Autonomous driving using an RC car with UWB positioning

Torun Nicander & Jakob Nyberg

November 8, 2020

Abstract

In this project we aimed to construct a system where an RC car can drive autonomously and be localized using ultra wideband signals. For the ultra wideband localization system we used Decawave's MDEK1001 development kit.

The result of the project is a car connected to a Raspberry Pi that can drive along a user-specified path using the pure pursuit algorithm. The software we created is built upon Python's `asyncio` library. It collects measurements from the localization system and an IMU and filters these using a Kalman filter. We also created a browser-based web client that that can visualize the state of the system and allow the user to specify a destination for the car.

Contents

1 Introduction

Autonomous vehicles have the potential to be very useful for society; as self-driving cars that may one day drive better than humans, or as robots that traverse dangerous environments without risking human life. The challenges faced with when creating an autonomous vehicle are many: How to localize it? How to track it? How to control it?

For cars; GPS, lidar and inertial measurement units are typical solutions for positioning. In an indoor environment however, GPS is not an ideal choice because of its accuracy. Ultra wideband technology is promising for localization purposes in indoor environments as it provides low latency positioning at low costs and higher accuracy than GPS.

Ultra wideband hardware has received increasingly more attention in recent years, and are slowly entering the market. The iPhone 11 which was released in 2019 included an ultra wideband chip for "spatial awareness" [1]. This could allow the phone to, for example, automatically unlock a door when it is close to it.

Although UWB as a technology has existed for a number of years, recent changes to regulations and new standards enable systems to utilize its unique capabilities without interfering with existing wireless technologies. Accessible hardware is also an important factor in its growing popularity, with chips like Decawave's DW1000 providing a cheap all-in-one piece of hardware for UWB communication.

Positioning the vehicle is only one part of an autonomous vehicle system. The vehicle has to plan its motion in some way, and then follow up on that plan with some level of accuracy. This involves the automatic control of the steering and motors in order to maintain a set course.

In this project we try to construct a self-driving car from a RC car. We use UWB positioning to localize the car in an indoor environment, and have it drive along a user-specified path.

2 Objectives

This project is a continuation of a project from the course *Open Advanced Course in Embedded Systems*.

The objective of this project is to make an RC car drive itself along a path generated by user specified points (coordinates) in an indoor environment, making use of Ultra Wideband (UWB) for location estimation and tracking. The user will use a browser based interface to specify destinations, and will have the ability to stop the car and also drive it manually from the interface.

3 Theory

Here we present theoretical concepts that are relevant to our project.

3.1 Ultra wideband (UWB)

Ultra wideband signals are commonly defined as signals that have a fractional bandwidth (B_{frac}) larger than 20% or an absolute bandwidth (B) of at least 500 MHz [18].

The absolute bandwidth, or -10 dB bandwidth, is defined as

$$B = f_H - f_L, \tag{1}$$

where f_H is the upper frequency of the -10dB emission point, and f_L is the lower frequency of the -10dB point [18, p. 45].

The fractional bandwidth is calculated by

$$B_{\text{frac}} = \frac{2(f_H - f_L)}{f_H + f_L}. \tag{2}$$

A commonly used technology for creating UWB signals is impulse radio (IR). With IR, very short duration waveforms are created and sent in a sequence called a *pulse train*. There are different ways of designing the waveforms, such as Gaussian pulse shapes or Hermite polynomials.

A number of pulses will code for an information symbol. The pulses are modulated and code for information in different ways. The information encoding can for example be the polarities, the phase or the amplitudes of the pulses. The pulses are sent in a specific sequence to avoid collisions with other UWB users in the same area, and this method is called time hopping [18, p. 21].

The UWB signals do not only share the spectrum with other UWB signals however as the UWB spectrum is so large as to potentially obstruct the signals of other wireless technologies as well. UWB may interfere with IEEE 802.11 (Wi-Fi), GPS, or Bluetooth technology. Because of this, UWB is regulated by the FCC to have a power spectral density below -41.3 dBm/MHz in the spectrum 3.1 GHz to 10.6 GHz, and in other ranges even less [18, p. 25]. Other regional agencies in Europe and China have imposed similar regulations.

To illustrate the suitability of UWB for transmissions, look at the Shannon-Hartley Capacity Theorem:

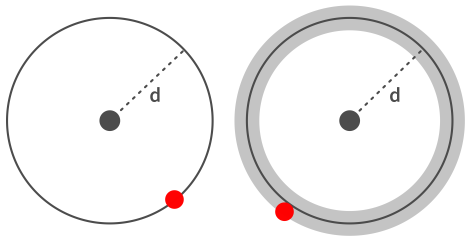


Figure 1: Estimating the distance to between the target and the reference using RSSI or TOA results a circle of possible locations. The left image shows the position estimation with no noise and the right with noise added, creating an uncertainty region.

$$C = B \log_2(1 + S/N), \tag{3}$$

where C is the channel capacity (bits/s), B is the bandwidth, and S/N is the signal divided by the noise, also known as the signal-to-noise ratio (SNR). With a fixed bandwidth, the way to achieve a higher channel capacity would be to increase to power of the signal. However, in UWB we can have a very high bandwidth, which means the signal power can be kept low while still maintaining a high channel capacity.

3.2 Wireless Localization

Wireless localization is the problem of estimating the position of a target node, with an unknown location, based on a number of reference nodes, whose location are known. Localization can be done by a number of different methods depending on the measurements available.

3.2.1 Distance estimation

The distance between a reference and a target with an unknown position can be estimated using the received signal strength (RSSI). Ideally the function describing the the power loss over distance is

$$\bar{P}(d) = P_0 - 10n \log_{10}(d/d_0), \tag{4}$$

but due to noise such as fading the power falloff typically behaves more like a normal distribution with mean $\bar{P}(d)$ [17, p. 65].

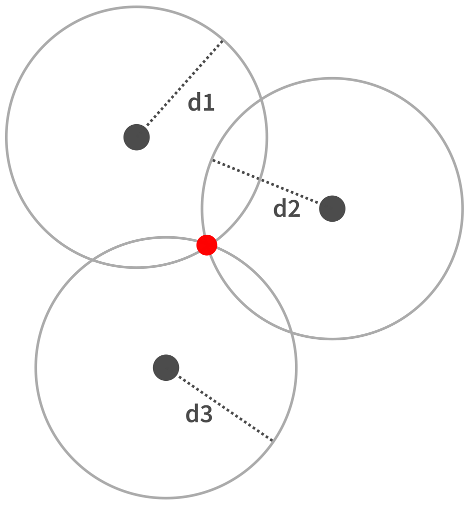


Figure 2: Trilateration. The distance measurements from the target point (red) to the reference points (grey) form three circles. The position of the target can be estimated by calculating the intersection of the three circles.

Time of arrival (TOA) is another method for distance estimation that uses the time difference between the sender sending a message and the receiver receiving it to estimate the distance. TOA requires synchronization between the nodes in order to correctly estimate the time difference [17, pp. 70–71].

3.2.2 Positioning

Both RSSI and TOA results give position estimates in the form of a circle, with radius equal to the distance between the target and the reference, as seen in Figure 1. Trilateration uses the intersection between three such circles to calculate the position of a target point, as shown in Figure 2 [17, p. 76].

If the transceiver supports it, the angle of arrival of the signal can be used. The position of the target can then be estimated using triangulation [17, p. 67].

Trilateration and triangulation are both geometric methods and do not take measurement noise into account. As such, they become less effective when noise is present since the equations may not have solutions [17, p. 79]. A better choice in a noisy system can be position estimation using machine learning such as k-NN or neural networks or probabilistic methods such as maximum likelihood estimation. The inputs of such methods are parameters such as RSSI or TOA and the output is the position of the target. By measuring the parameters at known locations a model can hopefully be trained to estimate the correct position [17, pp. 74–75].

3.2.3 Position Tracking

When the target to be localized is in motion, the task of localizing it can be described as a state space problem. The inputs to the problem are the distance measurements and the state is the location of the target. A Kalman filter can be used to reduce the effects of measurement noise and produce a smooth estimate of the position of the moving target [17, pp. 93–94].

3.3 Kalman Filters & State-space Models

A Kalman filter makes use of a state-space model to recursively do a state estimation. For each time step/update it uses the previous state and new input data (measurements) to estimate the new state. It is computationally efficient as it has no need to recompute an estimator for each time step, and does not need to store all previous measurements. The Kalman filter is very suited for tracking a moving target [8].

