Transformée de Hough, compte rendu

La transformée de Hough est une technique utilisée en traitement d'images pour détecter des formes géométriques, comme des lignes ou des cercles, dans une image. Elle représente les formes dans un espace paramétrique et détecte les paramètres correspondants aux formes recherchées. C'est une méthode puissante pour la détection de formes dans les images.

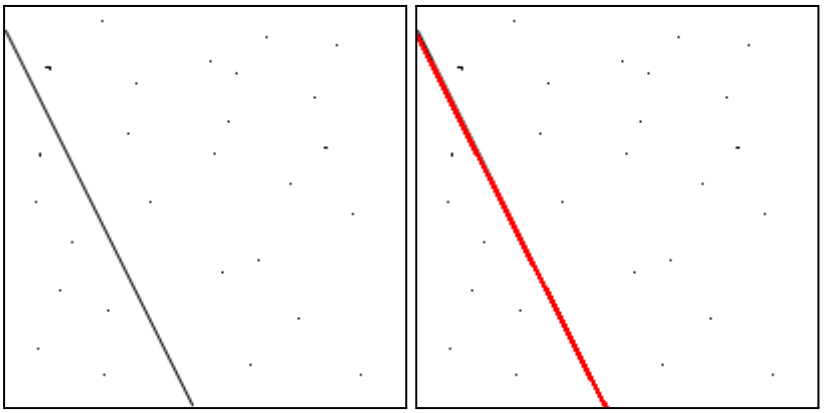


Image d'origine

Après détections
(Droites en rouge)

Sommaire

Méthode naïve

A/ Mathématiques

L'idée de la transformation de Hough est donc de représenter des formes dans un espace paramétrique qu'on appellera logiquement l'espace de Hough. Pour ce faire nous aurons dans un premier temps besoin de l'équation de la forme à représenter, nous commencerons avec l'objectif de détecter des droites dans une image et dans cette méthode naïve nous aurons donc besoin de l'équation cartésienne simplifiée d'une droite.

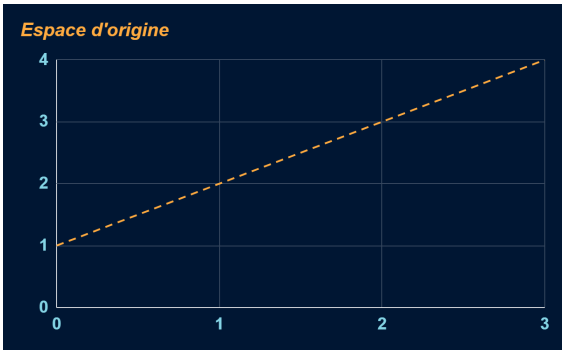
$$y = xm + b$$

Ici m correspond au coefficient directeur de la droite et b à l'intersection entre la droite et l'axe des ordonnées.

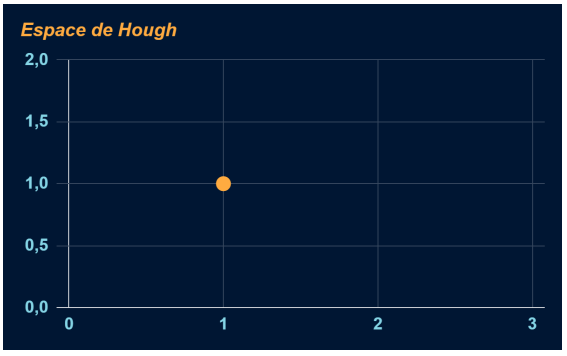
Le principe va donc être de prendre ces deux paramètres m et b et d'en faire un nouvel espace qui aura m en abscisse et b en ordonnée c'est l'espace de Hough. Dans cet espace chaque point aura donc un couple (m, b) associé et donc une droite associée, une droite dans l'espace de paramètre (x, y) habituel que l'on appellera l'espace d'origine sera donc représenté par un point dans l'espace de Hough et un point dans cet espace d'origine sera donc représenté par une droite dans l'espace de Hough en suivant la formule ci-dessous.

