



INSTITUT
POLYTECHNIQUE
DE PARIS

TÉLÉCOM PARIS

IMA 201

IMPLÉMENTATION D'UN PAPIER DE RECHERCHE
RAPPORT

Segmentation du ventricule gauche d'images 3D+T

Élèves :

Maël LE GUILLOUZIC
Martin DE BATS

Enseignant :

Loic LE FOLGOC

Table des matières

1	Introduction	2
2	Le DataSet	3
2.1	Le Jeu de données ACDC	3
2.2	Contenu d'un patient	3
3	Placement des Seeds	6
3.1	Méthode de Hough : Théorie	6
3.1.1	Paramètres	6
3.1.2	Cas des cercles de rayon inconnu	7
3.1.3	Prétraitement des Images	8
3.2	Hough Naïf : pratique	8
3.3	Elaboration d'un masque	9
3.4	La motivation derrière le choix de Best Hough	11
3.5	Best Hough	12
3.6	Métrique d'évaluation des performances	15
3.7	Multiple Slices	16
3.8	Résultats du Placement des Seeds	17
4	Segmentation du ventricule gauche	18
4.1	Segmentation par croissance de région	18
4.2	Adaptation de l'algorithme à notre contexte	19
4.3	Choix du Treshold	20
4.4	Evaluation de la segmentation 2D	23
4.5	Meilleures Segmentations et Pires Segmentations	24
5	Passage au Volume	25
5.1	Algorithme pour obtenir la segmentation en 3D	25
5.2	Extension des métriques à la 3D	26
5.3	Performances 3D	27
5.4	Observation sur un patient	27
6	Passage de la 3D à la 3D+t	29
6.1	Étapes du processus	29
6.2	Implémentation	29
6.3	Resultats	30
6.4	Interprétation des Résultats	30
6.5	Problème des slices extrêmes	31
7	Pour aller plus loin	32
8	Annexe	33

1 Introduction

Selon l'Organisation Mondiale de la Santé (OMS), les maladies cardiovasculaires sont la **principale** cause de décès au niveau mondial. D'après les estimations, 17,9 millions de personnes sont mortes de maladies cardiovasculaires en 2019, soit **32%** de tous les décès dans le monde. Le détection précoce de ces pathologies constitue donc un réel enjeu de santé publique, en particulier dans les pays en voie de développement qui disposent d'un accès très inégal à la santé.

Pour cela, les cliniciens ont besoin d'effectuer des segmentations des cavités cardiaques. Ainsi ils peuvent accéder aux caractéristiques du muscle cardiaque comme par exemple son volume, sa masse, la fraction d'éjection. . . Cependant, cette tache fastidieuse à effectuer à la main, d'où la nécessité de produire des outils de segmentation semi-automatique voire automatique.

Etat de l'Art : Si aujourd'hui des approches basées sur des réseaux convolutifs (U-Net) et des Transformers dominent le sujet grâce à leur précision et leur robustesse, la segmentation d'imageries cardiaques existe depuis bien plus longtemps au travers de contours actifs et de croissance de région.

Objet du projet : L'objet de notre project est ainsi d'implémenter une méthode de segmentation par croissance de région pour segmenter le ventricule gauche. Pour l'intérêt scientifique de la chose, nous nous contentons d'utiliser des méthodes "*au pixel*".

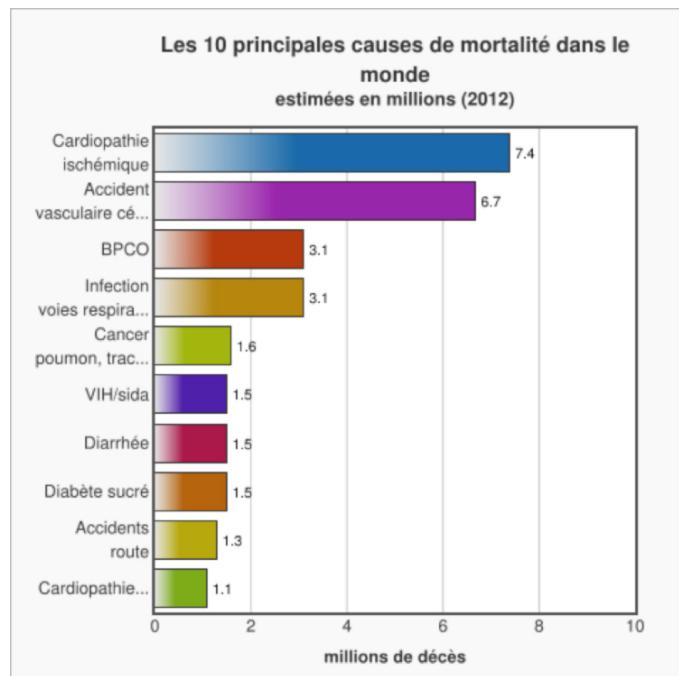


FIGURE 1 – Causes de mortalité dans le monde

2 Le DataSet

2.1 Le Jeu de données ACDC

Pour notre projet, nous utilisons le jeu de données ACDC (Automatic Cardiac Diagnosis Challenge) élaboré en 2017 à l'Hopital de Dijon dans le cadre d'un challenge de segmentation. Ce dernier qui offre un ensemble standardisé d'images 3D+T, ainsi que d'annotations utiles pour évaluer les performances de nos méthodes de segmentation.

Le DataSet est constitué de 150 patients, dont nous avons utilisé les 100 premiers pour l'élaboration de notre code ainsi que nos test récurrents. Les 50 autres n'ont jamais été touchés jusqu'à l'évaluation finale de nos performances sur ces derniers. En effet nous voulions à tout prix éviter des biais d'entraînement et cette partie test joue le rôle d'un test grandeur nature avec un patient nouveau sur lequel nous n'avons aucune information qui arrive.

2.2 Contenu d'un patient

Pour chaque patient, on dispose :

- D'une image 4D c'est à dire de plusieurs images 3D selon un axe temporel
- D'une ground truth, qui est en quelques sorte la solution au problème de segmentation que nous cherchons à résoudre. Cette GT n'est disponible que pour les instants de dystole et systole.

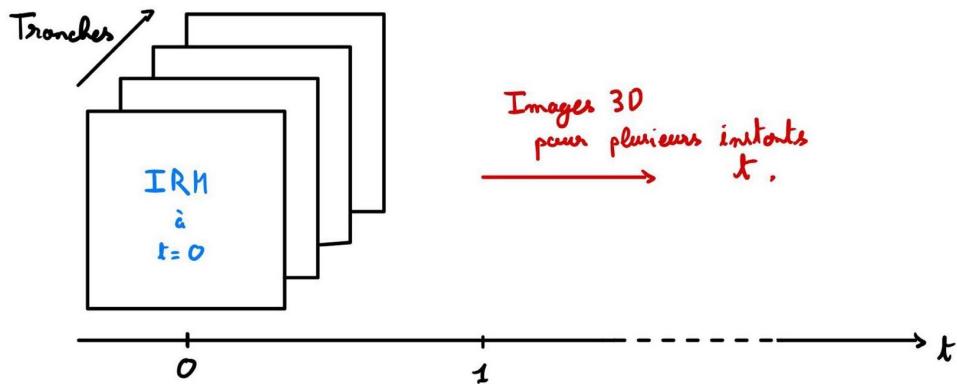


FIGURE 2 – Présentation du Dataset

Observons un patient. Pour cela nous importons les bibliothèques nécessaires ; nous utiliserons torchio pour visualiser nos images.

```
import torch
import torchio as tio

import numpy as np
import matplotlib.pyplot as plt
from skimage import measure
import pandas as pd

import cv2
from collections import deque

from ipywidgets import interact
import ipywidgets as widgets
```

Puis, chargeons un patient dont on observe une tranche centrale, à l'instant $t = 0$.

Pour cela on appellera la fonction `load_patient` que nous utiliserons tout au long du projet et qui renvoie pour un numéro i donné un object `tio.Subject` **patient** qui contient l'image 4D et la ground truth de ce patient i .

```
Tps = 0 # instant temporel d'observation
Slice = 11 // 3 # coupe à observer
Numero_patient = 2 # numéro du patient à observer

def load_patient(num, testing=False):
    path = 'database/training'
    if testing:
        path = 'database/testing'
        num += 100
    # load the dict of information on the patient
    with open(path + f'/patient{num:03}/info.cfg') as f:
        info = [line.replace("\n", "").replace(":", "").split(" ") for line in
                f]
    for i in range(len(info)):
        if info[i][1].isnumeric():
            info[i][1] = int(info[i][1])
    # create the patient object with (4Dimages, grandTrue, information)
    patient = tio.Subject(
        img=tio.ScalarImage(path +
                             f'/patient{num:03}/patient{num:03}_4d.nii.gz'),
        gt=tio.LabelMap(path +
                       f'/patient{num:03}/patient{num:03}_frame{dict(info)[["ED"]]:02}_gt.nii.gz'),
        info=dict(info))
    return patient
```

La région que nous cherchons à segmenter est ainsi la zone bleue, correspondants aux points de valeur 3 sur la ground truth.

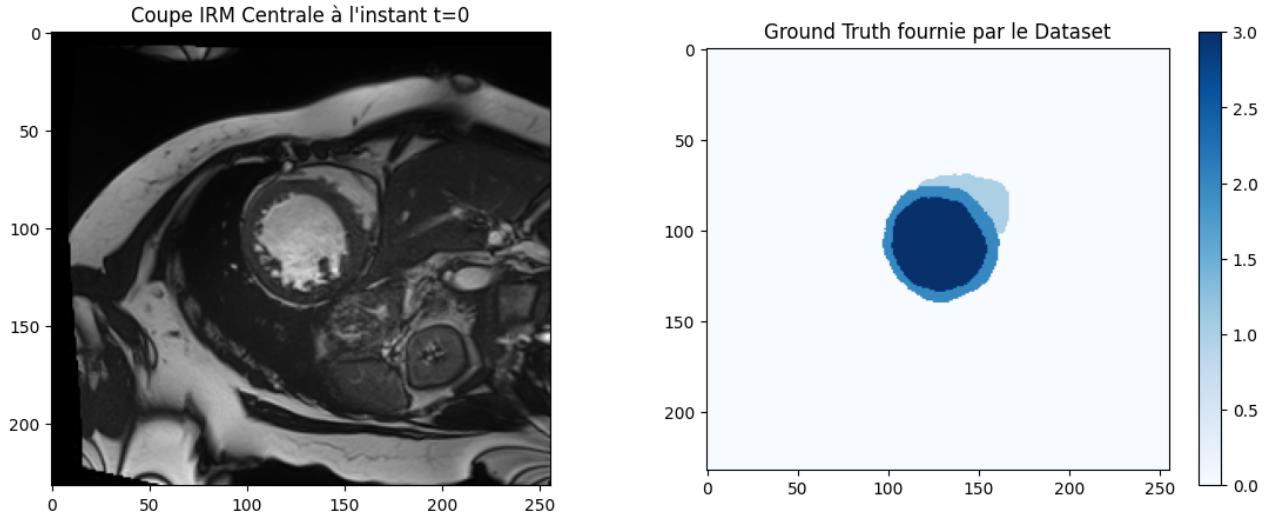


FIGURE 3 – Contenu d'un patient du Dataset

De plus chaque patient possède des informations supplémentaires dans un fichier texte associé. Ces dernières nous serons utiles plus loin, et on va donc les convertir en un dictionnaire de manière à les utiliser.

