DS1 - INFORMATIQUE

DETECTION D'OBJETS DANS LE CADRE DE LA CONDUITE AUTONOME

Les calculatrices sont interdites.

Le sujet est composé de quatre parties indépendantes.

Important : vous pouvez librement utiliser les fonctions issues des questions précédentes, même si vous ne les avez pas traitées. Vous devez répondre directement sur le **Document Réponse**, soit à l'emplacement prévu pour la réponse lorsque celle-ci implique une rédaction, soit en complétant les différents programmes en langage Python.

Énoncé : 11 pages
Annexe : 1 page
Document Réponse (DR) : 8 pages

Seul le Document Réponse doit être rendu dans son intégralité
(le QR Code doit être collé sur la première page du DR).

Dans ce sujet, les fonctions sont définies avec leur signature :

`ma_fonction(arg1:type1, arg2:type2)`→`type3`

Cette notation permet de définir une fonction qui se nomme `ma_fonction` qui prend deux arguments en entrée `arg1` de type `type1` et `arg2` de type `type2`. Cette fonction renvoie une valeur de type `type3`.
Il n'est pas nécessaire de recopier les signatures des fonctions dans le **Document Réponse**, il suffit d'écrire directement :

```
def ma_fonction(arg1,arg2) :
    #liste d'instructions
```

L'entreprise Easymile a mis en place à Toulouse des bus 100 % autonomes sur le campus universitaire de Paul Sabatier. Il s'agit d'un projet de recherche mené par l'IRIT et l'Université, mais dès aujourd'hui, les minibus (**figure 1**) qui peuvent accueillir vingt personnes sont en fonction et se déplacent dans un rayon de 5 km. Ces véhicules autonomes nécessitent un niveau élevé d'informations pour fonctionner en toute sécurité et sont donc équipés d'une gamme complète de capteurs.

Ces capteurs collectent et analysent les données enregistrées pour créer une image à 360 degrés de l'environnement, y compris les infrastructures, les autres véhicules, les piétons et tout ce qui se trouve sur le chemin.

Le traitement en temps réel des données permet au système du véhicule autonome de décider comment se comporter pour progresser en toute sécurité sur la route (s'arrêter, partir, ralentir, etc.).

Ce sujet s'intéresse à une partie des algorithmes mis en place afin de rendre le projet possible, notamment la partie détection d'obstacles dans l'espace.

En effet, pour détecter les objets, les piétons et les véhicules avec précision et rapidité, il est nécessaire de mettre en place des programmes robustes avec une complexité limitée.

Dans un premier temps, nous nous intéresserons à un algorithme simple de détection d'obstacles, basé sur l'apprentissage supervisé. Il faudra tout d'abord réfléchir à la méthode de localisation dans l'espace du piéton. Dans un deuxième temps, nous nous pencherons sur le traitement de l'image et la détection des obstacles. Enfin, dans le but d'améliorer et de comprendre le fonctionnement des programmes du bus, la collecte d'un grand nombre de données est réalisée à chaque déplacement, c'est l'objet de la dernière partie du sujet.

Figure 1 - Bus Easymile

Partie I - Localisation dans l'espace

Un premier calcul important pour détecter les piétons et les obstacles est la prise en compte de la distance focale de la caméra de détection. Pour cela, un calcul simplifié est réalisé à partir d'une taille réelle standard. Les caméras permettent en effet de relever une matrice de pixels qui donne ainsi une dimension aux formes et aux distances.

