Python Files\main.py

```python
import numpy as np
import matplotlib.pyplot as plt
import imageio as io
from time import time
from random import choice
from math import pi, cos, sin, floor
from skimage.feature import peak_local_max
from os import system
import subprocess

from modules import *
# importation les fonctions permettant d'enregistrer des images
from images import *

## Variables globales

BLANC = 254 # définit le seuil pour la couleur blanche
MAX_INT = 42e42
## Liens des images sources


# Fonctions
## Fonction de base sur l'image

# Inverse l'image selon la coordonnée y
def inv_img_y(img: list) -> list:
    return(img[::-1])

# lecture de l'image liée au lien [lien] depuis le serveur web
def read_img(lien: str) -> np.array:
    im = io.v2.imread(lien)
    im = np.array(im)
    return(inv_img_y(im))


# affiche l'image [img].
# [cmap='gray'] premet d'afficher correctement les images en noir et blanc ainsi que les images en couleur
```

```python
# pour les images en niveaux de gris, le pixel le plus sombre sera noir et le pixel le plus clair sera blanc.
# [rev] premet d'inverser l'image
def show_image(img: list, rev: bool = False) -> None:
    plt.axis('off') # enlève les axes
    if rev :
        plt.imshow(img, cmap='gray',origin='lower', aspect='auto')
    else :
        plt.imshow(img, cmap='gray', aspect='auto')


## Traitement des images

# color_to_grayscale [image] convertis l'image [image] en RGB en une image (np.array) en niveau de gris
def color_to_grayscale(image: list) -> np.array:
    image = list(image)
    r = [[] for x in range(len(image))]                                                    #image vide
    for i in range (len(image)) :
        for j in range(len(image[i])) :
            valeur = 0.2126*image[i][j][0] + 0.7152*image[i][j][1] + 0.0722*image[i][j][2]  #formule de conversion de noir et blanc a
couleur
            r[i].append(valeur)

    return np.array(r)

# grayscale_to_black_and_white img prend en entrée l'image [img] et renvoie un image en noir et blanc. les pixels
# de couleur superieure à [seuil] vont être laissé dans leur couleur originale et ceux inferieur a seuil vont etre mis en noir
# par default, la fonction conserve uniquement les pixels parfaitement blanc
def grayscale_to_black_and_white(img: np.array, seuil:int = BLANC) -> np.array:
    r = [[] for x in range(len(img))]
    for i in range(len(img)):
        for j in range(len(img[i])):
            if img[i][j] < seuil :
                r[i].append(0)
            else :
                r[i].append(int(img[i][j]))
    return(np.array(r))

## Matrice de Hough

# retourne la taille du tableau qui represente l'espace de hough.
# l'option precision change la taille du tableau selon la dimention x, c'est a dire selon les theta
```

```python
def taille_Hough(img: list, precision: int = 1) -> tuple:
    Px = len(img)
    Py = len(img[0])
    hauteur = (Px ** 2 + Py ** 2)**0.5 # distance maximum (coin en bas à gauche)
    largeur = 360
    return(round(hauteur), largeur*precision)


# rho_f(x, y, θ) retourne la distance a l'origine de la droite passant par le point (x, y) et d'angle θ
def rho_f(x: int, y: int, θ: float) -> float:
    return(x*cos(θ) + y*sin(θ))


#retourne la matrice de l'espace Hough de l'image img dans l'ordre m[r][theta]
def espace_Hough(img: list, precision: int = 1) -> list:
    hauteur_H, largeur_H = taille_Hough(img, precision)
    hauteur_I, largeur_I = len(img), len(img[0])
    #creer le tableau qui va contenir la matrice
    matrice_hough = [[0 for i in range (largeur_H)] for i in range(hauteur_H)]
    #on parcours toute l'image
    for y in range(hauteur_I):
        for x in range(largeur_I):
            if img[y][x] >= BLANC:
                for theta_deg in range(largeur_H):
                    theta_rad = theta_deg /(180 * precision) * pi
                    rho = round(rho_f(x, y, theta_rad))
                    if rho>0: #comme on fait varier θ jusqu'a 2π, on ne garde que les valeurs positives
                        matrice_hough[rho][theta_deg] += 1 #on met l'origine du repere en bas de l'image
    return(matrice_hough)


