26/08/2021

Parcial #1 Análisis numérico 2021-30

Natalia Gaona Salamanca

Punto 1 B

Condiciones que debe cumplir: Como vamos a trabajar por el método de secante, en este método se requiere conocer el valor de la primera derivada de la función entrante en un punto. En muchas ocasiones se dificulta encontrar la derivada de esa función; cuando esto sucede se debe recurrir al método de la secante.

Para el método de la secante es necesario conocer dos puntos iniciales los cuales van a estar dados por Xi-1 Xi, con el fin de encontrar el Xi+1, conduciéndonos a la raíz más exacta de la función.

Como se menciona anteriormente es necesario conocer dos puntos, teniendo en cuenta que estos no deben ser afectados por asíntotas, puntos de inflexión, mínimos o máximos locales y pendientes que se aproximen a cero.

Además de esto debemos saber las condiciones para K porque en el llegado caso que K sea una raíz con índice par su convergencia solo sería los números positivos.

Código:

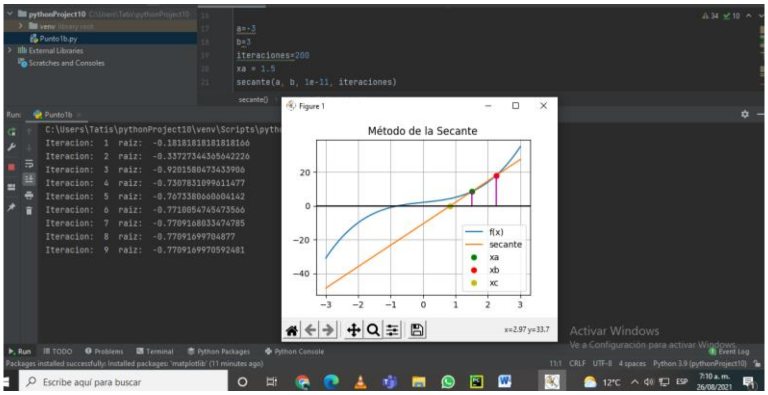
Punto1b.py

```python
import numpy as np
from matplotlib import pyplot as plt
#Realizado por: Natalia Gaona Salamanca
def f(x):
    return x**3+2*(x)+2
def secante(a, b, tol, max_iter):
    iter = 0
    c = (a + b) / 2
    i = 0
    while (abs(f(c)) > tol and iter < max_iter):
        c = b - (f(b) * (b - a) / (f(b) - f(a)))
        a = b
        b = c
        i = i + 1
        print("Iteracion: ", i, " raiz: ", c)

a=-3
b=3
iteraciones=200
xa = 1.5
secante(a, b, 1e-11, iteraciones)

# See PyCharm help at https://www.jetbrains.com/help/pycharm/
xi = np.linspace(a,b,iteraciones+1)
```

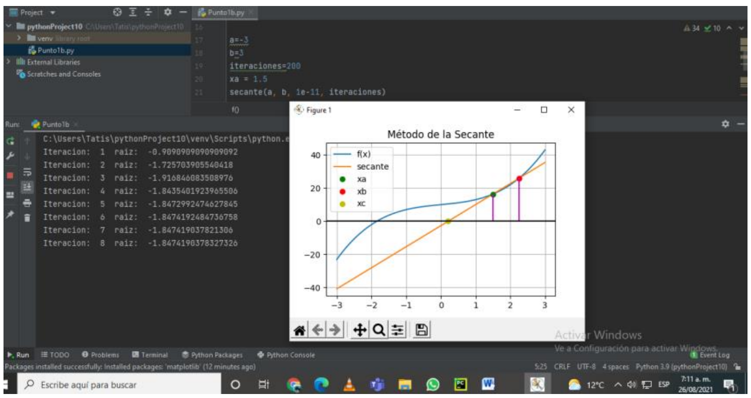
```python
fi = f(xi)
c = (b-xa)/2
pendiente = (f(xa+c)-f(xa))/(xa+c-xa)
b0 = f(xa) - pendiente*xa
tangentei = pendiente*xi+b0

fxa = f(xa)
xb = xa + c
fxb = f(xb)

plt.plot(xi,fi, label='f(x)')
plt.plot(xi,tangentei, label='secante')
plt.plot(xa,f(xa),'go', label='xa')
plt.plot(xa+c,f(xa+c),'ro', label='xb')
plt.plot((-b0/pendiente),0,'yo', label='xc')

plt.plot([xa,xa],[0,fxa],'m')
plt.plot([xb,xb],[0,fxb],'m')

plt.axhline(0, color='k')
plt.title('Método de la Secante')
plt.legend()
plt.grid()
plt.show()
```

