

Chapitre # (NUM) 4

Suites numériques

Résumé & Plan

Le but de ce chapitre est de savoir mettre en place les différents algorithmes rencontrés dans le chapitre de Mathématiques (savoir retourner le n -ième terme, la liste des termes, seuil d'atteinte d'une limite, etc.).

- 1 **Algorithmes sur les suites**
- 2 **Exercices**
- 3 **Solutions des exercices**

1. ALGORITHMES SUR LES SUITES



Résumé des attendus

Voici ce qu'il faut savoir faire en Python à propos des suites :

- Les fonctions permettant de calculer un terme donné d'une suite.
- Les fonctions permettant de calculer le premier terme ou le premier indice d'une suite pour lequel une condition donnée est vérifiée pour la première fois.
- Construire la liste des termes d'une suite jusqu'à un indice donné/ce qu'une condition soit vérifiée.
- Tracer le graphe de la suite en exploitant la liste des termes précédents.

Nous illustrerons ces différents programmes sur les trois suites suivantes :

- **[Explicite]** La suite (u_n) , définie explicitement, vérifiant :

$$\forall n \in \mathbb{N}^*, \quad u_n = \left(1 + \frac{a}{n}\right)^n$$

où $a \in \mathbb{R}$ est choisi par l'utilisateur. On peut prouver qu'elle converge vers e^a .

- **[Récurrence d'ordre 1]** La suite (v_n) , définie par une relation de récurrence d'ordre 1, vérifiant :

$$\begin{cases} v_0 = a \in \mathbb{R} \text{ choisi par l'utilisateur} \\ \forall n \in \mathbb{N}, \quad v_{n+1} = v_n + e^{v_n}. \end{cases}$$

On peut prouver qu'elle est croissante quel que soit $a \in \mathbb{R}$ et en déduire, par l'absurde, qu'elle tend vers $+\infty$.

- **[Récurrence d'ordre 2]** La suite (w_n) , définie par une relation de récurrence d'ordre 2, vérifiant :

$$\begin{cases} w_0 = a \in \mathbb{R} \text{ choisi par l'utilisateur} \\ w_1 = b \in \mathbb{R} \text{ choisi par l'utilisateur} \\ \forall n \in \mathbb{N}, \quad w_{n+2} = \frac{\cos(w_n + w_{n+1})}{n+2}. \end{cases}$$

On peut prouver par encadrement qu'elle tend vers 0 puis en déduire que :

$$w_n \underset{n \rightarrow \infty}{\sim} \frac{1}{n}.$$

Remarque 1 (Nommage des variables) Dans tous nos programmes, on respectera les deux conventions suivantes : les variables $n, i, j \dots$ serviront à stocker des valeurs d'**indices**, les variables $u, v, w \dots$ serviront quant à elles à stocker des valeurs de **termes** des suites. Même si la suite s'appelle autrement que (u_n) , on appelle u la variable stockant son terme.

1.1. Calcul du n -ième terme

SUITE EXPLICITE. C'est le cas le plus simple, il suffit de retourner l'expression correspondant au terme saisi par l'utilisateur. Voici par exemple le code de la fonction `terme_u(a, n)` qui retourne le terme u_n avec a et n en paramètre de fonction :

■ Terme n d'une suite définie explicitement

```
def terme_u(a, n):
    """
    Renvoie la valeur de u_n lorsque u_1=a
```

```
"""
u = (1+a/n)**n
return u
```

```
>>> terme_u(2, 1)
3.0
>>> terme_u(0, 1)
1.0
```

CAS PARTICULIERS DES SOMMES (SÉRIES) ET PRODUITS. Des suites peuvent être définies à l'aide d'une somme ou d'un produit. On utilisera alors les méthodes vues dans le chapitre sommes/produits du cours de Mathématiques.

>_☞ (Calcul de $\sum_{k=p}^n a_k$)

```
def somme_a(p, n):
    S = 0
    for k in range(p, n+1):
        S += a_k # le terme a_k est à taper à la main |
                ↪ fonction de la somme
    return S
```

Par exemple, la fonction ci-après réalise le calcul de $\sum_{k=p}^n \cos(kx)$, avec $x \in \mathbb{R}$.

```
def somme_cos(p, n, x):
    S = 0
    for k in range(p, n+1):
        S += ma.cos(k*x)
    return S

>>> somme_cos(0, 10, 1)
-0.4174477464559059
>>> somme_cos(0, 10, 0) # résultat attendu car on somme 1, |
↪ onze fois
11.0
```

>_☞ (Calcul de $\prod_{k=p}^n a_k$)

```
def produit(p, n):
    P = 1
```

```
for k in range(p, n+1):
    P *= a_k # à adapter en fonction de la somme
return P
```

Par exemple, la fonction ci-après réalise le calcul de $\prod_{k=p}^n e^{kx}$, avec $x \in \mathbb{R}$.

```
def produit(p, n, x):
    P = 1
    for k in range(p, n+1):
        P *= ma.exp(k*x)
    return P

>>> produit(0, 10, 1)
7.694785265142015e+23
>>> produit(0, 10, 0) # résultat attendu
1.0
```

SUITE RÉCURRENTE D'ORDRE 1. Pour calculer v_n on procède ainsi.

1. On prévoit un test **if** pour la condition initiale, puis :
2. on initialise une variable u avec la valeur de v_0 .
3. On parcourt à l'aide d'une boucle **for** tous les indices i de 1 à n (l'indice mathématique correspondant). Pour chaque valeur de i , on remplace u (qui contient v_{i-1}) par sa nouvelle valeur, v_i , à l'aide de la formule de récurrence.
4. En sortie de boucle, u contient la valeur de v_n ; il suffit donc de renvoyer u .

Voici par exemple le code de la fonction `terme_v(a, n)` qui renvoie le terme v_n avec $v_0 = a$ et n en paramètre de fonction :

■ ■ Terme n d'une suite récurrente d'ordre 1

```
def terme_v(a, n):
    """
    Renvoie la valeur de v_n lorsque v_0 = a
    """
    if n == 0:
        return a
    else:
        u = a
        for i in range(1, n+1):
            # u est ici la valeur précédente
            u = u + ma.exp(u)
            # v est ici la valeur suivante
```

```
return u
```

```
>>> terme_v(0, 1)
1.0
>>> terme_v(0, 2)
3.718281828459045
```

Remarque 2 (Version « universelle » sans if) Le test `if` n'est ici pas obligatoire. En effet, si $n = 0$ alors la boucle `for` ne s'exécutera pas (bornes dans le mauvais sens) et donc on retournera bien $v = a$.

