Task 5: Image Stitching & RANSAC

Name: Laurent Gérin **Degree:** MASc **ID:** 20505439

Problem 1: Separable Filter

(a) What is a separability of the Gaussian kernel? When do we use this separability and why does this characteristic reduce the computing cost?

The separability of the Gaussian kernel means that the kernel can be represented as the multiplication of two vectors. For instance, a 3x3 Gaussian kernel, we solve the system of equations:

$$G = \begin{bmatrix} 0.0113 & 0.0838 & 0.0113 \\ 0.0838 & 0.6193 & 0.0838 \\ 0.0113 & 0.0838 & 0.0113 \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \begin{bmatrix} d & e & f \end{bmatrix} = \begin{bmatrix} ad & ae & af \\ bd & be & bf \\ cd & ce & cf \end{bmatrix}$$

First extracting the middle column vectors from both matrices, we get three relationships:

$$\begin{bmatrix} 0.0838 \\ 0.6193 \\ 0.0838 \end{bmatrix} = \begin{bmatrix} ae \\ be \\ ce \end{bmatrix}$$
$$\begin{aligned} \therefore\ a &= 0.0838/e, \\ b &= 0.6193/e, \\ c &= 0.0838/e \end{aligned}$$

Similarly with the middle row vectors:

$$\begin{bmatrix} 0.0838 & 0.6193 & 0.0838 \end{bmatrix} = \begin{bmatrix} bd & be & bf \end{bmatrix}$$
$$\begin{aligned} \therefore\ d &= 0.0838/b, \\ f &= 0.0838/b \end{aligned}$$

We can substitute b:

$$\begin{aligned} d &= 0.1353e, \\ f &= 0.1353e \end{aligned}$$

We are thus left with this:

$$G = \begin{bmatrix} 0.0838/e \\ 0.6193/e \\ 0.0838/e \end{bmatrix} \begin{bmatrix} 0.1353e & e & 0.1353e \end{bmatrix}$$

Removing e from both vectors, it is clear that e cancels out. Thus we obtain our final separated kernel:

$$G = \begin{bmatrix} 0.0838 \\ 0.6193 \\ 0.0838 \end{bmatrix} \begin{bmatrix} 0.1353 & 1 & 0.1353 \end{bmatrix}$$

This separability feature becomes quite useful when performing convolution to blur an image. We can define an original image I and a blurred image I_b, the result of convolution with the Gaussian kernel:

$$I_b = G * I$$

Let us now separate G, where G_1 is the vertical vector and G_2 is the horizontal vector:

$$I_b = (G_1 G_2) * I$$

It turns out that $G_1 G_2$ is the same as finding their 2D convolution, i.e. $G_1 * G_2$ (http://www.songho.ca/dsp/convolution/convolution2d_separable.html). We can also use the property of commutativity of convolution, and rewrite:

$$I_b = (G_1 * G_2) * I = G_1 * G_2 * I = G_1 * (G_2 * I)$$

This brings significant computational savings. If I is of size $m \times n$,and G, of size $\ell \times \ell$, regular 2D convolution would take $\ell^2 mn$ multiplications, and $(\ell^2 - 1)mn$ additions (since for each pixel we multiply each neighbours reached by G with their respective weight and then sum them). This means that larger Gaussian kernels will very rapidly increase the computation cost.

However, if we first convolve with G_2, which has size ℓ, convolution takes ℓmn multiplications and $(\ell - 1)mn$ additions. Then if we convolve this result with G_1, we again need ℓmn multiplications and $(\ell - 1)mn$ additions. Thus we get a total of $2lmn$ multiplications and $2(l-1)mn$ additions.

In Big-O notation, the first method, naive 2D convolution, has complexity $O(\ell^2 mn)$, while the second method, using the separated kernel, has complexity $O(lmn)$. This is huge! Now the computational time only increases linearly with the size of the kernel instead of quadratically.

(b) What is the Laplacian of Gaussian (LoG)? When do we use LoG?

The Laplacian of Gaussian is a kernel which allows two operations at the same time. Essentially, we take the 2nd derivative of our image (Laplacian), and blur it out (Gaussian). We blur it out because 2nd derivatives are extremely sensitive to noise. The LoG allows combining the two operations at once and thus saves computational requirements.

The LoG becomes useful for corner and blob detection - since it has a round shape and strongly defined edges, local extrema after the convolution will indicate where strong blobs were detected.

(c) Is LoG separable?

The LoG is not separable. As an example, if we look at a 5x5 LoG kernel:

$$LoG = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 1 & 2 & -16 & 2 & 1 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

We can see that the rows are clearly not linear combinations of each other, as they are in the Gaussian kernel. Ie, there is no way to multiply a row-vector by scalars a and b that will give rows 1 and 2 in the matrix.

(d) What is a difference of the Gaussian (DoG)? When do we use it and what is DoG advantageous compared to LoG?

The difference of Gaussian is very similar to LoG, with a slightly less "pointy" blob. Essentially, we take the Gaussian blur of an image with a certain σ, and then we change σ, take another Gaussian blur, and find the difference between the two images. The advantage of DoG is that we can use the Gaussian kernel's separability to accelerate processing, and also that since we are taking the Gaussian with two different σ values, we can quickly look at different blob sizes in a "chain" manner, further reducing the computational time compared to the LoG.b

Problem 2: Least Squares

(a) Explain the approach 1 and approach 2 for least squares line fitting in your words. Please refer to the course slide and tutorials.

Both approaches are mathematically the same, the only difference being that approach 2 represents the least-square fitting method in matrix form. In both cases, we are minimizing the error between the fitted model and the data. First, we define an error term:

$$E = \sum_{i=1}^{n} (y_i - mx_i - b)^2$$

or in matrix form (approach 2):

$$E = ||Y - XB||^2$$

This error term is the difference between the measured y values and their predicted value by the model. Squaring this difference, we eliminate negative values and thus avoid the case where negative error terms cancel positive error terms. We also put more emphasis on values which have larger error terms, ie which are further away from our fitted line.