# retourne une liste de maximum locaux.
def maximum_global(m: list, d: int = 100):
    return(peak_local_max(np.array(m), min_distance=d))


## Géometrie

# calcule l'équation d'une droite passant par deux points en coordonnées carthésiennes.
# la valeur de retour sera un couple (a, b) représentant la droite sous forme y = ax + b
def equation(point1: tuple, point2: tuple) -> tuple:
```

```
    (x1,y1),(x2,y2) = point1,point2
    assert(x1 != x2), "les points sont alignés, la droite ne peut pas être prise en compte"
    a = (y1 - y2) / (x1 - x2) #si la droite n'est pas verticale
    b = y2 - a*x2
    return(a,b)


# retourne les coefficients de l'equation de la droite représentée par une distance a l'origine r et un angle theta
# sous la forme d'un couple (a, b) représentant la droite sous forme y = ax + b
def droite(r: float, theta: float, precision: int = 1) -> tuple:
    theta = theta / precision
    if theta == 0 or theta == 180:
        return (0,r)
    else :
        theta_rad = (pi/180)*theta
        a = -(cos(theta_rad)/sin(theta_rad))
        b = r/sin(theta_rad)
    return(a,b)

# Renvoie la distance euclidienne entre les points a: (x1, y1) et b : (x2, y2)
def distance (a: tuple, b: tuple) -> int:
    (x1,y1),(x2,y2) = a,b
    distance = ( (x1-x2)**2 + (y1 - y2)**2 )**0.5
    return(distance)


# calcule la matrice de Hough a l'aide de d'un programme en C qui utilise du multithreading.
# permet une vitesse d'execution beaucoup plus élevée : temps d'exécution divisé par 100
# l'argument bypass permet de tester d'autres partie du code plus rapidement
def hough_c(image: list, precision: int = 10, bypass: bool = False) -> list:
    if not bypass:
        save_img_c(image)
        subprocess.run(['../c/multi.out', str(precision)])
    mat = read_c()
    return(mat)

# renvoit la matrice renvoyée par le calcul de l'espace de hough en C
def read_c() -> list:
    f = open("../c/out.txt","r")
    m_read = f.readlines()
    f.close()
```

```python
    return([[int (x) for x in y.split(",")] for y in m_read])


# enregistre la liste [image] dans le dossier C sous le nom in.txt, pour pouvoir partager des informations entre C et python
def save_img_c(image: list) -> None:
    f = open(f"../c/in.txt","w")
    for i in image:
        f.write(str(list(i)).replace(" ","").replace("[","").replace("]","") + "\n")
    f.close()


#donne les intervalles de déb fin d'un segment de droite
def intervalle_une_droite(r: float, theta: float, plan: list, precision_h: int, blanc: int = BLANC) -> list:
    (a,b) = droite(r, theta, precision_h)
    largeur = len(plan) #axe des y
    longueur = len(plan[0]) #axe des x
    segments = []
    x, deb, fin = 0, 0, 0
    while x < longueur :
        y = round(a*x + b)
        if 0 < y < largeur :
            if plan[y][x] >= blanc :
                deb = x, y
                while (0 < y < largeur) and x < longueur and (plan[y][x] >= blanc) :
                    x += 1
                    y = round(a*x +b)
                fin = x, y
                segments.append((deb, fin))
            else :
                x += 1
        else :
            x += 1
    return(segments)