SUITE RÉCURRENTÉ D'ORDRE 2. Pour calculer w_n on procède ainsi :

1. On prévoit un test `if` pour les deux conditions initiales, puis :
2. on initialise **deux** variables, u et v , avec les valeurs de w_0 et de w_1 .
3. On parcourt à l'aide d'une boucle `for` tous les indices i de 2 à n (l'indice mathématique correspondant). Pour chaque valeur de i , on calcule le terme suivant noté w à l'aide de la relation de récurrence puis on remplace **simultanément** (donc au moyen d'une **double-affectation**) u et v (qui contiennent respectivement les valeurs de w_{i-1} et de w_i) par les nouvelles valeurs (qui contiennent alors respectivement les valeurs de w_i et w_{i+1}).
4. En sortie de boucle, v contient la valeur de w_n ; il suffit donc de renvoyer w .

Voici par exemple le code de la fonction `terme_w(a, b, n)` qui retourne le terme w_n avec $w_0 = a$, $w_1 = b$ et n en paramètre de fonction.

■ Terme n d'une suite récurrente d'ordre 2

```
def terme_w(a, b, n):
    """
    Renvoie la valeur de  $w_n$  lorsque  $w_0 = a$  et  $w_1 = b$ 
    """
    if n == 0:
        return a
    elif n == 1:
        return b
    else:
        u, v = a, b
        for i in range(2, n+1):
            u contient  $w_{i-1}$  et v contient  $w_i$ 
            u, v = v, ma.cos(u+v)/i
            u contient désormais  $w_i$  et v contient  $w_{i+1}$ 
```

```
return v
```

```
>>> terme_w(0, 1, 0)
0
>>> terme_w(0, 1, 1)
1
>>> terme_w(0, 1, 2)
0.2701511529340699
```

Remarque 3 (Version « universelle » sans if) Là encore, le test `if` n'est pas indispensable. Il est possible d'adapter la seconde partie de la fonction (changement de boucle `for` et dans la récurrence) afin qu'elle convienne également aux cas $n = 0$ et $n = 1$.

```
def terme_w_bis(a, b, n):
    """
    Renvoie la valeur de  $w_n$  lorsque  $w_0 = a$  et  $w_1 = b$ 
    """
    u, v = a, b
    for i in range(1, n+1):
        u, v = v, ma.cos(u+v)/(i+1)
    return u

>>> terme_w_bis(0, 1, 0)
0
>>> terme_w_bis(0, 1, 1)
1
>>> terme_w_bis(0, 1, 2)
0.2701511529340699
```

Elle retourne bien également les bons termes.

1.2. Calcul du premier terme/indice vérifiant une condition

Pour réaliser ces fonctions, il va falloir calculer les termes successivement jusqu'à ce que la condition soit vérifiée. Pour cela on utilisera une boucle `while` : tant que la condition **n'est pas vérifiée**, on calcule le terme suivant ; reste alors à renvoyer le dernier terme/indice. On parle en général *d'algorithme de seuil*.

! Attention

Contrairement aux boucles `for`, une boucle `while` ne permet pas de parcourir automatiquement les différents indices. Il faudra donc dans nos programmes

! introduire une variable contenant la valeur de l'indice, l'initialiser correctement et l'augmenter de 1 à chaque passage dans la boucle.

SUITE EXPLICITE. Par définition de la limite, on sait par exemple que comme la suite (u_n) converge vers e^a , on a :

$$\forall \varepsilon > 0, \exists n_0 \in \mathbb{N}, n \geq n_0 \implies |u_n - e^a| < \varepsilon.$$

Voici une fonction cherchant l'entier n_0 en question.

■ Algorithme de seuil pour une suite explicite

```
def seuil_u(a, eps):
    """
    Renvoie le premier indice n pour lequel |u_n-exp(a)|<eps
    """
    n = 1
    u = (1+a/n)**n
    while abs(u-exp(a)) >= eps:
        n += 1
        u = (1+a/n)**n
    return n
```

Remarque 4 Il est parfois possible de calculer l'entier n_0 explicitement en résolvant une équation/inéquation, mais cela n'est pas possible sur cet exemple (même chose pour les suivants).

SUITE RÉCURRENTÉ D'ORDRE 1. Pour réaliser ces fonctions, il y a un unique changement à apporter aux fonctions précédentes : remplacer la boucle **for** par une boucle **while**.

On sait par exemple que la suite (v_n) tend en croissant vers $+\infty$, donc :

$$\forall A \in \mathbb{R}, \exists n_0 \in \mathbb{N}, n \geq n_0 \implies v_n \geq A.$$

Voici la fonction qui renvoie l'indice n_0 , a et A étant en paramètre de fonction.

■ Algorithme de seuil pour une suite récurrente d'ordre 1

```
def seuil_v(a, A):
    """
    Renvoie le premier indice n pour lequel v_n >= A
    """
    n = 0
    v = a
```

```
while v < A:
    n += 1
    v = v + ma.exp(v)
return n
```

```
>>> n_0 = seuil_v(1, 10)
>>> n_0
2
>>> terme_v(1, n_0)
44.911837503175164
>>> terme_v(1, n_0-1)
3.718281828459045
```

SUITE RÉCURRENTÉ D'ORDRE 2. Pour réaliser ces fonctions, il y a un unique changement à apporter aux fonctions précédentes : remplacer la boucle **for** par une boucle **while**.

On sait par exemple que la suite (w_n) converge vers 0, donc :

$$\forall \varepsilon \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}, n \geq n_0 \implies |w_n| < \varepsilon.$$

Voici la fonction qui renvoie l'indice n_0 , a , b et ε étant en paramètre de fonction.

