Física Computacional II (510240)

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

Pauta corrección – Tarea 2

Roberto Navarro

Pregunta 1

(a) **La ecuación que describe los modos (ondas electrostáticas o electromagnéticas) que se pueden propagar en un plasma fuera del equilibrio termodinámico, se puede escribir en términos de la *función de dispersión de plasmas modificada* [Summers1991], definida como (modificada para que no contenga polos):**

$$Z_\kappa(\xi) = \sum_{\ell=0}^{\kappa} \frac{(\kappa+\ell)!}{\ell!\, 2^\ell} \left(1 + \frac{\xi}{i\sqrt{\kappa}}\right)^\ell , \tag{1}$$

donde $\kappa > 0$ es un número natural y $\xi = x + iy$ es un variable compleja con $x = \mathbf{Re}(\xi)$ e $y = \mathbf{Im}(\xi)$.

Solución:

La ecuación (1) es un polinomio de grado κ de la forma:

$$p_\kappa(w) = a_0 + a_1 w + \cdots + a_\kappa w^\kappa , \tag{2}$$

donde $w = 1 - i\xi/\sqrt{\kappa}$ y los coeficientes a_ℓ se pueden calcular por recurrencia:

$$a_0 = \kappa! , \qquad a_\ell = \frac{\kappa+\ell}{2\ell} a_{\ell-1} . \tag{3}$$

En el script `src/p1-fractal/plasma_dispersion_function.py` se implementa esta función usando la clase de polinomios `Poly`. En particular, la porción de código relevante para calcular los coeficientes a_ℓ para cierto valor de κ se incluye a continuación.

0.2 puntos

```
1  class PolyZ(Poly):
2      def __init__(self, k):
3          super().__init__([0])              # hereda atributos de clase Poly
4
5          # calcula coeficientes de funcion Zk
6          An = [np.math.factorial(k)] + [0.5*(k/n+1.0) for n in range(1,k+1)]
7          self.coef = np.cumprod(An, dtype=np.complex128)
```

La variable `An` es una lista cuyo primer elemento es $a_0 = \kappa!$ y el resto de sus elementos corresponden a los factores $(k+\ell)/(2\ell)$, con $\ell = \{1, 2, \ldots, \kappa\}$, necesarios para usar la relación

de recurrencia (3). Finalmente, `self.coef` representa los coeficientes a_ℓ que son calculados usando la función `numpy.cumprod`.

De esta forma, los coeficientes a_ℓ son calculados una sola vez para ser usados las veces necesarias para cierto valor de κ.

Luego, para evaluar $p(w)$ con $w = 1 - i\xi/\sqrt{\kappa}$, implementamos la función miembro `__call__`:

```
class PolyZ(Poly):
    def __call__(self, xi):
        w = 1 - (1j/np.sqrt(self.k)) * np.asarray(xi)

        return super().__call__(w)
```

En este caso, `super().__call__(w)` simplemente evalúa $p(w) = a_0 + w(a_1 + w(a_2 + \cdots))$.

Finalmente, para calcular las raíces de $Z_\kappa(\xi)$ usaremos el método de Newton-Raphson, por lo que necesitamos calcular la derivada de $Z_\kappa(\xi)$. Este se implementa con la función miembro `deriv`:

0.2 puntos

```
class PolyZ(Poly):
    def deriv(self, m=1):
        dp = PolyZ(0)           # crea objeto con propiedades de clase PolyZ

        # calcula coeficientes derivada de polinomio y usa regla de la cadena
        dp.coef = super().deriv(m).coef * (-1j/np.sqrt(self.k))**m

        dp.k = self.k    # preserva valor de k al derivar

        return dp
```

Note que usamos la función `super().deriv(m)`, el cual simplemente retorna los coeficientes de la m-ésima derivada de un polinomio $p(w)$ con respecto a w. Luego, por regla de la cadena, multiplicamos por $(dw/d\xi)^m = (-i/\sqrt{\kappa})^m$. Con esto, procedemos a seguir los pasos indicados en esta tarea.

En este problema estudiaremos cómo converge el método de Newton-Raphson a las soluciones de $Z(\xi) = 0$. Para ello,

- **Elija N puntos $\xi_n = x_n + iy_n$, con $n = \{1, 2, \ldots, N\}$ y N lo suficientemente grande, en el rango $-5 < x_n < 5$ y $0 < y_n < 8$.**

  Solución:

  En el script `src/p1-fractal/p1a_zeros.py` se usa la instancia `numpy.mgrid`:

0.2 puntos

```
x, y = np.mgrid[-3:3:80j, -5:0:80j]
xi = x + 1j*y
```

  el cual crea una grilla de 80×80 nodos en el plano $-3 < x < 3$ y $-5 < y < 0$. Finalmente, construye 80×80 números complejos $xi = x + iy$ distintos.

