Machine Vision Assignment 2
John Goodacre – January 2016

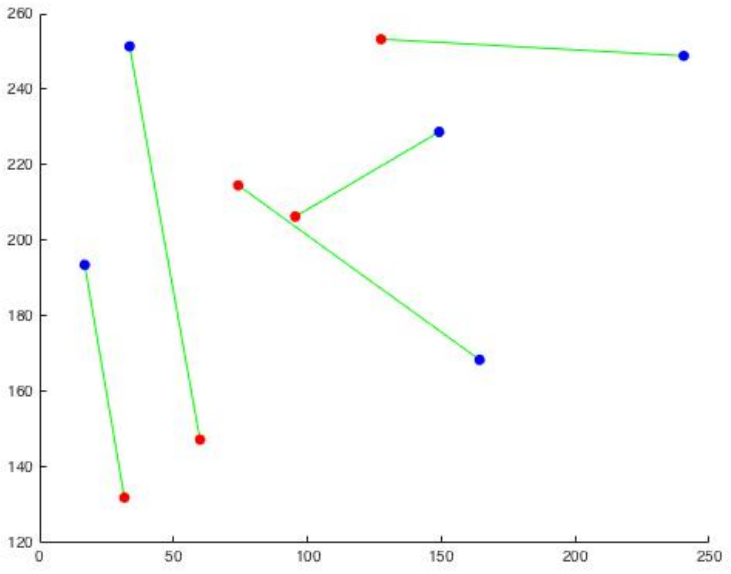
Homographies Part 1

A) Complete the TO DO's.

Given 5 respective points in Cartesian coordinates we are are asked to calculate a Homography between these points. There are various geometrical transformations possible from a translation to rigid body transformations, similarity transformations to affine and finally to homography. In general each of these transformations can be represented via the homography matrix. This is projective geometry.

Depending upon the type of transformation and what is preserved then that sets the degrees of freedom of our matrix and thus the number of data points needed. The homography matrix for image points is applied to the homogenous coordinates and is thus a 3x3 matrix with 8 degrees of freedom (thus needing 4 points). An affine transformation would only have 6 degrees of freedom and really only require the first two rows, a translation, really only the first two points of the last column.

The original points generated are shown below. We apply a full homography.



The code created is written in the function, calcBestHomography below.

```
function H = calcBestHomography(pts1Cart, pts2Cart)

%**** TO DO ****;
%first turn points to homogeneous
UV = [pts1Cart;ones(1,size(pts1Cart,2))];
XY = [pts2Cart;ones(1,size(pts2Cart,2))];

%then construct A matrix which should be (10 x 9) in size
%to speed up this has now been vectorised so the matrix rows are not
%in the same order, so i can put blocks in the matrix

U = UV(1,:); V = UV(2,:);
X = XY(1, :); Y = XY(2,:);
n = size(pts1Cart, 2);
rowsZero = zeros(3, n);
rowsUV = -[U; V; ones(1,n)];
AX = [rowsUV; rowsZero; X.*U; X.*V; X];
AY = [rowsZero; rowsUV; Y.*U; Y.*V; Y];
A = -[AX AY]';

%solve Ah = 0 by calling
h = solveAXEqualsZero(A);
H = reshape(h',3,3)';
```

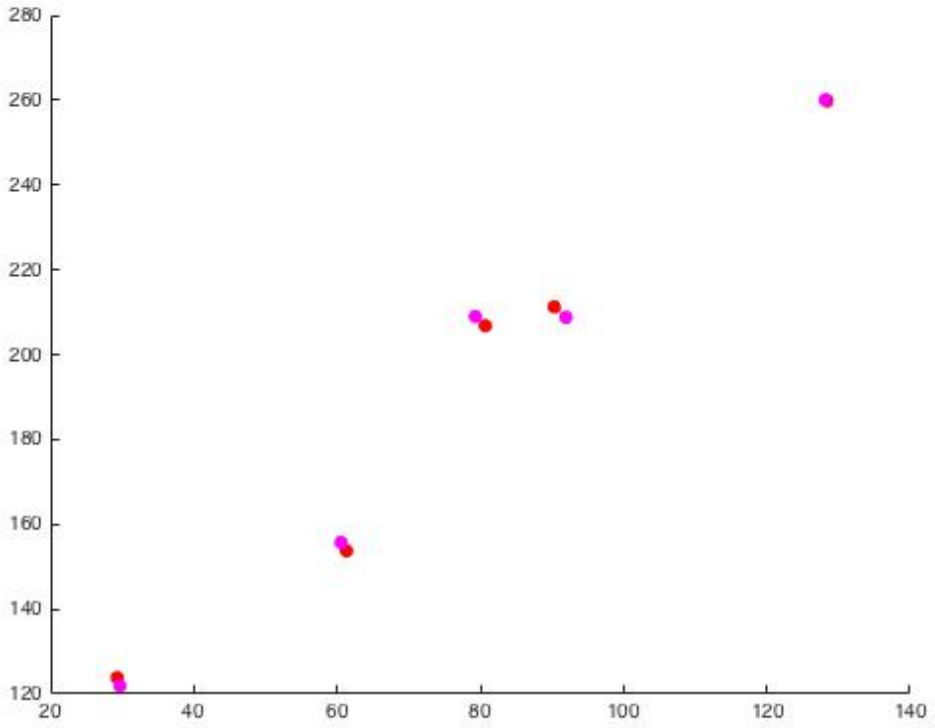
The code first translates our respective cartesian points to homogenous points (by adding a dimension of ones). Then the matrix for calculating the Homography is populated. Finally We solve Ah=0 in another function and TO DO which I will move onto later.