■ Algorithme de seuil pour une suite récurrente d'ordre 2

```
def seuil_w(a, b, eps):
    """
    Renvoie le premier indice n pour lequel |w_n|<eps
    """
    n = 0
    u, v = a, b
    while abs(u) >= eps:
        n += 1
        u, v = v, ma.cos(u+v)/n
    return n
```

```
>>> n_0 = seuil_w(1, 1, 10**(-3))
>>> n_0
1001
>>> terme_w(0, 1, n_0)
0.000998998999007198
>>> terme_w(0, 1, n_0-1)
```

0.00099999799399421

1.3. Construction de la liste des termes et tracé

On construit la liste de proche en proche à l'aide d'une boucle **for** ou **while** et de la méthode `append` sur les listes. Vous noterez que les versions avec seuil permettent de retrouver les algorithmes de seuil précédents (en retournant la longueur de la liste obtenue).

SUITE EXPLICITE. On donne à titre d'exemple les fonctions qui renvoient la liste des termes u_1 à u_n .

■ Liste de termes sous condition ou non – Suite explicite

```
def liste_terme_u(a, n):
    """
    Renvoie la liste [u_1,...,u_n] (u_0 n'existe pas !)
    """
    L = []
    for i in range(1, n+1):
        L.append((1+a/i)**i)
    return L
```

```
>>> liste_terme_u(1, 10)
[2.0, 2.25, 2.37037037037037, 2.44140625, 2.4883199999999994, \
↳ 2.5216263717421135, 2.546499697040712, 2.565784513950348, \
↳ 2.5811747917131984, 2.5937424601000023]
```

```
def liste_seuil_u(a, eps):
    """
    Renvoie la liste [u_1,...,u_n] où n est le premier indice n \
↳ pour lequel |u_n-exp(a)|<eps"""
    n = 1
    L = [(1+a/n)**n]
    while abs(L[-1] - ma.exp(a)) >= eps:
        n += 1
        L.append((1+a/n)**n)
    return L
```

```
>>> liste_seuil_u(1, 10**(-1))
[2.0, 2.25, 2.37037037037037, 2.44140625, 2.4883199999999994, \
↳ 2.5216263717421135, 2.546499697040712, 2.565784513950348, \
↳ 2.5811747917131984, 2.5937424601000023, 2.6041990118975287, \
↳ 2.613035290224676, 2.6206008878857308]
```

SUITE RÉCURRENTE D'ORDRE 1. On construit une liste L telle que $L[i]$ contienne la valeur de v_i . Il n'est alors plus nécessaire de conserver le terme précédent dans une variable : lors du calcul de v_i , on dispose de la valeur de v_{i-1} , c'est précisément $L[-1]$, le dernier terme ajouté.

On donne à titre d'exemple une fonction qui renvoie la liste des termes v_0 à v_n et une autre qui renvoie la liste de tous les termes de (v_n) jusqu'à ce que $v_n \geq A$.

■ Liste de termes sous condition ou non – Suite d'ordre 1

```
def liste_terme_v(a, n):
    """
    Renvoie la liste [v_0,...,v_n]
    """
    L = [a]
    for _ in range(1, n+1):
        L.append(L[-1] + ma.exp(L[-1]))
    return L
```

```
>>> liste_terme_v(1, 3)
[1, 3.718281828459045, 44.911837503175164, \
↳ 3.1986240606431162e+19]
```

```
def liste_seuil_v(a, A):
    """
    Renvoie la liste [v_0,...,v_n] où n est le premier indice n \
↳ pour lequel v_n>=A
    """
    n = 0
    L = [a]
    while L[-1] < A:
        n += 1 ici cet entier n ne sert finalement pas car la récurrence a des coefficients \
↳ indépendants de n
        L.append(L[-1] + ma.exp(L[-1]))
    return L
```

```
>>> liste_seuil_v(1, 3)
[1, 3.718281828459045]
```

SUITE RÉCURRENTE D'ORDRE 2. On construit une liste L telle que $L[i]$ contienne la valeur de w_i . Là encore, il n'est alors plus nécessaire de conserver les termes précédent dans des variables : lors du calcul de w_i , on dispose de la valeur de w_{i-1} dans $L[i-1]$ et de w_{i-2} dans $L[i-2]$. On donne à titre d'exemple une fonction qui renvoie la liste des termes w_0 à w_n et une autre qui renvoie la liste de tous les termes de (w_n) jusqu'à ce que $|w_n| < \epsilon$. Notons que dans deux fonctions, et ce afin d'éviter la gestion de cas particuliers, on suppose que la liste finale contient au moins w_0 et w_1 .

■ Liste de termes sous condition ou non – Suite d'ordre 2

```
def liste_terme_w(a, b, n):
    """
    Renvoie la liste [w_0, w_1, ..., w_n] (n >= 1)
    """
    if n == 0:
        return [a]
    elif n == 1:
        return [a, b]
    else:
        L = [a, b]
        for i in range(2, n+1):
            L.append((ma.cos(L[-2]+L[-1])/i))
        return L
```

```
>>> liste_terme_w(1, 1, 10)
[1, 1, -0.2080734182735712, 0.23415848554264707, \
↪ 0.24991495098085725, 0.1770213081859069, 0.15170644429544455, \
↪ 0.13520769153107332, 0.11989021517204179, 0.10751539939271154, \
↪ 0.09742545791160233]
```

```
def liste_seuil_w(a, b, eps):
    """
    Renvoie la liste [w_0, w_1, ..., w_n] (n >= 1) où n est le \
↪ premier indice pour lequel |w_n| < eps
    """
    n = 0
    L = [a, b]
    while abs(L[n]) >= eps:
```

```
n += 1 ici cet entier n sert
L.append((ma.cos(L[-2]+L[-1])/n)
return L
```

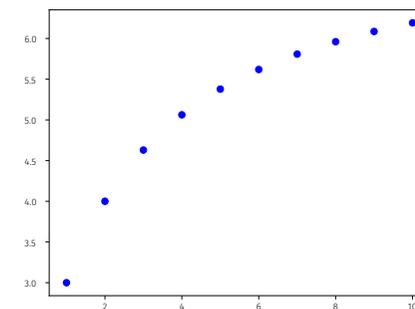
Remarque 5 (Suites imbriquées) Il faut savoir également en pratique adapter ces algorithmes à des suites récurrentes imbriquées.

