Institut Polytechnique de Paris

Projet segmentation d'image SIM202

Étudiants :
Mohamed BENLOUGHMARI
Ewerthon MELZANI
Arnaud PELISSIER

Professeur :
Luiz Faria

17 mars 2025

Table des matières

1 Introduction

Ce projet vise à détecter automatiquement les frontières des objets présents dans une image en utilisant l'algorithme de Chan et Vese. Pour ce faire, nous implémenterons des algorithmes permettant de partitionner l'image en différentes régions partageant des caractéristiques similaires, telles que la couleur, avant d'extraire leurs contours. Par souci de simplicité, nous nous concentrerons dans un premier temps sur des images en niveaux de gris et supposerons que les objets à détecter sont plus sombres que leur environnement.

2 Positionnement du problème

Dans ce travail, nous modélisons une image sous la forme d'une fonction $f : \Omega \to \mathbb{R}$, où $\Omega = [-1, 1] \times [-1, 1]$ représente le domaine de l'image. L'objectif est de construire une courbe fermée Γ qui partitionne ce domaine en deux régions disjointes : une région intérieure, notée Ω^-, et une région extérieure, notée Ω^+. L'algorithme de Chan-Vese repose sur la construction d'une fonction constante par morceaux, définie comme suit :

$$u(x) = \begin{cases} c^+ & \text{si } x \in \Omega^+, \\ c^- & \text{si } x \in \Omega^-, \end{cases}$$

où c^+ et c^- sont des constantes représentant respectivement les valeurs moyennes de l'image dans les régions Ω^+ et Ω^-. Cette fonction $u(x)$ est déterminée de manière à minimiser une fonctionnelle d'énergie, donnée par :

$$\mathcal{I}[u] = \mu \,\text{length}(\Gamma) + \nu \,\text{area}(\Omega^-) + \lambda_1 \int_{\Omega^-} |f(x) - c^-|^2 \, dx + \lambda_2 \int_{\Omega^+} |f(x) - c^+|^2 \, dx,$$

où μ, ν, λ_1 et λ_2 sont des paramètres strictement positifs ($\mu, \nu, \lambda_1, \lambda_2 > 0$). Le premier terme, $\mu \,\text{length}(\Gamma)$, régularise la longueur de la courbe Γ, le second, $\nu \,\text{area}(\Omega^-)$, contrôle la taille de la région intérieure, tandis que les deux derniers termes mesurent la fidélité de l'approximation par rapport à l'image f.

Le problème de segmentation peut alors être formulé mathématiquement comme suit : étant donnée une image représentée par $f : \Omega \to \mathbb{R}$, déterminer les constantes c^+ et c^- ainsi que la courbe Γ qui minimisent la fonctionnelle $\mathcal{I}[u]$. Cela revient à résoudre :

$$\arg \min_{c^+, c^-, \Gamma} \mathcal{I}[u]. \tag{1}$$

2.1 Méthode des surfaces de niveau

L'idée fondamentale de la méthode des surfaces de niveau [1] consiste à représenter l'interface Γ sous la forme de l'ensemble de niveau $\phi^{-1}(0)$, où $\phi : \mathbb{R}^2 \to \mathbb{R}$ est une fonction définie de manière appropriée. Cette approche permet de reformuler toute intégration dans la fonctionnelle d'énergie précédente sous la forme d'une intégration sur le domaine Ω. Ainsi, l'énergie $\mathcal{I}[u]$ s'exprime comme suit :

$$\mathcal{I}[u] = \mu \int_\Omega \delta(\phi(x)) |\nabla \phi(x)| \, dx + \nu \int_\Omega H(-\phi(x)) \, dx$$

$$+\lambda_1 \int_\Omega |f(x) - u^-|^2 H(-\phi(x)) \, dx + \lambda_2 \int_\Omega |f(x) - u^+|^2 H(\phi(x)) \, dx,$$

où H désigne la fonction de Heaviside, δ la distribution de Dirac, et u^- et u^+ les valeurs de la fonction u respectivement à l'intérieur et à l'extérieur de Γ.

