1. Introduction and the Problem

The task was to write code which generated a seed estimate of, and then optimised, the intrinsic model of a pinhole camera for use in computer vision. The aim was to return a reasonable estimate of the camera's **KMatrix**, a 3x3 matrix which encodes parameters of the camera.

This report will summarise my solution to the problem, including the data structures used and the flow of information through the code. I also discuss the mathematical algorithms used within my code (namely RANSAC and Levenberg-Marquardt minimisation) and the considerations I made while realising them. I will then briefly outline some notable aspects of my code. Finally, I discuss the testing procedure and analyse how my code performed, concluding with subsequent steps that can be taken.

2.1. The Solution: Overview

In this task, data from real images was not used. Instead, a camera, calibration grid and the images of the grid were **simulated** by first building a model of a camera – in this case, the simulated camera was loosely based on that of a Samsung Galaxy S5 mobile phone.

A set of correspondences was generated between [x,y] points on the object being viewed and [u,v] pixel coordinates on the camera's sensor chip. Linear algebra was used to estimate homographies between the object and the chip. Constraints on the perspectivity between the camera's unit plane (the camera's frame of reference) and the object's frame were then used to extract a seed estimate of the camera's KMatrix, which was passed to the optimisation algorithm, returning a final estimate of the camera's model. This KMatrix was then compared to the initial model used in the simulation.

It is useful, here, to highlight that the homography (a description of the affine transformation between two planes) is split into two parts: the extrinsic model (the perspectivity), describing the relative location of the camera to the object; and the intrinsic model (the KMatrix), which defines the transformation between the unit plane and the camera's sensor chip.

2.2. Simulating the Camera

First, I built a model of my camera; I chose to store all of the information on the camera (the KMatrix and then, separately, the sensor chip width and height) in a structure. This allowed me to pass all camera model information to successive functions succinctly. Functions were then built which positioned the camera and an object in space by creating (almost) random homogeneous transformations, thus implicitly building the perspectivity. The functions were built separately so the camera could be oriented to be pointing at the object and placed a set distance away. To test these functions, a cube was constructed and then 'viewed' by the camera according to the above transformations, with the output on the camera's sensor chip displayed (Figure 1). I also built a function which showed a 3-D representation of the camera viewing the cube (Figure 2), providing a sanity check on the result.

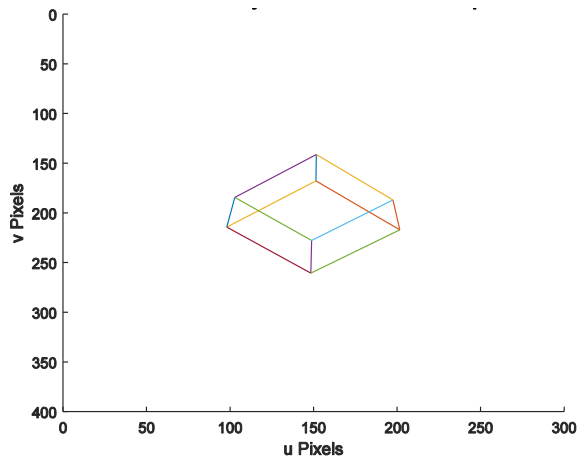

Figure 1: A simulation of a camera's view of a cube

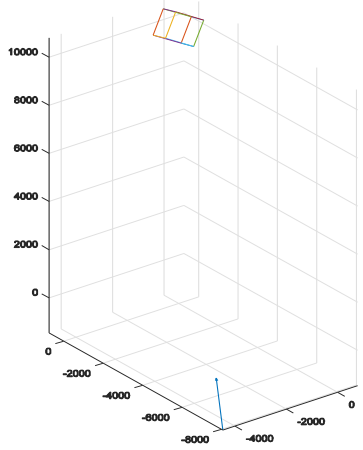

Figure 2: Visualisation of camera looking at a cube

2.3. Simulating a calibration grid and estimating the homography:

A single homography estimate was achieved by a (simulated) image of an object's calibration grid, which would in reality be a checkerboard of squares. This grid was built by defining the corner points ([x,y] coordinates) of each of the constituent squares, and consisted of one hundred squares per row or column. This object was then positioned randomly in space, with the 'PositionCamera' function (referred to earlier) adapted to orientate the camera such that it was close to perpendicular to the grid's plane and positioned so the grid filled the camera's image. Again, I built a function, similar to that in Figure 2, which visualised the scenario.

