CARTOGRAPHIE 3D MULTI-VUES

Zhengyu Wang & Domingos Bravo

GE5

Table des matières

1. Introduction

L'objectif de ce projet est de développer un programme de reconstruction 3D à partir de plusieurs vues :

- Extraire et faire correspondre des points d'intérêt d'une séquence d'images (par exemple filmées par un drone en vol autour d'une scène, ou par une caméra à la main) ;
- Estimer la pose (rotation et translation) de la caméra à chaque image ;
- Obtenir les coordonnées 3D des points d'intérêt par triangulation et étendre progressivement le nuage de points de la scène au fil de la séquence ;
- Associer les informations de texture (RGB) provenant des images originales à ces points 3D, afin de produire un nuage de points coloré (fichier .ply).

Comparé au traitement d'images purement 2D, la reconstruction 3D multi-vues est largement utilisée en métrologie, photogrammétrie, navigation robotique, AR/VR et ingénierie inversée. Dans ce projet, nous adoptons les principes du SLAM visuel (Simultaneous Localization and Mapping) : à partir des contraintes géométriques entre deux (ou plusieurs) vues, on détermine le mouvement de la caméra et on reconstruit la scène.

2. Principe de la reconstruction

2.1 Objectifs du projet et aperçu du SLAM visuel

Dans un SLAM visuel, « localiser simultanément et construire la carte » signifie :

- Localiser en continu la pose (R, t) de la caméra au fil des images,
- Mettre à jour en même temps la carte 3D du monde (nuage de points).

Lorsque l'on traite hors ligne une série d'images, cela peut s'assimiler à de la reconstruction multi-vues (Structure from Motion, SfM), centrée sur la géométrie projective de la caméra et la correspondance de points d'intérêt entre images.

2.2 Modèle de caméra et repère de coordonnées

Dans le modèle de caméra à trou d'aiguille (pinhole), un point 3D X=(X,Y,Z,1) est projeté en un point 2D x=(x,y,1) via la matrice de projection P=K[R|t].

- La matrice intrinsèque K regroupe les focales (fx,fy) et le centre principal (cx,cy). Sans distorsion, la projection suit :

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} [R \mid t] \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

- La matrice d'extrinsèques [R|t] décrit la rotation et la translation de la caméra dans le référentiel du monde (ou le passage du monde à la caméra).

2.3 Extraction et correspondance de caractéristiques (SIFT + BFMatcher + Ratio Test)

Pour faire correspondre des points d'intérêt entre deux images, on utilise un détecteur-descripteur robuste (SIFT), invariant à l'échelle et à l'orientation, puis on emploie un algorithme de matching (Brute-Force Matcher, par exemple). Afin de réduire les fausses correspondances, on applique le « test du ratio » de Lowe (souvent 0,7), qui écarte les appariements ambiguës ou trop similaires.

2.4 Estimation du mouvement de la caméra et contraintes géométriques (Matrice Essentielle, PnP)

Une fois les correspondances 2D établies, on peut estimer la matrice essentielle EE (via cv2.findEssentialMat + RANSAC) pour récupérer la rotation et la translation relatives (cv2.recoverPose). Pour les images suivantes, si l'on possède déjà un certain nombre de points 3D, on peut employer cv2.solvePnPRansac (PnP) pour affiner la pose de la nouvelle vue à partir des paires (points 3D, points 2D).

2.5 Triangulation et extension incrémentale (Triangulation & Incremental SFM)

Connaissant les matrices de projection P1,P2 de deux caméras et les points correspondants en 2D, on peut trianguler pour retrouver les coordonnées 3D (cv2.triangulatePoints). Dans un schéma

incrémental, on répète ce processus :

- On estime la pose (R, t) de la nouvelle vue,
- On triangule de nouveaux points,
- On étend progressivement le nuage de points global (Structure from Motion incrémental).

2.6 Association des textures et visualisation du nuage de points (.ply avec couleurs RGB)

Pour une visualisation plus réaliste, on associe à chaque point 3D la couleur (R, G, B) lue dans l'image. On exporte enfin l'ensemble au format .ply (ou équivalent), qui peut être ouvert dans des logiciels comme CloudCompare ou Meshlab.

3. Environnement du projet et dépendances

Ce projet repose sur l'utilisation de Python (version 3.8 ou ultérieure recommandée) ainsi que sur plusieurs bibliothèques :

- **OpenCV (cv2)** : fournit différentes fonctionnalités de vision par ordinateur (lecture d'images, extraction SIFT, RANSAC, PnP, triangulation, appariement stéréoscopique, etc.).
- **NumPy** : bibliothèque de calcul numérique, dédiée aux opérations sur tableaux et matrices.
- **Matplotlib** : utile, entre autres, pour tracer les courbes d'erreur de reprojection au fil des images.
- **scipy.optimize (least_squares)** : permet de réaliser l'optimisation de type bundle adjustment (moindres carrés non linéaires).
- **tqdm** : génère une barre de progression, facilitant la visualisation de l'avancement du traitement.

De plus, il est nécessaire de disposer du fichier contenant les paramètres intrinsèques de la caméra (K.txt) ainsi que d'un dossier renfermant les images (plusieurs fichiers .jpg ou .png) à reconstruire.

4. Aperçu de l’algorithme et de la structure du code

Cette section décrit les principales classes (ou « modules ») et leurs fonctions, ainsi que la logique globale de leur enchaînement. Elles couvrent successivement :

1. Le chargement des images et de la matrice intrinsèque,
2. L’extraction et la mise en correspondance de caractéristiques (points d’intérêt),
3. L’estimation de la pose de la caméra (rotation, translation),
4. La triangulation pour reconstruire les points 3D,
5. Le calcul de l’erreur de reprojection,
6. (En option) le bundle adjustment (optimisation globale),
7. Puis l’export du nuage de points coloré,
8. Enfin, la recherche de points communs entre différentes vues.

Le diagramme ci-dessous illustre la logique d’initialisation (sur deux premières vues) suivie d’une **boucle incrémentale** (pour toutes les vues suivantes) **(Figure 1)**. On y voit comment chaque module (4.2 à 4.8) s’articule jusqu’à l’export final du nuage de points :

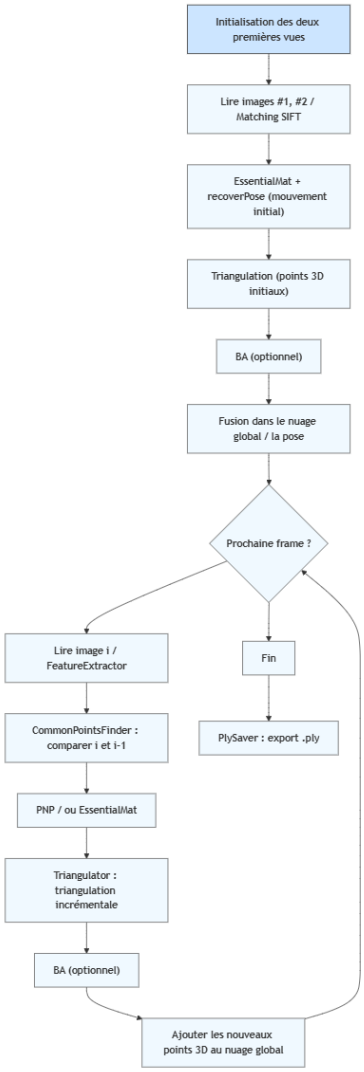

Figure 1 : Schéma du flux incrémental de l’algorithme

Vous trouverez ensuite, dans les sous-sections, une description détaillée de chaque partie et la référence vers le code complet.

4.1 Flux principal : classe Sfm

Fonctionnalités

- Lit les images, détecte et apparie leurs points d'intérêt, puis estime le mouvement entre la première paire via la matrice essentielle (findEssentialMat) et recoverPose.
- Réalise une triangulation pour obtenir un lot initial de points 3D et calcule l'erreur de reprojection.
- Pour les images suivantes, emploie PnP afin d'estimer la pose, puis triangule de nouveaux points pour enrichir le nuage 3D.
- (Optionnel) lance un Bundle Adjustment après chaque étape (ou toutes les nn images).
- Fusionne enfin l'ensemble des points 3D (avec leur couleur) et enregistre le résultat au format .ply.

Processus général (simplifié)

1. **Charger les deux premières images** → Extraire/apparier les points SIFT → cv2.findEssentialMat + cv2.recoverPose → on obtient (R,t).
2. Faire appel à Triangulator.triangulation(...), filtrer les points incohérents (Z négatifs, etc.).
3. Si le BA est activé, utiliser BundleAdjuster.bundle_adjustment(...) pour optimiser la pose et les points.
4. Pour chaque image ultérieure (3e, 4e, etc.) :
    - Apparier les points, repérer ceux qui sont communs (CommonPointsFinder),
    - Résoudre PnP → Trianguler plus de points,
    - (Optionnel) BA,
    - Sauvegarder le nuage 3D et les poses finales.

Répertoire A : le code de la classe Sfm.

4.2 Chargement des images et matrice K : classe Image_loader

Fonctionnalités

- Lit K.txt (matrice 3×3) et recense les fichiers image dans le répertoire spécifié (triés dans l'ordre).
- Éventuellement, effectue un sous-échantillonnage : on peut appliquer downscale à la matrice K et downscale_image aux images (via cv2.pyrDown), pour alléger les calculs.

Méthodes clés

- __init__(self, img_dir, downscale_factor) : lit K.txt, récupère la liste d'images, gère downscale_factor.
- downscale(self) : divise (fx,fy,cx,cy) par downscale_factor.
- downscale_image(self, image) : applique plusieurs fois cv2.pyrDown (si factor=4, on le fait 2 fois).

Répertoire 2 : le code de la classe Image_loader.

4.3 Extraction et correspondance SIFT : classe FeatureExtractor

Fonctionnalités

- Rassemble l'utilisation de SIFT sur deux images, puis un BFMatcher().knnMatch(...) pour déterminer les correspondances,
- Applique le test du ratio de Lowe (typ. <0,70) pour enlever les matches ambigus,
- Retourne deux tableaux (N×2) de positions 2D pour chaque image.

Méthode clé

- find_features(self, image_0, image_1) :
    1. Crée un détecteur SIFT (cv2.SIFT_create()).
    2. Extrait keypoints + descripteurs dans image_0 et image_1.
    3. Fait bf.knnMatch(desc_0, desc_1, k=2) pour trouver la meilleure et la deuxième meilleure correspondance.
    4. Conserve celles satisfaisant m.distance < ratio * n.distance.
    5. Retourne deux tableaux (N×2) de points 2D en correspondance.

Répertoire 3 : le code de la classe FeatureExtractor.

4.4 Triangulation : classe Triangulator

Fonctionnalités

- Données : deux matrices de projection 3×4 et paires de points 2D.
- Appelle cv2.triangulatePoints(P1, P2, pts1, pts2), qui renvoie un tableau (4, N) de points homogènes.
- Normalise (division par la 4^{e} composante), écarte les points en Z négatif ou trop lointains.

Méthode clé

- triangulation(self, projection_matrix_1, projection_matrix_2, point_2d_1, point_2d_2) :
    1. Formate les points 2D en (2, N).
    2. Appelle cv2.triangulatePoints.
    3. Divise par la coordonnée homogène (4^{e} ligne).
    4. Filtre selon la profondeur (Z>0, etc.).
    5. Retourne les points 3D et les points 2D filtrés.

Répertoire 4 : le code de la classe Triangulator.

4.5 Estimation de la pose (PnP) : classe PnPSolver

Fonctionnalités

- Lorsque certaines coordonnées 3D (déjà triangulées) et leurs positions 2D dans la nouvelle image sont connues, on peut résoudre la pose via cv2.solvePnPRansac.
- On obtient (rvec,tvec) (rotation en Rodrigues + translation), puis on convertit rvecrvec en matrice 3×3.
- On renvoie également la liste d'inliers.

