



NEWTON-RAPHSON

ECUE323 - Méthodes Numériques et Optimisation



EPISEN
MAUREAU CLEMENT

Table des matières

Newton-Raphson IHM PyQt5	2
1. Equation :	2
2. Résultat	3
Question 1 :	4
Pas fixe	4
1. Equation	4
2. Résultat	5
Pas accéléré	6
1. Equation	6
2. Résultat	7
Bisection	8
1. Equation	8
2. Résultat	8
Question 1 : Conclusion	8
Question n°2	9
1. Equation	9
2. Résultat	10
GitHub : MAUREAU-projet/Newton_Raphson_Optimisation	11

Newton-Raphson IHM PyQt5

1. Equation :

On définit la fonction f que l'on souhaite étudier, ici comme étant un polynôme de degrés cinq maximum dans notre interface.

La bibliothèque `scipy.misc` nous permet d'importer la fonction `derivative` qui permet d'avoir la fonction f' , qui est la fonction dérivée de $f(x)$ et f'' dérivée de f' .

```
# Partie Fonction NewtonRaphson
def Fonction(self,x):
    f= self.k5*x**5+self.k4*x**4+self.k3*x**3+self.k2*x**2+self.k1*(x**1) + self.k0
    return f

def Derivee(self,x):
    return derivative(self.Fonction,x)

def Seconde(self,x):
    return derivative(self.Derivee,x)
```

La méthode de Newton-Raphson nous permet de déterminer un optimum local de f .

Pour cela on calcule des x successifs, en partant d'un x de départ, tel que $x = x - f'(x)/f''(x)$, et pour chaque nouveau x , on regarde la valeur de $|f'(x)|$, dès que celle-ci passe en dessous d'un seuil ϵ , nous avons l'optimum de la fonction f pour la dernière valeur de x .

```
def NewtonRaphson(self):
    i=0

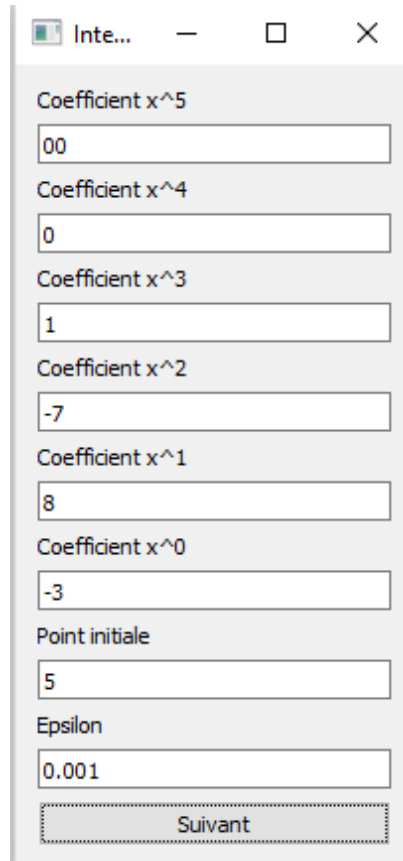
    x=self.ki #Point initiale

    f=self.Fonction(x)
    fprime=self.Derivee(x)
    fseconde=self.Seconde(x)

    while (abs(fprime)>self.ka): #ka =epsilon
        print('Etape ',i,' : x = ', x)
        i+=1
        print('f(x) = ',f)
        print("f'(x) = ",self.Derivee(x))
        print("f''(x) = ",self.Seconde(x)," | Fin d'etape")
        x=x- fprime/fseconde
        f=self.Fonction(x)
        fprime=self.Derivee(x)
        fseconde=self.Seconde(x)
        print('\n')
    return x
```

Le reste du fichier sert à design l'interface et récupérer les coefficients de la fonction f , lancer l'interface, le bouton ainsi que la fonction de ce dernier qui attribue les coefficients et lance la fonction Newton Raphson.

