# [420] Assignment 4 (on Windows)

Tanuj Kumar 1002197133 ta.kumar@mail.utoronto.ca

November 2017

## 1 Question 1

### 1.1 1. Brainstorming

A ballperson's role can be divided into these steps:

1. Stand on the side of the court, by the side lines
2. Keep watch of who is serving, where the referee is, where the sidelines are, and where the ball is
3. When the ball passes the side line, it is out and will hit the side net, so the ballperson must retrieve it
4. The ballperson will look at where the ball landed by the net, identify it, move towards it, grab it, and take it back to the server.
5. If the ball is lost, the ballperson will have to retrieve a new ball. Also, in a worst-case scenario or a specific foul set, the ballperson may have to give the ball back to the referee themselves.
6. When play continues, the ballperson must repeat this.

From here, we know that the robot must autonomously do all of the above tasks. The first basis is that the robot must cognitively identify three major objects: the two players, so the robot can quickly identify which one to return the ball to, and the ball itself, differentiated accurately among other image data. Because this is the most important part of the robot's processing, the highest reliability of these tasks can be achieved through object recognition algorithms that are themselves pre-trained. In the code below DPM is used but CNNs can easily be substituted for an even more accurate result. The robot's stereo vision also only receives 2D images as input, so these must be converted into 3D world projections to allow movement coordinates when facing particular directions or moving to locations. The robot must also continually update its knowledge of its surroundings in all directions, to ensure that the location of the ball is not missed in any direction it can see, and so this requires panorama-stitching using SIFT and RANSAC. This also provides a foolproof if future safety developments need to be added to the robot, such as avoiding being hit by balls.

## 1.2 2. Processing Pipeline

1. **Initialize** initial positions of the robot (origin), referee locations, extra tennis balls, scores, sideline and side net locations, camera parameters, and service counter.

2. **Train** a DPM detector on training data including player one, player two, and the ball, in other to identify with high accuracy. For better accuracy a convolutional neural network can be used.

3. **Define** three helper functions: **returnToOrigin**, which encapsulates all the code that returns the robot from wherever it is to the origin, **serviceSwitch**, which keeps track of which player is the current server, and **panoramaCapture**, which using the stereo cameras captures a 360 panorama image of the robot's current surroundings using SIFT and RANSAC.

4. On each second iteration (time can be tuned):

   (a) Capture a panorama and use DPM Object detection to detect the ball, converting its coordinates to 3D space.

   (b) If the ball is passed the side line, the robot needs to go retrieve it.

      i. The robot will **faceDirection** towards the side net and use panoramaCapture to generate an image which it will then convert into a 3D world, and it will detect the ball location.

         A. If there are 0 or 2 or more detections there is a detection issue so the robot will assume the ball is lost and **faceDirection** and **moveToLocation** to the pre-entered extra tennis balls location coordinates to retrieve a new one.

         B. If there is just 1 detection, the ball is in that part of the net, so the robot will **moveToLocation** to the 3D coordinates captured above, and **grabObjectAtLocation** to grab the ball.

      ii. The robot will **faceDirection** towards the court and use DPM to detect the player currently registered in the server counter. It will call panoramaCapture again and generate a 3D map to identify the player's coordinates, and then **moveToLocation** to give the ball back to this server, before finally calling **returnToOrigin** to restart when play begins again.

         A. If detection of the server fails, the robot will simply **faceDirection** and **moveToLocation** of the pre-entered referee location in world coordinates and give the ball to them.

