Smart Parking Lot Management System

Park Sense
Andrei Koshelev, Boryana Buyuklieva, Dart Musta,
Loran Pllana, Mariya Kezdekbayeva, Jana Elagamy, Volodymyr Cherednychenko
Technical University of Munich
Course: Embedded Systems, Cyber-Physical Systems, and Robotics
Professor: Amr Alanwar

August 29, 2025

Contents

Abstract

Parking availability, along with the parking access control, is crucial for solving big challenges in modern urban areas, where scaling is needed in order to solve the problem for a large number of vehicles. Traditional parking systems depend heavily on ticketing, RFID tags or sensors for each parking slot. These approaches are not as productive as they were because they cause congestion and wasted time. Our project presents the design and implementation of a parking lot prototype that encompasses in it license plate recognition(LPR) and camera-based occupancy. Also, a servomotor-controlled barrier actuation is used in this unified Cyber-Physical System.
The prototype implementation integrates multiple software and hardware components in modular subsystems, which are all implemented on the Raspberry Pi 5 platform. The entrance camera uses both OpenCV and EasyOCR to perform LPR. OpenCV is used for preprocessing and EasyOCR for text extraction, which enables the system to authenticate vehicles from a set of authorized plate that should have access on the parking. If the plate corresponds with one in the authorized plates list, the servomotor that serves as a barrier is triggered to be opened, simulating secure gate access. Simultaneously, the overhead camera monitors parking spaces through contour detection and background subtraction, and an LCD is updated dynamically whenever there is a change in any slot. Since the software design is modular, ParkingLotManager, LPRManager, UltrasonicSensorManager, and GateManager can operate independently while cooperating under a central MainManager.
Based on our testing, the system operates reliably in real time. The license plate recognition system achieved 88% accuracy under controlled lighting where as the occupancy detection reached over 80% accuracy with the servomotor responding consistently with a delay of less than 500ms. The project is a prime example to demonstrate that camera based vision, when optimized carefully, serves as an alternative to sensor-heavy infrastructures. The project also highlights and leaves space for future improvements, such as adding infrared lighting for night-time operation, incorporating deep learning-based detection, and extending to cloud-based dashboards. Eventually, the work shows us how CPS principles—sensing, computation, and actuation—can be practically applied to address real-world challenges

Chapter 1

Introduction

The rapid growth of cities and metropolises has increased the number of vehicles that pass through these cities everyday, so efficient parking management has become an urgent problem that requires immediate action. Traditional parking systems rely heavily on physical tokens, such as RFID cards or paper tickets, and often deploy individual sensors like ultrasonic or infrared modules to monitor each slot. These solutions are functional, but they have considerable drawbacks like high installation costs, difficulty in scaling when the number of vehicles to be processed is very large and maintenance issues. Also, most of them do not provide instant feedback to drivers, which can lead to congestion, wasted fuel, and driver frustration.

In comparison to the traditional, the smart parking solutions leverage advances in computer vision, embedded computing, and real-time control which provides a seamless user experience while reducing all of the high infrastructure costs. By using cameras for occupancy detection and authentication, these modern systems eliminate the need for sensors in every slot and manual ticketing systems. However, even this modern approach has its own challenges in terms of computational load, its robustness under different lighting conditions and also with the integration of all possible complex components into one main Cyber-Physical-System

This project was imagined as a practical demonstration of how CPS principles can be applied to a certain problem in the real world, in this case to the actual problem of parking management. The core objectives are as follow: first, authentication of vehicles that want to enter the parking through license plate recognition without requiring any physical tag; second is to automate the gate control using a servo motor in response as commanded through the authentication results; and third to monitor the parking available spaces by using the overhead camera. Also an ultrasonic sensor is used to let the vehicles exit the parking lot. If there is an object detected 7cm away from the inside end of the sensor then the servomotor is triggered and opened to let the vehicle exit. These goals are realized by using a modular software architecture that divides the responsibilities among the components: the ParkingLotManager monitors slot occupancy, the LPRManager recognizes license plates, UltrasonicSensorManager handles the exit lane and the GateManager controls the servo barrier. All subsystems are orchestrated by MainManager.

The system provides real time user feedback using LCD display and LED indicators, making it very user-friendly and also intuitive for the drivers. During our testing, the system achieved positive results: license plate recognition was accurate in most conditions, occupancy detection functioned reliably, and gate actuation was smooth and responsive. This functional prototype demonstrates how vision based techniques can reduce costs without changing anything on the performance domain, and how CPS design principles-sensing, computation and actuation can be integrated into a real-time embedded environment. The report describes in detail the design, implementation, and evaluation of the system, focusing on interaction between software modules and embedded hardware.

