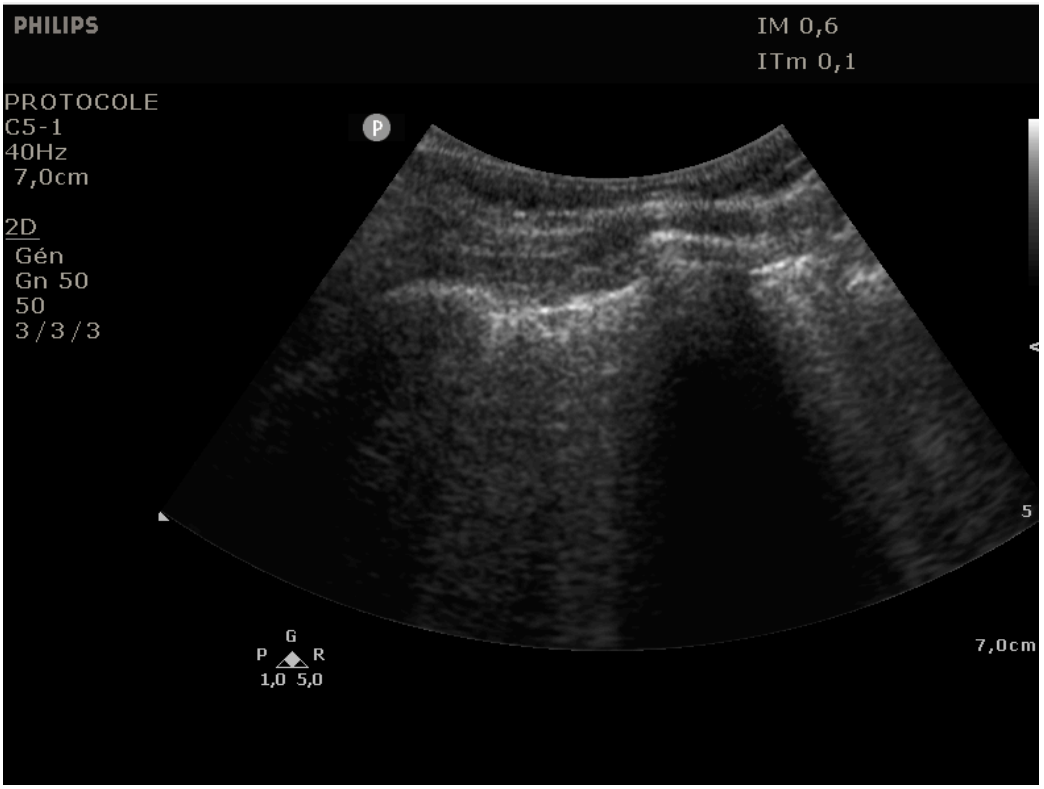
Projet génie logiciel

Analyse d’échographies de poumons



Antoine Delenclos
Esteban Fernandes

Rendu à M.Makris
Le 19/01/2024

Delenclos Antoine
Fernandes Esteban

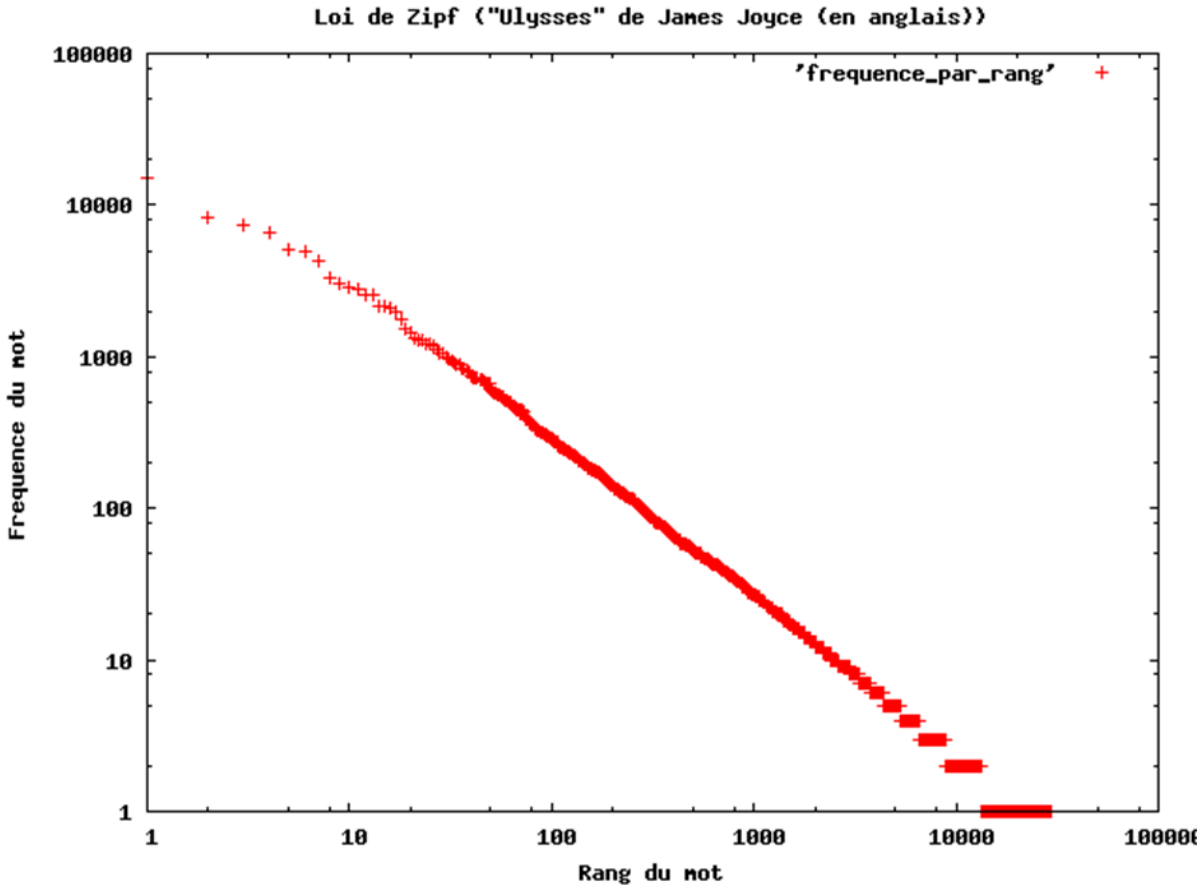
Tuteur : Pascal Makris

Sommaire

Remise en contexte et cahier des charges

Remise en contexte:

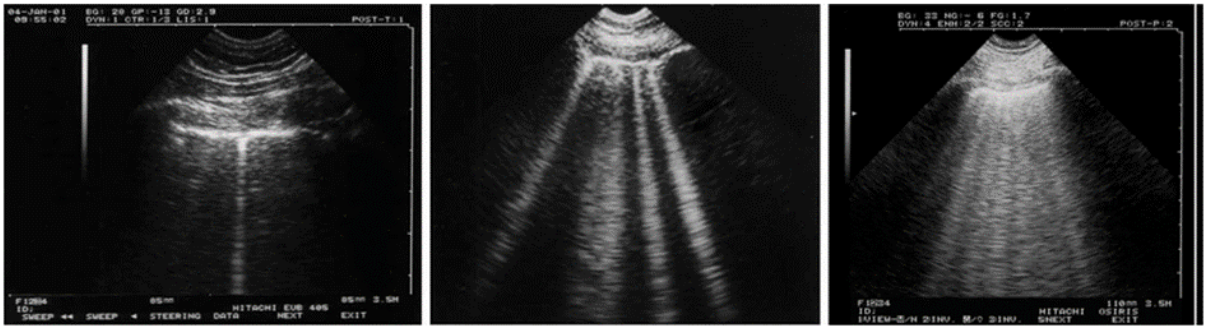
Ce projet a été réalisé d'octobre à janvier avec pour objectif la mise en pratique des bonnes méthodes de génie logiciel.
Le tuteur de ce projet est Pascal Makris et le sujet était l'application de la loi de Zipf aux images échographiques de poumons.
D'abord il nous semble important de préciser que l'année dernière un projet similaire a été réalisé mais en C++, après analyse nous n'avons pas pu reprendre grand chose de ce qui avait été réalisé. Nous sommes donc repartis de zéro. De plus un projet a été lancé par des 5èmes années sur l'analyse de même image par des 5èmes années, eux travaillait en revanche avec de l'IA.
Nous allons étayer un petit peu le sujet du projet car de prime abord c'est un sujet assez vaste.
Premièrement la loi de Zipf,
La loi de Zipf est une loi formulée en 1935 par George Kingsley Zipf.
Le principe est assez simple, dans un livre le mot qui apparaît le plus de fois apparaît deux fois plus que le deuxième mot qui apparaît le plus.
Ce principe fut appliqué à l'époque sur le livre Ulysse de James Joyce.
Pour faire transparaître cette loi d'un point de vue mathématique, on prend un diagramme log/log dans lequel on trie les mots par ordre d'apparition. On a en abscisse le classement et en ordonnée la fréquence d'apparition.



Maintenant qu'on a construit notre graphe qu'est-ce qu'il faut y voir ?
Cette loi permet d'observer une linéarité, ici on voit bien qu'on a une quasi-droite. Cela veut dire que les mots dans le livre sont répartis équitablement.
A ce stade-là, il n'y a pas de rapport avec les échographies de poumons ainsi que les images. Il va falloir faire la correspondance entre un livre et des mots à une image et quelques choses pour représenter des mots.

On pourrait se demander l'intérêt d'appliquer la loi de Zipf sur des échographies de poumons ?
D'abord il faut dire que les échographies sont moins utilisées que les tomographies pour les poumons. Pourtant une échographie est moins lourde car cela ne fonctionne pas avec des radiations et on peut déduire des choses d'une échographie de poumon.
La chose que nous voulons exploiter via la loi de Zipf s'appelle les lignes B, ce sont des lignes assez visuelles qui se développent sous plusieurs formes.

De gauche à droite: sujet sain, une seule ligne B, au centre des lignes B7 et à droite des lignes B3.



Les lignes B traduisent différents problèmes liés aux poumons d'un patient, et cela se remarque d'un point de vue graphique, donc si cela se remarque d'un point de vue graphique la courbe de Zipf sera différente d'un poumon normal.

Voici donc l'objectif final du projet :

Pouvoir analyser des images d'échographies de poumon et tester si on peut en tirer quelque chose grâce à Zipf. C'est un projet expérimental où ne pas avoir de résultat représenterait déjà un résultat.

Cahier des charges :

Le logiciel devait donc comprendre :

-Une interface graphique

-Pouvoir ouvrir des images d'échographie

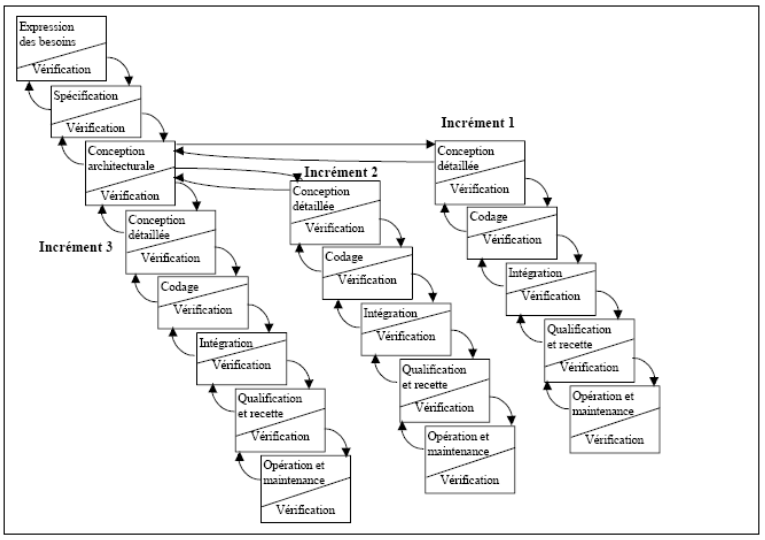
-Pouvoir choisir le motif que l'on applique

