CSC420 Assignment 4

Mengning Yang
Student Number: 1002437552

November 20, 2019

1. Attached is an image um 000038.png recorded with a camera mounted on a car. The focal length of the camera is 721.5, and the principal point is (609.6, 172.9). We know that the camera was attached to the car at a distance of 1.7 meters above ground.

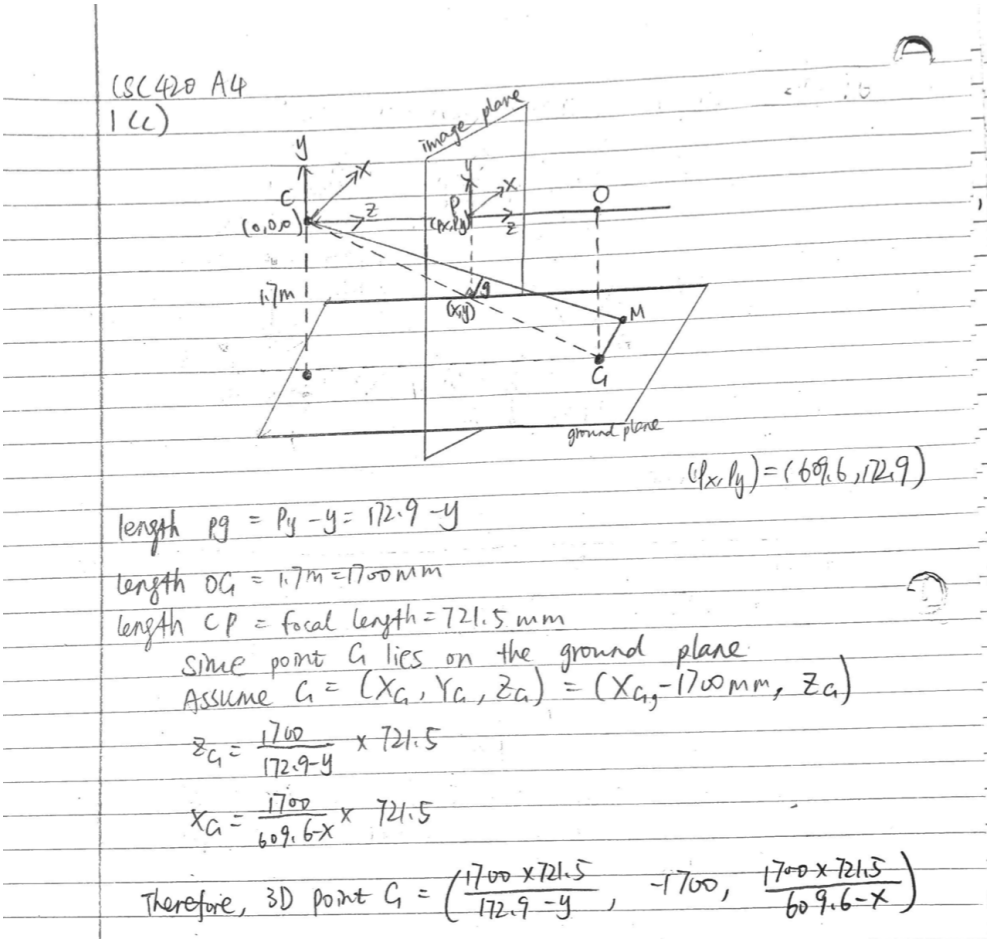
   (a) Write the internal camera parameter matrix K.

   Since we know that formula of the internal camera parameters is $K = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix}$ where (px , py) is the principal point and f is the focal length. We can derive our internal camera parameters as $K = \begin{bmatrix} 721.5 & 0 & 609.6 \\ 0 & 721.5 & 172.9 \\ 0 & 0 & 1 \end{bmatrix}$

   (b) Write the equation of the ground plane in camera's coordinate system. You can assume that the camera's image plane is orthogonal to the ground.

   Assume the camera coordinate is using millimeter as unit and assume the camera is the origin point. Since the camera is 1.7m (1700mm) above the ground, the ground plane is 1700mm below the camera. So ground plane equation is f(x, y, z) = (x, -1.7m, z) = (x, -1700mm, z), or y = -1.7m = -1700mm.

   (c) Compute the 3D location of a 2D point (x,y) in the image by assuming that the point lies on the ground? You can assume that the camera's image plane is orthogonal to the ground. Write a mathematical explanation.



2. In this exercise you are given stereo pairs of images from the autonomous driving dataset KITTI (http://www.cvlibs.net/datasets/kitti/index.php). The images have been recorded with a car driving on the road. Your goal is to create a simple system that analyzes the road ahead of the driving car. Include your code to your solution document.

