Video Stabilization Algorithm Using a Block-Based Parametric Motion Model

Ting Chen

EE392J Project Report, Winter 2000

Information Systems Laboratory
Department of Electrical Engineering, Stanford University, CA 94305, USA

ABSTRACT

Video footage from hand-held camcorders is typically jerky due to small, unwanted camera movements. Video stabilization techniques are used to generate a stabilized video sequence where image motion by the camera's undesirable shake or jiggle is removed. In this report we will describe a video stabilization algorithm using a block-based parametric motion model. In particular we show how to apply our algorithm to translational and rotational camera motions. Good performance is obtained experimentally.

Keywords: Motion Estimation, Block-Matching, Affine Motion Model, Motion Smoother, Motion Correction

1. INTRODUCTION

Video footage from hand-held camcorders is typically jerky due to small, unwanted camera movements. Removal of those undesired movements requires video stabilization techniques. As its name suggests, video stabilization is the process of generating a compensated video sequence where image motion by the camera's undesirable shake or jiggle is removed. Digital image processing techniques are often used to perform such a task and are favorable over mechanical or optical video stabilization approaches since modern VLSI techniques will allow a more compact camera design.

Many methods for video stabilization have been reported over the past few years. Some methods[1–5] use a global motion model followed by filtering to remove motion related artifacts from video frames. Morimoto et al.[6] describe an algorithm using a feature-based 2D rigid motion model[7] that deals with rotational and translational camera motion. Ratakonda[8] describes an integral projection matching algorithm for compensating translational camera movement. Ko et.al [9,10] describe an video stabilization algorithm with Gray-coded bit-plance matching for motion estimation and with median-based motion correction. In this report we describe a video stabilization algorithm whose a few aspects can be found similar to the algorithm based on the 2D rigid motion model.[6] But instead of using feature-tracking, our parametric motion model is obtained by fitting the dense optical flow information from block-matching motion estimates. We will show how our algorithm applies to translational and rotational camera motion separately, even though the former can be included in the latter as a special case.

The report is organized as follows. In Section 2, we will briefly describe the structure of the video stabilizatio algorithm. In Section 3 translational camera movement is assumed and the algorithm is shown in detail. Some simulation results on test sequences will also be included. In Section 4 we will show how the video stabilization algorithm described in Section 3 can be expanded to non-translational motions using parametric motion models. More specifically, rotational camera motion will be used as an example to illustrate the basic idea involved. Finally in Section 5 we will describe some interesting observations and provide miscellaneous information relevant to our project.

Correspondence: Email: tingchen@isl.stanford.edu; Telephone: 650-725-9696; Fax: 650-723-8473

2. VIDEO STABILIZATION ALGORITHM STRUCTURE

Our video stabilization algorithm consists of a motion estimation (ME) block, a motion smooth (MS) block and a motion correction (MC) block, as shown in figure 1. ME estimates the motion between frames and can be divided as a local motion estimator and a global motion decision unit. Basically the local motion estimator will return the estimated dense optical flow information between successive frames using typical block-based or gradient-based methods. The global motion decision unit will then determine an appropriate global transformation that best characterizes the motion described by the given optical flow information. The global motion parameters will be sent to the MS, where the motion parameters are often filtered to remove the unwanted camera motion but retaining intentional camera motion. Finally MC warps the current frame using the filtered global transformation information and generates the stabilized video sequence.

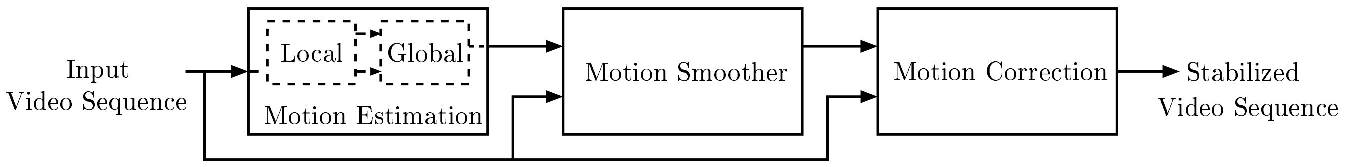


Figure 1. Block Diagram of Video Stabilization Algorithm

As in any video stabilization algorithm, the crucial component of our algorithm lies on motion estimation. Since in real world, camera motion often involves some type of global transformation, it is particularly important to get an accurate estimate of the global motion when performing motion estimation. To account for various types of motions such as rotation, zoom and so on, applying parametric motion models is highly preferred over pure block-based motion methods. On the other hand, to better illustrate the algorithm, it is, however, always advantageous to start with the simplest scenario, which in our case is the translational camera motion which we shall describe in next section.

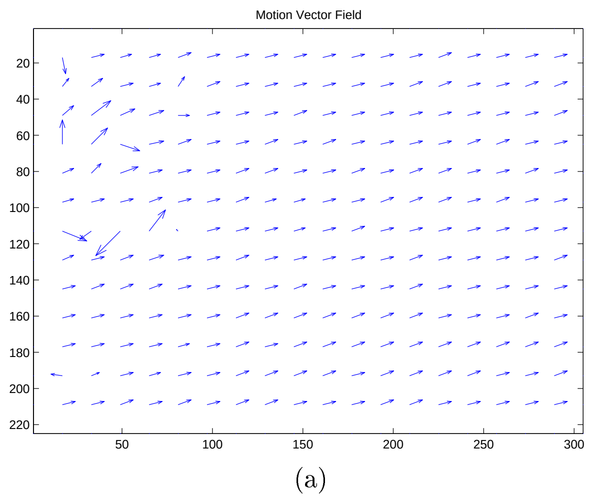
3. VIDEO STABILIZATION WITHOUT PARAMETRIC MOTION MODEL

Translational camera motion is likely the most often encountered camera motion in real world. The simplicity of characterizing such type of camera motion lies on the fact that motion between adjacent frames can be accurately represented by a global motion displacement vector. Parametric motion models are obviously not required for characterizing such an easy motion. We will follow the block diagram shown previously in figure 1 as we describe our video stabilization algorithm in detail.

3.1. Motion Estimation – Translational Camera Motion

Local motion estimation is used to obtain the dense optical flow information for successive frames and is typically done with block-based or gradient-based methods. In our algorithm implementation, we use a block-matching method with full search strategy. The block size is chosen as 16x16 and search range between -15 and +15 pixels in both horizontal and vertical directions. The search criterion is the mean squared error between corresponding blocks.

Given the dense optical flow information, the global motion decision unit will first try to determine the appropriate type of camera motion, which we will provide some detail in Section 4. For now assume the motion type is known to the global motion decision unit and a single motion displacement vector needs to be determined. For translational camera motion, it is expected that the majority of the local block motion vectors will be the same, as illustrated in figure 2 for a typical translational movement. And with simple statistical analysis, for instance, taking the median of the motion vectors, we can remove the effect of those outlier motion vectors and get an accurate estimate of the actual camera translation. In the example shown in figure 2, we get the global motion displacement vector $[u, v] = [8, -2]$. The ME block will compute the global motion displacement vector for each pair of successive frames.



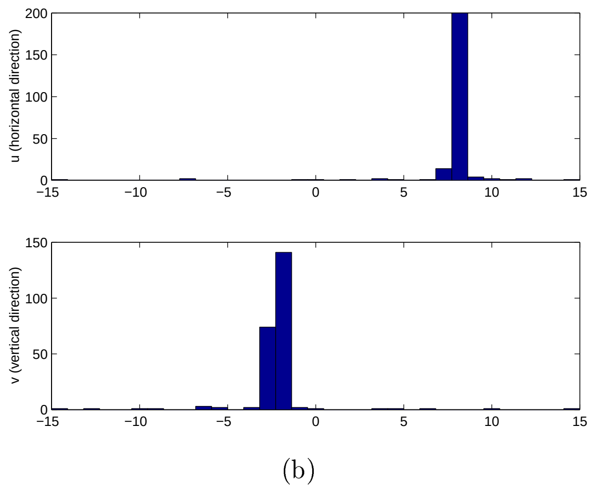


Figure 2. Motion estimation results for a typical translational camera motion (frames taken from test sequence "street lamp") (a) quiver plot for motion vectors (b) histograms of motion vectors

3.2. Motion Smoother – Translational Camera Motion

Motion smoother is used to smooth out the abrupt camera motions. The video algorithm needs to distinguish the intentional camera movement from the unwanted camera motion. To achieve such a goal, we have to make some assumptions about the nature of camera motions. In particular, we assume that intentional camera motion is usually smooth with slow variations from frame to frame (*i.e.*, temporally). On the other hand, unwanted camera motion involves rapid motion variations over time. In short, the high frequency components in the motion vector variations over time are considered to be effects of unwanted camera motion and therefore need to be removed. Removing the high frequency components can be done by applying low pass filters on the motion vectors. We choose to use a Moving Average (MV) filter, as defined in equation 1.

$$h[n] = \begin{cases} \frac{1}{N} & \text{for n = 0,1,...,N-1} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

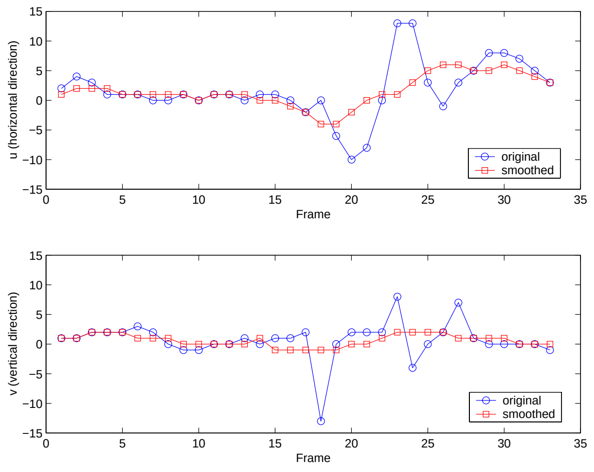
The smoothed motion vectors are obtained by convolving the original motion vectors with the MV filter and are shown in figure 3 with $N = 7$ for test sequences.

Notice even though in our simulation we obtained the global motion vectors for the entire video sequence first, it is not really required. Depending on the support of the filter, we only need to store motion vectors for a few frames at any time. For instance, we need to store motion vectors for 7 frames when $N = 7$. Therefore, real-time implementation is achieveable.

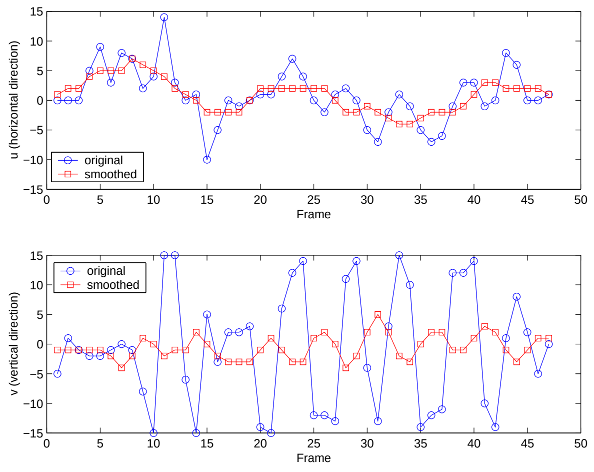
3.3. Motion Correction – Translational Camera Motion

Motion correction is where stabilization of the video sequences occurs. Using the smoothed motion vectors from motion smoother, we can perform motion compensation frame by frame. We obtain stabilized frame 2 by performing motion compensation on original frame 1 using the corresponding smoothed motion vector. To get stabilized frame 3, we use stabilized frame 2 and its corresponding smoothed motion vector. In such a fashion, the entire stabilized video sequence can be generated.

In our algorithm, since each stabilized frame is built upon previous stabilized frame, it is very likely that an error caused by the motion compensation at the beginning of the video sequence can propagate to subsequent frames. The accumulation of such errors can be quite significant. For the case of translational camera motion, all the errors occur at the frame boundaries and appear as stripe-pattern artifacts, as shown in figure 4(a). To remove these annoying artifacts, error propagation control techniques must be used.

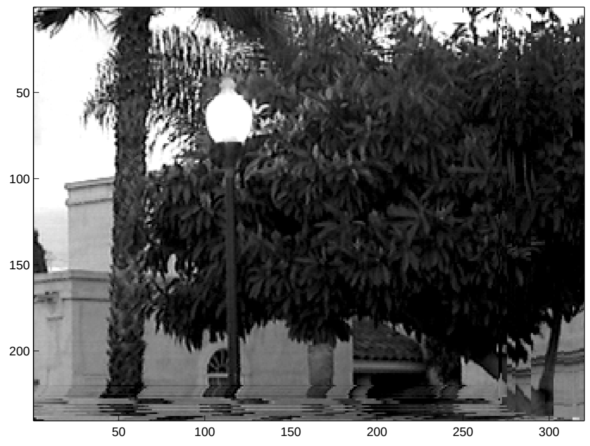


(a)



(b)

Figure 3. Original motion vectors and smoothed motion vectors for (a) test sequence : "street lamp" (b) test sequence : "yard fountain"


(a)

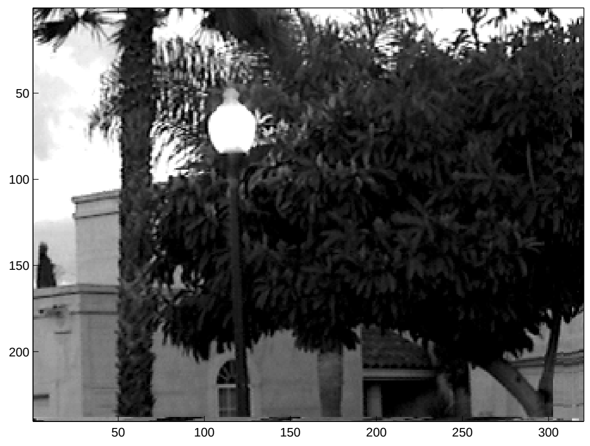

(b)

Figure 4. Frame 34 from test sequence "street lamp"(a)Stabilized frame has annoying artifacts near the boundary (bottom) due to error propagation (b)With error propagation control techniques, artifacts can be reduced significantly

Our error propagation control technique works as follows. At each frame instance, we calculate the accumulated motion vector for both original sequence and stabilized sequence. If the accumulated original MV is within a distance threshold from the accumulated smoothed MV, instead of using the current stabilized frame to estimate next frame, we will use the corresponding original frame to "synchronize" the video sequence. This can be illustrated as :

```
At each frame i,
if   $|\sum_{j=1}^{i-1} u_{j,original} - \sum_{j=1}^{i-1} u_{j,smoothed}|^2 + |\sum_{j=1}^{i-1} v_{j,original} - \sum_{j=1}^{i-1} v_{j,smoothed}|^2 \leq threshold^2$,
     stabilized frame $i+1 = f(\text{original frame } i,\ \text{smoothed MV } i)$
     update smoothed MV $i$
else
     stabilized frame $i+1 = f(\text{stabilized frame } i,\ \text{smoothed MV } i)$
end
```

where $[u_j\ v_j]$ is the estimated global motion vector for frame j and frame $j+1$. $f(\cdot\ ,\ \cdot)$ represents the motion compensation function.

By using a small threshold value, we can make sure there is no evident abrupt movement resulted when replacing the stabilized frame with original frame. On the other hand, by using a large threshold, we synchronize the stabilized video sequence more with original sequence, therefore less error propagation is expected. Such a scheme allows us to trade off the smoothness of video sequence with artifacts generated by the video stabilization algorithm. We also choose to force a frame synchrnoization every 12 frames in the case where the above scheme isn't invoked. In figure 5, we plot the original motion vectors and smoothed motion vectors with and without error propagation control techniques at each frame. Figure 4(b) shows the result of applying our error propagation control technique.

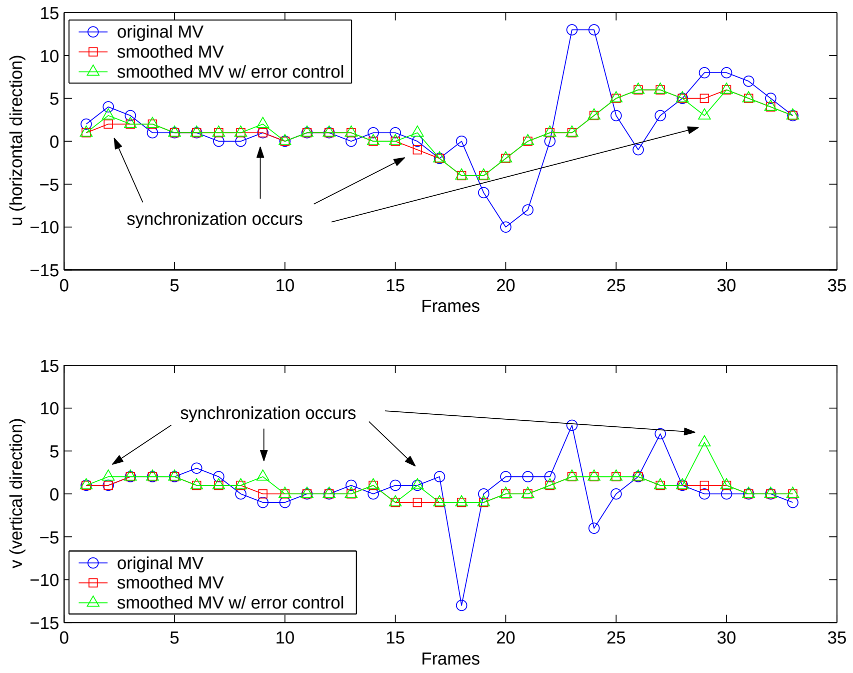


Figure 5. Original motion vectors, smoothed motion vectors with/without error propagation control

Our experiments have shown good results visually. Indeed we are able to obtain much smoother video sequences and abrupt camera motions appear a lot stable after the processing. Unfortunately, only still images can be shown on paper and are not very appropriate for displaying a dynamic process such as video stabilization. Nevertheless, we will try to give readers a feel in figure 6. The upper left image in (a) shows one frame of the original sequence while the upper right image in (b) is the stabilized frame at the same time instance. The bottom left image in (c) shows the difference between the frame in (a) and its previous frame from original sequence. Similarly (d) shows their difference after stabilization. In both cases, only the pixels with absolute difference higher than 25 are displayed. Since the algorithm attempts to remove large and rapid camera movement, the decrease of the difference between images indicates that more smooth motion is resulted.

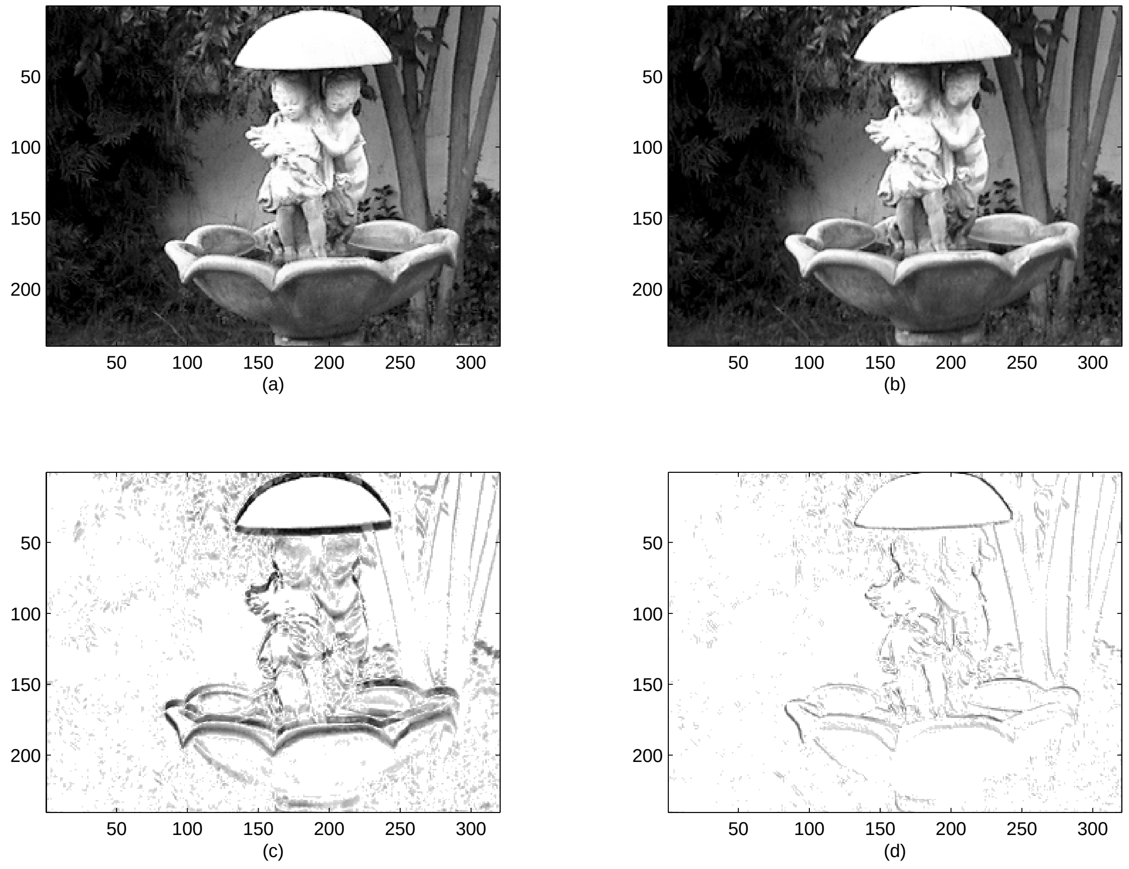


Figure 6. Results of stabilization algorithm shown with still images

4. VIDEO STABILIZATION WITH PARAMETRIC MOTION MODEL

In this section, we will describe how our algorithm can be extended to account for other types of motion via the help of parametric motion model. More specifically, we will look at how this algorithm can be applied to rotational camera motion.

4.1. Motion Estimation – Rotational Camera Motion

We use the same approach, *i.e.* block-matching method, to obtain the local optical flow information as described in Section 3.1. The error criterion, however, becomes a weighted MSE, as shown in Figure 7 for the 16×16 block. We will explain the reason for using such a weighted MSE criterion later in this section. The global motion decision unit next has to determine the appropriate type of camera motion. Since in our project study, we only consider translational and rotational camera motion. Distinguishing one from the other becomes straightforward by examining the statistics of block motion vectors. Compared to translational camera motion, rotational camera motion typically results close to uniform distribution of MVs, as shown in figure 8. Therefore, to decide whether the motion is translational or not, we calculate the variance of the distribution. If the variance is within a threshold value, the motion is considered as translation, otherwise it is taken as rotation. In our simulation, the threshold is set to 5. Figure 9 plots the variance for the local block MV distributions for two sequences with rotational and translational camera movement, respectively. Notice in Figure 9(a), the distribution of block MVs on one of the frame has a large variance exceeding the threshold, this is due to the fact that the frame is motion blurred and block-matching ME fails essentially. We will describe this phenomenon with a little bit more detail in Section 5.

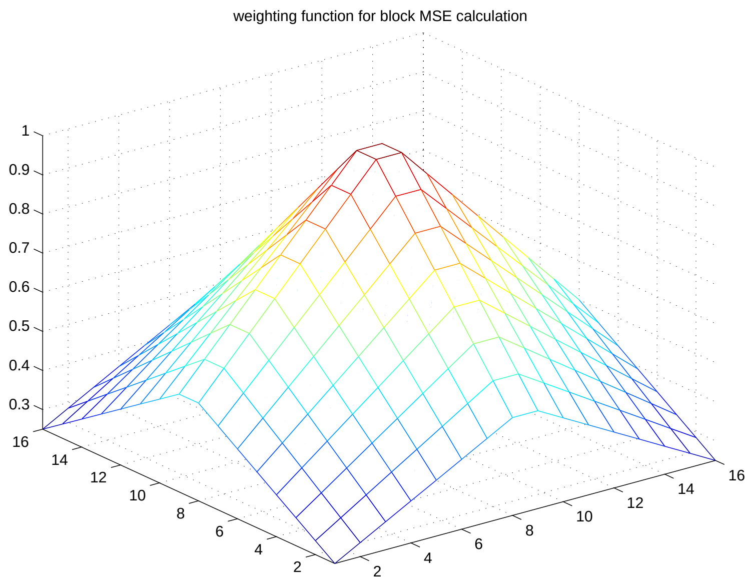


Figure 7. Applying weighted MSE can be used emphasize the accuracy of motion estimation in desired region within a block. The weighting function is shown here

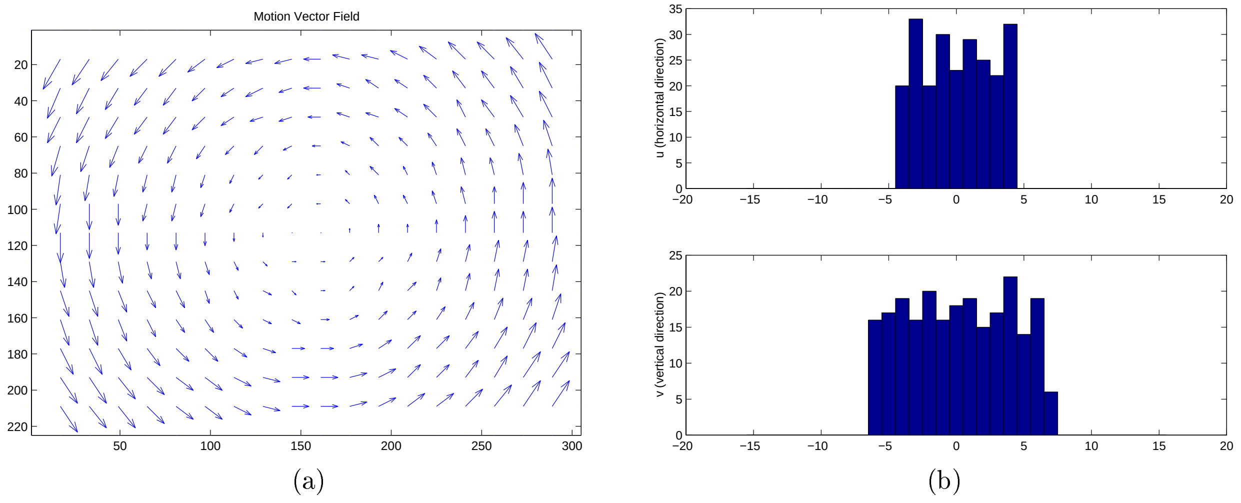
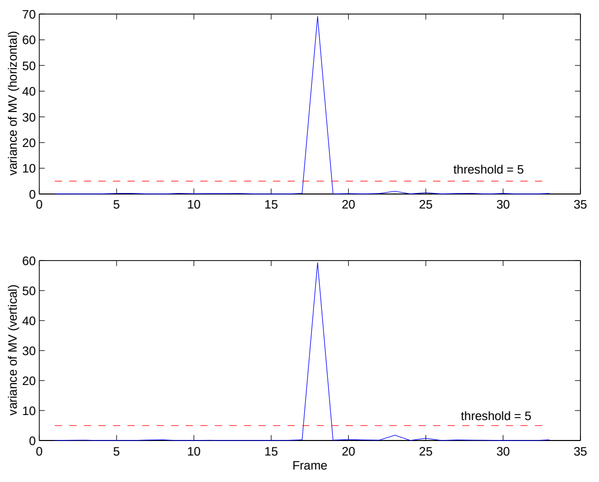
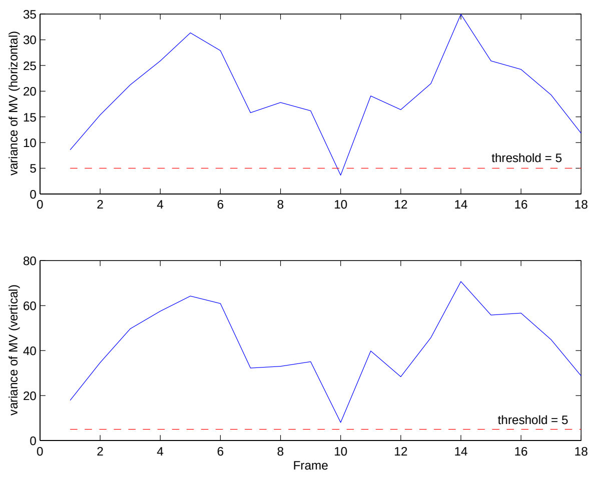


Figure 8. Motion estimation results for a typical translational camera motion (frames taken from test sequence “street lamp”) (a) quiver plot for motion vectors (b) histograms of motion vectors



(a)



(b)

Figure 9. Variances of block MV distributions are different for translation and rotation, therefore can be used to distinguish one from the other. (a) Translation – from test sequence "street lamp" (b) Rotation – from test sequence "flower"

Once it is determined that the camera motion involved is rotation. We will use affine motion model to characterize the global motion. In particular, we assume there is no zoom or scaling occurred. Therefore the affine transformation can be specified by a rotation matrix and two offsets, as shown in the following equations.

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} \alpha & -\beta \\ \beta & \alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x_{offset} \\ y_{offset} \end{bmatrix} \tag{2}$$

$$= \sqrt{\alpha^2+\beta^2} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x_{offset} \\ y_{offset} \end{bmatrix} \tag{3}$$

where $\theta = \arctan(\frac{\beta}{\alpha})$ represents the angle of rotation.

To determine the global transformation parameters, we will find correspondence points using the estimated local block motion vectors. For each pair of corresponding blocks, we select the center points as correspondence points. And because of this particular choice, it becomes more crucial to have an accurate motion estimation for the center points than for the peripheral points within a block. This justifies our weighted MSE criterion as depicted in Figure 7. To obtain the affine motion parameters $\alpha, \beta, x_{offset}$ and y_{offset}, we rearrange equation 2 and concatenate the equations for all the correspondence points together in equation 4.

$$\begin{bmatrix} X_1 \\ Y_1 \\ X_2 \\ Y_2 \\ . \\ . \\ . \\ X_N \\ Y_N \end{bmatrix} = \begin{bmatrix} x_1 & -y_1 & 1 & 0 \\ y_1 & x_1 & 0 & 1 \\ x_2 & -y_2 & 1 & 0 \\ y_2 & x_2 & 0 & 1 \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ x_N & -y_N & 1 & 0 \\ y_N & x_N & 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ x_{offset} \\ y_{offset} \end{bmatrix} \tag{4}$$

Equation 4 describes an over-determined system and therefore the least-square solution is required. The result will give us the desired global affine motion parameters.

4.2. Motion Smoother – Rotational Camera Motion

The motion smoother performs similarly as in section 3.2. The only difference is that instead of applying an MV filter on the motion displacement vectors, we need to filter the angle of rotation together with the offset vectors which characterize the translational motion involved. Equation 5 shows how the smoothed affine motion parameters can be generated from the smoothed angles and offsets.

$$\begin{bmatrix} \alpha_s \\ \beta_s \\ x_{offset,s} \\ y_{offset,s} \end{bmatrix} = \begin{bmatrix} \sqrt{\alpha_o^2 + \beta_o^2}\cos\theta_s \\ \sqrt{\alpha_o^2 + \beta_o^2}\sin\theta_s \\ x_{offset,s} \\ y_{offset,s} \end{bmatrix} \tag{5}$$

where subscript "o" implies "original" and "s" stands for "smoothed". Figure 10 shows the results of applying an MV filter of length 5 on the rotation angle and offsets. It can be seen that the original rotation was actually very smooth, consequently the smoothed rotation isn't too much different.

