ESSLINGEN UNIVERSITY

For people and technology.

ADMIT14 - Camera based Real-Time Localization and Tracking

ASM-VS (2023-25)

Esslingen University of Applied Sciences, Germany

Authors: M.S. Sri Harish, V. Mascarenhas, L. Cheng-Yu, K. A. N. Noushiq Mohmmed

Supervisor: Prof. Dr. Ralf Schuler

Abstract

Generally, the navigation data for an autonomous vehicle on a large scale is provided by a GPS system. In this project, an attempt has been made to create navigation data for a 1:14 scaled traffic environment by utilizing a bird's-eye view from a hall-roof-mounted camera. The visuals of the map from the camera are processed, and the required objects are detected. Subsequently, the specific object is tracked, and the data is shared with other systems in the environment in real-time. This data captures the entire traffic environment and supplies necessary information for autonomous driving, vehicle-to-X communication, and traffic management.

This project develops a system that transforms the map into a global coordinate system, implements a machine learning-based detection algorithm, and tracks and predicts the object centroid using the Kalman filter. The entire system, including the hardware, is implemented in a high-performance data processing unit, and the desired data transmission is carried out.

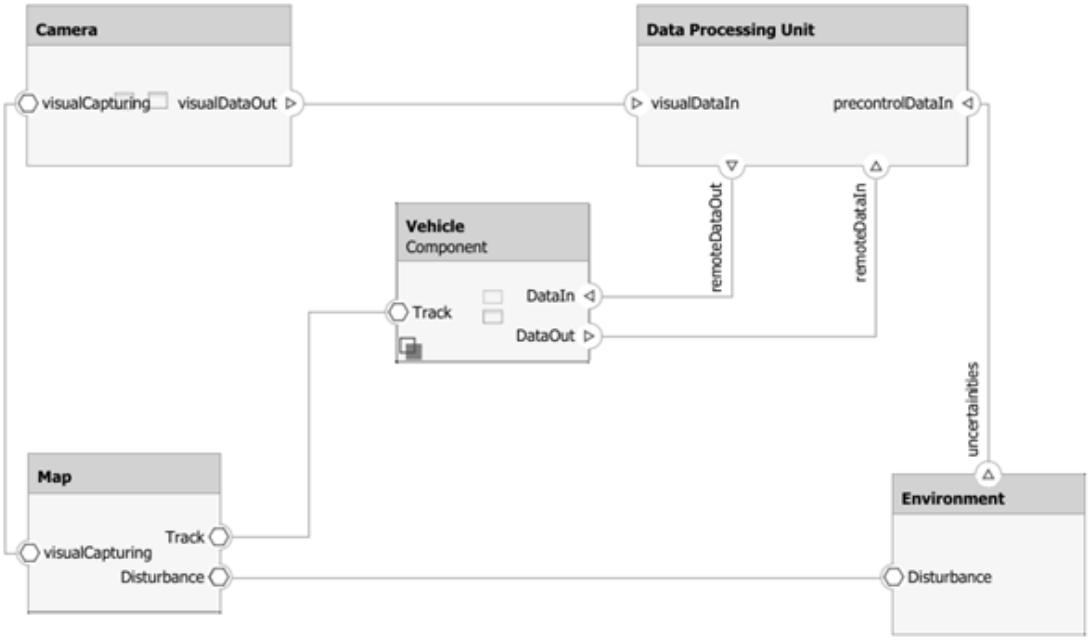

Fig. 0.1: System context diagram (SysComposer)

Contents

1 Introduction

The development of autonomous vehicles has seen significant progress in recent years. However, ensuring their safety and reliability requires extensive testing in diverse environments. While virtual or simulation-based testing offers a controlled setting, it may not fully capture the complexities and unpredictable nature of real-world scenarios. On contrary traditional methods, which involve physically driving prototypes on public roads, are expensive, time-consuming, and carry safety risks. They require a multitude of safety precautions, permits, and specialized personnel.

To address this challenge, we have created scaled-down autonomous traffic environments that provide a more controlled and cost-effective testing platform. This project focuses on developing a camera-based real-time localization and tracking system specifically designed for a scaled-down autonomous traffic environment. This system offers several advantages over both the virtual and traditional testing methods which are discussed in the subsequent sections.

1.1 Advantages over Virtual Testing

- **Real-World Sensor Data:** The system utilizes real-world sensor data (cameras) like those used in actual autonomous vehicles. This allows for testing and evaluation of sensor performance and perception algorithms under realistic conditions, which may not be fully replicated in simulations.
- **Unpredictable Elements:** The scaled-down environment can incorporate physical elements like lighting variations, surface textures, and slight imperfections that can be difficult to model realistically in simulations. These elements challenge the autonomous vehicle's perception and decision-making capabilities, providing a more robust evaluation.

1.2 Advantages over Traditional Testing

- **Cost-Effectiveness:** Scaled-down environments require less expensive equipment and infrastructure compared to real-world testing. Additionally, the controlled nature minimizes risks associated with accidents, reducing insurance and liability costs.
- **Versatility:** The environment can be easily modified to represent various scenarios (e.g., highways, intersections, urban areas) without the logistical challenges associated with real-world locations. This allows for broader testing of autonomous vehicles capabilities in a shorter time-frame.
- **Repeatability:** Tests can be easily replicated in the scaled-down environment, allowing for consistent evaluation and comparison of different algorithms and vehicle behavior under controlled conditions.
- **Safety:** Testing in a controlled environment minimizes the risk of accidents and injuries that could occur during real-world testing.

This report details the design, implementation, and evaluation of the camera-based localization and tracking system. We will discuss the motivation behind the project, including the rationale for choosing a 1:14 scale. Here, the selection of a 1:14 scale offers specific benefits:

- **Commercially Available Vehicles:** A wide range of commercially available RC car models are built in 1:14 scale. This facilitates the selection and modification of vehicles for the testing environment, reducing development time and costs.
- **Focus on Commercial autonomous vehicles**: There is growing evidence suggesting that the initial development and implementation of autonomous driving technologies may prioritize commercial vehicles such as trucks and buses for several reasons. These reasons include:
  - **Controlled Routes:** Commercial vehicles often operate on predetermined routes with predictable traffic patterns, simplifying the initial challenges of autonomous navigation.
  - **Predictable Behavior:** Commercial vehicle fleets can be outfitted with communication technologies to facilitate communication between vehicles (V2X), further enhancing safety and efficiency in controlled environments.
  - **Economic Benefits:** The adoption of autonomous trucks has the potential to significantly reduce transportation costs and improve logistics efficiency, creating a strong economic incentive for early implementation.

This project focuses on developing a camera-based real-time localization and tracking system specifically designed for a scaled-down autonomous traffic environment. This system leverages computer vision techniques to achieve real-time localization and tracking of objects within the environment, providing valuable data for testing and evaluating autonomous vehicle algorithms.

In essence, this system acts as a camera-based GPS system for the environment, enabling the precise location and movement of objects to be determined in real-time. Furthermore, the rich tracking data generated by the system can be used to validate the effectiveness of different motion planning algorithms for autonomous vehicles navigating within the scaled-down environment. This allows for evaluation of how well these algorithms perform under controlled conditions before deploying them in real-world scenarios.

Here's a breakdown of the core functionalities within this camera-based system:

- **Object Detection:** This involves identifying and classifying objects of interest within the camera image. The system likely employs a deep learning model, such as YOLOv4, which has been trained on a large dataset of labeled images containing various objects relevant to the traffic environment (e.g., vehicles, pedestrians, traffic signs). When a new image frame is captured by the camera, the YOLOv4 model analyzes it and outputs bounding boxes around the detected objects, along with their corresponding class labels (e.g., "car," "truck," "person").
- **Coordinate Transformation:** Camera images captured from a specific viewpoint present a distorted perspective of the real world. To enable accurate localization and tracking, the system needs to transform the detected object's bounding box coordinates from the image plane (pixels) to a real-world coordinate system (e.g., meters). This transformation process may involve techniques like camera calibration to account for the intrinsic properties of the camera lens and its extrinsic parameters (position and orientation) within the environment.

- **Object Tracking:** Once objects are identified and their locations are established within each frame, the system employs object tracking algorithms to link detections across consecutive video frames. This creates trajectories that depict the movement of objects over time. This project specifically utilizes an Extended Kalman Filter (EKF) for object tracking. Kalman filters are a powerful tool for linear state estimation, and the EKF allows the system to account for non-linearities in object motion that may be present in the scaled-down environment. By incorporating the object's velocity and acceleration into the tracking process, the EKF can provide more accurate predictions of an object's future position, even during manoeuvres or temporary occlusions.

By integrating these functionalities, the camera-based system facilitates real-time localization and tracking of objects within the scaled-down environment, providing valuable data for testing and development of autonomous vehicle algorithms.

