Mid-Term

ENPM673

Vishnu Mandala
119452608

Problem 1:

A. **Let's say we want to design a robot to perform indoor wall painting tasks. List all the perception sensors that would be needed for this task and why they would be needed.**
B. **Similarly, assume that we want to design a robot that will fix an underwater broken pipe. This robot should be able to work in any type of underwater water environment. What type of perception sensors can be used in this case, and why?**

Solution:

A. To design a robot for indoor wall painting, several perception sensors would be required. These sensors would allow the robot to perceive its surroundings and interact with the environment to paint the walls effectively.

   Here are the perception sensors that would be needed for this task:

   a. **Camera:** A camera would be essential for the robot to see and analyze the walls, corners, and edges it needs to paint. A high-resolution camera would capture details and colors of the walls, identify the areas that need painting, and detect any obstacles or objects that may be present in the room. The camera would also allow the robot to measure the height, width, and length of the walls and calculate the amount of paint needed for each section.
   b. **LIDAR Sensor:** Light Detection and Ranging (LIDAR) sensors use laser beams to detect and measure the distance between the robot and the objects in the room. This sensor would enable the robot to create a 3D map of the room, including the walls, ceiling, and floor. The robot could use this map to navigate around the room without bumping into walls, furniture, or other objects, ensuring that it paints the walls efficiently and without mistakes.
   c. **Color Sensor:** The color sensor would allow the robot to detect and recognize the colors on the wall accurately. It could be used to match the paint colors on the walls with the ones in the paint can, ensuring that the robot is using the correct paint for each wall. This sensor would also allow the robot to detect any imperfections in the walls, such as cracks, holes, or bumps, and paint over them to achieve a smooth and uniform finish.
   d. **Pressure Sensor:** The pressure sensor would detect the pressure applied by the robot's painting mechanism on the walls. This sensor would ensure that the robot applies consistent pressure on the walls, achieving a uniform paint layer. It would also detect when the robot is painting over rough or uneven surfaces and adjust the pressure accordingly to achieve the desired finish.
   e. **Infrared Sensor:** The infrared sensor would allow the robot to detect the temperature of the walls. It could be used to determine if the walls are too hot or too cold, affecting the paint's ability to adhere to the wall. The robot could use this sensor to adjust its painting speed, direction, and pressure based on the temperature of the walls, ensuring that the paint adheres well and dries uniformly.
   f. **Motion Sensor:** A motion sensor can detect the robot's movement and orientation, providing information on its speed, direction, and acceleration. With the motion sensor, the robot can accurately control its movements and maintain stability while painting.

   In conclusion, designing a robot for indoor wall painting tasks would require multiple perception sensors to allow the robot to perceive its surroundings accurately. These sensors would enable the robot to analyze the walls, detect colors, navigate around the room, apply consistent pressure, and adjust its painting mechanism based on the temperature of the walls. Using these sensors, the robot could paint the walls efficiently and achieve a uniform and smooth finish.

B. To design a robot that can fix an underwater broken pipe, we need to consider the various challenges of the underwater environment, such as low visibility, pressure, and the presence of marine life. Perception sensors are essential for the robot to sense the environment and perform the task accurately. Here are some types of sensors that can be used in this case:

   a. **Sonar Sensor:** Sound Navigation and Ranging (SONAR) sensors emit sound waves that bounce off objects in the water, allowing the robot to perceive the environment. These sensors can detect the distance, size, and shape of objects, including the broken pipe. Sonar sensors can be used in low visibility conditions and are ideal for mapping the environment and locating the broken pipe.

b. **Camera:** Cameras can be used to capture images of the underwater environment, providing visual information to the robot. These sensors can detect the position and orientation of the broken pipe, enabling the robot to approach it accurately. However, cameras may not work well in low visibility conditions, such as murky water.
c. **Pressure Sensor:** Pressure sensors can be used to measure the pressure of the water, providing information about the depth and location of the robot. These sensors can also be used to detect leaks in the pipe by measuring the pressure difference between the pipe and the surrounding water.
d. **Magnetic Sensor:** Magnetic sensors can be used to detect the presence of metal objects, such as the broken pipe. These sensors work by detecting changes in the magnetic field caused by the metal object, allowing the robot to locate the broken pipe accurately.
e. **Chemical Sensor:** Chemical sensors can be used to detect changes in the water's chemical composition, such as the presence of oil or gas. These sensors can be used to detect leaks in the pipe or other environmental hazards.
f. **Bionic Sensor:** Depending on the complexity of the task, bionic sensors can be employed. Bionic sensors work by mimicking biological systems. For example, some underwater robots have been designed with artificial lateral lines that mimic the lateral line system found in fish. This allows the robot to sense changes in water pressure and flow and navigate its environment. This can be use as an advanced pressure sensor in exceptional cases.

In conclusion, the selection of perception sensors for an underwater pipe repair robot will depend on several factors such as the type of fault, depth of the water, and nature of the pipe. A combination of different sensors such as sonar, pressure, visual, magnetic, and chemical sensors can be used to provide a comprehensive view of the underwater environment, ensuring that the robot can operate effectively and safely.

Problem 2:

Using the video from the first homework, assuming that the diameter of the ball is around 11 pixels, use Hough transform to detect the ball.

