- EXERCICE 1 -

On considère une suite $(u_n)_{n\in\mathbb{N}}$ définie par son premier terme $u_0 = 1$ et pour tout entier n, $u_{n+1} = u_n + \frac{1}{u_n}$.

1. **Écrire un programme en Python qui calcule et affiche la valeur de u_n lorsque l'utilisateur entre la valeur de n au clavier. *Pour* $n = 100$, *on obtient* $u_{100} = 14.284064040284603$.**

```
n = int(input('Donnez la valeur de n : '))
u = 1                    # u0=1
for i in range(1,n+1) : # (n+1) dans range pour arriver à u_n
    u = u + 1/u ;       # calcul u1, ..., un (écrase l'ancienne valeur)
print(u)
```

2. (a) **Écrire une fonction en Python d'entête `"def suite_u(n) :"` prenant comme paramètre un entier n et renvoyant la valeur de u_n.**

```
def suite_u(n) :    # paramètre n (pas de input dans une fonction)
    u = 1 ;                 # u0=1
    for i in range(1,n+1) : # même programme
        u = u+1/u
    return(u)               # valeur renvoyé par la fonction
                            # (pas de print dans une fonction)
```

(b) **Quelle commande faut-il écrire pour calculer u_{100} ?**
L'écriture de la fonction précédente ne génère aucun affichage. Si on veut faire un calcul explicite, il faut appeler la fonction avec le paramètre souhaité. Ici, il faut choisir $n = 100$.

```
suite_u(100)  # on n'écrit pas "def" lorsqu'on appelle la fonction
```

3. **Écrire (de deux manières différentes) une fonction en Python prenant comme paramètre un entier n et renvoyant toutes les valeurs de u_0 à u_n rangées dans une liste.**
Pour $n = 4$, ***on obtient*** $[1, 2.0, 2.5, 2.9, 3.2448275862068963]$.

(a) **Une fonction d'entête `"def Liste_U(n)"` utilisant la méthode `"append"`.**

```
def Liste_U(n):
    u = 1              # u0=1
    L = [1]            # initialisation de la liste avec u0
    for k in range(1,n+1) : # n passages
        u = u+1/u          # calcul du terme suivant
        L.append(u)        # on ajoute ce terme en fin de liste
    return(L)          # on renvoie la liste contenant tous les termes
```

(b) **Une fonction d'entête `"def Tableau_U(n)"` utilisant la fonction `"zeros"` du module `"numpy"`.**
Il faut importer le module `"numpy"` pour accéder aux fonctions sur les tableaux.

```
import numpy as np        # on importe le module numpy
```

Ici, `"np"` est juste un "alias" pour ne pas avoir à écrire numpy à chaque fois.
L'algorithme est le même, seule la syntaxe change.

```
def Tableau_U(n) :
    L = np.zeros(n+1)    # création d'une liste contenant (n+1) 0
    L[0] =  1            # on range u0 dans la case 0
    u = 1
    for k in range(1,n+1) : # Attention à la gestion des indices
        # calcul et stockage du suivant : L[k] contient u_k
        u = u + 1/u
        L[k] = u
    return(L)  # on renvoie la liste contenant tous les termes
```


On peut aussi utiliser les valeurs de la suite déjà stockées dans L :

```
def Tableau_U(n):
    L = np.zeros(n+1)    # création d'une liste contenant (n+1) 0
    L[0] =  1            # on range u0 dans la case 0
    for k in range(n) : # Attention à la gestion des indices
        # calcul et stockage du suivant : L[k] contient u_k
        L[k+1] = L[k] + 1/L[k]
    return(L)  # on renvoie la liste contenant tous les termes
```

4. **Écrire un programme en Python qui détermine et affiche le plus petit entier naturel n pour lequel $u_n \geqslant 100$. *On obtient* $n = 4998$.**

```
n = 0                # on retient l'indice (c'est lui qu'on cherche)
u = 1                # premier terme u0
while u<100 :         # tant que u<100
    u = u+1/u         # calcul du terme suivant
    n = n+1          # on augmente l'indice
print(n)             # en sortie de boucle, u>=100
```

5. **Écrire un programme en Python qui calcule et affiche la valeur de la somme $\sum_{n=0}^{100} u_n$. *On obtient*** 971.654.....

