**Lab 6**

**Speech and Image Processing**

SUBMITTED BY:

**Abdul Ghaffar Kalhoro 194699**

**Hamad Nasir 120312**

**Mohammad Rehman Rabbani 137364**

Group assignment up to 3 students per group.

Using MATLAB or Octave, perform the following tasks

**Part (a)** 2 marks

Take a jpeg image and read it in MATLAB using imread() function and display it using imshow() function now convert it to double using double() and again display it using imshow() (note: for double image the color range is from 0 to 1 and not from 0 to 255)

convert this RGB image array into grayscale array “A” and display it

**Part (b)** 3 marks

Now make a small image B as a square sub-image of A

B = A(101:200, 101:200);

Now B is a 100x100 pixel grayscale image. Here the number 101:200 are arbitrary; they can be changed for selecting an interesting portion of image as per requirement (see the three images at the end of document)

**Part (c)** 5 marks

Using for loops, cross-correlate the image A with image B, and the resultant image is saved as C (do not use MATLAB’s built-in cross correlation function. In this new resultant image C, the location on image A which match the template B will have large values, while other places have low values. In this way we can check at which locations a particular object B occurs in an image A.

C[i,j] = A[i+u, j+v] B[u,v]

Remember to subtract the mean grayscale value from the images, so that the pixel grayscale intensities contain both positive and negative numbers).

**Deliverables:** MATLAB code and images A, B and C (images should be unique for all groups)

**SOURCE CODE**

**Lab6.m**

%task1: input image and display it.

%reading image.

cup = imread('lab6.png');

figure, imshow(cup);

title('original image');

%whos cup

%convert image to double

cup2 = im2double(cup);

%imshow(cup2)

%whos cup2

%convert to gray scale

cup3 = rgb2gray(cup2);

%display image

figure, imshow(cup3);

title('RGB to Gray scale image');

%part(b)

cup4 = cup3(210:309, 210:309);

figure, imshow(cup4);

title('squared image');

whos cup4

%c = normxcorr2(cup4,cup3);

c = normxcorr2\_general(cup4,cup3);

%normxcorrn(cup4,cup3,'full',[]);

%normcor(cup3,cup4);

%figure, surf(c), shading flat

[ypeak, xpeak] = find(c==max(c(:)));

yoffSet = ypeak-size(cup4,1);

xoffSet = xpeak-size(cup4,2);

figure

imshow(cup3);

imrect(gca, [xoffSet+1, yoffSet+1, size(cup4,2), size(cup4,1)]);

**Normxcorr2\_general.m**

function [C,pixelsOverlaping] = normxcorr2\_general(varargin)

[T, matrix\_A, pixelsOverlaping] = ParseInputs(varargin{:});

imgSize\_A = size(matrix\_A);

imgSize\_T = size(T);

% Find the number of pixels used for the calculation as the two images are

% correlated. The size of this image will be the same as the correlation

% image.

pixelsOverlaping = sumation\_local(ones(imgSize\_A),imgSize\_T(1),imgSize\_T(2));

sumOf\_A = sumation\_local(matrix\_A,imgSize\_T(1),imgSize\_T(2));

sumOf\_A2 = sumation\_local(matrix\_A.\*matrix\_A,imgSize\_T(1),imgSize\_T(2));

% Note: diff\_local\_sums should be nonnegative, but it may have negative

% values due to round off errors. Below, we use max to ensure the radicand

% is nonnegative.

sumOf\_A2\_diff = ( sumOf\_A2 - (sumOf\_A.^2)./ pixelsOverlaping );

clear sumOf\_A2;

denom\_A = max(sumOf\_A2\_diff,0);

clear sumOf\_A2\_diff;

% Flip T in both dimensions so that its correlation can be more easily

% handled.

rotate\_T = rot90(T,2);

sumOf\_T = sumation\_local(rotate\_T,imgSize\_A(1),imgSize\_A(2));

sumOf\_T2 = sumation\_local(rotate\_T.\*rotate\_T,imgSize\_A(1),imgSize\_A(2));

clear rotate\_T;

diff\_local\_sums\_T = ( sumOf\_T2 - (sumOf\_T.^2)./ pixelsOverlaping );

clear sumOf\_T2;

denominator\_T = max(diff\_local\_sums\_T,0);

clear diff\_local\_sums\_T;

denom = sqrt(denominator\_T .\* denom\_A);

clear denominator\_T denom\_A;

xcorr\_TA = xcorelation\_fast(T,matrix\_A);

clear matrix\_A T;

numerator = xcorr\_TA - sumOf\_A .\* sumOf\_T ./ pixelsOverlaping;

clear xcorr\_TA sumOf\_A sumOf\_T;

% denom is the sqrt of the product of positive numbers so it must be

% positive or zero. Therefore, the only danger in dividing the numerator

% by the denominator is when dividing by zero. We know denominator\_T~=0 from

% input parsing; so denom is only zero where denom\_A is zero, and in these

% locations, C is also zero.

C = zeros(size(numerator));

total = 1000\*eps( max(abs(denom(:))) );

i\_nonzero = find(denom > total);

C(i\_nonzero) = numerator(i\_nonzero) ./ denom(i\_nonzero);

clear numerator denom;

% Remove the border values since they result from calculations using very

% few pixels and are thus statistically unstable.

% By default, pixelsOverlaping = 0, so C is not modified.

if( pixelsOverlaping > max(pixelsOverlaping(:)) )

error(['ERROR: pixelsOverlaping ' num2str(pixelsOverlaping) ...

' must not be greater than the maximum number of overlap pixels ' ...

num2str(max(pixelsOverlaping(:))) '.']);

end

C(pixelsOverlaping < pixelsOverlaping) = 0;

%-------------------------------

% Function sumation\_local

%

function sumOf\_A = sumation\_local(matrix\_A,rows,columns)

% This algorithm depends on precomputing running sums.

% If rows,columns are equal to the size of matrix\_A, var\_a faster method can be used for

% calculating the local sum. Otherwise, the slower but more general method

% can be used. The faster method is more than twice as fast and is also

% less memory intensive.

if( rows == size(matrix\_A,1) && columns == size(matrix\_A,2) )

cum\_sum = cumsum(matrix\_A,1);

c = [cum\_sum; repmat(cum\_sum(end,:),rows-1,1) - cum\_sum(1:end-1,:)];

cum\_sum = cumsum(c,2);

clear c;

sumOf\_A = [cum\_sum, repmat(cum\_sum(:,end),1,columns-1) - cum\_sum(:,1:end-1)];

else

% Break the padding into parts to save on memory.

B = zeros(size(matrix\_A,1)+2\*rows,size(matrix\_A,2));

B(rows+1:rows+size(matrix\_A,1),:) = matrix\_A;

cum\_sum = cumsum(B,1);

c = cum\_sum(1+rows:end-1,:)-cum\_sum(1:end-rows-1,:);

d = zeros(size(c,1),size(c,2)+2\*columns);

d(:,columns+1:columns+size(c,2)) = c;

cum\_sum = cumsum(d,2);

sumOf\_A = cum\_sum(:,1+columns:end-1)-cum\_sum(:,1:end-columns-1);

end

%-------------------------------

% Function xcorelation\_fast

%

function corelation\_cross = xcorelation\_fast(T,matrix\_A)

T\_size = size(T);

A\_size = size(matrix\_A);

size\_out = A\_size + T\_size - 1;

% Figure out when to use spatial domain vs. freq domain

conv\_time = time\_conv2(T\_size,A\_size); % 1 conv2

fft\_time = 3\*time\_fft2(size\_out); % 2 fft2 + 1 ifft2

if (conv\_time < fft\_time)

corelation\_cross = conv2(rot90(T,2),matrix\_A);

else

corelation\_cross = xcorelation\_freq(T,matrix\_A,size\_out);

end

%-------------------------------

% Function xcorelation\_freq

%

function xcorelation\_absolt = xcorelation\_freq(var\_a,var\_b,size\_out)

% Find the next largest size that is var\_a multiple of var\_a combination of 2, 3,

% and/or 5. This makes the FFT calculation much faster.

size\_optimal(1) = valid\_dimensionClosest(size\_out(1));

size\_optimal(2) = valid\_dimensionClosest(size\_out(2));

% Calculate correlation in frequency domain

Fa = fft2(rot90(var\_a,2),size\_optimal(1),size\_optimal(2));

Fb = fft2(var\_b,size\_optimal(1),size\_optimal(2));

xcorelation\_absolt = real(ifft2(Fa .\* Fb));

xcorelation\_absolt = xcorelation\_absolt(1:size\_out(1),1:size\_out(2));

%-------------------------------

% Function time\_conv2

%

function time = time\_conv2(size\_obs,size\_ref)

K = 2.7e-8;

% convolution time = K\*prod(size\_obs)\*prod(size\_ref)

time = K\*prod(size\_obs)\*prod(size\_ref);

%-------------------------------

% Function time\_fft2

%

function time = time\_fft2(size\_out)

% time var\_a frequency domain convolution by timing two one-dimensional ffts

R = size\_out(1);

S = size\_out(2);

% Tr = time\_fft(R);

% K\_fft = Tr/(R\*log(R));

% K\_fft was empirically calculated by the 2 commented-out lines above.

K\_fft = 3.3e-7;

Tr = K\_fft\*R\*log(R);

if S==R

Ts = Tr;

else

% Ts = time\_fft(S); % uncomment to estimate explicitly

Ts = K\_fft\*S\*log(S);

end

time = S\*Tr + R\*Ts;

%-----------------------------------------------------------------------------

function [T, matrix\_A, pixelsOverlaping] = ParseInputs(varargin)

if( nargin < 2 || nargin > 3 )

error('ERROR: The number of arguments must be either 2 or 3. Please see the documentation for details.');

end

T = varargin{1};

matrix\_A = varargin{2};

if( nargin == 3 )

pixelsOverlaping = varargin{3};

else

pixelsOverlaping = 0;

end

% The following requires the image processing toolbox, so it is commented

% out here for generality.

%iptcheckinput(T,{'logical','numeric'},{'real','nonsparse','2d','finite'},mfilename,'T',1)

%iptcheckinput(matrix\_A,{'logical','numeric'},{'real','nonsparse','2d','finite'},mfilename,'matrix\_A',2)

checkSizesTandA(T,matrix\_A)

% See geck 342320. If either matrix\_A or T has var\_a minimum value which is negative, we

% need to shift the array so all values are positive to ensure numerically

% robust results for the normalized cross-correlation.

matrix\_A = shiftData(matrix\_A);

T = shiftData(T);

checkFlat\_func(T);

%-----------------------------------------------------------------------------

function B = shiftData(matrix\_A)

B = double(matrix\_A);

is\_unsigned = isa(matrix\_A,'uint8') || isa(matrix\_A,'uint16') || isa(matrix\_A,'uint32');

if ~is\_unsigned

min\_B = min(B(:));

if min\_B < 0

B = B - min\_B;

end

end

%-----------------------------------------------------------------------------

function checkSizesTandA(T,matrix\_A)

if numel(T) < 2

eid = sprintf('Images:%cum\_sum:invalidTemplate',mfilename);

msg = 'TEMPLATE must contain at least 2 elements.';

error(eid,'%cum\_sum',msg);

end

%-----------------------------------------------------------------------------

function checkFlat\_func(T)

if std(T(:)) == 0

eid = sprintf('Images:%cum\_sum:sameElementsInTemplate',mfilename);

msg = 'The values of TEMPLATE cannot all be the same.';

error(eid,'%cum\_sum',msg);

end

%-----------------------------------------------------------------------------

function [newNumber] = valid\_dimensionClosest(columns)

% Find the closest valid dimension above the desired dimension. This

% will be var\_a combination of 2s, 3s, and 5s.

% Incrementally add 1 to the size until

% we reach var\_a size that can be properly factored.

newNumber = columns;

result = 0;

newNumber = newNumber - 1;

while( result ~= 1 )

newNumber = newNumber + 1;

result = num\_factorize(newNumber);

end

%-----------------------------------------------------------------------------

function [columns] = num\_factorize(columns)

for ifac = [2 3 5]

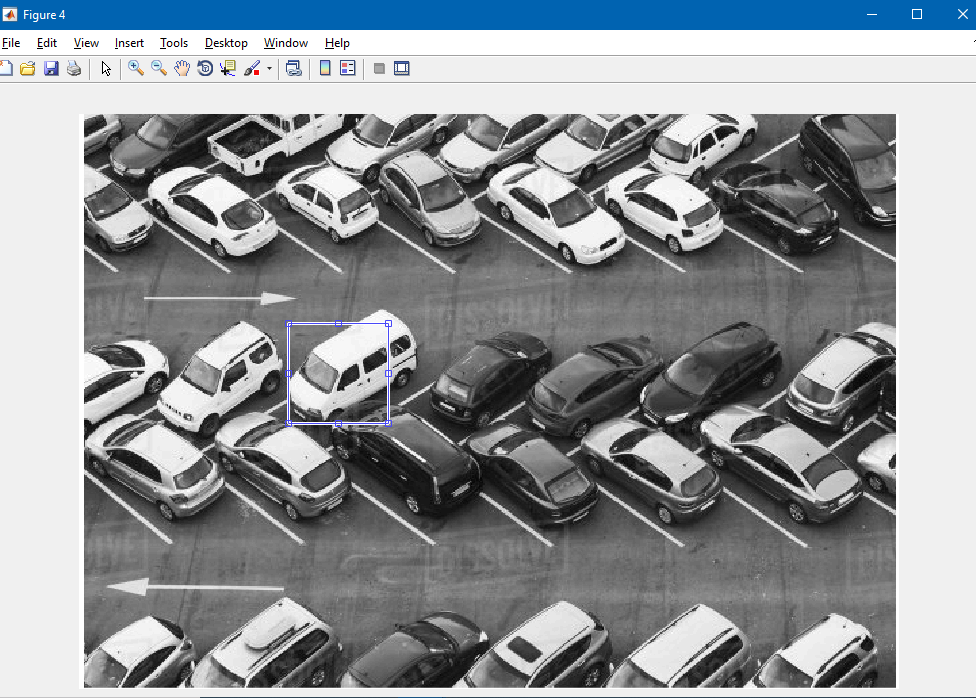
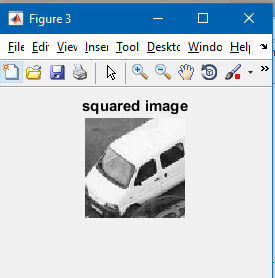
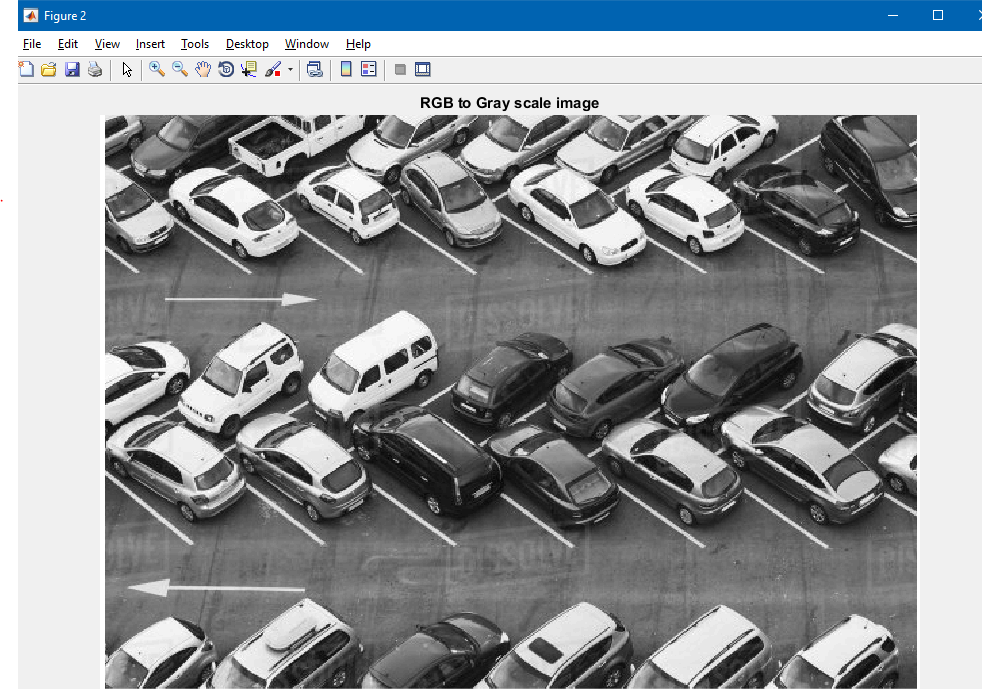
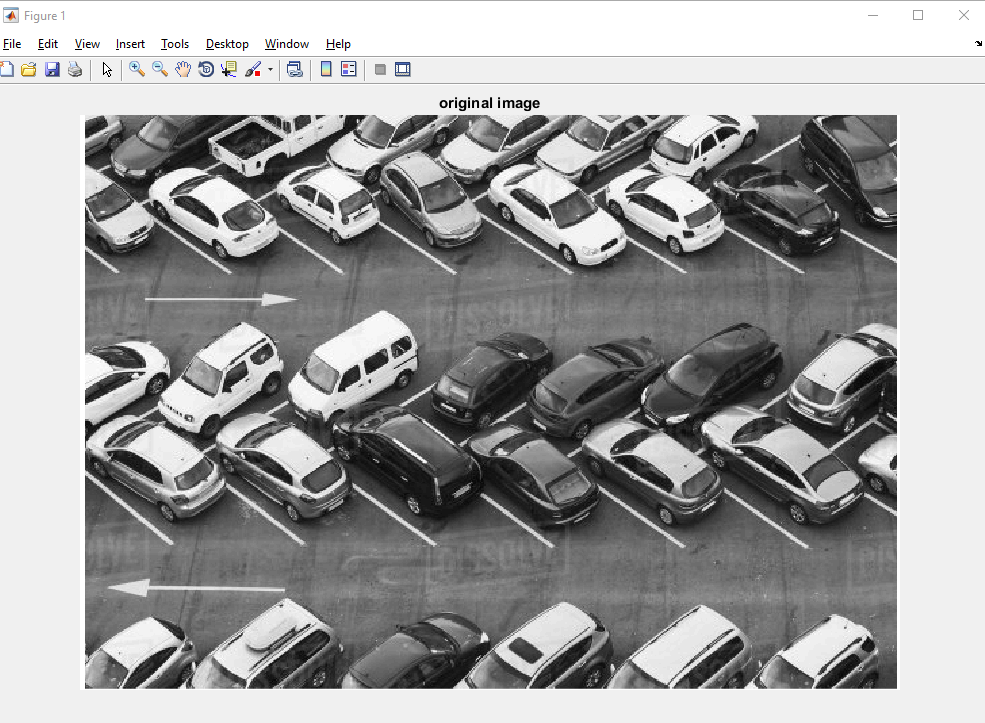
while( rem(columns,ifac) == 0 )

columns = columns/ifac;

end

end

**OUTPUT**

****