A general state-space model with state x and measurements z can be described as such:

$$x_{k+1} = F_k x_k + B u_k + w \tag{5}$$

$$z_k = H_k x_k + v. \tag{6}$$

F is a matrix that describes the process evolution, and B describes how the control signal u affects the model. Process noise w is assumed to be white noise with covariance Q, and the measurement noise v is assumed to be white noise with covariance R.

The procedure of implementing a Kalman filter is to first initialize the filter with values for a prior state. Then measurements are collected, the model is updated with these measurements (which is called the measurement update and is where the estimate for the current time step along with its mean square error is calculated) and finally a prediction is made (time update). In the prediction, the estimate for the next time step and its MSE is calculated.

3.3.1 State space models of moving object

[16] presented a simple state space for estimating movement in x- and y-directions. The state space variables used were x and y for the position in the x- and y-directions, and $\dot{x}, \dot{y}$ for the velocities in the x- and y-directions. This model

assumes constant velocity:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \dot{x}_{k+1} \\ \dot{y}_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \\ \dot{x}_k \\ \dot{y}_k \end{bmatrix} \tag{7}$$

Acceleration is modeled as a random addition component to the velocities in the process noise.

$$Q = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \Delta t^2 & 0 \\ 0 & 0 & 0 & \Delta t^2 \end{bmatrix} \tag{8}$$

Finally, the H matrix looks as such:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \tag{9}$$

and the measurement noise:

$$R = \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix} \tag{10}$$

3.3.2 State-space of a moving vehicle

A process model more specifically suited for the task of estimating a moving car is one that models a moving vehicle by incorporating acceleration and angular velocity with the positioning.

This model was based on [3] but modifications were made, as we have different sensors and available measurements than the article. The article makes use of an Extended Kalman filter as their state space involves cosine terms. We were able to use a linear Kalman filter as our acceleration measurements from the IMU were given in both the car's coordinate system and a world coordinate system. By using the world coordinate system, we did not need to project the acceleration and velocity components to the x- and y-plane, so no cosine terms were needed in the state-space model. Presumably, the IMU transforms the acceleration components to the world coordinate system internally using its estimated heading.

The movement in x- and y-directions can be described with these mechanics formulas which were based on formulas in [3]:

$$x_{k+1} = x_k + \dot{x}_k \Delta t + \frac{1}{2} \Delta t^2 \ddot{x}_k \tag{11}$$

$$y_{y+1} = y_k + \dot{y}_k \Delta t + \frac{1}{2} \Delta t^2 \ddot{y}_k \tag{12}$$

$$\dot{x}_{k+1} = \dot{x}_k + \Delta t \ddot{x} \tag{13}$$

$$\dot{y}_{k+1} = \dot{y}_k + \Delta t \ddot{y} \tag{14}$$

The accelerations will be considered constant with noise terms:

$$\ddot{x}_{k+1} = \ddot{x}_k \tag{15}$$

$$\ddot{y}_{k+1} = \ddot{y}_k \tag{16}$$

The car's heading is described as such,

$$\theta_{k+1} = \theta_k + \dot{\theta}_k \Delta t \tag{17}$$

and the angular velocity is modeled after how the the wheel angle and the angular velocity of a car are related in Ackermann Steering, as described by [15]:

$$\dot{\theta}_{k+1} = \frac{s}{L} \phi, \tag{18}$$

where s is the speed of the car, L is the length of the car, and ϕ is the control signal for the steering.

As we don't track the speed, and the calculation of it is nonlinear ($\sqrt{x_{k+1}^2 + y_{k+1}^2}$), we set the fraction as a constant term c instead.

$$\dot{\theta}_{k+1} = c\phi. \tag{19}$$

This gives us the following state variables:

$$x = \begin{bmatrix} x \\ y \\ \theta \\ \dot{\theta} \\ \dot{x} \\ \dot{y} \\ \ddot{x} \\ \ddot{y} \end{bmatrix} \tag{20}$$

With F, B and u as:

$$F = \begin{bmatrix} 1 & 0 & 0 & 0 & \Delta t & 0 & \frac{1}{2}\Delta t^2 & 0 \\ 0 & 1 & 0 & 0 & 0 & \Delta t & 0 & \frac{1}{2}\Delta t^2 \\ 0 & 0 & 1 & \Delta t & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{21}$$

$$B = \begin{bmatrix} 0 \\ 0 \\ 0 \\ c \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, u = \begin{bmatrix} \phi \end{bmatrix}. \tag{22}$$

The measurements are position in x and y-dimensions, the heading, and the acceleration in x and y-dimensions.

$$z = \begin{bmatrix} x \\ y \\ \theta \\ \ddot{x} \\ \ddot{y} \end{bmatrix} \tag{23}$$

and the corresponding H matrix is

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{24}$$

The process covariance matrix is modeled as

$$Q = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma^2_{w,\dot{\theta}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma^2_{w,\dot{x}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma^2_{w,\dot{y}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \sigma^2_{w,\ddot{x}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \sigma^2_{w,\ddot{y}} \end{bmatrix}. \tag{25}$$

The measurement covariance matrix as:

$$R = \begin{bmatrix} \sigma^2_{v,x} & 0 & 0 & 0 & 0 \\ 0 & \sigma^2_{v,y} & 0 & 0 & 0 \\ 0 & 0 & \sigma^2_{v,\theta} & 0 & 0 \\ 0 & 0 & 0 & \sigma^2_{v,\ddot{x}} & 0 \\ 0 & 0 & 0 & 0 & \sigma^2_{v,\ddot{y}} \end{bmatrix} \tag{26}$$

where the $\sigma^2_{w,i}$ values represent the noise of the process, and $\sigma^2_{v,i}$ represents the noise of the measurements.

This Kalman filter makes use of measurements from two different sensor systems. Combining measurement data from different sensor sources in this way into for example a Kalman filter is called sensor fusion.

3.3.3 Extended Kalman Filter

The Kalman filter does not perform well for nonlinear state-space models as it is designed for linear relationships.

Instead, one can use the Extended Kalman filter that handles the nonlinearities by linearising them around the current state estimate.

A nonlinear state-space model is given by

$$x_{k+1} = f(x_k, u) + w \tag{27}$$
$$y_k = h(x_k) + v, \tag{28}$$

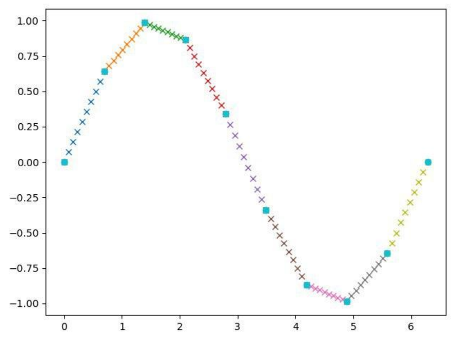


Figure 3: Bresenham's line drawing algorithm being used to create line segments between destination points.

where $a(\cdot)$ and $h(\cdot)$ are nonlinear functions and w and v are white noise. This filter has no optimality property [12].

3.4 Motion planning and path tracking

While deciding on a destination for a vehicle is easy, reaching said destination can be decidedly harder. The straight path to the destination is not always viable, but even if the straight path is chosen it might not be possible for the vehicle to move in such a way. For example, a car has to turn in order to go in different directions, as opposed to a drone that can fly in all directions. The problem of how to reach a destination can be divided into two sub-problems, path planning and path tracking.

3.4.1 Path Planning

Path planning is the problem of plotting a path for a vehicle to follow, or setting a destination and finding the shortest path to that destination. This is often done using traditional pathfinding algorithms, such as Dijkstra's algorithm or the A* algorithm [13]. The algorithms may be modified to take into account factors such as a path curvature limits. In our case, we do not use any path finding algorithms but pick the intended path manually by selecting a number of destinations in the form of (x,y) coordinates.

Rather than having the vehicle move to a discrete sequence of destinations, it is generally preferable to have the vehicle follow some interpolated path. A relatively simple method to achieve this is Bresenham's line drawing algorithm [4] to interpolate between the destinations, an example can be seen in Figure 3. This produces straight lines drawn between the points, but if a continuous line is desired then cubic splines may be used instead.

After generating a path for the car to follow, an algorithm for making the car follow said path is needed. A variety of control algorithms for this purpose have been proposed [13], and in this project we use the Pure Pursuit algorithm.

3.4.2 Path Tracking using a Pure Pursuit Controller

Pure Pursuit is a path tracking algorithm that has been used for various forms of autonomous vehicles since the 1990's [5]. The tracking works by having the vehicle drive toward a point on the reference path that is always a certain distance, L, from the vehicle.

