IPCV_Project_Andrea_Perna

February 20, 2025

1 Robot Navigation Project - Image Processing and Computer Vision - a.y. 2024/25

Professor: Luigi Di Stefano

Student: Andrea Perna - Automation Engineering

1.1 1) Introduction

Abstract This project focuses on developing an obstacle detection system for autonomous navigation using stereo vision. Given synchronized video sequences from a stereo camera mounted on a moving vehicle, the system estimates the distance to the obstacle (i.e., the chessboard) by computing disparity maps through two methods: the dense Semi-Global Block Matching (SGBM) algorithm and a sparse keypoint-based approach leveraging the chessboard's corners. The latter method, applied to chessboard patterns in the scene, enhances stability by reducing noise and fluctuations in distance estimation compared to dense disparity mapping. The project includes real-time distance monitoring, chessboard dimension estimation, and safety alerts when the vehicle approaches the critical distance, providing reliable input for collision avoidance.

Dataset The dataset comprises synchronized stereo videos captured in a cluttered scenario. A centrally placed chessboard pattern is a key reference for computing disparity maps and obstacle distances.

1.2 2) Settings

Libraries

[389]:
```
#import libraries
import cv2
import numpy as np
import matplotlib
from matplotlib import pyplot as plt
import math
import os
import plotly.graph_objects as go

#suppress warnings
matplotlib.use('Agg')
```

Parameters Control Parameters

[390]:
```
method = "Keypoints"  #stereo matching method: "SGBM" for dense matching or␣
 ↪"Keypoints" for sparse, chessboard-based matching.
robust_mean = True  #if True, the median is used instead of the mean for␣
 ↪estimating the main disparity, improving robustness to noise.
corners_refinement = True #if True, it allows refinement of extracted corners␣
 ↪ofthe chessboard
enable_plots = True #if True, it enables metrics visualization and saving
generate_maps = True #if True, enables disparity and depth map creation and␣
 ↪storage
threshold = False #if True, it plots the width and height' plots with visible␣
 ↪thresholds
stereo_block_size = 15  #size of the block used to compare pixel intensities␣
 ↪between left and right images in StereoSGBM algorithm.
central_window_radius = 10  #used to crop the disparity map to focus on a␣
 ↪smaller region in the center of the frame for efficient noise reduction.
```

Configuration Parameters

[391]:
```
crop_height = 0  #initial vertical crop of the disparity region, allowing␣
 ↪adjustments for focusing on central areas.
crop_width = 0  #initial horizontal crop of the disparity region, ensuring that␣
 ↪computations focus on the region of interest.
disparity_SGBM = 0 #initialization of the disparity value between the two images
first_frame = True #boolean flag indicating the first frame for computing the␣
 ↪initial disparity
```

Paths Configuration

[392]:
```
base_dir = os.getcwd()
output_directory = os.path.join(base_dir, "Project_Outcomes")
left_video_path = os.path.join(base_dir, "robot-navigation-video", "robotL.avi")
right_video_path = os.path.join(base_dir, "robot-navigation-video", "robotR.
 ↪avi")
output_video_path = os.path.join(output_directory, "output_video_final.avi")
```

Camera and Chessboard Parameters

[393]:
```
#camera parameters
f = 567.2  #focal length in pixels
b = 92.226  #baseline distance between stereo cameras in mm

#chessboard parameters
chessboard_grid = (8, 6)  #chessboard grid size
real_width = 125  #real chessboard width (mm)
real_height = 178  #real chessboard height (mm)
```

Visualization Parameters

[394]:
```
FONT = cv2.FONT_HERSHEY_DUPLEX #font type
FONT_SCALE = 0.6 #general font scale
ALERT_FONT_SCALE = 0.8 #font scale of alert message
TEXT_COLOR = (255, 255, 255)  #white text
SAFE_COLOR = (0, 255, 0)  #green for safe navigation
RED_ALERT_COLOR = (0, 0, 255)  #red for warnings
SHADOW_COLOR = (0, 0, 0)  #shadow for contrast
THICKNESS = 1 #font tickness
alarm_counter = 0  #counter for triggering warnings
min_dist = 800  #minimum safe distance in mm

#font alternatives
cv2.FONT_HERSHEY_SIMPLEX   #a basic sans-serif font, commonly used for␣
 ↪general-purpose text due to good readability.
cv2.FONT_HERSHEY_COMPLEX   #a serif-style font with more detail, suitable for␣
 ↪formal or stylized text.
cv2.FONT_HERSHEY_TRIPLEX   #a A bold serif-style font, ideal for emphasis and␣
 ↪headings.
cv2.FONT_HERSHEY_PLAIN     #a minimalist font with very thin, simple text, good␣
 ↪for compact or subtle labeling.
cv2.FONT_HERSHEY_DUPLEX    #a thicker version of the basic sans-serif font,␣
 ↪providing better visibility while maintaining readability.
```

[394]:
```
2
```

Metrics Storage and Initialization

[395]:
```
metrics_data = { #data structure to store key metrics

    #distances
    "distances_keypoints": [],
    "distances_SGBM": [],

    #chessboard dimensions
    "widths_keypoints": [],
    "heights_keypoints": [],
    "widths_SGBM": [],
    "heights_SGBM": [],

    #angles estimations
    "angles_keypoints": [],
    "angles_SGBM": [],

    #disparity
    "disparity_maps": [],
    "depth_maps": []
```

```
}
```

1.3 3) Functions

1.3.1 Initialization Functions

Video Initialization This function initializes the video processing pipeline, it opens the left and right synchronized video files, checks for loading issues, and retrieves key properties like frame dimensions and frame rate. It also initializes an output video writer to save the processed frames as an AVI file, which will contain the algorithm's results.

