



Python pour les Blondes

Quelques éléments de programmation et modélisation de systèmes chimiques

Alain Demolliens

1	On commence en douceur ?	6
1	Quelques rappels « théoriques »	6
1.1	Types de données	6
1.2	Un premier mot sur les variables	6
1.3	Entrées/sorties élémentaires	8
1.4	Avec des si	8
1.5	Retour sur . . . les boucles	11
1.6	While ou for ?	14
2	A vous de jouer	14
2.1	Blondes	15
2.2	Châtain clair	15
2.3	Brunes ou chauves !	16
3	Quelques pistes	17
3.1	Blondes	17
3.2	Châtain clair	17
3.3	Brunes ou chauves !	18
4	Solutions !	19
4.1	Blondes	19
4.2	Châtain clair	19
4.3	Chifoumi	21
5	Annexes	24
5.1	Environnement de développement Sypder	24
5.2	Importation de bibliothèques	25
2	Indispensables listes . .	27
1	Quelques rappels « théoriques »	27
1.1	Listes python	27
1.2	Accès aux éléments d'une liste	27
1.3	Création d'une liste	29
1.4	Pas d'opération arithmétiques sur les listes !	30
1.5	Quelques mots sur NumPy	30
2	A vous de jouer	30
2.1	Blondes	30
2.2	Châtain clair	30
2.3	Réactions successives (Acte I) !	31
3	Quelques pistes	32
3.1	Blondes	32
3.2	Châtain clair	32
3.3	Réactions successives	32
4	Solutions	33
4.1	Blondes	33

4.2	Châtain clair...	34
4.3	Réactions successives	36
3	Fonctions, tracé de courbes	38
1	Quelques rappels « théoriques »...	38
1.1	Fonctions et procédures	38
1.2	Application : tracé de courbes	39
1.3	Application : fonction récursive	40
2	A vous de jouer...	40
2.1	Blondes	40
2.2	Châtain clair...	40
2.3	Réactions successives (Acte II)	41
2.4	Régression linéaire	41
3	Quelques pistes...	43
3.1	Blondes	43
3.2	Châtain clair...	43
3.3	Réactions successives	43
3.4	Régression linéaire	43
4	Solutions...	44
4.1	Blondes	44
4.2	Châtain clair...	44
4.3	Triangle de PASCAL	44
4.4	Réactions successives	45
4.5	Régression linéaire	47
5	Annexe : bibliothèque matplotlib	49
4	Avant d'aller plus loin...	52
1	Structure générale d'un programme	52
2	Gestion des variables	54
2.1	Variables globales, locales	54
2.2	Variables mutables ou non mutables	55
3	Errare humanum est...	59
3.1	Erreurs lors de l'édition de code	59
3.2	Utilisation du débogueur	59
3.3	Gestion des erreurs lors de l'exécution du programme	60
4	A vous de jouer...	60
4.1	Exploitation de résultats expérimentaux	60
4.2	Doublet de triplet ou triplet de doublet ?	61
5	Quelques pistes...	62
5.1	Exploitation de résultats expérimentaux	62
5.2	Doublet de triplet ou triplet de doublet ?	62
6	Solutions...	63
6.1	Exploitation de résultats expérimentaux	63
6.2	Doublet de triplet ou triplet de doublet ?	65
5	Courbes de répartition	67
1	A vous de jouer	67
1.1	Courbes de répartition d'un diacide	67
1.2	To be or not to be!	67
1.3	Courbes de répartition d'un polyacide	68
1.4	MLL's Challenge	68
2	Quelques pistes...	70

2.1	Courbes de répartition d'un diacide	70
2.2	To be or not to be!	70
2.3	Courbes de répartition d'un polyacide	70
2.4	MLL's Challenge	71
3	Solutions...	72
3.1	Courbes de répartition d'un diacide	72
3.2	To be or not to be!	72
3.3	Courbes de répartition d'un polyacide	73
3.4	MLL's Challenge	75
6	De GGAAATT... à Met-Asp-Gly...	76
1	Lecture - écriture d'un fichier texte	76
1.1	Lecture d'un fichier texte	76
1.2	Écriture d'un fichier texte	78
2	Code de contrôle d'un texte	78
3	Du génome de l'ADN à la séquence de l'ARN	79
4	A la recherche du codon « start »	79
4.1	Recherche « naïve » d'une chaîne de caractères	79
4.2	Recherche du codon start	79
5	Structure des protéines	80
5.1	Recherche du premier enchainement	80
5.2	Et ainsi de suite...	80
5.3	Structure des protéines	80
6	Pour le fun... à vous de jouer les MERRIFIELD!	81
6.1	Classe Robot	81
6.2	Instanciation et appel de fonctions	82
7	Images, traitement d'images	83
1	Quelques aspects « théoriques »...	83
1.1	Un tableau (ou une matrice) comme liste de listes	83
1.2	Image bitmap	84
1.3	Traitement d'image	86
1.4	Niveau de gris	86
2	A vous de jouer...	87
2.1	Création d'images	87
2.2	Coucou, Lenna	89
2.3	S'il reste du temps... stéganographie : une image dans une image	90
8	A la recherche du zéro	92
1	Méthodes	92
1.1	Méthode de dichotomie	92
1.2	Méthode de NEWTON	93
2	Trois challenges de chimistes...	93
2.1	Taux de conversion en fonction de la température	93
2.2	A la recherche de l'hétéroazéotrope	94
2.3	Courbe de titrage	95
3	Quelques pistes	96
3.1	Synthèse de l'ammoniac	96
3.2	A la recherche de l'hétéroazéotrope	96
3.3	Courbe de titrage	96
4	Solutions	96
4.1	Synthèse de l'ammoniac	96

4.2	A la recherche de l'hétéroazéotrope	97
4.3	Courbe de titrage	98
9	Intégration numérique	100
1	Méthode des rectangles et des trapèzes	100
2	Spectre RMN ou de RMN :) et sa courbe d'intégration...	100
3	Cadeau de Noël	101
10	Il est né le divin Euler	102
1	Résolution numérique d'une équation différentielle, ou d'un système!	102
1.1	Méthode d'Euler explicite	102
1.2	Quelques remèdes...	104
2	Exemples d'application	105
2.1	L'oscillateur harmonique	105
2.2	Et en chimie...	106

Activité 1

On commence en douceur ?

Vous trouverez, si nécessaire, en annexe 5.1 de ce document, une présentation de l'environnement de développement que j'utilise.

1 Quelques rappels « théoriques »...

1.1 Types de données

Le langage de programmation python utilise plusieurs *types de données*. Les plus courants sont les suivants :

Types de données	Nom en python
Nombre entier	<code>int</code>
Nombre « à virgule »	<code>float</code>
Variable booléenne : vrai ou faux (True/False)	<code>bool</code>
Liste (ordonnée)	<code>list</code>
Chaîne de caractères (du texte...)	<code>str</code>

Ponctuellement, on pourra en rencontrer d'autres... comme par exemple le type tuple correspondant à un couple de valeurs (de n'importe quel type) séparées par une virgule.

Lors de la conception d'un programme, il est important de savoir le type de données de chacune des variables utilisées. Par exemple, si on tente de faire des opérations mathématiques sur une variable de type chaîne de caractères, on obtient une erreur (même si le texte est constitué de chiffres). Pour savoir le type de données d'une variable, on peut utiliser la fonction ***type()***.

1.2 Un premier mot sur les variables

En python, comme dans tout autre langage, l'affectation d'une valeur (issue d'une initialisation, d'un calcul, du résultat de l'appel d'une fonction...) se fait dans une variable.

Un nom de variable, valide en python, est constitué de lettres majuscules ou minuscules (non accentuées si possible!), de chiffres et de tirets bas « `_` ». Elle ne doit pas commencer par un chiffre et ne doit pas contenir d'espace ni de caractères comme `+`, `-`, `*`...

Le langage python est sensible à la casse : une différence est faite entre majuscules et minuscules.

Chaque fois qu'une affectation est exécutée :

- une variable est créée dans la « liste » des variables (si elle ne l'était pas déjà) ;

- cette variable « pointe » vers une zone mémoire dans laquelle est stockée le contenu de la variable : entier, flottant, chaîne de caractères, liste...



Aucun nom de variable ne peut apparaître à droite du signe = si elle n'a pas été affectée au préalable !

a

Stockage des variables

a.1 Codage des entiers

L'information élémentaire est stockée sous forme de bit qui peut valoir soit **0** soit **1**. Ces bits sont regroupés par octet : un octet = 8 bits.

Ainsi, par exemple, le nombre $43 = 32 + 8 + 2 + 1 = 2^5 + 2^3 + 2^1 + 2^0$ peut être codé, en base 2 par 101011 et être stocké dans un octet : 00101011. Si cette représentation est fort pratique au niveau machine, pour communiquer la représentation en hexadécimal est plus facile : et le nombre s'écrira finalement : 2B en hexadécimal.

Voici, pour la petite histoire, un petit bug à 370 millions d'euros !

Dans le système de pilotage de la fusée Ariane 4, une variable était allouée à l'accélération horizontale. Sa valeur décimale étant d'environ 64 (dans leur système d'unités), on décida de la stocker dans un registre mémoire de 8 bits. Lors du passage à Ariane 5, « on » ne prit pas garde de réviser cette partie du système qui fonctionnait à la perfection. Mais... Ariane 5 était bien plus puissante et l'accélération pouvait atteindre une valeur de 300...

La plupart des systèmes traitent maintenant les entiers sur 32 bits ou 64 bits. En python, la taille n'est limitée que par la mémoire puisque les entiers sont stockés sous forme continue d'octets.

a.2 Codage des nombres à virgule flottante

Ouh, là, là... ça devient trop compliqué pour nous.

Par exemple que pour coder un nombre flottant en simple précision sur 32 bits on écrit ce nombre sous la forme $(-1)^S \times M \times 2^{E-127}$

- S est le bit de signe ;
- E est l'exposant codé sur 8 bits
- M est la mantisse : c'est un nombre réel compris entre 1 et 2 : $1 \leq M < 2$ dont on code les chiffres après la virgule sur 23 bits (comme $2^{23} = 8388608$ on a environ 7 chiffres significatifs)

On se fiche un peu de cette salade interne mais, il faudra parfois être vigilant. Certains nombres réels, même très simple (comme par exemple 0,1) n'est pas stocké sous forme exacte. Et donc, si on teste l'égalité entre les nombre 0.3 et $3*0.1$; le résultat sera faux. Ainsi, si le test de l'égalité entre deux nombres réels est critique au niveau du programme, il sera préférable de vérifier si la valeur absolue de la différence entre les nombres n'est pas plus petite qu'une valeur fixée arbitrairement (10^{-16} conviendrait dans le cas de $3*0.1 - 0.3$).

b

Codage d'un caractère

On ne citera que le code ASCII qui, depuis les années 1960, permet de coder les caractères sur un octet. Ce système ne permet pas, en particulier, la gestion des accents !

En python, on accède au code ASCII d'un caractère grâce à l'instruction **ord(caractere)** et, inversement, au caractère à l'aide de **chr(code)**. Par exemple `chr(65)` retourne la lettre "A" et `ord("A")` retourne l'entier 65. Le code ASCII de la lettre "a" vaut 97.

Ces instructions, qui vous seront rappelées si besoin, sont fort utiles pour tout ce qui concerne le codage de message (code de CÉSAR, VIGÉNÈRE ...).

1.3 Entrées/sorties élémentaires

a Demander une information à l'utilisateur

C'est l'instruction `reponse = input("texte affiché à l'écran")` qui permet de stocker dans la variable `reponse` la réponse de l'utilisateur.



`reponse` est une variable de type chaîne de caractères, même si on demande d'introduire un nombre ! Si l'on souhaite obtenir un nombre entier ou réel, une conversion explicite est nécessaire : elle se fera à l'aide, respectivement, des instructions `entier = int(reponse)` ou `reel = float(reponse)`.



Si vous utilisez Spyder, la saisie se fait dans la console ! Si vous écrivez `ch = input("machin")`, `machin` apparaît dans la console, il faut cliquer à droite de cette ligne et valider.

b Imprimer un résultat

C'est, bien sûr l'instruction `print(chaine)` qui permet d'imprimer la chaîne de caractère `chaine`. Pour imprimer un nombre (entier ou réel), on utilisera l'instruction `chaine = str(nombre)` si la conversion implicite n'est pas possible.

Une instruction comme `print("R = ", R, " J.K-1.mol-1")` est valable (pour peu que l'on ait écrit la variable `R = 8.314` au préalable).

1.4 Avec des si...

a Objectif

Il s'agit de permettre l'exécution d'une série d'instruction uniquement si une condition est vérifiée.

```
INITIALISATION de paramètres
si condition(s) sur ces paramètres alors
| ... suite d'instruction à effectuer
fin
... instructions effectuées systématiquement
```

On peut, parfois souhaiter exécuter certaines instructions si une condition est vérifiée, et d'autres instructions dans le cas contraire :

```
INITIALISATION de paramètres
si condition(s) sur ces paramètres alors
| ... suite d'instruction à effectuer si la condition est vérifiée
fin
sinon
| ... suite d'instruction à effectuer si la condition n'est pas vérifiée
fin
... instructions effectuées systématiquement
```

Les conditions peuvent être multiples... il faudra prendre soin de s'assurer s'il s'agit d'un *et* ou d'un *ou* entre les différents tests mis en jeu.

b Implémentation

En python, `si` se traduit par `if` et `sinon` se traduit par `else` .
La structure générale est la suivante :

```
1 initialisations
2 if condition à vérifier :
3     ... suite d'instruction à effectuer
4 suite du programme
```

ou

```
1 initialisations
2 if condition à vérifier :
3     ... suite d'instruction à effectuer si la condition est
      vérifiée\\
4 else :
5     ... suite d'instruction à effectuer si la condition n'est
      vérifiée\\
6 suite du programme
```

ou encore

```
1 initialisations
2 if condition1 :
3     ... suite d'instruction à effectuer si la condition1 est
      vérifiée\\
4 elif condition2 :
5     ... suite d'instruction à effectuer si la condition2 n'est
      vérifiée\\
6 else :
7     ... à effectuer si ni la condition1 ni la condition2 n'est
      vérifiée
8
9 suite du programme
```

La condition mise en jeu dans l'instruction `if` ou `elif` résulte la plupart du temps du résultat :

- du test d'égalité entre deux nombres : on utilise alors l'opérateur `==`
- du test d'une différence entre deux nombres à l'aide de l'opérateur `!=`
- d'une comparaison de deux nombres à l'aide des opérateurs : `<`, `<=`, `>`, `>=`

c Pièges et difficultés



L'instruction pour tester une égalité n'est pas « `if a = b :` » mais « `if a == b :` » . Le symbole « `=` » est réservé aux affectations de valeurs ; l'égalité se teste à l'aide de l'instruction `==`

Si deux conditions sont imbriquées, on n'écrit pas « `else if :` » mais `elif :` .



Dès que les tests à effectuer se compliquent un peu, il faudra veiller à bien gérer les indentations ! Par exemple, on passe par ici, par là, ou là ?

```
1 for a in [True, False]:
2     for b in [True, False]:
```

```

3         if a :
4             print("a : ", a, " b : ", b, " ici")
5         elif b :
6             print("a : ", a, " b : ", b, " là")
7         else :
8             print("a : ", a, " b : ", b, " ou là")
9
10    for a in [True, False]:
11        for b in [True, False]:
12            if a :
13                print("a : ", a, " b : ", b, " ici")
14                if b :
15                    print("a : ", a, " b : ", b, " là")
16            else :
17                print("a : ", a, " b : ", b, " ou là")
18
19    for a in [True, False]:
20        for b in [True, False]:
21            if a :
22                print("a : ", a, " b : ", b, " ici")
23                if b :
24                    print("a : ", a, " b : ", b, " là")
25            else :
26                print("a : ", a, " b : ", b, " ou là")

```

d Opérations booléennes

d.1 Opérations booléennes de base Supposons que a et b soient deux variables booléennes. On peut envisager, quelques opérations booléennes de base.

Et : `and` La proposition a et b ne peut être vraie que si les deux propositions sont vraies en même temps.
La structure de test en python s'écrira : `if a and b :` .

Ou : `or` La proposition a ou b est vraie dès que l'une des deux propositions est vraie.
La structure de test en python s'écrira : `if a or b :` .

Ou exclusif : `^` Le ou exclusif entre a et b peut se traduire par soit a , soit b . La proposition est vraie si une seule des deux conditions soit a , soit b est vraie.
La structure de test en python s'écrira : `if a ^ b :` .

Négation : `not` Pour prendre la négation d'une proposition, on utilise, en python, l'instruction `not` .

d.2 Combinaison d'opérateurs de base Toute opération logique entre ces opérateurs est alors possible... même si le résultat n'est pas toujours simple à prévoir !

```

1    for a in [True, False]:
2        for b in [True, False]:

```

```
3 |         print(a, " ", b, " ", (a ^ (not b)) and (not (a or
      |         b)))
```

1.5 Retour sur... les boucles

L'utilisation des instructions séquentielles est fondamentale en informatique !

a Tant que...

a.1 Objectif

Répéter certaines instructions tant qu'une condition n'est pas réalisée.

```

valeur_test ← valeur numérique
compteur ← valeur numérique
tant que compteur ≤ valeur_test faire
|   ... suite d'instruction à effectuer
|   modification de la valeur de compteur
fin
```

Plus généralement, la syntaxe devra obligatoirement obéir à la structure suivante :

```

INITIALISATION de paramètres
tant que condition(s) sur ces paramètres faire
|   ... suite d'instruction à effectuer
|   MODIFICATION des paramètres
fin
```

Les conditions peuvent être multiples... il faudra prendre soin de s'assurer s'il s'agit d'un *et* ou d'un *ou* entre les différents tests mis en jeu.

a.2 Implémentation En python, `tant que` se traduit par `while` (en minuscule).

La structure générale de la boucle *while* est la suivante :

```

1 | initialisations
2 | while condition à vérifier :
3 |     ...
4 |     suite d'instruction à effectuer
5 |     modification de la condition
6 | suite du programme
```

Les premiers exemples d'utilisation de la boucle *while* peuvent se mettre sous la forme suivante :

Utilisation d'un compteur de boucle

```
i = debut
while i <= fin :
    ...
    ... instructions à exécuter
    ...
    i = i + pas
... suite du programme
```

Utilisation d'un test

```
flag = True
while flag :
    ...
    ... instructions à exécuter
    ...
    if (condition)
        flag = False
... suite du programme
```

Une utilisation courante de la boucle *while* consiste à remplir un tableau de valeur (afin de tracer une courbe par exemple). Dans l'exemple suivant, on remplit la liste *T* avec toutes les valeurs réelles comprises entre *debut* et *fin* espacées de *increment*.

```
1 debut = 0
2 fin = 1
3 increment = 0.1
4 compteur = debut
5 T = []
6 while compteur <= fin :
7     T.append(compteur)
8     compteur = compteur + increment
9 print (T)
```

b

Pièges et difficultés



Le plus grand piège est de ne pas modifier les paramètres dans la boucle *while*. ... on obtient alors une boucle infinie (ou jamais parcourue) puisque la condition est toujours (ou jamais) vérifiée. Si vous utilisez Spyder, il y a un carré rouge dans la barre d'entête de la console IPython pour mettre fin à l'infini !

Exemple : remplir le tableau *T* avec des nombres réels compris entre 0 et 1 inclus espacés de 0,01.

```
1 T=[]
2 x=0
3 while x <= 1 :
4     T.append(x)
5     x = x+0.01
6 print("T : ", T)
```

Bien sûr, si on initialise mal les paramètres ou si on code mal la condition d'arrêt, le programme ne fera pas ce que l'on souhaite.

c

Pour...

c.1 Objectif La principale utilité de ce type de boucle est de répéter certaines instructions un certain nombre de fois.

Une autre application intéressante en python est de pouvoir parcourir une liste sans avoir à gérer les indices.

c.2 Implémentation

En réalité, l'implémentation d'origine de la boucle Pour en python, est justement le parcours

```
pour i variant de début à fin par pas de pas faire  
| ... suite d'instruction à effectuer  
| ... on peut accéder à différents éléments par l'intermédiaire de la variable i  
fin
```

```
TableauT ← valeurs  
pour x dans T faire  
| ... suite d'instruction à effectuer  
| ... on accède directement à la variable x (on ne dispose pas de son indice dans le tableau)  
fin
```

d'éléments d'une collection.
L'instruction :

```
for x in T :
```

permet de stocker successivement dans la variable *x* tous les éléments initialement contenus dans le tableau *T*.
Par exemple le script suivant permet d'imprimer dans la console les différents jours de la semaine.

```
1 semaine = ["dimanche", "lundi", "mardi", "mercredi", "jeudi",  
             "vendredi", "samedi"]  
2 for jour in semaine :  
3     print(jour)
```

Le problème, ici, est que l'on ne peut pas accéder à l'indice du jour dans le tableau semaine.

L'instruction **in range(*debut*, *fin*, *pas*)** permet de générer une liste de valeurs entières : [*debut*, *debut* + *pas*, *debut* + 2*pas*, ... dernière valeur strictement inférieure à *fin*].
En combinant les deux instructions on va pouvoir récupérer soit une liste d'indices, soit une liste de valeurs.
La boucle **Pour** se construit alors à partir de l'instruction :

```
for i in range(début, fin (valeur exclue), pas (= 1 par défaut)) :
```



ATTENTION : les variables *début*, *fin* et *pas* sont obligatoirement des ENTIERS.



ATTENTION : La valeur *fin* n'est pas atteinte!
Si l'incrément (ou le pas) vaut 1, on n'est pas obligé de le spécifier.¹
Par exemple :

Instruction	Valeurs successives de <i>i</i>
for <i>i</i> in range(0, 4)	0 ; 1 ; 2 ; 3
for <i>i</i> in range(-10, 10, 2)	-10 ; -8 ; -6 ; -4 ; -2 ; 0 ; 2 ; 4 ; 6 ; 8
for <i>i</i> in range(10, 5, -1)	10 ; 9 ; 8 ; 7 ; 6

1. On rencontre également une instruction du type for *i* in range(5). Dans ce cas, le début vaut forcément 0, on ne précise que la *fin* (non atteinte) et le pas vaut 1. Dans le cas présent, *i* vaut successivement 0, 1, 2, 3, 4.

c.3 Pièges et difficultés



Attention aux deux erreurs principales dans l'utilisation de `for i in range(...)` :

- la valeur *fin* n'est pas atteinte ;
- les valeurs sont des entiers (cette erreur est facilement détectée lors de l'exécution mais à l'écrit, c'est une autre histoire !)

Comme pour toutes les structures de contrôle, il ne faut pas oublier les « : » à la fin de la ligne ; et bien sûr, comme ailleurs en python, seules les instructions indentées convenablement seront exécutées !

Signalons enfin qu'il ne faut pas modifier la valeur du « compteur de boucle » à l'intérieur de la boucle.

1.6 While ou for ?

A priori, la réponse est assez simple :

- on utilise la boucle **for** :
 - pour le parcours des éléments d'une liste : `for i in range(len(liste)) ;`
 - si on connaît facilement le nombre d'itérations
- on utilise la boucle **while** : dans tous les autres cas !

Parfois, les deux solutions sont possibles sans pouvoir privilégier l'une ou l'autre. A titre purement personnel, je privilégie la boucle **while** !

Si vous avez un peu de mal avec l'écriture des boucles while :

- commencez par écrire l'initialisation de la boucle ;
- passez à la ligne *while* en codant la condition
- laissez un peu de place et codez la modification de la condition
- terminez par le corps de la boucle

2 A vous de jouer...

Comme vous le dites à vos élèves... « jouez le jeu ! » ; je vous donne quelques pistes de résolution et la solution par la suite... essayez de résoudre sans aide !

N'hésitez pas à ajouter des commentaires dans votre code. Pour mettre tous les exercices sur le même fichier python et ne pas être embêté (pour être poli) avec tous les `input` vous pouvez mettre tout un bloc de code en commentaire :

- soit en le sélectionnant et en utilisant (sous spyder) le menu Edit puis Comment/UnComment ;
- soit en mettant 3 guillemets avant et 3 guillemets après le bloc de code correspondant

```

1  a = 1 #et le commentaire pour une seule ligne
2
3  #a = 2
4  #b = 3
5  #print(a+b)
6
7  """
8  a = 3
9  b = 4
10 print(a+b)
11 """
```

```

12 |
13 | b = 2
14 | print(a+b)

```

L'importation de fonctions de la bibliothèque mathématique (cf Annexe 5.2) sera nécessaire pour certains exercices.

2.1 Blondes

.1 Écrire à l'aide d'une boucle **for** un programme qui calcule la somme de nombres pairs entre 1 et 100 (inclus) et affiche le résultat.

