**Introduction**

The goal of this project is to estimate the intrinsic model of a camera using as inputs pictures of Tsai Tiles. At the end, the program should receive at least 4 pictures of corner recognized Tsai Tiles and output its intrinsic camera model. To imitate the likely input to the program, a camera model and some pictures were simulated in the script. Since the pictures in real life will probably be noisy, those were also simulated. The report follows the recommendation on chapter 8 of the handout, analysing each algorithm as the program builds up. The sections covered in this report are: 1) Building Camera Model; 2) Estimating Homography from noisy measurements; 3) Estimating Homography from measurements with outliers; 4) Estimating Homography from noisy measurements with outliers; 5) Estimating KMatrix from Homographies; 6) Optimizing KMatrix using Levenberg-Marquadt Algorithm; 7) GUI and Likely Inputs 8) Conclusions and Suggestions.

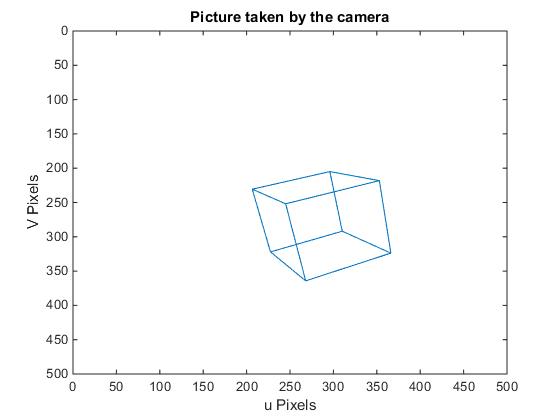
1. **Building Camera Model**

The first step is to create a camera model to imitate a real camera. In order to do that, three assumptions are made: 1) The camera is fixed focus; 2) The distance from the object to the camera is bigger than the camera’s Focus Length (which is in the order of 10s of millimetres); 3) The distortion caused by the camera’s lens is negligible. Assumption number one makes the K Matrix consistent with different photographs. Assumptions number two and three makes the Projective Geometry linear, and therefore much easier to implement. In the real world, those assumptions are often met.

There are 8 variables listed in page 12 of the handout to consider. When designing the code for the camera model, the strategy implemented was to use a list of vectors as inputs that encapsulates the same “families” of variables. For the Chip Width and Height, a 2D vector was used as input, and the same idea applies for the Effective Width and Height of a pixel and for the Principal Points Offset in the x and y directions. This way, the problem of a long list of parameters vs a long single vector of parameters posed in the handout is tackled. The output is the usual triangular matrix with the 5 variables and 1.0 at the bottom right corner.

To test the Camera Model, some pictures of a 1mx1mx1m cube taken at various random distances were simulated. Figure 1 shows one instance of that. The pictures are consistent with Projective Geometry in the sense that parallel lines appear to not be parallel. This instance needs to be look as if the biggest face of the cube is closer to the camera than the smaller one. In fact, if the cube was opaque, the smaller face wouldn’t be visible at all.

Figure Instance of Picture taken using Camera Model (500x500 pixels chip size, 0 skewness, 35mm focal length, 0.5x0.5 Points Offset, 0.08x0.08 Effective Pixel Size) of a 1mx1mx1m Cube placed at a random location from camera.



1. **Estimating Homography from Noisy Measurements**

In this part of the code, it is considered that the input to the functions is a matrix of points that were identified by a computer vision algorithm. We know that the computer vision algorithm will not be perfect and the points will be affected by noise. To deal with that, the picture of a noisy checkerboard was simulated using the MATLAB randn function (as suggested in the handout). To create the noisy measurements, there are two different methods, where (u,v) is the usual definition given in the handout:

Code 1 Code 2

For each (u,v): For each (u,v):

U = U + level\*randn(1,1); a = rand\*4\*pi

V = V + level\*randn(1,1); b = level\*randn(1,1);

U = U + b\*cos(a);

V = V + b\*sin(a);

