# Perception, Planning and Control System for Automated Slalom with Porsche Panamera

**Jiri Zahora, Michal Sojka, and Zdenek Hanzalek**

*Czech Technical University in Prague*
*Jugoslavskych partyzanu 1580/3, 16000 Prague, Czech republic (E-mail: jiri.zahora@cvut.cz, michal.sojka@cvut.cz, zdenek.hanzalek@cvut.cz)*

**This is a draft version of the paper J. Zahora, M. Sojka, Z. Hanzalek: Perception, Planning and Control System for Automated slalom with Porsche Panamera, 38th FISITA 2021 World Congress, Prague 2021. https://doi.org/10.46720/F2020-ACM-064**

**ABSTRACT:** Automation of various driving tasks is an important aspect of today's car industry. Not requiring drivers to simultaneously concentrate on several tasks can lead to increased safety. Carmakers use automation to improve quality and reproducibility of car testing, independently on particular driver. In this paper, we focus on automation of one test scenario performed by carmakers – a slalom drive. We present hardware and software architecture that allows a stock Porsche Panamera car to drive automatically through a slalom track formed by traffic cones. Our solution relies on few additional sensors like a differential GPS, a camera, and a 16-layer LiDAR. Our control algorithms run on the NVIDIA TX2 platform and are integrated using Robotic Operating System (ROS) middleware. The perception layer detects cones in camera images using a Support Vector Machine and Histogram of Oriented Gradients feature vectors. Interfacing with the car Electronic Control Units (ECUs) is made via CAN and FlexRay buses. We validate our system in experiments with a real car.

**KEY WORDS:** automated car, slalom, cone detection, robotic operating system, trajectory planning

## 1. Introduction

Robtic cars as fully self-driving vehicles will show up in the future, but even nowadays, new automated functions are already helping drivers daily. Advanced driver-assistance systems (ADAS) already help drivers in many on-road traffic situations, preventing or mitigating car accidents. Driving automation is also useful in other areas. For example, testing new car prototypes requires a high number of test drives to be conducted by professional drivers, especially in premium cars, which are known for their excellent performance in extreme driving conditions. Automating this kind of testing can lead to the significantly easier development of advanced control algorithms thanks to the reproducibility of the test drives.

In this paper, we show an automated vehicle platform for software prototyping and demonstration. The core of the platform is a series production car. We extend it with add-on sensors, and a control system, and we validate it by driving a slalom between traffic cones. Our prototype aims to close the perception/planning/control loop and demonstrate its behavior. In other words, we focus on having a complete working application rather than on improving or optimizing just a single aspect of such a complex application.

The Porsche Panamera car (see Fig. 1) is instrumented by a control system communicating with the car via CAN and FlexRay buses. We send steering and speed commands to the actuation-related Electronic Control Units (ECUs) and read signals from internal car sensors, such as the odometry. In addition to the internal sensors, we equipped the car with an external monocular camera, LiDAR, and a differential GPS (DGPS). All the computations are carried out on an NVIDIA TX2 embedded computer.

A Kalman filter fuses the DGPS position data and internal car signals to obtain a robust estimation of the car's heading. We detect the positions of traffic cones by combining data from the camera and LiDAR. The cone positions are stored in a map, which is then used to plan the trajectory.

Overall, our system is a universal, lightweight, and compact. It is a holistic control system integrated with the car infrastructure and fine-tuned in outdoor experiments on a complex case study.


Figure 1. Automated car slalom

The paper structure follows: The overall goal, concept, and architecture are described in Section 2. Section 3 details the individual blocks of our control system, namely the cone detection algorithm, localization of the car and cones, construction of the cone map and the trajectory planning and tracking. The results of the experimental drives in the outdoor environment are analyzed in Section 4. Finally, the outcomes of this paper are discussed in Section 5.

