

Corrigé de l'épreuve d'Informatique CNC 2026

Exercice Préliminaire (4 points)

Q.1 – Fonction existe(L, x)

```
1 def existe(L, x):  
2     for e in L:  
3         if e == x:  
4             return True  
5     return False
```

Ou tout simplement

```
1 def existe(L, x):  
2     return x in L
```

Q.2 – Complexité de existe

Dans le pire des cas, la fonction parcourt toute la liste (les n éléments de L ($n=\text{len}(L)$)) en faisant appel à des instructions élémentaires (c'est à dire des instructions en $\mathcal{O}(1)$).
La fonction existe(L,x) est donc de **complexité linéaire** $\mathcal{O}(n)$.

Q.3 – Fonction communs(A, B)

```
1 def communs(A, B):  
2     C = []  
3     for x in A:  
4         if existe(B, x) and not existe(C, x):  
5             C.append(x)  
6     return C
```

Q.4 – Fonction grand(L)

```
1 def grand(L):  
2     m = L[0]  
3     n=len(L)  
4     for i in range(1,n):  
5         if L[i] > m:  
6             m = L[i]  
7     return m
```

Q.5 – Fonction diviseurs(x)

```
1 def diviseurs(x):  
2     d = []  
3     for i in range(1, x+1):  
4         if x % i == 0:  
5             d.append(i)  
6     return d
```

Q.6 – Fonction premier(x)

```

1 def premier(x):
2     if x < 2:
3         return False
4     d=diviseurs(x)
5     if len(d) != 2 :
6         return False
7     else:
8         return True

```

Ou tout simplement

```

1 def premier(x):
2     return len(diviseurs(x))==2

```

Q.7 – Fonction pgcd(x, y)

```

1 def pgcd(x, y):
2     Lx=diviseurs(x)
3     Ly=diviseurs(y)
4     L=communs(Lx,Ly)
5     return grand(L)

```

Ou tout simplement

```

1 def pgcd(x, y):
2     return grand(communs(diviseurs(x),diviseurs(y)))

```

Partie I – Base de données et langage SQL

Q.8 – Ajout du type 'Mangrove'

```

1 INSERT INTO Types
2 VALUES (45, 'Mangrove');

```

Q.9 – Trois plus grandes superficies

```

1 SELECT codP, superficie
2 FROM Forêts
3 ORDER BY superficie DESC
4 LIMIT 3;

```

Q.10 – Statistiques sur les superficies

```

1 SELECT COUNT(*), MIN(superficie), MAX(superficie), AVG(superficie)
2 FROM Forêts;

```

Q.11 – Algèbre relationnelle (type Boréale)
$$\pi_{\text{nomPays}}(\text{Pays} \bowtie (\sigma_{\text{type}='Boréale'}(\text{Types} \bowtie \text{Forêts})))$$
Q.12 – SQL (type Boréale, tri alphabétique)

```

1 SELECT DISTINCT P.nomPays
2 FROM Pays P
3 JOIN Forêts F ON P.codPays = F.codP
4 JOIN Types T ON F.idTyp = T.id
5 WHERE T.type = 'Boréale'
6 ORDER BY P.nomPays ASC;

```

ou

```

1 SELECT DISTINCT P.nomPays
2 FROM Pays P, Forêts F, Types T
3 WHERE P.codPays = F.codP and F.idTyp = T.id and T.type = 'Boréale'
4 ORDER BY P.nomPays ASC;

```

Q.13 – Superficie totale par type ($> 5M \text{ km}^2$)

```

1 SELECT T.type, SUM(F.superficie) AS total
2 FROM Types T
3 JOIN Forets F ON T.id = F.idTyp
4 GROUP BY T.id
5 HAVING total > 5000000
6 ORDER BY total DESC;

```

ou

```

1 SELECT T.type, SUM(F.superficie) AS total
2 FROM Types T, Forets F
3 Where T.id = F.idType
4 GROUP BY T.id
5 HAVING total > 5000000
6 ORDER BY total DESC;