Méthode clé

- PnP(self, obj_point, image_point, K, dist_coeff, rot_vector, initial) :

1. Ajuste parfois les formats selon initial.
2. Exécute cv2.solvePnPRansac(...).
3. Convertit rvec → matrice de rotation (3×3).
4. Filtre les correspondances outliers.
5. Retourne R, t, les points 2D/3D inliers.

Répertoire 5 : le code de la classe PnPSolver.

4.6 Erreur de reprojection et bundle adjustment

4.6.1 Classe ReprojectionErrorCalculator

Fonctionnalités

- Calcule l'erreur de reprojection : compare la projection des points 3D (via R, t, K) aux coordonnées 2D observées,
- Fournit aussi la fonction optimal_reprojection_error(...), destinée à être appelée par least_squares (moindres carrés).

Méthodes clés

- reprojection_error(self, obj_points, image_points, transform_matrix, K, homogenity) :
  - Extrait R, t de transform_matrix[:3, :3] et transform_matrix[:3, 3],
  - Convertit (X, Y, Z, 1) → (X, Y, Z) si nécessaire,
  - Utilise cv2.projectPoints, compare à image_points,
  - Retourne l'erreur moyenne.
- optimal_reprojection_error(self, obj_points) :
  - Décode un grand vecteur [R|t, K, points2D, points3D],
  - Reprojette,
  - Retourne le vecteur des erreurs pour chaque point (pour least_squares).

4.6.2 Classe BundleAdjuster

Fonctionnalités

- Emballe la pose (R|t), la matrice K, les points 3D et 2D dans un vecteur unique,
- Lance least_squares(self.reproj_calc.optimal_reprojection_error, x0, …) pour minimiser l'erreur globale,
- Renvoie la pose, la K et les 3D mises à jour.

Méthode clé

- bundle_adjustment(self, _3d_point, opt, transform_matrix_new, K, r_error) :
  1. Construit [transform_matrix_new, K, points2D, points3D],
  2. Appelle least_squares (avec gtol=r_error),
  3. Décode la solution pour retrouver R, t, K, 3D.

Répertoire 6 : le code des classes ReprojectionErrorCalculator & BundleAdjuster.

4.7 Sauvegarde du nuage de points (avec couleurs) : classe PlySaver

Fonctionnalités

- Assemble (X, Y, Z) et (B, G, R) dans un même tableau,
- Calcule la distance de chaque point au barycentre, supprime les outliers,

- Écrit un fichier ASCII .ply (lisible par Meshlab, CloudCompare, etc.).

Méthode clé

- to_ply(self, path, point_cloud, colors, image_list) :
    1. Éventuellement multiplie les coordonnées par un facteur,
    2. Concatène (XYZ) et (BGR),
    3. Filtre selon la distance au centre,
    4. Crée le fichier .ply et y écrit l'en-tête plus les sommets.

Répertoire 7 : le code de la classe PlySaver.

4.8 Recherche de points communs : classe CommonPointsFinder

Fonctionnalités

- Dans une reconstruction incrémentale, on veut parfois repérer les mêmes points 2D apparaissant entre deux images, pour mieux trianguler ou résoudre PnP.
- Compare les positions (x, y) et établit un masque.

Méthode clé

- common_points(self, image_points_1, image_points_2, image_points_3) :
    1. Pour chaque point de image_points_1, cherche s'il figure dans image_points_2,
    2. Masque ces indices dans image_points_2 / image_points_3,
    3. Retourne les indices communs et les listes filtrées.

Répertoire 8 : le code de la classe CommonPointsFinder.

4.9 Conclusion

Grâce à l'enchaînement de ces classes et à leurs appels successifs, le programme exécute, pour chaque image, la détection et la mise en correspondance (SIFT), l'estimation de la pose (EssentialMat / PnP), la triangulation incrémentale et, si besoin, le bundle adjustment, pour finalement produire un nuage 3D coloré et les poses de la caméra.

5. Expérimentations et résultats

Dans ce chapitre, nous menons plusieurs expériences afin d'évaluer l'impact de certains facteurs clés sur la qualité de la reconstruction 3D multi-vues :

1. L'utilisation ou non du **SIFT Ratio Test** pour filtrer les appariements,
2. Le recours ou non au **Bundle Adjustment (BA)**,
3. L'application ou non d'un **sous-échantillonnage** (downscale) des images (avec mise à l'échelle de la matrice intrinsèque).

En outre, nous observons le temps de traitement, la répartition des mauvais appariements (outliers), l'évolution de l'erreur de reprojection et, enfin, la qualité globale du nuage de points.

5.1 Jeu de données et paramètres d'expérience

- **Jeu de données et matrice intrinsèque**
  Pour cette démonstration, nous utilisons le dossier Datasets/fountain-P11, qui contient plusieurs images du même site (une fontaine) sous différents angles de vue, ainsi qu'un fichier K.txt (matrice intrinsèque 3×3 selon le modèle pinhole).
  S'il existe une distorsion optique, on peut la corriger ; ici, on suppose qu'elle est négligeable.
- **Facteur de sous-échantillonnage**
  - Si downscale_factor=2.0, on applique cv2.pyrDown pour réduire la largeur et la hauteur de moitié, puis on divise (f_x, f_y, c_x, c_y) par 2 dans la matrice K.
  - Sans sous-échantillonnage, on garde la résolution originelle et la matrice K initiale.
- **Configuration algorithmique**
  - **Détection SIFT** : on extrait les points-clés (keypoints) et leurs descripteurs (128 dimensions) pour chaque image.
  - **Appariement + filtrage** : BFMatcher pour trouver le plus proche voisin. Dans certains tests, on applique le **Lowe Ratio Test** (seuil ~ 0,70).
  - **Seuils RANSAC** :
    - cv2.findEssentialMat : threshold=0.4,
    - cv2.solvePnPRansac : valeurs proches de la configuration standard.
  - **Tolérance de convergence du BA** : dans les expériences où le BA est activé, on fixe gtol=0.5 (ou moins) dans least_squares, afin de doser la précision et le temps de calcul.
- **Critères d'évaluation**
  - **Erreur de reprojection** : on la calcule après chaque étape de pose ou après BA, en pixels (distance moyenne entre projections 2D et observations).
  - **Temps d'exécution** : temps total (depuis la lecture des images jusqu'à la sortie du nuage de points) ou par étapes clés.
  - **Nuage de points** : on génère un fichier .ply (avec couleurs RGB), visualisable sous Meshlab/CloudCompare pour vérifier la densité et la cohérence.

5.2 Déroulement et visualisation

En invoquant Sfm("Datasets/fountain-P11"), on obtient un flux de traitement typique :

1. **Récupération du mouvement initial (premières images)**
    - Extraction et appariement de points d'intérêt (SIFT),
    - Évaluation de la matrice essentielle via cv2.findEssentialMat + RANSAC, puis décomposition pour extraire la rotation et la translation (cv2.recoverPose).
    - Triangulation pour un ensemble initial de points 3D, contrôle de l'erreur de reprojection.
2. **Extension incrémentale aux images suivantes**
    - Pour la kk-ième image, on la met en correspondance avec la (k−1)(k-1)-ième,
    - On combine les correspondances 2D + les points 3D déjà connus pour résoudre la pose (PnP ou EssentialMat),
    - On re-triangule les nouveaux appariements afin d'enrichir le nuage 3D,
    - Si BA est activé, on lance régulièrement least_squares pour réduire la dérive cumulée.
3. **Sortie et visualisation**
    - Une fois toutes les images traitées, on sauvegarde le nuage 3D (fichier .ply) avec les couleurs et on peut tracer la courbe de l'erreur de reprojection ou l'imprimer à la console.

Dans ce projet, on peut également, pour certains tests, lancer un **appareil stéréo dense** (type StereoSGBM) entre deux vues proches, afin de générer un nuage plus dense. Cette étape est optionnelle.

5.3 Première comparaison : avec ou sans Ratio Test

Dans ce test, la seule différence réside dans l'utilisation (ou non) du **Lowe Ratio Test** lors de l'appariement. Tous les autres réglages (downscale éventuel, BA, etc.) restent constants. Nous comparons deux cas :

(A) Sans Ratio Test

- Après knnMatch (qui renvoie (m,n)), on sélectionne toujours le plus proche m comme correspondance, sans vérifier m.distance < 0.7 * n.distance.
- On obtient un nombre plus élevé de correspondances, mais aussi beaucoup plus de faux appariements ; certaines images montrent alors une forte explosion de l'erreur de reprojection.

Par exemple, on observe sur la 6^{e} image un pic d'erreur dépassant 600, comme le montre la **Figure 2** :

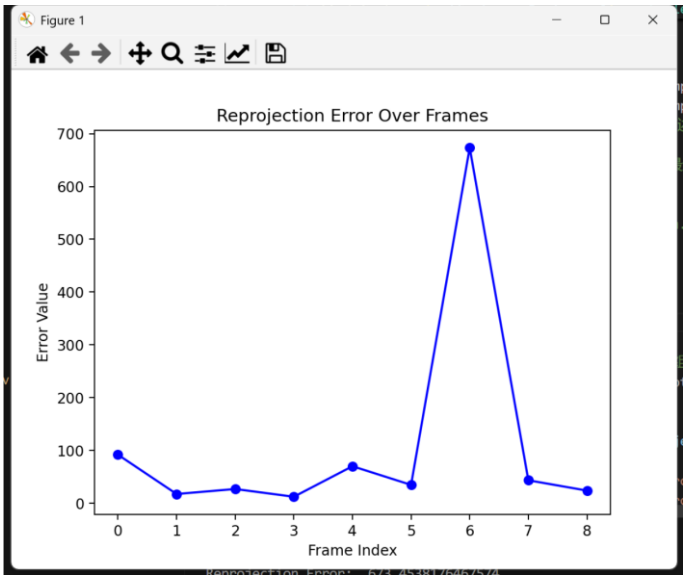

Figure 2 : Statistiques d'erreur sans Ratio Test, présence d'un pic très élevé

Le nuage de points s'avère plus dispersé localement, comme l'illustre la **Figure 3**

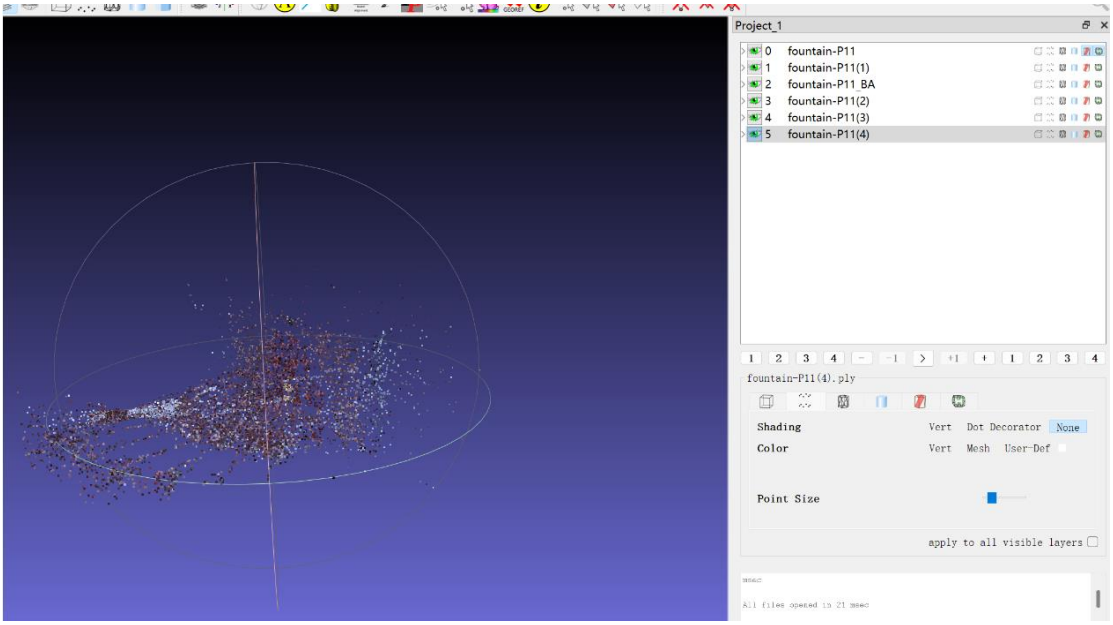

Figure 3 : Nuage partiel avec grande dispersion sans Ratio Test

(B) Avec Ratio Test

- On ne conserve que ceux où m.distance < 0.70 * n.distance, ce qui élimine les zones floues ou très répétitives.
- On obtient un nombre de matches moins élevé, mais nettement moins d'erreurs flagrantes. La courbe d'erreur de reprojection est plus régulière et sans pics soudains.

La **Figure 4** montre que, pour certaines images, l'erreur de reprojection reste autour de 0~1 pixel :

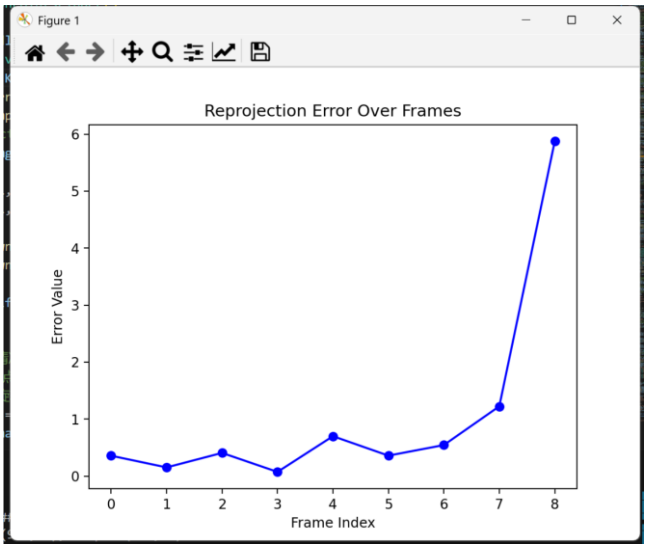

Figure 4 : Distribution d'erreur plus compacte grâce au Ratio Test

De même, la **Figure 5** illustre un nuage de points plus dense et plus cohérent, avec peu de points isolés :

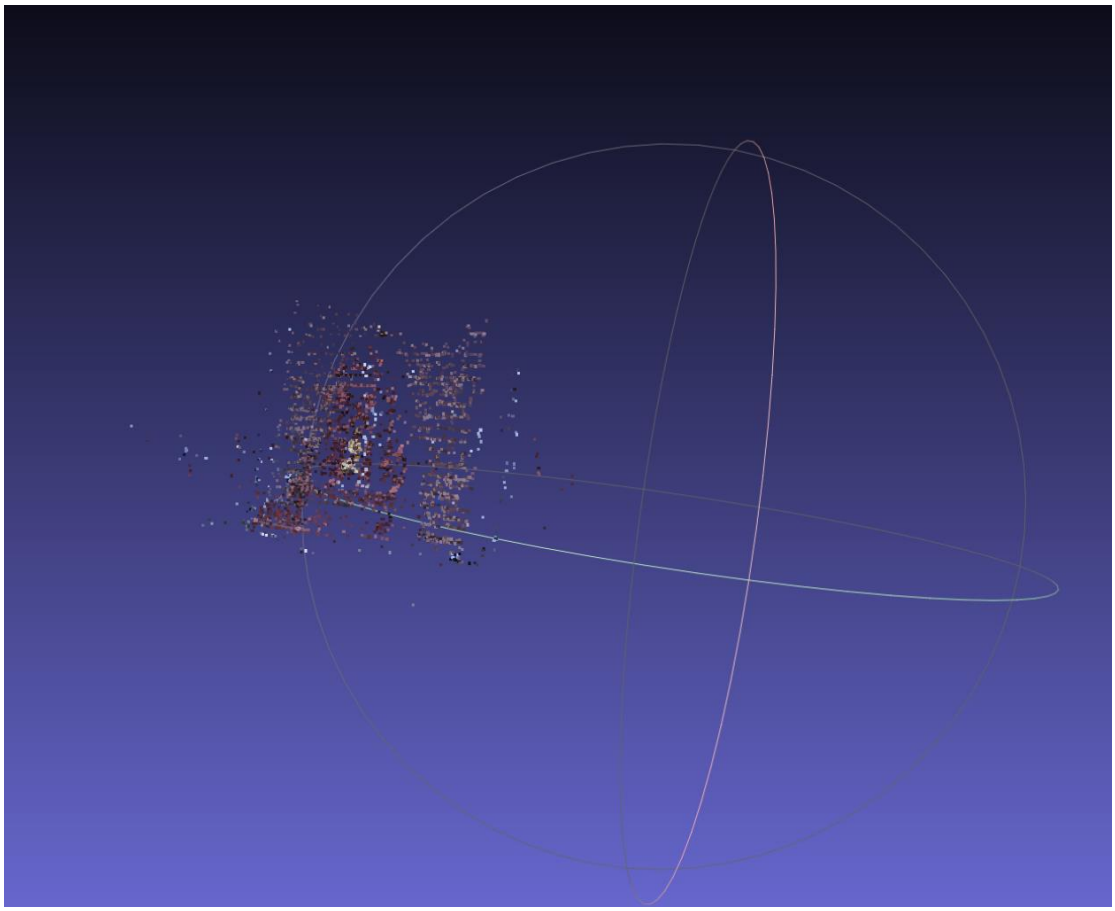

Figure 5 : Nuage plus homogène avec Ratio Test

Conclusion :

- **Sans Ratio Test** : potentiellement plus de matches dans des scènes simples, mais si la scène contient des motifs répétitifs, on subit de nombreux faux appariements provoquant un écart accru lors de la triangulation.
- **Avec Ratio Test** : plus robuste dans la plupart des cas, car on élimine la plupart des

ambiguïtés de matching ; la reconstruction en ressort globalement plus fiable.

5.4 Deuxième comparaison : avec ou sans Bundle Adjustment (BA)

Ici, on conserve l'approche SIFT + Ratio Test + RANSAC, et on compare le fait d'activer ou non le BA (optimisation globale).

(A) Sans BA

- À chaque vue, on effectue PnP (ou EssentialMat) puis on triangule. On se contente d'éliminer les outliers, sans lancer d'optimisation globale.
- Inconvénient : des erreurs mineures peuvent s'accumuler avec le nombre de vues, provoquant une élévation notable de l'erreur de reprojection (et un nuage 3D partiellement dégradé).

La **Figure 6** montre ainsi que l'erreur peut grimper à 5 ou 6 pixels en fin de séquence :

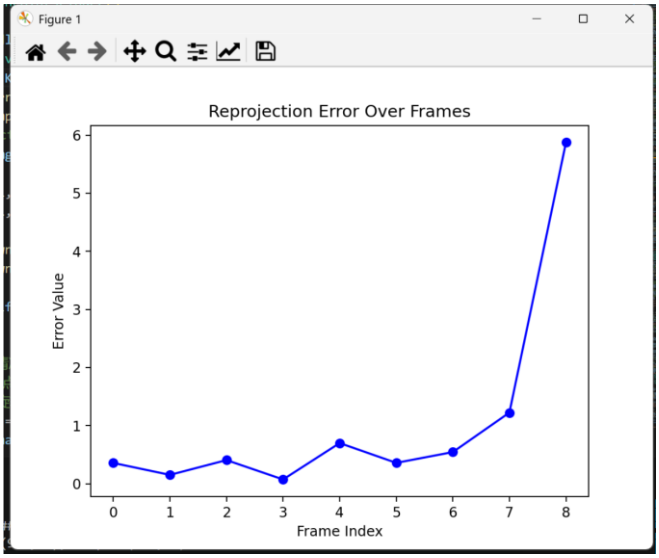

Figure 6 : Courbe d'erreur croissante sans BA, finale autour de 5–6

Le nuage de points est globalement correct, mais on observe parfois un certain flou ou des trous localisés (**Figure 7**) :

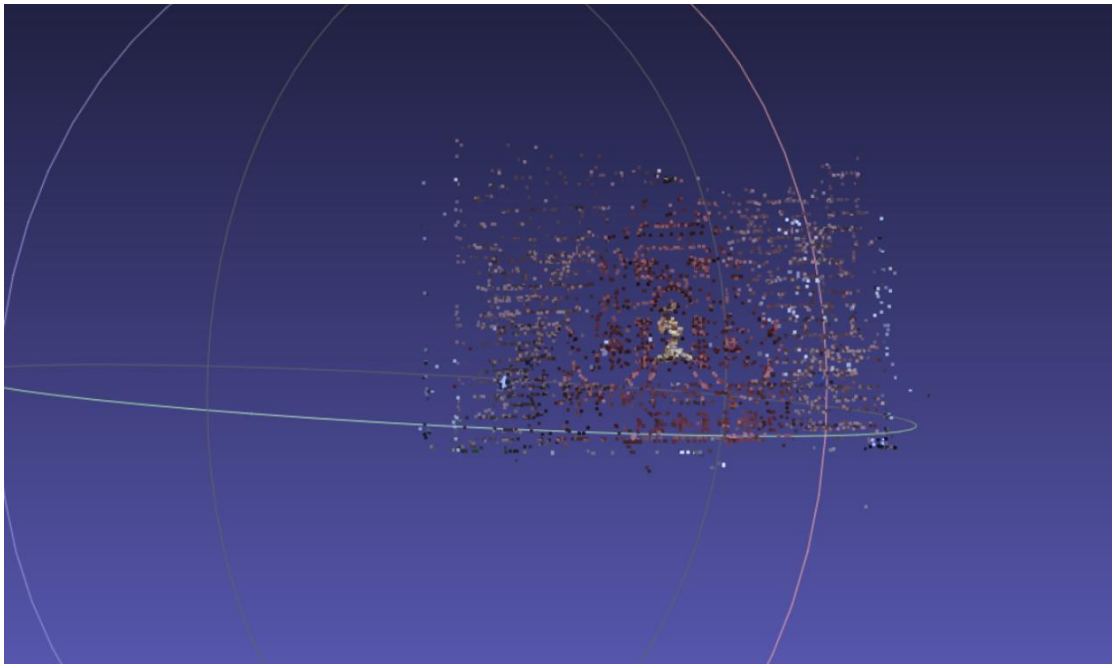

Figure 7 : Nuage plus clairsemé, avec quelques points aberrants sans BA

(B) Avec BA

- Après chaque étape (ou à une fréquence définie), on lance un bundle adjustment (moindres carrés non linéaires) pour ajuster simultanément la pose et les points 3D.
- Résultat : on corrige l'erreur cumulative et on obtient un nuage plus précis.

Par exemple, la **Figure 8** affiche une courbe d'erreur plus stable, souvent confinée entre 0.01 et 0.05 pixel :

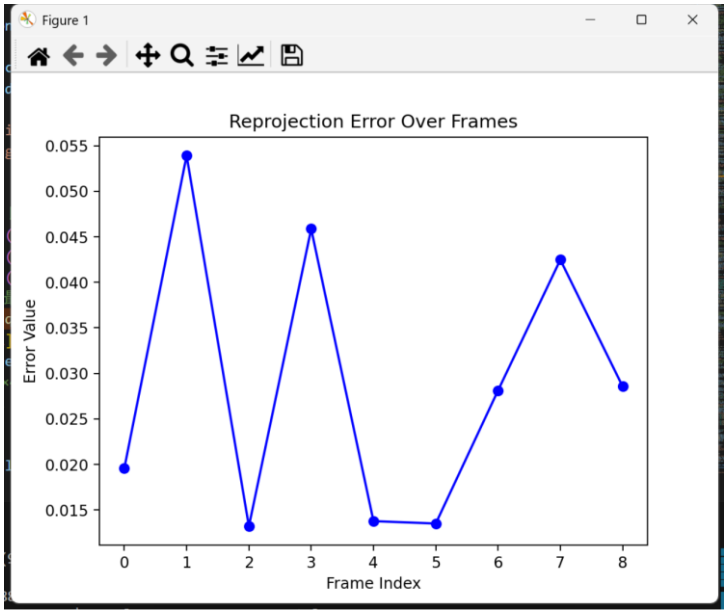

Figure 8 : Courbe d'erreur plus stable grâce au BA

Le nuage **(Figure 9)** est plus régulier, reflétant mieux la géométrie réelle :

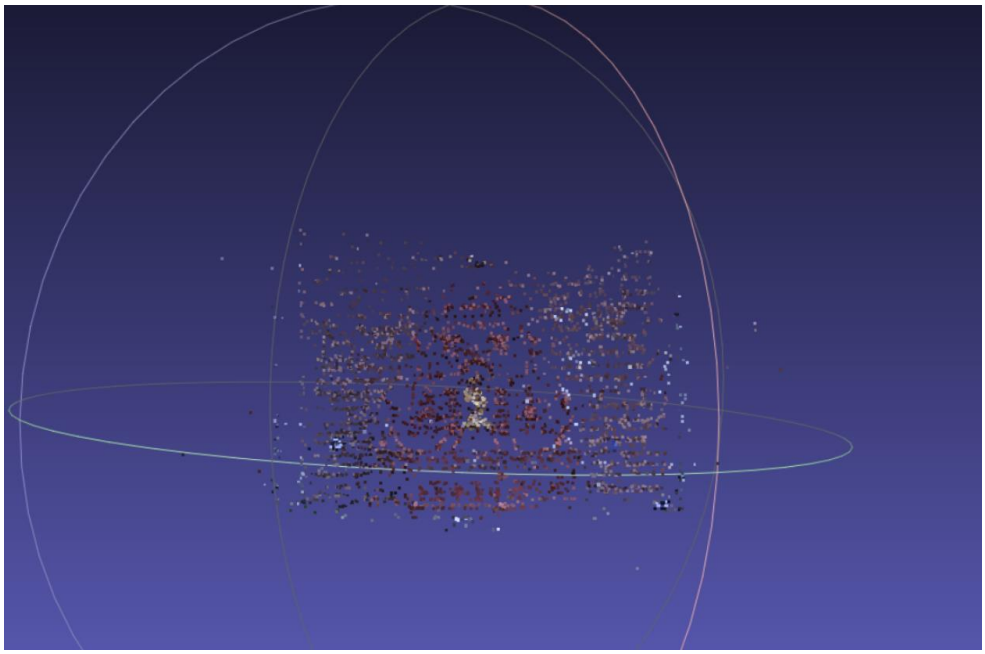

Figure 9 : Nuage plus fidèle après BA

Ces deux images côte à côte sont encore plus parlantes :

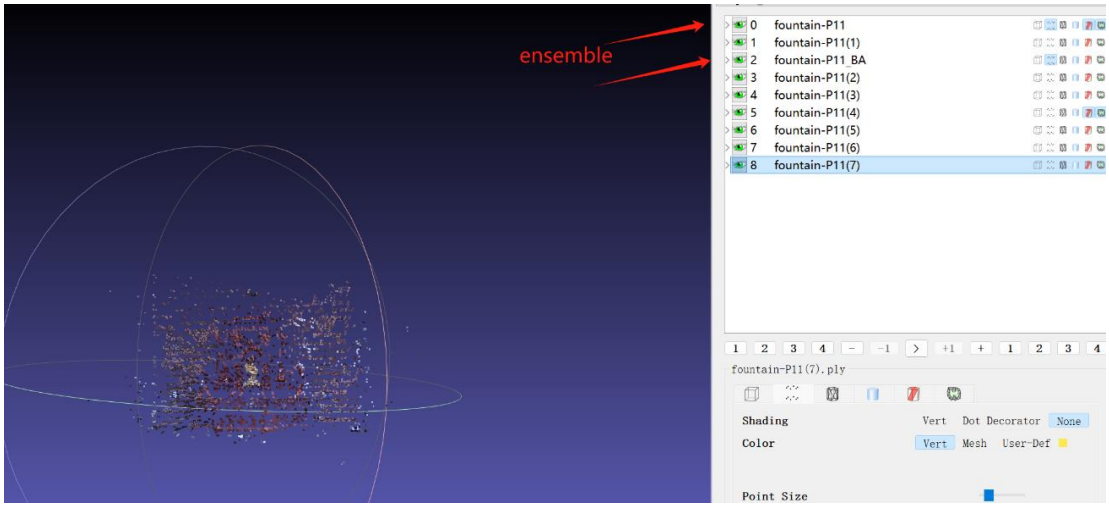

Figure 10: Superposition des deux nuages

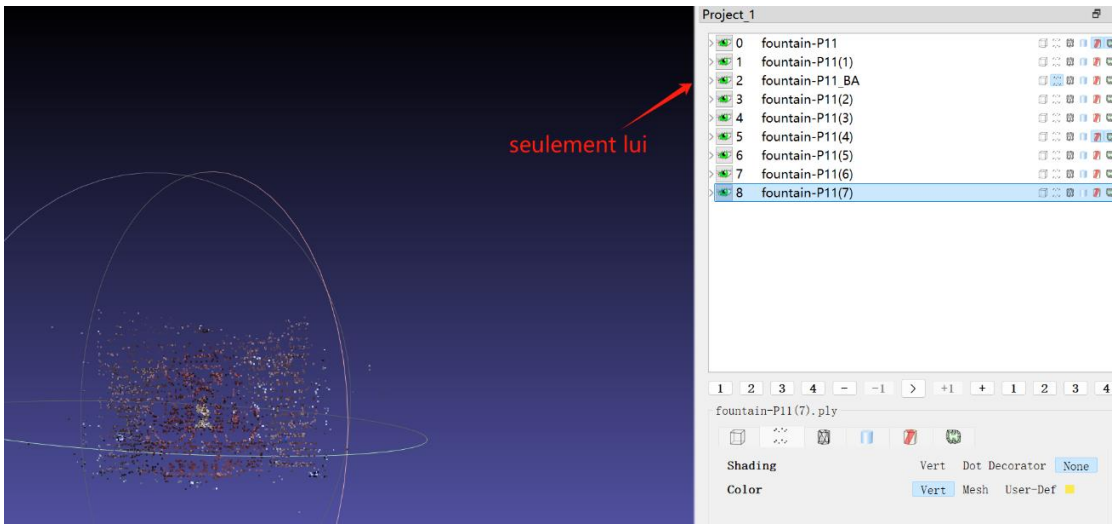

Figure 11: Nuage correspondant au cas « avec BA » uniquement

Ici, on pourrait croire à un simple « déplacement » ou « alignement », alors qu'en réalité, cela a permis de corriger de nombreuses erreurs de triangulation en arrière-plan.

Analyse :

- Le BA, en ajustant à la fois (R, t) et les points 3D, réduit la dérive cumulative.
- Dans la reconstruction multi-vues, c'est souvent un outil indispensable pour augmenter la précision.
- Inconvénient : plus de calculs, ce qui peut être significatif si le nombre d'images ou de points est très élevé.

En résumé :

- **Sans BA** : moins de calcul, mais potentiellement plus d'erreurs accumulées, surtout sur de longues séquences.
- **Avec BA** : meilleure précision finale, au prix d'un temps de traitement supplémentaire.

5.5 Troisième comparaison : sous-échantillonnage ou non

Nous examinons maintenant l'effet du downscale (et de la mise à l'échelle de K) sur la précision et la rapidité de la reconstruction. Pour être complet, nous étudions successivement la situation **sans BA** puis **avec BA**, et dans chaque cas nous comparons « non downscale » et « downscale ». Les autres paramètres (SIFT + Ratio Test + RANSAC) restent identiques.

5.5.1 Sans BA : comparer « non downscale » vs « downscale »

(A) Non downscale

- On ne modifie pas la résolution ni la matrice K,
- On traite un petit nombre d'images, avec PnP + triangulation incrémentale, sans BA.

Résultats :

- Temps d'exécution : ~15 s,
- À la fin, l'erreur de reprojection peut atteindre ~10 pixels.

La **Figure 12** illustre la console indiquant ~15 s de calcul :

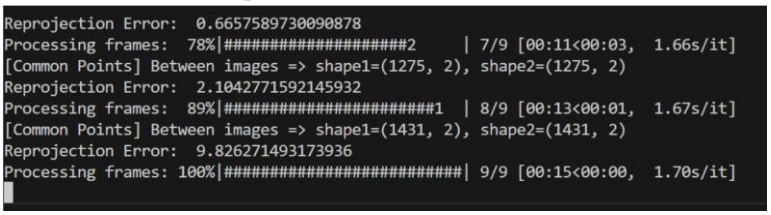

Figure 12 : Console, non downscale + sans BA, ~15 s

La **Figure 13** montre qu'en fin de séquence, l'erreur dépasse 10 pixels :

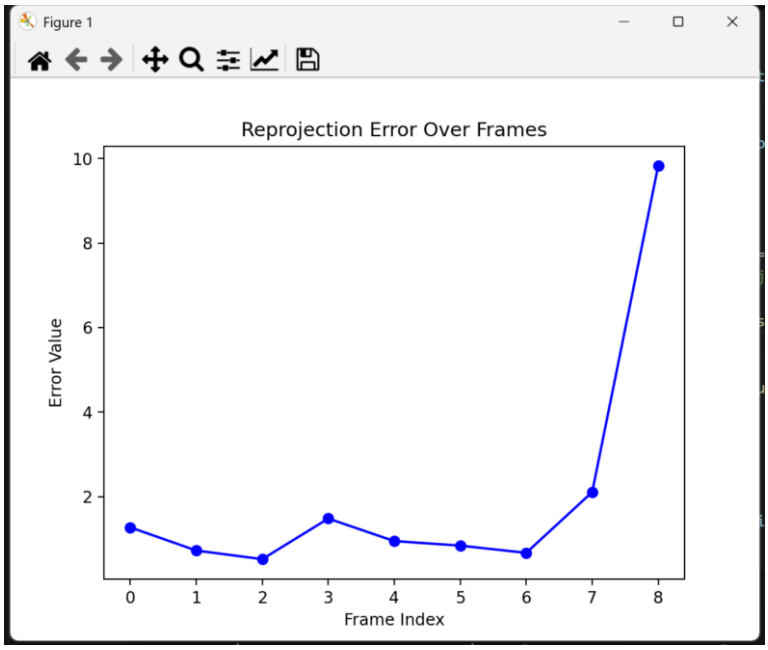

Figure 13 : Pic d'erreur à 10, suggérant beaucoup d'outliers

Le nuage contient pas mal de points aberrants :

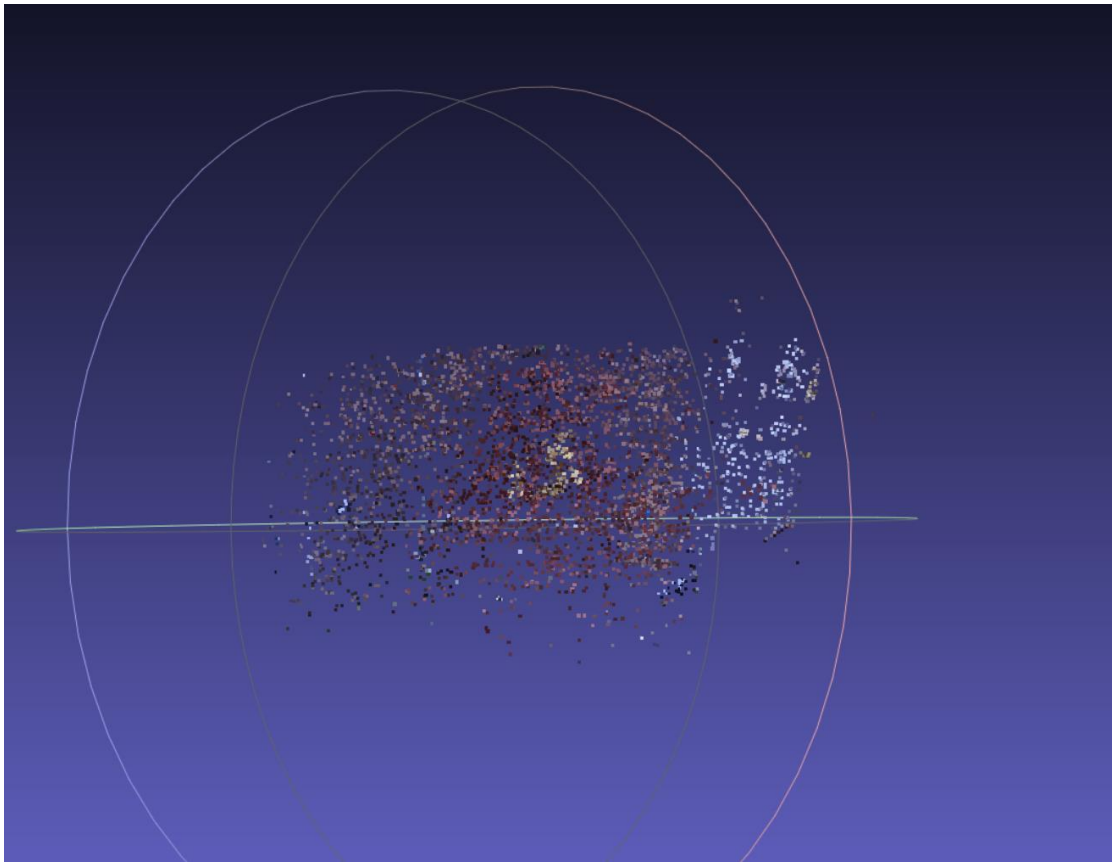

Figure 14 : Nuage éparpillé sans BA, haute résolution

Analyse :

- À haute résolution, on trouve plus de correspondances, mais sans BA, les erreurs s'accumulent, faisant dériver la reconstruction.

(B) Downscale

- On applique downscale_factor=2.0, et donc downscale_image() + division de (f_x, f_y, c_x, c_y) par 2,
- Toujours sans BA (PnP + triangulation).

Résultats :

- Temps d'exécution : ~4 s,
- L'erreur maximale est ~6 pixels, mieux que 10,
- Moins de faux appariements dans le nuage.

La **Figure 15** indique qu'en ~4 s, on termine :

```
Reprojection Error:  0.5447959916892302
Processing frames:  78%|####################2     | 7/9 [00:03<00:01,  1.88it/s]
[Common Points] Between images => shape1=(993, 2), shape2=(993, 2)
Reprojection Error:  1.2245528222266875
Processing frames:  89%|########################1  | 8/9 [00:04<00:00,  1.86it/s]
[Common Points] Between images => shape1=(1042, 2), shape2=(1042, 2)
Reprojection Error:  5.875138744346569
Processing frames: 100%|#########################| 9/9 [00:04<00:00,  1.82it/s]
```

Figure 15 : Console, downscale + sans BA, seulement 4 s

La courbe montre une erreur autour de 0~1 la plupart du temps, culminant à ~6 :

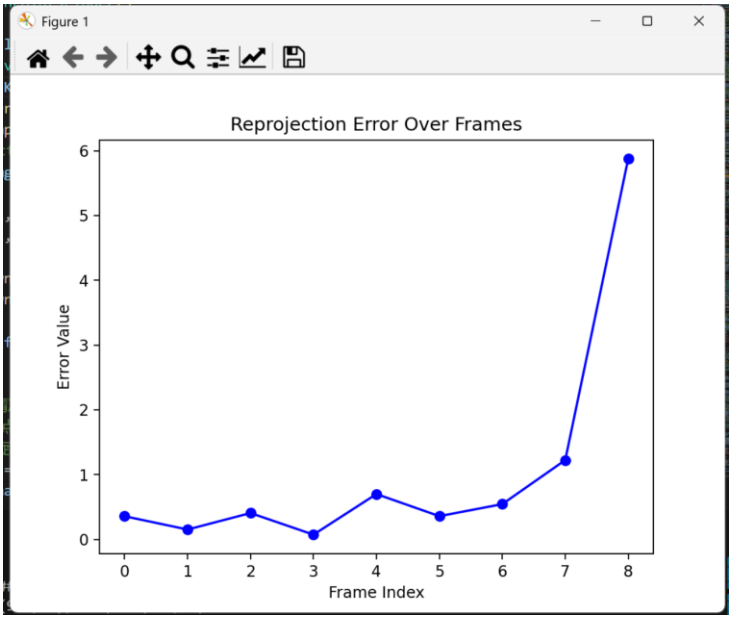

Figure 16 : Erreur max ~6, mieux que 10 pour le non downscale

Le nuage affiche moins d'outliers :

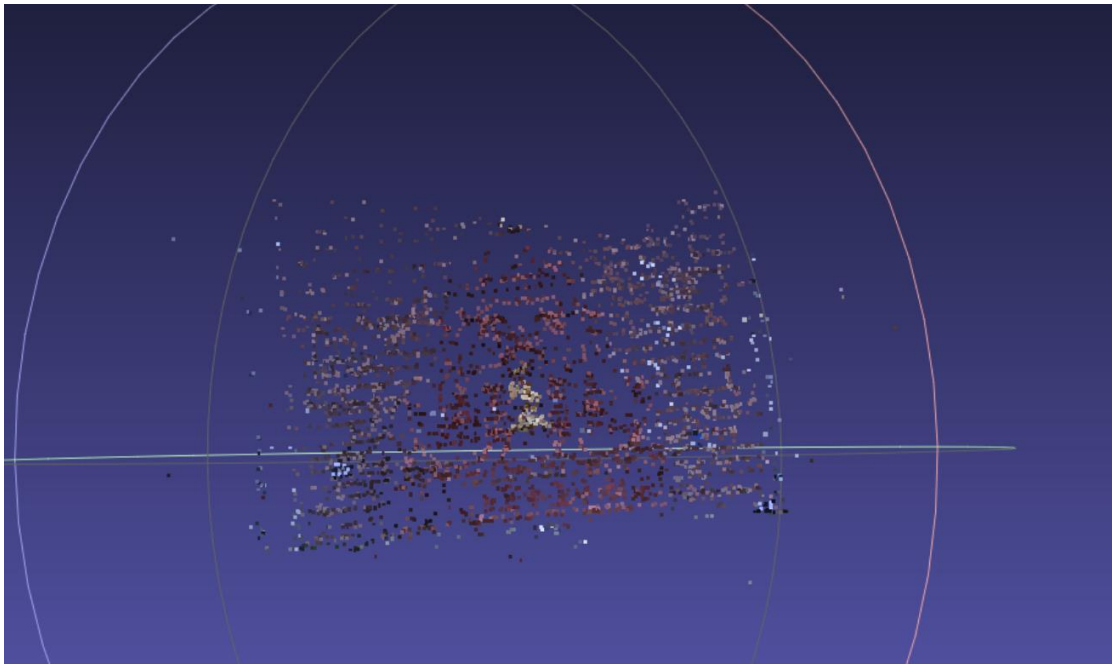

Figure 17 : Nuage plus propre, downscale sans BA

Analyse :

- Le sous-échantillonnage réduit la probabilité de faux matches et accélère grandement le traitement (4 s vs 15 s).
- Sans BA, l'erreur finale (~6 px) reste imparfaite, mais déjà meilleure qu'en haute résolution (~10 px).

Conclusion (sans BA) :

- **Non downscale** : plus long (15 s), davantage d'erreurs en fin de séquence (10 px).
- **Downscale** : plus rapide (4 s), moins d'erreurs (6 px).
- Globalement, sans BA, réduire la résolution aide à contenir les faux appariements et accélère la reconstruction, mais ne résout pas entièrement la dérive finale.

5.5.2 Avec BA : comparer « non downscale » vs « downscale »

On active désormais le BA et on refait la comparaison.

(A) Non downscale + BA

- Pas de réduction de l'image, BA appliqué après chaque (ou plusieurs) itérations.
- **Résultat** :
    - Le temps de calcul peut monter à ~14 min, car on manipule un grand nombre de features et de variables dans le BA,
    - L'erreur finale est très basse (0.008~0.02 pixel),
    - Le nuage est très détaillé (haute densité).

Les **Figures 18 à 20** illustrent la console (14 min), l'évolution de l'erreur (0.008~0.02) et le nuage :

