

## Activité 4

---

# Avant d'aller plus loin...

---

Toujours là après ces trois intenses chapitres de révisions... surtout pour celles et ceux qui avaient un peu de mal.

Pour l'instant vous avez réalisé des programmes relativement courts qui ne nécessitaient pas forcément une grande rigueur dans l'organisation du programme. Avant d'aborder des programmes plus complexes, j'espère vous apporter, ici, quelques conseils de vieux singe !

### 1

### Structure générale d'un programme

Voici un programme qui fait quelque chose... Vous serez capable de le faire avec vos propres mimines d'ici quelques semaines !

```
1  """ importation de bibliothèques """
2  import matplotlib.pyplot as plt
3  from math import pow# on peut aussi utiliser a**b au lieu de
   pow(a,b)
4
5  """
6      constantes (variables) globales nécessaires à
       l'ensemble du programme
7  """
8  Ke = pow(10,-14)
9  BBT = [7.0,250,254,199,218,254,239]#pka rgb acide rgb basique
10 HEL = [3.7,255,149,149,254,208,146]
11 PHIPHI = [9.6,255,240,252,254,171,230]
12 Indicateur = ["Hélianthine", "Bleu de bromothymol",
   "Phénol-phtaléine"]
13 ChoixIndicateur=[HEL, BBT, PHIPHI]
14
15 """
16     routines utilitaires
17 """
18 def f(h,C0,V0,Ct,Vt,Ka): # commentaire supprimé pour vous
   demandez ce que fait la fonction f ?
19     return (h - Ke/h)*(V0 + Vt) + Ct*Vt - C0*V0/(1 + h/Ka)
20
21 def zorglub(C0,V0,Ct,Vt,Ka): # commentaire supprimé pour vous
   demandez ce que fait la fonction zorglub ?
22     a=0
23     b=14
24     fa = f(pow(10,-a), C0, V0, Ct, Vt, Ka)
```

```

25     while b - a > 0.01 :
26         c = (a+b)/2
27         fc = f(pow(10,-c), C0, V0, Ct, Vt, Ka)
28         if (fa *fc < 0):
29             b = c
30         else:
31             a = c
32             fa = fc
33     return (a+b)/2
34
35     """
36     calcul des 3 composantes de la couleur pour l'indicateur
37     coloré pour un pH donné par un modèle simple à expliciter
38     """
39 def couleur(Indicateur, pH): #RGB de l'indicateur coloré pour
40     un mélange donné
41     h = pow(10,-pH)
42     Ka = pow(10,-Indicateur[0])
43     pctA = 100/(1 + Ka/h)
44     pctB = 100/(1 + h/Ka)
45     r = (Indicateur[1]*pctA + Indicateur[4]*pctB)/100
46     g = (Indicateur[2]*pctA + Indicateur[5]*pctB)/100
47     b = (Indicateur[3]*pctA + Indicateur[6]*pctB)/100
48     return (r/256,g/256,b/256)
49
50     """
51     programme principal
52     """
53 V = [] # tableau de valeur pour stocker les volumes
54 PH = [] # tableau de valeur pour stocker les pH correspondants
55 v = 0 # volume "courant"
56 Ca = 0.01 #concentration en acide titré
57 Va = 100 #volume titré
58 Cb = 0.1 #concentration en base
59 pas = 0.01 #pas d'incrémentement du volume
60 Vmax = 25 # volume maximal de réactif titrant
61
62 while v <= Vmax :
63     V.append(v)
64     ppH = zorglub(0.01, 100, 0.1, v, pow(10,-4))
65     PH.append(ppH)
66     v = v + pas
67
68 # on affiche une première fois la courbe
69 plt.plot(V, PH)
70 plt.show()
71
72 # on demande le choix de l'indicateur
73 reponse = input("Choix de l'indicateur coloré : 0 pour
74     hélianthine, 1 pour bleu de bromothymol ou 2 pour phénol
75     phtaléine) ?")
76 choix = int(reponse)
77
78 #on affiche la courbe si le choix est valide

