

Tipe Pierre TELLIER, n°30166

arborescence

```
├── Python Files
│   ├── images.py
│   ├── kmoy.py
│   ├── main.py
│   └── modules.py
├── c
│   ├── hough.c
│   └── hough_multi.c
├── images
│   └── in.png
```

Python Files\main.py

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import imageio as io
4 from time import time
5 from random import choice
6 from math import pi, cos, sin, floor
7 from skimage.feature import peak_local_max
8 from os import system
9 import subprocess
10
11 from modules import *
12 # importation les fonctions permettant d'enregistrer des images
13 from images import *
14
15 ## Variables globales
16
17 BLANC = 254 # définit le seuil pour la couleur blanche
18 MAX_INT = 42e42
19 ## Liens des images sources
20
21
22 # Fonctions
23 ## Fonction de base sur l'image
24
25 # Inverse l'image selon la coordonnée y
26 def inv_img_y(img: list) -> list:
27     return(img[::-1])
28
29 # lecture de l'image liée au lien [lien] depuis le serveur web
30 def read_img(lien: str) -> np.array:
31     im = io.v2.imread(lien)
32     im = np.array(im)
33     return(inv_img_y(im))
34
35
36 # affiche l'image [img].
37 # [cmap='gray'] permet d'afficher correctement les images en noir et blanc ainsi que les images en couleur
```

```

38 # pour les images en niveaux de gris, le pixel le plus sombre sera noir et le pixel le plus clair sera blanc.
39 # [rev] premet d'inverser l'image
40 def show_image(img: list, rev: bool = False) -> None:
41     plt.axis('off') # enlève les axes
42     if rev :
43         plt.imshow(img, cmap='gray',origin='lower', aspect='auto')
44     else :
45         plt.imshow(img, cmap='gray', aspect='auto')
46
47
48 ## Traitement des images
49
50 # color_to_grayscale [image] convertis l'image [image] en RGB en une image (np.array) en niveau de gris
51 def color_to_grayscale(image: list) -> np.array:
52     image = list(image)
53     r = [[] for x in range(len(image))] #image vide
54     for i in range (len(image)) :
55         for j in range(len(image[i])) :
56             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
57             r[i].append(valeur)
58
59     return np.array(r)
60
61 # grayscale_to_black_and_white img prend en entrée l'image [img] et renvoie un image en noir et blanc. les pixels
62 # de couleur superieure à [seuil] vont être laissé dans leur couleur originale et ceux inferieur a seuil vont être mis en noir
63 # par default, la fonction conserve uniquement les pixels parfaitement blanc
64 def grayscale_to_black_and_white(img: np.array, seuil:int = BLANC) -> np.array:
65     r = [[] for x in range(len(img))]
66     for i in range(len(img)):
67         for j in range(len(img[i])):
68             if img[i][j] < seuil :
69                 r[i].append(0)
70             else :
71                 r[i].append(int(img[i][j]))
72     return(np.array(r))
73
74 ## Matrice de Hough
75
76 # retourne la taille du tableau qui represente l'espace de hough.
77 # l'option precision change la taille du tableau selon la dimation x, c'est a dire selon les theta

```

```

78 def taille_Hough(img: list, precision: int = 1) -> tuple:
79     Px = len(img)
80     Py = len(img[0])
81     hauteur = (Px ** 2 + Py ** 2)**0.5 # distance maximum (coin en bas à gauche)
82     largeur = 360
83     return(round(hauteur), largeur*precision)
84
85
86 # rho_f(x, y, θ) retourne la distance a l'origine de la droite passant par le point (x, y) et d'angle θ
87 def rho_f(x: int, y: int, θ: float) -> float:
88     return(x*cos(θ) + y*sin(θ))
89
90
91 #retourne la matrice de l'espace Hough de l'image img dans l'ordre m[r][theta]
92 def espace_Hough(img: list, precision: int = 1) -> list:
93     hauteur_H, largeur_H = taille_Hough(img, precision)
94     hauteur_I, largeur_I = len(img), len(img[0])
95     #creer le tableau qui va contenir la matrice
96     matrice_hough = [[0 for i in range (largeur_H)] for i in range(hauteur_H)]
97     #on parcourt toute l'image
98     for y in range(hauteur_I):
99         for x in range(largeur_I):
100             if img[y][x] >= BLANC:
101                 for theta_deg in range(largeur_H):
102                     theta_rad = theta_deg / (180 * precision) * pi
103                     rho = round(rho_f(x, y, theta_rad))
104                     if rho > 0: #comme on fait varier θ jusqu'a 2π, on ne garde que les valeurs positives
105                         matrice_hough[rho][theta_deg] += 1 #on met l'origine du repere en bas de l'image
106     return(matrice_hough)
107
108
109 # retourne une liste de maximum locaux.
110 def maximum_global(m: list, d: int = 100):
111     return(peak_local_max(np.array(m), min_distance=d))
112
113
114 ## Géométrie
115
116 # calcule l'équation d'une droite passant par deux points en coordonnées carthésiennes.
117 # la valeur de retour sera un couple (a, b) représentant la droite sous forme y = ax + b
118 def equation(point1: tuple, point2: tuple) -> tuple:

```

```

119     (x1,y1),(x2,y2) = point1,point2
120     assert(x1 != x2), "les points sont alignés, la droite ne peut pas être prise en compte"
121     a = (y1 - y2) / (x1 - x2) #si la droite n'est pas verticale
122     b = y2 - a*x2
123     return(a,b)
124
125
126 # retourne les coefficients de l'equation de la droite représentée par une distance a l'origine r et un angle theta
127 # sous la forme d'un couple (a, b) représentant la droite sous forme y = ax + b
128 def droite(r: float, theta: float, precision: int = 1) -> tuple:
129     theta = theta / precision
130     if theta == 0 or theta == 180:
131         return (0,r)
132     else :
133         theta_rad = (pi/180)*theta
134         a = -(cos(theta_rad)/sin(theta_rad))
135         b = r/sin(theta_rad)
136     return(a,b)
137
138 # Renvoie la distance euclidienne entre les points a: (x1, y1) et b : (x2, y2)
139 def distance (a: tuple, b: tuple) -> int:
140     (x1,y1),(x2,y2) = a,b
141     distance = ( (x1-x2)**2 + (y1 - y2)**2 )**0.5
142     return(distance)
143
144
145 # calcule la matrice de Hough a l'aide de d'un programme en C qui utilise du multithreading.
146 # permet une vitesse d'execution beaucoup plus élevée : temps d'exécution divisé par 100
147 # l'argument bypass permet de tester d'autres partie du code plus rapidement
148 def hough_c(image: list, precision: int = 10, bypass: bool = False) -> list:
149     if not bypass:
150         save_img_c(image)
151         subprocess.run(['../c/multi.out', str(precision)])
152     mat = read_c()
153     return(mat)
154
155 # renvoie la matrice renvoyée par le calcul de l'espace de hough en C
156 def read_c() -> list:
157     f = open("../c/out.txt","r")
158     m_read = f.readlines()
159     f.close()

```

