VISUAL UNDERSTANDING– CM50248

Coursework Report

Unit Leader:

Dr. Christian Richardt

Degree Programme:

MSc Digital Entertainment
University of Bath

Table of Contents

1. Introduction

We were given the task to tackle a set of computer vision problems, each with its own method and algorithmic approaches to be solved. An overview of the problems to be thoroughly examined throughout this report is shown:

- Convolution
    - Basic Convolution
    - Convolution with border handling
    - Image filtering
        - Kernels for horizontal, vertical and diagonal image gradient computation
        - Unsharp Masking
        - Computation of Gaussian kernels
    - Exploiting the convolution theorem using Fast Fourier Transform (FFT)
- Features
    - Canny edge detection
    - Harris corner detection
    - Difference-of-Gaussian interest point detection
- Matching
    - Brute force matching
    - Homography estimation
    - Visualising the homography
    - Homography estimation using RANSAC
- Camera calibration
    - Estimating intrinsic parameters
        - Lens distortion
        - Intrinsic Camera Matrix
- Fundamental matrix and Essential matrix
    - 8-Point algorithm using correspondences for fundamental matrix
    - RANSAC for fundamental matrix estimation
    - Essential matrix decomposition
        - Extrinsic calibration between two cameras
- Triangulation of 3D points
    - Linear Triangulation
    - 3D scene reconstruction

Each problem will correspond to a new section and the process of formulating and computing the solution will be described in a detailed manner. The code accompanying this report is in MATLAB format and given certain tasks, the algorithms derived will be compared to MATLAB's built in function to emphasize the correctness of the desired output.

2. Convolution

Convolution is a very popular procedure used in a lot of areas of computer science, and it is a form of linear filtering. It is frequently used in computer vision as it is able to produce visual effects on images such as smoothing, blurring, sharpening etc, (Richardt and Hall, 2017). Algorithmically speaking, convolution is the weighting of the multiplication of a neighbourhood of image pixel values, i.e. the pixel's value, with a pre-formed kernel, or commonly known as mask. A kernel, and the values it contains as well as its dimensions' sizes, directly affects the result of convolution, as to which filtering effect will be applied on the given image. Other than visual effects, convolution can provide necessary data from images such as their horizontal, vertical and diagonal gradients which can be further examined to obtain additional information from the images. Details on convolution can be seen in the following subsections.

2.1 Basic Convolution

The predetermined kernels are of arbitrary size but must be square, i.e. both the height and width must be of the same size, and it is preferred to have a centre, meaning that each side must be of an odd number length. Kernels of size 3x3 are most commonly used in practise. Sliding the kernel over the image and computing the corresponding pixel value of the resulting image is convolution. The formula on which convolution is based upon is:

$$(K * I)\ (x, y) = \sum_i \sum_j K(i, j)\ I(x - i, y - j)$$

Where K is the kernel, I is the image and (x,y) is the current pixel value. Even though the symbol of convolution resembles that of multiplication, it does not operate the same way. In simple terms, convolution on a specific pixel is the weighted sum of itself with the rest of the neighbourhood pixels, and the weighting coefficients supplied by the actual kernel. The sliding of window over the image is an iterative and inefficient algorithmic brute-force approach. Given that the kernel can be of any size, some of the values required to produce the filtering are missing, i.e. when the window is on the corner of an image. This is where a difference is distinguished between the basic and extended convolution, which will be explained shortly. To account for those missing values the image is padded with layers of zeros depending on the kernel size. For a kernel of size 3x3, the image is padded with a single layer of zeros, for a kernel of size 5x5, two layers of zeros are used for padding and so on. Furthermore, to preserve colour intensity and brightness of a single pixel value even after the filtering procedure, the kernel values are normalised, i.e. the values are divided by the total sum of the kernel values. Before explaining the algorithm steps for convolution, transforming the image into grayscale is required so that it is not necessary to handle three dimensions, third being colour (RGB), and allowing for 2-dimensional calculations. The algorithm for basic convolution is given:

Basic Convolution(I, K):

1. Pad the image, I, with layers of zeros depending on the kernel's, K, size.
2. For each row i in K:
    a. For each column j in K:
        i. For each row x in I:
            1. For each column y in I:
                a. F(x,y) = $\sum_i \sum_j K(i, j) * I(x - i, y - j)$, multiply the kernel value with the image neighbourhood pixels to derive F(x,y), where F is the filtered image.
3. Recover the image, F', in its original size, by removing the previously padded layers from F.
4. Return the filtered image F'.

After performing the basic convolution described above, the logical step is to check whether the results are identical to MATLAB's built-in corresponding function, **conv2(I, K, 'same')**. An error metric is used, in this case the Sum of Squared Differences (SSD), to compare the performance of each one of the implementations. Let us first define a kernel:

$$K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

This is called the identity kernel, since it does not impose any effect on the convolved image, given that is composed of ones throughout. The results can be seen in Figure 2.1.

Notice in the figure, the black outermost pixel layers on all the filtered image, including those produce using MATLAB's conv2. This is the result from filtering the image after padding it with layers of zeros. The resulting SSD of the implemented function's and MATLAB's conv2 result, was really small, specifically $2.8332e^{-27}$, which is translated by MATLAB as 0.

Figure 2.1 (left) The original image (up) and its grayscale counterpart (down); (right) Filtered image with identity kernel (up) using the described function and MATLAB's conv2 built in function corresponding results (down).

2.2 Extended Convolution

Convolution performed with padded layers of zeros across all sides of the image, results to the black lines surrounding the image as seen in Figure 2.1. To avoid this, we can take a step further and instead of padding the image with zeros, we can replicate the outer most pixels of the image and pad those values. The rest of the algorithmic steps are exactly the same as those performed in Basic Convolution. The output of Extended Convolution can be seen in Figure 2.2, using the

Figure 2.2 (left) Original image (up) and grayscale (down); (right) Filtered image using described function (up) and filtered image using MATLAB's imfilter (down)

identity kernel once again. MATLAB's built-in function corresponding to this extension of convolution is **imfilter(I, K, 'replicate', 'conv')**. Notice that this time, there are no black outer pixel layers, since there were not there prior to the convolution. Once again, the SSD came out to be a very small number, close to 0.

2.3 Image Filtering

As mentioned earlier, there are a lot of effects that can be achieved when passing an image under convolution, and all them are based on a specific kernel. Firstly, the most commonly used kernels for blurring an image are derived from a Gaussian distribution, and therefore called Gaussian kernels, or smoothing operators, (Gonzalez and Woods, 2002). Such a kernel is used to define a probability distribution to show noise with its values obtained from a Gaussian curve. Furthermore, it is considered a low-pass filter as it removes high frequencies from the image pixels, i.e. details. To build a Gaussian kernel, there are two hyperparameters to be provided; the side of the square kernel and the standard deviation, σ, both of which drive the intensity of the blurriness effect. The blur produced is called the Gaussian Blur and the probability distribution is defined as:

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2}\exp(-(x^2+y^2)/2\sigma^2)$$

Where the x and y are coordinates of the matrix to be simulated. The values passed for x and y are actually the normalised coordinates of the Gaussian kernel, meaning that the middle column will have an x-coordinate of 0 while the middle row will have a y-coordinate of 0. Figure 2.3 is a code snippet showing the MATLAB commands used to compute a Gaussian kernel from scratch.

```
% Used to compute gaussian kernels based on the side and standard
% deviation given as input.
function gaus = compute_gaussian(side,sigma)
    gaus = ones([side side]);
    for m = -floor(size(gaus,1)/2):floor(size(gaus,1)/2)
        for n = -floor(size(gaus,1)/2):floor(size(gaus,1)/2)
            gaus(m+ceil(size(gaus,1)/2),n+ceil(size(gaus,2)/2)) = (1/(2*pi*sigma^2))*exp(-(m^2 + n^2)/(2*(sigma^2)));
        end
    end
    %normalize
    gaus = (1/(sum(sum(abs(gaus))))).*gaus;
end
```

The effect of the change in standard deviation when computing the Gaussian kernel can be seen in Figure 2.3.

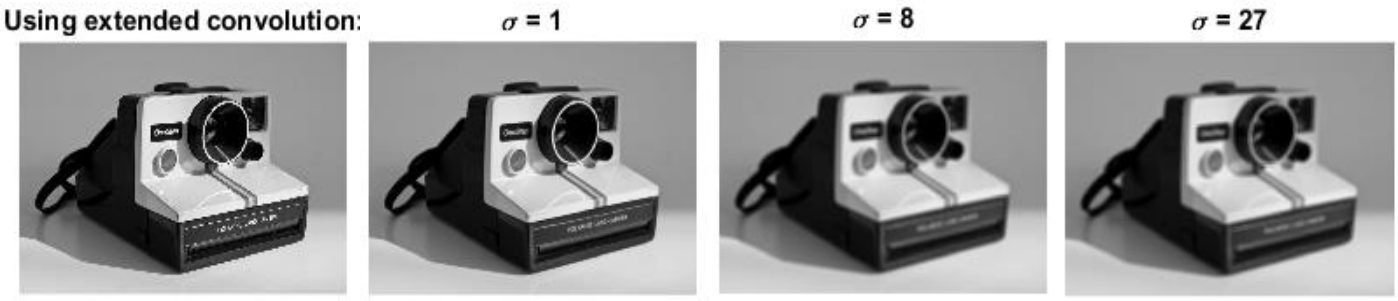


Figure 2.3 Image filtering using the extended convolution function based on Gaussian kernels with standard deviation as shown above the corresponding images. All convolutions were produced using Gaussian kernels of size 11x11.

Additionally, there are kernels especially constructed to derive the image gradients in all directions individually. These kernels are built depending on the gradient direction a user wants to visualise. For example, the vertical image gradient kernel contains a column of zeros right in the middle, whereas the diagonal gradient kernel has a diagonal line containing zeros. The kernels used to derive the results in Figure 2.5 are shown in Figure 2.4.

Vertical Gradient

1	0	-1
1	0	-1
1	0	-1

Horizontal Gradient

1	1	1
0	0	0
-1	-1	-1

Diagonal Gradient

1	1	0
1	0	-1
0	-1	-1

Unsharp Masking

0	-1	0
-1	5	-1
0	-1	0

Figure 2.4 Gradient and unsharp masking kernels.

Another effect achievable from convolution is unsharp masking, which is an image sharpening technique. This technique takes its name from the steps required to produce it. Firstly, the image is blurred, while then adding the noisy blurred image to the original image. Thus, it allows for higher frequencies to be amplified resulting to a sharper, more detailed version of the original image. The kernel is shown on the right.

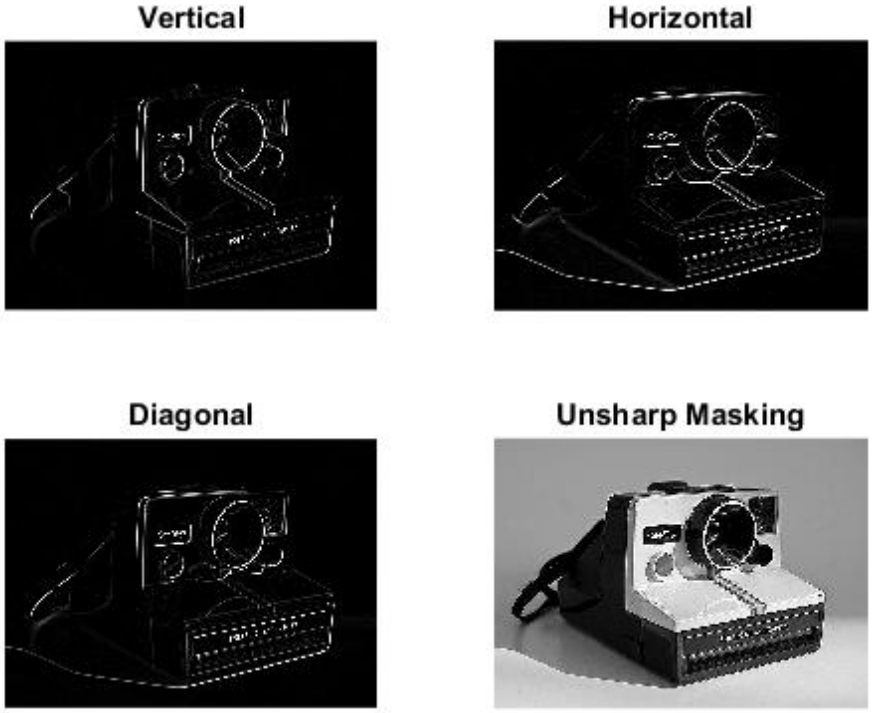


Figure 2. 5 Image gradients and unsharp masking filtering, through convolution. Extended convolution was used to create these images.

2.4 Exploiting the convolution theorem using Fast Fourier Transform (FFT)

Convolution, as explained above, is a timely procedure and inefficient when iterating along all image pixels and kernel values. An optimised alternative is using Fast Fourier Transform (FFT), while exploiting the convolution theorem. The convolution theorem simply states that convolution in the time domain translates to multiplication in the frequency domain and vice versa, with the proof and explanation can be seen in (Weisstein, 2018), (Richardt and Hall, 2017) and (Gonzalez and Woods, 2002). Therefore, passing both the image and the kernel into the frequency domain using FFT, the filtered image can be retracted by taking the pointwise product of the two matrices and applying the inverse FFT (iFFT), denoted as:

$$K * I = iFFT\{\, FFT\{K\}\;\; FFT\{I\}\,\}$$

Where the left side is the convolution of the image, I, using kernel K, while the right side denotes the inverse FFT of the product of the two FFTs. The image needs to be padded with replicated layers of

pixel values depending both on the size of the image and the kernel, while centred to get the optimal results. The function used to achieve translation the inverse FFT and FFT are MATLAB's built-in functions. The main difference here, is that it is not required to iteratively slide the window over each image pixel like with standard convolution, but instead the whole of the image and kernel are just pointwise multiplied to get the filtered image. To prove time efficiency of convolution using FFT, the time taken for each algorithm was recorder in multiple runs, each run using a Gaussian kernel of increased size with a constant $\sigma = 2.4$. In fact, FFT performs faster when the size of the matrices is larger. Figure 2.6 shows the results from each convolution technique and Figure 2.7 shows the average time taken for each algorithm when increasing the size of the kernel.

Figure 2.6 Convolution using FFT compared to standard convolution using the extended convolution.