In approach 2, this is defined as the Euclidian norm, i.e. the "distance" from the origin in n dimensions. Since the distance is calculated as such:

$$||Y - XB|| = \sqrt{(y_1 - (mx_1 + b))^2 + (y_2 - (mx_2 + b))^2 + \dots}$$

We square this to remove the square root, and thus we can see how this is exactly the same as the error term in approach 1.

After defining our error term, we take the partial derivatives with respect to m and b, or in approach 2, B. Setting these equal to zero and solving for m and b or B, we will find the model which minimizes our error term:

$$m = \frac{\sum x_i y_i - (1/n) \sum x_i \sum y_i}{\sum x_i^2 - (1/n)(\sum x_i)^2}$$

$$b = \frac{\sum y_i \sum x_i^2 - \sum x_i \sum x_i y_i}{n \sum x_i^2 - (\sum x_i)^2}$$

$$B = (X^T X)^{-1} X^T Y$$

A response yi is measured from a linear system when an input is xi. The measurement data is provided (`prob2_data1.mat`). A model of your system can be approximated as yi = mxi + b. Please find m and b using the following methods.

(b) Least squares (approach 1)

(c) Least squares (approach 2)

We find with both methods that m is 1.6540 and b is 2.3197.

```
load('task_files/prob2_data1.mat')

%approach 1:
n = length(x);
SX = sum(x);
SY = sum(y);
SXY = sum(x .* y);
SX2 = sum(x.^2);
S2X = SX^2;

m = (SXY - 1/n * SX * SY)/(SX2 - 1/n * S2X)
b = (SY * SX2 - SX * SXY)/(n * SX2 - S2X)

%approach 2:
X = [x; ones(1,n)]';
B = (X' * X) \ X' * y'
```

Here is another measurement data (`prob2_data2.mat`). Please find m and b using the following methods.

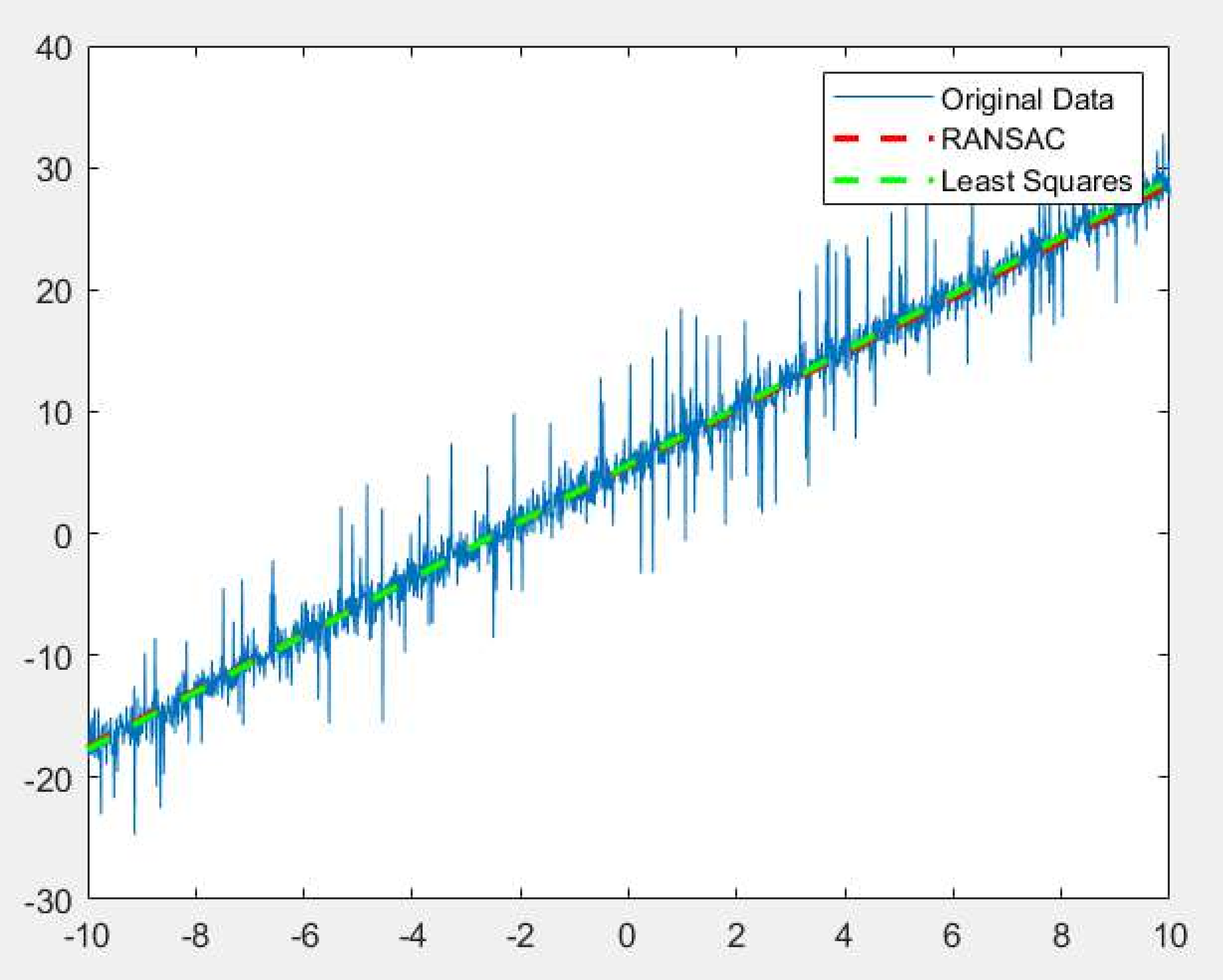
(d) Least squares (either approach 1 or approach 2)

We find m = 2.3348 and b = 5.6458.

```
load('task_files/prob2_data1.mat')
X = [x; ones(1,length(x))]';
B = (X' * X) \ X' * y'
```

(e) Use of RANSAC. You need to have your own RANSAC implementation without using existing functions in MATLAB (Do not use `ransac` or any other relevant functions)