A set of 2-D homogeneous Points was then returned by transforming the grid through the generated perspectivity and KMatrix. These Points were scaled by the homogeneous multiplier to recover a set of [u,v] pixel coordinates in the camera's sensor plane. The addition of Gaussian noise and outliers to these points resulted in a reasonable simulation of an image of a calibration grid as seen by the camera. The output from this stage was a matrix of correspondence vectors of the form [u,v,x,y]', named 'Correspond'. It is these vectors that would in reality be generated by a vision algorithm, from a real image of a calibration object.

This matrix was then passed through an iterative RANSAC algorithm which attempted to return the homography between the grid plane and the camera's sensor plane. The algorithm worked by selecting four points at random from 'Correspond' and building a full rank regressor matrix, φ, and a data vector, **p**, such that:

$$\varphi \mathbf{h} = \mathbf{p} \qquad (1)$$

'**h**' was a vector containing the eight independent parameters of the homography to be estimated. The homography is a 3x3 matrix which, after scaling the bottom right entry to 1, has eight independent entries. The regressor was then inverted using the quick 'backslash' (`mldivide`) method to return an estimate of the homography. [u*, v*] estimates of the 'true' [u,v] points in 'Correspond' were generated by running the grid points through this homography, and errors measured against the true values, with sufficiently accurate estimates placed within a consensus set. This process was repeated a number of times, and the largest consensus retained and passed on.

The best consensus set was used to build a much larger regressor matrix, thus making equation (1) over-constrained. A pseudo-inverse using the singular value decomposition of the regressor was then used to return a least squares estimate of the homography.

2.4. Estimating the KMatrix: Noting: Homography = KMatrix*Perspectivity (2)

The orthogonality properties of the perspectivity can be used to manipulate equation (2) to give two independent equations relating the homography elements and the elements of a matrix, Φ, such that: $\phi = \left(\mathbf{K}^{-1}\right)^{T} \mathbf{K}^{-1}$ (3)

Thus, the use of three different homographies (from three different images) is sufficient to compute the six independent elements of Φ, from which the KMatrix can be estimated using the Cholesky Product. Upon recommendation from the notes ([1]), I used a minimum of six images for the KMatrix estimation. Thus data on six different images and their homographies ('Homography', 'Correspond', 'CorrespondIndex (*defining the best consensus set*)') needed to be passed on from the homography estimate block of the code. This data was stored in a cell array, which was preferable to a structure to simplify indexing (for example, in loops). This cell array, along with the KMatrix estimate, was passed on to the optimiser.

2.5. Optimising the KMatrix:

At this point, the code had returned an initial estimate of the intrinsic model of the camera, the KMatrix. The optimiser code computed a cost function related to the error vector between sensor points 'predicted' by the estimated KMatrix and the 'true' points stored in 'Correspond'. The purpose of the optimiser was to then minimise the cost function with respect to the 11 parameters involved in computing the predicted points – five parameters related to the KMatrix and six to the extrinsic model. However, data based on a minimum of six images is available at this point in the code, therefore the optimiser minimises a cost function based on the parameters relating to all of them (the number of parameters is equal to 5+6*NumberOfImages: the KMatrix parameters are common to all images, with six additional parameters for the extrinsic model of each image).

The minimisation algorithm used was Levenberg-Marquardt, an amalgamation of steepest descent and Newton steps. The parameters here were stored, again, in cell arrays, with one row for parameters relating to each image. The cell arrays were especially advantageous here as the various components of the L-M optimisation (such as the Jacobian, gradient and error vectors) were built with respect to all of the images at once, resulting in numerous cascaded loops. In this situation, the use of a structure array would have made the indexing especially cumbersome.

3.1. Analysis of algorithms - RANSAC

Assuming the initial homography from the sample of four is reasonably accurate, RANSAC is a robust method for removing outliers from a dataset (see section 5.1 for further discussion of this).

An outlier point (as set in the process of building the noisy 'Correspond') is placed anywhere on the camera chip, which has an area of the order 10^6-10^7 pixels. As the maximum admissible error for inclusion in the consensus set was 5 pixels, it can be seen without detailed analysis that an outlier point has a very low probability of being included in the best consensus set.

Contrastingly, RANSAC itself does nothing to mitigate the effect of noise. The standard deviation of the noise was generally lower (mostly between 0 and 3 pixels) than the maximum admissible error, meaning almost all purely noisy points were included in the consensus set. However, observation showed that the estimates of the homographies were generally very close to the simulation. This was due to the method used to calculate the final homography estimate from the best consensus set. The SVD 'pseudo-inverse' method of the data resulted in a least squares estimation of the true homography, which mitigated the effect of the noise. I believe this to have been especially effective as the noise was white and Gaussian (it had no systematic component).

