Rapport technique de stage

Sébastien Canu

25 septembre 2012

Table des matières

Chapitre 1

Présentation du problème

Les crânes et les fossiles retrouvés lors des fouilles archéologiques peuvent être déformés. Cela dépend de plusieurs facteurs, la pression des sédiments, les éboulements dans les cavernes... Afin de pouvoir correctement les étudier il est nécessaire de les rétro-déformer. Pour cela, il existe plusieurs technique. Le but du stage est d'en étudier une.

La méthode de rétro-déformation employée est la suivante. On suppose que le crâne a été déformé selon une seule direction sous la pression des sédiments. Pour retrouver la forme originale du crâne, on réalise l'expérience numérique suivante. Le crâne est placé dans une matière molle et on y applique une contrainte. Il y a deux inconnues a déterminer dans ce modèle : la direction et l'intensité de la contrainte. Pour les déterminer, on a défini des critères évaluant la qualité de la rétro-déformation, puis on a cherché la direction et l'intensité qui donnaient les meilleurs résultats. L'expérience a été réalisé avec SOFA (un simulateur mécanique pour le domaine médical) sur deux crânes Sts52 et AragoXXI.

Chapitre 2

Travail réalisé

Ce qui suis décrit, de manière syntaxique, le travail que j'ai réalisé durant mon stage. Pour plus de précision technique je vous invite a consulter les morceaux de code commenté disponibles en annexe.

2.1 La simulation avec sofa

SOFA est un simulateur mécanique appliqué au domaine médical. Les scènes simulées dans SOFA sont représentées sous forme de graphe. La scène utilisée est disponible en annexe (voir A). Elle a été écrite par Benjamin Gilles ; je n'en maîtrise pas tous les aspects techniques, mais je comprends son fonctionnement général. C'est une simulation calculant des déformations plastiques par la méthode des éléments finis. Elle est composée du nœud root et de 2 nœuds principaux (voir le schémas 2.1). Le premier composant du nœud root charge un mesh 3d, et créé une image 3d. Les voxels à l'intérieur du mesh ont une valeur de un et ceux a l'extérieur ont une valeur nulle. Après il y a le nœud behavior, c'est le cœur de la simulation. Il créé la grille des éléments finis et la simule. On définit le coefficient d'élasticité et le coefficient de poisson pour le crane et pour la matière molle. Le nœud visuale permet de représenter l'objet. Le nœud CollisionBox permet d'appliquer un champ de force pour représenter la contrainte. La figure 2.2 présente la simulation.

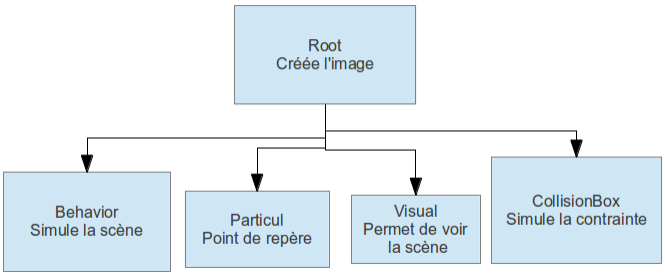


FIGURE 2.1 – Le schéma de la scène.

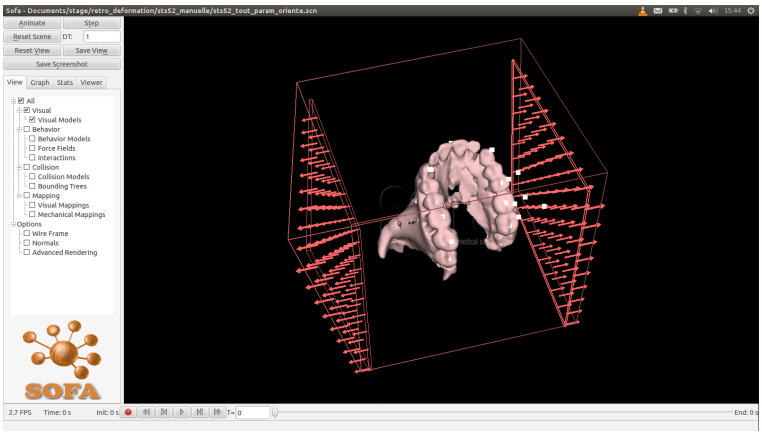


FIGURE 2.2 – La simulation dans SOFA.

2.2 Critères évaluant la qualité de la rétro-déformation

Grâce à la scène décrite ci dessus on peut rétro-déformer les crâne. Il s'agit maintenant de définir et calculer des critères évaluant la qualité de la rétro-déformation. On a définit deux critères l'un est basé sur la correspondance entre certain points du repère et l'autre basé sur la symétrie.

2.2.1 Critère de correspondance

Le critère de correspondance peut être utilisé lorsqu'une partie de l'objet étudié n'est pas déformée. On peut donc placer des points de repère sur la partie a rétro-déformer et les faire correspondre avec leurs homologues sur la partie non déformée. Par exemple, Sts52 est composé de deux parties : le maxillaire et la mandibule. Seul le maxillaire semble être déformé ; donc pour évaluer la qualité

de la rétro-déformation il suffit de calculer un critère basé sur la correspondance des dents du maxillaire avec celles de la mandibule. Sur chacune des deux parties on a placé des points de repère aux emplacements qui doivent correspondre. Le critère est calculé par la formule suivante :

Soit $A_{1..n}$ les points de repère placés sur la partie à rétro-déformer et $B_{1..n}$ les points places sur la partie déformé.

$$critere = \sqrt{\sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{(A_i A_j - B_i B_j)^2}{n * (n+1)/2}} \tag{2.1}$$

2.2.2 Critère de symétrie

J'ai utilisé un critère de symétrie simple et facile à mettre en œuvre. On place un ensemble de points qui caractérise le plan de symétrie et deux autres ensembles de point qui doivent être respectivement symétrique. A partir du premier ensemble de point on évalue l'équation du plan de symétrie. Pour cela j'ai testé deux méthodes, celle des moindres carrés et celle d'analyse par composante principal. La seconde donnait des résultats plus précis c'est donc celle que j'ai utilisé. L'équation du plan de symétrie est recalculée à chaque itération de la simulation dans SOFA. Le critère est ensuite calculé en symétrisant les points puis en prenant la racine carré de la somme des carrés des distances entre les symétriques des points repérés et leurs homologues. Tout ceci est résumé par la formule suivante :

Soit P le plan de symétrie. Soit $A_{1..n}$ les points de repères et $B_{1..n}$ les points de repère devant être le symétrique des précédents. Soit $A'_{1..n}$ le symétrique des points $A_{1..n}$ par rapport au plan P.

$$critere = \sqrt{\sum_{i=1}^{n} \frac{(A_i A'_i)^2}{n}} \tag{2.2}$$

2.3 Module SofaPython

Le module SofaPyhon permet d'exécuter du code python dans la scène. On peut dynamiquement créer ou supprimer des nœuds et modifier ou enregistrer des donnés de la simulation. Cependant il n'est pas encore très développé. Son fonctionnement est simple, on place dans la scène un composant qui fait référence à un scripte python. Ensuite le composant appellera les fonctions du scripte à certain moment (lorsque le graphe est créé, lorsque l'utilisateur appuie sur un bouton, à chaque itération de la simulation ...). On peut placer plusieurs composants python faisant appelle au même scripte.

Je me suis servi du module pour placer les points de repère et pour calculer les critères. La solution que j'ai mise en place n'est pas la plus élégante mais elle a le mérite de marcher. La scène et le scripte python sontdisponibles en annexe (A et B). Dans la scène j'ajoute un nœud pour chaque point de repéré et dedans

je place le composant faisant appelle à python et si nécessaire un composant pour mapper le point au modèle. J'ai d'abord essayé de créer un seul script qui créé tout les points en même temps mais j'ai eu des problèmes lors de la création du graphe avec les composants de mappage. Dans le script python, lors de la création du graphe, je créé les points je les stocke dans une variable globale. Il est peut être possible de les récupérer en explorant le graphe depuis un module python (ce qui éviterai de les stocker) mais je n'ai pas trouvé comment faire. Enfin, a chaque itération de la simulation je calcule les critères. Pour cela j'utilise des opérations de l'algèbre linéaire. J'ai d'abord essayé d'utiliser une bibliothèque de calcul scientifique pour python : numpy mais j'avais des erreur lorsqu'elle se chargait. J'ai donc ressortit mes cours d'algèbre linéaire et réécrit les opérations dont j'avais besoin comme la résolution de Gaus, la multiplication matricielle, la décomposition en valeur singulière... Une fois les critères calculés je les enregistre dans un fichier.

2.4 Script Python

La méthode décrite ci dessus permet d'évaluer la qualité de la rétro-déformation pour diffèrents critères. Il s'agit maintenant de trouver les paramètres qui minimisent ces critères. La rétro-déformation dépend de la direction, de l'intensité la contrainte et des caractéristiques des matériaux (module et coefficient de poisson du crâne et de la matière molle). La direction de la contrainte peut être modifiée en utilisant la fonction rotation de SOFA. Celle ci permet d'effectuer une rotation en angle d'Euler au crâne. L'intensité de la contrainte peut être modifiée en jouant sur la taille des champs de force ainsi que sur le nombres et le module des forces. Les caractéristiques des matériaux sont définies lors de la construction de la scène.

Il est très facile de modifier les paramètres décrits ci dessus en éditant manuellement le fichier décrivant la scène mais pour écrire des programmes qui recherchent automatiquement de la meilleure rétro-déformation il faut que la simulation SOFA se comporte comme une fonction qui a pour argument les paramètres à optimiser et qui retourne une évaluation de la qualité de la rétro-déformation. Pour cela j'ai écrit une fonction en python qui édite le fichier décrivant la scène SOFA. Ensuite elle lance SOFA pendant un certain temps. Une fois le temps écoulé la fonction tue le processus SOFA. Puis elle lit les fichiers dans lequel le critère a été enregistré. Si celui ci s'est stabilisé elle le renvoie comme résultat d'évaluation. Il existe sûrement une méthode plus élégante et plus efficace qui évite de lire et d'écrire dans des fichiers. Peut être qu'une amélioration du module sofaPython permettra de le faire. Je n'ai cependant pas eu le temps de chercher une autre méthode. J'ai gardé celle ci qui était facile à mettre en œuvre et qui marchait.