Chapter 2

Methodology and Implementation

2.1 Parking Lot Monitoring

The parking lot monitoring module is responsible for determining whether some predefined parking spaces are available or not. In order to achieve this, the system first captures a background image of the lot when it is empty. This then serves as a reference against which all following frames are compared to

Listing 2.1: Capturing and preprocessing background frame.

```
ret, background_frame = self.cap.read()
background_gray = cv2.cvtColor(background_frame, cv2.COLOR_BGR2GRAY)
self.background_blur = cv2.GaussianBlur(background_gray, (21, 21), 0)
```

This code begins by capturing a frame from the camera, which stores it as a background reference. The captured frame is then converted into grayspace, as color information is redundant, and this way we are reducing the computational load. Finally, Gaussian blurring with a kernel of size 21×21 is applied to smooth out minor noise that could come into the system, such as shadows, which prevents false positives during the comparison stage.
Once the initial frame is established, the system continuously compares new frames with the background. The absolute difference is computed. Also it is followed by a threshold to highlight significant changes and contour detection to extract forms and shapes of objects regardless if they are moving or stationary.

Listing 2.2: Detecting contours for slot occupancy.

```
frame_diff = cv2.absdiff(self.background_blur, current_blur)
_, thresh = cv2.threshold(frame_diff, 30, 255, cv2.THRESH_BINARY)
dilated = cv2.dilate(thresh, None, iterations=2)
contours, _ = cv2.findContours(dilated, cv2.RETR_EXTERNAL, cv2.
   CHAIN_APPROX_SIMPLE)
```

The function `cv2.absdiff` calculates pixel-wise differences between the blurred background and the current blurred frame, which focuses and highlight only the areas that have changed. Binary threshold then converts these differences into black and white pixels, with a special marking of potential objects with white. Finally, contours are extracted from the threshold image, which allows the system to examine their position and to determine whether it overlaps with the predefined parking slot region

After the identification of available slots is finished, the system provides feedback in both textual and visual forms. The green LED is illuminated when the space is free, while the red color is used if the lot is occupied. This is all done concurrently with the updating of the current count of available spaces on the LCD display.

Listing 2.3: Updating LEDs and LCD based on availability.

```
if free_count > 0:
    self.led_green.on(); self.led_red.off()
else:
    self.led_green.off(); self.led_red.on()

self.lcd.lcd_display_string("Empty␣parking", 1)
self.lcd.lcd_display_string(f"spaces:␣{free_count}.", 2)
```

The snippet provides a look into the feedback mechanism. If the count of free slots is greater than zero,the system activates the green LED and the red one is turned off, signaling that cars may still enter it. Conversely, when the lot is full, the red LED is switched on, and LCD display shows the current number of remaining spaces, making sure that the drivers are informed in real time.

2.2 License Plate Recognition

The license plate recognition (LPR) subsystem authenticates incoming vehicles based on their plates. To avoid blocking the main program loop, the recognition task is offloaded to a worker thread.

Listing 2.4: Starting the OCR worker thread.

```
self.ocr_thread = threading.Thread(target=self.ocr_worker, daemon=True)
self.ocr_thread.start()
```

The code initializes a new thread that runs `ocr_worker` function in an independent manner from the main loop. It being marked as a daemon, ensures that the process terminates cleanly when the program ends. This design allows the main camera feed to continue uninterrupted while OCR runs in parallel.

Within the worker thread, EasyOCR is used to process regions of interest cropped from the bottom of each frame. The following snippet illustrates how OCR results are processed and cleaned:

Listing 2.5: Using EasyOCR to recognize plate text.

```
results = reader.readtext(roi)
for (_, text, conf) in results:
    cleaned = ''.join(filter(str.isalnum, text.upper()))
    if conf > best_conf and 2 <= len(cleaned) <= 10:
        best_match = cleaned
```

We use OCR reader to return a list of recognized strings with some associated confidence levels. The loop performs the iteration through each potential candidate and filters out non-alphanumeric characters, normalizing to uppercase to match license plate conventions. Only string of length from 2 to 10 characters are acceptable, and the results with the highest confidence is the one that is stored as the best match.

