

Novel Assistive Systems Applying Machine Vision and Human-Machine Interaction Technologies for Vulnerable Populations

Y3890959

*School of Physics, Engineering and Technology
University of York
York, England*

Abstract—This article investigates machine vision and human-machine interaction technologies to present two novel systems designed to assist vulnerable populations, encompassing older individuals, people with disabilities, and workers in treacherous conditions. A machine vision-based driver assistance system involving lane, driver's eye-gaze and blink tracking introduces an aid for workers in dangerous environments. Embodying human-machine interaction aspects into this system forms a novel design for cost-effective 'railroad-style' autonomous transportation for older or disabled individuals. Discussions include technical analysis, safety and ethics considerations, and the potential impact on user livelihoods.

Index Terms—machine vision, human-machine interaction, lane detection, driver assistance, gaze detection, autonomous transport

I. MACHINE VISION-BASED DRIVER ASSISTANCE SYSTEM

To gauge the driver's focus, a simple driver assistance system must track the vehicle's lane position and the driver's eye gaze and blinks. A vehicle's deviation from the centre of the lane, paired with repetitive blinks or an off-centred gaze in the driver's eyes, denotes extended fatigue or distraction of the driver. With the implementation of an alert system, which alerts not only the driver but other road users, this system can help workers in dangerous conditions, such as truck drivers traversing roads with narrow lanes and combatting fatigue. Abstracting into three fundamental mechanisms can drastically help with the development of this assistance system:

- lane detection and tracking: detection of the lanes and calculating the distances from each side to track the average position of the vehicle.
- gaze detection and tracking: detecting and tracking the pupils of the driver's eyes to calculate the average time spent looking off-centre.
- blink detection and tracking: detection and tracking of blinking eyelids to calculate the average blinks or the average time of closed eyelids over some time.

A. Lane Detection and Tracking

The lane departure warning system is quite a mature product, first unveiled in 1989 as a concept and working prototype fitted to the Rover SD1, but later debuted in 2000 as a production-ready system for the Mercedes Actros line of commercial trucks [1]. Given the safety benefits, lane departure systems have become a mandatory feature for all cars in the European Union as of 2022 [1] [2]. The most popular method for detecting lanes across the industry is to employ the Canny edge detection algorithm, The Hough Transform, and various principles of camera calibration, perspective transforms, and colour filtering to determine the lane line edges from a live feed [1]. Three stages contribute to the lane detection system [3]:

- pre-processing: low-level image processing consisting of an image crop to reduce the region of interest for more efficient

computing, grayscale conversion in preparation for Canny edge detection, and noise reduction techniques such as erosion, dilation and image soothing for more accurate detection.

- post-processing: implementation of Canny edge detection resides in this stage using the pre-processed frames to detect the discontinuous lane markings, which The Hough Transform connects into a continuous line.
- road lane modelling: computation to detect the left and right lane markings will utilise the continuous line output from the post-processing stage, establishing the positions of the lane boundaries to calculate the vehicle positioning within the lane to detect deviation.

The Hough Transform provides a highly efficient method for detecting straight lines in images, even with noise and missing data [4]. The Hough Transform works by converting the image space into a parameter space called The Hough Space, and through this space, it is possible to detect geometric shapes by identifying patterns [5]. This ability of The Hough Transform to identify shapes makes it an ideal tool for detecting lane lines for a self-driving car [6].

1) *Prototype for Lane Detection and Tracking:* Extracting individual image frames from a continuous video stream simplifies the logic of this system. The work in [3] forms a robust starting point for achieving a lane detection system. As the paper states, there is a cropping step in the pre-processing stage to reduce computational overheads by removing unnecessary parts of the input image. A single-channel conversion to grayscale will reduce image data complexity to enable faster Canny edge detection. Although the mention of a noise reduction step in [3], a detailed explanation nor a visual description is present, leading me to believe its exclusion in the proposed system. As the work in [7] discusses improvements in the Canny edge detection method by pre-processing the image by stretching the image histogram, the inclusion of a histogram equalisation step will occur post-grayscale in this system. Inspired by the work in [6], applying a Gaussian Blur filter ensures the disregard of as many false edges as possible by reducing noise and smoothening the image.

The work in [3] mentions, "Canny edge detection is implemented in the post-processing stage." But confusingly, later in the paper, a diagrammatical outline shows this step within the 'pre-processing' stage. The Canny edge detection step will occur in the post-processing stage to avoid confusion and enhance stage separation. Despite applying an image crop, the Canny edge detection will still provide results outside the region of interest (ROI). Inspired by the work in [6], cropping the Canny output to the triangular ROI will reduce additional computational overheads and improve The Hough Transform step. Finally, in the post-processing stage, The Hough Transform will generate continuous lines representing the lane marking detections,

providing a foundation for lane tracking.

Overlaying The Hough Transform results over the image will provide a visualisation of the machine vision-based road lane marking detection helpful in gauging the system performance. Harnessing the fixed perspective characteristic of the system makes it possible to track vehicle lane positioning by measuring the x-axis location of the vertical base of the left/right lane lines. Veering towards the left lane can be denoted by an increase in the x-axis position of the left-hand lane line base, and veering towards the right can be denoted by a decrease in the x-axis position of the right-hand lane line base. Vehicular deviation from a lane is denoted by whether the base position of each lane line exceeds/falls from a fixed threshold x-axis position value. Threshold optimisation must happen before system implementation by accounting for the width of the vehicle and the perspective/positioning of the camera. Tracking the mid-point x-axis position of the automobile over time, normalising the value w.r.t. the lane positions, and then calculating the variance of this data can measure the driver's dispersion within the lane. This result forms the basis of the metric to gauge the driver's fatigue and focus levels, with a higher variance denoting high fatigue or low focus.

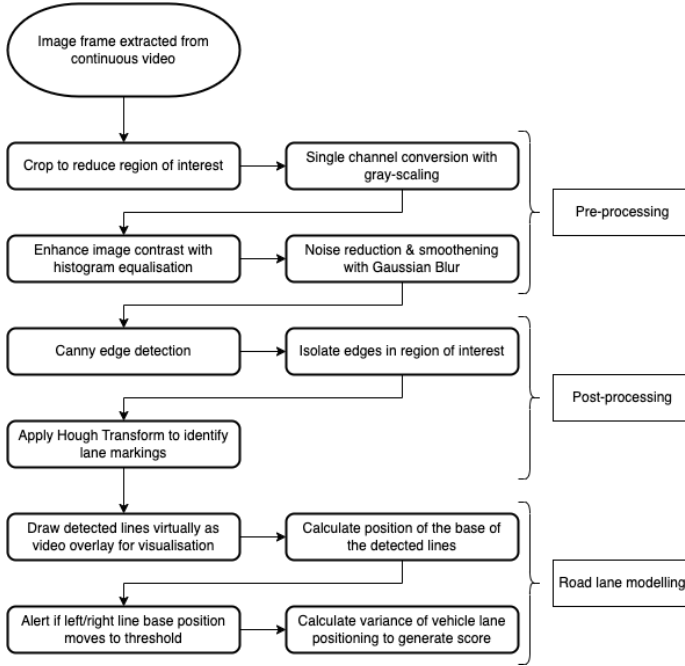


Fig. 1. Flow of lane detection and tracking logic.

The developed software, source code in Appendix A, utilises the Python (3.11.2) programming language and employs the OpenCV library to access highly optimised and efficient computer vision algorithms, reducing development overheads tremendously. The software incorporates a cropping stage before processing the frame to ensure compatibility with a wide range of input video dimensions. A constant definition for the height and width of the input video feed is present, which the initial crop utilises for generating a window aligned at the bottom of the original frame and horizontally centred, ensuring a 640(w)x480(h) crop (dimensions as defined by the constants) for any higher resolution video feed. This cropping is merely for input video dimension standardisation, and regions outside of the ROI will still be present to ensure easy data visualisation. A YouTube video [8] forms the perfect testing input for this system, consisting of an

almost 4-hour video feed of a nighttime drive from Busan to Seoul through various types of roads and lighting conditions.

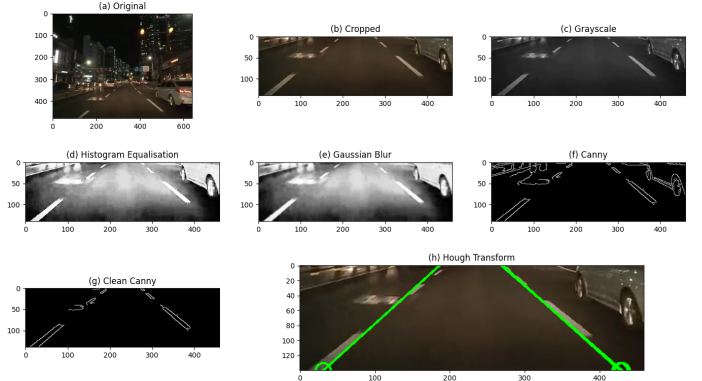


Fig. 2. Stage-by-stage plots of the lane detection and tracking system outputs. Original frame shown in (a) which is then cropped in (b), grayscale filter applied in (c), with histogram equalisation in (d) and a Gaussian blur in (e), output from the Canny edge detector in (f), with results cropped and shown in (g), the road lane markings identified by The Hough Transform are overlaid on to the frame in (h).

After the input video feed standardisation to 640x480, the first step in the pre-processing stage is an additional frame crop to reduce the ROI. Using the matplotlib Python library, a plot of the normalised video with the axis labels can help choose the lane width with the lane markings and how far forward the system needs to see. With the ROI region coded as shown in Figure 2, grayscale occurs to reduce computational complexity by converting to a single-channel image. The application of histogram equalisation, then a Gaussian blue filter, prepares the frame for the post-processing stage of the system.

