1 Raíces de funciones no lineales - Parte 3

1.1 Presentacion de la covergencia del metodo Newton-Rapshon

En este apartado queremos ilustrar como converge el metodo Newton-Rapshon. Para ello, primero recordamos que se denominan a K constante y a k orden de convergencia, a los números tales que:

$$\lim_{n\to\infty} \frac{|x_0^{n+1} - x_0|}{|x_0^n - x_0|^k} = K$$

Es conocido que en el caso de Newton-Rapshon el orden de convergencia es cuadrático y su constante de convergencia tiene valor:

$$K = \left| \frac{1 f''(x_0)}{2 f'(x_0)} \right| \qquad \text{En donde } f \text{ es la funcion analizada y } x_0 \text{ es la raiz obtenida} \tag{1}$$

No obstante, no conocemos como se comporta esta K, según el orden de convergencia. Por ello, para ilustrar dicho comportamiento realizaremos un ejemplo practica donde anotaremos en una tabla los valores de K y E_n (error entre la estimación y la raiz real), según el numero de iteraciones (n) y según el valor del orden de convergencia.

1.2 Convergencia para f(x) $= x^3 - 3x + 2$ y $x_0 = -3$

Para analizar la convergencia de f(x), tomamos una tolerancia de $\varepsilon = 1 * 10^{-6}$ y modificamos el programa inicial del metodo de Newton-Rapshon para que admitiera un nuevo parámetro que le indicase la iteración en la que debía detenerse. Además, creamos una función auxilliar que encapsula al metodo de Newton-Rapshon, cuyo objetivo es el de devolver el valor final del metodo para una iteración dado, asi como el error cometido en esa iteración y un array con los K para los ordenes de convergencia k=1, k=2 y k=3. Que tiene el siguiente aspecto:

```
def conv_NewtonRapshon(f,x_m,epsilon,limit,x_0):

    if(limit>0):
        #SI limit NO ES 0,APLICAMOS NEWTON_RAPSON HASTA LA ITERACION limit(n)
        aux=metodo_NewtonRaphson(f,x_m,epsilon,limit=limit)
        """
        Haciendo limit =limit, basicamente estamos especificando que la variable
        limit independientemente de donde se declarese en la cabecera de la
        funcion metodo_NewtonRaphson tenga valor limit
        ''"""
    else:
        # SI limit ES CERO, ENTONCES X_N=x_m
        aux=[x_m]
    #VALOR DEVUELTO POR NEWTON_RAPSON EN LA ITERACION limit(n+1)
    aux1 = metodo_NewtonRaphson(f,x_m,epsilon,limit=limit+1)

    #ERRORES E_n y E_n+1
    e_n = aux[0]-x_0
    e_n1= aux1[0]-x_0

    k=[]
    #ALMACENAMOS LOS K SEGUN la k
    for i in range(1,4):
        k.append(abs(e_n1)/(abs(e_n)**i))
    return aux[0],e_n,k
```

Tras esto, implementamos un bucle en el se mostraban los resultados de la aplicacion de este metodo para valores de limit entre 0 y 4 (iteracion a partir de la cual el metodo converge por lo que los siguientes resultados carecerián de significado), donde obtuvimos los resultados que se reflejan en la siguiente tabla:

	x_0^n	E_n	$\|E_{n+1}\|/\|E_{n+1}\|^k$ k = 1	$\|E_{n+1}\|/\|E_{n+1}\|^k$ k = 2	$\|E_{n+1}\|/\|E_{n+1}\|^k$ k = 3
n= 0	-3	-1	0.055555	0.055555	0.055555
1	-2.055555	-0.055555	0.035087	0.631578	11.36842
2	-2.001949	-0.001949	0.001297	0.665369	341.334630
3	-2.000002	-0.000002	0.000001	0.666659	263679.089
4	-2.000000	$-4.2614 * 10^{-12}$	0	0	0

Tabla 1.

De aquí podemos observar varias cosas, la primera es que la convergencia del metodo se da en n=5 como indicamos antes lo cual explica porque las contantes de convergencia acaban siendo 0 para n=4.

Por otro lado, podemos observar que mientras la K para el orden de convergencia 3 parece divergir, la de los ordenes 1 y 2 parece estabilizarse en los valores 0 y $0.\hat{6}$. Por tanto podemos asegurar que el orden de convergencia es 2 ya que es el mayor orden que obtenemos y que la constante de convergencia es algo cercano a $0.\hat{6}$, lo cual podemos verificar haciendo uso de la formula (1), en donde tendriamos:

$f'(x) = 3x^2 - 3 \quad , \quad f''(x) = 6x \quad , \quad x_0 = -2$

por lo que :

$$K = \left| \frac{1 * -12}{2 * (12 - 3)} \right| = \left| \frac{1 * -4}{2 * 3} \right| = \frac{2}{3} = 0.\hat{6}$$

Con lo que no solo comprobamos de manera analítica que el orden de convergencia del metodo es de orden 2, si no que tambien vemos que la formula que conocemos para obtener la constante de convergencia se cumple.

