Passive Two-Camera System to Generate a Depth Map of a Scene and Depth Estimation using Optical Flow Techniques.

Mark Ovinis
ovinis@yahoo.com

Sooyong Lee
sooyong@mengr.tamu.edu

Technical Report
Department of Mechanical Engineering
Texas A&M University
May 31, 2003

Abstract

This paper presents a method to recover depth information from a 2-D image taken from different viewpoints. An implementation of a passive two-camera system and a region-based matching algorithm implemented in Matlab generates a depth map of a scene based on a pair of stereo images. Matlab scripts implementing optical flow algorithms were written. The optical flow for a motion sequence was computed and the vector field that shows both the direction and magnitude of the corresponding motion displayed.

Introduction

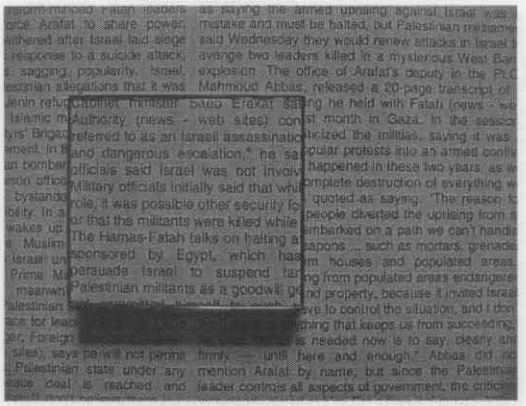


Figure 1: Image of an object against a flat background

A point on an object in a scene is usually described using 3 coordinates. In the Cartesian space, this would be the *x, y* and *z*-coordinates. Unfortunately, when we take a picture of a scene, we lose a dimension of information. For example, in the image of the scene above, we do not have information on the height of the object or its *z*-coordinates. If we were to take 2 images of the same object from different viewpoints, the depth information or the *z*-coordinates of the object is actually hidden in the relative displacements between the left and right view of an image, which is also known as the image disparities.

Figure 2: Stereo image of an object.

This is the basis of stereovision. Stereovision is a vision-based method that enables one to recover 3-D depth information from images taken from different viewpoints.

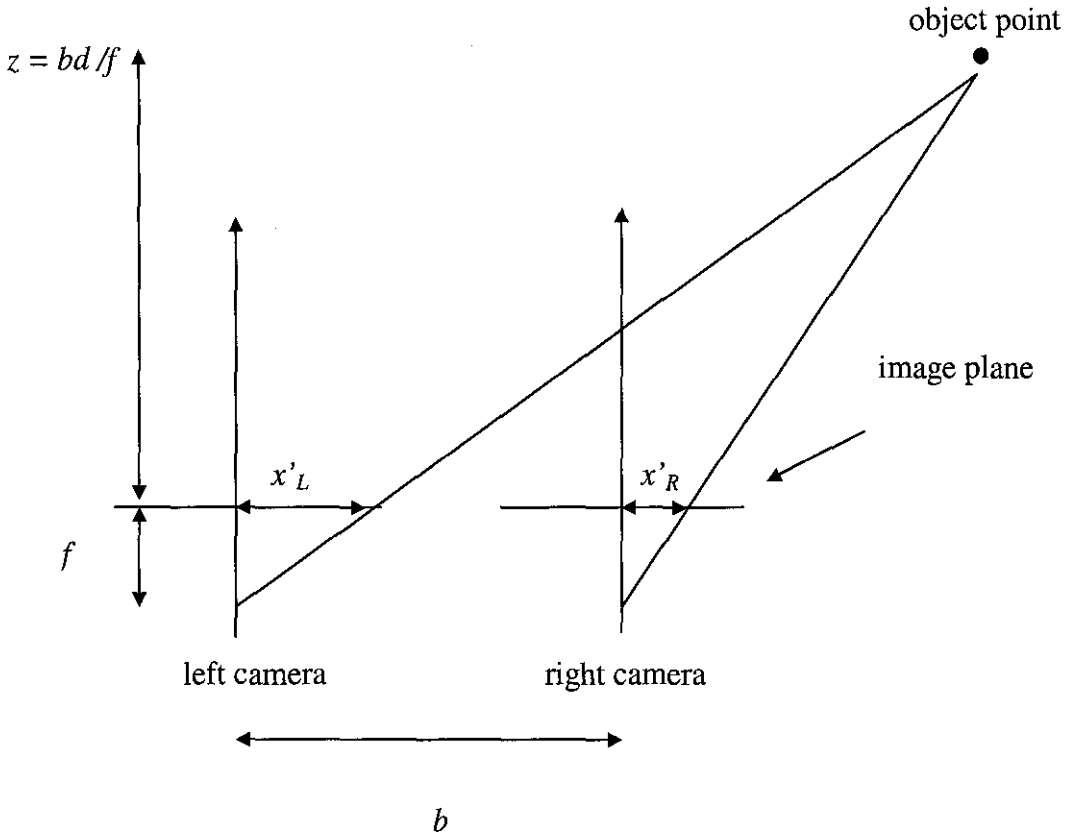


Figure 3: A simple stereo system

The simplest case for stereovision is where two cameras are pointed in the same direction and separated by some *baseline* distance b. For simplicity the cameras are offset only along the x-axis. In this case, the distance z to the point is the same for both cameras. With this simple geometry, we can solve for z. The difference, $d = x'_L - x'_R$ is called the horizontal disparity. Based on the geometry and from triangulation,

$$z = bf / d$$

where f is the focal length of the camera.

Note that if the baseline distance, b and focal length, f is unknown, reconstruction up to a scale factor is possible. The relative depth, z', is then defined as the inverse of the horizontal disparity, d.

$$z' = 1 / d$$

A variation to the parallel case is when two cameras point to a common position. This is known as vergence. If the point of interest is the point of vergence, there is no disparity – the distance can then be obtained based on the camera orientation. All other variations of

parallel cameras are basically equivalent, simply translate or rotate as needed to reduce to the simplest case.
In principle, stereo range estimation is simple. There are 2 basic problems:

1. Camera calibration. The process of aligning the cameras physically, and mathematically determining the geometric relationships for the camera.

2. The "correspondence problem" or stereo matching. Essentially, this means that for a given stereo image pair, determining which points in one image corresponds to points in the other image.

More problems with correspondence

1. Occlusion: Points visible in one image and not the other. This would result in either a false match or no match in a stereo-matching or correspondence algorithm.

2. Featureless surfaces: An example of a featureless surface would be a completely white background. In this case, there exist no discernable features to differentiate one region from another. This would also result in no correspondence.

3. Ambiguous correspondence: A case in point is surfaces that have repeating textures. Without any unique features, the correspondence algorithm is rendered useless.

One solution is to use multiple cameras/images. There is less chance of occlusion (some subset must have seen it). Also, there will be more information for correspondence. However, solving for depth from multiple cameras is over constrained. We need to solve for depth using a least-squared-error or similar approach.

Constraints to assist in stereo matching

This leads to the introduction of constraints to assist in stereo matching. The basic idea in applying constraints in stereo matching is to limit the search space, which reduces both the number of false matches and computation time.

One method is to look for similarities such as intensity, edges, other features, etc. One can also take advantage of coherence - when in doubt, assume depth is changing smoothly. In addition to coherence, the use of geometric limitation such as limits on depth values, initial estimates of depth may assist in constraining the search space.
Stereo matching algorithms

Traditionally, there are 2 stereo matching algorithms commonly used, feature-based and region-based algorithms.

In feature-based matching, the idea is to pick a feature type (e.g. edge), define a matching criteria (e.g. orientation or contrast sign) and then look for matches within the search space or disparity range.

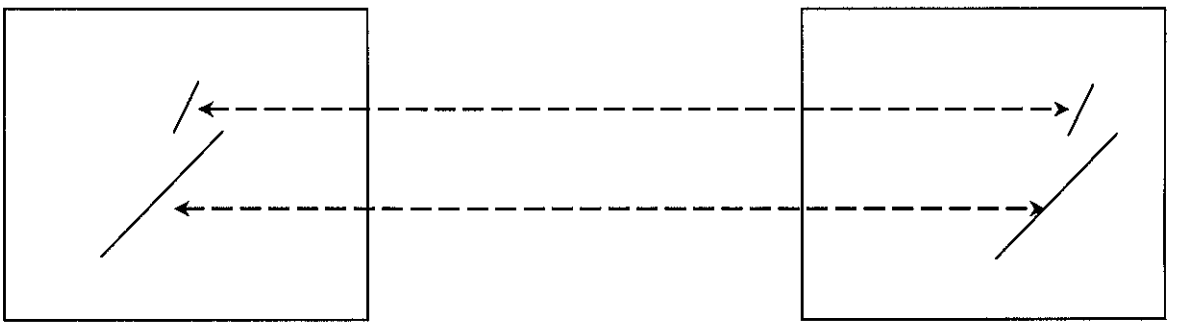
In region-based matching, the idea is to pick a region in the image, represent them in vector form, and attempt to find the matching region in the second image by maximizing some measure such as the normalized cross correlation, normalized sum of squared difference, etc.

Feature-based vs. Region-based

Feature-based matching algorithms are sensitive to feature 'drop-outs' but produce sparse depth or disparity map. As a result, we have to interpolate to fill in the gaps.

An area-based matching algorithm computes a confidence measure for regions but works only where there is texture.

Feature-based



Region-based

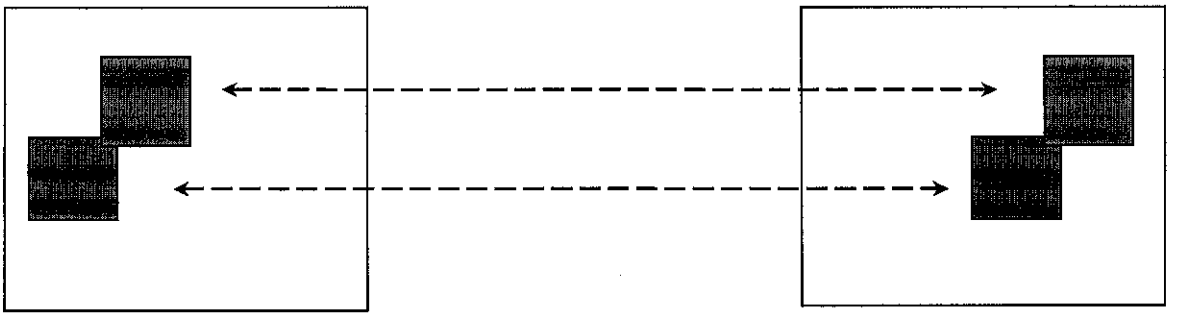


Figure 4: Feature based and region based matching

One good aspect of a region-based matching algorithm is that the technique is well documented and researched. Also, there has been more recent success in this field.