A point to note, in the lab I first populated this matrix by hand (painfully from the lecture notes), and then in a loop. I looked to vectorise and found the idea on the web whereby by changing the row order, blocks of zeros and ones could be put in in one go, leading to the fully vectorised code above. Strictly speaking the idea to populate as above isn't mine, although the code is (I am not sure there is a monopoly on how to populate a small matrix, but just in the aims of full disclosure!) there is a lot of material on homographies on the web and I wouldn't be able to cite a single source.

TO DO – solve Ax=0

This is done using singular value decomposition, the intuition is that grabbing the last column of V in U'DV, is the eigenvector corresponding to the smallest singular value and thus the solution to least squares. Code is in the function `solveAXEqualsZero(A)`.

The chart below shows the points successfully mapped by the homography calculation.



Note:- 1) noise was added to make this more realistic 2) The matrix for homographies has 8 degrees of freedom thus requiring 4 pairs of points. I believe is of course possible to attempt to map degeneracies such as points on a line to points not on a line. So this is not a blanket statement but true in general. Despite only needing 4 points, more can of course be helpful due to noise.

Complete the last TO DO's

```
**** TO DO ****
%1. Convince yourself that the homography is ambiguous up to scale
(by multiplying it by a constant factor and showing it does the same
thing).
```

This is demonstrated in the code with the following two lines

```
scale = -100000
pts2EstHom = scale*pHEst*pts1Hom
```

This creates identical results. Mathematically scaling doesn't matter because we divide by the last row when we convert back to Cartesian and indeed ties into the reason why despite having 9 matrix entries the homography matrix has 8 degrees of freedom.

```
%TO DO
%2. Show empirically that your homography routine can EXACTLY map any
%four points to anyother four points

pts1Cart = [  100   150   200  400;...
              1     100   200  50 ];
pts2Cart = [  10000   10   1000  20000;...
              100     54   500  10000 ];
Hest =calcBestHomography(pts1Cart, pts2Cart);

pts1Hom = [pts1Cart; ones(1,size(pts1Cart,2))];
pts2EstHom = Hest*pts1Hom;
pts2EstCart = pts2EstHom(1:2,:)./repmat(pts2EstHom(3,:),2,1);

%calculate mean squared distance from actual points
sqDiff = mean(sum((pts2Cart-pts2EstCart).^2))
```

The code above shows this empirically, sqdiff = 5.5142e-10.

B) **Here we are given 3 images. We will use homographies to create a panorama.** To do this we require 4 points in image 1, mapped to 4 points in image 2, and the same between image 1 and image 3.

Without loss of generality, we concentrate on the homography between image 1 and image 2. This enables us to loop through every pixel in image 1, map it to image 2. If that pixel actually exists within image 2 then we know we can add that pixel back to image 1. We do this for every pixel. (And then the same for image 1 and 3).

The code stub is given below.

```
for i =1:size(im1,2)
    for j=1:size(im1,1)
%transform this pixel position with your homography to %find where it
is in the coordinates of image 2
        pt2H = HEst*[i;j;1];
        pt2C = pt2H(1:2,:)./pt2H(3,:);
        px_pos = round(pt2C);
%if it the transformed position is within the boundary of image 2
%then copy pixel colour from image 2 pixel to current %position in
image 1 draw new image1 (use drawnow to force it to draw)
        %end
        x = px_pos(1,1);
        y = px_pos(2,1);
        if x >= 1 && y>=1 && x<=size(im2,2) && y<=size(im2,1)
               im1(j,i,:) =  im2(y,x,:);
        end
    end
end;
```

The process between image 1 and 3 is identical so I won't comment further.