Sur la **figure 2**, lorsque A' est confondu avec F', la relation entre la focale f et la distance réelle est la suivante :

$$D_{réelle} = \frac{\left\|\overrightarrow{OF'}\right\| \cdot \left\|\overrightarrow{AB}\right\|}{\left\|\overrightarrow{A'B'}\right\|} = f \cdot \frac{H_{objet}}{H_{image}} \quad (1)$$

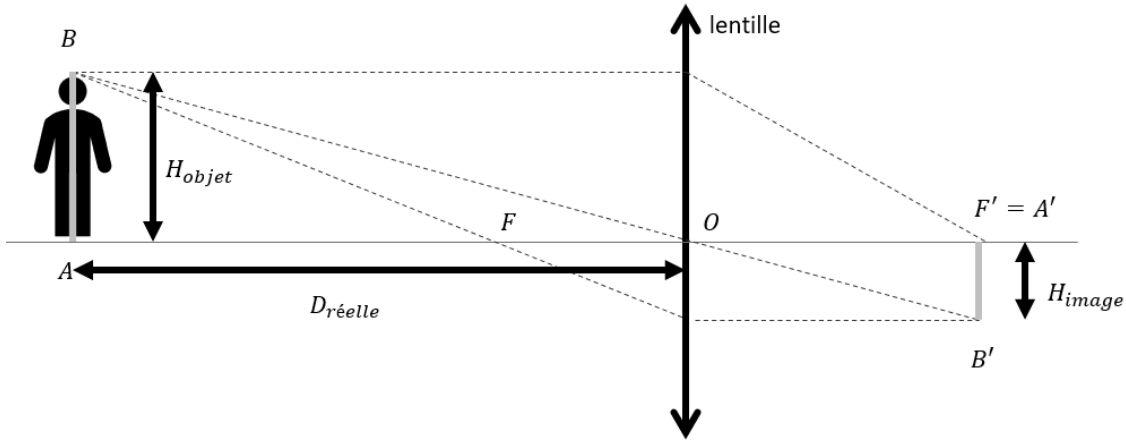


Figure 2 - Calcul simplifié de la focale f

Nous allons tout d'abord créer un dictionnaire avec 5 obstacles " classiques " qui doivent être détectés.

Q1. Créer un dictionnaire nommé `dico`, possédant 5 clés de type `str` : `'adulte'`, `'enfant'`, `'animaux'`, `'véhicule'`, `'indéterminé'`. Chaque clé sera initialisée avec comme valeur une liste vide.

Pour chacune des clés du dictionnaire créé à la question **Q1**, nous allons construire un intervalle de hauteur pouvant correspondre à l'obstacle et réparti à 20 % de part et d'autre de la hauteur moyenne de celui-ci, sous la forme d'une liste de deux éléments :

- pour l'adulte, la hauteur moyenne est fixée à 175 cm, la liste associée à la clé sera alors `[175*0.8, 175*1.2]` ;
- pour l'enfant, la hauteur moyenne est fixée à 110 cm ;
- pour les animaux, la hauteur moyenne est fixée à 50 cm ;
- pour les véhicules, la hauteur moyenne est fixée à 200 cm ;
- pour les autres valeurs indéterminées, le couple associé à la clé sera composé des valeurs les plus basses `[0, 50*0.8]`.

Q2. Écrire une suite d'instructions qui modifie votre dictionnaire initialisé à la question **Q1** en associant le couple d'intervalles à chacune des clés du dictionnaire.

