TP2_correctionPartielle

November 9, 2023

1 TP2 – Méthode de Newton – Éléments de correction

[ ]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
```

Programmons la méthode de Newton pour une fonction F, de différentielle dF supposée calculable explicitement.

[ ]:
```
def newton(f,df,x0,eps,Nmax):
    tol = 1
    nbiter = 0
    xt, x = x0, x0
    listx = [x]
    while (tol>eps) and (nbiter<Nmax):
        nbiter += 1
        x = x - f(x)/df(x)
        listx.append(x)
        tol = abs(x-xt)
        xt = x
    return x, nbiter, listx
```

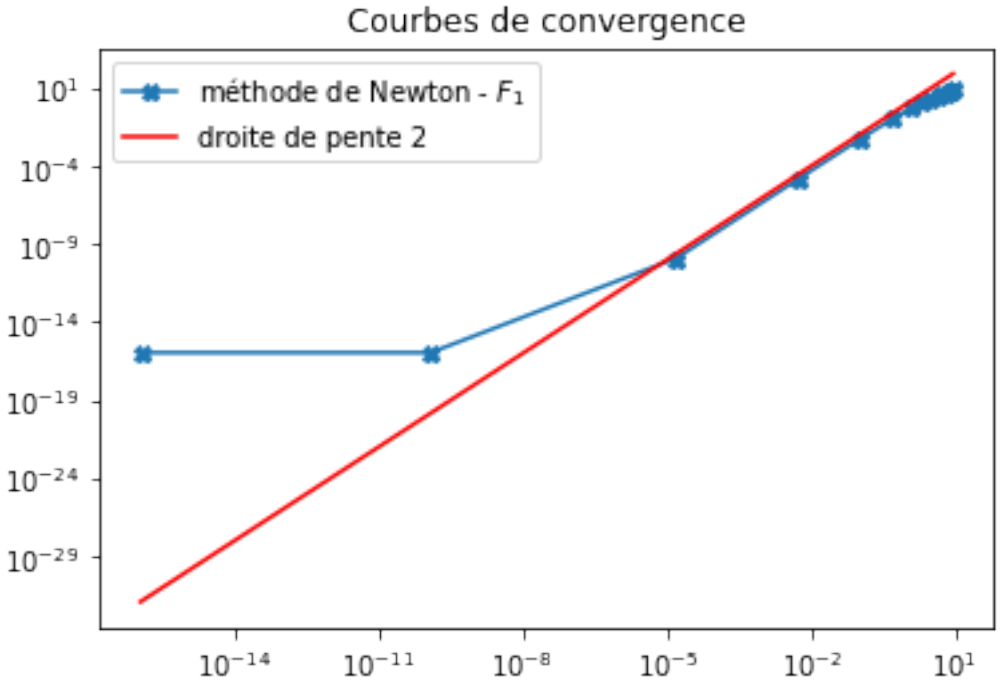
Remarques. La programmation précédente se limite au cas monodimensionnel (formule d'itération avec `/` et utilisation de la valeur absolue pour l'incrément `tol`). Les itérés sont stockés successivement avec la commande `append` dans une liste `listx`.

Testons la méthode sur l'exemple monodimensionnel $F_1(x) = e^x - e$, et illustrons graphiquement la vitesse de convergence.

[ ]:
```
def f(x):
    return np.exp(x)-np.exp(1.)

def df(x):
    return np.exp(x)

x, nbiter, listx = newton(f,df,10,1e-16,15)
print(listx)
```

```
[10, 9.000123409804086, 8.000458831035166, 7.001370294696947, 6.003845652578774,
5.010557737534697, 4.028681021916206, 3.077060429355986, 2.202358423033882,
1.5028431285659987, 1.1076517927211909, 1.0055920040385937, 1.0000156061511463,
1.0000000001217753, 0.9999999999999999, 0.9999999999999999]
```

