Joel Chaverie
1002924393

CSC 2503: Assignment 3

A.1. To begin, I will define λ_1 = MintLeft, λ_2 = MintRight, σ_1 = MattLeft and σ_2 = MattRight. As σ_i denote the extrinsic parameters of the two cameras, they can equivalently be written as:

$\sigma_i = [R_i, T_i]$, where R_i is a 3×3 rotation matrix and T_i is a 3×1 translation vector of camera i.

Now the general projection equations for two cameras is defined to be:

$$\alpha_1 x_1 = \lambda_1 \sigma_1 X \quad (1)$$
$$\alpha_2 x_2 = \lambda_2 \sigma_2 X \quad (2)$$

where X is a 4×1 homogeneous representation of a 3D point, α_i is a scalar, and x_i is the observed position of X in camera i in 3D homogeneous coordinates.

To simplify the computation of the fundamental matrix, which captures the geometric relation between the two cameras, start by normalizing (1) and (2) by multiplying by λ_i^{-1} on both sides. Then one has:

$$\lambda_1^{-1} \alpha_1 x_1 = \sigma_1 X$$
$$\lambda_2^{-1} \alpha_1 x_2 = \sigma_2 X$$

Next, in order to represent the rotation of one camera with respect to the other, we need to multiply σ_1 and σ_2 by a Euclidean transformation H, where H is defined to be:

$$\left[\begin{array}{c|c} R_1^T & -R_1^T T_1 \\ \hline 000 & 1 \end{array}\right] \quad (\Delta, \text{ see why this is used at the end})$$

and X by H^{-1}. This allows us to treat the

world coordinates as being centered on the first camera, so that

$$O_1 H = \begin{bmatrix} R_1 & T_1 \end{bmatrix} \begin{bmatrix} R_1^T & -R_1^T T_1 \\ 000 & 1 \end{bmatrix} = \begin{bmatrix} I & 0 \end{bmatrix}$$

$$O_2 H = \begin{bmatrix} R_2 & T_2 \end{bmatrix} \begin{bmatrix} R_1^T & -R_1^T T_1 \\ 000 & 1 \end{bmatrix} = \begin{bmatrix} \underbrace{R_2 R_1^T}_{\Omega} & \underbrace{-R_2 R_1^T T_1 + T_2}_{t} \end{bmatrix}$$

(Here I Reparametrize for Ω and t)

Yielding: $\lambda_1^{-1} \alpha_1 x_1 = [I, 0] H^{-1} X$

$\lambda_2^{-1} \alpha_1 x_2 = [\Omega \; t] H^{-1} X$

As the coordinate system for X has been simply shifted by H^{-1}, we can ignore the H^{-1} term, or reparametrize and treat $H^{-1}X$ as X'. Then, we can consider expanding out the first equation,

$$\lambda_1^{-1} \alpha_1 \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} u' \\ v' \\ w' \\ 1 \end{bmatrix} = \begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} \quad (*)$$

where $X' = \begin{bmatrix} u' \\ v' \\ w' \\ 1 \end{bmatrix}$

$(*)$ simplifies to: $\lambda_1^{-1} \alpha_1 x_1 = X' \quad (3)$

Similarly, for the second camera we get:

$$\lambda_2^{-1} \alpha_2 \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & t_1 \\ w_{21} & w_{22} & w_{23} & t_2 \\ w_{31} & w_{32} & w_{33} & t_3 \end{bmatrix} \begin{bmatrix} u' \\ v' \\ w' \\ 1 \end{bmatrix} = \begin{bmatrix} u' \\ v' \\ w' \end{bmatrix}.$$

or: $\lambda_2^{-1} \alpha_2 x_2 = \Omega X' + t \quad (4)$

Substituting (3) into (4), we get:

$$\lambda_2^{-1} \alpha_2 x_2 = \alpha_1 R \lambda_1^{-1} x_1 + t \quad (5)$$

This simply means that there is a constraint on possible positions for corresponding points x_1, x_2 in the images, which is parametrized by the rotation and translation $\{R, t\}$ of camera 2 relative to camera 1.

Next, take the cross product of all the terms in (5) with the translation vector t. The last term disappears on the RHS as $\forall$ vectors $\vec{a}$, $\vec{a} \times \vec{a} = \vec{0}$. (5) then becomes:

$$\alpha_2 t \times \lambda_2^{-1} x_2 = \alpha_1 t \times R \lambda_1^{-1} x_1 \quad (\times \text{ denotes cross product})$$

Next take the inner product of both sides with $\lambda_2^{-1} x_2$. The LHS disappears as $t \times \lambda_2^{-1} x_2$ must be perpendicular to x_2, so we yield:

$$(\lambda_2^{-1} x_2)^T \alpha_2 t \times \lambda_2^{-1} x_2 = 0 = (\lambda_2^{-1} x_2)^T \alpha_1 t \times \lambda_1^{-1} x_1$$

$$\Leftrightarrow x_2^T \lambda_2^{-T} t \times \lambda_1^{-1} x_1 = 0 \quad (\text{as } \alpha_1 \text{ is just a constant scaling factor it can be divided out}).$$

Then, note that $t\times$ can be expressed as a matrix:

$$t\times = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}$$

So we ultimately have $x_2^T \lambda_2^{-T} E \lambda_1^{-1} x_1$, where E is the essential matrix, and is defined as $t\times R$.

Finally, we can reparametrize this to yield that

$0 = x_2^T F x_1$, where $F = \lambda_2^{-T} E \lambda_1^{-1}$, is the fundamental matrix.

Hence in order to compute F_0, I compute:

1. R, t based on relating camera 2's rotation and translation relative to camera 1, through MextRight and MextLeft multiplied by the Euclidean transformation H.
2. E, where E is defined to be $t \times R$
3. F_0 by computing $\lambda_2^{-T} E \lambda_1^{-1}$, where $\lambda_1 =$ MintLeft and $\lambda_2 =$ MintRight.

□

This yields the ground truth F_0.

(Δ) Lemma: To transform an extrinsic camera matrix $[R\ t]$ to $[I\ 0]$, multiply it by $H = \left[\begin{array}{c|c} R^T & -R^T t \\ \hline 000 & 1 \end{array}\right]$

Pf: We want to compute $[R\ t]^{-1}$, but this is not possible as this matrix is 3×4. To alleviate this, pad it with a row $[0\ 0\ 0\ 1]$ at the bottom. We then have:

$$\left[\begin{array}{c|c} R & t \\ \hline 000 & 1 \end{array}\right] = \left[\begin{array}{c|c} I & t \\ \hline 000 & 1 \end{array}\right] \left[\begin{array}{c|c} R & 0 \\ \hline 000 & 1 \end{array}\right]$$

So

$$\left[\begin{array}{c|c} R & t \\ \hline 000 & 1 \end{array}\right]^{-1} = \left[\begin{array}{c|c} R & 0 \\ \hline 000 & 1 \end{array}\right]^{-1} \left[\begin{array}{c|c} I & t \\ \hline 000 & 1 \end{array}\right]^{-1}$$

$$= \left[\begin{array}{c|c} R^T & 0 \\ \hline 000 & 1 \end{array}\right] \left[\begin{array}{c|c} I & -t \\ \hline 000 & 1 \end{array}\right]$$ (as $R^{-1} = R^T$ by the properties of rotation)

$$= \left[\begin{array}{c|c} R^T & -R^T t \\ \hline 000 & 1 \end{array}\right]$$ □

Joel Cheverie
1002924393

CSC 2503: Assignment 3

A.1.

Following my derivation of F_0 based on the extrinsic and intrinsic camera matrices, I compute it in the dinoTestF script and find it to be:

F_0 =

```
       0   -0.0032         0
  -0.0032         0    0.9487
       0   -0.9487         0
```

A.2.

I follow the algorithm specified in the handout, specifically I compute the epipolar lines for a grid of points in the right image. To generate a grid of points, I am considering 441 points spread out over the x interval [-150, 150] and the y interval [-100, 100]. I am considering (x,y) point combinations of 21 incremented points along the x axis incremented in jumps of 15 and 21 incremented points along the y axis incremented in jumps of 10 (i.e. (-150, -100, 1), (-135, -100, 1), (-150, -90, 1), (-135, -90, 1), etc...) Multiplying these points with F_0 and F, I generate epipolar lines in the left image. I then crop the epipolar lines generated using F_0 to lie inside the box of the image.

If the cropped lines have endpoints that evaluate to be NaN (so not in the image), I ignore them, otherwise I check both endpoints of these cropped epipolar lines. In order to compute the maximum perpendicular error from the cropped epipolar lines to the uncropped epipolar lines, I compute the distance from the two endpoints of these cropped lines to a point (potentially outside of image region) on the uncropped epipolar lines generated by F. It is impossible for the perpendicular distance to be larger anywhere in the cropped box than at these two endpoints, so I do not consider any other points on the epipolar lines generated by F_0. I then take the maximum of the two endpoint distances as the perpendicular error between the epipolar lines generated by F_0 and F for a given grid point. Finally, I take the maximum of all these maximum perpendicular errors to be the overall error measure between F and F_0. I compute this maximum perpendicular distance in the dinoTestF script, where using the default generated F matrix I find this error to be: 4.1925e-13

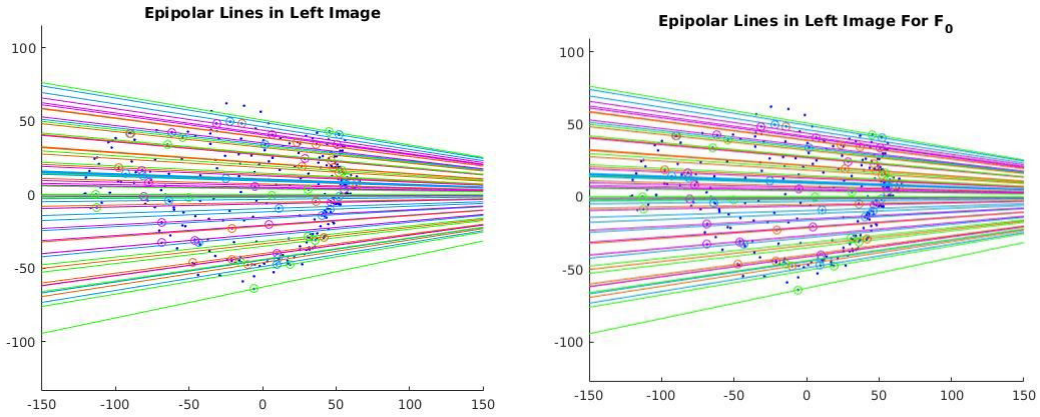


Figure A.1. A visual comparison between the epipolar lines generated by the matrix F and the matrix F_0 on 64 of the provided imPts. The error here is incredibly small, hence the two images appear essentially identical.

