

TP3 - Reconstruction de fresque

Traitement d'images

Julien Save

Alexis Da Costa

2021-10-30

Contents

Introduction	3
I. Associations des points d'intérêt et RANSAC	4
II. Filtrage des associations par RANSAC	8
III. Filtre des associations par le Lowe Ratio Tests	11
IV. Calcul des coordonnées du fragment	13
V. Construction de la fresque	15

Introduction

L'objectif du TP est de faire une reconstruction de fresque à partir des fragments, mais sans utiliser cette fois la liste de positions des fragments. Dans ce cas, l'objectif est donc de créer vous même le fichier de solutions, qui contient pour chaque fragment considéré comme faisant partie de la fresque une ligne du type : <index> <posx> <posy> <angle>

Pour cela, nous allons suivre le protocole suivant en 6 étapes :

1. **Détection des points caractéristiques - Feature Detection**
2. **Description des points caractéristiques - Feature Description**
3. **Correspondance des points caractéristiques - Feature Matching**
4. **Filtrage des associations - Matches Filtering.** Plusieurs méthodes possibles :
 - Par l'utilisation d'un RANSAC pour identifier et rejeter les outliers (les associations avec des valeurs abérantes)
 - Par les propriétés de la distance euclidienne entre deux points d'intérêts lors d'une association
5. **Calcul de la position des coordonnées à partir des associations filtrées - Coordinates position calculation from the good matches**
6. **Reconstruction de l'image - Image reconstruction**

