HMIN318M: Imagerie médicale et 3D

Rendu TP : Recalage

Tianning MA

M2 IMAGINA

16/12/2020

Table de matière

A. Introduction 2
B. Compilation 2
C. Rendu et explication des exercices 2
1. Visualisation des données 3
2. Calcul des profils d'intensité 3
3. Calcul des profils de similarité 4
SSD 4
NCC 4
4. Calcul des correspondances 6
5. Tests 6
Annexe 9
1. Images des profils 9
2. Images des distances 10
3. Code sources pour les fonctions demandées 11

A. Introduction

Ce compte rendu est dédié au TP Recalage du cours d'Imagerie médicale 3D (HMIN318M).

Toutes les questions sont répondues. Vous trouverez le code source et les images demandés dans la partie Annexe de ce rapport.

Vous pouvez aussi trouver l'intégralité de ce tp dans mon espace git avec l'adresse ci-dessous :

https://github.com/matianning/ImagerieMedicale3D

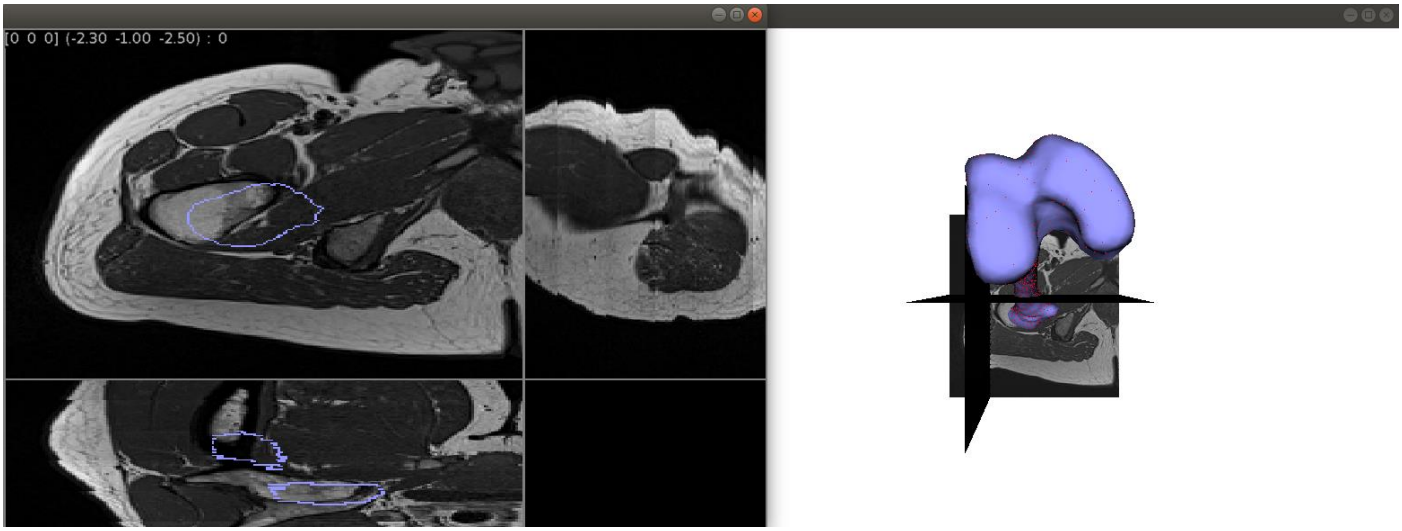
B. Compilation

Pour compiler sous linux : g++ -o main.exe main.cpp -O2 -L/usr/X11R6/lib -lm -lpthread -lX11

