Calibrating a digital camera for computer vision

Nihaar R. Shah (Candidate No. 641033)

18 January 2016

1 Introduction

Resectioning or calibrating a camera means to estimate the parameters of the lens and the sensor device of the camera. To estimate these parameters one needs 3D world coordinates and their *corresponding* 2D image points using multiple images of a calibration pattern. Assuming there aren't any inaccuracies (due to noise or outliers) in these correspondences and the position of camera relative to object is known then we can simply apply equations to obtain the required parameters.
However, in reality there will always be errors in matching up the corresponding points of image distance between a projected point and a measured one. We can apply RanSac algorithm to reduce outliers by choosing the best fitting points only. Assuming the noise in the correspondences is mainly Gaussian, we can minimize the variance between image locations and predicted locations. This optimization is known as bundle–adjustment. Our project implements these two procedures to find a near accurate K-Matrix of required parameters.

1.1 Aim

The aim is to estimate and optimize the intrinsic parameters of a simplified ideal pinhole based model of a camera. My report attempts to *test* and *explain* the algorithms used and suggests further actions that can be taken to achieve a commercial solution.

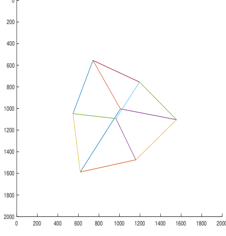
1.2 Motivation

Knowledge of intrinsic parameters is very useful in finding real world distances of planar objects (from a single camera) if we know the position where the camera is mounted. Inversely, if we know distances in an image then we can estimate the camera location. Using stereo cameras we can estimate depth as well, which is useful in **visual odometry** which has many applications in robotics for example in localization for **self-driving cars**.

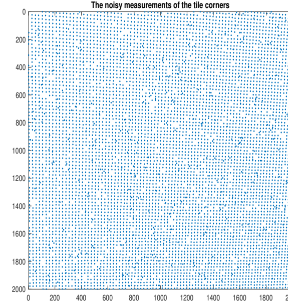
2 Outline

2.1 Simulation

In practice we would use an actual calibration object like a checkerboard pattern with known distances between edges. In this project, we simulate the calibration pattern. We use a known K-matrix with pre-defined parameters to make correspondences. In reality the K matrix is unknown. Through algorithms edge detection of the calibration object is possible. We then feed the known distances between these edges to the program to build the correspondences. In addition we input the 6 degrees of freedom needed to define the location of object in world coordinates (e.g. using lidar sensors). Instead, in this project we have functions to position the camera and position the object in world coordinates. A good test that these functions are working as expected is to view a simulated cube as seen by the camera as shown in figure (1). Note the 4 MegaPixel(MP) camera hence 2000 x 2000 pixel dimensions of the image. The cube was built by inputting each edge as a combination of cube vertices.



(a) Simulated cube in camera's field of vision.



(b) Noisy grid image appears on camera sensor.

2.2 Initial Estimate

2.2.1 Estimating Homography

We build correspondences between object coordinates (xy) and image points (uv) using position of object relative to camera. **K** parameters and the positional parameters are packed in the Homography elements $h_{i,j}$ and our task is to estimate **H**. Using j = 1 to 4 known point correspondences we can stack the 4 matrices to build a determinate system of equations having 8 unknowns and 8 equations. Solving this system will give us an estimate of Homography. But how do we know that the 4 points we have chosen to estimate it are all inliers? Thus we perform RanSac to select those pixel points which fit well with the **H**. Now we have an over-determined system with as many equations as the inliers. This system of equations is $\mathbf{P} = \Phi\ \mathbf{H}$. We expect the correspondences to be noisy, hence no exact solution will exist. We therefore use **least squares minimization**.

$$min|\mathbf{\Phi H} - \mathbf{P}|^2 \implies \frac{\partial|\mathbf{\Phi H} - \mathbf{P}|^2}{\partial(\mathbf{parameters})} = 0 \implies \mathbf{H} = [\mathbf{\Phi}^T\mathbf{\Phi}]^{-1}\mathbf{\Phi P} \tag{1}$$

Φ (i.e. Regressor made of uv and xy coordinates) and **P** (i.e. uv coordinates) are *known*. We use **singular value decomposition** (svd) to compute the inverse in Equation 5 because that is computationally cheaper than using matlab inverse (See Section 4.2). This is called pseudo-inverse. Now we have estimated the Homography.

