MO446 Computer Vision: Project 2

Juan Hernández
RA: 163128
Email: juan.albarracin@ic.unicamp.br

Miguel Rodriguez
RA: 192744
Email: m.rodriguezs1990@gmail.com

I. Introduction

The objective of this work is to stabilize a video that contains movements at the moment of being captured. In order to achieve this goal it is defined that the stabilization can be performed by searching for the homogeneous matrix that transforms a frame t into the previous frame $t-1$. In order to find this matrix it is necessary to construct a model based on the similar key points between both frames using the RANSAC algorithm, which is a heuristic that allows finding an optimal homogeneous matrix to perform this transformation. In turn it is also necessary to find the key points of both frames, so it is necessary to find and use a descriptor that is invariant to all the movements or events that happen in the video to be corrected, so we decided to use SIFT, which is invariant to translation, rotation, scale and illumination.

We performed three types of experiments, the first experiments were to test our implementation of SIFT, then experiments to test our implementation of RANSAC to find an affine transformation and finally experiments to test our RANSAC looking for a projective transformation. These two last experiments were using the SIFT implementation of OpenCV, because it is a more optimized implementation (execution time) and gives us more security in the quality of the key points found.

II. Interest Point Detection and Description

Find the key points in images is a common problem in computer vision area. When we want to match the key point of two images that have the same scale, rotation, etc., a simple border descriptor can work to solve this matching problem. But in the case of images with different size, scales, angles, etc., a more powerful descriptor is necessary.

For our specific problem, we know that the images we want match can be rotated, scaled, translated. So it is necessary that out descriptor complies with having at least invariance in these three points. Our choice of which descriptor we use to extract the key points is SIFT (Scale-invariant feature transform). We Chose this descriptor because it meets the requirements of invariance, is a well known in the literature and have different implementation in free access libraries.

A. *SIFT: Scale-invariant feature transform.*

The main idea of SIFT algorithm is find the key points in the image and create a descriptor based on the orientations of this and its neighborhood. SIFT complies with being invariant to scale, translation, rotation and illumination. To find the key points and then describe them the SIFT algorithm can be divided into the following steps:

1) Constructing a scale space and LoG Approximation: The main idea of this stage is construct a Gaussian pyramid of O levels, each level is called octave and is composed of S images of the same size, called scales. To construct this pyramid it is necessary create O images from the original image, the first level is twice the size of original image, the next one is the original image, the third is the half and so on dividing by half. To construct the S images of each octave, it is necessary to apply a convolution with a Gaussian filter, which is created from a standard deviation that increases in reason to the scale level. In this stage the pyramid is constructed because looking for key points in different resolutions of the image, this becomes scale invariant.

After constructing the Gaussian pyramid, it is necessary create a new LoG structure (Laplacian of Gaussians), which is composed of the differences of the scales in each octave, for which we obtain $S-1$ scales in each octave. This structure of Gaussian differences allows to detect the edges quickly, and is necessary to find the key points.

2) Finding key points: The process of obtaining a key point is performed for each pixel, of each scale in each octave of the Gaussian differences. For each pixel visited is necessary to know if that point is a possible key point, so we need to check if the pixel is a maximum or a minimum in both the neighborhood and the upper/lower scale (See Listing 1). If the point is a max/min it is marked as a possible key point.

```
def is_local_opt(cube):
  min = True
  max = True
  for i in range(cube.shape[0]):
      for j in range(cube.shape[1]):
          for k in range(cube.shape[2]):
              if i == 1 and j == 1 and k == 1:
                  continue
              if max and cube[i,j,k] >= cube[1,1,1]:
                  max = False
                  if not min:
                      return False
              if min and cube[i,j,k] <= cube[1,1,1]:
                  min = False
                  if not max:
                      return False
  return True
```

Listing 1: Function that verifies if the point of interest is a possible key point verifying if it is a maximum or a minimum in its cubic neighborhood.