J'ai écrit plusieurs programmes pour optimiser la fonction décrite si dessus. La première fonction était une simple recherche par la force brute. Mais elle était très longue. En effet l'évaluation de la fonction prenait, en fonction de la précision voulue, entre 6 et 60 secondes. Pour le second programme, j'ai

tenté d'implanter la méthode du gradient conjugué en l'évaluant par différence finie. Mais ça ne donnait pas de bons résultats, l'algorithme ne convergeait pas rapidement. Le troisième programme est une recherche par raffinements successifs. On balaye d'abord rapidement l'espace de recherche puis on ré-explore là ou les résultats ont été les meilleurs. Il faut faire attention avec cette méthode. Elle peut converger vers un optimum local. Ce programme est disponible en annexe (C). Je me suis aussi intéressé a des algorithmes d'optimisation plus complexes, adaptés pour les fonctions longues à évaluer comme l'algorithme IAGO mais je n'ai pas eu le temps de l'implémenter.

Chapitre 3

Simulationss et résultats

3.1 Simulation

Les premières simulations que j'ai lancé cherchent uniquement la meilleure direction de rétro-déformation. Au début j'utilisais les angles euler soit un espace de recherche à trois dimensions. Or la direction de la contrainte peut être représentée par un vecteur unitaire. En coordonné sphérique, il ne ne suffit que de deux angles pour caractériser la direction d'un vecteur. Donc j'ai ajouté une fonction qui calcule les angle d'Euler de rotation du crane a partir des angle sphériques qui caractérisent la direction de la contrainte. Ansi l'espace de recherche a été réduit d'une dimension.

Il faut faire attention avec le critère de symétrie, en effet il y a plusieurs configuration pour lesquelles le critère de symétrie est optimal (voir figure 3.1). Cela peut poser des problèmes avec la recherche par affinements successifs. Lorsque que le programme balaye grossièrement l'espace de recherche il peut "rater" le bon optimum et converger vers le mauvais. (voir la figure 3.2)

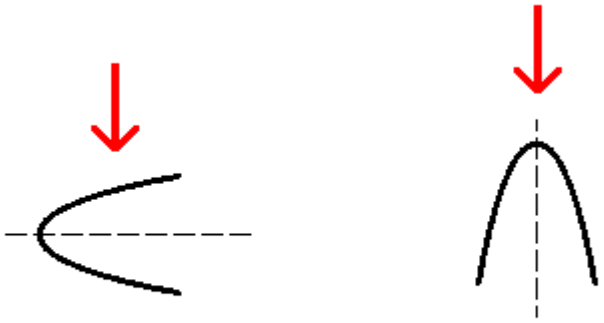


FIGURE 3.1 – Voici deux positions pour lequel le critère de symétrie est optimal.

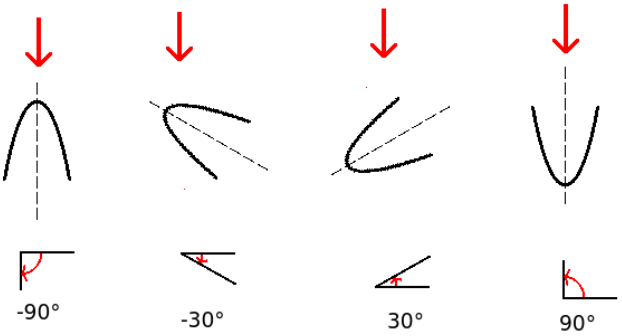


FIGURE 3.2 – Le premier balayage part de -90 degrés jusqu'à 90 degrés tous les 60 degrés. Les meilleurs résultat sont -90 et 90 degrés .On a "raté" l'optimum à 0 degrés. On a le même problème avec Sts52.

Il faut également faire attention a la précision des calculs. On peut la régler dans la simulation avec plusieurs paramètres. Il y a la taille des voxels lors de la construction de l'image 3d et la finesse de la grille des éléments finis. Il faut aussi adapter d'autre paramètres comme la finesse du champs de force et peut être le temps entre chaque itération. Au début je travaillais une précision faible pour que les calculs soit plus rapides mais lorsque que je faisais tourner le crâne autour de l'axe de direction de la contrainte, je constatais une grande différence sur le critère (entre 30% et 50% d'écart relatif). J'ai donc amélioré la précision afin d'avoir un bon compromis avec le temps de calcul. Pour une minute de

calcul, j'avais un écart relatif relatif de 10

L'expérience a été réalisée sur deux crânes Sts52 et AragoXXI. J'ai d'abord cherché la direction de la contrainte. Puis j'ai élargi le champs de recherche en ajoutant l'intensité de la contrainte et d'autres paramètres (module d'élasticité et coefficient de poissons).

3.2 Résultats pour sts52

Sts52 est composé de deux parties le maxillaire et la mandibule. Seul le maxillaire semble être déformé ; donc pour évaluer la qualité de la rétro-déformation on a utilisé un critère basé sur la correspondance des dents du maxillaire avec celles de la mandibule. Sur chacune des deux parties on a placé des points de repère aux emplacements qui doivent correspondre.

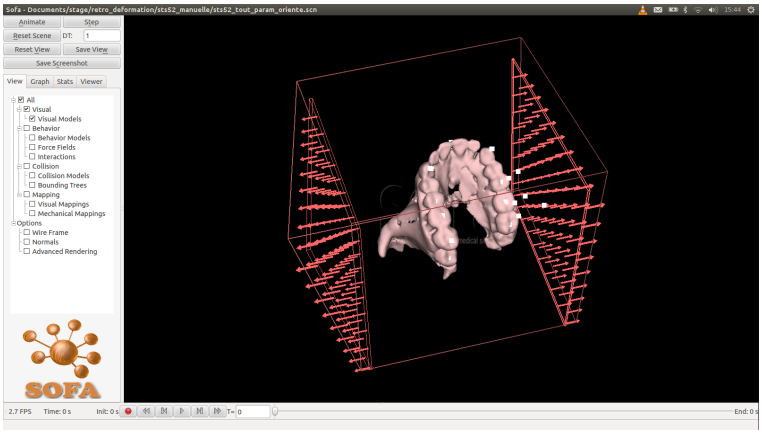


FIGURE 3.3 – la simulation avec Sts52 dans SOFA.

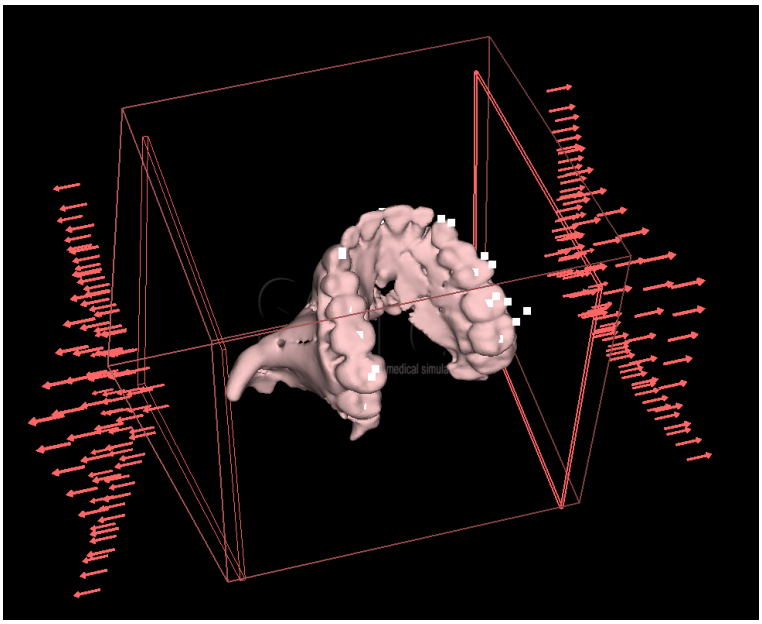


FIGURE 3.4 – Sts52 rétro-déformé.

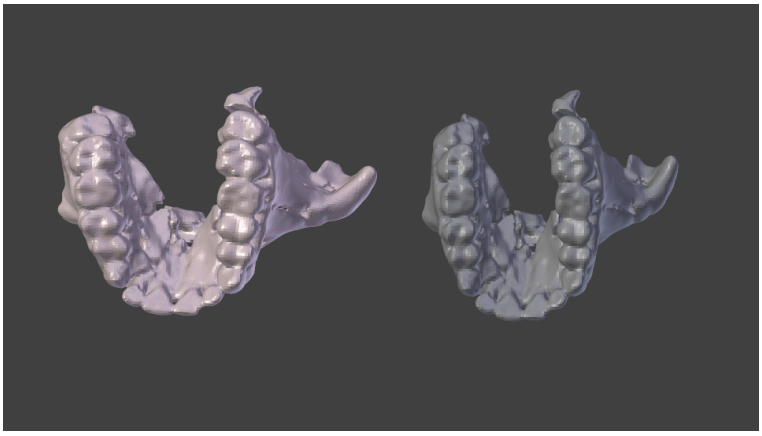


FIGURE 3.5 – Comparaison avec le modèle non rétro-déformé (en gris).

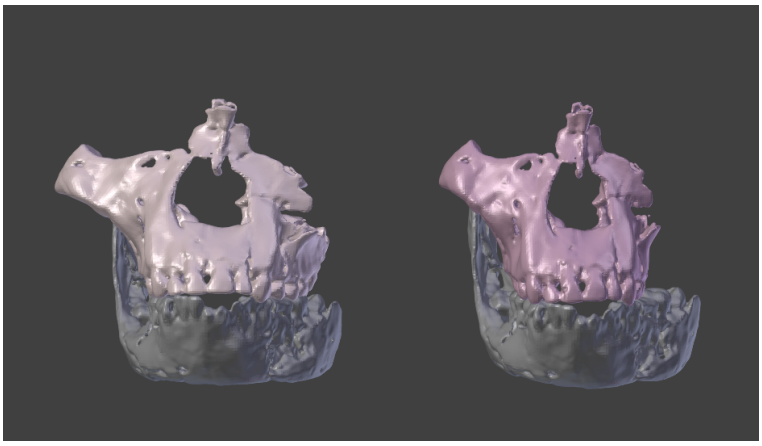


FIGURE 3.6 – A droite le maxillaire rétro-déformé, à gauche le maxillaire original.