However, it should be noted that there is no upper limit on how many iterations of RANSAC need be performed before the optimal model is struck upon (aside from all possible combinations of points being selected for the initial estimate). As such, the method can be computationally expensive and slow, though I argue that, in a real application, robustness is of greater use than speed.

3.2. Levenberg-Marquardt

$$\left(\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}\right)\delta\mathbf{p} = -\mathbf{J}^T\mathbf{e}. \qquad (4)$$

Equation (4) defines the Levenberg-Marquardt equation. **J** is the Jacobian matrix (discussed later in this section), **dp** is the direction of parameter change for minimisation, and **e** is an error vector between 'true' values and the estimates according to the current parameters.

This method is good for reaching local minima, and is a linear combination of steepest descent and Gauss-Newton in an attempt to mitigate the weaknesses of both – slow convergence in narrow canyons in the case of the former, and high sensitivity to the choice of seed estimate for the other. My experience of the algorithm, for this application, is that it is still highly sensitive to choice of seed estimate, though it converges with relatively few iterations if the seed is good enough. The

algorithm has flexibility in choice of adjustment of the damping parameter (μ), though this was not explored in this project – I used the method proposed by Madsen, Nielsen and Tingleff [2].

I reason that the main problem with the optimisation problem here is the presence of multiple minima. The problem is non-linear with respect to the (at least) 11 parameters to be optimised and as such the problem space is likely littered with stationary points. This means that this algorithm will, as would most, converge to a local minimum instead of the global one. As a result, the optimised result, though related to a lower value of the cost function, may actually be further away from the true camera model.

At the moment, the code tries to provide a seed thought to be a good estimate of the KMatrix – one that will converge to the true camera model through the optimisation code. However, it was seen that with only a slightly poorer seed (for example, with slightly more noise), the optimisation would converge to the wrong model. An alternate method would be to feed the optimiser with seeds periodically scattered around the parameter space. A fine enough spacing of the seeds would theoretically guarantee a convergence on the optimal model. This is, however, a significantly more computationally expensive method.

I, at first, questioned the need to optimise with respect to extrinsic parameters as well as the intrinsic ones, as it is ultimately only the intrinsic ones of concern to us. I reasoned that minimising with respect to only the intrinsic parameters would greatly reduce the dimensions of the space in which the optimiser was moving, and thus reduce the complexity of the problem. However, I ultimately realised that such a method would only give the minimum in one multi-dimensional surface (another form of ‘local minimum’).

The stopping criterion suggested in the notes (page 69 of [1], stopping when the gradient is near to zero) will result in the code terminating at any local maxima or minima. As such, I did not think this to be the best method. I considered stopping when the error had dipped below a certain acceptable value. Observation showed that a cost function value of less than roughly 0.001 resulted in a KMatrix that was close to the true model. This value is, of course, a function of the number of points in the consensus set and the simulated camera, and a further development of the

code would be to (empirically) model this function. There were, of course, times when this method resulted in the code getting ‘stuck’ as it converged to local stationary point in smaller and smaller steps. In these cases, I experimented with placing a ‘break’ condition if the gradient was too small or letting the code run to a pre-set maximum number of iterations (I found the latter did not take too much longer, so this is the method in my final code). Another method was to ‘jump’ the parameters by a random (but significant) step if the code was ‘sticking’ at a local stationary point – this method was predictably unsuccessful, due to the size of the parameter space.

Finally, I discuss the Jacobian matrix. My function ‘SingleImageJacobian’ builds the Jacobian relating to one of the images, before another block of code collates the results from all of the images. The Jacobian contains partial derivatives of ‘predicted measurement functions’ (see page 67 of [1]) with respect to all parameters that are inputs to the cost function, which were computed using a first-order, forward difference method. The perturbations used in this computation were crucial to obtaining a reasonable approximation of the derivative – too small and the code would be poorly conditioned, too large and the approximation would be inaccurate. Ultimately, I chose step sizes that were 0.01% of the parameter value, except for the skewness, which I incremented by 0.001 always. My check on the suitability of these steps was a by-eye analysis of the partial derivatives with respect to the KMatrix parameters. I noted that all of the derivatives of the translations (entries (1,3) and (2,3) in the KMatrix) with respect to themselves were 1.00, as one would expect. The ‘measurement functions’ split into two classes, those related to the ‘u’ coordinate on the camera’s sensor chip and those related to the ‘v’. I noted that the derivatives of ‘v-measurement functions’ with respect to KMatrix parameters related to the ‘u’ coordinates were 0.00, again as would be expected. I took these two indicators, along with the observation that all derivatives were of a similar order, that my perturbations were well chosen.