Pour une interface Γ fixée, les constantes c^+ et c^- peuvent être calculées explicitement selon les expressions suivantes :

$$c^+ = \frac{\int_{\Omega^+} f(x)\,dx}{\int_{\Omega^+} dx}, \quad c^- = \frac{\int_{\Omega^-} f(x)\,dx}{\int_{\Omega^-} dx},$$

où Ω^+ et Ω^- représentent respectivement les régions intérieure et extérieure délimitées par Γ.

Afin d'éviter les difficultés numériques et théoriques liées à l'utilisation d'une distribution δ dans l'équation, nous introduisons une approximation régulière de la fonction de Heaviside, notée H_ϵ, et de la distribution de Dirac, notée δ_ϵ, définies par :

$$H_\epsilon(t) = \frac{1}{2}\left(1 + \frac{2}{\pi}\arctan\left(\frac{t}{\epsilon}\right)\right), \tag{2}$$

$$\delta_\epsilon(t) = \frac{d}{dt}H_\epsilon(t) = \frac{\epsilon}{\pi(\epsilon^2 + t^2)}, \tag{3}$$

où $\epsilon > 0$ est un paramètre de régularisation contrôlant la largeur de la transition.

La minimisation de la fonctionnelle $\mathcal{I}$ est réalisée par une méthode d'Euler-Lagrange. En remplaçant les fonctions H et δ par leurs versions régularisées H_ϵ et δ_ϵ, on obtient l'équation suivante :

$$\delta_\epsilon(\phi) = \frac{1}{\mu}\left[\nu + \lambda_1(f - c^-)^2 - \lambda_2(f - c^+)^2 - \mu\nabla\cdot\left(\frac{\nabla\phi}{|\nabla\phi|}\right)\right] \quad \text{sur } \Omega, \tag{4}$$

$$\frac{\partial\phi}{\partial\vec{n}} = 0 \quad \text{sur } \partial\Omega. \tag{5}$$

où $\vec{n}$ désigne la normale extérieure à la frontière $\partial\Omega$.

Cette équation, qui constitue un problème aux valeurs aux limites non linéaire, est difficile à résoudre directement. Une approche couramment utilisée consiste à introduire une variable temporelle artificielle t et à résoudre un problème d'évolution associé :

$$\frac{\partial\phi}{\partial t} = \delta_\epsilon(\phi)\left[\mu\nabla\cdot\left(\frac{\nabla\phi}{|\nabla\phi|}\right) - \nu - \lambda_1(f - c^-)^2 + \lambda_2(f - c^+)^2\right] \quad \text{sur } \Omega, \tag{6}$$

$$\frac{\partial\phi}{\partial\vec{n}} = 0 \quad \text{sur } \partial\Omega. \tag{7}$$

L'idée sous-jacente est de faire évoluer ϕ dans ce temps artificiel jusqu'à atteindre un état stationnaire, ce qui revient à résoudre le problème initial. Cette approche, plus simple à traiter numériquement que le problème aux valeurs aux limites original, est celle que nous adopterons ici.

L'algorithme suivant résume les étapes principales de cette méthode :

Algorithm 1 Algorithme principal pour la résolution de l'équation d'évolution

1: Initialiser la fonction de niveau ϕ ;
2: Initialiser $\tau \leftarrow \infty$;
3: **while** $\tau >$ tolérance **do**
4: Calculer c^+ et c^- ;
5: Faire évoluer ϕ dans le temps sur un intervalle Δt pour obtenir ϕ_{nouveau} en utilisant (6) ;
6: Estimer l'erreur$\tau \leftarrow \|\phi - \phi_{\text{nouveau}}\|_2$;
7: **end while**

2.2 Discrétisation

Considérons un maillage cartésien de Ω constitué de $M+1$ points par dimension. Les nœuds du maillage sont alors définis par $x_{i,j} = (ih, jh)$ pour $0 \leq i, j \leq M$, où $h = 1/M$ représente le pas de maillage. La valeur de ϕ au point $x_{i,j}$ est notée $\phi_{i,j}$.

