6-DOF Position from Monocular Image

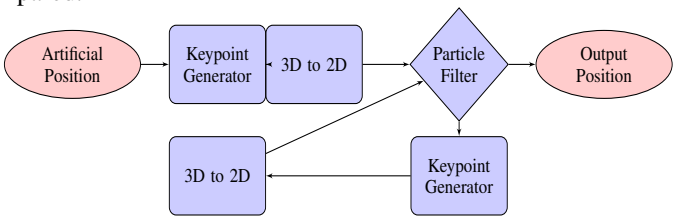
Kristina Collins, kvc2@case.edu; Connor Wolfe, cbw36@case.edu

***Abstract*—This report presents a MATLAB implementation of a particle filter, intended as a means of deducing a 6 DOF position for an object of known dimension based on a monocular camera image. The object used for demonstration is a solved Rubik's cube. The paper includes a high level discussion of implementation, a detailed summary of the particle filter, and a discussion of computer vision techniques specific to this problem. The code for the particle filter is appended.**

I. Introduction

Stereo vision, with the use of perspective transforms, can be used to identify objects' relative position with robustness and accuracy. However, this and LIDAR-based methods such as the Xbox Kinect can be cost-prohibitive and require repeated calibration. For a single monocular camera, however, an estimate of position can be obtained for an object of known dimensions. Our work was based around a solved 2x2 Rubik's cube, but could be extended to any set of 2D keypoints generated for an object of known 3D dimension.

The following flow chart shows the testing method used in this paper. An artificial position is generated for the cube in the form of a 6x6 matrix. This is fed into the keypoint generator node, which determines the 3D coordinates of segmentation points used for the cube. The points in question may be any points relative to the cube; for our testing, the corner points were used. These are converted from 3D to 2D, using the camera parameters, and the 2D points are fed into the particle filter. The particle filter does all its work in 2D: it compares the keypoints of the ground state to keypoints for each position it generates, and returns an output position. The artificially generated "true" position and the output position can then be compared.



II. Prior Art

In (1), particle filters were used to produce a robust position estimate of IKEA cups using edge-based segmentation, the results of which are available as an OpenCV library. (2) demonstrates the use of particle filters to obtain position estimates of a geometrically simple object (a tea box) and a visually complex one (car door). This project was informed by the work in both papers.

III. Particle Filter

Particle filters are used to estimate non-Gaussian, nonlinear processes. (3) They do so by representing the N guesses of the posterior density function $p(X_t|Z_{1:t})$, each guess referred to as a particle $X_t = [X_t^1, ...X_t^N]$. The particles are then assigned weights $\pi_t = [\pi_t^1 ...\pi_t^N]$ which represents the likelihood function $p(Z_t|X_t$ for the corresponding particle. With these variables, we can now describe the three step process of particle filtering.

Next, in the measurement model, we compare the guess from the measurement model to a sample input (an image in our case). Based on the accuracy of the guess, we assign a weight to each particle, representing how strongly the particle represents the true state. In the resampling step, we chose N particles with repetition based on their weights. This ensures that more likely states will be more densely represented than less likely states among the particles, and the particles will converge to the true state.

A. Motion Model

In the motion model, the next state of every particle is guessed based on the previous state, its action and random noise added to allow variance from the previous state's model.

State Representation: Our particles represent 6-DOF poses of the tracked object. Each $X_t^i \epsilon$ SE(3) represents the 4x4 transform matrix of the object. This 4x4 matrix contains in its upper left quadrant the 3x3 rotation matrix, in its upper right quadrant the 3x1 translation vector, in the lower left quadrant a 3x1 0-value vector, and in the lower right quadrant the value 1. This state representation is useful for representing robot kinematics because it accounts for the translation and rotation of the body, and allows for simple multiplication of matrices to describe motion.

State Equations: In order to propagate to X_t, we first attempted to implement the motion model from (1), but found more success using random motion models.