   (a) Describe (in mathematical form, no code) how to compute disparity for a pair of parallel stereo cameras. Please include an algorithm (pseudo-code) that includes mathematical details. What is the computational complexity of this algorithm? How do you compute depth for each pixel?

   The formula for disparity given in the lecture slides is $x_l - x_r$ where x_l and x_r are two matching points in the left and right parallel cameras' image planes respectively. We can find the matching point by searching along the horizontal line (also called scanline), we scan the line and compare patches to the one in the left image. We are looking for a patch on the scanline that's most similar to the patch on the right. And we can find the most similar patch by calculating the SSD (sum of squared differences) between the two patches. Formula of SSD is as the following:

$$SSD(\text{patch}_l, \text{patch}_r) = \sum_x \sum_y (I_{\text{patch}_l}(x,y) - I_{\text{patch}_r}(x,y))^2$$

   pseudo-code of computing disparity for a pair of parallel stereo cameras.

```
import numpy as np
from PIL import Image

def stereo_match(left_img, right_img, kernel, max_offset):
    # Load in both images
    left_img = Image.open(left_img).convert('L')
    left = np.asarray(left_img)
    right_img = Image.open(right_img).convert('L')
    right = np.asarray(right_img)
    # assume that both images are same size
    w, h = left_img.size

    # Depth (or disparity) map
    depth = np.zeros((w, h), np.uint8)
    depth.shape = h, w

    kernel_half = int(kernel / 2)

    # this is used to map depth map output to 0-255 range
    offset_adjust = 255 / max_offset

    for y in range(kernel_half, h - kernel_half):

        for x in range(kernel_half, w - kernel_half):
            best_offset = 0
            prev_ssd = 65534

            for offset in range(max_offset):
                ssd = 0
                ssd_temp = 0
```

```
                #v and u are the x,y of our local window search
                #used to ensure a good match
                #by the squared differences of the neighbouring pixels
                for v in range(-kernel_half, kernel_half):
                    for u in range(-kernel_half, kernel_half):
                        # iteratively sum the sum of squared differences
                        #value for this block
                        ssd_temp = int(left[y+v, x+u]) -
                        int(right[y+v, (x+u) - offset])

                        ssd += ssd_temp * ssd_temp

                # if this value is smaller than the previous ssd at this
                # block then it's theoretically a closer match. Store
                # this value against this block
                if ssd < prev_ssd:
                    prev_ssd = ssd
                    best_offset = offset

            # set depth output for this x,y location to the best match
            depth[y, x] = best_offset * offset_adjust

    # Convert to PIL and save it
    Image.fromarray(depth).save('disparity.png')
```

The computational complexity of this algorithm is approximately:
O(ImageWidth*ImageHeight*WindowSize*WindowSize)

Then, in order to compute depth Z, we use $Z = \frac{f.T}{x_l - x_r}$ where f is the focal length of the camera, T is the baseline value. Both of these can be determined from the camera intrinsic parameters matrix. See code of how to compute the depth in part (c).

(b) What is a fundamental matrix? Describe an algorithm (pseudo-code) to compute the fundamental matrix from a pair of uncalibrated (non-parallel) cameras. Include mathematical details. What is the computational complexity of the algorithm? Is it possible to compute depth for a pair of images taken from non-parallel cameras for which you do not know the relative pose/distance? Please provide an argument for your answer.

The fundamental matrix F is a 3x3 matrix and is defined as $I_r = Fp_l$ where I_r is the right epipolar line corresponding to the point p_l. F has 9 elements but since we do not care about scaling, it only has 8 elements and we can effectively estimate F with 7 pairs of matching points in both images (but in this course we need 8 as professor mentioned in the lecture). To compute F, we need to solve a linear system: where f is 3x3 where $n \geqslant 8$

$$f \begin{bmatrix} x_{r,1}x_{l,1} & x_{r,1}y_{l,1} & x_{r,1} & y_{r,1}x_{l,1} & y_{r,1}y_{l,1} & y_{r,1} & x_{l,1} & y_{l,1} & 1 \\ & & & & \vdots & & & & \\ x_{r,n}x_{l,n} & x_{r,n}y_{l,n} & x_{r,n} & y_{r,n}x_{l,n} & y_{r,n}y_{l,n} & y_{r,n} & x_{l,n} & y_{l,n} & 1 \end{bmatrix} = 0$$

To use F for the stereo problem, we can compute homographies that transform each image plane such that they are parallel and since we can easily solve stereo for parallel cameras, then we can proceed as the case of parallel cameras. See the following code to estimate the fundamental matrix (assume that we already found 8 reliable matches across two images without any constraints by using SIFT algorithm demonstrated in assignment 3.):

```
def compute_fundamental(x1,x2):
"""
Computes the fundamental matrix from corresponding points
(x1,x2 3*n arrays) using the 8 point algorithm.
Each row in the A matrix below is constructed as
[x'*x, x'*y, x', y'*x, y'*y, y', x, y, 1]
"""

    n = x1.shape[1]
    if x2.shape[1] != n:
        raise ValueError("Number of points don't match.")

    # build matrix for equations
    A = zeros((n,9))
    for i in range(n):
        A[i] = [x1[0,i]*x2[0,i], x1[0,i]*x2[1,i], x1[0,i]*x2[2,i],
                x1[1,i]*x2[0,i], x1[1,i]*x2[1,i], x1[1,i]*x2[2,i],
                x1[2,i]*x2[0,i], x1[2,i]*x2[1,i], x1[2,i]*x2[2,i] ]

    # compute linear least square solution
    U,S,V = linalg.svd(A)
    F = V[-1].reshape(3,3)

    # constrain F
    # make rank 2 by zeroing out last singular value
    U,S,V = linalg.svd(F)
    S[2] = 0
    F = dot(U,dot(diag(S),V))

    return F/F[2,2]
```

Professor Fidler mentioned that we only need to summarize what's in the lecture slides into a short algorithm that computes the fundamental matrix. Since in the slides it didn't cover how to solve a linear system and I personally asked Professor Fidler if we need to know Singular Value Decomposition for this course and she said no. So I'm using linalg.svd() in the code above. If we perform SVD on a m*n matrix, the computational complexity of solving the fundamental matrix is approximately $O(min(m*n^2, n*m^2))$.

It is possible to compute depth for a pair of images taken from non-parallel cameras for which we do not know the relative pose or distance, because the fundamental matrix can be computed by any pair of corresponding points in both images alone, and no knowledge of camera internal parameters is required and no knowledge of relative pose is required. If we know the fundamental matrix, we can then compute the camera projection matrices Pl and Pr (under some ambiguity), which means that we don't need the relative poses of the two cameras, we can compute it. This is very useful in scenarios where I just grab pictures from the web.

(c) For each image compute depth. In particular, compute depth which is a n m matrix, where n is the height and m the width of the original image. The value depth(i,j) should be the depth of the pixel (i,j). In your solution document, include a visualization of the depth matrices.

```
import numpy as np
from scipy import spatial, cluster
```

```
import cv2 as cv
import math
import re
import os
from collections import Counter
from matplotlib import pyplot as plot

def display_image(img, file_name=None, save_norm=True, save_type=np.uint8):
    """
    Shows an image (max-min normalized to 0-255),
    and saves it if a filename is given
    save_norm = whether to save the normalized image
    save_type = what datatype to save the image as
    """

    flt_img = img.astype(float)
    img_max, img_min = np.max(flt_img), np.min(flt_img)
    norm = (((flt_img-img_min)/(img_max-img_min))*255).astype(np.uint8)

    if len(img.shape) == 2:
        plot.imshow(norm, cmap='gray')
    elif (len(img.shape) == 3):
        plot.imshow(cv.cvtColor(norm, cv.COLOR_BGR2RGB))
    plot.show()

    to_save = norm if save_norm else flt_img
    if file_name:
        cv.imwrite(file_name, to_save)

def depth_image(disparity_image, base_width, focal_length):
    """
    Creates a depth image using a disparity image and
    some camera parameters (base width, focal length)
    """
    #using the formula stated in lecture slides
    numerator = base_width * focal_length

    # Avoid "division by zero" errors
    denominator = disparity_image.astype(float)
    denominator[denominator == 0] = 1
    result = numerator / denominator

    # Set what should have been infinity to maximum
    result[denominator == 0] = np.max(result)

    return result

def get_camera_params(param_path):
    """
    Returns dictionary of camera parameter values
    stored in the file at param_path
    """
    # Lines of the file are of form (param : value)
```

```
    line_regex =
    re.compile(r"(?P<param>(\w+)):(\s+)(?P<value>(\d+(\.\d+)?))")

    with open(param_path, "r") as file:

        # Matching the regex to the lines in the file
        matches = (line_regex.match(line) for line in file.readlines())

        # Organize the parameters into a dictionary for easy access
        return {
            match.group("param") : float(match.group("value"))
            for match in matches
        }

if __name__ == '__main__':
    test_image_ids =[
     x.strip()
     for x in open("/Users/CYANG/Desktop/CSC420A4/data/test/test.txt",
     "r").readlines()]

    base_path = "/Users/CYANG/Desktop/CSC420A4/data/test/"
    for img_id in test_image_ids[:3]:#only use the first 3 images
        #txt files
        camera_params = get_camera_params(base_path +
        "calib/{}_allcalib.txt".format(img_id))

        disparity_img = cv.imread(base_path +
        "results/{}_left_disparity.png".format(img_id),cv.IMREAD_GRAYSCALE)

        depth_img = depth_image(disparity_img, camera_params["baseline"],
        camera_params["f"])

        display_image(depth_img, "q2c-{}-depth.png".format(img_id),
        save_norm=False, save_type=np.float32)
```

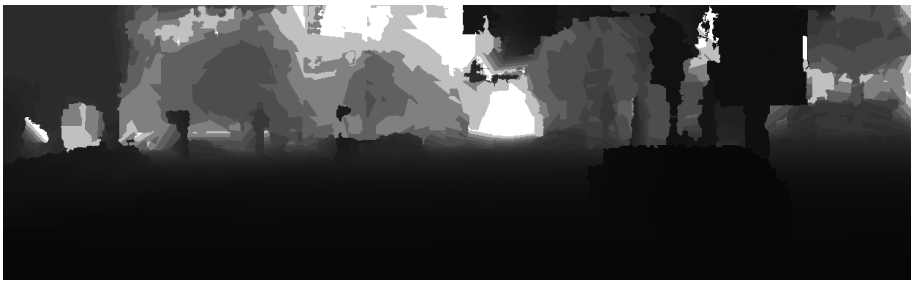


Figure 1: depth map of image 004945.png

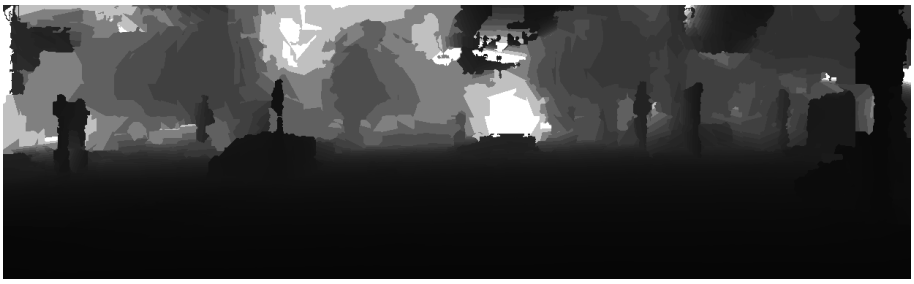


Figure 2: depth map of image 004964.png

Figure 3: depth map of image 005002.png

(d) In your solution document, include a visualization of the first three images with all the car, person and cyclist detections. Mark the car detections with red, person with blue and cyclist with cyan rectangles. Inside each rectangle (preferably the top left corner) also write the label, be it a car, person or cyclist.

Since in the detections.mat files, there are no detections of cyclist whatsoever, so in my visualization there is no cyclist either. For 004945.png, there are only three cars, for 004964.png, there are only three cars and one person, for 005002.png, there are only three cars.

```
import numpy as np
from scipy import spatial, cluster
import scipy.io as spio
import cv2 as cv
import math
import re
import os
from collections import Counter
from matplotlib import pyplot as plot

def display_image(img, file_name=None, save_norm=True, save_type=np.uint8):
    """
    Shows an image (max-min normalized to 0-255),
    and saves it if a filename is given
    save_norm = whether to save the normalized image
    save_type = what datatype to save the image as
    """

    flt_img = img.astype(float)
    img_max, img_min = np.max(flt_img), np.min(flt_img)
```

```
    norm = (((flt_img-img_min)/(img_max-img_min))*255).astype(np.uint8)

    if len(img.shape) == 2:
        plot.imshow(norm, cmap='gray')
    elif (len(img.shape) == 3):
        plot.imshow(cv.cvtColor(norm, cv.COLOR_BGR2RGB))
    plot.show()

    to_save = norm if save_norm else flt_img
    if file_name:
        cv.imwrite(file_name, to_save)



if __name__ == '__main__':
   test_image_ids =[
    x.strip()
    for x in open(
    "/Users/CYANG/Desktop/CSC420A4/data/test/test.txt", "r").readlines()]

   CAR, PERSON, BICYCLE = 0, 1, 2
   cls_lst = [CAR, PERSON, BICYCLE]

   cls_to_name = {
        PERSON : "PERSON",
        BICYCLE : "BICYCLE",
        CAR : "CAR",
    }

   cls_to_col = {
        PERSON:(255, 0, 0), # Blue
        BICYCLE:(255, 255, 0),# Cyan
        CAR:(0, 0, 255), # Red
    }


   for img_id in test_image_ids[:3]:
       base_path = "/Users/CYANG/Desktop/CSC420A4/data/test/"
       detections = spio.loadmat(base_path +
       "results/dets-test/{}_dets".format(img_id))
       img = cv.imread(base_path+"left/{}.jpg".format(img_id))

       num_class = detections['dets'].size

       class_lst = []
       for cls in range(num_class):#3
           object_lst = []
           #every object in each class
           for i in range(len(detections['dets'][cls][0])):
               obj = []

               for index in range(6):
                   obj.append(int(detections['dets'][cls][0][i][index]))
```

```
            object_lst.append(obj)

        class_lst.append(object_lst)
        print(class_lst)

    for i in range(num_class):
        for obj in class_lst[i]:
            cls = cls_lst[i]
            cv.rectangle(img,(obj[0], obj[1]),
            (obj[2], obj[3]),cls_to_col[cls] ,2)
            label_txt = cls_to_name[cls]
            label_coords = (obj[0], obj[1])
            cv.putText(img,label_txt,label_coords,
            cv.FONT_HERSHEY_SIMPLEX, 0.65,cls_to_col[cls],2)

            display_image(img,"q2d-{}-detections.png".format(img_id))
```