4.3. Motion Correction – Rotational Camera Motion

Motion correction is done in the same fashion as described earlier in Section 3.3 for translational camera movement. Each stabilized frame is obtained from previous stabilized frame and corresponding smoothed affine motion parameters. We use the same type of error control techniques where both smoothed rotation angles and offsets are accumulated at each frame and compared to the values from the parameters before being filtered. Figure 10 also shows the effect of error propagation control. Finally a still image example is shown in figure 11. As expected, since the original sequence is quite smooth, the error difference in (c) and (d) appears similar, which indicates that smooth movements are considered as intentional and therefore preserved by our algorithm.

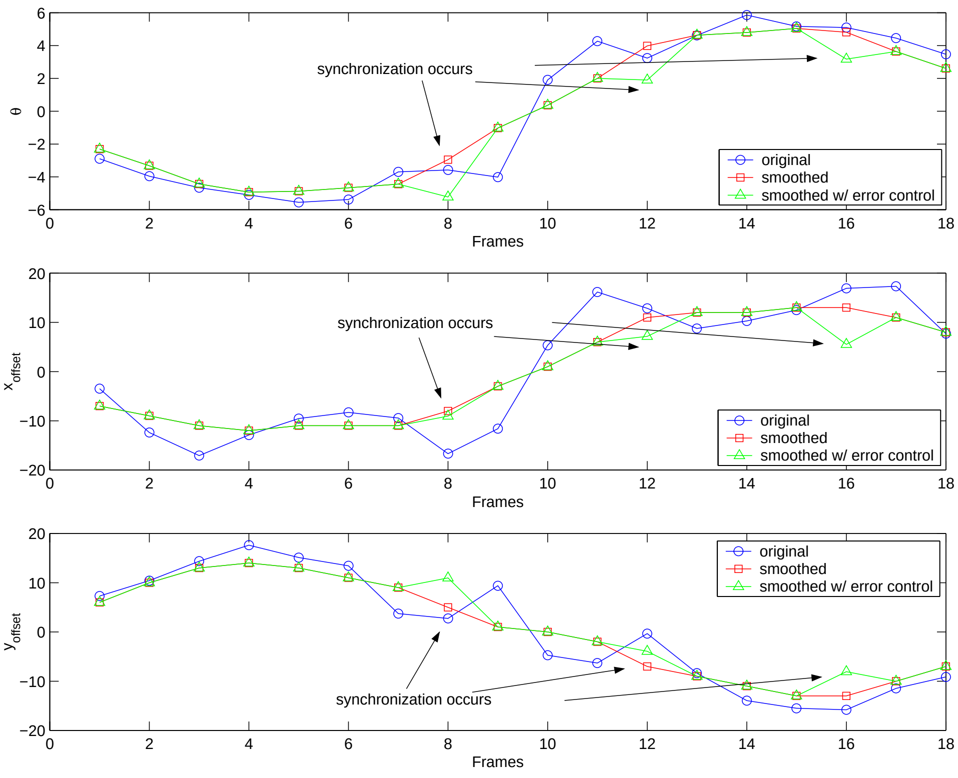


Figure 10. Original motion vectors, smoothed motion vectors with/without error propagation control

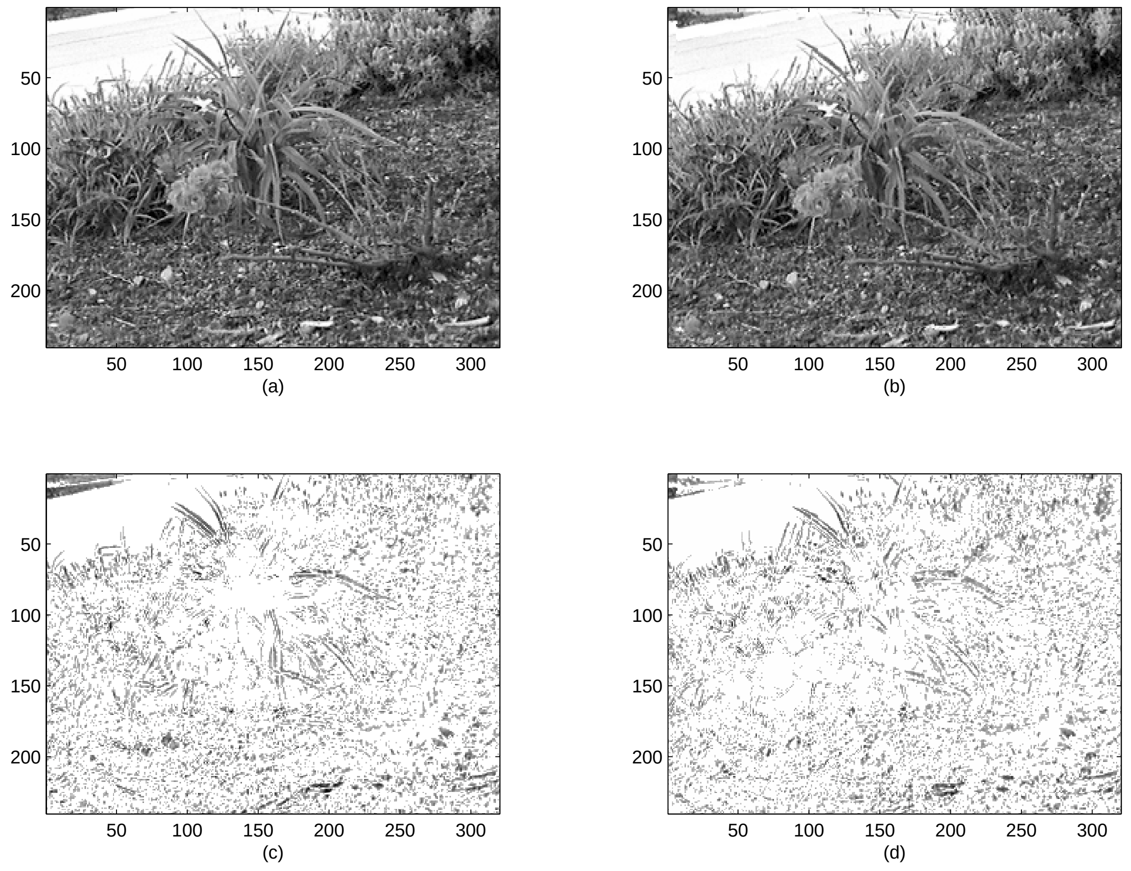


Figure 11. Results of stabilization algorithm shown with still images

5. OTHER OBSERVATION AND RELEVANT PROJECT INFORMATION

In this section we briefly discuss one interesting observation regarding to the effect of motion blur on motion estimation. We will also provide some information for the project implementation.

5.1. Effect of Motion Blur on Algorithm Performance

As described in previous sections, local motion estimation is crucially related to the overall performance of the algorithm since other steps, such as determining the type of motion and estimating the global motion transformation parameters, rely on the accuracy of the local motion estimates. Unfortunately because of camera movement, it is very likely that some of the frames captured become blurred and we have observed that the blur has very large but negative impact on the motion estimation. Figure 12(b) shows the block motion vectors obtained from the two frames in (a). The two frames are taken from the test sequence with translational camera movement only and the first frame is blurred due to camera motion. As a result, the motion vectors appear a lot more random which is not the typical characteristics as seen in figure 2. Also notice that if the blur is due to the camera motion in one direction primarily, it is in the other direction that motion estimation becomes difficult and often wrong. In the example frame where the translational motion is in vertical direction with little motion horizontally, the motion estimation fails in horizontal direction but still is able to capture the movement in vertical direction, as shown in Figure 12(c).

Given the fact that motion blur destroies the motion estimation accuracy, it seems necessary to apply the video sequence to a deblurring function before performing any motion estimation. Alternatively, sophisticated motion estimation algorithms that can detect the blurred frames and remove or correct corresponding errors need to be utilized. On the other hand, because in our algorithm each stabilized frame is built upon previous stabilized frame, we can actually eliminate any blurred frame provided that our sequence doesn't start with a blurred one and all the frame "synchronizations" occur on clear frames.

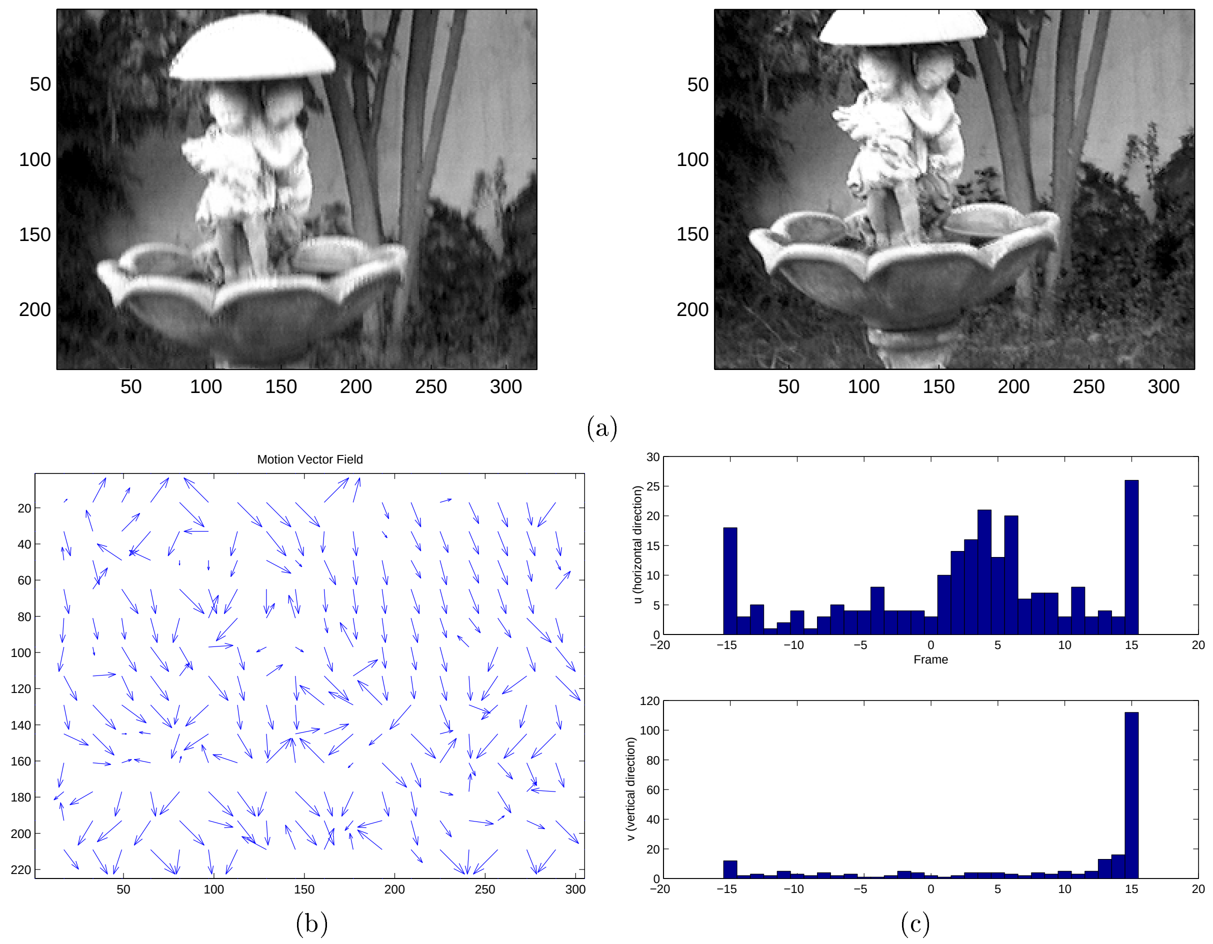


Figure 12. Effect of motion blur on motion estimation for translational camera movement (a)two successive frames with the first frame motion blurred (b)quiver plot of block motion vectors shows randomness (c)histograms of motion vectors in horizontal direction reveal a spread distribution (instead of a compact distribution as seen in Figure 2)

5.2. Project Implementation Information

A few notes regarding to our project implementation :

- **Test sequences** are shot with a SONY camcorder on 8mm tape, subsequently uploaded into a PC through a TV tuner card in .avi format. A few C programs are written to perform deinterlacing, re-scaling and decoding .avi into raw data. Before loaded into Matlab, all the test sequences are of size 320x240 and are colored in RGB raw format. We worked with both gray-scale and color test sequences where in both cases motion estimation is done on the Y plane after color conversion from RGB to YIQ color space.

