CS 217 Homework 1

Zachary DeStefano, 15247592

Due Date: April 16, 2015

Problem 1

Using our assumptions about lenses, we have the following diagram:

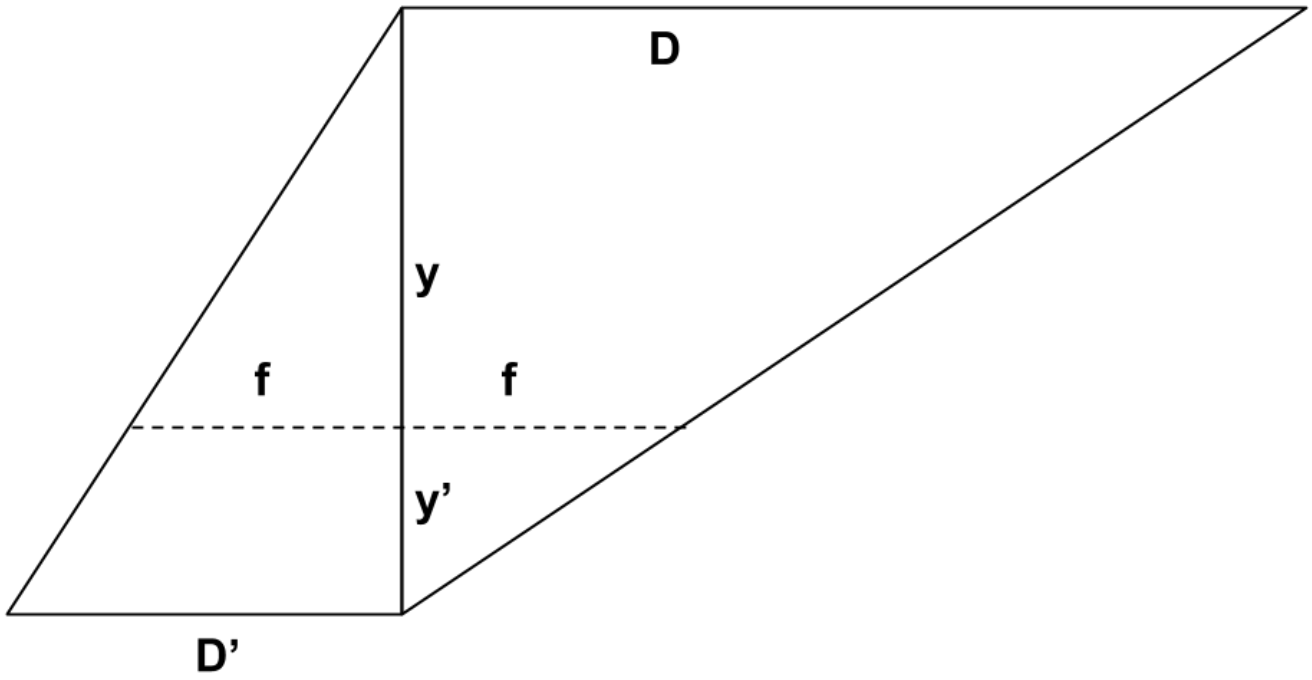

Figure 1: Diagram of lens and its focal length

The left side of the lens gives us the following equation using similar triangles

$$\frac{y}{f} = \frac{y + y'}{D'}$$

The right side of the length gives us the following equation using similar triangles

$$\frac{y'}{f} = \frac{y + y'}{D}$$

Adding together the two equations, we get the following

$$\frac{1}{f}(y + y') = (\frac{1}{D} + \frac{1}{D'})(y + y')$$

Cancelling the term $y + y'$ we end up with

$$\frac{1}{f} = \frac{1}{D} + \frac{1}{D'}$$

This is the thin lens equation

Problem 2

Here is my triangulate.m function. It is also attached in the code folder. After running testtriangulation.m, the points matched almost exactly before noise was added and they were quite close after the noise was added.

```
function X = triangulate(xL,xR,camL,camR)
%
% INPUT:
%
% xL,xR : points in left and right images (2xN arrays)
% camL,camR : left and right camera parameters
%
%
% OUTPUT:
%
% X : 3D coordinate of points in world coordinates (3xN array)
%
%

%{
ray shooting out from left camera is (x_l,y_l,1)*z + t_l
ray shooting out from right camera is R*(x_r,y_r,1)*z_r + t_r
R is equal to inv(R_l)*R_r
We need to minimize Au - t where
    A = [(x_l,y_l,1) -R*(x_r,y_r,1)]
    u = [z,z_r] and t = t_r-t_l
%}

N = size(xL,2);
X = zeros(3,N);

tVec = camR.t-camL.t;

for curNum = 1:N

    xLcur = (xL(:,curNum)-camL.c)./camL.m;
    xRcur = (xR(:,curNum)-camR.c)./camR.m;
    pixelLocL = [xLcur(1)/camL.f;xLcur(2)/camL.f;1];
    pixelLocR = [xRcur(1)/camL.f;xRcur(2)/camR.f;1];
    AmatL = pixelLocL;
    AmatR = inv(camL.R)*camR.R*pixelLocR;
```

```
    Amat = [AmatL -AmatR];

    %finally, u vector is pinv(A)*t
    u = pinv(Amat)*tVec;

    %averages the two locations together
    locL = AmatL*u(1)+camL.t; locR = AmatR*u(2)+camR.t;
    meanLoc = mean([locL locR],2);

    %rotate it back to get the final location
    finalLoc = camL.R*meanLoc;

    X(:,curNum) = finalLoc;

end


end
```

Part 1

The following diagram illustrates the cross-section showing the horizontal and vertical field of view along with the focal length. The field of view will be 2θ