```

```

75 | if choix > 2 :
76 |     print("Il fallait taper un nombre entre 0 et 1 !")
77 | else:
78 |     for i in range(len(V)-1):
79 |         plt.fill([V[i], V[i+1], V[i+1], V[i]], [0,0,12,12],
                    color = couleur(ChoixIndicateur[choix],
                    (PH[i]+PH[i+1])/2), alpha = 0.5)
80 |     plt.plot(V, PH)
81 |     plt.xlabel("Volume (mL)")
82 |     plt.ylabel("pH")
83 |     plt.title("Indicateur utilisé : " + Indicateur[choix])
84 |     plt.show()

```

A chacun son organisation, en ce qui me concerne j'adopte systématiquement la présentation suivante. Je propose :

- de commencer par toutes l'importation des différentes bibliothèques (lignes 2 et 3);
- de définir tout ce qui est constante globale du système que l'on n'a pas à modifier d'une modélisation à une autre (lignes 8 à 13);
- on écrit ensuite toutes les fonctions nécessaires (lignes 18 à 47), si possible dans l'ordre dans lesquelles on en a besoin;
- on passe ensuite au corps du programme. Je commence, là encore par toutes les variables (globales) que l'on va exploiter par la suite et qui ont vocation à être, éventuellement modifiées (lignes 51 à 58).
- vient, enfin, le corps du programme!

## 2

## Gestion des variables



Ne sous-estimez pas, dans vos programmes, la gestion des variables!

Vous n'êtes pas obligés de tout comprendre ce qui suit (en particulier la partie 2.) mais il faut avoir conscience que l'on ne peut pas faire n'importe quoi avec les variables.

### 2.1

### Variables globales, locales

Dans un programme, on peut très bien se trouver dans le cas de figure suivant :

- une fonction est définie par « *def maFonction(x) :* »;
- quelque part on attribue à *x* une certaine valeur

Que se passe-t-il quand on appelle *maFonction* ?

```

1 | x = 2
2 |
3 | def maFonction(x):
4 |     return x*x
5 |
6 | y = maFonction(x)
7 | z = maFonction(3)
8 | print("x : ", x, " y : ", y, " z : ", z)

```

Et bien... tout se passe bien car à l'intérieur de la fonction *maFonction*, *x* apparaît comme une variable **locale**. Cette variable a sa propre zone de stockage en mémoire, indépendamment de celle de la variable **globale** *x* définie dans le cœur du programme.

Pour éviter des conflits entre les différentes variables (on parle d'effet de bord) ; une fonction ne peut pas modifier de variable globale sauf si on spécifie explicitement que c'est une variable globale. Que va-t-on avoir comme impression en exécutant le programme suivant ?

```

1  a = 3#variable globale
2
3  def fonction1(x):
4      y = a * x #on utilise la variable globale
5      print("Dans la fonction1 a = ", a)
6      return y
7  def fonction2(x):
8      a = 2 * x #a est redéfinie comme variable locale, x en est
          une aussi
9      print("Dans la fonction2 a = ", a)
10     return x*x
11  def fonction3(x):
12     global a# on précise que l'on veut utiliser la variable
          globale
13     a = 10 * x#ce qui permet de la modifier... à nos risques
          et périls
14     print("Dans la fonction3 a = ", a)
15     return x*x
16
17  print("Au départ : a = ", a)
18  y = fonction1(4)
19  print("Après fonction1 : a = ", a, " y = ", y)
20  y = fonction2(4)
21  print("Après fonction2 : a = ", a, " y = ", y)
22  y = fonction3(4)
23  print("Après fonction3 : a = ", a, " y = ", y)

```

La fonction *fonction3* est « dangereuse » car elle modifie la valeur d'une variable globale... en principe, seul le programme devrait être autorisé à le faire!!!

## 2.2 Variables mutables ou non mutables

Quesako ?

Un objet **mutable** est un objet que l'on peut modifier après sa création. Lorsqu'on modifie une liste, la liste est modifiée sans que sa place en mémoire change.

