TEL AVIV UNIVERSITY

School of Electrical Engineering

Simulation of Formula Student Driverless Racing Challenge

Author
Idan Timsit

Supervisors
Dr. Dan Raphaeli
Dr. Igal Bilik

September 2021

Table of Contents

Acknowledgements
I would like to thank my supervisors, Dr. Dan Raphaeli of Tel Aviv University and Dr. Igal Bilik of Ben-Gurion University for their support during this work.
In addition, I would also like to thank other colleagues and contributors: Itay Cohen, Moshe Levy-Israel, Yonatan Menaker, Avi Resler and Ishay Eljarat, for their time and advice.

Abstract

Race car projects have been a central part of any major university's engineering faculty for over a decade. Due to a growing interest in electric and autonomous vehicles in recent years, more and more racing projects move from revolving around internal combustion engines and into a combination of electric motors with autonomous features.
There exists a gap between academic studies and industry applications in practice. Students are expected to implement advanced methodologies with system-wide cooperation, while having only little or no experience at all in the industry.
This document proposes a complete, end-to-end solution to the inter-institutional competition mission defined by the FSAE, along with explanations and suggestions for improvements. Moreover, the suggested solution is validated using a simulation environment designed within Unreal Engine 4, and controlled via Python using the Microsoft AirSim plugin.
The same simulation environment can then be used to verify and debug a modified or improved set of algorithms developed by the students.
This allows the conduction of repeatable experiments with near-exact ground-truth, without the cost and time needed for setting up a real competition environment, and most importantly, eliminating the dangers of possible injury or equipment damage associated with such activities.

Preface

Many of us surely remember what it was like to graduate from engineering college and move forward to work at the industry. Understanding that what we've learned is mostly theoretical, and that many things don't work as well in practice as they do on paper.
The idea for this work came to me after visiting a group of students from a respected university, participating in the Formula SAE (an international student design competition held by the Society of Automotive Engineers). The meeting was held in the year preceding to their transformation from internal combustion engine to an all-out electric vehicle confronting the autonomous challenge.
Even though professional guidance from industry leading companies was in plan, I felt that there should be a starting point for every student to get familiar with the entire process of autonomous driving and its glossary.
This project was intended for providing a common ground to every student wanting to join such a team, before the reading of countless papers or going through long online courses.
I am also in hope that one or more students from such a group would take the role of being in charge of the simulation, and by doing so, provide their team with a much needed means of running algorithms through a simulated environment before testing it in the field.
Simulation is an expertise by itself, and knowing how to adapt it to one's needs can be a powerful tool.
Even though this work may point out likely pitfalls when necessary, its contents only propose a suggestion for solving each autonomous stage. It is expected from the students to learn, improve, perfect and test their own algorithms in order to complete the challenge which may also evolve in the upcoming years.
This document is intentionally targeted for being comprehensive rather than concise, in order to bridge the said gap that exists between academic teachings and industrial practices.

You may find all the client source files in the Github repository:
https://github.com/StevenCoral/formula_student_sim_client

I hope you will enjoy reading this work as much as I did writing it.

Table of Symbols

Symbol	Remarks	Symbol	Remarks
V	Voltage [V]	X_c, Y_c, Z_c	Point coordinates in camera frame
L	Inductance [H]	f, f_x, f_y	Camera focal lengths [m]
R	Resistance [ohm]	c_x, c_y	Pixel offsets from principle axis
I	Current [A]	$\tilde{x}, \tilde{y}$	Pixel coordinates in camera frame
t	Time [sec]	x_p, y_p	Pixel coordinates in image frame
T	Torque [N*m]	W_i	Image width [pixels]
$\dot{\theta}, \omega$	Angular velocity [rad/sec]	$HFOV$	Horizontal field of view [rad]
k_e	Back-EMF constant [V*sec/rad]	r_L	Left wheel turn radius [m]
k_v	Velocity constant [V*sec/rad]	r_R	Right wheel turn radius [m]
k_t	Torque constant [N*m/A]	r_B	Bicycle model turn radius [m]
k_{sys}	System gain	l_d	Lookahead distance [m]
τ	System time constant	g_{xy}	Lookahead point on path
G_x	Generic transfer function	β	Pure pursuit steering angle [rad]
P_x	Plant transfer function	v_c	Vehicle speed [m/sec]
ω_n	Natural frequency [rad/sec]	d_c	Closest point on path
ζ	Damping coefficient [N*sec/m]	θ_e	Aligning steering angle [rad]
v_s	Voltage step size [V]	θ_f	Closing steering angle [rad]
k_0	Canonical form numerator	a	Canonical form constant
k_{str}	Steering coefficient	δ	Final Stanley steering angle [rad]
k_p	Proportional control coefficient	W_{bb}	Bounding-box width coefficient
k_i	Integral control coefficient	H_{bb}	Bounding-box height coefficient
k_d	Derivative control coefficient	V_c	Vector in camera coordinates
k_f	Feed-forward control coefficient	$corr$	Step settling corridor [%]
$\bar{p}_w$	Point in world coordinates	ST	Settling time [sec]
$\bar{p}_v$	Point in vehicle coordinates	t_s	Sampling time [sec]
$\bar{p}_c$	Point in camera coordinates	T_v^c	Transformation from camera to vehicle coordinates
T_w^v	Transformation from vehicle to world coordinates	T_v^c	Transformation from camera to world coordinates

1. Introduction

These days, the field of autonomous driving is gathering momentum at a staggering pace.
Many automotive and technology manufacturers invest billions of dollars on research and development in an attempt to stay in the race. Diverse teams of engineers, programmers, UI/UX designers and others are assembled in order to approach different aspects of the autonomous driving challenge.
This challenge is usually broken down into several stages, recurring in a loop until the mission is complete (in fact, many of these are run in parallel using multi-processing). This decomposition can occur in several ways, depending on the point-of-view. From this work's perspective, the stages of autonomous driving are broken down to the following items:

- Sensing – The process of acquiring data from sensors, in a manner that enables a practical use of that data in real-time. This might sound simple, but many aspects have to be taken into account: choosing which sensors and how many of them to use, where to install them over the vehicle, writing or utilizing an efficient driver software, and which data structures to use in order to represent the data. Real time data might need different representation than the same data being saved for debugging purposes.
  Data rates and synchronizing between sensors must also be taken into consideration. Different sensors (even those of the same type) may have different connectors, hardware ports, communication protocols, data formats, and even support different FPS rates. Time-stamping a data acquisition frame when it arrives at the processor does not necessarily mean that it holds true to the time where the data was actually acquired within the sensor. Acquired data is usually received in the sensor-ego frame of reference, and each sensor's data needs to be transformed into a common frame of reference. Another capability which may be desirable is the visualization of the input data. For example, ROS's Rviz renders sensor output in 3D and gives an idea about how the system grasps its surroundings.
- Perception – the process of understanding what the input data actually means. This could be a difficult signal processing task, depending on the desired level of detail. For example, one might only want to detect and recognize road surface markings so they can follow a lane, but more complex tasks such as detecting obstacles and their type should require the fusion of multiple sensors. In urban autonomy, there is great importance to tell the difference between a pole, a rock, another vehicle and a pedestrian. If it is a pedestrian, some actions might be taken in case it is a jogger, and other behaviors in case it is an old lady with a baby carriage. This carriage may be black, or it can be gray, or even blue and so on.
  The vast amount of possibilities is one of the reasons why perception algorithms are leaning more and more towards machine learning techniques [1][2].

- Decision making – How is the vehicle supposed to act given the perceived information. A set of predefined behaviors are assigned to certain situations. For example, if an autonomous vehicle driving on a highway detects that the truck in front of it is driving slower, and that the distance between them is closing, that vehicle might want to initiate a bypassing maneuver. Yet, in case it had also detected that the adjacent lane is not vacant for entry, the best immediate action might be to slow down instead.
  A problematic situation may be where the vehicle is in a state for which no action was defined. An educated set of fallback actions should be defined for these cases.
- Planning – Once the vehicle's intent is known, a trajectory must be generated such that the vehicle can follow it with respect to its dynamics and levels of holonomy. A heavy, fast-moving car with Ackermann steering is less agile and has more variables to consider (tire friction, wind drag, etc.) than a light, slow moving, differential drive robot that can turn 360 degrees in place.
  Once the intended trajectory is calculated, a set of high-level instructions is sent to the vehicle in order to guide it along the desired path. These instructions serve as the reference signal inputs for the next stage, to be followed by the vehicle's actuators.
- Control – The stage where all the decisions made in the previous stages are manifested into actual, physical movement of the vehicle. At the high level, this usually splits into lateral (steering) and longitudinal (throttle, brakes) control. In internal combustion engine vehicles, there is much low-level control going on under the hood such as automatic gear shifting, fuel injection, oil pumping and more. Electric vehicles have motor controllers responsible for voltage regulation, current commutation, regenerative braking, etc.
  Low level control is usually covered by the controller's manufacturer, but when designing a vehicle from scratch, in many cases a deep-dive into controller implementation is required. System identification techniques can be used in order to design efficient controllers.
  As a side note, this stage requires its own sensing scheme in order to deliver feedback for a closed control loop. Signal processing and estimation take an integral part here as well, but on a different level than those utilized in the Sensing and Perception stages.
  Needless to say, different missions are defined for different challenges. The set of possible situations, and therefore the set of decisions to be made during highway driving is completely different than those of urban driving, for example.

In the FSAE autonomous challenge, a closed path made of traffic cones is placed over a level asphalt lot. Along the path's driving direction, the cones to the right of the vehicle are colored yellow and have a black stripe over their mid-height, while the cones to the left of the vehicle are colored blue and have a white stripe over their mid-height. There is an equal amount of yellow and blue cones along the said path.

The mission itself is to navigate along the path during two full laps. During the first lap, the task is to detect and recognize the cones, and use them to build a known path for the vehicle to follow in the next iteration. During the second lap, and assuming that the path to be followed is now already known, the task is to complete the lap as fast as possible, without tipping the cones over or deviating off-course.
Consecutive laps might be required, and assuming that the task is completed without errors, completion in minimum time is the scoring criterion.

2. Background

Presented below is a short overview of several topics which appear in this work and require basic understanding.

2.1. Unreal Engine 4

Abbreviated UE4, this is a professional, open-source 3D gaming engine used in the gaming industry to design, program and distribute top-notch computer, mobile and VR games.
Its main competitor in the open source field is called Unity Engine, but while Unity is only programmable through C#, Unreal can be programmed by C++, Blueprints (a proprietary node-based language similar to Matlab's Simulink), or a combination of both. This makes it a convenient candidate for basing the simulation upon.
Getting familiar with the UE4 framework and what it has to offer requires its own learning curve and by itself is not part of the autonomous work.
It is important to note that quality game design is a long and expensive task, with a large portion of the effort invested into art (that is, 2D textures, PBR material editing, vertex-based 3D modeling, etc.). Since art requires artistic expertise and is purely non-engineering, almost every art-related item used in this work was either purchased or downloaded freely from the web.

2.2. AirSim

This is an open-source plugin for UE4 initiated and supported by Microsoft [3]. It enables the manipulation of the game (simulation) scenario using an external client, which may be run on the host computer or even on a remote machine.
The client enables the user to control the ego-vehicle in real-time (change the throttle, steering, etc.) and also poll for information. This information can be a parameter of the vehicle itself, such as position, speed and rotation, or it can be the data acquired by the onboard sensors, such as a lidar point cloud, RGB image, depth image and so on.
AirSim too has alternatives on the market, the most notable being Carla [4].