## 1.1. Related work

The automotive community distinguishes six levels of automation (**levels_of_automation**) from no automation (level 0) to a fully self-driving vehicle (level 5). Nowadays, most of the available cars offer assistive technologies at level 2, such as adaptive cruise control or a lane-keeping assistant. These technologies are covered in many papers **acc**, **emergency_braking**, **driver_decision** Vehicles with advanced automated functions up to fully self-driving cars start to appear in the news, but only for well-defined use cases. For example, the well-known Tesla, or Audi with the recently presented automatic parking or traffic jam autopilot **audi_selfdriving** Compared to the parking and highway use cases, the cone slalom is an appropriate use case for university research and development teams. It does not require large experimental space, keeps developers reasonably safe, and still stresses the developed system in many ways. Our use case falls between levels 2 and 3 because the driver still needs to oversee whether the system works correctly and act accordingly, if not.

All autonomous cars are distributed systems with a nearly similar architecture **reference_architecture**, **distributed_architecture**, **behereFunctionalReferenceArchitecture2016** our autonomous car is not an exception.

Experimenting with a real vehicle is a time-consuming and resource-demanding activity. Many applications can be tested in virtual simulations **airsim** or with the use of available datasets **datasets** However, experimenting

with a real car in a real environment allows us to gain more widely applicable knowledge.

To encourage the development of autonomous cars in academia, universities organize various racing competitions. A team of our students regularly participates in the F1/10 competition **okelly2019f110** Vehicles used in other competitions were also an inspiration for us. A beautiful example is an autonomous formula developed at ETH Zurich for racing in Formula Student competitions **ETH_driverless** or vehicles developed for different competitions such as **zeus** and **caroline** Inspiration can also be taken from the Autoware.AUTO project **autoware** the open-source software for autonomous driving based on ROS. Some universities provide courses introducing the autonomous driving problem **deep_learning_course**

One can turn an arbitrary vehicle into an autonomous car by mounting an additional steering and braking system to the steering wheel and pedals. For example, the system **ab_dynamic** implements a robust path following, which is targeted at repetitive car testing, similarly as our platform.

Many autonomous/automated driving projects require massive computational power. We try to compose all essential functions needed for the automated driving task into one cheap ECU. This allows us to easily benchmark the platform's performance and reliability without a trunk full of powerful servers.

## 2. Problem and architecture overview

### 2.1. Goals of the platform

The goal of this work is to drive a car through a traffic cone slalom. The environment is a flat obstacle-free surface with an arbitrary number of cones placed approximately in a line, not necessarily aligned. The drive starts with the car at one end of the line of the cones heading towards the cones. After driving through the cone slalom (so-called *slalom part*), at the end of the line, the car turns around the last cone (makes a *U-turn*) and drives through the slalom part back. Then, it makes another U-turn, and the whole process repeats indefinitely.

To achieve this goal, we need to solve several sub-problems. First, the cones need to be detected in the camera images. This is called *cone detection* in the following text. Then, the positions of the cones have to be transformed from image coordinates to global (world) coordinates (*cone localization*). The localized cones are stored in the map, where the accuracy of the positions is improved as the car arrives closer (*cone mapping*). *Car localization* refers to the algorithms for determining precise vehicle position. *Trajectory planning* and *tracking* deal with generating a trajectory for the slalom and the U-turn and driving the car along this trajectory. The *interfacing of sensors and the vehicle* needs to be solved as well.

### 2.2. Hardware and instrumentation

The central computation unit in our platform is an *NVIDIA TX2* embedded computer. Cones are detected from images taken with a single monocular *Basler ace*. We use the camera with a wide-angle lens to detect the cones that are close to the car. A multi-layer *Velodyne LP-16* LiDAR is mounted near to the camera. We use its data to localize the cones because computing the cone distance solely from a monocular camera is highly inaccurate.

We employ a differential GPS (DGPS) composed of two Trimble BX982 receivers to localize the car. The first, static, GPS receiver forms a base station. The second one, called a rover, is mounted on top of the vehicle and measures its position, receiving corrections from the base station. The rover can be equipped with a second antenna to measure the car's heading, but we do not use it because the heading was often inaccurate due to the car being too short. To improve the car localization accuracy, we use the data provided by the car ECUs – namely, the longitudinal and angular speeds and the steer commands.

All the sensors are sampled at 10 Hz because the TX2 computer cannot perform all the computations faster.

We use a Vector VN8900 computer that interacts with the in-car FlexRay bus. It is used merely as an interface between TX2 and the onboard network because the TX2 computer supports only CAN bus and not FlexRay.

