TD Modélisation 1 : Chimie et données expérimentales
Proposition de correction

Contents

1 Données, graphes et statistiques

Dans l'esprit de la réforme des programmes pour l'année 2021, nous utiliserons le langage de programmation python dans divers contextes en physique et en chimie. Dans cette première séance, nous allons nous concentrer sur les méthodes les plus utiles en T.P. et les plus « immédiates » en python : il faut cependant bien avoir à l'esprit que ces notions de python vont au-delà de vos connaissances actuelles, et que ces compétences seront à acquérir tout au long des deux années.

Le travail a effectuer est lié au fichier python *Python_TP.py* **consignant les techniques qui pourront être utiles dans le cadre de T.P.** : gardez ce fichier sur votre clef USB, vous serez encouragés à l'utiliser en T.P. en remplacement de *Regressi* ou *LatisPro*. Les commandes python résentés ici se trouvent dans ce fichier.

1.1 Validation d'un modèle linéaire - Déduction d'une valeur inconnue grâce à une droite d'étalonnage

1.1.1 Présentation du problème

Dans cette partie, nous allons prendre l'exemple d'un dosage spectro-photométrique : celui des ions Fe^{2+}. On dispose de 5 solutions de complexe $[Fe(SCN)]^{2+}$ soigneusement préparées et on mesure l'absorbance de chacune de ces solutions à l'aide d'un spectrophotomètre.

On souhaite valider que la loi de Beer-Lambert qui permet de décrire correctement les résultats expérimentaux obtenus :

$$A = \epsilon \cdot \ell \cdot C \text{ avec } \begin{cases} A \text{ absorbance} \\ \ell \text{ largeur de solution traversée en cm} \\ C \text{ en mol L}^{-1} \\ \epsilon \text{ en L mol}^{-1} \text{ cm}^{-1} \end{cases}$$

Données du problème :

- Les résultats sont consignés dans le tableau ci-dessous ($\lambda = 580$ nm)

C (mol L^{-1})	$2.5 \cdot 10^{-4}$	$5.0 \cdot 10^{-4}$	$1.0 \cdot 10^{-3}$	$1.5 \cdot 10^{-3}$	$2.0 \cdot 10^{-3}$
A	0.143	0.264	0.520	0.741	0.998

- Dans la notice du spectrophotomètre, le constructeur indique que la précision sur la mesure de A est $\Delta A = \pm$ 5 %. On l'interprète : l'incertitude-type sur un mesurage de A est $u(A) = \frac{\Delta A}{\sqrt{3}}$ (nous verrons peut-être pourquoi plus tard dans l'année);
- Les incertitudes-types sur C sont négligées.

On mesure l'absorbance d'une solution inconnue contenant des ions Fe^{2+}, et on trouve 0.574. Quelle est la concentration en ions Fer II dans cette solution ?

1.1.2 Création des données

La première étape consiste à créer les données. Pour cela, on utilise des tableaux numpy : ceux-ci nécessite l'importation de la bibliothèque numpy.

[1]:
```
import numpy as np
```

On a ici importé numpy avec un alias `np` : ainsi, nous avons accès à de nouvelles fonctions et objets python, et ces objets seront appelés avec la syntaxe `np.objet` . Nous allons avoir besoin de la structure de donnée tableau, nommé `array` dans numpy : cette structure sera appelé par `np.array`.

Par exemple `M = np.array([18.5, 16, 13.5, 14, 16])` est un tableau (à une dimension), nommé M, contenant des notes. Attention on notera que le séparateur décimal est le point et non la virgule !

Question 1 : Créer les deux tableaux ici nécessaires, qui contiennent les vaelurs de concentration et d'absorbance.

[2]:
```
C = np.array([2.5e-4, 5.0e-4, 1.0e-3, 1.5e-3, 2.0e-3]) # Compléter ici avec les␣
↪valeurs numériques pour la concentration
A = np.array([0.143, 0.264, 0.520, 0.741, 1.008]) # Compléter ici avec les valeurs␣
↪numériques pour l'absorbance
```

[3]:
```
print(C)
```

```
[0.00025 0.0005  0.001   0.0015  0.002  ]
```

Question 2 : Ajouter un tableau contenant les incertitudes-types sur la mesure de l'absorbance, en effectuant le calcul $u(A) = \frac{\Delta A}{\sqrt{3}} = \frac{0.05A}{\sqrt{3}}$. Comme on travaille avec des structure numpy on utilisera la fonction `np.sqrt(...)` pour prendre la racine carré.

