Tableaux et simulations informatiques

I Tableaux

Il n'existe pas de type « tableau » en python (hormis dans des modules à importer ; on étudiera prochainement les tableaux fournis par le module `numpy`). Mais on peut coder des tableaux à l'aide de listes de listes. *Par exemple :* `[[1,3,8,3,8],[3,1,0,0,0],[8,0,1,0,0],[3,0,0,1,0],[8,0,0,0,1]]` Pour représenter :

1	3	8	3	8
3	1	0	0	0
8	0	1	0	0
3	0	0	1	0
8	0	0	0	1

I.1 Création de tableaux

À l'aide de boucles avec `append` ou en compréhension.

```
def ligneNulle(p):
    L = []
    for j in range(p):
        L.append(0)
    return L

def tableauNul(n,p): #crée un tableau à n lignes et p colonnes rempli de zéros
    T = []
    for i in range(n):
        T.append(ligneNulle(p))
    return T

#ou
def tableauNul(n,p):
   return [[0 for j in range(p)] for i in range(n)]
```

I.2 Modification d'un tableau déjà créé

- `T[i][j]=x # remplace l'élément de la ligne i et la colonne j de T par x`
- `T[i]=L # remplace la ligne i par L (ligne)`

Pour créer un tableau, classiquement on commence par créer un tableau de la taille voulue avec des coefficients quelconques (par exemple des zéros), puis on modifie les coefficients un par un, ou par groupe, pour obtenir les coefficients voulus.

II Les listes (ou tableaux) comme paramètres de fonctions

Une bonne pratique d'écriture est de **ne pas modifier à l'intérieur d'une fonction la valeur des paramètres de cette fonction :**

```
#ne pas écrire :
def f(x):
    x = x + 1 #écrasement de la valeur de x donnée en paramètre !
    return x

#mais plutôt :
def g(x):
    y = x + 1
    return y
```

Dans le cas d'un paramètre de type simple (entier, flottant, booléen, caractère) la variable ainsi modifiée ne l'est qu'à l'intérieur du déroulement de la fonction ; ainsi :

```
a = 33
print(f(a)) #affiche 34
print(a) #affiche 33
```

Une exception importante à ce principe est lorsque le paramètre est d'un type composé, notamment dans le cas d'une liste de grande taille, que l'on souhaite modifier dans le cadre d'une fonction sans faire de copie (pour des raisons de taille mémoire, par exemple) ; modifier une grande structure de données sans faire de copie s'appelle travailler **en place**.
Par exemple :

```
def f(L,x,y): #procédure qui ajoute x à la fin de la liste L et modifie son premier élément pour que ce soit y
    L.append(x)
    L[0] = y
    #pas de return

L = [1,2,3]
f(L,33,34)
print(L) #affiche [34,2,3,33]
```

On remarquera que, dans ce cas, on a utilisé une procédure (fonction qui ne renvoie "rien" du point de vue utilisateur, mais "None" du point de vue informatique).

```
>>> type(f(L,33,34))
<class 'NoneType'>
>>> print(f(L,33,34))
None
```

Si l'on voulait conserver la liste `L` de départ, il aurait plutôt fallu écrire :

```
def f(L,x,y): #fonction qui ajoute x à la fin de la liste L modifie son premier élément pour que ce soit y sans modifier L
    M = [y]
    for i in range(1,len(L)):
        M.append(L[i])
    M.append(x)
    return M

L = [1,2,3]
print(f(L,33,34)) #affiche [34,2,3,33]
print(L) #affiche [1 2 3]
```

Attention : « `M = L` » n'effectue pas de copie de la liste `L`.

III Vocabulaire important des énoncés

mot de l'énoncé	*commentaire*
renvoie	il faut un `return` dans la fonction
affiche	il faut un `print` ou un `plot` dans la fonction (pas de `return`)
modifie	calcul en place ; procédure (sans `return`)

IV Principes de la modélisation en temps discret

L'informatique permet d'étudier le comportement de systèmes complexes lors d'une succession d'instants numérotés par un entier à condition de savoir décrire le passage d'un instant au suivant.