Implementing a zero-parameter automatic Canny edge detection logic simplifies the system greatly [9]. Though not truly a 'zero-parameter' due to the necessity of a threshold 'sigma' parameter, it vastly simplifies logic using statistical offsets, enabling single parameter tweaking for optimisation. A further crop with an isosceles trapezoid-shaped masking window on the Canny algorithm output eliminates all edge information outside the lane markings, further isolating the ROI. The clean Canny output enables a more efficiently performed Hough Transform to detect the road lane markings. The Hough Transform implementation has further logic to ensure only lines within a specified diagonal angle range are output from the step to ensure only lane markings are detected. The definition of programmational constants for the angle range values enables simple optimisation. The Hough Transform generated lines drastically extend the frame size due to the presence of a multiplier as per the OpenCV tutorial documentation [10]. Modifications to The Hough Transform logic in this lane assist implementation include scaling the detected and drawn lines to fit within the border of the image frame. Programmational detection of road lane markings consists of parsing the start and end coordinates of the detected lines. The Hough Transform generates two coordinate points for each detected line, with a starting point at the bottom of a positive gradient line and at the top of a negative gradient line. More simply, starting points will be on the left and ending points on the right.

Left-hand lane markings will start at a y-axis value equal to the frame height and finish at a y-axis value equal to 0, with arbitrary x-axis values. Right-hand lane markings begin at the top and end at the bottom of the frame, with arbitrary x-axis values. This understanding makes it easier to differentiate between the two sides of lane markings

and to parse the position of the base of each line. The departure warning implementation will use the base positions of each lane marking, alerting once each position passes an x-axis threshold (programmatically defined as a constant). Shown in Figure 2 is the implementation in action with lane detections shown in green and the bases of each also circled in green. When the driver approaches too closely to either lane, the lines will turn red, alerting a lane departure. The vehicle lane tracking implementation collects the base location of left-hand and right-hand lane markings, using linear interpolation for intermediate predictions for frames with undetected lane markings. The centre point between the two lane markings is the vehicle's position. Calculating the driver's lane position variance (LPV) metric, showing the variance in the vehicle's position w.r.t. the lane markings, is possible using an array of vehicle positions. Including garbage collection to limit the maximum and minimum number of collected vehicle position data by deleting old data once passing the maximum threshold prevents a gradual slow-down of the program due to a massively growing array.

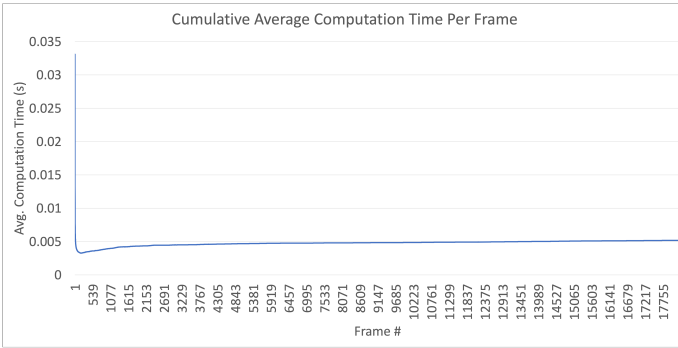


Fig. 3. Cumulative average of the time taken to compute each frame for lane detection and tracking.

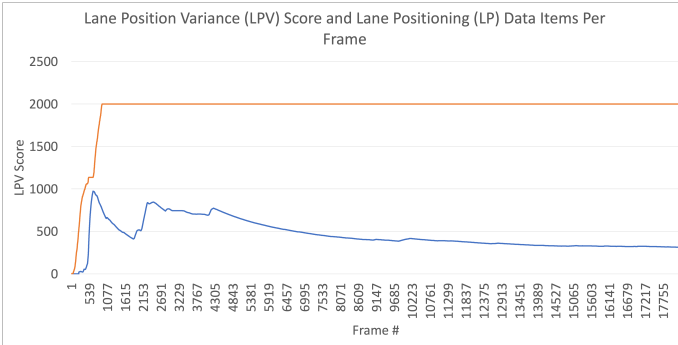


Fig. 4. (Blue trace) cumulative average of the LPV score at each frame for lane detection and tracking, calculated using the footage between 0:18-13:00 from the testing video [8]. (Orange trace) number of lane positioning data items used to compute LPV score.

The mission-critical essence of this system permeates from its fundamental requirement of detecting and alerting hazards as they occur. The processing time for each frame must be as low as possible to ensure the software can keep up with the input video feed. The system implements start and finish timestamping to acquire metrics for real-time performance. The YouTube video as before [8] but only including the footage between 0:18 and 13:00 forms the basis of these real-time metrics plots in Figure 3 and Figure 4. Shown in Figure 3 is the cumulative average for the time taken to compute each frame.

We can see an average frame compute time of $5.2\mu s$, resulting in an average of roughly 192 frames per second. It is unclear what caused the tremendous spike in compute time during the first frame, but I assume this to be OpenCV-related, resulting from populating GUI windows for the first time.

Shown in Figure 4 is a plot of the lane position variance, a score calculated from the cumulative average of the variance in the vehicle's position w.r.t. the lane markings from each frame. The spikes in this score at the beginning of the footage are due to the driver making multiple lane changes and swaying within the lane. However, once the driver enters the motorway, the score starts to settle down, a result of fewer to no lane changes and swaying. The graph also has an overlaid plot of the number of lane positioning data items, a count of the total number of left-hand and right-hand lane base positions. To ensure as much interpolation accuracy as possible, the calculation of the LPV score only starts when there is a minimum of 500 previously recorded lane marking positions. The higher the amount of lane position data points, the slower the linear interpolation algorithm will be. The system's effort to maintain a maximum of 2000 points to ensure performance is visible as the graph plateaus at this value.

B. Blink and Gaze Detection and Tracking

Although separating blink and gaze aspects into individual blocks helps with conceptualising the system, in implementation, it is much easier and more efficient to merge them into a single block. The first driver monitoring implementation was introduced by Toyota in 2006 for its and Lexus' latest models, incorporating face tracking using IR LED detectors and CCD cameras [11]. Although 18 years have passed since then, only a handful of car manufacturers have adopted this technology, including BMW, Ford, Cadillac, NIO, XPeng, and Mercedes-Benz, incorporating this technology in some of their models. Albeit regulated, this technology is not yet mandatory as per the European Union [12], which could be the primary cause for this lacklustre adoption. As a result, there is no industry-standard methodology for this system. This system follows the same three-stage abstraction as the lane detection and tracking flow. The pre-processing stage is identical. The post-processing now includes Dlib machine learning-based functions [13] to detect faces and predict facial landmark coordinates, utilising Dlib's pre-trained 68-point facial model [14]. Finally, a binary inversion threshold filter before a Simple Blob Detection step facilitates pupil/iris detection. The final stage consists of driver monitoring logic, where results computation from the pre and post-processing stages occur to calculate the driver's fatigue and focus levels.

1) *Prototype for Gaze and Blink Detection and Tracking:* Following a similar approach to [15], the programmatical implementation for this system, source code in Appendix B, uses a similar logic from the lane detection and tracking implementation. A crop is present to reduce the resolution of the feed to 640×480 , programmatically defined as a constant, to speed up frame processing time. Grayscale, histogram equalising and Gaussian filtering steps are present to improve system efficiency for detection and processing time, in addition to drastically enhancing the low light capabilities of this system. The detection of faces and facial landmarks is possible using the Dlib detection function with the trained 68-point landmark model. The eye aspect ratio (EAR) enables the detection of blinks by deriving from the positions of detected eye landmarks using the dimensions of the left and right eyes. The system calculates a cumulative average EAR over a fixed amount of time to form the driver tiredness score, currently set to a 600 frame sample size, which equates to 10 minutes

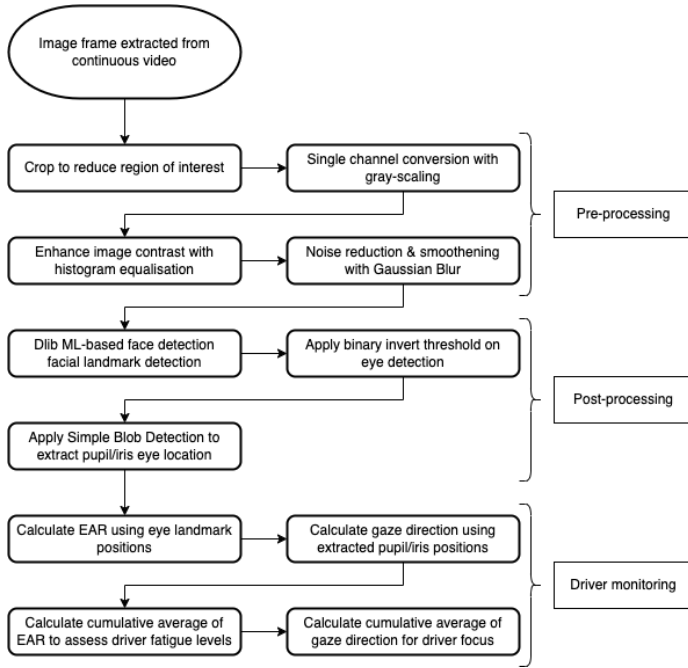


Fig. 5. Flow of eye blink and gaze detection and tracking logic.

for a 30 FPS input. The implementation alerts if the tiredness score drops below a certain threshold, notifying the driver that most of the 10-minute sample size consists of shut or near-shut eyelids.

Isolating each eye as a window is possible since landmarking accurately identifies each eye's coordinate bounds. With each isolated eye, applying a binary threshold allows for a clear separation between the pupils/iris and the sclera, forming an ideal preparation for the OpenCV Simple Blob Detection algorithm [16]. The initialisation of the Simple Blob Detector consists of deactivating the thresholding stage since the implementation already grayscales the frame. Filtering by colour is enabled, correctly setting the thresholded colour of the pupils/iris. Filtering by area is enabled since testing revealed that the dark spots of the pupils/iris take up most of the space in the frame.