$$y = xm + b$$

$$\Longleftrightarrow b = y - mx$$



Cette droite à clairement un m =1 et b =1



Son point dans l'espace de Hough est donc en (1, 1)

B/ Présentation de OpenCV

OpenCv est le module que j'ai utilisé tout au long de mon projet, elle permet de transformer des images en matrices et donc permet de modifier des pixels, ainsi que de tracer des droites et autres formes. Il contient également plusieurs fonctions utiles sur des images comme par exemple la détection de contour canny, la modification des couleurs d'une image ou encore simplement la transformée de Hough, cette fonction m'a permis de comparer mes résultats à ceux de openCV. Ce module permet enfin la création d'images vides en utilisant la fonction cv2.zeros ou cv2.ones par exemple.

C/ Algorithme

Le principe de l'algorithme utilisé est de récupérer toutes les coordonnées (x, y) composant le contour de notre image d'origine et pour chaque coordonnée, tracer la droite correspondante dans l'espace de Hough.

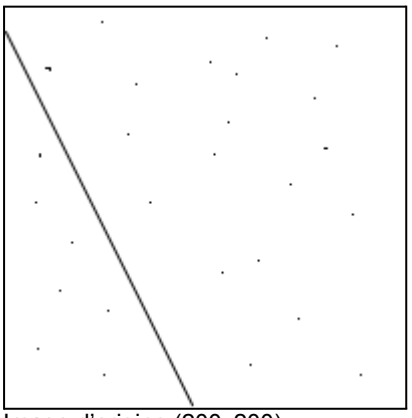
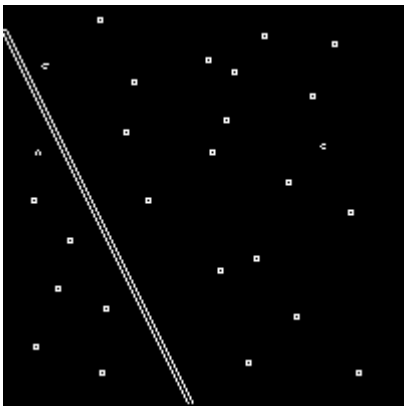
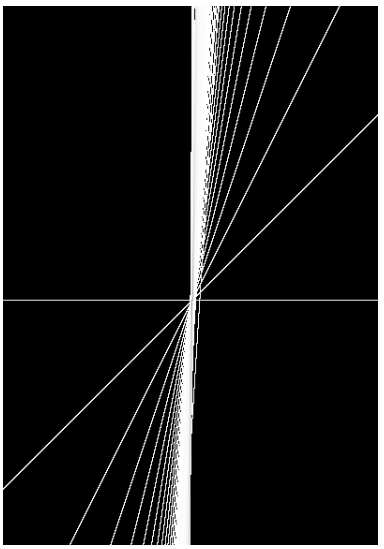
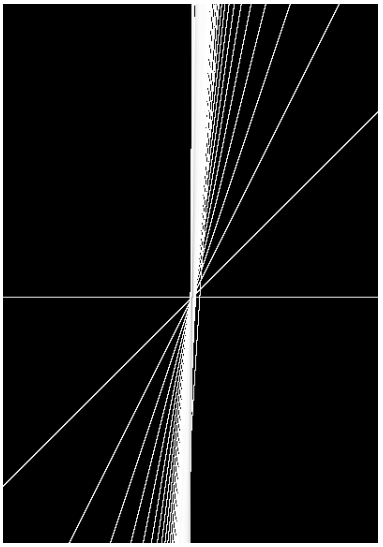


Image d'origine (200x200)