(a) **En modifiant le programme de la question 1.**
Le calcul des termes de la suite est le même que pour la question 1 mais il faut ajouter une variable S qui permet de calculer la somme au fur et à mesure (car les termes de la suite sont écrasés à chaque passage dans la boucle).

```
n = 100
u = 1                    # u0=1
S = 1                    # S=u0 au début
for i in range(1,n+1) : # boucle de calcul
    u = u + 1/u ;        # calcul de u1, ..., u100
    S = S + u            # calcul de la somme au fur et à mesure
print(S)
```

(b) **En utilisant la fonction `"Tableau_U"` de la question 3b et la fonction `"sum"` du module `"numpy"`.**
Ici, on va d'abord appelé la fonction `"Tableau_U"` pour construire un tableau contenant tous les termes puis on va utiliser la fonction `"sum"` qui permet d'additionner les termes d'un tableau (ce qui calculera la somme voulue).

```
# le module numpy a déjà été importé !
T = Tableau_U(100) # T contient les termes de la suite de u0 à u100
S = np.sum(T)      # on additionne les éléments de T
print(S)
```

6. **On considère le programme Python et le graphique associé ci-dessous.**

```
import matplotlib.pyplot as plt
import numpy as np
from math import sqrt

u = 1
eq = np.zeros(200)
for i in range(1,200) :
    u = u + 1/u          # calcul de u_i
    eq[i] = u/sqrt(2*i) # clacul et stockage de u_i/sqrt(2*i)
abs = [k for k in range(1,200)]
plt.plot(abs, eq[1:], 'b.')   # tracé du graphe
plt.show()          # affichage du graphe
```

A l'aide du graphique, conjecturer un équivalent de la suite $(u_n)_{n\in\mathbb{N}}$ en $+\infty$.
Ce graphe représente les éléments du tableau `"abs"` en abscisses les éléments du tableau `"eq"` en ordonnées .
Or, `"abs"` contient tous les entiers entre 1 et 199 et `"eq[1:]"` contient tous les termes de la suite $\frac{u_i}{\sqrt{2i}}$ pour i entre 0 et 199 (la commande `"[1:]"` permet de spécifier les indices souhaités ; ici à partir de 1).
La suite de points représentant `"eq[1:]"` semble converger vers 1.
Ainsi, on conjecture que : $\lim\limits_{i\to+\infty} \frac{u_i}{\sqrt{2i}} = 1$ soit $u_i \underset{+\infty}{\sim} \sqrt{2i}$.

Python

Si T est un tableau `"numpy"`, alors la commande `"np.cumsum(T)"` du module `numpy` renvoie un autre tableau `"numpy"` contenant les sommes cumulées du tableau T.
Par exemple : `"np.cumsum([1,4,6])"` renvoie `"[1,5,11]"`.

- EXERCICE 2 -
On considère les suites $(u_n)_{n\in\mathbb{N}^*}$ et $(v_n)_{n\in\mathbb{N}^*}$ suivantes : $u_n = \sum\limits_{k=1}^{n} \frac{1}{k^2}$ **et** $v_n = u_n + \frac{1}{n}$.

1. **On rappelle que la commande `"np.arange(a,b)"` renvoie un tableau `numpy` contenant les entiers de l'intervalle $[\![a; b-1]\!]$ et que les opérations algébriques sur les tableaux `"numpy"` sont appliquées à chaque coefficient.**
   Écrire une fonction Python d'entête `"def U(n) :"` qui renvoie un tableau `"numpy"` contenant les n premiers termes de la suite $(u_n)_{n\in\mathbb{N}^*}$.

```
import numpy as np

def U(n) :
    L = np.arange(1,n+1) # tableau des entiers de 1 à n
    T = 1/L**2           # formule appliquée à chaque coefficient
    return np.cumsum(T)  # calcul des sommes cumulées
```

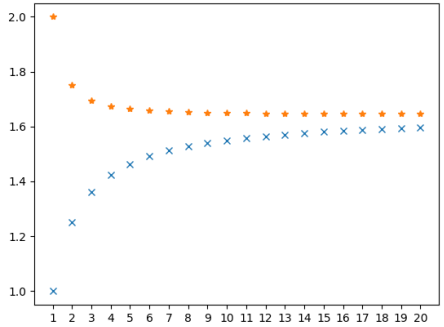
2. **Écrire une fonction Python d'entête `"def V(n) :"` qui renvoie un tableau `"numpy"` contenant les n premiers termes de la suite $(v_n)_{n\in\mathbb{N}^*}$.**

```
def V(n) :
    C = 1/np.arange(1,n+1) # tableau des inverses des entiers de 1 à n
    return U(n)+C          # on récupère u(n) et on ajoute 1/n
```