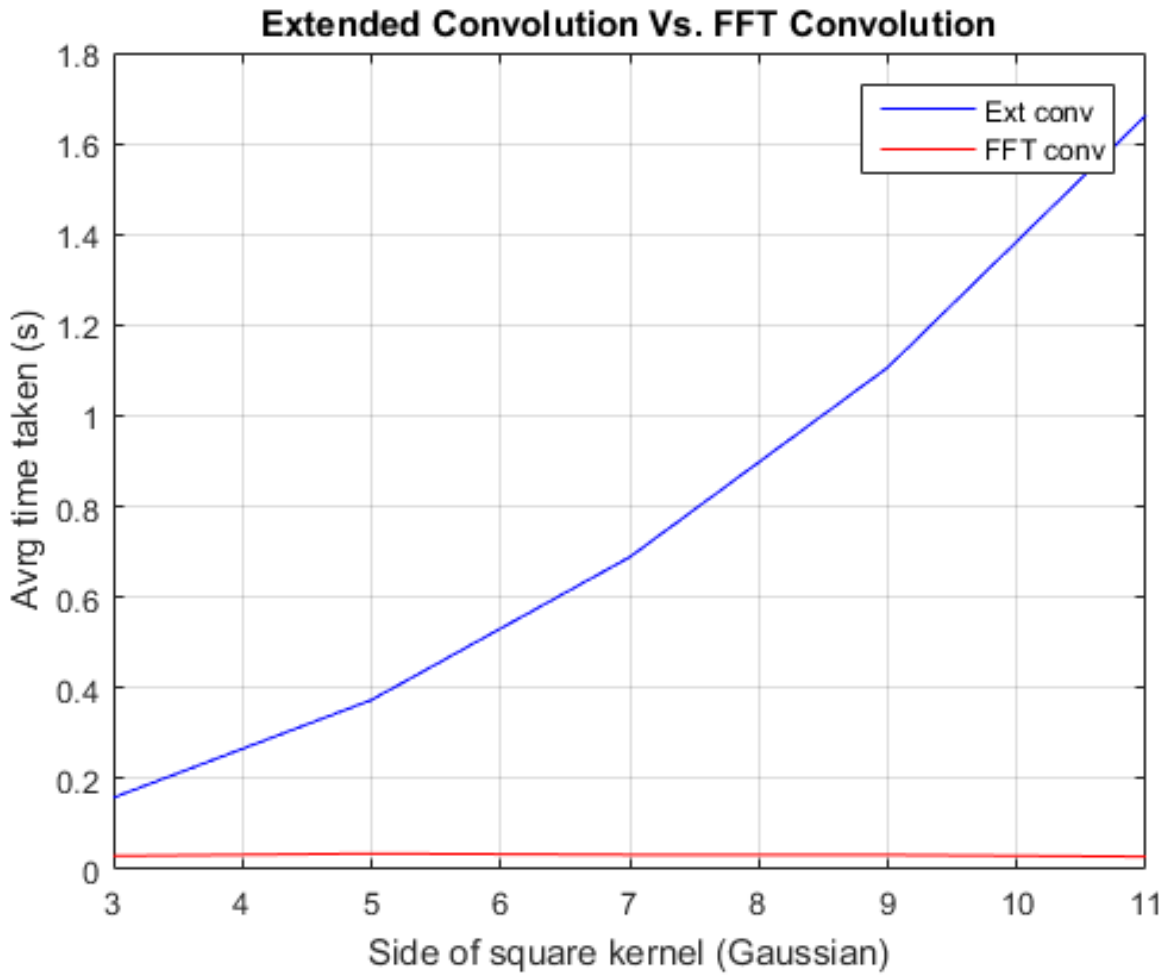


Figure 2.7 Average time taken versus the side of the kernel for the two convolution methods. Notice how the time taken by extended convolution increases exponentially when the size of the kernel increases, while convolution with FFT stays constant regardless of the kernel size.

3. Features

As humans, we have the ability to evaluate images, using our sight, based on our limitless brain capacity given that we have seen the objects in the images before. Computers do not possess this ability, at least not in such a volume. In order for a computer to distinguish on what object or topography an image portrays, it has to process the image in such a way that it resolves so called features. Features in images could be anything from corners to colours to edges. With the aid of these features the computer can analyse them and given further processing recognise objects in the images just like humans do. In this section, we will discuss and analyse three different feature extraction algorithms and show their corresponding results given different hyperparameter initialisations.

3.1 Canny Edges

Feature extraction, as mentioned previously, can be used to extract edges from images. Edges are the outlines of objects, their shape, which can then be further processed so that a computer can narrow down the vast list of possible objects in the scene to distinguish what that object is. Moreover, they can be defined as sudden changes in image brightness along a direction. Figure 2.5 above, shows that with the extraction of gradients from the image using the referred kernels, edges are formed. These edges are not actual edges but image gradients, (Richardt and Hall, 2017), which the computer cannot interpret. What a computer can understand, though, are binary maps representing edges, two-dimensional arrays of 1s and 0s, and that is exactly what the Canny Edge detector is able to do, (Canny, 1986). Canny derived a series of steps that eventually turn grayscale images to computer interpretable edges. Before moving onto listing the steps required, let us first define specific jargon involved in this process.

First, let us understand what exactly is encoded by the gradient of an image. From Figure 2.5, we can see the outline of the image emerging from a black background. The black background suggests that there is not change, based on the orientation of the gradient required, in that direction, while the edges show that there is a certain change in gradient, again based on the gradient direction. The resulting edges hold valuable information about the image. Specifically, they encode the gradient orientation at that point in the image as well as its magnitude. These can be obtained using the following equations, given that I_X and I_Y are the derivative images, in the x and y direction respectively:

$$r = \sqrt{(I_x^2 + I_y^2)}$$

$$\theta = \arctan(\frac{I_y}{I_x})$$

Where $\mathbf{r}$ denotes the magnitude of the gradient and θ denotes the orientation of the gradient in degrees.

Next, is Non-Maximum Suppression (NMS). NMS, as the name suggests, is a brute-force, iterative technique able to find local extrema in image neighbourhoods or, in the case of edge detection, magnitude neighbourhoods. For Canny edge detection, the process provided by NMS, is called edge thinning, as it suppresses a neighbourhood of weak magnitude pixels to a single strong magnitude pixel in that neighbourhood, which will allow us to visualise a cleaner and a visually simpler representation of edges. Lastly, hysteresis thresholding should be discussed. Hysteresis thresholding is a commonly used thresholding mechanism in computer vision as well as other areas of computer science. The main difference between hysteresis thresholding and absolute thresholding, is that hysteresis thresholding uses an upper and a lower threshold, whereas absolute thresholding only uses a single threshold. This allows for accounting pixel values between the two thresholds as well as below or above them, which

might be the deciding factors when connecting two edges to produce a single continuous one. Hysteresis thresholding is also an iterative approach.

With everything explained, the algorithm is comprised of the following steps, as seen in the lecture slides:

Canny Edges (I, k-size, σ): (Inputs are the image I, the side of the Gaussian kernel k-size, and standard deviation σ)

1. Get the grayscale representation of the input image, I.
2. Perform convolution on the grayscale image using a Gaussian kernel to obtain a blurred image, thus removing the noise.
3. Perform convolution on the blurred images to obtain image spatial gradients, I_x and I_y, using the vertical and horizontal kernels mentioned in 2.3.
4. Find the magnitude and orientation of the gradients using the formulas above to obtain, I_r and I_θ respectively.
5. Add 360 degrees to each pixel in the gradient orientation image, I_θ, thus making all orientations positive, and proceed by performing binning. This will result to generalising each pixel's orientation in bins of 45 degrees multiples.
6. Produce an initial binary map by performing edge thinning using NMS on the magnitude image, I_r, by iteratively checking whether each pixel has the maximum magnitude from its neighbouring pixels depending on the orientation bin that pixel belongs to. For example, if that pixel belongs to the orientation bin of 0 degrees then check the pixel above and below it. If that pixel has the maximum magnitude in that direction then set it to 1, otherwise 0.
7. Multiply the binary map from step 6, with the original magnitude image, I_r, to recover the magnitudes of the strong pixels.
8. Finally, perform hysteresis thresholding on the resulting image form step 7, by setting both a low and a high threshold. This will produce a second and final binary map, the desired edges. For hysteresis thresholding, we check whether a pixel has a magnitude greater than the high threshold and set it to be equal to 1. If it is lower than the low threshold we set it to 0. These are the 2 main cases. Now if the magnitude of the current pixel is greater than the low threshold but lower than the high threshold, we check all of its neighbours, and if they are all above the high threshold, we set that pixel to 1.
9. Repeat step 8 until no further change to the binary map is observed.
10. Out put the resulting binary map, the edges.

The hyperparameters to be pre-determined in this algorithm are: 1) size of Gaussian kernel, 2) standard deviation for Gaussian distribution, 3) the low and high thresholds for hysteresis thresholding. Each produce different results when varied. As for the size and standard deviation for the formulation of the Gaussian kernel, as these are increased, they produce a smoother, more blurred image. Our goal when filtering the image using this kernel is to remove the noise so as to avoid unwanted edges, but doing this in a large degree will remove valuable information from the image, entailing the loss of desired edges and creation of weird artefacts. The same concept stands for the thresholds of hysteresis thresholding. Having thresholds with large values means that certain magnitudes will not be accounted as edges, and having low values means that pixels that would otherwise not be edges will also be outputted. To find the best values for all these hyperparameters, which may vary depending on the image at hand, trial and error is the logical approach. For simplicity purposes, the thresholds for hysteresis were kept constant at high = 0.15 and low = 0.4 * high, (40% of the high threshold). Figure

3.1 shows some example runs of the Canny edge detector using different kernel sizes and standard deviations.

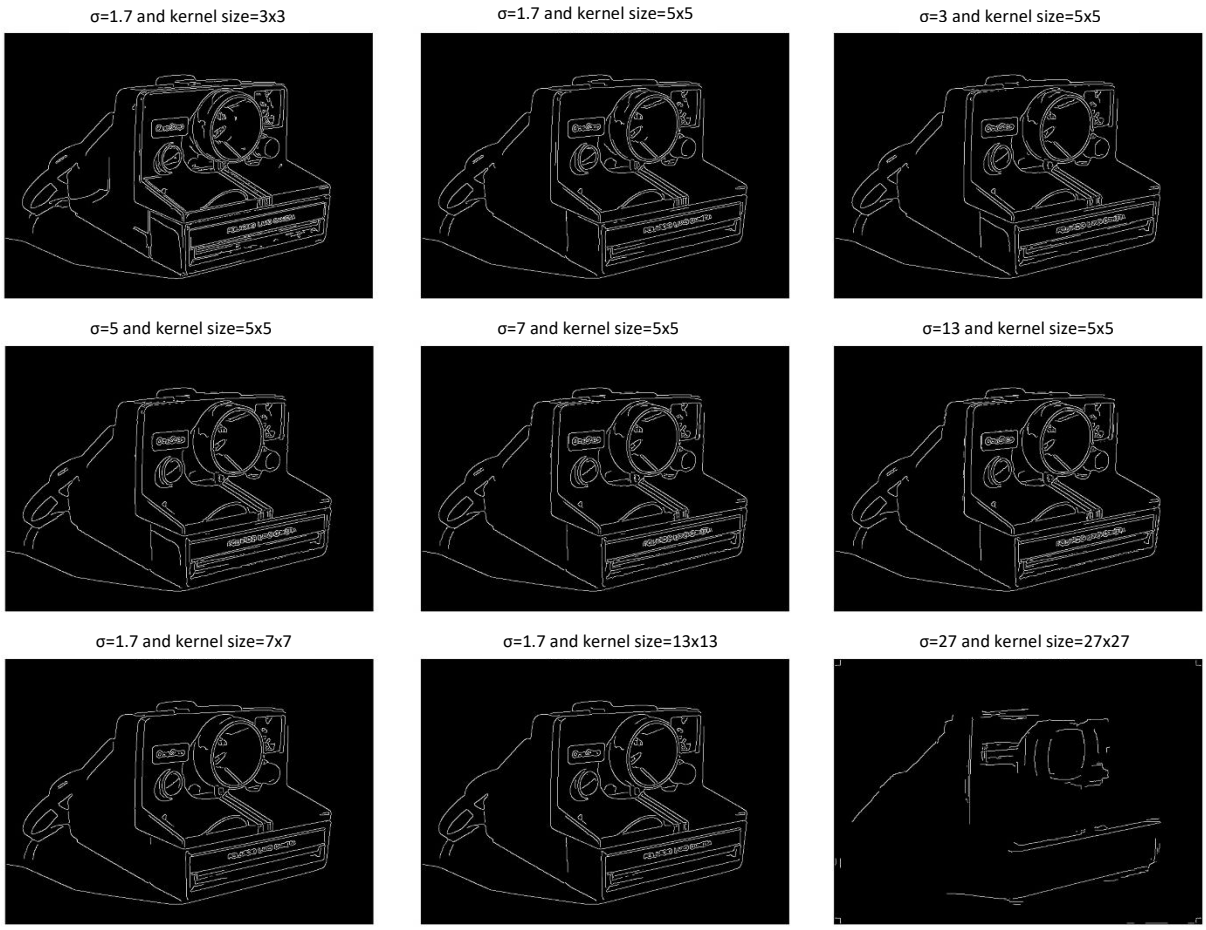


Figure 3.1 Results of the Canny edge detector given the parameters shown above each image.

3.2 Harris Corners detection

Features are not only edges. Actually, edges are even less important when compared to features such as corners. Corners are commonly referred to as interest points due to their ability to be invariant to rotation, translation as well as illumination of images. Consequently, an algorithm for detecting corners in images was produced by Harris et al. in 1988, which is still widely used as one of the most important feature extracting algorithms in computer vision, (Harris and Stephens, 1988). The algorithm for the Harris Corner detector has certain steps that are identical to the Canny edge detector, but generally it depends greatly on a matrix composed of terms which are derived from the image's spatial derivatives. This matrix, also known as structure tensor, is as follows:

$$A = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

Matrix A can then be decomposed to produce a product given in terms of an orthonormal matrix U and a diagonal matrix L:

$$A = ULU^{-1}$$

Matrices U and L are the eigenvectors and eigenvalues of A which will be used in order to come up with a cornerness score, or commonly referred to as Harris response, for each pixel in an image. This ranking

score is the decision factor when looking for corners in an image, so if they pass a certain threshold, they are considered as corners. To obtain a value for the Harris response for each pixel we can define it this way:

$$M_c = L_1 L_2 - k\,(L_1 + L_2)^2$$

Where L_1 and L_2 are the eigenvalues of A and k is a hyperparameter which is commonly set to a value between k = 0.04 and k = 0.15. Eigenvalue decomposition is a timely and inefficient procedure and therefore the alternative is to replace the terms containing the eigenvalues with the following:

$$L_1 L_2 = \det(A) = I_x^2\, I_y^2 - I_{xy}^2$$

$$L_1 + L_2 = trace(A) = I_x^2 + I_y^2$$

Therefore, concluding to a Harris response formula:

$$M_c = \det(A) - k\,\big(\,trace(A)\big)^2$$

Using the above derivation, the algorithm for the Harris corner detection can be broken down to simple steps as shown:

Harris Corners (I, k-size, σ): (Inputs are the image I, the side of the Gaussian kernel k-size, and standard deviation σ)

1. Get the grayscale representation of the input image, I.
2. Perform convolution on the grayscale image using a Gaussian kernel to obtain a blurred image, thus removing the noise.
3. Perform convolution on the blurred images using the same Gaussian kernel to obtain image spatial gradients, I_x and I_y, using the vertical and horizontal kernels mentioned in 2.3.
4. Use the image derivatives to compute the structure tensor A as described above.
5. For every pixel in the image compute the cornerness score M_c given the last formula in the derivation above.
6. Perform NMS to find the local maxima for cornerness score, by iteratively going through every pixel, that has a score above a certain threshold, and comparing it with its 8 surrounding neighbours, setting the pixel with the maximum score as 1 producing a binary map.
7. Superimpose the original image with the binary map to show corners.

Just like the Canny edge detector, there are certain hyperparameters to pre-determine before running the algorithm. These include: 1) size of Gaussian kernel, 2) standard deviation for Gaussian distribution, 3) the value of k used to compute the cornerness score. As mentioned above, the value of k is kept constant at k = 0.08 since the algorithm performed best at that setting. In the case of the Gaussian kernel parameters, they are once again used to formulate a Gaussian kernel which will be used to smooth the image thus removing the noise. To determine these parameters, the same intuition is seen as when deciding these for the Canny edge detector. Over-smoothing the image will result to loss of valuable corners, and optimal parameters would result to most corners detected. Below are some of the results explored with their corresponding parameters.

Figure 3.2 Harris corner detector with the parameters shown above each image. As the size of the kernel increases, less corners (green dots) are detected.