## 1.3 3. Pseudocode

The below pseudocode is an adoption of the processing pipeline described above. Here, the details are fleshed out with a number of different helper functions.

```
Initialization:

initialPosition = [0 0 0];
extraTennisBallsBucket % an input that denotes the location of where to
    find extra tennis balls if an error happens or the ball is loss, in
    world coordinates
refereeLoc % an input that denotes the world coordinates location of
    where the referee is
scoreP1 = 0
scoreP2 = 0
time = 0
sideLineLocation % an array input in world coordinates for the out of
    bounds line with (0,0,0) as the initial position of the robot,
sideNetLocation % an array input in world coordinates for the location
    of the out of bounds net with (0,0,0) as the initial position of
    the robot

DPMObject = trainDeformablePartModel(P1train, P2train, ballTrain)

service = input("Winner of coin toss, 0 if P1, 1 if P2")
calibMat = K (given)
t = cameraDifference % built-in based on difference in camera eyes for
    the robot stereo vision

--- Panorama Helper Function Creation to stitch a 3D Scene: ---

imgArray;
SIFTArray;
rotationArray;

% taking a photo every 10 degrees

for i = 1 to 36:
   im = takePhoto()
   imgArray(i) = im
   rotationArray(i) = faceDirection(rotate(10,right)) % rotate is a
       helper function that outputs the (X,Y,Z) coordinates relative to
       (0,0,0) of staying in place and rotating a certain number of
       degrees
   SIFTArray(i) = sift(im)

% panorama stitching using RANSAC

panoramaBuildArray = imgArray(1);
3DStitch = computeWorldCoordinates(imgArray(1),K,rotationArray(i),t) %
    returns X Y Z from image x y with depth Z from K R t stereo
    information

for i = 2 to 36:
```

```
   featMatches = RANSAC(SIFTArray(i-1), SIFTArray(i)) % use RANSAC to
       get matched SIFT features
   stitch(panoramaBuildArray,imgArray(i),featMatches) % stitch the next
       image to the current built up panorama image

return panoramaCapture

---

--- Service Switch Function ---

if service switches, just switch the service variable to the other player

---

--- Return to Origin Function ---

   faceDirection(0,0,0)
   moveToLocation(0,0,0)

---

pano;
3Dworld;

every second update:
   serviceSwitch()
   pano = panoramaCapture()
   3Dworld = computeWorldCoordinates(pano,K,rotationArray,t) % returns X
       Y Z from image x y with depth Z from K R t stereo information
   [x,y] = DPMObject.detect(pano,ball)
   3DBallLoc = computeWorldCoordinates([x,y],K,0,t)
   if 3DBallLoc is past the sideLineLocation:
      faceDirection(sideNetLocation)
      im = takePhoto()
      currentOrientation = self.rotate;
      detections = DPMObject.detect(im,ball) % detecting where the ball
          is on the net
      if size(detections,1) != 1:
         faceDirection(extraTennisBallBucket)
         moveToLocation(extraTennisBallBucket)
         grabObjectAtLocation(extraTennisBallBucket)
      else:
         NetBallLoc =
             computeWorldCoordinates([x,y],K,currentOrientation,t)
         moveToLocation(NetBallLoc)
         grabObjectAtLocation(NetBallLoc)
      pano = panoramaCapture()
      servDetector = DPMObject.detect(pano,service)
      if servDetector != 1:
```

```
        faceDirection(refereeLoc)
        moveToLocation(refereeLoc)
        giveObject(refereeLoc)
    else:
        3DServerLoc =
            computeWorldCoordinates(servDetector,K,currentOrientation,t)
        faceDirection(3DServerLoc)
        moveToLocation(3DServerLoc)
        giveObject(3DServerLoc)
    returnToOrigin()
```

# 2 Question 2

## 2.1 2a

```
function depth = depth(image, imset)
    cameraParams = getData(image, imset, 'calib');
    disp = getDisparity(imset, image); % returns the n x m disparity mat
    focal = cameraParams.f;
    base = cameraParams.baseline;

    dispInv = 1./disp;
    numerator = focal * base;
    depth = numerator.*dispInv;
end
```

```
images = getData([],'test','list');
ids = images.ids(1:3);

depthImg1 = depth(ids{1},'test');
depthImg2 = depth(ids{2},'test');
depthImg3 = depth(ids{3},'test');

infs1 = find(depthImg1==Inf);
infs2 = find(depthImg2==Inf);
infs3 = find(depthImg3==Inf);

% prevent infinities
depthImg1(infs1) = max(max(depthImg1))+5;
depthImg2(infs2) = max(max(depthImg2))+5;
depthImg3(infs3) = max(max(depthImg3))+5;

% image(depthImg3) shows depth visualization
```

The depth of a pixel is defined with the following formula:

$$Z = \frac{f \cdot T}{x_r - x_l}$$

where the denominator is the disparity, $f$ is focal length, and $T$ is baseline. The depth visualizations are below.