Le schéma de discrétisation spatio-temporelle que nous considérons est donné par :

$$\frac{\phi_{i,j}^{n+1} - \phi_{i,j}^{n}}{\Delta t} = \delta_\epsilon\left(\phi_{i,j}^n\right)\left[\mu\left(\frac{\nabla_x^- \phi_{i,j}^{n+1}}{\sqrt{\eta^2 + \left(\nabla_x^+ \phi_{i,j}^n\right)^2 + \left(\nabla_y^0 \phi_{i,j}^n\right)^2}} + \frac{\nabla_y^- \phi_{i,j}^{n+1}}{\sqrt{\eta^2 + \left(\nabla_x^0 \phi_{i,j}^n\right)^2 + \left(\nabla_y^+ \phi_{i,j}^n\right)^2}}\right)\right]. \tag{8}$$

Nous avons ici supposé que $\nabla_y^- \phi_{i,j}^{n+1} \approx \nabla_y^- \phi_{i,j}^n$ et $\nabla_y^+ \phi_{i,j}^{n+1} \approx \nabla_y^+ \phi_{i,j}^n$. Cette approximation permet d'exprimer $\phi_{i,j}^{n+1}$ en fonction de $\phi_{i,j}^n$, ce qui simplifie grandement l'implémentation de l'algorithme. Cependant, cette approche peut légèrement réduire la stabilité du schéma.

2.3 Conditions aux Limites

Les conditions aux limites sont définies comme suit :

1. $\phi_{0,j}^{n+1} = \phi_{1,j}^{n+1} \quad \forall j$ et $\phi_{M,j}^{n+1} = \phi_{M-1,j}^{n+1} \quad \forall j$.
2. $\phi_{i,0}^{n+1} = \phi_{i,1}^{n+1} \quad \forall i$ et $\phi_{i,M}^{n+1} = \phi_{i,M-1}^{n+1} \quad \forall i$.

Avec cette discrétisation en place, il est possible de faire évoluer ϕ sur une durée donnée ΔT. À chaque itération, les valeurs des constantes $c^\pm$ sont mises à jour via l'équation (6). L'erreur est ensuite estimée et l'algorithme continue jusqu'à ce qu'une tolérance prédéfinie soit atteinte, comme décrit dans l'algorithme 1.

3 Implémentation

Le problème développé ci-dessus manipule des données volumineuses et demande de nombreux calculs. Afin d'implémenter efficacement une solution il faut une combinaison de structure de données et d'abstractions adaptées permettant d'effectuer des modifications facilement sans pour autant perdre en performance.

Nous allons dans un premier temps s'intéresser à la representation des données du problème. Puis à la structure globale du projet et à l'implémentation de l'algorithme 1.

3.1 Représentation des images

Afin de sauvegarder et de visualiser les images manipulées, il est nécessaire de choisir un format de travail. Les plus connus sont les formats `JPEG` et `PNG` mais nous avons fait le choix d'utiliser le format de fichier `BITMAP`. En effet ce dernier est sans perte comparé au `JPEG` et il est beaucoup plus facile à implémenter que le `PNG` sans avoir à utiliser de bibliothèque extérieure. Ce choix nous a permis d'obtenir un prototype utilisable très rapidement. Ceci définit donc le format de l'image sur le disque dur mais il reste à représenter l'image en mémoire.