Pruebas:

Primera prueba: Cuando k es 2:



Segunda prueba: Cuando k es 10:



Tercera prueba: Cuando es$\sqrt{k+2}$: donde K puede ser de -2 a infinitos positivos, para el caso de la prueba reemplazaremos k como -1/3

```
import numpy as np
from matplotlib import pyplot as plt
#Realizado por: Natalia Gaona Salamanca
def f(x):
    return x**3+2*(x)+np.sqrt((-1/3)+2)
def secante(a, b, tol, max_iter):
    iter = 0
    c = (a + b) / 2
    i = 0
    while (abs(f(c)) > tol and iter < max_iter):
        c = b - (f(b) * (b - a) / (f(b) - f(a)))
```

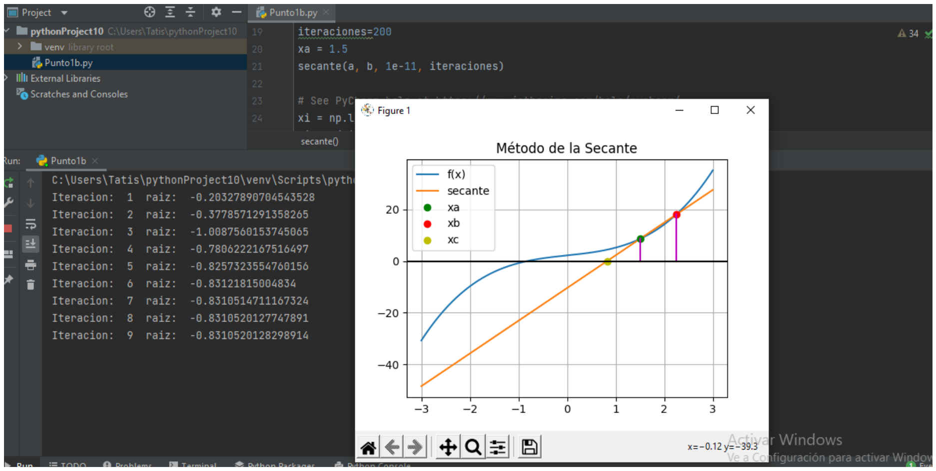
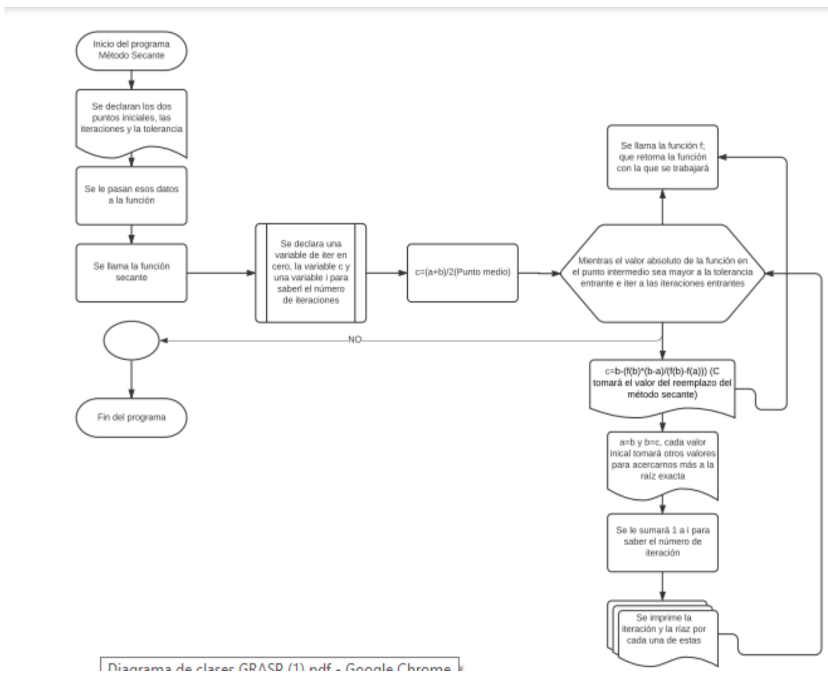


Diagrama de flujo:



Gráfica: Este método mantiene la convergencia superlineal más rápida y efectiva, pero se debe recalcar que esta raíz de la primera aproximación no es segura, que no es suficientemente cercana, ni cuando es una raíz múltiple, haciendo que este método pueda divergir.