At the moment of finding a key point candidate, we know that this is a possible local max/min, but when working with discrete spaces, we can not assume that this is really max/min, so it is necessary to perform a reconstruction of the function in that neighborhood, for this we use the Tylor interpolation, which consists in approximate the real function from a sum of its derivatives of different orders. Finding the Taylor approximation allows us to find the $\hat{x}$, which represents the coordinates of the true max/min of the neighborhood and $f(\hat{x})$ the real value of the maximum (See Listing 2).

```

dx = (octave[s, i+1, j] - octave[s, i-1, j]) * 0.5 / 255
dy = (octave[s, i, j+1] - octave[s, i, j-1]) * 0.5 / 255
ds = (octave[s+1, i, j] - octave[s-1, i, j]) * 0.5 / 255

dxx = (octave[s, i+1, j] + octave[s, i-1, j] - 2 * octave[s, i, j]) / 255
dyy = (octave[s, i, j+1] + octave[s, i, j-1] - 2 * octave[s, i, j]) / 255
dss = (octave[s+1, i, j] + octave[s-1, i, j] - 2 * octave[s, i, j]) / 255

dxy = (octave[s, i-1, j-1] + octave[s, i+1, j+1] - octave[s, i-1, j+1] - octave[s, i+1, j-1]) * 0.25 / 255
dxs = (octave[s-1, i-1, j] + octave[s+1, i+1, j] - octave[s-1, i+1, j] - octave[s+1, i-1, j]) * 0.25 / 255
dys = (octave[s-1, i, j-1] + octave[s+1, i, j+1] - octave[s-1, i, j+1] - octave[s+1, i, j-1]) * 0.25 / 255

dD = np.matrix([[dx] , [dy] , [ds]])
H = np.matrix([[dxx, dxy, dxs], [dxy, dyy, dys], [dxs, dys, dss]])

x_hat = np.linalg.inv(H.transpose() * H) * (H.transpose() * dD)

d_x_hat = octave[s,i,j] + np.dot(dD.transpose(), x_hat) * 0.5
```

Listing 2: Implementation of Tylor interpolation.

This interpolation process is performed to verify that this key point meets the three requirements to be considered a real key point:

Corner: a key point candidate to be considered, this must meet being a corner. To know if a key point is a corner, it is necessary to use the Eq. 1, introduced by Harris and Stephens [1], where **H** is a Hessian matrix of neighborhood.

$$\frac{Tr(\mathbf{H})}{Det(\mathbf{H})} < \frac{(r+1)^2}{r} \tag{1}$$

high contrast: To be considered as a final key point, it is necessary to present a high contrast, so the absolute real value of max / min $f(\hat{x})$ must be greater than a contrast threshold.

Distance to the real max/min: After calculating the actual position of the max / min with the interpolation of Tylor ($\hat{x}$) if it is a distance greater than 0.5 from the candidate key point, it is discarded. This is because if it is at a greater distance, it means that the max/min is actually closer to a other pixel of the neighborhood.

Los valores utilizados para los thresholds mensionados fueron $r = 10$ y $cthreshold = 0.03$, estos valores fueron escogidos debido a que el autor original de SIFT (Lowe [2]) dice que luego de mucha experimentacion encontro que esos son los valores optimos.

The values used for the measured thresholds were $r = 10$ and $cthreshold = 0.03$, these values were chosen because the original author of SIFT (Lowe [2]) says that after much experimentation that these are the optimum values.

3) Get the orientation of the key points: After obtaining all the key points of the image, it is necessary to find the magnitude and the orientation of each one. For each key point we take a window size of 5×5 and calculate the finite differences for each pixel and then use the Eq 2 and Eq 3 to calculate the magnitude and orientation respectively.

$$m = \sqrt{(L_{x+1,y} - L_{x-1,y})^2 + (L_{x,y+1} - L_{x,y-1})^2} \tag{2}$$

$$\theta = tan^{-}1\left(\frac{L_{x,y+1} - L_{x,y-1}}{L_{x+1,y} - L_{x-1,y}}\right) \tag{3}$$

After obtaining the magnitude and orientation of each pixel in the window, a histogram is created, with 36 bins, in which the frequency of appearance of the angles is not calculated, but the magnitudes of all the orientations are added belong to the same bin. Finally, the bin that has the greatest value as a key point is chosen, in case there is another orientation that can reach at least 80% of the maximum value, it is also considered as another key point with the same coordinate but with different main orientation.

4) Generate feature vector: The last step is to create the descriptors for each key point found. To do this, it is necessary to create a window on the key point, but this time the window is 16×16, which is sub divided into blocks of 4×4 and for each block a histogram is calculated of the same form that in the step of obtaining the orientation, with the difference that the histograms are calculated using eight bins.

