L2-S3 : UE Méthodes numériques

SEANCE 7

Résolution d'équations, Interpolation

18 novembre 2024

Recherche des zéros d'une fonction et équations non linéaires

L'outil numérique peut être très utile pour **résoudre des équations non linéaires**. Dans toute la suite on s'intéressera à l'équation $x^2 = x + 1$.

Résoudre cette équation revient à en rechercher les **racines**, et par conséquent les zéros de la fonction $f(x) = x^2 - x - 1$.

Notons tout de suite que la racine positive de cette équation est le **nombre d'or** $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803398875...$. C'est un nombre irrationnel et rechercher la racine positive de $f(x)$ revient donc à déterminer une **approximation numérique** de ce nombre particulier.

Recherche des zéros d'une fonction et équations non linéaires

Le nombre d'or est compris entre 0 et 2. Posons ainsi le problème :

```
1  import numpy as np
2
3  def f(x):
4      return x**2 - x - 1
5
6  xmin = 0   # intervalle
7  xmax = 2
8  N = 101   # nombre de points (100 intervalles)
9
10 xs = np.linspace(xmin,xmax,N)
```

Recherche du minimum et du maximum d'une liste

Comment trouver le minimum absolu ou le maximum absolu d'une fonction $f(x)$? Un algorithme simple consiste d'abord à **discrétiser** la fonction sur un intervalle $x \in [x_{min}, x_{max}]$ sur **N points**.

Algorithme simple de recherche d'un minimum :

```
test = xs[0]
minimum = xs[0]
for x in xs:
    if f(x) < test :
        test = f(x)
        minimum = x
print('min(f(x))=', test, 'en x =', minimum)
```

A l'aide des fonctions de numpy :

```
minimum = xs[np.argmin(f(xs))]
print('min(f(x))=', np.min(f(xs)), 'en x =', minimum)
```

Recherche du minimum et du maximum d'une liste

Algorithme simple de recherche d'un maximum :

```
test = f(xs[0])
maximum = xs[0]
for x in xs:
    if f(x) > test :
        test = f(x)
        maximum = x
print('min(f(x))=', test, 'en x =', maximum)
```

A l'aide des fonctions de numpy :

```
minimum = xs[np.argmax(f(xs))]
print('min(f(x))=', np.max(f(xs)), 'en x =', minimum)
```

Recherche du minimum et du maximum d'une liste

Si on l'applique à la **recherche du zéro d'une fonction**, alors il faut rechercher le minimum de la **fonction** $|f(x)|$:

```
for N in 10**np.arange(2, 8, 1)+1:   # 10**N intervalles
  xs = np.linspace(xmin, xmax, N)
  index = np.argmin(abs(f(xs)))
  print('Zero en', xs[index], 'avec ', N-1,' intervalles)
print('  comparaison 1.61803398875')
```

```
Out[0]:     Zero en 1.62 avec 100 intervalles
            Zero en 1.618 avec 1000 intervalles
            Zero en 1.618 avec 10000 intervalles
            Zero en 1.6180400000000001 avec 100000
            Zero en 1.618034 avec 1000000 intervalles
            Zero en 1.618034 avec 10000000 intervalles
                        comparaison 1.61803398875
```

On voit donc que cet algorithme nécessite un nombre très important d'évaluations de la fonction dès lors que l'on veut une **précision** correcte sur son minimum. C'est donc un algorithme **fiable** sur le principe, mais **très couteux en temps de calcul**.

Méthode par dichotomie

Il s'agit d'une recherche dans un intervalle que l'on découpe en sous-intervalles.