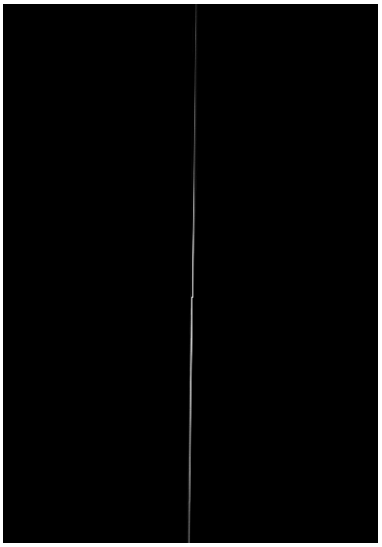

Détection de contour canny


Espace de Hough

C'est ce que nous avons faits ici En blanc sur un fond noir et nous voyons que toutes les droites convergent en un seul point, ce point va avoir pour coordonnées (m, b) les coordonnées de la droite venant de notre image d'origine. L'objectif va donc être de récupérer les coordonnées de ce pixel sur lequel se croisent toutes les droites. Pour cela nous allons créer ce que l'on appelle un accumulateur et plutôt que de simplement tracer les droites en blanc on va éclairer les pixels pour chaque droite passant sur eux, ce qui veux dire que plus un pixel est clair, plus un grand nombre de droite passe par celui-ci.



Espace de Hough



Accumulateur

On remarque que dans l'accumulateur les pixels centraux sont plus clairs, on récupère donc les coordonnées de ce pixels et on trace la droite correspondante sur notre image d'origine.

D/ Problème

Nous allons rapidement rencontrer un problème en utilisant cette méthode… Effectivement pour des droites verticales nous aurons des coefficients directeurs très grands. Cela va poser problème car pour chercher le point le plus clair de notre accumulateur nous devrons alors parcourir énormément de pixels et cela nous prendrait trop de temps de calcul.

Coordonnées polaires

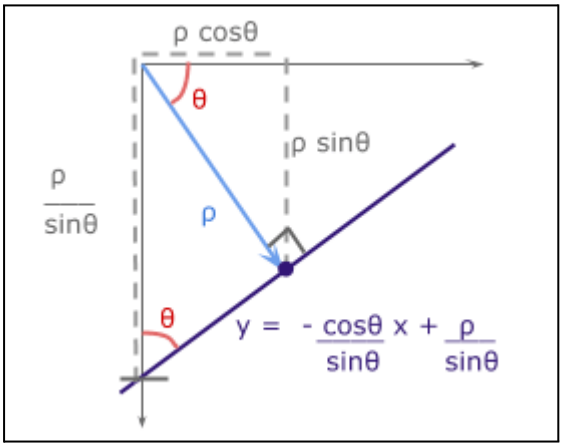
Une solution au problème énoncé ci-dessus est d'utiliser les coordonnées polaires d'une droite.

A/ Mathématiques

Voici donc l'équation polaire d'une droite. Cette fois ci nos deux paramètres ne seront plus m et b mais θ et ρ.

$$y = \frac{-cos(\theta)}{sin(\theta)}x + \frac{\rho}{sin(\theta)}$$

θ correspond à l'angle entre la droite et l'axe des abscisse et ρ à la distance entre la droite et l'origine.



Comme précédemment en utilisant la formule ci-dessous on peut prendre un point dans notre espace d'origine et tracer dans notre nouvel espace de Hough une sinusoïdale cette fois-ci.

$$\rho = xcos(\theta) + ysin(\theta)$$

L'avantage principal de cette technique est que l'on peut connaître la taille de notre accumulateur. effectivement θ va de 90 à -90 et ρ va de pmax à -pmax, pmax étant la diagonale de notre image, il suffit donc d'utiliser le théorème de pythagore.