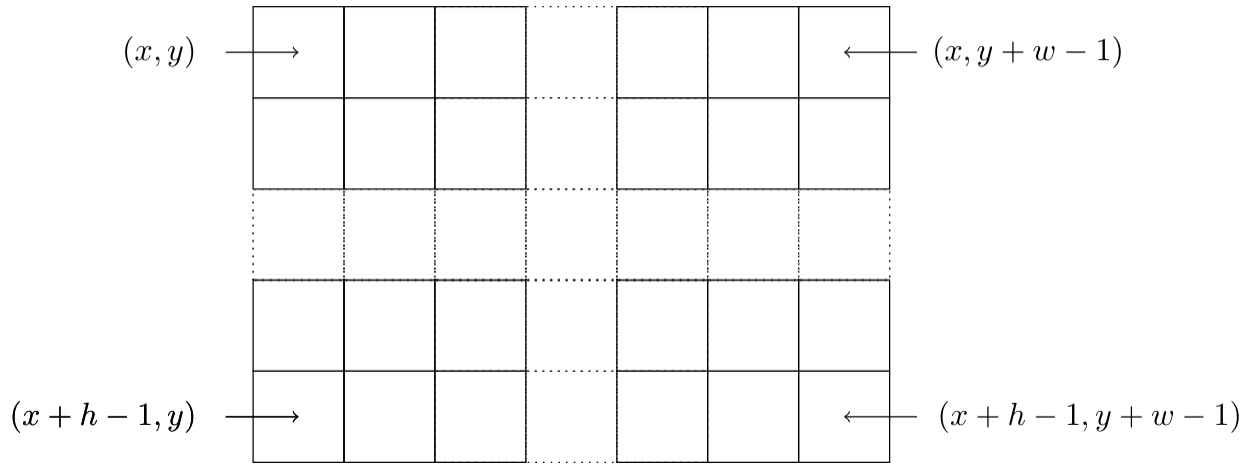
Q3. Écrire une fonction `obstacles_rencontres(dico : dict, hauteur : float)→list[str]` qui permet de vérifier que pour une hauteur donnée, le nom de l'obstacle est bien renvoyé. Elle prend donc en entrée une hauteur réelle en cm et le dictionnaire créé aux questions précédentes, contenant les couples clé, intervalle de hauteur. Dans le cas d'au moins une possibilité, tous les noms seront renvoyés dans une liste de chaîne de caractères. Si par contre la hauteur n'est dans aucun intervalle, la liste vide sera renvoyée.

Par exemple, pour une hauteur de 170 cm, la fonction renvoie `['adulte', 'véhicule']`.

Les algorithmes de détection d'obstacles doivent traiter les images provenant des caméras. Les images issues de la caméra sont ici considérées comme un tableau à deux dimensions qui contient des pixels.

Nous allons ici travailler dans l'une de ces images de pixels pour comprendre la détection des obstacles. On dispose d'une liste `faces_detec` constituée de quadruplets `(x,y,h,w)` correspondant chacun à un contour détecté dans une image captée. Ici `x,y` désignent les coordonnées du coin supérieur gauche dans l'image, `h` la hauteur du contour, `w` sa largeur.

Nota : il est important de rappeler que les images sont parcourues de haut en bas avec `x` et `h` travaillant sur les lignes, `y` et `w` sur les colonnes, `x, y, h` et `w` sont en nombre de pixels. Le dernier pixel de l'image se situe donc à la coordonnée $(x + h - 1, y + w - 1)$.



Q4. Écrire une fonction `focale(faces_detec : list, Dr : float, Hr : float)→ list[float]` prenant pour argument une liste `faces_detect` de contours, une distance `Dr` réelle et une hauteur `Hr` réelle données, et qui renvoie la liste contenant la valeur de la focale définie dans l'équation (**1**) pour chaque contour détecté.

Q5. Écrire une fonction `moy(L:list[float])→float` qui renvoie la moyenne des éléments de la liste `L`.

Partie II - Traitement de l'image

II.1 - Calcul de luminance

Nous allons nous intéresser maintenant au format des images et plus particulièrement au passage en niveau de gris, c'est-à-dire le calcul de la luminance moyenne utile à la détection de contour.
La teinte d'un pixel peut être représentée de plusieurs façons. Une méthode courante, basée sur la synthèse additive, consiste à la décomposer en trois composantes qui correspondent aux couleurs rouge, vert et bleu.
On parle de représentation RVB (Rouge, Vert et Bleu). Chacune des trois composantes donne l'intensité de la couleur correspondante dans la teinte finale sous forme d'un nombre entier compris entre 0 et 255. 0 indique l'absence de cette couleur et 255 l'intensité maximale. Ainsi, le triplet (0, 0, 0) désigne un pixel noir et (255,255,255) un pixel blanc.

Rappelons que les images sont représentées sous la forme d'une liste de lignes où chaque ligne est une liste de triplets RVB. Ainsi, on accède au pixel de coordonnées `(x,y)` de l'image `I` par l'expression `I[x][y]`.

