Image Processing Pattern Recognition and Computer Vision

- **Image Processing** is the general term for a wide range of techniques that exist for manipulating and modifying images in various ways, i.e. for digital images are acquired, stored, represented, manipulated, enhanced, etc. Some examples:

  1. Bartline Cable Transmission of images in the 1920's
  2. Contrast Enhancement in X-Ray Images
  3. Removal of Motion Blur (i.e. deblurring license plates of speeding cars)

4. Special effects such as Image Warping
5. Correcting Image Distortion (for example in the Mariner and Apollo space programs)
6. Aerial and Satellite Imagery
7. Particle detection and trajectory measurement in High Energy Physics
8. Lossy and Lossless Image Compression

- **Pattern Recognition** refers to the process of converting a, potentially complex, image into a set of names (symbols) and descriptors (feature vectors) that describe the image. Some examples:

  1. Character Recognition (OCR - Optical Character Recognition) of printed and written text.

2. Shape/Object Recognition in images (where is the apple or where is John?)
3. Face/Fingerprint/Eye/Voice Recognition
4. Texture Analysis

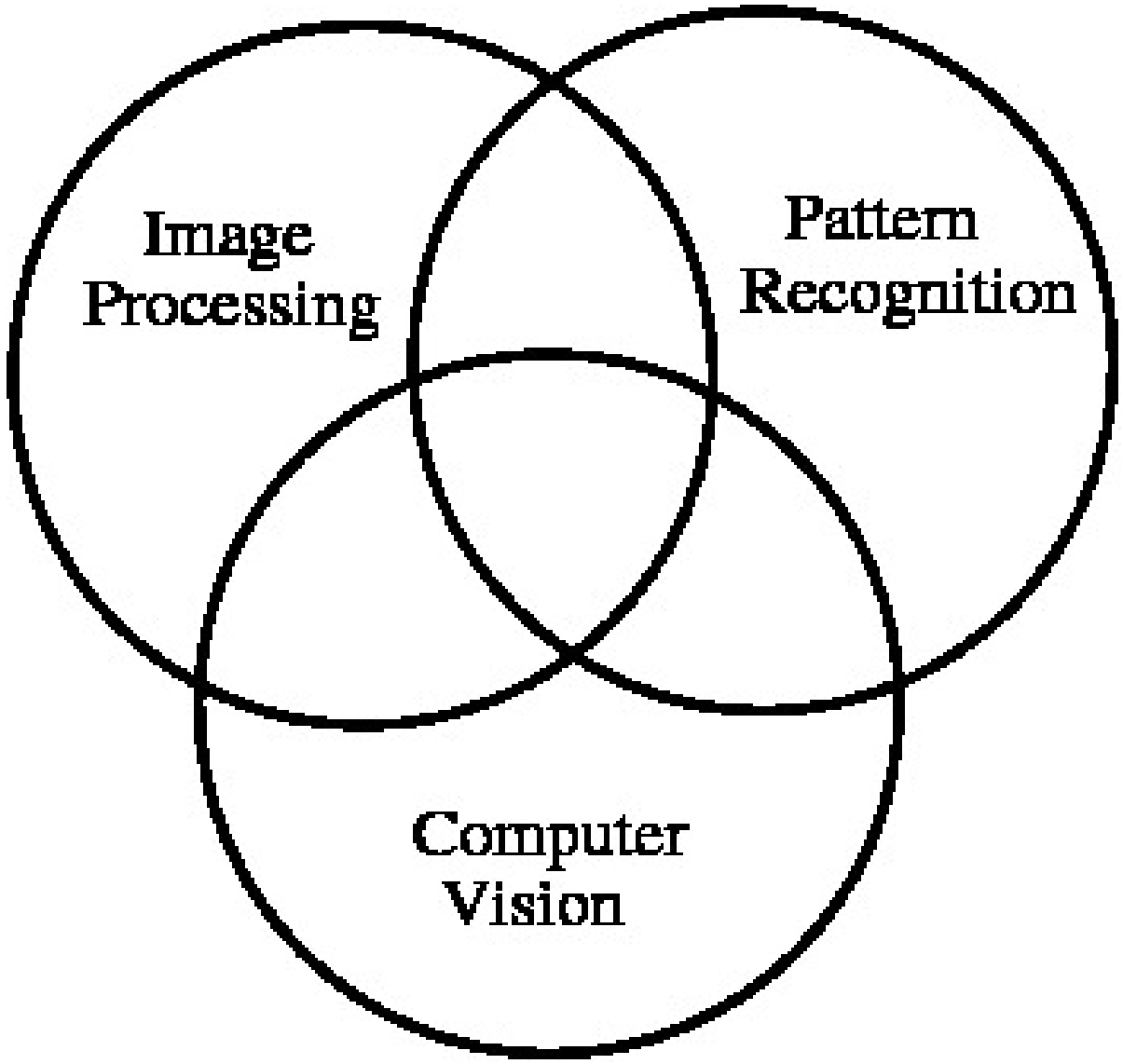
- **Computer Vision** refers to methods to "understand" in 3D what is happening in images. In a simple way, it can be thought of as the reverse of Computer Graphics. In Computer Graphics, one uses a 3D description of an object to make a 2D (pseudo-realistic) image of it. In Computer Vision one take one or more 2D images in a sequence and tries to recover 3D information from them. Some examples of **Low Level Computer Vision**:

1. Recovering 3D sensor motion and 3D scene structure from an image sequence made by a camera translating and rotating in some environment (autonomous vehicle navigation is an application that comes to mind).
2. Recovering 3D depth information from stereo analysis (two cameras) or shape from shading (relative depth from the way a scene is illuminated).
3. 2D and 3D image motion measurement (i.e. in a sequence of 2D intensity images or in a sequence of 3D MRI or range images).

- Some examples of **High Level Computer Vision** (basically AI/Robot Vision):

1. Scene Understanding
2. Object Tracking
3. Scene Manipulation
4. Path Planning (for robots in both known/unknown environments)

- There is a huge overlap in these 3 areas!!!
- For example, OCR is considered in both Pattern Recognition and Image Processing and Image segmentation via edge detection or region detection is considered in both Image Processing and Computer Vision.
- There is also a large overlap with **Image Processing** and **Signal Processing**, after all images are 2D signals!



Overlap of Image/Medical Image Processing, Pattern Recognition and Computer Vision Areas.

Histogram Equalization

- A histogram on an image is the count of the number of times each gray-value occurs in the image. The pseudo code below shows how to compute a histogram for image $f(x, y)$.

```
#define MAXIMUM_GRAYVALUE 255
int i,j;
unsigned char f[n][n]; /* an n*n image */
/* A size MAXIMIM_GRAYVALUE+1 histogram */
int histogram[MAXIMUM_GRAYVALUE+1];

for(i=0;i<MAXIMUM_GRAYVALUE;i++)
   histogram[i]=0;

for(i=0;i<n;i++)
for(j=0;j<n;j++)
   {
   histogram[(int) f[i][j]]++;
   }
```

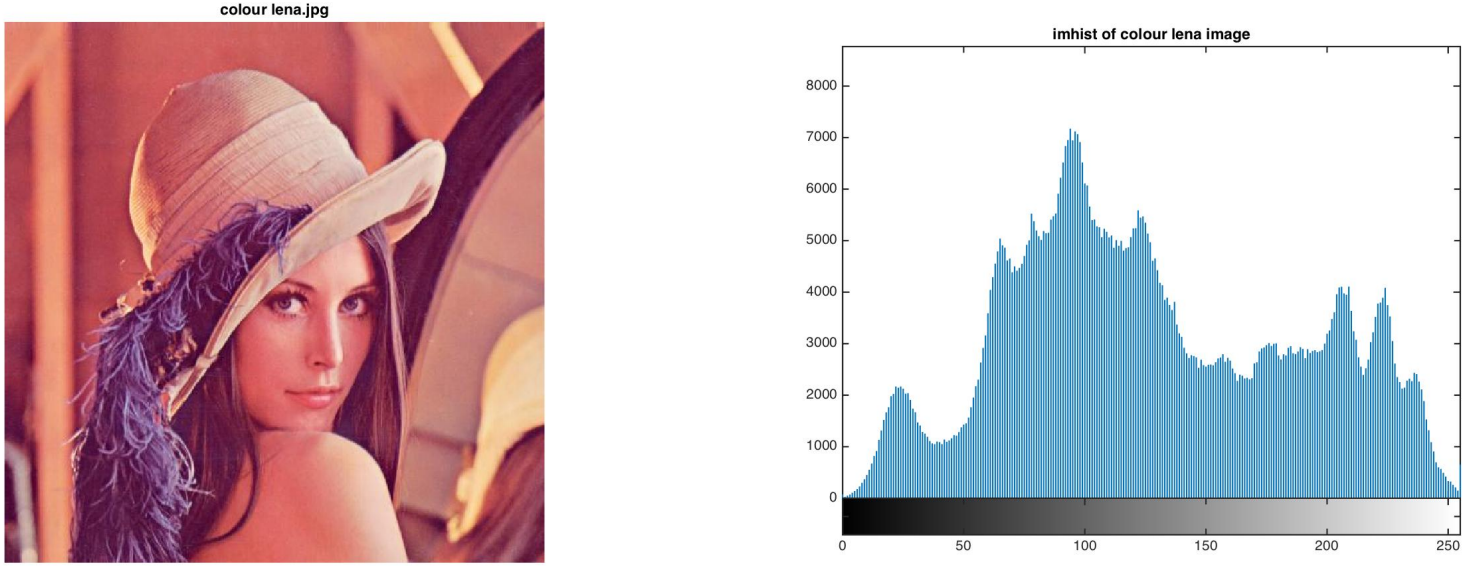
- In MatLab we can use `imhist` to do this. Consider the following MatLab code in **L19histogram_lena.m** shown below.

```
I=imread('lena.jpg');

imshow(I,[]);
title('colour lena.jpg');
print L19colour_lena.jpg -djpeg

figure
imshow(squeeze(I(:,:,1)),[]);
title('red lena.jpg');
print('L19red_lena.jpg','-djpeg');

figure
imshow(squeeze(I(:,:,2)),[]);
title('green lena.jpg');
print('L19green_lena.jpg','-djpeg');

figure
imshow(squeeze(I(:,:,3)),[]);
title('blue lena.jpg');
print('L19blue_lena.jpg','-djpeg');
```

```
figure
imhist(squeeze(I(:,:,1)))
title('imhist of red lena image');
print('L19red_lena_histogram.jpg','-djpeg');

figure
imhist(squeeze(I(:,:,2)))
title('imhist of green lena image');
print('L19green_lena_histogram.jpg','-djpeg');

figure
imhist(squeeze(I(:,:,3)))
title('imhist of blue lena image');
print('L19blue_lena_histogram.jpg','-djpeg');

figure
imhist(I(:))
title('imhist of colour lena image');
print('L19colour_lena_histogram.jpg','-djpeg');
```

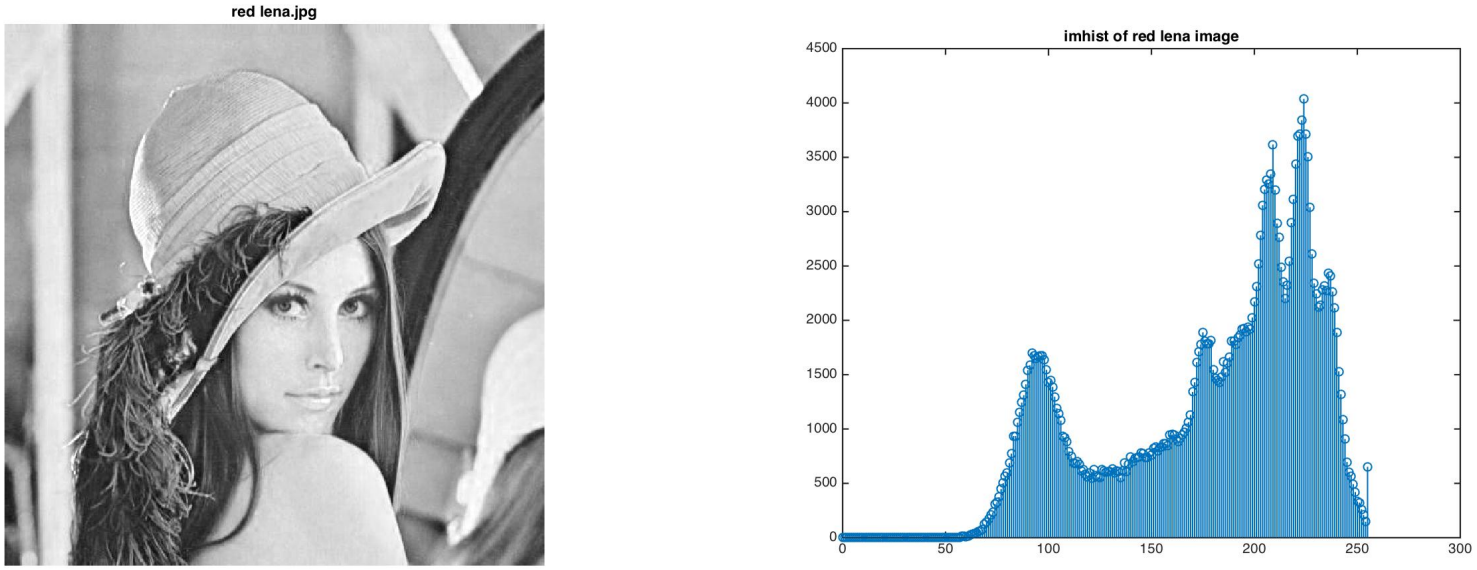
- We show the images and their histograms for the colour, blue, green and

red Lena images. Note that the colour image is collapsed into a 1D array using `(:)` before its histogram is computed. Thus all red, green and blue values are considered as a whole.



(a) (b)

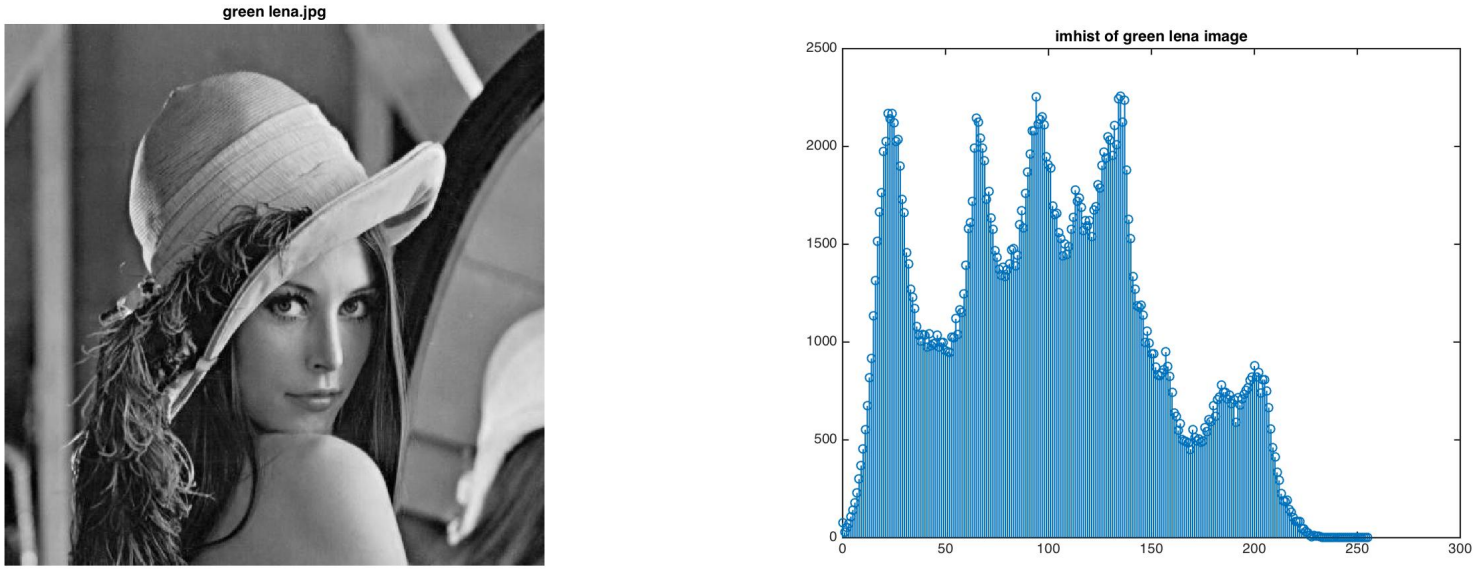
The colour lena image and its histogram. All 3 colour planes are considered as grayvalues using the (:) MatLab operator.



(a) (b)

The red lena image (as a grayvalue image) and its histogram.

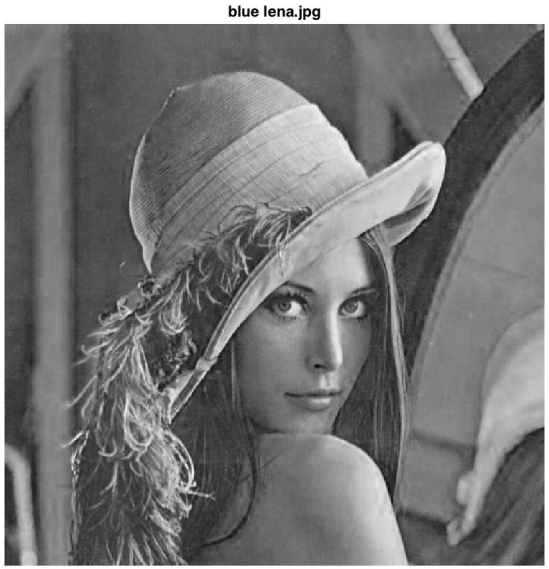
- If we treat a histogram as a function, $f(r)$, where r represents a pixel grayvalue, we can change the shape of this histogram and the image accordingly so that the histogram has a different shape (and the correspond-



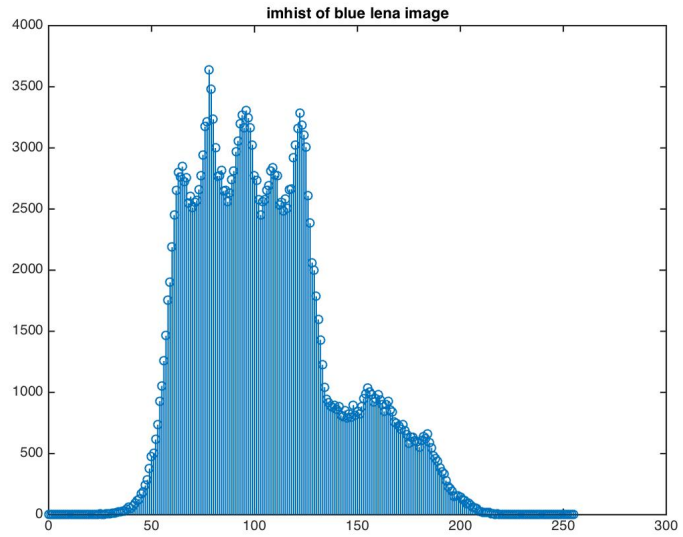
(a) (b)

The green lena image (as a grayvalue image) and its histogram.

ingly modified image now has this histogram). **Histogram modification** means the new histogram has a different distribution of grayvalues than the old histogram



(a)



(b)

The blue lena image (as a grayvalue image) and its histogram.