2. Résultat



Interface window titled 'Inte...' with the following fields:

- Coefficient x^5 : 00
- Coefficient x^4 : 0
- Coefficient x^3 : 1
- Coefficient x^2 : -7
- Coefficient x^1 : 8
- Coefficient x^0 : -3
- Point initiale: 5
- Epsilon: 0.001
- Button: Suivant

```
C:\Users\Clément\Bureau\Cours\ecole\Semestre 3 bis\optimisation\test>python interface.py
Warning: QT_DEVICE_PIXEL_RATIO is deprecated. Instead use:
  QT_AUTO_SCREEN_SCALE_FACTOR to enable platform plugin controlled per-screen factors.
  QT_SCREEN_SCALE_FACTORS to set per-screen DPI.
  QT_SCALE_FACTOR to set the application global scale factor.

Le polynome de la fonction : 0 * x**5 + 0 * x**4 + 1 * x**3 + -7 * x**2 + 8 *x**1 + x -3
Point initiale : 5
Epsilon : 0.001

Etape 0 : x = 5
f(x) = -13
f'(x) = 14.0
f''(x) = 16.0 | Fin d'etape

Etape 1 : x = 4.125
f(x) = -18.919921875
f'(x) = 2.296875
f''(x) = 10.75 | Fin d'etape

Etape 2 : x = 3.9113372093023258
f(x) = -18.96139153394611
f'(x) = 0.13695536438616251
f''(x) = 9.46802325581396 | Fin d'etape

3.89687216621088
```

Question 1 :

```

1
2  def f(x):
3      return x**5 - 5*x**3 - 20*x + 5
4
5  def fprime(x):
6      return 5*x**4 - 15*x**2 - 20
7
8  def fseconde(x):
9      return 20*x**3 - 30*x
10

```

Pas fixe

1. Equation

Le principe du pas fixe pour détecter un optimum, consiste à comparer $f(x)$ et $f(x+p)$ pour s'assurer que la fonction est toujours monotone (c'est-à-dire soit toujours croissante, soit toujours décroissante), dès que la fonction n'est plus monotone, c'est-à-dire qu'elle passe de croissante à décroissante ou inversement, on a un optimum local de la fonction.

Pour cette fonction, j'ai pris $x=0$ comme départ, i sert à compter le nombre de calcul.

```

12 def PasFixe(p):
13
14     i=1
15     x=0
16     print('Etape 0')
17     print("x + i*p = ",x,"|          f'(0) = ",f(x))
18     print('\n')
19
20     while f(x+i*p)<f(x+(i-1)*p):
21         i+=1
22         print("x + i*p = ",float(round(x+i*p,2)), "|          f(x+i*p) = ",float(round(f(float(round(x+i*p,2))),2)), "|
23         # Attention float(round()) sert à tronquer la valeur de x, donc à adapter par rapport au pas
24
25     i+=1
26     print("x + i*p = ",float(round(x+i*p,2)), "|          f(x+i*p) = ",float(round(f(float(round(x+i*p,2))),2)), "|
27
28

```

```

f(xi) > f(x(i-1)) ? ",f(float(round(x+i*p,2)))>f(float(round(x+(i-1)*p)))

f(xi) > f(x(i-1)) ? ",f(x+i*p)>f(x+(i-1)*p))

```

Suite des lignes 22 et 26

Le « while » correspond au tant que la fonction est monotone.

Je me suis retrouvé dans l'obligation de tronquer la valeur de x avec `float/round`, je tronque à la même décimale que mon pas ici 2 mon pas étant 0.05. Je dois tronquer car lorsque j'attribue à x la valeur $x = x+i*p$ j'obtenais ce genre de résultat. Ce qui décale mon affichage en plus d'être incompréhensible mais qui ne perturbe pas la fonction.

$x + i*p = 0.1$	$f(x+i*p) = 3.0$	$f(xi) > f(x(i-1)) ?$ False
$x + i*p = 0.15000000000000002$	$f(x+i*p) = 1.98$	$f(xi) > f(x(i-1)) ?$ False
$x + i*p = 0.2$	$f(x+i*p) = 0.96$	$f(xi) > f(x(i-1)) ?$ False
$x + i*p = 0.25$	$f(x+i*p) = -0.08$	$f(xi) > f(x(i-1)) ?$ False
$x + i*p = 0.30000000000000004$	$f(x+i*p) = -1.13$	$f(xi) > f(x(i-1)) ?$ False
$x + i*p = 0.35000000000000003$	$f(x+i*p) = -2.21$	$f(xi) > f(x(i-1)) ?$ False
$x + i*p = 0.4$	$f(x+i*p) = -3.31$	$f(xi) > f(x(i-1)) ?$ False

