cours-non-lineaire

October 10, 2023

1 Résolution de systèmes non linéaires

Objectifs : - Implémentation des méthodes de la sécante, de dichotomie, de Picard, et de Newton-Raphson. - Estimation de leur vitesse de convergence. - Application à un problème de modélisation.

1.1 Méthode de la sécante

1.1.1 L'algorithme

Soit F une fonction sur $\mathbb{R}$. Étant donnés deux réels distincts x_0 et x_1, on approxime F par la fonction affine

$$x \mapsto F(x_0) + \frac{x - x_0}{x_1 - x_0}(F(x_1) - F(x_0)).$$

Cette fonction s'annule en

$$x_2 = \frac{x_0 F(x_1) - x_1 F(x_0)}{F(x_1) - F(x_0)}.$$

On considère alors les deux réels x_1 et x_2, et on itère l'algorithme pour trouver x_3 et ainsi de suite. Cela définit une suite $(x_n)_{n\in\mathbb{N}}$. Cette suite peut malheureusement ne pas converger. Étant donnés un seuil d'erreur $\epsilon > 0$ et un nombre d'itérations maximal $N \in \mathbb{N}$, l'algorithme s'arrête soit lorsque $|F(x_n)| < \epsilon$ auquel cas il renvoie x_n, soit lorsque le nombre d'itérations a dépassé N et il renvoie le message "L'algorithme n'a pas convergé".

[1]:
```
def secante(F,x0,x1,eps,N):
    x=x0
    y=x1
    z=0
    n=0
    e=2*eps
    while e>eps and n<N:
        z=x-(y-x)*F(x)/(F(y)-F(x))
        x=y
        y=z
        n=n+1
        e=abs(F(y))
    if n<N:
        return(y)
    else:
```

```
        return("l'alorithme n'a pas converge")
```

On illustre la fonction `secante(F,a0,b0,eps,N)` ci-dessous en déterminant la solution de $x^2-2=0$ sur l'intervalle $[1,2]$.

[47]:
```
def F2(x):
    return(x**2-2)
secante(F2,1,2,0.000000001,100)
```

[47]:
```
1.4142135620573204
```

1.1.2 La routine de scipy

La fonction `scipy.optimize.newton` peut être utilisée à la fois pour la méthode de la sécante, et pour la méthode de Newton. Si on précise seulement deux arguments `scipy.optimize.newton(f,x0)`, la méthode de la sécante est utilisée (voir plus loin pour son utilisation pour la méthode de Newton).

[52]:
```
import scipy.optimize as sc_opt
sc_opt.newton(F2,1)
```

[52]:
```
1.414213562373095
```

1.2 Méthode de dichotomie

1.2.1 L'algorithme

On suppose que F est une fonction continue sur $[x_0,y_0]$ qui prend des valeurs de signes opposés aux extrémités de l'intervalle : $F(x_0)F(y_0)\leq 0$. Alors F admet un zéro sur $[x_0,y_0]$. La méthode par dichotomie consiste à poser $z_0=\frac{x_0+y_0}{2}$. Si $F(z_0)$ est du même signe que $F(x_0)$ alors on pose $x_1=z_0$ et $y_1=y_0$. Sinon, on pose $x_1=x_0$ et $y_1=z_0$. Dans les deux cas, on a $F(x_1)F(y_1)\leq 0$, et on peut itérer l'algorithme. Cela définit deux suites $(x_k)_{k\in\mathbb{N}}$ et $(y_k)_{k\in\mathbb{N}}$. Étant donné un seuil d'erreur $\epsilon>0$, l'algorithme s'arrête lorsque $|F(z_k)|<\epsilon$ et renvoie z_k.

[39]:
```
def dichotomie(F,a0,b0,eps):
    a=a0
    b=b0
    c=0
    e=2*eps
    while e>eps:
        c=(a+b)/2
        if F(c)*F(a)>=0:
            a=c
        else:
            b=c
        e=abs(F(c))
```

```
    return(c)
```

On teste la fonction `dichotomie(F,a0,b0,eps)` ci-dessous en déterminant la solution de $\sin(x) = \frac{1}{2}$ sur $[-1.5, 1.5]$.

[42]:
```
def F3(x):
    return(np.sin(x)-1/2)
dichotomie(F3,-1.5,1.5,0.0001)
```

[42]:
```
0.523681640625
```

1.2.2 La routine de scipy

La fonction `scipy.optimize.bisect(f,a,b)` implemente la methode de la dichotomie. On la teste ci-dessous avec l'exemple précédent.

