Removing Shakiness Using Mesh Flow

Kavya Chopra, 2021CS10081

May 9, 2023

1 Aim

In important fields like forensics, sometimes, the videos we have access to are extremely shaky, and it is difficult to discern the content due to this. Video shakiness may also be caused due to handheld camera movements, or due to the equipment being jittery. This completely ruins the user experience. In our project, we aim to solve this, in a computationally efficient manner.

2 Processing the Video

Knowing that a video is nothing but a collection of frames, which are discrete time image signals, strung together. With this knowledge, we consider the most elementary approach that comes to mind

2.1 Tracking pixel movement across frames

A pixel profile consists of motion vectors collected at the same pixel location. Tracking pixel movement in time can help us see where each object is going, and then we can try to average out the displacement vectors in order to smooth the video. The problem? There are too many pixels in the image. As a result, we try to extract those pixels/features where there is a lot of movement, namely, the corners.

2.2 Corner Detection

Here, we use the current SOTA algorithm for corner detection, aka the Shi-Tomasi Corner detection method. It returns the "good" corners we can track, that is, those where there is a sharp change in pixel values (it does so by tracking those pixels which have high gradients across them). Now, we have found the "nice" features, whose motion vectors we can calculate.

2.3 Optical Flow

We use the Lucas Kanade optical flow method for the same. The process starts by applying a Gaussian filter to the original image, which smooths the image and removes high-frequency

noise. The resulting image is then downsampled by a factor of two, which reduces its size and creates a new image with half the number of pixels in each dimension. This process is repeated on the downsampled image to create a sequence of images with progressively lower spatial resolution. This helps in the easier computation of optical flows, by going from the most downsampled image to the original one while having a ballpark of the parameters from the previous layer. It then computes the gradient of the image at each pixel, which represents the direction and magnitude of the changes in intensity in the x and y directions, followed by computing small window around each pixel and using the gradients in this window to estimate the change in position of the pixel in the x and y directions. This estimate is then refined using the neighboring pixels, by computing the error between the estimated flow and the actual flow in a larger window around the pixel. The algorithm then updates the optical flow estimate using the refined estimate, and repeats this process until convergence is achieved.

3 Generating vertex profiles

Now, due to the LK Optical Flow Algorithm as defined above, we have an array of 2D points representing the location of each point of the previous frame in the current frame. We now

3.1 Homography Generation

The homography matrix is a 3x3 matrix that describes the transformation between two views of the same scene, typically from different camera positions. The homography matrix maps points in one view to corresponding points in the other view, accounting for perspective distortions and camera movements. For this, we use the RANSAC method. The method involves selecting a random subset of the data, using this subset to estimate the model parameters, and then determining the set that fits the model best based on some predefined tolerance threshold in openCV. The model with the largest set is chosen, and that gives us our homography matrix.

3.2 Motion Mesh Generation

We divide the frame into rows and columns depending on a predefined mesh size (16 by 16 here). For each of the vertices in this mesh, we use the homography to map these mesh vertices to newer points for each mesh vertex in both X and Y directions. For each mesh vertex, we then iterate over the feature points we had, and if they have an euclidean distance less than a prespecified value, then we assume that the feature propagates its velocity vector to said mesh vertex. Over time, we get that many such feature points contribute to the motion of any given mesh vertex so to smooth out their motion, we take the median of all of these velocity vectors, and that's what the final velocity vector of each mesh is.

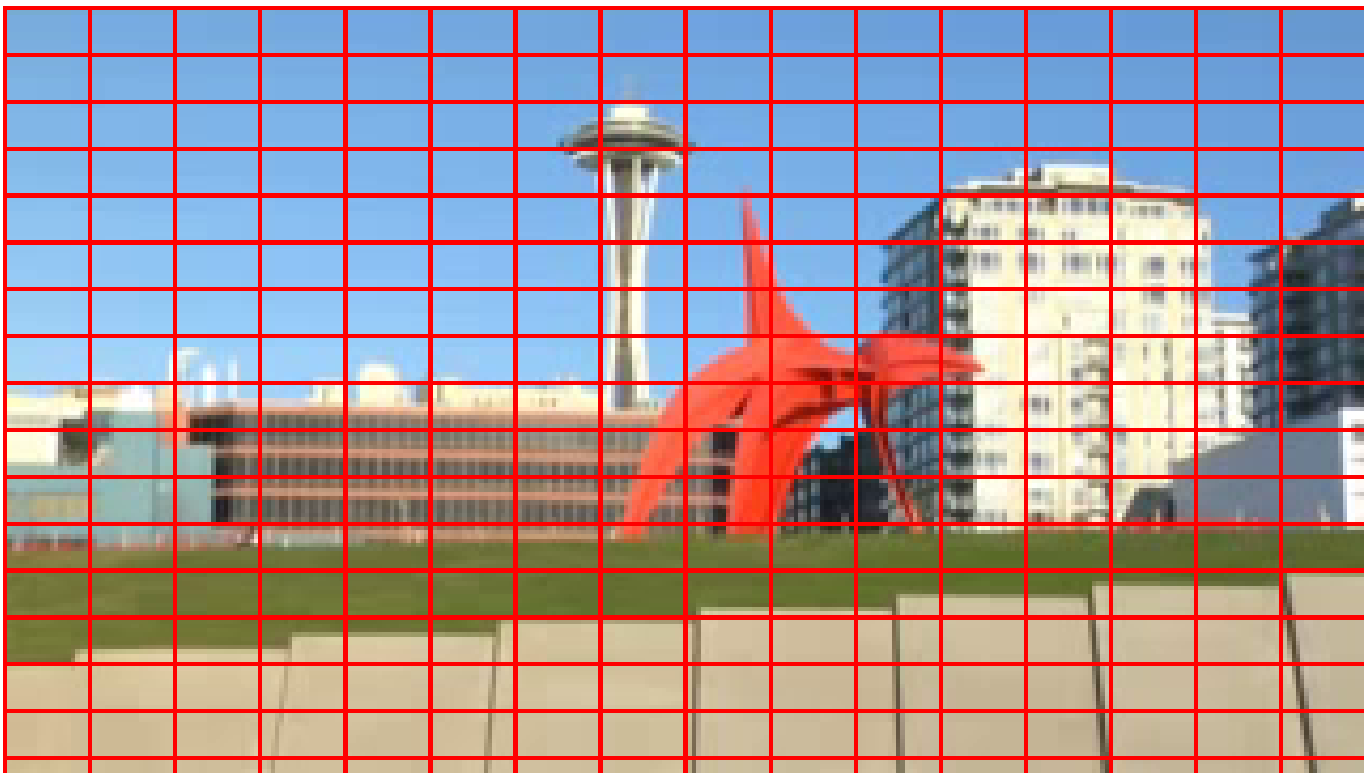
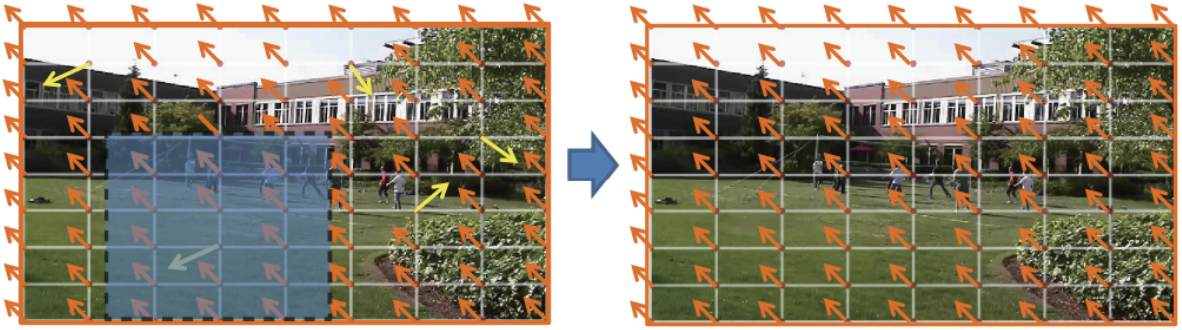


Fig. 3. A grid accumulates multiple motion vectors from several nearby features. We assign each grid a unique motion vector by applying median filter f_1 to the candidates. (Color figure online)

3.3 Outlier removal

We see that while most of the mesh vertices have velocity vectors pointing in the same direction, some of them don't, and so we apply yet another median filter to remove the noise. We now have our final motion meshes!



3.4 Vertex Profile

We then calculate the new vertex profiles by adding the current motion mesh to the previous vertex profiles, and append the new profiles to the existing ones. We now have an array of vertex profiles of each frame in the video!

4 Predictive Adaptive Path Smoothing

We can smooth all the vertex profiles for the smoothed motions. Firstly, we take the vertex profiles generated by the previous step, and perform the following steps

4.1 Gaussian Filters

The gauss function calculates the spatial Gaussian weights over a window size. If the difference between the current index and the index of the point in the window is greater than the window size, then the function returns 0. Otherwise, it returns the spatial Gaussian weight.

4.2 Optimising Camera Trajectory

We now use the online smoothing technique that the paper defines, by using an iterative approach to minimise the energy function and obtain the most optimal paths. The hyperparameters for this optimisation are

1. Temporal smoothness weight : a balancing weight which is set to 1
2. Spatial smoothness weight (determines how stabilised the video is)
3. Number of iterations of the optimiser (higher the number of iterations, the more time it takes, and the better the video quality is)

4. Buffer size: At the beginning, the buffer size is small, it increases gradually to a fixed size frame by frame and becomes a moving window that holds the latest frames and drops the oldest ones. Each time, we smooth the motions in the buffer by minimizing over the energy function

5. Window Size: Determines the size of the gaussian filter

We then code up a Jacobi solver to solve the given equation, and run it for 20 iterations per vertex per frame.

5 Re-warp frame, and generate video

We now finally map these mesh blocks to their displaced analogs, whilst mapping points between mesh patches in the process. Points not included in the mesh are mapped separately. We then use these new frames to generate the final video.

Figure 1: An example plot, with blue representing the pre-optimization vertex profile, and orange the post-optimization one for a vertex over the course of many frames. The x axis here is the number of frames, and the y axis is the magnitude of the velocity of the vertex profile

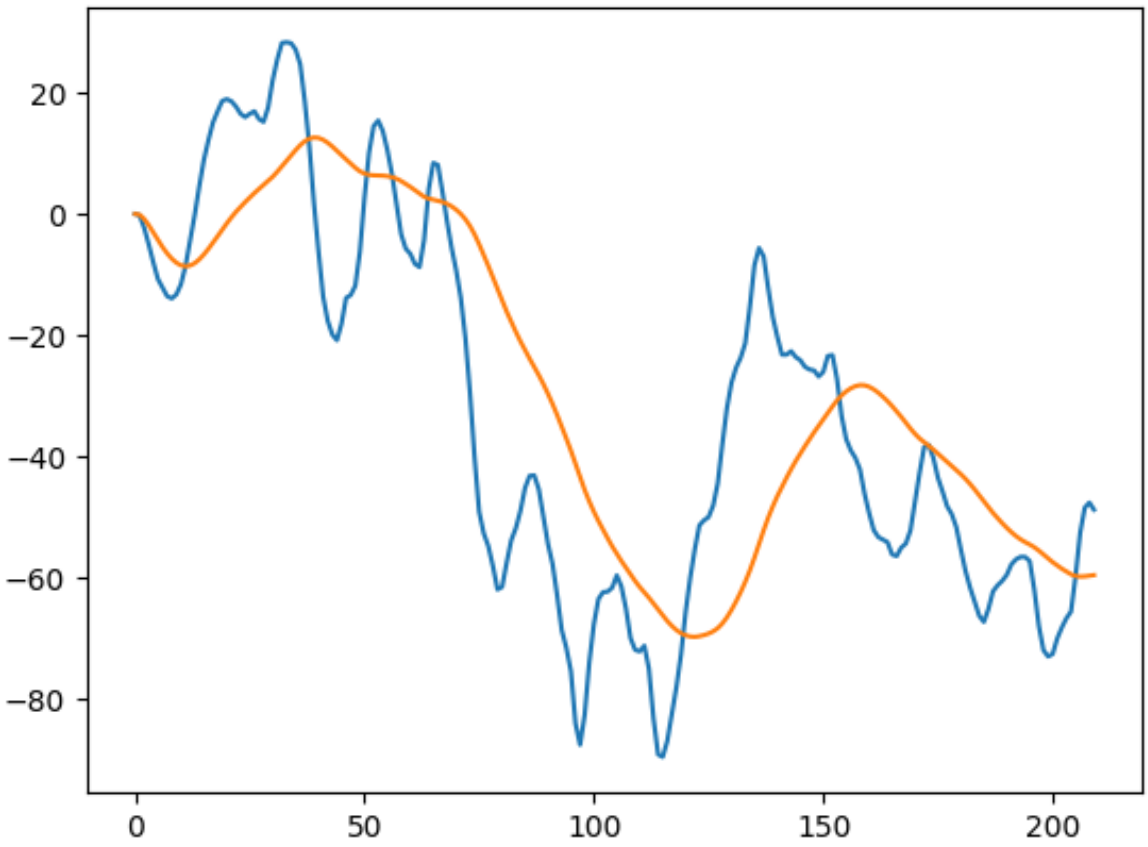
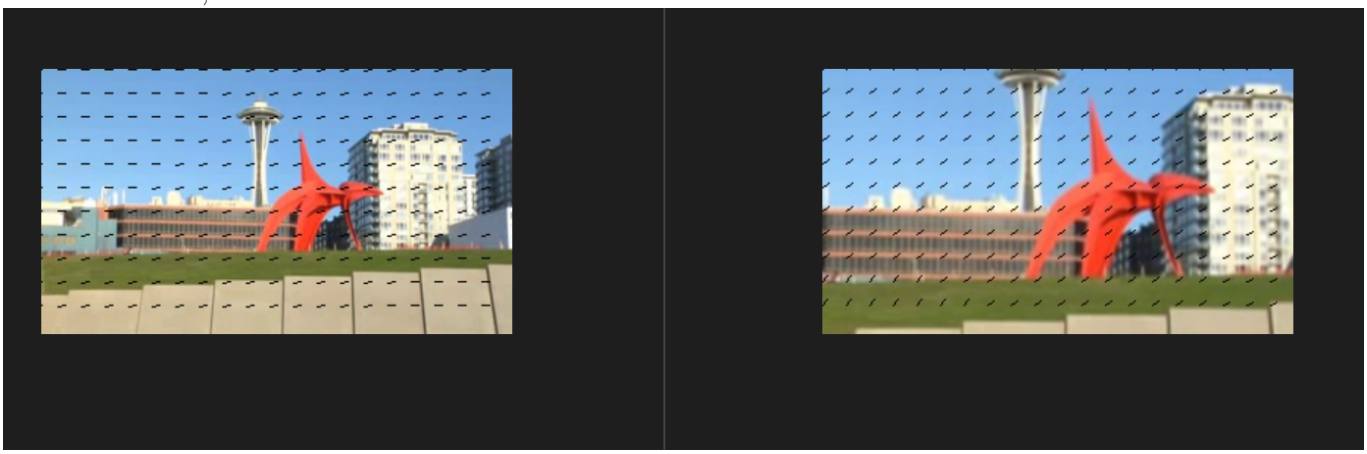


Figure 2: A direct comparison between the old and new motion vectors. The figure on the left shows the original motion mesh, and the one on the right shows a post-PAPS and frame warped motion mesh. Notice how the vectors in the right are mostly pointing towards the same direction, whereas those in the left are not



6 Notes

The research paper mentioned that we should sketch an ellipse around the mesh vertices, although I found that a circle gave better results, I have included the ellipse in the code for the sake of completeness.

7 Acknowledgements

The research paper upon which this project was based is http://www.liushuaicheng.org/eccv2016/meshflow. and some of the functions for video processing, and the idea of including progress bars to keep track (Since for higher resolution videos, the code may take upto 30 minutes to run), and using graphs and drawing motion vectors on the image to substantiate results, and the Jacobi solver were taken from https://github.com/sudheerachary/Mesh-Flow-Video-Stabilization/tree/master. The rest of the logic, however, is my own.