The point to drive toward can be calculated algorithmically by finding the path point that is closest to the car and then searching through the path until the first point where the distance is greater than L. The angle error α between the vehicle's heading angle and the angle towards the reference is given by Equation 29, where $(x_{\text{ref}}, y_{\text{ref}})$ is the reference point.

Geometrically this can be seen as fitting a semi-circle between the car and the reference point that the car will follow. For large values of L the controller will be stable but slow, and for low values it will be increasingly responsive but less stable. An illustration of how the pure pursuit controller works can be seen in Figure 4.

$$\alpha = \arctan\left(\frac{y_{\text{ref}} - y}{x_{\text{ref}} - x}\right) - \theta \tag{29}$$

3.5 PID-controller

A PID-controller is a linear control system with three parts, a proportional term, a derivative term and an integrating term.

$$u(t) = K_p e + K_d \dot{e} + \sum_i e_i \Delta t \tag{30}$$

The car has two systems that need automatic control, the forward velocity and the steering. We treat these as separate systems, controlled independently. The steering control uses the angle error from the pure pursuit controller as error input, and the output is the angle to set the wheels to. The velocity controller is set to maintain a speed of 0.5 m/s, but the control signal is added to a base speed of roughly 0.4 m/s. This is to prevent the car from stopping when the control error goes to 0.

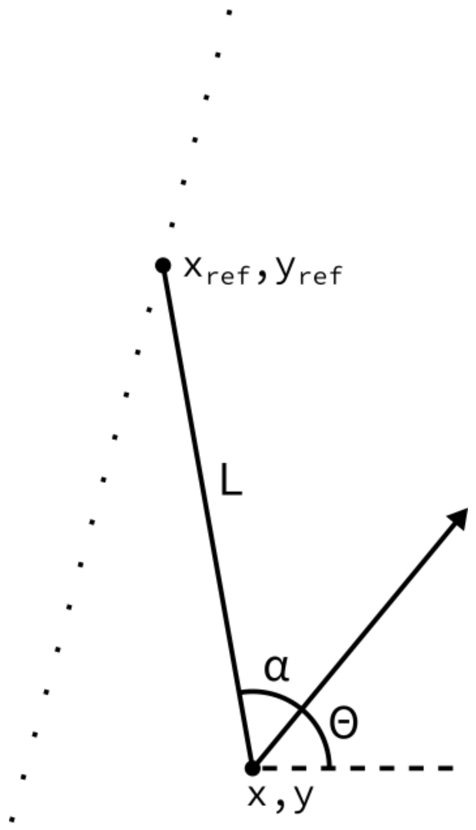


Figure 4: Visualization of the pure pursuit algorithm. The car has a position x, y and a heading angle θ. x_{ref}, y_{ref} is a point on the path with distance L from the car's current position. α is the heading error, the difference between the heading angle and the angle of the vector $(x_{ref} - x, y_{ref} - y)$.

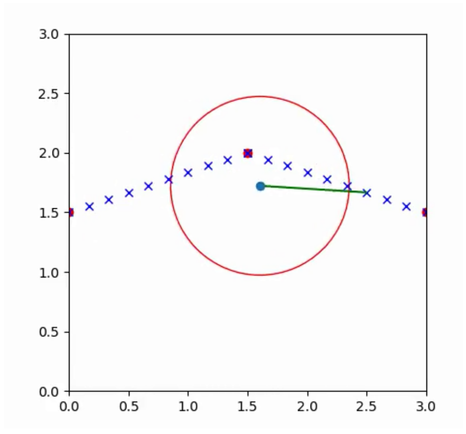


Figure 5: Simulation of pure pursuit algorithm. The filled red circles are reference points and the blue crosses are the interpolated path. The blue circle represents the vehicle, and the red circle surrounding it is the lookahead distance, L. The green line indicates the current reference point.

3.6 Concurrency with `asyncio`

The Python standard library has multiple libraries for handling concurrency. One of these is the `asyncio` library. `asyncio` enables the use of coroutines, callbacks and other components for asynchronous concurrency in Python [2]. A coroutine is a function that can "pause" its execution while waiting for a condition, enabling other scheduled coroutines to run. Asynchronous concurrency is often used in I/O-heavy programs and network interfaces, as these spend a lot of their running time waiting for new input. `asyncio` only uses a single thread, and thus only one task can ever run. For tasks that block execution for a long time before suspending, using another thread or multiprocessing can be preferable to a coroutine.

`asyncio` has ready classes for task synchronization. These include FIFO queues, semaphores and events. The Event class represent flags that can be used to signal that something has happened in the program. Tasks can suspend their execution until the event is set. Events are used in our software to signal that new measurements, estimated states and control signals are available, ensuring that tasks run in the desired order.

4 Implementation

4.1 Material

The following hardware was used.

- Raspberry Pi 3 Model B
- Arduino Nano
- MPU6050 IMU
- Reely Cyclone Brushed Motor RC Car
- Reely 40A Brushed Motor Speed Controller
- Decawave MDEK1001 Development Kit
- Asus Wireless Router + Mobile broadband, Modem for internet access.
- Android device with Decawave's RTLS application installed
- Laptop with internet access

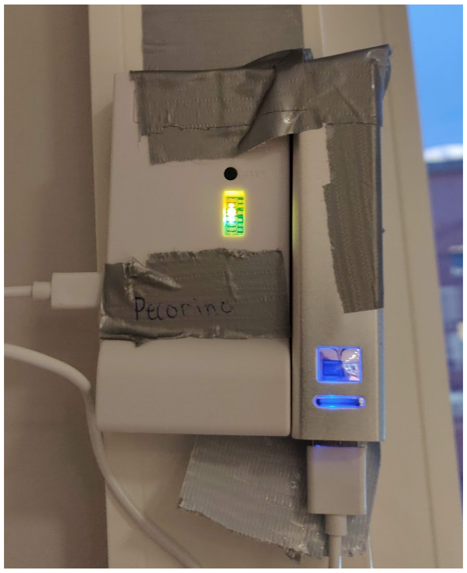


Figure 6: Anchor node on wall with power bank attached.

4.2 System overview

A representation of the system can be seen in Figure 9. A real time localization system was set up in a room, with four stationary nodes (anchors) and one moving node (tag). The tag calculates its position by communicating with the anchors via UWB and using time of arrival (TOA), and then forwards the position estimation to the Raspberry Pi. The tag was attached to a Raspberry Pi via GPIO pins. We use the real-time localization system as-is, using Decawaves's Android application for setup and calibration. We also use Decawave's firmware that is running on the nodes.

The car we are using is a Reely Cyclone Brushed Motor RC Car, with a retrofitted speed controller. The original speed controller was replaced as it did not have any PWM input ports. The speed controller serves as an interface, allowing different types of batteries, receivers and motors to be used together. It powers both the motor and receiver unit. The motor controller can be treated as an RC servo, with forward and backwards gas controlled by PWM signals from a receiver. In our case, the receiver is a Raspberry Pi. The Raspberry Pi is also connected to an Arduino Nano through USB, and the Arduino communicates with the IMU. From the IMU we read the pitch, yaw and roll angles, as well as the acceleration. These components are all mounted on top of the car in a plastic box. The car with additional hardware mounted on it can be seen in Figure 8.

4.2.1 Inertial Measurement Unit (IMU)

We use a MPU6050 IMU, which is a 6-DOF IMU with an accelerometer and gyroscope. It measures acceleration in g:s and rotation in degrees per second. Through firmware it can also estimate the yaw, pitch and roll of the device, something usually measured using a magnetometer.

4.2.2 Decawave

In this project, Decawave's MDEK1001 development kit is used for UWB localization. The included firmware used TOA distance measurements to estimate the position of a specified node.

The MDEK1001 development kit consists of twelve DWM1001-DEV boards in plastic cases, which are also called nodes in the real time location system. The DWM1001-DEV board mainly functions as a interface board for the DWM1001, providing connections and a J-Link debugger chip. The DWM1001 module has a DW1000 UWB transceiver, antennas for UWB and Bluetooth communication, a motion sensor and a nRF52832 microcontroller.

The development board is pre-loaded with Decawave's own firmware that run on the nRF52832 mounted on DWM1001. The firmware is based on the eCos real-time operating system and handles the real-time location system. The firmware provides an API that the user can interface with externally using UART, SPI or Bluetooth [7]. Decawave also provides the files necessary for recompiling the firmware with custom tasks and functions defined in order to perform low-level operations.