1.4. Tracer une suite

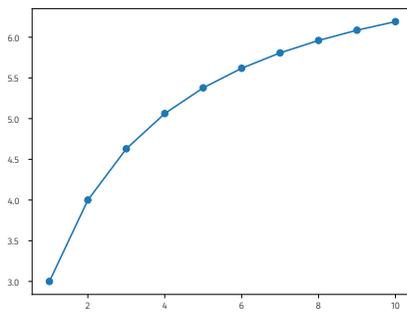
On s'y prend comme pour les fonctions, on a besoin donc de la liste des termes de ladite suite. Traçons par exemple (u_n) .

■ Tracé de la suite (u_n) sur $[[0, 10]]$

```
import matplotlib.pyplot as plt
n = 10
X = list(range(1, n+1)) # entiers entre 1 et n
Y = liste_terme_u(2, n)
plt.plot(X, Y, "bo") # o : style de marker, des points non \
↪ reliés
```



```
plt.plot(X, Y, marker = 'o') # des points reliés cette fois, un \
↪ petit peu plus visuel
```



2. EXERCICES

Soient $\ell, u_n \in \mathbb{R}$ et $\varepsilon > 0$. Rappelons que « $u_n \in \mathbb{R}$ est une valeur approchée de ℓ à ε près » signifie que : $|u_n - \ell| < \varepsilon$.

Vous l'aurez compris, avec ces notations, des suites convergentes vers ℓ fournissent une valeur approchée de la limite à ε près pour n assez grand. L'enjeu est parfois d'atteindre une certaine précision lorsqu'on ne connaît pas la valeur de ℓ (des suites adjacentes peuvent alors être utiles).

2.1. Pour les suites

Exercice 1 | Réurrence d'ordre 1 [Solution](#) On considère la suite arithmético-géométrique définie par :

$$u_0 = 4, \quad \forall n \geq 0, \quad u_{n+1} = 2 - \frac{u_n}{2}.$$

1. Écrire un programme itératif prenant en argument un entier n et qui calcule u_n .
2. Quelle est la limite ℓ de la suite (u_n) ? Écrire un programme prenant en argument $\varepsilon > 0$ et qui affiche le premier entier n tel que $|u_n - \ell| \leq \varepsilon$.

Exercice 2 | Réurrence d'ordre 2 – Nombre d'or et FIBONACCI [Solution](#) On rappelle que la suite de FIBONACCI (F_n) est définie par :

$$F_0 = 1, \quad F_1 = 1, \quad \forall n \in \mathbb{N}, \quad F_{n+2} = F_{n+1} + F_n.$$

On appellera dans la suite *nombre d'or* le réel $\alpha = \frac{1 + \sqrt{5}}{2}$, et on note $\beta = \frac{1 - \sqrt{5}}{2}$.

1. Écrire un programme itératif, prenant en argument un entier n à l'utilisateur et retournant la liste des termes successifs u_0, \dots, u_n de la suite de FIBONACCI. Tracer ces termes sur un graphique.

2. Dans le cours de Mathématiques, nous avons établi l'expression suivante :

$$\forall n \in \mathbb{N}, \quad F_n = \left(\frac{5 + \sqrt{5}}{10} \right) \alpha^n + \left(\frac{5 - \sqrt{5}}{10} \right) \beta^n.$$

Montrer par le calcul que $\lim_{n \rightarrow \infty} \frac{F_{n+1}}{F_n} = \alpha$.

3. En déduire à l'aide de Python une valeur approchée du nombre d'or.

Exercice 3 | Réurrences imbriquées [Solution](#) On considère deux suites (a_n) et (j_n) vérifiant pour tout $n \in \mathbb{N}$:

$$\begin{cases} j_{n+1} = j_n + 8a_n, \\ a_{n+1} = 0,25j_n, \end{cases} \quad a_0 = 0, \quad j_0 = 2.$$

La suite a_n décrit le nombre d'adultes au temps $n \in \mathbb{N}$ d'une population structurée en deux classes d'âge, et j_n le nombre de juvéniles (enfants).

1. Interpréter le système en terme de dynamique de la population.
2. Créer une fonction Python d'en-tête `population(n)` qui retourne deux listes, l'une contenant j_0, \dots, j_n , l'autre a_0, \dots, a_n . Tracer alors les deux suites sur $[0, 40]$. Qu'en déduire sur notre population? Une classe d'âge semble-t-elle devenir prédominante?

Exercice 4 | Suites adjacentes et approximation de la limite [Solution](#) On considère les suites (a_n) et (b_n) définies par :

$$\forall n \in \mathbb{N}, \quad b_{n+1} = \frac{a_n + b_n}{2}, \quad a_{n+1} = \frac{2}{b_{n+1}}, \quad a_0 = 1, \quad b_0 = 2.$$

On peut montrer que ces suites sont adjacentes, convergent vers $\sqrt{2}$, et que :

$$\forall n \in \mathbb{N}, \quad a_n \leq \sqrt{2} \leq b_n.$$

Proposer une fonction d'en-tête `approx_racine2(eps)` qui retourne une valeur approchée de $\sqrt{2}$ à ε près utilisant les suites $(a_n), (b_n)$.

Exercice 5 | Conjecturer un équivalent [Solution](#) Soit la suite (u_n) définie par :

$$\forall n \in \mathbb{N}^*, \quad u_{n+1} = \left(1 + \frac{1}{n} \right) \sin(u_n), \quad u_1 \in]0, \pi[.$$

Conjecturer, en tracer les suites, la nature des suites $(u_n)_{n \in \mathbb{N}^*}, (\sqrt{n}u_n)_{n \in \mathbb{N}^*}$. Conjecturer alors un équivalent simple de $(u_n)_{n \in \mathbb{N}^*}$.

Exercice 6 | Produit [Solution](#) On définit, pour tout $n \in \mathbb{N}^*$, le produit :

$$p_n = \prod_{k=1}^n \left(1 + \frac{k}{n^2} \right).$$

1. Écrire un script d'en-tête $p(n)$ qui retourne la valeur de p_n pour tout $n \in \mathbb{N}^*$.
2. Conjecturer la valeur de la limite : $\lim_{n \rightarrow \infty} p_n^2$.