Le schéma général d'une telle modélisation est le suivant :

1. **Initialisation** : création de la structure de données qui va représenter la situation à un instant précis.
2. **Règle d'évolution :** calcul de la situation à l'instant $n+1$ en fonction de la situation à l'instant n.
3. **Simulation :** boucle qui permet d'effectuer la totalité de la simulation. Il y a plusieurs possibilités, notamment :
   - boucle `for` si le nombre d'instants à simuler est connu à l'avance ;
   - boucle `while` si l'on veut effectuer la simulation jusqu'à ce le système présente une caractéristique donnée, ou jusqu'à ce qu'il n'évolue plus, ou peu ;
   - boucle `for` interrompue ou boucle `while` si l'on est dans le cas précédent, mais l'on n'est pas sûr que la simulation va se terminer.

   ⇝ lorsqu'il est complexe, le test d'arrêt de la boucle peut faire l'objet d'une fonction auxiliaire.
4. *Éventuellement* **mise en forme du résultat**, par exemple d'une des manières suivantes :
   - renvoi (`return`) de tout ou d'une partie de la structure de données obtenue à la fin, ou du nombre d'itérations effectuées ;
   - affichage (`print` ou `plot`) de tout ou d'une partie de la structure de données obtenue à la fin.

La **règle d'évolution** peut être écrite de deux manières :

- soit avec une **fonction** qui **crée et calcule** la situation `TT` à l'instant $n+1$ à partir de la situation `T` à l'instant n, sans modifier `T` et qui **renvoie** `TT`
- soit avec une **procédure** qui **modifie** la situation `T` à l'instant n pour qu'elle devienne la situation `T` à l'instant $n+1$, et qui ne renvoie rien.
  Cette deuxième option n'est possible que si les calculs pour faire évoluer la valeur d'un élément de `T` de l'instant n à l'instant $n+1$ ne nécessitent pas d'utiliser les valeurs de `T` à l'instant n déjà modifiées dans le processus global de passage de n à $n+1$.

```python
def suivant(L):
    M = nouveauL(...)  #initialisation du résultat à calculer
    for i in range(...):
        for j in range(...)
            M[i][j] = ...
    return M

def simulation(...):
   L = initialisation(...)
   for k in range(n): #nombre de fois que l'on veut faire la simulation
       L = suivant(L) # #ou, pour S initialisée, S[k]=suivant(L); L = suivant(L) si l'on souhaite conserver les états intermédiaires de la simulation
   return L

#ou bien (avec une procédure) :
def suivant(L):
    for i in range(...):
        for j in range(...)
            L[i][j] = ...  #modification de L en place

def simulation(...):
   L = initialisation(...)
   for k in range(n): #nombre de fois que l'on veut faire la simulation
       suivant(L)
   return L
```

V Exercices

Q1 Qu'affiche le programme suivant ?

```python
def f(l):
    l.append(2)
    return l

>>> t = [0]
>>> print(f(t),t)
```

Q2 Écrire une procédure `suivant(T,i)` qui prend en argument le tableau de nombres `T` carré à `n` lignes et `n` colonnes et un entier `i` qui appartient à $[\![0,n[\![$. Cette procédure modifie en place la ligne `i` du tableau `T` pour que, sur cette ligne l'élément `0` et l'élément `i` valent 1 et que, entre les deux, chaque élément soit la somme de l'élément situé sur la ligne du dessus et du voisin de gauche de ce dernier ; les autres éléments de `T` ne sont pas modifiés.
Par exemple : si `T` vaut `[[1,0,0,0],[1,1,0,0],[0,0,0,0],[0,0,0,0]]`, alors `suivant(T,2)` le modifie en `[[1,0,0,0],[1,1,0,0],[1,2,1,0],[0,0,0,0]]`.

Q3 Utiliser la fonction précédente pour écrire une fonction `Pascal(n)` qui renvoie le triangle de Pascal contenant les coefficients binomiaux jusqu'à l'ordre `n-1`.
Par exemple : `Pascal(4)` → `[[1,0,0,0],[1,1,0,0],[1,2,1,0],[1,3,3,1]]`

VI Problème : Simulation informatique de la propagation d'une épidémie

On pourra utiliser les fonctions suivantes du module `random` que l'on supposera avoir importées :

```
from random import randrange, random
randrange(n) #renvoie un entier choisi aléatoirement entre 0 et n-1 (càd. n exclu)
random() # renvoie un flottant choisi aléatoirement dans l'intervalle [0,1[
```

On cherche à modéliser la propagation d'une épidémie par une méthode de simulation informatique.
Dans ce qui suit, n est un entier strictement positif. Une *grille* `G` de taille n est une liste de n listes de longueur n (c'est-à-dire un tableau à deux dimensions à n lignes et n colonnes). Chaque case de la grille représente un individu de la population qui peut être dans un des quatre états suivants : sain, infecté, rétabli, décédé. On choisit de représenter ces quatre états par les entiers :

0 (sain), 1 (infecté), 2 (rétabli), 3 (décédé)

Par exemple, `G[i][j]` vaut `0` si et seulement si l'individu représenté par la case de la ligne `i` et la colonne `j` est sain.