B/ Algorithme

Mon premier algorithme est composé de trois fonctions principales appelées houg_line, accumulator et co_max

I. houg_line

```
def houg_line(img, nb):

  # On effectue canny sur notre image
  canny = cv.Canny(img, 100, 200)

  # On récupère l'accumulateur
  acc = accumulator(canny, 1)

  # On récupère la taille de l'image
  haut = len(img)
  larg = len(img[1])

  # On récupère les coordonnées des nb pixels les plus clair
de l'accumulateur
  c_max = coordonneesMax(acc, nb)

  # On en sort les thetas en repassant les degrés en radians
et les rayons
  co_pol = []
  for co in c_max:
    theta = np.deg2rad(co[0] - 90)
    p = co[1] - int(m.sqrt(haut ** 2 + larg ** 2))
    co_pol.append((theta, p))

  # On calcule 2 y par coordonnées en utilisant la formule y
= (-cos(O)/sin(O))*x+r/sin(O)
  tab_y = []
  for co in co_pol:
    y = int((( - m.cos(co[0])) / m.sin(co[0])) + co[1] /
m.sin(co[0]))
```

```
    y_max = int((( - m.cos(co[0])) / m.sin(co[0])) * larg +
co[1] / m.sin(co[0]))
    tab_y.append((y, y_max))

  # On trace des lignes sur les coordonnées trouvées
  for line in tab_y:
    cv.line(img, (1, line[0]), (larg, line[1]), (0, 0, 255),
3)

  return img
```

II. accumulator

```
def accumulator(canny:list, incr:int)->list:
  # On récupère la taille de l'image
  haut = len(image)
  larg = len(image[0])
  # On calcule pmax en utilisant le théorème de pythagore
  pMax = int(m.sqrt(haut ** 2 + larg ** 2))
  # On fait un tableau de 180 radians (entre pi/2 et -pi/2)
  thetas = np.deg2rad(np.arange(-90, 90))
  # On crée notre accumulateur
  accumulator = np.zeros((dMax * 2, len(thetas)),
dtype=np.uint8)

  # On boucle ensuite dans tous les pixels de canny
  for y in range(haut):
      for x in range(larg):
        # Si le pixel est clair on continue
        if img[y][x][0] < 100:
          # On boucle dans les thêta de l'accumulateur
          for o in range(len(thetas)):
            # On calcule p en fonction de thêta, x et y
            r = x * m.cos(thetas[o]) + y * m.sin(thetas[o])
            # Ensuite nous incrémentons l'accumulateur de
incr
            accumulator[int(r) + dMax, o] += incr
  return accumulator
```

III. co_max

```
def co_max(acc:list, n:int)->list:

  # On définit comme mémoire d'origine un tableau de n (0,0).
  res=[(0, 0)] * n

  # Puis nous allons boucler sur tous les pixels de acc.
  for y in range(len(acc)):
    for x in range(len(acc[0])):
      # on définit le minimum à l'indice 0
      min = 0
      # On boucle sur la taille longueur du tableau -1
      for i in range(len(res) - 1):
          # On compare le point stocker dans res à l'indice
minimum avec le point dans res à l'indice i + 1
          if acc[res[min][1]][res[min][0]] > acc[res[i +
1][1]][res[i + 1][0]]:
            # On met à jour l'indice min
            min = i + 1
      # Si le point aux indices est plus clair que celui
qu'on avait dans res à l'indice min, on le remplace.
      if acc[y][x] > acc[res[min][1]][res[min][0]]:
        res[min] = (x, y)

  return res
```