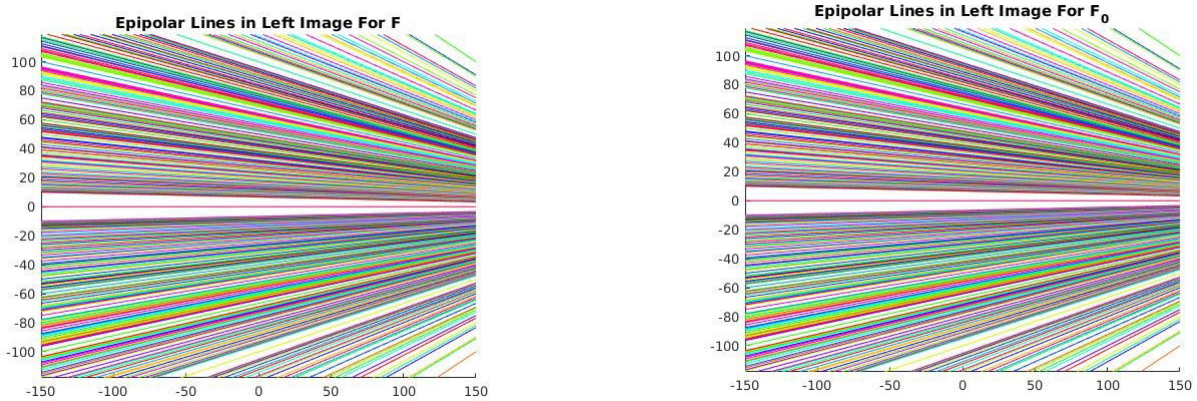
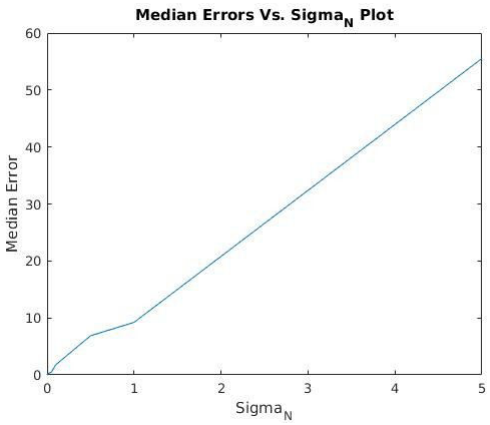


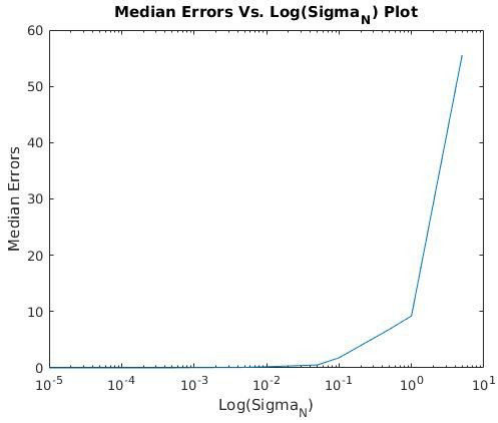
Figure A.2. A visual comparison between the epipolar lines generated by the matrix F and the matrix F_0 on my grid of 441 regularly spaced points. Again note that the two plots are essentially identical.

A.3.

For this and all subsequent questions in this section of the assignment, I am simply using linEstF to estimate the matrix F. I am using 8 random points that are being perturbed by Gaussian noise in each of their two first coordinates. In order to investigate the effect of noise on the error metric, I have chosen four different intervals of sigma_n. In each case I am plotting the error as a function of sigma_n on a normal and a log scale. During my trials I am adding unique random noise to each of the first two coordinates for each of the corresponding points 100 times. I collect the error during each of these 100 trials, compute the median of these errors and I then plot the median of these errors that I am observing for a given value of sigma_n. Once the median error has passed a value of one, I deem that the estimation of F has broken down. I first choose a massive interval to get an overview of trends of sigma_n:

i) sigma_n = [0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0, 5.0];

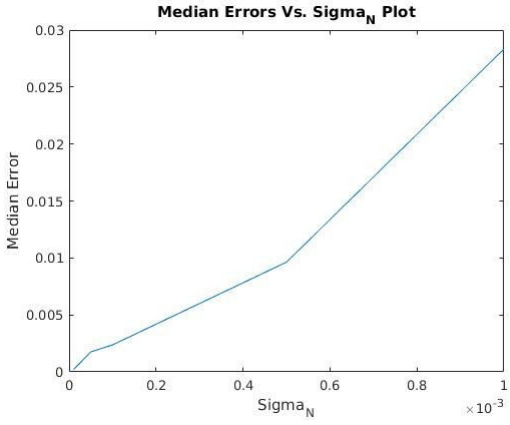


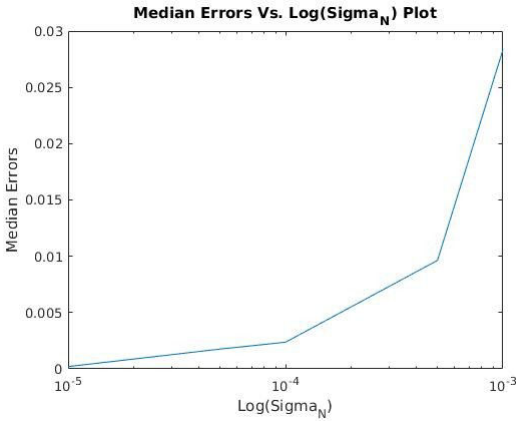


Case 1: Sigma_N values range over a large scale, but as one can see, the error begins to grow quickly well before a sigma_n value of 1.

Upong observing the error plot, it seems like the error has already begun to expand rapidly well before a sigma_n value of 1. It appears to be linearly growing for much smaller values of sigma_n, so I narrow it down to an exponential scale of values less than 0.001 and see what happens here:

ii) sigma_n = [0.00001, 0.00005, 0.0001, 0.0005, 0.001];

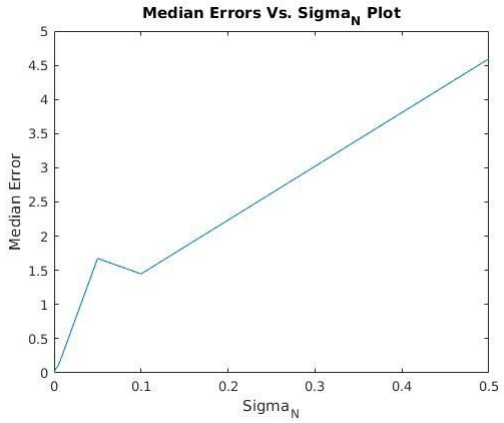


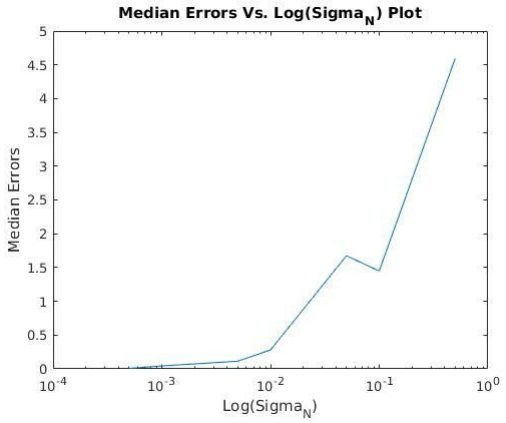


Case 2: I consider Sigma_N values that are much smaller than 1, specifically in an exponential range between 1.0E-5 and 1.0E-3. I find that the error is still proportional to sigma_n in this interval and that the error is still reasonably small.

In the above range I see that the error is still growing proportionally with sigma_n. Fortunately, the estimate still has not broken down in this range. I next try with a range of sigma_n values that are centered around 0.01 as I am still trying to narrow down where the blowup in error begins. By looking into this window I aim to find an approximate location where the blowup takes off. Based on the intial results in my large spread, I anticipate it should occur around this point.

iii) sigma_n = [0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5];

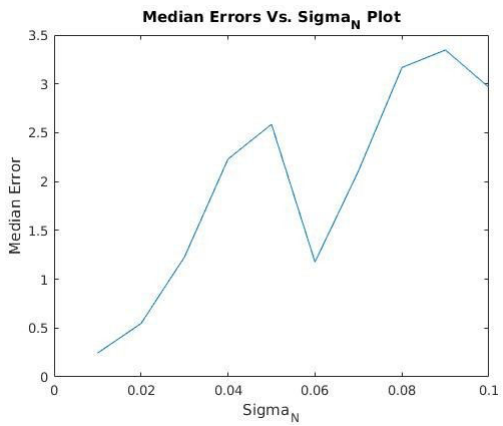


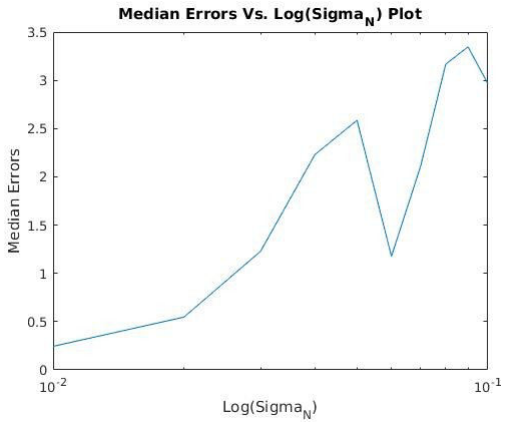


Case 3: Sigma_N is in a smaller exponential range centered around 0.01. Here it seems like the blowup in error passing a value of 1 tends to occur even before a value of 0.1.