- **Use cada uno de estos puntos ξ_n como semilla para el método de Newton-Raphson y encontrar soluciones de $Z_\kappa(\xi) = 0$.**

Solución:

Como `xi` es un arreglo de 80×80 números, entonces debemos usar un doble ciclo-for. Cada uno de estos valores es usado como semilla para el método de Newton-Raphson, los cuales entregamos al polinomio $p(xi)$ (que representa a la función $Z_\kappa(\xi)$ definido mediante `p=PolyZ(kappa)`) y su derivada `p.deriv()`. Luego, la solución a la que converge el método de Newton-Raphson para cada valor de `xi` es guardado en un arreglo `sol` también de 80×80 elementos.

0.2 puntos

```
p = PolyZ(kappa)
sol = np.zeros_like(xi)
for i in range(xi.shape[0]):
    for j in range(xi.shape[1]):
        sol[i,j] = NewtonRaphson(p, p.deriv(), xi[i,j])
```

- **En el plano x-y, grafique los puntos al que converge el método de Newton-Raphson.**
- **Para comparar, incluya en el mismo gráfico anterior los contornos de $\mathrm{Re}[Z(x+iy)] = 0$ e $\mathrm{Im}[Z(x+iy)] = 0$. Explique e interprete lo que se observa en esta figura en relación con las soluciones de $Z(\xi) = 0$.**
- **Repita el procedimiento para 4 valores distintos de κ.**

Solución:

La variable `fun=p(xi)` representa la función $p(xi)$ evaluada para cada valor de xi. Por lo tanto, `fun` es un arreglo de 80×80 valores complejos. Los puntos donde $\mathrm{Re}[p(xi)] = 0$ o $\mathrm{Im}[p(xi)] = 0$ son curvas de nivel o contornos en el plano x-y, los cuales podemos encontrar rápidamente con la función `matplotlib.pyplot.contour`:

0.2 puntos

```
plt.contour(x, y, fun.real, levels=[0], colors="k")
plt.contour(x, y, fun.imag, levels=[0], colors="r")
```

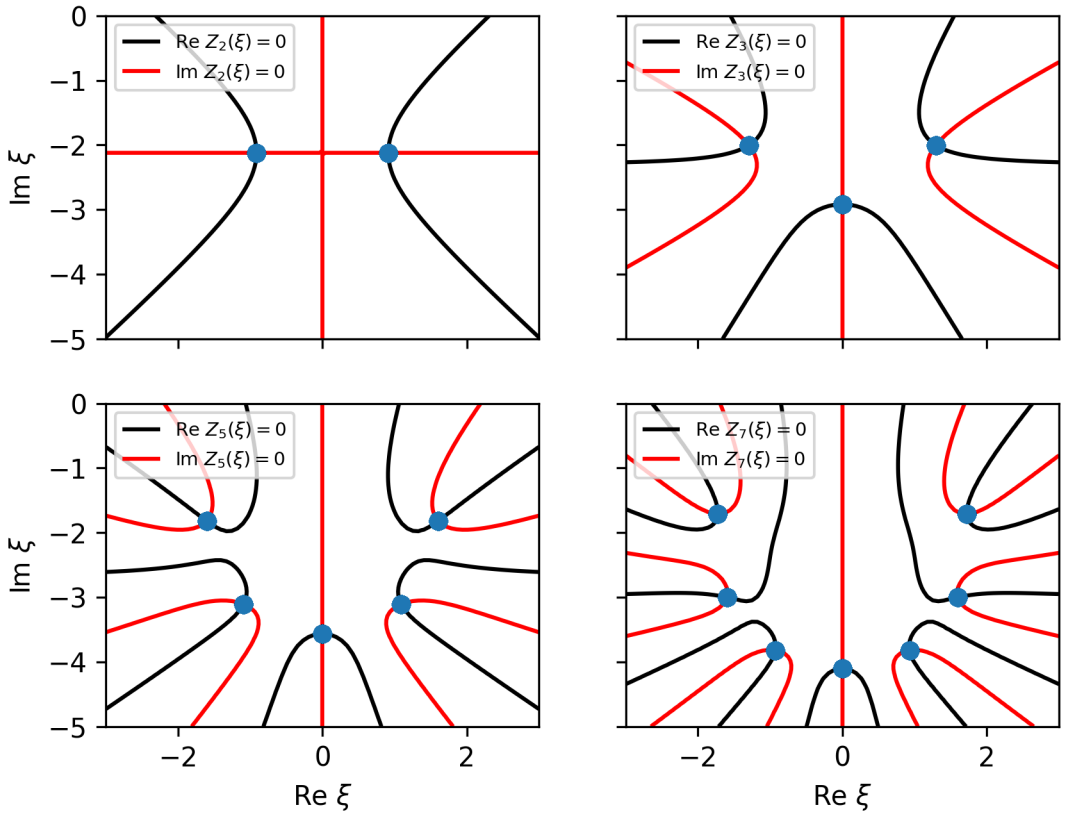
Por otro lado, las soluciones `sol` a las que converge el método de Newton-Raphson son graficadas mediantes puntos:

0.2 puntos

```
plt.scatter(sol.real, sol.imag)
```

0.2 puntos

Los resultados para $\kappa = \{2, 3, 5, 7\}$ se observan en la figura 1. Notemos que los ceros de $Z_\kappa(\xi)$, que son los puntos azules en la figura 1, coinciden con los cruces de las curvas de nivel de $\mathrm{Re}[Z_\kappa(\xi)] = 0$ (lineas negras) e $\mathrm{Im}[Z_\kappa(\xi)]$ (lineas rojas). Esto tiene sentido pues una función compleja es cero si sus partes real e imaginaria son ambas cero.



0.2 puntos Figura 1: Contornos de la parte real (negro) e imaginaria (rojo) de $Z_\kappa(\xi) = 0$ para $\kappa = 2$ (arriba-izquierda), $\kappa = 3$ (arriba-derecha), $\kappa = 5$ (abajo-izquierda) y $\kappa = 7$ (abajo-derecha). Los puntos azules representan los ceros de la función $Z_\kappa(\xi)$.

(b) **Dependiendo de cada semilla ξ_n, el método de Newton-Raphson convergerá a una de las soluciones $\xi = \xi_{\text{sol}}$ de $Z_\kappa(\xi) = 0$. Asigne un color distinto a cada una de esas soluciones. Luego, grafique las semillas ξ_n en el plano x-y con el color de la solución a la que converge. Haga esto para 4 valores distintos de κ.**

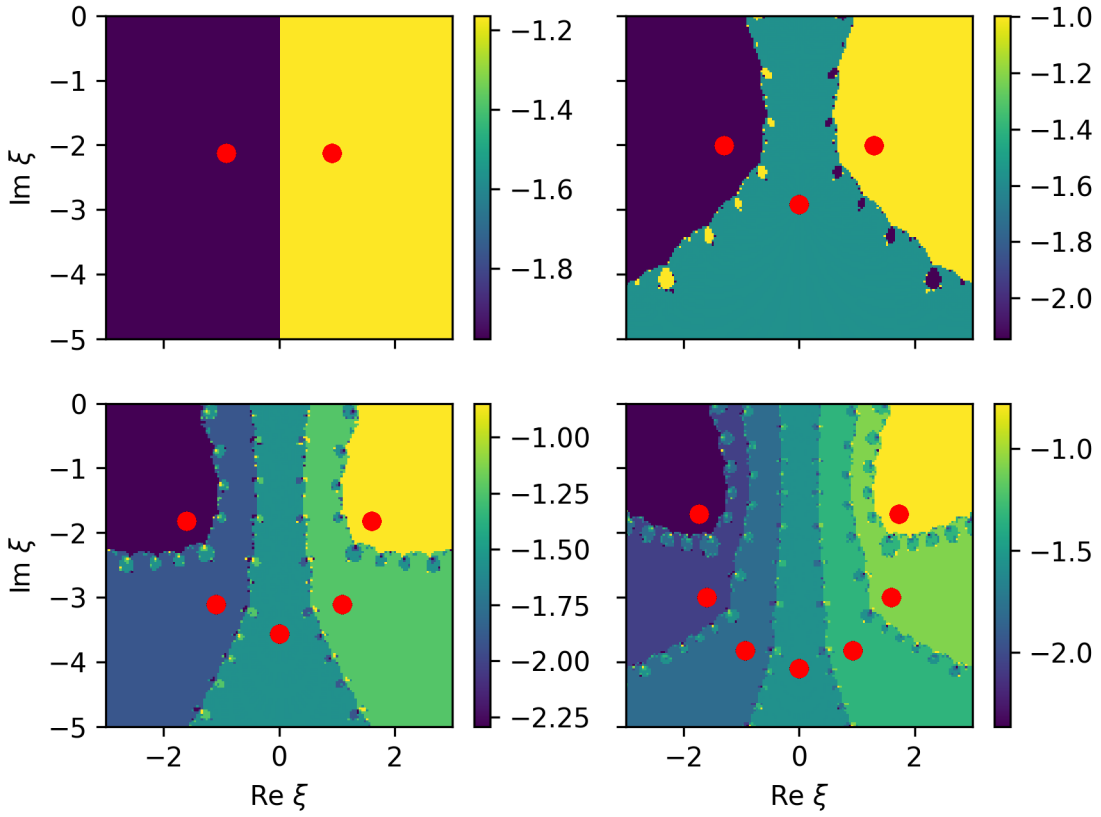
Solución:

