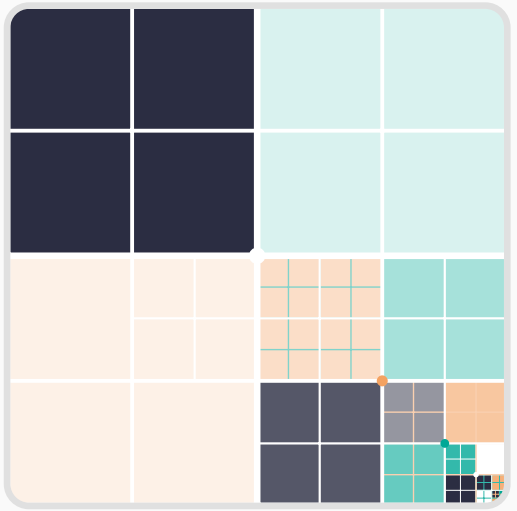
MP 2026 • Num Scei : 18260

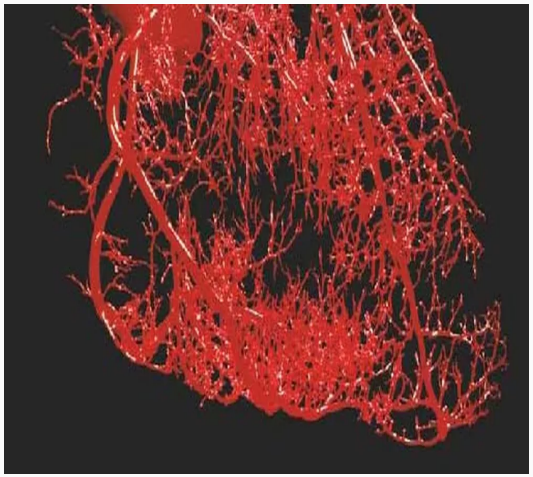
Super-résolution fractale : Étude comparative des cycles PIFS et FIF appliqués à l'imagerie médicale.



Mohamed Amine Hzikar

Chou Romanesco

Fig. 1



Réseau vasculaire

Fig. 2

Fougère

Fig. 3

Illustration de l'auto-similarité et du motif itératif dans la nature

"Face à l'interpolation polynomiale, quelle approche fractale, PIFS ou FIF, optimise le compromis PSNR/SSIM pour la super-résolution d'images médicales ?"

Théorème du Point Fixe de Banach

Soit (E, d) un espace métrique complet.
Soit $W : E \to E$ une application contractante de rapport $c \in [0, 1[$:

$$d(W(x), W(y)) \leq c \cdot d(x, y)$$

$\Longrightarrow$ **Existence et unicité** d'un point fixe $a \in E$ tel que $W(a) = a$.

Application au Contexte Médical

- **L'espace** E : L'espace des images.
- **Notation** : $f : \Omega \to \mathcal{C}$ (Fonction spatiale vers intensité lumineuse).
- **La métrique** L^2 : Erreur Quadratique Moyenne :

$$d_2(f, g) = \left(\frac{1}{|\Omega|} \iint_\Omega |f(x, y) - g(x, y)|^2 \, dxdy \right)^{\frac{1}{2}}$$

- **L'opérateur** W : Application contractante de transformation d'image.

Convergence absolue des itérées :

$$\forall x_0 \in E, \quad \lim_{n\to\infty} W^{\circ n}(x_0) = a$$

Application à l'image : $\lim_{n\to\infty} W^{\circ n}(I_0) = A$

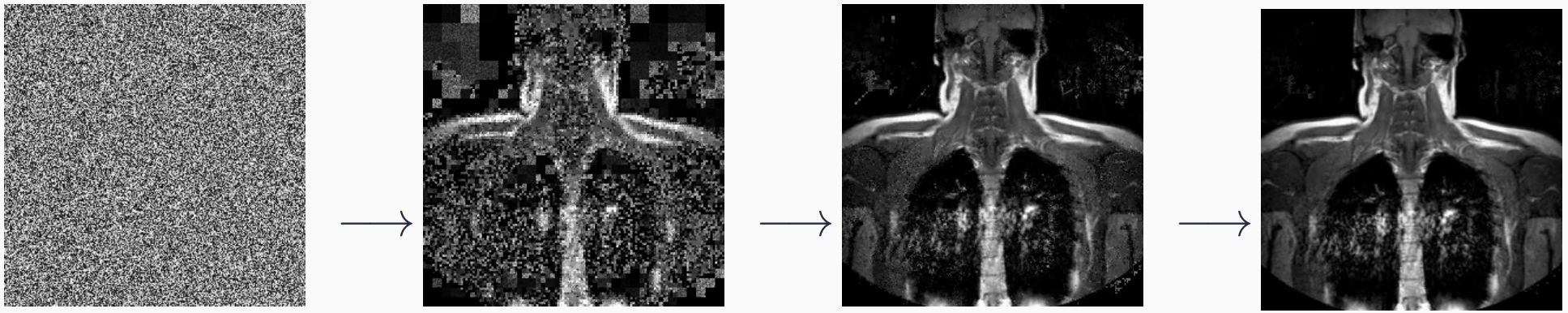


Fig. 4 (Kaggle)

Bruit / I_0 $W^{\circ 5}(I_0)$ $W^{\circ 10}(I_0)$ Attracteur A

Le Théorème du Collage

$$d(I, A) \leq \frac{d(I, W(I))}{1 - c}$$

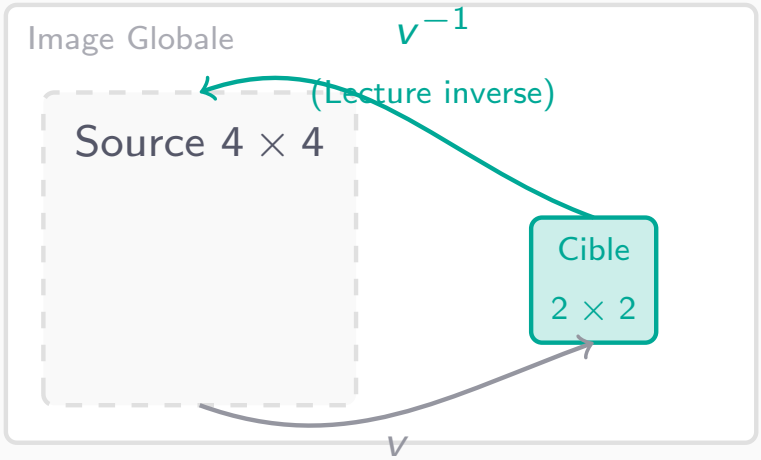
où c est le facteur de contraction

L'objectif algorithmique est de minimiser $d(I, W(I))$ pour forcer l'attracteur A à cloner notre image médicale I.

$$W_{local}(f)(x,y) = s \cdot f(v^{-1}(x,y)) + \text{shift}(x,y)$$

Exemple élémentaire :

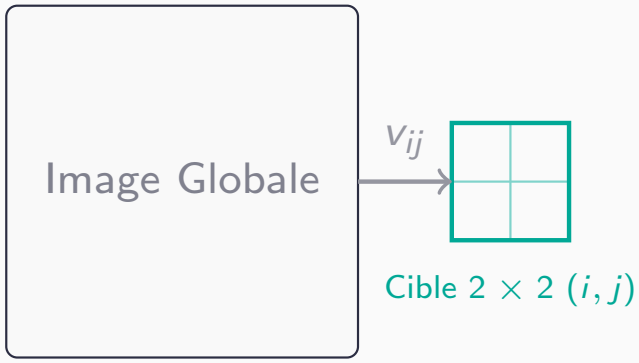
Si v contracte une source 4×4 vers une cible 2×2 (ratio $1/2$), l'ordinateur utilise l'inverse v^{-1} (ratio 2) pour multiplier les coordonnées du pixel cible afin d'aller lire la couleur correspondante dans la source.



La condition de contraction absolue : $\forall$ morceau, $|s| < 1$

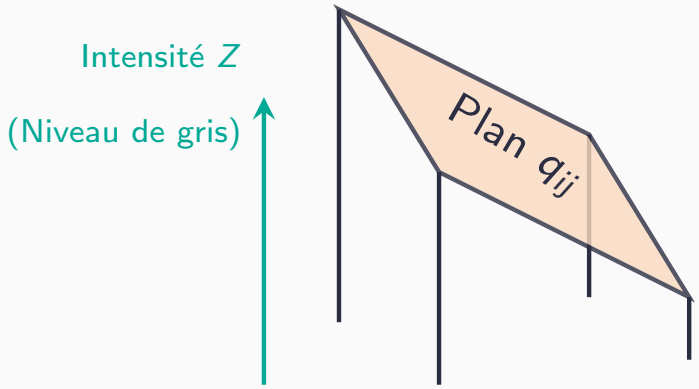
Équation FIF (par bloc d'indices i, j)

$$W_{ij}(f)(x,y) = s \cdot f(v_{ij}^{-1}(x,y)) + q_{ij}(x,y)$$



$$q_{ij}(x,y) = ax + by + cxy + d$$

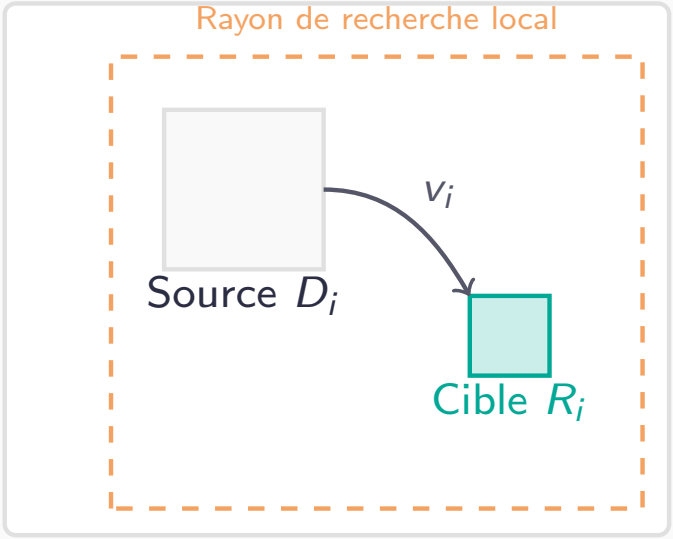
- s est la *rugosité* fractale (fixée globalement).
- q_{ij} est un plan d'ajustement bilinéaire, calculé par un système linéaire 4×4 exact.



Le but : Forcer une continuité absolue C^0 : $W(I) = I$

Équation PIFS (par bloc cible R_i)

$$W_i(f)(x,y) = s_i \cdot f(v_i^{-1}(x,y)) + o_i$$



On minimise l'EQM :
$\min \sum_{(x,y)} (s_i \cdot D_i + o_i - R_i)^2$

Calcul par moindres carrés :

$$s_i = \frac{n \sum R_i D_i - \sum R_i \sum D_i}{n \sum D_i^2 - (\sum D_i)^2} \quad ; \quad o_i = \frac{1}{n}\left(\sum R_i - s_i \sum D_i\right)$$