[396]:
```
def video_initialization(left_video_path, right_video_path, output_video_path):

    #open the left and right video streams
    left_video = cv2.VideoCapture(left_video_path)
    right_video = cv2.VideoCapture(right_video_path)

    #check if videos were successfully opened
    if not (left_video.isOpened() and right_video.isOpened()):
        raise IOError("ERROR: Problem encountered during the opening of the␣
↪videos.")
    else: print("Videos opened correctly.\n")

    #retrieve video properties (assume both videos have the same properties)
    frame_width = int(left_video.get(cv2.CAP_PROP_FRAME_WIDTH))
    frame_height = int(left_video.get(cv2.CAP_PROP_FRAME_HEIGHT))
    frame_rate = left_video.get(cv2.CAP_PROP_FPS)
    video_length = int(left_video.get(cv2.CAP_PROP_FRAME_COUNT))

    #initialize the output video writer
    fourcc = cv2.VideoWriter_fourcc(*'DIVX')
    output_video = cv2.VideoWriter(output_video_path, fourcc, frame_rate,␣
↪(frame_width, frame_height))

    print(f"Video Properties: Frame_Width={frame_width},␣
↪Frame_Height={frame_height}, FPS={frame_rate}, Video_Length={video_length}")

    return left_video, right_video, output_video, frame_width, frame_height,␣
↪frame_rate, video_length
```

Frame Preprocessing This function reads and validates frames from the left and right video streams. Valid frames are converted to grayscale for stereo matching and keypoint detection. If frames are missing or corrupted, they are skipped to maintain smooth and uninterrupted execution of the video processing pipeline.

[397]:
```
def frame_preprocessing(left_video, right_video, frame_number):
```

```
    #read frames from both videos
    rL, frameL = left_video.read()
    rR, frameR = right_video.read()

    #check if frames are missing or corrupted
    if not rL or not rR or frameL is None or frameR is None:
        print(f"\nSkipping frame {frame_number}: Unable to read video frames.")
        return None  #indicate that frames should be skipped

    #convert frames to grayscale
    frameL_gray = cv2.cvtColor(frameL, cv2.COLOR_BGR2GRAY)
    frameR_gray = cv2.cvtColor(frameR, cv2.COLOR_BGR2GRAY)

    return frameL, frameR, frameL_gray, frameR_gray
```

1.3.2 Visualization Functions

Save Map This function saves the disparity map of a given video frame as an image for visualization. The map is normalized to range [0, 255] to enhance visibility and saved in the specified output directory.

[398]:
```
def save_map(map_data, map_type, frame_number, output_directory):

    #normalize the map to [0, 255] for proper visualization
    normalized_map = cv2.normalize(map_data, None, 0, 255, norm_type=cv2.
↪NORM_MINMAX)
    normalized_map = np.uint8(normalized_map)

    #create the output directory if it doesn't exist
    os.makedirs(output_directory, exist_ok=True)

    #generate the file name
    file_name = os.path.join(output_directory,␣
↪f"{map_type}_map_frame{frame_number}.png")

    #save the image
    cv2.imwrite(file_name, normalized_map)
    print(f"Saved {map_type} map as {file_name}")
```

Display Information This function overlays estimated information on each video frame, such as obstacle distance, chessboard dimensions, angle estimates, and progress status. If the distance to the obstacle is below the safety threshold, a red visual alarm is triggered and managed by the alarm counter. Chessboard corners are shown when detected, and the processed frame is written to the output video.