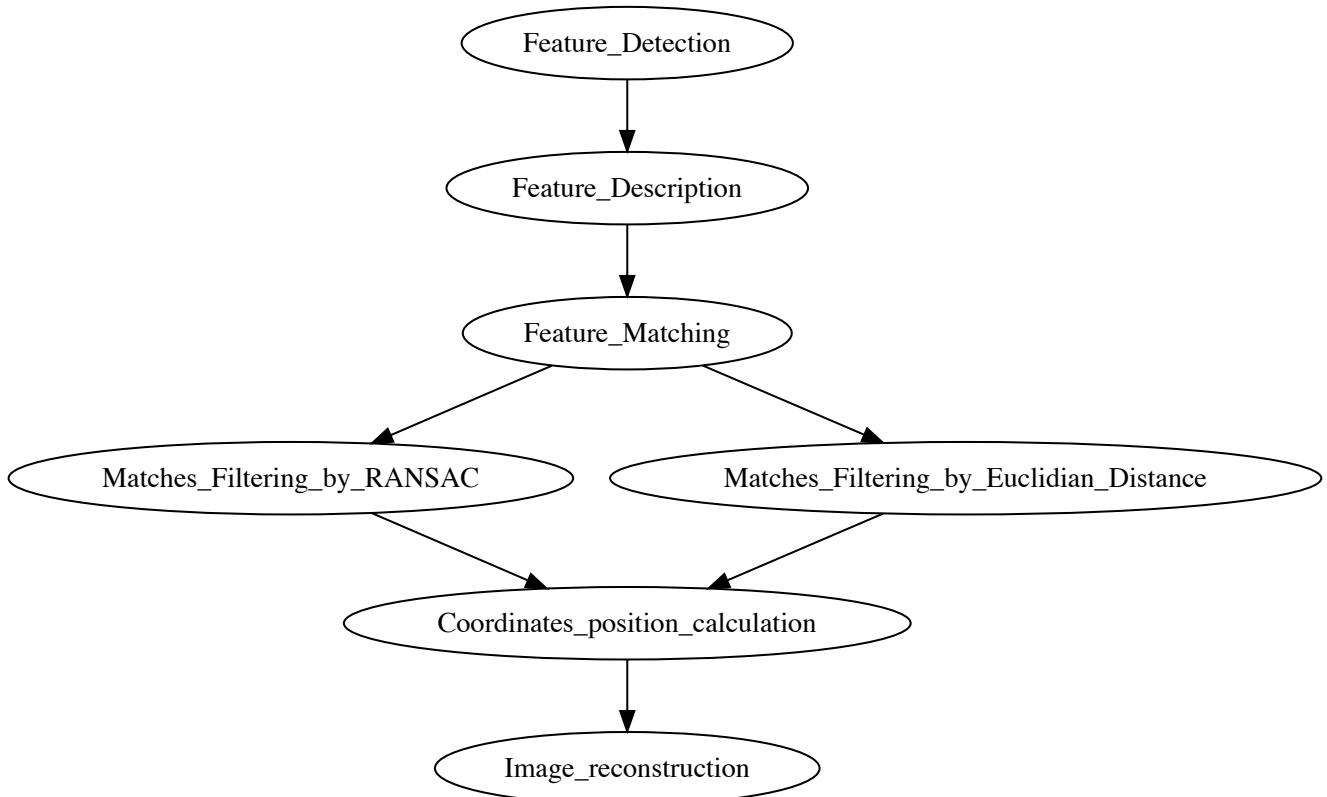


Figure 1: Schéma du protocole expérimental pour la reconstruction de la fresque

I. Associations des points d'intérêt et RANSAC

Dans ce premier exercice, nous allons essayer d'associer plusieurs points d'intérêt entre un fragment et la la fresque. Le but est ainsi d'essayer de localiser la position du fragment dans la fresque. Pour cela nous devons utilisé un algorithme utilisant un détecteur ainsi qu'un descripteur. OpenCV en propose plusieurs, chacun ayant des caractéristiques différents.

Présentons rapidement et brièvement l'ensemble des algorithmes de détecteurs et de descripteurs de points d'intérêts présent dans OpenCV :

ORB : est un détecteur rapide et robuste de caractéristiques locales, crée par Ethan Rublee en 2011, qui peut être utilisé dans des tâches de traitement d'images comme pour la reconnaissance d'objets ou la reconstruction 3D. Il est basé sur le détecteur de points d'intérêts FAST et une version modifiée du descripteur visuel BRIEF (Binary Robust Independent Elementary Features). Son objectif est de fournir une alternative rapide et efficace à SIFT.

SIFT : est un algorithme de traitement d'images permettant de détecter, de décrire et de faire correspondre des caractéristiques locales dans des images, inventé par David Lowe en 1999. Les applications comprennent la reconnaissance d'objets, la cartographie et la navigation robotiques, l'assemblage d'images, la modélisation 3D, la reconnaissance de gestes, le suivi vidéo, l'identification individuelle d'animaux sauvages et le déplacement de correspondances.

SURF : est un détecteur et descripteur de caractéristiques locales breveté et donc payant sous licence. Il peut être utilisé pour des tâches telles que la reconnaissance d'objets, le recalage d'images, la classification ou la reconstruction 3D. Il est partiellement inspiré du descripteur SIFT (scale-invariant feature transform). La version standard de SURF est plusieurs fois plus rapide que SIFT et ses auteurs affirment qu'elle est plus robuste que SIFT contre différentes transformations d'images.

Pour détecter les points d'intérêt, SURF utilise une approximation entière du déterminant du détecteur de blob de Hessian, qui peut être calculé avec 3 opérations entières en utilisant une image intégrale précalculée. Son descripteur de caractéristiques est basé sur la somme de la réponse en ondelettes de Haar autour du point d'intérêt. Ceux-ci peuvent également être calculés à l'aide de l'image intégrale.

FAST : est une méthode de détection des coins, qui est utilisée pour extraire des points caractéristiques et, par la suite, pour suivre et cartographier des objets dans de nombreuses tâches de traitement d'images. Le détecteur de coins FAST a été développé à l'origine par Edward Rosten et Tom Drummond, et a été publié en 2006.

L'avantage le plus prometteur du détecteur de coins FAST est son efficacité de calcul. Comme son nom l'indique, il est en effet plus rapide que de nombreuses autres méthodes d'extraction de caractéristiques bien connues, telles que la différence de gaussiennes (DoG) utilisée par les détecteurs SIFT, SUSAN et Harris.

