Long exposure photography for hand-held cameras

Seddik Mekki

Abstract

The goal of this project (Github project link: https://github.com/SedMek/Long-Exposure-Photography-for-Hand-held-Cameras) is to allow photography amateurs to take stylish long exposure photos using only the regular camera mode, without the need of a tripod. We will shoot a video of a scene using a hand held camera, stabilize it, then merge the average of different frames to get the long exposure photo using only deterministic methods, no former training needed.

1. Introduction

Being a photography amateur, one of my favorite photography techniques is long exposure, which is the process of creating a photo that shows the effect of passing time, something that traditional photography does not capture.

It is particularly used to make water becomes silky smooth, stars in a night sky leave light trails as the earth rotates, and car headlights/taillights illuminate highways in a single band of continuous motion (fig 1).

The issue is that to be able to get such pictures, one needs to have a set of hardware (tripod and professional camera or phone camera with ”pro” mode), which is not always the case. So in this article, we will try to offer another solution that uses OpenCV in Python.

1.1. Classic Method

The regular way to capture these types of shots, we need to take a methodical approach: mounting your camera on a steady tripod to eliminate motion blur, applying various filters to reduce the reduce the amount of light that goes to the sensor, computing exposure values, etc. The needed tools are displayed in fig 2.

1.2. Suggested alternative

To get rid of the professional camera and the filters, we can simulate long exposures by applying image/frame averaging. By averaging the images captured from a mounted camera over a given period of time, we can (in effect) simulate long exposures.

And since videos are just a series of images, we can easily construct long exposures from the frames by averaging all frames in the video together. The result is stunning long exposure-esque images.

And to get rid of the tripod, we will capture the video with a hand held camera, then we will use Digital Video Stabilization: This method does not require special sensors for estimating camera motion.

	Classic	**Alternative**
Scene stability	Tripod	Digital Video Stabilization
Passing time effect	Professional camera sensor and filters	Image Averaging

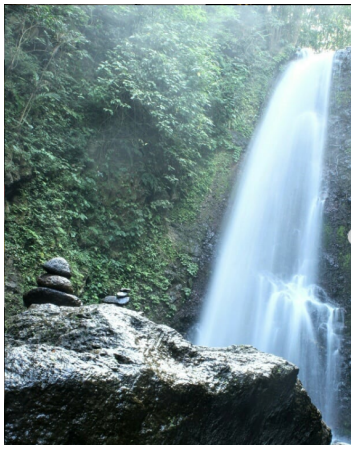

Figure 1. Silky smooth waterfall.

Figure 2. Classic long exposure photography tools.

1.3. Programming language and kits

This project has been developed in Python, using OpenCV 4.0.0.21 and Numpy 1.15.4 librairies.

2. Presenting the solution

As explained in the introduction, our solution is composed of two blocks.

2.1. Video Stabilization

Video stabilization refers to a family of methods used to reduce the effect of camera motion on the final video. The motion of the camera would be a translation (i.e. movement in the x, y, z-direction) or rotation (yaw, pitch, roll).

2.1.1 Context

It is extremely important in consumer and professional videography. Therefore, many different mechanical, optical, and algorithmic solutions exist. Even in still image photography, stabilization can help take handheld pictures with long exposure times.
In medical diagnostic applications like endoscopy and colonoscopy, videos need to be stabilized to determine the exact location and width of the problem.
Similarly, in military applications, videos captured by aerial vehicles on a reconnaissance flight need to be stabilized for localization, navigation, target tracking, etc. The same applies to robotic applications.

2.1.2 Different approaches

Video Stabilization approaches include mechanical, optical and digital stabilization methods. These are discussed briefly below:

- **Mechanical Video Stabilization:** Mechanical image stabilization systems use the motion detected by special sensors like gyros and accelerometers to move the image sensor to compensate for the motion of the camera.
- **Optical Video Stabilization:** In this method, instead of moving the entire camera, stabilization is achieved by moving parts of the lens. This method employs a moveable lens assembly that variably adjusts the path length of light as it travels through the camera's lens system.
- **Digital Video Stabilization:** It is the solution that we will be using, further explained in the next section.

2.2. Image Averaging

Given an input video, this block will average all frames together (weighting them equally) to create the long exposure effect.