Afin de simplifier le problème, on suppose que les images sont toutes en niveau de gris. Il est toujours possible de travailler avec des images en couleurs, le programme va alors convertir les pixels $(r, g, b) \in [0, 1]^3$ en niveau de gris $c \in [0, 1]$. Pour effectuer cette conversion, la formule suivante est utilisée :

$$c = 0,3r + 0,59g + 0,11b$$

Le niveau de gris est obtenu par une combinaison linéaire des couleurs, plutôt que par leur simple moyenne. Les différents poids utilisés dans cette combinaison reflètent la sensibilité de l'œil humain aux différentes couleurs. [2]

L'algorithme de résolution du problème nécessite de manipuler de façon fréquente des fonctions du type $f : [0, 1]^2 \mapsto [0, 1]$ où $f(x, y)$ représente le niveau de gris du pixel de coordonnéesx, y.

Afin de rendre ce processus le plus simple possible, nous avons fait le choix de n'utiliser qu'une classe `GridFunction` qui représente à la fois une image et une fonction. Elle contient donc à la fois les fonctions de sauvegarde et de chargement d'images depuis le format `BITMAP` mais aussi les fonctions mathématiques telles que le calcul de gradient.

Enfin, une problématique supplémentaire sont les valeurs que peuvent prendre les coordonnées x et y. En effet les valeurs sauvegardées sur le disques sont discrètes :

$$x \in [|0, width - 1|], y \in ||0, height - 1|]$$

Le choix qui a été initialement fait était de rendre les images chargés continues en programmant les fonctions d'accès aux pixels de façons opaques et en interpolant linéairement la valeur des 4 pixels les plus proches sur la grille.

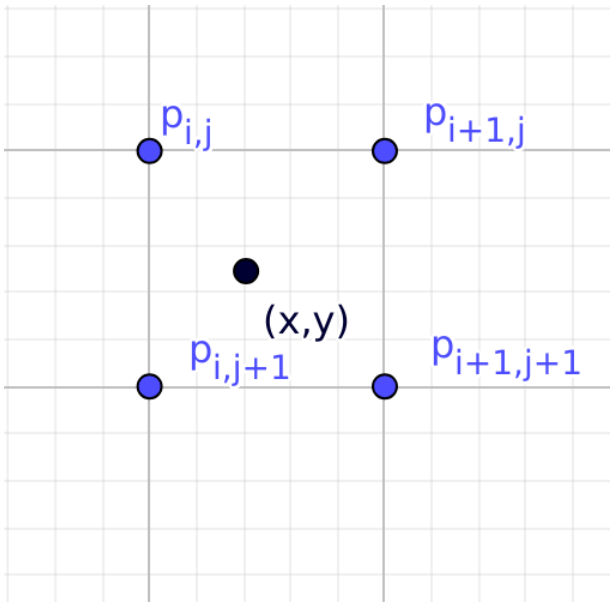

FIGURE 1 – Calcul de la valeur du pixel (x, y) comme interpolation des pixels $p_{i,j}$, $p_{i+1,j}$,$p_{i,j+1}$ et $p_{i+1,j+1}$

Or cette Approche s'est avérée problématique lors du calcul du gradient en un pixel car la connaissance de la résolution est alors essentielle. Par conséquent nous avons fait le choix de travailler avec des coordonnées discrètes.

```
class GridFunction
{
public:
    GridFunction(uint32_t width, uint32_t height, double base_value = 0.0f);

    inline double &operator()(uint32_t x, uint32_t y) {
        return m_data[x + y*m_width];
    }

    // ...
private:
    // ...
    uint32_t m_width = 0, m_height = 0;
    std::vector<double> m_data;

};
```

FIGURE 2 – Variables de la classe GridFunction

3.2 Structure du programme

L'algorithme principale requiert de nombreux paramètres ainsi que plusieurs fonctions auxilliaires, afin de simplifier la structure du code nous avons choisi d'envelopper cet algorithme dans une classe `EdgeDetector` et de définir les paramètres du problèmes (λ_i,

μ, ...) mais aussi les paramètres de la simulation (`max_iterations`, `log_iterations`, ...) dans une structure `Parameters`.

