TP7_sujet

March 5, 2024

1 Recherche des racines d'équations non linéaires

Nom :
Prénom :

On commence par importer les bibliothèques qui vont bien :

[3]:
```
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import display
```

1.1 Recherche incrémentale

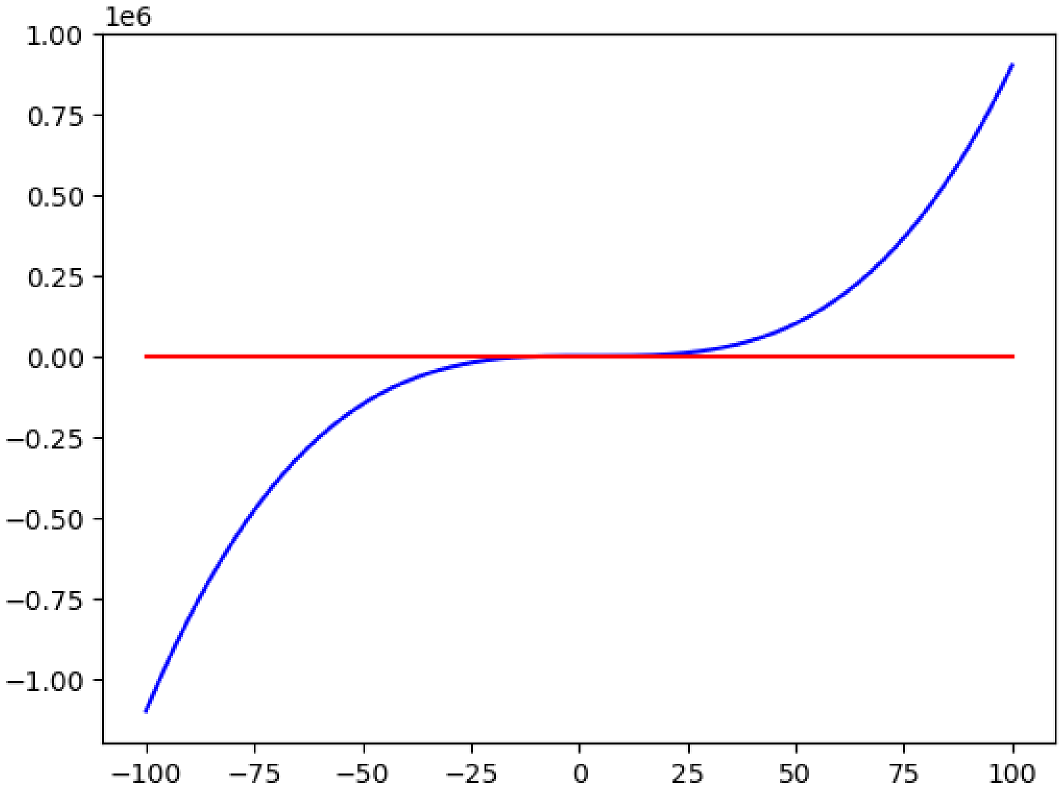
Dans cet exercice, nous allons nous intéresser à la résolution "approchée" de l'équation $f(x) = x^3 - 10x^2 + 5 = 0$

- Ecrire la fonction `f(x)` (fonction python classique) qui correspond à la fonction ci-dessus.

Tracer le graphe de la fonction f sur l'intervalle $[-100, 100]$ (on fera figurer l'axe en rouge ci-dessous).

[5]:
```
# A écrire
```

[5]:
```
[<matplotlib.lines.Line2D at 0x10c5d4640>]
```



- A l'aide d'une boucle que vous devez écire, rechercher les racines de $f(x)$ sur l'intervalle [-100,100] (c'est à dire les valeurs t telles que f(t)=0) en utilisant une méthode de recherche incrémentale, en explorant l'intervalle par pas de 1e-2. Pour réaliser cela,on écrira :
  - Une fonction qui prend en argument les bornes de l'intervalle et le pas, et qui renvoie un tableau numpy à deux dimensions: on aura 2 lignes de N colonnes (s'il y a N racines) : pour chaque colonne il y aura dans la première ligne la borne inférieure et pour la deuxième ligne la borne supérieure pour chacun des N intervalles.
  - Une boucle qui parcourrera l'intervalle de travail de pas en pas et testera si $f(x) \times f(x + pas) < 0$
- Mesurer le temps de calcul (pour cela, précéder l'appel de la commande `%timeit` : ce qui donnera l'appel `%timeit recherche(-100,100,1e-2)`). Attention du coup le calcul n'est pas instantané car plusieurs boucles sont faites! (vous testerez d'abord votre fonction et ensuite vous ajoutez le `%timeit`.