2.2. Pour les séries

Les séries seront étudiées plus en détail en deuxième année, ce sont des suites de la forme $\left(\sum_{k=n_0}^n u_k\right)_{n \geq n_0}$ où :

- (u_n) est une suite réelle,
- n_0 est un entier généralement égal à 0.

L'objectif des exercices qui suivent est d'analyser leur convergence.

Attention Trouver la nature d'une série n'est pas chose facile!

Il n'y a aucun lien clair *a priori* entre la nature de $\left(\sum_{k=n_0}^n u_k\right)_{n \geq n_0}$ et la nature de $(u_n)_{n \geq n_0}$. Par exemple, nous avons vu en cours que :

- $\sum_{k=0}^n \frac{1}{k} \xrightarrow{n \rightarrow \infty} \infty$ alors que $\frac{1}{n} \xrightarrow{n \rightarrow \infty} 0$,
- $\left(\sum_{k=0}^n \frac{1}{k^2}\right)_{n \geq 1}$ converge alors que $\frac{1}{n^2} \xrightarrow{n \rightarrow \infty} 0$.

Exercice 7 | Limite mystère *Solution* Pour tout $x \in \mathbb{R}$ et n entier, on note :

$$S_n(x) = \sum_{k=0}^n \frac{x^k}{k!}.$$

1. Rappeler une fonction d'en-tête `factorielle(n)`, où n est un entier, qui renvoie la valeur de $n!$. *De manière itérative ou récursive selon votre préférence.*
2. Écrire une fonction d'en-tête `somme_1(n, x)` prenant en argument un réel x et un entier n , et retournant $S_n(x)$.
3. Vers quoi semble converger la suite $(S_n(x))_{n \in \mathbb{N}}$ pour tout $x \in \mathbb{R}$? Le constater avec Python en commençant par tester pour $x = 0; 1$. Puis, une fois le résultat conjecturé, confirmer la conjecture en traçant sur un même graphique (par exemple sur l'intervalle $[0, 10]$) :
 - le graphe d'une fonction judicieusement choisie,
 - et le graphe de $x \mapsto S_5(x)$, $x \mapsto S_{10}(x)$

Exercice 8 | Série inverse des carrés *Solution* On note ici $S_n = \sum_{k=1}^n \frac{1}{k^2}$. On a montré en cours (en vérifiant que la suite $(S_n)_{n \in \mathbb{N}^*}$ est croissante majorée) que la suite $(S_n)_{n \in \mathbb{N}^*}$ converge. On souhaite ici conjecturer la valeur de sa limite $S \in \mathbb{R}$. On admet l'inégalité suivante : $\forall n \in \mathbb{N}^*, |S - S_n| \leq \frac{1}{n}$.

1. Conjecturer à l'aide de Python l'existence et la valeur de $\lim_{n \rightarrow \infty} \sqrt{6S_n}$.
2. On souhaite écrire une fonction d'en-tête `approx_S(eps)` qui prend en argument un réel $\text{eps} > 0$, et affichant une valeur approchée de S à eps près, en utilisant l'inégalité admise.
 - 2.1) **[Méthode 1 : calculer n]** Soit une précision $\text{eps} > 0$ fixée. Trouver un entier $n_0 \in \mathbb{N}$ tel que $|S - S_n| \leq \text{eps}$. En déduire une première version de `approx_S(eps)`.
 - 2.2) **[Méthode 2 : un `while`]** Écrire une seconde version de `approx_S(eps)` à l'aide d'une boucle `while`.
3. En admettant le résultat de la première question, déduire une valeur approchée de $\frac{\pi^2}{6}$ à 10^{-3} près.

2.3. Planche Agro-Véto

Cet exercice n'est à traiter que si tout le reste a été terminé.

Exercice 9 | Développement en série entière d'arctan – Extrait Agro-Véto *Solution*

Pour $n \in \mathbb{N}$ et $x \in \mathbb{R}$ on note

$$u_n(x) = \sum_{k=0}^n \frac{(-1)^k x^{2k+1}}{2k+1}.$$

1. **>_☛** Écrire un script Python permettant de calculer $u_n(x)$ pour $n \in \mathbb{N}$ et $x \in \mathbb{R}$. Que peut-on conjecturer sur le comportement asymptotique de la suite pour $|x| \leq 1$ et $|x| > 1$?
2. Soit $n \in \mathbb{N}$ et $x \in [-1, 1]$. On note $I_n(x) = \int_0^x \frac{t^{2n+2}}{1+t^2} dt$. Quelle est la limite de $(I_n(x))$ quand $n \rightarrow +\infty$?
3. **3.1)** Soit $n \in \mathbb{N}$ et $x \in [-1, 1]$. Montrer que :

$$\sum_{k=0}^n \int_0^x (-1)^k t^{2k} dt = \int_0^x \frac{dt}{1+t^2} + (-1)^n I_n(x)$$
- 3.2) En déduire que pour $x \in [-1, 1]$, $(u_n(x))_{n \geq 0}$ converge vers $\arctan(x)$, et que :

$$\forall n \in \mathbb{N}, |4u_n(1) - \pi| \leq \frac{4}{2n+3}.$$

- 3.3)  Grâce au résultat de la question précédente, écrire un script Python qui, étant donné un réel $\varepsilon > 0$, calcule une approximation de π à ε près.
- 3.4) Montrer que les suites $(u_{2n}(1))$ et $(u_{2n+1}(1))$ sont adjacentes. En déduire que
4. Justifier que $\arctan\left(\frac{1}{2}\right) + \arctan\left(\frac{1}{3}\right) = \frac{\pi}{4}$. *Indication : On pourra utiliser la formule $\tan(a + b) = \frac{\tan(a) + \tan(b)}{1 - \tan(a)\tan(b)}$ valable dès que tous les termes sont définis.*
5.  En déduire une autre méthode d'approximation de π et comparer informatiquement leur efficacité.

