Assignment 2

Tanuj Kumar ta.kumar@mail.utoronto.ca 1002197133

October 2017

NOTE: BECAUSE OF LATEX HANDLING IMAGES BADLY, KEEP SCROLLING DOWN IF YOU DO NOT SEE A FIGURE RIGHT BELOW WHERE IT SAYS IT SHOULD BE

1 Question 1

1.1 A

The code for Harris corner detection is as follows:

```
sig = 1;
r = 1;
shape = 2*r+1;
t = 100;
a = 0.04;
wfilt = 6;
gw = fspecial('gaussian', max(1, fix(wfilt*sig)), sig);
cdm = fspecial('disk',r)>0;

% STEP 0: Read image, convert to grayscale

buildbase = imread('building.jpg');
build = rgb2gray(buildbase);

% STEP 1: Get x and y directional gradients of image

[Ix, Iy] = imgradientxy(build, 'prewitt');

% elementwise multiplication for moment
Ix2 = Ix.^2;
Iy2 = Iy.^2;
Ixy = Ix.*Iy;

GIx2 = conv2(Ix2, gw, 'same');
GIy2 = conv2(Iy2, gw, 'same');
GIxy = conv2(Ixy, gw, 'same');
```

```
% determinant and trace

detM = GIx2 .* GIy2 - GIxy.^2;
trM = GIx2 + GIy2 + eps;

% harris measure

R2 = detM - a * trM.^2;
R2=(1000/max(max(R2)))*R2;
R=R2;

% R = detM ./ trM;

% non-maxima suppression

maxes = ordfilt2(R, shape^2, cdm);

% harris points:
Rt = R > t;
hp = (R == maxes) & (R > t);

[r,c] = find(hp);

%imshow(hp);


figure, image(buildbase), hold on,
plot(c, r, 'ro'), title('harris corners');
```

And the resulting plot of the Harris corner metric is as follows:

1.2 B

The inclusion of non-maximum suppression is in the above code, and is specifically this code tidbit:

```
shape = 2*r+1;
t = 100;
a = 0.04;
wfilt = 6;
gw = fspecial('gaussian', max(1, fix(wfilt*sig)), sig);
cdm = fspecial('disk',r)>0;

% ...

maxes = ordfilt2(R, shape^2, cdm);
```

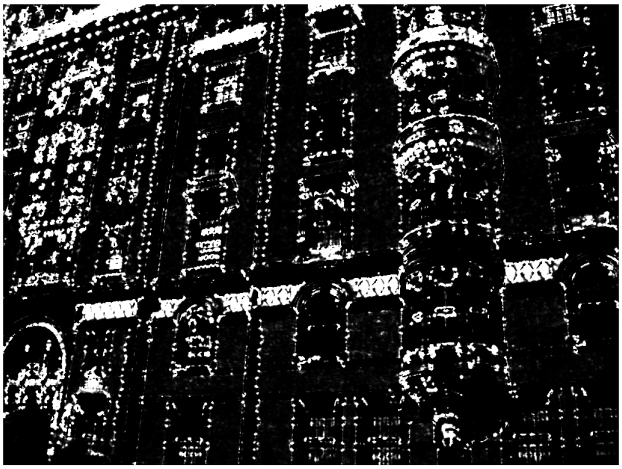

Figure 1: An image showing the raw Harris corner metric data. Note the "ghostly outline" consisting of "chunked maxima points" from the Harris filter.

The plot of the NMS points:

Figure 2: After applying non-maximal suppression, we get the key "maximal" points (a bit hard to see in this image due to sizing and resolution), which cleans up our corner detection and will allow us to pinpoint the key edges in the photograph.

The plot of the corners, cleaned up through NMS:

Figure 3: The corners of the image, as detected through the Harris filter. Note that there was a slight pick-up of noise in some situations (such as the tiny corners between window bars in individual windows), while some corners were altogether not detected (such as along the bottom by the doors). Fine-tuning the alpha value and increasing intensity contrast may be helpful to capture those and remove the noise.

Some key notes:

- **What happens when r changes for the radius of the disk?** It is noted that based on the mechanism of ordfilt2 and how non-maximum suppression works, we get a more "equalized scattering of points" and less precise corners that are captured. A plot with a larger radius value would result in a sparser distribution of the maximum points, as opposed to the above example where there are *larger clusters of points*. This is because the disk size that suppresses

all other non-maxima has grown larger, and so only specific maximum points in a larger suppression space are produced.