# Donne les intervalles (début,fin) d'une liste de droites
# in - listes_droites : couple (r, theta), plan: list
# out - [[(deb, fin) (...) (deb,fin)] [...] [(deb, fin) (...) (deb, fin)]
def intervalle(liste_droites: list, plan, precision_h:int) -> list:
    n = len(liste_droites)
    liste_intervalles =[]
```

```python
    for i in range (n):
        r, theta = liste_droites[i][0], liste_droites[i][1]
        segments = intervalle_une_droite(r, theta, plan, precision_h)
        liste_intervalles.append(segments)
    return(liste_intervalles)


# raccorde des segments qui sont peu éloignés (a une distance inferieur a distance_min) pour en faire une droite
def raccordement_un_intervalle(segments: list, distance_min:int) -> list:
    new_segments = []
    retry = False
    i = 0
    while i < len(segments) :
        if (i < len(segments) -1 ) and (distance(segments[i][1], segments[i+1][0]) < distance_min) :
            retry = True
            new_segments.append((segments[i][0], segments[i+1][1]))
            i += 1
        else :
            new_segments.append(segments[i])
        i += 1
    if retry :
        return(raccordement_un_intervalle(new_segments, distance_min))
    return(new_segments)


# raccorde des segments pour une liste d'intervalles d'une liste de droites
def raccordement(liste_segments: list, distance_min:int = 50) -> list:
    new_segments = []
    for i in liste_segments:
        new_segments.append(raccordement_un_intervalle(i, distance_min=distance_min))
    return(new_segments)


#enlève les intervalles trop petits, qui sont probablement des erreurs
def enlever_petits_intervalles_une_droite(segments: list, longueur_min: int) -> list:
    new_segments =[]
    for i in range (len(segments)):
        if(distance(segments[i][0], segments[i][1])) > longueur_min:
           new_segments.append(segments[i])
    return(new_segments)

```

```python

# applique la fonction enlever_petits_intervalles_une_droite sur chaque segement
def enlever_petits_intervalles_general(liste_segments: list, longueur_min:int = 150) -> list:
    new_segments =[]
    for i in liste_segments:
        new_segments.append(enlever_petits_intervalles_une_droite(i, longueur_min))
    return(new_segments)


# Retourne l'ensemble des points d'intersection entre deux segments
def intersection_deux_segments(segment1: tuple, segment2: tuple) -> tuple:
    ((xdeb1,ydeb1),(xfin1,yfin1)),((xdeb2,ydeb2),(xfin2,yfin2)) = segment1,segment2
    (a1,b1), (a2,b2) = equation((xdeb1,ydeb1),(xfin1,yfin1)), equation((xdeb2,ydeb2),(xfin2,yfin2))
    if a1 != a2:
        x_sol = (b2 - b1) / (a1 - a2)
        y_sol = a1 * x_sol + b1
        if ((xdeb1 < x_sol < xfin1) or (xfin1 < x_sol < xdeb1)) and ((xdeb2 < x_sol < xfin2) or (xfin2 < x_sol < xdeb2))  and ((ydeb1 < y_sol < yfin1) or (yfin1 < y_sol < ydeb1)) and ((ydeb2 < y_sol < yfin2) or (yfin2 < y_sol < ydeb2)):
            return (x_sol, y_sol)

# calcule toutes les intersections entre une liste de segments
# type : [[droite i] [...] [droite j] ] -> ((droite i, droite j), (x, y))
# avec (x,y) : les coordonnées du point d'intersection
def intersection_general(liste_segments):
    n = len(liste_segments)
    liste_intersection = []
    for i in range (n):
        for j in range (n):
            if i > j: # permet d'eviter les doublons. La matrice est alors inferieur gauche.