[399]:
```

```

```
def display_information(frame, alarm_counter, distance_keypoint, min_dist,␣
↪chessboard_width, chessboard_height, angle_tau, chessboard_grid, corners,␣
↪found, frame_number, video_length, output_video):

    #subfunction for drawing text with shadow
    def draw_text(text, position, color=TEXT_COLOR, font_scale=FONT_SCALE,␣
↪thickness=THICKNESS):

        """
        Draw text with a shadow to improve readability on any background.
        """
        shadow_offset = (position[0] + 2, position[1] + 2)  # Slight offset for␣
↪shadow effect
        cv2.putText(frame, text, shadow_offset, FONT, font_scale, SHADOW_COLOR,␣
↪thickness + 1, cv2.LINE_AA)  # Shadow
        cv2.putText(frame, text, position, FONT, font_scale, color, thickness,␣
↪cv2.LINE_AA)  # Main text

    #trigger visual alarm if distance is below the threshold
    if distance_keypoint < min_dist:

        frame[:, :, 1] //= [4, 2, 3][((alarm_counter // 2) % 6) % 3]  #reduce␣
↪green
        frame[:, :, 0] //= [4, 2, 3][((alarm_counter // 2) % 6) % 3]  #reduce␣
↪blue
        alarm_counter += 1  #update alarm counter

        #warning message in bold red
        draw_text(f"ALERT - OBSTACLE AHEAD: {np.around(distance_keypoint /␣
↪1000, 3)} m", (20, 40), RED_ALERT_COLOR, ALERT_FONT_SCALE, 2)

    #safe navigation message in bold green
    else: draw_text(f"SAFE ROBOT NAVIGATION: {np.around(distance_keypoint /␣
↪1000, 3)} m", (20, 40), SAFE_COLOR, ALERT_FONT_SCALE, 2)

    #header for estimated information
    draw_text(f"ESTIMATED INFORMATION ({method})", (20, 110))

    #display chessboard and angle information
    info_texts = [
        f"-Minimum Safety Distance: {np.around(min_dist/1000, 3)} m",
        f"-Chess width: {np.around(chessboard_width, 3)} mm",
        f"-Chess height: {np.around(chessboard_height, 3)} mm",
        f"-Angle: {np.around(angle_tau, 3)} DEG",
    ]
```

```
    #draw all information lines with consistent spacing
    for i, text in enumerate(info_texts): draw_text(text, (20, 140 + i * 30))

    #display progress information
    progress_text = f"Progress: {(frame_number / video_length) * 100:.1f}% |␣
↪Frame {frame_number + 1}/{video_length}"
    draw_text(progress_text, (20, frame.shape[0] - 20), font_scale=FONT_SCALE)

    #draw chessboard corners if detected
    if found: cv2.drawChessboardCorners(frame, chessboard_grid, corners, found)

    #write frame to output video
    output_video.write(frame)

    return alarm_counter  #return updated alarm counter
```

Metrics This function analyzes and compares the performance of the keypoint and SGBM methods by plotting distance, width, and height measurements across frames. It also computes the relative percentage differences between the two methods and highlights keyframes for disparity map visualization. The plots and some representative disparity maps are saved in the output directory.