```
[ ]: xsol = 1.
Erreur = np.array([abs(x - xsol) for x in listx])

plt.loglog(Erreur[:-1],Erreur[1:],'X-',label = 'méthode de Newton - $F_1$')
plt.loglog(Erreur[:-1],Erreur[:-1]**2,'r',label = 'droite de pente 2')
plt.legend()
plt.title("Courbes de convergence")
plt.show()

fit = np.polyfit(np.log(Erreur[10:12]), np.log(Erreur[11:13]), 1)
print('Ordre effectif de convergence (avant la saturation): ', fit[0])
```

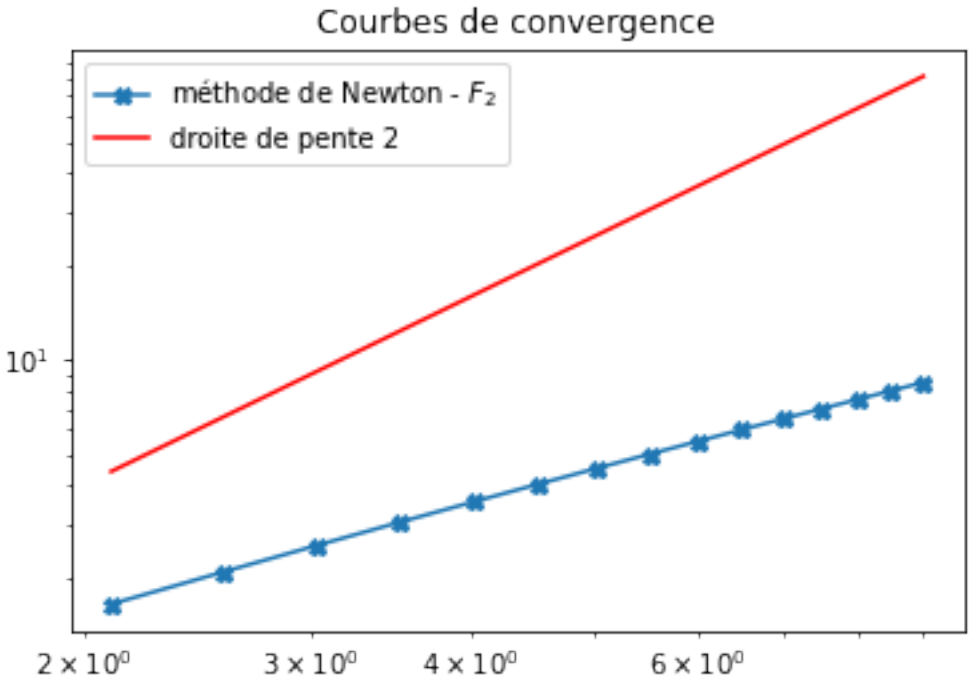



```
Ordre effectif de convergence (avant la saturation):  1.9886053412547247
```

Remarques. La représentation en échelle log-log de e_{n+1} en fonction de e_n se prête bien à la comparaison avec la droite de pente 2 qui témoigne de la convergence à l'ordre 2. Il y a une saturation sur l'erreur machine (qui peut apparaître ou ne pas apparaître selon les cas).

[ ]:
```
def f2(x):
    return (np.exp(x)-np.exp(1))**2

def df2(x):
    return 2*np.exp(x)*(np.exp(x)-np.exp(1))

x, nbiter, listx = newton(f2,df2,10,1e-16,15)
print(listx)

xsol = 1.
Erreur = np.array([abs(x - xsol) for x in listx])

plt.loglog(Erreur[:-1],Erreur[1:],'X-',label = 'méthode de Newton - $F_2$')
plt.loglog(Erreur[:-1],Erreur[:-1]**2,'r',label = 'droite de pente 2')
plt.legend()
plt.title("Courbes de convergence")
plt.show()