                try:
                    x,y = intersection_deux_segments(liste_segments[i],liste_segments[j])
                    liste_intersection.append(((liste_segments[i], liste_segments[j]), (x,y)))
                except Exception as e:
                    # Il peut ne pas y avoir d'intersection entre ces deux segements, la fct intersection_deux_segments ne renvoie
                    # rien, on a donc une erreur et c'est pour cela qu'on passe.
                    pass

    return (liste_intersection)


# transforme la une liste de liste en liste d'elements
```

```python
def flatten(liste: list) -> list:
    flat_list = []
    for sublist in liste:
        for item in sublist:
            flat_list.append(item)
    return(flat_list)


# calcule la matrice d'intersections
# renvoie : matrice[i][j] =(x,y) <=> (x,y) est le point d'intersection entre le ième et le jème segment de [liste_segments]
def matrice_intersections(liste_intersections: list, liste_segments: list):
    len_segments = len(liste_segments)                                  # nombres de routes
    len_intersection = len(liste_intersections)                         # nombre d'intersections
    m = [[ (-1,-1) for i in range(len_segments)] for j in range(len_segments)]
    for z in range (len_intersection):
        (droite_1,droite_2), (x,y) = liste_intersections[z]
        index_i = liste_segments.index(droite_1)
        index_j = liste_segments.index(droite_2)
        m[index_i][index_j] = (x,y)
    return(m)


# renvoie la matrice d'adjacence correspondant au graphe de notre ville
# on definit m[i][i] comme les coordonnées du point i et m[i][j] la distance entre le sommet i et le sommet j si ils sont relié. sinon, -1
# add_str permet de modifier le nom du fichier
def mat_adjacence(mat_intersections: list, liste_intersections: list, img_NB: list, img: list, add_str: str = ""):
    liste_pts = list(set([x[1] for x in liste_intersections]))              # enlève les doublons
    n_r = len(liste_pts)                                                    # nb de routes
    n_i = len(mat_intersections[0])                                         # nb de sommets
    graphe_return = [[-1 for y in range(n_r)] for x in range(n_r)]
    for i in range(n_r):
        graphe_return[i][i] = liste_pts[i]                                  # definition de m[i][i]
        for j in range(n_r):
            if i > j:
                if voisin(liste_pts[i], liste_pts[j], img_NB):
                    d = int(distance(liste_pts[i], liste_pts[j]))
                    graphe_return[i][j] = d
                    graphe_return[j][i] = d
                    plt.plot((liste_pts[i][0], liste_pts[j][0]),(liste_pts[i][1], liste_pts[j][1]) )
    return(graphe_return)
```

```python


# renvoie si a et b sont des voisins.
# pour cela, on verifie si il y a plus de [p]% de points de route entre les deux
def voisin(pt1: tuple, pt2 : tuple, img : list,  p: int = 60) -> tuple:
    a, b = equation(pt1, pt2)
    pts = [x for x in range(int(min(pt1[0], pt2[0])),int(max(pt1[0], pt2[0])))] # valeurs que doit prendre x sur le trajet
    nb_tot = len(pts)
    def f(x):                                       # équation de la droite
        return(int(a*x+b))
    c = 0                                           # compteur de points blanc
    for x in pts:                                   # on compte le nombre de points blanc entre les deux points
        if f(x) < len(img):
            if img[f(x)][x] == BLANC :
                c += 1
    if nb_tot == 0 :
        return(False)
    return((c/(nb_tot)*100)>p)


# Applique l'algorithme de dijkstra à une matrice d'adjacence et un sommet de début donné
# Renvoie la liste des distances du sommet de départ à tout les autres
#         la liste des sommets qui ont découvert t[i] (liste des parents)
def dijkstra(mat_adj: list, sommet_deb:int, sommet_fin: int):
    vue = [sommet_deb]                                                  # liste des sommets déjà vu
    n = len(mat_adj)
    dist = [ -1 for i in range(n)]                                      # liste des distances
    dist[sommet_deb] = 0
    liste_parents = [ -1 for i in range(n)]
    nb = 0
    for _ in range(n-1):
        nb += 1
        mini = MAX_INT                                                  # distance minimum des voisins
        sommet_min = -1
        a_decouvert = 0
        for cur in range(n):                                            # on parcours les sommets