Un objet **non mutable** ne peut être modifié. Si on le modifie, on crée en fait une nouvelle instance de la variable à un nouvel emplacement mémoire. Les entiers, réels, chaînes de caractères sont non mutables.

Si on ne comprend pas tout, ce n'est pas grave mais il faut avoir, peut-être, conscience des conséquences.

```

1  a = 2
2  print("Avant : a=",a, " à l'adresse : ", id(a))
3  a = 4
4  print("Après : a=",a, " à l'adresse : ", id(a))
5
6  u = [4,3,2,1]

```

```

7 print("Avant : u=",u,"à l'adresse : ", id(u))
8 u.append(5)
9 print("Après : u=",u,"à l'adresse : ", id(u))
10
11 ch = "abcde"
12 print("Avant : ch=",ch,"à l'adresse : ", id(ch))
13 ch = ch + "f"
14 print("Après : ch=",ch,"à l'adresse : ", id(ch))

```

Lors de l'exécution du script précédent, on obtient les résultats suivants<sup>1</sup>.

Avant : a= 2 à l'adresse : 4297327264

Après : a= 4 à l'adresse : 4297327328

Avant : u= [4, 3, 2, 1] à l'adresse : 4573137736

Après : u= [4, 3, 2, 1, 5] à l'adresse : 4573137736

Avant : ch= abcde à l'adresse : 4576922624

Après : ch= abcdef à l'adresse : 4567965624

L'adresse de *a* ou *ch* a changé, mais pas celle de *u*

#### a Conséquences sur l'affectation $x = y$

D'où piège lors de l'exécution du programme suivant :







```

1 a = 2
2 b = a
3 print("Avant : a=",a,"b=",b)
4 b = 3
5 print("Après : a=",a,"b=",b)
6
7 u = [4,3,2,1]
8 v = u
9 print("Avant : u=",u,"v=",v)
10 v[2] = 5
11 print("Après : u=",u,"v=",v)
12 v = [1,2,3,4]
13 print("Enfin : u=",u,"v=",v)

```

Aucun souci pour les types entier, réel, booléen, chaîne de caractère ; mais pour les listes l'instruction  $v = u$  crée une nouvelle variable *v* qui pointe vers le même contenu mémoire que la variable *u*. Ces variables sont mutables, donc quand on modifie l'une d'elle, on n'en crée pas une nouvelle copie quelque part dans la zone mémoire mais on modifie le contenu de la zone mémoire commun aux deux variables.

1. L'instruction *id()* retourne l'adresse mémoire d'une variable donnée

Variable et zone mémoire					
variable non mutable			variable mutable		
booléen, entier, réel, chaîne de caractères			liste		
Instruction :	Variable(s) :	Zone mémoire :	Instruction :	Variable(s) :	Zone mémoire :
<code>a = 2</code>	<code>a</code>		<code>u = [4,3,2,1]</code>	<code>u</code>	
<code>b = a</code>	<code>a</code> <code>b</code>		<code>v = u</code>	<code>u</code> <code>v</code>	
<code>b = 3</code>	<code>a</code> <code>b</code>		<code>v[2] = 5</code>	<code>u</code> <code>v</code>	

C'est grave, docteur ?

Un peu quand même ; pour y remédier il faut remplacer l'instruction `v = u` :

- soit par une copie dans le vecteur `v` de tous les éléments de `u` un par un
- soit, si on vous la donne, par l'utilisation de l'instruction `copy()` qui crée une nouvelle variable dans une nouvelle zone mémoire pour y stocker le contenu souhaité.

```

1 u = [4,3,2,1]
2 v = [x for x in u]
3 print("Avant : u=",u,"v=",v," aux adresses", id(u),"et",id(v))
4 v[2] = 5
5 print("Après : u=",u,"v=",v," aux adresses", id(u),"et",id(v))
6
7 u = [4,3,2,1]
8 v = u.copy()
9 print("Avant : u=",u,"v=",v," aux adresses", id(u),"et",id(v))
10 v[2] = 5
11 print("Après : u=",u,"v=",v," aux adresses", id(u),"et",id(v))

```

A l'exécution `u` et `v` ne pointent pas vers la même zone mémoire et on peut donc modifier l'un sans modifier l'autre.