[12]:
```
def recherche(xmin,xmax,pas):
    # A compléter ....

# A compléter ....
# A compléter ....
```

```
8.16 ms ± 20.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
array([[-0.69,  0.73,  9.94],
```

```
       [-0.68,  0.74,  9.95]])
```

- Ecrire ensuite une fonction qui réalise la même opération mais sans boucle, en se basant sur les tableaux de numpy (utiliser arange, where.....) Mesurer le temps et comparer.

[11]:
```
def recherche_vec(xmin,xmax,pas):
    # A compléter ....

# A compléter ....
# A compléter ....
%timeit recherche_vec(-100,100,1e-2)
racines = recherche_vec(-100,100,1e-2)
print(racines)
```

```
504 µs ± 547 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[11]:
```
array([[-0.69,  0.73,  9.94],
       [-0.68,  0.74,  9.95]])
```

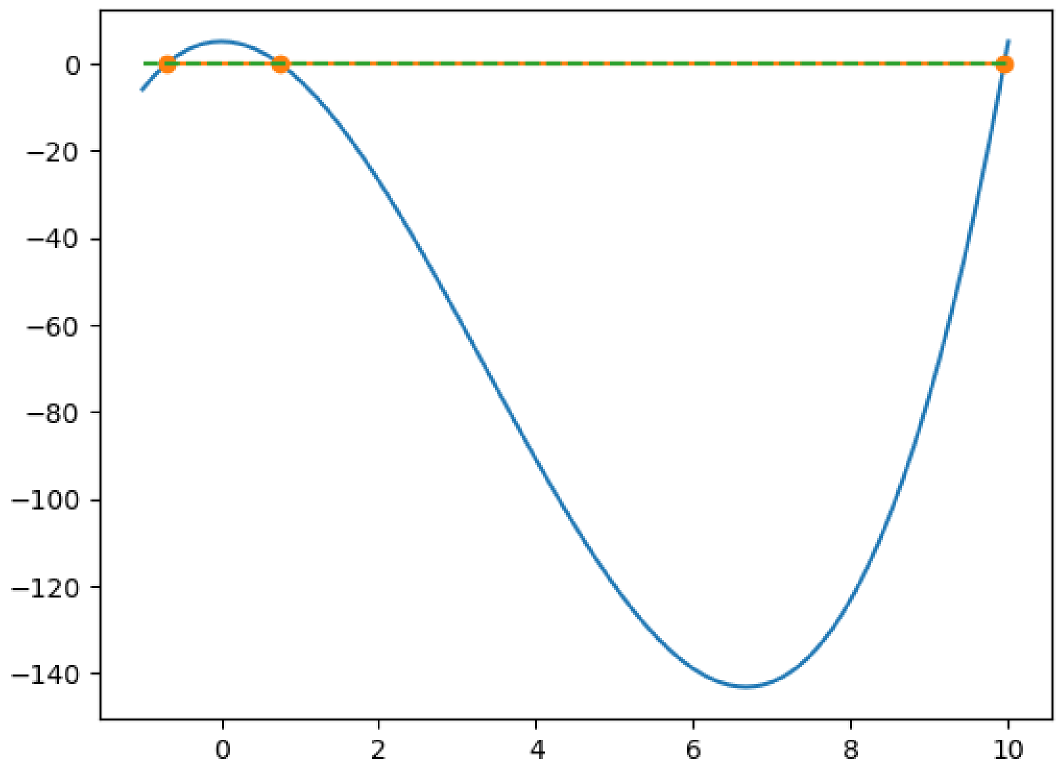
- Refaire le graphique de la première question en limitant le dessin aux valeurs de x dans $[-1, 10]$ (on changera le linspace).
- Ajouter sur le schéma les milieux des intervalles obtenus avec les recherches ci-dessus.

[13]:
```
# Milieux des intervalles
# a compléter ...