Both of them add white noise to the measured points. The function *randn* will generate numbers with a variance of 1 and mean of zero, but it would be appropriate to have some control over the distribution of noisy points. The variable “level” adjusts for that. Using simple Probability Theory,

The error function that will be minimized use the Euclidian Distance (*D*) as a factor. This is where the difference in the two codes comes in. Using Code 2 was preferable because this way the Distance to the points are normally distributed in *X*. The distance between the points will be positive, so that the ideal distribution is:

This is called the Half-Normal Distribution, whose mean and variance are:

To reject the noisy, the Least Square Method (LSM) was implemented, which finds the set of points that will give the minimum error when compared to the actual points. If this algorithm is well implemented, it will follow the properties of a Half-Normal distribution when compared to the noisy image regardless of the level input. It also has to be considered that the bigger the level input, the bigger will be the difference from the estimated points to the actual points. Figure 2 shows the three matrices of points that will be used in the analysis of the algorithm.

Figure Noisy Image was created using Level equal to 1. The plot on the left is the simulated matrices of points identified by the computer vision algorithm. The plot in the centre is the points estimated using the LSM. The plot on the right is the precise points in case the computer vision algorithm had 0 noise output.

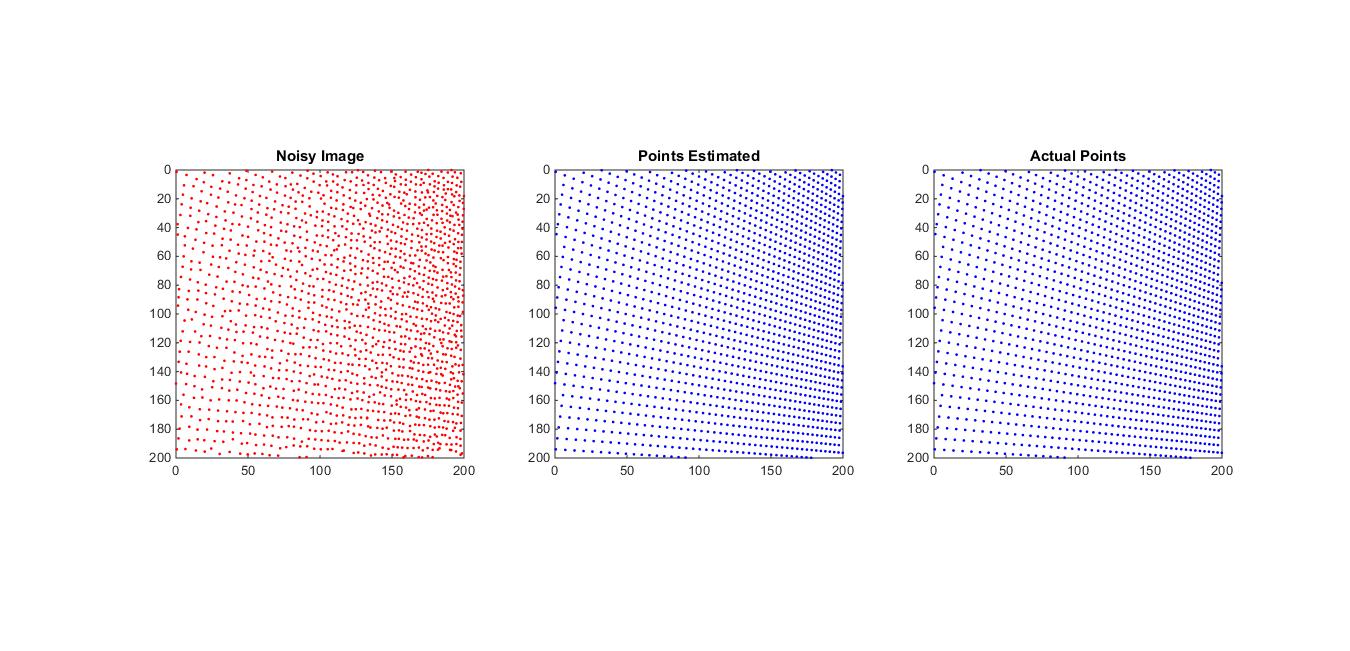


Table 1 is the result of taking 100 photographs for each Level of noise and analysing how the implementation of the Least Square Algorithm performs. The column “Level” is the level of the noise as described above. The column “Ideal Mean” and “Ideal Variance” are the expected mean and variance of the noise using the Half-Normal distribution equations. The columns “Mean Estimated” and “Variance Estimated” are the values found when considering the Points estimated by the LSM and the Noisy Image. We can see that the values are very close to the ideal ones, which is evidence that the modelled Noisy Image is working properly. The Columns “Mean Pixel Difference” and “Pixel Difference Variance” are the statistics found when comparing the Points estimated by the LSM and the actual points. Ideally, the last two columns should be zero for all values. It can be seen that those values increase with the increase of the Level variable, as expected. This table however shows that our LSM is well implemented and that the code is robust for any reasonable value of Level. The LSM implemented performs well for images affected purely by white noise.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Level | Ideal Mean | Ideal Variance | Mean Estimated | Variance Estimated | Mean Pixel Difference | Pixel Difference Variance |
