Real-Time Digital Image Stabilization

Alan C. Brooks

***Abstract*—In this paper, digital image stabilization (DIS) algorithms are investigated with respect to implementation feasibility and complexity. One of the most computationally efficient algorithms was simulated to produce qualitative results to characterize performance.**

***Index Terms*—Image Registration, Image Stabilization, Motion Estimation**

I. Introduction

DIGITAL image sequences captured from physical imaging devices often display unwanted high frequency jittering motion. The amount of motion present in an image sequence depends on the physics of the motion capture device relative to objects in the captured sequence. The depth of the scene and the instability in the imager's mount (dependant on the mount's weight, inertia, balance) combine to create undesired global motion. While some other approaches to stabilization such as Steady-Cam and optical systems have been very popular in the past, efficient real-time digital implementations are now becoming common.

A digital image stabilization system first estimates unwanted motion and then applies corrections to the image sequence. As described in [1] on pp. 497-511, image motion can be estimated using spacio-temporal or region matching approaches. Spacio-temporal approaches include parametric block matching [2], direct optical flow estimation [3], and a least mean-square error matrix inversion approach [4]. Region matching methods include bit-plane matching [5], point-to-line correspondence [6], feature tracking [7], pyramidal approaches [8], and block matching [9].

In this paper, I first analyze the computational efficiency of several DIS approaches that are intended to remove unwanted global motion from an image sequence. Then, results from a computer simulation are presented, illustrating the performance of one of the most efficient algorithms.

This paper is organized as follows. DIS algorithms efficiency is studied in Section II. In Section III, a description of the Matlab simulation of the Gray-Coded Bit-Plane Matching (GC-BPM) algorithm is presented. The results of this simulation are detailed in Section IV and Section V gives some conclusions.

A. C. Brooks is a part-time graduate student in the Electrical Engineering Department, Northwestern University, Evanston, IL 60208 USA, (e-mail: a-brooks@northwestern.edu).

II. Efficiency of Stabilization Algorithms

Much can be inferred about the computational complexity of the algorithms in Table I by considering the motion correction transform each approach is able to support. Because the other transforms are all subsets of a full affine transform, less work is generally required to estimate any individual transform parameters.

When comparing the algorithmic efficiency of alternate solutions to a problem, it is important to consider the specific implementation options. Software implementations might have access to libraries of optimized high-level mathematical functions whereas a *real-time* hardware implementation would be created with simpler basic elements, such as Boolean operators. It is likely that hardware motion capture systems would include image stabilization in the future, so this study focuses on simulation and implementation of an algorithm likely to be efficiently realized in a hardware design. Since most of its calculations are done using simple Boolean exclusive-or operators easily implemented in hardware, the GC-BPM algorithm was chosen for further investigation.

TABLE I
Digital Image Stabilization Algorithms

Detection Method	Resolution	Transform	Ref.
Parametric Block Matching	Sub-pixel	Translation, Rotation	[2]
Optical Flow Estimation	Sub-pixel	Translation, Rotation	[3]
Linear Region Matching	Sub-pixel	Affine*	[4]
Gray-Coded Bit Plane Matching	Pixel	Translation	[5]
Point-to-Line Correspondence	Pixel	Warping	[6]
Feature Tracking	Pixel	Warping	[7]
Pyramidal/Decimated	Pixel	Warping	[8],[9]
Block Matching	Pixel	Translation	[10]

Detection refers to the approach used to estimate the image sequence's motion. Transform is the algorithm's motion correction capability.

*An *affine* transform combines translational, rotational, and warping transforms into a unified matrix representation.

III. GC-BPM Algorithm Simulation

Generic DIS systems are commonly broken into the following major blocks: local motion estimation, global motion estimation, motion smoothing (i.e. filtering or integration), and motion compensation.