Avant : `u= [4, 3, 2, 1]` `v= [4, 3, 2, 1]` aux adresses 4585847560 et 4571253896

Après : `u= [4, 3, 2, 1]` `v= [4, 3, 5, 1]` aux adresses 4585847560 et 4571253896

Avant : `u= [4, 3, 2, 1]` `v= [4, 3, 2, 1]` aux adresses 4579894088 et 4585847560

Après : `u= [4, 3, 2, 1]` `v= [4, 3, 5, 1]` aux adresses 4579894088 et 4585847560

Remarque : dans le listing ci-dessus, on utilise une liste par compréhension, on pourrait tout aussi bien écrire une des solutions du code suivant :

```

1 v = [u[i] for i in range(len(u))]
2 # ou
3 v = []
4 for uu in u :
5     v.append(uu)
6 # ou
7 v = []
8 for i in range(len(u)):
9     v.append(u[i])

```

Nous ne reviendrons plus sur ces différentes alternatives par la suite.

## **b** Conséquences sur les paramètres d'une fonction

Le fait d'accéder aux listes par leur adresse mémoire et non pas directement par leur valeur a également quelques conséquences (qui peuvent être bénéfiques !) lors du passage de paramètre à une fonction.

Qu'obtient-on à l'exécution du programme suivant ?

```
1 def permuteNombre(x,y):
2     print("Début de la fonction : x = ", x, "y = ", y)
3     temp = x
4     x = y
5     y = temp
6     print("Fin de la fonction : x = ", x, "y = ", y)
7
8 def permuteListe(T):
9     print("Début de la fonction : T = ", T)
10    temp = T[0]
11    T[0] = T[1]
12    T[1] = temp
13    print("Fin de la fonction : T = ", T)
14
15 a = 2
16 b = 3
17 print("Avant : a = ", a, "b = ", b)
18 permuteNombre(a,b)
19 print("Après : a = ", a, "b = ", b)
20
21 u = [2,3]
22 print("Avant : u = ", u)
23 permuteListe(u)
24 print("Après : u = ", u)
```

Avant : a = 2 b = 3

Début de la fonction : x = 2 y = 3

Fin de la fonction : x = 3 y = 2

Après : a = 2 b = 3

Avant : u = [2, 3]

Début de la fonction : T = [2, 3]

Fin de la fonction : T = [3, 2]

Après : u = [3, 2]

On constate :

- Que les valeurs des variables non mutables ne sont pas changées dans le corps du programme (même si on appelait *a* et *b* le nom des variables dans la fonction *permuteNombre* car ce sont des variables « muettes »). Pour se faire, il faudrait rajouter *return x, y* à la fin de la fonction *permuteNombre* et faire l'appel de la fonction sous la forme *a, b = permuteNombre(a, b)*.
- Que le tableau *u* est changé dans le corps du programme après l'appel de la fonction *permuteListe*. En fait, on a passé à la fonction non pas directement la liste mais l'adresse de la liste et, dans la fonction, on travaille sur l'adresse de cette liste.

Par conséquent, lorsqu'on travaille sur une liste ou un tableau :

- ce n'est pas nécessaire de faire un *return* pour récupérer le tableau modifié dans le corps du programme ;
- en contre-partie, il faut faire attention car si on modifie le tableau dans la fonction, le tableau est également modifié dans le corps du programme...

### 3 Errare humanum est...

#### 3.1 Erreurs lors de l'édition de code

Lors de l'écriture de programmes, on ne peut pas ne pas faire d'erreurs ou d'étourderies de syntaxe. Dans la console un message tente de nous expliquer l'origine mais, ce n'est pas toujours évident.