[4]:
```
u_A = (5/100)*A/np.sqrt(3)
```

[5]:
```
print(u_A)
```

```
[0.00412805 0.00762102 0.01501111 0.02139083 0.02909845]
```

1.1.3 Représentation sous forme d'un graphe

Représentons les données sous forme d'un graphe de l'absorbance en fonction de la concentration. Pour cela, il nous faut importer la bibliothèque matplotlib, avec l'alias `plt` :

[6]:
```
import matplotlib.pyplot as plt
```

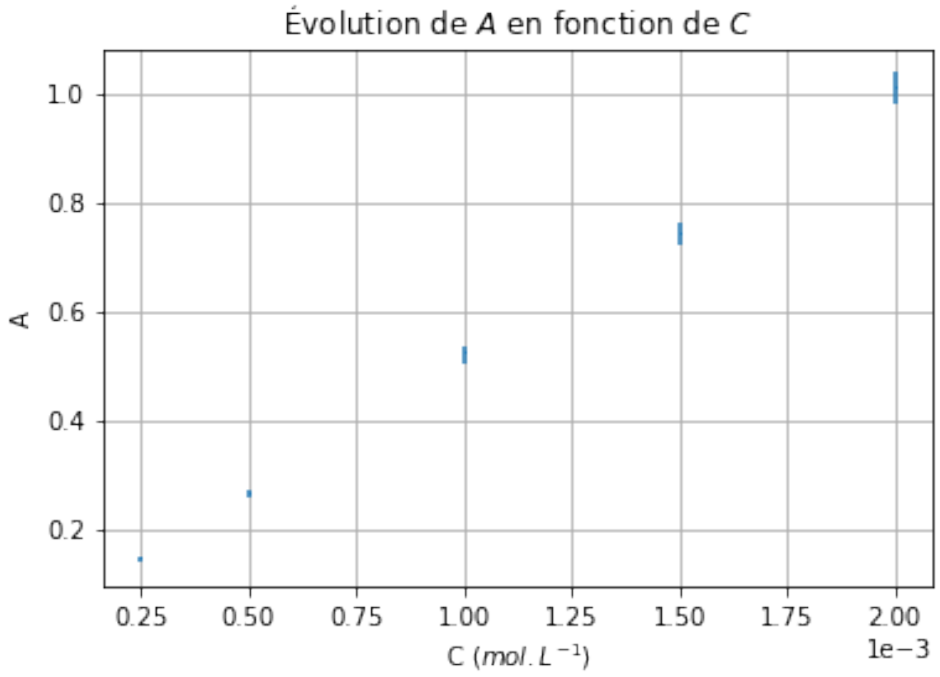
La syntaxe permettant d'afficher le graphe comporte plusieurs lignes :

[ ]:
```
plt.figure()    # permet d'afficher une nouvelle fenêtre (avec thonny, non necessaire␣
 ↪ici)
plt.plot(X, Y, 'o') # graphe de Y en fonction de X ( avec des points, grâce à `o`; on␣
 ↪peut autrement utiliser '+' pour des croix)
plt.xlabel('...')  # légende des abscisses
plt.ylabel('...')   # légende des ordonnées
plt.title('...')   # titre du graphe
plt.grid()       # Pour afficher la grille (non nécessaire)
plt.show()       # Pour afficher le graphe
```

On peut remplacer `plt.plot(X, Y, 'o')` par `plt.errorbar(X, Y, yerr = u_Y, fmt=',')` pour afficher les barres d'incertitudes.

Question 3 : Afficher le graphe voulu (à faire sans puis avec les barres d'incertitudes) :

[18]:
```
plt.figure()
#plt.plot(C, A, 'o')     # Tracé des points sans barres d'incertitude
plt.errorbar(C, A, yerr = u_A, fmt=',')     # Tracé des points avec barres␣
 ↪d'incertitude
plt.xlabel('C ($mol.L^{-1}$)')
plt.ylabel('A' )
plt.ticklabel_format(axis = 'x', style = 'sci', scilimits= (0,0))  # non necessaire,␣
 ↪pour améliorer la visibilité des graduations en abscisses
plt.title('Évolution de $A$ en fonction de $C$')
plt.grid()
plt.show()
```



1.1.4 Régression linéaire : détermination et affichage

Le modèle théorique liant l'absorbance à la concentration étant linéaire (loi de Beer-Lambert), nous allons ajouter le modèle de la régression linéaire au graphe, afin de valider ou non ce modèle théorique.

Pour cela, on utilise la fonction `polyfit` de numpy. Elle permet d'ajuster ("fit" en anglais) un modèle polynomial aux données. Nous nous limiterons généralement à `np.polyfit(X, Y, 1)` qui permet d'obtenir les paramètres (a et b) du modèle avec fonction affine $y = ax + b$ (polynôme d'ordre 1).