Le but : Minimiser l'erreur : $W(I) \approx I$

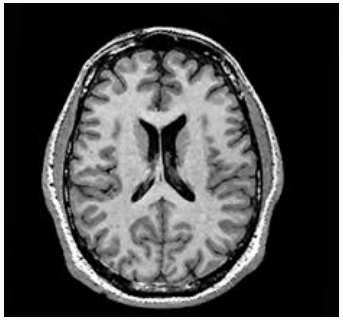


Fig. 5 (Kaggle)
IRM Cérébrale

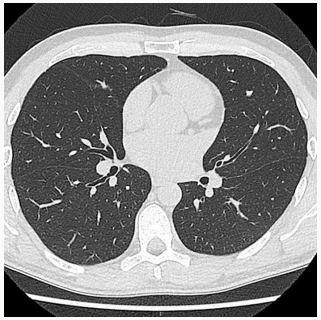


Fig. 6 (Kaggle)
Scanner Pulmonaire

Fig. 7 (Kaggle)
IRM du Rachis

Fidélité Absolue : PSNR

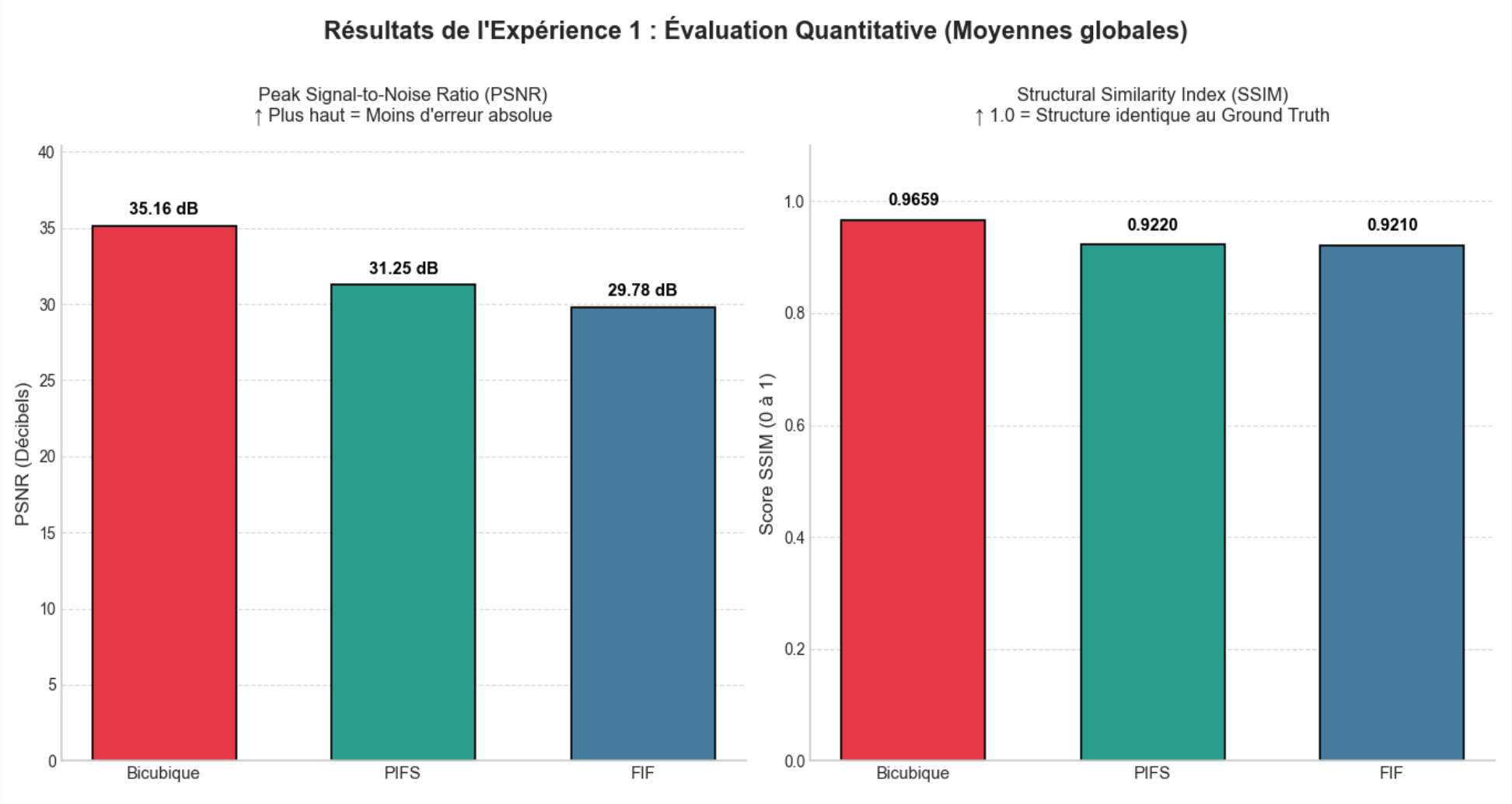
$$\mathrm{PSNR} = 20 \log_{10} \left(\frac{\mathrm{MAX}_I}{\sqrt{\mathrm{EQM}}} \right)$$

MAX_I correspond à la valeur maximale possible du pixel (ex : 255).
L'EQM (Erreur Quadratique Moyenne) est la transposition numérique de notre distance théorique d_2.

Fidélité Clinique : SSIM

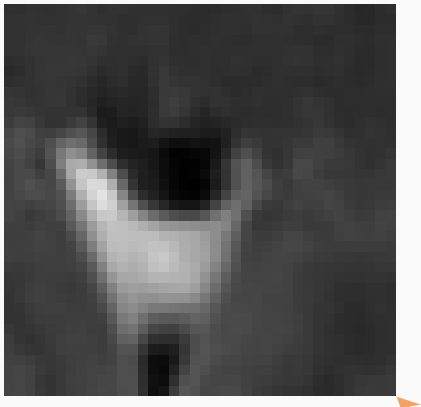
Indice perceptuel complexe évaluant la **luminance**, le **contraste** et la **corrélation spatiale** des structures, s'affranchissant du biais du pixel-à-pixel pour capter la texture tissulaire globale.

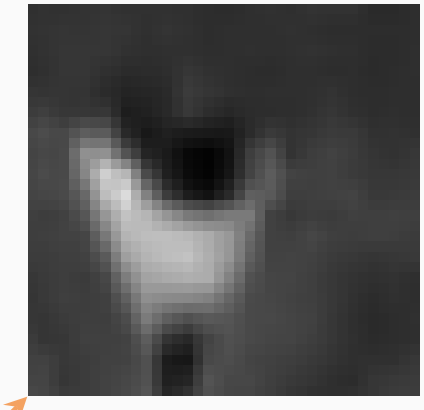
Résultats de l'Expérience 1



1. Vérité Clinique (GT)

3. Approche Algébrique (FIF)





Vérité Clinique (Ground Truth)

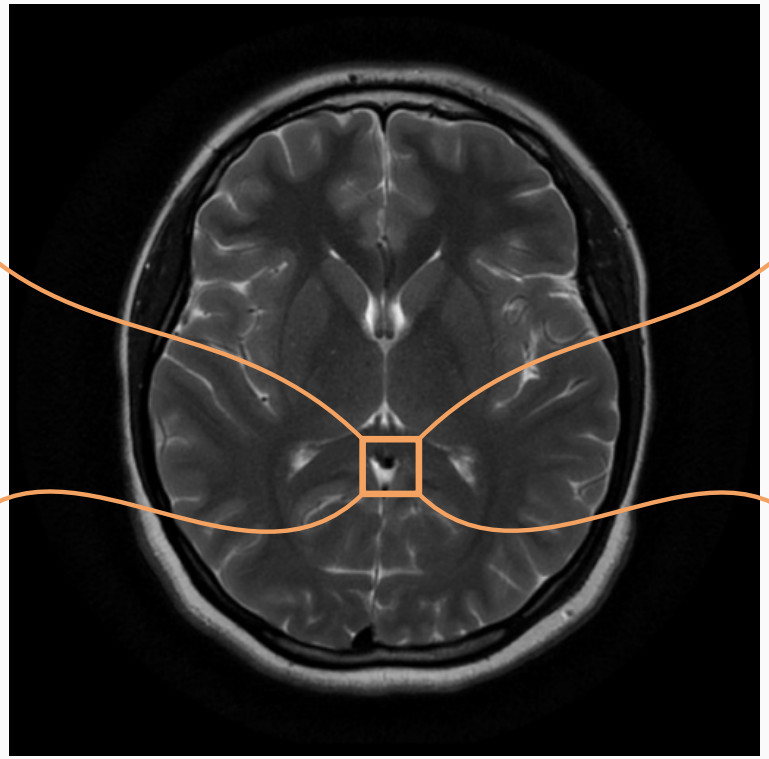
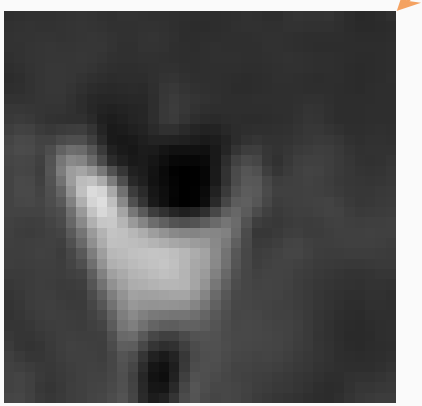
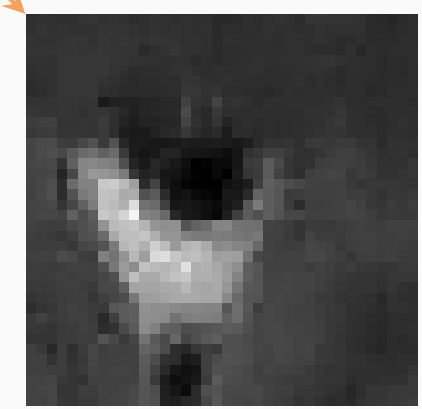


Fig. 8 (Kaggle)

