

Chapitre (ANN) 3 Synthèse pour le numérique

Résumé & Plan

Un condensé de commandes importantes et de scripts classiques à connaître parfaitement pour les concours, mais axées sur le numérique.

- 1 **Graphismes**
- 2 **Algorithmes sur les suites**
- 3 **Tableaux**
- 4 **Méthodes numériques**
- 5 **Aléatoire & Statistiques**

1 GRAPHISMES

On importe le sous-module `matplotlib.pyplot` avec `import matplotlib.pyplot as plt`

1.1 Généralités

PARAMÈTRES OPTIONNELS POUR plot

Propriétés	Rôle	Valeurs possibles
<code>color</code>	Trace le nuage de points	'red', 'blue' etc. d'abscisses X et ordonnées Y

PARAMÈTRES OPTIONNELS POUR plot

label	Légende de la courbe	une chaîne
<code>linestyle</code>	type de ligne (pointillés, etc.)	'-', '-.', ':'
<code>linewidth</code>	épaisseur du trait	un entier
<code>marker</code>	forme des points	'+', ',', 'o', '1', '2' etc. ('o' adapté aux suites)

! Attention

La commande `savefig` doit être placée avant `show` et après `plot`.

OPTIONS POUR linestyle

Option	Rôle
" "	les points ne sont pas reliés
"_"	tracé en trait plein
"- -"	tracé avec des tirets
": "	tracé en pointillés
"- ."	alternance de tirets et de points

COMMANDES DE FORMATAGE DES TITRES, LÉGENDES, AXES

Nom	Rôle
-----	------

<code>plt.legend(loc = 'upper left')</code>	permet de placer les légendes (ici haut gauche)
<code>plt.title('titre')</code>	permet de placer un titre au tracé
<code>plt.axis('off')</code>	permet d'enlever les axes
<code>plt.axis('scaled')</code>	impose la même échelle aux deux axes
<code>plt.axis(xmin = .., xmax = .., ..)</code>	impose des valeurs min/max à chaque axe en spécifiant xmax, ymax, xmin, ymin
<code>plt.grid(True)</code>	trace le quadrillage
<code>plt.xlabel('texte')</code>	nom pour l'axe des abscisses
<code>plt.ylabel('texte')</code>	nom pour l'axe des ordonnées

1.2 Tracé d'une suite ou d'une fonction

Soit f une fonction définie sur $[a, b]$, avec $a, b \in \mathbb{R}, a < b$, définie sur cet intervalle et que l'on souhaite tracer.

■ Tracer une fonction

```
import numpy as np
import matplotlib.pyplot as plt
```

```
X = np.linspace(a, b, 10**3) # création d'une subdivision des \
↪ abscisses
Y = [f(x) for x in X] # les ordonnées
# OU, si et seulement si f ne contient que des fonctions numpy
Y = f(X)
plt.plot(X, Y, options)
plt.show()
```

Soit (u_n) une suite définie sur \mathbb{N} , que l'on souhaite tracer par exemple sur $[[1, 10]]$. On suppose cette codée par une fonction Python notée u .

■ Tracer une suite

```
import numpy as np
import matplotlib.pyplot as plt
```

```
X = list(range(1, 11))
Y = [u(n) for n in X] # les ordonnées
```

```
plt.plot(X, Y, marker = 'o', autres options)
plt.show()
```

2 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 :

$$\begin{cases} w_0 = a \in \mathbb{R} \text{ choisi par l'utilisateur} \\ w_1 = b \in \mathbb{R} \text{ choisi par l'utilisateur} \\ \forall n \in \mathbb{N}, \quad w_{n+2} = \frac{5}{6}w_{n+1} - \frac{1}{6}w_n. \end{cases}$$

On peut prouver qu'elle converge vers zéro.

Remarque 1 (Nomage 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.

SUITE EXPLICITE. C'est le cas le plus simple, il suffit de renvoyer l'expression correspondant au terme saisi par l'utilisateur. Voici par exemple le code de la fonction `terme_u(a, n)` qui renvoie 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
    """
    return (1+a/n)**n
```

```
>>> 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 en \
                ← 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 RÉCURRENTTE 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)
        # u 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 renverra 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 à l'aide de la relation de récurrence puis on remplace simultanément (donc au moyen d'une double-affectation) u et v par les nouvelles valeurs.
4. En sortie de boucle, v contient la valeur de w_n .