|
| 0.1 | 0.0798 | 0.0036 | 0.0799 | 0.0036 | 0.0032 | 0.0000 |
| 0.5 | 0.3989 | 0.0908 | 0.3964 | 0.0889 | 0.0164 | 0.0002 |
| 0.8 | 0.6383 | 0.2326 | 0.6381 | 0.2319 | 0.0237 | 0.0003 |
| 1 | 0.7979 | 0.3634 | 0.8001 | 0.3610 | 0.0322 | 0.0007 |
| 2 | 1.5958 | 1.4535 | 1.6124 | 1.4532 | 0.0631 | 0.0040 |
| 3 | 2.3937 | 3.2704 | 2.4063 | 3.2622 | 0.1070 | 0.0108 |
| 4 | 3.1915 | 5.8141 | 3.1743 | 5.7126 | 0.1369 | 0.0132 |
| 5 | 3.9894 | 9.0845 | 4.0000 | 9.1600 | 0.2055 | 0.2261 |
| 10 | 7.9788 | 36.3380 | 8.0581 | 35.9530 | 0.7389 | 0.8746 |
| 15 | 11.9683 | 81.7606 | 12.1182 | 80.9609 | 1.2380 | 1.8097 |

Table Results of the Least Square algorithm implemented with different levels of noise

1. **Estimating Homography from measurements with outliers**

The computer vision algorithm will also interpret points that are not part of the grid as corners and return some outliers. The problem with outliers is that they are not linear or predictable in the same way that Gaussian-White Noise is. The approach to tackle this problem is to use the RANSAC algorithm. It will be assumed that the measurements have only outliers, without any noise for now. Later, the combined RANSAC LSM algorithm will be analysed.

For this test, all of the points are exact inliers, with exception of the outliers. Therefore, the reject error norm variable is not relevant, and can be ignored. The only things that are relevant are the percentage of outliers and the number of loops. We can analyse this simpler case statistically. Given *x* points in the grid, being *y* percent of those outliers, we can calculate the probability that the algorithm will randomly select any outliers (so 1 to 4 outliers), which is a simple case of probability without replacement. The following formula gives this probability:

For all purposes of this algorithm, the number *x* of points in the grid will be very big (usually in the 1000s), so we can assume that . Using this approximation, we get following equation:

|  |  |  |
| --- | --- | --- |
| Percentage of Outliers | Ratio at least 1 outlier | Expected Probability |
| 0.01 | 3.55% | 3.94% |
| 0.02 | 7.70% | 7.76% |
| 0.03 | 10.25% | 11.47% |
| 0.04 | 15.25% | 15.07% |
| 0.05 | 18.10% | 18.55% |
| 0.1 | 35.98% | 34.39% |
| 0.15 | 48.23% | 47.80% |
| 0.2 | 59.53% | 59.04% |