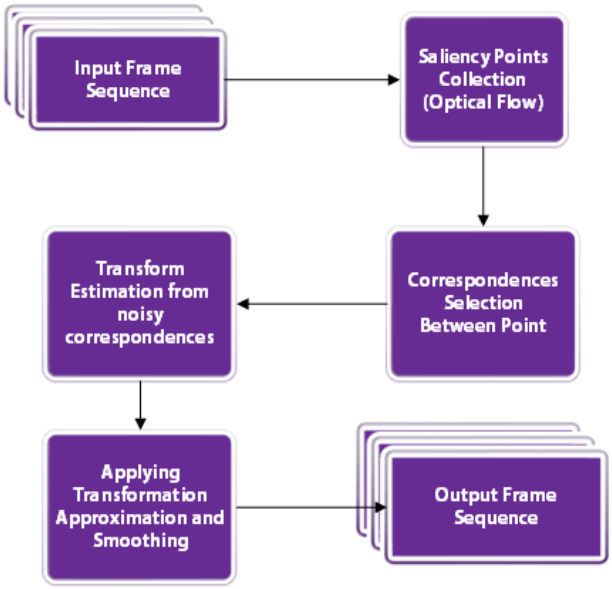

Figure 3. Video Stabilization Flowchart.

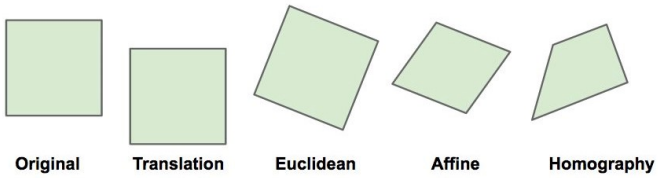

Figure 4. Motion Models.

3. Functional components breakdown

We will present here the methodological details of our solution.

3.1. Digital Video Stabilization

There are three main steps:

1. motion estimation: deriving the transformation parameters between two consecutive frames.
2. motion smoothing: filtering out unwanted motion.
3. image composition: reconstructing the stabilized video.

This method is explained by the flowchart in fig 3.

The model that we used is based on a two-dimensional motion model where we apply an Euclidean (a.k.a Similarity) transformation incorporating translation, rotation, and scaling.

As you can see in fig 4, in an Euclidean motion model, a square in an image can transform to any other square with a different location, size or rotation. It is more restrictive

than affine and homography transforms but is adequate for motion stabilization because the camera movement between successive frames of a video is usually small.

For this purpose, we use **Point Feature Matching**. This method involves detecting some feature points in two consecutive frames and tracking them. the tracked features allow us to estimate the motion between frames and apply its opposite to compensate for it.

1. Step 1: Set Input and Output Videos

2. Step 2: Read the first frame and convert it to grayscale: For video stabilization, we need to capture two frames of a video, estimate motion between the frames, and finally correct the motion.

3. Step 3: Find motion between frames: This is the most crucial part of the algorithm. We will iterate over all the frames, and find the motion between the current frame and the previous frame. We will explain it in details in section 3.1.1.

4. Step 4: Calculate smooth motion between frames: In the previous step, we estimated the motion between the frames and stored them in an array. We now need to find the trajectory of motion by cumulatively adding the differential motion estimated in the previous step. We will explain this step in details in section 3.1.2

5. Step 5: Apply smoothed camera motion to frames: All we need to do now is to loop over the frames and apply the transforms we just calculated. This step is further explained in section 3.1.3

3.1.1 Finding motion between frames

In this step, we loop over all the frames, track feature points and estimate the motion. It is not necessary to know the motion of each and every pixel. The Euclidean motion model requires that we know the motion of only 2 points in the two frames. However, in practice, it is a good idea to find the motion of 50-100 points, and then use them to robustly estimate the motion model.

Good Features to Track[2]: What points should we choose for tracking. Keeping in mind that tracking algorithms use a small patch around a point to track it, such tracking algorithms suffer from the aperture problem. So, smooth regions are bad for tracking and textured regions with lots of corners are good. Fortunately, OpenCV has a fast feature detector that detects features that are ideal for tracking. It is called **goodFeaturesToTrack**. For this video, You can check in fig 5 a set of points to track in one frame.