Q6. Sachant qu'une composante de couleur est représentée sur 8 bits, préciser l'espace mémoire nécessaire pour stocker un pixel, puis pour stocker une image de taille 8 000 x 6 000 de pixels. La réponse pour l'image sera donnée en Mo (1 Mo = un million d'octets).

Pour représenter une image en niveau de gris, il suffit d'une valeur par pixel représentant l'intensité de gris entre le noir et le blanc, c'est la luminance. Pour convertir une image en couleurs en niveaux de gris, plusieurs méthodes sont possibles. Faire la simple moyenne des composantes *R,V* et *B* d'un pixel donne visuellement des résultats décevants.

On procède donc selon le protocole suivant :

- notons C une des composantes *R, V* ou *B* d'un pixel. On pose $C_1 = C/255$; elle sera transformée en une variable C_{lin} selon la définition suivante :
  - $C_{lin} = \frac{C_1}{12{,}92}$ si $C_1 \leq 0{,}04045$;
  - $C_{lin} = \left(\frac{C_1 + 0{,}055}{1.055}\right)^{2.4}$ sinon ;

- puis on calcule la valeur " linéaire " du niveau de gris Y_{lin} avec la formule :
  $Y_{lin} = 0{,}2126 \cdot R_{lin} + 0{,}7152 \cdot V_{lin} + 0{,}0722 \cdot B_{lin}$;

- enfin l'intensité Y du pixel de l'image en niveau de gris sera calculée comme suit :
  - $Y = \lfloor 255 * 12{,}92 * Y_{lin} \rfloor$ si $Y_{lin} \leq 0{,}0031308$;
  - $Y = \left\lfloor 255 * \left(1{,}055 * Y_{lin}^{1/2.4} - 0{,}055\right) \right\rfloor$ sinon.
    ($\lfloor x \rfloor$ désigne la partie entière de x)

Q7. Écrire une fonction `Clinear(C:int)→ float`, qui prend en argument une composante de couleur `C` d'un pixel et qui renvoie la valeur C_{lin} décrite plus haut.

Q8. Écrire une fonction `Intensite(pix:tuple)→int` qui prend en argument un triplet de trois composantes correspondant à un pixel au format RVB et qui renvoie la valeur Y du niveau de gris correspondant.

Q9. Écrire une fonction `init(h:int,w:int)→list` prenant en argument les 2 entiers `h` et `w` caractérisant la taille d'une image et qui renvoie une liste de `h` listes remplie de `w` zéros.

Par exemple, `init(2,3)` renvoie `[[0,0,0],[0,0,0]]`.

Q10. Écrire une fonction `NiveauxGris(I:list)→list` prenant en argument une image `I` au format RVB et qui renvoie une image de même taille en niveau de gris.

Q11. Estimer la complexité asymptotique pour calculer la luminance de tous les pixels d'une image ayant h lignes de pixels et w colonnes, en fonction des variables h et w.

II.2 - Le concept d'image-intégrale

On travaille ici avec des images en niveau de gris. Comme vu précédemment, la valeur du niveau de gris d'un pixel s'appelle la luminance du pixel. Lors du processus de détection des contours, on est amené à faire un très grand nombre de fois la moyenne des luminances sur des sections (des portions) de l'image d'origine. Un calcul naïf de cette moyenne entraîne alors une complexité temporelle importante.
La méthode de l'image-intégrale permet, au prix du pré-calcul initial de l'image-intégrale, d'accélérer considérablement le calcul des luminances moyennes.

La **figure 3** montre un exemple du principe de l'image-intégrale qui consiste à sommer les intensités au fur et à mesure que l'on balaye les pixels au lieu de garder la valeur d'intensité du pixel seul.

