$\ddot{U}$ber Control

Atharv Sathe
assathe@andrew.cmu.edu

Jan 2021 - May 2021

1 Introduction

Control algorithms are at the heart of robotics and Cyber Physical Systems (CPS). A typical CPS operation is dependent on the correct data from the sensors and running complicated control algorithms to drive the actuators. The ever increasing inventions of smart devices have multiple sensors to achieve autonomous behaviour. This makes security of sensor data important as tampered data will result in incorrect control actions. We use UberSpark: a verifiable object abstraction for compositional security analysis of a hypervisor. This provides a simple yet powerful approach for handling attacks against sensors to guarantee integrity of the data. UberSpark provides an open source micro-hypervisor framework which serves as a root of trust and guarantees memory protection of the sensor data. Currently we build an uberObject around the I2C communication protocol. Any sensor communicated via I2C bus is protected under the framework.

This project explores the impact of UberSpark while designing end to end control algorithms for a CPS or robotic system. As a poof of concept, we limit our experiments to a Pi-Car: a rover built around a Raspberry Pi. We demonstrate a simple control strategy for line following using the inbuilt IR sensors on the Pi-Car. Various approaches are identified and implemented to analyse the effects of cyber security framework on performance of control algorithms. Initial experiments are conducted with a single sensor, the line following sensor, and later we interface multiple sensors via I2C and propose a cascade control strategy for speed control while following the line.

2 PiCar

2.1 Model

PiCar is a rover modelled from a real life car. The rear wheels are each driven by a DC motor. The front wheels are connected to an Ackerman Steering mechanism which is driven by a servo motor. The raspberry pi allows to control the speed of each of the individual motors driving the rear wheels. It also allows to drive the steering servo to a specific angle from to . The default servo angle is set to that drives the PiCar in a straight line.

The Ackerman steering for the front wheels makes PiCar a non-holonomic systems. For simplicity, throughout the experiments, we drive both the real wheel motors at the same speed to avoid differential turning effects due to rear wheels. We can thus limit our analysis to a simple car with Ackerman steering. Fig 1 shows the relevant Ackerman geometry.
Let V_l be the velocity of the left rear wheel.
V_r be the velocity of the right rear wheel.
ϕ be the steering angle of the car.
We limit our analysis when angular velocities of both rear wheels is same. This constraint assumes the width of the car $(2 * d)$ is negligible with respect to the radius of curvature R. We say $V_l = V_r$ and is denoted by V.

$$\begin{bmatrix} \phi \\ V \end{bmatrix} = \begin{bmatrix} \tan^{-1}\left(\frac{l}{R}\right) \\ V_d \end{bmatrix} \tag{1}$$

Equation 1 gives a way to compute steering angle and velocity for a chosen radius of curvature R and linear velocity V_d of the centre of mass of the car.

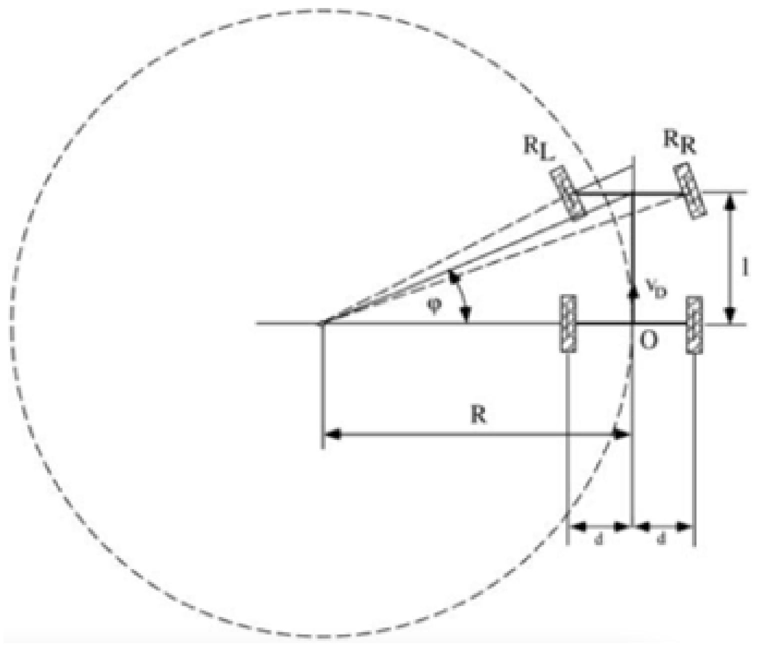

Figure 1: Ackerman steering kinematics

The state space model of the car with above kinematics and simplifications is very similar a holonomic differential drive model but with an addition constraint for the steering. Equation 2 denotes the state space model of the system and equation 3 gives the non-holonomic constraint of the system.