3. **Écrire un programme Python qui affiche les 20 premiers termes des suites u et v sur un même graphique.**

```
n = 20
Suite_U = U(n)                # tableau des valeurs de u(n)
Suite_V = V(n)                # tableau des valeurs de v(n)
Abs = np.arange(1,n+1)        # tableau des abscisses
plt.plot(Abs, Suite_U, 'x')   # tracé du graphe de u
plt.plot(Abs, Suite_V, '*')   # tracé du graphe de v
plt.show()
```

4. **Le programme de la question précédente renvoie le graphique suivant.**



(a) **Que peut-on conjecturer sur les suites $(u_n)_{n\in\mathbb{N}^*}$ et $(v_n)_{n\in\mathbb{N}^*}$?**
Les suites $(u_n)_{n\in\mathbb{N}^*}$ et $(v_n)_{n\in\mathbb{N}^*}$ semblent être adjacentes.

(b) **En admettant que la conjecture est vraie, écrire un programme Python qui, pour une valeur donnée de $\epsilon > 0$, renvoie une approximation de $\sum\limits_{k=1}^{+\infty} \frac{1}{k^2}$ à ϵ près.**

Les suites $(u_n)_{n\in\mathbb{N}^*}$ et $(v_n)_{n\in\mathbb{N}^*}$ étant adjacentes, elles convergent vers une même limite ℓ.
Ainsi, $\lim\limits_{n\to+\infty} u_n = \sum\limits_{k=1}^{+\infty} \frac{1}{k^2} = \ell$.
D'autre part, comme $u_n \leqslant \ell \leqslant v_n$ et que $\lim\limits_{n\to+\infty} v_n - u_n = 0$, on a : $0 \leqslant \ell - u_n \leqslant v_n - u_n$.
Ainsi, on obtient une valeur approchée de ℓ à ϵ près en renvoyant une valeur de u_n vérifiant :
$0 \leqslant \ell - u_n \leqslant v_n - u_n \leqslant \epsilon$.
Il faut donc calculer les termes successifs des suites u et v jusqu'à ce que $v_n - u_n \leqslant \epsilon$.
On obtient alors le programme suivant :

```
def approx_sum(eps) :
    n, u , v = 1, 1, 2
    while v-u > eps :
        n = n + 1
        u = u + 1/(n**2)
        v = u + 1/n
    return(u)
```

- EXERCICE 3 -

1. **On considère la suite à récurrence linéaire d'ordre 2 (u_n) définie, pour tout $n \in \mathbb{N}$, par :**

$$u_0 = 0,\ u_1 = 2,\ \text{et}\ u_{n+2} = 7u_{n+1} - 3u_n.$$

Écrire une fonction Python d'entête `"def Suite_Rec2_U(n) :"` afin qu'elle renvoie le terme général u_n en fonction de n.
Ici, il est nécessaire de créer une variable auxiliaire pour ne pas écraser les valeurs de la suite dont on a besoin pour calculer les termes suivants. On calcule et stocke le nouveau terme u_{n+2} dans la variable auxiliaire. On peut alors oublier u_n dont on n'a plus besoin. Le terme u_{n+1} prend la place de u_n et le terme u_{n+2} la place de u_{n+1}.

```
def Suite_Rec_U(n) :
    Uold = 0
    Unew = 2
    for i in range(2,n+1) :
        # On calcule le nouveau terme à partir des deux précédents
        Uaux =  7*Unew - 3*Uold
        # On met à jour les termes précédents (décalage)
        Uold = Unew
        Unew = Uaux
    return(Unew)
```

2. **On considère les suites $(u_n)_{n\in\mathbb{N}^*}$ et $(v_n)_{n\in\mathbb{N}^*}$ suivantes :**

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

Écrire une fonction Python d'entête `"def Suites_ab(n) :"` afin qu'elle renvoie les valeurs de a_n et b_n lorsque n est donné en paramètre.
Ici aussi on a besoin d'une variable auxiliaire pour (a_n) sinon ce sera la valeur de a_{n+1} qui sera utilisée pour le calcul de b_{n+1}.

```
def Suites_ab(n) :
    a, b = 1, 2
    for k in range(n) :
        a_aux = a
        a = a_aux**2 / (a_aux + b)
        b = b**2 / (a_aux + b)
    return a,b
```


- EXERCICE 4 -

On considère une suite $(v_n)_{n\in\mathbb{N}^*}$ définie par : $v_1 = 1$ et pour tout entier n non nul, $v_{n+1} = v_n + \frac{1}{n^2 v_n}$.