Del ítem anterior, partiendo de cada valor de `xi=x+iy`, el método de Newton-Raphson converge a una solución distinta `sol`. Eso quiere decir que podemos *pintar* cada pixel ubicado en (x, y) con un color que asociamos a cada valor de `sol`.

Lo descrito en el párrafo anterior se puede hacer de múltiples formas; acá escogemos el color
0.2 puntos según la *fase* o ángulo θ=`numpy.angle(sol)` del número complejo `sol`=$re^{i\theta}$. Es decir:

0.2 puntos
```
1 plt.pcolormesh(x, y, np.angle(sol))
```

Esto se hace en el script `src/p1-fractal/p1b_fractal.py` para distintos valores de $\kappa = \{2, 3, 5, 7\}$, con lo que se obtienen los gráficos de la figura 2.



0.2 puntos Figura 2: Mapa de convergencia asociado a la ecuación $Z_\kappa(\xi) = 0$ para $\kappa = 2$ (arriba-izquierda), $\kappa = 3$ (arriba-derecha), $\kappa = 5$ (abajo-izquierda) y $\kappa = 7$ (abajo-derecha). Los puntos rojos representan los ceros de la función $Z_\kappa(\xi)$. la escala de colores representa el ángulo o fase de `sol`. Por ejemplo, los pixeles pintados en tonos amarillos son aquellas semillas del método de Newton-Raphson que convergen al cero de $Z_\kappa(\xi)$ con ángulo más cercano a $\theta = 0$; mientras que los tonos azules son las semillas que convergen a ceros con ángulos cercanos a $\theta = -\pi$.

(c) **Dependiendo de ξ_n y de la tolerancia del método de Newton-Raphson, este converge a una solución $\xi = \xi_{\text{sol}}$ en algún número de iteraciones M_n. Asigne un color a cada M_n y grafique ξ_n con el color que usted asignó para M_n. Repita el procedimiento para 4 valores distintos de κ.**

Solución:

En este punto, debemos tener información sobre cuantas iteraciones requiere el método de Newton-Raphson para converger. Para ello, en el método se implementa un contador el cual será devuelto si cierta condición `niter=True` se cumple. Esto se implementa en el script `src/FindRoots.py`:

0.2 puntos

```
1  def NewtonRaphson(f, df, x0, tol=1e-5, niter=False, *args, **kwargs):
2      stop = False
3      x = x0
```

```
    n = 0    # CONTADOR
    while(not stop):
        val = f(x, *args, **kwargs)
        error = val/df(x, *args, **kwargs)
        x = x - error

        stop = abs(error)<tol and abs(val)<tol
        n = n + 1

    if niter:
        return x, n   # si niter=True retorna solucion y num. iteraciones
    else:
        return x      # si niter=False retorna solucion solamente
```

Luego, debemos realizar una pequeña modificación al código mostrado en los ítems anteriores, lo cual puede encontrar en el script `src/p1-fractal/p1c_iterations.py`:

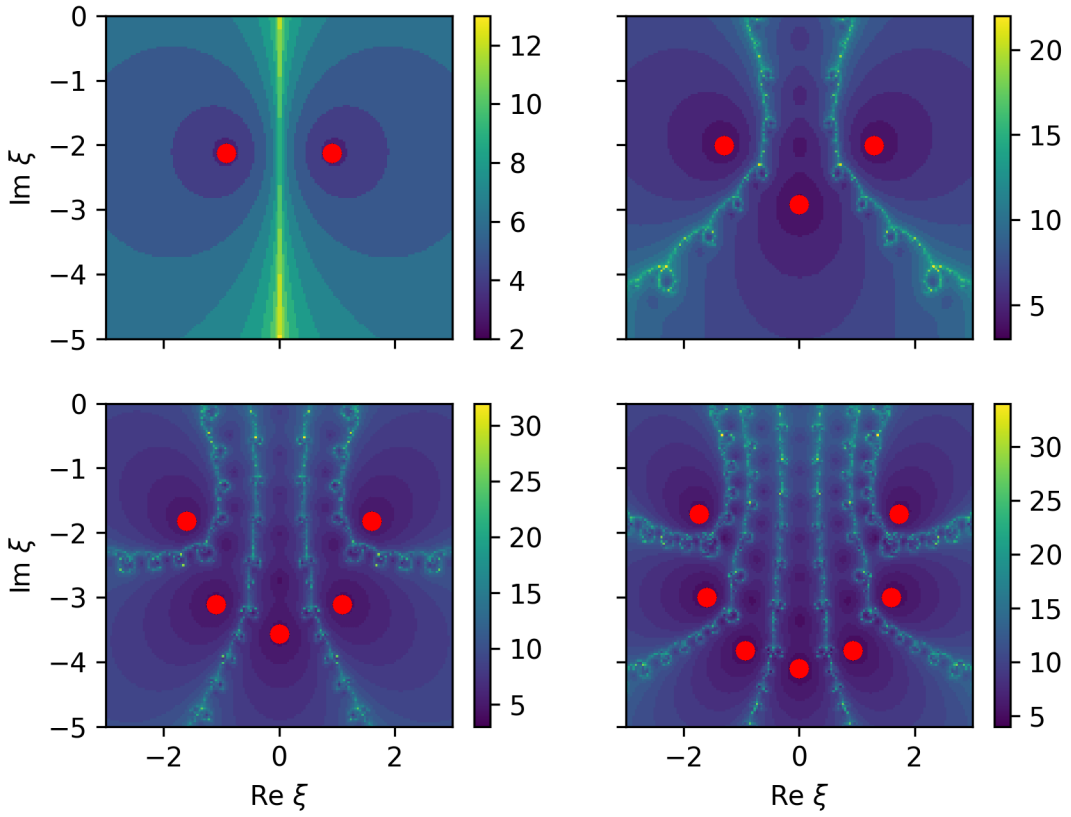
0.2 puntos

```
sol = np.zeros_like(w)
niter = np.zeros(w.shape, dtype="int64")

for i in range(w.shape[0]):
    for j in range(w.shape[1]):
        sol[i,j], niter[i,j] = NewtonRaphson(p, p.deriv(), w[i,j], tol=1e-3,
        ↪  niter=True)

plt.pcolormesh(x, y, niter)
```

Note que al lado izquierdo se definen dos cantidades distintas, `sol` y `niter`, y que dentro de la función `NewtonRaphson` usamos la etiqueta `niter=True`. Luego, graficamos con un código de colores que depende de los valores de `niter`, cuyo resultado puede ver en la figura 3 para distintos valores de $\kappa = \{2, 3, 5, 7\}$.

0.2 puntos

En la figura 3, notemos que las zonas con tonos amarillos (es decir, donde el método de Newton-Raphson requiere más iteraciones para converger a una solución) coinciden con las zonas donde hay transiciones de color en la figura 2. Esto quiere decir que esas zonas corresponden a semillas inestables, muy sensibles a errores numéricos, del método de Newton-Raphson. En general, es recomendable buscar ceros de funciones con otros métodos rudimentarios como el de la secante y luego refinar el cálculo con el método de Newton-Raphson una vez que estamos seguros que las semillas se encuentran cerca de los ceros de una función.



0.2 puntos Figura 3: Número de iteraciones que requiere el método de Newton-Raphson para converger a una solución de $Z_\kappa(\xi) = 0$, lo cual es representado con una barra de colores, para $\kappa = 2$ (arriba-izquierda), $\kappa = 3$ (arriba-derecha), $\kappa = 5$ (abajo-izquierda) y $\kappa = 7$ (abajo-derecha). Los puntos rojos representan los ceros de la función $Z_\kappa(\xi)$.

Pregunta 2

En una esfera sólida de radio $r = a$ conductora de calor, la temperatura se difunde con una longitud de onda λ que satisface:

$$x \cos x + (1 - h) \sin x = 0 \tag{4}$$

donde $x = a/\lambda$ y h es un parámetro que depende del material de la esfera.