Table 2 Results of RANSAC for points without noise

Which is independent of the number of points *x*, so as long as this number is big (which is often met), we don’t need to be concerned about how many points will be in the photo. To test the accuracy of the RANSAC implementation in the program, the algorithm ran 20 times for each percentage (the numbers of loop for the RANSAC was set to 200). The Algorithm then returned a matrix of points, which was compared to the real points. If all of the points were in the exact same position (up to rounding errors and arithmetic precision), then the algorithm had picked 4 inliers. Otherwise, the Algorithm would have picked at least one outlier. Table 2 shows the results for different values of *y*. For the test condition, it was assumed a grid with 10,201 corners. The first column is the percentage of outliers; the second column is the ratio of the number of times the algorithm picked at least one outlier divided by the total number of runs; the third column is the expected probability using equation (3.2). From the table, it is evident that the algorithm is performing as expected. The values in the second columns are different than the ones in the third column because of the finite number of samples. It can be concluded that this algorithm as implemented works well and can be used reliably for more complex situations, which will involve using corrupted data with both noise and outliers.

1. **Estimating Homography from noisy measurements with outliers**

The likely image inputs to the final program will contain both noise and outliers. To an extent, an outlier is essentially an extreme case of noise, so it is harder for the RANSAC to identify which points are outliers and which points are just noise. To do this, a reasonable value for the accepted error norm must be set. This is the likely state of the images that will be input to the program, and a good estimate of the homography is necessary to get the best estimation of the KMatrix afterwards.

Even though finding an optimal value for the number of runs and the Maximum Error allowed is not possible analytically, a statistical view on how those numbers influence the effectiveness of the algorithm is helpful to get insights on how the code performs and thus making it easier to decide on the number of runs and the maximum error allowed from intuition.

For the first attempt of estimation, it will be considered that a “Good Consensus” is the set of four points that are inliers and that fall within 0.5 pixel away from original points. The probability of this happening is calculated by the CDF of the Half-Normal Distribution and is:

The probability of a “Good Consensus” is then given by the probability calculated by the CDF above to the fourth power (four points selected that are “good”) multiplied by the probability that none is an outlier (given by Table 2). Therefore, the probability of a “Good Consensus” will depend on both the level and the percentage of outliers. Since the goal is to find the number of runs needed for each case, it is beneficial to calculate the number of runs that will give a probability of 99% that the algorithm will pick a “Good Consensus” according to the definition above. Figure 3 shows the results of the number of runs for a variety of different values of level and percentage of outliers. This graph is consistent with the runs suggested in the handout since for a level noise of 0.707 pixel (equivalent to 0.5 of the handout), the appropriate number of runs is 71, which is even more conservative than the 50 suggested in the handout. These numbers are not totally precise, since not all points that follow those criteria for a “Good Consensus” will be appropriate (the algorithm could get 4 collinear points for example), but it is accurate enough for all practical purposes.

