Project Overview

**Introduction**

This project aims to develop a self-driving car using a Raspberry Pi 5, equipped with a camera and an AI Halio 8L Machine Learning Kit. The vehicle will navigate a predefined track by detecting blue cones on the left and orange cones on the right using OpenCV and machine learning techniques. As the system gains confidence in its navigation, it will progressively accelerate, enhancing performance and efficiency.

**System Components**

1. **Hardware:**
   * **Raspberry Pi 5:** Central processing unit for running the software and handling computations.
   * **Camera Module:** Captures real-time video feed for environment sensing.
   * **AI Halio 8L Machine Learning Kit:** Provides additional processing power for machine learning tasks.
   * **Motor Controllers & Motors:** Drive the vehicle based on control signals.
   * **Power Supply:** Ensures all components receive adequate power.
2. **Software:**
   * **Operating System:** Raspberry Pi OS or a suitable Linux distribution.
   * **OpenCV:** For image processing and color detection.
   * **Machine Learning Frameworks:** TensorFlow – This is the machine learning development software for integration into pyton. I think this is the easiest way to intgrate
   * **Custom Python Scripts:** Comms, decision making, others TBD

**Functional Flow**

1. **Image Capture:**
   * The camera captures live video feed of the track environment.
2. **Color Detection:**
   * OpenCV processes each frame to identify blue cones on the left and orange cones on the right.
3. **Decision Making:**
   * Based on the detected cone positions, the system determines the steering angle and speed.
   * Machine learning algorithms assess confidence levels in navigation and adjust acceleration accordingly.
4. **Control Signals:**
   * The system sends appropriate signals to the motor controllers to steer and drive the vehicle.
5. **Learning and Adaptation:**
   * The system records navigation data to learn the track layout, improving accuracy and speed over time.

**File Structure and Descriptions**

To facilitate collaboration between two developers, the project is organized into distinct modules with clear responsibilities.

* Main.py – Integrates all modules and manages main loop. Initilizes all components and handles syncro
* Camera
  + Camera Initilization – configure and initialize camera hardware set resolution frame rate and error handling
  + Camera frames python – continuously capture frames from the camera provide frames to perception module for proccessing
* Perception
  + Color detection python script – detect blue and orange cones in captures frames.. Implement color and edge detection. Find point at top of cone or center of body? Edge detection?
  + Image Processing – possibly refine images to improve processing speed? Apply filters? Edge detection, otother image processing techniques
  + Vision utiulities – helper functions for vision related task. Drawing boxes, onscreen debugging tool, coordinate transformation and other GUI to help prototyping
* Controls
  + Steering control – determine steering angle based on received cone position. Calculate and send updated steering angle to wheels. Log angles with respect to time to give a default control technique for second lap. Assume 2x as fast.
  + Speed control
    - Determine accuracy of of detection, adjust speed depending on success. Control 3 initial laps to set speeds. Compute predicted controls for next lap.
  + Motor interface? This is optional and my be delegated to MCU – Is this all done over rs232? Is that fast enough?
* Machine learning
  + Model training
  + Training machine learning models for navigation confidence collect data and train. Save for future use. This likely needs to be done on a desktop computer with a GPU and video footage?
  + Model inference – preforme realtime inference to assess navigation confidence. Load trained models and predict levels based on input data.
  + Datacollection collect and pre process data for training. Store navigation data label cone cdetection and prep datasets
* Utilities
  + Config store configuration parameters and constrains for easy adjusting and fine tuning later
  + Logger – handle logging for debugging and monitoring during testing
  + Helpers
* Communication
  + Frame design
  + Sending receiving to MCU?
* Requirements – list all includes and sections.
* Readme

TASK LIST

1. Develop and maintain the camera/ and perception/ modules.
2. Implement color detection algorithms in color\_detection.py.
3. Handle image preprocessing in image\_processing.py.
4. Create utility functions in vision\_utils.py.
5. Develop and maintain the control/ and machine\_learning/ modules.
6. Implement steering and speed control logic in steering\_control.py and speed\_control.py.
7. Interface with motors via motor\_interface.py.
8. Handle model training, inference, and data collection in the machine\_learning/ submodules.

Bellow are AI generated examples of each files.

CAMERA

### \*\*Camera Module Implementation for Raspberry Pi 5-Based Self-Driving Car\*\*

The \*\*Camera Module\*\* is a crucial component of the self-driving car project, responsible for capturing real-time video feeds of the track environment. This module consists of two main scripts:

1. \*\*`camera\_setup.py`\*\*: Configures and initializes the camera hardware, setting parameters such as resolution and frame rate while handling potential initialization errors.

2. \*\*`capture\_frames.py`\*\*: Continuously captures frames from the camera and provides them to the perception module for further processing.

Below are the detailed implementations of these scripts, along with explanations to facilitate understanding and integration into the larger project.

---

#### \*\*1. `camera\_setup.py`\*\*

This script initializes the camera, sets the desired resolution and frame rate, and ensures that the camera is ready for capturing frames. It uses the `picamera` library, which is optimized for Raspberry Pi camera modules.