[54]:
```
sc_opt.bisect(F3,-1.5,1.5)
```

[54]:
```
0.5235987755982023
```

1.3 Méthode du point fixe de Banach

1.3.1 La méthode

On cherche à résoudre un système non-linéaire écrit sous la forme

$$(1) \qquad F(x) = x$$

où F est une application définie sur un sous-ensemble fermé de $\mathbb{R}^m$, à valeurs dans $\mathbb{R}^m$. Si F est une contraction sur cet ensemble, alors le théorème de Banach-Picard assure qu'elle y admet un unique point fixe x^*, qui est donc solution de (1). De plus, pour n'importe quel x_0, on a $\lim_{n\to\infty} F^n(x_0) = x^*$. L'algorithme consiste donc à choisir un point x_0 arbitrairement, puis de calculer $x_1 = F(x_0)$, $x_2 = F(x_1)$ et ainsi de suite. Cela définit une suite suite $(x_n)_{n\in\mathbb{N}}$. Il se peut que l'algorithme ne converge pas, si x_0 n'appartient pas à un ensemble fermé où F est une contraction. Étant donnés un seuil d'erreur $\epsilon > 0$ et un nombre maximal d'itérations $N \in \mathbb{N}$, l'algorithme s'arrête soit lorsque $|F(x_n) - x_n| < \epsilon$ auquel cas il renvoie x_n, soit lorsque le nombre d'itérations a dépassé N et il renvoie le message "L'algorithme n'a pas convergé".

[89]:
```
def Banach(f,x0,eps,N):
    e=2*eps
    x=x0
    n=0
    while n<N and e>eps:
        x=f(x)
        n=n+1
        e=np.linalg.norm(x-f(x))
    if n<N:
```

```
        return(x)
    else:
        return("l'algorithme n'a pas converge")
```

On illustre `Banach(f,x0,eps,N)` pour trouver le point fixe de la fonction $f(x,y) = (\arctan(y), \frac{\cos(x)}{2})$

[93]:
```
def F4(x):
    return(np.array([np.arctan(x[1]),np.cos(x[0])/2]))
Banach(F4,np.array([0,0]),0.001,10)
```

[93]:
```
array([0.42688717, 0.45485883])
```

1.3.2 La routine de scipy pour les problèmes de point fixe

La fonction `scipy.optimize.fixed_point(f,x0)` calcule numériquement la solution du problème $x = f(x)$ en débutant à x_0. Elle implémente une autre méthode que celle du point fixe de Banach, plus robuste et qu'on ne détaillera pas. On l'illustre en reprenant l'exemple précédent.

[95]:
```
sc_opt.fixed_point(F4,np.array([0,0]))
```

[95]:
```
array([0.42707859, 0.45508986])
```

1.4 Méthode de Newton-Raphson

On cherche à résoudre

$$F(x) = 0,$$

où F est une fonction différentiable sur $\mathbb{R}^m$. Étant donné un point $x_0 \in \mathbb{R}^m$, on approche F par la fonction affine qui correspond à sa linéarisation en x_0:

$$x \mapsto F(x_0) + JF(x_0)(x - x_0)$$

où JF est la matrice Jacobienne de F, i.e. celle de sa différentielle. Cette fonction s'annule au point $x_1 = x_0 + h$ où h est solution de

$$JF(x_0)h = -F(x_0).$$

On détermine la solution h de cette équation non pas en inversant $JF(x_0)$ mais en utilisant une méthode de résolution de ce système linéaire plus rapide, par exemple la méthode du pivot.

On peut alors répéter l'opération en x_1, ce qui va définir un point x_2, et ainsi de suite. Cela définit une suite $(x_n)_{n\in\mathbb{N}}$. Il se peut que cette suite ne converge pas, par exemple si le point d'initialisation x_0 est trop loin du vrai zéro x^* de F. Étant donnés un seuil $\epsilon > 0$ et un nombre maximum d'itérations $N \in \mathbb{N}$, l'algorithme s'arrête lorsque $|F(x_n)| < \epsilon$ et renvoie x_n, ou bien lorsque le nombre d'itérations dépasse N auquel cas il renvoie le message "l'algorithme n'a pas convergé".