1.3 C

The code for Laplacian of Gaussian detection is as follows. It is based on a combination of features.m code and code from A1:

```
img = imread('synthetic.png');
img = double(img);
img = mean(img,3);

imgS = img;
cnt = 1;
clear responseLoG
k = 1.1;
sigma = 2.0;
s = k.^(1:30)*sigma;
maxes = zeros(size(img,1),size(img,2),length(s));
responseLoG = zeros(size(img,1),size(img,2),length(s));

T = 60.00;

for si = 1:length(s)
    sL = s(si); % grab the sigma value at scale # si
    hs = max(25,min(floor(sL*3),128)); % laplacian of gaussian kernel
        size
    HL = fspecial('log',[hs hs],sL); % laplacian of gaussian filter
    imgFiltL = conv2(imgS,HL,'same'); % convolve the image with the
        laplacian of gaussian
    %Compute the LoG
    responseLoG(:,:,si) = abs((sL^2)*imgFiltL); % normalize the result
        and store in responseLoG at layer si
    % edit - we used the abs() here because we are looking at the maximum
    % magnitude of the LoG -- will this work?
end
fg = figure;imagesc(img);axis image;hold on;colormap gray;
drawnow;


[maxValues, maxAtThisScale] = max(responseLoG, [], 3);
% goes through responseLoG 3D matrix
% outputs 2 (m x n) matrices

% maxValues -> at each pixel of the image (x,y), is the maximum value
% among all possible scales
% maxAtThiScale -> at each pixel of the image (x,y), is the scale at
    which
% the maximum value was found
```

```
[candY, candX] = find(maxValues > T);
numMaxPts = size([candY, candX], 1);
scaleLocation = zeros(numMaxPts, 1);

% [candY, candX] are two arrays that respectively output the
% y and x coordinate of all the max values greater than threshold T
% the number of max points is the number of elements in candY (or candX)
% scaleLocation is going to be the scale each max pt is found

for z = 1 : numMaxPts
    scaleLocation(z) = maxAtThisScale(candY(z), candX(z));
end

% go through every max point
% put in the scaleLocation array for that max point,
% the scale that the particular max was found


for currMaxPt = 1 : numMaxPts
    % for each max point
    currentScaleLayer = responseLoG(:,:,scaleLocation(currMaxPt));
    % get the LoG image for the max point's scale
    row = candY(currMaxPt);
    col = candX(currMaxPt);
    % get the coordinates of the max point
    if isLocalMaximum(col,row,currentScaleLayer) == 1
        % if the point is NOT a local maxium ...
        scaleValue = s(scaleLocation(currMaxPt));
        %scaleValue = sqrt(2) * scaleValue;
        % get the scale value from the s array
        % multiply it by sqrt(2) as based in the slides
        xc = scaleValue*sin(0:0.1:2*pi)+col;
        yc = scaleValue*cos(0:0.1:2*pi)+row;
        plot(xc,yc,'r');
        drawnow;
        % draw the circle
    end
end
```

The resulting plot at a threshold of 80.0 is as follows:

Some key points:

- **Why was the large top left circle not captured?** It is conjectured that this is because of the use of *max* as opposed to findpeaks. Due to time constraints in completing A2 a version of this to capture the top left could not fully be implemeneted. However it is conjectured that the reason the central extrema

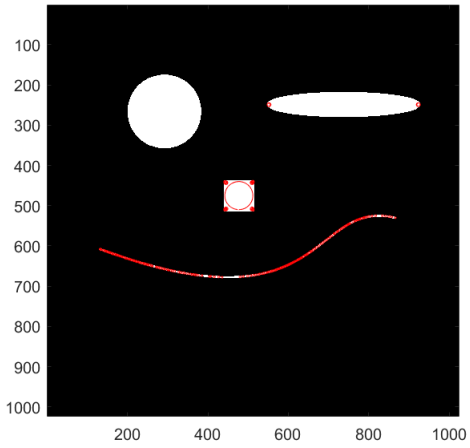

Figure 4: Blob detection at a threshold of 80.0. Note that the top left circle isn't fully captured. In lower thresholds, the top-left circles edges are captured as tiny blobs with small circles, and at the lowest thresholds, four circles appear roughly equidistant from each other within the circle at four different corners.

of the top left circle was not captured in the LoG scale layer corresponding to the radius of the circle was because the use of *max* and an absolute value on the negative *min* values did not capture the case where *all values were* > 0 *but still a local minimum exceeded a local maximum*.

1.4 D

Harris corner detection on the synthetic image:

Laplacian of Gaussian blob detection on the building image:

Some key comparison notes:

- In the Harris detection, note again that the top left circle is not captured. However, other corners are fairly adequately captured in terms of being proper "corners" rather than blobs.
- In the LoG building detection, many corners are suitably captured with the tiny circular "blobs". Some larger blobs capture distinctive features, such as the inside of windows. The corners captured on the left side of the image produce less noise than that of the Harris detector.

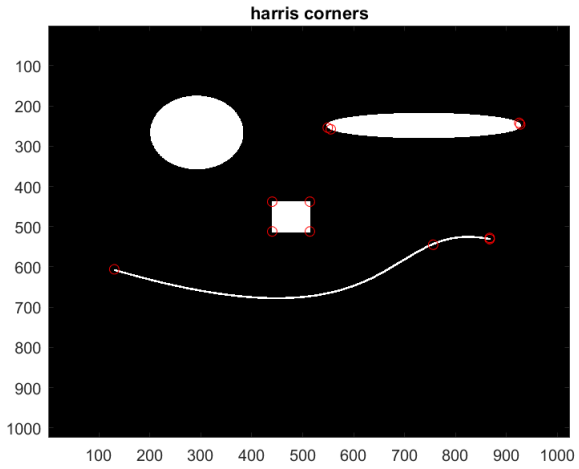

Figure 5: Harris detection on the synthetic image.

Figure 6: LoG on the building image.

- Some keypoints are different (such as the lack of detecting the entire smile with blobs with Harris) primarily because the LoG response (especially using max/min values explicitly) looks for situations where two edges "combine" to

form the "Mexican hat" signal shape that we can capture using a Laplacian of Gaussian. The Harris metric in contrast simply looks at intensity changes within specific windows. For this reason the Laplacian of Gaussian's lack of capturing more corners in the building image means that the ones that had been captured are considered *scale-invariant*.

- In contrast the Harris corners difference in the synthetic image are best attributable to the fact that roundness of shapes are not captured within a particular threshold where pixel jaggedness forming "corners" is not taken into account (a result that was seen even with the LoG on a low threshold).