2.2.2 Estimating K-Matrix from Homography

Finally we want to estimate the K matrix from this Homography using 3 images of the grid taken from different angles. We pre-multiply both sides by K inverse. $\mathbf{H_j} = \lambda_j \mathbf{K_j}[\mathbf{r_1 r_2 t}]_j \implies \mathbf{K}^{-1}\mathbf{H_j} = \lambda_j[\mathbf{r_1 r_2 t}]_j$ We use the orthogonal property of the rotation matrix by pre-multiplying the Homography with the rotation matrix transpose (or inverse) $\mathbf{H_j}^T(\mathbf{K_j^{-1}})^T\mathbf{K_j}^{-1}\mathbf{H_j} = \lambda^2\mathbf{I}$. This equation holds true for each image j and can be used to make a homogeneous equation of form Ax=0 to solve for $\phi = (\mathbf{K^{-1}})^\mathbf{T}\mathbf{K}^{-1}$ again using $svd(\mathbf{A}) = \mathbf{UDV}^T$ and right singular vector of matrix ϕ associated with the *smallest* singular value. This is because for a homogeneous system any vector **x** in the null space of **A** is a solution hence any column of **V** whose corresponding singular value is zero is a solution. If we want a particular solution then we might want to pick the solution **x** with the smallest length $|\mathbf{x}|^2$. We then obtain **K** from Φ using *cholesky factorization* because we note that $(\mathbf{K}^{-1})^T$ is an upper triangular matrix and K^{-1} a lower triangular matrix.

2.3 Optimization

So far we have reduced outlier effects using RanSac. Gaussian noise still prevails. We can tackle this using optimization of the cost-function ($\mathbf{E}$) defined as the least squares error between estimated and true values of uv. $\mathbf{E} = \frac{1}{2}\Sigma(Estimated[u'v'] - True[uv])^2$ Note that least squares form of cost function is an efficient way to eliminate noise but not outliers as it heavily penalizes Outliers. Thus, we have chosen the two step approach - RanSac followed by least squares optimization. This is a convex optimization problem (cost function is quadratic) hence we expect the global and local minimum to be the same. We decompose the Homography into the K-Matrix and positional parameters and then use this to predict the positions of points in the image using these transformation matrices and the current best camera matrix. The square of this minus the actual positions serves as our convex cost function.

We will use the **Levenberg-Marquardt algorithm** to adjust parameters and decide when the minimum is reached. Steepest descent is a method to reach minimum by varying parameters δ $\mathbf{p}$ along the steepest *downhill* direction on the surface i.e. pointing along the *negative* gradient vector. $\delta\mathbf{p} = -(\frac{\partial\mathbf{E}}{\partial\mathbf{p}})^T = \mathbf{g_n}$. Newton steps method is based on taking the derivative of Taylor's expansion in N-dimensions as shown in Equation 2

$$\nabla(f(\mathbf{x} + \delta\mathbf{x})) = 0 \implies \mathbf{g_n} + \mathbf{H_n}\delta p = 0 \implies \delta\mathbf{p} = -\mathbf{H_n}^{-1}\mathbf{g_n} \quad (2)$$

This gives the iterative update $\mathbf{x_{n+1}} = \mathbf{x_n} - \mathbf{H_n}^{-1}\mathbf{g}_n$ where $\mathbf{H_n}$ is the Hessian matrix $\frac{\partial^2\mathbf{E}}{\partial\mathbf{p}^2}$. It is better to perform line search which ensures global convergence hence we introduce the parameter μ in Equation 3

$$\mathbf{x_{n+1}} = \mathbf{x_n} - \mu\mathbf{H_n}^{-1}\mathbf{g}_n \text{ And combining Equations 2 and 3 } (\mathbf{H_n} + \mu\mathbf{I})\delta\mathbf{p} = -\mathbf{g}_n \quad (3)$$

By solving for $\delta\mathbf{p}$ we we know the vector along which to change our parameters for the next iteration. We re-compute the cost at the new parameters and assess whether to reject the parameters or accept them as well as whether to increase or decrease μ i.e. we can control between Newton steps and gradient descent.