Voici par exemple le code de la fonction `terme_w(a, b, n)` qui renvoie 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, v = v, (5/6)*v-(1/6)*u
    return v

```

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

```
0
```

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

```
1
```

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

```
0.8333333333333334
```

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, (5/6)*v-(1/6)*u
    return u

```

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

```
0
```

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

```
1
```

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

```
0.8333333333333334
```

Elle renvoie bien également les bons termes.

2.2 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.

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 :

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

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 not (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 not (abs(u) < eps):
        n += 1
        u, v = v, (5/6)*v - (1/6)*u
    return n
```

```
>>> n_0 = seuil_w(1, 1, 10**(-3))
```

```
>>> n_0
```

```
12
```

```
>>> terme_w(0, 1, n_0)
```

```
0.0014535536914610499
```

```
>>> terme_w(0, 1, n_0-1)
```

```
0.002895817324383145
```

2.3 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 renvoyant 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, 2.5216
263717421135, 2.546499697040712, 2.565784513950348, 2.58117479171319
84, 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 not 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, 2.5216
263717421135, 2.546499697040712, 2.565784513950348, 2.58117479171319
84, 2.5937424601000023, 2.6041990118975287, 2.613035290224676, 2.620
6008878857308]
```

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 $v_n > A$.

■ 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>=M
    """
    L = [a]
    while L[-1] < A:
        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((5/6)*L[-1] - (1/6)*L[-2])
        return L
```

```
>>> liste_terme_w(1, 1, 10)
[1, 1, 0.6666666666666667, 0.38888888888888906, 0.2129629629629631
3, 0.11265432098765445, 0.05838477366255152, 0.029878257887517197, 0
.015167752629172412, 0.007660084209724144, 0.0038554447365747183]
```

```
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
    """
    L = [a, b]
    while abs(L[-2]) >= eps:
        L.append((5/6)*L[-1] - (1/6)*L[-2])
    return L
```

```
>>> liste_seuil_w(1, 1, 10**(-1))
[1, 1, 0.6666666666666667, 0.38888888888888906, 0.2129629629629631
3, 0.11265432098765445, 0.05838477366255152, 0.029878257887517197]
```

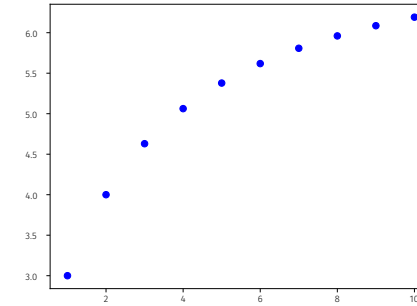
Remarque 5 (Suites imbriquées) Il faut savoir également en pratique adapter ces algorithmes à des suites récurrentes imbriquées.

2.4 Tracer une suite

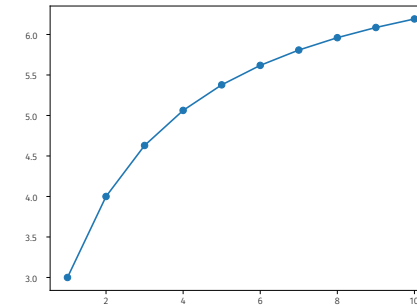
On s'y prend comme pour les fonctions, on a besoin donc de la liste des termes de ladite suite. Traçons par exemple (u_n) .

■ 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
```



3

TABLEAUX

CONSTRUCTEURS DE TABLEAUX

Operations	Commande	Commentaire
Création	<code>A = np.array(...)</code>	On indique une liste de listes en argument

Matrice nulle	<code>A = np.zeros((n, p))</code>	Un tuple est demandé en argument, donc deux parenthèses
Matrice ATILA J_n	<code>A = np.ones((n, p))</code>	Un tuple est demandé en argument, donc deux parenthèses
Matrice identité	<code>A = np.identity(n)</code> ou <code>A = np.eye(n)</code> †	
Matrice diagonale	<code>A = np.diag(L)</code>	La liste L contient la diagonale
Subdivision de pas h de [a, b]	<code>np.arange(a, b, h)</code>	Analogue de <code>list(range(a, b, h))</code>
Subdivision à n points de [a, b]	<code>np.linspace(a, b, n)</code>	On s'en est servi pour tracer des suites, fonctions, etc.
Copie	<code>B = A.copy()</code>	Important pour pouvoir modifier B sans modifier A, comme pour les listes
Coefficients	<code>A[i, j]</code> ou <code>A[i][j]</code>	Terme i, j du tableau
Ligne i	<code>A[i]</code>	
Colonne j	<code>A[:, j]</code>	« : » signifie en <i>slicing</i> « on prend tout »

† eye comme identity en anglais.