Se pide encontrar todas las soluciones $x = x(h)$ de la ecuación (4) en el intervalo $0 < x < 4\pi$. Para ello, resuelva la ecuación de Davidenko usando el método de Runge-Kutta de cuarto orden. En cada iteración, corrija la solución usando el método de Newton-Raphson.

Solución:

Asumimos que los ceros de la ecuación (4) cumplen que $x = x(h)$ es una función de h. Luego, la ecuación de Davidenko asociada a (4) es:

0.5 puntos

$$\frac{dx}{dh} = \frac{\sin x}{(2 - h) \cos x - x \sin x} \,. \tag{5}$$

Notemos que la ecuación (4) tiene una solución trivial en $x = 0$ para todo valor de h, por lo que ignoraremos esta solución en lo que resta de esta respuesta.

Para resolver la ecuación de Davidenko, usaremos el método de Runge-Kutta de cuarto orden. Así, si conocemos $x_i = x(h_i)$, entonces $x(h_{i+1})$ con $h_{i+1} = h_i + \Delta h$ se puede encontrar con el algoritmo mostrado a continuación, donde $a = a(x, h)$ es el lado derecho de la ecuación (5):

0.5 puntos

```
def RungeKutta4th(a, xi, hi, dh):
    K1 = a(xi, hi)
    K2 = a(xi + 0.5*dh*K1, hi+0.5*dh)
    K3 = a(xi + 0.5*dh*K2, hi+0.5*dh)
    K4 = a(xi + dh*K3, hi+dh)

    return xi + (dh/6.0)*(K1 + 2*K2 + 2*K3 + K4)
```

Esta función será llamada para cada valor de h_i y, en cada iteración, la solución que entrega el método de Runge-Kutta será refinada usando el método de Newton-Raphson (definido en la librería `FindRoots`):

0.5 puntos

```
for i in range(h.size-1):
    # Resuelve ecuación de Davidenko para h=hi+dh
    sol = RungeKutta4th(davidenko, xh[i], h[i], dh)

    # Refina solución con el método de Newton-Raphson
    xh[i+1] = [NewtonRaphson(f, df, seed, h=h[i+1]) for seed in sol]
```

Grafique $x(h)$ como función de $0 < h < 20$. Compare estos gráficos con las soluciones analíticas de la ecuación (4) en los casos límite $h = 1$ y $h \to \infty$.

Solución:

Para poder resolver la ecuación diferencial dada por (5), necesitamos condiciones iniciales. Así, buscaremos todas las soluciones de (4) para $h = 0$ en el rango $0 < x(0) < 4\pi$, ignorando la solución trivial $x(0) = 0$. Esto lo haremos mediante el método de la bisección, también definido en el módulo `Biseccion`:

0.5 puntos

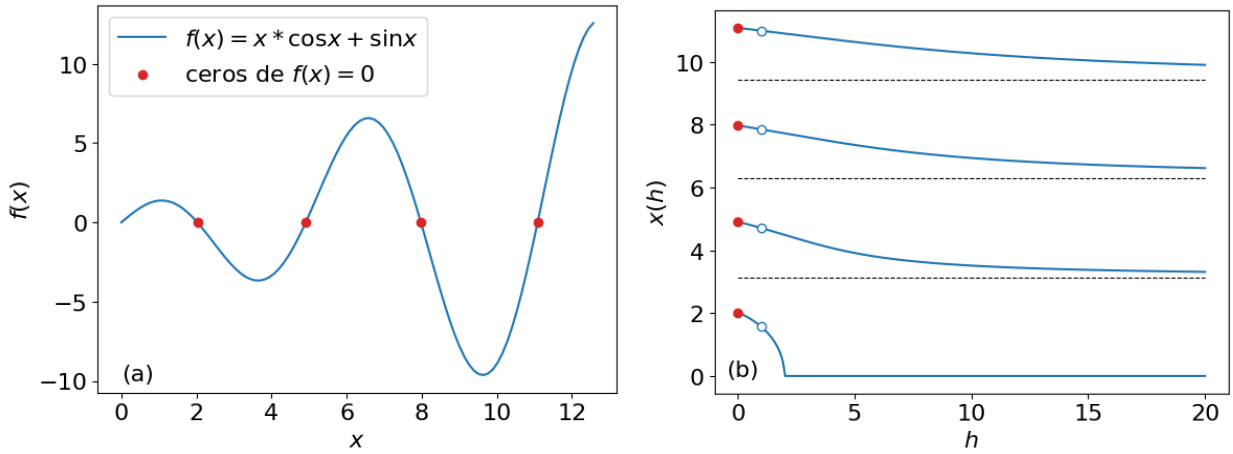
```
# Encuentra todos los ceros de f(x,h)=0 en h=0
x0 = Biseccion(f, 0, 4*np.pi, h=h[0])
```

Solo para comprobar que las condiciones iniciales son correctas, en la figura 4(a) graficamos la función $f(x) = x\cos x + \sin x$ [correspondiente al lado izquierdo de (4) para $h = 0$] y los resultados del método de la bisección en el intervalo $0 < x < 4\pi$.