Looking at these results, it appears that the error takes off well before a sigma_n value of 0.1. I will thus consider a strict interval between 0.01 and 0.1 to truly narrow down where the breakdown of our estimation of F occurs, which I deem to be when the median error reaches a value of 1.

iv) sigma_n = [0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1];





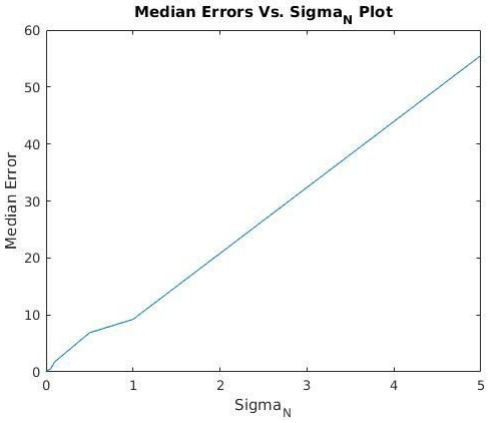
Case 4: I look into the strict interval between 0.01 and 0.1. Sure enough, the error blows up before 0.1.

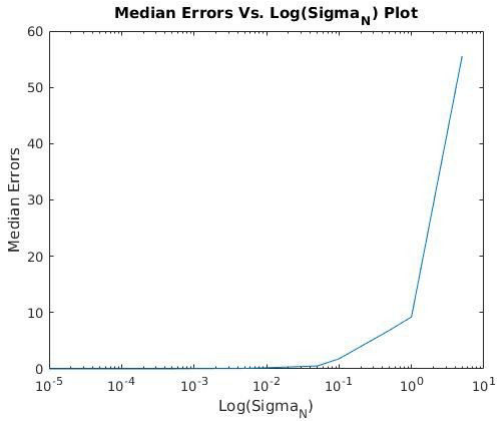
Investigating this interval, one can see that the error begins to take off at a sigma_n value of around 0.01. This is supported by the logarithmic plots I obtained for larger intervals as well. Thus, for sigma_n values beginning at 0.01, the error begins to take on a significant value relative to the size of the image (a value on the same order of magnitude as an error of 1). In particular, I found that the median error passes the value of 1 at a sigma_n value of around 0.03. Thus the maximum value of sigma_n that allows for small errors is a sigma_n value of around 0.01. Anything beyond that leads to errors that are large proportional to the size of the image.

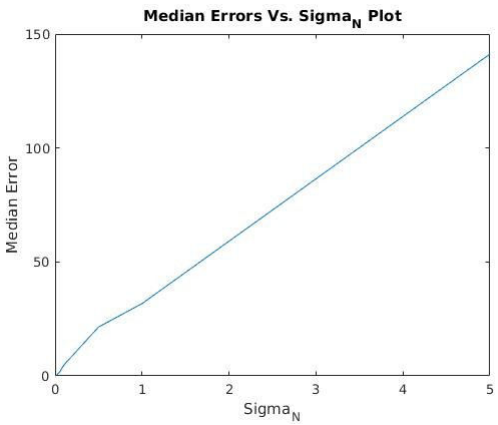
A.4.

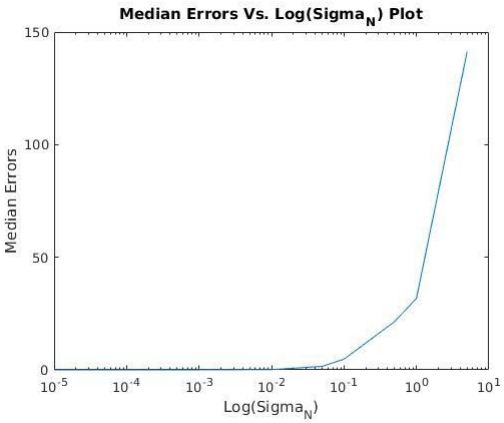
I repeat my experiments from A.3 on sigma_n, but without normalization of the imPts during the estimation of F using linEstF. First I take a large interval for sigma_n to investigate global patterns.

i) sigma_n = [0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0, 5.0];





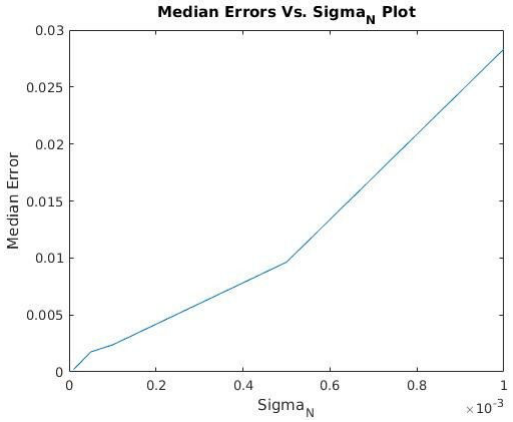


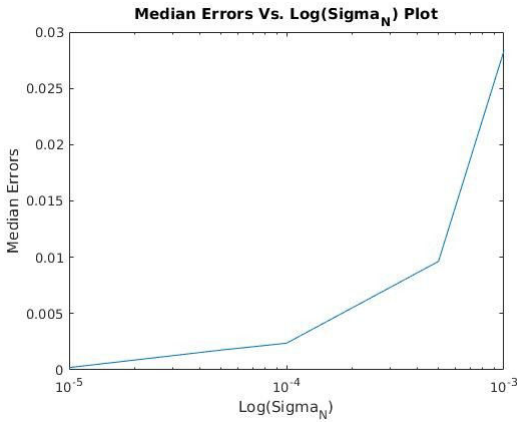


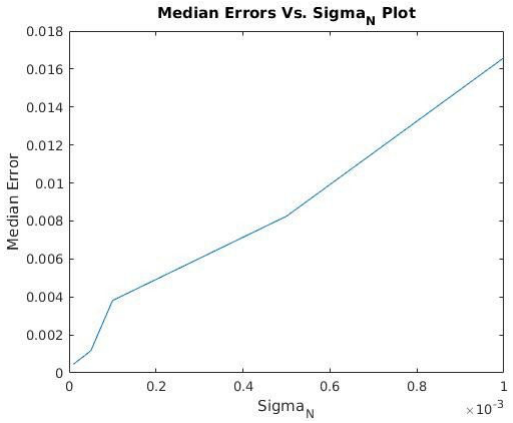
Case 1: Top Row: Normalized, Bottom Row: Unnormalized. With a large spread of sigma_n values I see that the error grows more rapidly and sooner than before. I will investigate smaller values to to see where the blowup begins.

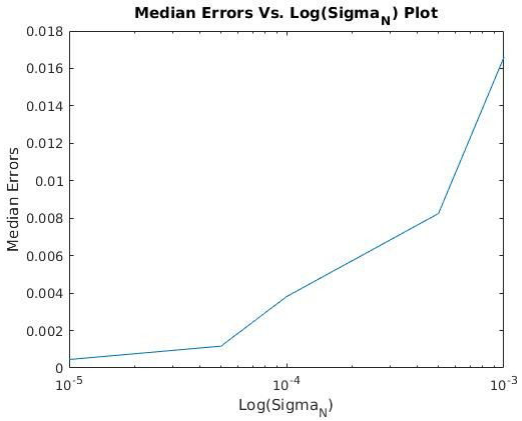
Here I notice that the error takes on larger values and seems to take off at a smaller value of sigma_n. In particular, it seems to be roughly doubled for sigma_n values of 1.0 and 5.0. Overall, not normalizing the imPts appears to make the approximation less stable than with normalization. To conclude that this is indeed the case however, I will need to investigate smaller ranges of sigma_n. Thus I will consider smaller values of sigma_n like I did in the previous question to narrow down at which point the approximation breaks down without normalization.

ii) sigma_n = [0.00001, 0.00005, 0.0001, 0.0005, 0.001];







