Tarea 1 Optimización

Roberto Vásquez Martínez
Profesor: Joaquín Peña Acevedo

13/Febrero/2022

1 Ejercicio 1 (6 puntos)

Programar y probar el método de la iteración de Halley para el cálculo de raíces de una función de una variable.

1.1 Descripción del método

El método de Halley usa una aproximación de la función $f(x)$ de segundo orden del desarrollo de Taylor de $f(x)$.

$$f(x_{k+1}) \approx f(x_k) + f'(x_k)\Delta x + \frac{1}{2}f''(x_k)(\Delta x)^2$$

Si igualamos a cero la aproximación tenemos que

$$\Delta x = -\frac{f(x_k)}{f'(x_k) + \frac{1}{2}f''(x_k)\Delta x}$$

El valor Δx en el lado izquierdo de la igualdad corresponde a $\Delta x = x_{k+1} - x_k$, mientras que el que está en el denominador se aproxima por el paso de Newton-Raphson:

$$\Delta x = -\frac{f(x_k)}{f'(x_k)},$$

de modo que

$$x_{k+1} - x_k = -\frac{f(x_k)}{f'(x_k) - \frac{1}{2}f''(x_k)f(x_k)/f'(x_k)},$$

es decir, el método de Halley propone generar la secuencia de puntos mediante la siguiente regla:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k) - \frac{f''(x_k)f(x_k)}{2f'(x_k)}}.$$

1. Escriba la función que aplique el método de Halley. Debe recibir como argumentos un punto inicial x_0, las función $f(x)$, sus derivadas $f'(x)$ y $f''(x)$, el número máximo de iteraciones y un tolerancia $\tau > 0$, similar a la función `NewtonRaphson()` vista en el ejemplo de la clase, de modo que se detenga cuando se cumpla que $|f(x_k)| < \tau$. Defina la variable `res` que indique el resultado obtenido (`res=0` se acabaron las iteraciones y no se encontró un punto que satisfaga el criterio de convergencia, `res=1` el algoritmo converge, `res=-1` hay un problema al evaluar la expresión. La función debe devolver el último punto x_k, $f(x_k)$, el número de iteraciones realizadas y la variable `res`.

2. Pruebe el algoritmo de Halley con las siguientes funciones y puntos inciales:

$$f_1(x) = x^3 - 2x + 1, x_0 = -1000, 1000.$$

$$f_2(x) = 1 + x - \frac{3}{2}x^2 + \frac{1}{6}x^3 + \frac{1}{4}x^4, x_0 = -1000, 1000.$$

En cada caso imprima x_0, $f(x_0)$, x_k, $f(x_k)$, el número de iteraciones k realizadas y el valor de la variable *res*.

3. Repita las pruebas anteriores con el método de Newton-Raphson y escriba un comentario sobre los resultados.

1.2 Solución:

La funciones que implementan tanto el método de Halley como el algoritmo de Newton-Raphson se encuentran el el módulo `lib_t1.py` con los nombres `Halley()` y `NewtonRaphson()`. Al igual que en la clase, considerarmos a lo más 100 iteraciones, es decir, $iterMax = 100$ y una tolerancia de $\tau = 10^{-8}$ que en el código definimos como `tol=1e-8`.

En las siguientes celdas de código probamos el algorimo de Halley con la función f_1.

Para el punto inicial $x_0 = -1000$ tenemos el siguiente resultado.

[1]:
```
from lib_t1 import *
iterMax=100
tol=1e-8
x0=[-1000.0,1000.0]
dic_f1_Halley=Halley(x0[0],f1,der1_f1,der2_f1,iterMax,tol)
print_results(dic_f1_Halley)
```

```
El algoritmo converge y los resultados son:
x0  :  -1000.0
f(x0)  :  -999997999.0
Raiz  :  -1.6180339887504553
f(Raiz)  :  -3.2809310823722626e-12
Iteraciones  :  12
Estado  :  1
```

Por otro lado, con el punto inicial $x_0 = 1000$ obtenemos el siguiente resultado