3.3 Difference-of-Gaussian (DoG): Interest point detection

The difference of Gaussian blob detection is a procedure which is iterative in nature. It requires the construction of a scale space for differently sized images of the same view. The scale space represents the image in different degrees of blurriness by increasing the standard deviation of the Gaussian kernel to be used with convolution, which in turn will aid at the retrieval of extrema points. This is done several times, each time resizing the image by a ratio, in this case by ½, and doing the same procedure all over again. The blurred images are used to construct a DoG tree which is comprised of the difference between the blurred images, i.e. the more blurred image is subtracted from the less blurred image of the original image. The DoG, Γ, of an image can be expressed as:

Figure 3.3 Grayscale image of the coins before DoG is performed.

$$\Gamma_{\sigma_1,\sigma_2} = \left(I * K_{\sigma_1} \right) - \left(I * K_{\sigma_2} \right)$$

Where $\sigma_2 > \sigma_1$ and K_{σ_i} are the gaussian kernels with parameter input the σ, and I is the grayscale image to be convolved. This can be seen in the figures below.

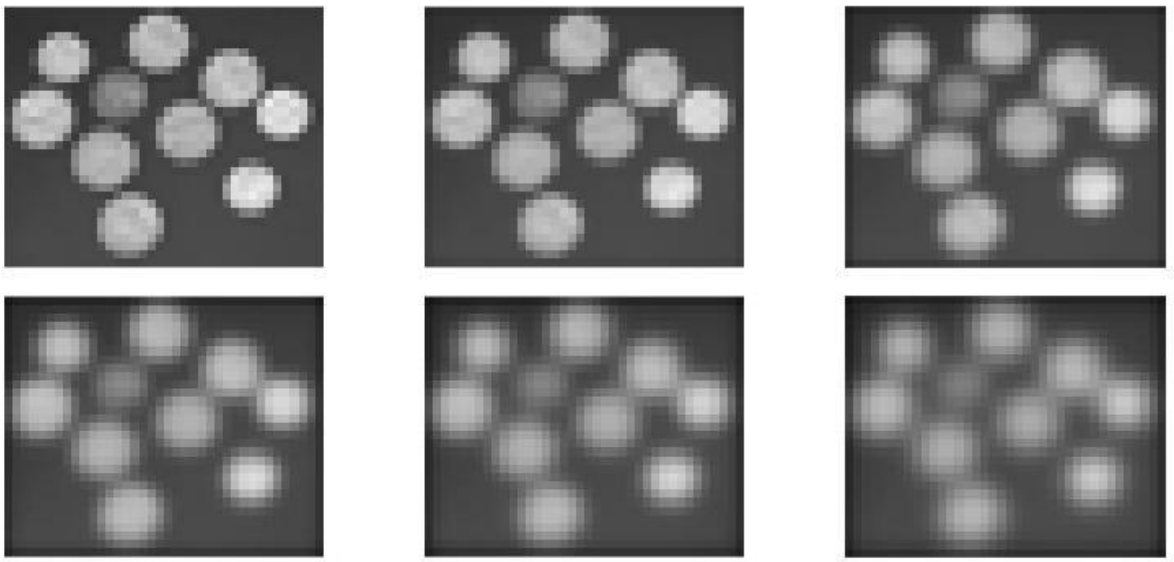


Figure 3.4 From top left to bottom right, these are the blurred with increasing σ before their subtraction to obtain the DoG. These have the size of the original image in resolution.

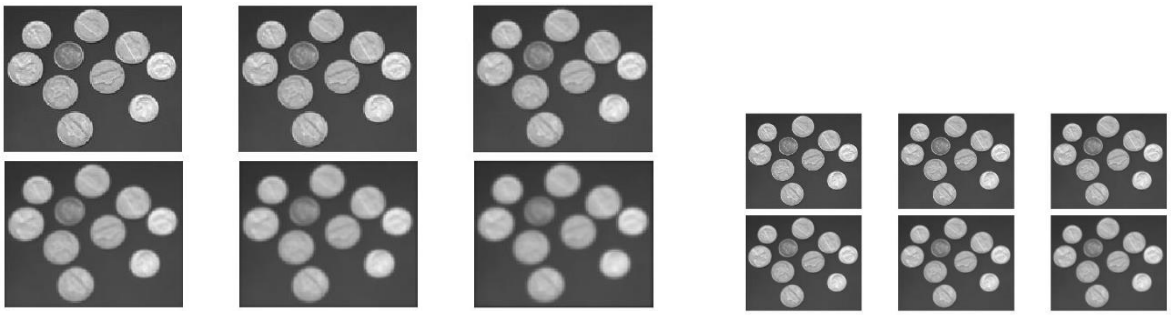


Figure 3.5 Resized and blurred again.

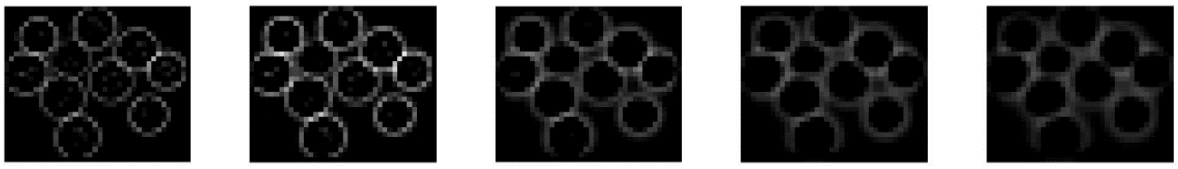


Figure 3.6 An example of the DoG from the subtraction of the blurred images in one run. Notice that there are 5 images instead of 6 as before since we only get 5 differences.

This subtraction allows to preserve the range of useful spatial frequencies between the blurred images, and the resulting images is then checked for extrema points with the value. Once resizing the image, the same procedure is done given the new image dimensions. Therefore, doing this will return the extrema point from all sized images and these will correspond to the scale invariant features to be detected. The points are compared with all 27 of its neighbouring pixels when placing the DoG images on top of each other, as explained in (Richardt and Hall, 2017). The final procedure of this approach is to actually find the centre and radius of the blobs to be drawn which was neglected in this implementation. The following figures show the extrema detected in the DoG images.

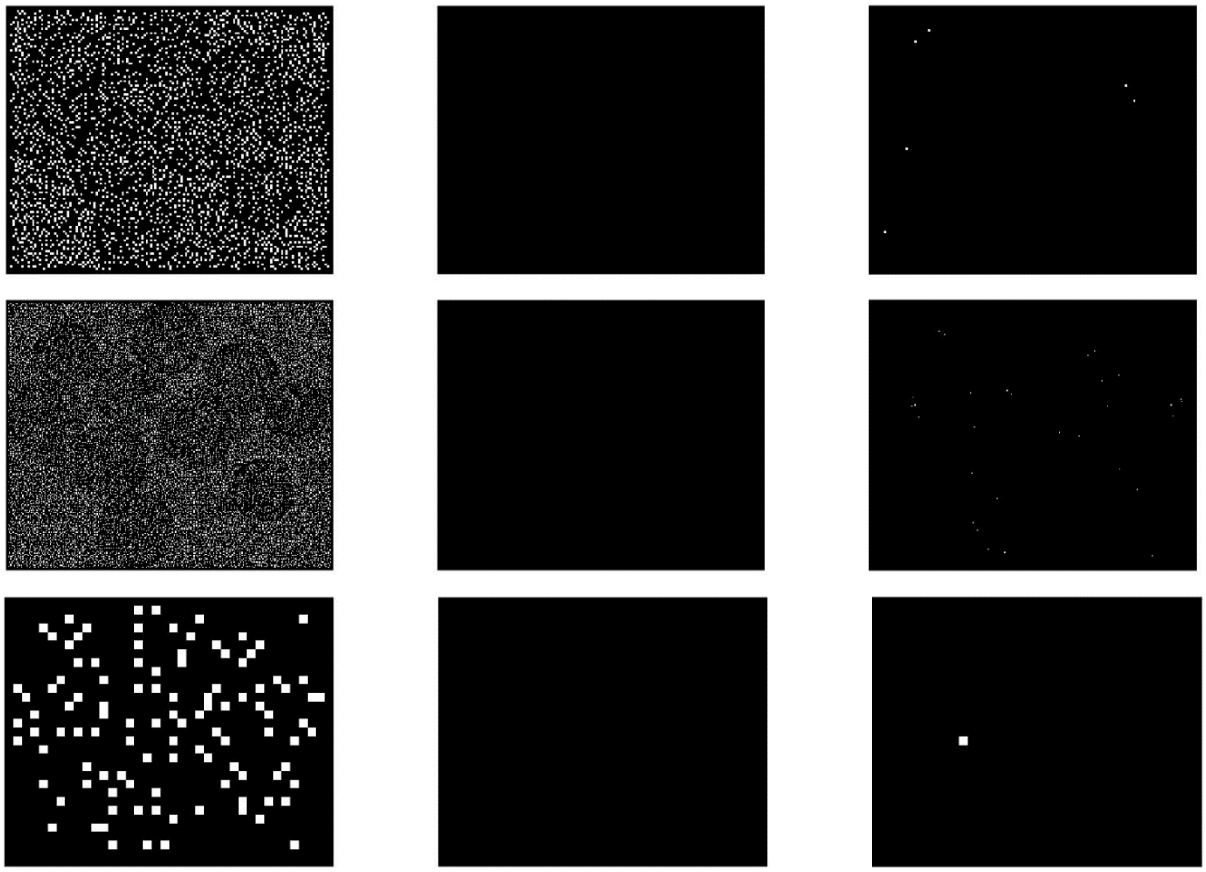


Figure 3.7 Bitmap of the extrema points detected in the DoG images. This is repeated for every resizing of the images. Notice there are 3 images instead of 5 since every combination of 3 images make 1 bitmap of extrema points.

4. Feature Matching

Features can be extracted from images with various methods, of which some are mentioned in the previous section. Nonetheless, features are not just used for object recognition, for example. They can also be used to obtain 3D scene geometry from multiple images of the same scene. To achieve this the features extracted from two or more images must first be matched in correspondence or else none of this would provide any essential information. Once the correspondence matched points are found, further processing can be done which allows for warping one of the images in order to align it with the other. In this section, we will study the use of a brute-force feature matching method, formulate an algorithm for estimating the homography based on two images, derive an image warping technique to align two images and finally improve on the homography estimation using RANSAC.

4.1 Brute-force feature matching

Feature extraction is only half the process of correspondence matching. The next step is the actual matching. For the rest of this report, we will assume the use of MATLAB's feature extraction, and description built-in mechanism for Speeded-Up Robust Features (SURF) detection. Once the features and their corresponding descriptors containing information about the features, such as location, scale etc., are extracted using SURF, matching is performed. Matching is done by calculating the vector norm between two descriptors, say **p** and **q**, equivalently the Euclidean distance denoted as:

$$d(\boldsymbol{p}, \boldsymbol{q}) = \sqrt{(\sum(q_i - p_i)^2)}$$

For i = 1, 2, …, N, where N is the size of each vector. One of the images might have more features than the other, meaning that there exist unique features that might be occluded or non-existent in the other image. In order to avoid accounting such unique features, we can apply the so-called ratio test. The ratio test is utilised by getting, for each feature in one image, the two minimal distances and by dividing those, achieve the retrieval of a ratio. If that ratio is over a certain number, r, commonly used r = 0.8, then that entails that the feature is not unique or maybe a random match and therefore we discard it. The next step to optimise our matches, is discarding non-mutually exclusive matches by comparing the matches obtained from two separate runs, with different ordering. If the match does not occur in both of the runs, then it is not considered a match and therefore discarded as well. The code snippet below shows one run of the described method together with the application of a ratio test.

```
% Finding matching features using: 1) Euclidean Distance (2 minimum distances) and 2) Applying the
% ratio test to avoid unique matches (1st run)
index_pairs = zeros(h,2);
for i=1:h
    [first_min,second_min,index] = min_distance(original_feat_descriptors(i,:), distorted_feat_descriptors);
    r = (first_min/second_min);
    if r<0.8
        index_pairs(i,1) = i;
        index_pairs(i,2) = index;
    end
end
```

The function **min_distance** is used to compute the Euclidean distance between a descriptor from one image and all of the descriptors from the other image, sort the distances in ascending order and eventually retrieve the two minimum distances as well as the index of the closest descriptor. This method for matching corresponding features is inefficient and very time consuming and does not guarantee the best matches. Results from performing brute-force feature matching can be seen in Figure 4.1 below.

Figure 4.1 (Up) The original images, two perspectives of the same vase. (Down) The features are matched and their correspondence, i.e. the match points locations, are connected by a straight line between the images.

4.2 Homography estimation

The relationship of a set of matched points can be studied with the use of a 3x3 matrix, commonly referred to as a homography, (Richardt, 2017). By making the assumption of a pinhole camera model present, the homography defines the translation and rotation of a set of points from one image of a certain surface plane, to the second image of the same surface plane. Using the matched features, corresponding points, from two image planes, this relationship matrix can be evaluated easily with only on a single constraint; at least 4 corresponding points are required, (Criminisi, 1997). This constrain comes from the fact that the homography matrix H is a 3x3 matrix with 8 DOF. The homogeneous solution for the estimation of the hoomography is discussed below.

$$\boldsymbol{H} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \; and \; \boldsymbol{p}' = \boldsymbol{H}\boldsymbol{q}', \boldsymbol{p}' = \begin{bmatrix} p_x \; p_y \; 1 \end{bmatrix}^T \; and \; \boldsymbol{q}' = \begin{bmatrix} q_x \; q_y \; 1 \end{bmatrix}^T$$

Where H is my homography matrix and **p', q'** are a pair of corresponding points from two images in homogeneous coordinates, i.e. 3D coordinates with a z-value of 1. In the homogeneous solution we first need to define a vector form for H, call it **h:**

$$\boldsymbol{h} = [\, \boldsymbol{h_{11}}, \boldsymbol{h_{12}}, \boldsymbol{h_{13}}, \boldsymbol{h_{21}}, \boldsymbol{h_{22}}, \boldsymbol{h_{23}}, \boldsymbol{h_{31}}, \boldsymbol{h_{32}}, \boldsymbol{h_{33}}]$$

We then use the linear system of equations by defining the design matrix **A** with size 2n x 9 where n is the number of corresponding point pairs to be used**,** which:

$$\boldsymbol{Ah} = \boldsymbol{0}$$

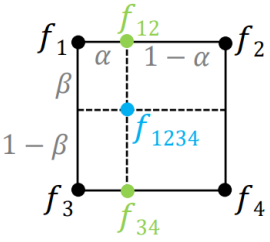
Matrix A can be easily formulated by manually setting each of its values just like the snippet of code below:

```
function H = homography_calc(points1,points2)
    [h,~] = size(points1);
    [h2,~] = size(points2);
    if h<4 || h2<4 || ~(isequal(h2,h))
        error('Function homography_calc(A,B) requires that A and B must have at least 4 points, each, and be of the same size to estimate the homography matrix');
    end

    A = zeros(2*h,9);

    % Easy construction of the A matrix
    for i = 1:h
        A((2*i-1),:) = [points1(i,1), points1(i,2), 1, 0, 0, 0, -points1(i,1)*points2(i,1), -points2(i,1)*points1(i,2), -points2(i,1)];
        A((2*i),:)   = [0, 0, 0, points1(i,1), points1(i,2), 1, -points1(i,1)*points2(i,2), -points2(i,2)*points1(i,2), -points2(i,2)];
    end

    [~,~,V] = svd(A);
    h=V(:,end);

    H = reshape(h,3,3);
    H = (1/H(end,end))*H;
end
```

Where points1 and points2, are the corresponding points. The for loop in the function **homography_calc**, is responsible for setting matrix A easily, since we know that for each corresponding point pair we will have 2 entries in the matrix. The last few lines of the snippet show exactly how to derive the formulation of H, by performing Singular Value Decomposition (SVD) and getting the null vector of A, which is the eigenvector of the least eigenvalue, more specifically the right null column vector of V. We then reshape vector **h**, to become the 3x3 matrix H, and divide all its values by the last element to transform it back from homogeneous coordinates.