$$\dot{q} = \begin{bmatrix} \dot{\phi} \\ \dot{x} \\ \dot{y} \end{bmatrix} = G(q)u = \begin{bmatrix} 0 & 1 \\ \cos\phi & 0 \\ \sin\phi & 0 \end{bmatrix} \begin{bmatrix} V_d \\ \omega \end{bmatrix} \tag{2}$$

where ω is rate of change of steering and $\dot{x}$ and $\dot{y}$ are velocities in X and Y axis respectively.

$$A(q)\dot{q} = \begin{bmatrix} 0 & \sin\phi & -\cos\phi \end{bmatrix} * \dot{q} = \dot{x}\sin\phi - \dot{y}\cos\phi = 0 \tag{3}$$

We spent some time developing simulator in OpenAI gym for the PiCar that would speed up the proof of concept process for complicated control algorithms. Though the development is in the initial stages and the most of the experiments were run on the actual PiCar, we hope the simulator will prove useful when there are multiple sensors and need to perform a full test coverage on the system. The Github for the current source code is stated in references.

2.2 Sensors

The PiCar is equipped with inbuilt line following sensors. The sensor is used is infrared (IR) transmitter receiver pair mounted at the front bottom of the car facing downwards enabling to detect a line under the car. There is an array of five such transmitter receiver pair mounted perpendicular to the direction of motion. This array enables to sense the relative position of the line with respect to the car. Each of pair generates an analog value from 0-1023. Lower value denotes darker colour while higher values denote a lighter colour. In our experiments, the colour of the line is black and the background is plain white or brown wooden flooring. The sensor array communicates with the raspberry pi via I2C communication.

The second built in sensor is ultrasonic sensor mounted facing forwards to detect obstacles. We have not experimented with this sensor but have plans to explore applications in future. The interfacing libraries for both of these inbuilt sensors can be found on Sun PiCar GitHub.

We added our own IMU sensor to the PiCar. The primary sensors in consideration are MPU9250 and ICM20469. Both of them offer 9 DOF with 3 axis accelerometer, 3 axis gyro and 3 axis magnetometer. The sensors communicate with raspberry pi via I2C. The PiCar breakout board has a port to add an extra I2C device and the IMU sensor communicates via this port.

3 Constraints with UberSpark

UberSpark framework creates UberObjects to guarantee memory protection. However the framework is still a work in progress and supports basic inbuilt C functions to assure security. An exhaustive list of acceptable functions is given at: `https://github.com/uberspark/uberspark/tree/master/docs/nextgen-toolkit`. However it is possible to add a third party library as long as it does not have any dependencies on the dangerous functions and is completely written in C.

There are a few limitation to be considered when designing control algorithms in conjunction with UberSpark. The main unsupported function that poses some limitations is getting UNIX time from the OS. Getting a timestamp for various events in the system is critical for designing control algorithms. However UberSpark supports ctime() function that returns the number of clock cycles elapsed since the micro processor has been powered on. Using ctime() in raspberry pi introduces a few complications. Raspberry pi has dynamic frequency scheduler and as a result operating frequency is constantly changing depending on the current load on the processor. This dynamic nature makes it difficult to map the number of clock cycles to the elapsed time. A simple workaround used throughout these experiments is to fix the operating frequency of the raspberry pi before running any control algorithms. We need to assign the frequency each time we power on the pi.

The framework does not support dynamic memory allocation. Hence the dimensions of all system matrices and other control related parameters should be defined beforehand and place holder variables must be declared to avoid dynamic memory calls. Currently UberSpark does not support matrix operations out of the box. We are in the process of including stand alone library that supports basic operations like matrix multiplications, norm and element wise matrix operations.

4 Evaluation and Experimental Setup

The main aim of the project is to analyse and compare the performance of the proposed control algorithms with and without using the UberSpark framework. We also test the performance with the framework when the system is under attack from the intruder. The development platform for all experiments is the PiCar which is controlled by the raspberry pi and runs the latest version of raspbian OS. The model application is line following using the PiCar. We have designed two test tracks: first track is a rectangular loop with curved edges as shown in fig 2 and second track is in shape of letter S as shown in fig 3.