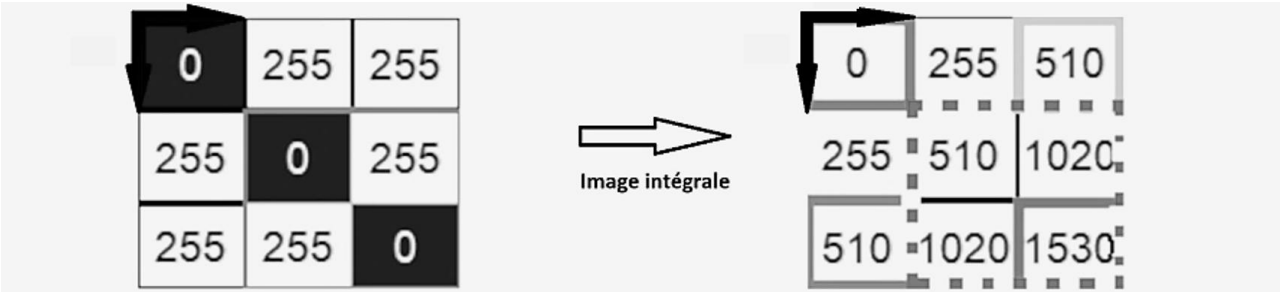


Figure 3 - Somme des pixels sur une section 3x3 avec la méthode d'image intégrale (à gauche, les 9 pixels d'origine, à droite la somme des pixels effectuée par balayage)

Dans ce qui suit, h désigne le nombre de lignes et w le nombre de colonnes de l'image. Les sections de l'image sur lesquelles on calculera la luminance moyenne auront n lignes et p colonnes. Ces quatre valeurs h, w, n, p sont fixées au début du traitement.

Q12. Expliquer les deux lignes du **DR**.

Q13. Créer une fonction `coef_integral(M:list,x:int,y:int)→int` qui renvoie l'intégrale de l'image jusqu'à un point `(x,y)` suivant la formule (**2**), la somme étant prise nulle quand elle ne comporte aucun terme :

$$I(x,y) = \sum_{j=0}^{x} \sum_{k=0}^{y} M_{j,k}\,. \tag{2}$$

Q14. À l'aide de cette fonction `coef_integral`, donner les instructions permettant de créer la matrice image-intégrale `M_int` d'une matrice `M` (liste de listes). Les coefficients `M_int[x][y]` de cette matrice sont donnés par la valeur `I[x][y]` définie à la formule (**2**).

La formule qui calcule ensuite la luminance moyenne $\overline{\mathrm{m}}$ sur une section de coin supérieure gauche (x, y) est la suivante :

$$\overline{m} = \frac{1}{n.p}\left(M\_int_{x+n,y+p} - M\_int_{x+n,y} - M\_int_{x,y+p} + M\_int_{x,y}\right), \tag{3}$$

n et p étant respectivement la hauteur et la largeur de la section de l'image.

Q15. Écrire une fonction `lumi_section(M_int:list,x:int,y:int,n:int,p:int)→float` qui prend pour arguments une image intégrale `M_int`, les coordonnées `x,y` du coin supérieur gauche d'une section, la hauteur `n` et la largeur `p` de cette section et qui renvoie la luminance moyenne de la section correspondante de l'image `M`.

II.3 - Détection des obstacles : algorithme de Viola-Jones

L'algorithme de Viola-Jones implémenté ici prend en argument la matrice image intégrale `M_int`.

Le calcul de luminance moyenne est ensuite réalisé sur chacune des sections de l'image en séparant les sections en deux sur la largeur, c'est-à-dire que l'on peut définir la moyenne gauche $\overline{\mathrm{m}}_g$ allant de y à $y + \lfloor p/2 \rfloor$ inclus et la moyenne droite $\overline{\mathrm{m}}_d$ allant de $y + \lfloor p/2 \rfloor$ à $y + p$.

Le coefficient C_{obs} associé à chaque section est alors calculé suivant l'équation (**4**).

$$C_{obs} = max\left(\overline{m}_d, \overline{m}_g\right) - min\left(\overline{m}_d, \overline{m}_g\right). \tag{4}$$

Ce coefficient est comparé à une constante `C_ref` caractéristique pseudo-Haar qui vient d'un classifieur (données associées à une base de données sur chaque objet particulier). Si la corrélation dépasse un seuil défini (compris lui aussi entre 0 et 1), c'est-à-dire si `C_obs/C_ref > seuil`, le coin supérieur gauche de cette section est alors ajouté à la liste renvoyée en sortie.