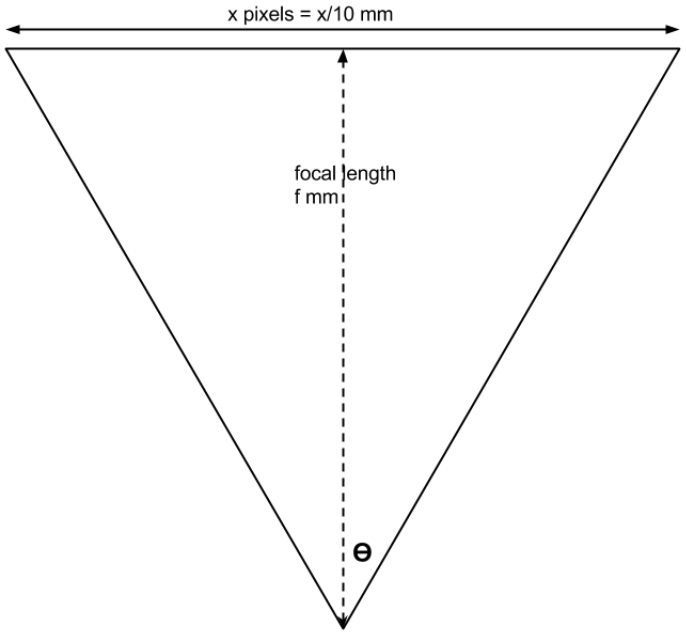

Figure 2: Diagram of focal length and horizontal/vertical field of view

Using trigonometry, it is easily observed that

$$\theta = arctan(\frac{x}{20f})$$

For the horizontal field of view, $x = 640$
For the vertical field of view, $x = 480$

When $f = 50$, the horizontal field of view (in degrees) is as follows:

$$2arctan(\frac{640}{20 \cdot 50}) = 65.2385$$

The vertical field of view (in degrees) is as follows:

$$2arctan(\frac{480}{20 \cdot 50}) = 51.2820$$

When $f = 100$, the horizontal field of view (in degrees) is as follows:

$$2arctan(\frac{640}{20 \cdot 100}) = 35.4893$$

The vertical field of view (in degrees) is as follows:

$$2arctan(\frac{480}{20 \cdot 100}) = 26.9915$$

Part 2

A rotation of θ about the y axis has the following form

$$\begin{pmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{pmatrix}$$

For us, $\theta = -45$ degrees, and we need homogeneous coordinates, thus our matrix R is the following

$$\begin{pmatrix} 1/\sqrt{2} & 0 & -1/\sqrt{2} & 0 \\ 0 & 1 & 0 & 0 \\ 1/\sqrt{2} & 0 & 1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translating to the right by 1 is the same as moving by one unit along the positive x-axis, which has the following matrix T

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Since we are translating to the right after rotation, we have to apply the matrix T in the new rotated coordinated system, thus to go from original world coordinate to final ones, you need to compute RT which ends up being

$$\begin{pmatrix} 1/\sqrt{2} & 0 & -1/\sqrt{2} & 1/\sqrt{2} \\ 0 & 1 & 0 & 0 \\ 1/\sqrt{2} & 0 & 1/\sqrt{2} & 1/\sqrt{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For $(1, 1, 1)$ in the world coordinate system, let p be the homogenous version of the point, so

$$p = (1, 1, 1, 1)^T$$

In order to find its coordinates in the new system, we need to find a vector x such that $RTx = p$ thus

$$x = (RT)^{-1}p$$

After computing x it turns out the point in the new coordinate system is as follows

$$(\sqrt{2} - 1, 1, 0)$$

Part 3

For this problem, I will assume that $y = z = 0$ for my left and right eye in the world coordinate system. I will assume that both eyes are 2 cm away from the bridge of my nose. Thus the left eye has $x = -2$ and the right eye has $x = 2$. Since I am looking at a computer monitor 40 cm away, I will also assume that I am looking at the point $(0, 0, 40)$ in the world coordinate system when estimating parameters. Here is a view from the top of my design:

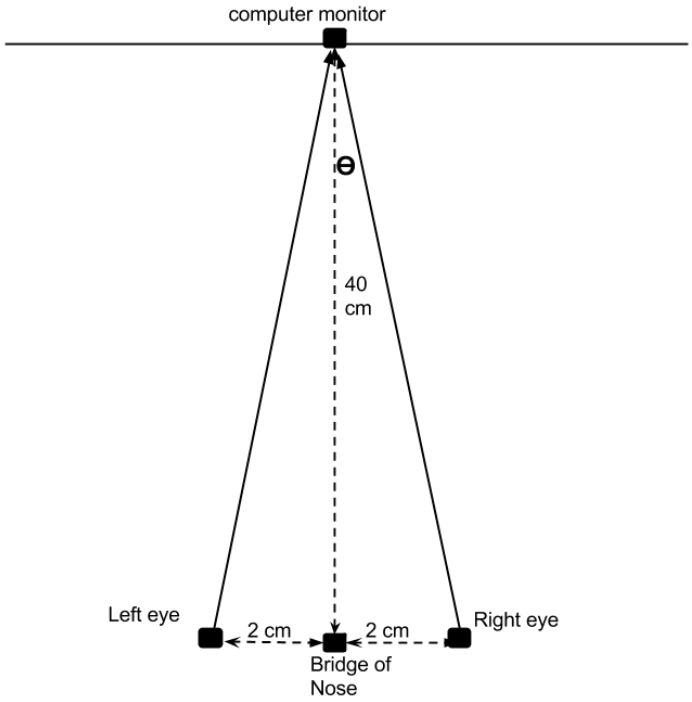

Figure 3: Diagram eye and monitor locations

We can easily specify the two translation vectors t_{left} and t_{right}

$$t_{left} = (2, 0, 0)^T$$
$$t_{right} = (-2, 0, 0)^T$$

We now just need the two rotation matrices R_{left} and R_{right} to finish estimating extrinsic parameters.