Fig. 6. Visualisation of pupil/iris detection using the Simple Blob Detector for the gaze tracking system.

Figure 6 shows the result of the blob detection, detecting the iris successfully. Though quite reliable at tracking left and right gaze directions, a paramount downside is that it does not track up and down movements competently, as the blob detection fails to detect. Tracking all gaze locations is crucial for a driver monitoring system. Workarounds are present, such as measuring the time the driver spends looking straight, left, or right since tracking this is comparatively reliable, then inverting this to generate the time the driver spends not looking ahead, allowing a gauge on the driver's focus. Despite lacking gaze tracking functionality, Figure 7 shows an

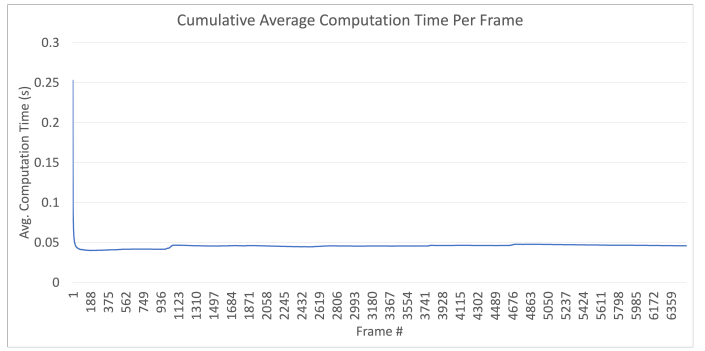


Fig. 7. Cumulative average of the time taken to compute each frame for eye detection and tracking.

average $45\mu s$ frame compute time, with the same spike in the first frame, just like the lane detection and tracking system. The average frame compute time value exhibits an approximate 25 frames per second compute rate for this system, only a five frames per second difference from my webcam capture rate.



Fig. 8. Eye gaze and blink tracking demonstration of a different system [18].

Unlike the lane detection and tracking system, a highly efficient frame computing capability is not a necessity for this system. Due to its reduced mission criticality, the fundamental goal would instead be to focus on highly accurate gaze tracking. The work in [17] implements a Regression Convolutional Neural Network on top of the facial landmark detection to track eye gaze. Initialising and running the Python code associated with this paper, located in a GitHub repository [18], reveals a veraciously impeccable gaze detection implementation achieved by accelerating traditional machine-vision methodologies using machine-learning techniques, shown in Figure 8. The improved tracking accuracy comes at a price of performance, but this substantial drop in frame rates is a sacrifice to gain reliable measurements.

C. Implementation of System

Interlinking the lane detection and tracking system with the blink and gaze detection and tracking implementation assembles the com-

plete driver assistance solution. Computing an overall score using the LPV and tiredness/focus score from the two individual systems is possible and can gauge the driving safety score for the driver. The prospect of employing several interaction modalities can improve the safety of the driver and surrounding traffic:

- a. Visual feedback: A visualisation of the detected lane markings coloured to portray lane departure can be overlayed on a video feed, perhaps augmented via a heads-up display, providing assistance to maintain centred lane positioning. Alerts to show in the instrument cluster or heads-up display when the driver struggles to maintain centre position (LPV score above threshold), when the driver's blinking is more frequent (tiredness score via blink tracking above threshold), or when the driver's focus is low (focus score via gaze tracking below threshold). External car lighting can alert the driver and surrounding traffic by implementing a similar design as Audi [19] by integrating results from this driver assistance system. Matrix LED headlight technology draws symbols on the road to emphasise lane markings, with warning triangles and concentrated light flashes to warn the driver and surrounding traffic and organic LED taillights to provide visual information to surrounding traffic.
- b. Auditory feedback: Employing spacial audio cues for lane departure warnings, developed in a way where the audio direction comes from the side of the lane departure, forms the ultimate warning system. It's not necessarily noise that keeps you awake - it's the sudden change in noise that's most likely to wake you up [20]. Harnessing this ideology can administer a prime solution to combat driver fatigue; playing a loud sporadic noise, such as a dog barking at random intervals when the tiredness score surpasses a certain threshold, can help wake the driver. A more ordinary solution is to display an alert on the instrument cluster or a heads-up display to suggest the driver take a coffee break.
- c. Haptic feedback: Vibrations or physical movements of the steering wheel can alert the driver of lane departure. Since the system knows the lane positioning, it can take a further step by applying steering inputs to ensure the vehicle maintains a centred position within the lane. Implementation of seat vibrations can enhance the alert system, especially in the case where the driver is asleep or unconscious with both hands off the steering wheel. A more aggressive seat jolt, implemented in the same fashion as the load sporadic dog barking, can wake the driver more efficiently.
- d. Climatory feedback: Upon driver fatigue detection, the driver assistance system can autonomously set the car's climate to an unbearably cold cabin temperature, catalysing an unpleasant thermal environment to wake the driver. Some car manufacturers, such as BMW [21], have implemented climate-control-based cabin scent diffusers, crafting a perfect method to wake the driver when fatigued by filling the cabin with citrus, coffee, or similar aromatherapeutic scents to stimulate the brain [22].

D. Safety, Ethics, and Impact of System

Statistics show that a lane assist system results in an 11% reduction in crashes of all severities and a 21% reduction in crashes with injuries, with results suggesting the saving of thousands of lives each year if every passenger vehicle in the US implements a lane departure warning system [23]. Studies show that human error, including fatigue or drowsiness, contributes to 80.6% of road accidents [24], with reports stating that 54% of adult drivers feel drowsy while driving and 28% attesting that they fall asleep while driving [25], the need for a reliable driver fatigue detection and monitoring system, which alerts the driver has never been more imperative as our reliance on

vehicles has exponentially increased, leading to higher levels of road users.

Despite the high praise for driver assistance technologies, a voluminous risk still exists involving the shift of trust to machine vision and computer algorithms to perform safety-critical decisions. For example, false lane detection alerts can potentially startle the driver and lead them to perform unsafe manoeuvres to mitigate a risk that was never present. Moreover, the driver monitoring system responsible for tracking driver focus and fatigue, when implemented into driving trackers (black-box trackers) for car insurance policies, can result in adverse insurance premium changes due to false results. Strategies to improve system performance for enhanced environment tracking are present, though a computer vision approach can never be perfect due to varying environmental conditions. A key factor to mitigate the risk is considering these systems as a helping hand and not holding predominant trust in them. Another aspect to consider is privacy and ethicality, especially with the constant recording of surrounding roads and the driver's face to ensure system functionality. Employment of edge computing, ensuring all data processing occurs within the vehicle, can mitigate privacy risks by preventing data transfer to servers. However, this can drastically bottleneck the system performance as power-limited and smaller form factor computers inside vehicles replace the powerful and centralised server farms.

70% of survey respondents believe that in-vehicle Driver Monitoring Systems can improve road safety and help reduce accidents caused by distracted or fatigued drivers, with only 28% of UK consumers knowing this new technology [26]. Carmakers and regulators must strive for widespread adoption, to raise awareness, and to improve road safety for all.

II. HUMAN-MACHINE INTERACTION-BASED AUTONOMOUS TRANSPORTATION SYSTEM.

In the amalgam of daily activities, there exists a spectrum of challenges, often unseen by the vast majority, that inhibit the lives of older individuals and those living with disabilities. From elementary tasks of dressing, bathing, feeding, and moving between spaces to the broader aspect of travelling independently, all pose hurdles, wielding a power to erode independence, strip away dignity, and confine individuals. In light of these challenges, integrating machine vision and human-machine interaction technologies emerges as viable solutions. Acting as a beacon of hope, leveraging these innovative technologies brings the potential to support vulnerable populations in their day-to-day lives. By extending the scope of machine vision applications, these systems can offer a transformative shift to ease the burdens and hurdles encountered in their routines.

The importance of such assistive systems embodies a promise to transform lives, drastically increasing the quality of life for many. Beyond the improvement of convenience, these systems represent a gateway to boost independence, bringing the joy and fulfilment of engaging in more activities. This second part of the article explores an autonomous transportation system, extending from the machine vision technologies explored previously and the fusion with machine vision methodologies, conceptualising an advanced assistive system seeking to redefine mobility for vulnerable populations. This novel approach aims to materialise as a mobility aid that transcends barriers in transportation, offering seamless navigation within large buildings such as supermarkets and hospitals.

A. Design Analysis

Large buildings, bustling with activity and often maze-like in structure and layout, pose substantial challenges for the vulnerable

population traversing these spaces. Supermarkets, hospitals, and similar expansive structures designed to accommodate multitudes of people and diverse functionality have become labyrinthine for individuals with mobility limitations. The two main challenges are the following:

- a. Navigational: The sheer expanse of these buildings can pose a significant hurdle, with lengthy corridors, multiple floors, and complex layouts, often extremely confusing to all, not just the vulnerable populations. The struggle to find specific sections within these structures, aisles in a supermarket or wards in a hospital, for example, can lead to immense frustration and exhaustion.
- b. Physical: Narrow passages, crowded areas, and high shelves in supermarkets can impede smooth movement. In hospitals, intricate floorplans, diverse elevations, and obstacles generated by unexpected medical equipment placement can add further complexity, hindering comfortable navigation and travel for vulnerable populations.