- **Algorithm implementation** is entirely carried out using Matlab and relevant Matlab files are included in Appendix.

6. CONCLUSION

We have described a video stabilization algorithm using a block-based parametric motion model in this report. In particular we showed how to apply our algorithm to translational and rotational camera motions. Experimental results have indicated good performance from our algorithm. However, due to time constraints, there are a number of issues that we didn't fully address but would certainly like to do so in the future. We list them below for future reference and improvement.

- Some components of the algorithm, such as dealing with blurred frames, are missing at this point and certainly need to be included in the future.
- Some features of the algorithm, such as selecting the appropriate type of camera motion, are implemented using ad-hoc approaches. Though they worked well for the simple cases we considered, more advanced techniques are definitely desired to work with many more difficult scenerios found in real world.
- Other parameteric motion models such as perspective and bilinear motion models need to be included to deal with other type of camera motions, especially when scaling or zoom occurs. However, to find a meaningful parameter (*i.e.* rotation angle, motion displacement...) to smooth with may present challenges to those complicated camera motions.
- Currently there is no objective measure developed to evaluate the performance of our video stabilization algorithm. The performance is judged visually. A good objective measure is necessary for systematically developing and evaluating any video stabilization algorithm.

ACKNOWLEDGEMENTS

The project was done to fulfill the class requirement of EE392J Digital Video Processing in winter 2000. The author would like to thank the course instructors John Apostolopoulos and Susie Wee, course teaching assistant Hareesh Kesavan for their great help. The author also wants to acknowledge the valuable discussion from his officemates in Packard 257.

REFERENCES

1. M.Hansen, P. Anandan, and K. Dana, "Real-Time Scene Stabilization and Mosaic Construction," in *Image Understanding Workshop Proceedings*, vol. 1, pp. 457–465, 1994.
2. Y. Yao, P. Burlina, , and R. Chellappa, "Electronic Image Stabilization Using Multiple Image Cues," in *Proceedings of ICIP*, vol. 1, pp. 191–194, 1995.
3. P. Pochec, "Moire based Stereo Matching Technique," in *Proceedings of ICIP*, vol. 2, pp. 370–373, 1995.
4. J. Tucker and A. de Sam Lazaro, "Image Stabilization for a Camera on a Moving Platform," in *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, vol. 2, pp. 734–737, May 1993.
5. K. Uomori, A. Morimura, H. Ishii, T. Sakaguchi, and Y. Kitamura, "Automatic Image Stabilizing System by Full-Digital Signal Processing," *IEEE Transactions on Consumer Electronics* **36**(3), pp. 510–519, August 1990.
6. C. Morimoto and R. Chellappa, "Fast Electronic Digital Image Stabilization," in *Proceedings of the 13th International Conference on Pattern Recognition*, vol. 3, pp. 284–288, August 1996.
7. Q. Zheng and R. Chellappa, "A Computational Vision Approach to Image Registration," *IEEE Transactions on Image Processing* (2), pp. 311–326, 1993.
8. K. Ratakonda, "Real-Time Digital Video Stabilization for Multi-Media Applications," in *Proceeding of the 1998 IEEE International Symposium on Circuits and Systems*, vol. 4, pp. 69–72, May 1998.
9. S.-J. Ko, S.-H. Lee, and K.-H. Lee, "Digital Image Stabilizing Algorithms Based on Bit-Plane Matching," *IEEE Transactions on Consumer Electronics* **44**(3), pp. 617–622, August 1998.
10. S.-J. Ko, S.-H. Lee, S.-W. Jeon, and E.-S. Kang, "Fast Digital Image Stabilizer Based on Gray-Coded Bit-Plane Matching," *IEEE Transactions on Consumer Electronics* **45**(3), pp. 598–603, August 1999.

Appendix

Here we list all the relevant Matlab codes written to implement our algorithm.

main.m

```
%% Main file for :
%% Video Stabilization Algorithm Using a Block-Based Parametric
%%                          Motion Model
%%
%% Class Project, EE392J Winter 2000
%% Author : Ting Chen
%% Date   : 02/24/2000
%%
clear all;
close all;

%% Step I : load original video sequences
%% sequence : structure
%%            fields :  name -- string
%%                      originalFrames -- [nRows x nCols x nFrames]
%%
sequence.name = 'flower';   % 'kids','car','lamp','flower'
sequence = loadSequence(sequence);
%filename = strcat(sequence.name,'File_step1');
%save(filename,'sequence')
%load(filename)

%% Step II : Perform motion estimation; and save all the relevant
%%           motion vector into structure sequence.estimatedMotion
%%
sequence.estimatedMotion = sequenceME(sequence);
%filename = strcat(sequence.name,'File_step2');
%save(filename,'sequence')
%load(filename)

%% Step III: Determine whether the motion is intentional or unwanted
%%           and smooth out/remove the unwanted motion
%%
sequence = motionClassification(sequence);
%filename = strcat(sequence.name,'File_step3');
%save(filename,'sequence')
%load(filename)

%% Step IV : Correct the unwanted motion and generate stabilized
%%           video sequence store the motion stabilized frame
%%           into structure sequence
%%
sequence.MCpara.dThreshold = 8;   % 4 for 'car', 1 for 'lamp' and 'kids'
                                  % 8 for 'flower'
sequence.MCpara.aThreshold = 3;   % for 'flower'
sequence.MCpara.resetLength = 12;
sequence = motionCorrection(sequence);
```

```
%filename = strcat(sequence.name,'File_step4');
%save(filename,'sequence')
%load(filename)

%% Step V : Display original/stabilized video sequence
%%
sequence.displayOption = 'color'; % by default it's b&w, i.e., the Y plane
M = sequenceDisplay(sequence);
movie(M,-1,12);
```

loadSequence.m

```
function sequence = loadSequence(sequence)
%% function sequence = loadSequence(sequence
%% Purpose : load original video sequence
%% INPUT   : sequence -- structure
%% OUTPUT  : sequence -- structure
%% Author  : T. Chen
%% Date    : 02/24/2000
%%

%% Assign local variables
name = sequence.name;

filepath = strcat('../data/',name,'/',name,'00');

switch name
   case 'lamp',
        frameindex = [0:17 24:28 39:49];  % hand edited lamp sequence
   case 'kids',
        frameindex = [12:59];
   case 'car',
        frameindex = [0:44];
   case 'flower',
        %frameindex = [33:51];              % flowerFile_step2new.mat
        frameindex = [33:2:51 50:-2:34];  % flowerFile_step2.mat
   otherwise
        error('Unrecognized sequence name');
end

originalYPlane = repmat(0,[240 320 length(frameindex)]);
originalIPlane = originalYPlane;
originalQPlane = originalYPlane;
for i = 1:length(frameindex),
    if frameindex(i) < 10,
       filename = strcat(filepath,'0',num2str(frameindex(i)),'.raw');
    else
       filename = strcat(filepath,num2str(frameindex(i)),'.raw');
    end
    currentFrame = readDatafile(filename);

    currentY = 0.3*currentFrame(:,:,1) + ...
              0.59*currentFrame(:,:,2) + 0.11*currentFrame(:,:,3);
    currentI = 0.6*currentFrame(:,:,1) - ...
              0.28*currentFrame(:,:,2) - 0.32*currentFrame(:,:,3);
    currentQ = 0.21*currentFrame(:,:,1) - ...
              0.52*currentFrame(:,:,2) + 0.31*currentFrame(:,:,3);

    originalYPlane(:,:,i) = currentY;
    originalIPlane(:,:,i) = currentI;
    originalQPlane(:,:,i) = currentQ;
end

sequence.originalYPlane = originalYPlane;
sequence.originalIPlane = originalIPlane;
sequence.originalQPlane = originalQPlane;
```

sequenceME.m

```
function estimatedMotion = sequenceME(sequence)
%% function estimatedMotion = sequenceME(sequence)
%% Purpose : perform motion estimation using appropriate motion
%%           estimation methods
%% INPUT   : sequence -- structure
%% OUTPUT  : estimatedMotion -- structure
%% Author  : T. Chen
%% Date    : 02/24/2000
%%

%% Assign local variables :
%%
%% Perform motion estimation on Y plane only
%%
originalFrames = sequence.originalYPlane;
nFrames = size(originalFrames,3);

%% Motion Estimation between adjacent frames
%%
estimatedMotion = [];
for i = 2:nFrames,

    previousFrame = originalFrames(:,:,i-1);
    currentFrame = originalFrames(:,:,i);

    fprintf('Motion Estimation for frame %d and %d\n', i-1,i);
    currentEstimatedMotion = ...
       motionEstimation(previousFrame,currentFrame,sequence);

    estimatedMotion = [estimatedMotion currentEstimatedMotion];

end
```