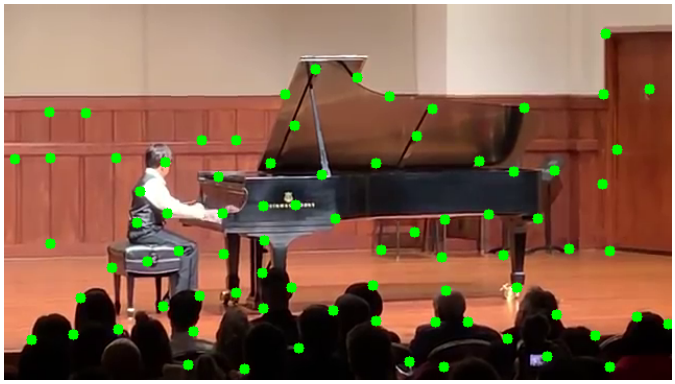

Figure 5. Example of Good Features To Track

Lucas-Kanade Optical Flow[1]: Once we have found good features in the previous frame, we can track them in the next frame using an algorithm called Lucas-Kanade Optical Flow named after the inventors of the algorithm.

It is implemented using the function **calcOpticalFlowPyrLK** in OpenCV. In the name **calcOpticalFlowPyrLK**, LK stands for Lucas-Kanade, and Pyr stands for the pyramid. An image pyramid in computer vision is used to process an image at different scales (resolutions).

calcOpticalFlowPyrLK may not be able to calculate the motion of all the points because of a variety of reasons. For example, the feature point in the current frame could get occluded by another object in the next frame. Fortunately, as you will see in the code below, the status flag in **calcOpticalFlowPyrLK** can be used to filter out these values.

Estimate Motion: To recap, we found good features to track in the previous frame. Then, we used optical flow to track the features. In other words, we found the location of the features in the current frame, and we already knew the location of the features in the previous frame. So we can use these two sets of points to find the rigid (Euclidean) transformation that maps the previous frame to the current frame.

Once we have estimated the motion, we can decompose it into x and y translation and θ rotation (angle). We store these values in an array so we can change them smoothly.

3.1.2 Calculating smooth motion between frames

Calculating trajectory: In this step, we will add up the motion between the frames to calculate the trajectory. Our ultimate goal is to smooth out this trajectory. In

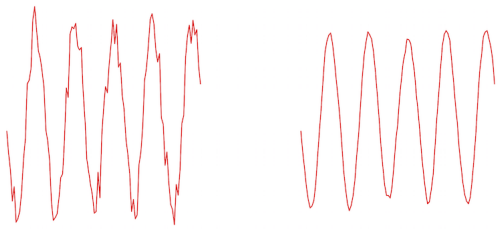

Figure 6. Before and after smoothing.

Python, it is easily achieved using cumsum (cumulative sum) in numpy.

smoothing the trajectory: in the previous step, we calculated the trajectory of motion. So we have three curves that show how the motion (x, y, and θ) changes over time.

In this step, we will show how to smooth these three curves.

The easiest way to smooth any curve is to use a moving average filter. As the name suggests, a moving average filter replaces the value of a function at the point by the average of its neighbors defined by a window. Let's look at an example.

Let's say we have stored a curve in an array c, so the points on the curve are $c[0] \dots c[n-1]$. Let f be the smooth curve we obtain by filtering c with a moving average filter of width 5.

The k^{th} element of this curve is calculated using:

$$f[k] = \frac{c[k-1]+c[k-1]+c[k]+c[k+1]+c[k+2]}{5}$$

As you can see in fig6, the values of the smooth curve are the values of the noisy curve averaged over a small window. The figure6 shows an example of the noisy curve on the left, smoothed using a box filter of size 5 on the right.

Calculating smooth transforms: So far we have obtained a smooth trajectory. In this step, we will use the smooth trajectory to obtain smooth transforms that can be applied to frames of the videos to stabilize it.

This is done by finding the difference between the smooth trajectory and the original trajectory and adding this difference back to the original transforms.

3.1.3 Applying smoothed camera motion to frames

If we have a motion specified as (x, y, θ), the corresponding transformation matrix is given by

$$T = \begin{bmatrix} \cos\theta & -\sin\theta & \text{x} \\ \sin\theta & \cos\theta & \text{y} \end{bmatrix}$$

When we stabilize a video, we may see some black boundary artifacts. This is expected because to stabilize the video, a frame may have to shrink in size.

We can mitigate the problem by scaling the video about its center by a small amount (e.g. 10%).