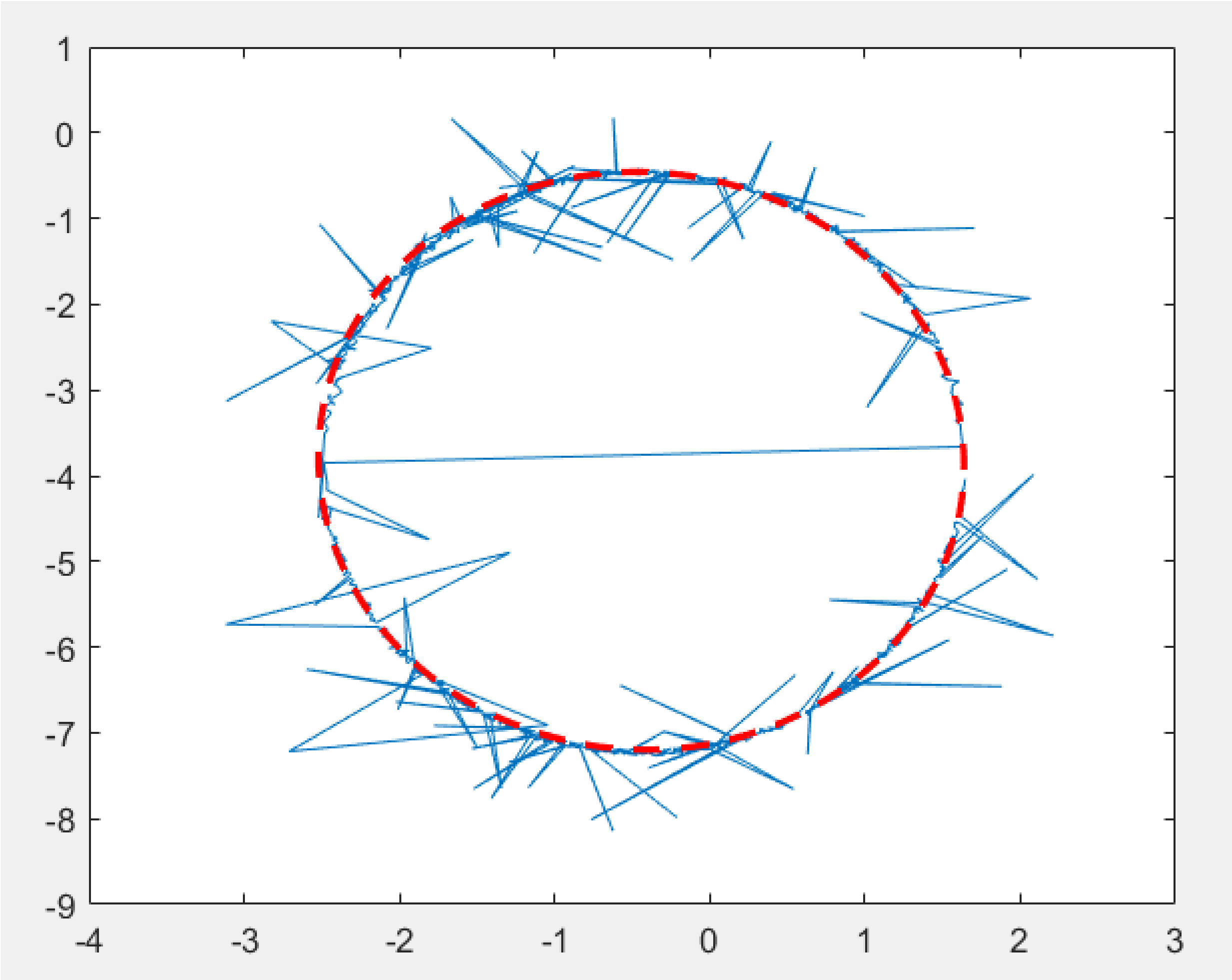
We find m = 2.3074 and b = 5.5433. Comparison of the two fitted lines shows this:



Problem 3: Fitting using RANSAC

(a) Fit an ellipse to the given data (`prob3_ellipse.mat`) using RANSAC

The ellipse was successfully fitted using RANSAC as shown here.