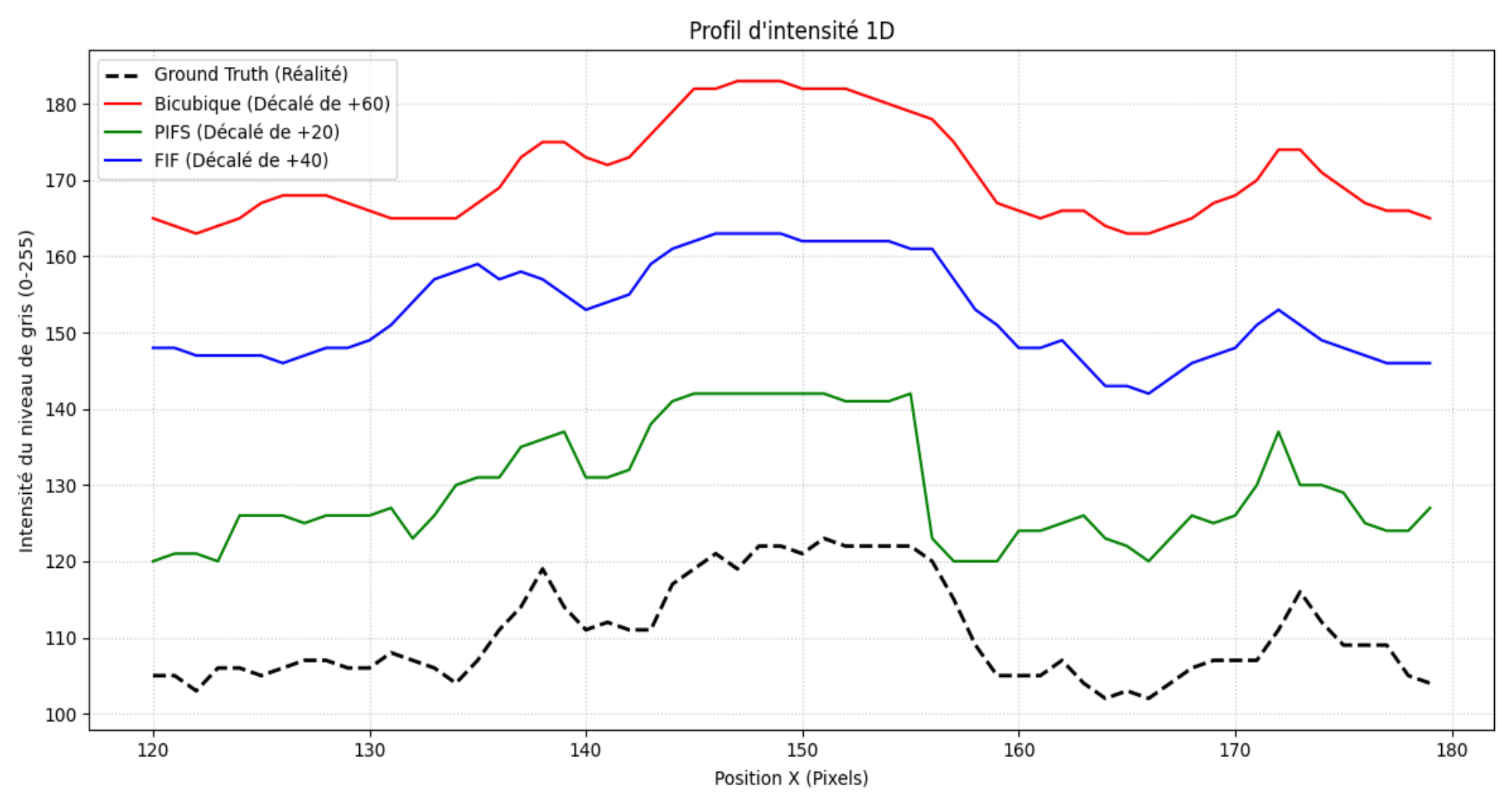




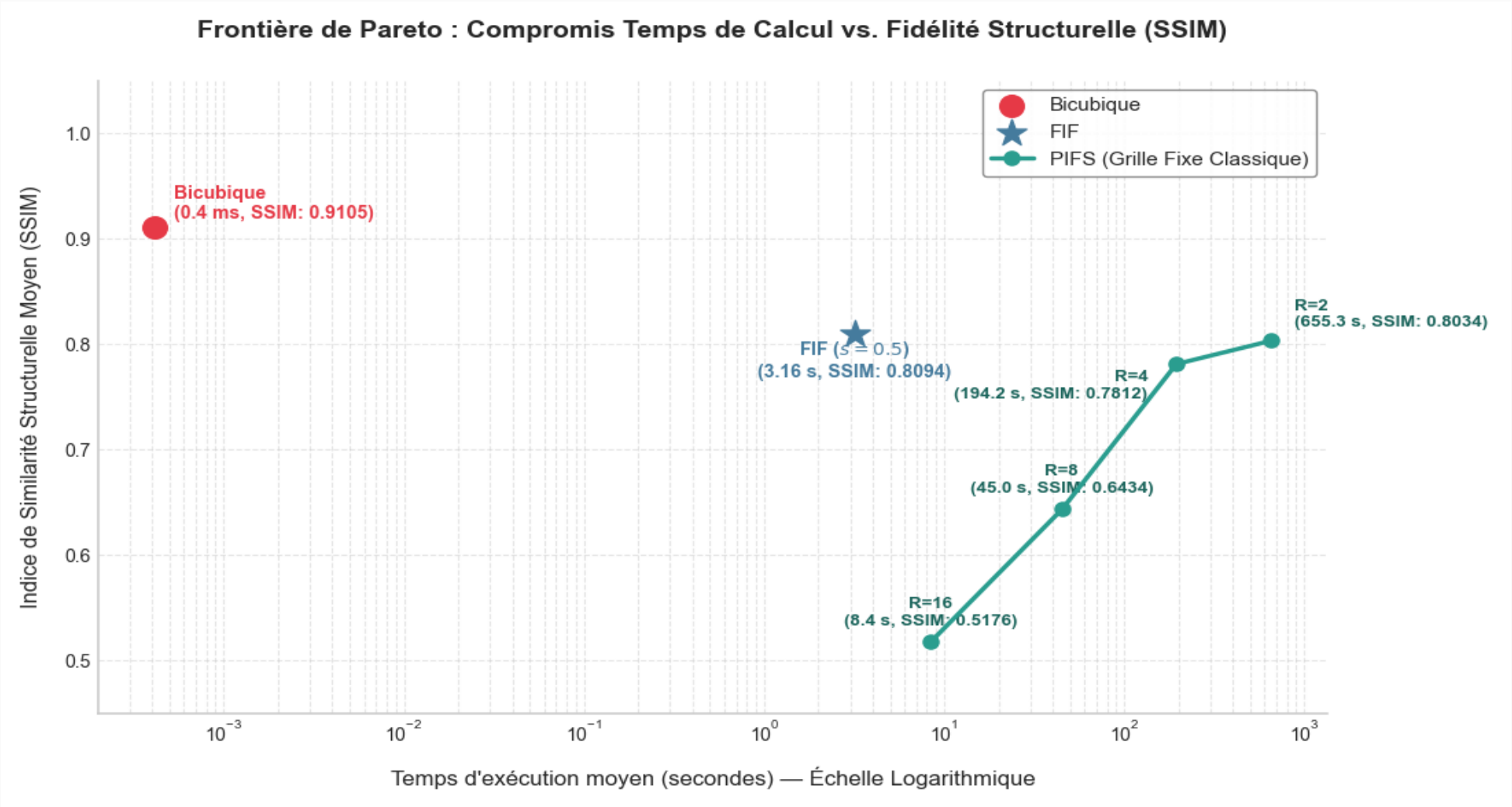
2. Interpolation Bicubique

4. Approche Spatiale (PIFS)

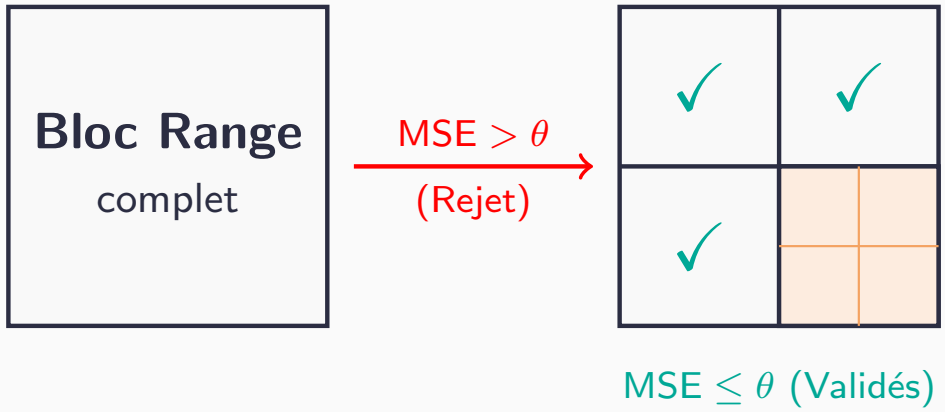
Résultats de l'Expérience 1



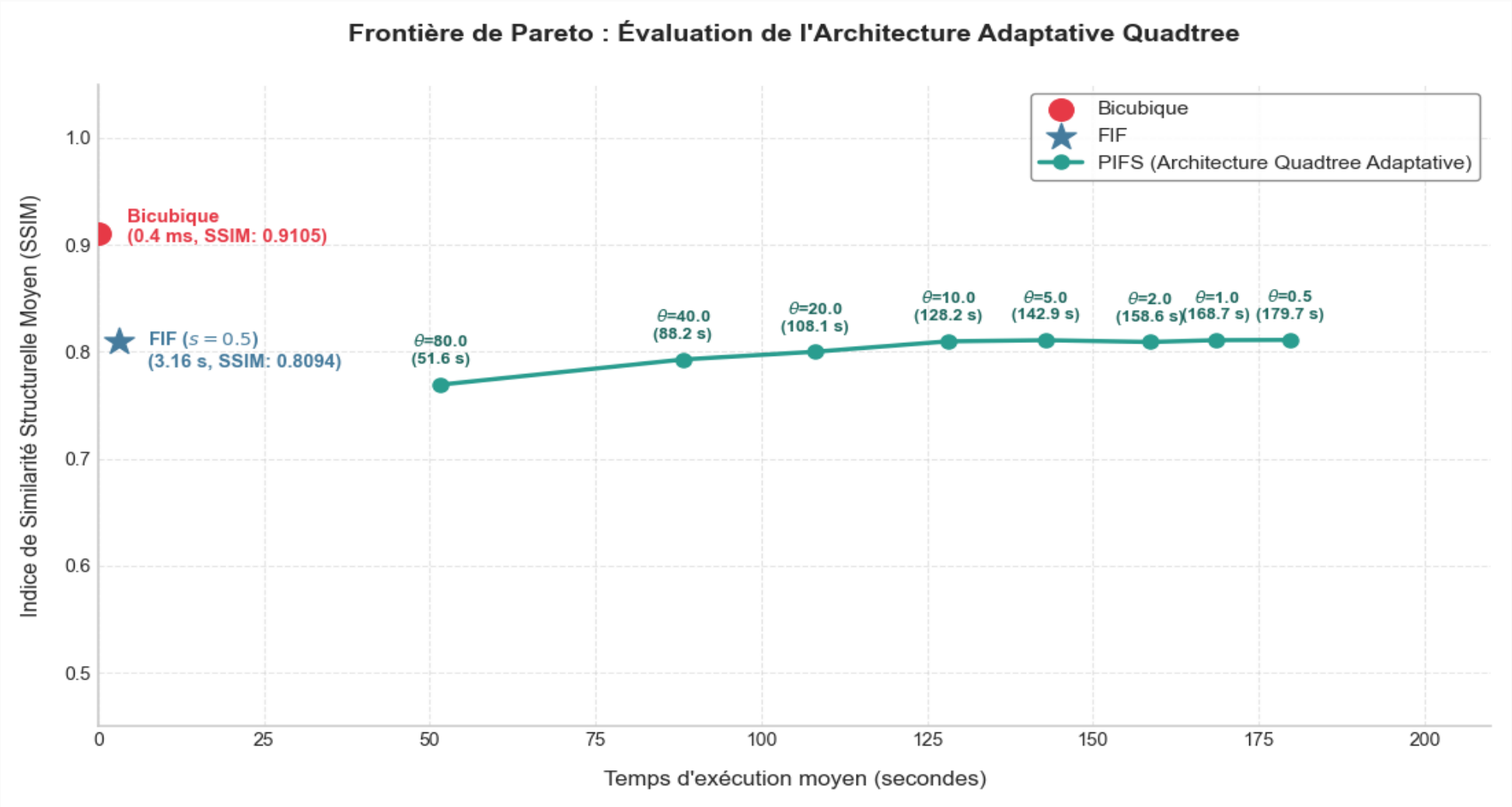
Résultats de l'Expérience 2

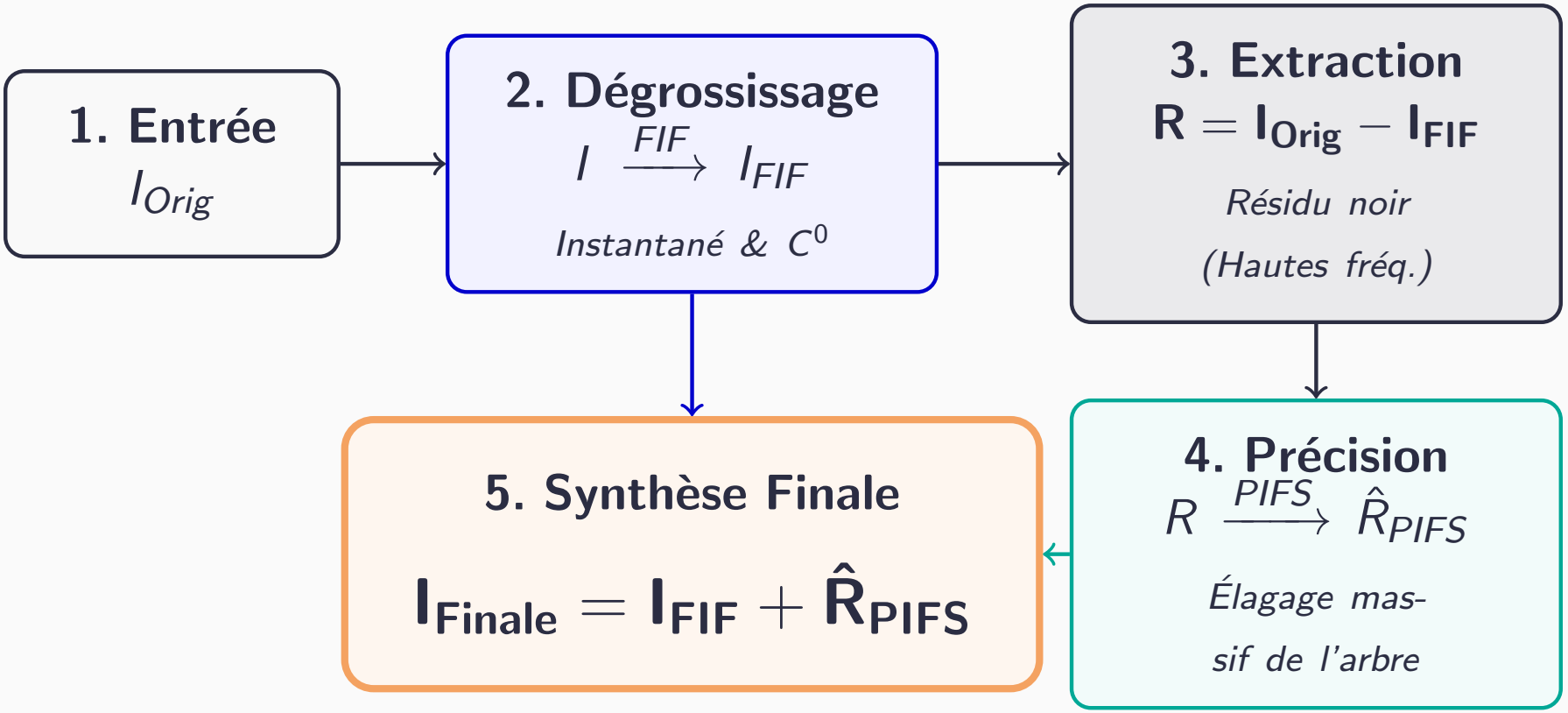


- **Taille dynamique :** Le partitionnement s'adapte à la topologie de l'image.
- **Évaluation de l'erreur :** $\mathrm{MSE}(R_i, W_i(D_i))$ calculée à chaque étape.
- **Seuil de tolérance θ :** Si l'erreur dépasse θ, le bloc est scindé récursivement.