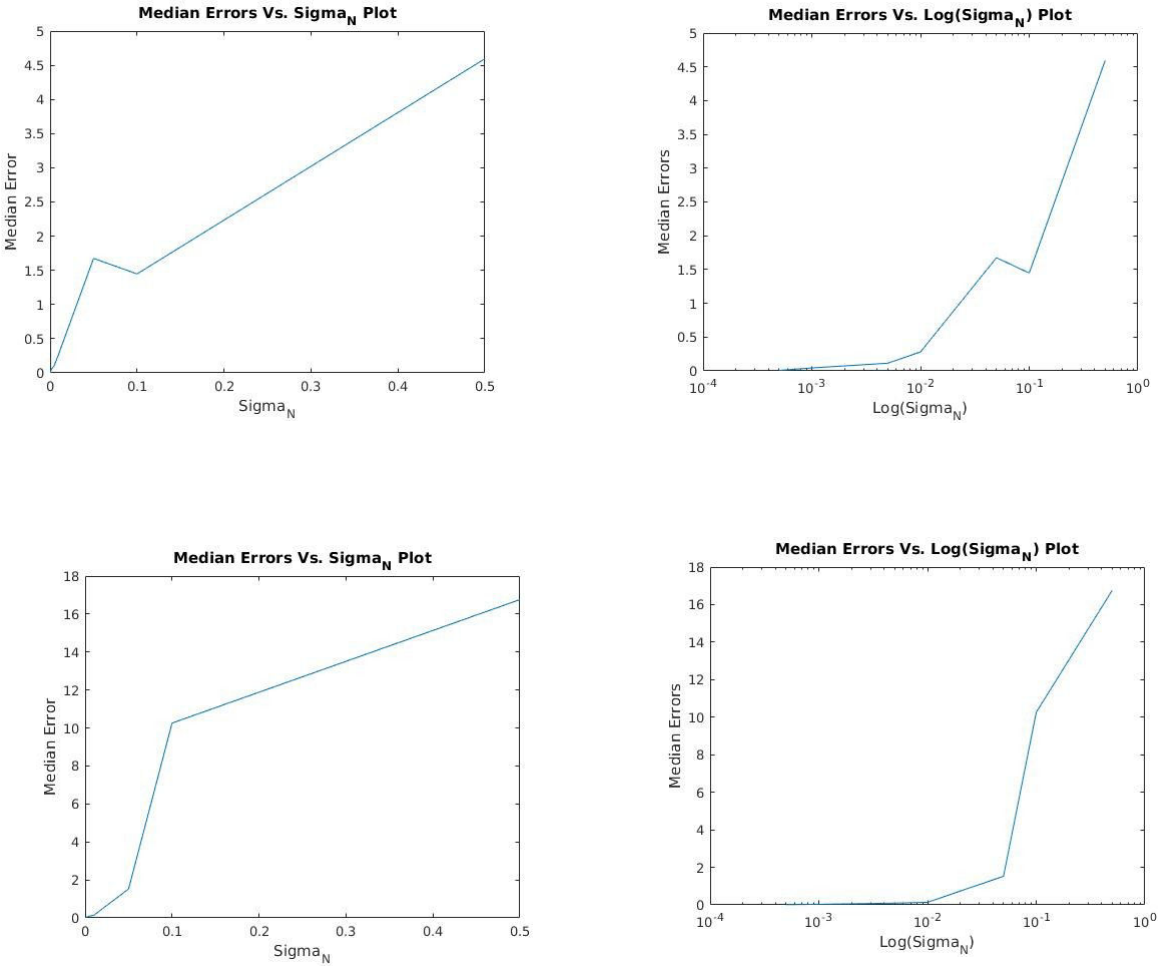


Case 2: Top Row: Normalized. Bottom Row: Unnormalized. I consider a very small exponential range for sigma_n. The results here are similar to those with normalization.

Looking at a very small range of exponential sigma_n values, I notice that the median error performance is similar as with normalization. The error is still proportional to sigma_n in this range

and F has not broken down (the error has not passed a value of 1). I will thus consider larger values of sigma_n to deduce where the divergence occurs without normalization. To do this, I will look into a larger scale exponential range of values of sigma_n being less than 0.5

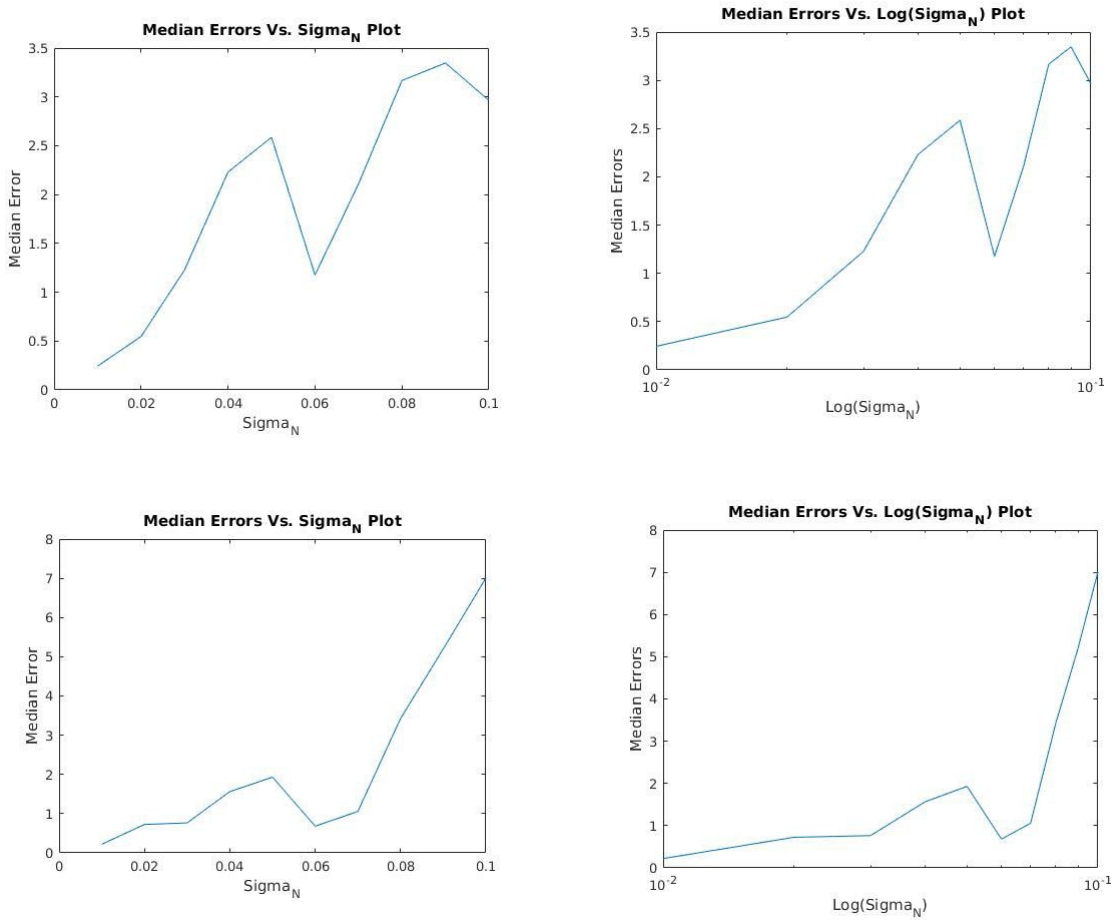
iii) sigma_n = [0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5];



Case 3: Top Row: Normalized. Bottom Row: Unnormalized. Here the difference is striking. The error is much larger for the unnormalized estimation than the normaled one.

In this range it becomes rapidly apparent that normalization makes a difference. The median error for a sigma_n value of 0.1 has increased by a factor of nearly 5. Similarly for a sigma_n value of 0.5 it has essentially tripled. Based on my results here, it is again the case that the blowup is occuring before a sigma_n value of 0.1. I suspect it will be somewhere in the range of 0.01 and 0.1 like before. I will thus investigate this range next.

iv) sigma_n = [0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1];



Case 4: Top Row: Normalized. Bottom Row: Unnormalized. A smaller exponential range between 0.001 and 0.1 Unlike in the normalized case, the error grows much rapidly and surpases a value of 1 at a lower value than before.

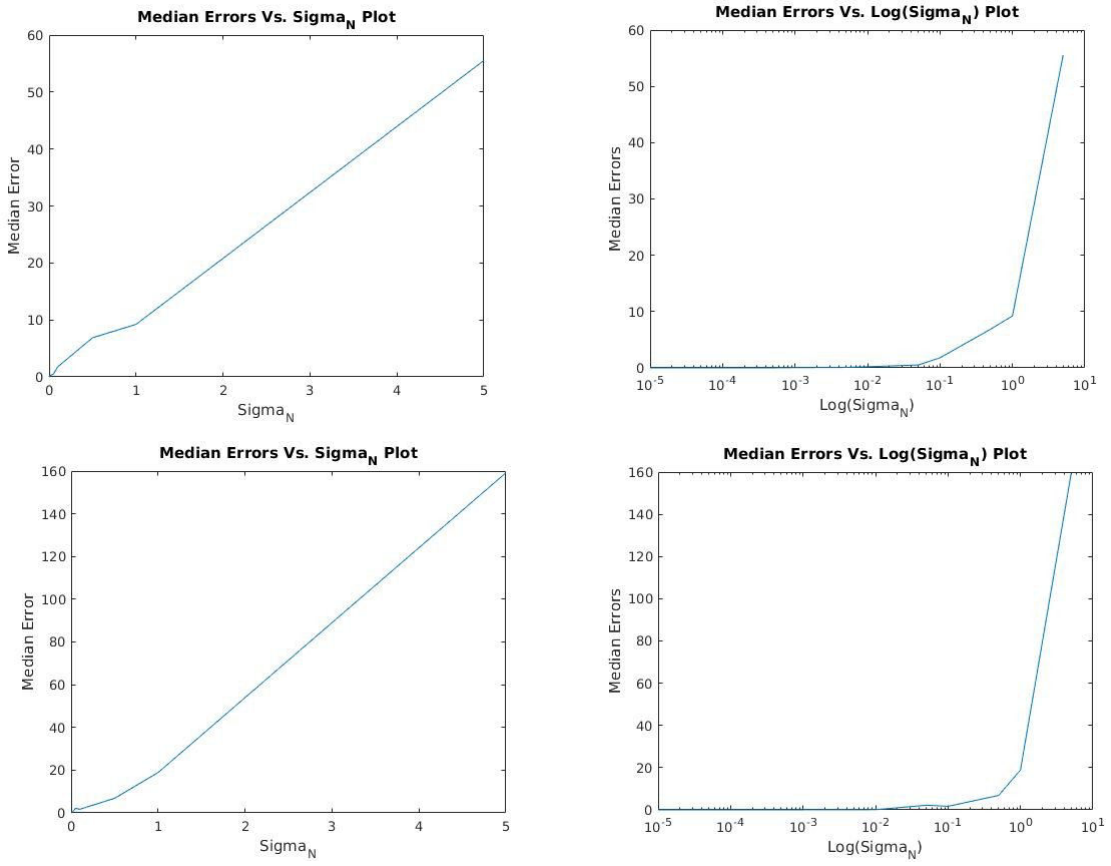
Thus, as with the normalized case, the error begins to blow up around a sigma_n value of 0.01 (as it has reached the same order of magnitude as an error of 1 at this point). At this point our estimation breaks down. The difference here however is that the errors are much larger than in the normalized case. Based on my results the error values are similar for sigma_n values between 0.01 and 0.06, but in the unnormalized case, it passes the value of 1.0 at a sigma_n value of 0.02, compared with 0.03 in the normalized case. Past sigma_n values of 0.06, the unnormalized approximation attains error values nearly 2 times higher than in the normalized scenario. Thus one can confidently say that the approximation is less stable without normalization. While it does not really change the point at which errors blow up, it does affect the error values one observes once divergence occurs.