# Graphique
# à compléter ...
```

```
[-0.685  0.735  9.945]
```

[13]:
```
[<matplotlib.lines.Line2D at 0x10c6bc820>]
```

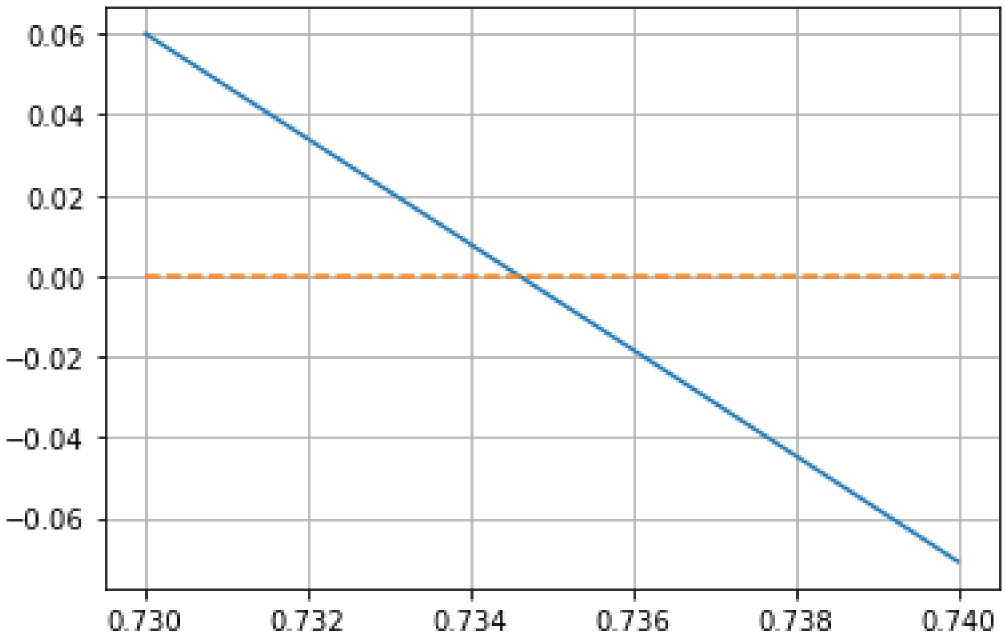


1.2 Recherche par dichotomie (ou bissection)

- Ecrire une fonction de recherche d'approximation de racine par dichotomie.Vous devez ici écrire une boucle. Cette fonction affinera la recherche d'une racine, à partir d'un intervalle qui ne contient qu'une unique racine. La fonction prendra comme argument les bornes inférieures et supérieures de l'intervalle à explorer, ainsi qu'un critère d'arrêt `xtol` qui représente la précision souhaitée (largeur de l'intervalle final).
- Nous prendrons comme exemple la fonction précédente ($f(x) = x^3 - 10x^2 + 5 = 0$) et nous intéresserons à la racine qui se trouve dans l'intervalle $[0.73, 0.74]$.

Commencer par tracer le graphe de la fonction sur cet intervalle puis calculer la racine par dichotomie. On évaluera le temps d'execution pour cette recherche.

[4]:
```
# Afficher le graphique ci-dessous
# à compléter ...
```



[14]:
```
def dicho(xmin,xmax,xtol):
    #  A compléter ...

%timeit  dicho(0.73, 0.74, 1e-7)
intervalle = dicho(0.73, 0.74, 1e-7)
 #  A compléter ...
sol= ???
print('intervalle :',intervalle)
print('solution : ',sol)
```

```
15.9 µs ± 28.9 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
intervalle : [0.7346035  0.73460358]
solution :  0.7346035385131837
```

- Effectuer la même recherche de racine,sans boucle, mais en utilisant la librairie `scipy` (fonction `scipy.optimize.bisect`).
- On calculera le temps d'exécution de cette méthode et on comparera avec la méthode précédente.

[15]:
```
import scipy as sp
import scipy.optimize

#  A compléter ...
#  A compléter ...
print('solution : ',sol)
```

```
15.8 µs ± 57.7 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
solution :  0.7346035003662108
```

1.3 Méthode de recherche de Newton-Raphson

- Comme précédement, on implémentera (en écrivant la boucle) la méthode de Newton-Raphson (voir cours) puis on utilisera celle de scipy en utilisant la librairie `scipy` (fonction `scipy.optimize.newton`).

Le critère d'arret sera exprimé sur les valeurs de f(x), avec $|f(x)| < tol$.

- Comparer les temps d'éxécution.

[17]:
```
#  A compléter ...

#  A compléter ...

def newton( x, tol=0.000001):
    #  A compléter ...
    #  A compléter ...
#  A compléter ...
sol = newton((0.73+ 0.74)/2,1e-10)
print('solution : ',sol)
```

```
1.27 µs ± 3.26 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
solution :  0.7346035077893033
```

[18]:
```
#  A compléter ...

#  A compléter ...
```

```
45.7 µs ± 195 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
solution :  0.7346035077893032
```

1.4 Combinaison de méthodes

- Combiner la recherche itérative avec la recherche de Newton-Raphson pour trouver précisément les 3 racines de la fonction f.

[18]:
```
import scipy as sp
import scipy.optimize
# Afficher Racines de la recherche itérative
# A compléter ...
# Afficher Racines par scipy.optimize.newton
# A compléter ...
```

```
Sols :  [-0.685  0.735  9.945]
Sols : [-0.6840945657036899, 0.7346035077893033, 9.949491057914384]
```