The panorama created is given below.

Although the projections are correct, this is not pleasing to my eye at least and a possible improvement might be to project this image onto a cylinder and unroll to render nicely.

Homographies Part 2

c) Complete TO DO's for geometry of a single camera.

My code within projective Camera.m, first converts the world Cartesian coordinates to homogeneous and then by pre-multplying by our extrinsic matrix gives maps our coordinates to the camera's system. This is normalised and mapped from the camera to the image system by pre-multiplying by the camera intrinsic matrix. Then finally converted to image Cartesian coordinates by dividing by the last row.

The next TO DO asks for some noise (standard deviaton of 1 pixel to be added), I did this by `xImCart = xImCart+randn(size(xImCart));`

Plane Pose – Extrinsic matrix

The final set of TO DO's is to estimate the plane pose relative to a camera (the extrinsic matrix) and is given Cartesian image and world points and the intrinsic matrix.

The function converted the image coordinates to homogeneous in the usual way by adding an extra dimension of ones but this time to go from the image coordinates back to the normalised camera coordinates I pre-multiply by the inverse of the intrinsic matrix. This gives me coordinates for both the camera and the real world for which I can use the function I wrote previously to calculate my homography matrix.

So far these geometrical mapping were 'reasonably' self-explanatory from the notes and a little head-scratching. However, for the next phase I applied the algorithm rather mechanically in order to calculate our estimated extrinsic matrix. I agree with Simon Prince in the course book that the algorithm is slightly messy.

The point is that the extrinsic matrix has a particular structure, with the first columns being equivalent to our rotation matrix, and the last column being our translation. I thus followed the algorithm to recover this structure from the homography matrix.

First it is possible to recover the rotation part of the extrinsic mapping by doing an SVD of the first two columns our homography but then multiplying by `R=U*[1 0;0 1;0 0]*V';` the third row of the extrinsic matrix is the cross-product of the first two columns `R = [R cross(R(:,1),R(:,2))];`

The previous mappings need not necessarily give a rotation matrix (ie determinant of +1). Thus if this is not the case we make it so by changing the sign of the third column.

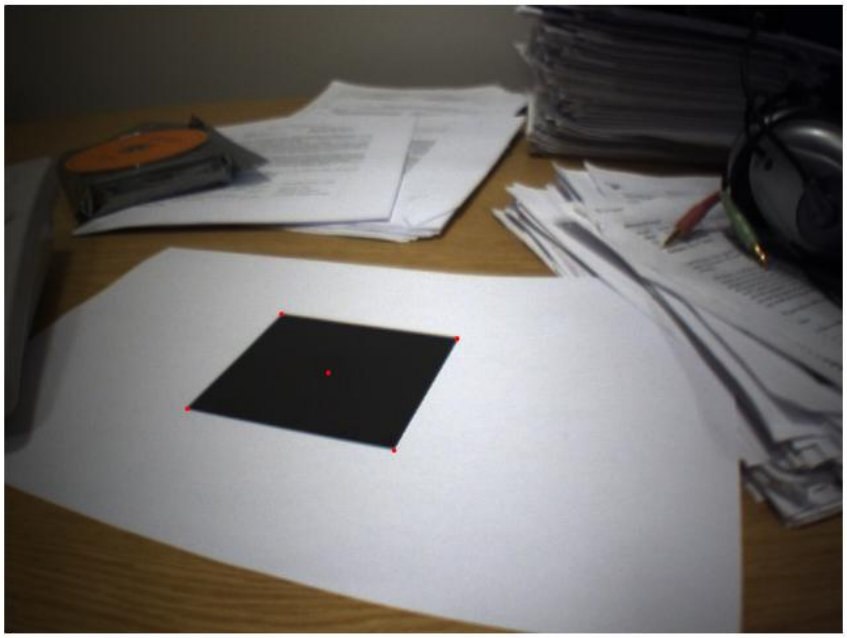
We now have an appropriate rotation matrix, however the first two columns of the Homography matrix will be a scale factor different than the first two columns of the rotation matrix we calculated (note that above we replaced the diagonal of the SVD of H with [1 0;0 1;0 0]. We can recover our translation values by finding this scale factor. I then used the third column of the homography matrix and divided by the scale factor to recover our translation values. At this point again as per the lecture notes we could have a number of different sign based outcomes. And so if the translation vector is negative multiply the first two columns of our rotation matrix by -1 and the translation column also if this is the case.