An obvious solution would be to minimize the sum of squares. Let R and R' be a region and candidate region in vector form.

$$\text{Sum of squared difference: SSD} = (R - R')^2$$

Note that we can change SSD by making the image brighter or dimmer, or by changing the contrast. As a result, it is common to subtract the mean of both images & normalize In this case, minimizing SSD is equivalent to maximizing R. This is also known as normalized cross correlation. The normalized cross-correlation algorithm is implemented in Matlab.

Normalized cross correlation:

$$s(r,c) = \frac{\sum_i \sum_j (W_L(i,j) - \mu_L)(W_R(i+r, j+c) - \mu_R)}{\sqrt{\sum_i \sum_j (W_L(i,j) - \mu_L)^2} \sqrt{(W_R(i+r, j+c) - \mu_R)^2}}$$

where W_L and W_R are the reference sub images taken from the left and right image, and μ_L, μ_R are the average intensities of the 2 sub images.

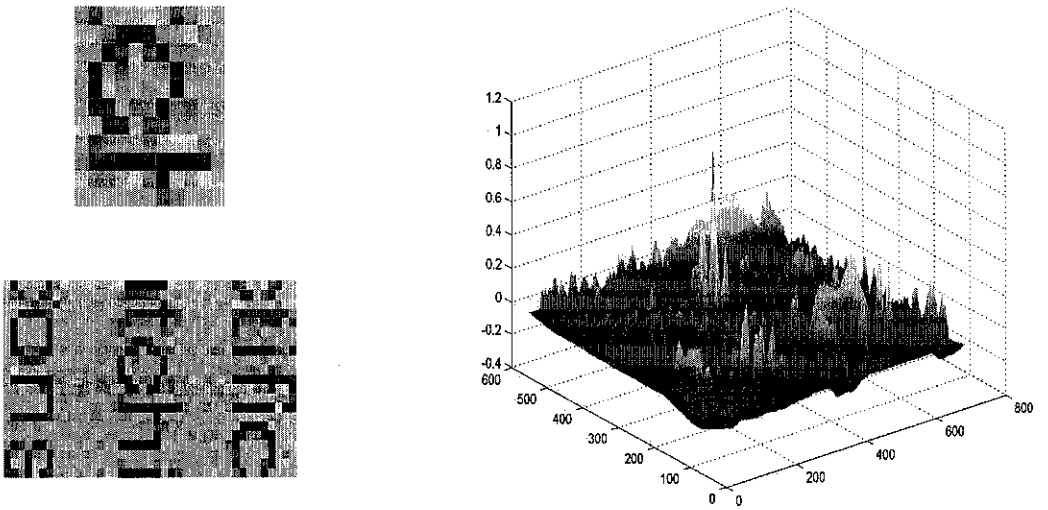


Figure 5: Correspondence search

The image above shows the result of a typical correspondence search process. Here, an attempt to match a region containing the letters 'c' and a partial 'h' to a candidate region containing a subset of the region is carried out. The peak occurs where the sub images are best correlated.

