UNIVERSIDAD ESTATAL
PENÍNSULA DE SANTA ELENA

"Simulación numérica de yacimientos Programa de Maestría en Petróleo Cohorte 3"

Asignatura: Simulación numérica de yacimientos
Nombre del Profesor: Ing. José Villegas
Autores: Alex Zambrano / Alisson Giler / Kevin Soto / Fabian Rivera

INFORME TALLER # 1

1. Dada la siguiente función $f(x) = 10e^{\frac{x}{2}} \cos(2x)$ en el intervalo [0,10]. Graficar la función y encontrar las raíces.

Este primer ejercicio se realizó mediante el programa Wolfram matemática mediante el método de bisección.

Para este método se siguen los siguientes pasos:

- Inicializamos con los valores de a y b para el intervalo de búsqueda.
- Se calcula el punto medio c del intervalo.
- Definimos el número de iteraciones n que se usarán en el bucle.
- Mediante el bucle For realizamos las n iteraciones para refinar el intervalo $[a,b]$ y acercarse a la raíz de la función.

En cada iteración:

Calcula el punto medio c del intervalo $[a,b]$

Se evalúa la función en c y ajusta el intervalo dependiendo del signo de $f(c)$:

Si $f(c)<0$ se ajusta a al valor de c.

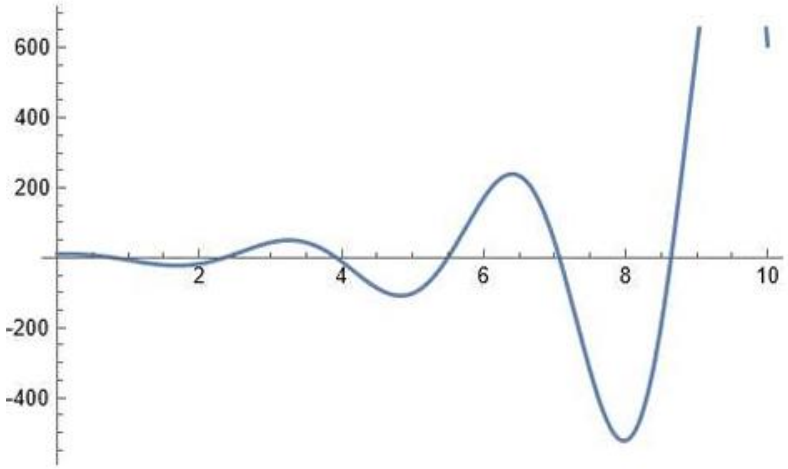
Si $f(c)\geq 0$, se ajusta b al valor de c.

- Por último, se imprime los valores de a, b, y c para cada iteración.

```
Plot[10^(x/2) Cos[2x],{x,0.0,10.0}]
f[x_]=10^(x/2) Cos[2x];
a=8.0;
b=10.0;
c=N[(b+a)/2];
n=10;
For[i=1,i<=n, i++,
c=N[(b+a)/2];
If[f[c]<0,
a=c,
b=c];
Print["iter= ",i," a= ", a, " b= ",b, " c=",c]
];
```

Gráfica de la función

Como se observa en la gráfica la función tiene varias raíces como solución dentro del intervalo [0,10], por este motivo se escogió el intervalo entre [8,10] debido a que el método considera que la función tiene una sola raíz.



Iteraciones

A continuación, se muestran las iteraciones del método para el intervalo entre [8,10].

```
iter= 1   a= 8.    b= 9.    c=9.
iter= 2   a= 8.5   b= 9.    c=8.5
iter= 3   a= 8.5   b= 8.75  c=8.75
iter= 4   a= 8.625 b= 8.75  c=8.625
iter= 5   a= 8.625 b= 8.6875  c=8.6875
iter= 6   a= 8.625 b= 8.65625  c=8.65625
iter= 7   a= 8.625 b= 8.64063  c=8.64063
iter= 8   a= 8.63281  b= 8.64063  c=8.63281
iter= 9   a= 8.63672  b= 8.64063  c=8.63672
iter= 10  a= 8.63867  b= 8.64063  c=8.63867
```

El método de bisección es una herramienta fundamental en el análisis numérico debido a su simplicidad, robustez y garantía de convergencia. Aunque puede no ser el método más rápido, su fiabilidad lo hace una opción valiosa, especialmente en situaciones donde otros métodos pueden fallar. Su uso en diversas disciplinas subraya su importancia en la resolución de problemas prácticos y teóricos.

