Learn OpenCV

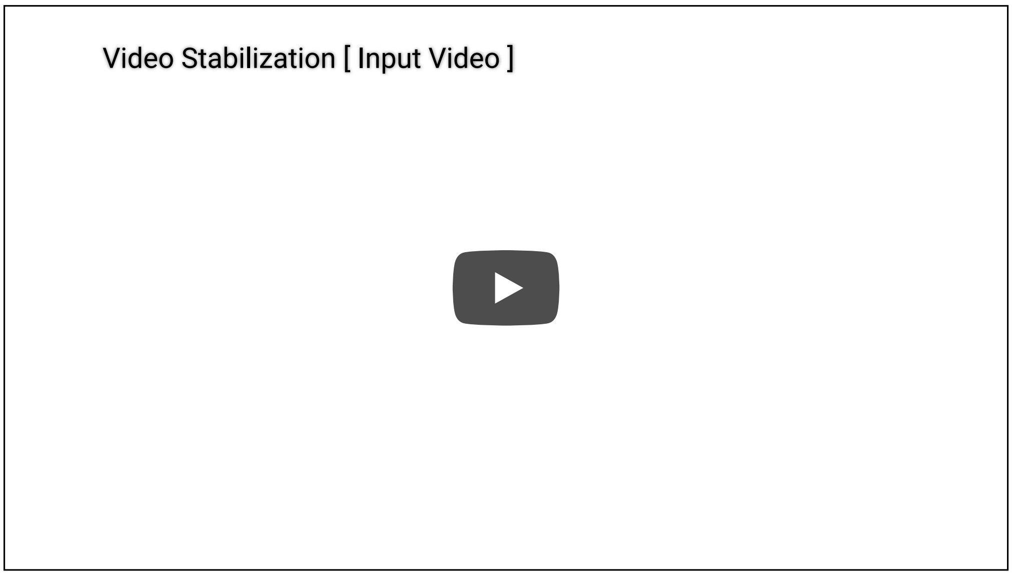
Video Stabilization Using Point Feature Matching in OpenCV

Abhishek Singh Thakur (https://www.learnopencv.com/author/abhi-una12/)

JANUARY 22, 2019

In this post, we will learn how to implement a simple Video Stabilizer using a technique called Point Feature Matching in OpenCV library. We will discuss the algorithm and share the code(in python) to design a simple stabilizer using this method in OpenCV. This post's code is inspired by work presented by Nghia Ho here (http://nghiaho.com/?p=2093) and the post (https://abhitronix.github.io/2018/11/30/humanoid-AEAM-3/) from my website.

Video Stabilization



Example of Low-frequency camera motion in video

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).

Applications of Video Stabilization

The need for video stabilization spans many domains.

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.

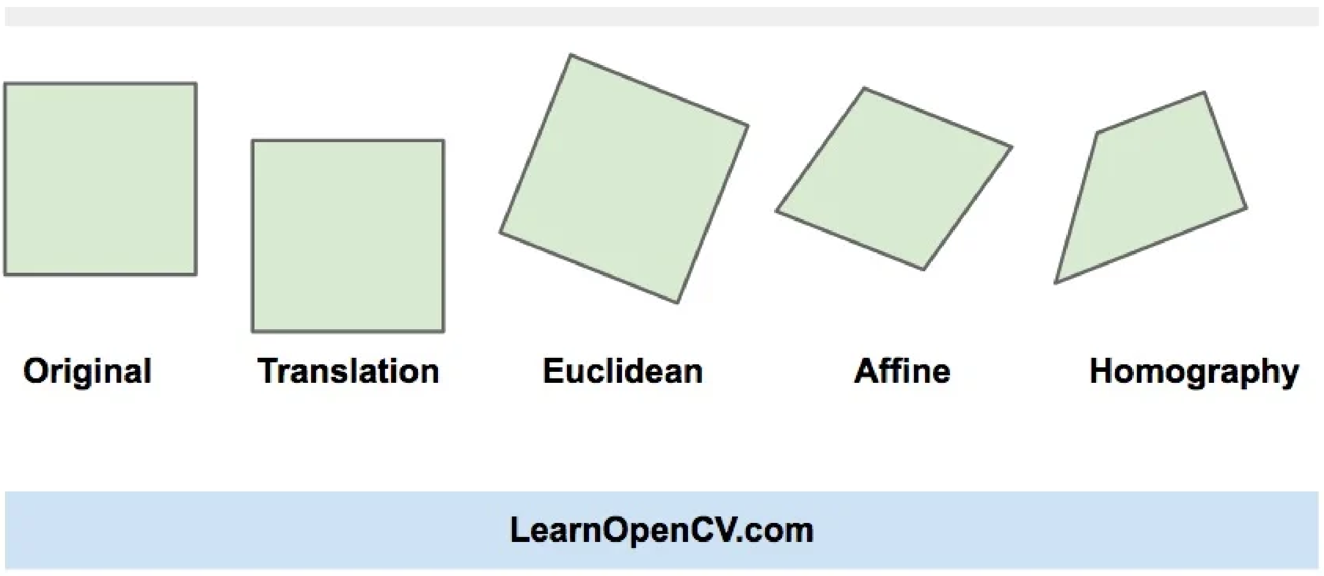
Similarly, in **military applications**, videos captured by aerial vehicles on a reconnaissance flight need to be stabilized for localization, navigation, target tracking, etc. The same applies to **robotic applications**.