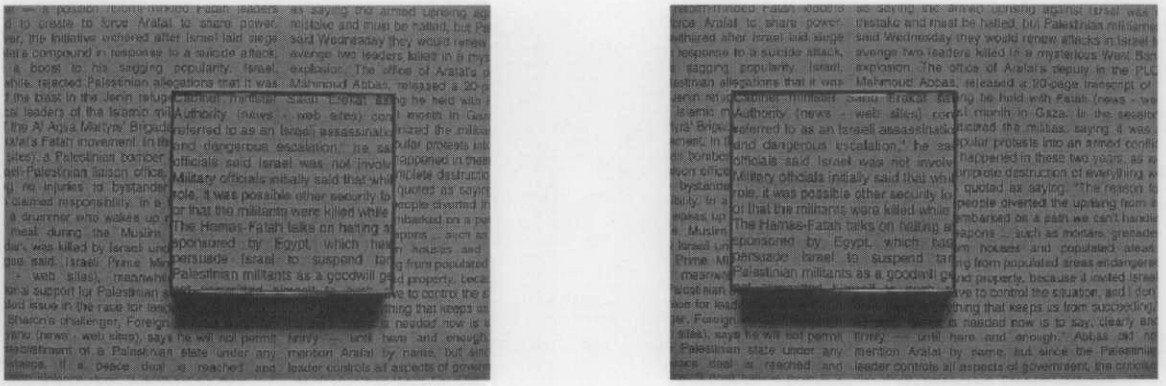


Figure 6: Test images

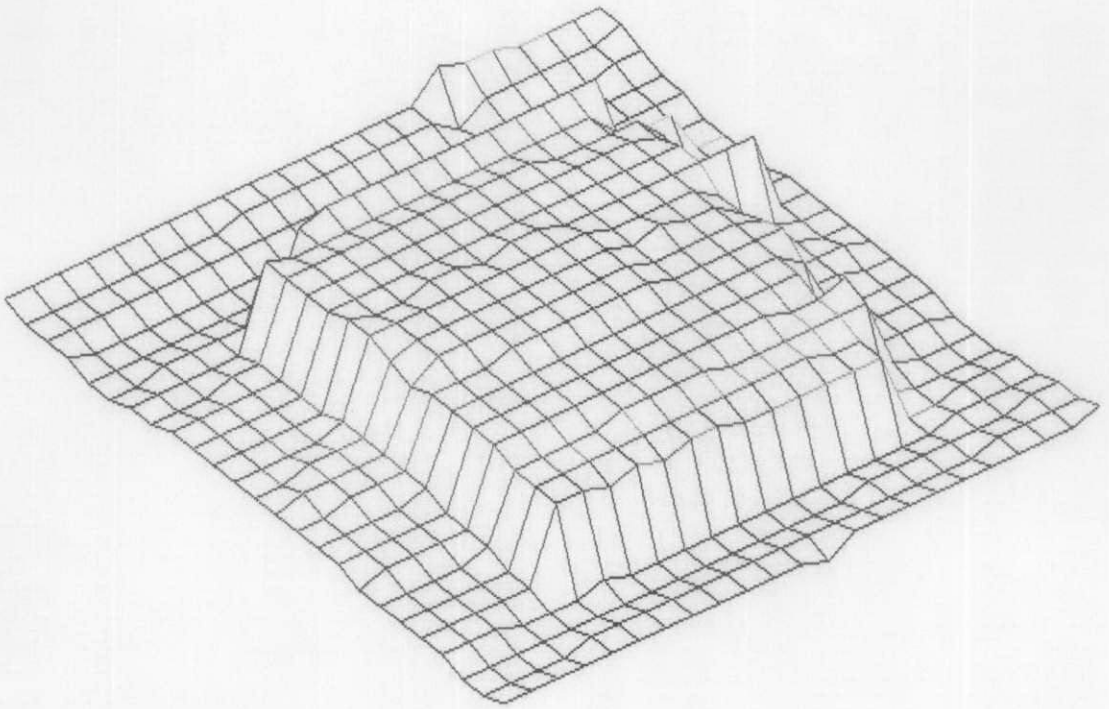


Figure 7: Depth map

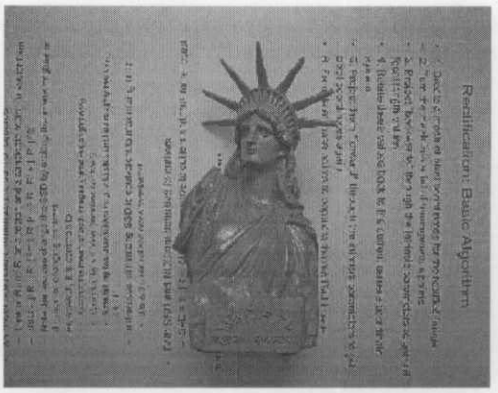


Figure 8: Test images

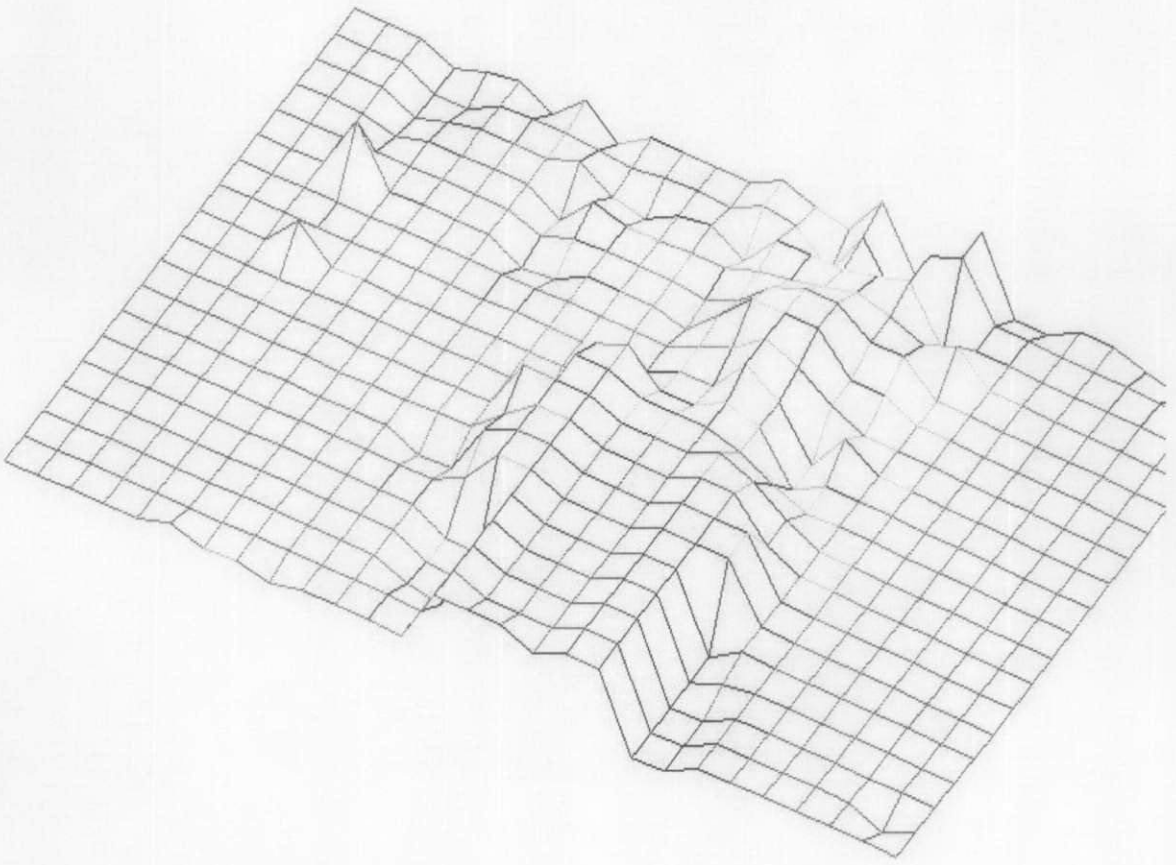


Figure 9: Depth map

It should be possible, even straightforward, to segment moving objects by their motion: by image differencing over successive pair of images. However, things are not so simple. The reason is that regions of constant intensity give no sign of motion, while edges parallel to the direction of motion also give the appearance of not moving: only edges

with a component normal to the direction of motion carry information about the motion. In addition, there is some ambiguity in the direction of the velocity vector. This is partly because there is too little information available to permit the full velocity vector to be computed.

These basic ideas can be expanded upon, and they lead to the notion of optical flow, where a local operator, which is applied to all pixels in the image, will lead to a motion vector field that varies smoothly over the whole image. The attraction lies in the use of a local operator, with its limited computational burden. Ideally, it would have an overhead comparable to an edge detector in a normal intensity image – though it would have to be applied locally to pairs of images in an image sequence.

Translation

An optical flow algorithm, translation.m was tested on two images, 8 x 8 pixels in size. Two images representing a simple motion sequences were used. The first image, $I(x, y, t)$, represents the image intensity at time, t and spatial coordinates, x and y. Displacing the original image in the horizontal and vertical direction pixel-wise produced a synthetic motion. The second image, $I(x+dx, y+dy, t+dt)$, represents the synthetic motion of dx and dy, and dt change in time. In the figure below, the first image was displaced up one pixel and to the right one pixel to produce the second image. Both images are as shown in figure 1.

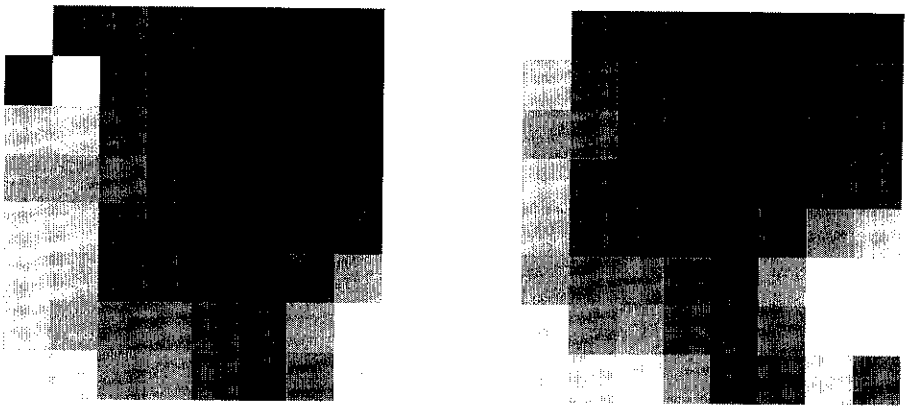


Figure 10: The two images used

Using a standard image processing technique, each image was smoothed by convolving the image with a 2d filter, first in x and then y. A gaussian filter of standard deviation equal to 3 was used and the image contrast was enhanced by histogram equalization.

$$f_x u + f_y v + f_t = 0$$

where f_x, f_y, f_t are spatiotemporal derivatives, and u, v are the optical flow components. Unfortunately, this is a scalar equation, and will not suffice for determining the two local components of the velocity field, as we require. There are two unknowns, u and v, in this equation. Instead of using one equation for one pixel, we considered a small neighborhood around a pixel, and got an over-constrained system, which was solved using the least square fit. Considering a 2 by 2 neighborhood, and assuming optical flow to be constant in this neighborhood, we got 4 optical flow equations:

$$f_{x1} u + f_{y1} v = -f_{t1}$$
$$\cdot$$
$$\cdot$$
$$f_{x4} u + f_{y4} v = -f_{t4}$$

The system, written in matrix form, is shown below

$$\begin{bmatrix} f_{x1} & f_{y1} \\ \cdot & \cdot \\ \cdot & \cdot \\ f_{x4} & f_{y4} \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -f_{t1} \\ \cdot \\ \cdot \\ -f_{t4} \end{bmatrix}$$

or

$$\mathbf{Au} = \mathbf{f_t}$$

$$\mathbf{u} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{f_t}$$

Figure 13: The linear equation, used for computing the optical flow components, u and v.

A linear system is constructed as shown in figure 4, to contain all the gradient information. The 4 x 2 matrix containing gradient information for x and y and 4 x 2 matrix containing the gradient information for t can be solved to yield the optical flow components, u and v

The flow vectors for different motion sequences are shown.

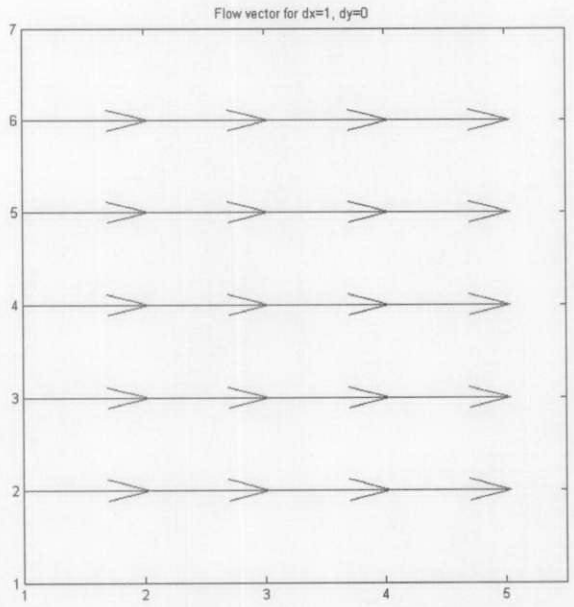


Figure 14: Flow vectors for dx=1, dy=0

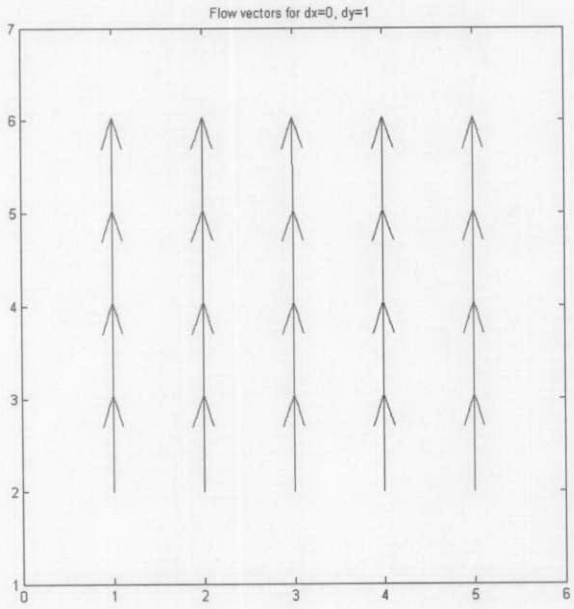


Figure 15: Flow vectors for dx=0, dy=1

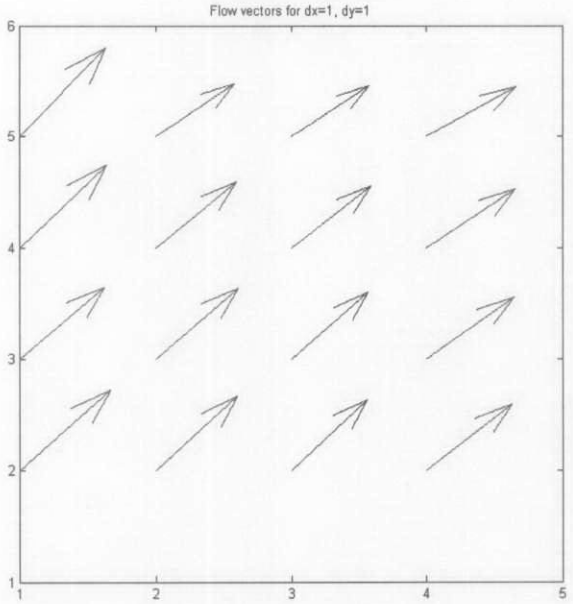


Figure 16: Flow vectors for dx=1, dy=1

The algorithm's accuracy in computing optical flow is investigated. The computed optical flow is compared to the actual displacement of pixels between images. The computed flow for increasing pixel displacements between images is shown.

Ground truth		Optical flow components	
dx	*dy*	*u*	*v*
0	0	0	0
1	0	1.0000	2.3592×10^{-16}
0	1	7.2858×10^{-17}	1.0000
1	1	1.0244	1.0411
2	0	2.3534	-0.0184
0	2	0.0876	2.0710

Figure 17: Comparison between actual displacement and computed displacement based on an optical flow algorithm.

The algorithm accurately computes the optical flow for motion sequences moving at 1 pixel/image. Visually, the flow vectors appear to be accurate. The algorithm is able to detect the corresponding motion, distinguishing between flow in x and y. When pixel displacement is greater than 1 pixel, the algorithm fails to correctly determine the actual displacement. The algorithm has a resolution of 1 pixel for an 8 x 8 pixel size image.