- **Histogram equalization** expands low contrast areas (uses more grayvalues in such areas) while at the same time compresses high contrast areas (uses fewer grayvalues in those areas. It does this by changing the

original image's histogram into a uniform histogram.

- **The Approach:** We map pixel values from f to s using a function T, where f is the original histogram and s is the modified histogram.

$$s(r) = T(f(r)), \quad 0 \leq r \leq r_{max}.$$

- If we treat $f(r)$ as a curve then the value of each $s(r)$ can be computed as

$$s(r) = \int_0^r f(w)dw.$$

  In other words, the value of $s(r)$ at each r value is the area under the curve in the original histogram between 0 and r.

- In the continuous case, the resulting histogram is always smooth, i.e. a straight line, but in the discrete case this is not so.

- Since the original histogram, $f(r)$, is discrete, T, must also be discrete. We made it discrete by rounding values to the highest integer $\Rightarrow$ a slightly bumpy histogram.

- **Example:** Consider a 64×64 image (4096 pixels) with only 8 grayvalues, 0-7. The following histogram has been computed from the image:

r	# pixels	Summation Values	Scaled Summation Values
0	790	790	1.35
1	1023	1813	3.10
2	850	2663	4.55
3	656	3319	5.67
4	329	3648	6.23
5	245	3893	6.65
6	122	4015	6.86
7	81	4096	7.0

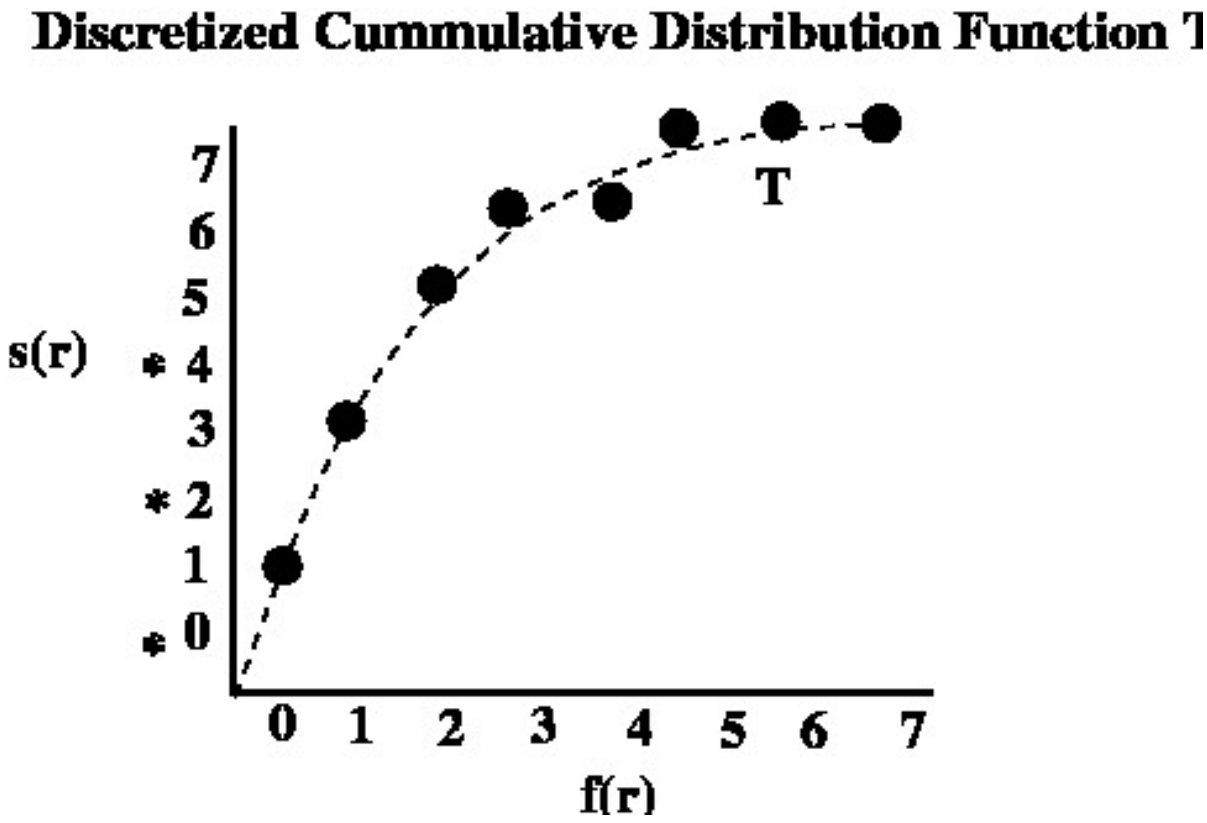
Scaled summation values are computed by multiplying the summation values by 7 (the maximum grayvalue) and dividing by 4096 (the number of pixels). The scaled summation values are the floating point $s(r)$ values, the result of applying T to $f(r)$.

- In the continuous case, the resulting histogram is always smooth, i.e. a straight line, but in the discrete case this is not so.

- Since the original histogram, $f(r)$, is discrete, T, must also be discrete. We made it discrete by rounding values to the highest integer $\Rightarrow$ a slightly bumpy histogram.

r	Scaled Summation Values	Discretized T Values
0	1.25	1
1	3.10	3
2	4.55	5
3	5.67	6
4	6.23	6
5	6.65	7
6	6.85	7
7	7.0	7

T is the **Discretized Cumulative Distribution Function**. There is no 0, 2 or 4. There are only 5 discrete values, 1, 3, 5, 6 and 7. Note that we assign all pixels at one or more grayvalues in the original image to a single grayvalue in the equalized histogram. The Figure below shows a rendering of this discrete function.

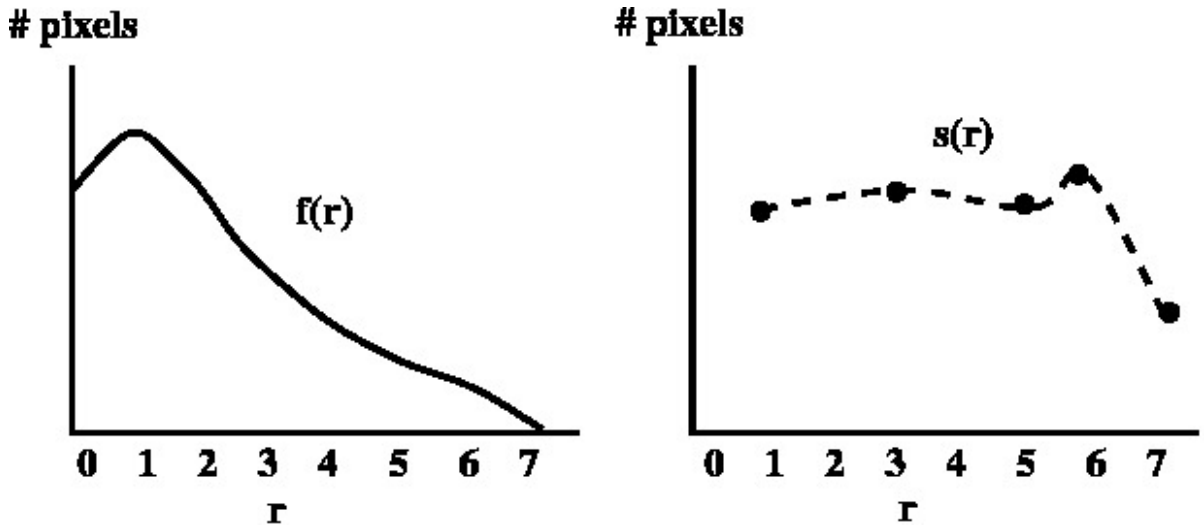


The Discretized Cumulative Distribution Function.

- Note that:

```
0 in r maps to 1 in s (790 values)
1 in r maps to 3 is s (1023 values)
2 in r maps to 5 in s (850 values)
3 and 4 in r map to 6 is s (656+329=985 values)
5, 6 and 7 in r map to 7 in s (245+122+81=448 values).
```

The Figure below shows the original and equalized histograms for comparison purposes.



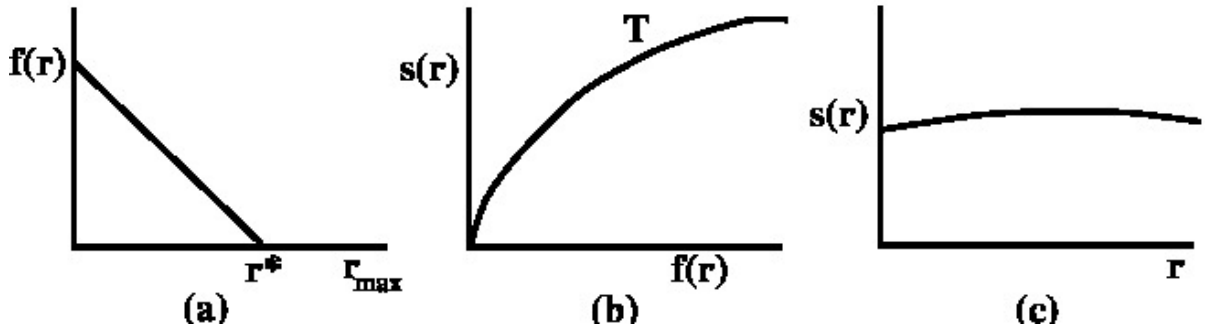
(a) The original histogram and (b) the equalized histogram.

- The discrete version of this integral is:

$$s(r) = \frac{\sum_{i=0}^{r} f(i)}{n} = s(r-1) + f(r),$$

where n is a scale parameter (n is the total number of pixels in the image divided by the maximum grayvalue of the image).

- For example, if the histogram looks like that in Figure below, then its resulting cumulative distribution function, i.e. T, looks like that in Figure b. Figure c shows the equalized histogram.



(a) The original histogram $(f(r)$, (b) the resulting cumulative distribution T and (c) the equalized histogram $s(r)$.

- Consider the code in **L19histogram_equalization_lena.m**. This code performs histogram equalization on the grayvalue versions of the red and green Lena images. We compare our results with those obtained using MatLab's **histeq** function.

```
close all
% Read colour lena image
I=imread('lena.jpg');

% display the green lena image
figure
greenI=squeeze(I(:,:,2));
imshow(greenI,[]);
title('green lena.jpg');
print('L19green_lena.jpg','-djpeg');

% display the red lena image
figure
redI=squeeze(I(:,:,1));
imshow(redI,[]);
title('red lena.jpg');
print('L19red_lena.jpg','-djpeg');

% compute the histogram of the green lena image
```

```
figure
[green_counts,bins]=imhist(greenI);
stem(bins,green_counts);
title('imhist of green lena image');
print('L19green_lena_histogram.jpg','-djpeg');

% compute the histogram of the red lena image
figure
[red_counts,bins]=imhist(redI);
stem(bins,red_counts);
title('imhist of red lena image');
print('L19red_lena_histogram.jpg','-djpeg');


% compute the cumulative histogram for the
% green and red lena images
fprintf('Cummumulative green and red counts as computed:');
fprintf('\ni=%3d %d  %d\n',1,green_counts(1),red_counts(1));
for i=2:256
  green_counts(i)=green_counts(i)+green_counts(i-1);
  red_counts(i)=red_counts(i)+red_counts(i-1);
  % fprintf('i=%3d %d  %d\n',i,green_counts(i),red_counts(i));
  end

% compute the max green and red counts for
% the original images - these will be used
% for scaling purposed for the histogram
% green and red counts returned by histeq
```

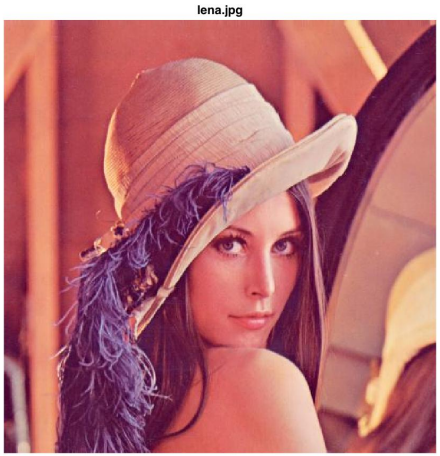
```
max_green_count=green_counts(256);
max_red_count=green_counts(256);

% display the green and red cumulative images
figure
stem(bins,green_counts);
title('cummulative histogram for green lena image');
print('L19green_cummulative_histogram.jpg','-djpeg');
figure
stem(bins,red_counts);
title('cummulative histogram for red lena image');
print('L19red_cummulative_histogram.jpg','-djpeg');

% Given grayvalues 0 to 255, compute the next
% grayvalues for the green or red lena image.
% Given a grayvalues in greenI or redI, find
% the corresponding value in equalized_greenI
% or equalized_redI by reading the new values
% off from the communicative histogram stored
% in counts.
% Before we can do this calculation we must
% normalize green_counts and red counts to be in [0,255]
green_counts=cast(255*green_counts./max(green_counts(:))+0.5,'uint8');
red_counts=cast(255*red_counts./max(red_counts(:))+0.5,'uint8');

for i=1:size(greenI,1) % same as size(redI,1)
for j=1:size(greenI,2) % same as size(redI,2)
   % Need to add 1 to avoid 0 index values
```

```
   equalized_greenI(i,j)=green_counts(greenI(i,j)+1);
   equalized_redI(i,j)=red_counts(redI(i,j)+1);
end
end

% display the green equalized image
figure
imshow(equalized_greenI)
title('histogram equalized green image');
print('L19computed_equalized_green_image.jpg','-djpeg');

% display the red equalized image
figure
imshow(equalized_redI,[])
title('histogram equalized red image');
print('L19computed_equalized_red_image.jpg','-djpeg');

% display the histogram of the green equalized image
figure
[green_counts,bins]=imhist(equalized_greenI);
stem(bins,green_counts);
title('Computed Equalized Green Histogram');
print('L19computed_equalized_green_histogram.jpg','-djpeg');

% display the histogram of the red equalized image
figure
[red_counts,bins]=imhist(equalized_redI);
stem(bins,red_counts);
```

```
title('Computed Equalized Blue Histogram');
print('L19computed_equalized_red_histogram.jpg','-djpeg');

% histogram equalized the original green lena image using histeq
[greenI2,greenT] = histeq(greenI);
figure
imshow(greenI2,[]);
title('Green image histogram equalized by histeq');
print('L19histeq_equalized_green_image.jpg','-djpeg');

% histogram equalized the original red lena image using histeq
[redI2,redT] = histeq(redI);
figure
imshow(redI2,[]);
title('Blue image histogram equalized by histeq');
print('L19histeq_equalized_red_image.jpg','-djpeg');

figure
% histeq gives greenT in normalized form
% undo the normalization before plotting
greenT=greenT*max_green_count;
stem(bins,greenT);
title('The unnormalized histeq green transformation T');
print('L19histeq_equalized_green_transformation.jpg','-djpeg');

figure
% histeq gives redT in normalized form
% undo the normalization before plotting
```

```
redT=redT*max_red_count;
stem(bins,redT);
title('The unnormalized histeq red transformation T');
print('L19histeq_equalized_red_transformation.jpg','-djpeg');
```

- For the **lena.jpg** image the figure below shows the red and green images (1^{st} and 2^{nd} colour planes).



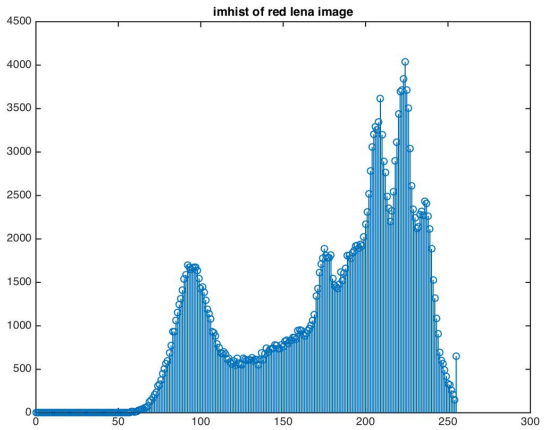
(a)

(b)

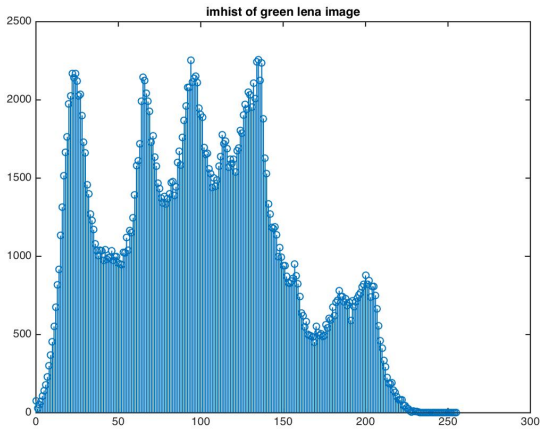
(c)

(a) The lena.jpg image, (b) its red image and (c) its green image.