-Appliquer la loi de Zipf a des images

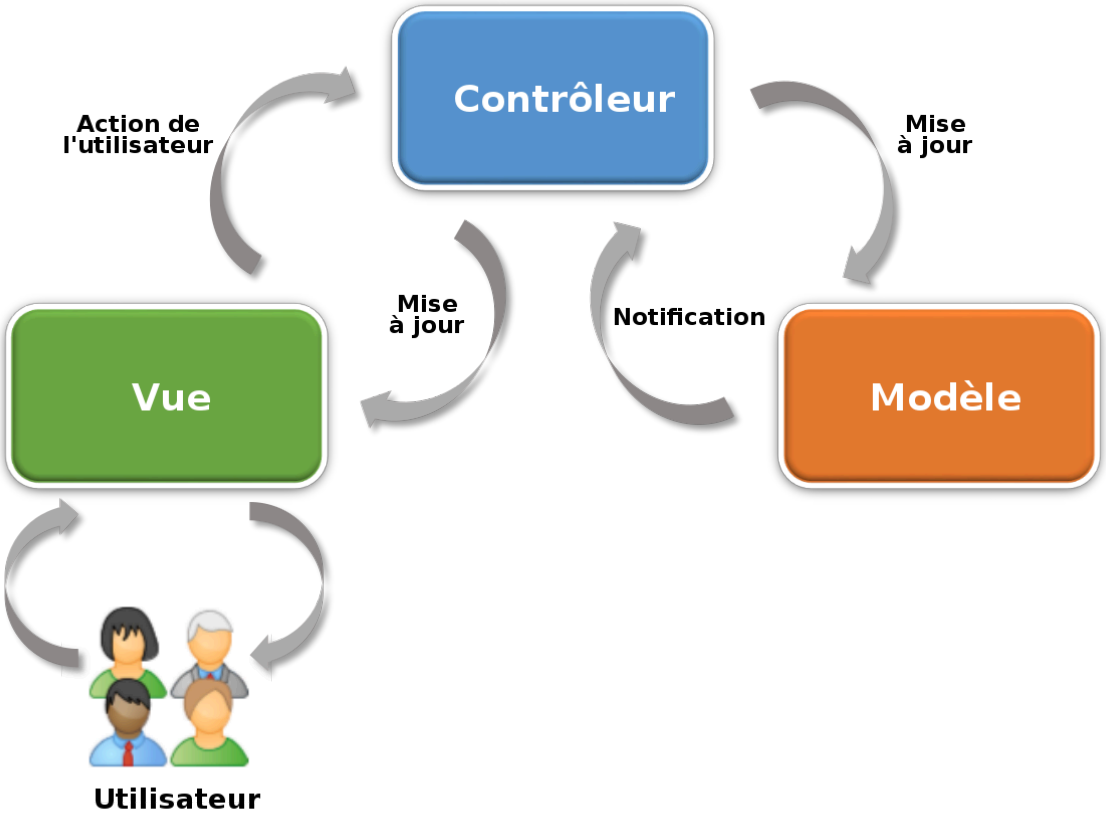
Le cahier des charges a été vu et ajusté avec notre tuteur tout au long du projet, nous avons fait beaucoup de réunions et avons revu à chaque fois les tenants et aboutissants du projet.

Bonne pratique de génie logiciel

L'un des principaux objectifs de ce projet était de mettre en œuvre des pratiques de génie logiciel.
Cela passe par plusieurs choses que nous avons vues et que nous avons mises en place.
Le but du génie logiciel est d'assurer le bon déroulement du projet.
Pour cela on commence souvent par parler de cycle de développement.
Un cycle de développement définit la manière dont on approche un projet, quelles étapes on réalise, dans quel ordre, avec qui. Il est important de bien choisir son cycle de développement car celui-ci constitue le squelette du projet, il doit au mieux refléter les conditions disponibles pour un projet.
Dans notre cas, nous avons utilisé un cycle incrémental qui fait partie des méthodes agiles car c'est ce qui correspondait le plus à notre situation.



Notre tuteur étant assez présent tout au long du projet c'était plus simple pour nous d'avancer petit à petit sur le logiciel et de montrer nos avancés à notre tuteur.
Cela nous a permis de réévaluer en permanence le projet.
Nous avons d'abord établi une conception détaillée, pour l'architecture même du logiciel ainsi que pour l'interface homme machine à l'aide de mockup.
Nous avons par la suite lancé les incréments un par un, nous travaillons toujours sur des parties différentes du projet afin de ne pas nous gêner.
tantôt l'un travaillait sur l'interface, et l'autre travaillait sur les algorithmes.
L'un des éléments qui a également permis cette fluidité de travail sur le projet a été la mise en place d'un design pattern, le MVC (modèle vue contrôleur).
C'est une architecture logicielle qui permet de dissocier d'un côté le traitement, de l'autre l'affichage et enfin le lien entre les deux



Les design patterns sont quelques choses que nous avons vu ce semestre et qu'il nous semblait intéressant d'ajouter pour donner du sens au projet.
Ce n'est pas le seul design pattern que nous avons utilisé, en effet pour gérer les données tout au long de l'ouverture du logiciel nous avons utilisé un design pattern nommé le singleton.
Je détaillerais cela dans la partie dédiée au logiciel.

Nous avons également mis en place un versioning grâce à Github, cela nous a permis de travailler conjointement sur le logiciel sans se gêner.
Une Javadoc a également été générée pour assurer la reprise et la maintenabilité du projet.
Pour générer la Javadoc nous avons utilisé le plugin Eclipse JAutodoc pour générer automatiquement un template de Javadoc sur le projet.

Point clés du projet

Le projet était donc clair, réaliser un logiciel avec interface graphique pour traiter des images.

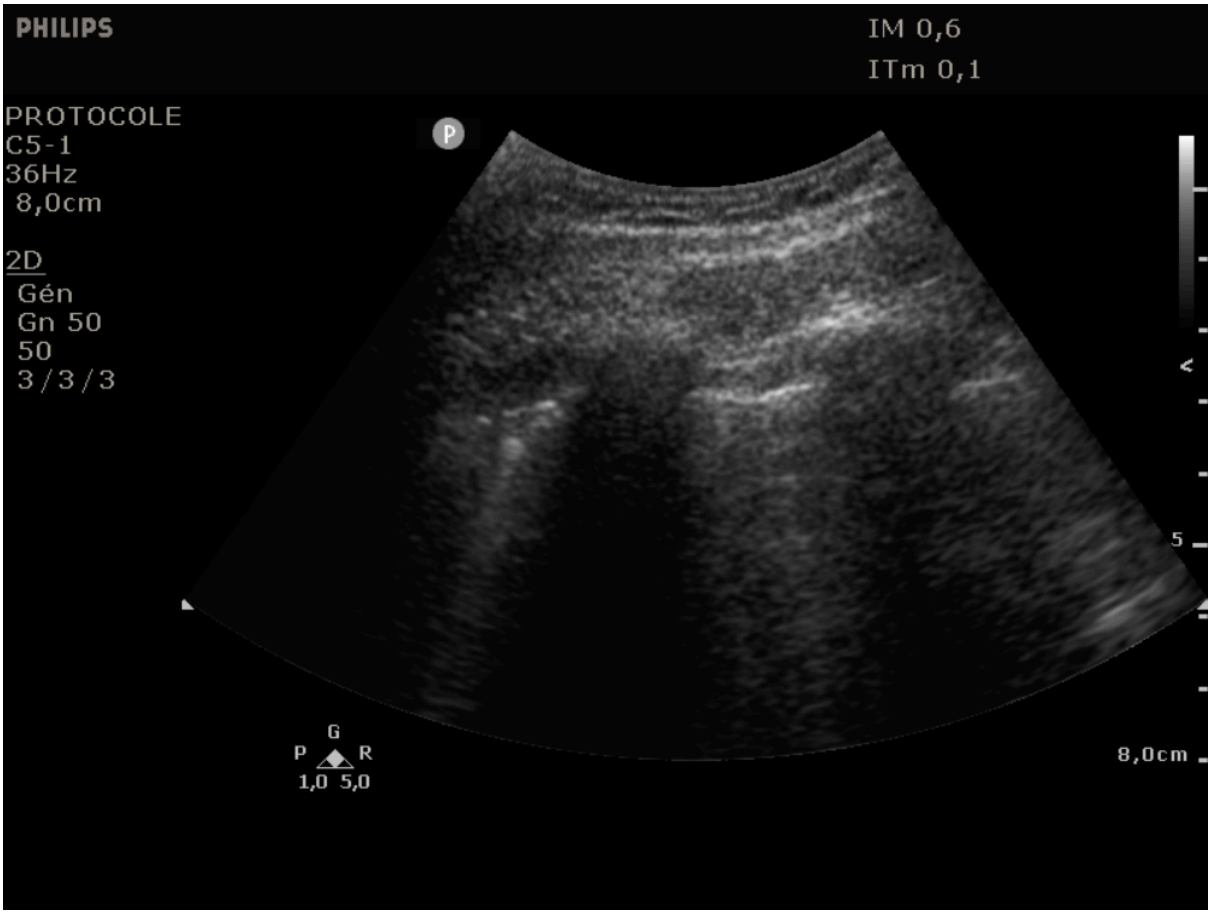
Le choix du langage s'est fait de manière assez naturelle. Il fallait pouvoir faire une IHM de manière concise et avoir si possible des librairies déjà existantes pour traiter certaines parties possiblement compliquées. De tous les langages que nous connaissions, le java était le plus adapté. Le fait de ne pas avoir à s'occuper de la mémoire et d'avoir une librairie standard si grande. Cela nous a été extrêmement utile, de plus nous avons également utilisé Maven pour simplifier l'installation de différentes librairies. Maven est une dépendance qui s'occupe de gérer les librairies externes que nous voulons ajouter, cela se fait via un fichier xml où il suffit de renseigner quelques informations, maven va alors trouver et charger les bonnes classes.

Cela nous a permis d'importer de quoi faire notre IHM (JavaFX ainsi que d'autres librairies utiles). Maven est fastidieux à installer mais une fois mis en place est un véritable plus pour notre programme.

Nous allons maintenant détailler les points importants du projet.

Conversion DICOM

Les fichiers que nous devions traiter sont des échographies. Ces images sont au format DICOM, format utilisé dans l'imagerie médicale. Une des particularités de ce format d'image est le multiframe. Ces fichiers binaires contiennent les informations de plusieurs images en eux.



On peut lire la fréquence de notre fichier DICOM. Celle-ci indique le nombre de frames par seconde, dans notre cas 36 frames de ce fichier DICOM représenteront 1 seconde d'enregistrement par la sonde.
La sonde étant fixe lors de l'enregistrement, les variations entre les frames sont liées à la respiration du patient.

Nous voulions donc réussir à traiter les informations présentes dans ces fichiers binaires et pour cela nous nous sommes tournés vers la bibliothèque open source DCM4CHE. Après l'avoir importé via Maven (en l'ajoutant dans le pom.xml), nous avons fait une classe DicomLoader qui nous permet de créer une matrice d'entiers (allant à de 0 à 255 car nous travaillions sur des images avec canaux 8 bits)

représentant l'intensité de chaque pixel de la frame choisie pour un fichier DICOM, notamment avec la méthode chargeImageDicomBufferise ci-dessous.

```
public BufferedImage chargeImageDicomBufferise(int value) throws IOException
{ //Value = frame du fichier dicom
	Iterator<ImageReader> iter =
ImageIO.getImageReadersByFormatName("DICOM");//spécifie l'image
	ImageReader readers = iter.next();//on se déplace dans l'image dicom
	DicomImageReadParam param1 = (DicomImageReadParam)
readers.getDefaultReadParam();//return DicomImageReadParam
// Adjust the values of Rows and Columns in it and add a Pixel Data attribute
with the bytearray from the DataBuffer of the scaled Raster
	ImageInputStream iis = ImageIO.createImageInputStream(dicomFile);
	readers.setInput(iis, false);//sets the input source to use the given
ImageInputSteam or otherObject
	BufferedImage image = readers.read(value,param1);//BufferedImage image
=reader.read(frameNumber, param); frameNumber = int qui est l'imageIndex
	readers.dispose();//Releases all of the native sreen resources used by
this Window, itssubcomponents, and all of its owned children
	return image;
}
```

Pour afficher ses images dans le logiciel nous avons ensuite une méthode qui permettait de convertir la BufferedImage retourné par la méthode ci-dessus, et qui crée une image au format PNG ou JPEG (au choix). Ces images ne sont utilisées qu'à des fins d'affichage, le travail sur les informations se fait sur des matrices.