Choi and Christenson Approach: The methodology from (1) calculates the nonlinear propagation function A_{t-1} and adds to it some gaussian noise, dW, to calculate:

$$X_t = X_{t-1} * exp(A_{t-1}\Delta t + dW_t\sqrt{\Delta t} \tag{1}$$

$$dW_t = \sum_{i=1}^{6} \epsilon_{t,i} E_i \tag{2}$$

$$A_{t-1} = \lambda_{ar} * log(X_{t-2}^{-1} X_{t-1} \tag{3}$$

Such that E is the basis elements of the se(3) group. We randomly sample ϵ_t in order to generate the Brownian noise term to add to the nonlinear mapping A.

Random Approach: Generating motion with simply random deviations from the current state works better than one would imagine. The particle filter is able to filter out bad random motions and keep the good ones in the later steps. In order to implement a random motion model, we propagated the 4x4 state matrix for each particle following a Gaussian distribution.

B. Measurement Model

In the measurement model, the newly generated states $[X_t^{(1)}$ are compared against a measurement Z_t. The particles are assigned weights according to the likelihood function $p(Z_t|X_t^{(n)}$. This project demonstrates the use of keypoint detection, therefore the likelihood function represents the error between the sample's keypoints and the object's keypoints. This paper uses the edges of the cube as keypoints. We calculate the edge positions in 2D for every particle and the object. We then check the error for every sample using the sum of the Euclidean norm of all keypoints.
It is important to realize that we want to assign large weights to the particles with low errors. We implement two models to determine the weight. First, following from (1) we implement the weight as

$$P(Z_t|X_t^{(n)}) = e^{-\lambda\bar{e}} \tag{4}$$

We found more success, however, by using

$$P(Z_t|X_t^{(n)}) = \frac{1}{e^3} \tag{5}$$

This model makes sense because as the error decreases, the probability should increase. We increased this effect by cubing the error term in the denominator.
The last step is to normalize the array of weights:

$$\bar{\pi}^{(i)} = \frac{\pi^{(i)}}{\sum_{j=1}^{N} \pi^j} \tag{6}$$

C. Resampling

Lastly, we resample with replacement corresponding to the weights calculated above. This allows the particle filter to converge on more likely particles and eliminate poor particles. In matlab, the implementation is simple, we input indices from 1 to N= the number of particles, and we generate a weighted sample with replacement, following the weighting scheme π. We implement as follows:

```
ind=1:N;
ind=randsample(ind, N, true, pi);
```

We now redraw the particles corresponding to the indices in 'ind.'

D. Initialization

We employed a random initialization in order to demonstrate our particle filter's ability to convgere even from a randomly generated samples. To initialize,, we created N particles with random rotation and translation matrices restricted the state space that the object could reasonably occupy. We then weighted and resampled these particles as outlined above, and sent the resampled particles into the particle filter.

Fig. 1. Checkerboard camera calibration pattern. This and several photos of the checkerboard from various angles were used to derive calibration parameters for the webcam we used.

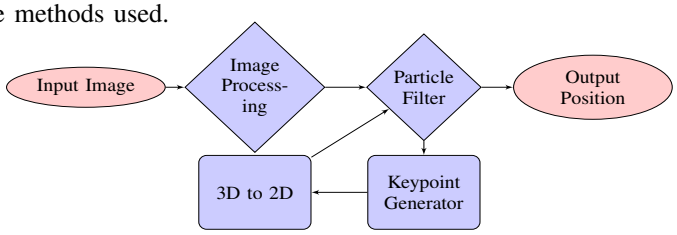
E. State Approximation

Thus far this algorithm has described how to sample particles that approximate the posterior density function, but we have not yet described how the algorithm will converge the particles into 1 state approximation. We will employ a weighted particle mean to estimate, shown in equation (7).

$$\bar{X}_t = \sum_{n=1}^{N} \pi_t^{(n)} * X_t^{(n)} \tag{7}$$

IV. Computer Vision

This flow chart describes the integration of computer vision with the particle filter. The input image is processed to segment out keypoints, which are then fed into the particle filter. This integration is not complete, but this section describes some of the methods used.