4.3 Visualising the homography

Once the homography matrix H is formulated, one can perform the rigid point transformation on each point of an image to obtain a new image. This is known as digital image warping. Digital image warping can be done in two ways, with one being forward mapping, i.e. for each pixel from a source image **p,** perform the transformation we wish on it, in this case **H**, and determine the corresponding pixel in the target image **p' = Hp** which we colour with the colour of **p**. Forward mapping does not produce the best results and this is mainly due to the fact that the target image pixels might not be integers. Since we address each pixel by integer value coordinates, a correction is usually made such as rounding up or down, meaning that the target image will result with gaps. The best results are given by the second method, backward mapping and what is used in this implementation. Backward mapping ensures that each pixel in the target image gets a colour value by reversing the transformation of point mapping. Therefore, each target image pixel will receive a colour value corresponding to the source image pixel coordinate given by $\mathbf{p} = \mathbf{H}^{-1}\mathbf{p'}$. This results in fewer gaps in the image but does not necessarily provide optimal results since coordinates for pixel **p** will be computed as a real number which again might need correction. This is where bilinear interpolation takes the stand**.** Bilinear interpolation is an extension to linear interpolation, by which we can use to find the colour corresponding to the pixel neighbourhood surrounding our real-valued pixel **p.** In the case of image warping, bilinear interpolation works as a kind of weighted average provider, by weighting the distance from the real-valued coordinate we calculated with the four pixels closest to it to give a back a weighted average of the RGB colour value for our target pixel. Figure 4.2 shows how to estimate the colour value for our calculated source image pixel coordinate. The colour value for f1234 is computed by first finding f1, which is up (ceil(y)) and left (floor(x)), then f2, which is up and right (ceil(x)) and so on. The value of distance a, can



$$f_{12} = (1-\alpha)f_1 + \alpha f_2$$

$$f_{34} = (1-\alpha)f_3 + \alpha f_4$$

$$f_{1234} = (1-\beta)f_{12} + \beta f_{34}$$

Figure 4.2 This is the concept of bilinear interpolation. f1234 is the pixel coordinates found after performing the homography transformation, (taken from lecture slides).

be computed from the x-dimension distance between f1234 and f1 and b can be computed by the y-dimension distance between f1234 and f1. Since we know the size of each pixel, 1, then 1-a and 1-b is logical. Calculating f12 and f34 is now feasible and f1234 is obtained as easily.

Here is a code snippet showing the image warping function implemented for this assignment.

```
function transformed = imgwarping2(src,trgt, H)
    transformed = zeros(size(trgt));
    for i = 1:size(trgt, 1)
        for j = 1:size(trgt, 2)

            point = [j i 1];
            p_prime = point/H;% Equivalent to point*inv(H), proposed by matlab
            p_prime = (1/p_prime(3))*p_prime;
            y = p_prime(1);
            x = p_prime(2);

            if (isinteger(x) && isinteger(y)) % if both x and y coordinates are integers, just set the corresponding pixel
                transformed(i, j, :) = src(x, y, :);
            else % else perform bilinear interpolation
                x_left = floor(x);
                y_down = floor(y);
                x_right = ceil(x);
                y_up = ceil(y);

                dis_left = x-x_left;
                dis_down = y-y_down;
                dis_right = 1-dis_left;
                dis_up = 1-dis_down;

                if (y_down > 0 && x_left > 0 && x_right <= size(src, 1) && y_up <= size(src, 2) )
                    % As seen from the lecture slides
                    f1 = src(x_left,y_up,:);
                    f2 = src(x_right,y_up,:);
                    f3 = src(x_left,y_down,:);
                    f4 = src(x_right,y_down,:);
                    f12 = dis_right*f1 + dis_left*f2;
                    f34 = dis_right*f3 + dis_left*f4;
                    f1234 = dis_down*f12 + dis_up*f34;
                    transformed(i, j, :) = f1234;
                end
            end
        end
    end
end
```

As you can see, there is a check for whether the calculated source image coordinates are integers and if they are not, then bilinear interpolation is utilised to compute the optimal colour value.

Here is an example of digital image warping with and without bilinear interpolation.

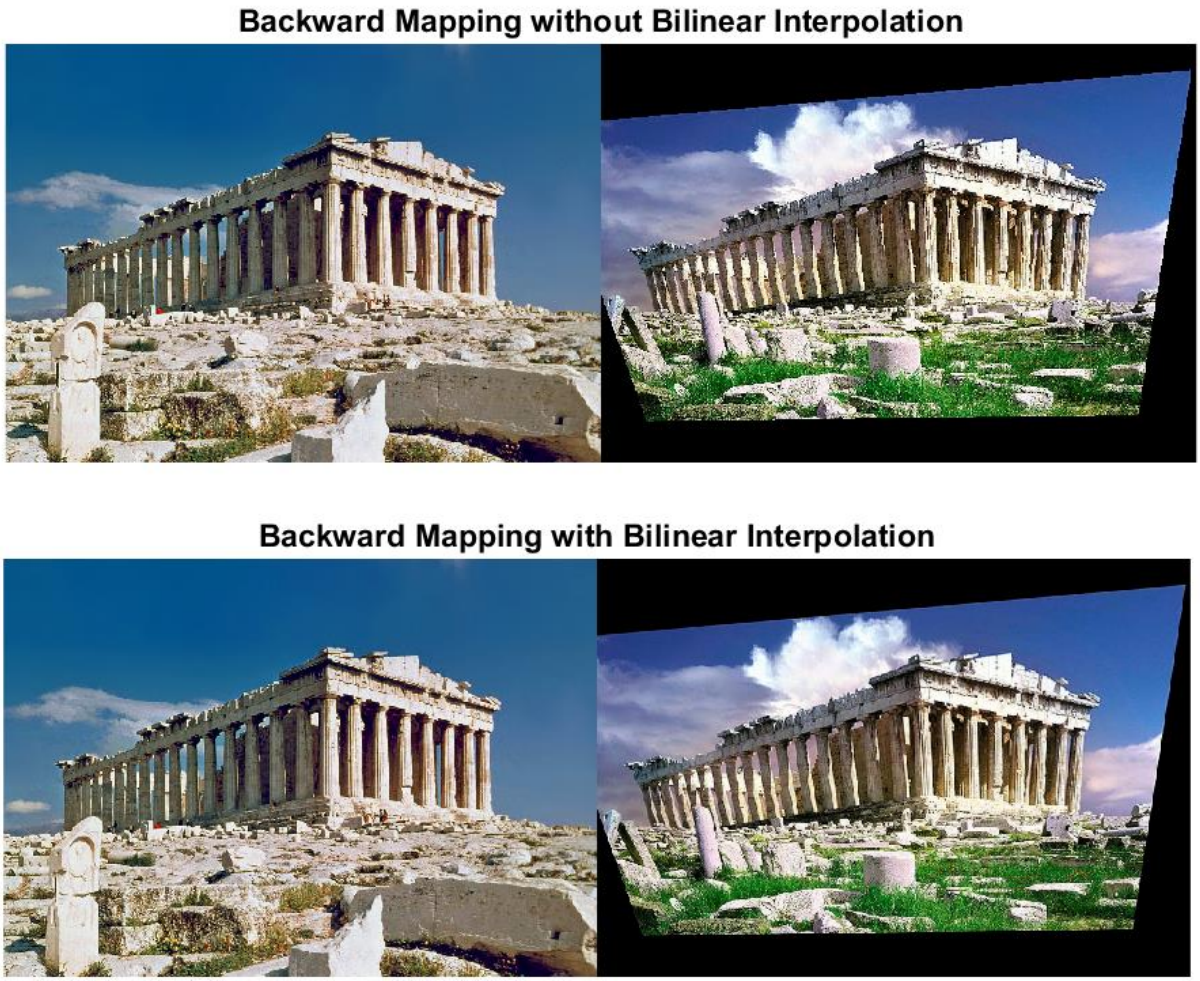


Figure 4.3 (Up) The original images before the homography matrix is calculated and digital image warping is performed. (down) Example warp using two different images of Parthenon in Greece. Notice that the warping of the image done using bilinear interpolation has smoother colour transitions and straighter borders when compared to the warp done without bilinear interpolation. The homography matrix between the two images was produced using 4 predetermined matched corresponding points.

4

Given a correspondence point set consisting of 4 matches, might not produce the best homography estimation, since the matched points may be outliers or a result of a false matching. We wish to avoid such a scenario and make the process of picking the best homography matrix automatic. RAndom SAmple Consensus (RANSAC) was delivered by Fischler et al. which can provide this automatic solution by randomly selecting the minimum required points from a dataset for a pre-determined number of iterations, until the best model is found. Therefore, RANSAC can be evaluated as a parameter tuning technique where the parameter in the case at hand is estimating the homography transformation that produces the most inliers when transforming each point and checking whether a distance error is within a threshold to the corresponding matched point. The steps for performing RANSAC are simple and easy to follow but more detailed explanation can be seen at (Fischler and Bolles, 1981) and (Hartley and Zisserman, 2004). The algorithm implemented here is following the steps of (Richardt and Hall, 2017) which in turn aided significantly in improving the homography choice. The steps used here to perform RANSAC for homography estimation are listed below.

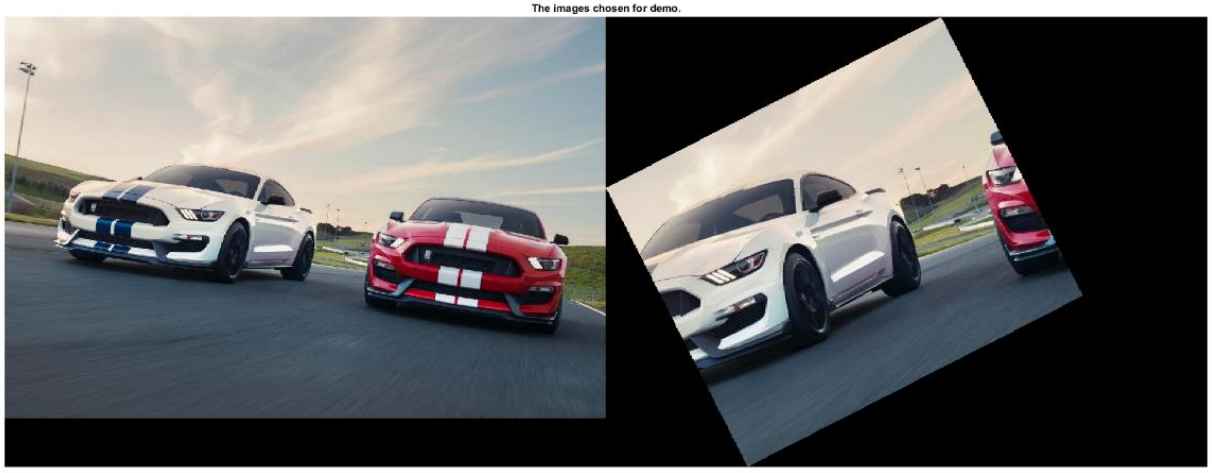
RANSAC for homography (N, m, d-thresh, p, q) : (Inputs are N, the number of iterations to run, m, number of points to be used for homography estimation, d-thresh, the distance threshold, p the matched points from image 1, q, the matched points from image 2)

1. For i = 1:N iterations:
    a. Randomly select m matches from p and q assumed to be inliers.
    b. Compute homography, H_i, with **homography_calc** function using the m matched points.
    c. For j = 1: number of matched points (size(p,1))
        i. Fit the model (homography) : $\boldsymbol{q}_j{}' = H_i^{-1}\boldsymbol{p}_j$
        ii. Check Sum of Squared Differences or Euclidean distance between $\boldsymbol{q}_j'$ and $\boldsymbol{q}_j$
        iii. If the error is below d-thresh then considered an inlier otherwise an outlier.
    d. Record random indices and number of inliers for comparison later on.
    e. Store the highest quality matcher (the one with most inliers)
2. Optionally refit model using the stored random indices of the best model, i.e. the model with most inliers to further improve model.
3. Return homography matrix H.

Since there are several parameters to be pre-determined, a value for each will be provided and explained. For the number for iterations, choosing a big number results in more combinations of matched points to be examined, further increasing the probability that the optimal model is computed. As for the points to be used to compute the model, m in the algorithm above, the minimum number required is chosen, in this case 4, as the homography estimation requires at least 4 corresponding points. Varying the threshold that accepts or rejects an inlier, changes the result significantly, as choosing a very low threshold will reject inliers frequently, but choosing a very high one might lead to outliers being considered inliers, which is not optimal. Trial and error based on a few runs will help with picking the best value. Below is the code snippet showing the parameters chosen for this implementation, while the full code can be seen in cwp4.m submitted on moodle:

```
ransac_parameters.it_num = 1000;     %how many iterations the ransac algorithm to run
ransac_parameters.num_for_homo = 4;          %how many points to be used to estimate homography on each run
ransac_parameters.distance_thr_inlier = 1;   % the threshold deciding the fate of the point. is it an inlier or outlier?
```

To validate the results of choosing the best homography using RANSAC in this assignment, here are a few runs using digital image warping to show the correct alignment of images, Figures 4.4 and 4.5.



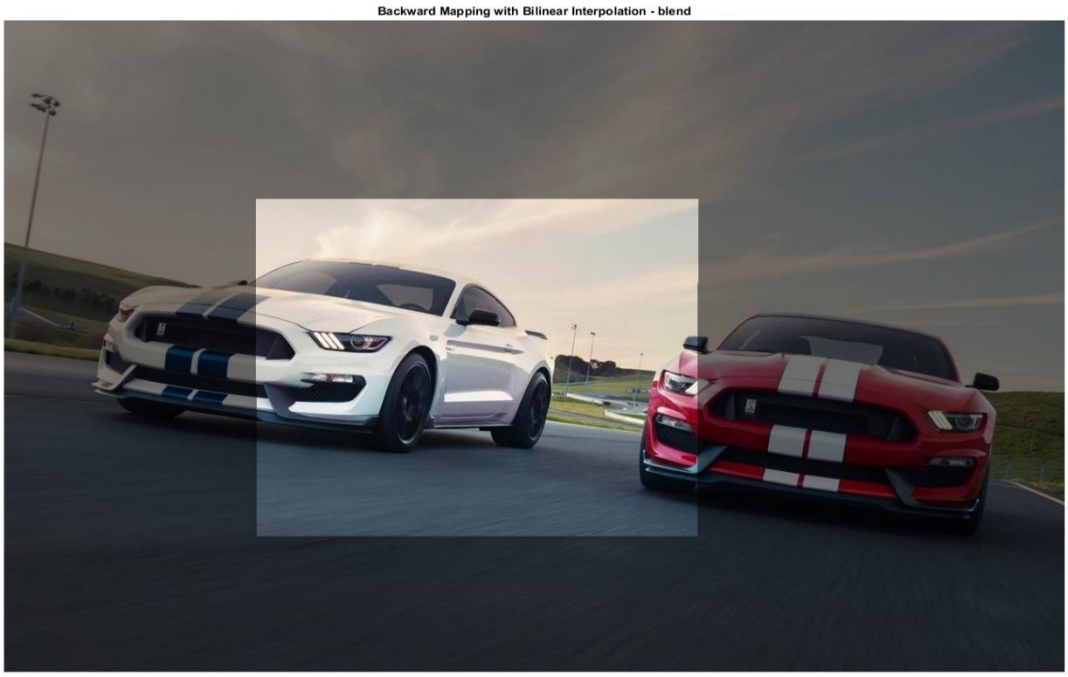


Figure 4.4 (Up) The original images passed into the algorithm. The second image is cropped, rotated and scaled to show the performance of the RANSAC algorithm on choosing the best model. (Down) The homography matrix derived was optimal since it rotated the second image back, rescaled it and aligned it perfectly to match the first image.