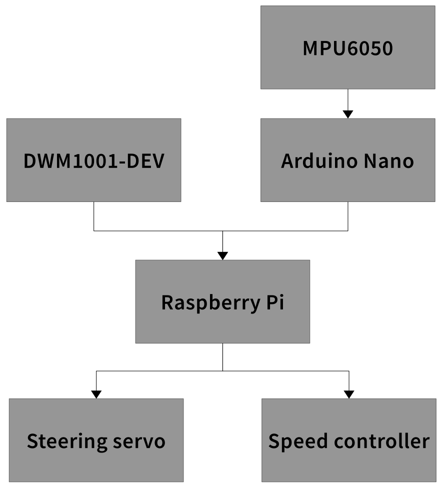


Figure 7: Systematic representation of the hardware connections.

The Raspberry Pi hosts a web interface via Wi-Fi, where the user can enter destination points for the car to drive to, and also take manual control of the car.

4.3 Software

The software running on the Raspberry Pi is written in Python 3.7, the most recent version of Python at the time of the start of the project. For mathematical functions we use the package `numpy`. We use `pandas` for data frames [11] and `matplotlib` for plots [9].

The program consists of a number of tasks that run with asynchronous concurrency. An illustration of the different tasks, and their relations, can be seen in Figure 10.

Serial task handles the connection to the tag and Arduino. Each loop iteration

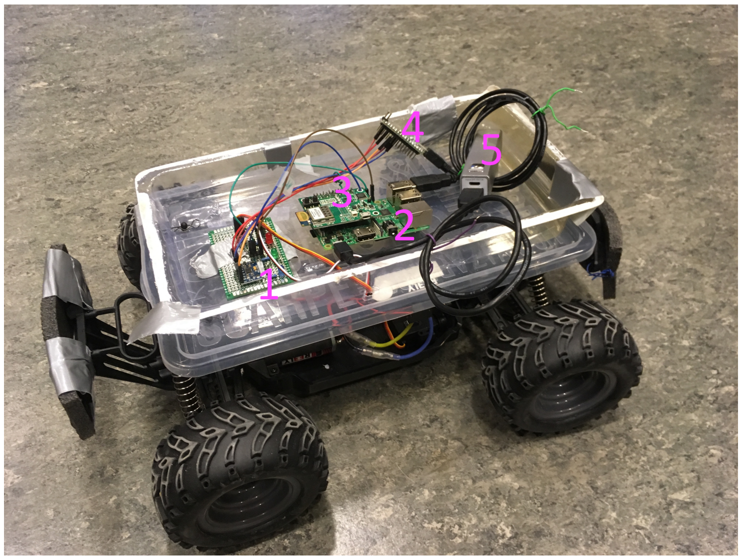


Figure 8: Car with attached hardware. 1. IMU/MPU6050. 2. Raspberry Pi. 3. DWM1001-dev board. 4. Arduino Nano. 5. Powerbank.

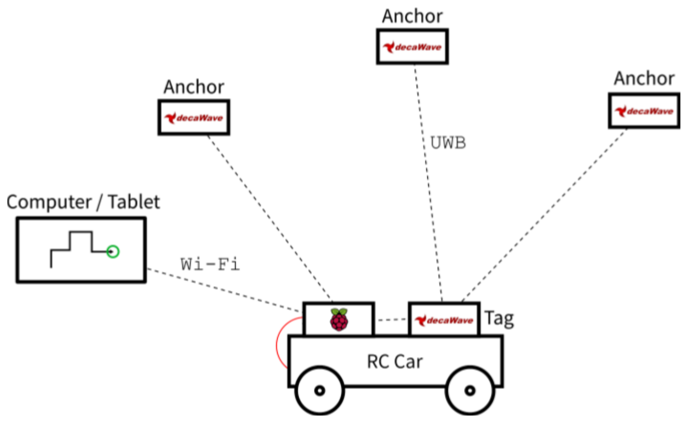


Figure 9: Illustration of the intended system.

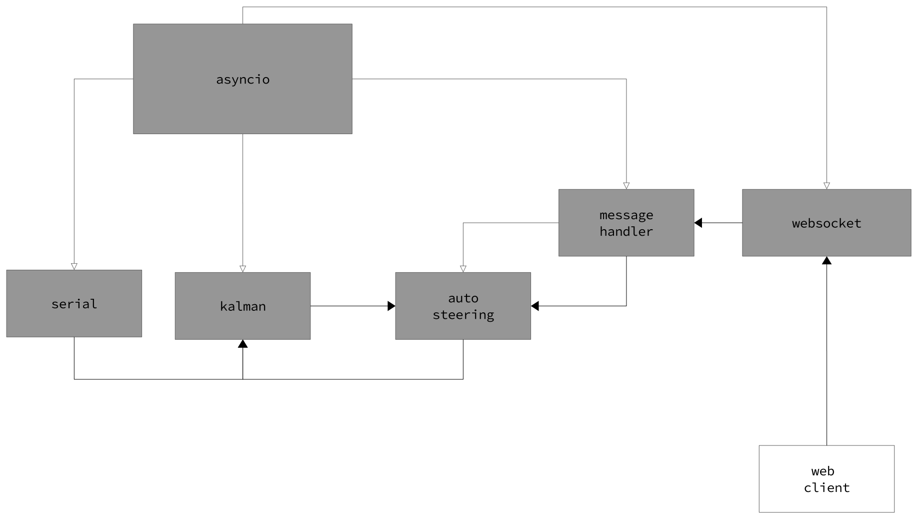


Figure 10: Tasks that run asynchronously during program execution. Grey arrows indicate parenthood. Black arrows indicate data flow.

consists of waiting for measurements from the tag and IMU, which are obtained from two respective functions. Both these functions will block the main thread while waiting for values from the serial interface, preventing other coroutines from running. Because of this, the two retrieval tasks run in separate threads. PySerial handles serial connections.

The program running on the Arduino Nano is a slightly modified version of an example program from the I2CDev library. The program waits for any UART event, at which it will respond with IMU data in a comma separated format. At startup, the IMU is calibrated and will set the 0° angle. The IMU scale ranges were set to ± 250 deg/s and ± 8 g for the gyroscope and accelerometer respectively.

The Kalman task handles Kalman filtering, described in more detail in subsection 4.5. For each timestep, the task awaits measurements from the serial task and a control signal from the steering task, if steering is enabled. When new values are available, it performs the filter operations and saves a new estimated state.

The websocket task handles the connection to the web client, sending and receiving data. The web client can send messages containing various orders to control the program, and the program will send data to the web client. The websocket task uses the `websocket` library, which is built upon the `asyncio` library.

The message handler task processes incoming messages, received from the websocket task. It is a utility task that handles manual control of the car as well as starting the auto steering task when prompted.

The auto steering task handles automatic control of the car. It awaits estimated states from the Kalman task and uses these to steer the car. The control signal is calculated using the pure pursuit algorithm and a PID-controller, as shown in Figure 11, and sent to the steering servo.

The path is defined using a set of checkpoints, supplied at the start of the program. The path is generated using Bresenham's line drawing algorithm based on these checkpoints. The auto steering task keeps track of all the checkpoints that have been passed within a certain distance threshold. Once all the checkpoints have been passed, the car will stop at the final checkpoint. The auto steering task also handles logging of data, used for creating plots after the run has been completed.

The steering servo and the speed controller are both controlled using PWM, produced with the Raspberry Pi's hardware timers. These are controlled using the program

`pigpiod`, which in turn is controlled using the Python library `GPIO Zero`. The program has a shared object called `Context`. The `Context` class contains general application data such as program settings. It also contains shared variables for

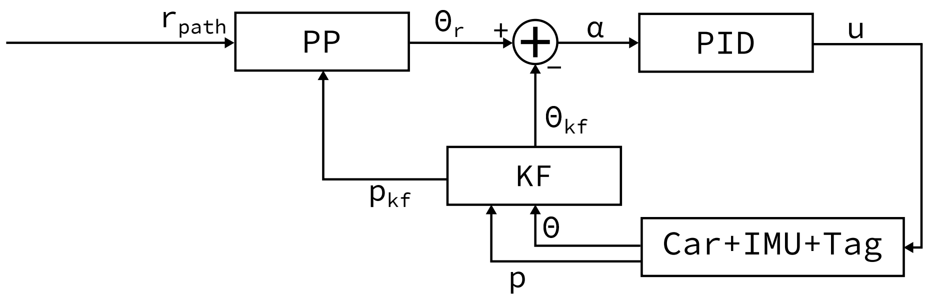


Figure 11: Block schematic of the automatic control loop. r_{path} is a vector of points that represent the path. The PP block represents the pure pursuit controller. p is the position of the car, represented as x and y coordinates. The angle error is passed through a PID controller, which is then used to set the wheel angle of the car. The IMU and tag will then give a new position and angle to be used in the next control iteration.

task communication.