            if cur in vue :                                             # si le sommet a déjà été vu
                for vois in range(n):                                   # on parcours ses voisins
                    if mat_adj[cur][vois] != -1 and not (vois in vue):      # si il est de distance inferieure
                        if mat_adj[cur][vois] + dist[cur] < mini :          # on le définit comme meilleur candidat et on continue
                            mini = mat_adj[cur][vois] + dist[cur]
```

```
                            sommet_min = vois
                            a_decouvert = cur
        liste_parents[sommet_min] = a_decouvert                             # on conserve le meilleur candidat
        dist[sommet_min] = mini
        vue.append(sommet_min)
        if sommet_min == sommet_fin:
            break
    return(liste_parents,dist, nb)

# Renvoie la liste des sommets à parcourir pour aller du sommet de début au sommet de fin à partir du tableau de dijkstra
# est appelé avec la liste des pères renvoyé par dijkstra
def chemin(liste_decouvre: list, sommet_fin:int) -> list:
    liste_sommet = []
    i = sommet_fin
    while liste_decouvre[i] != -1 :             # -1 est le sommet de départ
        liste_sommet.append(i)                  # on est sensé ajouter les elements au début de la liste. On les ajoutes à la fin, puis on retourne la liste
        i = liste_decouvre[i]
    liste_sommet.append(i)
    liste_sommet = liste_sommet[::-1]
    return(liste_sommet)


# Renvoie la liste des sommets à parcourir à partir du graphe d'adj,
# le sommet de deb, repéré par son indice dans la matrice et le sommet de fin, repéré de même
def plus_court_chemin_dijkstra(mat_adj: list, sommet_deb:int, sommet_fin:int) -> list:
    start_dijkstra = time()
    liste_dec, _, nb_sommets = dijkstra(mat_adj, sommet_deb, sommet_fin)
    pcc = chemin(liste_dec,sommet_fin)
    return(pcc, nb_sommets, (time() - start_dijkstra))


# Algorithme A*
# [mat_adj] : matrice d'adjacence avec t[i][i] les coordonnées du point
# [h] heuristique : fonction d'estimation de la distance
# [s] sommet de depart
# [t] sommet d'arrivé
def a_etoile(mat_adj, h, s, t):
    n = len(mat_adj[0]) # nb de sommets
    f = FP()
    f.push(s, 0)
```

```
    liste_distance = [ MAX_INT for i in range(n)]
    liste_distance[s] = 0
    nb = 0
    while not(f.is_empty()):
        nb +=1
        v = f.pop()
        if v == t :                                                    # si on a atteint le sommet voulu
            return(reconstruire(liste_distance, mat_adj, t), nb)
        else :                                                         # sinon on s'applique sur chaque voisins
            for u in voisins_a(v, mat_adj):
                if liste_distance[u] > liste_distance[v] + mat_adj[v][u] :
                    liste_distance[u] = liste_distance[v] + mat_adj[v][u]
                    f.push(u, liste_distance[u] + h(mat_adj, u, t))


# voisins pour la fonction A*
# s est le sommet que l'on considère
def voisins_a(s: int, matrice_adj: list) -> list:
    n = len(matrice_adj)
    l = []                                                  # liste des voisins
    for i in range(n):
        if i != s and matrice_adj[s][i] != -1:    # si [i] et [s] sont voisin dans la matrice d'adjacence
            l.append(i)
    return(l)


# reconstuit le chemin de t a s, pour la fonction A*
# d est la liste des distances entre le sommet original et le sommet d[i]
# t est le sommet d'arrivée
def reconstruire(d: list, mat_adj: list, t: int) -> list:
    r = d[t]                                    # r est la distance restante avec le sommet de depart
    chemin = [t]                                # on initialise le chemin avec le sommet d'arrivé
    while r!=0 :                                # tant qu'il reste de la distance jusqu'au sommet de de depart
        for u in range(len(mat_adj)):           # on parcours tout les voisind de t
            if u!=t :                           # -
                if r == d[u] + mat_adj[u][t] :  # si la distance restante est la distance entre le sommet u et le sommet de
                    chemin.append(u)            # départ + la distance entre u et le sommet actuel
                    r = r - mat_adj[u][t]       # on diminue la distance
                    t = u                       # on passe par ce sommet et on recommence l'algorithme
                    break
    chemin.reverse()
```

```
444     return(chemin)
445
446
447 # Heuristique pour A*
448 # On utilise la distance à vol d'oiseau.