A. Webcam Calibration

The camera used was an HP webcam, calibrated with the MATLAB camera calibration toolbox with images like the one in Figure 1. The MATLAB command **worldToImage()** translates a point in 3D to its 2D projection in the camera's view, using parameters defined during calibration. The particle filter compares generated particles to the keypoints registered in this 2D space.

B. Rubik's Cube Coordinate System

The chosen demonstration object was a solved 2x2 Rubik's cube, 44 mm to a side. (Since the relative placement of colors is standard to all Rubik's brand cubes, this may be extended to

Rubik's cubes of larger size.) The cube has several advantages for computer vision:

1) As a cube, it's easily measured and projected. Each square on the cube is 22 mm wide, the same size as a printed copy of the checkerboard pattern MATLAB uses for camera calibration, so the checkerboard generator can be used to generate artificial keypoints for the cube.
2) Its faces are color-coded in saturated colors, and the black edges of the squares may be used (as is the case here) as built-in fiducial marks or keypoints. The colors are also standardized across different sizes and numbers
3) It's inexpensive and easily obtained by anyone seeking to replicate an imaging project using it.

We defined the X-axis along the red/white interface, the y-axis along the white/blue interface, and the Z-axis along the blue/red interface. This places the cube's logo adjacent to the origin in the XY plane. The table below lists the colors corresponding to the XYZ coordinate for each vertex in the cube's reference frame.

XYZ Coordinates	Colors
000	Blue/Red/White
001	Blue/Red/Yellow
010	Blue/Orange/White
011	Blue/Orange/Yellow
100	Green/Red/White
101	Green/Red/Yellow
110	Green/Orange/White
111	Green/Orange/Yellow

The camera's reference frame is such that a vector pointing out of the camera lens is considered to be pointing downwards in Z.

C. Keypoint Generation

A function, **generateKeypoints()**, was written to convert a 3D position matrix to a set of 3D keypoints. The function can return the points either in the cube coordinate frame or the camera coordinate frame, and can provide either all keypoints or only visible ones. [1]

Without referencing the colors of the cube, there's no way to determine which face(s) are being shown; therefore, we may assume the camera is looking "through" the cube, towards the origin, then check the colors of the visible faces to determine which rotation matrix should be applied to obtain the correct state. This particular issue can be extrapolated to other rotationally symmetric shapes - a pen, for example - and is most easily and robustly addressed by the addition of fiducial marks to the object in question.

D. Computer Vision

In practical execution, the most logical keypoints for the Rubik's cube are the black lines on the edges and faces, which are easy to segment out in a wide range of lighting conditions.

[1]The occlusion test: If the dot product of the vector for a given keypoint and the camera vector is greater than 0, the vectors are pointing in the same direction, and the keypoint is assumed to be invisible.

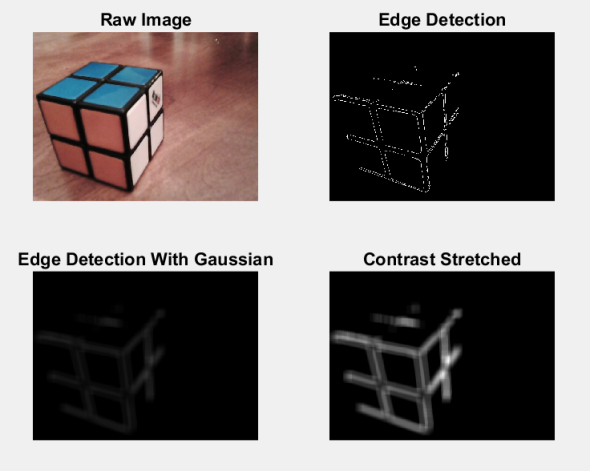


Fig. 2. Demonstration of edge segmentation.