\*\*Dependencies:\*\*

- `picamera`: For interfacing with the Raspberry Pi Camera Module.

- `logging`: For logging initialization processes and errors.

\*\*Installation:\*\*

Ensure that the `picamera` library is installed. If not, install it using pip:

```bash

pip install picamera

```

\*\*Code:\*\*

```python

# camera\_setup.py

import logging

from picamera import PiCamera

from time import sleep

class CameraSetup:

def \_\_init\_\_(self, resolution=(640, 480), framerate=30):

"""

Initialize the camera with specified resolution and frame rate.

Args:

resolution (tuple): The desired resolution (width, height).

framerate (int): The desired frame rate.

"""

self.resolution = resolution

self.framerate = framerate

self.camera = None

self.logger = logging.getLogger('CameraSetup')

self.logger.setLevel(logging.DEBUG)

ch = logging.StreamHandler()

ch.setLevel(logging.DEBUG)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

ch.setFormatter(formatter)

self.logger.addHandler(ch)

self.initialize\_camera()

def initialize\_camera(self):

"""Configure and initialize the camera hardware."""

try:

self.logger.info("Initializing camera...")

self.camera = PiCamera()

self.camera.resolution = self.resolution

self.camera.framerate = self.framerate

self.camera.iso = 100 # Set ISO for consistent lighting

sleep(2) # Allow camera to warm up

self.logger.info(f"Camera initialized with resolution {self.resolution} and framerate {self.framerate}fps.")

except Exception as e:

self.logger.error(f"Failed to initialize camera: {e}")

raise

def get\_camera(self):

"""Return the initialized camera object."""

if self.camera is None:

self.logger.error("Camera not initialized.")

raise ValueError("Camera not initialized.")

return self.camera

```

\*\*Explanation:\*\*

- \*\*Logging Setup:\*\* Initializes a logger to track the camera initialization process and handle errors.

- \*\*Initialization Parameters:\*\* Sets default resolution to 640x480 and frame rate to 30fps, which can be adjusted as needed.

- \*\*`initialize\_camera` Method:\*\* Attempts to initialize the PiCamera with the specified settings. It includes a warm-up period (`sleep(2)`) to allow the camera's sensors to stabilize.

- \*\*Error Handling:\*\* Catches and logs any exceptions that occur during initialization, ensuring that failures are reported clearly.

---

#### \*\*2. `capture\_frames.py`\*\*

This script captures frames from the initialized camera in real-time and makes them available to other modules (e.g., perception) for processing. It uses multithreading to ensure that frame capturing does not block other processes.

\*\*Dependencies:\*\*

- `picamera`: For interfacing with the Raspberry Pi Camera Module.

- `cv2` (OpenCV): For handling image frames.

- `logging`: For logging frame capture processes.

- `threading` and `queue`: For managing concurrent frame capturing and processing.

\*\*Installation:\*\*

Ensure that OpenCV is installed. If not, install it using pip:

```bash

pip install opencv-python

```

\*\*Code:\*\*

```python

# capture\_frames.py

import logging

import cv2

import threading

import time

from queue import Queue, Empty

from camera\_setup.py import CameraSetup

class FrameCapture:

def \_\_init\_\_(self, camera\_setup, frame\_queue, queue\_size=10):

"""

Initialize the frame capture process.

Args:

camera\_setup (CameraSetup): An instance of the CameraSetup class.

frame\_queue (Queue): A queue to store captured frames.

queue\_size (int): Maximum number of frames to store in the queue.

"""

self.camera\_setup = camera\_setup

self.frame\_queue = frame\_queue

self.queue\_size = queue\_size

self.logger = logging.getLogger('FrameCapture')

self.logger.setLevel(logging.DEBUG)

ch = logging.StreamHandler()

ch.setLevel(logging.DEBUG)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

ch.setFormatter(formatter)

self.logger.addHandler(ch)

self.thread = threading.Thread(target=self.capture\_frames, daemon=True)

self.stop\_event = threading.Event()

def start(self):

"""Start the frame capturing thread."""

self.logger.info("Starting frame capture thread.")

self.thread.start()

def stop(self):

"""Stop the frame capturing thread."""

self.logger.info("Stopping frame capture thread.")

self.stop\_event.set()

self.thread.join()

def capture\_frames(self):

"""Continuously capture frames from the camera and put them into the queue."""

camera = self.camera\_setup.get\_camera()

stream = cv2.VideoCapture(0)

if not stream.isOpened():

self.logger.error("Failed to open video stream.")

return

while not self.stop\_event.is\_set():

ret, frame = stream.read()

if not ret:

self.logger.warning("Failed to read frame from camera.")

continue

# Resize frame if necessary

frame = cv2.resize(frame, self.camera\_setup.resolution)

# Put frame into queue if not full

try:

self.frame\_queue.put\_nowait(frame)

self.logger.debug("Frame captured and added to queue.")

except:

self.logger.warning("Frame queue is full. Dropping frame.")

# Optional: Add a small delay to reduce CPU usage

time.sleep(0.01)

stream.release()

self.logger.info("Frame capture thread terminated.")