The structure of the extrinsic matrix is now in place. Ie Rotation plus translation and a row of [0 0 0 1] added.

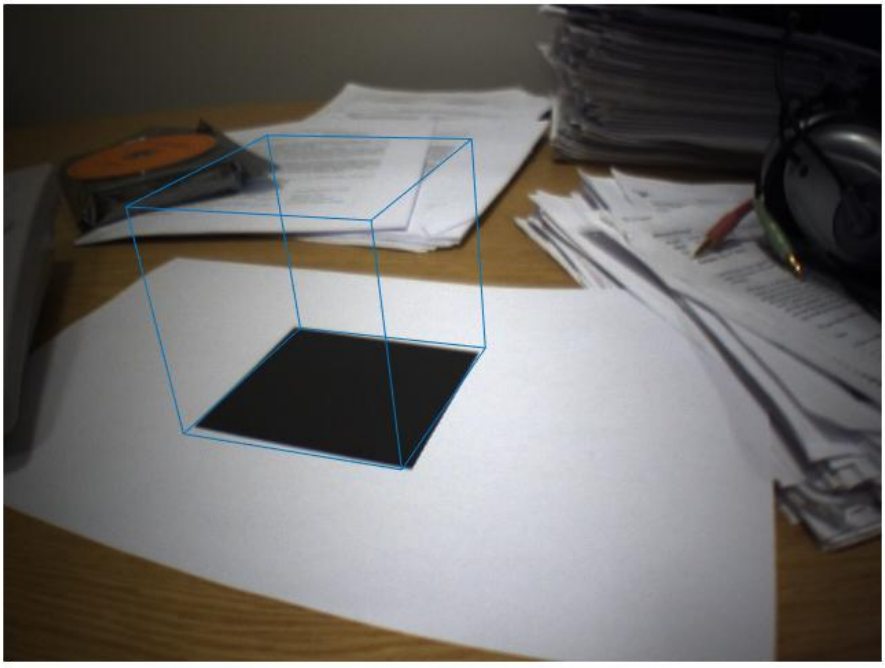
At the end of the code I tested to see if the estimated extrinsic matrix was close in value to the one given. And found this to be the case with of course some small inaccuracies.

D) Complete TO DO's for practical 2b

Practical 2b is a short application of the work done before. We are given cartesian image points (5 shown in red below), together with 3D plane points and the camera's intrinsic matrix.



The first TO DO asks us to calculate the extrinsic matrix. This is using the function I wrote above. We then define a cube in our plane coordinate system built from our 4 corners of the square above. We use the projective camera function to calculate our cube estimated image points.



My results are given in the image above. Note that the points are semi-ok but certainly not exact. Indeed the bottom left and top right cube mappings are off slightly skewing the cube.

At each stage errors could be introduced, the given mapping between given image coordinates and world coordinates may not be exact and indeed the intrinsic camera matrix was also given and could introduce errors – the result of the above would give an inaccurate extrinsic matrix and propagate to an inaccurate wireframe.

The matrix was calculated on very few points, although one would have hoped that the corner points being easier to map exactly would be reasonably accurate. The middle point of the square may be less certain as an exact mapping compared to a corner and of course if any of the point to point mappings are inaccurate with so few points then we are highly likely to calculate an inaccurate extrinsic matrix.