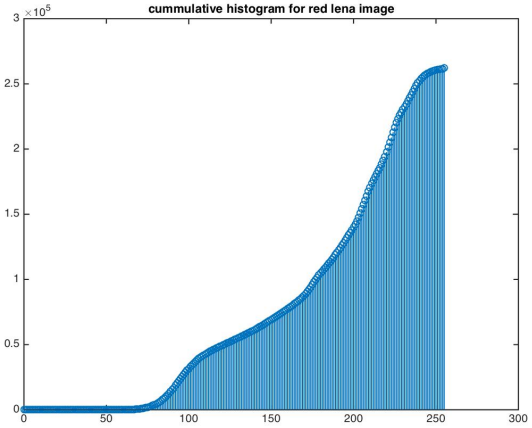
- Next, we compute the histograms of these red and green images and then their cumulative histograms.



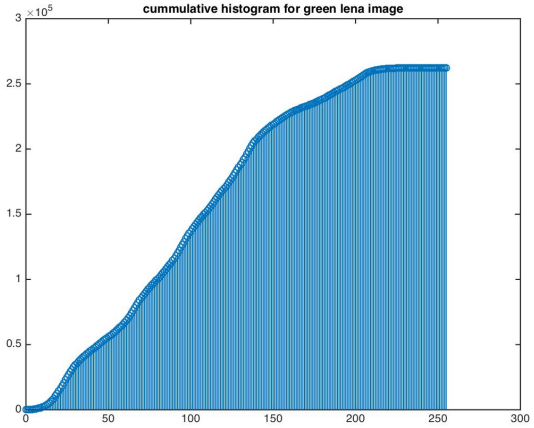
(a)



(b)



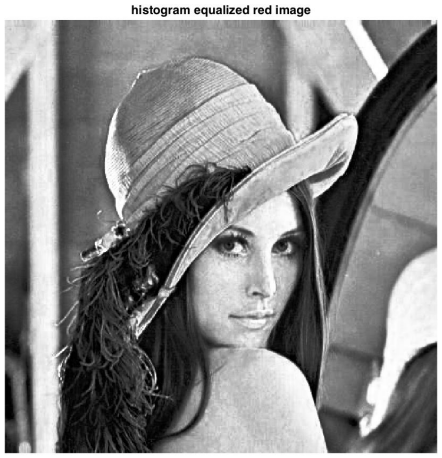
(c)



(d)

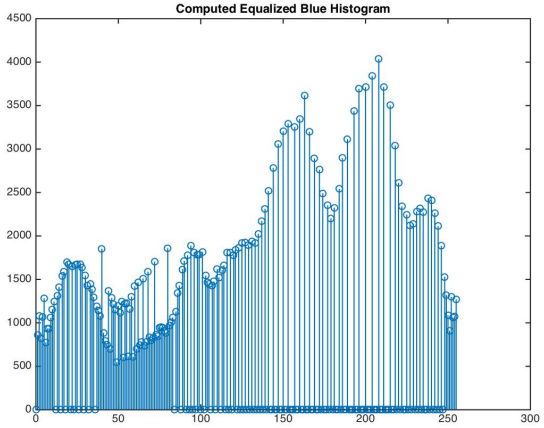
(a) and (b) The red and green histograms and (c) and (d) their computed cumulative histograms.

- Next, we compute the equalized red and green images. We also show the equalized image histograms below. We can see the equalized images are much clearer than the original red and green images. The equalized histograms are much ”flatter” than the original red and green images’ histograms.

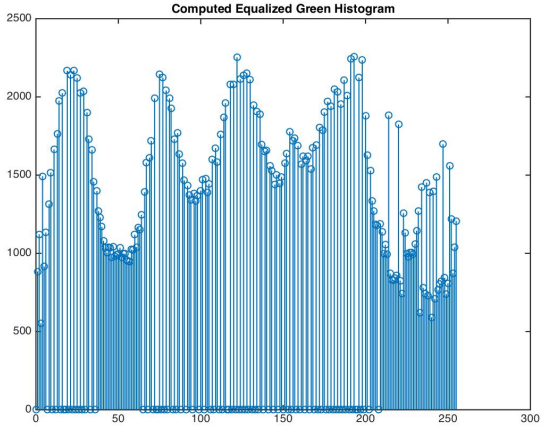


(a)

(b)



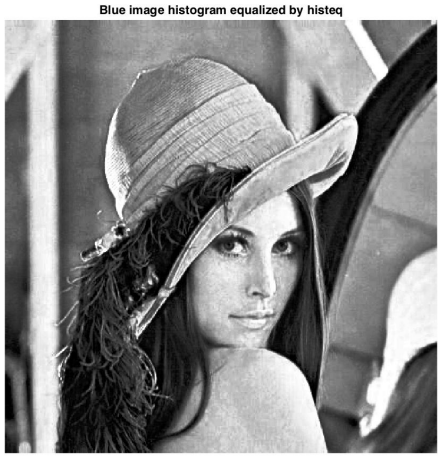
(c)



(d)

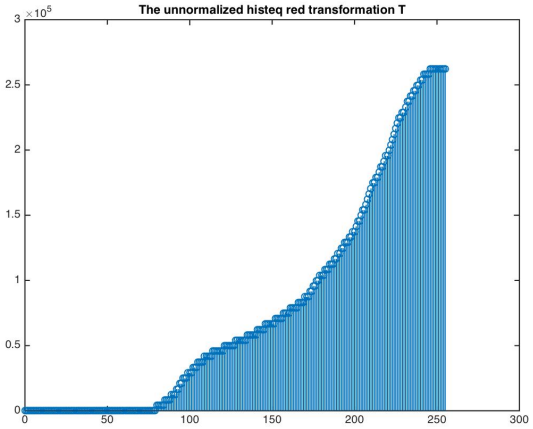
(a) and (b) The equalized images and (c) and (d) their histograms.

- Finally, we show the **histeq**'ed images and their histograms. Note that these images are identical to the images on the previous slide. We also plot the cumulative histograms returned by **histeq** and see that they are the same as our computed cumulative histograms.

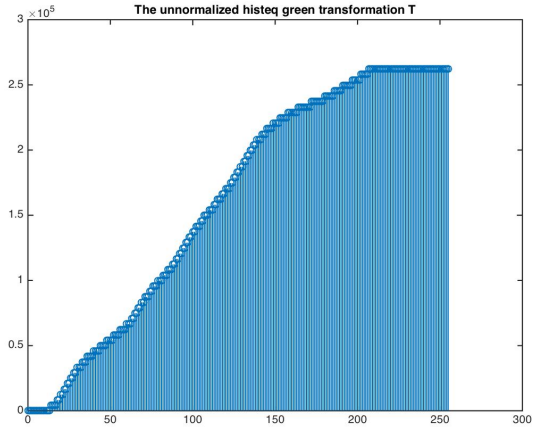


(a)

(b)



(c)



(d)

(a) and (b) The **histeq** equalized red and green images and (c) and (d) the cumulative histograms (the T transformation) returned by **histeq**.

Histogram Equalization of a Chest Xray Image

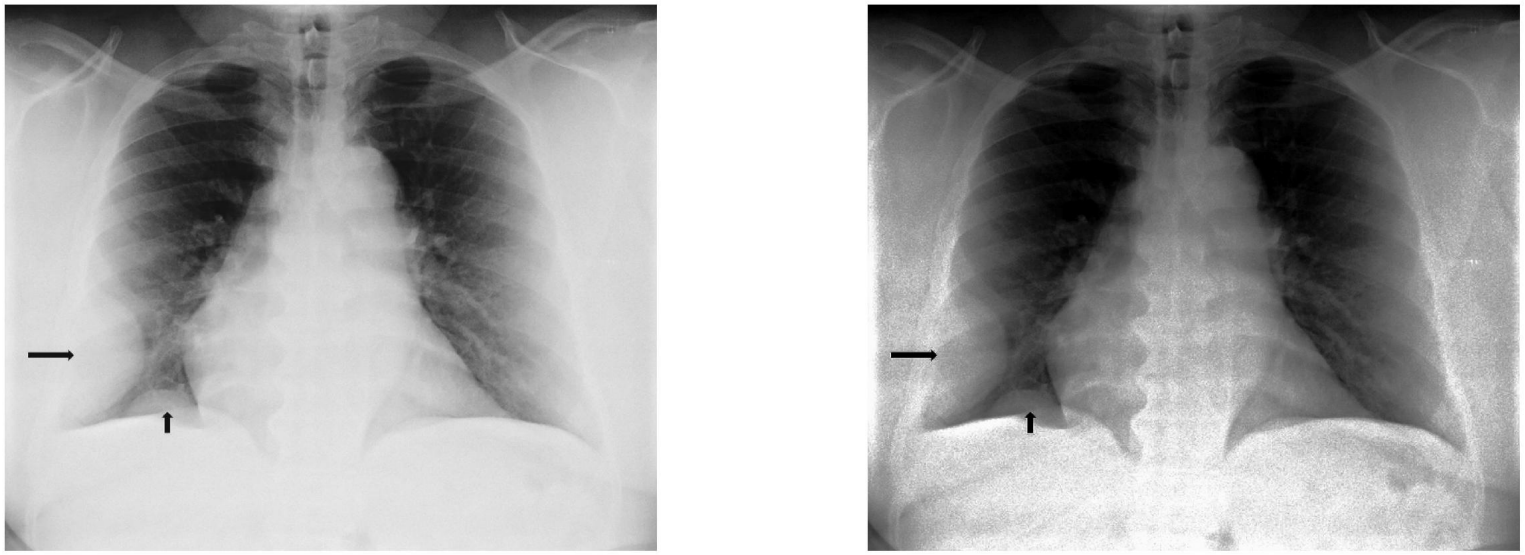
- Consider the following MatLab code (in **L19chest_equalization.m**):

```
close all
I1=imread('chest_xray2.jpg');
size(I1)
% I1(:,:,1), I1(:,:,2) and I1(:,:,3) are the same
% as the chest image is a grayvalue colour image
% Set I1 to the 2D green image of 'chest_xray2.jpg'.
I1=squeeze(I1(:,:,2));

figure
imshow(I1,[]);
print original_low_contrast_xray_chest_image.jpg -djpeg

figure
imhist(I1(:));
print original_low_contrast_xray_chest_image_histogram.jpg -djpeg

I2=histeq(I1);

figure
```

```
imshow(I2,[]);
print equal_contrast_xray_chest_image.jpg -djpeg

figure
imhist(I2(:));
print equal_constrast_xray_chest_image_histogram.jpg -djpeg
```

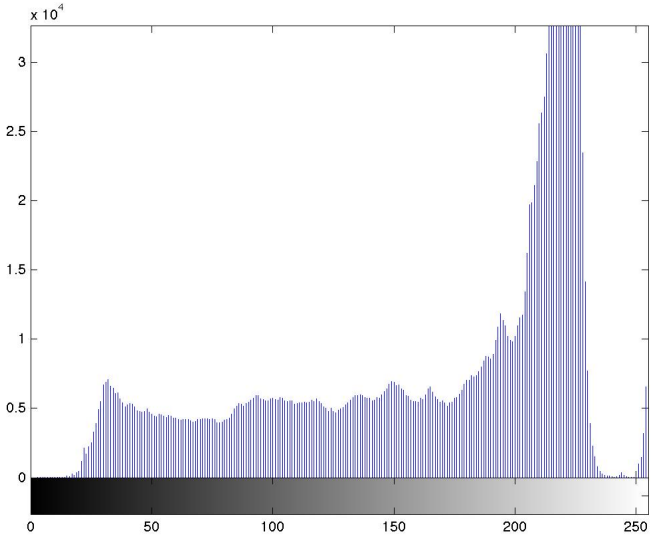
- The left figure below shows the original image while the right figure shows the histogramed equalized image.

- The 2 figures below show the histograms of the original image and the histogramed image.

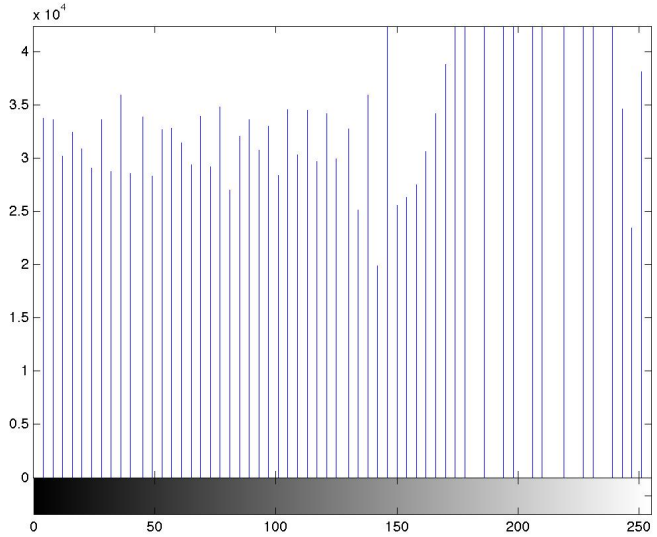


(a) (b)

(a) Original low contrast chest xray image and (b) the histogramed equalized version of that image.



(a)



(b)

(a) Original low contrast xray chest image histogram and (b) Histogram equalized image's histogram. Note how the histogram in (b) is much more uniform than the histogram in (a).

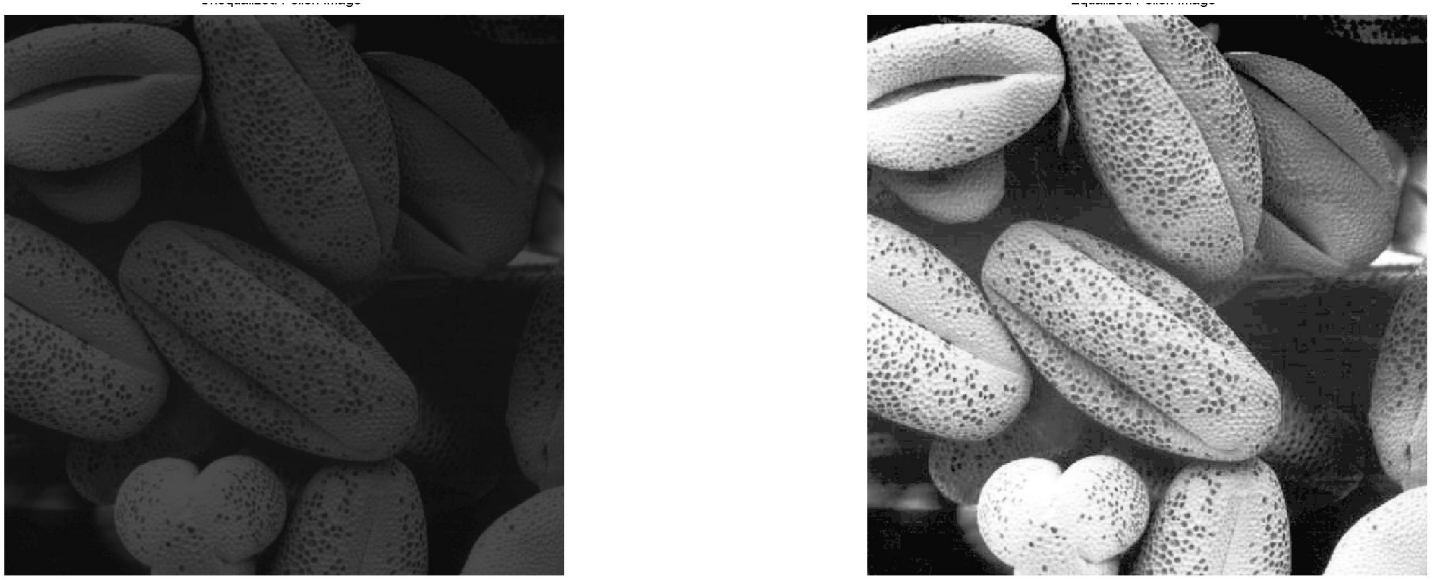
Histogram Equalization of an Electron Microscope Image of Pollen

- Let's look at another histogram equalization example of an electron microscope image of pollen, magnified about 700 times. The image is dark and has low dynamic range. The histogram equalized image show much improved contrast range. The code (in **L19pollen_equalization.m** show the processing done:

```
close all
I=imread('pollen.tif');
figure
imshow(I);
title('Unequalized Pollen Image');
print('unequalized_pollen_image.jpg','-djpeg');
figure
imhist(I)
title('Unequalized Histogram');
```

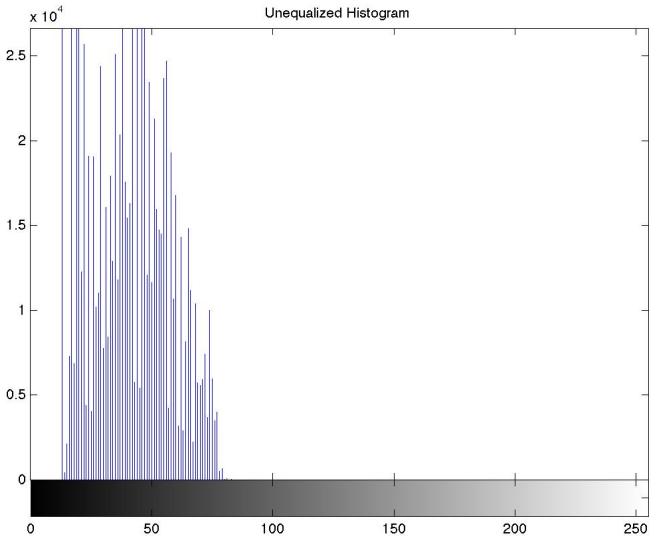
```
print('unequalized_pollen_histogram.jpg','-djpeg');

G=histeq(I,256);
figure
imshow(G);
title('Equalized Pollen Image');
print('equalized_pollen_image.jpg','-djpeg');
figure
imhist(G)
title('Equalized Histogram');
print('equalized_pollen_histogram.jpg','-djpeg');
```

- The images below show the unequalized and equalized images and their histograms.