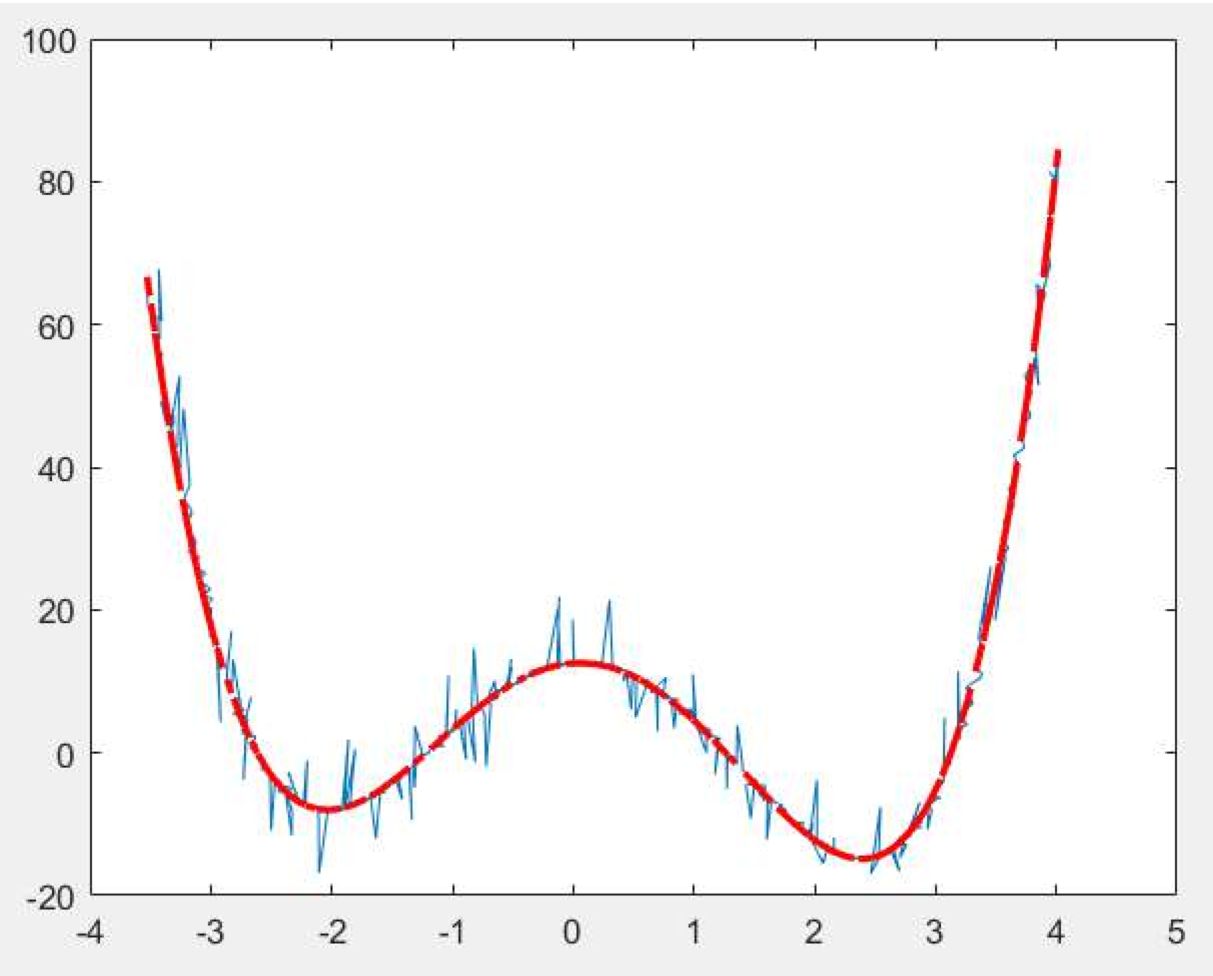



```
%Fit an ellipse
%Using general quadratic curve ax^2+ bxy + cy^2 + dx + ey + f = 0

load('task_files/prob3_ellipse.mat')
plot(x,y);

n_tries = 10000;
thres = 0.1;

combinations = zeros(n_tries,7);   %a, b, c, d, e, f, count

for i = 1:n_tries
    [xi, idx] = datasample(x,5);
    yi = y(idx);

    bigMatrix = zeros(5,6);
    for j = 1:5
        bigMatrix(j,:) = [xi(j)^2, xi(j) * yi(j), yi(j)^2,xi(j),yi(j),1];
    end
    params = null(bigMatrix);   %a, b, c, d, e, f
    if size(params,2) >1
        params = params(:,1);
    end
    %we have an implicit equation which makes it difficult to find the vertical distance.  Instead we use the
```

```
    %value obtained by putting x and y into the ellipse equation, which should be zero.
    count = sum( abs(params(1)*x.^2 + params(2)*x.*y + params(3)*y.^2 + params(4)*x + params(5)*y + params(6)) <= thres);
    combinations(i,:)=[params',count];
end

[maxcount, maxidx] = max(combinations(:,7));
a = combinations(maxidx,1);
b = combinations(maxidx,2);
c = combinations(maxidx,3);
d = combinations(maxidx,4);
e = combinations(maxidx,5);
f = combinations(maxidx,6);
hold on
fimplicit(@(x,y) a*x.^2 + b*x.*y + c*y.^2 + d*x + e*y + f,'r--','LineWidth',2);
```

(b) Fit a fourth degree polynomials to the given data (`prob3_polynomial.mat`) using RANSAC

The polynomial is successfully fitted as shown:




```
%Fit 4th order polynomial
%y = ax^4 + bx^3 + cx^2 + dx + e

load('task_files/prob3_polynomial.mat')
```

```
plot(x,y);

n_tries = 10000;
thres = 0.1;

combinations = zeros(n_tries,6);   %a, b, c, d, e, count

for i = 1:n_tries
    [xi, idx] = datasample(x,5);
    yi = y(idx);

    bigMatrix = zeros(5,5);
    for j = 1:5
        bigMatrix(j,:) = [xi(j)^4, xi(j)^3, xi(j)^2, xi(j), 1];
    end
    params = bigMatrix\yi';   %a, b, c, d, e

    count = sum( abs(y - (params(1)*x.^4 + params(2)*x.^3 + params(3)*x.^2 + params(4)*x + params(5))) <= thres);
    combinations(i,:)=[params',count];
end

[maxcount, maxidx] = max(combinations(:,6));
a = combinations(maxidx,1);
b = combinations(maxidx,2);
c = combinations(maxidx,3);
d = combinations(maxidx,4);
e = combinations(maxidx,5);

hold on
y_fit = a * x.^4 + b*x.^3 + c*x.^2 + d*x + e;
plot(x,y_fit, 'r--','LineWidth',2)
```

Problem 4: Improved 3D Planar Measurement Tool

(a) Build your own measurement tool and evaluate your measurement using the images provided (see the folder of `prob4_img`). You may need to estimate homography based on SIFT feature matching. The exact size of the booklet is 24 cm x 31.5 cm and use `cover.jpg` to solve this problem.

The table below shows each image and the measurement taken to be from the 0 mark to the 10 cm mark on the ruler in the pictures. Note that inaccuracies also come from selection of the points on the picture.