[400]:
```
def metrics(metrics_data, output_directory, min_dist):

    ########################################
    ######### Data Preprocessing ###########
    ########################################

    #convert distances into NumPy arrays
    distances_keypoints = np.array(metrics_data["distances_keypoints"])[1:] /␣
↪1000  #skip first frame
    distances_SGBM = np.array(metrics_data["distances_SGBM"])[1:] / 1000  #skip␣
↪first frame
    min_dist_m = min_dist / 1000  #convert min_dist to meters

    #convert widths, heights and angles to NumPy arrays
    widths_keypoints = np.array(metrics_data["widths_keypoints"])[1:]
    widths_keypoints, heights_keypoints = np.
↪array(metrics_data["widths_keypoints"][1:]), np.
↪array(metrics_data["heights_keypoints"][1:])
    widths_SGBM, heights_SGBM = np.array(metrics_data["widths_SGBM"][1:]), np.
↪array(metrics_data["heights_SGBM"][1:])
    angles_keypoints = np.array(metrics_data["angles_keypoints"][1:])
    #angles_SGBM = np.array(metrics_data["angles_SGBM"])[1:]

    #compute relative percentage differences for distances
```

```
    relative_distance_difference = (np.abs(distances_SGBM - distances_keypoints) / distances_keypoints) * 100
    avg_relative_distance_diff = np.mean(relative_distance_difference)

    ########################################
    ######## Depth & Disparity Maps ########
    ########################################

    #identify representative frames (25%, 50%, 75%)
    total_frames = len(distances_keypoints)
    representative_frames = [total_frames // 4, total_frames // 2, 3 * total_frames // 4]

    if generate_maps:

        for frame_num in representative_frames:

            #extract the full maps
            full_disparity_map = metrics_data["disparity_maps"][frame_num]
            full_depth_map = metrics_data["depth_maps"][frame_num]

            #to save maps with overlayed text
            save_map(full_disparity_map, "Disparity Map", frame_num + 1, output_directory)
            save_map(full_depth_map, "Depth Map", frame_num + 1, output_directory)

    ########################################
    ############ Metrics Plots #############
    ########################################

    if enable_plots:

        # -------- Plot 1: Distance Comparison --------
        frames = np.arange(1, len(distances_keypoints) + 1)  #start from frame 1

        plt.figure(figsize=(10, 6))
        plt.plot(frames, distances_keypoints, label='Keypoints Distance', color='blue', linewidth=2)
        plt.plot(frames, distances_SGBM, label='SGBM Distance', color='red', linestyle='dashed', linewidth=2)
        plt.axhline(y=min_dist_m, color='green', linestyle='dotted', linewidth=2, label=f'Min Safe Distance ({min_dist_m:.2f} m)')
        plt.xlabel("Frame Number")
        plt.ylabel("Distance (m)")
```

```
        if robust_mean: plt.title("Distance Comparison Across Frames [MEDIAN␣
↪DISPARITY]")
        else: plt.title("Distance Comparison Across Frames [MEAN DISPARITY]")
        plt.legend()
        plt.grid(True)
        output_file = f"{output_directory}/distance_comparison.png"
        plt.savefig(output_file, bbox_inches='tight')
        plt.close()
        print(f"Distance comparison plot saved to {output_file}")

        # -------- Plot 2: Chessboard Width Comparison --------
        plt.figure(figsize=(10, 6))
        plt.plot(frames, widths_keypoints, label='Keypoints Width',␣
↪color='blue', linewidth=2)
        plt.plot(frames, widths_SGBM, label='SGBM Width', color='red',␣
↪linestyle='dashed', linewidth=2)
        if threshold: plt.axhline(y=real_width, color='green',␣
↪linestyle='dotted', linewidth=2, label=f'Real Width ({real_width:.2f} mm)')
        plt.xlabel("Frame Number")
        plt.ylabel("Width (mm)")
        plt.title("Chessboard Width Comparison")
        plt.legend()
        plt.grid(True)
        output_file = f"{output_directory}/width_comparison.png"
        plt.savefig(output_file, bbox_inches='tight')
        plt.close()
        print(f"Chessboard width comparison plot saved to {output_file}")

        # -------- Plot 3: Chessboard Height Comparison --------
        plt.figure(figsize=(10, 6))
        plt.plot(frames, heights_keypoints, label='Keypoints Height',␣
↪color='blue', linewidth=2)
        plt.plot(frames, heights_SGBM, label='SGBM Height', color='red',␣
↪linestyle='dashed', linewidth=2)
        if threshold: plt.axhline(y=real_height, color='green',␣
↪linestyle='dotted', linewidth=2, label=f'Real Height ({real_height:.2f} mm)')
        plt.xlabel("Frame Number")
        plt.ylabel("Height (mm)")
        plt.title("Chessboard Height Comparison")
        plt.legend()
        plt.grid(True)
        output_file = f"{output_directory}/height_comparison.png"
        plt.savefig(output_file, bbox_inches='tight')
        plt.close()
        print(f"Chessboard height comparison plot saved to {output_file}")
```

```
        # -------- Plot 4: Relative Distance Difference --------
        plt.figure(figsize=(10, 6))
        plt.plot(frames, relative_distance_difference, label='Relative Distance␣
↪Difference (%)', color='purple', linewidth=2)
        plt.xlabel("Frame Number")
        plt.ylabel("Relative Difference (%)")
        plt.title(f"Relative Distance Difference (Avg:␣
↪{avg_relative_distance_diff:.2f}%)")
        plt.legend()
        plt.grid(True)
        output_file = f"{output_directory}/relative_distance_difference.png"
        plt.savefig(output_file, bbox_inches='tight')
        plt.close()
        print(f"Relative distance difference plot saved to {output_file}")

        # -------- Plot 5: Relative Distance Difference (Semi-log scale)␣
↪--------
        plt.figure(figsize=(10, 6))
        plt.semilogy(frames, relative_distance_difference, label='Relative␣
↪Distance Difference (%)', color='purple', linewidth=2)
        plt.xlabel("Frame Number")
        plt.ylabel("Relative Difference (%)")
        plt.title(f"Relative Distance Difference (Avg:␣
↪{avg_relative_distance_diff:.2f}%)")
        plt.legend()
        plt.grid(True, which="both", linestyle='--')  #grid for both major and␣
↪minor ticks
        output_file = f"{output_directory}/relative_distance_difference_log.png"
        plt.savefig(output_file, bbox_inches='tight')
        plt.close()
        print(f"Relative distance difference plot (log scale) saved to␣
↪{output_file}")

        # -------- Plot 6: Angle Plot (Keypoints) --------
        plt.figure(figsize=(10, 6))
        plt.plot(frames, angles_keypoints, label="Chessboard Angles",␣
↪color='blue', linestyle='-')
        #plt.plot(angles_SGBM, label="SGBM Angle", color='red', linestyle='--')
        plt.xlabel("Frame Number")
        plt.ylabel("Angle (degrees)")
        plt.title("Chessboard Angles Across Frames")
        plt.legend()
        plt.grid(True)
        plt.savefig(os.path.join(output_directory, "angles.png"))
        plt.close()
```

Point Cloud (Sketchfab) The following function generates a **3D point cloud** from a given disparity map and saves it as a PLY (Polygon File Format) file. The disparity map, computed from a stereo camera setup, provides depth information, which is then converted into 3D world coordinates using the known camera's intrinsic parameters: focal length (f) and baseline distance (b). Key steps follow:

1. **Depth Estimation:** The disparity map is used to compute the depth $Z = (f * b)/d$.
2. **Point Projection:** Using known camera intrinsics, each pixel's (X, Y, Z) coordinates in 3D space are determined.
3. **Color Association:** The color of each 3D point is taken from the corresponding pixel in the left camera frame.
4. **PLY File Saving:** The computed 3D points along with their color information are written to a PLY file (visualized with Sketchfab).