L'état d'une case d'une grille évolue au cours du temps selon les règles suivantes : l'état d'une case à l'instant $t+1$ ne dépend que de son état à l'instant t et de l'état de ses huit cases voisines à l'instant t (une case du bord n'a que cinq cases voisines et trois pour une case d'un coin).

Les *règles d'évolution* sont les suivantes :

- Une case décédée à l'instant t reste décédée à l'instant $t+1$.
- une case infectée à l'instant t devient décédée à l'instant $t+1$ avec une probabilité p_1 ou rétablie avec une probabilité $1-p_1$.
- une case rétablie à l'instant t reste rétablie à l'instant $t+1$.
- une case saine à l'instant t devient infectée à l'instant $t+1$ avec une probabilité p_2 si elle a au moins une case voisine infectée et elle reste saine sinon.

Q4 Écrire une fonction `grille(n)` qui renvoie une grille `G` de taille $n \times n$ dont toutes les cases sont saines.
Par exemple : `grille(3)` → `[[0,0,0],[0,0,0],[0,0,0]]`

On initialise toutes les cases dans l'état sain, sauf une case choisie au hasard dans l'état infecté.

Q5 Écrire en Python une fonction `init(n)` qui renvoie une grille `G` de taille $n \times n$ ne contenant que des cases saines sauf exactement une case infectée choisie aléatoirement (on utilisera la fonction `grille` comme fonction auxilliaire).

Q6 Écrire en Python une fonction `nombre_de(G,x)` qui a pour arguments une grille `G` et un entier `x` qui renvoie le nombre de cases de la grille `G` dont l'état est égal à l'entier `x`.

Q7 Écrire en Python une fonction `compte(G)` qui a pour argument une grille `G` et renvoie la liste `[n0,n1,n2,n3]` où `n0` est le nombre de cases de la grille `G` dans l'état 0, `n1` est le nombre de cases de la grille `G` dans l'état 1, `n2` est le nombre de cases de la grille `G` dans l'état 2, `n3` est le nombre de cases de la grille `G` dans l'état 3.

D'après les règles d'évolution, pour savoir si une case saine peut devenir infectée à l'instant suivant, il faut déterminer si elle est exposée à la maladie, c'est-à dire si elle possède au moins une case infectée dans son voisinage. Pour cela, on écrit en Python la fonction `est_exposee(G,i,j)` suivante :

```
1  def est_exposee(G,i,j):
2      n=len(G)
3      if i==0 and j==0:
4          return (G[0][1]-1)*(G[1][1]-1)*(G[1][0]-1)==0
5      elif i==0 and j==n-1:
6          return (G[0][n-2]-1)*(G[1][n-2]-1)*(G[1][n-1]-1)==0
7      elif i==n-1 and j==0:
8          return (G[n-1][1]-1)*(G[n-2][1]-1)*(G[n-2][0]-1)==0
9      elif i==n-1 and j==n-1:
10         return (G[n-1][n-2]-1)*(G[n-2][n-2]-1)*(G[n-2][n-1]-1)==0
11     elif i==0:
12         #à compléter
13     elif i==n-1:
14         return (G[n-1][j-1]-1)*(G[n-2][j-1]-1)*(G[n-2][j]-1)*(G[n-2][j+1]-1)
       *(G[n-1][j+1]-1)==0
15     elif j==0:
16         return (G[i-1][1]-1)*(G[i-1][1]-1)*(G[i][1]-1)*(G[i+1][1]-1)*(G[i
       +1][0]-1)==0
17     elif j==n-1:
18         return (G[i-1][n-1]-1)*(G[i--1][n-2]-1)*(G[i][n-2]-1)*(G[i+1][n-2]-1)
       *(G[i+1][n-1]-1)==0
19     else:
20         #à compléter
```

Q8 Quel est le type du résultat renvoyé par la fonction `est_exposee` ?

Q9 Compléter les lignes 12 et 20 « `#à compléter` » de la fonction `est_exposee` (on ne donnera sur sa copie que les lignes en question).

Q10 Écrire une fonction `bernouilli(p)` qui renvoie 1 avec la probabilité `p` et 0 avec la probabilité `1-p`.