L'algorithme permet ainsi de donner la liste de tous les coins supérieurs des objets détectés avec les caractéristiques pseudo Haar du classifieur.

Q16. Écrire la fonction `detection(M_int:list,n:int,p:int,C_ref:int,seuil:float)` →`list` qui prend en argument une matrice image intégrale `M_int`, la taille des sections associées `n, p,` la caractéristique pseudo-Haar `C_ref` et le `seuil` imposé. Elle réalise l'algorithme de Viola-Jones et renvoie en sortie la liste des coins des objets détectés.

On considère que les initialisations et définitions des matrices `M_int` et tailles des sections ont bien été réalisées en amont. Les fonctions natives de Python `min,max` sont autorisées.

Partie III - Optimisation - Méthode KNN

La méthode KNN est une méthode d'apprentissage dit supervisé ; les données sont déjà classées par groupes clairement identifiés et on cherche à quels groupes appartiennent de nouvelles données. Son utilisation semble pertinente ici car on dispose d'un volume important de données étiquetées.
Le principe de la méthode est simple. Après avoir calculé la distance euclidienne entre toutes les données connues des obstacles et les données d'un nouvel obstacle à classer, on extrait les K données connues les plus proches. L'appartenance du nouvel obstacle à un groupe est obtenue en cherchant le groupe majoritaire, c'est-à-dire, le groupe qui apparaît être le plus représentatif parmi les K données connues.

On note $X_i, i \in [\![0, N-1]\!]$, le i^e vecteur ligne du tableau de données `data`, correspondant aux données d'un obstacle i déjà classé. Ce vecteur possède n coordonnées. Le tableau `data` contient N lignes correspondant à N obstacles différents, sa dimension est donc $N * n$. On cherche alors à déterminer à quelle classe appartient un nouvel obstacle qui lui aussi sera représenté par un vecteur Z de coordonnées $(z_0, z_1, \ldots, z_{n-1})$.

Les calculs matriciels sont beaucoup plus faciles à manipuler avec le module `numpy` qui est explicité en **Annexe**. Il est cependant possible de réaliser la totalité des questions suivantes en utilisant des listes de listes ou des `array` de `numpy`.

Le candidat choisira librement le type qu'il préfère manipuler (liste de listes ou `array` avec les fonctions `numpy` associées) mais pour la suite des questions, nous considérons que le module `numpy` est importé comme suit :

```
import numpy as np.
```

III.1 - Données

Avant de calculer la distance euclidienne entre les différents vecteurs, il est préférable de normaliser ceux-ci pour éviter que l'un d'entre eux ait plus de poids par rapport à un autre. Nous allons ainsi ramener toutes les coordonnées d'un vecteur entre 0 et 1 grâce à une technique de normalisation reposant sur une transformation affine.

Considérons un vecteur $X = (x_0, x_1, \ldots, x_{n-1})$.On note $min(X)$ son minimum et $max(X)$ son maximum. Le vecteur normalisé associé à X, noté $X_{norm} = (x_{norm,0}, x_{norm,1}, \ldots, x_{norm,n-1})$ est défini par :

$$\forall\, j \in \{0,1,\ldots,n-1\},\ x_{norm,j} = \frac{x_j - min\,(X)}{max(X) - min\,(X)}. \qquad (5)$$

Q17. Écrire une fonction `min_max(X :list)→tuple` qui renvoie les valeurs du minimum et du maximum d'un vecteur X passé en argument. La fonction devra être de complexité linéaire. Dans cette question uniquement, l'usage de fonctions `min, max` prédéfinies dans le langage Python est interdit.

Q18. Écrire une fonction `coeff_normalise(data:list)→list` qui prend en argument un tableau de données non normalisées `data` et qui renvoie un nouveau tableau `data_nom` contenant cette fois les données normalisées.