.2 Écrire un programme qui demande un nombre x , calcule $\frac{x^2}{2}$ si $x < -2$ ou $x > 4$; $4 - \frac{x^2}{2}$ sinon, puis affiche le résultat.

.3 Écrire un programme qui affiche les nombres réels compris entre 0 et 1 (inclus) espacés de 0,1 :

- a) à l'aide d'une boucle **while**;
- b) à l'aide d'une boucle **for**

2.2 Châtain clair...

.1 Écrire à l'aide de deux boucles **for** imbriquées un programme qui calcule la somme : $S = \sum_{i=1}^n \sum_{j=1}^n ij$. Que vaut cette somme pour $n = 10$?

.2 Écrire un programme qui calcule la somme des entiers k positifs tels que $k + k^2 + k^3 \leq n$, n étant un nombre introduit dans le programme. Que vaut cette somme pour $n = 1000$?

.3 Écrire un programme qui calcule la limite de la suite $S_n = \sum_{i=1}^n \frac{1}{i^2}$ sachant que l'on arrête l'exécution lorsque $|S_{n+1} - S_n| < \epsilon$ avec ϵ proche de zéro.

Vous vérifierez que la limite de la suite est une bonne approximation de $\frac{\pi^2}{6}$ et en déduirez une valeur approchée de π .

.4 On introduit un entier positif n . Écrire un programme qui calcule la somme S_n des termes suivants :

$$\begin{array}{ccccccc}
 1 \times 1 & + & 1 \times 2 & + & \cdots & + & 1 \times n \\
 & & + & 2 \times 2 & + & \cdots & + & 2 \times n \\
 & & & & & \ddots & & \vdots \\
 & & & & & & \ddots & \vdots \\
 & & & & & & & + & n \times n
 \end{array}$$

Que vaut S_8 ?

.5 Écrire un programme qui demande les nombres a , b et c dans l'équation $ax^2 + bx + c = 0$ et qui donne en retour la(les) solution(s) de cette équation (ou affiche un message si elles n'existent pas).

2.3 Brunes ou chauves !

L'objectif est d'écrire un programme de **chifoumi** (pierre - feuille - ciseaux). Le cahier des charges est le suivant :

- une partie se joue en 3 manches gagnantes ;
- pour chaque manche, pour éviter de tricher, on fait une proposition ; l'ordinateur (qui ne triche pas non plus) tire au hasard sa proposition, affiche la comparaison des deux propositions ainsi que le résultat ;
- à la fin d'une partie on affiche "Tapez q ou Q pour quitter". On quitte le jeu si l'utilisateur tape *q* ou *Q* et on rejoue sinon.

Pour choisir aléatoirement ce que joue l'ordinateur, on peut utiliser la bibliothèque **random** :

```
1 import random as rd
2 n = rd.randint(0,100) # génère un entier aléatoire entre 0 et
    100 (inclus)
```


3 Quelques pistes...

3.1 Blondes

.1

- une boucle `for i in range` fera l'affaire (attention à la borne supérieure!);
- pour sommer, on initialise une variable que l'on incrémente!

.2

- la gestion du `if/else` ne devrait pas poser de problème;
- pensez à convertir la chaîne de caractères (obtenue grâce à `input`) en entier!

.3

- de classiques parcours de boucles...;
- que l'on peut rater en oubliant l'incrémentation (boucle `while`) ou en se trompant dans les bornes (boucle `for`).

3.2 Châtain clair...

.1

- si l'on sait faire une boucle pour faire une somme... ce n'est pas plus difficile d'en imbriquer deux!
- vérifiez que pour $n = 10$ la somme vaut 3025.

.2

- on ne connaît pas, *a priori*, la plus grande valeur de k telle que $k + k^2 + k^3 \leq n$; il faut donc utiliser une boucle `while`!
- il faudra alors gérer la variable k d'une part, la variable de test $k + k^2 + k^3$ à la fois dans l'initialisation et dans la boucle `while`.
- vérifiez que pour $n = 1000$ la somme vaut 45.

.3

- cette fois encore, on ne sait pas immédiatement quand on va devoir s'arrêter donc... on fait une boucle tant que l'incrément est plus grand qu'une limite fixée à l'avance.
- la fonction racine carrée est codée par `sqrt`; il faut l'importer ou importer la bibliothèque `math`:

.4

- la principale difficulté, ici, est la gestion des indices : on a un indice l pour la ligne qui doit aller de 1 à n inclus; et un indice c pour la colonne qui doit aller de l (inclus) à n inclus².
- pour $n = 8$, je trouve 750.

2. J'ai toujours eu du mal avec les indices i et j pour savoir qui était la ligne et qui était la colonne; un jour, j'ai eu l'idée d'appeler l l'indice pour la ligne et c l'indice pour la colonne... c'est con mais c'est efficace!

.5 Equation du second degré

- cette fois encore il faut convertir les chaînes de caractère en réel ;
- la fonction racine carrée est codée par `sqrt` ; il faut l'importer ou importer la bibliothèque `math` (c'est la dernière fois que je le rappelle!).

3.3

Brunes ou chauves !

Bon, cette fois, ça se complique un peu. Comme on ne dispose pas encore des listes et des fonctions, il va falloir se débrouiller avec ce que l'on vient de réviser !

A priori, 4 problèmes se posent :

- problème de structure de données : va-t-on traiter des chaînes de caractères (ou des caractères) : "P", "F", "C" ou des entiers 0, 1, 2 ? C'est LE problème auquel vous serez très souvent confronté, quelle structure de données allez vous utiliser pour modéliser votre système ? Le choix n'est pas forcément évident !
- sinon il va falloir gérer une manche : comparer les propositions du joueur et de l'ordinateur pour savoir qui va marquer le point (s'il n'y a pas égalité) ;
- puis aussi la partie, il faut 2 points d'avance pour gagner !
- et enfin, savoir si on doit rejouer ou pas.

Pour la gestion de la structure de données, le plus simple est peut-être de demander une réponse du joueur sous forme de nombre entier 0, 1 ou 2 plutôt que sous forme de caractère "P", "F", "C".

Pour le programme proprement dit, il est illusoire, j'espère, d'envisager l'écriture linéaire ! Plusieurs solutions s'offrent à vous, sans qu'il y en ait une de meilleure... pour les débutants, le plus sage serait peut-être :

- d'écrire d'un côté le cœur du programme :
 - on commence par écrire la gestion d'une manche et l'affichage du résultat de la comparaison du choix (si on ne veut toujours avoir à répondre aux input, on peut très bien coder le choix du joueur et tester les différents cas de figure) ;
 - on modifie ce programme pour tenir compte maintenant de la gestion d'une partie.
- d'écrire à part la gestion nouvelle partie
- de rabouter tout ça !

Bon, ce qui est pénible, c'est que l'on va passer son temps à changer les indentations et qu'on risque d'en oublier une. Avec un peu plus d'expérience, le plus simple est peut-être la stratégie « qu'y a-t-il à l'intérieur d'une noix ? »

- on commence par programmer la boucle : tant que l'on ne quitte pas on joue !
- quand on joue, c'est pour commencer une nouvelle partie ; il faut donc initialiser les points
- et tant que la différence entre ces points est plus petite que 2
- il faut faire une manche... et on termine par la programmation d'une manche !

4 Solutions !

En tête du fichier j'introduis les bibliothèques nécessaires. Je ne supprime pas la ligne avec utf-8, elle permet d'avoir (avec spyder) des caractères accentués.

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Mon Aug 28 21:28:24 2017
4
5  @author: Alain
6  """
7
8  from math import sqrt
9  import random as rd

```

4.1 Blondes

```

.1
1  S = 0
2  for i in range(0,102,2):
3      S = S + i
4  print("Somme des nombres pairs de 1 à 100 : ", S)

```

Bien sûr, si la valeur 100 doit être incluse, il faut aller jusqu'à 102.

```

.2
1  ch = input("Entrez un nombre !")
2  x = float(ch)
3  if x < -2 or x > 4 :
4      y = x*x/2
5  else :
6      y = 4 - x*x/2
7  print("résultat = ", y)

```

On pourrait aussi écrire `x = float(input("Entrez un nombre !"))`.

```

.3
1  print("Avec boucle while")
2  x = 0
3  while x <= 1 :
4      print(x)
5      x = x + 0.1
6
7  print("Avec boucle for")
8  for i in range(0,11):
9      print(i/10)

```

Comme les matheux semblent préférer la boucle `for` à la boucle `while`, vous verrez bon nombre de vos élèves gérer des listes de réels de la sorte !

4.2 Châtain clair...

```
.1
1 n = 10
2 S = 0
3 for i in range(1,n+1):
4     for j in range(1, n+1):
5         S = S + i*j
6 print(S)
```



il faut bien écrire `in range(1,n+1)` pour atteindre la valeur `n` ; bon OK je ne vous le redis plus !

```
.2
1 n = 1000
2 S = 0
3 k = 1
4 test = 3
5 while test < n :
6     S = S + k
7     k = k + 1
8     test = k + k**2 + k**3
9 print (S)
```

C'est l'écriture typique d'une boucle `while` sur un test logique, il faut prendre garde à bien initialiser le test avant d'entrer dans la boucle... et de le modifier ensuite.

```
.3
1 epsilon = 1e-12
2 S = 0
3 i = 1
4 increment = 1
5 while increment > epsilon :
6     S = S + increment
7     i = i + 1
8     increment = 1/(i*i)
9 print("Limite = ", S)
10 print("pi = ", sqrt(6*S))
```

La division est évaluée avant la multiplication : si on écrit `1/i*i` il comprend $(1/i) * i$. Par contre la puissance est évaluée avant toute autre opération (sauf le moins unitaire) donc si on écrit `1/i**2` ; c'est bon. Bon, si on écrit `1/i^2`, autant dire que l'on a une erreur de syntaxe !

```
.4
1 n = 8
2 S = 0
3 for l in range (1,n+1):
4     for c in range(1, n+1):
5         S = S + l*c
6 print ("Somme = ", S)
```

Il me semble avoir dis que je ne vous disais plus de faire attention aux indices...

```
.5 Equation du second degré
1 cha = input("a = ?")
```

```

2 chb = input("b = ?")
3 chc = input("c = ?")
4 a = float(cha)
5 b = float(chb)
6 c = float(chc)
7 if a == 0 :
8     print("Solution unique : ", -c/b)
9 else :
10     delta = b*b - 4*a*c
11     if delta > 0 :
12         print("Deux racines : ", (-b + sqrt(delta))/(2*a), "
              et ", (-b - sqrt(delta))/(2*a))
13     elif delta == 0 :
14         print("Solution unique : ", - b/(2*a))
15     else :
16         print("Aucune racine")

```

4.3 Chifoumi

Voici les différentes étapes de ma progression :

- on joue ou on joue plus ?

```

1 while onjoue :
2
3     ch = input("Tapez q ou Q pour quitter, sinon on rejoue !")
4     if ch == "q" or ch == "Q" :
5         onjoue = False
6 print ("Merci")

```

- si on joue, il faut initialiser le nombre de points du joueur et de l'ordinateur et faire une boucle tant que la différence est plus petite que 2.

```

1 while onjoue :
2     pointJoueur = 0
3     pointOrdi = 0
4     while abs(pointJoueur - pointOrdi) < 2 :
5
6         ch = input("Tapez q ou Q pour quitter, sinon on rejoue !")
7         if ch == "q" or ch == "Q" :
8             onjoue = False
9 print ("Merci")

```

- tant qu'on y est dans la gestion des points, autant dire qui a gagné!

```

1 while onjoue :
2     pointJoueur = 0
3     pointOrdi = 0
4     while abs(pointJoueur - pointOrdi) < 2 :
5
6
7     if pointJoueur > pointOrdi :
8         print("Bravo, vous avez gagné ", pointJoueur, " à ",
              pointOrdi)

```

```

9     else:
10         print("Pas de chance pour vous, j'ai gagné",
                pointOrdi, " à ", pointJoueur)
11     ch = input("Tapez q ou Q pour quitter, sinon on rejoue !")
12     if ch == "q" or ch == "Q" :
13         onjoue = False
14     print ("Merci")

```

- dans une manche, maintenant il faut demander au joueur de faire un choix (pour me simplifier la tâche j'ai demandé d'écrire un nombre). L'ordinateur tire un nombre au hasard entre 0 et 2.

```

1  while onjoue :
2      pointJoueur = 0
3      pointOrdi = 0
4      while abs(pointJoueur - pointOrdi) < 2 :
5          choixJoueur = int(input("Quel est votre choix : 0 pour
                                   pierre, 1 pour feuille, 2 pour ciseaux ?"))
6          choixOrdi = rd.randint(0,2)
7
8      if pointJoueur > pointOrdi :
9          print("Bravo, vous avez gagné ", pointJoueur, " à ",
                pointOrdi)
10     else:
11         print("Pas de chance pour vous, j'ai gagné",
                pointOrdi, " à ", pointJoueur)
12     ch = input("Tapez q ou Q pour quitter, sinon on rejoue !")
13     if ch == "q" or ch == "Q" :
14         onjoue = False
15     print ("Merci")

```

- il nous reste alors à gérer la comparaison entre les deux choix. Petite ruse de sioux, pour limiter tout un tas de else j'ai fixé un choix arbitraire au début. Au lieu d'un elif contenant tous les cas de figure, on aurait pu les traiter un par un!

```

1  while onjoue :
2      pointJoueur = 0
3      pointOrdi = 0
4      while abs(pointJoueur - pointOrdi) < 2 :
5          choixJoueur = int(input("Quel est votre choix : 0 pour
                                   pierre, 1 pour feuille, 2 pour ciseaux ?"))
6          choixOrdi = rd.randint(0,2)
7          resultat = - 1 # reste à détecter l'égalité ou le
                          joueur vainqueur
8          if choixJoueur == choixOrdi :
9              resultat = 0
10         elif (choixJoueur == 0 and choixOrdi == 2) or
                (choixJoueur == 1 and choixOrdi == 0) or
                (choixJoueur == 2 and choixOrdi == 1) :
11             resultat = 1
12         if resultat == 0 :
13             print("J'ai joué ", choixOrdi, " - égalité -
                    Joueur : ", pointJoueur, " ordinateur : ",
                    pointOrdi)

```

```
14         elif resultat == 1 :
15             pointJoueur = pointJoueur + 1
16             print("J'ai joué ", choixOrdi, " - vous avez gagné
              - Joueur : ", pointJoueur, " ordinateur : ",
              pointOrdi)
17         else :
18             pointOrdi = pointOrdi + 1
19             print("J'ai joué ", choixOrdi, " - j'ai gagné -
              Joueur : ", pointJoueur, " ordinateur : ",
              pointOrdi)
20     if pointJoueur > pointOrdi :
21         print("Bravo, vous avez gagné ", pointJoueur, " à ",
              pointOrdi)
22     else:
23         print("Pas de chance pour vous, j'ai gagné",
              pointOrdi, " à ", pointJoueur)
24     ch = input("Tapez q ou Q pour quitter, sinon on rejoue !")
25     if ch == "q" or ch == "Q" :
26         onjoue = False
27     print ("Merci")
```

Et en principe, ça marche !

5

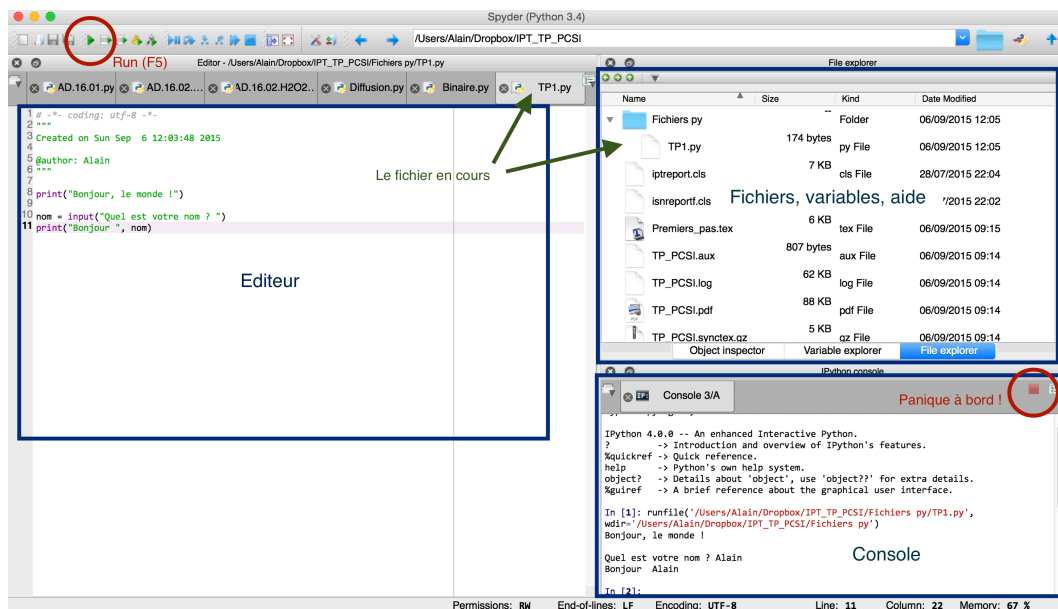
Annexes

5.1 Environnement de développement Spyder

Le langage python est un langage simple, puissant et gratuit. Il en existe de nombreuses distributions, plus ou moins conviviales et plus ou moins « volumineuses ». Moi, j'aime bien l'environnement *Spyder*. Vous pouvez en trouver une version pour tout type d'ordinateur à l'adresse : <http://http://continuum.io/downloads> (il suffit de taper Anaconda python sous Google). Vous choisissez votre système d'exploitation et, surtout, vous choisissez une version 3.x

Pour celles et ceux qui ont des machines peu performantes, vous pouvez utiliser un environnement plus « léger » comme la distribution pyzo (www.pyzo.org/downloads.html) ou Edupython (<http://edupython.tuxfamily.org/>).

Si vous avez téléchargé Spyder, vous vous retrouvez avec l'écran suivant ; les autres distributions se présentent de façon similaire.



Que doit-on repérer en priorité ?

- Une zone **Editeur** qui sert... à éditer le programme dont vous souhaitez l'exécution ; ce sera votre terrain de jeu !
- Une zone **Console** dans laquelle seront affichés les résultats de vos calculs, les graphes, très souvent... les messages d'erreur ! C'est également là qu'il faudra introduire éventuellement certaines données demandées par le programme. C'est donc une zone d'interface entre le programme et l'utilisateur³. On peut, si on veut se servir de la console comme d'une calculatrice ou effectuer quelques tests simples de programmation.
- Le bouton **Run** (équivalent clavier F5) vous permet de lancer l'exécution du programme (parfois, des problèmes de connexion à la console interviennent et un message d'erreur du type « Open an IPython console » apparaît, il faut alors ouvrir une telle console à partir du menu Console).
- Parfois votre programme va se bloquer (suite à une erreur de programmation !); on peut arrêter son exécution à l'aide du bouton « carré rouge » dans la console.

3. Parfois, aucune console ne se trouve associée au programme que l'on souhaite exécuter. Le logiciel va vous demander d'ouvrir une console ; un menu *Console* est prévu à cet effet, choisir une « IPython Console »

- Une fenêtre en haut à droite avec différents onglets qui permet d'accéder, entre autre : à l'arborescence des fichiers sur votre disque dur, aux valeurs des différentes variables que vous allez utiliser, à l'aide en ligne sur certaines fonctions. . .

5.2 Importation de bibliothèques

a Opérations arithmétiques de base

Un certain nombre d'opération de base sont disponible. Elles sont représentées, sans surprise, par les symboles : $+$, $-$, $*$, $/$. On peut également utiliser des parenthèses.

Ajoutons ****** pour une puissance ($a**b$ revient à a^b) ou encore **%** pour un modulo, **abs** pour la valeur absolue.

Attention aux ordres de priorité des opérateurs. Exécuter le programme suivant et, s'il ne fait pas ce que vous souhaitez, ajoutez des parenthèses !

```
1 a = 3
2 b = 2
3 print(a + b/a - b)
4 print(a/b*b)
5 print(a/b**2)
```

Mais, la plupart des opérations usuelles (fonctions trigonométriques, π , logarithme ou exponentielle. . .) ne sont pas disponibles et il faudra quasiment systématiquement importer la bibliothèque mathématique.

b Utilisation de bibliothèques

On souhaite maintenant calculer la racine carrée ou le sinus d'un nombre. Le langage python ne dispose pas, de base, de la fonction racine carrée. Pour pouvoir en bénéficier, il faut utiliser une bibliothèque de fonctions (que l'on appelle aussi module). Dans notre cas, nous allons importer la bibliothèque **math**. Pour cela, il faut écrire, en tête du programme, une des lignes suivantes :

- « `from math import *` » ; on importe alors la totalité de la bibliothèque mathématique ;
- « `from math import sqrt, sin, pi` » ; on importe alors uniquement les deux fonctions souhaitées ;
- « `import math as m` » et l'on écrira alors⁴ `m.sqrt(3)` ou `m.sin(pi/4)`

Voici quelques fonctions de la bibliothèque math :

4. Cette méthode peut sembler un peu lourde de prime abord. Elle permet toutefois d'éviter des conflits si une même fonction est définie dans plusieurs bibliothèques. Elle permet également de disposer d'une aide contextuelle lors de la frappe ; il suffit de taper `m.` et ensuite un menu déroulant apparaît avec toutes les fonctions disponibles

<code>sqrt()</code>	Racine carrée d'un nombre
<code>pow(x ,y)</code> ou <code>x**y</code>	Calcule x à la puissance y
<code>sin()</code>	Sinus d'un angle (en radian)
<code>cos()</code>	Cosinus d'un angle (en radian)
<code>tan()</code>	Tangente d'un angle (toujours en radian)
<code>pi</code>	Une valeur approchée précise du nombre π
<code>log</code>	Logarithme népérien
<code>log10</code>	Logarithme décimal
<code>exp</code>	Exponentielle

Activité 2

Indispensables listes...

1 Quelques rappels « théoriques »...

1.1 Listes python

Sous Python, on peut définir une liste comme une collection ordonnée d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets. Les différents éléments ne sont pas forcément de même type.

Chaque élément est repéré par son indice (ou index, ou rang), mais attention, le premier élément est celui d'indice 0, le deuxième d'indice 1 etc...

ATTENTION - ATTENTION - ATTENTION - ATTENTION

L'indice commence à 0!

ATTENTION - ATTENTION - ATTENTION - ATTENTION

c'est malain.

Fonctions de base de manipulation de listes :

<code>maliste = []</code>	crée une liste vide nommée « maliste »
<code>maliste.append(9)</code>	ajoute l'élément « 9 » en queue de liste
<code>len(maliste)</code>	renvoie le nombre d'éléments (la longueur) de la liste
<code>maliste[i]</code>	renvoie l'élément de rang i contenue dans maliste

Un peu plus tard, on compliquera en introduisant les listes de listes.

1.2 Accès aux éléments d'une liste

a Accès à l'aide d'un indice

La solution la plus simple pour obtenir un indice permettant d'accéder à tous les éléments d'une liste L est :

```
for i in range(len(L)) :
```

On peut alors accéder à l'élément d'indice i via $valeur = L[i]$ ou le modifier via $L[i] = valeur$

Si on ne souhaite pas accéder à tous les éléments de la liste, on utilisera une boucle *for* « classique » :

```
for i in range(début, fin (valeur exclue), pas (= 1 par défaut)) :
```



ATTENTION : les variables *début*, *fin* et *pas* sont obligatoirement des entiers.

ATTENTION : la valeur *fin* n'est pas atteinte :

Vous verrez chez certains de vos élèves l'accès à un élément d'une liste avec des indices négatifs. Si $L = [1,2,3]$, $L[-1]$ vaut 3, $L[-2]$ vaut 2. Ce n'est pas, à mon humble avis, une démarche à recommander...

b

Accès sans indice

Le parcourt des éléments d'une liste L (sans accéder aux indices) peut se faire directement par l'instruction `for x in L`. Dans ce cas, la variable x va prendre successivement (à chaque itération) la valeur de chacun des éléments de la liste (qui ne sont pas forcément des nombres).



On accède, ainsi, directement à la variable x (on ne dispose pas de son indice dans le tableau) dont on ne peut pas changer la valeur

c

« Slicing »

Bien sûr, ce n'est pas un scoop, on accède à l'élément d'indice i du tableau T à l'aide de l'instruction $T[i]$ en faisant, attention :

- au fait que les indices commencent à ZÉRO ;
- que le dernier élément du tableau a donc comme indice $\text{len}(T) - 1$;
- qu'un indice hors limite est facilement détecté à l'exécution, mais que par écrit... c'est le correcteur qui aura la joie de repérer vos « index out of range ! ». Oh ! combien de points perdus là-dessus!!!

Parfois on ne souhaite par récupérer un élément unique de la liste mais plutôt une sous-liste de la liste d'origine.

On peut faire ça « à la main » à l'aide de notre bonne vieille instruction `for i in range(debut, fin, pas)` !