2 Question 2

The entire code (including colours) **all comments here**):

sift.m:

```
% sift

% sift with colours

% read images
I10 = imread('colourTemplate.png');
I20 = imread('colourSearch.png');

% convert to grayscale
I1 = single(rgb2gray(I10));
I2 = single(rgb2gray(I20));

% SIFT keypoints and features
[keypoints1, descriptors1] = vl_sift(I1);
[keypoints2, descriptors2] = vl_sift(I2);

% transpose the results
k1 = transpose(keypoints1);
k2 = transpose(keypoints2);

d10 = transpose(descriptors1);
d20 = transpose(descriptors2);

% ONLY FOR 2e :
% loop from 1 to size of k1
% for each feature in d1, set cols 129:131 to be I10(x,y,1:3)

d1 = d10;
d2 = d20;

for q = 1:size(k1,1)
```

```
    xc = round(k1(q,1));
    yc = round(k1(q,2));
    d1(q,129) = I10(xc,yc,1);
    d1(q,130) = I10(xc,yc,2);
    d1(q,131) = I10(xc,yc,3);
end


for p = 1:size(k2,1)
    xc = round(k2(p,1));
    yc = round(k2(p,2));
    d2(q,129) = I20(xc,yc,1);
    d2(q,130) = I20(xc,yc,2);
    d2(q,131) = I20(xc,yc,3);
end

% END 2e-only section

% compute a vector the size of the number of keypoints in k1, and each
    non-zero corresponds to the index of the k2 pt
[matchedIndices, matchedFeatDistances] = matchFeatures(d1, d2, 0.8);
mIsize = size(matchedIndices,1);

% matchedFeatDistances - an M x 3 matrix where
% col 1 = points 1, col 2 = points 2, col 3 = their feature distance


matchedSize = size(find(matchedIndices > 0));
matchedNumber = matchedSize(1,1);

matchedPts1 = zeros(matchedNumber, 2);
matchedPts2 = zeros(matchedNumber, 2);

matchedWithDistances = zeros(1,5);

% matchedPts -> the points matched to each other based on matchFeatures
    results
% matchedWithDistances -> the matched points with their Euclidean
    distances (for affine)
j = 1;
for i = 1:mIsize
    currIndex = matchedIndices(i);
    if (currIndex > 0)
        matchedPts1(j,:) = k1(i,1:2);
        matchedPts2(j,:) = k2(currIndex,1:2);
        matchedWithDistances(j,1:2) = k1(i,1:2);
        matchedWithDistances(j,3:4) = k2(currIndex,1:2);
        matchedWithDistances(j,5) = matchedFeatDistances(i,2);
        j = j + 1;
    end
```

```
end

figure; ax = axes;
showMatchedFeatures(I10,I20,matchedPts1,matchedPts2,'montage','Parent',ax);
title(ax, 'Candidate point matches');
legend(ax, 'Matched points 1','Matched points 2');

% calculates the affine matrix
affines = findAffineTransform(matchedWithDistances, 5);

A = zeros(2,3); % our affine transformation matrix

A(1,1) = affines(1,1); % a
A(1,2) = affines(2,1); % b
A(1,3) = affines(5,1); % e
A(2,1) = affines(3,1); % c
A(2,2) = affines(4,1); % d
A(2,3) = affines(6,1); % f

maxY = size(I1,2); % 320
maxX = size(I1,1); % 499

c1 = [1 1 1];
c2 = [maxY 1 1]; % (1,320)
c3 = [1 maxX 1]; % (499, 1)
c4 = [maxY maxX 1]; % (499, 320)

cornersBase = zeros(4,2);
cornersTrfm = zeros(4,2);

% grabs the corners

cornersBase(1,:) = c1(1:2);
cornersBase(2,:) = c2(1:2);
cornersBase(3,:) = c3(1:2);
cornersBase(4,:) = c4(1:2);

c1 = transpose(c1);
c2 = transpose(c2);
c3 = transpose(c3);
c4 = transpose(c4);

% transforms the corners using the affine transform

cT1 = A * c1;
cT2 = A * c2;
cT3 = A * c3;
cT4 = A * c4;

cx1 = transpose(cT1);
```

```
cx2 = transpose(cT2);
cx3 = transpose(cT3);
cx4 = transpose(cT4);

cornersTrfm(1,:) = cx1;
cornersTrfm(2,:) = cx2;
cornersTrfm(3,:) = cx3;
cornersTrfm(4,:) = cx4;

% plots the results

figure; ax = axes;
showMatchedFeatures(I10,I20,cornersBase,cornersTrfm,'montage','Parent',ax);
title(ax, 'Affine corner matches');
legend(ax, 'Matched points 1','Matched points 2');

figure; ax = axes;
imshow(I20); hold on;
plot([cornersTrfm(1,1) cornersTrfm(2,1)], [cornersTrfm(1,2)
    cornersTrfm(2,2)],'g','LineWidth',3);
plot([cornersTrfm(1,1) cornersTrfm(3,1)], [cornersTrfm(1,2)
    cornersTrfm(3,2)],'g','LineWidth',3);
plot([cornersTrfm(3,1) cornersTrfm(4,1)], [cornersTrfm(3,2)
    cornersTrfm(4,2)],'g','LineWidth',3);
plot([cornersTrfm(2,1) cornersTrfm(4,1)], [cornersTrfm(2,2)
    cornersTrfm(4,2)],'g','LineWidth',3);
title(ax, 'Affine boundary');
```