Image	Measurement (should be 10 cm)
	10.11
	9.64

Image	Measurement (should be 10 cm)
	9.81
	9.50

Image	Measurement (should be 10 cm)
	10.11
	10.15

```
cover = imread('task_files/cover.jpg');
cover = imresize(cover, [1000, NaN]);   %Resize to have 1000 rows
coverBW = single(rgb2gray(cover));


%fileName = char(strcat(num2str(numpictures,'task_files/%03.f'),".JPG"));
fileName = 'task_files/prob4_img/006.JPG';
img = imread(fileName);
```

```
img = imresize(img, [1000, NaN]);   %Resize to have 1000 rows
imgBW = single(rgb2gray(img));

%detect cover features
[f_cover,d_cover] = vl_sift(coverBW);
if(0)
    imshow(cover);
    hold on
    h1 = vl_plotframe(f_cover) ;
    h2 = vl_plotframe(f_cover) ;
    set(h1,'color','k','linewidth',3) ;
    set(h2,'color','y','linewidth',2) ;
end

%detect image features
[f_img,d_img] = vl_sift(imgBW);
if(0)
    %show these image features
    figure
    imshow(img);
    hold on
    h1 = vl_plotframe(f_img) ;
    h2 = vl_plotframe(f_img) ;
    set(h1,'color','k','linewidth',3) ;
    set(h2,'color','y','linewidth',2) ;
end
%Match features
[matches, scores] = vl_ubcmatch(d_cover, d_img) ;

numMatches = length(scores);

%Get matching X and Y values
Pts_cover = [f_cover(1:2,matches(1,:));ones(1,numMatches)];   %  add ones at the end to get homogeneous coordinates
Pts_img = [f_img(1:2,matches(2,:));ones(1,numMatches)];



%Now we estimate the homography
n_tries = 1000;
thres = 2;  %pixels: distance to be considered outlier
%tform = estimateGeometricTransform(Pts_img',Pts_cover','projective','MaxNumTrials',n_tries,'MaxDistance',5);

score = zeros(n_tries,1);
H = zeros(3,3,n_tries);
for i = 1:n_tries
    x = zeros(1,4);
    y = zeros(1,4);
    xprime = zeros(1,4);
    yprime = zeros(1,4);
    %Find 4 points at random to estimate homography
```

```
    for j = 1:4
        index = randi(numMatches);
        x(j) = Pts_cover(1,index);
        y(j) = Pts_cover(2,index);
        xprime(j) = Pts_img(1,index);
        yprime(j) = Pts_img(2,index);
    end

    %Find homography matrix for these 4 points
    bigMatrix=zeros(8,9);
    for j=1:2:7
        k = (j+1)/2;
        bigMatrix(j:j+1,:) = [x(k) y(k) 1 0 0 0 -x(k)*xprime(k) -xprime(k)*y(k) -xprime(k);...
            0 0 0 x(k) y(k) 1 -x(k)*yprime(k) -yprime(k)*y(k) -yprime(k)];
    end
    h = null(bigMatrix);
    h = h(:,1);
    H(:,:,i) = reshape(h,3,3);

    Pts_coverH = (Pts_cover' * H(:,:,i))'; %Transform cover points to img homography
    dx = Pts_coverH(1,:)./Pts_coverH(3,:)-Pts_img(1,:);
    dy = Pts_coverH(2,:)./Pts_coverH(3,:)-Pts_img(2,:);
    score(i) = sum(dx.^2+dy.^2 <= thres^2);
end

[best, bestidx] = max(score);
H = H(:,:,bestidx);

%Rectify and show image
if(0)
    imgH = imwarp(img,projective2d(inv(H)));
    figure
    imshow(imgH)
    hold on
end

% %Bring cover to image plane
% coverH = imwarp(cover,projective2d(H));
% figure
% imshow(coverH)

%Now we know that the cover has dimensions 24 cm x 31.5 cm.  We can use H
%to find the scaling factor to be used for our measurements.

dimX = 24;      %cm
dimY = 31.5;    %cm

figure
imshow(img)
hold on

%Find scale of cover picture
```

```
scX = 24/size(cover,2);      %cm/pixel
scY = 31.5/size(cover,1);    %cm/pixel

sc = mean([scX,scY]);

%First line set
[xline1, yline1] = getline();    %Get first polyline - select vertices and hit enter once all are chosen
plot(xline1,yline1,'b','LineWidth',2)     %Show chosen lines

ptsLine1 = [xline1';yline1';ones(1,length(xline1))];

ptsLineRect1 = H' \ ptsLine1;
ptsLineRect1(1:2,:) = ptsLineRect1(1:2,:)./ptsLineRect1(3,:);

dist = zeros(length(xline1)-1,1);
for i = 1:(length(xline1)-1)
    dist(i) = sqrt((ptsLineRect1(1,i) - (ptsLineRect1(1,i+1)))^2 + (ptsLineRect1(2,i) - (ptsLineRect1(2,i+1)))^2) * sc;
    formatSpec = 'Length of line %1u is %2.3f cm\n';
    fprintf(formatSpec,i,dist(i))
end
```

(b) Prepare your own calibration paper and take photos similar to the ones in (a). Then, evaluate your tool.

This was done using a fridge magnet from UW's Science Society... Dimensions were measured as **8.91 x 5.08 cm**. The table below shows a few pictures and measurements on the ruler, again taken at 10 cm. In general, the measured dimensions are not as good as in the previous picture. It's probably because the magnet is much smaller, thus we get a higher relative error. Additionally, this doesn't correct for lens distortion - this also adds additional error, particularly in the first picture where we can see some lines aren't parallel.