```

1  T = [1,2,3,4,5,6,7,8,9]
2  # les 4 premiers éléments du tableau
3  x = [T[i] for i in range(0,4,1)]
4  # tous les éléments à partir du 3ème
5  y = [T[i] for i in range(2,len(T),1)]
6  # les 5 derniers éléments du tableau
7  z = [T[i] for i in range(len(T)-5,len(T),1)]
8  # 3 éléments du tableau à partir du 5ème
9  t = [T[i] for i in range(4,7,1)]
10 # un élément sur 3 à partir du second
11 u = [T[i] for i in range(1,len(T),3)]

```

Bien sûr, on peut omettre le paramètre 1 comme valeur du pas et le 0 comme premier indice dans certaines expressions.

Pour faire plus savant dans les salons de thé, on peut aussi faire du « slicing » ; littéralement, du « découpage en tranche » grâce à l'instruction $T[\text{debut} : \text{fin} : \text{pas}]$. Jusque là, pourquoi pas ; mais il faut faire attention aux raccourcis :

- si le *pas* vaut 1 on peut omettre d'écrire : 1 à la fin ;
- si le début vaut 0, on peut omettre d'écrire sa valeur (mais il faut quand même garder les « : ») ;

- si la fin est le dernier élément de la liste, on peut omettre d'écrire sa valeur (mais il faut garder les « : » qui précède).

On peut ainsi proposer le même programme que précédemment mais avec « : » au lieu de *in range(...)*.

Listing 2.1 – Extraction des éléments par slicing

```

1  T = [1,2,3,4,5,6,7,8,9]
2  # les 4 premiers éléments du tableau
3  x = T[:4]
4  # tous les éléments à partir du 3ème
5  y = T[2:]
6  # les 5 derniers éléments du tableau
7  z = T[len(T)-5:]
8  # 3 éléments du tableau à partir du 5ème
9  t = T[4:7]
10 # un élément sur 3 à partir du second
11 u = T[1::3]
```

Cette technique de slicing est au programme donc vous verrez, peut-être, quelques uns de vos élèves l'utiliser. En ce qui me concerne, je ne l'utilise pas... mais c'est peut-être mon côté « anti raccourcis de matheux! ».

1.3

Création d'une liste

```

1  # création "à la main" de vecteurs
2  L = [4,3,2,1]
3  T = [8,2,5,4,1,3,7,0]
4  # ou encore pour la même liste L
5  L = []
6  for i in range(4):
7      L.append(5-i)
8  # ou encore
9  L = []
10 for i in range(4,0,-1):
11     L.append(i)
12 # ou encore, en plus compact...
13 L = [4-i for i in range(4)]
14 #on peut aussi créer une liste de 0 de la même façon
15 T = [0 for i in range(10)]
16 # et pourquoi pas un tableau de valeurs pour une fonction
17 S = [sin(i*pi/100) for i in range(100)]
```

La siouxerie proposée sur les trois derniers exemples (liste dite par compréhension) consiste à remplir une liste à partir des éléments d'une autre liste en effectuant, éventuellement, une opération.

En cachette, l'instruction *range(debut, fin(exclue), pas)* crée une liste $[debut, debut + pas, debut + 2\ pas, \dots]$. Et *for i in machin* va successivement chercher tous les éléments de la liste *machin* pour les stocker dans la variable *i*. Ainsi *i* vaut, successivement : *debut*, *debut + pas*, *debut + 2 pas*, ... et la partie gauche de l'instruction décrit, en fait, quelle fonction (de *i*) utiliser pour remplir la liste : en a en fait : $[f(i) \text{ for } i \text{ in } machin]$.



Il faut juste avoir conscience que plusieurs solutions existent... vous utilisez celle que vous voulez, mais pas un patchwork! Et, si vous avez peur de vous tromper : vous créer une liste vide et vous

faites un `append()` !

1.4 Pas d'opération arithmétiques sur les listes !

Il faut absolument se garder de faire toute opération arithmétique sur les listes !

```
1 L1 = [1,2,3] + [4,5,6]
2 L2 = 3*[1,2,3]
3 L3 = sqrt([1,2,3])
```

Les deux instructions précédentes créent deux listes :

- L1 correspond à [1,2,3,4,5,6] ;
- L2 correspond à [1,2,3,1,2,3,1,2,3]
- la troisième ligne conduit à une erreur de syntaxe

L'utilisation des deux premières expressions n'est pas interdite, elle conduit la concaténation de listes.

1.5 Quelques mots sur NumPy

NumPy est une bibliothèque fondamentale dans l'utilisation de python pour des applications scientifiques ; surtout si on y associe sa copine SciPy. Le plupart des méthodes numériques utiles au calcul scientifique sont présentes dans ces bibliothèques. On fera une séance particulière sur NumPy plus tard !

2 A vous de jouer...

2.1 Blondes

- .1 A l'aide d'une boucle, remplir une liste :
 - a) qui contient tous les nombres entiers de 0 à 10 inclus ;
 - b) qui contient tous les nombres entiers pairs de 10 à -10 inclus ;
 - c) qui contient tous les nombres compris entre 0 et 10 (inclus) espacés de 0,1.
 - d) qui contient les 26 lettres minuscules de l'alphabet.

- .2 Initialiser les 3 tableaux suivants :

```
1 Eleves = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"]
2 Trim1 = [10, 12, 9, 14, 8, 15, 18, 13, 6, 10]
3 Trim2 = [11, 12, 8, 16, 12, 16, 16, 11, 8, 9]
```

Ecrire le code d'un programme permettant de :

- a) déterminer le nombre d'élèves ayant progressé du premier au second trimestre ;
- b) calculer la moyenne du premier trimestre ;
- c) déterminer la meilleure note du premier trimestre ;
- d) déterminer l'élève qui a le meilleur note au premier trimestre ;
- e) déterminer la listes des élèves qui ont la meilleure note au second trimestre.

2.2 Châtain clair...

- .1 **Nombres premiers** Construire la table des nombres premiers inférieurs à $N = 1000$.

- première méthode : on construit la table, nombre premier par nombre premier en testant chaque nouveau nombre susceptible d'être premier.
- deuxième méthode : on utilise la méthode du crible d'ERATOSTHÈNE (on élimine d'une table des entiers de 2 à N tous les multiples d'un entier premier).

.2 Tri Triez, par une méthode de votre choix (autre que `sort`!), le tableau *Trim1* défini précédemment par valeurs croissantes. Il faudrait si possible trier « sur place » sans créer un nouveau tableau.

2.3 Réactions successives (Acte I) !

On considère le célèbre système $A \longrightarrow B \longrightarrow C$ ¹.

On propose une approche statistique en modélisant le système par un **automate cellulaire**. Partant d'une population de N entités, on va voir, pour chacune d'elle quelle est sa probabilité de disparaître (ou d'apparaître). On note p la probabilité de disparaître par unité de temps $\Delta t = 1$ (unité arbitraire).

Pour chaque entité du système, considérée comme un réactif, on tire un nombre au hasard (entre 0 et 1 grâce à l'instruction `rd.random()`), si ce nombre est supérieur à p l'entité se transforme, sinon il ne se passe rien².

On réitère alors le processus un certain nombre de fois.

On définit, ici, deux probabilités de transitions p_1 (de A vers B) et p_2 de (B vers C).

Afin de tracer l'évolution des quantités en fonction du temps, on stocke (tous les Δt) dans 4 tableaux : T, A, B, C respectivement le temps, le nombre de molécules A, B et C.

Voici le début, et la fin du programme à réaliser... yapluka !

```

1  import random as rd
2  import matplotlib.pyplot as plt
3
4  #paramètres du système
5  N = 10000
6  p1 = 0.90
7  p2 = 0.95
8  tmax = 100
9
10  """ A vous de jouer """
11
12
13  plt.plot(T, A, 'r')
14  plt.plot(T, B, 'g')
15  plt.plot(T, C, 'b')
16  plt.show()
```

1. Aïe, Corinne, pas sur la tête! Bon, OK, je contextualise... en DS je posais une filiation radioactive utilisée en médecine nucléaire où le $^{99}_{42}\text{Mo}$ (noté A) est précurseur du $^{99}_{43}\text{Tc}$ (noté C). Un intermédiaire métastable (noté B) est formé par une première transformation de demi-vie 66 h et décomposé par une seconde transformation de demi-vie 6 h.

2. Pour des réactions d'ordre 1 : $p = 1 - e^{-k\Delta t}$ avec $\Delta t = 1$ (unité arbitraire) correspondant à 1 tirage au sort

3

Quelques pistes...

3.1

Blondes

.1 A tout hasard... la table des codes ASCII permet de passer d'une lettre à un nombre et vice versa.

.2

- a) fastoche !
- b) un jeu d'enfant.
- c) idem.
- d) ne me dites pas que vous n'y arrivez pas !
- e) un peu plus malin ! Essayez de récupérer la liste des meilleurs élèves à l'aide d'une seule boucle !

3.2

Châtain clair...

.1 Nombres premiers

- On ajoute dans la liste des nombres premiers en cours de construction tout nombre qui n'admet pas de diviseur dans cette liste de nombre premiers. On pourra utiliser l'opérateur modulo (%) : $a \% b$ renvoie le reste de la division euclidienne de a par b . Ainsi : $a \% 2 = 0$ si a est pair, 1 si a est impair.
- L'idée est de partir d'une liste qui contient $[0, 0, 2, 3, 4 \dots, N]$. On parcourt la table en mettant à 0 tous les multiples d'un nombre non nul. Et enfin, on récupère les nombres non nuls !

.2 **Tri** Un des plus simples à programmer est le tri par sélection :

- on prend le premier élément de la liste ; on le permute avec le plus petit élément de la liste (à sa droite) ;
- on passe au second que l'on permute avec le plus petit élément de liste (à sa droite) ;
- et on recommence !

3.3

Réactions successives

Une fois la modélisation du système effectuée, on se retrouve, classiquement devant deux difficultés :

- comment implémenter le modèle proposé ?
- comment résoudre ?

Avec un peu d'habitude, et pour des systèmes pas trop complexes, vous deviendrez des as en programmation et il restera juste à se poser les questions : comment modéliser ? et quelle est l'implémentation la plus simple du modèle ?

Bref, ici, on a manifestement 3 états (A, B et C). Le plus simple est de coder cet état par un entier valant respectivement (0, 1 ou 2). L'automate sera alors implémenté par une liste de N entiers, initialement initialisés à 0.

Tous les $\Delta t = 1$, et pour chaque cellule, on tire un nombre au hasard entre 0 et 1 :

- si la cellule est dans l'état 0 et que le nombre est supérieur à p_1 , elle passe dans l'état 1 ;
- si la cellule est dans l'état 1 et que le nombre est supérieur à p_2 , elle passe dans l'état 2 ;
- si la cellule est dans l'état 2... il n'y a rien à faire

Il reste alors à compter combien on a de 0, de 1 et de 2 et stocker ces valeurs dans chaque tableau.

4

Solutions...

4.1

Blondes

```

.1
1 L1 = []
2 for i in range(0,11):
3     L1.append(i)
4 # ou L1 = [i for i in range(0,11)], idem pour les suivantes
5
6 L2 = []
7 for i in range(10,-12,-2):
8     L2.append(i)
9
10 L3 = []
11 x = 0
12 while x <= 10 :
13     L3.append(x)
14     x = x + 0.1
15
16 # ou
17 #L3 = []
18 #for i in range(0,101):
19 #    L3.append(i/10)
20
21 L4 = []
22 for i in range(ord("a"), ord("a")+26):
23     L4.append(chr(i))
24 print(L4)

```

- au début, on construit les listes avec des `append` puis, avec l'habitude, on utilise les listes par compréhension !
- `ord` récupère le caractère ASCII de la lettre, on incrémente ce code et on récupère la caractère à l'aide de `chr`

```

.2
1 #a)
2 n = 0
3 for i in range(len(Trim1)):
4     if Trim2[i] > Trim1[i]:
5         n = n + 1
6 print ("Nombre de progrès : ", n)
7
8 #b)
9 s = 0
10 for note in Trim1 :
11     s = s + note
12 moyenne = s / len(Trim1)
13 print("Moyenne : ", moyenne)
14
15 #c)
16 maxi = Trim1[0]

```

```

17 for i in range(1, len(Trim1)):
18     if Trim1[i] > maxi :
19         maxi = Trim1[i]
20 print("Meilleure note : ", maxi)
21
22 #d)
23 maxi = Trim1[0]
24 indice = 0 # ou best = Eleves[0]
25 for i in range(1, len(Trim1)):
26     if Trim1[i] > maxi :
27         maxi = Trim1[i]
28         indice = i # ou best = Eleves[i]
29 print("Meilleure note : ", maxi, " pour ", Eleves[indice])
30 # ou print("Meilleure note : ", maxi, " pour ", best)
31
32 #e)
33 maxi = Trim2[0]
34 best = [Eleves[0]]
35 for i in range(1, len(Trim2)):
36     if Trim2[i] == maxi :
37         best.append(Eleves[i])
38     elif Trim2[i] > maxi :
39         maxi = Trim2[i]
40         best = [Eleves[i]]
41 print("Meilleure note : ", maxi, " pour ", best)

```

- a) rien à dire puisque je ne vous parle plus des problèmes d'indices !
- b) un classique
- c) on initialise maxi, ici, avec la valeur du premier élément de la liste, on aurait tout aussi bien pu prendre -1 comme valeur.
- d) Il faut, bien sûr, repérer en même temps que le maximum, l'indice correspondant (ou directement le nom dans la table Eleves).
- e) Une solution « bourrin » consisterait à rechercher la meilleure note puis de parcourir la liste Eleves à la recherche des élèves qui ont cette note. En une seule boucle, on fait comme précédemment, simplement best n'est plus un nom (ou un indice) mais la liste des noms. Si on trouve une meilleure note, il faut réinitialiser cette liste (ligne 40).

4.2 Châtain clair...

.1 Nombres premiers

```

1 N = 10000
2
3 start = time.time()
4 P = []
5 for i in range (2,N+1):
6     trouve = False
7     j = 0
8     while j < len(P) :
9         if i%P[j] == 0:
10             trouve = True
11             j = j + 1
12     if not trouve :
13         P.append(i)

```

```

14 interval = time.time() - start
15 print (P)
16 print (interval)
17
18 start = time.time()
19 E = [0,0]# 0 et 1 ne sont pas premiers !
20 for i in range(2, N+1):
21     E.append(i)
22 for i in range(2, int(sqrt(N))+1):
23     if E[i] != 0 : # c'est un nombre premier
24         for j in range(2*i, N+1, i):
25             E[j]= 0
26 Pr = []
27 for i in range(N+1):
28     if E[i]!= 0 :
29         Pr.append(i)
30 interval = time.time() - start
31 print (Pr)
32 print (interval)

```

Pour le fun, j'ai ajouté un décompte du temps nécessaire à l'exécution d'un bloc de code. La méthode « bourrin » prend (pour $N = 1000$) 3,41 s alors que celle utilisant le crible d'ERTOSTHÈNE ne prend que 5,4 ms (sur mon ordinateur).

Sinon, initialiser le tableau E avec 0 et 0 (pour les indices 0 et 1) permet d'avoir l'indice d'un nombre égal à ce nombre.

.2 tri

```

1 for i in range(len(Trim1)):
2     mini = Trim1[i]
3     indice = i
4     for j in range(i+1, len(Trim1)):
5         if Trim1[j] < mini :
6             mini = Trim1[j]
7             indice = j
8     tmp = Trim1[i]
9     Trim1[i] = mini
10    Trim1[indice] = tmp
11 print (Trim1)

```

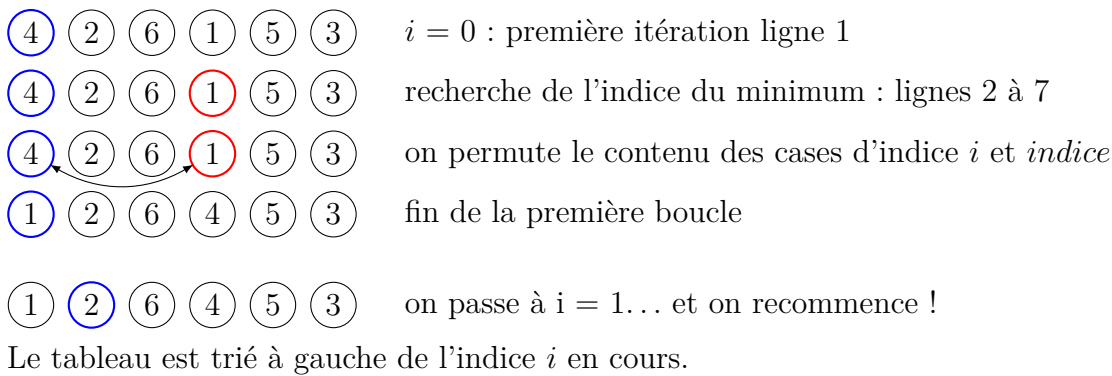
Il faut, bien sûr, deux boucles imbriquées ici :

- la première, indice i , permet de pointer vers la position que l'on est en train de traiter dans la liste ;
- la seconde, indice j , permet de chercher le minimum dans la liste à droite de i .

Lorsque ce minimum est trouvé, on permute le contenu de $L[i]$ et $L[\text{indice du minimum}]$. Une variable intermédiaire tmp est nécessaire pour ne pas perdre la valeur de $Trim1[i]$ ³.

Ci dessous un exemple (avec moins de valeurs pour que ça tienne en largeur) :

3. Vous verrez certains de vos élèves utiliser une astuce python pour permuter 2 nombres ; il suffit simplement d'écrire $a,b = b,a$ et le tour est joué ! Je n'aime pas trop ce type de raccourci car c'est vraiment spécifique à ce langage).



4.3

Réactions successives

```

1  import random as rd
2  import matplotlib.pyplot as plt
3
4  #paramètres du système
5  N = 10000
6  p1 = 0.90
7  p2 = 0.95
8  tmax = 100
9
10 #initialisation de l'automate et des tableaux
11 Automate = [0 for i in range(N)]
12 T = [0]
13 A = [N]
14 B = [0]
15 C = [0]
16
17 t = 0
18 while t < tmax :
19     #on gère les transitions
20     for i in range(N):
21         x = rd.random()#nombre aléatoire entre 0 et 1
22         if Automate[i] == 0 and x > p1 :
23             Automate[i] = 1
24         elif Automate[i] == 1 and x > p2 :
25             Automate[i] = 2
26     #on compte le nombre d'états
27     nbA = 0
28     nbB = 0
29     nbC = 0
30     for i in range(N):
31         if Automate[i]==0 :
32             nbA = nbA+1
33         elif Automate[i]==1 :
34             nbB = nbB+1
35         else :
36             nbC = nbC + 1
37     # on n'oublie pas d'incrémenter le temps
38     t = t+1
39     #on stocke les valeurs 1 fois sur 10
40     T.append(t)
41     A.append(nbA)

```

```
42     B.append(nbB)
43     C.append(nbC)
44
45 plt.plot(T, A, 'r')
46 plt.plot(T, B, 'g')
47 plt.plot(T, C, 'b')
48 plt.show()
```

Ce n'est finalement pas si compliqué...

- on définit les paramètres du système ;
- on initialise les variables nécessaires. Par rapport à ce qui était donné dans l'énoncé, il faut « juste » rajouter l'automate !
- on fait une boucle (ici une boucle `for t in range(1, tmax+1)` aurait été tout à fait possible) ;
- dans cette boucle on teste les transitions pour chacune des cellules
- et on compte combien on en a.
- Il n'y plus qu'à tracer les courbes !

Activité 3

Fonctions, tracé de courbes

1 Quelques rappels « théoriques »...

1.1 Fonctions et procédures

a Objectif

Dès qu'un programme s'étoffe un peu, il devient peu lisible s'il est écrit « linéairement ». On préfère en général (et c'est ce que nous avons vu dans le chapitre précédent) le décomposer en plusieurs « sous programmes » (ensemble d'instructions) appelés par le programme principal.

Parfois certaines actions doivent être effectuées plusieurs fois... les écrire dans un sous programme est alors un choix judicieux.

Certaines instructions peuvent également être utilisées dans différents programmes ; les inclure dans un sous programme (puis éventuellement dans une bibliothèque) permettra de les avoir facilement à disposition.

b Implémentation

On distingue deux types de « sous programme » :

- les `procédures` qui se « contentent » d'exécuter une série d'instruction sans retourner de valeur ;
- les `fonctions` qui retournent systématiquement une (ou plusieurs) valeurs grâce à l'instruction `return`.

Procédures et fonctions sont :

- définis par leur nom grâce à l'instruction `def maFonction(x,y) :` où *maFonction* est le nom que l'on souhaite donner à la fonction ou à la procédure et *x*, *y*... les paramètres que l'on souhaite transmettre à la fonction (ou à la procédures). Ces paramètres, en nombre quelconques sont introduits entre parenthèses et séparés par des virgules ;
- appelés dans le corps du programme principal par « *maFonction*(2,3) par exemple ;
- dans le cas d'une fonction, la(les) valeur(s) retournée doivent être stockées dans des variables : « *z = maFonction*(2,3) » ; la variable *z* contient alors le résultat du calcul effectué par *maFonction* lorsqu'elle reçoit les valeurs 2 et 3 comme paramètres.

c Pièges et difficultés

La principale difficulté, au début, est de bien comprendre la notion de paramètres à passer à une fonction ou à une procédure. Comme sur votre calculatrice, il ne suffit pas d'écrire *sin*, par exemple,

pour que le sinus du nombre auquel on pense soit calculé!

L'erreur classique, ensuite, consiste à supposer que du moment qu'on appelle une fonction avec les bons paramètres, le résultat du calcul est stocké quelque part... Le calcul est bien effectué mais la valeur de retour est perdue si on ne la stocke pas dans une variable : il faut absolument, dans le cas d'une fonction, avoir une structure du type : `variable_résultat = maFonction(paramètres)`. Abordons, ici, un cas plus délicat ; celui où une fonction retourne plusieurs valeurs.



On ne peut avoir qu'un seul « return » par fonction... les différentes valeurs ne doivent donc pas être retournées par plusieurs « return » de suite.

Par contre, on peut écrire « return valeur1, valeur2 » ; et on récupérera les résultats sous la forme `variable1, variable2 = maFonction(paramètres)`.

Par exemple, la fonction suivante retourne le périmètre et la surface d'un carré.

```
1 def carre(a):
2     p = 4*a
3     s = a*a
4     return p, s
5
6 perimetre, surface = carre(3)
7 print("Périmètre : " + str(perimetre) + " surface : " +
      str(surface))
```

1.2 Application : tracé de courbes

Une des principales applications des fonctions est quand même de pouvoir définir... une fonction ! et tracer sa courbe représentative.

Soit, par exemple, la représentation de la fonction $\sin(k \times x)$ entre 0 et $2 * \pi$.

Le programme « élémentaire » pourrait être le suivant. Il fait appel à la bibliothèque « matplotlib.pyplot » pour le tracé de courbes. Cette bibliothèque est extrêmement complète mais son utilisation élémentaire est très simple.