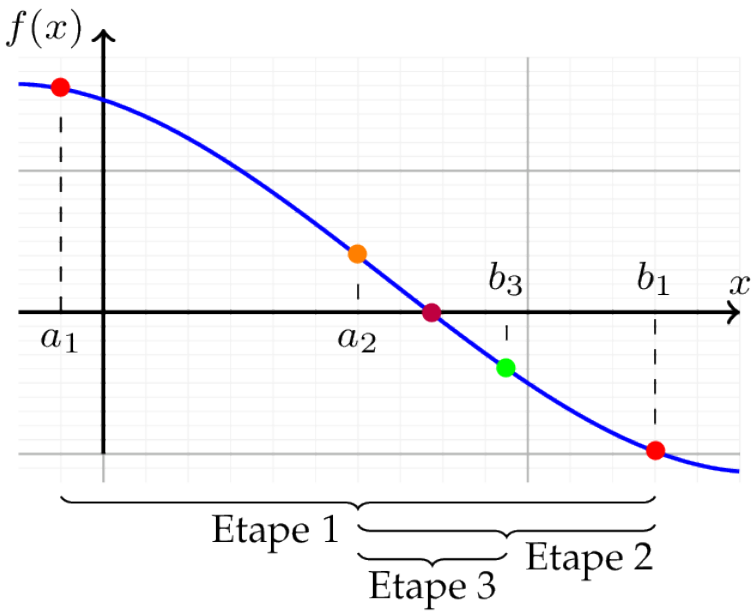
Admettons que sur l'intervalle $[a, b]$ la fonction $f(x)$ ait un zéro et un seul : elle **change de signe** une seule fois sur l'intervalle et le produit $f(a)f(b)$ est donc négatif.

Coupons alors l'intervalle $[a, b]$ en deux, ce qui nous donne le point $x = (a + b)/2$:

- si le produit $f(a)f(x)$ est négatif, c'est que le zéro recherché se trouve dans l'intervalle $[a, x]$
- sinon il se trouve dans l'intervalle $[x, b]$.

Il reste à rediviser en deux l'intervalle où se trouve le zéro et à refaire la même recherche. On répète encore la même opération plusieurs fois jusqu'à ce que la précision requise soit atteinte. C'est un processus **itératif**.

Méthode par dichotomie



En écriture algorithmique, cela donne :

Tant que (b - a) $> \epsilon$

    Calcul m = (a + b) / 2
    Calcul f(m)
    Si ((f(a)*f(m)) > 0) **alors**
        m remplace a
    sinon
        m remplace b

Afficher m

avec ϵ la précision souhaitée.

Méthode par dichotomie

Avantage :
La méthode de la dichotomie est assez pédestre, elle est sûre, sans surprise, et le niveau de précision est contrôlable par le paramètre ϵ.

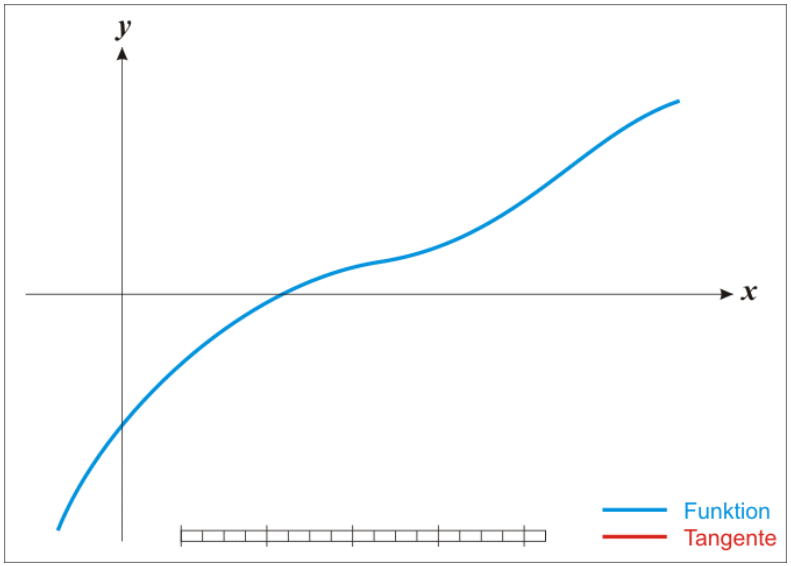
Inconvénients :
Elle nécessite cependant une connaissance préalable de la fonction puisqu'elle suppose qu'il y ait un et un seul zéro dans l'intervalle de recherche : il faut donc pouvoir choisir celui-ci judicieusement.

Méthode de Newton

La méthode de Newton est à la fois **plus efficace et plus dangereuse**. Il s'agit, à partir d'un point x_0 judicieusement choisi, d'approcher la fonction par sa tangente et de chercher l'intersection de celle-ci avec l'axe des x.

Cet algorithme est plus efficace que la méthode de la dichotomie car il bénéficie d'une **information supplémentaire** pour rechercher le zéro (la dérivée première de la fonction), mais il peut **ne pas converger dans certains cas** (si l'intersection de la tangente ne tombe pas dans le domaine de définition de la fonction, par exemple pour $\sqrt{x}$).