The right eye will rotate counter-clockwise by θ to look at the center point. The left will rotate clockwise by the same amount, so it will rotate by $-\theta$.

R_{right} will be the following:

$$\begin{pmatrix} cos(\theta) & sin(\theta) & 0 \\ -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

R_{left} will be the following:

$$\begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

From the diagram you can tell that

$$sin(\theta) = \frac{1}{\sqrt{401}} \approx 0.04993$$

$$cos(\theta) = \frac{20}{\sqrt{401}} \approx 0.99875$$

Part 4

As it turns out, noise in the pixel locations greatly increases the error in the recovered values. As the noise increases, the error increases in almost a linear fashion. Here is the code I used to test this. The arrays X,xL, and xR are the data points from the triangulation test script of the hemisphere. This code is also in the code folder as prob2part4.m.

```
%%
noise = [0.01 0.02 0.04 0.06 0.08 0.10 0.12 0.14 0.16 0.18 0.20 0.22 0.24];
numVals = length(noise);
error = zeros(1,numVals);
for i = 1:numVals

    xLnoisy = xL + noise(i)*randn(size(xL));
    xRnoisy = xR + noise(i)*randn(size(xR));
    Xrecov = triangulate(xLnoisy,xRnoisy,camL,camR);

   %this gives sum of squared errors
    error(i) = sum(sum((Xrecov-X).^2,1));

end

plot(noise,error);
```

This is the resulting plot:

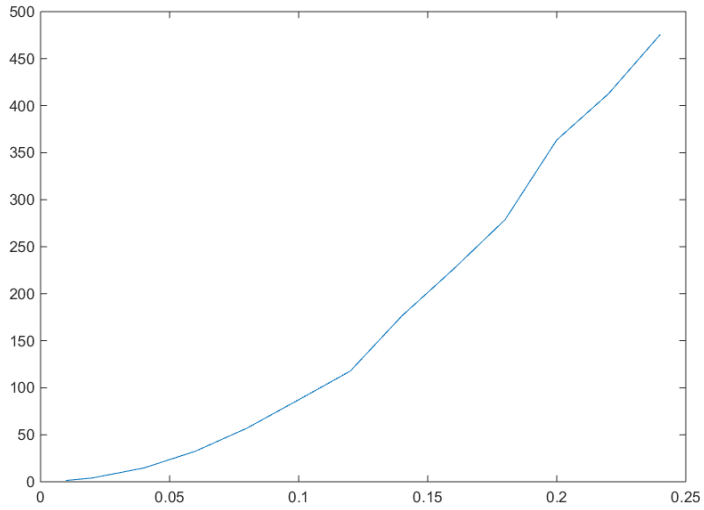

Figure 4: Sum of Squared Errors as function of amount of noise inserted

Part 5

In order to see how sensitive each of these were, I ran 50 tests for each part A to D and each test had a random variation of that specific camera parameter. I looked at the error for each test and averaged the errors together. These were my results:

Parameter changed	Average Sum of Squared Errors
Camera Center	403.8366
Focal Length	19.56
Camera Location	4.1943e4
Rotation Angle	2.34e6

As it turns out, the greatest amount of error comes from varying the rotation angle.

Instead of specifying specific units and the amounts for those units, we should be specifying the percent error possible in the camera parameters.

For Camera Center, we would specify a percentage of the image plane that the center could vary by
For Focal Length, we would specify a percentage of the focal length that the camera could vary by
For Camera Location, we would look at the range of x, y, z values in the real woord coordinate system. You would then specify a percentage and the amount in each dimension that the camera location is allowed to vary by would be a percentage of the range found.

For Rotation Angle, we could still specify a number of degrees since that is a fixed percentage of the allowed range of angles.

Here is the code I used to generate the numbers above. The beginning of it generated x, xL, and xR. The code is also in the code folder as prob2part5.m.

```
%%

%I am assuming that world coordinates are in meters

numVals = 50;

%Section A
error = zeros(1,numVals);
variation = 0.2; %number of pixels in 2 mm
cLorig = camL.c;
cRorig = camR.c;
for i = 1:numVals
    camL.c = cLorig + randn(2,1)*variation - [variation/2;variation/2];
    camR.c = cRorig + randn(2,1)*variation - [variation/2;variation/2];
```

```
    Xrecov = triangulate(xL,xR,camL,camR);
    %this gives sum of squared errors
    error(i) = sum(sum((Xrecov-X).^2,1));
end
errorA = mean(error);

%recover original location of camera center
camL.c = cLorig;
camR.c = cRorig;

%Section B
error = zeros(1,numVals);
variation = 0.01; %number of meters in 10 mm
for i = 1:numVals
    camL.f = 1 + 2*randn(1,1)*variation - variation;
    camR.f = 1 + 2*randn(1,1)*variation - variation;
    Xrecov = triangulate(xL,xR,camL,camR);
    %this gives sum of squared errors
    error(i) = sum(sum((Xrecov-X).^2,1));
end
errorB = mean(error);

%recover original focal length
camL.f = 1;
camR.f = 1;

%Section C
error = zeros(1,numVals);
variation = 0.1; %number of meters in 100 mm
tLorig = camL.t;
tRorig = camR.t;
for i = 1:numVals
    camL.t = tLorig + 2*randn(3,1)*variation - repmat(variation,3,1);
    camR.t = tRorig + 2*randn(3,1)*variation - repmat(variation,3,1);
    Xrecov = triangulate(xL,xR,camL,camR);
    %this gives sum of squared errors
    error(i) = sum(sum((Xrecov-X).^2,1));
end
errorC = mean(error);

%recover original translation vectors
```

```
camL.t= tLorig;
camR.t= tRorig;

%Section D
error = zeros(1,numVals);
variation = 5*pi/180; %5 degrees in radians

thyLorig = atan2(camL.t(1),10);
thyRorig = atan2(camR.t(1),10);

for i = 1:numVals
    thyL = thyLorig + 2*randn(1,1)*variation - variation;
    camL.R = [  cos(thyL)   0  -sin(thyL) ; ...
                0    1         0 ; ...
          sin(thyL)   0  cos(thyL) ];

    thyR = thyRorig + 2*randn(1,1)*variation - variation;
    camR.R = [  cos(thyR)   0  -sin(thyR) ; ...
                0    1         0 ; ...
          sin(thyR)   0  cos(thyR) ];

    Xrecov = triangulate(xL,xR,camL,camR);
    %this gives sum of squared errors
    error(i) = sum(sum((Xrecov-X).^2,1));
end
errorD = mean(error);

%recover original matrices
thy = atan2(camL.t(1),10);
Ry = [  cos(thy)   0  -sin(thy) ; ...
                0    1         0 ; ...
        sin(thy)   0  cos(thy) ];
camL.R = Ry;

thy = atan2(camR.t(1),10);
Ry = [  cos(thy)   0  -sin(thy) ; ...
                0    1         0 ; ...
        sin(thy)   0  cos(thy) ];
camR.R = Ry;
```

