Project 2

(191079) Nathana Facion
(192803) Rafael Mariottini Tomazela

Universidade Estadual de Campinas
Instituto de Computação

Prof. Adín Ramírez Rivera

September 2017

Contents

1 Video Stabilization Algorithm **3**
1.1 Interest Points and Descriptors . . . 3
1.1.1 Question . . . 3
1.1.2 Answer . . . 3
1.2 Match Hypothesis . . . 8
1.2.1 Question . . . 8
1.2.2 Answer . . . 8
1.3 Affine Transformation Fitting . . . 9
1.3.1 Question . . . 9
1.3.2 Answer . . . 11
1.4 Transform . . . 13
1.4.1 Question . . . 13
1.4.2 Answer . . . 13

2 Experiments **17**

Bibliography **21**

Chapter 1

Video Stabilization Algorithm

1.1 Interest Points and Descriptors

1.1.1 Question

The first step in the stabilization process is to find the interest points in your images and describing them. For this task you need to select an algorithm (or set of algorithms) to find your image descriptors. Some of the descriptors you can look at and implement include: SIFT, SURF, BRIEF, or ORB. Note that there are many other type of descriptors that you can read and implement. You need to explain the descriptor you selected and briefly explain your implementation. Put more emphasis on the explanation on the selection of parameters (like thresholds) of your algorithm and how did you selected. I recommend you to perform several experiments to find good parameters, and to understand what the parameters are doing (see 4).

1.1.2 Answer

We implemented the Sift [1] keypoint descriptor. Most of our implementation is based on the paper from *Rey Otero and Delbracio* [2].

Description

Sift works by first selecting candidate keypoints from the image, and then describing then in a way that is invariant to rotation.

To select the keypoints, it computes n_{oct} different octaves of the original image, with the first octave being the image double, the second being the original image, and the third and so on being half of the previous octave size. For each octave, it then calculates a scale of n_{spo} images of the same size, where each scale image has a gaussian applied to it, with higher σ for higher numbered scales. The first scale of a octave (except for the first one which is the original image) is gotten from the second scale of a previous octave.

Then for each pair of scales A and B of each octave, we calculate C = B - A. C will then be added to the Difference of Gaussians (DoG) pyramid for that octave. Each DoG image is then searched for extrema points, which can be a keypoint. Keypoints that are not sufficiently distinct are discarded. We also need to find the real value for each keypoint position and intensity instead of the discretized one, so we don't lose information when scaling back to the original image.

Figure 1.1: p2-1-7

Figure 1.2: p2-1-8

Parameter	Value
σ_{min}	0.8
δ_{min}	0.5
σ_{in}	0.5
n_{oct}	8
n_{spo}	2
C_{DoG}	0.015
C_{edge}	20
n_{bins}	36
λ_{ori}	1.6
t	0.8
n_{hist}	3
n_{ori}	8
λ_{descr}	6

Table 1.1: Sift parameters

Then for each keypoint, we first find it's main orientations θ of the gradient, discretizing in n_{bins} bins. For each keypoint, there will be one descriptor for each bin that has higher value than t (after smoothing it), that is, a keypoint can have more than one descriptor.

Finally, for each keypoint and its respectives θs, we calculate a matrix with size $n_{hist} \times n_{hist} \times n_{ori}$ for each patch close to the keypoint and oriented in respect to θ, where we will have a description in terms of gradient histogram for each region of the patch. The gradient for each pixel within the patch is then calculated, and its value distributed between close regions. The matrix is then transformed into a feature vector and normalized

We also need, in all these steps, to be careful to transform between the scaled octaves and the original image.

Parameters

There are a lot of parameters to tune, our choices can be seen in Table **??**.