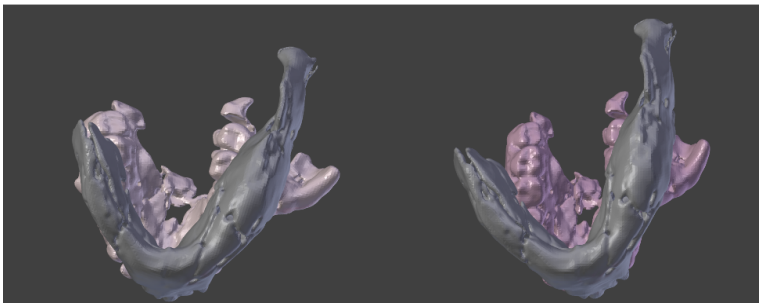


FIGURE 3.7 – A droite le maxillaire rétro-déformé, à gauche le maxillaire original.

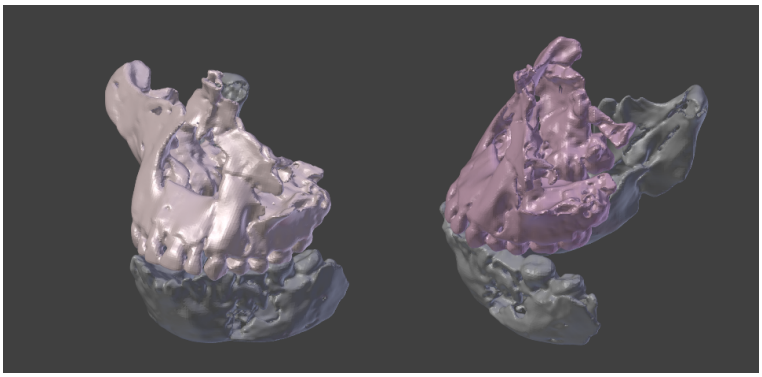


FIGURE 3.8 – A droite le maxillaire rétro-déformé, à gauche le maxillaire original.

3.3 Résultats pour AragoXXI

Pour AragoXXI, on a choisi un critère de symétrie. Les résultats présentés ici pour AragoXXI sont les meilleurs que nous avons. Mais ils peuvent être amélioré en choisissant mieux les points de repère.

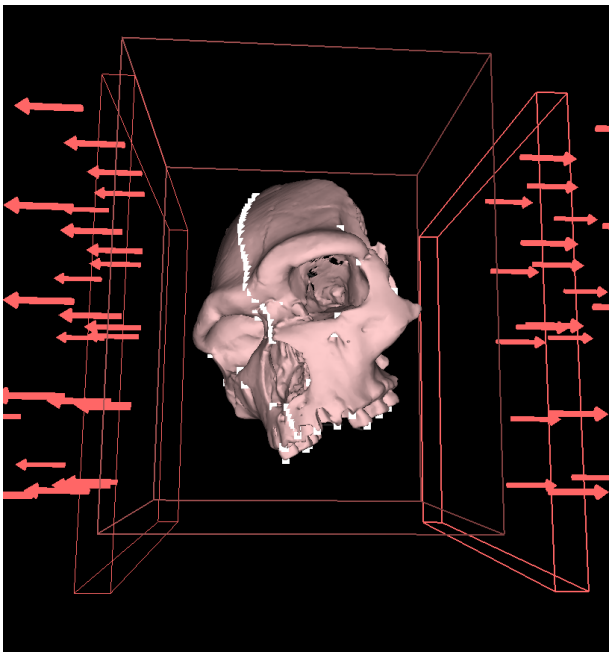


FIGURE 3.9 – AragoXXI rétro-déformé.

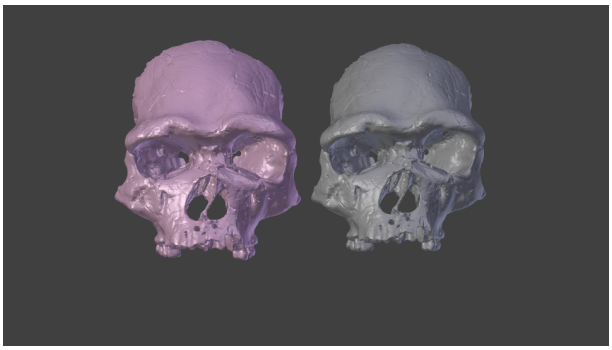


FIGURE 3.10 – Comparaison avec le modèle non rétro-déformé (en gris)

Chapitre 4

Expérience de rétro-déformation d'attaches parisiennes

4.1 Présentation

Le but de cette expérience est de valider la méthode de rétro-déformation employée pour sts52. Dans cette méthode, on suppose que le crâne a été déformé selon une seule direction sous la pression des sédiments. Pour retrouver la forme originale du crâne, on a imaginé l'expérience numérique suivante. Le crâne est placé dans une matière molle et on y applique une force. Il y a deux inconnues a déterminer dans ce modèle : la direction et l'intensité de la contrainte. Pour les déterminer, on a défini des critères évaluant la qualité de la rétro-déformation. Puis on a cherché la direction et l'intensité qui donnaient les meilleurs résultats. Dans la simulation, la déformation est plastique ; elle est calculée par éléments finis à l'aide de SOFA (un simulateur mécanique pour le domaine médical). Il y a donc quatre paramètres a déterminer : le module d'élasticité et le coefficient de poisson de la matière molle et du crâne.

Pour valider cette méthode j'ai effectué l'expérience suivante : je mets un objet déformable dans du sable. Je pose sur le sable un poids puis je récupère l'objet déformé. Enfin je numérise l'objet déformé et avec SOFA j'applique la méthode de rétro-déformation sur l'objet.

4.2 Le dispositif expérimental

l'objet utilisé est une attache parisienne à laquelle on a fait prendre la forme d'un U. La figure 4.1 présente l'objet. J'ai mis l'objet dans un seau rempli de sable mouillé (voir figures 4.2 et 4.3). J'ai utilisé du sable fin provenant de la plage et je l'ai mouillé car le sable sec est trop fluide. Le sable n'est pas tassé,

il a été mis a l'aide d'un tamis pour qu'il garde une certaine homogénéité (voir figure 4.5 et 4.6). Enfin j'ai appliqué un poids de 20 kg à l'aide d'un autre seau (voir figure 4.4). Le diamètre du fond du seaux est de 9 cm. Un poids de 20 kg permet de bien tasser le sable. Un poids inférieur n'est pas suffisant et ne déforme pas assez l'objet. On en déduit la contrainte exercée sur le sable par la formule $\frac{masse*g_0}{\pi*rayon^2}$ ce qui nous donne 3.08 Newton par centimètre carré. L'expérience a été réalisée deux fois avec des objet orientés différemment. La figure 4.7 préssente les différentes orientations. La figure 4.8 montre les objets déformés.

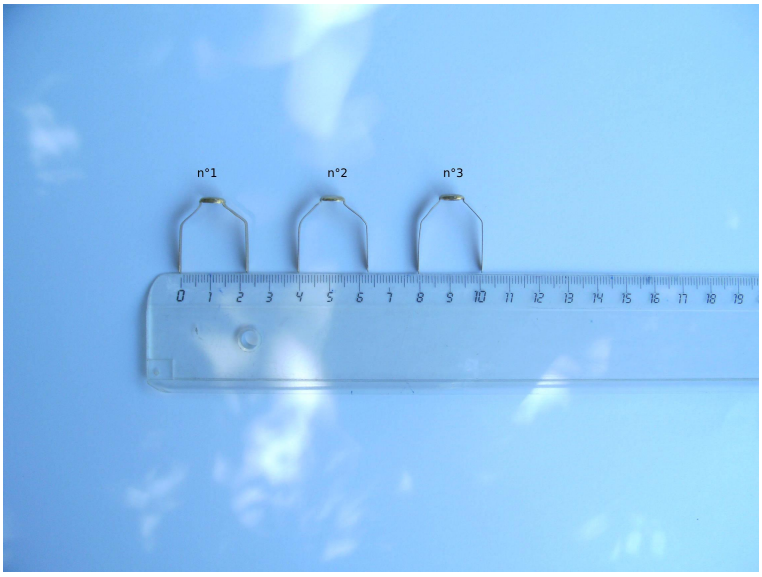


FIGURE 4.1 – Les attaches parisienne mises en forme de U avant la déformation.

FIGURE 4.2 – La première attache dans le sable.

FIGURE 4.3 – La seconde attache dans le sable.

FIGURE 4.4 – Le seau rempli de sable avec le poids.

FIGURE 4.5 – Le sable a disperser dans le seau a l'aide d'un tamis.

FIGURE 4.6 – Le seau rempli de sable avec le poids.

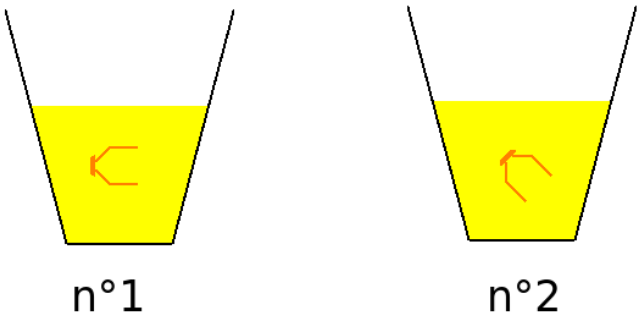


FIGURE 4.7 – Les différentes orientations des attaches dans le sable.

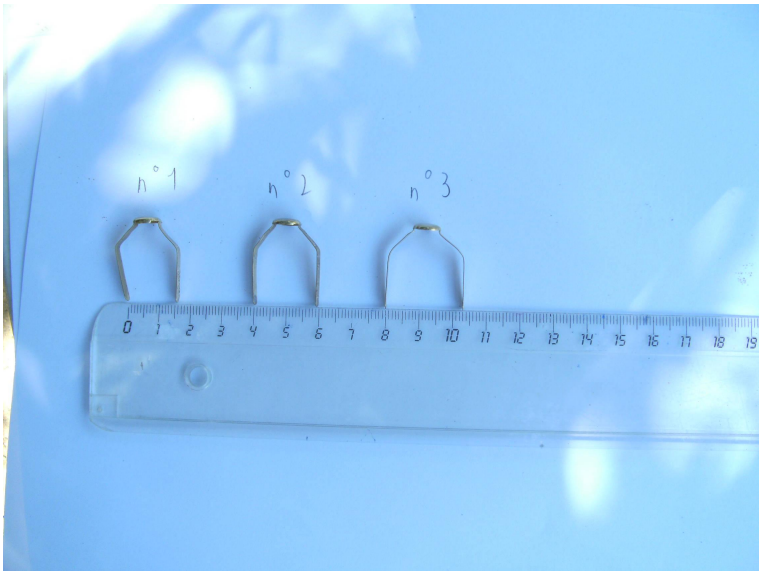


FIGURE 4.8 – Les attaches après déformation.