2. Résultat

```
79
80 PasFixe(0.05)
81 #PasAccelere(0.05)
82 #Bissection(0,2.5,0.05)
```

```
C:\Users\Clément\Bureau\Cours\ecole\Semestre 3 bis\optimisation\test>python question1.py
```

Etape 0

```
x + i*p = 0 | f'(0) = 5
```

x + i*p = 0.1	f(x+i*p) = 3.0	f(xi) > f(x(i-1)) ?	False
x + i*p = 0.15	f(x+i*p) = 1.98	f(xi) > f(x(i-1)) ?	False
x + i*p = 0.2	f(x+i*p) = 0.96	f(xi) > f(x(i-1)) ?	False
x + i*p = 0.25	f(x+i*p) = -0.08	f(xi) > f(x(i-1)) ?	False
x + i*p = 0.3	f(x+i*p) = -1.13	f(xi) > f(x(i-1)) ?	False
x + i*p = 0.35	f(x+i*p) = -2.21	f(xi) > f(x(i-1)) ?	False
x + i*p = 0.4	f(x+i*p) = -3.31	f(xi) > f(x(i-1)) ?	False
x + i*p = 0.45	f(x+i*p) = -4.44	f(xi) > f(x(i-1)) ?	False
x + i*p = 0.5	f(x+i*p) = -5.59	f(xi) > f(x(i-1)) ?	False
x + i*p = 0.55	f(x+i*p) = -6.78	f(xi) > f(x(i-1)) ?	False
x + i*p = 0.6	f(x+i*p) = -8.0	f(xi) > f(x(i-1)) ?	True
x + i*p = 0.65	f(x+i*p) = -9.26	f(xi) > f(x(i-1)) ?	True
x + i*p = 0.7	f(x+i*p) = -10.55	f(xi) > f(x(i-1)) ?	True
x + i*p = 0.75	f(x+i*p) = -11.87	f(xi) > f(x(i-1)) ?	True
x + i*p = 0.8	f(x+i*p) = -13.23	f(xi) > f(x(i-1)) ?	True
x + i*p = 0.85	f(x+i*p) = -14.63	f(xi) > f(x(i-1)) ?	True
x + i*p = 0.9	f(x+i*p) = -16.05	f(xi) > f(x(i-1)) ?	True
x + i*p = 0.95	f(x+i*p) = -17.51	f(xi) > f(x(i-1)) ?	True
x + i*p = 1.0	f(x+i*p) = -19.0	f(xi) > f(x(i-1)) ?	False
x + i*p = 1.05	f(x+i*p) = -20.51	f(xi) > f(x(i-1)) ?	False
x + i*p = 1.1	f(x+i*p) = -22.04	f(xi) > f(x(i-1)) ?	False
x + i*p = 1.15	f(x+i*p) = -23.59	f(xi) > f(x(i-1)) ?	False
x + i*p = 1.2	f(x+i*p) = -25.15	f(xi) > f(x(i-1)) ?	False
x + i*p = 1.25	f(x+i*p) = -26.71	f(xi) > f(x(i-1)) ?	False
x + i*p = 1.3	f(x+i*p) = -28.27	f(xi) > f(x(i-1)) ?	False
x + i*p = 1.35	f(x+i*p) = -29.82	f(xi) > f(x(i-1)) ?	False
x + i*p = 1.4	f(x+i*p) = -31.34	f(xi) > f(x(i-1)) ?	False
x + i*p = 1.45	f(x+i*p) = -32.83	f(xi) > f(x(i-1)) ?	False