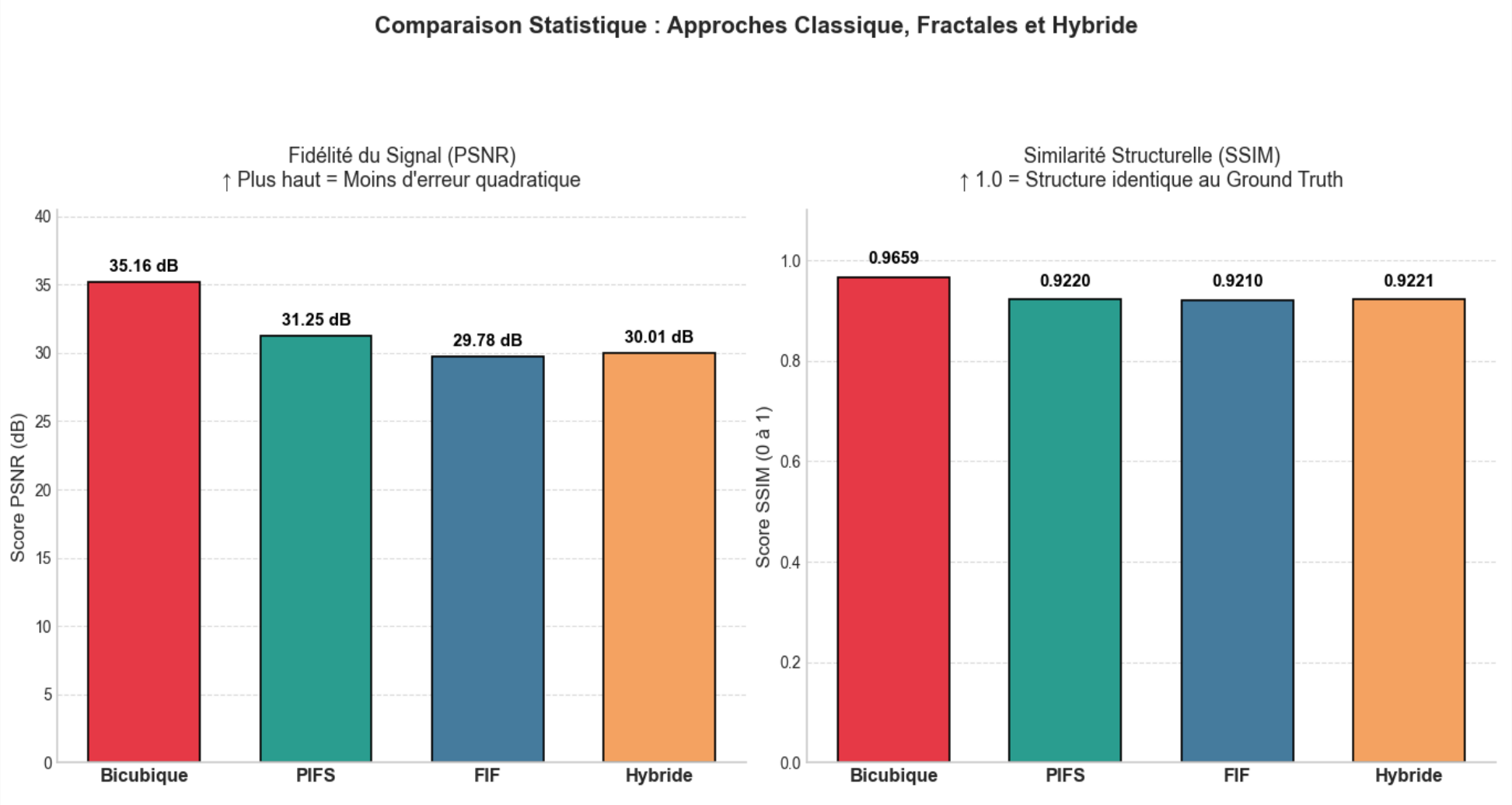
Résultats de l'Expérience 2




1. Entrée
I_{Orig}
2. Dégrossissage
$I \xrightarrow{FIF} I_{FIF}$
Instantané & C^0
3. Extraction
$R = I_{Orig} - I_{FIF}$
Résidu noir
(Hautes fréq.)
4. Précision
$R \xrightarrow{PIFS} \hat{R}_{PIFS}$
Élagage massif de l'arbre
5. Synthèse Finale
$I_{Finale} = I_{FIF} + \hat{R}_{PIFS}$

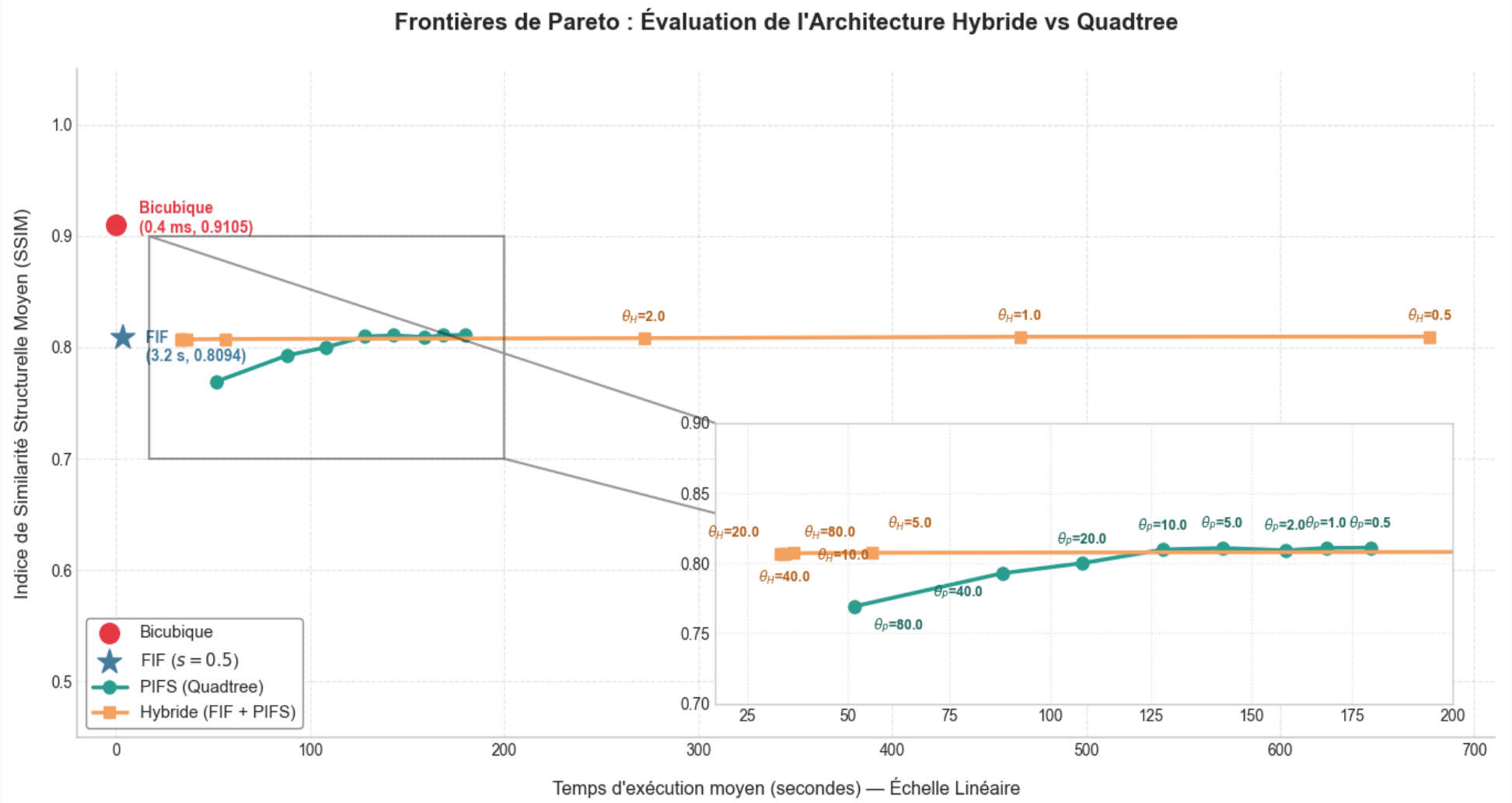
Résultats de Expérience 3/1

Synthèse Comparative des Performances



Résultats de Expérience 3/2

Le Nouvel Optimum de Pareto



Approche Spatiale (PIFS)

Texture excellente, mais coût temporel élevé et artefacts locaux.

VS

Approche Algébrique (FIF)

Vitesse fulgurante, mais lissage excessif de la rugosité clinique.

Synthèse Différentielle (Le Modèle Hybride)

- **Sécurité Topologique** (Structure continue C^0 garantie par le FIF)
- **Précision Clinique** (Rugosité restaurée localement par le PIFS)
- **Viabilité Hospitalière** (Temps optimisé par l'élagage Quadtree)

Merci pour votre attention.

Super-résolution fractale en imagerie médicale

Illustrations Naturelles

Fig. 1 : Chou-fleur Romanesco

https://www.graines-bocquet.fr/blog/post/1131/tout-savoir-sur-le-chou-fleur-romanesco

Fig. 2 : Le cœur fractal et chaotique

https://www.pourlascience.fr/sd/medecine/le-coeur-fractal-et-chaotique-5157.php

Fig. 3 : Fougères à l'ombre

https://lejardindemamere.fr/jardin/fougeres-a-lombre-les-plantes-parfaites-pour-les-zones-difficiles/

Datasets Cliniques (Kaggle)

Fig. 4 & 7 : Spinal Vertebrae Segmentation

Auteur : Unique Data

https://www.kaggle.com/datasets/trainingdatapro/spinal-vertebrae-segmentation

Fig. 5 & 8 : Brain MRI Images for Tumor Detection

Auteur : Navoneel Chakrabarty

https://www.kaggle.com/datasets/navoneel/brain-mri-images-for-brain-tumor-detection

Fig. 6 : CT-Medical-Images

Auteur : Rishan Hasan Tenis

https://www.kaggle.com/datasets/rishantenis/ct-medical-images

Cadre : Espace métrique complet (E, d), opérateur contractant $W : E \to E$ de rapport $k \in [0, 1[$.

Objectif : Prouver l'unicité du point fixe $A \in E$ et la convergence de la suite $x_{n+1} = W(x_n)$.

Preuve de l'Unicité :

Soient deux points fixes A et B (tels que $W(A) = A$ et $W(B) = B$). On a $d(A, B) = d(W(A), W(B)) \leq k \cdot d(A, B)$. Puisque $k < 1$, cette inégalité stricte impose obligatoirement $d(A, B) = 0$, d'où $A = B$.

Preuve de l'Existence (Convergence) :

Pour $x_0 \in E$ arbitraire, on évalue la distance entre itérations successives : $d(x_{n+1}, x_n) \leq \cdots \leq k^n \cdot d(x_1, x_0)$.
Pour prouver que (x_n) est une suite de Cauchy, on évalue $d(x_m, x_n)$ pour $m > n$ par l'inégalité triangulaire :

$$d(x_m, x_n) \leq \sum_{i=n}^{m-1} d(x_{i+1}, x_i) \leq d(x_1, x_0) \sum_{i=n}^{m-1} k^i \leq d(x_1, x_0) \frac{k^n}{1-k}$$

Puisque $\lim_{n\to\infty} k^n = 0$, la suite (x_n) est bien de Cauchy. L'espace E étant complet, elle converge vers $A \in E$.
W étant contractante, elle est continue. Par passage à la limite :
$W(A) = W(\lim_{n\to\infty} x_n) = \lim_{n\to\infty} W(x_n) = A.$

Cadre : On cherche un opérateur W dont l'attracteur A approxime une image cible I.
Objectif : Borner la distance $d(I, A)$ par l'erreur de "collage" $d(I, W(I))$.

Preuve :
D'après l'inégalité triangulaire appliquée à l'image cible I, à sa transformée $W(I)$, et à l'attracteur A, on a :

$$d(I, A) \leq d(I, W(I)) + d(W(I), A)$$

Or, l'attracteur A est le point fixe de W (donc $W(A) = A$), et l'opérateur W est contractant de rapport $k < 1$. Par conséquent :

$$d(W(I), A) = d(W(I), W(A)) \leq k \cdot d(I, A)$$

En réinjectant cette majoration dans la première inégalité, on obtient directement :

$$d(I, A) \leq d(I, W(I)) + k \cdot d(I, A)$$

Puisque $k < 1$, on peut isoler la distance à l'attracteur $d(I, A)$:

$$\boxed{d(I, A) \leq \frac{1}{1-k} \cdot d(I, W(I))}$$

Conclusion : Rendre l'attracteur indiscernable de la cible ($d(I, A) \to 0$) revient algorithmiquement à minimiser l'erreur locale $d(I, W(I))$. C'est le fondement de l'optimisation PIFS/FIF.

Cadre : L'opérateur unifié W agit sur des morceaux indexés i. L'équation locale est :
$(Wf)(x,y) = s_i \cdot f(v_i^{-1}(x,y)) + \text{shift}_i(x,y) \quad$ pour tout $(x,y) \in$ morceau i

Majoration locale (Norme du maximum L^∞) :
Soient $f, g \in E$. Sur un morceau i, la différence absolue s'écrit (les termes shift_i s'annulent) :
$|(Wf) - (Wg)| = |s_i| \cdot |f(v_i^{-1}) - g(v_i^{-1})|$
Prenons le supremum de cette différence **sur le morceau cible** i. Le point $(u,v) = v_i^{-1}(x,y)$ parcourt alors un domaine source D_i :
$\sup_{\text{morceau } i} |Wf - Wg| = |s_i| \cdot \sup_{(u,v)\in D_i} |f(u,v) - g(u,v)|$
Or, le supremum sur le sous-domaine D_i est nécessairement majoré par celui sur l'image entière Ω :
$\sup_{D_i} |f(u,v) - g(u,v)| \leq \sup_{\Omega} |f(u,v) - g(u,v)| = d_\infty(f,g)$
On obtient donc la majoration locale : $\sup_{\text{morceau } i} |Wf - Wg| \leq |s_i| \cdot d_\infty(f,g)$

Contraction globale :
Pour obtenir la distance globale sur l'image entière, on prend le maximum sur l'ensemble des morceaux i :

$$d_\infty(Wf, Wg) = \max_i \left(\sup_{\text{morceau } i} |Wf - Wg| \right) \leq \left(\max_i |s_i| \right) \cdot d_\infty(f,g)$$

Dans l'espace L^∞, W est donc contractant si et seulement si $\forall i, |s_i| < 1$.