Conclusión: El método de la secante es muy efectivo cuando la derivada de una función es muy compleja, es cuando utilizamos este método; es importante conocer cuáles son las condiciones para aplicarlo, a diferencia de otros métodos no itera tanto.

Punto 3c:

Condiciones que debe cumplir: Existencia: si g (la función) es continua en [a,b] y g(x) Î [a,b] para todo "x" Î [a,b], entonces la función tiene punto fijo.

Unicidad: si además, | g'(x) | < 1 para todo "x" Î [a,b], entonces el punto fijo es único.

Convergencia: si además, x0 Î [a,b], entonces la sucesión xn = g(xn-1) converge a ese punto fijo.

Código:

```python
import numpy as np

#Realizado por: Natalia Gaona Salamanca

gx= lambda x: x**3-2*x-5

a=-5
b=1
tolerancia=1e-5
iteramax=15
i=1 #procedimiento
b=gx(a)
tramo=abs(b-a)
while not(tramo<=tolerancia or i>=iteramax):
    a=b
    b=gx(a)
    tramo = abs(b - a)
respuesta=b
#validar convergencia
if(i>=iteramax):
        respuesta=np.nan#no fue un numero aceptable
print(respuesta)
print(i)
print(tramo)
```

Cálculo:

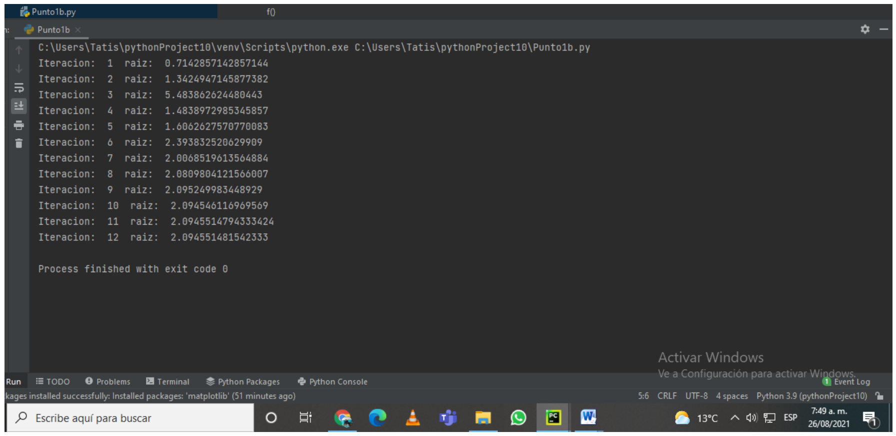
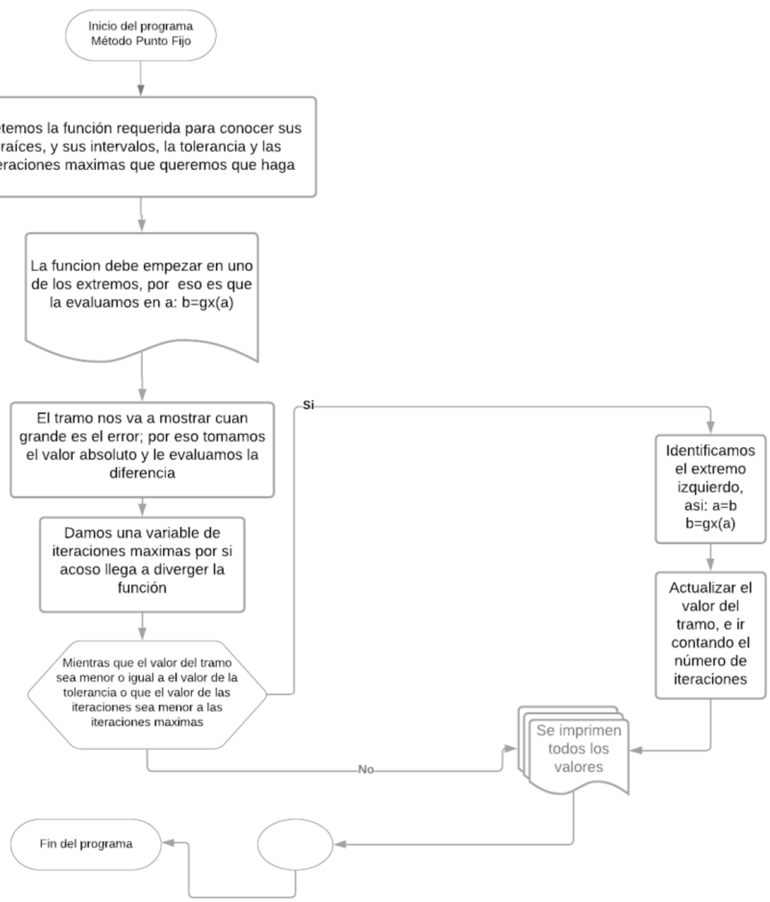