3. SOLUTIONS DES EXERCICES

Solution (exercice 1) [Énoncé](#)

```
1. def suiteU_arithmgeo(n):
    u = 4
    for _ in range(1, n+1):
        u = 2 - u/2
    return u
```

On peut aussi proposer une version récursive.

```
def suiteU_arithmgeo_rec(n):
    if n == 0:
        return 4
    else:
        return 2 - suiteU_arithmgeo_rec(n-1)/2
```

2. C'est une suite arithmético-géométrique, de raison $-1/2 \in]-1, 1[$. Donc en explicitant la suite en fonction de n , nous avons : $u_n \xrightarrow{n \rightarrow \infty} \frac{4}{3}$, c'est le point fixe de la suite. On en déduit le script suivant :

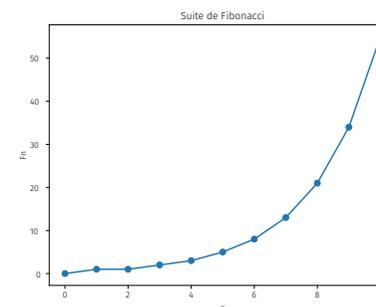
```
def approx_lim_U_arithmgeo(eps):
    u = 4
    n = 0
    while abs(u - 4/3) > eps:
        u = 2 - u/2
        n += 1
    return n
>>> approx_lim_U_arithmgeo(10**(-2))
9
```

Solution (exercice 2) [Énoncé](#)

```
1. def termes_fibo(n):
    if n == 0:
        return [1]
    elif n == 1:
        return [1, 1]
    else:
        L = [0, 1]
        for i in range(2, n+1):
            L.append(L[-2]+L[-1])
        return L
```

On peut ensuite tracer les termes de la suite.

```
plt.plot(termes_fibo(10), marker = "o")
plt.title("Suite de Fibonacci")
plt.xlabel("n")
plt.ylabel("Fn")
```



2. Soit $n \in \mathbb{N}$, alors :

$$\begin{aligned} \frac{F_{n+1}}{F_n} &= \frac{\left(\frac{5+\sqrt{5}}{10}\right)\alpha^{n+1} + \left(\frac{5-\sqrt{5}}{10}\right)\beta^{n+1}}{\left(\frac{5+\sqrt{5}}{10}\right)\alpha^n + \left(\frac{5-\sqrt{5}}{10}\right)\beta^n} \\ &= \alpha^{n+1} \times \frac{\left(\frac{5+\sqrt{5}}{10}\right) + \left(\frac{5-\sqrt{5}}{10}\right)\left(\frac{\beta}{\alpha}\right)^{n+1}}{\left(\frac{5+\sqrt{5}}{10}\right) + \left(\frac{5-\sqrt{5}}{10}\right)\left(\frac{\beta}{\alpha}\right)^n} \\ &\xrightarrow{n \rightarrow \infty} \alpha \times \frac{5+\sqrt{5}}{5+\sqrt{5}} = \boxed{\alpha}. \end{aligned}$$

$\left|\frac{\beta}{\alpha}\right| < 1$

3. Pour obtenir une valeur approchée de α , on peut donc retourner $\frac{F_{n+1}}{F_n}$ pour n assez grand.

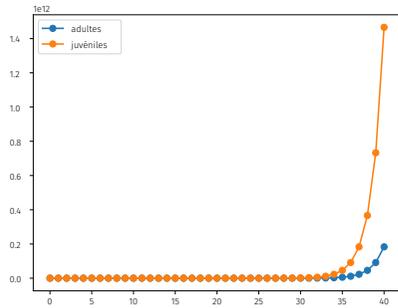
```
>>> L = termes_fibo(100)
>>> L[-1]/L[-2] # valeur approchée du nombre d'or
1.618033988749895
>>> (1+5**(0.5))/2
1.618033988749895
```

Solution (exercice 3) [Énoncé](#)

- La seconde ligne du système signifie que seul 1/4 des juvéniles passent à l'âge adulte au temps suivant.
- La première ligne signifie qu'entre deux temps n et $n + 1$, il y a un apport de juvéniles égal 8 par adulte de la génération précédente.

```
def population(n):
    a, j = 0, 2
    A = [a]
    J = [j]
    for _ in range(1, n+1):
        a, j = 0.25*j, j+8*a
        A.append(a)
        J.append(j)
    return A, J
```

```
A, J = population(40)
plt.plot(A, label="adultes", marker="o")
plt.plot(J, label="juvéniles", marker="o")
plt.legend()
```



La classe des juvéniles prend largement le dessus sur la population des adultes.

Solution (exercice 4) [Énoncé](#) L'idée est alors de calculer successivement les termes des deux suites, jusqu'à ce que l'écart entre elles soit inférieur à la précision fixée.

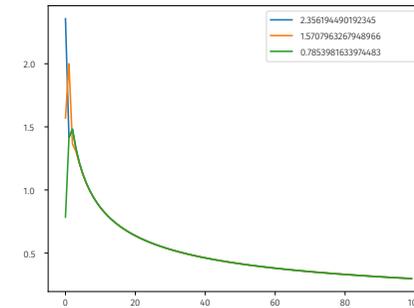
```
def approx_racine2(eps):
    a, b = 1, 2
    while b - a > eps:
        b = (a+b)/2
        a = 2/b
    return (a+b)/2

>>> approx_racine2(10**(-3))
1.4142135623746899
>>> ma.sqrt(2) # valeur exacte
1.4142135623730951
```

Solution (exercice 5) [Énoncé](#)

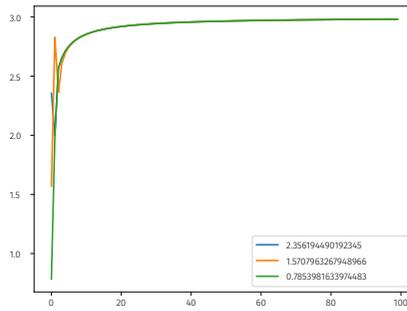
```
def liste_termes_U(n, u_1):
    """
    retourne la liste des termes de la suite et de la racine \
    ↪ x la suite
    """
    U = [u_1]
    for k in range(2, n+1):
        U.append((1+1/(k-1))*ma.sin(U[-1]))
    return U

# testons avec plusieurs conditions initiales
cond_init = [3*ma.pi/4, ma.pi/2, ma.pi/4]
for u_1 in cond_init:
    plt.plot(liste_termes_U(100, u_1), label=str(u_1))
plt.legend()
```