4.3.1 Web interface

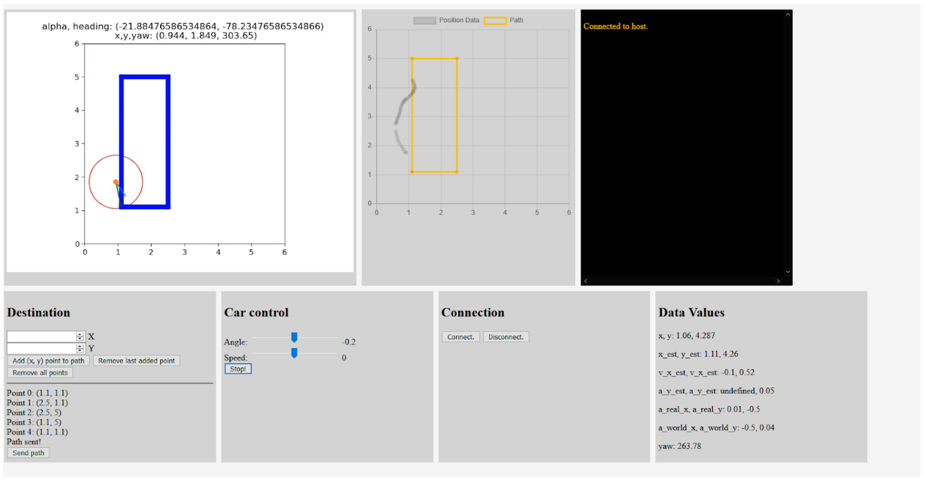
The web server runs in a separate thread. It consists of an HTTP server that serves the web client to incoming connections. The web client is built using HTML and JavaScript. A screenshot of the web interface can be seen in Figure 12.

4.3.2 Usage Example for Following Rectangular Path

The car is placed in the direction of the positive x-axis. The Raspberry Pi and speed controller are turned on. The software is launched with the web interface using the command:

```
main.py -s [PATH TO SETTINGS FILE] gui [IP ADDRESS OF PI]
```

The web interface can then be accessed by going to the specified IP address with a browser. The interface is hosted on port 8080. Five path points can then be entered in the destination section. Five points are necessary to create a complete path, where the first and last points are the same position. The path tracking is started by pressing send path. After the car has reached the final path point, it will render an animation of the trajectory which is shown on the web interface. The same process can also be automated by running the subcommand `collect data`, where the path is entered in the settings file. A sample settings file can be seen in Appendix B.



Connected to host.

Point 0: (1.1, 1.1)
Point 1: (2.5, 1.1)
Point 2: (2.5, 5)
Point 3: (1.1, 5)
Point 4: (1.1, 1.1)
Path sent!

Angle: -0.2
Speed: 0

x, y: 1.06, 4.287
x_est, y_est: 1.11, 4.26
v_x_est, v_x_est: -0.1, 0.52
a_y_est, a_y_est: undefined, 0.05
a_real_x, a_real_y: 0.01, -0.5
a_world_x, a_world_y: -0.5, 0.04
yaw: 263.78

Figure 12: Screenshot of browser based interface. Panels in order from top left shows: Animation of car path, real time location of car and path, status messages, path point input, manual car control, connection manager and raw data values.

```
main.py -s [PATH TO SETTINGS FILE] collect-data --sleep-time [DURATION OF RUN]
```

4.4 Collecting Location Data

By using the Python package pySerial [14] we were able to perform serial communication with the DWM1001-DEV. The API Guide provided by Decawave [6] lists available TLV commands (Type-Length-Value) that can be sent via UART.

We used the function `dwm_loc_get`, represented as TLV values as `0x0C 0x00`, to get continuous location estimates from the network. The response from the DWM1001-DEV is a location estimate in x, y and z-dimensions, as well as a quality measure of the provided location estimate that ranges from 0-100 percent.

4.5 Kalman Filter

We implemented the two Kalman filters described in subsubsection 3.3.1 and 3.3.2 using the class `KalmanFilter` from the python library `filterpy` [10], which uses `numpy` for its matrix calculations.

4.5.1 Kalman Filter with RTLS measurements

The filter described in this section is defined in subsubsection 3.3.1. This filter uses the the x and y positions from Decawave's RTLS for its measurements.

The collection of measurement data takes a different, albeit small, amount of variant time each update. We measure the time for each serial read and store as a dt variable. We then update the F matrix with this value for each update, which makes our Kalman filter time variant. We have a toggle to turn off the variable dt, if one wants to have it update with a static timestep.

In the case we for a time update do not receive new values (for example if the serial read took too long and we skip it), we use the last estimates but set the R with high values (500), which makes the Kalman filter prefer the process model in the update.

Good parameter values for R and Q were found through trial-and-error.

4.5.2 Kalman Filter with RTLS and IMU sensor fusion

We implemented the Kalman filter described in subsubsection 3.3.2. It uses measurements from both the UWB-based RTLS and the acceleration in two dimensions from the IMU as well as the car's heading.

As we could make this filter linear by only neglecting one nonlinear term (the speed value that should be multiplied with ϕ, see Equation 18), we opted to do that instead of working with the Extended Kalman filter. We replaced $\frac{s}{L}$ with a constant c, which we approximated as 0.9.

Good parameter values for R and Q were again found through trial-and-error. We compared the two filters using the same values, see Results and Discussion.

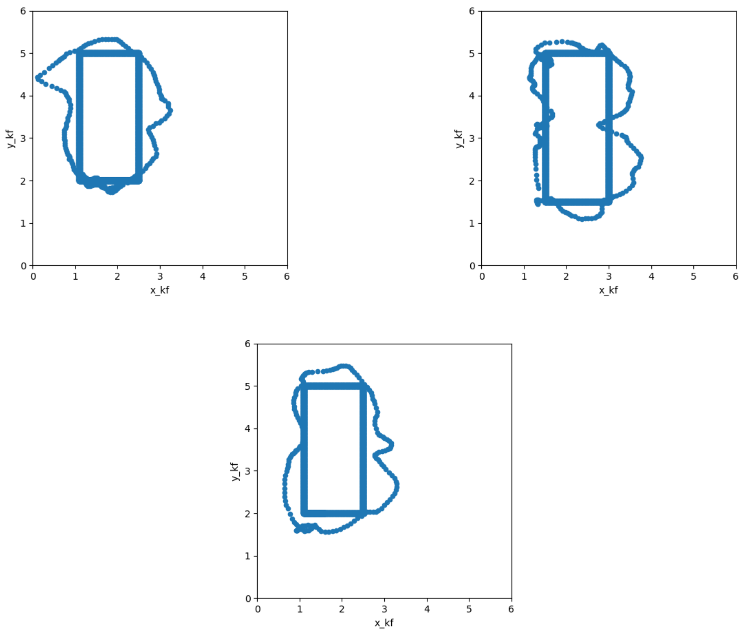


Figure 13: Three trajectories after car has completed a lap.

5 Results and Discussion

In this section we present and discuss the result of out work.

5.1 Automatic control

As seen in Figure 13, the car can drive along a square path. The resulting trajectory is somewhat oval-shaped, due to the nature of the pure pursuit controller and the car's steering. We found that a lookahead distance of 0.8 to 1.5 meters gave acceptable results. After tracing the path, the software can produce a rendered animation showing the movement of the car and the point it is driving toward, a snapshot from such an animation can be seen in Figure 14. As is apparent from Figure 13, the car will often overshoot the curves and go outside the "track". This happened more often with value of L lower than 0.8, likely because at that lookahead distance the car does not have time to react to the curve before it is too late.

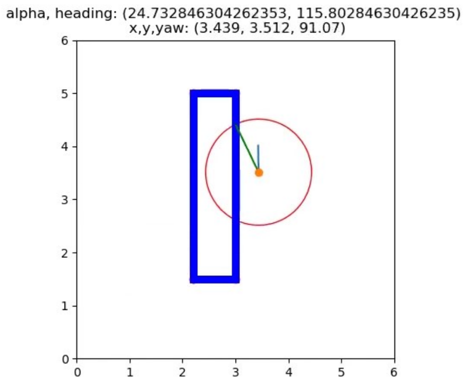


Figure 14: In progress path tracking, from animation.

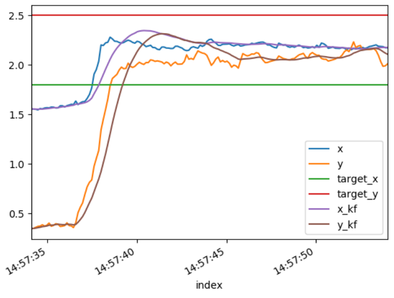


Figure 15: The original performance of the Kalman filter for a moving object. The x-axis shows time. The target values were the intended position values for the car to reach, but that performance is irrelevant to the filter and was improved on later.