Grâce à ce choix, notre solveur est simple à utiliser et pourrait facilement être converti en une bibliothèque.

```
GridParameters parameters;
EdgeDetector edge_detector(parameters);
GridFunction photo_image("input.pgm");
auto result = edge_detector.solve(photo_image);
photo_image.save_to_pgm("output.pgm");
```

FIGURE 3 – API de détéction de bords

```
Project
├── include
│   ├── GridFunction.hpp
│   ├── Parameters.hpp
│   ├── EdgeDetector.hpp
│   └── Config.hpp
├── src
│   ├── main.cpp
│   ├── GridFunction.cpp
│   ├── Parameters.cpp
│   ├── EdgeDetector.cpp
│   └── Config.cpp
└── CMakeLists.txt
```

FIGURE 4 – Arboréscence du projet

- Les fichiers `Config.*pp` permettent de charger les paramètres du problème depuis un fichier texte afin de ne pas avoir à recompiler le programme lorsque l'on souhaite modifier un paramètre.
- `GridFunction.*pp` contiennent les fonctions de chargement et de création d'images ainsi que les opérations mathématiques sur ces images.
- `EdgeDetector.*pp` contient l'implémentation de l'algorithme 1.
- `Parameters.hpp` contient la structure qui stocke les paramètres du programme.
- `main.cpp` analyse les paramètres donnés en ligne de commande par l'utilisateur et applique l'algorithme sur ces derniers.

Enfin, les abstractions définies ci-dessus permettent d'exprimer efficacement l'algorithme de résolution sans avoir à s'attarder sur de détails trop bas niveau. Elle semblent donc bien adaptées au problème.

```
GridFunction EdgeDetector::solve(const GridFunction &image) {
    double tho=1e10;
    size_t current_iteration = 0;
    GridFunction phi_current = create_sin_image(
        image.get_width(),
        image.get_height(),
        30.0, 0.0, 1.0
    );
    GridFunction phi_new(phi_current);
    while(
        tho > m_parameters.tolerance &&
        current_iteration < m_parameters.max_iterations) {
        // Compute C-, and C+
        double c_positive = image.C_positive(phi_current);
        double c_negative = image.C_negative(phi_current);
        // Use euler method for
        phi_new = calc_step(phi_current,image,c_positive,c_negative,phi_new);
        // Calculate the error
        tho = img_distance(phi_current, phi_new);
        // One step
        ++current_iteration;
        phi_current = phi_new;
    }
    return phi_current;
```

FIGURE 5 – Algorithme de détéction des bords

4 Résultats

Pour évaluer les performances de notre implémentation, nous avons initialement testé notre algorithme sur une forme géométrique simple, à savoir un cercle. La figure 6 illustre l'évolution de notre programme à travers différentes itérations, mettant en évidence la progression du traitement de l'image.

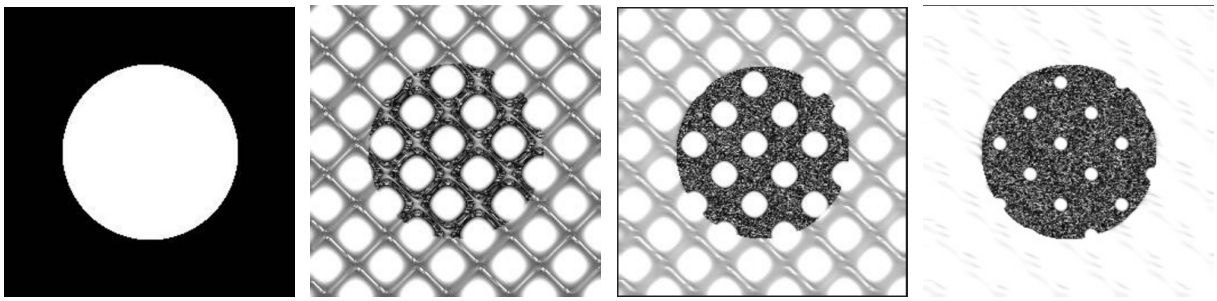

Image initiale | Itération 5 | Itération 15 | Itération30

FIGURE 6 – Évolution du traitement de l'image du cercle à travers différentes itérations.

