Quelsques activités Python sur les dérivées , vecteurs et suites COURS HATTEMER

1ère Spécialité Mathématiques

**écrire la liste des coefficients directeurs des sécantes pour un pas donné(En priorité les 1ère 5/6)**

Plutôt qu’écrire, on préférera ici produire (renvoyer comme le fait toute fonction) la liste. Le pas est donné directement dans le corps de la fonction, et nommé dx. Cela permet de modifier légèrement le script de calcul de longueur de courbe vu en Seconde (au lieu de l’hypoténuse on calcule la tangente de l’angle) :

1. **def** lcds(f,a,b):
2. dx = 0.000001
3. x = a
4. liste\_coeff = []
5. **while** x<b:
6. liste\_coeff.append((f(x+dx)-f(x))/dx)
7. x += dx
8. **return** liste\_coeff

Que signifient les lettres lcds ? **l**iste des **c**oefficients **d**irecteurs des **s**écantes.

**Détermination d’une valeur approchée de e à l’aide de la suite (1+1/n)n**

Au début du 18e siècle, [Jacques Bernoulli](http://fr.wikipedia.org/wiki/Jacques_Bernoulli) a commencé à s’intéresser à l’application des maths aux finances. Cela l’a amené à se poser cette question : De combien un capital augmente-t-il lorsqu’on lui applique des augmentations de pourcentages infinitésimaux équivalents en tout à 100 pourcents ?

Autrement dit, si on augmente n fois une quantité d’un pourcentage de 100/n, l’augmentation totale revient à une multiplication par e. Or augmenter de 100/n pourcents revient à appliquer un coefficient multiplicateur égal à 1+1/n et dans le langage des suites, le résultat de Jacques Bernoulli revient à dire que la suite explicite de terme général égal à (1+1/n)n tend vers e [[8](http://revue.sesamath.net/spip.php?article1226#nb8)].

La suite étant définie explicitement, une liste en compréhension s’impose :

1. **def** approx\_e(n):
2. **return** [(1+1/k)\*\*k **for** k **in** range(1,n+1)]

**Construction de l’exponentielle par la méthode d’Euler**

Jacques Bernoulli était le père de [Daniel Bernoulli](http://fr.wikipedia.org/wiki/Daniel_Bernoulli). Lorsque celui-ci habitait encore à [Bâle](http://fr.wikipedia.org/wiki/B%C3%A2le), il jouait parfois avec le fils du pasteur, dont Jacques a rapidement remarqué le talent en mathématiques. Cet enfant admiré par toute la famille Bernoulli s’appelait [Leonhard Euler](http://fr.wikipedia.org/wiki/Leonhard_Euler). Calculateur acharné, il a trouvé un moyen de résoudre graphiquement des équations différentielles, en remplaçant le coefficient directeur de la sécante par celui de la tangente. Avec l’équation différentielle y’=y cela donne le remplacement de ex par (ex+dx-ex)/dx avec dx petit. On a alors ex+dx-ex≈ ex×dx soit ex+dx≈ex+ex×dx=ex(1+dx) : Les valeurs prises par l’exponentielle sur une subdivision d’un intervalle du type [1,b] à pas constant sont approchées par les termes d’une suite géométrique de raison 1+dx.

Incidemment ce résultat généralise celui sur le calcul approché de e, puisque cela revient à dire que si le nombre n de pas tend vers l’infini, la quantité (1+x/n)n tend vers ex.

La suite géométrique peut se calculer par une compréhension :

1. **def** exp\_euler(n,x):
2. r = 1+x/n *# raison de la suite*
3. y = 1 *# exp(0)==1*
4. **return** [y\*r\*\*k **for** k **in** range(n+1)]

1ère 3/4/5/6

On peut faire suite au cours de 2nde (vecteurs représentés par des couples de flottants) avec le produit scalaire :

1. **def** produit\_scalaire(u,v):
2. (xu,yu) = u
3. (xv,yv) = v
4. **return** xu\*xv+yu\*yv

Cette fonction (qui associe un nombre à deux vecteurs) peut être utilisée pour calculer le cosinus de l’angle entre deux vecteurs non nuls :

1. **def** cosinus(u,v):
2. **assert** u!=(0,0) **and** v!=(0,0)
3. **return** produit\_scalaire(u,v)/norme(u)/norme(v)
5. **def** angle(u,v):
6. **return** degrees(acos(cosinus(u,v)))

Au passage on voit comment, à partir d’un cosinus, Python peut donner l’angle en degrés correspondant : La fonction *acos* permet de retrouver l’angle en radians, et en la composant par la fonction *degrees*, on convertit cet angle en degrés.