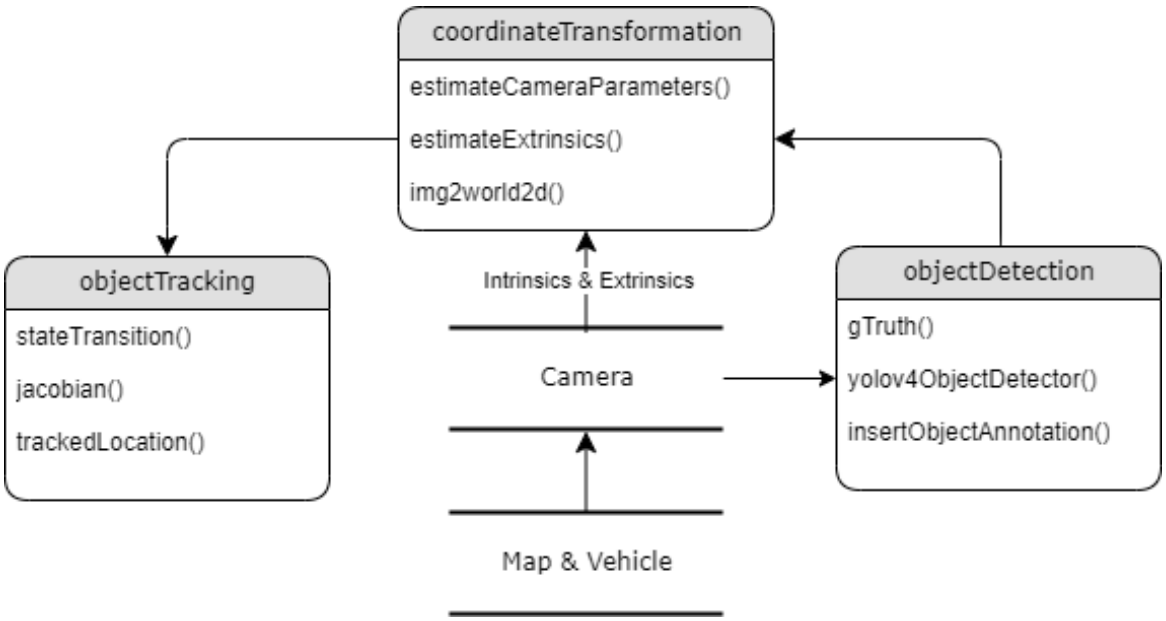

Fig. 1.1: System functionality flow diagram

2 Camera Setup

The Camera that we used was Reolink camera b4k11 POE (Power over Ethernet) IP Camera. To access the camera streaming directly through MATLAB we use ipcam function. For this function to work, some settings need to be modified in the camera client application. The changes required are listed in the steps below:

- The username and password for the camera is ‘admin’ and ‘Esslingen123’ respectively. The password could be changed by hard reset.
- Make sure the camera and PC should be connected to the same wi-fi network.
- Change the streaming protocol from http to rtsp in the camera streaming settings in the client app.
- From the Reolink website for this current camera model ip address for rtsp protocol is ’rtsp://192.168.1.101:554/’.
- If different model of camera is used, the rtsp ip address can be found in the respective website.

Matlab Code:

```
cam = ipcam('rtsp://192.168.1.101:554/','admin','Esslingen123');
cam2 = preview(cam); % To check for stable connection
frame=snapshot(cam);
```

The cam object created using ipcam function can be used in the while loop of the object detection and object tracking to extract frame rates in real time.

Disclaimer: Make sure to use the same resolution for video streaming as well as camera calibration

3 Camera Calibration

3.1 Overview

Camera calibration is necessary for any object detection system involving accurate estimation of world coordinates of the objects detected. It is the process carried out to estimate camera parameters using a set of calibration patterns whose world coordinates are known in advance. The parameters include camera intrinsics, distortion coefficients and camera extrinsics. The estimated camera parameters are used to remove lens distortion effects from an image, measure planar objects, reconstruct 3-D scenes from multiple cameras and perform other computer vision applications. It works for both single and stereo camera systems. Based on the type of the camera (Fisheye, Pinhole) different distortions in images occur that prevents use from using Affine transformations to convert the image coordinates to world coordinate system.

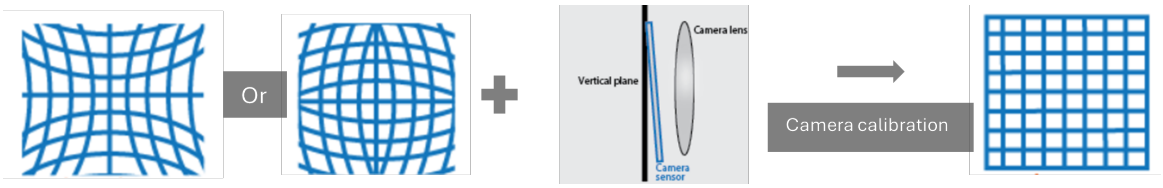

Fig. 3.1: Commonly seen lens distortions include pincushion (negative radial displacement), barrel distortion (positive radial displacement) and tangential distortions due to the inclination of lens to the image plane.

3.2 Camera Parameters

The calibration algorithm calculates the camera matrix using the extrinsic and intrinsic parameters. The extrinsic parameters represent a rigid transformation from 3-D world coordinate system to the 3-D camera's coordinate system. The intrinsic parameters represent a projective transformation from the 3-D camera's coordinates into the 2-D image coordinates.

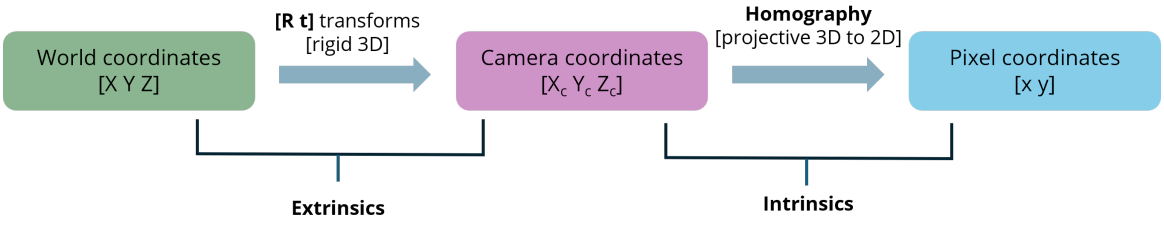

Fig. 3.2: Camera coordinate conversion overview

3.2.0.1 Extrinsic Parameters:

The extrinsic parameters consist of a rotation, R, and a translation, t. The origin of the camera's coordinate system is at its optical centre and its x- and y-axis define the image plane. Extrinsics parameters changes with change camera position with respect to the map.

Fig. 3.3: World coordinate to camera coordinate system

3.2.0.2 Intrinsic Parameters:

The intrinsic parameters include the focal length, the optical centre, also known as the principal point, and the skew coefficient. The camera intrinsic matrix, K, is defined as:

$$\begin{bmatrix} f_x & s & C_x \\ 0 & f_y & C_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

where,

$[C_x C_y]$ - optical centre in pixels,

$[f_x f_y]$ - focal length in pixels,

$f_x = \frac{F}{P_x}; f_y = \frac{F}{P_y}$, where F is the focal length and P_x, P_y are pixel sizes in world units,

s - skew coefficient, which is non-zero if the image axes are not perpendicular.

Intrinsics are fixed for a given camera and doesn't have to be calculated repeatedly as long as the focal length and the resolution of the image remains fixed.

3.3 Calibration Method

- Prepare the calibration pattern. Most common is the checkerboard patterns with odd number of squares on each side. Pattern should be placed to cover at least 20% of the image. We used 9×6 checkered pattern.

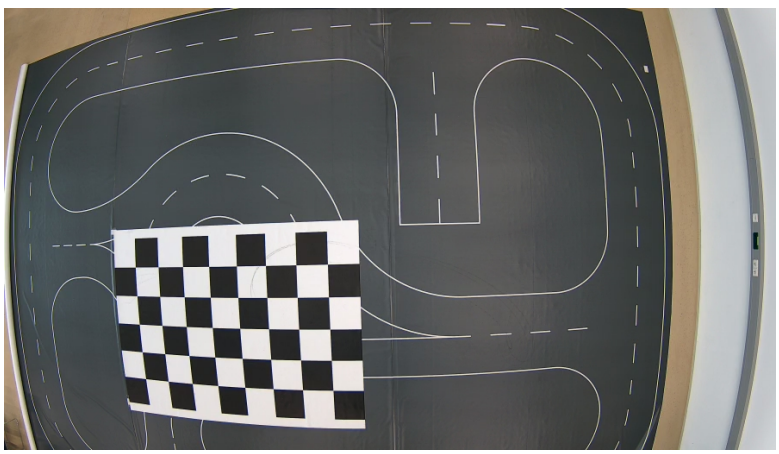

Fig. 3.4: Calibration pattern placed in the field of view of camera

- Take multiple images of a calibration pattern from different angles.