If the returned plate matches one from the database of vehicles that are authorized, the system grants access by instructing the gate manager to open. If not, the frame is annotated to show the plate is not recognized

Listing 2.6: Granting or denying access.

```
if self.plate_text in AUTHORIZED_PLATES:
    self.gate_manager.open_gate()
    cv2.putText(frame, "ACCESS␣GRANTED", (10, 60),
                cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0,255,0), 2)
```

```
else:
    cv2.putText(frame, "UNKNOWN␣PLATE", (10, 60),
                cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0,0,255), 2)
```

In this section, the integration of recognition and actuation is demonstrated. Results of a successful match are the opening of the gate and a message “ACCESS GRANTED” being overlaid on the video.If no match is to be found the text “UNKNOWN PLATE” is displayed in red, denying entry.

2.3 Gate Control

The gate control subsystem uses a micro servo motor connected via GPIO to simulate a barrier. First, the servo must be initialized and configured to receive PWM signals.

Listing 2.7: Initializing servo PWM.

```
GPIO.setup(self.servo_pin, GPIO.OUT)
self.pwm = GPIO.PWM(self.servo_pin, 50)
self.pwm.start(0)
```

In this code the servo pin is configured as an output and the PWM is set with a frequency of 50Hz, which is considered to be the standard for servomotor. The servo then starts with a duty cycle of 0, ensuring that it doesn’t perform any unexpected behavior during it’s initialization

The servo angle is then set by converting the desired rotating angle into a duty cycle, which is handled by the following code:

Listing 2.8: Setting servo angle.

```
def set_angle(self, angle):
    duty = angle / 18 + 2
    self.pwm.ChangeDutyCycle(duty)
    time.sleep(0.5)
    self.pwm.ChangeDutyCycle(0)
```

The formula `angle/18 + 2` converts an angle in degrees into a corresponding PWM duty cycle.

After setting the duty cycle, the program waits 500ms for the servomotor to reach the position it is targeting and then to prevent jittering, it resets the duty cycle back to zero. This ensures precise and stable gate movements.

The opening and the closing of the gate are done simply as a wrapper function using `set_angle`, with angles chosen to simulate the real barrier.

Listing 2.9: Open and close methods for servo barrier.

```
def open_gate(self):
    self.set_angle(150)
    self.is_open = True

def close_gate(self):
    self.set_angle(60)
    self.is_open = False
```

The 150 degrees here corresponds to the open position of the barrier, while the 60 degrees represents the closed position. By tracking the state of the gate with `is_open` variable, the system is able to ignore redundant commands, which ensures efficient actuations.

2.4 Ultrasonic Exit Trigger

Taking into consideration that license plate recognition is responsible for the entry, the exit lane is the one that benefits from a faster, frictionless mechanism that doesn't require OCR. In order to achieve this we mounted an ultrasonic sensor to the opposite of the lane, in the direction facing the vehicles while they exit the barrier. This design ensures that no authorization or verification is needed when leaving the lot, thereby improving throughput and user convenience. Exit sensor is managed by a class of its own and executed in a special thread, ensuring that distance polling doesn't interfere with other CPS tasks.

Listing 2.10: Ultrasonic sensor GPIO setup and worker thread for exit.

```
GPIO.setmode(GPIO.BCM)
GPIO.setup(self.trig_pin, GPIO.OUT)
GPIO.setup(self.echo_pin, GPIO.IN)
self.sensor_thread = threading.Thread(target=self.sensor_worker, daemon
    =True)
self.sensor_thread.start()
```

In this snippet the ultrasonic sensor is initialized using BCM numbering to remain consistent with the rest of the GPIO pins. The trigger pin is the one which is configured as an output, and the echo pin the one as an input, matching the typical C-SR04 wiring convention. Then a daemon thread is created so that the sensor measures the distance in the background continuously, leaving the main loop free to handle other subsystems of the parking lot

Listing 2.11: Time-of-flight distance measurement for exit trigger.

```
GPIO.output(self.trig_pin, False)
time.sleep(0.000002)
GPIO.output(self.trig_pin, True)
time.sleep(0.00001)
GPIO.output(self.trig_pin, False)

# wait for echo rise/fall with timeouts
while GPIO.input(self.echo_pin) == 0:
    pulse_start = time.time()
    if time.time() - timeout_start > 1: return 99999
while GPIO.input(self.echo_pin) == 1:
    pulse_end = time.time()
    if time.time() - timeout_start > 1: return 99999

pulse_duration = pulse_end - pulse_start
distance = (pulse_duration * 34300) / 2
```