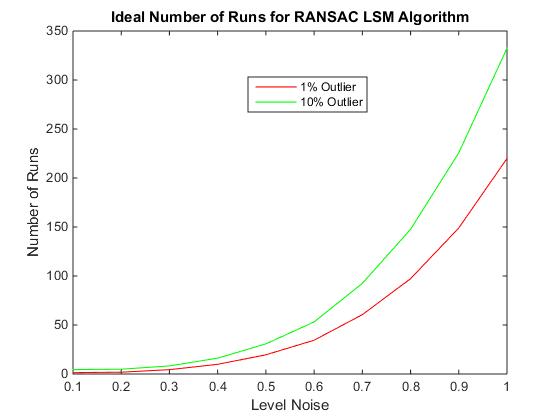
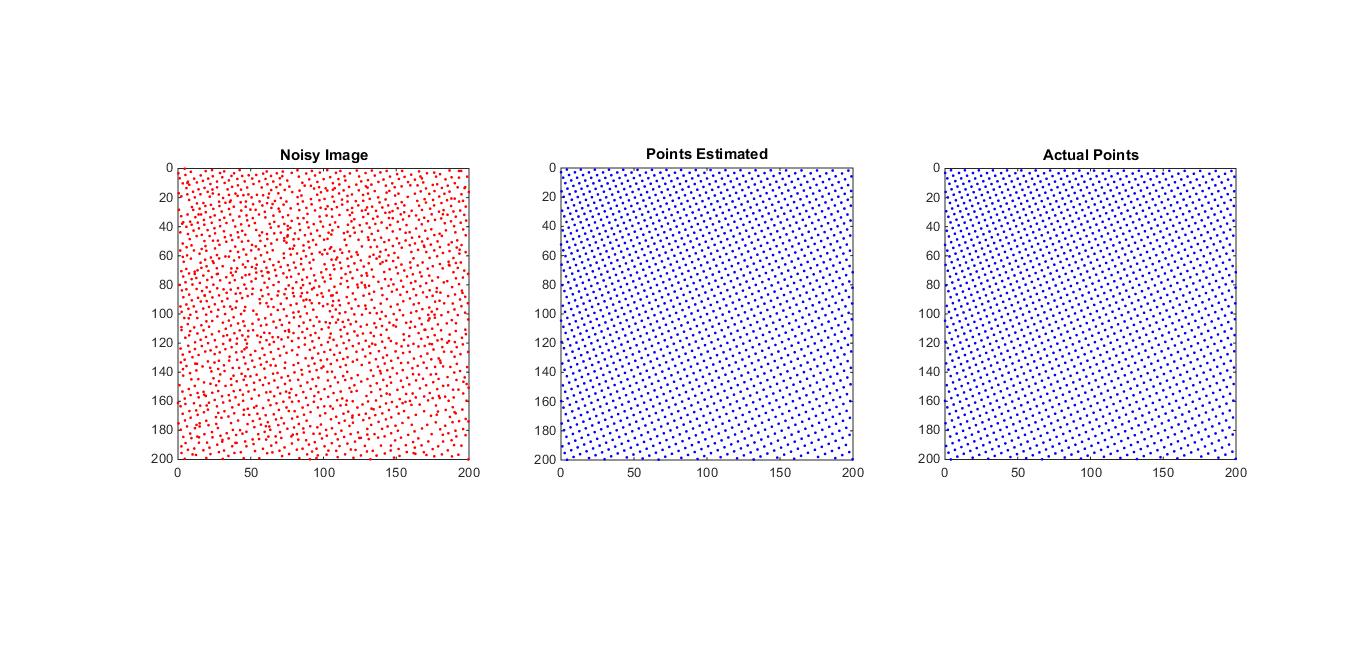


Figure Number of runs that guarantees 99% confidence that the consensus picked by the RANSAC is a good one.

An interesting insight that this graph shows is that the algorithm is exponentially sensitive to the noise level, but just linearly sensitive to the percentage of outliers (for any reasonable amount of outliers). Which means that if there is a choice between a computer vision algorithm that gives high values of noise but low values of outliers or another one that returns high numbers of outliers but low values of noise, the user should choose the latter.

With this idea in mind, the algorithm implemented was tested for level inputs from 0.1 to 1, outliers from 1% to 5%, considering the Maximum Error norm as twice the mean according to Table 1. This approach proved to be robust. One of the results is shown in Figure 4.

Figure Result of RANSAC LSM for noise 0.5, percentage of outliers of 5% and number of runs of 100. The left plot is the input image to the algorithm. The plot in the centre is the output of the algorithm. The plot on the right is the position where the points should be in case the input image had zero outliers and zero noise.