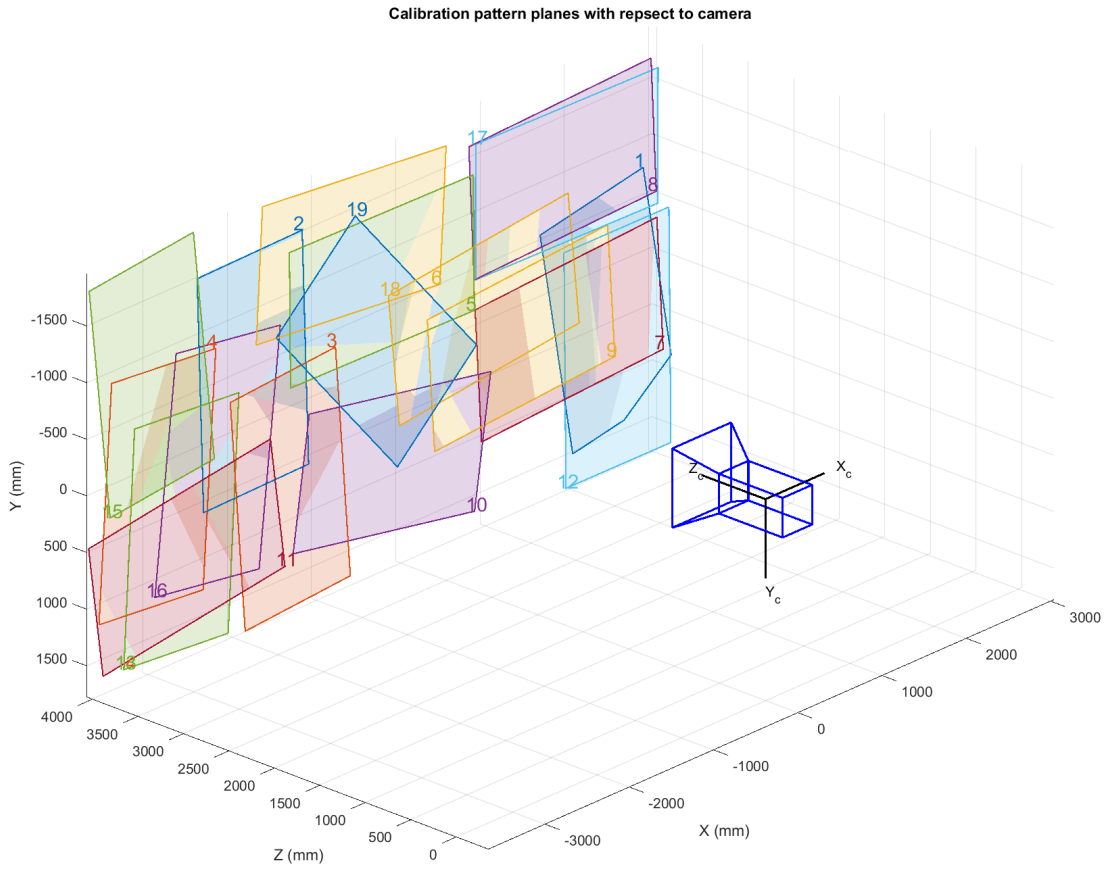

Fig. 3.5: Series of calibration patterns arranged to cover the whole field of view through multiple snapshots

- Next step involves estimation of the camera intrinsics based on the multiple images of checkered pattern at the different positions of the camera viewing plane and their corresponding world coordinates. Below image shows the lines in the script used for that purpose.

Matlab Code:

```
%% Estimate Camera Parameters

% Detect the checkerboard corners in the images.
[imagePoints, boardSize] = detectCheckerboardPoints(files);

% Generate the world coordinates of the checkerboard
% corners in the pattern-centric coordinate system,
% with the upper-left corner at (0,0).
squareSize = 300; % in millimeters
worldPoints = generateCheckerboardPoints(boardSize, squareSize);

% Calibrate the camera.
imageSize = [size(I, 1), size(I, 2)];
cameraParams = estimateCameraParameters(imagePoints,worldPoints,...
ImageSize = imageSize);

intrinsics = cameraParams.Intrinsics;
```

- **detectCheckerboardPoints** is a computer vision function that automatically detects the vertices of the checkered pattern from the images.
- Based on the image points from the above function and the corresponding world points generated from the actual grid dimensions in millimetres, **estimateCameraParameters** function deduces the camera intrinsics. It is not a pure mathematical function.

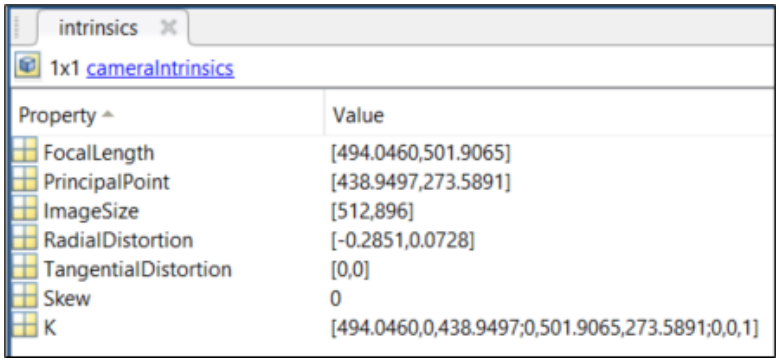

Fig. 3.6: Typical intrinsics of the camera

- Before we can estimate the extrinsics, the image has to be undistorted and the edges of the checkered pattern in the image should be parallel lines.

Fig. 3.7: Steps involved in extraction of camera extrinsics

- **undistortImage** function is used for this purpose. It uses the camera intrinsics to transform the original image into a undistorted image.
- Estimate the extrinsics for certain reference camera position using the function: **estimateExtrinsics(imagePoints, worldPoints, intrinsics)**. World points used in this case are the inner vertices of the checkered pattern. For a 9×6 grid there are 8×5 inner vertices.

Matlab Code:

```
%% Map points in image to real cordinates
% Detect the checkerboard.
[imagePoints, boardSize] = detectCheckerboardPoints(im);
% Adjust the imagePoints so that they are expressed in
% the coordinate system used in the original image,
% before it was undistorted.
% This adjustment makes it compatible with the
% cameraParameters object computed for the original image.
newOrigin = intrinsics.PrincipalPoint - newIntrinsics.PrincipalPoint;
imagePoints = imagePoints + newOrigin;
% adds newOrigin to every row of imagePoints
count = 1;
for k = 8:-1:1  % There are 8 inner vertices in a 9*6
% checkered grid length-wise
for j = 1:5 % There are 5 inner vertices in a 9*6 grid width-wise
worldPoints(count,1) = 300*(k-1);
worldPoints(count,2) = 300*(j-1);
count = count + 1;
end
end
% Compute extrinsic parameters of the camera.
camExtrinsics = estimateExtrinsics(imagePoints,...
                worldPoints, intrinsics);
```

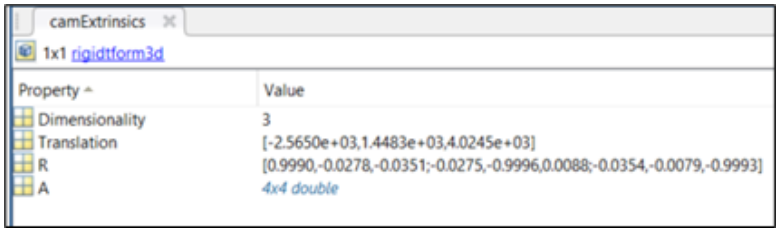

Fig. 3.8: Typical extrinsics of the camera

- Obtain the world coordinates using the function: **img2world2d(imagePoints, extrinsics, intrinsics)**.

Matlab Code:

```
% Get the world coordinates of the corners
worldPoints1 = img2world2d(imagePoints, camExtrinsics, intrinsics);
```

4 Object Detection Methods: From Machine Learning to Deep Learning

4.1 Aggregate Channel Features (ACF) Detector

The Aggregate Channel Features (ACF) detector, a significant milestone in traditional machine learning approaches to object detection, offers a computationally efficient alternative to complex deep learning models while maintaining robust performance. Developed as an extension of the channel features framework, ACF operates by extracting and combining multiple feature channels from input images, capturing various aspects of visual information relevant to object detection. ACF extracts features directly as pixel values in extended channels without computing rectangular sums at various locations and scales, streamlining the process.

The workflow of ACF typically involves the following steps:

1. **Input Processing:**
   The ACF detector begins by processing the input image to create multiple channels. These channels often include color channels (like RGB), gradient magnitude, and possibly other features that capture different visual properties of the image. These channels provide a richer set of features compared to using the raw image alone.

2. **Feature Extraction:**
   In this step, features are directly extracted from the created channels. The ACF method extracts pixel-level features without complex computations, which helps in maintaining computational efficiency. The simplicity of this step is one of the key aspects that differentiate ACF from more complex methods.

3. **Aggregation:**
   The extracted features are aggregated, typically using pooling methods. This aggregation helps in reducing the dimensionality of the features while summarizing the essential information. This step is crucial for making the data manageable and for improving the performance of subsequent classifiers.

4. **Classification:**
   A classifier is trained on these aggregated features to identify objects. This step involves using the features to distinguish between different object categories. The classifier could be based on traditional machine learning methods or simple decision rules, depending on the implementation.

5. **Multi-scale Detection:**
   The process is applied at multiple scales to detect objects of various sizes. This multi-scale approach is important for handling the variability in object sizes within images, ensuring that the detector can identify both small and large objects effectively. This approach allows ACF to achieve efficient object detection while maintaining effectiveness for specific tasks.

The key components of ACF include:

4.1.1 Feature Extraction in ACF

The feature extraction process in ACF is crucial to its performance:
Histogram of Oriented Gradients (HOG):

- HOG computes gradient orientations in localized portions of an image.
- It creates histograms of gradient directions, capturing local shape information.
- HOG is particularly effective for detecting objects with distinct shape characteristics.

Color Channels:

- ACF typically uses the LUV color space, which separates luminance from color information.
- This separation allows the detector to be more robust to lighting variations.

Gradient Channels:

- Gradient magnitude channels highlight edges and texture in the image.
- These channels are particularly useful for detecting objects with strong contours.

4.1.2 Channel Aggregation and Classification

After feature extraction, ACF proceeds with:

- **Aggregation:** Features are summed over local rectangular regions, reducing the feature dimensions.
- **Boosting:** AdaBoost is used to train a cascade of decision trees on these aggregated features.
- **Cascade Structure:** The detector uses a cascade structure, allowing for quick rejection of non-object regions, enhancing efficiency.