motionClassification.m

```
function sequence = motionClassification(sequence)
%% function sequence = motionClassification(sequence)
%% Purpose : determine whether the motion is intentional and
%%           smooth out/remove unwanted motion (vectors)
%% INPUT   : sequence -- structure
%% OUTPUT  : sequence -- structure
%%
%% Author  : T. Chen
%% Date    : 02/24/2000
%%

%% Assign local variables
%%
estimatedMotion = sequence.estimatedMotion;
[nRows,nCols,nFrames] = size(sequence.originalYPlane);
kernelLength = 5;  % 5 for 'flower'
MAFilter = ones(1,kernelLength)/kernelLength;

%% Assume the entire sequence can be described by the same
%% type of motion model. In reality, we can always segment
%% the sequence so that each segment of sequences only describes
%% one type of motion.
%%
switch estimatedMotion(1).method

   case 'block',

   %% Examine the estimated motion vectors
   %%
      u = [];
      v = [];
      for i = 1:nFrames-1,
          u = [u estimatedMotion(i).motionVector(1)];
          v = [v estimatedMotion(i).motionVector(2)];
      end

   %% Smooth out the motion vectors
   %% For now, use a Moving Average (MA) filter
   %%
      us = conv(u,MAFilter);
      vs = conv(v,MAFilter);
      if rem(kernelLength,2) == 1,
         us = us((kernelLength+1)/2:(length(us)-(kernelLength-1)/2));
         vs = vs((kernelLength+1)/2:(length(vs)-(kernelLength-1)/2));
      else
         us = us(kernelLength/2+1:(length(us)-(kernelLength/2-1)));
         vs = vs(kernelLength/2+1:(length(vs)-(kernelLength/2-1)));
      end

   %% Only integer pixel movement is allowed
   %%
      us = round(us);
      vs = round(vs);

      figure;
   subplot(211); plot(u,'-o'); hold on; plot(us,'r-s'); hold off
   subplot(212); plot(v,'-o'); hold on; plot(vs,'r-s'); hold off

%% Add new fields to sequence
   sequence.smoothedMVs = [us' vs'];

case 'affine',

   %% Examine the estimated motion vectors
   %%
   alpha = [];
   beta = [];
   xoffset = [];
   yoffset = [];
   theta = [];

   for i = 1:nFrames-1,
       alpha = [alpha estimatedMotion(i).relevantInfo(1)];
       beta = [beta estimatedMotion(i).relevantInfo(2)];
       theta = [theta atan(beta(i)/alpha(i))];
       xoffset = [xoffset estimatedMotion(i).relevantInfo(3)];
       yoffset = [yoffset estimatedMotion(i).relevantInfo(4)];
   end

   %% Smooth out the rotation (angle) and translation (offsets)
   %% For now, use a Moving Average (MA) filter
   %%
   thetas = conv(theta,MAFilter);
   xoffsets = round(conv(xoffset,MAFilter));
   yoffsets = round(conv(yoffset,MAFilter));

   if rem(kernelLength,2) == 1,
      thetas = ...
      thetas((kernelLength+1)/2:(length(thetas)-(kernelLength-1)/2));
      xoffsets = ...
      xoffsets((kernelLength+1)/2:(length(xoffsets)-(kernelLength-1)/2));
      yoffsets = ...
      yoffsets((kernelLength+1)/2:(length(yoffsets)-(kernelLength-1)/2));
   else
      thetas = thetas(kernelLength/2+1:(length(thetas)-(kernelLength/2-1)));
      xoffsets = ...
      xoffsets(kernelLength/2+1:(length(xoffsets)-(kernelLength/2-1)));
      yoffsets = ...
      yoffsets(kernelLength/2+1:(length(yoffsets)-(kernelLength/2-1)));
   end

   %% Only integer pixel movement is allowed
   %%

   figure;
   subplot(311); plot(theta/pi*180,'-o');
   hold on; plot(thetas/pi*180,'r-s'); ylabel('Angle of rotation (^o)');
   legend('original','smoothed');
```

```
      hold off
      subplot(312); plot(xoffset,'-o');
      hold on; plot(xoffsets,'r-s'); ylabel('x_{offset}');
      legend('original','smoothed');
      hold off
      subplot(313); plot(yoffset,'-o');
      hold on; plot(yoffsets,'r-s'); ylabel('y_{offset}');
      legend('original','smoothed');
      hold off

      %% Add new fields to sequence
      smoothedAlpha = cos(thetas).*sqrt(alpha.^2+beta.^2);
      smoothedBeta = sin(thetas).*sqrt(alpha.^2+beta.^2);
      smoothedXoffset = xoffsets;
      smoothedYoffset = yoffsets;

      sequence.smoothedPara = ...
      [smoothedAlpha' smoothedBeta' smoothedXoffset' smoothedYoffset'];

otherwise

      error('Motion Model Not supported');

end
```

motionCorrection.m

```
function sequence = motionCorrection(sequence)
%% function sequence = motionCorrection(sequence)
%% Purpose : determine whether the motion is intentional and
%%           smooth out/remove unwanted motion (vectors)
%% INPUT   : sequence -- structure
%% OUTPUT  : sequence -- structure
%%
%% Author  : T. Chen
%% Date    : 02/29/2000
%%

%% Assign local variables
%%
[nRows,nCols,nFrames] = size(sequence.originalYPlane);
originalYPlane = sequence.originalYPlane;
originalIPlane = sequence.originalIPlane;
originalQPlane = sequence.originalQPlane;

stabilizedYPlane = originalYPlane;
stabilizedIPlane = originalIPlane;
stabilizedQPlane = originalQPlane;

switch sequence.estimatedMotion(1).method,
   case 'block',

     smoothedMVs = sequence.smoothedMVs;
```

```
  originalMVs = smoothedMVs;
  for i = 1:nFrames-1,
      originalMVs(i,:) = sequence.estimatedMotion(i).motionVector;
  end

%% Motion Correction
%%
  threshold = sequence.MCpara.dThreshold;   % threshold for synchronization
  resetLength = sequence.MCpara.resetLength; % make sure synchronization
                                             % occurs at least
                                             % every resetLength frames
  count = 0;
  finalMVs = smoothedMVs;  % resulting MVs with the effect of
                           % synchronization included, it should be
                           % approximately the same as smoothedMVs

  for i = 1:nFrames-1,
    blockSize = sequence.estimatedMotion(i).relevantInfo(1);
    %% calculate the motion vectors difference
    %%
    if i > 1,
       MVsDifference = ...
       sum(abs(sum(originalMVs(1:i-1,:))-sum(finalMVs(1:i-1,:))).^2);
    end

  %% determine whether synchronization should take place
  %%
    if (i == 1) | (MVsDifference > threshold^2 & count < resetLength),
    % if no
       stabilizedYPlane(:,:,i+1) = ...
           motionCompensation(stabilizedYPlane(:,:,i),...
                              originalYPlane(:,:,i+1),...
                              originalMVs(i,:),smoothedMVs(i,:));
       stabilizedIPlane(:,:,i+1) = ...
           motionCompensation(stabilizedIPlane(:,:,i),...
                              originalIPlane(:,:,i+1),...
                              originalMVs(i,:),smoothedMVs(i,:));
       stabilizedQPlane(:,:,i+1) = ...
           motionCompensation(stabilizedQPlane(:,:,i),...
                              originalQPlane(:,:,i+1),...
                              originalMVs(i,:),smoothedMVs(i,:));
       count = count + 1;
    else      % if yes
       fprintf('synchronization occurs at frame %d\n',i);
       stabilizedYPlane(:,:,i+1) = ...
           motionCompensation(originalYPlane(:,:,i),...
                              originalYPlane(:,:,i+1),...
                              originalMVs(i,:),smoothedMVs(i,:));
       stabilizedIPlane(:,:,i+1) = ...
           motionCompensation(originalIPlane(:,:,i),...
                              originalIPlane(:,:,i+1),...
                              originalMVs(i,:),smoothedMVs(i,:));
       stabilizedQPlane(:,:,i+1) = ...
```

```
             motionCompensation(originalQPlane(:,:,i),...
                                originalQPlane(:,:,i+1),...
                                originalMVs(i,:),smoothedMVs(i,:));

         count = 0;
         finalMVs(i,:) = smoothedMVs(i,:) + ...
                         sum(originalMVs(1:i-1,:)) - ...
                         sum(smoothedMVs(1:i-1,:));
      end
    end

    %% Add new fields to sequence
    %%
    sequence.stabilizedYPlane = stabilizedYPlane;
    sequence.stabilizedIPlane = stabilizedIPlane;
    sequence.stabilizedQPlane = stabilizedQPlane;
    sequence.finalMVs = finalMVs;

    %% Compare the three different motion vectors
    %%
    figure;
    subplot(211);
    plot(originalMVs(:,1),'-o'); hold on;
    plot(smoothedMVs(:,1),'r-s'); hold on;
    plot(finalMVs(:,1),'g-^'); hold off;
    temp = axis; axis([temp(1:2) -15 15]);
    xlabel('Frames'); ylabel('u (horizontal direction)');
    legend('original MV','smoothed MV','smoothed MV w/ error control');
    subplot(212);
    plot(originalMVs(:,2),'-o'); hold on;
    plot(smoothedMVs(:,2),'r-s'); hold on;
    plot(finalMVs(:,2),'g-^'); hold off;
    temp = axis; axis([temp(1:2) -15 15]);
    xlabel('Frames'); ylabel('v (vertical direction)');
    legend('original MV','smoothed MV','smoothed MV w/ error control ');

  %% End of case 'block'

  case 'affine',

    smoothedAngles = ...
atan(sequence.smoothedPara(:,2)./sequence.smoothedPara(:,1))/pi*180;
    originalAngles = smoothedAngles;
    smoothedOffsets = sequence.smoothedPara(:,3:4);
    originalOffsets = smoothedOffsets;
    for i = 1:nFrames-1,
        originalAngles(i) = ...
    atan(sequence.estimatedMotion(i).relevantInfo(2)/...
         sequence.estimatedMotion(i).relevantInfo(1))/pi*180;
        originalOffsets(i,:) = ...
    sequence.estimatedMotion(i).relevantInfo(3:4)';
    end
```

```
     %% Motion Correction
     %%
     aThreshold = sequence.MCpara.aThreshold;
     dThreshold = sequence.MCpara.dThreshold;
     resetLength = sequence.MCpara.resetLength;
     count = 0;
     finalAngles = smoothedAngles;
     finalOffsets = smoothedOffsets;

     %temp = [];
     %temp1 = [];
     for i = 1:nFrames-1,
        para = sequence.estimatedMotion(i).relevantInfo;
        matAtoB = [para(1) -para(2); para(2) para(1)];
        matAtoB_inv = inv(matAtoB);
        affAtoB = [para(3); para(4)];
        affAtoB_inv = -matAtoB_inv*affAtoB;

if i > 1,
          OffsetDifference = ...
        sum(abs(sum(originalOffsets(1:i-1,:))-sum(finalOffsets(1:i-1,:))).^2);
          AngleDifference = ...
      abs(sum(originalAngles(1:i-1)) - sum(finalAngles(1:i-1)));
%  temp = [temp OffsetDifference];
%  temp1 = [temp1 AngleDifference];
end

if i == 1 | (AngleDifference < aThreshold & OffsetDifference ...
      < dThreshold^2 & count >= 3) | count >= resetLength ,
   fprintf('synchronization occurs at frame %d\n',i);
           stabilizedYPlane(:,:,i+1) = ...
                 affineRec(matAtoB_inv, affAtoB_inv, ...
                 originalYPlane(:,:,i),originalYPlane(:,:,i+1));
           stabilizedIPlane(:,:,i+1) = ...
                 affineRec(matAtoB_inv, affAtoB_inv, ...
                 originalIPlane(:,:,i),originalIPlane(:,:,i+1));
           stabilizedQPlane(:,:,i+1) = ...
                 affineRec(matAtoB_inv, affAtoB_inv, ...
                 originalQPlane(:,:,i),originalQPlane(:,:,i+1));
   count = 0;
   finalAngles(i) = smoothedAngles(i) + ...
                          sum(originalAngles(1:i-1)) - ...
                          sum(smoothedAngles(1:i-1));
           finalOffsets(i,:) = smoothedOffsets(i,:) + ...
                          sum(originalOffsets(1:i-1,:)) - ...
                          sum(smoothedOffsets(1:i-1,:));

else
   stabilizedYPlane(:,:,i+1) = ...
                 affineRec(matAtoB_inv, affAtoB_inv, ...
                 stabilizedYPlane(:,:,i),originalYPlane(:,:,i+1));
           stabilizedIPlane(:,:,i+1) = ...
                 affineRec(matAtoB_inv, affAtoB_inv, ...
                 stabilizedIPlane(:,:,i),originalIPlane(:,:,i+1));
           stabilizedQPlane(:,:,i+1) = ...
                 affineRec(matAtoB_inv, affAtoB_inv, ...
                 stabilizedQPlane(:,:,i),originalQPlane(:,:,i+1));
   count = count + 1;
end   %% End of 'if' statement
     end %% End of 'for' statement

     %figure;
     %subplot(211); plot(temp1,'-o');
     %subplot(212); plot(temp,'-o');
     %% Add new fields to sequence
     %%
     sequence.stabilizedYPlane = stabilizedYPlane;
     sequence.stabilizedIPlane = stabilizedIPlane;
     sequence.stabilizedQPlane = stabilizedQPlane;

     %% Compare the different motion parameters
     %%
     figure;
     subplot(311);
     plot(originalAngles,'-o'); hold on;
     plot(smoothedAngles,'r-s'); hold on;
     plot(finalAngles,'g-^'); hold off;
     xlabel('Frames'); ylabel('\theta');
     legend('original','smoothed','smoothed w/ error control');
     subplot(312);
     plot(originalOffsets(:,1),'-o'); hold on;
     plot(smoothedOffsets(:,1),'r-s'); hold on;
     plot(finalOffsets(:,1),'g-^'); hold off;
     xlabel('Frames'); ylabel('x_{offset}');
     legend('original','smoothed','smoothed w/ error control');
     subplot(313);
     plot(originalOffsets(:,2),'-o'); hold on;
     plot(smoothedOffsets(:,2),'r-s'); hold on;
     plot(finalOffsets(:,2),'g-^'); hold off;
     xlabel('Frames'); ylabel('y_{offset}');
     legend('original','smoothed','smoothed w/ error control');

   %% End of case 'affine'

   otherwise

     error('Not supported');

end %% End of 'switch' statement
```