x + i*p = 1.45	f(x+i*p) = -32.83	f(xi) > f(x(i-1)) ?	False
x + i*p = 1.5	f(x+i*p) = -34.28	f(xi) > f(x(i-1)) ?	False
x + i*p = 1.55	f(x+i*p) = -35.67	f(xi) > f(x(i-1)) ?	True
x + i*p = 1.6	f(x+i*p) = -36.99	f(xi) > f(x(i-1)) ?	True
x + i*p = 1.65	f(x+i*p) = -38.23	f(xi) > f(x(i-1)) ?	True
x + i*p = 1.7	f(x+i*p) = -39.37	f(xi) > f(x(i-1)) ?	True
x + i*p = 1.75	f(x+i*p) = -40.38	f(xi) > f(x(i-1)) ?	True
x + i*p = 1.8	f(x+i*p) = -41.26	f(xi) > f(x(i-1)) ?	True
x + i*p = 1.85	f(x+i*p) = -41.99	f(xi) > f(x(i-1)) ?	True
x + i*p = 1.9	f(x+i*p) = -42.53	f(xi) > f(x(i-1)) ?	True
x + i*p = 1.95	f(x+i*p) = -42.88	f(xi) > f(x(i-1)) ?	True
x + i*p = 2.0	f(x+i*p) = -43.0	f(xi) > f(x(i-1)) ?	False
x + i*p = 2.05	f(x+i*p) = -42.87	f(xi) > f(x(i-1)) ?	True
x + i*p = 2.1	f(x+i*p) = -42.46	f(xi) > f(x(i-1)) ?	True

Pas accéléré

1. Equation

Le pas accéléré au même principe que le pas fixe, mais au lieu d'ajouter une valeur fixe à x , on vient augmenter cette valeur de pas à chaque itération.

```
34 def PasAccelere(p):
35     i=1
36     x=0
37     pdebut=p
38     print('Étape 0')
39     print("x + i*p = ",x,"|          f'(0) = ",f(x))
40     print('\n')
41
42     while f(x+i*p)<f(x+(i-1)*p):
43         i+=1
44         print('Pas : ',p,' | ', "          x + i*p = ",float(round(x+i*p,2)),"|          f(x+i*p) = ",float(round(f(float(round(x+i*p,2))),2)), " |
45         p=2*p
46
47
48     i+=1
49     print('Pas : ',p,' | ', "          x + i*p = ",float(round(x+i*p,2)),"|          f(x+i*p) = ",float(round(f(float(round(x+i*p,2))),2)), " |
50
51     i=i-2 # Le point précédent étant plus proche de l'optimum on choisit de revenir 2 points en arrière et de recommencer le processus
52
53     print('\n')
54     print('Repartons en arrière et reprenons un pas plus petit')
55     print('\n')
56
57     p=pdebut
58     while f(x+i*p)<f(x+(i-1)*p):
59         i+=1
60         print('Pas : ',p,' | ', "          x + i*p = ",float(round(x+i*p,2)),"|          f(x+i*p) = ",float(round(f(float(round(x+i*p,2))),2)), " |
61         p=p
62
63
```

La fonction est la même que le pas fixe avec la ligne 45 qui vient se rajouter $p=2*p$

Et qu'une fois fini, pour se rapprocher encore plus de la valeur de l'optimum, on reprend 2 valeurs de pas avant, et on recommence avec un pas fixe.