Q11 Écrire une fonction `suivant(G,p1,p2)` qui fait évoluer toutes les cases de la grille `G` à l'aide des règles d'évolution et renvoie une nouvelle grille correspondant à l'instant suivant. Les arguments `p1` et `p2` sont les probabilités qui interviennent dans les règles d'évolution pour les cases infectées et les cases saines.

Q12 Écrire en Python une fonction `simulation(n,p1,p2)` qui réalise une simulation complète avec une grille de taille $n \times n$ pour les probabilités `p1` et `p2`, et renvoie la liste `[x0,x1,x2,x3]` formée des proportions de cases dans chacun des quatre états à la fin de la simulation (une simulation s'arrête lorsque la grille ne comporte plus aucune case infectée).

Q13 Écrire une fonction `calculSeuil(N,n,nb,p1,seuil)` qui pour la valeur p_1 donnée et pour chacune des `N+1` valeurs $\frac{0}{N}, \frac{1}{N}, \frac{2}{N}, \ldots \frac{N-1}{N}, \frac{N}{N}$ de p_2 calcule la moyenne sur `nb` simulations de la proportion de personnes infectées au cours de l'épidémie (décédées ou rétablies), affiche le graphique de la proportion de personnes infectées en fonction de p_2 et détermine par dichotomie la valeur de p_2 à partir de laquelle cette proportion dépasse le pourcentage `seuil` — on supposera que les moyennes ainsi obtenues croissent strictement quand p_2 augmente.

VII Complexité

Q14 Quelle est la complexité de `v` ?

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

def v(p):
    s = 0
    for i in range(p+1):
        s = s + u(i)
    return s

```

Q15 On donne les fonctions suivantes :

```
def max_somme(T:list, i:int)->int:
    n = len(T)
    M,S = T[i],T[i]
    for k in range(i+1,n):
        S += T[k]
        if S > M:
            M = S
    return M

def somme_maxi2(T:list)->int:
    n = len(T)
    M = T[-1]
    for i in range(n-1):
        S = max_somme(T,i)
        if S > M:
            M = S
    return M
```

Que font-elles et quelle est leur complexité (dans le pire des cas) en fonction de la taille n de la liste de nombres `T` ?

Q16 On donne les fonctions suivantes :

```
def somme(T:list, i:int, j:int)->int:
    S = 0
    for k in range(i,j):
        S += T[k]
    return S

def somme_maxi1(T:list)->int:
    n = len(T)
    M = T[-1]
    S = 0
    for i in range(n-1):
        for j in range(i+1,n):
            S = somme(T,i,j)
            if S > M:
                M = S
    return M
```

Que font-elles et quelle est leur complexité (dans le pire des cas) en fonction de la taille n de la liste de nombres `T` ?

Corrigé

Q1

```
def suivant(T,i):
    T[i][0],T[i][i] = 1,1
    for k in range(1,i):
        T[i][k] = T[i-1][k-1] + T[i-1][k]
```

Q2

```
def Pascal(n):
    T = [[0 for j in range(n)] for i in range(n)]
    for i in range(n):
        suivant(T,i)
    return T
```

Q3

```
def ligneZeros(n):
    L = []
    for j in range(n):
        L.append(0)
    return L

def grille(n):
    G = []
    for i in range(n):
        G.append(ligneZeros(n))
    return G

#ou
def grille2(n):
    return [[0 for j in range(n)] for i in range(n)]
```

Q4

```
def init(n):
    G = grille(n)
    G[randrange(n)][randrange(n)] = 1
    return G
```

Q5

```
def nombre_de(G,x):
    n=len(G)
    nombrex = 0
    for i in range(n):
        for j in range(n):
            if G[i][j] == x:
                nombrex += 1
    return nombrex
```

Q6

```
def compte(G): #ou de complexité O(n**2)
    n=len(G)
    n0,n1,n2,n3 = 0,0,0,0
    for i in range(n):
        for j in range(n):
            if G[i][j] == 0:
                n0 += 1
            elif G[i][j] == 1:
                n1 += 1
```

```
            elif G[i][j] == 2:
                n2 += 1
            elif G[i][j] == 3:
                n3 += 1
    return [n0,n1,n2,n3]

#ou de complexité O(4n**2) càd. O(n**2)
def compte2(G):
    L=[]
    for i in range(4):
        x=nombre(G,i)
        L.append(x)
    return L
```

Q7 La fonction `est_exposee` renvoie un booléen (`True` ou `False`).

Q8 Un produit est nul lorsque l'un des facteurs est nul, donc `(G[i][j]-1)*...*(G[ii][jj]-1)==0` vaut `True` dès que l'une des cases concernées vaut 0 (ie. est infectée).
Le nombre de cases voisines à tester dépend de l'emplacement de la case par rapport aux bords. Dans la fontion proposée, les quatre premiers cas concernent les bords ; les deux cas suivants les bords supérieur et inférieur ; les deux suivants les bords gauche et droit ; et le dernier cas est le cas général des cases qui ne sont pas au bord.