Problème métrique : En clinique et dans notre code, la qualité n'est pas mesurée par la norme uniforme L^∞, mais via l'Erreur Quadratique Moyenne (EQM), qui dérive de la métrique L^2. La condition de contraction $|s_i| < 1$ y est-elle toujours valable ?

Dimension Finie :

Une image médicale numérique n'est pas une surface spatiale continue infinie. C'est un objet discret mathématiquement défini par une matrice de pixels $N \times M$.

Notre espace des images E est donc un espace vectoriel isomorphe à $\mathbb{R}^{N \times M}$, qui est de **dimension finie**.

Théorème fondamental :

En topologie, **dans un espace vectoriel de dimension finie, toutes les normes sont équivalentes**.

Conclusion pour l'Algorithme :

Nous avons prouvé la contraction sous L^∞. Par équivalence topologique, cette contrainte agit comme un garde-fou universel. Elle garantit automatiquement que l'opérateur W sera contractant sous L^2. L'algorithme converge donc inconditionnellement vis-à-vis de l'EQM.

Cadre de l'Évaluation Globale

Corpus : 5 images par catégorie (Total : 15 images)
Résolution cible (Ground Truth) : 256×256
Dégradation : Facteur de zoom 2 ($128 \rightarrow 256$)

Exp. 1 : Algorithme PIFS pur

- Quadtree : $16 \rightarrow 2$
- Seuil tolérance θ : 1.0
- Recherche : 32 — Pas : 4

Exp. 1 : Algorithme FIF pur

- Facteur rugosité s : 0.5

Exp. 3 : Synthèse Différentielle (Modèle Hybride)

1. Phase de Dégrossissage (FIF)

- Rugosité maintenue à $s = 0.5$

(Identique à l'expérience 1)

2. Phase de Précision (PIFS)

- Quadtree max réduit à $8 \rightarrow 2$

(Cible directement le résidu fin)

- Seuil strict θ : 0.5

(Garantit la capture des micro-textures)

		Bicubique		PIFS		FIF		Hybride	
Catégorie	**Image**	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM	**PSNR**	**SSIM**
Cerveau	1 no.jpeg	33.77	0.964	29.96	0.916	29.25	0.926	**29.38**	**0.925**
Cerveau	10 no.jpg	39.91	0.982	34.83	0.946	33.59	0.941	**33.90**	**0.943**
Cerveau	11 no.jpg	32.89	0.979	29.07	0.950	28.39	0.951	**28.43**	**0.952**
Cerveau	12 no.jpg	34.53	0.976	29.21	0.920	29.21	0.938	**29.46**	**0.939**
Cerveau	13 no.jpg	33.98	0.974	29.45	0.924	27.68	0.933	**27.85**	**0.927**
Poumon	000108 (3).png	35.58	0.964	32.72	0.923	29.80	0.906	**30.25**	**0.911**
Poumon	000109 (2).png	35.86	0.958	31.69	0.911	30.02	0.911	**30.37**	**0.913**
Poumon	000109 (4).png	35.22	0.956	31.68	0.916	29.46	0.905	**29.84**	**0.908**
Poumon	000109 (5).png	34.41	0.952	31.32	0.911	29.66	0.900	**29.88**	**0.901**
Poumon	000112 (2).png	35.31	0.955	32.07	0.917	30.03	0.905	**30.31**	**0.907**
Rachis	IM000001.jpg	31.65	0.941	27.25	0.865	26.86	0.871	**27.19**	**0.873**
Rachis	IM000002.jpg	33.03	0.951	29.23	0.889	28.09	0.892	**28.41**	**0.894**
Rachis	IM000003.jpg	35.48	0.966	30.92	0.919	29.46	0.918	**29.93**	**0.920**
Rachis	IM000004.jpg	37.73	0.982	34.46	0.960	32.42	0.956	**32.32**	**0.956**
Rachis	IM000005.jpg	38.05	0.982	34.85	0.958	32.65	0.956	**32.55**	**0.955**

Le Défi du Passage à l'Échelle (Haute Résolution)

Corpus : 1 image par catégorie (Total : 3 images)
Résolution cible (Ground Truth) : 512×512
Dégradation massive : Facteur de zoom 4 ($128 \rightarrow 512$)

Exp. 2A : PIFS Classique

Variable de balayage :

- Taille de bloc $R \in \{16, 8, 4, 2\}$

Constantes spatiales :

- Rayon : 32 — Pas : 4

Exp. 2B : PIFS Quadtree

Variable de balayage :

- Seuil $\theta \in [0.5, \ldots, 80.0]$

Constantes spatiales :

- Blocs limités de $16 \rightarrow 2$
- Rayon : 32 — Pas : 4

Le Point de Référence Algébrique

Algorithme FIF pur **fixé** avec un facteur de rugosité $s = 0.5$

Configuration	Cerveau *1 no.jpeg*		Poumon *000108 (3).png*		Rachis *IM000001.jpg*	
	Temps (s)	SSIM	Temps (s)	SSIM	Temps (s)	SSIM
Bicubique	< 0.01	0.931	< 0.01	0.929	< 0.01	0.870
FIF ($s = 0.5$)	**3.01**	**0.857**	**3.13**	0.835	**3.33**	**0.735**
PIFS ($R = 16$)	8.23	0.617	8.70	0.557	8.19	0.378
PIFS ($R = 8$)	42.74	0.688	43.66	0.721	48.57	0.520
PIFS ($R = 4$)	192.32	0.814	198.50	0.836	191.90	0.692
PIFS ($R = 2$)	652.27	0.839	666.78	**0.848**	646.82	0.721

Configuration	Cerveau *1 no.jpeg*		Poumon *000108 (3).png*		Rachis *IM000001.jpg*	
	Temps (s)	SSIM	Temps (s)	SSIM	Temps (s)	SSIM
Bicubique	< 0.01	0.931	< 0.01	0.929	< 0.01	0.870
FIF ($s = 0.5$)	**3.55**	0.857	**5.14**	0.835	**4.54**	0.735
PIFS ($E = 0.5$)	149.96	0.841	178.83	0.857	210.27	0.734
PIFS ($E = 1.0$)	137.72	0.838	167.08	0.859	201.15	0.734
PIFS ($E = 2.0$)	129.63	0.832	155.68	0.861	190.46	0.733
PIFS ($E = 5.0$)	115.90	0.835	132.36	0.864	180.45	0.732
PIFS ($E = 10.0$)	101.24	0.835	110.61	0.862	172.69	0.731
PIFS ($E = 20.0$)	81.72	0.835	88.95	0.839	153.68	0.725
PIFS ($E = 40.0$)	68.80	0.835	68.03	0.828	127.76	0.714
PIFS ($E = 80.0$)	59.93	0.819	37.53	0.801	57.42	0.686

Objectif : Superposition des courbes de Pareto sur le même environnement de test

(Mêmes conditions : Résolution 512 $\times$ 512 • *Dégradation Zoom* $\times 4$ • *3 images cibles)*

Paramétrage du Modèle Hybride pour le Balayage

1. Phase FIF (Ajustée)

- Rugosité abaissée à $s = 0.4$

(Dégrossissage optimisé pour le calcul du résidu)

2. Phase PIFS (Alignement strict)

- Quadtree intégral : $16 \rightarrow 2$

(Garantit une comparaison équitable avec l'Exp. 2B)

Variable de Balayage Temporel & Qualitatif

Seuil de tolérance dynamique pour l'hybride :

$$\theta \in \{0.5, 1.0, 2.0, 5.0, 10.0, 20.0, 40.0, 80.0\}$$

Paramètre	Cerveau *1 no.jpeg*		Poumon *000108 (3).png*		Rachis *IM000001.jpg*	
	Temps (s)	SSIM	Temps (s)	SSIM	Temps (s)	SSIM
0.5	455.76	**0.856**	816.39	**0.837**	758.62	0.735
1.0	192.45	0.855	665.05	0.837	540.05	**0.736**
2.0	31.08	0.852	411.59	0.836	373.47	0.735
5.0	**24.92**	0.852	61.41	0.835	81.59	0.734
10.0	26.25	0.852	40.34	0.835	42.93	0.733
20.0	25.93	0.852	43.32	0.835	**31.73**	0.733
40.0	25.37	0.852	41.65	0.835	35.39	0.733
80.0	30.51	0.852	**35.92**	0.835	33.08	0.733

Configuration Matérielle

- **Processeur (CPU) :**
  Intel Core i5-1035G1
  @ 1.00GHz (Base 1.19GHz)
- **Architecture :**
  4 cœurs physiques / 8 threads
- **Mémoire Vive (RAM) :**
  16 Go

Environnement Logiciel

- **Système d'Exploitation :**
  Windows (64-bit)
- **Interpréteur Core :**
  Python 3.10.11
- **Parallélisme (Multiprocessing) :**
  Distribution inter-images
  (Aucune accélération intra-image)

Protocole de Mesure Temporelle

Le temps de calcul de chaque algorithme est mesuré de manière stricte sur un seul thread CPU.

Le parallélisme n'a servi qu'à traiter le corpus global sans impacter les scores individuels.

```python
import numpy as np
import cv2


def base_plane(pillars, corners, s):
    # interpolation: q(u,v) = au + bv + cuv + d
    # subtract the self-similar contribution first
    z0 = pillars[0] - s * corners[0]
    z1 = pillars[1] - s * corners[1]
    z2 = pillars[2] - s * corners[2]
    z3 = pillars[3] - s * corners[3]

    d = z0
    a = z1 - z0
    b = z2 - z0
    c = z3 - a - b - d
    return a, b, c, d


def roughness(p, max_s=0.5):
    contrast = np.max(p) - np.min(p)
    s = (contrast / 255.0) * max_s
    return np.clip(s, 0.0, 0.99)


def encode_fif(image, max_roughness=0.5):
    h, w = image.shape

    corners = [image[0, 0], image[0, w-1], image[h-1, 0], image[h-1, w-1]]

    # pad by 1 so we can always grab a 2x2 neighborhood
    padded = cv2.copyMakeBorder(image, 0, 1, 0, 1, cv2.BORDER_REPLICATE)

    S_mat = np.zeros((h, w))
    A_mat = np.zeros((h, w))
    B_mat = np.zeros((h, w))
    C_mat = np.zeros((h, w))
    D_mat = np.zeros((h, w))

    for y in range(h):
        for x in range(w):
            p = [padded[y, x], padded[y, x+1], padded[y+1, x], padded[y+1, x+1]]

            s = roughness(p, max_s=max_roughness)
            a, b, c, d = base_plane(p, corners, s)

            S_mat[y, x] = s
            A_mat[y, x] = a
            B_mat[y, x] = b
            C_mat[y, x] = c
            D_mat[y, x] = d

    return {
        'metadata': {'original_shape': (h, w), 'max_roughness': max_roughness},
        'matrices': {'s': S_mat, 'a': A_mat, 'b': B_mat, 'c': C_mat, 'd': D_mat}
    }


