Milestone 3

# Anton Khabbaz

Penn Key: akhabbaz

# Data Organization

This project involves analyzing a set of images to get the best correspondences and the best possible estimate of positions in the images. We have 6 images, a K matrix and files of correspondences. We also developed through milestones 1 and 2 linear and Ransac methods to estimate positions and locations.

Since we have 6 images, there are 15 possible correspondences beginning with image1 and image2. The results of each stage of analysis is stored in a separate directory under Milestone3. Analysis of the pair of images 1 and 2 are kept in directory 1\_2 etc, and the pair 5 and 6 is kept in directory 5\_6.

Since there are lots of pairs and lots of different parameters, those are kept in a structure or class of ImagePair.m records. Those records hold all the relevant information for the pair say 1 and 2. For example the directory name ‘1\_2’, the F matrix, the essential matrix, the camera poses, the ransac parameters to select fits and the filenames for all the analysis files are kept in that structure. In the root directory (milestone3) there is a structure StartData.m which holds 15 records of all the data pairs. There is also a copy of the individual records from the specific directory, for example. The pairs are indexed , 1 refers to pair 1\_2 , 5 refers to pair 1\_6.

Not every pair of images produces correspondences, so some pairs are empty. For example, there are no correspondences from image 1 and 5, 1 and 6, 2 and 5 and 2 and 6.

In running the code, I stay in directory Milestone3 and not in the subdirectories. The path names are relative to the root, so what is stored is the relative name from the root directory. The root is used to make directories but it is not stored in the data structure. This package should run on another machine if you start in the desired root directory and the data files are in the project defined path.

## Routines to Organize and produce the linear Estimate:

## Milestone3.m

The first step is to run Milestone3.m. This has the path to the images and the matching[1..5].txt. It also reads in the calibration file, and creates the paths for milestone1 and milestone2 code. The main routines there are not changed nor moved, so that if bugs are found they can be corrected for all the code that uses them. Milestone2 and milestone1 paths are stored in startData.mat. The K matrix is read in readCalibrationMatrix.m. Next, it creates relative paths for all the analysis, and makes subdirectories , ‘1\_2’, ‘1\_3’, etc. Now it initializes the 15 records allotting space for all the variables and getting specific filenames for the figures. It also sets the Ransac thresholds and parameters. These get stored in the ‘pairs’ record in the startData.mat file. This file should be run once at the beginning of the analysis.

## LinearEstimate.m

This file does the linear Estimate on the pairs. Call it with a pair number (1 to 15), and an optional threshold for Ransac. This uses Ransac to get an Estimate of F, and makes graphs to determine if the correspondence is good or not. It essentially completes the analysis in Milestone 1 and 2 for any pair of images. It is important to generate figures for this so that it is clear how good the correspondences are. The data for this is stored in StartData.m and also in the pair class in the subdirectory of the image pair.

### Step1: load the data and images

We use the pair number to get the path and the dataname. We then read the correspondences using ‘readCorrespondenceFile’(filename), and that checks to see how many records are stored, and reads each line, creating a record, called ‘MatchedPoint’, that stores all the information about the correspondence. It stored the rgb value, the correspondence in the file itself, and the corrrespondences and their filenumber for all the other images. An array of MatchedPoint records is created that has all the data from one correspondence file.

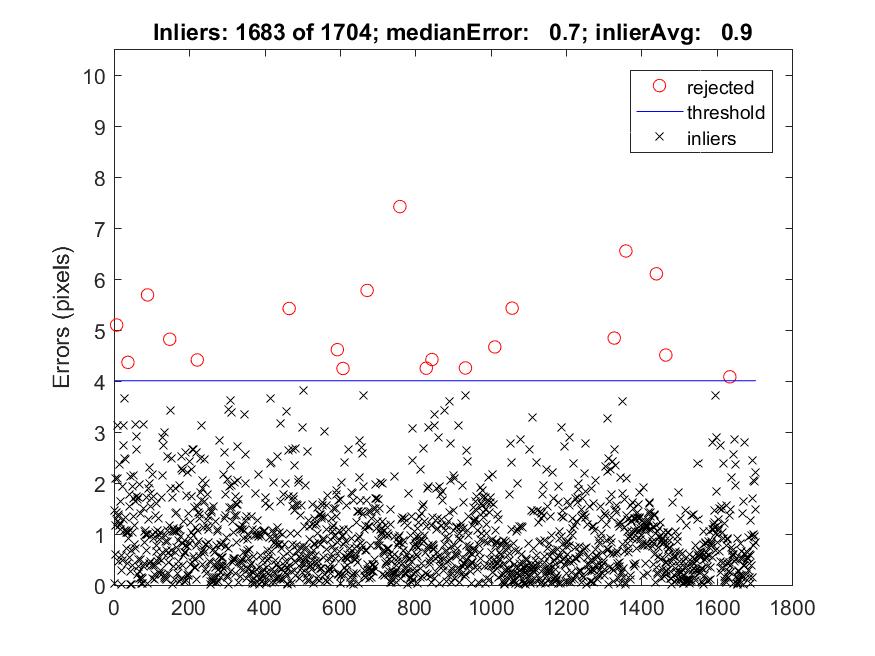
The first number of this pair determines which file has the correspondences and the second selects all the correspondence in that file that refers to the particular image. ‘CorrespondencesFromMatchedRecords(records, imagenumber)’ takes that list of records and only selects those correspondences that relate the pair of images. This creates 2 n X 2 correspondences that holds the matching correspondences.

Now it turns out in the data file, some of these are duplicated. One can see this in the example in the project description. I don’t think the duplicates should be there since each correspondence should be weighted the same. Thus ‘removeDuplicateRow’ gets rid of them and makes all unique records.

Now Ransac is performed and the next code came from milestone2r2 but the steps are. Estimate the number of iterations needed, perform ransac, and estimate the errors.

Here is the error plot for one pair, images 2 and 3, pair number 6. One can see here that most if not all

Of the correspondences were kept and that was fit by Ransac.



Here are the correspondences:

Before: (same pair)



After:



One very important part of this pair is that the correspondences span the entire image and there are plenty in the forground. The tree and building give both very good correspondences.

### Ransac Improvements

The ransac I used picks random correspondences but in a stratified manner. This is done in a function called ‘striatedSampling’ in milestone2. A random permutation is made so that no sample is picked twice. The left image is divided into a grid (usually 16 boxes), and once a point is picked from a box, the next random point will only be accepted if it falls in another box. If in the end the 8 points are not found, then the points in duplicate boxes (not duplicate points) are taken. This improvement gets better F fits because most fits tried will already be sampled from all over the image.

Here in the Ransac step, some pairs---1 5, 1 6, 2 5, 2 6 produce no correspondences.

### Estimate E, R, C and check the Pose:

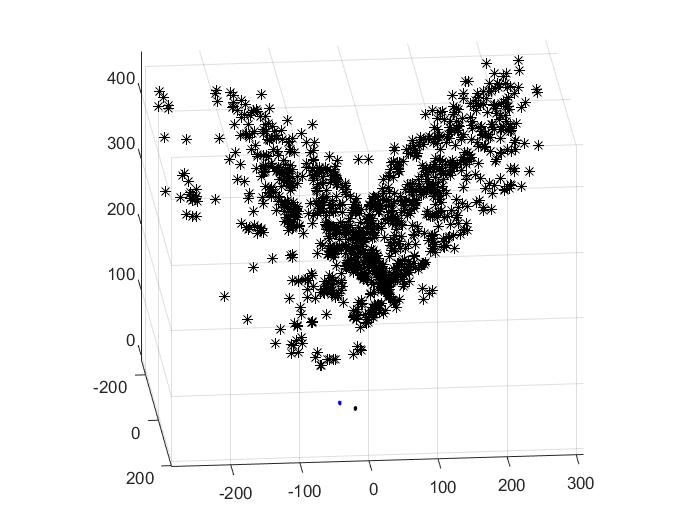
Next, the F matrix is made to have the degeneracy it must have. The 3rd singular value is zeroed, the E is created and R, C, all possibilities are made. The one that meets the chierality condition is accepted. This was done in milestone1 and here the main point is to use that and also verify that code is working with the new data. Here is the camera pose for this data:

### 



Here the data looks good. That is the 3D point reconstructions look like where you would expect them given the picture. One can see here that all the points are in front of the cameras as they should be. In other sets this is not always the case. It is also clear if one looks closer, one can see the two cameras and the wall of correspondences. The cameras are pointing in the same direction.

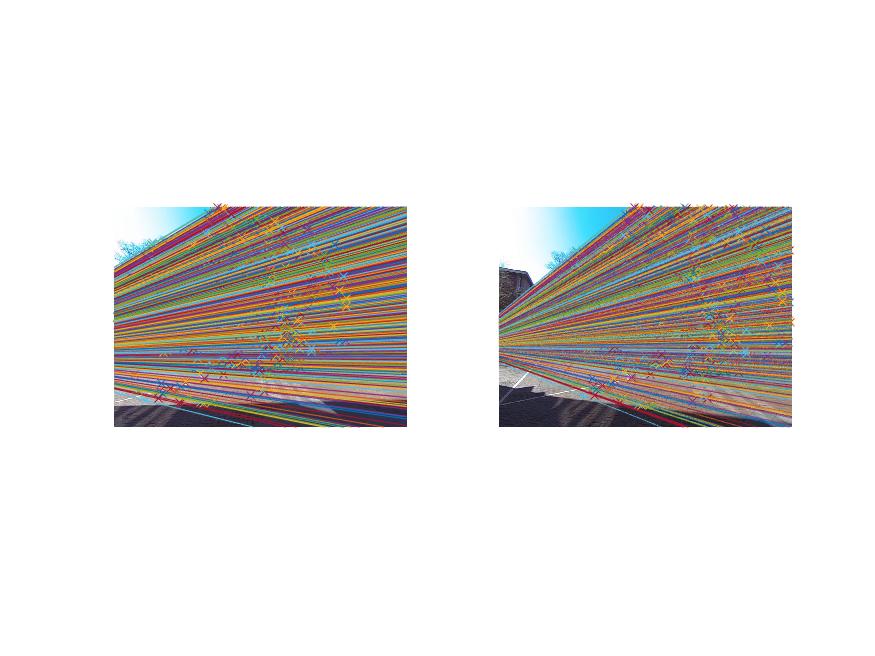
What we understand, from this is that very good estimates of R, C, E are made using the Swift correspondences and my code.



### Epipole Check,

It is also important to check the epipoles:

This was done with a Fundamental Matrix that has Rank 2, so its epipole lines cross at a point. Here is the image:



It is clear that the Epipoles are at ground level so they certainly are plausible. Prior to making the F matrix Rank 2, the errors are shown above, and so we expect that the actual errors after getting E that the pixel errors are still about 0.9 pixels.



The Linear Estimate verifies that the code from milestone1 and 2 works very well on this data and so we have good linear estimates of all the parameters. Having the images organized by folders and an easy routine to run, makes it possible to easily analyze all pairs.

## Milestone 3.1

The goal in this milestone is to use a nonlinear fit to improve the 3D locations using triangulation. The linear routine that we used earlier, triangulation in Milestone1, makes sure that the direction of the image point and the real point are on the same physical line and this does not minimize the actual pixel errors.

In this section we find the locations in physical space, X, that produce the minimum error when projected back to the two cameras. We keep the camera poses fixed and only minimize X. The function that is minimized takes the 3D points, X, and for each point produces 4 numbers, [u1 v1 u2 v2]’, the (u, v) pixel location in the two cameras, called the projected locations. We want X such that the difference between the projected and actual locations is minimized.

This is done in a routine called ‘NonlinearTriangulation’, that uses lsqnonlin. I defined a function called ‘triangulationFunction’ that takes the K matrix and the camera poses and importantly the X positions and produces a column vector ( 4 \* n x 1), where n is the number of 3D points included. The layout of the output is [u1 v1 u2 v2 u1 v1 u2 v2…]’ where 1 refers to camera 1 and 2 to camera 2. That is f and the b vector is the same format but has the actual pixel crossings in uv coordinates. lsqnonlin minimizes the length of the difference between f and b.

The main input is the 3D locations X. I keep X as a matrix, but it is important that if the input is reshaped into a 1D structure the order is consistent. I thus use X’ as the imput which has x is row 1, y is row 2, and z is row 3. It is 3 x n in size. This is crucial for below and also saves the need to transpose X for each function call.

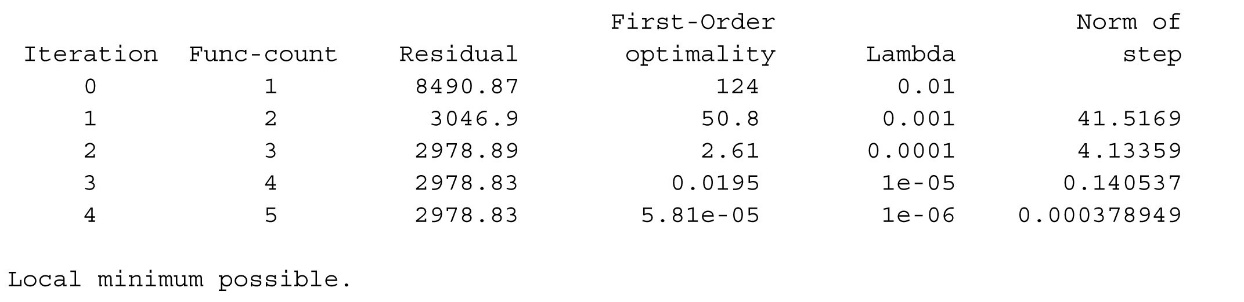
To make lsqnonlin work easier and better, I also supply the Jacobian in a function ‘triangulationJacobian’. This produces a matrix which is (4\*n , 3\* n) where the columns are (X1 Y1 Z1 X2 Y2 Z2 …), where 1, 2 refer to the different physical points in space, and X, Y, Z are the coordinates. By using X transpose as above, the order of the variables are (X1 Y1, Z1 …) and not (X1 X2 X3…), so when multiplied by the Jacobian the order is correct. One part of the normal equation multiplies dfdx by delta x, which is part of the Hessian. If the X is a matrix, presumably Matlab reshapes it and if the multiplication works, then the order of the reshaped X should be X1 Y1 Z1 X2 Y2 Z2. For this reason the input should be the transpose. The Jacobian then produces, given a delta x, how quickly each component (u1 ,v1 ) varies with a particular coordinate.