4.3 Modélisation

J'ai utilisé Blender pour modéliser l'attache parisienne. Le modèle est plus simple que l'attache. Il est composé d'un chapeau, de 4 cubes et de deux demis

cylindres. Le chapeau a été réalié par révolution et les différents éléments ont été assemblés. Dans le modèle, on a fixé l'épaisseur de l'attache a 2 millimètres. Les autres dimensions ont été mesurées a l'aide d'une règle. Les figures 4.9, 4.10 et 4.11 présentent l'attache modélisée. Les attaches ont aussi été numérisées a l'aide d'un scanner 3d. Le modèle doit être retravaillé avant d'être exploité.

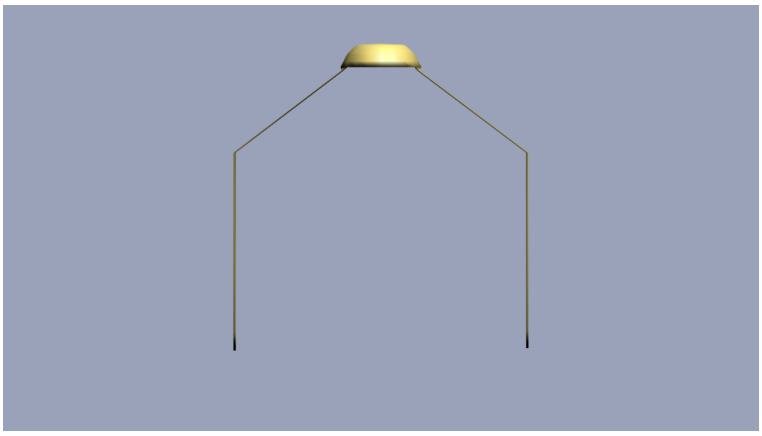


FIGURE 4.9 – L'attache n°1 modélisée.

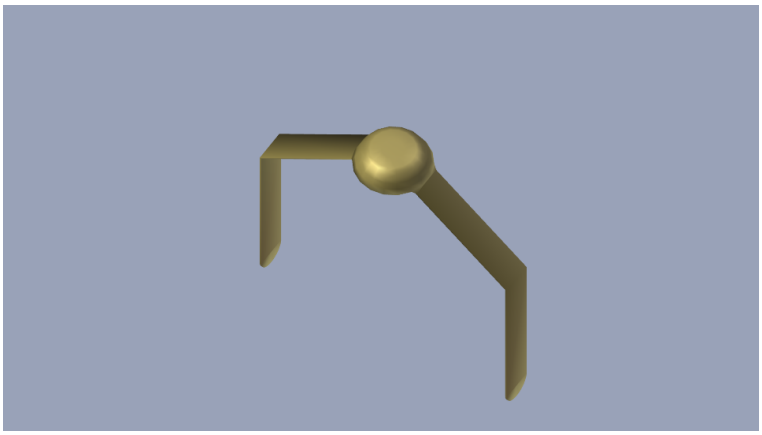


FIGURE 4.10 – L'attache n°1 modélisée.

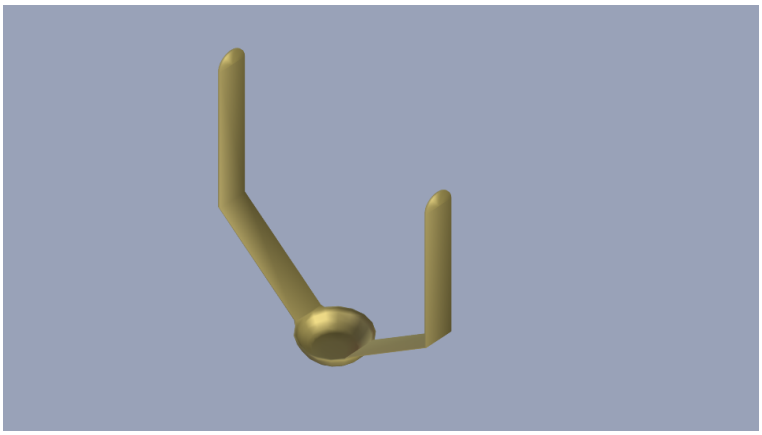


FIGURE 4.11 – L'attache n°1 modélisée.

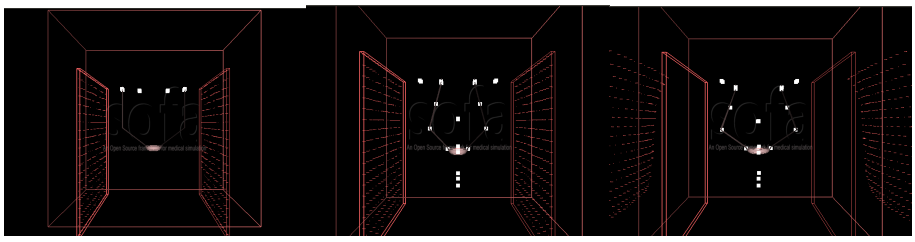
4.4 Simulation avec SOFA

Pour lancer la simulation avec SOFA il a fallu déterminer différents paramètres. Je n'ai pas d'informations sur la matière de l'attache parisienne mais la plus part d'entre elles sont fabriquées en laiton. D'après un document trouvé sur Wikipedia le module d'élasticité du laiton varie entre 102 et 128 GPa et son coefficient de poisson est 0,35. J'ai pris 128 GPa et 0,35 pour les caractéristiques de l'attache. Pour la matière molle j'ai gardé les même valeurs qu'avec sts52 soit un module d'élasticité valant 5 GPa et un coefficient de poisson nul.

La première simulation que j'ai faite est de mettre l'attache dans la bonne direction avec la bonne contrainte pour voir si on retrouvait l'attache originale. Mais ce ne fut pas le cas (voir figures 4.12a, 4.12b et 4.12c). J'ai essayé de modifier les diffèrents paramètres mais je ne suis pas arrivé à un résultat satisfaisant. Cela est peut être du à la modélisation de l'attache trop simpliste. On pourrait arrondir les angles ou peut être faut-il changer la simulation dans SOFA.

Néanmoins en cherchant la direction qui optimise un critère basé sur l'écartement des bouts des pattes on retrouve la direction de déformation. Le critère est le même que celui utilisé avec sts52. On a placé quatre points aux extrémités des pattes et on cherche avec les quatre points du modèle non déformé. Le modèle rétro-déformé qu'on obtient à la fin ne ressemble pas trop au modèle original (voir figure 4.12c) mais il est dans la bonne direction.

J'ai enfin tenté une denière simulation. J'ai fixé tous les paramètres connus : la direction et l'intensité de la contrainte, le module d'élasticité et le coefficient de poisson de l'attache. J'ai lancé le programme de recherche pour trouver le module d'élasticité et le coefficient de poisson de la matière molle donnant le meilleur résultat. A la fin de la recherche on avait 0,266 pour module d'élasticité et 0,2833 comme coefficient de poisson. Mais encore une fois le modèle rétro-déformé obtenu ne correspond pas au modèle original.



(a) Attache d'origine. (b) Attache déformée. (c) Attache après une rétro déformation.

FIGURE 4.12 – Test de rétro-déformation.