The function is indeed using the perspective projection equation, but working with its inverse transformation to go from 2D pixel coordinates (u, v) back to 3D world coordinates (X, Y, Z).

$$X = \frac{(u - c_x) \cdot Z}{f}, \quad Y = \frac{(v - c_y) \cdot Z}{f}, \quad Z = \frac{f \cdot b}{d}$$

[401]:
```
def generate_point_cloud(disparity_map, frameL, f, b,␣
↪output_filename="point_cloud.ply"):

    #ensure proper scaling
    disparity_map = disparity_map.astype(np.float32) / 16.0

    #apply a bilateral filter for smooth depth estimation
    disparity_map = cv2.bilateralFilter(disparity_map, d=5, sigmaColor=50,␣
↪sigmaSpace=50)

    #mmage size
    h, w = disparity_map.shape

    #compute depth (Z) from disparity
    valid_mask = disparity_map > 0.8  #ignore invalid disparities
    Z = np.zeros_like(disparity_map, dtype=np.float32)
    Z[valid_mask] = (f * b) / disparity_map[valid_mask]

    #define the piercing point
    cx = w/2
    cy = h/2

    #compute 3D coordinates
    v, u = np.indices((h, w))
    X = (u - cx) * Z / f
    Y = (v - cy) * Z / f

    #apply mask and reshape to valid points
```

```
    points = np.column_stack((X[valid_mask], Y[valid_mask], Z[valid_mask]))

    #extract color information from the image
    colors = cv2.cvtColor(frameL, cv2.COLOR_BGR2RGB)[valid_mask] if frameL is␣
↪not None else np.zeros_like(points)

    #save as PLY file
    with open(output_filename, "w") as ply:
        ply.write(f"ply\nformat ascii 1.0\nelement vertex {len(points)}\n")
        ply.write("property float x\nproperty float y\nproperty float z\n")
        ply.write("property uchar red\nproperty uchar green\nproperty uchar␣
↪blue\nend_header\n")
        for (x, y, z), (r, g, b) in zip(points, colors):
            ply.write(f"{x} {y} {z} {r} {g} {b}\n")

    print(f"Saved improved point cloud as {output_filename}")
```

1.3.3 Estimation Functions

Refine Chessboard Corners This function refines the initial chessboard corner detections to improve the accuracy of subsequent disparity and distance calculations. Using OpenCV's cornerSubPix method, the corner positions are iteratively adjusted to sub-pixel accuracy based on image gradients, enhancing accuracy. The function also ensures that the corners are ordered correctly by checking their spatial layout and flipping them if necessary. Proper ordering of corners ensures accurate correspondence between the left and right images, which is essential for reliable stereo matching and distance estimation.

[402]:
```
def refine_chessboard_corners(frameL_gray, frameR_gray, cornersL, cornersR):

    #define termination criteria
    termination_criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_COUNT,␣
↪30, 1)

    #refine corners using sub-pixel accuracy
    cv2.cornerSubPix(frameL_gray, cornersL, (5, 5), (-1, -1),␣
↪termination_criteria)
    cv2.cornerSubPix(frameR_gray, cornersR, (5, 5), (-1, -1),␣
↪termination_criteria)

    #ensure correct order of corners, look at 1st and 8th corners
    if cornersL[0][0][0] > cornersL[8][0][0]: cornersL = np.flip(cornersL, 0)
    if cornersR[0][0][0] > cornersR[8][0][0]: cornersR = np.flip(cornersR, 0)

    return cornersL, cornersR
```

Distance & Disparity Estimation In the keypoint-based method, disparity is computed as the difference in x-coordinates of corresponding chessboard corners: $d = uL - uRd = uL \ - uR$

. These values form a (6×8) disparity matrix, averaged column-wise, then across columns for a robust final disparity. The chessboard's distance is then estimated using the stereo vision formula: $z_k = \frac{f \cdot b}{d_{mean}}$. This approach leverages stable keypoints, offering more reliable distance estimation than SGBM, especially in noisy conditions. The formulas used for the mean disparity computation are shown below:

$$d_{\text{columns}}(j) = \frac{1}{8} \sum_{i=1}^{8} \left(x_L^{i,j} - x_R^{i,j} \right) \quad \text{and} \quad d_{\text{keypoints}} = \frac{1}{6} \sum_{j=1}^{6} d_{\text{columns}}(j)$$

The **StereoSGBM** (Semi-Global Block Matching) algorithm computes a dense disparity map by matching pixels between rectified grayscale stereo frames within an adaptive disparity range. It minimizes the energy function $E(D)$, balancing local matching costs and global smoothness constraints across multiple scanline directions. To optimize computation, only a central region is processed, considering padding factors like block size, minimum disparity, and search range. A focused window is then extracted for distance estimation, reducing overhead. The disparity is refined using the median (for robustness) or mean, and converted into obstacle distance via the stereo vision depth formula. Optimal disparity is found as the solution of the following optimization problem:

$$D(x,y) = \arg\min_d E(D) \quad \text{where} \quad E(D) = \sum_{(x,y)} C(x,y,D(x,y)) + \sum_{(x,y)} P(D(x,y), D(x',y')) \quad \text{and} \quad P(D(x,y), D(x',$$