C/ Test

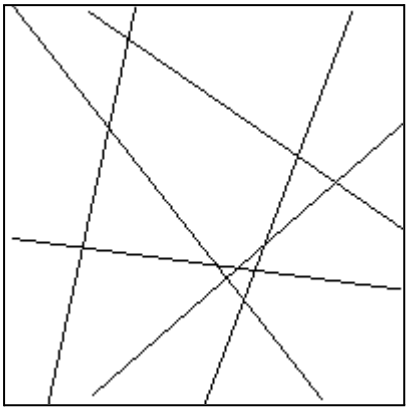
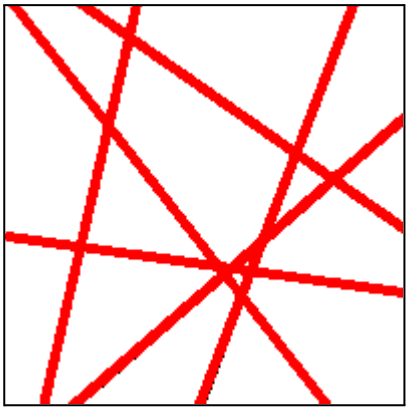
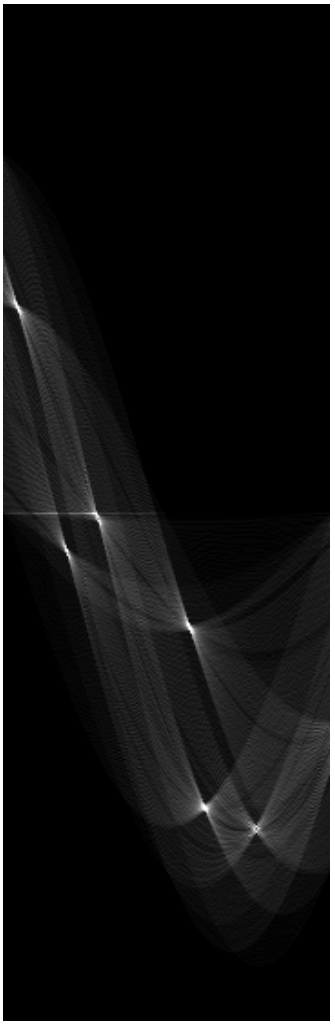


image d'origine (200x200)



Lignes en rouges (3s)



Accumulateur

Nous pouvons voir qu'en 3 secondes nous avons un résultat satisfaisant cependant ce n'est pas encore suffisant car nous voulons si possible faire du temps réel.

Méthode “probabilistic”

A/ Algorithme

Notre objectif va donc être d’améliorer l’algorithme précédent pour avoir un résultat plus rapide. Pour cela nous allons mettre en place une boucle qui va tourner sur 3 étapes.

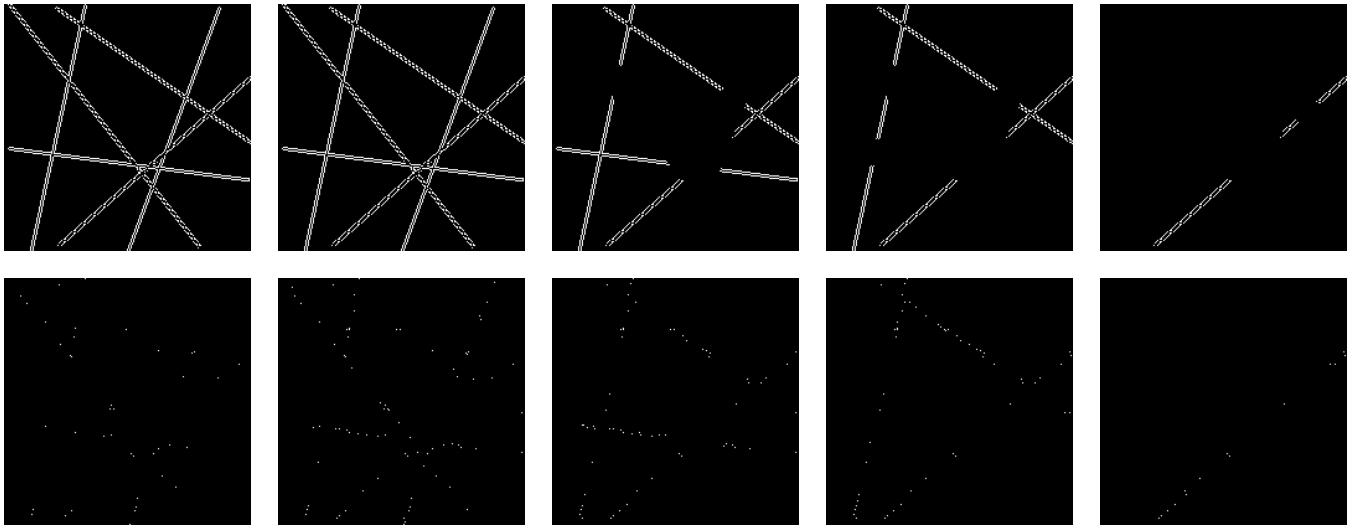
La première étape va donc consister à prendre un certain pourcentage (assez bas) des points blancs de notre détection de contour et de les tracer dans notre accumulateur.