On peut ensuite créer une fonction Python pour tester si deux vecteurs (nuls ou non) sont orthogonaux :

1. **def** sont\_orthogonaux(u,v):
2. **return** produit\_scalaire(u,v)==0

1ère ¾ Pour le moment

Une première question à poser, avant d’étudier les algorithmes sur les suites, est celle de la définition d’une suite : **Qu’est-ce que c’est qu’une suite ?**

Comme le programme insiste sur la notion de fonction, on peut proposer cette définition : Une suite est une fonction définie sur **N**. Plus précisément, si l’ensemble d’arrivée est un ensemble de nombres (typiquement **R**) la suite est dite *numérique*.

Se pose alors le problème de la représentation d’une suite en Python : La machine sur laquelle tourne Python ne disposant que d’une quantité de mémoire finie, on ne peut y caser une infinité de nombres u(0)=u0, u(1)=u1, etc.

Le langage Haskell gère les suites qu’il considère comme des listes infinies. Pas Python. On a alors deux solutions :

* On opte pour l’infini potentiel en représentant les suites par [des itérateurs](http://irem.univ-reunion.fr/spip.php?article997)
* On reste dans le domaine fini en ne calculant qu’un nombre fini de termes de la suite.

C’est le second choix qui sera fait ici, et les termes en question seront stockés dans une liste Python. Celle-ci sera typiquement construite par compréhension pour une suite explicite, et par ajouts successifs (avec *append*) pour une suite récurrente.

**Calcul de termes d’une suite, de sommes de termes, de seuil**

*Termes*

Un exemple de suite définie explicitement est celle des carrés dits « parfaits » :

[carré(n) **for** n **in** range(20)]

s’évalue à

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361]

Si on affecte une variable u à cette liste, le premier terme u0 sera obtenu par u[0], le second terme u1 par u[1] etc. Et la somme des termes peut s’obtenir en appliquant la fonction *sum* à cette liste.

La suite des carrés est un exemple de nombres figurés. Les [nombre figuré](http://fr.wikipedia.org/wiki/nombre_figur%C3%A9) sont surtout connus parce que [Pythagore](http://fr.wikipedia.org/wiki/Pythagore) leur accordait des propriétés magiques.

Un exemple de suite récurrente est présent dans l’étude de la [tour d’Hanoï](http://fr.wikipedia.org/wiki/tour_d%27Hano%C3%AF), jeu à un joueur inventé par [Édouard Lucas](http://fr.wikipedia.org/wiki/%C3%89douard_Lucas) vers la fin du 19e siècle :

Pour résoudre la tour d’Hanoï à zéro disque, on effectue zéro mouvement : h0=0. Pour résoudre la tour d’Hanoï à n+1 disques, on effectue

* hn mouvements pour transporter tous les disques sauf le plus grand vers le piquet intermédiaire ;
* 1 mouvement pour placer le plus grand disque sur le piquet final ;
* hn mouvements pour transporter les disques restants depuis le piquet intermédiaire vers le piquet final.

Donc hn+1=hn+1+hn=2×hn+1 :

1. **def** mouvements\_Hanoi(n):
2. h = [0]
3. **for** n **in** range(1,n):
4. h.append(2\*h[-1]+1)
5. **return** h

Explication de la ligne 4 : Le dernier élément de la liste h se note h[-1]. On ajoute ("append") 1 de plus que son double (relation de récurrence) à la fin de la liste h.