Exécution : ./main.exe

C. Rendu et explication des exercices

1. Visualisation des données



2. Calcul des profils d'intensité

```
CImg<unsigned char> computeProfiles(const MESH& mesh,
                                    const IMG<unsigned char,float>& img,
                                    const unsigned int Ni,
                                    const unsigned int No,
                                    const float l,
                                    const unsigned int interpolationType=1)
{
    CImg<unsigned char> prof(Ni + No, mesh.getNbPoints());


    for(unsigned int i = 0; i < mesh.getNbPoints(); i++){

        float p[3]; mesh.getPoint(p, i);
        float n[3]; mesh.getNormal(n, i);

        //******À l'intérieur du modèle******
        for(float j = 0; j < Ni; j++){
            float tmp[3];
            tmp[0] = p[0] - n[0] * j * l;
            tmp[1] = p[1] - n[1] * j * l;
            tmp[2] = p[2] - n[2] * j * l;
            unsigned char val = img.getValue(tmp,interpolationType);
            prof(Ni - j - 1, i ) = val;
        }

        //******À l'extérieur du modèle*****
        for(float k = 0; k < No; k++){
            float tmp[3];
            tmp[0] = p[0] + n[0] * k * l;
            tmp[1] = p[1] + n[1] * k * l;
            tmp[2] = p[2] + n[2] * k * l;
            unsigned char val = img.getValue(tmp, interpolationType);
            prof(Ni + k - 1, i ) = val;
        }
    }

    prof.display();

    return prof;

}
```

Pour calculer des profils d'intensité :

Sachant que l'image des profils est de taille N * (Ni + No), donc pour chaque point du maillage, on a besoins de chercher la valeur correspondante dans l'image puis ajouter cette valeur dans l'image des profils.

Puisque l'image est échantillonnée le long de la normale, on a besoin de récupérer la normale du point pour calculer les échantillons et chercher ses valeurs dans l'image.

Le raisonnement serait parail pour l'image des profils cibles. Par contre, la taille des profils cibles sera agrandie de S (la distance de recherche).

Après mon implémentation, j'obtiens le résultat comme l'image (extrait de l'image résultante) ci-dessous.

Sur la partie gauche de l'image, on voit les intensités qui se trouve à l'intérieur de maillage, et celle de droite, les intensités qui se trouve à l'extérieur de maillage.

Vous trouverez dans la partie *Annexe* l'image des profils complète.

3. Calcul des profils de similarité

SSD

La méthode SSD (la somme des distances au carré) est une des mesures de similarité.

Voici mon implémentation :

```
if(metric==SSD)
{
    for(unsigned int i = 0; i < nbpoints; i++){

        for(unsigned int j = 0; j < 2 * S; j++){
            float currentSSD = 0.0;

            for(unsigned int k = 0; k < N; k++){
                float distance = sourceProf(k, i) - targetProf(j+k, i);
                float squaredDist = distance * distance;
                currentSSD += squaredDist;
            }

            dist(j,i) = currentSSD;
        }
    }
```

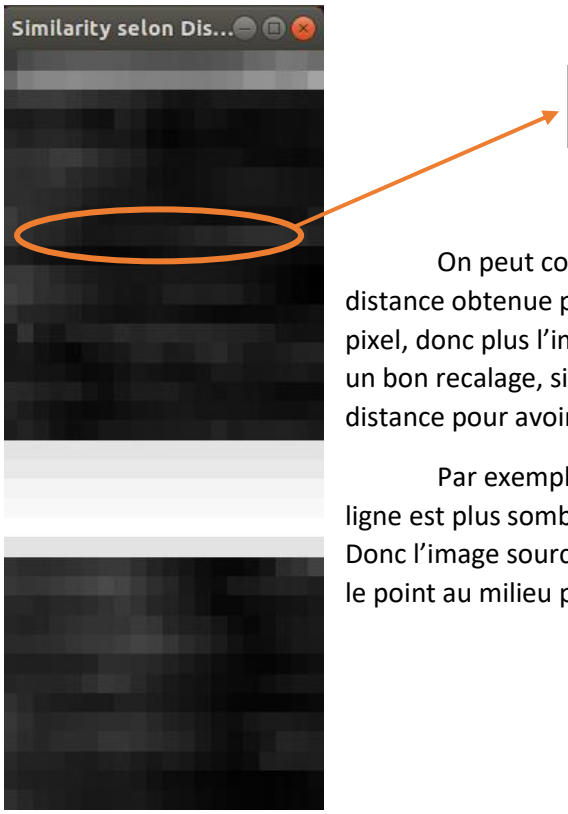


Figure 1 Extrait de l'image distance (SSD)

On peut constater que sur l'image ci-contre (un extrait de l'image des distance obtenue par la méthode SSD), plus la distance est grande, plus blanc sur la pixel, donc plus l'image de source et l'image cible sont différente. Donc, pour avoir un bon recalage, si on utilise la méthode SSD, on aura besoin de minimiser cette distance pour avoir un bon résultat.