The characterisation of challenges aids the conception of the system requirements for this autonomous transportation system. For the development of a system that provides effective aid to the vulnerable population in navigating these expansive and challenging environments, several critical considerations and requirements emerge:

- a. Intuitiveness: Simplicity and straightforward guidance are paramount for vulnerable populations. The system must be intuitive, offering clear and accessible directions, incorporating visual and auditory cues aligned with clarity, and minimising confusion and cognitive overload. Intuitive UI, with readable fonts, icons, and a pleasant colour scheme, is critical as vulnerable populations may not have experience with complex technological systems.
- b. Adaptability: The system should possess the ability to adapt, catering to the diverse layouts and structures of different buildings. Whether in a hospital, supermarket, or any other expansive structure, the system must seamlessly and precisely accommodate various floor plans and spatial intricacies, such as crowds, hazards, and obstacles.
- c. Real-time: In vastly changing environmental conditions, accurate real-time updates are crucial. The system must factor the efficiency of frame processing times to ensure system actions are instant without a large delta to the environment. Similar to the lane detection and tracking system, it's imperative to action dangers such as path deviation of the transportation system or the detection of newly spawned hazards as soon as possible.
- d. Accessibility: Incorporating accessibility features is essential when developing a system that wholly encompasses the vulnerable population. The system should accommodate users with different mobility and sensory abilities, implementing functionality such as voice-guided navigation, tactile interfaces, or eye tracking to ensure inclusivity.
- e. Safety: Similar to real-time requirements, maintaining concise requirements for safety is vital. There must be multiple override levels, one that the user of this technology can utilise and another that a secondary person, such as a caregiver or shop floor assistant, can utilise during emergencies. The system must provide live and accurate metrics, such as location and the route to the destination, to both the user and secondary personnel. Monitoring this information closely and constantly, either by the user, a shop-floor assistant, or the caregiver, will ensure the

mitigation of emergencies before they emerge.

- f. Modularity: This system must have a small footprint and be easily interchangeable with vehicles, such as various mobility scooter brands. There must also be easy integration, such as simple system-level steering, acceleration, and braking controls. Implementation of user overrides for critical controls ensures the preservation of safety.

B. System Architecture

The envisioned autonomous transportation system integrates various technologies to facilitate navigation, obstacle avoidance, and user interaction. By leveraging machine vision and sensor fusion with human-machine interaction, this system amalgamates lane detection and eye-tracking methodologies into a unified and adaptable architecture.

1) *Lane Detection Transformation*: Affixing tape to the floor contributes to a cost-effective 'railroad-style' autonomous transportation method. The tape can have prints of different shapes of different colours, where the differentiation of orientation is possible with a simple isosceles triangle-based arrow inside said shape. Different shape and colour combinations can denote each intended destination, with the orientation of the arrow denoting the direction of travel.

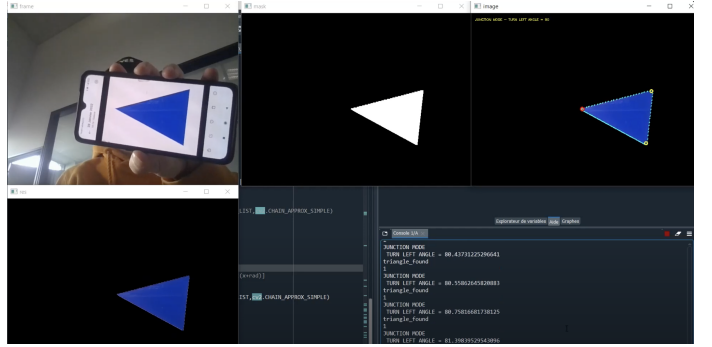


Fig. 9. Demonstration of the arrow direction tracking solution [29] [30].



Fig. 10. Diagram of autonomous transportation system architecture.

The lane detection and tracking system, designed for the driver assistance system's vehicular lane monitoring functionality, will undergo adaptation to facilitate a line-following mechanism for the mobility scooter. This adaptation involves repurposing the lane detection algorithm to recognise and track specific tape shapes and colours affixed to the ground, interpreting these markings as navigational cues. At the beginning of a journey, the user can select a destination, thus setting the colour-masking threshold parameters

to forward the correct shape colour. A camera positioned in a way pointing to the floor can detect the correct path of shapes since the colour-masking will ensure only the correct shape colour is passed into the frame, with surrounding pixels staying black. OpenCV simplifies the implementation of shape detection, as the work in [27] outlines an efficient method to achieve this using the Ramer-Douglas-Peucker contour approximation algorithm [28]. Derivation of the shape direction is possible by cropping to the region within the shape, isolating the isosceles triangle to detect the triangle's vertices outputs the direction, similar to the work in [29] [30], demonstration in Figure 9. A forward-facing camera with light detection and ranging (LiDAR) sensors enables the implementation of the simultaneous localisation and mapping (SLAM) method to detect obstacles, generate a map of the surroundings, and navigate to avoid obstructions when required to deviate from the tape-implemented path [31] [32]. Figure 10 shows the overall architecture involving this path detection and tracking system.

2) *Eye Blink and Gaze Control*: The earlier implemented eye blink and gaze detection system is pivotal in providing this system with user control and override functionalities. Gaze direction can potentially act as a means to alter the vehicle's direction or stop its movement. For example, a prolonged gaze or blinks in predefined patterns can signal a direction change or a halt.

3) *Retro-Fitting to Existing Vehicular System*: To achieve the required system modularity, the implementation of removable sensors and controls utilising magnetic components to attach to the metallic body of a vehicle, such as a mobility scooter. Creating a universal control interface ensures compatibility with different scooter models. A simplified and adaptable control interface would encompass steering, acceleration, and braking controls, allowing seamless integration with different vehicular systems. Servo implementation for controlling the vehicle's steering is perhaps the most difficult due to the size constraints and technical aspects of installing it onto a mobility scooter. A touch-screen with embedded speakers will be in front of the user for easy control of the system. There's a possibility to implement gaze and blink tracking to control the screen, thus controlling the system. Implementing user overrides for critical controls ensures safety, including accessible emergency stop buttons or switches that can instantly halt the scooter's autonomous operation, placing control back in the user's hands. Despite the universal control interface, preserving the original control surfaces, such as the accelerator, brake pedals and steering wheel, will ensure the means of a safe and natural user override.

4) *Interaction Modalities*: Discussing the interaction modalities provides a detailed summary of the whole system. Employing several interaction modalities can improve the safety of the user and the surrounding environment:

- a. Tape-based navigational cues: Employ tape markings of distinct shapes and colours on the floor to indicate destinations. Different shape-color combinations depict various destinations; the orientation of arrow shapes shows the direction of travel. The user selects the destination, and the system follows the correct tape path using colour masking and shape recognition.
- b. Obstacle detection and navigation: The camera and LiDAR array with the SLAM method enable obstacle detection and mapping. The sensor array detects the surroundings to assemble a map and navigate around obstructions. With this method, the deviation from the tape-implemented path to avoid obstacles is possible.
- c. Blink and gaze control: Eye tracking can implement user control and override functionalities. The user's gaze or blinking pattern

can signal direction changes or halts, allowing the user to influence the direction or overall control of the system.

- d. Physical controls: Implementation of a touch screen panel and preservation of the original vehicular control interfaces provide physical touch controls to the user. The user can interact with these surfaces to influence the direction and overall control of the system, including safety overrides.

C. Safety, Ethics, and Impact of System

Incorporating safety features for accident prevention should be of utmost priority for any autonomous transportation system. Implementing a range of safety features and protocols is a must to mitigate potential risks and ensure accident prevention. All the discussion points on this aspect of the driver monitoring system apply here.

A constant and live metrics feed reporting the system's performance, location and route must be available to caregivers and shop floor assistants. This vigilance ensures proactive management and immediate intervention in case of deviations or unforeseen hazards to minimise the risk of accidents. There must be a kill switch to terminate the system at the press of a button, both in the vehicle accessible by the user and remotely accessible by the caregiver or shop floor assistant.

The system must emphasise intuitive user interfaces, employing readable fonts, clear icons and a friendly colour scheme to ensure ease of use. The system must focus on user-empowering design choices rather than those that intrude upon their independence, striking a balance between autonomy and assistance. It operates as a reliable aid, allowing users to navigate expansive structures with confidence and minimal external intervention, fostering a sense of self-reliance and dignity.

III. CONCLUSION

The convergence of machine vision and human-machine interaction yields a new era of support systems aiming to improve the lives of diverse populations facing various challenges. Mentions of driver assistance and autonomous transportation systems in this article represent significant strides in this domain, each catering to distinct yet intersecting needs.

The driver assistance system is a beacon of safety and assistance, offering invaluable aid to drivers in harsh conditions traversing complex roadways. Leveraging machine vision technologies, the system pioneers lane detection and tracking, promising heightened safety by tracking lane departures, driver fatigue and focus levels, generating quantitative and definitive scores to gauge both metrics. To enhance its efficacy further, continuous refinements in the lane detection algorithms and integration with more advanced sensor technologies like LiDAR could bolster its ability to detect and respond to dynamic environments, integrating the detection and tracking of other road users. Furthermore, positioning cameras in better locations, such as below the wing mirrors for lane markings capture, improves the system's immunity to varying environmental conditions.

On the other hand, the autonomous transportation system emerges as a revolutionary mobility aid, particularly beneficial to the vulnerable population when traversing intricate buildings. This system's marriage of machine vision and human-machine interaction leads to a refined architecture enabling tape-based navigational cues, obstacle detection, and user interaction through gaze and blink control. To augment its capabilities, ongoing improvements in obstacle detection algorithms, including machine learning methodologies, and the seamless integration of more robust safety overrides could fortify its reliability. Further advancements to emphasising modularity can

amplify their universality and applicability across many vehicles and user scenarios.

The potential impact of these systems exceeds mere convenience, bringing to light a transformative shift in the lives of vulnerable populations by augmenting safety and fostering independence with assistance. Workers subjected to challenging environments may find relief through driver assistance, mitigating risks and ensuring safer work environments for all road users. The autonomous transportation system opens doors to unprecedented mobility, offering vulnerable populations a renewed sense of freedom and confidence in navigating complex spaces. Moreover, these systems alleviate the burdens on caregivers by providing dependable aids that reduce the constant need for supervision and assistance, redefining the relationship between caregivers and those under their care, and encouraging greater independence while ensuring safety and peace of mind.