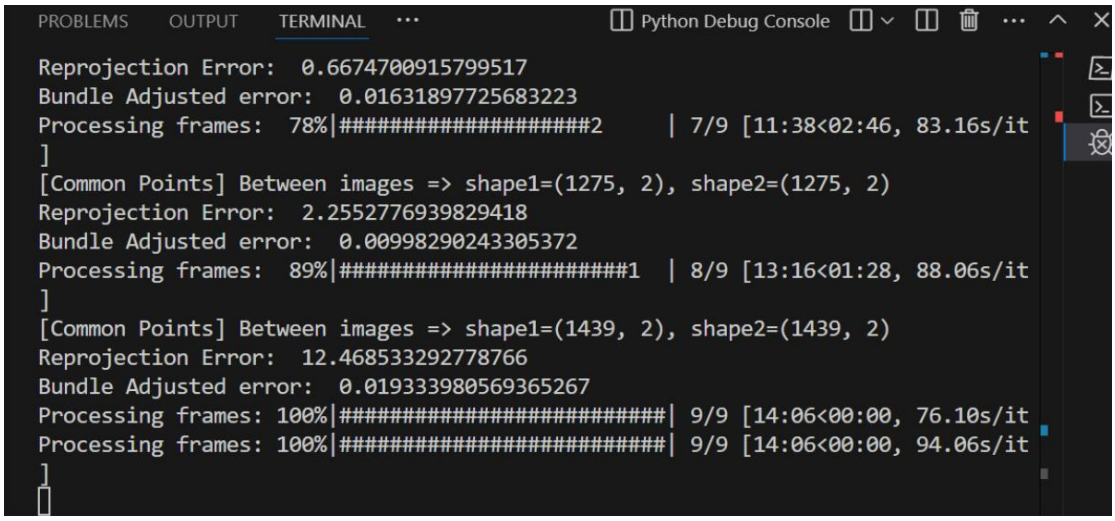

Figure 18 : 14 minutes de calcul, console

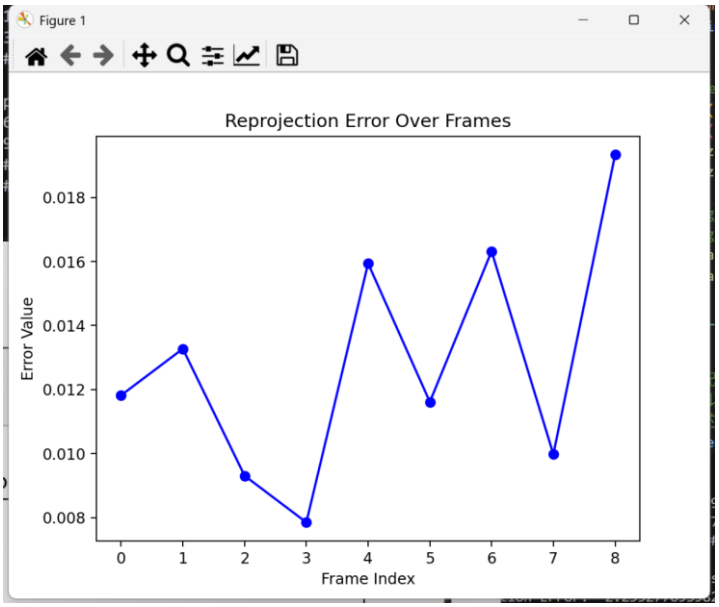

Figure 19 : Erreur 0.008~0.02, très précise

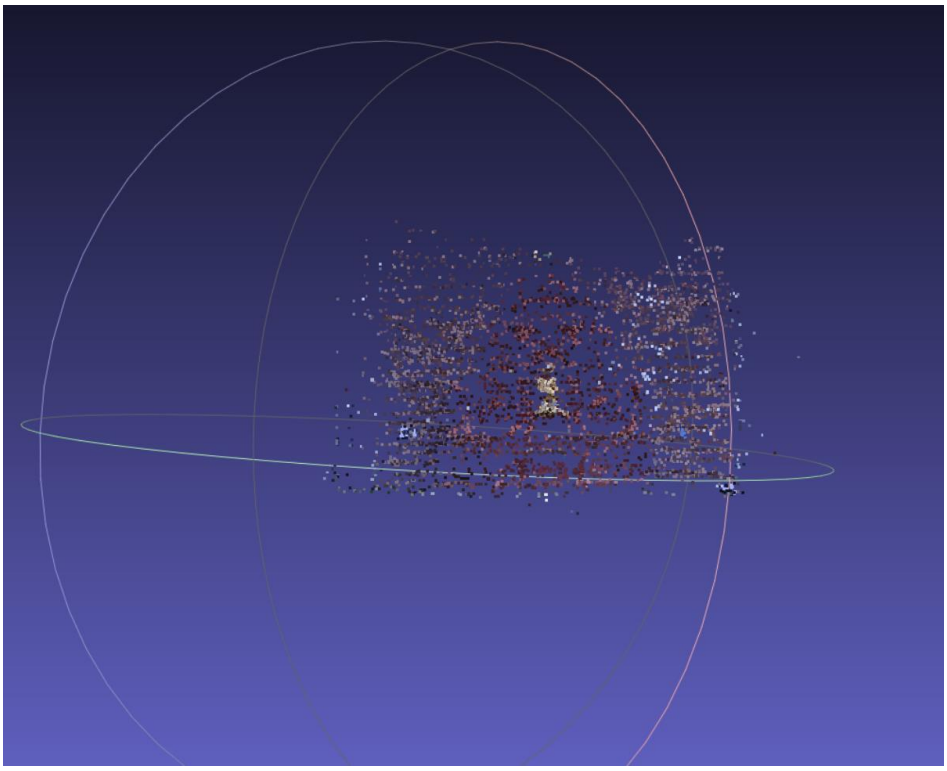

Figure 20 : Nuage dense, haute résolution + BA

(B) Downscale + BA

- On réduit par 2 la résolution (et K), on applique PnP + BA.
- **Résultat** :
    - Temps d'environ 5 min 29 s, nettement plus court que 14 min,
    - Erreur finale 0.015 ~ 0.055 pixel, un peu supérieure à la version haute résolution, mais tout de même stable,
    - Le nuage est légèrement moins détaillé, mais suffisant pour la plupart des usages.

Les **Figures 21 - 23** montrent respectivement les ~5 minutes 30 de calcul, la courbe d'erreur oscille et le nuage final :

```
Reprojection Error:  0.6657589730090878
Processing frames:  78%|####################2     | 7/9 [00:11<00:03,  1.66s/it]
[Common Points] Between images => shape1=(1275, 2), shape2=(1275, 2)
Reprojection Error:  2.1042771592145932
Processing frames:  89%|#######################1  | 8/9 [00:13<00:01,  1.67s/it]
[Common Points] Between images => shape1=(1431, 2), shape2=(1431, 2)
Reprojection Error:  9.826271493173936
Processing frames: 100%|########################| 9/9 [00:15<00:00,  1.70s/it]
```

Figure 21 : Enregistrement partiel montrant une durée d'environ 5 minutes 30

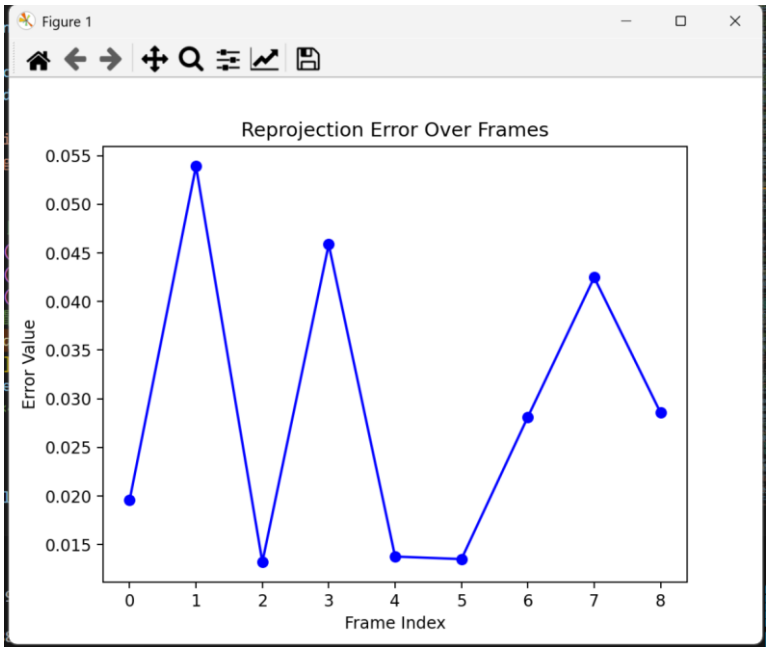

Figure 22 : Downscale + BA : la courbe d'erreur oscille entre 0,015 et 0,055

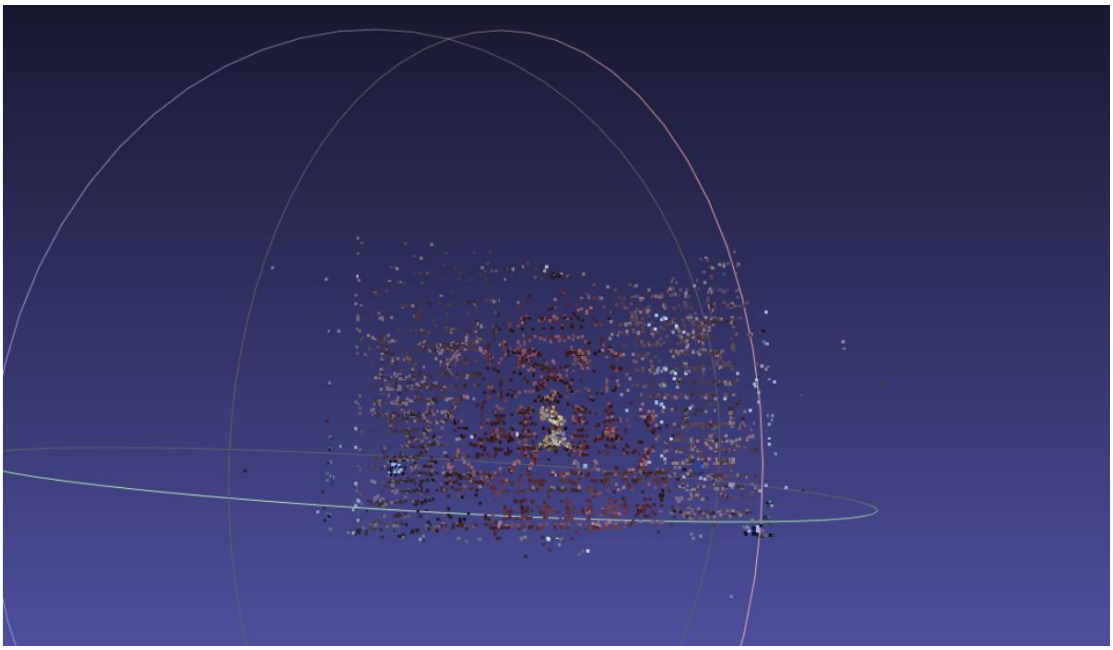

Figure 23 : Nuage obtenu en downscale + BA, moins dense et détaillé qu'en haute résolution

Conclusion (avec BA) :

- **Non downscale + BA** : excellente précision (<0.02 pixel), mais un temps très long (14 min).
- **Downscale + BA** : environ 5 min 30 s, la précision descend légèrement (0.015 ~ 0.055), mais reste largement acceptable.

5.5.3 Discussion générale

En combinant « BA ou pas » et « downscale ou pas », on obtient quatre variantes. Selon les tests :

1. **Sans BA**
    - Non downscale → 15 s, +d'erreurs, ~10 px,
    - Downscale → 4 s, moins d'erreurs, ~6 px.

2. **Avec BA**
    - Non downscale → ~14 min, précision <0.02 px,
    - Downscale → ~5 min 30 s, précision 0.015 ~ 0.055 px.

On note un compromis évident entre la **résolution**, l'**erreur finale**, le **temps de calcul** et la **quantité d'outliers**.

- **Pour un résultat rapide**, un downscale + BA est souvent très satisfaisant.
- **Pour une précision extrême**, on peut rester en pleine résolution + BA, au prix d'un coût de calcul bien plus long.

Selon le contexte (temps réel vs. haute fidélité), il est donc opportun de jouer sur le facteur de downscale, la fréquence du BA, voire un traitement GPU, afin d'optimiser le rapport temps/précision.

6. Conclusion et perspectives

Dans ce projet, nous avons conçu et expérimenté un pipeline de reconstruction 3D multi-vues (détection et appariement de caractéristiques, estimation de pose, triangulation, ajustement en faisceau). À travers diverses comparaisons, nous concluons :

- **SIFT Ratio Test** :
  Il élimine efficacement les correspondances ambiguës en cas de zones répétitives, réduisant nettement les outliers.
- **Bundle Adjustment (BA)** :
  Il corrige la dérive cumulative, améliorant significativement la précision, et demeure une étape cruciale dans la plupart des approches SLAM ou SfM.
- **Downscale vs. non-downscale** :
  - Le downscale accélère sensiblement le calcul, diminue la probabilité de faux appariements et demeure suffisant dans de nombreux scénarios,
  - La pleine résolution est idéale pour des besoins de précision ou de détails extrêmes, mais au prix d'un coût de calcul très élevé (surtout avec BA).

En pratique, il convient d'adapter la méthode à la tâche : niveau de détail requis, ressources matérielles, contrainte temps, etc. (par exemple en modulant le ratio de downscale ou la fréquence du BA).

Au-delà de l'approche actuelle (reconstruction parcimonieuse), d'autres évolutions sont possibles :

A. Densification du nuage (MVS)

- **Multi-View Stereo** :
  À partir de poses caméra connues, on peut estimer la profondeur (disparité) à chaque pixel (plusieurs vues), pour obtenir un nuage bien plus dense (ex. OpenMVS, COLMAP).
- **Nettoyage de nuage** :
  Les points très incertains peuvent être filtrés selon un critère de cohérence géométrique ou photométrique.