The GC-BPM algorithm can be broken down into a preprocessing step, followed by four major blocks (see Fig. 1). The algorithm begins by pre-process the image data to

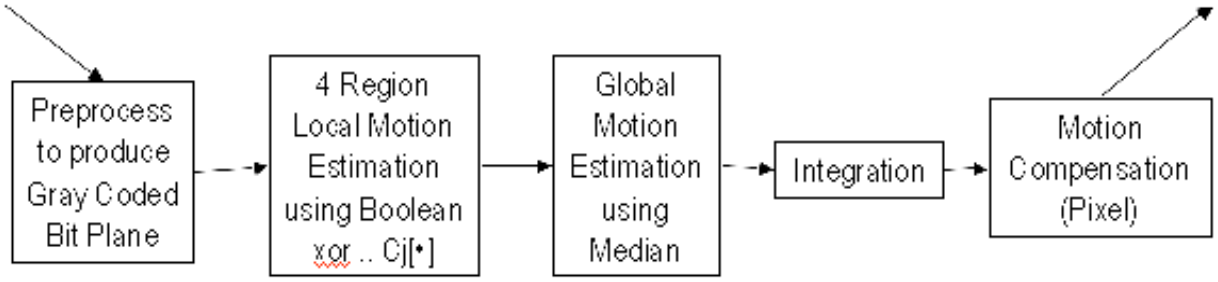

Fig. 1. Block diagram of the GC-BPM image stabilization algorithm.

produce gray-coded bit-planes from a standard binary representation. Let the *t*-th image from a sequence of K-bit grayscale images be denoted as

$$f^t(x,y) = a_{K-1}2^{K-1} + a_{K-2}2^{K-2} + \ldots + a_0 2^0. \quad (1)$$

Let the bit planes of this image are denoted as $b_0^t(x,y)$ to $b_7^t(x,y)$ (least-significant to most-significant bit, respectively). Then, as detailed in [5], the 8-bit gray code g_{K-1} to g_0 can be computed as

$$g_{K-1} = a_{K-1}$$
$$g_k = a_k \oplus a_{k+1},\ 0 \le k \le K-2 \quad (2)$$

where a_k are the standard binary coefficients. The gray coding is important in this algorithm because it allows the next block to estimate motion using a single bit-plane by encoding most of the useful image information into a few of the planes. With the gray code, small changes in gray level yield a small, uniform change in the binary digits representing the intensity.

For this reason, the next GC-BPM block can estimate local motion using an extremely efficient Boolean binary operator to compare the gray-coded bit-plane of the previous image to the current one. Let the error operator to be minimized be denoted as

$$\Box(m,n) = 1/(W^2) \Box\Box\ g_k^t(x,y) \oplus g_k^{t-1}(x+m,y+n),$$
$$, -p \le m,n \le p \quad (3)$$

where W is the sliding block used to search and p is the number of pixels to search over, as illustrated in Fig. 2.

By minimizing this error operator, the resulting values of (m,n) produce a local motion vector for each of the four sub-images depicted in Fig. 2. Then, the four local motion vectors along with the previous global motion vector are subject to a median operator to produce the current global motion vector estimate, V_g^t.

Then, the global motion estimate is passed through a filter that is tuned to let intentional camera motion (e.g. intentional panning) be preserved while removing the undesirable high frequency motion. The final filtered motion estimate is then compensated for by shifting the current frame by an integer number of pixels in the opposite direction of the motion.

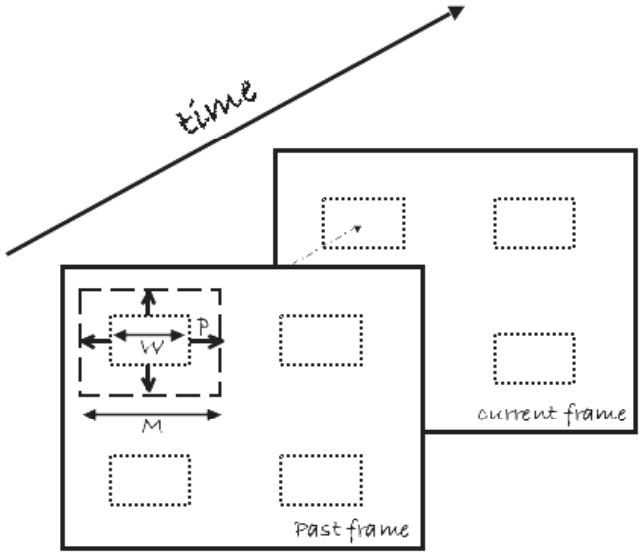

Fig. 2. Illustration of block searching method for comparing previous frame to current from for motion estimation.

Despite the fact that GC-BPM, as formulated, works well for translational (horizontal and vertical) motion, it includes no estimation or compensation capability for rotational or zooming motion. I experimented with adding a rotational motion detector and compensator to GC-BPM using an approach [2] that employs the spatio-temporal relationship to solve for rotational and translational motion. However, this approach did not work well with only four local motion vectors as it is intended to be used with the optical flow of each image pixel.

IV. Simulation Results

The most interesting simulation results for this project were the stabilized videos. Side-by-side comparisons of these videos are available in a document supporting this paper, FinalPresentationResultsVid.mp4 (compressed with QuickTime to 4.1 MB using mpeg-4).

The simulation is written and optimized for performance in Matlab by storing all of the image data as unsigned 8-bit integers and applying fast vector operations as much as possible (see source code in the Appendix). The Matlab code is able to process the 256x256 grayscale image sequences at a rate of 3 frames per second on a 1 Ghz Pentium 3 machine.

Clearly, more efficient implementations both in lower-level software or in hardware could readily achieve *real-time* rates of 15-30 frames per second.

Another useful way to visualize the image stabilization process is by looking at plots of the estimated motion vectors. If one is adding jitter to images after they have been digitized, these plots can include the "truth" data for comparison. For natural unstable image sequences, this visualization approach is useful for characterizing the amount of jitter present in a test sequence.

Fig. 3 shows the estimated global motion vectors for a test movie I captured and stabilized, "temple.avi." This sequence was shot from the 24th floor of a building, looking at some interesting architecture with a 26x optical zoom fully "zoomed in" on a Sharp consumer digital video camera. It is interesting to notice that while the total motion deviation over all 63 frames spans about +/- 25 pixels, the motion vectors do not change more than about 8 pixels from one frame to the next.

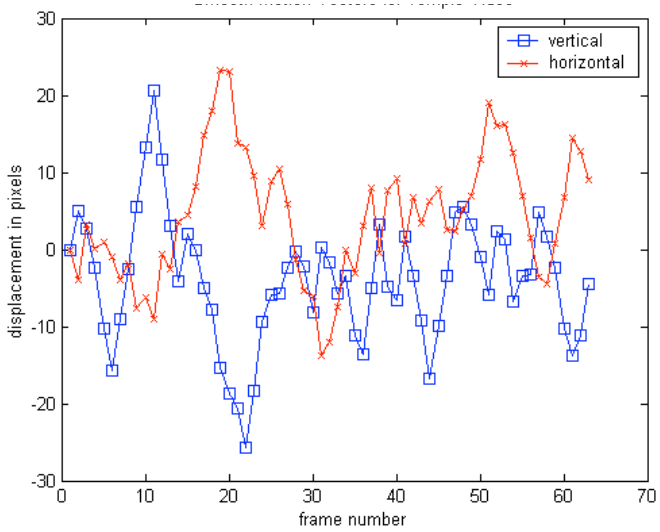

Fig. 3. Original vertical and horizontal estimated motion vectors.

Another way of visualizing the effectiveness of image stabilization is to difference consecutive frames of the original and stabilized image sequences. Fig. 4. shows an example of this technique, taking the difference between frames 16 and 17 and mapping the output values ranging from -255 to +255 to black and white, respectively, so that gray is zero.

(a) (b) (c)

Fig. 4. Difference between frames before and after motion correction. (a) original frame 16. (b) difference between original frames 16 and 17. (c) difference between GC-BPM motion-corrected frames 16 and 17.

Since there is quite a bit of motion between frames 16 and 17 of the "temple" sequence, Fig. 4 (b) clearly shows that the difference between original frames can be quite significant. Fig. 4 (c) shows that the motion compensation correctly aligned the images.