5.2 Kalman Filters

The performance of the two Kalman filters are presented here. The parameters for the Q- and R-matrices were found through trial and error and are presented in Appendix B.

5.2.1 Kalman Filter for Object Tracking

The first Kalman filter performed a quite good smoothing of the noise from the position values. This can be observed in Figure 15 where "x" and "y" are the original position values from the RTLS, and filtering has resulted in a smoother position estimation. However, the approximation lags behind a bit in the start and around $y = 2$. These are points where the car accelerated and decelerated, which our model didn't handle very well.

At first we also modeled the measurement noise covariance R as time variant. We wanted to make use of the quality measure that we get with each tag reading, as mentioned in subsection 4.4, from the real-time location system and have this change the noise variance. If the values in R were high, the Kalman filter prefers the process model in the update, so with a low quality measure we thought it would be good if the model put more weight on the process model and less on the tag reading. This was only implemented in this first filter.

Our measurement covariance was at that point modeled as

$$R = \begin{bmatrix} \text{Distrust value} & 0 \\ 0 & \text{Distrust value} \end{bmatrix}$$

However, during testing it turned out that Decawave's system gave high quality measures for the locations when it should have been bad (with purposefully worsened line-of-sight), and lower quality for locations where it should have been better. The documentation claimed that a 100% quality was the best measure. After discovering this, we tried to read up on the way they calculated their quality and found it confusing, and ultimately decided to use static variance values instead.

$$R = \begin{bmatrix} {\sigma_x}^2 & 0 \\ 0 & {\sigma_y}^2 \end{bmatrix}$$

These values were decided on experimentally.

5.2.2 Sensor Fusion Kalman Filter for Vehicle Tracking

After seeing that the Kalman filter for a moving object performed badly during accelerations, we decided to implement a Kalman filter for *vehicle* tracking. To do this, we installed the IMU for more measurements (the heading of the car, and accelerations in x, y directions) and programmed an Arduino to communicate with it. During the construction of this filter, we also made many improvements to the code and its performance.

The filter performs quite well, as can be seen in Figure 16 where the car was set to drive a straight path.

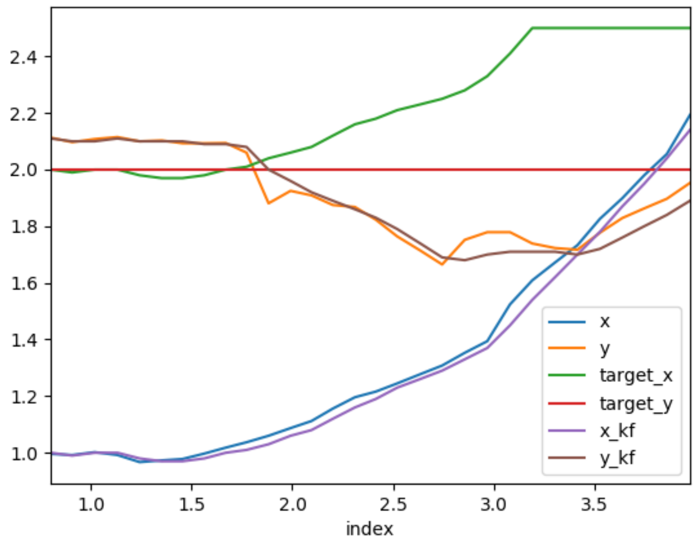


Figure 16: A comparison of the estimations of x and y positions by the sensor fusion filter and the RTLS provided positions

The filter performance during a full rectangular path can be seen in Figure 17, and it smooths out the small noise terms quite nicely.

The heading estimation and the acceleration estimations during this run can be seen in Figure 18. While the original heading (yaw) in a) goes between 0 and 360, the estimated yaw dips below 0. This creates some "jankiness" in the steering, so we decided to use the raw heading measurement for the automatic control. It already is quite noiseless, and works better than the estimation for the control.

The acceleration smoothing is quite nice. To clarify the variable names in the graph, as the IMU outputs acceleration in both the car's coordinates and the world's x and y coordinates, we have named these a_w_x and a_w_y to denote that these accelerations are in world coordinates.

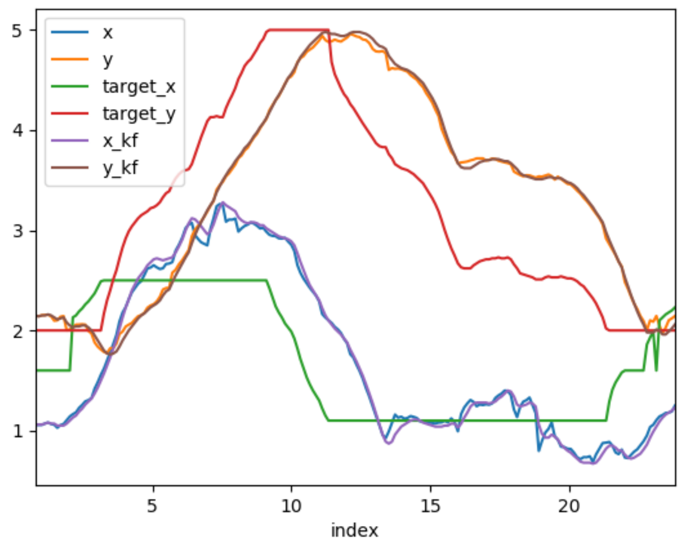
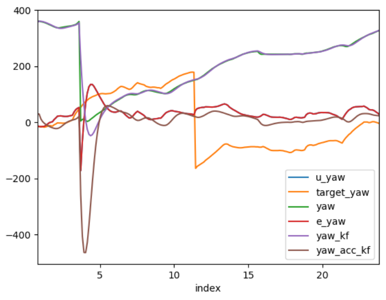
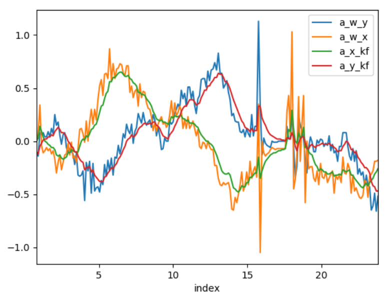


Figure 17: A comparison of the sensor fusion filter's position estimations and the RTLS ones.



(a) Heading estimation.



(b) Acceleration estimations.

Figure 18: Two subplots with estimations compared to the original values. These are from the same run and filter as in Figure 17.

5.2.3 Comparison of the filters during the path following

One thing we noticed as we did test runs using after having finished this filter and developing the whole program more is that the original Kalman filter at this point performed better than it did before, and even equaled the sensor fusion filter. You can see the updated runs with the first filter in Figure 19 where the car follows a straight path, and Figure 20 where the car tracked a rectangular path.

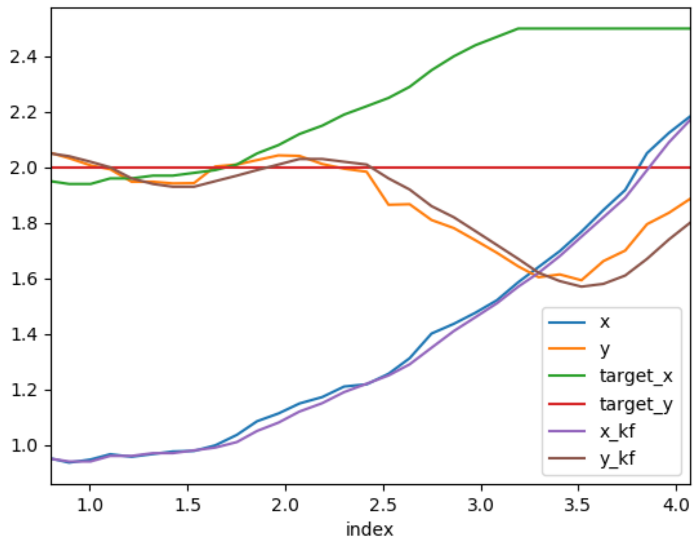


Figure 19: A comparison of the object tracking filter's estimations and the RTLS provided positions during a straight path.

We are a bit puzzled as to its improved performance, and how it rivals the filter that includes additional measurements and state variables. Perhaps the acceleration doesn't affect much, and we get so many measurements from the positioning system that the original Kalman filter has no trouble keeping up. In the beginning we weren't able to get as many measurements per second as the code wasn't as refined, so perhaps that's why our filter lagged at that point but not at the end of the project.

