Lab 02. Searching with interpolation methods

Mohamed Stifi

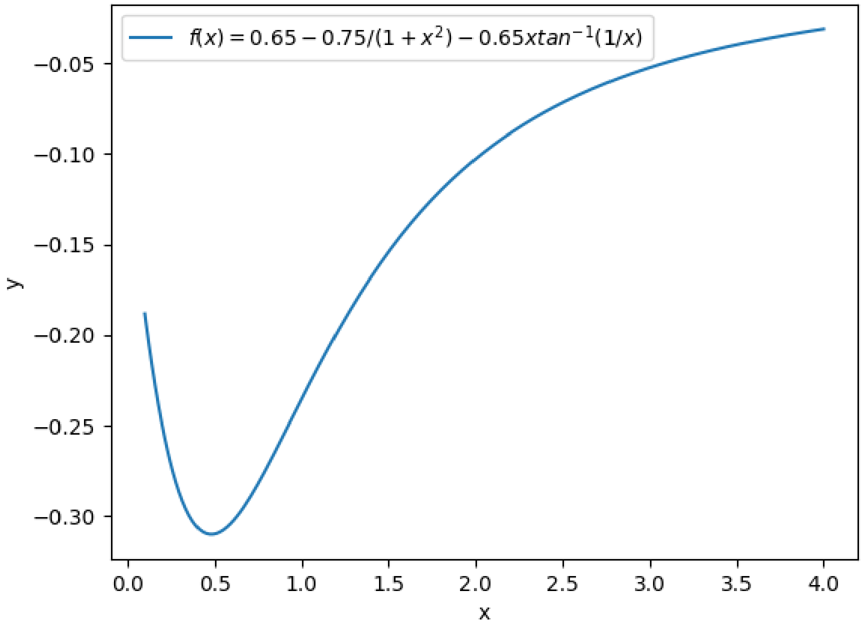
2IA

1 Code

1.1 global variable and library

```python
import numpy as np
from numdifftools import Derivative as Df
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from time import time

#--------------------------| global variable |--------------
f = lambda x : 0.65 - 0.75/(1 + x**2) - 0.65*x*np.arctan(1/x)
df = Df(f)
d2f = Df(df)
x0 = 0.2
a = 0.2
b = 3
delta = 1e-3
X = np.linspace(0.1, 4, 200)
Y1 = f(X)
Y2 = df(X)
#--------------------------| plot the function |------------
plt.plot(X, Y1, label = "$f(x) = 0.65 - 0.75/(1+x^2) -0.65xtan^{-1}(1/x)$")
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

1.1.1 Screenshots



1.2 A python implementation of the optimization methods

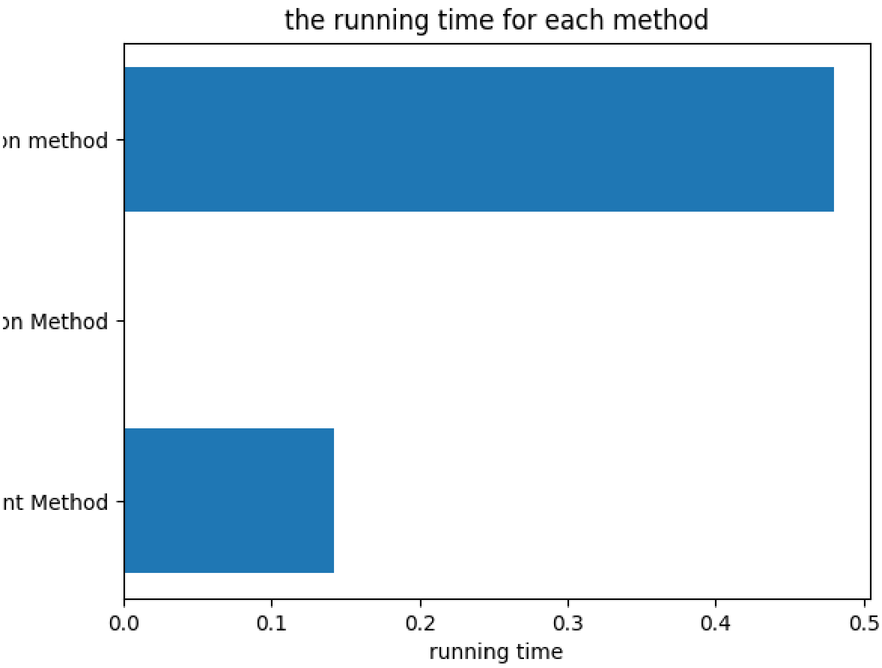
```python