An image of the cube is taken with the webcam, producing a 480x 640 RGB image. In subsequent steps, shown in the code snippet below, the image is binarized and edge detection is used. The resulting image is blurred, then contrast stretched (normalized). Each pixel's value, between 0 and 1, then serves as an indication of proximity to a keypoint. As in Figure 2, the segmentation won't necessarily catch all the keypoints, and some noise is expected from other edges in the image, but tuning of the edge detection parameters and Gaussian filter, as well as careful lighting of the cube, can help ameliorate that.

```
cam=webcam('HP') %Turns on HP USB camera
H = fspecial('gaussian',20,25); %declare
    Gaussian filter
img=snapshot(cam);
I=edge(im2bw(img, .1)); %Edge detection
gaussPic=zeros(size(I, 1), size(I, 2), 3);
gaussPic(:,:,1)=I; gaussPic(:,:,2)=I; gaussPic
    (:,:,3)=I;
gaussPic = imfilter(gaussPic,H,'replicate'); %
    Gaussian filter
stretched=gaussPic.*(1/(max(max(gaussPic)))); %
    contrast stretch
```

As noted above, segmentation of the cube's edges fails to take rotation into account, making an additional color segmentation step necessary. This is best undertaken in HSV space, rather than RGB. An example of color segmentation for orange is shown in Figure 3.

V. Testing & Results

We first test the particle filter on artificial data in which the object moves in a random fashion following a gaussian distribution at every step. We could not properly implement the motion model from (1) as the multiplication of the last state by the motion model (A*dW) led to inaccurate next states. Instead, the data below shows 100 iterations of the particle filter on simulated data. We generate 50 particles, all drawn randomly within the state space as shown in Figure 4. We then calculate their keypoints as well as the keypoints of the object.

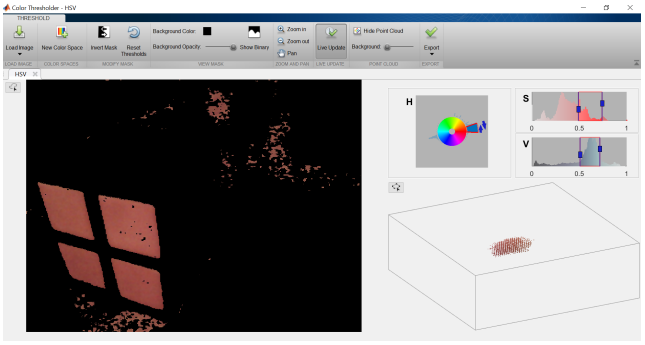


Fig. 3. Demonstration of color segmentation in HSV, using MATLAB's Color Thresholder tool on the raw image from Figure 2.

Next, we compute the weighting vector π and resample and calculate the mean of the particles to compute the guess after initialization. This is shown in Figure5. We then iterate through the particle filter 100 times, and the results of the filtering are shown in Figures 6, 7, and 8. Figure 6 shows the weighted guess after 10 iterations, which is a clear improvement over the initialization. We then see the guess after 100 iterations in Figure 7 and all 50 particles after 100 iterations in 8. These last two figures show a robust job of tracking the object, as the particles well-capture the object's keypoints.

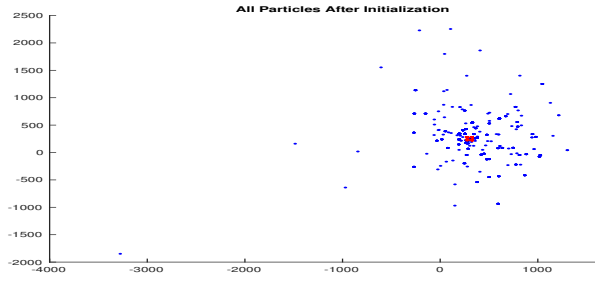


Fig. 4. Plot of edge keypoints of all particles upon random initiali blue, and the ojbect in red. Note their poor representation of the ot to their random initialization.

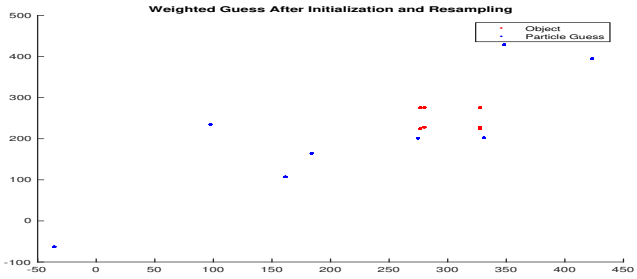


Fig. 5. Plot of edge keypoints of weighted guess of particles after random initialization and resampling. Note these keypoints better capture the object than before resampling and weighting as in Figure 4, but still the particles need to propagate a few times to find the object.

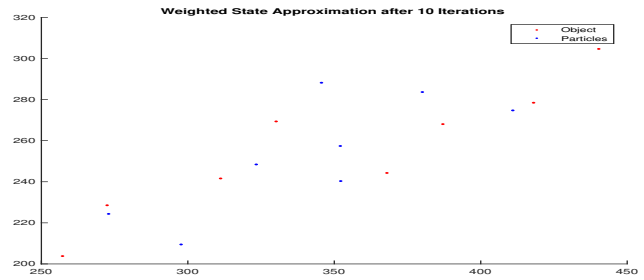


Fig. 6. The weighted guess of the object's state after 10 iterations The guess is far more accurate than after initialization which is promising after only 10 iterations

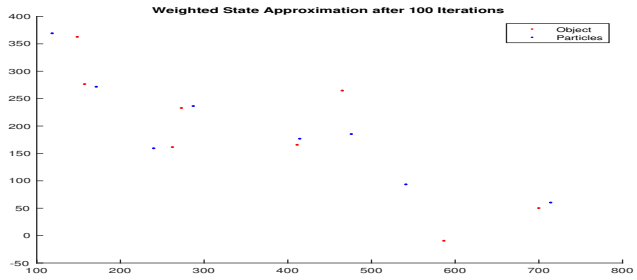


Fig. 7. The weighted guess after 100 iterations. There is clear improvement over 10 iterations, but it seems there is room for improvement with a more effective motion model.

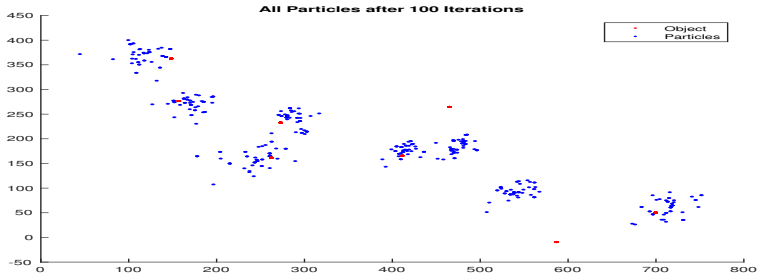


Fig. 8. Plot of edge keypoints of all particles after 100 iterations. They seem to cluster well around each keypoint of the object

VI. Conclusion

We are proud to report that we successfully tracked the artificial object with our particle filter, but see room to make the particle filter more robust in a few areas. Most importantly, we believe that employing a more robust motion model would greatly improve the filter. We see that we can quickly begin to localize around the particle filter, as shown by the improvement between Figures 5 and 6. These show that our particle filter is able to move towards the general space of the object quickly, which is important. However, we struggle to capture the smaller movements of the object and cannot localize much further than we did after 10 iterations by iteration 100. We believe this is because our motion model is random and therefore

cannot do much better than a rough estimate no matter how many iterations we allow (we tested up to 1000). We would like to continue this project by improving the keypoint generation functionality, fully integrating the imaging, and developing a motion model which captures the movement of the object well and therefore can make more accurate tracking as it approaches the object in later iterations.

VII. Acknowledgment

The authors gratefully acknowledge Professor Cavusoglu's guidance and patience, and Big Fun for providing Rubik's cubes.

References

[1] C. Choi and H. I. Christensen, "3d textureless object detection and tracking: An edge-based approach," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2012, pp. 3877–3884.

[2] ——, "Robust 3d visual tracking using particle filtering on the se(3) group," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 4384–4390.

[3] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents series)*. The MIT Press, 2005.

.

```
% MATLAB Script for Monocular Vision
% Connor Wolfe and Kristina Collins, EECS 499, Spring 2017

%% SECTION I: CAMERA AND OBJECT ORIENTATION
N=50; %Number of particles


%% Step 1: Set webcam calibration parameters
load('webcamParams.mat'); %loads calibration data for HP webcam
as the variable cameraParams
camR=[1 0 0; 0 1 0; 0 0 1]; %no rotation from camera to, well,
camera
camT=[0;0;0]; %no translation from camera to camera

%% Step 2: Initialize Artificial State of Cube
% First, input artificial state of cube as 4x4 position matrix
at t=0
objR=[1 0 0; 0 1 0; 0 0 1]; %Simulated cube rotation at t=0, in
mm with camera facing
objT=[-22 -22 300]; % Simulated translation at t=0
objP=rt2dof(objR, objT); %converts to 4x4 position matrix

%Then, initialize keypoints of cube at t=0
obj_3D_keypoints=generateKeypoints(objP, 'camera', 'corners');
obj_2D_keypoints = worldToImage(cameraParams,camR,
camT,obj_3D_keypoints);
num_keypoints=size(obj_2D_keypoints, 1);

%% SECTION II: INITIALIZE PARTICLES
%% Step 1: Initializa N=5 Particles
%Initialize at a normal distribution away from input object
state
particles=cell(1,N);
for n=1:N
    state_t=zeros(4,4); state_t(4,4)=1;
    for i=1:3
        for j=1:4
            state_t(i,j)=normrnd(objP(i,j),20);
        end
    end
    particles{n}=state_t;
end

%% Step 2: Generate keypoints of initial particles
keypoints_3D=cell(1,N);
keypoints_2D=cell(1,N);
for n=1:N
```

```
    keypoints_3D{n}=generateKeypoints(particles{n}, 'camera',
'corners');
    keypoints_2D{n}=worldToImage(cameraParams,camR,
camT,keypoints_3D{n});
end

%% Step 3: Weight particles based on distance to image
pi=zeros(1,N);
for n=1:N
    err=0;
    for i=1:num_keypoints
        err=err + sqrt((obj_2D_keypoints(i,1)-
keypoints_2D{n}(i,1))^2 + (obj_2D_keypoints(i,2)-
keypoints_2D{n}(i,2))^2);
    end
    %pi(n)=exp(err*lambda_e* -1);
    pi(n)=(1/err).^5;
end
%Normalize P
denom=sum(pi);
pi(:)=pi(:)/denom;


%% Step 4: Resample particles based on weights and Plot

ind=1:N;
ind=randsample(ind, N, true, pi);

%Resample following indices above
particles_uf=particles;
keypoints_2D_uf=keypoints_2D;
for n=1:N
    particles{n}=particles_uf{ind(n)};
    keypoints_2D{n}=keypoints_2D_uf{ind(n)};
end

keypoint_avg=zeros(4,4);
for i=1:8
    for j=1:2
        cur_point=0;
        for n=1:N
            cur_point=cur_point+keypoints_2D{n}(i,j);
        end
        keypoint_avg(i,j)=(1/N)*cur_point;
    end
end
%And Plot
```

```
for n=1:N
    figure(1)
    hold on
    scatter(obj_2D_keypoints(:,1),obj_2D_keypoints(:,2), 60,
'r.');
    scatter(keypoint_avg(:,1),keypoint_avg(:,2), 60, 'b.');
    title('Weighted Guess After Initialization and Resampling');
    legend('Object', 'Particle Guess');
    hold off
end

%% SECTION III: PARTICLE FILTER
for t=2:1000
    % Step 1: Move the artificial object via normal distribution
and get KP
    for i=1:3
        for j=1:4
            objP(i,j)=normrnd(objP(i,j),0.1);
        end
    end
    obj_3D_keypoints=generateKeypoints(objP, 'camera',
'corners');
    obj_2D_keypoints = worldToImage(cameraParams,camR,
camT,obj_3D_keypoints);


    % Step 2: Motion model
    for n=1:N
        for i=1:3
            for j=1:4

particles{n}(i,j)=normrnd( particles{n}(i,j),20);
            end
        end
    end