### 2.3. Architecture

The overall functional architecture of our platform is shown in Fig. 2. The depicted blocks are responsible for solving different sub-problems, as outlined above.

We use the Robot Operating System (ROS) as the software platform. Each of the blocks, called a *ROS node*, runs in a separate Linux process. ROS implements peer-to-peer communication between the different processes via TCP/IP sockets. Besides the main components shown in Fig. 2, we adopt 3$^{rd}$ party ROS packages implementing sensor drivers or providing data recording and visualization capabilities.

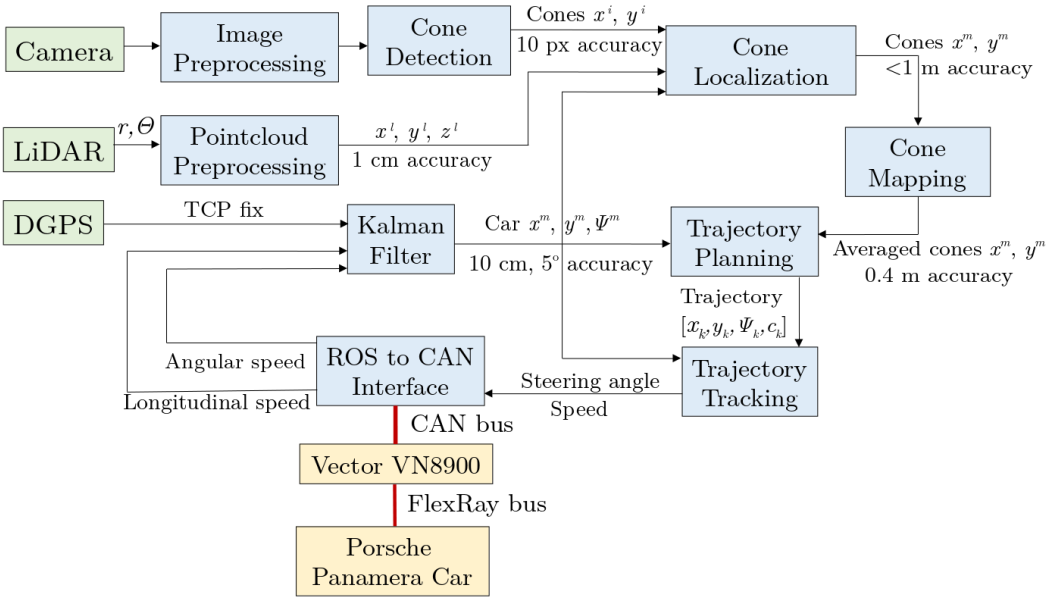



Figure 2. Functional block diagram – the software components (ROS nodes) are in blue, the sensors are in green and the interface to the car is in yellow. The thin lines represent the data flow within the software, the thick red lines are the communication buses. $x^m$ is the fixed coordinate system (map). $x^l$ is the LiDAR coordinate system, with the origin in the LiDAR center with $x$ axis in the direction of sight. $x^i$ is the camera image coordinate.

## 3. System blocks

In this section, we describe all the relevant software blocks from Fig. 2 and their algorithms in detail.

### 3.1. Cone detection

The purpose of the cone detection block is to detect the cones in the camera images. The whole algorithm is composed of several stages, each implemented using the OpenCV library.

First, the camera takes the image. Since we use a wide-angle lens, the raw image is distorted by the fisheye effect. We compensate for this effect by using the "undistortion" transformation computed by the OpenCV libraries.

The next stage is color filtration. For our experiment, we use traffic cones with orange and white stripes. The image is converted to the HSV color space, and areas with orange color are found (see Fig. 3a). The color filtration is quite sensitive to light conditions. Thus, we need to switch between two different filtration threshold setups depending on sunny or cloudy weather. After the color filtration, we calculate the centers of orange regions – see the blue dots in Fig. 3b.