Each of these histograms is normalized with respect to the main orientation of the key point, this to be able to give invariant with respect to the rotation. Also each of these histograms is normalized to the range $[0 - 1]$, then all values greater than 0.2 are updated to that value and again it is normalized, this double normalization is to gain invariant to the illumination.

Then each histogram obtained is weighted by the distance that has the key point, this power is made through a Gaussian filter, which makes the values of the histograms closest to the key point have a greater relevance in the description, than the more distant descriptors. As a final part each of these descriptors is concatenated in order in a large descriptor, which is of size $4 \times 4 \times 8$, which gives us descriptors of size 128.

It should be noted that each of the values used for thresholds and magic numbers are those proposed in the original paper. We set ourselves experimenting by changing these values, but the best results were using these parameters.

Comparing our implementation of SIFT with the implementation of the OpenCV library, we can see that our implementation has two major flaws, first is the number of key points that our implementation finds is much greater than that found by the OpenCV implementation (See Fig. 1), this may be due to some implementation error, because our thresholds have the same values as those of the OpenCV implementation.

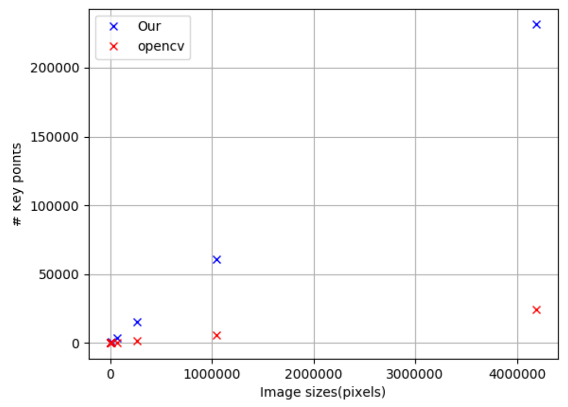

Fig. 1: Number of key points obtained depending on the size of the image, in blue our implementation and in red the implementation of OpenCV.

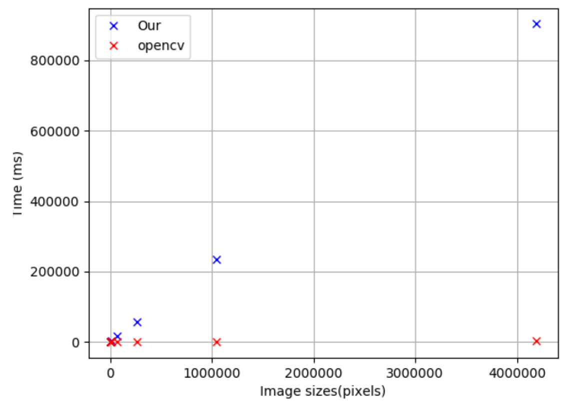

Fig. 2: The execution time in function of the size of the image, in blue our implementation and in red the implementation of OpenCV.

The other big difference between our implementation and OpenCV is the time it takes to execute the extraction and description of the points. In spite of trying to make an implementation using a array programming to do all the operations of extraction of features more quickly, our code to be implemented in Python, does not manage to surpass the implementation realized in C ++ of OpenCV, this can be seen in the Fig. 2, where you can see that as the number of pixels in the image grows, the execution time of our algorithms grows exponentially with respect to OpenCV. This is a problem to try to create a solution to our problem in real time, because our SIFT fails to make a revision of 30 frames per second.

B. Hypothesis Matching and Search

Once the key points and their corresponding features are found, we made a matching process. Our first approach consisted in making a comparison of all against all, which has an asymptotic complexity of $O(NM)$, being N and M the number of key points of each image respectively.

To be able to get real-time, we decided to make a more intelligent matching, using KDTree with descriptors (See Listing 3). To do this matching it is important to choose a threshold that indicates when there is a matching candidate, so the value of the threshold is a distance or an error depending on the way in which our descriptors are created, if the descriptors are normalized, the error will have a $[0-1]$ range, however if they are not, the distance can be any number greater than 0.

```
def kp_matcher(des0, des1, threshold):
    tree = spatial.KDTree(des0)
    _, matches = tree.query(des1,
    distance_upper_bound = threshold)

    return matches
```

Listing 3: Matching function using KDTree.