En la figura 4(b) graficamos las soluciones de la ecuación de Davidenko (5) para cada una de las condiciones iniciales encontradas en la figura 4(a). El código para crear ambas figuras se encuentra en `src/p2-davidenko/davidenko_RK4_newton.py`.

0.5 puntos

Finalmente, notemos que la ecuación (4) permite soluciones analíticas para $h = 1$, i.e. $x(1) = \{\pi/2, 3\pi/2, 5\pi/2, 7\pi/2\}$ [mostrados en la figura 4(b) con puntos blancos]. Además, las soluciones de (4) se acercan asimptóticamente a las soluciones $x(h \to \infty) = \{\pi, 2\pi, 3\pi, 4\pi\}$, las cuales son representadas por líneas segmentadas en la figura 4(b).



0.5 puntos Figura 4: (a) Gráfico de $f(x) = x\cos x + \sin x$ [correspondiente al lado izquierdo de (4) para $h = 0$] y sus ceros en el intervalo $0 < x < 4\pi$. (b) Soluciones de la ecuación de Davidenko (5) entre $0 < h < 20$, donde los puntos rojos corresponden a las condiciones iniciales o soluciones de (4) para $h = 0$ mostrados en el panel (a). La línea segmentada representa la solución asintótica $x(h \to \infty) = m\pi$ con m un número par y los puntos blancos son las soluciones analíticas $x(h = 1) = k\pi/2$ con k un número impar.

Pregunta 3

(a) **Considere una función $f(x)$ cuya expansión en series de Taylor está bien definida hasta segundo orden:**

$$f(x + \Delta) = f(x) + f'(x)\Delta + \frac{1}{2}f''(x)\Delta^2 + O(\Delta^3)\,. \tag{6}$$

Demuestre que esta expansión resulta en una generalización del método de Newton-Raphson, tal que

$$x_{n+1} = \begin{cases} x_n - \dfrac{2f(x_n)}{f'(x_n) + \mathbf{signo}[f'(x_n)]\sqrt{f'(x_n)^2 - 2f''(x_n)f(x_n)}}\,, \\ x_n - \dfrac{f'(x_n) + \mathbf{signo}[f'(x_n)]\sqrt{f'(x_n)^2 - 2f''(x_n)f(x_n)}}{f''(x_n)}\,. \end{cases} \tag{7}$$

Si esta serie converge $x_{n\to\infty} \to x_{\text{sol}}$, lo hará a una de las soluciones de $f(x_{\text{sol}}) = 0$.

Solución:

Supondremos que $x^* = x + \Delta$ es solución de $f(x^*) = 0$, con x un punto muy cercano a la solución x^* y Δ un número lo suficientemente pequeño. Luego, usando la expansión de Taylor (6) en torno a x e ignorando el error $O(\Delta^3)$ en la expansión, entonces:

$$0 = f(x) + f'(x)\Delta + \frac{1}{2}f''(x)\Delta^2\,. \tag{8}$$

Resolviendo para Δ, se encuentra rápidamente que:

$$\Delta = \frac{1}{f''(x)}\left[-f'(x) \pm \sqrt{f'(x)^2 - 2f''(x)f(x)}\right]\,. \tag{9}$$

Por ahora, notemos que si cambiamos el símbolo $\pm$ por $\pm\text{signo}[f'(x)]$, donde $\text{signo}[f'(x)]$ representa el signo de $f'(x)$, el resultado mostrado en (9) no cambia:

0.5 puntos

$$\Delta = \frac{1}{f''(x)}\left[-f'(x) \pm \text{signo}[f'(x)]\sqrt{f'(x)^2 - 2f''(x)f(x)}\right], \tag{10}$$

puesto que si $f'(x) > 0$, entonces $\pm$ mantiene su significado; mientras que si $f'(x) < 0$, entonces solo se invierte el signo de $\pm$.