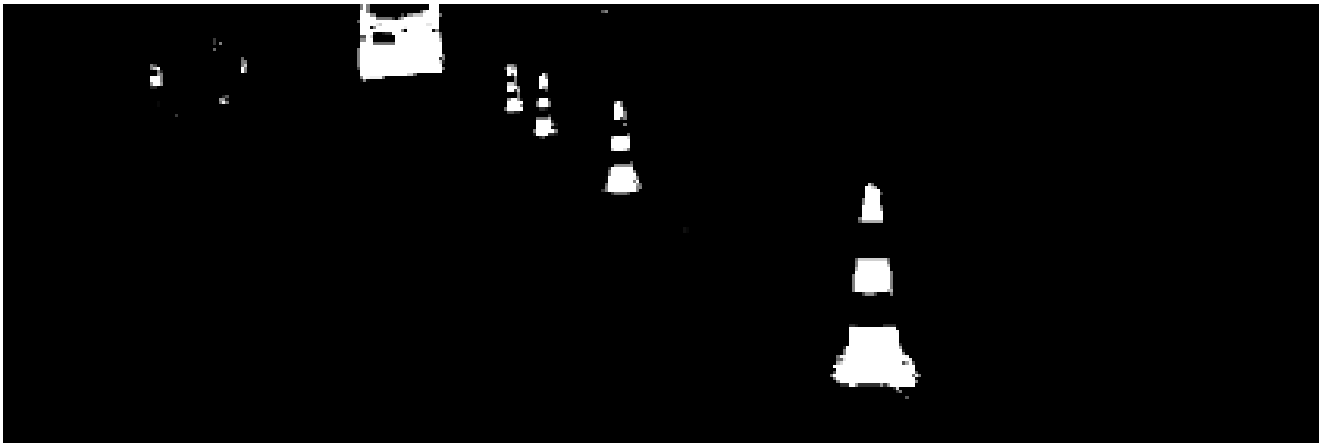
Next, we determine a region of interest (ROI) for the following processing phase. The cone in the filtered image consists of three orange regions with vertically aligned centers. All such triplets of regions are found, and ROI rectangles are constructed around these triplets.

Finally, we use a linear support vector machine (SVM) classifier to determine whether the ROI contains a cone or not. We adopt a method **object_recognition** for the classification of the cone in ROI. A histogram of oriented gradients (HOG) is computed for every ROI. The HOG defines a feature vector representation for the classification. The SVM classifier is trained on a set of 500 images of cones and 4500 images of non-cones. The classified cones are shown in Fig. 3c.

The output of cone detection block is a sequence of positions of the bottom centers of the detected cone ROIs, i.e. $[x^i, y^i]$.

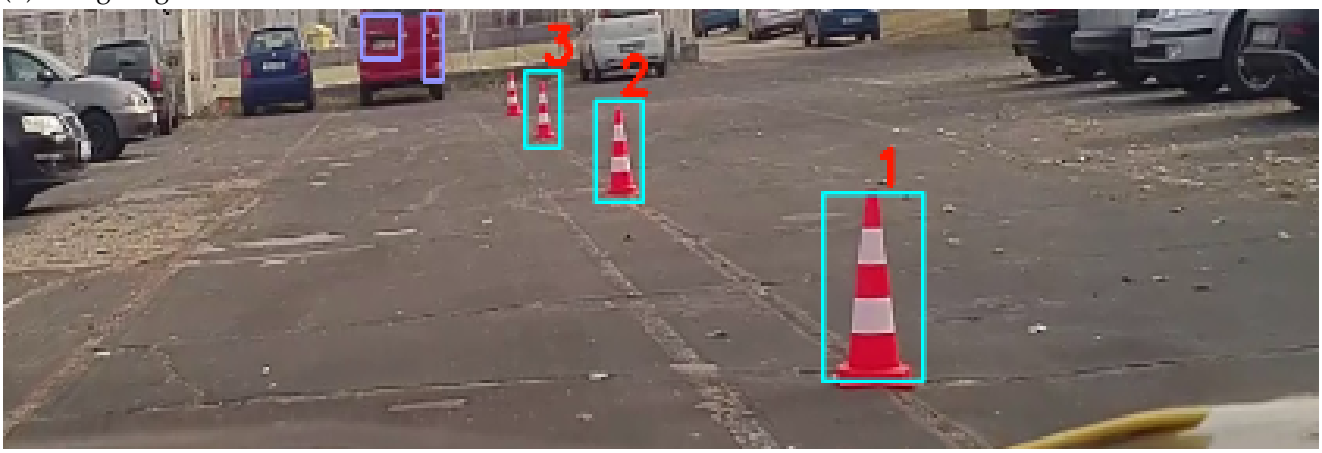
### 3.2. Cone localization

This block computes the position of cones in global coordinates. The inputs are the camera and LiDAR data. It is impossible to compute the cone position solely from the camera image provided the camera position and