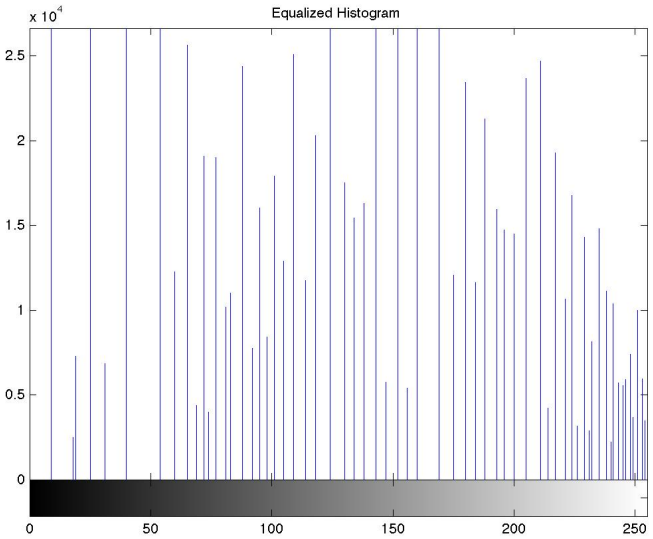


(a) (b)

(a) Unequalized and (b) Equalized Pollen images.



(a)



(b)

(a) Unequalized and (b) Equalized Pollen histograms.

Histogram Equalization of Colour Images

- How might one histogram equalize a colour image. **imhisteq** only operates on 2D arrays (which are necessarily grayvalue images). The MatLab code in **L19colour_yiq_equalization.m** below shows the wrong way and then the right way to do this.

```
fRGB=imread('lena.jpg');
imshow(fRGB,[]);

% NTSC computes the YIQ colour transformation
% This transformation allows downwards
% compatibility with black and white and
% colour images for TV
% Y is the intensity image
% I and Q encode the colour information
% NTSC - National Television System Committee
fYIQ=rgb2ntsc(fRGB);

% Perform histogram equalization on Y intensity
% image and on the I (inphase) and Q (quadrature)
```

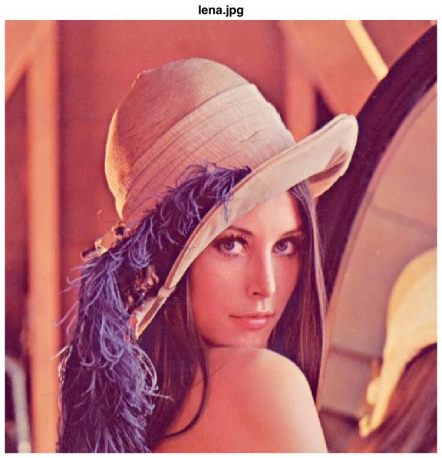
```
% images and recombined into colour image
g1=squeeze(fYIQ(:,:,1));
g2=squeeze(fYIQ(:,:,2));
g3=squeeze(fYIQ(:,:,3));

% display and print individual Y, I and Q images
figure
imshow(g1,[]);
title('Y lena image');
print('Y_grayvalue_lena.jpg','-djpeg');
figure
imshow(g2,[]);
title('I lena image');
print('I_grayvalue_lena.jpg','-djpeg');
figure
imshow(g3,[]);
title('Q lena image');
print('Q_grayvalue_lena.jpg','-djpeg');

h1=histeq(g1);
h2=histeq(g2);
h3=histeq(g3);
% combine the YIQ equalized images into a
% colour RGB image
fYIQ(:,:,1)=h1;
fYIQ(:,:,2)=h2;
fYIQ(:,:,3)=h3;
fRGB=ntsc2rgb(fYIQ);
```

```
figure
imshow(fRGB);
title('All YIQ Images Histogram Equalized');
print lena.yiq.histogram_equalized.jpg -djpeg

% Perform histogram equalization on Y image only
fYIQ(:,:,1)=h1; % h1 is Y equalized image
fYIQ(:,:,2)=g2; % g2 is original I image plane
fYIQ(:,:,3)=g3; % g3 is original Q image plane
fRGB=ntsc2rgb(fYIQ);
figure
imshow(fRGB);
title('Only Y Image Histogram Equalized');
print lena.y.histogram_equalized.jpg -djpeg

% Perform histogram equalization on I image only
fYIQ(:,:,1)=g1;
fYIQ(:,:,2)=h2;
fYIQ(:,:,3)=g3;
fRGB=ntsc2rgb(fYIQ);
figure
imshow(fRGB);
title('Only I Image Histogram Equalized');
print lena.i.histogram_equalized.jpg -djpeg

% Perform histogram equalization on Q image only
fYIQ(:,:,1)=g1;
fYIQ(:,:,2)=g2;
```

```
fYIQ(:,:,3)=h3;
fRGB=ntsc2rgb(fYIQ);
figure
imshow(fRGB);
title('Only Q Image Histogram Equalized');
print lena.q.histogram_equalized.jpg -djpeg
\end{itemize}
```

- The figure below shows the original lena.jpg images and its Y, I and Q images computed using the **rgb2ntsc** transformation function. The Y image is the luminance or intensity image while the I (inphase) and Q (quadrature) images encode colour only. The separates intensity and colour image information. This is the image transformation scheme used by NTSC (National Television System Committee) to allow backwards compatibility of colour TV with black and white TV.

- The Y image is a grayvalue version of the lena.jpg image (roughly 60% green, 30% red and 10% blue of the three colour planes).



(a)

(b)

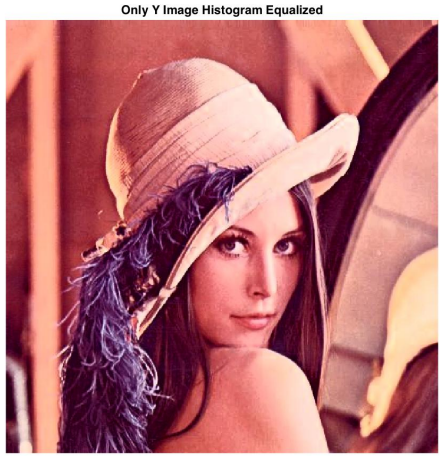
(c)

(d)

(a) The lena.jpg image and (b), (c) and (d) the Y, I and Q images on lena.jpg computed **rgb2ntsc** MatLab function.

- The figure below the colour images resulting from:
  1. histogram equalizing all three Y, I and Q images and then recombining these results back into a colour image,
  2. histogram equalizing the Y image and then recombining this image and the original I and Q images back into a colour image,
  3. histogram equalizing the I image and then recombining this image and the original Y and Q images back into a colour image and
  4. histogram equalizing the Q image and then recombining this image and the original Y and I images back into a colour image.
- We can see the second operation (equalizing the Y image only) produces the correct histogram equalized colour images. The other three colour images are "interesting"!!!

(a)



(b)

(c)

(d)

(a) The YIQ equalized image, (b) the Y image only equalized, (c) the I image only equalized and (d) the Q image equalized. We can see that (b) yields the correct histogram equalized colour image.

Median Filtering and Salt & Pepper Noise

- Salt and Pepper noise is where random image pixels are set to 255 (white salt) or 0 (black pepper). We attempt to remove this noise using **median** filtering. At each pixel, for a small neighbourhood (for example 3×3 neighbourhoods) we sort the pixel values in the neighbourhood and replace the pixel with its **median** value (the value in the middle of the sorted neighbourhood list). We illustrate this idea using **L19median_small_image.m** below:

```
A=[50   62  70  83  90 103 110;
   55   61  75  81  95 103 115;
   62   70  81  95  93 110 120;
   65   71  85  93 104 118 125;
   71   86  90  80 110 120 130;
   73   81  95  80 115 125 130;
   81   88  90 108 120 125 140];
```

```
% save the median filter image of A in B
% leave A alone so that all filtering
% calculations use original data
B=A;

% add white noise at element (3,3)
A(3,3)=255;
% add black noise at element (5,5)
A(5,5)=0;

% 3*3 median filter A(3,3)
% s1 is 3*3 neighbourhood of A(3,3)
s1=A(2:4,2:4);
% sort s1
s1=sort(s1(:))'
% replace A(3,3) by its median value, s1(5)
B(3,3)=s1(5);
fprintf('A(3,3)=%3d replaced by %3d in B\n',A(3,3),B(3,3));

% 3*3 median filter A(5,5)
% s2 is 3*3 neighbourhood of A(5,5)
s2=A(4:6,4:6);
% sort s2
s2=sort(s2(:))'
% replace A(5,5) by its median value, s2(5)
B(5,5)=s2(5);
fprintf('A(5,5)=%3d replaced by %3d in B\n',A(5,5),B(5,5));
```

```
% 3*3 median filter A(3,5) and A(5,3)
% s3 is the 3*3 neighbourhood of A(3,5)
s3=A(2:4,4:6);
% s4 is the 3*3 neighbourhood of A(5,3)
s4=A(4:6,2:4);
% sort s3
s3=sort(s3(:))'
% replace A(3,5) by its median value, s3(5)
B(3,5)=s3(5);
% sort s4
s4=sort(s4(:))'
% replace A(5,3) by its median value, s4(5)
B(5,3)=s4(5);

fprintf('A(3,5)=%3d replaced by %3d in B\n',A(3,5),B(3,5));
fprintf('A(5,3)=%3d replaced by %3d in B\n',A(5,3),B(5,3));
```

- The output that results to this point in the program is:

```
s1=61 70 71 75  81  85  93  95 255
A(3,3)=255 replaced by  81 in B
s2= 0 80 80 93 104 115 118 120 125
A(3,3)=255 replaced by  81 in B
A(5,5)=  0 replaced by 104 in B
s3= 81 93 93 95 95 103 104 110 118
```

```
A(3,5)= 93 replaced by  95 in B
s4= 71 80 80 81 85  86  90  93  95
A(5,3)= 90 replaced by  85 in B

Original Image:
 50  62  70  83  90 103 110
 55  61  75  81  95 103 115
 62  70 255  95  93 110 120
 65  71  85  93 104 118 125
 71  86  90  80   0 120 130
 73  81  95  80 115 125 130
 81  88  90 108 120 125 140
```

- `A(3,3)` was 255 (salt noise) and is replaced by 81 while `A(5,5)` was 0 (pepper noise) and is replaced by 104. 81 and 104 are much more similar to the neighbouring pixel values than 255 or 0 were.
- `A(3,5)` and `A(5,3)` were 93 and 90 respectively and are replaced by 95 and 85, values that are very similar to the original values.
- Median filtering at these 4 pixel locations improved the image (at the salt

and pepper locations) but left the image at non-noise pixels reasonably alone.

- The last part of this code uses the **medfilt2** MatLab function to median filter the entire 7×7 array (the 2 refers to 2D). We use the **'symmetric'** option of **medfilt2** to handle the border calculations. Basically, the 1^{st} and last columns and rows are duplicated and the corner pixel values are mirror reflected diagonally. This allows the calculations at border points to proceed. Of course, this is "wrong" but better than the alternates: (1) make the rows and columns and reflected corner points all 0 (padding with zeros), (2) use wraparound so that the 0^{th} column is the n^{th} column and the $(n+1)^{th}$ column is the 0^{th} column or (3) median filter on the array shrank so that median values are not computed for the outer columns

or rows (this makes the image 5×5 in our case).

- We can see that the median filter image at `A(3,3)`, `A(3,5)`, `A(5,3)` and `A(5,5)` for our median filtering and medfilt2's filtering are the same.

```
% perform MatLab median filtering
% Option 'symmetric' handles the border by
% using mirror reflections to get data when
% array indexing goes outside the array.
% For 3*3 median filtering, the 1st and last columns
% and the 1st and last rows are duplicated.
% The 4 corner points are mirror reflected
% For example:
%     1 2 3
%     4 5 6
%     7 8 9
% becomes
%   1 1 2 3 3
%   1 1 2 3 3
%   4 4 5 6 6
%   7 7 8 9 9
%   7 7 8 9 9
% Now median filtering upper left corner
```

```
% 1 uses the neighbourhood:
%     1 1 2
%     1 1 2
%     4 4 5
% Of course, this is not correct but
% looks better than the alternatives of
% padding the left/right columns and above/below
% row with 0's or making the median results
% have 2 fewer rows and columns

% the 2 here refers to 2D
C=medfilt2(A,'symmetric');

fprintf('\nOriginal Image:\n');
for i=1:size(A,1)
fprintf('%3d %3d %3d %3d %3d %3d %3d\n',A(i,:));
end
fprintf('\nOur Median Filtered Image\n');
fprintf('with just 4 pixels filtered:\n');
for i=1:size(B,1)
fprintf('%3d %3d %3d %3d %3d %3d %3d\n',B(i,:));
end
fprintf('\nMatLab Median Filtered Image:\n');
for i=1:size(C,1)
fprintf('%3d %3d %3d %3d %3d %3d %i3d\n',C(i,:));
end

fprintf('\n');
```

```
fprintf('A(3,3)=%3d replaced by %3d in C\n',A(3,3),C(3,3));
fprintf('A(5,5)=%3d replaced by %3d in C\n',A(5,5),C(5,5));
fprintf('A(3,5)=%3d replaced by %3d in C\n',A(3,5),C(3,5));
fprintf('A(5,3)=%3d replaced by %3d in C\n',A(5,3),C(5,3));
```

- The output for this code is:

```
Our Median Filtered Image
with just 4 pixels filtered:
 50  62  70  83  90 103 110
 55  61  75  81  95 103 115
 62  70  81  95  95 110 120
 65  71  85  93 104 118 125
 71  86  85  80 104 120 130
 73  81  95  80 115 125 130
 81  88  90 108 120 125 140

MatLab Median Filtered Image:
 55  62  70  83  90 103 1103d
 61  62  75  90  95 103 1103d
 62  70  81  93  95 110 1183d
 70  71  86  93  95 118 1203d
 71  81  85  90 104 120 1253d
 81  86  88  90 115 125 1303d
 81  88  90 108 120 125 1303d

A(3,3)=255 replaced by  81 in C
A(5,5)=  0 replaced by 104 in C
A(3,5)= 93 replaced by  95 in C
A(5,3)= 90 replaced by  85 in C
```

Denoising Lena Image with Salt and Pepper Noise

- Consider adding 20% salt and pepper noise to a grayvalue image of the lena image (we use the green plane as the grayvalue version of this image).
- We add 20% salt and pepper noise to that image. Salt noise means a randomly pixel is changed to white (255) while Pepper noise means a randomly pixel is changed to black (0).
- Median filtering works by sorting the pixels in the 3×3 neighbour of a pixel and replacing that pixel with its median value. In this way, "outlier" pixels (most likely noise are removed).

- Consider the following code (in **L19median_filtering_salt_pepper_noise.m**):

```
close all
% read image
I=imread('lena.jpg');

% close green image of I image
G=squeeze(I(:,:,2));
figure
imshow(G,[]);
print lena_green_image.jpg -djpeg

% 20% salt and pepper noise
H=imnoise(G,'salt & pepper',0.2);
figure
imshow(H,[]);
print green_lena_salt_pepper_20_percent_image.jpg -djpeg

% apply 3*3 median filtering once
I=medfilt2(H);
figure
imshow(I,[]);
print green_lena_median_filtering_1_image.jpg -djpeg

% apply 3*3 median filtering again
J=medfilt2(I);
figure
imshow(J,[]);
```

```
print green_lena_median_filtering_2_image.jpg -djpeg
```

- The first two figures below the original green lena image (as a grayvalue image) and that image with 20% randomly placed salt and pepper noise.

(a)

(b)

(a) Original “green” grayvalue image and (b) that image with 20% salt and pepper noise added to it.

- The last two figures show the lena images after 1 and 2 applications of 3×3 median filtering.

(a)

(b)

(a) One application of median filtering and (b) a second application of median filtering. The image is pretty much "cleaned" of salt and pepper noise after 2 applications of this filter.

Image Registration

- An application of image registration was shown in a MatLab webinar by Garima Sharma and Andy Thé.[1] Suppose you have 2 images of a person, one image is a colour Webcam image and the second is an infrared image. The 2 images are acquired from roughly the same viewpoint and are the same size (but it is not a problem if the latter is not true as MatLab provides lots of tools to resize images).

- The Task: determine the temperature of the person with these images! This type of software would be especially valuable at airports when it is necessary to determine if incoming passengers are sick (think about

[1] http://www.mathworks.com/company/newsletters/articles/automating-image-registration-with-matlab.html

how useful this would have been during the SARS epidemic in the early 2003).

- We use the MatLab Image Acquisition toolbox to acquire the IR and Webcam images of a man (MatLab kindly supplied these images, please note they are copyrighted images and not to be dispersed to the world!!!). Below, we show this acquisition MatLab code. All the code is commented out as the colour and IR images are read from files here.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% set these parameters and take a colour Webcam and a IR image
%
% IR Camera Object creation and configuration:
% irCam = imaq.VideoDevice('gige',1,'Mono16');
% irCam.ReturnedDataType = 'native';
% irCam.DeviceProperties.IRFormat = 'TemperatureLinear100mK';
% irCam.DeviceProperties.ObjectDistance = 0.91;
% irCam.DeviceProperties.AtmosphericTemperature = 295.15;
% irCam.DeviceProperties.ObjectEmissivity = 0.98;
```

```
%% This stuff needed if you are capture image
%% right now, we read our images from files
%% so this is not needed now

%% Webcam object creation
% webCam = imaq.VideoDevice('winvideo');

%% Acquire snapshots
% irImage = step(irCam);
% webImage = step(webCam);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- We read these images in (the complete MatLab code is in **L19fever.m**).
- The fixed image is the IR image and the moving image is the Webcam image. We register the moving image onto the fixed image, use the Viola-Jones eye detection algorithm in the Computer Vision Systems toolbox to detect the eyes in this warped image (as a bounding box) and then read off the eyes' temperature from the same bounding box in the IR image.