Détection de la zone de l'échographie

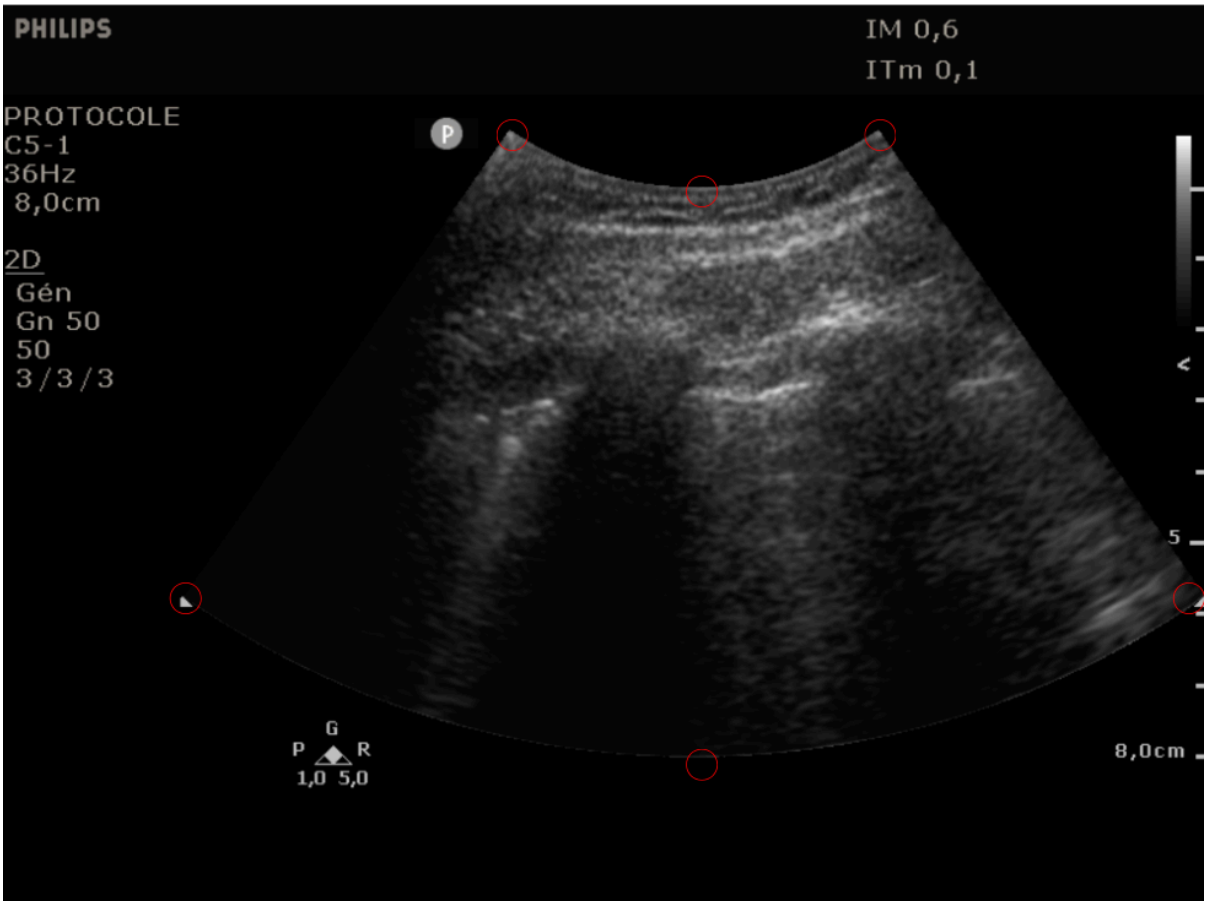
Une des parties qui nous a pris une grande partie de notre temps, tant dans la réflexion que dans la mise en place puis l'optimisation, a été de détecter parfaitement la zone de l'échographie.
Tout d'abord il a fallu convertir la bufferImage en matrice de pixels de nuances de gris. Pour cela nous avons utilisé le code ci-dessous.

```
int col = bufferImg.getRGB(j, i);
int red = col & 0xff0000 >> 16;  //Utilise un masque pour lire le canal rouge et on va ensuite décaler de 16 bits vers la droite pour ensuite lire le vert
int green = col & 0xff00 >> 8;
int blue = col & 0xff;
// Utilisation de la norme rec 709
greyPixelsLevels[i][j] = (int) (0.2126 * red + 0.7152 * green + 0.0722 * blue);
```

Il y a 2 choses importantes à noter dans ce morceau de code. La première est la manipulation de l'entier retourner par la méthode getRGB d'une BufferedImage, manipulation qui consiste à appliquer un masque à l'entier pour n'y lire que le canal qui nous intéresse. Une fois les trois canaux de couleurs séparés, on va calculer l'intensité avec la norme Rec. 709 qui assure une meilleure retranscription des contrastes.

En effet, maintenant que nous avons la matrice des pixels d'une des frames, nous devons réussir à déterminer la zone d'information utile. Nous avons donc pris plusieurs semaines avant de trouver un moyen d'avoir tous les points (cf. figure

ci-dessous) utiles à la reconnaissance de la zone.



Après avoir observé différentes images, nous avons trouvé un schéma de détection. En effet, les images d'échographies possèdent toujours 2 triangles blancs dans les coins inférieurs de la zone échographique et de plus le triangle droit est dans la même colonne que la barre de graduation. Nous avons donc cherché un moyen de détecter ce triangle, et la méthode que nous avons utilisée est celle du parcours de la dernière colonne de l'image. En effet seul le triangle droit possède un pixel blanc sur la dernière colonne de l'image.

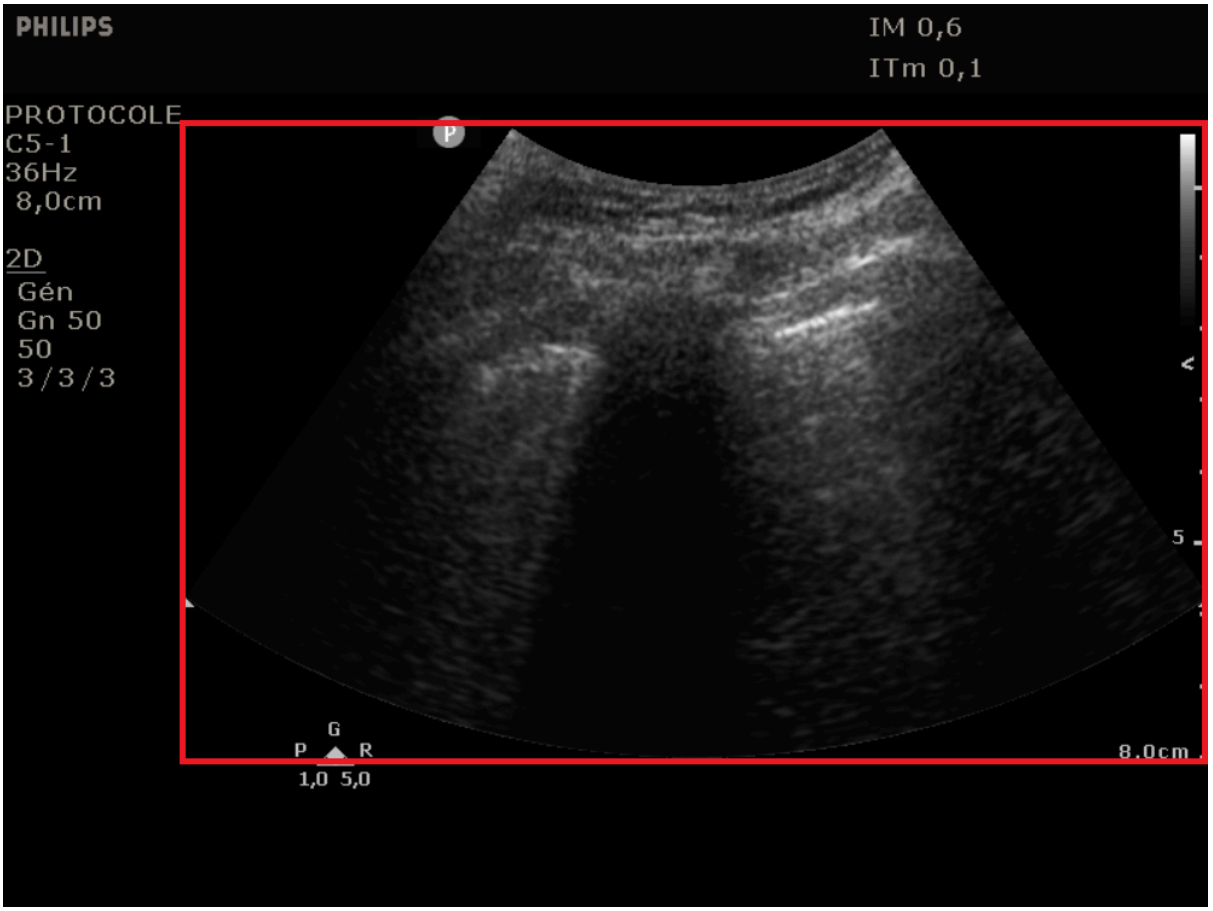
Une fois qu'on a détecté point on va partir de la gauche de l'image mais à la hauteur du triangle de droite pour détecter le triangle de gauche. Dès qu'un pixel sur cette ligne est blanc alors c'est le triangle de gauche. Cependant dans notre code nous faisons bien attention à bien détecter le triangle de droite dans tous les cas, car lorsque celui-ci est superposé avec la barre de graduation cela décale le triangle de gauche. La détection du triangle de gauche se fait donc de bas en haut pour ces raisons.

Maintenant qu'on a les 2 points aux extrémités gauche et droite de l'échographie, on peut déterminer le point le plus bas de l'échographie qui est dans la colonne entre les 2 triangles et sur la dernière ligne avec un pixel non noir.

Pareil pour le point le plus bas de la courbe haute.

On peut déduire les sommets gauche et droite de l'échographie en se mettant dans les colonnes entre l'extrémité correspondante et le milieu de l'échographie.

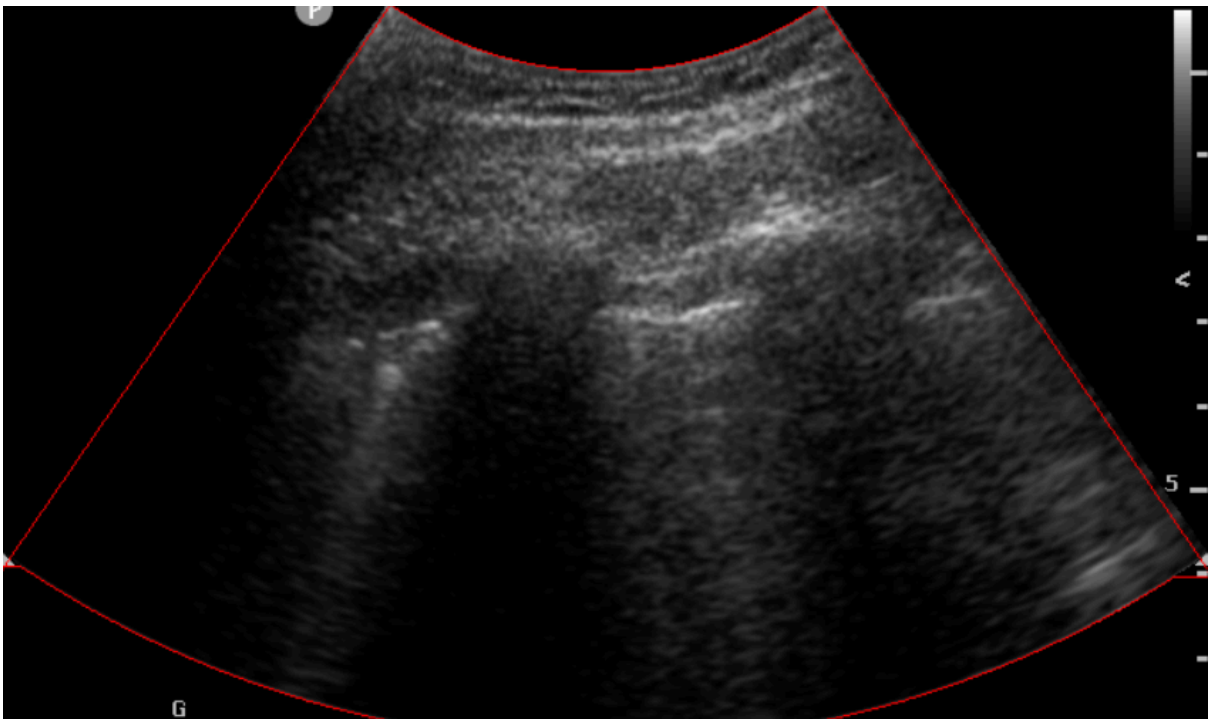
Maintenant on se retrouve avec tous ces points, ce qui nous permet d'obtenir une zone croppée de l'image de originelle (cf. image ci-dessous)



Il nous reste alors encore à déterminer les contours de la véritable échographie.
Pour les pentes gauches et droites, connaissant les points aux extrémités de chacune des pentes, il faut calculer la dérivée de la droite associée puis parcourir via une liste de flottants (car le coefficient directeur n'est pas forcément un entier) et ajouter les points de cette pente. Il faut comprendre dans ces points, les points intermédiaires qui auraient pu être sautés. Par exemple, si on a un coefficient directeur de 1.5, alors on va louper 1 point tous les 3 points sur la pente.