Diagrama de flujo:



Conclusión: El método de punto fijo converge con más rapidez y cuando lo hace, lo hace con más precisión, no necesita de un intervalo para funcionar sino de únicamente un punto perteneciente al intervalo donde esté la raíz. Lo malo de este método es que no garantiza una convergencia, a veces es muy compleja de encontrar la segunda función ya que hay infinidad para escoger una g(x) y no hay una regla correcta y concreta para escogerla.

Punto 5

Condiciones: La fórmula de iteración del método de Newton resulta ser la fórmula del método iterativo de punto fijo para la función g(x) = x - f(x)/f '(x).

Se pueden aplicar las condiciones suficientes del método de punto fijo a esta función para encontrar un intervalo donde obtener el punto inicial, de manera de asegurar la convergencia de la fórmula. Dichas condiciones son:

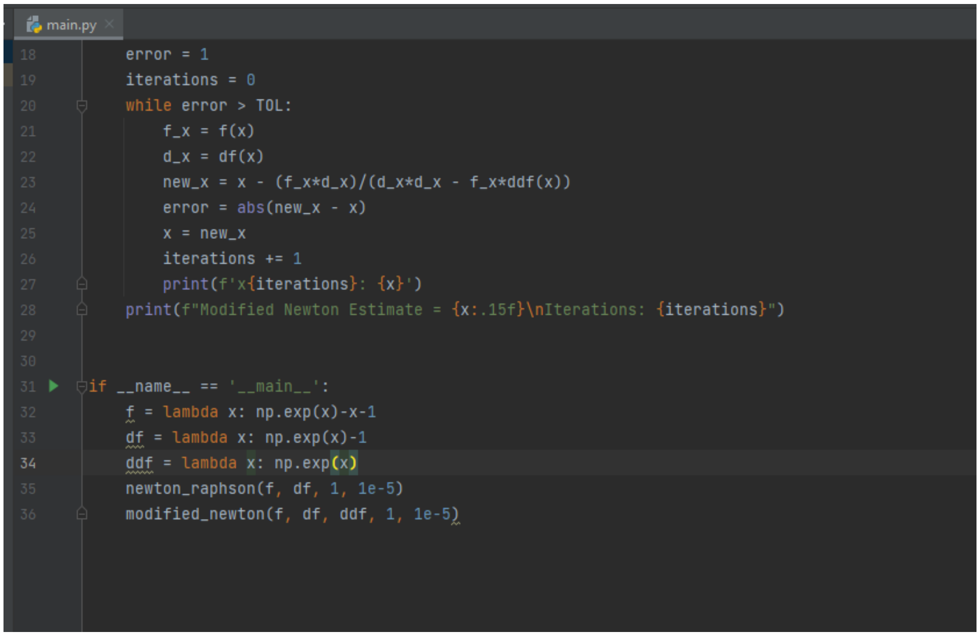
g(x) continua en [a, b], tal que " x Î [a, b], g(x) Î [a, b]

$ 0 < k < 1 / |g'(x)| £ k " x Î [a, b]

Bajo estas condiciones, cualquier punto inicial dentro del intervalo hallado sirve para garantizar la convergencia de la fórmula de Newton.