Most of the parameters were chosen according to the article [2], since they have made some tests to find the best one. Still, we've tried to change some other parameters, such as C_{DoG}, increasing or decreasing it in order to find more points. But we found that the current value was a nice compromise between finding many keypoints but avoiding noisy ones. It is also the case that $n_{spo} = 3$ in the original paper, but we wanted to avoid long computations, and the results were still good (note that the value is 4 in our code, because in practice there are four layers for each Sift octave, but in theory there is only 3 layers of DoG, and only the layer in the middle will be searched for extrema points).

We can see some examples of keypoint detection of the algorithm in the images 1.3, 1.4, 1.5, 1.6, 1.7 and 1.8, where the keypoints are circled in green (each image corresponds with one of the same name in the input folder, which is the same image without the green circles). In the following sections we will show that descriptor works since the matches were found.

Figure 1.3: Sift keypoints: output/p2-1-0.png

Figure 1.4: Sift keypoints: output/p2-1-1.png

Figure 1.5: Sift keypoints: output/p2-1-2.png

Figure 1.6: Sift keypoints: output/p2-1-3.png

Figure 1.7: Sift keypoints: output/p2-1-4.png

1.2 Match Hypothesis

1.2.1 Question

For each pair of images, use your interest point extractor and describe all the points. You need to find the candidate matches between these images. For this operation you need to use a metric function (commonly an ‘ 2 norm, or a metric define for your descriptor) and find the closest match for every point. Prune your candidates using a threshold to have a higher confidence for the matches. Note that if your metric is not a distance you need to perform the process both ways. That is, compare the features of I1 with the features of I2, and compare the features of I2 with the ones in I1. Then, you keep the ones that are consistent. At the end of this step you should end with a set of pairings for the interest points between your two images. Lets try to recover the transformation. Explain your algorithm for the matching, and your metric to compare the interest points. Show some examples of the matches in your experiments, and explain your results.

1.2.2 Answer

We first calculated the descriptor for two images, A and B. Then we compared each keypoint in A to all keypoints in B, calculating the l2 norm of the two feature vectors. We then stored the two best matches,

Figure 1.8: Sift keypoints: output/p2-1-5.png

that is, for each keypoint in A, we stored the two keypoints in B with the smallest distance. If $dist1$ and $dist2$ are the distances between the keypoint A and the first and second best match in B, we then accepted this match if $dist1 < 0.7 \cdot dist2$. The idea is that if the match is real, and not the result of noise, then the second best match is probably noise and the distance is probably much lower. If the match is noisy, then both the first and the second distances are noisy and the difference between the distances are probably small. The parameter 0.7 was chosen because it was a nice compromise between finding few points and finding good matches.

Some matches can be seen in images 1.9, **??** and 1.11 (they use the keypoints found previously by Sift). The two images are put side by side, and the lines between the images are the matches that were found. We can see the correspondences were found even with slightly scaled images, and with rotations (which are not exactly 90 degrees).

1.3 Affine Transformation Fitting

1.3.1 Question

Our problem is to learn the transformation between our two images. For simplicity, we will assume an

Figure 1.9: Matches in yellow between the images: output/p2-2-0.png

Figure 1.10: Matches in yellow between the images: output/p2-2-1.png

Figure 1.11: Matches in yellow between the images: output/p2-2-2.png

$$x' = ax + by + c, \tag{1}$$
$$y' = dx + ey + f, \tag{2}$$

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \end{bmatrix},$$
$$XA = Y,$$

$$A = \left(X^T X\right)^{-1} \left(X^T Y\right).$$

affine transformation (but in general you need a projection). We will consider the transformation where (x',y') are our transformed coordinates and (x, y) are our original coordinates. Our objective is to learn the parameters a, b, c, d, e, and f for our pair of images. Since our affine transformation has 6 degrees of freedom we need 3 point matches to solve the equations (since each point gives two equations). We will use RANSAC to find the best transformation for our images. To do so, we need to select three point matches and solve the transformation they define. We will use least squares1 to solve our transformation. That is, we need to change the shape of our transformations for x'(1) and y'(2) to a matrix form. Our system is where X, A, and Y represent the corresponding matrix. Hence the affine transformation A can be computed by Note that you need to implement this operation instead of using a solver. Once you have the transformation A from a set of three points, you need to transform all the points from the first image using A, and see how many matches confirm the hypothesis on the transformed image. Repeat this step N times (explain how do you compute the number of iterations N needed in your RANSAC implementation). In your report you need to explain your implementation of RANSAC (in particular focus on explaining your assumptions and the way you select the iterations and other parameters). Also, explain how you compute (or solve) the affine parameters (5).