Different Approaches to Video Stabilization

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:** This method does not require special sensors for estimating camera motion. There are three main steps — 1) motion estimation 2) motion smoothing, and 3) image composition. The transformation parameters between two consecutive frames are derived in the first stage. The second stage filters out unwanted motion and in the last stage the stabilized video is reconstructed.

We will learn a fast and robust implementation of a digital video stabilization algorithm in this post. It is based on a two-dimensional motion model where we apply a **Euclidean (a.k.a Similarity) transformation** incorporating translation, rotation, and scaling.



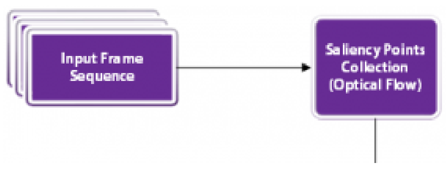
(https://www.learnopencv.com/wp-content/uploads/2015/07/motion-models.jpg)

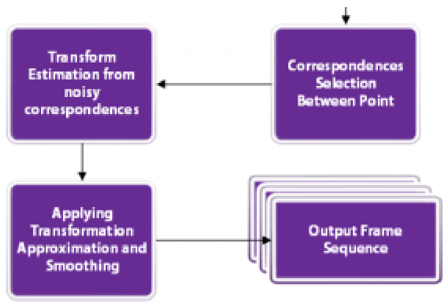
As you can see in the image above, in a Euclidean motion model, a square in an image can transform to any other square with a different location, size or rotation. It is more restrictive than affine and homography transforms but is adequate for motion stabilization because the camera movement between successive frames of a video is usually small.

Video Stabilization Using Point Feature Matching

This method involves tracking a few feature points between two consecutive frames. The tracked features allow us to estimate the motion between frames and compensate for it.

The flowchart below shows the basic steps.




Transform Estimation from noisy correspondences
Correspondences Selection Between Point
Applying Transformation Approximation and Smoothing
Output Frame Sequence

(https://www.learnopencv.com/wp-content/uploads/2019/01/AEAM-3.png)

Block Diagram

Let's go over the steps.

Download Code To easily follow along this tutorial, please download code by clicking on the button below. It's FREE!

DOWNLOAD CODE (HTTPS://BIGVISIONLLC.LEADPAGES.NET/LEADBOX/143948B73F72A2%3A173C9390C346DC/5649050225344512/)

Step 1 : Set Input and Output Videos

First, let's complete the setup for reading the input video and writing the output video. The comments in the code explain every line.

Python

```python
# Import numpy and OpenCV
import numpy as np
import cv2

# Read input video
cap = cv2.VideoCapture('video.mp4')&nbsp;

# Get frame count
n_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))

# Get width and height of video stream
w = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
h = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

# Define the codec for output video
fourcc = cv2.VideoWriter_fourcc(*'MJPG')

# Set up output video
out = cv2.VideoWriter('video_out.mp4', fourcc, fps, (w, h))
```

C++

```cpp
// Read input video
VideoCapture cap("video.mp4");
```

```
// Get frame count
int n_frames = int(cap.get(CAP_PROP_FRAME_COUNT));

// Get width and height of video stream
int w = int(cap.get(CAP_PROP_FRAME_WIDTH));
int h = int(cap.get(CAP_PROP_FRAME_HEIGHT));

// Get frames per second (fps)
double fps = cap.get(CV_CAP_PROP_FPS);

// Set up output video
VideoWriter out("video_out.avi", CV_FOURCC('M','J','P','G'), fps, Size(2 * w, h));
```

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.