3 Overview of the Structure

3.1 Data Structure

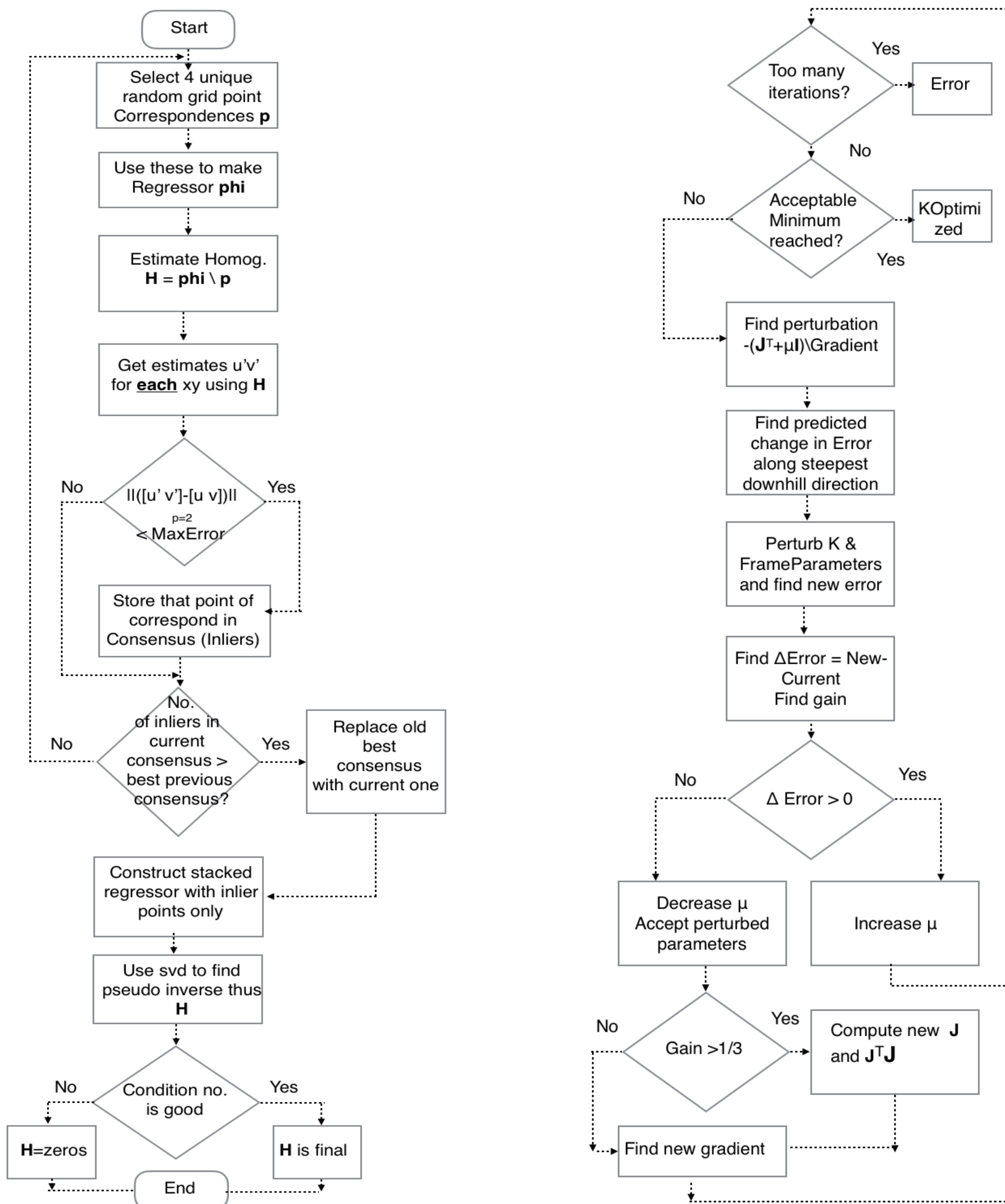
When designing code you have to consider how to encapsulate the data so that flows are clear and match the control structure. I stuck to follow a structure of many interacting functions instead of creating large objects with many tasks. For e.g. in the Jacobian function I used Rodrigues function to make Rotation matrix from angle-axis instead of doing that calculation inside the Jacobian function (shown later). Another e.g. in Jacobian function is I made a function for the task of perturbation because: less scope of error and **re-usability**. This modular structure also is compiler efficient and useful to scale up. Someone using this code for calibration may not input valid parameters so I validate parameters before running a function but there's a risk of passing parameters in the wrong order. Thus structures are good for passing long lists of parameters as the named