sequenceDisplay.m

```
function M = sequenceDisplay(sequence)
%% function M = sequenceDisplay(sequence)
%% Purpose : Display the original and smoothed video sequence
%% INPUT   : sequence -- structure
%% OUTPUT  : M -- array representing the video sequences
%% Author  : T. Chen
%% Date    : 03/08/2000
%%

rgb2yiq = [0.3 0.59 0.11; 0.6 -0.28 -0.32; 0.21 -0.52 0.31];
yiq2rgb = inv(rgb2yiq);

[nRows,nCols,nFrames] = size(sequence.originalYPlane);

figure;
M = [];
M = moviein(nFrames);
if strcmp(sequence.displayOption,'color'),
   for i = 1:nFrames,
       originalYPlane = sequence.originalYPlane(:,:,i);
       originalIPlane = sequence.originalIPlane(:,:,i);
       originalQPlane = sequence.originalQPlane(:,:,i);
       originalFrame(:,:,1) = yiq2rgb(1,1)*originalYPlane + ...
                              yiq2rgb(1,2)*originalIPlane + ...
                              yiq2rgb(1,3)*originalQPlane;
       originalFrame(:,:,2) = yiq2rgb(2,1)*originalYPlane + ...
                              yiq2rgb(2,2)*originalIPlane + ...
                              yiq2rgb(2,3)*originalQPlane;
       originalFrame(:,:,3) = yiq2rgb(3,1)*originalYPlane + ...
                              yiq2rgb(3,2)*originalIPlane + ...
                              yiq2rgb(3,3)*originalQPlane;
       originalFrame = max(originalFrame,0);
       originalFrame = originalFrame/max(originalFrame(:));

       stabilizedYPlane = sequence.stabilizedYPlane(:,:,i);
       stabilizedIPlane = sequence.stabilizedIPlane(:,:,i);
       stabilizedQPlane = sequence.stabilizedQPlane(:,:,i);
       stabilizedFrame(:,:,1) = yiq2rgb(1,1)*stabilizedYPlane + ...
                                yiq2rgb(1,2)*stabilizedIPlane + ...
                                yiq2rgb(1,3)*stabilizedQPlane;
       stabilizedFrame(:,:,2) = yiq2rgb(2,1)*stabilizedYPlane + ...
                                yiq2rgb(2,2)*stabilizedIPlane + ...
                                yiq2rgb(2,3)*stabilizedQPlane;
       stabilizedFrame(:,:,3) = yiq2rgb(3,1)*stabilizedYPlane + ...
                                yiq2rgb(3,2)*stabilizedIPlane + ...
                                yiq2rgb(3,3)*stabilizedQPlane;
       stabilizedFrame = max(stabilizedFrame,0);
       stabilizedFrame = stabilizedFrame/max(stabilizedFrame(:));

       compareFrame = [originalFrame stabilizedFrame];
       imshow(compareFrame); axis image; truesize;
       M(:,i) = getframe;
   end

else
   for i = 1:nFrames,
       originalFrame = sequence.originalYPlane(:,:,i);
```

```
        stabilizedFrame = sequence.stabilizedYPlane(:,:,i);

        compareFrame = [originalFrame stabilizedFrame];
        colormap(gray(256)); imagesc(compareFrame); axis image; truesize;
        M(:,i) = getframe;
    end
end
```

readDatafile.m

```
function outImage = readDatafile(filename);
% function outImage = readDatafile(filename)
% Purpose : loads the image specified by filename into matrix outImage
%
% Argument : INPUT : filename -- string
%            OUTPUT: outImage -- matrix of size NxMx3 in RGB color space
%
x = [];
fid = fopen(filename,'r');
x = fread(fid);
fclose(fid);

x = reshape(x,3,320,240);
xs = zeros(240,320,3);
xs(:,:,1) = squeeze(x(1,:,:))';
xs(:,:,2) = squeeze(x(2,:,:))';
xs(:,:,3) = squeeze(x(3,:,:))';

outImage = xs;
```

motionEstimation.m

```
function estimatedMotion = motionEstimation(previousFrame,currentFrame,sequence)
%% function estimatedMotion = motionEstimation(previousFrame,currentFrame,sequence)
%% Purpose : perform motion estimation using appropriate motion
%%           estimation methods
%% INPUT   : previousFrame -- array [nRows x nCols]
%%           currentFrame  -- array [nRows x nCols]
%%           sequence      -- structure
%% OUTPUT  : estimatedMotion -- structure
%%                    fields : method -- string
%%                             relevantInfo -- array
%%                             motionVector -- array [u v];
%%                             MVmap -- structure with fields U and V
%% Author  : T. Chen
%% Date    : 02/24/2000
%%

%% Need to decide whether the motion fits in translational model or not
%%
motionModel = selectMotionModel(previousFrame,currentFrame,sequence);

%% Assign local variables
```

```
%%
U = motionModel.MVmap.U;
V = motionModel.MVmap.V;
blockSize = 16;            % block size in block-matching ME
searchRange = 15;          % search range
searchType = 1;            % '1' == full search

%% Remove the MVs for the blocks on the frame boundary (they are all
%% pre-set to zero)
%%
[s1,s2] = size(U);
temp = U(2:s1-1,2:s2-1);
Us = reshape(temp,prod(size(temp)),1);
temp = V(2:s1-1,2:s2-1);
Vs = reshape(temp,prod(size(temp)),1);

%% Generate appropriate output structure
%%
switch motionModel.method

   case 'block',

      %% Obtain the appropriate global motion transformation parameters
      %%
      u = round(median(Us));
      v = round(median(Vs));

      %% Generate the estimatedMotion structure
      %%
      estimatedMotion.method = 'block';
      estimatedMotion.relevantInfo = [blockSize searchRange 1];
      estimatedMotion.motionVector = [u v];
      estimatedMotion.MVmap.U = U;
      estimatedMotion.MVmap.V = V;

   case 'affine',

      %% Obtain the appropriate global motion transformation parameters
      %%
      parameters = affinePara(previousFrame,currentFrame,U,V);
      u = round(median(Us));
      v = round(median(Vs));

      %% Generate the estimatedMotion structure
      %%
      estimatedMotion.method = 'affine';
      estimatedMotion.relevantInfo = parameters;
      estimatedMotion.motionVector = [u v];   % Not required,
                                              % but included anyway
      estimatedMotion.MVmap.U = U;
      estimatedMotion.MVmap.V = V;

   otherwise,
      error('Unsupported Motion Estimation Model!');

end
```