Annexe A

Le fichier XML décrivant la scène

```
<?xml version="1.0"?>
<Node   name="root" gravity="0 0 0" dt="0.5" animate="true" >
        <VisualStyle displayFlags="showVisual" />

        <!-- filename est le nom du mesh de l'objet a charger. -->
        <!- ARemplacerRota est remplace automatiquement par le script python. C
        <MeshObjLoader name="mesh" filename="../modele/attache1.obj" triangulate
        <!-- Les composants suivant charge et creent l'image 3d. -->
        <!- padSize determine la taille de l'image donc la taille du bord. -->
        <MeshToImageEngine template="ImageB" name="rasterizer" src="@mesh" voxel
        <TransferFunction name="allFilled" template="ImageB,ImageB" inputImage="
        <ImageContainer template="ImageB" name="image" image="@allFilled.outputIm

        <!-- param determine la taille de la grille fem. -->
        <ImageFilter template="ImageB,ImageB" name="subsample" filter="7" param=
        <ImageSampler template="ImageB" name="sampler" image="@subsample.outputIm
        <Mesh name="mesh" src="@sampler" />

        <MechanicalObject name="parent" useMask="0" src="@sampler" />
        <UniformMass totalMass="250" />

        <StaticSolver />
        <CGLinearSolver iterations="25" />

        <BarycentricShapeFunction nbRef="8" />

        <!-- Le noeuds behavior est le coeur de la simulation, il cree la grille
        <Node name="behavior" >
```

```
                <TopologyGaussPointSampler name="sampler" inPosition="@../sample
                <MechanicalObject  template="F331" name="F"
showObject="0" showObjectScale="0.05" />
                <LinearMapping template="Mapping&lt;Vec3d,F331&gt;" assembleJ="f

                <Node name="E" >
                        <!-- param determine les raideurs associees a 0 (dehors)
                        <TransferFunction name="youngTF" template="ImageB,ImageD
                        <ImageValuesFromPositions name="youngM" position="@../sa
                        <!-- param determine les coeffisients de poisson associe
                        <TransferFunction name="poissonTF" template="ImageB,Imag
                        <ImageValuesFromPositions name="poissonR" position="@../

                        <MechanicalObject  template="E331" name="E"
/>
                        <GreenStrainMapping template="Mapping&lt;F331,E331&gt;"
                        <HookeForceField  template="E331" name="ff" youngModulus
                </Node>
        </Node>

        <!-- Les noeuds suivant sont des points de controlle qui servent a calcu
        <!-- Le compossant <LinearMapping template="MechanicalMapping&lt;Vec3d,E
        <!-- La position des points est initialisee dans le scripte python. -->
        <Node name="particle_node1">
                <PythonScriptController filename="attache_tout_param.py"/>
        </Node>
        <Node name="particle_node2">
                <PythonScriptController filename="attache_tout_param.py"/>
        </Node>
        <Node name="particle_node3">
                <PythonScriptController filename="attache_tout_param.py"/>
        </Node>
        <Node name="particle_node4">
                <PythonScriptController filename="attache_tout_param.py"/>
        </Node>
        <Node name="particle_node5">
                <PythonScriptController filename="attache_tout_param.py"/>
                <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node6">
                <PythonScriptController filename="attache_tout_param.py"/>
                <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node7">
                <PythonScriptController filename="attache_tout_param.py"/>
                <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
```

```
        </Node>
        <Node name="particle_node8">
            <PythonScriptController filename="attache_tout_param.py"/>
            <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node9">
            <PythonScriptController filename="attache_tout_param.py"/>
            <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node10">
            <PythonScriptController filename="attache_tout_param.py"/>
            <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node11">
            <PythonScriptController filename="attache_tout_param.py"/>
            <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node12">
            <PythonScriptController filename="attache_tout_param.py"/>
            <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node13">
            <PythonScriptController filename="attache_tout_param.py"/>
            <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node14">
            <PythonScriptController filename="attache_tout_param.py"/>
            <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node15">
            <PythonScriptController filename="attache_tout_param.py"/>
            <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node16">
            <PythonScriptController filename="attache_tout_param.py"/>
            <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node17">
            <PythonScriptController filename="attache_tout_param.py"/>
            <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node18">
            <PythonScriptController filename="attache_tout_param.py"/>
            <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node19">
```

```
                <PythonScriptController filename="attache_tout_param.py"/>
                <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node20">
                <PythonScriptController filename="attache_tout_param.py"/>
                <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node21">
                <PythonScriptController filename="attache_tout_param.py"/>
                <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node22">
                <PythonScriptController filename="attache_tout_param.py"/>
                <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node23">
                <PythonScriptController filename="attache_tout_param.py"/>
                <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node24">
                <PythonScriptController filename="attache_tout_param.py"/>
                <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>
        <Node name="particle_node25">
                <PythonScriptController filename="attache_tout_param.py"/>
                <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;"
        </Node>

        <!-- Ce noeud permet de visualiser l'objet. -->
                <Node name="visual"    >
                        <VisualModel  fileMesh="../modele/attache1.obj" color="1 0.8 0.8
                        <LinearMapping template="MechanicalMapping&lt;Vec3d,ExtVec3f&gt;
                </Node>

                <!-- Ce noeud permet de simuler la contrainte par un champs de force. --
                <Node name="CollisionBox">
                        <VisualStyle displayFlags="showWireframe showVisual showBehavior

                        <!-- On cree un cube rempli de points. Les attributs min et max
nx, ny et nz definissent le nombre de points qu'il y aura sur chaque axes.-->
                <CubeTopology name="topo" nx="20" ny="20" nz="20" min="-23 -23 -
                <MechanicalObject name="PointSet" useMask="false"/>
                <Triangle />
                <LinearMapping template="MechanicalMapping&lt;Vec3d,Vec3d&gt;"/>

                <!-- Puis on cree un boite. Tout le point contenu dedans seront
```

```
                <BoxROI template="Vec3d" box="-22.5 -23 -23 -23.5 23 23" drawBox
                <!-- Le parametre force determine d'intensiter de la force appli
                <ConstantForceField template="Vec3d" name="default6" points="@RO

                <BoxROI template="Vec3d" box="22.5 -23 -23 23.5 23 23" drawBoxes
                <ConstantForceField template="Vec3d" name="default6" points="@RO
                </Node>

</Node>
```