Par exemple sur cette ligne (l'image ci-dessus), on a le pixel au milieu de la ligne est plus sombre (proche de noir), ça veut dire que la distance est plus petite. Donc l'image source et l'image cible sont moins différente. Ce serait mieux de choisir le point au milieu pour (calculer la tranformation puis) le recalage.

NCC

La méthode NCC (Corrélation croisée normalisée) consiste à calculer la covariance et la corrélation entre deux images afin de mesurer la similarité entre eux.

Au contraire de SSD, pour que l'image de source et l'image de cible soient moins différente, il faut maximiser cette valeur. Donc choisir le point dont la valeur de NCC est plus élever pour faire le recalage.

```
else // NCC
{
    /***** Calculer la moyenne ******/
    float moyenne_source = 0.0, moyenne_cible = 0.0;
    for(unsigned int i = 0; i < nbpoints; i++){
        for(unsigned int j = 0; j < N; j++){
            moyenne_source += sourceProf(j,i);
        }
    }
    moyenne_source = moyenne_source / (N * nbpoints);
    for(unsigned int i = 0; i < nbpoints; i++){
        for(unsigned int j = 0; j < N + 2 * S; j++){
            moyenne_cible += targetProf(j,i);
        }
    }
    moyenne_cible = moyenne_cible / ((N + 2 * S) * nbpoints);

    /***** Calculer la covariance et la normaliser *****/
    for(unsigned int i = 0; i < nbpoints; i++){

        for(unsigned int j = 0; j < 2 * S; j++){
            float sommeCovariance = 0.0;
            float covarianceSource = 0.0;
            float covarianceCible = 0.0;

            for(unsigned int k = 0; k < N; k++){
                float covariance = (sourceProf(k, i) - moyenne_source) * (targetProf(j+k, i) - moyenne_cible);
                float covariance_I  = (sourceProf(k, i) - moyenne_source) * (sourceProf(k, i) - moyenne_source);
                float covariance_J = (targetProf(j+k, i) - moyenne_cible) * (targetProf(j+k, i) - moyenne_cible);
                sommeCovariance += covariance;
                covarianceSource += covariance_I;
                covarianceCible += covariance_J;
            }

           dist(j,i) = sommeCovariance / (covarianceSource * covarianceCible);
        }
    }

}
```

Comme vu en cours, les formules pour calculer NCC :

$$\frac{\sum_{x,y}(I(x,y) - \bar{I})^2 * (J(x,y) - \bar{J})}{\sum_{x,y}(I(x,y) - \bar{I})^2 * \sum_{x,y}(J(x,y) - \bar{J})^2}$$