[4]:
```
import numpy as np
def Newton_Raphson(F,JF,x0,eps,N):
    e=2*eps
    n=0
    x=x0
    while e>eps and n<N:
        x=x-np.linalg.solve(JF(x),F(x))
        e=np.linalg.norm(F(x))
        n=n+1
    if n<N:
        return(x)
    else:
        return("l'alogrithme n'a pas converge")
```

Ci-dessous on teste `Newton_Raphson(F,JF,eps,N)` pour déterminer numériquement la solution (x, y) de

$$\begin{cases} \cos(x) = \sin(y), \\ e^{-x} = \cos(y) \end{cases}$$

[24]:
```
def F1(x):
    return(np.array([np.cos(x[0])-np.sin(x[1]),np.exp(-x[0])-np.cos(x[1])]))
def JF1(x):
    A=np.zeros([2,2])
    A[0,0]=-np.sin(x[0])
    A[1,0]=-np.exp(x[0])
    A[0,1]=-np.cos(x[1])
    A[1,1]=np.sin(x[1])
    return(A)
Newton_Raphson(F1,JF1,np.array([0,0]),0.0001,100)
```

[24]:
```
array([0.5884784 , 0.98231792])
```

1.4.1 Les routines de scipy

Si f est à valeurs réelles, on a déjà mentionné que `scipy.optimize.newton(f,x0)` implémente la méthode de la sécante. Si on donne en plus en argument la dérivée, alors `scipy.optimize.newton(f,x0,f')` implémente la méthode de Newton-Raphson. On reprend l'exemple précédent de la solution de $x^2 - 2 = 0$ sur l'intervalle $[1, 2]$.

[57]:
```
def F2prime(x):
    return(2*x)
sc_opt.newton(F2,1.5,F2prime)
```

[57]:
```
1.4142135623730951
```

Si f est à valeur vectorielle, alors il vaut mieux utiliser `scipy.optimize.root(f,x0)` qui, par défaut (voir `help(scipy.optimize.root)` pour toutes les possibilités de cette routine), cherche

un zéro de f par une méthode hybride qui, grosso modo, est un analogue multi-dimensionel des méthodes de la sécante et de dichotomie. C'est la routine la plus robuste, et on peut traiter tous les exemples précédents avec son aide.

[60]:
```
sc_opt.root(F2,1.5)
```

[60]:
```
    fjac: array([[-1.]])
     fun: array([4.4408921e-16])
 message: 'The solution converged.'
    nfev: 7
     qtf: array([-4.5106141e-12])
       r: array([-2.82842719])
  status: 1
 success: True
       x: array([1.41421356])
```

[62]:
```
sc_opt.root(F1,np.array([0,0]))
```

[62]:
```
    fjac: array([[ 0.73246374,  0.68080604],
       [-0.68080604,  0.73246374]])
     fun: array([-1.09912079e-13,  1.58317803e-13])
 message: 'The solution converged.'
    nfev: 13
     qtf: array([4.43898629e-11, 3.09942844e-10])
       r: array([-0.78045973,  0.15963468,  0.9871113 ])
  status: 1
 success: True
       x: array([0.58853274, 0.98226358])
```

[63]:
```
sc_opt.root(F3,0)
```

[63]:
```
    fjac: array([[-1.]])
     fun: array([-5.55111512e-17])
 message: 'The solution converged.'
    nfev: 8
     qtf: array([3.13638004e-14])
       r: array([-0.86602541])
  status: 1
 success: True
       x: array([0.52359878])
```

1.5 Exercices

1.5.1 Exercice 1

On rappelle d'abord la méthode de la fausse position pour résoudre $F(x) = 0$. Si F est une fonction continue sur $\mathbb{R}$, et $x_0 < y_0$ sont tels que $F(x_0) < 0 < F(y_0)$, alors F admet un zéro sur l'intervalle

(x_0, y_0). On peut approximer F sur cet intervalle par la fonction affine :

$$x \mapsto F(x_0) + \frac{x - x_0}{y_0 - x_0}(F(y_0) - F(x_0))$$

Cette fonction s'annule en $z_0 \in (x_0, y_0)$ donné par

$$z_0 = \frac{x_0 F(y_0) - F(x_0) y_0}{F(y_0) - F(x_0)}.$$

Dans le premier cas où $F(z_0) < 0$, alors F admet un zéro sur (x_0, z_0). Dans le second cas où $F(z_0) > 0$, alors F admet un zéro sur (z_0, y_0).

Dans le premier cas, on définit $x_1 = x_0$ et $y_1 = z_0$. Dans le second cas, on définit $x_1 = z_0$ et $y_1 = y_0$.

On construit alors x_2 et y_2 en fonction de x_1 et y_1 par le même raisonnement, et ainsi de suite. Cela définit deux suites $(x_n)_{n\in\mathbb{N}}$ et $(y_n)_{n\in\mathbb{N}}$.