Rotation

The rotation of an image based on a user-specified angle is determined using an optical flow algorithm, rotation.m. However, the algorithm is accurate for images larger than 100 x 100 pixels in size only. In pure rotation, each image points moves at right angles to the line joining it to a fixed point. In rotating an image, each image points moves at a speed proportional to its distance from the fixed point. This can be written very simply in terms of image coordinates. Suppose Cartesian coordinates (x,y) are set up in the image plane with the origin at a fixed point. Then, if the rate of rotation is R, for a point at (x,y), the equation is given by:

$$\text{Rotation, } R = (x \times v) - (y \times u)$$

Where x, y are the coordinates relative to the center of the image and u, v are the pixel velocities in the x and y directions respectively. However, if the sample rate between the 2 images is known, the amount of rotation may be determined. The signs in the equation are consistent with the conventions used in the figure. Positive and negative values of R correspond to clockwise and anticlockwise rotation respectively.

Based on an optical flow algorithm, the velocity and coordinates of a pixel can be determined. As it turns out, rotation is a function of both these variables. In pure rotation, the rotation rates of each pixel in an image are the same. Since the pixels at the edges rotate more than pixels closer to the center, the rotation rate is proportional to the distance between the a pixel and the rotation center or image center as well as the pixel velocity.

The geometrical interpretation is give by figure 8.

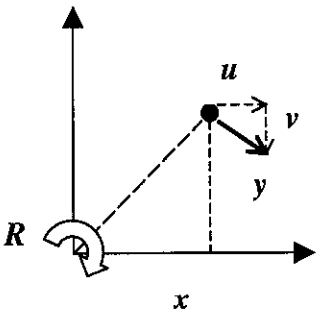


Figure 18: The components of the optical flow vector for an image point at (x,y).

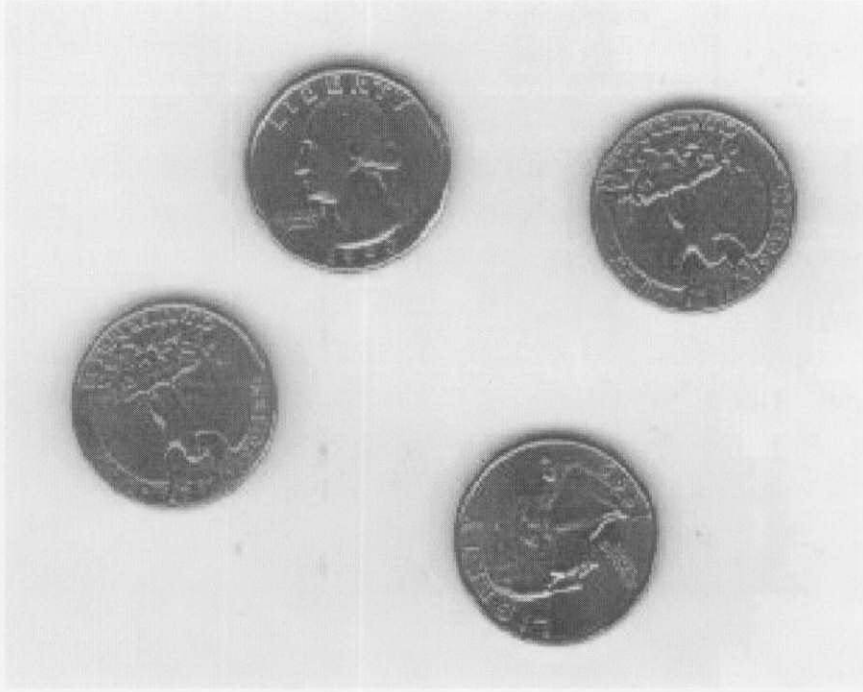


Figure 19: Sample image used to test an optical flow algorithm to compute rotation

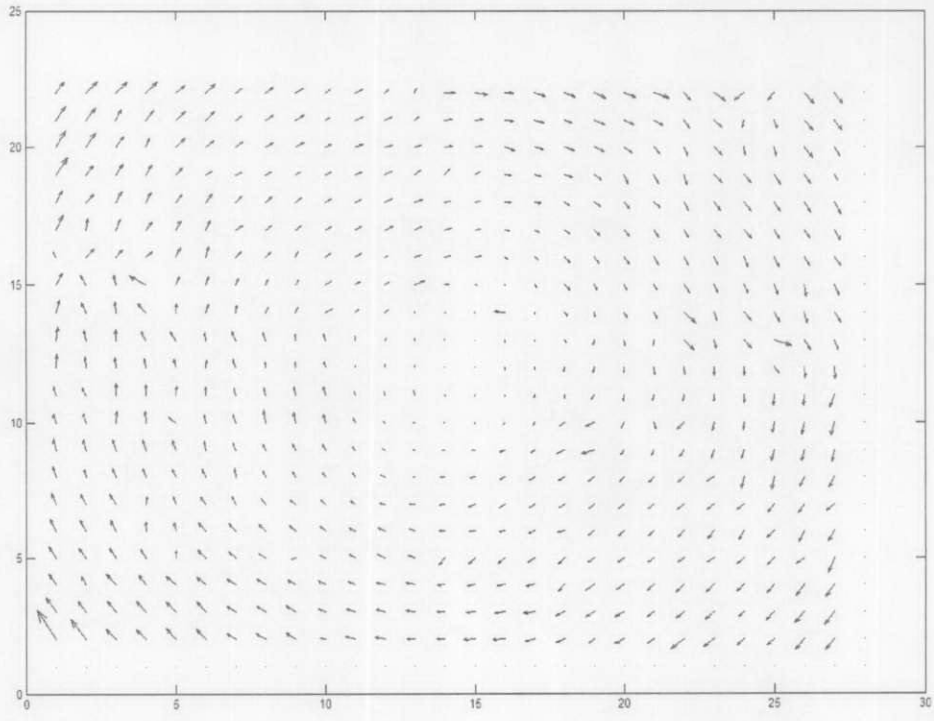


Figure 20: Optical flow vectors of a sampled image rotated 2^{o}

angle	*computed angle*	*difference (%)*
0	0	0.00
1	1.0073	0.73
2	2.0030	0.15
3	3.0422	1.40
4	4.0671	1.68
5	5.0319	0.64
6	5.9330	1.12
7	6.8759	1.77
8	7.9650	0.44
9	9.1080	1.20
10	10.0877	0.88

Figure 21: Comparison between actual rotation and computed rotation based on an optical flow algorithm.

Perturbation

When scenes contain moving objects, the analysis becomes more complex than for scenes where everything is stationary, since temporal variations in intensity have to be taken into account. Optical flow fields must be interpreted in terms of moving objects and camera. The first case is where there is no motion. In this case, the velocity field image contains only vectors of zero length. Next, the camera is moved, or perturbed. In the following scene, the two objects are moved equal distance towards the right.

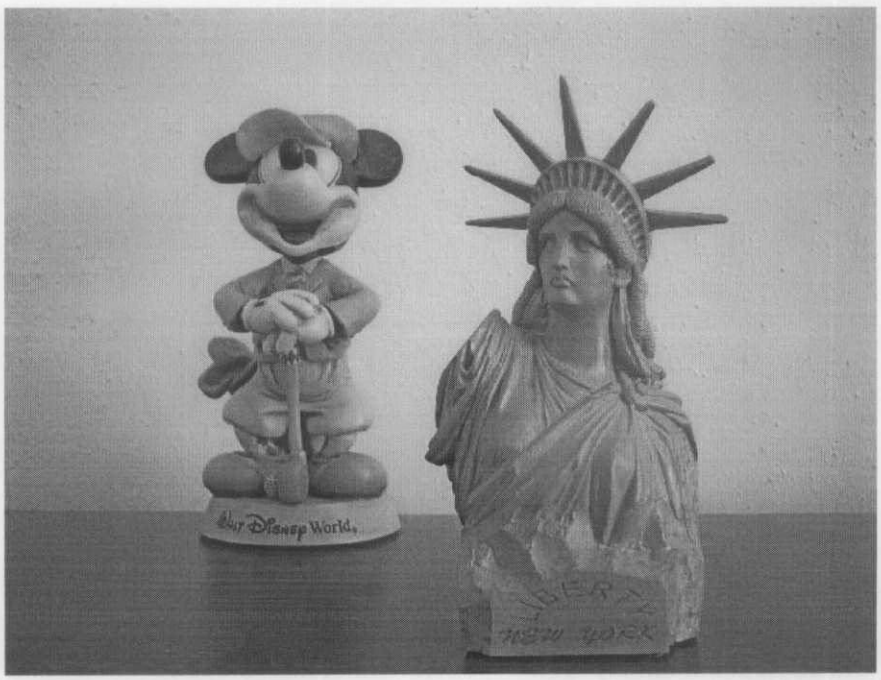


Figure 22: Scene where there is no motion.