```

\*\*Explanation:\*\*

- \*\*Logging Setup:\*\* Initializes a logger to track frame capturing processes and handle issues.

- \*\*Threading:\*\* Utilizes a separate thread (`self.thread`) to continuously capture frames without blocking the main application.

- \*\*Frame Queue:\*\* Uses a `Queue` to store captured frames. The `queue\_size` parameter limits the number of frames stored to prevent memory overflow.

- \*\*`start` and `stop` Methods:\*\* Control the lifecycle of the frame capturing thread.

- \*\*`capture\_frames` Method:\*\*

- Opens the video stream using OpenCV's `VideoCapture`.

- Continuously reads frames from the camera.

- Resizes frames to match the initialized resolution.

- Attempts to add frames to the queue; if the queue is full, it drops the frame and logs a warning.

- Includes a small delay (`time.sleep(0.01)`) to prevent excessive CPU usage.

- Releases the video stream upon termination.

\*\*Note:\*\* The `VideoCapture(0)` may need to be adjusted based on the camera setup. Alternatively, you can use the `picamera`'s native interface to capture frames directly, which can offer better performance and integration with Raspberry Pi.

---

#### \*\*3. Example Usage\*\*

To demonstrate how the camera module can be integrated and tested independently, below is an example script that utilizes both `camera\_setup.py` and `capture\_frames.py` to capture and display frames in real-time.

\*\*Dependencies:\*\*

- `camera\_setup.py`

- `capture\_frames.py`

- `opencv-python`

\*\*Code:\*\*

```python

# test\_camera\_module.py

import cv2

import time

from camera\_setup import CameraSetup

from capture\_frames import FrameCapture

from queue import Queue, Empty

def main():

# Initialize camera setup

camera = CameraSetup(resolution=(640, 480), framerate=30)

# Create a queue to hold frames

frame\_queue = Queue(maxsize=10)

# Initialize frame capture

frame\_capture = FrameCapture(camera\_setup=camera, frame\_queue=frame\_queue, queue\_size=10)

frame\_capture.start()

try:

while True:

try:

# Retrieve frame from the queue

frame = frame\_queue.get(timeout=1)

# Display the frame

cv2.imshow('Camera Feed', frame)

# Break the loop on 'q' key press

if cv2.waitKey(1) & 0xFF == ord('q'):

break

except Empty:

print("No frame available.")

continue

except KeyboardInterrupt:

print("Interrupted by user.")

finally:

# Stop frame capturing

frame\_capture.stop()

cv2.destroyAllWindows()

if \_\_name\_\_ == "\_\_main\_\_":

main()

```

\*\*Explanation:\*\*

- \*\*Initialization:\*\*

- Creates an instance of `CameraSetup` with desired resolution and frame rate.

- Initializes a `Queue` to hold incoming frames.

- Starts the `FrameCapture` thread to begin capturing frames.

- \*\*Frame Display Loop:\*\*

- Continuously retrieves frames from the queue with a timeout to prevent blocking.

- Displays each frame in a window titled 'Camera Feed'.

- Allows the user to exit the loop and terminate the program by pressing the 'q' key.

- \*\*Graceful Shutdown:\*\*

- Ensures that the frame capturing thread is stopped and OpenCV windows are closed properly, even if the program is interrupted.

\*\*Running the Test:\*\*

1. \*\*Ensure all dependencies are installed:\*\*

```bash

pip install picamera opencv-python

```

2. \*\*Execute the test script:\*\*

```bash

python test\_camera\_module.py

```

You should see a window displaying the live camera feed. Press 'q' to exit.

---

#### \*\*4. Integration with the Perception Module\*\*

To integrate the camera module with the perception module, ensure that the `FrameCapture` class's `frame\_queue` is accessible to the perception scripts. Here's a high-level example of how the main application (`main.py`) might utilize the camera module and pass frames to the perception module:

```python

# main.py

import cv2

import time

from camera\_setup import CameraSetup

from capture\_frames import FrameCapture

from perception.color\_detection import ColorDetection

from queue import Queue, Empty

def main():

# Initialize camera

camera = CameraSetup(resolution=(640, 480), framerate=30)

# Create a queue for frames

frame\_queue = Queue(maxsize=10)

# Initialize frame capture

frame\_capture = FrameCapture(camera\_setup=camera, frame\_queue=frame\_queue, queue\_size=10)

frame\_capture.start()

# Initialize perception module

color\_detector = ColorDetection()

try:

while True:

try:

frame = frame\_queue.get(timeout=1)

# Process frame for color detection

detected\_cones = color\_detector.detect\_cones(frame)

# Implement further decision making and control logic here

# Display frame with detections (optional)

cv2.imshow('Processed Frame', detected\_cones)

if cv2.waitKey(1) & 0xFF == ord('q'):

break

except Empty:

print("No frame available.")

continue

except KeyboardInterrupt:

print("Interrupted by user.")

finally:

frame\_capture.stop()

cv2.destroyAllWindows()

if \_\_name\_\_ == "\_\_main\_\_":

main()

```

\*\*Explanation:\*\*

- \*\*Perception Integration:\*\*

- Assumes a `ColorDetection` class exists within the `perception.color\_detection` module, which processes frames to detect blue and orange cones.