For the experiments performed with our implementation of SIFT, the descriptors to be normalized, we decided to use an error of 0.05, instead for the OpenCV descriptors we decided to use a distance 50 (because the OpenCV descriptors are not normalized) , these values were obtained after many experiments.

III. Affine Transformation Solver

The affine transformation model was implemented as an object with fit and $predict$ methods. Listing 4 shows the code written to fit the model, predict with the fitted model and to clone the model, since RANSAC needs it. The methods receive numpy arrays, so all the tasks can be done with only matrix operations, which is faster for many points.

```
class AffineTransformationModel(Model):

  def __init__(self):
    Model.__init__(self, 3, 2, 2)
    self.A = None

  def fit_model(self, x, y):

    m = x.shape[0]

    Xt = np.matrix(np.zeros((2 * m, 6), dtype=np.
    float64))
    Y = np.matrix(y.reshape(2 * m, 1))

    for i in range(m):
      Xt[2*i,0:2] = x[i]
      Xt[2*i,2] = 1.0
      Xt[2*i+1,3:5] = x[i]
      Xt[2*i+1,5] = 1.0

    self.A = (np.linalg.inv(Xt.transpose() * Xt) * (
    Xt.transpose() * Y)).reshape((3,2), order = 'F')

  def predict_model(self, x):
    x_ = np.append(x, np.ones((x.shape[0], 1)), axis
    =1)
    return np.matrix(x_) * self.A

  def clone(self):
    cl = AffineTransformationModel()
    cl.A = None if self.A is None else self.A.copy()
    return cl
```

Listing 4: Affine transformation model

IV. Projective Transformation Solver

The Projective Transformation model was implementing, by solving the next system:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x'_1x_1 & -x'_1y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x'_2x_2 & -x'_2y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x'_2x_2 & -x'_2y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x'_3x_3 & -x'_3y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x'_3x_3 & -x'_3y_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x'_4x_4 & -x'_4y_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x'_4x_4 & -x'_4y_4 \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix} \tag{4}$$

Once the h_{ij} are found, the transformation is done with:

$$\left(x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31} + h_{32} + h_{33}}, y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31} + h_{32} + h_{33}} \right) \tag{5}$$

Listing 5 shows the implementation of the model class:

```
class ProjectiveTransformationModel(Model):

  def __init__(self):
    Model.__init__(self, 4, 2, 2)
    self.A = None
    self.B = None

  def fit_model(self, x, y):
    m = x.shape[0]
    Xt = np.matrix(np.zeros((2 * m, 8), dtype=np.
    float64))
    Y = np.matrix(y.reshape(2 * m, 1))

    for i in range(m):
      Xt[2*i,0:2] = x[i]
      Xt[2*i,2] = 1.0
      Xt[2*i,6] = - y[i,0] * x[i,0]
      Xt[2*i,7] = - y[i,0] * x[i,1]
      Xt[2*i+1,3:5] = x[i]
      Xt[2*i+1,5] = 1.0
      Xt[2*i+1,6] = - y[i,1] * x[i,0]
      Xt[2*i+1,7] = - y[i,1] * x[i,1]


    H = np.linalg.inv(Xt.transpose() * Xt) * (Xt.
    transpose() * Y)
    self.A = np.zeros((3,2), dtype=np.float64)
    self.B = np.zeros((3,2), dtype=np.float64)

    self.A[0,0] = H[0,0]
    self.A[1,0] = H[1,0]
    self.A[2,0] = H[2,0]
    self.A[0,1] = H[3,0]
    self.A[1,1] = H[4,0]
    self.A[2,1] = H[5,0]

    self.B[0] = H[6,0]
    self.B[1] = H[7,0]
    self.B[2] = 1

  def predict_model(self, x):
    x_ = np.append(x, np.ones((x.shape[0], 1)), axis
    =1)
    return (np.matrix(x_) * self.A) / (np.matrix(x_)
     * self.B)
```