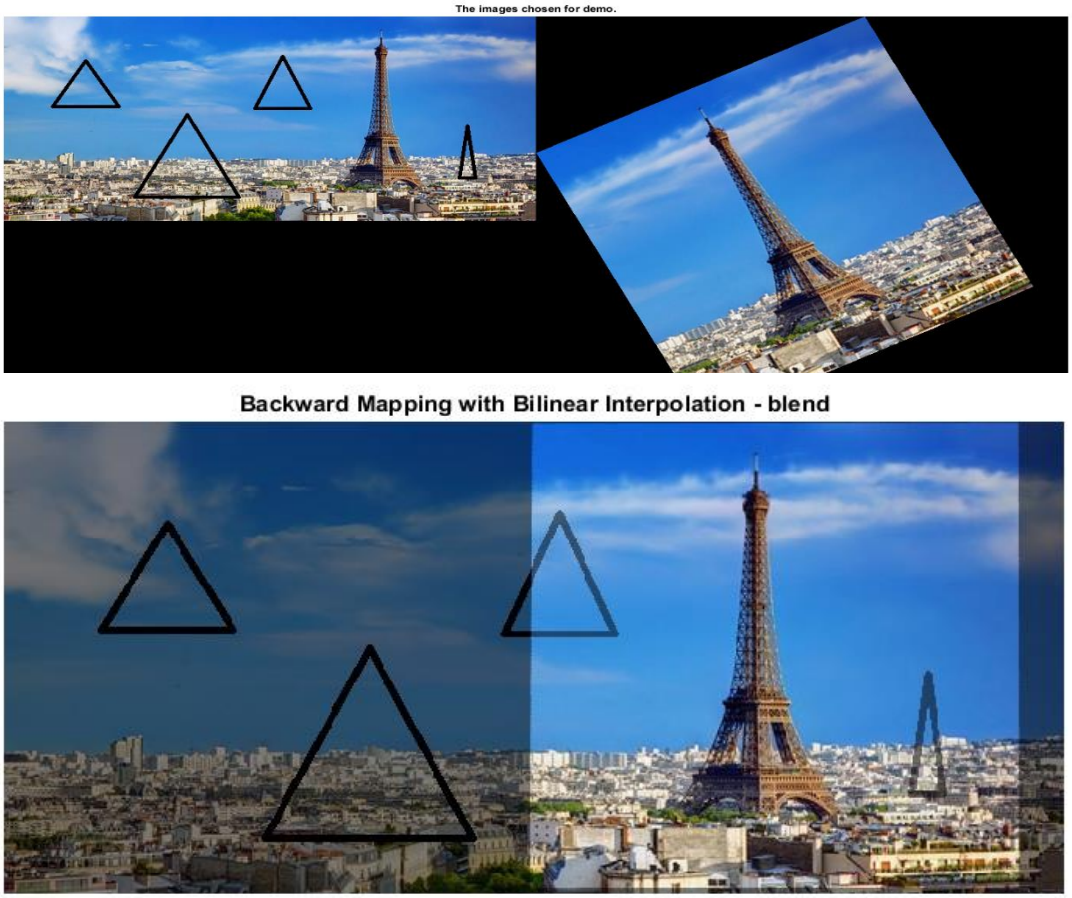


Figure 4.5 (Up) The images used to be aligned. The first image is edited with some distractors (triangles) and the second image is again rotated and rescaled. (Down) The estimation of homography performed by RANSAC, once again, performed efficiently without any problems that might have been caused by either the distractors or the rotation and scaling. This is the final alignment of the images.

As expected, RANSAC optimised the results of homography estimation. The images shown above, demonstrate that the best homography matrix is chosen, given that the images are rotated to fit the other image, rescaled to match the scale characteristics of the other image and placed at the correct location when the images are warped together or stitched. The fact that the distractors in Figure 4.5 did not ruin the homography estimation, shows that the combination of the strongest features is used producing the most inliers.

5. Camera Calibration

Camera calibration is the process of determining, or better estimating, the internal parameters of a camera, such as its principal point, and distortion exerted by the camera. The implementation followed in this report is directly related to that of Zhang, (Zhang, 2000). Zhang proposes a technique that is cheap to follow, since it only involves a computer's built-in web cam and a printed checkerboard pattern. Since there were some technical difficulties taking images from a web cam, the images of the checkerboard pattern taken at different angles were downloaded from an online database and were referenced in the code. The process of camera calibration can be broken down into smaller steps and that is how it will be explained below, in the closed form solution.

5.1 Intrinsic camera parameters

To estimate the camera intrinsic parameters requires the process of taking photographs of the same checkerboard pattern at different orientations and distance from the camera. This process was skipped by using the database found online. Using MATLAB's built-in function detectCheckerboardPoints, we can extract the locations of the squares' corners based on the image coordinate system. Using this function, given a number of images, n, MATLAB was able to return only the location of the points corresponding to corners of the checkers and a board's dimension size. Based on the images used, the checkerboard pattern contained 13x14 squares. Therefore, for each image passed in, we have 13x14 points, which we denote each one as **m.** The corresponding mapping taking into consideration the intrinsic camera matrix **A** is:

$$s\begin{bmatrix}u\\v\\1\end{bmatrix} = \boldsymbol{A}\,[\,\boldsymbol{R}\mid t\,]\begin{bmatrix}X\\Y\\0\\1\end{bmatrix} \quad where\ \boldsymbol{m}' = \begin{bmatrix}u\\v\\1\end{bmatrix}\ \ and\ \boldsymbol{M} = \begin{bmatrix}X\\Y\\0\\1\end{bmatrix}\ and\ \boldsymbol{A} = \begin{bmatrix}\alpha & \gamma & u_0\\0 & \beta & v_0\\0 & 0 & 1\end{bmatrix}$$

In the above formula, **A** is the matrix desired to estimate, the camera's intrinsic parameters, and the matrix consisting of the rotation matrix concatenated with a translation vector are the camera's extrinsic parameters. **M** is the model plane homogeneous coordinates corresponding to **m'** which is the homogeneous image point**.** In order for this to work, we must assume that all our points lie on the same plane, i.e. on the plane with a z-coordinate of 0, thus the inclusion of a 0 on our model point homogeneous representation. We must first decide our model point coordinates, and for this we can create our own coordinate system based on the checkerboard size, which we will refer to as normalised or model coordinates. In this case, a simple function was used:

```
function checkPoints = translate_to_checker(b_size)
    checkPoints = [];
    for i = 0:b_size(2)-2
        for j = 0:b_size(1)-2
            checkPoints = [checkPoints; [j i]];
        end
    end
end
```

The next step in the procedure, included the estimation of a homography matrix between each set of image points with our model coordinates, using again 4 corresponding points. The procedure did not change from that of Section 4.2, other than the fact that the points were normalised in a sense that they became zero centred and at a maximum distance of $\sqrt{2}$ from the origin, before the homography estimation, and the homography matrix was then unnormalized before returned. RANSAC could be

utilised but an explanation of the avoidance will be given later. Given the homography estimation we can utilise certain constraints on the intrinsic matrix, **A.** These are:

$$h_1^T A^{-T} A^{-1} h_1^T = 0 \quad and \quad h_1^T A^{-T} A^1 h_1 = h_2^T A^{-T} A^{-1} h_2$$

Where:

$$H = [\, h_1 \; h_2 \; h_3 \,] = \lambda A \, [\, r_1 \; r_2 \; t]$$

Based on these constraints we can construct a 2nx6 matrix, **V**, which we can use as $\boldsymbol{Vb} = 0$ to define a linear equations solution to **b**. Since we achieve the retrieval of 2 values from each homography, and the matrix **B** (corresponding to vector **b**), has 6 degrees of freedom, this entails that at least 3 images are required. Every 2 rows of the V matrix correspond to one homography and are obtained by:

$$\begin{bmatrix} v_{12}^T \\ (v_{11} - v_{12})\text{^}T \end{bmatrix} \boldsymbol{b} = \mathbf{0}$$

Each **v** can be composed from the entries of the homography matrix by:

$$\mathbf{v}_{ij} = [h_{i1}h_{j1}, h_{i1}h_{j2} + h_{i2}h_{j1}, h_{i2}h_{j2}, \\ h_{i3}h_{j1} + h_{i1}h_{j3}, h_{i3}h_{j2} + h_{i2}h_{j3}, h_{i3}h_{j3}]^T .$$

Or in code:

```
function v = con_v(H,i,j)

    v = [H(1,i)*H(1,j),H(1,i)*H(2,j)+H(2,i)*H(1,j),H(2,i)*H(2,j), ...
            H(3,i)*H(1,j)+H(1,i)*H(3,j),H(3,i)*H(2,j)+H(2,i)*H(3,j),H(3,i)*H(3,j)];
end
```

```
[~,~,V] = svd(V_mat);
% b = [B11,B12,B22,B13,B23,B33]
b = V(:,end);
B11 = b(1);
B12 = b(2);
B22 = b(3);
B13 = b(4);
B23 = b(5);
B33 = b(6);

% y coordinate of principal point
v0 = ((B12*B13)-(B11*B23))/((B11*B22)-power(B12,2));

%lambda
lambda = B33-((power(B12,2)+v0*((B12*B13)-(B11*B23)))/B11);

% a
alpha = sqrt(lambda/B11);

% b
temp = (lambda*B11)/(B11*B22-power(B12,2));
beta = sqrt(temp);

%gamma -> skewness
gamma = (-B12*power(alpha,2)*beta)/lambda;

% x coordinate of principal points
u0 = ((gamma*v0)/beta)-((B13*power(alpha,2))/lambda);

% Intrinsic parameter matrix A
A = [alpha,gamma,u0;0,beta,v0;0,0,1];
```

The **b** vector can be derived by performing SVD decomposition on **V** and taking the right singular vector of V (not the same V, but from SVD(V)) corresponding to the smallest singular value. Naturally, vector **b** is a vector consisting of 6 values $\boldsymbol{b} = [\, B_{11}, B_{12}, B_{22}, B_{13}, B_{23}, B_{33} \,]^T$ with values of the symmetric matrix $B = A^{-T} A^{-1}$. The final step towards deriving A is using the values of the **b** vector to construct each term individually, just like in the code snippet on the right, with the final step of constructing the intrinsic parameter matrix at the bottom, just as explained in Zhang's paper.

Discussion:

During the implementation procedure of coding the intrinsic parameters' retrieval, there was an issue with the values that made up the intrinsic matrix. Namely, the problem was that some of its values were found to contain imaginary numbers. Using RANSAC to compute the best homography, repeatedly lead to this. Instead, random indices were used to decide on the four points to estimate the homography until the intrinsic parameters did not contain imaginary numbers. The indices were then kept constant throughout to ensure the acceptability of the intrinsic matrix for later use. This issue is believed to be present based on the images used in the first place and affected the derivation of the extrinsic parameters to be explained in the next section.

Example runs of camera calibration using 3, 4 and 5 images:

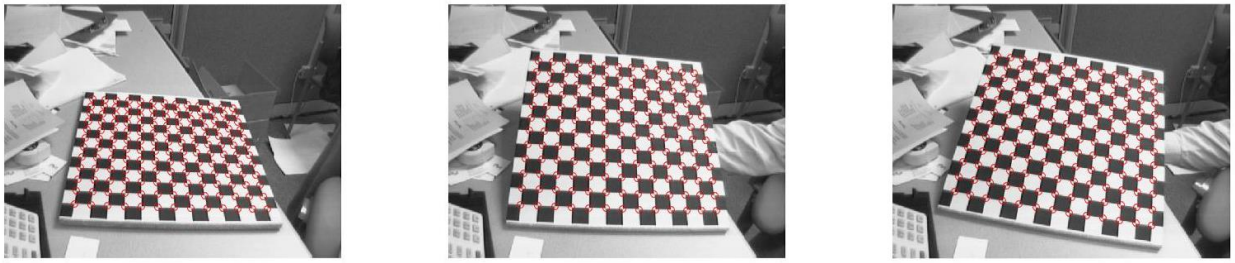


Figure 5.1 The original images used and their detected checkerboard corners with red circles.

The intrinsic matrix A produced:

```
A =

   60.4069   78.3739    7.4853
         0   16.9787   14.7209
         0         0    1.0000
```

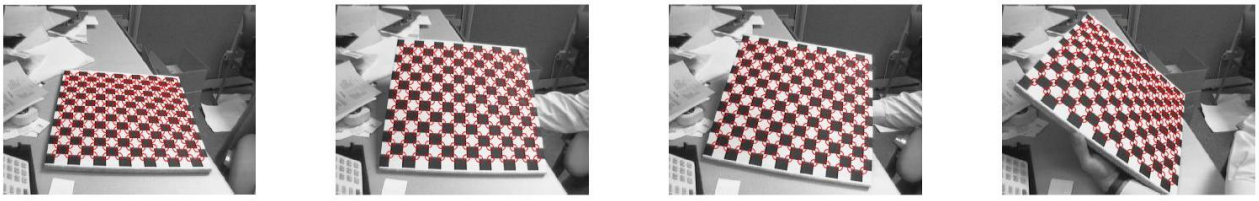


Figure 5.2 The original images used and their detected checkerboard corners with red circles.

The intrinsic matrix A produced:

```
A =

   32.5694   44.1443   -2.9226
         0   17.1950   11.5658
         0         0    1.0000
```

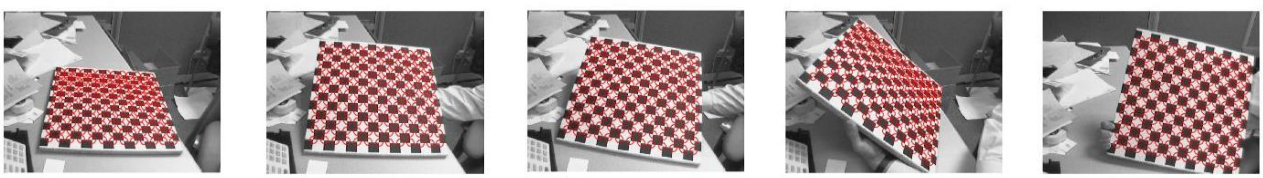


Figure 5.3 The original images used and their detected checkerboard corners with red circles.

The intrinsic matrix A produced:

```
A =

   35.6164   37.3121    1.5277
         0   13.6675   13.4200
         0         0    1.0000
```

As one can tell from the intrinsic parameters estimation, the results are not completely consistent. This is something that could be fixed but given the urgency of this submission was left to be pointed out during feedback, as hours of debugging were already spent.