2. Resolver el siguiente sistema de ecuaciones no lineales.

$$\begin{cases} f1(x1,x2,x3) = 3x1 - cos(x2x3) - \frac{1}{2} \\ f1(x1,x2,x3) = x1^2 - 81(x2+0.1)^2 + sin(x3) + 1.06 \\ f1(x1,x2,x3) = e^{-x1x2} + 20x3 + \frac{10\pi - 3}{3} \end{cases}$$

Para la resolución de este sistema de ecuaciones no lineales elegimos el método de Newton-Raphson y utilizamos el software de programación Python.

- Como primer paso importamos las librerías que necesitaremos

```
import numpy as np
from sympy import symbols, cos, sin, exp, Matrix, lambdify
```

- Luego se definen las variables y funciones del sistema

```
# Definir las variables
x1, x2, x3 = symbols('x1 x2 x3')

# Definir las funciones del sistema
f1 = 3*x1 - cos(x2 * x3) - 1/2
f2 = x1**2 - 81*(x2 + 0.1)**2 + sin(x3) + 1.06
f3 = exp(-x1 * x2) + 20 * x3 + (10 * 3.14 - 3) / 3
```

- Definimos el sistema de ecuaciones no lineales como una matriz
  Calculamos la matriz Jacobiana

```
# Definir el sistema de ecuaciones como una matriz
system = Matrix([f1, f2, f3])

# Calcular la matriz Jacobiana
jacobian = system.jacobian([x1, x2, x3])

# Convertir las funciones y la jacobiana a funciones numéricas
F = lambdify((x1, x2, x3), system, 'numpy')
J = lambdify((x1, x2, x3), jacobian, 'numpy')
```

- Inicializamos la solución con los siguientes valores iniciales (0.1, 0.1, -0.1).

```
# Inicializar la solución
x = np.array([0.1, 0.1, -0.1], dtype=np.float64)

# Método de Newton-Raphson
def newton_raphson(F, J, x0, tol=1e-10, maxiter=100):
    x = x0
    for _ in range(maxiter):
        # Evaluar la Jacobiana y la función en el punto actual
        J_eval = np.array(J(*x), dtype=np.float64)
        F_eval = np.array(F(*x), dtype=np.float64).flatten()
```

- Luego calculamos la inversa de la matriz Jacobiana

```
    # Calcular la inversa de la Jacobiana
    J_inv = np.linalg.inv(J_eval)

    # Calcular el cambio
    delta = np.dot(J_inv, F_eval)

    # Actualizar la solución
    x = x - delta
```

- Finalmente se verifica la convergencia para encontrar la solución de las raíces

```
        # Verificar la convergencia
        if np.linalg.norm(delta, ord=2) < tol:
            break
    return x

# Encontrar la solución
solution = newton_raphson(F, J, x)
print(solution)
```

```
[ 5.00000000e-01  1.42091776e-05 -5.23332978e-01]
```

Iteraciones

k	x1	x2	x3	‖x(k)-x(k-1)‖
0	0.1000	0.1000	-0.1000	
1	0.499870	0.019467	-0.521520	0.586567
2	0.500014	0.001589	-0.523557	0.017994
3	0.500000	0.000012	-0.523598	0.001577
4	0.500000	0.000000	-0.523599	0.000012
5	0.500000	0.000000	-0.523599	0.000000
6	0.500000	0.000000	-0.523599	0.000000

Procedimiento exitoso

Como se puede observar el método iterativo se repite hasta que la función inversa de la Jacobina sea suficientemente pequeña, indicando que hemos encontrado una solución aproximada al sistema de ecuaciones no lineales. Este método puede ser computacionalmente intensivo debido a la necesidad de calcular e invertir la matriz jacobiana en cada iteración, pero es muy poderoso para encontrar soluciones a sistemas complejos de ecuaciones no lineales.

Además, en estas iteraciones la rápida convergencia y la precisión alcanzada en pocas iteraciones reflejan la eficacia del método cuando se aplican correctamente. La salida también muestra cómo los valores se estabilizan a medida que la norma de la diferencia entre iteraciones consecutivas tiende a cero, confirmando la convergencia del método.

3. Resolver el siguiente sistema de ecuaciones lineales:

$$\begin{cases} 6x1 + 3x2 - x3 = 40 \\ -2x1 + 4x2 + 2x3 = 20 \\ x1 - 3x2 + 8x3 = 2 \end{cases}$$

Para su resolución utilizamos el método de punto fijo.