Listing 5: Projective transformation model

V. RANSAC

Listing 6 shows the implementation of the RANSAC algorithm. It works by default with an Euclidean error, but the distance can be generalized to any grade of the Minkowski. It can be the case that the selected points yield a non invertible matrix, so this exception must be managed.

```
def ransac(x, y, model, k, t, d, l = 2.0, seed =
    None):

  n = model.n_min

  if seed is not None:
    np.random.seed(seed)

  best = None
  e_min = np.inf
  inds = np.array(range(x.shape[0]))
  i = 0
  while(i < k):
    model_ = model.clone()
    np.random.shuffle(inds)

    try:
      model_.fit(x[inds[:n]], y[inds[:n]])
    except np.linalg.LinAlgError:
      continue

    y_ = model_.predict(x[inds[n:]])

    mkw = np.array(minkowski(y[inds[n:]], y_)).
    flatten()

    alsoinliers = np.where(mkw < t)[0]

    if alsoinliers.size <= d:
      i += 1
      continue

    alsoinliers = np.append(inds[:n], inds[
    alsoinliers + n])

    model_.fit(x[alsoinliers], y[alsoinliers])
    e = np.mean(minkowski(y[alsoinliers], model_.
    predict(x[alsoinliers]), l))

    if e < e_min:
      e_min = e
      best = model_

    i += 1

  return best, e_min
```

Listing 6: The RANSAC algorithm

The selection of parameters depended on assumptions that consider either the nature of data, the descriptor used, and the type of transformation, following the next assumptions:

- The videos contain small perturbations such as rotation, zooming and shaking. Sufficiently great perturbations simply cannot be reconstructed, since there is a significant loss of information and key points are likely to disappear, so there would not be matches.
- Since SIFT is invariant to all the motions that can be corrected by an affine transform, there is a really high probability of inliers inside the matched points, which are the input of RANSAC.

- For the same reasons listed above, the Euclidean distance, used as error, of the real points to the predicted by the model cannot be more than 1.0 pixel.

It was empirically found that the impact of the number of iterations in the total processing time is low with respect to the rest of stages of the whole pipeline. If the restrictions imposed by the threshold error t and the minimum number of inliers d to accept a model are high, the number of iterations k must be high enough to find a model that satisfies these restrictions. The time of an iteration of RANSAC depends on the number of matches, which do not increase too much each time, so we settled $k = 100$ and try to set d and t as high as possible, resulting in $t = 0.1$ and d to be 80% of the total number of matches. Although these parameters look very restrictive, we obtained good results with our videos, that could not be achieved by being less restrictive with d and t. Using the formula to compute the number of iterations, expecting probabilities $p = 0.99$ and $w = 0.8$ (as already), the number of iterations is 72, so we are using more iterations.

The last analysis was done for the pipeline considering the OpenCV implementation of SIFT. For our implementation of SIFT, the model had to be way less restrictive. We settled $d = 0.1$, $t = 5.0$, and $k = 150$, which is a little more iterations than the value suggested by the formula (130).

VI. Final Transformation and Alignment

Once the best model is found, we create an empty frame and, for each pixel of the original next frame, we use the model to map the value of this pixel at coordinates (i, j) to the coordinate (i', j') provided by the model, in the empty frame. Listing 7 shows the method that transforms the next frame (img) into the frame aligned with respect to the last one, by applying the provided model.

```
from scipy.interpolate import griddata

def transform(img, model, p):

  p_ = model.predict(p.astype(np.float64))
  img_ = np.zeros(img.shape, dtype = np.float64)

  for ch in range(img.shape[2]):
    grid = griddata(p_, img[:,:,ch].reshape((np.prod
    (img.shape[:2]))), p)
    grid[np.isnan(grid)] = 0
    for j in range(len(p)):
      img_[p[j,0],p[j,1],ch] = grid[j]

  return img_
```

Listing 7: Final transformation

The 2D-array p contains the standard pixels coordinates (i, j) of all the frames of the video. The model is applied to p to yield the new coordinates $p\_$ and, for each channel, the SciPy function $griddata$ is used to interpolate the integer coordinates to map to the blank frame, since the transforms coordinates not necessarily have integer values. The values at the borders of the frame, which could no be interpolated are set to 0.