2 Resolución numérica de sistemas lineales

En este apartado estamos interesados en la resolución de sistemas de ecuaciones lineales. En este caso centramos nuestra atención en los algoritmo de Gauss-Seidel y en el de Jacobi los cuales descibiremos a continuación.

Para ello vamos a tratar con sistemas de ecuaciones donde se considerará la siguiente notación.

Notación 1.

Dado un sistema de ecuaciones Ax=b denotaremos los elementos de A como a_{kj} *donde la k representa la fila y j la columna,* x_j^i *donde i es la iteración y j el elemento del vector y* b_j *donde j es el elemento del vetor.*

2.1 Descripción e implementación de los algoritmos

Los dos algoritmos que se van a implementar, pertenencen a la categoría de algoritmos iterativos, este tipo de algoritmos destacan por su simplicidad y por estar adaptados a sistemas grandes con matrices dispersas.

Además, estos algoritmos resultan ser convergentes si se da la condición de que son diagonalmente dominantes o lo que es lo mismo, si:

$$|a_{\mathrm{ii}}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{\mathrm{ij}}|$$

La condición de convergencia de ambos algoritmos es identica y se basa en comprobar elemento a elemento el valor absoluto de la resta de la solución en la iteración anteriror con la solución en la iteración siguiente. De tal forma que si que en todos los elementos de los vectores se cumple que el valor absoluto de las restas es menor a epsilon entonces el algoritmo habrá alcanzado la condición de parada.

Este comprobación fue implementada en un metodo, la cual devuelve un booleano a True si se da la condición de parada. Este metodo se presenta a continuación:

```
def convergencia(e,x_k,x_k1):
    i=0
    conv = True
    while(i<len(x_k) and conv):
        if(abs(x_k[i]-x_k1[i])> e):
            conv = False
        i+=1
    return conv
```

tras esto se procedió a la implementación de los dos algoritmos.

Gauss-Seidel

El metodo Gauss-Seidel se retro alimenta puesto que los elementos del vector x se van actualizando no solo segun los elementos x de la iteración anterior si no que tambien con los elementos x ya calculados de esta iteración, la forma en la que se lleva a cabo esto es siguiendo la siguiente fórmula:

$$x_j^i = \frac{b_j - \sum_{k=1}^{j-1} a_{\mathrm{jk}} * x_k^i - \sum_{k=i+1}^{n} a_{\mathrm{jk}} * x_k^{i-1}}{a_{\mathrm{jj}}} \tag{2}$$

Dicha formula y el algoritmo como tal fueron implementados en una función como se muestra a continuación:

```
def Gauss_Seidel(A,b,x_0,e):
    #De momento  es igual a jacobi
    x_k1 = np.copy(x_0)+2*e
    k=1
    x_k= np.copy(x_0)
    while(not convergencia(e,x_k1,x_k)):

        x_k1=np.copy(x_k)

        for i in range(len(x_0)):
            aux = b[i]
            for j  in range(len(x_0)):
                if(j != i):
                    aux -=A[i,j]*x_k[j]#En vez de usar el vector de la iteracion
anterior
                    #usamos los valores del de la itearacion actual que depende de
la fila
                    # en la que estemos habran sido ya modificados
            x_k[i]=aux/A[i,i]
        print(k,":",x_k)
        k+=1

    return x_k
```

En donde se puede observar que el algoritmo usa para implementar la fórmula (2) un vector que se inicializa con los valores de la iteración anterior y que se actualiza tras realizar el calculo sobre uno de sus elementos, por lo que acaba conteniendo las soluciones de esta iteración y de la anterior.

Jacobi

Este otro metodo, es practicamente idéntico al de Gauss-Seidel, con la peculiaridad de que solo usa los valores de la iteración actual sin importar que elemento del vector x se este actualizando. Por ello, la fromula para actualizar los elementos de x queda simplificada a :

$$x_j^i = \frac{b_j - \sum_{\substack{k=1 \\ k \neq j}}^{n} a_{\mathrm{jk}} * x_k^{i-1}}{a_{\mathrm{jj}}} \tag{3}$$