Python

```python
# Read first frame
_, prev = cap.read()

# Convert frame to grayscale
prev_gray = cv2.cvtColor(prev, cv2.COLOR_BGR2GRAY)
```

C++

```cpp
// Define variable for storing frames
Mat curr, curr_gray;
Mat prev, prev_gray;

// Read first frame
cap &gt;&gt; prev;

// Convert frame to grayscale
cvtColor(prev, prev_gray, COLOR_BGR2GRAY);
```

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. 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.

3.1 Good Features to Track

The question now is what points should we choose for tracking. Keep in mind that tracking algorithms use a small patch around a point to track it. Such tracking algorithms suffer from the **aperture problem** as explained in the video below

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** (no kidding!).

3.2 Lucas-Kanade Optical Flow

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.

3.3 Estimate Motion

To recap, in step 3.1, we found good features to track in the previous frame. In step 3.2, 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. This is done using the function **estimateRigidTransform**.

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

The code below goes over steps 3.1 to 3.3. Make sure to read the comments in the code to follow along.

Python

```python
# Pre-define transformation-store array
transforms = np.zeros((n_frames-1, 3), np.float32)

for i in range(n_frames-2):
  # Detect feature points in previous frame
  prev_pts = cv2.goodFeaturesToTrack(prev_gray,
                                     maxCorners=200,
                                     qualityLevel=0.01,
                                     minDistance=30,
                                     blockSize=3)

  # Read next frame
  success, curr = cap.read()
  if not success:
    break

  # Convert to grayscale
  curr_gray = cv2.cvtColor(curr, cv2.COLOR_BGR2GRAY)

  # Calculate optical flow (i.e. track feature points)
  curr_pts, status, err = cv2.calcOpticalFlowPyrLK(prev_gray, curr_gray, prev_pts, None)

  # Sanity check
  assert prev_pts.shape == curr_pts.shape

  # Filter only valid points
  idx = np.where(status==1)[0]
  prev_pts = prev_pts[idx]
  curr_pts = curr_pts[idx]

  #Find transformation matrix
  m = cv2.estimateRigidTransform(prev_pts, curr_pts, fullAffine=False) #will only work with OpenCV-3 or
less

  # Extract traslation
  dx = m[0,2]
  dy = m[1,2]

  # Extract rotation angle
  da = np.arctan2(m[1,0], m[0,0])

  # Store transformation
  transforms[i] = [dx,dy,da]

  # Move to next frame
  prev_gray = curr_gray

  print("Frame: " + str(i) +  "/" + str(n_frames) + " -  Tracked points : " + str(len(prev_pts)))
```

C++

In the C++ implementation, we first define a few classes that will help us store the estimated motion vectors. The **TransformParam** class below stores the motion information (dx — motion in x, dy — motion in y, and da — change in angle), and provides a method **getTransform** to convert this motion into a transformation matrix.

```
struct TransformParam
{
  TransformParam() {}
  TransformParam(double _dx, double _dy, double _da)
  {
      dx = _dx;
      dy = _dy;
      da = _da;
  }

  double dx;
  double dy;
  double da; // angle

  void getTransform(Mat &amp;T)
  {
    // Reconstruct transformation matrix accordingly to new values
    T.at&lt;double&gt;(0,0) = cos(da);
    T.at&lt;double&gt;(0,1) = -sin(da);
    T.at&lt;double&gt;(1,0) = sin(da);
    T.at&lt;double&gt;(1,1) = cos(da);

    T.at&lt;double&gt;(0,2) = dx;
    T.at&lt;double&gt;(1,2) = dy;
  }
};
```