Attribut	Valeur
ED	1
ES	12
Group	DCM
Height	160.0
NbFrame	30
Weight	70

TABLE 1 – Données des attributs du patient.

3 Placement des Seeds

La méthode par croissance de région nécessite un point de départ.

En effet, la croissance de région repose sur l'**expansion** de la region segmentée de manière **itérative**. A chaque étape, les pixels au bord de la région regardent leurs voisins (dans notre code nous avons testés plusieurs choix de voisins différents, nous détaillons cela plus loin) et pour chacun d'entre eux, regardent si la différence d'intensité est inférieure à un seuil.

Ce seuil est crucial pour une bonne segmentation ; on comprend aisement que :

- un seuil trop faible bloque l'expansion de la region.
- un seuil trop grand étend notre région petit à petit à toute l'image. La region segmentée est ainsi bien trop grande par rapport à ce qui est attendu.

Ainsi, il apparaît que deux sujets deviennent importants à traiter :

- (a) Placer le point de départ dans le ventricule gauche.
- (b) Pouvoir choisir un seuil idéal pour chaque image et chaque patient.

Commençons par tenter de placer une seed.

3.1 Méthode de Hough : Théorie

En observant les images IRM de patients, nous remarquons que la région que nous souhaitons segmenter est la plupart du temps de forme à peu près ronde. Serait-il donc possible de détecter les cercles d'une image, et espérer que le cercle trouvé soit celui qui approxime le ventricule gauche.

Pour cela nous utilisons la méthode de la transformée de Hough Circulaire, qui nous permettra d'identifier la structure circulaire du ventricule gauche et ainsi de placer notre seed au bon endroit.

3.1.1 Paramètres

Un cercle dans un espace géométrique peut être défini par trois paramètres : a et b les coordonnées du centre du cercle et R le rayon de ce cercle. L'équation paramétrique s'écrit alors :

$$x = a + R \cos(\theta)$$

$$y = b + R \sin(\theta)$$

où θ varie sur $[0, 2\pi]$

L'idée de la méthode est alors le suivant : parmi les points représentant les contours de l'image, chacun génère un cercle de rayon R dans un espace des paramètres (a, b, R) .

Pour tout point de ce cercle généré, on incrémente un compteur de 1. Après avoir itéré sur toute l'image, les maxima du compteur correspondent aux emplacements les plus probables des centres de cercles de rayon R fixé.

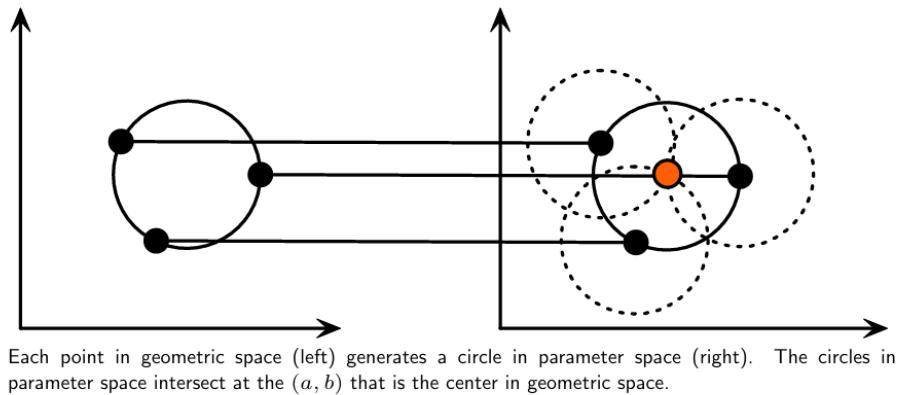


FIGURE 4 – Méthode de Hough pour R fixé

On vote alors dans une matrice dite **d'accumulation** pour toute les intersections proportionnellement au nombre de cercles qui s'y croisent.

3.1.2 Cas des cercles de rayon inconnu

Le cas des cercles de rayon connu, dérive simplement du cas suivant en ne considérant l'opération que dans un espace paramétrique bi-dimensionnel avec un rayon R fixé. Dans le cas des cercles de rayon inconnu :

Pour chaque rayon R :

- **Étape 1 :** Chaque point $P(x, y)$ de l'image génère une courbe circulaire de rayon R dans l'espace des paramètres (a, b)
- **Étape 2 :** Les intersections des cercles sont sauvegardées dans une matrice d'accumulation. Le maximum de cette matrice fournit un triplet (a, b, R) qui désigne le cercle le plus voté.

Remarque : pour le cas des rayons inconnus, on peut aussi le voir sous la forme : chaque point $P(x, y)$ génère une surface conique d'axe (a, b) .

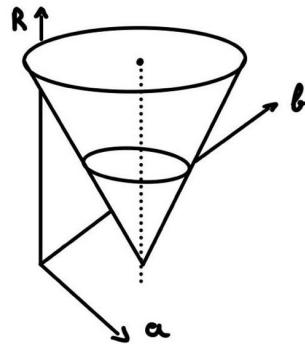


FIGURE 5 – Surface Conique parcourue pour Hough de rayon inconnu

3.1.3 Prétraitement des Images

Comme nous l'avons vu, il faut générer un cercle pour chaque pixel de l'image sur laquelle on cherche un cercle. Ainsi pour gagner en efficacité et en compléxité, on cherche le plus souvent à isoler les contours en effectuant diverses opérations :

- conversion en niveau de gris
- canny edge detector

3.2 Hough Naïf : pratique

Utilisons dans un premier temps une implémentation naïve, c'est à dire Hough (non codée par nous) sur une image du patient directement.

On choisit le premier instant temporel $t = 0$ et la tranche centrale.

```

hough_entry = np.array(t_img[Tps, :, :, Slice], dtype=np.uint8) # l'image doit
→ être convertie en uint8
circles = cv2.HoughCircles(hough_entry, cv2.HOUGH_GRADIENT, 1, 20,
                           param1=80, param2=40, minRadius=1, maxRadius=0) # → Hough

fig = plt.figure(figsize=(10,7)) # affichage
plt.imshow(hough_entry, cmap='gray')
if circles is not None:
    circles = np.uint16(np.around(circles))
    for i in circles[0,:]:
        cv2.circle(hough_entry, (i[0], i[1]), i[2], (255,255,255), 1) # on
→ trace l'extérieur du cercle
        cv2.circle(hough_entry, (i[0], i[1]), 2, (255,255,255), 1) # on trace
→ le point central
    plt.imshow(hough_entry, cmap='gray')
    plt.title('Cercles détectés avec un Hough Naïf')
    plt.show()

```

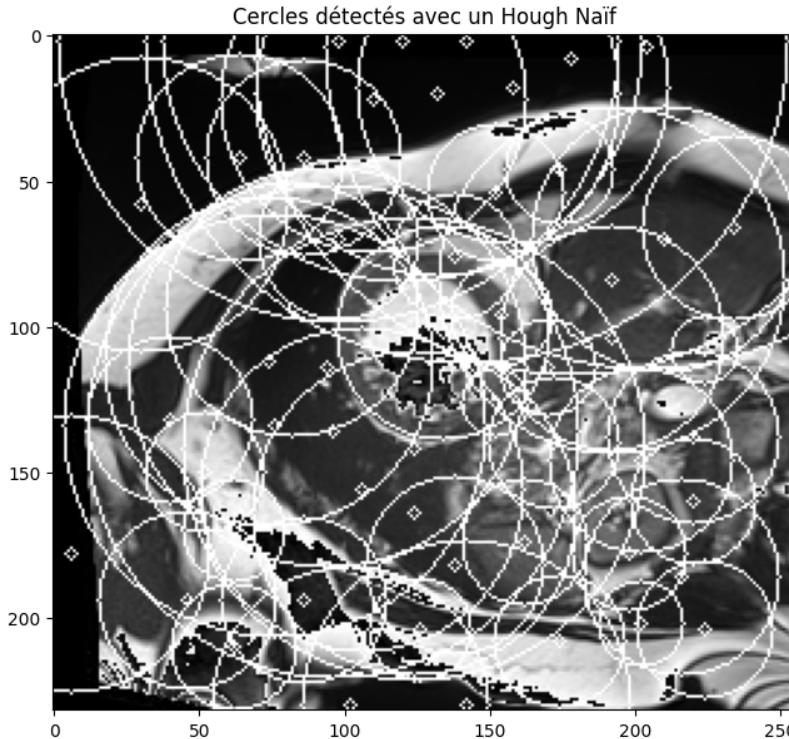


FIGURE 6 – Cercles détectés avec un Hough Naïf

On remarque deux choses principales :

- Le nombre de cercles détectés est trop important si on ne restreint pas les paramètres de la fonction de Hough. On va donc devoir agir sur le rayon minimal, maximal, ainsi que les paramètres utilisés pour le canny.
- Les cercles sont répartis sur l'intégralité de l'image. L'idée selon laquelle nous pouvons établir une carte du maximum de vraisemblance parmi de nombreux cercles nous semble compromise : il nous semble y avoir qu'un seul cercle correctement centré sur le ventricule gauche.

Cherchons donc une solution.

3.3 Elaboration d'un masque

On propose de définir une fonction *mask* qui va chercher à isoler une meilleure zone en amont de la recherche de cercles. Pour cela, on récupère dans les informations de chaque patient, la fréquence cardiaque au moment de l'IRM :

$$\text{Frquence Cardiaque} = \frac{L_t}{t_{systole} - t_{dystole}}$$

Où :

- L_t désigne le nombre d'instants temporels du patient étudié
- $t_{systole} - t_{dystole}$ désigne une demi période cardiaque.