4.1.3 Advantages of ACF Detector

- **Reduced Data Requirements:** ACF requires significantly less training data compared to deep learning methods.
- **Efficient Training:** The training process is considerably faster than that of deep neural networks.
- **Computational Efficiency:** During detection, ACF demands less computational power, making it suitable for resource-constrained environments.
- **High Accuracy for Specific Classes:** ACF demonstrates high accuracy for well-defined object classes.

4.1.4 Limitations of ACF

Despite its advantages, ACF has notable limitations:

- **Reduced Flexibility:** It may struggle with highly variable object classes.
- **Complex Scene Handling:** ACF's performance can degrade in complex visual scenarios.

4.1.5 Ground Truth Data Preparation for ACF Detector Using MATLAB's Video Labeler

The creation of accurate and comprehensive ground truth data is a critical step in training an Aggregate Channel Features (ACF) detector. This process involves meticulous annotation of video frames to identify and localize objects of interest. MATLAB's Video Labeler application provides an efficient and user-friendly interface for this task. The following steps detail the systematic approach to ground truth data creation:

1. **Video Importation**: The initial step involves importing the source video into the Video Labeler application. This video should be representative of the scenarios in which the ACF detector will be deployed, encompassing various lighting conditions, object orientations, and environmental factors.

2. **Label Definition**: Prior to annotation, it is essential to define the label categories that correspond to the objects of interest. In the context of vehicle detection, typical labels might include "Car," "Truck," or "Motorcycle." The labeling scheme should be consistent with the intended application of the ACF detector.

3. **Bounding Box Annotation**: The core of the ground truth creation process is the manual annotation of objects within each video frame. This is accomplished by drawing rectangular bounding boxes around the target objects. The process can be approached in two ways:

   (a) **Manual Frame-by-Frame Annotation**:
   - This method involves individually examining each frame and drawing bounding boxes around objects of interest.
   - While time-consuming, this approach ensures high accuracy and allows for precise handling of challenging cases.

   (b) **Semi-Automated Annotation**:
   - MATLAB's Video Labeler offers built-in automation algorithms to expedite the annotation process.
   - These algorithms can interpolate object positions between key frames, significantly reducing the manual workload.
   - However, it is crucial to verify and adjust the automated annotations to maintain accuracy.

4. **Data Export and Verification**: Upon completion of the annotation process, the ground truth data is exported to the MATLAB workspace. This dataset typically comprises:

   - Bounding box coordinates (x, y, width, height) for each labeled object
   - Corresponding frame numbers or timestamps
   - Object class labels

   It is important to perform a thorough verification of the exported data to ensure:

   - Consistency in labeling across frames
   - Accuracy of bounding box placements
   - Completeness of the dataset

4.1.6 ACF Detector Training and Application

Matlab Code:

```
[bbox, score] = detect(detector, frame);
[imList, boxLabels] = objectDetectorTrainingData(gTruth2, SamplingFactor=1);
bboxes = data.car;
data = gTruth.LabelData;
detector = trainACFObjectDetector(imsWithBoxLabels, NumStages=7);
dir *.png;
figure('Position', [100, 100, 800, 600]);
files = imList.Files;
frameLabeled = insertObjectAnnotation(frame, "rectangle", bbox, score);
histogram(score, 10);
imshow(frameLabeled);
imsWithBoxLabels = combine(imList, boxLabels);
```

Explanation:

- **Data Preparation:**
  - `gTruth.LabelData` extracts label data from the ground truth object.
  - `bboxes = data.car` retrieves bounding boxes specifically for the 'car' label.
- **Training Data Organization:**
  - `objectDetectorTrainingData` function organizes the ground truth data into a format suitable for detector training.
  - `SamplingFactor=1` indicates that all frames are used for training.
- **File Management:**
  - `dir *.png` lists all PNG files in the current directory.
  - `files = imList.Files` stores the list of image files.
- **Data Combination:**
  - `combine(imList, boxLabels)` merges image lists with their corresponding bounding box labels.
- **ACF Detector Training:**
  - `trainACFObjectDetector` trains the ACF detector with 7 cascade stages.
- **Object Detection:**
  - `detect(detector, frame)` applies the trained detector to a frame, returning bounding boxes and confidence scores.
- **Visualization:**
  - `insertObjectAnnotation` overlays detection results on the frame.
  - The resulting image is displayed, along with a histogram of detection scores.

4.1.7 Object detection in video

Matlab Code:

```
carVideo = VideoReader("video.mp4");
centroid = [bbox(1)+bbox(3)/2 bbox(2)+bbox(4)/2];
while hasFrame(carVideo)
    frame = readFrame(carVideo);
    [bbox, score] = detect(detector, frame);
    bbox = bbox(score>0, :);
    score = score(score>0);
    if ~isempty(bbox)
        strongestBbox = selectStrongestBbox(bbox, score,...
        NumStrongest=1);
        centroid = [strongestBbox(1)+strongestBbox(3)/2,...
        strongestBbox(2)+strongestBbox(4)/2];
        frame = insertShape(frame, "filled-circle",...
        [centroid 50], 'Color',"green");
    end
    imshow(frame)
    drawnow
end
```

Explanation:

- **Video Processing Loop:**
  - Each frame of the video is processed sequentially.
- **Prediction and Detection:**
  - The ACF detector is applied to each frame.
- **Detection Filtering:**
  - Only detections with positive scores are considered.
- **Centroid Calculation:**
  - The centroid of the bounding box is calculated for position estimation.
- **Object Tracking:**
  - A green circle is drawn at the detected position.
- **Visualization:**
  - `insertShape` overlays detection results on the frame.
  - The frame is displayed with the annotated circles.

4.2 Transition to Deep Learning: YOLO v4

In our pursuit of improving object detection accuracy and robustness, we identified significant limitations in traditional approaches such as Aggregated Channel Features (ACF), particularly in handling diverse object classes and complex visual environments. To address these challenges, we transitioned to a deep learning-based methodology, adopting the You Only Look Once (YOLO) v4 architecture due to its superior performance and versatility.

However, YOLO v4's default configuration was inadequate for detecting our specific object of interest—a remote control car. To leverage the advantages of YOLO v4 without the need to construct a Convolutional Neural Network (CNN) from the ground up, we implemented transfer learning techniques. This approach allowed us to fine-tune the pre-trained YOLO v4 model to effectively recognize and accurately detect the remote control car, thereby enhancing the overall efficiency and accuracy of our object detection system.

4.2.1 The architecture of YOLO

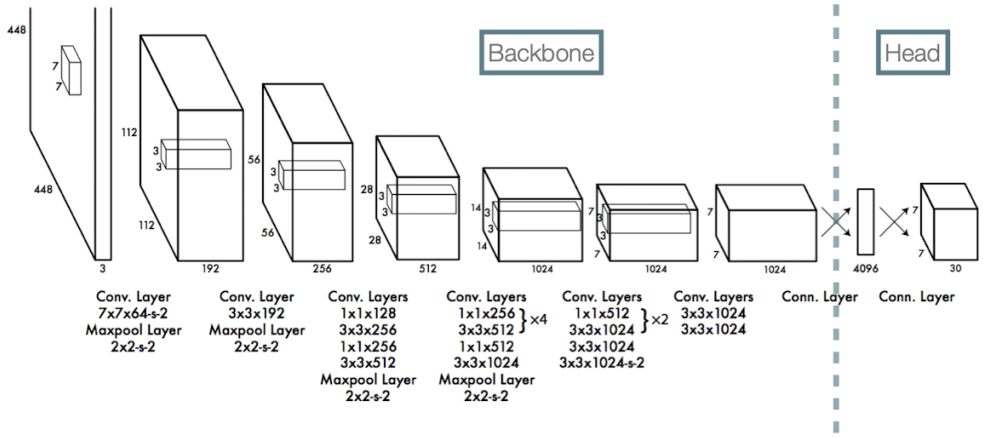

Fig. 4.1: The architecture of YOLO

- **Convolutional Layers:** These layers use filters of different sizes and depths to extract features from the input image.
- **Maxpool Layers:** These layers reduce the size of the feature maps by taking the maximum value in a small window, which helps to reduce the amount of data and focus on the most significant features.
- **Fully Connected Layers:** These layers (shown at the end with the labels "4096" and "30") flatten the feature maps into a one-dimensional vector, allowing the network to make final predictions, such as object class and location in the image.

4.2.2 Advantages of YOLO v4:

- **Versatility:** YOLO v4 excels in detecting a wide range of object classes.
- **Robustness:** It demonstrates superior performance in complex and varied visual environments.
- **End-to-End Learning:** YOLO v4 learns feature representations automatically, potentially capturing more nuanced and task-specific features.
- **Real-Time Processing:** YOLO v4 is designed for real-time applications, providing fast and accurate object detection.
- **High Accuracy:** The model achieves high accuracy in object detection tasks due to its advanced architecture and training techniques.
- **Efficiency:** YOLO v4 is computationally efficient, making it suitable for deployment on devices with limited resources.
- **Scalability:** The architecture of YOLO v4 allows for easy scaling to accommodate different levels of performance and computational power.

4.2.3 How Transfer Learning Works

Pre-training Phase

- A neural network is trained on a large dataset for a general task.
- This network learns a hierarchical representation of features, from low-level to high-level abstractions.

Transfer Phase

- The pre-trained network is then used as a starting point for a new, related task.
- Depending on the similarity of the new task and the amount of available data, different approaches can be taken:

  a. **Feature Extraction**
     * The pre-trained network, excluding the final classification layers, is used as a fixed feature extractor.
     * These extracted features are then fed into a new classifier trained on the target task.

  b. **Fine-tuning**
     * Some or all of the pre-trained network's layers are unfrozen and fine-tuned on the new dataset.
     * This allows the network to adapt its learned features to the specific nuances of the new task.