Attention
Attention aux copies, comme pour les listes. On peut utiliser ici `np.copy` pour effectuer une « copie en dur » d'un tableau.

Méthode (ANN) 3.1 (Créer une matrice) Plusieurs options s'offrent à vous.

- Si la matrice est de petite taille, on écrit directement les coefficients.
- Si la matrice est de grande taille (typiquement dépendant d'un certain entier n), on peut :
 - ◊ soit utiliser des commandes existantes si la matrice est proche d'une matrice usuelle. Par exemple, `np.eye`, `np.zeros`, `np.ones`, etc.
 - ◊ Soit partir d'une matrice nulle initialisée à la bonne taille (avec `np.zeros`), puis la compléter des bons coefficients à l'aide d'une boucle `for`.

■ ■ En indice

```
n, p = len(A), len(A[0])
for i in range(n):
    for j in range(p):
        ...
```

■ ■ En valeur

```
for ligne in M:
    for x in ligne:
        ...
```

Méthode (ANN) 3.2 (Calcul des puissances d'une matrice avec Python) Soit M un tableau carré correspondant à une matrice M carrée. Il n'y a pas de fonction toute faite dans numpy pour calculer M^n . Rappelons également que M^{**n} élève les coefficients de M à la puissance n mais n'effectue pas le produit matriciel. On procède comme suit :

- on initialise un tableau P à l'identité.
- On effectue n fois l'affectation $P = P @ M$.
- On renvoie P.

4 MÉTHODES NUMÉRIQUES

4.1 Équations différentielles

>_ (Méthode d'EULER)

```
def euler(f, y0, tau, N):
    """
    f : fonction, y0 : valeur en zéro,
    tau : borne max de l'intervalle,
    N : nombre d'intervalles de la subdivision->renvoie le couple
    (subdivision, solution approchée) selon la méthode d Euler
    """
    h = tau/N
    T = np.linspace(0, tau, N+1) # N+1 points dans la \
    ← discrétisation
    Y = [y0]
    for i in range(N):
        y = Y[-1] + h*f(T[i], Y[-1])
        Y.append(y) # Y[-1] est le dernier y ajouté
    return T, Y
```

>_☞ (Dichotomie)

```
def dico(a, b, f, prec):
    """
    renvoie une valeur approchée d'un zéro de f entre a et b \
    ← avec précision
    prec
    """
    g, d = a, b
    while d - g > prec:
        m = (g + d)/2
        if f(g)*f(m) <= 0:
            # changement de signe sur [g,m]
            d = m
        else:
            # pas de changement de signe sur [g,m]
            g = m
    return (g + d)/2
```

On peut aussi adopter une version récursive.

```
def dico_rec(a, b, f, prec):
    """
    renvoie une valeur approchée d'un zéro de f entre a et b \
    ← avec précision prec
    """
    if b - a <= prec:
        return (a + b)/2
    else:
        m = (a + b)/2
        if f(a)*f(m) <= 0:
            # changement de signe sur [a,m]
            return dico_rec(a, m, f, prec)
        else:
            # pas de changement de signe sur [a,m]
            return dico_rec(m, b, f, prec)
```

>_☞ (Méthodes des rectangles)

```
def rectangle_RG(f, a, b, n):
    """
    Calcule la somme des rectangles gauche associée à f
    """
    S = 0
    h = (b-a)/n
    for k in range(n):
        S += f(a+h*k)
    return S*h

def rectangle_RD(f, a, b, n):
    """
    Calcule la somme des rectangles droite associée à f
    """
    S = 0
    h = (b-a)/n
    for k in range(1, n+1):
        S += f(a+h*k)
    return S*h
```

5 ALÉATOIRE & STATISTIQUES**5.1 Statistiques descriptives**

>_☞ (Modalités)

```
def sans_doublon(L):
    """
    renvoie la liste des éléments de L, chaque élément \
    ↪ apparaissant une unique fois
    """
    M = []
    for x in L:
        if x not in M:
            M.append(x)
    return M
```

>_☞ (Dictionnaire d'effectifs ↔ Liste des observations)

```
def dico_occu(L):
    D = {}
    for x in L:
        if x not in D:
            D[x] = 1
        else :
            D[x] += 1
    return D

def dico_occu_vers_liste(D):
    L = []
    for x in D:
        # x est une modalité, que l'on duplique autant de fois \
        ↪ que nécessaire
        eff_x = D[x]
        for _ in range(eff_x):
            L.append(x)
    return L
```