Ici, dans notre cas, nous avons opté pour l'utilisation de l'algorithme SIFT de par sa gratuité, sa simplicité et efficacité.

Voici, le détail de l'implémentation¹ ci-dessus permettant d'établir les associations entre un fragment et la fresque initiale.

Etape 1 - Utiliser un détecteur de point d'intérêt.

OpenCV en propose plusieurs. *Exemple : SIFT, SURF, FAST, ORB, Harris.* Voici un exemple avec le détecteur SIFT :

```
def detect_interest_point_using_SIFT(img_fragment, img_fresco):
    -- Detect keypoint using SIFT detector
    detector = cv.SIFT.create()
    keypoints_fragment, descriptors_fragment = detector.detectAndCompute(img_fragment, None)
    keypoints_fresco, descriptors_fresco = detector.detectAndCompute(img_fresco, None)

    return keypoints_fragment, descriptors_fragment, keypoints_fresco, descriptors_fresco
```

Etape 2 - Associer les coins correspondants en s'appuyant sur la similarité des descripteurs visuels calculés dans les coins respectifs.

Pour cela, nous devons utiliser un descripteur, qui va nous permettre d'obtenir l'ensemble des "matches", des associations de points d'intérêts entre la fresque et le fragment sélectionné. Voici son implémentation :

```
def match_interest_point_using_SIFT(descriptors_fragment, descriptors_fresco):
    -- Match descriptor vectors
    matcher = cv.DescriptorMatcher_create(cv.DescriptorMatcher_FLANNBASED)
    all_matches = matcher.knnMatch(descriptors_fragment, descriptors_fresco, 2)

    return all_matches

def draw_matches(img_fragment, img_fresco, keypoints_fragment, keypoints_fresco, matches):
    -- Draw matches and display it
    img_matches = np.empty((max(img_fragment.shape[0], img_fresco.shape[0]), img_fragment.shape[1]+img_fresco.shape[1]), np.uint8)
    cv.drawMatches(img_fragment, keypoints_fragment, img_fresco, keypoints_fresco, matches, img_matches, flags=2)

    return img_matches
```

Voici le résultat final que nous obtenons pour le fragment 92 :