(a) color filtering

(b) orange region centers




(c) final classification

Figure 3. Steps of cone detection algorithm.

parameters are known. However, the result would be extremely imprecise due to errors caused by the imprecise image undistortion and car swinging. Therefore, we project the point cloud from the LiDAR measurements to the camera image and look for points that are close to the cone's centers as provided by the cone detection component.

To localize the cones, the camera has to be calibrated. We use the OpenCV libraries to compute the camera intrinsic matrix $\boldsymbol{K}$ **camera_calib** This step is performed only once.

The transformation from the LiDAR coordinate frame ($[x, y, z]^l$) to the camera frame ($[x, y, z]^c$) is defined by rotation matrix $\boldsymbol{R}$ and translation vector $\boldsymbol{t}$. Their values result from manual measurements of the physical mounting of the camera and LiDAR. We use the extended coordinates where each point is defined as $[x, y, z, 1]^T$, which allows us to use simpler equations combining the rotation and translation into one transformation. Therefore, point $\boldsymbol{x}^l$ in the LiDAR frame maps to point $x^c$ in the camera frame according to this formula:

$$\boldsymbol{x}^c = \begin{bmatrix} \boldsymbol{K}[\boldsymbol{R}\boldsymbol{t}] \\ 0\,0\,0\,1 \end{bmatrix} \boldsymbol{x}^l. \tag{1}$$

We use the pinhole camera model, where the position in the image is computed from the camera coordinates $[x^i,\ y^i]$ as

$$\boldsymbol{x}^i = \begin{bmatrix} x^i \\ y^i \end{bmatrix} = \frac{1}{z^c} \begin{bmatrix} x^c \\ y^c \end{bmatrix}. \tag{2}$$

Figures 5 and 12 show the resulting projection of LiDAR data to the camera image.

We search for cones in the LiDAR pointcloud, projected to the camera image, in the regions close to the cones detected in the camera image. Two situations can occur (see Fig. 4a):

- the cone is hit by the LiDAR beams directly – we call this situation *direct localization*;
- the cone is not hit and we need to estimate its position indirectly – this is called *approximate localization*.

The cone localization algorithm works as follows: If it finds a cone shape pattern in the beam planes (see Fig. 4b), it determines the middle point of the pattern and declares it as cone position $\boldsymbol{x}^l_{\text{cone}}$. If the cone shape is not found, then we compute an approximate cone position from the closest LiDAR points below and above the cone base by the linear interpolation of these points.

The mean error of the cone positions obtained with the direct localization method is about 10 centimeters, while the error of approximate localization is about ten times higher. We use the approximate localization only for far cones. The closer cones are always localized directly.

Once we have the cone position in the LiDAR coordinates, the last step is to transform it into the global coordinate system by translating and rotating it according to the actual car position.

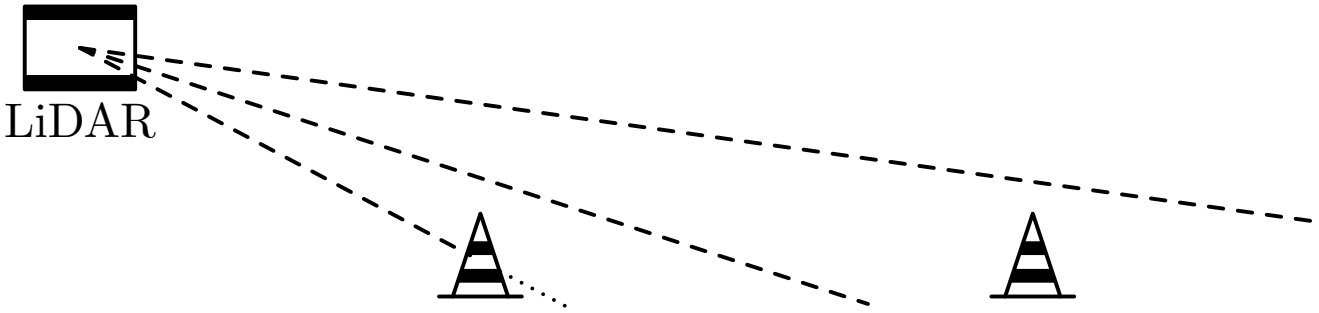
$$\boldsymbol{x}_{\text{cone}} = \boldsymbol{T}^m_l \boldsymbol{x}^l_{\text{cone}} \tag{3}$$