2.3. Direct Current Electric Motors

While these may vary in configuration, the de-facto standard when speaking about DC motors are those which have a fixed magnetic field originating from a permanent magnet, or PMDC (Permanent Magnet Direct Current) motors.
Whether brushed or brushless, the dynamic behavior of these motors is similar in nature.

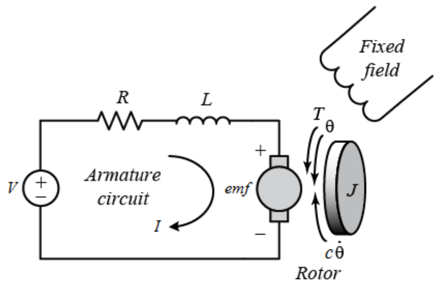


Figure (1): a circuit representation of an electric DC motor

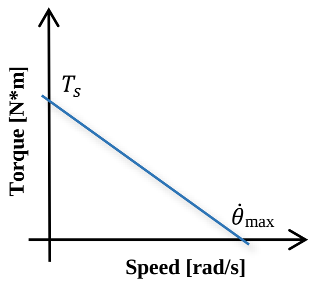


Figure (2): a characteristic speed-torque curve for an electric motor

The electrical equation derived from the circuit depicted in figure (1) is:

$$V = L\frac{dI}{dt} + RI + k_e\dot{\theta} \ (1)$$

with k_e being the motor's back-emf constant. Such a motor's torque depends linearly on the current that runs through it, and is winding-dependent, regardless of its other characteristics:

$$T = k_t \cdot I \ (2)$$

with k_t being the motor's torque constant. There is an additional important constant k_v called the motor's speed constant, which has the same value as k_t [5].
Substituting (2) into (1) and neglecting the small values of $L\frac{dI}{dt}$ yields:

$$T = \frac{k_v}{R}(V - k_e\dot{\theta}) \ (3)$$

This is the equation shown in figure (2), from which the motor's stall torque T_s and the line slope can be extracted, given that the other arguments are known.
Other properties can be extracted as well, such as the motor's theoretical power curve.

2.4. Low Order Dynamic Systems

One can define the order of a dynamic system as the degree of the characteristic polynomial (denominator) of its transfer function in the Laplace plane, assuming it is proper.
First order systems can have the following canonical form:

$$G_1 = \frac{k_{sys}}{\tau S+1} \quad (4)$$

with k_{sys} being the system's DC gain and τ being a measurable time constant.
In this form, step responses are characterized by an immediate ascent upwards without any overshoot. It is easy to check that in a closed feedback loop, there will always be a steady state error, unless compensated by a controller with an integrator.
It is often the case where the transfer function between PMDC motors' voltage and angular velocity behaves as a first-order system, neglecting static friction and other nonlinear terms.

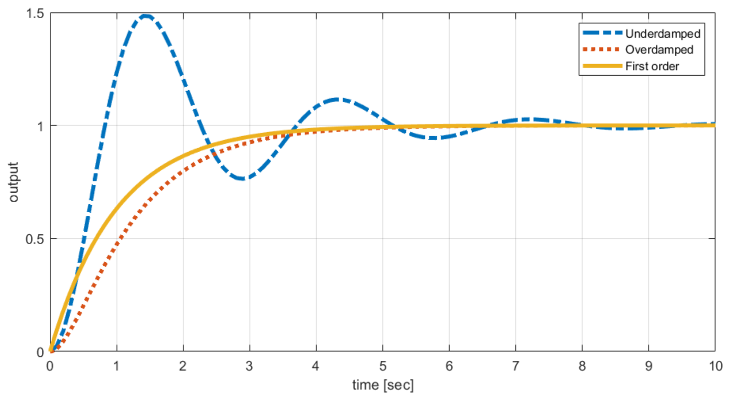


Figure (3): characteristic responses of arbitrary 1st-order and underdamped/overdamped 2nd-order systems

Second order systems are often depicted using their own canonical form:

$$G_2 = k_{sys} \cdot \frac{\omega_n^2}{S^2+2\zeta\omega_n+\omega_n^2} \quad (5)$$

With ω_n being the natural frequency, and ζ being the damping coefficient.
If $0 < \zeta \leq 1$, the system is called underdamped, while when $\zeta \geq 1$, the system is overdamped. Undamped systems ($\zeta = 0$) are not considered stable and will not be discussed.
Second order systems are characterized by a gradual convex ascension, and in cases where the system is underdamped, an overshoot followed by convergent oscillation will occur.
In some cases, the transfer function between a motor's voltage and its angular velocity can be passed through an integrator, then simplified to get the second-order transfer function between the voltage and the angle itself (rather than the angular velocity). Combined with a wheel or gear radius, angles can be converted into linear distance.

2.5. System Modeling and Identification

Controllers are almost never guessed “out of the blue”. A good controller must take its system dynamics into consideration, be it is electrical, mechanical, chemical or other.
Some level of system modeling should take place. One reasonable approach would be to create a linear model using differential equations, and a more complex one taking nonlinear effects into consideration, such as dead-zone, hysteresis, transport delay, etc.
A controller may be designed using classical or modern approaches for the linear model, then fine-tuned using the complex model, simulated and solved in a software such as Simulink.
As a general rule of thumb, the more knowledge there is about the system, the better. This is one of the reasons why a state-space representation is considered richer than SISO transfer functions. Note, however, that knowing some states may require a dedicated sensor, state observer, or even be completely unobservable.
Even when deriving the linear model, it is often the case where some system parameters have uncertainties or are difficult to measure, such as moments of inertia, viscous friction, etc.
A different approach is to record a set of inputs and outputs from the desired system, and then run a system identification procedure along with a priori structural assumption, in order to estimate the system unknowns. Many such procedures exist, and Matlab has a dedicated toolbox for this task.
An important simple-yet-analytic method is called a bump test, and it is effective in estimating the parameters of a 1st-order system like the electric motor (even with inertial loads attached to it). The method is based on the canonical form depicted in equation (4) and the entire process needs to be recorded. It consists of injecting the system with a small step input in an open loop in order to overcome static friction; When the system reaches a steady state, a second, higher step is injected. Once the system reaches a steady state again, the recorded data can be analyzed.
The parameter k_{sys} (steady state gain) would be the ratio between the input voltage and the measured steady state output velocity, deducting the initial, smaller step.
Injecting a step input voltage with amplitude v_s into the canonical form would result in:

$$\Omega(s) = \frac{v_s \cdot k_{sys}}{\tau \cdot s + 1} \quad (6)$$

The inverse Laplace of the above expression can be taken to get the transient response of the motor’s velocity with respect to time:

$$\omega(t) = v_s \cdot k_{sys} \cdot \left(1 - e^{-\frac{t}{\tau}}\right) \quad (7)$$

It is easy to check that when $t = \tau$, the velocity would be roughly 0.632 times the steady state. Hence τ, the motor’s time constant, is the time it takes the step response to reach 63.2% of its final value.

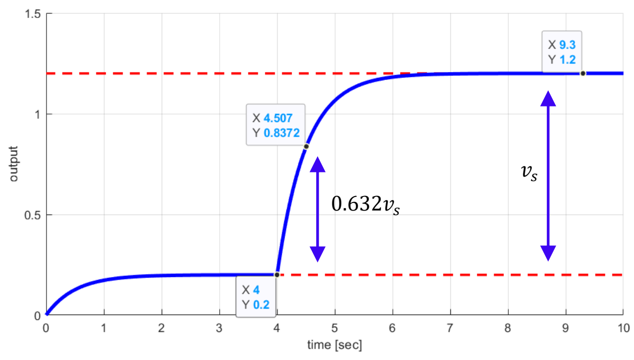


Figure (4): transient response of a typical bump-test with key values marked

Convenient methods for identifying 2nd-order systems from their outputs exist as well.

2.6. Transformations

It is known that displacements in 3D can be represented by $\mathbb{R}^3$ vectors. Rotations can be represented by rotation matrices, quaternions, Euler angles and more. Conversions between these forms are practiced a lot and it is an important know-how, even if using an existing library such as Python's Scipy or Transforms3D. Euler angle representation is *heavily* dependent on the order of rotation and whether the action is intrinsic or extrinsic [6]. Transformation matrices are entities that combine rotations and translations into a single matrix. While these can be handled separately, there is an advantage in combining them so that matrix inverses explicitly depict the same transform in the opposite direction. It enables to save the transform information into a single data structure and freely move from one frame of reference to the other. Moreover, transformations can be cascaded by using left-hand matrix multiplication, so that compound frame relations are easily defined.

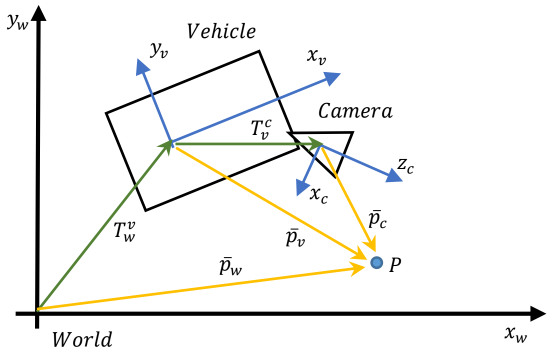


Figure (5): representations of the same spatial point from different point-of-views

$$R_{3x3} = \begin{bmatrix} a & \cdots & b \\ \vdots & \ddots & \vdots \\ c & \cdots & d \end{bmatrix} \qquad \bar{d}_{3x1} = \begin{bmatrix} k \\ m \\ n \end{bmatrix}$$

$$Pad_{1x4} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{4x4} = \begin{bmatrix} a & \cdots & b & k \\ \vdots & \ddots & \vdots & m \\ c & \cdots & d & n \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure (6): assembly of a transformation matrix out of a rotation matrix, translation vector and a padding bottom row

Sown in figure (5), the point P can be represented in the camera frame as the vector $\bar{p}_c$, or in the vehicle frame / global world frame as (respectively):

$$\bar{p}_v = T_v^c \bar{p}_c \ (8)$$

$$\bar{p}_w = T_w^v \bar{p}_v = T_w^v T_v^c \bar{p}_c \ (9)$$

The most convenient form can be used at any point, depending on the processing context. One can observe that given $\bar{p}_w$, the others can be computed in the opposite direction:

$$\bar{p}_c = [T_v^c]^{-1} [T_w^v]^{-1} \bar{p}_w = [T_w^v T_v^c]^{-1} \bar{p}_w \ (10)$$

2.7. Camera Intrinsics

Every camera has its own unique extrinsic parameters, in the form of a 3-by-3 matrix and a set of distortion coefficients (the amount depends on the model). It is important to know this information since it enables the conversion of points from the world space into the camera's pixel space and vice-versa [7]. The values can be extracted by applying simple calibration procedures using tools like Matlab and OpenCV. Before coordinates can be converted, image rectification must take place (undistorting the image, and cropping out residual pixels).

Let $\bar{p}_w$ be a point in space, with respect to some frame of reference. As previously mentioned, it can be represented as $\bar{p}_c$ in the camera frame of reference, assuming that the translation and rotation are known.

This may be a good time to note that a camera's canonical coordinate system has the X axis pointing to the right, Y axis pointing downwards, and Z axis protruding forward, contrary to other conventional coordinate systems.

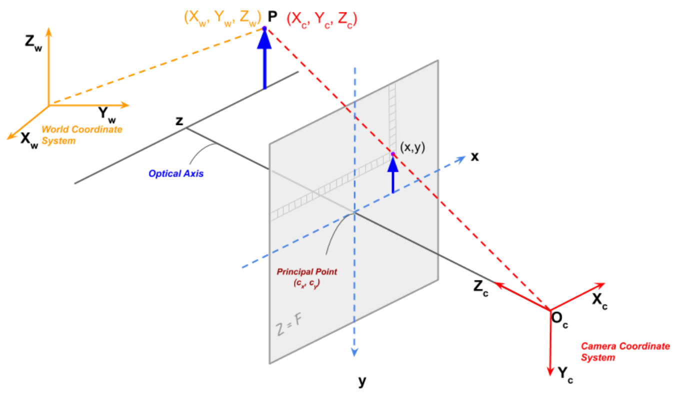


Figure (7): depiction of the relations between world, camera and sensor (pixel) coordinate systems.

Using triangle resemblance, it can be inferred that the quants depicting $\bar{p}_c$ are:

$$\tilde{x} = f \cdot \frac{X_c}{Z_c} \quad (11)$$

$$\tilde{y} = f \cdot \frac{Y_c}{Z_c} \quad (12)$$

Where f, called the "focal length", is the ratio between the distances of the focal plane and the CCD/CMOS sensor plane, as shown in figure (7).
Mathematically, the sensor pixels may be rectangular, hence f_x and f_y are used, yet in practice they are often the same (or very similar due to calibration quantization errors), and can be computed using a closed formula:

$$f = \frac{W_i}{2 \cdot \tan\left(\frac{HFOV}{2}\right)} \quad (13)$$

With W_i being the output image width in pixels, and HFOV being the camera's horizontal field-of-view angle in radians (measured across the entire aperture).
As the camera coordinate system origin is based at the principle axis, i.e. the "center of the image", and the pixels' frame origin is based at the top-left corner of the image, a displacement must be added to x and y, in the form of two parameters: c_x and c_y.
Their value will be exactly half of the image resolution width and height, respectively, unless the camera lens or sensor are misaligned (which in practice they usually are).
All-in-all, the result is called the camera matrix:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \quad (14)$$

If the vector P_c is normalized by Z_c, all of the above can be taken into account and rewritten:

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c/Z_c \\ Y_c/Z_c \\ 1 \end{bmatrix} \quad (15)$$

While x_p and y_p on the left-hand side are the pixel indices to which the world point $\bar{p}_c$ maps.
Notice how the Z axis does not play a part, since in the absence of additional information a pixel's depth value cannot be extracted from a non-stereo camera image.
It is important to state that usually, an automatic calibration procedure will take place using either Matlab, OpenCV or another proprietary tool.

2.8. Wheel steering

In all modern cars, the front wheels are never parallel to each other, except for when the car is driving straight ahead. The front wheels' axes of rotation are mounted not on a parallelogram, but on a trapezoid. This mechanism is called "Ackermann steering".

It attempts to keep all wheels in the car perpendicular to its turning radius, in order to prevent the tires from experiencing lateral slip - a process that generates a lot of noise and heat and rapidly accelerates tire wear.
Designing an Ackerman trapezoid is a cumbersome task, involving high nonlinearity due to trigonometric expressions. The result is usually not perfect for the entire turning range, and often a piecewise-linear diagram is used to describe it and extract values using interpolation.
When referring to a "steering angle", the intention is to that which is equivalent to the "bicycle model" steering angle.

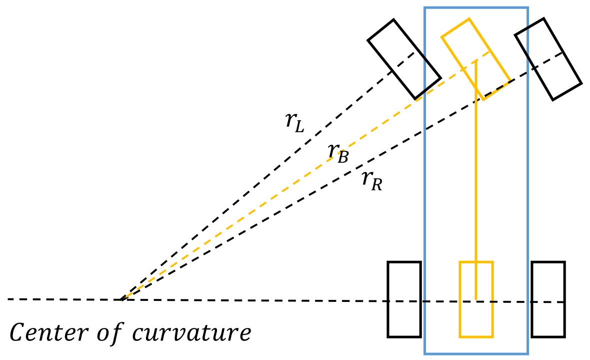


Figure (8): different steering angles applied to the left and right wheels, and the bicycle model within the same turning radius

A bicycle does not suffer from left and right wheel angle discrepancy as the Ackermann model does. Notice in figure (8) how the left turn radius r_L is not the same as the right one, r_R, which generates a difference between the left and right steering angle.
As mentioned, the function is nonlinear and thus it would be wrong to suggest that the bicycle model is an average of the two. It can be calculated trigonometrically assuming the other angles are known.

2.9. Planning

There are many algorithms in literature for generating and following a path. They vary depending on the mission, movement constraints, complexity and more.
The most common and intuitive path following algorithm is called pure pursuit. Like many other heuristic algorithms, it too has several variants, one of which will be presented.
Given a known path and a driving direction, one can use a lookahead distance l_d and pick a point g_{xy} along the path that is roughly in that distance from the vehicle. The vehicle would then calculate the error between its own heading vector and the direction vector between g_{xy} and its own location. Multiplied by a coefficient, the desired steering angle β would be acquired.

Some variants use the error angle's tangent or sine multiplied by a steer coefficient. Many other variations exist as well [8]. A typical tradeoff with this method resides in the lookahead distance – too close, and the vehicle will oscillate sideways. Too far, and the vehicle would "cut corners". One might want to base that distance on the vehicle speed.

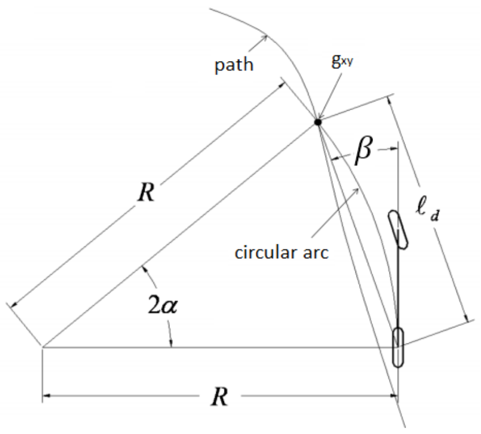


Figure (9): a sketch of the parameters taking part in a pure-pursuit following schema

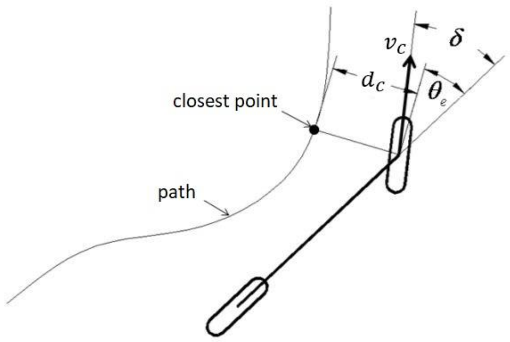


Figure (10): a sketch of the parameters taking part in a Stanley-method following schema

Another popular motion planning algorithm is called the Stanley method, named after the winning vehicle in the 2006 DARPA Grand Challenge competition upon which it was introduced [9]. This method constructs the desired steering angle out of 2 separate terms, both based to some extent on the point along the path which is closest to the vehicle.
The first term θ_e depends on the path's own curvature, and its goal is to align the vehicle with the path. The angle is regarded as the difference between the vehicle's heading and the direction of the path's tangent at the point closest to the vehicle.
The second term θ_f attempts to bring the vehicle closer to the path, by calculating the arctangent of the closest distance d_c divided by the vehicle own speed v_c, multiplied by some steering coefficient:

$$\theta_f = \text{atan}\left(\frac{k_{str} \cdot d_c}{v_c}\right) \quad (16)$$

As mentioned, the final steering angle δ will be the sum of the two terms.
In addition to the steering algorithm, the target speed of the vehicle also changes when using this method. The faster a car travels, the harder it is to take turns. Hence, maximum and minimum speed values are chosen, and the desired speed would be the maximum, subtracted by a value proportional to the curvature of the path at the lookahead distance. The higher the curvature, the lower the target speed will be, yet never below the minimum.

2.10. Ego-Sensing

Besides the need to sense the environment, a vehicle needs to sense its own states for estimating its pose (position + rotation) and velocity, and use these estimations as control feedback or as transformation data for fusing other sensors (e.g. transforming a detected cone from vehicle frame to world frame in order to build a map, etc.).
Some of these may consist of:

- Rotary encoders: installing an incremental encoder on each rear wheel and an absolute encoder over the steering wheel (or steering mechanism) is important for calculating the car’s velocity, and through dead-reckoning integration - also its pose.
- IMU: another useful sensor is the inertial measurement unit. These usually provide 3-axis linear acceleration data (accelerometer) and angular velocity data (gyroscope). Some already come wrapped with filtering algorithms and drivers, then called AHRS. They can be cheap or highly expensive, depending on the technology and the manufacturer. Several can be placed over the vehicle for redundancy.
- GPS: if the car is out on the open, precision DGPS with real-time kinematics (RTK) corrections can be used to further estimate the vehicle’s states.

Note – accurate time synchronization is important when fusing sensors together.

3. Goals

The ultimate goal of the project is to provide a simulation environment for the development of FSAE student projects. For manual driving, it should enable the user to drive a vehicle using controller inputs (keyboard / joystick / wheel & pedals). For the autonomous challenge, it should contain the assets necessary for defining and executing autonomous driving algorithms, as well as an operational end-to-end example of the proposed solution. This solution should describe a scientific starting point for each sub-team to start working, in absence of previous experience, in all the relevant fields. Being more specific:

3.1. Environment

- Build and adjust correct settings for an Unreal Engine 4 project.
- Provide at least 1 racing course map for manual driving usage and 1 parking lot map for autonomous driving experiments.
- Provide a way to generate a mission course consisting of yellow cones with black stripes on the right side and blue cones with white stripes on the left. Provide the actual cone assets as well.
- Define a racing car actor to be used in the challenge. This includes a skeletal mesh, physics asset, tire assets, movement controller and wrapping it all up with a pawn class.

3.2. Sensing

- Make sure the system is able to output data for a camera and a lidar upon user request.
- User should be allowed to define multiple sensor instances and place them over the vehicle.

3.3. Perception

- Detect cones and estimate their location on the map.
- Classify cone type/color from the sensors input.

3.4. Planning

- Use all available data to plan a route during the first and second laps.
- Generate reference signals such that the vehicle is guided to follow the intended path.

3.5. Control

- Close a velocity control loop over the vehicle throttle reference (longitudinal control).
- Close a position control loop over the wheel steering angle reference (lateral control).

3.6. Mission

- The vehicle should do all the necessary mapping in the first lap at a low velocity.
- Second lap must be made at a significantly higher velocity than the first one.
- The information in the project outcome should cover all aspects of the autonomous life cycle, on a basic level.

4. Scope of work

4.1. Coverage

Beyond requirements expected from this work, some subjects are not covered:

- Simulated vehicle tuning. Institutes who haven't participated yet in the autonomous FSAE do not have an existing functional electric car. Moreover, every year a new car is made. In the absence of an actual vehicle to investigate nor the means to make measurements on such, there is no benefit in attempting to tune the dozens of parameters so that a car's simulated dynamic response would match those of a specific real car. The simulated car(s) were designed to have a generic behavior suitable for electric vehicles.
- Physical vehicle implementation. Inside a simulated environment in which a car is already built and functional, sensors can be placed anywhere in the volume with perfect knowledge of their transformation, and they provide readily available data on-demand. Much work needs to be done in order to turn simulation into reality. Sensor physical installations, wiring, power distribution, software data acquisition (drivers/protocols), interference, noise, calibration, alignment, etc. will not be discussed and are left for the students to assess and figure out.