V. Conclusion

Gray-coded bit plane matching is a robust optimization of the maximum-likelihood block matching approach that affords efficient implementations that achieve quite acceptable qualitative performance.

Further interesting work might relate the performance quality of GC-BPM and other motion estimation algorithms to a bound based on human perception, allowing a more quantitative and useful description of the motion stabilization system's performance when used to produce image sequences for human viewing.

Acknowledgment

This project was done to fulfill the requirement for a computer project in EE 420 Digital Image Processing in winter 2003. The author would like to thank Prof. Aggelos K. Katsaggelos and Gordon Thompson for their encouragement, ideas, and advice.

References

[1] J. S. Lim. Two-Dimensional Signal and Image Processing. Prentice-Hall, Inc., Upper Saddle River, new Jersey. 1990.

[2] T. Chen. Video Stabilization Using a Block-Based Parametric Motion Model. Technical report, Stanford University, Information Systems Laboratory, Dept. of Electrical Engineering, Winter 2000.

[3] J. Chang, W. Hu, M. Cheng, and B. Chang. Digital image translational and rotational motion stabilization using optical flow technique. *IEEE Transactions on Consumer Electronics*, vol. 48, no. 1, pp. 108-115, Feb. 2002.

[4] C. Erdem and A. Erdem. An illumination invariant algorithm for subpixel accuracy image stabilization and its effect on MPEG-2 video compression. *Elsevier Signal Processing: Image Communication*, vol. 16, pp. 837-857, 2001.

[5] S. Ko, S. Lee, S. Jeon, and E. Kang. Fast digital image stabilizer based on gray-coded bit-plane matching. *IEEE Transactions on Consumer Electronics*, vol. 45, no. 3, pp. 598-603, Aug. 1999.

[6] M. Ben-Ezra, S. Peleg, and M. Werman. A Real-Time Video Stabilizer Based on Linear Programming.

[7] A. Censi, A. Fusiello, and V. Roberto. Image Stabilization by Features Tracking. Technical report, University of Udine, Machine Vision Laboratory, Dept. of Mathematics and Informatics, 1998.

[8] J. Jin, Z. Zhu, and G. Xu. Digital Video Sequence Stabilization Based on 2.5D Motion Estimation and Inertial Motion Filtering. *Real-Time Imaging*, vol. 7, pp. 357-365, 2001.

[9] J. Jin, Z. Zhu, and G. Xu. A Stable Vision System for Moving Vehicles. *IEEE Transactions on Intelligent Transportation Systems*, vol. 1, no. 1, pp. 32-39, Mar. 2000.

[10] K. Uomori, A. Morimura, H. Ishii, T. Sakaguchi, and Y. Kitamure. Automatic image stabilizing system by full-digital signal processing. *IEEE Transactions on Consumer Electronics, vol. 36, no. 3, ppl 510-519, Aug. 1990.*

APPENDIX

The Matlab code used to implement the simulation follows. This simulation runs on Matlab version 6 release 11.