The cone localization method presented above works well only when the LiDAR can reliably measure the distance of the points on the road. On reflective surfaces, the laser beam does not reflect back to the LiDAR, and the distance cannot be measured. Such a situation occurs during the rain when the road becomes wet and reflective, as shown in Fig. 5.

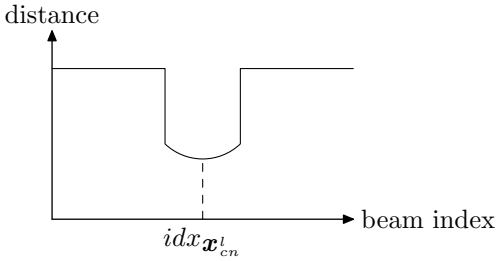
## 3.3. Cone mapping

The cone localization, described above, provides cone positions $\boldsymbol{x}_{\text{cone}}$, which we call *cone samples* below. The accuracy of cone samples is insufficient for trajectory planning. The two sources of low accuracy are the approximate localization and the inaccuracy of car localization. Specifically, the estimate of the car heading from the Kalman filter can be inaccurate in U-turns (due to the inaccuracy of our bicycle model in sharp turns) or at the beginning of the drive (as described in Section 3.4). We improve the precision by calculating the cone position as a weighted average of multiple cone samples grouped into clusters by a modified version of the k-means clustering algorithm. The precision obtained with this method is sufficient for trajectory planning.

The clustering algorithm assumes that the cones are at fixed positions (in the global coordinate system) at least during one slalom drive between the U-turns and that the minimum distance between the cones is $D_{\text{min}}$.




(a) LiDAR beams in cones localisation. The left cone shows the situation for the *direct localization*, the right cone for the *approximate localization*.




(b) Distance measured by the LiDAR in the direct localization case.

Figure 4. Cone localization

Figure 5. Example of missing LiDAR measurements on reflective surfaces. Many points on the road are missing.

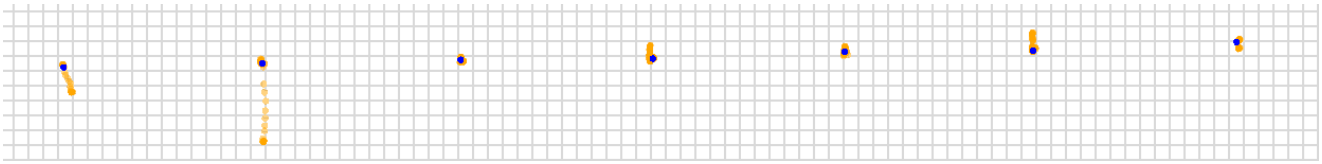


Figure 6. Cone samples as detected by the LiDAR (orange) and cluster centers (blue). The size of the grid is 1 meter.

In the first step, we cluster the cone samples into groups that correspond to the individual cones. Each cluster is represented by center position $\boldsymbol{c} = [x_{\text{center}}, y_{\text{center}}]^T$ and sample count $n$. If the new cone sample is closer than $0.8D_{\min}$ to an existing cluster center, we add the sample the that cluster. Otherwise a new cluster is created for the sample and $\boldsymbol{c} = \boldsymbol{x}_{\text{cone}}$. Only clusters with more than $N_{\min} = 4$ cone samples are used for the estimation of the cone positions. If the total number of clusters is higher than the maximum number of cones $c_{\max}$, we ignore the clusters with the lowest number of samples.