- Miscellaneous. Similarly, other system-related considerations such as the choice and distribution of processing platforms (PC/embedded), processing units (CPU/GPU), operating system, architecture (pub-sub/ROS/IPC), runtime complexity requirements, power consumption, etc. are also out of the scope of this work.

4.2. Previous Work

Many driving simulators exist on the market [9]. Some based on open-source engines, others are proprietary and may require paying thousands of dollars for a license.

- Monash motorsport – In 2018, an FSAE team from Monash University at Melbourne, Australia has developed a dedicated driving simulator based on Assetto Corsa, a highly-moddable racing game. It focuses on high vehicle model fidelity and manual driving [11].
- FSAE simulator – In 2018, an FSAE team from the Technion institute in Israel has developed a simulator based on Unreal Engine 4.18 [12].
- FSDS – In 2020, based on the simulator above, a thorough project was created incorporating ROS usage and enabling multiplayer competition [13].
- Other Technion projects were based over the FSAE simulator, for purposes such as reinforcement learning [14].

In addition, a large number of FSAE teams' past projects are available online and are a valuable source of information regarding driverless car implementations.

5. Methodology

All experiments will be conducted using the same PC for the simulator and the client. This might hamper processing performance but mitigates transport delay over the network.

5.1. Environment

5.1.1. 3D Engine

The underlying engine powering the simulation world was chosen to be Unreal Engine 4.26 by Epic Games. It was found to be the most convenient environment to manipulate providing powerful capabilities, strong rendering, and being supported by many plugins. The plugin which was used is Microsoft AirSim. It is easy to install, provides enough functionality to adhere to the project goals, and has a documented API.

There had been attempts to reuse the FSDS' Unreal project as a basis for this work. Many assets were not found and references were broken, probably due to upgrading from version 4.18 to the currently newest version 4.26. Also, that project heavily depends on ROS and network gameplay, and was intended for conducting a competition (with lap time, rules, and other additions). The goal for this work was to bring basic functional abilities so that anyone may create their own use-cases in the future without the need to remove functionality. Therefore, a new project was created from scratch using some of what could be salvaged from the FSDS work.

5.1.2. Coordinate Systems

There are differences in coordinate systems between several entities involving this work. UE4 is using an arbitrary left-handed coordinate system (X points to the front, Y points to the right, Z points upwards).
Rotations within the engine are specified by Euler angles, in an intrinsic order of rotation of yaw-pitch-roll, namely around Z, then Y, then X. Positive rotations are directed clockwise for the X and Y axes, while it is counter-clockwise for the Z axis.
The AirSim API uses the same coordinate system, with the exception of having the Z axis point downwards. This is due to it being originally dedicated for use with aerial platforms such as drones.
Rotations within the API are specified by quaternions, and therefore conversion is to be made between those and Euler angles, both for getter and setter functions. The coordinates origin is always assigned to the ego-vehicle's spawning point, regardless of its position within the UE4 map.
This work is using a different coordinate system, referred to here as the *engineering* coordinate system. All algorithmic work is based on this coordinate system and is converted back and forth as needed. A right-handed system where X points to the front, Y points to the left, and Z points upwards, it is the same system used by other robotics and AV platforms such as ROS.
Rotations are specified using the same Euler angles order as in UE4 in order to minimize discrepancies. Positive rotations are counter-clockwise for all axes.

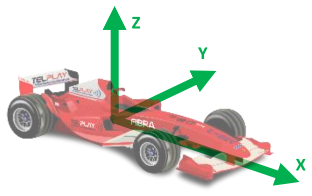


Figure (11): Engineering coordinate system

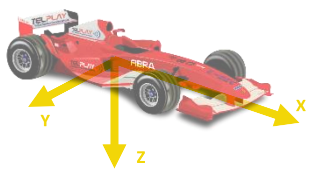


Figure (12): AirSim coordinate system

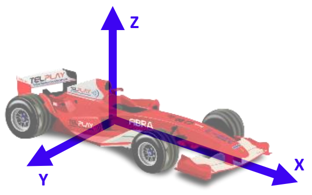


Figure (13): Unreal Engine coordinate system

For convenience, a Python module named "spatial_utils.py" was created, implementing basic functions for conversion between each of the 3 coordinate systems to any of the other ones, and vice-versa (these conversion functions and their inverses are the same). It can convert between the engineering coordinate system to the camera-canonical one as well, for image processing purposes. It also has wrapper functions for polling information and setting states, mostly converting between the AirSim / Engineering coordinate systems and Euler / quaternions, since they are usually the endpoints of this data.

5.1.3. Assets

As mentioned before, art is its own area of expertise which rarely has a lot in common with pure engineering. 2D and 3D assets were acquired from third parties where applicable.

Two race track maps were made available for the purpose manual driving. These maps would have an asphalt road with grass or dirt margins or both, meddling with the tire friction coefficients.

The map for autonomous driving would consist of a large asphalt lot which will allow great flexibility.

Blue, yellow and orange cones are allowed to be placed freely around the map or procedurally as a result of defining a spline along the floor plane. This spline would automatically generate yellow cones on the left side and blue cones on the right side of the path (depending on the defined direction), at predefined distances, both between cones of the same color and those of opposite colors.

5.1.4. Vehicle

The vehicle assets are complex classes built from a set of 3D objects hierarchically linked into a skeletal mesh. It is a cumbersome process which also requires defining physics assets, movement components, suspension and tire data, animation blueprints and more.

Two game modes exist:

- Vehicle Game Mode is intended for manual driving using a keyboard, joystick or wheel and pedal.
- AirSim Game Mode is intended for controlling the vehicle by connecting via Python and the AirSim API, or driving the vehicle using a keyboard only.

Changing between autonomous practice and driver practice is done by altering the game mode (as well as the pawn class) within UE’s world settings.

There would be at least 1 vehicle for each game mode, be it the same or a different one. Each game mode accepts a different Blueprint class as its pawn, and they are not cross-compatible (i.e. same vehicle might need to have 2 versions).

The method would be to get a vehicle by some means (create from scratch, migrate or purchase) and change its parameters so that its dynamic behavior would resemble that of a true electric vehicle.

5.2. Sensing

5.2.1. World

Different sensors have different limitations, and using a multitude of them can compensate for each one's weaknesses. Having a redundancy is even desirable at times, using fusion algorithms to estimate the true values. To sense the environment, the following composition is proposed:

- Cameras: a single camera proved not to have enough coverage for detecting cones on the right and left sides. Therefore, 2 cameras were placed at the front of the vehicle, each one having a resolution of 640 by 360 pixels, and a horizontal FOV of 70 [deg] (both reasonable requirements for commercial cameras).
  The left and right cameras were placed at the front of the vehicle, at a height of 0.5 [m], and offset by 0.5 [m] to the left and right, respectively. They are rotated facing away from the car's front axis by ±40 [deg] (yaw), and then tilted down by 10 [deg] (pitch). Both have zero distortion coefficients, i.e. ideal cameras.
  A Python module named "camera_utils.py" was created, including an AirSimCamera class which calculates the camera matrix and helps with the handling of camera space projections.
- LIDAR: a single 360-degrees lidar was placed at the middle front of the vehicle, between the cameras, at a height of 10 [cm]. Due to a software bug, its capabilities were limited to 3 laser channels – a horizontal one and two additional ones, offset at ±1 [deg] from the horizontal plane. The lidar itself is only looking at ±90 [deg] towards the front, i.e. the front half of the vehicle's heading direction.
- Other: Other sensors can provide both depth and color data in a single sensor, providing that there is enough computation power. Examples are structured light cameras such as MS Kinect or Orbbec Astra (which work poorly outdoors [15]), or stereo vision cameras such as Intel RealSense or ZED Camera. These were not used in the simulation since their emulation is of low fidelity (simulated data is "too good" to reflect the true input coming from these sensors).

5.2.2. Ego-Vehicle

Sensing the vehicle states and implementing a SLAM algorithm is a complex task by itself and is outside the scope of this work. The car's position, rotation and velocity were taken as polled by the API and treated as ground truth.

5.3. Perception

The mission here is to detect the locations of cones and their type. As mono cameras are poor at estimating depth, lidars are poor at estimating colors. Therefore, cameras and lidars were working in conjunction with each other in the following manner:

Pseudo-algorithm (1): perception procedure

```
Images are taken in advance, before further processing.
Lidar detects all front surroundings.
Detections are filtered by height and distance to remove ground and far objects.
Resulting pointcloud goes through a DBSCAN clustering algorithm.
Clusters whose extents are too large are filtered out.
Resulting clusters are fed into a tracker.
Trackers with 3 or more detections in their vicinity are labeled “active”.
Actively tracked cone positions are transformed from lidar into camera frame.
Camera frame position is converted to image pixel space.
A bounding box is opened around each pixel with size depending on distance.
Each “cropped” bounding box is converted from RGB into HSV format.
Each HSV bounding box is masked by saturation and then classified by hue.
```

The pointcloud polled from the lidar will typically contain the detected cones points, along with circular patterns characteristic of the cross section between the laser shots and the ground plane.
Scipy’s DBSCAN is used to cluster the detections, with an algorithm complexity of $O(n \cdot \log(n))$ [16]. In fact, it is the most time consuming task in the entire loop, hence the more noise that can be filtered out before running DBSCAN – the better.
The ground patterns make it difficult to segment the point cloud into clusters. In addition, we only care for discovering the close vicinity of the car. Therefore, all detections that are farther than 8 [m] and closer than 3 [m] are filtered out.
DBSCAN is then used with an epsilon of 0.3 [m] and a minimum of 3 detections per cluster.
Sometimes, the fact that the sensor is not completely level will result in some residue of the ground patterns, looking like arcs from top-view. Their overall extent is much larger than that of a typical cone, and so every calculated cluster that has an extent of more than 0.5 [m] is automatically filtered out. A Python module named “dbscan_utils.py” was created to implement the above filtering, clustering, finding cluster centroids and calculating their extent.

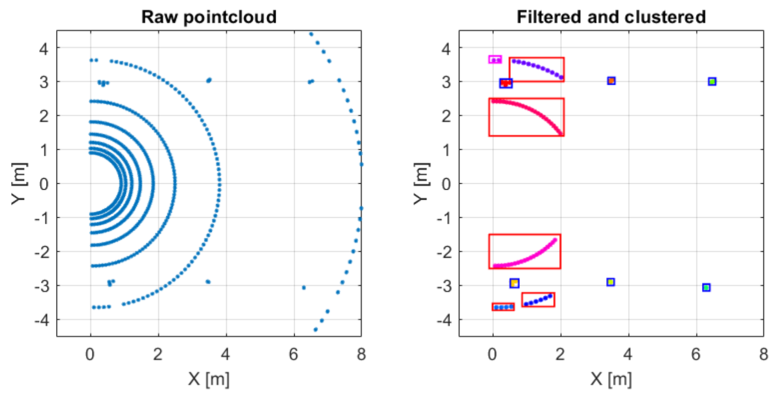


Figure (14): LIDAR pointcloud output in top-view, raw detections vs. filtered by distance and height with DBSCAN clustering

Notice in figure (14) how the large clusters get disqualified while the small ones do not. With that being said, the top-left cluster is not a cone, but circumstantial noise with low extent. It is expected not to pass the needed detection count to be considered active, or at the very least not have its color recognized (remaining as “unknown”).
Next stage is creating trackers. A Python module named “tracker_utils.py” was created to handle tracking, containing an abstract class and an inheriting class for cone tracking. It holds methods for comparing location proximity, storing a moving average, determining the cone color, etc.
Centroids of the clusters resulting from DBSCAN are iterated through, comparing their location to those of any existing trackers. If they are closer than 0.5 [m] to any tracker, it is assumed that they are the same object, and that location is added to the tracker’s moving average. If no tracker fulfills the 0.5 [m] proximity criteria, it is assumed to be a new object and a new tracker instance is created. It is required that a tracker would have at least 3 detections accumulated before it can be labeled as “active”, assumed *not* to be noise.
Every active tracker on the list has its position vector transformed from the lidar frame of reference to the camera frame of reference, denoted V_c. It is then normalized by its Z factor and multiplied by the camera intrinsic matrix, resulting in the $\{x_p, y_p\}$ pixel indices to which that vector points in image space. In other words, we find the pixel which best represent the tracked centroid’s location, taken from the camera’s point of view.
A bounding box is stretched around that pixel by the heuristic formula:

$$W_{bb} = \frac{200}{|V_c|} \quad (17)$$

$$H_{bb} = \frac{100}{|V_c|} \quad (18)$$

with W_{bb} being the width of the bounding box and H_{bb} being its height, in pixels.