Au bout d'un certain temps, vous arriverez facilement à corriger vos erreurs, par contre corriger celles de vos élèves peut être un peu plus délicat !

Quelques messages sont très explicites :

Code	Message d'erreur
<code>a = ln(2+3*(4-5))</code>	<code>NameError : name 'ln' is not defined</code>
<code>a = u + log(2+3*(8-5))</code>	<code>NameError : name 'u' is not defined</code>
<code>a = log(2+3*(4-5))</code>	<code>ValueError : math domain error</code>
<code>a = log(2+3*(8-5)))</code>	<code>SyntaxError : invalid syntax</code>
<code>print(T[3])#avec T=[1,2,3]</code>	<code>IndexError : list index out of range</code>

d'autres un peu moins !

Code	Message d'erreur
<code>a = log(2+3*(8-5)(2+8))</code>	<code>TypeError : 'int' object is not callable</code>

'machin' object is not collable est une erreur très fréquente ; en gros on essaie de passer des paramètres à machin qui n'est pas une fonction... on cherche un peu et c'est la signature de l'oubli d'une opération entre 2 parenthèses !

Et enfin, une autre erreur très fréquente :

```
1 R = 8,31
2 RT = R*(25+273)
3 a = RT/1000
4 print(a)
```

`TypeError : unsupported operand type(s) for / : 'tuple' and 'int'`

Quand on a `unsupported ... 'tuple'` dans le même message d'erreurs, c'est que l'on a utilisé la virgule ou lieu du point comme séparateur décimal.

Dernière remarque enfin, le message d'erreur apparaît lorsque l'interpréteur ne comprend plus rien ; il se peut que l'erreur provienne de la ligne précédente. C'est le cas, en particulier pour l'oubli de parenthèse fermante.

#### 3.2 Utilisation du débogueur

Une fois éliminées toutes les erreurs de syntaxe, on peut exécuter le programme. Il se peut cependant que ce dernier ne donne pas les résultats escomptés. Une phase de débogage peut alors être nécessaire.



Une façon élémentaire de déboguer le programme est de mettre des `print(nomdelavariable)` un peu partout pour savoir ce que vaut effectivement la variable qui nous intéresse.

Si l'on est dans le corps du programme, l'onglet « Variable Explorer » contient les différentes variables globales utilisées. Mais, par exemple, si elle se trouve dans une boucle, on ne peut pas savoir ce qu'elle vaut pour chaque étape.

La solution la plus efficace est d'utiliser le débogueur de spyder. Pour son utilisation la plus simple, on fait un double clique dans la colonne de gauche au niveau d'une ligne qui nous intéresse. On insère alors un point d'arrêt. Au lieu de cliquer sur « Run » on clique sur « Debug ». Le programme tourne et s'arrête à la ligne souhaitée.

On peut ensuite continuer l'exécution ligne par ligne, jusqu'à un autre point d'arrêt... et à chaque fois, dans la console on peut introduire le nom d'une variable à droite du prompteur `ipdb>` pour connaître sa valeur.

### 3.3 Gestion des erreurs lors de l'exécution du programme

Après, le programme peut fonctionner mais que se passe-t-il si on n'introduit pas des données cohérentes ? On risque, par exemple, une division par zéro, des opérations sur des listes de tailles non adaptées...

La gestion de telles erreurs lors de l'exécution d'un programme est extrêmement complexe. Ici, nous n'allons jamais nous en occuper.

Si on souhaite un minimum de contrôle, on peut utiliser l'expression `assert`.

Par exemple, dans le programme sur les réactions successives du chapitre précédent, on ne peut pas faire de simulation pour  $k_1$  et  $k_2$ . On pourrait écrire :

```
1 def B(k1,k2,t):
2     assert k1 != k2
3     return (A0*k1/(k2-k1))*(exp(-k1*t)-exp(-k2*t))
```

si on appelle la fonction avec  $k_1 = k_2$ , le programme stoppe et apparaît « AssertionError » dans la console.