Puis on effectue une **transformée de Fourier** de notre image et on ne conserve que la fréquence cardiaque précédemment isolée.

$$\text{Masque} = \text{TF}(\text{image})[\text{Fréquence Cardiaque}]$$

Enfin on obtient après dilatation, érosion et floutage, le masque final qu'on utilise pour chaque patient. En le multipliant à l'image d'origine, le résultat est une image IRM dont la majeure partie de l'image sauf le cœur est assombrie. Observons cela et l'effet sur la détection de cercles.

$$\text{Masque final} = \text{Blur}(\text{Erode}(\text{Dilate}(\text{Masque})))$$

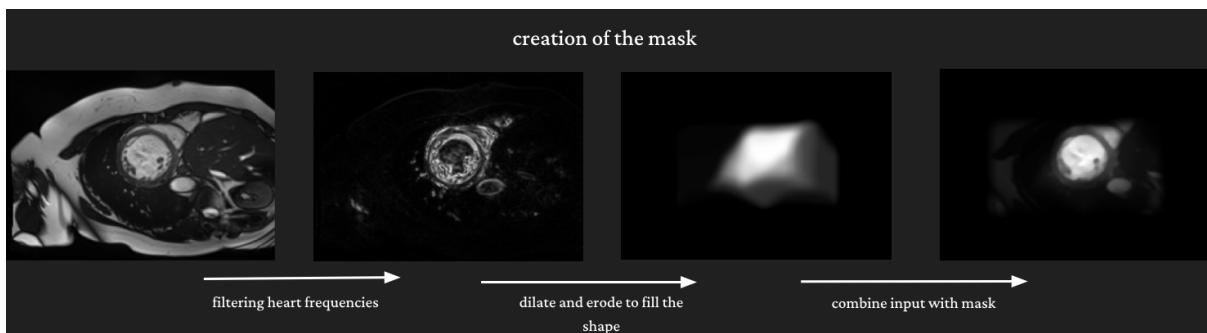


FIGURE 7 – Crédit d'un masque

```
def tf_mask(image4d, Tps=Tps, Slice=Slice, heartrate=2):
    fft_img = torch.fft.fft(image4d, axis=0)
    filtered = fft_img
    # interpolation fréquence cardiaque
    heartrate_i = int(heartrate)
    heartrate_f = heartrate - heartrate_i
    filtered[:heartrate_i, :, :, :] = 0.0
    filtered[heartrate_i+1:, :, :, :] = 0.0
    filtered[heartrate_i] *= (1-heartrate_f)
    filtered[heartrate_i+1] *= heartrate_f
    # fft inverse with the filtered input
    timg2 = torch.abs(torch.fft.ifft(filtered, axis=0))

    mask = np.array(timg2[Tps, :, :, Slice])
    mask = np.uint8(2*mask/np.max(mask)*255)
    # using morphology to go from a cercle to a disk
    mask = cv2.dilate(mask,
                      cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(110,110)), 1)
    mask = cv2.erode(mask,
                      cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(130,130)), 1,
                      borderType=cv2.BORDER_CONSTANT, borderValue=0)
    mask = cv2.blur(mask,(21, 21))
    # combine mask and input
    img_m = np.array(image4d[Tps, :, :, Slice], dtype=np.float64)
    img_m = np.uint8(img_m/np.max(img_m)*np.float64(mask))
    img_m = cv2.medianBlur(img_m,5)

    return img_m, mask
```

Affichons le résultat de l'opération de masquage sur le patient précédent.

On observe bien le résultat souhaité, mais avant de l'appliquer sur notre patient, on va essayer de définir une meilleure implémentation de Hough, qui serait capable d'ajuster les paramètres automatiquement.

```
def affichage_masque():
    # permet l'affichage de l'image suivante
    Fonction en annexe
```

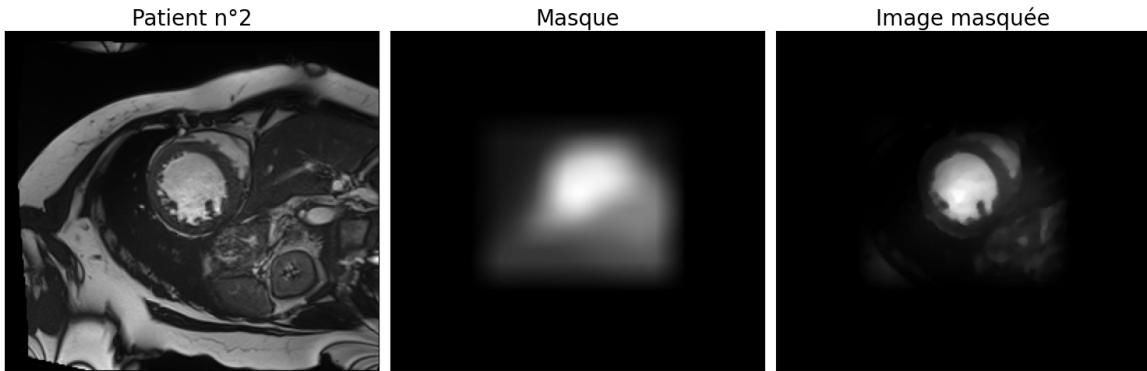


FIGURE 8 – Aperçu de l'image masqué pour le patient n°2

On définit simplement une fonction d'affichage des cercles, afin de pouvoir l'appeler simplement lors des usages suivants.

```
def to_rgb(gray):
    return cv2.cvtColor(gray, cv2.COLOR_GRAY2RGB)

def draw_cercles(image4d, circles=None, Tps=Tps, Slice=Slice):
    # construit une image constitué de la tranche étudiée ainsi que le ou les
    # cercles trouvés
    Fonction en annexe
```

3.4 La motivation derrière le choix de Best Hough

Pour déterminer le cercle décrivant notre ventricule gauche et ainsi trouver une seed située au centre de ce dernier nous avons techniquement deux options à notre disposition :

- Chercher de très nombreux cercles avec des paramètres de Hough permissifs, puis établir une carte de probabilité de présence du ventricule gauche. Si de nombreux cercles se trouvent à proximité de la région à trouver, alors cela peut fonctionner.
- Restreindre les paramètres de Hough pour n'obtenir qu'un seul cercle, qui sera alors on l'espère le bon.

Essayons d'afficher les cercles trouvés avec Hough et des paramètres permissifs à partir de l'image masquée.

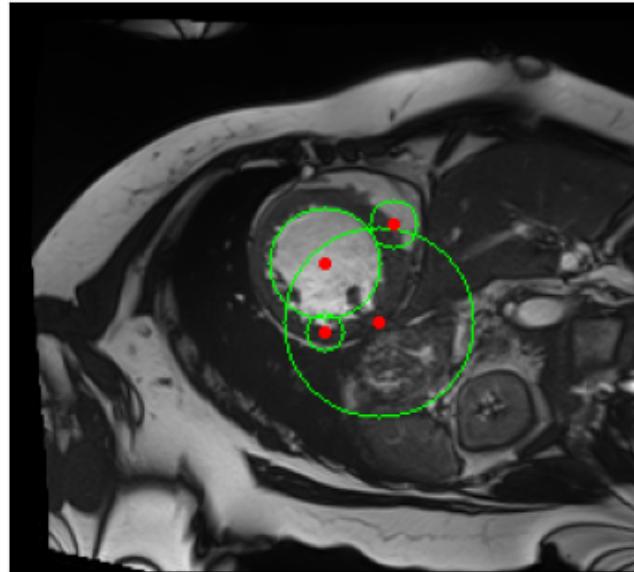


FIGURE 9 – Cercles trouvés pour des paramètres permissifs

On observe un élément clé sur cette image : les cercles trouvés ne sont pas tous centrés sur le ventricule. Il n'y en a même qu'un seul qui l'est correctement ...

Cette conclusion nous pousse à choisir la deuxième option, nous allons coder une fonction best_hough qui cherche les meilleurs paramètres de Hough pour n'obtenir qu'un seul cercle.

3.5 Best Hough

Notre première intuition est de chercher les paramètres de Hough par dichotomie. En effet, pour des paramètres **grands**, on obtient **aucun cercle** et pour des paramètres **faibles** on en obtient beaucoup. Une dichotomie permettra alors d'isoler les bons. On propose alors une dichotomie classique :

```
def hough_dichotomie(image):
    Initialisation : Paramètre_1, Paramètre_2 = 0,200
    while : on a pas isolé un unique cercle
        param_mean = (Paramètre_1+Paramètre_2)/2
        Effectuer Hough(Param_mean,Param_mean/2)
        if on a un seul cercle :
            return ce cercle.
        else:
            On ajuste Paramètre_1 ou Paramètre_2
```

FIGURE 10 – Pseudo-Code d'une dichotomie sur Hough

Néanmoins, lorsque nous implémentons cette approche, on observe qu'aucune seed n'est trouvée pour de nombreux patients du dataset. Cela s'explique par le fait que notre fonction peut tomber dans un minimum local, comme on le représente sur la figure ci dessous.

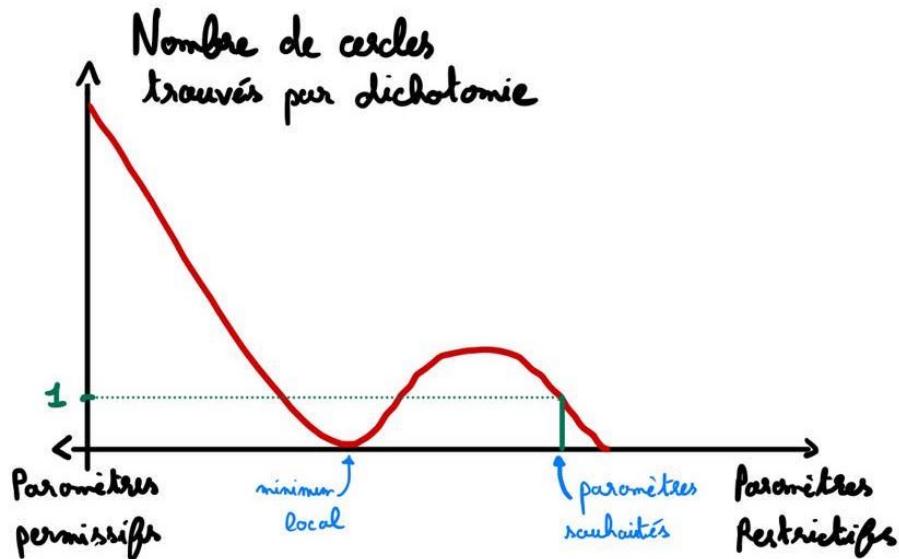


FIGURE 11 – Motivation d'une dichotomie améliorée

De plus, expérimentalement nous observons que pour un cercle unique, les paramètres qui ont tendance à être les plus grands trouvent le meilleur cercle. Ainsi nous proposons une deuxième implémentation de recherche des meilleurs paramètre :

```
def best_hough(image):
    pas = 5
    1. En partant de 200, on fait décroître le paramètre tant que aucun cercle est trouvé.
    2. Dès que un ou plusieurs cercles sont trouvés, on repart de notre avant dernier
       paramètre et on re-décroît avec un pas de 1.
    3. On return le cercle ainsi trouvé.
```