Dans tous les cas de condition initiale testés, on observe une convergence de la suite vers une limite finie. Traçons maintenant la suite multipliée par la racine.

```
cond_init = [3*ma.pi/4, ma.pi/2, ma.pi/4]
for u_1 in cond_init:
    U = liste_termes_U(100, u_1)
    racU = [ma.sqrt(i)*U[i-1] for i in range(1, 101)]
    plt.plot(racU, label=str(u_1))
plt.legend()
```



On conjecture d'autre part que :

$$\sqrt{n}u_n \xrightarrow{n \rightarrow \infty} 3 \implies u_n \underset{n \rightarrow \infty}{\sim} \frac{3}{\sqrt{n}}$$

Solution (exercice 6) [Énoncé](#)

```
1. def p(n):
    P = 1
    for k in range(1, n+1):
        P *= 1+k/n**2
    return P

>>> p(1)
2.0
>>> p(2)
1.875

2. >>> p(10)**2
2.8961962061931783
>>> p(10*4)**2 # on ne reconnait rien ?
2.7633932648235526
>>> ma.exp(1) # ben si bien sûr !
2.718281828459045
```

On conjecture : $\lim_{n \rightarrow \infty} p_n^2 = e$.

Solution (exercice 7) [Énoncé](#)

```
1. """
    retourne la factorielle de n de manière itérative
    """
    P = 1
    for k in range(1, n+1):
        P *= k
```

```
    return P
<It>

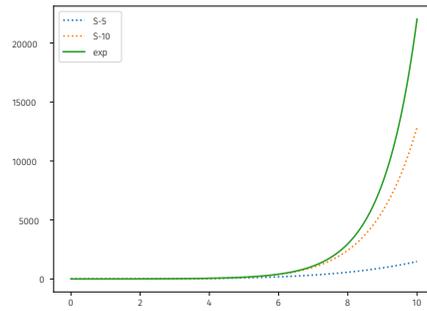
#[Rec]
def factorielle_rec(n):
    """
    Retourne la factorielle de n, de manière récursive
    """
    if n == 0:
        return 1
    return n*factorielle_rec(n-1)
<Rec>

2. def sommel(n, x):
    S = 1
    for k in range(1, n+1):
        S += x**k / factorielle(k)
    return S

3. >>> sommel(10**3, 0) # proche de e^0
1.0
>>> sommel(10**3, 1) # proche du nombre e
2.7182818284590455
```

La suite semble converger vers l'exponentielle de x. Constatons-le par exemple avec un graphique.

```
import matplotlib.pyplot as plt
X = np.linspace(0, 10, 10**3)
Y = [ma.exp(x) for x in X]
Z = [sommel(5, x) for x in X]
Zbis = [sommel(10, x) for x in X]
plt.plot(X, Z, linestyle = ":", label="S-5")
plt.plot(X, Zbis, linestyle = ":", label="S-10")
plt.plot(X, Y, label="exp")
plt.legend()
```



Solution (exercice 8) [Énoncé](#)

1. On peut par exemple écrire une fonction retournant la valeur de S_n .

```
def S(n):
    S = 0
    for k in range(1, n+1):
        S += 1 / k**2
    return S

>>> (6*S(10))**(1/2)
3.04936163598207
>>> (6*S(100))**(1/2)
3.1320765318091053
>>> (6*S(1000))**(1/2)
3.1406380562059946
```

On conjecture alors : $\sqrt{6S_n} \xrightarrow[n \rightarrow \infty]{} \pi$.

2. Soit $\varepsilon > 0$. L'inégalité admise nous apprend que si on choisit n de sorte que $\frac{1}{n} \leq \varepsilon$, alors S_n est une valeur approchée de S à ε près.

2.1) [Méthode 1 : calculer n] On résout $\frac{1}{n} \leq \varepsilon \iff n \geq \frac{1}{\varepsilon}$. Ceci est encore

équivalent à : $n \geq \left\lceil \frac{1}{\varepsilon} \right\rceil + 1$. On retourne donc la valeur de S_n pour cet entier n .

```
def approx_S(eps):
    n = ma.floor(1/eps)+1
    return S(n)

>>> eps = 10**(-1)
>>> S = approx_S(eps)
>>> S
1.558032193976458
>>> ma.pi**2/6
```

```
1.6449340668482264
>>> abs(S-ma.pi**2/6) <= eps
True
```

2.2) [Méthode 2 : un while]

```
def approx_S(eps):
    S = 1
    n = 1
    while 1/n > eps:
        n += 1
        S += 1 / n**2
    return S

>>> S = approx_S(10**(-3))
>>> S
1.6439345666815615
>>> ma.pi**2/6
1.6449340668482264
>>> abs(S-ma.pi**2/6) <= 10**(-3)
True
```

Solution (exercice 9) [Énoncé](#)

1. >_☰



```
def u(n, x):
    """
    retourne la valeur de u(n,x)
    """
    S = 0
    for k in range(n):
        S += (-1)**k*x**(2*k+1)/(2*k+1)
    return S
```

Testons.

```
>>> u(10, 1.2)
-0.07289063746835467
>>> u(50, 1.2)
-407996.6991505616
>>> u(100, 1.2)
-16880754195420.87
>>> u(10, 0.9)
0.7298154560111498
>>> u(50, 0.9)
0.732814969864735
>>> u(100, 0.9)
0.7328151017847537
```

On constate que pour $x > 1$, la série ne semble pas converger, alors que pour $x \leq 1$ il semble y avoir convergence.

