TP4_sujet

March 2, 2022

1 Recherche des racines d'équations non linéaires

Nom : Chahir
Prénom : Youssef

On commence par importer les bibliothèques qui vont bien :

[4]:
```
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import display
```

1.1 Recherche incrémentale

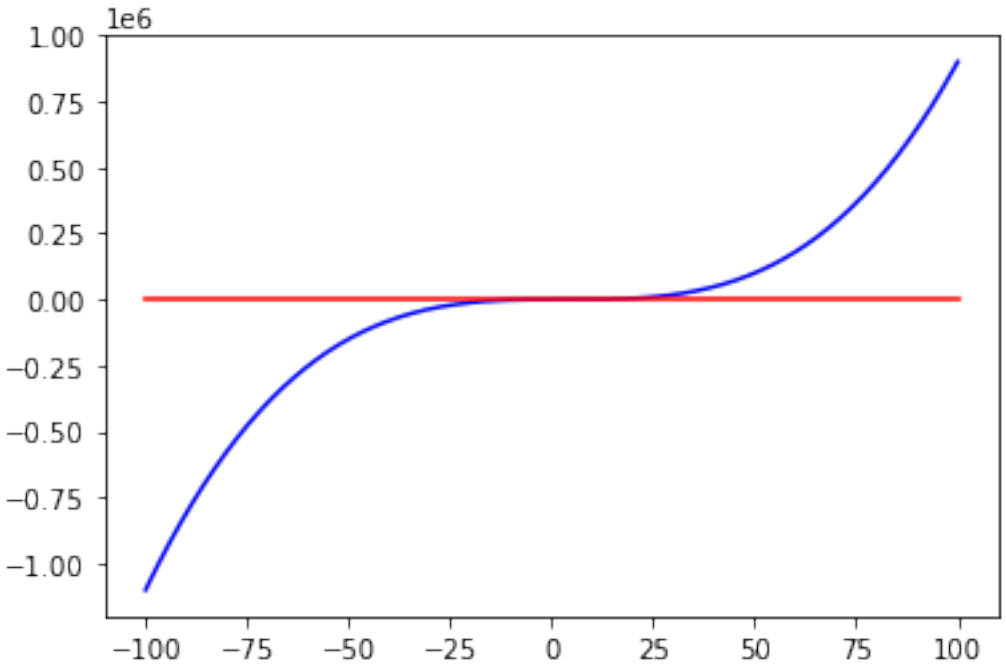
Dans cet exercice, nous allons nous intéresser à la résolution "approchée" de l'équation $f(x) = x^3 10x^2 + 5 = 0$

- 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 0x1221efed0>]
```



- 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 parcourera 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`.

[6]:
```
def recherche(xmin,xmax,pas):
    # A compléter ....

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

```
15.8 ms ś 1.95 ms per loop (mean ś 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.

[7]:
```
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)
```

```
1.36 ms ś 42.1 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
```

[7]:
```
array([[-0.69,  0.73,  9.94],
       [-0.68,  0.74,  9.95]])
```

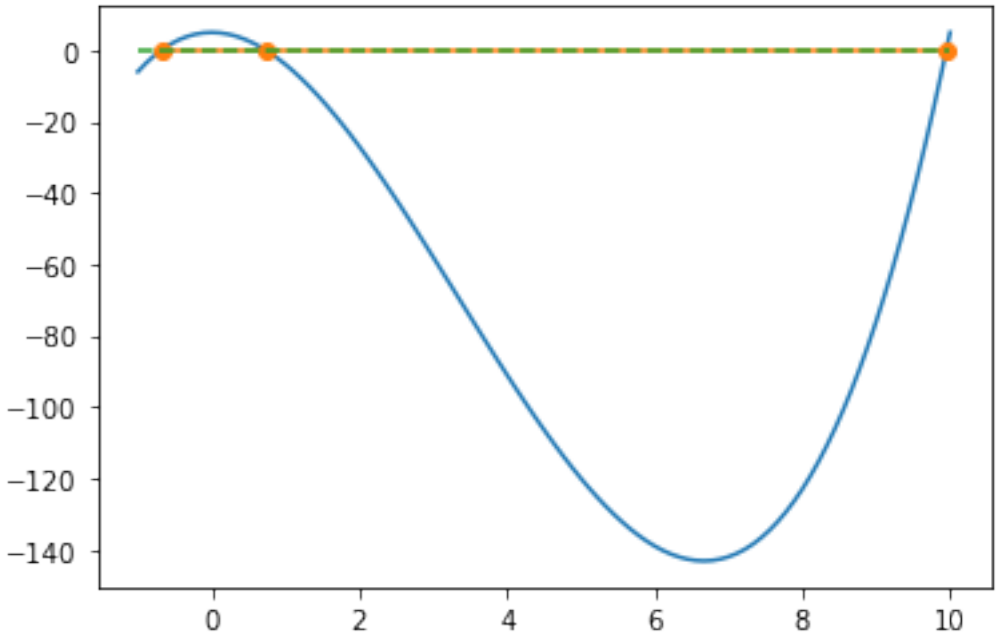
- Ajouter sur le schéma les milieux des intervalles obtenus avec les recherches ci-dessus.
- Refaire le graphique de la première question en limitant le dessin aux valeurs de x dans $[-1, 10]$ (on changera le linspace).

[8]:
```
# Milieux des intervalles
# a compléter ...