Annexe B

Le scripte pour le module sofaPython

```
import Sofa
import math
import copy
import os

########################################################################################
# script attache_retro_deformation_scene.py
########################################################################################
#
#
#
#
#

# Cette variable est une liste qui contidera les references des objets python pe
particles = []

## Les varibles suivantes definissent les coordones des points de repere.

# Les points de reference. Ils ne bougent pas.
posPartXAlinBas =       [1.5    ,-1.5    ,1.5     ,-1.5    ]
posPartYAlinBas =       [-22.65 ,-22.65 ,-22.65 ,-22.65 ]
posPartZAlinBas =       [-12.5  ,-12.5   ,12.5    ,12.5    ]

# Les points a aligner avec les points de reference.
posPartXAlinHaut =      [1.5    ,-1.5    ,1.5     ,-1.5    ]
posPartYAlinHaut =      [-22.15 ,-22.15 ,-22.15 ,-22.15 ]
posPartZAlinHaut =      [-5.6   ,-5.6    ,5.6     ,5.6     ]
```

```
# Les points definisant le plan de symetrie.
posPartXPlanSym =       [0      ,0      ,0      ,-10    ,10     ,10     ,-10
posPartYPlanSym =       [0      ,10     ,-10    ,0      ,0      ,10     ,-10
posPartZPlanSym =       [0      ,0      ,0      ,0      ,0      ,0      ,0

# Les points de repere pour le critere de symetrie.
posPartXSymGauche =     [0      ,0      ,0      ,0      ]
posPartYSymGauche =     [-22.6  ,-15.0  ,-6.8   ,0.0    ]
posPartZSymGauche =     [-5.65  ,-7.3   ,-9.4   ,-3.4   ]

# Les points de repere pour le critere de symetrie.
posPartXSymDroite =     [0      ,0      ,0      ,0      ]
posPartYSymDroite =     [-22.65 ,-15.0  ,-6.8   ,0      ]
posPartZSymDroite =     [5.6    ,7.3    ,9.4    ,3.4    ]

posPartX = posPartXAlinBas + posPartXAlinHaut + posPartXPlanSym + posPartXSymGau
posPartY = posPartYAlinBas + posPartYAlinHaut + posPartYPlanSym + posPartYSymGau
posPartZ = posPartZAlinBas + posPartZAlinHaut + posPartZPlanSym + posPartZSymGau

# On lit le fichier direction. Dedans il est ecrit l'angle de rotation de l'obje
src = open(os.path.dirname(os.path.abspath(__file__))+"/direction", "r")
rotX = float(src.readline().rstrip('\n\r'))
rotY = float(src.readline().rstrip('\n\r'))
rotZ = float(src.readline().rstrip('\n\r'))
src.close()

nbExecutionsMax = 0
nbExecutions = 0

# Cette fonction est appele lorsque le composant est charche.
# A ce moment la, on va creer un point de repere.
# A chaque appelle de cette fonction, on cree un point de repere.
# Le premier point est insialiser avec la premiere valeur dans posPartX, posPart
# Il doit donc avoir autant de points definis dans la scene que de coordones def
def onLoaded(node):
 global nbExecutionsMax
 nbExecutionsMax = nbExecutionsMax + 1

 global particles
 global posPartX
 global posPartY
 global posPartZ
 part = Sofa.createObject(node,Sofa.BaseObjectDescription("particle{0}".format(l
 part.findData('position').value = [posPartX[len(particles)], posPartY[len(parti
 part.findData('showObject').value = '1'
```

```
 part.findData('showObjectScale').value = '10'
 part.findData('rotation').value = [rotX, rotY, rotZ]
 particles.append(part)

 return 0


oldErrRes = 0
errRes = 0

# Cette fonction est appelee a chaque iteration de la simulation. Elle est aussi
# Donc pour eviter que le critere soit calculer trop de fois on utilise les vari
# nbExecutionsMax stock un chiffre qui est egale au nombre de fois que cette fon
def onBeginAnimationStep(dt):

 global nbExecutions
 global nbExecutionsMax
 if nbExecutions == nbExecutionsMax-1:
  # Calcul du critere d'alignement.
  global particles
  particlesPlacmentDentHaut = particles[0:len(posPartXAlinHaut)]
  particlesPlacmentDentBas = particles[len(posPartXAlinHaut):len(posPartXAlinHau
  errAllin = 0
  for i in range(0,len(posPartXAlinHaut)-1):
   for j in range(i+1,len(posPartXAlinHaut)):
    vectMaxilar = []
    vectMandibul = []
    for k in range(0,3):
     vectMaxilar.append([ particlesPlacmentDentHaut[i].findData('position').valu
     vectMandibul.append([ particlesPlacmentDentBas[i].findData('position').valu
    errAllin = errAllin + (norm(vectMaxilar) - norm(vectMandibul))**2

  errAllin = math.sqrt(errAllin)/len(posPartXAlinHaut)

  # Calcul du critere de symetrie.
  particlesPlanSym = particles[len(posPartXAlinHaut)+len(posPartXAlinBas):len(po
  # Evaluation de l'equation du plan par la methode d'analyse par composante pri
  M=[[],[],[]]
  for part in particlesPlanSym:
   M[0].append(part.findData('position').value[0])
   M[1].append(part.findData('position').value[1])
   M[2].append(part.findData('position').value[2])
  Xcentre = [[sommeElt([elt])/len(M)] for elt in M]
  Mcentre = diffTaT(M,dot(Xcentre,[[1 for elt in M[0]]]))
  McentreT = transpose(Mcentre)
  U, s, Vt = svd(McentreT)
```

```
V=transpose(Vt)
minimum = s[0]
indice = 0
for i,elt in enumerate(s):
 if elt < minimum:
  minimum = elt
  indice = i
indicMax = [0,1,2]
indicMax.remove(indice)
noramalSVD = produitVectorielle(V[indicMax[0]],V[indicMax[1]])
dSVD = -produitScalaire(noramalSVD,Xcentre)
# Calcul du critere de symetrie.
particlesSymGauche = particles[len(posPartXAlinHaut)+len(posPartXAlinBas)+len(
particlesSymDroite = particles[len(posPartXAlinHaut)+len(posPartXAlinBas)+len(
normalPlan = noramalSVD
d = dSVD
errSym = 0
# L'erreur de sym est la distance entre le point de gauche et le point symetri
for i,part in enumerate(particlesSymGauche):
 partInit = [[part.findData('position').value[0]],[part.findData('position').v
 partSym = sommeTaT(mulScal((-2)*(produitScalaire(partInit,normalPlan)+d)/prod
 partDroit = []
 for k in [0,1,2]:
  partDroit.append([particlesSymDroite[i].findData('position').value[k]])
 diffVect = diffTaT(partDroit,partSym)
 errSym = errSym + produitScalaire(diffVect,diffVect)

errSym = math.sqrt(errSym)/len(particlesSymGauche)

# Ecriture des resultats dans un fichier.
f_resultat_sym = open(os.path.dirname(os.path.abspath(__file__))+"/errattache_s
f_resultat_sym.write("{0};".format(errSym))
f_resultat_sym.close()

# Ecriture des resultats dans un fichier.
f_resultat_align = open(os.path.dirname(os.path.abspath(__file__))+"/errattach
f_resultat_align.write("{0};".format(errAllin))
f_resultat_align.close()

# Calcul du critere globale.
err = math.sqrt(errAllin**2 + errSym**2)

# Ecriture des resultats dans un fichier.
f_resultat = open(os.path.dirname(os.path.abspath(__file__))+"/errattache.csv"
f_resultat.write("{0};".format(err))
f_resultat.close()
```

```
    # Ecriture du dernier resultat dans le fichier res.
    global oldErrRes
    global errRes
    oldErrRes = errRes
    errRes = errAllin
    if abs(oldErrRes-errRes)>0.01:
     # Le resutat n'est pas stable on renvoie donc -1 pour ignorer le resultat.
     err = -1
    else:
     err = errRes
    f_der_res = open(os.path.dirname(os.path.abspath(__file__))+"/res", "w")
    f_der_res.write("{0}".format(err))
    f_der_res.close()

    nbExecutions = 0
   else:
    nbExecutions += 1

   return 0

# Toutes les fonctions ci dessu manipulent des listes de listes, pour faire du c
# Ces listes de listes representants generalement des matrices. La taille des li
# Les vecteurs sont reprenter soit par une liste contenant des listes de taille

# Fonction transposant une matrice.
def transpose(mat):
 matRet = []
 j=0
 while j < len(mat[0]):
  matRet.append([])
  i=0
  while i < len(mat):
   matRet[j].append(mat[i][j])
   i=i+1
  j=j+1
 return matRet

# Fonction multipliant une matrice par un scalaire.
def mulScal(sca,mat):
 matRet = []
 for elt in mat:
  matRet.append(list(elt))
 i=0
 while i < len(mat):
  j=0
```

```
    while j < len(mat[i]):
     matRet[i][j] = mat[i][j]*sca
     j=j+1
    i=i+1
  return matRet

# Fonction multipliant terme a terme de deux matrices.
def mulTaT(mat1,mat2):
  matRet = []
  for elt in mat1:
    matRet.append(list(elt))
  i=0
  while i < len(matRet):
    j=0
    while j < len(matRet[i]):
     matRet[i][j] = matRet[i][j]*mat2[i][j]
     j=j+1
    i=i+1
  return matRet

# Fonction calculant la somme terme a terme de deux matrices.
def sommeTaT(mat1,mat2):
  matRet = []
  for elt in mat1:
    matRet.append(list(elt))
  i=0
  while i < len(matRet):
    j=0
    while j < len(matRet[i]):
     matRet[i][j] = matRet[i][j]+mat2[i][j]
     j=j+1
    i=i+1
  return matRet

# Fonction calculant la difference terme a terme de deux matrices.
def diffTaT(mat1,mat2):
  matRet = []
  for elt in mat1:
    matRet.append(list(elt))
  i=0
  while i < len(matRet):
    j=0
    while j < len(matRet[i]):
     matRet[i][j] = matRet[i][j]-mat2[i][j]
     j=j+1
    i=i+1
```

```
 return matRet

# Fonction calculant la somme de tous les termes de la matrice.
def sommeElt(mat):
 ret=0
 i=0
 while i < len(mat):
  j=0
  while j < len(mat[i]):
   ret = ret + mat[i][j]
   j=j+1
  i=i+1
 return ret

# Fonction qui renvoie le produit vectoriel de deux vecteurs de dimention 3.
def produitVectorielle(vect1,vect2):
 return [[vect1[1]*vect2[2]-vect2[1]*vect1[2]],[vect1[2]*vect2[0]-vect2[2]*vect1

# Fonction calculant le produit scalair de deux vecteurs.
def produitScalaire(vect1,vect2):