#--------------------------| Newton-Rapson method |---------
def Newton_Rapson_method( df , d2f, x0,num_iter_max= 1000, eps = 1e-9) :
    """
    parametrs:
        df : the first derivative of f
        d2f : the second derivative of f
        x0 : The initial guess for x optimal
        num_iter_max : Maximum number of iterations (default set to 1000)
        eps : A small value (default set to 1e-9)
    return :
        x0 : the final x  optimal
        iter_ : the final iterations
    """
    f1  = df(x0)
    iter_ = 0
    while np.abs(f1) > eps and num_iter_max > iter_ :
        f2 = d2f(x0)
        if f2 != 0 :
            x0 -= f1/f2
            f1  = df(x0)
            iter_ += 1
        else :
            x0 -= f1/eps
            f1  = df(x0)
            iter_ += 1

    return x0 , iter_

#---------------------------|Quasi Newton Method |----------
```

```python
def Quasi_Newton_Method(f, x0, delta,num_iter_max = 1000, eps = 1e-8):
    """
    parametrs:
        f : The target function f
        x0 : The initial guess for x optimal
        num_iter_max : Maximum number of iterations (default set to 1000)
        delta : A small value
        eps : A small value (default set to 1e-9)
    return :
        x0 : the final x  optimal
        iter_ : the final iterations
    """
    f1 = f(x0 + delta)
    f2 = f(x0 - delta)
    delta_2 = 2*delta
    df = (f1 - f2)/delta_2
    iter_ = 0
    while np.abs(df) > eps and num_iter_max > iter_ :
        x0 -= (delta*(f1 - f2))/(2*(f1 - 2*f(x0) + f2))
        f1 = f(x0 + delta)
        f2 = f(x0 - delta)
        df = (f1 - f2)/delta_2
        iter_ += 1
    return x0 , iter_