Ensuite nous allons mettre en place un seuil que nous allons passer dans notre accumulateur et tous les points plus clair que ce seuil seront donc récupérés comme faisant partis des résultats

La dernière étape est de tracer les droites correspondant aux résultats (s’il y en a) sur l’image d’origine mais également en noir su le canny pour effacer les points de contours qui sont sur des droites déjà trouvées.

Nous reprenons donc un pourcentage de points que nous ajoutons à ceux d’avant et continuons la boucle jusqu’à avoir autant de résultats que voulus.

B/ Test



Etape 1 Etape 2 Etape 3 Etape 4 Etape 5

En haut c'est l'image canny depuis laquelle on prend les points et en bas ce sont les points réellement tracés dans l'accumulateur. Nous observons bien que les lignes disparaissent petit à petit au fur et à mesure que des droites sont détectées.

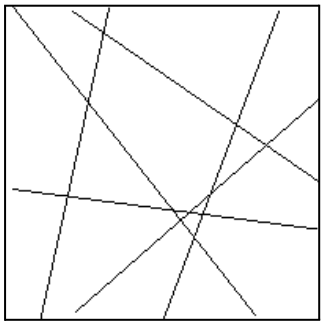
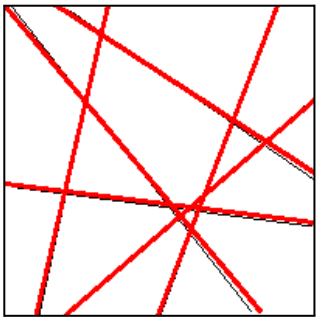


Image d'origine (200x200)



Détection de lignes en rouge (<1s)

Nous avons ici un résultat très satisfaisant avec cette fois une attente de moins d'une seconde, preuve que notre algorithme fonctionne mieux.

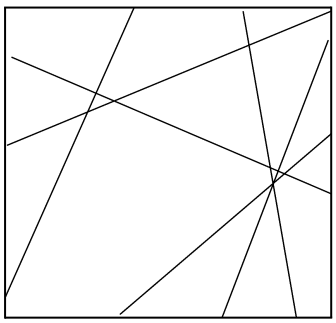
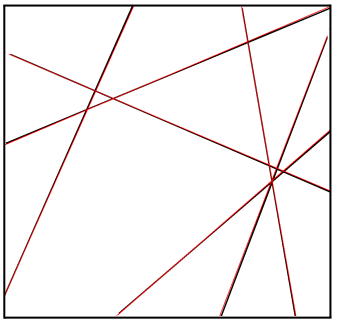


Image d'origine (1000x1000)



Détection de ligne en rouge (10s)

On se rend compte que sur de plus grandes images comme celle-ci la détection est bien plus longue, cela est dû aux nombreuses boucles effectuées sur l'image directement. Nous avons donc un problème.

Méthode Aléatoire

A/ Algorithme

L'objectif ici va donc être de créer un nouvel algorithme qui tourne moins directement dans l'image pour gagner du temps sur les images contenant beaucoup de pixels. Ce nouvel algorithme dit Aléatoire va se dérouler encore une fois dans une boucle principale, en 3 grandes étapes. Tout d'abord nous allons récupérer la détection de contour de notre image et prendre 2 points aléatoirement sur ces contours. Ensuite nous allons définir pour ces deux points la droite correspondante en calculant m (coefficient directeur) et b (intersection entre la droite et l'axe de ordonnées). Ensuite s'ils n'y sont pas déjà on ajoute ce couple (m, b) comme clef d'un dictionnaire avec la valeur 1 s'ils n'y sont pas déjà et sinon on incrémente simplement leurs valeurs. Enfin on regarde si une valeur du dictionnaire dépasse un seuil donné en entrée de la fonction et si c'est le cas on récupère le couple (m, b) en question pour tracer la droite correspondante. On fait tourner cette boucle jusqu'à avoir le bon nombre de droites.