Pour les courbes cela est plus complexe. On doit séparer la courbe haute en 2 parties et idem pour la courbe gauche. On va regarder si le pixel au-dessus du point actuel est noir et si c'est le cas on va dire qu'on est sur un point de courbe et regarder le prochain point (+1 en abscisse). Idem pour la partie de droite.
Pour la courbe basse on fait la même méthode mais en observant si c'est le point en dessous de celui que nous observons, qui est noir.

A la fin on obtient la zone à l'intérieur des contours rouges (cf. figure ci-dessous), qui contient l'ensemble des points de l'échographie, points qui seront traités par la loi de Zipf.



Thread

Le logiciel traite des images, c'est quelque chose de chronophage car on va devoir itérer sur toute celle-ci.
Pour pallier cela on peut utiliser des thread,
Un thread est un fil d'exécution, un programme de base ne contient souvent pas plus d'un thread pour exécuter le code, on parle alors d'exécution monothread.
A contrario, le multi-threading permet dans certains cas de gagner du temps, lorsqu'on parle de traitement d'image c'est l'un de ces cas.
On va donc utiliser plusieurs thread sur notre programme pour gagner du temps, les thread sont particulièrement adaptés au traitement d'image car le traitement est souvent assez léger, donc le threader permet de faire plus vite plein de petits traitements.
Pour générer la matrice des niveaux de gris par exemple, voici le code exécuté, présenté plus haut.

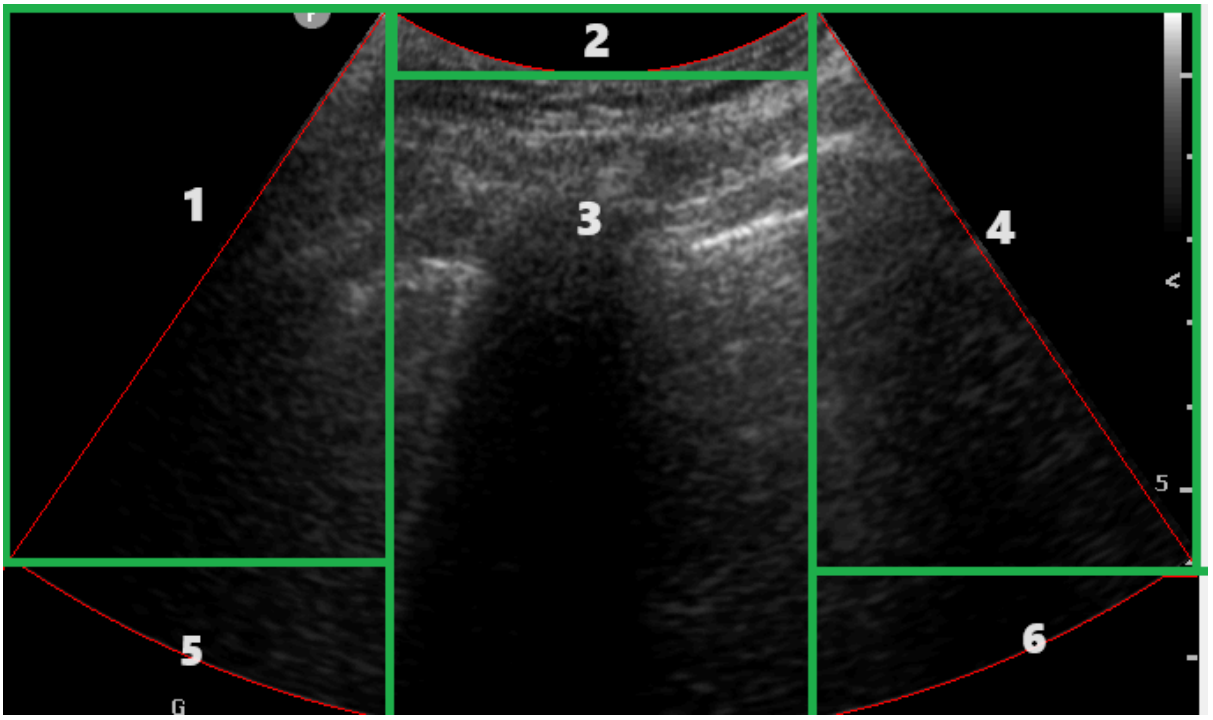
```
for (int i = height_min; i < height_max; i++) {
      for (int j = 0; j < widthImg; j++) {
            int col = bufferImg.getRGB(j, i);
            int red = col & 0xff0000 >> 16;
            int green = col & 0xff00 >> 8;
            int blue = col & 0xff;
                        //Utilisation de la norme rec 709 pour une meilleure
représentation des nuances de gris
            returnpixellevel[i][j] = (int)(0.2126 * red + 0.7152 * green +
0.0722 * blue);
      }
}
```

Ce programme est lancé 15 fois en parallèle, et les seules choses qui changent sont les valeurs de height_min et height_max.
Ces valeurs sont définies selon le numéro du thread et avec cela on itère sur toute l'image plus rapidement.
On utilise les thread également pour définir une matrice de booléen qui nous sert à savoir si on est dans la zone de l'échographie.
C'est plus intéressant ici car on va threader cette partie avec un nombre précis, et selon le nombre ce n'est pas le même code qui est exécuté :

```
switch (nThread) {
                                case 0: // Partie supérieur gauche : pente gauche
                                        // Pour chaque point : chercher si le point de la
pente est avant ou après (sur
                                        // une même ligne)
                                        if (tempIndex != -1) {
                                                if (x > penteGauche.Points.get(tempIndex).x)
                                                        booleanZipf[i][j] = true;
                                        }
                                        break;
                                case 1: // Partie supérieur au dessus de la courbe haute

                                        tempIndex = pointsCourbeHaute.lookFornearX(j, 2);
                                        if (tempIndex != -1) {
                                                if                      (y                      >
pointsCourbeHaute.Points.get(tempIndex).y)
                                                        booleanZipf[i][j] = true;
                                        }
                                        break;
                                case 2: // Partie centrale
                                        booleanZipf[i][j] = true;
                                        break;
                                case 3: // Partie supérieur droite : pente droite
                                        if (tempIndex != -1) {
                                                if                      (x                      <
pointsPenteDroite.Points.get(tempIndex).x) {
                                                        booleanZipf[i][j] = true;
                                                }
                                        }
                                        break;
                                case 4: // Partie inférieure gauche : courbe basse gauche
                                        if(tempIndex!=-1)
                                        {
                                                if                      (x                      >
pointsCourbeBasseGauche.Points.get(tempIndex).x) {
                                                        booleanZipf[i][j] = true;
                                                }
                                        }
                                        break;
                                case 5: // Partie inférieure droite : courbe basse droite
                                        if(tempIndex!=-1)
                                        {
                                                if                      (x                      <
pointsCourbeBasseDroite.Points.get(tempIndex).x) {
                                                        booleanZipf[i][j] = true;
                                                }
                                        }
                                        break;
                                }
```

On vient utiliser un switch et selon on regarde différemment, voici le schéma qui indique dans quel ordre on fait quelle zone de l'image.



Nous voulions également threader la partie du traitement de la loi de Zipf, néanmoins après plusieurs tests nous avons perdu des informations, nous avons préféré avoir une version plus lente mais fonctionnelle.

Appliquer la loi de Zipf

Généralités

Pour appliquer la loi de Zipf, il va falloir passer par plusieurs étapes, en sortant d'une image dicom on récupère la matrice de booléen, puis ce traitement vient réaliser les différents motifs, c'est la première chose concernant Zipf.

```
for (int i = 0; i < max_row_iteration; i++) {
      for (int j = 0; j < max_col_iteration; j++) { // Pour chaque pixel on
va maintenant regarder son
            int[] listMotif = new int[motifSizeX * motifSizeY];
            int count = 0;
            boolean check = true;
            int newI = i*motifSizeX;
            int newJ = j*motifSizeY;
            for (int ki = newI ; ki < newI + motifSizeX; ki++) {
                  for (int kj = newJ; kj < newJ + motifSizeY; kj++) {
                        int temp = greyMatrix[ki][kj];
                        if (ZipfOrNot[ki][kj] == false) {
                              check = false;
                              break;
                        }
                        listMotif[count] = temp;
                        count++;
```

```
                        if(max_y<kj)
                              max_y=kj;
                  }
                  if (check == false)
                        break;
                  if(max_x<ki)
                        max_x=ki;
            }
            if (check != false) {
                  ArrayList<Integer> codedMotif = codeMotif(listMotif);
                  String strCodedMotif = codedMotifToString(codedMotif);
                  if  (mapMotifNombreOccurence.containsKey(strCodedMotif))  {
// Si le motif est déjà présent dans notre image
                  int                        old_value                        =
mapMotifNombreOccurence.get(strCodedMotif); // Alors on augmente son nombre
d'occurence de 1

mapMotifNombreOccurence.replace(strCodedMotif, old_value + 1);
            } else {
mapMotifNombreOccurence.put(strCodedMotif, 1);
            }
      }
}
```

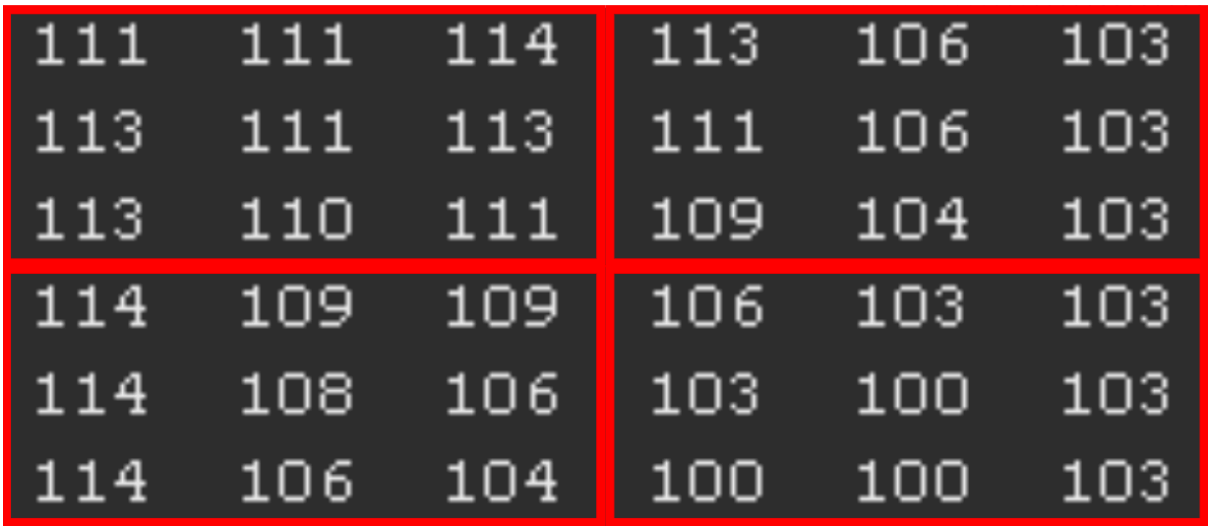
Ce n'est que la première partie du traitement de Zipf, car comme précisé plus tôt la loi de zipf consiste en deux étapes. Récolter les infos sur les récurrences des mots, puis les trier et les afficher dans un graphe log/log.