A.5.

The true F_0 matrix does not change if I change sclZ. This makes sense, as the computation of F_0 is done independently of the scale of the object the cameras are observing. It depends solely on the extrinsic and intrinsic camera matrices and their relationship to each other. Here we note that even if the sclZ changes, the extrinisc and intrinsic camera parameters remain constant as their positions are not changed with scaling, hence F_0 does not change either. The only thing that does change here is the position and spread of the imPts in the image. Consequently our approximation of F using linEstF will be affected.

In order to estimate how the changing of scale has affected our estimation of F, I will again consider various values of sigma_n. To start I will look at a large exponential interval of values.

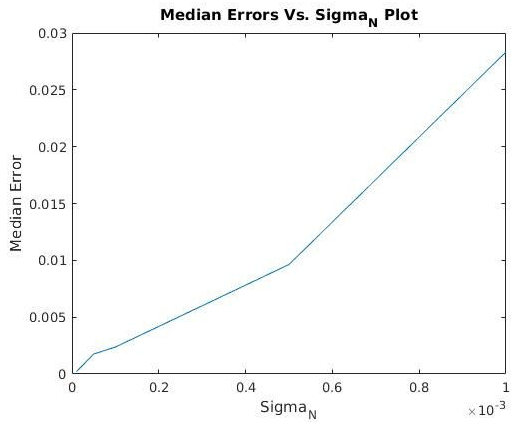
i) sigma_n = [0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0, 5.0];

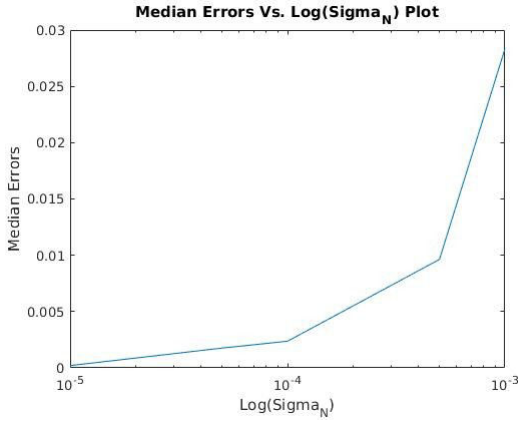


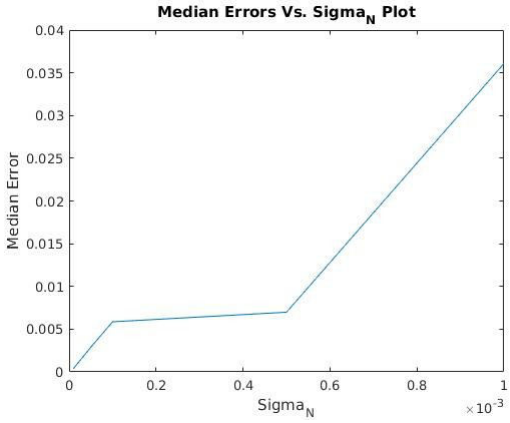
Case 1: Top Row: sclZ of 1.0. Bottom Row: sclZ of 0.1. Over a large spread of values the error rate for a smaller sclZ is higher. It appears as though it may be more stable than the normal scale of 1.0 at small values of sigma_n however.

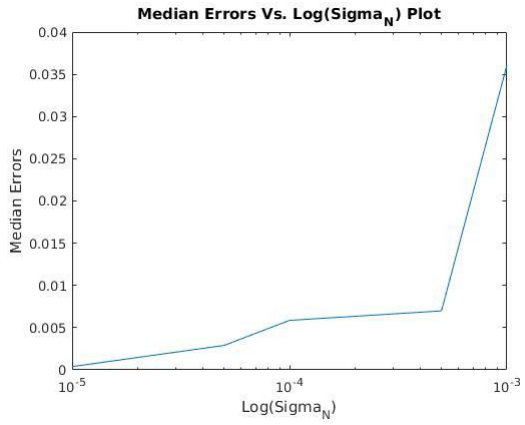
Looking at the plots here, it appears that the error takes on higher values in the smaller sclZ case than before. For sigma_n values of 1.0 and 5.0 the median error value is more than doubled. It appears that it may in fact be less erroneous for smaller values of sigma_n however (less than one). I will look into a very small exponential scale of sigma_n next to investigate this.

ii) sigma_n = [0.00001, 0.00005, 0.0001, 0.0005, 0.001];





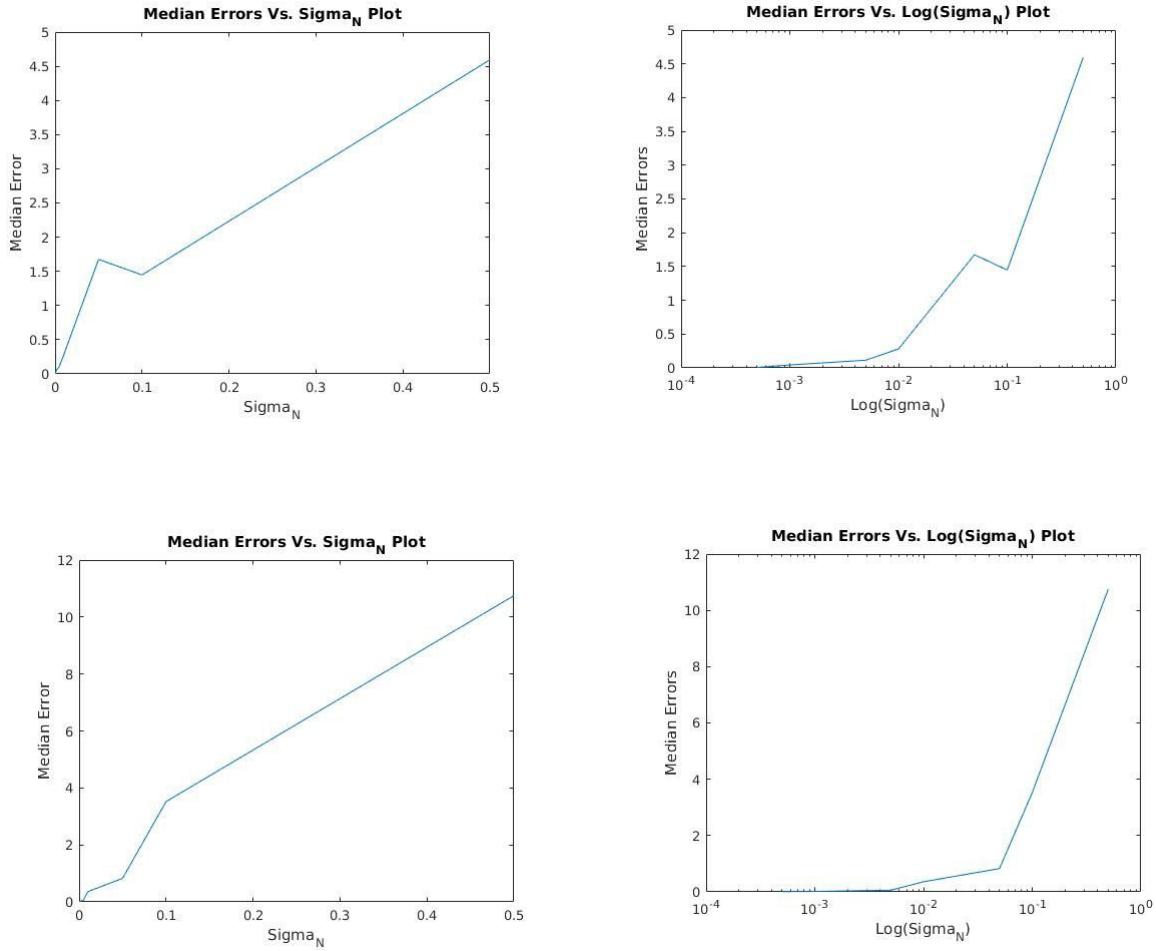




Case 2: Top Row: sclZ=1.0. Bottom Row: sclZ=0.1. The results appear similar here.

Looking at the small exponential range of sigma_n values from 1E-5 to 1E-3, it appears that the performance is similar regardless of the sclZ factor. In this range again, the error remains roughly proportional to the value of sigma_n. The error has not begun to diverge in this range, as was the case before. I suspect it will diverge in a larger range of values capped at 0.5 like before, so I will investigate that next.