```

Implémentation FIF

PYTHON

```python
def decode_fif(fractal_data, iterations=8, zoom_factor=2):
    orig_h, orig_w = fractal_data['metadata']['original_shape']
    m = fractal_data['matrices']

    h = orig_h * zoom_factor
    w = orig_w * zoom_factor

    img = np.ones((h, w)) * 128.0

    # sub-pixel grid within each zoomed cell
    u = np.linspace(0, 1, zoom_factor, endpoint=False)
    v = np.linspace(0, 1, zoom_factor, endpoint=False)
    U, V = np.meshgrid(u, v)

    for _ in range(iterations):
        nxt = np.zeros((h, w))

        # global attractor shrunk to zoom_factor x zoom_factor
        g = cv2.resize(img, (zoom_factor, zoom_factor), interpolation=cv2.INTER_AREA)

        for y in range(orig_h):
            for x in range(orig_w):
                s = m['s'][y, x]
                a = m['a'][y, x]
                b = m['b'][y, x]
                c = m['c'][y, x]
                d = m['d'][y, x]

                q = a*U + b*V + c*(U*V) + d
                block = s * g + q

                nxt[y*zoom_factor:(y+1)*zoom_factor, x*zoom_factor:(x+1)*zoom_factor] = block

        img = nxt

    return np.clip(img, 0, 255).astype(np.uint8)
```

```python
import numpy as np
import cv2


def isometries(block):
    variants = []
    for k in range(4):
        variants.append(np.rot90(block, k))
    flipped = np.fliplr(block)
    for k in range(4):
        variants.append(np.rot90(flipped, k))
    return variants


def fit_affine(R, D):
    # least squares fit for s and o such that s*D + o ~ R
    # with Banach contractivity enforced: |s| < 1
    R_f = R.astype(np.float64).flatten()
    D_f = D.astype(np.float64).flatten()
    n = len(R_f)
    sR = np.sum(R_f)
    sD = np.sum(D_f)
    denom = n * np.sum(D_f**2) - sD**2
    if denom == 0:
        s = 0.0
    else:
        s = (n * np.sum(R_f * D_f) - sR * sD) / denom
    s = np.clip(s, -0.99, 0.99)
    o = (sR - s * sD) / n
    return s, o


def encode_pifs(image, range_size=8, search_radius=32, domain_step=4):
    h, w = image.shape
    dom_size = range_size * 2
    transforms = []

    for ry in range(0, h, range_size):
        for rx in range(0, w, range_size):
            R = image[ry:ry+range_size, rx:rx+range_size]

            best_err = float('inf')
            best = None

            y0 = max(0, ry - search_radius)
            y1 = min(h - dom_size, ry + search_radius)
            x0 = max(0, rx - search_radius)
            x1 = min(w - dom_size, rx + search_radius)

            for dy in range(y0, y1, domain_step):
                for dx in range(x0, x1, domain_step):
                    D_large = image[dy:dy+dom_size, dx:dx+dom_size]
                    D_small = cv2.resize(D_large, (range_size, range_size), interpolation=cv2.INTER_AREA)

                    for idx, D_iso in enumerate(isometries(D_small)):
```

```
                        s, o = fit_affine(R, D_iso)
                        err = np.mean((R - (s * D_iso + o))**2)

                        if err < best_err:
                            best_err = err
                            best = {
                                'rx': rx, 'ry': ry,
                                'dx': dx, 'dy': dy,
                                'iso': idx,
                                's': s, 'o': o
                            }

            transforms.append(best)

    return {
        'metadata': {
            'original_shape': (h, w),
            'range_size': range_size,
            'search_radius': search_radius,
            'domain_step': domain_step
        },
        'transformations': transforms
    }


def decode_pifs(fractal_data, iterations=8, zoom_factor=1):
    orig_h, orig_w = fractal_data['metadata']['original_shape']
    bsz = fractal_data['metadata']['range_size']
    transforms = fractal_data['transformations']
    h = orig_h * zoom_factor
    w = orig_w * zoom_factor
    rsz = bsz * zoom_factor
    dsz = rsz * 2
    canvas = np.ones((h, w)) * 128.0

    for _ in range(iterations):
        nxt = np.zeros((h, w))

        for t in transforms:
            rx = t['rx'] * zoom_factor
            ry = t['ry'] * zoom_factor
            dx = t['dx'] * zoom_factor
            dy = t['dy'] * zoom_factor

            D_large = canvas[dy:dy+dsz, dx:dx+dsz]
            D_small = cv2.resize(D_large, (rsz, rsz), interpolation=cv2.INTER_AREA)
            D_iso = isometries(D_small)[t['iso']]

            nxt[ry:ry+rsz, rx:rx+rsz] = t['s'] * D_iso + t['o']

        canvas = nxt

    return np.clip(canvas, 0, 255).astype(np.uint8)
```

```python
import numpy as np
import cv2


def isometries(block):
    variants = []
    for k in range(4):
        variants.append(np.rot90(block, k))
    flipped = np.fliplr(block)
    for k in range(4):
        variants.append(np.rot90(flipped, k))
    return variants


def build_pools(image, step, sizes):
    # precompute downsampled domain blocks and their scalar stats
    # so the inner loop can use dot products instead of materializing arrays
    h, w = image.shape
    pools = {}

    for S in sizes:
        D_size = S * 2
        pool = []

        for y in range(0, h - D_size + 1, step):
            for x in range(0, w - D_size + 1, step):
                dom = image[y:y+D_size, x:x+D_size]
                dom_small = cv2.resize(dom, (S, S), interpolation=cv2.INTER_AREA).astype(np.float64)
                flat = dom_small.ravel()

                sD  = np.sum(flat)
                sD2 = np.dot(flat, flat)

                isos = [iso.ravel() for iso in isometries(dom_small)]
                pool.append((x, y, isos, sD, sD2))

        pools[S] = pool

    return pools


def encode_pifs(image, max_size=16, min_size=2, threshold=5.0, search_radius=8, domain_step=4):
    h, w = image.shape
    image = image.astype(np.float64)

    sizes = []
    s = max_size
    while s >= min_size:
        sizes.append(s)
        s //= 2

    print(f"[PIFS] sizes={sizes}, threshold={threshold}")
    pools = build_pools(image, domain_step, sizes)
```

```
    blocks = []

    def split(x, y, size):
        R_flat = image[y:y+size, x:x+size].ravel()
        n = size * size
        sR  = np.sum(R_flat)
        sR2 = np.dot(R_flat, R_flat)

        best_err = float('inf')
        best = None

        for dx, dy, isos, sD, sD2 in pools[size]:
            if search_radius is not None:
                if abs(dx - x) > search_radius or abs(dy - y) > search_radius:
                    continue

            for dir_idx, D_flat in enumerate(isos):
                sRD = np.dot(R_flat, D_flat)

                denom = n * sD2 - sD**2
                if denom == 0:
                    s_val = 0.0
                else:
                    s_val = (n * sRD - sR * sD) / denom

                s_val = np.clip(s_val, -0.99, 0.99)
                o_val = (sR - s_val * sD) / n

                # scalar SSE — avoids allocating the residual array
                sse = sR2 - 2*s_val*sRD - 2*o_val*sR + s_val**2*sD2 + 2*s_val*o_val*sD + n*o_val**2
                err = max(0.0, sse / n)

                if err < best_err:
                    best_err = err
                    best = {'rx': x, 'ry': y, 'size': size,
                            'dx': dx, 'dy': dy, 'dir': dir_idx,
                            's': s_val, 'o': o_val}

        # fallback if nothing passed the radius filter (can happen at borders)
        if best is None:
            sdx = max(0, min(x, w - size*2))
            sdy = max(0, min(y, h - size*2))
            best = {'rx': x, 'ry': y, 'size': size,
                    'dx': sdx, 'dy': sdy, 'dir': 0,
                    's': 0.0, 'o': sR / n}

        if best_err > threshold and size > min_size:
            half = size // 2
            split(x,        y,        half)
            split(x + half, y,        half)
            split(x,        y + half, half)
            split(x + half, y + half, half)
```

```python
        else:
            blocks.append(best)

    for y in range(0, h, max_size):
        for x in range(0, w, max_size):
            split(x, y, max_size)

    print(f"[PIFS] done, {len(blocks)} blocks")
    return {
        'metadata': {
            'original_shape': (h, w),
            'quadtree_max': max_size,
            'quadtree_min': min_size,
            'threshold_used': threshold
        },
        'transformations': blocks
    }


def decode_pifs(fractal_data, iterations=8, zoom_factor=1):
    orig_h, orig_w = fractal_data['metadata']['original_shape']
    blocks = fractal_data['transformations']

    h = orig_h * zoom_factor
    w = orig_w * zoom_factor

    canvas = np.ones((h, w), dtype=np.float64) * 128.0

    for _ in range(iterations):
        nxt = np.zeros((h, w), dtype=np.float64)


        for b in blocks:
            rx = b['rx'] * zoom_factor
            ry = b['ry'] * zoom_factor
            rsz = b['size'] * zoom_factor

            dx = b['dx'] * zoom_factor
            dy = b['dy'] * zoom_factor
            dsz = b['size'] * 2 * zoom_factor

            D_large  = canvas[dy:dy+dsz, dx:dx+dsz]
            D_small  = cv2.resize(D_large, (rsz, rsz), interpolation=cv2.INTER_AREA)
            D_trans  = isometries(D_small)[b['dir']]

            nxt[ry:ry+rsz, rx:rx+rsz] = b['s'] * D_trans + b['o']

        canvas = nxt

    return np.clip(canvas, 0, 255).astype(np.uint8)
```

```python
import numpy as np
import cv2
import time


# ------------------------------------------------
# FIF internals (float64 throughout)
# ------------------------------------------------

def _fif_base_plane(P, C, s):
    d = P[0] - s * C[0]
    a = (P[1] - s * C[1]) - d
    b = (P[2] - s * C[2]) - d
    c = (P[3] - s * C[3]) - a - b - d
    return a, b, c, d


def _fif_encode(img, max_roughness=0.5):
    h, w = img.shape
    S = np.zeros((h, w), dtype=np.float64)
    A = np.zeros((h, w), dtype=np.float64)
    B = np.zeros((h, w), dtype=np.float64)
    CC = np.zeros((h, w), dtype=np.float64)
    D = np.zeros((h, w), dtype=np.float64)

    corners = [img[0, 0], img[0, w-1], img[h-1, 0], img[h-1, w-1]]

    for y in range(h):
        for x in range(w):
            yn = min(y+1, h-1)
            xn = min(x+1, w-1)
            p = [img[y, x], img[y, xn], img[yn, x], img[yn, xn]]

            contrast = np.max(p) - np.min(p)
            rough = (contrast / 255.0) * max_roughness

            a, b, c, d = _fif_base_plane(p, corners, rough)
            S[y, x] = rough
            A[y, x] = a
            B[y, x] = b
            CC[y, x] = c
            D[y, x] = d

    return {'meta': {'shape': (h, w)}, 'mats': {'s': S, 'a': A, 'b': B, 'c': CC, 'd': D}}


def _fif_decode(data, iters=8, zoom=2):
    orig_h, orig_w = data['meta']['shape']
    m = data['mats']
    h = orig_h * zoom
    w = orig_w * zoom
    canvas = np.ones((h, w), dtype=np.float64) * 128.0
    u = np.linspace(0, 1, zoom, endpoint=False)
    v = np.linspace(0, 1, zoom, endpoint=False)
    U, V = np.meshgrid(u, v)
    for _ in range(iters):
        nxt = np.zeros((h, w), dtype=np.float64)
        # cv2.resize needs float32
```

```
        g = cv2.resize(canvas.astype(np.float32), (zoom, zoom), interpolation=cv2.INTER_AREA).astype(np.float64)
        for y in range(orig_h):
            for x in range(orig_w):
                s  = m['s'][y, x]
                a  = m['a'][y, x]
                b  = m['b'][y, x]
                c  = m['c'][y, x]
                d  = m['d'][y, x]
                base = a*U + b*V + c*U*V + d
                nxt[y*zoom:(y+1)*zoom, x*zoom:(x+1)*zoom] = base + s * g

        canvas = nxt

    return canvas


# ------------------------------------------------------------
# PIFS internals (float64, supports negative residuals)
# ------------------------------------------------------------

def _isometries(block):
    variants = []
    for k in range(4):
        variants.append(np.rot90(block, k))
    flipped = np.fliplr(block)
    for k in range(4):
        variants.append(np.rot90(flipped, k))
    return variants



def _build_pools(img, step, sizes):
    h, w = img.shape
    pools = {}
    for S in sizes:
        dsz = S * 2
        pool = []
        for y in range(0, h - dsz + 1, step):
            for x in range(0, w - dsz + 1, step):
                dom = img[y:y+dsz, x:x+dsz]