# Graphique
# à compléter ...
```

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

[8]:
```
[<matplotlib.lines.Line2D at 0x1222c4f90>]
```

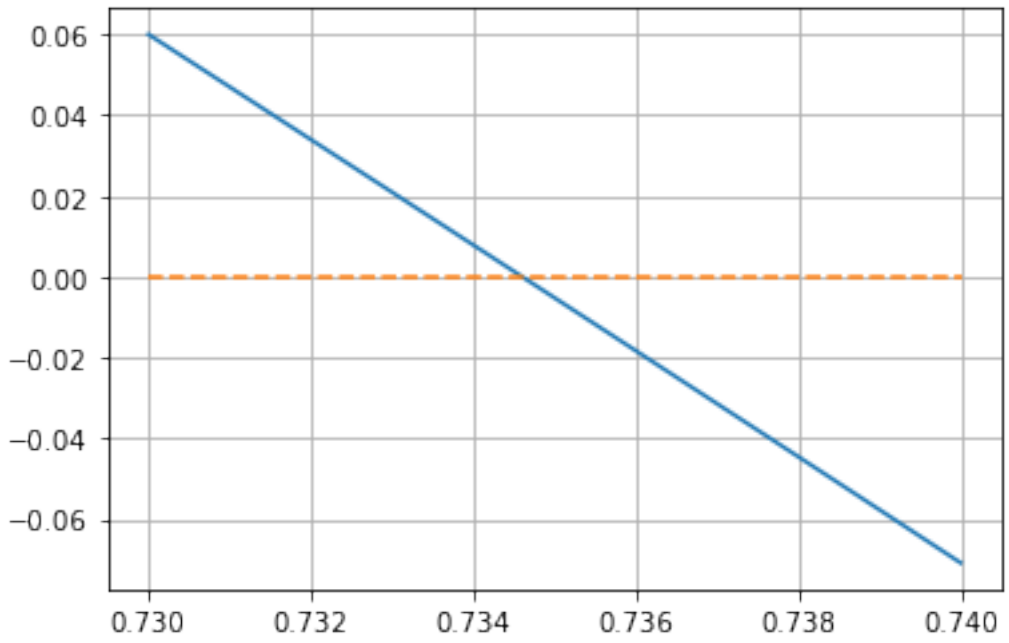


1.2 Recherche par dichotomie (ou bissection)

- 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 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 ...
```



[10]:
```
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)
```

```
35.1 ţs ś 5.59 ţs per loop (mean ś std. dev. of 7 runs, 10000 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.

[11]:
```
import scipy as sp
import scipy.optimize

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

```
9.28 µs ś 133 ns per loop (mean ś std. dev. of 7 runs, 100000 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.
  Le critère d'arret sera exprimé sur les valeurs de f(x), avec $|f(x)| < tol$.
- Comparer les temps d'éxécution.

[14]:
```
def newton( x, tol=0.000001):
    #  A compléter ...
    #  A compléter ...
#  A compléter ...
sol = newton((0.73+ 0.74)/2,1e-10)
print('solution : ',sol)
```

```
2.22 µs ś 68.9 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
solution :  0.7346035077893033
```

[13]:
```
%timeit  scipy.optimize.newton(f,(0.73+ 0.74)/2)
sol = scipy.optimize.newton(f,(0.73+ 0.74)/2)
print('solution : ',sol)
```

```
137 µs ś 16.1 µs per loop (mean ś std. dev. of 7 runs, 10000 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]
```

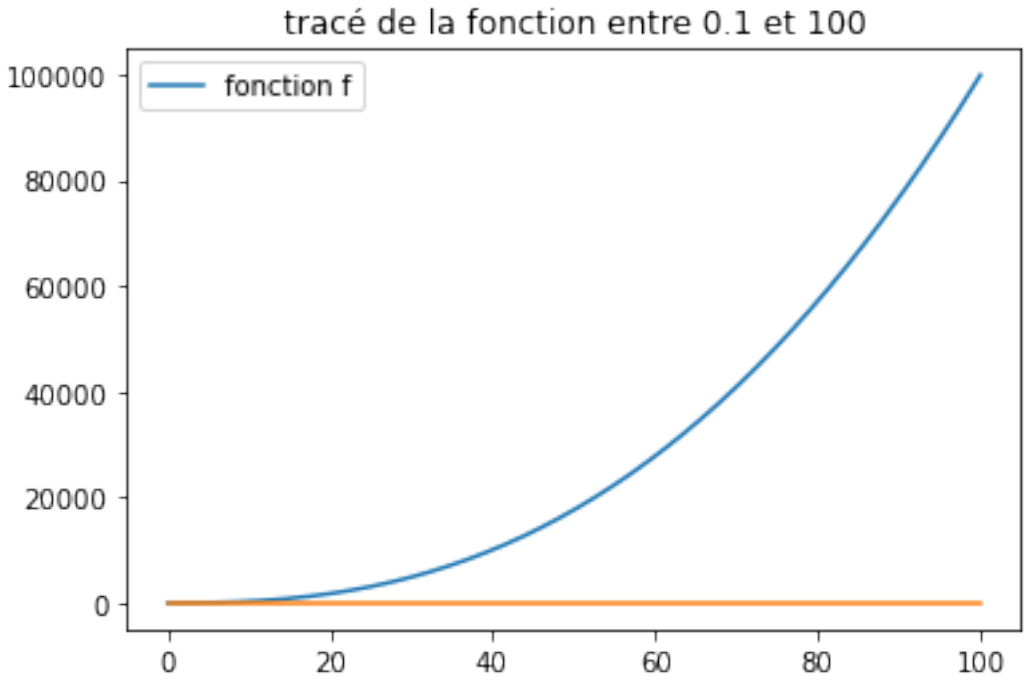
2 Exemple d'exercice noté de TP

Dans cet exercice, nous allons nous intéresser à la recherche des racines de la fonction $f(x) = x^3 sin(\frac{1}{\sqrt{(x)}})$ c'est à dire aux solutions de l'équation $f(x) = 0$ qui sont positives strictement. f n'est pas définie en 0, on commencera donc l'étude en 0.001 dans un premier temps.

Définir la fonction $f(x)$ (fonction python classique) qui correspond à la fonction ci-dessus. Tracer le graphe de la fonction f sur l'intervalle $[0.001, 100]$ (on fera figurer l'axe y =0 qui est en orange sur le graphique ci-dessous).

[15]:
```
# A compléter ...
```

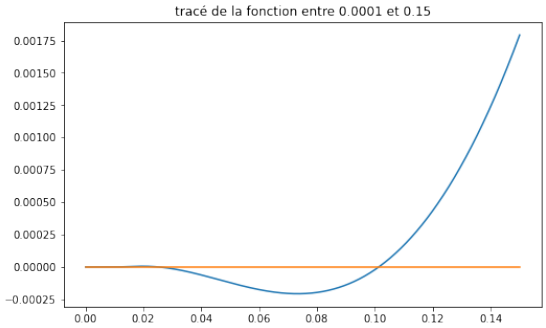
[15]:
```
Text(0.5, 1.0, 'tracé de la fonction entre 0.1 et 100')
```

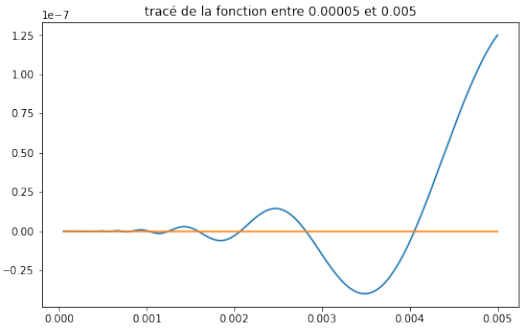


Vu le graphique obtenu, on va refaire deux graphiques,un pour x variant entre 0.0001 et 0.15, puis un en "zoomant" c'est à dire pour x variant entre 0.00005 et 0.005. On fera figurer les deux shémas sur le même dessin comme dans le dessin ci-dessous que l'on doit reproduire.

[16]:
```
# A compléter ...
```

[16]:
```
Text(0.5, 1.0, 'tracé de la fonction entre 0.00005 et 0.005')
```





On va s'intéresser d'une part à la racine qui semble proche de 0.1 et aux racines situées entre 0.001 et 0.003.

Ecrire une fonction qui étant données a, début de l'intervalle , b fin de l'intervalle et un 'pas' renvoie sous la forme d'un tableau à deux lignes les intervalles dans lequel se situent les solutions (la première ligne du tableau contient les extrémités gauches de des intervalles, la deuxième ligne, les extrémités droites de ces mêmes intervalles, les intervalles étant de taille pas). On utilisera la méthode de son choix. On utilisera cette fonction aux alentours de 0.1 puis entre 0.0001 et 0.0004

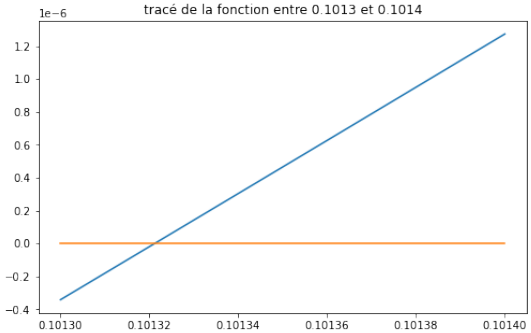
[17]:
```
# A compléter ...
```

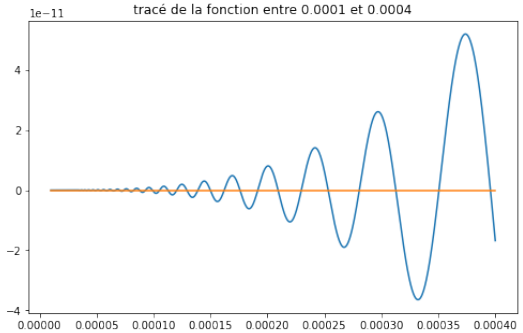
```
racine proche de 0.1, tableau obtenu  [[0.10132118]
 [0.10132119]]
racines  entre 0.0001 et 0.0004, tableau obtenu  [[0.00010543 0.00011257
0.00012047 0.00012923 0.00013898 0.00014988
  0.00016211 0.0001759  0.00019153 0.00020934 0.00022975 0.0002533
  0.00028066 0.00031271 0.00035059 0.00039578]
 [0.00010544 0.00011258 0.00012048 0.00012924 0.00013899 0.00014989
  0.00016212 0.00017591 0.00019154 0.00020935 0.00022976 0.00025331
  0.00028067 0.00031272 0.0003506  0.00039579]]
```

Refaire le graphique (comme ci-dessus et ci-dessous) pour mieux visualiser les racines.

[18]:
```
# A compléter ...
```

[18]:
```
Text(0.5, 1.0, 'tracé de la fonction entre 0.0001 et 0.0004')
```

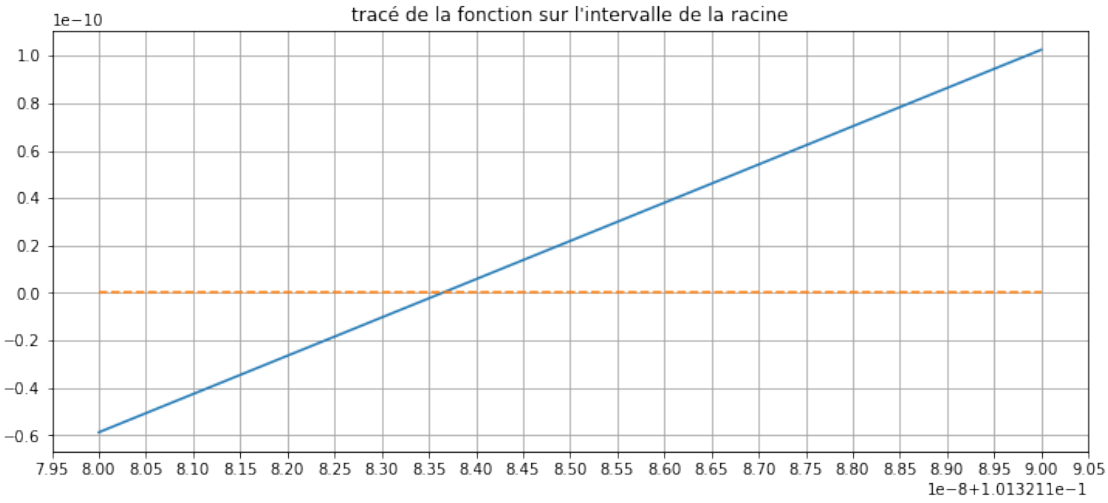




On veut calculer une valeur approchée précise pour la solution aux alentours de 0.1 Commencer par faire un dessin pour l'intervalle obtenu à la question précédente.

[19]:
```
# A compléter ...
```

[19]:
```
Text(0.5, 1.0, "tracé de la fonction sur l'intervalle de la racine")
```



Ecrire ensuite une fonction de recherche d'approximation de racine par dichotomie.

Elle affinera la recherche d'une racine, à partir d'un intervalle obtenu comme ci-dessus qui ne contient qu'une unique racine proche de celle qu'on cherche. On utilisera la méthode de son choix.

On utilisera cette fonction pour donner une approximation de la racine proche de 0.1 avec une erreur inférieure à 10^{-15}

[20]:
```
# A compléter ...
```

```
solution 0.10132118364233766
f(sol) -1.7203149052829069e-18
```