fields are safer. Cell structure was used for storing data as the objects stored had unknown and variable sizes. MATLAB's sparse matrix data structure is a good choice for storing the entire Jacobian as there are several 0 terms. However, as I later show this was not the preferred way of finding Jacobian.

```
S = sparse(A) %  converting the matrix to sparse storage saves memory.
```

3.2 Control Path

FlowCharts for Algorithms used



(a) Estimation task involving RanSac

(b) Optimization:the L-M algorithm

Figure 2: Flow charts for the two algorithms used to reduce Outliers and Noise respectively.

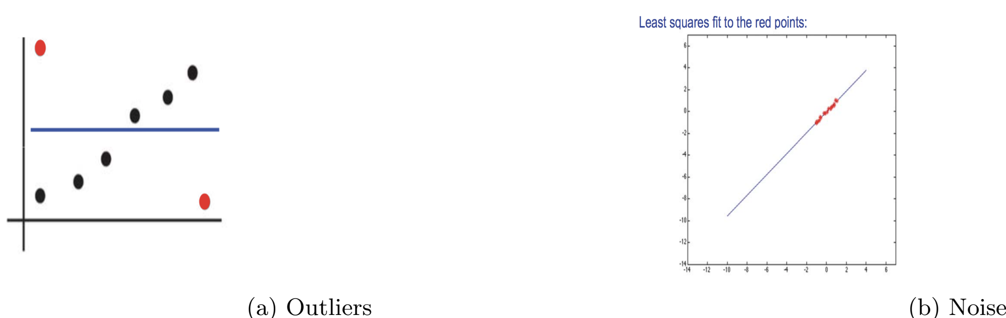
RanSac flow

RanSac algorithm eliminates outlier points - in this case pixels displaced from their true positions in the calibration image. While estimating homography from the edges of the checkerboard we take 4 random point correspondences to build our regressor Φ. To ensure these points are a true representation of the objects (inliers) we perform RanSac. We use RanSac to choose only *that* homography estimate which best agrees with maximum number of points on the checkerboard. The flowchart in Figure 3a shows this algorithm. Note that UV cordintes are the known points or true points. We multiply the estimated homography with the xy object coordinates to obtain estimated U'V'. RanSac compares the difference between the U'V' estimated via homography and the true UV with a threshold MaxError and populates a vector BestConsensus that satisfies this threshold as acceptable inliers. We construct our Regressor of the over-constrained system and use svd to find the least squares estimate of our final homography if the condition number is "good" as accurate inverse is possible. As good coding practice I catch error using 'try' 'catch' to ensure inverse exists. With a good homography estimate it is straightforward to compute K using svd as mentioned in Section 2.2.2.

Optimization flow

The algorithm controls the search to move from steepest descent to newton steps depending on how well the last search performed. The feedback mechanism is that if error has increased after changing the parameters along a certain direction then the weighting μ is increased reducing the step-size and causing the algorithm to just search downhill. If the error has reduced then we are *closer* to the minimum so we search in the neighborhood itself using Newton steps.

3.3 Why our approach?



(a) Outliers (b) Noise

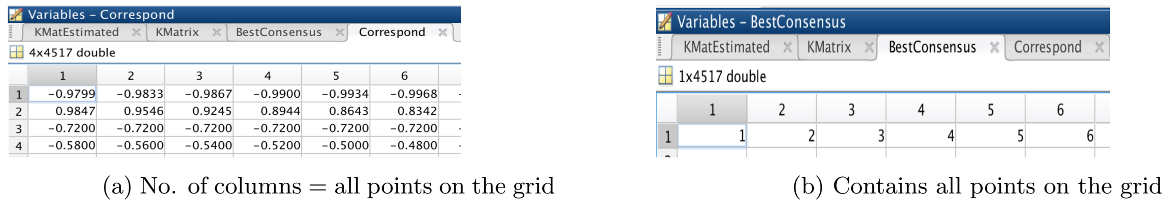
Figure 3: Graphical representation of impact of Least Squares minimization on Noise and Outliers.