---

matchedFeatures.m:

```
% for each feature vector in F1,
% use pdist to get the minimum and second minimum
% check if their quotient is less than the threshold
% if true, then the index of the smallest in F2 is the matchIndex value
    corresp. to F1 index

function [matchIndex, matchFeatDist] = matchFeatures(F1, F2, t)
    featEuclidDist = pdist2(F1,F2);
    numFeats = size(F1,1);
    matchIndex = zeros(numFeats,1);
    matchFeatDist = zeros(numFeats,2);
    for featIndex = 1:numFeats
        feat = featEuclidDist(featIndex,:);
        featSorted = sort(feat);
        featMin = featSorted(1);
        featSecondMin = featSorted(2);
        threshold = featMin / featSecondMin;
        if (threshold < t)
            minIndex = find(feat == featMin);
```

```
            matchIndex(featIndex,1) = minIndex;
            matchFeatDist(featIndex,1) = minIndex;
            matchFeatDist(featIndex,2) = featMin;
        end
    end
end
```

findAffineTransform.m:

```
function A = findAffineTransform(matchedPoints, k)
    matchedPointsSorted = sortrows(matchedPoints,5);
    kCorrespondences = matchedPointsSorted(1:k,:);

    P = zeros(k*2,6);
    Pht = zeros(k*2,1);

    % builds the matrix as seen in the lecture slides

    j = 1;
    for i = 1:k*2
        if (mod(i,2) ~= 0)
            P(i,1:2) = kCorrespondences(j,1:2);
            P(i,5:6) = [1 0];
            P(i+1,3:4) = kCorrespondences(j,1:2);
            P(i+1,5:6) = [0 1];
            Pht(i,1) = kCorrespondences(j,3);
            Pht(i+1,1) = kCorrespondences(j,4);
            j = j + 1;
        end
    end

    % calculates the moore-penrose inverse and
    % does the matrix calculation

    Pt = transpose(P);
    PInv = pinv(Pt * P);
    PRight = Pt * Pht;
    A = PInv * PRight;

end
```

2.1 A

Code subsection:

```
I1 = single(rgb2gray(I10));
I2 = single(rgb2gray(I20));
```

```
[keypoints1, descriptors1] = vl_sift(I1);
[keypoints2, descriptors2] = vl_sift(I2);
```

Book features:

Figure 7: Caption

Findbook features:

2.2 B

Code subsection:

```
for q = 1:size(k1,1)
    xc = round(k1(q,1));
    yc = round(k1(q,2));
    d1(q,129) = I10(xc,yc,1);
    d1(q,130) = I10(xc,yc,2);
    d1(q,131) = I10(xc,yc,3);
end
```

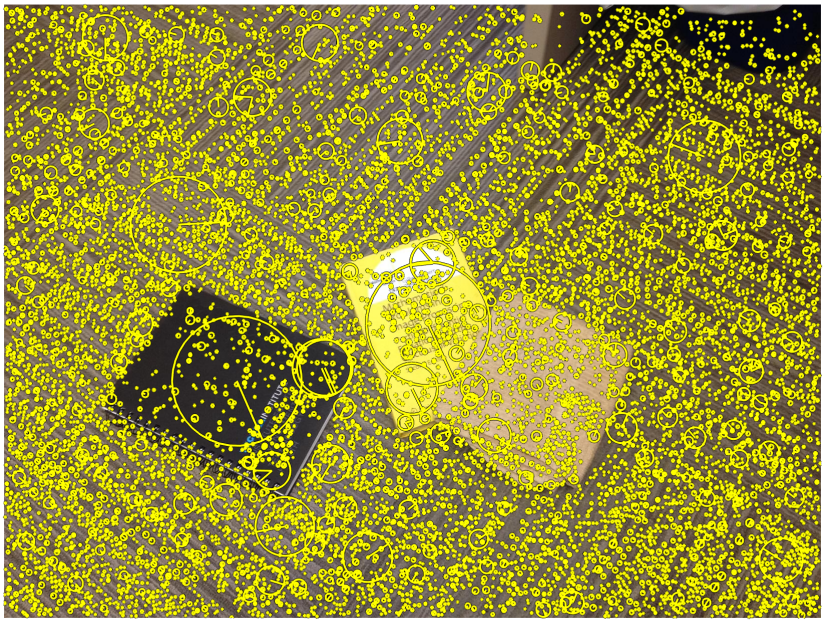

Figure 8: Caption

```
for p = 1:size(k2,1)
    xc = round(k2(p,1));
    yc = round(k2(p,2));
    d2(q,129) = I20(xc,yc,1);
    d2(q,130) = I20(xc,yc,2);
    d2(q,131) = I20(xc,yc,3);
end

[matchedIndices, matchedFeatDistances] = matchFeatures(d1, d2, 0.8);
mIsize = size(matchedIndices,1);

% matchedFeatDistances - an M x 3 matrix where
% col 1 = points 1, col 2 = points 2, col 3 = their feature distance



matchedSize = size(find(matchedIndices > 0));
matchedNumber = matchedSize(1,1);

matchedPts1 = zeros(matchedNumber, 2);
matchedPts2 = zeros(matchedNumber, 2);
```

```
matchedWithDistances = zeros(1,5);

j = 1;
for i = 1:mIsize
    currIndex = matchedIndices(i);
    if (currIndex > 0)
        matchedPts1(j,:) = k1(i,1:2);
        matchedPts2(j,:) = k2(currIndex,1:2);
        matchedWithDistances(j,1:2) = k1(i,1:2);
        matchedWithDistances(j,3:4) = k2(currIndex,1:2);
        matchedWithDistances(j,5) = matchedFeatDistances(i,2);
        j = j + 1;
    end
end

figure; ax = axes;
showMatchedFeatures(I10,I20,matchedPts1,matchedPts2,'montage','Parent',ax);
title(ax, 'Candidate point matches');
legend(ax, 'Matched points 1','Matched points 2');
```

```
function [matchIndex, matchFeatDist] = matchFeatures(F1, F2, t)
    featEuclidDist = pdist2(F1,F2);
    numFeats = size(F1,1);
    matchIndex = zeros(numFeats,1);
    matchFeatDist = zeros(numFeats,2);
    for featIndex = 1:numFeats
        feat = featEuclidDist(featIndex,:);
        featSorted = sort(feat);
        featMin = featSorted(1);
        featSecondMin = featSorted(2);
        threshold = featMin / featSecondMin;
        if (threshold < t)
            minIndex = find(feat == featMin);
            matchIndex(featIndex,1) = minIndex;
            matchFeatDist(featIndex,1) = minIndex;
            matchFeatDist(featIndex,2) = featMin;
        end
    end
end
```

Matched features:

2.3 C

Code subsection:

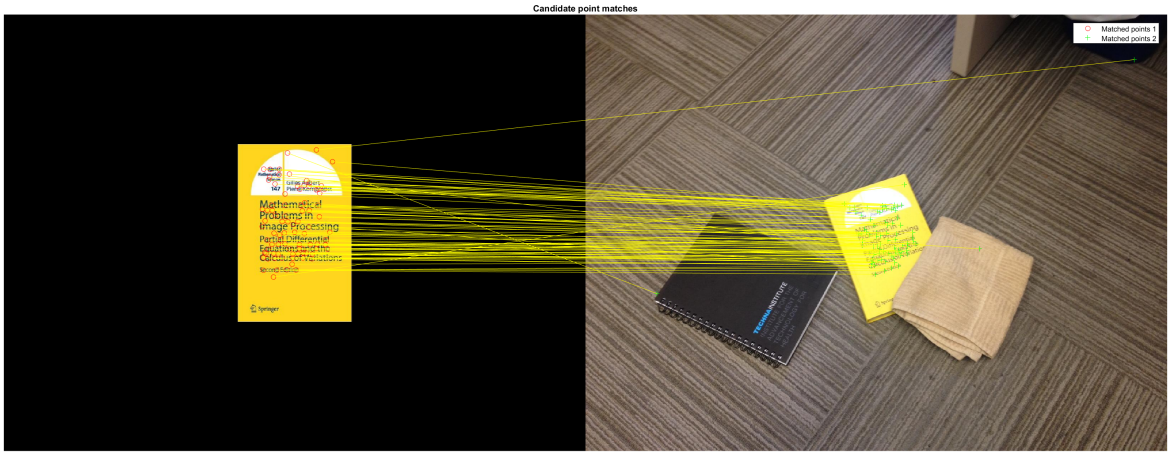

Figure 9: Caption

```
affines = findAffineTransform(matchedWithDistances, 5);

A = zeros(2,3); % our affine transformation matrix

A(1,1) = affines(1,1); % a
A(1,2) = affines(2,1); % b
A(1,3) = affines(5,1); % e
A(2,1) = affines(3,1); % c
A(2,2) = affines(4,1); % d
A(2,3) = affines(6,1); % f

maxY = size(I1,2); % 320
maxX = size(I1,1); % 499

c1 = [1 1 1];
c2 = [maxY 1 1]; % (1,320)
c3 = [1 maxX 1]; % (499, 1)
c4 = [maxY maxX 1]; % (499, 320)

cornersBase = zeros(4,2);
cornersTrfm = zeros(4,2);
```

```
cornersBase(1,:) = c1(1:2);
cornersBase(2,:) = c2(1:2);
cornersBase(3,:) = c3(1:2);
cornersBase(4,:) = c4(1:2);

c1 = transpose(c1);
c2 = transpose(c2);
c3 = transpose(c3);
c4 = transpose(c4);

cT1 = A * c1;
cT2 = A * c2;
cT3 = A * c3;
cT4 = A * c4;

cx1 = transpose(cT1);
cx2 = transpose(cT2);
cx3 = transpose(cT3);
cx4 = transpose(cT4);

cornersTrfm(1,:) = cx1;
cornersTrfm(2,:) = cx2;
cornersTrfm(3,:) = cx3;
cornersTrfm(4,:) = cx4;

figure; ax = axes;
showMatchedFeatures(I10,I20,cornersBase,cornersTrfm,'montage','Parent',ax);
title(ax, 'Affine corner matches');
legend(ax, 'Matched points 1','Matched points 2');

figure; ax = axes;
imshow(I20); hold on;
plot([cornersTrfm(1,1) cornersTrfm(2,1)], [cornersTrfm(1,2)
    cornersTrfm(2,2)],'g','LineWidth',3);
plot([cornersTrfm(1,1) cornersTrfm(3,1)], [cornersTrfm(1,2)
    cornersTrfm(3,2)],'g','LineWidth',3);
plot([cornersTrfm(3,1) cornersTrfm(4,1)], [cornersTrfm(3,2)
    cornersTrfm(4,2)],'g','LineWidth',3);
plot([cornersTrfm(2,1) cornersTrfm(4,1)], [cornersTrfm(2,2)
    cornersTrfm(4,2)],'g','LineWidth',3);
title(ax, 'Affine boundary');
```

```
function A = findAffineTransform(matchedPoints, k)
    matchedPointsSorted = sortrows(matchedPoints,5);
    kCorrespondences = matchedPointsSorted(1:k,:);

    P = zeros(k*2,6);
    Pht = zeros(k*2,1);
```

```
    j = 1;
    for i = 1:k*2
        if (mod(i,2) ~= 0)
            P(i,1:2) = kCorrespondences(j,1:2);
            P(i,5:6) = [1 0];
            P(i+1,3:4) = kCorrespondences(j,1:2);
            P(i+1,5:6) = [0 1];
            Pht(i,1) = kCorrespondences(j,3);
            Pht(i+1,1) = kCorrespondences(j,4);
            j = j + 1;
        end
    end

    Pt = transpose(P);
    PInv = pinv(Pt * P);
    PRight = Pt * Pht;
    A = PInv * PRight;

end
```

2.4 D

The plots for different parameters:

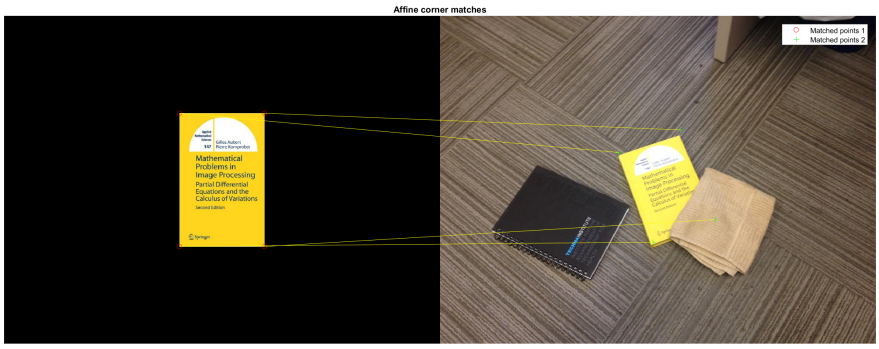

Figure 10: K = 5

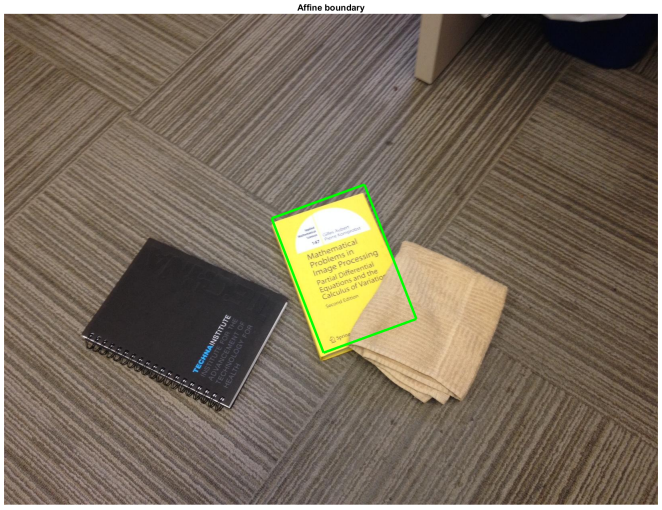

Figure 11: K = 5 Boundary

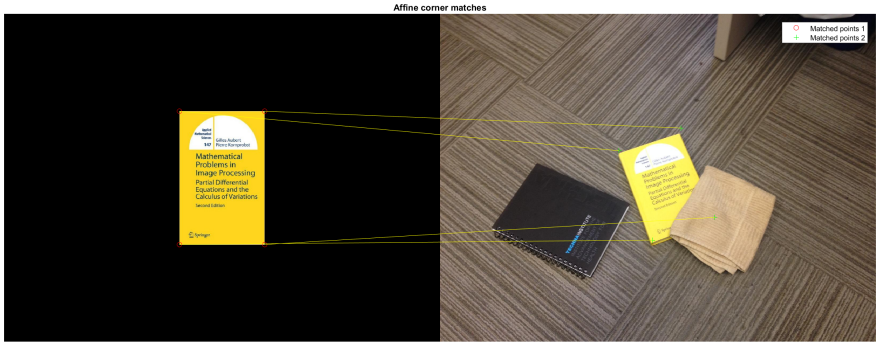

Figure 12: K = 10

2.5 E

All the above code was presented inclusive of the colour factors. **Note that I did not take colour into account for our initial book-matching. This is just the**

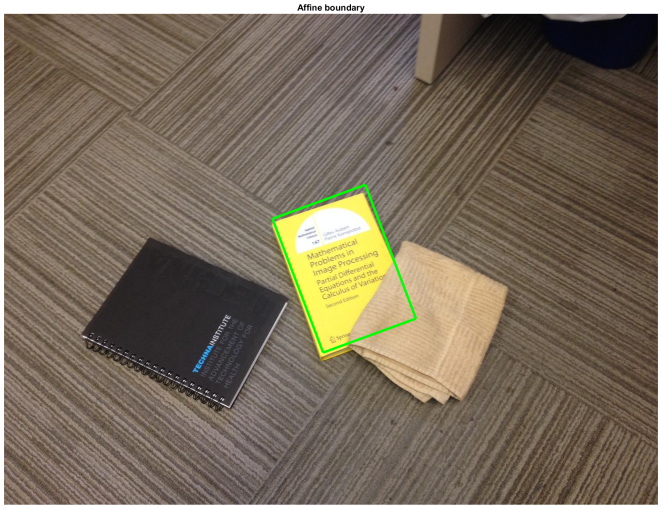

Figure 13: K = 10 Boundary

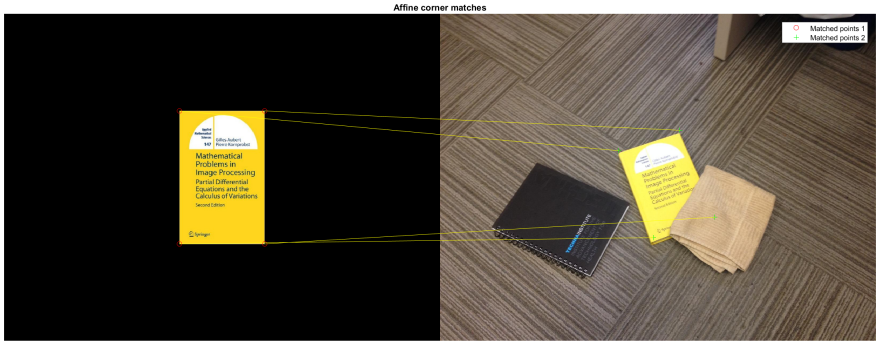

Figure 14: K = 50

current iteration of the code, inclusive of colour matching.

- **Method of adding colour**. After extracting SIFT features, each individual vector is 1×128. We *extend this* to a 1×131 vector, with $(1, 129)$ being the **red intensity channel**, $(1, 130)$ being the **blue intensity channel**, and $(1, 131)$ being the **green intensity channel**.
- **Justification.** The desire is to *include colour information in the features*. It is understood that point-matching is predicated upon the *Euclidean distance of the feature vectors*. Thus, by including colour information in each feature vectors, two similar feature vectors would be closer if there was a smaller differ-

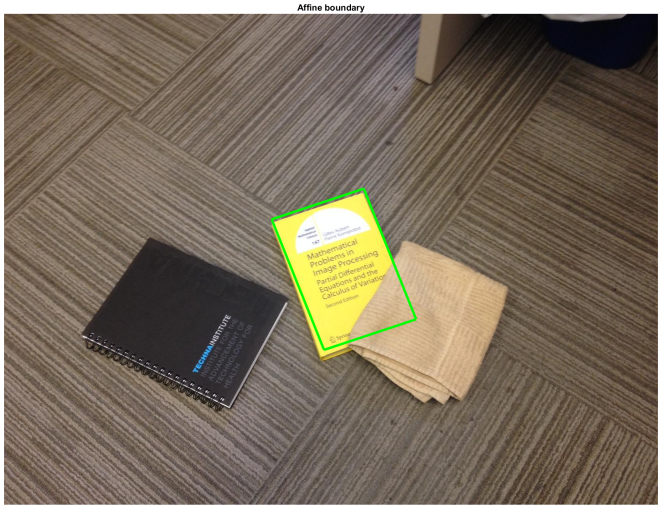

Figure 15: K = 50 Boundary

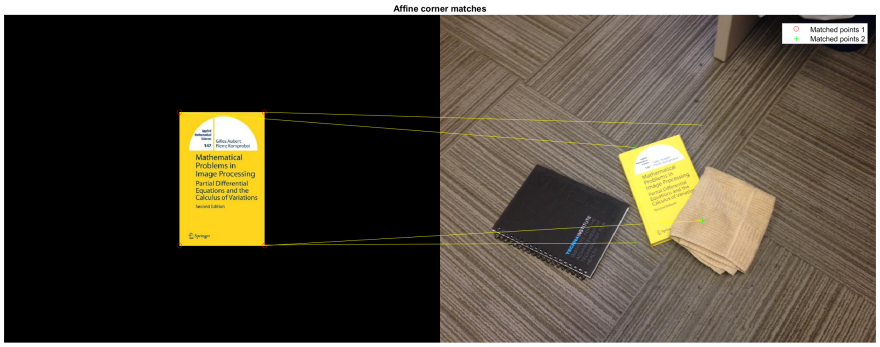

Figure 16: K = 100

ence in their colour components, AKA, if the colour channel values matched. Thus, the rationale was that including the three colour channel values in each feature vector would simply include the colour component when calculating the Euclidean distance between descriptors, and would thus influence matching, being able to match points that were "closer" in colour along with other detected SIFT features.

- **Results**: the following matching and affine transformation is seen below. Note that our corner points and thus parallelogram are scaled because of the size of the red square, but is otherwise identical (and can easily be scaled down

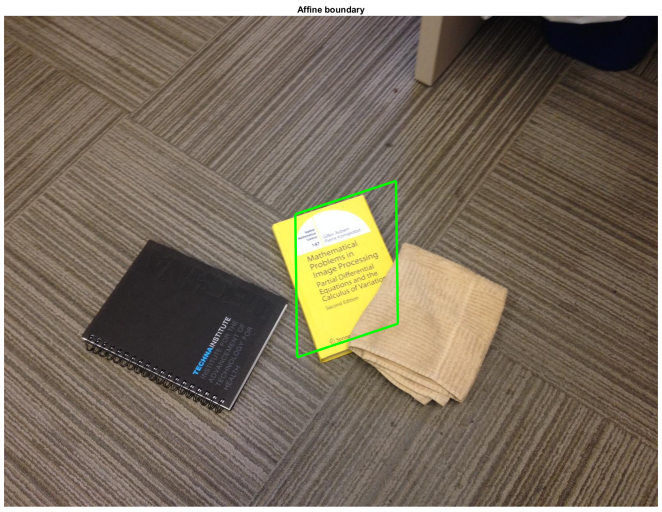

Figure 17: K = 100 Boundary. Note how distorted this transformation is in comparison. This is likely because the excess of points has exaggerated the degree of "affine"-ness when in reality this is not an exact affine transform.

if necessary to match, indeed the inclusion of the black space around the red square was taken into account when matching to the right-side image, which harbours the exact orientation).

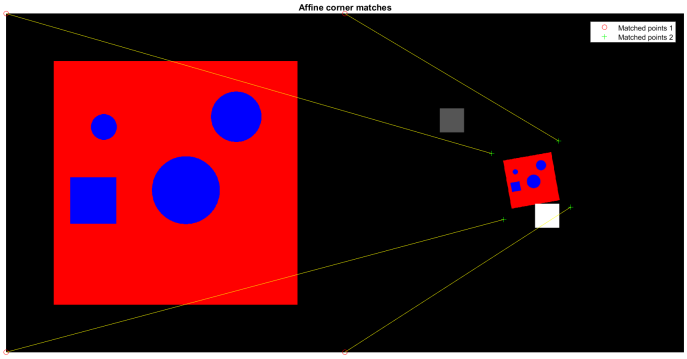

Figure 18: The colour match.

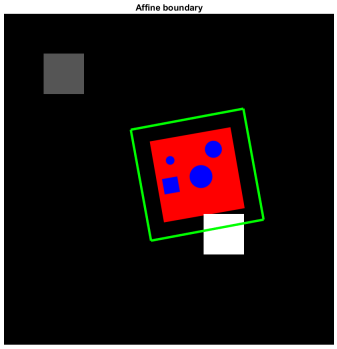

Figure 19: The matched boundary. The colour feature inclusion appears to have succeeded in this context.

3 Question 3

3.1 A

3.1.1 Brute Force

The following code was used to compute a few brute-force results (note in a Piazza answer that it was okay to omit tolerance and simply use fspecial):

```
% logsep

% brute force:

sig1 = 1.0;
sig2 = 5.0;
sig3 = 10.0;

kernel1 = fspecial('log', 3, sig3);
kernel2 = fspecial('log', 5, sig3);
kernel3 = fspecial('log', 7, sig3);
kernel4 = fspecial('log', 9, sig3);

[U1,S1,V1] = svd(kernel1);
[U2,S2,V2] = svd(kernel2);
[U3,S3,V3] = svd(kernel3);
[U4,S4,V4] = svd(kernel4);
```

```
% sig1
% S1: 0.3101 and 0.2030
% S2: 0.3856, 0.1260, 0.0013
% S3: 0.3872, 0.1052, 0.0003
% S4: 0.3871, 0.1045

% sig2
% S1: (1.0 x e-03) * 0.5022, 0.4951
% S2: 0.0010, 0.0010
% S3: 0.0015, 0.0014
% S4: 0.0019, 0.0016

% sig3
% S1: (1.0e-04) * 0.3142, 0.3131
% S2: (1.0e-04) * 0.6658, 0.6574
% S3: (1.0e-04) * 0.9770, 0.9519
% S4: (1.0e-03) * 0.1270, 0.1216
```

All resulting filter kernels had 2 or 3 singular values. Thus, out of these brute force tests, *no LoG kernel was separable*. This leads to the hypothesis that the Laplacian of Gaussian in general is *not separable*.

3.1.2 Analysis

When analytically computing the Laplacian of the Gaussian function (in two dimensions), inspection will show that unlike the separability of the Gaussian where an exponential product with independent x and y can be performed, there is no way to separate the Laplacian of the Gaussian making x and y independent of each other.

3.2 B

Here is the code used for the approximation:

```
log = logfilt(1.55, 99); % s = 1.55
g1 = gaussfilt(2.0, 99); % s = 2.0
g2 = gaussfilt(1.0, 99); % s = 1.0
dog = g1 - g2;

figure; ax = axes; hold on;
plot(log, 'r');
plot(dog, 'g');
title(ax, 'LoG compared to DoG');
legend(ax, 'LoG', 'DoG');
```

The resulting plot:

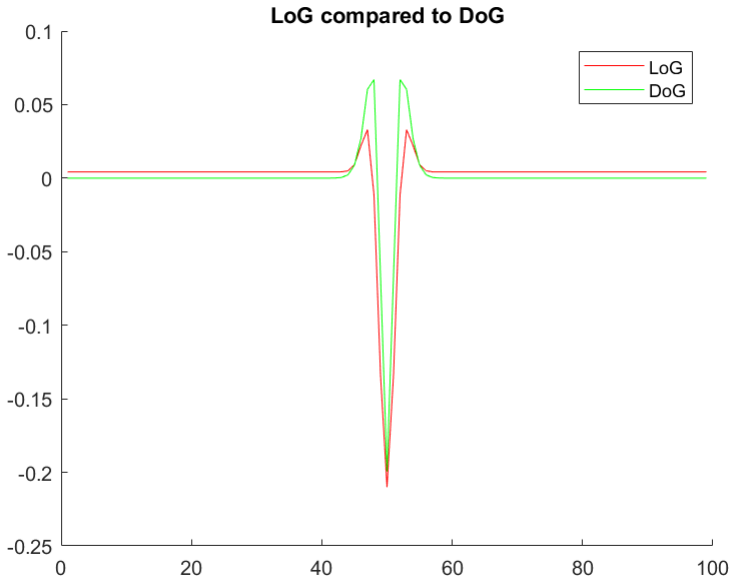

Figure 20: DoG and LoG plot with the below sigmas and k = 99

- **Sigma declarations**.
  - **LoG sigma**: 1.55
  - **DoG sigma 1**: 2.0
  - **DoG sigma 2**: 1.0
  - The ideal LoG sigma is approximately between the DoG sigmas in this specific case.