## 4 A vous de jouer...

Il ne s'agit pas, pour l'instant, ici de se lancer dans la réalisation d'un grand programme. On peut tenter toutefois de réinvestir quelques outils...

### 4.1 Exploitation de résultats expérimentaux

Supposons que vous disposez d'un fichier Excel (ou assimilé) contenant quelques données expérimentales ; par exemple, la mesure du pH et de la conductance lors d'un titrage de 50 mL d'un mélange d'acide chlorhydrique et de chlorure de magnésium. Vous enregistrez le fichier au format csv (en prenant soin de remplacer les « , » par des « . »).

On obtient, par exemple dans le fichier `MgCl2HCl.txt` les données suivantes :

```
V;pH;g (mS)
0;1.69;8.36
0.5;1.71;7.99
1;1.74;7.54
1.5;1.76;7.26
2;1.78;6.96
...
```

Le code suivant permet d'ouvrir le fichier texte et de récupérer son contenu dans la variable *lignes* sous forme d'une liste de chaînes de caractères, chacune d'elle correspond à une ligne.

```
1 fichier = open("MgCl2HCl.txt")
2 lignes = fichier.readlines()
3 fichier.close()
```

Malheureusement, les caractères de fin de ligne ne sont pas les mêmes selon le système d'exploitation et il se peut que dans la liste de chaînes on ait une chaîne vide sur deux. Si *l* est une des chaînes de caractères de la liste *lignes*, on pourra tester sa longueur à l'aide de l'instruction `if len(l)>0` :  
# la ligne *l* est non vide.

#### a Extraction des données et tracé des courbes

Exploiter le fichier donné afin de tracer, sur le même graphe, les courbes  $\text{pH} = f(V)$  et  $g = g(V)$  représentées en annexe du chapitre 3. La trame du programme est donnée, attention à bien placer le fichier de données dans le même dossier que votre programme python.

#### b Exploitation du titrage conductimétrique

Superposer sur le même graphe le tracé, en fonction du volume, de la conductivité et de la conductivité corrigée de la dilution.

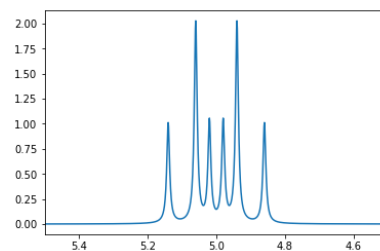
Faire, à l'aide de la routine créée dans le chapitre précédent, trois régressions linéaires pour des volumes compris entre 0 et 8 ; 10 et 12 et 15 et 20. Programmer le calcul et l'affichage des volumes équivalents.

### 4.2 Doublet de triplet ou triplet de doublet ?

Allez, on complique un peu<sup>2</sup> !

L'objectif est de simuler l'allure d'un signal RMN d'un proton soumis à un couplage de type AMX.

Pour tracer un spectre entre *deltaMin* et *deltaMax* dans le bon sens, il suffit de spécifier les valeurs à l'aide de l'instruction : `plt.xlim(deltaMax,deltaMin)`.



On peut représenter un pic unique sous forme du lorentzienne : 
$$L(\delta) = \frac{I}{1 + \left( \frac{\delta - \delta_{ref}}{5\Delta\delta} \right)^2}$$
 où  $\delta_{ref}$

est le déplacement chimique de référence du signal, *I* son intensité.  $\delta$  est le déplacement chimique en cours et  $\Delta\delta$  l'écart entre deux points tracés ; le 5 est là pour avoir un pic assez large.

Le nombre de combinaisons de *n* éléments pris *p* à *p* vaut :  $\binom{n}{p} = C_p^n = \frac{n!}{p!(n-p)!}$ .

2. Euh... beaucoup peut-être ? Vous me direz quoi !

## 5 Quelques pistes. . .

### 5.1 Exploitation de résultats expérimentaux

Ce programme ne devrait pas poser de difficultés majeures. Il faudra gérer la première ligne du fichier ainsi que les éventuelles lignes vides !

### 5.2 Doublet de triplet ou triplet de doublet ?

Bon, là, c'est plus difficile.