Q19. Écrire une fonction `distance(Z:list,data:list)→list` qui parcourt les N lignes du tableau `data` où les données sont normalisées et calcule les distances euclidiennes entre le n-uplet Z et chaque n-uplet X du tableau de données connues (on rappelle que X représente une ligne du tableau `data`). La fonction doit renvoyer une liste de taille N contenant les distances entre chaque n-uplet X et le n-uplet Z.

III.2 - Détermination des K plus proches voisins

Pour déterminer les K plus proches voisins avec K un entier choisi arbitrairement, il suffit d'utiliser un algorithme de tri efficace. La liste `L` à trier est une liste de listes à deux éléments contenant :

- la distance entre le n-uplet à classer et un n-uplet connu (on trie par ordre croissant sur ces valeurs) ;
- la valeur du type d'obstacle correspondant au n-uplet connu.

On retient l'algorithme suivant :

```
def tri (L):
    """Reçoit une liste L en argument.
    Renvoie la liste triée correspondante"""
    if len(L) <= 1:
        return T
    else:
        m= len(L) //2
        L1 = []
        for x in range(m):
            L1.append(L[x])
        L2 = []
        for x in range(m, len(L)):
            L2.append(L[x])
        return fusion ( tri (L1) , tri (L2))

def fusion (T1,T2):
    """ Reçoit en arguments deux listes triées T1 et T2.
    Renvoie une liste triée obtenue en fusionnant les deux listes T1 et T2 """
    if T1 == []:
        . . . . . #ligne 1 à compléter
    if T2 == []:
        . . . . . #ligne 2 à compléter
    if T1[0][0] <T2[0][0]:
        return [T1[0]]+fusion (T1[1:] ,T2)
    else:
        . . . . . #ligne 3 à compléter
```

Q20. Compléter sur le **DR** les lignes 1 à 3 de la fonction `fusion` pour que la fonction `tri` effectue le travail spécifié.

Q21. Lorsque l'on fait des tests de performance sur cet algorithme, on obtient des résultats très décevants par rapport à d'autres algorithmes de tris. Identifier la ou les causes de ces problèmes de performance. Le coût du "slicing" est explicité en **Annexe**.

L'algorithme de la méthode KNN est décrit par la fonction python suivante :

```
def KNN(data ,typeObs ,z,K,nb):
    #partie 1
    T= []
    dist = distance(z,data)
    for i in range(len(dist)):
        T.append([dist[i] , i ])
    L=tri (T)

    #partie 2
    select = [0]*nb
    for i in range(K):
        select [typeObs[L[i][1]]]+=1

    #partie 3
    ind = 0
    res = select [0]
    for k in range(1,nb):
        if select [k] > res:
            res = select [k]
            ind = k
    return ind
```

`typeObs` permet de récupérer la classe d'un objet déjà classifié, K représente le nombre de voisins proches retenus, nb correspond au nombre de type d'obstacles (5 dans l'exemple de la question **Q1**) et z est le vecteur représentant l'obstacle à classer.

Q22. Expliquer ce que font globalement les parties 1, 2 et 3 de l'algorithme. Préciser ce que représentent les variables locales `T, dist, select, ind`.

III.3 - Validation de l'algorithme

Pour tester l'algorithme, on utilise un nouveau jeu de données normalisées représentants 200 obstacles connus de 5 types différents (exemple de la question **Q1**).

On note `datatest` le tableau contenant ces données et `typetest` la liste des types connus pour ces obstacles. Le but est de déterminer si les prévisions de type fournies par l'algorithme KNN pour chaque obstacle correspondent aux types réels des obstacles stockés dans `typetest`. Pour cela, on applique l'algorithme sur chaque élément de ce jeu de données pour une valeur de K fixée.
On définit la fonction suivante qui renvoie une matrice `mat` :

```
def test_KNN(datatest , typetest ,data ,typeObs ,K,nb):
    typepredit = []
    for i in range(len(datatest)):
        res =KNN(data ,typeObs ,datatest [i] ,K,nb)
        typepredit.append(res)

    mat = [[0 for j in range(nb)] for i in range(nb)]
    for i in range(len(etattest)):
        mat[typetest[i]][typepredit[i]] += 1
    return mat
```