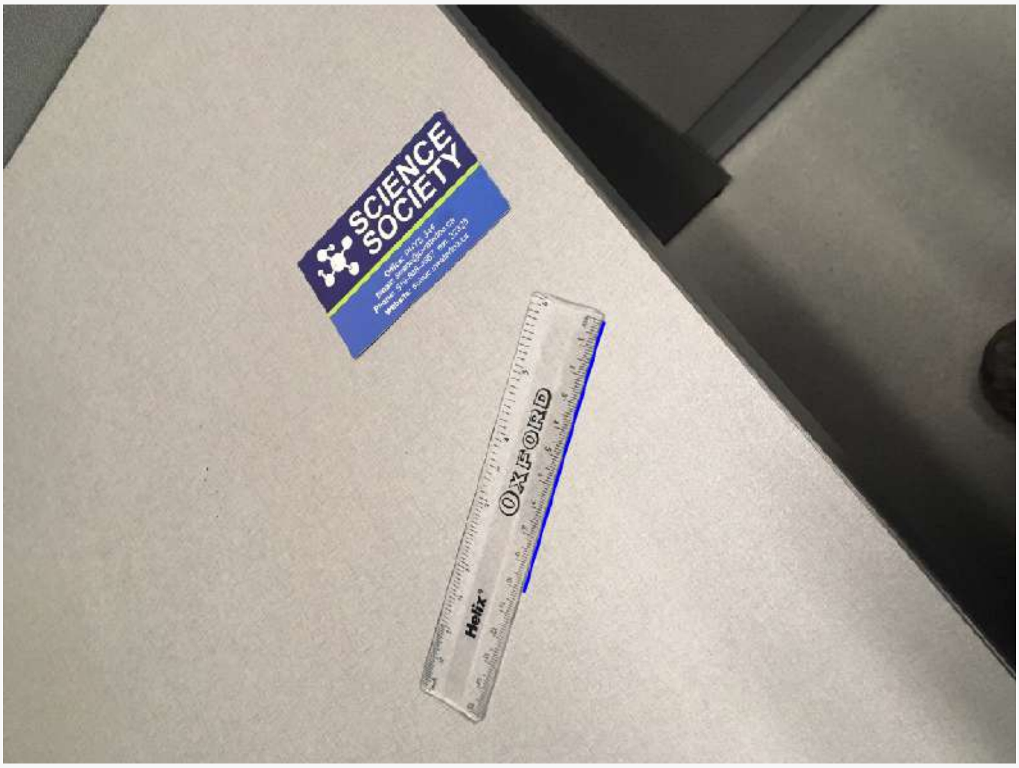
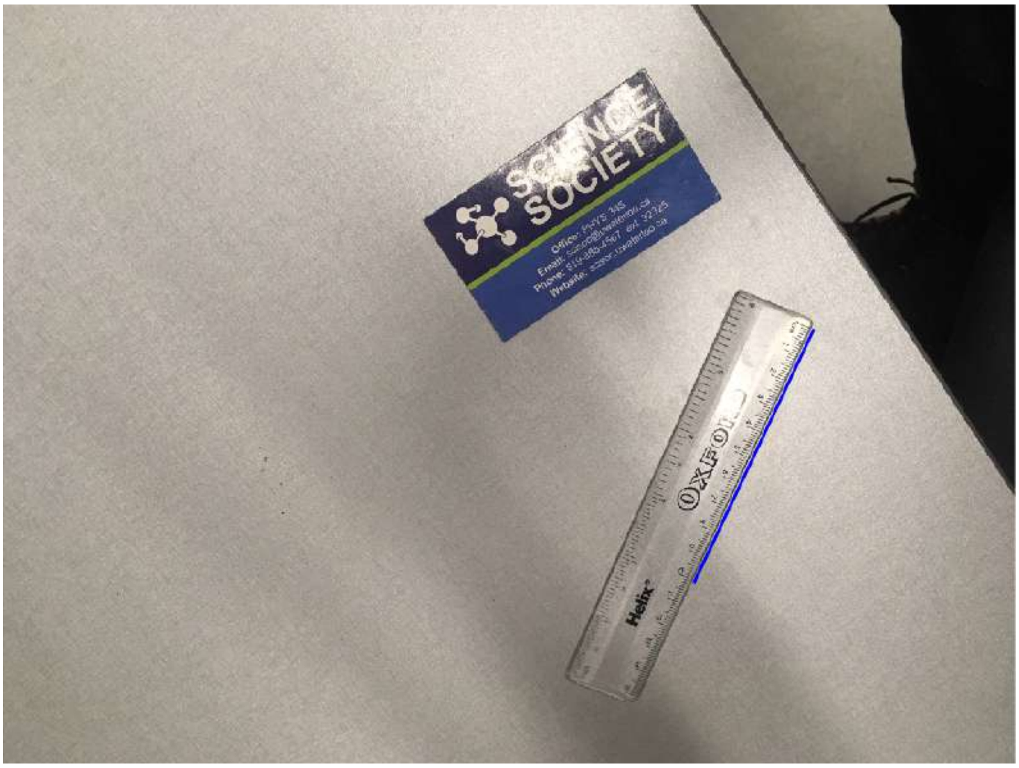
Image	Measurement (should be 10 cm)
	11.912
	9.419

Image	Measurement (should be 10 cm)
	9.978

(c) Compare measurements using this new tool and the one developed in Task 3.

In general, the measurements were at similar levels of accuracy. This is mostly because there is large imprecision in the selection of points with the mouse cursor. The additional distortion from camera parameters and the lens adds additional error as well. Also, since in this case we are doing feature-based matching for points, we can't exploit the full size of our "marker". In Task 3, we were using all 4 corners of the sheet, which made for larger distances which could "swallow" some of the error, whereas since features may be very close to each other, there is potential for small errors to be magnified once the homography is applied to the whole image.

Problem 5: Book Classification using SIFT

You are going to categorize books on images. Here are the input images and expected outcomes.

Test image	Result
	
	

Your code needs to automatically compute the outlines (boundary) of each book and its identity (book name). There are 17 images (see the folder of `prob5_img`) and in each image, four books are placed on a desk. Your code should not fail to identify a book or estimate its boundary more than 5 books among all books (85 = 17 images x 5 books). Note that you should not do hard coding as well.

This was done using SIFT. Images, with all books identified, are shown in the table below:

Original	Classified
	
	
	
	

<table>
<tr><th>Original</th><th>Classified</th></tr>
<tr><td></td><td>
</td></tr>
<tr><td></td><td>
</td></tr>
<tr><td></td><td>
</td></tr>
<tr><td></td><td>
</td></tr>
</table>

Original	Classified
	
	
	
	

Original	Classified
	
	
	
	