B/ Test

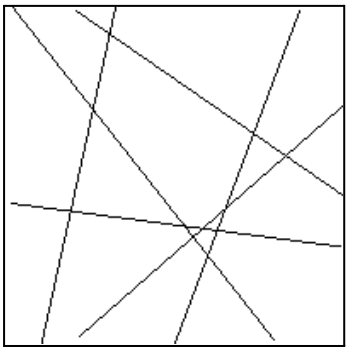
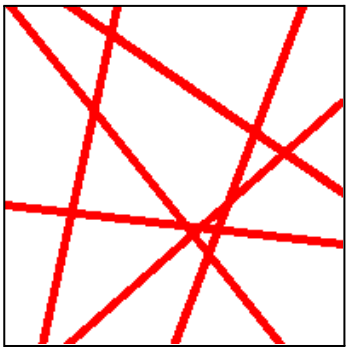


Image d'entrée (200x200)



Détection en rouge (1s)

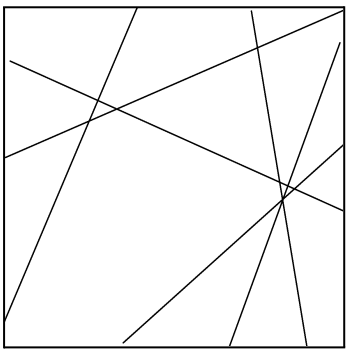
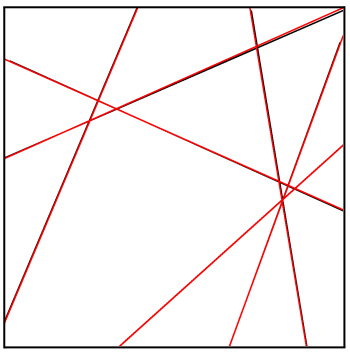


Image d'entrée (1000x1000)



Détection en rouge (4s)

Sur la plus petite image, on est légèrement moins efficace qu'avant mais le résultat reste très satisfaisant. Nous avons clairement une grande amélioration sur l'image la plus grande, l'algorithme est bien plus rapide, il est certainement possible d'optimiser l'algorithme pour qu'il fonctionne parfaitement en temps réel avec un peu plus de travail sur le code.

Autres formes (Exemple du cercle)

A/ Mathématiques

Pour détecter une autre forme qu'une droite il suffit simplement d'avoir l'équation de cette autre forme, ici nous prendrons l'exemple du cercle, voici donc la formule d'un cercle.

$$r^2 = (x - x_0)^2 + (y - y_0)^2$$

Cette fois-ci nous avons donc 3 paramètres qui sont r, x0 et y0. r correspond au rayon du cercle, x0 au x de son centre et y0 au y de son centre. Notre espace de Hough sera donc à 3 dimensions mais fonctionnera de la même manière que celui d'une droite, on va incrémenter les valeurs de chaque point de l'espace et prendre les points avec les valeurs les plus hautes. Le fonctionnement est donc le même quelque soit la forme à détecter, tout ce qui change est l'équation qui va nous permettre de trouver le nombre de dimensions de notre espace de Hough et ses paramètres.

B/ Adaptation des algorithmes

Les deux premiers algorithmes vont fonctionner de la même manière quelque soit la forme à détecter, la seule différence se trouve au niveau de la détection de lignes aléatoires. Effectivement il ne suffira plus de prendre 2 points aléatoirement mais il faudra en prendre autant qu'il y à de paramètre dans notre équation donc ici avec l'exemple du cercle c'est 3 par exemple.