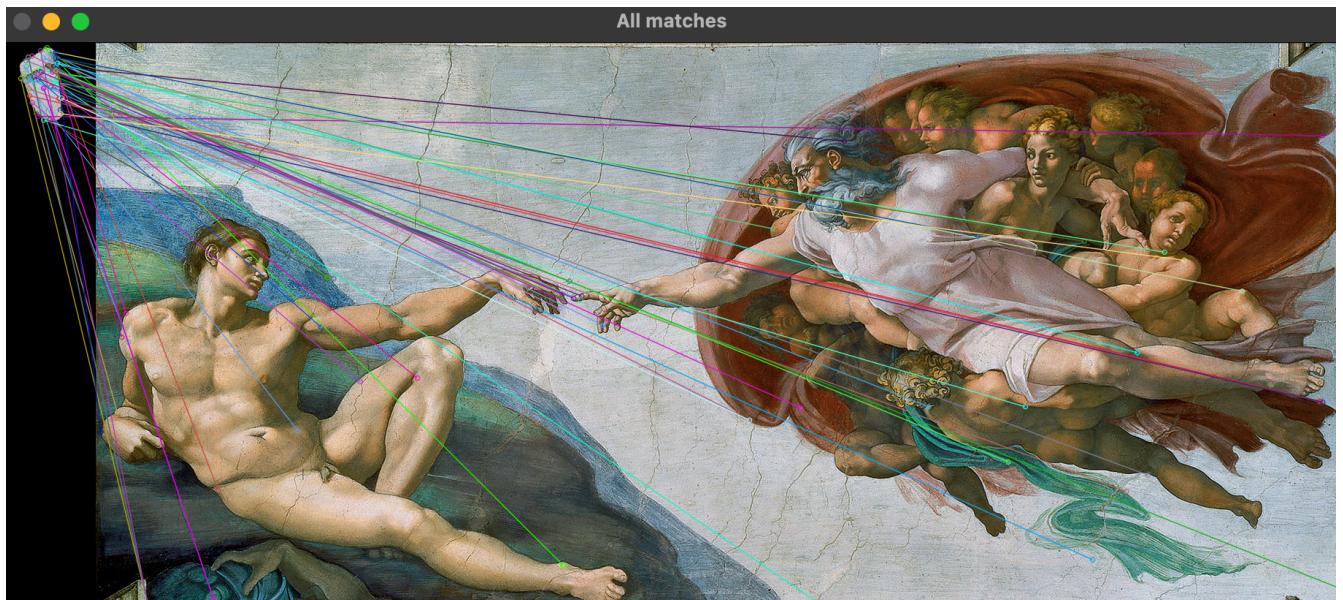


Figure 2: Associations entre le fragment 92 et la fresque

¹L'ensembles des implémentations dans ce TP seront fait en python.

Cependant, nous pouvons constater que nous avons un grand nombre d'association sur cette image. Seule une (très) petite partie de ces associations est correcte.

C'est pourquoi, par la suite on s'appuiera sur ce petit sous-ensemble pour deviner la position correcte du fragment. Nous devons donc filter la liste de ces associations.

II. Filtrage des associations par RANSAC

RANSAC, *RANdom SAmple Consensus*, est une méthode pour estimer les paramètres de certains modèles mathématiques. Plus précisément, c'est une méthode itérative utilisée lorsque l'ensemble de données observées peut contenir des valeurs aberrantes (outliers).

Elle est utilisée pour un jeu de donnée avec de nombreuses valeurs aberrantes pour lesquelles une ligne doit être ajustée.



Figure 3: Jeu de donnée avant passage au RANSAC

Voici ci-dessous le résultat après passage au RANSAC

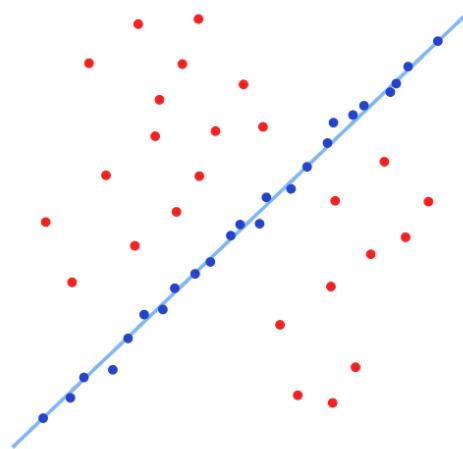


Figure 4: Jeu de donnée après passage au RANSAC

Dans notre cas, K, le nombre minimum d'associations qui vous permet de calculer une paramétrisation x, y, θ pour la transformation appliquée au fragment est de 2. En effet, pour le calcul de l'angle de rotation, nous avons besoin d'au moins deux associations entre fragment et fresque.

Nous pouvons appliquer un algorithme de type RANSAC. Pour cela, deux choix s'offre à nous :