```
1 from math import sin, pi
2 import matplotlib.pyplot as plt
3
4 X = [] #tableau de valeurs pour les abscisses
5 Y = [] #tableau de valeurs pour les ordonnées
6 dX = pi/100 #incrément des valeurs sur x
7
8 def sinkx(k,x): #on définit la fonction que l'on veut étudier
9     return sin(k*x)
10
11 x = 0 #valeur de x en cours
12 while x <= 2*pi :#on parcourt une boucle tant que x est <= à 2
13     pi (cf chapitre suivant)
14     X.append(x) # on complète le tableau des abscisses
15     Y.append(sinkx(4, x)) # et celui des ordonnées
16     x = x + dX # on n'oublie pas d'incrémenter la valeur de x
17
18 """ cette partie vous est donnée """
19 plt.plot(X, Y)# on a, au minimum, besoin des deux tableaux de
    valeurs X et Y
20 plt.show()
```

On se reportera à l'annexe pour enrichir éventuellement ce graphe.

1.3 Application : fonction récursive

Une fonction (ou un programme) est dite récursive si elle s'appelle elle-même, une ou plusieurs fois. Un exemple très classique est la programmation de la fonction factorielle.

$$fact(n) = \begin{cases} 1 & \text{si } n = 0 \\ n \times fact(n-1) & \text{sinon} \end{cases}$$

Fonction $fact(n)$

Entrées : un entier n

Sorties : la quantité $n!$

si $n == 0$ **alors**

| retourner 1

sinon

| retourner $n \times fact(n-1)$

fin

```
1 def fact(n) :
2     if n==0 :
3         return 1
4     else :
5         return n*fact(n-1)
```

2

A vous de jouer...

2.1

Blondes

Il n'y en a plus désormais !

2.2

Châtain clair...

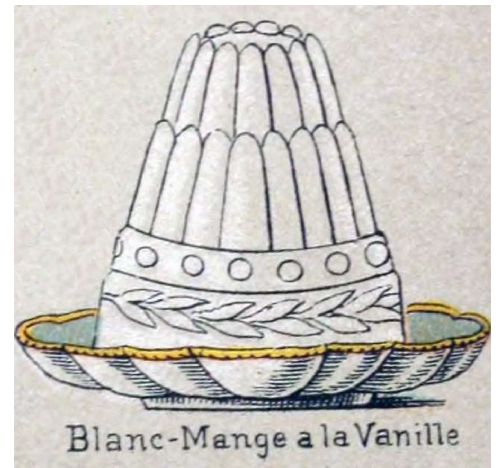
.1 Courbe du Blanc-Mange

Dans cet exercice, on cherche à tracer une approximation de la courbe du Blanc-Mange (en référence à un dessert qui ressemble à cette courbe).

On définit les fonctions B_n , avec n un entier par :

$$B_n(x) = \sum_{k=0}^n \frac{1}{2^k} \left| 2^k x - E \left(2^k x + \frac{1}{2} \right) \right|$$

- La fonction E est la fonction partie entière. On la trouve en python sous le nom `floor`, dans la bibliothèque `math`. (`floor(12.95) = 12`)
- Les barres verticales désignent la valeur absolue.



Plus n est grand, plus la courbe de B_n ressemble à la "vraie" courbe du blanc-mange.

1. Créer une fonction python $B(n, x)$ qui renvoie la valeur de $B_n(x)$.
2. Tracer la courbe de B_n sur $[0, 1]$. A tester pour $n = 10, 100, 1000$ par exemple.

.2 Triangle de Pascal Écrire un programme qui affiche le triangle de pascal à n lignes. On pourra d'abord programmer une fonction qui prend en paramètre une liste qui contient une ligne du triangle, et renvoie la ligne suivante.

L'exemple suivant montre une façon de formater le texte avec des colonnes bien alignées.

```

1 liste1 = [2,12,52]
2 liste2 =[152,0, 6]
3 chaine1 = ""
4 chaine2 = ""
5 for n in liste1 :
6     chaine1 = chaine1 + "{:<4d}".format(n)
7 for n in liste2 :
8     chaine2 = chaine2 + "{:<4d}".format(n)
9 print(chaine1)
10 print(chaine2)
```

2.3 Réactions successives (Acte II)

Retour sur les réactions successives : $A \xrightarrow{k_1} B \xrightarrow{k_2} C$.

Partant d'une concentration A_0 en **A**, l'expression analytique des concentrations en fonction du temps est :

$$A(t) = A_0 e^{-k_1 t}; B(t) = \frac{k_1 A_0}{k_2 - k_1} (e^{-k_1 t} - e^{-k_2 t}); C(t) = A_0 \left(1 + \frac{k_1 e^{-k_2 t} - k_2 e^{-k_1 t}}{k_2 - k_1} \right)$$

1. Définir les fonctions $A(k_1, k_2, t)$, $B(k_1, k_2, t)$ et $C(k_1, k_2, t)$ et tracer pour des valeurs de k_1 et k_2 de votre choix les courbes représentatives.
2. Écrire une fonction $maximum(k_1, k_2)$ qui pour les valeurs de k_1 et k_2 passées en paramètre retourne le temps pour lequel la concentration en **B** passe par un maximum et la valeur de ce maximum de **B**. Faire afficher les valeurs dans les deux cas de figure précédents.

On complique un peu...

3. On prend la valeur $k_1 = 1$. On définit le rapport des constantes de vitesse $r = \frac{k_2}{k_1}$. Tracer la courbe qui donne la concentration maximale atteinte par **B** en fonction du logarithme décimal de r pour $\log_{10}(r)$ variant de -2 à +2.
4. On peut considérer que l'on peut appliquer l'AEQS si la concentration en **B** ne dépasse pas 5% de la concentration initiale A_0 ; quelle est la valeur de r correspondant à cette limite.

2.4 Régression linéaire

On considère un nuage de n points $M_i(x_i, y_i)$ que l'on désire ajuster au mieux par une courbe $y = a \times x + b$. Dans la méthode des moindres carrés, on cherche à minimiser la somme des distances entre les points M_i et la droite.

On note :

- \bar{x} et \bar{y} la moyenne des X et des Y ;
- $\sigma_x = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2}$ et $\sigma_y = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2}$ les écarts types correspondants ;

- $Cov(X, Y) = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x}) \times (y_i - \bar{y})$ la covariance (moyenne du produit des écarts à la moyenne)

Dans ces conditions :

- le coefficient de corrélation linéaire vaut $r(X, Y) = \frac{Cov(X, Y)}{\sigma_x \sigma_y}$;
 - la pente de la droite de corrélation : $a = \frac{Cov(X, Y)}{(\sigma_x)^2}$
 - l'ordonnée à l'origine de la droite de corrélation vaut $b = \bar{y} - a\bar{x}$
1. Écrire la routine *regrelin*(*X*, *Y*). Les paramètres d'appel de cette fonction sont les deux tableaux *X* et *Y* (de même dimension) ; la valeur retournée : un tuple¹ contenant, dans l'ordre, *a*, *b* et le coefficient de corrélation *r*.
 2. Tester à l'aide des tableaux de valeurs *X* = [1, 2, 3, 4, 5] et *Y* = [3.1, 4.9, 7, 8.9, 11.2]. On trouve, sur Excel *a* = 2,02, *b* = 0,96 et *r*² = 0,99843.
 3. Etude de la réaction de saponification du propanoate d'éthyle. On donne les tableaux de valeur *theta* = [35, 40, 45, 50] et *k* = [0.188, 0.257, 0.354, 0.477]. Le premier correspond aux températures (en °C) ; le second aux constantes de vitesse (en $\text{mol}^{-1} \cdot \text{L} \cdot \text{s}^{-1}$).
 - a. Introduire les tableaux de valeur *theta* et *k* ; construire les tableaux de valeurs *unSurT* et *lnk* contenant, respectivement, $\frac{1}{T(\text{enK})}$ et $\ln(k)$.
 - b. Tracer la courbe en ne matérialisant que les points expérimentaux.
 - c. Faire la régression linéaire à l'aide de votre routine *regrelin*. Afficher la valeur de l'énergie d'activation de la réaction.
 - d. Superposer au graphique précédent la droite de régression.
 4. Écrire une routine *regrelinEntre*(*X*, *Y*, *xmin*, *ymin*) qui effectue une régression linéaire sur les tableaux *X* et *Y* en se limitant aux abscisses comprises entre *xmin* et *ymin*.
 5. Copier l'intégralité du code correspondant aux regressions linéaires dans un fichier nommé, par exemple utilitaire.py que vous placez dans votre dossier de travail. Ecrire from utilitaire import * permet alors d'utiliser ces routines dans n'importe quel programme. Essayez !

1. 3 valeurs séparées par des virgules

3 Quelques pistes...

3.1 Blondes

Le début de l'exercice sur les courbes de répartition est très simple. L'exercice sur la régression linéaire montre comment articuler les différents appels de fonctions pour résoudre un problème posé.

3.2 Châtain clair...

.1 Courbe du Blanc-Mange

C'est surtout un exercice de gestion de parenthèses ! Si on a peur de se tromper dans les priorités, mieux vaut en mettre plus (sans abuser toutefois). Avec spyder, cliquer juste après une parenthèse permet de mettre en surbrillance la parenthèse ouvrante ou fermante correspondante.

Pour $n = 1000$, j'ai pris un pas de 0,001.

.2 Triangle de Pascal

On pourra programmer une fonction qui prend en paramètre une liste qui contient une ligne du triangle, et renvoie la ligne suivante ! Cette approche est peut-être plus simple à programmer qu'utiliser la formule du nombre des combinaisons.

3.3 Réactions successives

1. Au lieu d'avoir k_1 et k_2 paramètres des fonctions, vous pouvez les introduire comme variables globales du système. Pour anticiper un peu sur la question 3, vous pouvez créer une fonction qui génère les 4 tableaux de valeurs nécessaires en passant à cette fonction les paramètres adéquats.
2. Il s'agit de réinvestir :) ce que vous aviez fait la semaine dernière... ne me dites pas que vous ne savez plus faire !
3. Pour éviter le cas de figure $k_1 \approx k_2$ dans l'expression de $B(t)$, je n'ai pas fait de calcul pour $|\log r| < 0,1$.
4. Au début j'avais pris un critère de 1%, il faut alors explorer aller jusqu'à $\log r = 3$.

3.4 Régression linéaire

1. En principe, vous n'avez que 3 fonctions (autre que `regrelin`) à programmer !
2. C'est l'heure de vérité pour votre fonction !
3. Essayez de construire les tableaux `unSurT` et `lnk` à l'aide de listes par compréhension, c'est une façon « élégante » de traiter des tableaux expérimentaux.
Avec `matplotlib`, pour tracer une droite, vous pouvez très bien faire `plt.plot([x1,x2], [y1,y2])`.
4. Bien sûr, `regrelinEntre` va se contenter de créer les sous-tableaux et appeler `regrelin`.
5. Pas de soucis pour peu que le fichier se trouve dans le répertoire de travail. Sinon, il faudra prévenir spyder ou pyzo pour lui dire où aller chercher les fichiers.

4

Solutions...

4.1

Blondes

Allez, courage... vous avez toutes les bases « théoriques », yapluka pratiquer !

4.2

Châtain clair...

.1 Courbe de Blanc-Mange

```

1 def B(n,x):
2     b = 0
3     for k in range(0, n+1):
4         u = (2**k)*x
5         b = b + abs((u-floor(u+0.5))/(2**k))
6     return b
7
8 n = 1000
9 X = []
10 Y = []
11 x = 0
12 while x <= 1 :
13     X.append(x)
14     Y.append(B(n,x))
15     x = x + 0.001
16
17 plt.plot(X,Y)
18 plt.show()

```

J'ai calculé $(2 * k) * x$ à part pour me simplifier la vie avec les parenthèses mais ce n'est pas nécessaire.

4.3

Triangle de Pascal

```

1 def printFormat(liste):
2     chaine= ""
3     for n in liste :
4         chaine = chaine + "{:4d}".format(n)
5     print (chaine)
6
7 def ligneSuivante(ligne) :
8     ligne2 = [1]
9     for i in range(len(ligne)-1):
10         ligne2.append(ligne[i]+ligne[i+1])
11     ligne2.append(1)
12     return ligne2
13
14
15 ligne = [1]
16 for i in range(8) :
17     printFormat(ligne)
18     ligne = ligneSuivante(ligne)

```

J'ai créé une fonction printFormat pour « clarifier » un peu le code, même si on pouvait s'en passer.

On retrouve, ici, la principale difficulté des listes... la gestion correcte des indices !

4.4 Réactions successives

```

1  from math import exp
2  #question 1
3
4  A0 = 1
5
6  def A(k1,k2,t):
7      return A0*exp(-k1*t)
8
9  def B(k1,k2,t):
10     return (A0*k1/(k2-k1))*(exp(-k1*t)-exp(-k2*t))
11
12 def C(k1,k2,t):
13     return A0 - A(k1,k2,t)-B(k1,k2,t)
14
15 def prepare(k1,k2,tmax,dt):
16     tT=[]
17     tA=[]
18     tB=[]
19     tC=[]
20     t = 0
21     while t < tmax :
22         tT.append(t)
23         tA.append(A(k1,k2,t))
24         tB.append(B(k1,k2,t))
25         tC.append(C(k1,k2,t))
26         t = t + dt
27     return tT, tA, tB, tC
28
29 T, cA, cB, cC = prepare(1,2,5,0.01)
30
31 plt.plot(T,cA, 'r')
32 plt.plot(T,cB, 'g')
33 plt.plot(T,cC, 'b')
34 plt.show()
35
36 #question 2
37
38 def rechercheMax(X):
39     maxi = X[0]
40     indice = 0
41     for i in range(len(X)):
42         if X[i] > maxi :
43             maxi = X[i]
44             indice = i
45     return indice, maxi
46
47 imax, bmax = rechercheMax(cB)
48 print("Bmax = ", bmax, " pour t = ", T[imax])
49
50 # question 3

```

```

51
52 LogR = []
53 Bmax = []
54 logr = -2
55 while logr <= 2 :
56     if abs(logr) > 0.1 : # problème si k1 = k2 dans B!
57         r = 10**logr
58         T, cA, cB, cC = prepare(1,r,10,0.01)
59         imax, bmax = rechercheMax(cB)
60         LogR.append(logr)
61         Bmax.append(bmax)
62         logr = logr + 0.1
63
64 plt.plot(LogR, Bmax)
65 plt.show()
66
67 # question 4
68
69 i = 0
70 trouve = False
71 while not trouve and i < len(Bmax):
72     if Bmax[i] <= 0.05 :
73         trouve = True
74         rAEQS = 10**LogR[i]
75         i = i + 1
76 if trouve :
77     print ("AEQS pour k2 > " ,round(rAEQS), " k1")

```

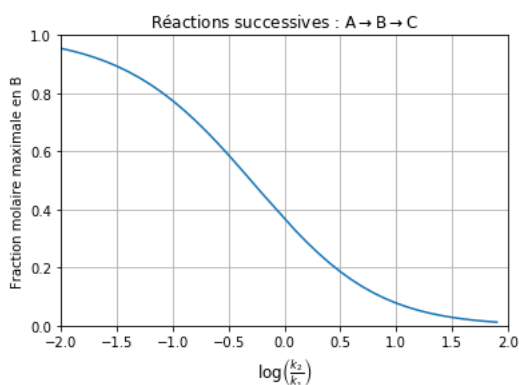
1. Le codage des différentes fonctions ne devrait pas poser trop de difficultés. Pour pouvoir facilement tester différents cas de figure et n'avoir à changer qu'une seule fois les valeurs de k_1 et k_2 , j'ai rajouté la fonction `prepare` qui génère les 4 tableaux nécessaires. On aurait pu mettre k_1 et k_2 comme variable globale s'il n'y avait pas eu la question 3! On pourrait enrichir le graphe!
2. On retrouve le code de recherche d'un extremum avec l'indice de cet extremum.
3. Pour chaque valeur de r , il faut générer le tableau de valeur $B(t)$. On aurait très bien pu écrire une fonction qui ne calcul que B , vue la rapidité des calculs sur les ordinateurs, autant conserver la fonction génère du 1. Attention à bien stocker toutes les variables, même celle qui ne servent à rien! C'est normal, au début, d'être un peu perdu par tous ces appels de fonction, j'en dirai quelques mots dans le chapitre suivant. Bref, en évitant un accident de parcours pour r très proche de 1, on a une jolie courbe.
4. Ici, on cherche une valeur seuil. On aurait très bien pu le faire en même temps que la question 3. J'ai préféré reprendre le code correspondant. Il faut faire attention à s'arrêter dès que B_{max} devient plus petit qu'une valeur déterminée et sortir de la boucle `while`.

Pour les latexiens, la bibliothèque `matplotlib` permet un enrichissement des graphes :

```

1 plt.xlabel(r"$\log\left(\frac{k_2}{k_1}\right)$", fontsize=12)
2 plt.ylabel("Fraction molaire maximale en B")
3 plt.title(r"Réactions successives :
    A$\rightarrow$B$\rightarrow$C")

```



4.5 Régression linéaire

```

1  def moyenne(X):
2      m = 0
3      for x in X :
4          m = m + x
5      return m/len(X)
6
7  def ecart_type(X):
8      m = moyenne(X)
9      u = 0
10     for x in X :
11         u = u + (x-m)**2
12     return sqrt(u/len(X))
13
14  def cov(X,Y):
15     mx = moyenne(X)
16     my = moyenne(Y)
17     c = 0
18     for i in range(len(X)):
19         c = c + (X[i]-mx)*(Y[i]-my)
20     return c/len(X)
21
22  def regrelin(X,Y):
23     pente = cov(X,Y)/ecart_type(X)**2
24     ordonnee = moyenne(Y)-pente*moyenne(X)
25     coeff = cov(X,Y)/(ecart_type(X)*ecart_type(Y))
26     return pente, ordonnee, coeff
27
28  def regrelinEntre(X,Y,xmin,xmax):
29     nX = []
30     nY = []
31     for i in range(len(X)):
32         if X[i] >= xmin and X[i] <= xmax :
33             nX.append(X[i])
34             nY.append(Y[i])
35     return regrelin(nX, nY)
36
37  # test
38  X=[1,2,3,4,5]
39  Y=[3.1, 4.9,7,8.9,11.2]
40

```

```

41 a,b,r = regrelin(X,Y)
42 print(" a = ", a, " b = ", b, " r^2 = ", r**2)
43
44 # détermination d'une énergie d'activation
45 theta=[35,40,45,50]
46 k=[0.188,0.257,0.354,0.477]
47
48 unSurT = [1/(t+273) for t in theta]
49 lnk = [log(x) for x in k]
50
51 a,b,r = regrelin(unSurT, lnk)
52
53 print("Energie d'activation : ", round(- 8.314*a/1000 ), "
      kJ/mol")
54 print("r^2 = ", r**2)
55
56 plt.plot(unSurT, lnk, 'o')
57 plt.plot([unSurT[0], unSurT[len(unSurT)-1]], [a*unSurT[0]+b,
      a*unSurT[len(unSurT)-1]+b])
58 plt.show()

```

1. Les paramètres des fonctions sont muets, `moyenn(X)` peut traiter n'importe quel nom de tableau, qu'il s'appelle *X*, *Y* ou *schtrumph_grincheux*.
On stocke, bien sûr, dans des variables les moyennes calculées dans les fonction `ecart_type` et `cov`.
2. Le test proposé devrait donner les bons résultats!
3. On aurait, bien sûr, pu créer les listes *unSurT* et *lnk* avec des `append`. Quoiqu'il en soit, c'est peut-être plus rapide que d'expliquer les fonctions de tableau dans Graph2D!
Qui a oublié le 273?

5 Annexe : bibliothèque matplotlib

Il ne s'agit pas, ici, de résumer toutes les possibilités de la bibliothèque matplotlib. On pourra se reporter à la page : <http://matplotlib.org> puis aux onglets « exemples » ou « docs » pour exploiter au mieux cette bibliothèque.

Cette bibliothèque permet le tracé de courbes. Dans la version minimale, il suffit de remplir deux tableaux de valeurs (de même taille) et d'appeler le tracé.

```

1 import matplotlib.pyplot as plt
2 X = []
3 Y = []
4 x = 0
5 while x <=5 :
6     X.append(x)
7     Y.append(x/(1+x))
8     x = x + 0.1
9
10 plt.plot(X, Y)

```

L'exemple ci-dessous permet d'enrichir le graphique :

- On ajoute une étiquette sur chaque axe, un titre et la légende des courbes.
- On précise, entre guillemet, que pour la courbe Y en fonction de X , on ne représente que les points tabulés sous forme de ronds rouges et que la courbe Z est représentée en pointillé bleu.
- On a spécifié (lignes 18 et 19) les intervalles de tracé sur chacun des axes.
- On ajoute, ligne 20, du texte aux abscisses et ordonnées spécifiées en noir et avec une fonte de taille 20.
- La ligne 21 permet de préciser les étiquettes représentées sur l'axe y .
- On ajoute une grille (ligne 22).

```

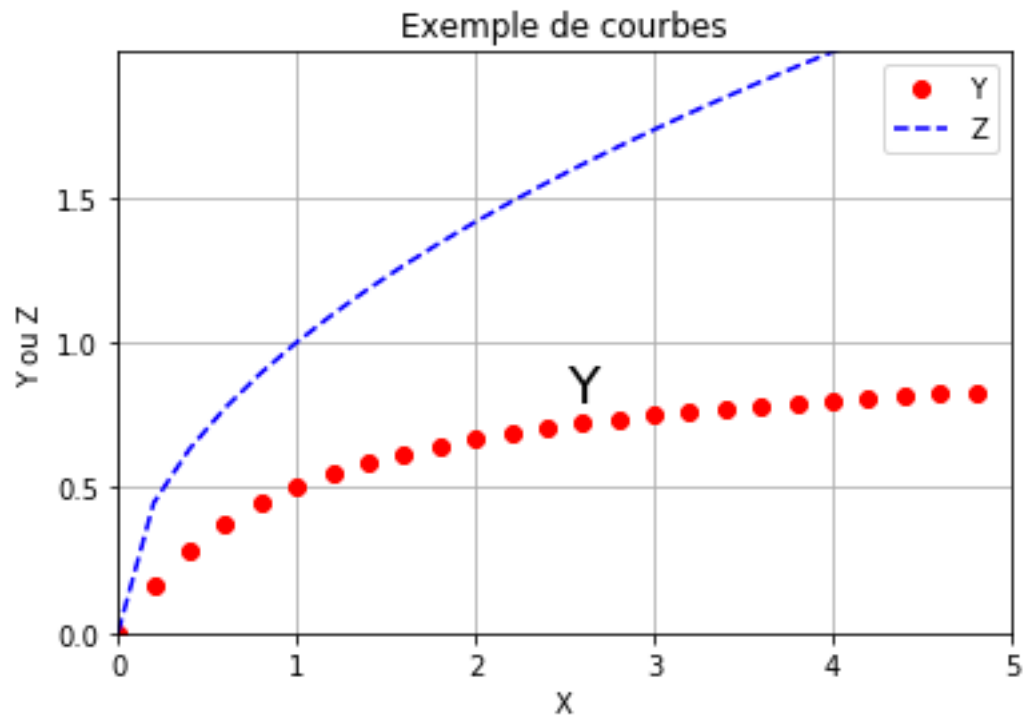
1 import matplotlib.pyplot as plt
2 from math import sqrt
3 X = []
4 Y = []
5 Z = []
6 x = 0
7 while x <=5 :
8     X.append(x)
9     Y.append(x/(1+x))
10    Z.append(sqrt(x))
11    x = x + 0.2
12
13 plt.plot(X, Y, "ro", label="Y")
14 plt.plot(X, Z, "b--", label="Z")
15 plt.xlabel("X")
16 plt.ylabel("Y ou Z")
17 plt.title("Exemple de courbes")
18 plt.xlim(0,5)
19 plt.ylim(0,2)
20 plt.text(2.5, 0.8, "Y", {'color': 'k', 'fontsize': 20})
21 plt.yticks([i/2 for i in range(0,4)])
22 plt.grid()

```

```

23 plt.legend()
24 plt.show()

```



En s'aidant de l'aide en ligne, on peut tracer un graphe avec deux ordonnées.

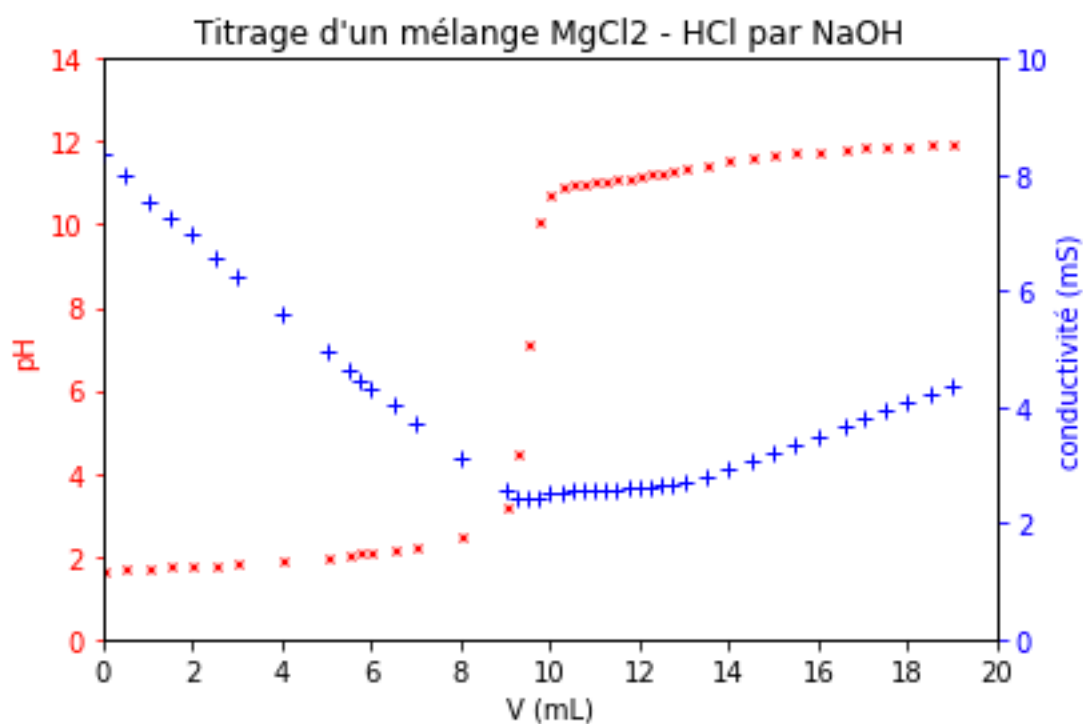
Les tableaux V , PH et G contiennent, respectivement, volume, pH et conductance mesurés lors d'un titrage.

Le code ne s'improvise pas mais, vive le copier-coller !

```

1  fig, ax1 = plt.subplots()
2  ax1.plot(V, PH, "rx", markersize=3)
3  ax1.set_xlabel("V (mL)")
4  ax1.set_ylabel("pH", color="r")
5  ax1.tick_params("y", colors="r")
6  ax1.set_ylim(0,14)
7
8  ax2 = ax1.twinx()
9  ax2.plot(V, G, "b+")
10 ax2.set_ylabel("conductivité (mS)", color="b")
11 ax2.tick_params("y", colors="b")
12 ax2.set_ylim(0,10)
13 ax2.set_xlim(0,20)
14 ax2.set_xticks([i for i in range(0,22,2)])
15
16 plt.title(r"$Titrage d'un mélange MgCl2 - HCl par NaOH")
17 plt.show()

```



On peut même réaliser des animations... en fait, c'est fabuleux tout ce qu'on peut faire avec matplotlib; et encore, on n'est « que » chimiste!

Les latexiens peuvent même écrire en latex²... en écrivant les chaînes de caractères attendues sous la forme `r"expression latex"`. Par exemple, avec `plt.title(r"Titrage d'un mélange MgCl2 - HCl par NaOH")` on obtient MgCl_2 ; on peut bien sûr, avoir également des lettres grecques...

2. Sans savoir comment, ça marche chez moi mais il y a peut-être des liens à établir?

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
   demander 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
   demander 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'incrémentatation 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 suivants¹.

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.

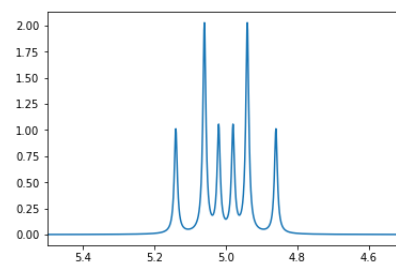
Faire, à l'aide de la routine créée dans le chapitre précédent, trois régressions linéaires pour des volumes compris entre 0 et 8 ; 10 et 12 et 15 et 20. Programmer le calcul et l'affichage des volumes équivalents.

4.2 Doublet de triplet ou triplet de doublet ?

Allez, on complique un peu ² !

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 δ_{ref}

est le déplacement chimique de référence du signal, *I* son intensité. δ 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 δ_{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         for i in range(0, nbFils):
35             Delta.append(deltaRef - d0 + i*d)
36             I.append(intensiteRef*combinaison(nbFils-1, i))
37     return Delta, I
38
39 """ programme principal """
40
41 deltaPic = []
42 intensitePic = []
43 dp, ip = genereFils(deltaA, 1, nbM+1, JAM)
44 for i in range(len(dp)):
45     dp2, ip2 = genereFils(dp[i], ip[i], nbX+1, JAX)
46     for j in range(len(dp2)):
47         deltaPic.append(dp2[j])
48         intensitePic.append(ip2[j])
49
50 deltaMin = deltaA-0.5
51 deltaMax = deltaA+0.5
52 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 !

Activité 5

Courbes de répartition

Rien de nouveau d'un point de vue « théorique » cette semaine... normal, vous savez tout !

Mais... il reste l'essentiel qui nous réunit ici : savoir passer de l'exercice 1 à l'exercice 3.

Pour résoudre l'exercice 1, n'importe quel « bricolage » convient. Il suffit, en gros de savoir comment définir une fonction et comment tracer une courbe pour y arriver. Mais, si on reprend la même question dans l'exercice 3, il faudra réfléchir à la façon de modéliser le système et traiter les données !

Une trame de programme est proposée pour les exercices 2 à 4.

1 A vous de jouer

1.1 Courbes de répartition d'un diacide

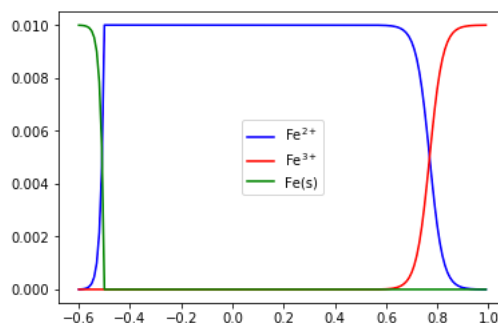
Bon, on commence par un cas simple¹ le tracé des courbes de répartition d'un diacide. Il s'agit donc de tracer le pourcentage du diacide, de l'ampholyte et de la dibase en fonction du pH.

A l'occasion, testez avec vos élèves en leur donnant $\%(A^{2-}) = \frac{100}{1 + \frac{h}{K_2} + \frac{h^2}{K_1 K_2}}$

1.2 To be or not to be !

On complique un peu avec un diagramme d'existence.

L'objectif est de tracer le diagramme suivant représentant l'évolution des quantités de matière (en mole) en fer(s), en ions Fe^{2+} et Fe^{3+} présents dans 1,0 litre de solution en fonction du potentiel.



On définit, comme variables globales du système, n_0 (quantité totale de matière (sous toute ses formes) de l'élément fer dans un litre d'eau), E_1 potentiel standard du couple $\text{Fe}^{2+}/\text{Fe(s)}$, E_2 potentiel standard du couple $\text{Fe}^{3+}/\text{Fe}^{2+}$.

Dans la mesure du possible (sauf si vous n'y arrivez pas autrement !) on ne cherchera pas à déterminer explicitement la valeur de la frontière du domaine d'existence du fer à l'aide d'une approximation de chimiste !

1. Au moins d'un point de vue théorique pour nous !

1.3 Courbes de répartition d'un polyacide

L'idée est de pouvoir tracer les courbes de répartition d'un polyacide sans avoir à programmer le monoacide, le diacide, le triacide... en gros, on aimerait bien, par exemple, tracer les courbes de répartition de l'acide phosphorique par le simple appel de `plot_repartition(H3PO4)`.

Forcément, cette variable *H3PO4* doit correspondre à un certain nombre de données, au minimum, les constantes d'acidité. Comment modéliser le système « H3PO4 » ?

On décide, par exemple, de définir un système sous la forme d'une liste

$systeme = [nom, charge, pK1, pK2]^2$:

- *nom* est le nom de la base (sans indiquer la charge : PO4 par exemple) ;
- *charge* est la charge de l'espèce la plus basique
- viennent ensuite les différents pK_a de l'acide

Par exemple, pour l'acide phosphorique on aurait `H3PO4=["P04",-3,2,7,12]`.

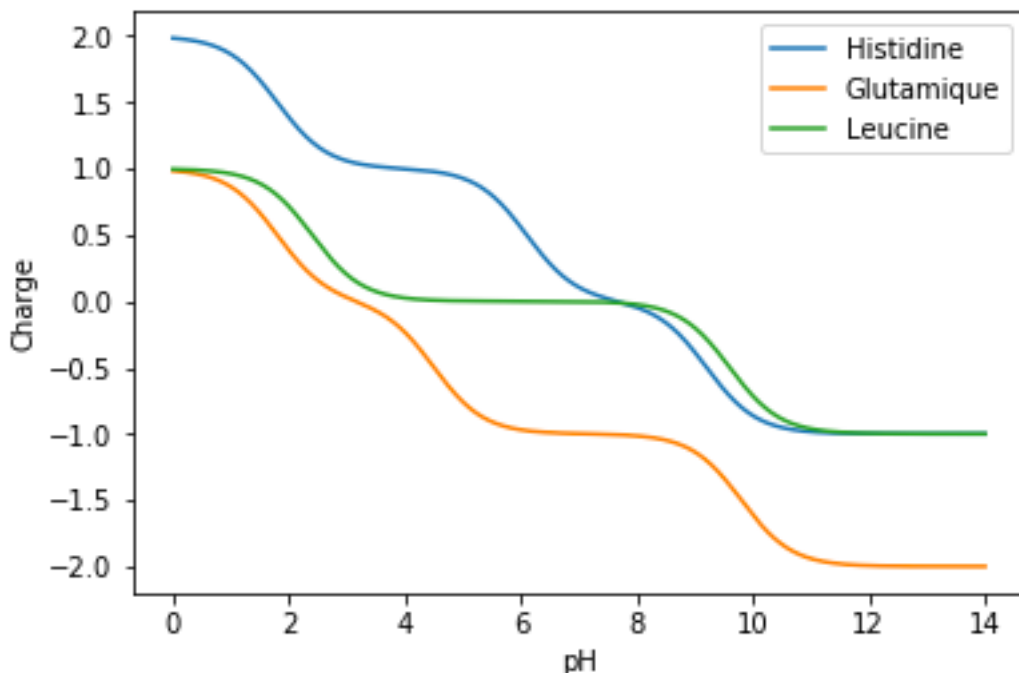
L'appel de `plot_repartition(systeme)` doit alors tracer la courbe de répartition du système étudié quel qu'il soit.

Pour ceux qui sont le plus à l'aise, on générera automatiquement les légendes en codant un éditeur de formule très rudimentaire. Celui-ci nommera les différentes espèces sous la forme : *H2PO4-* ou *HPO42-* par exemple.

1.4 MLL's Challenge

Alors, en utilisant la trame du programme fournie et quelques fonctions de l'exercice précédent, vous pourrez « aisément :) » tracer la charge des acides aminés proposés dans les données en fonction du pH.

Gare au regard noir de MLL si vous n'y arrivez pas !



On utilise la même modélisation des données que dans l'exercice précédent. La trame du programme est :

```
2 Histidine = ["Histidine", -1, 1.8, 6.1, 9.2]
3 Glutamique = ["Glutamique", -2, 1.8, 4.5, 9.8]
4 Leucine = ["Leucine", -1, 2.4, 9.6]
5
6
7 def plot_charge(AA, C0):# AA acide aminé étudié C0
   concentration du système
8     PH = []
9     Q = []
10    ...
11    plt.plot(PH, Q, label = AA[0])
12
13 plot_charge(Histidine, 1)
14 plot_charge(Glutamique, 1)
15 plot_charge(Leucine, 1)
16 plt.ylabel("Charge")
17 plt.xlabel("pH")
18 plt.legend()
19 plt.show()
```

L'objectif est donc de compléter le corps de la routine `plot_charge` afin de tracer la courbe charge Q en fonction du pH PH .

2 Quelques pistes...

2.1 Courbes de répartition d'un diacide

La solution la plus simple est vraisemblablement de coder les fonctions pourcentages du diacide, de l'ampholyte et de la dibase... ou alors, anticiper un peu pour l'exercice 3!

L'appel de ces 3 fonctions dans une boucle `while` sur le pH permet alors de créer 4 tableaux de valeurs PH , $H2A$, HA et A .

2.2 To be or not to be !

To be or not to be! se dit le fer... mais comment savoir s'il y a ou s'il n'y a pas de fer pour une valeur du potentiel donné?

Une première approche consisterait à dire que vu l'écart des potentiels, on n'a pas de Fe(III) à la frontière Fe/Fe(II). On cherche alors le potentiel de cette frontière. Puis on code trace les courbes en faisant un test entre la valeur du potentiel de cette frontière et le potentiel courant.

Peut-être de façon plus élégante, on pourrait définir des fonctions donnant la quantité de Fe(II), Fe(III) et Fe en présence de fer ainsi que la quantité de Fe(II) et Fe(III) en absence de fer. Pour chaque valeur du potentiel, on teste s'il y a ou s'il n'y a pas de fer et on utilise la fonction appropriée.

2.3 Courbes de répartition d'un polyacide

Pas simple tout ça...

Il y a, certe, une méthode de bourrin. On code les pourcentages des différentes formes dans les différents cas de figure et on fait des tests. Mais bon, on n'est pas là pour ça!

Alors, je vous guide un peu vers une solution possible...

En fait, le plus dur est, dans le cas général, de pouvoir calculer, pour un pH et une concentration de référence donnée³ la concentration de l'espèce la plus basique.

Plutôt qu'écrire une fonction mathématique générale, tentons une récurrence du chimiste! on doit

calculer $\frac{C0}{1 + 10^{pK1-pH}}$, $\frac{C0}{1 + 10^{pK2-pH} + 10^{pK2+pK1-2pH}}$,
 $\frac{C0}{1 + 10^{pK2-pH} + 10^{pK2+pK1-2pH} + 10^{pK3-pH} + 10^{pK3+pK2-2pH} + 10^{pK3+pK2+pK1-3pH}}$ etc s'il y a, respectivement, 1, 2, 3 etc acidi-

tés.

Diable, un bel exercice de travail sur les listes en perspectives. Vous essaieriez d'encapsuler tout ça dans une fonction `c_base(C0, pH, pKa)` qui accepte comme paramètre la concentration totale $C0$, le pH de la solution pH et la liste des différents pKa (dans l'ordre dans la variable pKa) et qui retourne la concentration de l'espèce la plus basique. Chez moi cette fonction n'a que 10 lignes.

Dans le début de la fonction `plot_repartition(acide)` il faudra extraire le tableau des pKa nécessaire à la fonction précédente puis... yapluka! Je vous conseille quand même de tracer les courbes une par une dans une boucle `for` sur les différentes espèces présentes.

Si vous y arrivez... bravo! et si en prime on a une légende (rudimentaire) pour les courbes, c'est le paradis!

3. On pourrait se passer de cette concentration mais on en a besoin dans l'exercice suivant et dans d'autres plus tard!

2.4 MLL's Challenge

Bon, le plus dur est passé.

Si vous n'avez pas trouvé la réponse à l'exercice précédent, copiez le code de la fonction `c_base(C0, pH, pKa)`, vous en aurez besoin.

Avec le code de cette fonction, le calcul devrait être plus facile !

3

Solutions...

3.1

Courbes de répartition d'un diacide

Pas de soucis, j'espère dans le code de cet exercice.

```

1  def pct_H2A(pH, pK1, pK2):
2      return 100/(1 + 10**(pH-pK1)+10**(2*pH-pK1-pK2))
3
4  def pct_HA(pH, pK1, pK2):
5      return 100/(10**(pK1-pH) + 1 + 10**(pH-pK2))
6
7  def pct_A(pH, pK1, pK2):
8      return 100/(10**(pK1 + pK2 - 2*pH) + 10**(pK2-pH) + 1 )
9
10 H2A = []
11 HA = []
12 A = []
13 PH = []
14 pKa1 = 4
15 pKa2 = 8
16 pH = 0
17 while pH < 14 :
18     PH.append(pH)
19     H2A.append(pct_H2A(pH, pKa1, pKa2))
20     HA.append(pct_HA(pH, pKa1, pKa2))
21     A.append(pct_A(pH, pKa1, pKa2))
22     pH = pH + 0.01
23
24 plt.plot(PH, H2A, 'r', label = "H2A")
25 plt.plot(PH, HA, 'g', label = "HA-")
26 plt.plot(PH, A, 'b', label = "A2-")
27 plt.show()

```

On pourrait, tout aussi bien avoir des fonctions mettant en jeu h , K_1 et K_2 .

Une prime à ceux qui n'utilisent qu'une seule fonction : la base par exemple et qui calculent de proche en proche le pourcentage de l'ampholyte puis du diacide.

3.2

To be or not to be !

```

1  N0 = 0.01
2  E1 = - 0.44
3  E2 = 0.77
4
5  def nFe2(E):
6      return N0/(1+10**((E-E2)/0.06))
7  def nFe3(E):
8      return N0/(1+10**(-(E-E2)/0.06))
9  def nFe2_Fe(E):
10     return 10**((E-E1)/0.03)
11 def nFe3_Fe(E):
12     return nFe2_Fe(E)*10**((E-E2)/0.06)
13 def nFe(E):
14     return N0 - (nFe2_Fe(E)+nFe3_Fe(E))

```



```

15
16 E=[]
17 FE=[]
18 FE2=[]
19 FE3=[]
20
21 pot = -0.6
22 while pot < 1 :
23     E.append(pot)
24     if nFe(pot)>=0:
25         FE.append(nFe(pot))
26         FE2.append(nFe2_Fe(pot))
27         FE3.append(nFe3_Fe(pot))
28     else :
29         FE.append(0)
30         FE2.append(nFe2(pot))
31         FE3.append(nFe3(pot))
32     pot = pot + 0.01
33
34 plt.plot(E, FE2, 'b', label="Fe2+")
35 plt.plot(E, FE3, 'r', label="Fe3+")
36 plt.plot(E, FE, 'g', label="Fe(s)")
37 plt.ylabel("Quantité de matière")
38 plt.xlabel("E(V)")
39 plt.legend()
40 plt.show()

```

Quelques commentaires sur ce programme :

- Lignes 5 à 8 : on définit les fonctions donnant les quantités de matière en absence de fer.
- En présence de fer : la loi de NERNST permet de déterminer la quantité de Fe^{2+} en présence de fer, d'en déduire celle de Fe^{3+} . On calcule ligne 14 la quantité de fer par différence entre la quantité de référence N_0 et les deux quantités précédentes.
- On remplit classiquement les tableaux de valeur à l'aide d'une boucle `while` sur le potentiel. La clé de la résolution de l'exercice est l'appel de la fonction `nFe(E)` qui permet de savoir s'il y a ou s'il n'y a pas de fer présent et donc de choisir les bonnes fonctions pour la suite.

3.3 Courbes de répartition d'un polyacide

```

1 monoacide = ["A", -1, 4]
2 diacide = ["A", -2, 4, 8]
3 triacide = ["A", -3, 2, 7, 12]
4
5
6 def c_base(C0, pH, pKa):
7     nb_acide = len(pKa)
8     denum = 1
9     for i in range(1, nb_acide+1):
10         s = 0
11         for j in range(len(pKa)-1, len(pKa)-1-i, -1) :
12             s = s + pKa[j]
13         denum = denum + 10**(s - i*pH)
14     c = C0/denum
15     return c

```

```

16
17 def nom(nbH, nomBase, chargeBase):
18     name = ""
19     if nbH > 0 :
20         name = "H"
21         if nbH >= 2 :
22             name = name + str(nbH)
23     name = name + nomBase
24     charge = chargeBase + nbH
25     if charge == -1 :
26         name = name + "-"
27     elif charge == 1 :
28         name = name + "+"
29     elif charge < 0 :
30         name = name + str(abs(charge)) + "-"
31     elif charge > 0 :
32         name = name + str(charge) + "+"
33     return name
34
35
36 def plot_repartition(acide):
37     pKa = [acide[i] for i in range(2, len(acide))]
38     nb_acide = len(acide) - 2
39     for i in range (nb_acide + 1):
40         PH = []
41         PCT = []
42         pH = 0
43         while pH <= 14 :
44             PH.append(pH)
45             c = c_base(100, pH, pKa)
46             for j in range(i) :
47                 c = c * 10**(pKa[len(pKa)-j-1]-pH)
48             PCT.append(c)
49             pH = pH + 0.01
50             plt.plot(PH, PCT, label = nom(i, acide[0], acide[1]))
51
52 plot_repartition(triacide)
53 plt.ylabel("Pourcentage")
54 plt.xlabel("pH")
55 plt.legend()
56 plt.show()

```

Dans la fonction `plot_repartition` :

- On laisse tomber les deux premières grandeurs stockées dans la liste passée en paramètre et on stocke les suivantes dans la variable *pKa*.
- La longueur de cette liste (ou celle d'origine - 2) correspond au nombre d'acides.
- Le plus simple des de tracer les courbes de répartition une par une, en partant de l'espèce la plus basique⁴.
- On a, bien sûr, *nb_acide* + 1 courbes à tracer...

4. Cette solution est loin d'être la plus performante, il vaudrait mieux pour un pH donné calculer toutes les concentrations de proche en proche en partant de la plus basique et de les stocker. Dans ces conditions, il faudrait utiliser des listes de liste et non pas une liste unique comme ici.

- Les lignes les plus difficiles à coder sont les lignes 46 et surtout 47 pour passer de la concentration de la base aux différents acides. Il faudra peut-être tracer quelques courbes aux allures surprenantes avant d'arriver au résultat... une solution empirique permet parfois d'arriver plus vite aux résultats qu'une prise de tête avec une formule mathématique mais, je ne suis pas matheux !

Passons à cette fameuse fonction `c_base`... là encore, j'avoue qu'un peu d'essai-erreur peu être une solution !

- Bien sûr, le nombre d'acide correspond à la longueur du tableau *pKa*.
- Pour la *i*ème acidité, on a, au dénominateur, $1 + i$ termes ; chacun de ces termes est une puissance de 10.
- Ces valeurs de *pKa* doivent être lus de la fin vers le début pour obtenir la somme des termes dans la puissance de 10.

Reste éventuellement à écrire le nom de l'espèce pour la légende. On fait une concaténation de chaîne (on verra une activité bientôt sur les chaînes de caractères) avec H puis le nombre de H (sauf si c'est 1) puis le nom de la base sans la charge puis la charge calculée.

3.4

MLL's Challenge

```

1 Histidine = ["Histidine", -1, 1.8, 6.1, 9.2]
2 Glutamique = ["Glutamique", -2, 1.8, 4.5, 9.8]
3 Leucine = ["Leucine", -1, 2.4, 9.6]
4
5 def plot_charge(acide, C0):
6     pKa = [acide[i] for i in range(2, len(acide))]
7     nb_acide = len(acide) - 2
8     PH = []
9     Q = []
10    pH = 0
11    while pH <= 14:
12        PH.append(pH)
13        q = 0
14        for i in range(nb_acide + 1):
15            c = c_base(C0, pH, pKa)
16            for j in range(i) :
17                c = c * 10**(pKa[len(pKa)-j-1]-pH)
18            q = q + c*(acide[1]+i)
19        Q.append(q)
20        pH = pH + 0.01
21    plt.plot(PH, Q, label = acide[0])
22
23 plot_charge(Histidine, 1)
24 plot_charge(Glutamique, 1)
25 plot_charge(Leucine, 1)
26 plt.ylabel("Charge")
27 plt.xlabel("pH")
28 plt.legend()

```

Ce code ressemble relativement au précédent.

Il faut toutefois inverser les boucles sur le pH et sur les espèces car, pour un pH donné (ligne 12), il faut parcourir toutes les espèces (ligne 15) afin de connaître leur concentration (ligne 16 à 18) puis leur contribution à la charge (ligne 19).

Activité 6

De GGAAATT... à Met-Asp-Gly...

Autrement dit « Quelques éléments de manipulation de texte... »

Je vous propose une version adaptée d'un cours-TD que je propose au lycée¹.

Otto MATO et Kurt KETCHUP, généticiens et cultivateurs de tomates, ont vu leur récolte décimée par un virus. Ils font appel à vous pour synthétiser la protéine responsable afin de mettre au point un antivirus. Afin de remplir votre mission, ils vous envoient :

- le fichier *GenomeTomatoBushy.txt* qui contient le génome de l'ADN du virus tomato bushy stunt virus mis en cause ;
- afin de vérifier l'intégrité du fichier, son code de contrôle est : 0xD009² ;
- une mini base de donnée *Genome.sqlite3* qui contient le codage des acides aminés
- le protocole à suivre... c'est à dire ce cours-TD !

On donne quelques éléments de syntaxe pour le traitement des chaînes de caractères.

ord	retourne le code ASCII (en base 10) du caractère : <code>ord('a') = 97</code>
hex	retourne un nombre en hexadécimal : <code>hex(124) = '0x7c'</code> (Ox pour hexadécimal)
float	convertit, si possible, la chaîne en réel : <code>x = float('3.4')</code>
str	écrit un nombre sous forme de chaîne de caractères : <code>s = str(3.4)</code>
replace	remplace une chaîne de caractères 'x' par une autre 'y' : <code>s = s.Replace('x','y')</code>
split	découpe la chaîne de caractères au caractère spécifié : <code>t = s.split(',')</code>

1	Lecture - écriture d'un fichier texte
1.1	Lecture d'un fichier texte
a	Séquence de l'ADN

Listing 6.1 – Lecture d'un fichier texte

3

1. Comme j'avais quelques doutes sur les résultats, avant de vous l'envoyer, j'ai fait relire la trame du sujet à une collègue de bio. Le principe de la méthode est bon ; c'est déjà ça... mais le fichier proposé ne correspond pas à un bout d'ADN ! Damned, afin d'être exploitable, j'ai pris je ne sais où le plus petit fichier de la base de données et, manifestement... il y a bug ! Le problème est que vous allez trouver des « protéines » qui contiennent, sauf erreur : 4, 34, 14, 28, 28 et 23 acides aminés ! On dira que ce sont des microprotéines ; je vous enverrai un bon fichier un jour. Désolé !

2. Ox devant une chaîne de caractères signifie qu'il s'agit d'un nombre en hexadécimal