Sin embargo, la ecuación (10) nos permite analizar mejor qué ocurre con el algoritmo cuando x es cercano a la solución exacta x^*. En este caso, entonces $f(x) \simeq 0$ y, por lo tanto,

$$\text{signo}[f'(x)]\sqrt{f'(x)^2 - 2f''(x)f(x)} \simeq f'(x)\left[1 - 2\frac{f''(x)f(x)}{f'(x)^2}\right],$$

donde hemos usado que $\text{signo}[f'(x)]|f'(x)| = f'(x)$.

Luego, en la ecuación (10), la solución correspondiente al signo $+$ resulta en:

$$\Delta \simeq -2\frac{f(x)}{f'(x)},$$

pero como $f(x) \simeq 0$, errores de redondeo podrían resultar en $\Delta = 0$.

Usualmente queremos evitar errores de redondeo. Como la solución en la ecuación (10) que puede tener errores de redondeo es la que tiene el signo $+$, entonces elegimos como solución correcta la que tiene el signo $-$:

0.5 puntos

$$\Delta_- = \frac{1}{f''(x)}\left[-f'(x) - \text{signo}[f'(x)]\sqrt{f'(x)^2 - 2f''(x)f(x)}\right], \tag{11}$$

mientras que para la otra solución (que es igualmente válida) la racionalizamos de forma inversa; es decir, la reescribimos de modo que el radical quede en el denominador al multiplicar y dividir por $-f'(x) - \text{signo}[f'(x)]\sqrt{\cdots}$. Luego,

0.5 puntos

$$\begin{aligned}\Delta_+ &= \frac{1}{f''(x)}\left[-f'(x) + \text{signo}[f'(x)]\sqrt{f'(x)^2 - 2f''(x)f(x)}\right], \\ &= \frac{2f(x)}{-f'(x) - \text{signo}[f'(x)]\sqrt{f'(x)^2 - 2f''(x)f(x)}}.\end{aligned} \tag{12}$$

De este modo, las dos soluciones dadas por (11) y (12) no (deberían) tener errores de redondeo cuando $f(x) \simeq 0$.

Finalmente, la solución del problema se encuentra como $x^* = x + \Delta_-$ o $x^* = x + \Delta_+$, o bien:

$$x^* = \begin{cases} x - \dfrac{1}{f''(x)}\left[f'(x) + \text{signo}[f'(x)]\sqrt{f'(x)^2 - 2f''(x)f(x)}\right], \\ x - \dfrac{2f(x)}{f'(x) + \text{signo}[f'(x)]\sqrt{f'(x)^2 - 2f''(x)f(x)}}. \end{cases} \tag{13}$$

(b) **Implemente un código que use el método descrito en la ecuación (7).**

Solución:

En el script `src/p3-NewtonCauchy/NewtonCauchy.py` se implementa el algoritmo definido por (13), el cual se muestra a continuación:

1.0 puntos

```
1  while(not stop):
2      fx   = f(x)     # evalua funcion
3      dfx  = df(x)    # derivada
4      ddfx = ddf(x)   # segunda derivada
5
6      denom = dfx + np.sign(dfx) * np.sqrt(dfx**2 - 2 * ddfx * fx)
7
8      # define correccion Delta+ y Delta-
9      minus = denom / ddfx
10     plus  = 2*fx / denom
11
12     # elige la correccion que es menor
13     error = minus if np.abs(minus)<np.abs(plus) else plus
14     x = x - error
15
16     stop = abs(error)<tol and abs(fx)<tol
```

En este código, definimos `minus` $= \Delta_-$ y `plus` $= \Delta_+$. Luego, como queremos que el método converja a una solución, entonces en la línea 13 elegimos el que sea menor en módulo:

0.5 puntos

$$\texttt{error} = \begin{cases} \Delta_- & \text{si } |\Delta_-| < |\Delta_+| \\ \Delta_+ & \text{si } |\Delta_-| > |\Delta_+| \end{cases}$$

(c) **Use el código del ítem anterior para encontrar las soluciones de la ecuación (4) para distintos valores de h (se espera un gráfico de $x(h)$, no valores aislados). Note que la ecuación (7) tiene dos posibles valores de x_{n+1}; elija la versión que haga que $|x_{n+1} - x_n|$ sea mínimo.**

Solución:

Esta sección se responde exactamente igual a la pregunta 2, cambiando la función `NewtonRaphson` del módulo `FindRoots` por la función que definimos en `src/p3-NewtonCauchy/NewtonCauchy.py`, por lo que no profundizaremos en esta parte. El puntaje se entregará a las partes 3(a) y 3(b) solamente.