Méthode de Newton



Partons d'une valeur x_0. La tangente de f au point $(x_0, f(x_0))$ est :

$$t(x) = f(x_0) + (x - x_0)f'(x_0)$$

Son intersection avec l'axe des x est x_1 tel que $t(x_1) = 0$: $x_1 = x_0 - f(x_0)/f'(x_0)$

Répétons l'opération en calculant la tangente au point x_1 et son intersection x_2 avec l'axe des x, etc :

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

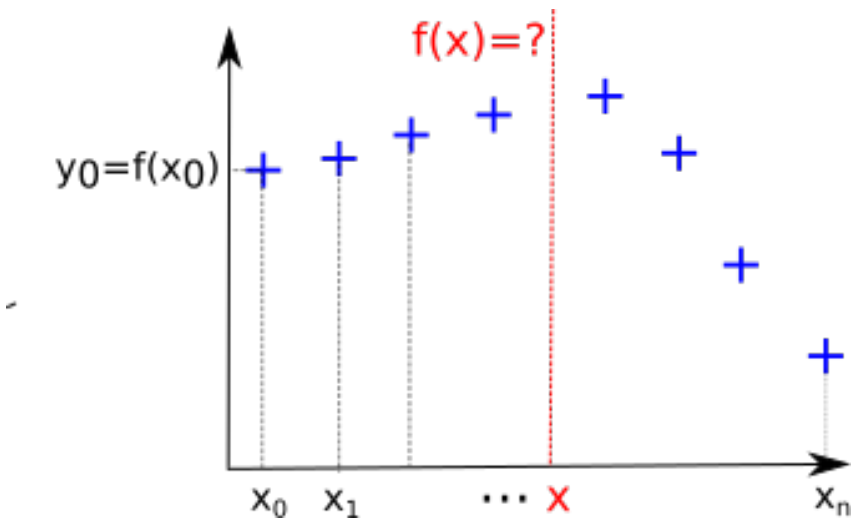
jusqu'à ce que $|x_{n+1} - x_n| < \epsilon$.

Cet algorithme peut ne pas fonctionner s'il tombe sur un point dont la dérivée première est nulle : il est important de partir d'un point x_0 **proche du zéro recherché**.

Retour sur l'interpolation

Position du problème • Soit f une fonction à valeurs réelles définie sur un intervalle $[a, b]$. On suppose que l'on ne connaît que les valeurs $y_i = f(x_i)$ qu'elle prend en n points donnés $x_0, x_1, ..., x_{n-1} \in [a, b]$.

• Comment donner une valeur approchée de $y = f(x)$ pour $x \in [a, b]$ quelconque, de la manière la plus précise et la plus rapide possible ?



Retour sur l'interpolation

Applications

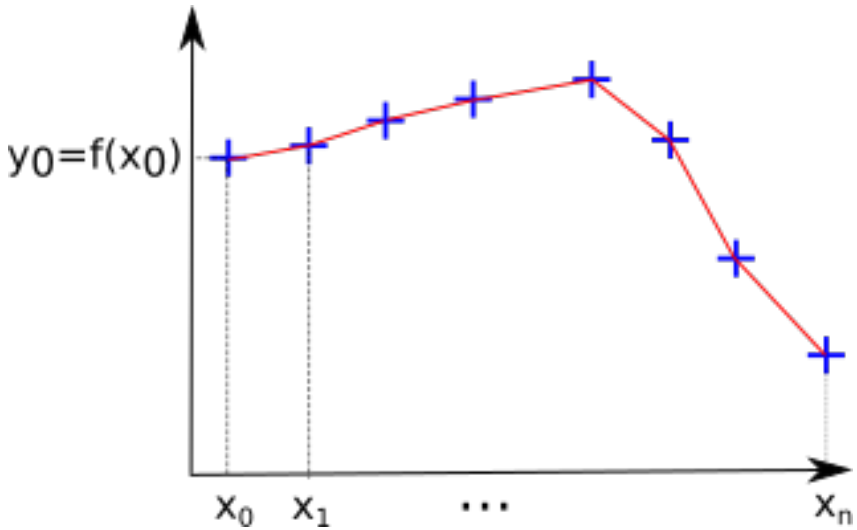
- Interpolation de données expérimentales ou de données tabulées
- Interpolation de résultats de calculs obtenus sur une grille de points, lors de la résolution d'une équation aux dérivées partielles
- Approximation d'une fonction ■ coûteuse ■ à calculer par une fonction plus simple à calculer et dont on peut trouver aisément la dérivée, une primitive... (typiquement, des polynômes)
- Affichage, graphisme, CAO (courbes de Bezier), montagnes russes...