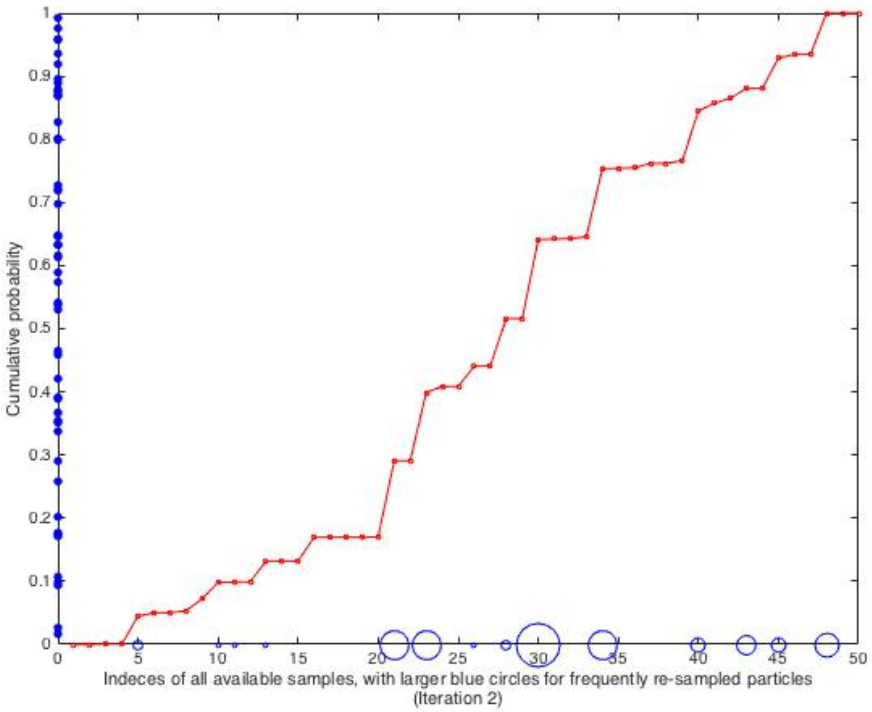
E) Condensation – Practical 9a

The first To DO's involve initialisation, which is equally weighting our 50 particles and propagating all samples. From these 50 random particles we map to the image space to have them represented in image space.

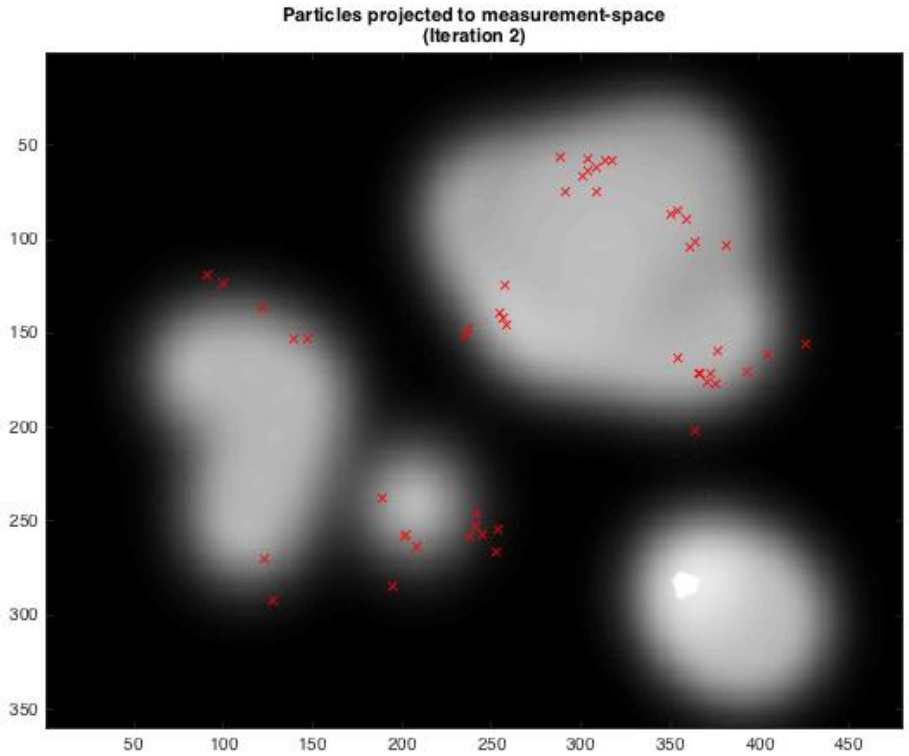
From here we begin looping, the idea is to take a prior distribution, sample particles and weights based on this prior and then to apply our motion model. In this case I simply add some random noise. This takes us to the next time-step where we can assess the likelihood of each particle. By pulling the intensity values from the image, these weights are normalised and we thus have a new set of particles and weights for the next iteration of the loop.

An observation compared to say the kalman filter, which is linear with gaussian noise is that whereas the kalman filter is unimodal the particle filter is not restricted in the type of distribution it can represent. This practical of course involves no 'actual' movement or tracking and so we are simulating looping through the same image and using intensities to 'guide' our particles.