fit = np.polyfit(np.log(Erreur[10:12]), np.log(Erreur[11:13]), 1)
print('Ordre effectif de convergence: ', fit[0])
```

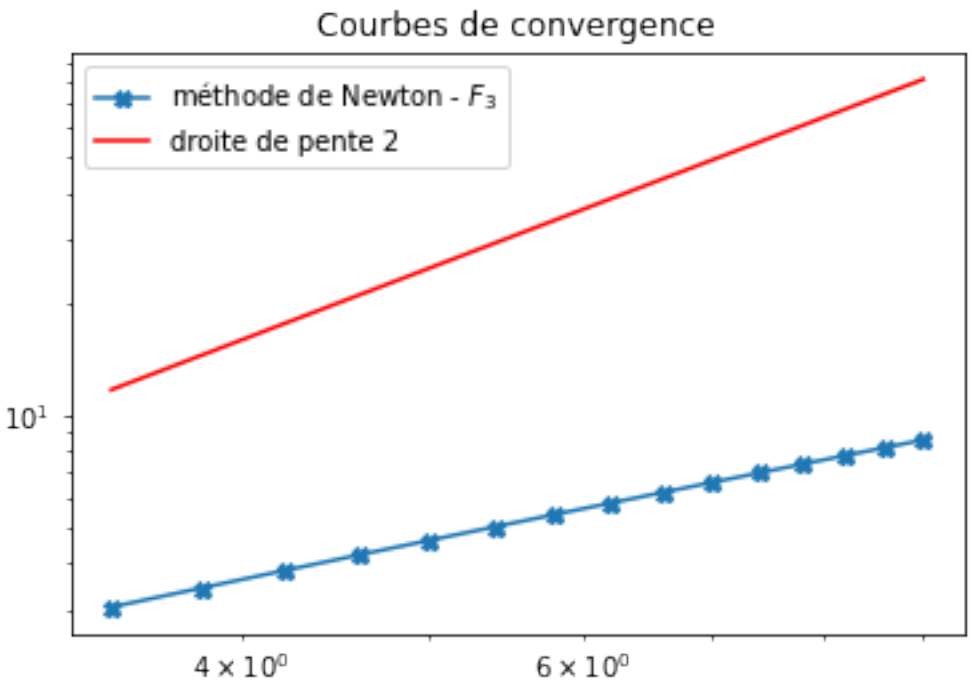
```
[10, 9.500061704902043, 9.000163432809245, 8.500331136712637, 8.000607587339601,
7.501063251382551, 7.0018141721369, 6.503051301821964, 6.0050884620573735,
5.508440336204689, 5.013948149934721, 4.522979121443937, 4.037734814481659,
3.56170649808558, 3.100292964359917, 2.661503243429142]
```



```
Ordre effectif de convergence:  1.1358682068657693
```

[ ]:
```python
def f3(x):
    return (np.exp(x)-np.exp(1))**(5/2)

def df3(x):
    return 5/2*np.exp(x)*(np.exp(x)-np.exp(1))**(3/2)

x, nbiter, listx = newton(f3,df3,10,1e-16,15)
print(listx)

xsol = 1.
Erreur = np.array([abs(x - xsol) for x in listx])

plt.loglog(Erreur[:-1],Erreur[1:],'X-',label = 'méthode de Newton - $F_3$')
plt.loglog(Erreur[:-1],Erreur[:-1]**2,'r',label = 'droite de pente 2')
plt.legend()
plt.title("Courbes de convergence")
plt.show()

fit = np.polyfit(np.log(Erreur[10:12]), np.log(Erreur[11:13]), 1)
print('Ordre effectif de convergence: ', fit[0])
```

```
[10, 9.600049363921634, 9.200123002603918, 8.800232850519496, 8.40039670635273,
8.000641110481277, 7.601005629495609, 7.201549229775193, 6.802359745381633,
6.403567912945528, 6.005368110901081, 5.608048860445591, 5.212037362305403,
4.817963822984778, 4.426752821091107, 4.039749732249343]
```