```

1 | fichier = open("GenomeTomatoBushy.txt", "r")
2 | seq = fichier.read()
4 | fichier.close()

```

Supposons une organisation des fichiers :

C :

Python

Test

fichier1.txt

Travail

fichier2.txt

script.py

Par défaut, le gestionnaire de fichier, recherche le fichier souhaité dans le dossier contenant le script python. Sur notre exemple, on peut accéder à *fichier2.txt* sans spécifier de chemin d'accès. Si ce n'est pas le cas, il faut le faire :

- soit en relatif par rapport au dossier en cours : pour accéder à *fichier1.txt*, il faudra écrire `open("../Test/fichier1.txt", "r")` ;
- soit en absolu par rapport à la racine du système de fichier (la structure est alors liée au système d'exploitation d'où la nécessité, parfois d'utiliser des slash ou antislash). Il faudra écrire `open("C:/Python/Test/fichier1.txt", "r")`.

Un fichier doit être ouvert `fichier = open(chemin d'accès, "r")` puis... fermé `fichier.close()`. Pour lire le contenu du fichier on peut :

- lire l'intégralité du fichier et le stocker dans une chaîne de caractères `s = fichier.read()` ;
- lire le fichier ligne après ligne et le stocker dans une chaîne de caractères `fichier.readline()`. On peut ainsi faire un traitement ligne après ligne ; il faudra s'assurer de parcourir l'intégralité du fichier ;
- lire toutes les lignes d'un fichier `s = fichier.readlines()` et le stocker dans une liste de chaînes de caractères.

Ici, la séquence n'est constituée que d'une seule chaîne de caractère, la première solution est préférable.

b Exploitation d'un fichier .csv

L'exploitation d'un tel fichier a déjà été abordé dans les activités précédentes !

Si le fichier contient, par exemple, des données expérimentales séparées par des virgules ou point virgules (fichier .csv par exemple) ; la dernière solution est fort intéressante.

Par exemple, lors de l'acquisition d'un signal sinusoïdal, on peut stocker dans un fichier pour chaque point le temps et la valeur de la ddp mesurée (les valeurs sont séparées par une virgule et les points sont séparés par un retour à la ligne) : 0,-3.807332002

0.0009,1.99171806

0.0018,7.155799425

0.0027,10.10245104

0.0036,9.9189054

... `signal = fichier.readlines()` contient alors la liste `['0,-3.807332002', '0.0009,1.99171806', '0.0018,7.155799425', '0.0027,10.10245104', '0.0036,9.9189054'...]`

`s1 = signal[1]` contient la chaîne de caractère `'0.0009,1.99171806'`³. On peut utiliser l'instruction `split` pour découper une chaîne de caractères : `r1 = s1.split(',')` contient alors une liste de deux chaînes de caractères : `['0.0009', '1.99171806']`. Il reste enfin à convertir les chaînes de caractères en nombres réels : `t1 = float(r1[0])`.

1.2 Écriture d'un fichier texte

Lorsque l'on aura terminé notre travail, on enregistrera dans le fichier *GenomeProteine.txt* la liste des acides aminés constituant la protéine du virus.

Listing 6.2 – Ecriture d'un fichier texte

```
1 res = open('GenomeProteine.txt', 'w', encoding='utf8')
2 for p in proteine:
3     print (p)
4     res.write(p)
5 res.close()
```

Il n'y a pas de soucis ici puisque l'on cherche à écrire des chaînes de caractères dans un fichier texte (excepté la précision `encoding = 'utf8'` pour gérer les caractères accentués). Pour écrire des nombres, il faudrait les convertir : la syntaxe la plus simple est `str` mais on peut formater la chaîne de caractère affichée (en particulier, pour préciser le nombre de chiffres significatifs à afficher).

2 Code de contrôle d'un texte

Afin de valider le transfert de données (ou d'un mot de passe!), on peut associer à un texte un code de contrôle (ou signature ou checksum). Une chaîne de caractères, ou un nombre est calculé grâce à un algorithme idoine à partir du texte d'origine et du texte transmis; les deux informations doivent, bien sûr, correspondre.

Question 1 Que fait cet algorithme? Vérifier l'intégrité du fichier transmis.

```
1 def checksum(txt):
2     s1 = 0
3     s2 = 0
4     y = 1
5     for i in range (len(txt)):
6         s1+=ord(txt[i])
7         s2+=y*ord(txt[i])
8         y += 1
9         if y==17:
10             y=1
11     s1 = s1 %256
12     s2 = s2 %256
13     return hex(256*s1+s2)
```

Il y a, bien sûr, plus compliqué!!!

3. avec certains types de fichiers (ou de système d'exploitation) on obtiendra plutôt `'0.0009,1.99171806\n'`; pour supprimer une partie de la chaîne de caractère, on utilise l'instruction `replace` : `s1 = s1.replace('\n', '')`

3 Du génome de l'ADN à la séquence de l'ARN

La variable *seq* (cf listing 6.1) contient la séquences des différentes bases azotées A (adénine), T (thymine), C (cytosine), G (guanine) constitutive d'un des deux brins d'ADN du virus.

Afin de reconstituer le brin d'ADN, il faut compléter la séquence initiale par la séquence, lue en ordre inverse, en permutant A et T d'une part, C et G d'autre part. Par exemple si l'on part de ATTC....TCGA ou obtiendra ATTC....TCGATCGA....GAAT.

Question 2 Ecrire une routine reconstituant le brin d'ADN complet à partir de la séquence *seq* passée en paramètre.

L'ADN, confinée dans le noyau de la cellule, transmet alors son code à l'ARN messenger. L'ADN des chromosomes est traduite en ARN ; la séquence reste inchangée à ceci près que l'on remplace la base T par U (uracile).

Question 3 Ecrire une routine retournant la séquence de l'ARN.

On suppose que la variable *seq_arn* contient cette séquence.
L'ARN messenger est ensuite traduit en protéine.

4 A la recherche du codon « start »

4.1 Recherche « naïve » d'une chaîne de caractères

L'algorithme naïf de recherche de caractère consiste, pour chaque position du texte, à comparer caractère par caractère de gauche à droite le texte et la chaîne recherchée et de continuer ainsi tant que les lettres sont les mêmes. Si on arrive à la fin de la chaîne recherchée, une occurrence de cette chaîne est trouvée et on continue. Tout se passe comme si on déplaçait une « fenêtre » sur le texte... et on regarde si le contenu de la fenêtre correspond à la chaîne recherchée.

Question 4 Ecrire le code python d'une routine `recherche(M,T)` implémentant l'algorithme de recherche naïf d'une chaîne de caractères *M* dans un texte *T*. On attend, en retour, un tableau de tous les indices de début de la chaîne *M* dans le texte *T*.
On testera cette fonction sur un exemple simple.

4.2 Recherche du codon start

On appelle codon la succession de 3 bases azotées (nucléotides). Le codage des diverses protéines dans l'ARN est compris entre un codon start (constitué de 'AUG') et un codon stop (constitué de 'UAA', 'UAG' ou 'UGA').

- entre ce codon start et un codon stop, il doit y avoir un multiple de 3 nucléotides (si on a AUGUGGAFUAA, UAA ne peut pas être un codon stop puisqu'il y a 5 nucléotides après le codon stop).
- le nombre de nucléotides doit être grand... au moins 3×190 dans le vrai virus de la tomate, ici on a des microprotéines!!!
- la séquence ainsi repérée doit posséder en amont et en aval des séquences qui ne sont pas traduites. Ces séquences ne sont pas forcément des multiples de 3. On peut donc avoir un codon start d'indice 100 par exemple, et un autre d'indice 200.

Question 5 Ecrire le code permettant de stocker dans la variable *starts* les indices des différentes occurrences de la chaîne 'AUG' dans *seq_arn*.

5 Structure des protéines

5.1 Recherche du premier enchaînement

Intéressons nous au premier codon start trouvé dans *seq_arn*. Il faut maintenant isoler la chaîne de caractère (chaque caractère correspond à une base azotée) comprise entre ce codon start (inclus) et un codon stop (exclus).

Question 6 Ecrire le code permettant d'extraire cette chaîne de caractères et de stocker dans une variable *sequence0* la liste des différents codons constitutifs. Le contenu de cette variable *sequence0* devrait correspondre à ['AUG', 'GAC', 'GGU', 'UUG']. Attention, on veut une liste de chaînes de 3 caractères !

5.2 Et ainsi de suite...

On complique un peu...

Question 7 Ecrire le code permettant de stocker dans une variable *sequences* la liste des différentes séquences rencontrées. Chacune de ces séquences est, elle-même, une liste de chaînes de caractères. Attention, une séquence 'AUG' comprise entre un codon start et un codon stop n'est pas un codon start!⁴.

5.3 Structure des protéines

On y est presque, un codon (3 bases azotées) codant un acide aminé, il reste à identifier les enchaînements d'acides aminés.

On verra en fin d'année, pour les survivants, quelques notions sur la gestion de bases de données. On l'utilise, ici comme une boîte noire. Je vous donne le code permettant, par exemple, de trouver l'acide aminé correspondant au premier codon de chaque enchaînement. Le codon AUG correspond bien à la méthionine.

```
1 import sqlite3
2 conn = sqlite3.connect('Genome.sqlite3')
3 cur = conn.cursor()
4
5 param = ("AUG",) # il faut la virgule, !!!
6 cur.execute('SELECT AcideAmine FROM GENOME WHERE Codon = ?',
7             param)
8 AA = cur.fetchone()
9 print("Acide aminé trouvé : ", AA)
10
11 cur.close()
12 conn.close()
```

4. *sequences* est donc une liste de listes; on abordera bientôt cette structure de donnée dans le traitement d'image. En fait, ça marche comme une liste... de listes. Par exemple *sequences[1]* donne accès à la première sous liste (d'indice 1) et *sequences[1][3]* donnera accès au quatrième élément de la seconde liste !

a Première microprotéine

Question 8 Modifier ce code afin de stocker dans la variable *proteine0* l'enchaînement des acides aminés correspondant à *sequence0*

b Pour conclure...

Question 9 Ecrire le code permettant de stocker dans une variables *proteines* la liste des différentes protéines ; chacune des protéines étant une liste d'acides aminés.

Question 10 Ecrire la structure de ces différentes protéines dans un fichier texte et... envoyez le moi !

Protéine n°1 - 4 acides aminés : méthionine-acide aspartique-glycine-leucine

Protéine n°2 - 34 acides aminés : ...

6 Pour le fun... à vous de jouer les Merrifield !

C'est juste pour vous montrer que l'on peut alors faire des choses « originales » avec la programmation objet. Mais, ce n'est plus très blond tout ça !

Je vous donne à la fin de la trame de cette activité, le code d'une classe *Robot* qui va simuler les différentes étapes de l'enchaînement des acides aminés et le code qui permet de l'exploiter.

6.1 Classe Robot

Listing 6.3 – Déclaration d'une classe

```

1  class Robot():
2      def __init__(self):
3          self.Resine = ""
4          self.Ajout = ""
5          self.AA = []
6
7      def imprime(self):
8          s = self.Resine
9          if len(self.AA) == 0 and self.Resine != "":
10             s = s + "C1"
11         else :
12             for i in range(len(self.AA)):
13                 if i == 0 and self.Resine == "":
14                     s = "HOOC-"
15                 else :
16                     s = s + "CO-"
17                 s = s + "[" + self.AA[i] + "]-NH"
18                 if i == len(self.AA)-1:
19                     s = s + "2"
20                 else:
21                     s = s + "-"
22             if self.Ajout != "":
23                 s = s + " + " + "HOOC-["+self.Ajout +"]-NH2"
24             print(s)

```

```
25
26     def preparer(self):
27         self.Resine = "Polystère-CH2-"
28         self.imprime()
29
30     def prelever(self, aa):
31         self.Ajout = aa
32         self.imprime()
33
34     def ajouter(self):
35         self.AA.append(self.Ajout)
36         self.Ajout = ""
37         self.imprime()
38
39     def decrocher(self):
40         self.Resine = ""
41         self.imprime()
```

6.2 Instanciation et appel de fonctions

Après, cette classe s'utilise comme n'importe quoi d'autre en python. Ce n'importe quoi d'autre est en fait, la plupart du temps, une classe!

Listing 6.4 – Instanciation et utilisation d'une classe

```
1 r = Robot()
2 r.preparer()
3 for i in range(len(proteine0)):
4     r.prelever(proteine[len(proteine0)-1-i])
5     r.ajouter()
6 r.decrocher()
```

Question 11 Tous les BCPSTistes vont s'empressez d'ajouter des fonctions protection, déprotection à la classe. Chicche :).

Activité 7

Images, traitement d'images

1 Quelques aspects « théoriques »...

1.1 Un tableau (ou une matrice) comme liste de listes

Les types vecteur et matrice n'existent pas en python; deux solutions s'offrent à nous :

- soit utiliser une bibliothèque annexe : **numpy** en l'occurrence; nous aborderons quelques aspects de cette bibliothèque ultérieurement;
- soit implémenter le type vecteur par une liste python et le type matrice par... une liste de liste.

Dans cet activité nous allons utiliser une matrice comme structure de données (chaque élément contient un pixel de l'image); nous verrons ultérieurement quelques éléments d'algèbre linéaire.

a Création

Considérons, par exemple la matrice $M = \begin{bmatrix} 5 & 3 \\ 0 & 7 \\ 4 & 1 \end{bmatrix}$ à trois lignes et deux colonnes.

Cette matrice sera créée par $M = \underbrace{[[\overbrace{5, 3}^{2 \text{ colonnes}}, [\overbrace{0, 7}^{2 \text{ colonnes}}, [\overbrace{4, 1}^{2 \text{ colonnes}}]]]}_{3 \text{ lignes}}.$

Plus généralement, on aura, pour une matrice de nl lignes et nc colonnes :

$$M = [ligne_0, ligne_1, \dots, ligne_i, \dots, ligne_{nl-1}] \text{ c'est à dire : } M = \begin{bmatrix} ligne_0 \\ \dots \\ ligne_i \\ \dots \\ ligne_{nl-1} \end{bmatrix}$$

Chaque ligne ayant nc éléments : $ligne_i = [a_{i,0}, a_{i,1}, \dots, a_{i,j}, \dots, a_{i,nc-1}]$.

Si on veut créer une matrice, il faut commencer par créer les listes « intérieures » puis faire un *append* sur la liste principale. Par exemple, pour créer une matrice de 3 lignes et 4 colonnes remplies de 0 on peut écrire.

Listing 7.1 – Création d'une matrice comme liste de liste

```
1 # création par append de liste
2 M= []
```

```

3 for l in range(3): # on boucle sur 3 lignes
4     L = [] # une ligne vide
5     for c in range(4): # on boucle sur 4 colonnes
6         L.append(0)
7     M.append(L)
8 # création par compréhension
9 M = [[0 for c in range(4)] for l in range(3)]

```

Plus généralement, lorsqu'on écrit la liste par compréhension, le plus simple est peut-être d'opérer de la façon suivante :

$M = []$

⇒ on veut créer une matrice comme liste de liste

$M = [[\text{for indice_colonne in range(nb_colonne)}]$

⇒ on gère le nombre de colonnes, c'est à dire le nombre d'éléments des listes « intérieures »

$M = [[\text{for indice_colonne in range(nb_colonne)}] \text{for indice_ligne in range(nb_ligne)}]$

⇒ on gère le nombre de lignes, c'est à dire le nombre de listes « intérieures »

$M = [[\text{fonction de } i_ligne \text{ et } i_colonne \text{ for } i_colonne \text{ in range(nb_colonne)}] \text{for } i_ligne \text{ in range(nb_ligne)}]$

⇒ on gère l'élément à ajouter, fonction de l'indice de la ligne et de la colonne.

b Accès aux éléments

ATTENTION : on a une liste de liste et non un tableau à deux dimensions ; on ne peut donc pas accéder un élément via l'instruction : $a = M[i, j]$

L'accès à un élément de la matrice se fait par $a = M[\text{indice de la ligne}][\text{indice de la colonne}]$.

Ainsi, par exemple : $M[2][1]$ vaut 1. Lors de l'exécution de cette instruction, on commence par récupérer $M[2]$ c'est à dire la troisième ligne : $[4, 1] \dots$ dont on récupère le second élément, de valeur 1.

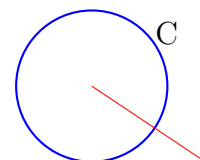
c Dimensions de la matrice

Réfléchissons...

- $len(M)$ retourne le nombre d'éléments dans la liste M , il s'agit donc du nombre de lignes ;
- pour avoir le nombre de colonnes, il suffit d'accéder à une ligne quelconque et de calculer la longueur de cette liste : $len(M[0])$ retourne donc le nombre de colonnes.

1.2 Image bitmap

La plus belle image au monde est certainement le dernier selfie que vous avez réalisé. C'est beaucoup moins fun, mais ce qui est représenté ci-contre est également une image !



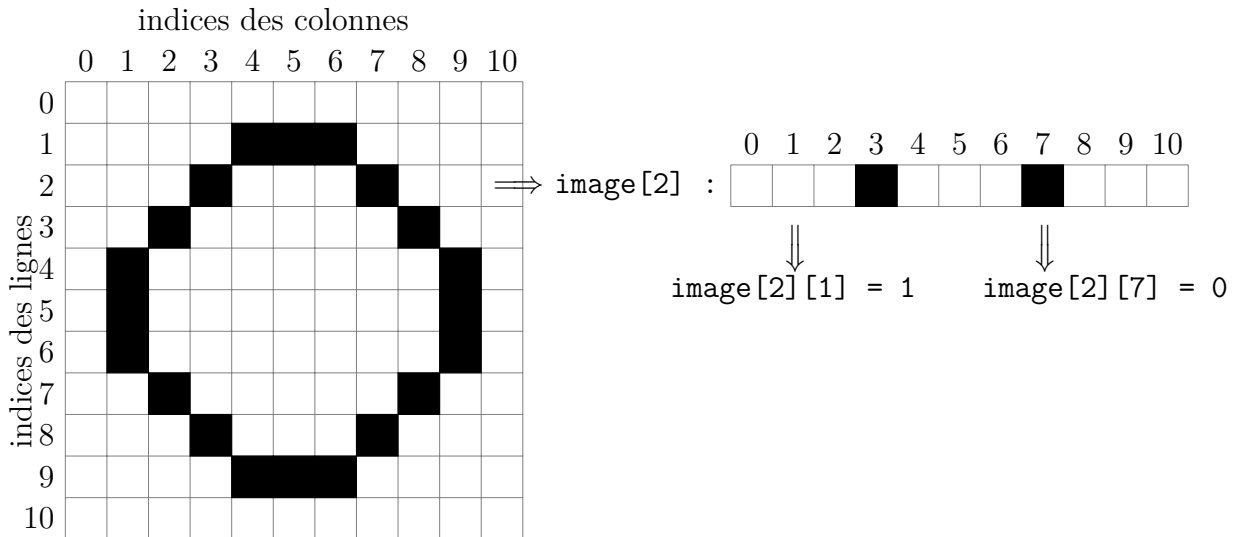
Dans le premier cas, il n'existe guère qu'une solution pour stocker l'information : décrire l'image comme un ensemble de points. Le plus simple des systèmes de codage des couleurs pour chacun de ces pixels est le mode RGB (ou RVB en français). La valeur de chacune des couleurs primaires correspond à un entier variant entre 0 (absence) et 255 (intensité maximale). Le codage de chaque couleur est alors possible sur un octet (8 bits). La combinaison des 3 entiers R , G et B permet ainsi de décrire : $256^3 = 16\,777\,216$ couleurs. Précisons également qu'il s'agit d'une synthèse additive des couleurs. **Le noir sera codé par (0,0,0) ; le blanc par (255,255,255).**

Nous allons aborder, ici, une manipulation de « bas niveau » de l'image en traitant l'image pixel par pixel à l'aide de code python !

Nous utiliserons pour celà la bibliothèque `matplotlib.pyplot` pour afficher les images. La façon la plus simple de préciser la « couleur » d'un pixel est alors de donner une liste de trois nombres réels `[r,g,b]` correspondant respectivement au rouge, vert, bleu (chaque nombre étant compris entre 0 et 1) ; ce qui revient à diviser la valeur RGB par 255 !

a De la matrice de points à l'image, et vice versa

L'image sera implémentée sous forme de liste de listes (une liste de lignes donc !). Prenons l'image d'un cercle.



Si la variable `image` est associée à la matrice précédente on peut :

- accéder à un élément de l'image par `image[indice ligne][indice colonne]` (attention à bien respecter l'ordre : l'instruction se lit de la gauche vers la droite, on accède d'abord à la ligne puis à la colonne !);
- accéder au nombre de ligne par `len(image)` ;
- accéder au nombre de colonne par `len(image[0])` (en fait on demande combien il y a d'éléments dans la première ligne!).

b Création d'une image à l'aide de code python

b.1 Image noire et blanc ou niveau de gris

On code le niveau de gris par un entier compris entre 0 et 1 : 0 correspond au noir et 1 au blanc. Pour créer une matrice (de petite taille) on pourra écrire quelque chose comme :

`image1 = [[ligne 1], [ligne 2], ..., [ligne nbLig - 1]]` et pour chaque ligne : `[colonne 0], [colonne 1], ..., [colonne nbCol-1]`.

C'est un peu fastidieux...

On peut aussi, créer une matrice qui ne contient, par exemple que des 1 (on a ainsi un fond blanc) et modifier les points un par un. `M = [[1 for c in range(nbCol)] for l in range(nbLig)]`, toujours dans cet ordre puisqu'on crée (en lisant de l'extérieur vers l'intérieur) `nbLig` listes qui contiennent `nbCol` valeurs.

Une fois l'image créée, on l'affiche à l'aide de l'instruction : `plt.imshow(image, cmap='gray')` (pour une image en niveau de gris).

b.2 Image en couleur Une liste de trois valeurs (réelles comprises entre 0 et 1) permet de définir les composantes rouge-vert-bleu : $[r, g, b]$...une image en couleur est donc une liste de listes de listes !

Pour afficher l'image *image*, on utilisera l'instruction `plt.imshow(image)`.

b.3 Importer une image Après avoir importé la bibliothèque `matplotlib.image` à l'aide de l'alias `img`, on peut importer une image de format .jpg ou .png à l'aide de l'instruction `img.imread(adresse de l'image)`.

1.3 Traitement d'image

Après avoir importé une image dans un logiciel de traitement d'image, on peut aisément modifier certains de ses aspects : luminosité, contraste, couleurs...

1.4 Niveau de gris

Convertir une image en niveaux de gris revient à :

- n'utiliser qu'une seule valeur pour coder le niveau de gris ;
- utiliser un triplet de trois valeurs identiques si on désire conserver le format d'origine de l'image rgb.

Pour convertir une image couleur en niveau de gris :

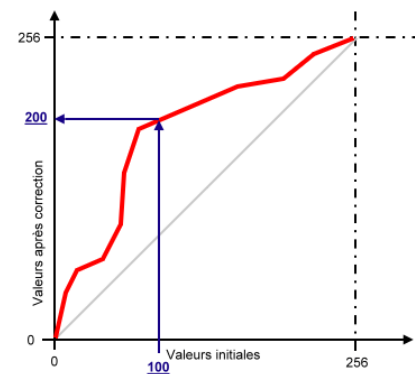
- soit on fait la moyenne des trois valeurs : $gris = \frac{(r + g + b)}{3}$;
- soit on utilise une formule un peu plus sophistiquée qui tient compte de la sensibilité de l'oeil humain. La plus courante est : $gris = 0,299 \times rouge + 0,587 \times vert + 0,114 \times bleu$.

a Luminosité, contraste

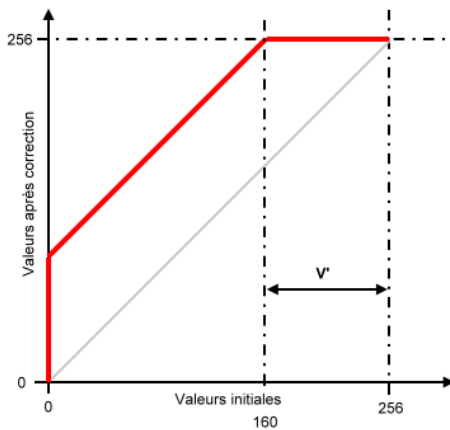
Retoucher une image revient à modifier les valeurs de certains pixels. On peut le faire localement (à un endroit précis de l'image) ou globalement. Dans ce dernier cas, on utilise un outil appelé « courbe tonale » (dessin ci-contre ^a).

Sur l'abscisse, on lit les valeurs originales des pixels et sur l'ordonnée les valeurs après modifications. La diagonale représente la courbe où il n'y a aucune modification

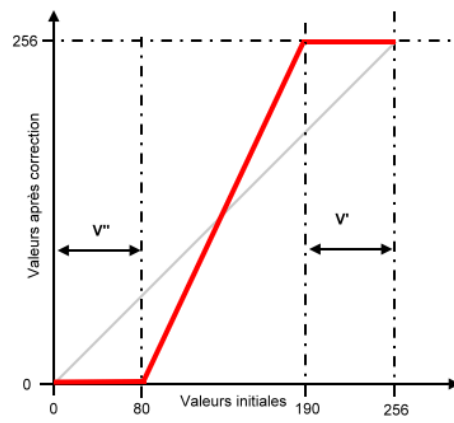
Dans l'exemple ci-contre, par exemple, tous les pixels de valeurs 100 prendront la valeur 200. Ici, l'image sera éclaircie.



^a. Il faudrait, en principe, représenter trois courbes tonales : une pour le rouge, une pour le vert et une pour le bleu.



Augmenter la luminosité

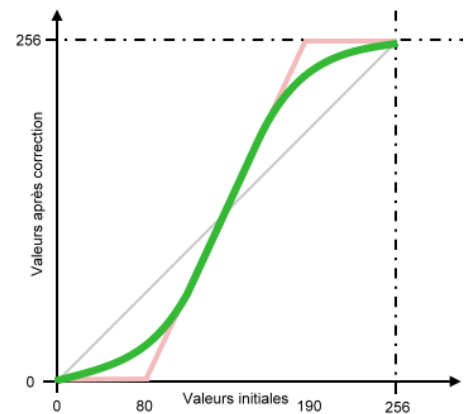


Augmenter le contraste

Le traitement proposé dans les schémas précédents est un peu « brutal » ; il y a perte d'information. Dans l'exemple de droite : tous les points ayant une valeur inférieure à 80 seront noirs et tous ceux ayant une valeur supérieure à 190 seront blancs.

Dans la « vraie vie » il est préférable d'utiliser une courbe lissée dans laquelle la courbe tonale rejoint les axes tangentiellement (comme dans l'exemple ci-contre).

Il nous faut alors connaître l'expression analytique de la fonction permettant cette transformation.



2

A vous de jouer...

2.1

Création d'images

a

Quelques routines utilitaires...

La trame du fichier à exploiter cette semaine contient l'importation des bibliothèques nécessaires ainsi que la définition des variables correspondant aux pixels *blanc*, *noir*, *gris*, *rouge* et *bleu*.