```

Q.14 – Pourcentage par pays (forêt Tropicale)

```

1 SELECT F.codP,
2         (F.superficie * 100.0 / (SELECT SUM(superficie)
3                                 FROM Forets
4                                 WHERE idTyp IN (SELECT id FROM Types
5                                                 WHERE type = 'Tropicale')) AS
6         pourcentage
7 FROM Forets F
8 WHERE F.idTyp IN (SELECT id FROM Types WHERE type = 'Tropicale')
9 ORDER BY pourcentage ASC;

```

Partie II – Modélisation d'un feu de forêt (Automate cellulaire)

Q.15 – Génération d'une matrice nulle

```

1 def matrice_zeros(n,p):
2     M=[]
3     for i in range(n):
4         L=[]
5         for j in range(p):
6             L.append(0)
7         M.append(L)
8     return M

```

Ou

```

1 def matrice_zeros(n,p):
2     return [ [0] * p for i in range(n) ]

```

Q16. – Génération d'une matrice nulle

```

1 def indices(R):
2     n = len(R)          # Nombre de lignes de R
3     p = len(R[0])       # Nombre de colonnes de R
4     liste_indices = []
5     for i in range(n):
6         for j in range(p):
7             liste_indices.append((i, j))
8     return liste_indices

```

Ou

```

1 def indices(R):
2     return [(i,j) for i in range(len(R)) for j in range(len(R[0]))]

```

Q.17 – Génération aléatoire d'une forêt

```

1 def foret_aleatoire(n, p, d):
2     R = matrice_zeros(n, p)
3     N = int((d * n * p) / 100) # Nombre total d'arbres en fonction de
4     la densité
5     c = 0 # Compteur des arbres placés
6     while c < N:
7         i = randint(0, n - 1)
8         j = randint(0, p - 1)
9         # Vérifier si la case (i,j) est libre (égale à 0)
10        if R[i][j] == 0:
11            R[i][j] = 1          # Placer l'arbre
12            c = c + 1
13    return R

```

Q.18 – Voisins d'une cellule

```

1 def voisines(R, i, j):
2     n = len(R)
3     p = len(R[0])
4     V = []
5     # Directions potentielles : (di, dj)
6     directions = [(-1, 0), (1, 0), (0, 1), (0, -1)]
7     for di, dj in directions:
8         vi = i + di
9         vj = j + dj
10        # Vérification si le voisin est bien dans les limites de la
11        matrice
12        if 0 <= vi < n and 0 <= vj < p:
13            V.append((vi, vj))
14    return V

```

Q.19 – mise en feu d'un arbre au hasard

```

1 def miseEnFeu(R):
2     n = len(R)
3     p = len(R[0])
4     arbres = []
5
6     # Trouver les coordonnées de tous les arbres vivants (valeur 1)
7     for i in range(n):
8         for j in range(p):
9             if R[i][j] == 1:
10                 arbres.append((i, j))
11
12     # Si la forêt contient au moins un arbre, on en enflamme un au
13     # hasard
14     if arbres != []:
15         i = randint(0, len(arbres)-1)
16         iFeu, jFeu = arbres[i]
17         R[iFeu][jFeu] = 2
18
19     return R

```

Q.20 – Evolution d'un forêt

Cette fonction applique les règles d'évolution de l'automate cellulaire en une seule étape temporelle (un pas de temps).