>_☞ (Grandeurs univariées)

```
def moyenne(L):
    """
    Renvoie la moyenne des éléments d'une liste
    """
    S = 0
    for x in L:
        S += x
    return S/len(L)
```

```
def moyenne_avec_eff(D):
```

```
    """
    Renvoie la moyenne d'une série associée au dictionnaire des \
    ↪ effectifs
    D
    """
    S = 0
    N = 0 # nombre d'éléments de la série
    for x in D:
        eff_x = D[x]
        S += x*eff_x
        N += eff_x
    return S/N
```

```
def variance(L):
```

```
    """
    Renvoie la variance, version KH
    """
    S2 = 0
    for x in L:
        S2 += x**2
    return S2/len(L) - moyenne(L)**2
```

```
def etendue(L):
```

```
    """
    Renvoie l'étendue de la série statistique des éléments de L
    """
    mini = L[0]
    maxi = L[0]
    for x in L[1:]:
        if x < mini:
            mini = x
        elif x > maxi:
            maxi = x
    return maxi - mini
```

```
def mediane(L):
```

```
    """
    Cherche la médiane d'une liste, après tri rapide des \
    ↪ observations
    """
    L_tri = tri_rapide(L)
```

```

n = len(L)
if n % 2 == 1:
    # Nombre impair d'observations
    return L_tri[n//2]
else:
    # Nombre pair d'observations
    return (L_tri[n//2-1] + L_tri[n//2])/2
def quartiles(L):
    """
    renvoie Q1, Q2, Q3, après tri rapide des observations
    """
    L_tri = tri_rapide(L)
    n = len(L)
    if n % 2 != 0:
        # Nombre impair d'observations
        Q2 = L_tri[n//2]
    else:
        # Nombre pair d'observations
        Q2 = (L_tri[n//2-1] + L_tri[n//2])/2
    if n % 4 != 0:
        # Nombre non multiple de 4 d'observations
        Q1 = L_tri[n//4]
        Q3 = L_tri[(3*n)//4]
    else:
        # Nombre multiple de 4 d'observations
        Q1 = L_tri[n//4-1]
        Q3 = L_tri[(3*n)//4-1]
    return Q1, Q2, Q3

```

>_🔗 (Grandeurs bivariées)

```

def covariance(L, M):
    """
    Renvoie la covariance des deux séries
    """
    Prod = [L[i]*M[i] for i in range(len(M))]
    return moyenne(Prod) - moyenne(L)*moyenne(M)
def coeff_cor(X, Y):
    """
    Renvoie le coefficient de corrélation des deux séries

```

```

    """
    return covariance(X, \
        ↪ Y)/(ma.sqrt(variance(X))*ma.sqrt(variance(Y)))
def covariance(L, M):
    """
    Renvoie la covariance des deux séries
    """
    Prod = [L[i]*M[i] for i in range(len(M))]
    return moyenne(Prod) - moyenne(L)*moyenne(M)
def regression_lin(X, Y):
    """
    renvoie les coefficients a, b de régression linéaire \
    ↪ associée au
    nuage de points (X, Y)
    """
    a = covariance(X, Y)/variance(X)
    b = moyenne(Y) - a*moyenne(X)
    return a, b

```

5.2 Simulation de lois classiques

On importe le module random avec :

```
import random as rd
```

Commande	Effet
rd.random()	Tirer un nombre entre 0 et 1 selon une $\mathcal{U}[0, 1]$
rd.randint(i, j)	Tirer un entier entre i et j selon une $\mathcal{U}[[i, j]]$
rd.choice(L)	Tirer un élément au hasard dans L
rd.shuffle(L)	Mélange les éléments de L de manière aléatoire

>_🔗 (Lois discrètes)

```
def unif_entier(a, b):
```

```
    """
    Renvoie une simulation de l'uniforme sur [a,b] entier
    """
    return np.floor(a+(b+1-a)*rd.random())
def bernoulli(p):
    """
    simule une bernoulli
    """
    if rd.random() < p:
        return 1
    else:
        return 0
import random as rd
def binomiale(n,p):
    """
    simule une binomiale
    """
    S = 0
    for i in range(n):
        if rd.random() < p:
            S += 1
    return S
```