Corrigé de la séance Python 1

1 Dichotomie

On commence par charger les modules `matplotlib` et `numpy` :

```
import matplotlib.pyplot as plt
import numpy as np
```

On initialise en mettant les valeurs a et b dans l'ordre (même si ce n'est pas nécessaire). Lors de la boucle, on peut tester si l'on est tombé exactement sur une racine, même si ce cas est extrêmement peu probable (on sort alors de la boule et de la fonction en renvoyant la valeur trouvée). En sortie de boucle, on dispose de deux valeurs $g \leqslant d$ encadrant la racine cherchée ; on peut renvoyer n'importe quelle valeur appartenant à cet intervalle (par exemple le milieu).

```
def dichotomie(f, a, b, epsilon):
    """données : f continue sur un intervalle,
    a et b tels que f(a)f(b) <= 0
    résultat : une approximation à epsilon près d'une solution de
    l'équation f(x)=0 obtenue par dichotomie"""
    g, d = min(a,b), max(a,b)
    while d-g > epsilon:
        m = (g+d)/2
        if f(m) == 0:
            return m
        elif f(g) * f(m) < 0:
            d = m
        else:
            g = m
    return (g+d)/2
```

À l'issue du $k^{\text{ème}}$ passage dans la boucle, l'intervalle $[g, d]$ est de longueur $\frac{b-a}{2^k}$. Le nombre n de passages dans la boucle est donc égal au plus petit entier n tel que $\frac{b-a}{2^n} \leqslant \varepsilon$, *i.e.* tel que $n \geqslant \log_2(\frac{b-a}{\varepsilon})$. On a donc $n = \left\lceil \log_2(\frac{b-a}{\varepsilon}) \right\rceil$. Si l'on veut une estimation deux fois plus précise, il faut diviser ε par deux. La formule $\log_2(2x) = 1 + \log_2 x$ montre qu'il faut faire un passage supplémentaire dans la boucle. C'est bien, mais d'autres algorithmes sont beaucoup plus efficaces. Si on écrivait les nombres en base 2, doubler la précision sur la valeur obtenue correspond à peu près à en obtenir un chiffre exact de plus. L'algorithme de dichotomie nécessite pour cela un passage supplémentaire dans la boucle (on dit que la convergence est *linéaire*). La méthode de Newton permet elle, grâce à un unique passage supplémentaire dans la boucle, de *doubler* le nombre des chiffres exacts !

Améliorons la fonction comme indiqué dans l'énoncé :

- pour afficher le nombre d'itérations effectuées, on utilise un compteur, incrémenté à chaque passage dans la boucle ;
- les valeurs $f(x)$ sont calculées plusieurs fois : lors de la comparaison du signe de $f(g)$ à celui de $f(m)$, la valeur $f(g)$ a déjà été calculée à l'étape précédente. Pour éviter ceci, on utilise deux autres variables : val_g et val_d, respectivement égales à $f(g)$ et $f(d)$.

On supprime également le test d'égalité de $f(m)$ à 0.

```
def dichotomie(f,a,b,epsilon):
    """Données : f continue sur un intervalle,
    a et b tels que f(a)f(b) <=0
    Résultat : une approximation à epsilon près d'une solution
    de l'équation f(x)=0 obtenue par dichotomie.
    Affiche le nombre d'itérations effectuées."""
    g, d = min(a,b), max(a,b)
    valg, vald = f(g), f(d)
    n = 0
    while d-g > epsilon:
```

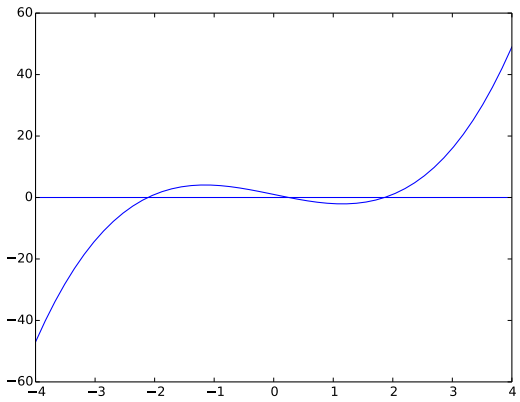
```
        n += 1
        m = (g+d)/2
        valm = f(m)
        if valg * valm < 0 :
            d = m
            vald = valm
        else:
            g = m
            valg = valm
    print("nombre d'itérations nécessaires :", n)
    return (g+d)/2
```

Testons (dans la console) pour trouver une valeur approchée de $\frac{\pi}{2}$:

```
>>> dichotomie(np.cos, 0, 3, 1e-10)
nombre d'itérations nécessaires : 35
1.5707963268068852
```

Pour trouver les racines de la fonction $f : x \mapsto x^3 - 4x + 1$, commençons par la tracer pour les localiser grossièrement :

```
def f(x):
    return x**3 - 4*x + 1
x = np.linspace(-4,4)
y = f(x)
plt.plot(x,y)
plt.axhline()
plt.show()
```



Le dessin montre que chacun des trois intervalles $[-3,-1]$, $[-1,1]$ et $[1,3]$ contient une racine (unique car ce polynôme de degré trois ne peut en avoir davantage). On obtient une approximation de la plus petite par

```
>>> dichotomie(f, -3, -1, 1e-10)
nombre d'itérations nécessaires : 35
-2.1149075414577965
```