Suppose you fit a straight line to data containing outliers, the usual method of least squares estimation is flawed. It will penalize outliers excessively and lead to poor fit. Thus we must first get rid of these outlier and **then** apply Least Squares optimization. This is precisely our method. The way I add random noise is by passing σ and μ as flags in the BuildNoisyCorrespondences which can be changed from RunEstimateHomography script.

```
% Adding Gaussian noise with a changeable variance to the image
CalibrationImage = CalibrationImage + random('norm',mu,std,m,n);
```

4 Detailed Considerations

4.1 How to measure accuracy


Variables - Correspond
KMatEstimated | KMatrix | BestConsensus | Correspond
4x4517 double

	1	2	3	4	5	6
1	-0.9799	-0.9833	-0.9867	-0.9900	-0.9934	-0.9968
2	0.9847	0.9546	0.9245	0.8944	0.8643	0.8342
3	-0.7200	-0.7200	-0.7200	-0.7200	-0.7200	-0.7200
4	-0.5800	-0.5600	-0.5400	-0.5200	-0.5000	-0.4800

Variables - BestConsensus
KMatEstimated | KMatrix | BestConsensus | Correspond
1x4517 double

	1	2	3	4	5	6
1	1	2	3	4	5	6

(a) No. of columns = all points on the grid (b) Contains all points on the grid

Figure 4: The workspace data structure dimensions verify all points are inliers.

Noise and Outlier Test

The simplest was to set the noise σ and $\mu = 0$ and pOutliers=0 in the Correspondence matrix.

Hypothesis: There are now *only* inliers i.e. all correspondences are accurate so estimating homography from any 4 points should give error of ≈ 0, since u'=u and v'=v. *No* points are excluded from Consensus vector.

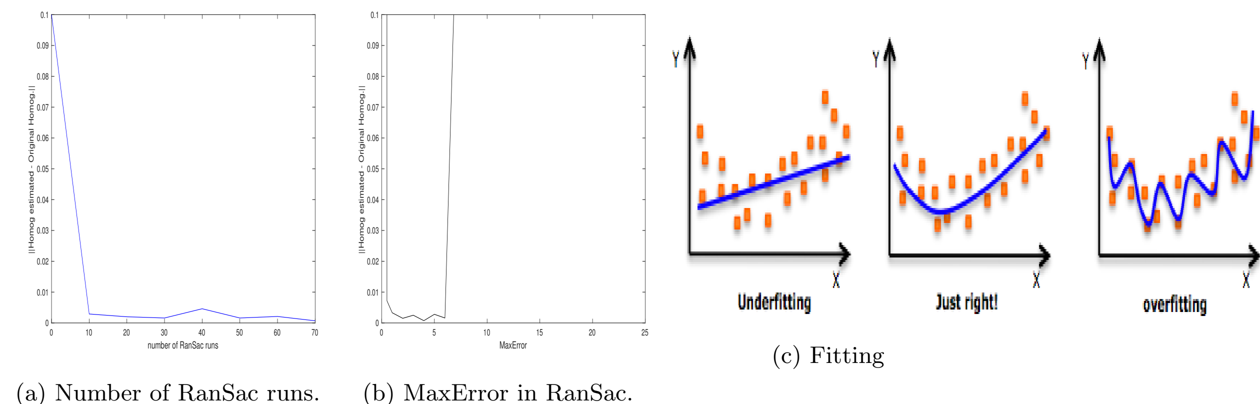
Result: This was verified as seen in Figures 4 (a) and (b). Also K estimated is the same as the K-Matrix.

RanSac runs Test

The second experiment was to vary number of RanSac runs and examine its effects on the accuracy of $K_{estimated}$. The accuracy metric used was the L_2 norm because $p > 2$ norm penalizes greater deviations too much while p=0 or p=inf norm maybe less representative of all elements. L_2 norm is a least squares so it makes sense to judge the performance of least squares minimization although L_1 norm is also a valid choice. A better metric may be % error but 4 of 5 elements of K are 0 hence it is hard to find % change from 0. The noise and pOutlier were control parameters.