Motifs

La loi de Zipf était principalement utilisée dans l'analyse de fréquences d'apparitions de mots. Toutefois, nous devons ici l'appliquer à des images.
Il faut donc un équivalent du mot dans un texte, pour les images. Nous appellerons cela le motif.
Un motif va être une matrice, carrée, ligne ou colonne, qui va représenter les intensités d'une zone de l'image.

111	111	114	113	106	103
113	111	113	111	106	103
113	110	111	109	104	103
114	109	109	106	103	103
114	108	106	103	100	103
114	106	104	100	100	103

Dans la figure ci-dessus, les intensités d'un endroit de la zone échographique. Nous avons majoritairement appliqué des motifs de taille 3x3 lors de nos tests, hors test de fonctionnalité. On obtient donc la figure ci-dessous où chaque motif est encadré.



Ici on a 4 motifs différents les uns des autres.
Ici on a un recouvrement nul, cela signifie que les motifs n'ont aucuns pixels en commun entre eux.
Toutefois ces motifs tel quel ne sont pas vraiment exploitables pour une observation avec la loi de Zipf. En effet, le nombre d'occurrence de chaque motif de notre image est le facteur déterminant pour faire des observations avec la loi de Zipf. Or, si nous laissons les motifs comme ci-dessus, nous avons un nombre total de motifs possibles, dans le cadre d'un motif de taille 9, donc pour un motif 3x3 par exemple, de 256^{9} motifs = 4 722 366 482 869 645 213 696 motifs.
Il faut donc une méthode pour réduire ce nombre de possibilités dans les motifs. C'est alors qu'intervient le codage du motif.
Pour chaque motif, on va regarder l'ensemble des intensités du motif et les coder de manière à avoir uniquement des valeurs allant de 0 à n-1, où n = nombre de pixels par motif.
Le codage va fonctionner de la manière suivante. Les pixels qui possèdent la valeur d'intensité la plus faible seront codés avec la valeur 0, puis les pixels avec la deuxième valeur d'intensité la plus faible prendront la valeur 1, etc.
Ainsi on obtient le codage suivant :

114	109	109
114	108	106
114	106	104

⇒

4	3	3
4	2	1
4	1	0

On passe alors à 9^{9} motifs = 387 420 489 motifs.

En dehors du fait qu'on a drastiquement plus de chance de trouver le même motif dans l'image une fois celui-ci codé, cela permet également d'avoir une meilleure détection. En effet ici si une ligne devient graduellement de moins en moins intense mais conserve une proportionnalité, par exemple tous les 10 pixels sur la colonne, l'intensité baisse de 2, on trouvera le même motif codé.
Voici la méthode qui va permettre de coder un motif :

```
public ArrayList<Integer> codeMotif(int[] motif) {
	int len = motif.length;
	ArrayList<Integer> Stock = new ArrayList<>(); // On stocke toutes les valeurs du motif
	for (int i = 0; i < len; i++) {
		ArrayList<Integer> Seuils = new ArrayList<>(); // On va stocker l'ensemble [motif-seuil; motif+seuil]
		int ensemble_seuil_bas = motif[i] - seuilPixelDifferenceDetection; // On vérifie c'est ensemble soit bien
// dans [0;255]
		if (ensemble_seuil_bas < 0) {
			ensemble_seuil_bas = 0;
		}
		int ensemble_seuil_haut = motif[i] + seuilPixelDifferenceDetection;
		if (ensemble_seuil_haut > 255) {
			ensemble_seuil_haut = 255;
		}
		for (int s = ensemble_seuil_bas; s <= ensemble_seuil_haut; s++) {
			Seuils.add(s);
		}
		if (Collections.disjoint(Stock, Seuils)) { // True si rien en commun
			Stock.add(motif[i]);
		}
	}
	Collections.sort(Stock); // Les stocks sont maintenant triés
	ArrayList<Integer> Coded = new ArrayList<>();
	for (int i = 0; i < len; i++) {
		int index;
		if (Stock.indexOf(motif[i]) != -1) { // Si la valeur du pixel existe déjà alors on l'ajoute
			index = Stock.indexOf(motif[i]);
		} else { // Sinon on va prendre l'index de la valeur qui est la plus proche de lui
			int seeked = motif[i];
			index = Stock.indexOf(Stock.stream().min(Comparator.comparingInt(k -> Math.abs(k - seeked))).orElseThrow(() -> new NoSuchElementException("Pas de valeur dans le motif")));
		}
```

```
            Coded.add(index); // On ajoute dans le motif coded l'indice du
rang de motif plus ou moins (qui est dans stock)
      }
      if (!specificOrientation) { // Soit on veut avoir l'orientation des
motifs soit on s'en moque
            Collections.sort(Coded);
      }
      return Coded;
}
```

Nous avons également ajouté une fonctionnalité qui n'était pas demandée mais qui nous semblait intéressante : les seuils.
L'utilisateur peut spécifier un seuil utilisé pour le codage des motifs. Ce seuil élargit l'ensemble des valeurs qui peuvent être assimilées à un même code.
Prenons l'exemple précédent.
Avec un seuil de 2, le pixel 104 → 0, mais les pixels 106 → 0.
Cela peut être utile dans des images très bruitées. Toutefois, mettre un seuil non nul va entraîner une forte perte d'information. Nous n'avons donc pas réellement utilisé cette méthode.

Interface Homme-Machine

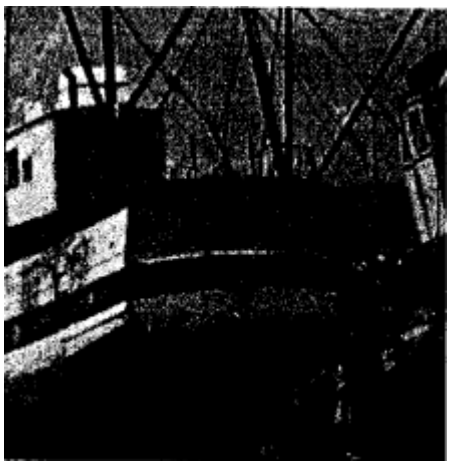
Aussi important que la partie traitement, l'IHM a été une partie essentielle du projet.
Nous avons choisi Java en partie car nous connaissions une librairie pour faire de manière assez concise les interfaces.
Nous avons donc choisi Java avec la librairie JavaFX associée au logiciel SceneBuilder.
JavaFX sert à créer des interfaces graphiques sous Java et SceneBuilder permet de faire du drag and drop pour poser des éléments comme des boutons, barre de menu et autre. L'alliance des deux permet de générer directement des fichiers controller et view. Cela nous permet de générer un modèle MVC

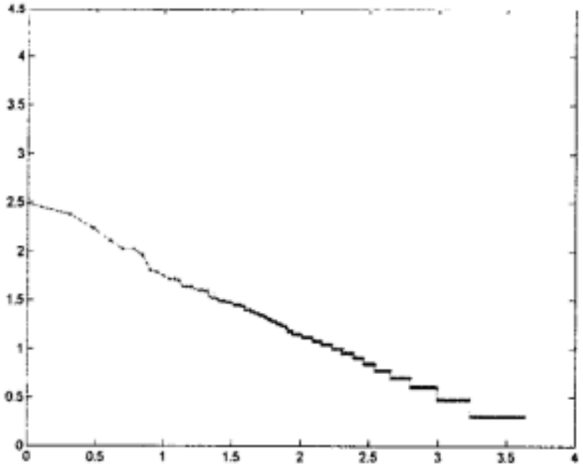
Nous avons au final 3 interfaces différentes: une lorsqu'on lance le logiciel, une lorsqu'une image est sélectionnée, et une dernière avec les paramètres de la loi de Zipf.
Se référer au guide d'utilisation pour voir comment utiliser le logiciel et à quoi il ressemble.

Une difficultée qui s'est posé sur cette partie était lors de l'affichage du diagramme pour la courbe de Zipf.
La particularité de ce diagramme est qu'il s'agit d'un diagramme log/log en base 10, hors avec JavaFX il n'y a pas d'affichage logarithmique de prévu. L'avantage est quand même que grâce à l'orienté objet et au fonctionnement de Java n'importe qui peut modifier les classes de bases des librairies de JavaFX afin d'avoir le comportement voulu. Cela nous a permis de récupérer le travail de Kevin Senachel qui a réalisé des axes logarithmiques pour JavaFX.

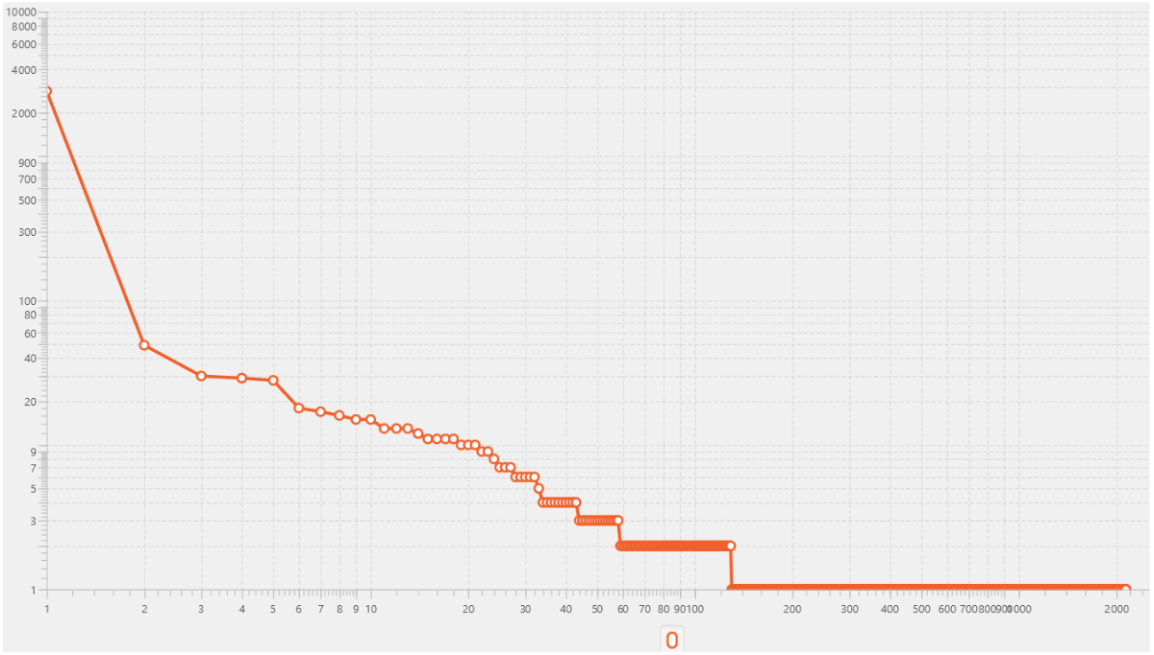
Tests

Dans un projet, la phase de tests est souvent déterminante pour la robustesse du projet. Ici, nous avons décidé d'implémenter des tests effectués régulièrement. En effet, de part le cycle incrémentale, nous avions régulièrement des nouvelles fonctionnalités à implémenter et nous nous efforcions de les tester dans plusieurs configurations différentes et en refaisant tout le traitement de l'image pour être sûr qu'une coquille ne se soit pas glissée. Cependant, les tests étant un pan entier du développement logiciel, nous n'avons pas fait exactement ce qui serait fait en entreprise avec une pipeline CI/CD, des tests unitaires automatisés, etc.
Nos tests se portant sur des images nous avons préférés faire des tests avec une bibliothèque d'image type puis analysé la cohérence de nos résultats.
Par exemple, nous avons pris quatre images tests : La première était une image totalement monochrome et le but était de voir que l'image se transformait en nuances de gris, avec un seul motif qui a un nombre d'occurrences correspondant avec la taille du motif et la taille de l'image.
La seconde image était une image avec plusieurs couleurs et plusieurs formes détectables. On a regardé ensuite dans notre histogramme ordonné que les nombre d'occurrences correspondaient et que la forme des motifs également.
Après, nous avons utilisé une image trouvée dans un document qui contenait sa courbe associée.





La courbe obtenue dans l'étude



Ci-dessus notre histogramme ordonné.
On a donc pu voir que notre programme permettait d'obtenir des résultats semblables à ceux obtenus lors d'études précédentes.