```
def est_exposee(G,i,j):
    n=len(G)
    if i==0 and j==0:
        return (G[0][1]-1)*(G[1][1]-1)*(G[1][0]-1)==0
    elif i==0 and j==n-1:
        return (G[0][n-2]-1)*(G[1][n-2]-1)*(G[1][n-1]-1)==0
    elif i==n-1 and j==0:
        return (G[n-1][1]-1)*(G[n-2][1]-1)*(G[n-2][0]-1)==0
    elif i==n-1 and j==n-1:
        return (G[n-1][n-2]-1)*(G[n-2][n-2]-1)*(G[n-2][n-1]-1)==0
    elif i==0:
#ligne 12 :
        return (G[0][j-1]-1)*(G[1][j-1]-1)*(G[1][j]-1)*(G[1][j+1]-1)*(G
    [0][j+1]-1)==0
    elif i==n-1:
        return (G[n-1][j-1]-1)*(G[n-2][j-1]-1)*(G[n-2][j]-1)*(G[n-2][j
    +1]-1)*(G[n-1][j+1]-1)==0
    elif j==0:
        return (G[i-1][1]-1)*(G[i-1][1]-1)*(G[i][1]-1)*(G[i+1][1]-1)*(G[
    i+1][0]-1)==0
    elif j==n-1:
        return (G[i-1][n-1]-1)*(G[i--1][n-2]-1)*(G[i][n-2]-1)*(G[i+1][n
    -2]-1)*(G[i+1][n-1]-1)==0
    else:
#ligne 20 :
        return (G[i-1][j-1]-1)*(G[i-1][j]-1)*(G[i-1][j+1]-1)*(G[i][j
    -1]-1)*(G[i][j+1]-1)*(G[i+1][j-1]-1)*(G[i+1][j]-1)*(G[i+1][j+1]-1)==0
```

Q9 La probabilité que le résultat de `random()` appartienne à un intervalle $[a, b[$ (inclus dans $[0, 1[$) vaut $b - a$. On peut donc écrire :

```
def bernoulli(p):
    x = random()
    if x < p:
        return 1
    return 0
```

Q10

```
def suivant(G,p1,p2):
    n = len(G); H = grille(n)
    for i in range(n):
        for j in range(n):
            #règle 1 sur les décédés
            if G[i][j] == 3: H[i][j] = 3
            #règle 2 sur les infectés décédés ou rétablis
            if G[i][j] == 1:
                if bernoulli(p1) == 1:
                    H[i][j] = 3
                else:
                    H[i][j] = 2
            #règle 3 sur les rétablis
            if G[i][j] == 2: H[i][j] = 2
            #règle 4 sur les saines infectés ou non
            if G[i][j] == 0:
                if est_exposee(G,i,j) and bernoulli(p2) == 1:
                    H[i][j] = 1
                else:
                    H[i][j] = 0
    return H
```

Q11

```
def simulation(n,p1,p2):
    G = init(n); n1 = 1
    while n1 != 0:
        G = suivant(G,p1,p2)
        n0,n1,n2,n3 = compte(G)
        x0,x1,x2,x3 = n0/n**2,n1/n**2,n2/n**2,n3/n**2
    return [x0,x1,x2,x3]
```

Q12