4.2.4 gTruth Data Preparation for YOLO v4 Using MATLAB's Video Labeler/Image Labeler

Effective training of the YOLO v4 object detection model necessitates a robust dataset, comprising image file paths and corresponding annotated Region of Interests (RoIs) or bounding boxes. Similar to the ACF detector, this preparation is critical for model accuracy. MATLAB provides two powerful tools for this purpose: Video Labeler and Image Labeler.

4.2.5 Structuring the Training Dataset with video labeler

The primary advantage of the Video Labeler tool lies in its ability to leverage built-in automation algorithms to expedite the annotation process. By capturing a sequence of frames from a recorded video, sufficient training images can be generated efficiently. This automation, however, is not without its drawbacks. Built-in algorithms can often lose precision, necessitating manual adjustments to some annotation boxes to ensure accuracy. Following the export of ground truth data, a series of MATLAB functions are employed to structure this information into a format optimized for object detector training.

The steps to operate the Video Labeler, as described in section 4.1.5, include Video Importation, Label Definition, Bounding Box Annotation, Data Export, and Verification.

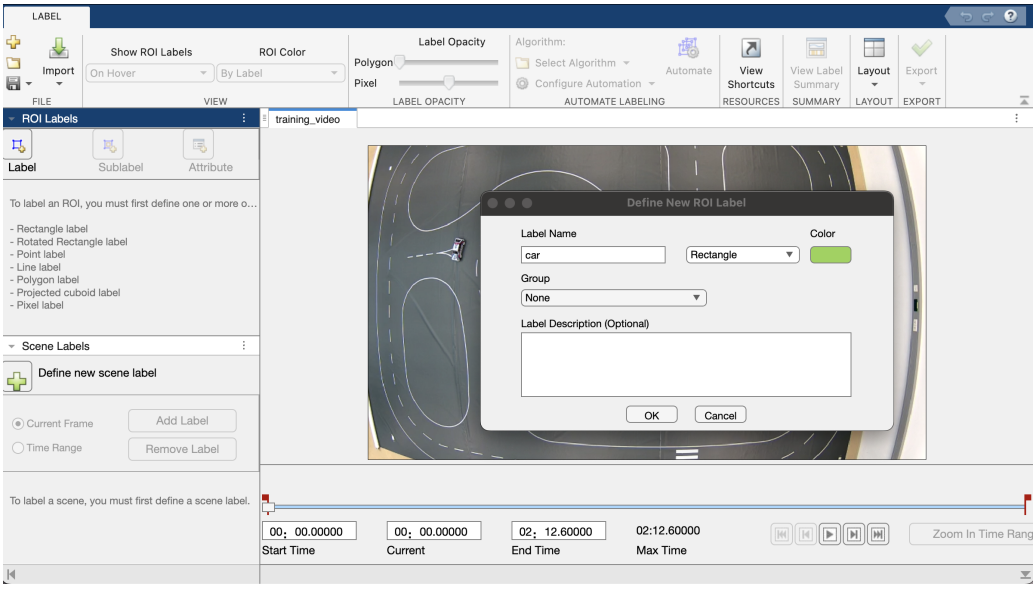

Fig. 4.2: Using the Video Labeler tool in MATLAB to define Regions of Interest (ROI) by creating labels and bounding boxes

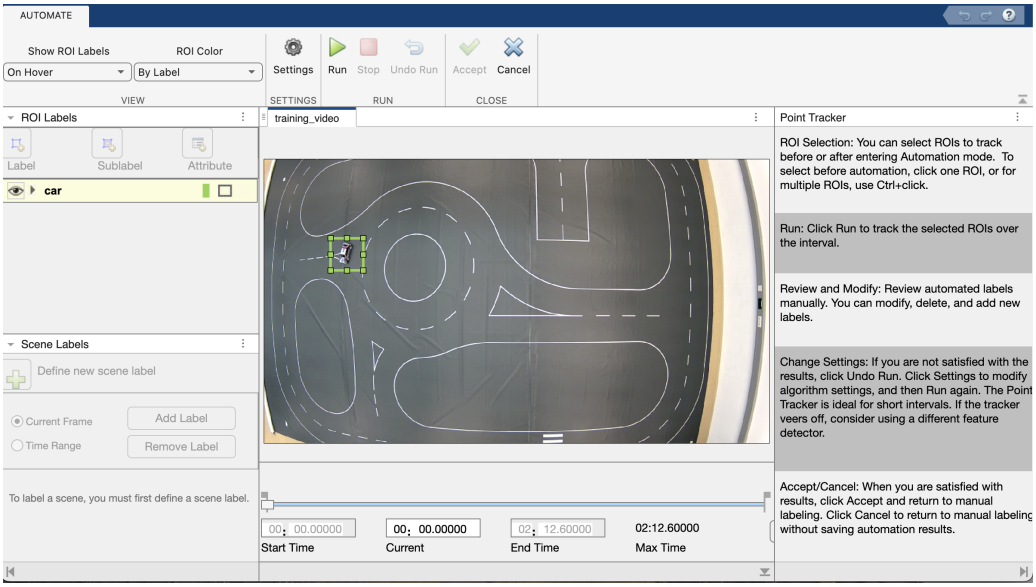

Fig. 4.3: Built-in automation algorithms

4.2.6 Extraction of Image and Label Data

Matlab Code:

```
[imds,blds] = objectDetectorTrainingData(gTruth)
```

The `objectDetectorTrainingData` function is utilized to parse the `gTruth` object and extract two critical components.

Where:

- imds (Image Datastore): A datastore containing file paths to individual video frames or images.
- blds (Box Label Datastore): A datastore containing the corresponding bounding box information and class labels for each image.

This function performs several important tasks:

- Extracts individual frames from the video if necessary.
- Associates each frame with its corresponding labeled bounding boxes.
- Organizes the data into efficient datastore objects for streamlined processing.

4.2.7 Structuring the Training Dataset with Image Labeler

On the other hand, the Image Labeler tool offers a meticulous approach to data preparation, ensuring high-quality training datasets. This tool allows for thorough and precise annotations, guaranteeing diverse perspectives and angles in the training images. The main disadvantage of this method is the considerable amount of time it requires, making it a labor-intensive process.

Matlab Code:

```
trainingData = load("filename of gTruth")
dataDir = "file path of images"
trainingData.imageFilename = fullfile(dataDir,trainingData.imageFilename);
imds = imageDatastore(pwd,'FileExtensions',...
       {'.png', '.jpg', '.JPEG', '.tif', '.tiff'});
blds = boxLabelDatastore(trainingData(:,2:end));
```

1. **Loading Labeled Data**:

   ```
   trainingData = load("filename of gTruth");
   ```

2. **Specifying Data Directory**:

   ```
   dataDir = "file path of images"
   ```

   This line defines the directory where the image files are stored. It's crucial to ensure that this path correctly points to the location of your image dataset.

3. **Updating Image Filename**:

   ```
   trainingData.imageFilename = fullfile(dataDir, trainingData.imageFilename);
   ```

4. **Creating Image Datastore and Box Label Datastore**:

```
imds = imageDatastore(pwd, 'FileExtensions',...
{'*.png', '*.jpg', '*.JPEG', '*.tif', '*.tiff'});
blds = boxLabelDatastore(trainingData(:, 2:end));
```

The data preparation process outlined above transforms raw labeled data from Image Labeler into a structured format suitable for training object detection models. By carefully organizing the image files and their corresponding annotations into datastores, we create a robust foundation for subsequent model training and evaluation stages.

4.2.8 Code Implementation of YOLO v4 Transfer Learning

Introduction
This section details the process of preparing data and training a YOLO v4 object detector using MATLAB. The procedure involves combining image and label data, estimating anchor boxes, and configuring the training process for the detector.

Matlab Code:

```
ds = combine(imds, blds);
inputSize = [416 416 3];
trainingDataForEstimation = transform(ds,...
@(data)preprocessData(data,inputSize));
numAnchors = 6;
[anchors, meanIoU] = estimateAnchorBoxes(trainingDataForEstimation,...
numAnchors);
area = anchors(:,1).*anchors(:,2);
[~,idx] = sort(area,"descend");
anchors = anchors(idx,:);
anchorBoxes = {anchors(1:3,:);anchors(4:6,:)};
classes = ["car"];
detector = yolov4ObjectDetector("tiny-yolov4-coco",classes,...
anchorBoxes,InputSize=inputSize);

options = trainingOptions("sgdm", ...
    InitialLearnRate=0.0001, ...
    MiniBatchSize=64,...
    MaxEpochs=40, ...
    ResetInputNormalization=false,...
    VerboseFrequency=30,Plots="training-progress");

 trainedDetector = trainYOLOv4ObjectDetector(ds,detector,options);
```

4.2.9 Combining Image and Label Data

```
ds = combine(imds, blds)
```

This line combines the image datastore (`imds`) and box label datastore (`blds`) into a single datastore `ds`, ensuring that images and their corresponding labels are paired correctly.

4.2.10 Defining Input Size

```
inputSize = [416 416 3];
```

Specifies the input size for the YOLO v4 network. The dimensions `[416 416 3]` represent width, height, and color channels respectively. The input size should always be a multiple of 32. This is due to the network architecture and how it downsamples the input. Larger input sizes (e.g., 608x608 or 512x512) can improve detection accuracy, especially for small objects. However, this increases computational cost.