Exemple simple : l'interpolation linéaire par morceaux

Cette méthode consiste à approcher la fonction à interpoler par des segments de droite reliant les points (x_i, y_i) consécutifs. La **fonction interpolante** $\Phi(x)$ est une **fonction affine par morceaux** :

$$\Phi_{\text{lin}}(x) = \{\Phi_j(x),\ x \in [x_j, x_{j+1}]\}$$

$$\Phi_j(x) = f(x_j) + \frac{f(x_{j+1}) - f(x_j)}{(x_{j+1} - x_j)}(x - x_j)$$



Exemple simple : l'interpolation linéaire par morceaux

Avantages :

- intuitif ;
- fonction interpolante très simple.

Inconvénients :

- il faut au préalable déterminer dans quel intervalle se trouve x (algorithme de recherche) ;
- l'écart à $f(x)$ peut être élevé si les points x_i sont distants ;
- la fonction Φ_{lin} n'est pas dérivable au niveau des points x_i.

Exemple simple : l'interpolation linéaire par morceaux

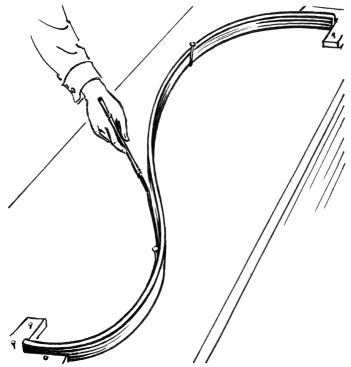
Fonction interp de la librairie Numpy : y = np.interp(x, xp, yp)

- x : tableau des valeurs pour lesquelles on veut déterminer f(x) par interpolation linéaire
- xp : tableau des valeurs de x pour lesquelles on connaît la valeur de la fonction f(x)
- yp : tableau des valeurs (connues) des f(xp)
- y : tableau des valeurs de f(x) calculées par interpolation linéaire

Attention : les valeurs dans le tableau xp doivent être strictement croissantes. Si ce n'est pas le cas, la fonction *interp* renvoie un résultat faux, mais aucun message d'erreur.

Interpolation par des splines

On peut obtenir une interpolation plus précise en utilisant des polynômes plutôt que des droites pour interpoler entre deux points. Ainsi, la fonction interpolante peut être non seulement continue, mais ses dérivées première et seconde peuvent aussi être continues : la fonction interpolante est ainsi beaucoup plus régulière.
Cette technique était bien connue des dessinateurs industriels, qui devaient concevoir des objets (par exemple pièces de carrosserie de voitures) à la fois solides (les discontinités sont en général des zones de faiblesse) et agréables à l'oeil.



Une baguette de bois souple (en anglais "spline") est contrainte en position et en angle à chaque extrémité (pour obtenir la continuité et la continuité des dérivées première et seconde), et en position en un ou plusieurs points intermédiaires.

Interpolation par des splines

Fonctions splrep et splev de la librairie scipy.interpolate

```
1 from scipy import interpolate
2
3 tck = interpolate.splrep(xp, yp)
4 y = interpolate.splev(x, tck)
```