4.1. Implementation - error checking

I checked for errors consistently throughout my code. As far as possible, if there was an unexpected output from a block of code, I would set the output to zero and pass it on, flagging the occurrence in the command window using `fprintf` – this way I could see if the rest of the code worked. Alternatively, I used the `error` function in MATLAB, however this terminates the code,

and I reserved it only for occasions when the code could not continue (for example, if there was an invalid set of input parameters at the beginning).

I made sure all of my functions had error checks on the inputs. For example, I wrote a 'TestTransformMatrix' function which ensured a valid size and bottom row of a 4x4 homogeneous transformation (such matrices were passed as inputs to many functions).

4.2. Interesting Code

One individual function in my code was my 'PositionCamera' function. The purpose of this function was to orientate the camera to be nearly perpendicular to the plane of the grid, and position the camera a specified distance from it. To ensure a reliable perturbation from perpendicular, I used Rodrigues' method (page 25 of [1]) to rotate the negative z-axis of the object's frame by a random angle (between 0 and 15 degrees) about a random axis in the object's plane. I generated the axis by a random combination of unit vectors in the x and y directions in the object's plane.

I would also mention the way in which I built my array of grid points, which had to define the corners of all squares on the two dimensional grid. This is best explained with the following code ('N' is the number of tiles in one row or column of the grid, 'GridWidth' the length of one tile, 'GridPoints' the matrix containing the [x,y] coordinates of the corners).

```
%The grid is defined by all permutations of the following two vectors
x = linspace(-GridWidth/2, GridWidth/2, N+1);
y = linspace(-GridWidth/2, GridWidth/2, N+1);

for j=1:N+1;
    %Here we systematically fill in sections of length N+1 of GridPoints by
    %first defining all points with x coordinates x(1), then x(2), and so
    %on
    GridPoints(1,(j-1)*(N+1)+1 : j*(N+1)) = x(j);
    GridPoints(2,(j-1)*(N+1)+1 : j*(N+1)) = y;
end
```

5.1. Testing - RANSAC

The most significant marker of the success of the iterative process in RANSAC is the size of the best consensus set. A larger consensus set indicates a model can be returned which matches more of the data. Meanwhile, the number of times I chose to iterate RANSAC (again noting that there is no reasonable upper limit for this) was an important decision. I therefore ran a test of

RANSAC runs ('nRuns') against the size of the best returned consensus set (Figure 3). It can be seen that the rate of growth decays as the runs increase, with the graph indicating an asymptote. The graph indicates to me that an 'nRuns' value of about 30 is sufficient.

I would have expected the asymptote to be at a far greater value; my 'Correspond' matrix contained 10201 noisy points, and for the above test I set my probability of an outlier as 0.025, so a reasonable estimate for the number of points in the best correspond set (after sufficient RANSAC runs) might be 9946. With this in mind, also noting that the primary purpose of RANSAC is to remove outliers, I tested how the size of the consensus set varied as the probability of outliers was increased from 0 to 100% (Figure 4). It can be seen that the size of the consensus set falls to zero as the probability of outliers increases to 100%, as one would expect. I suggest that the reason for the lower than expected asymptote in Figure 3 is that any consensus set is generated from a homography based on only four noisy points. Thus the noise on any of those points will significantly bias the homography and thus reduce the size of the related consensus set.