- The MatLab code to read these images and display them individually and in montage fashion (side by side) is given below:

```
%% Read in the image pair
Fixed=imread('IR_image.png');
Moving=imread('Webcam_image.png');
% Convert colour image to grayvalue image
Moving=rgb2gray(Moving);

%% View and save the 2 images
figure
imshow(Moving,[]);
title('\fontsize{16}Moving Image (Webcam Image)');
print('Moving_Image.jpg','-djpeg');

figure
imshow(Fixed,[]);
title('\fontsize{16}Fixed Image (IR Image)');
print('Fixed_Image.jpg','-djpeg');

%% View the image side by side in montage style
figure
% montage places the 2 images, Fixed and Moving, side by side
imshowpair(Fixed,Moving,'montage');
title('\fontsize{16}Images in Montage Fashion');
print('Montage_Images.jpg','-djpeg');
```

- The Figure below shows the montage image pair (the same as the 2 individual images placed side by side).

Montage display of the moving (Webcam) and fixed (IR) images using imshowpair.

- Next we perform the **default** pure translation registration of the fixed (IR image) and moving (Webcam image) images. We use the **imregister()** function to do this. The form of this command is:

```
registered=imregister(moving,fixed,...
          transform_type,optimizer,metric);
```

  where the **moving** image is warped to be registered to the **fixed** image. **transform_type** is one of **'translation'** (only), **'rigid'** (translation and rotation), **'affine'** (translation, rotation, scale and sheer) or **'similarity'** (translation, rotation and scale, but not shear). Most likely, **imregister** uses the intensity based ICP (Iterative Closest Point) method to do its registration.

- What about **metric** and **optimizer**?

- Getting good results from optimization-based image registration usually requires modifying **optimizer** and/or **metric** settings for the pair of images being registered. The **imregconfig** function provides a default configuration that should only be considered a starting point for these values. We use:

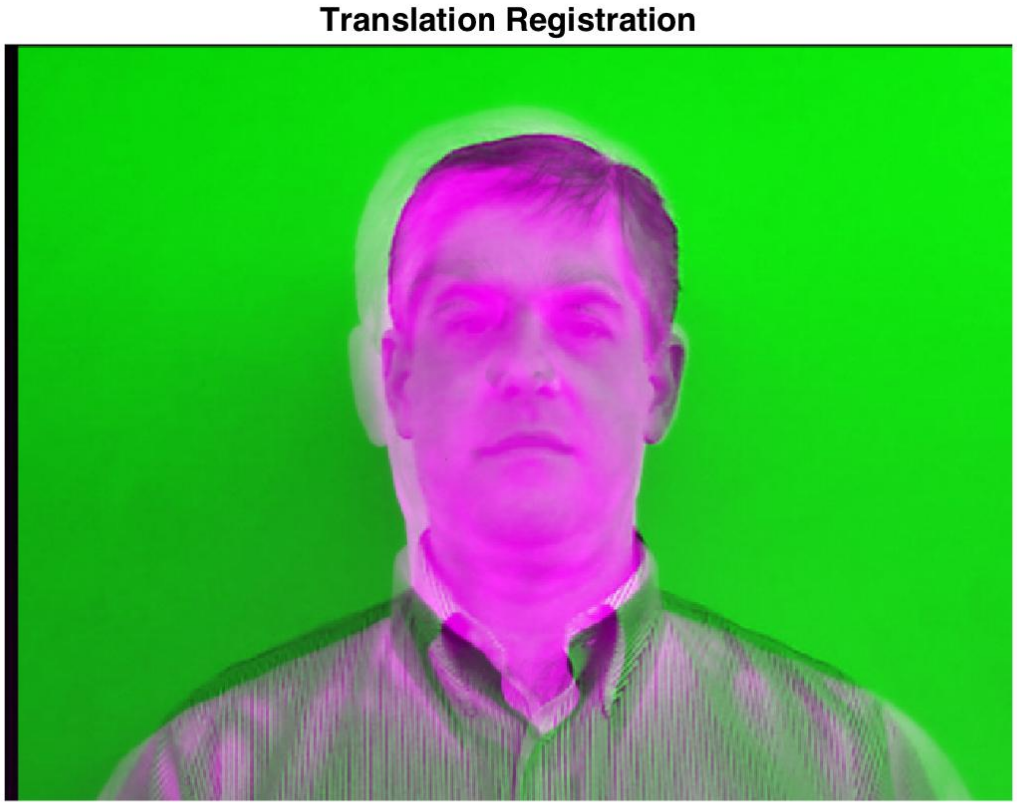
```
% No colon: print the values of optimizer and metric
[optimizer,metric] = imregconfig('Multimodal')
registered=imregister(moving,fixed,...
          transform_type,optimizer,metric);
```

to obtain the default values for **optimizer** and **metric** and then do the registration. Since we are using Webcam and IR images, we use node **Multimodal** (versus **Monomodal**). The default values for **optimizer** and **metric** used by `imregconfig` are given below:

```
optimizer =
  registration.optimizer.OnePlusOneEvolutionary
  Properties:
         GrowthFactor: 1.050000e+00
              Epsilon: 1.500000e-06
        InitialRadius: 6.250000e-03
    MaximumIterations: 100
metric =
  registration.metric.MattesMutualInformation
  Properties:
    NumberOfSpatialSamples: 500
     NumberOfHistogramBins: 50
               UseAllPixels: 1
```

These parameters are used by the **imregister** algorithm.

- The figure below shows the translation registration result.



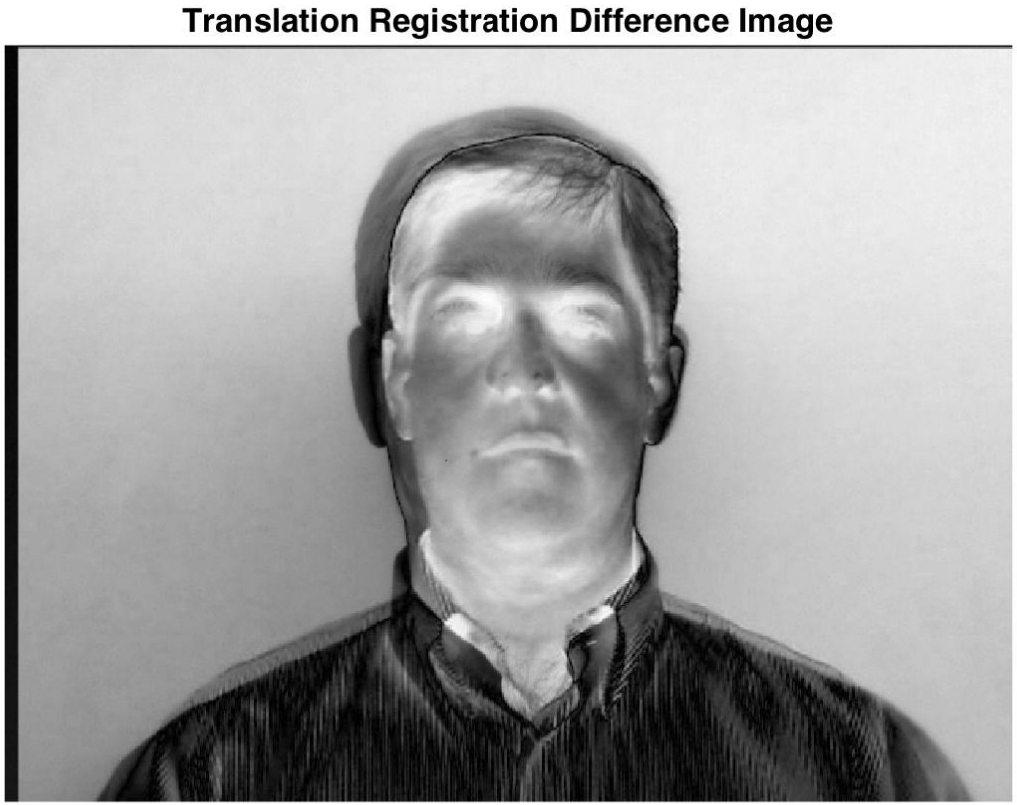
Default registration using **imregister** with transformation type **translation**.

The colours **magenta** represents the fact that the top image is brighter while **green** represents the fact that the lower image is brighter (the intensities of the 2 images are very distinct and are colored to indicate this). Gray regions in the composite image show where the two images have roughly the same intensities. These colours **may** indicate misalignment.

- We can look at the difference image using **imshowpair**. We use:

```
figure
%% Show the difference image
% 'diff' creates a difference image from registered to Fixed
% It seems to invoke the option 'falsecolor' where green
% and magenta are assigned to distinct visible intensity
% areas of the 2 images
imshowpair(registered,Fixed,'diff');
title('\fontsize{16}Translation Registration Difference Image');
print('Difference_Image_from_Translation_Reg.jpg','-djpeg');
```

which produces the following figure:

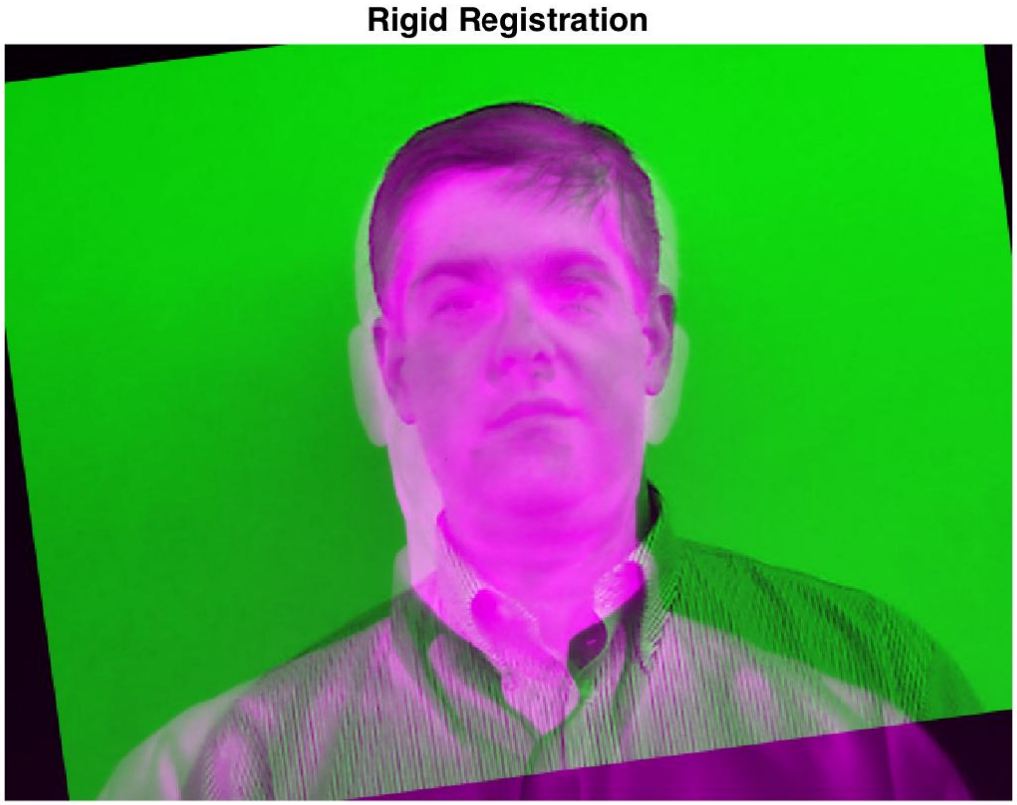


Difference image between the registered and fixed images. The outline of the man in the 2 images is somewhat misaligned. There are gaps between the images around the head and the shoulders indicate problems with not including scale and rotation in the image registration transformation.

- We can change the transformation type from **translation** to **rigid** to try to get a better result. Now the image registration allows for rotation as well as translation. The following MatLab code shows how this might be done:

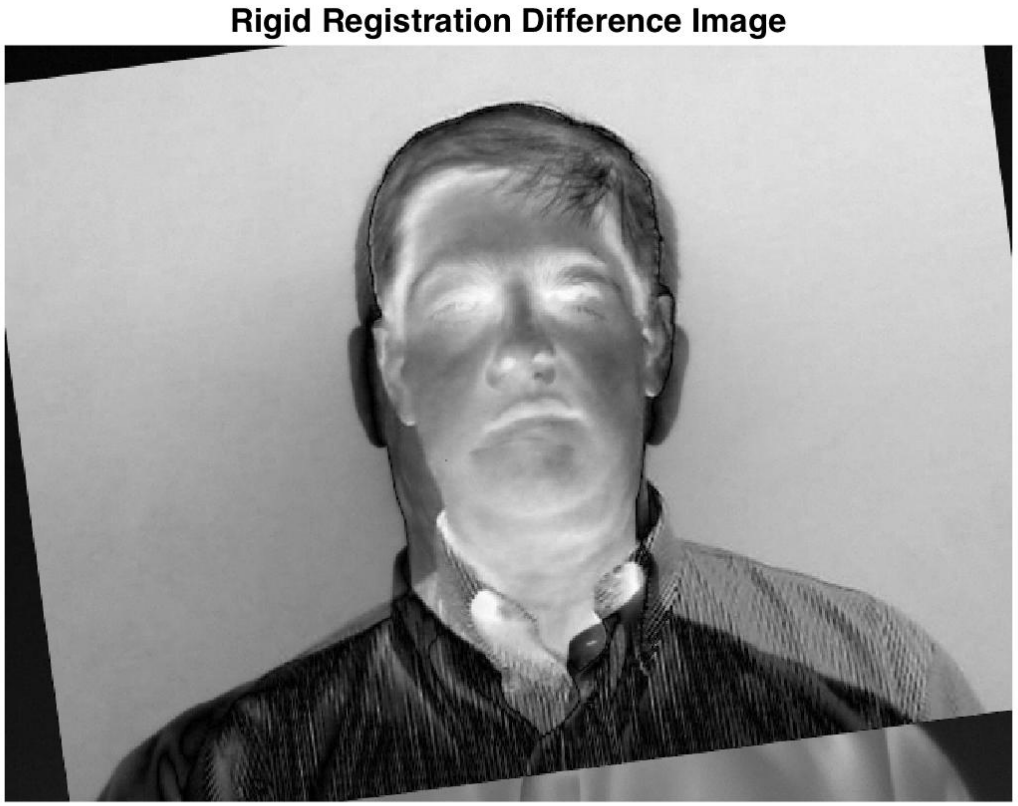
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% RIGID REGISTRATION: both translation and rotation
%% transformations are used in the registration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Change transformType in imregister to 'rigid'
registered=imregister(Moving,Fixed,'rigid',optimizer,metric);

figure
imshowpair(registered,Fixed);
title('\fontsize{16}Rigid Registration');
print('Fever_Detection_Rigid_Reg.jpg','-djpeg');

figure
%% Show the difference image
imshowpair(registered,Fixed,'diff');
title('\fontsize{16}Rigid Registration Difference Image');
print('Difference_Image_from_Rigid_Reg.jpg','-djpeg');
```

- The figure below shows the rigid registration result. Obviously, there is too much rotation.



Registration using **imregister** with transformation type **rigid**.

- The figure below shows the rigid registered difference image:

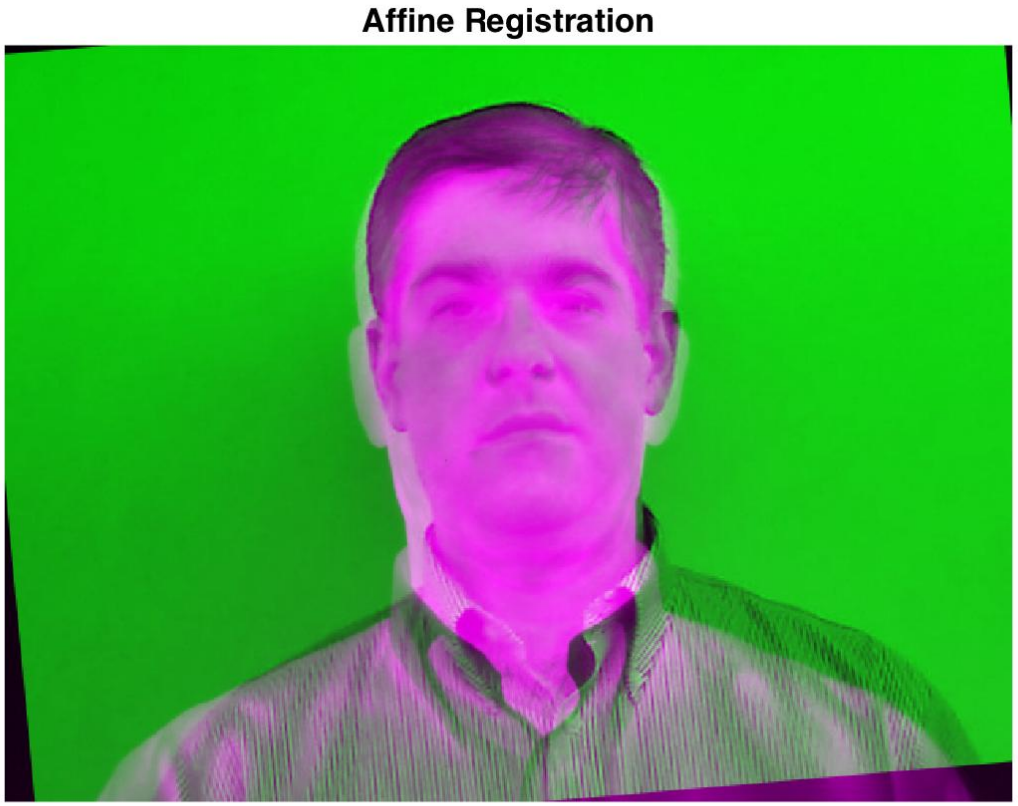


Difference image between the registered and fixed images for a rigid transformation. The outline of the man in the 2 images is somewhat misaligned. There are gaps between the images around the head and the shoulders indicate problems with not including scale and rotation in the image registration transformation.

The following MatLab code shows how an affine transformation can be used in the registration calculation is done:

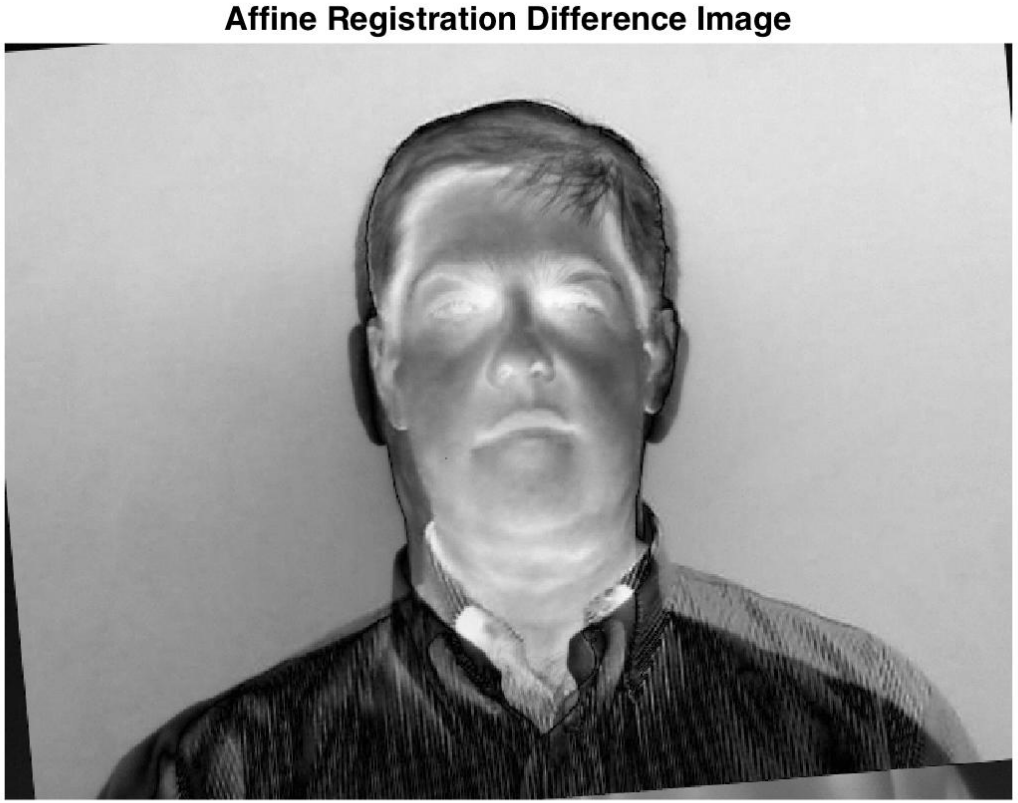
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% AFFINE REGISTRATION: An affine transformation consisting of
%% translation, rotation, scale, and shear are used in the
%% registration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Change transformType in imregister to 'affine'
registered=imregister(Moving,Fixed,'affine',optimizer,metric);

figure
imshowpair(registered,Fixed);
title('\fontsize{16}Affine Registration');
print('Fever_Detection_Affine_Reg.jpg','-djpeg');

figure
%% Show the difference image
imshowpair(registered,Fixed,'diff');
title('\fontsize{16}Affine Registration Difference Image');
print('Difference_Image_from_Affine_Reg.jpg','-djpeg');
```

