

2. Revisa y resuelve, en la guía de ejercicios para C1, el problema 2.6: "Un triángulo y dos preguntas".

El solucionario de esta pregunta corresponde a la sección 4.4 de la guía mencionada.