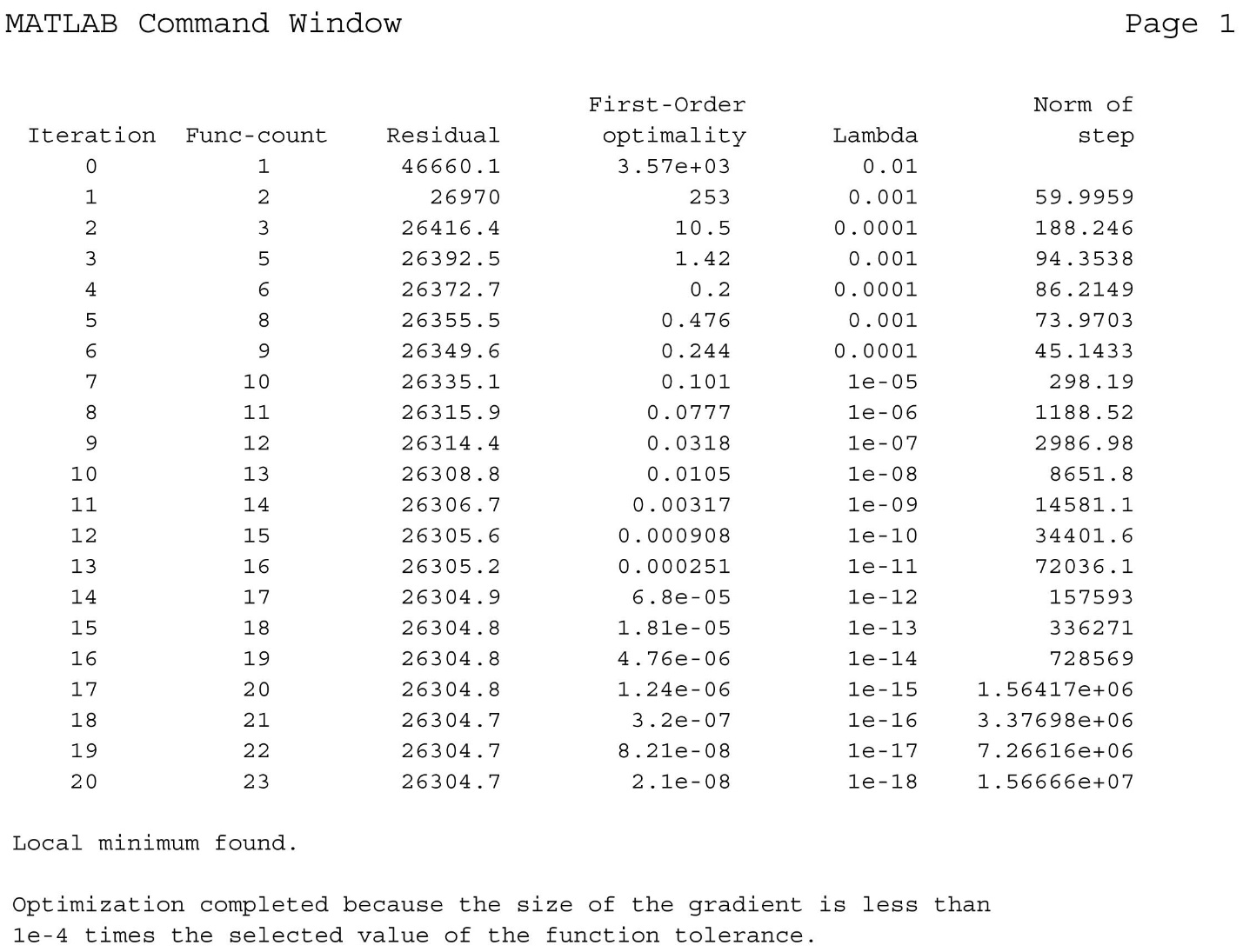
The Jacobian is block diagonal as follows:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 75.5 | 0.0 | -9.7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 75.5 | -12.9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 91.8 | 1.1 | -24.3 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1.6 | 85.5 | -14.4 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 57.0 | 0.0 | -12.2 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 57.0 | 24.8 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 69.8 | 0.8 | -21.8 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | -7.1 | 64.5 | 32.3 | 0.0 | 0.0 | 0.0 |

The rows are for u1 v1 u2 v2 and the columns are X1, Y1, Z1, X2, Y2, Z2. This is the Jacobian for pair 2, 3 at the best position point calculated from the linear fit. I added this Jacobian to the function definition. I ran it so I can see the iterations and after some 7 steps the routine converges to within 1 part in 10 ^ (-7).

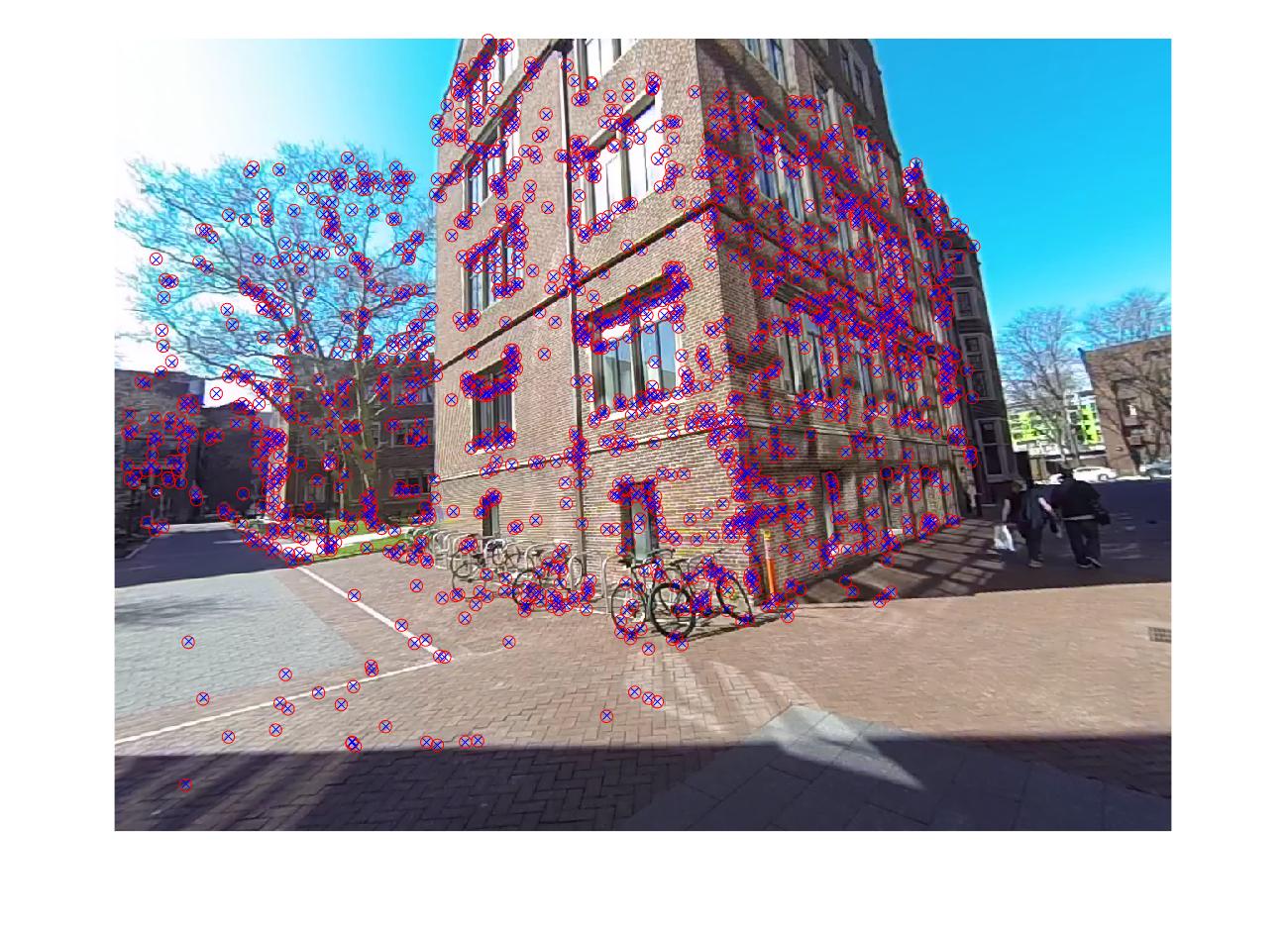
Here is the convergence of the fitting routine:





Second example from set 13, pictures 3 and 4. One can see that λ oscillates damping the choices a lot, but in the end gets small and so big steps are taken. The algorithm converges.

Here are the results of fitting. The red circles are the original correspondences and the blue crosses are the reprojections after nonlinear fitting. This is the correspondence between pair 2 and 3.



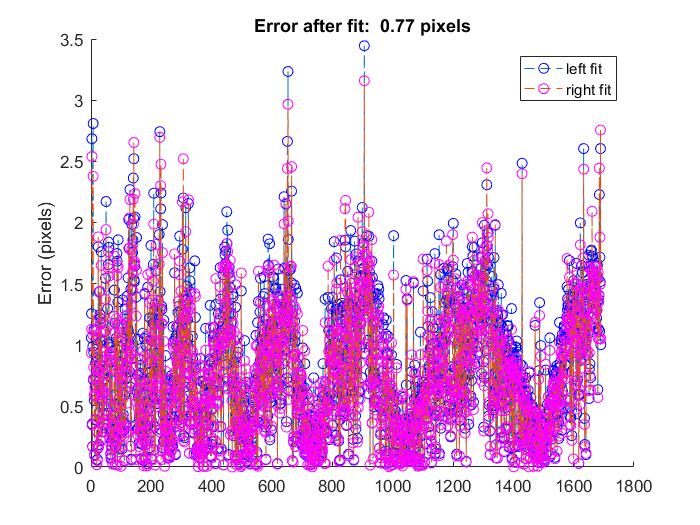
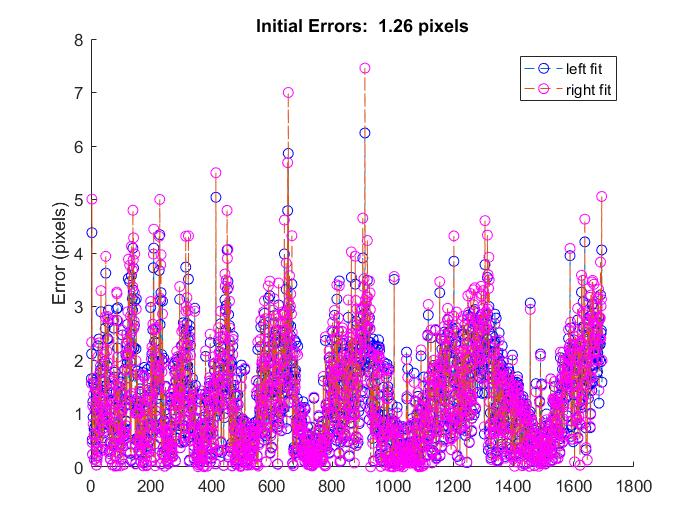
Left image: Image 2

### 



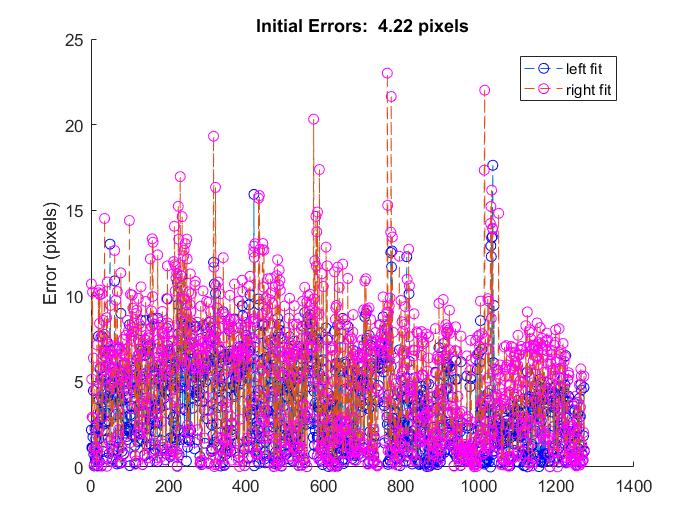
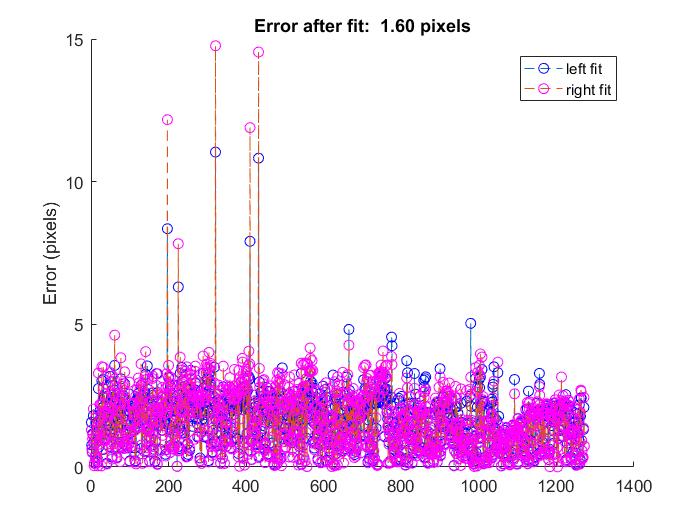
Right image, Image 3.

One can see also the errors in pixel:



Here you can see that the average error by using the nonlinear fit gets reduced nearly in half. That is clear from the scale of the errors. This verifies that the linear based errors were not optimal and even with the same pose, one could find a better solution.

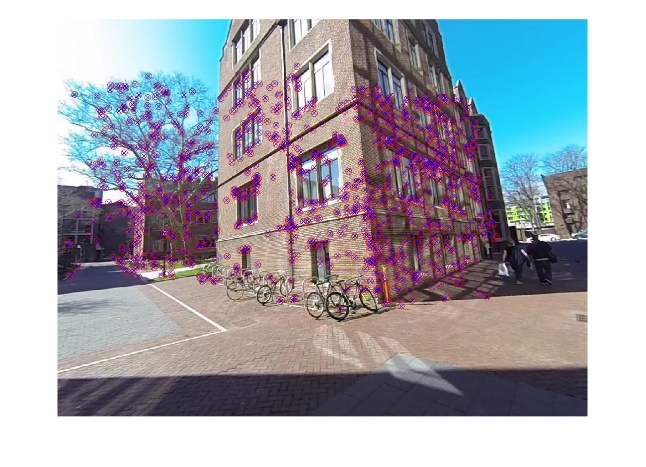
Here is another example of the reduction in the error this one due to the pair 1-2

This fit took 20 iterations and improved but did not fully converge. You can see that many of the bad points got removed with nonlinear adjustments and the overall error per point improved by a factor of 4.