*Somme de termes*

Pour additionner les n premiers termes de la suite u représentée par une liste de n nombres, on peut faire

1. S = 0
2. **for** terme **in** u:
3. S += terme

ou bien

sum(u)

*calcul de seuil*

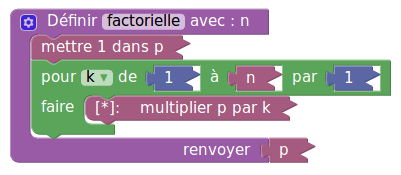
Pour ceux qui envisagent de faire MATHS EXPERTES

Par exemple, ion se demande combien il faut de disques à la tour d’Hanoï pour nécessiter au moins 1000 mouvements :

1. **def** disques\_Hanoi(nmvts):
2. h = 0
3. ndisques = 0
4. **while** h<1000:
5. h = 2\*h+1
6. ndisques += 1
7. **return** ndisques

**Calcul de factorielle**

Au fait c’est quoi une [factorielle](http://fr.wikipedia.org/wiki/factorielle) ? On propose de définir la factorielle de n comme produit des n premiers entiers non nuls. La question étant de savoir comment ça se calcule. On propose l’algorithme de sommation, dans lequel ici on remplace les additions par des multiplications. Algorithme trouvé difficile [par plusieurs étudiants de BTS](http://irem.univ-reunion.fr/spip.php?article950). Voici l’algorithme décrit en Sofus :



Et en Python ça donne (par exemple à l’aide de SofusPy) :

1. **def** factorielle(n):
2. **assert** n==int(n) **and** n>=0
3. p = 1
4. **for** k **in** range(1,n+1):
5. p \*= k
6. **return** p

Explication de la ligne 2 : La fonction factorielle n’est définie que pour les entiers naturels, on restreint donc son calcul aux entiers naturels.

Remarque : Une fois importées les fonctions du module *math*, l’expression suivante donne le même résultat :

[factorial(n) **for** n **in** range(10)]

Voici [les factorielles sur OEIS](https://oeis.org/A000142)

**Liste des premiers termes d’une suite : suite de Syracuse, suite de Fibonacci**

*Suite de Collatz*

[Remarque historique](javascript:;)

La suite de Collatz consiste à itérer une transformation appelée ici *mélange* car elle mélange les entiers naturels :

1. **def** est\_pair(n):
2. **return** n%2==0
4. **def** mélange(n):
5. **if** est\_pair(n):
6. **return** n//2
7. **else**:
8. **return** 3\*n+1
10. **def** Collatz(u0,n):
11. u = [u0]
12. **for** k **in** range(n):
13. u.append(mélange(u[-1]))
14. **return** u

Le premier argument de cette fonction est le premier terme de la suite (un entier naturel au choix) et le second argument est le nombre de termes à calculer.

*Fibonacci*

[Remarques historiques](javascript:;)

Voici [la suite de Fibonacci sur OEIS](https://oeis.org/A000045).

Et en Python :

1. **def** Fibonacci(n):
2. **assert** n==int(n) **and** n>=0
3. **if** n==0:
4. **return** [1]
5. **else**:
6. F = [1,1]
7. **for** k **in** range(2,n+1):
8. F.append(F[-1]+F[-2])
9. **return** F

On peut voir d’autres exemples de suites dans [ces regards croisés](http://revue.sesamath.net/spip.php?article571) sur le sujet.

Une première question à poser, avant d’étudier les algorithmes sur les suites, est celle de la définition d’une suite : **Qu’est-ce que c’est qu’une suite ?**

Comme le programme insiste sur la notion de fonction, on peut proposer cette définition : Une suite est une fonction définie sur **N**. Plus précisément, si l’ensemble d’arrivée est un ensemble de nombres (typiquement **R**) la suite est dite *numérique*.

Se pose alors le problème de la représentation d’une suite en Python : La machine sur laquelle tourne Python ne disposant que d’une quantité de mémoire finie, on ne peut y caser une infinité de nombres u(0)=u0, u(1)=u1, etc.

Le langage Haskell gère les suites qu’il considère comme des listes infinies. Pas Python. On a alors deux solutions :

* On opte pour l’infini potentiel en représentant les suites par [des itérateurs](http://irem.univ-reunion.fr/spip.php?article997)
* On reste dans le domaine fini en ne calculant qu’un nombre fini de termes de la suite.