Commençons par la fin. . .

- Pour tracer la courbe il nous faut connaître la valeur de  $\delta_{ref}$  et l'intensité de chacun des pics du massif.
- Le plus délicat est de déterminer le déplacement chimique et l'intensité de chacun de ces pics. Une piste de résolution pourrait être de coder une fonction `genereFils(deltaRef, intensiteRef, nbFils, J)`. Etant donné un pic de référence de déplacement chimique *deltaRef*, d'intensité *intensiteRef* couplé avec (*nbFils*-1) protons avec une constante de couplage *J*, cette fonction doit retourner deux listes, l'une correspondant aux déplacements chimiques des différents « fils » et l'autre aux intensités de ces signaux.
- Après, il suffit d'essayer, dans le corps du programme principal de programmer ce que vous tracez au tableau devant vos chers élèves !  
Un premier appel de la fonction `genereFils` doit permettre de gérer le couplage entre A et M ; « reste » ensuite à découpler chacun des signaux précédents par un nouvel appel à la fonction `genereFils` pour tenir compte du couplage entre A et X.
- Quand on a *n* pics, les coefficients du triangle de PASCAL valent  $\binom{p}{n}$  avec *p* variant de 0 à *n* - 1.

## 6 Solutions...

### 6.1 Exploitation de résultats expérimentaux

#### a Extraction des données

```

1 import matplotlib.pyplot as plt
2 import utilitaire as util
3
4 fichier = open("MgCl2HCl.txt")
5 lignes = fichier.readlines()
6 fichier.close()
7
8 V=[]
9 PH=[]
10 G=[]
11 first = True
12 for l in lignes :
13     if first :
14         first = False
15     else :
16         if len(l) >0 :
17             ls = l.split(";")
18             V.append(float(ls[0]))
19             PH.append(float(ls[1]))
20             G.append(float(ls[2]))

```

Quelques commentaires sur cette première partie du programme.

- On extrait les lignes une à une (inutile d'avoir leur indice, mais ce n'est pas interdit !)
- Il faut tester si ce n'est pas la première car, c'est du texte !
- Pour une ligne qui contient des données, il faut tester que ce n'est pas une ligne vide (ligne 16).
- Ligne 17, on coupe la chaîne de caractères *l* en sous chaînes de caractères. Le séparateur est ici le point-virgule. *ls* est une liste de chaînes de caractères.
- Il faut donc faire une conversion avec `float` avant de stocker les bonnes valeurs dans les bons tableaux.

La suite a été décrite dans l'annexe du chapitre 3.

#### b Exploitation du titrage conductimétrique

```

1 V0 = 50
2 Gc=[]
3 for i in range(len(V)):
4     Gc.append(G[i]*(V0+V[i])/V0)
5
6 plt.plot(V,G,"r+")
7 plt.plot(V,Gc,"bx")
8 plt.show()
9
10 a1,b1,r1 = util.regrelinEntre(V,Gc,0,8)

```

```
11 print(a1,b1)
12 a2,b2,r2 = util.regrelinEntre(V,Gc,10,12)
13 print(a2,b2)
14 a3,b3,r3 = util.regrelinEntre(V,Gc,15,20)
15 print(a3,b3)
16
17 Veq1 = (b1 - b2)/(a2 - a1)
18 Veq2 = (b2 - b3)/(a3 - a2)
19 print("Premier volume équivalent : ", round(Veq1,1), " mL")
20 print("Second volume équivalent : ", round(Veq2,1), " mL")
```

Peu de choses à préciser ici en principe.

- On commencer par créer le tableau de valeur correspondant à la conductivité corrigée, et on trace les courbes.
- Si on a bien travaillé la semaine dernière, on a une routine `regrelinEntre` dans une bibliothèque qui permet de faire une régression linéaire sur les différentes parties de la courbes.
- Bon, bien sûr, il faut penser à stocker les valeurs de retour des fonctions pour les exploiter par la suite!