```

160     return([[int (x) for x in y.split(",")] for y in m_read])
161
162
163 # enregistre la liste [image] dans le dossier C sous le nom in.txt, pour pouvoir partager des informations entre C et python
164 def save_img_c(image: list) -> None:
165     f = open(f"../c/in.txt", "w")
166     for i in image:
167         f.write(str(list(i)).replace(" ", "").replace("[", "").replace("]", "") + "\n")
168     f.close()
169
170
171 #donne les intervalles de déb fin d'un segment de droite
172 def intervalle_une_droite(r: float, theta: float, plan: list, precision_h: int, blanc: int = BLANC) -> list:
173     (a,b) = droite(r, theta, precision_h)
174     largeur = len(plan) #axe des y
175     longueur = len(plan[0]) #axe des x
176     segments = []
177     x, deb, fin = 0, 0, 0
178     while x < longueur :
179         y = round(a*x + b)
180         if 0 < y < largeur :
181             if plan[y][x] >= blanc :
182                 deb = x, y
183                 while (0 < y < largeur) and x < longueur and (plan[y][x] >= blanc) :
184                     x += 1
185                     y = round(a*x +b)
186                 fin = x, y
187                 segments.append((deb, fin))
188             else :
189                 x += 1
190         else :
191             x += 1
192     return(segments)
193
194
195 # Donne les intervalles (début,fin) d'une liste de droites
196 # in - listes_droites : couple (r, theta), plan: list
197 # out - [(deb, fin) (...) (deb,fin)] [...] [(deb, fin) (...) (deb, fin)]
198 def intervalle(liste_droites: list, plan, precision_h:int) -> list:
199     n = len(liste_droites)
200     liste_intervalles =[]

```

```

201     for i in range (n):
202         r, theta = liste_droites[i][0], liste_droites[i][1]
203         segments = intervalle_une_droite(r, theta, plan, precision_h)
204         liste_intervalles.append(segments)
205     return(liste_intervalles)
206
207
208 # raccorde des segments qui sont peu éloignés (a une distance inferieur a distance_min) pour en faire une droite
209 def raccordement_un_intervalle(segments: list, distance_min:int) -> list:
210     new_segments = []
211     retry = False
212     i = 0
213     while i < len(segments) :
214         if (i < len(segments) -1 ) and (distance(segments[i][1], segments[i+1][0]) < distance_min) :
215             retry = True
216             new_segments.append((segments[i][0], segments[i+1][1]))
217             i += 1
218         else :
219             new_segments.append(segments[i])
220             i += 1
221     if retry :
222         return(raccordement_un_intervalle(new_segments, distance_min))
223     return(new_segments)
224
225
226 # raccorde des segments pour une liste d'intervalles d'une liste de droites
227 def raccordement(liste_segments: list, distance_min:int = 50) -> list:
228     new_segments = []
229     for i in liste_segments:
230         new_segments.append(raccordement_un_intervalle(i, distance_min=distance_min))
231     return(new_segments)
232
233
234 #enlève les intervalles trop petits, qui sont probablement des erreurs
235 def enlever_petits_intervalles_une_droite(segments: list, longueur_min: int) -> list:
236     new_segments =[]
237     for i in range (len(segments)):
238         if(distance(segments[i][0], segments[i][1])) > longueur_min:
239             new_segments.append(segments[i])
240     return(new_segments)
241

```

```

242
243 # applique la fonction enlever_petits_intervalles_une_droite sur chaque segment
244 def enlever_petits_intervalles_general(liste_segments: list, longueur_min:int = 150) -> list:
245     new_segments =[]
246     for i in liste_segments:
247         new_segments.append(enlever_petits_intervalles_une_droite(i, longueur_min))
248     return(new_segments)
249
250
251 # Retourne l'ensemble des points d'intersection entre deux segments
252 def intersection_deux_segments(segment1: tuple, segment2: tuple) -> tuple:
253     ((xdeb1,ydeb1),(xfin1,yfin1)),((xdeb2,ydeb2),(xfin2,yfin2)) = segment1,segment2
254     (a1,b1), (a2,b2) = equation((xdeb1,ydeb1),(xfin1,yfin1)), equation((xdeb2,ydeb2),(xfin2,yfin2))
255     if a1 != a2:
256         x_sol = (b2 - b1) / (a1 - a2)
257         y_sol = a1 * x_sol + b1
258         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)):
259             return (x_sol, y_sol)
260
261 # calcule toutes les intersections entre une liste de segments
262 # type : [[droite i] [...] [droite j] ] -> ((droite i, droite j), (x, y))
263 # avec (x,y) : les coordonnées du point d'intersection
264 def intersection_general(liste_segments):
265     n = len(liste_segments)
266     liste_intersection = []
267     for i in range (n):
268         for j in range (n):
269             if i > j: # permet d'eviter les doublons. La matrice est alors inferieur gauche.
270                 try:
271                     x,y = intersection_deux_segments(liste_segments[i],liste_segments[j])
272                     liste_intersection.append(((liste_segments[i], liste_segments[j]), (x,y)))
273                 except Exception as e:
274                     # Il peut ne pas y avoir d'intersection entre ces deux segments, la fct intersection_deux_segments ne renvoie
275                     # rien, on a donc une erreur et c'est pour cela qu'on passe.
276                     pass
277
278     return (liste_intersection)
279
280
281 # transforme la une liste de liste en liste d'elements

```