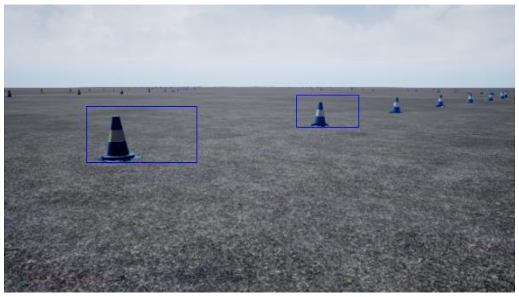


Figure (15): detected blue cones with bounding-boxes on the left camera

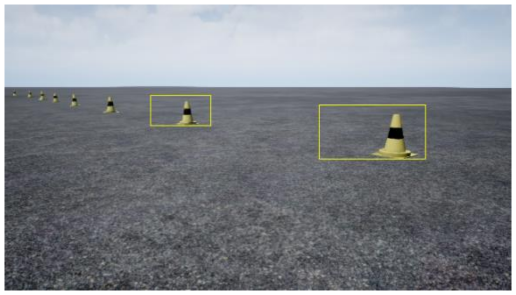


Figure (16): detected yellow cones with bounding-boxes on the right camera

Once that task is completed, each of these bounding boxes is cropped out and converted into HSV format, which works well for color recognition [17]. One of the downsides of this method in this case is that the blue cone and the grey background have similar hue values (see figure (17) below). Therefore, the saturation channel should be used to create a mask, in order to separate between the pixels belonging to the cones and those belonging to the background. Once the relevant pixels are masked, a histogram is computed over their hue channels.
The histogram values are summed through each color's (yellow and blue) typical hue range, and whichever color had more counts is declared as the official cone color. A declared color may change during runtime since closer objects are more likely to be classified correctly (it is also another reason for filtering out far detections in the first place, since an incorrect classification may lead to wrong planning). A cone which had already been classified may change its color, but can never revert back into being tagged as "unknown".

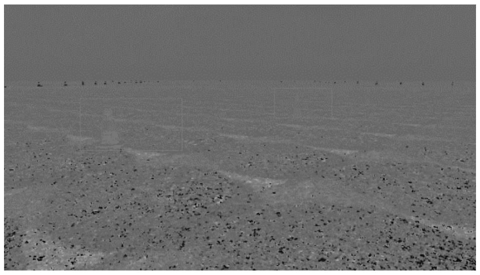


Figure (17): hue channel of left camera, blue cones over grey road background

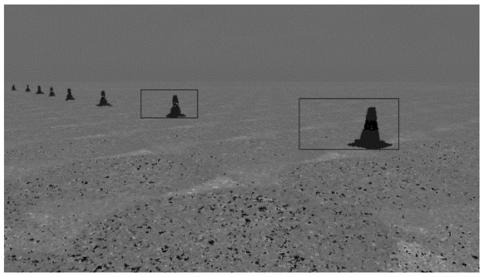


Figure (18): hue channel of right camera, yellow cones over grey road background

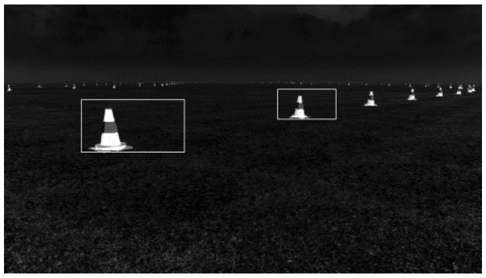


Figure (19): saturation channel of left

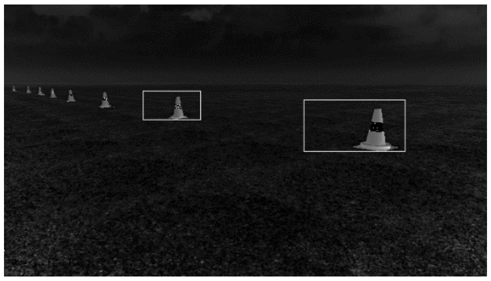


Figure (20): saturation channel of right camera

5.4. Planning / Decision Making

In this work's mission, they are combined into one. The sole mission is to identify the cones and drive by them during the first lap, and then make a faster trip during the second lap as the entire path is known. A Python module named "path_control.py" was created in order to provide pure pursuit and Stanley method classes that are easily executed and maintained.

5.4.1. First Lap (Mapping)

In this round, the car is being throttled at 20% the whole time.

Pseudo-algorithm (2): first-lap loop procedure

```
Set throttle to a constant low rate
Loop while first lap has not ended:
        Acquire camera images
        Acquire lidar pointcloud
        Filter, cluster and track detections
        Classify active trackers by color
        Find most recently discovered blue and yellow cones
        Average their position to get a pursuit point
        Pick the most suitable pursuit point from a list
        Calculate the direction to the chosen point in vehicle coordinates
        Multiply by a coefficient to get the desired steering angle
        Saturate and apply steering angle to vehicle.
```

Before comparing the discovered clusters with existing trackers, the algorithm sorts them in an ascending order by distance. This ensures that the most newly discovered trackers are always the furthest ones away from the vehicle.
With every loop iteration, the algorithm scans the list of existing trackers in reversed order until it encounters the first blue and yellow cones (namely, the newest ones). It then averages their locations to generate a "pursuit point". The calculated point is added to the list of pursuit points only if the distance between it and the previous one is above 1 [m]. This ensures we do not have multiple instances of the same point within the list.
The list of pursuit points is also scanned in reversed order until we encounter a point which is closer than 6 [m] but farther than 2 [m] from the vehicle. This picking is done in order to prevent cutting corners and steering oscillations, since the car will be using pure pursuit with constant speed while navigating to its target pursuit point.
If no such point exists, the car will move straight forward, hopefully finding new points or getting in range for an eligible pursuit point within the desired distances.
In that way, as the car progresses it discovers more and more pursuit points roughly in the middle of the track (unless at sharp turns), constantly changing its direction towards them.

5.4.2. Second Lap (Following)

In every lap, there is a latch trigger on a disk of a certain radius around the track origins. It is activated once the vehicle exits that area, and triggers the “end of lap” once the car returns to that area (namely got “back around” close enough to the starting point). Once the first lap is over, it is assumed that all the cones were detected, or at least that the difference between the amount of yellow and blue detected cones is miniscule. During that round, an implementation of a Stanley method is commenced.

Pseudo-algorithm (3): second-lap loop procedure

```
Separate tracker list into (only active) blue and yellow cones lists
Make sure both lists have the same number of elements
Average the locations of every blue/yellow pair of the same index
Downscale the number of resulting points
Construct a closed cubic spline with a smoothing factor between the points
Loop while second lap has not ended:
        Find the closest point to the vehicle along the path
        Find the heading of the path tangent at that point
        # Calculate the first term of the Stanley method
        theta_e ← tangent_heading – vehicle_heading
        Find the closest distance from the vehicle to the path
        # Calculate the second term of the Stanley method
        theta_f ← arctan(steer_coefficient * distance / car_velocity)
        Combine terms to get the final desired steering angle
        Find the point along the path corresponding to the lookahead distance
        Compute the path’s curvature at that point
        # Calculate desired vehicle speed
        desired_ speed ← max_ speed – speed_coefficient * path_curvature
        Apply desired speed and steering values to the vehicle through controllers
```

The list of trackers is divided into lists of blue and yellow cones, then these lists are trimmed to the length of the shortest one. The locations of the cones from the same index in each list are paired and averaged, resulting in a set of points that should be roughly in the middle of the cone track. The resulting set of points is used as a basis for creating a closed cubic spline. Much effort was put into creating a Python module named “spline_utils.py” so it could manage all the aspects regarding the creation of a path and the query of values relevant for following it. The car uses the PathSpline class in conjunction with the StanleyMethod class in order to guide itself towards the intended route at higher velocities than those of the first lap. Due to the discretization of the spline into points, the type of curvature used for speed calculations is the Menger curvature [18].

5.5. Control

Reference commands to the vehicle controls include the desired car speed and steering.
In this work there was no use of any sort of brakes control. It involves high hysteresis and would probably require a dynamic analysis of the car.
A Python module named "pidf_control.py" was created in order to easily manage both velocity and position control loops, including implementations for feed-forward, derivative on measurement, windup mitigation and more [19].

5.5.1. Throttle

The throttle control is the manipulation of the throttle pedal in such a way that the desired velocity it achieved. An electric vehicle's motor is a 1st-order system and as such requires only a PI controller. Adding an integrator does turn the system into a 2nd-order one, but as long as the I term is kept at reasonable levels then there should not be any overshoot, thus the D term whose goal is to dampen the overshoot becomes unnecessary. This can be shown mathematically.
Let us rephrase equation (4):

$$G_1 = \frac{k_{sys}}{\tau S+1} = \frac{k_{sys}/\tau}{S+1/\tau} \equiv \frac{k_0}{S+a} \triangleq P_1 \text{ (19)}$$

A PI compensator would be of the form:

$$C_1 = k_p + \frac{k_i}{S} = \frac{k_p \cdot S + k_i}{S} \text{ (20)}$$

Using the common closed-loop feedback schema, we achieve:

$$G_{cl1} = \frac{C_1 P_1}{1+C_1 P_1} = \frac{k_0 k_p S + k_0 k_i}{S^2 + (k_o k_p + a)S + k_0 k_i} \text{ (21)}$$

Note how the system ended up as a 2nd form due to the addition of an integrator.
The system step response requirements for the car's speed control were:

- No overshoot ($\zeta = 1$).
- Settling time of 2 [sec] (into a corridor of 2% from the step size).

The value of ω_n was rounded up from 1.95 to 2, calculated using the formula:

$$\omega_n \approx \frac{-\ln\left(\frac{corr}{100}\right)}{\zeta \cdot ST} \text{ (22)}$$

With $corr$ being the step size settling corridor percentage, and ST is the settling time in seconds. The raw control coefficients are calculated by coefficient comparison between the desired characteristic polynomial and the existing one:

$$k_i = \frac{\omega_n^2}{k_0} \text{ (23)}$$

$$k_p = \frac{2\zeta\omega_n - a}{k_0} \text{ (24)}$$

The values of k_0 and a would be extracted from a bump test to be conducted on the vehicle inside the simulation environment.
In addition, a feedforward term k_f will be added as the inverse value of k_{sys} in order to improve performance. This minimizes the work load done by the compensator.