```
1 import matplotlib.pyplot as plt
2 import matplotlib.image as img
3
4 blanc=[1.0,1.0,1.0]
5 noir=[0.0,0.0,0.0]
6 gris=[0.5,0.5,0.5]
7 rouge=[1.0,0.0,0.0]
8 bleu=[0.0,0.0,1.0]
```

1. Ecrire le code d'une fonction *largeur(image)* retournant la largeur de l'image (liste de listes) passée en paramètre.
2. Ecrire le code d'une fonction *hauteur(image)* retournant la hauteur de l'image (liste de listes) passée en paramètre.
3. Ecrire le code d'une fonction *fond(nl,nc,color)* qui à partir de la couleur d'un pixel passé en paramètre crée une image de cette couleur comptant *nl* lignes et *nc* colonnes et retourne cette image.

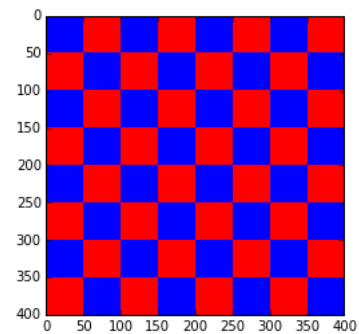
4. Ecrire le code d'une fonction `insert(fond, image, pos_x, pos_y)` qui permet d'insérer l'image `image` dans le fond `fond` aux positions spécifiées (l'origine du repère étant en haut à gauche, l'axe x orienté vers la droite et l'axe y orienté vers le bas). La variable `image` n'est pas modifiée. Le fond, quant à lui, est modifié. Si l'image insérée dépasse du fond, elle doit être tronquée.
5. Tester votre code sur quelques exemples de votre choix.

b**Séquence patriotique**

1. Compléter le programme afin de tracer un drapeau français.
2. Compléter le programme afin de tracer un drapeau japonais. Le rapport entre la hauteur et la largeur du drapeau est de $2/3$, et le diamètre du disque rouge est trois cinquièmes de la hauteur du drapeau.

c**Un damier pour jouer aux échecs**

1. Ecrire le code d'une fonction `damier(n,p,color1,color2)` permet d'afficher un damier de n cases horizontales et n cases verticales alternativement de couleurs `color1` et `color2`; chaque case étant représentée par un carré de $p \times p$ pixels.
2. Ecrire le code permettant de stocker le damier représenté ci-contre dans la variable `echec`, chaque case de l'échiquier étant un carré de 25 pixels de côté.
3. Importer l'image du fichier `cavalier.png` (l'image doit être au même niveau que votre programme sur le disque dur pour éviter à gérer les chemins d'accès). Une conversion de format est nécessaire ici (dans le fichier d'origine, on a un format `rgba`) : il faut écrire le code suivant :



```

1  cavalier=img.imread("cavalier.png","png")
2
3  def rgba2rgb(image):
4      nl = len(image)
5      nc = len(image[0])
6      rgb = fond(nl,nc,blanc)
7      for l in range(nl):
8          for c in range(nc):
9              pixel = image[l][c]
10             rgb[l][c]= [pixel[0],pixel[1],pixel[2]]
11     return rgb
12
13  cavalier2 = rgba2rgb(cavalier)

```

Vérifier que la taille de l'image est bien 50×50 et afficher l'image.

4. A l'aide de l'instruction `randint(0,7)` de la bibliothèque `random`, tirer deux nombres entiers au hasard entre 0 et 7 (inclus) et placer le cavalier à cette position sur l'échiquier.
5. Pour experts... un peu de transparence

On aimerait bien pouvoir voir la couleur de la case sur laquelle est posé l'image du cavalier à l'extérieur de la forme (fermée) qui représente le cavalier.

Pour cela, au lieu de décrire chaque pixel par 3 valeurs $[r, g, b]$ on ajoute un quatrième paramètre a qui gère la transparence ($a = 1$ pour une opacité maximale; $a = 0$ pour une transparence totale). On a alors un codage RGBA de la couleur par une liste $[r, g, b, a]$.

Modifier les différentes routines précédentes afin d'afficher une image du cavalier comme ci-contre.



6. Et, pourquoi pas maintenant... colorier en gris toutes les cases accessibles par le cavalier !

2.2

Coucou, Lenna



Cette image, libre de droits, sert d'image de test pour les algorithmes de traitement d'image et est devenue de facto un standard industriel et scientifique. Wikipedia vous précisera son origine.

Le code proposé ci-dessous charge l'image dans la variable *lenna* et affiche l'image. Ensuite, on appelle une routine qui retourne l'image verticalement (cet exemple peut servir de modèle pour la plupart des exercices suivants).

```
1 lenna = img.imread("lenna.png")
2 plt.imshow(lenna)
3 plt.show()
```

a

Retourner l'image

1. Écrire une routine qui retourne l'image horizontalement.
2. Écrire une routine qui retourne l'image verticalement.

b

Niveaux de gris

1. Créer une routine *niveauGris(image)* qui, à partir d'une image passée en paramètre, retourne l'image en niveaux de gris. Pour gagner du temps sur la suite, on codera le niveau de gris sur un seul réel compris entre 0 et 1.
2. Après avoir appelé cette routine et stocké le résultat dans la variable *lennaGris*, afficher cette image (`plt.imshow(lennaGris, cmap='gray')`).

c

Retouche d'image

Pour simplifier, on travaillera, ici, sur l'image en niveau de gris *lennaGris*.

1. Proposer une routine *eclaircir* (acceptant éventuellement un paramètre) qui traite une image en niveau de gris afin de l'éclaircir. Tester sur l'image *lennaGris*
2. On pourrait aussi :
 - a. obtenir le négatif de l'image couleur (ou niveaux de gris) en remplaçant toutes les valeurs x de tous les pixels par $1 - x$.
 - b. effectuer un seuillage de l'image (niveaux de gris ou couleur) : on remplace toutes les valeurs x de tous les pixels par 1 si $x < seuil$ et par 0 si $x > seuil$
 - c. ... on peut imaginer aussi son propre traitement d'image : pourquoi pas ceux correspondants aux courbes tonales proposées.

d Application d'un filtre

Un traitement « classique » d'images (que l'on retrouve dans la plupart de logiciels) consiste à appliquer un filtre. Par exemple, à l'aide de Photoshop (ou gimp), pour flouter quelques défauts sur votre selphie !

On effectue alors ce qu'on appelle une convolution.

$$\text{Prenons, par exemple, l'exemple d'un filtre } 3 \times 3 : F = \begin{pmatrix} F_{0,0} & F_{1,0} & F_{2,0} \\ F_{0,1} & F_{1,1} & F_{2,1} \\ F_{0,2} & F_{1,2} & F_{2,2} \end{pmatrix}$$

L'image filtrée J s'obtient en effectuant le produit de convolution entre l'image I et le filtre F . Le pixel de coordonnées (i,j) dans l'image filtrée est obtenu en faisant la moyenne pondérée (par les coefficients du filtre F) du pixel $I_{i,j}$ de l'image initiale et de ses 8 proches voisins.

$$\begin{aligned} J_{i,j} = & F_{0,0}I_{i-1,j-1} + F_{1,0}I_{i,j-1} + F_{2,0}I_{i+1,j-1} \\ & F_{0,1}I_{i-1,j} + F_{1,1}I_{i,j} + F_{2,1}I_{i+1,j} \\ & F_{0,2}I_{i-1,j+1} + F_{1,2}I_{i,j+1} + F_{2,2}I_{i+1,j+1} \end{aligned}$$

On s'assure toutefois que $J_{i,j}$ est compris entre 0 et 1 : si $J_{i,j} < 0$, alors $J_{i,j} = 0$ et si $J_{i,j} > 1$ alors $J_{i,j} = 1$.

On propose quelques filtres « classiques » : lissage, accentuation, embossage, convolution (dans l'ordre).

$$\begin{pmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{pmatrix}; \begin{pmatrix} 0 & -0.5 & 0 \\ -0.5 & 3 & -0.5 \\ 0 & -0.5 & 0 \end{pmatrix}; \begin{pmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{pmatrix}; \begin{pmatrix} -\frac{1}{6} & -\frac{2}{3} & -\frac{1}{6} \\ \frac{2}{3} & \frac{13}{3} & \frac{2}{3} \\ -\frac{1}{6} & -\frac{2}{3} & -\frac{1}{6} \end{pmatrix}$$

1. Ecrire le code d'une routine *convolution(image, filtre)* qui applique à l'image *image* le filtre *filtre* constitué d'une matrice 3×3 et qui retourne l'image filtrée. On traite une image en niveaux de gris avec le gris codé sur 1 réel.
2. Tester le résultat sur *lennaGris* avec l'un des filtres proposés.
3. On peut aussi proposer son propre filtre (il faut juste s'assurer que la somme des termes vaille soit 0 soit 1).

2.3

S'il reste du temps... stéganographie : une image dans une image

Contrairement à la cryptographie, qui chiffre des messages de manière à les rendre incompréhensibles, la stéganographie cache les messages dans un support, par exemple des images ou un texte qui semble anodin.

On peut cacher un texte dans une image numérique et cela de manière parfaitement invisible à l'œil nu. Cette technique s'appelle le tatouage (watermarking en anglais). Elle est utilisée notamment pour protéger des images par copyright, mais on peut aussi transmettre des messages cachés. Nous proposons ici une méthode simple, mais qui fonctionne seulement avec certains formats d'images. En effet, beaucoup de formats compresse les données et donc modifient les bits de l'image, ce qui a pour effet de détruire le message caché.

On propose de traiter chaque composante x de chaque pixel de la façon suivante :

- on convertit x en nombre entier n compris entre 0 et 255 (dans l'image donnée, x varie entre 0 et 1);
- on récupère les 4 bits faibles de n et on les décale de 4 bits vers la gauche.

	bits de poids fort				bits de poids faible			
$n =$	1	0	0	1	0	1	1	0
reste de la division par 16 : $n \% 16 =$	0	0	0	0	0	1	1	0
que l'on multiplie par 16 : $16 \times (n \% 16) =$	0	1	1	0	0	0	0	0

- on divise la valeur obtenue par 255 pour la convertir en réel compris entre 0 et 1.
1. Faire cette conversion pixel par pixel sur l'image mystere.png ;
 2. Découvrir l'image cachée (le calcul est un peu long) !

Activité 8

A la recherche du zéro

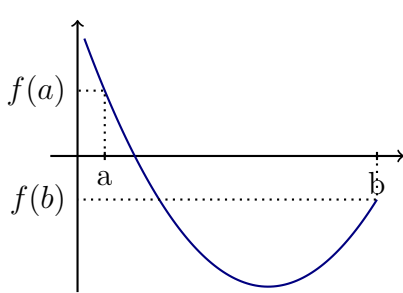
1

Méthodes

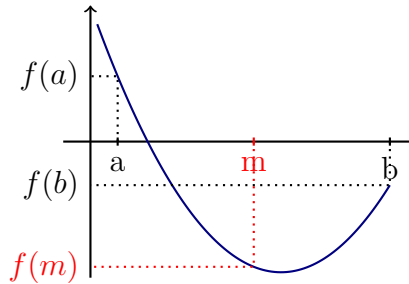
1.1

Méthode de dichotomie

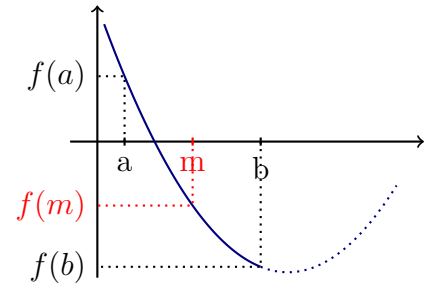
C'est une méthode relativement simple et efficace pour trouver l'unique zéro d'une fonction $f(x)$ sur un intervalle $[a, b]$ donné avec une précision eps souhaitée.



La fonction ne s'annulant qu'une seule fois sur l'intervalle d'étude, on doit avoir $f(a) \times f(b) < 0$. Considérons le cas de figure ci-contre.



On prend m , milieu de $[a, b]$ et on calcule $f(m)$



Ici la fonction s'annule entre a et m .

Il suffit donc de réitérer le processus en prenant comme nouvelle valeur de b la valeur de m .

On continue ainsi de suite tant que $b - a > eps$.

Un programme implémentant la méthode de dichotomie a, typiquement, la structure suivante :

```
1 def fTest(x):
2     return x*x - 2
3
4 def dichotomie(f, a, b, eps) :
5     while b - a > eps :
6         m = (a+b)/2 # milieu de [a,b]
7         if f(a)*f(m) <= 0 :
8             b = m
9         else:
10            a = m
11     return m
12
13 solution = dichotomie(fTest, 0, 4, 0.001)
14 print(solution)
```

1.2 Méthode de Newton

Je vous la présente quand même...ceci étant, dans l'étude de systèmes chimiques, vous utiliserez exclusivement la méthode de dichotomie¹.

On cherche à trouver une bonne approximation du zéro d'une fonction d'une variable réelle $f(x)$ en considérant son développement de TAYLOR au premier ordre. Pour cela, partant d'un point x_0 , on approche la fonction au premier ordre : $f(x) \approx f(x_0) + f'(x_0)(x - x_0)$.

Pour trouver un zéro de cette fonction d'approximation, il suffit de calculer l'intersection de la droite tangente avec l'axe des abscisses.

Il suffit donc de résoudre l'équation :

$$0 = f(x_0) + f'(x_0) \times (x - x_0).$$

On trouve ainsi un point (si la dérivée en

x_0 n'est pas nulle!) $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$. Cette

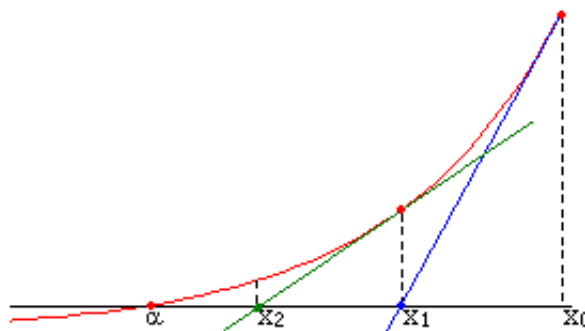
relation traduit la définition même de la dérivée :

$$f'(x_0) = \frac{f(x_0) - 0}{x_0 - x_1}.$$

On réitère le processus afin de construire, par

réurrence, une suite : $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$.

On arrête lorsque la valeur absolue de la différence entre x_{k+1} et x_k est inférieure à la précision souhaitée sur la détermination du zéro.



Un programme implémentant la méthode de NEWTON a, typiquement, la structure suivante :

```

1 def fTest(x):
2     return x*2 - 2
3
4 def deriveTest(x):
5     return 2*x
6
7 def newton(f,df,x0,deltaX):
8     x1=x0-f(x0)/df(x0)
9     while abs(x1-x0)>deltaX:
10         x0=x1
11         x1=x0-f(x0)/df(x0)
12     return x1
13
14 solution = newton(fTest, deriveTest, 2, 1e-8)
15 print (solution)

```

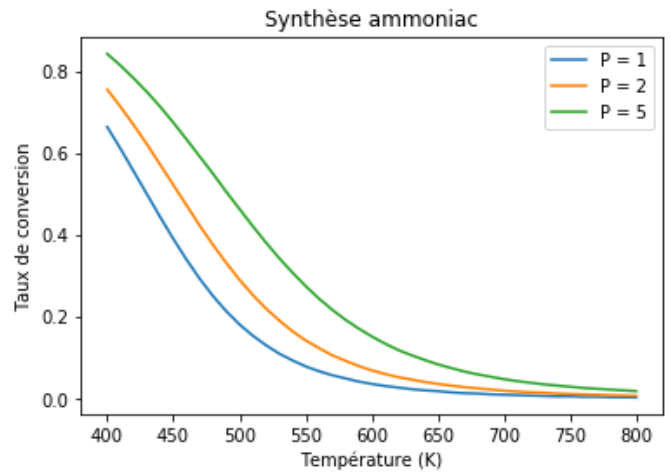
2 Trois challenges de chimistes...

2.1 Taux de conversion en fonction de la température

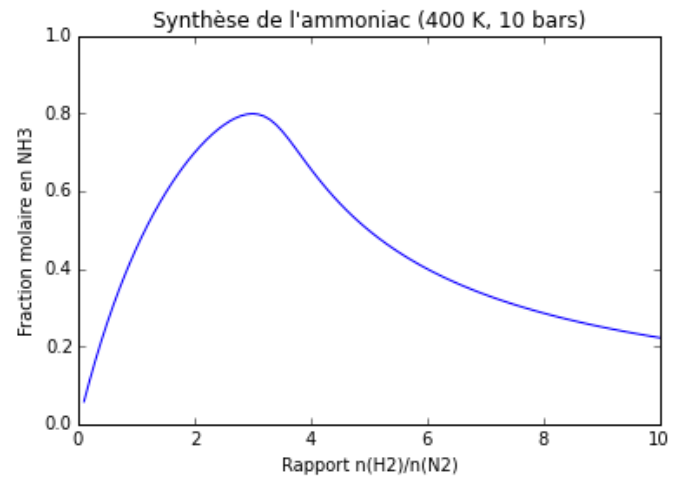
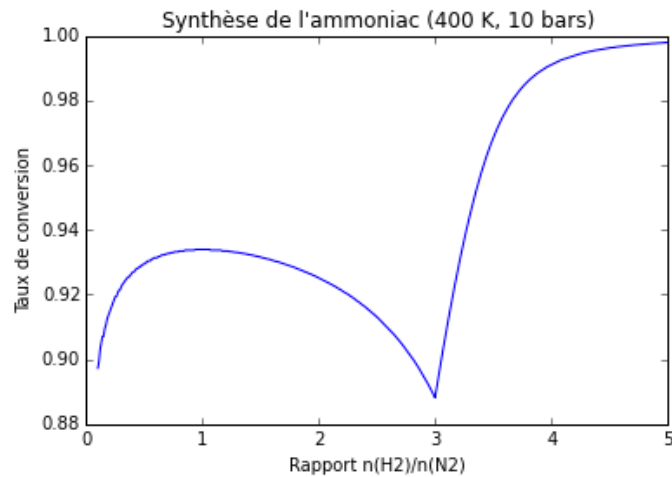
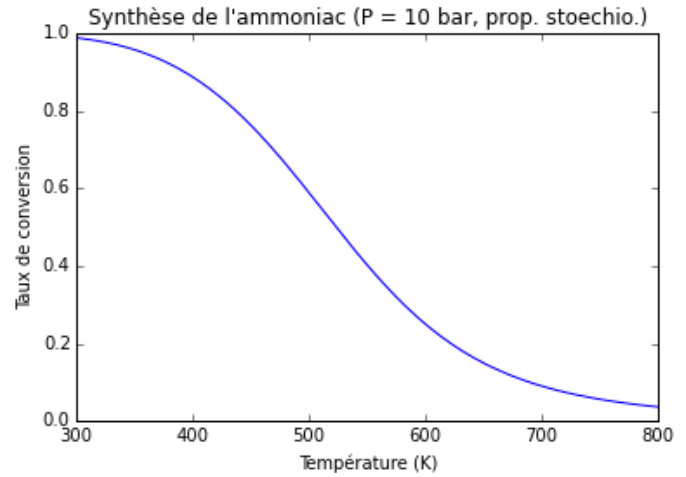
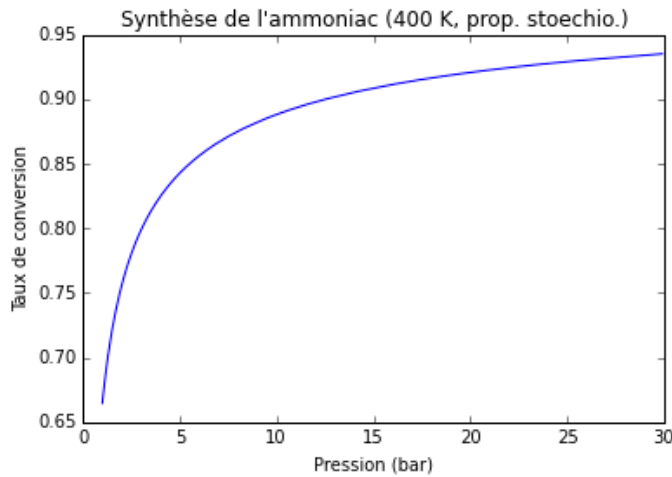
On considère l'équilibre de synthèse de l'ammoniac pour lequel $\Delta_r H^0 = -92 \text{ kJ.mol}^{-1}$ et $\Delta_r S^0 = -200 \text{ J.K}^{-1}.\text{mol}^{-1}$.

1. Pour des applications « chimiques », je n'ai vu d'utilisation de la méthode de NEWTON que pour l'étude du zéro d'une fonction de plusieurs variables. Pour le coup, la méthode de dichotomie ne peut pas être exploitée efficacement

On part des proportions stoechiométriques.
 On étudie le taux de conversion en fonction pour
 différentes valeurs de la pression totale.
 On désire obtenir la courbe ci-contre.
 A vous de jouer !



Après on peut tout faire pour son cours sur les déplacements d'équilibre !



2.2

A la recherche de l'hétéroazéotrope

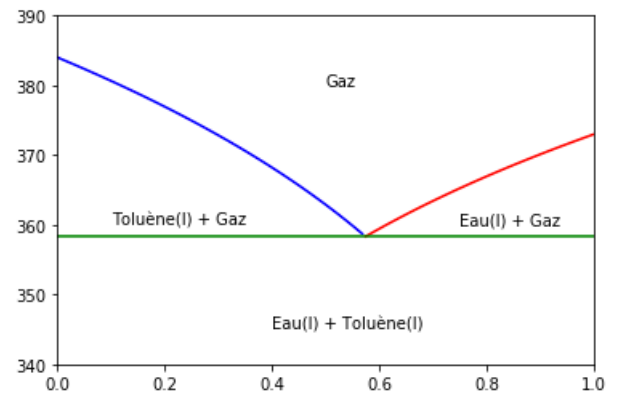
Soit par exemple le mélange binaire eau-toluène.

On donne l'enthalpie standard de vaporisation et la température d'ébullition :

- pour le toluène : $LA = 38 \text{ kJ} \cdot \text{mol}^{-1}$, $TA = 384 \text{ K}$
- pour l'eau : $LB = 42 \text{ kJ} \cdot \text{mol}^{-1}$, $TB = 373 \text{ K}$

L'objectif est de tracer le diagramme binaire toluène-eau.

L'instruction `plt.annotate('Gaz', (0.5, 380))` permet d'écrire le libellé souhaité aux coordonnées souhaitées.



2.3

Courbe de titrage

Ecrire un programme simulant le titrage d'un monoacide faible par une base forte.

Pour experts... en exploitant les résultats de l'activité 5 sur les courbes de répartition, on peut envisager le titrage d'un polyacide quelconque ; voire même le titrage d'un mélange d'acides !

Si on arrive là, on peut « sans trop » de difficultés envisager le titrage de n'importe quoi par n'importe quoi pour peu qu'une seule particule soit échangée. Chiche !

3 Quelques pistes

3.1 Synthèse de l'ammoniac

Si on ne souhaite tracer que le premier graphe, on peut faire un peu ce qu'on veut. En anticipant sur le suivant et un peu logiquement aussi... () il faudrait :

- définir R , $DrH0$, $DrS0$ comme variables globales ;
- coder une fonction $K0(T)$;
- coder une fonction $Qr(P, ksi, n0, n1, n2)$ (où $n0$, $n1$ et $n2$ sont les quantités initiales en N_2 , H_2 et NH_3) ;
- coder une fonction $affinite(T, P, ksi, n0, n1, n2)$

3.2 A la recherche de l'hétéroazéotrope

Après avoir défini une fonction $osee(x, Teb, Lvap)$, pour trouver l'intersection des courbes de rosée, toute l'astuce consiste à définir une fonction auxiliaire retournant la différence entre la courbe de rosée relative à l'eau et celle relative au toluène.

3.3 Courbe de titrage

L'idée est de résoudre l'équation d'électroneutralité par dichotomie.

- on écrit l'équation d'électroneutralité ;
- on écrit l'équation de conservation sur le couple acide/base étudié ce qui permet d'obtenir une équation dont l'unique inconnue est h ;
- on écrit une fonction de dichotomie (celle-ci doit travailler sur le pH et non sur h !) ;
- on boucle sur les différentes valeurs du volume.

4 Solutions

4.1 Synthèse de l'ammoniac