Problem 3

Here is my calibrate.m function. It is also in the code folder.

```
function cam = calibrate(X,x)
%INPUT:
%
% x : points in the image (2xN array)
% X : points in 3D (3xN array)
%
% OUTPUT:
%
% cam : estimated camera parameters
%

%%
%this sets up the A matrix
%       for Ac=0

n = size(X,2);
Pmatrix = [X(:,1:n);ones(1,n)];
xVals = x(1,1:n);
yVals = x(2,1:n);

%assemble the A matrix as specified in the slides
A = zeros(2*n,12);
for i = 1:n
    PiT = transpose(Pmatrix(:,i));
    start = 2*i-1;
    A(start:start+1,:) = ...
        [0 0 0 0 -PiT PiT.*yVals(i);...
        PiT 0 0 0 0 PiT.*(-xVals(i))];
end

%calibration is the last column of V in the SVD
[U,S,V] = svd(A);
calib = V(:,end);

%make the matrix have uniform scale
calib = calib./calib(12);
cam.C = (reshape(calib,[4 3]))';
```

```
translationCol = cam.C(:,4);
mainMat = cam.C(:,1:3);
[Kmat,Rinv] = rq(mainMat);

%normalizes by last entry
KmatNorm = Kmat./Kmat(9);
cam.K = KmatNorm;
cam.m = [KmatNorm(1,1);KmatNorm(2,2)];
cam.f = 1;
cam.c = [KmatNorm(1,3);KmatNorm(2,3)];

%gets rotation
Rmat = -inv(Rinv);
cam.R = Rmat;
cam.t = Rmat*inv(Kmat)*translationCol;

end
```

It uses the function rq.m that I found on the internet and was written by a professor at NYU. That function is also in the code folder.

```
function [R,Q]= rq(A)
% full RQ factorization
% returns [R 0] and Q such that [R 0] Q = A
% where R is upper triangular and Q is unitary (square)
% and A is m by n, n >= m  (A is short and fat)
% Equivalent to   A' = Q' [R']  (A' is tall and skinny)
%                         [0 ]
% or
%                A'P = Q'[P 0] [P R' P]
%                         [0 I] [  0   ]
% where P is the reverse identity matrix of order m (small dim)
% This is an ordinary QR factorization, say Q2 R2.
% Written by Michael Overton, overton@cs.nyu.edu (last updated Nov 2003)

m = size(A,1);
n = size(A,2);
if n < m
   error('RQ requires m <= n')
end
P = fliplr(eye(m));
AtP = A'*P;
```

```
[Q2,R2] = qr(AtP);
bigperm = [P zeros(m,n-m); zeros(n-m,m) eye(n-m)];
Q = (Q2*bigperm)';
R = (bigperm*R2*P)';

end
```

Part 1

I used the following code to generate the data set and then test the calibrate function:

```
%%
%this part generates the data set

%
% first generate our test figure in 3D
%
X = generate_hemisphere(2,[0;0;10],1000);

%
% set intrinsic parameters shared by both camers
%

%focal length
camL.f = 1;
camR.f = 1;

%pixel magnification factor
camL.m = [100;100];
camR.m = [100;100];

%location of camera center
camL.c = [50;50];
camR.c = [50;50];

camL.K = generateIntrinsic(camL);
camR.K = generateIntrinsic(camR);

%
% extrinsic params for left camera
```

```
%
camL.t= [-0.2;0;0];
thy = atan2(camL.t(1),10);
Ry = [  cos(thy)   0  -sin(thy) ; ...
              0    1          0 ; ...
        sin(thy)   0  cos(thy) ];
camL.R = Ry;

%
%extrinsic params for right camera
%
camR.t= [0.2;0;0];
thy = atan2(camR.t(1),10);
Ry = [  cos(thy)   0  -sin(thy) ; ...
              0    1          0 ; ...
        sin(thy)   0  cos(thy) ];
camR.R = Ry;

%
% now compute the two projections
%
xL= project(X,camL);
xR = project(X,camR);

camL.Ext = generateExtrinsic(camL);
camR.Ext = generateExtrinsic(camR);

camL.C = camL.K*camL.Ext;
camL.C = camL.C./camL.C(3,4);
camR.C = camR.K*camR.Ext;
camR.C = camR.C./camR.C(3,4);

%%

%this runs the calibration function
camLtest = calibrate(X,xL);
camRtest = calibrate(X,xR);

%verifies the calibration matrix
calibDiffL = sum(sum(abs(camLtest.C-camL.C)));
calibDiffR = sum(sum(abs(camRtest.C-camR.C)));
```

```
%verify m and c
mDiffL = sum(abs(camL.m-camLtest.m));
cDiffL = sum(abs(camL.c-camLtest.c));
mDiffR = sum(abs(camR.m-camRtest.m));
cDiffR = sum(abs(camR.c-camRtest.c));

%verify R
RdiffL = sum(sum(abs(camL.R-camLtest.R)));
RdiffR = sum(sum(abs(camR.R-camRtest.R)));

%verify t
tDiffL = sum(abs(camL.t-camLtest.t));
tDiffR = sum(abs(camR.t-camRtest.t));
```

All of the differences ended up being trivial and were likely due to floating point error, thus the function itself works very well.