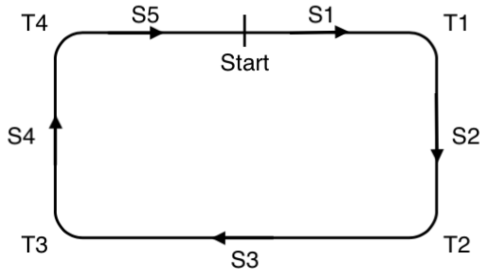

Figure 2: Rectangular Test Track

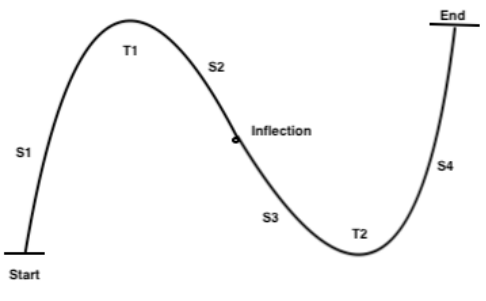

Figure 3: 'S' Test Track

A baseline result is generated by running the car on the tracks that implements a required control algorithm natively without the UberSpark framework. We measure the time required to complete the track and record the error for each loop iteration throughout the run. Definition and computation of error differs based on the control algorithm as explained in detail in Section 5. Intuitively error is a metric of the quality of the line tracking. When the car is perfectly tracking the line, error is zero. The time vs error plot is used for all the analysis throughout the project.

Then we run the same track and record the error when implanted the same control algorithm with the cyber security framework. The comparison with the baseline allows us to analyse the overheads induced by the framework. Finally we run the same experiment when an intruder is actively trying to access the secured sensor data. The process of fending off the attacks introduces extra overheads and its effects on the performance and readily seen in the time vs error graph. The comparison between these multiple settings are used to gauge the performance and viability of the control algorithm when implemented in connection with the UberSpark framework.

5 Line Following with PID

The primary sensor used for this application is the inbuilt IR transmitter receiver array mounted on the PiCar. We use PID control algorithm as shown in fig 4 to control the servo steering of the car to follow the line. The project included developing a generalised PID shared library programs in C taking into restrictions imposed by UberSpark. This approach will make it easier to convert the underlying algorithm into an uberObject in the future. The input to the algorithm in the error and the output is the deviation of the steering angle from 90. We experimented with two definitions of error to compare the performance and computation overhead of the same control algorithm. In this application, we are not interested in controlling the forward velocity of the car. Hence the motor speeds are constant. However we try modifying the velocity from 40% to 80% of the maximum velocity to gauge the robustness and response time of the control algorithm.

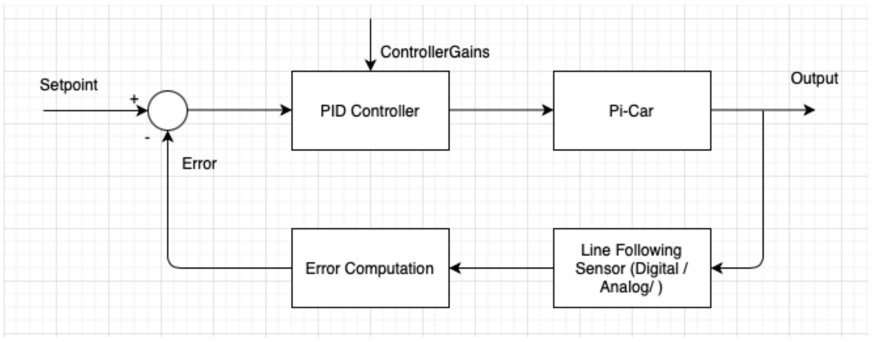

Figure 4: Feedback Loop for PID Controller

5.1 Simple Error

In this approach as given in algorithm 1 we convert the analog readings of the IR sensor array into 1 digit digital value. The ON state or digital high denotes that particular sensor as detected a line while an OFF state or digital 0 denotes a line has not been detected. We bifurcate the analog values into digital by a pre calibrated threshold. The threshold is average of the minimum value on the line and the maximum value of the background in a particular environment. We have developed a calibration script that computes this threshold value and then the user can input it to the control algorithm.

The error computation is then carried on these digital values. The error is the weighted average of all the ON infrared sensors. The weights are given as [-4, -2, 0, 2, 4]. The central sensor has no weight, thus when line is exactly below this sensor, the error value would be zero as this is the required position of the line. As the relative position of the line changes with the car, the error value can change between -4 to 4 depending on the position of the line. Negative valued denote the line is on the left side of the car while positive value denoted it is on the right side of the car. If the line is not detected by any of the sensors in the array, the algorithm retains the latest error value hoping it will find the line in the same direction in which the car went off the track.