selectMotionModel.m

```
function motionModel = selectMotionModel(previousFrame,currentFrame,sequence)
%% function motionModel = selectMotionModel(previousFrame,currentFrame,sequence)
%% Purpose : distinguish rotation from translation (for now)
%% INPUT   : previousFrame -- array [nRows x nCols]
%%           currentFrame  -- array [nRows x nCols]
%%           sequence -- structure
%% OUTPUT  : motionModel -- structure
%%
%% Author  : T. Chen
%% Date    : 03/07/2000
%%

%% Assign local variables
%%
[nRows,nCols] = size(previousFrame);
varThreshold = 5;
searchRange = 15;

%% First need to obtain local optical flow information.
%% by default use the standard block-matching, full search and MSE criterion
%%
blockSize = 16;
searchRange = 15;
searchType = 1;  % '1' == full search

U = zeros(nRows/blockSize,nCols/blockSize);
V = U;
%% Need to speed up considerably :
%%
for i=2:nRows/blockSize-1,
  for j=2:nCols/blockSize-1,
%%    fprintf('\nME for block %d-%d\n',i,j);
    [u,v] = fullBlockME(i,j,blockSize,searchRange,previousFrame,currentFrame);
    U(i,j) = u;
    V(i,j) = v;
  end
end

%% Testing scripts :
%% display the motion field
%%
X = 1:blockSize:nCols;         % horizontal direction
Y = 1:blockSize:nRows;         % vertical direction
figure(1); quiver(X,Y,U,V); title('Motion Vector Field');
axis ij; axis equal; axis tight
%% display histogram of the motion vectors
```

```
%%
%% First remove the MVs on the frame boundary
%%
[s1,s2] = size(U);
temp = U(2:s1-1,2:s2-1);
Us = reshape(temp,prod(size(temp)),1);
temp = V(2:s1-1,2:s2-1);
Vs = reshape(temp,prod(size(temp)),1);
figure(2);
subplot(211); hist(Us,-searchRange:searchRange);
xlabel('Frame'); ylabel('u (horizontal direction)');
subplot(212); hist(Vs,-searchRange:searchRange);
ylabel('Frame'); ylabel('v (vertical direction)');

%% Determine the appropriate motion model :
%% pre-processing to filter out the extreme outliners
%% Here assume the extreme outliners are the ones with less than
%% certain percentage of probability
%%
prefilterThreshold = 0.02;
Us_median = median(Us);
Vs_median = median(Vs);
num = length(Us);
for i = -searchRange:searchRange,
    index = find(Us==i);
    if length(index)/num < prefilterThreshold,
       Us(index) = Us_median;
     end
     index = find(Vs==i);
     if length(index)/num < prefilterThreshold,
        Vs(index) = Vs_median;
     end
end

%% create structure motionModel
%% Notice right now this decision making process only works well when
%% the frames are not motion blurred so the MEs are accurate.
%% So ideally there should be a motion-deblurring routine before the
%% block-matching motion estimation
%% However, since I do not have a chance to implement such a function
%% right now, and I did not hand edit the video sequence and threw out
%% the blurred frames, I will cheat a little bit here.
%% Actually, what I really want to do here is to come up an algorithm
%% that can be "smart" enough to detect the blurred frames and
%% consequently remove their effects. Also the case where occasionally
%% the wrong motion type is mistakenly determined should be able to
%% be corrected somehow by exploiting the motion characteristics of
%% the overall video sequence. For this to work, we still assume that
%% the same type of motions is always associated with a cluster of frames
%%
%%
if strcmp(sequence.name,'lamp') | strcmp(sequence.name,'car') | ...
   strcmp(sequence.name,'kids'),
   motionModel.method = 'block';
elseif var(Us) <= varThreshold & var(Vs) <= varThreshold,
   motionModel.method = 'block';
else
   motionModel.method = 'affine';
   %% regenerate U and V by applying weighted MSE
   %%
   U = zeros(nRows/blockSize,nCols/blockSize);
   V = U;
   for i=2:nRows/blockSize-1,
      for j=2:nCols/blockSize-1,
         [u,v] = ...
   fullBlockME_weighted(i,j,blockSize,searchRange,previousFrame,currentFrame);
         U(i,j) = u;
         V(i,j) = v;
      end
   end
end

motionModel.MVmap.U = U;
motionModel.MVmap.V = V;
```

fullBlockME.m

```
function [u,v] = fullBlockME(i,j,blk_length,search_range,f1_Y,f2_Y)
%% function [u,v] = fullBlockME(i,j,blk_length,search_range,f1_Y,f2_Y)
%% Purpose : returns the motion vector for block i-j using MSE
%%           criterion
%% INPUT   : i,j -- block index
%%           blk_length -- size of block
%%           search_range -- [-search_range search_range]
%%           f1_Y -- previous frame
%%           f2_Y -- current frame
%% OUTPUT  : [u, v] -- motion displacements in horizontal and vertical
%%                     direction, respectively
%% Author  : T. Chen
%% Date    : 02/24/2000
%%

sqrtMSEs = [];
%% store the target block into a 16x16 matrix block2
%%
block2 = f2_Y(blk_length*(i-1)+1:blk_length*i,....
              blk_length*(j-1)+1:blk_length*j);
vector2 = reshape(block2,1,blk_length^2);  % reshape to 1-D array

%% search and calculate MSE and store MSE results into MSEs
%%
for y_offset = -search_range:search_range,
  for x_offset = -search_range:search_range,
    block1 = f1_Y(blk_length*(i-1)+1+y_offset:blk_length*i+y_offset,....
              blk_length*(j-1)+1+x_offset:blk_length*j+x_offset);
    vector1 = reshape(block1,1,blk_length^2);
    sqrtMSEs = [sqrtMSEs norm(vector2-vector1)];
  end
end

[temp, index] = min(sqrtMSEs);
%% calculate motion vector
v = -search_range + floor((index(1)-1)/(2*search_range+1));
u = -search_range + rem(index(1)-1, 2*search_range+1);
```

fullBlockME_weighted.m

```
function [u,v] = fullBlockME_weighted(i,j,blk_length,search_range,f1_Y,f2_Y)
%% function [u,v] = fullBlockME_weighted(i,j,blk_length,search_range,f1_Y,f2_Y)
%% Purpose : returns the motion vector for block i-j using weighted MSE
%%           criterion
%% INPUT   : i,j -- block index
%%           blk_length -- size of block
%%           search_range -- [-search_range search_range]
%%           f1_Y -- previous frame
%%           f2_Y -- current frame
%% OUTPUT  : [u, v] -- motion displacements in horizontal and vertical
%%                     direction, respectively
%% Author  : T. Chen
%% Date    : 02/24/2000
%%

%% Generate weighting function
temp = linspace(1,2,blk_length/2);
r = [temp fliplr(temp)];
c = r';
w = c*r;
w = w/max(w(:));
w = reshape(w,1,blk_length^2);
w_sqrt = sqrt(w);

sqrtMSEs = [];
%% store the target block into a 16x16 matrix block2
%%
block2 = f2_Y(blk_length*(i-1)+1:blk_length*i,....
              blk_length*(j-1)+1:blk_length*j);
vector2 = reshape(block2,1,blk_length^2);  % reshape to 1-D array

%% search and calculate MSE and store MSE results into MSEs
%%
for y_offset = -search_range:search_range,
  for x_offset = -search_range:search_range,
    block1 = f1_Y(blk_length*(i-1)+1+y_offset:blk_length*i+y_offset,....
              blk_length*(j-1)+1+x_offset:blk_length*j+x_offset);
    vector1 = reshape(block1,1,blk_length^2);
    sqrtMSEs = [sqrtMSEs norm((vector2-vector1).*w_sqrt)];
  end
end
```

```
[temp, index] = min(sqrtMSEs);
%% calculate motion vector
v = -search_range + floor((index(1)-1)/(2*search_range+1));
u = -search_range + rem(index(1)-1, 2*search_range+1);
```

affinePara.m

```
function parameters = affinePara(previousFrame,currentFrame,U,V)
%% function motionModel = affinePara(previousFrame,currentFrame,U,V)
%% Purpose : perform motion estimation using affine motion model on
%%           motion vectors from block matching motion estimation
%% INPUT   : previousFrame -- array [nRows x nCols]
%%           currentFrame  -- array [nRows x nCols]
%%           U             -- array [mRows x mCols]
%%           V             -- array [mRows x mCols]
%% OUTPUT  : parameters -- array
%%
%% Author  : T. Chen
%% Date    : 03/03/2000
%%

%% Assign local variables
%%
[nRows,nCols] = size(previousFrame);
[mRows,mCols] = size(U);
blockSize = nRows/mRows;
offset = round(blockSize/2);

currentXYs = [];
Hxy = [];
for i = 2:mRows-1,
  y = blockSize*(i-1)+offset;
  for j = 2:mCols-1,
    x = blockSize*(j-1)+offset;
    Y = y + V(i,j);
    X = x + U(i,j);
    currentXYs = [currentXYs; X; Y];
    Hxy = [Hxy; x -y 1 0; y x 0 1];
  end
end

parameters = pinv(Hxy)*currentXYs;
```

motionCompensation.m

```
function outFrame = motionCompensation(inFrame1, inFrame2, MVr, MVs)
%% function outFrame = motionCompensation(inFrame, MV)
%% Purpose : Motion compensation on current frame using given MVs
%% INPUT   : inFrame -- [nRows x nCols]
%%           MV      -- [u v]
%% OUTPUT  : outFrame -- motion compensated frame
%%
%% Author  : T. Chen
%% Date    : 02/29/2000
%%

nRows = size(inFrame1,1);
nCols = size(inFrame1,2);
u = MVs(1);
v = MVs(2);

ur = MVr(1);
vr = MVr(2);
uf = u - ur;
vf = v - vr;
outFrame = inFrame2;
%outFrame = inFrame1;

if uf >= 0 & vf >= 0,
  outFrame(1:nRows-vf,1:nCols-uf) = inFrame2(1+vf:nRows,1+uf:nCols);
elseif uf >=0 & vf < 0,
  outFrame(1-vf:nRows,1:nCols-uf) = inFrame2(1:nRows+vf,1+uf:nCols);
elseif uf < 0 & vf >= 0,
  outFrame(1:nRows-vf,1-uf:nCols) = inFrame2(1+vf:nRows,1:nCols+uf);
else
  outFrame(1-vf:nRows,1-uf:nCols) = inFrame2(1:nRows+vf,1:nCols+uf);
end


if u >= 0 & v >= 0,
  outFrame(1:nRows-v,1:nCols-u) = inFrame1(1+v:nRows,1+u:nCols);
elseif u >=0 & v < 0,
  outFrame(1-v:nRows,1:nCols-u) = inFrame1(1:nRows+v,1+u:nCols);
elseif u < 0 & v >= 0,
  outFrame(1:nRows-v,1-u:nCols) = inFrame1(1+v:nRows,1:nCols+u);
else
  outFrame(1-v:nRows,1-u:nCols) = inFrame1(1:nRows+v,1:nCols+u);
end
```

affineRec.m

```
function estimatedB = affineRec(matAtoB_inv,affAtoB_inv,A,B);
%% function estimatedB = affineRec(paraAtoB,A,B);
%% calculate the predicted B image from image A and affine
%% model parameters
%%

[nRows,nCols] = size(A);
estimatedB = B;

for i = 1:nRows,
    for j = 1:nCols,
       xy = matAtoB_inv*[i;j] + affAtoB_inv;
       xy = round(xy);
       %% take care of points outside the image boundary
       if xy(1) <= nRows & xy(2) <= nCols & min(xy) > 0,
            estimatedB(i,j) = A(xy(1),xy(2));
        end
    end
end
```