We wonder if the sensor fusion filter might outperform the object tracking filter if it was made using the acceleration values in real coordinates, and not the world coordinates. The IMU must be doing its own coordinate transformation to get these world values. Perhaps these acceleration values are to be considered of nonlinear origin, and that the Kalman filter can't optimize an estimate using them. The discrepancy in performance could also be the product of a software

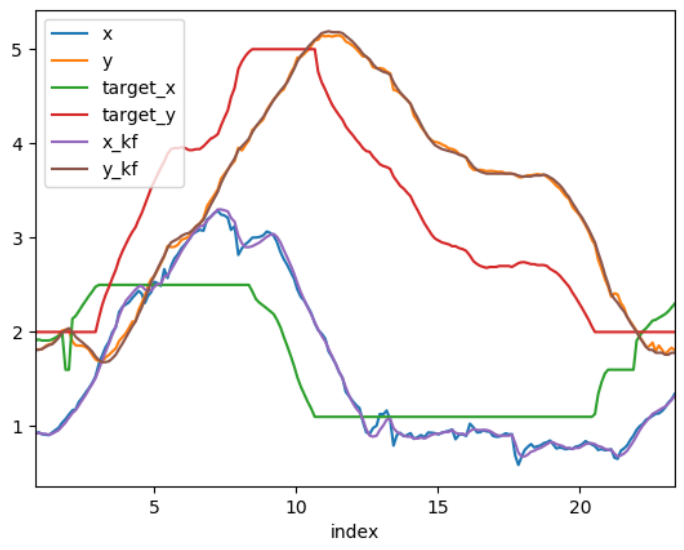


Figure 20: A comparison of the object tracking filter's estimations and the RTLS provided positions during a rectangular path.

bug.

An additional source of error could be that we base the heading estimate on the angular velocity which is calculated with a constant term c multiplied with the control signal for the steering, instead of using the car's speed s as described in Equation 18. The speed would be calculated from $s = \sqrt{\dot{x}^2 + \dot{y}^2}$, which is nonlinear. The heading does not affect the x and y estimations, so even though this approximation might be questionable it would not affect the positioning.

Perhaps it should be seen as a point in ultra wideband communication's favor that the Kalman filter for a moving object performs as well as the one for a moving vehicle.

An attempt to deal with the nonlinearities was done using an Extended Kalman filter. The matrices for it can be seen in Appendix A. This filter failed to perform very well, which either means the implementation was bad or that the model was faulty. Unfortunately we do not have the result graphs to show. Additionally, we are a bit unsure how this filter is supposed to be linearised by taking the partial derivatives (the Jacobian) of the different terms as the derivatives of our nonlinearities are still nonlinear. Cosine differentiates as (-)sine and sine differentiates as cosine. We could not find any literature that described this problem specifically, but people do seem to use cosine terms in H matrices. Perhaps it is still functional. Another thing to consider is that Extended Kalman filters work worse the more nonlinear the problem is, so perhaps our bad results can be explained by this state space being too nonlinear.

5.2.4 Dead reckoning

Our filter only updates for each serial read, so there is no dead reckoning done between the position measurements. However, if the RTLS does not return a position during a serial read, both Kalman filters can still update using the process model. The sensor fusion filter can also use the data from the IMU.

5.3 UWB performance

Our experience with the UWB positioning has been quite positive. We have received a lot of measurements per second, and the powerbanks that have powered our anchor nodes have not needed to be recharged very often.

The positions given by the system have had a little bit of noise, which is to be expected, and have over all been quite accurate. The latency between our system asking for a measurement and receiving one has been good as well, at around 0.12 ms.

5.4 Hardware performance

The car we use has a fairly wide steering radius, with a maximum steering angle of about 30°. This makes taking sharp turns more difficult. If accurate path tracing is desired, a vehicle that can turn on the spot would be more suitable than a car. Most literature on self-driving cars focus on following straight lines with little curvature (typically roads), with occasional turns.

A problem with the motor is that it is somewhat hard to drive at slow speeds. Automatic control is harder at high speeds, as the system becomes more unstable. The relatively high speed of the car in combination with the size and wide steering radius means that quite a large space has to be dedicated for testing. An advantage with the large car is that it can hold the Raspberry Pi and the other hardware components without much problem.

The MPU6050 is not a true 9-DOF IMU. It only has an accelerometer and a gyroscope, which measures accelerations and rotation speed. It can however also provide yaw pitch and roll angles using firmware, something otherwise provided by a magnetometer. The fact that the yaw is approximated likely affects its accuracy. It also means that the car has to be started in a very specific position every time, so that the IMU is calibrated correctly. For better performance, an actual 9-DOF IMU with a magnetometer should probably be used instead.

We set the range of the gyroscope to the smallest available, ±250 deg/s. Although the car would not reach an acceleration higher than 2 g during normal operation, we set the accelerometer range to ±8 g. This was because the smaller ranges seemed too noisy. We did not test the sensitivity settings in any detail however.

5.5 Software Performance

We did not use a real-time operating system in this project, as is often used in time-critical systems. We did not however observe any major problems with lag in the program, with most of the delay being caused by awaiting measurements. A potential reason for the lack of problems is that the 1.2 GHz quad core that the Raspberry Pi is equipped with is fairly powerful, to the point of wastefulness for the amount of operations we perform.

The fact that the car is equipped with what is essentially a small-form computer does however make programming convenient as programs can be directly uploaded and run on the Pi:s operating system. The operating system also provides conveniences such a network access, a file system and multiprocessing.

Python provides a large number of ready-to-use libraries and is a very widely used scripting language. The fact that it is a dynamically typed interpreted language makes prototyping faster, but also makes static analysis of the code harder. As our code base grew, it became increasingly difficult to ensure that function parameters and variables types were correct. These bugs are often not found until the execution of the program, sometimes resulting in loss of data or incorrect results. Should work continue on the program sometime in the future, it could be good to start by porting the software to another, statically typed, language to make bug-finding easier.

We have used `asyncio` as the framework for concurrency in our program. This has worked fairly well. As a large portion of the system is based around awaiting measurement values, the ability to easily suspend running tasks for the duration of the wait has been useful. The fact that `asyncio` only runs in a single thread has not been an issue for most of the tasks, as they have to be run in a sequential order. The only exception to this is the reading of the IMU and tag, which was done in two threads, and the web server that also runs in a separate thread. One thing to note about `asyncio` is that if a task fails, the exception message is silenced and the rest of the program keeps running. This made errors during the implementation hard to trace.

5.6 Future improvements

- Port the software to a compiled language, such as C++, Go or Rust. Another approach is to use ROS instead.

- Use a 9-DOF IMU.
- Use UWB distances directly instead of the position estimation from Decawave's firmware.
- Use a more "flexible" vehicle, although the controlling the car is a interesting problem unto itself.

6 Conclusion

We created a self driving car that can drive along a specified path. The path tracing is not very accurate, partly due to the construction of the car, but it can reach the designated goal. Ultimately the quality of the results depends on the intended application of the technology. If the intended purpose is for the vehicle to accurately trace a path, the system underperforms somewhat. However, if the system is considered a smaller version of a self-driving car, then we are satisfied with the result, as it can successfully complete a path that we assign it. Further tuning of the software and parameters can probably increase the performance further, but we leave that task to future contributors.

7 Attributions

We would like to thank:

- Ping Wu for providing all the hardware we needed.
- Christian Rohner for borrowing us his broadband modem.
- Tobias Holmberg for helping us with a tricky solder job.

8 Grading criteria

The following criteria were agreed upon between the students and the course supervisor.

Grade 3

- Good knowledge of ultra wideband technology
- Good knowledge of localization technology
- Good knowledge about Kalman filters and application of position estimation techniques
- Good knowledge of automatic control
- Text based UI in browser that show the position values of the anchors and the car/tag.
- The Raspberry Pi, and by extension the car, will be controlled by a simple command line based interface
- The car works along the straight line, and can stop at the assigned endpoint with an accuracy of no more than the car's length

Grade 4

- Implementation of a PID control system to steer the car with good stability
- A web based UI that shows position data and has manual control of the car, such as [STOP], [START]
- he car works along a curved line, and can stop at the assigned endpoint with an accuracy of no more than the car's length

Grade 5

- The car works along a specified closed path, and can stop at the assigned endpoint (= start point) with an accuracy of no more than the car's length
- The web based UI can visualize the whole path of the car

References

[1] Apple. *iPhone 11 Tech Specs*. 2019. URL: `https://www.apple.com/iphone-11/specs/` (visited on 10/15/2019).

[2] *asyncio - Asynchronous I/O*. URL: `https://docs.python.org/3/library/asyncio.html` (visited on 12/28/2019).