 return sommeElt(mulTaT(vect1,vect2))

# Fonction calculant la norme d'un vecteur.
def norm(mat):
 return math.sqrt(sommeElt(mulTaT(mat,mat)))

# Fonction qui extrait une colonne d'une matrice.
def getColonne(mat,indice):
 vectRet = []
 for elt in mat:
  vectRet.append([elt[indice]])
 return vectRet

# Fonction calculant le produit matricielle de deux matrices.
def dot(mat1,mat2):
 matRet = []
 i=0
 while i < len(mat1):
  matRet.append([])
  j=0
  while j < len(mat2[0]):
   matRet[i].append(produitScalaire(transpose([mat1[i]]),getColonne(mat2,j)))
   j=j+1
  i=i+1
 return matRet
```

```
# Fonction resolvant un syteme lineaire par la methode de gauss.
def solveGauss(mat,b):
 n=len(mat)
 B = []
 i=0
 while i < n:
  B.append(list(mat[i]+[b[i][0]]))
  i=i+1
 i=0
 while i < n-1:
  j=i+1
  while j < n:
   m=B[j][i]/B[i][i]
   k=i
   while k < n+1:
    B[j][k]=B[j][k] - m*B[i][k]
    k=k+1
   j=j+1
  i=i+1
 x = getColonne(B,0)
 x[n-1][0] = B[n-1][n]/B[n-1][n-1]
 i=n-2
 while i >= 0:
  somme=0
  j=i+1
  while j < n:
   somme=somme+B[i][j]*x[j][0]
   j=j+1
  x[i][0]=(B[i][n] - somme)/B[i][i]
  i=i-1
 return x

# Almost exact translation of the ALGOL SVD algorithm published in
# Numer. Math. 14, 403-420 (1970) by G. H. Golub and C. Reinsch
#
# Copyright (c) 2005 by Thomas R. Metcalf, helicity314-stitch <at> yahoo <dot> c
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
```

```
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
#
# Pure Python SVD algorithm.
# Input: 2-D list (m by n) with m >= n
# Output: U,W V so that A = U*W*VT
#  Note this program returns V not VT (=transpose(V))
#  On error, a ValueError is raised.
#
# Here is the test case (first example) from Golub and Reinsch
#
# a = [[22.,10., 2.,  3., 7.],
#   [14., 7.,10.,  0., 8.],
#   [-1.,13.,-1.,-11., 3.],
#   [-3.,-2.,13., -2., 4.],
#   [ 9., 8., 1., -2., 4.],
#   [ 9., 1.,-7.,  5.,-1.],
#   [ 2.,-6., 6.,  5., 1.],
#   [ 4., 5., 0., -2., 2.]]
#
# import svd
# import math
# u,w,vt = svd.svd(a)
# print w
#
# [35.327043465311384, 1.2982256062667619e-15,
#  19.999999999999996, 19.595917942265423, 0.0]
#
# the correct answer is (the order may vary)
#
# print (math.sqrt(1248.),20.,math.sqrt(384.),0.,0.)
#
# (35.327043465311391, 20.0, 19.595917942265423, 0.0, 0.0)
#
# transpose and matrix multiplication functions are also included
# to facilitate the solution of linear systems.
#
# Version 1.0 2005 May 01

def svd(a):
 #'''Compute the singular value decomposition of array.'''

 # Golub and Reinsch state that eps should not be smaller than the
 # machine precision, ie the smallest number
```

```
# for which 1+e>1.  tol should be beta/e where beta is the smallest
# positive number representable in the computer.
eps = 1.e-15  # assumes double precision
tol = 1.e-64/eps
assert 1.0+eps > 1.0 # if this fails, make eps bigger
assert tol > 0.0  # if this fails, make tol bigger
itmax = 50
u = copy.deepcopy(a)
m = len(a)
n = len(a[0])
#if __debug__: print 'a is ',m,' by ',n

if m < n:
 if __debug__: print 'Error: m is less than n'
 raise ValueError,'SVD Error: m is less than n.'

e = [0.0]*n  # allocate arrays
q = [0.0]*n
v = []
for k in range(n): v.append([0.0]*n)

# Householder's reduction to bidiagonal form

g = 0.0
x = 0.0

for i in range(n):
 e[i] = g
 s = 0.0
 l = i+1
 for j in range(i,m): s += (u[j][i]*u[j][i])
 if s <= tol:
  g = 0.0
 else:
  f = u[i][i]
  if f < 0.0:
   g = math.sqrt(s)
  else:
   g = -math.sqrt(s)
  h = f*g-s
  u[i][i] = f-g
  for j in range(l,n):
   s = 0.0
   for k in range(i,m): s += u[k][i]*u[k][j]
   f = s/h
   for k in range(i,m): u[k][j] = u[k][j] + f*u[k][i]
```

```
 q[i] = g
 s = 0.0
 for j in range(l,n): s = s + u[i][j]*u[i][j]
 if s <= tol:
  g = 0.0
 else:
  f = u[i][i+1]
  if f < 0.0:
   g = math.sqrt(s)
  else:
   g = -math.sqrt(s)
  h = f*g - s
  u[i][i+1] = f-g
  for j in range(l,n): e[j] = u[i][j]/h
  for j in range(l,m):
   s=0.0
   for k in range(l,n): s = s+(u[j][k]*u[i][k])
   for k in range(l,n): u[j][k] = u[j][k]+(s*e[k])
 y = abs(q[i])+abs(e[i])
 if y>x: x=y
# accumulation of right hand gtransformations
for i in range(n-1,-1,-1):
 if g != 0.0:
  h = g*u[i][i+1]
  for j in range(l,n): v[j][i] = u[i][j]/h
  for j in range(l,n):
   s=0.0
   for k in range(l,n): s += (u[i][k]*v[k][j])
   for k in range(l,n): v[k][j] += (s*v[k][i])
 for j in range(l,n):
  v[i][j] = 0.0
  v[j][i] = 0.0
 v[i][i] = 1.0
 g = e[i]
 l = i
#accumulation of left hand transformations
for i in range(n-1,-1,-1):
 l = i+1
 g = q[i]
 for j in range(l,n): u[i][j] = 0.0
 if g != 0.0:
  h = u[i][i]*g
  for j in range(l,n):
   s=0.0
   for k in range(l,m): s += (u[k][i]*u[k][j])
   f = s/h
```

```
    for k in range(i,m): u[k][j] += (f*u[k][i])
   for j in range(i,m): u[j][i] = u[j][i]/g
  else:
   for j in range(i,m): u[j][i] = 0.0
  u[i][i] += 1.0
#diagonalization of the bidiagonal form
eps = eps*x
for k in range(n-1,-1,-1):
 for iteration in range(itmax):
  # test f splitting
  for l in range(k,-1,-1):
   goto_test_f_convergence = False
   if abs(e[l]) <= eps:
    # goto test f convergence
    goto_test_f_convergence = True
    break  # break out of l loop
   if abs(q[l-1]) <= eps:
    # goto cancellation
    break  # break out of l loop
  if not goto_test_f_convergence:
   #cancellation of e[l] if l>0
   c = 0.0
   s = 1.0
   l1 = l-1
   for i in range(l,k+1):
    f = s*e[i]
    e[i] = c*e[i]
    if abs(f) <= eps:
     #goto test f convergence
     break
    g = q[i]
    h = pythag(f,g)
    q[i] = h
    c = g/h
    s = -f/h
    for j in range(m):
     y = u[j][l1]
     z = u[j][i]
     u[j][l1] = y*c+z*s
     u[j][i] = -y*s+z*c
  # test f convergence
  z = q[k]
  if l == k:
   # convergence
   if z<0.0:
    #q[k] is made non-negative
```

```
    q[k] = -z
    for j in range(n):
     v[j][k] = -v[j][k]
   break  # break out of iteration loop and move on to next k value
  if iteration >= itmax-1:
   if __debug__: print 'Error: no convergence.'
   # should this move on the the next k or exit with error??
   #raise ValueError,'SVD Error: No convergence.'  # exit the program with erro
   break  # break out of iteration loop and move on to next k
  # shift from bottom 2x2 minor
  x = q[l]
  y = q[k-1]
  g = e[k-1]
  h = e[k]
  f = ((y-z)*(y+z)+(g-h)*(g+h))/(2.0*h*y)
  g = pythag(f,1.0)
  if f < 0:
   f = ((x-z)*(x+z)+h*(y/(f-g)-h))/x
  else:
   f = ((x-z)*(x+z)+h*(y/(f+g)-h))/x
  # next QR transformation
  c = 1.0
  s = 1.0
  for i in range(l+1,k+1):
   g = e[i]
   y = q[i]
   h = s*g
   g = c*g
   z = pythag(f,h)
   e[i-1] = z
   c = f/z
   s = h/z
   f = x*c+g*s
   g = -x*s+g*c
   h = y*s
   y = y*c
   for j in range(n):
    x = v[j][i-1]
    z = v[j][i]
    v[j][i-1] = x*c+z*s
    v[j][i] = -x*s+z*c
   z = pythag(f,h)
   q[i-1] = z
   c = f/z
   s = h/z
   f = c*g+s*y
```

```
      x = -s*g+c*y
      for j in range(m):
       y = u[j][i-1]
       z = u[j][i]
       u[j][i-1] = y*c+z*s
       u[j][i] = -y*s+z*c
     e[l] = 0.0
     e[k] = f
     q[k] = x
     # goto test f splitting


  #vt = transpose(v)
  #return (u,q,vt)
  return (u,q,v)

def pythag(a,b):
  absa = abs(a)
  absb = abs(b)
  if absa > absb: return absa*math.sqrt(1.0+(absb/absa)**2)
  else:
   if absb == 0.0: return 0.0
   else: return absb*math.sqrt(1.0+(absa/absb)**2)

def transpose(a):
  '''Compute the transpose of a matrix.'''
 m = len(a)
 n = len(a[0])
 at = []
 for i in range(n): at.append([0.0]*m)
 for i in range(m):
  for j in range(n):
   at[j][i]=a[i][j]
 return at

def matrixmultiply(a,b):
  '''Multiply two matrices.
  a must be two dimensional
  b can be one or two dimensional.'''

 am = len(a)
 bm = len(b)
 an = len(a[0])
 try:
  bn = len(b[0])
 except TypeError:
```

```
  bn = 1
 if an != bm:
  raise ValueError, 'matrixmultiply error: array sizes do not match.'
 cm = am
 cn = bn
 if bn == 1:
  c = [0.0]*cm
 else:
  c = []
  for k in range(cm): c.append([0.0]*cn)
 for i in range(cm):
  for j in range(cn):
   for k in range(an):
    if bn == 1:
     c[i] += a[i][k]*b[k]
    else:
     c[i][j] += a[i][k]*b[k][j]

 return c


# optionnally, script can create a graph...
def createGraph(node):
 #print 'createGraph called (python side) from node %s'%node.findData('name').va
 return 0

# called once graph is created, to init some stuff...
def initGraph(node):
 #print 'initGraph called (python side) from node %s'%node.findData('name').valu
 return 0

def onEndAnimationStep(dt):
 return 0

# called when necessary by Sofa framework...
def storeResetState():
 #print 'storeResetState called (python side)'
 return 0

def reset():
 #print 'reset called (python side)'
 return 0

def cleanup():
 #print 'cleanup called (python side)'
 return 0
```

```
# called when a GUIEvent is received
def onGUIEvent(controlID, valueName, value):
 return 0

# key and mouse events; use this to add some user interaction to your scripts
def onKeyPressed(k):
 return 0

def onKeyReleased(k):
 return 0

def onMouseButtonLeft(x, y, pressed):
 return 0

def onMouseButtonRight(x, y, pressed):
 return 0

def onMouseButtonMiddle(x, y, pressed):
 return 0

def onMouseWheel(x, y, delta):
 return 0
```