1.3.2 Answer

The classical RANSAC was used. We tried the adaptive RANSAC, but the results were worse. We could then calculate N based on the amount of expected outliers in our matches, but it is not possible to easily know the value. We then simply used a big enough value for N, which is 10000. It's possible that the adaptive implementation which changed the value of N based on new information would be better, and there was possibly a bug in our code when we implemented that. Still, $N = 10000$ is fast enough compared to the time Sift takes to run, and it yields good result. The problem is that this value should probably be changed when testing different videos or images, increasing it if the image has low entropy (that is, not a lot of interesting points, and probably a lot of noisy keypoints).

To detect inliers we simply found the affine matrix A for the three sampled points, we then proceeded to transform all the keypoints from X using this matrix, and we compared each resulting point to the corresponding one in Y. If the euclidean distance between the points was less than 4, we considered it an inlier. This parameter was also chosen after testing some values ranging from 1 (which mostly selected

Figure 1.12: Matches in yellow between the images, only inliers selected by RANSAC: output/p2-3-0.png

Figure 1.13: Matches in yellow between the images, only inliers selected by RANSAC: output/p2-3-1.png

Figure 1.14: Matches in yellow between the images, only inliers selected by RANSAC: output/p2-3-2.png

only the three sample points) and 10 (which found some inliers which shouldn't be)

Mini extra: One really small modification we did is that, instead of simply selecting the model which has the biggest amount of inliers, we selected the one which has the lowest summed distance between the ones with more inliers. This improved the results considerably.

We can see the result of applying RANSAC to 1.9, **??** and 1.11 respectively in the images 1.12, 1.13 and 1.14.

1.4 Transform

1.4.1 Question

After all the iterations on RANSAC you will select the affine transformation A that has themost amount of matches. Now compute a final transformation, A^, by solving (5) with all the matches (instead of three). Note that the matrix X can have a general form of 2M x 6 and Y can be 2M x 1, where M is the number of points you have (in this case your number of final matches). Explain your generalization of 5. You can explain it together with the explanation on how you solve the parameters for brevity. Using the final transformation A^ transform your image. You can, for example, transform the second frame into the reference frame of the first one. Then for the following frame (the third one) you want to align it with the aligned version of the second one. Repeat this process until all the frames in the video are aligned. Note that you may need to increase the spatial size of the video and fill the gaps in the frames with zeros to accommodate the variations on the video. (Explain your technique to deal with the resizing problem.) Explain your approach to align the images using the final transformation, and your assumptions to make it work.

1.4.2 Answer

To transform the image, we first found the affine matrix A, but now instead of finding it to transform image B into the frame of image C, we found it to transform image C into the frame of image B. The idea is that this matrix works like and inverse of matrix A (although it's not the inverse), so now we can use this matrix to compute the real value position x, y in image B, for each discrete value m, n of the resulting image B', which is B transformed to the frame of reference of C. We simply used the nearest neighbor of x, y in image B to attribute to m, n in the destination image.

To have the entirety of the original image in the transformed one, we need to pad the sides of the destination image with zeros. If we don't do that, some values will be cropped. That's not exactly bad depending on the purpose of the transform (we didn't pad it for the video), but we have the option to transform the image with a pading.