```

282 def flatten(liste: list) -> list:
283     flat_list = []
284     for sublist in liste:
285         for item in sublist:
286             flat_list.append(item)
287     return(flat_list)
288
289
290 # calcule la matrice d'intersections
291 # renvoie : matrice[i][j] =(x,y) <=> (x,y) est le point d'intersection entre le ième et le jème segment de [liste_segments]
292 def matrice_intersections(liste_intersections: list, liste_segments: list):
293     len_segments = len(liste_segments) # nombres de routes
294     len_intersection = len(liste_intersections) # nombre d'intersections
295     m = [[ (-1,-1) for i in range(len_segments)] for j in range(len_segments)]
296     for z in range (len_intersection):
297         (droite_1,droite_2), (x,y) = liste_intersections[z]
298         index_i = liste_segments.index(droite_1)
299         index_j = liste_segments.index(droite_2)
300         m[index_i][index_j] = (x,y)
301     return(m)
302
303
304 # renvoie la matrice d'adjacence correspondant au graphe de notre ville
305 # 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,
306 # -1
307 # add_str permet de modifier le nom du fichier
308 def mat_adjacence(mat_intersections: list, liste_intersections: list, img_NB: list, img: list, add_str: str = ""):
309     liste_pts = list(set([x[1] for x in liste_intersections])) # enlève les doublons
310     n_r = len(liste_pts) # nb de routes
311     n_i = len(mat_intersections[0]) # nb de sommets
312     graphe_return = [[-1 for y in range(n_r)] for x in range(n_r)]
313     for i in range(n_r):
314         graphe_return[i][i] = liste_pts[i] # definition de m[i][i]
315         for j in range(n_r):
316             if i > j:
317                 if voisin(liste_pts[i], liste_pts[j], img_NB):
318                     d = int(distance(liste_pts[i], liste_pts[j]))
319                     graphe_return[i][j] = d
320                     graphe_return[j][i] = d
321                     plt.plot((liste_pts[i][0], liste_pts[j][0]),(liste_pts[i][1], liste_pts[j][1]) )
322     return(graphe_return)

```

```

322
323
324 # renvoie si a et b sont des voisins.
325 # pour cela, on verifie si il y a plus de [p]% de points de route entre les deux
326 def voisin(pt1: tuple, pt2 : tuple, img : list, p: int = 60) -> tuple:
327     a, b = equation(pt1, pt2)
328     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
329     nb_tot = len(pts)
330     def f(x):                                     # équation de la droite
331         return(int(a*x+b))
332     c = 0                                         # compteur de points blanc
333     for x in pts:                                 # on compte le nombre de points blanc entre les deux points
334         if f(x) < len(img):
335             if img[f(x)][x] == BLANC :
336                 c += 1
337     if nb_tot == 0 :
338         return(False)
339     return((c/(nb_tot)*100)>p)
340
341
342 # Applique l'algorithme de dijkstra à une matrice d'adjacence et un sommet de début donné
343 # Renvoie la liste des distances du sommet de départ à tout les autres
344 #     la liste des sommets qui ont découvert t[i] (liste des parents)
345 def dijkstra(mat_adj: list, sommet_deb:int, sommet_fin: int):
346     vue = [sommet_deb]                           # liste des sommets déjà vu
347     n = len(mat_adj)
348     dist = [ -1 for i in range(n)]               # liste des distances
349     dist[sommet_deb] = 0
350     liste_parents = [ -1 for i in range(n)]
351     nb = 0
352     for _ in range(n-1):
353         nb += 1
354         mini = MAX_INT                           # distance minimum des voisins
355         sommet_min = -1
356         a_decouvert = 0
357         for cur in range(n):                     # on parcourt les sommets
358             if cur in vue :                      # si le sommet a déjà été vu
359                 for vois in range(n):           # on parcourt ses voisins
360                     if mat_adj[cur][vois] != -1 and not (vois in vue): # si il est de distance inferieure
361                         if mat_adj[cur][vois] + dist[cur] < mini : # on le définit comme meilleur candidat et on continue
362                             mini = mat_adj[cur][vois] + dist[cur]

```

```

363         sommet_min = vois
364         a_decouvert = cur
365         liste_parents[sommet_min] = a_decouvert           # on conserve le meilleur candidat
366         dist[sommet_min] = mini
367         vue.append(sommet_min)
368         if sommet_min == sommet_fin:
369             break
370     return(liste_parents,dist, nb)
371
372 # Renvoie la liste des sommets à parcourir pour aller du sommet de début au sommet de fin à partir du tableau de dijkstra
373 # est appelé avec la liste des pères renvoyé par dijkstra
374 def chemin(liste_decouvre: list, sommet_fin:int) -> list:
375     liste_sommet = []
376     i = sommet_fin
377     while liste_decouvre[i] != -1 :           # -1 est le sommet de départ
378         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
379         i = liste_decouvre[i]
380     liste_sommet.append(i)
381     liste_sommet = liste_sommet[::-1]
382     return(liste_sommet)
383
384
385 # Renvoie la liste des sommets à parcourir à partir du graphe d'adj,
386 # le sommet de deb, repéré par son indice dans la matrice et le sommet de fin, repéré de même
387 def plus_court_chemin_dijkstra(mat_adj: list, sommet_deb:int, sommet_fin:int) -> list:
388     start_dijkstra = time()
389     liste_dec, _, nb_sommets = dijkstra(mat_adj, sommet_deb, sommet_fin)
390     pcc = chemin(liste_dec,sommet_fin)
391     return(pcc, nb_sommets, (time() - start_dijkstra))
392
393
394 # Algorithme A*
395 # [mat_adj] : matrice d'adjacence avec t[i][i] les coordonnées du point
396 # [h] heuristique : fonction d'estimation de la distance
397 # [s] sommet de depart
398 # [t] sommet d'arrivé
399 def a_etoile(mat_adj, h, s, t):
400     n = len(mat_adj[0]) # nb de sommets
401     f = FP()
402     f.push(s, 0)

```

```

403 liste_distance = [ MAX_INT for i in range(n)]
404 liste_distance[s] = 0
405 nb = 0
406 while not(f.is_empty()):
407     nb +=1
408     v = f.pop()
409     if v == t : # si on a atteint le sommet voulu
410         return(reconstruire(liste_distance, mat_adj, t), nb)
411     else : # sinon on s'applique sur chaque voisins
412         for u in voisins_a(v, mat_adj):
413             if liste_distance[u] > liste_distance[v] + mat_adj[v][u] :
414                 liste_distance[u] = liste_distance[v] + mat_adj[v][u]
415                 f.push(u, liste_distance[u] + h(mat_adj, u, t))
416
417
418 # voisins pour la fonction A*
419 # s est le sommet que l'on considère
420 def voisins_a(s: int, matrice_adj: list) -> list:
421     n = len(matrice_adj)
422     l = [] # liste des voisins
423     for i in range(n):
424         if i != s and matrice_adj[s][i] != -1: # si [i] et [s] sont voisin dans la matrice d'adjacence
425             l.append(i)
426     return(l)
427
428
429 # reconstitue le chemin de t a s, pour la fonction A*
430 # d est la liste des distances entre le sommet original et le sommet d[i]
431 # t est le sommet d'arrivée
432 def reconstruire(d: list, mat_adj: list, t: int) -> list:
433     r = d[t] # r est la distance restante avec le sommet de depart
434     chemin = [t] # on initialise le chemin avec le sommet d'arrivé
435     while r!=0 : # tant qu'il reste de la distance jusqu'au sommet de de depart
436         for u in range(len(mat_adj)): # on parcourt tout les voisins de t
437             if u!=t : # -
438                 if r == d[u] + mat_adj[u][t] : # si la distance restante est la distance entre le sommet u et le sommet de
439                     chemin.append(u) # départ + la distance entre u et le sommet actuel
440                     r = r - mat_adj[u][t] # on diminue la distance
441                     t = u # on passe par ce sommet et on recommence l'algorithme
442                 break
443     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 doit 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

```