B. Reconstruction de surface et texturage

- **Maille 3D** :
  Convertir un nuage dense en maillage triangulaire (Poisson Surface Reconstruction ou Ball Pivoting).
- **Texture** :
  Appliquer les images initiales sur la surface pour un rendu réaliste (baking).
- **Export** :
  Produire un modèle .obj ou .ply texturé pour exploitation ultérieure.

C. Optimisations avancées

- **Bouclage et optimisation globale** :
  Si le parcours caméra comporte des boucles, détecter ces « loop closures » permet une optimisation plus large, réduisant le drift.
- **BA global** :
  Au lieu d'un BA local ou incrémental, on peut envisager un ajustement total (sur l'ensemble des images et points).

7. Références bibliographiques

1. R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, Cambridge University Press, 2003.
2. D. Lowe, « Distinctive Image Features from Scale-Invariant Keypoints », *International Journal of Computer Vision*, 60(2), 91–110, 2004.
3. Documentation OpenCV : https://docs.opencv.org/
4. Documentation Scipy optimize : https://docs.scipy.org/doc/scipy/reference/optimize.html
5. Courbon J. et al., « Vision based SLAM for an autonomous helicopter », *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Répertoire A : le code de la classe Sfm.

```
class Sfm():
    def __init__(self, img_dir:str, downscale_factor:float = 2.0) -> None:
        '''
         Initialise un objet Sfm.
         '''
        self.img_obj = Image_loader(img_dir,downscale_factor)
        self.feature_extractor = FeatureExtractor()
        self.triangulator = Triangulator()
        self.pnp_solver = PnPSolver()
        self.reproj_calc = ReprojectionErrorCalculator()
        self.bundle_adjuster = BundleAdjuster(self.reproj_calc)
        self.ply_saver = PlySaver()
        self.common_finder = CommonPointsFinder()

    def __call__(self, enable_bundle_adjustment:boolean=True):#True ou False
        cv2.namedWindow('image', cv2.WINDOW_NORMAL)
        pose_array = self.img_obj.K.ravel()
        transform_matrix_0 = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0]])
        transform_matrix_1 = np.empty((3, 4))
        #    Matrice de projection
        pose_0 = np.matmul(self.img_obj.K, transform_matrix_0)
        pose_1 = np.empty((3, 4))
        total_points = np.zeros((1, 3))
        total_colors = np.zeros((1, 3))

        #image_0 = self.img_obj.downscale_image(cv2.imread(self.img_obj.image_list[0]))
        #image_1 = self.img_obj.downscale_image(cv2.imread(self.img_obj.image_list[1]))
        image_0 = cv2.imread(self.img_obj.image_list[0])
        image_1 = cv2.imread(self.img_obj.image_list[1])

        feature_0, feature_1 = self.feature_extractor.find_features(image_0, image_1)

        # Matrice essentielle : estimation par RANSAC.

        essential_matrix, em_mask = cv2.findEssentialMat(feature_0, feature_1, self.img_obj.K,
method=cv2.RANSAC, prob=0.999, threshold=0.4, mask=None)
        feature_0 = feature_0[em_mask.ravel() == 1]
        feature_1 = feature_1[em_mask.ravel() == 1]

        # recoverPose renvoie R, t à partir de la matrice essentielle (il existe 4 solutions possibles).
```

```
        _, rot_matrix, tran_matrix, em_mask = cv2.recoverPose(essential_matrix, feature_0, feature_1, self.img_obj.K)
        feature_0 = feature_0[em_mask.ravel() > 0]
        feature_1 = feature_1[em_mask.ravel() > 0]
        # Mise à jour de la pose en cumulant la rotation/translation sur la pose précédente.


        # R1•=RR0
        transform_matrix_1[:3, :3] = np.matmul(rot_matrix, transform_matrix_0[:3, :3])
        #• t1•=t0•+R0T•t
                    transform_matrix_1[:3, 3] = transform_matrix_0[:3, 3] + np.matmul(transform_matrix_0[:3, :3], tran_matrix.ravel())
        # P=K[R|t]
        pose_1 = np.matmul(self.img_obj.K, transform_matrix_1)
        # Triangulation pour obtenir un nuage de points 3D.
        feature_0, feature_1, points_3d = self.triangulator.triangulation(pose_0, pose_1, feature_0, feature_1)
        # Calcul de l'erreur de reprojection.
            error, points_3d = self.reproj_calc.reprojection_error(points_3d, feature_1, transform_matrix_1, self.img_obj.K, homogenity = 1)
        #ideally error < 1
        print("REPROJECTION ERROR: ", error)
        # PnP pour affiner la pose caméra et filtrer les outliers avec RANSAC.

        _, _, feature_1, points_3d, _ = self.pnp_solver.PnP(points_3d, feature_1, self.img_obj.K, np.zeros((5, 1), dtype=np.float32), feature_0, initial=1)

        total_images = len(self.img_obj.image_list) - 2
        pose_array = np.hstack((np.hstack((pose_array, pose_0.ravel())), pose_1.ravel()))

        threshold = 0.5
        errors = []

        for i in tqdm(range(total_images), desc="Processing frames", ascii=True, ncols=80):
            #image_2 = self.img_obj.downscale_image(cv2.imread(self.img_obj.image_list[i + 2]))
            image_2 = cv2.imread(self.img_obj.image_list[i + 2])
            features_cur, features_2 = self.feature_extractor.find_features(image_1, image_2)

            if i != 0:
                feature_0, feature_1, points_3d = self.triangulator.triangulation(pose_0, pose_1, feature_0, feature_1)
                feature_1 = feature_1.T
                points_3d = cv2.convertPointsFromHomogeneous(points_3d.T)
                points_3d = points_3d[:, 0, :]
```

```
                cm_points_0, cm_points_1, cm_mask_0, cm_mask_1 = self.common_finder.common_points(feature_1, features_cur, features_2)
            cm_points_2 = features_2[cm_points_1]
            cm_points_cur = features_cur[cm_points_1]

                rot_matrix, tran_matrix, cm_points_2, points_3d, cm_points_cur = self.pnp_solver.PnP(points_3d[cm_points_0], cm_points_2, self.img_obj.K, np.zeros((5, 1), dtype=np.float32), cm_points_cur, initial = 0)
            transform_matrix_1 = np.hstack((rot_matrix, tran_matrix))
            pose_2 = np.matmul(self.img_obj.K, transform_matrix_1)
            # Calcul de l'erreur de reprojection avant la triangulation suivante
                error, points_3d = self.reproj_calc.reprojection_error(points_3d, cm_points_2, transform_matrix_1, self.img_obj.K, homogenity = 0)

             # Triangulation supplémentaire pour obtenir plus de points 3D à partir des vues successives
             cm_mask_0, cm_mask_1, points_3d = self.triangulator.triangulation(pose_1, pose_2, cm_mask_0, cm_mask_1)
                error, points_3d = self.reproj_calc.reprojection_error(points_3d, cm_mask_1, transform_matrix_1, self.img_obj.K, homogenity = 1)
            print("Reprojection Error: ", error)
            pose_array = np.hstack((pose_array, pose_2.ravel()))
            # Si l'ajustement de faisceau est activé, on l'effectue ici.
            # Attention : cela peut prendre beaucoup de temps.

            if enable_bundle_adjustment:
                                points_3d, cm_mask_1, transform_matrix_1 = self.bundle_adjuster.bundle_adjustment(points_3d, cm_mask_1, transform_matrix_1, self.img_obj.K, threshold)
                pose_2 = np.matmul(self.img_obj.K, transform_matrix_1)
                    error, points_3d = self.reproj_calc.reprojection_error(points_3d, cm_mask_1, transform_matrix_1, self.img_obj.K, homogenity = 0)
                print("Bundle Adjusted error: ",error)
                total_points = np.vstack((total_points, points_3d))
                points_left = np.array(cm_mask_1, dtype=np.int32)
                color_vector = np.array([image_2[l[1], l[0]] for l in points_left])
                total_colors = np.vstack((total_colors, color_vector))
            else:
                total_points = np.vstack((total_points, points_3d[:, 0, :]))
                points_left = np.array(cm_mask_1, dtype=np.int32)
                color_vector = np.array([image_2[l[1], l[0]] for l in points_left.T])
                total_colors = np.vstack((total_colors, color_vector))
```

```
            transform_matrix_0 = np.copy(transform_matrix_1)
            pose_0 = np.copy(pose_1)
            errors.append(error)

            image_0 = np.copy(image_1)
            image_1 = np.copy(image_2)
            feature_0 = np.copy(features_cur)
            feature_1 = np.copy(features_2)
            pose_1 = np.copy(pose_2)
            if cv2.waitKey(1) & 0xff == ord('q'):
                break

        plt.plot(range(len(errors)), errors, 'o-', color='blue')
        plt.xlabel("Frame Index")
        plt.ylabel("Error Value")
        plt.title("Reprojection Error Over Frames")
        plt.savefig("reprojection_error_plot.png", dpi=200)
        plt.show()

        cv2.destroyAllWindows()

        print("Printing to .ply file")
        print(total_points.shape, total_colors.shape)
        self.ply_saver.to_ply(self.img_obj.path, total_points, total_colors,self.img_obj.image_list)
        print("Completed Exiting ...")
            np.savetxt(self.img_obj.path + '\\res\\' + self.img_obj.image_list[0].split('\\')[-
2]+'_pose_array.csv', pose_array, delimiter = '\n')
```

Répertoire 2 : le code de la classe Image_loader.

```
class Image_loader():
    def __init__(self, img_dir:str, downscale_factor:float):
        #    Chargement des paramètres intrinsèques de la caméra K
        with open(img_dir + '\\K.txt') as f:
            # Conversion de la matrice intrinsèque en matrice NumPy de float
                self.K = np.array(list((map(lambda x:list(map(lambda x:float(x), x.strip().split('
'))),f.read().split('\n')))))
            self.image_list = []
        # Chargement de l'ensemble d'images
        for image in sorted(os.listdir(img_dir)):
            if image[-4:].lower() == '.jpg' or image[-5:].lower() == '.png':
                self.image_list.append(img_dir + '\\' + image)

        self.path = os.getcwd()
        self.factor = downscale_factor
        # Un sous-échantillonnage a été effectué sur les images, ce qui réduit leur taille.
        # Par conséquent, la distance focale doit être réduite de la même proportion
        # et les coordonnées du point principal doivent également être mises à l'échelle.
        #self.downscale()



    def downscale(self) -> None:
        '''
        Réduit les paramètres intrinsèques de l'image en fonction du facteur de réduction.
        '''
        self.K[0, 0] /= self.factor
        self.K[1, 1] /= self.factor
        self.K[0, 2] /= self.factor
        self.K[1, 2] /= self.factor

    def downscale_image(self, image):
        # Si factor = 2, la boucle s'exécute une fois; si factor = 4, la boucle s'exécute deux fois.
        for _ in range(1,int(self.factor / 2) + 1):
            # À chaque itération, la largeur et la hauteur de l'image sont réduites de moitié.
            image = cv2.pyrDown(image)
        return image
```