Syntaxe :

[ ]:
```
reg = np.polyfit(X, Y, 1)    # Régression linéaire de Y en fonction de X (y=ax+b)
a = reg[0]                   # a pente
b = reg[1]                   # b ordonnée à l'origine
```

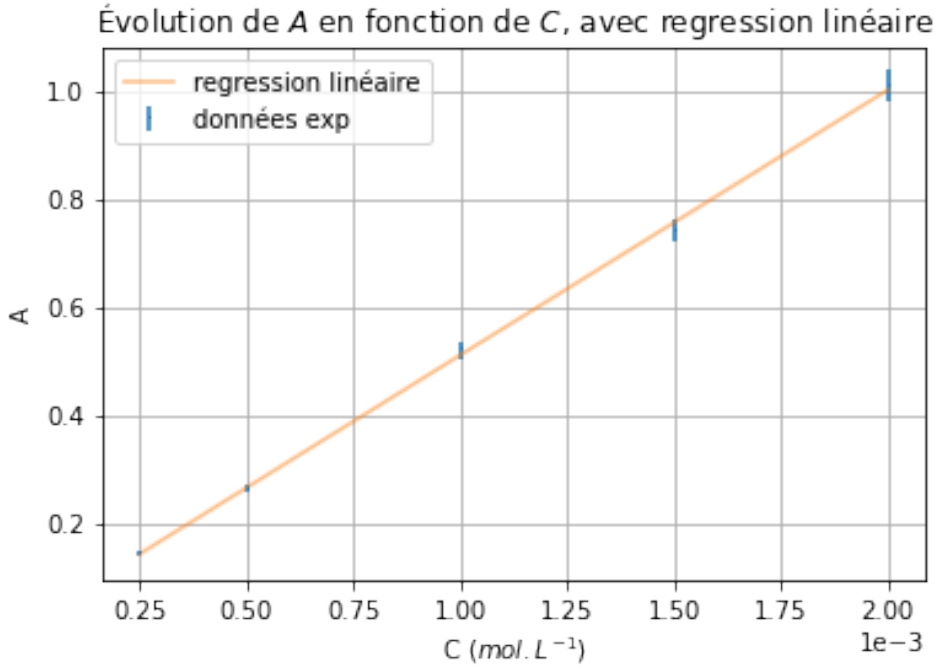
Dans une figure, on pourra utiliser `plt.plot(X, np.polyval(reg,X), alpha=0.3)` pour tracer la droite du modèle de la régression linéaire (`alpha = 0.3` impose une transparence de la droite, pour laisser les données expérimentales bien visibles).

Question 4 : Afficher le graphe avec les barres d'incertitudes et la droite du modèle de la régression linéaire. Afficher la valeur de la pente et de l'ordonnée à l'origine.

[23]:
```
plt.figure()
plt.errorbar(C, A, yerr = u_A, fmt=',', label='données exp')     # Tracé des points␣
↪avec barres d'incertitude
reg = np.polyfit(C, A, 1)                    # Régression linéaire de Y en fonction de␣
↪X (y=ax+b)
plt.plot(C, np.polyval(reg,C), alpha=0.5, label='regression linéaire')   # Tracé de la␣
↪droite de régression, en transparence
plt.xlabel('C ($mol.L^{-1}$)')
plt.ylabel('A' )
```

```
plt.ticklabel_format(axis = 'x', style = 'sci', scilimits= (0,0))  # non necessaire,␣
 ↪pour améliorer la visibilité des graduations en abscisses
plt.title('Évolution de $A$ en fonction de $C$, avec regression linéaire')
plt.grid()
plt.legend()
plt.show()

print('La pente de la droite est', reg[0], '/ L.mol^-1')
print('Ordonnée origine de la droite est', reg[1])
```




```
La pente de la droite est 490.4634146341462 / L.mol^-1
Ordonnée origine de la droite est 0.020213414634146703
```