#---------------------------| Secant Method |----------------
def Secant_Method(df, a, b, num_iter_max = 1000, eps = 1e-8):
    """
    parametrs:
        df
        a : The initial value of the lower bound
        b : The initial value of the upper bound
        num_iter_max : Maximum number of iterations (default set to 1000)
        eps : A small value (default set to 1e-9)
    return :
        x0 : the final x  optimal
        iter_ : the final iterations
    """
    dfx = df(a)
    x = a
    iter_ = 0
    while np.abs(dfx) > eps and num_iter_max > iter_ :
        df1 = df(a)
        df2 = df(b)
        x = a - (df1*(b- a))/(df2 - df1)
        dfx = df(x)
        if dfx >= 0 :
            b = x
            df2 = dfx
        else :
            a = x
            df1 = dfx
        iter_ += 1
    return x , iter_
```

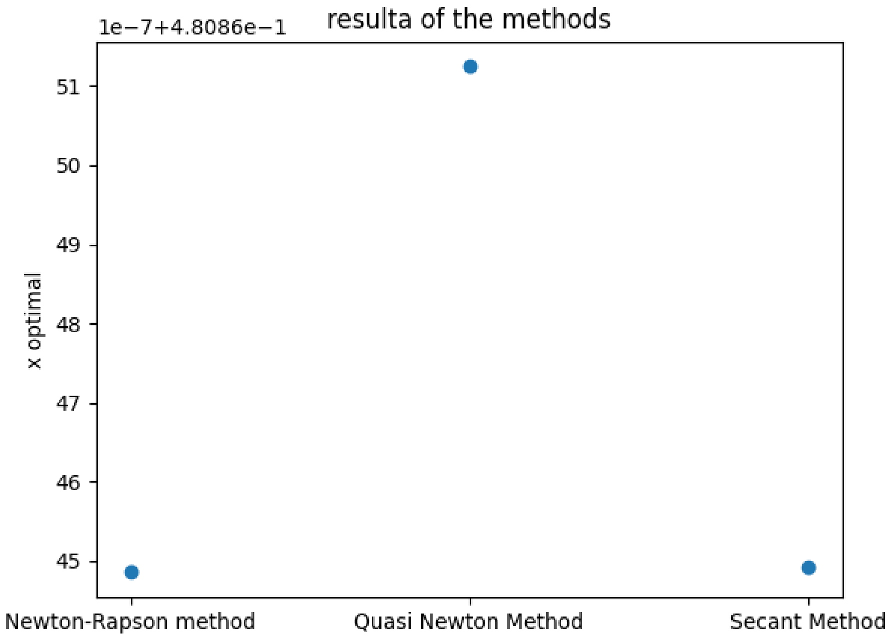
1.3 Compute the running time for each method

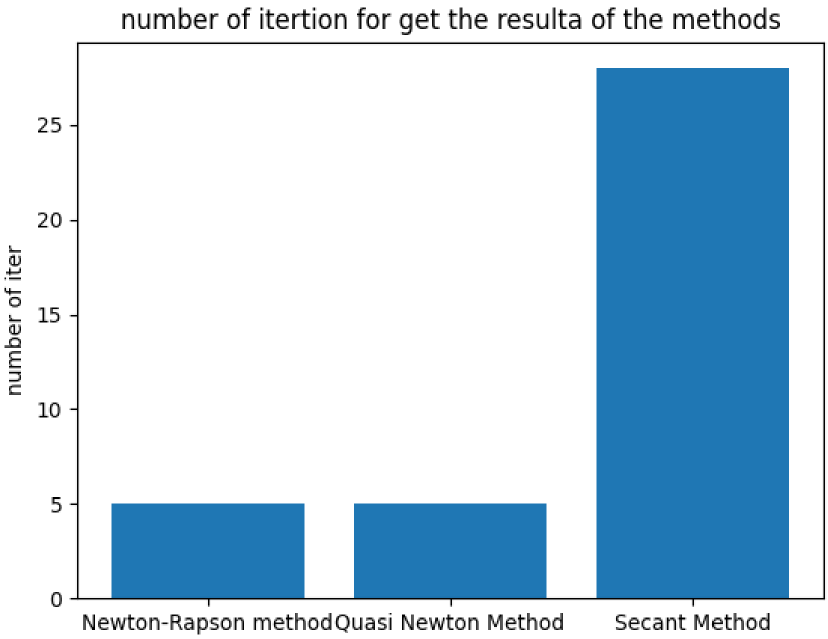
```python
#----------------------------| time complixity |------------
methods = ['Newton-Rapson method', 'Quasi Newton Method', 'Secant Method']
times = []
x_optm = []
optm_iter = []
t1 = time()
N_x , N_iter_ = Newton_Rapson_method( df, d2f, x0,num_iter_max= 1000, eps = 1e-9)
t2 = time()
times.append(t2-t1)
x_optm.append(N_x)
optm_iter.append(N_iter_)
# ----------------------------------------------------
t1 = time()
Q_x , Q_iter_ = Quasi_Newton_Method(f, x0, delta,num_iter_max = 1000, eps = 1e-8)
t2 = time()
times.append(t2-t1)
x_optm.append(Q_x)
optm_iter.append(Q_iter_)
# ----------------------------------------------------
t1 = time()
S_x , S_iter_ = Secant_Method(df, a, b, num_iter_max = 1000, eps = 1e-8)
t2 = time()
times.append(t2-t1)
x_optm.append(S_x)
optm_iter.append(S_iter_)

# ---------------------------------------------------------------
print('times : ', times)
fig, ax = plt.subplots()
y = np.arange(len(times))
ax.barh(y ,times, align='center')
ax.set_yticks(y, labels=methods)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_xlabel('running time')
ax.set_title('the running time for each method')

plt.show()