Solution:

The Hough Transform maps each point in the image to a parameter space, where the parameters correspond to the parameters of the shape being detected. In the case of detecting circles, the parameter space is three-dimensional, with parameters (x,y,r), where (x,y) are the coordinates of the circle's center and r is its radius.

For each point (x,y) in the image, a set of (x',y',r) values that could correspond to a circle centered at (x,y) is computed and added to the accumulator array. The set of possible (x',y',r) values is given by the equation of a circle:

$$(x'-x)^2+(y'-y)^2=r^2$$

This equation can be rewritten as:

$$x'=x+r*cos(\theta)$$
$$y'=y+r*sin(\theta)$$

where Θ is the angle between the positive x-axis and the line connecting (x',y') to (x,y)

For each (x,y) point, a range of r values is tested to cover all possible circle radii.
The accumulator array is a three-dimensional array with dimensions (width, height, radius), where (width, height) is the size of the image and radius is the maximum radius of the circles to be detected. For each (x',y',r) value that corresponds to a circle centered at (x,y), the corresponding bin in the accumulator array is incremented.

After all points in the image have been processed, the accumulator array is scanned to find the bins with the highest values. Each high-value bin corresponds to a circle in the image, and its center and radius can be calculated from the bin coordinates (x,y,r).

The cv2.HoughCircles() function in OpenCV implements this algorithm with some additional optimizations, such as using the gradient information to improve the detection accuracy and avoiding duplicate detections by suppressing nearby circles.

The param1 and param2 arguments control the detection threshold. param1 is the higher threshold value for Canny edge detection, and it is used to detect the edges of the circles in the image. param2 is the accumulator threshold value, and only circles that have a number of votes greater than this threshold value will be detected.

The minDist argument controls the minimum distance between the centers of detected circles. If two circles are too close, only one of them will be detected. The minRadius and maxRadius arguments control the minimum and maximum radius of the circles to be detected.

Code:

```
cap = cv2.VideoCapture('ball.mov')  # Read the video file

while(cap.isOpened()):
    ret, frame = cap.read()
    if ret == False:
        break

    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)    # Convert the frame to HSV color space
    mask = cv2.inRange(hsv, (0,150,130), (6,255,255))   # Filter the Red Channel using color thresholding

    # Find the center point of the ball using color thresholding
    ycoords, xcoords = np.where(mask > 0)
    if len(ycoords) > 0 and len(xcoords) > 0:   # Check if the ball is in the frame
        center = (int(np.mean(xcoords)), int(np.mean(ycoords)))
    else:
        center = None

    if center is not None:
        # Define the search region around the known center
        search_region = frame[max(center[1]-50, 0):min(center[1]+50, frame.shape[0]), max(center[0]-50, 
0):min(center[0]+50, frame.shape[1])]

        gray = cv2.cvtColor(search_region, cv2.COLOR_BGR2GRAY)  # Convert the search region to grayscale
        blur = cv2.GaussianBlur(gray, (5, 5), 0)    # Apply Gaussian blur to reduce noise
        edges = cv2.Canny(blur, 50, 150)    # Apply Canny edge detection
        circles = cv2.HoughCircles(edges, cv2.HOUGH_GRADIENT, dp=1, minDist=0.1, param1=80, param2=19, 
minRadius=5, maxRadius=11)  # Perform Hough Transform to detect circles

        # Draw circles on the original frame
        if circles is not None:
            circles = np.round(circles[0, :]).astype("int")
            for (x, y, r) in circles:
                # Convert circle coordinates from search region to frame coordinates
                x += center[0]-50
                y += center[1]-50
                cv2.circle(frame, (x, y), r, (0, 255, 0), 2)

    # Display the frame with detected circles
    cv2.imshow('Problem 2', frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release the video file and close all windows
cap.release()
cv2.destroyAllWindows()

print('\nProblem 2 - Done!')
```

Output:

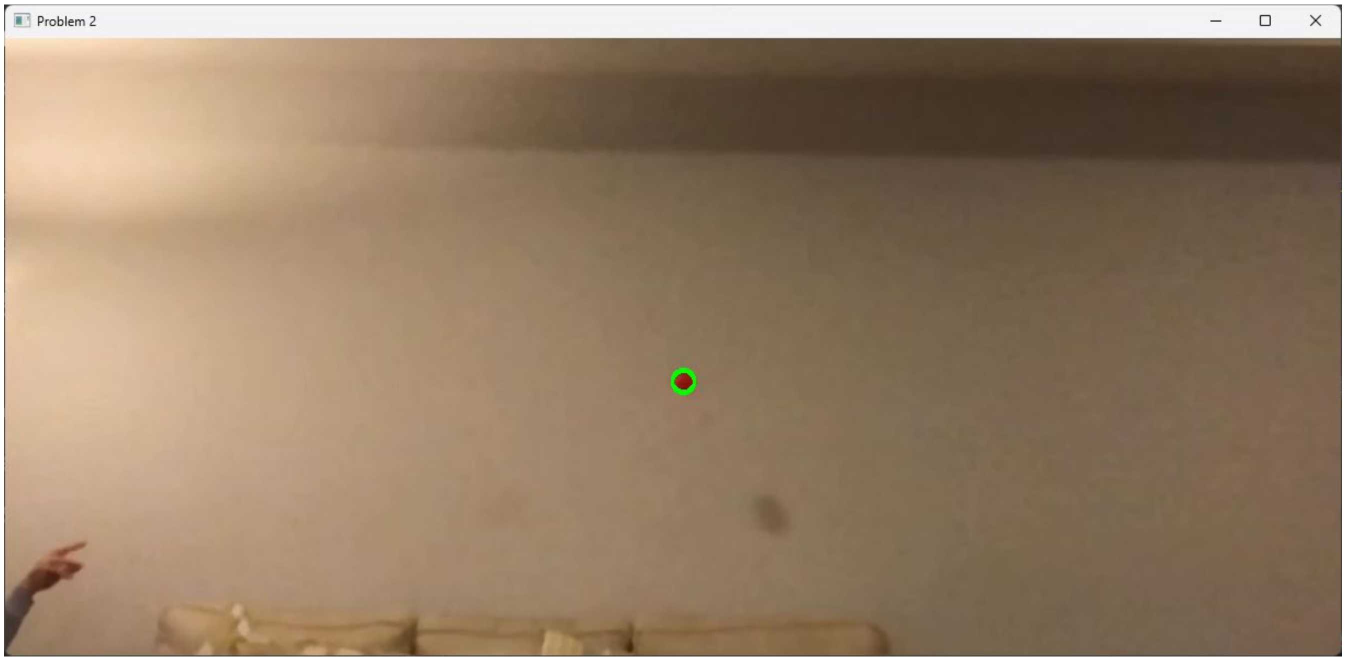

Figure 1Hough Circles detecting the ball in each frame

Pipeline:

The code first reads in a video file called 'ball.mov' using the cv2.VideoCapture function. The video is then processed frame by frame in a while loop until the end of the video is reached or the loop is terminated. For each frame, the code performs the following steps:

1. Convert the frame from the default BGR color space to the HSV color space using the cv2.cvtColor function. This is done to make it easier to identify the ball based on its color.
2. Apply color thresholding to the frame using the cv2.inRange function. This filters out all colors except for the ones that fall within the specified range of values. In this case, the range of values is (0,150,130) to (6,255,255), which corresponds to a shade of red. This produces a binary mask that highlights the pixels that belong to the ball.
3. Find the center point of the ball in the frame by calculating the mean x and y coordinates of the pixels that belong to the ball in the binary mask. If there are no such pixels, then the ball is not in the frame.
4. If the ball is in the frame, define a search region around the known center of the ball. This search region is a square of size 100x100 pixels that is centered on the ball.
5. Convert the search region to grayscale using the cv2.cvtColor function. This reduces the amount of data that needs to be processed and makes it easier to identify edges.
6. Apply Gaussian blur to the grayscale image using the cv2.GaussianBlur function. This is done to reduce noise in the image and make it easier to identify edges.
7. Apply Canny edge detection to the blurred image using the cv2.Canny function. This produces a binary image that highlights the edges in the search region.
8. Perform Hough Transform to detect circles in the edge image using the cv2.HoughCircles function. This is done by scanning the image for circles of various radii and positions. The parameters dp, minDist, param1, param2, minRadius, and maxRadius control the sensitivity and specificity of the circle detection algorithm. maxRadius is taken as 11 as given.
9. If circles are detected in the search region, draw them on the original frame using the cv2.circle function. The coordinates of the circles are converted from the search region coordinates to the frame coordinates by adding the offset of the search region from the frame center.
10. Display the frame with the detected circles using the cv2.imshow function. The loop continues until the user presses the 'q' key or the end of the video is reached.
11. Release the video file and close all windows using the cap.release and cv2.destroyAllWindows functions.

Finally, the code prints a message indicating that Problem 2 is done.

This Pipeline was chosen because previously, for Project 1, I have already computed the center of the ball. Restricting the search region around the center makes it easier to process the video and detect the ball through Hough Transformation.

Problem 3:

Given the photo of a train track, transform the image so that you get a top view of the train tracks, find the average distance between the train tracks for the warped image. Show the intermediate results.

Solution:

Hough Transformation -

Hough Transformation is a computer vision technique used to detect lines and other geometric shapes in an image. It works by representing the lines in the image as points in a parameter space, where each point represents a line in the image. The parameter space is called the Hough space.

To understand the concept of Hough Transformation, let's consider the standard form of a line equation:

$$y = mx + c$$

where m is the slope of the line and
c is the y-intercept.

We can rewrite this equation in polar coordinates as:

$$\rho = x cos\theta + y sin\theta$$

where ρ is the perpendicular distance of the line from the origin
θ is the angle between the line and the x-axis.

We can represent each line in the image as a point in the Hough space, where the x-axis represents θ and the y-axis represents ρ. The Hough space is a 2D parameter space, where each point represents a line in the image. To detect lines using Hough Transformation, we need to follow these steps:

i. **Edge Detection:** We need to first detect edges in the image using techniques like Canny Edge Detection. This gives us a binary image with white pixels representing the edges.
ii. **Hough Accumulator:** We then create a 2D array called the Hough Accumulator that represents the Hough parameter space. Each point (ρ, θ) in this parameter space corresponds to a line in the image. We initialize the accumulator to all zeros.
iii. **Voting:** For each edge pixel in the binary image, we calculate its distance ρ to the origin and the angle θ between the line passing through the origin and the edge pixel and the x-axis. We then increment the corresponding cell in the Hough Accumulator.
iv. **Thresholding:** Once we have accumulated all the votes, we threshold the accumulator to only keep the points that have a high number of votes. These points correspond to the lines in the image.
v. **Line Extraction:** We extract the lines from the thresholded accumulator by converting each point (ρ, θ) back to the image space and drawing the corresponding line.