Hypothesis: More RanSac runs means more number of homographies to choose from to use as best fit for correspondence points. Better outlier rejection, hence a more accurate K estimate.

Result: The general trend corroborates this expectation as seen in Figure 5a.



(a) Number of RanSac runs. (b) MaxError in RanSac. (c) Fitting

Figure 5: Varying Ransac runs and MaxError to find effect on Accuracy.

MaxError Test

Hypothesis: Decreasing the MaxError means we are decreasing the threshold of accuracy (more strict) for an

estimated u'v' to be part of consensus set. So we will have more strictly selected inliers discarding even slight deviations hence more accuracy.

Result: For both very high or very low MaxError values the accuracy drops. For very high MaxError there are too many points including many outliers so lowered accuracy in Fig. 5b . Why lower accuracy for low MaxError then? Because of **over-fitting** i.e. having too many parameters relative to the number of observations so estimate can overreact to minor fluctuations in the training data. If the max error is too low, there will be very few points (very selectively chosen) in the consensus and in the regressor for the robust inverse (svd) calculation. I wrote extra script to output how many points are there in Bestconsensus set for a given Max Error value and found that for MaxError of 10^{-4} and a noise σ of 0.1 we just have 10 elements and this reduces if MaxError is further reduced or if noise σ is increased. Homography has 9 parameters.

Test for Jacobian calculation

For optimization I calculate the jacobian matrix by writing *forward-difference* function. The perturbation was set to 0.001 % of the parameter and we used a for-loop to manually differentiate (perturb, subtract and divide) with respect to each parameter and construct our blocks as shown in the code below:

```
 % We are estimating the derivative using f'(x)=[f(x+dx)-f(x)]/dx w.r.t each of 11 parameters
% Perturb each K parameter by a small percent of itself turn by turn and store each perturbed matrix
k1 = KMatrix;
k1(1,1) = KMatrix(1,1)*perturbation; % ... and so on until k5(2,3)=KMatrix(2,3)*perturbation;

% Find new estimated uv from each perturbed kmatrix. This is f(x+dx) in f'(x)=[f(x+dx)-f(x)]/dx
fk1 = k1 * P *[XY; 1]; % ... and so on until fk5 = k5 * P *[XY; 1];

% This is f'(x). We divide by fk(3) to scale it back to original.
dk1 = (fk1(1:2)/fk1(3)-UVest)/(KMatrix(1,1)*(perturbation-1)); % ... and so on until dk5;

% Finally constructing our K parameters block consisting of 5 columns and 2*no. of points rows
NKMATJACOB(2*a-1:2*a,:) = [ dk1 dk2 dk3 dk4 dk5 ];
```

To **test the accuracy** of this function, I wrote another function using **symbolic MATLAB toolbox** which textitanalytically finds the *entire* Jacobian and then I select the required blocks from that, as shown (some lines omitted for conciseness):

```
% Parameters come from KMatrix, Rotation axis and Translation vector
K = sym('k',[3,3]);
RotAxis = sym('rot',[1,3]); % etc. for other parameters

% fin is calculated symbolically in terms of the 6 extrinsic parameters in P and 5 intrinisic in K
for i = 1:n
   XY = Correspond(3:4,BestConsensus(i));
   f = K * P * [XY' 1]';
   f = f/f(3);
   fin = [fin;f(1:2,1)];
end
Var = [(1,1),K(1,2),K(1,3),K(2,2),K(2,3),RotAxis(1,1),RotAxis(1,2),RotAxis(1,3),t(1,1),t(2,1),t(3,1)];
% symbolically forming the two blocks using the jacobian library function
JKMat = jacobian(fin,Var(1,1:5)); JFram = jacobian(fin,Var(1,6:11));

% Substituting in the parameters one by one like shown for K parameter below
J1 = subs(JKMat,K,KMatrixValues);
```

MATLAB's 'jacobian' library function uses **symmetric central difference** and other algorithms. It's an analytic approach and more accurate than simply forward difference. However, it's computationally expensive as the zero-terms of the Jacobian are computed which can otherwise be avoided by only calculating the required

blocks (as done in the former above). Both results were nonetheless similar, thus we prefer the first method.

4.2 Is this simulation a good model of the real world?

There are approximations and assumptions in any model. Here are a few:

A) I.I.D. Noise Principal sources of Gaussian noise in digital images arise during acquisition e.g. sensor noise caused by poor illumination, high temperature, transmission e.g. electronic circuit noise. A typical model of image noise is Gaussian, additive, independent at each pixel. Obviously, there are other noise sources that are *not Gaussian* e.g. Salt-and-pepper noise (caused by analog-to-digital converter errors), Shot noise (statistical quantum fluctuations), Quantization noise (quantizing the pixels of a sensed image to a number of discrete levels), Anisotropic noise (noise sources show up with a significant orientation) etc. Thus Gaussian noise is an *approximation* not totally representative of truth. Based on this assumption we have chosen cost function as a least squares minimization.

B) Outliers An outlier in an image is often a region that has been occluded, an object that suddenly appears in one of the images, or a region that undergoes an unexpected motion (due to slight camera motion). Outlier modelling is a field on its own. We have used a *simplistic* way of adding outliers at random pixels.

C) Lens distortion We have used an ideal pinhole simplification *excluding* radial and tangential distortion. Radial Distrortion occurs when light rays bend more near the edges of a lens than they do at its optical center. The smaller the lens, the greater the distortion. Tangential distortion occurs when the lens and the image plane are not parallel.

D) Algebraic approximations: For mathematical and computational simplicity we have used certain approximations. For instance to calculate matrix inverse we use svd *'pseudo inverse'*. For determining stopping criterion, instead of comparing Gradient to 0 we set the criterion to a *really low number* such as 1e-5. For computing the *Hessian* as second partial derivatives we approximate it as $J^T J$ thus using the jacobian already calculated.

5 Measures of Code Performance

5.1 Noise

Fig. 6 shows the trend that more the noise variance i.e. the image pixels of calibration image are more apart from their true values then the worse the error between the norm between true Homography matrix and the estimated. Although there is one outlier in the above trend, that is reasonable since we are randomly choosing points each time we run the estimate. So even with the same value of standard deviation and every other parameter, there will be variations in the error values which is why we can explain that as long as the trend is towards worse error

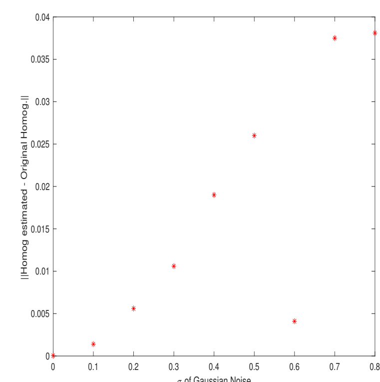


Figure 6: Effect of varying noise variance on accuracy of estimation.

with increased variance we can ignore this particular outlier.

5.2 Speed

Profile Summary

Generated 30-Dec-2016 01:05:31 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
RunOptimizeKMatrix	1	108.647 s	0.046 s	
OptimiseKMatrix	1	108.237 s	0.039 s	
mupadmex (MEX-file)	30268	104.794 s	104.191 s	
sym.sym>sym.double	30	56.142 s	0.034 s	
JacobianSym	6	42.859 s	0.264 s	

(a) Using library jacobian function.

Profile Summary

Generated 30-Dec-2016 00:49:14 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
RunOptimizeKMatrix	1	0.497 s	0.051 s	
OptimiseKMatrix	1	0.173 s	0.030 s	
BuildNoisyCorrespondences	6	0.125 s	0.018 s	
SingleImageJacobianMax	6	0.118 s	0.068 s	

(b) Using forward difference.

Figure 7: Timing performance for the two methods to compute Jacobian.

The Figure 7 shows how the analytic jacobian compares with the forward difference method. The self-time (i.e. the time taken in that particular function excluding the 'child' functions) for finding the Jacobian compares so:

```
% improvement = 0.068/(0.264-0.068) = 35 %
```

While the total time taken i.e. includes the cumulative time over all the iterations and Ransac runs is incomparable 0.497s vs 108.64s. For calibration speed is less important than robustness but the two methods are almost equally accurate. Thus, we clearly prefer the approximate forward difference method.