                dom_small = cv2.resize(dom.astype(np.float32), (S, S), interpolation=cv2.INTER_AREA).astype(np.float64)
                pool.append((x, y, dom_small))
        pools[S] = pool
    return pools


def _pifs_encode_qt(residual, max_size, min_size, threshold, search_radius, domain_step):
    sizes = [max_size // (2**i) for i in range(int(np.log2(max_size / min_size)) + 1)]
    pools = _build_pools(residual, domain_step, sizes)
    transforms = []

    def split(x, y, size):
        R = residual[y:y+size, x:x+size]
        best_err = float('inf')
        best = None
        for dx, dy, D_dom in pools[size]:
```

```
            if abs(dx - x) > search_radius or abs(dy - y) > search_radius:
                continue

            for dir_idx, D_t in enumerate(_isometries(D_dom)):
                D_mean = np.mean(D_t)
                D_var  = np.var(D_t)
                R_mean = np.mean(R)

                if D_var < 1e-6:
                    s_val = 0.0
                    o_val = float(R_mean)
                else:
                    cov   = np.cov(R.flatten(), D_t.flatten())[0, 1]
                    s_val = np.clip(cov / D_var, -1.0, 1.0)
                    o_val = float(R_mean - s_val * D_mean)

                err = np.mean((R - (s_val * D_t + o_val))**2)

                if err < best_err:
                    best_err = err
                    best = {'dx': dx, 'dy': dy, 'dir': dir_idx, 's': float(s_val), 'o': o_val}

        if best_err <= threshold or size <= min_size:
            transforms.append({'rx': x, 'ry': y, 'size': size, **best})
        else:
            half = size // 2
            split(x,        y,        half)
            split(x + half, y,        half)
            split(x,        y + half, half)
            split(x + half, y + half, half)

    h, w = residual.shape
    for y in range(0, h, max_size):
        for x in range(0, w, max_size):
            split(x, y, max_size)

    return transforms


def _pifs_decode(transforms, orig_shape, iters=8, zoom=2):
    orig_h, orig_w = orig_shape
    h = orig_h * zoom
    w = orig_w * zoom
    # residual starts at 0, not 128
    canvas = np.zeros((h, w), dtype=np.float64)
    for _ in range(iters):
        nxt = np.zeros((h, w), dtype=np.float64)
        for t in transforms:
            rx   = t['rx']   * zoom
            ry   = t['ry']   * zoom
            rsz  = t['size'] * zoom
            dx   = t['dx']   * zoom
            dy   = t['dy']   * zoom
            dsz  = t['size'] * 2 * zoom
            D_large  = canvas[dy:dy+dsz, dx:dx+dsz]
```

```
            D_small  = cv2.resize(D_large.astype(np.float32), (rsz, rsz), interpolation=cv2.INTER_AREA).astype(np.float64)
            D_trans  = _isometries(D_small)[t['dir']]
            nxt[ry:ry+rsz, rx:rx+rsz] = t['s'] * D_trans + t['o']

        canvas = nxt

    return canvas


# ------------------------------------------------------------
# Public API
# ------------------------------------------------------------

def encode_hybrid(image, fif_roughness=0.5, pifs_max=16, pifs_min=2, pifs_theta=3.0):
    t0 = time.perf_counter()
    print("[hybrid] step 1: FIF base layer")
    img_f = image.astype(np.float64)
    fif_data   = _fif_encode(img_f, max_roughness=fif_roughness)
    fif_lr     = _fif_decode(fif_data, iters=8, zoom=1)
    residual   = img_f - fif_lr
    print("[hybrid] step 2: PIFS residual layer")
    pifs_data  = _pifs_encode_qt(
        residual,
        max_size=pifs_max, min_size=pifs_min,
        threshold=pifs_theta,
        search_radius=32, domain_step=4
    )

    elapsed = time.perf_counter() - t0
    print(f"[hybrid] done in {elapsed:.2f}s, {len(pifs_data)} residual blocks")

    return {
        'fif_data':  fif_data,
        'pifs_data': pifs_data,
        'metadata':  {'original_shape': img_f.shape}
    }


def decode_hybrid(hybrid_data, iterations=8, zoom_factor=2):
    print(f"[hybrid] decoding at zoom x{zoom_factor}")
    orig_shape = hybrid_data['metadata']['original_shape']
    fif_hr  = _fif_decode(hybrid_data['fif_data'],  iters=iterations, zoom=zoom_factor)
    pifs_hr = _pifs_decode(hybrid_data['pifs_data'], orig_shape, iters=iterations, zoom=zoom_factor)
    result = np.clip(fif_hr + pifs_hr, 0, 255).astype(np.uint8)
    print("[hybrid] done")
    return result
```

```python
import os
import cv2
import numpy as np
import csv
import time
from concurrent.futures import ProcessPoolExecutor

from skimage.metrics import structural_similarity as calculate_ssim
from skimage.metrics import peak_signal_noise_ratio as calculate_psnr

SAMPLE_SIZE_PER_CAT = 5
TEST_RESOLUTION = 256
ZOOM_TARGET = 2

# PIFS quadtree parameters
PIFS_MAX_SIZE = 16
PIFS_MIN_SIZE = 2
PIFS_THRESHOLD = 0.5
PIFS_SEARCH_RADIUS = 64

FIF_ROUGHNESS = 0.5

DATASET_BASE_PATH = r"C:\Users\hp\Desktop\TIPE\Datasets"
OUTPUT_CSV = "experiment_1_results_new_new.csv"
OUTPUT_DIR = "exp1_visual_proofs_new_new"


def get_images(folder, n):
    extensions = ('.jpg', '.jpeg', '.png')
    found = []
    for root, dirs, files in os.walk(folder):
        for f in files:
            if f.lower().endswith(extensions):
                found.append(os.path.join(root, f))
                if len(found) == n:
                    return found
    return found


def process_single_image(args):
    category, img_path, idx, pifs_max, pifs_min, pifs_thresh, pifs_radius, fif_roughness, res, zoom, output_dir = args
    filename = os.path.basename(img_path)
    img_raw = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    img_gt = cv2.resize(img_raw, (res, res), interpolation=cv2.INTER_AREA)
    low_res = res // zoom
    img_degraded = cv2.resize(img_gt, (low_res, low_res), interpolation=cv2.INTER_AREA)
    img_bicubic = cv2.resize(img_degraded, (res, res), interpolation=cv2.INTER_CUBIC)

    import PIFS
    import FIF

    encoded_pifs = PIFS.encode_pifs(
        img_degraded,
        max_size=pifs_max,
        min_size=pifs_min,
        threshold=pifs_thresh,
        search_radius=pifs_radius
```

```
    )
    img_pifs = PIFS.decode_pifs(encoded_pifs, iterations=8, zoom_factor=zoom)

    encoded_fif = FIF.encode_fif(img_degraded, max_roughness=fif_roughness)
    img_fif = FIF.decode_fif(encoded_fif, iterations=8, zoom_factor=zoom)

    psnr_bic = calculate_psnr(img_gt, img_bicubic, data_range=255)
    ssim_bic = calculate_ssim(img_gt, img_bicubic, data_range=255)

    psnr_pifs = calculate_psnr(img_gt, img_pifs, data_range=255)
    ssim_pifs = calculate_ssim(img_gt, img_pifs, data_range=255)

    psnr_fif = calculate_psnr(img_gt, img_fif, data_range=255)
    ssim_fif = calculate_ssim(img_gt, img_fif, data_range=255)

    if idx == 0:
        os.makedirs(output_dir, exist_ok=True)
        cv2.imwrite(os.path.join(output_dir, f"{category}_1_GroundTruth.png"), img_gt)
        cv2.imwrite(os.path.join(output_dir, f"{category}_2_Degraded.png"), img_degraded)
        cv2.imwrite(os.path.join(output_dir, f"{category}_3_Bicubic.png"), img_bicubic)
        cv2.imwrite(os.path.join(output_dir, f"{category}_4_PIFS.png"), img_pifs)
        cv2.imwrite(os.path.join(output_dir, f"{category}_5_FIF.png"), img_fif)

    return [category, filename, psnr_bic, ssim_bic, psnr_pifs, ssim_pifs, psnr_fif, ssim_fif]


def run_experiment():
    print("Démarrage de l'expérience 1...")
    print(f"Dataset : {DATASET_BASE_PATH}")
    print(f"Chargement de {SAMPLE_SIZE_PER_CAT} images par catégorie...\n")

    categories = {
        "Brain": get_images(os.path.join(DATASET_BASE_PATH, "brainMRI"), SAMPLE_SIZE_PER_CAT),
        "Lung":  get_images(os.path.join(DATASET_BASE_PATH, "lungCT"),   SAMPLE_SIZE_PER_CAT),
        "Spine": get_images(os.path.join(DATASET_BASE_PATH, "spineXray"), SAMPLE_SIZE_PER_CAT)
    }

    for cat, paths in categories.items():
        print(f"  {cat} : {len(paths)} images trouvées")

    tasks = []
    for category, image_paths in categories.items():
        for idx, img_path in enumerate(image_paths):
            tasks.append((
                category, img_path, idx,
```

```
                PIFS_MAX_SIZE, PIFS_MIN_SIZE, PIFS_THRESHOLD, PIFS_SEARCH_RADIUS,
                FIF_ROUGHNESS, TEST_RESOLUTION, ZOOM_TARGET, OUTPUT_DIR
            ))

    print(f"\nTraitement de {len(tasks)} images en parallèle...")
    start_time = time.time()

    csv_data = []
    headers = ["Category", "Filename", "Bicubic_PSNR", "Bicubic_SSIM", "PIFS_PSNR", "PIFS_SSIM", "FIF_PSNR", "FIF_SSIM"]
    stats = {
        "Bicubic": {"psnr": [], "ssim": []},
        "PIFS":    {"psnr": [], "ssim": []},
        "FIF":     {"psnr": [], "ssim": []}
    }

    with ProcessPoolExecutor() as executor:
        for res in executor.map(process_single_image, tasks):
            csv_data.append(res)
            print(f"  {res[1]} ({res[0]}) — terminé")

            stats["Bicubic"]["psnr"].append(res[2])
            stats["Bicubic"]["ssim"].append(res[3])
            stats["PIFS"]["psnr"].append(res[4])
            stats["PIFS"]["ssim"].append(res[5])
            stats["FIF"]["psnr"].append(res[6])
            stats["FIF"]["ssim"].append(res[7])

    elapsed = time.time() - start_time
    print(f"\nCalculs terminés en {elapsed:.2f}s\n")

    print("Résultats (moyennes globales)")
    print("-" * 40)
    for algo in ["Bicubic", "PIFS", "FIF"]:
        avg_psnr = np.mean(stats[algo]["psnr"])
        avg_ssim = np.mean(stats[algo]["ssim"])
        print(f"  {algo}")
        print(f"    PSNR : {avg_psnr:.2f} dB")
        print(f"    SSIM : {avg_ssim:.4f}")
    print("-" * 40)

    with open(OUTPUT_CSV, mode='w', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(headers)
        writer.writerows(csv_data)

    print(f"\nDonnées exportées dans '{OUTPUT_CSV}'")
    print(f"Images sauvegardées dans '{OUTPUT_DIR}'")


if __name__ == "__main__":
    run_experiment()
```

```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

CSV_FILE = "experiment_1_results.csv"

COLORS = ['#E63946', '#2A9D8F', '#457B9D']
LABELS = ['Bicubique', 'PIFS', 'FIF']


def generate_bar_charts(csv_path):
    df = pd.read_csv(csv_path)

    psnr_means = [df['Bicubic_PSNR'].mean(), df['PIFS_PSNR'].mean(), df['FIF_PSNR'].mean()]
    ssim_means = [df['Bicubic_SSIM'].mean(), df['PIFS_SSIM'].mean(), df['FIF_SSIM'].mean()]

    plt.style.use('seaborn-v0_8-whitegrid')
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))
    fig.suptitle("Résultats de l'Expérience 1 : Évaluation Quantitative (Moyennes globales)",
                 fontsize=16, fontweight='bold')

    bars1 = ax1.bar(LABELS, psnr_means, color=COLORS, edgecolor='black', linewidth=1.2, width=0.6)
    ax1.set_title("Peak Signal-to-Noise Ratio (PSNR)\n↑ Plus haut = Moins d'erreur absolue", fontsize=12, pad=15)
    ax1.set_ylabel("PSNR (Décibels)", fontsize=12)
    ax1.set_ylim(0, max(psnr_means) * 1.15)
    for bar in bars1:
        yval = bar.get_height()
        ax1.text(bar.get_x() + bar.get_width() / 2, yval + 0.5,
                 f"{yval:.2f} dB", ha='center', va='bottom', fontsize=11, fontweight='bold')

    bars2 = ax2.bar(LABELS, ssim_means, color=COLORS, edgecolor='black', linewidth=1.2, width=0.6)
    ax2.set_title("Structural Similarity Index (SSIM)\n↑ 1.0 = Structure identique au Ground Truth", fontsize=12, pad=15)
    ax2.set_ylabel("Score SSIM (0 à 1)", fontsize=12)
    ax2.set_ylim(0, 1.1)
    for bar in bars2:
        yval = bar.get_height()
        ax2.text(bar.get_x() + bar.get_width() / 2, yval + 0.02,
                 f"{yval:.4f}", ha='center', va='bottom', fontsize=11, fontweight='bold')

    for ax in [ax1, ax2]:
        ax.spines['top'].set_visible(False)
        ax.spines['right'].set_visible(False)
        ax.spines['left'].set_linewidth(1.2)
        ax.spines['bottom'].set_linewidth(1.2)
        ax.tick_params(axis='x', labelsize=11)
        ax.tick_params(axis='y', labelsize=10)
        ax.grid(axis='y', linestyle='--', alpha=0.7)
        ax.grid(axis='x', visible=False)
```

```
    plt.tight_layout()
    fig.subplots_adjust(top=0.82)

    output_filename = "exp1_bar_charts.png"
    plt.savefig(output_filename, dpi=300, bbox_inches='tight')
    print(f"Graphique sauvegardé : {output_filename}")
    plt.show()


if __name__ == "__main__":
    generate_bar_charts(CSV_FILE)
```

```python
import cv2