In the next step, we update the position of cluster centers $\boldsymbol{c}$ based on the new sample, using exponential forgetting. The cluster center is updated as

$$\boldsymbol{c}_{n+1} = (\rho - 1)\boldsymbol{c}_n + \rho \boldsymbol{x}_{\text{cone}}, \tag{4}$$

where the forgetting rate $\rho$ depends on how the sample was localized. Samples obtained with direct localization (see Section 3.2) have $\rho = 0.1$, the approximate localized samples have $\rho = 0.01$.

Figure 6 shows the cone map created from the cone samples. The car drives from left to right. At the beginning of the ride, the cones samples are affected by the uninitialized heading in the Kalman filter (see Section 3.4). Because of the exponential forgetting of the samples, this initial error does not affect the map once the Kalman filter is initialized.

## 3.4. Car localization with Kalman filter

This section presents how an application of the extended Kalman filter improves accuracy of car localisation by fusing data from DGPS and car-internal acceleration and velocity sensor. Kalman filtration is a well-known method widely used for state estimation **ekf** The algorithm can be split into two steps. The *prediction* of the state in the future and the *update* of the prediction by the new measurement.

The Kalman filter is based on the bicycle kinematic model **bicycle_model**

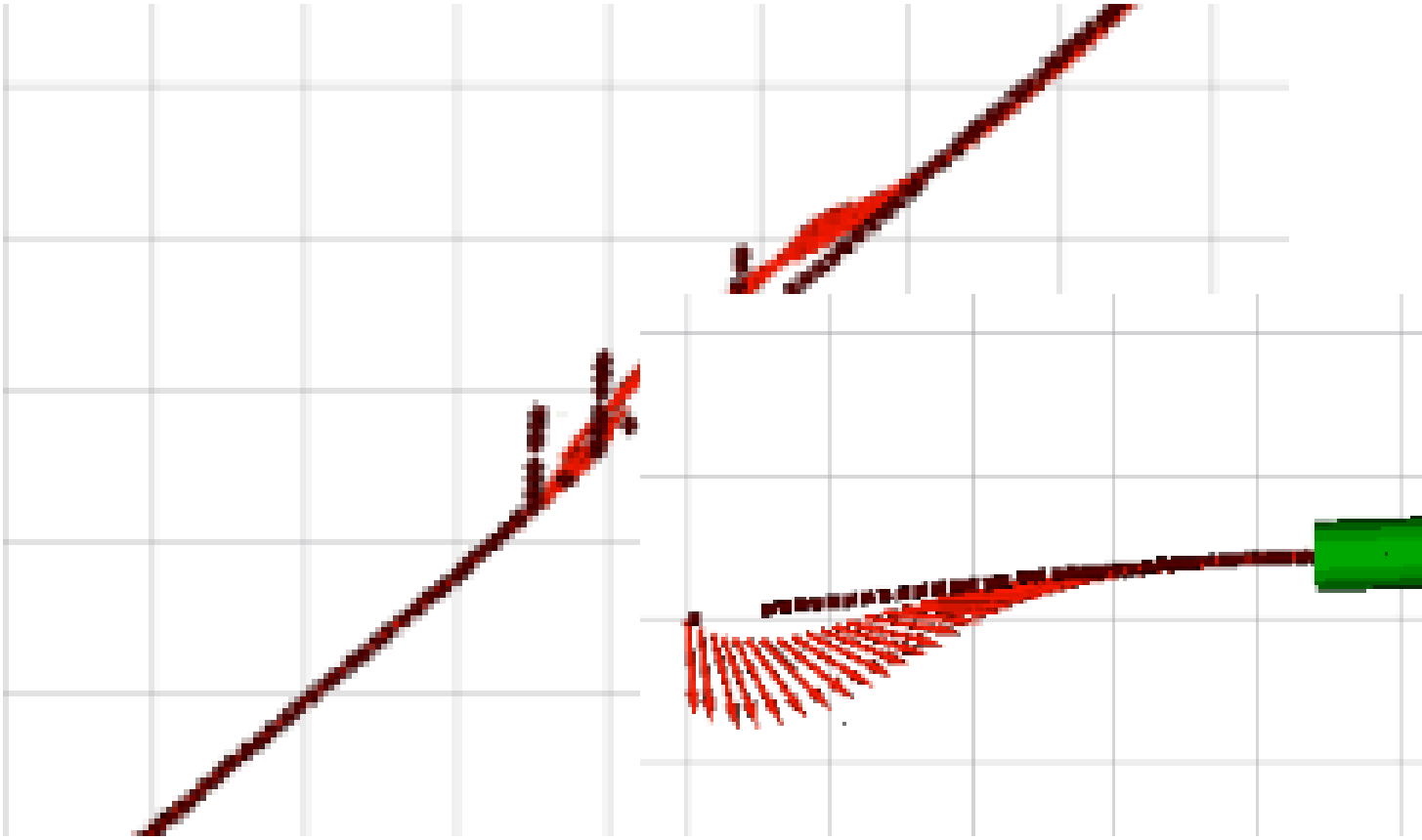
$$\begin{aligned} x_{k+1} &= v\cos(\psi_k + \beta_k)\cdot T + x_k \\ y_{k+1} &= v\sin(\psi_k + \beta_k)\cdot T + y_k \\ \psi_{k+1} &= \frac{v}{W}\delta \cdot T + \psi_k \end{aligned} \tag{5}$$