Récapitulatif

D'un point de vue global, ce projet nous a beaucoup appris sur le génie logiciel et la manière dont il faut s'organiser.
Le grand nombre de réunions avec notre tuteur nous a permis de toujours savoir où aller et même si ce n'est pas quelque chose que l'on retrouve tout le temps en entreprise cela reste intéressant.

Concernant le sujet en lui même, nous ne sommes pas sûr d'avoir trouver l'utilité attendue pour la loi de Zipf malgré les tests effectués. Nous pensons que le logiciel permet d'obtenir des résultats cohérents. Nous avons manqué de lucidité vis à vis de certaines parties,

Continuation et ouverture

Ce projet a donc, selon nous, des fondations solides et il peut facilement être repris l'année prochaine. Plusieurs fonctionnalités restent à ajouter pour permettre une meilleure analyse des images échographiques. Notamment mettre en place le recouvrement, comme décrit précédemment, optimiser certains threads et rajouté des threads sur certains traitements qui n'en possèdent pas, et enfin appliquer ces méthodes là sur des portions de l'image dans lesquelles on détecte un changement lors de la respiration.
Enfin dans l'analyse de nos résultats nous n'avons vraiment réussi à déterminer des changements dans les images et à savoir à quoi ils correspondaient sur l'image. Nous ne pouvons donc pas nous avancer quant à l'intérêt d'utiliser la loi de Zipf sur des échographies de poumons.

ANNEXES

Guide d'installation

Pour installer le projet sur votre machine, il faut d'abord récupérer les sources qui sont disponibles en accès libre sur le dépôt GitHub : https://github.com/Dastu6/Zipf_Lung_Java. Ce dépôt possède une licence MIT qui permet à tout le monde de reprendre l'entièreté du dépôt, et donc du code source, et de le modifier à sa guise.

Il existe deux méthodes principales pour avoir le projet sur votre IDE préféré. Cependant le guide sera expliqué pour Eclipse.

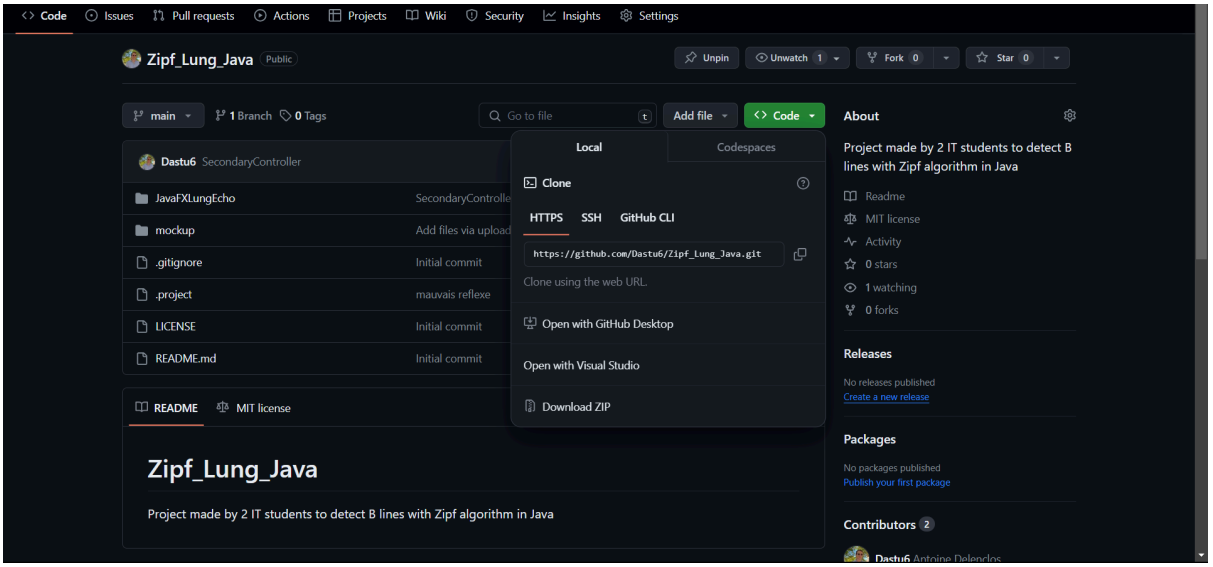
Pour avoir l'ensemble des fichiers :

- Méthode 1 : Télécharger les sources et créer un nouveau projet avec celles-ci.
- Méthode 2 : Cloner le dépôt GitHub.

Nous vous conseillons la méthode 1 car plus rapide et plus flexible par la suite pour reprendre sur un nouveau dépôt Git.

Méthode 1 :

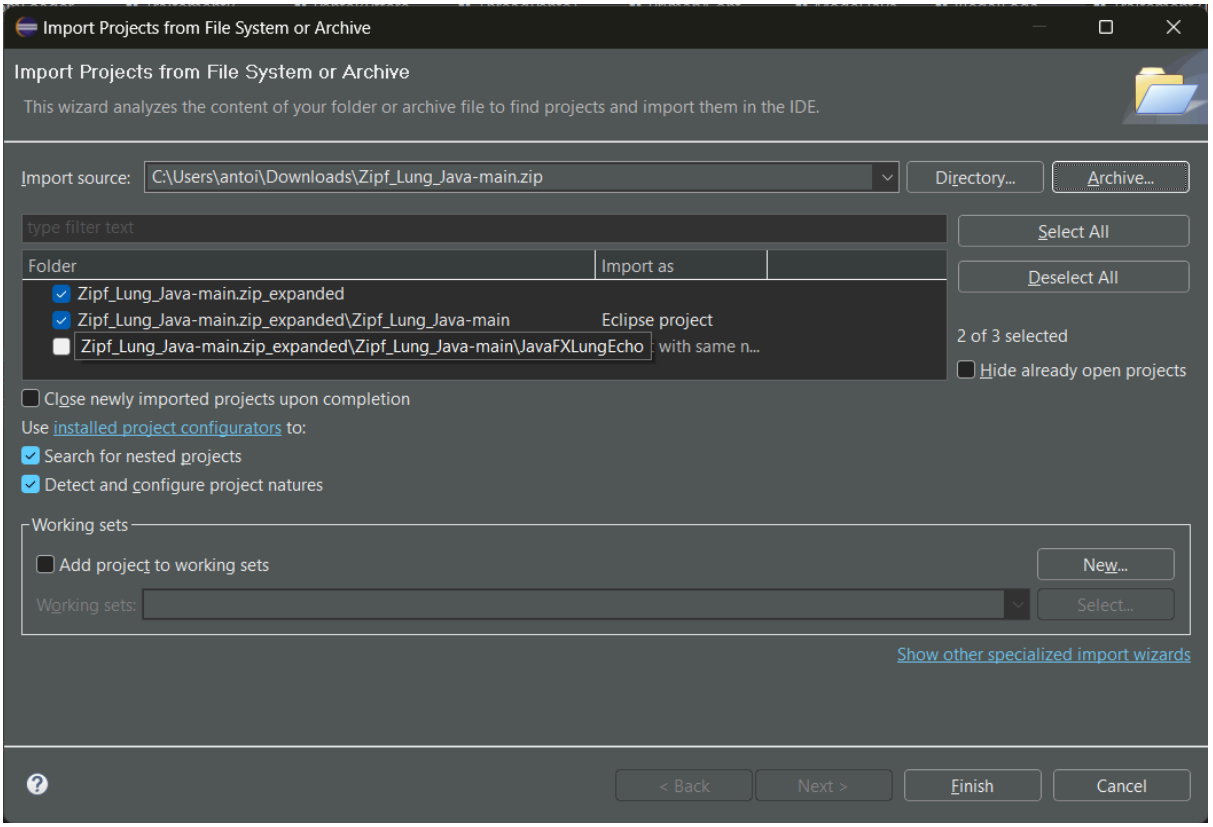
1) Télécharger le ZIP du GitHub : Cliquer sur <> Code → Download ZIP



2) Ajouter JavaFX, Maven et Git : Sur Eclipse → Help → Eclipse Marketplace… → puis installer e(fx)clipse, Eclipse m2e, EGit.

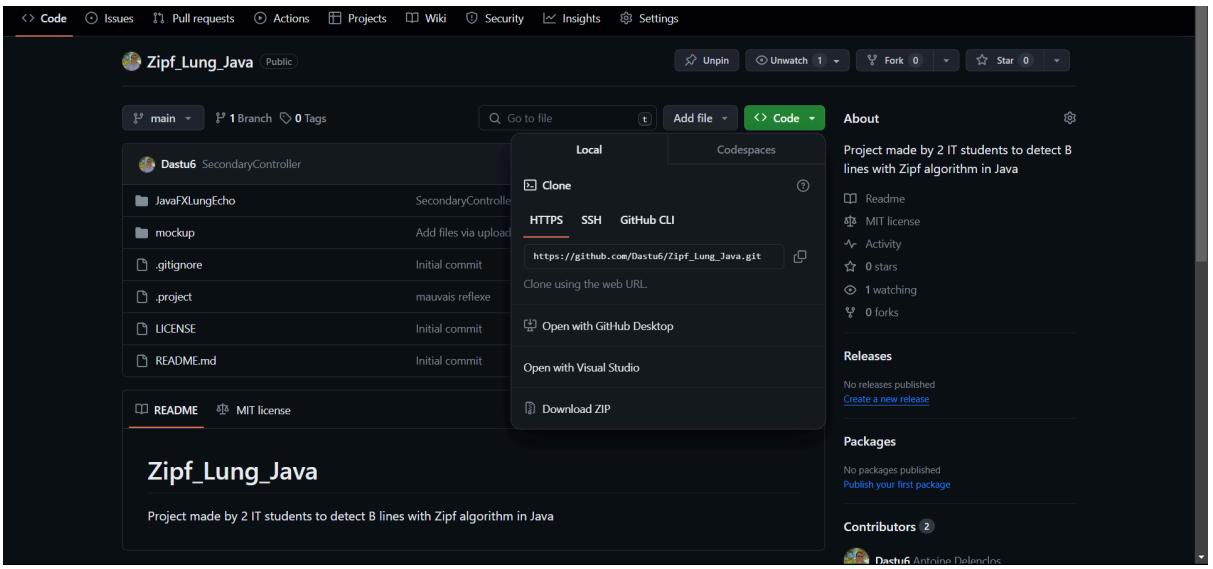
3) Télécharger SceneBuilder à l'adresse suivante : https://gluonhq.com/products/scene-builder/#download

4) Ajouter SceneBuilder si vous souhaitez éditer les fichiers FXML : Sur Eclipse → Window → Preferences → Cliquer sur JavaFX → Mettre le PATH vers l'exécutable de SceneBuilder
5) Sur Eclipse : File → Open Projects from File System → Import Source (Archive) et mettre le ZIP que vous venez de télécharger, puis cliquer sur Finish



Vous avez maintenant le projet sur votre machine et vous pouvez le modifier et l'exécuter.

Méthode 2 :
1) Récupérer le lien : Cliquer sur <> Code → Copy url to clipboard

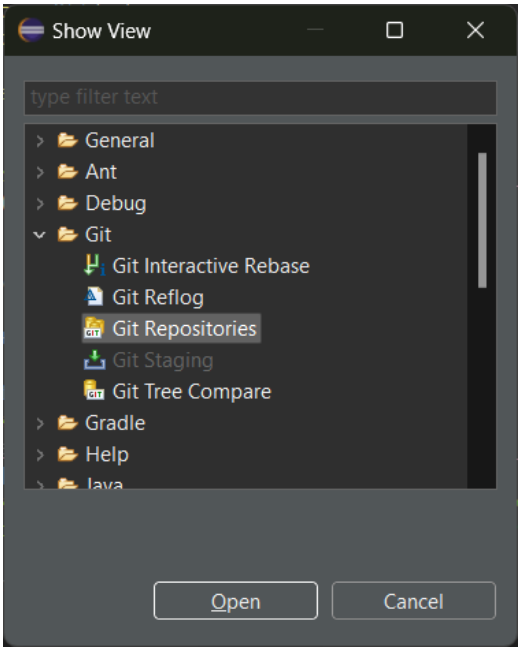


2) Ajouter JavaFX, Maven et Git : Sur Eclipse → Help → Eclipse Marketplace… → puis installer e(fx)clipse, Eclipse m2e, EGit.