1. Utiliser une librairie. Exemple : *scikit-image*.
2. Implémenter nous même un algorithme de type RANSAC.

Par soucis d'efficacité, nous avons choisi de partir sur l'utilisation d'une librairie. Voici sont utilisation pour filtrer les associations :

```
def filter_matches_using_RANSAC(keypoints_fragment, keypoints_fresco, all_matches):
    matches = []
    for m,n in all_matches: matches.append(m)

    src_pts = np.float32([ keypoints_fragment[m.queryIdx].pt for m in matches ]).reshape(-1, 2)
    dst_pts = np.float32([ keypoints_fresco[m.trainIdx].pt for m in matches ]).reshape(-1, 2)

    rng = np.random.default_rng()

    # generate coordinates of line
    x = np.arange(-200, 200)
    y = 0.2 * x + 20
    data = np.column_stack([src_pts, dst_pts])

    # add gaussian noise to coordinates
    noise = rng.normal(size=data.shape)
    data += 0.5 * noise
    data[::2] += 5 * noise[::2]
    data[::4] += 20 * noise[::4]

    # fit line using all data
    model = LineModelND()
    model.estimate(data)

    # robustly fit line only using inlier data with RANSAC algorithm
    model_robust, inliers = ransac(data, LineModelND, min_samples=2,
                                    residual_threshold=8, max_trials=10000)
    outliers = inliers == False

    # generate coordinates of estimated models
    line_x = np.arange(-500, 500)
    line_y = model.predict_y(line_x)
    line_y_robust = model_robust.predict_y(line_x)

    fig, ax = plt.subplots()
    ax.plot(data[inliers, 0], data[inliers, 1], '.b', alpha=0.6,
             label='Inlier data')
    ax.plot(data[outliers, 0], data[outliers, 1], '.r', alpha=0.6,
             label='Outlier data')
    ax.plot(line_x, line_y_robust, '-b', label='Robust line model')
    ax.legend(loc='lower left')
    plt.show()

    n_inliers = np.sum(inliers)

    inlier_keypoints_left = [cv.KeyPoint(point[0], point[1], 1) for point in src_pts[inliers]]
```

```

inlier_keypoints_right = [cv.KeyPoint(point[0], point[1], 1) for point in dst_pts[inliers]]
good_matches = [cv.DMatch(idx, idx, 1) for idx in range(n_inliers)]

src_pts = np.float32([ inlier_keypoints_left[m.queryIdx].pt for m in good_matches ]).reshape(-1, 2)
dst_pts = np.float32([ inlier_keypoints_right[m.trainIdx].pt for m in good_matches ]).reshape(-1, 2)

return good_matches

```

Cependant, voici également une implémentation “maison” moins performante d’un algorithme de type RANSAC :

```

import random
import numpy as np

def ransac(data, estimate, is_inlier, sample_size, goal_inliers, max_iterations, stop_at_goal=True, random_seed=0):
    best = 0
    best_model = None
    random.seed(random_seed)
    data = list(data)
    for i in range(max_iterations):
        print(sample_size)
        s = random.sample(data, int(sample_size))
        m = estimate(s)
        ic = 0
        for j in range(len(data)):
            if np.abs(m.dot(augment([data[j]]).T)) < 0.1:
                ic += 1

        if ic > best_ic:
            best = ic
            best_model = m
        if ic > goal_inliers and stop_at_goal:
            break
    return best_model, best_ic

```

Nous obtenons les résultats suivants :

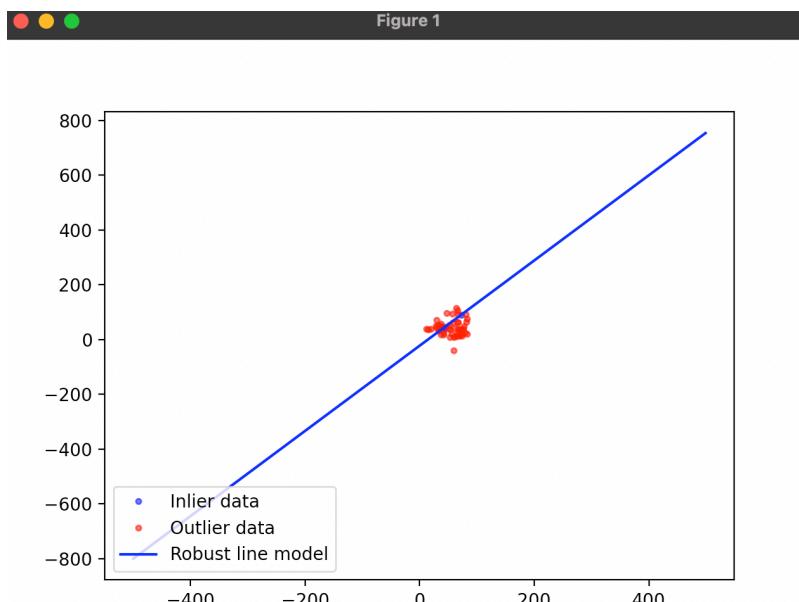


Figure 5: Résultats du filtre par RANSAC détectant les inliers et les outliers

III. Filtre des associations par le Lowe Ratio Tests

Pour filtrer les associations, nous pouvons également utiliser la méthode du Lowe's ratio test. Ce test de ratio à pour but d'augmenter la robustesse de l'algorithme SIFT.