6 Conclusions

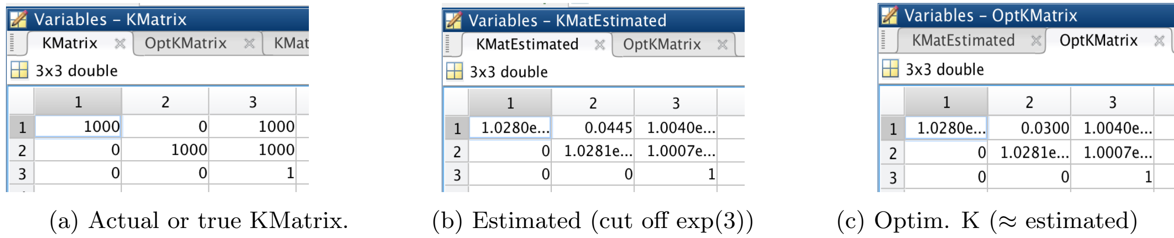
6.1 How would I input real data?

This project simulated the calibration image. In reality we would use an image of an actual calibration image and after calibration only deal with real images.The image would be treated as a matrix of intensities. Using edge detection we can find boundaries of objects within image **by detecting discontinuities** in brightness. Common algorithms include Sobel, Canny, Prewitt, Roberts, and fuzzy logic methods. The MATLAB command:

```
BW = edge(I,<type of algorithm>,threshold,direction)
```

returns a binary image BW containing 1s where the function finds edges in the input image I and 0s elsewhere. It returns edges that are stronger than threshold. The input image I is an intensity or a binary image. Then we can enter known distances between them in the calibration image. After it is calibrated, we can use K-Matrix and Extrinisic parameters (from the position where we mount camera with respect to object) to find the real-world distances simply from an image of the planar objects.

6.2 How well did you do?



(a) Actual or true KMatrix. (b) Estimated (cut off exp(3)) (c) Optim. K (≈ estimated)

Figure 8: Comparing the final K-optimized and K-Estimated. Noise σ =0.2, Stopping criterion value =1e-8

Optimization performance of both methods is compared below by varying the Stopping Criterion (S.C.).

Method	**S.C.=1**	**S.C. = 1e-3**	**S.C.= 1e-8**	**S.C. = 1e-9**
Forward Difference	42.0531	28.137	3.0238	too many iterations
Analytical	41.432	27.438	2.983	too many iterations

Accuracy metric used was L_1 norm as discussed in Section 4.1 although other metrics are also valid. Results from both methods were similar so I concluded that forward difference approximation might suffice in this case. It nevertheless served as a good test to verify Jacobian. To compare the two methods I kept noise σ as an important control variable because the it has a direct impact on the optimization's performance.

The stopping criterion is chosen such that we can be reasonably sure that the minimum of the function has been reached i.e. the norm(gradient) is ≈ 0. Clearly keeping a high stopping threshold such as SC = 1 (see table) leads to lower accuracy. At the same time a very low SC = 10^{-9} meant the code never exited the optimization loop even after 100 iterations. As seen in Fig 8 the Estimated and Optimized K is nearly the same and this is expected as both are obtained by minimizing a least squares cost function and this is a convex problem.

6.3 Commercial product

Finally, how close are we to make this into a commercial software? This has partly been answered in Section 4.2 where we have discussed the assumptions, approximations and simplifications of this model. A commercial standard product may be able to improve on some of those for instance take the lens distortion into account or apply statistical noise and outlier models depending on the kind of image taken. Adding a GUI that can let the user input an image and automatically output the K-Matrix or the real-world distances from an image, are required for good user experience. If more computational power is needed then the commercial solution should be able to harness GP-GPUs remotely if needed. For instance this could be a web-application which utilizes immense computational hardware in a data center. Lastly, there is a possibility to apply statistical testing to predict with a certain confidence level how accurate is our estimate and the Type of Error (I or II).

7 References

1) Prof. Zisserman's and Prof. Daniel's slides on optimization
2) Noise: Jun Ohta (2008). Smart CMOS Image Sensors and Applications. CRC Press. ISBN 0-8493-3681-3.
3) Distortion:https://uk.mathworks.com/help/vision/ug/camera-calibration.html