5.2 Lens distortion

With the intrinsic parameters acquired, a further step implies the estimation and refinement of the radial distortion coefficients, k_1 and k_2. Distortion coefficients can be used to get rid of barrel, tangential and other types of distortion from an image. To find these coefficients, the extrinsic parameter matrix [R|t] based on each camera was required, so that a projection using the first formula in Section 5.1 can be made possible. The extrinsic parameters are derived by the homography matrix of each image with the model coordinates and the acquired intrinsic matrix; they were produced by the following calculations found in Zhang's paper:

```
function E = extrinsic_params(A,H)
    h1 = H(:,1);
    h2 = H(:,2);
    h3 = H(:,3);

    temp1 = sqrt(inv(A)*h1);
    lamda1 = 1/dot(temp1,temp1);

    temp2 = sqrt(inv(A)*h2);
    lamda2 = 1/dot(temp2,temp2);

    lamda3 = (lamda1+lamda2)/2;

    r1 = lamda1*inv(A)*h1;

    r2 = lamda2*inv(A)*h2;

    r3 = cross(r1,r2);

    t = lamda3*inv(A)*h3;

    E = [r1 r2 r3 t];

end
```

```
extrinsics(:,:,1) =

   -0.0876   -0.4452   -0.0007  -58.4066
   -0.0748    0.0447    0.4176   59.0873
   -0.8376    0.5101   -0.0372  373.0374


extrinsics(:,:,2) =

    0.0436   -0.4334   -0.0259 -340.4710
   -0.0959    0.0297    0.3495   90.4072
   -0.8605    0.5369   -0.0402  281.0020


extrinsics(:,:,3) =

    0.0843   -0.2380   -0.1310 -673.8404
   -0.0950   -0.0808    0.1379  126.2894
   -0.8207    0.6812   -0.0294   74.3828
```

Where the **extrinsics** = [R|t], seen in the results on the right above. It is only natural that every viewpoint has a different set of extrinsic parameters, thus for 3 images we extract 3 extrinsic parameters sets. The lens distortion coefficients can then be derived using the Alternation method in Zhang's paper to find a first estimate of these values. This process requires 4 sets of coordinates, namely: 1) [u, v] the ideal image coordinates, 2) [u', v'] the corresponding real image coordinates (can be seen in the image), 3) [x, y] the model coordinates and 4) [u0, v0] the principal point from the intrinsic matrix. Using these sets of coordinates, we can construct a matrix, D, and a vector, **d**, which in turn can be utilised to estimate the vector $\boldsymbol{k} = [\boldsymbol{k_1}, \boldsymbol{k_2}]^{\boldsymbol{T}}$, holding the distortion coefficients, by stacking the following formulas:

$$\begin{bmatrix} (\boldsymbol{u}-\boldsymbol{u_0})(\boldsymbol{x^2}+\boldsymbol{y^2}) & (\boldsymbol{u}-\boldsymbol{u_0})(\boldsymbol{x^2}+\boldsymbol{y^2})^2 \\ (\boldsymbol{v}-\boldsymbol{v_0})(\boldsymbol{x^2}+\boldsymbol{y^2}) & (\boldsymbol{v}-\boldsymbol{v_0})(\boldsymbol{x^2}+\boldsymbol{y^2})^2 \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} u'-u \\ \boldsymbol{v'}-\boldsymbol{v} \end{bmatrix}$$

Which is equal to: **D** **k** = **d**

Once this is done on every image point from a single image, and the entries of **D** and **d** are stacked, we can proceed by performing this rearrangement:

$$\boldsymbol{k = \left(D^T D\right)^{-1} D^T d}$$

Although this returns the radial distortion coefficients, it does not necessarily mean that they are the best ones. Instead, in his paper, Zhang proposes 2 methods of optimising these by either alternation, which was partially done here, or using complete Maximum Likelihood Estimation (MLE) to minimise a certain functional, since the problem is now non-linear. In this implementation, no refinement or optimisation is performed, as it felt difficult to follow and given there was no prior knowledge of the curve fitting algorithm called Levenberg-Marquardt, it was chosen to be left out. Even though its concept is very straightforward, the paper suggests using the method described in the previous sections to estimate A and vector k, as initial estimates, and using the minimisation technique find their optimal values iteratively, but a confusion as to which parameters should vary existed. The only optimisation performed in this implementation is by visualising the results using the undistortImage built-in function after passing it the intrinsic parameters computed and the initial radial distortion coefficients. Then manually dividing the radial distortion by a step size of 0.1 to get rid of lens distortion, from visually checking the undistorted images. Here are some of the results, even though not completely optimal:

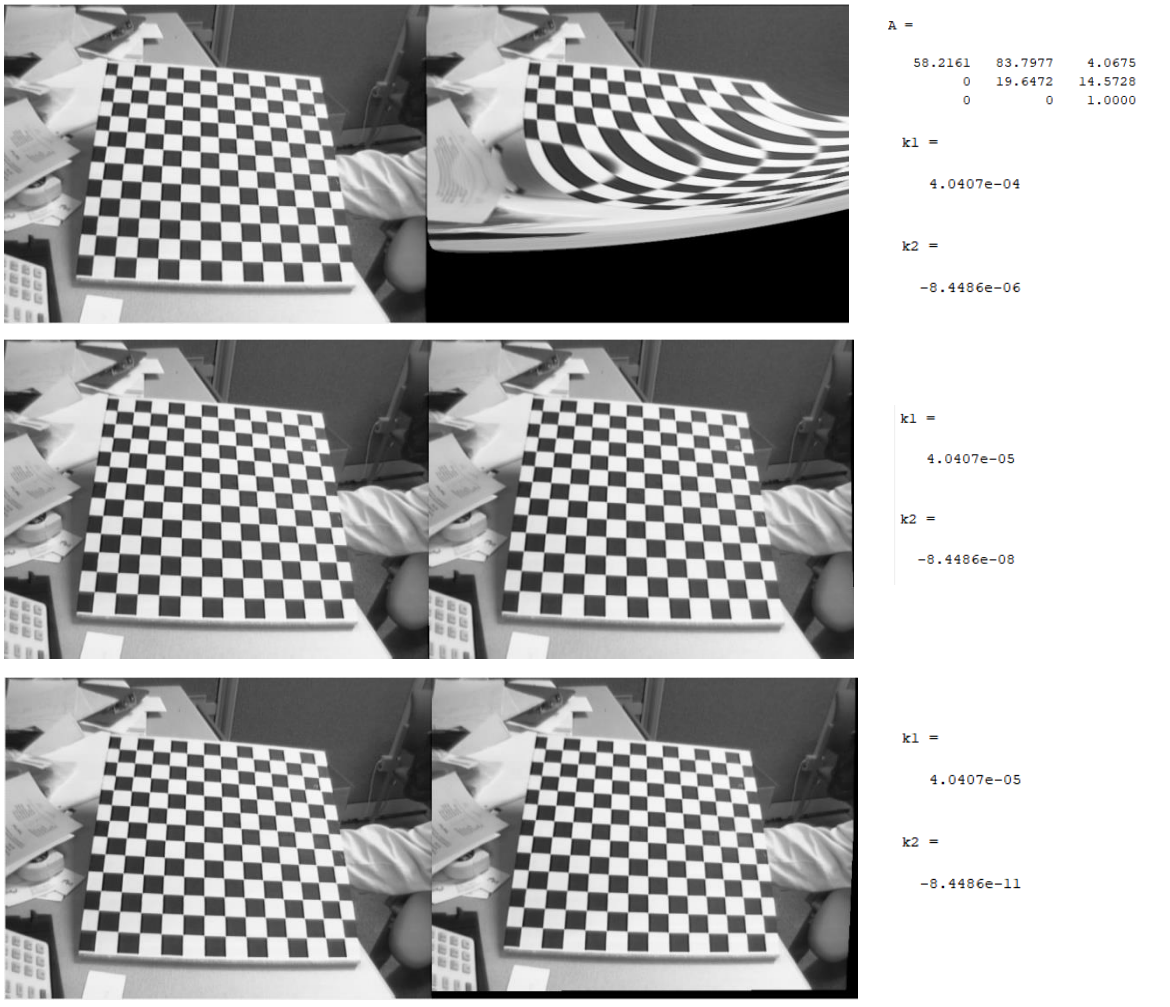


Figure 5.4 As you can see, optimisation by visualisation is not optimal and the results are not as undistorted as desired. On the right of each image are the values for the intrinsic parameters matrix, A, and the distortion coefficients with their values changed for every run. With the MLE approach described in Zhang's paper. the results would have been rid from the error in the estimation of both the intrinsic parameters and the distortion coefficients.

6. Fundamental matrix and Essential matrix

This section includes the process of estimating the fundamental matrix, which is used to relate corresponding points from 2-view geometry or stereo images. Additionally, a further refinement of the fundamental matrix to improve the estimation procedure is explained and finally the procedure for computing the essential matrix and the cameras' extrinsic parameters given that we have the fundamental matrix will be derived. For this part of the coursework, SURF features are used to find the corresponding matched points.

6.1 Fundamental matrix estimation

The notion of the fundamental matrix, F, is not that different of that of a homography matrix. It is essentially a matrix that can be used to derive a point from one image given its corresponding matched point on the other image. In fact, the formula associated with the fundamental matrix is the coplanarity constraint, i.e. the transformation of one image point to the other given the fundamental matrix will result on the same plane:

$$\boldsymbol{x}'^{T}\, F\, \boldsymbol{x} = \mathbf{0}$$

Or more clearly:

$$[x', y', 1] \begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0$$

Where **x'** and **x** are corresponding points in homogeneous coordinates. This derivation comes from the epipolar geometry further explained in (Hartley and Zisserman, 2004), which is the "geometry of intersection of the image planes with the pencil of planes having the baseline as axis". The above formula implies the linear dependency of the functions of each element, and thus every term in the fundamental matrix depends on the known parameters which are the corresponding points from the 2 images. This linearity is explained thoroughly in the online course produced by Stachniss, (Stachniss, 2017). He explains that each unknown from the fundamental matrix, F_{ij}, can be derived using the Kronecker Product:

$$(\boldsymbol{x_n} \otimes \boldsymbol{x'_n})^{T}\, vecF = \boldsymbol{a}_n^{T}\, \boldsymbol{f} = 0 \qquad for\ n = 1,2, \ldots . N,$$

$$N = \ number\ of\ corresponding\ points$$

Where $\boldsymbol{a}$ is a 9-value vector holding the Kronecker Product between **x** and **x'**, and **f** is the vector form of the fundamental matrix **F.** Then we can stack all the transpose vectors, $\boldsymbol{a}_{\boldsymbol{n}}^{\boldsymbol{T}}$, into a single matrix, denoted as A:

$$A = \begin{bmatrix} \boldsymbol{a}_{\boldsymbol{1}}^{\boldsymbol{T}} \\ \ldots \\ \boldsymbol{a}_{\boldsymbol{n}}^{\boldsymbol{T}} \\ \ldots \\ \boldsymbol{a}_{\boldsymbol{N}}^{\boldsymbol{T}} \end{bmatrix} \qquad and\ solve\ for \quad A\,\boldsymbol{f} = \mathbf{0}$$

Which is once again a system of linear equations, based on the coplanarity constraint on all corresponding points, and can be solved using SVD on A, retrieving the singular vector associated with the smallest singular value, which would ideally be 0. In theory, we know that the fundamental matrix F has 7 degrees of freedom given one of its properties, det(F) = 0, and the matrix A explained above has a rank of 8, from the Kronecker Product, which means that we need at least 7 point correspondences to estimate F. For this implementation, the algorithm used is called the 8-Point algorithm as it requires

that 8 corresponding point pairs are utilised. However, there is an uncertainty of estimating the fundamental matrix using the 8-point algorithm given that the corresponding points will contain noise, (Sur, Noury and Berger, 2008), which does not imply that our fundamental matrix is of rank 2 as proposed by Hartley and Zisserman. This entails that the smallest singular value of the smallest singular vector from the SVD of A will not be 0. Just like the homography matrix estimation, the process of using random correspondence matches implies that some of them may be outliers, entailing the formulation of an error, such as the noise mentioned above, that will ultimately ruin the estimation of the fundamental matrix. To account for this and resolve the issue we can force the F matrix to be of rank 2 by manually setting the smallest singular value to be 0. Therefore, from the SVD of A we can formulate a temporary fundamental matrix, call it F', which is a very close approximation to our desired fundamental matrix but with a smallest singular value not equal to zero. Then we can use F', or Fhat in the snippet below, to enforce this constraint by:

```
%SVD on A
[~,~,V] = svd(A);

f = V(:,end);
Fhat = reshape(f,3,3)';

% Find an approximation to F by manually setting the elements
% close to 0 to be equal to 0 (constrain enforcement)
[U,D,V_] = svd(Fhat);
F = U * diag( [D(1,1), D(2,2), 0]) * V_';
```

So, we perform SVD on F' and setting the diagonal matrix D to have a smallest value of 0. This is considered one of the main difficulties of estimating the fundamental matrix. Another liability of the fundamental matrix estimation is the fact that our problem may be ill-conditioned, the coordinate system may not be the same in both images, given high resolution images. In order to improve the stability of our estimation we formulate a normalisation matrix which transforms the coordinates of the points to be zero centred, by extracting the column means, and scale them so that they all lie in a distance of $\sqrt{2}$ from the centre. This normalisation is done in this way:

```
function [normal_p, T_p] = normalisePoints(p)

    % Normalises the points and return the normalisation matrix T
    n = size(p,1);
    mu_p = mean(p);

    % subtract the mean to make the points zero centred
    zero_centred = [p(:,1)-mu_p(1) p(:,2)-mu_p(2)];
    sd_p = std(zero_centred);
    T_sd_p = [sqrt(2)/sd_p(1),0,0;0,sqrt(2)/sd_p(2),0; 0,0,1];
    T_mu_p = [1,0,-mu_p(1); 0,1,-mu_p(2);0,0,1];

    % formulate normalisation matrix T
    T_p = T_sd_p * T_mu_p;
    normal_p = T_p * [p'; ones(1,n)];
    normal_p = normal_p';

end
```

Once we have the normalised points, as well as the corresponding normalisation matrices T and T', we can estimate the fundamental matrix as described above and finally unnormalize it by:

$$F = T_x'^{T} F_n T_x$$

Where F_n denotes the normalised fundamental matrix. The whole process of the normalised 8-point algorithm can be seen in the code snippet below:

```
function F = estimateF(p1,p2)
% inspired by the photogrammetry course by Cyrill Stachniss (very good
% lecturer and very in depth explanation)
% https://www.youtube.com/playlist?list=PLgnQpQtFTOGRsi5vzy9PiQpNWHjq-bKN1

    n = size(p1,1);

    [normal_p1, T_p1] = normalisePoints(p1);
    [normal_p2, T_p2] = normalisePoints(p2);

    for i=1:n
        A(i,:) = kron(normal_p2(i,:),normal_p1(i,:));
    end

    %SVD on A
    [~,~,V] = svd(A);

    f = V(:,end);
    Fhat = reshape(f,3,3)';

    % Find an approximation to F by manually setting the elements
    % close to 0 to be equal to 0 (constrain enforcement)
    [U,D,V_] = svd(Fhat);
    Fn = U * diag( [D(1,1), D(2,2), 0]) * V_';

    % Denormalise
    F = T_p2'*Fn*T_p1;
end
```

6.2 RANSAC for fundamental matrix estimation

The results from section 6.1 are still sub-optimal. They depend greatly on the fact that the 8 corresponding point pairs do not include any outliers and are perfect for producing a reliable estimation. Just like the case of homography estimation, RANSAC can be utilised. The procedure does not change from that of section 4.3, except for the fact that now our mapping is not $\boldsymbol{q}_j{}' = H_i^{-1}\boldsymbol{p_j}$ but instead of performing the mapping we calculate the error from $\boldsymbol{x}'^T\, F\, \boldsymbol{x} = \mathbf{0}$. The error is the value resulting from this and thus its distance from 0, and if it is less than a specific pre-determined threshold, then the point is considered an inlier, otherwise an outlier. Thus, the RANSAC approach, depending on the number of iterations set prior, will randomly select a combination of matched points from the images, estimate a matrix based on the estimateF function above, and return the one producing the most inliers. In this case, the RANSAC was given the parameters to run for 20000 iterations, with error threshold of 0.01 and at least 8 random points are used to find an estimation of the fundamental matrix. Additionally, there was no visualisation of results for the previous section given the many liabilities of not using RANSAC, but in this case, the following function is set to refit the model and based on the sorted errors of inliers, it returns the 30 best matches, to be used for drawing the epipolar lines on the images. Since no code snippet was provided in section 4.3, here is a code snippet showing how the RANSAC approach is utilised:

```
function [F, best_match1, best_match2] = ransac_fundamental(params, feat_points1, feat_points2)
   % based on https://moodle.bath.ac.uk/pluginfile.php/850436/mod_resource/content/6/CM50248%20Notes%20%282017-10-01%29.pdf
    feat_num = size(feat_points1, 1)
    best_num_inliers = 0;
    best_F = zeros(3,3);
    best_indices = [];
    p1 = [feat_points1 ones(feat_num,1)];
    p2 = [feat_points2 ones(feat_num,1)];
    for iter = 1: params.it_num

        % Step 1a: Randomly choose at least 8 points to calculate
        % fundamental matrix.
        random_indices = randsample(feat_num, params.num_for_F);
        points1 = feat_points1(random_indices,:);
        points2 = feat_points2(random_indices,:);

        % Step 1b: Compute model using points chosen randomly
        curr_F = estimateF(points1,points2);

        % Step 1c: Fit the model on the data to further check for
        % percentage of inliers
        error = sum((p2 .* (curr_F * p1')'),2);
        numb_of_inliers = size(find(abs(error) <= params.err_thr_inlier),1);

        % Step 2: Get the highest quality matcher (i.e the most inliers)
        if (numb_of_inliers > best_num_inliers)
            best_num_inliers = numb_of_inliers
            best_F = curr_F;
            best_indices = random_indices;
        end
    end

    F = best_F;
    % Step 3: Optionally refit model to all inliers to improve model
    % Consequently getting the best matches for epipolar lines plotting
    final_error = sum((p2 .* (F * p1')'),2);
    [~,indices] = sort(abs(final_error),'ascend');
    best_match1 = feat_points1(indices(1:30),:);
    best_match2 = feat_points2(indices(1:30),:);
```