In this part the sensor is the one emitting a short trigger pulse, which then measures the duration of the returning echo to compute distance using the speed of sound. Then a division by two is responsible for the round trip of the sound wave. Timeout checks are the ones preventing the loop from hanging in special cases of reflection or any sensor misreading. This type of measuring gives an approximation of car distance to the barrier

Listing 2.12: Exit barrier opening on proximity threshold.

```
distance = self.get_distance()
if distance < DETECTION_DISTANCE_CM and not self.gate_manager.is_open:
    self.gate_manager.open_gate()
time.sleep(0.5)
```

This worker compares the measured length between the car and the sensor against a defined threshold of 7cm to decide whether to trigger the exit barrier. The condition ensures the gate opens when in the moment it is closed , this way any unnecessary actuation is prevented. The 500ms delay between polls reduces the CPU load while also being responsive to the human time scaling. In comparison to the entry lane, where OCR is the one deciding the access, the exit path depends only on proximity sensing to streamline vehicle departure.

2.5 Integration

The integration of these subsystems coordinates four managers:the GateManager for servo actuation, the ParkingLotManager for occupancy detection and user feedback, the LPRManager for OCR-based authorization, and the new UltrasonicSensorManager for proximity-based exit triggers. The main loop splits the time only for the managers with active loop cycles which are parking and LPR, while on the other hand the gate and ultrasonic managers operate event-driven via their own threads or callbacks.

Listing 2.13: Manager setup and run loop (updated).

```
gate_manager = GateManager(SERVO_PIN)
lpr_manager = LPRManager(LPR_CAM_INDEX, gate_manager)
park_manager = ParkingLotManager(PARKING_LOT_CAM_INDEX, led_green,
   led_red, lcd_display)
ultrasonic_manager = UltrasonicSensorManager(ULTRASONIC_TRIG_PIN,
   ULTRASONIC_ECHO_PIN, gate_manager)

self.loop_managers = [park_manager, lpr_manager]
self.all_managers  = [gate_manager, park_manager, lpr_manager,
   ultrasonic_manager]
```

This configuration achieves the cooperative scheduling model in the main thread while delegating continuous ranging to the daemon work. Cleanup is extended so it includes all of the managers and a final `GPIO.cleanup()` call to return pins to a safe state.

Chapter 3

Conclusion

This project set out to design but also implement a smart parking lot management functional prototype that demonstrates how vision-based methods can be integrated into a unified Cyber-Physical System. The results show that is feasible to replace the traditional ticketing system and per slot sensors with monitoring of parking spaces using a camera, provided that the system is naturally optimized for embedded hardware use.

From the technical point of view, our project has achieved its main objectives and goals set out in the beginning. License plate recognition using EasyOCR reached an accuracy of approximately 88% under optimal lighting. This proves that lightweight OCR libraries can perform well on small devices. Parking space detection using background subtraction and contour analysis achieved accuracy levels between 82% and 90%, which mostly depends on environmental factors that can be shadows and reflections. The barrier, implemented by the servomotor, responded in a reliable way within 500ms and the system's modular software architecture made the smooth integration of all subsystems possible.

Beyond the raw performance metrics, the project offers a real valuable perception into the challenges of implementing CPS in the real world and for its coming into practice. Threading and queue-based communication were critical for balancing the computational load that was shared between OCR and occupancy detection. The system proved sensitive to changes in the environment, such as the lighting, revealing the need for more advanced preprocessing or infrared support in real-world deployments in actual large-scale parking lot cases. Nevertheless, the results demonstrate that vision-based smart parking lots are a promising domain and also low cost alternatives to sensor-heavy infrastructures, especially in parking lots of moderate size parking lots.

Looking into the future, several improvements could enhance the system's robustness and scalability. Integrating infrared illumination or high dynamic range (HDR) cameras will most likely, with a high certainty, improve accuracy under unfavorable lighting . Adopting deep learning based object detections models such as SSD or YOLO could make the detection of available spaces more robust against noise such as occlusions and shadows. Furthermore, the extension of the system with cloud based dashboard or any designated mobile app would allow remote monitoring of parking availability, increasing usability for both operators and drivers.

In conclusion, the project demonstrates the potential of our project being integrated and scaled into a larger, real world smart parking. By integrating sensing and computation into a real case and real-time embedded system, the strength of CPS is highlighted by enabling more intelligent, efficient, and accessible urban infrastructure