- The figure below shows the affine registration result:



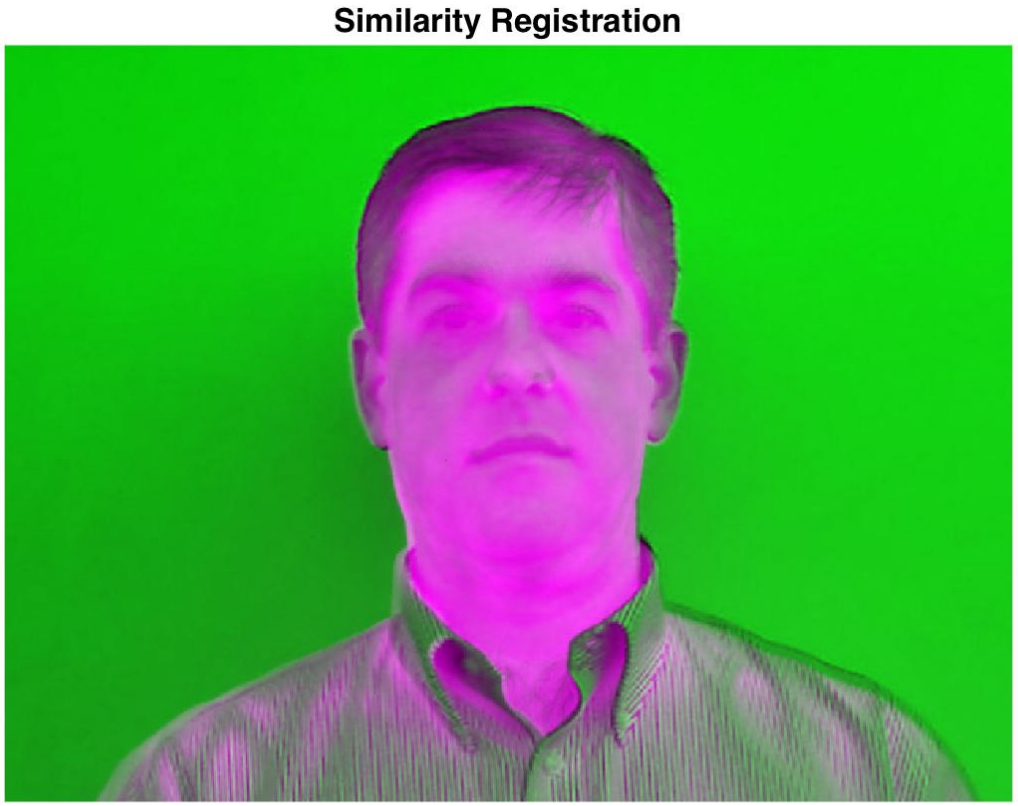
Default registration using **imregister** with transformation type **affine**.

The figure below shows the difference image (not much difference from the translation and rigid registrations):



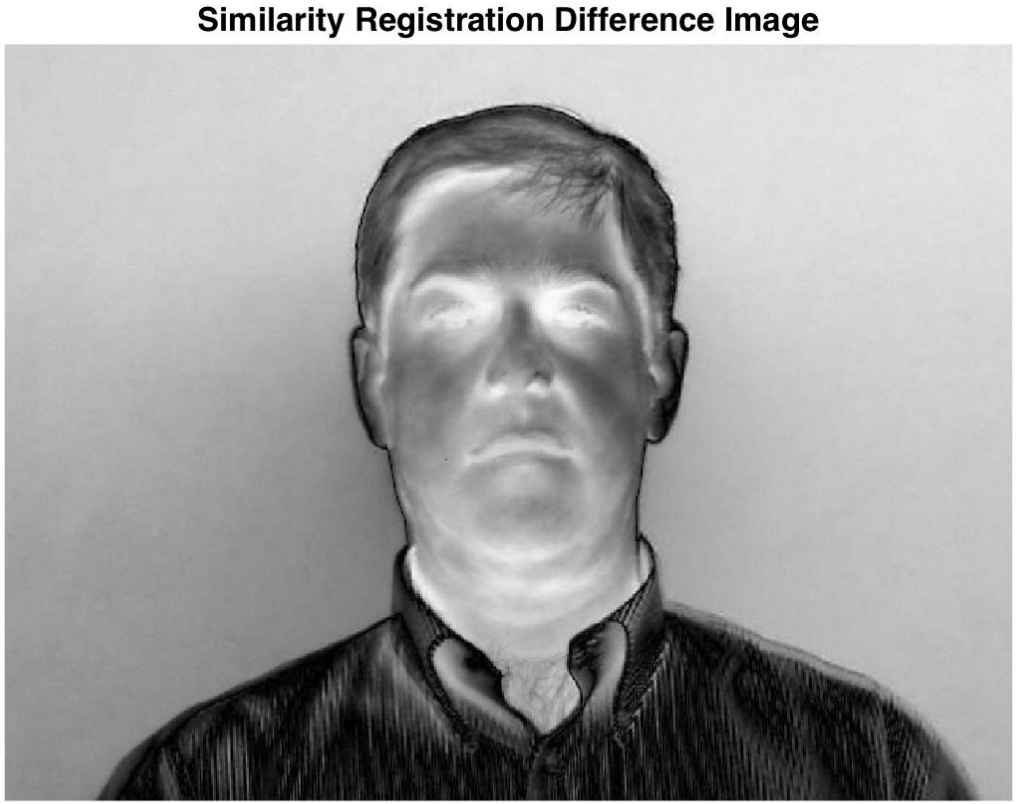
Difference image between the registered and fixed images for an affine transformation used in the image registration. The outline of the man in the 2 images is somewhat misaligned. There are gaps between the images around the head and the shoulders indicate problems are not resolved using rotation, scale and shear in the image registration transformation.

- The figure below shows the similarity registration result:



Default registration using **imregister** with transformation type **similarity**.

The figure below shows the similarity difference:



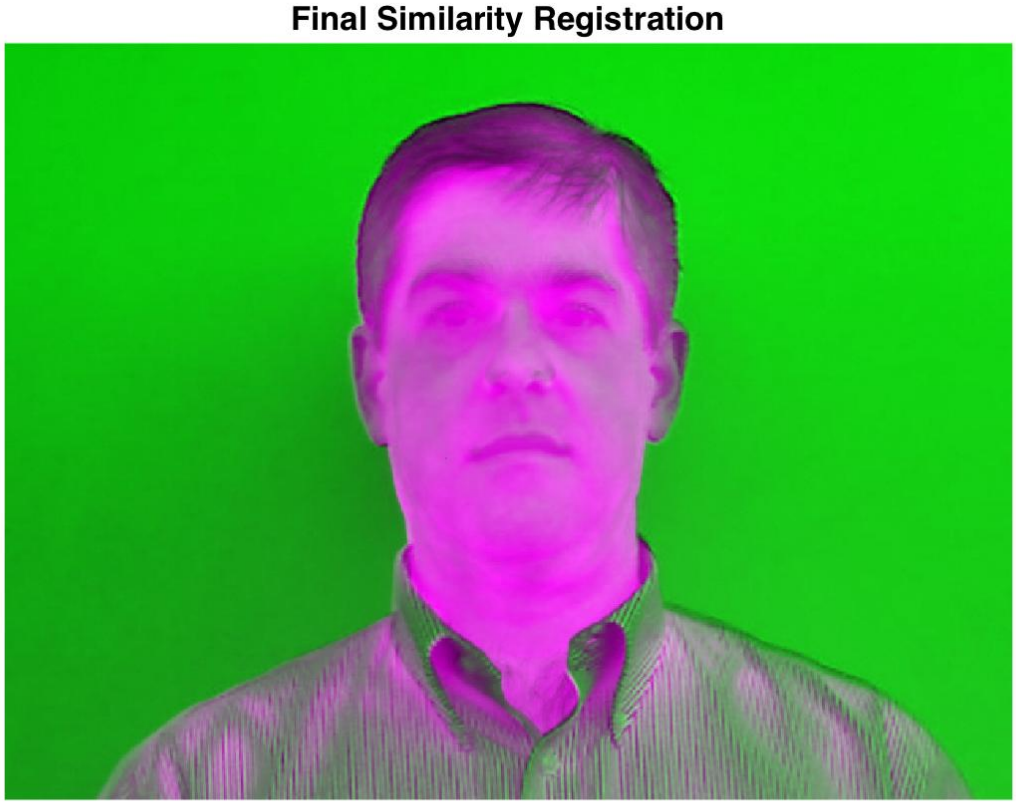
Difference image between the registered and fixed images for a similarity transformation used in the image registration. The outline of the man in the 2 images is somewhat misaligned. There are gaps between the images around the head and the shoulders indicate problems are not resolved using rotation, scale and shear in the image registration transformation.

- To obtain the final best results (according to MathWorks), we use the **similarity** transformation and modify the **optimizer** and **metric** values (probably determined by educated trial and error guesses by MathWorks). The following MatLab code is used:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Modification of the optimize/metric values
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
optimizer.GrowthFactor=1.16974;
optimizer.Epsilon=0.75401;
metric.NumberOfSpatialSamples=500;
metric.NumberOfHistogramBins=90;
optimizer.InitialRadius=0.00675;
optimizer.MaximumIterations=100;

% Compute the final registration to be used in the
% eyes detection and temperature assignment
figure
registered=imregister(Moving,Fixed,'similarity',optimizer,metric);
imshowpair(registered,Fixed);
title('\fontsize{16}Final Similarity Registration');
print('Fever_Detection_Final_Similarity_Reg.jpg','-djpeg');

figure
```

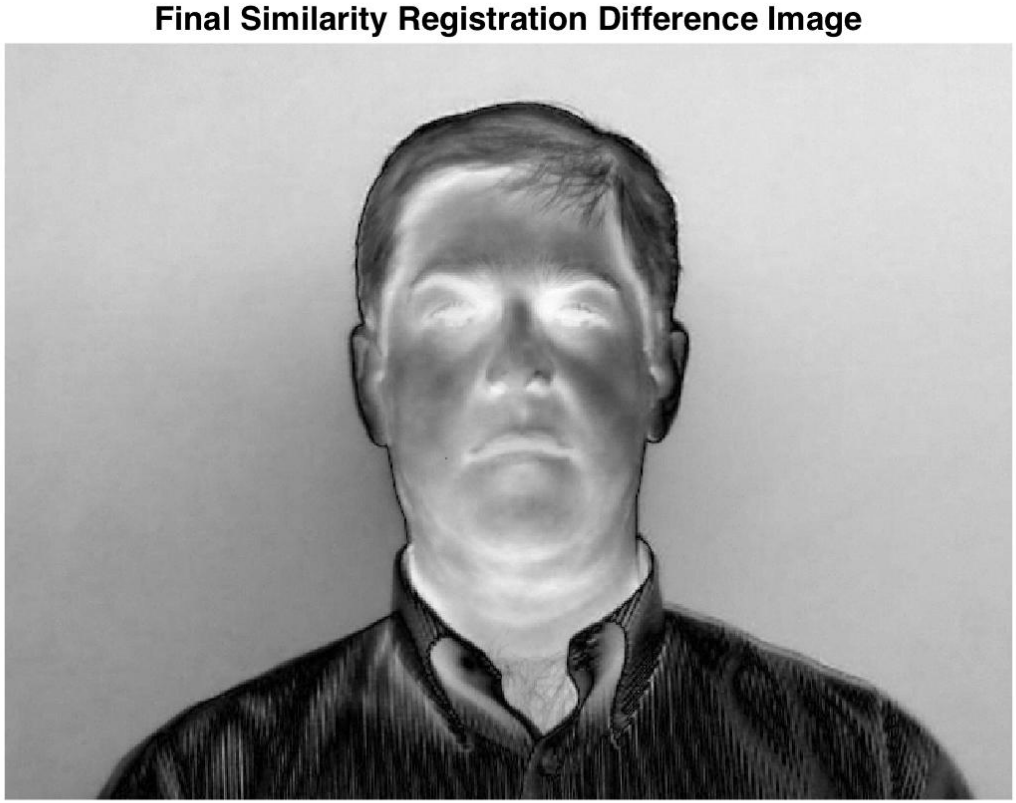
```
%% Show the difference image
imshowpair(registered,Fixed,'diff');
title('\fontsize{16}Final Similarity Registration Difference Image');
print('Difference_Image_from_Final_Similarity_Reg.jpg','-djpeg');
```

- The figure below shows the final similarity registration result:



Default registration using **imregister** with transformation type **similarity** for the MathWorks optimizer and metric parameters.

- The figure below shows the final difference image for the registration of the moving and fixed images:



Final and best registration using the best **similarity** transformation for the MathWorks optimizer and metric parameters.

- Next we have to detect the eyes in this registration result and read off the temperature of the eyes. To detect the eyes, we use the **cascade object detector** in the Computer Vision System Toolbox. This detector uses the Viola-Jones algorithm, which detects eyes using Harr-like features. A Harr-like feature considers adjacent rectangular regions at specific locations in a detection window, sums the pixel intensities and computes the differences of these sums.

- A common observation is that for faces, the region of the eyes is darker than the region around the cheeks. The position of these rectangles is defined relative to a detection window that acts like a bounding box of the target object. A red bounding box is drawn near the eyes to highlight the region of interest on the registered image. The following MatLab

code does this:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Detect the eyes in the RGB Image using the final similarity
%% registered image
%% Use the Viola-Jones Algorithm in the Computer Vision Toolbox
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
eyesDet=vision.CascadeObjectDetector('EyePairSmall');
% Compute the bounding box for the eyes
bbox=step(eyesDet,Moving);
drawBox=vision.ShapeInserter('BorderColor','Black');
image=step(drawBox,registered,int32(bbox));
hold on;
rectangle('Position',bbox,'EdgeColor',[1 0 0]);
% scale the bounding box coordinates to
% make it a bit bigger for display purposes
subsIR=int32(bbox(:,1:2)+bbox(:,3:4)/2);
```

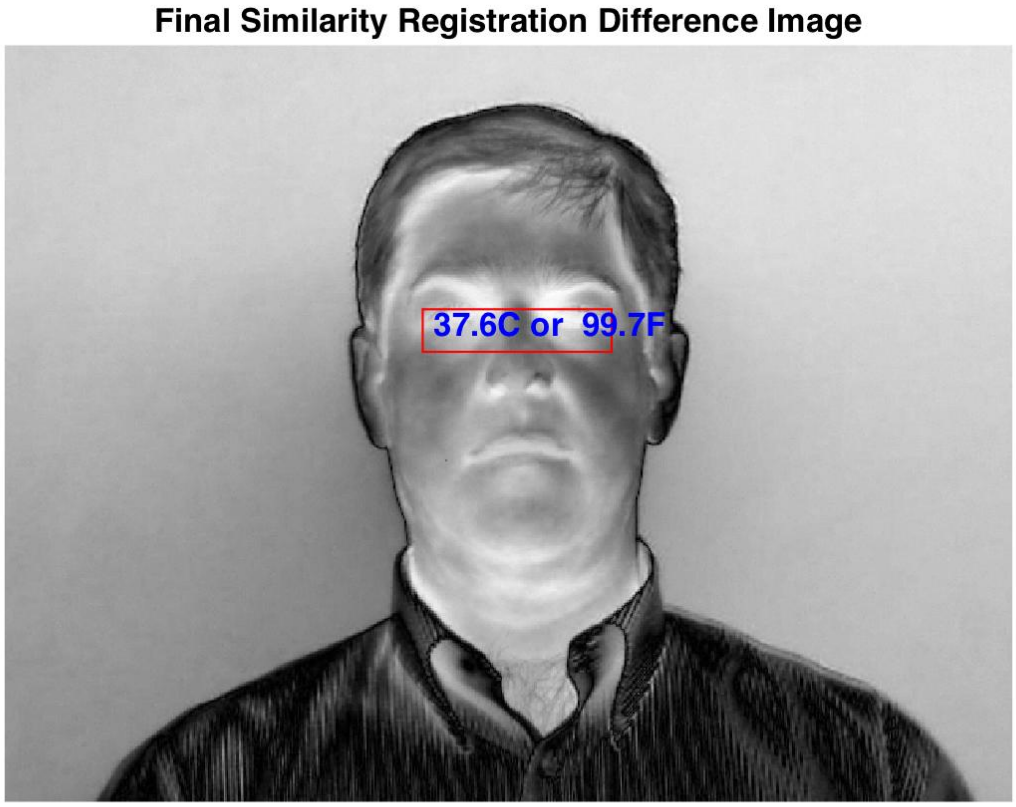
- Because the images are already registered, the bounding box in the Web-cam image can be used to sample temperature values near the eyes in the IR image. The mean is computed from the area enclosed by the bounding box.