A. A listing of the main test routine "imageStabilizeMain.m" follows.

```
% filename: imageStabilizeMain.m
% author: Alan Brooks
% description: dunno yet
% version history:
%   06-Mar-2003 created
%   08-Mar-2003 image I/O added
%   09-Mar-2003 v0_1 archived
%   10-Mar-2003 v0_2, v0_3 archived

function [] = imageStabilizeMain(fileName)
%profile on
% setup vars
if ~exist('fileName','var')
%fileName = 'flag.avi';
%fileName = 'temple.avi';
%fileName = 'elevator.avi';
%fileName = 'smoke_building.avi';
%fileName = 'cars_and_zoom.avi';
%fileName = 'shaky_math.avi';
%fileName = 'twist_couch.avi';
%fileName = 'building2_small_25fps.avi';
%fileName = 'Light_jitter_qt.avi';
fileName = 'Heavy_jitter_qt.avi';
%fileName = 'die_another_day_small_grey.avi';
end
nFrames = []; %20;%[]; % num desired frames to process ([] gets all)

% read file
mov = aviread(fileName);
movInfo = aviinfo(fileName);
nFrames = min([movInfo.NumFrames nFrames]);

% setup figure
H1 = figure; set(H1,'name','Original Movie')
scrz = get(0,'ScreenSize');
set(H1,'position',...    % [left bottom width height]
     [60 scrz(4)-100-(movInfo.Height+50) ...
         movInfo.Width+50 movInfo.Height+50]);

% play orig movie
movie(H1,mov,1,movInfo.FramesPerSecond,[25 25 0 0])
close(H1)

% convert from indexed image seq to grayscale double [0,255] ... make uint8 ??
M = uint8(zeros(movInfo.Height,movInfo.Width,nFrames));
for i = 1:nFrames
     M(:,:,i) = uint8(floor(256*ind2gray(mov(i).cdata,mov(i).colormap)));
end
%figure,imshow(M(:,:,i),[0 255]); % show last frame
```

```
% do GC-BPM stabilization (gather statistics)
  tic
[Ms,Va,Vg,V] = stabilizeMovie_GCBPM(M);
  t = toc; fprintf('%.2f seconds per frame\n',t/(nFrames-1));

% assemble for playback of final movie
H2 = figure; set(H2,'name','generating final movie ...')
for i = 1:length([Ms(1,1,:)])
     imshow(Ms(:,:,i),[0 255]);
  movStab(i) = getframe(H2);
end
close(H2)
H3 = figure; set(H3,'name','Final Stabilized Movie')
imshow(Ms(:,:,1),[0 255]);
curPos = get(H3,'position');
set(H3,'position',...    % [left bottom width height]
     [60 scrz(4)-100-(movInfo.Height+50) curPos(3:4)]);
movie(H3,movStab,1,movInfo.FramesPerSecond)

% save out final movie & workspace
movie2avi(movStab,[fileName(1:end-4) '_out.avi'], ...
     'fps',movInfo.FramesPerSecond,'compression','None');
save(sprintf('Wkspace_at_d%d-%02d-%02d_t%02d-%02d-%02d',fix(clock)))

%profile report runtime
return
```