REFERENCES

- [1] "Lane departure warning system," Wikipedia, https://en.wikipedia.org/wiki/Lane_departure_warning_system (accessed Dec. 27, 2023).
- [2] "Press corner," European Commission - European Commission, https://ec.europa.eu/commission/presscorner/detail/en/ip_22_4312 (accessed Dec. 27, 2023).
- [3] C. Y. Low, H. Zamzuri and S. A. Mazlan, "Simple robust road lane detection algorithm," 2014 5th International Conference on Intelligent and Advanced Systems (ICIAS), Kuala Lumpur, Malaysia, 2014, pp. 1-4, doi: 10.1109/ICIAS.2014.6869550.
- [4] L. Chandrasekar and G. Durga, "Implementation of Hough Transform for image processing applications," 2014 International Conference on Communication and Signal Processing, Melmaruvathur, India, 2014, pp. 843-847, doi: 10.1109/ICCSP.2014.6949962.
- [5] A. R., "Hough transforms in image processing," Scaler Topics, <https://www.scaler.com/topics/hough-transform-in-image-processing/> (accessed Dec. 28, 2023).
- [6] N. Vykari, "Understanding Hough Transform with a lane detection model," Paperspace Blog, <https://blog.paperspace.com/understanding-hough-transform-lane-detection/> (accessed Dec. 28, 2023).
- [7] T. Tasneem and Z. Afroze, "A New Method of Improving Performance of Canny Edge Detection," 2019 2nd International Conference on Innovation in Engineering and Technology (ICIET), Dhaka, Bangladesh, 2019, pp. 1-5, doi: 10.1109/ICIET48527.2019.9290676.
- [8] RideScapes, "ASMR highway driving at night (no talking, no music) - busan to Seoul, Korea," YouTube, https://www.youtube.com/watch?v=nABR88G_2cE (accessed Dec. 30, 2023).
- [9] A. Rosebrock, "Zero-parameter, automatic canny edge detection with python and opencv," PyImageSearch, <https://pyimagesearch.com/2015/04/06/zero-parameter-automatic-canny-edge-detection-with-python-and-opencv/> (accessed Dec. 30, 2023).
- [10] "Hough Line transform," OpenCV, https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html (accessed Dec. 30, 2023).
- [11] "Driver monitoring system," Wikipedia, https://en.wikipedia.org/wiki/Driver_monitoring_system (accessed Jan. 3, 2024).
- [12] Council of European Union, Council regulation (EU) no 2019/2144, <https://eur-lex.europa.eu/eli/reg/2019/2144/oj> (accessed Jan. 3, 2024).
- [13] "Classes," Dlib C++ Library, <http://dlib.net/python/index.html> (accessed Jan. 3, 2024).
- [14] "Trained 68-Point Facial Landmark Model," Dlib C++ Library, http://dlib.net/files/shape_predictor_68_face_landmarks.dat.bz2 (accessed Jan. 3, 2024).
- [15] B. Yagyesh, "Eye blink detection with opencv, python, and dlib," GeeksforGeeks, <https://www.geeksforgeeks.org/eye-blink-detection-with-opencv-python-and-dlib/> (accessed Jan. 3, 2024).
- [16] "CV::SimpleBlobDetector class reference," OpenCV, https://docs.opencv.org/3.4/d0/d7a/classcv_1_1SimpleBlobDetector.html (accessed Jan. 3, 2024).
- [17] G. Malhotra, "Building a Prototype to Track Eye Gaze," dissertation, 2023
- [18] G. Malhotra, "GagandeepMalhotra/Building-a-Prototype-to-Track-Eye-Gaze," GitHub, <https://github.com/GagandeepMalhotra/Building-a-Prototype-to-Track-Eye-Gaze> (accessed Jan. 3, 2024).
- [19] MediaInfo, "How Audi's light digitization is pointing the way toward the future," Audi MediaCenter, <https://www.audi-mediacycenter.com/en/press-releases/how-audis-light-digitization-is-pointing-the-way-toward-the-future-14624> (accessed Jan. 4, 2024).
- [20] M. Dillon, "White Noise vs. Brown noise: Which one is best for sleep?," CNET, <https://www.cnet.com/health/sleep/white-noise-vs-brown-noise-which-one-is-best-for-sleep/> (accessed Jan. 4, 2024).
- [21] Passport BMW, "How to perfume the interior of a BMW with Ambient Air Package," BMW, <https://www.passportbmw.com/blogs/846/uncategorized/how-to-perfume-the-interior-of-a-bmw-with-ambient-air-package/> (accessed Jan. 4, 2024).
- [22] J. Lyons, "Struggling to stay alert? 10 best scents for waking up," WebMD, <https://www.webmd.com/sleep-disorders/features/best-scents-for-waking-up> (accessed Jan. 4, 2024).
- [23] J. B. Cicchino, "Effects of lane departure warning on police-reported crash rates," Journal of Safety Research, vol. 66, pp. 61-70, May 2018. doi:10.1016/j.jsr.2018.05.006
- [24] M. F. Ani, S. R. Kamat, M. Fukumi, and N. A. Noh, "A Critical Review on Driver Fatigue Detection and Monitoring System," International Journal of Road Safety, vol. 1, pp. 53-58, Nov. 2020.
- [25] T. Arakawa, "Trends and future prospects of the drowsiness detection and estimation technology," Sensors, vol. 21, no. 23, p. 7921, Nov. 2021. doi:10.3390/s21237921
- [26] Seeing Machines Limited, Majority of UK road users believe new eye-tracking technology to check driver attentiveness could help improve road safety, but public awareness remains low, according to survey, <https://www.prnewswire.com/apac/news-releases/majority-of-uk-road-users-believe-new-eye-tracking-technology-to-check-driver-attentiveness-could-help-improve-road-safety-but-public-awareness-remains-low-according-to-survey-301838223.html> (accessed Jan. 4, 2024).
- [27] A. Rosebrock, "OPENCV shape detection," PyImageSearch, <https://pyimagesearch.com/2016/02/08/opencv-shape-detection/> (accessed Jan. 5, 2024).
- [28] "Ramer-Douglas-Peucker algorithm," Wikipedia, <https://w.wiki/3X7H> (accessed Jan. 5, 2024).
- [29] H. I. Adikari, "HishanIndrajith/ArrowDirectionPythonOpenCV," GitHub, <https://github.com/HishanIndrajith/ArrowDirectionPythonOpenCV> (accessed Jan. 5, 2024).
- [30] A. Sicius, "Arrow (triangle) orientation detection based on OpenCV Python," YouTube, <https://www.youtube.com/watch?v=zL6M0vYCr4s> (accessed Jan. 5, 2024).
- [31] MathWorks, "What is SLAM (simultaneous localization and mapping) - matlab & simulink," What Is SLAM (Simultaneous Localization and Mapping) - MATLAB & Simulink - MATLAB & Simulink, <https://uk.mathworks.com/discovery/slam.html> (accessed Jan. 5, 2024).
- [32] M. Chghaf, S. Rodriguez, and A. E. Ouadi, "Camera, Lidar and multi-modal SLAM systems for Autonomous Ground Vehicles: A survey," Journal of Intelligent & Robotic Systems, vol. 105, no. 1, 2022. doi:10.1007/s10846-022-01582-8