FIGURE 12 – Pseudo-Code de notre fonction Best Hough

Voici l'implémentation python de notre fonction :

```

def best_hough(image): # iterate + max dicothomie
    step = 5
    for param1 in range(180, 20, -step):
        # for param1 in range(20, 180, 10):
        circles = cv2.HoughCircles(image, cv2.HOUGH_GRADIENT, 1, 20,
                                    param1=param1, param2=param1/2, minRadius=10,
                                    maxRadius=50)

        if circles is not None:
            for param in range(param1+step, param1, -1):
                circles = cv2.HoughCircles(image, cv2.HOUGH_GRADIENT, 1, 20,
                                            param1=param, param2=param/2, minRadius=10, maxRadius=50)
                if circles is not None and len(circles[0]) == 1:
                    return circles[0][0]
            # return best_hough_max(image, param_max=param1+step,
            #                         param_min=param1, max_iter=10)
    return None

```

```

def draw_cercle(image4d, circles=None, Tps=Tps, Slice=Slice):
    # Image avec un unique cercle trouvé
    Fonction en annexe

def hough_apperçu(n1, n2):
    # génération de l'image ci dessous
    Fonction en annexe

```

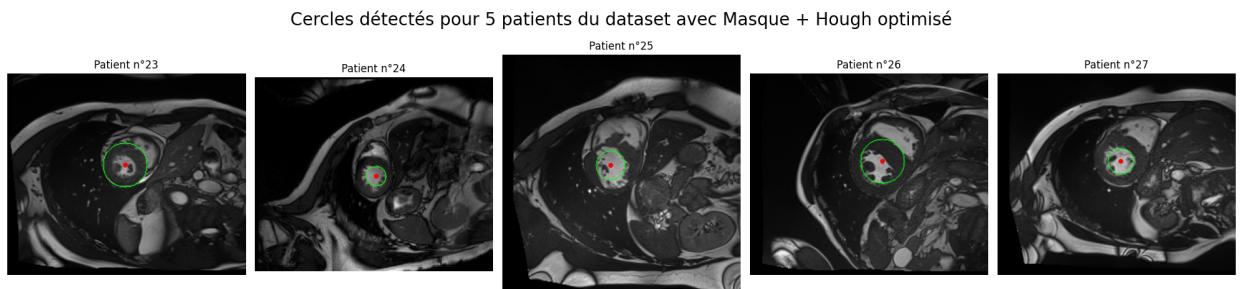


FIGURE 13 – Résultat de notre implémentation sur 5 patients au hasard

La fonction **find_seed** ci-dessous reprend les éléments présentés précédemment, et réalise pour une image donnée la recherche d'une seed.

```

def find_seed(patient, Tps=Tps, Slice=Slice):
    # first create the image masked and then performs a Hough Transform
    t_img = patient.img.data.clone().detach().float()
    hr = t_img.shape[0] / (patient.info["ES"] - patient.info["ED"])
    img_m, _ = tf_mask(t_img, heartrate=hr, Tps=Tps, Slice=Slice)
    circles = best_hough(img_m)
    return circles

```

3.6 Métrique d'évaluation des performances

Pour évaluer la performance de notre méthode, on dispose d'une **ground truth** pour chaque patient que nous avons décrite dans la partie 2 de ce papier.

Il est donc possible à partir de cette dernière de définir deux métriques :

- Distance : La distance entre la seed trouvée et le centre de la ground truth

$$\text{Distance} = \|\text{seed} - \text{centre(gt)}\|_2$$

- Hit : un booléen qui indique simplement si la seed est **dans** ou **hors** du ventricule gauche de la ground truth.

$$\text{Hit} = \begin{cases} \text{True} & \text{si } \text{seed} \in \text{gt} = 3, \\ \text{False} & \text{sinon.} \end{cases}$$

On en profitera pour renvoyer une troisième métrique, qui dira simplement si une seed (donc un cercle) a été trouvée ou non.

```
def patient_metrics(patient, circles, Slice=Slice):
    # this give 3 metrics: (1: circle found ?  2: distance between center and
    # → groundtruth  3: center in the LV ?)
    t_gt = patient.gt.data.clone().detach().float()
    center = (t_gt[0, :, :, Slice]==3).argwhere().mean(axis=0,
    # → dtype=torch.float32)
    if circles is None:
        return False, 0., False

    dist = torch.dist(center, torch.tensor(circles[:2]))
    gt_map = patient.gt.data.clone().detach().float()
    hit = (gt_map[0, int(circles[1]), int(circles[0]), Slice] == 3)

    return True, dist, hit
```

Evaluons ces 3 métriques sur notre dataset de training, puis sur celui de test (que nous n'avons ouvert qu'une fois le projet terminé et que plus aucune modification ne fut apportée au code).

```
-= results =-
accuracy: 98.0% (over found circles: 98.0)
miss: 0
avg dist: 42.95237731933594
=====
```

Les performances sur la partie **training** ainsi que la partie **test** sont les suivantes. L'accuracy correspond au pourcentage de *Hits* sur l'ensemble sur lequel on travaille.

	Training	Test
Cercle Trouvé	93%	98%
Accuracy	93%	98%
Average Distance	41.16	42.95

TABLE 2 – Performances sur les ensembles training et test.

3.7 Multiple Slices

Comme on peut le remarquer, il reste quelques images sur lesquelles on ne trouve pas de seed. Cela est problématique pour la suite car sans point de départ, on peut dire adieu à toute segmentation. Essayons donc de trouver une méthode qui pourrait aller chercher ces quelques pourcents manquants. Pour cela, on introduit la fonction suivante :

```
find_seed_multiple_slices(patient, Tps=Tps)
```

Son objectif est simple : au lieu de chercher une seed sur une tranche du patient à l'instant t , on va chercher des seeds sur toutes les tranches. On espère alors que si l'une des tranches ne permet pas de trouver de cercle, ce n'est pas le cas de toutes les tranches.

Une fois les seeds trouvées, on conserve uniquement la seed qui a la distance minimale par rapport à toutes les autres. Et avec S l'ensemble des Seeds :

$$\text{seed} = \min(\text{dist}(\text{autres seeds})) = \underset{s_1 \in S}{\text{argmin}} \sum_{s_2 \in S} \|s_1 - s_2\|^2$$

```
def find_seed_multiple_slices(patient, Tps=Tps):
    seeds = []
    for slice in range(0, patient.img.data.shape[3]):
        seed = find_seed(patient, Tps=Tps, Slice=slice)
        if seed is not None:
            seeds += [seed]
    seeds = np.array(seeds)

    if len(seeds) == 0:
        return None

    dist = []
    for s in seeds:
        d = seeds - s
        d = d * d
        dist += [np.sum(d)]

    avg_slice = np.argmin(dist)
    return seeds[avg_slice], avg_slice
```

Et ainsi, en mesurant les performances à l'aide de notre nouvelle méthode, sur le **training** ainsi que sur le **testing**, on obtient les résultats suivants :

	Training	Test
Cercle Trouvé	97%	100%
Accuracy	97%	100%

TABLE 3 – Performances de la nouvelle méthode sur les ensembles training et test.

Voici les résultats obtenus, avec une liste des patients manqués et les statistiques globales :

```
miss on patient n° 24, n°33, n°37
-= results --
accuracy: 97.0% (over found circles: 97.0)
miss: 0
avg dist: 42.5
=====
```

3.8 Résultats du Placement des Seeds

On synthétise les performances de nos divers algorithmes avec le graphique suivant. Comme on peut l'observer, la performance sur le test est supérieure à celle sur le training, ce qui peut paraître surprenant mais peut s'expliquer par le faible nombre d'images.

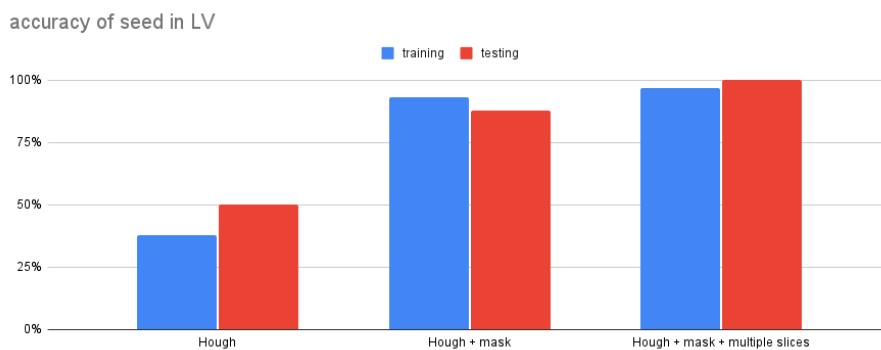


FIGURE 14 – Accuracy sur les deux parties du DataSet pour les différentes méthodes

100% de réussite sur la partie test du dataset !

C'est un score suffisant pour passer à la suite du projet.

Remarque : Si on observe les 3 images qui posent problème, la seed trouvée est systématiquement très proche du ventricule, voire juste à la bordure.

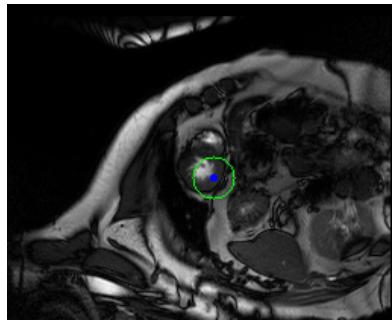


FIGURE 15 – Patient 24.

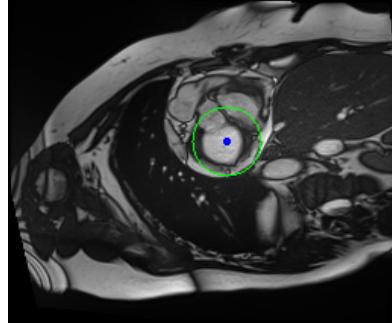


FIGURE 16 – Patient 33.

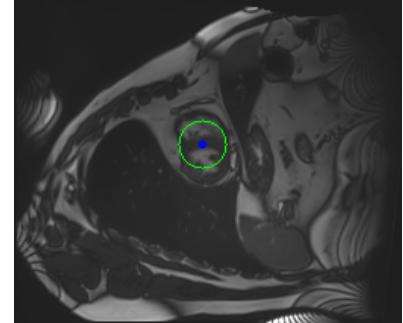


FIGURE 17 – Patient 37.

4 Segmentation du ventricule gauche

Une fois une seed correctement placée au centre du ventricule gauche, nous pouvons chercher à segmenter ce dernier en utilisant une approche par croissance de région.

4.1 Segmentation par croissance de région

Le principe de la segmentation par croissance de région est simple et peut être décrit par les étapes suivantes :

- **Initialisation** : On initialise la région comme vide. On marque tous les pixels de l'image comme non explorés, et on ajoute la seed (non explorée) à la région. On détermine un **threshold** qui sera le critère d'arrêt de notre segmentation.
- **Tant qu'il existe** des pixels de la région marqués comme non explorés :
 - on choisit l'un de ces pixels
 - on regarde ses 4 voisins directs. Pour chacun d'entre eux, si la différence en valeur absolue de l'intensité du pixel examiné et de son voisin est inférieure au threshold, on ajoute le voisin à la région.
 - on marque le pixel comme étant examiné.