2. Résultat

```
78
79
80 #PasFixe(0.05)
81 PasAccelere(0.05)
82 #Bissection(0,2.5,0.05)
```

```
C:\Users\Clément\Bureau\Cours\ecole\Semestre 3 bis\optimisation\test>python question1.py
Etape 0
x + i*p = 0 | f'(0) = 5

Pas : 0.05 | x + i*p = 0.1 | f(x+i*p) = 3.0 | f(xi) > f(x(i-1)) ? False
Pas : 0.1 | x + i*p = 0.3 | f(x+i*p) = -1.13 | f(xi) > f(x(i-1)) ? False
Pas : 0.2 | x + i*p = 0.8 | f(x+i*p) = -13.23 | f(xi) > f(x(i-1)) ? True
Pas : 0.4 | x + i*p = 2.0 | f(x+i*p) = -43.0 | f(xi) > f(x(i-1)) ? False
Pas : 0.8 | x + i*p = 4.8 | f(x+i*p) = 1904.08 | f(xi) > f(x(i-1)) ? True

Repardons en arrière et reprenons un pas plus petit
```

```
Repardons en arrière et reprenons un pas plus petit

Pas : 0.05 | x + i*p = 0.25 | f(x+i*p) = -0.08 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 0.3 | f(x+i*p) = -1.13 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 0.35 | f(x+i*p) = -2.21 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 0.4 | f(x+i*p) = -3.31 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 0.45 | f(x+i*p) = -4.44 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 0.5 | f(x+i*p) = -5.59 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 0.55 | f(x+i*p) = -6.78 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 0.6 | f(x+i*p) = -8.0 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 0.65 | f(x+i*p) = -9.26 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 0.7 | f(x+i*p) = -10.55 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 0.75 | f(x+i*p) = -11.87 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 0.8 | f(x+i*p) = -13.23 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 0.85 | f(x+i*p) = -14.63 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 0.9 | f(x+i*p) = -16.05 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 0.95 | f(x+i*p) = -17.51 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 1.0 | f(x+i*p) = -19.0 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 1.05 | f(x+i*p) = -20.51 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 1.1 | f(x+i*p) = -22.04 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 1.15 | f(x+i*p) = -23.59 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 1.2 | f(x+i*p) = -25.15 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 1.25 | f(x+i*p) = -26.71 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 1.3 | f(x+i*p) = -28.27 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 1.35 | f(x+i*p) = -29.82 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 1.4 | f(x+i*p) = -31.34 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 1.45 | f(x+i*p) = -32.83 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 1.5 | f(x+i*p) = -34.28 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 1.55 | f(x+i*p) = -35.67 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 1.6 | f(x+i*p) = -36.99 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 1.65 | f(x+i*p) = -38.23 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 1.7 | f(x+i*p) = -39.37 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 1.75 | f(x+i*p) = -40.38 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 1.8 | f(x+i*p) = -41.26 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 1.85 | f(x+i*p) = -41.99 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 1.9 | f(x+i*p) = -42.53 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 1.95 | f(x+i*p) = -42.88 | f(xi) > f(x(i-1)) ? True
Pas : 0.05 | x + i*p = 2.0 | f(x+i*p) = -43.0 | f(xi) > f(x(i-1)) ? False
Pas : 0.05 | x + i*p = 2.05 | f(x+i*p) = -42.87 | f(xi) > f(x(i-1)) ? True
```

Ré-utiliser le pas fixe ne semble pas très optimisé au vu du nombre d'étape de calcul que cela vient rajouter après l'utilisation du pas accéléré.

Bissection

1. Equation

Le principe de la bisection consiste à regarder une fonction sur un intervalle qu'on coupe en deux, on regarde lorsque la fonction passe de positif à négatif, ou inversement, pour déterminer la racine de cette fonction. Par conséquent pour trouver l'optimum de notre fonction, on regarde la racine de sa dérivée, qui est dérivable car fonction polynomiale.

```
def Bissection(a,b,e):  
    while abs(a-b)>e:  
        x=(a+b)/2  
        print(x)  
        if fprime(a)*fprime(x)<0:  
            b=x  
        else:  
            a=x
```

2. Résultat

```
else:  
    a=x  
C:\Users\Clément\Bureau\Cours\ecole\Semestre 3 bis\optimisation\test>python question1.py  
1.25  
1.875  
2.1875  
2.03125  
#PasFixe(0.05) 1.953125  
#PasAccelere(0.05) 1.9921875  
Bissection(0,2.5,0.05)
```

Question 1 : Conclusion

Le résultat vaut 2, il est atteint en **6 étapes** pour la **Bissection**, en **4 étapes** pour le **pas accéléré** si l'on ne compte pas la réutilisation du pas fixe, et en beaucoup **trop d'étapes** pour le **pas fixe** en comparaison de ces deux précédentes méthodes.

Le résultat est néanmoins le même pour chacune de ces méthodes, c'est-à-dire $x=2$, donc on peut en conclure que même si leur complexité algorithmique les rend plus ou moins appréciable au niveau du temps de calcul, le résultat reste fiable quel que soit la méthode utilisée.

Le fichier doit être téléchargé, l'une des trois dernières lignes doit être décommentée avant d'être exécuter à l'aide d'un compilateur, par exemple l'invité de commande Windows comme dans l'exemple.