El cuál sigue los siguientes pasos:

- Primero se definen los parametros iniciales como:

Número de iteraciones definido como 20

Xn: que es el vector inicial, con cuatro componentes inicializadas a 0.
B: El vector constante con tres componentes.
M: La matriz de coeficientes de dimensión 3x3.

- Luego se establece un bucle For que itera desde i = 1 hasta i <= 20.

Dentro del cual se calcula:
Se calcula Xnp1 como la suma del vector B y el producto de la matriz M con el vector Xn.
Se imprime la iteración actual i, el vector Xn y el nuevo vector Xnp1.
Se actualiza Xn con el valor de Xnp1.

```
niters=20;

Xn={0.0,0.0,0.0};
B={40/6,5,1/4};
M={{0.0,-1/3,1/6},{1/2,0.0,-1/2},{-1/8,3/8,0.0}};

For[i=1, i<= niters, i++,
Xnp1 = B + M.Xn;

 Print["iter= ",i," Xn= ", Xn, " Xnp1= ",Xnp1];

 Xn=Xnp1;

]
```

Iteraciones

```
iter= 1 Xn= {0., 0., 0.} Xnp1= {6.66667, 5., 0.25}
iter= 2 Xn= {6.66667, 5., 0.25} Xnp1= {5.04167, 8.20833, 1.29167}
iter= 3 Xn= {5.04167, 8.20833, 1.29167} Xnp1= {4.14583, 6.875, 2.69792}
iter= 4 Xn= {4.14583, 6.875, 2.69792} Xnp1= {4.82465, 5.72396, 2.3099}
iter= 5 Xn= {4.82465, 5.72396, 2.3099} Xnp1= {5.14366, 6.25738, 1.7934}
iter= 6 Xn= {5.14366, 6.25738, 1.7934} Xnp1= {4.87977, 6.67513, 1.95356}
iter= 7 Xn= {4.87977, 6.67513, 1.95356} Xnp1= {4.76722, 6.46311, 2.1432}
iter= 8 Xn= {4.76722, 6.46311, 2.1432} Xnp1= {4.8695, 6.31201, 2.07776}
iter= 9 Xn= {4.8695, 6.31201, 2.07776} Xnp1= {4.90896, 6.39587, 2.00832}
iter= 10 Xn= {4.90896, 6.39587, 2.00832} Xnp1= {4.86943, 6.45032, 2.03483}
iter= 11 Xn= {4.86943, 6.45032, 2.03483} Xnp1= {4.8557, 6.4173, 2.06019}
iter= 12 Xn= {4.8557, 6.4173, 2.06019} Xnp1= {4.87093, 6.39775, 2.04953}
iter= 13 Xn= {4.87093, 6.39775, 2.04953} Xnp1= {4.87567, 6.4107, 2.04029}
iter= 14 Xn= {4.87567, 6.4107, 2.04029} Xnp1= {4.86981, 6.41769, 2.04456}
iter= 15 Xn= {4.86981, 6.41769, 2.04456} Xnp1= {4.8682, 6.41263, 2.04791}
iter= 16 Xn= {4.8682, 6.41263, 2.04791} Xnp1= {4.87044, 6.41014, 2.04621}
iter= 17 Xn= {4.87044, 6.41014, 2.04621} Xnp1= {4.87099, 6.41211, 2.045}
iter= 18 Xn= {4.87099, 6.41211, 2.045} Xnp1= {4.87013, 6.41299, 2.04567}
iter= 19 Xn= {4.87013, 6.41299, 2.04567} Xnp1= {4.86995, 6.41223, 2.04611}
iter= 20 Xn= {4.86995, 6.41223, 2.04611} Xnp1= {4.87027, 6.41192, 2.04584}
```

El método del punto fijo aplicado en este caso muestra una convergencia adecuada hacia una solución estable para el sistema de ecuaciones no lineales. La tabla de iteraciones indica que el procedimiento se estabiliza y converge a valores precisos en un número razonable de iteraciones. Este comportamiento es indicativo de que las condiciones necesarias para la convergencia del método del punto fijo están siendo satisfechas en este sistema particular. Su aplicación se extiende a numerosos campos, y su extensión a sistemas multivariables es fundamental en la solución de problemas complejos en ciencia e ingeniería. Al entender las condiciones de convergencia y la implementación del método, se puede aprovechar su simplicidad y eficiencia para resolver una amplia gama de problemas.