import matplotlib.pyplot as plt
import os
import numpy as np

DIR = "exp1_visual_proofs(for_intensity_profile)"
IMG_GT_NAME      = "Lung_1_GroundTruth.png"
IMG_BICUBIC_NAME = "Lung_3_Bicubic.png"
IMG_PIFS_NAME    = "Lung_4_PIFS.png"
IMG_FIF_NAME     = "Lung_5_FIF.png"

ROW_INDEX = 50


def plot_intensity_profile():
    img_gt      = cv2.imread(os.path.join(DIR, IMG_GT_NAME),      cv2.IMREAD_GRAYSCALE)
    img_bicubic = cv2.imread(os.path.join(DIR, IMG_BICUBIC_NAME), cv2.IMREAD_GRAYSCALE)
    img_pifs    = cv2.imread(os.path.join(DIR, IMG_PIFS_NAME),    cv2.IMREAD_GRAYSCALE)
    img_fif     = cv2.imread(os.path.join(DIR, IMG_FIF_NAME),     cv2.IMREAD_GRAYSCALE)

    profile_gt      = img_gt[ROW_INDEX, :].astype(np.float32)
    profile_bicubic = img_bicubic[ROW_INDEX, :].astype(np.float32)
    profile_pifs    = img_pifs[ROW_INDEX, :].astype(np.float32)
    profile_fif     = img_fif[ROW_INDEX, :].astype(np.float32)

    x_range = range(120, 180)

    offset_pifs = 20
    offset_fif  = 40
    offset_bic  = 60

    plt.figure(figsize=(12, 6))
    plt.plot(x_range, profile_gt[x_range], label='Ground Truth', color='black', linewidth=2, linestyle='--')
    plt.plot(x_range, profile_bicubic[x_range] + offset_bic, label=f'Bicubique (+{offset_bic})', color='red', linewidth=1.5)
    plt.plot(x_range, profile_pifs[x_range] + offset_pifs, label=f'PIFS (+{offset_pifs})', color='green', linewidth=1.5)
    plt.plot(x_range, profile_fif[x_range] + offset_fif, label=f'FIF (+{offset_fif})', color='blue', linewidth=1.5)

    plt.title("Profil d'intensité 1D")
    plt.xlabel("Position X (pixels)")
    plt.ylabel("Intensité (0–255)")
    plt.legend()
    plt.grid(True, linestyle=':', alpha=0.7)
    plt.tight_layout()
    plt.show()


if __name__ == "__main__":
    plot_intensity_profile()
```

```python
import os
import cv2
import numpy as np
import csv
import time
from concurrent.futures import ProcessPoolExecutor

from skimage.metrics import structural_similarity as calculate_ssim

SAMPLE_SIZE_PER_CAT = 1
TEST_RESOLUTION = 512
ZOOM_TARGET = 4

PIFS_RANGE_SIZES = [16, 8, 4, 2]
FIF_ROUGHNESS = 0.5

DATASET_BASE_PATH = r"C:\Users\hp\Desktop\TIPE\Datasets"
OUTPUT_CSV = "experiment_2_raw_data.csv"


def get_images(folder, n):
    extensions = ('.jpg', '.jpeg', '.png')
    found = []
    for root, dirs, files in os.walk(folder):
        for f in files:
            if f.lower().endswith(extensions):
                found.append(os.path.join(root, f))
                if len(found) == n:
                    return found
    return found



def evaluate_algorithm(task):
    category, img_path, res, zoom, algo_type, param = task
    filename = os.path.basename(img_path)

    img_raw = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    img_gt = cv2.resize(img_raw, (res, res), interpolation=cv2.INTER_AREA)

    low_res = res // zoom
    img_degraded = cv2.resize(img_gt, (low_res, low_res), interpolation=cv2.INTER_AREA)

    start_time = time.perf_counter()

    if algo_type == 'BICUBIC':
        img_recon = cv2.resize(img_degraded, (res, res), interpolation=cv2.INTER_CUBIC)

    elif algo_type == 'PIFS':
        import PIFS_old
        encoded = PIFS.encode_pifs(img_degraded, range_size=param)
        img_recon = PIFS.decode_pifs(encoded, iterations=8, zoom_factor=zoom)

    elif algo_type == 'FIF':
        import FIF
        encoded = FIF.encode_fif(img_degraded, max_roughness=param)
```

```
        img_recon = FIF.decode_fif(encoded, iterations=8, zoom_factor=zoom)

    duration = time.perf_counter() - start_time
    ssim_val = calculate_ssim(img_gt, img_recon, data_range=255)

    return (category, filename, algo_type, param, duration, ssim_val)


def run_experiment():
    print("Démarrage de l'expérience 2 : Frontière de Pareto")
    print(f"Chargement de {SAMPLE_SIZE_PER_CAT} image(s) par anatomie...\n")

    categories = {
        "Brain": get_images(os.path.join(DATASET_BASE_PATH, "brainMRI"), SAMPLE_SIZE_PER_CAT),
        "Lung":  get_images(os.path.join(DATASET_BASE_PATH, "lungCT"),   SAMPLE_SIZE_PER_CAT),
        "Spine": get_images(os.path.join(DATASET_BASE_PATH, "spineXray"), SAMPLE_SIZE_PER_CAT)
    }

    tasks = []
    for cat, paths in categories.items():
        for img_path in paths:
            tasks.append((cat, img_path, TEST_RESOLUTION, ZOOM_TARGET, 'BICUBIC', 0))
            tasks.append((cat, img_path, TEST_RESOLUTION, ZOOM_TARGET, 'FIF', FIF_ROUGHNESS))
            for r in PIFS_RANGE_SIZES:
                tasks.append((cat, img_path, TEST_RESOLUTION, ZOOM_TARGET, 'PIFS', r))

    print(f"Lancement de {len(tasks)} benchmarks en parallèle...")

    raw_csv_data = []
    headers = ["Category", "Filename", "Algorithm", "Parameter", "Execution_Time_Sec", "SSIM"]
    results = []

    max_workers = max(1, os.cpu_count() - 1)
    with ProcessPoolExecutor(max_workers=max_workers) as executor:
        for res in executor.map(evaluate_algorithm, tasks):
            results.append(res)
            cat, filename, algo, param, duration, ssim_val = res

            param_str = f"R={param}" if algo == "PIFS" else (f"s={param}" if algo == "FIF" else "N/A")
            print(f"  {cat} | {algo:7} ({param_str:5}) | {filename[:15]}... | {duration:.4f}s")

            raw_csv_data.append([cat, filename, algo, param_str, duration, ssim_val])

    with open(OUTPUT_CSV, mode='w', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(headers)
```

```
        writer.writerows(raw_csv_data)

    print(f"\nDonnées exportées dans '{OUTPUT_CSV}'")

    # Compute and display Pareto averages
    pareto_points = {}
    for cat, filename, algo, param, duration, ssim_val in results:
        key = (algo, param)
        if key not in pareto_points:
            pareto_points[key] = {'time': [], 'ssim': []}
        pareto_points[key]['time'].append(duration)
        pareto_points[key]['ssim'].append(ssim_val)

    print("\nMoyennes — Frontière de Pareto")
    print(f"{'Algorithme':<10} | {'Paramètre':<10} | {'Temps moyen (s)':<17} | SSIM moyen")
    print("-" * 60)
    for (algo, param), data in pareto_points.items():
        mean_time = np.mean(data['time'])
        mean_ssim = np.mean(data['ssim'])
        param_str = f"R={param}" if algo == "PIFS" else (f"s={param}" if algo == "FIF" else "N/A")
        print(f"{algo:<10} | {param_str:<10} | {mean_time:<17.4f} | {mean_ssim:.4f}")


if __name__ == "__main__":
    run_experiment()
```

```python
import pandas as pd
import matplotlib.pyplot as plt

CSV_FILE = "experiment_2_raw_data(256).csv"


def generate_pareto_chart(csv_path):
    df = pd.read_csv(csv_path)
    df['Parameter'] = df['Parameter'].fillna('N/A')

    grouped = df.groupby(['Algorithm', 'Parameter'], as_index=False)[['Execution_Time_Sec', 'SSIM']].mean()

    bicubic_data = grouped[grouped['Algorithm'] == 'BICUBIC'].iloc[0]
    fif_data     = grouped[grouped['Algorithm'] == 'FIF'].iloc[0]
    pifs_data    = grouped[grouped['Algorithm'] == 'PIFS'].copy()

    pifs_data['R_val'] = pifs_data['Parameter'].str.extract(r'(\d+)').astype(int)
    pifs_data = pifs_data.sort_values('R_val', ascending=False)

    plt.style.use('seaborn-v0_8-whitegrid')
    fig, ax = plt.subplots(figsize=(11, 6.5))

    ax.scatter(bicubic_data['Execution_Time_Sec'], bicubic_data['SSIM'],
               color='#E63946', s=160, zorder=5, label='Bicubique')
    ax.annotate(
        f"Bicubique\n({bicubic_data['Execution_Time_Sec']*1000:.1f} ms, SSIM: {bicubic_data['SSIM']:.4f})",
        xy=(bicubic_data['Execution_Time_Sec'], bicubic_data['SSIM']),
        xytext=(10, 5), textcoords='offset points',
        fontsize=10, fontweight='bold', color='#E63946', ha='left'
    )

    ax.scatter(fif_data['Execution_Time_Sec'], fif_data['SSIM'],
               color='#457B9D', marker='*', s=260, zorder=5, label=r'FIF')
    ax.annotate(
        f"FIF ($s=0.5$)\n({fif_data['Execution_Time_Sec']:.2f} s, SSIM: {fif_data['SSIM']:.4f})",
        xy=(fif_data['Execution_Time_Sec'], fif_data['SSIM']),
        xytext=(0, -25), textcoords='offset points',
        fontsize=10, fontweight='bold', color='#457B9D', ha='center'
    )

    ax.plot(pifs_data['Execution_Time_Sec'], pifs_data['SSIM'],
            color='#2A9D8F', linestyle='-', linewidth=2.5, marker='o', markersize=8,
            zorder=4, label='PIFS (Grille Fixe Classique)')
```

```python

    offsets = {
        'R=16': (0, 10),
        'R=8':  (0, 10),
        'R=4':  (-15, -20),
        'R=2':  (12, 8)
    }

    for _, row in pifs_data.iterrows():
        p_name = row['Parameter']
        ox, oy = offsets.get(p_name, (0, 10))
        ha_val = 'left' if p_name == 'R=2' else ('right' if p_name == 'R=4' else 'center')
        ax.annotate(
            f"{p_name}\n({row['Execution_Time_Sec']:.1f} s, SSIM: {row['SSIM']:.4f})",
            xy=(row['Execution_Time_Sec'], row['SSIM']),
            xytext=(ox, oy), textcoords='offset points',
            fontsize=9.5, fontweight='semibold', color='#1b655c', ha=ha_val
        )

    ax.set_xscale('log')
    ax.set_title("Frontière de Pareto : Compromis Temps de Calcul vs. Fidélité Structurelle (SSIM)",
                 fontsize=14, fontweight='bold', pad=25)
    ax.set_xlabel("Temps d'exécution moyen (secondes) — Échelle Logarithmique", fontsize=12, labelpad=12)
    ax.set_ylabel("Indice de Similarité Structurelle Moyen (SSIM)", fontsize=12, labelpad=12)
    ax.set_ylim(0.2, 1.05)

    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.spines['left'].set_linewidth(1.2)
    ax.spines['bottom'].set_linewidth(1.2)
    ax.grid(True, which="both", linestyle='--', alpha=0.5)
    ax.legend(loc='upper right', frameon=True, facecolor='white', edgecolor='gray', framealpha=0.9, fontsize=11)

    plt.tight_layout()
    plt.show()


if __name__ == "__main__":
    generate_pareto_chart(CSV_FILE)
```

```python
import os
import cv2
import numpy as np
import PIFS

INPUT_IMAGE_PATH = "C:/Users/hp/Desktop/TIPE/code/exp1_visual_proofs(for_intensity_profile)/Spine_1_GroundTruth.png"
OUTPUT_DIR = "C:/Users/hp/Desktop/TIPE/code/snapshots"

TOTAL_ITERATIONS = 5
NUM_SNAPSHOTS = 4

MAX_SIZE = 16
MIN_SIZE = 2
THRESHOLD = 10.0
SEARCH_RADIUS = 8


def main():
    os.makedirs(OUTPUT_DIR, exist_ok=True)

    original_img = cv2.imread(INPUT_IMAGE_PATH, cv2.IMREAD_GRAYSCALE)
    h, w = original_img.shape

    print("Encodage PIFS en cours...")
    fractal_data = PIFS.encode_pifs(
        original_img,
        max_size=MAX_SIZE,
        min_size=MIN_SIZE,
        threshold=THRESHOLD,
        search_radius=SEARCH_RADIUS
    )
    transformations = fractal_data['transformations']
    print("Encodage terminé.")

    # Uniformly sample iteration indices from 0 to TOTAL_ITERATIONS
    # e.g. for 5 iterations and 4 snapshots: [0, 1, 3, 5]
    snapshots_to_save = np.linspace(0, TOTAL_ITERATIONS, NUM_SNAPSHOTS, dtype=int)
    print(f"Itérations ciblées : {snapshots_to_save}")

    canvas = np.random.randint(0, 256, (h, w)).astype(np.float64)
    file_index = 1

    if 0 in snapshots_to_save:
        path = os.path.join(OUTPUT_DIR, f"snapshot_{file_index}_iter_0.png")
        cv2.imwrite(path, canvas.astype(np.uint8))
        print(f"Sauvegardé : {path}")
        file_index += 1

    for i in range(1, TOTAL_ITERATIONS + 1):
        next_canvas = np.zeros((h, w), dtype=np.float64)

        for code in transformations:
            rx, ry, r_size = code['rx'], code['ry'], code['size']
            dx, dy = code['dx'], code['dy']
```

```
            d_size = code['size'] * 2

            D_large = canvas[dy:dy + d_size, dx:dx + d_size]
            D_shrunk = cv2.resize(D_large, (r_size, r_size), interpolation=cv2.INTER_AREA)

            isometries = PIFS.get_isometries(D_shrunk)
            D_trans = isometries[code['dir']]
            R_rebuilt = code['s'] * D_trans + code['o']

            next_canvas[ry:ry + r_size, rx:rx + r_size] = R_rebuilt

        canvas = next_canvas

        if i in snapshots_to_save:
            path = os.path.join(OUTPUT_DIR, f"snapshot_{file_index}_iter_{i}.png")
            cv2.imwrite(path, np.clip(canvas, 0, 255).astype(np.uint8))
            print(f"Sauvegardé : {path}")
            file_index += 1

    print("Snapshots générés avec succès.")


if __name__ == "__main__":
    main()
```