```

485 # liste est la liste des droites paramétrée selon r et theta
486 # dico est un dictionnaire représentant les droites traitées et non traitées
487 # nb_lim est le nombre de pixel maximum de différence permettant la fusion de deux droites
488 # a_fact et b_fact sont les facteurs permettant de choisir les droites à prendre en compte entre elles, en paramétrage cartésien
489 def fusion_droites (liste: list, imageNB, dico: dict, a_fact: float, b_fact: float, nb_lim: int, precision_hough: int) -> list:
490     nouvelle_liste = []
491     modif = False
492     for i in range(len(liste)):
493         (r1,theta1) = liste[i]
494         for j in range(1, len(liste)):
495             (r2, theta2) = liste[j]
496             a1, b1 = droite(r1, theta1, precision_hough)
497             a2, b2 = droite(r2, theta2, precision_hough)
498             if r1 == r2 and theta1==theta2: # on ne se considère pas soit même
499                 pass
500             else :
501                 if (dico[(r1, theta1)]!=1) and (dico[(r2, theta2)]!=1): # on a déjà traité la droite
502                     min_a = min(abs(a1), abs(a2))
503                     moy_a = abs(a1 + a2)/2
504                     a_lim = a_fact + a_fact * min_a**2
505                     if (abs(a1-a2) < a_lim): # si les droites sont suffisamment proche selon les a
506                         b_lim = b_fact + moy_a**2 * b_fact # si les droites sont suffisamment proche selon des b
507                         if abs(b1 - b2) < b_lim :
508                             n1 = nombre_points_blanc((a1,b1), imageNB, 5)
509                             n2 = nombre_points_blanc((a2,b2), imageNB, 5)
510                             if (abs(n1-n2) < nb_lim): # on conserve la droite moyenne entre les deux
511                                 ntot = n1 + n2 # --
512                                 r_moy = (r1 + r2)/2 # --
513                                 theta_moy = (theta1 + theta2)/2 # --
514                                 nouvelle_liste.append((r_moy, theta_moy)) # --
515                             elif ( n1 > n2 ): # on conserve uniquement la droite 1
516                                 nouvelle_liste.append((r1, theta1)) # --
517                             else : # on conserve uniquement la droite 1
518                                 nouvelle_liste.append((r2, theta2)) # --
519                             modif = True
520                             dico[(r1, theta1)] = 1
521                             dico[(r2, theta2)] = 1
522             if dico[(r1,theta1)]!=1:
523                 nouvelle_liste.append((r1, theta1))
524     return(list(set(nouvelle_liste)), modif, dico)
525

```

```

526 # applique en boucle la fonction fusion_droites jusqu'à qu'il n'y ai plus de modifications
527 def fusion_droites_rec(liste: list, image: list , a_fact: int, b_fact: int, nb_lim: int, precision_hough: int) -> list:
528     dico = {} # Cas initial
529     for (r1,theta1) in liste:
530         dico[(r1,theta1)] = 0
531     nv_liste, booleen, dico = fusion_droites(liste, image, dico, a_fact, b_fact, nb_lim, precision_hough)
532
533     while booleen: # Le booleen représente si une modification a été faite ou non
534         dico = {} # si c'est le cas on recommence
535         for (r1,theta1) in nv_liste:
536             dico[(r1,theta1)] = 0
537         nv_liste, booleen, dico = fusion_droites(nv_liste, image, dico, a_fact, b_fact, nb_lim, precision_hough)
538     return(nv_liste)
539
540
541 # trie une liste selon la deuxieme coordonnée
542 def sort_list_theta(liste: list) -> list:
543     liste = [(x[1], x[0]) for x in liste]
544     liste.sort()
545     return([(x[1], x[0]) for x in liste])
546
547
548 # Permet de fusionner les intersections qui sont proches les une des autres, à partir de la matrice d'adjacence.
549 # distance_max est la distance à partir de laquelle on arrête de fusionner les droites.
550 def cluster(matrice_adj: list, distance_max: int) -> list:
551     for i in range(len(matrice_adj)): # on parcourt l'intégralité des sommets
552
553         for j in range(i+1, len(matrice_adj)):
554             point1 = matrice_adj[i][i]
555             point2 = matrice_adj[j][j]
556             x1, y1 = point1
557             x2, y2 = point2
558             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
559                 matrice_adj[i][i] = (int((x1+x2)/2), int((y1+y2)/2)) # la coordonnée du point est la
moyenne des deux
560                 matrice_adj[j][j] = (-1, -1)
561                 for k in range(len(matrice_adj)):
562                     if k!=i and k != j:
563                         if matrice_adj[j][k] != -1:
564                             if matrice_adj[i][k] != -1:
matrice_adj[i][k] = (matrice_adj[i][k] + matrice_adj[j][k])/2

```