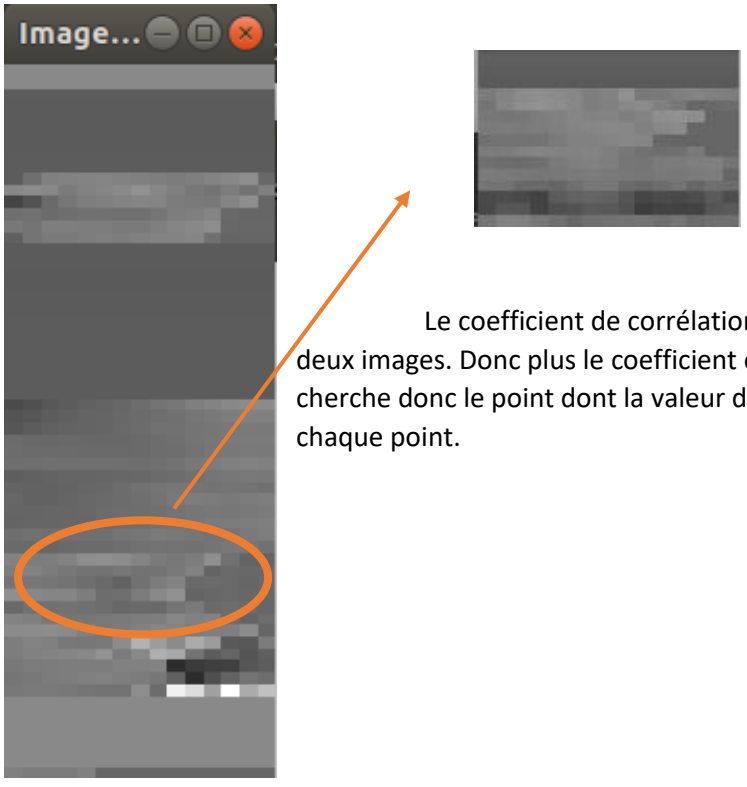


Figure 2 Extrait de l'image des distances (NCC)

Le coefficient de corrélation permet d'étudier l'intensité de liaison entre deux images. Donc plus le coefficient est élevé, plus les deux images sont liées. On cherche donc le point dont la valeur de NCC est plus élevé (plus proche de blanc) pour chaque point.

4. Calcul des correspondances

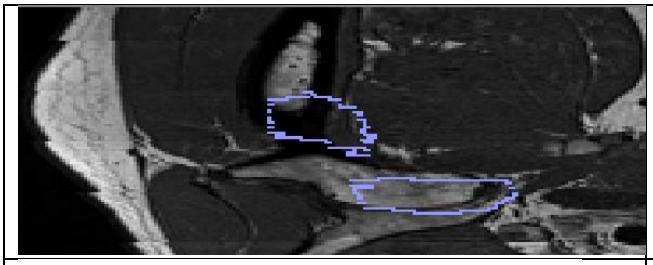
Afin de calculer des correspondances des points, on va se servir de l'image des distances. En effet, en cherchant la distance minimale dans l'image des distances, cela nous permet d'avoir à quel point on va décaler le point d'original. Ensuite, selon le décalage, on va simplement mettre à jours ce point selon le décalage et la direction de sa normale (Pc = Pi + l * Ni). Mon implémentation est comme ci-dessous.

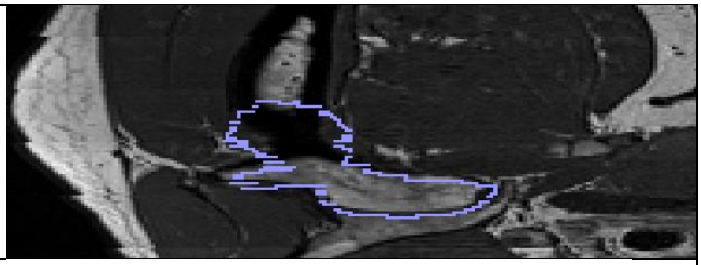
```
void computeCorrespondences(MESH& mesh, const CImg<float> &dist,const float l)
{
    unsigned int nbpoints=dist.height();
    unsigned int S=dist.width()/2;


    for(unsigned int i=0;i<nbpoints;i++)
    {
        //******* Chercher la distance minimale *******
        float min_dist = 255.0;
        unsigned int index = 0;
        for(unsigned int j = 0; j < S * 2; j++){
            if(dist(j,i) <= min_dist){
                min_dist = dist(j,i);
                index = j;
            }
        }

        //****** Mettre en correspondance ********
        float p[3]; mesh.getPoint(p,i);
        float n[3]; mesh.getNormal(n,i);
        float decalage = index - S;
        p[0] = p[0] + n[0] * decalage * l;
        p[1] = p[1] + n[1] * decalage * l;
        p[2] = p[2] + n[2] * decalage * l;

        mesh.setCorrespondence(p,i);
    }

}
```