- Processes each frame retrieved from the `frame\_queue` and performs color detection.

- \*\*Further Development:\*\*

- Implement decision-making and control logic based on detected cones.

- This example focuses on integrating the camera and perception modules; additional modules (e.g., control) can be integrated similarly.

---

### \*\*Conclusion\*\*

The Camera Module, comprising `camera\_setup.py` and `capture\_frames.py`, provides a robust foundation for capturing and streaming real-time video data necessary for the self-driving car's perception and decision-making processes. By utilizing multithreading and efficient queue management, the module ensures that frame capturing is performed smoothly without hindering the performance of other system components. Proper logging and error handling facilitate easier debugging and maintenance, contributing to the project's overall reliability and effectiveness.

---

### \*\*Additional Recommendations\*\*

1. \*\*Thread Safety:\*\* Ensure that access to shared resources (e.g., frame queue) is thread-safe to prevent race conditions.

2. \*\*Performance Optimization:\*\* Depending on the processing speed, consider optimizing frame resolution and frame rate to balance between image quality and processing load.

3. \*\*Resource Management:\*\* Properly release camera resources and handle exceptions to prevent resource leaks.

4. \*\*Modular Testing:\*\* Develop unit tests for the camera module to validate its functionality independently before integrating with other modules.

5. \*\*Documentation:\*\* Maintain comprehensive documentation within each script to aid future developers in understanding and maintaining the codebase.

By following these guidelines and utilizing the provided scripts, the Camera Module can be effectively developed and integrated into the larger self-driving car project, enabling accurate and efficient environment sensing.

PRECEPTION

### \*\*Perception Module Implementation for Raspberry Pi 5-Based Self-Driving Car\*\*

The \*\*Perception Module\*\* is responsible for interpreting the visual data captured by the camera to identify and locate blue and orange cones on the track. This module leverages OpenCV for image processing and color detection, enabling the vehicle to understand its environment and make informed navigation decisions.

The Perception Module comprises three main scripts:

1. \*\*`color\_detection.py`\*\*: Detects blue and orange cones in captured frames using color filtering and contour detection.

2. \*\*`image\_processing.py`\*\*: Enhances image quality and prepares frames for efficient processing through filtering and edge detection.

3. \*\*`vision\_utils.py`\*\*: Provides utility functions for drawing detections, debugging, and other vision-related tasks.

Below are the detailed implementations of these scripts, along with explanations to facilitate understanding and integration into the larger project.

---

## \*\*1. `color\_detection.py`\*\*

This script is responsible for detecting blue and orange cones within the provided image frames. It utilizes color space transformations and contour detection to identify and locate the cones.

### \*\*Dependencies:\*\*

- `opencv-python`: For image processing and computer vision tasks.

- `numpy`: For numerical operations.

- `logging`: For logging detection processes and errors.

### \*\*Installation:\*\*

Ensure that OpenCV and NumPy are installed. If not, install them using pip:

```bash

pip install opencv-python numpy

```

### \*\*Code:\*\*

```python

# color\_detection.py

import cv2

import numpy as np

import logging

class ColorDetection:

def \_\_init\_\_(self, config):

"""

Initialize the ColorDetection class with color ranges and logging.

Args:

config (dict): Configuration dictionary containing color ranges.

"""

self.blue\_lower = config['blue\_lower']

self.blue\_upper = config['blue\_upper']

self.orange\_lower = config['orange\_lower']

self.orange\_upper = config['orange\_upper']

# Setup logger

self.logger = logging.getLogger('ColorDetection')

self.logger.setLevel(logging.DEBUG)

ch = logging.StreamHandler()

ch.setLevel(logging.DEBUG)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

ch.setFormatter(formatter)

self.logger.addHandler(ch)

def detect\_cones(self, frame):

"""

Detect blue and orange cones in the given frame.

Args:

frame (numpy.ndarray): The image frame in BGR format.

Returns:

dict: Dictionary containing lists of detected blue and orange cone positions.

"""

self.logger.debug("Starting cone detection.")

processed\_frame = frame.copy()

# Convert BGR to HSV

hsv = cv2.cvtColor(processed\_frame, cv2.COLOR\_BGR2HSV)

# Detect blue cones

blue\_mask = cv2.inRange(hsv, self.blue\_lower, self.blue\_upper)

blue\_contours = self.\_get\_contours(blue\_mask)

blue\_cones = self.\_get\_cone\_centers(blue\_contours)

self.logger.debug(f"Detected {len(blue\_cones)} blue cones.")

# Detect orange cones

orange\_mask = cv2.inRange(hsv, self.orange\_lower, self.orange\_upper)

orange\_contours = self.\_get\_contours(orange\_mask)

orange\_cones = self.\_get\_cone\_centers(orange\_contours)

self.logger.debug(f"Detected {len(orange\_cones)} orange cones.")

return {

'blue\_cones': blue\_cones,

'orange\_cones': orange\_cones

}

def \_get\_contours(self, mask):

"""

Find contours in the given mask.

Args:

mask (numpy.ndarray): Binary mask image.

Returns:

list: List of contours found in the mask.

"""

contours, \_ = cv2.findContours(mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

self.logger.debug(f"Found {len(contours)} contours.")

return contours

def \_get\_cone\_centers(self, contours, min\_area=500):

"""

Calculate the center points of contours that are likely to be cones.

Args:

contours (list): List of contours.

min\_area (int): Minimum area threshold to filter out noise.

Returns:

list: List of (x, y) tuples representing cone centers.

"""

centers = []

for cnt in contours:

area = cv2.contourArea(cnt)

if area > min\_area:

M = cv2.moments(cnt)

if M['m00'] != 0:

cX = int(M['m10'] / M['m00'])

cY = int(M['m01'] / M['m00'])

centers.append((cX, cY))

self.logger.debug(f"Detected cone at ({cX}, {cY}) with area {area}.")

return centers

```

### \*\*Explanation:\*\*

- \*\*Initialization (`\_\_init\_\_`):\*\*

- Accepts a configuration dictionary containing HSV color ranges for blue and orange cones.

- Sets up a logger to track the detection process and handle issues.

- \*\*`detect\_cones` Method:\*\*

- Converts the input frame from BGR to HSV color space for better color segmentation.

- Creates binary masks for blue and orange colors using the specified HSV ranges.

- Finds contours in each mask to identify potential cones.

- Filters contours based on a minimum area to eliminate noise and irrelevant objects.

- Calculates the center coordinates of each valid contour, representing the detected cones.

- \*\*Helper Methods (`\_get\_contours` and `\_get\_cone\_centers`):\*\*

- `\_get\_contours`: Extracts contours from a binary mask.

- `\_get\_cone\_centers`: Filters contours based on area and computes their center points.

---

## \*\*2. `image\_processing.py`\*\*

This script enhances the quality of image frames to facilitate more accurate and efficient cone detection. It applies various image processing techniques such as Gaussian blurring and edge detection.

### \*\*Dependencies:\*\*

- `opencv-python`: For image processing and computer vision tasks.

- `logging`: For logging processing steps and errors.

### \*\*Installation:\*\*

Ensure that OpenCV is installed. If not, install it using pip:

```bash

pip install opencv-python

```

### \*\*Code:\*\*

```python

# image\_processing.py

import cv2

import logging

class ImageProcessing:

def \_\_init\_\_(self, config):

"""

Initialize the ImageProcessing class with processing parameters and logging.

Args:

config (dict): Configuration dictionary containing processing parameters.

"""

self.gaussian\_kernel = config.get('gaussian\_kernel', (5, 5))

self.canny\_threshold1 = config.get('canny\_threshold1', 50)

self.canny\_threshold2 = config.get('canny\_threshold2', 150)

# Setup logger

self.logger = logging.getLogger('ImageProcessing')

self.logger.setLevel(logging.DEBUG)

ch = logging.StreamHandler()

ch.setLevel(logging.DEBUG)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

ch.setFormatter(formatter)

self.logger.addHandler(ch)

def preprocess(self, frame):

"""

Apply preprocessing steps to the input frame.

Args:

frame (numpy.ndarray): The image frame in BGR format.

Returns:

numpy.ndarray: The preprocessed image.

"""

self.logger.debug("Starting image preprocessing.")

processed = frame.copy()

# Apply Gaussian Blur to reduce noise

processed = cv2.GaussianBlur(processed, self.gaussian\_kernel, 0)

self.logger.debug("Applied Gaussian Blur.")

# Convert to grayscale

gray = cv2.cvtColor(processed, cv2.COLOR\_BGR2GRAY)

self.logger.debug("Converted to grayscale.")

# Apply Canny Edge Detection

edges = cv2.Canny(gray, self.canny\_threshold1, self.canny\_threshold2)

self.logger.debug("Applied Canny Edge Detection.")

return edges

```

### \*\*Explanation:\*\*

- \*\*Initialization (`\_\_init\_\_`):\*\*

- Accepts a configuration dictionary containing parameters for Gaussian blurring and Canny edge detection.

- Sets up a logger to track the preprocessing steps and handle issues.

- \*\*`preprocess` Method:\*\*

- \*\*Gaussian Blur:\*\* Reduces image noise and smoothens the image, which helps in more accurate edge detection.

- \*\*Grayscale Conversion:\*\* Simplifies the image by converting it to a single color channel, which is necessary for edge detection.

- \*\*Canny Edge Detection:\*\* Detects edges in the image, highlighting the boundaries of cones for better contour detection in subsequent steps.

---

## \*\*3. `vision\_utils.py`\*\*

This script provides utility functions that assist in visualization and debugging, such as drawing bounding boxes around detected cones and displaying information on the frames.

### \*\*Dependencies:\*\*

- `opencv-python`: For image processing and computer vision tasks.

- `logging`: For logging utility operations.