We can see the second image in both 1.12, 1.13 and 1.14 transformed to the reference of the other image in 1.15, 1.16 and 1.17. The padding on the sides were removed for the report, but it's originally 250 pixels.

Figure 1.15: The images transformed to the reference of the other and combined: output/p2-4-0.png

Figure 1.16: The images transformed to the reference of the other and combined: output/p2-4-1.png

Figure 1.17: The images transformed to the reference of the other and combined: output/p2-4-2.png

Chapter 2

Experiments

Our Sift implementation was really slow, so we tried to test only in really short videos. Although our Sift implementation worked fine with images which are rotated or cropped, it didn't work as well with videos, but the ending results were good.

Some problems and solutions we faced:

- Our algorithm works only with videos that don't change too much the content of the image, but just rotate/translate it. To make it work with videos that have movement, we would need to differentiate between unintended (jittery) hand movements, and purposeful ones, which is not a simple problem.
- Doing as suggested with the frame, that is, putting a frame in the frame of reference of a previous frame which was already transformed, didn't work so well. The problem is that errors in the transformations (which happens frequently) are accumulated fast, and at some point the transformations stop making sense. Since the videos have the same content, we found it enough to transform all images to the frame of the first one, which also sped up processing since we needed to describe each image only one time.
- If we didn't pad the video, some part of the image would be cropped after the affine transformation. Still, we didn't pad it since we found it better than having the entire image rotate or translate (which looked confusing), so we decided to maintain the cropped video. This is intended.
- We discarded the 30 first frames of each video. If we didn't do that, then the results get progressively worse since in the first frames the camera is still focusing, and the image is blurred.
- Some frames, after transformed, were entirely distorted. We can see such example in 2.2. The explanation is simply that the combination of Sift + matching + RANSAC didn't work so well for these frames. Still, a lot of tweaking was done to reach such result.
- All the videos were too big to be processed in a realistic time. We then cropped the videos, downsampled its size, and sample one frame for each two frames, effectively reducing the video fps by half. Although the matches were not as good as with the original video, at least the run time was realistic.

We didn't compare different parameters, since RANSAC is random and the results had a high variance. Different parameters had no immediate effect in the MSE of the aligned image, since in some runs, some

Figure 2.1: The first frame from the video: my-output/first_frame.png

frames were badly matched by RANSAC. We could probably make a good plot by taking the average MSE of all frames in many runs (removing the effect of the high variance), for different parameters, but since each run takes a long time to finish, it wasn't really possible to do so. In the end, we decided against plotting MSE values which could me misleading.

We could also probably make a better RANSAC implementation, in order to it behave less randomly. But since the runs for each video were already taking too long, we found that limiting the value of N was a good compromise.

The output video can be seen in the file *p2-5-0.avi*, it is 30 frames shorter than the original one, *p2-5-13.avi*, as explained before. We can also see the first frame in image 2.1, a somewhat distorted but good frame in 2.3 (which scaled the image in a bad way), and a good match in 2.4 (which rotate the image so it matches the first one without a big distortion, as expected).

For the test, we changed the position of the camera and rotate it. The resulting video after stabilizing is, ironically, jittery. But it preserves the orientation of all the objects in the scene, even if the camera is rotated. Although the algorithm is not ready for real stabilization of videos, it shows that it can be done. Improvements that would make the video less jittery includes a more robust description of the image (which implies in a more efficient Sift to run it with other parameters) and a improvement of RANSAC that would, somehow, avoid distortions in the image. Also using projective transform would probably help, since many of the problems with the matches are probably due to minor perspective changes in the camera.

Figure 2.2: A distorted frame from the video: my-output/distorted.png

Figure 2.3: A slightly less distorted frame from the video: my-output/distorted_good.png

Figure 2.4: A good transformation: my-output/good.png

Bibliography

[1] David G. Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[2] Ives Rey Otero and Mauricio Delbracio. Anatomy of the SIFT Method. *Image Processing On Line*, 4:370–396, December 2014.