#----------------------------------------------------------
print('x optimal : ',x_optm)
plt.scatter(methods,x_optm)
plt.ylabel('x optimal')
plt.title('resulta of the methods')
plt.show()
## ---------------------------------------------------------
print('number of iterition : ',optm_iter)
plt.bar(methods,optm_iter)
plt.ylabel('number of iter')
plt.title('number of itertion for get the resulta of the methods')
plt.show()
```

1.3.1 Screenshots




```
times :  [0.48031139373779297, 0.0, 0.14236807823181152]
x optimal :  [0.4808644852929133, 0.48086512483253496, 0.48086449227165107]
number of iterition :  [5, 5, 28]
```


resulta of the methods
1e−7+4.8086e−1
x optimal
51
50
49
48
47
46
45
Newton-Rapson method
Quasi Newton Method
Secant Method


number of itertion for get the resulta of the methods
number of iter
25
20
15
10
5
0
Newton-Rapson method
Quasi Newton Method
Secant Method

1.4 Newton-Rapson method animation

```
# Create the figure and subplots
fig, axs = plt.subplots(1, 2)
# Set the x-axis and y-axis labels for each subplot.
axs[0].set_title("f")
axs[1].set_title("f'")
axs[0].set_xlabel('X')
axs[0].set_ylabel('f')
axs[1].set_xlabel('X')
axs[1].set_ylabel("f'")
# Initialize empty scatter plots for animation
axs[0].plot(X,Y1, c='b')
axs[1].plot(X,Y2, c='b')
sc1 = axs[0].scatter([], [], c='y')
sc2 = axs[1].scatter([], [], c='y')

# Function
def Newton_Rapson_method(f, df, d2f, x0, num_iter_max=1000, eps=1e-9):
    xv, y1, y2 = [x0], [f(x0)], [df(x0)]
    iter_ = 0
    while abs(y2[-1]) > eps and num_iter_max > iter_:
        f2 = d2f(xv[-1])
        if f2 != 0:
            xv.append(xv[-1] - y2[-1] / f2)
            y1.append(f(xv[-1]))
            y2.append(df(xv[-1]))
            iter_ += 1
        else:
            xv.append(xv[-1] - y2[-1] / eps)
            y1.append(f(xv[-1]))
            y2.append(df(xv[-1]))
            iter_ += 1
    return xv, y1, y2

# get the values
xv, y1, y2 = Newton_Rapson_method(f, df, d2f, x0, num_iter_max=1000,
    ↪ eps=1e-9)

# Animation update function
def update(frame):
    x1, y1_, x2, y2_ = xv[:frame], y1[:frame], xv[:frame], y2[:frame]
    sc1.set_offsets(np.c_[x1, y1_])
    sc2.set_offsets(np.c_[x2, y2_])
    return sc1, sc2

# Create the animation
ani = FuncAnimation(fig, update, frames=len(xv), blit=True, interval
    ↪ =300)

# Display the plot
plt.show()
```

1.4.1 Screenshots

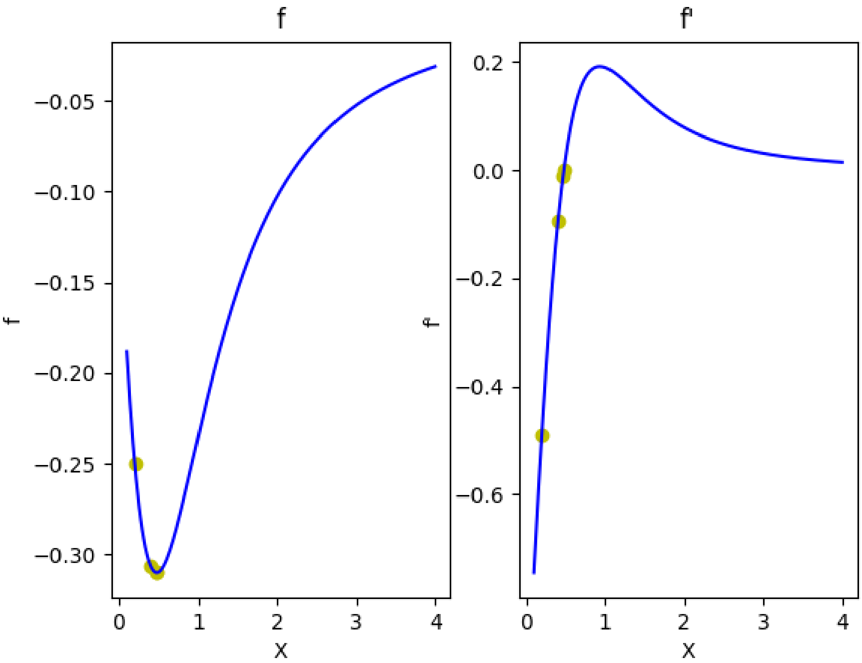


Figure 1: les valeurs de f et f' à chaque étape de méthode Newton-Rapson

1.5 Quasi Newton Method animation