1.1.5 Validation du modèle

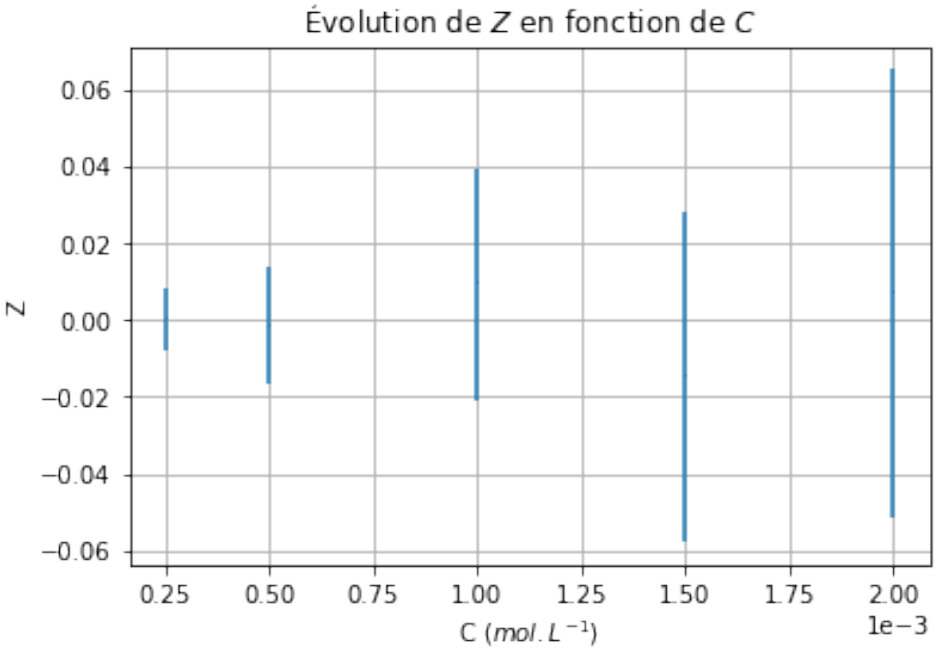
Nous voulons désormais valider (ou invalider) notre modèle (loi de Beer-Lambert) avec ces données expérimentales. Pour cela, nous allons adopter une autre représentation : celle des résidus ou écarts. Il s'agit de représenter l'écart entre les données et le modèle, avec des barres doublées par rapport aux barres d'incertitude.

Question 5 : Nous noterons `Z = Y - np.polyval(reg,X)` ces résidus ou écarts, et on tracera ces résidus avec barres doublées, en fonction de la concentration.

[10]:
```
Z = (A-np.polyval(reg,C))

plt.figure()
plt.errorbar(C, Z, yerr = 2*u_A, fmt=',')      # Tracé des points avec barres␣
 ↪d'incertitude doublées
plt.xlabel('C ($mol.L^{-1}$)')
plt.ylabel('Z' )
```

```
plt.ticklabel_format(axis = 'x', style = 'sci', scilimits= (0,0))  # non necessaire,␣
 ↪pour améliorer la visibilité des graduations en abscisses
plt.title('Évolution de $Z$ en fonction de $C$')
plt.grid()
plt.show()
```



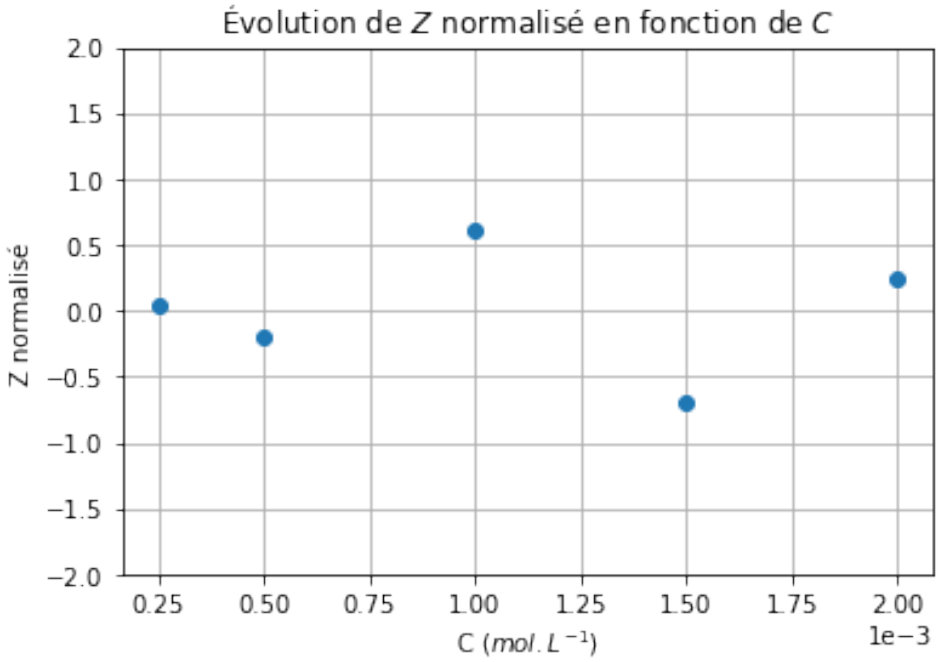
Comment interpréter ce graphe ? Les donnés valident le modèle théorique choisi (ici la loi de Beer-Lambert), dans le cadre de cette l'expérience, si la plupart des valeurs de Z, avec les barres, contiennent 0.

Pour être plus rigoureux : la probabilité qu'une des données de Z, avec ces barres doublées, ne passe pas par 0, est d'au maximum 5% si on considère le modèle valide et que l'expérimentateur a fait fait son travail (mesure et estimation de l'incertitude). Ce doit donc être un évènement rare. Si ce genre d'évènement n'est pas rare, il est peut être judicieux de remettre en cause le modèle dans le cadre de l'expérience faite, ou le travail de l'expérimentateur.