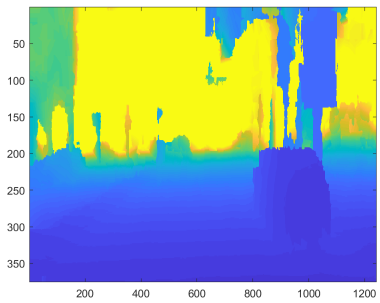


Figure 1: depth map of image 1

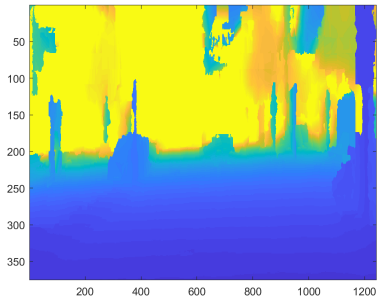


Figure 2: depth map of image 2

## 2.2 2b

```
function [ds, bs] = detector(imName, detectionType, resize, imThresh)

    data = getData([], [], detectionType);
    model = data.model;
    col = 'r';

    imdata = getData(imName, 'test', 'left');
    im = imdata.im;
```

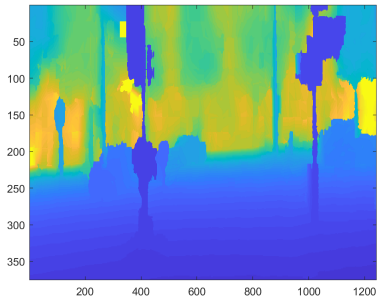


Figure 3: depth map of image 3

```
    imr = imresize(im,resize); % if we resize, it works better for small
        objects

    % detect objects
    fprintf('running the detector, may take a few seconds...\n');
    tic;
    %[ds, bs] = imgdetect(imr, model, model.thresh); % you may need to
        reduce the threshold if you want more detections
    [ds, bs] = imgdetect(imr, model, imThresh);
    e = toc;
    fprintf('finished! (took: %0.4f seconds)\n', e);
    top = nms(ds, 0.5);
    % if model.type == model_types.Grammar
    %   bs = [ds(:,1:4) bs];
    % end

    if ~isempty(ds)
        % resize back
        ds(:, 1:end-2) = ds(:, 1:end-2)/resize;
        bs(:, 1:end-2) = bs(:, 1:end-2)/resize;
    end

    showboxesMy(im, reduceboxes(model, bs(top,:)), col);
    fprintf('detections:\n');
    ds = ds(top, :);
    bs = bs(top, :);
end
```

---

```
% detection script
addpath('dpm') ;
addpath('devkit') ;

images = getData([],'test','list');
```

```
ids = images.ids(1:3);

[DS1_cars,BS1_cars] = detector(ids{1},'detector-car',1.5,-0.2);
[DS1_people,BS1_people] = detector(ids{1},'detector-person',1.5,-0.5);
[DS1_bikes,BS1_bikes] = detector(ids{1},'detector-bicycle',1.5,-0.9);

[DS2_cars,BS2_cars] = detector(ids{2},'detector-car',1.5,-0.4);
[DS2_people,BS2_people] = detector(ids{2},'detector-person',1.5,-0.65);
[DS2_bikes,BS2_bikes] = detector(ids{2},'detector-bicycle',1.5,-0.8);

[DS3_cars,BS3_cars] = detector(ids{3},'detector-car',1.5,0.0);
[DS3_people,BS3_people] = detector(ids{3},'detector-person',1.5,-0.5);
[DS3_bikes,BS3_bikes] = detector(ids{3},'detector-bicycle',1.5,-1.0);

save('ds1data.mat','DS1_cars','DS1_people','DS1_bikes');
save('ds2data.mat','DS2_cars','DS2_people','DS2_bikes');
save('ds3data.mat','DS3_cars','DS3_people','DS3_bikes');
```

## 2.3 2c

Visualization is provided below.

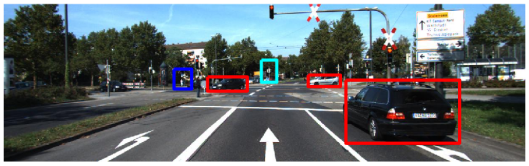

Figure 4: detections of image 1. red = cars, blue = people, cyan = bicycles. some false positives exist

## 2.4 2d

The 3D location of the center of mass was computed in two ways to try and figure out the most accurate segmentation.