C'est la région obtenue en sortie de la boucle qui doit correspondre à notre ventricule gauche.

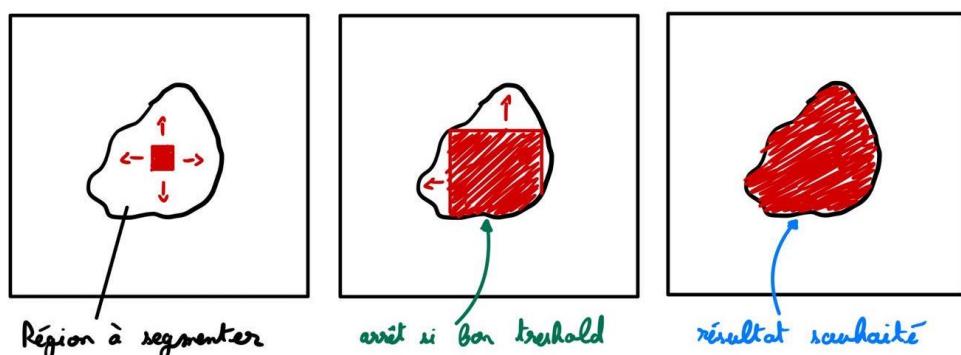


FIGURE 18 – Schéma simplifié région growing

L'algorithme est assuré de se terminer, puisque le nombre de pixels de l'image est fini, et sa compléxité dans le pire des cas est $O(n^2)$.

4.2 Adaptation de l'algorithme à notre contexte

L'algorithme peut être ajusté pour mieux s'adapter à notre situation spécifique. Les améliorations suivantes ont été introduites :

1. Utilisation de la médiane d'intensité autour de la seed :

- Au lieu de comparer directement les différences d'intensité entre un pixel et ses voisins, nous estimons d'abord la valeur médiane d'intensité du ventricule gauche dans une fenêtre de 11×11 autour de la seed.
- Ensuite, les nouveaux pixels sont comparés à cette médiane.
- Cette approche réduit les phénomènes de "fuite", qui peuvent se produire à cause du gradient d'intensité souvent observé dans le ventricule gauche.

2. Prise en compte de la zone sombre entourant le ventricule :

- Il est intéressant de noter que le ventricule gauche est généralement entouré d'une zone d'intensité plus faible.
- Pour exploiter cette propriété, nous adaptons le seuil de comparaison pour privilégier les valeurs supérieures à la médiane.
- Cela permet de mieux segmenter les zones de forte intensité situées au centre du ventricule, tout en excluant efficacement les zones environnantes.

```
def region_growth(image, seed, error=10):  
    H, L = image.shape  
    # Estimation de la moyenne  
    moy = np.median([image[seed[1] + i, seed[0] + j] for i in range(-2, 3) for  
    → j in range(-2, 3)])  
  
    # Initialisation  
    to_explore = deque([np.array(seed)])  
    dxy = np.array([[0, -1], [-1, 0], [1, 0], [0, 1]])  
    left_V = np.zeros_like(image)  
    valide = (moy - image < error) & (np.arange(H)[:, None] > 0) &  
    → (np.arange(H)[:, None] < H-1) & (np.arange(L) > 0) & (np.arange(L) <  
    → L-1)  
  
    # Croissance de région  
    while to_explore:  
        y, x = to_explore.popleft()  
        if valide[y, x] and not left_V[y, x]:  
            left_V[y, x] = 1  
            to_explore.extend(dxy + [x, y])  
  
    return left_V, np.sum(left_V)
```

4.3 Choix du Threshold

L'étape délicate d'une telle méthode de segmentation réside dans le choix du threshold. En effet, observons la segmentation décrite ci dessus pour quelques valeurs de threshold différentes.

Pour cela nous définissons la fonction *region_growth* qui reprend l'approche expliquée ci-dessus, à partir d'un threshold fourni en entrée. La fonction retourne la région segmentée sous forme d'une région **connexe** de 1 dans une image de 0, ainsi que la taille de cette région.

On peut ainsi observer la segmentation produite, pour 3 valeurs de thresholds.

$$\text{threshold} = [30, 120, 150, 200]$$

Le résultat est le suivant.

```
thresholds = [30, 120, 150, 200]
def comparaison_threshold(Numero_patient, thresholds):
    # affichage de l'image ci-dessous
    Fonction en annexe
comparaison_threshold(Numero_patient, thresholds)
```

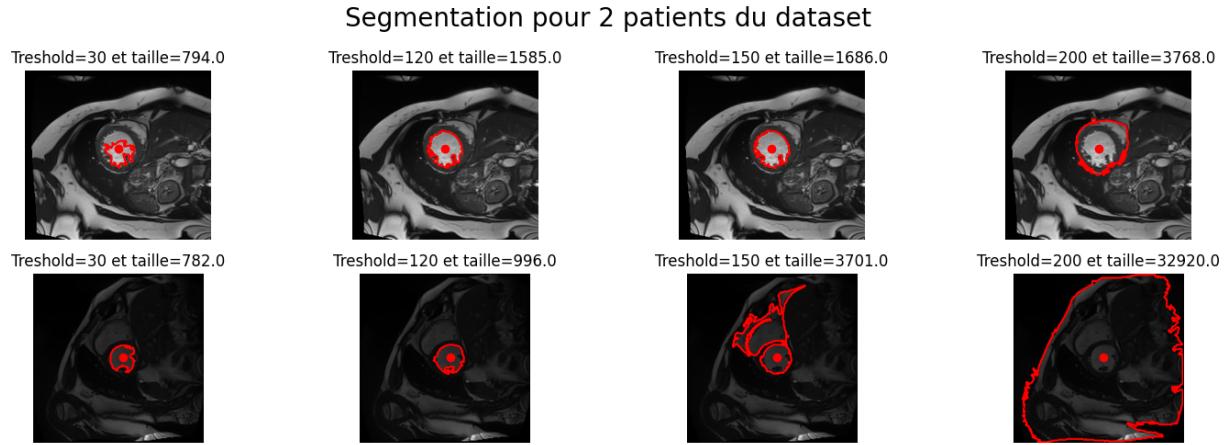


FIGURE 19 – Visualisation de thresholds différents pour 2 patients

Pour deux patients différent du dataset, les thresholds optimaux sont différents. En effet pour le patient n°28, la région segmentée explose au delà de 120, tandis que pour le patient n°33, un threshold de 150 ne suffit toujours pas à faire exploser la région. Les variations de luminosité entre les images et le contraste du ventricule gauche soulignent un point important : **le seuil (threshold) doit être adapté pour chaque patient.**

Pour cela, nous proposons l'approche suivante :

1. **Réaliser une segmentation par region growing avec différents paramètres de seuil.**
2. **Évaluer chaque segmentation à l'aide d'une métrique spécifique pour déterminer la segmentation la plus plausible :**
 - La métrique que nous utilisons s'appuie sur le disque détecté par la transformée de Hough.
 - Nous cherchons une segmentation qui correspond au disque tout en évitant les débordements.
 - Soit D le disque détecté, la métrique est définie comme suit :

$$m(S, D) = \frac{2 \cdot |S \cap D| - |S \cap \bar{D}|}{|S| + |D|}$$

où S est la segmentation, D est le disque, et \bar{D} représente la région extérieure au disque.

3. **Sélectionner la segmentation qui minimise cette métrique, ainsi que le seuil correspondant.**

```
def leftv_seg(img, full_seed=None, Tps=Tps, Slice=Slice):
    if full_seed is None:
        return None

    seed = [int(full_seed[0]), int(full_seed[1])]
    rayon = full_seed[2]

    xx, yy = np.mgrid[:img.shape[0], :img.shape[1]]
    circle = (xx - seed[1]) ** 2 + (yy - seed[0]) ** 2
    circle_map = np.zeros(img.shape, np.bool_)
    circle_map[circle < rayon*rayon] = True

    treshold = 0
    dices = []

    step = 5
    for treshold in range(5, 150, step):
        lv_seg, taille_1 = region_growth(np.array(img), seed, treshold)
        lv_seg = np.bool_(lv_seg)

        inter = np.sum(lv_seg & circle_map)
        diff_ext = np.sum(lv_seg & np.logical_not(circle_map))
        dice = (2*inter-diff_ext) / (np.sum(lv_seg) + np.sum(circle_map))
        dices += [(dice, treshold)]

    treshold = max(dices)[1]
    lv_seg, taille_1 = region_growth(np.array(img), seed, treshold)

    lv_out = cv2.dilate(lv_seg,
                        cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(59,59)), 1)
    lv_out = cv2.erode(lv_out,
                        cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(59,59)), 1)
    return lv_out

def draw_region(imageRGB, lv_seg):
    # Affiche le résultat de la segmentation précédente
    Fonction en annexe
```

pour évaluer nos segmentations, on introduit la mesure du DICE entre deux ensembles :

$$DICE(A, B) = \frac{2 \cdot |A \cap B|}{|A| + |B|}$$

où :

- A : segmentation prédictive
- B : vérité terrain
- $|A \cap B|$: taille de l'intersection des ensembles

Interprétation du DICE :

- **0** : aucun chevauchement
- **1** : chevauchement parfait

```
def dice(patient, lv_seg, Slice=Slice):
    t_gt = patient.gt.data.clone().detach().float()
    gt_map = t_gt[0, :, :, Slice]
    gt_map = (gt_map == 3)

    lv_seg = np.bool_(lv_seg)
    gt_map = np.bool_(gt_map)
    inter = np.sum(lv_seg & gt_map)

    return 2 * inter / (np.sum(lv_seg) + np.sum(gt_map))
```

4.4 Evaluation de la segmentation 2D

A la lumière de ce que nous venons de décrire, il est donc possible de calculer la segmentation 2D à l'instant 0 pour tous les patients du Dataset, et de calculer la mesure de DICE pour chacun d'entre eux. Le résultat d'une telle fonction est donné ci dessous :

```
def evaluate_avg_dice2D(nb):
    # Fonction qui réalise la segmentation 2D pour tous les patients du DataSet
    # → à l'instant 0 et calcule le DICE
    Fonction en annexe
```

```
==== results ===
avg dice: 0.8574507052659606
=====
```

En exécutant l'algorithme sur les ensembles d'entraînement et de test, nous obtenons les résultats suivants :

- **DICE sur l'entraînement** : 85.7%
- **DICE sur le test** : 86.7%

Interprétation : Un DICE supérieur à 0.8 indique des segmentations très proches de la vérité terrain.