APPENDIX

A. Lane Detection and Tracking Source Code

```
import cv2
import numpy as np
import time
import math
import matplotlib.pyplot as plt
from consoledraw import Console

# Frame capture parameters
VIDEO_CAPTURE = "/full/path/to/video.mp4"
CAPTURE_WIDTH = 640
CAPTURE_HEIGHT = 480
# Frame crop parameters
CROP_MIN_HEIGHT = 340
CROP_MAX_HEIGHT = 480
CROP_MIN_WIDTH = 100
CROP_MAX_WIDTH = 560
# Canny parameters
CANNY_THRESHOLD_SIGMA = 0.4
# Canny output crop parameters
RECT_0_0 = (165, 0)
RECT_1_0 = (280, 0)
RECT_0_1 = (0, 140)
RECT_1_1 = (460, 140)
```



```

# The Hough Transform Parameters
# The resolution of the parameter r in pixels. default is 1 pixel. (OpenCV docs)
HT_RHO = 1
# The resolution of the parameter theta in radians. default is 1 degree. (OpenCV docs)
HT_THETA = np.pi / 180
# The minimum number of intersections to "detect" a line. (OpenCV docs)
HT_THRESHOLD = 60
HT_MIN_LINE_ANGLE = 130          # Minimum angle for line to be detected
HT_MAX_LINE_ANGLE = 140          # Maximum angle for line to be detected
# Lane departure parameters
LEFT_ALERT_THRESHOLD = 100
RIGHT_ALERT_THRESHOLD = 350
LANE_POSITION_VARIANCE_THRESHOLD = 3
MIN_LANE_POSITION_RECORDINGS = 500
MAX_LANE_POSITION_RECORDINGS = 2000
LPV_ALERT_THRESHOLD = 1500
# Program output parameters
# If true, parameters are output in a user-friendly way. If false, real-time metrics
# are output for each frame for efficient transferring into MS Excel for processing
USE_CONSOLEDRAW = True

# IMPORTANT: when running this Python script with ConsoleDraw output enabled, it
# is imperative that the console windows that's used to start this
# script
# is big enough for the output, the program will fail to start with an
# error:
# "ValueError: The console is too small to display the buffer."

# NOTE: this program will start in paused mode, therefore when executed the video
# feed
# will pause on the first frame and will not start until you press the 's'
# key.
# This feature gives the user enough time to arrange the output windows
# before
# starting.

# instantiate console from consoledraw package
if USE_CONSOLEDRAW:
    console = Console()
    format = """
        Tot. frames: {}
        Avg. FPT: {} ms      (target: {} ms)
        Avg. FPS: {} fps     (target: {} fps)
        Lane Departure:
        L: {}
        V: {}
        R: {}
        LPV: {}             (# readings: {})
        {}
        {}
    """
else:
    print("Frame# FPT LPV LanPosItems#")

# initialise video capture
capture = cv2.VideoCapture(VIDEO_CAPTURE)
# break program if capture cannot be opened
if not capture.isOpened():
    print("Cannot open video capture")
    exit()

# Initialise capture resolution
capture.set(cv2.CAP_PROP_FRAME_WIDTH, CAPTURE_WIDTH)
capture.set(cv2.CAP_PROP_FRAME_HEIGHT, CAPTURE_HEIGHT)

# initialise windows for output
cv2.namedWindow("(a) Original")
cv2.namedWindow("(b) Cropped")
cv2.namedWindow("(c) Grayscale")
cv2.namedWindow("(d) Histogram Equalisation")
cv2.namedWindow("(e) Gaussian Blur")
cv2.namedWindow("(f) Canny")
cv2.namedWindow("(g) Clean Canny")
cv2.namedWindow("(h) Hough Transform")

# initialise the array to store frame processing times
frame_processing_times = []

# get target FPS and FPT from input stream
capture_fps = round(capture.get(cv2.CAP_PROP_FPS), 0)
target_fpt = round((1/capture_fps) * 1000, 2)

# initialise lane assist data
right_lane_alert = False
left_lane_alert = False
lane_positions = []
car_positions = []
lane_position_variance = 0

# capture first frame of stream
success, img = capture.read()

# track if feed is paused by user
paused = True

# continuously capture video feed
while success:
    # Get the dimensions of the frame
    frame_height, frame_width = img.shape[:2]

    # Calculate the center of the frame
    center_x, center_y = frame_width // 2, frame_height // 2

    # calculate the cropping boundaries based on the specified dimensions
    crop_start_x = max(center_x - (CAPTURE_WIDTH // 2), 0)
    crop_end_x = min(center_x + (CAPTURE_WIDTH // 2), frame_width)
    crop_start_y = max(center_y - (CAPTURE_HEIGHT // 2), 0)
    crop_end_y = min(center_y + (CAPTURE_HEIGHT // 2), frame_height)

    # adjust the cropping if it exceeds the frame boundaries

    if crop_end_x - crop_start_x < CAPTURE_WIDTH:
        if crop_end_x < frame_width:
            crop_end_x = min(crop_end_x + (CAPTURE_WIDTH - (crop_end_x -
            crop_start_x)), frame_width)
        else:
            crop_start_x = max(crop_start_x - (CAPTURE_WIDTH - (crop_end_x -
            crop_start_x)), 0)

    # perform the cropping
    img = img[crop_start_y:crop_end_y, crop_start_x:crop_end_x]

    # Get key stroke
    key = cv2.waitKey(1)

    # exit if 'e' key pressed
    if key == ord('e'):
        break

    # pause and enter measure mode when 'm' is pressed
    if key == ord('m'):
        # initialise matplotlib sub plots
        fig, ax = plt.subplots(3, 3, figsize=(10, 10))
        # load in subplots
        ax[0,0].imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB), origin='upper')
        ax[0,1].imshow(cv2.cvtColor(cropped, cv2.COLOR_BGR2RGB), origin='upper')
        ax[0,2].imshow(grayScale, cmap='gray', origin='upper')
        ax[1,0].imshow(histequ, cmap='gray', origin='upper')
        ax[1,1].imshow(gaussian, cmap='gray', origin='upper')
        ax[1,2].imshow(canny, cmap='gray', origin='upper')
        ax[2,0].imshow(clean_canny, cmap='gray', origin='upper')
        # turn off axis for the columns that is spanned
        ax[2,1].axis('off')
        ax[2,2].axis('off')
        # The Hough plot to span two columns
        ax[2,1] = plt.subplot2grid((3, 3), (2, 1), colspan=2)
        # load in the Hough output as a subplot
        ax[2,1].imshow(cv2.cvtColor(cropped_lines, cv2.COLOR_BGR2RGB), origin='
        upper')
        # set subplot titles
        ax[0,0].set_title("(a) Original")
        ax[0,1].set_title("(b) Cropped")
        ax[0,2].set_title("(c) Grayscale")
        ax[1,0].set_title("(d) Histogram Equalisation")
        ax[1,1].set_title("(e) Gaussian Blur")
        ax[1,2].set_title("(f) Canny")
        ax[2,0].set_title("(g) Clean Canny")
        ax[2,1].set_title("(h) Hough Transform")
        plt.draw()
        plt.show()

    # enter paused mode with 'p' key
    if key == ord('p'):
        paused = True

    # record the start time of frame processing iteration
    frame_processing_start = time.time()

    # Display original video feed frame in window
    cv2.imshow("(a) Original", img)

    # Crop the image to isolate ROI
    cropped = img[crop_min_height:crop_max_height, crop_min_width:crop_max_width]
    # Display cropped video feed frame in window
    cv2.imshow("(b) Cropped", cropped)

    # Single-channel conversion using grayscale filter
    grayScale = cv2.cvtColor(cropped, cv2.COLOR_BGR2GRAY)
    # Display cropped video feed frame in window
    cv2.imshow("(c) Grayscale", grayScale)

    # Histogram equalisation to improve image contrast
    histequ = cv2.equalizeHist(grayScale)
    # Display histogram equalised frame in window
    cv2.imshow("(d) Histogram Equalisation", histequ)

    # Gaussian Blur to reduce noise and smoothening
    gaussian = cv2.GaussianBlur(histequ, (5,5), 0)
    # Display blurred output frame in window
    cv2.imshow("(e) Gaussian Blur", gaussian)

    # Compute the median single-channel pixel intensities
    gaussian_median = np.median(gaussian)
    # Compute threshold values using image median and constant sigma offset
    lower_threshold = int(max(0, (1.0 - CANNY_THRESHOLD_SIGMA) * gaussian_median))
    upper_threshold = int(min(255, (1.0 + CANNY_THRESHOLD_SIGMA) * gaussian_median)
    )

    # Perform the Canny edge detection
    canny = cv2.Canny(gaussian, lower_threshold, upper_threshold)
    # Display all Canny-detected edges
    cv2.imshow("(f) Canny", canny)

    # Remove Canny detections outside of the region of interest
    mask = np.zeros_like(canny)
    # using Numpy, a mask can be created to eliminate Canny data outside of POLY
    vertices = np.array([(RECT_0_0, RECT_1_0, RECT_1_1, RECT_0_1]), dtype=np.int32)
    cv2.fillPoly(mask, vertices, 255)
    # apply bitmask to eliminate data outside of ROI
    clean_canny = cv2.bitwise_and(canny, mask)
    cv2.imshow("(g) Clean Canny", clean_canny)

    # The Standard Hough Line Transform
    lines = cv2.HoughLines(clean_canny, HT_RHO, HT_THETA, HT_THRESHOLD, None, 0, 0)
    # Copy images that will display the results in BGR
    cropped_lines = cropped.copy()
    # Draw the lines
    if lines is not None:
        for i in range(0, len(lines)):
            rho = lines[i][0][0]
            theta = lines[i][0][1]
            a = math.cos(theta)
            b = math.sin(theta)