- The temperature has to be converted 100millikelvin to millikelvin (divide by 10). Room temperature was $295.15°\text{K}$ (Absolute zero in Kelvini is $0°\text{K}$ or $-273.15°\text{C}$ so $295.15°\text{K}$ is $20°\text{C}$). The final temperature is in Celsius (but was also computed to Fahrenheit by multiplying by 9/5 and adding 32). The MatLab code below does the computation:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Compute temperature near the eyes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compute the mean of the pixels in Fixed in the bounding box
value=mean2(imcrop(Fixed,bbox));
CelsiusEyesTemperature=value/10-273.15+2;
% Convert degrees Celsius to fahrenheit
FahrenheitEyesTemperature=(CelsiusEyesTemperature*9/5)+32;

%% Print the temperature on IR image being displayed
ti=vision.TextInserter('Color',[255 0 0]);
% scale the bounding box coordinates to
% make it a bit bigger for display purposes
ti.Location = int32(bbox(:,1:2)+bbox(:,3:4)/2)*2.5;
ti.Text=sprintf('%5.1fC or %5.1fF',...
                CelsiusEyesTemperature,...
```

```
                    FahrenheitEyesTemperature);
% Adjust the contrast of the Fixed image
contAdj=vision.ContrastAdjuster('CustomProductInputDataType',...
                                  numerictype([],32,8));
imageContrastAdjusted=step(contAdj,Fixed);
textAdded = step(ti,imageContrastAdjusted);
text(267,180,ti.Text,'Color',[0 0 1],'fontsize',16',...
                     'fontweight','bold','linewidth',2);
print('Final_Similarity_Temperature.jpg','-djpeg');
```

The figure below shows the eyes' temperature display within a bounding box with values of 37.6°C or 99.7°F (not quite normal body temperature).

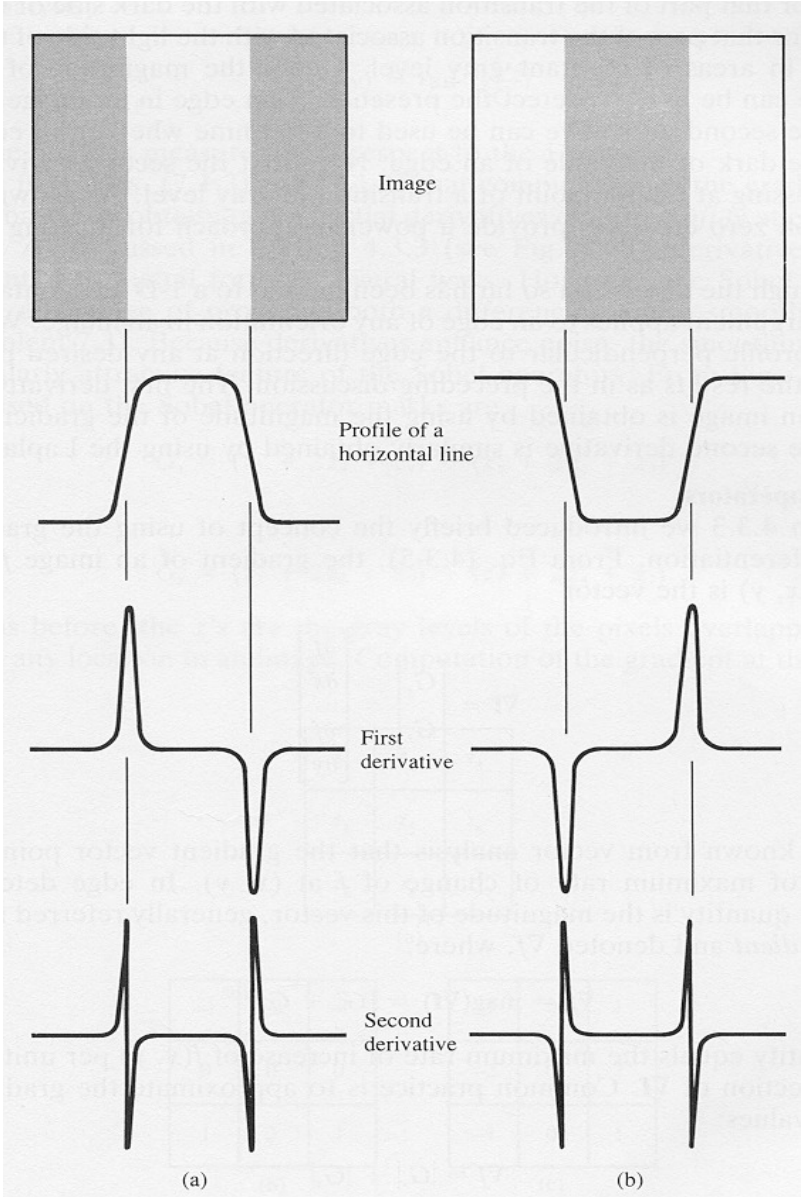

Final Similarity Registration Difference Image
37.6C or 99.7F

- Normal body temperature is an oral temperature of $37.0°$C or $98.6°$F. This is actually an average of normal body temperatures. Your temperature may actually be $1.0°$F ($0.6°$C) or more above or below $37.0°$C or $98.6°$F. Also, your normal body temperature changes by as much as $1.0°$F ($0.6°$C) throughout the day, depending on how active you are and the time of day. So the temperature found by the above MatLab program is within the normal temperature range for a healthy human.

Edge Detection

- An **edge** is the boundary between two regions with relatively distinct grayvalues. A pixel that belongs to an edge is often referred to as a **edgel**. The assumption is that the regions are sufficiently homogeneous that the transition from one region to the other can be determined by grayvalue discontinuity only.

- Figure a shows an image of a light stripe on a dark background while Figure b shows an image of a dark stripe on a light background.



Edge detection be derivatives: (a) light stripe on dark background and (b) dark stripe on light background.

- Below the images in the Figure are their grayvalue profiles. Below the profiles are the 1^{st} order derivative responses and below that are the 2^{nd} order responses.

- Note that the 1^{st} and 2^{nd} order responses are generally smooth rather than abrupt as images are generally blurred as a result of sampling in their acquisition.

- Note that the maximum and minimum values of the two profiles in Figures a and b are reversed. Dark to light transitions have large positive values while light to dark transitions have large negative values. In general, the absolute value of large values indicate the location of edges. We would need to use a threshold, T, to determine "edgeness" of a pixel. The 1^{st} order derivatives of an image are usually represented by the mag-

nitudes of the local image gradient.

- Note that there are large negative and positive values for the 2^{nd} order derivatives for the two images. In general, zero-crossings in 2^{nd} derivative values indicate the location of edges. No threshold is needed to determine "edgeness". The 2^{nd} order derivatives of an image are obtained by the Laplacian.

- Naturally 2^{nd} order derivatives are more prone to noise effects than 1^{st} orders derivatives.

Gradient Operators

- We have already seen the gradient of an image $f(x, y)$ at pixel (x, y) is defined as:

$$\nabla f(x, y) = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{bmatrix}.$$

  The gradient vector points in the direction of maximum rate of change in $f(x, y)$ at (x, y).

- The gradient magnitude is given as:

$$\text{mag}(\nabla f(x, y)) = \left[G_x^2 + G_y^2\right]^{1/2}.$$

  An approximation to this quantity that is often used is:

$$\text{mag}(\nabla f(x, y)) = |G_x| + |G_y|,$$

which is simpler/cheaper to implement.

- The direction of the gradient vector is also important:

$$\alpha(x, y) = \tan^{-1}\left(\frac{G_y}{G_x}\right),$$

where α is measured with respect to the positive x axis.

- **Sobel** operators can be used to obtain estimates of G_x and G_y. Consider the following 3×3 neighbourhood of grayvalues and the 3×3 Sobel operators:

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

Image

-1	-2	-1
0	0	0
1	2	1

G_x Mask

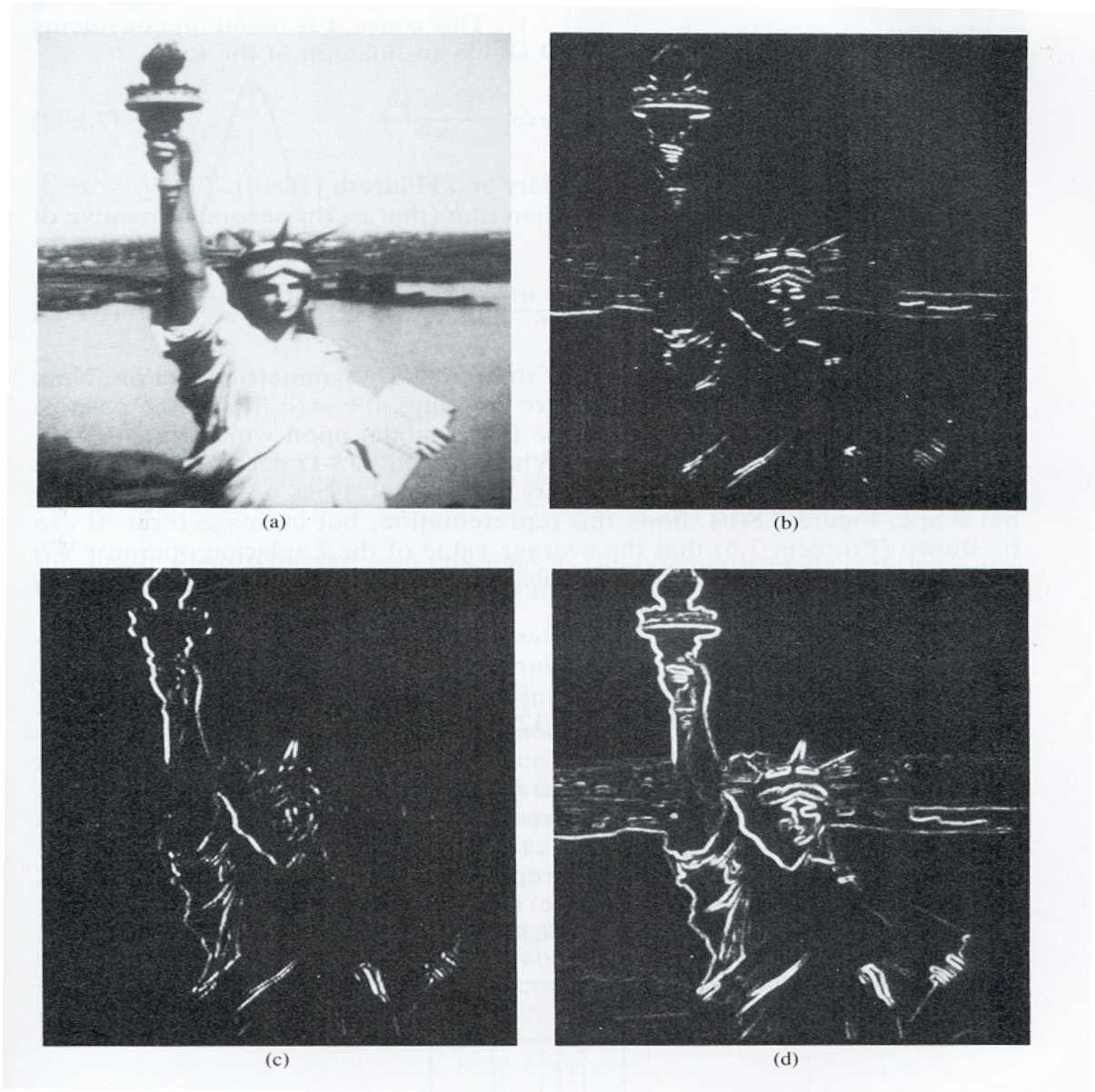
-1	0	1
-2	0	2
-1	0	1

G_y Mask

- Application of these masks to a 3×3 image region (shown above) yields:

$$\begin{aligned} G_x &= (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3) \quad \text{and} \\ G_y &= (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7). \end{aligned}$$

- Figure a below shows an original image of the statue of liberty in New York. Figure b shows the G_x image computed using the appropriate Sobel operator. Figure c shows the G_y image computed using the appropriate Sobel operator. Finally Figure d shows the gradient magnitude image, computed as $\text{mag}(\nabla f(x, y)) = |G_x| + |G_y|$. Note the lack of response to vertical edges in Figure b and the lack of responses to horizontal edges in Figure c. Figure d is just the combination of the G_x and G_y images and exhibits strong horizontal and vertical edges.



(a) An original New York Statue of Liberty image, (b) its $|G_x|$ image, (c) its $|G_y|$ image and (d) the $|G_x| + |G_y|$ image.

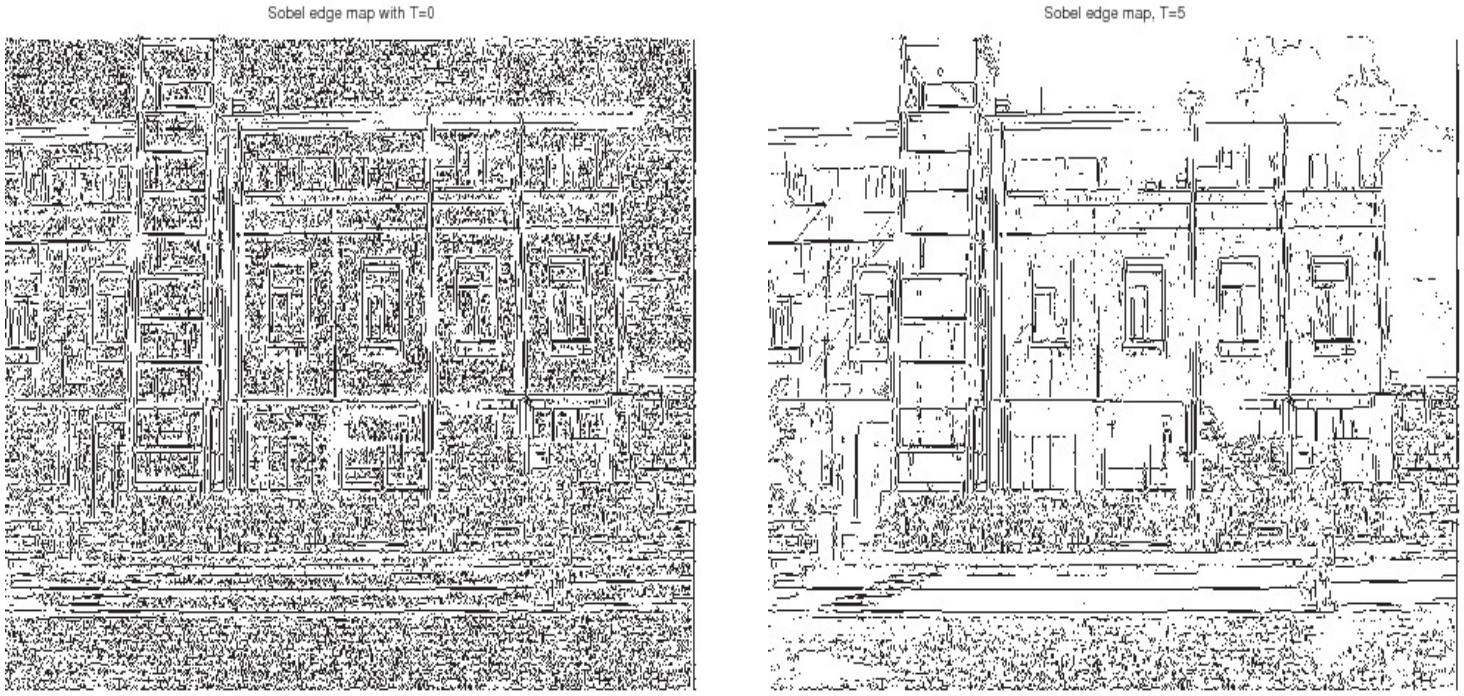
- In Matlab, we can compute Sobel edge maps using the `edge` function for the grayvalue version of the Tamaki Computer Science Building at the University of Auckland, New Zealand shown in the Figure below:

The grayvalue version of the coloured Computer Science building on the Tamaki campus at the University of Auckland.

- The MatLab code for two thresholds:

```
f=double(imread('gray_tamaki_building.ras'));
imshow(f,[]);
title('Original Grayvalue Tamaki Building');