Annexe C

Le scripte python cherchant le minimum du critère de rétro-déformation

```
#!/usr/bin/python3.2

import functools
import copy
import os
import math
import time
import numpy as np

#############################################################################
# script
#############################################################################
#
#


# Fonction d'evalution du scrite, ici on evalu la performence d'une retro-deform
# X[0],X[1] direction de la contrainte en coordone spherique.
# X[2] Raideur de la matiere molle.
# X[3] Raideur de l'objet.
# X[4] Coeffisient de poisson de la matiere molle.
# X[5] Coeffisient de poisson de l'objet.
# X[6] Intensite de la contrainte.

def fonctionGen(X):
```

```
# Definit l'emplacement de sofa.
runSofa = "/home/canu/Documents/stage/Sofa/bin/runSofa"

# On nome les parametres pour une meillieur comprehension.
angleEuler = matRotToEuler(X[0],X[1])
angleX = angleEuler[0]
angleY = angleEuler[1]
angleZ = angleEuler[2]
raideurExt = abs(X[2])
raideurObj = abs(X[3])
poissonExt = X[4]
poissonObj = X[5]
normeForce = X[6]

# On ecrit dans un fichier la direction de recherche. Ce fichier vas etr
f_dir = open(os.path.dirname(os.path.abspath(__file__))+"/direction", "w
f_dir.write("{0}\n".format(angleX))
f_dir.write("{0}\n".format(angleY))
f_dir.write("{0}\n".format(angleZ))
f_dir.close()

# On lit le fichier model de scene et le copie en specifiant la bonne di
src = open(os.path.dirname(os.path.abspath(__file__))+"/attache_tout_para
dst = open(os.path.dirname(os.path.abspath(__file__))+"/attache_tout_par
for ligne in src:
        donnees = ligne.rstrip('\n\r')
        if "ARemplacerRota" in donnees:
                donnees = donnees.replace("ARemplacerRota","{0} {1} {2}"
        if "ARemplacerRaideurs" in donnees:
                donnees = donnees.replace("ARemplacerRaideurs","0 {0} 1 
        if "ARemplacerPoisson" in donnees:
                donnees = donnees.replace("ARemplacerPoisson","0 {0} 1 {
        if "ARemplacerNormeForce1" in donnees:
                donnees = donnees.replace("ARemplacerNormeForce1","-{0} 
        if "ARemplacerNormeForce2" in donnees:
                donnees = donnees.replace("ARemplacerNormeForce2","{0} 0
        dst.write(donnees+'\n')
src.close()
dst.close()

# On prepare le ficher resultat pour le critere global (alignement des d
f_resultat = open(os.path.dirname(os.path.abspath(__file__))+"/errattach
f_resultat.write("\n")
f_resultat.write("{0};{1};{2};{3};{4};{5};{6};{7};{8};{9};;".format(angle
f_resultat.close()
```

```
# On prepare le ficher resultat pour le critere global (alignement des d
f_resultat_align = open(os.path.dirname(os.path.abspath(__file__))+"/err
f_resultat_align.write("\n")
f_resultat_align.write("{0};{1};{2};{3};{4};{5};{6};{7};{8};{9};;".forma
f_resultat_align.close()

# On prepare le ficher resultat pour le critere global (alignement des d
f_resultat_sym = open(os.path.dirname(os.path.abspath(__file__))+"/erratt
f_resultat_sym.write("\n")
f_resultat_sym.write("{0};{1};{2};{3};{4};{5};{6};{7};{8};{9};;".format(
f_resultat_sym.close()

# On prepare le fichier contant le resultat qui va etre calculer par le
f_der_resultat = open(os.path.dirname(os.path.abspath(__file__))+"/res",
f_der_resultat.close()

# On execute sofa.
os.system(runSofa+" "+os.path.dirname(os.path.abspath(__file__))+"/attach
time.sleep(60)
os.system("pkill -SIGKILL runSofa")

# On recupere les resultats ecrit dans le fichier res.
f_der_resultat = open(os.path.dirname(os.path.abspath(__file__))+"/res",
resStr =f_der_resultat.readline().rstrip('\n\r')
f_der_resultat.close()

# Si le fichier res est vide cela veut dire qu'il y a eu une erreur donc
if resStr == "":
        os.system(runSofa+" "+os.path.dirname(os.path.abspath(__file__))-
        time.sleep(60)
        os.system("pkill -SIGKILL runSofa")
        f_der_resultat = open(os.path.dirname(os.path.abspath(__file__))-
        resStr =f_der_resultat.readline().rstrip('\n\r')
        f_der_resultat.close()

# Si le fichier res ne contient pas un nombre cela veut dir qu'il y a eu
try:
        resFloat = float(resStr)
except:
        resFloat = 100000
        f_instable = open(os.path.dirname(os.path.abspath(__file__))+"/p
        f_instable.write("\n")
        f_instable.write("angle : {0},{1},{2},{3},{4} ; raideur : {5},{6
        f_instable.close()
```

```
        # Si on a -1 c'est que le calcul dans sofa n'etait pas stabiliser donc o
        if resFloat == -1:
                resFloat = 100000
                f_instable = open(os.path.dirname(os.path.abspath(__file__))+"/p
                f_instable.write("\n")
                f_instable.write("angle : {0},{1},{2},{3},{4} ; raideur : {5},{6
                f_instable.close()

        return resFloat

# Les fonctions suivantes servent au scripte.
# Fonction qui remplace un element dans une liste par un meillieur element.
def replaceParMeillieur(listVal,listElt,val,elt):
        ret = []
        if not elt in listElt:
                k=0
                while k<len(listVal):
                        if val < listVal[k]:
                                ret=[copy.copy(listVal[k]),copy.copy(listElt[k])
                                listVal[k] = copy.copy(val)
                                listElt[k] = copy.copy(elt)
                                k=len(listVal)
                        k = k+1
        return ret

# Fonction qui convertit un axe de rotation et un angle en Quad.
def axeRotToQuad(axeX,axeY,axeZ,angle):
        angle = np.radians(angle)
        sinA = np.sin(angle / 2)
        cosA = np.cos(angle / 2)
        qX = axeX * sinA
        qY = axeY * sinA
        qZ = axeZ * sinA
        qW = cosA
        normQ = np.sqrt(qW*qW + qX*qX + qY*qY + qZ*qZ)
        qX = qX/normQ
        qY = qY/normQ
        qZ = qZ/normQ
        qW = qW/normQ
        return [qX,qY,qZ,qW]

# Fonction qui convertit un Quad en matrice de rotation.
def QuadToMatRot(q):
        qX = q[0]
        qY = q[1]
        qZ = q[2]
```

```
        qW = q[3]
        xx = qX * qX
        xy = qX * qY
        xz = qX * qZ
        xw = qX * qW
        yy = qY * qY
        yz = qY * qZ
        yw = qY * qW
        zz = qZ * qZ
        zw = qZ * qW
        mat = np.zeros((4,4),dtype='d')
        mat[0][0] = 1 - 2 * ( yy + zz )
        mat[0][1] =     2 * ( xy - zw )
        mat[0][2] =     2 * ( xz + yw )
        mat[1][0] =     2 * ( xy + zw )
        mat[1][1] = 1 - 2 * ( xx + zz )
        mat[1][2] =     2 * ( yz - xw )
        mat[2][0] =     2 * ( xz - yw )
        mat[2][1] =     2 * ( yz + xw )
        mat[2][2] = 1 - 2 * ( xx + yy )
        mat[3][3] = 1
        return mat

# Fonction qui convertit une matrice de rotation en angle d'euler.
def matRotToEuler(angle1,angle2):
        mat1=QuadToMatRot(axeRotToQuad(0,0,1,angle1))
        mat2=QuadToMatRot(axeRotToQuad(-np.sin(angle1),np.cos(angle1),0,angle2))
        mat=np.dot(mat1,mat2)
        angleY = -np.arcsin(mat[0][2])
        C = np.cos(angleY)
        angleY = np.degrees(angleY)
        if (abs(C) > 0.005): # Gimbal lock ?
                TRx    =  mat[2][2] / C
                TRy    = -mat[1][2] / C
                angleX = np.degrees(np.arctan2(TRy, TRx))
                TRx    =  mat[0][0] / C
                TRy    = -mat[0][1] / C
                angleZ = np.degrees(np.arctan2(TRy, TRx))
        else: # Gimbal lock
                angleX = 0
                TRx    = mat[1][1]
                TRy    = mat[1][0]
                angleZ = np.degrees(np.arctan2(TRy, TRx))
        return [angleX,angleY,angleZ]
```

```
# Fonction pour tester l'algorithme.
def fonctionTest(X):
        angleX = X[0]
        angleY = X[1]
        angleZ = X[2]
        angleA = X[3]
        f_resultat = open(os.path.dirname(os.path.abspath(__file__))+"/angle.csv
        f_resultat.write("\n")
        f_resultat.write("{0};{1};{2};{3};;{4};".format(angleX,angleY,angleZ,ang
        f_resultat.close()
        return X[0]*X[0] + X[1]*X[1] + X[2]*X[2] + 10*X[0] + 1


# Cette fonction cherche le minimume du critere evaluant la qualite de la retro-
#
# Arguments :
#  espaceRecherche : Definit le nombre d'evaluations de la fonction.
#  nbNivRafinement : Definit le niveau de rafinement.
#  nbRetenuParRafinement: Definit le nombre de X qui sera retenu a chaque iterat
#  fonctionScript : Definit la fonction a minimiser.
# Par exemple :
#  espaceRecherche = [[0,360,7],[90,270,3]]
#  nbNivRafinement = 3
#  nbRetenuParRafinement = 4
#  fonctionScript = fonctionTest
#  Le script va evaluer la fonction 7 dans l'interval [0,360] 3 fois dans l'inte
#  La fonction va etre evlue au point :
#  45,135  45,180  45,225
#  90,135  90,180  90,225
#  135,135 135,180 135,225
#  180,135 180,180 180,225
#  225,135 225,180 225,225
#  270,135 270,180 270,225
#  305,135 305,180 305,225
#  nbRetenuParRafinement vaut 4 danc le scrit memorisera les 4 meilleurs resulta
#  Pour chaque resultats, il recomencera la demarche ci dessus en divisant la ta
# nbNivRafinement vaut 3 donc le scripte vas s'arreter au bout de 3 rafinements
def recherOptimamum(espaceRecherche,nbNivRafinement,nbRetenuParRafinement,fonctio

        nbEval = functools.reduce(lambda x,y: x*y,[elt[2] for elt in espaceReche
        print("")
        print("nombre d'evaluation de fonction : {0}".format(nbEval + (nbNivRafi
        print("espacement entre deux evaluation pour dernier niveau de raffinage
        print("")

        meillieurResulX = [] # Contient les elements de l'espace de recherche ay
```

```
meillieurResulY = [] # Contient leur evaluation corespandante.

for espRech in espaceRecherche:
        espRech.append((espRech[1]-espRech[0])/espRech[2]) # Corespond a
        espRech.append((espRech[1]-espRech[0])/2) # Corespond a l'espace

for nivRafinement in range(1,nbNivRafinement+1):
        print("//////////////␣nieau␣de␣rafinage␣:␣{0}␣//////////////".for
        # On met a jour les variales : le pas et la demi taille de l'esp
        for espRech in espaceRecherche:
                espRech[3] = (espRech[1]-espRech[0])/((espRech[2]+1)**ni
                espRech[4] = (espRech[1]-espRech[0])/(2*(espRech[2]+1)**
        # On specifier les elements sur lesquelle l'espace de rechreche
        if nivRafinement == 1:
                XRechercheInit = [[(elt[0]+elt[1])/2 for elt in espaceRe
        else:
                XRechercheInit = copy.deepcopy(meillieurResulX)

        for XInit in XRechercheInit:
                print("————————————————X␣retenu␣:␣{0}————————————————
                X = [var-espaceRecherche[i][4]+espaceRecherche[i][3] for

                # Boucle principal.
                fini = False
                while not fini:

                        # Evalution de la fonction pour le X courant.
                        Y = fonctionScript(X)

                        # On regarde si on a un meillieur resultat.
                        if len(meillieurResulX) < nbRetenuParRafinement:
                                meillieurResulX.append(list(X))
                                meillieurResulY.append(Y)
                        finiMR=False
                        while not finiMR:
                                Xtmp = list(X)
                                tmp = replaceParMeillieur(meillieurResulY
                                if tmp==[]:
                                        finiMR=True
                                else:
                                        Y=tmp[0]
                                        Xtmp=tmp[1]

                        # On calcule le nouveau X.
                        incrFini = False
                        i = len(X)-1
```

```
                                        while not incrFini and i >= 0:
                                                if X[i] >= XInit[i]+espaceRecherche[i][4]
                                                        X[i] = XInit[i]-espaceRecherche[
                                                        incrFini = False
                                                else:
                                                        X[i] = X[i] + espaceRecherche[i]
                                                        incrFini = True
                                                i = i-1
                                        fini = not incrFini
                                        # Fin du while.

        print("")
        print("resultat : ")
        for i,elt in enumerate(meillieurResulX):
                print("X={0} avec Y={1}".format(meillieurResulX[i],meillieurResul

        k=1
        meillval = meillieurResulY[0]
        meillk=0
        while k<len(meillieurResulY):
                if meillieurResulY[k] < meillval:
                        meillk = k
                k = k+1
        return meillieurResulX[meillk]


##########################################################################################
# debut du script
##########################################################################################

YoungExt = 5
YoungOs = 150
AngleUn = 0
AngleDeux = 0
Force = 200

def fonctionAngle(X):
        return fonctionGen([X[0],X[1],YoungExt,YoungOs,0,0,Force])
def fonctionForce(X):
        return fonctionGen([AngleUn,AngleDeux,YoungExt,YoungOs,0,0,X[0]])
def fonctionAngleForce(X):
        return fonctionGen([X[0],X[1],YoungExt,YoungOs,0,0,X[2]])
def fonctionAngleForce(X):
        return fonctionGen([X[0],X[1],YoungExt,YoungOs,0,0,X[2]])
def fonctionYoung(X):
        return fonctionGen([AngleUn,AngleDeux,X[0],X[1],0,0,Force])
```

```
for i in range(6):
        resul = recherOptimamum([[-90,90,5],[-90,90,5]],3,1,fonctionAngle)
        time.sleep(60)
        os.system("pkill -SIGKILL runSofa")
        time.sleep(60)
        AngleUn = resul[0]
        AngleDeux = resul[1]

        resul = recherOptimamum([[0,400,7]],3,1,fonctionForce)
        time.sleep(60)
        os.system("pkill -SIGKILL runSofa")
        time.sleep(60)
        Force = resul[0]

        resul = recherOptimamum([[-55,305,5],[-55,305,5]],3,1,fonctionYoung)
        time.sleep(60)
        os.system("pkill -SIGKILL runSofa")
        time.sleep(60)
        YoungExt = resul[0]
        YoungOs = resul[1]

print("#############################################################################################
print("resultat")
print("angle : {0},{1} \nraideur : {2},{3} \nforce {4}".format(AngleUn,AngleDeux
```