I then introduced noise as was done in the function. I used the noisy pixel locations and kept the original 3D locations to see what camera parameters were obtained. The original script used a noise factor of 0.01. I decided to also use 0.02 and 0.1 as noise factors to see what increasing noise does to the parameters. Here was my script:

```
xLnoisy1 = xL + 0.01*randn(size(xL));
xRnoisy1 = xR + 0.01*randn(size(xR));

camLtest1 = calibrate(X,xLnoisy1);
camRtest1 = calibrate(X,xRnoisy1);

xLnoisy2 = xL + 0.02*randn(size(xL));
xRnoisy2 = xR + 0.02*randn(size(xR));

camLtest2 = calibrate(X,xLnoisy2);
camRtest2 = calibrate(X,xRnoisy2);

xLnoisy3 = xL + 0.1*randn(size(xL));
xRnoisy3 = xR + 0.1*randn(size(xR));

camLtest3 = calibrate(X,xLnoisy3);
camRtest3 = calibrate(X,xRnoisy3);

%%
```

```
camL.K
camLtest1.K
camLtest2.K
camLtest3.K

%%
camL.R
camLtest1.R
camLtest2.R
camLtest3.R

%%
[camL.t camLtest1.t camLtest2.t camLtest3.t]
```

I then compared the parameters obtained using the output. In order from original K matrix to the recovered K matrix as it gets noisier, here are the K matrices:

```
   100     0    50
     0   100    50
     0     0     1

  100.0364    0.0014   49.9896
         0  100.0330   49.9683
         0         0    1.0000

  100.0540   -0.0030   49.9785
         0  100.0624   50.0167
         0         0    1.0000

   99.8103    0.0289   50.1466
         0   99.8229   50.5030
         0         0    1.0000
```

As can be observed, the pixel magnification factors do start to vary as noise is increased but not by much. The pixel center location also starts to vary, but not by much. The skew factor however becomes non-zero suggesting that it appears there is skew in the lens when there is noise.

Doing the same thing but for the R matrices, here are the rotation matrices obtained

```
 0.9998         0    0.0200
      0    1.0000         0
-0.0200         0    0.9998

 0.9998    0.0000    0.0199
-0.0000    1.0000   -0.0003
-0.0199    0.0003    0.9998

 0.9998   -0.0000    0.0199
 0.0000    1.0000    0.0000
-0.0199   -0.0000    0.9998

 0.9998    0.0002    0.0207
-0.0002    1.0000    0.0033
-0.0207   -0.0033    0.9998
```

As can be observed, the rotation matrix does start to vary from the original but not by too much as the pixel locations get noisier.

Lastly, I compared the camera location parameter t. Since it was a vector, I lined by the vectors obtained in order from left to right by amount of noise in test. This was the result:

```
-0.2000   -0.1991   -0.1999   -0.1965
      0    0.0002    0.0014    0.0017
      0   -0.0008    0.0019   -0.0305
```

The camera location parameter obtained does start to increasingly vary as noise increases. Again though, the small amount of noise we introduced did not change it a whole lot.

All of the code for this part is included in the file prob3part1.m in the code folder. The functions generateIntrinsic.m and generateExtrinsic.m were used during execution and their code is in the code folder.

Part 2

This is the code I used to test it on image 1. It is in the code folder as prob3part2.m:

```
[yy,xx] = meshgrid(linspace(0,25.2,10),linspace(0,19.55,8));
zz = zeros(size(xx(:)));
X = [xx(:) yy(:) zz(:)]';

%gets the x values

%obtained using paint and visually recording the pixel location
origin = [588;775];
vectorInYdir = [626;734]-origin;
vectorInXdir = [629;812]-origin;

x = zeros(2,80);
index = 1;
for i = 0:8
    for j=0:10
        x(:,index) = origin+vectorInXdir*j+vectorInYdir*i;
        index = index+1;
    end
end

camEst = calibrate(X,x);
```

I ended up getting the following error:

```
Warning: Matrix is singular, close to singular or badly scaled.
         Results may be inaccurate. RCOND = NaN.
```

As it turns out, performing the SVD on our matrix fails to find the proper vector c that indicates a calibration matrix. The last column of the V matrix is all 0 values except for a -1 in the second to last entry. This likely occured because every point in the 3D coordinates has the same z value so there are many possible placements of the camera and the image plane. The 0 values result from the fact that the equation is not linearly independent.

The linear dependence comes from the fact that we have a plane in 3D. Thus we can place the image plane wherever we want and then find the other camera parameters. Likewise, we can place the camera wherever we want and then come up with a proper image plane.

Part 3

After running the program on the first image, it was able to extract these grid points:

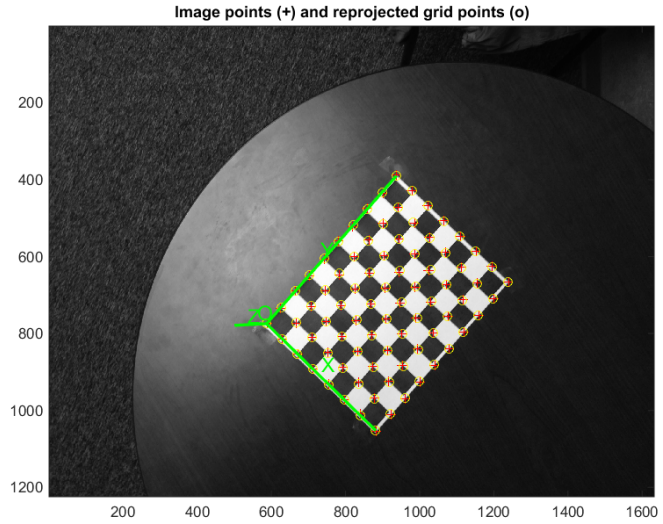

Figure 5: Results for Image 1

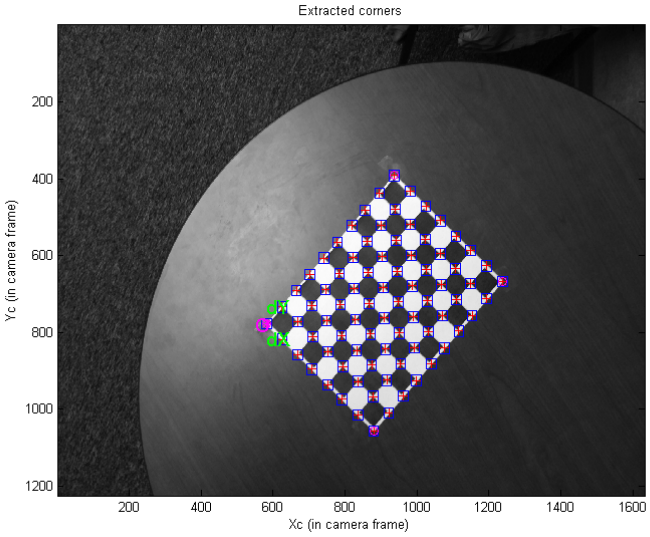

Figure 6: Results for Image 1

I continued specifying corners for the rest of the images and these were the instrinsic parameters that resulted:

```
Calibration results (with uncertainties):

Focal Length:          fc = [ 1602.72820   1627.13289 ]  [ 75.45401   76.76484 ]
Principal point:       cc = [ 830.78219   563.59275 ]  [ 41.54681   54.40452 ]
Skew:             alpha_c = [ 0.00000 ]  [ 0.00000  ]
=> angle of pixel axes = 90.00000  0.00000 degrees
Distortion:            kc = [ 0.61729   -2.11774   -0.01065   0.02242  0.00000 ]
 [ 0.20311   1.40105   0.01976   0.01779  0.00000 ]
Pixel error:          err = [ 3.59673   2.89028 ]

Note: The numerical errors are approximately three times the standard deviations (f
```

Here is the extrinsic matrix visual

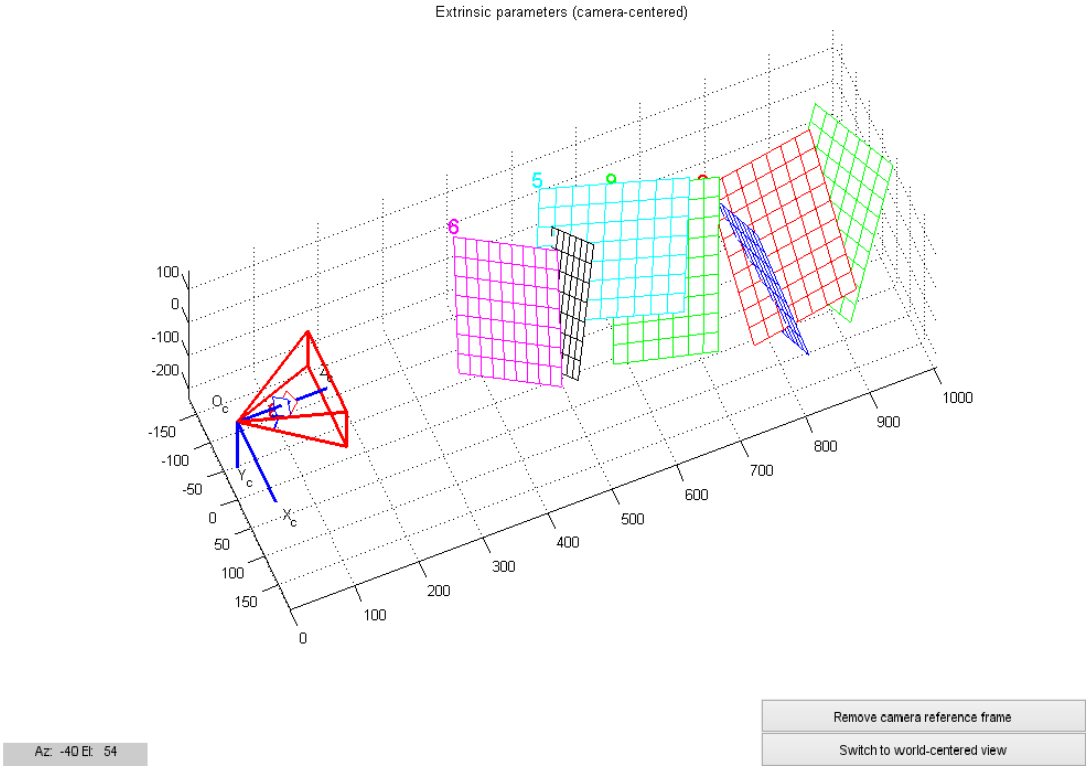

Figure 7: Extrinsic parameters with single camera

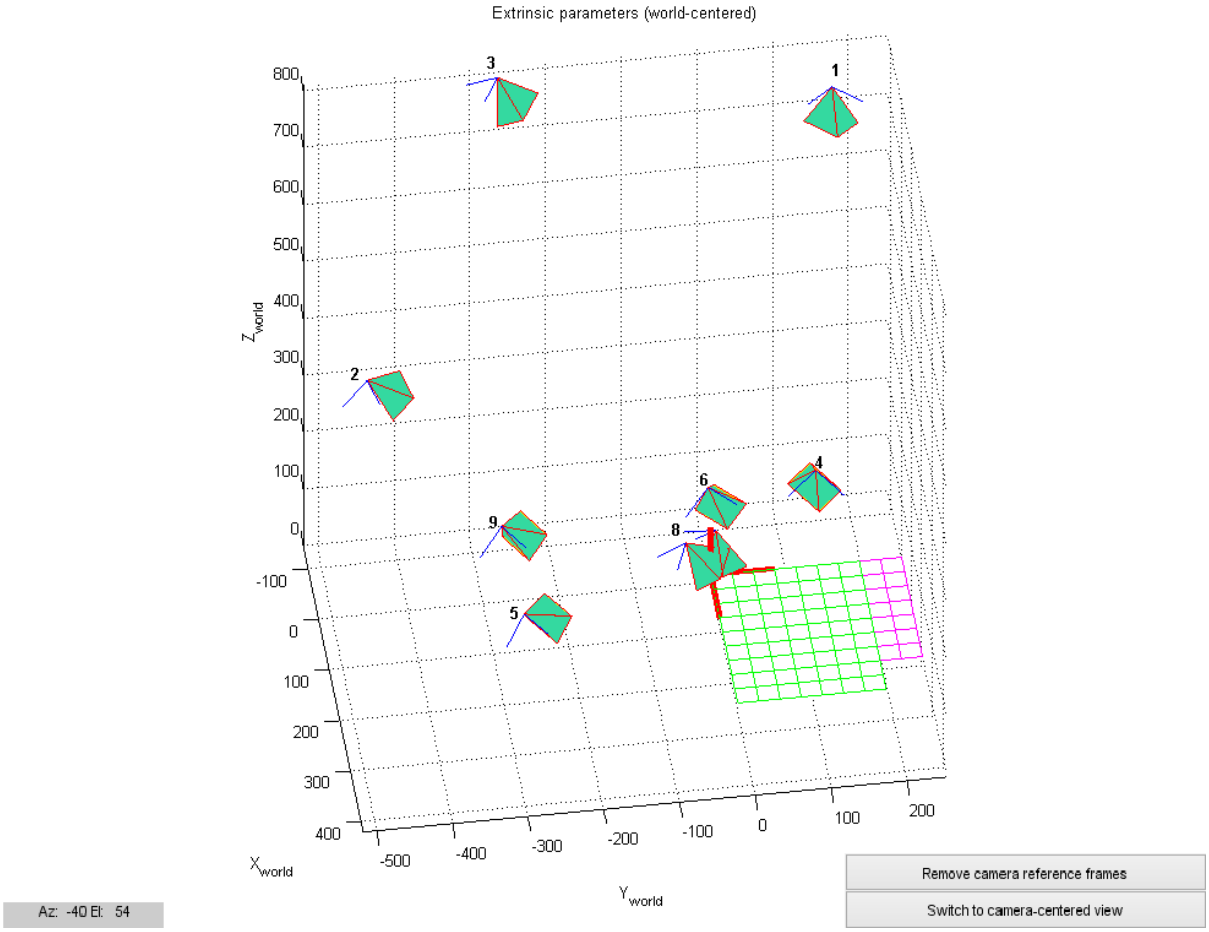

Figure 8: Extrinsic parameters with world viewpoint

When running this program, it ran through all the images and thus had many more points to use. Some of the differences that occured were likely the result of how the data was brought into the program. In some of the images, the grid corners were not very accurate whereas in other ones they were quite accurate. Also, the program seemed to arbitrarily label X and Y.

Part 4

Here are my estimated dimensions:
length = 15.0212
width = 3.4682
height = 0.6473