[ ]:
```
def distance_disparity_estimation(cornersL, cornersR, initial_disparity,␣
↪frameL_gray, frameR_gray, frame_width, frame_height):

    def extract_disparity_crop(full_disparity_map, frame_height, frame_width,␣
↪central_window_radius, stereo_block_size, NumDisparities, offset):

        """extracts a cropped region from the disparity map"""

        #define cropping factors
        half_h, half_w = frame_height // 2, frame_width // 2 #compute the␣
↪center of the frame
        stereo_adjustment = (stereo_block_size - 1) // 2 #adjust for block size␣
↪to ensure proper alignment
        disparity_adjustment = NumDisparities + offset #adjust width to␣
↪accommodate the disparity search range

        #compute the cropping indeces
        h_start, h_end = half_h - central_window_radius - stereo_adjustment,␣
↪half_h + central_window_radius + stereo_adjustment
        w_start, w_end = half_w - central_window_radius - disparity_adjustment␣
↪- stereo_adjustment, half_w + central_window_radius + stereo_adjustment
```

```
        #return the cropped central region for disparity calculations
        return full_disparity_map[h_start:h_end, w_start:w_end]

    ########################################
    ########## Keypoints Disparity #########
    ########################################

    #calculate disparity columns from chessboard corners (hor. coordinates)
    disparity_matrix = cornersL[:, 0, 0].reshape(8, 6) - cornersR[:, 0, 0].
↪reshape(8, 6)

    #average over rows to get column disparities
    disparity_columns = np.mean(disparity_matrix, axis=0).reshape(6, 1)

    #compute the mean disparity across all columns
    disparity_keypoints = np.mean(disparity_columns)

    ########################################
    ############ SGBM Disparity ############
    ########################################

    #adjust disparity search range
    offset = max(0, int(initial_disparity - 32)) if initial_disparity >= 32␣
↪else 0
    NumDisparities = min(max(64 + offset, 64), 128)  #minimum 64, cap at 128␣
↪for stability

    #define the penalty term for SGBM
    P1 = 2 * stereo_block_size**2
    P2 = 12 * stereo_block_size**2

    #create StereoSGBM object
    stereo = cv2.StereoSGBM_create(minDisparity = offset, numDisparities =␣
↪NumDisparities, blockSize = stereo_block_size, P1 = P1, P2 = P2)

    #compute dense disparity map
    full_disparity_map = stereo.compute(frameL_gray, frameR_gray).astype(np.
↪float32) / 16

    #crop the disparity map
    cropped_disparity_map = extract_disparity_crop(full_disparity_map,␣
↪frame_height, frame_width, central_window_radius, stereo_block_size,␣
↪NumDisparities, offset)

    #compute main disparity for distance estimation
```

```
    if robust_mean: disparity_SGBM = np.
↪median(cropped_disparity_map[cropped_disparity_map > 0])
    else: disparity_SGBM = np.mean(cropped_disparity_map)

    ########################################
    ############ Depth Estimation ##########
    ########################################

    #estimate the depth with both methods
    distance_keypoints, distance_SGBM = (f * b) / disparity_keypoints, (f * b) /
↪ disparity_SGBM

    #compute the depth map
    depth_map = np.where(full_disparity_map > 0, (f * b) / full_disparity_map,␣
↪0)

    ########################################
    ############### Results ################
    ########################################

    return {
        "distance_keypoints": distance_keypoints,
        "disparity_keypoints": disparity_keypoints,
        "disparity_columns": disparity_columns,
        "distance_SGBM": distance_SGBM,
        "disparity_SGBM": disparity_SGBM,
        "disparity_map_full": full_disparity_map,
        "disparity_map_crop": cropped_disparity_map,
        "depth_map": depth_map
    }
```

Chessboard Size Estimation This function calculates the real-world width and height of the chessboard pattern by processing corner points detected in both the left and right stereo images. For each view, it sums the distances between corresponding corner points along rows (for width) and columns (for height) and scales them using the known distance to the chessboard and the camera's focal length. Finally, the averaged distances from both rows and columns are combined across views to provide robust estimates of the chessboard pattern's dimensions in millimeters. Perspective Projection reads:

$$W = \frac{w \cdot Z}{f} \quad H = \frac{h \cdot Z}{f}$$

The keypoint-based method demonstrates more stability, with fewer fluctuations, due to the reliability of chessboard corner detection. On the other hand, the SGBM-based method shows larger variability, especially in regions where the disparity map is less reliable (e.g., around frames 150–250). Indeed, after frame 250, accuracy worsens due to the robot's rapid rotation and closer proximity, causing significant changes in the 3D-to-2D projection. Spikes in the SGBM graph may be due to noise in dense disparity calculations, which can be subject of future improvements.

[404]:
```
def chessboard_size_estimation(corners_list, z, f):

    #initialize results
    total_width = 0
    total_height = 0

    #access number of rows and columns
    n_rows = chessboard_grid[0]
    n_columns = chessboard_grid[1]

    #precompute scaling factor
    scale_factor = z / f

    #loop through both cornersL and cornersR
    for corners in corners_list:

        #ensure corners have the correct layout (6x8)
        assert len(corners) >= 48, "Insufficient chessboard corners detected. Expected at least 48."

        #calculate the width of the chessboard by summing distances between corners in each row
        width_sum = sum((abs(corners[i][0][0] - corners[40 + i][0][0]) * scale_factor) for i in range(n_rows))

        #calculate the height of the chessboard by summing distances between corners in each column
        height_sum = sum((abs(corners[8 * j][0][1] - corners[8 * j + 7][0][1]) * scale_factor) for j in range(n_columns))

        #accumulate the averages along chessboard's dimensions
        total_width += width_sum / n_rows  #average across the 8 rows
        total_height += height_sum / n_columns #average across the 6 columns

    #compute the final average width and height using both left and right views
    chessboard_width = total_width / 2
    chessboard_height = total_height / 2

    return chessboard_width, chessboard_height
```