```

```

x0 = a * rho
y0 = b * rho
pt1 = (int(x0 + 1000*(-b)), int(y0 + 1000*(a)))
pt2 = (int(x0 - 1000*(-b)), int(y0 - 1000*(a)))
# scale lines to fit frame
frame_width = CROP_MAX_WIDTH - CROP_MIN_WIDTH
frame_height = CROP_MAX_HEIGHT - CROP_MIN_HEIGHT
# ensure that pt1 and pt2 fit within the frame using scaling and
transforming
if pt1[0] < 0 or pt1[0] > frame_width or pt1[1] < 0 or pt1[1] >
frame_height or pt2[0] < 0 or pt2[0] > frame_width or pt2[1] < 0
or pt2[1] > frame_height:
# Calculate line equation y = mx + c
m = (pt2[1] - pt1[1]) / (pt2[0] - pt1[0]) if pt2[0] - pt1[0] != 0
else 1
c = pt1[1] - m * pt1[0]
# Clip points to fit within the frame
if pt1[0] < 0 or pt1[0] > frame_width or pt1[1] < 0 or pt1[1] >
frame_height:
if pt1[0] < 0:
pt1 = (0, int(m * 0 + c))
elif pt1[0] > frame_width:
pt1 = (frame_width, int(m * frame_width + c))
if pt1[1] < 0:
pt1 = (int(-c / m), 0)
elif pt1[1] > frame_height:
pt1 = (int((frame_height - c) / m), frame_height)
if pt2[0] < 0 or pt2[0] > frame_width or pt2[1] < 0 or pt2[1] >
frame_height:
if pt2[0] < 0:
pt2 = (0, int(m * 0 + c))
elif pt2[0] > frame_width:
pt2 = (frame_width, int(m * frame_width + c))
if pt2[1] < 0:
pt2 = (int(-c / m), 0)
elif pt2[1] > frame_height:
pt2 = (int((frame_height - c) / m), frame_height)
# compute the angle of the detected road lane marking
line_angle = np.degrees(np.arctan2(pt1[1] - pt2[1], pt1[0] - pt2[0]))
# only process the lines that are within the specified angle range to
filter false detections
if (HT_MIN_LINE_ANGLE < abs(line_angle) < HT_MAX_LINE_ANGLE):
if (pt1[1] == frame_height):
# this is the detection of the left lane marking
lane_positions.append((pt1[0], -1))
if (pt1[0] > LEFT_ALERT_THRESHOLD):
# alert if position is past the threshold
left_lane_alert = True
line_colour = (0,0,255)
else:
# deactivate alert otherwise
left_lane_alert = False
line_colour = (0,255,0)
# draw the circle to mark the base position of the lane marking
cv2.circle(cropped_lines, center=pt1, radius=10, thickness=2,
color=line_colour)
else:
# this is the detection of the right lane marking
lane_positions.append((-1, pt2[0]))
if (pt2[0] < RIGHT_ALERT_THRESHOLD):
# alert if position is past the threshold
right_lane_alert = True
line_colour = (0,0,255)
else:
# deactivate alert otherwise
right_lane_alert = False
line_colour = (0,255,0)
# draw the circle to mark the base position of the lane marking
cv2.circle(cropped_lines, center=pt2, radius=10, thickness=2,
color=line_colour)
# display the line
cv2.line(cropped_lines, pt1, pt2, line_colour, 2)

cv2.imshow("(h) Hough Transform", cropped_lines)

# check if there are any recorded lane position
if (len(lane_positions) > MIN_LANE_POSITION_RECORDINGS):
# retrieve left and right lane marking positions
left_positions = list(zip(*lane_positions))[0]
right_positions = list(zip(*lane_positions))[1]
# convert to numpy arrays for faster computation
left_positions_arr = np.array(left_positions)
right_positions_arr = np.array(right_positions)
# find indices of valid numbers (numbers not equal to -1)
left_valid_indices = np.where(left_positions_arr != -1)[0]
right_valid_indices = np.where(right_positions_arr != -1)[0]
# find indices where -1 (invalid values) needs replacing with interpolated
values
left_replace_indices = np.where(left_positions_arr == -1)[0]
right_replace_indices = np.where(right_positions_arr == -1)[0]
# calculate the interpolated values
left_interpolated_values = np.interp(left_replace_indices,
left_valid_indices, left_positions_arr[left_valid_indices])
right_interpolated_values = np.interp(right_replace_indices,
right_valid_indices, right_positions_arr[right_valid_indices])
# replace -1 with interpolated values
left_positions_arr[left_replace_indices] = left_interpolated_values
right_positions_arr[right_replace_indices] = right_interpolated_values
# update non-numpy array
lane_positions = list(zip(left_positions_arr.tolist(), right_positions_arr.
tolist()))
# calculate the car's position w.r.t the lanes
car_positions.append(
(lane_positions[-1][0] + lane_positions[-1][1])/2
)
# calculate variance of car's position w.r.t the lanes
lane_position_variance = np.var(car_positions)
# clear older lane position readings (garbage-collection of older
recordings)
if (len(lane_positions) > MAX_LANE_POSITION_RECORDINGS):
# get number of excess
excess_vals = len(lane_positions) - MAX_LANE_POSITION_RECORDINGS
# clear the excess from the start of the list (clearing older excess
values)
del lane_positions[:excess_vals]

# record the end time of frame processing iteration
frame_processing_end = time.time()

# calculate and append frame processing time to array
frame_processing_times.append(frame_processing_end - frame_processing_start)

if USE_CONSOLEDRAW:
with console:
num_frames = len(frame_processing_times)
sum_fpt = np.sum(frame_processing_times)
try:
current_lane_pos = lane_positions[-1]
current_car_pos = car_positions[-1]
if left_lane_alert:
lane_alert = "LEFT LANE DEPARTURE ALERT!"
elif right_lane_alert:
lane_alert = "RIGHT LANE DEPARTURE ALERT!"
else:
lane_alert = "----"
if lane_position_variance > LPV_ALERT_THRESHOLD:
lpv_alert = "LANE POSITION VARIANCE ALERT!"
else:
lpv_alert = "----"
except:
current_lane_pos = "----"
current_car_pos = "----"
lane_alert = "----"
lpv_alert = "----"
if num_frames > 0:
avg_fpt = sum_fpt / num_frames
console.print(
format.format(
num_frames,
round(avg_fpt * 1000, 2),
target_fpt,
math.trunc(1/avg_fpt),
capture_fps,
current_lane_pos[0],
current_lane_pos[0],
current_car_pos[1],
math.trunc(lane_position_variance),
len(lane_positions),
lane_alert,
lpv_alert
)
)
else:
# print a space-delimited output of real-time metrics for easy copy-pasting
into MS Excel for graphing
print(len(frame_processing_times), frame_processing_end -
frame_processing_start, lane_position_variance, len(lane_positions))

# Pause at start to give enough time to move windows to good locations
if paused:
while True:
key = cv2.waitKey(1)
# Resume feed with 's' key
if key == ord('s'):
paused = False
break

# Capture the next frame of the video feed
success, img = capture.read()

# destroy windows and release video capture for a clean exit
cv2.destroyAllWindows()
capture.release()

```

Listing 1. Source code for the lane detection and tracking implementation, submitted as lane.py

B. Gaze and Blink Detection and Tracking Source Code

```

import cv2
import time
import math
import numpy as np
from consoledraw import Console
import dlib
from imutils import face_utils
from scipy.spatial import distance as dist

# Frame capture parameters
VIDEO_CAPTURE = 0
CAPTURE_WIDTH = 640
CAPTURE_HEIGHT = 480
# Path to shape predictor data
# download the file from http://dlib.net/files/shape_predictor_68_face_landmarks.
dat.bz2,
# extract the .dat file to a location, then paste in the full path of the .dat
file in the
# constant below
FACE_LANDMARK_DATA_PATH = "/full/path/to/shape_predictor_68_face_landmarks.dat"
# Blink detection parameters
BLINK_EAR_THRESHOLD = 0.4
BLINK_LOG_MAX_RECORDINGS = 600 # store a max of 10 mins worth of blink logs for 30
FPS footage
TIREDNESS_EAR_THRESHOLD = 0.35
# Gaze detection parameters
EYE_BININV_MAX_THRESHOLD = 255
EYE_BININV_MIN_THRESHOLD = 25

console = Console()
format = ""
Tot. frames: {}