B. A listing of the main computational routine "stabilizeMovie_GCBPM.m" follows.

```
% filename: stabilizeMovie_GCBPM.m
% author: Alan Brooks
% description: Given gray scale double matrix, M(width,height,frameNum),
%   this function stabilizes the sequence using the Grey-Coded Bit-Plane
%   Matching (GC-BPM) algorithm.
% version history:
%   08-Mar-2003 created
%   09-Mar-2003 tested and optimized
%   10-Mar-2003 output more vars
%   11-Mar-2003 added NSS log algorithm
%               (improve from 0.8 FPS to 3 FPS on 1GHz PIII)

function [Ms,Va,Vg,V] = stabilizeMovie_GCBPM(M)

% init global variables
debug_disp = 0;                 % display debugging plots

% Gray Coded Approach based on:
% [8] S. Ko, S. Lee, S. Jeon, and E. Kang. Fast digital image stabilizer
%     based on gray-coded bit-plane matching. IEEE Transactions on
%     Consumer Electronics, vol. 45, no. 3, pp. 598-603, Aug. 1999.
% ------------------------------------------------------------------------

% init algorithm specific variables (uint8's for speed)
bit = 5;                        % best grey code bit (det by paper)
N = 112; %64;                   % matching block size (assumed NxN square)
D1 = 0.95;                      % damping coeff (0 < D1 < 1)
logSearchEnable = 1;            % 1 = use logrigtmic 3-2-1 matching search
                                % 0 = use original exhaustive matching search
nSteps = 3;                     % number of steps for log search
                                %  (3 is pixel, 2 is 2 pixel, etc...)
rotEnable = 0;                  % 0 = use original translational GCBPM
                                % 1 = enable matrix-based rotation correction (Poor performance)
                                % 2 = enable simple/efficient rot. correction

[h,w,nFr] = size(M);            % height, width, # of frames
if ( ~rem(w,2) & ~rem(h,2) )
    S = uint8(zeros(h/2,w/2,2,4)); % subimage holder (must be even)
else
    error('video width/height must both be even # of pixels')
end
hw = waitbar(0,'Please wait...');
p = (h/2-N)/2;                  % max search window displacement
bxor = uint8(zeros(N));         % intermediate value
Cj = 1e9*ones(2*p+1);           % correlation measure
V = zeros(4,2,nFr);             % motion vectors
Vg = zeros(nFr,2); %[0 0];      % global motion vector
Va = zeros(nFr,2);              % integrated motion vectors
Ms = uint8(zeros(h,w,nFr));     % init stabilized image sequence

% loop over subsequent frames
for fr = 1:nFr
    waitbar((fr-1)/nFr,hw) % display progress
```

```
% get grey coded bit plane
   [Mg] = uint8(getGrayCodeBitPlane(M,bit,fr,debug_disp));

  % break into 4 sub-images, (S1, S2, S3, S4)
  %   SX(:,:,2,1) is S1 current image, SX(:,:,1,1) is past
  %   could probably optimize as a reshape
  S(:,:,2,1) = Mg( 1:h/2,       1:w/2       ); % UL, S1
  S(:,:,2,2) = Mg( 1:h/2,       w/2+1:end   ); % UR, S2
  S(:,:,2,3) = Mg( h/2+1:end,   1:w/2       ); % LL, S3
  S(:,:,2,4) = Mg( h/2+1:end,   w/2+1:end   ); % LR, S4

  if fr > 1 % start algorithm after 1st frame

      for j = 1:4 % loop over sub-images
          if ~logSearchEnable
              % exhaustive search (slower, more precision)
              for m_pos = 1:2*p+1 % loop over possible displacements
                  for n_pos = 1:2*p+1

                      % calculate correlation measure
                      bxor = bitxor( ...  % could be very fast HW
                          S(p+1:p+N,p+1:p+N,2,j) , ...
                          S(m_pos:m_pos+N-1,n_pos:n_pos+N-1,1,j) );
                      Cj(m_pos,n_pos) = sum(bxor(:)); % could mult by 1/N^2

                  end
              end % displacement loops

              % find min Cj arguments: m_pos_min & n_pos_min
              [tmp,m_pos_min] = min(Cj);
              [tmp,n_pos_min] = min(tmp); clear tmp;
              m_pos_min=m_pos_min(n_pos_min);
          else
              % log search (faster)
              %  currently works only for 256x256 images
              firstJmp = 4;
              prev_m_pos = 9; prev_n_pos = 9; % start in center
              for iter = 1:nSteps
                  % iter'th step
                  Cj = 1e9*ones(2*p+1); % reset correlation measure
                  curJmp = firstJmp./2.^(iter-1);
                  for m_pos = prev_m_pos-curJmp:curJmp:prev_m_pos+curJmp
                      for n_pos = prev_n_pos-curJmp:curJmp:prev_n_pos+curJmp

                          % calculate correlation measure
                          bxor = bitxor( ...  % could be very fast HW
                              S(p+1:p+N,p+1:p+N,2,j) , ...
                              S(m_pos:m_pos+N-1,n_pos:n_pos+N-1,1,j) );
                          Cj(m_pos,n_pos) = sum(bxor(:));

                      end
                  end

                  % find min Cj arguments: m_pos_min & n_pos_min
                  [tmp,m_pos_min] = min(Cj);
                  [tmp,n_pos_min] = min(tmp); clear tmp;
                  m_pos_min=m_pos_min(n_pos_min);
```

```
                % store for loop
                prev_m_pos = m_pos_min;
                prev_n_pos = n_pos_min;

            end % iteration loop

        end % log search if

        % store motion vector for this sub-image
        V(j,:,fr) = [m_pos_min n_pos_min]-p-1; % V[1] V[2]
    end % sub-image loop

    % calc current global motion vector
    Vg(fr,:) = median([V(:,:,fr);Vg_prev]);

    % apply damping to generate this frame's integrated motion vector
    Va(fr,:) = D1 * Va_prev + Vg(fr,:);

end % fr>1 if

% store current image as past image
S(:,:,1,:) = S(:,:,2,:);    % gray coded sub-images
Vg_prev = Vg(fr,:);         % global motion vector
Va_prev = Va(fr,:);         % integrated motion vector

% stabilize the image with the current motion vector
%Ms(:,:,fr) = M(:,:,fr); % add Xform !!! ???
switch rotEnable
case 0
    % original translational correction
    %  (not sub-pixel for now)
    r = round(Va(fr,1)); % num rows moved
    c = round(Va(fr,2)); % num columns moved
    Ms(max([1 1+r]):min([h h+r]),max([1 1+c]):min([w w+c]),fr) = ...
        M(max([1 1-r]):min([h h-r]),max([1 1-c]):min([w w-c]),fr);
case 1
    % correct for rotational & translational motion: matrix method
    %  POOR PERFORMANCE
    %B = IMROTATE(A,ANGLE,METHOD,'crop')
    cnst = 12; %128;
    theta = zeros(1,20);
    Mrs = uint8(zeros(size(M))); Mrs(:,:,1) = M(:,:,1);
    for fr=2:20
        B=[V(:,:,fr) + cnst*[-1 1; 1 1; -1 -1; 1 -1]]';B=B(:);
        A=zeros(2*4,4);
        A(1:2:end,1)=V(:,1,fr-1) + cnst*[-1 1 -1 1]';  %1st col
        A(2:2:end,1)=V(:,2,fr-1) + cnst*[1 1 -1 -1]';
        A(2:2:end,2)=V(:,1,fr-1) + cnst*[-1 1 -1 1]';  %2nd col
        A(1:2:end,2)=V(:,2,fr-1) - cnst*[1 1 -1 -1]';
        A(1:2:end,3)=1;          %3rd col
        A(2:2:end,4)=1;          %4th col
        X=A\B
        theta(fr)=atan2(X(1),X(2))*180/pi-90; theta(fr)
        %err=V(1,:,fr)-X(3:4)'
        Mrs(:,:,fr) = ...
            imrotate(M(:,:,fr),sum(theta(1:fr)),'bilinear','crop');
```

```
                figure,imshow(Mrs(:,:,fr))
            end
        case 2
             % correct for rotational: simple/efficient rot. correction

        end

    end % frame loop

    close(hw) % close progress bar

    return
```