1. **Écrire une fonction en Python d'entête `"def Suite_V(n) :"` prenant comme paramètre un entier n et renvoyant la valeur de v_n.**
Comme n apparait dans la formule de récurrence, il faut faire attention à la gestion des indices.
Dans la boucle, k désigne l'indice courant. Au premier passage, $k = 1$ et on calcule v_2, puis $k = 2$ et on calcule v_3, etc... Il faut donc arrêter la boucle à $k = n-1$ pour calculer v_n en dernier (d'où le `"range(n)"`).

```
def   Suite_V(n) :
  v = 1                          # v_1 = 1
  for k in range(1,n) :
     v = v + 1/(k**2 * v)     # calcul de v_(k+1)
  return(v)
```

2. **On admet que la suite (v_n) converge vers une limite ℓ vérifiant : pour tout $p \geqslant 2$, $0 \leqslant \ell - v_p \leqslant \frac{1}{p-1}$.**
Écrire un programme Python qui renvoie une valeur approchée de ℓ à 10^{-4} près.

- La relation $0 \leqslant \ell - v_p \leqslant \frac{1}{p-1}$ montre que l'écart entre v_p et ℓ est inférieur à $\frac{1}{p-1}$. Ainsi, il suffit de déterminer une valeur de p pour laquelle $\frac{1}{p-1} \leqslant 10^{-4}$ et de donner la valeur de v_p correspondante.

```
v = 1                  # v_1 = 1
# calcul de v_2 car la relation n'est
# valable que pour p>=2
v = v + 1/v
p = 2
# tant que l'approximation n'est pas assez précise
# on calcule la nouvelle valeur de v
while 1/(p-1) > 10**(-4) :
    v = v + 1/(p**2 * v)     # Calcul  de v_3, v_4, ...
    p = p + 1             # On augmente l'indice
# en sortie de boucle, l'approximation est valable
print(v)
```

- Autre méthode : On détermine à la main la valeur de p qui convient, à savoir :

$$\frac{1}{p-1} \leqslant 10^{-4} \iff p-1 \geqslant 10^4 \iff p \geqslant 10^4 + 1$$

et on calcule avec la fonction de la question précédente :

```
p = 10**4 + 1      # indice calculé à la main
v = Suite_V(p)    # calcul du terme d'indice p de la suite v
print(v)
```

- EXERCICE 5 (DICHOTOMIE) -

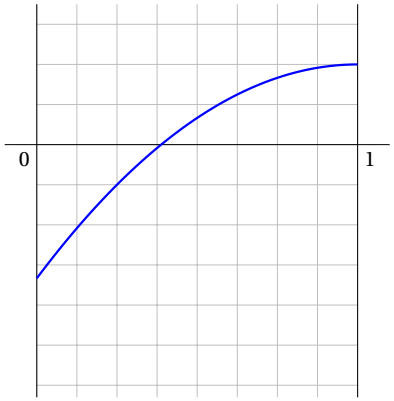
Compléter la fonction suivante pour qu'elle renvoie une valeur approchée à `eps` près de la racine de l'équation $f(x) = 0$ par la méthode de dichotomie.

1. Dans le cas général en utilisant le signe du produit $f(a) \times f(c)$.

```
def Dichotomie(f,eps) :
    # initialisation
    a = 0
    b = 1
    # tant que l'intervalle
    # est trop grand
    while (b-a) > eps  :
        c = (a+b)/2 # milieu de [a,b]
        if f(a)*f(c)>0 :    # a et c sont du même côté
            a = c            # donc a prend la place de c
        else :               # a et c sont de part et d'autre
            b = c       # donc b prend la place de c
    return(c)
```

2. Dans le cas d'une fonction f strictement croissante sur l'intervalle $[0,\ 1]$.

```
def Dichotomie_Croiss(f,eps) :
    # initialisation
    a = 0
    b = 1
    # tant que l'intervalle
    # est trop grand
    while (b-a) > eps  :
        c = (a+b)/2 # milieu de [a,b]
        if f(c)>0 : # côté gauche
            b = c
        else :      # côté droit
            a = c
    return(c)
```



3. Dans le cas d'une fonction f strictement décroissante sur l'intervalle $[0,\ 1]$.

```
def Dichotomie_Decroiss(f,eps) :
    # initialisation
    a = 0
    b = 1
    # tant que l'intervalle
    # est trop grand
    while (b-a) > eps  :
        c = (a+b)/2 # milieu de [a,b]
        if f(c)>0 : # côté droit
            a = c
        else :      # côté gauche
            b = c
    return(c)
```