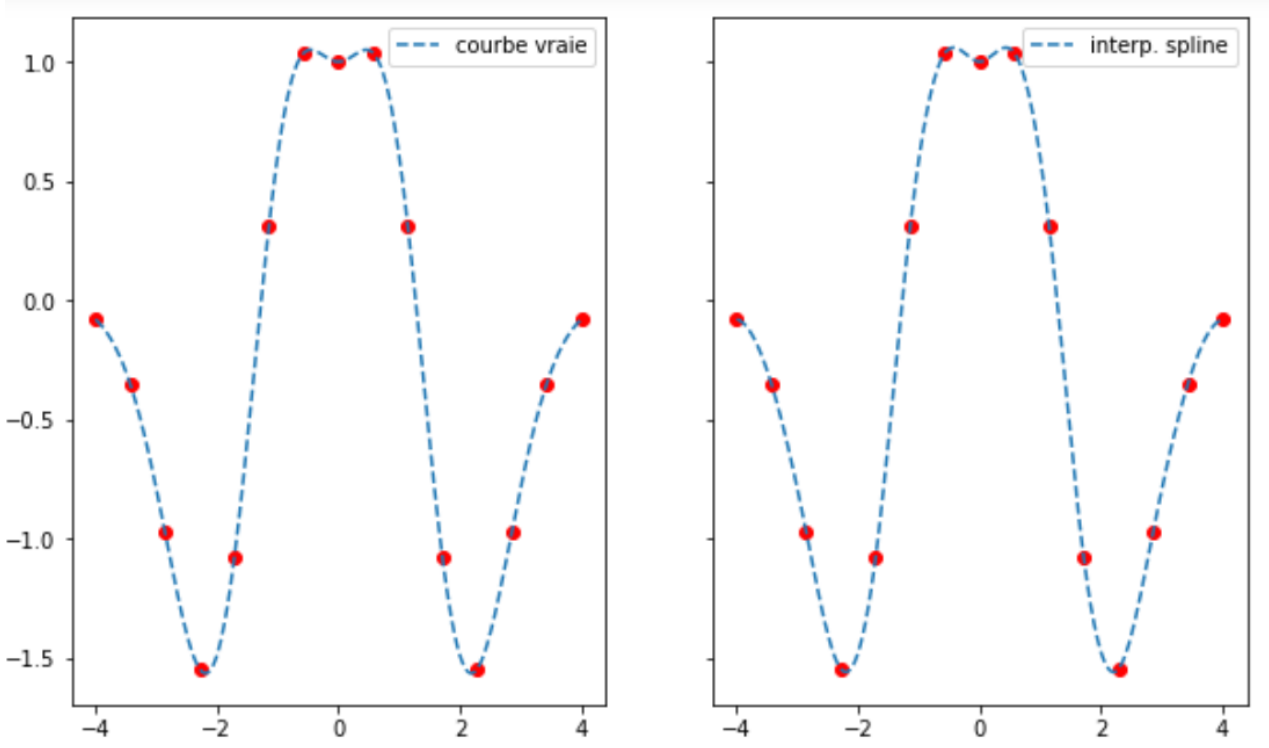
- x : tableau des valeurs pour lesquelles on veut déterminer f(x) par interpolation
- xp : tableau des valeurs de x pour lesquelles on connaît la valeur de la fonction f(x)
- yp : tableau des valeurs (connues) des f(xp)
- tck : tuple contenant toute l'information sur les polynômes interpolants par morceaux
- y : tableau des valeurs de f(x) calculées par interpolation

Interpolation de la fonction $f(x) = (1 + x^2 - x^4)e^{-x^2/2}$

xp : 15 valeurs entre -4 et $+4$

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from scipy import interpolate
4
5  def f(x):
6      return (1+x**2-x**4)*np.exp(-x**2/2)
7
8  xp=np.linspace(-4,4,15)   # points connus
9  yp=f(xp)
10 x=np.linspace(-4,4,1000)   # representation de la fonction
11 y=f(x)
12 tck = interpolate.splrep(xp, yp)   # interp. splines
13 y_spline = interpolate.splev(x, tck)
14
15 fig, axs = plt.subplots(1, 2, figsize=(10,6), sharey=True)
16 im = axs[0].scatter(xp, yp, color='r')
17 im = axs[0].plot(x,y,linestyle = '--', label='courbe vraie')
18 axs[0].legend()
19 im = axs[1].scatter(xp, yp, color='r')
20 im = axs[1].plot(x,y_spline,linestyle = '--', label='interp.
       spline')
21 axs[1].legend()
22 plt.show()
```

Interpolation de la fonction $f(x) = (1 + x^2 - x^4)e^{-x^2/2}$



Mieux que les splines : les courbes de Bézier

Extension aux fonctions de deux variables (surfaces)

- **Pierre Bézier** (1910-1999), a développé la CFAO chez Renault. A inventé les courbes de Bézier en 1962. A enseigné à l'IUT de Cachan de 1968 à 1979.
- **Paul de Casteljau** (1930-2022), a développé la CFAO chez Citroën. A inventé les courbes de Bézier en 1959 (couvert par le secret industriel chez Citroën)

La création d'images de synthèse repose en grande partie sur la capacité à représenter des surfaces qui passent par un nombre de points prédéterminés par le calcul ou par l'observation (points caractéristiques sur un visage ou sur un corps).