De esta forma el algoritmo que implementa este metodo es practicamente una copia del de Gauss-Seidel pero en vez de usar el mismo vector que actualizamos para realizar los calculos de los elementos de x para esta iteración, usaremos un vector copia del de la iteración anterior pero sin actualizarlo, quedando con el siguiente aspecto:

```
def jacobi(A,b,x_0,e):
    x_k1 = np.copy(x_0) + 2*e #Copiamos el vector x_0 y sumamos a todos sus
elementos
    #2e para que el bucle se ejecute al menos una vez (python no tiene repeat
until)
    x_k = np.copy(x_0) #Se copia un vector con los mismos elementos
    k=1 #Variable que alamcena el numero de la iteración

    while (not convergencia(e, x_k1, x_k)):#Mientras no haya convergencia
        x_k1 = np.copy(x_k)
        for i in range(len(x_0)): # recorrido de las filas
            aux = b[i]
            for j in range(len(x_0)): # recorrido en columnas
                if (j != i):
                    aux -= A[i, j] * x_k1[j]
            x_k[i] = aux/A[i,i] #Dividimos por el elemento de la diagonal
        print(k,":",x_k)
        k+=1
    return x_k
```

2.2 Problemas planteados y su resolución

Para comprobar el fucnionamiento de ambos algoritmos, dos problemas fueron planteados. En esta sección enunciaremos dichos problemas y veremos un par de pasos iteratiivos asi como las soluciones que ambos metodos arrojaron.

$$1)\quad \begin{aligned} 3x_1 - 0.1x_2 - 0.2x_3 &= 7,85 \\ 0.1x_1 + 7x_2 - 0.3x_3 &= -19.3 \\ 0.3x_1 - 0.2x_2 + 10x_3 &= 71.4 \end{aligned}$$

En este sistema se pedia obtener una solución partiendo el vector inicial 0 y una toleracia variable, en este caso obtamos por una tolerancia $\varepsilon = 0.1$ y sabiendo que los resultados eran $x_1 = 3; x_2 = -2.5;$ $x_3 = 7$. Obtuvimos los resultados recogidos en la siguiente tabla

iteración	x^1	x^2	Final
Gauss-Seidel	[2.6166 -2.7945 7.0056]	[2.9905 -2.4996 7.0002]	$x^3 =$ [3.00003 -2.4999 6.999]
Jacobi	[2.6166 -2.7571 7.14]	[3.0007 -2.4885 7.0063]	$x^3 =$ [3.0008 -2.4997 7.0002]

Tabla 2.

$$
\begin{aligned}
& 5x_1 + 2x_2 - x_3 + x_4 = 12 \\
2) \quad & x_1 + 7x_2 + 3x_3 - x_4 = 2 \\
& -x_1 + 4x_2 + 9x_3 + 2x_4 = 1 \\
& x_1 - x_2 + x_3 + 4x_4 = 3
\end{aligned}
$$

En este sistema se pedia obtener una solución partiendo el vector inicial 0 y una toleracia variable, en este caso obtamos por una tolerancia $\varepsilon = 0.1$ y sabiendo que los resultados eran $x_1 = 2.7273$; $x_2 = -0.4040; x_3 = 0.6364: x_4 = -0.1919$. Obtuvimos los resultados recogidos en la siguiente tabla

iteración	x^1	Final
Gauss-Seidel	[2.4 -0.057 0.4031 0.0349]	$x^4 =$ [2.6533 -0.3441 0.5841 -0.1453]
Jacobi	[2.4 0.2857 0.1111 0.75]	$x^5 =$ [2.5183 -0.2006 0.4362 0.0148]

Tabla 3.

Con estos dos ejemplos ya se puede observar que el metodo de Gauss-Seidel converge más rapido que el de Jacobi. Además dicha convergencia se da en numeros más cercanos a la solución real. Para hacerlo más evidente vamos a ver 4 tablas más con los resultados arrojados al imponer un $\varepsilon_1 = 0.01, \varepsilon_2 = 0.001$

Ejercicio 1(ε_1)

	Final
Gauss-Seidel	$x^3 =$ [3.00003 -2.4999 6.99999]
Jacobi	$x^4 =$ [3.00002 -2.500002 6.99998]

Tabla 4.

Ejercicio 1(ε_2)

	Final
Gauss-Seidel	$x^4 =$ [3.0000003 -2.50000004 6.99999]
Jacobi	$x^4 =$ [3.00002 -2.500002 6.99998]

Tabla 5.

Ejercicio 2(ε_1)

iteración	Final
Gauss-Seidel	$x^8 =$ [2.7182 -0.3967 0.6300 -0.1862]
Jacobi	$x^{12} =$ [2.6974 -0.3765 0.6073 -0.1633]

Tabla 6.

Ejercicio 2(ε_2)

	Final
Gauss-Seidel	$x^{12} = [\ 2.7261\ -0.4031\ \ 0.6355\ -0.1912]$
Jacobi	$x^{21} = [\ 2.7248\ -0.4018\ \ 0.6340\ -0.1896]$

Tabla 7.

Con lo que verificamos la tendencia que habiamos observado.