```

565         matrice_adj[k][i] = (matrice_adj[k][i] + matrice_adj[k][j])/2
566     else :
567         matrice_adj[i][k] = matrice_adj[j][k]
568         matrice_adj[k][i] = matrice_adj[k][j]
569         matrice_adj[j][k] = -1
570         matrice_adj[k][j] = -1
571     return(cluster(matrice_adj, distance_max))
572 return(matrice_adj)
573
574
575 def matrice_recadre(points, mat):
576     belle_matrice = [[0 for i in range(len(points))] for i in range(len(points))]
577     for i in range(len(points)):
578         for j in range(len(points)):
579             belle_matrice[i][j] = mat[points[i]][points[j]]
580     return(belle_matrice)
581
582
583
584 def main(lien: str, precision_hough:int = 1):
585     print("Lecture de l'image.") # uniquement a partir de plan
586     image = read_img(lien)
587     print("Conversion en niveau de gris.")
588     image_grayscale = color_to_grayscale(image)
589     print("Conversion en noir et blanc.")
590     image_NB = grayscale_to_black_and_white(image_grayscale)
591     print("Calcul de l'espace de Hough.")
592     espace_h = hough_c(image_NB, precision_hough)
593     save_hough(espace_h)
594     print("Récupération des droites.")
595     liste_droites1 = peak_local_max(np.array(espace_h), min_distance=70, num_peaks=500, threshold_rel=0.23)
596     print(f"Nombre de droites : {len(liste_droites1)}.")
597     liste_droites1 = [list(x) for x in list(liste_droites1)]
598     save_with_droites(image, liste_droites1, precision_hough, "01. toutes les droites")
599     liste_droites2 = fusion_droites_rec(sort_list_theta(liste_droites1), image_NB, 0.5, 50, 10, precision_hough)
600     save_with_droites(image, liste_droites2, precision_hough, "02. premier filtrage des droites")
601     liste_droites3 = fusion_droites_rec(sort_list_theta(liste_droites2), image_NB, 0.3, 75, 10, precision_hough)
602     save_with_droites(image, liste_droites3, precision_hough, "03. deuxieme filtrage des droites")
603     print("Récupération de la listes des intervalles.")
604     liste_intervalles = intervalle(liste_droites3, image_NB, precision_hough)
605     save_liste_inter(liste_intervalles, image, "04. intervalles")

```



```

606 print("Raccorde les droites.")
607 liste_segments = raccordement(liste_intervalles, 28)
608 save_liste_inter(liste_segments, image, "05. raccordement (28)")
609 print("enlève les petits bouts de droite")
610 liste_segments = enlever_petits_intervalles_general(liste_segments)
611 save_liste_inter(liste_segments, image, "06. segments (28)")
612 liste_segments = flatten(liste_segments)
613 liste_intersections = intersection_general(liste_segments)
614 save_intersection(liste_intersections, image)
615 print("transformation en matrice d'intersections")
616 matrice_inter = matrice_intersections(liste_intersections, liste_segments)
617 matr_adj = mat_adjacence(matrice_inter, liste_intersections, image_NB, image)
618 print("clusterisation")
619 matrice_adj_cluster = cluster(matr_adj, 28)
620 save_cluster(matrice_adj_cluster, image)
621 save_graphe(matrice_adj_cluster, image)
622 #save_full_graphe(matrice_adj_cluster, image)
623
624
625 sommet_depart = 63
626 sommet_arrivee = 61
627 # d'autres valeurs intéressantes sont possibles : 61->73 ; 1->24
628
629
630 chemin_dijksra, nb_sommets_dijksra, duree_dijksra = plus_court_chemin_dijkstra(matr_adj, sommet_depart, sommet_arrivee)
631
632 start_a_star = time()
633 chemin_a_star, nb_sommets_a_star = a_etoile(matr_adj, heuristique, sommet_depart, sommet_arrivee)
634 duree_a_star = (time() - start_a_star)
635
636 print(f"chemin dijkra : {chemin_dijksra}, temps d'execution : {duree_dijksra}, sommets parcourus : {nb_sommets_dijksra}")
637 print(f"chemin A*: {chemin_a_star}, temps d'execution : {duree_a_star}, sommets parcourus : {nb_sommets_a_star}")
638 save_chemin(chemin_dijksra, matrice_adj_cluster, image, "dijkstra", "g")
639 save_chemin(chemin_a_star, matrice_adj_cluster, image, "a star", "k")
640
641 return(matrice_adj_cluster)
642
643
644
645
646 mat = main("https://server.ip/TIPE/paris2.png", 100)

```

Python Files\images.py

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 from math import pi, cos, sin
4 from PIL import Image
5
6
7 # cf main
8 def droite(r: float, theta: float, precision: int = 1) -> tuple:
9     theta = theta / precision
10    if theta == 0 or theta == 180:
11        return (0,r)
12    else :
13        theta_rad = (pi/180)*theta
14        a = -(cos(theta_rad)/sin(theta_rad))
15        b = r/sin(theta_rad)
16
17    return(a,b)
18
19
20 # enregistre l'image [image] avec l'ensemble des intersections détectées, présentes dans la liste [liste_intersections]
21 def save_intersection(liste_intersections: list, image: list) -> None:
22     fig, ax = plt.subplots()
23     fig.set_size_inches(18.5, 10.5)
24     plt.clf()
25     liste_pts = []
26     for i in range(len(liste_intersections)):
27         _,(x,y) = liste_intersections[i]
28         liste_pts.append((x,y))
29     for i in range(len(liste_pts)):
30         x,y = liste_pts[i]
31         plt.plot(x,y,marker="+",color="red")
32     plt.axis("off")
33     plt.imshow(image)
34     plt.savefig("07. intersections.png", dpi = 199)
35     plt.close()
36
37
```

```

38 # enregistre l'espace de Hough dans un fichier
39 def save_hough(espace_hough: list) -> None:
40     fig, ax = plt.subplots()
41     fig.set_size_inches(18.5, 10.5)
42     ax.axis('off') #enlève les axes
43     ax.imshow(espace_hough, cmap="gray")
44     plt.savefig("00. Hough.png", dpi=1600)
45     plt.close()
46
47
48 # permet d'obtenir les intersections, mais a partir de la matrice d'adjacence, qui a été modifié par la fonction cluster
49 def save_cluster(matrice_adj: list, image: list) -> None:
50     fig, ax = plt.subplots()
51     fig.set_size_inches(18.5, 10.5)
52     for i in range(len(matrice_adj)):
53         x, y = matrice_adj[i][i]
54         plt.plot(x,y,marker="+",color="red")
55     ax.axis('off') #enlève les axes
56     ax.imshow(image, extent=[0,len(image[0])-1, len(image)-1, 0]) # permet d'avoir l'image dans le bon sens et à la bonne taille
57     plt.savefig("08. cluster.png")
58     plt.close()
59
60
61 # Enregistre un chemin renvoyé par un algorithme
62 # de plus court chemin tel que Dijkstra ou A*
63 # chemin_liste est une liste de points qui constituent le chemin
64 def save_chemin(chemin_liste: list, mat_adj: list, image: list, texte: str, color: str) -> None:
65     fig, ax = plt.subplots()
66     fig.set_size_inches(18.5, 10.5)
67     c = -1
68     for k in range(len(mat_adj)): # affichage des points
69         i, j = mat_adj[k][k]
70         c +=1
71         plt.plot(i, j, 'bo')
72         if k in chemin_liste :
73             plt.text(i+5, j-15, f"n°{c}", c="red", fontsize=16) # affichage des numero de sommet si il appartient au sommet
74     liste_sommet = []
75     for i in range(len(mat_adj)):
76         liste_sommet.append(mat_adj[i][i])
77     for i in range(len(chemin_liste)-1):
78         x1,y1 = liste_sommet[chemin_liste[i]]