To calculate the epipolar lines and eventually find 2 points to draw the line across them, we can follow the epipolar geometry, and more specifically the properties of the fundamental matrix. An epipolar line, l', on the second image can be computed using the fundamental matrix and a point from the first image, or vice versa:

$$l' = F\boldsymbol{x} \quad and \quad l = F\boldsymbol{x}'$$

Two points can then be calculated given the image's dimensions and finally can be superimposed on top of the corresponding second image. The epipoles of the images are where the baseline of the cameras intersects with the images and therefore the intersection point of all epipolar lines. To visualise the epipolar lines, the following function was used:

```
function vis_epipolar_lines( F, img1, img2, points1, points2)
   % F is the fundamental matrix
   % points1 and points2 are the best inliers of the point correspondences
   % will use these to draw epipolar lines
   % by getting two points from the epipolar line
    len1 = cross([1 1 1],[1 size(img1,1) 1]);
    len2 = cross([size(img1,2) 1 1],[size(img1,2) size(img1,1) 1]);

    figure; imshow(img2);
    for i = 1:size(points1,1)
        e = F*[points1(i,:) 1]';
        pos1 = cross(e,len1);
        pos2 = cross(e,len2);
        x1 = [pos1(1)/pos1(3) pos2(1)/pos2(3)];
        y1 = [pos1(2)/pos1(3) pos2(2)/pos2(3)];
        line(x1,y1)
    end

    hold on
    plot(points2(:,1),points2(:,2),'go','MarkerFaceColor','b','MarkerSize',7);
    hold off

    figure;imshow(img1)
    for i = 1:size(points2,1)
        e = F'*[points2(i,:) 1]';
        pos1 = cross(e,len1);
        pos2 = cross(e,len2);
        x2 = [pos1(1)/pos1(3) pos2(1)/pos2(3)];
        y2 = [pos1(2)/pos1(3) pos2(2)/pos2(3)];
        line(x2,y2)
    end

    hold on
    plot(points1(:,1),points1(:,2),'go','MarkerFaceColor','b','MarkerSize',7)
    hold off
end
```

Figures 6.1 and 6.2 show the epipolar lines in each image, based on the computation of the fundamental matrix and the corresponding matched image points from the other image.

Figure 6.1 Epipolar lines on two images of a vase based on the fundamental matrix estimated using the matched points and the RANSAC iterative method counting inliers. The points of one image are drawn on the other to show that the epipolar lines indeed pass through the matched points. The epipoles are located outside of the images as the camera was mostly translated along the x-axis direction and not a forward leap motion.

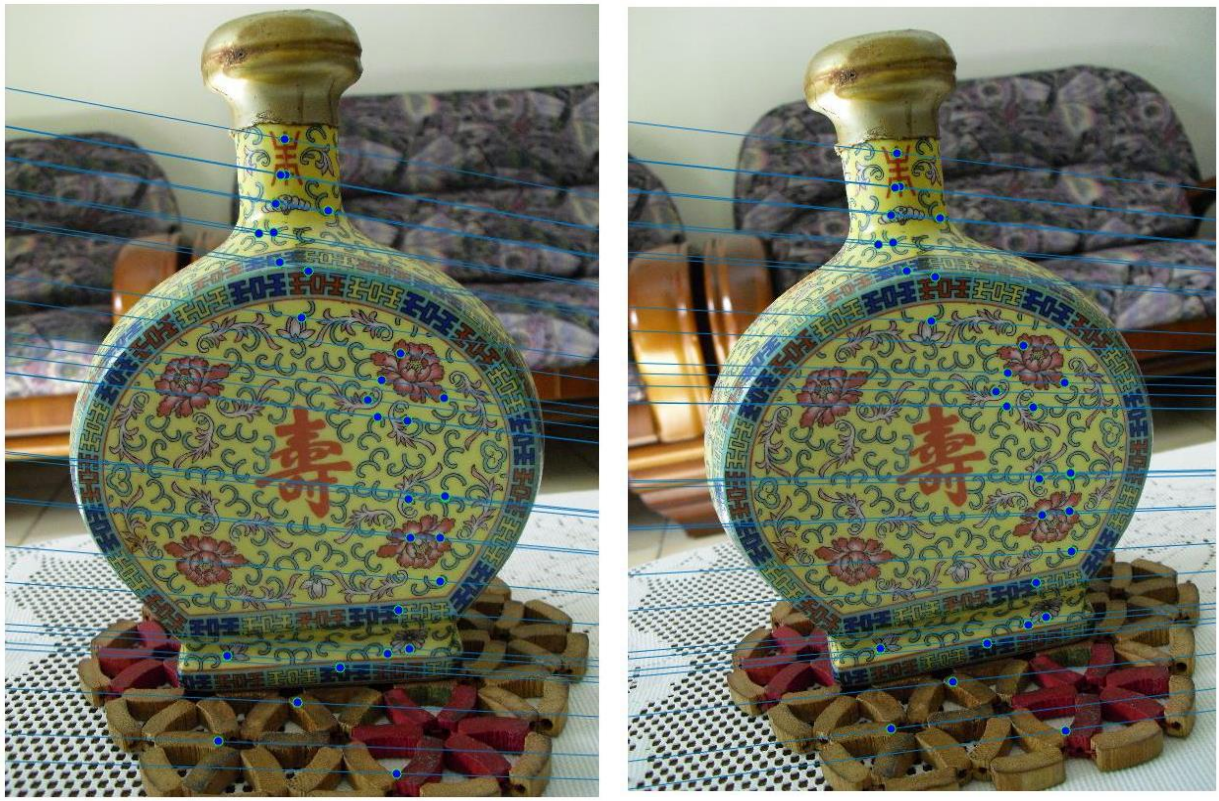


Figure 6.2 A set of images of a Chinese wine bottle. Another example of images and their epipolar lines visualised in the other image, derived from the estimated fundamental matrix.

6.3 The essential matrix

The essential matrix is a matrix with 6 degreess of freedom, where Hartley and Zisserman describe it as "the specialization of the fundamental matrix to the case of normalized image coordinates". This statement is based on the fact that during computation of the fundamental matrix we assumed that

we had a pair of calibrated cameras, whereas this assumption is now rejected in order to compute the essential matrix. The formula to compute the essential matrix is derived in this manner:

$$E = K'^T F K$$

Where K and K' are the matrices holding the intrinsic parameters of the two cameras used to take the photos (referred to as A in section 5, but since the book uses K, this is changed here), (Hartley and Zisserman, 2004). In the case where one camera is used, then K' = K, and we can use only one intrinsic parameter matrix. The intrinsic parameters can be estimated and refined using the method described in section 5 (with further improvements of course), but since the results were not optimal, the intrinsic parameters of the vase in Figure 6.1 were found online and used in this case. Therefore, using the above formula with K' = K and the fundamental matrix estimated in section 6.2, we can get the essential matrix.

The essential matrix can be further decomposed to retrieve the camera projection matrices, holding the rotation and translation characteristics associated with each viewpoint's position and orientation. As explained in the book, since there are ambiguities in the derivation of E from F, there are four possible choices for the construction of the second camera projection matrix. The first camera projection is set to $P_1 = [\, I \mid 0 \,]$ in order to derive the second one based on this, so we actually set it manually. The four possibilities for the second camera projection matrix are derived from the SVD of E and given the additional constraint of having two equal values in its diagonal, so as to only have 5 DOF and not 6, we have:

$$UDV^T = SVD(E) \ \ and\ forcing\ the\ constraint\ we\ get$$

$$E = U\ diag(1,1,0)\ V^T$$

The four possible derivations, given a second SVD on E, for the second projection matrix are:

$$P_2 = [\, UWV^T \mid u_3 \,] \quad or$$

$$P_2 = [\, UWV^T \mid -u_3 \,] \quad or$$

$$P_2 = [\, UW^TV^T \mid u_3 \,] \quad or$$

$$P_2 = [\, UW^TV^T \mid -u_3 \,]$$

Where u_3 is the 3rd column vector of U and W is manually set to equal:

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Now, we have our 4 possible solutions, and we have to test them in order to decide which one is the correct one, as only one will be valid. In order to do so, we need to perform triangulation using the projection matrices and the matched points, to retrieve a z-value, i.e. the depth of the points. Triangulation will be discussed in section 7 where the topic is reconstruction of a 3D scene from triangulation. In their book, Hartley and Zisserman suggest that using a single point and testing whether the point is in front of both cameras, will result in the correct selection of the corresponding second projection matrix. In this implementation, all retrieved 3D points are used as reference and the depth of each point is cross checked with the position of both the camera views to see if they are in front of them, thus selecting the projection matrix resulting in the most points in front of both views. In order to find the depth of the points we need the camera projection matrices to find the depth at which each

camera is located. To do so we can refer to the section in the book called “Depth of points”, page 162 (Multiple view Geometry 2nd Edition). In this section, we can find the following formula:

$$depth(X;P) = \frac{sign(\det M)\, w}{T\, ||m^3||}$$

M is obtained by turning the camera projection matrix into a calibrated version, since it only holds the extrinsic parameters, $P = [R|t]$. Therefore, $P = K\,[R|t]$, where M consists of the first 3 columns of P. T is the fourth dimension of a triangulated 3D point before turning it back into homogeneous coordinates, i.e. (X, Y, Z, T). Here is a code snippet of the function used to calculate the depth and decide whether they are in front or behind the two cameras:

```
function [pif1,pif2] = find_points_in_front(points3D,K,P1, P2)
% based on Multiple View Geometry page 162, Hartley and Zisserman
% points3D are actually 4D (not in homogeneous coordinates)
    R1 = P1(1:3,1:3);
    R2 = P2(1:3,1:3);

    % Pass the calibrated camera projection matrices to find the position
    % of the cameras.
    C1 = get_camera_coords(K*P1);
    C2 = get_camera_coords(K*P2);

    pif1 = 0; % points in front of the first camera
    pif2 = 0; % points in front of the second camera
    for i = 1:size(points3D,1)

        temp = (points3D(i,1:3)-C1)*R1(3,:)'; % this is w from formula in the book
        temp2 = (points3D(i,1:3)-C2)*R2(3,:)'; % w for the second camera

        depth = (sign(det(R1))*temp)/points3D(i,4)*norm(R1(3,:));
        depth2 = (sign(det(R2))*temp2)/points3D(i,4)*norm(R2(3,:));

        if depth>0
            pif1 = pif1 + 1;
        end
        if depth2>0
            pif2 = pif2 + 1;
        end
    end
end
```

With the get_camera_coords(P) function retrieving the camera centre using the projection matrix:

```
function C = get_camera_coords(P)
    % Multiple View Geometry page 163
    x =  det([ P(:,2), P(:,3), P(:,4) ]);
    y = -det([ P(:,1), P(:,3), P(:,4) ]);
    z =  det([ P(:,1), P(:,2), P(:,4) ]);
    t = -det([ P(:,1), P(:,2), P(:,3) ]);

    C = [x/t y/t z/t];

end
```

Discussion:

Even though the book was followed step-by-step there seemed to be ambiguities when searching for the projection matrix leading to the most points in front of both cameras and therefore an inability to

find the correct solution. In order to find the correct solution manually, all projection matrices were used in the reconstruction procedure in section 7, in case no decision is made on the best camera projection matrix. Furthermore, visualisation of the results is to be shown again in section 7 since without the correct projection matrix the reconstruction of a 3D scene via triangulation will produce false results. Feedback on how to improve on this implementation and further fix such problems will be kindly appreciated since time run out while trying to debug and solve any issue. Below is a code snippet showing the compromise required, i.e. if no projection matrix resulted in the most points in front of the camera, then all are returned and all are visualised during reconstruction.

```
[pif11,pif12] = find_points_in_front(points3D1,K,P1, P21);
[pif21,pif22] = find_points_in_front(points3D2,K,P1, P22);
[pif31,pif32] = find_points_in_front(points3D3,K,P1, P23);
[pif41,pif42] = find_points_in_front(points3D4,K,P1, P24);

if pif11 == max([pif11,pif21,pif31,pif41]) && pif12 == max([pif12,pif22,pif32,pif42])
    P2 = P21;
elseif pif21 == max([pif11,pif21,pif31,pif41]) && pif22 == max([pif12,pif22,pif32,pif42])
    P2 = P22;
elseif pif31 == max([pif11,pif21,pif31,pif41]) && pif32 == max([pif12,pif22,pif32,pif42])
    P2 = P23;
elseif pif41 == max([pif11,pif21,pif31,pif41]) && pif42 == max([pif12,pif22,pif32,pif42])
    P2 = P24;
else
    P2(:,:,1) = P21;
    P2(:,:,2) = P22;
    P2(:,:,3) = P23;
    P2(:,:,4) = P24;
end
```

7. Triangulation and 3D reconstruction

Given that we have corresponding points from two images, i.e. the matched features and their corresponding locations in both images, and the two camera projection matrices from section 6.3, as well as the intrinsic parameters of the cameras, we can proceed with retrieving a depth for each point correspondence. This process is called structure from motion and it involves the triangulation of point correspondences given their epipolar geometry. As with every computer vision technique, triangulation can be done using a number of algorithms. In this implementation, the algorithm used is referred to as the homogeneous method of Direct Linear Transformation (DLT), also known as linear triangulation (Hartley and Zisserman, 2004) page 312.

