Solutions pour la première partie du T.P. 13

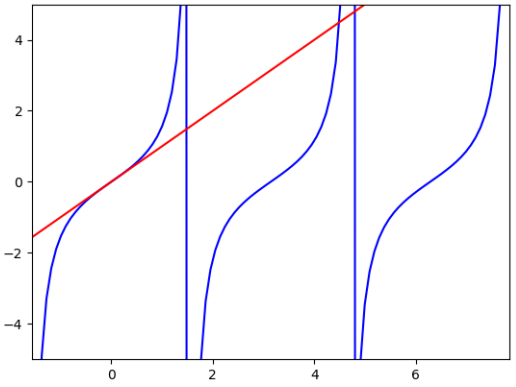
1 Révisions et compléments sur les tracés de graphes de fonctions, recherches de zéros

Pour les commandes que vous ne connaissez pas sur plot, voir l'appendice du TP

a) Tracez le graphe de la fonction tan sur $]-\pi/2, 5\pi/2[$.
Rappel : on doit définir un tableau `X` d'abscisse, puis un tableau `Y=pl.tan(X)` puis faire `pl.plot(X,Y)` puis enfin `pl.show()`.

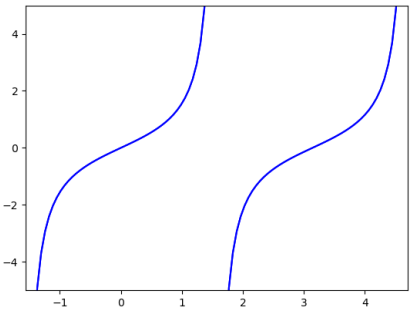
```
import pylab as pl
xmin=-pl.pi/2+0.01
xmax=5*pl.pi/2-0.01
x=pl.linspace(xmin,xmax,100)
y=pl.tan(x)
pl.axis([xmin,xmax,-5,5])
pl.plot(x,y,"b")
```

Avec le résultat (où il y a déjà aussi la droite demandée au b)) :



Remarque : PYLAB va joindre les points à travers les discontinuités, comment éviter ce phénomène ?
On peut tracer morceau par morceau en découpant l'intervalle des x en deux, un intervalle avant $\pi/2$ et un après comme dans ce qui suit :

```
xmin1=-pl.pi/2+0.01
xmax1=pl.pi/2-0.01
xmin2=pl.pi/2+0.01
xmax2=3*pl.pi/2-0.01
X1=pl.linspace(xmin1,xmax1,50)
X2=pl.linspace(xmin2,xmax2,50)
pl.figure("Graphe de Tan, v2")
Y1=pl.tan(X1)
Y2=pl.tan(X2)
pl.axis([xmin1,xmax2,-3,3])
pl.plot(X1,Y1,"b")
pl.plot(X2,Y2,"b")
pl.show()
```



b) Pour tracer la première bissectrice en rouge, on rajoute la ligne :

```
pl.plot(x,x,"r")
```

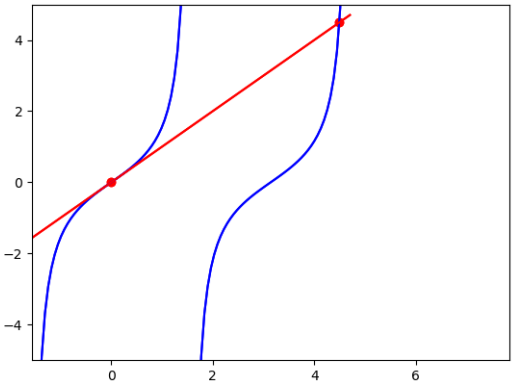
c) Détermination graphique des coordonnées des points d'intersection entre les deux courbes dans $[0, 2\pi]$ (la figure PYLAB donne les coordonnées du curseur).

Bien sûr, il y a le point $(0, 0)$, on le savait.

Avec le curseur, on voit des coordonnées proches de $(4.5, 4.5)$ pour le second vrai point d'intersection.

d) Déterminez numériquement ces points d'intersections avec la fonction `fsolve` du module `scipy.optimize` et rajoutez les sur la figure avec le symbole `o` d'une autre couleur. On va chercher le point d'intersection d'abscisse proche de 4.5 en donnant cette valeur à `fsolve`.

```
import scipy.optimize as scp
# Comme cette fonction prend en argument une fonction, on définit une fonction.
def difference(x):
    return pl.tan(x)-x
absci=scp.fsolve(difference,4.5)
# le résultat est une liste a un élément : [4.49340946]
#Puis le tracé :
pl.figure("Avec points d'intersections marqués")
pl.plot(absci,absci,"or") # le point d'intersection trouvé le "o" dit de faire un rond
pl.plot([0],[0],"or") # l'origine  le "r" dit que le rond est rouge.
pl.plot(X1,Y1,"b")
pl.plot(X2,Y2,"b")
pl.plot(X,X,'r')
pl.axis([xmin,xmax,-5,5])
pl.show()
```