Le but de ce test est de se débarrasser des points qui ne sont pas assez distincts. L'idée générale est qu'il doit y avoir une différence suffisante entre la première meilleure correspondance et les secondes meilleures correspondances.

Pour cela, nous extraisons une deux correspondances et nous comparerons leurs mesures de distance :

- Si la distance de la première meilleure correspondance est très éloignée de la distance de la deuxième correspondance, cela signifie que le descripteur de ce point est probablement suffisamment distinct et donc ces deux points devrait être une bonne correspondance.
- Si la première correspondance est assez proche de la deuxième correspondance, cela signifie qu'ils ne peuvent pas être une bonne correspondance.

Pour comparer l'éloignement des valeurs de ces deux distances respectives, nous utilisons un facteur ratio. Dans notre cas, la valeur est de $r = 0.55$.

Voici ci-dessous une image expliquant la réalisation de ce test :

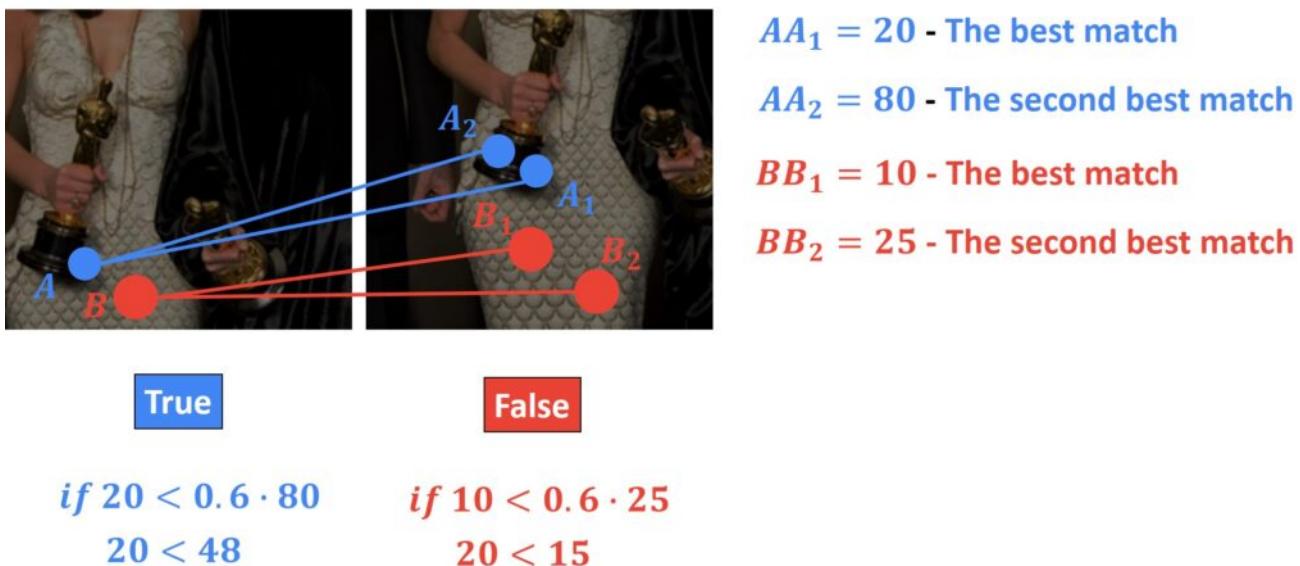


Figure 6: Explication du Lowe Ratio Test

Nous pouvons voir deux points caractéristiques A et B dans la première image. Nous allons faire correspondre les descripteurs de ces points avec la première meilleure correspondance et la deuxième meilleure correspondance dans une deuxième image. Le test de Lowe vérifie alors que les deux mesures de distance sont suffisamment différentes. Si elles le sont, ce point est préservé. En revanche, si elles ne sont pas assez distinctes, alors le point clé est éliminé et ne sera pas utilisé pour d'autres calculs.