5.5.2. Steering
This would mean to control an actuator for pointing the front wheels left or right as in the bicycle model, according to some Ackermann-bicycle conversion schema.
In the simulation, it is assumed that the angle provided to the car by the API setter function is already in bicycle model terms.
A position control is a 2nd-order system. If a rotary electric motor actuator is used to turn the wheels (which is often the case), then it can be thought of as a simple 1st-order system combined with an integrator:

$$P_2 = \frac{1}{S} \cdot P_1 = \frac{k_0}{S(S+a)} \text{ (25)}$$

The compensator in this case would be a PD controller. Adding an integrator would impair the transient response of the system, while adding no benefit for the steady state.
As a side note, a position control integrator is used in applications where a steady, accurate ending point is desired and the system holds enough static friction to prevent it from being achieved with a PD controller alone. In the car, however, there is constant wheel movement and direction change, along with vibrational dithering.
The compensator would hence be of the form:

$$C_2 = k_p + k_d s \text{ (26)}$$

The closed-loop schema in this case would not be the common one, but that which implements “derivative on measurement”. Mathematically, it prevents the appearance of a first order term in the transfer function’s numerator. Practically, it also prevents the phenomenon called “derivative spikes” when a new step reference is introduced [18].

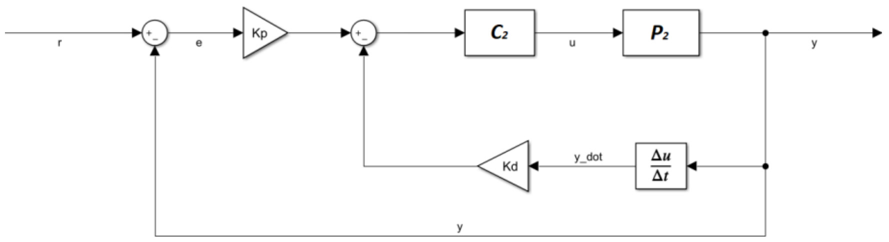


Figure (21): a closed loop model of the 2nd-order system with derivative on measurement.

Adhering to that schema, the closed loop expression would be:

$$G_{cl2} = \frac{k_p P_2}{1+C_2 P_2} = \frac{k_0 k_p}{S^2+(k_o k_d+a)S+k_0 k_p} \quad (27)$$

In the simulation, issuing a steering command and then immediately polling for the steering angle of the vehicle results in getting the same value consecutively.
The assumption was that the angle change is instantaneous, and thus there is no dynamic model to identify.
Therefore, to demonstrate position control, an actual plant needed to be emulated. Namely, that the reference signal should first go through a compensator, then enter an emulated plant system, and only the final output should be set in the simulation by the API.
The emulated system was intentionally chosen to be slow-settling in a closed-loop:

$$P_2 = \frac{10}{S(S+4)} \quad (28)$$

$$P_{cl} = \frac{10}{S^2+4S+10} \quad (29)$$

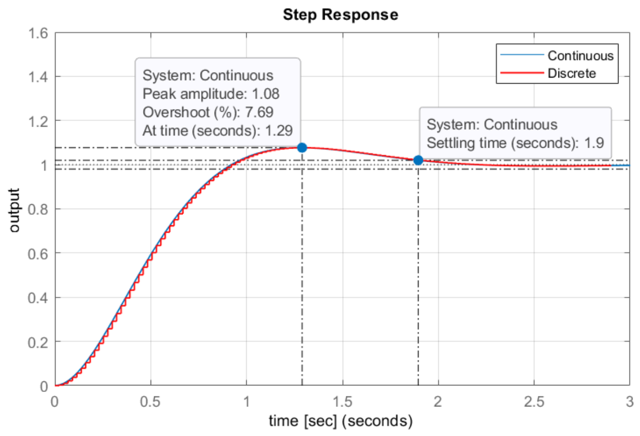


Figure (22): closed-loop step response of the arbitrary system chosen to represent the steering mechanism

The emulation itself was done by discretizing the plant P_2 using bilinear transform (Tustin method [20]) and then converting it to a difference equation using inverse Z-transform:

$$P_d(z) = \frac{k_0 t_s^2 Z^2+2k_0 t_s^2 Z+k t_s^2}{(2at_s+4)Z^2-8Z+4-2at_s} = \frac{y(z)}{u(z)} \quad (30)$$

$$y_k - 8\alpha y_{k-1} - \alpha(2at_s-4)y_{k-2} = \alpha k_0 t_s^2 u_k + 2\alpha k_0 t_s^2 u_{k-1} + \alpha k_0 t_s^2 u_{k-2} \quad (31)$$

with t_s being the sample period, and $\alpha = (2at_s+4)^{-1}$.

A Python class named "DiscretePlant" was created to generate transfer functions of this form parametrically, accepting desired values for t_s, k_0 and a. It also supports the iteration of the difference equation using a new input value, and provides the necessary output as if it was a real plant. Later, a shared-memory multiprocessing method was added.
The desired response was to reduce the settling time to about 0.05 [sec], and bring the overshoot down to about 5% using the formula:

$$OS = \exp\left(\frac{-\pi\zeta}{\sqrt{1-\zeta^2}}\right) \quad (32)$$

which eventually yielded $\omega_n = 113.4$, and $\zeta = 0.69$. Since further tuning will be required after discretization, for the sake of using simple numbers ω_n was rounded to 100, while ζ was brought down to 0.65 in order to round the value of the expression $2\zeta\omega_n$ to 130.
This results in a desired target transfer function of:

$$G_d = \frac{10000}{S^2+130S+10000} \quad (33)$$

to which the raw control coefficients were designed to match:

$$k_p = \frac{\omega_n^2}{k_0} \quad (34)$$

$$k_d = \frac{2\zeta\omega_n - a}{k_0} \quad (35)$$

Let it be reminded that a state space control is richer than using the classical approach. It can utilize MIMO to make use of more information and generate better control.

5.6. Final Program Architecture

The main loop of the program runs 2 loops as separate functions one after the other.

Pseudo-algorithm (4): complete loop procedure

```
While not finished first lap:
    Detect cones using lidar and tracker
    Classify cones using camera
    Generate pursuit points from classified cones
    Generate reference signals to follow target point using pure pursuit
    Send compensated steering signal to the vehicle
Create a spline from all the cones detected during the first lap.
While not finished second lap:
    Generate reference signals to follow spline using Stanley method
    Send compensated control signals to the vehicle
Bring throttle down to zero and apply full brakes
```

6. Results

6.1. Environment

6.1.1. Maps

Three maps are provided in the project. The default map (for autonomous driving) was created by using the UE4 simple vehicle example map. All meshes were removed from it except for the ground floor, resulting in a large flat lot. Two materials were created using generic asphalt textures found online. The textures were assimilated into materials with added macro-variation to mitigate tiling repetitiveness. Some parameters can be changed by the use of material instances.

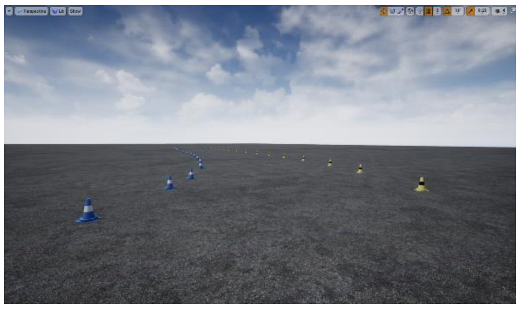


Figure (23): ground view of the autonomous driving map with visible cones

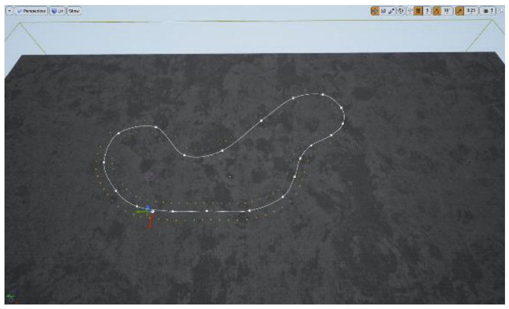


Figure (24): aerial view of the autonomous driving map with cone track

There exist 2 additional maps, acquired from the Unreal Marketplace for the purpose of simulating human driver racing in case this need should arise. The full kits supply additional assets that are free to use such as tire walls, structures, trees, generic cones, PBR materials, etc.

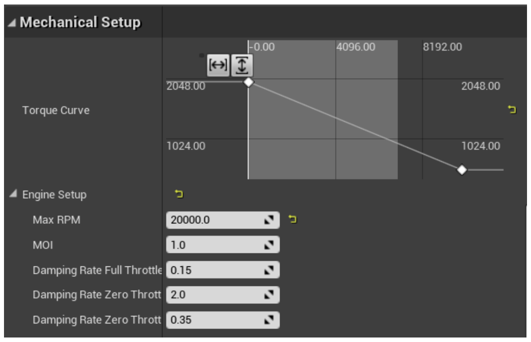
Figure (25): aerial view of the larger race track

Figure (26): aerial view of the smaller race track

Any of these maps may be altered, or new ones may be created using the kits.

6.1.2. Vehicles

After several attempts at exporting the BGU and Technion race cars into blender and importing them back into UE4, the decision was to purchase the (only) race car model available at the Marketplace. This asset had its settings heavily modified and renamed "FormulaOne". The most important modification was to change the car's torque curve to behave like an electric motor, similar to figure (2). Arbitrarily, a stall torque of 2000 [N*cm] was chosen, reducing linearly to 500 [N*cm] at 10,000 [RPM].



Engine Setup

Max RPM	20000.0
MOI	1.0
Damping Rate Full Throttle	0.15
Damping Rate Zero Thrott	2.0
Damping Rate Zero Thrott	0.35

Figure (27): a small portion of the vehicle's mechanical setup in UE4

Although only the "FormulaOne" car and AirSim's default SUV car can be used in the AirSim game mode, the other two cars can be driven in Vehicle game mode. Tweaking of parameters is recommended since arbitrary values were used.

Figure (28): autonomously drivable AirSim SUV

Figure (29): autonomously drivable race car

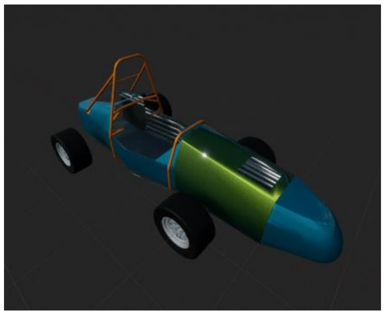


Figure (30): manually drivable BGU car

Figure (31): manually drivable Technion car

6.1.3. Cones

The blue, yellow and orange cones, as well as the spline-based path constructor Blueprint were salvaged from the FSDS project and lightly adapted for the use of this work.
Their size, color and UV mapping already adhere to the FSAE standards and are readily available. Spaces between cones along the path can be changed as desired.
An arbitrary splined course path was constructed within the autonomous map, with distances of 6 [m] between left and right cones, and 4 [m] between cones of the same color along the path spline.
To broaden the ability of quickly generating more assets in the future, the Marketplace package of automotive materials was added to the project as well, along with the UE4 starter content.

6.2. Perception

A series of 10 full runs was made to gather the data for the analysis.

6.2.1. Execution Time

The first round loop rate was set to be 10 FPS, due to a relatively high number of calculations within each iteration. Execution time per iteration averaged to 0.037 [sec]. This value, of course, depends on the computation platform.

