Tarea 1

February 10, 2022

1 Curso de Optimización (DEMAT)

1.1 Tarea 1

Descripción:	Fechas
Fecha de publicación del documento:	**Febrero 3, 2022**
Fecha límite de entrega de la tarea:	**Febrero 13, 2022**

1.1.1 Indicaciones

El propósito de esta tarea es poner en praćtica lo que hemos revisado sobre Python, por lo que los ejercicios son de programación.

Puede escribir el código de los algoritmos que se piden en una celda de este notebook o si lo prefiere, escribir las funciones en un archivo `.py` independiente e importar la funciones para usarlas en este notebook. Lo importante es que en el notebook aparezcan los resultados de la pruebas realizadas y que:

- Si se requieren otros archivos para poder reproducir los resultados, para mandar la tarea cree un archivo ZIP en el que incluya el notebook y los archivos adicionales.
- Si todos los códigos para que se requieren para reproducir los resultados están en el notebook, no hace falta comprimirlo y puede anexar sólo el notebook en la tarea del Classroom.
- Exportar el notebook a un archivo PDF y anexarlo en la tarea del Classroom como un archivo independiente. **No lo incluya dentro del ZIP**, porque la idea que lo pueda accesar directamente para poner anotaciones y la calificación de cada ejercicio.

En la descripción de los ejercicios se nombran algunas variables para el algoritmo, pero sólo es para facilitar la descripción. En la implementación pueden nombrar sus variables como gusten.

En los algoritmos se describen las entradas de las funciones. La intención es que tomen en cuenta lo que requiere el algoritmo y que tiene que haber parámetros que permitan controlar el comportamiento del algoritmo, evitando que dejen fijo un valor y que no se puede modificar para hacer diferentes pruebas. Si quieren dar esta información usando un tipo de dato que contenga todos los valores o usar variables por separado, etc., lo pueden hacer y no usen variables globales si no es necesario.

Lo mismo para los valores que devuelve una función. Pueden codificar como gusten la manera en que regresa los cálculos. El punto es que podamos tener acceso a los resultados, sin usar variables globales, y que la función no sólo imprima los valores que después no los podamos usar.

1.2 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.

1.2.1 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.

1.2.2 Solución:

[ ]:
```
import sympy

# Clase Funciones de R -> R, con derivadas
class RealFunction:

    #Para que se pueda evaluar la funcion como f(x0), x0\in R
    def __call__(self, x):
        return self._evaluate(x)

    #Inicialización de función
    def __init__(self, parameters = [],  expression = "", function = None):
        self._paramaters = parameters
        self._my_symbols = {'x': sympy.Symbol('x', real=True)}
        if expression != "":
            self._expression = expression
            self._my_func = sympy.parsing.sympy_parser.parse_expr(expression,␣
 ↪self._my_symbols)
        else:
            self._my_func = function
        self._evaluate = sympy.utilities.lambdify(self._my_symbols['x'], self.
 ↪_my_func, "sympy")
        self._der = dict()

    #Derivadas de la función
    def der(self, ord):
        if(ord not in self._der):
            _derk = self._my_func
            for _ in range(ord):
                _derk = sympy.diff(_derk, self._my_symbols['x'])
            self._der[ord] = self.__class__(parameters = self._paramaters,␣
 ↪function = _derk)
        return self._der[ord]
i=100
f = RealFunction(expression = "x")
print(type(lambda x : x))
```

```
<class 'function'>
```

[ ]:
```
# Implementación del método de Newton-Raphson.
# Pueden modificar la función si lo desean.
def NewtonRaphson(x0, f, iterMax, tol):
    xk  = x0
    res = 0
    for k in range(iterMax):
        fk = f(xk)
        if fk<tol and fk>-tol:
            res = 1
            break
        else:
            dfk = f.der(1)(xk)
            if dfk!=0:
                xk = xk - fk/dfk
            else:
                res = -1
                break
    return xk,k,res
```

[ ]:
```
# Implementación del método de Halley.
def Halley(x0, f, iterMax, tol):
    xk  = x0
    res = 0
    for k in range(iterMax):
        fk = f(xk)
        if fk<tol and fk>-tol:
            res = 1
            break
        else:
            dfk = f.der(1)(xk)
            if dfk!=0:
                d2fk = f.der(2)(xk)
                if (dfk - (d2fk*fk)/(2*dfk)) != 0:
                    xk = xk - fk/(dfk - (d2fk*fk)/(2*dfk))
                else:
                    res = -1
                    break
            else:
                res = -1
                break
    return xk,k,res
```