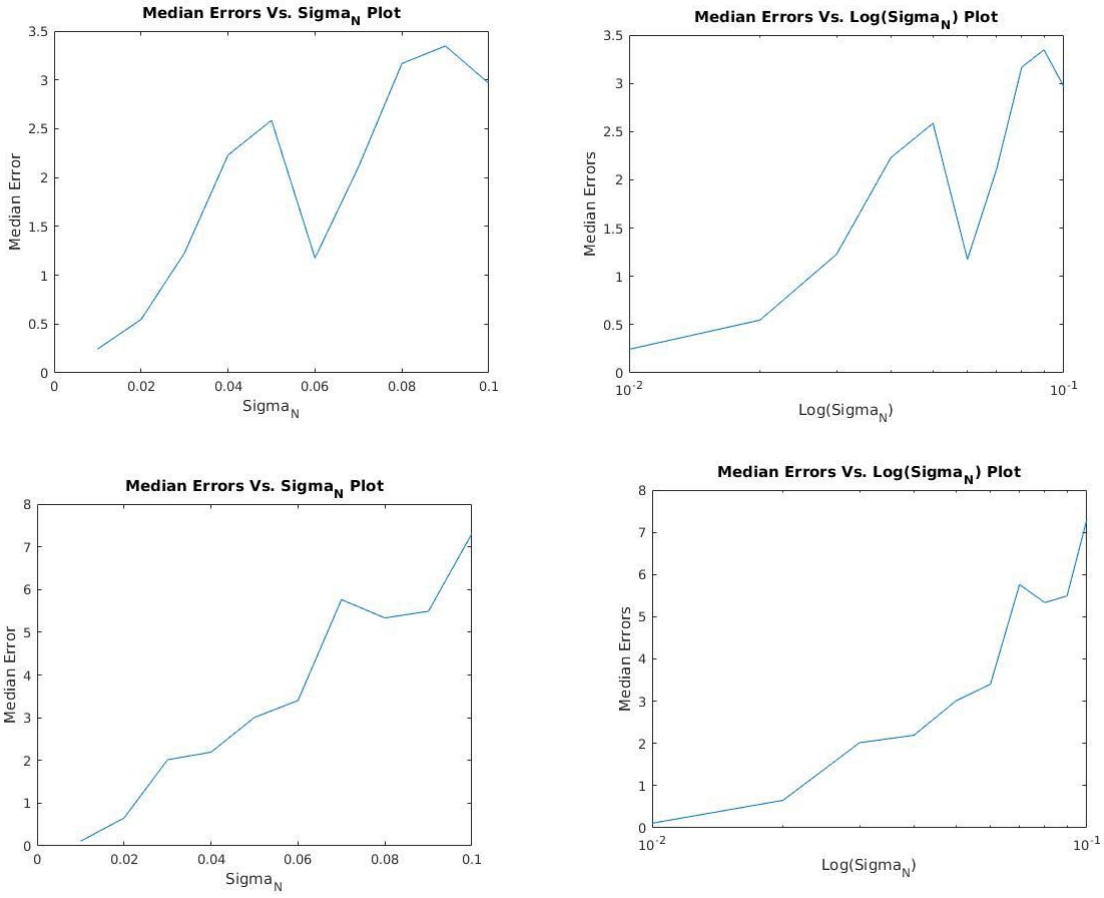
iii) sigma_n = [0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5];



Case 3: Top Row sclZ=1.0. Bottom Row: sclZ=0.1. As was the case with unnormalized pts, the error is noticeable in this range. It appears to take on values almost double the case where sclZ=1.0

Looking into this larger exponential range of sigma_n values the error differences are noticeable. The error is nearly twice as large when the sclZ=0.1 for sigma_n values of 0.1 and 0.5. At smaller sigma_n values the behaviour is similar however (although for sigma_n =0.5 it appears that the error is smaller for sclZ=0.1). The approximation seems to diverge again around a value of 0.01, so I will turn to a final interval of 0.01 to 0.1 like before to see where the approximation truly starts to break down.

iv) sigma_n = [0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1];



Case 4: Top Row: sclZ=1.0. Bottom Row: sclZ=0.1. Divergence behaviour is similar here, but errors are twice as large with the smaller scale.

Looking into the final narrow range of sigma_n values, I observe that the median error values are approximately twice as large for identical values of sigma_n when sclZ=0.1 as for sclZ=1.0. In both cases, the approximation of F begins to break down at a sigma_n value of 0.01. The median error crosses a threshold of 1 at a sigma_n value of 0.03 when sclZ=1.0 and at a sigma_n value of ~0.02 when sclZ=0.1. So as with the normalized case, while the point of error divergence has not really changed, the actual error values have increased by changing the scale. Thus by changing the scale we have made the approximation more sensitive to image position noise. This makes sense as image points are now spread over a smaller range of values, implying that they will be more sensitive to smaller perturbations, consequently affecting our estimation of F. As a result our approximation is worse than in the default case.

A.6.

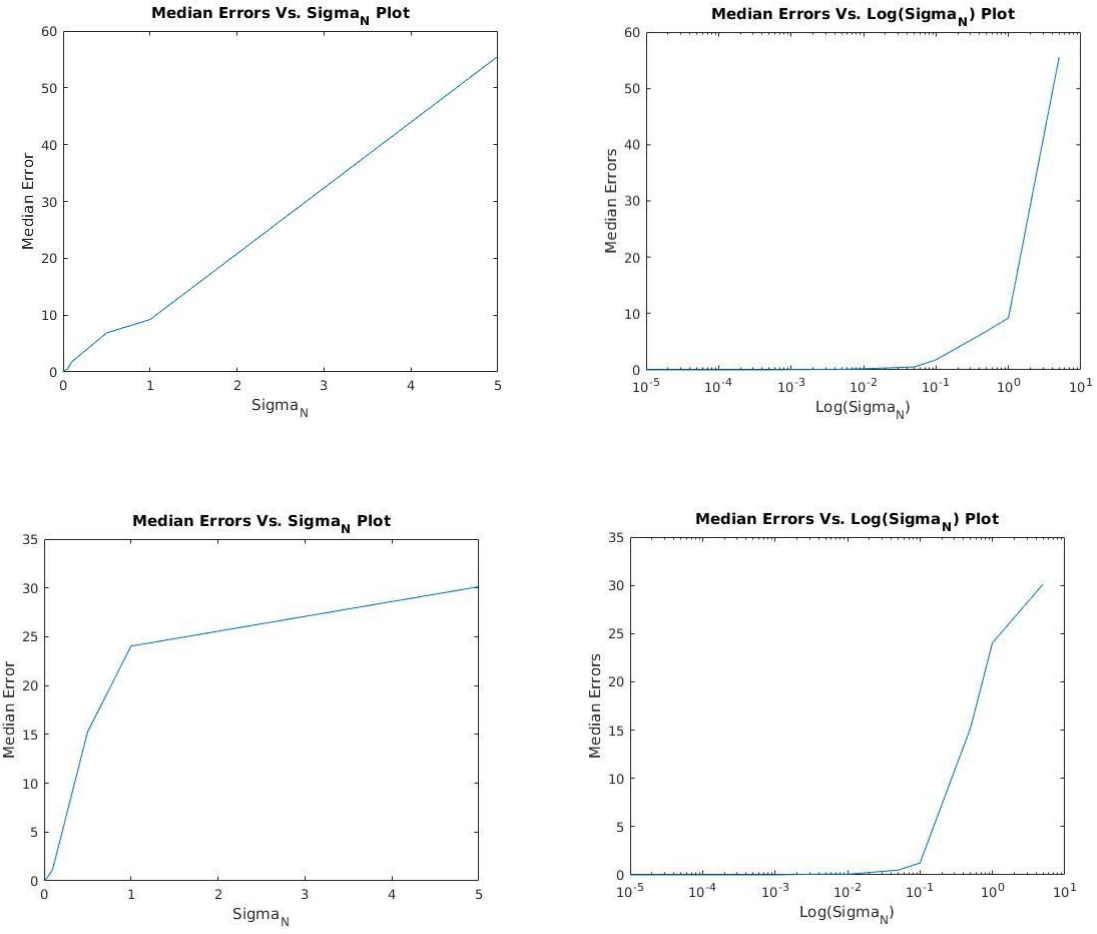
As with the previous questions, I will investigate various ranges of sigma_n to see how changing the cameras' nodal points to (5, 0, -150)^T and (-5, 0, -150)^T respectively affects the approximation. I note that as expected my ground truth matrix of F_0 has changed. Computing it like before, it is now:

F_0 =

```
       0   -0.0000         0
  -0.0000         0    0.0999
       0   -0.0999         0
```

To start, I look towards a large exponential range of sigma_n values in order to observe global trends like before.

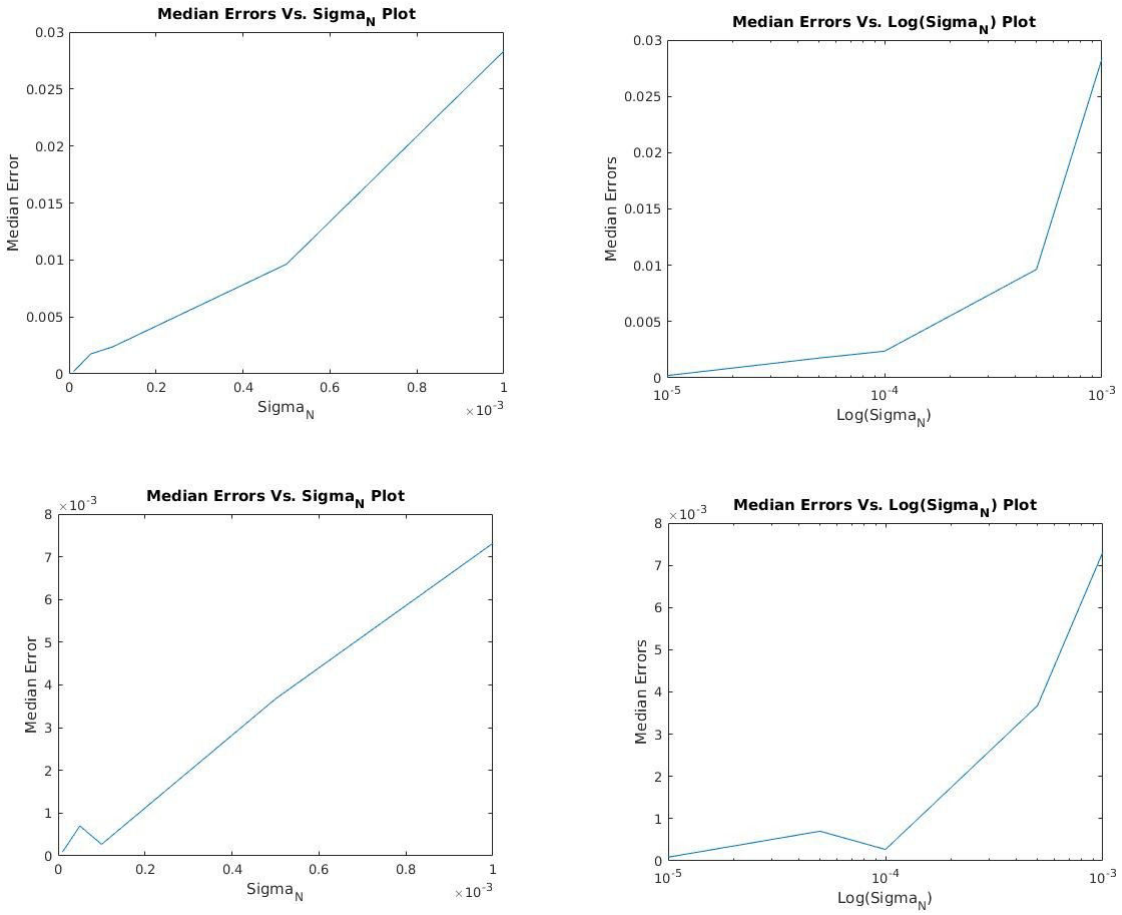
i) sigma_n = [0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0, 5.0];