On peut aussi parler de résidus (ou écarts) normalisés : on divise alors Z par l'incertitude-type, et on compare alors les valeurs obtenues à 2 et -2.

[11]:
```
Zn = Z/u_A

plt.figure()
plt.plot(C, Zn,'o' )
plt.ylim(-2,2)
plt.xlabel('C ($mol.L^{-1}$)')
plt.ylabel('Z normalisé' )
plt.ticklabel_format(axis = 'x', style = 'sci', scilimits= (0,0))  # non necessaire,␣
 ↪pour améliorer la visibilité des graduations en abscisses
plt.title('Évolution de $Z$ normalisé en fonction de $C$')
plt.grid()
plt.show()
```



On dira alors que les donnés valident le modèle théorique choisi (ici la loi de Beer-Lambert), dans le cadre de cette l'expérience, si la plupart des valeurs de Z normalisé, sont comprises entre -2 et 2.

1.1.6 Détermination d'une valeur inconnue

Revenons au problème initial : la concentration en ions Fe II pour la solution inconnue. Pour cela, nous allons utiliser le résultat de la régression linaire.

Question 6 : Inverser la relation entre A et C (en utilisant la régression linéaire $y = ax + b$), et donner la concentration de la solution inconnue.

[14]:
```
A_inc = 0.574 # absorbance de la solution inconnue
reg = np.polyfit(C, A, 1)    # Régression linéaire de Y en fonction de X (y=ax+b)
a = reg[0]                   # a pente
b = reg[1]                   # b ordonnée à l'origine
C_inc = (A_inc - b)/a #x = (y-b)/a en inversant la relation affine (régression␣
 ↪linéaire)
print('La concentration en ions Fe II de la solution inconnue␣
 ↪est',round(C_inc*1e3,2),'mmol/L')
```

```
La concentration en ions Fe II de la solution inconnue est 1.13 mmol/L
```

1.2 Détermination statistique d'une grandeur (et incertitudes de type A)

1.2.1 Présentation du problème

On souhaite déterminer la distance focale d'une lentille. Pour cela, sur un banc optique, on repère la position x_A de l'objet, x_O de la lentille et $x_{A'}$ de l"écran sur lequel on obtient une image nette de l'objet.

Comment exploiter ces mesures pour obtenir une valeur de la distance focale et l'incertitude liée ?

Données, disponile dans le fichier *M1_Chimie_exp.py* :

[15]:
```
X_O = np.array([0.394, 0.39, 0.392, 0.393, 0.397,  0.403, 0.41, 0.619, 0.446, 0.402, 0.
↪425])
X_A_prime = np.array([1.497, 1.953, 1.723, 1.483, 1.258, 1.038, 0.9, 0.766, 0.717, 1.
↪069, 0.781])
X_A = 0.287
N = len(X_O) # nombre de couples de mesure

# On convertit les distance en cm, pour une meilleure lisibilité
D_obj =  (-X_O + X_A)*100 # distance objet-lentille en cm
D_ecr =  (X_A_prime-X_O)*100 # distance lentille-écran en cm
```

1.2.2 Détermination de la distance focale

Pour déterminer cette distance focal, on va simplement appliquée la relation de conjugaison de Descartes, qui nous donne :

$$f' = \frac{1}{\frac{1}{\overline{OA'}} - \frac{1}{\overline{OA}}}.$$

Parce qu'on a effectué N observations des couples $(\overline{OA}, \overline{OA'})$, on calcule N valeurs de f'.

Question 7 : Créer le tableau `fi` contenant la distance focale calculée, pour chaque observation, puis en calculer et afficher la moyenne `f` (ce qu'on appelle la valeur mesurée)) grâce à la fonction `np.mean`.