```
Ordre effectif de convergence:  1.0864700595381995
```

1.1 Application à la résolution d'un problème

Détermination approchée du paramètre $\lambda \in [0, 10]$ tel que $\int_0^1 \sin(\lambda e^{x^2})dx = 1/2$.

Il s'agit de résoudre $F(\lambda) = 1/2$ avec $F(\lambda) = \int_0^1 \sin(\lambda e^{x^2})dx*$.
On a en particulier $F \in \mathcal{C}^1([0, 10])$ avec $F'(\lambda) = \int_0^1 e^{x^2} \cos(\lambda e^{x^2})dx$.
La méthode de Newton appliquée à ce problème prend la forme :

$$\lambda_{n+1} = \lambda_n - F(\lambda_n)/F'(\lambda_n).$$

Dans cette itération, nous pouvons calculer les intégrales qui interviennent de façon approchée par une quadrature maison ou par la fonction `scipy.integrate.quad`.

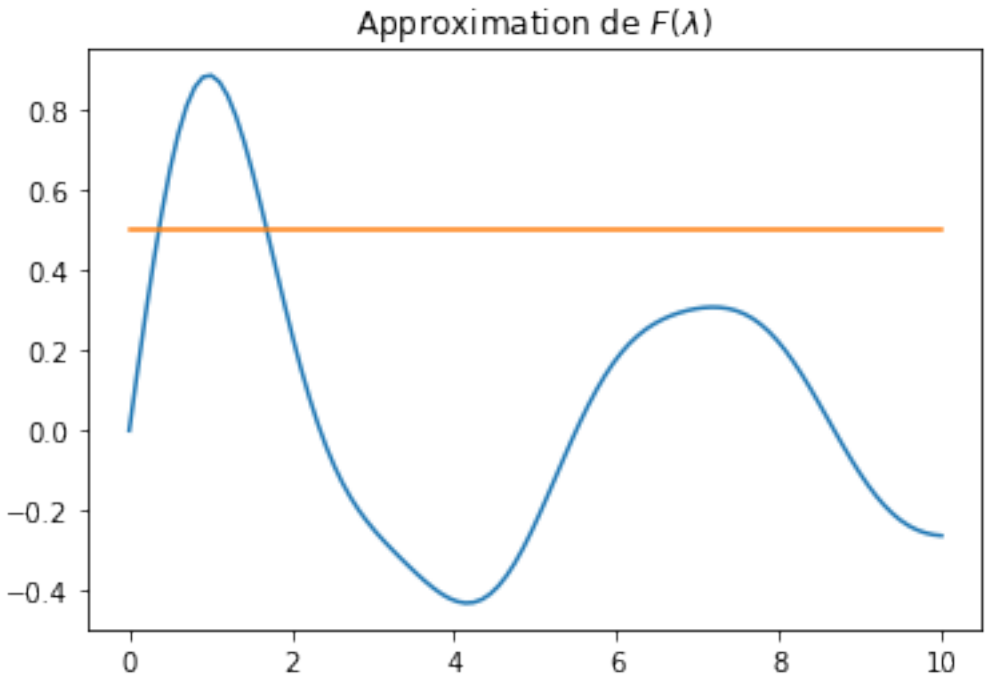
Une alternative est de considérer la méthode de Newton, appliquée à une approximation de F:

$$\lambda_{n+1} = \lambda_n - F_N(\lambda_n)/F'_N(\lambda_n),$$

avec par exemple $F_N(\lambda_n)$ l'approximation obtenue par la méthode des rectangles à gauche.

[ ]:
```
def f(x):
    return np.sin(lam*np.exp(x**2))

def df(x):
    return np.exp(x**2)*np.cos(lam*np.exp(x**2))
```

```
N = 500
X = np.linspace(0,1,N)
Lam = np.linspace(0,10,100)
Integral = []
for lam in Lam:
    I=(X[1]-X[0])*np.sum(f(X))
    Integral.append(I)

plt.plot(Lam,Integral,Lam,0.5*np.ones_like(Lam));
plt.title('Approximation de $F(\lambda)$');
```



[ ]:
```
for N in [100,500,1000,5000,10000] :

    print('-----------------')
    print('quadrature N=',N)

    lam = 2
    for p in range(15):
        X = np.linspace(0,1,N)
        Y = f(X)
        DY = df(X)
        F = (X[1]-X[0])*np.sum(Y)-0.5
        DF = (X[1]-X[0])*np.sum(DY)
```