449 # C'est bien une heuristique acceptante.
450 def heuristique(mat_adj: list, sommet_act:int , sommet_fin:int) -> int:
451     x1,y1 = mat_adj[sommet_act][sommet_act]
452     x2,y2 = mat_adj[sommet_fin][sommet_fin]
453     dist_min = distance((x1,y1), (x2,y2))
454     return (dist_min)
455
456
457 # determine le nombre de points blanc d'une droite, en pouvant avancer de d points sans reset le compte
458 # droite est donnée sous la forme d'un couple (a, b)
459 # l'image dois être l'image en noir et blanc
460 # [d] est la distance maximale entre deux points blancs
461 def nombre_points_blanc(droite: tuple, img: list, d: int = MAX_INT) -> int:
462     n = len(img)        # dimension de l'image
463     m = len(img[1])     # dimension de l'image
464     distance = d        # distance max
465     liste_dist = []     # liste des suites de points blanc
466     (a,b) = droite
467     nmbre_points = 0    # nombre de points blanc
468     for x in range (m):
469         y = a*x+b
470         if (y<n) and (y>=0):                        # si on est bien dans l'image
471             if distance == 0 :                      # si on a trop avancé dans l'image sans croiser de points blancs
472                 liste_dist.append(nmbre_points)     #
473                 nmbre_points = 0                    # on reset le compte de points
474                 distance = d                        # on reset la distance
475             if (img[int(y)][int(x)] >= BLANC):      # si le pixel est blanc
476                 nmbre_points+=1                     # on incremente la valeur du nb de points
477                 distance = d                        # on reset la distance
478             else :
479                 distance -= 1                       # on decremente la distance
480     liste_dist.append(nmbre_points)
481     return(max(liste_dist))     # valeur maximale d'une suite de points blanc
482
483
484 # fusionne les droites similaires
```

```python
# liste est la liste des droites parametrée selon r et theta
# dico est un dictionnaire représentant les droites traitées et non traitées
# nb_lim est le nombre de pixel maximum de difference permettant la fusion de deux droites
# a_fact et b_fact sont les facteurs permettant de choisir les droites a prendres en comptes entre elles, en parametrage carthesien
def fusion_droites (liste: list, imageNB, dico: dict, a_fact: float, b_fact: float, nb_lim: int, precision_hough: int) -> list:
    nouvelle_liste = []
    modif = False
    for i in range(len(liste)):
        (r1,theta1) = liste[i]
        for j in range(1, len(liste)):
            (r2, theta2) = liste[j]
            a1, b1 = droite(r1, theta1, precision_hough)
            a2, b2 = droite(r2, theta2, precision_hough)
            if r1 == r2 and theta1==theta2:                # on ne se considere pas soit même
                pass
            else :
                if (dico[(r1, theta1)]!=1) and (dico[(r2, theta2)]!=1): # on a déjà traiter la droite
                    min_a = min(abs(a1), abs(a2))
                    moy_a = abs(a1 + a2)/2
                    a_lim = a_fact + a_fact * min_a**2
                    if (abs(a1-a2) < a_lim):                                # si les droites sont suffisament proche selon les a
                        b_lim = b_fact + moy_a**2 * b_fact
                        if abs(b1 - b2) < b_lim :                           # si les droites sont suffisament proche selon des b
                            n1 = nombre_points_blanc((a1,b1), imageNB, 5)
                            n2 = nombre_points_blanc((a2,b2), imageNB, 5)
                            if (abs(n1-n2) < nb_lim):                                   # on conserve la droite moyenne entre les deux
                                ntot = n1 + n2                                          # --
                                r_moy = (r1 + r2)/2                                     # --
                                theta_moy = (theta1 + theta2)/2                         # --