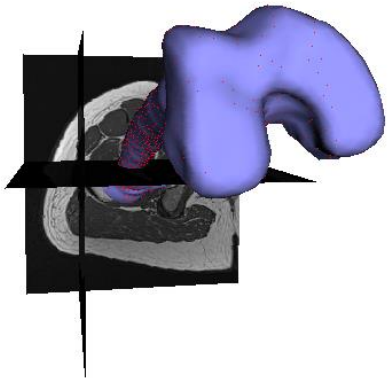
5. Tests

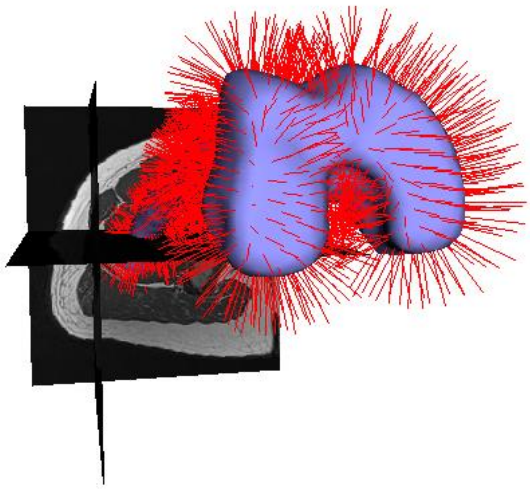
Vous trouverez les images obtenues (pour les profils et les distances) dans la partie d'Annexe.

Avant recalage	Après recalage
	





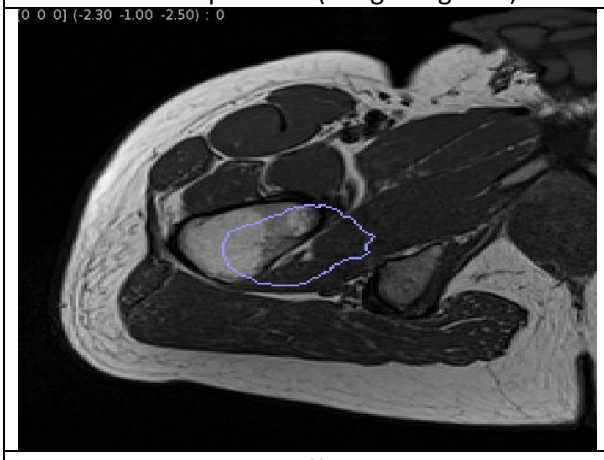
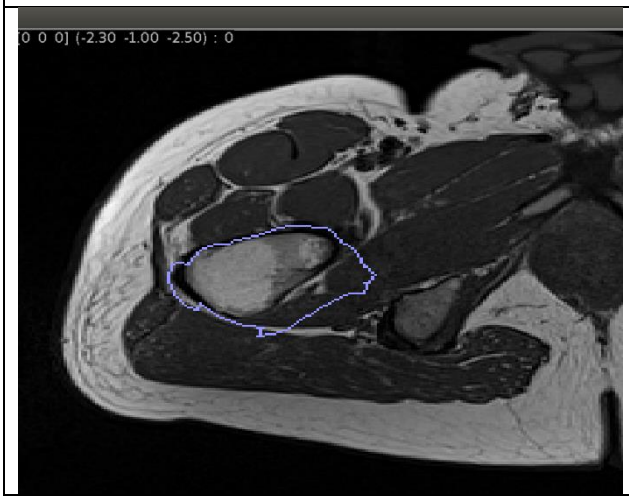