Attention cruciale : Il faut créer une copie de la matrice (ou une nouvelle matrice **Rsuivante**) pour lire l'état actuel et écrire le nouvel état simultanément, sinon les modifications en chaîne faussent le voisinage des cellules suivantes

```

1 def evolution(R):
2     n = len(R)
3     p = len(R[0])
4     # Création d'une nouvelle matrice pour stocker l'état suivant
5     Rsuivante = [[R[i][j] for j in range(p)] for i in range(n)]
6
7     for i in range(n):
8         for j in range(p):
9             # Règle 1 : Un arbre en feu (2) devient un arbre en cendre
10            # (3)
11            if R[i][j] == 2:
12                Rsuivante[i][j] = 3
13
14            # Règle 2 : Un arbre vivant (1) prend feu s'il a au moins
15            # un voisin en feu (2)
16            elif R[i][j] == 1:
17                # On regarde la liste des coordonnées des voisins
18                for vi, vj in voisines(R, i, j):
19                    if R[vi][vj] == 2:
20                        Rsuivante[i][j] = 2
21                        break # Inutile de vérifier les autres voisins
22                            si l'arbre est déjà en feu

```

[return](#) Rsuivante

Q.21 – Fonction propagation(R)

Cette fonction gère le cycle de l'incendie du **début** à la **fin**.
 Elle commence par appeler **miseEnFeu(R)** pour allumer l'incendie, puis applique la fonction **evolution(R)** de manière répétée.
 La boucle s'arrête dès qu'il n'y a plus aucun arbre en feu (2) dans la forêt.

```

1 def propagation(R):
2     R = miseEnFeu(R)
3     continuer = True
4     while continuer:
5         continuer = False #variable qui indique si on doit continuer ou
                             arrêter
6         # On vérifie s'il reste au moins un '2' dans la matrice
7         for i in range(len(R)):
8             for j in range(len(R[0])):
9                 if R[i][j] == 2:
10                    continuer = True
11         # Si un arbre en feu a été trouvé, on calcule le pas de temps
            suivant
12         if continuer:
13             R = evolution(R)
14     return R

```

Q.22 – Fonction taux_brule(R)

Le taux d'arbres brûlés (les arbres en cendre représentés par le nombre 3) est le pourcentage de ces cendres par rapport au nombre total d'arbres initialement présents.

N.B : Le nombre total d'arbres correspond aux arbres vivants restant (1) + les arbres brûlés (3).
 Les sols nus (0) ne comptent pas comme des arbres.

```

1 def taux_brule(R):
2     nbCendres = 0
3     nbTotal = 0
4
5     for i in range(len(R)):
6         for j in range(len(R[0])):
7             if R[i][j] == 3:
8                 nbCendres += 1
9                 nbTotal += 1
10            elif R[i][j] == 1:
11                nbTotal += 1
12
13    if nbTotal == 0:
14        return 0
15
16    return (nbCendres / nbTotal) * 100

```

Q.23 – Fonction percolation(n, p)

Cette fonction doit générer une forêt pour chaque niveau de densité $d \in 40, 42, 44, \dots, 90$, simuler l'incendie complet, calculer la proportion finale de destruction, et tracer la courbe avec matplotlib.

```

1 def percolation(n, p):
2     Ld = [] #Liste des densités (abscisses)
3     Ltaux = [] #Liste des taux (ordonnées)
4     # Parcours des densités de 40 à 90 inclus, avec un pas de 2
5     for d in range(40, 92, 2):
6         # Générer une forêt aléatoire de densité d
7         R1 = foret_aleatoire(n, p, d)
8         # Lancer la propagation complète de l'incendie
9         R2 = propagation(R1)
10        # Calculer le taux d'arbres brûlés
11        taux = taux_brule(R2)
12        Ld.append(d)
13        Ltaux.append(taux)
14    # Tracé de la courbe à l'aide du module matplotlib.pyplot
15    plt.plot(Ld, Ltaux)
16    plt.grid()
17    plt.show()

```

Partie III : Détection d'objets dans une image

Histogramme des gradients orientés (HOG)

Q.24 – Fonction som_prods(A, B)

Cette fonction prend deux matrices carrées de dimension 3×3 et effectue la somme du produit terme à terme (analogue à un produit scalaire aplati ou une convolution élémentaire).

```

1 def som_prods(A, B):
2     S = 0
3     for i in range(3):
4         for j in range(3):
5             S = S + A[i][j] * B[i][j]
6     return S