In the case of the HoughLinesP function used in the code, it is used to detect lines in the image. The function takes as input a binary image and returns a list of lines that have been detected. The algorithm works by performing a voting procedure on the parameters of a line in parameter space, where the parameters are the angle of the line and the distance of the line from the origin. The lines that receive the most votes are selected as the detected lines in the image.

Perspective Transformation –

Perspective transformation is a type of image transformation that maps a 3D object onto a 2D plane, taking into account the effects of perspective. This transformation is commonly used in computer vision and graphics applications, such as in camera calibration, image rectification, and image stitching.

The transformation matrix for a perspective transformation can be defined as:

$$\begin{bmatrix} sx * r11 & sy * r12 & sz * r13 & tx \\ sx * r21 & sy * r22 & sz * r23 & ty \\ sx * r31 & sy * r32 & sz * r33 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where sx, sy, and sz are the scaling factors along the x, y, and z axes, respectively
tx, ty, and tz are the translation values along these axes
r11, r12, r13, r21, r22, r23, r31, r32, and r33 represent the rotation angles along these axes, respectively.

To apply this transformation to a point (x, y, z), we first represent the point as a homogeneous coordinate vector [x, y, z, 1], and then multiply it with the transformation matrix as follows:

$$\begin{bmatrix} sx * r11 & sy * r12 & sz * r13 & tx \\ sx * r21 & sy * r22 & sz * r23 & ty \\ sx * r31 & sy * r32 & sz * r33 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This gives us the transformed point [x', y', z', w'], where

$$x' = (sx*r11*x + sy*r12*y + sz*r13*z + tx*w')/w'$$
$$y' = (sx*r21*x + sy*r22*y + sz*r23*z + ty*w')/w'$$
$$z' = (sx*r31*x + sy*r32*y + sz*r33*z + tz*w')/w'$$
$$w' = (sx*x + sy*y + sz*z + 1).$$

Note that the transformation is represented using a 4x4 matrix, but the last row is always [0, 0, 0, 1], which is used for normalization purposes. Also, the perspective projection can result in distorted images, where objects that are far away appear smaller than objects that are closer to the camera.

This transformation maps the points in the original image to new points in the output image, resulting in a different view of the scene. Perspective transformation is used, for example, to correct for perspective distortion in images, such as when taking a photo of a flat surface at an angle.

The math behind perspective transformation involves finding a 3x3 transformation matrix M, which maps points in the original image to points in the output image. To find M, we need to find four corresponding points in the original image and the output image. These points can be manually selected or detected automatically. The transformation matrix M can then be computed using these points, and the warpPerspective function is used to apply the transformation to the image.

Code:

```
img = cv2.imread('train_track.jpg')  # Read the image file
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)    # Convert the image to grayscale
blur = cv2.GaussianBlur(gray, (191, 191), 2.5)  # Apply Gaussian blur to reduce noise
thresh = cv2.threshold(blur, 210, 255, cv2.THRESH_BINARY)[1]    # Apply a threshold to obtain a binary image
lines = cv2.HoughLinesP(thresh, 1, np.pi/180, threshold=80, minLineLength=850, maxLineGap=1)    # Perform Hough Transform to detect lines
line_img = np.zeros_like(img)   # Create a blank image to draw the lines on

# Find the two lines corresponding to the train tracks
left_line, right_line = None, None
for line in lines:
    x1, y1, x2, y2 = line[0]
    if abs(x2 - x1) > 5 and abs((y2 - y1) / (x2 - x1)) > 0.5: # filter out horizontal and too vertical lines
        if (slope:= x1 - x2) < 0 and (left_line is None or x1 < left_line[0]):  # Find the left line
            left_line = (x1, y1, x2, y2)
```

```python
        elif (right_line is None or x1 > right_line[0]): # Find the right line
            right_line = (x1, y1, x2, y2)

left_x1, left_y1, left_x2, left_y2 = left_line
cv2.line(line_img, (left_x1, left_y1), (left_x2, left_y2), (0, 255, 0), 10)    # Draw the left line
right_x1, right_y1, right_x2, right_y2 = right_line
cv2.line(line_img, (right_x1, right_y1), (right_x2, right_y2), (0, 255, 0), 10)    # Draw the right line

height, width = img.shape[:2]
src_pts = np.array([(left_x2, left_y2),(right_x1, right_y1), (left_x1, left_y1),(right_x2, right_y2)],
np.float32)  # Define the source points
dst_pts = np.array([[0,0], [height, 0], [0, width], [height, width]], np.float32)       # Define the
destination points
M = cv2.getPerspectiveTransform(src_pts, dst_pts)   # The transformation matrix M can be used to warp the
image
warped_img = cv2.warpPerspective(img, M, (height, width))   # Warp the image using the transformation
matrix M

gray_warped = cv2.cvtColor(warped_img, cv2.COLOR_BGR2GRAY)  # Convert the warped image to grayscale
thresh_warped = cv2.threshold(gray_warped, 250, 255, cv2.THRESH_BINARY)[1]    # Apply a threshold to
obtain a binary image
lines_warped = cv2.HoughLinesP(thresh_warped, 1, np.pi/180, 100, minLineLength=100, maxLineGap=10)  #
Perform Hough Transform to detect lines
line_img_warped = np.zeros_like(warped_img)  # Create a blank image to draw the lines on

distances = [abs(y2 - y1) for line in lines_warped for x1, y1, x2, y2 in line]  # Compute the distances
between the lines
for x1, y1, x2, y2 in lines_warped[:, 0]:
    cv2.line(line_img_warped, (x1, y1), (x2, y2), (0, 255, 0), 10)  # Draw the lines
avg_distance = sum(distances) / len(distances)  # Compute the average distance between the lines

# Display the images
fig, axes = plt.subplots(2, 3, figsize=(10, 6)) # Create a figure with a 2x3 grid of Axes
plt.gcf().suptitle('Problem 3 - Average distance between train tracks: {} pixels'.format(avg_distance))  #
Set the window title
axes[0, 0].imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
axes[0, 0].set_title('Original Image')
axes[0, 1].imshow(thresh, cmap='gray')
axes[0, 1].set_title('Binary Image')
axes[0, 2].imshow(line_img, cmap='gray')
axes[0, 2].set_title('Lines')
axes[1, 0].imshow(cv2.cvtColor(warped_img, cv2.COLOR_BGR2RGB))
axes[1, 0].set_title('Warped Image')
axes[1, 1].imshow(thresh_warped, cmap='gray')
axes[1, 1].set_title('Binary Warped Image')
axes[1, 2].imshow(line_img_warped, cmap='gray')
axes[1, 2].set_title('Lines in Warped Image')
plt.show()

print('Problem 3 - Average distance between train tracks:', avg_distance, 'pixels')
```

Output:

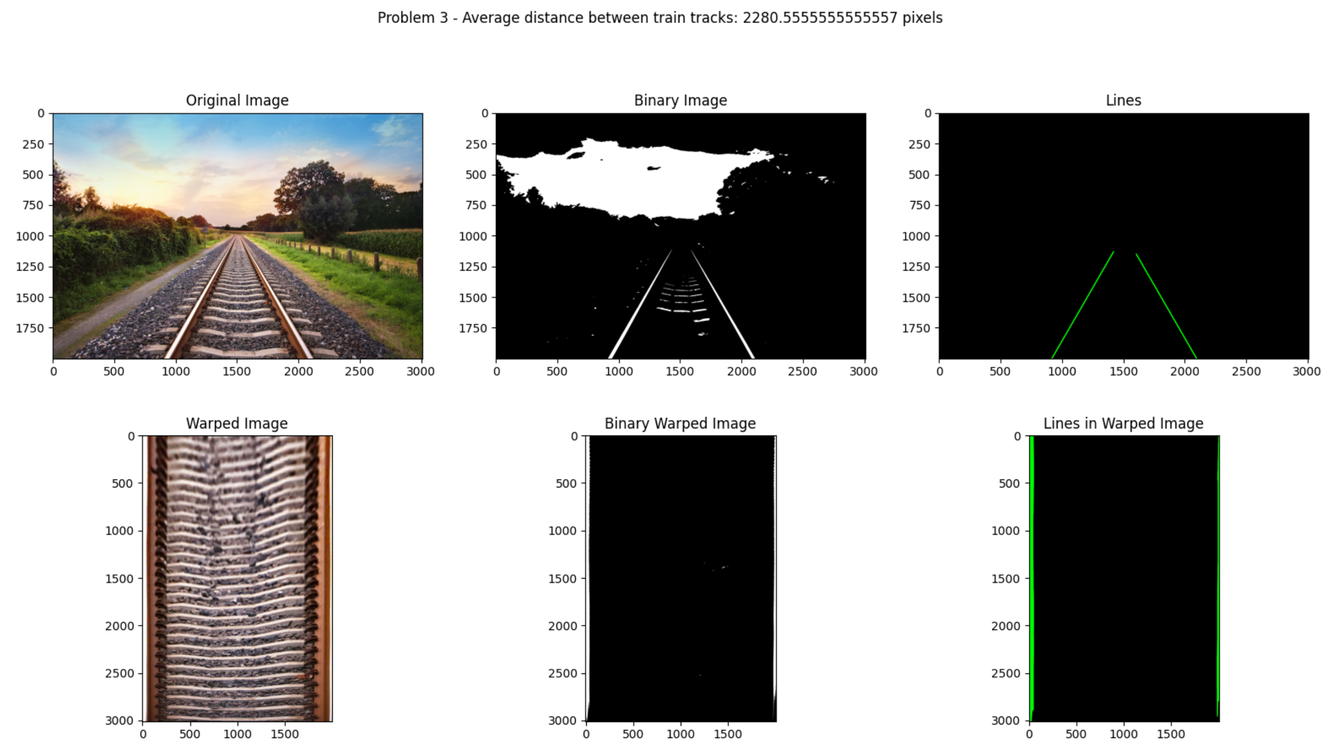

Figure 2 Plot displaying the intermediate stages and the final output (Top View) of the train track

Pipeline:

1. Read the image file train_track.jpg using the OpenCV library function cv2.imread().
2. Convert the image to grayscale using the function cv2.cvtColor() with the argument cv2.COLOR_BGR2GRAY.
3. Apply Gaussian blur to the grayscale image using the function cv2.GaussianBlur() with kernel size (191, 191) and standard deviation 2.5 to reduce noise.
4. Apply a binary threshold to the blurred image using the function cv2.threshold() with threshold value 210, maximum pixel value 255, and thresholding mode cv2.THRESH_BINARY. This produces a binary image where the train tracks are highlighted in white.
5. Perform the Hough Transform on the binary image using the function cv2.HoughLinesP(). This detects lines in the image that correspond to the train tracks.
6. Create a blank image to draw the detected lines on using the function np.zeros_like().
7. Iterate over the detected lines and filter out any lines that are too horizontal or too vertical. Find the left and right lines corresponding to the train tracks based on their slopes and x-coordinates.
8. Draw the detected lines on the blank image using the function cv2.line().
9. Define the source points and destination points for a perspective transformation that will warp the image such that the train tracks appear straight. I used the coordinates of the left and right lines as source points and the flipped height and width of the image as destination points.
10. Compute the perspective transformation matrix using the function cv2.getPerspectiveTransform() with the source and destination points as arguments.
11. Warp the image using the perspective transformation matrix using the function cv2.warpPerspective().
12. Convert the warped image to grayscale using the function cv2.cvtColor().
13. Apply a binary threshold to the warped grayscale image using the function cv2.threshold() with threshold value 250, maximum pixel value 255, and thresholding mode cv2.THRESH_BINARY.
14. Perform the Hough Transform on the binary warped image using the function cv2.HoughLinesP(). This detects lines in the image that correspond to the train tracks.
15. Create a blank image to draw the detected lines on using the function np.zeros_like().
16. Compute the distances between the detected lines.
17. Iterate over the detected lines and draw them on the blank image using the function cv2.line().

18. Compute the average distance between the detected lines by summing the distances and dividing by the number of distances.
19. Display the original image, binary image, and detected lines in the original image, as well as the warped image, binary warped image, and detected lines in the warped image using the function plt.subplots(). The average distance between the detected lines is displayed as the window title.
20. Print the average distance between the detected lines.

This Pipeline was chosen because using Hough Lines to identify the track makes it easier to compute the coordinates that can be used as source points when performing Perspective Transformation to obtain the top view of the train tracks.

Problem 4:

Detect each hot air balloon in a given image of hot air balloons. Find the number of balloons automatically. The final results should show each hot air balloon labeled with different colors.
Note: Do not worry about resolving occlusion, occluded balloons can be combined as one.

Solution:

Thresholding –

Thresholding is a common image processing technique used to create a binary image from a grayscale or color image. The goal of thresholding is to separate the foreground (object of interest) from the background based on pixel intensity values. In a grayscale image, each pixel has a value between 0 and 255, where 0 is black and 255 is white. Thresholding sets all pixels below a certain threshold value to black (0) and all pixels above the threshold value to white (255). The result is a binary image where the object of interest is white and the background is black.

Contours –

Contours are curves that connect continuous points along the boundary of an object in an image. Contours can be used to identify the shape and location of objects in an image.
The chain code algorithm is a method for tracing the boundary of an object in an image by following the edge pixels of the object. The algorithm works by examining each pixel in the image and determining whether it belongs to the object or the background. Once a pixel belonging to the object is found, the algorithm traces the boundary of the object by following the edge pixels.

The chain code algorithm works as follows:

1. Starting from a point on the object boundary, move clockwise around the boundary, pixel by pixel.
2. At each pixel, examine the eight neighboring pixels (called the 8-connected neighborhood) to determine the direction in which the boundary continues.
3. Assign a code to the direction from the current pixel to the next pixel on the boundary. The code can be a number or a letter that represents the direction. For example, if the next pixel is to the right of the current pixel, the code might be 0. If the next pixel is to the upper right, the code might be 1.
4. Continue tracing the boundary by moving from the current pixel to the next pixel in the direction specified by the code.
5. Repeat steps 2-4 until the boundary has been traced back to the starting point.
6. The resulting sequence of codes represents the chain code for the object boundary. The chain code can be used to reconstruct the object boundary or to perform various image processing tasks such as object recognition and tracking.

In OpenCV, the chain code algorithm is used as part of the contour detection process. The algorithm starts with a binary image and finds the contours by tracing the boundaries of the objects in the image using the chain code algorithm.
The cv2.findContours function can also take optional arguments that specify the contour retrieval mode and the contour approximation method. The contour retrieval mode determines how the contours are retrieved from the

binary image. The two most common retrieval modes are cv2.RETR_EXTERNAL, which retrieves only the external contours, and cv2.RETR_TREE, which retrieves all contours and organizes them in a hierarchical tree structure. The contour approximation method determines how the contours are approximated. The most common approximation method is cv2.CHAIN_APPROX_SIMPLE, which approximates the contour boundary with fewer points by removing unnecessary points along straight line segments.

Code:

```
image = cv2.imread('hotairbaloon.jpg')  # Read the image
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)  # Convert the image to grayscale
blur = cv2.GaussianBlur(gray, (5, 5), 0)    # Apply a Gaussian blur to the grayscale image
_, thresh = cv2.threshold(blur, 110, 255, cv2.THRESH_BINARY)    # Apply thresholding to create a binary
image
contours, _ = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)  # Find contours in the
binary image

# Draw bounding boxes around each contour and count the number of balloons
count = 0
for contour in contours:
    area = cv2.contourArea(contour)
    if area > 5000: # Only consider contours with area greater than 5000
        color = tuple(map(int, np.random.randint(0, 255, size=3)))  # Generate a random color
        x, y, w, h = cv2.boundingRect(contour)  # Get the bounding box coordinates
        cv2.rectangle(image, (x, y), (x + w, y + h), color, 2)  # Draw the bounding box
        cv2.putText(image, f'Balloon {count}', (x, y-10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, color, 2)  #
Label the bounding box
        count += 1

# Display the final image with the count of balloons
cv2.putText(image, f'Total Balloons: {count-1}', (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)
cv2.imshow('Problem 4', cv2.resize(image, (1800, int(image.shape[0] * 1800 / image.shape[1])))) # Resize
the image to fit the screen
cv2.waitKey(0)
cv2.destroyAllWindows()
print('Problem 4 - Total Number of balloons:', count,'\n')
```

Output:

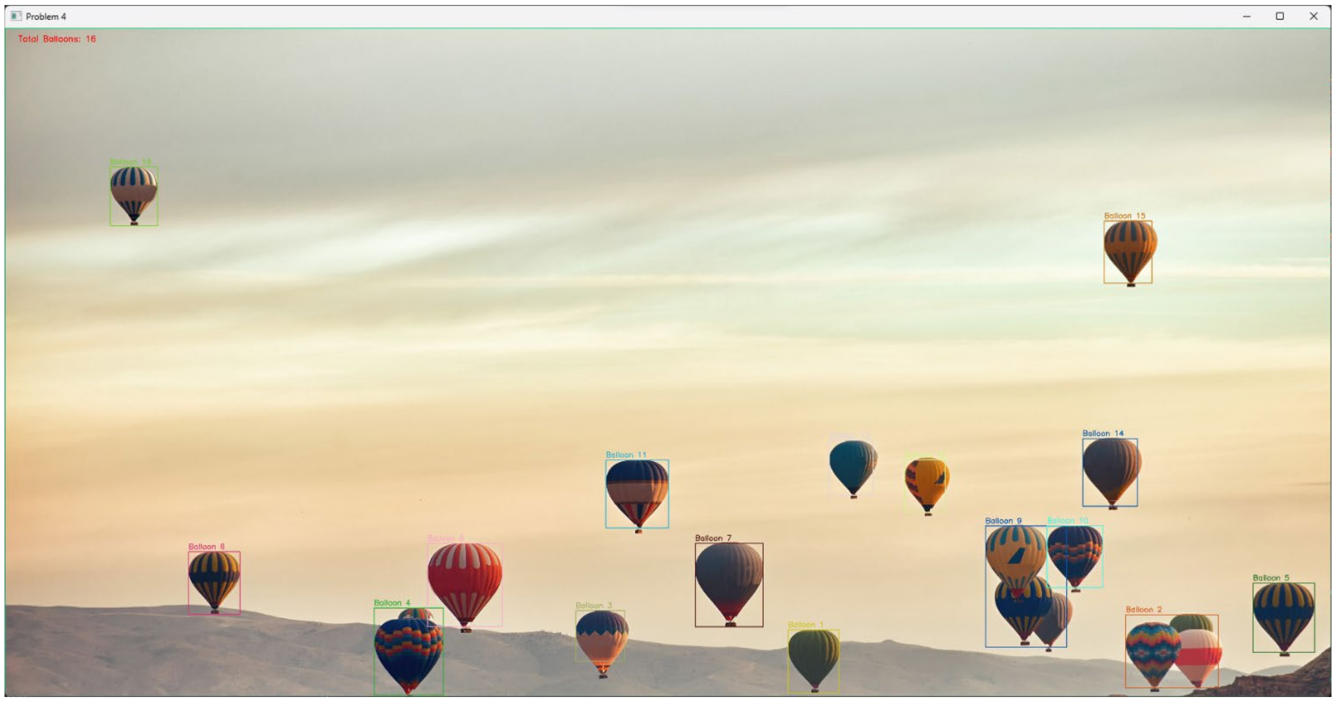

Figure 3 Figure displaying all the detected balloons with the count in different colors

Pipeline:

1. The code reads the input image 'hotairbaloon.jpg' using the cv2.imread() function and stores it in the 'image' variable.
2. The cv2.cvtColor() function is used to convert the image to grayscale and the result is stored in the 'gray' variable.
3. The cv2.GaussianBlur() function is used to apply a Gaussian blur filter with a kernel size of (5, 5) to the grayscale image. The result is stored in the 'blur' variable.
4. The cv2.threshold() function is used to apply thresholding to the blurred image to create a binary image. The threshold value is set to 110, which means that all pixel values above 110 are set to 255 (white) and all pixel values below 110 are set to 0 (black). The result is stored in the 'thresh' variable.
5. The cv2.findContours() function is used to find contours in the binary image. The mode is set to cv2.RETR_TREE, which means that all contours are retrieved and a full hierarchy is computed. The method is set to cv2.CHAIN_APPROX_SIMPLE, which means that only the endpoints of the contour are stored. The result is stored in the 'contours' variable.
6. A for loop is used to iterate through each contour in the 'contours' variable. The cv2.contourArea() function is used to compute the area of the contour and the area is stored in the 'area' variable.
7. If the area of the contour is greater than 5000, a bounding box is drawn around the contour using the cv2.boundingRect() function. The coordinates of the bounding box are stored in the variables 'x', 'y', 'w', and 'h'. A random color is generated using the np.random.randint() function and the color is stored in the 'color' variable. The cv2.rectangle() function is used to draw the bounding box around the contour, and the cv2.putText() function is used to label the bounding box with the text 'Balloon {count}'.
8. The count variable is incremented by 1 for each contour that meets the condition of having an area greater than 5000.
9. After all contours have been processed, the total number of balloons is displayed on the image using the cv2.putText() function. The final image is displayed using the cv2.imshow() function, which takes the 'image' variable as input. The image is also resized to fit the screen using the cv2.resize() function.
10. The cv2.waitKey() function waits for a key event, and the cv2.destroyAllWindows() function closes all windows created by OpenCV.
11. Finally, the total number of balloons is printed to the console using the print() function.

This Pipeline was chosen because using contour detection gives flexibility in detecting balloons of different shapes and sizes, which can then be bounded by a rectangles of different colors for identification.

Terminal Output for Problems 2, 3, 4:

```
PS C:\Users\manda\OneDrive - University of Maryland\Perception For Autonmous Robots\Mid-Term> & "C:/Program
Files/Python311/python.exe" "c:/Users/manda/OneDrive - University of Maryland/Perception For Autonmous Robots/Mid-
Term/midterm.py"

Problem 2 - Done!
Problem 3 - Average distance between train tracks: 2280.5555555555557 pixels
Problem 4 - Total Number of balloons: 17
```

Problem 5:

Given these numbers below:
25 13 2 11 4 6 15 22
Apply k-means to get 3 clusters. Show detailed work.

Solution:

K-means clustering is a popular unsupervised machine learning algorithm used for data clustering and pattern recognition. It is a simple yet powerful algorithm that divides a dataset into K clusters based on the similarity of the data points. The K in K-means represents the number of clusters to be created.

The algorithm starts by randomly selecting K points from the dataset to act as the initial centroids for the clusters. Each data point is then assigned to the nearest centroid based on the Euclidean distance between the data point and the centroid. This process is repeated iteratively until the centroids no longer move significantly.

The main objective of the K-means algorithm is to minimize the sum of squared distances between the data points and their respective centroids. This objective function is known as the within-cluster sum of squares (WCSS) and is used to determine the optimal number of clusters for the dataset.

K-means clustering has several advantages over other clustering algorithms, including its simplicity, speed, and scalability. It can be applied to a wide range of datasets and is particularly useful for large datasets with many dimensions. K-means clustering is also widely used in image processing, text mining, and recommendation systems.

However, K-means clustering also has some limitations. It is sensitive to the initial placement of the centroids and may converge to a suboptimal solution. It is also not suitable for datasets with non-linear boundaries or clusters of varying sizes and densities.

In conclusion, K-means clustering is a powerful algorithm for data clustering and pattern recognition. It is widely used in machine learning and data science and is an essential tool for data exploration and analysis.

Step 1: Initialize cluster centers

We need to start by randomly choosing three initial cluster centers from the data points. Let's choose 4, 13, and 22 as our initial centers.

Step 2: Assign points to clusters

Next, we assign each data point to the cluster whose center is closest to it. Using the Euclidean distance formula, we get the following assignments:

Check for Cluster 1 (4)	**Check for Cluster 2 (13)**	**Check for Cluster 3 (22)**
25-4 = 21	25-13 = 12	25-22 = 3
13-4 = 9	13-13 = 0	|13-22| = 9
|2-4| = 2	|2-13| = 11	|2-22| = 20
11-4 = 7	|11-13| = 2	|11-22| = 11
4-4 = 0	13-4 = 9	|4-22| = 18
6-4 = 2	|6-13| = 7	|6-22| = 16
15-4=11	15-13 = 2	|15-22| = 7
22-4=18	22-13 = 9	22-22 = 0

By allocating the numbers closest to each centroid, we obtain the following clusters -

Cluster 1: {2, 4, 6}

Cluster 2: {11, 13, 15}

Cluster 3: {22, 25}

Step 3: Update cluster centers

We now calculate the mean of each cluster and use those values as the new cluster centers:

Cluster 1: $Mean = \frac{2+4+6}{3} = 4$

Cluster 2: $Mean = \frac{11+13+15}{3} = 13$

Cluster 3: $Mean = \frac{22+25}{2} = 23.5$

Step 4: Repeat the above steps till the cluster alignments stop changing

Check for Cluster 1 (4)	**Check for Cluster 2 (13)**	**Check for Cluster 3 (23.5)**
25-4 = 21	25-13 = 12	25-23.5 = 1.5
13-4 = 9	13-13 = 0	|13-23.5| = 10.5
|2-4| = 2	|2-13| = 11	|2-23.5| = 21.5
11-4 = 7	|11-13| = 2	|11-23.5| = 12.5
4-4 = 0	13-4 = 9	|4-23.5| = 19.5

6-4 = 2	|6-13| = 7	|6-23.5| = 17.5
15-4=11	15-13 = 2	|15-23.5| = 8.5
22-4=18	22-13 = 9	|22-23.5| = 1.5

By allocating the numbers closest to each centroid, we obtain the following clusters -
Cluster 1: {2, 4, 6}
Cluster 2: {11, 13, 15}
Cluster 3: {22, 25}

As we get the same assignments and centers as in the previous iteration, so the algorithm has converged. Therefore, the final clusters with k = 3 are:

Cluster 1: {2, 4, 6}
Cluster 2: {11, 13, 15}
Cluster 3: {22, 25}

This may change based on the values of the initial centroids.