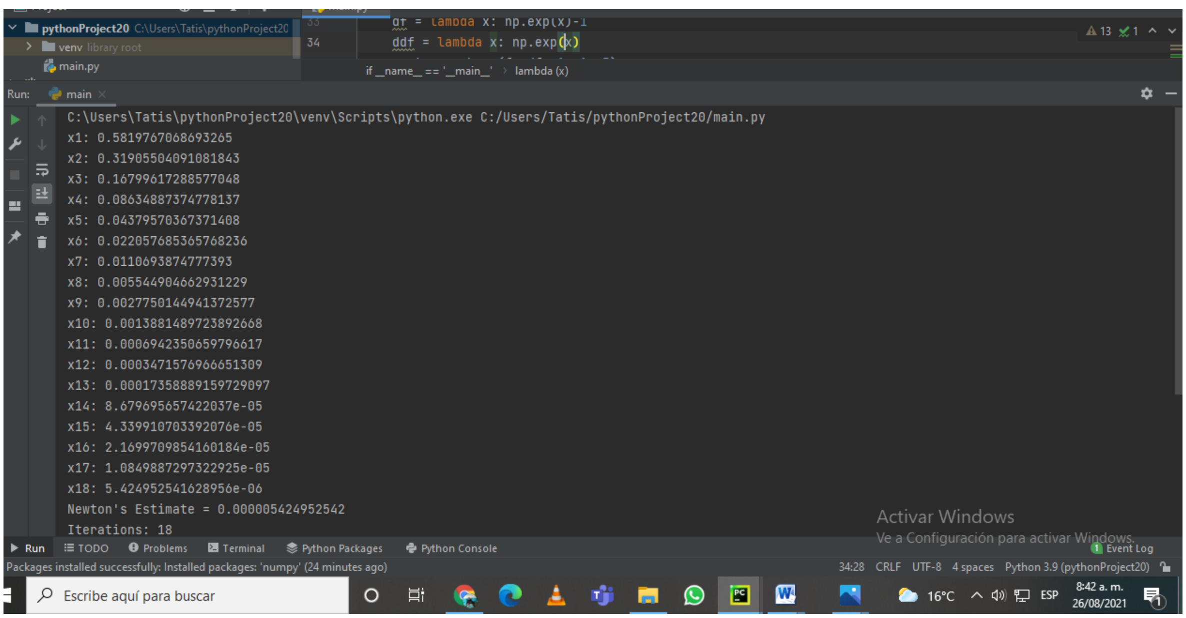
Código:

```
import math
import numpy as np
e = math.e
pi = math.pi

def newton_raphson(f, df, x, TOL):
    error = 1
    iterations = 0
    while error > TOL:
        new_x = x - f(x)/df(x)
        error = abs(new_x - x)
        x = new_x
        iterations += 1
        print(f'x{iterations}: {x}')
    print(f"Newton's Estimate = {x:.15f}\nIterations: {iterations}")

def modified_newton(f, df, ddf, x, TOL):
    error = 1
    iterations = 0
    while error > TOL:
        f_x = f(x)
        d_x = df(x)
        new_x = x - (f_x*d_x)/(d_x*d_x - f_x*ddf(x))
        error = abs(new_x - x)
```



```python
    error = 1
    iterations = 0
    while error > TOL:
        f_x = f(x)
        d_x = df(x)
        new_x = x - (f_x*d_x)/(d_x*d_x - f_x*ddf(x))
        error = abs(new_x - x)
        x = new_x
        iterations += 1
        print(f'x{iterations}: {x}')
    print(f"Modified Newton Estimate = {x:.15f}\nIterations: {iterations}")


if __name__ == '__main__':
    f = lambda x: np.exp(x)-x-1
    df = lambda x: np.exp(x)-1
    ddf = lambda x: np.exp(x)
    newton_raphson(f, df, 1, 1e-5)
    modified_newton(f, df, ddf, 1, 1e-5)
```