4.2.11 Preprocessing Data

```
trainingDataForEstimation = transform(ds,@(data)preprocessData(data,inputSize));
```

Applies a preprocessing function to the combined datastore, resizing images to the specified input size.

```
numAnchors = 6;
[anchors, meanIoU] = estimateAnchorBoxes(trainingDataForEstimation,numAnchors);
```

Estimates optimal anchor box sizes based on the training data. Anchor boxes are predefined bounding box shapes that the network uses as references for detection.

4.2.12 Sorting and Organizing Anchor Boxes

```
area = anchors(:,1).*anchors(:,2);
[ ,idx] = sort(area,"descend");
anchors = anchors(idx,:);
anchorBoxes = anchors(1:3,:);anchors(4:6,:);
```

Sorts anchor boxes by area in descending order and organizes them into two groups of three for use in the YOLO v4 architecture.

4.2.13 Defining Classes and Creating Detector

```
classes = ["car"];

detector = yolov4ObjectDetector("tiny-yolov4-coco",...
classes,anchorBoxes,InputSize=inputSize);
```

Specifies the object classes to detect (in this case, only "car") and creates a YOLO v4 detector using a pre-trained "tiny-yolov4-coco" model as a starting point.

4.2.14 Configuring Training Options

```
options = trainingOptions("sgdm", ...
InitialLearnRate=0.0001, ...
MiniBatchSize=64, ...
MaxEpochs=40, ...
ResetInputNormalization=false, ...
VerboseFrequency=30, Plots="training-progress");
```

Training Options Setup:

- **Optimization Algorithm:** Stochastic Gradient Descent with Momentum (SGDM) was chosen for its faster convergence with our training data compared to other algorithms like ADAM. Further SGDM prevents the algorithm from getting stuck at local minima and saddle points of the loss function.
- **Learning Rate:** The learning rate for SGDM determines the step size at each iteration while moving toward a minimum of the loss function. It was set to 0.0001, based on experimentation, to allow the training loss function to converge successfully. This relatively low value helps prevent overshooting the minimum and allows for more precise updates to the model parameters.
- **Batch Size:** Batch size refers to the number of training examples utilized in one iteration. A mini-batch size of 64 was selected, balancing computational limitations and model performance. This size allows for more frequent updates to the model compared to full-batch training, while still providing a reasonable estimate of the gradient and taking advantage of hardware acceleration for matrix operations.
- **Number of Epochs:** An epoch represents one complete pass through the entire training dataset. The maximum number of epochs was determined by monitoring the training process and halting it when the loss reached a sufficiently low value (e.g., 0.01). This approach ensures that the model trains long enough to learn effectively, but stops before overfitting occurs or when further training yields diminishing returns.
- **Additional Settings:**
  - `ResetInputNormalization=false` to maintain consistent input normalization.
  - `VerboseFrequency=30` to control the frequency of progress updates.
  - `"Training-progress"` plot option enabled to visualize the training process.

Fine-tuning these options is crucial for achieving a well-trained and effective object detection model.

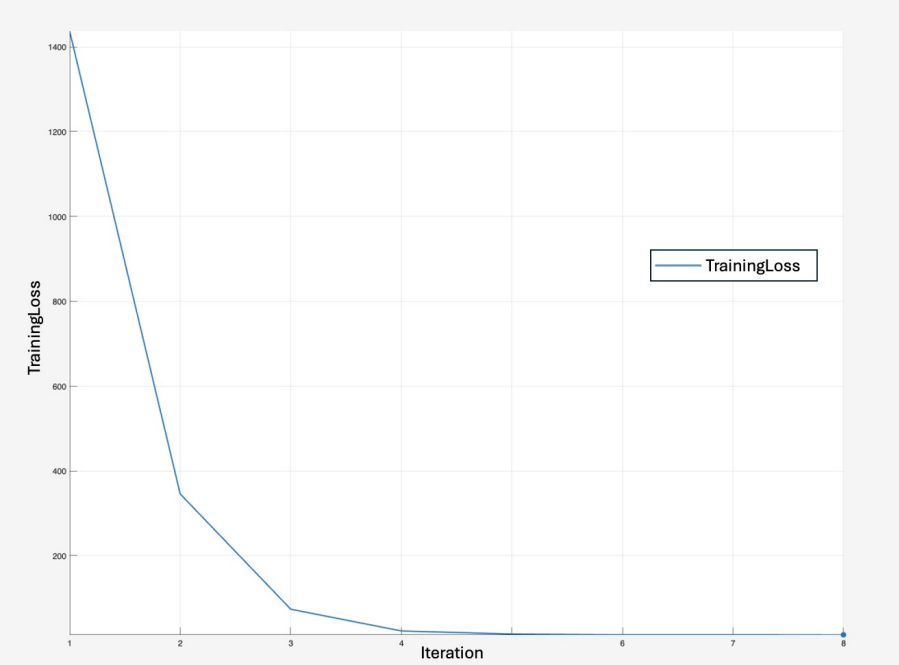

Fig. 4.4: Training loss function

4.2.15 Training the Detector

```
trainedDetector = trainYOLOv4ObjectDetector(ds, detector, options);
```

This line of code executes the training process using the prepared data, detector configuration, and training options.

4.2.16 Implementing Object Detection in Various Media

Static Image Detection:

Matlab Code:

```
I = imread("01.JPEG");
[bboxes, scores, labels] = detect(trainedDetector, I, Threshold=0.05);
detectedImg = insertObjectAnnotation(I, "Rectangle", bboxes, labels);
figure
imshow(detectedImg)
```

This code segment loads an image, applies the trained detector with a threshold of 0.05, and visualizes the results. The `detect` function returns bounding boxes, confidence scores, and labels for detected objects. These are then overlaid on the original image using `insertObjectAnnotation`.

Video File Processing:

Matlab Code:

```
videoFile = 'video file name';
videoReader = VideoReader(videoFile);
outputVideoFile = 'output file name'.mp4';
videoWriter = VideoWriter(outputVideoFile, 'MPEG-4');
open(videoWriter);

while hasFrame(videoReader)
    frame = readFrame(videoReader);
    [bboxes, scores, labels] = detect(trainedDetector,...
    frame, 'Threshold', 1);
    detectedImg = insertObjectAnnotation(frame, 'rectangle',...
    bboxes, labels);
       writeVideo(videoWriter, detectedImg);
end
close(videoWriter);
```

This section demonstrates object detection in a video file. It reads frames sequentially, applies the detector to each frame, and writes the annotated frames to a new video file. The threshold is set to 1, which may need adjustment depending on the specific use case.

Real-time Camera Feed Detection:

Matlab Code:

```
v = videoinput("cam");
figure;
while true
    frame = getsnapshot(v);
    [bboxes, scores, labels] = detect(trainedDetector,...
    frame, 'Threshold', 0.02);
    detectedImg = insertObjectAnnotation(frame,..
    'rectangle', bboxes, labels);
    imshow(detectedImg);
end
```

This code snippet illustrates real-time object detection using a camera feed. It continuously captures frames, applies the detector, and displays the results. The detection threshold is set to 0.02, allowing for sensitive detection suitable for real-time applications.
To perform real-time video object detection in MATLAB, a multi-step process is required:

- **Hardware Setup**
  - Connect a compatible camera to the computer.
  - Ensure proper drivers are installed for seamless communication.
- **MATLAB Image Acquisition Toolbox**
  - Utilize MATLAB's Image Acquisition Toolbox to interface with the camera.
- **Camera Configuration**
  - In the Image Acquisition Explorer, detect and select the connected camera.
  - Configure camera settings such as resolution, frame rate, and color space as needed for the application.
- **Video Stream Acquisition**
  - Create a video input object in MATLAB using the `videoinput` function.
  - Specify the appropriate adapter and device ID for the camera.

4.3 Conclusion of Object Detection

Through the steps above, we have explained how to achieve object detection using transfer learning, from data preparation to fine-tuning the YOLOv4 detector, and finally implementing detection in static images, videos, and real-time video. In the next section, we will explain how to use an Extended Kalman Filter (EKF) for object tracking

5 Object Tracking

Object tracking is a fundamental task in computer vision, widely used in applications such as robotics, surveillance, and autonomous driving. It involves detecting and following objects of interest within a sequence of video frames. Accurate object tracking is crucial for many applications, including autonomous vehicle navigation, where understanding the movement of surrounding vehicles is essential for safe and efficient driving.

In this report, we explore the implementation of an Extended Kalman Filter (EKF) for tracking a vehicle detected by a YOLOv4 deep learning model. The Kalman Filter is well-suited for linear systems; however, many real-world systems exhibit non-linear behaviour. The EKF extends the Kalman Filter to handle these non-linear systems by linearizing around the current estimate using a first-order Taylor expansion. This makes the EKF more versatile and accurate for tracking objects in dynamic environments where the state transition and measurement models are non-linear. EKF is chosen due to its ability to handle non-linearities in the system dynamics and measurement processes, providing robust tracking even under noisy conditions.

5.1 Components of Object Tracking

The object tracking consists of two main components:

- **Object Detection:** To integrate Extended Kalman Filter with object detection in real time setup we will use EKF Algorithm with the help of trained detector. The EKF will use detected positions as measurements and estimate the vehicle's position, velocity and heading angle over time.
- **Tracking with EKF:** Once the objects are detected, the second component, the Extended Kalman Filter (EKF), is employed to track these detected objects. The EKF plays a crucial role in smoothing the detected objects' trajectories and predicting their future positions. This is achieved by considering the non-linearities in the motion and measurement processes, allowing the EKF to provide robust tracking even in dynamic and noisy environments.