6.2.2. Cone Detection

It must be acknowledged that misdetections and false alarms are possible in the real world. Misdetections are considered as cones that were not picked up at all during a run. Since a lidar is used along with a relatively slow drive compared to the sensor rate, it is not surprising that no misdetections occurred during any run.
False alarms are considered to be non-activated tracker objects, i.e. with less than 3 detections within a 0.5 [m] proximity radius. Even though they were encountered during development, eventually zero false alarms were collected during all runs, which may be attributed to effective parameter tuning. With 89 cones from each color along the track, 1780 cones were analyzed, each one logging numerous detections.
The common speed the car was traveling under a 20% throttle command was 4.5 [m/s]. DBSCAN clustering was done between 3 and 8 meters from the front of the vehicle, leaving a 5 [m] window for cone detection (assuming a straight-line travel). Under these conditions, it is estimated that an average of 11 detections will be achieved per cone.
Time synchronization, car pose estimation, sensor noise and clustering all induce detection errors, therefore each consecutive detection of a cone enters a calculation of a moving average. Since the car is constantly changing its position and heading, a squared error statistic between detected positions and the known ground truth was calculated, rather than separating the errors into x and y axes.

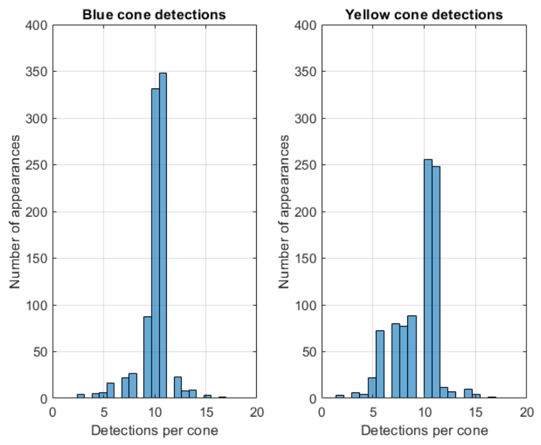


Figure (32): distribution of the number of detections for every cone

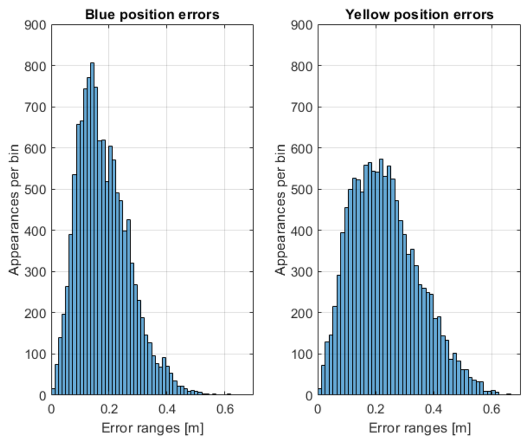


Figure (33): distribution of the squared error between known positions and every

As shown in figure (32), the highest amount of detections per cone is indeed 11, with follow-ups at 10 and/or 12 detections, as expected.
Blue cones had overall 12,616 detections while yellow cones had 12,712 detections, whose squared errors shown in figure (33) may or may not suggest a Rayleigh-like distribution.

6.2.3. Cone Classification

After being detected, cones may also be misclassified, namely being assessed with the wrong color (or no color at all). This behavior was observed to be more prominent the farther the cones were from the vehicle. This makes sense, since far targets tend to be less in focus by the camera and more sensitive to lidar-camera synchronization issues, especially while the vehicle is turning. On top of that, far targets are smaller and more prone to pixel quantization errors.
Over all runs, the yellow cones experienced 207 misclassifications, while the blue ones only had 51. All misclassifications were attributed to the “unknown” type (and not a determined false color), which may point to the fact that the sensor time sync is not optimal. Sensor data is polled from the simulation through the API and does show some delays especially during turns, regardless of the polling order or attempting to use multiprocessing techniques.
Since the underlying code is obscure, some mitigating actions were taken such as widening the cropping window and compensating for the travel time of the vehicle when processing the clustered centroids.
The numbers above refer to the system after the said improvements were applied.

6.3. Planning

6.3.1. Pure Pursuit

Pursuit points are created on the fly and do not wait for the discovery of parallel blue/yellow cone pairs, which means that a pursuit point might fall on the diagonal between distant blue and yellow cones. This can be seen clearly in figure (34); the green circles indicate the average points between parallel blue and yellow cones (downscaled by a factor of 2 for visibility). At times, there will be a pursuit point right in the center of these circles, while other times these circles would be empty (usually at tight corners). This indicates that there was a consecutive detection of at least 2 cones of the same color prior to the detection of a cone of the other color.

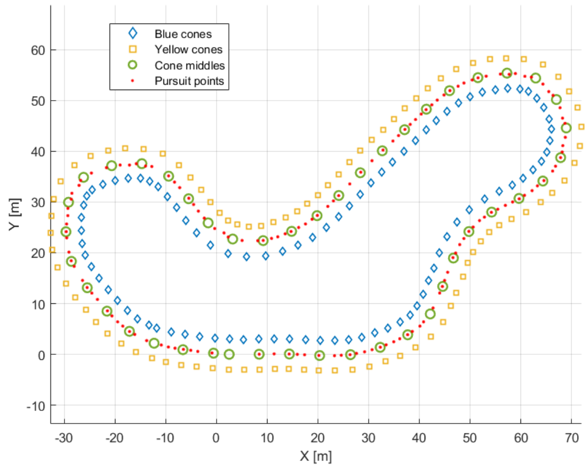


Figure (34): blue and yellow cone locations, their midpoints and pursuit points in between

6.3.2. Stanley Method

In addition to downscaling the midpoints density, the generated spline is smoothed (spline does not have to pass directly within every point, but at its vicinity). Both of these actions are meant to minimize fluctuations along the path in case something went wrong during the mapping round.

The Stanley method has been working as expected, following the designated course with acceptable precision (not knocking off any cones). It does lower the vehicle reference speed as path curvature at the 5 [m] lookahead point rises. The higher the curvature, the lower the speed command sent to the vehicle controller in order to be able to take the turn. These parameters and more can be controlled via the StanleyFollower class and will affect the time it takes the car to complete the second lap, at the cost of navigational stability.

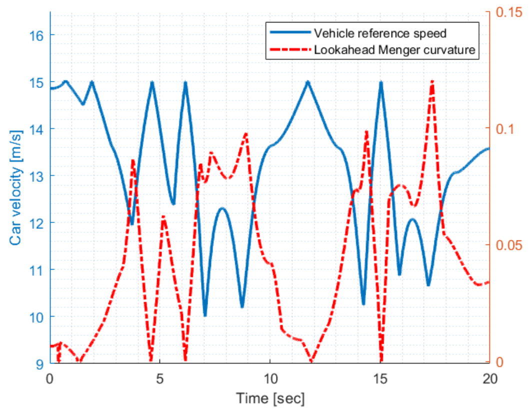


Figure (35): car reference speed command vs. curvature at the lookahead index

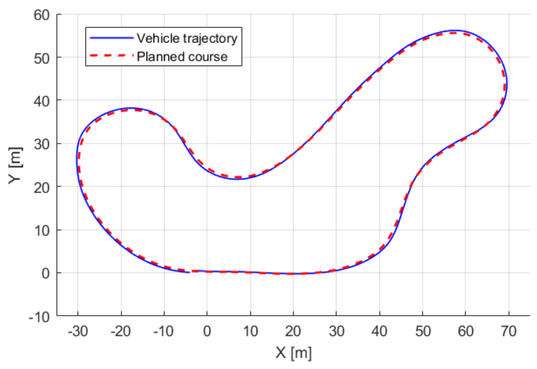


Figure (36): planned course vs. vehicle following trajectory

6.4. Control

6.4.1. System Identification

The car went through a bump test within the simulation to get its motor coefficients extracted. There exists an unexplained and unsolvable change of behavior when crossing the 8 [m/s] boundary, therefore the bump test was kept at relatively lower speeds, starting at 10% throttle for 6 [sec] and then raised to 30% for an additional 12 [sec].

One must bear in mind that the simulated vehicle is not only a simulated motor, but a series of differential equations solved in real time, including synthetically emulated forces such as gear friction, arbitrary linear and angular movement drags, and more.

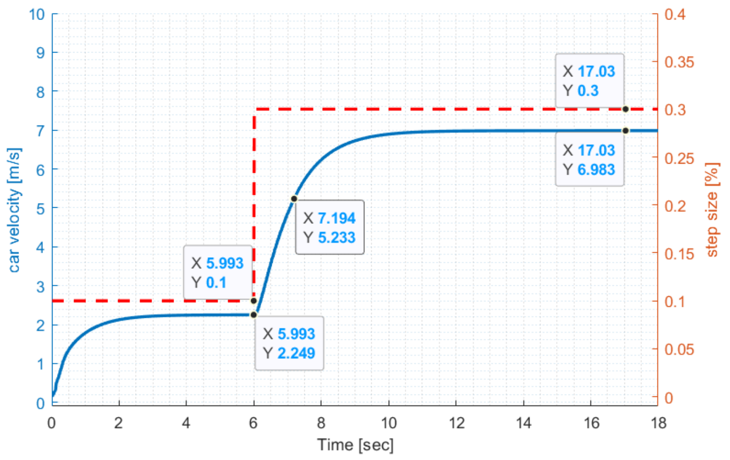


Figure (37): bump test result with critical values marked

It is easy to deduce from figure (37) that the car motor's $k_{sys} \approx 23.27$ and $\tau \approx 1.2$, Which in turn implies that $k_0 \approx 19.4$ and $a \approx 0.83$.

6.4.2. Velocity Control

All the control recordings and experiments were conducted at the simulation in 100 [Hz]. Difficulties with accurate controls were expected due to hysteresis, as the signals are saturated between 0 and 1, leaving the "slowing down" section to friction alone.

The values extracted in the previous section yield:

$$G_1 = \frac{23.27}{1.2S+1} \equiv \frac{19.4}{S+0.83} \triangleq P_1 \quad (36)$$

Control requirements lead to the characteristic polynomial of $S^2 + 4S + 4$, which in turn leads to control coefficients of magnitude:

$$k_i \approx 0.2 \quad (37)$$

$$k_p \approx 0.16 \quad (38)$$

Since the closed loop had a first order term added to its numerator due to the introduction of an integrator, it was expected not to provide the exact desired response. Additional experimentation had led to the finding of the coefficients: $k_p = k_i = 0.1$ to be satisfactory. Albeit, injecting these values into the discrete controller implementation resulted in an oscillating response of borderline stability (which often occurs with high integration coefficients and no integrator limitation in a discrete control scheme).

It was decided to add a feed-forward term, k_f, to lower the gap that the controller needs to compensate for. This generally lowers the value of the other coefficients, resulting in: $k_p = k_i = 0.01$, $k_f = \frac{1}{k_{sys}} \approx 0.043$. In addition, the maximum accumulation of the integrator contribution was restricted to 0.01% throttle to attenuate the oscillations.