### \*\*Installation:\*\*

Ensure that OpenCV is installed. If not, install it using pip:

```bash

pip install opencv-python

```

### \*\*Code:\*\*

```python

# vision\_utils.py

import cv2

import logging

class VisionUtils:

def \_\_init\_\_(self):

"""

Initialize the VisionUtils class with logging.

"""

# Setup logger

self.logger = logging.getLogger('VisionUtils')

self.logger.setLevel(logging.DEBUG)

ch = logging.StreamHandler()

ch.setLevel(logging.DEBUG)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

ch.setFormatter(formatter)

self.logger.addHandler(ch)

def draw\_cones(self, frame, cones, color, label):

"""

Draw circles and labels around detected cones.

Args:

frame (numpy.ndarray): The image frame in BGR format.

cones (list): List of (x, y) tuples representing cone centers.

color (tuple): BGR color tuple for drawing.

label (str): Label for the cone type.

Returns:

numpy.ndarray: The image frame with drawings.

"""

for (x, y) in cones:

cv2.circle(frame, (x, y), 10, color, -1)

cv2.putText(frame, label, (x + 15, y + 5), cv2.FONT\_HERSHEY\_SIMPLEX,

0.5, color, 2)

self.logger.debug(f"Drew {label} cone at ({x}, {y}).")

return frame

def display\_fps(self, frame, fps):

"""

Display the current FPS on the frame.

Args:

frame (numpy.ndarray): The image frame in BGR format.

fps (float): Frames per second value.

Returns:

numpy.ndarray: The image frame with FPS displayed.

"""

cv2.putText(frame, f"FPS: {fps:.2f}", (10, 30), cv2.FONT\_HERSHEY\_SIMPLEX,

1, (0, 255, 0), 2)

self.logger.debug(f"Displayed FPS: {fps:.2f}.")

return frame

```

### \*\*Explanation:\*\*

- \*\*Initialization (`\_\_init\_\_`):\*\*

- Sets up a logger to track utility operations and handle issues.

- \*\*`draw\_cones` Method:\*\*

- Iterates over the list of detected cone centers.

- Draws a filled circle at each cone's location.

- Adds a label (`"Blue"` or `"Orange"`) next to each cone for identification.

- Useful for visual debugging and verifying detection accuracy.

- \*\*`display\_fps` Method:\*\*

- Overlays the current Frames Per Second (FPS) value on the frame.

- Helps in monitoring the performance and efficiency of the perception pipeline.

---

## \*\*4. Example Usage and Integration\*\*

To demonstrate how the Perception Module can be integrated and tested independently, below is an example script that utilizes `color\_detection.py`, `image\_processing.py`, and `vision\_utils.py` to process and display detected cones in real-time.

### \*\*Dependencies:\*\*

- `color\_detection.py`

- `image\_processing.py`

- `vision\_utils.py`

- `opencv-python`

- `numpy`

### \*\*Code:\*\*

```python

# test\_perception\_module.py

import cv2

import time

import logging

from color\_detection import ColorDetection

from image\_processing import ImageProcessing

from vision\_utils import VisionUtils

def load\_config():

"""

Load configuration parameters for color detection and image processing.

Returns:

dict: Configuration dictionary.

"""

config = {

'blue\_lower': (100, 150, 50), # HSV lower bound for blue

'blue\_upper': (140, 255, 255), # HSV upper bound for blue

'orange\_lower': (10, 100, 20), # HSV lower bound for orange

'orange\_upper': (25, 255, 255), # HSV upper bound for orange

'gaussian\_kernel': (5, 5),

'canny\_threshold1': 50,

'canny\_threshold2': 150

}

return config

def main():

# Setup logging for the main script

logging.basicConfig(level=logging.DEBUG,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')

logger = logging.getLogger('TestPerceptionModule')

# Load configurations

config = load\_config()

# Initialize modules

color\_detector = ColorDetection(config)

image\_processor = ImageProcessing(config)

vision\_utils = VisionUtils()

# Initialize video capture (replace with FrameCapture from camera module if integrated)

cap = cv2.VideoCapture(0)

if not cap.isOpened():

logger.error("Failed to open video stream.")

return

# For FPS calculation

prev\_time = time.time()

frame\_count = 0

fps = 0

try:

while True:

ret, frame = cap.read()

if not ret:

logger.warning("Failed to read frame from camera.")

break

# Preprocess the frame

processed = image\_processor.preprocess(frame)

# Detect cones

detections = color\_detector.detect\_cones(frame)

blue\_cones = detections['blue\_cones']

orange\_cones = detections['orange\_cones']

# Draw detections

frame = vision\_utils.draw\_cones(frame, blue\_cones, (255, 0, 0), 'Blue')

frame = vision\_utils.draw\_cones(frame, orange\_cones, (0, 165, 255), 'Orange')

# Calculate FPS

frame\_count += 1

current\_time = time.time()

elapsed = current\_time - prev\_time

if elapsed >= 1.0:

fps = frame\_count / elapsed

frame\_count = 0

prev\_time = current\_time

# Display FPS

frame = vision\_utils.display\_fps(frame, fps)

# Show the frame

cv2.imshow('Perception Module - Cone Detection', frame)

# Exit on 'q' key press

if cv2.waitKey(1) & 0xFF == ord('q'):

logger.info("Exiting perception module test.")

break

except KeyboardInterrupt:

logger.info("Interrupted by user.")

finally:

cap.release()

cv2.destroyAllWindows()

if \_\_name\_\_ == "\_\_main\_\_":

main()

```

### \*\*Explanation:\*\*

- \*\*Configuration (`load\_config`):\*\*

- Defines HSV color ranges for blue and orange cones. These values may need to be adjusted based on the lighting conditions and camera characteristics.