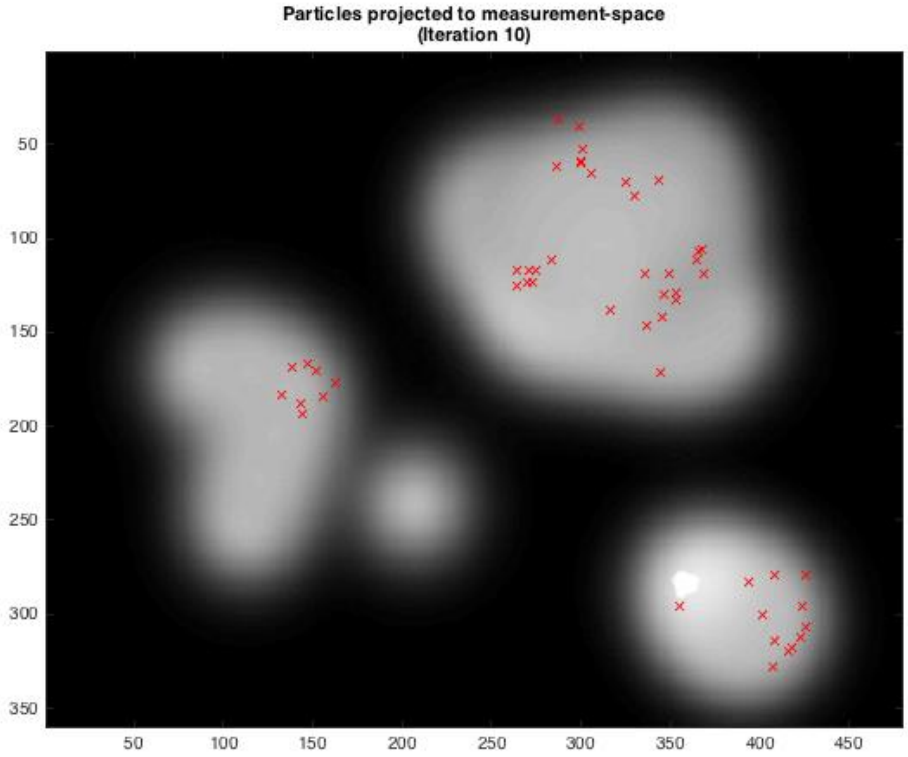
The first image illustrates the sampling method. Where we use a cumulative histogram and thresholds to see which particles should propagate. The size of the circles represents more likely particles, which would have been resampled more frequently.



The next images illustrate the journey of the particles by showing their locations at various iterations, for different times that I ran the code, obviously the 2nd iteration has a few misses, but at iteration 10, there are no particles located at low intensity pixels.



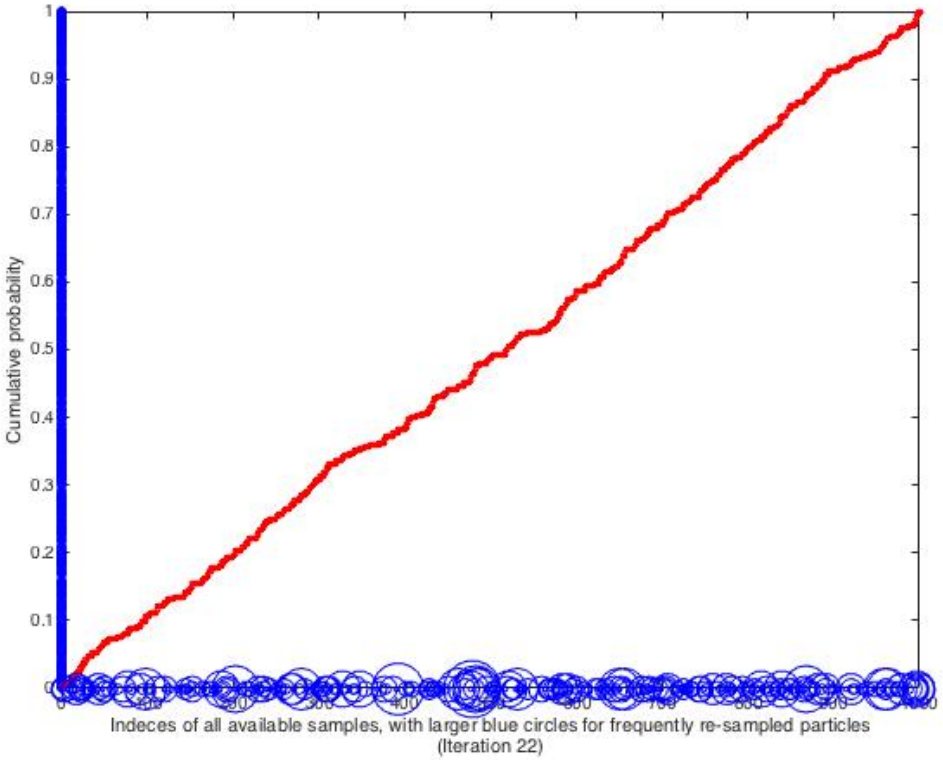
After iteration 10 we can see the particles have tracked to 3 more likely locations. This shows an advantage compared to say a Kalman filter. However, reaching a multi-modal distribution is not a given. And given the inherent randomness we sometimes end up with particles in one location.



F) Condensation – Practical 9b

Practical 9b is almost identical to the previous practical in terms of the TO DO's. Indeed the code we have to implement is identical. The key difference is that rather than repeating the same image we are now tracking through multiple frames and use the provided MeasurePatchSimilarity function for our likelihood.

The histogram (same description as above) and the 10th and 22nd iteration of the particle tracking is provided below.



Particle-tracking the silver car -Frame 10.

Frame 22, particle filter tracking of the silver car.

G) Combining tracking and Homographies – Practical 9c

Be careful with the directory structure for the code for Practical 9c and tracking and homographies. The directory this should be run from is HW2/TrackHomog/.

This practical takes what we have used in F, and we apply to a different video. Much of the initialisation and code is the same and the noise model is as follows.

```
noise = 10*randn(1,2);
particles_new(particleNum,:) = particles_old(…
samples_to_propagate(particleNum),: ) + noise;
```

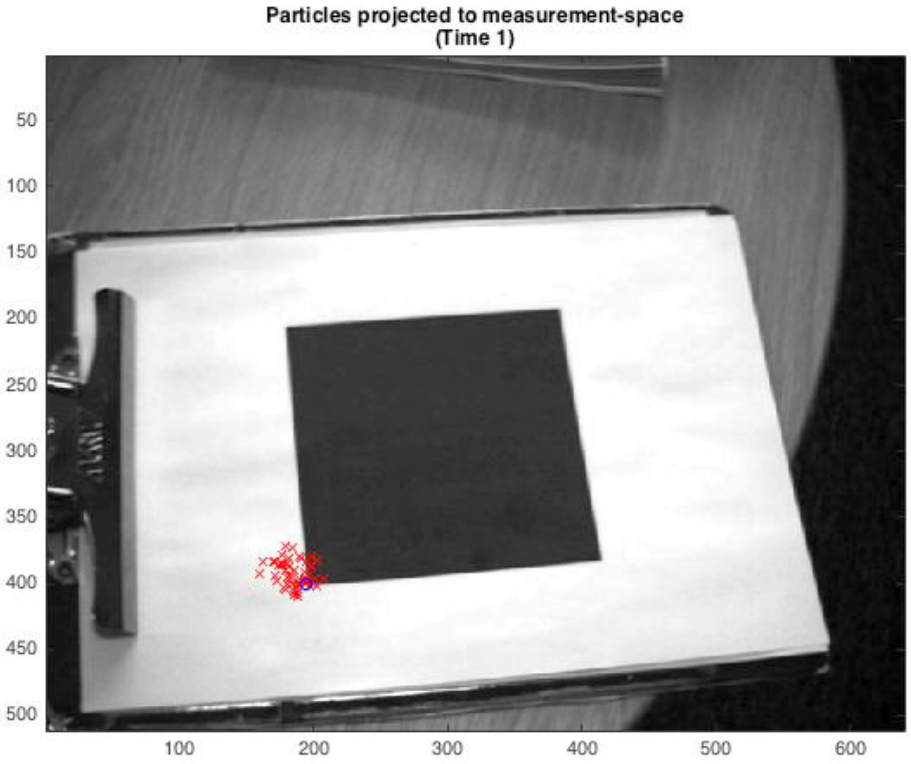
This practical is mainly an application of what we have learned already although we are now asked to calculate the MAP location for our tracker, which is the weighted average of all the particles. This is shown as the blue circle in the video and the code I used below.

```
weightedAve = [0 0];
weightedAve = sum(particles_new.*repmat(weight_of_samples,1,2));
middleOfTrackedPatch = weightedAve + patchOffset;
```

I chose to save each of the .png's and create an avi to illustrate the effect of the tracking. The video below can be played by right-clicking on the image.

What can be seen are the red particles attempting to track the bottom right corner of the black square. The blue circle our MAP location for our tracker. Ironically, this particular video is one of the poorer runs (many others were somewhat more accurate) as you can see the particle filter loses the location of the corner for a period of time.

One point to note is that the code returns the MAP coordinates through time. This is used in the next function.