Le résultat final obtenu est présenté dans la figure 7 :

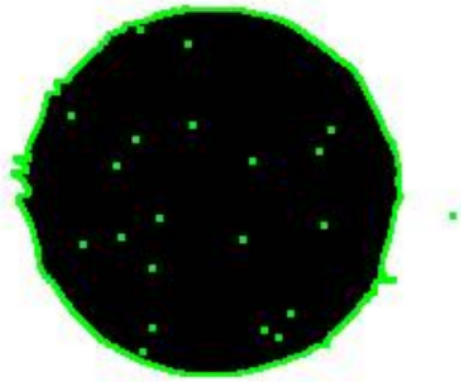

FIGURE 7 – Image finale après convergence complète de l'algorithme.

Les paramètres utilisés pour cette simulation sont les suivants :

- $\epsilon = 0.2$
- $v = 0.0$
- $\mu = 0.1$
- $\lambda_1 = 0.6$
- $\lambda_2 = 0.6$
- Tolérance = 5.0
- $\Delta t = 2.0$
- $\eta = 0.1$
- Nombre maximal d'itérations = 100
- Intervalle de journalisation = 1
- Intervalle de sauvegarde = 1

On observe que la convergence est effectivement atteinte. Cependant, la présence de bruit dans l'image empêche une segmentation binaire claire en deux couleurs distinctes. Pour pallier ce problème, nous avons calculé la moyenne des intensités de couleur dans l'image bruitée afin de définir un seuil. Pour chaque pixel (i, j), si son intensité est inférieure à ce seuil, il est assigné à la couleur noire ; sinon, il est assigné à la couleur blanche. Par la suite, le contour est extrait à partir de cette binarisation.

4.1 Tests sur d'autres images

Pour valider la robustesse de notre approche, nous avons appliqué l'algorithme à deux images supplémentaires : une image inspirée de l'ensemble de Mandelbrot et une image nommée "PengBrew".

4.1.1 Image PengBrew

La figure 8 illustre les étapes du traitement de l'image "PengBrew" :
Le résultat final est présenté en figure 9 :

4.1.2 Image Mandelbrot

De manière similaire, la figure 10 montre les étapes du traitement appliqué à l'image de type Mandelbrot :

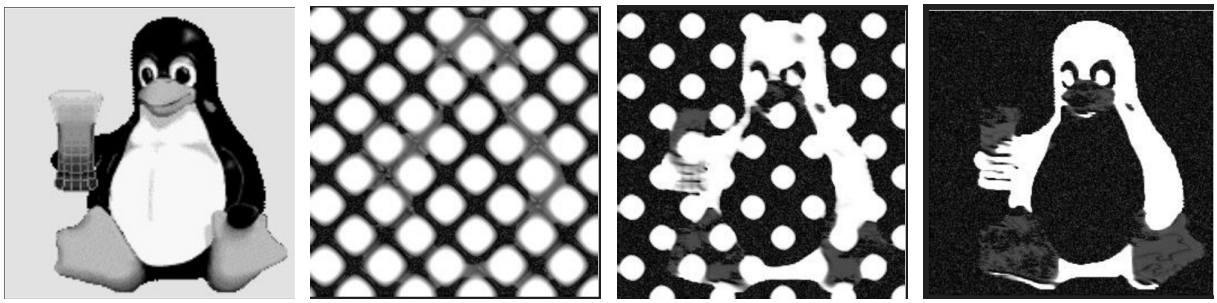

Image initiale | Itération 5 | Itération 100 | Itération 300

FIGURE 8 – Évolution du traitement de l'image PengBrew à travers différentes itérations.

FIGURE 9 – Image finale après traitement complet.