In order to implement this method, we use **getRotationMatrix2D** from OpenCV because it scales and rotates the image without moving the center of the image. All we need to do is call this function with 0 rotation and scale 1.1 (i.e. 10% upscale).

3.1.4 Stabilization results

You can check and example of an original shaky video and the stabilized output here

Pros

- This method is good against zooming(scaling) jitter in the video.
- It provides good stability against low-frequency motion (slower vibrations).
- This method has low memory consumption.

Cons

- This method performs poorly against high-frequency perturbations.
- If there is a heavy motion blur, feature tracking will fail and the results would not be optimal.

3.2. Image Averaging

3.2.1 Method explanation

The idea of simulating long exposures via averaging is hardly a new idea.

In fact, if you browse popular photography websites, you'll find tutorials on how to manually create these types of effects using your camera and tripod.

This brick of the project is a lot simpler than the previous one: We first specify the input and output file, then we initialize RGB channel averages which we will later merge into the final long exposure image. After that, we loop over all the frames of the video and calculate the running average for each color channel: The averaging computation is quite simple — we take the total number of frames times the channel-average, add the respective channel, and then divide that result by the floating point total number of frames (we add 1 to the total in the denominator because

Figure 7. Averaged waterfall video.

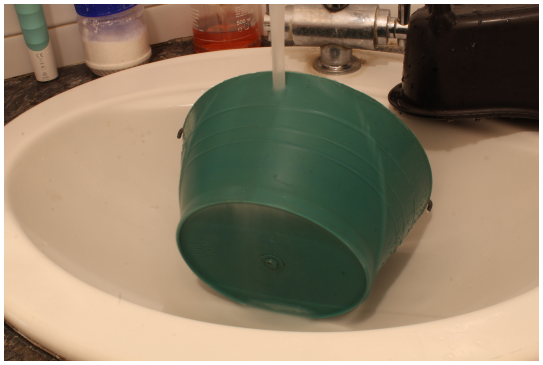

Figure 8. Classic method: professional camera and tripod

this is a fresh frame). We store the calculations in respective RGB channel average arrays. Finally, we increment the total number of frames, allowing us to maintain our running average. Once we have looped over all frames in the video file, we can merge the (average) channels into one image and write it to disk using **merge** from OpenCV.

3.3. Image Averaging results

Example: From this video, we get figure 7.

4. Application Results

Since I do not have any waterfall nearby, I have simulated a "waterfall" in my own sink. You can check in:

- fig **??**: The result using the classic photography method: with camera manual mode and tripod.
- fig **??**: The result using tripod and a recorded video
- fig **??**: the result using a video recorded with a hand-held camera.

As you can clearly notice, our alternative solution does not give a satisfying result.

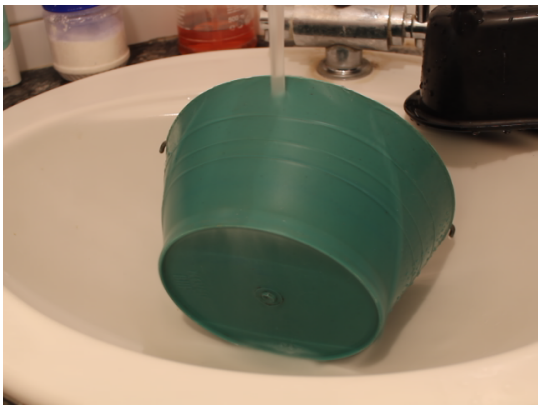

Figure 9. averaged video with tripod

Figure 10. hand-held video stabilized and averaged.

5. Conclusion

This Project offer a solution for amateur photographers who want to shoot stylish long exposure photos on the fly without using a camera and a tripod. Even though the video stabilization does a great work and gives an output that seems stable to the human observer, its result is not pixel perfect, there is always some shaking movement in the video. When averaging and merging the frames of the video, the blur caused by the pixel shaking becomes clearer and more remarkable and gives poor results, but still better than without stabilization.

Some next steps that we might suggest to enhance the performance of this solution is to focus further on video stabilization. We can choose another algorithm to detect another type of features in the frames.
Another idea could be using Deep Neural Network, LSTMs in particular since they have a memory, we think they might come handy in improving the video stabilization.

References

[1] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. *Carnegie-Mellon University*, 1981.

[2] C. Tomasi and J. Shi. Good features to track. *Cornell University*, 1993.