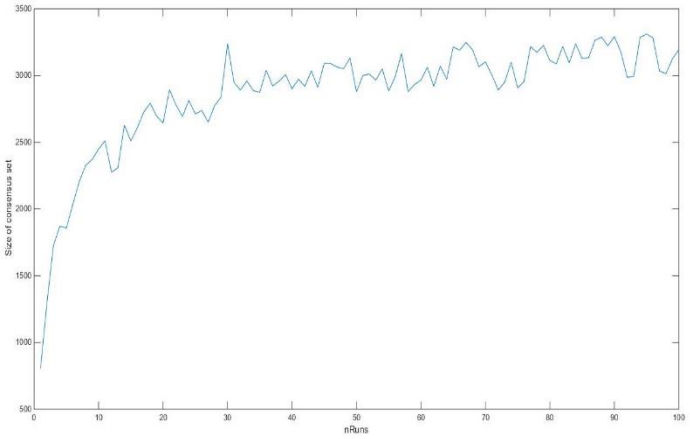

Figure 3: Size of the best consensus set against RANSAC runs

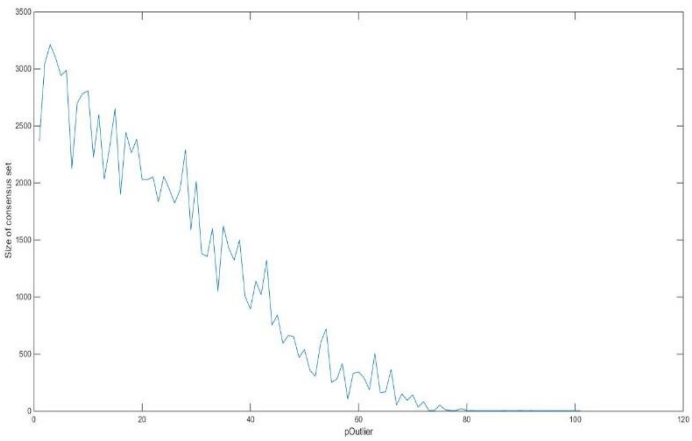

Figure 1: Size of best consensus set against probability of an outlier (%)

5.2. Testing – Levenberg-Marquardt

Finally, I tested the performance of the optimiser. As stated earlier, the optimisation block generally showed that an accurate seed estimate would result in an optimised model very close to the true value, but poor seed estimates did not. Thus I ran a test which could characterise the strength of the optimisation code alone. I varied the noise power on the 'Correspond' points and observed how much (as a percentage) the optimiser would reduce the value of the cost function (Figure 5). The large reduction in percentage error that can be achieved shows that the optimiser itself works well.

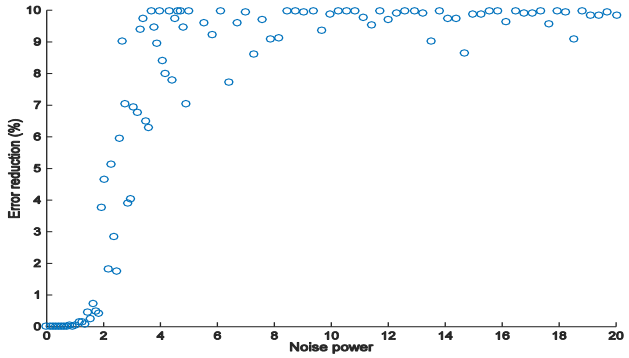

Figure 2: Noise power against percentage reduction of the cost function

The results show that a higher noise results in a higher percentage reduction of the error – at higher noise values, the cost function has a far greater initial value and there is more for work for the optimiser to do. Using a 3-D analogy, I would suggest that the cost function is shaped like a bowl with steep sides, and a relatively flat base containing numerous stationary points. This would explain why all (very) poor seed estimates result in great reduction of the cost function, to parameter values which can be significantly away from the true model.

6. Conclusion and further steps

At a lower level, I have developed a good optimiser, which can significantly improve models which have a large initial error. A RANSAC algorithm was also written which could, to a high degree of accuracy, estimate the homography between the camera chip and object. Ultimately, my code could reliably produce an accurate representation of a camera's intrinsic model given good conditions for the seed estimate, such as low noise. Noise might occur through shot noise in the pixels or through quantisation error, suggesting my code would work best with a high resolution sensor chip.

Looking ahead, a GUI could be developed for the code, where one could input the parameters for the camera simulation easily. A step further would be to develop code that could build a set of corresponding points from an input image of a calibration grid. This code could be built as a separate module, so long as the output was a matrix of vectors in the same form as 'Correspond'. A more complete code would automatically detect the corners of the grid (given one manual identification), while a more basic version could rely on manual selection of all grid corners. This would be a significant undertaking, though MATLAB has a useful Camera Calibration Toolbox [3].

[1] B1 Mini-Project – Calibrating a digital camera for computer vision Notes, Ron Daniel (2016)

[2] https://people.maths.bris.ac.uk/~maxmr/opt/nielsen_constrained.pdf, Madsen, Nielsen, Tingleff, (2004)

[3] https://www.vision.caltech.edu/bouguetj/calib_doc/#start, Jean-Yves Bouguet (1999)