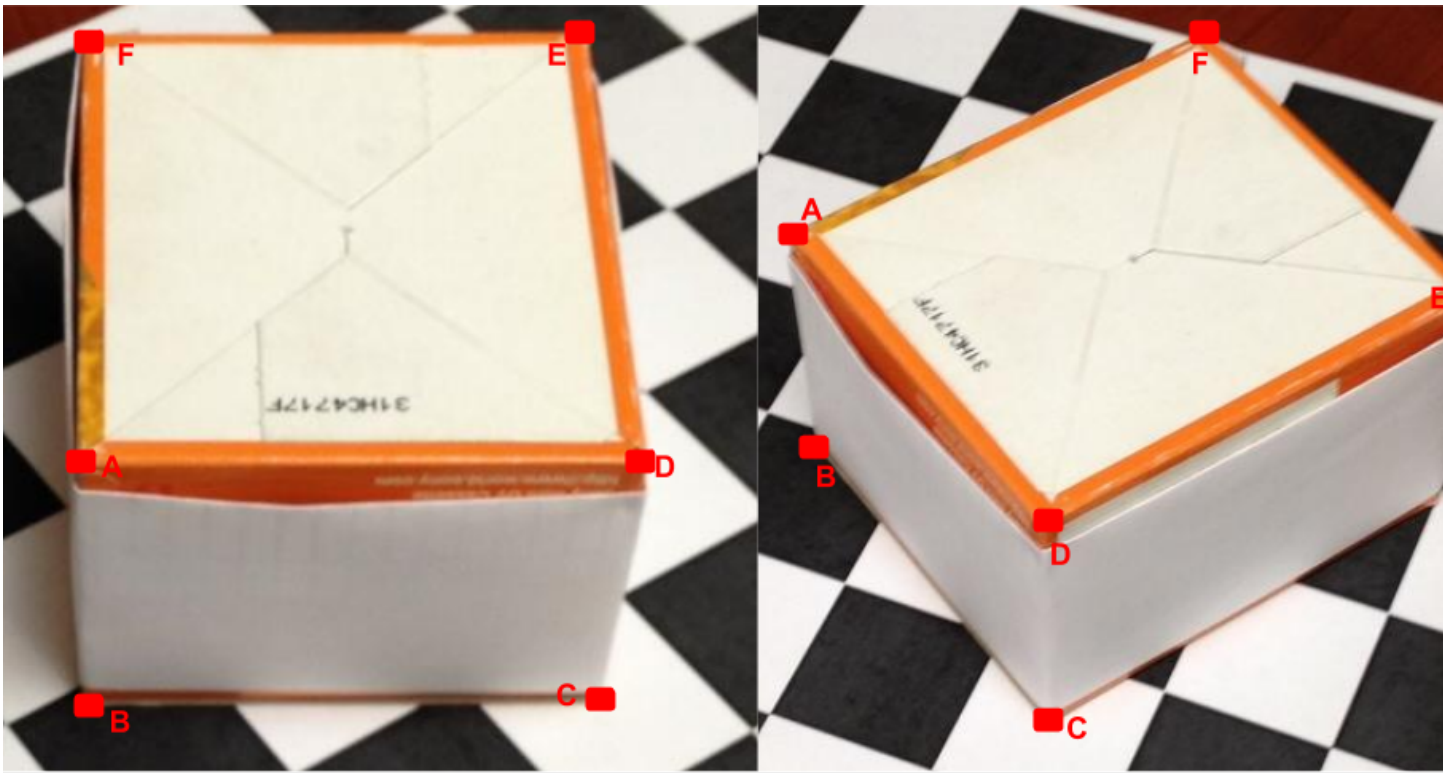

Figure 9: The points used to acquire dimensions

Here is the matlab code used to get the dimensions. It is attached in the code folder as prob3part4.m. The functions part4Input.m and part4function.m where used in execution and their code is attached in the code folder.

```
%gets the 3D checkerboard point values
[yy,xx] = meshgrid(linspace(0,25.2,10),linspace(0,19.55,8));
zz = zeros(size(xx(:)));
X = [xx(:) yy(:) zz(:)]';

%gets x values for image 13
%obtained using paint and visually recording the pixel location
origin = [739;1081];
vectorInYdir = [677;983]-origin;
vectorInXdir = [843;1015]-origin;
x13 = ones(3,80);
index = 1;
for i = 0:7
    for j=0:9
        x13(1:2,index) = origin+vectorInXdir*j+vectorInYdir*i;
        index = index+1;
    end
end

%gets x values for image 14
%obtained using paint and visually recording the pixel location
origin = [344;657];
vectorInYdir = [433;689]-origin;
vectorInXdir = [399;583]-origin;
x14 = ones(3,80);
index = 1;
for i = 0:7
    for j=0:9
        x14(1:2,index) = origin+vectorInXdir*j+vectorInYdir*i;
        index = index+1;
    end
end

%-- Focal length from previous problem
fc = [ 1602.728199544142200 ; 1627.132891974834400 ];
cam13.m = fc; cam14.m = fc;
cam13.f = 1; cam14.f = 1;
```

```
%— Principal point from previous problem
cc = [ 830.782186100948930 ; 563.592745345227400 ];
cam13.c = cc; cam14.c = cc;

%gets the modified points after considering instrinic matrix
Kmatrix = [fc(1) 0 cc(1);0 fc(2) cc(2);0 0 1];
invK = inv(Kmatrix);
newX13 = invK*x13;
newX14 = invK*x14;

%uses non-linear least squares to obtain extrinsic matrix
%   parameters are theta_1,theta_2,theta_3 as well as x,y,z
%   the first set are the angles for the rotation matrices
%   the second set is the translation vector
Xhom = [X;ones(1,size(X,2))];
Result13 = lsqnonlin(@(YY) part4function(YY,Xhom,newX13),[0 0 0 0 0 0]);
Result14 = lsqnonlin(@(YY) part4function(YY,Xhom,newX14),[0 0 0 0 0 0]);

%gets the R,t result for camera 13
[Rinv,tResult] = part4Input(Result13);
Rmatrix13 = inv(Rinv);
tVector13 = -Rmatrix13*tResult;
cam13.R = Rmatrix13; cam13.t = tVector13;

%gets the R,t result for camera 14
[Rinv,tResult] = part4Input(Result14);
Rmatrix14 = inv(Rinv);
tVector14 = -Rmatrix14*tResult;
cam14.R = Rmatrix14; cam14.t = tVector14;

%points are in order A,B,C,D,E,F as shown in my diagram
pts13=[792 794 1066 1091 1060 793;564 692 690 564 335 337];
pts14=[673 688 822 820 1068 909;401 520 682 566 433 286];
boxCoords = triangulate(pts13,pts14,cam13,cam14);
Acoord = boxCoords(:,1);
Bcoord = boxCoords(:,2);
Ccoord = boxCoords(:,3);
Dcoord = boxCoords(:,4);
Ecoord = boxCoords(:,5);
Fcoord = boxCoords(:,6);
```

```
%two possible heights (how much the box rises from the checkerboard)
height1 = norm(Acoord-Bcoord);
height2 = norm(Ccoord-Dcoord);
height = mean([height1 height2]);

%two possible widths (the letters travel along the width dimension)
width1 = norm(Acoord-Dcoord);
width2 = norm(Bcoord-Ccoord);
width = mean([width1 width2]);

%two possible lengths (final dimension that is neither of the above)
length1 = norm(Acoord-Fcoord);
length2 = norm(Dcoord-Ecoord);
length = mean([length1 length2]);

length
width
height
```