```

# Create the figure and subplots
fig, axs = plt.subplots(1, 2)
# Set the x-axis and y-axis labels for each subplot.
axs[0].set_title("f")
axs[1].set_title("f'")
axs[0].set_xlabel('X')
axs[0].set_ylabel('f')
axs[1].set_xlabel('X')
axs[1].set_ylabel("f'")
# Initialize empty scatter plots for animation
axs[0].plot(X,Y1, c='b')
axs[1].plot(X,Y2, c='b')
sc1 = axs[0].scatter([], [], c='y')
sc2 = axs[1].scatter([], [], c='y')

# Function

def Quasi_Newton_Method(f, x0, delta,num_iter_max = 1000, eps = 1e-8):
    f1 = f(x0 + delta)
    f2 = f(x0 - delta)
    delta_2 = 2*delta
    df = (f1 - f2)/delta_2
    xv, y, y1 = [x0],[f(x0)],[df]
    iter_ = 0
    while np.abs(df) > eps and num_iter_max > iter_ :
        x0 -= (delta*(f1 - f2))/(2*(f1 - 2*f(x0) + f2))
        f1 = f(x0 + delta)
        f2 = f(x0 - delta)
        df = (f1 - f2)/delta_2
        xv.append(x0)
        y.append(f(x0))
        y1.append(f1)
        iter_ += 1
    return xv, y, y1

# get the values
xv, y1, y2 = Quasi_Newton_Method(f, x0, delta,num_iter_max = 1000, eps
    ↪ = 1e-8)

# Animation update function
def update(frame):
    x1, y1_, x2, y2_ = xv[:frame], y1[:frame], xv[:frame], y2[:frame]
    sc1.set_offsets(np.c_[x1, y1_])
    sc2.set_offsets(np.c_[x2, y2_])
    return sc1, sc2

# Create the animation
ani = FuncAnimation(fig, update, frames=len(xv), blit=True, interval
    ↪ =300)

# Display the plot
plt.show()
```

1.5.1 Screenshots

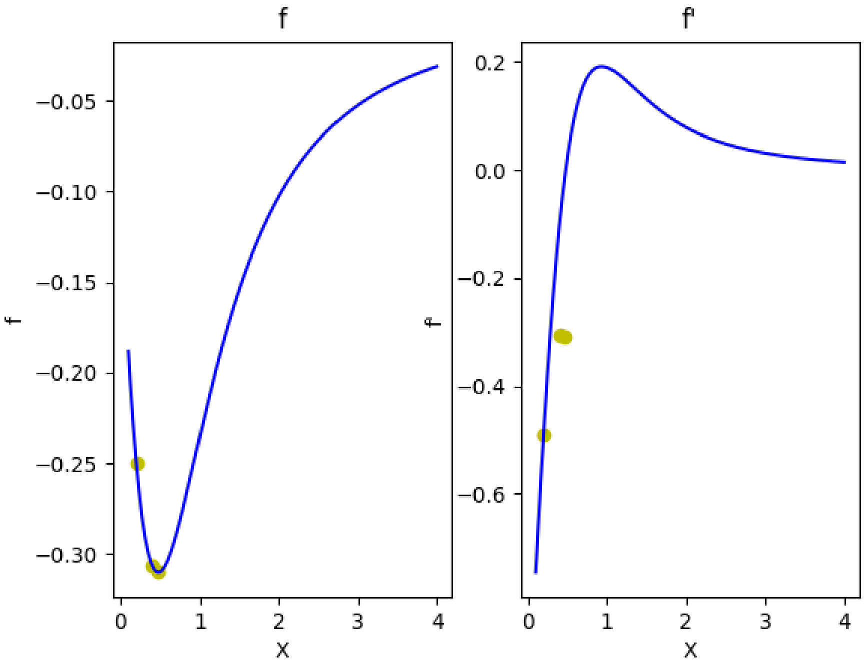


Figure 2: les valeurs de f et f' à chaque étape de Quasi méthode Newto

1.6 Secant Method Animation