C’est le second choix qui sera fait ici, et les termes en question seront stockés dans une liste Python. Celle-ci sera typiquement construite par compréhension pour une suite explicite, et par ajouts successifs (avec *append*) pour une suite récurrente.

**Calcul de termes d’une suite, de sommes de termes, de seuil**

*Termes*

Un exemple de suite définie explicitement est celle des carrés dits « parfaits » :

[carré(n) **for** n **in** range(20)]

s’évalue à

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361]

Si on affecte une variable u à cette liste, le premier terme u0 sera obtenu par u[0], le second terme u1 par u[1] etc. Et la somme des termes peut s’obtenir en appliquant la fonction *sum* à cette liste.

La suite des carrés est un exemple de nombres figurés. Les [nombre figuré](http://fr.wikipedia.org/wiki/nombre_figur%C3%A9) sont surtout connus parce que [Pythagore](http://fr.wikipedia.org/wiki/Pythagore) leur accordait des propriétés magiques.

Un exemple de suite récurrente est présent dans l’étude de la [tour d’Hanoï](http://fr.wikipedia.org/wiki/tour_d%27Hano%C3%AF), jeu à un joueur inventé par [Édouard Lucas](http://fr.wikipedia.org/wiki/%C3%89douard_Lucas) vers la fin du 19e siècle :

Pour résoudre la tour d’Hanoï à zéro disque, on effectue zéro mouvement : h0=0. Pour résoudre la tour d’Hanoï à n+1 disques, on effectue

* hn mouvements pour transporter tous les disques sauf le plus grand vers le piquet intermédiaire ;
* 1 mouvement pour placer le plus grand disque sur le piquet final ;
* hn mouvements pour transporter les disques restants depuis le piquet intermédiaire vers le piquet final.

Donc hn+1=hn+1+hn=2×hn+1 :

1. **def** mouvements\_Hanoi(n):
2. h = [0]
3. **for** n **in** range(1,n):
4. h.append(2\*h[-1]+1)
5. **return** h

Explication de la ligne 4 : Le dernier élément de la liste h se note h[-1]. On ajoute ("append") 1 de plus que son double (relation de récurrence) à la fin de la liste h.

*Somme de termes*

Pour additionner les n premiers termes de la suite u représentée par une liste de n nombres, on peut faire

1. S = 0
2. **for** terme **in** u:
3. S += terme

ou bien

sum(u)

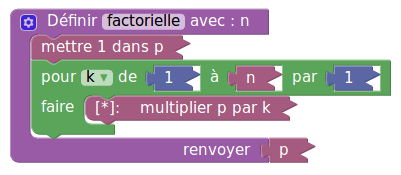
*calcul de seuil*

Par exemple, ion se demande combien il faut de disques à la tour d’Hanoï pour nécessiter au moins 1000 mouvements :

1. **def** disques\_Hanoi(nmvts):
2. h = 0
3. ndisques = 0
4. **while** h<1000:
5. h = 2\*h+1
6. ndisques += 1
7. **return** ndisque

**Calcul de factorielle**

Au fait c’est quoi une [factorielle](http://fr.wikipedia.org/wiki/factorielle) ? On propose de définir la factorielle de n comme produit des n premiers entiers non nuls. La question étant de savoir comment ça se calcule. On propose l’algorithme de sommation, dans lequel ici on remplace les additions par des multiplications. Algorithme trouvé difficile [par plusieurs étudiants de BTS](http://irem.univ-reunion.fr/spip.php?article950). Voici l’algorithme décrit en Sofus :



Et en Python ça donne (par exemple à l’aide de SofusPy) :

1. **def** factorielle(n):
2. **assert** n==int(n) **and** n>=0
3. p = 1
4. **for** k **in** range(1,n+1):
5. p \*= k
6. **return** p

Explication de la ligne 2 : La fonction factorielle n’est définie que pour les entiers naturels, on restreint donc son calcul aux entiers naturels.