```

```

Avg. FPT: {} ms      (target: {} ms)
Avg. FPS: {} fps     (target: {} fps)
Eye tracking:
Blink: {} (Avg. EAR: {})
Blink score: {}

"""
{}

# initialise video capture
capture = cv2.VideoCapture(VIDEO_CAPTURE)
# break program if capture cannot be opened
if not capture.isOpened():
    print("Cannot open video capture")
    exit()

# Initialise capture resolution
capture.set(cv2.CAP_PROP_FRAME_WIDTH, CAPTURE_WIDTH)
capture.set(cv2.CAP_PROP_FRAME_HEIGHT, CAPTURE_HEIGHT)

# initialise windows for output
cv2.namedWindow("(a) Original")
cv2.namedWindow("(b) Grayscale")
cv2.namedWindow("(c) Histogram Equalisation")
cv2.namedWindow("(d) Gaussian Blur")
cv2.namedWindow("(e) Face Detection")
cv2.namedWindow("(f) Left Eye Detection")
cv2.namedWindow("(f) Right Eye Detection")

# initialise the array to store frame processing times
frame_processing_times = []

# get target FPS and FPT from input stream
capture_fps = round(capture.get(cv2.CAP_PROP_FPS), 0)
target_fpt = round((1/capture_fps) * 1000, 2)

# initialise models for landmark and face detection
detector = dlib.get_frontal_face_detector()
landmark_predict = dlib.shape_predictor(FACE_LANDMARK_DATA_PATH)
tiredness_score = 0

# initialise eye landmarks
(L_start, L_end) = face_utils.FACIAL_LANDMARKS_IDXS['left_eye']
(R_start, R_end) = face_utils.FACIAL_LANDMARKS_IDXS['right_eye']

# initialise Simple Blob Detector parameters for iris\pupil tracking
sbd_params = cv2.SimpleBlobDetector_Params()
# disable thresholding process since input will already be grayscale
sbd_params.thresholdStep = 255
sbd_params.minRepeatability = 1
# detect blobs based on grayscale colour
sbd_params.filterByColor = True
sbd_params.blobColor = 255
# detect blobs by area
sbd_params.filterByArea = True
sbd_params.minArea = 100000
sbd_params.maxArea = 500000
# detect blobs based on circularity
sbd_params.filterByCircularity = False
sbd_params.minCircularity = 0.0001
sbd_params.maxCircularity = np.Inf
# detect blobs based on how diameter changes with angle
sbd_params.filterByInertia = True
sbd_params.minInertiaRatio = 0.0001
sbd_params.maxInertiaRatio = np.Inf
# detect blobs based on convexity
sbd_params.filterByConvexity = True
sbd_params.minConvexity = 0.0005
sbd_params.maxConvexity = np.Inf

# initialise Simple Blob Detector function
sbd_detector = cv2.SimpleBlobDetector_create(sbd_params)

# initialise eye monitoring data
blink_detected = False
EAR_alert = False
frame_EAR_log = []
frame_gaze_log = []
avg_EAR = 0
avg_gaze = 0

# capture first frame of stream
success, img = capture.read()

# track if feed is paused by user
paused = True

# continuously capture video feed
while success:
    # Get the dimensions of the frame
    frame_height, frame_width = img.shape[:2]

    # Calculate the center of the frame
    center_x, center_y = frame_width // 2, frame_height // 2

    # calculate the cropping boundaries based on the specified dimensions
    crop_start_x = max(center_x - (CAPTURE_WIDTH // 2), 0)
    crop_end_x = min(center_x + (CAPTURE_WIDTH // 2), frame_width)
    crop_start_y = max(center_y - (CAPTURE_HEIGHT // 2), 0)
    crop_end_y = min(center_y + (CAPTURE_HEIGHT // 2), frame_height)

    # adjust the cropping if it exceeds the frame boundaries
    if crop_end_x - crop_start_x < CAPTURE_WIDTH:
        if crop_end_x < frame_width:
            crop_end_x = min(crop_end_x + (CAPTURE_WIDTH - (crop_end_x -
            crop_start_x)), frame_width)
        else:
            crop_start_x = max(crop_start_x - (CAPTURE_WIDTH - (crop_end_x -
            crop_start_x)), 0)

    # perform the cropping

    img = img[crop_start_y:crop_end_y, crop_start_x:crop_end_x]

    # Get key stroke
    key = cv2.waitKey(1)

    # exit if 'e' key pressed
    if key == ord('e'):
        break

    # enter paused mode with 'p' key
    if key == ord('p'):
        paused = True

    # record the start time of frame processing iteration
    frame_processing_start = time.time()

    # Display original video feed frame in window
    cv2.imshow("(a) Original", img)

    # Single-channel conversion using grayscale filter
    grayscale = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Display cropped video feed frame in window
    cv2.imshow("(b) Grayscale", grayscale)

    # Histogram equalisation to improve image contrast
    histequ = cv2.equalizeHist(grayscale)
    # Display histogram equalised frame in window
    cv2.imshow("(c) Histogram Equalisation", histequ)

    # Gaussian Blur to reduce noise and smoothening
    gaussian = cv2.GaussianBlur(histequ, (5, 5), 0)
    # Display blurred output frame in window
    cv2.imshow("(d) Gaussian Blur", gaussian)

    # detect faces using the dlib detector
    faces = detector(gaussian)

    # display the faces as boxes overlayed on the video feed
    for face in faces:
        # get face location coordinates
        x = face.left()
        y = face.top()
        w = face.width()
        h = face.height()

        # draw a box around detected face overlayed on video feed
        cv2.rectangle(img, (x, y), (x + w, y + h), (0, 0, 255), 2)

        # detect facial landmarks
        face_landmarks = face_utils.shape_to_np(landmark_predict(gaussian, face))

        # parse landmark data to extract left and right eye landmarks
        left_eye = face_landmarks[L_start:L_end]
        right_eye = face_landmarks[R_start:R_end]

        # calculate height of each eye
        left_eye_y1 = dist.euclidean(left_eye[1], left_eye[5])
        left_eye_y2 = dist.euclidean(left_eye[2], left_eye[4])
        right_eye_y1 = dist.euclidean(right_eye[1], right_eye[5])
        right_eye_y2 = dist.euclidean(right_eye[2], right_eye[4])

        # calculate horizontal distance of each eye
        left_eye_x1 = dist.euclidean(left_eye[0], left_eye[3])
        right_eye_x1 = dist.euclidean(right_eye[0], right_eye[3])

        # calculate eye aspect ratio (EAR)
        left_EAR = (left_eye_y1 + left_eye_y2) / left_eye_x1
        right_EAR = (right_eye_y1 + right_eye_y2) / right_eye_x1
        avg_EAR = (left_EAR + right_EAR) / 2

        # log average EAR value
        frame_EAR_log.append(avg_EAR)

    # clear older logged EAR readings (garbage-collection of older recordings)
    if (len(frame_EAR_log) > BLINK_LOG_MAX_RECORDINGS):
        # get number of excess
        excess_vals = len(frame_EAR_log) - BLINK_LOG_MAX_RECORDINGS
        # clear the excess from the start of the list (clearing older excess
        # values)
        del frame_EAR_log[:excess_vals]

    # blink detected if EAR falls below defined threshold
    if (avg_EAR < BLINK_EAR_THRESHOLD):
        blink_detected = True
    else:
        blink_detected = False

    # get top left and bottom right coordinates for each eye
    left_eye_top_left = (left_eye[0][0], left_eye[1][1])
    left_eye_bottom_right = (left_eye_top_left[0] + int(left_eye_x1),
        left_eye_top_left[1] + int(left_eye_y1))
    right_eye_top_left = (right_eye[0][0], right_eye[1][1])
    right_eye_bottom_right = (right_eye_top_left[0] + int(right_eye_x1),
        right_eye_top_left[1] + int(right_eye_y1))

    # crop each eye out of the full sized image with the face
    left_eye_img = gaussian[left_eye_top_left[1]:left_eye_bottom_right[1],
        left_eye_top_left[0]:left_eye_bottom_right[0]]
    right_eye_img = gaussian[right_eye_top_left[1]:right_eye_bottom_right[1],
        right_eye_top_left[0]:right_eye_bottom_right[0]]

    # enlarge the extracted eye images by 50x for easier viewing
    left_eye_img = cv2.resize(left_eye_img, (50 * left_eye_img.shape[1], 50 *
        left_eye_img.shape[0]))
    right_eye_img = cv2.resize(right_eye_img, (50 * right_eye_img.shape[1], 50 *
        right_eye_img.shape[0]))

    # draw rectangles around the left and right eyes in the original feed frame
    cv2.rectangle(img, left_eye_top_left, left_eye_bottom_right, (255, 0, 0),
        1)
    cv2.rectangle(img, right_eye_top_left, right_eye_bottom_right, (0, 255, 0),

```

```

1)

# apply binary invert threshold in prep for blob detection
_, left_eye_img_thresh = cv2.threshold(left_eye_img,
    EYE_BININV_MIN_THRESHOLD, EYE_BININV_MAX_THRESHOLD, cv2.
    THRESH_BINARY_INV)
_, right_eye_img_thresh = cv2.threshold(right_eye_img,
    EYE_BININV_MIN_THRESHOLD, EYE_BININV_MAX_THRESHOLD, cv2.
    THRESH_BINARY_INV)

# detect eye blob keypoints
left_eye_kp = sbd_detector.detect(left_eye_img_thresh)
right_eye_kp = sbd_detector.detect(right_eye_img_thresh)

# draw keypoints to visualise
left_eye_img = cv2.drawKeypoints(left_eye_img, left_eye_kp, None, color
    =(0,0,255), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
right_eye_img = cv2.drawKeypoints(right_eye_img, right_eye_kp, None, color
    =(0,0,255), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

# Display the cropped left eye image
cv2.imshow("(f) Left Eye Detection", left_eye_img)
cv2.imshow("(f) Right Eye Detection", right_eye_img)

cv2.imshow("(e) Face Detection", img)

# calculate tiredness score (rolling average of EAR)
tiredness_score = np.average(frame_EAR_log)

# alert if tiredness score is below threshold
if tiredness_score < TIREDNESS_EAR_THRESHOLD:
    tiredness_alert = True
else:
    tiredness_alert = False

# record the end time of frame processing iteration
frame_processing_end = time.time()

# calculate and append frame processing time to array
frame_processing_times.append(frame_processing_end - frame_processing_start)

print(len(frame_processing_times), frame_processing_end -
    frame_processing_start)

with console:
    num_frames = len(frame_processing_times)
    sum_fpt = np.sum(frame_processing_times)
    if EAR_alert:
        tiredness_alert = "TIREDNESS ALERT!"
    else:
        tiredness_alert = "----"
    if num_frames > 0:
        avg_fpt = sum_fpt / num_frames
        console.print(
            format.format(
                num_frames,
                round(avg_fpt * 1000, 2),
                target_fpt,
                math.trunc(1/avg_fpt),
                capture_fps,
                blink_detected,
                round(avg_EAR, 2),
                round(tiredness_score, 2),
                tiredness_alert
            )
        )

# Pause at start to give enough time to move windows to good locations
if paused:
    while True:
        key = cv2.waitKey(1)
        # Resume feed with 's' key
        if key == ord('s'):
            paused = False
            break

# Capture the next frame of the video feed
success, img = capture.read()

```

Listing 2. Source code for the eye gaze and blink detection and tracking implementation, submitted as eyes.py

C. Python Dependencies for Project

```

appnope==0.1.3
asttokens==2.4.0
backcall==0.2.0
certifi==2023.11.17
charset-normalizer==3.3.2
cloudpickle==2.2.1
coloredlogs==15.0.1
comm==0.1.4
consoledraw==2.3.13
contourpy==1.2.0
cycler==0.12.1
debugpy==1.8.0
decorator==5.1.1
dlib==19.24.2
executing==2.0.0
filelock==3.13.1
flatbuffers==23.5.26
fonttools==4.45.1
fsspec==2023.12.1
humanfriendly==10.0
idna==3.6
imutils==0.5.4
ipykernel==6.25.2
ipython==8.16.1
jedi==0.19.1

```

```

Jinja2==3.1.2
jupyter_client==8.3.1
jupyter_core==5.3.2
kiwisolver==1.4.5
MarkupSafe==2.1.3
matplotlib==3.8.2
matplotlib-inline==0.1.6
mpmath==1.3.0
nest-asyncio==1.5.8
networkx==3.2.1
numpy==1.26.0
onnx==1.15.0
onnxruntime==1.16.3
opencv-contrib-python==4.8.1.78
opencv-python==4.8.1.78
packaging==23.2
parso==0.8.3
pexpect==4.8.0
pickleshare==0.7.5
Pillow==10.1.0
platformdirs==3.11.0
prompt-toolkit==3.0.39
protobuf==4.25.1
psutil==5.9.5
ptyprocess==0.7.0
pure-eval==0.2.2
Pygments==2.16.1
pyparsing==3.1.1
PyQt6==6.6.1
PyQt6-Qt6==6.6.1
PyQt6-sip==13.6.0
pyqtgraph==0.13.3
python-dateutil==2.8.2
pyzmq==25.1.1
requests==2.31.0
scipy==1.11.4
six==1.16.0
spyder-kernels==2.4.4
stack-data==0.6.3
sympy==1.12
torch==2.1.1
torchvision==0.16.1
tornado==6.3.3
traitlets==5.11.2
typing_extensions==4.8.0
urllib3==2.1.0
wcwidth==0.2.8
wurlitzer==3.0.3

```

Listing 3. List of pip dependencies that require installing prior to running the Python implementation code, submitted as requirements.txt