[ ]:
```
# Esta celda o en otras adicionales pueden poner las pruebas realizadas.

#Impresión del resultado del metodo de Halley
def solve(f, x0, iterMax = 100, tol = 1):
    xk1,k1,res1 = Halley(x0, f, iterMax, tol)
```

```
    xk2,k2,res2 = NewtonRaphson(x0, f, iterMax, tol)

    print("Para la funcion real f(x) = {}\nCon punto inicial x0 = {}, donde␣
 ↪f(x0) = {}".format(f._my_func, x0, f(x0)))
    print("\tEl metodo de Haylle, obtuvo res = {}, es decir:\n".format(res1))
    if res1==1:
        print('\t\tEncontro una raiz cerca de xk = {}\n\t\tdonde f(xk) =␣
 ↪{},\n\t\ten {} iteraciones'.format(xk1,f(xk1), k1))
    elif res1==0:
        print('\t\tNo converje.')
    else:
        print('\t\tTuvo un problema al evaluar la expresion.')
    print("")

    print("\tEl metodo de Newton, obtuvo res = {}, es decir:\n".format(res2))
    if res2==1:
        print('\t\tEncontro una raiz cerca de xk = {}\n\t\tdonde f(xk) =␣
 ↪{},\n\t\ten {} iteraciones'.format(xk2,f(xk2), k2))
    elif res2==0:
        print('\t\tNo converje.')
    else:
        print('\t\tSe cancelo la derivada en algun momento')
    print("")
    print("")

f1 = RealFunction(expression = "x**3-2*x+1")
f2 = RealFunction(expression = "x**4/4 + x**3/6 - x**2*3/2 + x + 1")

solve(f1, -1000, iterMax = 1000000, tol = 1e-3)
solve(f1, 1000, iterMax = 100, tol = 1e-3)
solve(f2, -1000, iterMax = 1000000, tol = 1e-3)
solve(f2, 1000, iterMax = 1000000, tol = 1e-3)
```

```
Para la funcion real f(x) = x**3 - 2*x + 1
Con punto inicial x0 = -1000, donde f(x0) = -999997999
        El metodo de Haylle, obtuvo res = 1, es decir:

                Encontro una raiz cerca de xk = -1.6181367432437983
                donde f(xk) = -0.0006015865378534713,
                en 11 iteraciones

        El metodo de Newton, obtuvo res = 1, es decir:

                Encontro una raiz cerca de xk = -1.6180433288492904
                donde f(xk) = -5.467831769623288e-05,
                en 19 iteraciones
```

```
Para la funcion real f(x) = x**3 - 2*x + 1
Con punto inicial x0 = 1000, donde f(x0) = 999998001
        El metodo de Haylle, obtuvo res = 1, es decir:

                Encontro una raiz cerca de xk = 1.000000008768147
                donde f(xk) = 8.768147319315744e-09,
                en 13 iteraciones

        El metodo de Newton, obtuvo res = 1, es decir:

                Encontro una raiz cerca de xk = 1.0000306594623785
                donde f(xk) = 3.0662282415105935e-05,
                en 21 iteraciones


Para la funcion real f(x) = x**4/4 + x**3/6 - 3*x**2/2 + x + 1
Con punto inicial x0 = -1000, donde f(x0) = 249831832334.33334
        El metodo de Haylle, obtuvo res = 1, es decir:

                Encontro una raiz cerca de xk = -2.9796542579895906
                donde f(xk) = 8.71868907736939e-07,
                en 14 iteraciones

        El metodo de Newton, obtuvo res = 1, es decir:

                Encontro una raiz cerca de xk = -2.9796666684075026
                donde f(xk) = 0.00015074476346566001,
                en 24 iteraciones


Para la funcion real f(x) = x**4/4 + x**3/6 - 3*x**2/2 + x + 1
Con punto inicial x0 = 1000, donde f(x0) = 250165167667.66666
        El metodo de Haylle, obtuvo res = 1, es decir:

                Encontro una raiz cerca de xk = -0.5467297372430081
                donde f(xk) = -1.4227718336812245e-08,
                en 18 iteraciones

        El metodo de Newton, obtuvo res = 0, es decir:

                No converje.
```

Esta celda es para el comentario:

Podemos ver con las pruebas anteriores que el método de Haylley converge más rapido a las raices (orden cuadrático de convergencia, en lugar del orden lineal del método de Newton) y por ello

encuentra como en el último caso raices que el de Newton no. Esto puesto que hace uso de más información de la función, la segunda derivada, en caso de no ser 2 veces diferenciable no podríamos hacer uso de este.

1.3 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.

1.3.1 Solución:

[ ]:
```
# En esta celda puede poner el código de la función
# o poner la instrucción para importar la función de un archivo .py

import math
import numpy as np

class FunctionSequence:

    def __call__(self, x, i):
        if(self._a[i] == None):
            self._a[i] = RealFunction(function = (-1)*self.
 ↪__call__(x,i-1)*(self._f._my_func**2/(2*i*(2*i-1))))
        return self._a[i](x)

    def __init__(self, N, a0):
        if(N > 1e6):
            raise ValueError("The number of elements is to large.")
        self._a = [None] * N
        self._a[0] = RealFunction(function = a0)
        self._f = RealFunction(expression = "x")

def cos_aproximation(x, N):
    cos1 = 1.0
    cos2 = 1.0
```

```
    c = FunctionSequence(10000, 1)
    ci = 1.0
    for i in range(1,N+1):
        cos1 += c(x, i)
        ci = (-ci*x**2)/(2*i*(2*i-1))
        cos2 += ci
    return cos1, cos2
```

[ ]:
```
# En esta celda o en otra adicionales puede poner las
# pruebas realizadas

print("El formato de los resultados es:\n0 | N\n{}\nx | cos(x;N)".
 ↪format(12*"-"))

print("\nAproximación de C(x;n)")
N = np.array([10,50,100,200])
x = np.array([2,8,12])*np.pi
x_1, N_1 = np.meshgrid(x, N)
aprox_cos_x_n = np.vectorize(cos_aproximation)(x_1, N_1)
print(np.vstack((np.concatenate([[0],x]),np.
 ↪column_stack((N,aprox_cos_x_n[1])))))
print("\nValor del error de C(x;n)-1:")
print(np.vstack((np.concatenate([[0],x]),np.
 ↪column_stack((N,aprox_cos_x_n[1]-1)))))

print("\nAproximación de C(x;n)")
N = np.array([10,50,100,200])
x = np.array([1,9,13])*np.pi
x_1, N_1 = np.meshgrid(x, N)
aprox_cos_x_n = np.vectorize(cos_aproximation)(x_1, N_1)
print(np.vstack((np.concatenate([[0],x]),np.
 ↪column_stack((N,aprox_cos_x_n[1])))))
print("\nValor del error de C(x;n)+1:")
aprox_cos_x_n = np.vectorize(cos_aproximation)(x_1, N_1)
print(np.vstack((np.concatenate([[0],x]),np.
 ↪column_stack((N,aprox_cos_x_n[1]+1)))))
```

```
El formato de los resultados es:
0 | N
------------
x | cos(x;N)

Aproximación de C(x;n)
[[0.00000000e+00 6.28318531e+00 2.51327412e+01 3.76991118e+01]
 [1.00000000e+01 1.00030122e+00 2.51526888e+09 1.08015596e+13]
 [5.00000000e+01 1.00000000e+00 1.00000049e+00 1.19176692e+00]
 [1.00000000e+02 1.00000000e+00 1.00000049e+00 1.13572456e+00]
```

```
 [2.00000000e+02 1.00000000e+00 1.00000049e+00 1.13572456e+00]]

Valor del error de C(x;n)-1:
[[ 0.00000000e+00  6.28318531e+00  2.51327412e+01  3.76991118e+01]
 [ 1.00000000e+01  3.01224042e-04  2.51526888e+09  1.08015596e+13]
 [ 5.00000000e+01 -4.66293670e-15  4.86928924e-07  1.91766915e-01]
 [ 1.00000000e+02 -4.66293670e-15  4.86928924e-07  1.35724564e-01]
 [ 2.00000000e+02 -4.66293670e-15  4.86928924e-07  1.35724564e-01]]

Aproximación de C(x;n)
[[ 0.00000000e+00  3.14159265e+00  2.82743339e+01  4.08407045e+01]
 [ 1.00000000e+01 -1.00000000e+00  2.90378712e+10  5.53930634e+13]
 [ 5.00000000e+01 -1.00000000e+00 -1.00001098e+00  1.93408879e+02]
 [ 1.00000000e+02 -1.00000000e+00 -1.00001098e+00  4.63608316e-01]
 [ 2.00000000e+02 -1.00000000e+00 -1.00001098e+00  4.63608316e-01]]

Valor del error de C(x;n)+1:
[[ 0.00000000e+00  3.14159265e+00  2.82743339e+01  4.08407045e+01]
 [ 1.00000000e+01  7.56507079e-11  2.90378712e+10  5.53930634e+13]
 [ 5.00000000e+01 -2.22044605e-16 -1.09779085e-05  1.94408879e+02]
 [ 1.00000000e+02 -2.22044605e-16 -1.09779085e-05  1.46360832e+00]
 [ 2.00000000e+02 -2.22044605e-16 -1.09779085e-05  1.46360832e+00]]
```

Esta celda es para el comentario:

Por las pruebas realizadas anteriormente podemos observar que mientras x se hace más grande también el error de nuestra aproximación de $cosx$ se hace más grande por lo que requeriría de una mayor cantidad de iteraciones para una mejor aproximación. Recordemos que cos es una función periódica por lo que podemos limitarnos a tomar $x \in [0, 2\pi)$