Together, these components ensure accurate and reliable object tracking, which is essential for applications such as autonomous driving and advanced surveillance systems.

5.2 Integration of Object Detection with EKF Algorithm

To integrate Extended Kalman Filter with object detection in real time setup we will use EKF Algorithm with the help of trained detector. The EKF will use detected positions as measurements and estimate the vehicle's position, velocity and heading angle over time.

We first load the trained detector:

```
load('trained_detector.mat');
```

5.2.1 EKF Algorithm Setup:

Initialization of parameters: We use the linear bicycle model for initializing the state vectors. The linear bicycle model describes motion of the vehicle in terms of position, velocity and heading angle. The State vector for our application is defined as

$$\mathbf{X} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{2}$$

Where:
x and y are the position of the vehicle in the map
θ is the heading angle
θ can be calculated from the positions (x and y) by,

$$\theta_k = \tan^{-1}\left(\frac{y_{k+1} - y_k}{x_{k+1} - x_k}\right) \tag{3}$$

Covariance Matrices: The initial covariance matrix P is assumed to be an identity Matrix, indicating the initial uncertainty in the state estimates. Similarly, process noise (R) and measurement covariance (Q) matrices are used to indicate uncertainties in process and measurement model respectively.

Initial Covariance Matrix is given by

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

The covariance matrix for motion model is given by

$$R = \begin{bmatrix} \sigma_x^2 & 0 & 0 \\ 0 & \sigma_y^2 & 0 \\ 0 & 0 & \sigma_z^2 \end{bmatrix} = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.05 \end{bmatrix} \tag{5}$$

The covariance matrix for measurement model is given by

$$Q = \begin{bmatrix} \sigma_{mx}^2 & 0 \\ 0 & \sigma_{my}^2 \end{bmatrix} = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix} \tag{6}$$

The suitable values of the uncertainties in position and measurements are chosen based on the trial and error method by observing the performance of the filter.

Matlab Code:

```
X=[ x_position; y_position;0] %Initial states
P = eye(3);     % Initial covariance matrix
R = diag([0.1, 0.1, 0.05]); % Process noise covariance
Q = diag([0.1, 0.1]); % Measurement noise covariance
```

Time step and control inputs: The time step dt is set to 1 second. The control input u (velocity and angular velocity) is also defined. The initial velocity is assumed to be 1m/s and initial angular velocity is 0.1 m/s.

```
dt = 1;
u = [1; 0.1]; % [velocity; angular_velocity]
```

5.2.2 State transition function:

The state transition function describes how the state evolves over time based on the control inputs and the current state. For the linear bicycle model, the state transition function can be written as:

$$x_{k+1} = f(x_k, u_k) \tag{7}$$

The initial State vector [$x_{centroid}$, $y_{centroid}$, θ], represents initial position (x,y) and orientation of the vehicle at time t in the map. The control inputs here are velocity and angular velocity of the vehicle. Here the state vector is given as state transition function.

Matlab Code:

```
function x_next = stateTransitionFcn(x, u, dt)
 % State vector x = [x_position; y_position; theta]
 % Control input u = [velocity; angular_velocity]
 % Time step dt
theta = x (3);
v = u (1);
omega = u(2);
x_next = x + [v * cos(theta) * dt;
v * sin(theta) * dt;
omega * dt];
end
```

5.2.3 Measurement function:

Here the measurements are x and y position of the vehicle. The measurement z is obtained from the simulated measurements. The measurements are converted into function.

Matlab Code:

```
function z = measurementFcn(x)
 % State vector x = [x_position; y_position; theta]
 z = [x(1); x(2)];  % We measure x and y positions
end

% Simulate some measurements
true_state = x;
measurements = [];
for t = 0:dt:100
 true_state = stateTransitionFcn(true_state, [1; 0.1], dt);
 measurements = [measurements; measurementFcn(true_state)'];
end
```

5.2.4 Derivation of Jacobian for state transition function:

The Jacobian matrix size depends on the number of states. The Jacobian of the state transition function, F, is required to develop EKF. To derive the Jacobian matrix F, we consider the state transition of the system over time dt. The kinematic model of the vehicle is given by,

$$x_{k+1} = x_k + v_k cos\ \Theta_k dt$$
$$y_{k+1} = y_k + v_k sin\ \theta_k dt \tag{8}$$

$$\theta_{k+1} = \theta_k \tag{9}$$

Now the state transition function f(x_k, u_k) can be written as,

$$f_k(x_k, u_k) = \begin{bmatrix} x_k + v_k \cos\theta_k dt \\ y_k + v_k \sin\theta_k dt \\ \theta_k \end{bmatrix} \tag{10}$$

The Jacobian matrix F is the partial derivative of function f with respect to the state vector X:

$$F = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial \theta} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial \theta} \\ \frac{\partial f_3}{\partial x} & \frac{\partial f_3}{\partial y} & \frac{\partial f_3}{\partial \theta} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -v\sin\theta_k dt \\ 0 & 1 & v\cos\theta_k dt \\ 0 & 0 & 1 \end{bmatrix} \tag{11}$$

Matlab Code:

```
function F = jacobianStateFcn(x, u, dt)
theta = x(3);
v = u(1);
F = [1, 0, -v * sin(theta) * dt;
0, 1, v * cos(theta) * dt;
0, 0, 1];
end
```

5.2.5 Derivation of Jacobian for measurement function:

We assume x and y are the measurements associated for this system. So, the measurement function h(x) can be written as:

$$h(x_k) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{12}$$

The Jacobian matrix H is the partial derivative of function h with respect to the state vector X:

$$H = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial \theta} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial \theta} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{13}$$

Matlab Code:

```
function H = jacobianMeasurementFcn(~)
H = [1, 0, 0;
0, 1, 0];
end
```

5.2.6 Setup Video Detection:

The initial parameters for the algorithm have been setup. Now the camera must be accessed and extract the data from object detection for the state estimation. A threshold of 0.5 has been set to filter the detections based on confidence score. Also, the centroid is calculated from the strongest bounding box.

Matlab Code:

```
v = videoinput("cam");
figure;
trackedLocations= [];
kfInitialized = false;
while hasFrame(v)
frame = readFrame(v);
[bbox, score, labels]= detect(trainedDetector, frame,...
'Threshold', 0.5);
idx = score >= threshold;
bbox = bbox(idx, :);
label = label(idx);
if ~isempty(bbox)
strongestBbox = selectStrongestBbox(bbox, score(idx),...
'OverlapThreshold', 0.5);
centroid = [strongestBbox(1) + strongestBbox(3) / 2,...
strongestBbox(2) + strongestBbox(4) / 2];
........
```

5.2.7 EKF Prediction stage:

Using the state space model, the state estimate is predicted at time t+1 based on the current estimate and control input at time t. Also the state covariance is estimated based on the previous covariance and some noise.

State prediction: The next state x_{pred} is predicted using state transition function, 'stateTransitionFcn', which models the vehicle's motion.

Jacobian calculation: The Jacobian Matrix F of the state transistion function is calculated using the current state and control inputs.

Covariance prediction: The predicted covariance matrix P_{pred} is calculated using the Jacobian Matrix F and the process noise covariance R.

Matlab Code:
While loop continues...

```
if ~kfInitialized
x = [centroid, 0]';
kfInitialized = true;
else
% EKF Prediction
u = [1; 0.1]; % control inputs, should be estimated
x_pred = stateTransitionFcn(x, u, dt);
F = jacobianStateFcn(x, u, dt);
P_pred = F * P * F' + R;
```

5.2.8 EKF-Update/Correction Stage:

Kalman gain: The Kalman Gain K is calculated using the predicted covariance matrix P_{pred} and the measurement noise covariance Q.

State update: The state update x is updated using Kalman Gain K and the residual $z - h(x_{pred})$.

Covariance update: The covariance matrix P is updated using Kalman gain K and Jacobian, H.

Estimation of velocity and angular velocity: Without direct measurements from an odometer or LIDAR, we rely on the predicted state and the process model to estimate the future states. Here the measurements are only the positions i.e. the centroid of the vehicle (x and y) which can be extracted from the results of the object detection. So, the velocity and heading angle can be estimated from the position and heading angle estimates by:

$$v_k = \frac{\sqrt{(x_{k+1} - x_k)^2 + (y_{k+1} - y_k)^2}}{dt} \tag{14}$$

$$\omega_k = \frac{\theta(t + dt) - \theta(t)}{dt} \tag{15}$$

Matlab Code:
While loop continues...

```
 % EKF Update
 z = centroid';
 H = jacobianMeasurementFcn(x_pred);
 K = P_pred * H' / (H * P_pred * H' + R);
 x = x_pred + K * (z - measurementFcn(x_pred));
 P = (eye(3) - K * H) * P_pred;
 end %(if ~kfInitialized loop ends...)

 % Save tracked location
 trackedLocation = x(1:2)';
 trackedLocations = [trackedLocations; trackedLocation];
 detectedFrame = insertShape(frame,'FilledCircle',
 [trackedLocation, 5], 'Color', 'green', 'LineWidth', 2);
else
 % Predict if there are no detections

 % Estimate Angular velocity
 theta_prev = x_pred(3);
 theta_curr = x(3);
 omega_est = (theta_curr - theta_prev) / dt;
 % Estimate Velocity
 pos_prev = x_pred(1:2);
 pos_curr = x(1:2);
 v_est = norm(pos_curr - pos_prev) / dt;

 u = [v_est; omega_est]; % should be estimated
 x_pred = stateTransitionFcn(x, u, dt);
 F = jacobianStateFcn(x, u, dt);
 P_pred = F * P * F' + R;
 x = x_pred;
 P = P_pred;



% Save predicted location
trackedLocation = x(1:2)';
trackedLocations = [trackedLocations; trackedLocation];
detectedFrame = insertShape(frame, 'FilledCircle',...
[trackedLocation, 5],'Color', 'blue', 'LineWidth', 2);
end %(if ~isempty(bbox) ends)
end(while loop ends)
```

For each frame, the detector identifies the object and EKF predicts and updates the state based on detected positions. The predicted and updated positions are drawn on the video frames. If there are no detections in certain video frames, the positions are estimated by calculating the velocity and angular velocity and states are estimated. This allows better tracking especially in corners and around the obstacles.