Voici l'implémentation de ce test :

```
def filter_matches_using_Lowe_ratio_test(all_matches):
    #-- Filter matches using the Lowe's ratio test
    ratio_thresh = 0.55
    good_matches = []
    for m,n in all_matches:
        if m.distance < ratio_thresh * n.distance:
            good_matches.append(m)

    return good_matches
```

Nous pouvons essayer voir sur la fresque le résultat de ce premier filtre :

```
#-- Draw the good matches points gave by the Lowe ratio test
```

```
img_fresco_with_goodmatches = draw_matches(img_fragment, img_fresco, keypoints_fragment, keypoints_fresco)
cv.imshow("Only good matches", img_fresco_with_goodmatches)
```

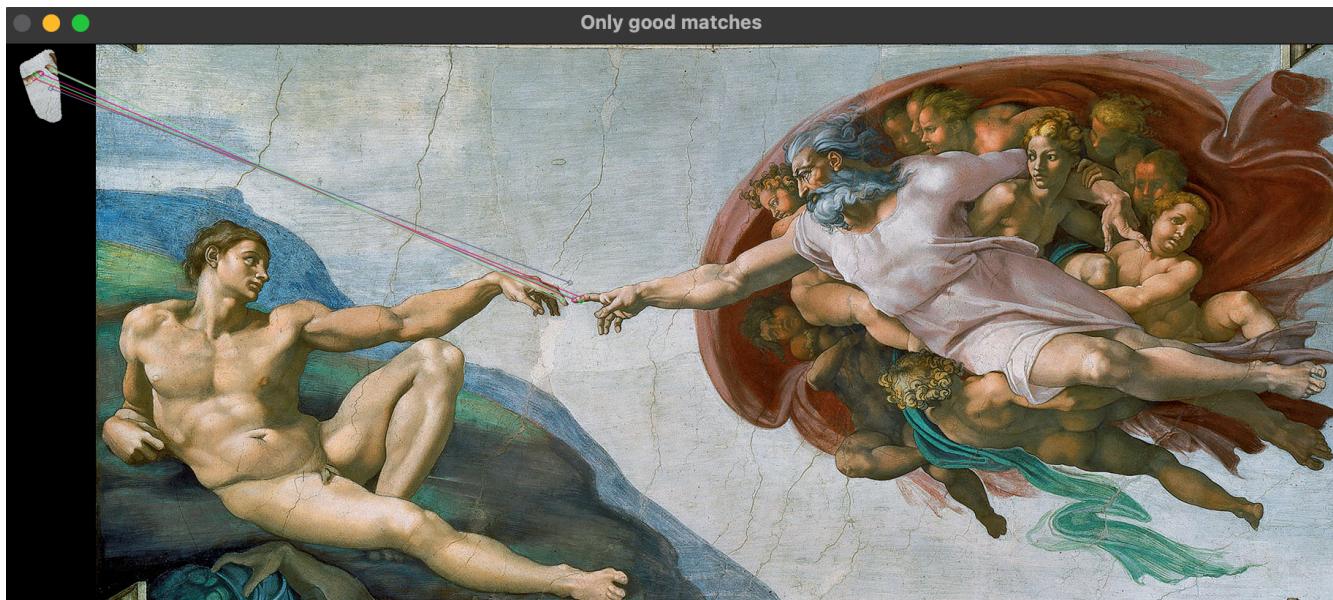


Figure 7: Associations filtrées avec le Lowe Ratio Test

Ensuite, une fois notre plus grand sous-ensemble obtenu, nous pouvons enfin calculer x, y, θ

IV. Calcul des coordonnées du fragment

En effet, pour calculer paramétrisation x, y, θ , nous pouvons réaliser des calculs mathématiques assez simple :

Pour x, y , nous calculons la moyenne des valeurs des bonnes associations (filtrées) dans la fresques.