Question n°2

```
def f(x):  
    return x**3 - 7*x**2 + 8*x - 3  
  
def fprime(x):  
    return 3*x**2 - 14*x + 8  
  
def fseconde(x):  
    return 6*x - 14
```

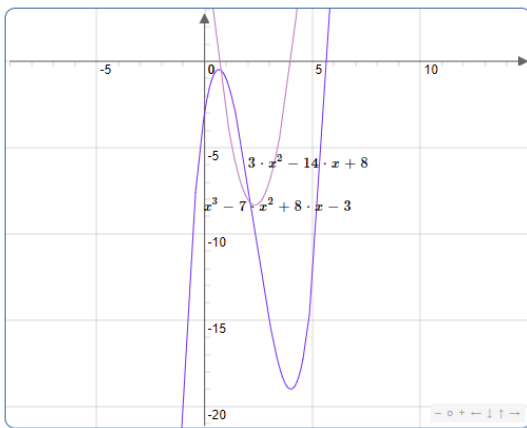
1. Equation

Le principe de la méthode de newton raphson est de déterminer une racine de la fonction f , en faisant varier x par rapport à la dérivée de la fonction f c'est-à-dire :

$$X = x - f(x)/f'(x)$$

On peut ainsi progresser sur la fonction jusqu'à ce que la valeur de notre fonction tombe en valeur absolue en dessous d'un seuil noté epsilon, et ainsi supposer que pour cette valeur de x la fonction f vaut approximativement zéro.

Ce qui correspond à la fonction NewtonRaphson du fichier question2.py or nous ne souhaitons pas trouver une racine de la fonction f mais comme précédemment nous voulons un optimum, nous référons donc la même étape sur f' et f'' .



Ici les fonctions f et f' sur un graphique pour vérifier les résultats.

```
9  
10 def NewtonRaphson(x,e):  
11     print('\n')  
12     i=0  
13     print('Etape ',i,'\n x = ',x, '\n f(x) = ', f(x))  
14  
15     x=x-f(x)/fprime(x)  
16     i=i+1  
17     print('\n')  
18  
19     print('Etape ',i,'\n x = ',x, '\n f(x) = ', f(x))  
20     print('\n')  
21  
22  
23  
24     while (abs(f(x))>e):  
25         i+=1  
26  
27  
28         x=x-f(x)/fprime(x)  
29         print('Etape ',i,'\n x = ',x, '\n f(x) = ', f(x))  
30         print('\n')  
31
```

```
35 def NewtonRaphsonPrime(x,e):  
36     print('\n')  
37     i=0  
38     print('Etape ',i,'\n x = ',x, '\n f'(x) = ', fprime(x))  
39  
40     x=x-fprime(x)/fseconde(x)  
41     i=i+1  
42     print('\n')  
43  
44     print('Etape ',i,'\n x = ',x, '\n f'(x) = ', fprime(x))  
45     print('\n')  
46  
47  
48  
49     while (abs(fprime(x))>e):  
50         i+=1  
51  
52  
53         x=x-fprime(x)/fseconde(x)  
54         print('Etape ',i,'\n x = ',x, '\n f'(x) = ', fprime(x))  
55         print('\n')
```

2. Résultat

```
C:\Users\Clément\Bureau\Cours\ecole\Semestre 3 bis\optimisation\test>python question2.py

Etape 0
x = 5
f'(x) = 13

Etape 1
x = 4.1875
f'(x) = 1.98046875

Etape 2
x = 4.009480337078652
f'(x) = 0.09507300115989636

Etape 3
x = 4.000026810533599
f'(x) = 0.00026810749240979703
```

En quatre étapes on retrouve le même résultat que sur le graphique, la fonction f possède un optimum local en $x = 4$.

Le fichier doit être téléchargé puis exécuter à l'aide d'un compilateur, par exemple l'invité de commande Windows comme dans l'exemple.

GitHUB : MAUREAU-projet/Newton_Raphson_Optimisation

Lien du github publique correspondant :

https://github.com/MAUREAU-projet/Newton_Raphson_Optimisation