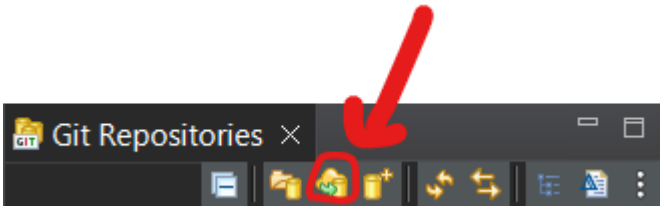
3) Télécharger SceneBuilder à l'adresse suivante : https://gluonhq.com/products/scene-builder/#download

4) Ajouter SceneBuilder si vous souhaitez éditer les fichiers FXML : Sur Eclipse → Window → Preferences → Cliquer sur JavaFX → Mettre le PATH vers l'exécutable de SceneBuilder

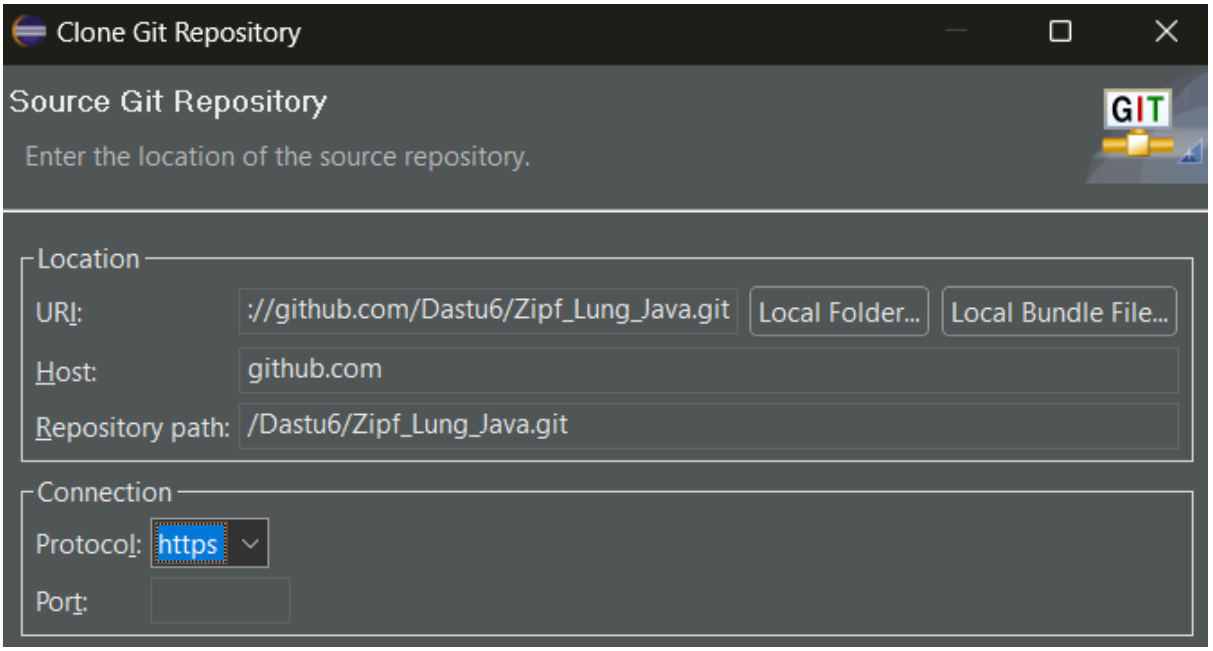
5) Montrer les dépôts Git sur Eclipse : Sur Eclipse → Window → Show View → Other… → Git → Git Repositories



6) Cloner le dépôt : Sur l'onglet Git Repositories d'Eclipse : Cliquer sur Clone a Git Repository and add the clone to this view



7) Dans la fenêtre qui vient de s'ouvrir mettre l'url copié dans la case URL tel que :



Et mettre vos identifiants de connexion à GitHub et cocher la case Store in Secure Store si vous ne souhaitez pas avoir à vous reconnecter à chaque démarrage d'Eclipse. Cliquer sur Next

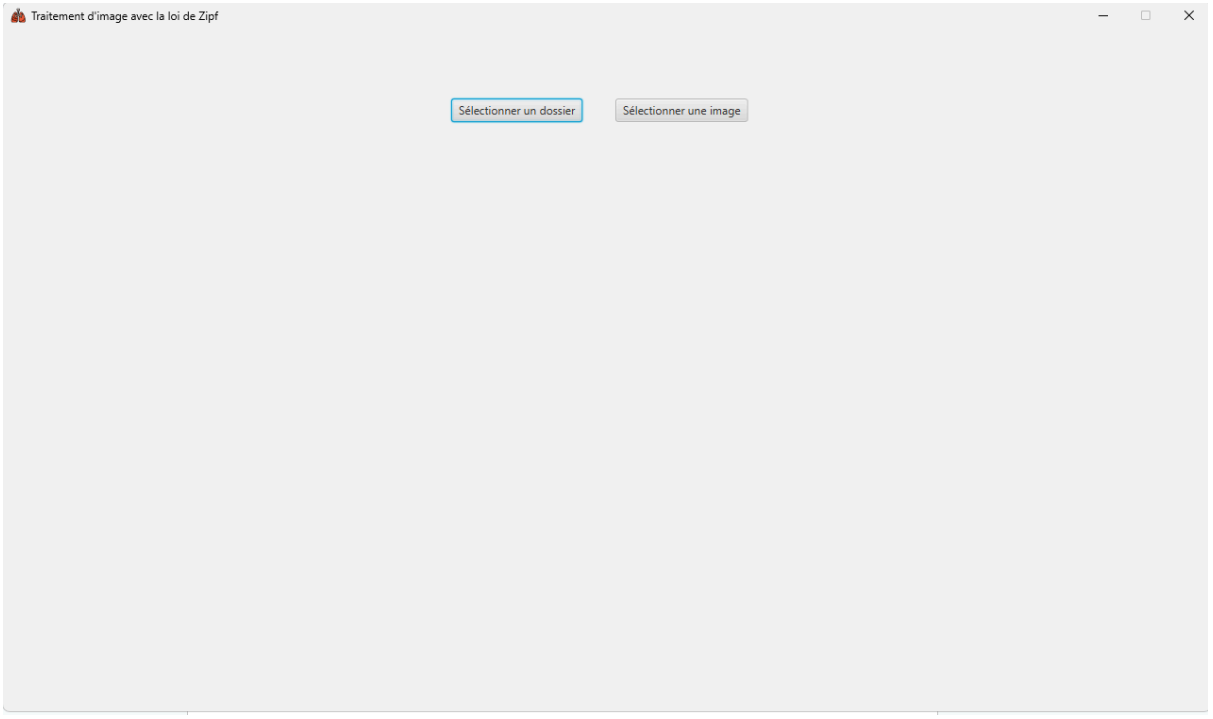
8) Sélectionner la branche main et "When fetching a commit, also fetch its tags" puis cliquer sur Next

9) Sélectionner un endroit dans lequel ajouter ce projet git : Mettez le PATH désiré dans Directory puis cliquer sur Finish

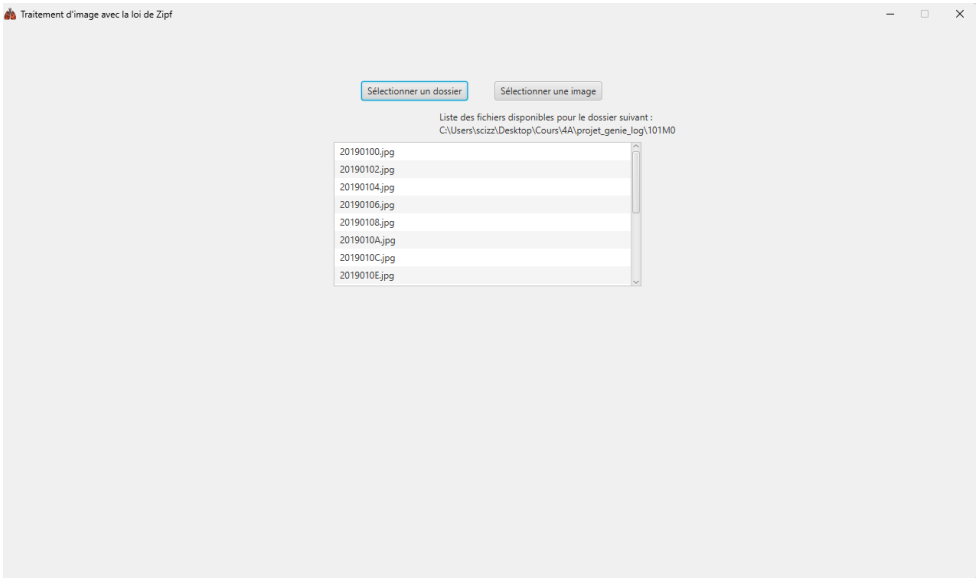
Vous avez maintenant cloné le projet sur votre machine et vous pouvez le modifier et l'exécuter. Vous pouvez également modifier son Git Local associé.

Guide d'utilisation

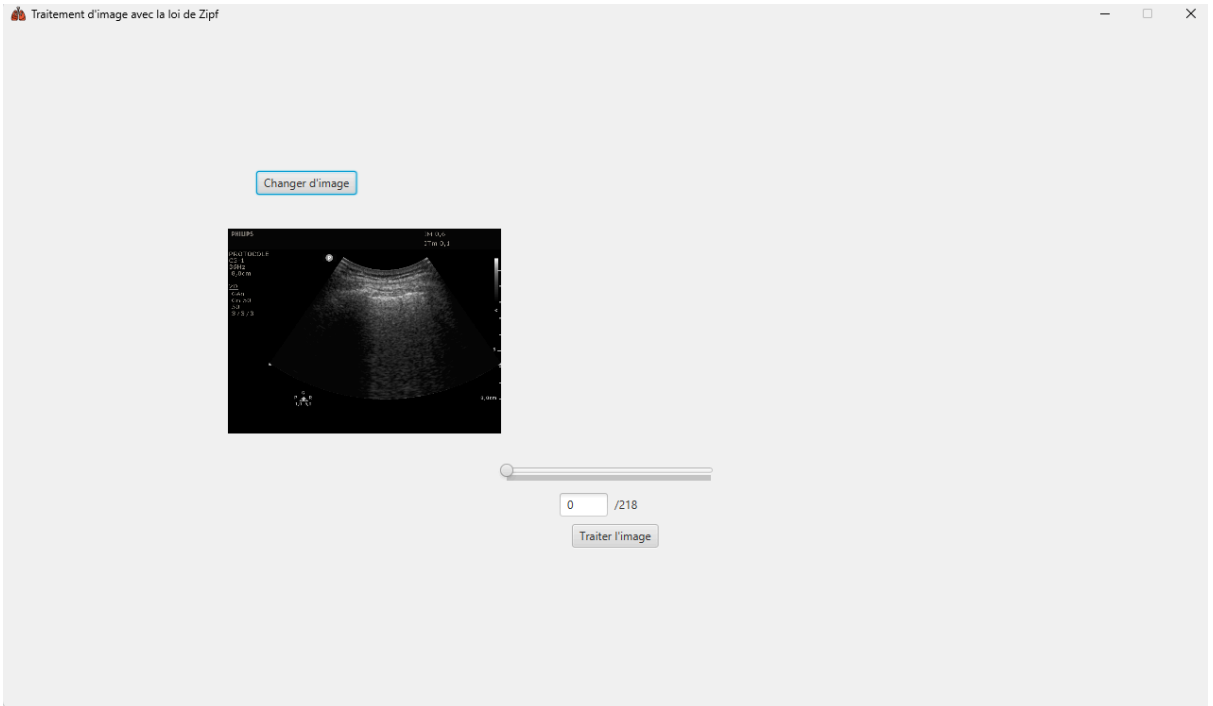
Le logiciel est assez simple d'utilisation, lorsqu'on l'allume le logiciel pour la première fois on est face à cela. On est invité à choisir soit un dossier, soit une image. Les deux n'affichent que les fichiers avec les extensions suivantes :
.png, .jpg, .dcm, .dicom.



