Assignment 2 - Global Motion Estimation

Author: Lorenzo Aldrighetti ID: 169301

University of Trento

Abstract. With this paper I explain how can be possible to estimate the global motion of a camera using the OpenCV library on c++ code.The tests are made with YouTube videos taken by smartphone or tablet in order to have a video that is shaky.

1 Introduction

The object of this assigment is estimate the global motion of a amatorial video using the OpenCV library.

The tests are do downloading some videos from YouTube where they show a goal during soccer match. The videos must be taken with smarthphone or tablet in order to have a video that is shaky. The minimum length must be of one minute. Next I export an output video with the original and an arrow that indicate the direction of the identified motion of the camera.

I compute many test with the same video at different resolution in order to set some parameters and test the qauality of the alghoritm. Next I compute the algorithm with another different video for ensure that all code work fine.

2 The Videos

I download two videos from Youtube that soddisfy the request. One (Last Minute Goal Sergio Ramos Champions Final) was downloaded at five different quality in term of resolution and with a variable bitrate at 30 frame-per-second setted automatically from YouTube service:

1. Resolution: 1920x1080, Overall bit rate: 4249 Kbps
2. Resolution: 1280x720, Overall bit rate: 3194 Kbps
3. Resolution: 854x480, Overall bit rate: 1236 Kbps
4. Resolution: 640x360, Overall bit rate: 852 Kbps
5. Resolution: 426x240, Overall bit rate: 376 Kbps

These ones are used for testing the code, the second video is used only for confirm the quality of the code. This was downloaded only with a resolition of 1280x720 at 3195 Kbps overall bit rate. In the next section i call the video with the respective number chosen before.

3 The code

The original code is taken from the website of Nghia Ho. Originally this project doing a video stabilization using the optical flow method implemented with Lukas Kanade Pyramidal algorithm using a good features tecnique to extract the point more significant.

I adapt the code in order to complete the requirement. Essentially I remove the last part of the code but the first is the same becouse the algoritm can be recyled.

Now I explain the main step of the code.

First of all the code import the video stored in the local workspace. Next i create some useful viariable and some matrix that will contain a set of frame elaborated to do some kind of operation. The next step is convert the original video in to a grayscale video in order to operate with the algorithm that find a set of good feature points, and next, the code store all points in a specific vector. This function is *goodFeaturesToTrack*.

3.1 GoodFeaturesToTrack function

This function essentially determines strong corners on an image or in a specific region. First of all the function calculate the corner quality using the cornerMinEigenVal() function. Next performs a non-maximum suppression (the local maximums in 3x3 neighborhood are retained) and the corner with the minimal eigenvalue less than a defined value are rejected. Now the remaining corners are shorted by the quality in descending order and finally, throws away each corner for which there is a strong corner at defined minimun distance.

It is implemented in this way:

> *void* ***goodFeaturesToTrack****(InputArray* ***image****, OutputArray* ***corners****, int* ***maxCorners****, double* ***qualityLevel****, double* ***minDistance****, InputArray* ***mask****=noArray(), int* ***blockSize****=3, bool* ***useHarrisDetector****=false, double* ***k****=0.04)*

where each parameters are:

- **image**: input image
- **corners**: output vector that will contain the coordinate of points found.
- **maxCorners**: indicate the max number of corner that the function return
- **qualityLevel**: is the minimal accepted quality of each point. The parameter value is multiplied by the best corner quality measure, which is the minimal eigenvalue, and the corners with quality measure less than the product, are rejected.
- **minDistance**: indicate the minimum Euclidean distance between the returned points
- **mask**: is optional and define a region of interest, if it's specified
- **blockSize**: size of an average block for computing a derivative covariation matrix over each pixel neighborhood. Is setted by default

- **useHarrisDetector**: parameter indicating whether to use a Harris detector, if not specified is set to false.
- **k**: indicate free parameter of the Harris detector only if it's used. Is setted by default

In my case I set the max number of good feature point to 200. This value is chose doing some test, in fact for the video **5** the overall number of good feature points detected are about 65 and the algorithm work quite well, for the video **4** are about 125 and from video **3** the number is about 210 and the algorithm, in this case, work very well. In conclusion i decide to extract a sufficent points for work well with video of major quality and resolution.

This value depends mainly from the *minDistance* parameter setted, in my case, to 30. Infact if I reduce the number of the half, in the case of video **1**, I obtain more than 200 points. This choice was made considering that in our scene there are lot of moving object becouse the video was recorded from fan stand, and in the scene compare a lot of moving arms and heads, these imply that the algorithm detect a movement of feature point but the camera is steady. Another consideration is do by the fact that modern camera and smartphone capture video with a high resolution, commonly starting from 1280x720 so lower value of *minDistance* could only reduce the computational speed.