Figure 23: The original scene perturbed by moving the camera

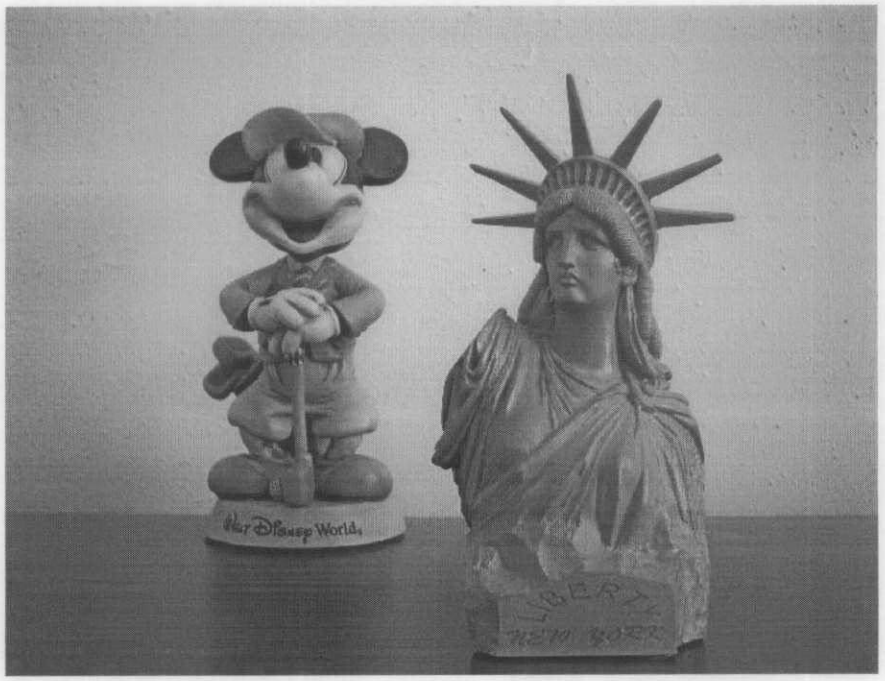


Figure 24: The original scene with objects moved equi-distance

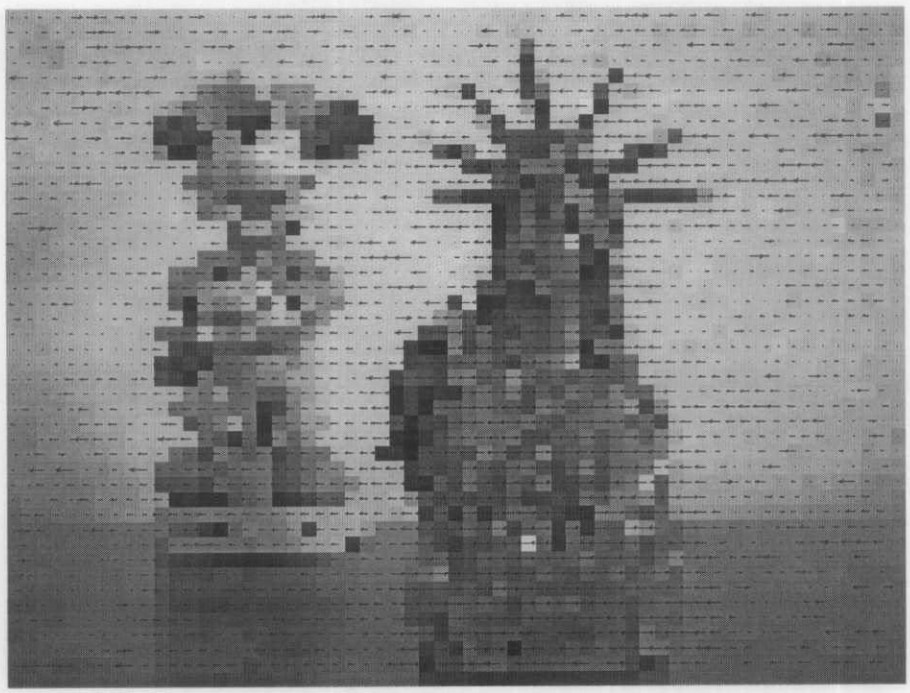


Figure 25: Optical flow vectors of a perturbed scene

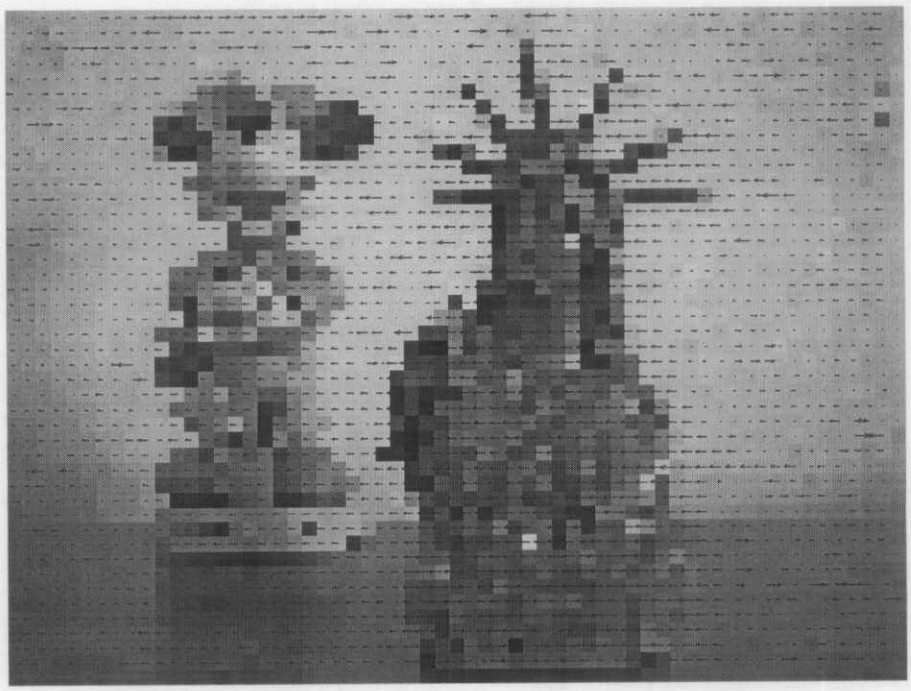


Figure 26: Optical flow vectors of a scene with objects moved equi-distance

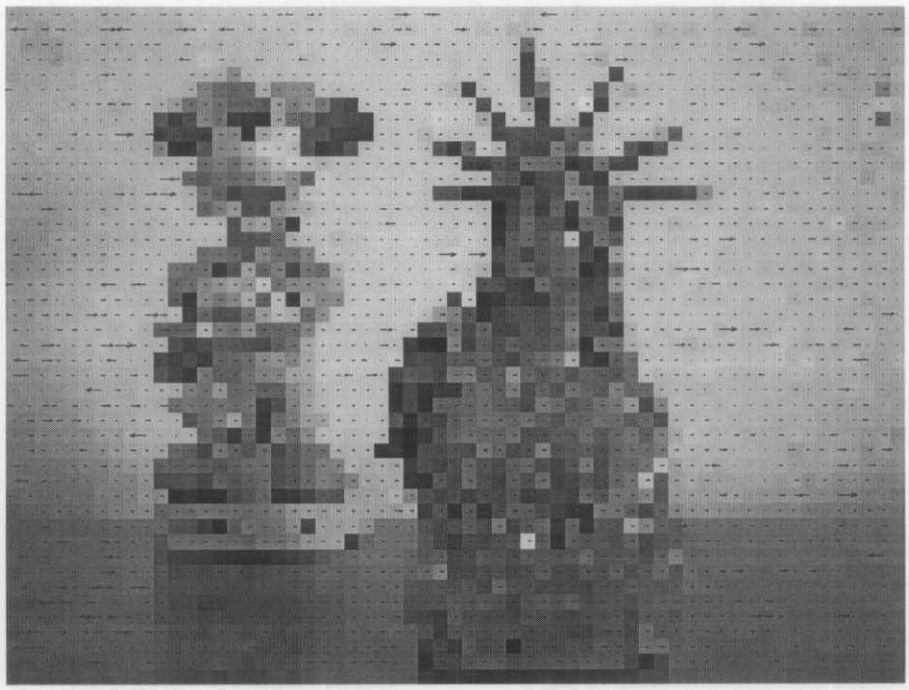


Figure 27: Compensated optical flow vectors

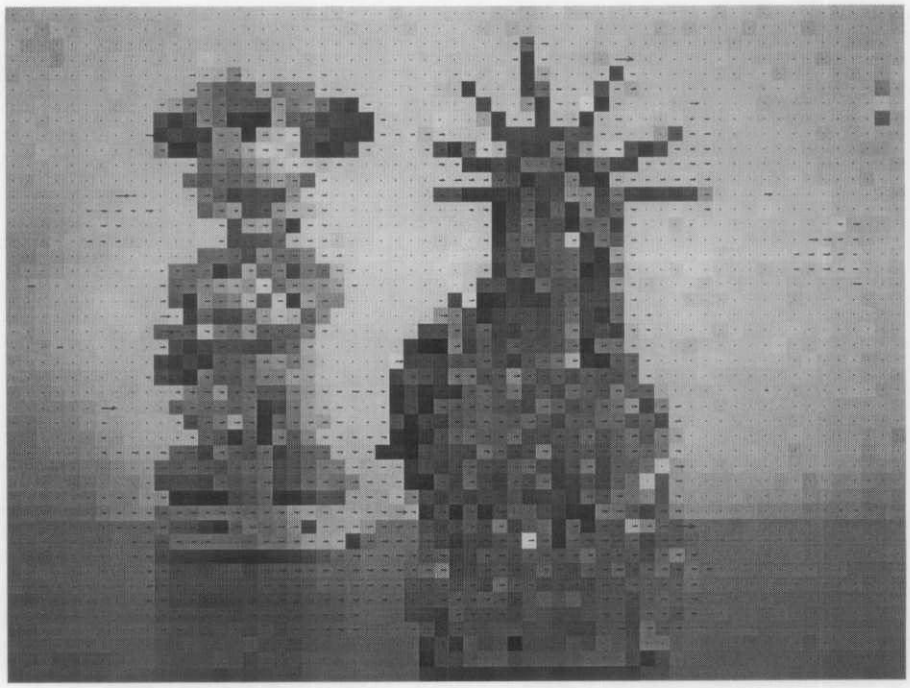


Figure 28: Compensated optical flow vectors without background noise

Pan

Panning an image results in uniform expansion or contraction. In panning an image, each image points moves along a line joining it to a fixed point, with a speed proportional to its distance from the fixed point. This can be written very simply in terms of image coordinates. Suppose Cartesian coordinates (x,y) are set up in the image plane with the origin at a fixed point. Then, if the rate of rotation is R, for a point at (x,y), the equation is given by:

$$P = (x \times u) + (y \times v)$$

Where x, y are the coordinates relative to the center of the image and u, v are the pixel velocities in the x and y directions respectively. However, if the sample rate between the 2 images is known, the amount of panning may be determined.

Based on an optical flow algorithm, the velocity and coordinates of a pixel can be determined. As it turns out, panning is a function of both these variables. In pure panning, the panning rates of each pixel in an image are the same. Since the pixels at the edges rotate more than pixels closer to the center, the panning rate is proportional to the distance between the a pixel and the pan center or image center as well as the pixel velocity. Note that the signs in the equation are consistent with the conventions used in the figure. The pan rate can be positive or negative, corresponding to pan in or pan out respectively.

The geometrical interpretation is given by figure 19.

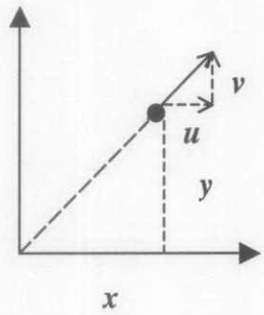


Figure 29: The components of the optical flow vector for an image point at (x,y).

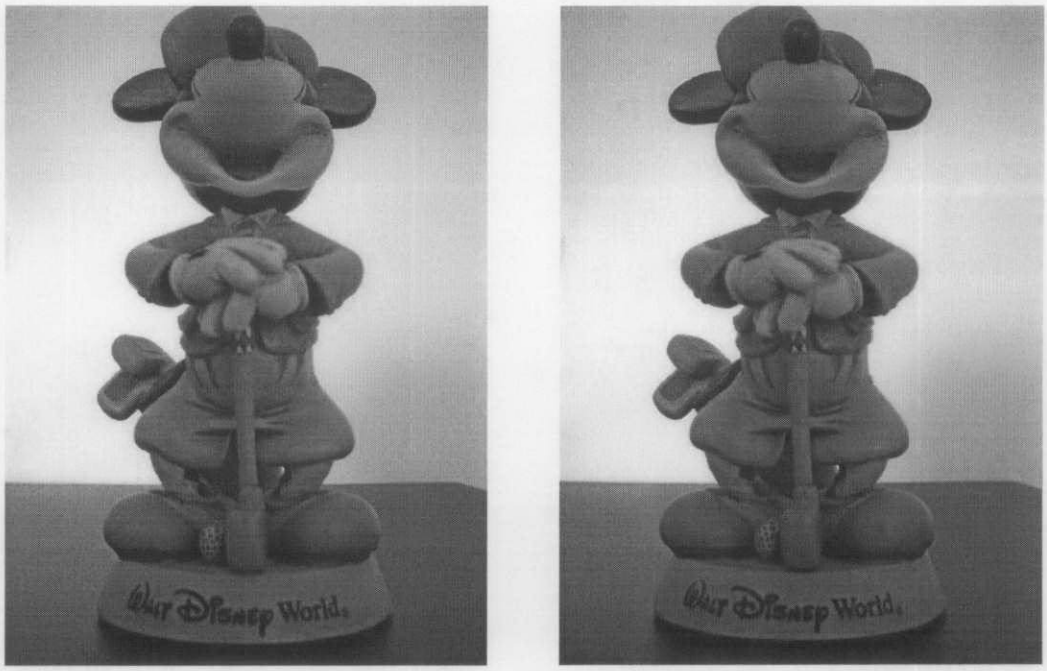


Figure 30: Sample images used to test an optical flow algorithm to compute pan

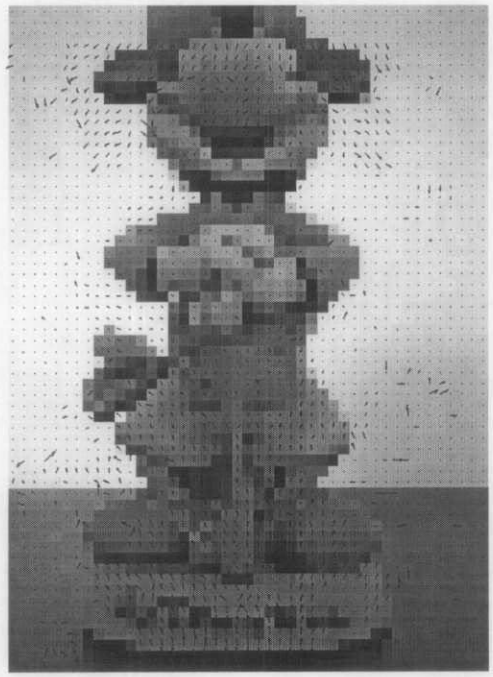


Figure 31: Optical flow vectors

pan distance	*computed pan distance*	*difference (%)*
0.0100	0.0100	0.0
0.0100	0.0096	4.0
0.0100	0.0097	3.0
0.0100	0.0098	2.0
0.0100	0.0099	1.0
0.0100	0.0096	4.0
0.0100	0.0098	2.0
0.0100	0.0094	6.0
0.0200	0.0199	0.5
0.0300	0.0305	1.7
0.0400	0.0409	2.3
0.0500	0.0504	0.8
0.0600	0.0580	3.3

Figure 32: Comparison between actual pan distance and computed pan distance based on an optical flow algorithm for different source images.

Conclusion

An algorithm to generate a depth map of a scene based on a pair of stereo images and to compute the optical flow vectors of a scene based on an image sequence was implemented. The equations used in the algorithm were explained. It is obvious based on the results obtained that there are hidden information that can be extracted from a visual system using cross correspondence and optical flow techniques. However, there are several aspects of the method that could be improved. First of all, a denser depth map could have been generated using smaller regions or finer grids. In order to accomplish this, a more sophisticated algorithm with minimal false matches has to be used. Another improvement would be to implement an algorithm that uses both left to right and right to left correspondence matching to reduce the number of false matches. Here it would be possible to try to generate a depth map of a more complex scene.

In general, for a given a pair of stereo images from 2 cameras with known relative positions, to obtain a depth map of an object:

1.Rectify the two images.
2.Compute image correspondence using either feature-based or area-based matching.
3.Use triangulation to compute distance. If baseline is unknown, simply invert disparity (reconstruction up to a scale factor).
4.Interpolate surface.

For a given a pair of images, to obtain the optical flow of a scene:

1. Smooth the images by standard image processing techniques.

2. Estimate the spatio-temporal gradients in x, y and t for each pixel by convolution with a differencing mask.
3. Set up a over-constrained linear system containing all the gradient information.
4. The equations can then be solved to yield the optical flow components, u and v

However, there are several aspects of the method that could be improved. First of all, more accurate optical flow vectors could have been obtained by using more advance image processing techniques. In order to accomplish this, a more sophisticated algorithm has to be used.

Another improvement would be to implement an algorithm to try to obtain optical flow components of a more complex scene.

References

E. R. Davies *Machine Vision: Theory, Algorithm, Practicalities* (Academic Press, 1997)

John C. Russ (2002) *The Image Processing Handbook* (CRC Press, 2002)

Michael Seul, Lawrence O'Gorman, Michael J. Sammon *Practical Algorithms for Image Analysis, Description, Examples and Codes* (Cambridge University Press, 2000)

Appendix

interactive.m

```
close all;
clear all;

left=imread('left7.jpg');
right=imread('right1.jpg');
l=rgb2gray(left);
r=rgb2gray(right);

% choose subregions of each image interactively
axis ij, title('Choose a region'), hold on;
[sub_l,rect_l] = imcrop(l);
axis ij, title('Choose a candidate region'), hold on;
[sub_r,rect_r] = imcrop(r);

% display sub images
subplot(2,1,1), imshow(sub_l);
subplot(2,1,2), imshow(sub_r);

% do normalized cross-correlation and find coordinates of peak. calculate the normalized cross-correlation
and display it as a surface plot.
% the peak of the cross-correlation matrix occurs where the sub_images are best correlated.
c = normxcorr2(sub_l(:,:,1),sub_r(:,:,1));
figure, surf(c), shading flat;

% find the offset between the images. the total offset or translation between images depends on the location
of the peak in the cross-correlation matrix, and on the size and position of the sub images.
% offset found by correlation
[max_c, imax] = max(abs(c(:)));
[ypeak, xpeak] = ind2sub(size(c),imax(1));
corr_offset = [(xpeak-size(sub_l,2))
        (ypeak-size(sub_l,1))];

% relative offset of sub images
rect_offset = [(rect_r(1)-rect_l(1))
        (rect_r(2)-rect_l(2))];

% total offset
offset = corr_offset + rect_offset;
xoffset = offset(1)
yoffset = offset(2)
```