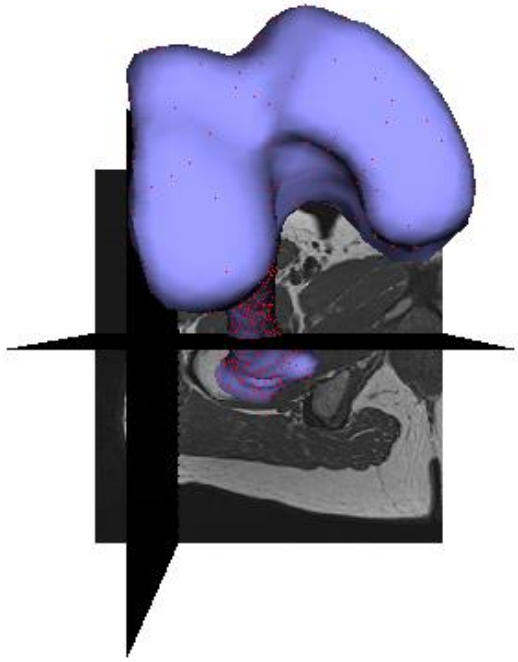
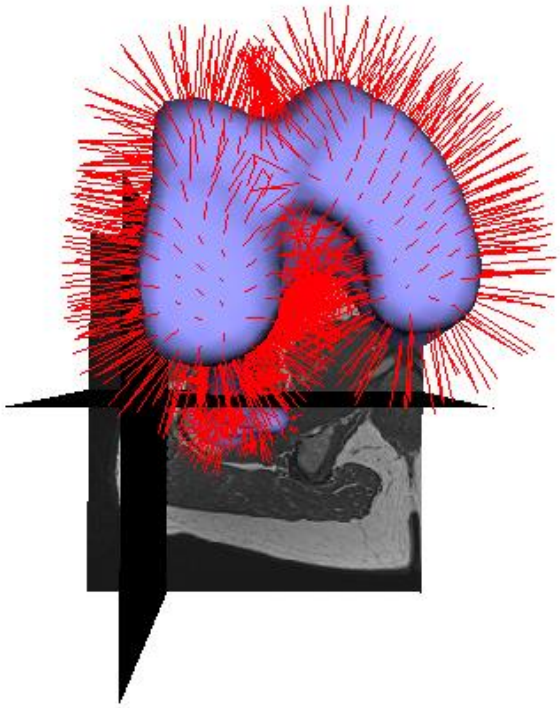
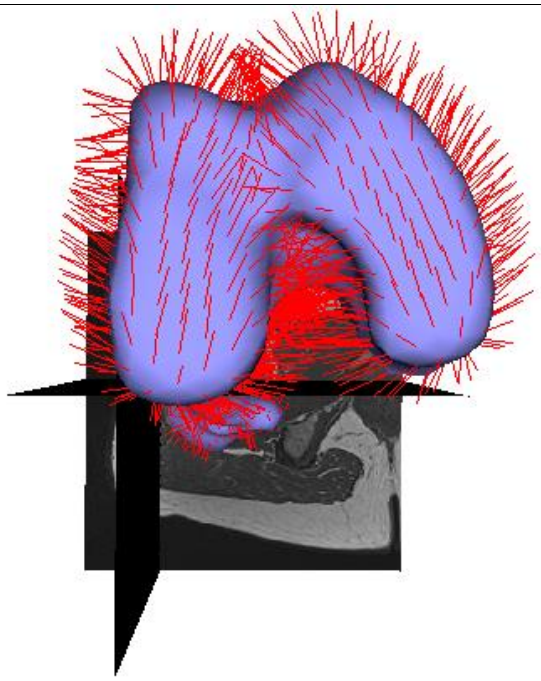
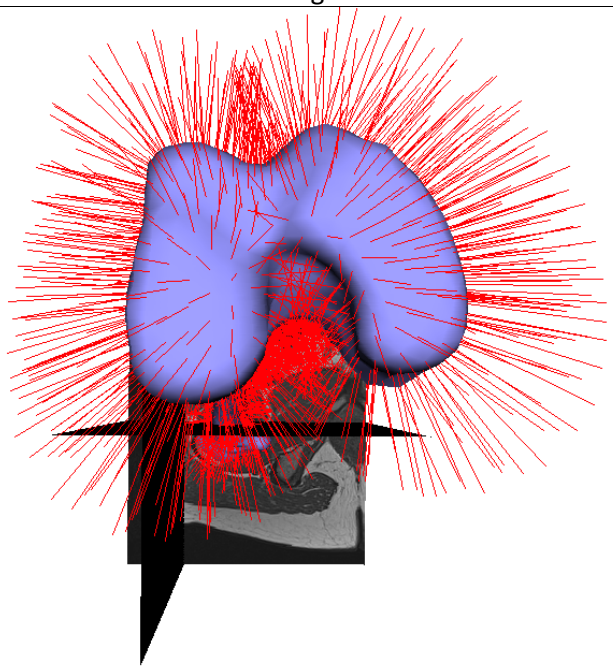


Obeservation : On observe que surtout sur l'image principal (les premieres paires d'image), le maillage se rapproche de vraie position de la partie voulue après l'opération de recalage. Donc la position des points du maillage se rapproche aussi la position souhaitée.