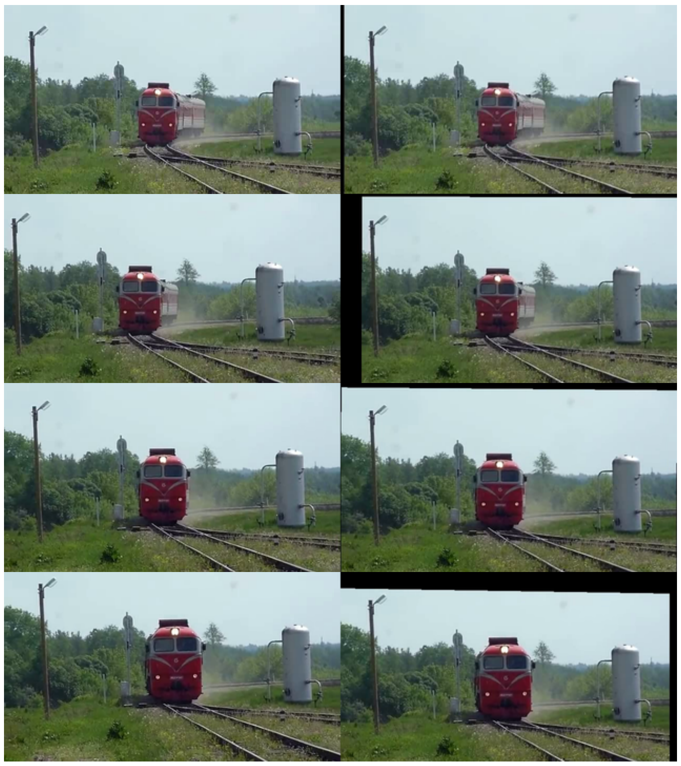

Fig. 3: Stabilized video of a moving train. Note how the train remains at the center of the frame.

VII. Results

We performed three sets of experiments, from which we will report the most relevant and informative. The first set includes the results for affine transformation with the OpenCV's implementation of SIFT. The second one includes the results for affine transformation with our implementation of SIFT. The last scenario includes results of the projective transformation, only with OpenCV's implementation of SIFT. The discussion for each scenario are on in their corresponding subsections.

A. *With OpenCV's SIFT*

Figures 3 and 4 show the results obtained in the stabilization using OpenCV's SIFT implementation. The reader can check the generated videos in the files `my-output/p2-com-1-affine.mp4`, `my-output/p2-com-3-affine.mp4` and `outuput/p2-com-2b.mp4`. The rest of the work was implemented by us, so this scenario was exclusively to test the RANSAC, the matching, the model, and the final transformation implementations, counting on the robust key points detection and description by the OpenCV implementation.

Figure 3 shows the stabilization of a video with a moving train (right) with respect to the original sequence (left) in different frames. Most of our videos had very restrictive perturbations, so the affine transformation could solve it, but this one includes moving targets in simple trajectories, and the affine transform yielded by RANSAC obtained very good results by maintaining the train and the white cylinder stable.

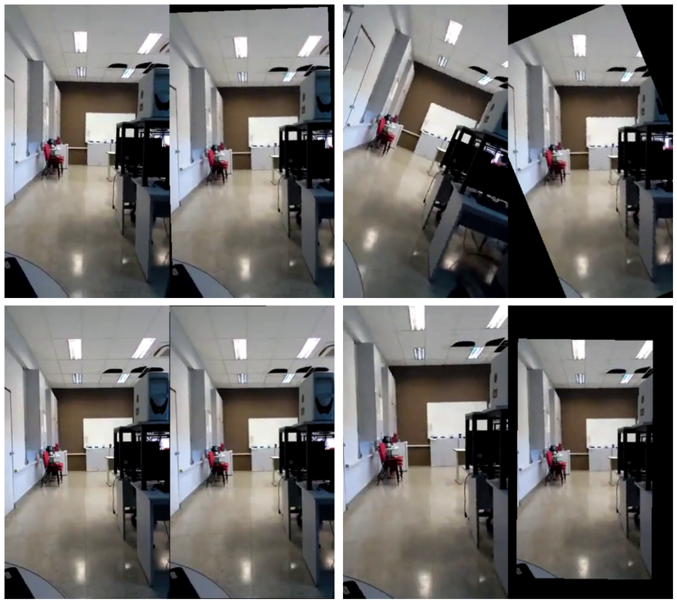
Fig. 4: Stabilized video of a static scene with shaking, rotation and zooming.

In particular, the input video was a longer sequence, in which a purposeful movement of the camera, where the train filled the whole frame and passed in front of the person who was filming. At that point, there was not enough matches that could be satisfied, so a transformation was not yielded, making the algorithm stop.