Original	Classified
	

```
display = 0;
covers{1} = imread('covers_prob5/deep_goodfellow.jpg');
covers{2} = imread('covers_prob5/handbook_bovik.jpg');
covers{3} = imread('covers_prob5/pattern_bishop.jpg');
covers{4} = imread('covers_prob5/modern_mikhail.jpg');

names = {'deep-goodfellow', 'handbook-bovik', 'pattern-bishop', 'modern-mikhail'};

%detect cover features
for i = 1:length(covers)
    coversBW{i} = single(rgb2gray(covers{i}));
    [f_covers{i},d_covers{i}] = vl_sift(coversBW{i});
    if(display)
        figure
        imshow(covers{i});
        hold on
        h1 = vl_plotframe(f_covers{i}) ;
        h2 = vl_plotframe(f_covers{i}) ;
        set(h1,'color','k','linewidth',3) ;
        set(h2,'color','y','linewidth',2) ;
    end
end

for nn = 1:17
fileName = char(strcat(num2str(nn,'task_files/prob5_img/%03.f'),".JPG"));
img = imread(fileName);
img = imresize(img, [1000, NaN]);   %Resize to have 1000 rows
imgBW = single(rgb2gray(img));

%detect image features
[f_img,d_img] = vl_sift(imgBW);
if(display)
    %show these image features
    figure
    imshow(img);
    hold on
    h1 = vl_plotframe(f_img) ;
    h2 = vl_plotframe(f_img) ;
```

```
    set(h1,'color','k','linewidth',3) ;
    set(h2,'color','y','linewidth',2) ;
end
%Match features
for n = 1:length(covers)
    [matches, scores] = vl_ubcmatch(d_covers{n}, d_img);
    numMatches = length(scores);
    %Get matching X and Y values
    Pts_cover = [f_covers{n}(1:2,matches(1,:));ones(1,numMatches)];   %  add ones at the end to get homogeneous coordinates
    Pts_img = [f_img(1:2,matches(2,:));ones(1,numMatches)];

    %Now we estimate the homography
    n_tries = 1000;
    thres = 3;  %pixels: distance to be considered outlier

    score = zeros(n_tries,1);
    H = zeros(3,3,n_tries);
    for i = 1:n_tries
        x = zeros(1,4);
        y = zeros(1,4);
        xprime = zeros(1,4);
        yprime = zeros(1,4);
        %Find 4 points at random to estimate homography
        for j = 1:4
            index = randi(numMatches);
            x(j) = Pts_cover(1,index);
            y(j) = Pts_cover(2,index);
            xprime(j) = Pts_img(1,index);
            yprime(j) = Pts_img(2,index);
        end

        %Find homography matrix for these 4 points
        bigMatrix=zeros(8,9);
        for j=1:2:7
            k = (j+1)/2;
            bigMatrix(j:j+1,:) = [x(k) y(k) 1 0 0 0 -x(k)*xprime(k) -xprime(k)*y(k) -xprime(k);...
                0 0 0 x(k) y(k) 1 -x(k)*yprime(k) -yprime(k)*y(k) -yprime(k)];
        end
        h = null(bigMatrix);
        h = h(:,1);
        H(:,:,i) = reshape(h,3,3);

        Pts_coverH = (Pts_cover' * H(:,:,i))'; %Transform feature points to img homography
        dx =  Pts_coverH(1,:)./ Pts_coverH(3,:)-Pts_img(1,:);
        dy =  Pts_coverH(2,:)./ Pts_coverH(3,:)-Pts_img(2,:);
        score(i) = sum(dx.^2+dy.^2 <= thres^2);
    end
    [best, bestidx] = max(score);
    H = H(:,:,bestidx);
```

```
    cornerArray = [0, 0, 1; size(covers{k},2),0, 1; size(covers{k},2),size(covers{n},1),1; 0, size(covers{n},1),1]; %corner locations in homog coords
    cornerArrayH = (cornerArray * H);
    cornerArrayH = cornerArrayH ./ cornerArrayH(:,3);
    cover{n} = cornerArrayH(:,1:2);
    cover{n} = [cornerArrayH(1,1:2), cornerArrayH(2,1:2), cornerArrayH(3,1:2), cornerArrayH(4,1:2) ];
    xyTB(n,:) = [ mean([min(cornerArrayH(:,1)), max(cornerArrayH(:,1))]),mean([min(cornerArrayH(:,2)), max(cornerArrayH(:,2))]) ];
end
imgAnnotated = img;

for i = 1:length(cover)
    imgAnnotated = insertShape(imgAnnotated,'FilledPolygon',cover{i},'Opacity',0.5);
    imgAnnotated = insertText(imgAnnotated,xyTB(i,:),names{i},'FontSize',24,'AnchorPoint','Center','BoxColor','red');
end

if(display)
    figure
    imshow(imgAnnotated);
end
fileNameEx = char(strcat(num2str(nn,'prob5_classified/%03.f'),".jpg"));
imwrite(imgAnnotated,fileNameEx)
end
```

Problem 6: Image Stitching

First, the MATLAB tutorial images as stitched together:

And here is my friend doing a pretty nice ski jump! Overall the panorama is a great quality I would say, which I think is because the trees provide a lot of great features to use for matching.

```
display = 0; %Show debug images

% Load images into an array of images imArray
buildingImageFiles = fullfile('img_prob6','*.JPG');
listImages = dir(buildingImageFiles);

for i=1:length(listImages)
    imArray{i} = imread(fullfile(listImages(i).folder, listImages(i).name));
    imArray{i} = imresize(imArray{i},[1000 NaN]);
end

if(display)
    montage(imArray)
end

%Total number of images
numImages = length(imArray);

% Read the first image from the image set.
I = imArray{1};

% Find features for I(1) - our "master image"
```

```
grayImage = single(rgb2gray(I));
[f, d] = vl_sift(grayImage);

% Initialize variable to hold image sizes.
imageSize = zeros(numImages,2);
imageSize(1,:) = size(grayImage);

arrayH = zeros(3,3,numImages);  %This is the array of transformations to get the nth image on the same homography as the first image. Ie, each layer is the multiplication of each previous image
arrayH(:,:,1) = eye(3,3);       %Make the first transformation matrix the identity matrix (the original image doesn't change)

% Iterate over remaining image pairs
for n = 2:numImages

    % Store points and features for I(n-1).
    fPrevious = f;
    dPrevious = d;

    % Read I(n).
    I = imArray{n};

    % Convert image to grayscale.
    grayImage = single(rgb2gray(I));

    % Save image size.
    imageSize(n,:) = size(grayImage);

    % Detect features for I(n).
    [f, d] = vl_sift(grayImage);

    %Find matches between I(n) and I(n-1).
    [matches, scores] = vl_ubcmatch(dPrevious, d);
    numMatches = length(scores);

    %Get matching X and Y values
    ptsPrevious = [fPrevious(1:2,matches(1,:));ones(1,numMatches)];   %  add ones at the end to get homogeneous coordinates
    ptsI = [f(1:2,matches(2,:));ones(1,numMatches)];

    %Now we estimate the homography
    n_tries = 50000;
    thres = 20;  %pixels: distance to be considered outlier

    score = zeros(n_tries,1);
    H = zeros(3,3,n_tries);
    for i = 1:n_tries
        x = zeros(1,4);
        y = zeros(1,4);
        xprime = zeros(1,4);
        yprime = zeros(1,4);
        %Find 4 points at random to estimate homography
```

```
        for j = 1:4
            index = randi(numMatches);
            x(j) = ptsI(1,index);
            y(j) = ptsI(2,index);
            xprime(j) = ptsPrevious(1,index);
            yprime(j) = ptsPrevious(2,index);
        end

        %Find homography matrix for these 4 points
        bigMatrix=zeros(8,9);
        for j=1:2:7
            k = (j+1)/2;
            bigMatrix(j:j+1,:) = [x(k) y(k) 1 0 0 0 -x(k)*xprime(k) -xprime(k)*y(k) -
xprime(k);...
                0 0 0 x(k) y(k) 1 -x(k)*yprime(k) -yprime(k)*y(k) -yprime(k)];
        end
        h = null(bigMatrix);
        h = h(:,1);
        H(:,:,i) = reshape(h,3,3);

        ptsIH = (ptsI' * H(:,:,i))'; %Transform feature points to img homography
        dx =  ptsIH(1,:)./ ptsIH(3,:)-ptsPrevious(1,:);
        dy =  ptsIH(2,:)./ ptsIH(3,:)-ptsPrevious(2,:);
        okArray{i} = dx.^2+dy.^2 <= thres^2;
        score(i) = sum(dx.^2+dy.^2 <= thres^2);
    end
    [best, bestidx] = max(score);
    H = H(:,:,bestidx);
    arrayH(:,:,n) = H * arrayH(:,:,n-1);

        % -------------------------------------------------------------------
        %                                                        Show matches
        % -------------------------------------------------------------------
    if(display)
        dh1 = max(size(I,1)-size(imArray{n-1},1),0) ;
        dh2 = max(size(imArray{n-1},1)-size(I,1),0) ;

        ok = okArray{bestidx};
        figure
        subplot(2,1,1) ;
        imagesc([padarray(imArray{n-1},dh1,'post') padarray(I,dh2,'post')]) ;
        o = size(imArray{n-1},2) ;
        line([fPrevious(1,matches(1,:));f(1,matches(2,:))+o], ...
         [fPrevious(2,matches(1,:));f(2,matches(2,:))]) ;
        title(sprintf('%d tentative matches', numMatches)) ;
        axis image off ;

        subplot(2,1,2) ;
        imagesc([padarray(imArray{n-1},dh1,'post') padarray(I,dh2,'post')]) ;
        o = size(imArray{n-1},2) ;
        line([fPrevious(1,matches(1,ok));f(1,matches(2,ok))+o], ...
        [fPrevious(2,matches(1,ok));f(2,matches(2,ok))]) ;
        title(sprintf('%d (%.2f%%) inliner matches out of %d', ...
```

```
              sum(ok), ...
              100*sum(ok)/numMatches, ...
              numMatches)) ;
        axis image off ;
        drawnow ;
    end
end

%Now we wish to re-transform our panorama such that the center image is the
%"normal" one.  This minimizes distortion in edge cases.

%Compute the output limits for each transform
for i = 1:numImages
    [xlim(i,:), ylim(i,:)] = outputLimits(projective2d(arrayH(:,:,i)), [1 imageSize(i,2)], [1 imageSize(i,1)]);
end

%find the average x limits of each picture. this is essentially the
%centroid of each transformed picture.  Assuming these are side by side
%pictures

avgXLim = mean(xlim,2);
[~, idx] = sort(avgXLim);
centerIdx = floor((numImages+1)/2);
centerImageIdx = idx(centerIdx);

Hinv = inv(arrayH(:,:,centerImageIdx));

for i = 1:numImages
    arrayH(:,:,i) = arrayH(:,:,i) * Hinv;
end


%Stitch our images together for the panorama

%Find the maximum output limits for the panorma to compute size
for i = 1:numImages
    [xlim(i,:), ylim(i,:)] = outputLimits(projective2d(arrayH(:,:,i)), [1 imageSize(i,2)], [1 imageSize(i,1)]);
end

maxImageSize = max(imageSize);

% Find the minimum and maximum output limits
xMin = min([1; xlim(:)]);
xMax = max([maxImageSize(2); xlim(:)]);

yMin = min([1; ylim(:)]);
yMax = max([maxImageSize(1); ylim(:)]);

% Width and height of panorama.
width  = round(xMax - xMin);
height = round(yMax - yMin);
```

```
% Initialize the "empty" panorama.
panorama = zeros([height width 3], 'like', I);

%Now we add our pictures!

blender = vision.AlphaBlender('Operation', 'Binary mask', ...
    'MaskSource', 'Input port');

% Create a 2-D spatial reference object defining the size of the panorama.
xLimits = [xMin xMax];
yLimits = [yMin yMax];
panoramaView = imref2d([height width], xLimits, yLimits);

% Create the panorama.
for i = [1 2 4 5 3]

    I = imArray{i};

    % Transform I into the panorama.
    warpedImage = imwarp(I, projective2d(arrayH(:,:,i)), 'OutputView', panoramaView);

    %Generate a binary mask.
    mask = imwarp(true(size(I,1),size(I,2)), projective2d(arrayH(:,:,i)), 'OutputView', panoramaView);

    %Overlay the warpedImage onto the panorama.
    panorama = step(blender, panorama, warpedImage, mask);
    if(0)
    figure, imshow(warpedImage)
    end
end

figure
imshow(panorama)
```