Introduction to Computer Vision – Homework 2
RA192617 – Edgar Rodolfo Quispe Condori
RA192618 – Darwin Ttito Concha
Institute of Computing, University of Campinas (UNICAMP)
Campinas-SP, Brazil, 13083-852

1 Keypoints Detector and Descriptor

The algorithms that we implemented are based on the ORB to detect and SIFT to describe keypoints, it has slithly modification that were done based in experimental results and requested task.

1.1 Keypoints Detector

We used ORB like interest point detector. The original approach uses FAST(Features from Accelerated Segment Test) to detect interests points and assings an orientation based in the intensity centroid.

Originaly, FAST uses non-maximal suppression for removing adjacent corners and machine learning to improve the performance of the algorithm. In our implementation we do not use machine learning because the explanation is not clear. Our implementation and experiment consider three parameters :

- *Threshold*: this parameter define if the sixteen neighbors point are lighter, darker or similar to reference point.
- *N* : condition to determine if a point is a keypoint(at least N consecutive neighbors must be lighter or darker to reference point).
- *Non-Maximal Suppression*: this parameter removes adjacent corners that are too close between them.

FAST does not produce a measure of cornerness (it has large responses along edges). As similar to the original implementation of ORB we employ a Harris corner measure to order and select N keypoints. Another limitaton of FAST is that it does not produce orientation. To compensate this issue we use a approach similar to SIFT(ORB use moments of the patch around the keypoint). This technique take a window around of each keypoint and collect gradient directions and magnitudes. Then, a histogram is created for this and the amount that is added to the bin of the histogram is proportional to the magnitud of gradient at each points. Also, we used a gaussian weighting function, this function is multiplied by the magnitude. The farther away, the lesser the magnitude added.

Figure xxx shows results with different parameter settings.

To do: poner ejemplos con diferentes configuraciones de parametros.

1.2 Keypoints Descriptor

The original SIFT considers a pyramid in order to be invariant to scale, the keypoint descriptor is then computed from a specific level of this pyramid. Analysing the data it is easy to see that the scale differents between adjacent frame are negligibles. Thus, in our implementation ignores this stage in order to improve the performance of the algorithm.

Figure xxx shows results of the implementation.

To do: poner ejemplos con diferentes configuraciones de parametros.

2 Match hypothesis

Our implementation considers a simple match algorithm(brute force). For each keypoints of frame i we find the closest match in the frame $i+1$. We compare *L2 Norm* and *cosine* metrics to measure the similarity. Note that this implementation can generate the same match for two different keypoints yielding an outlier.

And idea of improvement of performance of this algorithm is to use *KNN(K-Nearest Neighbors)*. For real time aplication this may be a good option because its complexity is $O(n \log n)$, while our approach is $O(n^2)$ (n is the number of keypoints).

Figure xxx shows the different result comparing *L2 Norm* and *cosine* metrics.

3 Affine Tranformation Fitting

Our implementation considers **Affine and Projective** tranformation.

The number of iteration of RANSAC has been determinated experimentaly, our implementation has the following modules:

```
def least_square(src, dst, matches, k_points, transformation = 'affine'):
def evaluate_transformation(src, dst, matches, trans_params, threshold = 1, evaluation_method = 'ramsac', transform
def ransac(src, dst, matches, k = 3, S = 35, threshold = 1, transformation = 'affine'):
```

- *least_square*: this funtion creates the matrix X, A and Y, depending of the parameter $transformation$. Given N points $x_i, y_i (1 \leq i \leq N)$, in case of affine transformation, the matrix X and Y has to rows for each point:

$$X = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_N & y_N & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_N & y_N & 1 \end{bmatrix} Y = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ \vdots \\ x'_N \\ y'_N \end{bmatrix}$$

The parameters we want to find are defined for matrix A, in this case at least three points are needed and the transformation is computed using equation 1.

$$\begin{aligned} x' &= ax + by + c \\ y' &= dx + ey + f \end{aligned} \tag{1}$$

$$A = \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}$$

In the case of the projective transformation, the matrix X and Y are defined by:

$$X = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x'_2x_2 & -x'_2y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -y'_2x_2 & -y'_2y_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \\ x_N & y_N & 1 & 0 & 0 & 0 & -x'_Nx_N & -x'_Ny_N \\ 0 & 0 & 0 & x_N & y_N & 1 & -y'_Nx_N & -y'_Ny_N \end{bmatrix} Y = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ \vdots \\ x'_N \\ y'_N \end{bmatrix}$$

The parameters we want to find are defined for matrix A, in this case at least four points are needed and the transformation is computed using equation 2.

$$\begin{aligned} x' &= ax + by + c - gx'x - hx'y \\ y' &= dx + ey + f - gy'x - hy'y \end{aligned} \tag{2}$$

$$A = \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix}$$

Matriz A in both cases can be computed using least square. We use $numpy$ for easy implementation, sometimes the inverse does not exists, thus we return and empty vector.

```
x_transpose = np.matrix.transpose(X)
A = np.dot(x_transpose, X)
if np.linalg.det(A) == 0:
  print('Points', k_points, 'are not suitable for the transformation')
  return []
A = np.dot(np.linalg.inv(A), np.dot(x_transpose, Y))
return A
```

- *evaluate_transformation*: this function evaluates how many points fit correctly the compute transformation based on a threshold. The book suggests it to be between one and three. In our implementation we consider a value of two because our matches in the previous step are not perfect and we want to avoid missfitting because of outliers. The output of this function are the indexes of the points that fit the given parameters.
- *ransac*: this function computes the RANSAC algorithm iterating S times picking k aleatory points. Our main assumptions is that least there is one valid solution that can be reached in S iterations. The parameter k is set to three for affine and four for proyective transformation.

4 Transform

In order to explain our algorithm consider the following notation: i refers to the $i-th$ frame in the original video, and i' refers to the $i-th$ frame in the stabilized video.

Initially, we consider the following algorithm to compute i':

1. extract keypoints(X) from $(i-1)'$
2. extract keypoints(Y) from i
3. find matches(M) between X and Y
4. compute transformation T from M.
5. apply transformation T to i

With this approach, we were obtaining a lot of misstransformations(Figure xxx) because the difference between i and $(i-1)'$ are complex. Thus we defined anothe approach:

1. extract keypoints(X) from $i-1$
2. transform keypoints location of $i-1$ based on $(i-1)'$
3. extract keypoints(Y) from i
4. find matches(M) between X and Y
5. compute transformation T from M.
6. apply transformation T to i

We assume that the difference between adjacent frame in the original video is small. Thus, we compare the feature vector of i and $i-1$, but the keypoints location of i and $(i-1)'$.

Once we have the transformation of parameters we apply it for every pixel in i, depending of the movement, some black holes appears(Figure xxx). Thus, we interpolate this points with the average of four neighbors and fill with zero the cases when the neighbors are also empty.