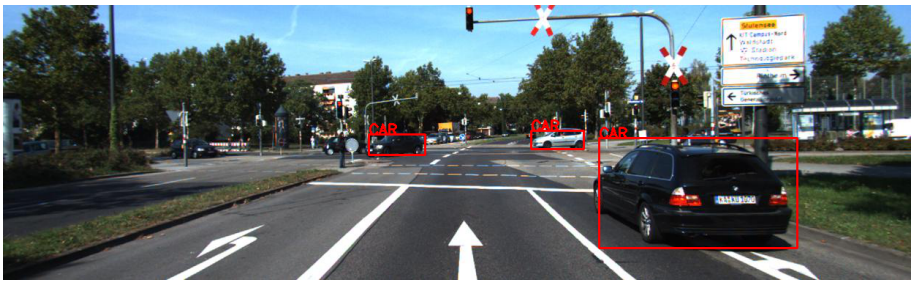


Figure 4: visualization of detections of image 004945.png

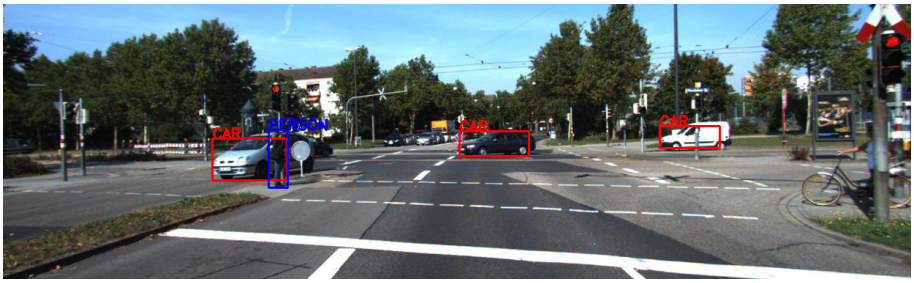


Figure 5: visualization of detections of image 004964.png

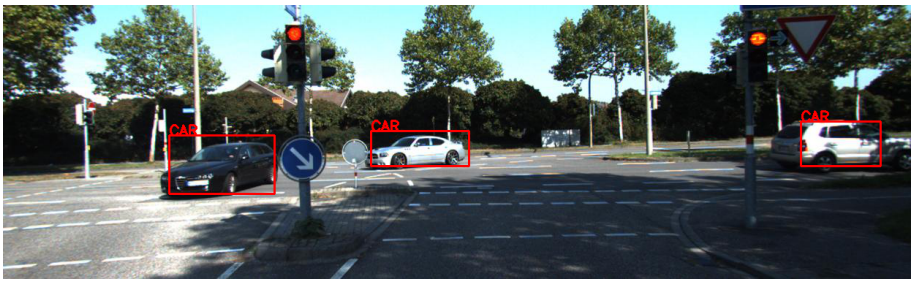


Figure 6: visualization of detections of image 005002.png

(e) Compute the 3D location of each detected object. How will you do that? Come up with an approach to get a (2D) segmentation of each object. Can you use depth to help you?

Part 1 : Compute the 3D location of each detected object.
Assuming points with equal mass, the center of mass of a collection of points will be the average point. However, note that within a detection box there are two distinct types of pixels: object and non-object pixels. If the object within the box is assumed to be the only object of significance, then the distribution of points will be bimodal between the object and the background pixels. Therefore, k-means clustering with k = 2 can be used to approximate the two means.
The following formulas will be used to calculate each point (x, y, z):

$$z = \frac{f \times b}{d}$$

$$x = \frac{x_L \times z}{f}$$

$$y = \frac{y_L \times z}{f}$$

Where:
z : z-coordinate of the pixel (meters)
x : x-coordinate of the pixel (meters)
y : y-coordinate of the pixel (meters)
f : focal length of the camera(s) (pixels)
b : base width of the two camera (meters)
d : x-disparity of the pixel (pixels)
x_L : x-coordinate of the pixel in the left image (pixels)
y_L : y-coordinate of the pixel in the left (or right, they're assumed to be the same) image (pixels)

```
import numpy as np
from scipy import spatial, cluster
import cv2 as cv
import math
import re
import os
from collections import Counter
from matplotlib import pyplot as plot
import scipy.io as spio

def depth_image(disparity_image, base_width, focal_length):
```

```
    """
    Creates a depth image using a disparity image and
    some camera parameters (base width, focal length)
    """
    #using the formula stated in lecture slides
    numerator = base_width * focal_length

    # Avoid "division by zero" errors
    denominator = disparity_image.astype(float)
    denominator[denominator == 0] = 1
    result = numerator / denominator

    # Set what should have been infinity to maximum
    result[denominator == 0] = np.max(result)

    return result

def image_3d_coords(disparity,base_width,focal_length,center_x,center_y):
    """
    Produces an image where each pixel is of form [x, y, z]
    """

    # Compute Z/depth first
    z = depth_image(disparity, base_width, focal_length)

    # Shift raw coordinates by optical center
    raw_y, raw_x = np.indices(z.shape[:2])
    shifted_y, shifted_x = raw_y - center_y, raw_x - center_x

    #d = xL-xR --> xL= d+xR (assuming x-coordinates of disparity are xR)
    #y is assumed to be the same across both images
    yL = shifted_y
    xL = shifted_x + disparity

    x = (xL * z) / focal_length
    y = (yL * z) / focal_length

    return np.dstack((x, y, z))

def find_closest_mean(vectors, k):
    """
    Returns the mean vector with the smallest magnitude.
    (vectors is assumed to be a k-modal distribution)
    """

    # Run K-means
    centroids, labels = cluster.vq.kmeans2(vectors, k)

    # Determine magnitude of centroids
    centroid_mags = (centroids ** 2).sum(axis=1) ** 0.5
    closest_label = np.argmin(centroid_mags)

    # Return the closest centroid
```

```
    return centroids[closest_label]


def centers_of_mass(coords_3d, detections):
    """
    Takes an image of coordinates (each pixel is of form [x, y, z]),
    and an array of detections each of form
    [y_top, x_left, y_bottom, x_right, class]

    Computes a numpy array of form [[x, y, z]], where the i-th element
    is equal to the center of mass of the i-th detection's points
    """
    # Slices of given coordinate matrix (for each detection box)
    slice_inds = (
        row[:4].astype(int)
        for row in detections
    )

    slices = (
        coords_3d[y_top:y_bottom, x_left:x_right]
        #note the order of the params!!!
        for (x_left, y_top, x_right, y_bottom) in slice_inds
    )

    # Flattened (x, y) slices
    flat_slices = (
        s.reshape(np.product(s.shape[:2]), 3)
        for s in slices
    )

    # Return closest mean of the bimodal distribution
    return np.array([
        find_closest_mean(vecs, 2)
        for vecs in flat_slices
    ])

def get_camera_params(param_path):
    """
    Returns dictionary of camera parameter values
    stored in the file at param_path
    """
    # Lines of the file are of form (param : value)
    line_regex =
    re.compile(r"(?P<param>(\w+)):(\s+)(?P<value>(\d+(\.\d+)?))")

    with open(param_path, "r") as file:

        # Matching the regex to the lines in the file
        matches = (line_regex.match(line) for line in file.readlines())

        # Organize the parameters into a dictionary for easy access
        return {
            match.group("param") : float(match.group("value"))
```

```
            for match in matches
        }

def load_dets_to_array(detections):
    '''
    given a dictionary loaded from a .mat file
    convert it to an array of the form
    [y_top, x_left, y_bottom, x_right, class]
    '''
    num_class = detections['dets'].size

    object_lst = []
    for cls in range(num_class):
        for i in range(len(detections['dets'][cls][0])):
            obj = []

            for index in range(4):
                obj.append(detections['dets'][cls][0][i][index])

            if obj != []:
                obj.append(cls)
                obj_array = np.asarray(obj)
                object_lst.append(obj_array)

    object_array = np.asarray(object_lst)
    return object_array


if __name__ == '__main__':

    base_path = "/Users/CYANG/Desktop/CSC420A4/data/test/"
    CAR, PERSON, BICYCLE = 0, 1, 2
    cls_lst = [CAR, PERSON, BICYCLE]

    cls_to_name = {
        PERSON : "PERSON",
        BICYCLE : "BICYCLE",
        CAR : "CAR",
    }

    cls_to_col = {
        PERSON:(255, 0, 0), # Blue
        BICYCLE:(255, 255, 0),# Cyan
        CAR:(0, 0, 255), # Red
    }

    test_image_ids =[
    x.strip()
    for x in open(base_path+"test.txt", "r").readlines()
    ]

    for img_id in test_image_ids[:3]:
```

```
        # Compute 3D coordinates of pixels
        camera_params = get_camera_params(base_path +
        "calib/{}_allcalib.txt".format(img_id))

        disparity = cv.imread(base_path +"results/{}_left_disparity.png"
        .format(img_id),cv.IMREAD_GRAYSCALE)

        baseline, f, px, py =
        [camera_params[k] for k in ["baseline", "f", "px", "py"]]
        coords = image_3d_coords(disparity, baseline, f, px, py)


        dets = spio.loadmat(base_path +
        "results/dets-test/{}_dets".format(img_id))

        detections = load_dets_to_array(dets)

        np.save("/Users/CYANG/Desktop/q2(e)1/{}_detections.npy"
        .format(img_id), detections)

        # Centers of mass for each detection, in order of detections
        mass_centers = centers_of_mass(coords, detections)

        # Save the centers of mass and coordinates for each image
        np.save("/Users/CYANG/Desktop/q2(e)1/{}_3d_coords.npy"
        .format(img_id), coords)

        np.save("/Users/CYANG/Desktop/q2(e)1/{}_mass_centers.npy"
        .format(img_id), mass_centers)
```

part 2 : 2D segmentation of each object.
My approach to get a 2D segmentation of each object: for each detection, compute the 3D location of the object, which is already done in part 1 of this question. Then, find all pixels inside each bounding box that are at most 3 meters away from the computed 3D location. Then create a segmentation image, by first creating a matrix that has the same number of columns and rows as the original RGB image, and initialize the matrix with all zeros. Then for the i-th detection (bounding box), assign a value i to all pixels that I found to belong to detection i. See the following code:

```
import numpy as np
from scipy import spatial, cluster
import cv2 as cv
import math
import re
import os
from collections import Counter
from matplotlib import pyplot as plot
import scipy.io as spio

def display_image(img,file_name=None,save_norm=True,save_type=np.uint8):
    """
    Shows an image (max-min normalized to 0-255), and saves it
    if a filename is given
    save_norm = whether to save the normalized image
```

```
    save_type = what datatype to save the image as
    """

    flt_img = img.astype(float)
    img_max, img_min = np.max(flt_img), np.min(flt_img)
    norm=(((flt_img-img_min)/(img_max-img_min))*255).astype(np.uint8)

    if len(img.shape) == 2:
        plot.imshow(norm, cmap='gray')
    elif (len(img.shape) == 3):
        plot.imshow(cv.cvtColor(norm, cv.COLOR_BGR2RGB))
    plot.show()

    to_save = norm if save_norm else flt_img
    if file_name:
        cv.imwrite(file_name, to_save)

def segment_image(coords, detections, centers, max_dist):

    # The image to return
    ret = np.zeros(coords.shape, int)

    # Go through each detection
    for center, (x_left, y_top, x_right, y_bottom, cls)
    in zip(centers, detections):

        # Int-ify each bounding box coordinate for slicing
        yT,yB,xL,xR = int(y_top),int(y_bottom),int(x_left),int(x_right)

        # Get slice of coordinates and the center of mass
        coord_slice = coords[yT:yB, xL:xR]

        # Compute euclidean distance from center for each coordinate
        center_dists = ((coord_slice - center) ** 2).sum(axis=2) ** 0.5

        # Set the points within max_dist to the class color
        ret[yT:yB, xL:xR][center_dists < max_dist] = cls_to_col[cls]

    return ret


if __name__ == '__main__':
    test_image_ids =[
    x.strip()
    for x in open("/Users/CYANG/Desktop/CSC420A4/data/test/test.txt",
    "r").readlines()]

    CAR, PERSON, BICYCLE = 0, 1, 2
    cls_lst = [CAR, PERSON, BICYCLE]

    cls_to_name = {
        PERSON : "PERSON",
        BICYCLE : "BICYCLE",
```

```
    CAR : "CAR",
}

cls_to_col = {
    PERSON:(255, 0, 0), # Blue
    BICYCLE:(255, 255, 0),# Cyan
    CAR:(0, 0, 255), # Red
}

for img_id in test_image_ids[:1]:

    coords = np.load("/Users/CYANG/Desktop/q2(e)1/{}_3d_coords.npy"
    .format(img_id))

    centers =
    np.load("/Users/CYANG/Desktop/q2(e)1/{}_mass_centers.npy"
    .format(img_id))

    detections =
    np.load("/Users/CYANG/Desktop/q2(e)1/{}_detections.npy"
    .format(img_id))

    print(detections)

    # Segment the image with maximum center-distance of 3 meters
    segmented = segment_image(coords, detections, centers, 3)

    display_image(segmented,"q2e-{}-segmentation.png".format(img_id))
```

Figure 7: image 004945.png

Figure 8: visualization of segmentation of image 004945.png

Figure 9: image 004964.png

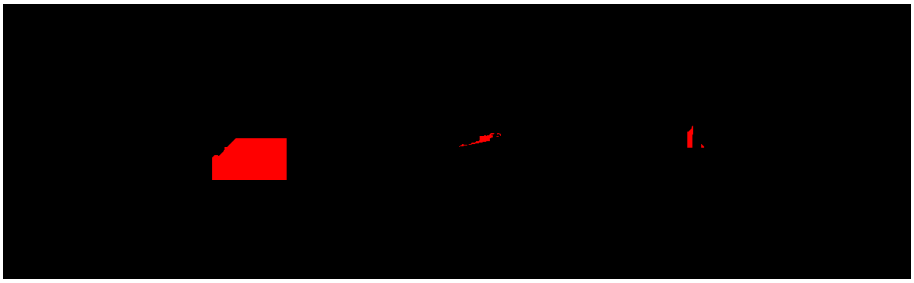


Figure 10: visualization of segmentation of image 004964.png

Figure 11: image 005002.png

Figure 12: visualization of segmentation of image 005002.png

Then, use the segmentation images obtained in the last step to apply the transparent effect by using code given in tutorial.

```
import numpy as np
```

```
import cv2
from matplotlib import pyplot as plt
import scipy.io
import matplotlib.colors as mcolors
import webcolors

if __name__ == '__main__':
    base_path = "/Users/CYANG/Desktop/A4/q2(e)2/"

    test_image_ids =[
    x.strip()
    for x in open(base_path+"test.txt", "r").readlines()
    ]

    for img_id in test_image_ids[:3]:

        gt_mask = cv2.imread(base_path +
        'q2e-{}-segmentation.png'.format(img_id),cv2.IMREAD_GRAYSCALE)
        plt.imshow(gt_mask)
        plt.show()
        gt_mask.shape
        print(gt_mask.max())
        print(gt_mask.min())
        # 16 is background
        obj_ids = np.unique(gt_mask)
        print(obj_ids)
        number_object = obj_ids.shape[0]

        # norm ids
        count = 0
        for o_id in obj_ids:
            gt_mask[gt_mask == o_id] = count
            count += 1

        base_COLORS = []

        for key, value in mcolors.CSS4_COLORS.items():
            rgb = webcolors.hex_to_rgb(value)
            base_COLORS.append([rgb.blue, rgb.green, rgb.red])
        base_COLORS = np.array(base_COLORS)

        np.random.seed(99)
        base_COLORS = np.random.permutation(base_COLORS)

        colour_id =
        np.array([(id) % len(base_COLORS) for id in range(number_object)])

        COLORS = base_COLORS[colour_id]
        COLORS = np.vstack([[0, 0, 0], COLORS]).astype("uint8")
        mask = COLORS[gt_mask]

        imgLeft = cv2.imread(base_path+'left/{}.jpg'.format(img_id))
        output = ((0.4 * imgLeft) + (0.6 * mask)).astype("uint8")
```

```
        plt.imshow(output[:,:,::-1])
        plt.show()
```

Figure 13: visualization of segmentation on image 004945.png

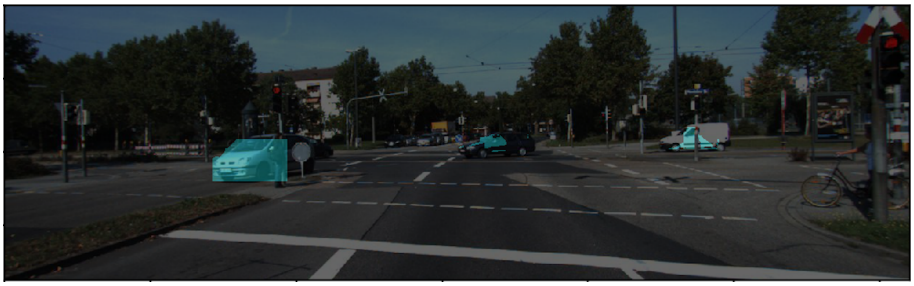


Figure 14: visualization of segmentation on image 004964.png

Figure 15: visualization of segmentation on image 005002.png