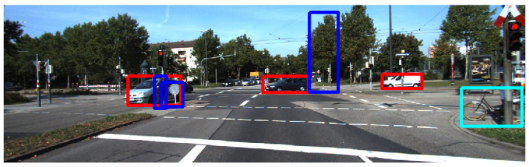

Figure 5: detections of image 2. red = cars, blue = people, cyan = bicycles. some false positives exist

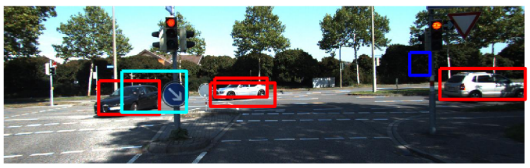

Figure 6: detections of image 3. red = cars, blue = people, cyan = bicycles. some false positives exist

- **Means over Components (CenterOfMass)**. The image pixels are converted to world coordinates, and then in each resulting cube represented within a box, the mean is computed by averaging each X, Y, Z component over the number of points present.
- **3D Midpoint (CenterOfMass2)**. The image pixels are converted to world

coordinates, and then in each resulting cube represented within a box, the X and Y midpoints are taken with the corresponding depth Z to get the center of mass.

In all cases the following formula is used to convert to world coordinates (seen here with the x component):

$$X = \frac{Z * (x - p_x)}{f}$$

For the segmentation, better results were received with the midpoint method so both code methods are provided as follows:

```
function [center, worldCubes] = centerOfMass(cameraCalib, imData,
    imDepth)
    % load f, px, py from cameraCalib

    % load the imData of the specific objClass as a matrix
    % note: objClass as a separate argument here may be unnecessary

    % iterate through each row of imData
    % the four box bound points:
    % > (x_l, y_t) (x_l, y_b) (x_r, y_t) (x_r, y_b)
    % > in indices of the row (0..5): (0,1) (0,3) (2,1) (2,3) % flip to
    % (y,x)
    % get the entries bound by these indices as a submatrix
    % > repmat vertical stack for x's, horizontal stack for y's
    % get the corresponding depth pixels from imDepth as a submatrix
    % elementwise subtract each entry by px, divide by f, multiply by
        depth
    % elementwise subtract each entry by py, divide by f, multiply by
        depth
    % will give world coordinates

    f = cameraCalib.f;
    k = cameraCalib.K;
    px = k(1,3);
    py = k(2,3);

    numRows = size(imData,1);

    center = zeros(numRows,3);
    worldCubes = {1,numRows};

    for i = 1:numRows
        box = imData(i,:);
        xL = round(box(1)); xR = round(box(3)); yT = round(box(2)); yB =
            round(box(4));
        if (xR >= size(imDepth,2))
            xR = xR - 4;
```

```
        end
        xRow = xL:xR; yCol = (yT:yB)';
        width = size(xRow,2); height = size(yCol,1);
        xMat = repmat(xRow,height,1); yMat = repmat(yCol,1,width);
        depthChunk = imDepth(yT:yB,xL:xR);
        numPts = size(depthChunk,1)*size(depthChunk,2);
        xMatpx = xMat - px;
        yMatpy = yMat - py;
        xMatd = xMatpx ./ f;
        yMatd = yMatpy ./ f;
        camX = depthChunk .* xMatd;
        camY = depthChunk .* yMatd;
        % need to do a translation into world coordinates!
        % then we find the mean of all (X,Y,Z) world points / num of pts
        worldCube = zeros(height, width, 3);
        worldCube(:,:,1) = camX; % X
        worldCube(:,:,2) = camY; % Y
        worldCube(:,:,3) = depthChunk; % Z

        % translation edit:
        [~,~,tl] = KRt_from_P(cameraCalib.P_left);
        worldCube(:,:,1) = worldCube(:,:,1) - tl(1);
        worldCube(:,:,2) = worldCube(:,:,2) - tl(2);
        worldCube(:,:,3) = worldCube(:,:,3) - tl(3);

        allX = sum(sum(worldCube(:,:,1)));
        allY = sum(sum(worldCube(:,:,2)));
        allZ = sum(sum(worldCube(:,:,3)));

        center(i,1:3) = [allX/numPts, allY/numPts, allZ/numPts];
        worldCubes{i} = worldCube;

    end

end
```

```
function [center, worldCubes] = centerOfMass2(cameraCalib, imData,
    imDepth)
    % load f, px, py from cameraCalib

    % load the imData of the specific objClass as a matrix