iteration.m

```
close all;
clear all;

left=imread('left7.jpg');
right=imread('right1.jpg');
l=rgb2gray(left);
```

```
r=rgb2gray(right);

% crop image for correspondence match
l=l(50:440,80:640);
r=r(50:440,1:580);

% choose subregions of each image.
m=1; n=1;
for i=1:20:541
   for j=1:20:361
      rect_l = [i j 19 19];
      rect_r = [i-10 j-5 58 38];
      sub_l = imcrop(l,rect_l);
      sub_r = imcrop(r,rect_r);

% do normalized cross-correlation and find coordinates of peak. calculate
% the normalized cross-correlation and display it as a surface plot. the
% peak of the cross-correlation matrix occurs where the sub_images are best
% correlated.
c = normxcorr2(sub_l(:,:,1),sub_r(:,:,1));
% figure, surf(c), shading flat;

% find the offset between the images. the total offset or translation
% between images depends on the location of the peak in the
% cross-correlation matrix, and on the size and position of the sub images.
% offset found by correlation
[max_c, imax] = max(abs(c(:)));
[ypeak, xpeak] = ind2sub(size(c),imax(1));
corr_offset = [(xpeak-size(sub_l,2))
          (ypeak-size(sub_l,1))];

% relative offset of sub images
rect_offset = [(rect_r(1)-rect_l(1))
          (rect_r(2)-rect_l(2))];

% total offset
offset = corr_offset + rect_offset;

% compensation for range of sub region
if i == 1
xoffset = offset(1) + 10;
else
   xoffset = offset(1);
end
if j == 1
   yoffset = offset(2) + 5;
else
   yoffset = offset(2);
end

warning off MATLAB:divideByZero
Z(m,n) = 1/abs(xoffset);

% assumption about maximum offset, smoothness of offset changes
if Z(m,n) > .2 & n ~= 1
   Z(m,n) = Z(m,n-1);
```

```
elseif Z(m,n) > .2 & n == 1
   Z(m,n) = Z(m-1,18);
else
   Z(m,n) = Z(m,n);
end

n = n + 1;
if n == 20;
   n = 1;
   m = m + 1;
end
end
end

% 3D surface plot
axis([0 25 0 25 0 1]), axis off, hold on, mesh(Z)

close all;
clear all;

left = imread ('stereo1.jpg');
right = imread ('stereo2.jpg');
l = rgb2gray (left);
r = rgb2gray (right);

% crop image to align for correspondence match

l = l (5:480,132:640);
r = r (1:480,1:520);

% choose sub regions of each image by iteration.

m = 1; n = 1;
for i = 1:20:481
   for j = 1:20:441
      rect_l = [i j 19 19];
      rect_r = [i-10 j-1 41 21];
      sub_l = imcrop (l, rect_l);
      sub_r = imcrop(r, rect_r);

% or interactively
% [sub_l, rect_l] = imcrop (l);
% [sub_r, rect_r] = imcrop(r);

% display sub images
% subplot (2,1,1), imshow (sub_l);
% subplot (2,1,2), imshow (sub_r);

% do normalized cross-correlation and find coordinates of peak. calculate the normalized % cross-
correlation and display it as a surface plot. the peak of the cross-correlation
% matrix occurs where the sub images are best correlated.

c = normxcorr2 (sub_l(:,:,1),sub_r(:,:,1));

% correlation plot;
figure, surf(c), shading flat;
```

```
% find the offset between the images. the total offset or translation between images
% depends on the location of the peak in the cross-correlation matrix, and on the size and
% position of the sub images. offset found by correlation

[max_c, imax] = max(abs(c(:)));
[ypeak, xpeak] = ind2sub(size(c),imax(1));
corr_offset = [(xpeak-size(sub_l,2))
        (ypeak-size(sub_l,1))];

% relative offset of sub images

rect_offset = [(rect_r(1)-rect_l(1))
        (rect_r(2)-rect_l(2))];

% total offset

offset = corr_offset + rect_offset;

% compensation for range of sub region

if i == 1
xoffset = offset (1) + 10;
else
   xoffset = offset (1);
end
if j == 1
   yoffset = offset (2) + 1;
else
   yoffset = offset (2);
end

Z (m, n) = abs (xoffset);

% assumption about maximum offset, offset smoothness to eliminate false matches and disambiguate
multiple matches

if Z (m, n) >= 12 & n ~= 1
   Z (m, n) = Z (m, n-1);
elseif Z (m, n) >= 12 & n == 1
   Z (m, n) = Z (m, 22);
else
   Z (m, n) = Z (m, n);
end

n = n + 1;
if n == 24;
   n = 1;
   m = m + 1;
end
end
end

% 3D surface plot
axis ([0 25 0 25 0 100]), axis off, hold on, mesh (Z)
```

translation.m

```
close all;
clear all;

image=imread('Quarters.jpg');
image=rgb2gray(image);

% select window
figure, axis ij,title('Specify window'),hold on;
image=imcrop(image);
image=histeq(image);
close all;
[rows,cols]=size(image);

% synthetic motion with dx and dy
dx=input('dx=');
dy=input('dy=');
image1=image(1:rows-dy,1:cols-dx);
image2=image(dy+1:rows,dx+1:cols);

image1=double(image1);
image2=double(image2);

sigma=5;  % degree of smoothing
hsize=ceil(6*sigma);
gmsk=fspecial('gaussian',hsize,sigma);  % gaussian mask
smthimg=conv2(.5*(image1+image2),gmsk); % smoothed image

hor=[-1,1;-1,1];   % horizontal convolution mask
ver=[-1,-1;1,1];   % vertical convolution mask

% spatial gradients
gx=conv2(smthimg,hor);  % horizontal gradient
gy=conv2(smthimg,ver);  % vertical gradient

% temporal gradients
smthimg1=conv2(image1,gmsk);
smthimg2=conv2(image2,gmsk);
gt=.5*(conv2(smthimg2,[1,1;1,1])+conv2(smthimg1,[-1,-1;-1,-1]));   % smoothed image difference
jump=10;     % sampling
edge=hsize;  % remove convolution edge
[rows,cols]=size(gx);
gxs=gx(1+edge:jump:rows-edge,1+edge:jump:cols-edge);
gys=gy(1+edge:jump:rows-edge,1+edge:jump:cols-edge);
gts=gt(1+edge:jump:rows-edge,1+edge:jump:cols-edge);

di=[gxs(:),gys(:)];
dt=gts(:);

% calculate u, v
[m,n]=size(gxs);
u=zeros(m,n);
v=zeros(m,n);

for j=1:n-1
```

```
   for i=1:m-1
      a=gxs(i:i+1,j:j+1);
      b=gys(i:i+1,j:j+1);
      c=gts(i:i+1,j:j+1);
      d=[a(:),b(:)];
      uv=-pinv(d)*c(:);
      u(i,j)=uv(1);
      v(i,j)=uv(2);

   end
end

[I,J]=size(u);
u=u(1:I,1:J);
v=v(1:I,1:J);
quiver(1:J,I:-1:1,u,v);
axis square

% calculate vx,vy
var=-pinv(di)*dt;
vx=var(1,1)
vy=var(2,1)
```

rotation.m

```
close all;
clear all;

% load image
image1=imread('Quarters.jpg');
image1=rgb2gray(image1);
image1=histeq(image1);
% select window
% figure
% axis ij,title('Specify window'),hold on;
% image1=imcrop(image1);
% close all;
[rows,cols]=size(image1);

% synthetic rotation
angle=input('angle=');
image2=imrotate(image1,angle,'crop');

image1=double(image1);
image2=double(image2);
sigma=5;   % degree of smoothing
hsize=ceil(6*sigma);
gmsk=fspecial('gaussian',hsize,sigma);  % gaussian mask
smthimg=conv2(.5*(image1+image2),gmsk); % smoothed image

hor=[-1,1;-1,1];   % horizontal convolution mask
ver=[-1,-1;1,1];   % vertical convolution mask

% spatial gradients
gx=conv2(smthimg,hor);   % horizontal gradient
```

```
gy=conv2(smthimg,ver);  % vertical gradient

% temporal gradient
smthimg1=conv2(image1,gmsk);
smthimg2=conv2(image2,gmsk);
gt=.5*(conv2(smthimg2,[1,1;1,1])+conv2(smthimg1,[-1,-1;-1,-1]));   % smoothed image difference

jump=10;     % sampling
edge=hsize;  % remove convolution edge
[rows,cols]=size(gx);
gxs=gx(1+edge:jump:rows-edge,1+edge:jump:cols-edge);
gys=gy(1+edge:jump:rows-edge,1+edge:jump:cols-edge);
gts=gt(1+edge:jump:rows-edge,1+edge:jump:cols-edge);

dt=gts(:);
[m,n]=size(dt);  % number of image points, n

di=zeros(n,1);  % spatial gradients matrix
[nrows,ncols]=size(gxs);
rctr=(nrows+1)/2;  % image centre coordinates
cctr=(ncols+1)/2;

i=1;   % index

for col=1:ncols
   x=(col-cctr)*jump;  % coordinate relative to center
   for row=1:nrows
      y=(row-rctr)*jump;
      fx=gxs(row,col);   % gradients function
      fy=gys(row,col);
      xfy=x*fy;
      yfx=y*fx;
      di(i,:)=[xfy-yfx];
      i=i+1;
   end;
end;

w=-pinv(di)*dt;
angle=rad2deg(w)

% calculate u, v
[m,n]=size(gxs);
u=zeros(m,n);
v=zeros(m,n);

for j=1:n-1
   for i=1:m-1
      a=gxs(i:i+1,j:j+1);
      b=gys(i:i+1,j:j+1);
      c=gts(i:i+1,j:j+1);
      d=[a(:),b(:)];
      uv=-pinv(d)*c(:);
      u(i,j)=uv(1);
      v(i,j)=uv(2);
   end
end
```

```
jump=1;
[I,J]=size(u);
u=u(1:jump:I,1:jump:J);
v=v(1:jump:I,1:jump:J);
quiver(1:jump:J,I:-jump:1,u,-v);
```

opticflow.m

```
close all;
clear all;
img1=imread('original1.jpg');
img2=imread('perturbed1.jpg');
img3=imread('offset1.jpg');

offset1=offset(img1,img2);
offset2=offset(img1,img3);

[u,v,image]=velocity(img1,img2,5,10,offset1);
[s,t]=velocity(img1,img3,5,10,offset2);

[m,n]=size(s);

warning off MATLAB:divideByZero

vx=s./u;  % distance compensation

for j=1:n-1
   for i=1:m-1
      if abs(vx(i,j))<10  % assumption about maximum allowable offset
        vx(i,j)=vx(i,j);
      else
        vx(i,j)=0;
      end
   end
end

ux=neighbor(vx,.5,1.5);

figure
axis ij,title('Optical flow vectors of a perturbed scene'),hold on;
imshow(image,'notruesize')
hold on
quiver(1:n,1:m,u,v);
figure
axis ij,title('Optical flow vectors of a scene with objects offset equi-distance'),hold on;
imshow(image,'notruesize')
hold on
quiver(1:n,1:m,s,t);

figure
axis ij,title('Compensated optical flow vectors'),hold on;
imshow(image,'notruesize')
hold on
quiver(1:n,1:m,vx,v);
```

```
figure
axis ij,title('Compensated optical flow vectors without background noise'),hold on;
imshow(image,'notruesize')
hold on
quiver(1:n,1:m,ux,v);

figure
axis ij,title('Choose a region for velocity estimation'),hold on;
[sub_l,rect_l]=imcrop(image);
close

figure
axis ij,title('Choose another region for velocity estimation'),hold on;
[sub_r,rect_r]=imcrop(image);
close

a=abs(imcrop(ux,rect_l));
optic_velocity1=abs(numel(a)/sum(a(:)))
b=abs(imcrop(ux,rect_r));
optic_velocity2=abs(numel(b)/sum(b(:)))
```