Étant donné un seuil d'erreur $\epsilon > 0$, l'algorithme s'arrête dès que $|F(x_k)| < \epsilon$ ou bien $|F(y_k)| < \epsilon$, et renvoie x_k ou y_k respectivement.

1. Écrire une fonction `fausse-pos(F,x_0,x_1,epsilon)` qui prend en entrée une fonction F, deux réels $x_0 < y_0$ et un seuil d'erreur $\epsilon > 0$, et renvoie le résultat de la méthode de la fausse position décrite ci-dessus (en supposant donc $F(x_0) < 0 < F(y_0)$ pour cette question)

2. Tester votre fonction `fauss-pos` pour la fonction $F(x) = e^x - 2$.

3. Adapter la méthode décrite ci-dessus au cas où l'on peut soit avoir $F(x_0) < 0 < F(y_0)$, soit avoir $F(x_0) > 0 > F(y_0)$. Écrire la fonction `fausse-pos2(F,x_0,x_1,epsilon)` correspondante.

4. Tester votre fonction `fauss-pos2` pour la fonction $F(x) = x^3 - 3x + 2$.

1.5.2 Exercice 2

1. Trouver numériquement la solution (x^*, y^*, z^*) de

$$\begin{cases} -x - 7y + z + xy = 1, \\ 2x + 4y + 2z + yz = \frac{1}{2}, \\ 2x + 6y + xz^2 = -1, \end{cases}$$

par les trois manières suivantes : la méthode de Newton-Raphson implémentée par la fonction `Newton_Raphson` précédente, la routine `sc_opt.root` sans préciser la différentielle, et la routine `sc_opt.root` en précisant la différentielle.

2. Choisir (x_0, y_0, z_0) proche, mais non égal, de (x^*, y^*, z^*), et essayer de résoudre le système en appliquant la méthode de Banach-Picard implémentée par la fonction `Banach` avec point initial (x_0, y_0, z_0). L'algorithme converge-t-il ?

3. Calculer numériquement le rayon spectral de la différentielle du système au point (x^*, y^*, z^*). Proposer une explication théorique pour votre réponse à la question 2.

Exercice 3

On cherche à implémenter une nouvelle méthode, exacte, pour trouver les décimales d'une solution d'une équation polynomiale à coefficient entiers, sans se limiter à la précision des flottants, ni en utilisant des bibliothèques spécifiques pour améliorer la précision des calculs. On va utiliser que Python manipule les entiers de manière exacte. Précisément, on cherche à résoudre

$$P(x) = 0$$

où $P(x) = a_0 + a_1 x + ... + a_n x^n$ avec $(a_0, ..., a_n) \in \mathbb{Z}^{n+1}$.

On suppose que l'on connaît $u_0 \in \mathbb{Z}$ tel que $P(u_0) \leq 0$ et $P(u_0 + 1) > 0$. Alors P admet une racine x^* dans $[u_0, u_0 + 1)$. On remarque que

$$\begin{aligned} P(x^*) = 0 &\Leftrightarrow a_0 + a_1 x^* + ... + a_{n-1} x^{*n-1} + a_n x^{*n} = 0 \\ &\Leftrightarrow 10^n a_0 + 10^{n-1} a_1 (10x^*) + ... + 10 a_{n-1} (10x^*)^{n-1} + a_n (10x^*)^n = 0 \\ &\Leftrightarrow P_1(10x^*) = 0, \end{aligned}$$

c'est-à-dire que $10x^*$ est une racine du polynôme $P_1(x) = 10^n a_0 + 10^{n-1} a_1 x + ... + 10 a_{n-1} x^{n-1} + a_n x^n$. Le polynôme P_1 admet donc une racine dans $[10u_0, 10u_0 + 10)$. On détermine alors k_1 un entier entre 0 et 9 tel que $P_1(10u_0 + k_1) \leq 0$ et $P_1(10u_0 + k_1 + 1) > 0$. On définit ensuite $u_1 = 10u_0 + k_1$.

On a que $u_1 \in \mathbb{Z}$ est tel que $P(u_1) \leq 0$ et $P(u_1 + 1) > 0$. On itère la procédure pour calculer de manière analogue P_2 et u_2, et ainsi de suite. Cela définit une suite de polynômes $(P_i)_{i \in \mathbb{N}}$ et d'entiers relatifs $(u_i)_{i \in \mathbb{N}}$.

1. Montrer à la main que la suite $10^{-i} u_i$ admet une limite x^* lorsque $i \to \infty$ qui est solution de $P(x^*) = 0$.

2. Utiliser cet algorithme pour trouver les 100 premières décimales de racine de deux.

[ ]: