1. (a) Matlab code for Harris Corner Detector using Harmonic Mean.

```
% Harris corner detection
function out = harris(filename)
im = imread(filename);
img = rgb2gray(im); %get intensity values

sigma = 3;
imgS = conv2(img,fspecial('Gaussian',[25 25],sigma),'same');

[Gx,Gy] = imgradientxy(imgS);

Gx2 = Gx.*Gx;
Gy2 = Gy.*Gy;
GxGy = Gx.*Gy;

%Guassian is rotationally symmetric, three conv in total
sigma_w = 16;
guass_filt = fspecial('gaussian', [25, 25], sigma_w);
Mx2 = conv2(Gx2, guass_filt, 'same');
My2 = conv2(Gy2, guass_filt, 'same');
Mxy = conv2(GxGy, guass_filt, 'same');

% Harmonic mean detector
determinant = Mx2.*My2 - Mxy.*Mxy;
trace = Mx2 + My2;
R_val = determinant ./ trace;

%threshold
max_R = max(max(R_val));
threshold = max_R*0.028;

[r, c] = size(img);
local_max = zeros(r, c);

%non maximal supression within neighbourhood of 3x3
%We check if R points are above threshold
thresh_suppress = (R_val > threshold);
[sx, sy] = size(R_val);
% Keep if pixel is greater than surrounding pixels
local_maxima_nonpad = (R_val(2:sx-1,2:sy-1) > R_val(1:sx-2,1:sy-2)) &
    ...
    (R_val(2:sx-1,2:sy-1) > R_val(1:sx-2,2:sy-1)) & ...
    (R_val(2:sx-1,2:sy-1) > R_val(1:sx-2,3:sy)) & ...
    (R_val(2:sx-1,2:sy-1) > R_val(2:sx-1,1:sy-2)) & ...
    (R_val(2:sx-1,2:sy-1) > R_val(2:sx-1,3:sy)) & ...
    (R_val(2:sx-1,2:sy-1) > R_val(3:sx,1:sy-2)) & ...
    (R_val(2:sx-1,2:sy-1) > R_val(3:sx,2:sy-1)) & ...
    (R_val(2:sx-1,2:sy-1) > R_val(3:sx,3:sy));

local_max(2:sx-1, 2:sy-1) = local_maxima_nonpad;
local_max = local_max & thresh_suppress;
```

```
%get coordinates of corner points
[pcols, prows] = find(local_max == 1);
imshow(img);
hold on;
plot(prows, pcols, 'r.');

end
```

Figure 1: Result of Harris Corner detector on building.jpg

(b) Matlab code for Non-Maximal Suppression function.

```
function mx = nonmax(im, radius)
%radius - radius of region considered in non-maximal
%        suppression.

%ordfilt2(A,order,domain) replaces each element in A by the orderth
%element in the sorted set of neighbors specified by the nonzero
%elements in domain.
% perform non-maximal suppression using ordfilt2
domain=fspecial('disk',radius)>0; %dilation mask, circular element
num_pixels_domain = sum(sum(domain));
dilated = ordfilt2(im, num_pixels_domain, domain); % Grey-scale dilate.
bin = (im == dilated);
mx = im(:, :);
mx(bin == 0) = 0;
```

Harris Corner Detector Code modified to use this function

```
% Harris corner detection with 'nonmax' suppression function
function out = harris(filename)
im = imread(filename);
img = rgb2gray(im); %get intensity values

sigma = 3;
imgS = conv2(img,fspecial('Gaussian',[25 25],sigma),'same');

[Gx,Gy] = imgradientxy(imgS);

Gx2 = Gx.*Gx;
Gy2 = Gy.*Gy;
GxGy = Gx.*Gy;

%Guassian is rotationally symmetric, three conv in total
sigma_w = 16;
guass_filt = fspecial('gaussian', [25, 25], sigma_w);
Mx2 = conv2(Gx2, guass_filt, 'same');
My2 = conv2(Gy2, guass_filt, 'same');
Mxy = conv2(GxGy, guass_filt, 'same');

% Harmonic mean detector
determinant = Mx2.*My2 - Mxy.*Mxy;
trace = Mx2 + My2;
R_val = determinant ./ trace;

%threshold
max_R = max(max(R_val));
threshold = max_R*0.15;

% Perform non-maximal surpression
mx = nonmax(R_val, 3);
% Find the coordinates of the corner points above threshold
corner_p = mx > threshold;

%get coordinates of corner points
[pcols, prows] = find(corner_p);
imshow(img);
hold on;
plot(prows, pcols,'r*');

end
```

As r, the radius of the non-maximal suppression disk becomes bigger, the window or area across which we search for the local maximum (suppressing all other values as zeros) becomes bigger. This means that the corners which correspond to local maximums become sparser in the image, because they are detected across a "local" window with greater radius.
ordfilt2(A,order,domain) replaces each element in a matrix copy of R-val (the corner detection scores for each pixel in the image) by the greatest value present in the circular window (specified by the nonzero elements in domain) centered at that pixel element. The true local maximum pixels

which have a greater value than all other pixels in the domain-sized circular window surrounding it will persist in value to the end of the convolution. All other pixels that are non-maximums in their respective circular window will be swallowed by the maximums in value (their values will change from the original). Thus, we collect the positions of pixels with unchanged values from the original R-val image-sized matrix, which represent the local maximums.
If we increase the radius of the circular window, we can have a subset of the previous coordinates of resulting maximum values because there is a greater chance that smaller maximums are within 'radius' distance of larger maximums, and they will be suppressed by our function. And the maximums (detected corner points) will be more sparsely distributed in the result because the general trend is that there will be fewer numbers of them, and they are at least 'radius' distance away from each other.

Harris corners with non-maximal suppression of radius 3, 25, 50 from top to bottom

(c) Helper function for Scale-invariant Feature Detection with LoG

```
function extrema = extrema(top, current, down)
% Function to find the extrema keypoints given 3 matrices
% A pixel is a keypoint if it is the extremum of its 26 neighbors (8 in
    current, and 9 each in top and bottom)

[sx, sy] = size(current);

% Look for local maxima
% Check the 8 neighbors around the pixel in the same level

local_maxima = (current(2:sx-1,2:sy-1) > current(1:sx-2,1:sy-2)) & ...
        (current(2:sx-1,2:sy-1) > current(1:sx-2,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) > current(1:sx-2,3:sy)) & ...
        (current(2:sx-1,2:sy-1) > current(2:sx-1,1:sy-2)) & ...
        (current(2:sx-1,2:sy-1) > current(2:sx-1,3:sy)) & ...
        (current(2:sx-1,2:sy-1) > current(3:sx,1:sy-2)) & ...
        (current(2:sx-1,2:sy-1) > current(3:sx,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) > current(3:sx,3:sy));

% Check the 9 neighbors in the level above it
local_maxima = local_maxima & (current(2:sx-1,2:sy-1) > top(1:sx-2,1:sy
    -2)) & ...
        (current(2:sx-1,2:sy-1) > top(1:sx-2,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) > top(1:sx-2,3:sy)) & ...
        (current(2:sx-1,2:sy-1) > top(2:sx-1,1:sy-2)) & ...
        (current(2:sx-1,2:sy-1) > top(2:sx-1,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) > top(2:sx-1,3:sy)) & ...
        (current(2:sx-1,2:sy-1) > top(3:sx,1:sy-2)) & ...
        (current(2:sx-1,2:sy-1) > top(3:sx,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) > top(3:sx,3:sy));

% Check the 9 neighbors in the level below it
local_maxima = local_maxima & (current(2:sx-1,2:sy-1) > down(1:sx-2,1:
    sy-2)) & ...
        (current(2:sx-1,2:sy-1) > down(1:sx-2,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) > down(1:sx-2,3:sy)) & ...
        (current(2:sx-1,2:sy-1) > down(2:sx-1,1:sy-2)) & ...
        (current(2:sx-1,2:sy-1) > down(2:sx-1,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) > down(2:sx-1,3:sy)) & ...
        (current(2:sx-1,2:sy-1) > down(3:sx,1:sy-2)) & ...
        (current(2:sx-1,2:sy-1) > down(3:sx,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) > down(3:sx,3:sy));

% Look for local minima
% Check the 8 neighbors around the pixel in the same level
local_minima = (current(2:sx-1,2:sy-1) < current(1:sx-2,1:sy-2)) & ...
        (current(2:sx-1,2:sy-1) < current(1:sx-2,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) < current(1:sx-2,3:sy)) & ...
        (current(2:sx-1,2:sy-1) < current(2:sx-1,1:sy-2)) & ...
        (current(2:sx-1,2:sy-1) < current(2:sx-1,3:sy)) & ...
        (current(2:sx-1,2:sy-1) < current(3:sx,1:sy-2)) & ...
```

```
        (current(2:sx-1,2:sy-1) < current(3:sx,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) < current(3:sx,3:sy)) ;

% Check the 9 neighbors in the level above it
local_minima = local_minima & (current(2:sx-1,2:sy-1) < top(1:sx-2,1:sy
    -2)) & ...
        (current(2:sx-1,2:sy-1) < top(1:sx-2,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) < top(1:sx-2,3:sy)) & ...
        (current(2:sx-1,2:sy-1) < top(2:sx-1,1:sy-2)) & ...
        (current(2:sx-1,2:sy-1) < top(2:sx-1,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) < top(2:sx-1,3:sy)) & ...
        (current(2:sx-1,2:sy-1) < top(3:sx,1:sy-2)) & ...
        (current(2:sx-1,2:sy-1) < top(3:sx,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) < top(3:sx,3:sy));

% Check the 9 neighbors in the level below it
local_minima = local_minima & (current(2:sx-1,2:sy-1) < down(1:sx-2,1:
    sy-2)) & ...
        (current(2:sx-1,2:sy-1) < down(1:sx-2,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) < down(1:sx-2,3:sy)) & ...
        (current(2:sx-1,2:sy-1) < down(2:sx-1,1:sy-2)) & ...
        (current(2:sx-1,2:sy-1) < down(2:sx-1,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) < down(2:sx-1,3:sy)) & ...
        (current(2:sx-1,2:sy-1) < down(3:sx,1:sy-2)) & ...
        (current(2:sx-1,2:sy-1) < down(3:sx,2:sy-1)) & ...
        (current(2:sx-1,2:sy-1) < down(3:sx,3:sy));

extrema_nonpad = local_maxima | local_minima;

extrema = zeros(size(current));
extrema(2:sx-1, 2:sy-1) = extrema_nonpad;

end
```

Matlab code for Scale-invariant Feature Detection with LoG

```
function out = blob_detector(filename)

source = rgb2gray(imread(filename));
img = double(source);
% Initial parameter settings
num_scales = 25;
antialias_sigma = 0.5;
contrast_threshold = 0.03;
curvature_const = 12;

%% Filter over a set of scales
LoG = zeros(size(img,1),size(img,2),num_scales+2);

k = 2^(16/num_scales); %1.1, 2.0
initSigma = 2.0;
imgS = img;
```

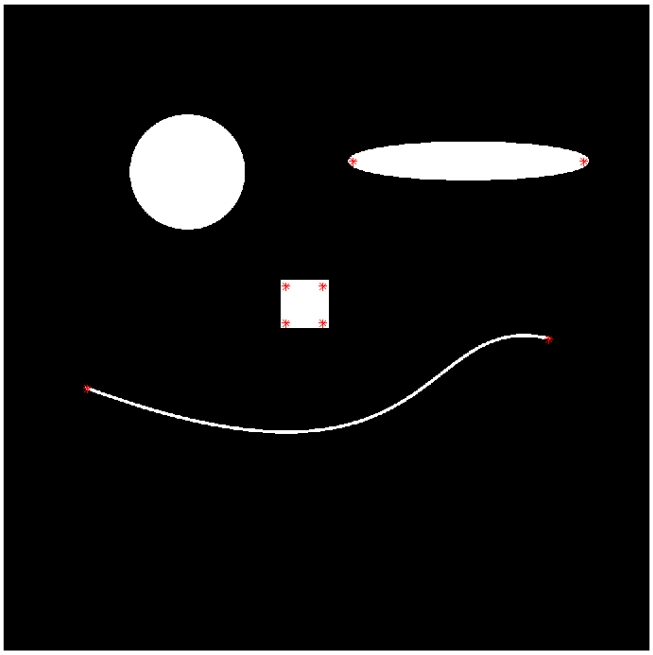
```
for sc = 1:num_scales+2
    sigma =  k.^(sc-1)*initSigma;
    hs= max(25,min(floor(sigma*6),128));
    HL =  fspecial('log', [hs hs], sigma); %normalized kernel
    imFiltered = conv2(imgS,HL,'same'); % filter the image with LoG
    % save square of the response for current level
    LoG(:, :, sc) = (sigma^2)* imFiltered;
end

disp('    Created LoG Scale Space');

%% Get Keypoints: LoG extrema
% top and bottom scales are only used to check for extrema within
    middle
% space. LoGs of scale 2, 3, 4 correspond to LoG_Keypoints of scales 1,
     2
LoG_Keypoints = zeros(size(img,1),size(img,2),num_scales);
for sc = 2:num_scales
        LoG_Keypoints(:,:,sc-1) = extrema(LoG(:,:,sc+1), LoG(:,:,sc),
            LoG(:,:,sc-1));
end

disp('    Got KeyPoints: LoG extrema');

%% Filter Keypoints: Remove low contrast and edge keypoints
for sc = 1:num_scales
    [points_x, points_y] = find(LoG_Keypoints(:,:,sc));  % indices of
        the Keypoints

    %find(DoG_Keypts{oct,sc})
    num_keypoints = length(points_x); % number of Keypoints
    scale = LoG(:, :, sc); %test all keypoints on this scale

    for k = 1:num_keypoints %for each keypoint
        x = points_x(k);
        y = points_y(k);
        %% Filter points with low contrast
        if (abs(scale(x,y)) < contrast_threshold)
            LoG_Keypoints(x,y,sc) = 0;
        else % Filter keypoints located on edges
            % Compute a 2x2 Hessian Matrix at (x,y) from LoG
            Dxy = scale(x-1,y-1) + scale(x+1,y+1) - scale(x-1,y+1) -
                scale(x+1,y-1);
            Dxx = scale(x-1,y) + scale(x+1,y) - 2*scale(x,y);
            Dyy = scale(x,y+1) + scale(x,y-1) - 2*scale(x,y);

            %% Detect edge points
            trace = Dxx + Dyy;
            determinant = Dxx*Dyy - Dxy*Dxy;
            curvature = (trace^2)/determinant;
            curv_thresh = ((curvature_const+1)^2)/curvature_const;
            if (curvature > curv_thresh || determinant < 0)
```

```
                LoG_Keypoints(x,y,sc)= 0;
            end
        end
    end
end

disp('    Filtered out keypoints with low contrast or on edges');

%% Visualize Result
imshow(source);
hold on;
for i=1:num_scales
    scale = LoG_Keypoints(:,:,i);
    [pcols, prows] = find(scale);
    plot(prows, pcols,'r.');
    for j=1:length(pcols)
        rad = i*2^(32/num_scales) * 1.1;
        xc = rad.*sin(0:0.1:(2*pi)) + pcols(j);
        yc = rad.*cos(0:0.1:(2*pi)) + prows(j);
        plot(yc, xc,'r');
    end
end

end
```

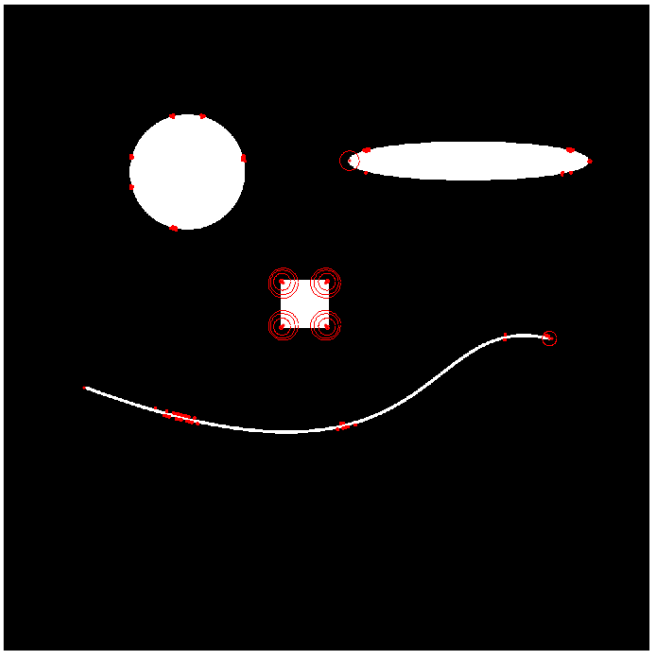
(d) The Harris corner detector and the Laplacian of Gaussian are both based on the same principle: distinctive patches of the images that can be extracted as features have the common property that shifting a window in any direction produces a large change in appearance. The Harris corner detector is more specialized towards corners because it detects points with large intensity change in the two principal orthogonal gradient directions at the point. Furthermore, the scale at which the corners are found is restricted: the patch which we limit our gradient filter to must contain a sharp distinctive corner for it be detected via the Harris method. Thus, all corners in the synthetic image are detected perfectly, and all the junctions in building.jpg image such as the corners of the window frames, the corners of the triangles in the wall detail, the tip of the downward facing cone at the bottom of the cylindrical window are all detected with high accuracy.

The Laplacian of Gaussian detector also builds upon the shifting window observation, however it allows a greater diversity in the features it detects because the Laplacian of Gaussian is a circularly symmetric operator, so it finds difference in all directions, not just two principal orthogonal directions like in Harris. So it is more suited to blob detection, and it does this across a number of scales (the Laplacian of Gaussian filter grows bigger and applies greater smoothing of the image to compensate for increased probability of greater amount of noise in blob contours). Intuitively, the Laplacian of Gaussian will also detect features at corners of an image because corners have high gradients in at least two different orientations, which correspond to maxima along two different directions in the 2nd derivative (the Laplacian is analog to second derivative), so we have a convex or elliptical point of change, similar to the image of a blob. In the synthetic image the LoG was able to detect the corners in the same places as Harris. It was also less immune to false corners such as the curved edges of the circle than Harris; Harris requires sharp change. In the building.jpg image the LoG was able to detect the corners among the smaller scales (smaller sigma and smaller LoG filter window), as well as detect obvious blob structures such as window patches, and the narrow spaces between two walls or frames. It is also more susceptible to noise which it regarded as distinctive patches as well (features detected on the uniform white wall).

Harris corners for synthetic.png



Laplacian of Gaussian detector for synthetic.png



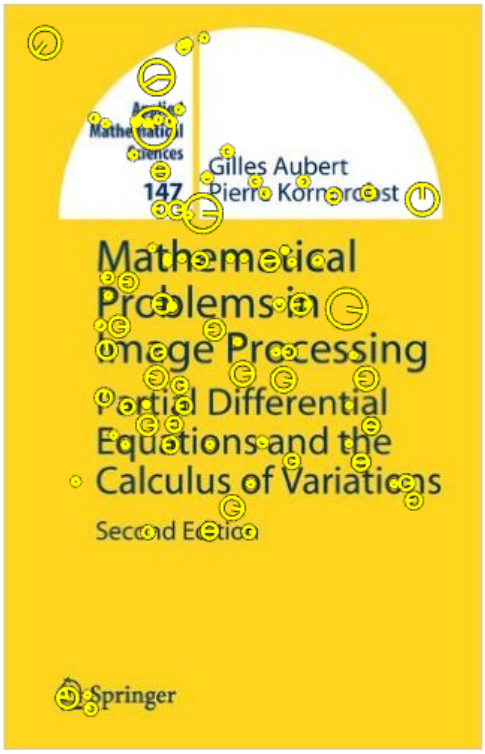
Harris corners for building.png

Laplacian of Gaussian detector for building.png

2. (a) Extracting SIFT keypoints and features

```
% SIFT features for reference.png and test.png
function out = sift_features(im1, im2)

% read images and grayscale
img1_col = imread(im1);
img1 = single(rgb2gray(img1_col)) ;
img2_col = imread(im2);
img2 = single(rgb2gray(img2_col)) ;

%compute the SIFT frames (keypoints) and descriptors
[f_im1,d_im1] = vl_sift(img1) ;
[f_im2,d_im2] = vl_sift(img2) ;

imshow(img1_col);
hold on;
% Plot images
perm = randperm(size(f_im1,2)) ;
sel = perm(1:100) ;
h1 = vl_plotframe(f_im1(:,sel)) ;
h2 = vl_plotframe(f_im1(:,sel)) ;
set(h1,'color','k','linewidth',3);
set(h2,'color','y','linewidth',2);
hold off;

imshow(img2_col);
hold on;
perm = randperm(size(f_im2,2)) ;
sel = perm(1:100) ;
h1 = vl_plotframe(f_im2(:,sel)) ;
h2 = vl_plotframe(f_im2(:,sel)) ;
set(h1,'color','k','linewidth',3);
set(h2,'color','y','linewidth',2);
hold off;

end
```

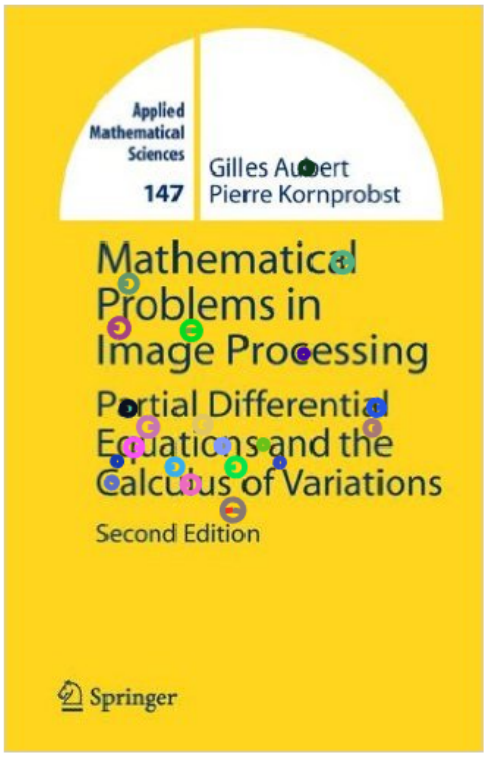
100 Randomly selected features from both book.jpg and findBook.jpg



(b) Matlab code for Sift feature matching algorithm

```
% top k correspondences
function [f_im1, f_im2, k_matches_im1, k_matches_im2, ksize] =
    matching_alg(ref, test, k, visualize)

% read images and grayscale
img1_col = imread(ref);
img1 = single(rgb2gray(img1_col)) ;
img2_col = imread(test);
img2 = single(rgb2gray(img2_col)) ;

%compute the SIFT frames (keypoints) and descriptors
[f_im1,d_im1] = vl_sift(img1) ;
[f_im2,d_im2] = vl_sift(img2) ;

[k_matches_im1, k_matches_im2, ksize] = match_k(d_im1, d_im2, k);
%ksize - limit on matches that could be made

if (visualize)
   colours = zeros(ksize, 3);
   for c = 1:ksize
    colours(c, :) = rand(1, 3);
   end
    % Plot images
    figure;
    imshow(img1_col);
    for i=1:ksize
        h1 = vl_plotframe(f_im1(:, k_matches_im1(i) ));
        set(h1,'color',colours(i, :),'linewidth',3) ;
    end
    figure;
    imshow(img2_col);
    for i=1:ksize
        h1 = vl_plotframe(f_im2(:, k_matches_im2(i) ));
        set(h1,'color',colours(i, :),'linewidth',3) ;
    end
end

end
```

Matched k=25 features for book.jpg and findBook.jpg



(c) Matlab code for solving for the Affine Transformation

```
% Solve for the affine transformation between features using top k
    correspondences
function out = affine_transf(ref_img, test_img, k)
% get top k correspondences from a2q2b;
[f_im1, f_im2, ind1, ind2, ks] = matching_alg(ref_img, test_img, k, 0);

% We assume we are using more than 3 keypoints,
% so we will use the Moore-Penrose psudeo inverse a = (P^T*P)^-1 * P^T*
    P'

P = zeros(2*ks, 6);
P_prime = zeros(2*ks, 1);

for i = 1:ks
    P(2*(i-1) + 1, :) = [f_im1(1, ind1(i)), f_im1(2, ind1(i)), 0, 0, 1,
        0];
    P(2*(i-1) + 2, :) = [0, 0, f_im1(1, ind1(i)), f_im1(2, ind1(i)), 0,
        1];

    P_prime(2*(i-1) + 1) = f_im2(1, ind2(i));
    P_prime(2*(i-1) + 2) = f_im2(2, ind2(i));
end

%Only use matrix operators (*, ^T, ^-1 (i.e. inv() in matlab)
%Do not use pinv, fsolve, maketform, mrdivide, mldivide,
%or the \ or / operators.

moore_penrose_inv = inv(P.'*P)*P.';
out = moore_penrose_inv*P_prime;

end
```

Results for various k:

```
>> affine_transf('book.jpg', 'findBook.jpg', 4)
ans =

    0.6987
    0.2390
   -0.2128
    0.6900
  675.2114
  500.5707

>> affine_transf('book.jpg', 'findBook.jpg', 5)
Warning: Converting non-floating point data to double.
ans =

    0.7323
```

```
    0.2835
   -0.2409
    0.6528
  659.9976
  513.2714

>> affine_transf('book.jpg', 'findBook.jpg', 10)
ans =

    0.7250
    0.2722
   -0.2421
    0.6546
  664.0128
  512.7061

>> affine_transf('book.jpg', 'findBook.jpg', 20)
ans =

    0.7252
    0.2651
   -0.2477
    0.6586
  665.9529
  512.1844

>> affine_transf('book.jpg', 'findBook.jpg', 50)
ans =

    0.7227
    0.2647
   -0.2473
    0.6455
  666.4668
  515.8869

>> affine_transf('book.jpg', 'findBook.jpg', 100)
ans =

    0.6845
    0.2317
   -0.2565
    0.6262
  681.5244
  522.6051
```

(d) Matlab code for visualizing the Affine Transformation

```
% Visualise affine
function out = visualize_affine(ref, test, k)

if nargin<3
    k = 25;
end;

%read the images in for visualization component
ref_img = imread(ref);
[r, c, ~] = size(ref_img);
[h, w, ~] = size(ref_img);
test_img = imread(test);

% fill in the rows of the P matrix where every 2 rows represent
% the x,y values of one of the four corners point of ref img
P = [1, 1, 0, 0, 1, 0; ...%top-left
    0, 0, 1, 1, 0, 1; ...
    1, r, 0, 0, 1, 0; ...%bottom-left
    0, 0, 1, r, 0, 1; ...

    c, 1, 0, 0, 1, 0; ... %top-right
    0, 0, c, 1, 0, 1; ...
    c, r, 0, 0, 1, 0; ... %bottom-right
    0, 0, c, r, 0, 1;];

affine_transf = affine_transf(ref, test, k);

points = P*affine_transf;

% Plot lines for dvd
imshow(test_img);
hold on;
line([points(1), points(3)],[points(2),points(4)],'Color','g', '
    Linewidth', 2);
line([points(5), points(7)],[points(6),points(8)],'Color','g', '
    Linewidth', 2);
line([points(1), points(5)],[points(2),points(6)],'Color','g', '
    Linewidth', 2);
line([points(3), points(7)],[points(4),points(8)],'Color','g', '
    Linewidth', 2);
hold off;

end
```

Mapped reference image book.jpg to test image findBook.jpg via Affine

(e) Helper function that does basic matching to establish feature correspondence (as in 2b)

```
function [k_matches_im1, k_matches_im2, size_match_array] = match_k(
    d_im1, d_im2, k)

% find euclidean distances between each pair of observations
% rows of each input matrix correspond to observations (transpose)
% (i,j) D entry equal to distance between observation
% i in X and observation j in Y
D = pdist2(d_im1.', d_im2.');
[n, ~] = size(D);

%thres ratio of nearest / 2nd nearest neighbour (should be < 0.8)
threshold = 0.8;

%find closest match (to one vector in test img)
%find second closest match
%match reliable if first dist much smaller than second

% calculate ratios and closest matches
%if A is a matrix, then sort(A,2) sorts the elements in each row
%sort(any i, j_fixed)
[D_rows_sorted, I] = sort(D, 2);
matches = zeros(1, n);

%index is feature in ref img, value is feature in second img
match_scores = zeros(1, n);

for i=1:n %each row - fix i (ref img), change j (second img)
    %closest/second_closest dist in that row
    ratio = D_rows_sorted(i,1) / D_rows_sorted(i,2);
    if ratio < threshold %check if match reliable, otherwise no match
        matches(i) = I(i, 1); %index of first closest
        match_scores(i) = ratio;
    else
        match_scores(i) = Inf;
        matches(i) = -1;
    end
end

% get top k correspondences
%choose matches with k smallest score (ratio) values
[~, score_index] = sort(match_scores);
k_matches_im1 = zeros(1, k);
k_matches_im2 = zeros(1, k);
size_match_array = 0;
for i = 1:k
    ith_smallest_index = score_index(i); %only 1 row
    if (isinf(match_scores(ith_smallest_index)))
        break;
    end
    k_matches_im1(i) = ith_smallest_index;
    k_matches_im2(i) = matches(ith_smallest_index);
```

```
    size_match_array = size_match_array + 1;
end

end
```

The algorithm for **SIFT with Color** *below* uses detected keypoints from SIFT on the grayscale image, however it computes SIFT feature descriptors on those same keypoints (as input to the *vl_sift* function) over the three channels of HSV color space, each of 128 dimension and stacks them to produce a 3x128 dimension feature descriptor vector. It then proceeds to match between the images like before using this new feature vector representation. The rationale behind choosing HSV space over RGB for incorporating color information is that it separates color from intensity and makes the algorithm somewhat less sensitive to lighting variations or shadows over same color regions as well as balancing informative intensity and color cues for better results. The H channel helps to extract regions of similar color-tone which are surrounded by different (all greater or all lower) color-tones, and the V channel should behave like the original grayscale image, giving us blobs which correspond to distinctive regions of intensity.

Main Function that embeds color information into the SIFT feature matching

```
%color_sift('colourTemplate.png', 'colourSearch.png', 22, 1);
function [f_im1, f_im2, k_matches_im1, k_matches_im2, ksize] =
    color_sift(ref, test, k, visualize)

img1_col_rgb = imread(ref);
img1 = single(rgb2gray(img1_col_rgb));
img2_col_rgb = imread(test);
img2 = single(rgb2gray(img2_col_rgb));

img1_col = rgb2hsv(img1_col_rgb);
img2_col = rgb2hsv(img2_col_rgb);

%compute the SIFT frames (keypoints) and descriptors
[f_im1,~] = vl_sift(img1) ;
[f_im2,~] = vl_sift(img2) ;

for i = 1:3
    [f1{i}, d1{i}] = vl_sift(single(img1_col(:,:,i)), 'frames', f_im1);
    [f2{i}, d2{i}] = vl_sift(single(img2_col(:,:,i)), 'frames', f_im2);
end

f_im1 = f1{1}; %f1{1}, f1{2}, f1{3} are identical features in same
    order
d_im1 = cat(1, d1{:}); %feature vector that is 3*128 in length
f_im2 = f2{1};
d_im2 = cat(1, d2{:});

[k_matches_im1, k_matches_im2, ksize] = match_k(d_im1, d_im2, k);
%ksize - limit on matches that could be made

if (visualize)
    colours = zeros(ksize, 3);
```

```
    for c = 1:ksize
        colours(c, :) = rand(1, 3);
    end
    % Plot images
    figure;
    imshow(img1_col_rgb);
    for i=1:ksize
        h1 = vl_plotframe(f_im1(:, k_matches_im1(i) ));
        set(h1,'color',colours(i, :),'linewidth',4) ;
    end
    figure;
    imshow(img2_col_rgb);
    for i=1:ksize
        h1 = vl_plotframe(f_im2(:, k_matches_im2(i) ));
        set(h1,'color',colours(i, :),'linewidth',4) ;
    end
end


end
```

Figure 2: Mapped reference image colourTemplate.png to test image colourSearch.png via Affine using new Color SIFT algorithm