C. A listing of a sub-routine that computes the Gray-coded bit plane, "getGrayCodeBitPlane.m" follows.

```
% filename: getGrayCodeBitPlane.m
% author: Alan Brooks
% description: Given gray scale double matrix, M(width,height,fr),
%   this function returns the Gray coded bit plane, bit.
% version history:
%   08-Mar-2003 created
%               optimized of debug_disp is off

function [Mg] = getGrayCodeBitPlane(M,bit,fr,debug_disp)

if debug_disp
    % slow (all bits)
    msb = 8;
    lsb = 1;
else
    % speedy
    msb = min([bit+1 8]);
    lsb = bit;
end

w = length(M(1,:,1)); % width
h = length(M(:,1,1)); % height

%  useful: bitget, dec2bin, num2bin, bitxor
M1bit = zeros(h,w,8);
M1bitGray = zeros(size(M1bit));
for b = msb:-1:lsb
    % compute original bit-planes (1 is LSB, 8 is MSB)
    M1bit(:,:,b) = bitget(M(:,:,fr),b); % fr'th frame
    % compute gray coded bit-planes
    if b==8 % MSB
        M1bitGray(:,:,b) = M1bit(:,:,b);
    else % LSB's
        M1bitGray(:,:,b) = bitxor(M1bit(:,:,b),M1bit(:,:,b+1));
    end
end

% Single bit output value
Mg = M1bitGray(:,:,bit);

% Debugging displays
if debug_disp
    for b = 8:-1:1
        figure,imshow(M(:,:,fr),[0 255]) % orig
        figure,set(gcf,'name',sprintf('Bit-Plane for bit # %d',b))
        imshow(M1bit(:,:,b)) % normal
        figure,set(gcf,'name',sprintf('Gray Bit-Plane for bit # %d',b))
        imshow(M1bitGray(:,:,b)) % gray
    end
end

return
```