Case 1: Top Row: Default camera separation. Bottom Row: Smaller Camera Separation. It appears that the error has been approximately halved here.

Looking at a large global range of sigma_n values, it appears that the error is smaller than in the default case for a sigma_n value of 5.0. For very small values of sigma_n the error seems to be approximately the same, but interestingtly the error is higher for a sigma_n value of 1.0 in the case of smaller camera separation. Consequently, it is difficult to conclude any concrete trends from this large exponential scale in this case. In order to conclude how the camera separation has affected the approximation, error divergence and overall error values I will thus look to a very small exponential scale of values.

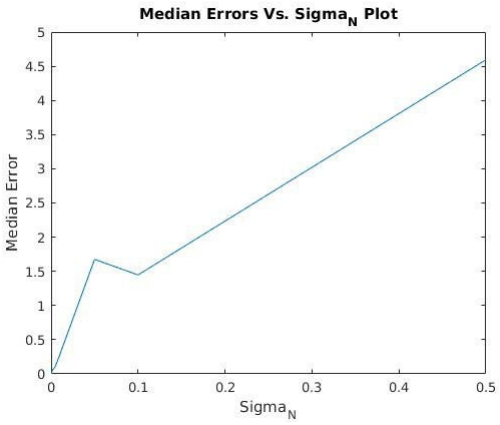
ii) sigma_n = [0.00001, 0.00005, 0.0001, 0.0005, 0.001];

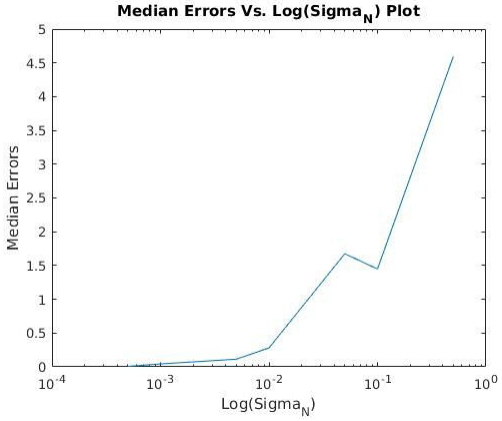


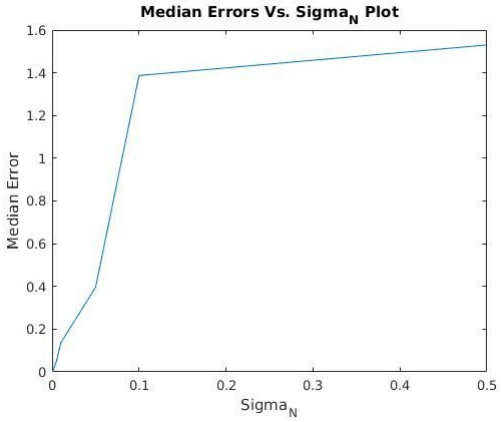
Case 2: Top Row: Default camera separation. Bottom Row: Shortened Camera Separation. Performance is comparable, but errors are smaller by a factor of 10 in the shortened case.

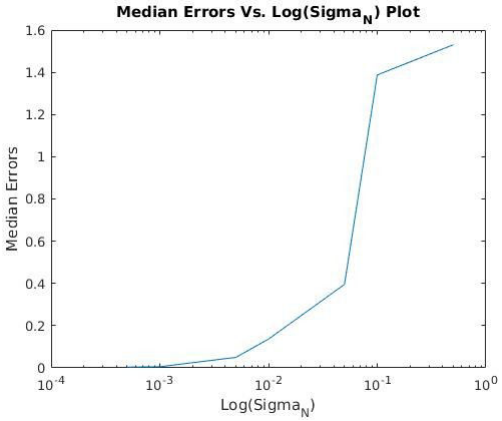
Looking to small exponential values between 1E-5 and 1E-3, the error seems to behave similarly to the default setup in this range. In both cases, the error is roughly proportional to sigma_n, and it has not diverged yet. I will thus consider a larger exponential range of sigma_n values, again with sigma_n less than 0.5 which should capture the point of error divergence like in the default case.

iii) sigma_n = [0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5];





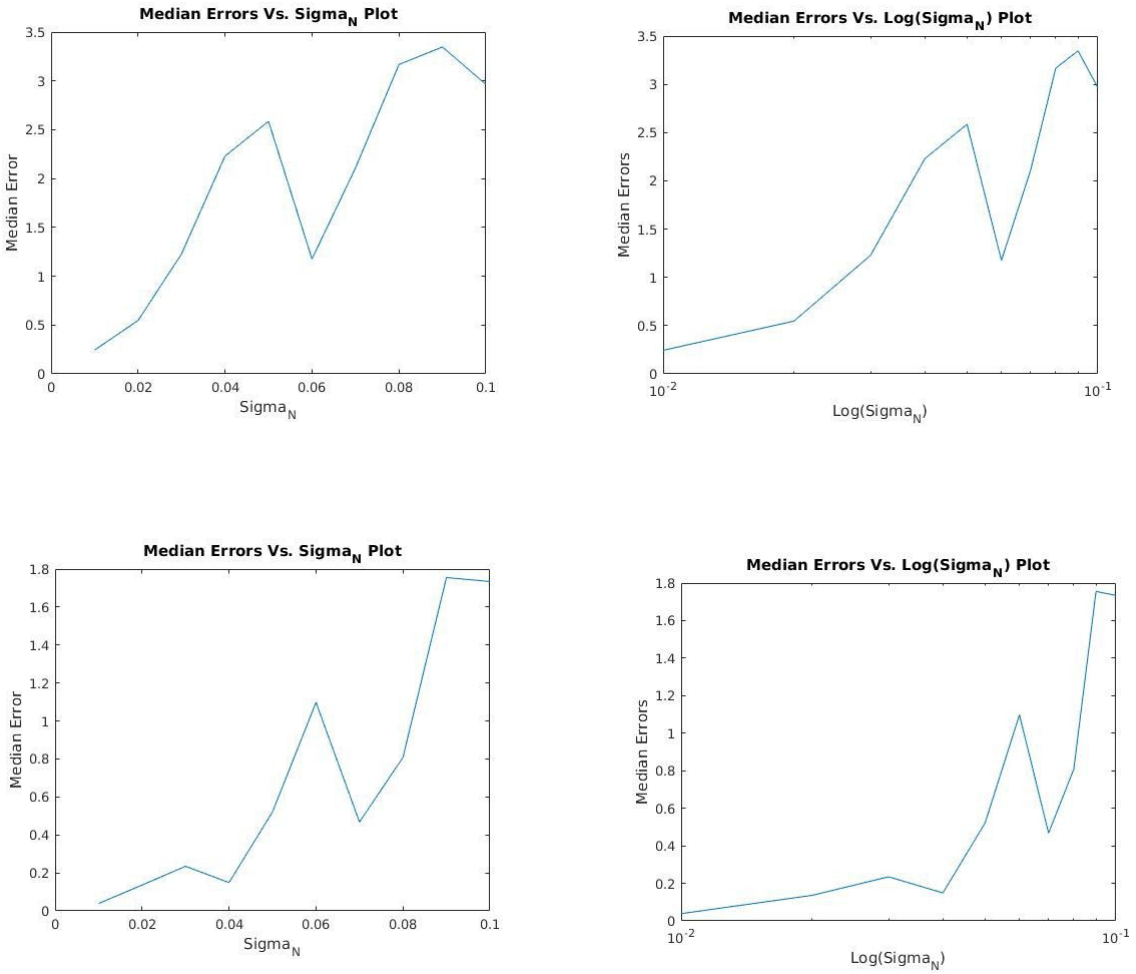




Case 3: Top Row: Default camera separation. Bottom Row: Shortened Camera Separation. Here the difference is noticeable. In the shortened case the error is more or less half as large.

Looking to this larger exponential range a difference has become noticeable between the two approximation error plots. In particular, in the shortened camera, the error does not pass my median error failure threshold of 1.0 until sigma_n reaches a value of 0.1. This contrasts with the default setup where the error passes this threshold at a sigma_n value of around 0.05. One can also see that the error rates for the shortened setup are overall about half as large as the default setup (with the exception of sigma_n =0.1, where it is comparable). To truly deduce that the error is halved however, I will look to a narrow interval between 0.01 and 0.1 like before. By doing so I will be able to conclude that the shortened camera is less sensitive to noise than the default setup.

iv) sigma_n = [0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1];



Case 4: Top Row: Default Camera Separation. Bottom Row: Shortened Camera Separation. The difference is settled here. The error is roughly half compared to the default setup.

Turning to the final narrow range of sigma_n values I can safely conclude that the shortened camera separation leads to an approximation of F that is less sensitive to Gaussian noise. Looking at the error plots, one sees that in the default setup the median error rates are roughly doubled for identical values of sigma_n compared to the shortened setup. Additionally, the median error does not cross my failure threshold of 1.0 in the shortened case until sigma_n is 0.06. This contrasts with the default setup where this occurs at a sigma_n value of 0.03. Consequently, one can see that not only is the shortened camera less sensitive to noise, but it reaches a failure point at sigma_n values roughly double that of the default setup. Hence the estimation of F is more tolerant to image position noise when the cameras are closer together.

B.1 To begin, expand out $H\vec{p}_K$.

$$H\vec{p}_K = \begin{pmatrix} \vec{h}^{1T}\vec{p}_K \\ \vec{h}^{2T}\vec{p}_K \\ \vec{h}^{3T}\vec{p}_K \end{pmatrix}$$ where the j^{th} row of the homography matrix H is written $\vec{h}^{jT}$.

As $\vec{q}_K = (q_{K,1}, q_{K,2}, 1)^T$, we yield an explicit equation for the cross product $\vec{q}_K \times (H\vec{p}_K) = \vec{0}$ (as $\vec{q}_K$, $H\vec{p}_K$ are in the same direction)

$$\vec{q}_K \times (H\vec{p}_K) = \begin{bmatrix} q_{K,2}\vec{h}^{3T}\vec{p}_K - \vec{h}^{2T}\vec{p}_K \\ \vec{h}^{1T}\vec{p}_K - q_{K,1}\vec{h}^{3T}\vec{p}_K \\ q_{K,1}\vec{h}^{2T}\vec{p}_K - q_{K,2}\vec{h}^{1T}\vec{p}_K \end{bmatrix} = \vec{0}$$

Now as $\vec{h}^{jT}\vec{p}_K = \vec{p}_K^T\vec{h}^j$ for $j = 1, 2, 3$ we get 3 equations in the entries of H, written as:

$$(1) \quad \begin{bmatrix} \vec{0}^T & -\vec{p}_K^T & q_{K,2}\vec{p}_K^T \\ \vec{p}_K^T & \vec{0}^T & -q_{K,1}\vec{p}_K^T \\ -q_{K,2}\vec{p}_K^T & q_{K,1}\vec{p}_K^T & \vec{0}^T \end{bmatrix} \begin{bmatrix} \vec{h}^1 \\ \vec{h}^2 \\ \vec{h}^3 \end{bmatrix} = \vec{0}$$

This forms a system of equations of the form

$A_K\vec{h} = 0$, where A_K is a 3×9 matrix and $\vec{h}$ is a 9-vector made up of the entries of H.

$$\vec{h} = \begin{pmatrix} \vec{h}^1 \\ \vec{h}^2 \\ \vec{h}^3 \end{pmatrix}, \quad H = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix},$$ where h_i is the i^{th} element of the 9-vector $\vec{h}$.

Note however that in (1) it is possible to obtain the third row by summing $-q_{K,1}$ times the first row and $-q_{K,2}$ times the second row. So the third row in (1) can be omitted as only the first two are linearly independent.

We thus yield the equation:

$$\begin{bmatrix} \vec{0}^T & -p_k^T & q_{k,2}p_k^T \\ p_k^T & \vec{0}^T & -q_{k,1}p_k^T \end{bmatrix} \begin{pmatrix} \vec{h}^1 \\ \vec{h}^2 \\ \vec{h}^3 \end{pmatrix} = \vec{0}$$

which can be written as $A_k\vec{h} = \vec{0}$
where A_k is a 2×9 matrix, $\vec{h}$ is a 9-vector.

With N points, we stack the N 2×9 matrices A_k into a total matrix A that is $2N\times 9$. Thus constraint matrix A is computed.

Our goal now becomes to find an $\vec{h}$ that minimizes $\|A\vec{h}\|$ subject to the constraint that $\|\vec{h}\|=1$ (to avoid the trivial $\vec{h}=\vec{0}$ solution).

The solution of this can be found by computing the SVD of A, defined as UDV^T, where U is an orthogonal-column matrix of size $2N\times 9$, D is 9×9 diagonal matrix and V is a 9×9 orthogonal matrix. To solve the minimization problem of $\|A\vec{h}\|$, take the last column of V, where $A=UDV^T$ is the SVD of A.

At this point we have obtain a 9-vector $\vec{h}_0$. It is not the true solution to our original problem however as we have normalized the coordinates. Thus, first reshape the 9-vector $\vec{h}_0$ into a 3×3 matrix H_0. Then one must denormalize. This is done by computing

(2) $H = T_2^{-1} H_0 T_1$, where T_1 is the matrix used to normalize points $\vec{p_k}$ and T_2 is used to normalize points $\vec{q_k}$.

To see that (2) holds, note that by normalizing we have $\vec{q_k}' = T_2\vec{q_k}$ thus $\vec{q_k} = H\vec{p_k} = T_2^{-1}\vec{q_k}'$

Similarly, $\vec{q}_k' = H_0 \vec{p}_k'$, where $\vec{p}_k' = T_1 \vec{p}_k$

Hence $\vec{q}_k = H\vec{p}_k = T_2^{-1}\vec{q}_k' = T_2^{-1} H_0 \vec{p}_k' = T_2^{-1} H_0 T_1 \vec{p}_k$.

So we can simplify to yield $H\vec{p}_k = T_2^{-1} H_0 T_1 \vec{p}_k \iff H = T_2^{-1} H_0 T_1$.

Finally, one must rescale H so that the sum of squares of its elements is 1. I do this by reshaping H into a 9-vector $\vec{h}$, then setting:

$$\vec{h} = \vec{h} / \text{norm}(\vec{h}).$$

Then I reshape $\vec{h}$ back into a 3×3 matrix to get my final solution H. ∎

Following the above derivation I am able to generate a homography estimation H in the file linEstH.m.

B.2.

In order to use RANSAC to estimate the 2D homography in grapple2DHomog.m I had to modify the metric used to qualify inliers in the estimation process. For my uses, I am using the following metric:

|| (q_k – Hp_k) + (p_k – H^-1q_k)|| < E, where E=4, and we are trying to approximate H s.t. Hp_k = q_k and p_k = H^-1q_k

With this metric I am able to measure the distance between points mapped using the homography to their corresponding points. If I find points that satisfy the above inequality, I keep them as my inliers. I then iterate using RANSAC and linEstH to refine my estimation.

After conducting RANSAC trials to approximate the homography, I run my script grapple2DHomog and I generate the following output, based on the two images we are given as input to our script:

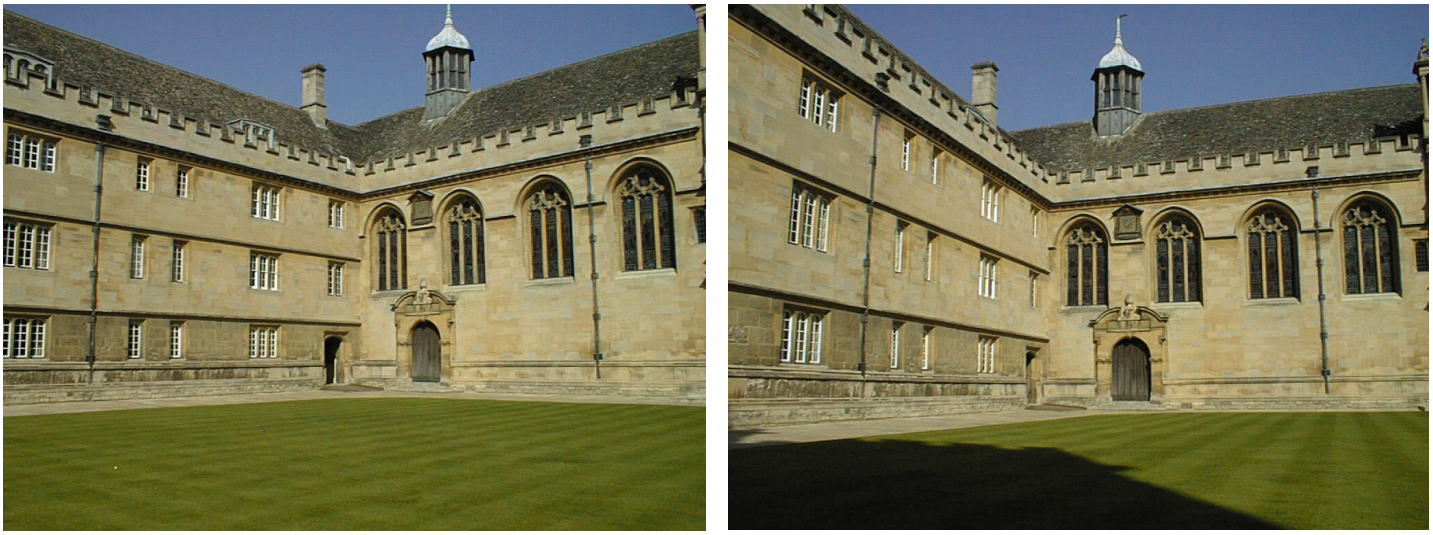


Figure B.1. ImLeft and ImRight Side by Side without Applying a Homography

Figure B.2. imLeft warped by the homography to correspond to imRight

Figure B.3: imLeft and imRight warped by the inverse of the homography to correspond to imLeft

One can see that the homography works as expected. It is doing its best to map what it deems to be corresponding points in each image to the other. As a result, the warped images seem like warped, but relatively decent approximations of the ground truth images. Obviously they are not exact replicas, but looking at parts of the image like the door, one can see that it is rotating/translating image points so they are oriented similarly and so that the overall images resemble each other.