Figure 4 shows the stabilization of a video that we took, which included the three perturbations that are supposed to be overcome by SIFT and an affine transformations: shaking, rotation, and zooming. The top-left image compares the first frames of the original video (left) and the stabilized video (right), the bottom-left image shows the stabilization in the shaking moment. The top-right image shows the stabilization of a rotation, and the bottom-right image shows with zooming. It can be seen that the stabilization was successful in all the moments of the video.

B. With our SIFT

Once we guaranteed that the implementations of RANSAC, the model, the matching and the final transform were correct, we executed the pipeline, fully implemented by us. Figure 5 shows the results obtained. From one frame on, the whole image went black, but before it could be seen that the points provided by SIFT were not totally correct. As it was said before, our implementation returned a significantly higher number of points, that may not be really relevant, so this caused the RANSAC algorithm to get confused and return a model that makes a non-ideal projection. The reader can see the resulting video in `my-output/p2-com-2-our-sift.mp4`.

C. Projective Transform

Finally, we tested our Projective Transform model with the OpenCV's SIFT implementation. When it was tested with the videos we already had, it obtained similarly good

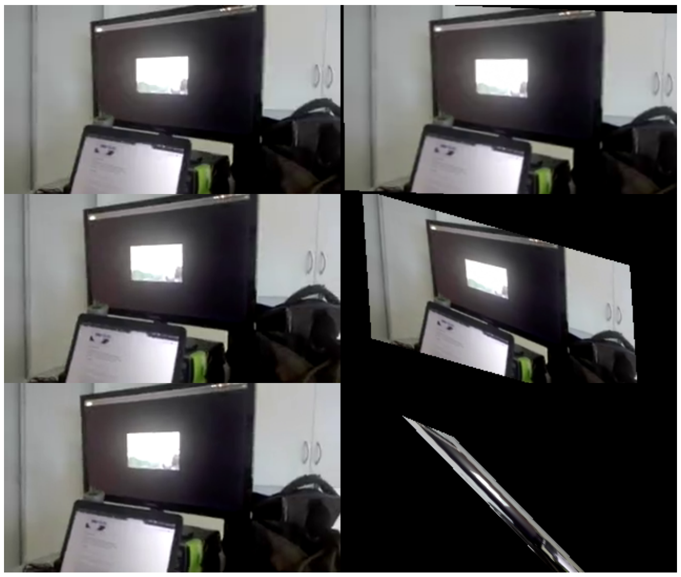
Fig. 5: Stabilized video of a static scene using our implementation of SIFT.

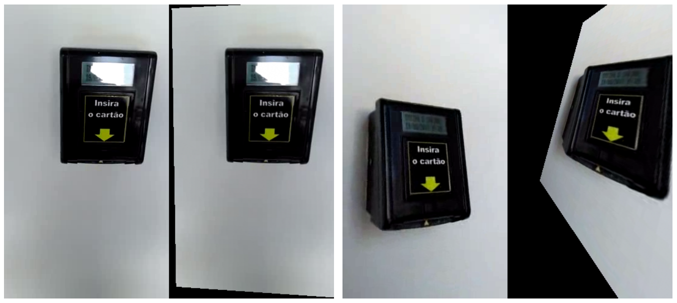

Fig. 6: Video stabilization of a static scene with pronounced viewpoint changes, using Projective Transmormation.

results as with the Affine Transformation, so we took a video with a pronounced viewpoint change, which is supposed to be corrected with this kind of transformation. Figure 6 shows how the stabilization was made at two different frames, and the reader can watch the video in `my-output/p2-com-4-projective.mp4`.

This stabilization was not successful, it was not because of the implementation of the model, but because of SIFT. With the other videos it worked correctly, while with this one it didn't. The reason is that the viewpoint changes in the other videos were not so pronounced as the current one, and SIFT, according to its paper, is invariant only for small changes of perspective. This caused that the obtained key points were not correctly detected and matched, causing the kind of projections showed in the figure. Nevertheless, the changes made in the images correspond to valid projections, showing that this model attains different results with respect to the Affine Transformation.

References

[1] C. Harris and M. Stephens, "A combined corner and edge detector," in *In Proc. of Fourth Alvey Vision Conference*, 1988, pp. 147–151.

[2] D. G. Lowe, “Object recognition from local scale-invariant features,” in *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ser. ICCV ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 1150–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850924.851523