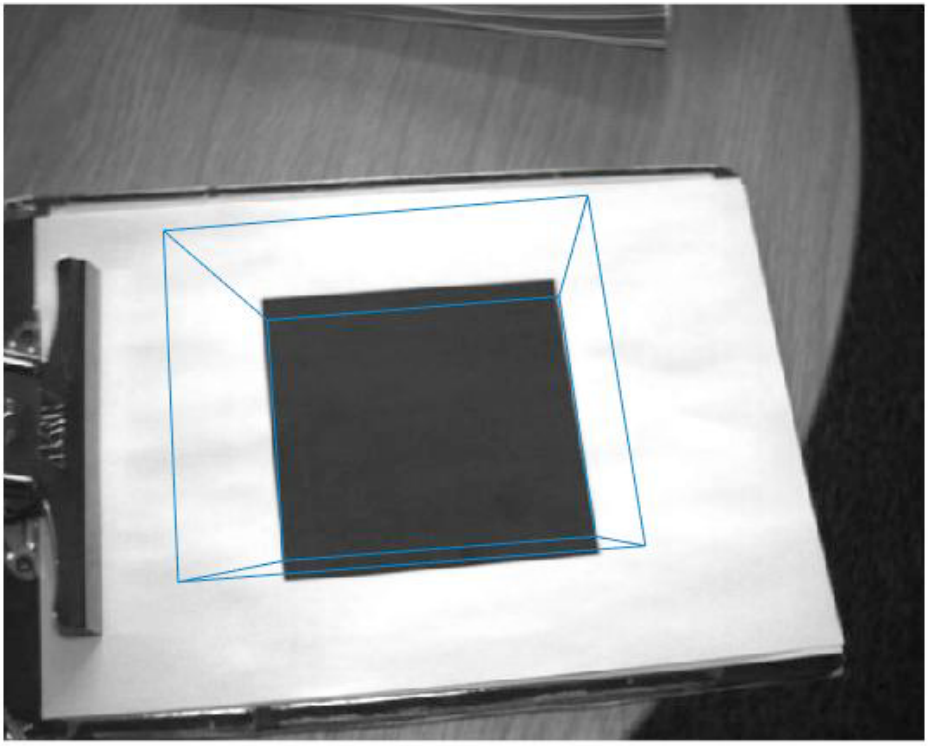
H) Combining tracking and Homographies

Whereas in the previous assignment we tracked the bottom left corner of the square in our video. In this assignment we call our function going through each corner of the square in turn. This will return each of the 4 corners MAP location for the particle filter.

Given the estimated intrinsic matrix of the footage, and the estimate world points of the 4 points of the square, we can use our 4 MAP tracked particle filter locations in image coordinates to estimate the extrinsic matrix relating the plane position to the camera position. To do this I use my function estimatePlanPose illustrated earlier. Clearly this is done frame by frame.

Now we have the extrinsic matrix we can use our world coordinates for a 3d-wireframe and use the projective camera function to translate to image coordinates. Thus giving an image wireframe through time.

As before the image below is a video I created and can be right clicked to see the results.



As can be seen from the video the cube is mostly mapped correctly but sometimes not at all.

A couple of quick areas I would explore would be the following.

First the logic of what we are doing…
We are using 4 points and mapping particle filters to each of those points – we then find the MAP location and map our homography.

Thus within our implementation there is a large amount of structure in the image that is NOT being utilised by our implementation and could certainly be used for error correction.

a) Geometric Structure – points on lines in our world space will be mapped to points on lines in our image space. We are not using the structure of our problem at all in this respect. Each corner is run independently. And **whatever** the result we map our wire-frame. Easiest would be to attempt post-processing to reduce errors. Although there will be skew in our image we can look at consistency of lines and angles for our skewed square and mapped cube before displaying.
b) Likelihood function and where particles are mapped to – We can either have an absolute restriction on where particles are mapped, again using the structure of the image, or embed more structure in the likelihood function. For example any square (or round) patch in the middle or on the outside of the image will be all black or all white, near the edges or corners the average intensity will vary massively. This is easy in terms of restricting to all edges. But if one goes further and shifts our patch along the gradient of greatest change in pixel intensity then we can also differentiate between edges and corners.
c) More data – If I could also track more points then I can reduce the impact of the particle filter 'losing' one corner.
d) No doubt many, many more, but these seemed reasonably simple!

Code Structure:
The code is self contained in 3 main folders, 07_Practical_Homographies, 09_PracticalCondensation and TrackHomog. All within HW2. The images are within the respective practical's folder as are the .avi's.