7.1 Linear Triangulation

The ultimate goal of linear triangulation is to project the points from each image, given the corresponding camera's projection matrix, to obtain a system of linear equations. The depth of each triangulated 3D point is where the two rays coincide, or at least pass over each other, since noise will lead to inaccurate projections. Therefore, the mappings can be formulated in this way:

$$\boldsymbol{x}_1 = P_1 \boldsymbol{X} \quad and \quad \boldsymbol{x}_2 = P_2 \boldsymbol{X}$$

Where $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are corresponding points, P_1 and P_2 are the camera projection matrices, and finally $\boldsymbol{X}$ is the corresponding triangulated 3D point. These equations can be further combined to form a system $A\,\boldsymbol{X} = 0$, a system of linear equations with respect to X. This entails that a solution to X is given by SVD on A and getting the "unit singular vector corresponding to the smallest singular value of A", just as described in the book. Using the cross product of the above mappings, $x \times PX = 0$, for each point we can deduce 3 linear equation of which 2 are linearly independent, page 312 of the MVG book. The A matrix can then be constructed for each corresponding point set in the following manner:

$$A = \begin{bmatrix} x_1\, p_3^T - p_1^T \\ y_1\, p_3^T - p_2^T \\ x_2\, p'^T_3 - p'^T_1 \\ y_2\, p'^T_3 - p'^T_2 \end{bmatrix}$$

Where $\boldsymbol{x_1} = (x_1, y_1, 1)$ and $\boldsymbol{x_2} = (x_2, y_2, 1)$ are now in homogeneous coordinates, p_i^T and p'^T_i are the row vectors of the projection matrices P_1 and P_2 respectively. So, for every point correspondence pair, a matrix A is calculated. Obtaining the last column of V from $UDV^T = SVD(A)$ will return the 4D vector containing the triangulated point. By dividing every column by the fourth one, and retrieving only the first 3 columns, we obtain the corresponding 3D point which is now of the form **X** = (X,Y,Z). A summary of the implementation is shown below:

Linear Triangulation (**x1, x2,** P1, P2): (Inputs are the corresponding points pairs and projection matrices)

1. Turn the corresponding points to homogeneous coordinates by concatenating a column of ones: $(x, y) \rightarrow (x, y, 1)$
2. For each corresponding points pair $\boldsymbol{x_1} \leftrightarrow \boldsymbol{x_2}$, now in homogeneous coordinates:
    a. Construct matrix A as shown above, using the respective rows from P_1 and P_2.
    b. Solve for **X** from the system of linear equation resolving to $A\,\boldsymbol{X} = 0$ by performing $UDV^T = SVD(A)$ and retrieve the unit vector with the smallest singular value of A, i.e. the last column of V^T.
    c. Divide **X** by its last column to turn it back to homogeneous coordinates.
3. Return the first 3 columns of each **X**, now a point in 3-dimensional space.

The code snippet corresponding to the above algorithm:

```
function points3D = linear_triangulation(points1, points2, P1, P2, mode)
% Correspondence points points1 and points2,
% together with the two camera projection matrices P1 and P2,
% The mode is used to decide whether the points are turned back into
% homogeneous coordinates if mode== 1 then points3D are left in 4D
% otherwise if mode ==2 then divide by fourth column and get 3 first
% column.

    % Homogeneous transformation
    ones_col = ones(size(points1,1),1);
    X1 = [points1 ones_col];
    X2 = [points2 ones_col];

    for i = 1:size(X1,1)
        % Hartley and Zisserman book page 312 (Second edition)
        A = [X1(i,1)*P1(3,:) - P1(1,:);
             X1(i,2)*P1(3,:) - P1(2,:);
             X2(i,1)*P2(3,:) - P2(1,:);
             X2(i,2)*P2(3,:) - P2(2,:)];

        [~,~,V] = svd(A);
        temp = V(:,end);
        points3D(i,:) = temp;

    end

    % Homogeneous transformation
    if mode==2
        for j = 1:size(points3D, 1)
            points3D(j,:) = points3D(j,:)./points3D(j,4);
        end
        points3D = points3D(:,1:end-1);
    end
end
```

The parameter called “mode” is used to specify how the points are returned. This is mainly due to the fact that in order to find the depth of each point from section 6.3, the fourth dimension was required.

7.2 3D Reconstruction

Based on the assumption that the camera projection matrices found as the correct pair in section 6.3 would produce a valid triangulation, and from the 3D points acquired, we can just plot the 3D points to reconstruct the 3D scene. Due to the ambiguities of choosing the best projection matrices as mentioned in section 6.3, the reconstruction of the scene from the triangulated points did not always make sense. Nonetheless, all the results will be shown, even some of the faulty cases, to thoroughly evaluate this implementation. Before passing the selected projection matrices in the triangulation method, they are first calibrated using the intrinsic parameter matrix, K, since they have the form $K^{-1}P = [I|0]$ and $K^{-1}P' = [R|t]$. To get the final projection matrices we just:

$$P = K\,[I|0] \;\; and \; P' = K\,[R|t]$$

The calibrated projection matrices are then passed as parameters in the triangulation function, to retrieve a 3D point from a pair of 2D matched points.

The following figures will show some of the results from the reconstruction procedure:

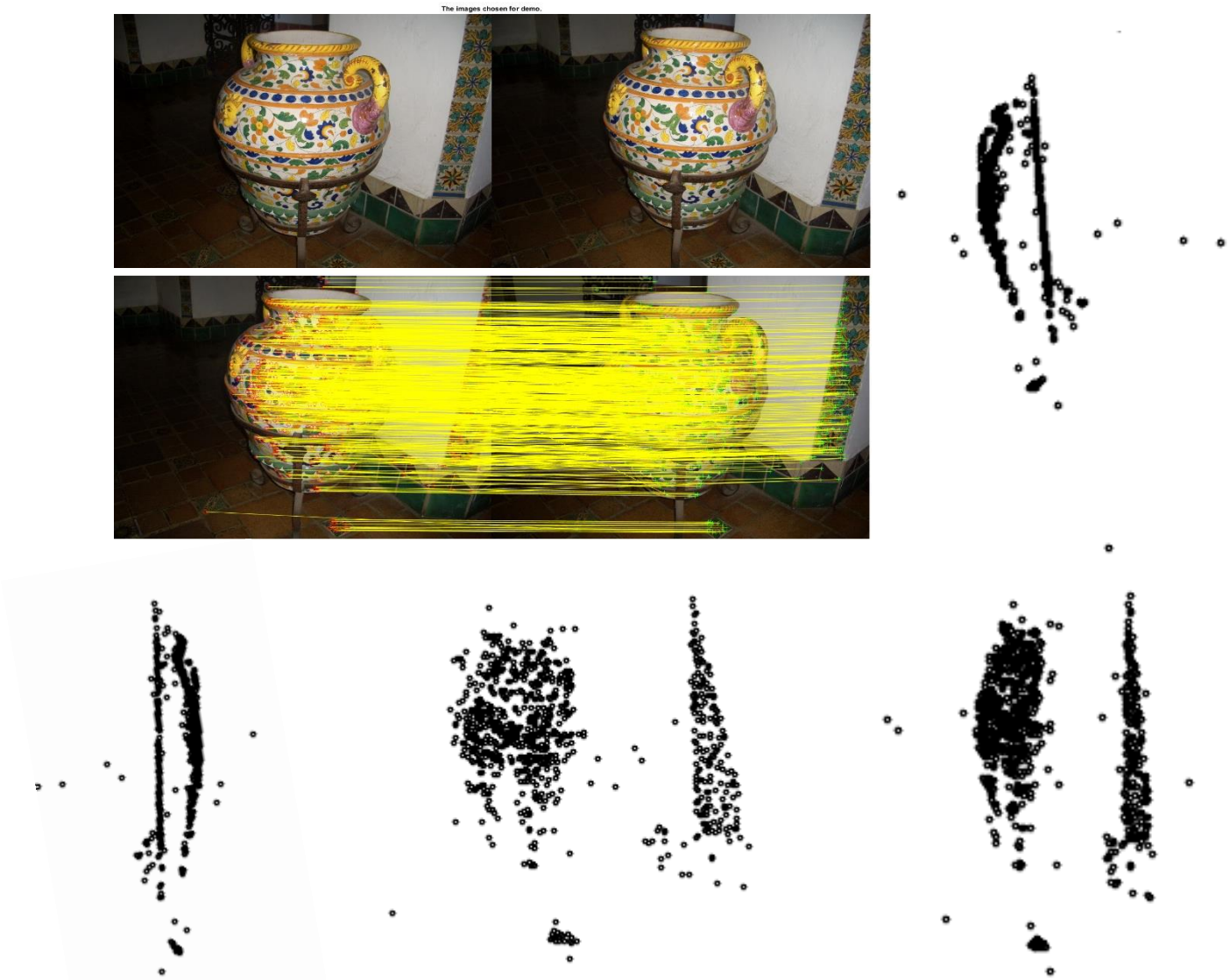


Figure 7.1 On the top left we can see the original images and their matched features. The 3D representation of the scene is reconstructed as a point cloud of triangulated points. The top right and bottom left images show the pose of the vase when looked from the side, i.e. profile, The middle one on the bottom shows the scene when looking at it head on and the final point cloud (bottom left) is from a diagonal view. The depth is incorporated and in these case the correct projection matrix was returned from the method in section 6.3.

This example was one that was by far the most successful, with a point cloud that can be clearly evaluated. Outliers exist, as you can see, which are actually points that were not triangulated in the correct manner. Nonetheless, the number of inliers overcomes the number of outliers and the results provide a good representation of the 3D scene. Please keep in mind that the output was rotated to have the same orientation as the images. This was the result from fear of trying to change the order in which the dimensions are represented, and has not been messed with to keep obtaining successful results. Figure 7.2 shows the actual output from MATLAB. Additionally, the dimensions came out to be normalised, but was left in this state in order to have something to show. After the retrieval of the scene 3D coordinates, a way to evaluate the selection of the correct camera matrix is by re-projecting them back into 2D points based on the corresponding camera projection matrix, and establish an error metric. This is called the reprojection error and it actually measures the Euclidean distance between the re-projected 2D point and the actual 2D point. A function was derived that performed the opposite of the mapping mentioned before to return the re-projected 2D points. The sum of all the distances is

measured as well, but due to the large resolution of the images, this came out to be large. Visually the reprojection error seemed appropriate and can be visualised in Figure 7.3.

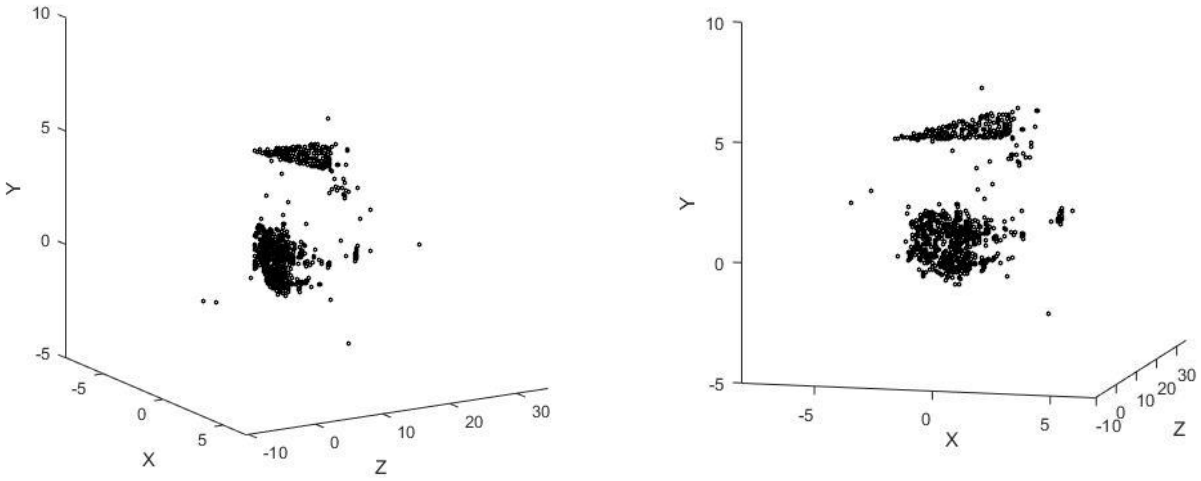


Figure 7.2 The output from the triangulation and point cloud reconstruction procedure (original). The scene is rotated 90 degrees anticlockwise, and the axes are normalised. To have results to show these were left unattended.

Figure 7.3 Reprojection error visualisation. The red crosses correspond to the re-projected 2D points from the 3D points given the camera projection matrix, while the green circles are the actual points of that image. Visually the reprojection error came out to be very good.

Below is the code snippet of the find_rep_error function:

```
function [error, projected2D] = find_rep_error( P, points3D, points2D)

proj = P*[points3D ones(size(points3D,1),1)]';
proj = proj';
u = proj(:,1)./proj(:,3);
v = proj(:,2)./proj(:,3);
error = sum(sqrt(power(u-points2D(:,1),2) + power(v-points2D(:,2),2)));
projected2D = [u v];

end
```

Nonetheless, not all results were successful, and sometimes the procedure must be run multiple times to visualise a valid 3D reconstruction. This problem is based on whether the correct projection matrix is derived from section 6.3, which would otherwise lead to upside-down or inside out reconstruction, as can be seen in Figure 7.4.

Figure 7.4 An example of an inside-out result. This can be seen from the triangle of points on the left of the first point cloud, since it is on the wrong side of the vase. The depth is incorporated better in this reconstruction (see middle and right image) and even the handle of the vase is more distinct, but it is a false result nonetheless.

Triangulation is shown to have worked and the depth of points in the world are found, even though not consistently. Please keep in mind that when running the triangulation and reconstruction in this implmentation the initial results might not be optimal, and a 2nd run is recommended to visualise a nice point cloud. There are some times that the implementation returns a flat point cloud that lies on a plane without any depth and the problem causing that is mentioned earlier.

8. Conclusion

With the final part being triangulation, we have reached the end of the report. In this report we have discussed and explained several algorithmic techniques from the field of computer vision and image processing, namely convolution, feature extraction, matching, image stitching and homography estimation, as well as many important aspects from stereo vision. The whole experience was very intriguing, and under different circumstances the implementations of the various computer vision procedures explained throughout this report, would have been dealt with in a more relaxed and focused manner. As time was the enemy, some parts were rushed and not fully completed, but the learning experience, both in the technical but also the time management sector, was crucial for what the future brings. Computer vision is a very important field and plays a huge role in the Digital Entertainment sector, therefore any feedback provided on how to fix some of the issues mentioned in the report would be kindly appreciated.

9. References

Canny, J. (1986). A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6), pp.679-698.

Criminisi, A. (1997). *3 Computing the plane to plane homography*. [online] Robots.ox.ac.uk. Available at: http://www.robots.ox.ac.uk/~vgg/presentations/bmvc97/criminispaper/node3.html.

Derpanis, K. (2010). Overview of the RANSAC Algorithm. *York University - EECS*, [online] pp.1-2. Available at: http://www.cse.yorku.ca/~kosta/CompVis_Notes/ransac.pdf.

Fischler, M. and Bolles, R. (1981). Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, [online] 24(6), pp.381-395. Available at: https://dl.acm.org/citation.cfm?doid=358669.358692.

Gonzalez, R. and Woods, R. (2002). *Digital image processing*. Upper Saddle River, N.J.: Prentice Hall, pp.276-278.

Harris, C. and Stephens, M. (1988). A Combined Corner and Edge Detector. *Procedings of the Alvey Vision Conference 1988*. [online] Available at: http://www.bmva.org/bmvc/1988/avc-88-023.pdf.

Hartley, R. and Zisserman, A. (2004). *Multiple view geometry in computer vision*. 2nd ed. Cambridge [u.a.]: Cambridge Univ. Press.

Richardt, C. (2017). CM20219 - Fundamentals of Visual Computing Course Notes. [online] pp.1-84. Available at: https://moodle.bath.ac.uk/pluginfile.php/966584/mod_resource/content/6/CM20219%20Course%20Notes%202017-8%20%28version%202017-10-06%29.pdf.

Richardt, C. and Hall, P. (2017). CM50248 - Visual Understanding 1: Course Notes. [online] pp.1-34. Available at: https://moodle.bath.ac.uk/pluginfile.php/850436/mod_resource/content/6/CM50248%20Notes%20%282017-10-01%29.pdf.

Stachniss, C. (2017). *Photogrammetry Course*. [video] Available at: https://www.youtube.com/playlist?list=PLgnQpQtFTOGRsi5vzy9PiQpNWHjq-bKN1.

Sur, F., Noury, N. and Berger, M. (2008). Computing the Uncertainty of the 8 point Algorithm for Fundamental Matrix Estimation. *Procedings of the British Machine Vision Conference 2008*. [online] Available at: http://www.bmva.org/bmvc/2008/papers/269.html.

Weisstein, E. (2018). *Convolution Theorem -- from Wolfram MathWorld*. [online] Mathworld.wolfram.com. Available at: http://mathworld.wolfram.com/ConvolutionTheorem.html.

Zhang, Z. (2000). A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(11), pp.1330-1334.