- Sets parameters for Gaussian blurring and Canny edge detection.

- \*\*Module Initialization:\*\*

- Instantiates the `ColorDetection`, `ImageProcessing`, and `VisionUtils` classes with the loaded configuration.

- \*\*Video Capture:\*\*

- Opens the default camera stream (`VideoCapture(0)`). Replace this with the `FrameCapture` class from the Camera Module when integrated.

- \*\*Frame Processing Loop:\*\*

- \*\*Preprocessing:\*\* Applies Gaussian blur and edge detection to the captured frame.

- \*\*Cone Detection:\*\* Identifies blue and orange cones in the preprocessed frame.

- \*\*Drawing Detections:\*\* Visualizes the detected cones by drawing circles and labels on the frame.

- \*\*FPS Calculation:\*\* Computes and displays the current FPS to monitor performance.

- \*\*Display:\*\* Shows the processed frame in a window titled 'Perception Module - Cone Detection'.

- \*\*Exit Condition:\*\* Allows the user to exit the loop by pressing the 'q' key.

- \*\*Graceful Shutdown:\*\*

- Ensures that the video stream is released and all OpenCV windows are closed properly, even if the program is interrupted.

### \*\*Running the Test:\*\*

1. \*\*Ensure all dependencies are installed:\*\*

```bash

pip install opencv-python numpy

```

2. \*\*Execute the test script:\*\*

```bash

python test\_perception\_module.py

```

A window should appear displaying the live camera feed with detected blue and orange cones highlighted. Press 'q' to exit the test.

---

## \*\*5. Integration with the Main Application (`main.py`)\*\*

To integrate the Perception Module with the Camera Module and other system components, the `main.py` script orchestrates the flow of data and control signals. Below is an example of how to incorporate the Perception Module into the main application.

### \*\*Code:\*\*

```python

# main.py

import cv2

import time

import logging

from camera\_setup import CameraSetup

from capture\_frames import FrameCapture

from perception.color\_detection import ColorDetection

from perception.image\_processing import ImageProcessing

from perception.vision\_utils import VisionUtils

from queue import Queue, Empty

def load\_config():

"""

Load configuration parameters for color detection and image processing.

Returns:

dict: Configuration dictionary.

"""

config = {

'blue\_lower': (100, 150, 50), # HSV lower bound for blue

'blue\_upper': (140, 255, 255), # HSV upper bound for blue

'orange\_lower': (10, 100, 20), # HSV lower bound for orange

'orange\_upper': (25, 255, 255), # HSV upper bound for orange

'gaussian\_kernel': (5, 5),

'canny\_threshold1': 50,

'canny\_threshold2': 150

}

return config

def main():

# Setup logging for the main script

logging.basicConfig(level=logging.DEBUG,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')

logger = logging.getLogger('Main')

# Load configurations

config = load\_config()

# Initialize camera setup

camera = CameraSetup(resolution=(640, 480), framerate=30)

# Create a queue to hold frames

frame\_queue = Queue(maxsize=10)

# Initialize frame capture

frame\_capture = FrameCapture(camera\_setup=camera, frame\_queue=frame\_queue, queue\_size=10)

frame\_capture.start()

# Initialize perception modules

color\_detector = ColorDetection(config)

image\_processor = ImageProcessing(config)

vision\_utils = VisionUtils()

# For FPS calculation

prev\_time = time.time()

frame\_count = 0

fps = 0

try:

while True:

try:

frame = frame\_queue.get(timeout=1)

# Preprocess the frame

processed = image\_processor.preprocess(frame)

# Detect cones

detections = color\_detector.detect\_cones(frame)

blue\_cones = detections['blue\_cones']

orange\_cones = detections['orange\_cones']

# Draw detections

frame = vision\_utils.draw\_cones(frame, blue\_cones, (255, 0, 0), 'Blue')

frame = vision\_utils.draw\_cones(frame, orange\_cones, (0, 165, 255), 'Orange')

# Calculate FPS

frame\_count += 1

current\_time = time.time()

elapsed = current\_time - prev\_time

if elapsed >= 1.0:

fps = frame\_count / elapsed

frame\_count = 0

prev\_time = current\_time

# Display FPS

frame = vision\_utils.display\_fps(frame, fps)

# Show the frame

cv2.imshow('Main - Cone Detection', frame)

# Implement further decision making and control logic here

# Exit on 'q' key press

if cv2.waitKey(1) & 0xFF == ord('q'):

logger.info("Exiting main application.")

break

except Empty:

logger.warning("No frame available in the queue.")

continue

except KeyboardInterrupt:

logger.info("Interrupted by user.")

finally:

# Stop frame capturing

frame\_capture.stop()

cv2.destroyAllWindows()

if \_\_name\_\_ == "\_\_main\_\_":

main()

```

### \*\*Explanation:\*\*

- \*\*Configuration (`load\_config`):\*\*

- Defines HSV color ranges for blue and orange cones.