This error value is then fed to the PID control algorithm and the deviation of the steering angle is given as the output. Due to the simplicity of the error computation and its direct relevance to the position of the line, we observe that just the proportional controller works well for this application. The gain values for the controller are Kp = 5 and Ki and Kd set to 0.

Algorithm 1: Simple Error

```
Result: Line Error
error = 0;
count = 0;
weights = [-4, -2, 0, 2, 4];
analogSignal = get_sensor_readings();
for (i = 0; i < 5; i++) do
    if analogSignal[i] < THRESHOLD then
        error += weights[i];
        count++;
    end
end
error = error / count;
```

5.2 Line Estimation

This is a more sophisticated approach where we aim to exploit the rich analog data provided by IR sensor array. The idea is to pinpoint the exact location of line in centimetres from the central sensor fusing the data from all five IR transmitter receiver pairs. This estimated distance will itself serve as an error to the control algorithm as when the line is exactly below the central sensor, the error would be 0 as expected.

In order to achieve this we manually graphed the sensor characteristics by recording the sensor values for different positions of line from the sensor. The calibration process recorded the sensor readings ranging the distance between the line and sensor from -3.5 cm to +3.5 cm with a resolution of 0.5 cm. This data was then used to find a polynomial curve that took input as the analog sensor read and returned the estimated distance from the sensor. Due to high non linearity and discontinuity of the sensor characteristics, we broke the range into 3 sections and each section has its own respective polynomial fitting. Combining the estimates from of the position of the line from all five sensors using Kalman filter gave us the final estimate of the line which serves as the error to PID controller. The goal for this method is to input a more accurate error to control algorithm so that the output will be more streamlined reducing the non essential jerks in the steering action.

The basic implementation of this approach is done, however we observed that the response time of the estimated position was slow when there were fast changes in the position of the line. For instance when there are angular junctions of the line as opposed to smooth curves, the estimated value took a long time to come close to the actual value. This is due to the fact that Kalman filter takes into account all the previous estimations and to overturn the effects of these estimates, it takes some time when there is a sudden change in the estimates. To tackle this, we tried tuning the PID algorithm by introducing Ki and Kd gains. However this approach worked for slower speeds of the car up to 50

6 Proposed Cascade Control

The previous approaches are focused on line following applications and the forward velocity of the car is kept constant throughout the run. This may not be ideal in real life situations as usually the speed of the car is reduced while turning. We decided to tackle this real life scenario by introducing an IMU sensor to the PiCar ecosystem. The idea is to generate an estimate of velocity from the gyroscope and accelerometer of the IMU sensor. We then propose a cascade feedback loop as shown in the fig 5. The outer loop senses the position of the line and controls the steering of the car. The output of this controller is then proportionally scaled to generate the velocity set point. This approach intuitively makes sense as higher error valued correspond to higher steering corrections which directly translates to sharp turns. Hence the set point of velocity is directly proportional to the error of the line following controller.

The inner loop of the cascade control receives the set point from the outer loop and takes the feedback of IMU which estimated the velocity. The velocity controller then runs its own PID controller that output the required PWM to the motors. We generate same PWM to both of the rear motors negating the effects of differential drive. In the future we plan to compute the differential velocities for each of the motor to avoid slipping and skidding of the wheels.

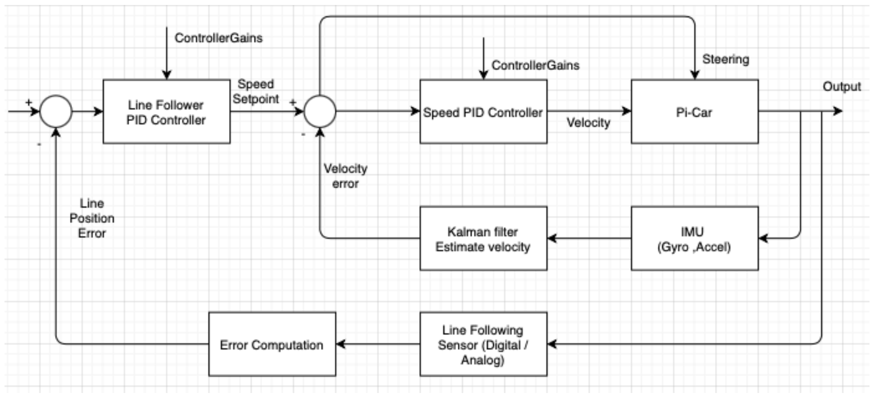

Figure 5: Cascade Feedback Loop for PID Controller

6.1 Velocity Estiation using IMU

The 9DOF IMU consists of 3 axis accelerometer, 3 axis gyroscope and 3 axis magnetometer. To estimate the linear velocity in X and Y dimension, we use the gyroscope and accelerometer data. We do not focus on magnetometer as the experimental setup is indoors and magnetometer readings are unreliable in an indoor setting without proper calibration.