Chessboard Angle Estimation This function estimates the angle between the obstacle (represented by a chessboard) and the camera's image plane, which is essential for determining the obstacle's orientation. The angle is calculated based on the disparity differences between the closest and farthest vertical stripes of the chessboard. Specifically, the function computes the difference in depth ΔZ as:

$$\Delta Z = \left| \frac{f \cdot b}{d_{\min}} - \frac{f \cdot b}{d_{\max}} \right|$$

where *dmin* and *dmax* are the disparities of the closest and farthest stripes. The depth difference ΔZ is then combined with the chessboard width w to calculate the angle with arctangent function. This formula holds due to Perspective Projection, which projects the chessboard's width line onto the image plane; however, the higher the angle the worse will be the geometric assumption.

$$\tau = \arctan\left(\frac{\Delta Z}{w}\right)$$

The angle was computed using the keypoint-based disparity method, which leverages the disparities at detected chessboard corners. Sudden spikes in the graph may indicate frames where the keypoints were poorly detected or temporarily lost. Overall, the stable regions with small angles suggest that the chessboard was largely aligned with the camera.

[405]:
```
def chessboard_angle_estimation(disparity_values, width_chess):

    #compute depth differences using disparity values at the first and last␣
 ↪columns
    delta_depth = abs((f * b) / disparity_values[0] - (f * b) /␣
 ↪disparity_values[5])

    #compute the angle using arctangent
    angle_tau = math.degrees(math.atan(delta_depth / width_chess))

    return angle_tau
```

1.4 3) Algorithm

Initialization As a first step, the algorithm initializes and opens the left and right video streams by retrieving their properties (e.g., frame size, rate, and length), and preparing the output video for writing.

[406]:
```
left_video, right_video, output_video, frame_width, frame_height, frame_rate,␣
 ↪video_length = video_initialization(left_video_path, right_video_path,␣
 ↪output_video_path)
```

```
Videos opened correctly.

Video Properties: Frame_Width=640, Frame_Height=480, FPS=15.0, Video_Length=389
```

Main Loop The main loop iterates through each frame of the stereo video, performing sequential operations to estimate the desidered information from the dataset. The main tasks in the loop include: - **Frame Pre-Processing**: reads and converts the left and right frames of the given videos to grayscale for further analysis by using frame_preprocessing() function. - **Chessboard Corner Detection**: detects the chessboard corners in both frames using cv2.findChessboardCorners(), followed by sub-pixel refinement using refine_chessboard_corners(). - **Disparity and Distance Estimation**: calculates disparity values by using both keypoint-based and dense SGBM methods, by returning depth and disparity information for further analyses. - **Chessboard Size Calculation**: computes the real-world width and height of the chessboard in millimeters by scaling corner

distances using the focal length and current distance. - **Angle Estimation**: estimates the angle between the chessboard and the camera's image plane using disparity differences across vertical stripes. - **Information Visualization**: displays obstacle information, such as distance, dimensions, and angle, within the video frames. Safety warnings are triggered if the obstacle is too close. - **Metrics Storage**: stores key data, including distances, widths, heights, and angles, for further analysis, evaluation and storage which will be taken at the end of the loop.