    % note: objClass as a separate argument here may be unnecessary

    % iterate through each row of imData
    % the four box bound points:
    % > (x_l, y_t) (x_l, y_b) (x_r, y_t) (x_r, y_b)
    % > in indices of the row (0..5): (0,1) (0,3) (2,1) (2,3) % flip to
    % (y,x)
    % get the entries bound by these indices as a submatrix
```

```
% > repmat vertical stack for x's, horizontal stack for y's
% get the corresponding depth pixels from imDepth as a submatrix
% elementwise subtract each entry by px, divide by f, multiply by
    depth
% elementwise subtract each entry by py, divide by f, multiply by
    depth
% will give world coordinates

f = cameraCalib.f;
k = cameraCalib.K;
px = k(1,3);
py = k(2,3);

numRows = size(imData,1);

center = zeros(numRows,3);
worldCubes = {1,numRows};

for i = 1:numRows
    box = imData(i,:);
    xL = round(box(1)); xR = round(box(3)); yT = round(box(2)); yB =
        round(box(4));
    if (xR >= size(imDepth,2))
        xR = xR - 4;
    end
    [~,~,tl] = KRt_from_P(cameraCalib.P_left);
    xRow = xL:xR; yCol = (yT:yB)';
    width = size(xRow,2); height = size(yCol,1);
    xMid = xL + round(width/2); yMid = yT + round(height/2);
    zMid = imDepth(yMid, xMid);

    xW = zMid * ((xMid - px) / f);
    yW = zMid * ((yMid - py) / f);
    zW = zMid;
    xw = xW - tl(1) ; yw = yW - tl(2); zw = zW - tl(3);
    center(i,:) = [xw yw zw];

    xMat = repmat(xRow,height,1); yMat = repmat(yCol,1,width);
    depthChunk = imDepth(yT:yB,xL:xR);

    xMatpx = xMat - px;
    yMatpy = yMat - py;
    xMatd = xMatpx ./ f;
    yMatd = yMatpy ./ f;
    camX = depthChunk .* xMatd;
    camY = depthChunk .* yMatd;
    % need to do a translation into world coordinates!
    % then we find the mean of all (X,Y,Z) world points / num of pts
    worldCube = zeros(height, width, 3);
    worldCube(:,:,1) = camX; % X
```

```
        worldCube(:,:,2) = camY; % Y
        worldCube(:,:,3) = depthChunk; % Z

        % translation edit:

        worldCube(:,:,1) = worldCube(:,:,1) - tl(1);
        worldCube(:,:,2) = worldCube(:,:,2) - tl(2);
        worldCube(:,:,3) = worldCube(:,:,3) - tl(3);



        worldCubes{i} = worldCube;
    end

end
```

---

```
% com calculation
images = getData([],'test','list');
ids = images.ids(1:3);

depthImg1 = depth(ids{1},'test');
depthImg2 = depth(ids{2},'test');
depthImg3 = depth(ids{3},'test');

infs1 = find(depthImg1==Inf);
infs2 = find(depthImg2==Inf);
infs3 = find(depthImg3==Inf);

depthImg1(infs1) = max(max(depthImg1))+5;
depthImg2(infs2) = max(max(depthImg2))+5;
depthImg3(infs3) = max(max(depthImg3))+5;

ccD1 = getData(ids{1},'test','calib');
ccD2 = getData(ids{2},'test','calib');
ccD3 = getData(ids{3},'test','calib');

load('ds1data.mat');
load('ds2data.mat');
load('ds3data.mat');

[D1centCar,D1Car] = centerOfMass2(ccD1, DS1_cars, depthImg1);
[D1centPpl,D1Ppl] = centerOfMass2(ccD1, DS1_people, depthImg1);
[D1centBike,D1Bike] = centerOfMass2(ccD1, DS1_bikes, depthImg1);

[D2centCar,D2Car] = centerOfMass2(ccD2, DS2_cars, depthImg2);
[D2centPpl,D2Ppl] = centerOfMass2(ccD2, DS2_people, depthImg2);
[D2centBike,D2Bike] = centerOfMass2(ccD2, DS2_bikes, depthImg2);

[D3centCar,D3Car] = centerOfMass2(ccD3, DS3_cars, depthImg3);
```

```
[D3centPpl,D3Ppl] = centerOfMass2(ccD3, DS3_people, depthImg3);
[D3centBike,D3Bike] = centerOfMass2(ccD3, DS3_bikes, depthImg3);

save('D1COM2.mat','D1centCar','D1centPpl','D1centBike');
save('D2COM2.mat','D2centCar','D2centPpl','D2centBike');
save('D3COM2.mat','D3centCar','D3centPpl','D3centBike');

save('D1WM2.mat','D1Car','D1Ppl','D1Bike');
save('D2WM2.mat','D2Car','D2Ppl','D2Bike');
save('D3WM2.mat','D3Car','D3Ppl','D3Bike');
```

## 2.5 2e

Segmentation code is as follows. Unfortunately the segmentation code was not perfect as the capturing of generally reasonable boundaries involved massive numbers far greater than 3 meters. Multiple methods of trying different situations to fix this yielded little in the way of substantial closer results.

```
% segmentation matrix calculation

function segOut = segmat2(centroid, matCubeCells, imData, i, segMatrix,
    thresh)
    numDetects = size(imData,1);

    for j = 1:numDetects
        box = imData(j,:);
        xL = round(box(1)); xR = round(box(3)); yT = round(box(2)); yB =
            round(box(4));
        if (xR >= 1242)
            xR = xR - 4;
        end
        matCube = matCubeCells{j};
        matCubeX = matCube(:,:,1); matCubeY = matCube(:,:,2); matCubeZ =
            matCube(:,:,3);
        % matCubeX = matCubeX - centroid(1); matCubeX = matCubeX .^ 2;
        % matCubeY = matCubeY - centroid(2); matCubeY = matCubeY .^ 2;
        matCubeZ = matCubeZ - centroid(3); matCubeZ = matCubeZ .^ 2;
        distVal = matCubeZ;
        dists = sqrt(distVal);
        isSegment = dists <= thresh;
        b = zeros(size(matCubeX,1),size(matCubeX,2));
        b(isSegment) = i;
        segMatrix(yT:yB,xL:xR) = b;
    end
    segOut = segMatrix;
end
```

```
% segmentation matrix chunks

load('ds1data.mat');
load('ds2data.mat');
load('ds3data.mat');

load('D1COM2.mat'); load('D1WM2.mat');
load('D2COM2.mat'); load('D2WM2.mat');
load('D3COM2.mat'); load('D3WM2.mat');

segM1 = zeros(375,1242);
segM2 = zeros(375,1242);
segM3 = zeros(375,1242);

DS1segCars = segmat2(D1centCar,D1Car,DS1_cars,10,segM1,50);
DS1segPpl = segmat2(D1centPpl,D1Ppl,DS1_people,30,segM1,10);
DS1segBikes = segmat2(D1centBike,D1Bike,DS1_bikes,1000,segM1,5);

DS1Segs = DS1segCars + DS1segPpl + DS1segBikes;

DS2segCars = segmat2(D2centCar,D2Car,DS2_cars,10,segM2,50);
DS2segPpl = segmat2(D2centPpl,D2Ppl,DS2_people,30,segM2,50);
DS2segBikes = segmat2(D2centBike,D2Bike,DS2_bikes,60,segM2,2);

DS2Segs = DS2segCars + DS2segPpl + DS2segBikes;

DS3segCars = segmat2(D3centCar,D3Car,DS3_cars,10,segM3,30);
DS3segPpl = segmat2(D3centPpl,D3Ppl,DS3_people,30,segM3,3);
DS3segBikes = segmat2(D3centBike,D3Bike,DS3_bikes,60,segM3,3);

DS3Segs = DS3segCars + DS3segPpl + DS3segBikes;

image(DS3Segs)
```

The segmentation visualizations of the three images are provided below.

## 2.6 2f

Below is code for the textual description of each scene. It takes a matrix of centre of mass points with associated class label number as an input, for a particular scene (image), and reports the locations of objects based on this, relative to the central origin position of the driver.

```
% location description

function describe(imCenters, centerLabels)
carCenters = zeros(1,3);
peopleCenters = zeros(1,3);
bikeCenters = zeros(1,3);
```

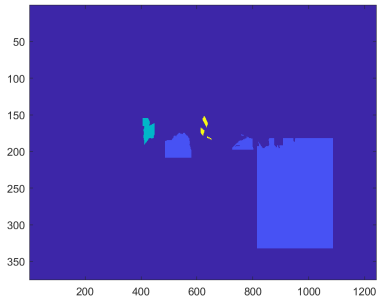


Figure 7: detections of image 1. purple = cars, cyan = people, yellow = bicycles. some false positives exist

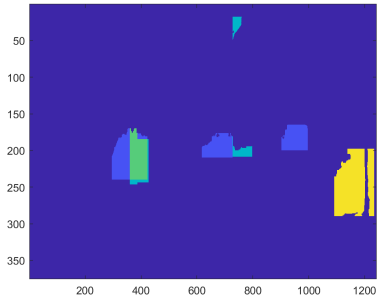


Figure 8: detections of image 2. purple = cars, cyan = people, yellow = bicycles. some false positives exist

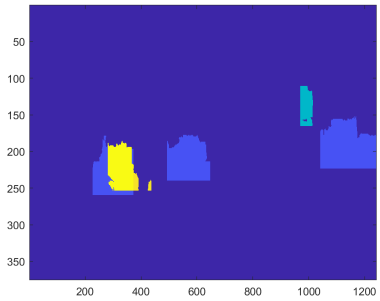


Figure 9: detections of image 3. purple = cars, cyan = people, yellow = bicycles. some false positives exist

```
d = max(max(imCenters)) + 50;
```

```
dLabel = 0;
cars = 0;
people = 0;
bikes = 0;
numCenters = size(imCenters,1);
for i = 1:numCenters
    center = imCenters(i);
    label = centerLabels(i);
    centerDist = norm(center);
    if centerDist < d
        d = centerDist;
        dLabel = label;
    end
    switch label
        case 1
            cars = cars + 1;
            carCenters(cars,:) = center;
        case 2
            people = people + 1;
            peopleCenters(people,:) = center;
        case 3
            bikes = bikes + 1;
            bikeCenters(bikes,:) = center;
        otherwise
            cars = cars + 0;
    end

end

switch dLabel
    case 1
        dLabel = 'car';
    case 2
        dLabel = ' person';
    case 3
        dLabel = ' bike';
    otherwise
        dLabel = 'n unidentified object';
end


fprintf("In the scene, there are %d cars, %d people, and %d
    bikes.\n",cars,people,bikes);
fprintf("The closest object to you is a %s.\n", dLabel);
fprintf("It is %0.1f meters away.\n",d);

numCars = size(carCenters,1); numPeople = size(peopleCenters); numBikes
    = size(bikeCenters,1);

for c = 1:numCars
```

```
    carC = carCenters(c);
    carX = carC(1);
    dirText = 0;
    if carX >= 0
        dirText = 'left';
    else
        dirText = 'right';
    end
    distance = norm(carC);
    fprintf("There is a car %0.1f meters away to your
        %s.\n",distance,dirText);
end

for p = 1:numPeople
    peepC = peopleCenters(p);
    peepX = peepC(1);
    dirText = 0;
    if peepX >= 0
        dirText = 'left';
    else
        dirText = 'right';
    end
    distance = norm(peepC);
    fprintf("There is a person %0.1f meters away to your
        %s.\n",distance, dirText);
end

for b = 1:numBikes
    bkC = bikeCenters(c);
    bkX = bkC(1);
    dirText = 0;
    if bkX >= 0
        dirText = 'left';
    else
        dirText = 'right';
    end
    distance = norm(bkC);
    fprintf("There is a bicycle %0.1f meters away to your
        %s.\n",distance, dirText);
end

end
```

The code works as follows: it first counts the number of total objects, differentiated by class, in the image as a whole, through a single loop through all provided center of mass values. While doing so, it keeps track of the object closest to the origin (the driver) through using a norm calculation, and then reports its relative distance (left or right) and exact distance in meters, in world coordinates. From here, the code simply outputs where each object is individually, in meters and in

relative left-right distance, with three for-loops for each of the three classes.