Ces résultats montrent que l'algorithme généralise bien. Le score légèrement plus élevé sur l'ensemble de test (0.86 contre 0.82) suggère que le modèle :

1. **N'est pas surentraîné**, évitant ainsi de sur-ajuster les données d'entraînement.
2. Pourrait bénéficier d'un ensemble de test contenant des cas plus homogènes.

Cela confirme que l'approche est robuste et bien adaptée à la segmentation du ventricule gauche dans divers scénarios.

4.5 Meilleures Segmentations et Pires Segmentations

Essayons d'observer les 5 meilleures segmentations de notre dataset, ainsi que les 5 moins bonnes.

```
def best_worst(seg):
    # Affichage des 5 pires cas et des 5 meilleurs
    Fonction en annexe
```

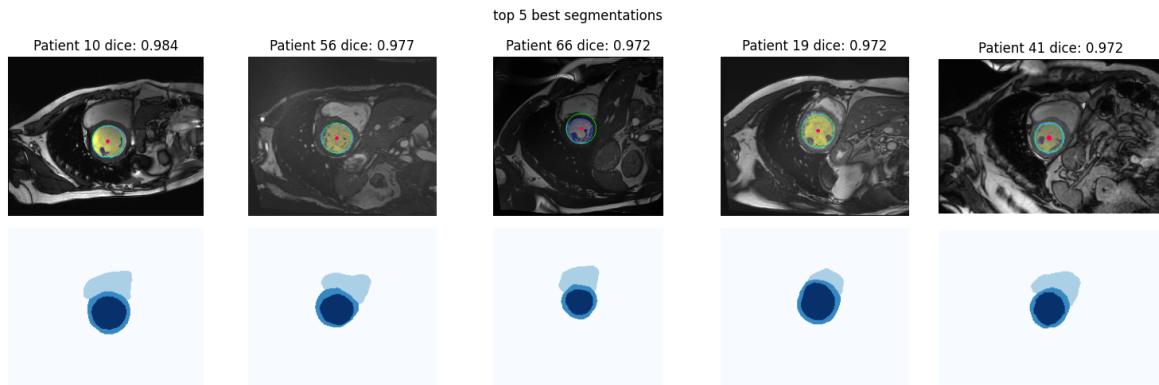


FIGURE 20 – 5 meilleures segmentations

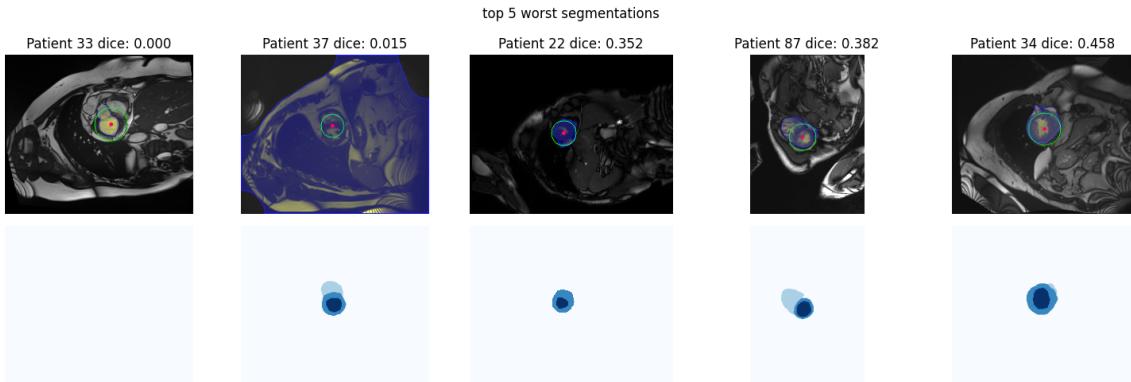


FIGURE 21 – 5 moins bonnes segmentations

5 Passage au Volume

Nous disposons désormais d'une méthode robuste pour segmenter le ventricule gauche sur des images 2D. L'étape suivante consiste à étendre cette segmentation aux images 3D+t.

Cependant, la ground truth n'est disponible que pour une seule coupe temporelle, correspondant à la diastole. Dans un premier temps, nous allons donc analyser la propagation de la seed en 3D, en nous concentrant sur cette unique coupe temporelle.

5.1 Algorithme pour obtenir la segmentation en 3D

Pour obtenir la segmentation en 3D, nous utilisons l'algorithme suivant :

1. Utiliser *find_seed_multiple_slices* pour trouver une seed et une slice initiale.
2. Effectuer la segmentation 2D sur cette slice initiale.
3. Propager la segmentation aux slices voisines : Pour chaque slice voisine d'une slice déjà segmentée, on utilise la segmentation précédente pour définir le cercle de la seed de la slice suivante.
 - o Soit S_{n-1} la segmentation de la slice précédente
 - Le centre c_n de la seed de la slice suivante est donné par le barycentre de la segmentation précédente :

$$c_n = \frac{1}{|S_{n-1}|} \sum_{p \in S_{n-1}} p$$

- Pour estimer le rayon R_n de la seed, on impose que le cercle de rayon R_n ait approximativement le même volume que la segmentation S_{n-1} , soit :

$$|S_{n-1}| = \pi R_n^2$$

D'où :

$$R_n = \sqrt{\frac{|S_{n-1}|}{\pi}}$$

```
def slice3D(patient):
    ED = patient.info["ED"] - 1 # Détermine l'indice de la tranche de la
    ↵ vérité terrain.
    seed0, sslice = find_seed_multiple_slices(patient, ED)

    img_3D = patient.img.data[ED]
    lv3D = np.zeros_like(img_3D)

    if seed0 is None:
        return lv3D

    # Segmentation initiale
    lv3D[:, :, sslice] = leftv_seg(img_3D[:, :, sslice], seed0, Tps=Tps,
    ↵ Slice=sslice)

    # Propagation de la seed vers les tranches suivantes
    for s in range(sslice+1, lv3D.shape[2]):
        lv_1 = lv3D[:, :, s-1]
        idxs = np.where(lv_1 == 1)
        if len(idxs[0]) == 0:
            break
        center = np.mean(idxs, axis=1)
        radius = np.sqrt(len(idxs[0]) / np.pi) # Rayon calculé à partir du
        ↵ volume
        lv3D[:, :, s] = leftv_seg(img_3D[:, :, s], [center[1], center[0],
        ↵ radius], Tps=Tps, Slice=sslice)

    # Propagation de la seed vers les tranches précédentes
    for s in range(sslice-1, -1, -1):
        lv_1 = lv3D[:, :, s+1]
        idxs = np.where(lv_1 == 1)
        if len(idxs[0]) == 0:
            break
        center = np.mean(idxs, axis=1)
        radius = np.sqrt(len(idxs[0]) / np.pi)
        lv3D[:, :, s] = leftv_seg(img_3D[:, :, s], [center[1], center[0],
        ↵ radius], Tps=Tps, Slice=sslice)

    return lv3D
```

5.2 Extension des métriques à la 3D

Après avoir obtenu cette segmentation en 3D, nous pouvons maintenant étendre les métriques pour les adapter à des mesures en 3D. Plusieurs approches sont possibles :

1. Calculer la moyenne des mesures DICE sur chaque slice individuelle.
2. Effectuer une mesure DICE globale en 3D sur l'ensemble de la segmentation 3D.

Dans notre cas, nous avons choisi la seconde méthode : calculer un DICE global en 3D sur la segmentation obtenue à partir de chaque segmentation 2D.

Bien que nous travaillions en 3D, la formule du DICE reste identique :

$$DICE(A, B) = \frac{2|A \cap B|}{|A| + |B|}$$

Cette approche nous permet d'évaluer directement la qualité de la segmentation 3D en tenant compte de l'ensemble du volume segmenté.

5.3 Performances 3D

On peut ainsi calculer les performances pour l'instant $t = 0$ (puisque la GT n'est disponible que sur cet instant) sur l'ensemble du Dataset. Les performances sont ci dessous :

```
avg_dice = 0
nb = 50
found = 0
for i in range (nb):
    patient = load_patient(i+1, testing=True)
    seg3D = slice3D(patient)
    if seg3D is not None:
        dice = dice3D(patient, seg3D)
        avg_dice += dice
        found += 1
        print(f"Patient {i+1} : {(100*dice):.2f} %")
avg_dice /= found
print(f"Average Dice : {100*avg_dice:.3f} % (found: {found})")
```

==== Results ===

AVG Dice3D sur Training: 72.010% (found: 100)
AVG Dice3D sur Test: 77.108 % (found: 50)

=====

5.4 Observation sur un patient

Visualisons notre segmentation 3D sur un patient du Dataset. Pour cela on affiche toutes ses tranches à l'instant 0.