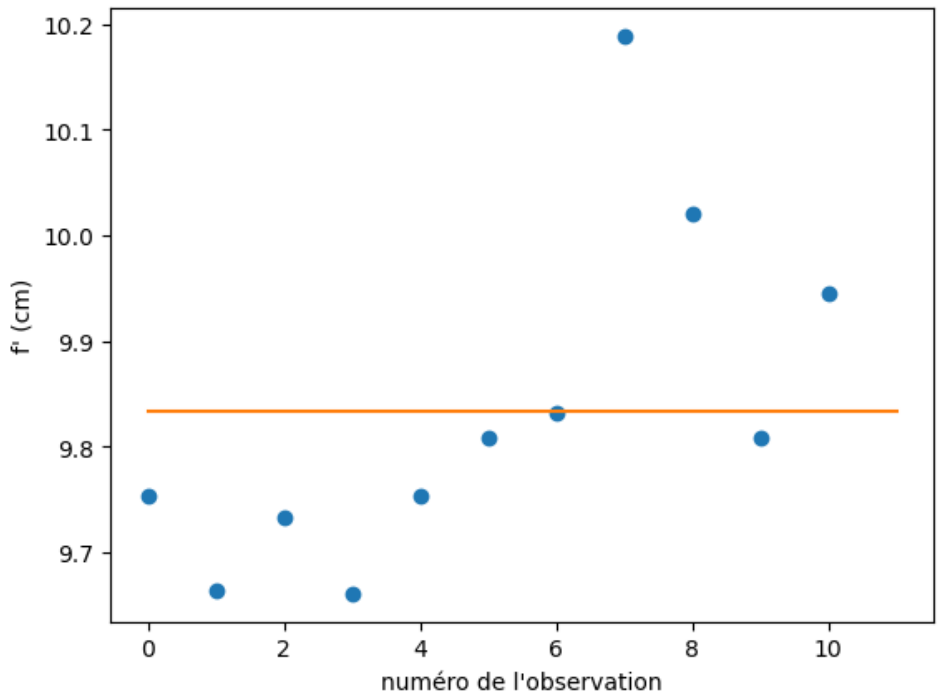
[16]:
```
fi = 1/(1/D_ecr-1/D_obj) # Calcul des distances focales
f = np.mean(fi)          # Calcul de la valeur moyenne
print("f' =",f, "cm")
```

```
f' = 9.833369357860914 cm
```

1.2.3 Variabilité et incertitude-type

Question 8 : Représenter les différentes valeurs calculées de la distance focale, en fonction du numéro de la mesure.

[13]:
```
plt.figure()
plt.plot(fi, 'o') # représentation des valeurs obtenues
plt.plot( (0,N),(f,f) ) # optionnel : tracé de droite représentant la valeur moyenne
plt.xlabel("numéro de l'observation") # légende sur l'axe des abscisses
plt.ylabel("f' (cm)") # légende sur l'axe des ordonnées
plt.show() # affichage de la figure
```



L'incertitude est liée ici au fait qu'il y a plusieurs résultats pour plusieurs observations : on parle de "variabilité" de la mesure. L'incertitude de cette mesure est directement liée à cette variabilité : plus celle-ci est grande, plus l'incertitude doit l'être aussi.

Le lien entre la variabilité des observations et l'incertitude est l'écart-type. La fonction de numpy permettant de calculer cette grandeur est `np.std` (std pour "standart deviation"). Il faut ensuite diviser cette variance par la racine du nombre d'observations pour obtenir l'incertitude-type de la mesure.

Question 9 : Calculer l'incertitude-type sur la série de mesure.

[18]:
```
uf = np.std(fi, ddof=1)/np.sqrt(N) # calcul de l'incertitude-type par une méthode de␣
↪type A pour 7 mesures
print("f' =",f, "cm",  " u(f') =", uf, "cm") # affichage du résultat
```

```
f' = 9.833369357860914 cm  u(f') = 0.04831338944258798 cm
```

On peut se limiter à 1 chiffre significatif sur l'incertitude-type, et prendre le nombre de chiffres significatifs associés pour la valeur de la mesure :

[19]:
```
print("f' =",round(f,2), "cm",  " u(f') =", round(uf,2), "cm") # affichage du résultat
```

```
f' = 9.83 cm  u(f') = 0.05 cm
```

2 Interlude : Quelques bonnes pratiques en programmation...

Avant d'aborder un problème de chimie, profitons-en pour revoir (ou simplement voir) de bonnes habitudes. En accord avec le programme d'informatique, nous allons ajouter aux fonctions une documentation. Commençons par découvrir une telle documentation :

[21]:
```
from math import sqrt
help(sqrt)
```

```
Help on built-in function sqrt in module math:

sqrt(x, /)
    Return the square root of x.
```

Ou :

[22]:
```
sqrt?
```

```
Signature: sqrt(x, /)
Docstring: Return the square root of x.
Type:      builtin_function_or_method
```

Ces documentations sont très utiles lorsqu'on veut utiliser une fonction jusqu'ici inconnue (ou oubliée…). Voyons comment écrire une telle documentation :

[23]:
```
def fonction(arguments):
    """
    Rentrer ici le texte de la documentation
    Quelques idées : dire ce que revoie la fonction, via quelle méthode ou calcul
    On peut aussi ajouter des infos sur les arguments à fournir
    """
```

Autre bonne pratique : pour un problème complexe, il est très utile de décomposer sa résolution à l'aide de plusieurs fonctions, et de tester le **bon fonctionnement de chacune de ces fonctions** à l'aide de multiples tests. Par exemple, dans l'exemple précédent de la fonction racine carré, on peut vérifier son bon fonctionnement avec la valeur de $\sqrt{4}$, ce qu'elle retourne pour des cas limites comme $\sqrt{0}$ ou aussi des cas hors de l'intervalle de définition comme $\sqrt{-1}$. Dans le cas d'intervention du hasard (tirage aléatoire), il est fortement recommandé de répéter de multiples fois ce test, et/ou de tester des valeurs particulières *à la main*.