Le résultat final est présenté dans la figure 11 :

4.2 Discussion

On constate que notre algorithme parvient à identifier des contours adaptés dans les trois cas étudiés. Cependant, des limitations apparaissent dans certains scénarios. Par exemple, pour l'image "PengBrew", les nuances de gris posent problème, rendant la segmentation binaire moins efficace. De même, dans le cas de l'image Mandelbrot, les détails fins ne sont pas toujours bien capturés. Ces observations suggèrent que des ajustements, tels qu'une gestion plus sophistiquée des niveaux de gris ou une amélioration de la résolution spatiale, pourraient être envisagés pour optimiser les performances de l'algorithme.

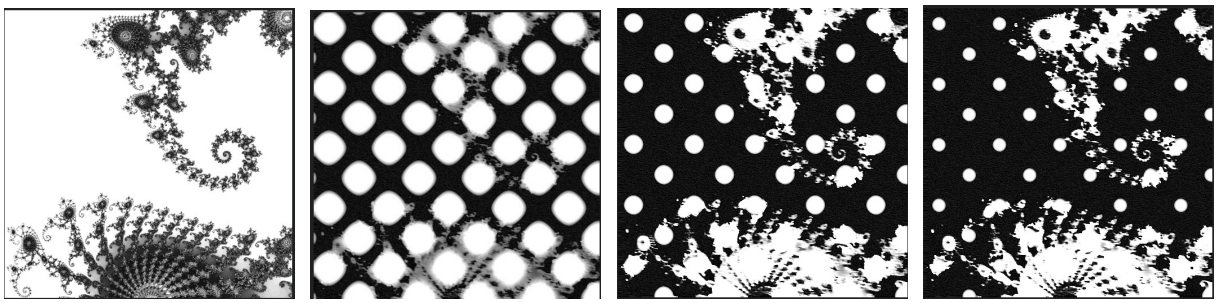

Image initiale | Itération 5 | Itération 100 | Itération 300

FIGURE 10 – Évolution du traitement de l'image Mandelbrot à travers différentes itérations.

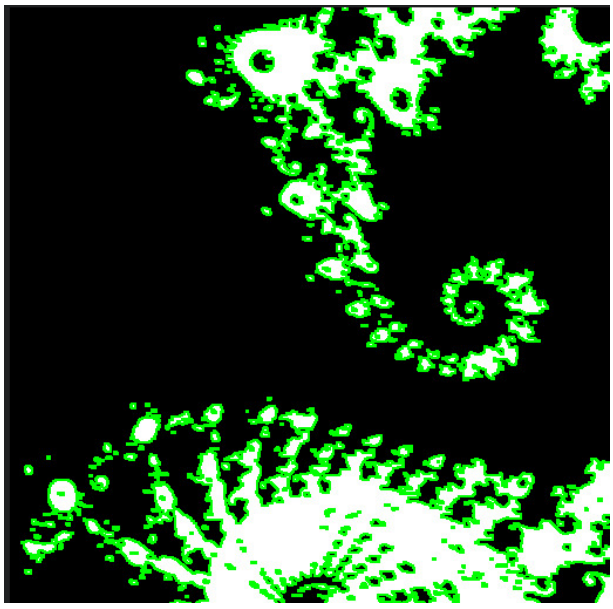

FIGURE 11 – Image finale après convergence.

5 Conclusion

Ce projet a permis de développer et d'implémenter avec succès l'algorithme de Chan-Vese pour la segmentation d'images, en s'appuyant sur la méthode des surfaces de niveau. Les tests réalisés sur des images variées, telles qu'un cercle, 'PengBrew' et une image de type Mandelbrot, ont démontré la capacité de l'algorithme à détecter des contours de manière robuste, bien que des limites aient été observées, notamment dans la gestion des nuances de gris et des détails fins. Ces résultats ouvrent la voie à des améliorations futures, comme l'intégration de techniques avancées de prétraitement ou une optimisation des paramètres, pour renforcer la précision et la généralisation de la méthode.

Références

[1] T.F. Chan and L.A. Vese. Active contours without edges. *IEEE Transactions on Image Processing*, 10(2) :266–277, 2001.

[2] Martin Čadík. Perceptual evaluation of color-to-grayscale image conversions.