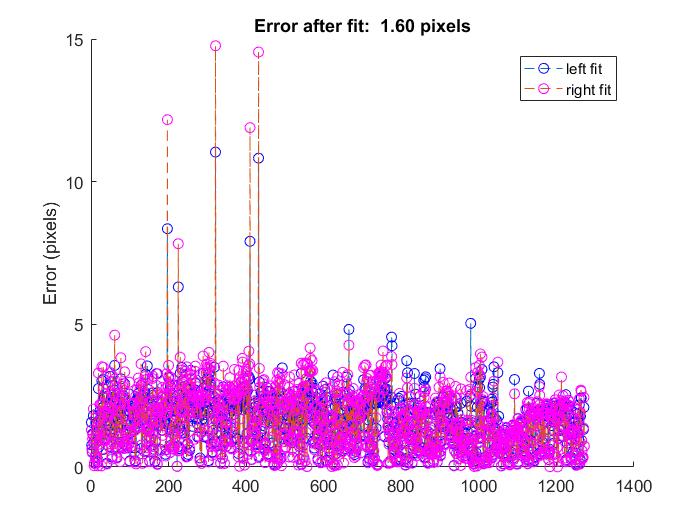
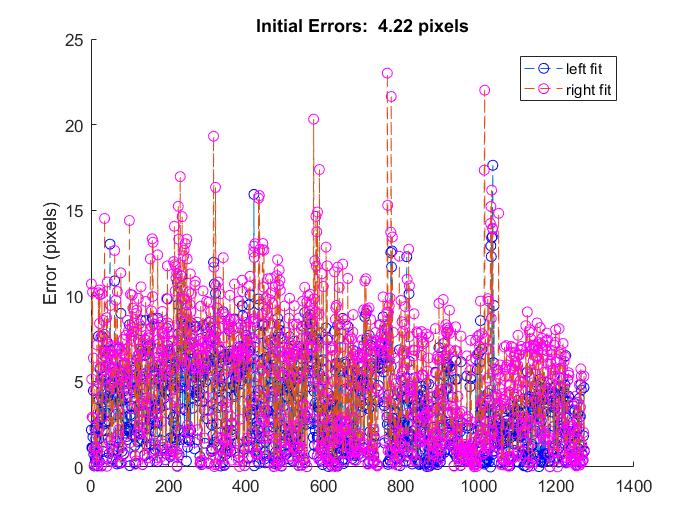
Here is one of the figures. Left is figure 1 right is figure 2. There are locations in the circled places where the projections do not align with the points.





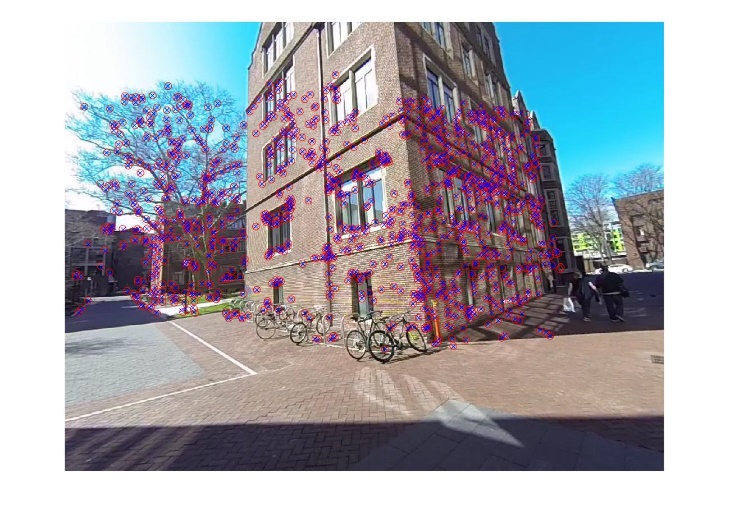


One More Example: Set 10 with Pair 3 -4:



Here there seems to be many errors of 20 pixels but after 20 iterations a solution was found that reduced those errors dramatically so that only a few pixels were off. The non linear routine finds a minimum effectively.

Here are the aligned points:



Each iteration took a few minutes of computation. I also tried doing the fit without the gradient and the fit gave up after a few minutes. Numerically it is difficult to estimate the gradient from the function because the Jacobian is sparse. Adding the gradient is definitely a good idea.

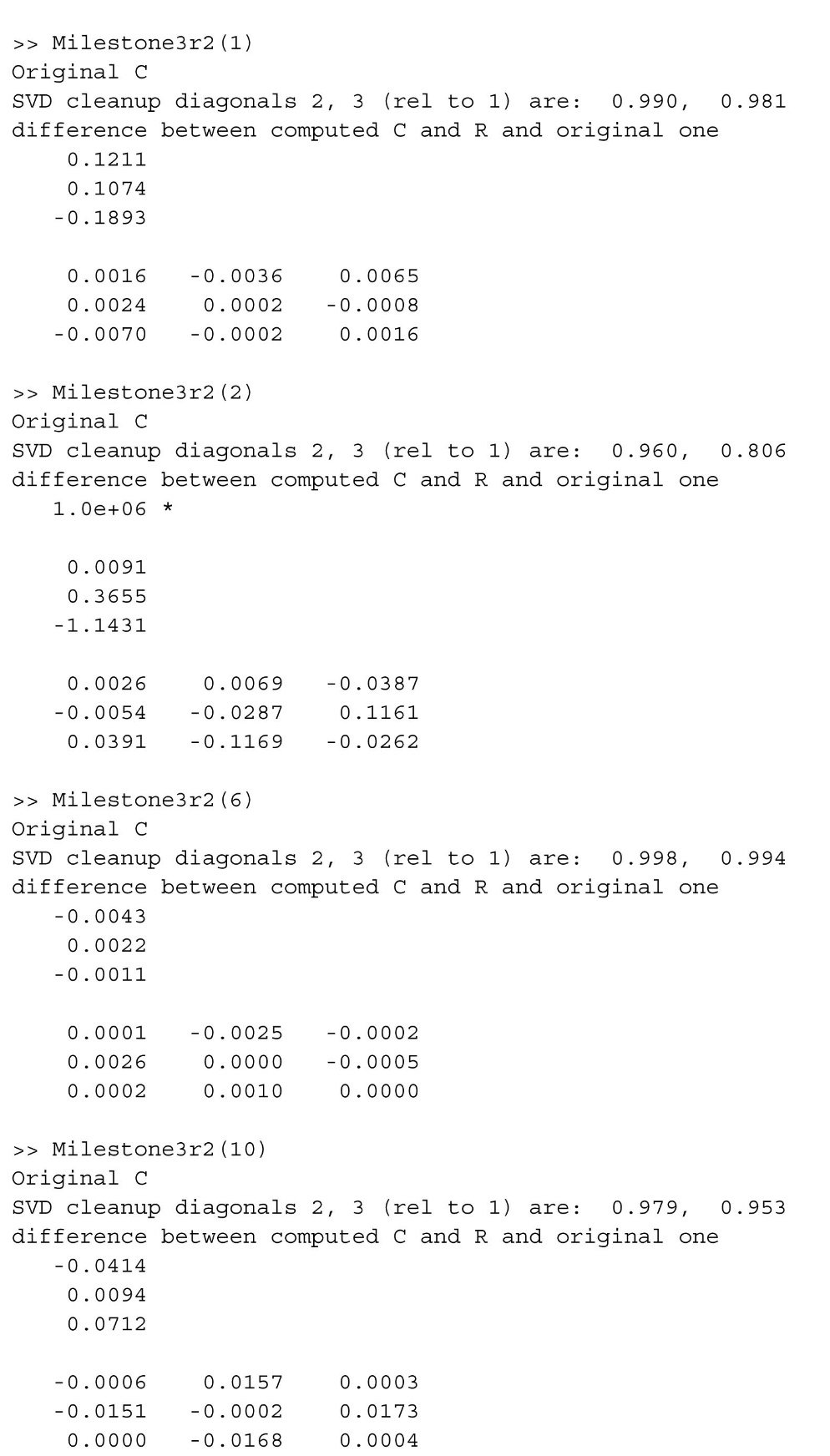
## MileStone 3r2

I wrote LinearPnP.m (again in the Milestone3 directory). I first used all the data to compute P, like the Homography, the relationship between X and x. I then used the fact that R is orthogonal and a rotation, so its singular values should all be the same. We have:

Where γ is the factor that scales P, K is the calibration matrix, R the rotation and t is the translation (Bob’s location from Alice’s perspective). The rotation is set by the first three columns of P and t is set by the last column. Using singular value decomposition on one can get the diagonals. The diagonal element, say the biggest one sets γ and that is used to set the scale of t. I added code to make sure det (R) is positive and in all cases γR was .

I ran this on many of the sets that I analysed and checked if the answers are reasonable.

Here is the output:



Set #2 had some poor correspondences, but the rest show reasonable answers. I checked the diagonals of the SVD on the rotation matrix to see how close they were to 1 and it turned out in many of the good sets (1, 10, 6, 7, 3) they were very close.

## Milestone 3r3 PnP RANSAC

I wrote the PnP RANSAC routine using the above linear PnP. The goal was to identify outliers and refine the best R, C of the second camera. We keep x and X and K all fixed. With these fixed, the camera 1 errors are not going to change at all, so this step relies only on the correspondences with Camera 2. The starting routine is “milestone3r3.m” and that takes parameters, the first is required, the rest are optional.

In doing the ransac, I used my striated Samples routine because that sampled around the image and so was less prone to fitting to a small region.

1 . the set Number (1 to 15),

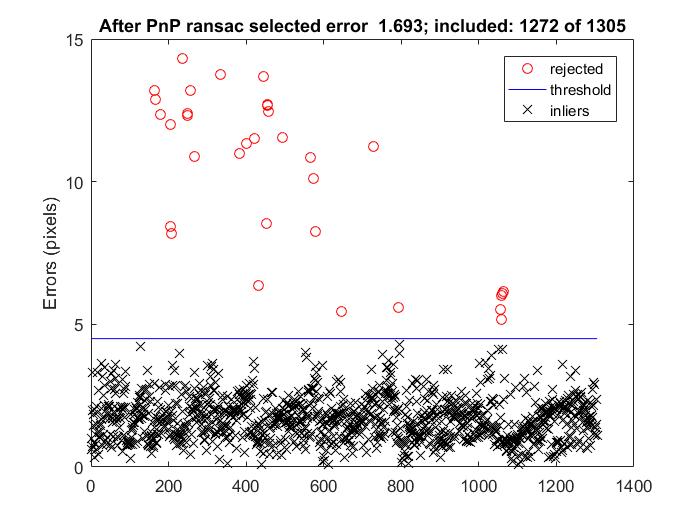
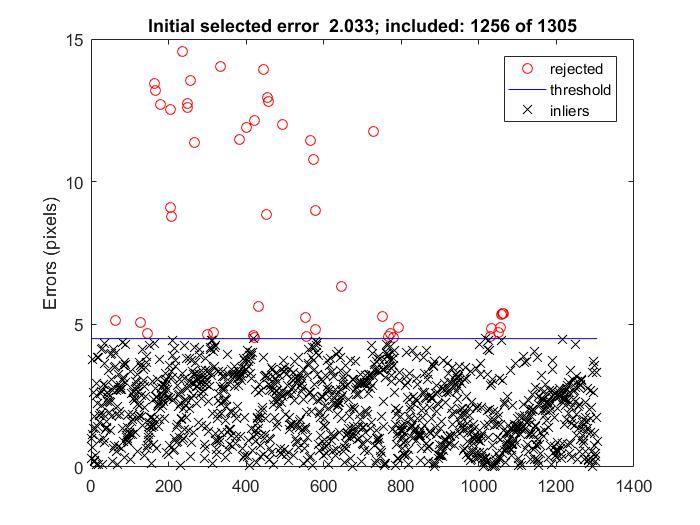
2. The threshold in pixels to exclude data,

3. the likelihood of success # inliers/total

4 the probability of ransac success (should be about .98)

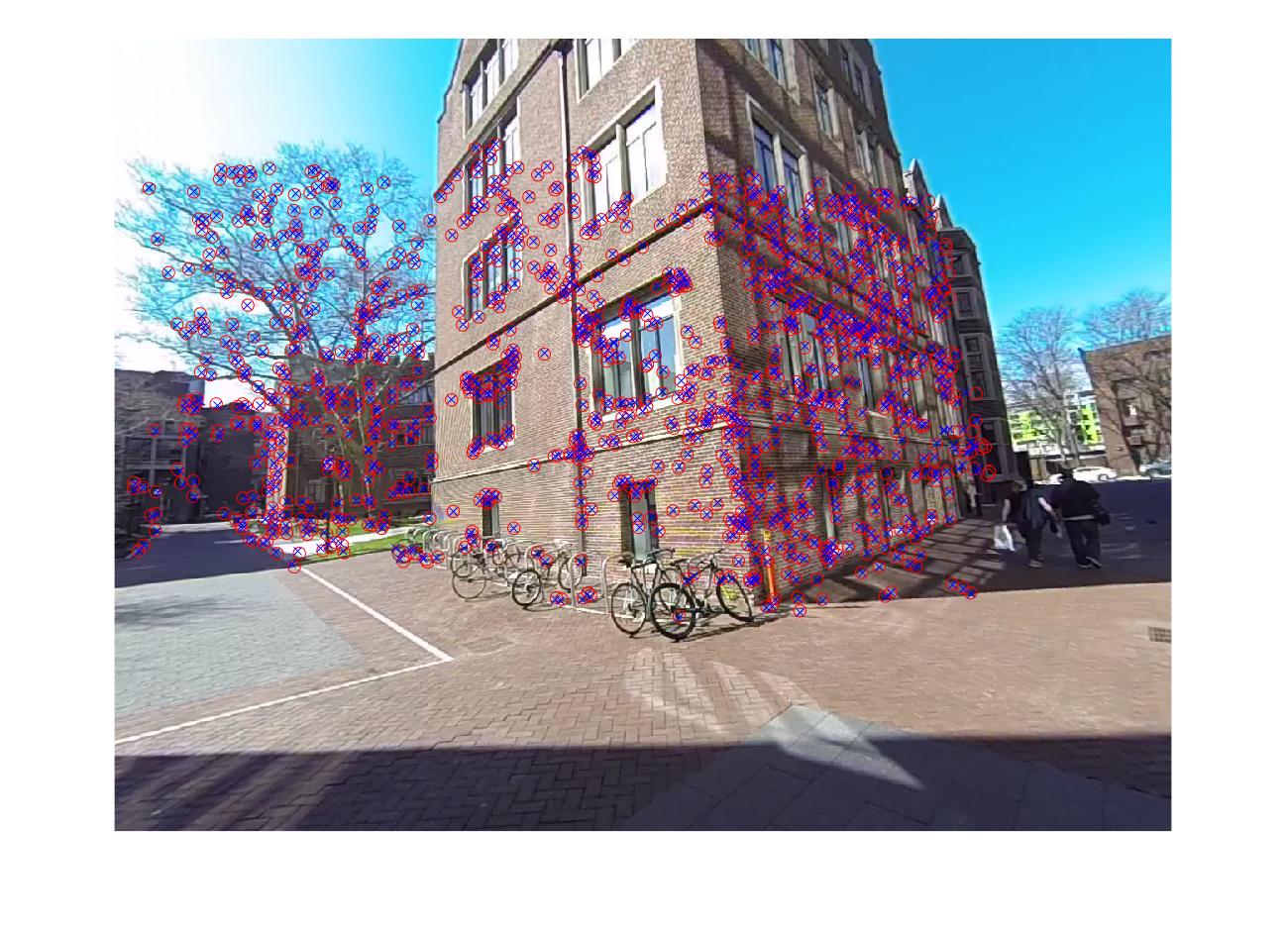
The first step was to calculate the number of outliers and given the above parameters how well would the current RC do? I made error plots and also a correspondence plot that shows the improved correspondences after PnP Ransac.

Set1:



PnP Ransac randomly picks 6 points, then determines the best R, C using LinearPnP, and then sees how many of the existing correspondences fit within the error threshold. With the initial R,C actually more points fell above the threshold, and so more were outliers. The RANSAC RC choice produced a better fit in two ways: it included more points and those included actually fit better.