```

```

79     x2,y2 = liste_sommet[chemin_liste[i+1]]
80     plt.plot([x1,x2], [y1,y2], color)
81     plt.imshow(image)
82     ax.imshow(image, cmap = "gray", extent=[0,len(image[0])-1, len(image)-1, 0])
83     ax.axis('off') #enlève les axes
84     plt.savefig(f"10. chemin - {texte}.png")
85     plt.close()
86
87
88 # enregistre les images avec les differents segments, mis dans la liste [liste_intervalles]
89 def save_liste_inter(liste_intervalles: list, image: list, nom: str) -> None:
90     fig, ax = plt.subplots()
91     fig.set_size_inches(18.5, 10.5)
92     for i in liste_intervalles :
93         if i != [] :
94             for k in i:
95                 x_values = [k[0][0], k[1][0]]
96                 y_values = [k[0][1], k[1][1]]
97                 plt.plot(x_values, y_values, linestyle="--")
98     plt.axis('off')
99     plt.imshow(image)
100    plt.savefig(f"{nom}.png", dpi = 199)
101    plt.close()
102
103
104 # enregistre l'image de la ville avec les intersections nommées
105 # add_str est une chaine de caractere ajoutée au nom du fichier
106 def save_graphe(mat_adj: list, img: list, add_str: str = "") -> None:
107     fig, ax = plt.subplots()
108     fig.set_size_inches(18.5, 10.5)
109     c = -1
110     for k in range(len(mat_adj)): # affichage des points
111         i, j = mat_adj[k][k]
112         c +=1
113         plt.plot(i, j, 'bo')
114         plt.text(i+5, j-15, f"n°{c}", c="red", fontsize=16)
115     plt.axis('off')
116     plt.imshow(img)
117     plt.savefig(f"09. graphe{add_str}.png", dpi = 199)
118
119

```

```

120 def save_full_graphe(mat_adj: list, img: list, add_str: str = "") -> None:
121     fig, ax = plt.subplots()
122     fig.set_size_inches(18.5, 10.5)
123     c = -1
124     for k in range(len(mat_adj)): # affichage des points
125         i, j = mat_adj[k][k]
126         c +=1
127         plt.plot(i, j, 'bo')
128         plt.text(i+5, j-15, f"n°{c}", c="red", fontsize=16)
129
130     for k in range(len(mat_adj)): # affichage des points
131         for l in range(k+1, len(mat_adj)):
132             if mat_adj[k][l] != -1 and mat_adj[k][k] != (-1, -1) and mat_adj[l][l] != (-1, -1):
133                 x1, y1 = mat_adj[k][k]
134                 x2, y2 = mat_adj[l][l]
135                 plt.plot([x1,x2], [y1,y2])
136
137     plt.axis('off')
138     plt.imshow(img)
139     plt.savefig(f"09. graphe{add_str}.png", dpi = 199)
140
141
142 # affiche l'image d'origine avec les droites
143 def save_with_droites(image: list, liste_r_theta: list, precision_hough: int = 1, name: str = "show_with_droite"):
144     fig, ax = plt.subplots()
145     fig.set_size_inches(18.5, 10.5)
146     x = np.array(range(len(image[0])))
147
148     for i in range(len(liste_r_theta)): #plot chaque droite
149         a, b = droite(liste_r_theta[i][0], liste_r_theta[i][1], precision_hough)
150         y = a*x+b
151         ax.axis('off') #enlève les axes
152         ax.plot(x, y, '--', linewidth=1, color="red")
153
154     img = ax.imshow(image, cmap='gray', extent=[0, len(image[0])-1, len(image)-1, 0])
155     plt.savefig(name + ".png", dpi=199)
156     plt.close()
157
158
159 def save_custom_graphe(mat_adj: list, img: list, pts: list, add_str: str = "") -> None:
160     fig, ax = plt.subplots()

```

```
161 fig.set_size_inches(18.5, 10.5)
162 c = 0
163
164 for k in pts: # affichage des points
165     for l in pts:
166         if mat_adj[k][l] != -1 and mat_adj[k][k] != (-1, -1) and mat_adj[l][l] != (-1, -1) :
167             x1, y1 = mat_adj[k][k]
168             x2, y2 = mat_adj[l][l]
169             plt.plot([x1,x2], [y1,y2], "b-")
170
171
172 for k in pts: # affichage des points
173     i, j = mat_adj[k][k]
174     c +=1
175     plt.plot(i, j, 'ro')
176     #plt.text(i+5, j-15, f"n°{c}", c="red", fontsize=16)
177
178
179
180 plt.axis('off')
181 plt.imshow(img)
182 plt.savefig(f"11. custom graphe{add_str}.png", dpi = 199)
183
184
185
```