1. **Estimating KMatrix from Homographies**

The first estimate of the KMatrix (or “seed” matrix) can thus be calculated from the 3 or more homographies obtained as described above. Naturally, the noisier the initial image is, the worse the estimation for the KMatrix will be. A way to analyse how robust the code is in regard to noise is to calculate the error of the back projection using the estimated KMatrix. The idea is to first calculate the KMatrix for *6* images, use it to project the position of the points according to our estimate, find the Total amount of error if compared to the ideal image and divide by the total number of images. This way we can judge for what values of noise and outliers the code still performs well.

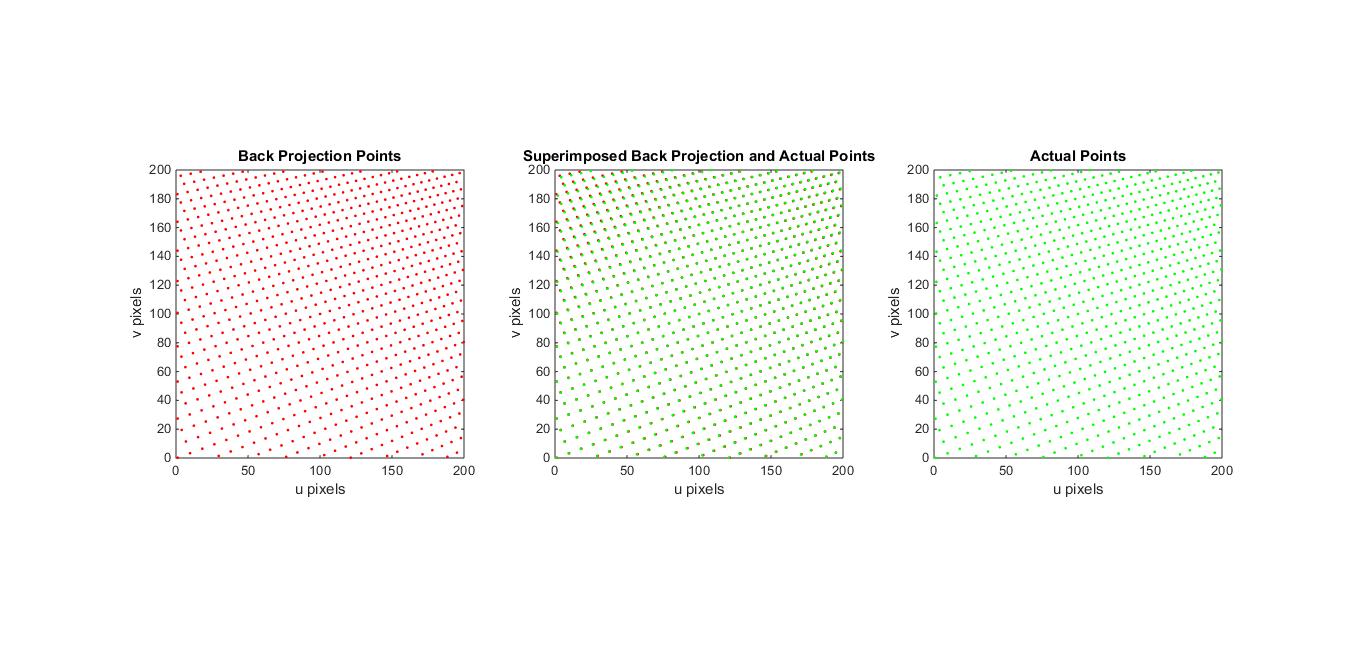
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | Noise Level | | | | | | |
| 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 |
| Outliers | 0% | 0 | 0.1197 | 0.2802 | 0.6808 | 1.3733 | 1.6212 | 2.461 |
| 5% | 0 | 0.1166 | 0.4047 | 0.8812 | 1.3558 | 1.907 | 3.3587 |
| 10% | 0 | 0.304 | 0.5288 | 1.074 | 1.8838 | 2.1715 | 6.2487 |

For this test, the errors were calculated using the unscaled projection of the Image, since it is more intuitive to imagine the impact of one pixel in a 200x200 pixel image. The results are shown in Table 3. It is evident that as expected the code performs well for a limited amount of noise and outliers. Between 0 and 1.5 level noise, the program can track the back-projected images quite reasonably. However, as the noise level starts to get bigger, the error grows exponentially, and the result of the estimated K matrix might not be accurate enough to use the Levenberg-Marquadt optimization method. Also, as expected, for 0 noise, the code just returns the actual K matrix (for a very big range of values of outliers, since the RANSAC algorithm works perfectly for 0 noise).