On obtient pour $K = 8$ la matrice suivante : $\begin{pmatrix} 27 & 4 & 3 & 5 & 2 \\ 0 & 18 & 2 & 4 & 3 \\ 5 & 2 & 40 & 7 & 2 \\ 2 & 5 & 7 & 36 & 4 \\ 4 & 3 & 1 & 1 & 19 \end{pmatrix}$

Q23. Exploiter les valeurs de la première ligne de cette matrice en expliquant les informations que l'on peut en tirer. Donner enfin l'efficacité (ou pourcentage de réussite) de l'algorithme KNN pour cet exemple.

ANNEXE

Définir une liste : **`L=[1,2,3]`**
Définir un dictionnaire : **`dic={'a':0, 'b':1, 'c':2}`**
Accéder à un élément : **`L[0]`** renvoie **`1`** **`dic['a']`** renvoie **`0`**
Extraire une sous-liste : **`L[1:2]`** renvoie **`[2]`**
Vérifier si une clé est dans un dictionnaire : **`'a' in dic`** renvoie **`True`**
Ajouter un élément à une liste : **`L.append(5)`**
Supprimer le dernier élément d'une liste et le renvoyer : **`a=L.pop()`**
Copier une liste : **`L2=L.copy()`**
Ajouter un élément à un dictionnaire : **`dic['d']=`**4
Définir une chaîne de caractères : **`mot='Python'`**
Taille d'une chaîne, d'une liste ou d'un dictionnaire : **`len(mot)`**
Extraire des caractères : **`mot[2:6]`**
Concaténer des chaînes ou des listes : **`'cc'+'inp'`** donne **`'ccinp'`**
Dupliquer des chaînes ou des listes : **`'c'*3`** donne **`'ccc'`**
Convertir en flottant, en entier, en chaîne, en liste : **`float(s), int(s), str(L), list(s)`**
Parcours en valeur d'un dictionnaire : **`for v in dic.values() : print(v)`**
Parcours des clés d'un dictionnaire : **`for c in dic : print(c)`**
Parcours des clés et des valeurs d'un dictionnaire : **`for c,v in dic.items() : print(c,v)`**

Module numpy

La bibliothèque NumPy permet d'effectuer des calculs numériques avec Python. Elle introduit une gestion facilitée des tableaux de nombres.
`import` `numpy` **`as`** `np`
`np.array(u)` crée un nouveau tableau contenant les éléments de la séquence `u`. La taille et le type des éléments de ce tableau sont déduits du contenu de `u`.
`np.empty(n, dtype), np.empty((n, m), dtype)` crée respectivement un vecteur à n éléments ou un tableau à n lignes et m colonnes dont les éléments, de valeurs indéterminées, sont de type dtype qui peut être un type standard (bool, int, float, …) ou un type spécifique numpy (np.int16, np.float32, …). Si le paramètre dtype n'est pas précisé, il prend la valeur float par défaut.
`np.zeros(n, dtype), np.zeros((n, m), dtype)` fonctionne comme np.empty en initialisant chaque élément à la valeur zéro pour les types numériques ou False pour les types booléens.
`a.shape` tuple donnant la taille du tableau a pour chacune de ses dimensions.
`len(a)` taille du tableau a dans sa première dimension, équivalent à `a.shape[0]`.
`a.size` nombre total d'éléments du tableau `a`.
`a.min(), a.max()` renvoie la valeur du plus petit (respectivement plus grand) élément du tableau `a` ; ces opérations ont une complexité temporelle en O(`a.size`).

Coût du "slicing" sur les listes Pyhton

Avec les listes Python, le slicing **`T1[1:n]`** a une complexité en $\boldsymbol{O(n-1)}$: le temps de recopie des éléments dans une nouvelle liste.