```
# Create the figure and subplots
fig, axs = plt.subplots(1, 2)
# Set the x-axis and y-axis labels for each subplot.
axs[0].set_title("f")
axs[1].set_title("f'")
axs[0].set_xlabel('X')
axs[0].set_ylabel('f')
axs[1].set_xlabel('X')
axs[1].set_ylabel("f'")
# Initialize empty scatter plots for animation
axs[0].plot(X,Y1, c='b')
axs[1].plot(X,Y2, c='b')
sc1 = axs[0].scatter([], [], c='y')
sc2 = axs[1].scatter([], [], c='y')

# Function

def Secant_Method(f, df, a, b, num_iter_max = 1000, eps = 1e-8):
    dfx = df(a)
    x = a
    xv, y, y1 = [x],[f(x)],[dfx]
    iter_ = 0
    while np.abs(dfx) > eps and num_iter_max > iter_ :
        df1 = df(a)
        df2 = df(b)
        x = a - (df1*(b- a))/(df2 - df1)
        dfx = df(x)
        xv.append(x)
        y.append(f(x))
        y1.append(dfx)
        if dfx >= 0 :
            b = x
            df2 = dfx
        else :
            a = x
            df1 = dfx
        iter_ += 1
    return xv, y, y1
# get the values
xv, y1, y2 = Secant_Method(f, df, a, b, num_iter_max = 1000, eps = 1e-8)

# Animation update function
def update(frame):
    x1, y1_, x2, y2_ = xv[:frame], y1[:frame], xv[:frame], y2[:frame]
    sc1.set_offsets(np.c_[x1, y1_])
    sc2.set_offsets(np.c_[x2, y2_])
    return sc1, sc2

# Create the animation
ani = FuncAnimation(fig, update, frames=len(xv), blit=True, interval=300)

# Display the plot
plt.show()
```

1.6.1 Screenshots

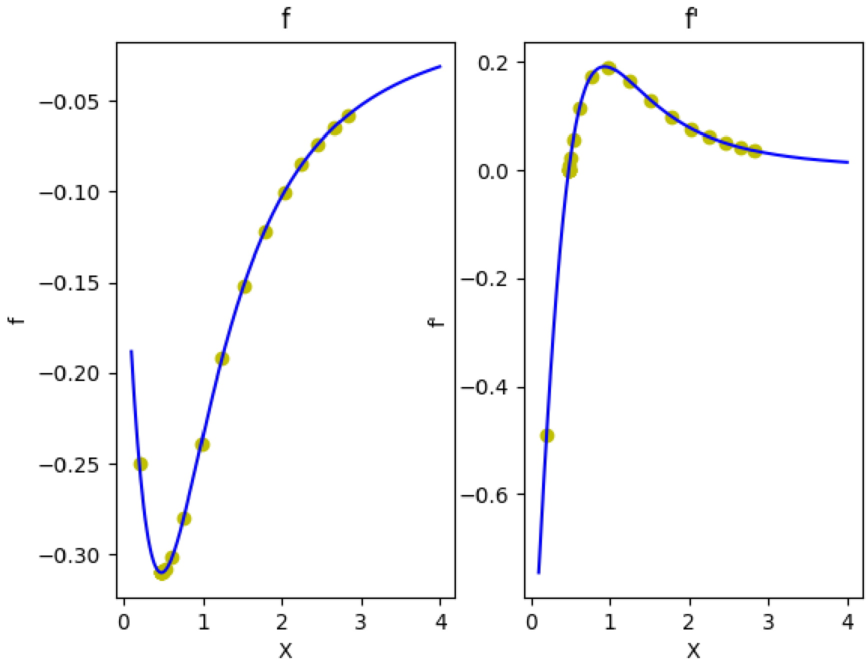


Figure 3: les valeurs de f et f' à chaque étape de Secant méthode Newto

1.7 Compare the running time of these algorithms and those from lab 1