Python Files\kmoy.py

```
1 from PIL import Image, ImageOps
2 import numpy as np
3 import random as rd
4 import math
5
6 # Lecture de l'image et changement du contraste
7 image = Image.open("./images/2.jpg",mode = 'r')
8 image = ImageOps.autocontrast(image, cutoff=10)
9
10 im = np.asarray(image)
11
12 nb_lignes, nb_colones, nb_subpixel = np.shape(im) # nb_subpixel est le nombre d'éléments d'un points : (R, G, B) = 3
13
14 classe = [[-1 for i in range(nb_colones)] for j in range(nb_lignes)] # Tableau des classes de chaque éléments
15 nb_classe = 8
16
17 # initialisation centres
18 centres = []
19
20 for i in range(nb_classe) : # On prend des pixels aléatoires dans l'image, et on vérifie qu'ils sont tous différents
21     ir = rd.randint(0,nb_lignes-1)
22     jr = rd.randint(0,nb_colones-1)
23     while [im[ir,jr,s] for s in range(nb_subpixel)] in centres :
24         ir = rd.randint(0,nb_lignes-1)
25         jr = rd.randint(0,nb_colones-1)
26     centres.append([im[ir,jr,s] for s in range(nb_subpixel)])
27
28 def distance(e1: list, e2: list) : # distance entre deux couleurs
29
30     d = len(e1)
31     res = 0
32     for i in range(d) :
33         res = max(res,(e1[i]-e2[i])**2)
34     return(res)
35
```

```

36 # ranger au plus proche des centres
37 def classer() :                               # permet de classer les elements dans chaque classe
38     global classe, im, nb_lignes, nb_colones, centres
39     b = False
40     for i in range(nb_lignes) :                # On parcours toute l'image
41         for j in range(nb_colones) :          # --
42             dmin = 0
43             if classe[i][j] == -1 :           # si il n'y a pas encore de classe
44                 dmin = math.inf              # la distance est la minimum
45             else :
46                 dmin = distance(im[i,j], centres[classe[i][j]])
47             for m in range(nb_classe) :
48                 dist = distance(list(im[i,j]), centres[m])
49                 if dist < dmin :
50                     dmin = dist
51                     classe[i][j] = m
52                     b = True
53     return(b)
54
55
56 # les centres bougent
57 def barycentre() :
58     global classe, im, nb_lignes, nb_colones, centres
59     dico = dict()
60     for i in range(nb_classe) :
61         dico[i] = []
62     for i in range(nb_lignes) :
63         for j in range(nb_colones) :
64             dico[classe[i][j]].append(im[i,j])
65     for i in range(nb_classe) :
66         r, g, b = 0, 0, 0
67         for e in dico[i] :
68             r += e[0]
69             g += e[1]
70             b += e[2]
71         m = len(dico[i])
72         centres[i] = [r//m, g//m, b//m]
73 # kmoyennes
74 i=0
75 while classer() :
76     i+=1

```



```

77     print("\nstep"+str(i))
78     print(centres)
79     barycentre()
80     im_classes = np.array([[centres[classe[i]][j]] for j in range(nb_colones)] for i in range(nb_lignes)])
81     res = Image.fromarray(im_classes, mode='RGB')
82
83
84 # affichage,sauvegarde
85
86 colors = [[0,0,0],[150,0,0],[0,150,0],[0,0,150],[128,128,0],[0,160,160],[192,0,192],[230,230,230]] # liste des couleurs pour bien
différencier les classes
87
88 # Enregistrer toutes les images
89 for lettre in range(nb_classe):
90     im_classes = np.array([[im[i][j] if classe[i][j] == lettre else (0, 0, 0) for j in range(nb_colones)] for i in
range(nb_lignes)],dtype=np.uint8)
91     print(im_classes)
92     res = Image.fromarray(im_classes, mode="RGB")
93     res.show()
94     res.save(f"./out/2_{lettre}.jpg")
95
96
97 b = np.array([[colors[classe[i]][j]] for j in range(nb_colones)] for i in range(nb_lignes)],dtype=np.uint8)
98 res = Image.fromarray( b,mode="RGB")
99 res.show()
100 res.save("./out/out.jpg")

```

Python Files\modules.py

```
1 class FP:
2     def __init__(self):
3         self.tab_elem = []    # elements
4         self.tab_prio = []    # priorité
5
6     def push(self, sommet, prio): #ajoute l'élément sommet a la file de priorité
7         self.tab_elem.append(sommet)
8         self.tab_prio.append(prio)
9
10    def pop(self): # sort l'élément de priorité minimale
11        indice = self.tab_prio.index(min(self.tab_prio))
12        elem = self.tab_elem[indice]
13        self.tab_prio.pop(indice)
14        self.tab_elem.pop(indice)
15        return(elem)
16
17    def is_empty(self):
18        return(len(self.tab_prio) == 0)
19
20
```

c\hough.c

```
1 // Objectif : utiliser du multithreading pour améliorer la rapidité
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <math.h>
6
7 int COLONNES;
8 int LIGNES;
9 int DISTANCE;
10
11 // taille du fichier. update global variables
12 int* taille(char* filepath){
13     FILE* ptr;
14     char ch;
15     ptr = fopen(filepath, "r");
16     if (NULL == ptr) {
17         printf("file can't be opened \n");
18     }
19     int lignes = 0;
20     int colonnes = 1;
21     int end = 1;
22     while (!feof(ptr)) {
23         ch = fgetc(ptr);
24         if (ch == '\n'){
25             lignes ++;
26             end = 0;
27         }
28         else if(ch == ',' && end) {
29             colonnes ++;
30         }
31     }
32
33     LIGNES = lignes;
34     COLONNES = colonnes;
35     DISTANCE = (int)round(sqrt(COLONNES *COLONNES + LIGNES *LIGNES));
36 }
37
```

```

38 //retourne la matrice
39 int* read_img(char* filepath){
40     int* matrice = malloc((COLONNES * LIGNES) * sizeof(int));
41
42     FILE* ptr;
43     char ch;
44     ptr = fopen(filepath, "r");
45
46     if (NULL == ptr) {
47         printf("file can't be opened \n");
48     }
49     int c = 0;
50     while (!feof(ptr)) {
51         ch = fgetc(ptr);
52         if (feof(ptr)){
53             fclose(ptr);
54             return matrice;
55         }
56         if (ch == '\n'){
57             c++;
58         }
59         else if(ch == ',') {
60             c++;
61         }
62         else if (ch == ' '){}
63         else {
64             //printf("%c\n", ch);
65             matrice[c] *= 10;
66
67             matrice[c] += (int)ch - 48;
68         }
69     }
70 }
71
72 void print_matrice(int* mat){
73     for (int i = 0; i<LIGNES; i++){
74         for (int j = 0; j < COLONNES; j++){
75             printf("%d | ", mat[i*COLONNES + j]);
76         }
77         printf("\n");
78     }

```

```

79 }
80
81 double rho(int x, int y, double theta){
82     return(x * cos(theta) + y * sin(theta));
83 }
84
85 int* espace_hough(int* image, int precision){
86     int* mat_hough = malloc(360 * precision * DISTANCE * sizeof(int));
87     int v_rho;
88     double theta_rad;
89     for (int i = 0; i < 360 * DISTANCE * precision; i++){
90         mat_hough[i] = 0;
91     }
92     for (int y = 0; y < LIGNES; y++){
93         for (int x = 0; x < COLONNES; x++){
94             if (image[y*COLONNES + x] > 250){
95                 for (int theta_deg = 0; theta_deg < 360 * precision ;theta_deg ++){
96                     theta_rad = theta_deg /(180.0 * precision) * M_PI;
97                     v_rho = rho(x, y, theta_rad);
98                     if (v_rho > 0){
99                         mat_hough[v_rho * 360 * precision + theta_deg] += 1;
100                 }
101             }
102         }
103     }
104 }
105 return(mat_hough);
106 }
107
108
109 void print_hough(int* mat){
110     for (int i = 0; i<360; i++){
111         for (int j = 0; j < DISTANCE; j++){
112             printf("%d | ", mat[j*360 + i]);
113         }
114         printf("\n");
115     }
116 }
117
118 void save_file(int* mat, int precision){
119     FILE *ptr;