Table 3 Tests of back projection for a "seed" K matrix using 6 homographies. Those results show the average error in pixels from the actual points to the estimated one using back projection for different vales of noise and outliers.

Figure 5 shows an example of a back projection using the estimated KMatrix described above. It is superimposed with the actual position of the points, so that it gets more intuitive what those values of errors actually mean.

Figure The left plot is the back projected by the "seed" K matrix. The one in the right is the actual position of the points. The one in the middle is the superimposition of both. Plots obtained by using a noise level of 1 and 5% of outliers.



1. **Optimising KMatrix using Levenberg-Marquadt algorithm**

After finding the “seed” K matrix, it is clear that the errors of the back projection using this model will be very small compared to the actual position of the points. From the definition of the LSM, those points were the ones that generated the smallest error when compared to the noisy image, but the rotation matrices generated are a bit off, which consequently makes the KMatrix slightly wrong. To find the “optimal” KMatrix under these conditions, the approach is then to minimize the variance of the error function. This last fine tuning is done using the Levenberg-Marquadt algorithm.

There are some specific and challenging problems about this algorithm. If the data input is using radians for the rotation matrices, the gradient will likely be huge, since it is a measure of the norm of the error of the back projection. By definition, the gradient will be the change of error per radian. One radian is a considerably big angle difference when thinking about pictures. Two homographies taken with a 57.3 degrees difference, all else being equal, will be considerably different from each other. Of course, since this value is big, the *dp* will be very small. However, the value for this then gradient will be disproportional compared to the change in error with respect to a 1mm difference in the transpose vector, for example, which will make a difference when considering the stopping criteria for the algorithm. The algorithm will stop once the norm of the gradient is small, however in my code I scaled the gradient by a vector of values so that the gradient with respect to angles were scaled to with respect to degrees (so they were multiplied by ). This scaling would make a bigger difference for bigger numbers of images.

Also, if the data is unscaled, then the Jacobian and the Hessian will have massive values, since they are defined as the difference in the number of pixels per unit change of a certain parameter. For a 3000x3000 chip size for example, a one radian change for a rotation matrix might cause a difference in pixels in the order of hundreds for one single data point, which means that the norm of the Jacobian for one parameter would be in the order of millions, and therefore the Hessian would be in the order of . To avoid any problems with rounding precision numbers due to very high or very low values, scaled data points are indeed most appropriate and robust approach.

To test the robustness of this code,

1. **Likely Inputs and GUI**

The whole idea of this program is to be used by a “common” user. To make it as user friendly as possible, some thought about the inputs and the GUI must be considered. The inputs to the script reported in this report will be the Chip sizes, at least 3 matrices of position of points and the size of the grid. The computer vision algorithm implemented will have to input those to the script, since it would be inconvenient for the user to have to do it themselves. Then at the end, all that the end user will have to do is to upload the pictures with the checkerboards at different angles, and the whole program will do the rest.

To illustrate the likely input and GUI for the program, everything was simulated on MATLAB. Figure 7 illustrates how the end product will handle data and deal with UX. Using the built in function *detectCheckerboardPoints*. The idea is that the user will upload the picture on the left side of the program. It will then recognize the points and send the points on the right to the script for camera calibration. The output will then be the homographies and the camera K matrix, which can be later used for distances estimation in a larger project.