## 6.2 Doublet de triplet ou triplet de doublet ?

```

1 import matplotlib.pyplot as plt
2
3 nuSpectro = 100 # fréquence du spectro en MHz
4 deltaA = 5 # déplacement chimique de A
5 nbM = 1 # nombre de protons de type M
6 JAM = 12 # constante de couplage AM
7 nbX = 2 # nombre de protons de type X
8 JAX = 8 # constante de couplage AX
9
10 def lorentz(delta, intensite, deltaRef, deltappm):
11     return intensite / (1 + ((delta - deltaRef) / (5*
12         deltappm) )**2)
13
14 def factorielle(n):
15     if n == 0 :
16         return 1
17     else :
18         return n*factorielle(n-1)
19
20 def combinaison(n,p):
21     return factorielle(n)/(factorielle(p)*factorielle(n-p))
22
23 def genereFils(deltaRef, intensiteRef, nbFils, J):
24     Delta = []
25     I = []
26     d = J/nuSpectro
27     nI = 0
28     for i in range(0,nbFils):
29         nI = nI + combinaison(nbFils-1, i)
30         if nbFils %2 == 0 :
31             d0 = (d/2) + ((nbFils/2)-1)*d
32         else :
33             d0 = ((nbFils-1)/2)*d
34         Delta.append(deltaRef - d0 + i*d)
35         I.append(intensiteRef*combinaison(nbFils-1, i))
36     return Delta, I
37
38 """ programme principal """
39
40 deltaPic = []
41 intensitePic = []
42 dp, ip = genereFils(deltaA, 1, nbM+1, JAM)
43 for i in range(len(dp)):
44     dp2, ip2 = genereFils(dp[i], ip[i], nbX+1, JAX)
45     for j in range(len(dp2)):
46         deltaPic.append(dp2[j])
47         intensitePic.append(ip2[j])
48
49 deltaMin = deltaA-0.5
50 deltaMax = deltaA+0.5
51 deltappm = 0.001

```

```

52 delta = deltaMin
53 D = []
54 I = []
55 while delta < deltaMax :
56     D.append(delta)
57     intensite = 0
58     for i in range(len(deltaPic)):
59         intensite = intensite + lorentz(delta,
60                                         intensitePic[i], deltaPic[i], deltappm)
61     I.append(intensite)
62     delta = delta + deltappm
63 plt.plot(D, I)
64 plt.xlim(deltaMax, deltaMin)
65 plt.show()

```

Quelques remarques sur ce programme :

- Le code de la fonction `lorentz` ne devrait pas poser de problèmes.
- J'ai écrit, pour le fun, la fonction `factorielle` sous forme récursive ; une programmation itérative aurait été tout à fait possible.
- Attention aux parenthèses au dénominateur dans la fonction `combinaison` !
- Dans la fonction `genereFils` le plus délicat est de s'assurer que les paramètres transmis à la fonction `combinaison` sont bien ceux qui correspondent au triangle de PASCAL (lignes 27 et 28).  
La position du pic le plus à droite ne se calcule pas de la même façon si le nombre de fils est pair ou impair ; d'où le test ligne 29. Ensuite, il suffit d'incrémenter pour avoir les abscisses des différents pics (lignes 33 à 35).
- Dans le programme principal, on commence par générer les « fils » du pic principal (ligne 42). On récupère deux listes *dp* et *ip* correspondant, respectivement, aux déplacements chimiques et aux intensités des pics issus du premier couplage entre A et M. Ces données ne sont, bien sûr, pas stockés dans les listes définitives *deltaPic* et *intensitePic*.
- Puis, pour chacun des pics précédents, il faut générer les « fils » en tenant compte du couplage entre A et X (ligne 44). Cette fois, pour chacun des appels, il faut stocker les valeurs dans les listes *deltaPic* et *intensitePic*.
- pour le tracer graphique, on somme les lorentziennes correspondant à chacun des pics du tableau *deltaPic* avec l'intensité correspondante.
- ... pas simple tout ça quand même !