Répertoire 3 : le code de la classe FeatureExtractor.

```
class FeatureExtractor():

    def find_features(self, image_0, image_1) -> tuple:
        '''
        Détection de caractéristiques à l'aide de l'algorithme SIFT et KNN.
        Renvoie les points-clés (caractéristiques) de image_0 et image_1.
        '''
        # Création du détecteur SIFT
        #sift = cv2.xfeatures2d.SIFT_create()
        self.sift = cv2.SIFT_create()

                key_points_0, desc_0 = self.sift.detectAndCompute(cv2.cvtColor(image_0, cv2.COLOR_BGR2GRAY), None)
                key_points_1, desc_1 = self.sift.detectAndCompute(cv2.cvtColor(image_1, cv2.COLOR_BGR2GRAY), None)
         # Brute-Force Matcher, un appariement exhaustif qui calcule la distance entre deux ensembles de descripteurs.
        bf = cv2.BFMatcher()
        # Pour chaque descripteur dans desc_0, trouve les deux descripteurs les plus proches dans desc_1.
        matches = bf.knnMatch(desc_0, desc_1, k=2)
        feature = []
        #Si la distance entre le meilleur et le deuxième meilleur appariement est trop proche,
         #c'est probablement du bruit. On l'écarte, nous avons besoin de m.distance nettement inférieure à n.distance
        #(test du ratio de Lowe).

        for m, n in matches:
            if m.distance < 0.70 * n.distance:
                feature.append(m)

        #feature = [m for (m, n) in matches]
        # En entrée, deux images. En sortie, deux tableaux NumPy de forme (N,2)
        # représentant les coordonnées des points correspondants dans chaque image.

                return np.float32([key_points_0[m.queryIdx].pt for m in feature]), np.float32([key_points_1[m.trainIdx].pt for m in feature])
```

Répertoire 4 : le code de la classe Triangulator.

```
class Triangulator():
    def triangulation(self, projection_matrix_1, projection_matrix_2, point_2d_1, point_2d_2) -> tuple:
        '''
        Fait la triangulation de points 3D à partir de vecteurs 2D et de matrices de projection.
        Renvoie la matrice de projection de la première caméra, celle de la deuxième caméra et le nuage de points.
        '''
        pts_2d_1 = point_2d_1.T    # (2, N)
        pts_2d_2 = point_2d_2.T    # (2, N)

            pt_cloud = cv2.triangulatePoints(projection_matrix_1,projection_matrix_2,pts_2d_1, pts_2d_2)
        # (pt_cloud / pt_cloud[3]) : coordonnées homogènes -> (coordonnées non homogènes) des points 3D triangulés.
        pt_cloud_4N = pt_cloud / pt_cloud[3]
        points_4d = pt_cloud_4N.T

        mask_valid = (points_4d[:, 2] > 0) & (points_4d[:, 2] < 1e4)
        points_4d_filtered = points_4d[mask_valid]    # (M,4)
        pts_2d_1_filtered = pts_2d_1[:, mask_valid]    # (2, M)
        pts_2d_2_filtered = pts_2d_2[:, mask_valid]

        points_4d_filtered_4N = points_4d_filtered.T

        #return point_2d_1.T, point_2d_2.T, (pt_cloud / pt_cloud[3])
        return pts_2d_1_filtered, pts_2d_2_filtered, points_4d_filtered_4N
```

Répertoire 5 : le code de la classe PnPSolver.

```
class PnPSolver():
    def PnP(self, obj_point, image_point , K, dist_coeff, rot_vector, initial) ->  tuple:
        '''
        Détermine la pose d'un objet à partir de correspondances 3D-2D en utilisant RANSAC.
        Renvoie la matrice de rotation, la matrice de translation, les points d'image,
        les points d'objet et le vecteur de rotation.
        '''
        if initial == 1:
            # (N,1,3) =》(N,3)
            obj_point = obj_point[:, 0 ,:]
            image_point = image_point.T
            rot_vector = rot_vector.T
        _, rot_vector_calc, tran_vector, inlier = cv2.solvePnPRansac(obj_point, image_point, K,
dist_coeff, cv2.SOLVEPNP_ITERATIVE)
        rot_matrix, _ = cv2.Rodrigues(rot_vector_calc)

        if inlier is not None:
            image_point = image_point[inlier[:, 0]]
            obj_point = obj_point[inlier[:, 0]]
            rot_vector = rot_vector[inlier[:, 0]]
        return rot_matrix, tran_vector, image_point, obj_point, rot_vector
```

Répertoire 6 : le code des classes ReprojectionErrorCalculator & BundleAdjuster.

ReprojectionErrorCalculator:

```
class ReprojectionErrorCalculator():
    def reprojection_error(self, obj_points, image_points, transform_matrix, K, homogenity) ->tuple:
        '''
        Calcule l'erreur de reprojection, c'est-à-dire la distance entre les points projetés et les points réels.
        Renvoie l'erreur totale et les points d'objet.
        '''
        rot_matrix = transform_matrix[:3, :3]
        tran_vector = transform_matrix[:3, 3]
        rot_vector, _ = cv2.Rodrigues(rot_matrix)
        # Convertit, si nécessaire, certains points encore au format homogène en coordonnées non homogènes
        if homogenity == 1:
            obj_points = cv2.convertPointsFromHomogeneous(obj_points.T)
        # 3D => 2D
        image_points_calc, _ = cv2.projectPoints(obj_points, rot_vector, tran_vector, K, None)
        image_points_calc = np.float32(image_points_calc[:, 0, :])
        # Assure ici le format (N,2)
        total_error = cv2.norm(image_points_calc, np.float32(image_points.T) if homogenity == 1 else np.float32(image_points), cv2.NORM_L2)
        return total_error / len(image_points_calc), obj_points
    def optimal_reprojection_error(self, obj_points) -> np.array:
        '''
        Calcule l'erreur de reprojection pendant le bundle adjustment.
        Retourne l'erreur.
        '''
        # La fonction least_squares l'appelle de façon itérative sur x0 afin de minimiser l'erreur de reprojection.
        # Elle se base sur les divers paramètres regroupés (pose, intrinsecs, points 2D, points 3D)
        # pour calculer l'erreur de reprojection.
        # Puis la fonction least_squares ou un autre solveur l'appelle de manière itérative pour mettre à jour les paramètres.

        transform_matrix = obj_points[0:12].reshape((3,4))
        K = obj_points[12:21].reshape((3,3))
        rest = int(len(obj_points[21:]) * 0.4)
```

```
        # 40% = points 2D
        p = obj_points[21:21 + rest].reshape((2, int(rest/2))).T
        # 60% = points 3D
        obj_points = obj_points[21 + rest:].reshape((int(len(obj_points[21 + rest:])/3), 3))
        rot_matrix = transform_matrix[:3, :3]
        tran_vector = transform_matrix[:3, 3]
        # Matrice de rotation -> vecteur de rotation
        rot_vector, _ = cv2.Rodrigues(rot_matrix)
        # 3D -> 2D
        image_points, _ = cv2.projectPoints(obj_points, rot_vector, tran_vector, K, None)
        image_points = image_points[:, 0, :]
        # Erreur au carré
        error = [ (p[idx] - image_points[idx])**2 for idx in range(len(p))]
        return np.array(error).ravel()/len(p)
```

BundleAdjuster :

```
class BundleAdjuster():
    def __init__(self, reprojection_error_calculator: ReprojectionErrorCalculator):
        # Nécessite un calculateur d'erreur de reprojection.
        self.reproj_calc = reprojection_error_calculator
    def bundle_adjustment(self, _3d_point, opt, transform_matrix_new, K, r_error) -> tuple:
        '''
        Bundle adjustment for the image and object points
        returns object points, image points, transformation matrix
        '''
        # Visant à minimiser l'erreur de reprojection
        # Optimise (corrige) les paramètres externes de la caméra, les paramètres internes,
        # ainsi que les coordonnées des points 3D/2D.

        opt_variables = np.hstack((transform_matrix_new.ravel(), K.ravel()))
        opt_variables = np.hstack((opt_variables, opt.ravel()))
        opt_variables = np.hstack((opt_variables, _3d_point.ravel()))
                values_corrected = least_squares(self.reproj_calc.optimal_reprojection_error,
opt_variables, gtol = r_error).x
        K = values_corrected[12:21].reshape((3,3))
        rest = int(len(values_corrected[21:]) * 0.4)
        # La fonction renvoie les points 3D, les points 2D et la matrice externe (3×4) de la caméra.

        return values_corrected[21 + rest:].reshape((int(len(values_corrected[21 + rest:])/3), 3)),
values_corrected[21:21 + rest].reshape((2, int(rest/2))).T, values_corrected[0:12].reshape((3,4))
```

Répertoire 7 : le code de la classe PlySaver.

```
class PlySaver():
    def to_ply(self, path, point_cloud, colors, image_list) -> None:
        '''
        Génère un fichier .ply permettant d'ouvrir le nuage de points.
        '''
        out_points = point_cloud.reshape(-1, 3) * 200
        out_colors = colors.reshape(-1, 3)
        print(out_colors.shape, out_points.shape)
        verts = np.hstack([out_points, out_colors])

        # Calcule la distance euclidienne de chaque point au centre de gravité.
        mean = np.mean(verts[:, :3], axis=0)
        scaled_verts = verts[:, :3] - mean
        dist = np.sqrt(scaled_verts[:, 0] ** 2 + scaled_verts[:, 1] ** 2 + scaled_verts[:, 2] ** 2)
        # Exclut les points trop éloignés (outliers).
        indx = np.where(dist < np.mean(dist) + 300)
        #threshold = np.mean(dist) * 3
        #indx = np.where(dist < threshold)

        verts = verts[indx]
        ply_header = '''ply
            format ascii 1.0
            element vertex %(vert_num)d
            property float x
            property float y
            property float z
            property uchar blue
            property uchar green
            property uchar red
            end_header
            '''
        name_part = image_list[0].split('\\')[-2]
        out_file = path + '\\res\\' + name_part + '.ply'
        with open(out_file, 'w') as f:
            f.write(ply_header % dict(vert_num=len(verts)))
            np.savetxt(f, verts, '%f %f %f %d %d %d')
```

Répertoire 8 : le code de la classe CommonPointsFinder.

```
class CommonPointsFinder():
    def common_points(self, image_points_1, image_points_2, image_points_3) -> tuple:
        '''
        Trouve les points communs entre image 1 et 2, et entre image 2 et 3.
        Renvoie les points communs de image 1-2, de image 2-3, ainsi que le masque correspondant
        pour image 1-2 et pour image 2-3.
        '''
        # Ici, on cherche à détecter les mêmes coordonnées de points 2D
        # dans image_points_1 et image_points_2,
        # puis on applique un masque correspondant sur image_points_3.
        # Cela retourne 4 éléments afin de les utiliser ensuite dans la triangulation, PnP, etc.
        cm_points_1 = []
        cm_points_2 = []
        for i in range(image_points_1.shape[0]):
            # L'égalité stricte (==) peut poser des problèmes de précision,
            # il serait préférable d'utiliser une tolérance de distance.

            a = np.where(image_points_2 == image_points_1[i, :])
            if a[0].size != 0:
                cm_points_1.append(i)
                cm_points_2.append(a[0][0])
        # On applique un masque et on compresse image_points_2 et image_points_3
        # en supprimant les indices trouvés.
        mask_array_1 = np.ma.array(image_points_2, mask=False)
        mask_array_1.mask[cm_points_2] = True
        mask_array_1 = mask_array_1.compressed()
        mask_array_1 = mask_array_1.reshape(int(mask_array_1.shape[0] / 2), 2)

        mask_array_2 = np.ma.array(image_points_3, mask=False)
        mask_array_2.mask[cm_points_2] = True
        mask_array_2 = mask_array_2.compressed()
        mask_array_2 = mask_array_2.reshape(int(mask_array_2.shape[0] / 2), 2)
            print(f"\n[Common Points] Between images => shape1={mask_array_1.shape},
shape2={mask_array_2.shape}")
        return np.array(cm_points_1), np.array(cm_points_2), mask_array_1, mask_array_2
```