% dir can be 'horizontal', 'vertical' or 'both'
dir='both';
T=0.0;
[g1,t1]=edge(f,'sobel',T,dir);
% Make edges black with white background
g1=~g1;
figure
imshow(g1,[]);
title(['Sobel edge map with T=',num2str(t1)]);
print edge_sobel_tamaki1.jpg -g
T=5.0;
[g2,t2]=edge(f,'sobel',T,dir);
g2=~g2;
figure
imshow(g2,[]);
title(['Sobel edge map, T=',num2str(t2)]);
print edge_sobel_tamaki2.jpg -g
```

- The two edge maps that result are shown in Figure below:



(a) (b)

Sobel edge maps for gradient threshold (a) $T = 0.0$ and (b) $T = 5.0$.

- Two comments are appropriate here:

  1. Edges are sensitive to the value of the threshold on the gradient values. For $T = 0.0$ the edges are very noisy. If the T value is too high, then few if any edges will be found.

  2. What is a "correct" edge. Here edges are defined as intensity discontinuities We have not defined "perceptual edges" but we hypothesize that intensity edges are highly correlated with perceptual edges. We can perform qualitative (subjective) analysis of edge maps but we cannot perform quantitative (objective) analysis of edge maps. Thus, it is actually sometimes hard to compare edge maps from different algorithms.

The Laplacian

- The Laplacian of a 2D function, $f(x, y)$, is defined as:

$$\nabla^2 f(x, y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}.$$

- The most frequently used mask to compute the Laplacian is:

0	-1	0
-1	4	-1
0	-1	0

which yields $\nabla^2 f(x, y) = 4z_5 - (z_2 + z_4 + z_6 + z_8)$ for our 3×3 grayvalue neighbourhood. The sum of the coefficients of any Laplacian operator must always be zero, hence the response of such a mask for a uniform image neighbourhood is 0.

- The Laplacian responds to transitions in intensity. However, because

  1. A 2^{nd} order Laplacian is very sensitive to noise,
  2. often produces double edges (see the Figure below and
  3. is unable to detect edge direction,

  it is seldom used to detect edgels (**ed**ge **el**ment**s**) directly.

- A much more useful application of the Laplacian is to find the **location** of edges using its zero-crossing property.

- MatLab provides two functions. One zero crossings of a a Laplacian of a Gaussian calculation, `log` edges, and the second does a zero crossing detection calculation, `zerocross`, using the fence filter specified by `H`.

- The Laplacian of a Gaussian zero crossing edges can be computing using:

```
sigma=2.0;
T=0.0;
[g1,t1]=edge(f,'log',T,sigma);
figure
imshow(g1,[]);
g1=~g1;
title(['Laplacian of Gaussian edge map with t=',num2str(t1)]);
print edge_log_tamaki1.jpg -g
T=1.0;
% Let log function select threshold
[g2,t2]=edge(f,'log',T,sigma);
g2=~g2;
figure
imshow(g2,[]);
title(['Laplacian of Gaussian edge map with t=',num2str(t2)]);
print edge_log_tamaki2.jpg -g
```

- We can compute the zero crossing using matrices `H1` and `H2` as the 4 and

8 point Laplacian operators:

$$H1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{and} \quad H2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

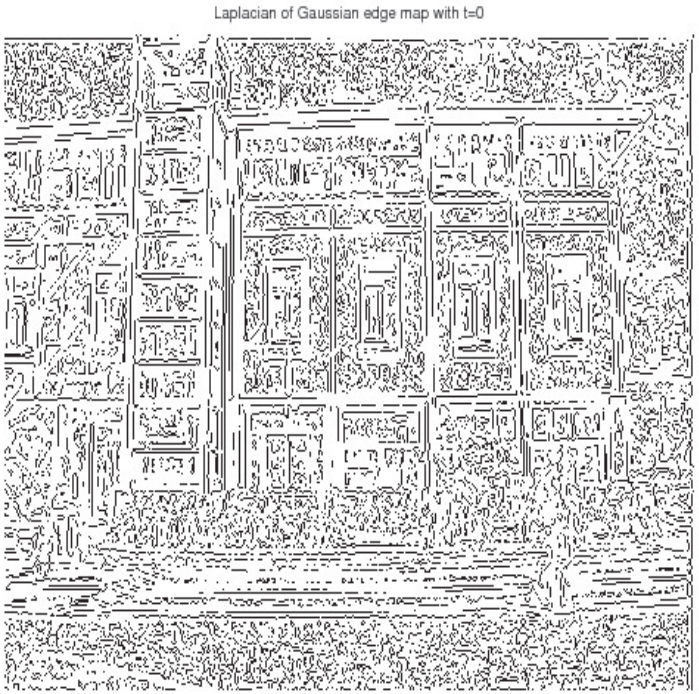
- The zero crossing edges can be computing using:

```
H1=[0 1 0; 1 -4 1; 0 1 0];
H2=[1 1 1; 1 -8 1; 1 1 1];
sigma=2.0;

% H1 is 4 point Laplacian approximation
T=0.0;
[g1,t1]=edge(f,'zerocross',T,H1);
g1=~g1;
figure
imshow(g1,[]);
title(['Zero Crossing edge map using H1, t=',num2str(t1)]);
print edge_zerocross_H1_tamaki1.jpg -g
% Let zero crossing function select threshold
[g2,t2]=edge(f,'zerocross',[],H1);
g2=~g2;
```

```
figure
imshow(g2,[]);
title(['Zero Crossing edge map using H1, t=',num2str(t2)]);
print edge_zerocross_H1_tamaki2.jpg -g

% H2 is 8 point Laplacian approximation
T=0.0;
[g1,t1]=edge(f,'zerocross',T,H2);
g1=~g1;
figure
imshow(g1,[]);
title(['Zero Crossing edge map using H2, t=',num2str(t1)]);
print edge_zerocross_H1_tamaki1.jpg -g
% Let zero crossing function select threshold
[g2,t2]=edge(f,'zerocross',[],H2);
g2=~g2;
figure
imshow(g2,[]);
title(['Zero Crossing edge map using H2, t=',num2str(t2)]);
print edge_zerocross_H1_tamaki2.jpg -g
```

- The figures below show the edge maps using threshold $T = 0$ and auto-

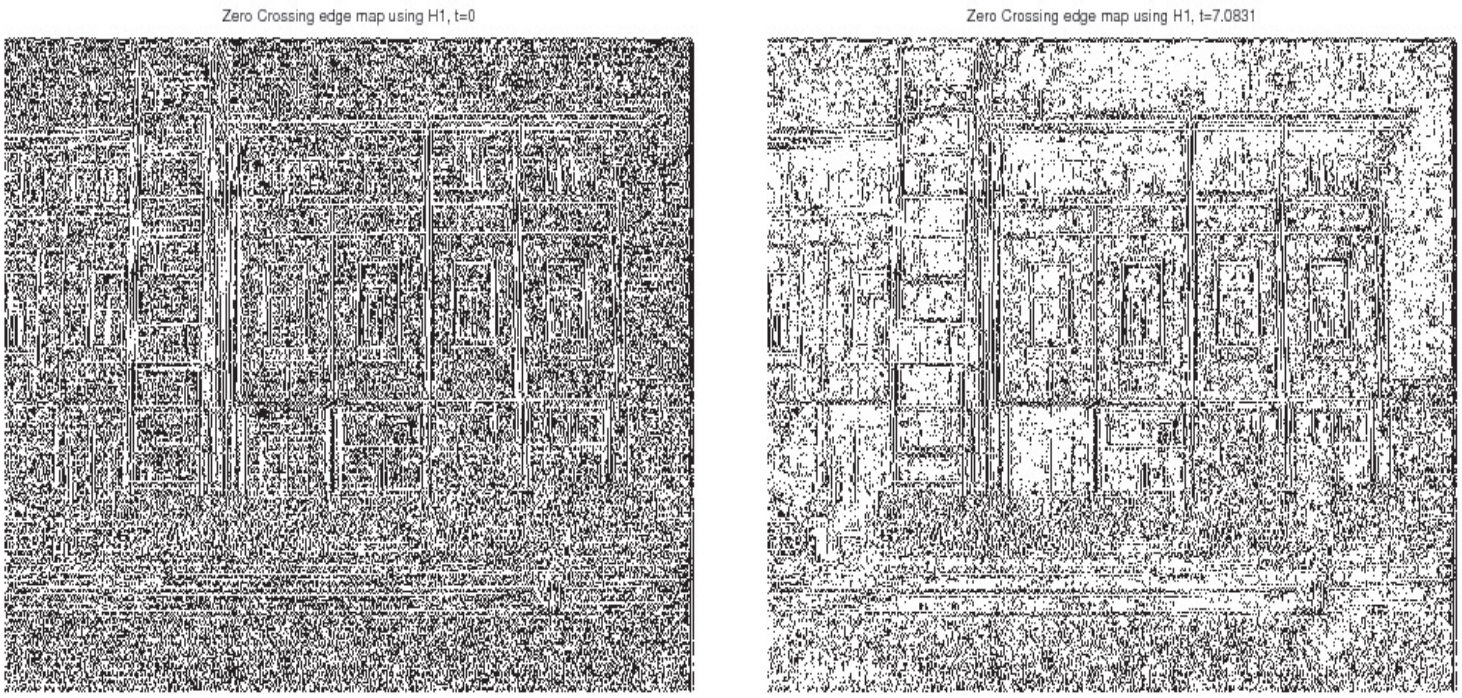
matic threshold determination. Notice that the $T = 0.0$ edges maps are very noisy for the images in the three (a) figures below.



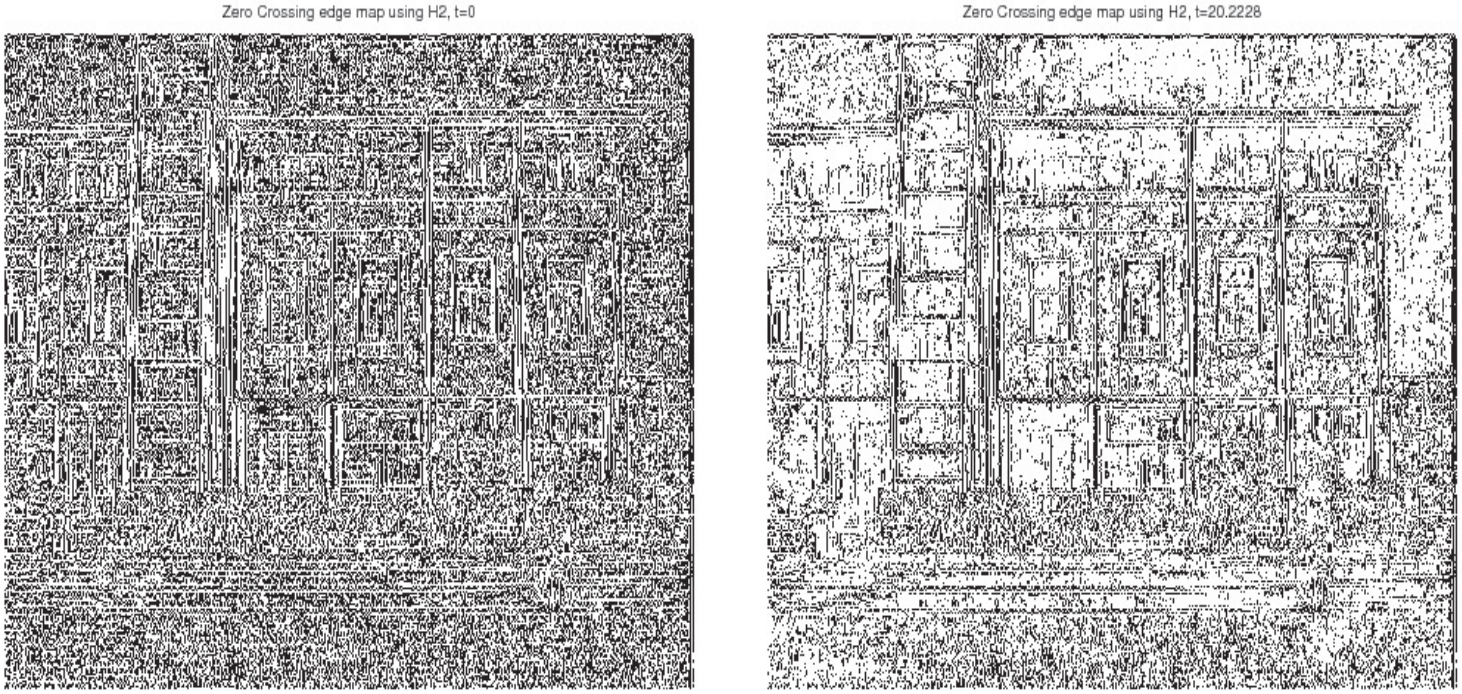
(a)

(b)

The Laplacian of a Gaussian zero crossing edge maps for (a) $T = 0.0$ and (b) T automatically determined.



(a) and (b): The edge maps for threshold $T = 0.0$ and for T automatically determined for $H1$.



(a) (b)

(a) and (d): the edge maps for threshold for $T = 0.0$ and for T automatically determined for $H2$.

Canny Edges

- Perhaps one of the best edge algorithms around was proposed by John Canny in 1986. It is certainly the best operator in MatLab. [John Canny, "A Computational Approach to Edge Detection", PAMI(8):6, 1986, 18,693 citations as of March 31st, 2014!!!]

- The steps of the algorithm can be summarized as:

    1. Smooth the image with a Gaussian with standard deviation σ to reduce noise effects.
    2. The local gradient magnitude $\sqrt{g_x^2 + g_y^2}$ and the edge direction $\tan^{-1}(g_y/g_x)$ are computed for each point. An edgel is defined as an edge point that is locally maximum in the direction of the

gradient.

3. The algorithm tracks edgels along the top of gradient ridges and sets all edges to 0 that do not lie on these ridges so that thin edges result. This is called **non-maximum suppression**. Ridge edgels are then thresholded using **hysteresis thresholding** using two thresholds T_1 and T_2, $T_1 < T_2$. For ridge edgels whose gradients are greater than T_2 are "strong" edges while ridge edges with gradient magnitude between T_1 and T_2 are "weak" edges.

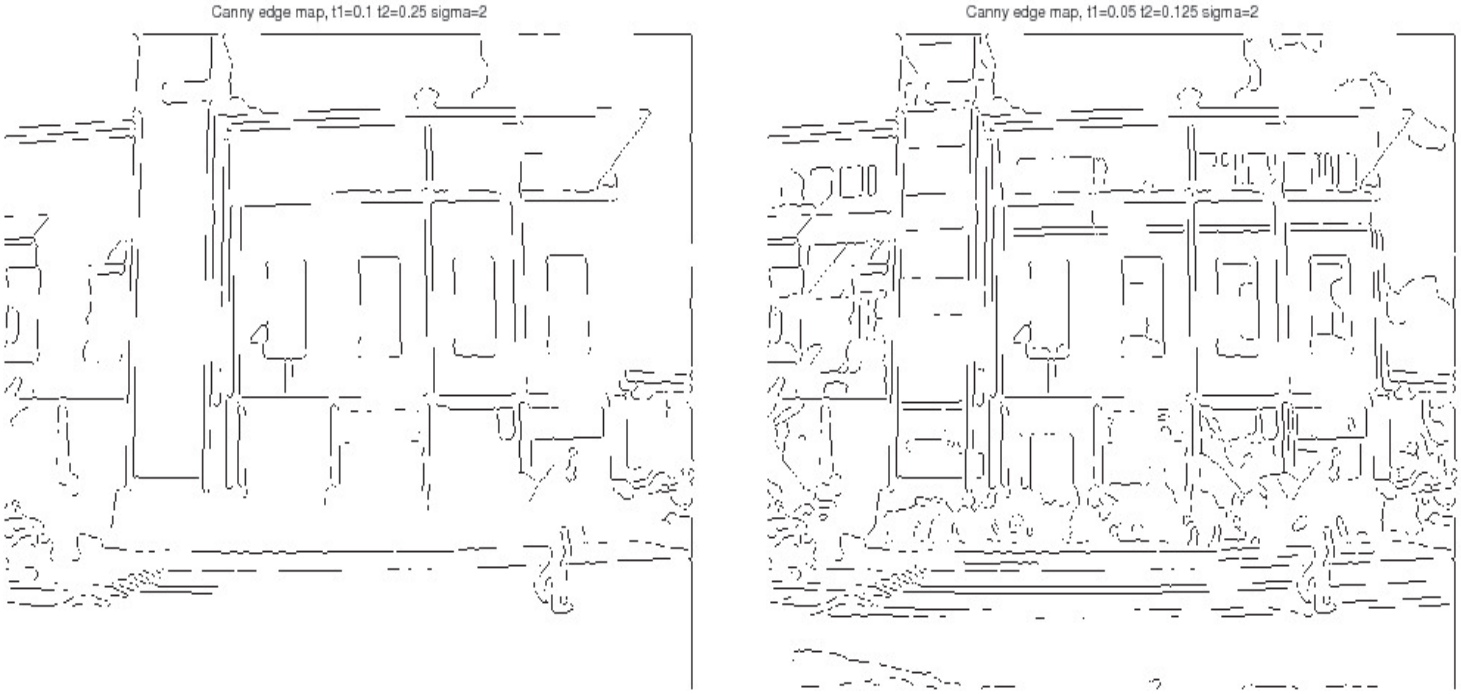
4. Finally edge linking is performed by incorporating weak edgels that are 8-connected to strong edgels.

- We can compute edge maps using the Canny operator as:

```
sigma=2.0;
```

```
[g1,t1]=edge(f,'canny',[0.10 0.25],sigma);
g1=~g1;
figure
imshow(g1,[]);
title(['Canny edge map, t1=',num2str(t1(1,1)),...
       ' t2=',num2str(t1(1,2)),' sigma=',num2str(sigma)]);
print edge_canny_tamaki1.jpg -g
% Let canny function compute thresholds
[g2,t2]=edge(f,'canny',[],sigma);
g2=~g2;
figure
imshow(g2,[]);
title(['Canny edge map, t1=',num2str(t2(1,1)),...
       ' t2=',num2str(t2(1,2)),' sigma=',num2str(sigma)]);
print edge_canny_tamaki2.jpg -g
```

- The figure below shows edge maps with specified values for T_1 and T_2 and automatically determined T_1 and T_2 values
- There are significantly less noise edgels and edges generally are located in strong gradient magnitude areas.



(a) (b)

Canny edge maps (a) for $T_1 = 0.10$ and $T_2 = 0.25$ and (b) with T_1 and T_2 automatically determined.