N.B. Nous avons déjà rencontré des fonctions de recherche de zéro au TP5 : la dichotomie et la fausse position. Nous allons voir dans le cours du chapitre 10, une nouvelle méthode : la méthode de Newton qui est celle utillisée le plus souvent en calcul numérique.

e) **Cas d'une fonction définie par cas :** Tracer le graphe de la fonction $f : x \mapsto \begin{cases} x^2, \text{ si } \lfloor x \rfloor \text{est paire,} \\ 1 - x^3 \text{ sinon} \end{cases}$ pour $x \in [-10, 10]$.

On définit sans mal ladite fonction.

```
def f(x):
    if int(x)%2==0:
        return x**2
    else :
        return 1-x**3
```

Mais si on essaie de l'utiliser avec un tableau numpy (ou pylab) :

```
# Premier essai
X=pl.linspace(-10,10,100)
Y=f(X)
pl.plot(X,Y)
```

On obtient le message :

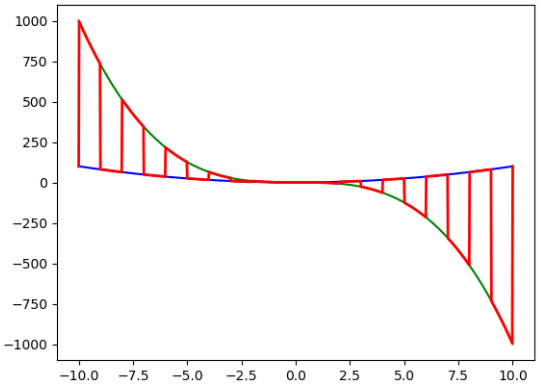
```
   if int(x)%2==0:
TypeError: only size-1 arrays can be converted #to Python scalars
```

Le problème ? La ligne `if int(x)` n'est pas adaptée pour manipuler un tableau numpy pour plusieurs raisons. Déjà l'opérateur `int`, mais surtout le `if` qui doit agir élément par élément.. On revient donc aux bonnes vieilles boucles qui parcourent le tableau `X`.

```
Y=[]
for x in X:
    Y.append(f(x))
```

Voici un affichage où on fait figurer aussi les deux fonctions $x \mapsto x^2$ et $x \mapsto 1-x^3$ et notre fonction avec une épaisseur de trait plus grande.

```
pl.figure("Fonction définie par cas")
Z1=X**2
Z2=1-X**3
pl.plot(X,Z1,"b")
pl.plot(X,Z2,"g")
pl.plot(X,Y,"r",linewidth=2)
pl.show()
```



2 Etude de familles de fonctions et de suites définies implicitement

2.1 La suites des polynômes de Taylor de l'exponentielle : programmation et tracé d'une famille de fonctions

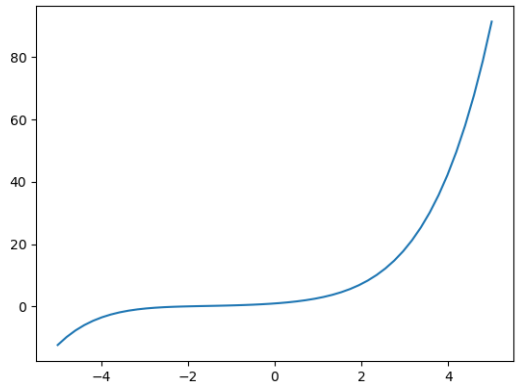
a) Ecrire une fonction `Taylor_exp(x,n)` qui prend comme arguments un entier `n` et un flottant `x` et renvoie la valeur de $f_n(x)$.

```
def Taylor_exp(x,n):
    acc=1
    S=0
    for i in range(n+1):
        S=S+acc
        acc=acc*x/(i+1)
    return S
```

Noter qu'on a évité le calcul de x^i et de $i!$ à chaque étape, avec un accumulateur.

b) Tracer une des fonctions obtenues par exemple pour $n = 5$, en l'appliquant à un tableau `x` représentant des abscisses entre -5 et 5.

```
pl.figure("Taylor Exp n=5")
pl.clf()
X=pl.linspace(-5,5,50)
Y=Taylor_exp(X,5)
pl.plot(X,Y)
pl.show()
```



2.2 La suites des polynômes de Taylor de l'exponentielle : tracés, zéros

a) Tracez sur une même figure les graphes des fonctions f_n pour $n \in [\![1, 20\ ]\!]$ et $x \in [-5, 5]$.

```
pl.figure("Famille des T_(n,exp)")
pl.clf()
x=pl.linspace(-10,2,50)
for i in range(1,21):
    Y=Taylor_exp(X,i)
    pl.plot(X,Y)
pl.show()
```