En vrac, d'autres bonnes pratiques :

- Utiliser des noms de variable avec du sens : par exemple, pour une vitesse, utiliser le nom `vitesse` ou au pire `v` : `x` est evidemment une mauvaise idée. Vouloir ne prendre que des noms de variables constitués d'une seule lettre pour gagner du temps est souvent une bien mauvaise idée, conduisant à des erreurs dans le code parfois chronophages à corriger…
- Prendre des habitudes : personnellement, j'utilise des minuscules pour des nombres, et des majuscules pour des listes ou tableaux. Ainsi je ne confonds pas les deux… (ou rarement…)
- Utiliser les commentaires, introduit par le caractère `#` : cela est utile pour le correcteur, mais aussi pour vous-même (particulièrement lorsqu'on reprend un code une semaine après l'avoir commencé…). Conseil pour les feignants comme moi : plus les noms de variables ont un sens clair, moins les commentaires sont nécessaires.

3 Résolution d'un problème de Chimie

3.1 Equilibre acide-base : résolution exacte

On considère la réaction d'un acide faible avec l'eau :

$$AH^+ + H_2O \leftrightarrow A + H_30^+$$

On note C_0 la concentration initiale en acide ajoutée dans l'eau. On pourra raisonner entièrement en concentrations (avancement volumique). Sinon, on prendra un volume de solution $V = 1L$. Le but de cette partie est de déterminer la concentration en acide à l'équilibre, ainsi que le pH.

Question 10 : Écrire une fonction `eq_acide(C0, pKa)`prenant comme argument la concentration initiale en acide ajoutée dans l'eau et le pKa du couple acide/base AH^+/A, et retournant, par calcul exact, la concentration à l'équilibre de l'acide et le pH de la solution obtenue.

[2]:
```
from math import log10, sqrt

def eq_acide(C0, pKa):

    """ Retourne la concentration en acide C (mol/L) pour un équilibre AH+ + H20 -> A␣
 ↪+ H30+ ainsi que le pH
    Arguments :
    C0 concentration initale en acide (mol/L)
    pKa : pKa du couple AH+ / A"""

    # Constante d'équilibre :
    K = 10**(-pKa)

    # x avancement volumique
    # Discriminant de l'équation x**2 + Kx - KC0 = 0
    Delta = K*(K+4*C0)

    # Résolution :
    x = - K/2 + sqrt(Delta)/2
    # La concentration en H30+ est directement l'avancement volumique :
    pH = - log10(x)
    # La concentration finale en acide est donnée par :
    C = C0 - x

    return C, pH
```

Vérifions le bon fonctionnement :

[3]:
```
C, pH = eq_acide(5e-3, 3.5)
print("Concentration finale en acide : ",C,"mol/L, pH final : ",pH)
```

```
Concentration finale en acide :  0.003890778556394271 mol/L, pH final :
2.954981743178542
```

Pour aller plus loin : Ecrire une fonction permettant de retourner un nombre en écriture scientifique, en choisissant le nombre de chiffres significatifs. L'utiliser alors dans une version améliorée de la fonction `eq_acide`. On pourra utiliser la fonction `round`.

[4]:
```
def ecriture_scientifique(nb, cs):

    """ Converti un nombre en son écriture scientifique (chaîne de caractère), avec cs␣
 ↪nombre max de chiffres significatifs """

    p = 0
```

```
    while nb > 10 :
        p += 1
        nb /= 10

    while nb < 1 :
        p -= 1
        nb *= 10

    nb = round(nb,cs-1)

    return str(nb)+"E"+str(p)
```

[7]:
```
def eq_acide_evol(C0, pKa, cs):

    """ Retourne la concentration en acide C (mol/L) pour un équilibre AH+ + H20 -> A␣
 ↪+ H30+ ainsi que le pH
    Arguments :
    C0 concentration initale en acide (mol/L)
    pKa : pKa du couple AH+ / A
    cs : nombre de chiffres significatifs souhaités

    Calculs exactes
    """

    # Constante d'équilibre :
    K = 10**(-pKa)

    # x avancement volumique
    # Discriminant de l'équation x**2 + Kx - KC0 = 0
    Delta = K*(K+4*C0)

    # Résolution :
    x = - K/2 + sqrt(Delta)/2
    # La concentration en H30+ est directement l'avancement volumique :
    pH = - log10(x)
    # La concentration finale en acide est donnée par :
    C = C0 - x

    return ecriture_scientifique(C,cs), round(pH,cs-1)
```