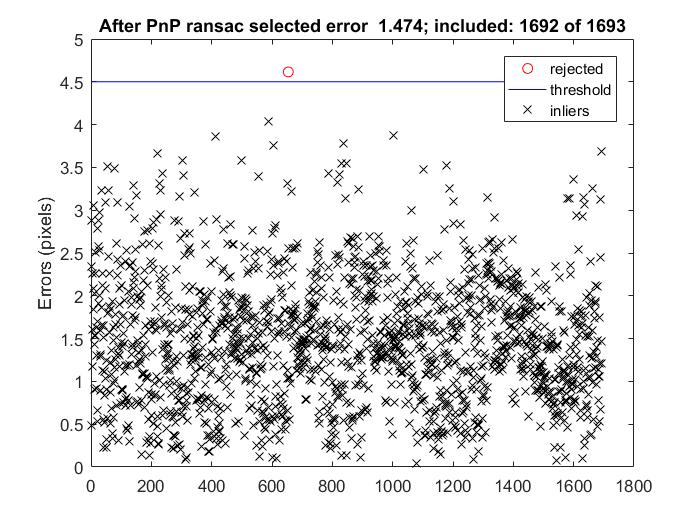
One can also see the results by looking at the improved correspondences:





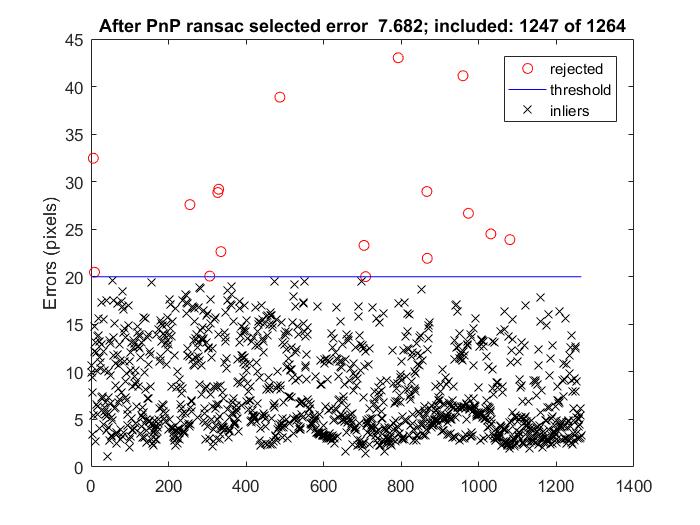
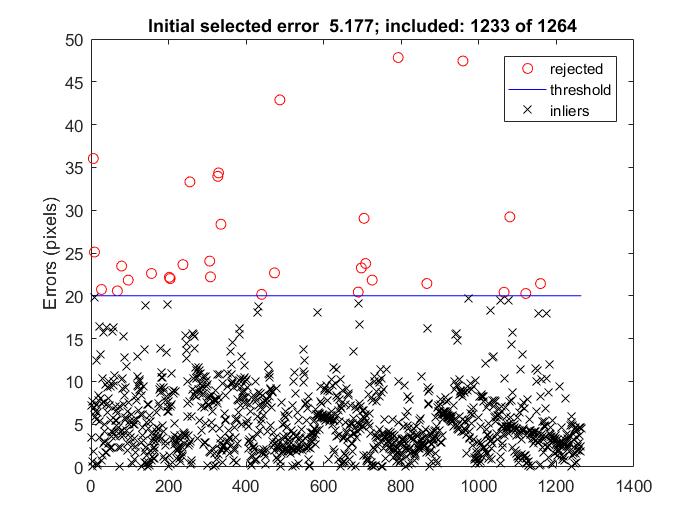
Here, the circled points show locations where the original points produce large errors of reprojection. Those are the outliers.

Another good set is Set # 6 (2 and 3)

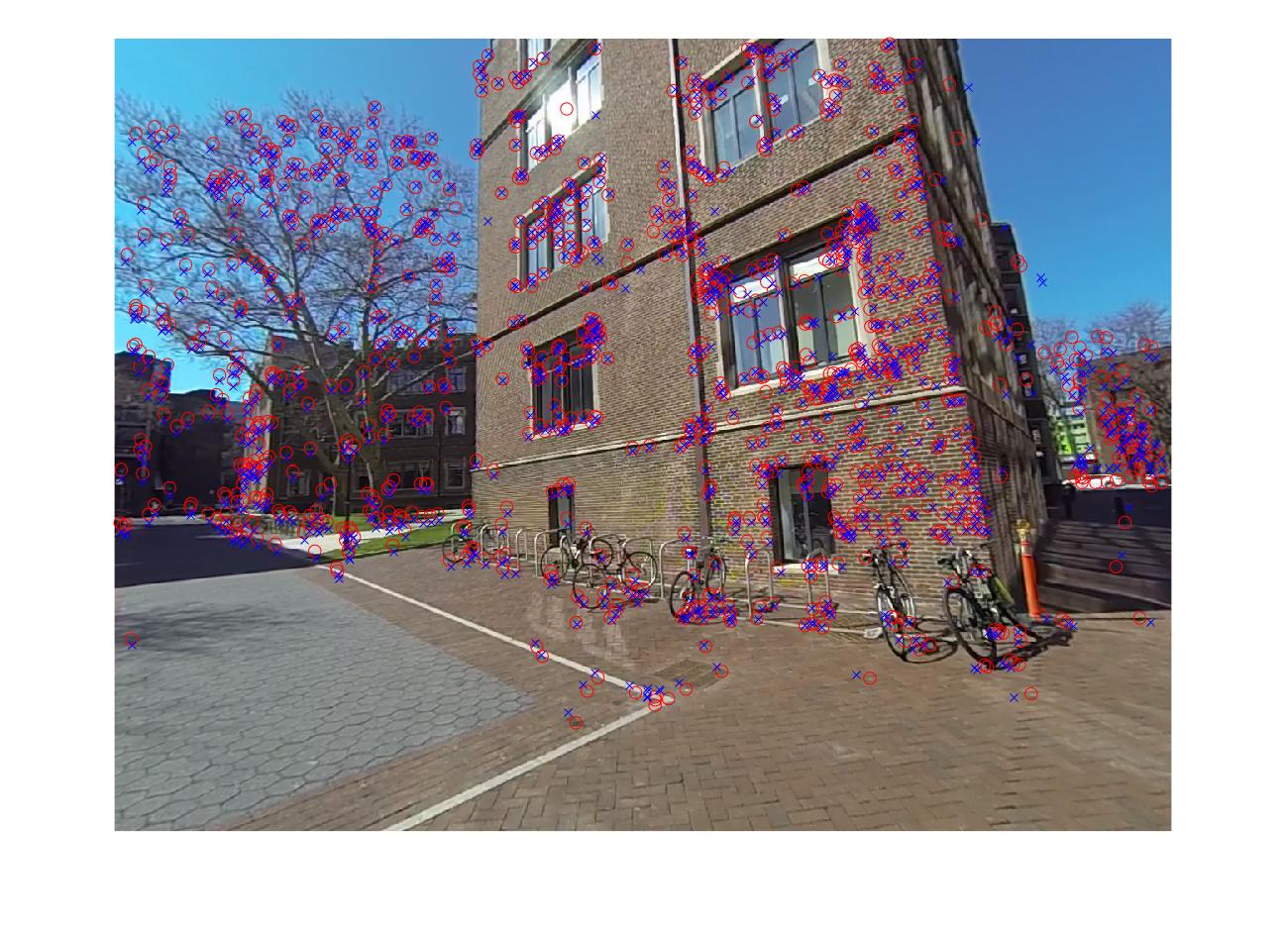


Here it identified one point that made the results worse and so the fit improved.

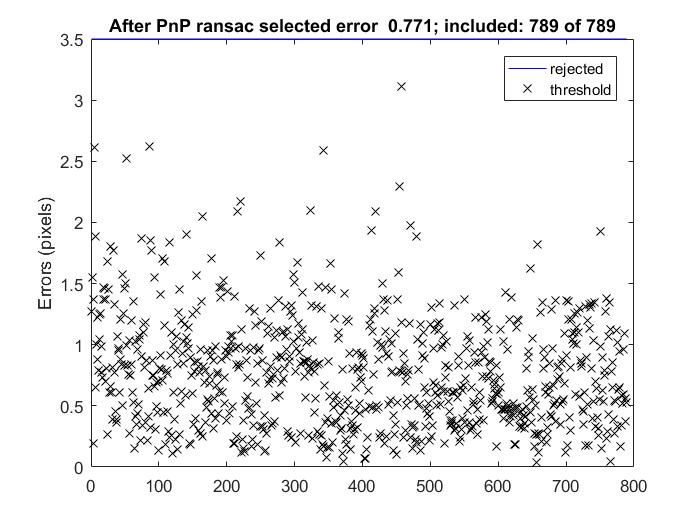
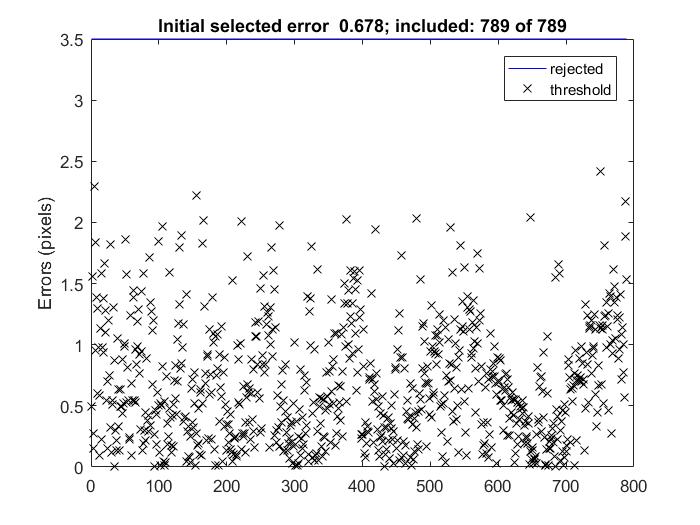
Another example is set of images 5 and 6:



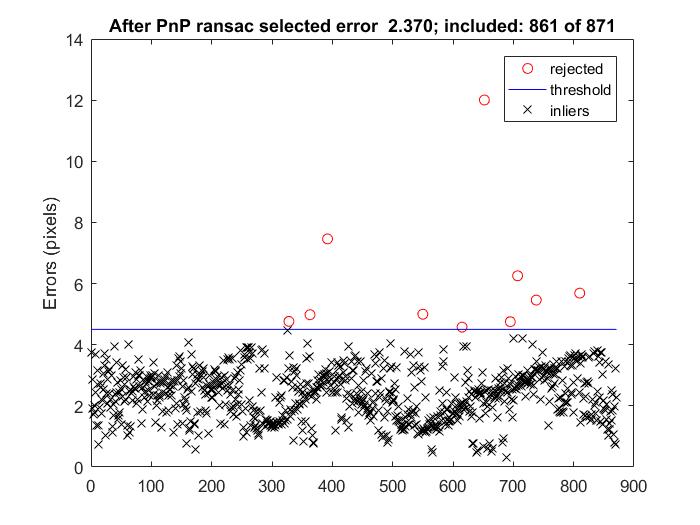
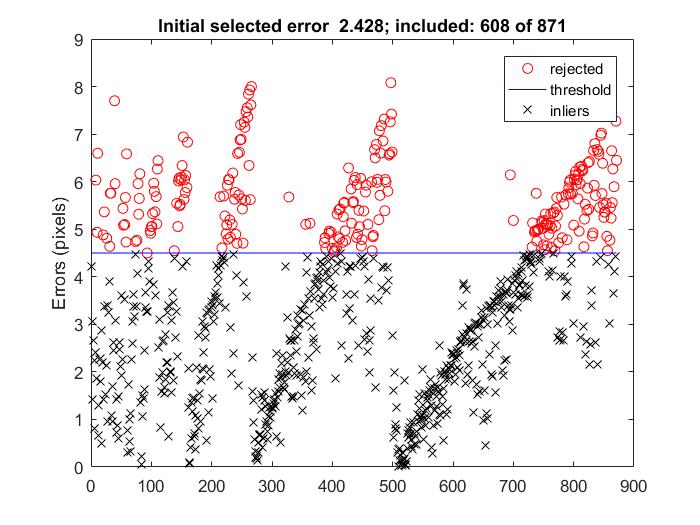
More points were included below the threshold after Ransac.

Here is the reprojection image. Not all points align here but still PnP Ransac helped importantly by identifyn

In some image sets such as 7 the Linear choice of R and C did not actually improve the fit. Those cases there are no outliers, so the R and C calculated using the entire data set actually produced a better fit.



Here is another Example of the dramatic success of PnP Ransac (below from images 4 and 6, set 14). With the first image with the RC selected by the fundamental matrix, it looks like the correspondences are off and that there is no hope. By selecting a better RC, suddenly all or nearly all the points fit well, and there are only a handful of outliers that were removed.



Even in these cases the PnP Ransac R, C is not much worse that that made using all the data. PnP Ransac has the advantage of removing outliers as well.

In all cases I used the Ransac selected R and C.

## MileStone 3.4 NonLinearPnP.

Here the objective was to improve the C and R estimates with a nonlinear fit. Prior to this we have a Ransac generated C and R and here we try search to see if there is a better solution.

NonlinearPnP that I wrote again uses two functions a function evaluator: given an input R, C calculate the best values that minimize reprojection errors. Here x and X are fixed. Also the camera 1 is taken as a reference and so R and C depend only on K, and the 3D points and the image points.

R normally has 9 parameters, but of all those rotations, many produce scaling or skewing. We want a lower dimensional space. I represented the rotation as a quaternion; given a quaternion and a C position, I calculated the pixel error. We initially have a Rotation matrix, so in order to start the fit one has to convert the Rotation estimate to a quaternion. I wrote two functions :quaternionTo a Rotation and a rotation to a quaternion. I tested them during PnP linear with many different random R matrices, and each time I could convert into a quaternion and back. My Rotation to quaternion routine finds the maximal component and uses that to complete the conversion.

I also got the gradient to work. This was hard to figure out. The Jacobian was 2 \* n x 7, where 3 are positions and 4 rotations. This conversion was done in two steps: first I converted to a rotation matrix and then to a quaternion. This is written in triangulationJacobianQ.m

The result was very fast and accurate conversion. In 4 iterations and a few seconds it converged.