Calculo:



```
C:\Users\Tatis\pythonProject20\venv\Scripts\python.exe C:/Users/Tatis/pythonProject20/main.py
x1: 0.5819767068693265
x2: 0.31905504091081843
x3: 0.16799617288577048
x4: 0.08634887374778137
x5: 0.04379570367371408
x6: 0.022057685365768236
x7: 0.0110693874777393
x8: 0.005544904662931229
x9: 0.002775014494137257 7
x10: 0.0013881489723892668
x11: 0.0006942350659796617
x12: 0.0003471576966651309
x13: 0.00017358889159729097
x14: 8.679695657422037e-05
x15: 4.339910703392076e-05
x16: 2.169970985416018 4e-05
x17: 1.0849887297322925e-05
x18: 5.424952541628956e-06
Newton's Estimate = 0.000005424952542
Iterations: 18
```

Acá ya se puede apreciar la modificación que se le hace:

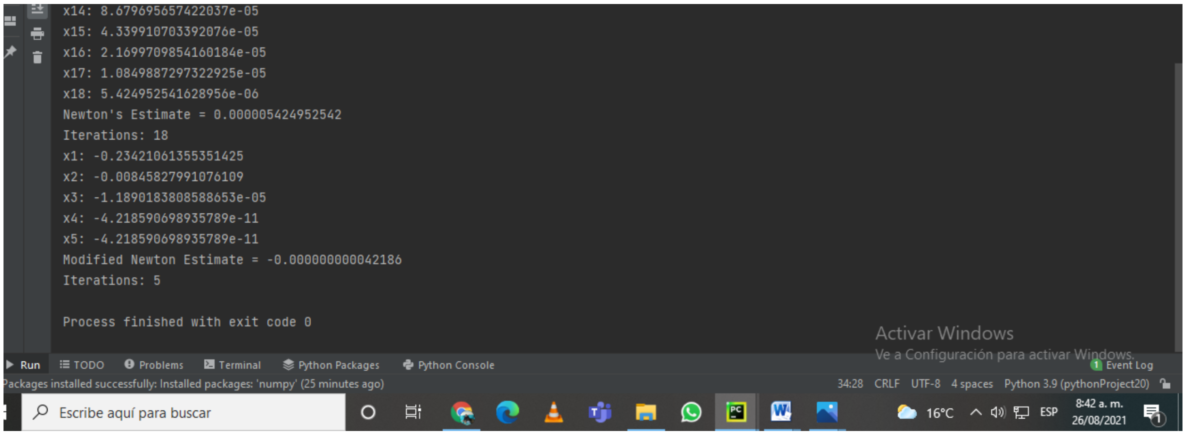
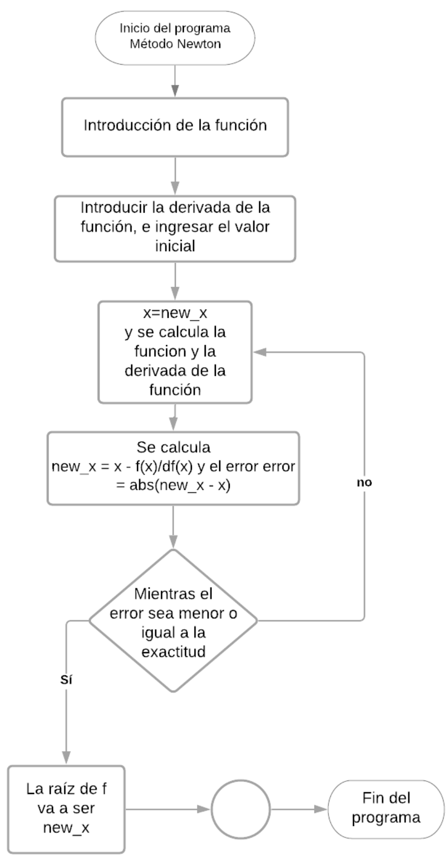


Diagrama de Flujo:



Conclusión: El método Newton-Raphson depende del valor inicial que ingresas. Es importante los máximos y mínimos, los puntos de inflexión y las asíntotas de la función, porque es cuando el método falla. Esto se debe a que la pendiente de la tangente puede aproximarse en gran magnitud a cero y el método puede fallar o presentar lenta convergencia. Es un método ideal para solucionar ecuaciones no líneales, que convergen rápidamente y los resultados obtenidos rápidamente.