We loop over the frames and perform steps 3.1 to 3.3, in the code below.

```
// Pre-define transformation-store array
  vector &lt;TransformParam&gt; transforms;

  //

  Mat last_T;

  for(int i = 1; i &lt; n_frames-1; i++)
  {
    // Vector from previous and current feature points
    vector &lt;Point2f&gt; prev_pts, curr_pts;
    // Detect features in previous frame
    goodFeaturesToTrack(prev_gray, prev_pts, 200, 0.01, 30);

    // Read next frame
    bool success = cap.read(curr);
    if(!success) break;

    // Convert to grayscale
    cvtColor(curr, curr_gray, COLOR_BGR2GRAY);

    // Calculate optical flow (i.e. track feature points)
    vector &lt;uchar&gt; status;
    vector &lt;float&gt; err;
    calcOpticalFlowPyrLK(prev_gray, curr_gray, prev_pts, curr_pts, status, err);

    // Filter only valid points
    auto prev_it = prev_pts.begin();
    auto curr_it = curr_pts.begin();
    for(size_t k = 0; k &lt; status.size(); k++)
    {
        if(status[k])
        {
          prev_it++;
          curr_it++;
        }
        else
        {
          prev_it = prev_pts.erase(prev_it);
          curr_it = curr_pts.erase(curr_it);
        }
    }


    // Find transformation matrix
    Mat T = estimateRigidTransform(prev_pts, curr_pts, false);

    // In rare cases no transform is found.
    // We'll just use the last known good transform.
    if(T.data == NULL) last_T.copyTo(T);
    T.copyTo(last_T);

    // Extract traslation
    double dx = T.at&lt;double&gt;(0,2);
    double dy = T.at&lt;double&gt;(1,2);

    // Extract rotation angle
    double da = atan2(T.at&lt;double&gt;(1,0), T.at&lt;double&gt;(0,0));

    // Store transformation
    transforms.push_back(TransformParam(dx, dy, da));

    // Move to next frame
    curr_gray.copyTo(prev_gray);

    cout &lt;&lt; "Frame: " &lt;&lt; i &lt;&lt; "/" &lt;&lt; n_frames &lt;&lt; " -  Tracked points : " &lt;&lt; prev_pts.size() &lt;&lt; endl;
  }
```

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.

Step 4.1 : Calculate 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.

Python

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

```python
# Compute trajectory using cumulative sum of transformations
trajectory = np.cumsum(transforms, axis=0)
```

C++

In C++, we define a class called **Trajectory** to store the cumulative sum of the transformation parameters.

```cpp
struct Trajectory
{
    Trajectory() {}
    Trajectory(double _x, double _y, double _a) {
        x = _x;
        y = _y;
        a = _a;
    }

    double x;
    double y;
    double a; // angle
};
```

We also, define a function **cumsum** that takes in a vector of **TransformParams** and returns trajectory by performing the cumulative sum of differential motion dx, dy, and da (angle).

```cpp
vector&lt;Trajectory&gt; cumsum(vector&lt;TransformParam&gt; &amp;transforms)
{
  vector &lt;Trajectory&gt; trajectory; // trajectory at all frames
  // Accumulated frame to frame transform
  double a = 0;
  double x = 0;
  double y = 0;

  for(size_t i=0; i &lt; transforms.size(); i++)
  {
      x += transforms[i].dx;
      y += transforms[i].dy;
      a += transforms[i].da;

      trajectory.push_back(Trajectory(x,y,a));

  }

  return trajectory;
}
```

Step 4.2 : Calculate smooth trajectory

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

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

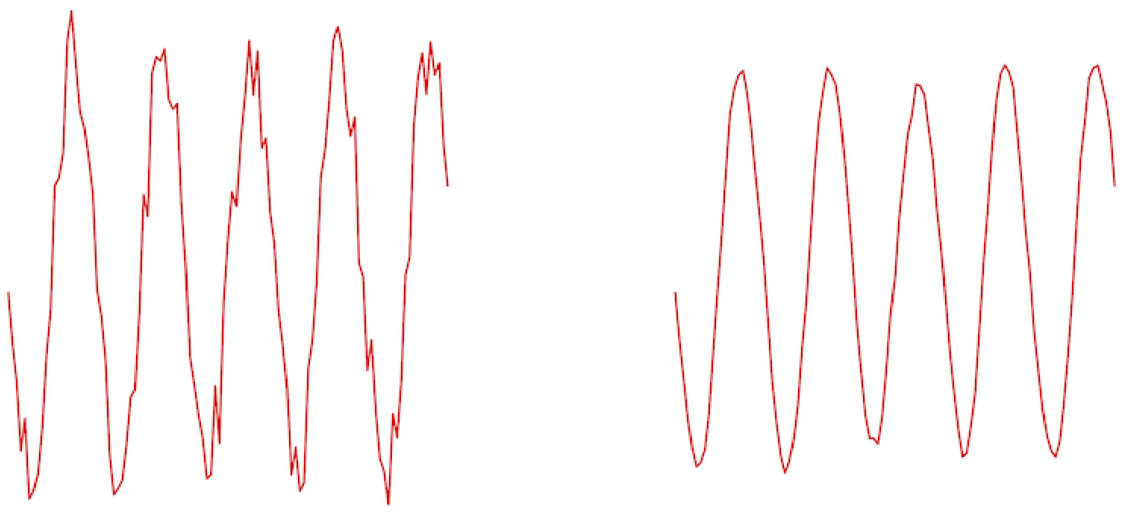
The easiest way to smooth any curve is to use a **moving average filter**. As the name suggests, a moving average filter replaces the value of a function at the point by the average of its neighbors defined by a window. Let's look at an example.