```

```
120 ptr = fopen("./out.txt","w");
121
122 for (int j = 0; j < DISTANCE; j++){
123     for (int i = 0; i<360 * precision; i++){
124         fprintf(ptr, "%d", mat[j*360 * precision + i]);
125         if (i != 360 * precision - 1){
126             fprintf(ptr, "%s", ",");
127         }
128     }
129     fprintf(ptr, "%s", "\n");
130 }
131
132 fclose(ptr);
133
134 }
135
136
137 int main()
138 {
139     int precision = 100;
140     printf("precision : %d\n", precision);
141     char* filepath = "./in.txt";
142     taille(filepath);
143     printf("Lignes : %d\nColonnes : %d\nnombre de pixels : %d\nDistance : %d\n", LIGNES, COLONNES, (LIGNES)*(COLONNES), DISTANCE);
144     int* image = read_img(filepath);
145     //print_matrice(image);
146     int* espace = espace_hough(image, precision);
147     //print_hough(espace);
148
149     save_file(espace, precision);
150     return 0;
151 }
152
153
```

c\hough_multi.c

```
1 // Objectif : utiliser du multithreading pour améliorer la rapidité
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <math.h>
6
7 #include <pthread.h>
8
9 int PRECISION = 10;
10 char* filepath = "./in.txt";
11
12 int COLONNES;
13 int LIGNES;
14 int DISTANCE;
15 int* IMAGE;
16 int* ESPACE;
17
18 // taille du fichier. update global variables
19 int* taille(char* filepath){
20     FILE* ptr;
21     char ch;
22     ptr = fopen(filepath, "r");
23     if (NULL == ptr) {
24         printf("file can't be opened \n");
25     }
26     int lignes = 0;
27     int colonnes = 1;
28     int end = 1;
29     while (!feof(ptr)) {
30         ch = fgetc(ptr);
31         if (ch == '\n'){
32             lignes ++;
33             end = 0;
34         }
35         else if(ch == ',' && end) {
36             colonnes ++;
37         }
```

```

38     }
39
40     LIGNES = lignes;
41     COLONNES = colonnes;
42     DISTANCE = (int)round(sqrt(COLONNES *COLONNES + LIGNES *LIGNES));
43 }
44 //retourne la matrice
45 int* read_img(char* filepath){
46     int* matrice = malloc((COLONNES * LIGNES) * sizeof(int));
47
48     FILE* ptr;
49     char ch;
50     ptr = fopen(filepath, "r");
51
52     if (NULL == ptr) {
53         printf("file can't be opened \n");
54     }
55     int c = 0;
56     while (!feof(ptr)) {
57         ch = fgetc(ptr);
58         if (feof(ptr)){
59             fclose(ptr);
60             IMAGE = matrice;
61             return NULL;
62         }
63         if (ch == '\n'){
64             c++;
65         }
66         else if(ch == ',') {
67             c++;
68         }
69         else if (ch == ' '){}
70         else {
71             //printf("%c\n", ch);
72             matrice[c] *= 10;
73
74             matrice[c] += (int)ch - 48;
75         }
76     }
77
78 }

```



```

79
80
81 double rho(int x, int y, double theta){
82     return(x * cos(theta) + y * sin(theta));
83 }
84
85
86 void* espace_hough(void* arg){
87     int thread = *(int*)arg; // numero du thread equivalent a la zone de degré qui va être parcourue
88     int v_rho;
89     double theta_rad;
90     for (int y = 0; y < LIGNES; y++){
91         for (int x = 0; x < COLONNES; x++){
92             if (IMAGE[y*COLONNES + x] > 250){
93                 for (int theta_deg = 360 * thread ; theta_deg < 360 * (thread+1) ;theta_deg++){
94                     theta_rad = theta_deg / (180.0 * PRECISION) * M_PI;
95                     v_rho = rho(x, y, theta_rad); // determine R
96                     if (v_rho > 0){
97                         ESPACE[v_rho * 360 * PRECISION + theta_deg] += 1;
98                     }
99                 }
100             }
101         }
102     }
103     return(NULL);
104 }
105
106
107
108 void save_file(int* mat, int PRECISION){
109     FILE *ptr;
110     ptr = fopen("./out.txt","w");
111
112     for (int j = 0; j < DISTANCE; j++){
113         for (int i = 0; i<360 * PRECISION; i++){
114             fprintf(ptr, "%d", mat[j*360 * PRECISION + i]);
115             if (i != 360 * PRECISION - 1){
116                 fprintf(ptr, "%s", ",");
117             }
118         }
119         fprintf(ptr, "%s", "\n");

```

```

120     }
121
122     fclose(ptr);
123
124 }
125
126
127 int main(int argc, char *argv[])
128 {
129     if (argc == 2){
130         PRECISION = strtol(argv[1], NULL, 10);
131     }
132     taille(filepath);
133
134     printf("Precision : %d\nLignes : %d\nColonnes : %d\nnombre de pixels : %d\nDistance : %d\n", PRECISION, LIGNES, COLONNES,(LIGNES)*
(COLONNES), DISTANCE);
135     read_img(filepath);
136
137     int* mat_hough = malloc(360 * PRECISION * DISTANCE * sizeof(int));
138     for (int i = 0; i<360 * DISTANCE * PRECISION; i++){
139         mat_hough[i] = 0;
140     }
141     ESPACE = mat_hough;
142
143     pthread_t* t = (pthread_t*)malloc(sizeof(pthread_t)*(PRECISION));
144     int* tab_int = malloc(PRECISION * sizeof(int));
145     for(int i = 0; i < PRECISION; i++){
146         tab_int[i] = i;
147     }
148
149     for (int i=0; i < PRECISION; i++) {
150         pthread_create(&t[i], NULL, espace_hough, (void*)&tab_int[i]);
151     }
152
153     for (int i=0; i < PRECISION; i++) {
154         pthread_join(t[i], NULL);
155     }
156
157     save_file(ESPACE, PRECISION);
158
159     return 0;}

```