First-Order Norm of

Iteration Func-count Residual optimality Lambda step

0 1 889.678 4.04e+05 0.01

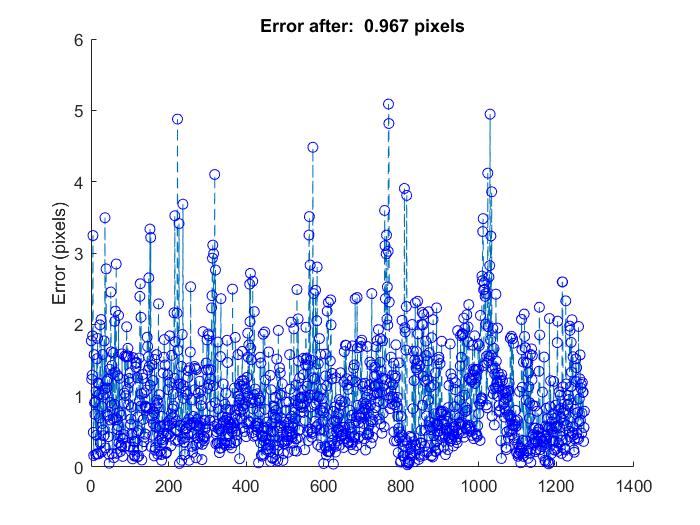
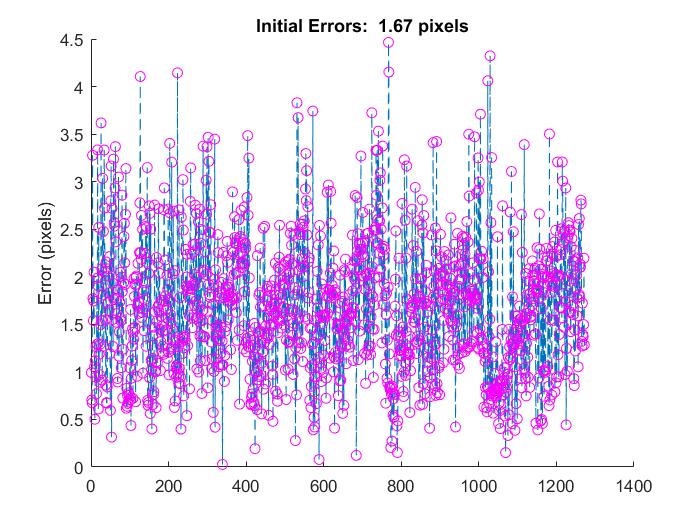
1 2 584.041 976 0.001 0.0102239

2 3 584.04 0.13 0.0001 1.14416e-05

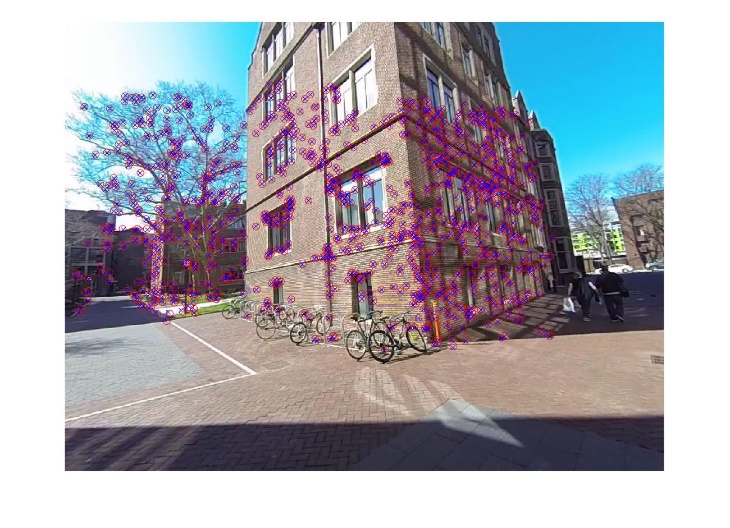
3 4 584.04 6.56e-05 1e-05 1.21847e-07

Local minimum possible.

For example Set 1:



And here are the correspondences:



Here is another Example from Set 15 (5 and 6):

## 

This Nonlinear fit is very fast and robust. In every set analysed (1, 2, 3, 6 7, 10, 11, and 15) NonlinearPnP improved the fit by about a factor of 2.

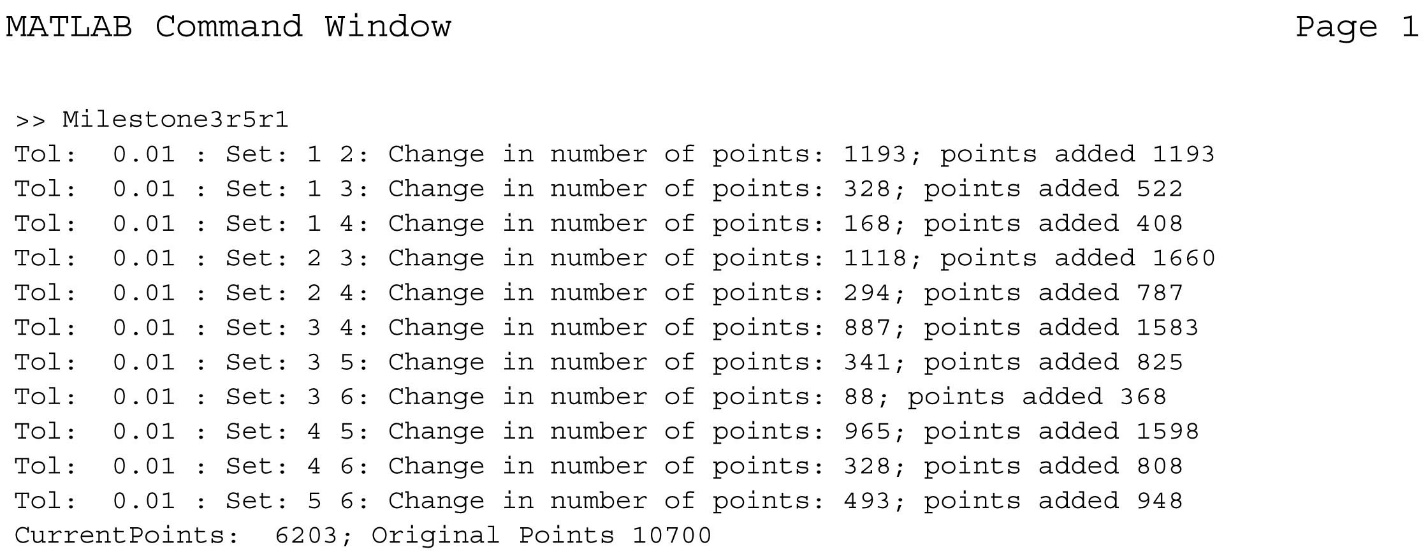
The images of all these sets are in their respective folders.

## Milestone 3.5.1 Adding the buildVisibility Matrix

I wrote Milestone 3r5r1 which builds the Visibility Matrix. This was tricky because it required that one decide how close two points are in projection in order for them to be considered equal. For the longest while, I understood that we were supposed to use our best estimate of the reprojection, based on our calculated positions and R, C. That is problematic because that would depend on how well the parameters are determined to begin with. I worked on improving that for a long time and settled with using that or the original unmodified reprojection errors. Now, after working on the bundle adjustment, I realize that only the swift based errors are needed and those are more accurate to begin with. Two points can be considered equal if the distance between them is of order of the accuracy of the swift point. Now that is set to 0.01.

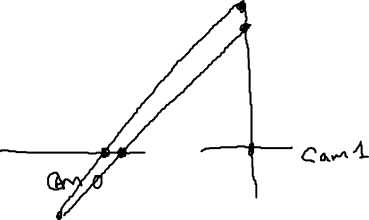
It can be justified as follows: If Two points align by swift, then there is a neighborhood of data that support that conclusion. That fact should be used.

Here is the output:



This shows how many points were in each set (last number in each row) and how many points were added to the total number of points (2nd to last number). In all the sets there is substantial overlap so one can benefit from the bundle adjustment.

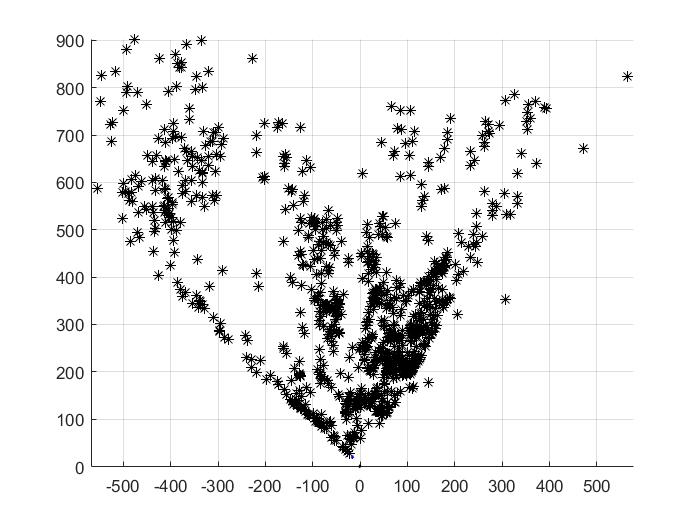
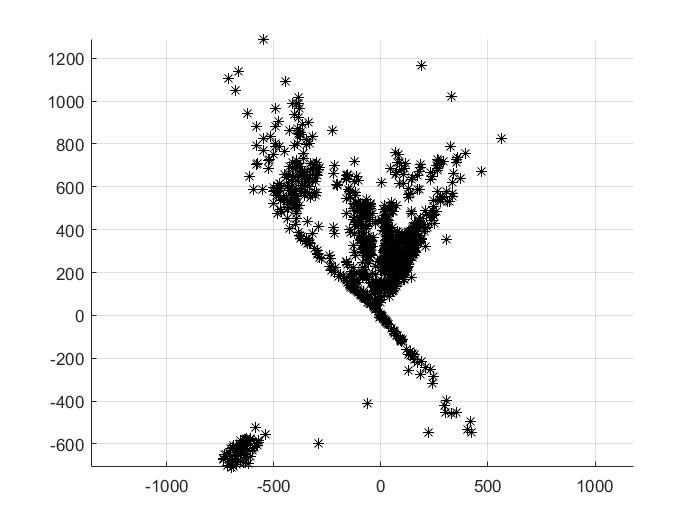
If In one view, two points may project to the same location but not in another view. This is demonstrated in the drawing below. The data has examples as one can see in set 1\_2 in the first few rows. The same point on image 2 maps to two different points on image 1. To handle this case I wrote that if an overlap is found with one camera and not the other and the slot was filled before, then it should be a new point.



One might think that if a point in image 1 corresponds to one in 2, and that 2nd point corresponds to a point in image 3, then there should be a correspondence between 1 and 3. I wrote code in Build VisibilityMatrix that would make a new point unless each of the three pairs of points correspond. When I ran this, I found very few overlaps. The total number of points was 9100, for the set shown above. This doesn’t work because for a pair to register as corresponding, it must match uniquely, and that may not occur in all pairs, even if the point is physically visible in all images. 3 and 2 may correspond, 1 and 3 may correspond but that doesn’t mean that in image pair 1 and 2 will correspond. I took this criteria out to produce the above result.

## Milestone Cheirality

If points do not obey the Cheirality condition, then those points should be disregarded. All correct 3D points should be in front of the camera. I wrote MilestoneCheirality to plot the error and pose before and after removing the non cheiral points. Below one can see the two cameras and all the nonchieral points were removed. In many cases there were no non Cheiral points and in other cases, there were some 10.





Left and right images. Red circles are the original points. Blue x are the points that were in front of the camera.

I also checked the pixel error and there was no change before and after non cheiral points were removed.

## Selecting the Data

Now that all the single files have been written, the data is selected for the bundle adjustment. The processing was done in the order of the milestones.

* LinearEstimate.m to get the linear estimate af all the parameter
* Milestone3r1 to get the positions using Nonlinear Least squares
* Mileston3r3 to use Ransac and eliminate some outliers
* Milestone3r4 to do nonlinear Least squares on the pose parameters using Quaternions.
* MilestoneCheirality to eliminate points behind the camera.

After one cycle the cycle could be repeated. I never reran LinearEstimate after the first round because that would overwrite the fit parameters.

The slowest step was Milestone3r1, because the fit was over many 100s of variables and often took 5 - 10 minutes. Nonlinear least squares with a gradient on the pose was extremely fast taking seconds at most.

In difficult cases, I performed a few cycles. With the set 5-6 a few cycles got the average error per point from near 10 pixels to about 1.7 pixels. Here are the final errors per set. These numbers can be easily seen by running MilestoneCheirality (1 to 15).

|  |  |  |
| --- | --- | --- |
| Pair of Poses (camera numbers) | Average Error (pixels) | Valid Correspondences |
| 1, 2 | 0.99 | 1193 |
| 1, 3 | 1.36 | 522 |
| 1, 4 | 1.13 | 408 |
| 2, 3 | 1.12 | 1660 |
| 2, 4 | 0.64 | 787 |
| 3, 4 | 0.33 | 1583 |
| 3, 5 | 0.69 | 825 |
| 3, 6 | 1.04 | 368 |
| 4, 5 | 0.65 | 1598 |
| 4, 6 | 0.76 | 808 |
| 5, 6 | 1.15 | 948 |

I reran sets 1, 3 to lower the average pixel error from 2.4 to 1.7. I started again with LinearEstimate and this time set the threshold to be relatively high, so only a few points were removed each iteration. In each cycle, I optimized the 3D positions with Milestone3r1, used Ransac to select the best RC and then optimized that. I picked a starting configuration that produced few non-Cheiral points. Now we have a good data set and all many overlaps. The trace of the overlaps is above.

## Milestone 3.5.2

## Bundle Adjustment Milestone 3.5.2

### Establishing a Global Coordinate System

I went forward and used the Visibility Matrix to prepare the data in order to complete the bundle adjustment non-linear fitting. Each pair of images produces 3D positions for the projected positions. In each case the 3D positions are relative to the first camera in the set. In order to perform a Bundle adjustment, the 3D positions need to be relative to a global coordinate system. To get the projection on camera i, we need to transform coordinates from 0 to i. so that upon applying the K matrix, the projections are correct. This is done in the routine (.m file, Milestone3)

BuildPose(C, R). Transforms into a global coordinate system. The bundle Adjustment then takes global coordinates and these transformation matrices [I C R 0], which represents the transform from 0 to I to measure coordinates in cam I’s frame, and then uses the K matrix to turn this into a projection.

C, R as input is arbitrary and sets

[ 1 C R 0] the transform from 0 frame to camera 1. It can be used for example, to do bundle adjusments in sections say 10 cameras at a time. Each fit would get the appropriate C R so that all the positions found refer to the same frame. Build Pose then takes relative C, R’s, like from cam3 and camera 5 and turns them into [3 C R 0] and [5 C R 0].

X = BundlePositions(Cset, Rset, traj, V) This routine takes those universal pose positions and finds the X coordinates of each 3D point in the global coordinate system (0). For example, the set cam3 and cam5 produces a set of 3D positions, but those are relative to camera 3’s coordinates. Here those local coordinates are transformed using [0 C R I] into universal (frame 0) coordinates. Also, 3D points may appear in several different camera pairs. In these cases the first pair that has this 3D coordinate is used to transform the point into the world frame.

The steps used here are:

### b = BundleOutput(traj, V). b (out x 1)

This takes the trajectory and the visibility matrix and produces one u, v pair for each projection. These are the actual measured projections not estimates of them. The ordering is given by the Visibility matrix. If the first column of V states that 3D point 1 is visible in cameras 2 and 4, then b will have (u2 v2 u4 v4) and it will be a column vector. The length of b is set by the number of points and the number of frames in which each 3D point is visible. It is given by sum(sum(V)).

% initialVector start

### startP = BundleInitialVector(Cset, Rset, Xset) (m x 1)

This produces the P0 the start point. This consists of the initial vector used to start the nonlinear fit. It consists of [cam2, cam3, …cam6, 3DX1, 3DX2… 3DXN]. Each camera has 7 parameters, the first 3 are the C vectors and the last 4 are the quaternion rotation parameters. For example, cam2 encodes the transformation [2 C R 0], transforming from the universal frame to camera 2’s frame. The quaternion encoded is for this rotation R, always from 0 to cam I. Similarly the 3D positions are with respect to the universal coordinate frame 0. In the Matlab version of this function, I dropped cam1 from the input list because its parameters are not varied.

### function inVector = BundleInitialVectorSBA(Cset, Rset, X)

This function is nearly identical to above but now the input vector is [cam1, cam2… cam6, 3DX1 …3DXN], where camera 1 is added. In the sparse bundle adjustment cam1 can be set to remain constant.

### function [outVector, dfdx ]= BundleTriangleFunction(C, R, inVector, K, V, b)

outVector: (out X 1)

dfdx : (out X m)

This function calculates the error incurred when using inVector and also the Jacobian. C, R is the pose of the first camera (which is not supplied in inVector and is constant), K is the calibration matrix and V is the Visibility matrix. b is the measured output. This function is designed to work with Matlab’s Nonlinear fit that requires a function with two outputs, the error and the Jacobian for each input vector. The main ideas for calculating this are

To get the outVector:

(u, v, w) = K \* R \* [X3D – C];

outVectoru, outVectorv = u/w, v/w – b;

in other words project the 3D point using R, C and then find the error.

The Jacobian is more complex. One needs the derivative of u,v with respect to the 3D positions and the 7 pose parameters. One also needs to put this block in the correct place in the giant matrix.

To calculate the Jacobian one can break it down into steps

dM =d(u/w, v/w)/ d(u, v, w) (2 x 3) Jacobian giving the change in u/w, v/w given the change in u, v, w.

d(u,v,w) /d(X3D) = K \* R; the change in u,v, and w given a change in 3D position.

D(u/w, v/w)/ D(X3D) = dM \* K \* R;

Similarly

D(u/w, v/w) /D (Cxyz) = - dM \* K\* R; the change with respect to the camera location.

To get the change due to varying the quaternion use

D(u,v, w) /D(R1…R9) = Rot; given in the code. R1..R9 are the rotation values

D(R1..R9)/Dq: calculated in quaternionJacobian(q) (matlab file).

This gives D(u/w, v/w)/Dq = dM \* Rot \* D(R1..R9)/Dq

### function m = projection(j, i, rt, xyz, r0, a)

This calculates performs the same projection calculation as BundleTriangleFunction and is used in the sparse bundle adjustment in C. The inputs are rotation and C (rt), and the 3D positon, and the non zero components of the calibration matrix.

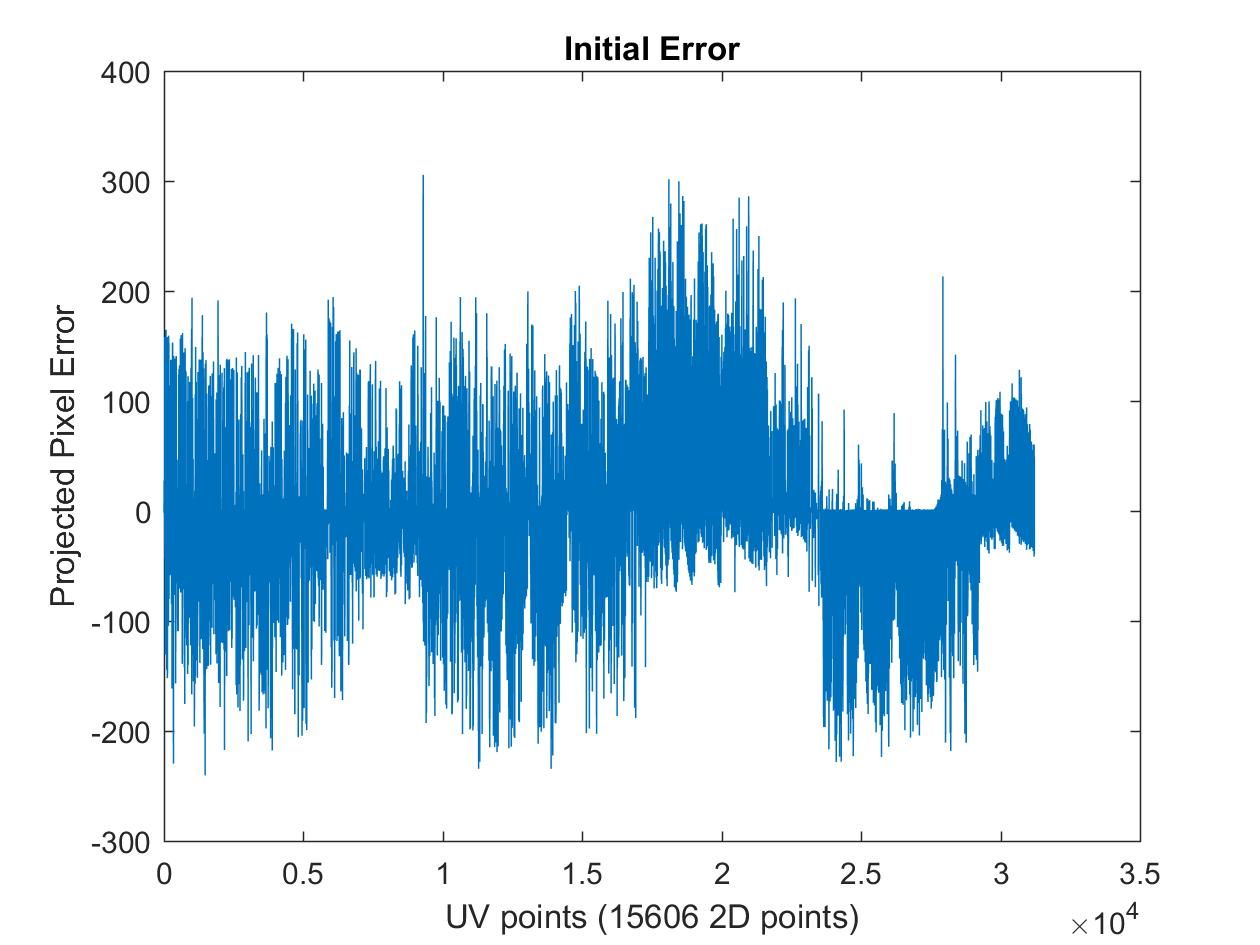
### function [jrt, jst] = projac(~, ~, rt, xyz, ~, a)

This produces a Jacobian in Matlab given a 3D point and the rt (really rotation and C). above the jacobian also put the nonzero elements in the correct location in the matrix, but here project just calculates the Jacobian with respect to R, C (jrt) and with respect to position (jst). This is the calling convention in the MATLAB example files in sba. By looking at the C code and the m code jrt would normally be 2 by 7. The example code shows that the jrt is a 1D array and it is ordered du/dq dvdq, ordered so the first elements are u and the last elements correspond to the next row in the jacobian. This is the Jacobian with respect to rt, xyz, and does not build the entire Jacobian Matrix.

The Sparse Bundle Adjustment code (SBA) does not build the entire Jacobian at once. Since the Structure portion of JT \* J is block diagonal the blocks are inverted, and then the Schurs compliment of this is used to calculate the new pose coordinates here a smallish 42 by 42 matrix.

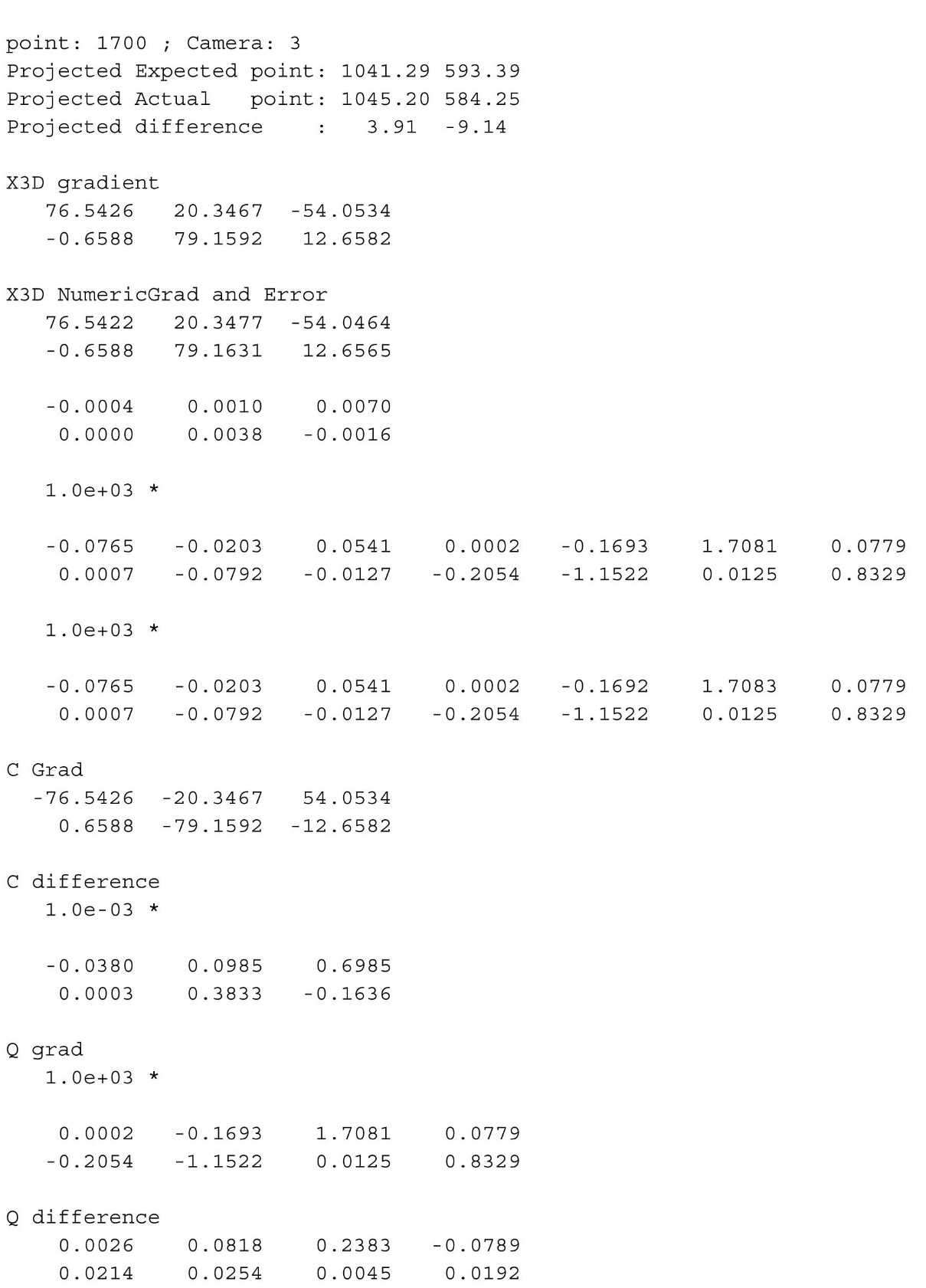
### Testing the function and the derivative

To demonstrate that the function produces the correct position I graphed measured projection with that predicted by the pose and the 3D positions using the initial estimate of the 3D positions:



This plot demonstrates that the projected pixel error is generally low initially, near 20, and so the projection is calculating correctly.

Also I wrote TestBundleTriangleSBAOnePoint(Cset, Rset, X, K, traj, V) which tests the gradient and position of any point (here point 1700). It uses projac to calculate the gradient and projection to calculate the position. Other TestBundleTriangle versions also test BundleTriangleFunction to see if the elements of the Jacobian that should be zero are zero. Here is an example output:



### [Cset, Rset, Xset] = unBundleVectorSBA( p, V);

Takes the visibility matrix and the returned position p and returns Cset, Rset, Xset. Another version of this works for the Matlab nonlinear fit.

### [Cset, Rset, XC] = BundleAdjustment(Cset, Rset, X, K, traj, V)

This routine performs the Matlab based Nonlinear fit. It uses all the functions above to perform the fit.

### [Cset, Rset, XC, info] = BundleAdjustmentSBA(Cset, Rset, X, K, traj, V);

Performs the bundle adjustment no using the sparse bundle adjustment package. It passes projac and projection for the function calls.

### traj = updateTrajectory(Cset, Rset, XC, traj);

### updateStartDataFile(Cset, Rset, traj, 'StartData.mat', 'StartData.mat');

These take the returned Cset Rset XC and trajectory and update the trajectory with the new 3D points. Those in turn are used in update StartDataFile and that updates the pair values for each pair, update the 3D positions and stores the new Cset, Rset so that they can be compared with the ones based on the pair only.

## Getting the Demo to Compile:

I figured that the best way to test if sba is working properly on my system was to run the demos and get them to produce the correct output. That way you know that Levenberg Marquardt is working and the libraries are all installed.

Here is a portion of the successful outputs in release mode:

C:\Users\anton\CloudStation\Drive\PENN\cis580\Project2\sba-1.6\demo\demo\x64\Release>

demo 7cams.txt 7pts.txt calib.txt

Starting BA with fixed intrinsic parameters

SBA using 465 3D pts, 7 frames and 1916 image projections, 1437 variables

C:\Users\anton\CloudStation\Drive\PENN\cis580\Project2\sba-1.6\demo\demo\x64\Release>demo 54camsvarKD.txt 54pts.txt

Starting BA with varying intrinsic parameters & distortion

SBA using 5207 3D pts, 54 frames and 24609 image projections, 16485 variables

……

Method BA\_MOTSTRUCT, expert driver, analytic Jacobian, without covariances, variable distortion (3 fixed), variable intrinsics (2 fixed)

SBA returned 36 in 36 iter, reason 4, error 0.128779 [initial 2.14707], 37/36 func/fjac evals, 36 lin. systems

Elapsed time: 3.21 seconds, 3214.00 msecs

C:\Users\anton\CloudStation\Drive\PENN\cis580\Project2\sba-1.6\demo\demo\x64\Release>

This output can be compared with the README.txt in sba/demo.



Here is the interpretation of the error codes and the info array returned from the fit: (2) stopped by small dp or small change in input parameters, 3D position or pose. (3) stopped by reaching max iterations; (4) stopped by small relative reduction in the error squared.

Now this agreement is nearly perfect. The errors that I found now (2017) are equal to the errors of 2009 in the readme.txt in all cases but in eucsbacdemo 9cams, 9pts and even then they are equal to 0.2%. In 2017, the reasons for terminating are always better and nearly always because the squared error, or the function error did not reduce any further. In nearly every case the run time was 10 or over 10 times faster. I believe this is due to a more parallel and better optimized LAPACK algorithm, a multithreaded machine (Surface Book with 2 cores) and a faster CPU.

This took me about 9 days to figure out. The first issue was to get Clapack installed and compiled. I downloaded the precompiled libraries and then tried to run the makefile using Nmake that was supplied with the sba code. That didn’t work.

Eventually I untarred clapack 3.2.1 and ran CMAKE. That got a project that I could open in Visual Studio. Eventally I built clapack.lib and f2c.lib and blas.lib. I got the 32bit version to compile without errors and couldn’t get the 64 bit version to work.

I tried to run the sba 1-6 makefile for the main project file. I couldn’t get the demo to compile. I noticed that it was never looking in the directory where I wrote in the locations of the lib files. I used Nmake /f Makefile.vc with the Makefile in the code. It recursively looked for other makefiles as it should, but never worked. I decided to take a step back and stop trying to get an old library to work.

I decided then that I would just set the correct compile options in visual studio.

I installed the Intel C++ compiler and that comes with a Math Kernel Library that has all the Lapack routines but now in a more modern format, not just FORTRAN or a c interface. That took a while to get working too. I read the developers guide to help. I got first Intel’s demo’s to work. Eventually I figured out all the compiler options and got the demo to compile.

Unfortunately the demo compiled but did not produce the correct output. It failed in levmar.c It was finding negative roots for matrices that should be positive definite.

I rewrote the file lapack.c primarily and took out all the old Lapack function calls and replaced them with modern Lapacke function calls. In most cases with some study there was an equivalent.

My version compiled but it failed to prevent errors. I tried 64 and 32 bit and neither worked.

Eventually after studying the code I realized that the data was ok and it only failed after succeeding for same 10 data points. Instead of solving the Matrix with a Cholesky routine (that should work because the Jacobian is positive Definite) I figured some rounding error was at play. I tried the LU decomposition and that worked.

This demo demonstrates that the SBA library works correctly and is installed and compiled properly.

## Compiling and creating SBA.mexw64

With the demo compiled, we had several libraries needed to connect.

* SBA.lib library of the sba files essentially bundling all the sparse code together
* Lapack libs All the lapacke routines.
* MATLAB libs necessary to interpret the matlab specific C commands

Now what was needed was to get those packaged and linked up to make an sba.mexw64 library. Because there were several libraries to link up, I chose to use Visual studio to create the dll. I felt that would make it simpler to see and fix the compiling and linking errors. It took me over a day, but eventually that worked and I was able to create the mexw64. The time was taken following the matlab instructions, setting up the project correctly, and figuring out what is being done and what is a dynamic linked library. While I was at it, I also compiled and created a projac.dll which has C code for the projection function. I never used that routine.

## Getting SBA.mexw64 to work

The next challenge was to get the SBA.mexw64 to work. It has 17 arguments and those need to be set correctly. It at first failed to run at all because the compiled dll needed to be be exported as a mexFunction. We then had to make sure all the inputs were correct. See above about the description of the functions projection and projac and the testing to make sure they function. After a few trials, it was clear that the mexFunction entry is working but there were argument errors. The one that took the longest to figure out was that the spmask, a version of the visibility matrix, was not the correct size. It was in MATLAB, but on the C side the number of projections was not correct. Eventually, I got it to work by not using a sparse matrix at all and converting the transformed visibility matrix to double. I did some testing using the Visual Studio debugger, attaching it to the MATLAB process. Prior to running it I tested projection and projac thoroughly on the MATLAB side, where it was easier to debug. This was perhaps a two day operation. Finally it worked.

# Bundle Results

Here is the results of the Bundle Adjustment using several configurations. This is my set-up.

|  |  |
| --- | --- |
| Number 3D points | 6203 |
| Number of Cameras | 6 |
| Number of projections | 15606 |
| CPU Windows Surface Book | i7-6600U CPU |
| CPU Matlab Lenovo workstation laptop | i7-3720QM |

I ran the sparse bundle adjustment (SBA) and MATLAB’S nonlinear fit (MatNon) to do the bundle adjustment. MatNon was run on a dedicated Lenovo workstation. The above describes the number of 3D points, projections and cameras.



The table describes various runs, the initial and final errors, the iterations and the total time.

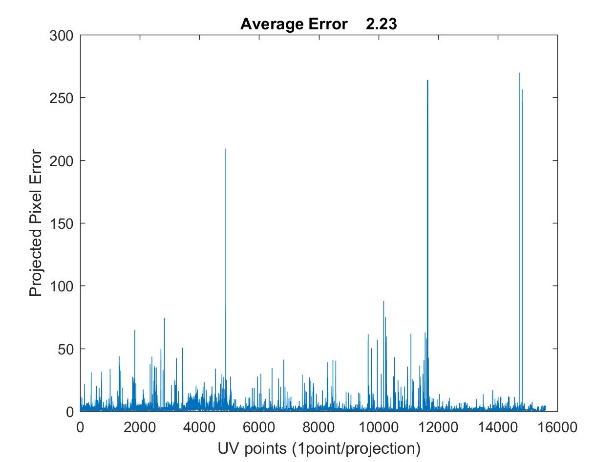
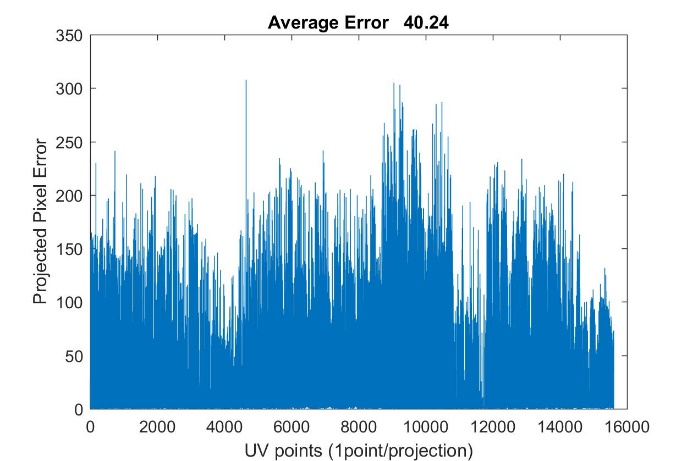
Matlab by itself actually could do the bundle adjustment (line 4). It took 42 hours for about 35 iterations. In each iteration it had to invert a 31000 by 18000 matrix to get the step and that took about 1 hour of computer time. It used BundleTriangleFunction described above that computes both the Jacobian and the projection. That was tested before, so it was clear that the Jacobian and projection were correct. I had set the max iterations to 20 prior and that was the reason it ended.

In contrast SBA produced a better answer in 90 seconds (line 1). It did 4 times as many iterations in a fraction of the time, about 1800 times faster than MATLAB. Numerically, I expect it to be more stable, because the sparsity is taken into account, making the entries that must be zero, exact, and not subject to rounding error.

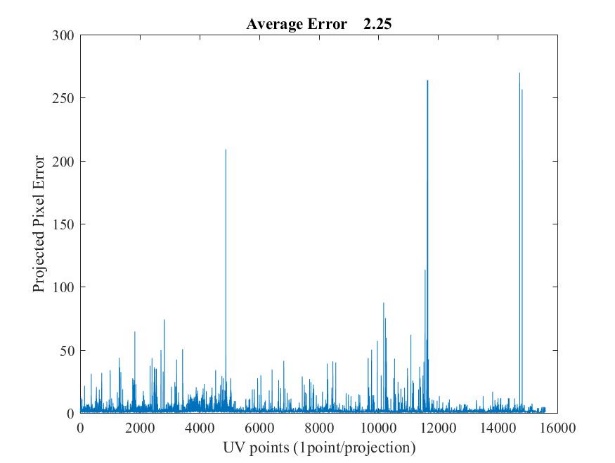
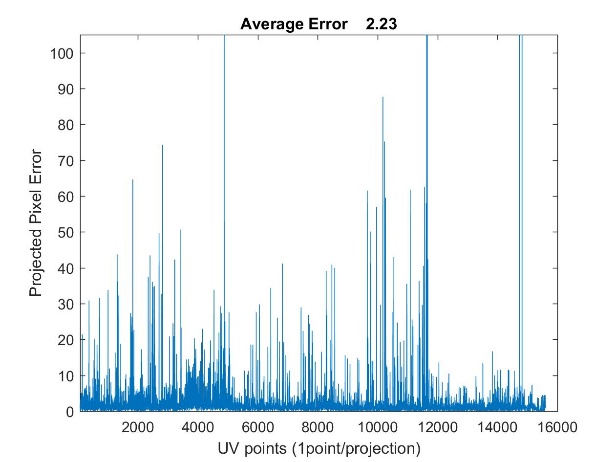
This function can be made much faster if the projection and projac functions are coded in C/C++, not MATLAB. That is what the projac.dll is for, but I didn’t use it. As it stands that expects 6 variables for the pose, not 7 as we have so that code would need to be modified. Examples in the Demo, for example with 54 cameras, took some 0.6 s, and that had more points and cameras. That run demonstrates that the bundle adjustment can be completed much faster.

Next, I tried removing projac and testing the result. That took about 8 times longer and produced nearly the same result. Without a Jacobian function, the Jacobian values are estimated using finite differences. Naively, one would need one function evaluation per Jacobian Column (some 18000) but using calculus, and the book cited in the SBA documentation, one can show that a few well chosen points are enough to calculate the entire Jacobian. The SBA run without the Jacobian had an excess of Jacobian calls and each one required several function calls (perhaps 3), and so that was likely the reason this was so much slower.

The next to last run in the table was SBA projection unscaled q. Essentially, q as a variable in the fit is unconstrained, yet the q to rotation transformation assumes a unit quaternion. To make the varying quaternion unit length, I normalized the q from the fit so that it was always unit length prior to calculating the rotation matrix. Similarly, the Jacobian is for a unit quaternion, so to make it work for an unscaled quaternion, I added the Jacobion dq scaled/dqunscaled , a 4 by 4 matrix. I tested these function again prior to running SBA and they did provide consistent answers for the projection—consistent with each other and consistent with the prior projection calculation. Now each function evaluates the square root, and each Jacobian also evaluates a square root and multiplies an additional 4 by 4 matrix. That increased the run time by 40%. The answer was the same.



The left image shows the errors before bundle adjustment, where each point is one projection. The right image shows the errors after bundle adjustment, It is clear that the error is reduced by at least 20 times. These left errors are as large as they are because the 3D position of those points is estimated from one pair of cameras, so when used in a different pair, the error will be larger. On the right, the bundle adjustment finds the best position. The points with very large errors (above 100) are the source of most of the error, and those are because the correspondence in swift must not be correct.

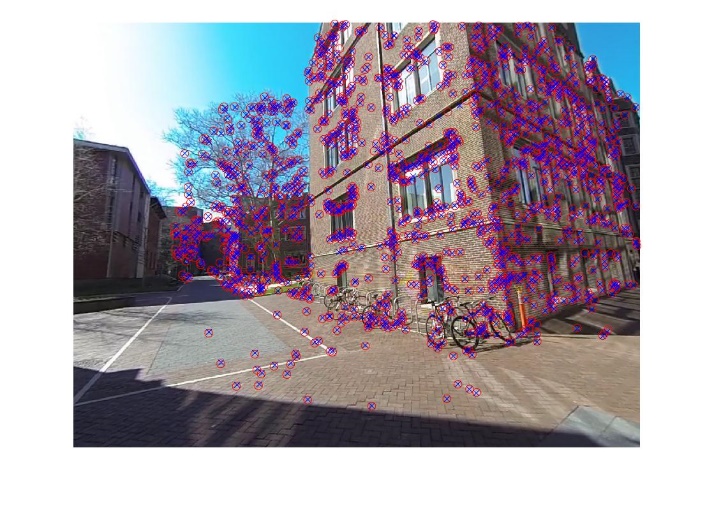
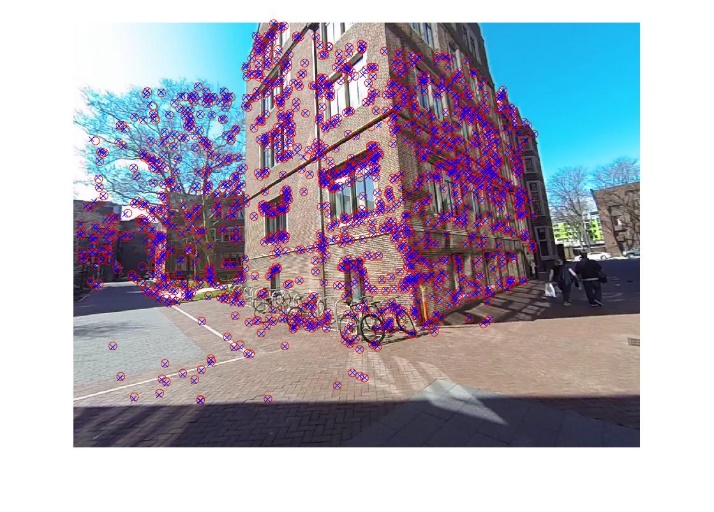


The left image is from the SBA fit just and is the same plot as the above right, but now zoomed in to see the errors. The right plot is the error plot as found from MATLAB’s Nonlinear fit. It matches the SBA fit, demonstrating that both algorithms are producing nearly the same result (but one is much faster).

## Checking the Projections and Errors After bundle adjustment.

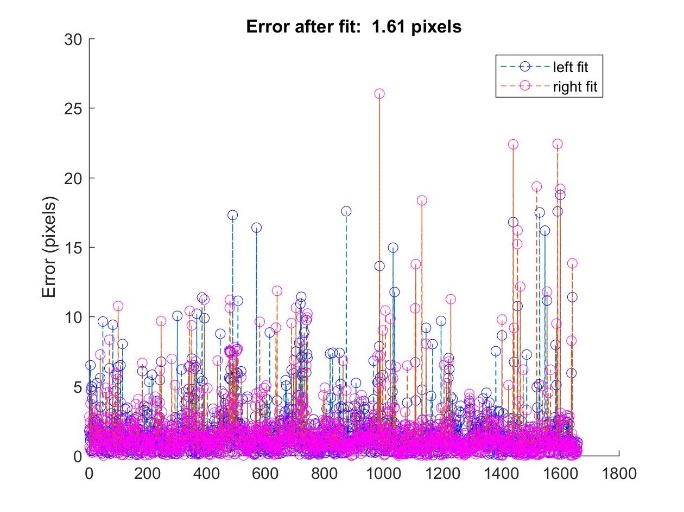
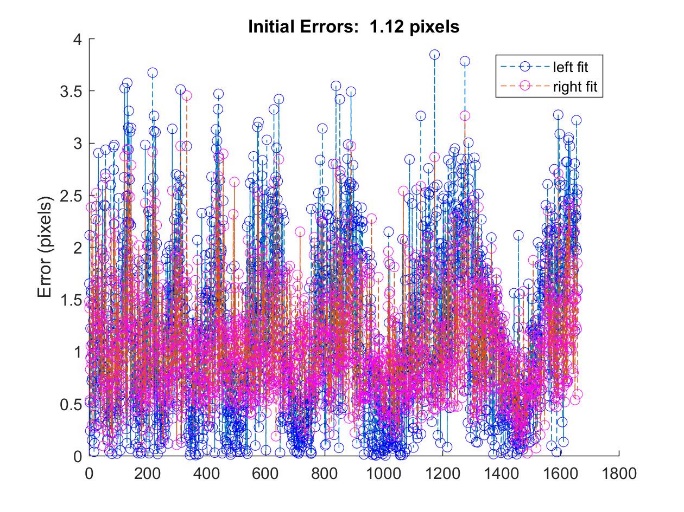
The above suggests that the errors are smaller, but it also helps to look at the pairs of images and to use the tools already developed to asses how close the projections are and whether the bundle adjustment improved the fit. This was done using a script MilestoneCheiralityB(setNo).

For image pair 2 -3 Here are the results:

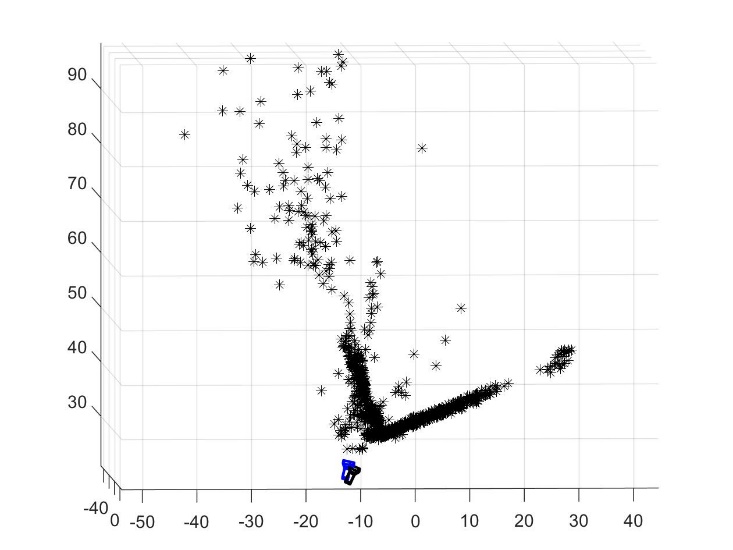


Left and right images showing that the bundle adjustment produces law projection errors.

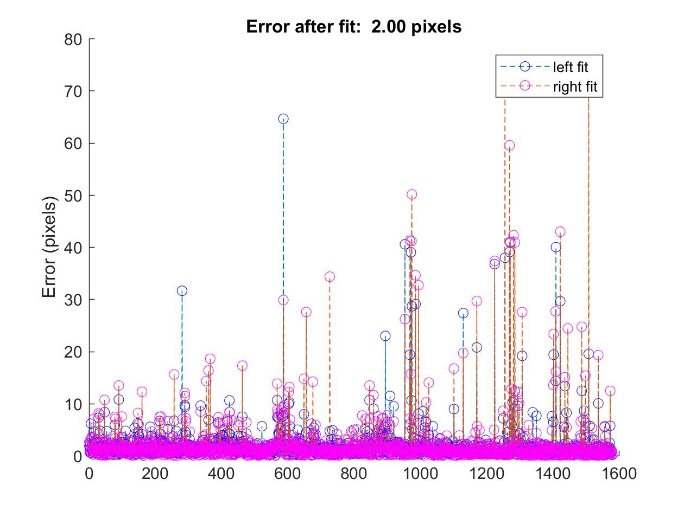
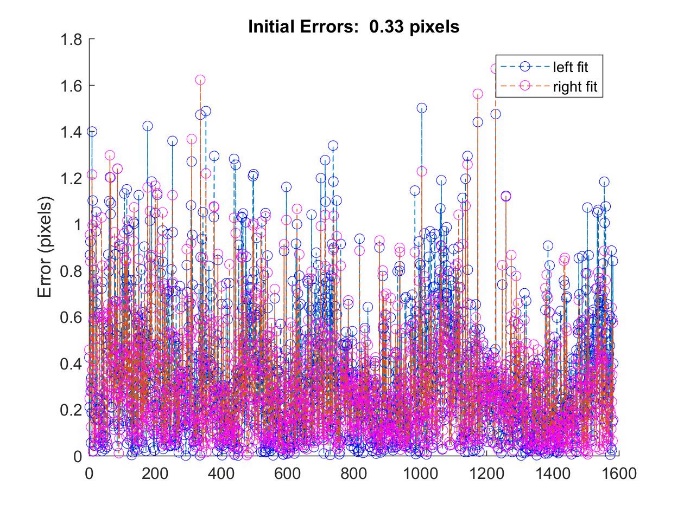
Here are the error plots for each projection:

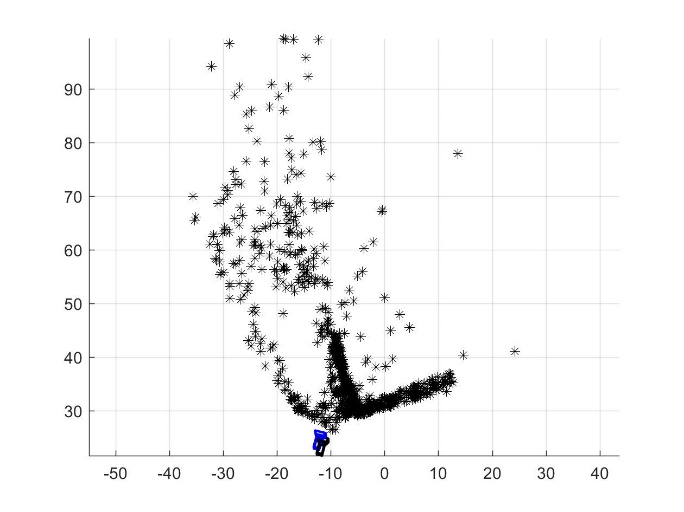
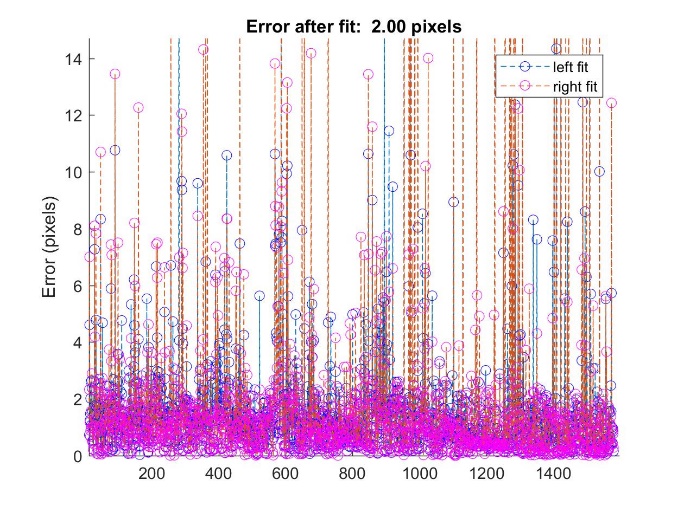


Here one can see that the Bundle adjustment improved the errors just considering the two poses 2 and 3 and their correspondences. The left figure shows the errors prior to the bundle adjustment. While the average is low, there is some pattern of errors because the errors oscillate. The right figure shows the errors after bundle adjustment: Here the oscillation is removed and instead of having many points with big pixel errors a handful have been isolated that produce big errors. If those were removed the error would be below the right hand side. Thus, the bundle adjustment improved this fit making, by identifying the few outliers and reducing the error on the inliers.

Here we see the camera pose for the two cameras after bundle adjustment. Many of the positions are correct. Camera 1 here is at a universal coordinate, so camera poses and points can be compared across image pairs. There are also points whose position could not be correct as evidence by the points within the building.

Another example shows the same pattern, this one from pictures 3-4





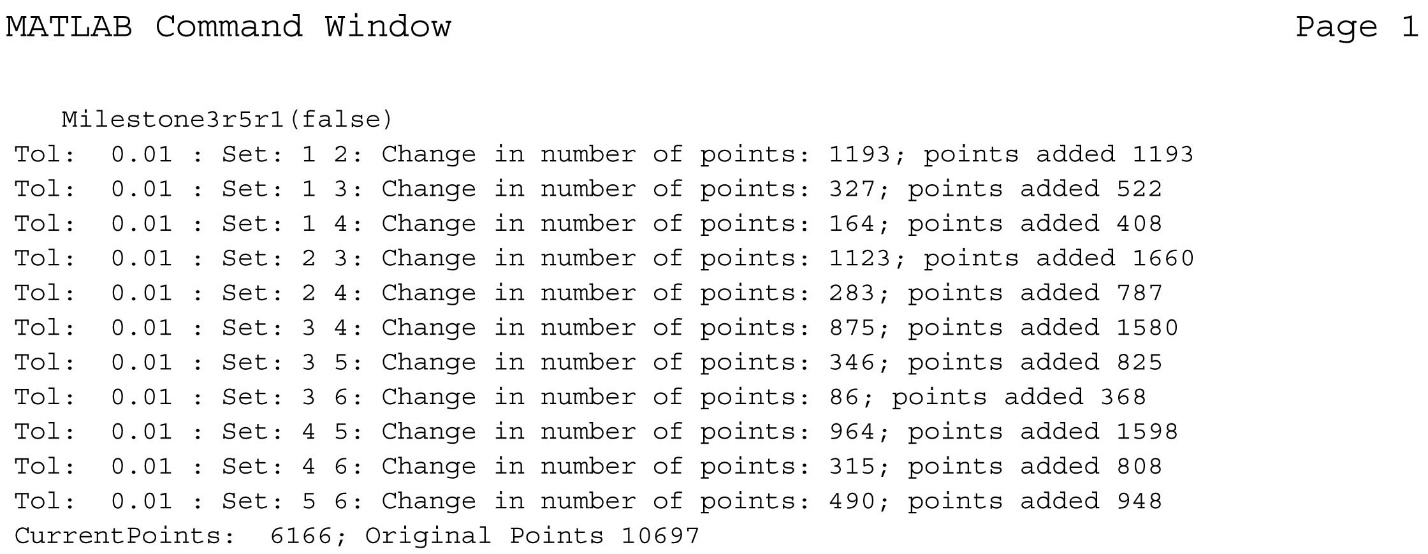
The top two figures show the error prior and after the bundle adjustment. As evidence in the zoomed in figure after the bundle adjustment, the pattern of residual errors has become nearly random as it should and outliers identified that result in large errors. The pose as shown in the lower right figure shows that the origin is the world coordinate system as used above and that most points follow the building contour. There are now a few points that violate Chierality, or have large errors (top right), or project into the building. With those eliminated, the average error will again drop.

### Post Script: Reducing the error by Elimintating Bundle Outliers

I wanted to be able to eliminate specific pairs of points in the bundle adjustment in order to remove isolated points that give large projection errors. To do this I revised the code for Milestone 5 parts 1 and 2. One part of the revision was to change C, R to be the Cset, Rset, so that the 3D coordinates are stored in the pairs in world coordinates. That required a change in data structure.

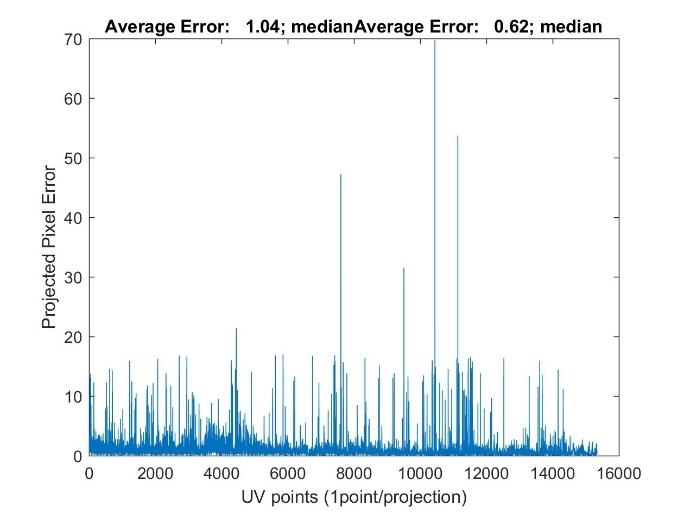
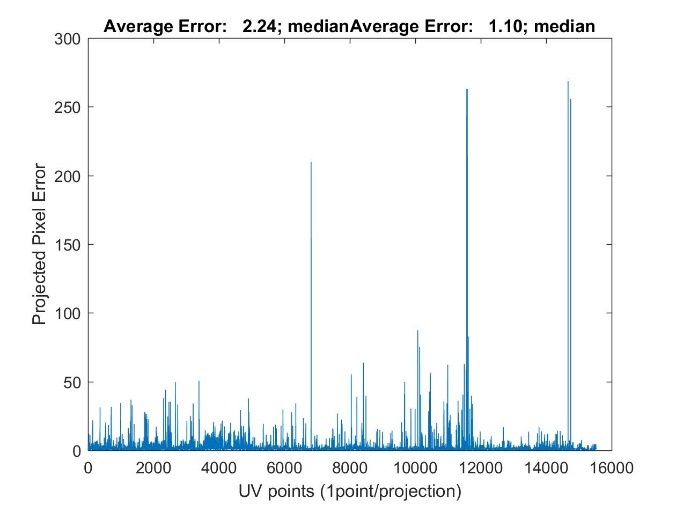
#### Change of data structure Milestone 5 r1

When building the visibility matrix, I found that before adding a new point, I should check if both cameras matched existing points first, then check if only one camera matches. That code overall allowed me to eliminate more 3D points with the same set.



This change resulted in about 35 fewer 3D points with the essentially the same data set. The only difference is that 3 nonChieral points after the first bundle adjustment were removed.

To reduce the error first sorted the 2D projections by their error and eliminated the worst outliers.



On the left is after the first bundle adjustment, similar to a picture above, where there are outliers with pixel errors of 250. The average and median are written in the title. On the right is the same data but after two rounds of elimination, first removing all projections with errors above 27, then removing all projections above 17 pixels. That reduced the median error by about half to 0.62 pixels and that is significant. This is comparable to the individual pairs but now the points are much better constrained because only one 3D point must be optimized to project to multiple cameras. This cut eliminated some 30 3D points to gain this improvement.

# Summary

The bundle procedure worked and was practical. The main idea is to take pairs and reconstruct the pose and the 3D positions using Ransac to find a good points and the pose, and using nonlinear fitting to optimize the 3D points by themselves, then the pose. This adjustment was done by using a quaternion for the rotations, and supplying a Jacobian function. That Jacobian in the final version even took into account that the quaternions were unscaled, and so calculated the derivatives and the projections by calculating conversion from an unscaled quaternion to normalized quaternion to a rotation matrix. The bundle adjustment also worked and produced low errors. It is really only practical with the sparse algorithm and preferably in C/C++. The sparse algorithm is so much faster mainly because of the Schur’s compliment algorithm. The Hessian can be considered 4 blocks, the biggest of which has only position information in is block diagonal where on the diagonal are 3 x 3 blocks, one for each of n 3D positions. Matlab looks for the inversion of that 3\*n x 3\*n matrix which takes probably one hour on my laptop and requires (3n)^3 operations. The Schur’s algorithm instead diagonalizes each 3 x 3 block which has about 3^3 \* n steps and scales linearly with the number of points. One can see this because it doesn’t much matter how many 3D points one has, say up to 10000, the sba algorithm finishes in a few minutes. For really big problems like a movie, one could analyze or 10 frames at a time (5 minutes per set).

The outliers give the largest contribution. After bundle adjustment the average error was about 2.2 pixels and the median error was about 1.1 pixels. By removing some 30 3D points, the ones with the largest projection errors the average and median reduced to 1.1 and 0.62 pixels, a very good improvement. There still are a substantial number of outliers in the range of 2 to 17 pixels. I think these are due to swift misalignment because the picture is from a brick wall that has many repeated patterns, that may be off by a few rows of bricks. I chose not to remove these. This shows that by performing the bundle adjustment, one merges many 3D positions and camera poses reducing the number of variables substantially, and furthermore reducing the projected errors. By removing the worst projections one produces an excellent fit to all the swift aligned points and a well constrained pose estimate.