[2]:
```
dic_f1_Halley=Halley(x0[1],f1,der1_f1,der2_f1,iterMax,tol)
print_results(dic_f1_Halley)
```

```
El algoritmo converge y los resultados son:
x0   :  1000.0
f(x0)   :  999998001.0
Raiz   :  1.000000008768147
f(Raiz)   :  8.768147319315744e-09
Iteraciones   :  13
Estado   :  1
```

Notamos que en ambos casos obtenemos una raíz aunque estas raíces son distintas, una positiva y una negativa. Podemos ver que las raíces de f_1 son

$$x_1 = 1,\ x_2 = \frac{-1-\sqrt{5}}{2} \approx -1.61 \text{ y } x_3 = \frac{-1+\sqrt{5}}{2} \approx 0.6180$$

Con las diferences condiciones iniciales hemos encontrado x_1 y x_2, para hallar x_3 quízas funcione inicializar el algoritmo de Halley con un valor menor a x_1, por ejemplo $x_0 = 0.5$, eso lo probamos en la siguiente celda.

[3]:
```
dic_f1_Halley=Halley(0.5,f1,der1_f1,der2_f1,iterMax,tol)
print_results(dic_f1_Halley)
```

```
El algoritmo converge y los resultados son:
x0   :  0.5
f(x0)   :  0.125
Raiz   :  0.6180339887498948
f(Raiz)   :  0.0
Iteraciones   :  3
Estado   :  1
```

De esta forma hemos encontrado x_3. Ahora procedemos a hallar raíces de f_2 a través del algoritmo de Halley con las condiciones iniciales $x_0 = -1000, 1000$. Para $x_0 = -1000$ tenemos el siguiente resultado.

[4]:
```
dic_f2_Halley=Halley(x0[0],f2,der1_f2,der2_f2,iterMax,tol)
print_results(dic_f2_Halley)
```

```
El algoritmo converge y los resultados son:
x0   :  -1000.0
f(x0)   :  249831832334.33334
Raiz   :  -2.979654185792593
f(Raiz)   :  0.0
Iteraciones   :  15
Estado   :  1
```

Y para $x_0 = 1000$ tenemos lo siguiente

[5]:
```
dic_f2_Halley=Halley(x0[1],f2,der1_f2,der2_f2,iterMax,tol)
print_results(dic_f2_Halley)
```

```
El algoritmo converge y los resultados son:
x0  :  1000.0
f(x0)  :  250165167667.66666
Raiz  :  -0.546729731825445
f(Raiz)  :  1.734723475976807e-17
Iteraciones  :  19
Estado  :  1
```

Se puede ver que ambas son raíces y además son las únicas dos raíces reales de la función f_2.

Repetiremos lo anterior con el método de Newton-Raphson. Para f_1 con condición inicial $x_0 = -1000$ tenemos lo siguiente.

[6]:
```
dic_f1_NR=NewtonRaphson(x0[0],f1,der1_f1,iterMax,tol)
print_results(dic_f1_NR)
```

```
El algoritmo converge y los resultados son:
x0  :  -1000.0
f(x0)  :  -999997999.0
Raiz  :  -1.6180339888222295
f(Raiz)  :  -4.234541606251696e-10
Iteraciones  :  20
Estado  :  1
```

Y para $x_0 = 1000$ obtenemos lo siguiente

[7]:
```
dic_f1_NR=NewtonRaphson(x0[1],f1,der1_f1,iterMax,tol)
print_results(dic_f1_NR)
```

```
El algoritmo converge y los resultados son:
x0  :  1000.0
f(x0)  :  999998001.0
Raiz  :  1.0000000028195468
f(Raiz)  :  2.8195468182445893e-09
Iteraciones  :  22
Estado  :  1
```

El algoritmo de Newton-Raphson encontró las mismas raíces que el algoritmo de Halley respecto a las condiciones iniciales, sin embargo, lo hizo en más iteraciones. Ahora probamos el método de Newton-Raphson con f_2 y la condición inicial $x_0 = -1000$.

[8]:
```
dic_f2_NR=NewtonRaphson(x0[0],f2,der1_f2,iterMax,tol)
print_results(dic_f2_NR)
```

```
El algoritmo converge y los resultados son:
x0  :  -1000.0
f(x0)  :  249831832334.33334
Raiz  :  -2.9796541859258454
f(Raiz)  :  1.6091874499579717e-09
Iteraciones  :  25
```

```
Estado  :  1
```

E inicializando con $x_0 = 1000$ tenemos

[9]:
```
dic_f2_NR=NewtonRaphson(x0[1],f2,der1_f2,iterMax,tol)
print_results(dic_f2_NR)
```

```
El algortimo no converge
```

Con esta función con la condición inicial $x_0 = -1000$ encontramos la mismas raíz que con el algoritmo de Halley, aunque otra vez con más iteraciones. Por otro lado, inicializando con $x_0 = 1000$ el algoritmo no converge, es decir, `res=0`, se alcanza el número máximo de iteraciones sin que se cumpla la condición de paro, por lo que se puede inferir que para hallar la segunda raíz de f_2 podría converge con un número mucho mayor de iteraciones que con las que lo hizo el algoritmo de Halley, y esto se debe a que la aproximación que usa el algoritmo de Halley es de orden mayor que la usado en Newton Raphson, y por lo tanto hay menos propagación de error en el algoritmo de Halley. Otra posibilidad es que se requiere una mejor elección de la condición inicial para el algoritmo de Newton-Raphson, si consideramos inicializar con un número mucho más cercano a la segunda raíz por la derecha que 1000, por ejemplo $x_0 = 1$ como condición inicial obtenemos lo siguiente.

[10]:
```
dic_f2_NR=NewtonRaphson(1.0,f2,der1_f2,iterMax,tol)
print_results(dic_f2_NR)
```

```
El algoritmo converge y los resultados son:
x0  :  1.0
f(x0)  :  0.9166666666666666
Raiz  :  -0.5467297318263349
f(Raiz)  :  -2.336894566745684e-12
Iteraciones  :  51
Estado  :  1
```

Por lo que también podemos inferir que el algoritmo de Newton-Raphson es mucho mas susceptible a la elección de condiciones iniciales. Y a pesar de la condición inicial seleccionada, el algoritmo de Halley es mucho mas eficiente en el número de iteraciones lo que muestra el poder de los métodos basados en una aproximación de orden mayor, con el costo de tener que obtener la segunda derivada.

2 Ejercicio 2 (4 puntos)

Una manera de aproximar la función $\cos(x)$ es mediante la función

$$C(x;n) = \sum_{i=0}^{n} c_i$$

donde n es un parámetro que indica la cantidad de términos en la suma y

$$c_i = -c_{i-1}\frac{x^2}{2i(2i-1)} \quad \text{y} \quad c_0 = 1.$$

1. Programe la función $C(x;n)$.
2. Imprima el valor del error $C(x;n)-1$ para $x \in \{2\pi, 8\pi, 12\pi\}$ y $n = 10, 50, 100, 200$.
3. Imprima el valor del error $C(x;n)+1$ para $x \in \{\pi, 9\pi, 13\pi\}$ y $n = 10, 50, 100, 200$.
4. Comente sobre el comportamiento de los errores obtenidos y cuál sería una manera apropiada de usar esta función.

2.1 Solución:

En primer lugar, programamos esta función cuyo código fuente se encuentra en el módulo `lib_t1.py`.

Para el numeral 2, imprimimos el vector $|C(x;n)-1|$ para $n \in \{10, 50, 100, 200\}$ y $x \in \{2\pi, 8\pi, 12\pi\}$, por propiedades básicas de la función coseno en cada uno de los valores x se tiene $\cos(x) = 1$, por lo que $|C(x;n)-1|$ es el vector de errores que se comete para cada x dado n. Imprimimos la evaluación de la función coseno en el vector de valores $[2\pi, 8\pi, 12\pi]$ iterando sobre cada valor de n.