2 Méthode de la fausse position

Notons g et d au lieu de g_n et d_n : la droite joignant les deux points a pour équation

$$y - f(g) = \frac{f(d) - f(g)}{d - g}(x - g).$$

L'intersection avec l'axe des abscisses ($y = 0$) donne

$$x = \frac{gf(d) - df(g)}{f(d) - f(g)}.$$

On écrit la fonction `faussePos` sur le même modèle que la fonction `dichotomie`, seule la nouvelle abscisse testée m étant donnée par une autre formule. Attention cependant au test d'arrêt : la longueur du segment $[g_n, d_n]$ ne tend plus forcément vers 0... On s'arrête lorsque $f(m-\varepsilon)$ ou $f(m+\varepsilon)$ est de signe opposé à $f(m)$ (il y a alors une racine dans l'intervalle $[m-\varepsilon, m]$ ou $[m, m+\varepsilon]$).

```
def faussePos(f,a,b,epsilon):
    """Données : f continue sur un intervalle,
    a et b tels que f(a)f(b) <=0
    Résultat : une approximation d'une solution de l'équation f(x)=0
    obtenue par la méthode de la fausse position.
    Test d'arrêt : m tel que f(m-eps) ou f(m+eps) de signe opposé à f(m).
    Affiche le nombre d'itérations effectuées."""
    g, d = a, b
    m = g
    valg, vald = f(g), f(d)
    valm = valg
    termine = False
    n = 0
    while not termine:
        n += 1
        m = (g*vald - d*valg) / (vald - valg)
        valm = f(m)
        if valg * valm <= 0:
            d = m
            vald = valm
        else:
            g = m
            valg = valm
        if valm * f(m-epsilon) <= 0 or valm * f(m+epsilon) <= 0 :
            termine = True
    print("nombre d'itérations nécessaires :", n)
    return m
```

Testons pour la fonction cos :

```
>>> faussePos(np.cos,0,3,1e-10)
nombre d'itérations nécessaires : 5
1.5707963267948963
```

C'est beaucoup plus efficace que la méthode de dichotomie. Pour la fonction f en revanche, c'est moins probant :

```
>>> faussePos(f,-3,-1,1e-10)
nombre d'itérations nécessaires : 27
-2.1149075414146727
```

3 Méthode des cordes

Comme on a besoin de connaître à chaque instant deux termes consécutifs de la suite $(x_n)_n$ pour pouvoir calculer le suivant, on manipule encore deux variables, notées x et y (dans lesquelles sont respectivement stockées les valeurs x_n et x_{n+1}).

```
def cordes(f,a,b,epsilon):
    """Données : f continue sur un intervalle,
    a et b tels que f(a)f(b) <=0
    Résultat : une approximation d'une solution de l'équation f(x)=0
    obtenue par la méthode des cordes.
    Test d'arrêt : deux termes consécutifs x,y tels que |x-y| <= epsilon.
    Affiche le nombre d'itérations effectuées."""
    x, y = a, b
    n = 0
    while abs(x-y) > epsilon and n < 50:
        n += 1
```

```
        valx, valy = f(x), f(y)
        z = (x*valy - y*valx) / (valy - valx)
        x, y = y, z
    if n < 50:
        print("nombre d'itérations nécessaires :", n)
        return y
    else:
        print("aucune solution trouvée en 50 itérations")
```

Les tests :

```
>>> cordes(np.cos,0,3,1e-10)
nombre d'itérations nécessaires : 6
1.5707963267948968

>>> cordes(f,-3,-1,1e-10)
nombre d'itérations nécessaires : 13
-2.114907541476756
```

4 Méthode de Newton

La tangente a pour équation

$$y - f(x_n) = f'(x_n)(x - x_n).$$

L'intersection de cette tangente avec l'axe des abscisses fournit

$$x = x_n - \frac{f(x_n)}{f'(x_n)}.$$

```
def newton(f,df,a,epsilon):
    """Données : f continue sur un intervalle,
    a : valeur approchée d'un zéro de f
    Résultat : une approximation d'une solution de l'équation f(x)=0
    obtenue par la méthode de Newton.
    Test d'arrêt : deux termes consécutifs x,y tels que |x-y| <= epsilon.
    Affiche le nombre d'itérations effectuées."""
    x = a
    y = x - f(x)/df(x)
    n = 0
    while abs(y-x) > epsilon and n < 50:
        n += 1
        z = y - f(y) / df(y)
        x, y = y, z
    if n < 50:
        print("nombre d'itérations nécessaires :", n)
        return y
    else:
        print("aucune solution trouvée en 50 itérations")
```

Testons pour la fonction cos. On ne peut pas prendre pour valeur initiale 0 car $f'(0) = 0$. On prend 1 pour valeur initiale.

```
>>> newton(np.cos, lambda x : -np.sin(x), 1, 1e-10)
nombre d'itérations nécessaires : 3
1.5707963267948966
```

Pour la fonction f :

```
>>> newton(f, lambda x : 3*x**2 - 4, -3, 1e-10)
nombre d'itérations nécessaires : 5
-2.1149075414767555
```