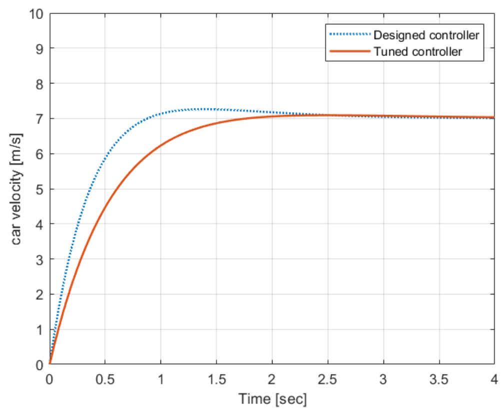


Figure (38): car speed step response to controllers in MATLAB

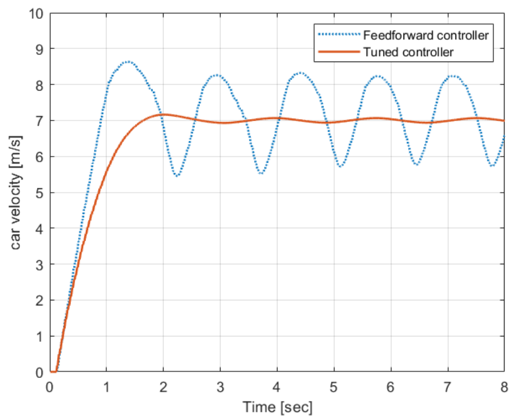


Figure (39): car speed step response to controllers within the simulation

Using the feed-forward did create a slight overshoot that is hard to mitigate. Still, adding a derivative term is tricky since deriving a velocity signal in practice is usually a noisy process and is often more trouble than its worth.

6.4.3. Steering Control

The requirements of position control were much stricter than those of the velocity control, demanding a settling time of 0.05 seconds. Considering the laps loop rate, this would mean to only have (at best) 5 samples in between a step entry and the convergence point, effectively causing the discrete plant emulator to diverge.

In order to mitigate this, the plant was run at a 1000 [Hz] on a parallel process, having inputs delivered to it and outputs polled from it at 10 or 100 [Hz] using shared memory. For this purpose, a class named "SteeringProcManager" was created to manage the sharing of data between different running processes with different scopes.

Since G_{cl2} and G_d are known from section 5.5.2, the control coefficients are:

$$k_p \approx 1000 \quad (39)$$

$$k_d \approx 12.6 \quad (40)$$

Further tuning within the emulated system showed that a slightly increased damping coefficients yields better results for the compensated closed-loop.

The resulting value of $k_d = 15.0$ was eventually used for steering inside the simulation during both the first and second laps.

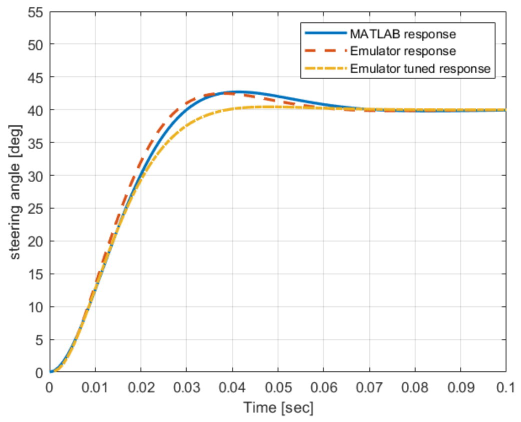


Figure (40): different step response curves for a controlled steering plant.

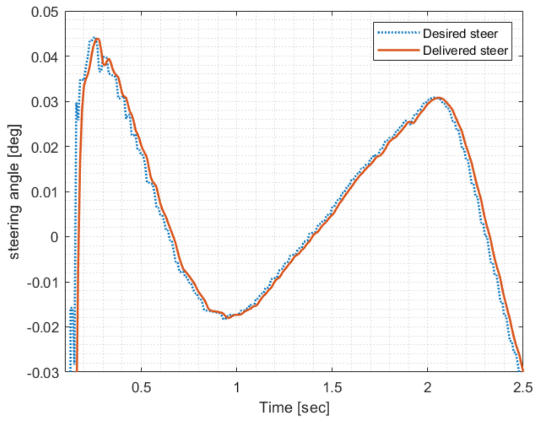


Figure (41): selected zoom over steering response during the second round.

It is important to note that a steering response should typically take more time to converge than 0.05 [sec], yet a human driver initiates the turn of the steering wheel before entering the curve, which could be imitated by utilizing predictive methods.

In addition, experiments have found that the steering command sent to the car does not cause an immediate change of the angle (as previously assumed), thus using the emulated plant added unnecessary phase delay to the system.

The system could not be modeled since setting a desired angle still caused every consecutive "get" command to return that set value, even though there is clearly a visible transient response of the turning wheels within the simulation.

7. Discussion

7.1. Capping

This work indeed adheres to all the intended goals that were set for it. It covers most aspects of the autonomous life cycle, while being able to separately run and test each sub-task.
The vehicle senses its environment using a lidar and 2 cameras, and fuses the information gathered from them to estimate the cones' position relative to the car (and by so also relative to the world), along with classifying their color. It is a good time to remind that without a good positioning system, which was not in the scope of this work, accurately transforming the cone locations from vehicle to map frames will be difficult. A combination of GPS, IMU and dead-reckoning systems is recommended.
Color classification was done by simple HSV space comparisons to differentiate between the cones and the background, and then pick which color best suits the pixels belonging to the cones. In that sense, the simulated environment is relatively sterile. Moving to a real-world camera input might introduce surface tone irregularities, background noise (such as objects in different colors) or a change in lighting conditions, which could challenge this method's robustness. A more modern, deep-learning based approach is one to consider.
After detecting the cones, pursuit points are generated and followed via pure-pursuit.
In retrospective, this part proved to be the weakest link in the cycle. It relies on the fact that all stages preceding it worked flawlessly, which might not always be the case. Possible alternatives would be the construction and following of a short open spline advancement according to past curves, or the use of probability trees to evaluate the best forward path [21].
This loop continues until a full lap is completed and a path is constructed of the middle points between detected blue and yellow cones. This may as well not be the optimal solution, since there may be a shorter path to complete the lap (closer to the inner-parts of turns).
This requires the solving of an optimization problem, probably once all points are known.
The following of this path is done smoothly by the Stanley method, although papers of other steering methods exist, heuristic or not, including Model Predictive Control.
Even though the author has full proficiency with ROS, it was not used in this work with the purpose of minimizing required dependency installations and opening development to Windows platforms as well. This is also the main reason to use AirSim rather than Carla, which is superior to it when it comes to flexibility, support, active development and dedication to cars. AirSim is smaller in size, easy to install and works "out of the box". However, using the end-to-end system on a new computer does take a certain amount of setup time installing UE4, downloading the UE project, cloning the Git repository and installing the Python dependency modules. It is also recommended to use Pycharm Community Edition and mark the "utils" folder as a source folder, otherwise it would have to be added to the Python PATH environment variables directly through code.

7.2. Future Work

There is always more work to be done in order to broaden the capabilities of any system. Some would be in the algorithmic department, and some would require knowledge in operating UE4 on several levels.

First thing would be to come up with a solid localization and mapping technique. One of the difficulties would be the simulated fidelity of sensors. While simulated IMU and GPS signals are not similar enough to real ones, other real-life sensors such as encoders do not exist at all in the plugins and are hard to program. It would appear that some algorithms will have to be developed on a physical vehicle, even if not in racing conditions.

Any of the proposed methods are subject to augmentation and improvement, for example the addition of braking control to quickly slow-down when entering a turn, thus enabling higher cruise speeds during the rest of the path.

Other work would be to study and investigate different approaches to the same tasks proposed in this work. These may include cone recognition neural networks (which have proven effective in some implementations, yet on a relatively small dataset [22]), Model Predictive Control for path following, Rapid Tree Planning for mapping and navigation and more.

Even if developed on a generic model, eventually the simulated car itself must look and feel the same as the car on which the algorithms will be used. This can be a tricky task to perform, requiring both a UE4 operator to tune parameters and a real-car driver to provide the feedback, or the design of a series of tests and sensors to be recorded both on the real and the simulated car, and tuning the simulation to behave as in the real world.

The algorithms, be as they may, would eventually have to be robust for a variety of situations. It might be a good idea to test them on a simulation environment with added variance, such as sun angles and color, cloudiness and luminosity, rain and puddles, ground rubble (cigarettes, road spots and cracks, grass bulges etc.), background objects (colorful clothing, buildings, walls, fences etc.) and more.

As one can see, the limits of future work are bounded only by imagination.

References

1. Kabzan, J. et al. AMZ Driverless: The Full Autonomous Racing System, ETH Zurich, May 2019.

2. Messikommer, N.; Schaefer, S. Cone Detection: Using a Combination of LiDAR and Vision-Based Machine Learning, ETH Zurich, November 2017.

3. “AirSim”, Microsoft, 2021, https://microsoft.github.io/AirSim/api_docs/html/

4. “Carla Documentation”, CARLA Simulator, 2021, https://carla.readthedocs.io/en/0.9.12/

5. “Motor constants”, Wikipedia, Wikimedia Foundation, September 2020, https://en.wikipedia.org/wiki/Motor_constants

6. “Rotation formalisms in three dimensions”, Wikipedia, Wikimedia Foundation, May 2021, https://en.wikipedia.org/wiki/Rotation_formalisms_in_three_dimensions

7. Mallick, S. “Geometry of image formation”, LearnOpenCV, February 2020, https://learnopencv.com/geometry-of-image-formation/

8. Paden, B.; Cap, M.; Yong, S.Z.; Yershov, D.; Frazzoli, E. A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles, IEEE Transactions on Intelligent Vehicles, April 2016.

9. Hoffman, G.M.; Tomlin, C.J.; Montemerlo, M.; Thrun, S. Autonomous Automobile Trajectory Tracking for Off-Road Driving: Controller Design, Experimental Validation and Racing, Stanford University, July 2007.

10. “Choosing a Simulator”, Autodrive, 2018, https://autodrive.readthedocs.io/en/latest/chapters/simulator/comparison.html

11. Behrendt, T. Implementation of a Driving Simulator Within a Formula Student Team, Monash Melbourne, 2017.

12. Hirshberg, T.; Zadok, D. Self Driving Simulation for Autonomous Formula Technion Project, Technion Haifa, 2018.

13. “Formula Student Driverless Simulator”, Github, March 2021, https://github.com/FS-Driverless/Formula-Student-Driverless-Simulator

14. Knafi, E. “Reinforcement Learning on Autonomous Race Car”, Github, 2021, https://github.com/eliork/Reinforcement-Learning-on-Autonomous-Race-Car

15. Kuan, Y.W.; Ee, N.O.; Wei, L. S. Comparative Study of Intel R200, Kinect v2, and Primesense RGB-D Sensors Performance Outdoors, IEEE Sensors Journal Vol. 19, October 2019.

16. Ester, M.; Kriegel H.P.; Sander J.; Xu X. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise, Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining, 1996.

17. Qie, L. et al. Cone Detection and Location for Formula Student Driverless Race, 6th International Conference on Dependable Systems and Their Applications, January 2020.

18. "Menger Curvature", Wikipedia, Wikimedia Foundation, May 2021, https://en.wikipedia.org/wiki/Menger_curvature

19. Beauregard, B. "Improving the beginner PID", Wordpress, April 2011, http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/

20. "Bilinear Transform", Wikipedia, Wikimedia Foundation, January 2021, https://en.wikipedia.org/wiki/Bilinear_transform

21. Feraco, S.; Luciani, S.; Bonfitto, A.; Amati, N.; Tonoli, A. A local trajectory planning and control method for autonomous vehicles based on the RRT algorithm, 2020 AEIT International Conference, November 2020.

22. Knafi, E. "Cone Detector tf", Github, 2021, https://github.com/fediazgon/cone-detector-tf