Remarque : Une fois importées les fonctions du module *math*, l’expression suivante donne le même résultat :

[factorial(n) **for** n **in** range(10)]

Voici [les factorielles sur OEIS](https://oeis.org/A000142)

**Liste des premiers termes d’une suite : suite de Syracuse, suite de Fibonacci**

*Suite de Collatz*

[Remarque historique](javascript:;)

La suite de Collatz consiste à itérer une transformation appelée ici *mélange* car elle mélange les entiers naturels :

1. **def** est\_pair(n):
2. **return** n%2==0
4. **def** mélange(n):
5. **if** est\_pair(n):
6. **return** n//2
7. **else**:
8. **return** 3\*n+1
10. **def** Collatz(u0,n):
11. u = [u0]
12. **for** k **in** range(n):
13. u.append(mélange(u[-1]))
14. **return** u

Le premier argument de cette fonction est le premier terme de la suite (un entier naturel au choix) et le second argument est le nombre de termes à calculer.

*Fibonacci*

[Remarques historiques](javascript:;)

Voici [la suite de Fibonacci sur OEIS](https://oeis.org/A000045).

Et en Python :

1. **def** Fibonacci(n):
2. **assert** n==int(n) **and** n>=0
3. **if** n==0:
4. **return** [1]
5. **else**:
6. F = [1,1]
7. **for** k **in** range(2,n+1):
8. F.append(F[-1]+F[-2])
9. **return** F

On peut voir d’autres exemples de suites dans [ces regards croisés](http://revue.sesamath.net/spip.php?article571) sur le sujet.

1ERE 3/4/5/6

Le produit scalaire permet de refaire avec les équations de cercle ce qui a été vu en 2nde sur les équations de droite :

*équation comme prédicat*

L’équation d’un cercle de diamètre donné peut se calculer ainsi :

1. **def** équation\_cercle\_diamètre(A,B):
2. **def** test(M):
3. **return** sont\_orthogonaux(vecteur(A,M),vecteur(B,M))
4. **return** test

*représentation textuelle de l’équation*

Par centre et rayon :

1. **def** équation\_cercle(centre,rayon):
2. (Cx,Cy) = centre
3. R = rayon
4. **return** "x²+y²+({})×x+({})×y = {}".format(-2\*Cx,-2\*Cy,R\*\*2-Cx\*\*2-Cy\*\*2)

**Approximation de π par la méthode d’Archimède**

On peut montrer que lorsque x est petit (en radians), sin(x) et tan(x) sont proches de x et de plus encadrent x. Alors si on découpe un cercle de rayon 1 en un grand nombre n d’arcs de longueur 2π/n, le demi-périmètre π est compris entre celui n×sin(2π/n)/2 du polygone régulier inscrit, et celui n×tan(2π/n)/2 du polygone régulier circonscrit au cercle. De plus, lorsque n est grand, ces deux nombres sont proches l’un de l’autre et *a fortiori* de π.

Vu comme ça cet algorithme semble tourner en rond puisque pour calculer sin(2π/n) et tan(2π/n) on s’attend à avoir besoin d’une valeur approchée de π. Mais la trigonométrie permet de calculer sin(x/2) et tan(x/2) à partir de sin(x) et tan(x), en n’utilisant que les 4 opérations et la racine carrée (calcul de moyennes arithmétique et géométrique). En partant comme l’a fait Archimède d’hexagones inscrit et circonscrit au cercle unité, on construit cette suite (a est le périmètre de l’hexagone inscrit, b celui de l’hexagone circonscrit) :

1. **def** archimede(n):
2. a = 3
3. b = 6/racine(3)
4. p = [moyenne(a,b)]
5. **for** k **in** range(n):
6. b \*= a/p[-1]
7. a = racine(a\*b)
8. p.append(moyenne(a,b))
9. **return** p

Cette fonction ne renvoie pas la suite des encadrements de π obtenus par Archimède mais celle des approximations de π. Pour retrouver le résultat d’Archimède il faut faire archimede(4).

Les fonctions *racine* et *moyenne* ont été définies en Seconde.

BON TRAVAIL !!!!

OMJS

PS : SI VOUS TROUVEZ DES ERREURS N’HESITEZ PAS A ME LES REMONTER OU ALORS DES METHODES PLUS RAPIDES (SUGGEREZ LES MOIS !!!)

L’ Enseignement est un apprentissage de tout instant avec des obstacles qu’il faut surmonter !!!