```

1  """
2  dichotomie
3  """
4  def dichotomie(f, xmin, xmax, delta):
5      while xmax - xmin > delta:
6          if f(xmin)*f((xmin+xmax)/2) < 0:
7              xmax = (xmin+xmax)/2
8          else:
9              xmin = (xmin+xmax)/2
10     return (xmin+xmax)/2
11
12     """
13     diagramme binaire
14     """
15     R = 8.314
16     TA = 384      #toluène
17     LA = 38000
18     TB = 373      #eau
19     LB = 42000

```



```

20
21 def rosee(x,Teb, Lvap):
22     return Teb/(1 - (R*Teb*log(x)/Lvap))
23
24 def fonction(x):
25     return rosee(x, TB, LB)-rosee(1-x, TA, LA) # attention au
        1-x !!!
26
27 XH = dichotomie(fonction, 0.1, 0.9, 0.001)
28 TH = rosee(XH,TB,LB)
29
30 print('Hétéroazéotrope : xH =', XH, ' TH = ', TH)
31
32 Xeau = [XH + i *(1 - XH)/100 for i in range(101) ]
33 Roseau = [rosee(x,TB,LB) for x in Xeau ]
34 Xtol = [i *XH/100 for i in range(100) ]
35 Rostol = [rosee(1-x,TA,LA) for x in Xtol ]
36
37 plt.plot(Xeau, Roseau, 'r')
38 plt.plot(Xtol, Rostol, 'b')
39 plt.plot([0,1], [TH,TH], 'g')
40 plt.axis([0, 1, 340, 390])
41 plt.annotate('Gaz', (0.5, 380) )
42 plt.annotate('Eau(l) + Toluène(l)', (0.4, 345) )
43 plt.annotate('Eau(l) + Gaz', (0.75, 360) )
44 plt.annotate('Toluène(l) + Gaz', (0.1, 360) )
45 plt.show()

```

4.2

A la recherche de l'hétéroazéotrope

```

1  #variables globales utilisées dans l'ensemble du programme
2  R = 8.314
3  n1 = 1.0 # quantité initiale en N2
4  n2 = 3.0 # quantité initiale en H2
5  n3 = 0.0 # quantité initiale en NH3
6  Tmin = 400.0 #température minimale de l'étude
7  Tmax = 800.0 #température maximale de l'étude
8  deltaT = 10.0 #intervalle de température entre deux calculs
9
10 DrH0 = - 92000 # enthalpie standard de réaction en J/mol
11 DrS0 = - 200 # entropie standard de réaction en J/K/mol
12
13 def A(ksi, T, P):#calcul de l'affinité chimique de la réaction
        en fonction de l'avancement, de T, de P
14     return -(DrH0 - T*DrS0)
        -R*T*log(pow(n3+2*ksi,2)*pow(n1+n2+n3-2.0*ksi,2.0)/((n1-ksi)*pow(n2
15
16 def ksi_eq(mini, maxi, precision, T, P):#calcul de
        l'avancement à l'équilibre entre mini et maxi avec la
        précision souhaitée
17     while maxi - mini > precision:
18         milieu = (maxi + mini)/2
19         if A(mini, T, P)*A(milieu, T, P) < 0 :

```

```

20         maxi = milieu
21     else:
22         mini = milieu
23     return (maxi + mini)/2
24
25 def plot_to(P):#abscisses et ordonnées d'une courbe taux de
conversion = f(T) pour P donnée
26     X=[]
27     Y=[]
28     T= Tmin
29     while T <= Tmax :
30         to = ksi_eq(0.00001, min(n1, n2/3), 0.001, T, P)
31         X.append(T)
32         Y.append(to)
33         T = T + deltaT
34     return X, Y
35
36
37 for P in [1, 2, 5]:# on trace la courbe pour les différentes
valeurs de la pression (en bar)
38     X, Y = plot_to(P)# on récupère abscisses et ordonnées
39     plt.plot(X, Y, label = "P = "+str(P))# et on les traces
40 plt.xlabel('Température (K)') ;
41 plt.ylabel('Taux de conversion') ;
42 plt.legend()
43 plt.title('Synthèse ammoniac') ;
44 plt.show()

```

4.3 Courbe de titrage

```

1 Ke = pow(10,-14)
2
3 """ routines utilitaires pour calculer le pH d'un mélange """
4 def f(h,C0,V0,Ct,Vt,Ka): # équation à annuler pour déterminer
le pH par dichotomie
5     return (h - Ke/h)*(V0 + Vt) + Ct*Vt - C0*V0/(1 + h/Ka)
6
7 def pH(C0,V0,Ct,Vt,Ka): # méthode de dichotomie pour calculer
le pH
8     a=0
9     b=14
10    fa = f(pow(10,-a), C0, V0, Ct, Vt, Ka)
11    while b - a > 0.01 :
12        c = (a+b)/2
13        fc = f(pow(10,-c), C0, V0, Ct, Vt, Ka)
14        if (fa *fc < 0):
15            b = c
16        else:
17            a = c
18            fa = fc
19    return (a+b)/2
20
21 """ programme principal """

```

```
22 V = [] # tableau de valeur pour stocker les volumes
23 PH = [] # tableau de valeur pour stocker les pH correspondants
24 v = 0 # volume "courant"
25 Ca = 0.01 #concentration en acide titré
26 Va = 100 #volume titré
27 Cb = 0.1 #concentration en base
28 pas = 0.01 #pas d'incrémentement du volume
29 Vmax = 25 # volume maximal de réactif titrant
30
31 while v <= Vmax :
32     V.append(v)
33     ppH = pH(0.01, 100, 0.1, v, pow(10,-4))
34     PH.append(ppH)
35     v = v + pas
36
37 plt.plot(V, PH)
38 plt.show()
```

Activité 9

Intégration numérique

1 Méthode des rectangles et des trapèzes

1. Définir une fonction de votre choix (dont vous pouvez connaître la primitive!).
2. Écrire le code des deux fonctions *rectangle*(*f*, *debut*, *fin*, *pas*) et *trapeze*(*f*, *debut*, *fin*, *pas*) calculant l'aire sous la courbe *f* entre *a* et *b* avec un incrément de *pas*, en utilisant, respectivement, la méthode des rectangles et celle des trapèzes.
3. Vérifier la validité de vos méthodes à l'aide de la fonction choisie.

2 Spectre RMN ou de RMN :) et sa courbe d'intégration...

Un format plus ou moins standardisé existe pour sauvegarder les enregistrements de spectres (IR ou RMN). A partir d'un fichier correspondant à une molécule, un autre fichier texte, plus facile à interpréter pour vous a été généré.

Le fichier *RMN.txt* compte environ 1400 lignes ; chacune d'elle contient, sous forme de chaînes de caractères séparées par un ; le déplacement chimique et l'intensité du signal. Le fichier commence ainsi :

```
0.0104518209488068;0.0060934236561018
0.0135146470021783;0.00519322683651741
0.0165774730555498;0.00608414558802262
```

Le fichier *RMN.py* contient la trame de résolution de l'exercice. On s'assura d'avoir le fichier *RMN.txt* dans le même dossier que le fichier *RMN.py*

Les premières instructions :

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

permettent d'ouvrir le fichier et de générer la variable *lignes* qui contient un tableau de chaînes de caractère (le premier élément du tableau est "0.121249699346168;0.00290495778376928").

Supposons que l'on récupère dans une variable *l* la chaîne de caractère "0.121249699346168;0.00290495778376928".

L'instruction *t = l.split(";")* permet de récupérer le tableau de chaîne de caractères : ["0.121249699346168", "0.00290495778376928"].

1. Compléter le fichier *RMN.py* afin de construire deux tableaux(listes) *delta* et *intensite* contenant, respectivement, les différentes valeurs de l'opposé du déplacement chimique et de l'intensité du signal.

2. Tracer la courbe correspondante et s'assurer que l'on a bien l'allure d'un spectre RMN ! L'instruction `plt.xlim(5, 0)` permettra d'orienter l'axe des déplacements chimiques dans le bon sens.
3. Compléter le programme afin de construire deux tableaux *deltaIntegre* et *integre* permettant de tracer la courbe d'intégration du signal.
4. Supprimer les commentaires en fin de fichier afin d'exploiter quelques possibilités de la bibliothèque matplotlib permettant la superposition de courbes avec deux échelles verticales différentes.

3**Cadeau de Noël**

Bon, vous avez bien travaillé ! Vous avez droit à un cadeau...

Le format plus ou moins standardisé évoqué est le format jdx. Vous pouvez trouver sur internet de nombreux spectres à ce format. Dans le logiciel « ChimPack » que j'avais commencé à développer, j'avais utilisé une collection de spectres expérimentaux. Le logiciel permettait ensuite d'afficher ou non la courbe d'intégration et de zoomer sur certains massifs.

Vous trouverez dans les fichiers compactés une soixantaine de spectres RMN et autant d'IR. Vous trouverez, par ailleurs, un fichier `jcamp.py` qui permet d'extraire les données et deux fichiers d'exemples `jcamp_rmnm.py` et `jcamp_ir.py` qui permet de tracer les spectres. Il y a peut-être quelques fichiers qui ne sont pas au format « officiel » et que l'on ne peut pas exploiter ; je n'ai pas eu le temps de tous les tester !

Activité 10

Il est né le divin Euler

J'initialise, je calcule les incréments, j'incrémente, je stocke !

1

Résolution numérique d'une équation différentielle, ou d'un système !

1.1

Méthode d'Euler explicite

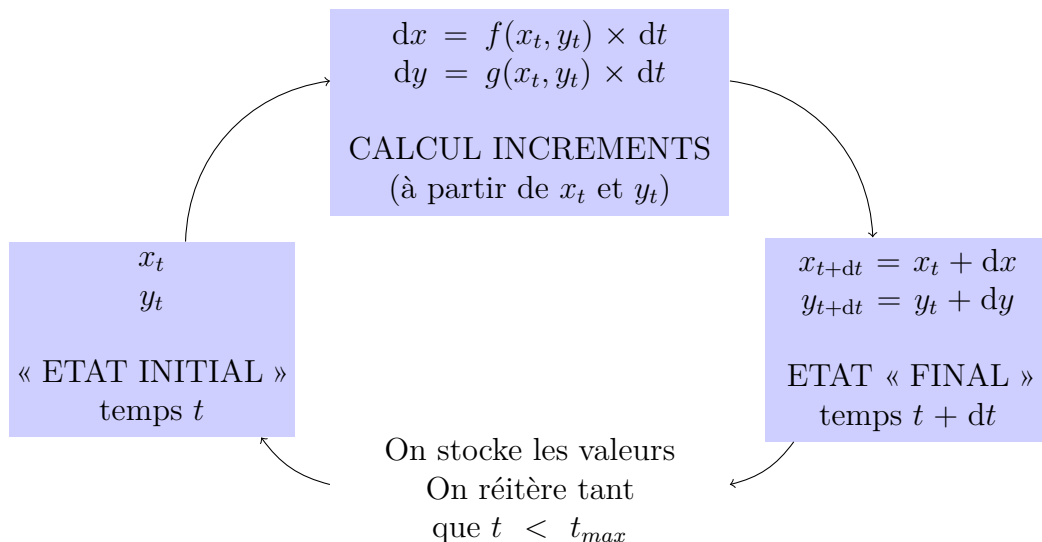
C'est la seule méthode explicitement au programme.

a

Rappel du principe de la méthode

On peut relativement simplement implémenter la méthode d'EULER en langage python. L'objectif est de déterminer l'évolution temporelle d'un système à partir de conditions initiales pour peu que l'on connaisse les conditions d'évolution de ce système.

Le principe consiste à appliquer un incrément à un jeu de variables :



La principale difficulté réside dans le choix de l'incrément dt de la méthode d'EULER.

- s'il est trop grand, les variations de N sont trop importantes et N peut avoir des valeurs qui deviennent négatives, puis positives, puis négatives... ;
- s'il est trop petit, les calculs peuvent devenir longs (même si les machines « modernes » sont performantes) et des erreurs d'arrondis de calculs peuvent s'accumuler.

On parle, ici, de méthode d'EULER explicite car les incréments pour passer de t à $t + dt$ sont calculés au temps t .

Dans une présentation plus « mathématicienne » des choses, on noterait h le pas d'incrément (équivalent du dt) et on construirait deux suites x_n et y_n telles que :

$$\begin{cases} x_{n+1} = x_n + f(x_n, y_n) \times h \\ y_{n+1} = y_n + g(x_n, y_n) \times h \end{cases}.$$

Nos élèves doivent savoir programmer la méthode d'EULER explicite pour une équation différentielle du type : $\frac{dx}{dt} = f(x, t)$.

b Implémentation

A titre personnel, quitte à écrire quelques lignes de code en plus, je conseille d'utiliser une variable « courante » en plus des variables de stockage.

Voici, sur l'exemple de la cinétique d'une réaction successive, comment je code la méthode d'EULER

```

1  import matplotlib.pyplot as plt
2
3  def euler(C0,k1,k2,tmax, dt):
4      # j'initialise
5      a = C0 # les variables courantes
6      b = 0
7      c = 0
8      t = 0
9      A = [a] # les tableaux de stockage
10     B = [b]
11     C = [c]
12     T = [t]
13     while t <= tmax : # la boucle d'itération
14         # je calcule les incréments
15         da = -k1*a*dt
16         db = (k1*a-k2*b)*dt
17         dc = k2*b*dt
18         # j'incrémente
19         a = a + da
20         b = b + db
21         c = c + dc
22         t = t + dt
23         #je stocke
24         A.append(a)
25         B.append(b)
26         C.append(c)
27         T.append(t)
28     #on retourne T, A, B, C
29     return T, A, B, C
30
31 T, A, B, C = euler(1,1,1,5,0.01)
32
33 plt.plot(T,A,'r')
34 plt.plot(T,B,'b')
35 plt.plot(T,C,'g')
36 plt.show()

```

L'intérêt de l'utilisation des variables courantes est d'éviter une erreur de méthode en codant, dans la boucle while quelque chose comme :

```

1   a = a - k1*a*dt
2   b = b + (k1*a - k2*b)*dt
3   c = c + k2*b*dt

```

Ici, la nouvelle valeur de a est calculée à partir de la valeur précédente, par contre b est calculé avec la nouvelle valeur de a et l'ancienne valeur de b . . . ce n'est pas la mort du petit cheval, mais ce n'est pas vraiment une méthode d'EULER explicite.

Pour les gens à l'aise, on peut se contenter uniquement des tableaux. . . voire même de tableaux avec des indices négatifs, ce qui permet de diminuer le nombre de lignes de code.

c

Pour les physiciens. . .

En physique, on est souvent amené à résoudre une équation différentielle du second ordre.

Supposons un système défini par sa position x , sa vitesse $v = \frac{dx}{dt}$, son accélération $a = \frac{dv}{dt} = f(x, v, t)$.

L'idée est de remplacer l'équation du second ordre : $a = \frac{d^2x}{dt^2} = f(x, v, t)$ par un système de deux

$$\text{équations : } \begin{cases} v = \frac{dx}{dt} \\ a = f(x, v, t) = \frac{dv}{dt} \end{cases} .$$

Il « suffira » donc, dans l'algorithme d'exprimer les incréments sous la forme : $\begin{cases} dx = v \times dt \\ dv = f(x, v, t) \times dt \end{cases}$

1.2

Quelques remèdes. . .

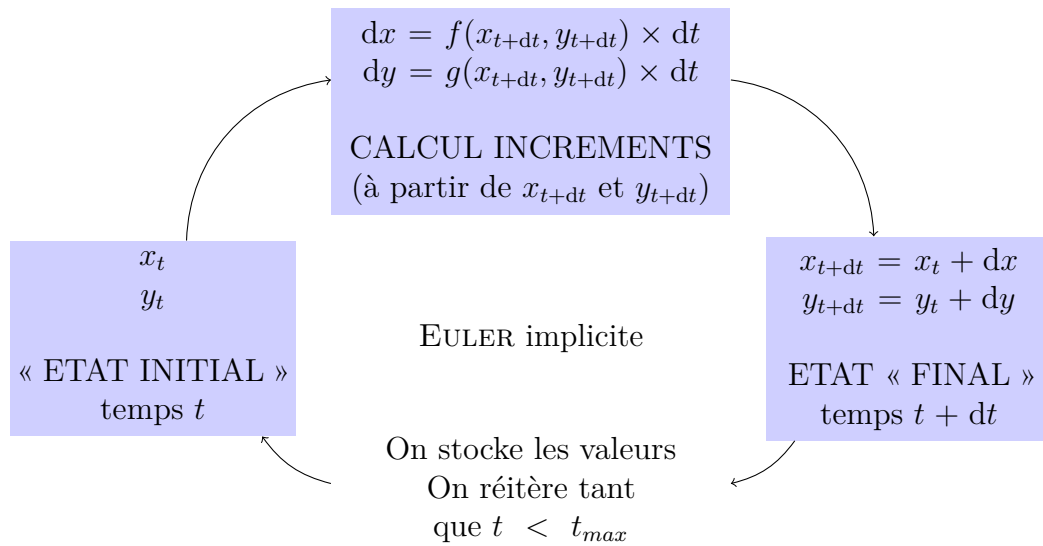
Parfois la méthode d'EULER explicite diverge (c'est le cas, par exemple de lors de l'étude d'un oscillateur harmonique). On peut alors proposer quelques méthodes alternatives relativement simples à mettre en oeuvre.

a

Méthode d'Euler implicite

L'idée est de calculer les incréments, non pas à partir des grandeurs au temps t mais à partir des grandeurs au temps $t + dt$. . . donc, *a priori*, à partir de grandeurs que l'on ne connaît pas !

On peut alors modifier le schéma de la façon suivante :



Une autre présentation consisterait à écrire :
$$\begin{cases} x_{n+1} = x_n + f(x_{n+1}, y_{n+1}) \times h \\ y_{n+1} = y_n + g(x_{n+1}, y_{n+1}) \times h \end{cases}.$$

b Méthode de Runge-Kutta

Une alternative à la méthode d'EULER est la méthode de RUNGE-KUTTA. La méthode la plus « classique » est la méthode d'ordre 4 dite *RK4*.

Pour une équation différentielle du première ordre $\frac{dx}{dt} = f(x, t)$ avec $x(t_0) = x_0$ on a, pour un pas d'intégration h :

$$x_{n+1} = x_n + \frac{h}{6} (k_1 + 2k_2 + 2k_3 + k_4) \text{ avec : } \begin{cases} k_1 = f(x_n, t_n) \\ k_2 = f\left(x_n + \frac{h}{2} \times k_1, t_n + \frac{h}{2}\right) \\ k_3 = f\left(x_n + \frac{h}{2} \times k_2, t_n + \frac{h}{2}\right) \\ k_4 = f(x_n + h \times k_3, t_n + h) \end{cases}$$

2 Exemples d'application

2.1 L'oscillateur harmonique

On cherche, ici, à résoudre l'équation différentielle : $\frac{d^2x}{dt^2} + \omega^2 x = 0$. Comme précédemment, on se

ramène à un système de deux équations :
$$\begin{cases} \frac{dx}{dt} = v \\ \frac{dv}{dt} = -\omega^2 x \end{cases}.$$

a Méthode d'Euler

1. Ecrire une routine permettant d'intégrer ce système d'équation différentielle à l'aide de la méthode d'EULER. On attend en retour 3 tableaux de valeur : X, V, T (position, vitesse, temps).

2. Appeler cette fonction avec, par exemple, $\omega^2 = 1$, $x_0 = 1$, $v_0 = 0$, une durée maximale de simulation de 100 et un incrément de 0,1 entre deux calculs.
3. Tracer la courbe $X = f(T)$ et conclure !
4. Tracer également le portrait de phase de l'oscillateur.
5. On pourrait se dire que l'on a pris un pas trop grand... tester pour un intervalle de temps de 0,01 ; observe-t-on une amélioration des résultats ?

b Méthode d'Euler implicite

Dans la méthode d'EULER explicite on écrivait : $\begin{cases} x_{t+dt} = x_t + v_t \times dt \\ v_{t+dt} = v_t - \omega^2 \times x_t \times dt \end{cases}$, ou encore $\begin{cases} x_{n+1} = x_n + v_n \times dt \\ v_{n+1} = v_n - \omega^2 \times x_n \times dt \end{cases}$
ou encore, sous forme matricielle : $\begin{pmatrix} x_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & h \\ -\omega^2 h & 1 \end{pmatrix} \times \begin{pmatrix} x_n \\ v_n \end{pmatrix}$

Dans la méthode d'EULER implicite on doit écrire : $\begin{cases} x_{t+dt} = x_t + v_{t+dt} \times dt \\ v_{t+dt} = v_t - \omega^2 \times x_{t+dt} \times dt \end{cases}$; un calcul est donc nécessaire pour exprimer l'incrément.

On trouve : $\begin{cases} x_{t+dt} = \frac{x_t}{1 + \omega^2 dt^2} + \frac{v_t}{1 + \omega^2 dt^2} \times dt \\ v_{t+dt} = \frac{v_t}{1 + \omega^2 dt^2} - \frac{\omega^2 x_t}{1 + \omega^2 dt^2} \times dt \end{cases}$

ou encore, sous forme matricielle : $\begin{pmatrix} x_{n+1} \\ v_{n+1} \end{pmatrix} = \frac{1}{1 + \omega^2 dt^2} \begin{pmatrix} 1 & h \\ -\omega^2 h & 1 \end{pmatrix} \times \begin{pmatrix} x_n \\ v_n \end{pmatrix}$

Reprendre les mêmes questions que précédemment et conclure.

c Méthode de Runge-Kutta

On souhaite écrire la routine résolvant l'équation différentielle de l'oscillateur harmonique par la méthode de RUNGE-KUTTA d'ordre 4.

La suite des valeurs calculés revient donc à : $\begin{cases} t_{n+1} = t_n + h \\ v_{n+1} = v_n + \frac{dt}{6} (a_1 + 2a_2 + 2a_3 + a_4) \\ x_{n+1} = x_n + \frac{dt}{6} (b_1 + 2b_2 + 2b_3 + b_4) \end{cases}$

On montre alors que $\begin{cases} a_1 = -\omega^2 \times x_n \\ b_1 = v_n \\ a_2 = -\omega^2 \times \left(x_n + \frac{b_1}{2}\right) \\ b_2 = v_n + \frac{a_1}{2} \end{cases}$ et $\begin{cases} a_3 = -\omega^2 \times \left(x_n + \frac{b_2}{2}\right) \\ b_3 = v_n + \frac{a_2}{2} \\ a_4 = -\omega^2 \times (x_n + b_3) \\ b_4 = v_n + a_3 \end{cases}$.

Reprendre les mêmes questions que précédemment et conclure.

2.2 Et en chimie...

A vous de jouer, vous en savez plus que moi !