                                nouvelle_liste.append((r_moy, theta_moy))               # --
                            elif ( n1 > n2 ):                                           # on conserve uniquement la droite 1
                                nouvelle_liste.append((r1, theta1))                     # --
                            else :                                                      # on conserve uniquement la droite 1
                                nouvelle_liste.append((r2, theta2))                     # --
                            modif = True
                            dico[(r1, theta1)] = 1
                            dico[(r2, theta2)] = 1
        if dico[(r1,theta1)]!=1:
            nouvelle_liste.append((r1, theta1))
    return(list(set(nouvelle_liste)), modif, dico)

```

```python
# applique en boucle la fonction fusion_droites jusqu'à qu'il n'y ai plus de modifications
def fusion_droites_rec(liste: list, image: list , a_fact: int, b_fact: int, nb_lim: int, precision_hough: int) -> list:
    dico = {}                               # Cas initial
    for (r1,theta1) in liste:
        dico[(r1,theta1)] = 0
    nv_liste, booleen, dico = fusion_droites(liste, image, dico, a_fact, b_fact, nb_lim, precision_hough)

    while booleen:                          # Le booleen représente si une modification a été faite ou non
        dico = {}                           # si c'est le cas on recommence
        for (r1,theta1) in nv_liste:
            dico[(r1,theta1)] = 0
        nv_liste, booleen, dico = fusion_droites(nv_liste, image, dico, a_fact, b_fact, nb_lim, precision_hough)
    return(nv_liste)


# trie une liste selon la deuxieme coordonnée
def sort_list_theta(liste: list) -> list:
    liste = [(x[1], x[0]) for x in liste]
    liste.sort()
    return([(x[1], x[0]) for x in liste])


# Permet de fusionner les intersections qui sont proches les une des autres, à partir de la matrice d'adjacence.
# distance_max est la distance à partir de laquelle on arrête de fusionner les droites.
def cluster(matrice_adj: list, distance_max: int) -> list:
    for i in range(len(matrice_adj)):                                                   # on parcours l'intégralité des sommets
        for j in range(i+1, len(matrice_adj)):
            point1 = matrice_adj[i][i]
            point2 = matrice_adj[j][j]
            x1, y1 = point1
            x2, y2 = point2
            if distance(point1, point2) < distance_max and (x1,y1) != (-1, -1) and (x2,y2) != (-1, -1): # on supprime la valeur j et on introduit les anciennes valeurs dans i
                matrice_adj[i][i] = (int((x1+x2)/2), int((y1+y2)/2))                    # la coordonnée du point est la moyenne des deux
                matrice_adj[j][j] = (-1, -1)
                for k in range(len(matrice_adj)):
                    if k!=i and k != j:
                        if matrice_adj[j][k] != -1:
                            if matrice_adj[i][k] != -1:
                                matrice_adj[i][k] = (matrice_adj[i][k] + matrice_adj[j][k])/2
```

```
                                matrice_adj[k][i] = (matrice_adj[k][i] + matrice_adj[k][j])/2
                            else :
                                matrice_adj[i][k] = matrice_adj[j][k]
                                matrice_adj[k][i] = matrice_adj[k][j]
                        matrice_adj[j][k] = -1
                        matrice_adj[k][j] = -1
                return(cluster(matrice_adj, distance_max))
    return(matrice_adj)


def matrice_recadre(points, mat):
    belle_matrice = [[0 for i in range(len(points))] for i in range(len(points))]
    for i in range(len(points)):
        for j in range(len(points)):
            belle_matrice[i][j] = mat[points[i]][points[j]]
    return(belle_matrice)



def main(lien: str, precision_hough:int = 1):
    print("Lecture de l'image.") # uniquement a partir de plan
    image = read_img(lien)
    print("Conversion en niveau de gris.")