The initial goal is to compute the sensor orientation with respect the earth. This can be achieved by finding the roll, pitch and yaw angles. Ideally the roll and pitch should be zero if the sensor is mounted perfectly parallel to the ground. The yaw angle helps us compute the velocity components in X and Y direction. We compute roll and pitch using complementary filter on accelerometer and gyroscope values. Value of α in equation 4 is take to be 0.04. The roll and pitch are computed with gyroscope and accelerometer readings for X and Y as respectively

$$angle = (1 - \alpha) * (angle + gyro * dt) + \alpha * accl \tag{4}$$

We need roll and pitch to transform the accelerations sensed by the IMU in sensor frame to be translated the acceleration in the earth frame. We are interested in the linear velocity with respect to earth and not with respect to the sensor.

Once we compute the roll, pitch and yaw angles, we compute a rotational matrix to transform 3 axis accelerometer values from the sensor frame to the earth frame as shown in equation 5. Then we accumulate these values by multiplying it with delta time for each loop iteration. This is used to approximate the integration operation. These accumulated values are an estimation of velocity in the X, Y and Z dimension as shown in equation 6.

$$a_m = a_B - R_I^B \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \implies a_I = R_B^I a_m + \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \tag{5}$$

where a_m acceleration measurement vector
a_B body frame acceleration vector
a_I inertial body frame acceleration
R_I^B rotational matrix from inertial frame to body frame of the sensor.

$$v_I[k+1] = v_I[k] + dt * a_I[k] \tag{6}$$

However we observed the velocity estimates were off by up to 1m/s after a 15 second run on the track. This drift is high for an indoor experiment setup. We are still working on methods to improve the velocity estimate to successfully execute the cascade feedback control system.

7 Conclusion

The project has successfully explored various control algorithms for a car like robot while taking into account the real life constraints and problems. All the algorithms were developed by keeping in mind the UberSpark framework and the code base is uberObject friendly. This will ensure the security of the sensor data while exploring real life control strategies.

The simple line following using a PID controller has been implemented in C and a shared library has been generated that can be called from any high level languages like Python. This shared library will handle all the security features and the interactions with UberSpark framework. We also the analyse the overheads imposed by the framework on the control algorithms. The comparison details are thoroughly mentioned in the AUTOBOT-X research paper.

8 Future Work

We continue to explore and perfect more sophisticated control algorithms that enable us to generate more accurate error for the position of the line and compute a stable estimate for the velocity to successfully implement the cascade controller. Moreover we explore other sensors like the ultrasonic sensor to implement failsafe state when the car experiences an obstacle. The underlying goal in developing these control algorithms and interfacing multiple sensors is to successfully build uberObjects around these applications that will guarantee security and protect the sensor data from intruders. The final aspect is to protect the data sent to the actuators as well to develop an end-to-end system offering all the security features provided by UberSpark.

9 Application Details

The GitHub repository houses all the latest code developed in this project. The PIDshared folder consist the code base related to the PID controller using the simple error approach. The PIDKalman folder has the current version of the algorithm that aims to find the exact position of the line while IMUvelocity folder has the algorithm for velocity estimate and the cascade control system. Each of these folders have a detailed README that describes the installation process and the usage of these files.

10 References

1. Amit Vasudevan, Sagar Chaki, Petros Maniatis, Limin Jia and Anupam Datta: überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor (2016)
2. Anton Hristozov, Amit Vasudevan, Bruce Krogh, Raffaele Romagnoli, Ruben Martins, Atharv Sathe and Ethan Joseph: AUTOBOT-X*: A Low-cost, Extensible Security Platform for Robotics (2021)
3. UberSpark Repository: `https://github.com/uberspark/uapp-SunFounder_PiCar-S`
4. Simulator Repository: `https://github.com/athens2000/piCarSimulator`
5. Raspberry Pi Model 3B: `https://www.raspberrypi.org/documentation/hardware/computemodule/datasheets/rpi_DATA_CM3plus_1p0.pdf`
6. Raspberry Pi Car Repository: `https://github.com/sunfounder/SunFounder_PiCar-S`
7. MPU9250 datasheet: `https://invensense.tdk.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf`