TP4_correction

February 18, 2020

1 Recherche des racines d'équations non linéaires

On commence par importer les bibliothèques qui vont bien :

```
In [1]: %matplotlib inline
        import matplotlib as mpl
        import matplotlib.pyplot as plt
        import numpy as np
```

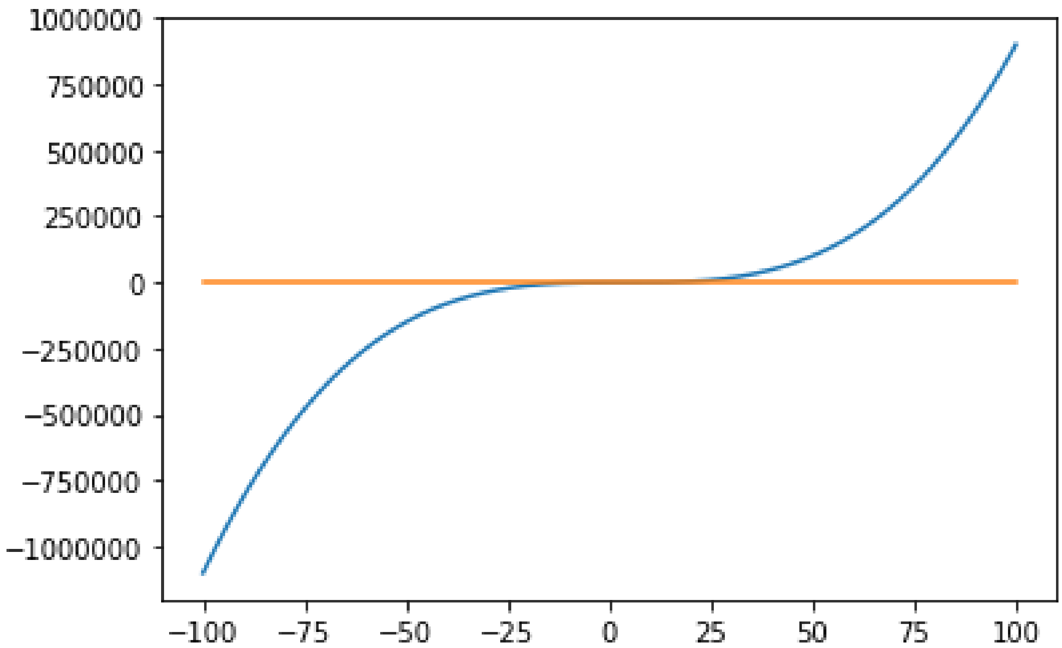
1.1 Recherche incrémentale

Dans cet exercice, nous allons nous intéresser à la résolution 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. Tracerle graphe de la fonction f sur l'intervalle $[-100, 100]$ (on fera figurer l'axe en orange ci-dessous).

```
In [2]: def f(x):
            return x**3-10*x**2+5
        x = np.linspace(-100,100,1000)
        fig,ax = plt.subplots()
        ax.plot(x,f(x))
        ax.plot(x,np.zeros_like(x))
```

```
Out[2]: [<matplotlib.lines.Line2D at 0x7fc5cfe8f2e8>]
```



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.

POour 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 l y aura dans la première ligne la borne inférieure et pour la deuxième ligne la borne supérieure pour chacun des N intervalles.

Pour cela, on écrira 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` : `%timeit recherche(-100,100,1e-2)`). Attention du coup le calcul n'est pas instantané.

```
In [4]: def recherche(xmin,xmax,pas):
            x=xmin
            l=np.empty((2,0))
            while(x<xmax):
                if f(x)*f(x+pas)<0:
                    l = np.concatenate((l,np.array([[x],[x+pas]])),axis=1)
                x=x+pas
            return l
        %timeit recherche(-100,100,1e-2)

        recherche(-100,100,1e-2)
```

```
22.6 ms ś 4.01 ms per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

```
Out[4]: array([[-0.69,  0.73,  9.94],
               [-0.68,  0.74,  9.95]])
```

Ecrire désormais 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.

```
In [7]: def recherche_vec(xmin,xmax,pas):
            x1 = np.arange(xmin,xmax,pas)
            x2 = x1 + pas
            p = f(x1)*f(x2)
            l = np.where(p<0)
            return np.asarray([x1[l],x1[l]+pas])
        %timeit recherche_vec(-100,100,1e-2)
        racines = recherche_vec(-100,100,1e-2)
        racines
```

```
5.81 ms ś 411 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```

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

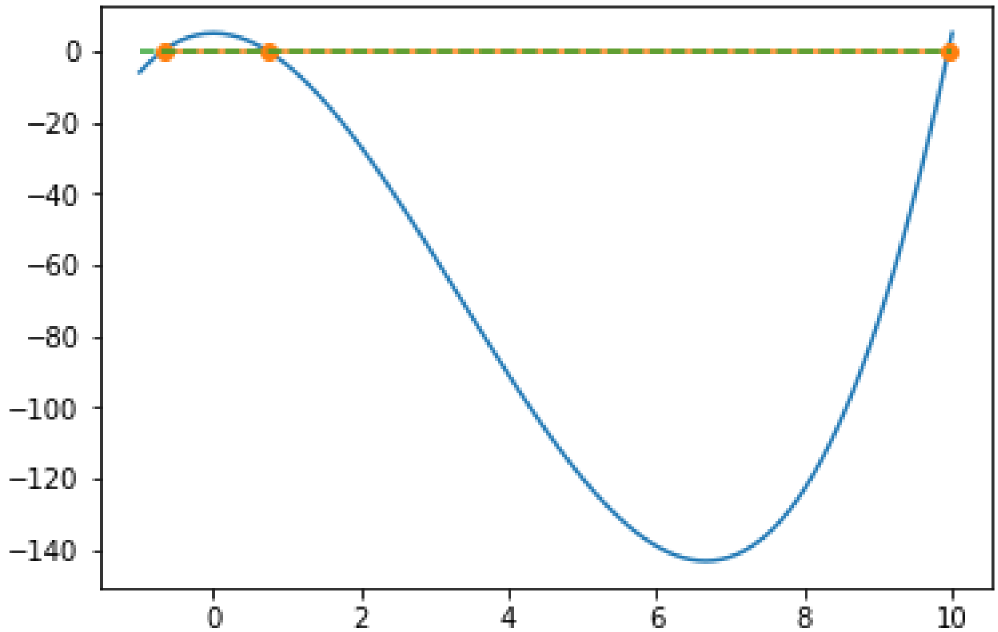
Refaire le graphique précendent en limitant le dessin aux valeurs de x dans $[-1, 10]$.
Ajouter les milieux des intervalles obtenus avec les recherches ci-dessus.

```
In [8]: racines_m = racines.mean(axis=0)
        print(racines_m)
        x = np.linspace(-1,10,100)
        fig,ax = plt.subplots()
        z = np.zeros_like(racines_m)
        zz = np.zeros_like(x)

        ax.plot(x,f(x))
        ax.plot(racines_m,z,marker='o')
        ax.plot(x,zz,ls='--')
```

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

```
Out[8]: [<matplotlib.lines.Line2D at 0x7fc5ffc43ba8>]
```



1.2 Recherche par dichotomie (ou bissection)

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

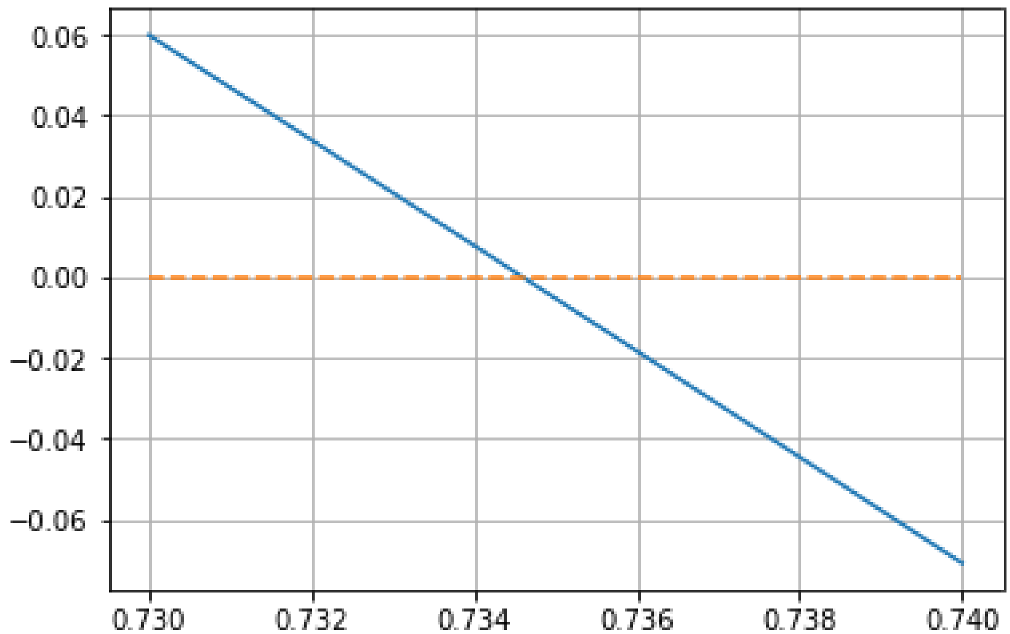
Elle affinera la recherche d'une racine, à partir d'un intervalle qui ne contient qu'une unique racine.

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 graphhe de la fonction sur cet ntervalle puis calculer la racine par dichotomie. On évaluera le temps d'execution pour cette recherche.

```
In [9]: x = np.linspace(0.73, 0.74,100)
        fig,ax = plt.subplots()
        ax.plot(x,f(x))
        ax.plot(x,np.zeros_like(x),ls='--')
        ax.grid()
```




```
In [8]: def dicho(xmin,xmax,xtol):
            xm = (xmin+xmax)/2
            while np.abs(xmin-xmax)>xtol :
                xm = (xmin+xmax)/2
                if f(xmin)*f(xm)<0:
                    xmax=xm
                else:
                    xmin=xm
            return np.array([xmin,xmax])

        %timeit  dicho(0.73, 0.74, 1e-7)
        sol = dicho(0.73, 0.74, 1e-7)
        print('solution',np.mean(sol),'f(sol)',f(np.mean(sol)))
        print('intervalle :',sol)
```

```
22.9 ţs ś 2.14 ţs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
solution 0.7346035385131837 f(sol) -4.016576964360752e-07
intervalle : [0.7346035  0.73460358]
```

Effectuer la même recherche de racine, mais en utilisant la librairie `scipy` (fonction `scipy.optimize.bisect`).

On calculera le temps d'excution decette méthode et on comparera avec la méthode précédente.

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

```
%timeit  scipy.optimize.bisect(f,0.73, 0.74,xtol=1e-7)
sol = scipy.optimize.bisect(f,0.73, 0.74,xtol=1e-7)
print(sol,f(sol))
```

```
12.3 ţs ś 732 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
0.7346035003662108 9.70431504043745e-08
```

1.3 Méthode de recherche de Newton-Raphson

Comme précédement, on implémentera 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.

```
In [10]: def newton(x,tol):
             xp = x - 1e-5
             while np.abs(f(x))>tol:
                 xs = x - (x-xp)/(f(x)-f(xp)) * f(x)
                 xp=x
                 x=xs
             return x
         %timeit  newton((0.73+ 0.74)/2,1e-10)
         sol = newton((0.73+ 0.74)/2,1e-10)
         print(sol,f(sol))
```

```
8.28 ţs ś 2.35 ţs per loop (mean ś std. dev. of 7 runs, 100000 loops each)
0.7346035077893034 -8.881784197001252e-16
```

```
In [11]: %timeit  scipy.optimize.newton(f,(0.73+ 0.74)/2)
         sol = scipy.optimize.newton(f,(0.73+ 0.74)/2)
         print(sol,f(sol))
```

```
2.71 ţs ś 144 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
0.7346035077893033 0.0
```

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.

```
In [12]: import scipy as sp
         import scipy.optimize

         pas = 1e-2
         sol = []
```

```python
racines = recherche_vec(-100,100,pas)
print(racines)
for i in range(racines.shape[1]):
    sol = sol + [scipy.optimize.newton(f,racines.mean(axis=0)[i])]
sol,f(np.array(sol))
```

```
[[-0.69  0.73  9.94]
 [-0.68  0.74  9.95]]
```

```
Out[12]: ([-0.6840945657036899, 0.7346035077893033, 9.949491057914384],
          array([-6.21724894e-15,  0.00000000e+00, -1.13686838e-13]))
```