```

Q.25 – Fonction voisins(M, i, j)

```

1 def voisins(M, i, j):
2     sM = []
3     # On parcourt les lignes de i-1 à i+1 et les colonnes de j-1 à j+1
4     for r in range(i - 1, i + 2):
5         Lv = []
6         for c in range(j - 1, j + 2):
7             Lv.append(M[r][c])
8         sM.append(Lv)
9     return sM

```

Q.26 – Fonction gradH(M)

Cette fonction calcule le gradient horizontal à l'aide du filtre de Sobel donné dans l'énoncé. Les pixels situés sur la bordure extérieure de la matrice conservent leur valeur d'origine. Pour les autres, on applique le produit avec le filtre de Sobel horizontal sur leur voisinage 3×3 .

```

1 def gradH(M):
2     n = len(M)
3     p = len(M[0])
4     # Création de la matrice X de même dimension que M, initialisée
      avec des zéros
5     X = [[0 for i in range(p)] for j in range(n)]
6     # Filtre horizontal de Sobel (matrice 3x3)
7     sobelH = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]
8     for i in range(n):
9         for j in range(p):
10            if i == 0 or i == n - 1 or j == 0 or j == p - 1:
11                X[i][j] = M[i][j]
12            else:
13                # Récupération de la sous-matrice 3 x 3 des voisins
14                V = voisins(M, i, j)
15                # Calcul de la valeur via la fonction définie à la
                  question 24
16                X[i][j] = som_prods(V, sobelH)
17     return X

```

Q.27 - Fonction gradV(M)

Cette fonction calcule le gradient vertical à l'aide de la matrice de Sobel verticale fournie dans l'énoncé. Les pixels de la bordure restent inchangés.

```

1 def gradV(M):
2     n = len(M)
3     p = len(M[0])
4     # Initialisation de la matrice Y de même dimension que M
5     Y = [[0 for i in range(p)] for j in range(n)]
6     # Filtre vertical de Sobel (matrice 3x3)
7     sobelV = [[-1, -2, -1], [0, 0, 0], [1, 2, 1]]
8
9     for i in range(n):
10        for j in range(p):
11            # Traitement des conditions de bord
12            if i == 0 or i == n - 1 or j == 0 or j == p - 1:
13                Y[i][j] = M[i][j]
14            else:
15                # Utilisation des fonctions som_prods (Q.24) et voisins
                  (Q.25)
16                V = voisins(M, i, j)
17                Y[i][j] = som_prods(V, sobelV)
18     return Y

```

Q.28 - Fonction gradients(M)

Cette fonction calcule l'intensité du gradient pour chaque pixel à l'aide de la formule de la norme :

$$G_{i,j} = \lfloor \sqrt{X_{i,j}^2 + Y_{i,j}^2} \rfloor.$$

N.B : Le module numpy est supposé déjà importé et contient toutes les fonctions mathématiques utiles pour répondre à la question (pas besoin d'importer le module math)

```

1 def gradients(M):
2     n = len(M)
3     p = len(M[0])
4     G = np.zeros((n,p))
5     # Récupération des matrices de gradients horizontaux et verticaux
6     X = gradH(M)
7     Y = gradV(M)
8     for i in range(n):
9         for j in range(p):
10             # Application de la formule mathématique (avec troncature
11                 entière)
12             G[i][j] = int(np.sqrt(X[i][j]**2 + Y[i][j]**2))
13     return G

```

Q.29 - Fonction orientations(M)