```
from LAB1 import search_with_fixe_step,search_with_accelerat_step,\
                   exhaustive_Search, dichotomous_Search,\
                   interval_Halving,Fibonacci,\
                   golden_Section
from main import Newton_Rapson_method,Quasi_Newton_Method,\
                Secant_Method
import time
import numpy as np
from numdifftools import Derivative as Df
import matplotlib.pyplot as plt
# -----------------------------------------------------------------
f = lambda x : 0.65 - 0.75/(1 + x**2) - 0.65*x*np.arctan(1/x)
df = Df(f)
d2f = Df(df)
x0 = 0.2
a = 0.2
b = 3
delta = 1e-3
X = np.linspace(0.1, 4 ,200)
Y = f(X)
# -------------------------------------------------------------------
methods = ['Search method with fixed step size',
    'Search method with accelerated step size','Exhaustive search
       ↪ method',
    'Dichotomous search method','Interval halving method',
    'Fibonacci method','Golden section method',
    'Newton-Rapson method', 'Quasi Newton Method', 'Secant Method']
```

```python
time_of_methods = []
#---------------------------------------------------------------------
t1 = time.time()
search_with_fixe_step(f, x_init = 0.01 ,step = 0.00005)
t2 = time.time()
time_of_methods.append(t2-t1)
#---------------------------------------------------------------------
t1 = time.time()
search_with_accelerat_step(f, x_init = 0.01, step = 0.00005)
t2 = time.time()
time_of_methods.append(t2-t1)
#---------------------------------------------------------------------
t1 = time.time()
exhaustive_Search(f, 0.01, 2.0, 10000)
t2 = time.time()
time_of_methods.append(t2-t1)
#---------------------------------------------------------------------
t1 = time.time()
dichotomous_Search(f,0.01, 2.0 , 0.001, eps = 1e-9, max_iter = 10000)
t2 = time.time()
time_of_methods.append(t2-t1)
#---------------------------------------------------------------------
t1 = time.time()
interval_Halving(f,0.01, 2.0 , eps = 1e-9, max_iter = 10000)
t2 = time.time()
time_of_methods.append(t2-t1)
#---------------------------------------------------------------------
t1 = time.time()
Fibonacci(f, 0.01, 2.0, 50)
t2 = time.time()
time_of_methods.append(t2-t1)
#---------------------------------------------------------------------
t1 = time.time()
golden_Section(f, x_l= 0.01, x_u = 2.0, eps = 1e-5)
t2 = time.time()
time_of_methods.append(t2-t1)
#---------------------------------------------------------------------
t1 = time.time()
Newton_Rapson_method( df, d2f, x0,num_iter_max= 1000, eps = 1e-9)
t2 = time.time()
time_of_methods.append(t2-t1)
#---------------------------------------------------------------------
t1 = time.time()
Quasi_Newton_Method(f, x0, delta,num_iter_max = 1000, eps = 1e-8)
t2 = time.time()
time_of_methods.append(t2-t1)
# --------------------------------------------------------------------
t1 = time.time()
Secant_Method(df, a, b, num_iter_max = 1000, eps = 1e-8)
t2 = time.time()
time_of_methods.append(t2-t1)
# --------------------------------------------------------------------
fig, ax = plt.subplots()
y = np.arange(len(time_of_methods))
ax.barh(y ,time_of_methods, align='center')
ax.set_yticks(y, labels=methods)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_xlabel('running time')
```

```
ax.set_title('the running time for each method')
plt.show()
```

1.7.1 Screenshots