[3] Stanley Baek et al. "Accurate vehicle position estimation using a Kalman filter and neural network-based approach". In: Nov. 2017, pp. 1–8. DOI: `10.1109/SSCI.2017.8285360`.

[4] Jack Bresenham. "A linear algorithm for incremental digital display of circular arcs". In: *Communications of the ACM* 20.2 (1977), pp. 100–106.

[5] R. Craig Coulter. *Implementation of the Pure Pursuit Path Tracking Algorithm*. Tech. rep. CMU-RI-TR-92-01. Pittsburgh, PA: Carnegie Mellon University, Jan. 1992.

[6] Decawave. *DWM1001 Firmware API Guide v2.2*. 2019. URL: `https://www.decawave.com/1001-license/`.

[7] Decawave. *DWM1001 Firmware User Guide*. 2017. URL: `https://www.decawave.com/1001-license/`.

[8] Simon Haykin. "Kalman Filters". In: *Adaptive Filter Theory*. 5th Ed. Pearson Education Limited, 2014, p. 558.

[9] John D. Hunter. "Matplotlib: A 2D Graphics Environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: `10.1109/MCSE.2007.55`. eprint: `https://aip.scitation.org/doi/pdf/10.1109/MCSE.2007.55`. URL: `https://aip.scitation.org/doi/abs/10.1109/MCSE.2007.55`.

[10] Roger Labbe. *FilterPy*. 2016. URL: `https://filterpy.readthedocs.io/en/latest/` (visited on 01/06/2020).

[11] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51–56.

[12] Ayça Özçelikkale. *Signal Processing 1TE651 2018 Kalman Filtering*. Lecture notes from Signal Processing 1TE651, Uppsala University. 2018.

[13] Brian Paden et al. "A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles". In: *arXiv e-prints*, arXiv:1604.07446 (Apr. 2016), arXiv:1604.07446. arXiv: `1604.07446 [cs.RO]`.

[14] pySerial. *pySerial documentation*. 2016. URL: `https://pyserial.readthedocs.io/en/latest/pyserial.html` (visited on 10/15/2019).

[15] Robert Eisele. *Ackermann Steering*. URL: `https://www.xarg.org/book/kinematics/ackerman-steering/` (visited on 01/06/2020).

[16] Zafer Sahinoglu, Sinan Gezici, and Ismail Güvenc. "Introduction". In: *Ultra-wideband Positioning Systems: Theoretical Limits, Ranging Algorithms, and Protocols.* Cambridge University Press, 2008, pp. 93–96. DOI: 10.1017/CBO9780511541056.002.

[17] Zafer Sahinoglu, Sinan Gezici, and Ismail Güvenc. "Position estimation techniques". In: *Ultra-wideband Positioning Systems: Theoretical Limits, Ranging Algorithms, and Protocols.* Cambridge: Cambridge University Press, 2008, pp. 63–100. DOI: 10.1017/CBO9780511541056.005.

[18] Zafer Sahinoglu, Sinan Gezici, and Ismail Güvenc. "Ultra-wideband signals". In: *Ultra-wideband Positioning Systems: Theoretical Limits, Ranging Algorithms, and Protocols.* Cambridge: Cambridge University Press, 2008, pp. 20–43. DOI: 10.1017/CBO9780511541056.003.

A Extended Kalman Filter

The mechanics formulas for the extended filter look as such:

$$x_{k+1} = x_k + \Delta t v_k \cos\theta + \frac{1}{2}\Delta t^2 a_k \cos\theta \tag{31}$$

$$y_{y+1} = y_k + \Delta t v_k \sin\theta + \frac{1}{2}\Delta t^2 a_k \sin\theta \tag{32}$$

$$v_{k+1} = v_k + \Delta t a_k \tag{33}$$

$$a_{k+1} = a_k \tag{34}$$

$$\theta_{k+1} = \theta + \Delta t\dot{\theta} \tag{35}$$

$$\dot{\theta}_{k+1} = v/L\phi. \tag{36}$$

where L is the length of the car. The references for these equations can be found in subsubsection 3.3.2.

This gives us the following state variables:

$$x = \begin{bmatrix} x \\ y \\ v \\ a \\ \theta \\ \dot{\theta} \end{bmatrix} \tag{37}$$

With $F = \frac{\delta f(x,u)}{\delta x}$ and f is described by the right hand sides of equations 31 to 36:

$$F = \begin{bmatrix} 1 & 0 & \Delta t\cos\theta & \frac{1}{2}\Delta t^2\cos\theta & -\sin\theta(\Delta t v + \frac{1}{2}\Delta t^2 a) & 0 \\ 0 & 1 & \Delta t\sin\theta & \frac{1}{2}\Delta t^2\sin\theta & \cos\theta(\Delta t v + \frac{1}{2}\Delta t^2 a) & 0 \\ 0 & 0 & 1 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{38}$$

The measurements are position in the x and y-dimensions, the acceleration in the car's coordinate system of x and y (see the orientation on the IMU).

$$z = \begin{bmatrix} x \\ y \\ a\cos\theta \\ a\sin\theta \\ \theta \end{bmatrix} \tag{39}$$

and the corresponding H matrix is calculated as $H = \frac{\delta h(x)}{\delta x}$

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \cos\theta & -a\sin theta & 0 \\ 0 & 0 & 0 & \sin\theta & a\cos\theta & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{40}$$

Q and R are modeled as diagonal matrices with different values like in the regular Kalman filter.

We were not able to use the predict function from FilterPy's `ExtendedKalmanFilter` class, as it did not incorporate control signals. Therefore we made our own, based on the predict function code in the class. The addition is a V-matrix, which is added in the the calculation of the MSE matrix P as such:

$$P = FMF^T + VMV^T, \tag{41}$$

and where V and M are described as:

$$V = \frac{\delta f(x,u)}{\delta u} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ v/L\frac{1}{\cos\phi\cos\phi}, \end{bmatrix} \tag{42}$$

and u=$[\phi]$, $M = [\sigma_\phi{}^2]$.

B Sample settings file

Note that the comments are not part of the original file.

```
{
  "lookahead": 1,  # lookahead parameter for pure pursuit
  "checkpoint_threshold": 0.75,  # how close the car should be to
       a checkpoint to consider it "passed"
  "goal_threshold": 0.35,  # how close the car should be to the
      goal to start breaking
  "generate_movie": false,  # boolean, should a movie be
      generated after the run or not
  "kalman": {
      "update_delay": 0.1,  # delta t, the timestep between each
          update
      "process_dev_pos": 0,  # Q-matrix value (process noise) for
           x and y.
      "process_dev_acc": 0.08,  # Q-matrix value for acc_x and
          acc_y
      "process_dev_heading": 0.0,  # Q-matrix value for theta
      "process_dev_heading_acc": 0.08,  # Q-matrix value for
          heading velocity/theta_dot (was named _acc erroneously)
      "process_dev_vel": 0.1,  # Q-matrix value for vel_x and
          vel_y (x_dot and y_dot)
      "meas_dev_pos": 0.1,  # R-matrix value (measurement noise)
          for x and y from the UWB-RTLS
      "meas_dev_heading": 0.08,  # R-matrix value for the heading
          /theta/yaw from the IMU
      "meas_dev_acc": 1.2,  # R-matrix value for the
          accelerations from the IMU
      "speed_div_by_length": 0.9,  # the value for c in the B-
          matrix, used in the Kalman filter for a moving vehicle
      "variable_dt": true,  # a toggle for if the Kalman filter
          updates should have a static or variable dt (static
          value is set in update_delay)
      "uwb_only_kf": false  # toggle between Kalman filters. if
          true is uses the Kalman filter for a moving object that
           only has UWB-measurements, if false it uses the larger
           Kalman filter for a moving vehicle that has
          measurements from the UWB-RTLS as well as the IMU
  },
"path": {
      "x": [1.1, 2.5, 2.5, 1.4, 1.4, 1.8], # x coordinates of
          path vertices
      "y": [2.0, 2.0, 5.0, 5.0, 2.0, 2.0] # y coordinates of path
           vertices
  }
  },
  "pid": {
      "steering": { # steering pid controller settings
          "enable_d": false, # enable derivative component of PID
          "enable_i": false, # enable integrating component of
              PID
          "p": 1, # multiplier for p component
          "d": 0.1, # multiplier for d component
          "i": 0.01 # multiplier for i component
    },
    "speed": { # speed pid controller settings
        "enable_d": false,
        "enable_i": false,
        "p": 1,
            "d": 0.1,
            "i": 0.01
          }
      }
  }
```