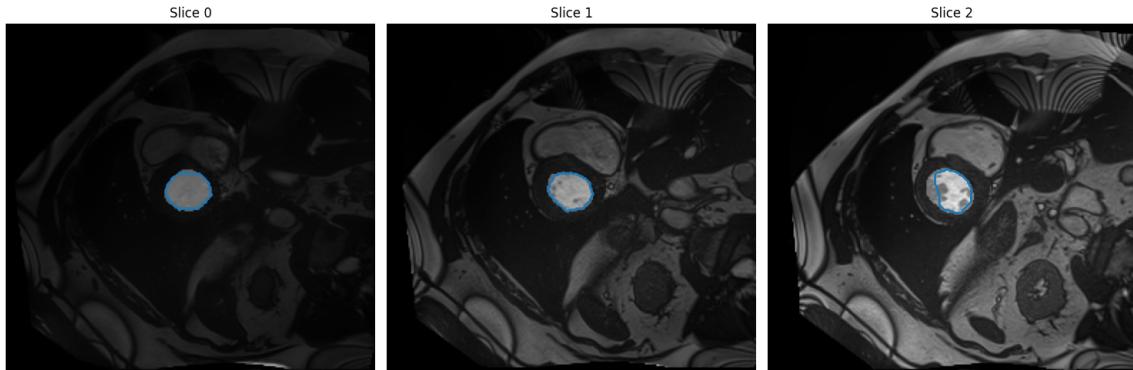


FIGURE 22 – 3 premières tranches

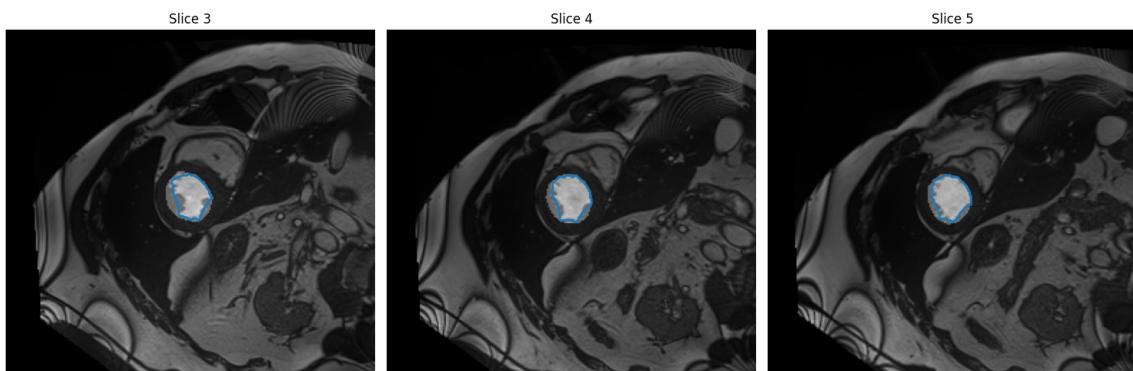


FIGURE 23 – 3 suivantes

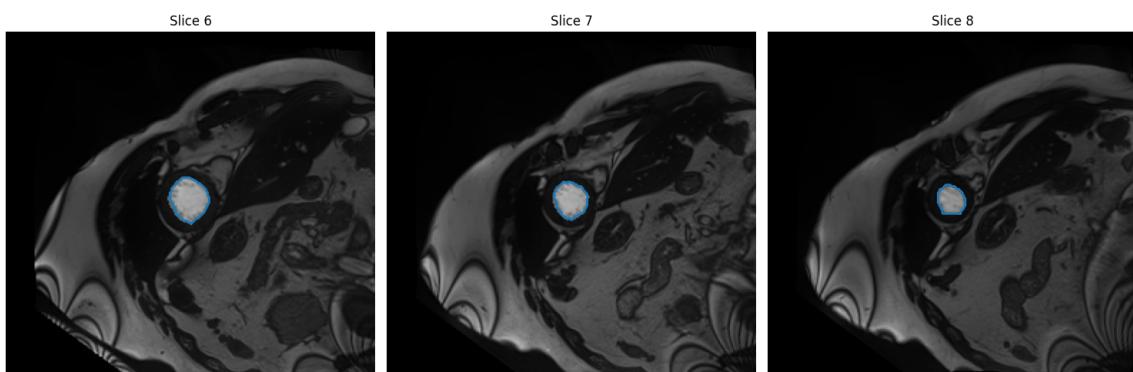


FIGURE 24 – Segmentation 3D sur les tranches du patient 32

6 Passage de la 3D à la 3D+t

Le processus de segmentation en 3D+t (avec une composante temporelle) est similaire à la segmentation classique en 3D. Cependant, il ajoute une dimension temporelle à la propagation.

6.1 Étapes du processus

1. Propagation temporelle :

- La segmentation commence sur une slice à un instant donné (ED).
- Cette segmentation est propagée dans le temps vers les autres instants temporels.

2. Propagation spatiale :

- Pour chaque instant temporel, on utilise la segmentation obtenue pour cet instant comme point de départ.
- La propagation est alors effectuée sur les autres slices de l'espace 3D, suivant la méthode classique de segmentation 3D.

Propagation de la seed:

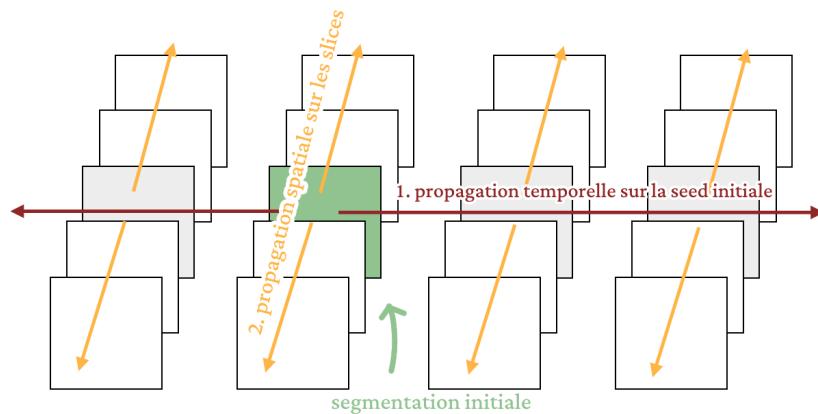


FIGURE 25 – Schéma Propagation des Seeds

6.2 Implémentation

Cette première fonction *expend3D* étend les seeds à toutes les slices de notre patient, pour un **unique** instant temporel.

```

def expend3D(img_3D, lv3D, sslice):
    # seed propagation
    for s in range(sslice+1, lv3D.shape[2]):
        lv_1 = lv3D[:, :, s-1]
        idxs = np.where(lv_1 == 1)
        if len(idxs[0]) == 0:
            break
        center = np.mean(idxs, axis=1)
        radius = np.sqrt(len(idxs[0])/np.pi) # find radius from volume
        lv3D[:, :, s] = leftv_seg(img_3D[:, :, s], [center[1], center[0],
                                                     radius], Tps=Tps, Slice=sslice)

    for s in range(sslice-1, -1, -1):
        lv_1 = lv3D[:, :, s+1]
        idxs = np.where(lv_1 == 1)
        if len(idxs[0]) == 0:
            break
        center = np.mean(idxs, axis=1)
        radius = np.sqrt(len(idxs[0])/np.pi)
        lv3D[:, :, s] = leftv_seg(img_3D[:, :, s], [center[1], center[0],
                                                     radius], Tps=Tps, Slice=sslice)

```

Puis, la fonction *slice3D_t* réalise la dernière opération qui condense toutes les précédentes : réaliser la segmentation sur toutes les slices d'un patient et pour tous les instants temporels.

```

def slice3D_t(patient):
    # Fonction qui reprend l'intégralité des opérations précédentes et réalise
    # la segmentation 3D à tous les instants temporels
    Fonction en annexe

```

6.3 Résultats

Après avoir exécuté nos algorithmes, nous mesurons la qualité des segmentations à l'aide du **DICE** sur la slice temporelle correspondant à la ground truth.

	Entraînement	Test
DICE 3D	76.1%	77.1%

TABLE 4 – Performances 3D

Visualisons notre segmentation sur un patient, en fonction du temps. De plus on peut grâce à cette dernière tracer la courbe du volume cardiaque en fonction du temps.

6.4 Interprétation des Résultats

Ces résultats représentent des **moyennes globales**, mais elles sont fortement influencées par quelques cas extrêmes où le DICE est faible.

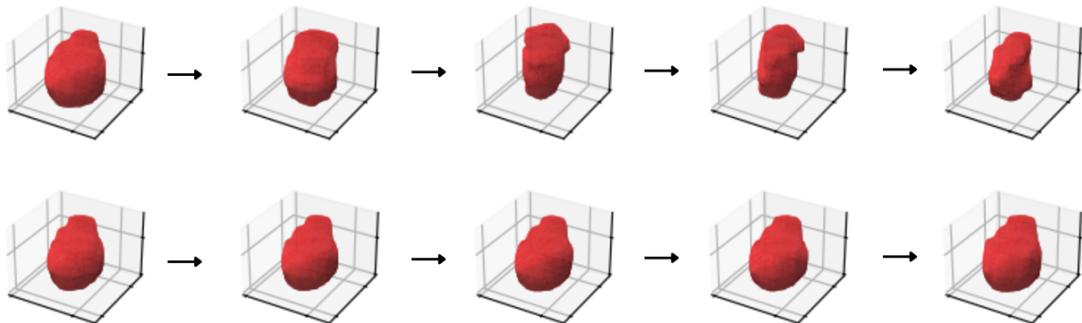


FIGURE 26 – Segmentation 3D+T

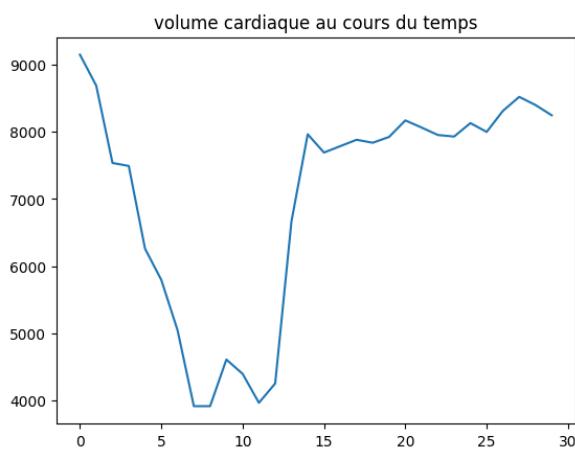


FIGURE 27 – Volume du ventricule gauche segmenté

- **Observation générale :** La majorité des scores DICE se situent autour de **88%**, indiquant des segmentations proches de la vérité terrain dans la plupart des cas.

6.5 Problème des slices extrêmes

Lors de la segmentation temporelle, certaines slices montrent un **volume anormalement grand**, dû à une croissance de région excessive. Ce phénomène se produit principalement :

- Sur les slices temporelles extrêmes (où le cœur est à peine visible).
- En raison du débordement de la segmentation sur des régions hors du ventricule.

Pour corriger ce problème, il serait pertinent d'implémenter un **système de détection des slices anormales**, basé sur des critères de cohérence spatiale et temporelle avec les slices adjacentes.

Un tel système permettrait de limiter les artefacts et d'améliorer encore la robustesse de l'algorithme.

7 Pour aller plus loin

Plusieurs pistes peuvent être explorées pour améliorer nos algorithmes et aller au-delà des approches actuelles :

1. **Approximation du ventricule par un plan** : Une première idée est de modéliser l'intensité du ventricule gauche par un plan, en exploitant le gradient d'intensité observé à travers le ventricule. Cette méthode, inspirée d'une publication, peut guider le processus de segmentation. Cependant, en pratique, cette approche a montré une plus grande propension à laisser le region growing s'étendre dans des zones en dehors du ventricule. Notre version modifiée de region growing s'est avérée plus performante sur les données disponibles.
2. **Détection des slices qui explosent dans la croissance de région** : Certaines slices présentent une croissance anormale en taille lors du processus de segmentation. Pour y remédier, une amélioration possible serait de détecter ces slices problématiques et de les remplacer par des segmentations issues :
 - De slices voisines sur le plan spatial.
 - Ou de slices temporellement adjacentes qui n'ont pas explosé.
3. **Optimisation des algorithmes** : Une autre piste d'amélioration concerne la rapidité des algorithmes. Actuellement, beaucoup de paramètres sont explorés par **force brute**, ce qui est coûteux en temps de calcul. Une optimisation des algorithmes permettrait d'obtenir des résultats plus rapidement, rendant l'approche plus adaptée aux applications en temps réel ou avec de grands volumes de données.
4. **Utilisation des réseaux de neurones** : La segmentation du ventricule gauche sur des images 3D est une tâche parfaitement adaptée à l'utilisation de *réseaux de neurones convolutionnels (CNN)*. Ces modèles, couramment employés en pratique, pourraient offrir des résultats nettement meilleurs que les approches classiques, en exploitant efficacement les structures spatiales et temporelles des données.