[407]:
```
for frame_number in range(video_length):  #iterate through the frames of videos

    #print progress as a percentage alongside the number of frames
    print(f"\rProgress: {((frame_number+1) / video_length) * 100:.1f}% | Frame␣
↪{frame_number + 1}/{video_length}", end="")

    ################################
    ##### Frame Pre-Processing ####
    ################################

    #read and pre-process current frames
    result = frame_preprocessing(left_video, right_video, frame_number)

    #unpack current frames
    if result is None: continue
    frameL, frameR, frameL_gray, frameR_gray = result

    ################################
    ###### Corners Extraction #####
    ################################

    #detect chessboard corners
    foundL, cornersL = cv2.findChessboardCorners(frameL_gray, chessboard_grid)
    foundR, cornersR = cv2.findChessboardCorners(frameR_gray, chessboard_grid)
    if not (foundL and foundR): continue #skip frame if corners are not found

    #refine corners using sub-pixel accuracy
    if corners_refinement: cornersL, cornersR =␣
↪refine_chessboard_corners(frameL_gray, frameR_gray, cornersL, cornersR)

    ################################
    ##### Distance & Disparity ####
    ################################

    if first_frame: #use keypoints disparity to initialize SGBM search range on␣
↪the first frame

        disparity_SGBM = np.mean(cornersL[:, 0, 0] - cornersR[:, 0, 0])
        first = False  #disable initialization for future frames
```

```
   #estimate disparity and distance
   results = distance_disparity_estimation(cornersL, cornersR, disparity_SGBM,␣
↪frameL_gray, frameR_gray, frame_width, frame_height)

   #extract relevant values
   distance_keypoints, disparity_keypoints, disparity_columns, distance_SGBM,␣
↪disparity_SGBM, full_disparity_map, cropped_disparity_map, depth_map =␣
↪results.values()

   ################################
   ####### Chessboard Size #######
   ################################

   #compute chessboard size using both distances
   width_keypoints, height_keypoints = chessboard_size_estimation([cornersL,␣
↪cornersR], distance_keypoints, f)
   width_SGBM, height_SGBM = chessboard_size_estimation([cornersL, cornersR],␣
↪distance_SGBM, f)

   ################################
   ####### Chessboard Angle ######
   ################################

   #estimate the angle between the chessboard and the camera's image plane
   angle_keypoints = chessboard_angle_estimation(disparity_columns,␣
↪width_keypoints)

   ################################
   ######## Visualization ########
   ################################

   #select visualization metrics
   distance = distance_SGBM if method == 'SGBM' else distance_keypoints
   chessboard_width, chessboard_height = ((width_SGBM, height_SGBM) if method␣
↪== 'SGBM' else (width_keypoints, height_keypoints))

   #display visual information
   alarm_counter = display_information(
       frameL,                    #frame where info is displayed
       alarm_counter,             #alarm counter for flickering effect
       distance,                  #distance to obstacle in mm
       min_dist,                  #minimum safe distance in mm
       chessboard_width,          #chessboard width in mm
       chessboard_height,         #chessboard height in mm
       angle_keypoints,           #angle of chessboard in degrees
       chessboard_grid,           #chessboard grid size
```

```
        cornersL,                    #refined corners of the left chessboard
        foundL,                      #whether the chessboard was detected
        frame_number,                #current frame number
        video_length,                #total number of frames
        output_video                 #object for writing frames to output video
    )

    ################################
    ####### Metrics Storage #######
    ################################

    metrics_data["distances_keypoints"].append(distance_keypoints)
    metrics_data["distances_SGBM"].append(distance_SGBM)
    metrics_data["widths_keypoints"].append(width_keypoints)
    metrics_data["heights_keypoints"].append(height_keypoints)
    metrics_data["widths_SGBM"].append(width_SGBM)
    metrics_data["heights_SGBM"].append(height_SGBM)
    metrics_data["angles_keypoints"].append(angle_keypoints)
    metrics_data["disparity_maps"].append(full_disparity_map)
    metrics_data["depth_maps"].append(depth_map)

    #save the image for point cloud generation
    if frame_number == 175: point_cloud_disp = full_disparity_map
```

```
Progress: 0.8% | Frame 3/389
```

```
/tmp/ipykernel_3553/3478726205.py:62: RuntimeWarning: divide by zero encountered
in divide
  depth_map = np.where(full_disparity_map > 0, (f * b) / full_disparity_map, 0)
/tmp/ipykernel_3553/2854069748.py:7: DeprecationWarning: Conversion of an array
with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you
extract a single element from your array before performing this operation.
(Deprecated NumPy 1.25.)
  angle_tau = math.degrees(math.atan(delta_depth / width_chess))
```

```
Progress: 100.0% | Frame 389/389
```

At the end of the loop, the algorithm releases the resources associated with the input video streams (left and right) as well as the output video file in order to free memory and avoid resource leaks.

[408]:
```
left_video.release()
right_video.release()
output_video.release()
```

Output Visualization Finally, metrics function is invoked to compare distances, widths, and heights estimation using both methods across all processed frames and it generates and saves relevant plots for visualization.

[409]:
```
comparison_results = metrics(metrics_data, output_directory, min_dist)
```

```
Saved Disparity Map map as
/home/andrea/Desktop/CV/[IPCV]_PernaAndrea/Project_Outcomes/Disparity
Map_map_frame87.png
Saved Depth Map map as
/home/andrea/Desktop/CV/[IPCV]_PernaAndrea/Project_Outcomes/Depth
Map_map_frame87.png
Saved Disparity Map map as
/home/andrea/Desktop/CV/[IPCV]_PernaAndrea/Project_Outcomes/Disparity
Map_map_frame174.png
Saved Depth Map map as
/home/andrea/Desktop/CV/[IPCV]_PernaAndrea/Project_Outcomes/Depth
Map_map_frame174.png
Saved Disparity Map map as
/home/andrea/Desktop/CV/[IPCV]_PernaAndrea/Project_Outcomes/Disparity
Map_map_frame260.png
Saved Depth Map map as
/home/andrea/Desktop/CV/[IPCV]_PernaAndrea/Project_Outcomes/Depth
Map_map_frame260.png
Distance comparison plot saved to /home/andrea/Desktop/CV/[IPCV]_PernaAndrea/Pro
ject_Outcomes/distance_comparison.png
Chessboard width comparison plot saved to
/home/andrea/Desktop/CV/[IPCV]_PernaAndrea/Project_Outcomes/width_comparison.png
Chessboard height comparison plot saved to /home/andrea/Desktop/CV/[IPCV]_PernaA
ndrea/Project_Outcomes/height_comparison.png
Relative distance difference plot saved to /home/andrea/Desktop/CV/[IPCV]_PernaA
ndrea/Project_Outcomes/relative_distance_difference.png
Relative distance difference plot (log scale) saved to /home/andrea/Desktop/CV/[
IPCV]_PernaAndrea/Project_Outcomes/relative_distance_difference_log.png
```

A point cloud of a particular scene's frame is generated and saved in a PLY file for visualization is Sketchfab

[410]:
```
point_cloud = generate_point_cloud(point_cloud_disp, frameL, f, b,␣
↪output_filename="robot_nav_pointcloud.ply")
```

```
Saved improved point cloud as robot_nav_pointcloud.ply
```