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

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

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



(https://www.learnopencv.com/wp-content/uploads/2019/01/box-filtering.png)

Python

In the Python implementation, we define a moving average filter that takes in any curve (i.e. a 1-D of numbers) as an input and returns the smoothed version of the curve.

```python
def movingAverage(curve, radius):
  window_size = 2 * radius + 1
  # Define the filter
  f = np.ones(window_size)/window_size
  # Add padding to the boundaries
  curve_pad = np.lib.pad(curve, (radius, radius), 'edge')
  # Apply convolution
  curve_smoothed = np.convolve(curve_pad, f, mode='same')
  # Remove padding
  curve_smoothed = curve_smoothed[radius:-radius]
  # return smoothed curve
  return curve_smoothed
```

We also define a function that takes in the trajectory and performs smoothing on the three components.

```python
def smooth(trajectory):
  smoothed_trajectory = np.copy(trajectory)
  # Filter the x, y and angle curves
  for i in range(3):
    smoothed_trajectory[:,i] = movingAverage(trajectory[:,i], radius=SMOOTHING_RADIUS)

  return smoothed_trajectory
```

And, here is the final usage.

```python
# Compute trajectory using cumulative sum of transformations
trajectory = np.cumsum(transforms, axis=0)
```

C++

In the C++ version, we define a function called **smooth**, that calculates the smoothed moving average trajectory.

```cpp
vector &lt;Trajectory&gt; smooth(vector &lt;Trajectory&gt;&amp; trajectory, int radius)
{
  vector &lt;Trajectory&gt; smoothed_trajectory;
  for(size_t i=0; i &lt; trajectory.size(); i++) {
      double sum_x = 0;
      double sum_y = 0;
      double sum_a = 0;
      int count = 0;

      for(int j=-radius; j &lt;= radius; j++) { if(i+j &gt;= 0 &amp;&amp; i+j &lt; trajectory.size()) {
              sum_x += trajectory[i+j].x;
              sum_y += trajectory[i+j].y;
              sum_a += trajectory[i+j].a;

              count++;
          }
      }

      double avg_a = sum_a / count;
      double avg_x = sum_x / count;
      double avg_y = sum_y / count;

      smoothed_trajectory.push_back(Trajectory(avg_x, avg_y, avg_a));
  }

  return smoothed_trajectory;
}
```

And we use it in the main function.

```cpp
// Smooth trajectory using moving average filter
  vector &lt;Trajectory&gt; smoothed_trajectory = smooth(trajectory, SMOOTHING_RADIUS);
```

Step 4.3 : Calculate 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.

Python

```python
# Calculate difference in smoothed_trajectory and trajectory
difference = smoothed_trajectory - trajectory
```

```
# Calculate newer transformation array
transforms_smooth = transforms + difference
```

C++

```
vector &lt;TransformParam&gt; transforms_smooth;

for(size_t i=0; i &lt; transforms.size(); i++)
{
  // Calculate difference in smoothed_trajectory and trajectory
  double diff_x = smoothed_trajectory[i].x - trajectory[i].x;
  double diff_y = smoothed_trajectory[i].y - trajectory[i].y;
  double diff_a = smoothed_trajectory[i].a - trajectory[i].a;

  // Calculate newer transformation array
  double dx = transforms[i].dx + diff_x;
  double dy = transforms[i].dy + diff_y;
  double da = transforms[i].da + diff_a;

  transforms_smooth.push_back(TransformParam(dx, dy, da));
}
```

Step 5: Apply smoothed camera motion to frames

We are almost done. All we need to do now is to loop over the frames and apply the transforms we just calculated.

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

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

Read the comments in the code to follow along.

Python

```
# Reset stream to first frame
cap.set(cv2.CAP_PROP_POS_FRAMES, 0)

# Write n_frames-1 transformed frames
for i in range(n_frames-2):
  # Read next frame
  success, frame = cap.read()
  if not success:
    break

  # Extract transformations from the new transformation array
  dx = transforms_smooth[i,0]
  dy = transforms_smooth[i,1]
  da = transforms_smooth[i,2]

  # Reconstruct transformation matrix accordingly to new values
  m = np.zeros((2,3), np.float32)
  m[0,0] = np.cos(da)
  m[0,1] = -np.sin(da)
  m[1,0] = np.sin(da)
  m[1,1] = np.cos(da)
  m[0,2] = dx
  m[1,2] = dy

  # Apply affine wrapping to the given frame
  frame_stabilized = cv2.warpAffine(frame, m, (w,h))

  # Fix border artifacts
  frame_stabilized = fixBorder(frame_stabilized)
```

```
    # Write the frame to the file
    frame_out = cv2.hconcat([frame, frame_stabilized])

    # If the image is too big, resize it.
    if(frame_out.shape[1] &gt; 1920):
      frame_out = cv2.resize(frame_out, (frame_out.shape[1]/2, frame_out.shape[0]/2));

    cv2.imshow("Before and After", frame_out)
    cv2.waitKey(10)
    out.write(frame_out)
```

C++

```
cap.set(CV_CAP_PROP_POS_FRAMES, 1);
Mat T(2,3,CV_64F);
Mat frame, frame_stabilized, frame_out;


for( int i = 0; i &lt; n_frames-1; i++) { bool success = cap.read(frame); if(!success) break; // Extract transform from translation and rotation angle. transforms_smooth[i].getTransform(T); // Apply affine wrapping to the given frame warpAffine(frame, frame_stabilized, T, frame.size()); // Scale image to remove black border artifact fixBorder(frame_stabilized); // Now draw the original and stabilised side by side for coolness hconcat(frame, frame_stabilized, frame_out); // If the image is too big, resize it. if(frame_out.cols &gt; 1920)
  {
      resize(frame_out, frame_out, Size(frame_out.cols/2, frame_out.rows/2));
  }

  imshow("Before and After", frame_out);
  out.write(frame_out);
  waitKey(10);
}
```

Step 5.1 : Fix border artifacts

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. 4%).

The function **fixBorder** below shows the implementation. We use **getRotationMatrix2D** 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 1.04 (i.e. 4% upscale).

Python

```
def fixBorder(frame):
  s = frame.shape
  # Scale the image 4% without moving the center
  T = cv2.getRotationMatrix2D((s[1]/2, s[0]/2), 0, 1.04)
  frame = cv2.warpAffine(frame, T, (s[1], s[0]))
  return frame
```

C++

```
void fixBorder(Mat &amp;frame_stabilized)
{

  Mat T = getRotationMatrix2D(Point2f(frame_stabilized.cols/2, frame_stabilized.rows/2), 0, 1.04);
  warpAffine(frame_stabilized, frame_stabilized, T, frame_stabilized.size());
}
```

Results

Left: Input video. **Right:** Stabilized video.

The result of the stabilization code we have shared is shown above. Our objective was to reduce the motion significantly, but not to eliminate it completely.

We leave it to the reader to think of a modification of the code that will eliminate motion between frames completely. What could be the side effects if you try to eliminate all camera motion?

The current method only works for a fixed length video and not with a real-time feed. We have to modify this method heavily to attain real-time video output which is out of the scope for this post but it is achievable, more information can be found here (https://abhitronix.github.io/2018/11/30/humanoid-AEAM-3/).

Pros and Cons

Pros

1. This method provides good stability against low-frequency motion (*slower vibrations*).
2. This method has low memory consumption thereby ideal for Embedded devices(*like Raspberry Pi*).
3. This method is good against zooming(scaling) jitter in the video.

Cons

1. This method performs poorly against high-frequency perturbations.

2. If there is a heavy motion blur, feature tracking will fail and the results would not be optimal.
3. This method is also not good with Rolling Shutter distortion.

Subscribe & Download Code

If you liked this article and would like to download code (C++ and Python) and example images used in this post, please subscribe (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) to our newsletter. You will also receive a free Computer Vision Resource (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.

SUBSCRIBE NOW (HTTPS://BIGVISIONLLC.LEADPAGES.NET/LEADBOX/143948B73F72A2%3A173C9390C346DC/5649050225344512/)

References:

1. Example video and Code reference from Nghia Ho's post (http://nghiaho.com/uploads/code/videostab.cpp)
2. Various References, data, and image from my website (https://abhitronix.github.io)

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

hbbs80 • 7 months ago

Thank you for this excellent text. With regards to your question at the end, what to do if we want to remove motion completely (The downside would of course be that there will be large black borders...): we could increase smoothing radius to the number of frames (or above). Is there a better way?

• Reply • Share ›

gtoguy01 • 7 months ago

Extra characters in the C++ code above. Last section.
void fixBorder(Mat &frame_stabilized)
leave out "amp;"

• Reply • Share ›

Giovanni • 7 months ago • edited

If you are working with OpenCV 4 or later you should replace

`m = cv2.estimateRigidTransform(prev_pts, curr_pts, fullAffine=False)`

with

`m, inliers = cv2.estimateAffinePartial2D(prev_pts, curr_pts)`

• Reply • Share ›

抹香 • 10 months ago

How to crop black boundary. I have using fixBorder method. But not affected.

• Reply • Share ›

Alvin • a year ago

What are suitable values for SMOOTHING_RADIUS?
Thanks

• Reply • Share ›

Giovanni → Alvin • 7 months ago

in the example code SMOOTHING_RADIUS is 50

• Reply • Share ›

Walid Aly • a year ago

Thanks a lot
If we skip smoothing, can we use the information from frame to frame and apply this on real time?

• Reply • Share ›

Masque du Furet → Walid Aly • a year ago

Excuse me, but before attempting to do reaal time, you should record videos, and try the algorithms who are given (for some problems, it is likely to work; what happens if

a) there are no interesting features to track -smooth/uniform images)? doe program break? does it handle nicely?

b) there are too many interesting features to track , eating too much CPU -on a RPi)

c) you try to compare two algorithms on the same dataset, near your problem?)

========
sum_x = trajectory[i].x;
for(int j=-radius; j <0; j++) {
if(i+j >= 0 && i+j < trajectory.size()) {
/* alpha near 1 : forgets fast; alpha near zero : forgets slowly -low pass

sum_x = (1 - alpha) * sum_x + alpha * trajectory[i+j].x;

```
}
}
```

======
This piece of untested c++ code (replaces a median average) is likely to mimic a real time behavior (one forgets, at a given rate, past observations; only past observations are used...)

But you should begin with recorded data, with a behavior like what you think your problem is .

• Reply • Share ›

Satya Mallick Mod Masque du Furet • a year ago

Well, if we remove smoothing, we can only stabilize it to the first frame. Smoothing is the thing that gives it a stabilized look.

• Reply • Share ›

Masque du Furet • a year ago

Well, the pianist is very good... and should be congratulated.
Something worries me :
phones have partial information on tilt (but likely to be unusable)
a Rapsberry camera (any web cam, but RPi is simple) can be glued with accelerometers (give a tilt information) and gysroscopes (give a noisy angular velocity information) and magnetometer (another angle position info). Software is very easy to get this information, AFAIK (tried on arduino). This makes mechanical video stabilization easy (and eats little CPU) on RPI and likes.
But can one put together mechanical and video stabilization (less noise)? Is there any theory?
BTW, mechanical stabilization, from your post, is likely to work better on smooth images (dull ones)...

• Reply • Share ›

Satya Mallick Mod Masque du Furet • a year ago

Thanks!

Yes, if you have access to the camera at the time of recording, mechanical stabilization is definitely an option. A lot of cell phones already do optical stabilization to some degree.

If camera orientation and translation information is available through mechanical means, the math to smooth the trajectory is well known. Basically, instead of using features to indirectly measure the motion, we will directly know it from the camera motion.

3 • Reply • Share ›

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Do Not Sell My Data**