5.3 Result of object tracking integrated with object detection

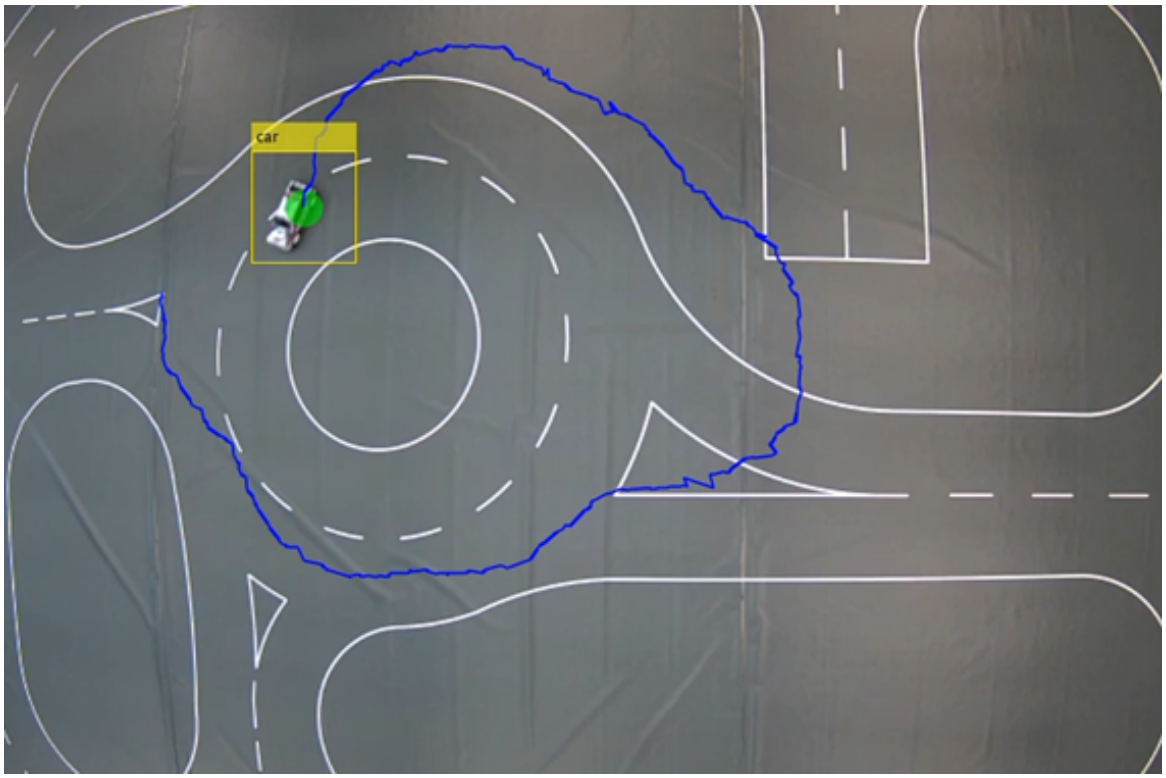

Fig. 5.1: Result of EKF tracking

The figure 5.1 shows the trajectory of the vehicle estimated with EKF. The green circle on the image is the centroid of the detected vehicle and the implemented EKF follows and estimates the future states of the vehicle. Unlike the standard Kalman filter, which struggles with non-linearities around corners, the Extended Kalman Filter (EKF) effectively estimates the trajectory and heading angle by accurately handling these non-linearities.

6 Conclusion

The object can be accurately located within a world coordinate system using the camera calibration algorithm. The high distortion previously observed at the camera's edges has been effectively corrected by positioning a checkerboard pattern at various locations within the camera's view. The overall mean calibration error from the 19 calibration images was 0.09 pixels. Accuracy in world coordinate estimation was higher in the region directly below the camera compared to the periphery. Additionally, using a higher number of calibration images results in better estimation of camera intrinsics.

While the ACF object detector has its merits, the YOLOV4 algorithm, when paired with the SGDM optimizer, consistently and successfully detected the object of interest in all frames. YOLOV4 outperforms ACF due to its real-time detection capabilities, higher accuracy, and implementation on a GPU, which significantly enhances its training and processing speed as well as efficiency. ACF, on the other hand, lacks GPU implementation, making it slower and less efficient for rapid object identification tasks.

The Kalman Filter algorithm implemented initially struggles with handling non-linearities in object motion. In contrast, the introduction of the Extended Kalman Filter (EKF) effectively predicts object movements with non-linearity by incorporating the heading angle and estimating the angular velocity. EKF is sensitive to measurement and process noise. It is important to tune covariance matrices based on empirical data to mitigate the noise effects. Also, poor initial state estimates can lead to divergence. It is better to have additional sensors like odometry and LIDAR for better state estimation and avoid noises in the trajectory. EKF can struggle with significant non-linearities in the system model. The solution is to implement higher-order filters like the Unscented Kalman Filter (UKF) to handle these non-linearities better.

Realtime tracking is done using ipcam function, which doesn't allow us to control the frame rate and resolution of the video. The changes in resolution makes it challenging to integrate the camera calibration and object detection results. While reading the video frames in Matlab, the high resolution leads to jittering and lag in processing. Object detection necessitates a large number of training images to enhance detection efficiency, which also alters the optimal values for the learning rate and number of iterations, thereby increasing computational cost and decreasing efficiency. Additionally, when the object is behind an obstacle or out of frame, the region of interest must be manually drawn for all consecutive frames using the image labeler app in MATLAB, making the process highly time-consuming and labor-intensive.

7 Future Work

While the initial implementation of the camera-based localization and tracking system has shown promising results in a scaled-down environment, there is significant potential for improvement and expansion. To enhance the system's capabilities and address limitations, the following areas should be explored:

System Enhancements

- **Multi-camera Integration:** Expanding the system to incorporate multiple cameras can enhance object detection and tracking accuracy, especially in complex scenarios with occlusions. By fusing data from different viewpoints, the system can achieve a more comprehensive understanding of the environment.
- **Real-time Performance Optimization:** Further optimizing the system's computational efficiency can enable real-time operation on resource-constrained platforms, making it suitable for deployment in various applications beyond scaled-down environments.
- **Image and Video Processing:** Developing methods to ensure consistent image and video resolution is crucial for accurate calibration and object measurement. Investigating techniques to address image distortion and noise can improve overall system performance.
- **System Stability:** Understanding and mitigating the causes of jittering in real-time visual data is essential for reliable object tracking. Optimizing processing pipelines and exploring hardware acceleration options can help reduce processing time and improve system responsiveness.

Algorithm Refinements

- **Multi-Object Tracking:** Expanding the system's capabilities to handle multiple objects simultaneously will be crucial for real-world applications. Implementing advanced multi-object tracking algorithms can improve accuracy and robustness.
- **State Estimation Model:** Refining the state transition model used for object tracking can enhance prediction accuracy and reduce tracking errors. Incorporating additional information about object dynamics and environmental factors can improve model performance.

By addressing these areas for future work, the camera-based localization and tracking system can be further developed into a versatile and robust tool for supporting the advancement of autonomous vehicle technology.

References

[1] P. Corke, W. Jachimczyk, R. Pillat, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, 3rd edition. Springer, 2023

[2] T. Omeragic, J. Velagic, "Tracking of Moving Objects Based on Extended Kalman Filter," in *Conf. International Symposium ELMAR*, Zadar, Croatia, 2020, pp.137-140.

[3] C. Stachniss. (2020). *Photogrammetry & Robotics Lab: Kalman Filter and Extended Kalman Filter* [Online]. Available: https://www.ipb.uni-bonn.de/html/teaching/photo12-2021/2021-pho2-14-ekf.pptx.pdf

[4] Mathworks. *Measuring Planar Objects with a Calibrated Camera* [Online]. (accessed April, 2024). Available: https://in.mathworks.com/help/vision/ug/measuring-planar-objects-with-a-calibrated-camera.html

[5] Mathworks. *Train ACF object detector* [Online]. (accessed April, 2024). Available: https://in.mathworks.com/help/vision/ref/trainacfobjectdetector.html

[6] Mathworks. *Object Detection Using YOLO v4 Deep Learning* [Online]. (accessed April, 2024). Available: https://in.mathworks.com/help/vision/ug/object-detection-using-yolov4-deep-learning.html

[7] Mathworks. *Train YOLO v4 object detector* [Online]. (accessed May, 2024). Available: https://in.mathworks.com/help/vision/ref/trainyolov4objectdetector.html

[8] Mathworks. *Extended Kalman Filter* [Online]. (accessed June, 2024). Available: https://in.mathworks.com/help/ident/ref/ekf_block.html

List of Figures