    % Step 3: Measurement Model
    % Step 3.a: Get keypoints of each particle
    for n=1:N
        keypoints_3D{n}=generateKeypoints(particles{n},
'camera', 'corners');
        keypoints_2D{n}=worldToImage(cameraParams,camR,
camT,keypoints_3D{n});
    end

    % Step 3.b: Calculate weights pi via error of image KP to
particle KP
```

```
    for n=1:N
        err=0;
        for i=1:num_keypoints
            err=err + sqrt((obj_2D_keypoints(i,1)-
keypoints_2D{n}(i,1))^2 + (obj_2D_keypoints(i,2)-
keypoints_2D{n}(i,2))^2);
        end
        pi(n)=(1/err).^5;
    end

     % Step 3.c: Normalize pi
    denom=sum(pi);
    pi(:)=pi(:)/denom;

    % Step 4: Resampling
    %Get indices of which weighted selection of X's w
replacement we send to next iteration
    ind=1:N;
    ind=randsample(ind, N, true, pi);

    %Redraw particles and keypoints based on indices above
    keypoints_2D_uf=keypoints_2D;
    particles_uf=particles;
    for n=1:N
        particles{n}=particles_uf{ind(n)};
        keypoints_2D{n}=keypoints_2D_uf{ind(n)};
    end

    % Step 5: Calculate mean of the particles to guess from this
iteration
    keypoint_avg=zeros(4,4);
    for i=1:8
        for j=1:2
            cur_point=0;
            for n=1:N
                cur_point=cur_point+keypoints_2D{n}(i,j);
            end
            keypoint_avg(i,j)=(1/N)*cur_point;
        end
    end

    % Step 6: Plot the guess (plot using particles{1:N} for all
particles
    %for n=1:N
    if mod(t,100)==0
        figure(t)
        hold on
```

```
            scatter(obj_2D_keypoints(:,1),obj_2D_keypoints(:,2), 60,
'r.');
            scatter(keypoint_avg(:,1),keypoint_avg(:,2), 60, 'b.');
            title(['pi = ' pi(n)]);
            hold off
        %end
        end

end
```

```
function allKeypoints = generateKeypoints(position, frame,
keypoints)
    %Take in 4x4 position matrix of particle in camera
frame, generate keypoints in 3D space. The "frame" command
indicates whether to return coordinates in cube or camera
frame; "keypoints" command indicates which set of keypoints
to generate. This revision of the function only works for
the frame ‘camera’ and the keypoint set ‘corners’.
    %origin=position(1:3, 4); %translation vector in camera
frame

    %Declare normal vectors:
    Normals=[0 0 -1; 0 -1 0; -1 0 0; 0 0 1; 1 0 0; 0 1 0];

    switch(keypoints)
        case('corners')
            cubeFrameKeypoints=22*[0 0 0; 0 0 1; 0 1 0; 0 1
1; 1 0 0; 1 0 1; 1 1 0; 1 1 1];
        case('edges')
            %TODO: change points
            cubeFrameKeypoints=22*[0 0 0; 0 0 1; 0 1 0; 0 1
1; 1 0 0; 1 0 1; 1 1 0; 1 1 1];
        case('faces') %finds the center of each blob of
color
            %TODO: change points
            cubeFrameKeypoints=22*[0 0 0; 0 0 1; 0 1 0; 0 1
1; 1 0 0; 1 0 1; 1 1 0; 1 1 1];
        case('centers')
```

```
            cubeFrameKeypoints=22*[.5 .5 0; .5 0 .5; 0 .5
0; 1 .5 .5; .5 1 .5; .5 .5 1];
    end

transpose(cubeFrameKeypoints(i,:)*position(1:3,1:3) +
transpose(position(1:3, 4)));
    end
    %Return 3D points of all corners
    switch(frame)
        case ('cube')
            allKeypoints=cubeFrameKeypoints;
        case('camera')
            allKeypoints=cameraFrameKeypoints;
        otherwise
            allKeypoints=cameraFrameKeypoints;
    end
end

```