2. Soit $n \in \mathbb{N}$ et $x \in [-1, 1]$. Rappelons que l'on ne peut pas permuter limite et intégrale. En revanche, on peut majorer :

$$0 \leq I_n(x) = \int_0^x \frac{t^{2n+2}}{1+t^2} dt \leq \int_0^x t^{2n+2} dt = \frac{x^{2n+3} - 0}{2n+3} \xrightarrow{n \rightarrow \infty} 0,$$

puisque $x \in [-1, 1]$ (si $|x| < 1$, on voit que le numérateur tend vers zéro en utilisant le résultat de convergence sur les suites géométriques, si $x = \pm 1$ ce sont des règles usuelles sur les limites). Ainsi, par théorème d'encadrement :

$$\lim_{n \rightarrow \infty} \int_0^x \frac{t^{2n+2}}{1+t^2} dt = 0.$$

3. 3.1) Soit $n \in \mathbb{N}$ et $x \in [-1, 1]$. Alors, par linéarité de l'intégrale :

$$\begin{aligned} \sum_{k=0}^n \int_0^x (-1)^k t^{2k} dt &= \int_0^x \sum_{k=0}^n (-1)^k t^{2k} dt \\ &= \int_0^x \sum_{k=0}^n (-t^2)^k dt \\ &= \int_0^x \frac{1 - (-t^2)^{n+1}}{1 + t^2} dt \quad \left. \begin{array}{l} \text{car } t^2 \neq -1 \end{array} \right\} \\ &= \int_0^x \frac{dt}{1+t^2} - (-1)^{n+1} \int_0^x \frac{t^{2n+2}}{1+t^2} dt \\ &= \int_0^x \frac{dt}{1+t^2} + (-1)^{n+2} I_n(x) \quad \left. \begin{array}{l} \text{car } n, n+2 \text{ ont même parité} \end{array} \right\} \\ &= \boxed{\int_0^x \frac{dt}{1+t^2} + (-1)^n I_n(x)}. \end{aligned}$$

- 3.2) Soit $x \in [-1, 1]$, alors $\int_0^x \frac{dt}{1+t^2} = \arctan(x)$. Par ailleurs la suite $((-1)^n I_n)$ est le produit d'une suite bornée multipliée par une suite qui tend vers zéro, elle converge donc vers zéro. Enfin, on peut calculer l'intégrale de gauche :

$$\int_0^x (-1)^k t^{2k} dt = (-1)^k \frac{x^{2k+1}}{2k+1},$$

en sommant on obtient donc $u_n(x)$. Donc, finalement :

$$\boxed{(u_n(x))_{n \geq 0} \text{ converge vers } \arctan(x)}.$$

- 3.3) Soit $n \in \mathbb{N}$. Évaluons 3.1) en $x = 1$. On obtient :

$$u_n(1) = \arctan 1 + (-1)^n I_n(x) = \frac{\pi}{4} + (-1)^n I_n,$$

soit

$$|4u_n(1) - \pi| = |4(-1)^n I_n(1)| \leq 4 \int_0^1 t^{2n+2} dt = \frac{4}{2n+3}.$$

Donc pour tout $n \in \mathbb{N}$:

$$\boxed{|4u_n(1) - \pi| \leq \frac{4}{2n+3}}.$$

- 3.4) >_☞



```
def approx_pi(prec):
    n = 0
    S = 1
    while 4/(2*n+1) > prec:
        n += 1
        S += (-1)**n/(2*n+1)
    return 4*S

>>> approx_pi(10**(-3))
3.1420924036835256
```

3.5) Soit $n \in \mathbb{N}$. Constatons déjà que :

$$u_n(1) = \sum_{k=0}^n \frac{(-1)^k}{2k+1}.$$

Alors, par télescopes :

$$\begin{aligned} u_{2n+2}(1) - u_{2n}(1) &= \frac{1}{2(2n+2)+1} - \frac{1}{2(2n)+1} \\ &= \frac{1}{4n+5} - \frac{1}{4n+1} \leq 0, \end{aligned}$$

$$u_{2n+3}(1) - u_{2n+1}(1) = -\frac{1}{4n+7} + \frac{1}{4n+3} \geq 0,$$

$$u_{2n+1}(1) - u_{2n}(1) = \frac{-1}{2(2n+1)+1} \xrightarrow{n \rightarrow \infty} 0.$$

Donc : $(u_{2n}(1))$ et $(u_{2n+1}(1))$ sont adjacentes.

4. Puisque

$$-\frac{\pi}{2} < \arctan\left(\frac{1}{2}\right) < \frac{\pi}{2}, \quad -\frac{\pi}{2} < \arctan\left(\frac{1}{3}\right) < \frac{\pi}{2},$$

la somme $\arctan\left(\frac{1}{2}\right) + \arctan\left(\frac{1}{3}\right)$ est dans $\left]-\frac{\pi}{2}, \frac{\pi}{2}\right[$, donc la tangente est bien définie. D'après la formule rappelée, on a :

$$\tan\left(\arctan\left(\frac{1}{2}\right) + \arctan\left(\frac{1}{3}\right)\right) = \frac{\frac{1}{2} + \frac{1}{3}}{1 - \frac{1}{6}} = 1 = \tan\left(\frac{\pi}{4}\right).$$

On peut ensuite appliquer \arctan de chaque côté, utiliser à nouveau que $\arctan\left(\frac{1}{2}\right) + \arctan\left(\frac{1}{3}\right) \in \left]-\frac{\pi}{2}, \frac{\pi}{2}\right[$ et $\frac{\pi}{4} \in \left]-\frac{\pi}{2}, \frac{\pi}{2}\right[$ donc on déduit en utilisant que $\arctan(\tan y) = y$ pour tout $y \in \left]-\frac{\pi}{2}, \frac{\pi}{2}\right[$,

$$\arctan\left(\frac{1}{2}\right) + \arctan\left(\frac{1}{3}\right) = \frac{\pi}{4}.$$

5. On sait d'après les questions précédentes (3.1 évaluée en $1/2, 1/3$) qu'une

bonne approximation de $\arctan\left(\frac{1}{2}\right) + \arctan\left(\frac{1}{3}\right)$ est

$$4\left(u_n\left(\frac{1}{2}\right) + u_n\left(\frac{1}{3}\right)\right)$$

avec n assez grand.



```
def approx_pi_bis():
    return 4*(u(10**3, 1/2)+u(10**3, 1/3))

>>> approx_pi_bis()
3.1415926535897922
```

Pour savoir qui est la plus efficace on peut comparer l'écart entre une valeur approchée et la vraie valeur, à nombre d'itérations égaux.

```
>>> import math as ma
>>> abs(4*(u(10**3, 1/2)+u(10**3, 1/2))) - ma.pi < \
<- abs(4*u(10**3, 1) - ma.pi)
```

False

La première méthode apparaît donc comme meilleure.