Cette fonction calcule la matrice des orientations D en degrés. Elle utilise la fonction **np.arctan2(y, x)** qui renvoie une valeur en radians comprise entre $-\pi$ et π . Il faut donc convertir ce résultat en degrés, puis recadrer la plage de valeurs sur $[0, 360[$.

```

1 def orientations(M):
2     n = len(M)
3     p = len(M[0])
4     D = np.zeros((n,p))
5     X = gradH(M)
6     Y = gradV(M)
7     for i in range(n):
8         for j in range(p):
9             # arctan2 prend en arguments (y, x) soit ici (X_ij, Y_ij)
10                selon l'énoncé
11            angleRad = np.arctan2(X[i][j], Y[i][j])
12            # Conversion en degrés
13            angleDeg = angleRad*180/np.pi
14            # Recadrage dans l'intervalle [0, 360[
15            if angleDeg < 0:
16                angleDeg = angleDeg + 360.0
17            D[i][j] = angleDeg
18     return D

```

Q.30 - Fonction `hist_secteur(M, i, j)`

Un secteur est un bloc de taille 8×8 dont le pixel supérieur gauche est situé aux coordonnées (i, j) . L'intervalle des angles est découpé en 8 sous-intervalles de largeur 45° . Chaque pixel du secteur "vote" dans l'histogramme en ajoutant l'intensité de son gradient $G_{i,j}$ dans la case associée à son orientation $D_{i,j}$.

```

1 def hist_secteur(M, i, j):
2     # Calcul global des caractéristiques de l'image
3     G = gradients(M)
4     D = orientations(M)
5     # Initialisation de l'histogramme de taille 8 avec des zéros
6     H = [0 for _ in range(8)]
7     # Parcours exclusif des pixels du secteur 8x8
8     for r in range(i, i + 8):
9         for c in range(j, j + 8):
10             valG = G[r][c]
11             valD = D[r][c]
12             # Détermination de l'indice de l'intervalle d'orientation
13             indexInt = int(valD // 45)
14             # Gestion de la borne supérieure stricte 360°
15             if indexInt >= 8:
16                 indexInt = 7
17             # Ajout du poids du gradient dans l'intervalle
18                 correspondant
19             H[indexInt] += valG
20     return H

```

Q.31 - Fonction `HOG(M)`

Cette fonction extrait le descripteur final de l'image. Elle subdivise l'image en blocs formés de 2×2 secteurs adjacents.

Les secteurs se chevauchant, le point de départ de chaque secteur se déplace avec un pas de 8 pixels (puisque chaque secteur mesure 8×8).

Un bloc de 2×2 secteurs couvre une zone de 16×16 pixels.

L'histogramme d'un bloc est la concaténation simple de ses 4 histogrammes de secteurs. Le vecteur HOG total est la concaténation des histogrammes de tous les blocs.

```
1 def HOG(M):
2     n = len(M)
3     p = len(M[0])
4     vecteurHog = []
5
6     # Le nombre de secteurs en hauteur et en largeur
7     nb_secteurs_n = n // 8
8     nb_secteurs_p = p // 8
9
10    # Parcours de tous les blocs de 2x2 secteurs
11    for s_i in range(nb_secteurs_n - 1):
12        for s_j in range(nb_secteurs_p - 1):
13            # Calcul des coordonnées réelles (pixels) des 4 secteurs du
14            # bloc courant
15            C1 = hist_secteur(M, s_i * 8, s_j * 8)
16            C2 = hist_secteur(M, s_i * 8, (s_j + 1) * 8)
17            C4 = hist_secteur(M, (s_i + 1) * 8, s_j * 8)
18            C5 = hist_secteur(M, (s_i + 1) * 8, (s_j + 1) * 8)
19
20            # Concaténation des listes pour former l'histogramme du
21            # bloc
22            hist_bloc = C1 + C2 + C4 + C5
23
24            # Ajout de l'histogramme du bloc au descripteur global HOG
25            vecteurHog.extend(hist_bloc)
26
27    return vecteurHog
```

Fin de la correction

*

Pour toute **question** et/ou **suggestion** ou **remarque** ou **demande**

Pr Fouad NAFIS : fouadnafis@gmail.com

CPGE : Meknès

06.75.38.39.48