```
import matplotlib.pyplot as plt

def calculSeuil(n,N,nb,p1,seuil):
    #liste des p2
    liste_p2 = [i/N for i in range(N+1)]
    #liste de la proportion moyenne des infectés pour chaque p2
    liste_moyenne1 = []
    for p2 in liste_p2:
        moyenne1 = 0
        for i in range(nb):
            x0,x1,x2,x3 = simulation(n,p1,p2)
            moyenne1 += x2 + x3
        moyenne1 /= nb
        liste_moyenne1.append(moyenne1)
    #valeur au dessus du seuil par dichotomie sur liste_moyenne1
    debut,fin = 0,N
    while fin-debut > 1:
        milieu = (debut+fin)//2
        if liste_moyenne1[milieu] < seuil:
            debut = milieu
        if liste_moyenne1[milieu] > seuil:
            fin = milieu
    p2 = fin / N #fin-debut==1 et la valeur demandée est fin/N
    #affichage de la courbe (p2,proportion_infectes)
    plt.plot(liste_p2,liste_moyenne1)
    plt.plot([0,p2,p2],[seuil,seuil,0])
    plt.show()
```

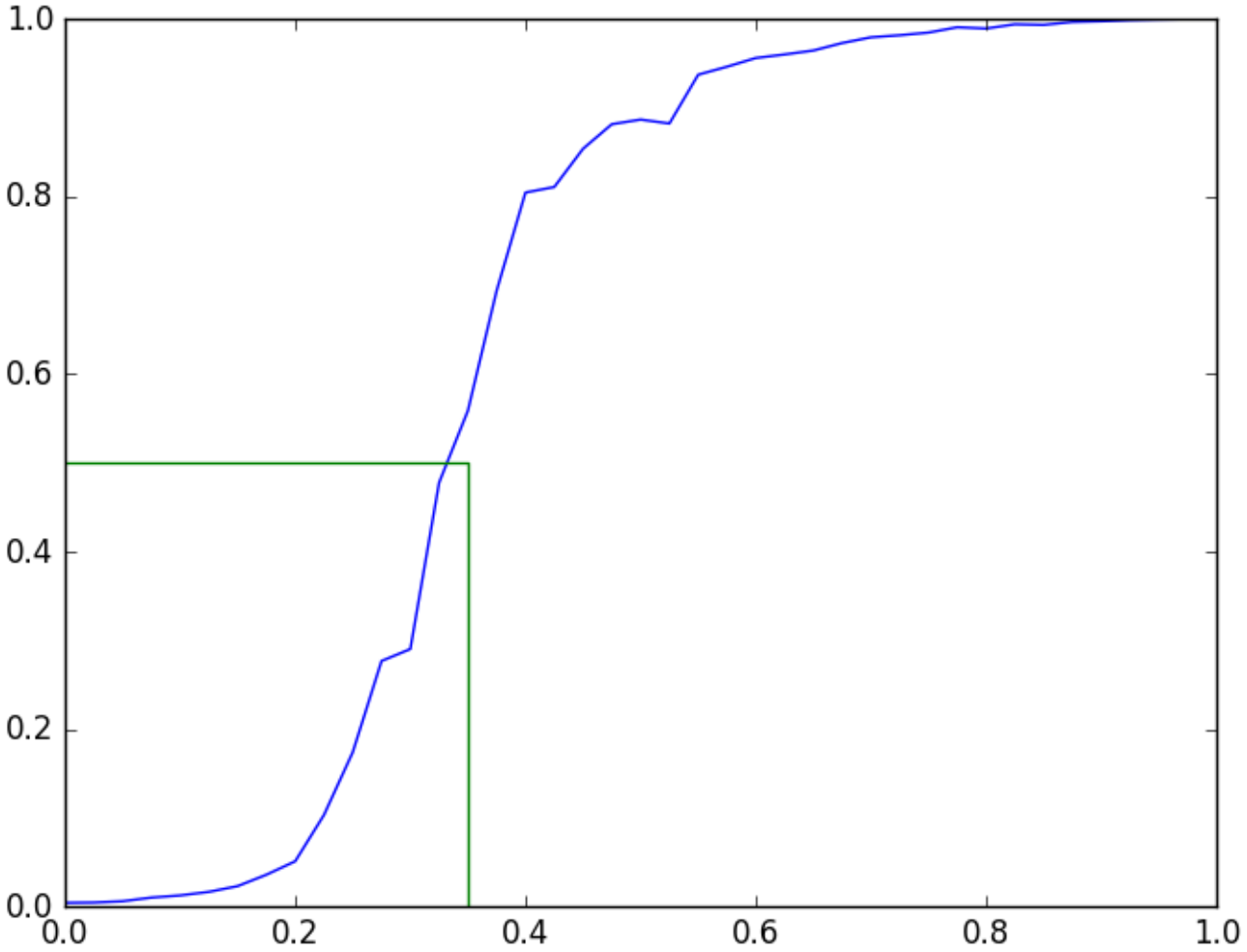

FIGURE 1 – calculSeuil(15,40,40,0.1,0.5)