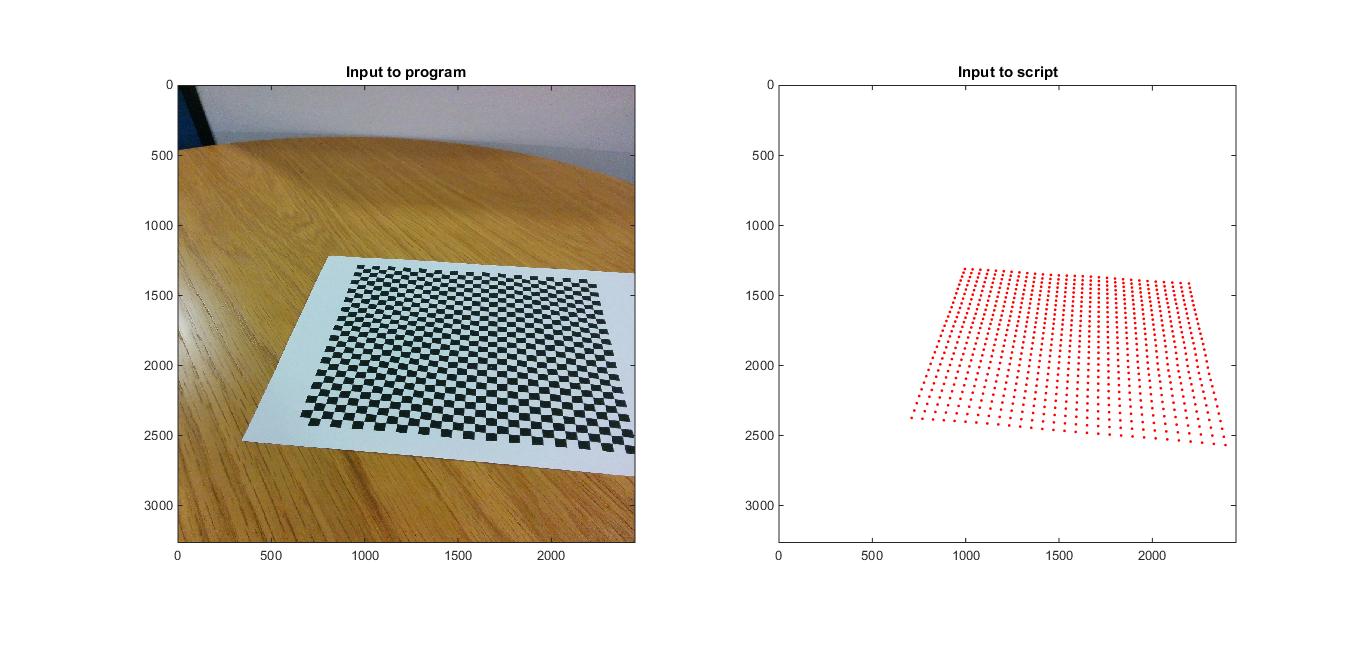


Figure The picture on the left was taken using a Nexus 5 smartphone. The picture on the right is the position of the corners recognized by the built in MATLAB function detectCheckerboardPoints, and the likely input to this mini-project.

1. **Conclusion and Suggestions**

It must be clear to the end user some restrictions: 1) The pictures must cover a wide angle difference, so that the code can perform well; 2) The checkerboard needs to be asymmetric, due to the restriction on the built in function, otherwise the recognized position of the points could be reversed; 3) The noise on the computer vision algorithm has to be low, but judging by Figure 6, this won’t be a problem for any reasonably well built code; 4) The camera has to be fixed focused and distant from the object, so that lens distortion won’t be a factor.

When following all of those restrictions, the program will give an accurate estimation of the parameters of the camera used to take those pictures. Considering the accuracy of the position of the corners detected by the MATLAB built in function, the code is very robust, since it can handle noises values and percentage of outliers bigger than created by the recognition algorithm.

Further development of the code might include a script that calculates the distance between any two points selected by the user, whether in 2D or 3D. This would prove itself a valid product that can be built using this mini project as a foundation.