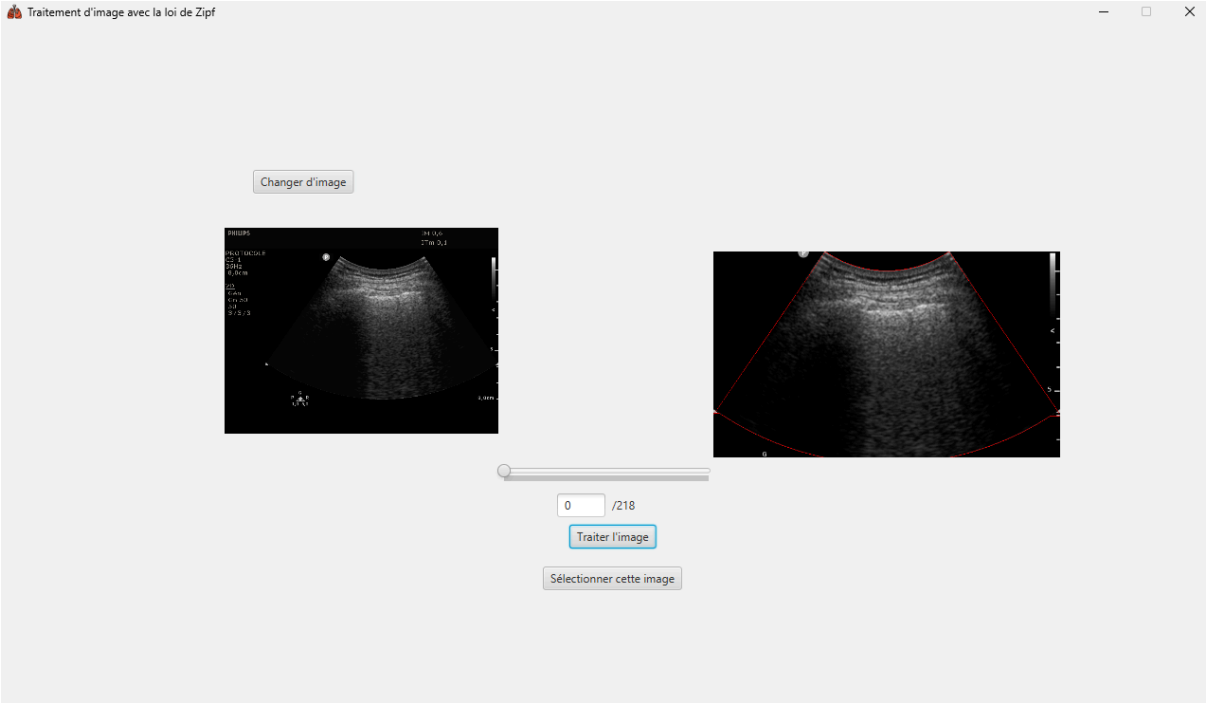
Si on choisit un dossier il remplit la listview en dessous, on peut ensuite cliquer pour sélectionner l'image souhaité, le comportement sera le même en cliquant ou en sélectionnant l'image depuis le bouton "sélectionner une image"



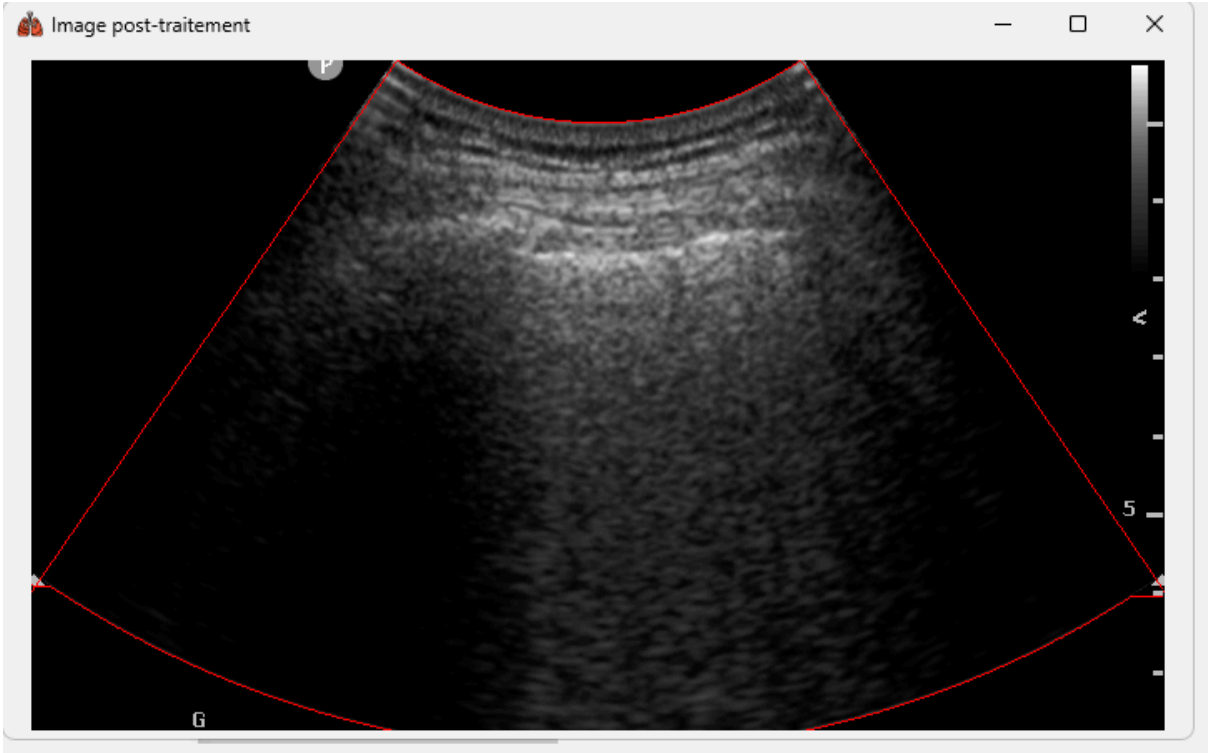
Ensuite on distingue deux cas,
si c'est une image dicom ou autre(.jpg,.png)
dicom :



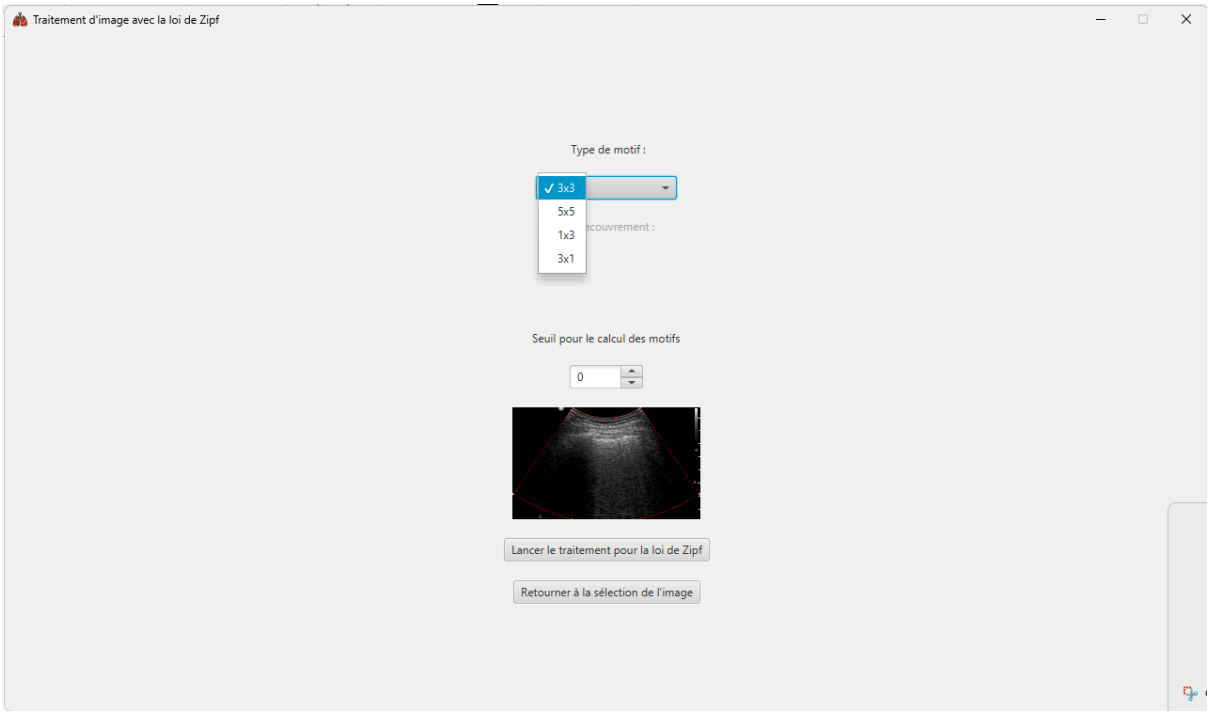
On est amené là-dessus, on peut donc sélectionner la frame désirée (comme indiqué dans la partie sur la conversion DICOM) et ensuite on est invité à traiter l'image pour réduire la zone, augmenter le contraste et détecter les contours de la zone d'échographie et pouvoir appliquer Zipf.



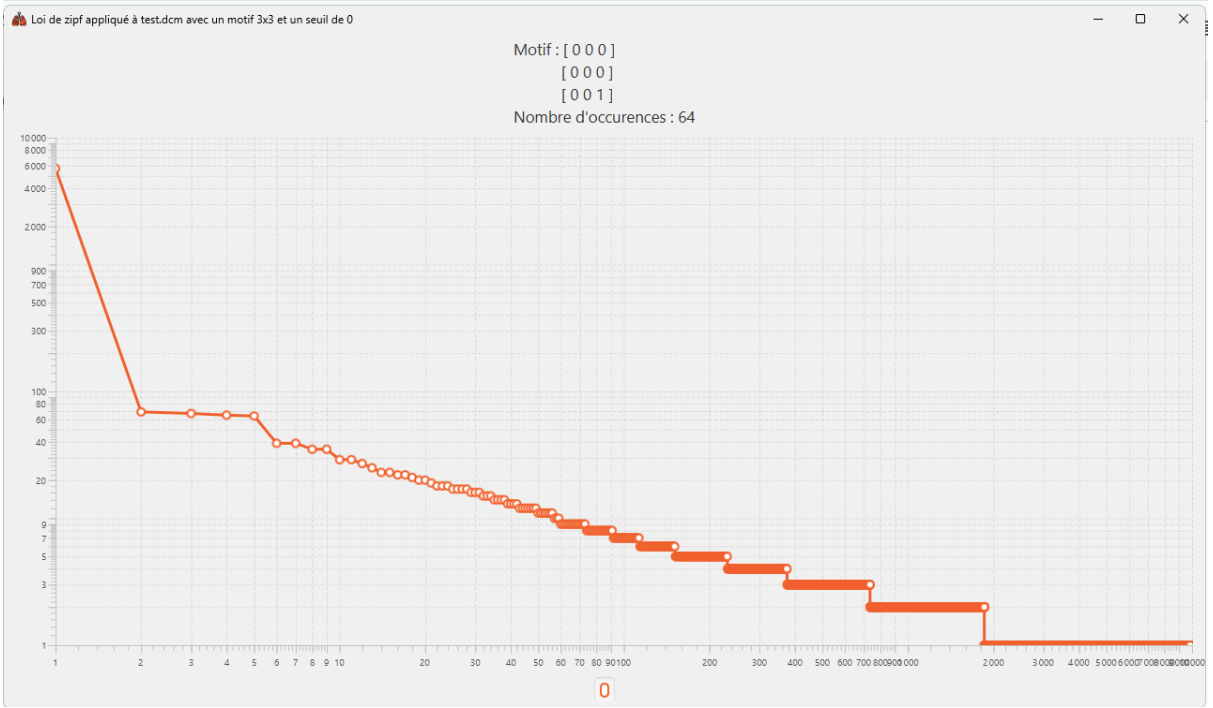
Une fois fait, on peut voir les contours de l'image et l'image recentrée.
On peut ici cliquer sur les images pour les voir en grand.



Ensuite, on peut cliquer sélectionner cette image afin de passer au dernier écran.



Ensuite on arrive sur cet écran où on peut choisir le motif, le seuil et lancer le traitement Zipf.



Une fois le traitement lancé, une nouvelle fenêtre s'ouvre avec la courbe de Zipf pour le motif et le seuil sélectionné.
Si on passe sa souris sur un point, on voit le motif auquel il correspond et le nombre d'occurrences associées.