ÉCOLE NATIONALE SUPÉRIEURE DES MINES DE NANCY
DÉPARTEMENT **ISDP**

Déformation d'objets imprimés en 3D

Projet 2A

Rapport

Groupe B:
Diane DEBORGIES
Jéssica MOURA ALVES
Yves-Gabriel KERISIT

Juin 2017

Sommaire

1 Introduction

Aujourd'hui, il est possible de créer des objets déformables par impression 3D. La pièce à imprimer doit alors satisfaire plusieurs contraintes. Elle doit tout d'abord respecter la forme géométrique imposée en entrée par le fabricant. Elle doit en second lieu respecter un certain nombre de paramètres mécaniques tels que le coefficient d'élasticité, la flexion, la dureté, etc. eux-aussi définis en entrée par le fabricant.

Afin de mesurer la validité ou non de ces contraintes, on doit comparer l'objet théoriquement attendu, et l'objet obtenu après impression.

- ❏ 1ère étape : Reconstruction de la géométrie de l'objet imprimé en 3D:

On va capturer la géométrie exacte de la pièce imprimée grâce à une table micrométrique (sur laquelle sera posé l'objet) et à une caméra (qui photographiera l'objet à intervalles de temps réguliers pendant sa rotation), pour ensuite reconstruire le modèle 3D, et le comparer par rapport au modèle qu'on a donné à l'impression, et en dégager les différences.

- ❏ 2e étape : Détermination des paramètres mécaniques de l'objet réel.

Pour comparer les paramètres mécaniques de la pièce imprimée par rapport à ceux de la pièce attendue, on doit comparer une déformation effectuée sur la pièce imprimée avec une simulation de cette même déformation. Dans ce projet, la comparaison se fait une fois de plus sur la géométrie.

Pour ce faire, on doit d'une part créer un dispositif capable d'appliquer une déformation précise sur l'objet 3D, expérimenter cette déformation, et reconstruire la géométrie déformée en 3D – comme dans la partie 1 –, et d'autre part simuler numériquement une

déformation de l'objet à partir du dessin 2D fourni par le fabricant. Puis nous comparons la simulation de déformation avec la déformation réelle.

Au cours du présent rapport, nous expliquerons comment s'opère la reconstruction de la géométrie d'un objet imprimé en 3D grâce à des méthodes de vision par ordinateur, puis nous traiterons de la mesure et de la simulation de déformation d'un objet imprimé en 3D, puis nous aborderons la division du travail au sein de notre groupe de projet avant de conclure par un bilan sur ce que nous a apporté ce projet.

2 Space Carving

2.1 Explications théoriques

Nous souhaitons reconstruire un objet en 3D à partir d'une série d'images de cet objet, en utilisant uniquement les informations contenues dans lesdites images. Le *space carving* est une méthode de reconstruction d'objet en 3D. Le principe en est le suivant :

On commence par initialiser un volume plein (un cube) qui va couvrir l'ensemble de l'objet. On segmente cet objet en voxels (des pixels en 3D). Puis on teste si chaque voxel est *consistant* dans la série d'images. Pour ce faire :

Pour chaque voxel v_i :

- On projette le voxel v_i dans toutes les images : on note p_{ij} la projection du voxel v_i dans l'image j (p_{ij} est un pixel).
- Dans chaque image j, on évalue la consistance du pixel p_{ij}, ie on vérifie si la couleur du pixel est suffisamment proche de la couleur des pixels de l'objet, auquel cas le pixel p_{ij} serait un des pixels constitutifs de l'objet dans l'image j. Par exemple, on peut utiliser le critère de la variance de couleurs : la variance de la couleur du pixel par rapport à la couleur supposée de l'objet ne doit pas excéder un certain seuil.
- S'il existe une image j pour laquelle p_{ij} n'est pas consistant, cela signifie que le pixel p_{ij} fait partie du fond de l'image, et non pas de l'ensemble des pixels de l'objet. Donc le voxel v_i ne fait pas partie de l'objet, et on supprime le voxel du volume.

- Si pour toutes les images j, le pixel p_{ij} est consistant, alors cela signifie que sur chaque image j, le voxel v_i se projette en un pixel p_{ij} constitutif de l'objet, et donc v_i appartient à l'objet, donc on garde le voxel dans le volume.

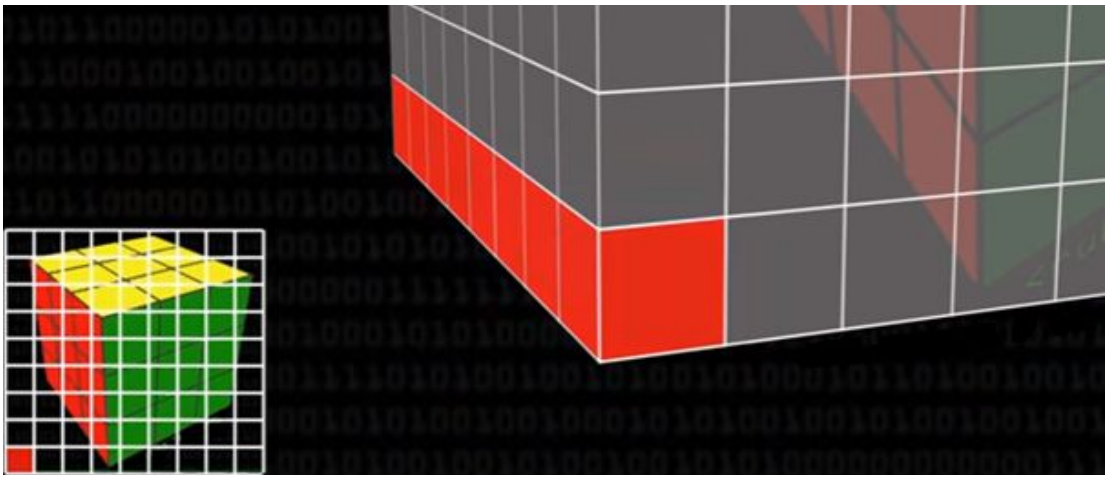

Figure 2.1: Exemple de projection d'une série de voxels dans une des images de la séquence

On répète ce test pour tous les voxels. On obtient à la fin l'objet reconstruit, à partir du volume initial creusé à chaque itération.

La méthode de *space carving* nécessite que les images données en entrée soient calibrées : la calibration des images doit donc être effectuée en amont.

2.2 Géométrie et calibration des caméras

Le modèle géométrique associé au processus de saisie d'image à l'aide d'une caméra est caractérisé par certains paramètres, que nous allons estimer par calibration :

- Les paramètres intrinsèques, qui sont les paramètres propres à la caméra ;
- Les paramètres extrinsèques, qui lient un repère associé à la caméra au repère associé à l'objet étalon (le cube à mire) : le repère monde.

Le modèle d'une caméra est caractérisé à l'aide de deux transformations :

- ❏ Une projection qui transforme un *point objet* de l'espace 3D en un *point image* (en 2D)
- ❏ Une transformation d'un repère *métrique* lié à la caméra, à un repère lié à l'image.

2.3 La projection

Soit 0 un point du plan image, appelé *point principal*. On appelle *axe optique* la droite perpendiculaire au plan image passant par 0. Soit F un point de l'axe optique, situé à une distance f du plan image. On appelle F le *centre de projection*, et f la *distance focale*.

Dans le repère de la caméra (F, x, y, z): le plan xFy est parallèle au plan image, et l'axe z est confondu avec l'axe optique.

Soit $B\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ un point objet. Le point B se projette dans le plan image le long d'une droite passant par B et F. Les coordonnées de la projection b de B dans le plan image sont, dans le repère caméra : $\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \frac{fx}{z} \\ \frac{fy}{z} \\ f \end{pmatrix}$.

2.4 Transformation caméra/image

- ❏ Comment passer du repère caméra au repère image ?

Les points image sont mesurés en pixels dans un repère (u, v) bidimensionnel associé à l'image. On note k_u le facteur d'échelle vertical (en pixels/mm) et k_v le facteur d'échelle horizontal (en pixels/mm). On a potentiellement , car les pixels sont rarement carrés.

Notons $\begin{pmatrix} u_0 \\ v_0 \end{pmatrix}$ les coordonnées (en pixels) de dans le repère image (centre optique). Alors les coordonnées $\begin{pmatrix} u \\ v \end{pmatrix}$ de b dans le repère image s'écrivent : $\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} -k_u x' + u_0 \\ k_v y' + v_0 \end{pmatrix}$

Donc la relation entre les coordonnées caméra de et les coordonnées image de est :

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} -k_u x' + u_0 \\ k_v y' + v_0 \end{pmatrix} = \begin{pmatrix} -k_u \frac{fx}{z} + u_0 \\ k_v \frac{fy}{z} + v_0 \end{pmatrix}.$$

2.5 Les paramètres intrinsèques à la caméra

En posant $a_u = -k_f$ et $a_v = k_v f$, on obtient finalement : $\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \alpha_u \frac{x}{z} + u_0 \\ \alpha_v \frac{y}{z} + v_0 \end{pmatrix}$

Si maintenant on introduit les coordonnées caméra sans dimension : $\begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix} = \begin{pmatrix} \frac{x}{z} \\ \frac{y}{z} \\ 1 \end{pmatrix}$, alors la relation entre les coordonnées image et les coordonnées caméra s'écrit : $\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \alpha_u x_c + u_0 \\ \alpha_v y_c + v_0 \end{pmatrix}$, ie : $\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K \begin{pmatrix} x_c \\ y_c \\ 1 \end{pmatrix}$ avec $K = \begin{pmatrix} \alpha_u & 0 & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{pmatrix}$

Ce modèle comporte donc 4 paramètres : $a_u,\ a_v,\ u_o,\ v_o$. Ces paramètres, intrinsèques à la caméra, vont être estimés par calibration.

2.6 Les paramètres extrinsèques

Afin de déterminer les paramètres du modèle de la caméra (la relation entre le repère monde et le repère caméra), on place devant la caméra un objet étalon : un cube avec une mire. Cette mire est un ensemble de points dont les coordonnées sont parfaitement connues dans le repère de la mire (coordonnées en mm). Chaque point de la mire se projette dans l'image, et on mesure ses coordonnées dans le repère image.

On va décomposer la transformation mire/image en deux transformations :

- ❏ Une transformation mire/caméra
- ❏ Une transformation caméra/image

La transformation mire/caméra se compose d'une rotation et d'une translation. En notant $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ les coordonnées d'un point dans le repère caméra, et $\begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$ les coordonnées de ce même point dans le repère mire, on a : $\begin{pmatrix} x \\ y \\ z \end{pmatrix} = R \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + T$, avec R une matrice de rotation (3x3), et T une matrice de translation (3x1).

La transformation caméra/image a déjà été traitée précédemment.

Ainsi, nous allons chercher à estimer les matrices de rotation R et de translation T, qui constituent les paramètres extrinsèques.

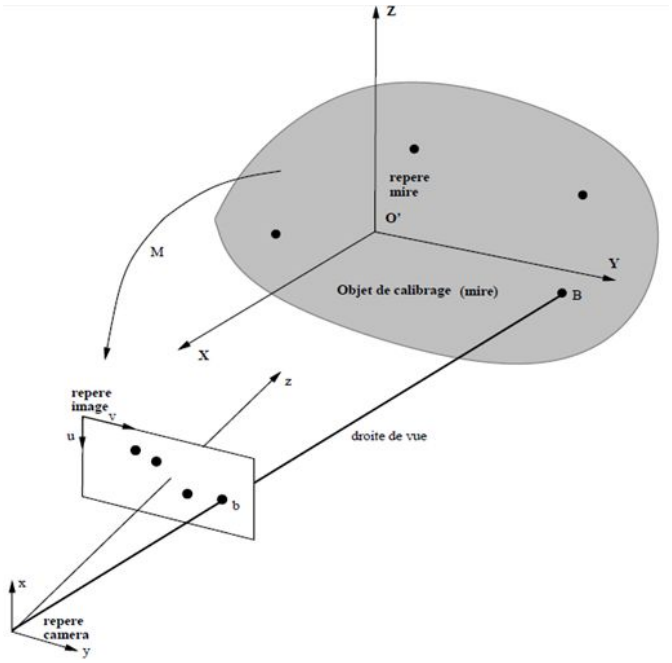

Figure 2.2: schéma explicatif du principe de calibration

Une fois que nous avons obtenu tous ces paramètres (intrinsèques, et extrinsèques pour chaque image), nous pouvons procéder à la reconstruction de l'objet en 3D.

3 Reconstruction Géométrique

La reconstruction de l'objet en 3D est faite à partir des photos de l'objet imprimé en 3D et du logiciel python, grâce aux modules de la bibliothèque **OpenCV**. **OpenCV** est une bibliothèque proposant des algorithmes pour la vision par ordinateur. Elle donne notamment accès au module **calib3d**, qui contient des fonctions permettant de reconstruire un objet en 3D à partir d'images acquises avec plusieurs caméras simultanément.

La partie photographique est composée d'une table robotique qui permet d'effectuer une rotation, autour d'un axe vertical par rapport à l'objet imprimé, pendant qu'une caméra fixe le filme. La figure 3.1 montre le schéma du processus de capture des images.

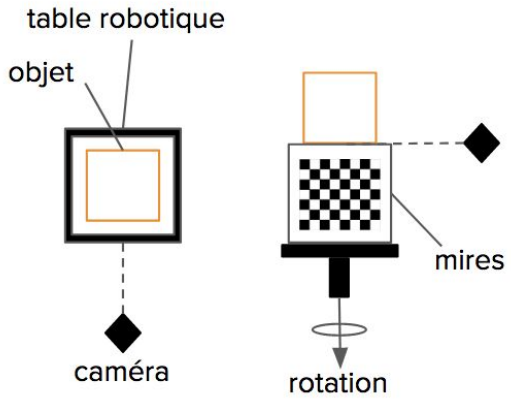

Figure 3.1: schéma du dispositif expérimental

À l'aide du logiciel python et principalement de sa bibliothèque **OpenCV**, nous avons créé des scripts ayant chacun une fonction pour la reconstruction. Les scripts sont les suivants:

1. **Chessboard:** il identifie les coins (corners) de la mire et les sauvegarde;
2. **Viewcorners:** il permet de visualiser les points détectés superposés à une image;

3. **Calib:** il permet de calibrer une série d'images dont les mires ont été détectées: il construit une matrice avec les paramètres intrinsèques de la caméra et les extrinsèques (informations de rotation et translation de l'objet) pour chaque image.
4. **Carve:** il reconstruit la pièce par space carving à partir de la série d'images calibrées.

Les trois premiers scripts sont utilisés pour l'étape de calibration, où l'on doit connaître précisément la position de la caméra par rapport à l'objet, et le dernier implémente la méthode dite de "space carving", c'est-à-dire, l'étape de sculpture.

Pour avoir une reconstruction de l'objet précise, il faut travailler sur la calibration. Pendant notre année de projet, nous avons développé des mécanismes pour essayer d'améliorer la reconstruction de l'objet final.

3.1 Calibration avec une mire

La première méthode utilisée a été la reconstruction avec une seule mire de référence. La figure 3.2 montre quelques photos prises pendant l'étape expérimentale.

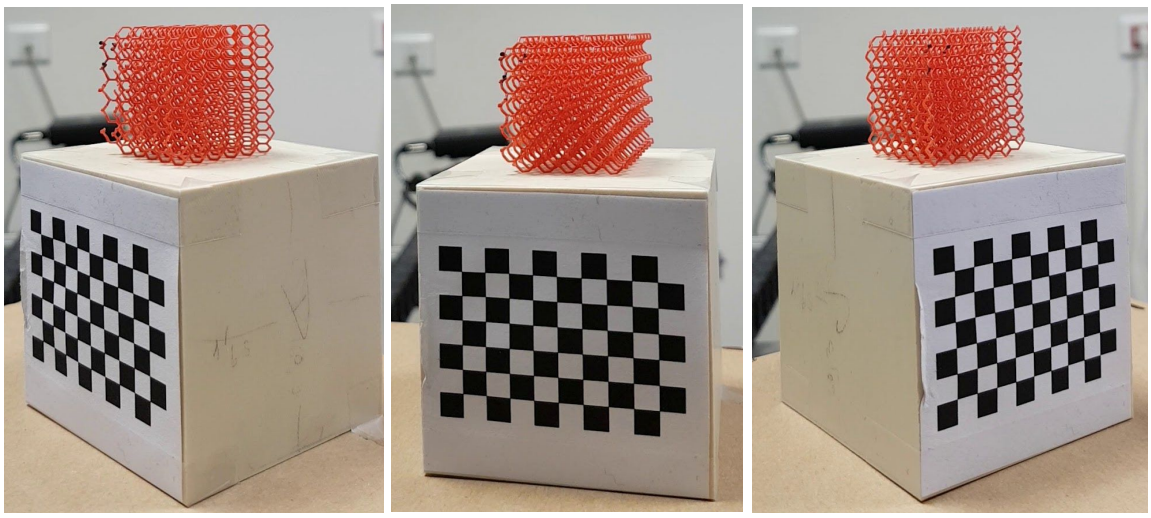

Figure 3.2: photos capturées par la caméra

Premièrement, on commence par détecter des points de la mire avec le script **chessboard**. Dans cette étape, on va sélectionner uniquement les images pour lesquelles le code est capable de lire tous les coins et les sauvegarder comme fichier *.npy*. Pour s'assurer que les mires ont été bien identifiées, on utilise le script **viewcorner** pour afficher les coins détectés pour chaque image, et on obtient des résultats comme dans la figure 3.3.

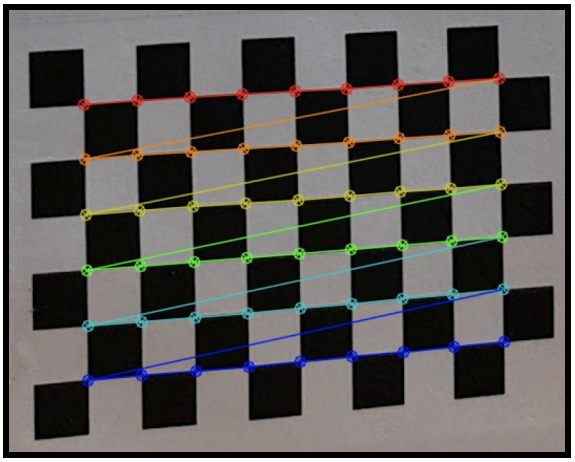

Figure 3.3: output du script viewcorner

Ensuite on applique le script **calib**, à partir des images *.npy* acquises dans le **chessboard**, pour trouver la localisation de la caméra et fournir la matrice de translation et rotation pour chaque image.

La matrice d'intrinsèques obtenue est la suivante:

$$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1.71265402e+03 & 0.00000000e+00 & 9.12635716e+02 \\ 0.0000000e+00 & 1.70964939e+03 & 5.43113761e+02 \\ 0.0000000e+00 & 0.0000000e+00 & 1.0000000e+00 \end{bmatrix}$$

f_x = 1.71265402e+03

f_y = 1.70964939e+03

c_x = 9.12635716e+02

c_y = 5.43113761e+02

Où f_x et f_y sont les distances focales de la caméra et (c_x, c_y) sont les centres optiques exprimés en coordonnées pixels.

Comme résultat du code calib, on obtient également les fichiers *.calib* pour chaque image, où se trouve une matrice 4x4: en haut, la matrice de rotation 3x3 et en bas, le vecteur de translation 1x3.

Pour finir, on part des images *.calib* pour faire la reconstruction à l'aide du script **carve**. Le résultat obtenu par cet échelon est dans la figure 3.4.

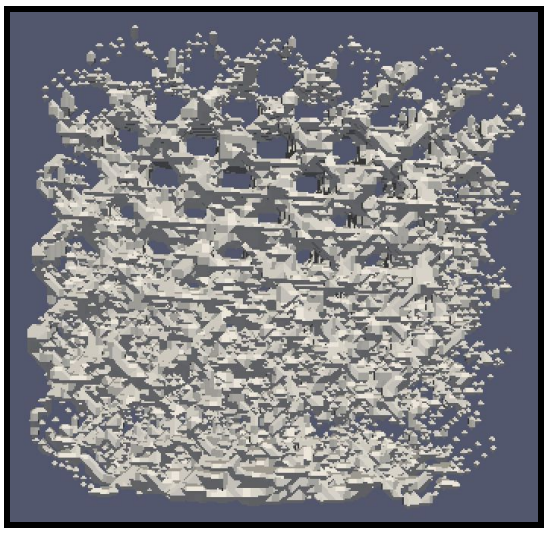

Figure 3.4: reconstruction de l'objet en 3D

Le résultat final est un cube, comme prévu, cependant il manque encore des détails. On en a conclu qu'en ayant travaillé avec une seule mire, on n'a pas réussi à afficher tous les côtés de l'objet, parce que les photos ne donnent pas la rotation complète (360°). Il faut donc augmenter le nombre de mires de calibration pour prendre en compte tous les côtés du cube.

Avec cette motivation, nous sommes partis sur une deuxième méthode de calibration: nous avons utilisé 4 mires de calibration.

3.2 Calibration avec quatre mires

Afin d'augmenter la précision de la reconstruction (ie pour se rapprocher au plus près de la géométrie réelle de l'objet), nous allons maintenant utiliser plusieurs mires de calibration au lieu d'une seule, afin d'afficher tous les côtés de l'objet, c'est-à-dire, capturer une rotation complète de 360°.

Ainsi, dans la partie précédente, nous avions une mire de calibration, les données de la mire sont dans le tableau 3.1.

Tableau 3.1: Données de la mire

Taille du maillage (nombre de coins du maillage, respectivement horizontalement et verticalement)	(9,6)
Taille d'un carré du maillage (en mm)	5

Maintenant, nous disposons de 4 mires, dont les caractéristiques sont dans le tableau suivant :

Tableau 3.2: Données des quatre mires

	Mire 1	**Mire 2**	**Mire 3**	**Mire 4**
Taille du maillage	(12,10)	(11,9)	(10,8)	(9,7)
Taille d'un carré du maillage (en mm)	4	4	5	5

Les 4 mires sont positionnées comme montre la figure 3.5 :

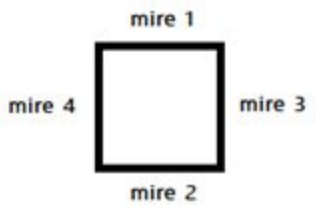

Tableau 3.5: Cube comportant les mires de calibration (vue de dessus)

3.2.1. Les changements opérés dans les scripts précédents

A présent, nous devons modifier les programmes **chessboard2Mire.py** et **calib.py** pour prendre en compte toutes les mires.

D'abord, le script **chessboard** a été modifié pour pouvoir prendre les points de toutes les mires. Un point qu'il est important de souligner est le changement du nombre d'images entre l'état initial (images format .jpg) et l'état après détection des coins dans les mires (tableaux format .npy). Avec une mire, le nombre de tableaux .npy est toujours plus petit que le nombre d'images en .jpg, parce que le code va sélectionner uniquement les images qui permettent de bien visualiser les coins. Avec quatre mires, le processus est le même, mais nous avons également des images qui détectent deux mires en même temps (chevauchement). Donc, pour ces images-ci, le code va donner deux tableaux au format .npy: une pour chaque mire. C'est pour cette raison que l'ensemble de tableaux numpy sera plus grand que le nombre d'images de la série en .jpg.

Le script suivant, **viewcorner**, n'a pas besoin de modification. Comme sa fonction est seulement d'afficher les coins, l'unique chose qui change est la taille de la mire donnée en entrée.

A partir de ces programmes, la calibration de la séquence d'images du cube nous permet d'obtenir une matrice des intrinsèques, et pour chaque image C_i la matrice de passage (R_i^J, T_i^J) qui permet de passer du repère de la mire J (la mire détectée dans l'image C_i) au repère de la caméra i. Cependant, pour pouvoir appliquer la méthode de reconstruction par *space carving*, nous devons exprimer toutes les matrices (R_i^J, T_i^J) dans le même repère de mire. Cela fait l'objet du programme **passage.py**.

Nous avons deux formules de transformation des matrices :

- Inverse : $(R, T)^{-1} = (R^t, -R^t.T)$ avec R^t la transposée de R ;

- Composition : $(R^B, T^B) * (R^A, T^A) = (R^B R^A, R^B T^A + T^B) =$ avec A, B les mires détectées dans les images considérées.

Ainsi, à partir des formules d'inversion et de composition, nous pouvons exprimer la matrice de passage du repère d'une mire A au repère d'une mire B :

$$(R_{B\leftarrow A}, T_{B\leftarrow A}) = (R^B, T^B)^{-1} * (R^A, T^A)$$
$$= \left(R^{B^t}, -R^{B^t} T^B\right) * (R^A, T^A)$$
$$= (R^{B^t} R^A, R^{B^t} T^A - R^{B^t} T^B)$$
$$= (R^{B^t} R^A, R^{B^t} (T^A - T^B))$$

Donc $(R_{B\leftarrow A}, T_{B\leftarrow A}) = (R^{B^t} R^A, R^{B^t} (T^A - T^B))$.

Pour pouvoir déterminer ces matrices de passage, nous allons utiliser les images qui se chevauchent, ie celles sur lesquelles deux mires ont pu être détectées. En théorie, nous devrions obtenir une unique matrice de passage du repère d'une mire A au repère d'une mire B pour toutes les images i détectant à la fois les mires A et B :

$$(R_{B\leftarrow A}, T_{B\leftarrow A}) = \left(R_{i_1}^{B^t} R_{i_1}^A, R_{i_1}^{B^t} (T_{i_1}^A - T_{i_1}^B)\right) = \left(R_{i_2}^{B^t} R_{i_2}^A, R_{i_2}^{B^t} (T_{i_2}^A - T_{i_2}^B)\right) = \cdots$$

Cependant, même si la matrice de passage $(R_{B\leftarrow A,}\, T_{B\leftarrow A})$ devrait être la même que l'on passe par la caméra en position i_1, i_2, etc., en pratique il y a tout de même une faible erreur entre les matrices $(R_{i\,B\leftarrow A,}\, T_{i\,B\leftarrow A})$ obtenues pour chaque image détectant les mires A et B.

De ce fait, nous allons devoir effectuer une moyenne sur les $(R_{i\,B\leftarrow A,}\, T_{i\,B\leftarrow A})$ obtenues pour déterminer la matrice $(R_{B\leftarrow A,}\, T_{B\leftarrow A})$. Nous avons effectué plusieurs séries de tests, en faisant des moyennes :

❑ Sur les $\left(R_{i_1}^{B\,t} R_{i_1}^{A}, R_{i_1}^{B\,t}\left(T_{i_1}^{A} - T_{i_1}^{B}\right)\right)$ de chaque image :

Cependant, toutes les matrices $R_{i_1\ B\leftarrow A}$ sont des matrices de rotation, mais $R_{B\leftarrow A} = \frac{1}{card(\{i_1, i_2, \ldots\})} \sum_{i}^{n} R_{i\ B\leftarrow A}$ n'en est pas une.

Une matrice de rotation R vérifie les propriétés suivantes : $det(R) = 1$ et $R^{-1} = R^{t}$.

Pour transformer $R_{B\leftarrow A}$ en matrice de rotation, on lui applique une fonction **rotation**, qui transforme $R = (v_1\ v_2\ v_3)$, avec v_1, v_2, v_3 les colonnes de la matrice R, en $R = (v_1^*\ v_2^*\ v_3^*)$ avec : $v_1^* = \frac{v_1}{\|v_1\|}$, $v_3^* = \frac{v_1 * v_2}{\|v_1 * v_2\|}$, $v_2^* = v_3^* * v_1^*$

❑ Sur les *rvecs* et les *tvecs* :

Les *rvecs* sont les vecteurs de rotation, et les *tvecs* sont les vecteurs de translation (il y en a un couple par image). Ce sont des vecteurs tels que : la combinaison du k^{ème} vecteur de rotation avec le k^{ème} vecteur de translation, permet de passer des coordonnées des points de la mire de calibration dans le repère de la mire, aux coordonnées de ces même points dans le repère associé à la k^{ème} image.

La matrice $R_{B\leftarrow A}$ obtenue à partir de la moyenne faite sur les rvecs est déjà une matrice de rotation, nous n'avons donc pas besoin de lui appliquer la fonction **rotation**.

❑ Sur les quaternions :

Un quaternion est un élément $(w, x, y, z) \in \mathfrak{R}^4$ qui permet de représenter une rotation de la façon suivante:

$$w = \alpha.w \times \sin\left(\frac{\theta}{2}\right) \qquad y = \alpha.y \times \sin\left(\frac{\theta}{2}\right)$$

$$x = \alpha.x \times \sin\left(\frac{\theta}{2}\right) \qquad z = \cos\left(\frac{\theta}{2}\right)$$

Avec α l'axe autour duquel on effectue la rotation, et θ l'angle de rotation autour de cet axe.

Ainsi, les quaternions permettent de stocker un axe de rotation α et un angle de rotation θ d'une façon qui simplifiera la combinaison des rotations. Deux fonctions **quaternion2rotation** et **rotation2quaternion** permettent de passer d'une configuration sous forme de quaternion à son équivalent en matrice de rotation, et inversement.

On applique également la fonction **rotation** à la matrice $R_{B\leftarrow A}$ obtenue par moyenne sur les quaternions pour s'assurer que cette dernière est bien une matrice de rotation.

Ainsi, si l'on choisit d'exprimer toutes les matrices (R_{i}, T_{i}) par rapport au repère de la mire A par exemple, toute matrice (R_{i}, T_{i}) d'une image C_i détectant uniquement une mire $B \neq A$ pourra s'exprimer par rapport au repère de la mire comme suit :

$$\left(R_i^A, T_i^A\right) = \left(R_i^B, T_i^B\right) * (R_{B\leftarrow A}, T_{B\leftarrow A}) = (R_i^B R_{B\leftarrow A}, R_i^B T_{B\leftarrow A} + T_i^B).$$

Puis nous pouvons appliquer le programme **carve.py** pour opérer la reconstruction de l'objet en 3D.

Le fichier **carve** permettant d'opérer la reconstruction n'a pas fait l'objet de beaucoup de modification.

3.2.2 La démarche

- ❏ Détections des coins (**chessboard**):

  Au cours de la première tentative de détection des coins de chaque mire, nous avons réussi à détecter les points dans chaque image, mais l'ordre de la séquence des points n'a pas été correcte: les coins ne sont pas détectés dans le même ordre dans toutes les images (on le constate en affichant uniquement les deux premiers coins détectés dans chaque image). Dans les images qui détectent les « grandes » mires, les coins sont détectés du haut vers le bas, de la gauche

vers la droite. Dans les images qui détectent les « petites » mires (à l'exception de deux images), les coins sont détectés du bas vers le haut, de la droite vers la gauche (donc il faut inverser les tableaux):

Début en haut à gauche	Début en bas à droite
Mire 3	Mire 2
Mire 1	Mire 4
Mire 4 pour les images 13 et 14	

Comme on le voit dans la figure 2.5, pour les images détectant les mires 2 et 4 (à l'exception des images 13 et 14), l'ordre des coins détectés est inversé. Donc on a ici une erreur de l'ordre des coins qui sera importante pour déterminer le centre de l'objet et les coordonnées de la caméra.

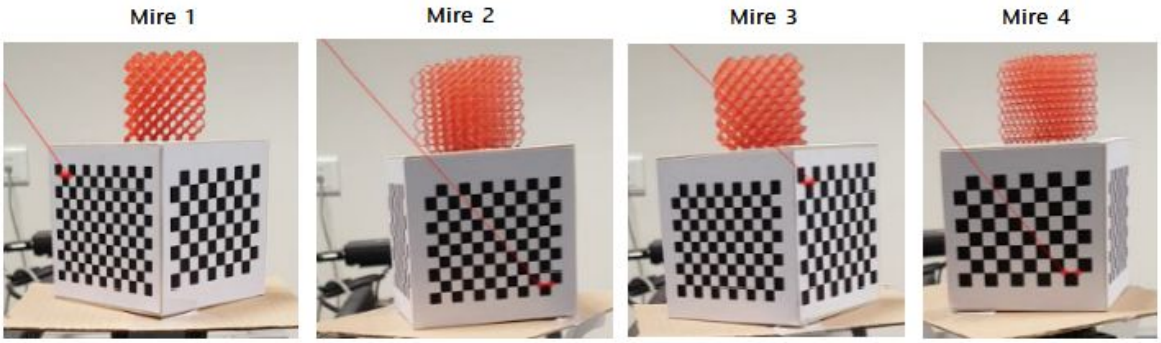

Figure 3.5: les deux premiers coins détectés par chaque mire

Pour corriger cette erreur, nous avons écrit un script permettant de lire l'ordre des coins de chaque image, et d'inverser l'ordre des tableaux des coins détectés pour toute image détectant la mire 2 ou 4 (à l'exception des images 13 et 14). Avec cette démarche, nous avons abouti aux résultats de la figure 3.6.

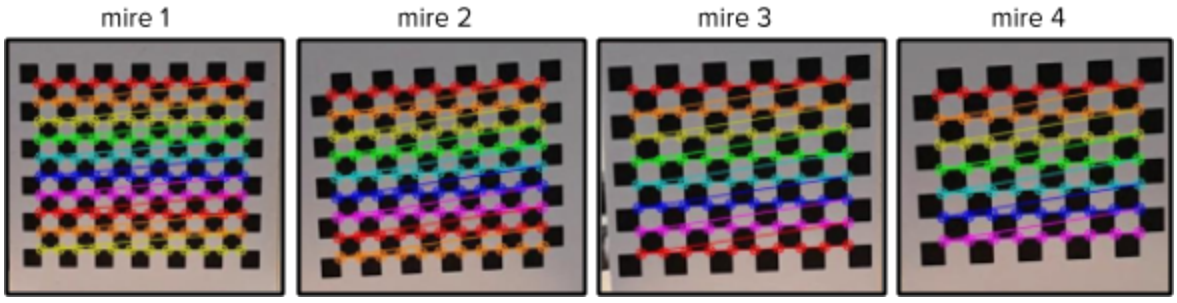

Figure 3.6: les coins détectés par chaque mire

❑ Calibration partie 1 (**calib**)

Quand nous avons essayé de générer les matrices des intrinsèques (matrice K), nous avons dans un premier temps utilisé la bibliothèque **opencv2**. Cependant, cette bibliothèque n'est pas capable d'utiliser les donnés des quatre mires ensemble pour générer directement la matrice K finale. Pour cette raison, nous avons standardisé tous les mires à une seule taille: nous avons utilisé la taille de la mire 4, qui est celle qui a le moins de coins détectés. Ainsi, nous avons réduit à 9x7=63 le nombre de coins détectés pris en compte pour chaque image.

Comme l'erreur de calibration par image n'était pas suffisamment faible, nous avons opté pour la bibliothèque **opencv3**, où nous n'avions pas besoin de standardiser les mires (et ainsi nous ne perdions pas de données). Nous avons ainsi obtenu une matrice K plus précise, avec une erreur par image plus petite.

Matrice K obtenue:

```
camera matrix:
[[  1.51959393e+03   0.00000000e+00   4.91958371e+02]
 [  0.00000000e+00   1.51977874e+03   1.06423939e+03]
 [  0.00000000e+00   0.00000000e+00   1.00000000e+00]]
```

Exemples d’erreur par image obtenues (toutes les erreurs sont inférieures à 0,25):

```
..\images2\selframe_0000002.npy:  0.133110881201
..\images2\selframe_0000003.npy:  0.185719814612
..\images2\selframe_0000012.npy:  0.143076939582
..\images2\selframe_0000013.npy:  0.175083187972
..\images2\selframe_0000022.npy:  0.143370581528
..\images2\selframe_0000023.npy:  0.171372418496
..\images2\selframe_0000032.npy:  0.147896506211
..\images2\selframe_0000042.npy:  0.16256904271
..\images2\selframe_0000052.npy:  0.170313119706
..\images2\selframe_0000062.npy:  0.17891388501
..\images2\selframe_0000072.npy:  0.190776413411
..\images2\selframe_0000082.npy:  0.180621786317
..\images2\selframe_0000092.npy:  0.153989579596
..\images2\selframe_0000102.npy:  0.159665247466
..\images2\selframe_0000112.npy:  0.167666244691
..\images2\selframe_0000122.npy:  0.158511382821
..\images2\selframe_0000132.npy:  0.156480761719
..\images2\selframe_0000134.npy:  0.111162215051
..\images2\selframe_0000142.npy:  0.152943447963
..\images2\selframe_0000144.npy:  0.117715902918
..\images2\selframe_0000152.npy:  0.155579659902
```

❑ Calibration partie 2 (**passage**)

- Résultats avec la moyenne sur les matrices de rotation:

En faisant des moyennes sur les matrices de rotation, nous avons obtenu des matrices de passage trop imprécises, qui ont conduit à une mauvaise reconstruction. De ce fait, pour augmenter la précision des matrices de passage obtenues, et donc augmenter la qualité de la reconstruction, nous avons opté pour une construction des matrices de passage via une moyenne sur les rvecs et les tvecs.

- Résultats avec la moyenne sur les rvecs et les tvecs:

En faisant une moyenne sur les rvecs et les tvecs, on obtient les résultats suivants:

➢ Pour la matrice de passage (R_{13}, T_{13}):

```
Ecart-type pour rvecs:  [ 0.00115956  0.00088506  0.00093346]
Ecart-type pour tvecs:  [ 0.05365337  0.01659547  0.10131359]
R_13 =
[[  8.16713374e-04   5.36170746e-03  -9.99985292e-01]
 [  1.30411061e-02   9.99900531e-01   5.37190399e-03]
 [  9.99914628e-01  -1.30453016e-02   7.46709541e-04]]
T_13 =
[ 52.58164533  -0.6814909    8.18626245]
```

Vérifions la forme de la matrice de rotation R_13 :

En prenant l'axe des z orienté vers l'intérieur du cube (pour avoir un repère des mires direct), on a les systèmes d'axes suivants:

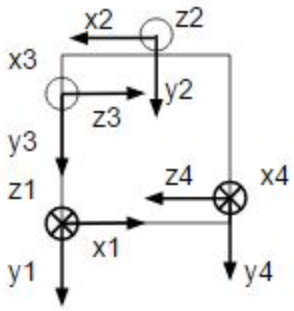

Figure 3.7: systèmes d'axes, avec la mire 1 devant, la mire 4 à droite, la mire 2 derrière et la mire 3 à gauche

D'où: $x_1 \rightarrow z_3$, $y_1 \rightarrow y_3$, et $z_1 \rightarrow -x_3$,

Donc R_{13} doit avoir la forme suivante: $\begin{pmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$.

Donc R_{13} a la bonne forme, mais la précision pourrait encore être améliorée.

Vérifions les propriétés que doit vérifier R_{13} pour être considérée comme une matrice de rotation:

```
Determinant de R_13:
1.0
Inverse de R_13:
[[  8.16713374e-04   1.30411061e-02   9.99914628e-01]
 [  5.36170746e-03   9.99900531e-01  -1.30453016e-02]
 [ -9.99985292e-01   5.37190399e-03   7.46709541e-04]]
Transposée de R_13:
[[  8.16713374e-04   1.30411061e-02   9.99914628e-01]
 [  5.36170746e-03   9.99900531e-01  -1.30453016e-02]
 [ -9.99985292e-01   5.37190399e-03   7.46709541e-04]]
Erreur entre inverse et transposée de R_13:
[[  0.00000000e+00   1.73472348e-18   2.22044605e-16]
 [  1.73472348e-18   3.33066907e-16  -3.46944695e-18]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00]]
```

Donc R_{13} est bien une matrice de rotation.

➢ Pour la matrice de passage (R_{14}, T_{14}) :

```
Ecart-type pour rvecs:  [ 0.0020618   0.00098745  0.00106479]
Ecart-type pour tvecs:  [ 0.05355152  0.02526669  0.13762597]
R_14 =
[[  8.79533253e-04  -3.85296537e-03   9.99992191e-01]
 [  1.27460906e-02   9.99911386e-01   3.84144333e-03]
 [ -9.99918378e-01   1.27426123e-02   9.28565560e-04]]
T_14 =
[ -9.36567402  -3.60525363  52.38902664]
```

Vérifions la forme de la matrice de rotation R_14 :

D'après la représentation des systèmes d'axes précédente, on a:

$x_1 \to -z_4$, $y_1 \to y_4$, et $z_1 \to x_4$,

Donc R_{14} doit avoir la forme suivante: $\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix}$.

Donc R_{14} a la bonne forme, mais la précision pourrait encore être améliorée.

Vérifions les propriétés que doit vérifier R_{14} pour être considérée comme une matrice de rotation:

```
Determinant de R_14:
1.0
Inverse de R_14:
[[  8.79533253e-04   1.27460906e-02  -9.99918378e-01]
 [ -3.85296537e-03   9.99911386e-01   1.27426123e-02]
 [  9.99992191e-01   3.84144333e-03   9.28565560e-04]]
Transposée de R_14:
[[  8.79533253e-04   1.27460906e-02  -9.99918378e-01]
 [ -3.85296537e-03   9.99911386e-01   1.27426123e-02]
 [  9.99992191e-01   3.84144333e-03   9.28565560e-04]]
Erreur entre inverse et transposée de R_14:
[[ -1.08420217e-19   0.00000000e+00   0.00000000e+00]
 [  1.30104261e-18   2.22044605e-16   1.73472348e-18]
 [  0.00000000e+00   1.30104261e-18   0.00000000e+00]]
```

Donc R_{14} est bien une matrice de rotation.

➢ Pour la matrice de passage (R_{12}, T_{12}) :

```
Ecart-type pour rvecs:  [ 0.0311766   2.19784676  0.00225801]
Ecart-type pour tvecs:  [ 0.05977128  0.0961015   0.11473521]
R_12 =
[[-0.62218995  0.02347515 -0.78251427]
 [ 0.02131416  0.99968774  0.01304304]
 [ 0.78257612 -0.00856339 -0.62249602]]
T_12 =
[ 41.08429828  -2.90457712  59.83576681]
```

Vérifions la forme de la matrice de rotation R_12 :

D'après la représentation des systèmes d'axes précédente, on a:

$x_1 \rightarrow -x_2$, $y_1 \rightarrow y_2$, et $z_1 \rightarrow -z_2$,

Donc R_{12} doit avoir la forme suivante: $\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$.

Donc R_{12} n'est pas une matrice de rotation, même si on a bien:

```
Determinant de R_12:
1.0
Inverse de R_12:
[[-0.62218995  0.02131416  0.78257612]
 [ 0.02347515  0.99968774 -0.00856339]
 [-0.78251427  0.01304304 -0.62249602]]
Transposée de R_12:
[[-0.62218995  0.02131416  0.78257612]
 [ 0.02347515  0.99968774 -0.00856339]
 [-0.78251427  0.01304304 -0.62249602]]
Erreur entre inverse et transposée de R_12:
[[  0.00000000e+00   6.93889390e-18   2.22044605e-16]
 [  0.00000000e+00   5.55111512e-16  -3.46944695e-18]
 [  0.00000000e+00   1.73472348e-18   0.00000000e+00]]
```

Pour trouver la source du problème, nous avons affiché les rvecs des R_{12_i} des images i détectant à la fois les mires 2 et 3 ayant servi à construire la matrice R_{12} :

```
In [8]: print R_rvecs
[[ -4.49734832e-02  -3.14054007e+00  -1.21196090e-03]
 [ -4.50825993e-02  -3.14091731e+00   8.26663576e-04]
 [  4.53639455e-02   3.14109041e+00  -2.60939322e-03]
 [ -4.21070867e-02  -3.13894668e+00  -6.16166504e-03]
 [ -4.25212408e-02  -3.13907777e+00  -5.46754446e-03]
 [ -4.32160184e-02  -3.13970946e+00  -4.27539295e-03]
 [ -4.41722778e-02  -3.13948612e+00  -2.77504369e-03]]
```

On constate que le deuxième terme des rvecs alterne entre $-\pi$ et π selon les images. Cela est dû au fait que les mires 1 et 2 sont à l'opposé l'une de l'autre, et que les rvecs peuvent être codés dans le sens direct comme dans le sens indirect. Par conséquent, par moment le programme passe par $-\pi$ et par moment par π, pour représenter deux points pourtant très proches dans l'espace.

Pour éviter ce problème, nous avons décidé de changer de méthode pour construire les matrices de passage: nous avons utilisé les quaternions (qui ont été introduits précédemment), susceptibles de donner des résultats plus précis.

- Résultats avec la moyenne sur les quaternions:

En faisant une moyenne sur les quaternions on obtient les résultats suivants:

➢ Pour la matrice de passage (R_{13}, T_{13}):

```
Ecart-type pour quaternion:  [ 0.00031764  0.00052127  0.00031169
0.0004199 ]
Ecart-type pour tvecs:  [ 0.05365337  0.01659547  0.10131359]
R_13 =
[[  8.17075022e-04   5.36169112e-03  -9.99985337e-01]
 [  1.30409570e-02   9.99900579e-01   5.37189236e-03]
 [  9.99914646e-01  -1.30451554e-02   7.47072219e-04]]
T_13 =
[ 52.58164533  -0.6814909    8.18626245]
```

R_{13} a bien la forme $\begin{pmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$, donc c'est bien une matrice de rotation. De plus, on obtient:

```
Determinant de R_13:
1.0
Inverse de R_13:
[[  8.17075022e-04   1.30409570e-02   9.99914587e-01]
 [  5.36169065e-03   9.99900460e-01  -1.30451536e-02]
 [ -9.99985218e-01   5.37189189e-03   7.47072103e-04]]
Transposée de R_13:
[[  8.17075022e-04   1.30409570e-02   9.99914646e-01]
 [  5.36169112e-03   9.99900579e-01  -1.30451554e-02]
 [ -9.99985337e-01   5.37189236e-03   7.47072219e-04]]
Erreur entre inverse et transposée de R_13:
[[  0.00000000e+00   0.00000000e+00  -5.96046448e-08]
 [ -4.65661287e-10  -1.19209290e-07   1.86264515e-09]
 [  1.19209290e-07  -4.65661287e-10  -1.16415322e-10]]
```

Ce qui valide la matrice R_{13}.

➢ Pour la matrice de passage (R_{14}, T_{14}):

```
Ecart-type pour quaternion:  [ 0.00034928  0.00092782  0.00034917
0.0004801 ]
Ecart-type pour tvecs:  [ 0.05355152  0.02526669  0.13762597]
R_14 =
[[  8.80231732e-04  -3.85322282e-03   9.99992192e-01]
 [  1.27460044e-02   9.99911427e-01   3.84169188e-03]
 [ -9.99918401e-01   1.27425231e-02   9.29266913e-04]]
T_14 =
[ -9.36567402  -3.60525363  52.38902664]
```

R_{14} a bien la forme $\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix}$, donc c'est bien une matrice de rotation. De plus, on obtient:

```
Determinant de R_14:
1.0
Inverse de R_14:
[[  8.80231673e-04   1.27460035e-02  -9.99918342e-01]
 [ -3.85322259e-03   9.99911368e-01   1.27425222e-02]
 [  9.99992192e-01   3.84169188e-03   9.29266913e-04]]
Transposée de R_14:
[[  8.80231732e-04   1.27460044e-02  -9.99918401e-01]
 [ -3.85322282e-03   9.99911427e-01   1.27425231e-02]
 [  9.99992192e-01   3.84169188e-03   9.29266913e-04]]
Erreur entre inverse et transposée de R_14:
[[ -5.82076609e-11  -9.31322575e-10   5.96046448e-08]
 [  2.32830644e-10  -5.96046448e-08  -9.31322575e-10]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00]]
```

Ce qui valide la forme de la matrice R_{14}.

- Pour la matrice de passage (R_{12}, T_{12}):

```
Ecart-type pour quaternion:  [  3.25285711e-04   9.90111207e-03
6.99783638e-01   7.20644935e-04]
Ecart-type pour tvecs:  [ 0.05977128  0.0961015   0.11473521]
R_12 =
[[-0.81931734  0.57105654 -0.05112321]
 [ 0.5717141   0.82045025  0.00211554]
 [ 0.04315213 -0.02749456 -0.99869013]]
T_12 =
[ 41.08429828  -2.90457712  59.83576681]
```

R_{12} n'a pas la forme $\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$, même si on obtient:

```
Determinant de R_12:
1.0
Inverse de R_12:
[[-0.81931734  0.5717141   0.04315213]
 [ 0.57105649  0.82045013 -0.02749456]
 [-0.0511232   0.00211554 -0.99869007]]
Transposée de R_12:
[[-0.81931734  0.5717141   0.04315213]
 [ 0.57105654  0.82045025 -0.02749456]
 [-0.05112321  0.00211554 -0.99869013]]
Erreur entre inverse et transposée de R_12:
[[  0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [ -5.96046448e-08  -1.19209290e-07   3.72529030e-09]
 [  3.72529030e-09  -4.65661287e-10   5.96046448e-08]]
```

,

Donc la matrice R_{12} obtenue n'est pas une matrice de rotation.

Pour chercher la cause du problème, on affiche les quaternions obtenus pour les couples d'images détectant les mires 2 et 3 servant à construire la matrice R_{12} :

```
..\Calib\selframe_0000003 ..\Calib\selframe_0000002
[  4.07846646e-04  -1.43256134e-02  -9.99897229e-01  -3.77258147e-04]
--------------------------------------------------
..\Calib\selframe_0000013 ..\Calib\selframe_0000012
[  2.53942096e-04  -1.43477284e-02  -9.99897002e-01   2.53942096e-04]
--------------------------------------------------
..\Calib\selframe_0000023 ..\Calib\selframe_0000022
[  5.16075543e-04   1.43641026e-02   9.99896364e-01  -8.17119609e-04]
--------------------------------------------------
..\Calib\selframe_0000683 ..\Calib\selframe_0000682
[ 0.00117547 -0.01341678 -0.99990737 -0.00196544]
--------------------------------------------------
..\Calib\selframe_0000693 ..\Calib\selframe_0000692
[ 0.00109956 -0.01355179 -0.99990604 -0.00174656]
--------------------------------------------------
..\Calib\selframe_0000703 ..\Calib\selframe_0000702
[  7.71770432e-04  -1.37742198e-02  -9.99903889e-01  -1.37412784e-03]
--------------------------------------------------
..\Calib\selframe_0000713 ..\Calib\selframe_0000712
[  8.96585343e-04  -1.40714089e-02  -9.99900200e-01  -8.84132768e-04]
--------------------------------------------------
```

On constate une fois encore que le troisième terme, qui correspond à : $y = \alpha . y \times \sin\left(\frac{\theta}{2}\right)$ alterne entre -1 et 1 suivant les images, ce qui correspond à $\theta = -\pi$ ou $\theta = \pi$. Nous retrouvons donc le même problème que dans le cas précédent, dû au fait que les mires 1 et 2 sont à l'opposé l'une de l'autre sur le

cube des mires de calibration, qui fait que deux points normalement extrêmement proches sont représentés extrêmement éloignés.

Deux cas de figures s'offrent alors à nous:

- ➢ nous pouvons revenir à une moyenne sur les rvecs, en opérant une transformation sur les rvecs:
  - nous faisons un premier calcul de rvecs (pour la 1ère image)
  - pour tous les calculs de rvecs suivants (images suivantes), si la différence entre le premier rvecs calculé et le $j^{ème}$ rvecs calculé est trop importante, on change le $j^{ème}$ rvecs en -rvecs. Dans le cas contraire, on le conserve tel quel.
- ➢ nous gardons les matrices de passage obtenues selon la moyenne sur les quaternions (qui offre plus de précision qu'une moyenne sur les moyennes de rotation ou sur les rvecs), et nous opérons la reconstruction 3D en ne conservant que les mires 1, 3 et 4 (soit donc en supprimant la mire 2).

  Nous avons choisi d'opter pour cette seconde solution.

❑ Reconstruction (**carve**)

Comme nous avons eu des difficultés pour construire l'objet en 3D avec les quatre mires ensemble (problèmes dans la partie de calibration), nous avons commencé par faire la reconstruction avec une seule mire pour regarder si les images prises dans l'étape expérimentale étaient suffisantes pour suivre notre objectif (la séquence d'images utilisées a été segmentée en ensembles d'images détectant la mire 1, détectant la mire 2, détectant la mire 3 et détectant la mire 4).

Nous avons utilisé nos nouveaux scripts pour faire la reconstruction de chaque mire, et nous sommes arrivés aux objets 3D dans toutes les cases. La figure 3.8 montre les maillages obtenus.

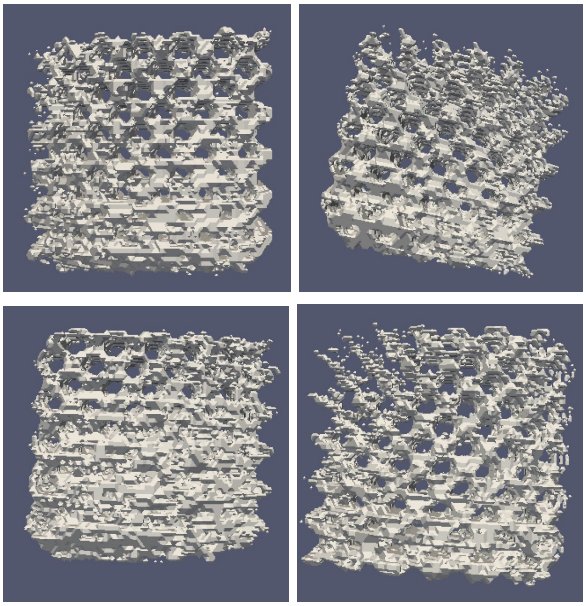

Figure 3.8: la reconstruction par chaque mire, respectivement 1,2, 3 et 4

Puis nous avons opéré la reconstruction de l'objet en 3D à partir de la séquence d'images détectant les mires 1, 3 et/ou 4 (soit donc en supprimant les images détectant uniquement les mires 2). Pour arriver à une bonne reconstruction du cube, nous avons joué sur les valeurs des paramètres suivants :

r: rapport minimal entre le rouge et les autres couleurs (vert et bleu);

low: intensité minimale pour le rouge;

taille: taille du cube (axb) où a est la taille en mm autour du centre de rotation et b est le nombre de pixels dans chaque direction.

Nous sommes partis des valeurs initiales:

r = 0,5

low = 16

taille = 50 x 200

Nous avons fixé la taille à 50, parce qu'il faut avoir au moins 5 cm pour être sûr de voir tout l'objet, et un nombre de 200 pixels. Initialement, nous avions pris une

taille de 100 mm autour du centre de rotation; cependant, pour augmenter la précision, ie pour assurer une meilleure résolution, nous pouvions:

- ➢ soit réduire la taille du cube autour du centre de rotation et garder le même nombre de pixels dans chaque direction
- ➢ soit garder la même taille de cube autour du centre de rotation et augmenter le nombre de pixels dans chaque direction. Néanmoins, cette deuxième solution aurait été beaucoup trop coûteuse en mémoire, et donc pour ne pas provoquer de bug, nous n'avons pas opté pour cette deuxième solution.

Avec ces paramètres, le résultat obtenu est affiché en figure 3.9.

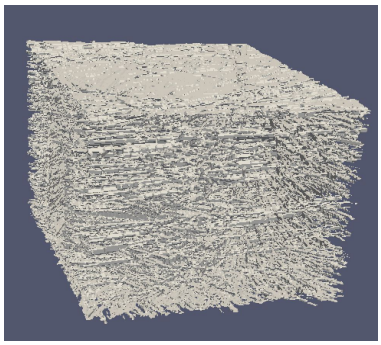

Figure 3.9: reconstruction avec r=0,5 ; low = 16; taille = 50 x 200

L'objet obtenu a bien la forme d'un cube, mais il n'a pas de détails, nous avons conclu que la limite entre le rouge et les autres couleurs a été choisie trop petite. Donc, nous avons augmenté juste le paramètre r de 0,5 à 0,7 pour dénoter son influence. La figure 3.10 montre la reconstruction obtenue après modification de ce paramètre.

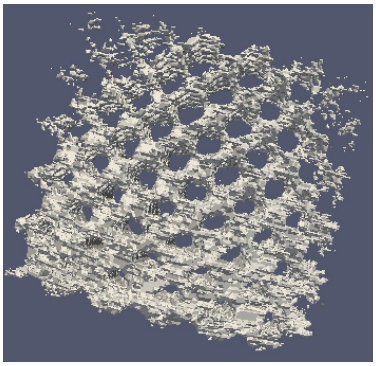

Figure 3.10: reconstruction avec r=0,7 ; low = 16; taille = 50 x 200

Comme le résultat obtenu a été amélioré de manière significative, nous avons augmenté **r** encore une fois (figure 3.11).

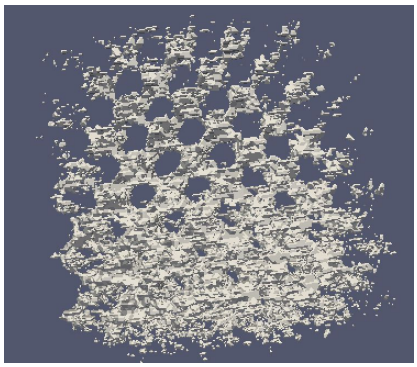

Figure 3.11: reconstruction avec r=0,8 ; low = 16; taille = 50 x 200

Maintenant le résultat obtenu a été moins bon que le résultat précédent, parce qu'avec une valeur trop grande de **r**, nous avons perdu quelques parties de l'objet, qui a été trop creusé.

Ainsi, nous avons fixé r = 0,7; la prochaine étape concerne donc la valeur de **low** à adapter. Les figures 3.12 et 3.13 montrent les résultats pour low=32 et low=64.

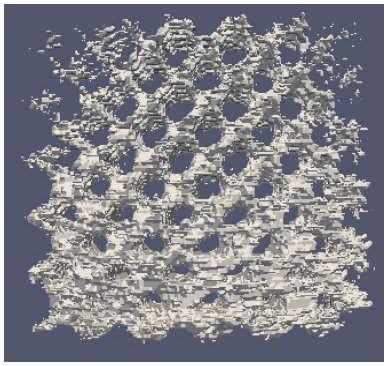

Figure 3.12: reconstruction avec r=0,7 ; low = 32; taille = 50 x 200

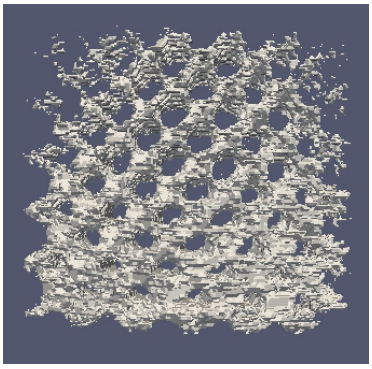

Figure 3.13: reconstruction avec r=0,7 ; low = 64; taille = 50 x 200

Les changements de **low** (passage de 16 à 32 à 64) montrent une petite amélioration dans la résolution.

À la fin des tests, nous avons conclu que le paramètre **r** a une influence importante dans la qualité de la reconstruction, tandis que le paramètre **low** a une influence plus faible.

Les paramètres qui nous ont permis d'obtenir la meilleure reconstruction sont:

r = 0,7

low = 64

taille = 50 x 200

4 Obtention des caractéristiques mécaniques de l'objet réel

Une fois développé un moyen de créer un maillage à partir d'un objet réel, on peut utiliser cette méthode pour déterminer précisément la valeur des paramètres mécaniques d'un objet quelconque. C'est à décrire le protocole mis en place que s'attache cette seconde partie.

4.1 Simple Finite Element in PYthon

Le programme utilisé pour les calculs d'éléments finis est Simple Finite Element in PYthon (SfePy), développé par Robert Cimrman. Il s'agit d'un programme de résolution de systèmes d'équations aux dérivées partielles en une, deux ou trois dimensions. Il se présente comme un package Python et possède de nombreuses équations préenregistrées sous forme de modules. Il est également doté d'un module de visualisation des résultats et permet d'avoir accès à des animations montrant l'évolution de la déformation d'un objet soumis à une contrainte mécanique depuis l'état initial jusqu'à l'état final de la transformation.

L'avantage de ce programme est double : outre le fait de ne nécessiter que la connaissance de Python pour être intégré directement dans un code plus large, il peut directement être appelé en ligne de commande, ce qui permet d'y faire appel au sein d'autres scripts, convenant parfaitement au rôle intégré que nous comptons lui donner pour ce projet.

- General use:
  - Plain text:

    R. Cimrman. SfePy - write your own FE application. In P. de Buyl and N. Varoquaux, editors, Proceedings of the 6th European Con- ference on Python in Science (EuroSciPy 2013), pages 65–70, 2014. http://arxiv.org/abs/1404.6391.

  - BibTeX:

```
@InProceedings{cimrman14:_sfepy_write_your_own_fe_applic,
  author =       {Robert Cimrman},
  title =        {{SfePy} - Write Your Own {FE} Application},
  booktitle =    {Proceedings of the 6th European Conference on
                  Python in Science (EuroSciPy 2013)},
  pages =        {65--70},
  year =         2014,
  editor =       {Pierre de Buyl and Nelle Varoquaux},
  note =         {http://arxiv.org/abs/1404.6391},
}
```

Crédits pour la réalisation du programme SfePy que l'on peut trouver sur le site Internet.

Installation

L'installation de SfePy nécessite la présence des programmes ou packages suivants sur la machine :

- Un compilateur C (mingw32 par exemple)
- Python 2.7 ou 3.x (IMPORTANT : le module de visualisation ne fonctionne que sous Python 2.7)
- NumPy
- Cython

Les packages suivants sont nécessaires :

- Pyparsing
- SciPy
- Scikit-umfpack
- Matplotlib
- PyTables
- SympPy
- Mayavi
- Pysparse

- Igakit
- Pymetis
- wxPython

IMPORTANT : s'assurer que les programmes nécessaires à l'installation de ces packages sont aussi installés (igakit nécessite un compilateur FORTRAN, scikit-umfpack nécessite UMFPACK, petsc4py nécessite PETSc, etc...)

❏ Protocole d'installation

L'installation de SfePy peut se faire depuis Anaconda ou l'invite de commande de toute machine possédant une connexion Internet. Les instructions relatives à l'installation de SfePy sont disponibles sur le site de SfePy en cliquant sur le lien suivant : http://sfepy.org/doc-devel/index.html

L'installation comporte plusieurs étapes et à chaque étape, plusieurs instructions sont données par l'éditeur du programme pour accomplir la même action, laissant le choix à l'utilisateur de la manière dont il souhaite procéder pour accomplir les différentes étapes de la procédure. Or, l'expérience a montré que les différents type de commande n'avaient pas en pratique un effet équivalent et que le choix de certaines d'entre elles pouvait aboutir à l'échec de l'installation. C'est la raison pour laquelle nous reproduisons ici la liste des instructions qui permettent de mener à bien l'installation.

```
conda install -c conda-forge sfepy

python setup.py build_ext --inplace

python setup.py install

python ./run_tests.py
```

❏ Documentation

L'ensemble de la documentation est disponible sur le site de SfePy. Néanmoins, il est

possible de générer la documentation sur une machine. Pour ce faire, Sphinx, numpydoc, LaTex et doxygen doivent être installés. Le manuel d'utilisation peut alors être généré grâce à la commande :

```
python setup.py doxygendocs
```

Pour utiliser l'outils de création de mesh, le programme utilise pexpect et gmsh ou tetgen.

4.2 Fonctionnement de SfePy

Comme de nombreux programmes similaires, SfePy fonctionne par traitement d'un fichier de description du problème appelé fichier d'entrée. Ce fichier référence un certains nombres de mot-clés qui permettent de définir l'ensemble des paramètres et équations nécessaires à la résolution d'un système d'équations aux dérivées partielles : équations, variables, domaine de résolution, sous-domaines, etc…

Ce fichier de description est lu par un script Python situé dans le dossier principal de SfePy, généralement simple.py qui génère une instance du problème décrit dans le fichier d'entrée. Puis, la fonction problem.time_update() est appelée pour traiter les conditions aux limites, les paramètres et les variables non-stationnaires. A cette étape, la résolution de l'instance problem se fait par l'appel de la commande problem.solve() avant d'être enregistrée par problem.save_state().

4En pratique : réaliser une simulation

Voici les étapes à suivre pour effectuer une simulation en utilisant SfePy en ligne de commande

1°) Effectuer un changement de dossier pour se placer dans le dossier principal de SfePy à l'aide de l'instruction `cd`

2°) Effectuer la lecture du fichier d'entrée à l'aide de simple.py. Par exemple pour le fichier linear-elastic.py qui est l'exemple d'un cube subissant sur l'une des faces une

contrainte mécanique à laquelle il produit une réponse linéaire, le code sera :

```
python simple.py\examples\linear_elasticity\linear_elastic.py
```

3°) Un fichier cylinder.vtk contenant les résultats est automatiquement généré dans le dossier principal de SfePy. La visualisation des résultats se fait en utilisant la procédure postproc.py :

```
python postproc.py cylinder.vtk
```

IMPORTANT : l'utilisation de la procédure postproc.py ne peut se faire que si le package Mayavi est installé sur la machine. Or, Mayavi ne peut être installé que sous Python 2.7

4.3 Installation de Mayavi

Mayavi n'est pas disponible à l'installation sous Windows en utilisant la commande:

conda install mayavi

il faut par conséquent télécharger le package directement pour pouvoir l'installer. On peut notamment utiliser le lien suivant :

https://pypi.python.org/pypi/mayavi

IMPORTANT : lors de l'installation de Mayavi, il est possible que l'installation du package entre en conflit avec des packages déjà installés. La désinstallation de ces packages à l'aide de la commande conda uninstall résoud le problème mais l'opération peut nécessiter la réinstallation complète de SfePy.

4.4 Principe du calcul sur les éléments finis

Le calcul en éléments finis a pour objectif l'études des déformations d'un milieu continu. L'idée est de discrétiser le domaine et moyennant des conditions limites, des paramètres mécaniques pour le milieu et une équation d'évolution, de propager la contrainte de proche en proche sur chacun des éléments jusqu'à obtenir un résultat convergeant. Ici, notre domaine est discrétisé en 3D à l'aide de trièdre qui constituent donc nos éléments finis. Cette grille, ou mesh, joue un rôle essentiel dans le processus.

lNotre objectif est d'étudier es grandes déformations d'une géométrie quelconque. Or, nous disposons déjà de deux fichiers inclus dans l'installation de SfePy qui vont nous permettre de mener ce projet à bien et par étape : le script balloon.py et le script linear_elastic.py. Le premier permet d'obtenir les grandes déformations d'une sphère, le second les petites déformations d'un cube.

Nous allons donc essayer d'obtenir les grandes déformations d'un cube, puis les grandes déformations en géométrie quelconque.

Pour effectuer cette transformation, il faut opérer quelques changements dans le script balloon.py. Si le champ étudié demeure un champ de déplacements, il faut cependant adapter les conditions limites et l'équation régissant l'évolution du champ des déplacement.

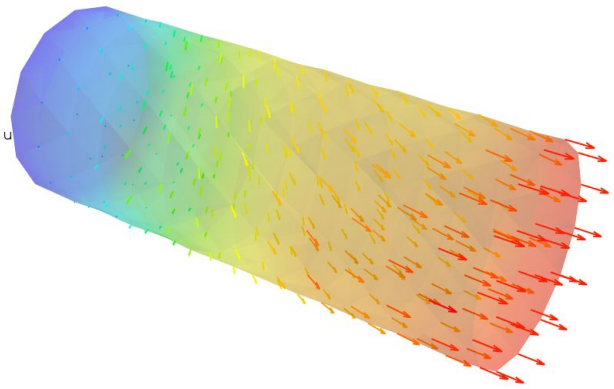

Figure 4.1: visualisation du champ de déplacement résultat du script linear_elastic.py

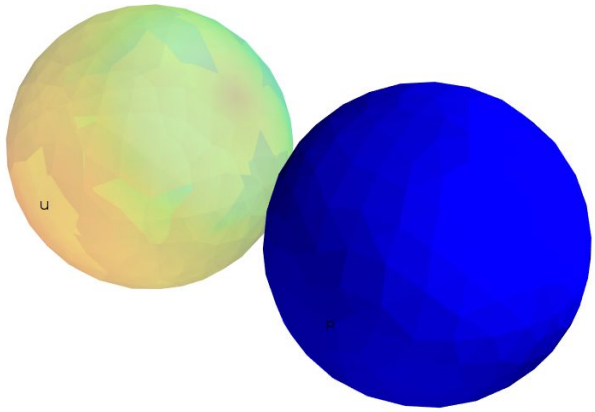

Figure 4.2: visualisation du résultat du script balloon.py

4.5 Listes des modifications effectuées

Meshage :

```
filename_mesh = data_dir + '/meshes/3d/unit_ball.mesh'
```

devient :

```
filename_mesh = data_dir + '/meshes/3d/cube_medium_hexa.mesh'
```

Délimitation du domaine :

```
regions = {
        'Omega'  : 'all',
        'Ax1' : ('vertices by get_ax1', 'vertex'),
        'Ax2' : ('vertices by get_ax2', 'vertex'),
        'Equator' : ('vertices by get_equator', 'vertex'),
        'Surface' : ('vertices of surface', 'facet'),
    }
```

Devient

```
regions = {
```

```
    'Omega' : 'all',
    'Left' : ('vertices in (x < -0.499999)', 'facet'),
    'Right' : ('vertices in (x > 0.4999999)', 'facet'),
     'Couronne' : ('vertices in (x>0.48) & (x<5) & (z>-0.49) &
(z<0.49) & (y<0.49) & (y>-0.49)', 'vertex'),
}
```

On crée deux faces sur lesquelles on appliquera les conditions limites, la charge pour l'une, l'encastrement pour l'autre (on rappelle qu'on cherche à obtenir les déformations d'un cube posé sur un support fixe sur lequel on exerce une pression uniformément répartie sur la face parallèle à la face en contact avec le support).

On remarquera la présence de la « couronne » il s'agit en fait d'un artefact créé pour modéliser une certaine rigidité au niveau des bords du cube qui se répercuterait sur la structure de la face externe dans sa globalité. Sans cette astuce, on remarque en effet que le comportement du cube n'est pas réaliste et évoque plus celui d'un liquide très visqueux qui se collerait aux parois du récipient qui le contient.

- ❏ Paramètres mécaniques

```
materials = {
    'solid' : ({'D': stiffness_from_lame(dim=3, lam=1e1, mu=1e0)},),
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}

integrals = {
    'i' : 1,
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

ebcs = {
    'TopFixed' : ('TopFixed', {'u.all' : 0.0}),
    'BotFixed' : ('BotFixed', {'u.all' : 0.0}),
    'Displaced' : ('Tail', {'u.2' : 10, 'u.[0,1]' : 0.0}),
}
```

Exemple de fichier d'entrée sous SfePy

Sous SfePy, l'équation de Mooney-Rivlin prend un seul paramètre en entrée, kappa. On modifie donc :

```
materials = {
        'solid' : ({
            'mu' : 50, # shear modulus of neoHookean term
            'kappa' : 0.0, # shear modulus of Mooney-Rivlin term
        },),
        'walls' : ({
            'mu' : 3e5, # shear modulus of neoHookean term
            'kappa' : 3e4, # shear modulus of Mooney-Rivlin term
            'h0' : 1e-2, # initial thickness of wall membrane
        },),
    }
```

Devient

```
materials = {
    'solid' : ({
        'kappa' : 10, # shear modulus of Mooney-Rivlin term
    },),
}
```

Conditions limites

```
   ebcs = {
        'fix1' : ('Ax1', {'u.all' : 0.0}),
        'fix2' : ('Ax2', {'u.[0, 1]' : 0.0}),
        'fix3' : ('Equator', {'u.1' : 0.0}),
    }
```

Devient

```
ebcs = {
    'Fixed' : ('Left', {'u.all' : 0.0}),
    'Displaced' : ('Right', {'u.0' : -0.05}),
    'Fixed2' : ('Couronne',{'u.2' :0.0 , 'u.1' : 0.0}),
}
```

Equation

```
equations = {
        'balance'
            : """dw_tl_he_neohook.2.Omega(solid.mu, v, u)
                 + dw_tl_he_mooney_rivlin.2.Omega(solid.kappa, v,
u)
                + dw_tl_membrane.2.Surface(walls.mu, walls.kappa,
walls.h0, v, u)
                + dw_tl_bulk_pressure.2.Omega(v, u, p)
                = 0""",
        'volume'
            : """dw_tl_volume.2.Omega(q, u)
              = dw_volume_dot.2.Omega(q, omega)""",
    }
```

Devient

```
equations = {

    'balance' :

    """dw_tl_he_mooney_rivlin.2.Omega(solid.kappa, v, u) = 0""",

}
```

Les autres instructions demeurent inchangées.

On obtient le résultat suivant visualisé au format .vtk

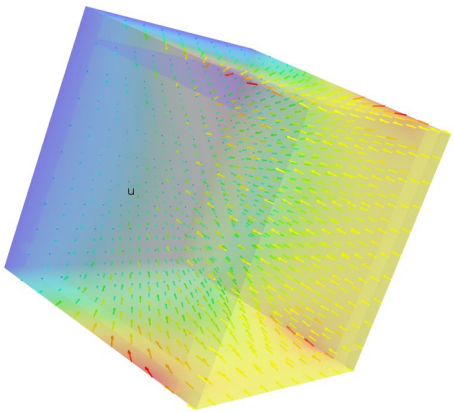

Figure 4.3: visualisation du résultat

Les mêmes résultats obtenus à l'aide du format .h5

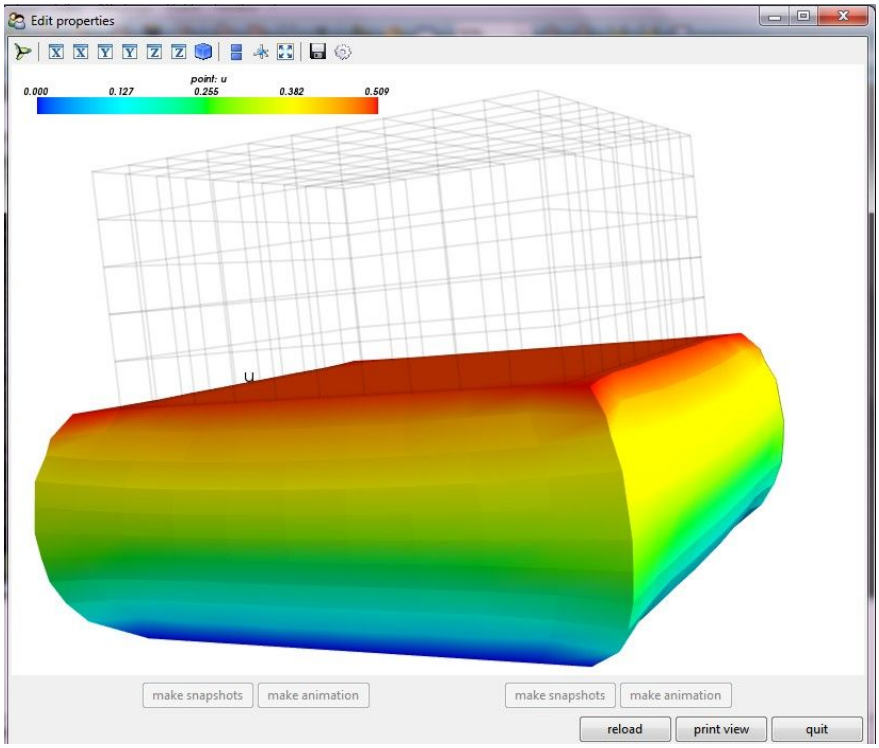

Figure 4.4: visualisation du résultat

❑ Passage à une géométrie quelconque

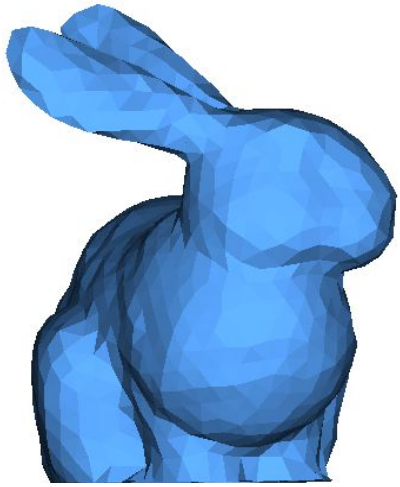

Figure 4.5: Exemple de géométrie quelconque visualisation d'un format .stl

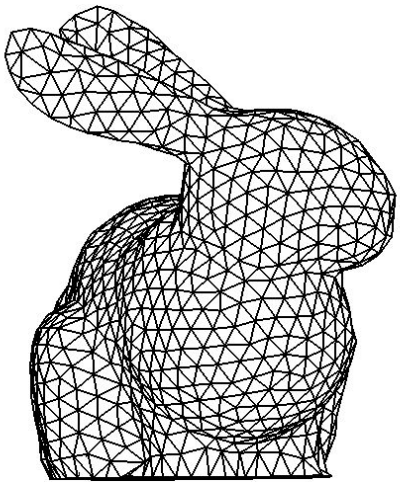

Figure 4.5: la même géométrie après meshage en utilisant Salomé

Le passage à une géométrie quelconque conserve globalement l'esprit du fichier tel qu'il est décrit précédemment. Les adaptations à réaliser sont les suivantes :

Trouver un moyen d'appliquer l'effet de la liaison encastrement à la partie inférieure de l'objet

Pour ce faire, l'idée retenue est de trouver l'ensemble des points dont la position suivant l'axe vertical, c'est-à-dire la différence entre la valeur de la composante suivant l'axe vertical et la valeur de la composante suivant l'axe vertical du point le plus bas de l'objet, c'est-à-dire ayant la plus faible valeur de composante verticale est inférieure à 5% de la hauteur totale de l'objet.

Trouver un moyen de sélectionner le point en lequel s'effectuera l'application de la contrainte mécanique. Etant donné que le dispositif d'application des forces tel qu'il a été imaginé et conçu pendant le projet permet une forte précision, on peut en effet imaginer sélectionner un point particulier du maillage pour y appliquer la contrainte mécanique. Il faudrait alors afficher le numéro de nœud attribué à chacun des points du maillage à l'aide Meshlab et sélectionner le point le plus adéquat.

Une autre méthode consiste évidemment à opérer de la même façon que pour l'application de la liaison encastrement et à créer une zone d'application de la force

similaire à la couronne décrite dans l'exemple de la grande déformation du cube. Cela part du présupposé de la convexité de l'objet, ce qui n'est pas toujours le cas (par exemple le lapin dont la partie la plus élevée n'est pas le sommet de la tête mais le sommet des oreilles), ce qui nécéssitera d'être corrigé dans un second temps.

4.6 Réalisation de l'algorithme

- ❏ Principe de l'algorithme

Notre objectif est de pouvoir comparer les propriétés mécaniques de l'objet réel avec les propriétés mécaniques souhaitées par le constructeur en utilisant la différence de réponse à des sollicitations mécaniques connues qui caractérise deux objets de paramètres mécaniques différents (on rappelle que SfePy n'utilise qu'un seul paramètre mécanique pour la résolution de l'équation de Mooney-Rivlin, appelé kappa)Pour ce faire, nous attribuons au hasard des propriétés mécaniques à la mesh modélisant l'objet à étudier, puis nous lui faisons subir une déformation sous une contrainte connue. Nous obtenons ainsi la même mesh mais déformée.

En parallèle nous faisons subir la *même* contrainte à l'objet réel (sens, direction, module et point d'application de la force identiques). Nous obtenons ainsi l'objet déformé. Sous hypothèse d'avoir un moyen d'obtenir à partir de l'objet réel déformé une mesh de cet objet, c'est-à-dire de pouvoir modéliser l'objet déformé sous la forme d'un ensemble cohérents de trièdres, on peut comparer les deux géométries obtenues, celle de l'objet réel déformé et celle du modèle déformé. Ainsi, on peut comparer la proximité de deux paramètres mécaniques à la valeur réelle des paramètres mécaniques de l'objet mécanique réel : dire que la valeur réelle du paramètre kappa de l'objet réel est plus proche de kappaA que de kappaB revient à dire que la distance entre la géométrie du modèle déformé et la géométrie de l'objet déformé est plus faible pour le modèle auquel on a attribué kappaA comme valeur de paramètre mécanique que celui auquel on a attribué kappaB.

Ainsi, en essayant plusieurs valeurs de paramètres et en conservant uniquement ceux

diminuant la distance relative entre la géométrie du modèle déformé et la géométrie de l'objet réel déformé, on obtient le meilleur paramètre de l'ensemble des paramètres testés, à savoir celui dont la valeur est la plus proche de la valeur réelle du paramètre réelle. Si nous parcourons suffisamment bien l'ensemble des solutions, nous pouvons obtenir une bonne approximation de la valeur réelle du paramètre kappa de l'objet réel. Nous choisissons de parcourir l'ensemble des solutions au hasard, en générant une valeur aléatoire du paramètre kappa entre deux bornes considérées fournies par l'utilisateur.

- ❏ La distance de Hausdorff

On le voit, l'algorithme précédant nécessite de choisir un moyen d'évaluation de la distance entre deux géométries. On propose de choisir pour ce faire de calculer la distance de Hausdorff entre les maillages représentant les deux géométries. Le calcul de cette distance consiste pour tout point d'un maillage A à calculer la distance euclidienne entre ce point A et tous les points d'un maillage B et à conserver la plus grande d'entre elles. Pour chaque point de A, on obtient une distance, on obtient donc un ensemble de distances après avoir réalisé cette démarche pour tous les points de A : le résultat final consiste à prendre la plus petite distance de cet ensemble.

- ❏ Réalisation pratique de l'algorithme

La réalisation pratique de l'algorithme nécessite la mise en place d'un certain nombre de fonctions qui répondent à plusieurs moments : génération d'une valeur de paramètre mécanique au hasard, génération d'un fichier d'entrée lisible par SfePy incluant ce paramètre, lecture de ce fichier, extraction des points du maillage déformé dans le fichier résultat généré par SfePy, calcul de la distance de Hausdorff entre le nuage de points obtenus et la géométrie de référence. Les résultats sont présentés dans le cas des grandes déformation d'un cube.

- ❏ Génération du fichier d'entrée

Le fichier d'entrée est préenregistré sous la forme d'un script Python. Nous ouvrons alors

ce script Python afin d'y définir kappa qui se trouve être défini sur la deuxième ligne du script.

```
def generate_file(kappa):
    print(kappa)

chdir('C:/Users/Yves-Gabriel/Documents/sfepy-master/sfepy-master
/projet_sfePy')
    f = open('mooney_rivlin_project_test.py','r')
    t = f.readlines()
    f.close()
    t[1] = 'kappa = '+str(kappa)+'\n'
    f = open('mooney_rivlin_project_test.py','w')
    for i in range(len(t)):
        f.write(str(t[i]))
    f.close()
```

- ❏ Extraction des points du maillage déformé

Le chemin d'enregistrement du fichier de sortie généré par la lecture par SfePy du fichier d'entrée est commandé par ce fichier d'entrée. Pour ne pas avoir à modifier le répertoire de travail, nous utilisons ici le dossier principal de SfePy. L'extension du fichier de sortie est également commandée par le fichier d'entrée. Nous choisissons ici le .vtk, facilement visualisable avec mayavi, qui est en outre facilement manipulable. Cette fonction renvoie la liste des points du maillage. Chaque point est représenté comme une liste de trois coordonnées.

```
def extract_point(path):
    f = open(path,'r')
    points = []
    s = f.readline()
    while s!= '\n':
        s = f.readline()
    s = f.readline()
    while s != '\n':
        s = f.readline()
        c = s.split()
        if c==[]:
            break
        c = [float(c[0]),float(c[1]),float(c[2])]
        points.append(c)
```

```
    print(points)
    return(points)
```

- ❏ Calcul de la distance de Hausdorff

Nous définissons une première fonction qui permet de calculer la distance euclidienne entre deux points en dimensions 3. Puis la deuxième fonction effectue le calcul de la distance de Hausdorff proprement dite entre deux géométries. Nous vérifions l'exactitude des résultats renvoyés par cette fonction à l'aide de deux tests : tout d'abord nous vérifions que la distance calculée entre deux géométries identiques est bien nulle. Ensuite, nous calculons à la main la distance entre deux géométries simples et nous vérifions que le résultat renvoyé par l'algorithme est correct.

```
def dist(point1, point2):

return(np.sqrt((point1[0]-point2[0])**2+(point1[1]-point2[1])**2
+(point1[2]-point2[2])**2))

def hausdorff(surface1,surface2):
     d = [] #liste des distances d'un point de s1 à tous les
autres points de s2
    D=[] #liste des plus petites distance d'un point de s1 à un
point de s2
    for point1 in surface1:
        for point2 in surface2 :
            d.append(dist(point1,point2))
        D.append(min(d))
        d=[]
    return(max(D))
```

- ❏ Calcul de l'objectif

C'est à cette étape qu'est calculée la déformation par éléments finis grâce à SfePy. On utilise ici la fonction os.system qui permet d'exécuter des instructions en ligne de commande au sein d'un script Python. On appréciera l'extrême simplicité de l'instruction faisant appel au calcul d'éléments finis.

```
def objectif(kappa,geomref,empl_file_sol) :
```

```
    generate_file(kappa) #génère le fichier sfepy
                                                                chdir(
'C:/Users/Yves-Gabriel/Documents/sfepy-master/sfepy-master')
                                    system('python        simple.py
examples/large_deformation/mooney_rivlin_project_test.py')
#génère le fichier contenant les points du modèle déformé

return(hausdorff(extract_point(geomref),extract_point(empl_file_
sol))) # calcule la distance d'Hausdorff entre le fichier
précédant et la géométrie de référence ; empl_file_sol est le
chemin jusqu'à l'emplacement du fichier solution
```

- ❑ Fonction de mutation

En dernier moment se trouve l'algorithme génétique à proprement parler. A chaque tour de boucle on produit un réel kappa, on calcule la valeur de la fonction objectif pour ce réel. On ne le conserve en mémoire que si on obtient un meilleur score que celui réalisé avec le paramètre stocké au début de la boucle.

```
def                mutation(kappa0,nmut,                kappamin,
kappamax,geomref,empl_file_sol):
      list_kappa = [kappa0]
      n_iter = [0]
      n=0
      kappa = kappa0
      kappa_opt = kappa
      obj = objectif(kappa0,geomref,empl_file_sol)
      opt = obj
      list_obj = [obj]
      while n<nmut:
      kappa = uniform(kappamin, kappamax)
      print(kappa)
      obj = objectif(kappa,geomref,apply_dis(empl_file_sol))
      if obj < opt:
            opt = obj
            kappa_opt = kappa
      n=n+1
      list_kappa.append(kappa_opt)
      n_iter.append(n)
      list_obj.append(opt)
      return([kappa_opt,list_kappa,n_iter,list_obj])
```

Lorsque l'on utilise des formats de fichier .vtk comme fichier de sortie, il ne faut pas oublier de calculer la position des points de la mesh après déformation. Pour cela on pourra utiliser la fonction suivante.

```
def apply_dis(path):
	f = open(path,'r')
	points = []
	disp = []
	points_disp = []
	s = f.readlines()
	i = 0
	while s[i][0] != 'P':
	i = i+1
	i=i+1
	while s[i] != '\n':
	c = s[i].split()
	if c == []:
		break
	c = [float(c[0]),float(c[1]),float(c[2])]
	points.append(c)
	i=i+1
	while s[i][0] != 'V'	:
	i=i+1
	i=i+1
	while s[i] != '\n':
	c = s[i].split()
	if c == []:
		break
	c = [float(c[0]),float(c[1]),float(c[2])]
	disp.append(c)
	i=i+1
	for j in range(len(points)-1):
	points_disp.append([points[j][0]+disp[j][0],
points[j][1]+disp[j][1], points[j][2]+disp[j][2] ])
	return(points_disp)
```

- ❏ Les extensions de fichiers

Les fichiers .stl, les fichiers .mesh et les fichiers .vtk

La plupart des imprimantes 3D utilisent en entrée des fichiers de type stl

(STereoLithography). Or, SfePy utilise en entrée des fichiers .mesh.

Il faut donc utiliser un outils pour convertir les fichiers .stl en fichier .mesh. Le programme Salome satisafait cette exigence. Nous allons évoquer ici le protocole à suivre pour obtenir un fichier .mesh utilisable par SfePy à partir d'un fichier .stl.

- ❏ Structure d'un fichier .mesh

Après un en-tête d'introduction, le nombre de sommet est donné, puis la position dans l'espace de chacun des points. Après les sommets, les tétraèdres constituant le maillage sont à leur tour énumérés un par un : à chaque sommet est attribué un numéro à chaque tétraèdre est attribuée la liste des numéros des quatre sommets qui le constituent.

```
MeshVersionFormatted 1
Dimension 3
Vertices
354
1.0000000000e-01 2.0000000000e-02 -1.2246063538e-18 0
1.0000000000e-01 1.8019377358e-02 8.6776747824e-03 0
1.0000000000e-01 1.2469796037e-02 1.5636629649e-02 0
1.0000000000e-01 4.4504186791e-03 1.9498558244e-02 0
1.0000000000e-01 -4.4504186791e-03 1.9498558244e-02 0
1.0000000000e-01 -1.2469796037e-02 1.5636629649e-02 0
1.0000000000e-01 -1.8019377358e-02 8.6776747824e-03 0
1.0000000000e-01 -2.0000000000e-02 3.6738190615e-18 0
1.0000000000e-01 -1.8019377358e-02 -8.6776747824e-03 0
1.0000000000e-01 -1.2469796037e-02 -1.5636629649e-02 0
1.0000000000e-01 -4.4504186791e-03 -1.9498558244e-02 0
1.0000000000e-01 4.4504186791e-03 -1.9498558244e-02 0
1.0000000000e-01 1.2469796037e-02 -1.5636629649e-02 0
1.0000000000e-01 1.8019377358e-02 -8.6776747824e-03 0
1.0000000000e-01 -1.2596809903e-02 -3.3134585250e-03 0

 4.8258669674e-02 -4.5830514282e-03 -3.9704032242e-03 0
 4.6159550548e-02 4.6954065328e-04 4.4654738158e-03 0
 3.3582992852e-02 5.0520147197e-03 3.0156029388e-03 0
 8.4256991744e-02 2.1215945017e-03 4.4717523269e-03 0
 1.7229551449e-02 6.7490423098e-03 -9.2394463718e-04 0
 7.2420634329e-02 -2.8333296068e-03 3.9018876851e-03 0
 4.1812837124e-02 -6.1458065175e-03 -4.2112097144e-03 0
 8.0029852688e-02 5.2159861661e-03 -9.7777221526e-05 0
 7.0254400373e-02 3.6161029129e-04 -1.1690315296e-04 0
 3.1963359565e-02 -1.0033597238e-02 9.6046030521e-03 0
 Tetrahedra
 1348
 29 61 46 30 6
 29 61 58 46 6
 29 58 28 46 6
 14 71 13 22 6
 14 24 1 70 6
 12 11 21 74 6
 12 74 21 73 6
 12 22 13 73 6
 12 21 22 73 6
 39 40 59 45 6
 39 49 45 59 6
```

❑ Structure d'un fichier .stl

```
endfacet
facet normal -9.349626e-001  3.428762e-001 -9.099968e-002
  outer loop
    vertex  3.724300e+002 -1.839770e+001  3.318500e+001
    vertex  3.740150e+002 -1.455770e+001  3.136860e+001
    vertex  3.730420e+002 -1.792120e+001  2.869250e+001
  endloop
endfacet
facet normal -9.649767e-001  1.927085e-001 -1.779986e-001
  outer loop
    vertex  3.730260e+002 -1.537450e+001  3.584600e+001
    vertex  3.741060e+002 -1.167550e+001  3.399580e+001
    vertex  3.740150e+002 -1.455770e+001  3.136860e+001
  endloop
endfacet
facet normal -9.428745e-001  1.197976e-001 -3.108638e-001
  outer loop
    vertex  3.730260e+002 -1.537450e+001  3.584600e+001
    vertex  3.730070e+002 -1.076690e+001  3.767930e+001
    vertex  3.741060e+002 -1.167550e+001  3.399580e+001
  endloop
endfacet
facet normal -9.441087e-001  1.184940e-001 -3.076002e-001
  outer loop
    vertex  3.716810e+002 -1.552630e+001  3.991570e+001
    vertex  3.730070e+002 -1.076690e+001  3.767930e+001
    vertex  3.730260e+002 -1.537450e+001  3.584600e+001
  endloop
endfacet
```

Un fichier .vtk, qui est le format des fichier de sortie de SfePy se divise en plusieurs partie qu'il est important de connaître pour pouvoir les manipuler

```
# vtk DataFile Version 2.0
step 0 time 0.000000e+00 normalized time 0.000000e+00, generated by simple.py
ASCII
DATASET UNSTRUCTURED_GRID

POINTS 354 float
1.000000e-01 2.000000e-02 -1.224606e-18
1.000000e-01 1.801938e-02 8.677675e-03
1.000000e-01 1.246980e-02 1.563663e-02
1.000000e-01 4.450419e-03 1.949856e-02
1.000000e-01 -4.450419e-03 1.949856e-02
1.000000e-01 -1.246980e-02 1.563663e-02
1.000000e-01 -1.801938e-02 8.677675e-03
1.000000e-01 -2.000000e-02 3.673819e-18
1.000000e-01 -1.801938e-02 -8.677675e-03
1.000000e-01 -1.246980e-02 -1.563663e-02
1.000000e-01 -4.450419e-03 -1.949856e-02
1.000000e-01 4.450419e-03 -1.949856e-02
1.000000e-01 1.246980e-02 -1.563663e-02
1.000000e-01 1.801938e-02 -8.677675e-03
1.000000e-01 -1.259681e-02 -3.313459e-03
1.000000e-01 6.331288e-03 -3.624797e-04
1.000000e-01 -8.812382e-03 -1.038914e-02
1.000000e-01 -1.056074e-02 4.882110e-03
1.000000e-01 -5.114428e-03 1.134626e-02
1.000000e-01 2.740151e-03 1.283293e-02
1.000000e-01 -6.388331e-04 -1.116539e-02
```

Après un en-tête, la liste des coordonnées des points est énumérée

```
CELLS 1348 6740
4 28 60 45 29
4 28 60 57 45
4 28 57 27 45
4 13 70 12 21
4 13 23 0 69
4 11 10 20 73
4 11 73 20 72
4 11 21 12 72
4 11 20 21 72
4 38 39 58 44
4 38 48 44 58
4 38 59 48 58
4 38 59 37 48
4 4 76 19 18
4 4 76 3 19
4 4 18 81 76
4 4 5 81 18
```

On trouve ensuite la liste des tétraèdres composant le solide : à chaque sommet est attribué un numéro, sur une même ligne sont rassemblés les sommets reliés entre eux pour former un tétraèdre.

```
VECTORS u float
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
-4.000000e+16 4.000000e+31 4.000000e+31
```

Enfin, et c'est l'originalité du format, dans le cas d'un objet déformé, on trouve le champ de vecteur de déplacement par rapport à la position standard du point. On obtient la mesh de l'objet déformé en sommant pour tout point chacune des composantes.

4.7 Modélisation 3D du dispositif de déformation

A terme, il faudra comparer une déformation effectuée sur la pièce imprimée avec une simulation numérique de cette même déformation. Pour effectuer et mesurer la déformation sur la pièce imprimée, il faut notamment créer un dispositif capable d'appliquer une déformation précise sur la pièce imprimée. La première version de ce dispositif a été réalisée sous AutoCAD :

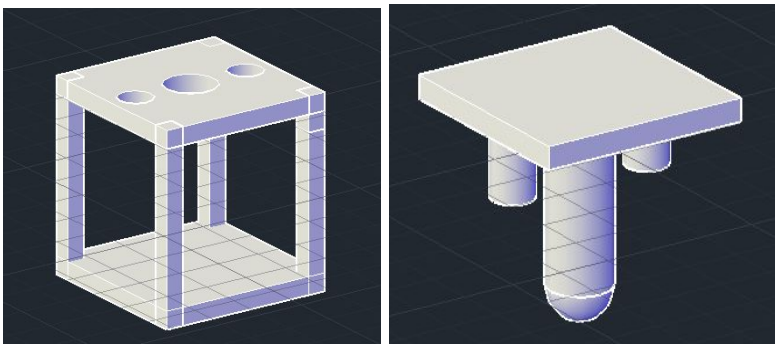

Figure 4.6: Premier modèle réalisé sous AutoCAD

Un problème a cependant été rencontré avec ce premier modèle: lorsque la pièce destinée à appliquer la déformation sur l'objet imprimé va venir appuyer sur l'objet en question, les guides de part et d'autre du pic d'application vont également être visibles dans la cage, et donc sur les images prises par la caméra. Les guides vont donc occulter des parties de l'objet sur certaines images pendant la rotation.

Nous avons donc conçu un deuxième modèle, dans lequel les guides ne seront pas visibles dans la cage lors de l'application de la force sur l'objet. Ce deuxième modèle a été réalisé sous CATIA. C'est un modèle valide, qui permettra d'appliquer une déformation précise sur la pièce imprimée sans que le mode d'application ne gène la capture des images par la caméra.

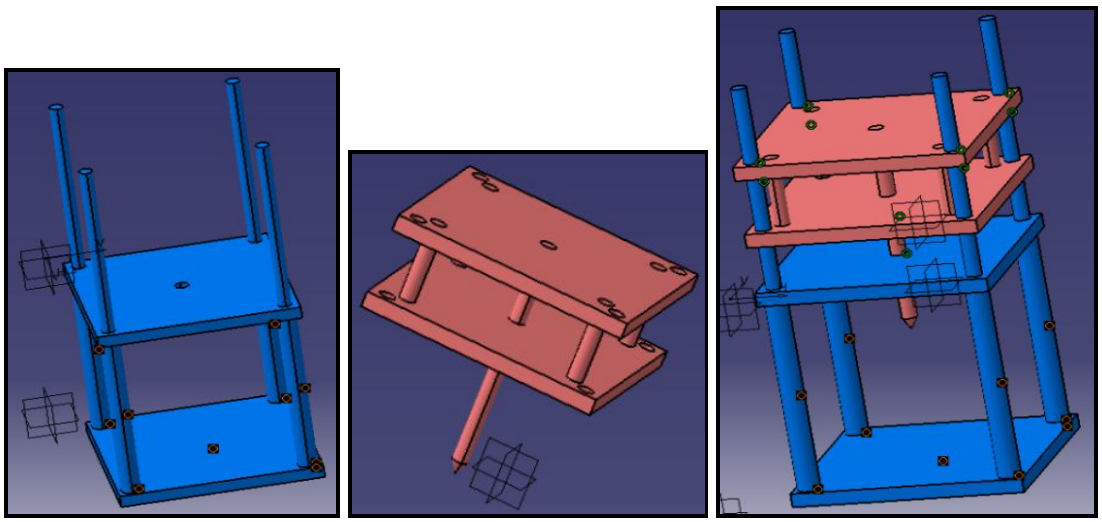

Figure 4.7: Deuxième modèle réalisé sous CATIA

5 La division du travail

Nous avons divisé les tâches en deux grandes parties: reconstruction géométrique et détermination des caractéristiques mécaniques d'un objet réel

La première partie a été réalisée par Diane et Jéssica. Jéssica s'est occupée plus spécifiquement de la calibration d'une mire et des modifications des scripts pour la partie des quatre mires, tandis que Diane a travaillé sur la modification du script **chessboard** pour la partie des quatre mires et sur le développement du script **passage**, qui exprime la matrice de passage du repère d'une mire à autre. La reconstruction et la partie d'analyse des résultats ont été faites par Diane et Jéssica ensemble.

Diane a également travaillé sur la modélisation 3D du dispositif pour la simulation de d'une déformation, et Jéssica dans la partie de création d'un maillage sur le logiciel *Tetgen,* pour passer du résultat de la reconstruction à la simulation mécanique.

Le travail d'Yves-Gabriel, pour cause de compatibilité logicielle, a porté sur la deuxième partie : installation et apprivoisement de SfePy, réalisation des scripts d'extraction de points, de calcul de distance de Hausdorff, adaptation des scripts préenregistrés dans SfePy, réalisation du script de l'algorithme génétique, installation des sous-modules de visualisation, application du champ de déplacement aux points de la mesh. Cette division du travail a été une division de fait après que le programme SfePy ait été uniquement installé sur la machine d'Yves-Gabriel, Diane étant absente le jour de l'installation pour raison médicale, Jéssica étant en attente d'un nouveau Mac.

La réflexion quant à l'élaboration du dispositif capable d'appliquer une déformation précise sur la pièce imprimée a réuni tous les membres du groupe. Le travail était

exigeant et volumineux et de surcroît sans lien direct avec les cours de département, ce qui a nécessité beaucoup d'adaptation de la part du groupe.

6 Conclusion

L'objet de notre étude portait sur le développement d'un nouveau procédé destiné à mesurer les différences de propriétés géométriques et mécaniques entre un objet réel imprimé en trois dimensions d'une part et d'autre part le modèle fourni par l'utilisateur à l'entrée de l'imprimante 3D. En effet, si les imprimantes 3D sont des dispositifs d'une relative précision, des biais existent toujours dans la réalisation pratique des objets.

Le premier mouvement de l'étude nous a permis d'obtenir un moyen de reconstituer un maillage numérique d'un objet déformé. Ce résultat, bien qu'il puisse encore être amélioré, est capital, puisqu'il nous permet d'obtenir un modèle numérique de l'objet réel déformé.

Cela ouvre sur le second moment de l'étude : la détermination des caractéristiques mécaniques proprement dites par comparaison du modèle numérique de l'objet réel déformé et d'une version déformé du modèle numérique de l'objet. La comparaison entre les deux modèles permet, moyennant le calcul d'une fonction de coût basée mettant en jeu la distance de Hausdorff, d'obtenir une bonne approximation du paramètre mécanique de l'objet.

En conclusion de ce travail nous aimerions soulever un autre point, celui de l'application pratique de la contrainte sur l'objet réel et la nécessité de développer un dispositif performant permettant d'appliquer cette contrainte aussi efficacement qu'un logiciel de calcul de FEM : ponctuellement.

Finalement, le projet nous a permis de développer nos compétences techniques en nous plongeant dans un domaine que nous ne connaissions pas du tout et nous a fait gagner en autonomie en nous faisant rechercher la solution des problèmes qui se posaient par nous-mêmes en utilisant la documentation ou les forums des différents

outils que nous utilisions. Cela a également été l'occasion d'une plongée dans l'univers feutré des laboratoires que nous ne fréquentons pas ordinairement au cours de notre formation.