Réflexion : la correspondance des points (pour le moment) est basé sur les valeurs de la première image (quand on fait getValue pour récupérer les valeurs des voxels) donc, serait-il mieux de prendre en compte les trois images, pour que la position finale du maillage se rapproche de vraie position.

Testes sur les différents types de transformation

Sans opération (Image originale)	Similitude
	
Affine	**Rigid**
	

Sans opération (Image originale)	Similitude
	
Affine	Rigid
	

Annexe

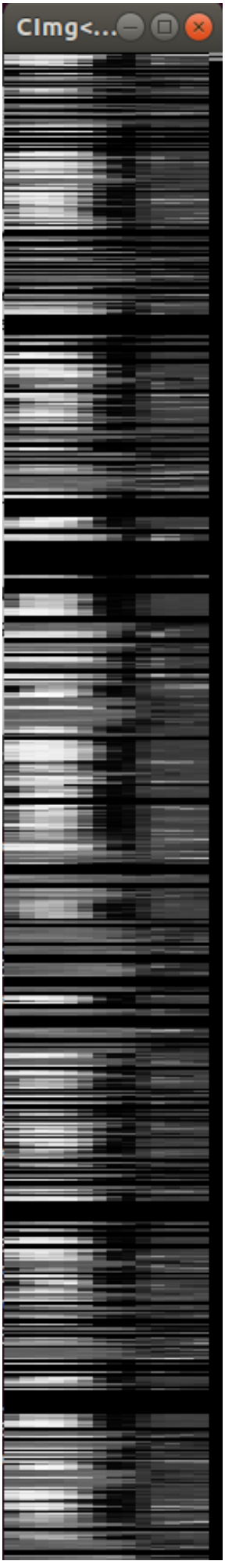
1. Images des profils

Vous trouvez les images des profils ci-contre.

L'image de gauche correspond à l'image des profils de source (taille = N * (Ni + No))

et celle de droite correspond à l'image des profils de cible (taille = N * (Ni + No + 2S)).

S : distance de recherche

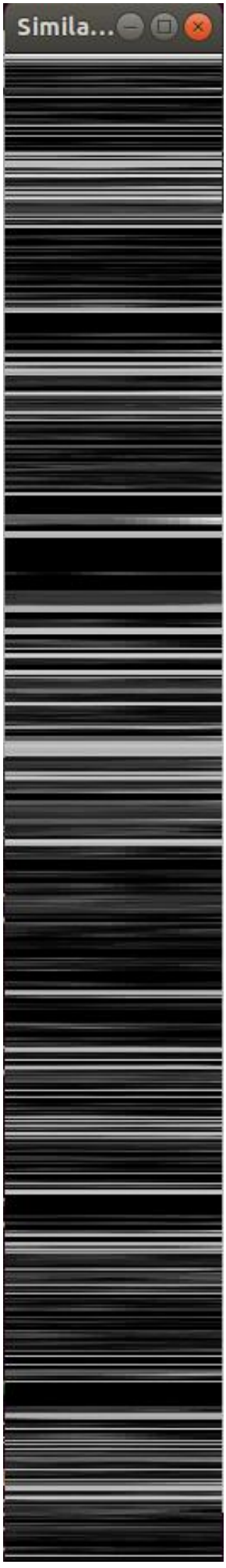


2. Images des distances

Vous trouverez ci-contre les images des distances :

L’image des distances par la méthode SSD (Image gauche)

L’image des distances par la méthode NCC (Image droite)