The final parameter, *qualityLevel* was set doing some test with all 5 videos and setted to 0.01, so all the points with the quality measure less then *quality*0.01* are rejected.

Next step is calculate the optical flow using the iterative Lucas-Kanade method with pyramids made by the function *calcOpticalFlowPyrLK*.

3.2 CalcOpticalFlowPyrLK function

This function calculate an optical flow for a sparse feature set using the iterative Lucas-Kanade method with pyramids implement by Jean-Yves Bouguet.

The formulation is:

> *void* ***calcOpticalFlowPyrLK****(InputArray* ***prevImg****, InputArray* ***nextImg****, InputArray* ***prevPts****, InputOutputArray* ***nextPts****, OutputArray* ***status****, OutputArray* ***err****, Size* ***winSize****=Size(21,21), int* ***maxLevel****=3, TermCriteria* ***criteria****=TermCriteria (TermCriteria::COUNT+TermCriteria::EPS, 30, 0.01), int* ***flags****=0, double* ***minEigThreshold****=1e-4)*

where each parameter is:

- **prevImg**: first input image or pyramid constructed
- **nextImg**: second input image or pyramid of the same size and the same type as *prevImg*
- **prevPts**: vector of 2D single-precision floating-point for which the flow needs to be found
- **nextPts**: output vector of 2D single-precision floating-point coordinates containing the calculated new positions of input features in the second image

- **status**: indicate the status of output vector, each element is set to 1 if the flow for the corresponding features has been found, otherwise, it is set to 0
- **err**: indicate the the errors of output vector, each element is set to an error for the corresponding feature, if the flow wasn't found then the error is not defined
- **winSize**: referred to the size of the search window at each pyramid level. Is setted by default
- **maxLevel**: indicate the number of level used in the iterative pyramid algorithm, if it is setted to 0, pyramid are not used, if set to 1, two level are used, and so on. Is setted by default
- **criteria**: specifying the termination criteria of the iterative search algorithm. Is setted by default
- **flags**: indicate a specific operation to add at the algorithm
- **minEigThreshold**: indicate the treshold where, if the minimum eigen value of a 2x2 normal matrix of optical flow equations, divided by number of pixels in a window, is less then this parameter, then corresponding feature is filtered out and its flow is not processed. Is setted by default

Essentially this function return a vector that contain the displacement from previus to current position of all good feature points of an image. In addition return if the operation is good or not by *status* parameter and, if the operation is good, the function explicit the quality with a number.

The next step is filter out all point in the case of the corresponding features has not be found.

Now I compute the transformation matrix of these point for exctract our coordinates, using the *estimateRigidTransform* function.

3.3 estimateRigidTransform function

This function computes an optimal affine transformation between two 2D point sets and return it in a 2x3 matrix. Is defined as:

> *Mat* ***estimateRigidTransform****(InputArray* ***src****, InputArray* ***dst****, bool* ***fullAffine****)*

where each parameter identify:

- **src**: first 2D point stored in array, matrix or immage
- **dst**: second 2D point set of the same type and size as the first
- **fullAffine**: if it is true, the function finds an optimal affine transformation with no additional restrictions (6 degrees of freedom). Otherwise, the class of transformations to choose from is limited to combinations of translation, rotation, and uniform scaling (5 degrees of freedom).

In our case the *fullAffine* parameter is setted to *false*, so the function compute only the rotation and traslation. In rare case the function do not found any transform, so in this case I use the last know good transform.

Now I can extract the info about dx in position of previus returned matrix (0, 2) and dy at (1, 2). Now I can draw an arrow starting that indicate de direction of the flow. In the output video this arrow is fill by red color.

In the first analisys,we can see that the value is accurate but there are a lot of element that disturb the goodnes of the algorithm. Infact the arrow is very quick and some time jump from a point to another too quickly. This happend becouse some moving objects in the scene are detected. So the next step is smoth the result in order to avoid big jump of value and have a more fluid flow. For do this i use an averaging window filter.

3.4 Averaging window filter

The averaging window filter is a calculation to analyze data points by creating a series of averages of different subsets of the full data set. Given a series of numbers and a fixed subset size, the first element of the moving average is obtained by taking the average of the initial fixed subset of the number series and next divived by the size of the current subset. Then the subset is modified by "shifting forward"; that is, excluding the first number of the series and including the next number following the original subset in the series. This creates a new subset of numbers, which is averaged. This process is repeated over the entire data series.

The subset is our case is set to 2 times the $SMOOTHING\_VAL$ constant and it is equal to 10. So the average is made by a group of 10 value. This parameter is set doing a lot of test in orfer to have a good reaction of the arrow. The arrow in the output video is fill with a blue color.