- Sets parameters for Gaussian blurring and Canny edge detection.

- \*\*Module Initialization:\*\*

- Instantiates the `CameraSetup` and `FrameCapture` classes to start capturing frames.

- Instantiates the `ColorDetection`, `ImageProcessing`, and `VisionUtils` classes with the loaded configuration.

- \*\*Frame Processing Loop:\*\*

- Retrieves frames from the `frame\_queue`.

- Applies preprocessing and detects cones using the Perception Module.

- Draws detections and displays the FPS on the frame.

- Displays the processed frame in a window titled 'Main - Cone Detection'.

- Allows the user to exit the loop by pressing the 'q' key.

- \*\*Further Development:\*\*

- Placeholder comments indicate where to implement decision-making and control logic based on detected cones.

- \*\*Graceful Shutdown:\*\*

- Ensures that the frame capturing thread is stopped and all OpenCV windows are closed properly, even if the program is interrupted.

### \*\*Integration Considerations:\*\*

1. \*\*Thread Safety:\*\*

- The `FrameCapture` class uses a thread-safe `Queue` to manage frames, ensuring that frame retrieval does not cause race conditions.

2. \*\*Performance Optimization:\*\*

- Adjust the resolution and frame rate in `CameraSetup` based on the processing capabilities of the Raspberry Pi to maintain real-time performance.

3. \*\*Error Handling:\*\*

- Comprehensive logging aids in identifying and troubleshooting issues during integration and testing.

4. \*\*Modular Development:\*\*

- Each module (`camera`, `perception`, `control`, etc.) is developed independently with clear interfaces, facilitating easier integration and maintenance.

---

## \*\*6. Additional Recommendations\*\*

1. \*\*Calibration:\*\*

- \*\*Color Calibration:\*\* Adjust the HSV color ranges (`blue\_lower`, `blue\_upper`, `orange\_lower`, `orange\_upper`) based on the actual colors of the cones and the lighting conditions of the environment.

- \*\*Camera Calibration:\*\* Perform camera calibration to correct for lens distortion and ensure accurate cone localization.

2. \*\*Performance Monitoring:\*\*

- Continuously monitor the FPS and processing times to ensure that the perception pipeline operates efficiently within real-time constraints.

3. \*\*Unit Testing:\*\*

- Develop unit tests for each Perception Module script to validate their functionality independently before integrating them into the main application.

4. \*\*Documentation:\*\*

- Maintain comprehensive documentation within each script and the overall project to aid future developers in understanding and maintaining the codebase.

5. \*\*Scalability:\*\*

- Design the Perception Module to be easily extendable, allowing for the addition of more cone colors or other environmental features in the future.

6. \*\*Integration with Control Module:\*\*

- Ensure seamless communication between the Perception and Control Modules. Consider using shared data structures or messaging queues to pass detected cone positions and confidence levels.

7. \*\*Error Recovery:\*\*

- Implement strategies to handle cases where cones are not detected or when detections are inconsistent, ensuring that the vehicle can recover gracefully from such scenarios.

---

### \*\*Conclusion\*\*

The Perception Module, comprising `color\_detection.py`, `image\_processing.py`, and `vision\_utils.py`, forms the backbone of the self-driving car's ability to interpret its environment. By accurately detecting blue and orange cones, the module enables the vehicle to navigate predefined tracks effectively. The provided scripts are modular, well-documented, and designed for easy integration with other system components, facilitating collaborative development and efficient project progression.

By following the outlined implementations and recommendations, developers can ensure that the Perception Module operates reliably and efficiently, contributing significantly to the overall success of the self-driving car project.

MACHINE LEARNING MODULE

**Machine Learning Module**

**Objective:**

The purpose of this module is to build a Machine Learning (ML) model to classify human activities based on sensor data from an embodied robot system. This module will implement a basic supervised learning algorithm to classify different actions, such as walking, running, sitting, etc., from the robot's sensory inputs.

**1. Data Collection**

Before implementing the machine learning model, you need a dataset that represents different human activities. For this module, you can either use a publicly available dataset (e.g., the [UCI HAR dataset](https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones)) or create a synthetic dataset if sensor simulation is implemented.

**Instructions**:

* Collect time-series data from the sensors integrated into the robot (accelerometer, gyroscope, etc.).
* Ensure that each sample is labeled according to the activity it represents (e.g., "walking," "running," "sitting").
* Preprocess the data by normalizing the sensor values to ensure consistency across features.

**2. Preprocessing**

In order to feed the data to the ML model, the raw sensor data needs to be processed.

**Steps for Preprocessing**:

* **Normalization**: Scale the sensor data to ensure that each feature has a mean of 0 and a standard deviation of 1.
* **Segmentation**: Break the time-series data into windows (e.g., 3-second windows) so that each window can be classified as a specific activity.
* **Feature Extraction**: If needed, extract statistical features like mean, standard deviation, and frequency-domain features from the sensor readings for each window.

python