    image_grayscale = color_to_grayscale(image)
    print("Conversion en noir et blanc.")
    image_NB = grayscale_to_black_and_white(image_grayscale)
    print("Calcul de l'espace de Hough.")
    espace_h = hough_c(image_NB, precision_hough)
    save_hough(espace_h)
    print("Récuperation des droites.")
    liste_droites1 = peak_local_max(np.array(espace_h), min_distance=70, num_peaks=500, threshold_rel=0.23)
    print(f"Nombre de droites : {len(liste_droites1)}.")
    liste_droites1 = [list(x) for x in list(liste_droites1)]
    save_with_droites(image, liste_droites1, precision_hough, "01. toutes les droites")
    liste_droites2 = fusion_droites_rec(sort_list_theta(liste_droites1), image_NB,   0.5, 50, 10, precision_hough)
    save_with_droites(image, liste_droites2, precision_hough, "02. premier filtrage des droites")
    liste_droites3 = fusion_droites_rec(sort_list_theta(liste_droites2), image_NB,   0.3, 75, 10, precision_hough)
    save_with_droites(image, liste_droites3, precision_hough, "03. deuxieme filtrage des droites")
    print("Récupération de la listes des intervalles.")
    liste_intervalles = intervalle(liste_droites3, image_NB, precision_hough)
    save_liste_inter(liste_intervalles, image, "04. intervalles")
```

```
    print("Raccorde les droites.")
    liste_segments = raccordement(liste_intervalles, 28)
    save_liste_inter(liste_segments, image, "05. raccordement (28)")
    print("enlève les petits bouts de droite")
    liste_segments = enlever_petits_intervalles_general(liste_segments)
    save_liste_inter(liste_segments, image, "06. segments (28)")
    liste_segments = flatten(liste_segments)
    liste_intersections = intersection_general(liste_segments)
    save_intersection(liste_intersections, image)
    print("transformation en matrice d'intersections")
    matrice_inter = matrice_intersections(liste_intersections, liste_segments)
    matr_adj = mat_adjacence(matrice_inter, liste_intersections, image_NB, image)
    print("clusterisation")
    matrice_adj_cluster = cluster(matr_adj, 28)
    save_cluster(matrice_adj_cluster, image)
    save_graphe(matrice_adj_cluster, image)
    #save_full_graphe(matrice_adj_cluster, image)


    sommet_depart = 63
    sommet_arrivee = 61
    # d'autres valeurs intéressantes sont possibles : 61->73 ; 1->24


    chemin_dijksra, nb_sommets_dijkstra, duree_dijksra = plus_court_chemin_dijkstra(matr_adj, sommet_depart, sommet_arrivee)

    start_a_star = time()
    chemin_a_star, nb_sommets_a_star = a_etoile(matr_adj, heuristique, sommet_depart, sommet_arrivee)
    duree_a_star = (time() - start_a_star)

    print(f"chemin dijkra : {chemin_dijksra}, temps d'execution : {duree_dijksra}, sommets parcourus : {nb_sommets_dijkstra}")
    print(f"chemin A*: {chemin_a_star}, temps d'execution : {duree_a_star}, sommets parcourus : {nb_sommets_a_star}")
    save_chemin(chemin_dijksra, matrice_adj_cluster, image, "dijkstra", "g")
    save_chemin(chemin_a_star, matrice_adj_cluster, image, "a star", "k")

    return(matrice_adj_cluster)



mat = main("https://server.ip/TIPE/paris2.png", 100)
```

```
647
648
649 #points = [25, 41, 3, 14, 52, 53, 54]
650 print(matrice_recadre([25, 41, 3, 14, 52, 53, 54], mat))
651
```