offset.m

```
function xoffsetmax=offset(image1,image2);

% OFFSET Estimates the maximum horizontal offset of a sampled image
% sequence.
%   XOFFSETMAX=OFFSET(IMAGE1,IMAGE2) estimates the maximum offset of a
%   sampled image sequence based on a normalized cross correlation method.

k=1;

[x y]=size(image1(:,:,1));

for j=50:50:y-50   % sampling to determine maximum offset
  for i=50:50:x-50
    rect_l=[j i 50 50];
    rect_r=[j-10 i-10 70 70];
    sub_l=imcrop(image1,rect_l);   % sampled region
    sub_r=imcrop(image2,rect_r);   % candidate region
    c=normxcorr2(sub_l(:,:,1),sub_r(:,:,1));   % normalized cross correlation
    [max_c,imax]=max(abs(c(:)));
    [ypeak,xpeak]=ind2sub(size(c),imax(1));
    corr_offset=[(xpeak-size(sub_l,2));(ypeak-size(sub_l,1))];
    rect_offset=[(rect_r(1)-rect_l(1));(rect_r(2)-rect_l(2))]; % relative offset of subimages
    xoffset(k)=abs(corr_offset(1)+rect_offset(1)); % total offset
    k=k+1;
    xoffsetmax=max(abs(xoffset(:)));
    if xoffsetmax<10
      xoffsetmax=xoffsetmax;
    else
      xoffsetmax=10;
    end
  end
end
```

velocity.m

```
function [u,v,image]=velocity(image1,image2,sigma,jump,xoffsetmax);

% VELOCITY Finds the velocity components of an image sequence.
%  [U,V,IMAGE]=VELOCITY(IMAGE1,IMAGE2,SIGMA,JUMP,XOFFSETMAX) computes the
%  horizontal velocity component U of an image sequence 1MAGE1, IMAGE2
%  based on a optical flow method. The images are convolved with a
%  Gaussian mask of parameter sigma. Jump is the sampling size and
%  xoffsetmax is the estimated maximum horizontal offset based on a
%  normalized cross correlation method.  Sampled image may also be
%  displayed.

img1=double(histeq(rgb2gray(image1)));
img2=double(histeq(rgb2gray(image2)));

hsize=ceil(6*sigma);   % convolution mask size
gmsk=fspecial('gaussian',hsize,sigma); % gaussian mask
smthimg=conv2(.5*(img1+img2),gmsk); % smoothed image

hor=[-1,1;-1,1];   % horizontal convolution mask
ver=[-1,-1;1,1];   % vertical convolution mask

% spatial gradients
gx=conv2(smthimg,hor);  % horizontal gradient
gy=conv2(smthimg,ver);  % vertical gradient

% temporal gradients
smthimg1=conv2(img1,gmsk);
smthimg2=conv2(img2,gmsk);
gt=.5*(conv2(smthimg2,[1,1;1,1])+conv2(smthimg1,[-1,-1;-1,-1]));   % smoothed image difference

edge=hsize;  % remove convolution edge
[rows,cols]=size(gx);
gxs=gx(1+edge:jump:rows-edge,1+edge:jump:cols-edge);
gys=gy(1+edge:jump:rows-edge,1+edge:jump:cols-edge);
gts=gt(1+edge:jump:rows-edge,1+edge:jump:cols-edge);

% calculate u, v
[m,n]=size(gxs);
u=zeros(m,n);
v=zeros(m,n);

for j=1:n-1
   for i=1:m-1
      fx=gxs(i:i+1,j:j+1);
      fy=gys(i:i+1,j:j+1);
      ft=gts(i:i+1,j:j+1);
      xy=[fx(:),fy(:)];
      uv=-pinv(xy)*ft(:);
      u(i,j)=uv(1);  % assumption about horizontal offset only
   end
end

for j=1:n-1
   for i=1:m-1
```

```
      if abs(u(i,j))<xoffsetmax+1 % assumption about maximum offset
        u(i,j)=u(i,j);
      else
        u(i,j)=0;
      end
    end
end
image=image1(1+edge:jump:rows-edge,1+edge:jump:cols-edge);
```

neighbor.m

```
function new=neighbor(old,lower,upper);

% NEIGHBOR Sets the velocity of a pixel to approximate its neighbors.
%   NEW=NEIGHBOR(OLD,LOWER,UPPER) sets the velocity of a pixel within a
%   lower and upper bound limit of its neighbors velocity. The assumption
%   is that the velocity is uniform in a small region.

[m,n]=size(old);

for j=1:n-1
   for i=1:m-1
      if sign(old(i,j))==sign(old(i,j+1))
        old(i,j)=old(i,j);
      else
        old(i,j)=0;
      end
   end
end

for j=1:n-1
   for i=1:m-1
      if sign(old(i,j))==sign(old(i+1,j))
        old(i,j)=old(i,j);
      else
        old(i,j)=0;
      end
   end
end
for j=1:n-1
   for i=1:m-1
      if lower*old(i,j+1)<old(i,j)<upper*old(i,j+1)
        old(i,j)=old(i,j);
      else
        old(i,j)=0;
      end
   end
end

for j=1:n-1
   for i=1:m-1
      if lower*old(i+1,j)<old(i,j)<upper*old(i+1,j)
        old(i,j)=old(i,j);
      else
        old(i,j)=0;
```

```
    end
  end
end

new=old;
```

xytranslation.m

```
clear all
close all

anim

xy1=xytrans(image1,image2,image3);
xy2=xytrans(image3,image4,image5);
xy3=xytrans(image5,image6,image7);
xy4=xytrans(image7,image8,image9);
xy5=xytrans(image9,image10,image11);
xy6=xytrans(image11,image12,image13);
xy7=xytrans(image13,image14,image15);
xy8=xytrans(image15,image16,image17);
xy9=xytrans(image17,image18,image19);

disp(' dx12    dx13    dy23    dy13');
disp(xy1)
disp(xy2)
disp(xy3)
disp(xy4)
disp(xy5)
disp(xy6)
disp(xy7)
disp(xy8)
disp(xy9)
```

anim.m

```
image1=imread('Picture 036.jpg');
image2=imread('Picture 037.jpg');
image3=imread('Picture 038.jpg');
image4=imread('Picture 039.jpg');
image5=imread('Picture 040.jpg');
image6=imread('Picture 041.jpg');
image7=imread('Picture 042.jpg');
image8=imread('Picture 043.jpg');
image9=imread('Picture 044.jpg');
image10=imread('Picture 045.jpg');
image11=imread('Picture 046.jpg');
image12=imread('Picture 047.jpg');
image13=imread('Picture 048.jpg');
image14=imread('Picture 049.jpg');
image15=imread('Picture 050.jpg');
image16=imread('Picture 051.jpg');
image17=imread('Picture 052.jpg');
image18=imread('Picture 053.jpg');
```

```
image19=imread('Picture 054.jpg');

axis off

for j=1:19
   M(j)=getframe;
end

M(1)=im2frame(image1);
M(2)=im2frame(image2);
M(3)=im2frame(image3);
M(4)=im2frame(image4);
M(5)=im2frame(image5);
M(6)=im2frame(image6);
M(7)=im2frame(image7);
M(8)=im2frame(image8);
M(9)=im2frame(image9);
M(10)=im2frame(image10);
M(11)=im2frame(image11);
M(12)=im2frame(image12);
M(13)=im2frame(image13);
M(14)=im2frame(image14);
M(15)=im2frame(image15);
M(16)=im2frame(image16);
M(17)=im2frame(image17);
M(18)=im2frame(image18);
M(19)=im2frame(image19);

movie(M)
```

xytrans.m

```
close all;
clear all;

image1=imread('Pan01.jpg');
image2=imread('Pan02.jpg');
img1=double(histeq(rgb2gray(image1)));
img2=double(histeq(rgb2gray(image2)));
sigma=5;  % degree of smoothing
hsize=ceil(6*sigma);
gmsk=fspecial('gaussian',hsize,sigma);  % gaussian mask
smthimg=conv2(.5*(img1+img2),gmsk); % smoothed image

hor=[-1,1;-1,1];   % horizontal convolution mask
ver=[-1,-1;1,1];   % vertical convolution mask

% spatial gradients
gx=conv2(smthimg,hor);  % horizontal gradient
gy=conv2(smthimg,ver);  % vertical gradient

% temporal gradients
smthimg1=conv2(img1,gmsk);
smthimg2=conv2(img2,gmsk);
gt=.5*(conv2(smthimg2,[1,1;1,1])+conv2(smthimg1,[-1,-1;-1,-1]));   % smoothed image difference
```

```
jump=10;     % sampling
edge=hsize;  % remove convolution edge
[rows,cols]=size(gx);
gxs=gx(1+edge:jump:rows-edge,1+edge:jump:cols-edge);
gys=gy(1+edge:jump:rows-edge,1+edge:jump:cols-edge);
gts=gt(1+edge:jump:rows-edge,1+edge:jump:cols-edge);

dt=gts(:);
[m,n]=size(dt);  % number of image points, n
di=zeros(n,1);  % spatial gradients matrix
[nrows,ncols]=size(gxs);
rctr=(nrows+1)/2;  % image centre coordinates
cctr=(ncols+1)/2;
k=1;   % index

for col=1:ncols
   x=(col-cctr)*jump;  % coordinate relative to center
   for row=1:nrows
      y=(row-rctr)*jump;
      fx=gxs(row,col);   % gradients function
      fy=gys(row,col);
      xfx=x*fx;
      yfy=y*fy;
      di(k,:)=[xfx+yfy];
      k=k+1;
   end;
end;

pan_distance=-pinv(di)*dt

% calculate u, v
[m,n]=size(gxs);
u=zeros(m,n);
v=zeros(m,n);

for j=1:n-1
   for i=1:m-1
      a=gxs(i:i+1,j:j+1);
      b=gys(i:i+1,j:j+1);
      c=gts(i:i+1,j:j+1);
      d=[a(:),b(:)];
      uv=-pinv(d)*c(:);
      u(i,j)=uv(1);
      v(i,j)=uv(2);
   end
end

[u,v]=region(5,.8,1.2,u,v);

[I,J]=size(u);
u=u(1:I,1:J);
v=v(1:I,1:J);
figure,axis ij,hold on;
image=image1(1+edge:jump:rows-edge,1+edge:jump:cols-edge);
imshow(image,'notruesize')
hold on, quiver(1:J,I:-1:1,u,v);
```