[8]:
```
C, pH = eq_acide_evol(5e-3, 3.5, 3)
print("Concentration finale en acide : ",C,"mol/L, pH final : ",pH)
```

```
Concentration finale en acide :  3.89E-3 mol/L, pH final :  2.95
```

3.2 Équilibre acide-base : résolution approchée par dichotomie

Dans cette partie, nous allons résoudre ce même problème, mais en utilisant une méthode approchée, en procédant par dichotomie. Pour cela, nous allons introduire la notion d'avancement normalisé x : il s'agit de l'avancement volumique divisé par la concentration initiale en acide (ou aussi l'avancement sur la quantité de matière ajouté initialement en acide). Ainsi :

- Lorsque cet avancement normalisé vaut 0, l'avancement de la réaction vaut 0 : réaction non encore commencée, il s'agit de l'état initial.
- Lorsque cet avancement normalisé vaut 1, l'avancement volumique de la réaction vaut C_0, la réaction est terminée et totale.
- Lorsque cet avancement normalisé prend une valeur intermédiaire, par exemple 0.5, l'avancement volumique de la réaction représente 50% de la concentration initiale en acide, on est à mi-chemin d'une réaction totale.

On peut multiplier cet avancement normalisé par 100, et ainsi l'exprimer en % : à 100%, la réaction est totale (cas d'un acide fort), et sinon la réaction n'est pas totale.

Pour procéder par dichotomie, on partira initialement de l'intervalle de valeurs possibles pour x suivant : $x \in [0, 1]$. Puis, comme précédemment, on divisera cet intervalle en 2 à chaque étape. On s'arrêtera lorsque la longueur de cet intervalle ne dépassera pas une certaine longueur, nommée précision.

Question 11 : Compléter le code de la fonction `eq_acide_dicho(C0, pKa, prec)` prenant comme argument la concentration initiale en acide ajoutée dans l'eau, le pKa du couple acide/base AH^+/A et la précision `prec` longueur maximale finale de l'intervalle des valeurs possibles pour l'avancement normalisé; retournant, par calcul exact, la concentration à l'équilibre de l'acide et le pH de la solution obtenue.

[9]:
```
def eq_acide_dicho(C0, pKa, prec):

    """
    Retourne la concentration en acide C (mol/L) pour un équilibre AH+ + H2O -> A +␣
↪H3O+ ainsi que le pH
    Arguments :
    C0 concentration initale en acide (mol/L)
    pKa : pKa du couple AH+ / A
    pres : précision souhaitée sur l'avancement normalisé, < 1

    Calculs approchés, par la méthode de la dichotomie
    """
    # Constante d'équilibre :
    K = 10**(-pKa)

    # x Avancement "normalisé" (compris entre 0 et 1, 0 pour non réaction, 1 pour␣
↪réaction totale)
    # Encadrement initial de l'avancement normalisé :
    x_inf = 0
    x_sup = 1

    # Valeur milieu, première hypothèse : la réaction est "semi-totale", soit un␣
↪avancement normalisé de 0.5 :
    x = x_inf + (x_sup - x_inf) / 2

    while x_sup-x_inf > prec :
        # Concentration en acide
        C = (1 - x) * C0

        # Quotient réactionnel
        Q = C0*x**2 / (1 - x)
```

```
        # Si Q > K : la réaction est allée trop loin ! Notre valeur de l'avancement␣
↪est surestimée
        if Q > K :
            # On peut améliorer notre encadrement :
            # La nouvelle valeur maximale de l'avancement est celle de notre hypothèse␣
↪:
            x_sup = x
            # La valeur inférieure ne change pas

        # Sinon, la réaction va aller plus loin, notre valeur de l'avancement est␣
↪sous-estimée
        else :
            # On peut améliorer notre encadrement :
            # La nouvelle valeur minimale de l'avancement est celle de notre hypothèse␣
↪:
            x_inf = x
            # La valeur supérieure ne change pas

        # Nouvelle valeur hypothétique de l'avancement, au milieu de l'encadrement :
        x = x_inf + (x_sup - x_inf) / 2

    C = (1 - x) * C0
    pH = - log10(x * C0)

    return C, pH
```

Vérifions le fonctionnement :

[10]:
```
C, pH = eq_acide_dicho(5e-3, 3.5, 0.0001)
print("Concentration finale en acide : ",C,"mol/L, pH final : ",pH)
```

```
Concentration finale en acide :  0.003890838623046875 mol/L, pH final :
2.9550052617665306
```

L'intérêt de cette méthode *approchée*, par rapport à la méthode *exacte* précédente, est qu'elle peut s'appliquer à toute réaction, dont certaines bien plus complexes. Pour de telles réactions, la résolution exacte par le calculs peut être inaccessible… (il ne s'agit pas toujours de rechercher les racines d'un simple polynôme de degré 2 !).