[12]:
```
import math as mt
import numpy as np
list_n=[10,50,100,200]
values_even=np.array([2.0*mt.pi,8.0*mt.pi,12.0*mt.pi])
for n in list_n:
    print(f'Para n={n} |C(x;n)-1| es igual a:␣
 ↪',abs(pol_taylor_cos(values_even,n)-1.0))
```

```
Para n=10 |C(x;n)-1| es igual a:  [3.01224042e-04 2.51526888e+09 1.08015596e+13]
Para n=50 |C(x;n)-1| es igual a:  [4.66293670e-15 4.86928924e-07 1.91766915e-01]
Para n=100 |C(x;n)-1| es igual a:  [4.66293670e-15 4.86928924e-07
1.35724564e-01]
Para n=200 |C(x;n)-1| es igual a:  [4.66293670e-15 4.86928924e-07
1.35724564e-01]
```

Haciendo el mismo ejercicio, imprimimos el vector de errores $|C(x;n)+1|$ para $n \in \{10, 50, 100, 200\}$ y $x \in \{3\pi, 9\pi, 13\pi\}$

[13]:
```
values_odd=np.array([3.0*mt.pi,9.0*mt.pi,13.0*mt.pi])
for n in list_n:
    print(f'Para n={n} |C(x;n)-1| es igual a:␣
 ↪',abs(pol_taylor_cos(values_odd,n)+1.0))
```

```
Para n=10 |C(x;n)-1| es igual a:  [2.07517810e+00 2.90378712e+10 5.53930634e+13]
Para n=50 |C(x;n)-1| es igual a:  [2.14495088e-13 1.09779085e-05 1.94408879e+02]
Para n=100 |C(x;n)-1| es igual a:  [2.14495088e-13 1.09779085e-05
1.46360832e+00]
Para n=200 |C(x;n)-1| es igual a:  [2.14495088e-13 1.09779085e-05
1.46360832e+00]
```

Sabemos que teóricamente el valor de $cos(x)$ es 1 para todo $x \in \{2\pi, 8\pi, 12\pi\}$ y -1 para todo $\{3\pi, 9\pi, 12\pi\}$, sin embargo, en base a los resultados anteriores la aproximación $C(x;n)$ para n fijo comete mayor error cuanto más crece el valor de x.

Para explicar esto recurrimos el **Teorema de Taylor**. Es claro que $C(x;n)$ es el polinomio de Taylor de grado $2n$ de la función coseno, luego por el Teorema de Taylor se tiene que

$$\cos x = C(x;n) + R_{2n}(x),$$

donde

$$R_{2n}(x) = \cos^{(2n+1)}(t)\frac{x^{2n+1}}{(2n+1)!} \text{ para algún } t \in [0,x],$$

y $\cos^{(2n+1)}$ representa la derivada de orden $2n+1$ de la función coseno.

Por lo tanto, tenemos que el error en la aproximación tiene la forma

$$|\cos x - C(x;n)| = |\cos^{(2n+1)}(t)| \cdot |\frac{x^{2n+1}}{(2n+1)!}|,$$

luego cuando x se hace grande el factor $\frac{x^{2n+1}}{(2n+1)!}$ crece exponencialmente con n fijo, por tanto el error en la aproximación crece exponencialmente a medida que x crece.

Sin embargo, podemo reducir el factor de crecimiento del error utilizando que la función coseno es periódica. Para $x \geq 0$ arbitrario sea

$$n_x = \max\{n \in \mathbb{N} \cup \{0\} |\ x - 2\pi \cdot n > 0\}.$$

Por lo tanto, si $t_x = x - 2\pi \cdot n_x$ entonces por la definición de n_x se tiene que $t_x \in [0, 2\pi)$ y t_x es el menor real positivo tal que

$$\cos(x) = \cos(t_x).$$

Para $x \leq 0$ utilizamos la paridad de la función coseno y consideramos lo anterior con $x := -x$.

Por lo que aproximamos $\cos x$ a través de $C(t_x;n)$, y en este caso podemos acotar el error de la siguiente forma

$$|\cos(x) - C(t_x;n)| \leq |\frac{t_x^{2n+1}}{(2n+1)!}| \leq \frac{(2\pi)^{2n+1}}{(2n+1)!},$$

y observamos que el error se va 0 más rápido cuando $n \to \infty$, pues hemos acotado por un valor que ya no depende de x.

Concluimos así que para utilizar la aproximación a $\cos(x)$ es más conveniente usar $C(t_x;n)$ que $C(x;n)$, especialmente para n pequeños pues es el caso en el que el denominador $(2n+1)!$ no tiene tanto impacto; además al utilizar t_x el error se irá mucho más rápido a 0 a medida que $n \to \infty$.