Image 1 *- A frame that rappresent the origina video and the arrow*

The next optional step, detect the “path” of the camera fitted in a plane. Is not a true operation infact the feature point is not all in the focal plane. But is a good way to have a confirm of the direction watching more clearly the result, infact it depends by the optical flow.

The implementation of this trajectory is very simple. I update the value of a variable starting from the center of a new black video with a grid, with a averange value of the optical flow. I prefer to use the average value respect the real one becouse is more clearly and light to watch and make analysis. The trajectory output video is called "trajectory" and the path is indicated by a red arrow.

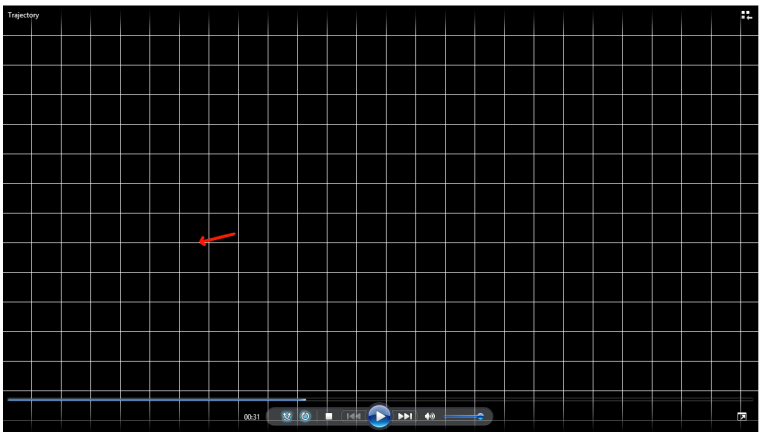

***Image 2** - A frame that rappresent the path using an arrow*

For draw the arrows i'm not able to use the default implemented OpenCV function, so i made it.

The function is very simple and is define as:

void ***drawArrow****(Mat* ***image****, Point* ***start****, Point* ***end****, Scalar* ***color****, int* ***arrow_magnitude****, int* ***thickness****, int* ***line_type****, int* ***shift****)*

where the parameters are:

- **image**: the matrix where the arrow must be drawed
- **start**: is the point where the line start
- **end**: in the point where the line end
- **color**: define the color of the line. Accept a scalar of three integer corrisponding to RGB value (in order BGR) from 0 to 255
- **arrow_magnetude**: indicate the magnetude of the segment that start from the end point of the line, with a defined inclination
- **thickness**: is the thickness of the line and the segment that compose the arrow
- **line_type**: is the type of the line that the function *line* accept
- **shift**: is the number of fractional bits in the point coordinates

This function simply drag a line from the start to the end point given, compute the angle of this segment and create anothe two segment that start from the and is direct at 45 degree respect the line in both sense.

4 Analysis

The analysis is made only looking the video with the arrow and, in my opinion the algorithm work very well. Can be see some noise specially in the trajectory video.

The Algorithm work fine also with poor video quality. In fact, even if the good feature points are low in number, the optical flow appear correcty.

The only one problem is when I tested the second video. I'm not sure that was taken with a tablet or smartphone, but in the first part the user that had taken the video, zooming in and the algorithm report a orizzontal flow. The fact is that when I compute the trasformation matrix I'm not consider the scaling factor, only the rotation and the translation. So the feature point move respectively to the left and right, and the optical flow algorithm return a wrong value.

5 Conclusion

In conclusion I can say that this algorithm work fine. The arrow follow the camera movement very well.

The algorithm can also be refine in this way:

- Considering the scale factor when the code compute the trasformation matrix, in order to avoid the problem in the case of zomming in or out.
- The algorithm is not created to work in real time, but can work with a little delay due to the average window filter, In my case, with my PC, the computational time is very long with a 1280x720 pixel of resolution video, so the code can be further improved in terms of speed.
- Another possible implementation, in my opinion, is adapt the good feature tracking parameters according to the resolution and bitrate of the video. In fact the paramaters used in this code, return to much points for a 2k video and there is a risk to have points that are too close each other. In the opposit case, with a 240p video the minimum Euclidean distance should be adjust and considering more points to track becouse the corner are less defined.

References

1. Nghia Ho - Simple video stabilization using OpenCV – http://nghiaho.com/?p=2093
2. Mliki Hazar - Draw an Arrow with OpenCV (and python version too) – http://mlikihazar.blogspot.it/2013/02/draw-arrow-opencv.html
3. OpenCV - OpenCV Documentation – http://docs.opencv.org/
4. Youtube - Last Minute Goal Sergio Ramos Champions Final – https://www.youtube.com/watch?v=A1ioiDN-UPs&index=63&list=WL
5. Youtube - Hellas Verona - Modena 2-1: goal di Lepiller, 1-1. – https://www.youtube.com/watch?v=5BOSE382mAA