```
        lam = lam - F/DF
    print('solution1=',lam)

    lam = 0.5
    for p in range(15):
        X = np.linspace(0,1,N)
        Y = f(X)
        DY = df(X)
        F = (X[1]-X[0])*np.sum(Y)-0.5
        DF = (X[1]-X[0])*np.sum(DY)
        lam = lam - F/DF
    print('solution2=',lam)
```

```
-----------------
quadrature N= 100
solution1= 1.6952363645841573
solution2= 0.35954377806477567
-----------------
quadrature N= 500
solution1= 1.6952564969396222
solution2= 0.3635314371366633
-----------------
quadrature N= 1000
solution1= 1.695258138658196
solution2= 0.3640305940763186
-----------------
quadrature N= 5000
solution1= 1.6952593147966903
solution2= 0.3644300320538716
-----------------
quadrature N= 10000
solution1= 1.695259453266675
solution2= 0.3644799688338149
```

[ ]:
```
from scipy.integrate import quad

print('------------------')
print('quadrature de scipy')

lam = 2
for p in range(15):
    F = quad(lambda x: f(x)-0.5,0,1)[0]
    DF = quad(lambda x: df(x),0,1)[0]
    lam = lam - F/DF
print('solution1=',lam)

lam = 0.5
```

```
for p in range(15):
    F = quad(lambda x: f(x)-0.5,0,1)[0]
    DF = quad(lambda x: df(x),0,1)[0]
    lam = lam - F/DF
print('solution2=',lam)
```

```
------------------
quadrature de scipy
solution1= 1.6952595898406824
solution2= 0.3645299071773117
```

1.1.1 Variante de la méthode du programme pour le cas de systèmes d'équations

[ ]:
```
def newton_system(f,df,x0,eps,Nmax):
    tol = 1
    nbiter = 0
    listx = []
    xt, x = x0, x0
    listx.append(x)
    while (tol>eps) and (nbiter<Nmax):
        nbiter += 1
        x = x - np.linalg.solve(df(x),f(x))
        listx.append(x)
        tol = np.linalg.norm(x-xt)
        xt = x
    return x, nbiter, listx
```

[ ]:
```
def f(x):
    return np.array([x[0]**2 + x[1]**2 - 2, x[0]**2 - x[1]**2 - 1])

def df(x):
    return np.array([ [2*x[0], 2*x[1]],
                      [2*x[0], -2*x[1]]  ])

x0 = np.array([2,4])
x, nbiter, listx = newton_system(f,df,x0,1e-16,12)
print(x)
```

```
[1.22474487 0.70710678]
```

1.1.2 Représentation graphique des solutions

[ ]:
```
t1 = np.linspace(0,2*np.pi,100)
t2 = np.linspace(-1.5,1.5,100)
plt.plot(np.sqrt(2)*np.cos(t1),np.sqrt(2)*np.sin(t1),'b')
```

```
plt.plot(np.cosh(t2),np.sinh(t2),'r',-np.cosh(t2),np.sinh(t2),'r')

x0 = np.array([2,4])
x, nbiter, listx = newton_system(f,df,x0,1e-16,12)
print(x)
plt.plot([listx[i][0] for i in range(len(listx))],[listx[i][1] for i in␣
 ↪range(len(listx))],'gx-')

x0 = np.array([3,-1])
x, nbiter, listx = newton_system(f,df,x0,1e-16,12)
print(x)
plt.plot([listx[i][0] for i in range(len(listx))],[listx[i][1] for i in␣
 ↪range(len(listx))],'yo-')

plt.legend(['equation1','equation2'])
plt.show()
```

```
[1.22474487 0.70710678]
[ 1.22474487 -0.70710678]
```