Bibliographie : Lee HY, Codella NC, Cham MD, Weinsaft JW, Wang Y. Automatic left ventricle segmentation using iterative thresholding and an active contour model with adaptation on short-axis cardiac MRI. IEEE Trans Biomed Eng. 2010 Apr ;57(4) :905-13. doi : 10.1109/TBME.2009.2014545. Epub 2009 Feb 6. PMID : 19203875.

8 Annexe

```
def affichage_masque():
    img_m, mask = tf_mask(t_img, heartrate=t_img.shape[0] / (patient.info["ES"]
    ↵ - patient.info["ED"]))

    fig = plt.figure(figsize=(15,5))
    plt.subplot(1,3,1)
    plt.imshow(t_img[Tps, :, :, Slice], cmap='gray')
    plt.title(f'Patient n°{Numero_patient}', fontsize=20)
    plt.xticks([])
    plt.yticks([])
    plt.subplot(1,3,2)
    plt.imshow(mask, cmap='gray')
    plt.title('Masque', fontsize=20)
    plt.xticks([])
    plt.yticks([])
    plt.subplot(1,3,3)
    plt.imshow(img_m, cmap='gray')
    plt.title('Image masquée', fontsize=20)
    plt.xticks([])
    plt.yticks([])
    plt.tight_layout()
    plt.show()

affichage_masque()
Fréquence = t_img.shape[0] / (patient.info["ES"] - patient.info["ED"])
print(f"Fréquence Cardiaque du Patient n°{Numero_patient}: {Fréquence:.3f}")
```

```
def draw_cercles(image4d, circles=None, Tps=Tps, Slice=Slice):
    cimg = np.array(image4d[Tps, :, :, Slice])
    cimg = np.uint8(cimg/np.max(cimg)*255)
    cimg = to_rgb(cimg)
    if circles is not None:
        circles = np.uint16(np.around(circles))
        for circle in circles:
            cv2.circle(cimg, (circle[0], circle[1]), circle[2], (0, 255, 0), 1)
            cv2.circle(cimg, (circle[0], circle[1]), 1, (255, 0, 0), 2)
    return cimg
```

```
def draw_cercle(image4d, circles=None, Tps=Tps, Slice=Slice):
    cimg = np.array(image4d[Tps, :, :, Slice])
    cimg = to_rgb(np.uint8(cimg/np.max(cimg)*255))
    if circles is not None:
        circles = np.uint16(np.around(circles))
        cv2.circle(cimg,(circles[0],circles[1]),circles[2],(0,255,0),1)
        cv2.circle(cimg,(circles[0],circles[1]),1,(255,0,0),2)
    else:
        print("no circles !")
    return cimg
```

```
def hough_appercus(n1,n2):
    circles_affichage = []
    fig, ax = plt.subplots(1, n2-n1, figsize=(20,5))
    for i in range(n2-n1):
        patient = load_patient(n1+i)
        t_img = patient.img.data.clone().detach().float()
        hr = t_img.shape[0] / (patient.info["ES"] - patient.info["ED"])
        img_m = tf_mask(t_img, heartrate=hr, Tps=Tps, Slice=Slice)
        img_m = np.array(img_m[0], dtype=np.uint8)
        circles = best_hough(img_m)
        circles_affichage.append(circles)
        cimg = draw_cercle(t_img, circles_affichage[i])
        ax[i].imshow(cimg)
        ax[i].set_title(f'Patient n°{i+n1}')
        ax[i].axis('off')
    plt.suptitle('Cercles détectés pour 5 patients du dataset avec Masque +\n→ Hough optimisé', fontsize=20)
    plt.tight_layout()
    plt.show()
hough_appercus(23,28)
```

```
thresholds = [30, 120, 150, 200]
def comparaison_threshold(Numero_patient, thresholds):
    patient_1, patient_2 = load_patient(Numero_patient),
    → load_patient(Numero_patient+5)
    img_1, img_2 = patient_1.img.data.clone().detach().float(),
    → patient_2.img.data.clone().detach().float()
    seed_1, seed_2 = find_seed(patient_1), find_seed(patient_2)
    seed_int_1, seed_int_2 = (int(seed_1[0]), int(seed_1[1])), (int(seed_2[0]),
    → int(seed_2[1]))
    fig, ax = plt.subplots(2, 4, figsize=(15,5))
    for t in thresholds:
        lv_seg_1, taille_1 = region_growth(img_1[Tps, :, :, Slice], seed_int_1,
        → error=t)
        lv_seg_2, taille_2 = region_growth(img_2[Tps, :, :, Slice], seed_int_2,
        → error=t)
        contours_1 = measure.find_contours(lv_seg_1, 0.5)
        contours_2 = measure.find_contours(lv_seg_2, 0.5)
        ax[0, thresholds.index(t)].imshow(patient_1.img.data[Tps, :, :, Slice],
        → cmap='gray')
        ax[0, thresholds.index(t)].plot(seed_1[0], seed_1[1], 'ro')
        ax[0, thresholds.index(t)].plot(contours_1[0][:, 1], contours_1[0][:,
        → 0], 'r')
        ax[0, thresholds.index(t)].set_title(f'Threshold={t} et
        → taille={taille_1}')
        ax[0, thresholds.index(t)].axis('off')
        ax[1, thresholds.index(t)].imshow(patient_2.img.data[Tps, :, :, Slice],
        → cmap='gray')
        ax[1, thresholds.index(t)].plot(seed_2[0], seed_2[1], 'ro')
        ax[1, thresholds.index(t)].plot(contours_2[0][:, 1], contours_2[0][:,
        → 0], 'r')
        ax[1, thresholds.index(t)].set_title(f'Threshold={t} et
        → taille={taille_2}')
        ax[1, thresholds.index(t)].axis('off')
    plt.suptitle('Segmentation pour 2 patients du dataset', fontsize=20)
    plt.tight_layout()
    plt.show()

comparaison_threshold(Numero_patient, thresholds)
```

```
def draw_region(imageRGB, lv_seg):
    le_contour = lv_seg - cv2.erode(lv_seg,
    → cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(3,3)), 1,
    → borderType=cv2.BORDER_CONSTANT, borderValue=0)
    lv_seg = np.bool_(lv_seg)
    le_contour = np.bool_(le_contour)
    imageRGB[lv_seg, 2] = 100
    imageRGB[le_contour, 2] = 255
    return imageRGB
```

```
def evaluate_avg_dice2D(nb):
    avg_dist = 0
    miss = 0
    accuracie = 0
    rad = []
    avg_dice = 0
    segs = []
    for i in tqdm(range (1,nb+1)):
        patient = load_patient(i, testing=True)
        t_img = patient.img.data.clone().detach().float()
        circles, slice = find_seed_multiple_slices(patient)
        lv_seg = leftv_seg(t_img[patient.info["ED"], :, :, slice], circles,
                           Slice=slice)
        if lv_seg is None:
            miss += 1
            continue
        c_img = draw_cercle(t_img, circles, Slice=slice)
        c_img = draw_region(c_img, lv_seg)
        found, dist, hit = patient_metrics(patient, circles, Slice=slice)

        miss += not found
        avg_dist += dist
        accuracie += int(hit)

        cv2.imwrite(f'training_segmentation/cercles_{i:03d}.png', c_img)
        dice_score = dice(patient, lv_seg, slice)
        avg_dice += dice_score
        segs += [(dice_score, i, slice, c_img)]
    avg_dist /= nb - miss
    avg_dice /= nb
    print(f"\n-- results --\n accuracie: {round(accuracie / nb * 100, 1)}%\n
          (over found circles: {round(accuracie / (nb-miss) * 100, 1)})\n miss:
          {miss}\n avg dist: {avg_dist}\n avg dice:
          {avg_dice}\n-----")
    return segs

seg = evaluate_avg_dice2D(50)
```

```
def show_best_and_worst_segmentation(seg):
    segs_s = sorted(seg)
    f, axs = plt.subplots(2, 5, figsize=(15, 5))
    for i in range(5):
        (dice_score, nb_patient, slice, segm) = segs_s[-i-1]
        patient = load_patient(nb_patient)
        img = patient.gt.data.clone().detach().float()[0, :, :, slice]
        axs[0, i].imshow(segm)
        axs[0, i].axis('off')
        axs[1, i].imshow(img, cmap="Blues")
        axs[1, i].axis('off')
    plt.suptitle("top 5 best segmentations")
    plt.tight_layout()
    plt.show()
    f, axs = plt.subplots(2, 5, figsize=(15, 5))
    for i in range(5):
        (dice_score, nb_patient, slice, segm) = segs_s[i]
        patient = load_patient(nb_patient)
        img = patient.gt.data.clone().detach().float()[0, :, :, slice]
        axs[0, i].imshow(segm)
        axs[0, i].axis('off')
        axs[1, i].imshow(img, cmap="Blues")
        axs[1, i].axis('off')
    plt.suptitle("top 5 worst segmentations")
    plt.tight_layout()
    plt.show()

show_best_and_worst_segmentation(seg)
```

```
def slice3D_t(patient):
    ED = patient.info["ED"] - 1 # get the index of the slice of the ground
    ↪ truth.
    seed0, sslice = find_seed_multiple_slices(patient, ED)

    # print(seed0)
    img3D_t = patient.img.data.clone().detach().float()
    lv3D_t = np.zeros_like(img3D_t)

    if seed0 is None:
        return lv3D_t
    # initial segmentation
    lv3D_t[ED, :, :, sslice] = leftv_seg(img3D_t[ED, :, :, sslice], seed0,
    ↪ Tps=ED, Slice=sslice)
    expend3D(img3D_t[ED], lv3D_t[ED], sslice)
    # seed propagation
    for t in tqdm(range(ED+1, img3D_t.shape[0])):
        lv_1 = lv3D_t[t-1, :, :, sslice]
        idxs = np.where(lv_1 == 1)
        if len(idxs[0]) == 0:
            break
        center = np.mean(idxs, axis=1)
        radius = np.sqrt(len(idxs[0])/np.pi) # find radius from volume
        lv3D_t[t, :, :, sslice] = leftv_seg(img3D_t[t, :, :, sslice],
        ↪ [center[1], center[0], radius], Tps=t, Slice=sslice)
        expend3D(img3D_t[t], lv3D_t[t], sslice)

    for t in tqdm(range(ED-1, -1, -1)):
        lv_1 = lv3D_t[t+1, :, :, sslice]
        idxs = np.where(lv_1 == 1)
        if len(idxs[0]) == 0:
            break
        center = np.mean(idxs, axis=1)
        radius = np.sqrt(len(idxs[0])/np.pi) # find radius from volume
        lv3D_t[t, :, :, sslice] = leftv_seg(img3D_t[t, :, :, sslice],
        ↪ [center[1], center[0], radius], Tps=t, Slice=sslice)
        expend3D(img3D_t[t], lv3D_t[t], sslice)
    return lv3D_t
```