- **Calcul de x :**

Soit n le nombre d'association,

$$x = \frac{\sum_{i=1}^n x_i}{n}$$

- **Calcul de y :**

Soit n le nombre d'association,

$$y = \frac{\sum_{i=1}^n y_i}{n}$$

Pour θ , l'arccosinus de produit des normes des associations de la fresques et du fragments centrée en [-1;1], tout cela transformé en dégré.

- **Calcul de θ :**

$$\theta = -\arccos(\|v_{\text{fresque}}\| * \|v_{\text{fragment}}\|)$$

centrée en $[-1, 1]$, en radian

$$\theta_{\text{deg}} = \frac{(\theta_{\text{rad}}) \times 180}{\pi}$$

centrée en $[-1, 1]$, en degré

Voici l'implémentation de ce calcul des coordonnées du fragment :

```
##-- Calcul of the position x by the average of the x fresco matches coordonates
def compute_fragment_position_cordonates(fragment_matches, fresco_matches):
    ##-- Calcul of the position x by the average of the x fresco matches coordonates
    x = 0
    for i in range(len(fresco_matches)):
        x = x + fresco_matches[i, 0]
    x = x / len(fresco_matches)

    ##-- Calcul of the position y by the average of the x fresco matches coordonates
    y = 0
    for i in range(len(fresco_matches)):
        y = y + fresco_matches[i, 1]
    y = y / len(fresco_matches)

    ##-- Compute of the rotation angle théta by the average of the x fresco matches coordonates
    v1 = np.array((fresco_matches[0][0]-fresco_matches[1][0], fresco_matches[0][1]-fresco_matches[1][1]))
    v2 = np.array((fragment_matches[0][0]-fragment_matches[1][0], fragment_matches[0][1]-fragment_matches[1][1]))
    v1_norm = v1 / np.linalg.norm(v1)
    v2_norm = v2 / np.linalg.norm(v2)
    theta = -np.arccos(np.clip(np.dot(v1_norm, v2_norm), -1.0, 1.0)) * 180 / np.pi
    if theta == np.nan or theta == np.NaN or theta == np.NAN :
        theta = 0

    return x, y, theta
```

Nous obtenons donc, pour le fragment 92, les coordonnées suivants :

Les résultats s'approchent énormément du fichier solution initial sans être exacte notamment, pour la valeur de la rotation.

```
===== TI - Fresco Reconstruction =====
APP5 Info - Polytech Paris-Saclay ©
Julien SAVE & Alexis DA COSTA

Fresco image path to compute : src/images/image.jpg
Fragment directory path to compute : src/images/frag_eroded/

The fragment 92 is at the coordonates :
92 654 345 -162.924
```

Figure 8: Les résultats des coordonnées obtenus par notre traitement sur le fragment 92

```
fragments.txt — Edited
87 1499 459 -60.9877
88 430 509 -280.545
89 1243 80 -125.713
90 958 286 -260.553
91 1278 441 -357.277
92 652 322 -199.022
93 854 661 -86.296
95 1049 109 -168.084
96 1043 387 -72.9027
97 882 97 -259.864
98 1181 128 -241.934
99 753 280 -358.011
100 1632 213 -324.021
101 1336 692 -160.09
```

Figure 9: Fichier fragment.txt des solutions contenant les positions des fragments à l'origine

V. Construction de la fresque

Voici le résultat obtenu à partir du filtrage des associations avec le Lowe Ratio Test :



Figure 10: Résultat de la reconstruction de la fresque

Comme nous pouvons voir, nos résultats ne sont pas forcément très satisfaisants.