3. Code sources pour les fonctions demandées

```
void computeCorrespondences(MESH& mesh, const CImg<float> &dist,const float l)
{
   unsigned int nbpoints=dist.height();
   unsigned int S=dist.width()/2;

   for(unsigned int i=0;i<nbpoints;i++)
   {
      //******* Chercher la distance minimale *******
      float min_dist = 100000000.0;
      int index = 0;
      for(unsigned int j = 0; j < S * 2; j++){
         if(j!=S){
            if(dist(j,i) <= min_dist){
               min_dist = dist(j,i);
               index = j;
            }
         }
      }

      //****** Mettre en correspondance *********
      float p[3]; mesh.getPoint(p,i);
      float n[3]; mesh.getNormal(n,i);
      int decalage = index - S;
      p[0] = p[0] + n[0] * decalage * l;
      p[1] = p[1] + n[1] * decalage * l;
      p[2] = p[2] + n[2] * decalage * l;

      mesh.setCorrespondence(p,i);
   }
}
```

```
CImg<float> computeDistance(const CImg<unsigned char> &sourceProf,
                            const CImg<unsigned char> &targetProf,
                            const Metric metric)
{
   unsigned int nbpoints=sourceProf.height();
   unsigned int N=sourceProf.width();
   unsigned int S=(targetProf.width()-sourceProf.width())/2;

   CImg<float> dist(2*S,nbpoints);

   if(metric==SSD)
   {
      for(unsigned int i = 0; i < nbpoints; i++){
         for(unsigned int j = 0; j < 2 * S; j++){
            float currentSSD = 0.0;
            for(unsigned int k = 0; k < N; k++){
               float distance = sourceProf(k, i) - targetProf(j+k, i);
               float squaredDist = distance * distance;
               currentSSD += squaredDist;
            }
           dist(j,i) = currentSSD;
```

```
        }
      }
    }
    else // NCC
    {
      /***** Calculer la moyenne ******/
      float moyenne_source = 0.0, moyenne_cible = 0.0;
      for(unsigned int i = 0; i < nbpoints; i++){
        for(unsigned int j = 0; j < N; j++){
          moyenne_source += sourceProf(j,i);
        }
      }
      moyenne_source = moyenne_source / (N * nbpoints);
      for(unsigned int i = 0; i < nbpoints; i++){
        for(unsigned int j = 0; j < N + 2 * S; j++){
          moyenne_cible += targetProf(j,i);
        }
      }
      moyenne_cible = moyenne_cible / ((N + 2 * S) * nbpoints);

      /***** Calculer la covariance et la normaliser *****/
      for(unsigned int i = 0; i < nbpoints; i++){

        for(unsigned int j = 0; j < 2 * S; j++){
          float sommeCovariance = 0.0;
          float covarianceSource = 0.0;
          float covarianceCible = 0.0;

          for(unsigned int k = 0; k < N; k++){
            float covariance = (sourceProf(k, i) - moyenne_source) * (targetProf(j+k, i) - moyenne_cible);
            float covariance_I  = (sourceProf(k, i) - moyenne_source) * (sourceProf(k, i) - moyenne_source);
            float covariance_J = (targetProf(j+k, i) - moyenne_cible) * (targetProf(j+k, i) - moyenne_cible);
            sommeCovariance += covariance;
            covarianceSource += covariance_I;
            covarianceCible += covariance_J;
          }
          dist(j,i) = sommeCovariance / (covarianceSource * covarianceCible);
        }
      }

    }

    dist.display("Image des distances");
    return dist;
}



CImg<unsigned char> computeProfiles(const MESH& mesh,
                                    const IMG<unsigned char,float>& img,
                                    const unsigned int Ni,
                                    const unsigned int No,
                                    const float l,
                                    const unsigned int interpolationType=1)
{
    CImg<unsigned char> prof(Ni + No, mesh.getNbPoints());
```

```
  for(unsigned int i = 0; i < mesh.getNbPoints(); i++){

    float p[3]; mesh.getPoint(p, i);
    float n[3]; mesh.getNormal(n, i);

    //******À l'intérieur du modèle******
    for(float j = 0; j < Ni; j++){
      float tmp[3];
      tmp[0] = p[0] - n[0] * j * l;
      tmp[1] = p[1] - n[1] * j * l;
      tmp[2] = p[2] - n[2] * j * l;
      unsigned char val = img.getValue(tmp,interpolationType);
      prof(Ni - j - 1, i ) = val;
    }

    //******À l'extérieur du modèle*****
    for(float k = 0; k < No; k++){
      float tmp[3];
      tmp[0] = p[0] + n[0] * k * l;
      tmp[1] = p[1] + n[1] * k * l;
      tmp[2] = p[2] + n[2] * k * l;
      unsigned char val = img.getValue(tmp, interpolationType);
      prof(Ni + k - 1, i ) = val;
    }
  }
  prof.display();
  return prof;
}
```