Física Computacional II (510240)

Universidad de Concepción
Facultad de Ciencias Físicas y Matemáticas
Departamento de Física

Pauta corrección – Tarea 3

Roberto Navarro

Pregunta

(a) La función de Bessel de orden cero suele aparecer en problemas con simetría cilíndrica. Esta función se puede definir como:

$$J_0(x) = \frac{1}{\pi}\int_0^\pi d\theta\, \cos(x\sin\theta)\,. \tag{1}$$

Use el método de Romberg para integrar numéricamente la ecuación (1). Luego, grafique $J_0(x)$ como función de $0 < x < 30$.

Note que el método de Romberg escrito en clases no está vectorizado (no acepta listas de valores de x). Por ello, deberá evaluar el método de Romberg individualmente para cada valor de x (usando un for-loop). Alternativamente, averígüe sobre la función `numpy.vectorize` (opcional, +0.2 puntos).

Solución:

Los códigos usados para responder a esta tarea se encuentran en `src/bessel.py`.

Para evaluar la integral (1), definimos la función `besselJ0`:

0.5 puntos

```
1 def besselJ0(x):
2     f = lambda ang: np.cos(x*np.sin(ang))
3
4     return Romberg(f, 0, np.pi)/ np.pi
```

Note que **podemos definir funciones dentro de funciones** en python. La variable `f` en la línea 2 corresponde a el integrando de (1). En este caso, usamos el concepto de *funciones lambda* o *funciones anónimas*. Es una forma simple de definir funciones, el cual se puede lograr de forma equivalente con la clave `def`:

```
1 def besselJ0(x):
2     def f(ang): return np.cos(x*np.sin(ang))
3
4     return Romberg(f, 0, np.pi)/ np.pi
```

Así, `f` es una función de `ang` que toma `x` como una variable global. Luego, la función `besselJ0` retorna la integral en (1) usando el método de Romberg, el cual es un algoritmo que usa el

método trapezoidal repetidamente subdividiendo el intervalo $0 < \theta < \pi$ hasta que converge, y que está definido en el módulo `src/Quadratures.py` [1].

Ahora bien, la función `Romberg` no está vectorizada, por lo que no es posible evaluar `besselJ0(x)` con una lista o array de valores `x`. Por ello, definiremos una función auxiliar `besselJ0vec` que evalúa `besselJ0` para cada elemento de una lista `x`:

```
1 def besselJ0vec(x):
2     return np.array([besselJ0(xi) for xi in x])
```

Alternativamente, se puede usar la clase `numpy.vectorize`, que hace algo similar a lo que describimos anteriormente:

+0.2 puntos

```
1 besselJ0vec = np.vectorize(besselJ0)
```

Los resultados de evaluar `besselJ0` para distintos valores de x se pueden observar en la línea azul de la figura 1(izquierda).

(b) Funciones de Bessel de orden $n > 0$ se pueden encontrar mediante la relación de recurrencia:

$$J_{n+1}(x) = -x^n \frac{d}{dx}\left[x^{-n} J_n(x)\right] . \tag{2}$$

Como ya conoce $J_0(x)$ del ítem anterior, evalúe (2) numéricamente usando derivadas centradas para encontrar $J_1(x)$, $J_2(x)$ y $J_3(x)$. Luego, grafique estas funciones en una misma figura en el rango $0 < x < 30$ (puede graficarlas en la misma figura pedida en el ítem anterior).

Solución:

Usaremos diferencias centradas para evaluar la derivada en (2).

Supongamos que, para una función $f(x)$ de una variable x, solo conocemos $f_i = f(x_i)$ en una serie de puntos $x = x_i$ ordenados. Luego, podemos evaluar diferencias finitas de la forma

$$f'(\xi_i) = \frac{f(x_{i+1}) - f(x_i)}{2(x_{i+1} - x_i)} , \tag{3}$$

donde $x_i < \xi_i < x_{i+1}$ es algún número. Para efectos prácticos, supondremos que $\xi_i = (x_{i+1} + x_i)/2$ es el punto medio entre x_i y x_{i+1}, lo que corresponde a una derivada centrada con un error de cálculo respecto a la derivada analítica del orden $O(|x_{i+1} - x_i|^2)$.

Luego, la ecuación (3) es implementada mediante la función `deriv`, el cual retornará ξ_i y la derivada centrada:

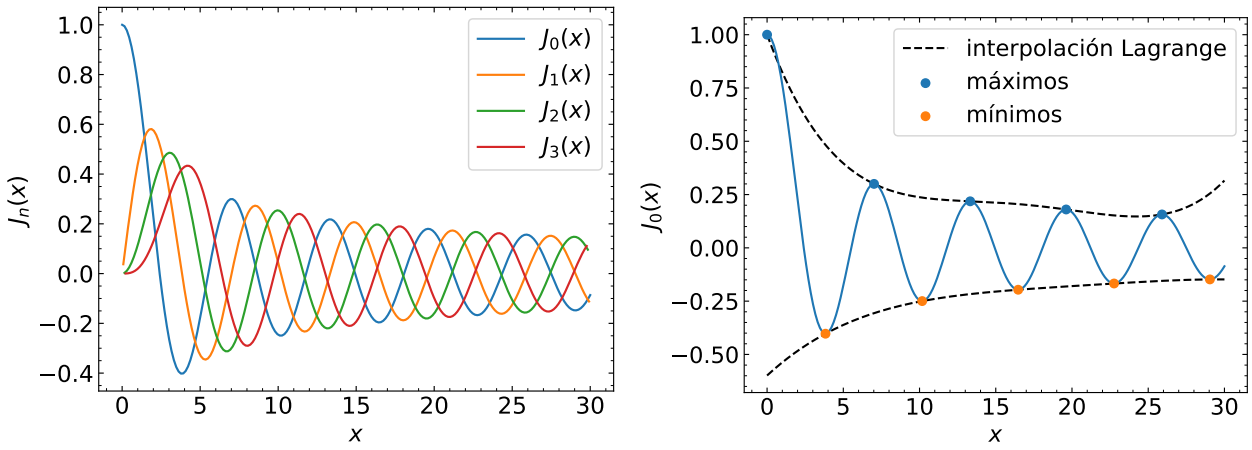
0.5 puntos

```
1 def deriv(x, y):
2     return 0.5*(x[1:]+x[:-1]), (y[1:] - y[:-1]) / (x[1:] - x[:-1])
```

Luego, usaremos `deriv` para evaluar la ecuación (2) para distintos valores de n:

0.75 puntos

```
1 # evalúa J0
2 x = np.linspace(0, 30, 200)
3 j = [besselJ0vec(x)]
4
```



1 punto Figura 1: (izquierda) Gráficos de $J_n(x)$ calculados según las ecuaciones (1) y (2), para $n = \{0, 1, 2, 3\}$.
1 punto (derecha) Interpolación polinomial de los máximos y mínimos de $J_0(x)$.

```
# calcula Jn
xhalf = x.copy()
for n in range(3):
    xhalf, jn = deriv(xhalf, j[n]/xhalf**n)
    j += [-xhalf**n * jn]

    plt.plot(xhalf, j[n+1], label=f"$J_{n+1}(x)$")
```

Note que acá no requerimos definir funciones para J_n, sino que solo estamos evaluando el valor de $J_n(x_i)$ en ciertos valores discretos de x_i cuando n es par, y $J_n(\xi_i)$ cuando n es impar. Es decir, primero evaluamos $J_0(x_i)$ mediante (1) y luego usamos esa información para calcular J_n recursivamente usando (2).

Los resultados de evaluar J_n de esta forma para distintos valores de x se pueden observar en la figura 1(izquierda). Note que $J_0(0) = 1$, mientras que $J_n(0) = 0$ para $n > 0$.

(c) Los máximos y mínimos de $J_0(x)$ corresponden a los ceros de $J_0'(x) = 0$. Use el método de Newton-Raphson para encontrar todos los máximos y mínimos de $J_0(x)$ en el rango $0 < x < 30$.

Solución:

Notemos que $J_1(x) = J_0'(x)$ según la ecuación (2). Por lo tanto, los máximos y mínimos de $J_0(x)$ se pueden encontrar evaluando los puntos donde $J_1(x) = 0$. Como ya hemos calculado $J_1(\xi_i)$ en puntos discretos de $x = \xi_i$, entonces localizaremos aproximadamente los ceros de $J_1(x)$ buscando simplemente para qué valores de i hay un cambio de signo entre $J_1(\xi_i)$ y $J_1(\xi_i)$:

```
j1 = j[1]          # extrae los valores de J1(ξi)
zeros = x[1:-1]
zeros = zeros[j1[1:] * j1[:-1] <0]  # localiza cambios de signo
```

Luego, refinamos los resultados usando el método de Newton-Raphson. En este caso, evaluamos $J_1(x) = J_0'(x)$ y su derivada $J_1'(x)$ usando derivadas centradas (puesto que x no necesariamente son los mismos valores calculados en ítems anteriores). Además, en el proceso evaluamos si los puntos encontrados son máximos (cuando $J_1'(x) < 0$) o mínimos de J_0 (cuando $J_1'(x) > 0$):

0.75 puntos

```
1  def NewtonJ1(seed, tol=1e-5, dx=1e-3):
2      stop = False
3      x = seed
4      while not stop:
5          # evalúa J0 en tres puntos x-dx, x y x+dx
6          Jminus, J0, Jplus = besselJ0vec([x-dx, x, x+dx])
7
8          f = (Jplus - Jminus)/(2*dx)
9          df = (Jplus -2*J0 + Jminus)/(dx**2)
10
11         # método de Newton-Raphson
12         error = f/df
13         x = x - error
14
15         # define criterio de convergencia
16         stop = abs(error)<tol and abs(f)<tol
17
18     # retorna x=cero de J1, y si x representa maximo de J0 (True/False)
19     return x, df<0
20
21 # agregamos x=0 manualmente
22 zeros, ismax = np.array([[0, True]] + [NewtonJ1(s) for s in zeros]).T
23
24 # `ismax` es retornado como lista de enteros. Convertimos a booleano
25 ismax = ismax.astype(np.bool)
26
27 # de los ceros de J1, extrae los que son máximos o mínimos de J0
28 maximos = zeros[ismax]
29 minimos = zeros[~ismax]  # Si imax=False, entonces ~imax=True
30
31 plt.scatter(maximos, besselJ0vec(maximos))
32 plt.scatter(minimos, besselJ0vec(minimos))
```

0.75 puntos

Notemos que en la línea 22 agregamos manualmente un máximo obvio de $J_0(x)$. En efecto, podemos evaluar (1) analíticamente para $x = 0$, de donde obtenemos que $J_0(0) = 1$. Además, su segunda derivada analítica en ese punto se puede calcular como:

$$J_0''(0) = -\frac{1}{\pi}\int_0^\pi d\theta\, \sin^2\theta = -\frac{1}{2}\,, \tag{4}$$

o sea $J''(0) < 1$, por lo que $x = 0$ es un máximo de $J_0(x)$

Los máximos de J_0 se grafican como puntos azules en la figura (1)(derecha), mientras que sus mínimos se grafican con puntos naranjos.

(d) **Considere solo los máximos de $J_0(x)$ calculados en el ítem anterior. Interpole un polinomio usando estos puntos y luego grafique este polinomio en el rango $0 < x < 30$. Haga lo mismo para los mínimos de $J_0(x)$. Estas curvas corresponden a la *envolvente* de la función $J_0(x)$, es decir, son curvas más suaves que acotan a esta función.**

Recuerde que para discriminar si un punto corresponde a un máximo o un mínimo, puede usar el criterio de la segunda derivada, $J_0''(x)$. Para ello, deberá estimar esta segunda derivada usando derivadas centradas.

Solución:

Usaremos interpolación por polinomios de Lagrange, para lo cual usaremos la función `Neville` implementado en el módulo `src/Neville.py` [1]:

0.75 puntos

```
polymax = Neville(maximos, besselJ0vec(maximos))
polymin = Neville(minimos, besselJ0vec(minimos))

plt.plot(x, polymax(x), "--k")
plt.plot(x, polymin(x), "--k")
```

Los polinomios interpolados se visualizan en la figura (1)(derecha) con líneas segmentadas. Ambos polinomios son de grado 4 pues encontramos 5 mínimos y 5 máximos en el intervalo $0 < x < 30$. Note que para el caso del polinomio que interpola los máximos [la curva segmentada superior en la figura (1)(derecha)], esta parece ser creciente para $x > 26$. En este intervalo estamos *extrapolando*, puesto que no existen puntos que interpolar para $x > 26$, lo cual resulta en este comportamiento no intuitivo (lo que significa que hay que tener mucho cuidado cuando queremos sacar conclusiones al extrapolar con polinomios de Lagrange).

Referencias

[1] Física Computacional II (510240), https://github.com/fiscomp2-510240-UdeC2021/fiscomp2, [Online; versión 12-enero-2021], 2021.