A typical state space representation of a nonlinear system is

$$\begin{aligned} \mathbf{x}_k &= \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_{k-1} \\ \mathbf{z}_k &= \mathbf{h}(\mathbf{x}_k) + \mathbf{v}_k. \end{aligned} \tag{6}$$

Here, $\mathbf{w}_k$ and $\mathbf{v}_k$ are the process and observation noises, which are both assumed to be zero-mean multivariate Gaussian noises with covariance matrices $\mathbf{Q}_k$ and $\mathbf{R}_k$ respectively. In our case, the state vector $\mathbf{x}_k = [x, y, \psi]^T$ and control vector $\mathbf{u}_k = [v, \delta]^T$. The observations $\mathbf{z}_k = [x_{\text{gps}}, y_{\text{gps}}]^T$, are obtained from the DGPS. The Kalman filter finds the estimation of $\boldsymbol{x}_k$ as $\hat{\boldsymbol{x}}_{k|k-1}$ or $\hat{\boldsymbol{x}}_{k|k}$.

To implement the discrete-time predict and update equations, we need the state transition and observation



(a) Filtering of DGPS disturbances (b) Heading initialization

Figure 7. Operation of the Kalman filter on a series of steps. The black dots are the DGPS positions, the red arrows are the output of the Kalman filter, the green arrow is the car's position in the last step.

matrices to be in the form of Jacobians. We compute the Jacobian $\mathbf{F}_k$ and $\mathbf{H}_k$ from the state space model (5).

$$\begin{aligned}\mathbf{F}_k &= \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\Big|_{\hat{\mathbf{x}}_{k-1|k-1},\mathbf{u}_k} \\ &= \begin{bmatrix} 1 & 0 & -v_{k-1}T\sin(\psi_{k-1}+\beta_{k-1}) \\ 0 & 1 & v_{k-1}T\cos(\psi_{k-1}+\beta_{k-1}) \\ 0 & 0 & 1 \end{bmatrix} \\ \mathbf{H}_k &= \frac{\partial \mathbf{h}}{\partial \mathbf{x}}\Big|_{\hat{\mathbf{x}}_{k|k-1}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.\end{aligned} \tag{7}$$

The model (5) is used for the prediction of the state as well as to calculate the Covariance matrix. The performance of the implemented filter tuned for our use case is visualized in Fig. 7.

## 3.5. Trajectory planning & tracking

When the cone map is created, we can start to plan the slalom trajectory. Many motion planning techniques were developed **planning_review** These span from planning in a complex urban environment **urban_planner** or planning with vehicle or surface limits **constrained_planning** Our environment for planning is relatively simple, so our primary constraint is the vehicle kinematics.

### 3.5.1. Global waypoints planning

First, we define a *waypoint* as a vector $[x, y, \psi, c]$ where $x,\ y$ are coordinates in the map, $\psi$ is the car heading, and $c$ is the steering curvature. The waypoint sequence is determined based on the cone map. We place one waypoint

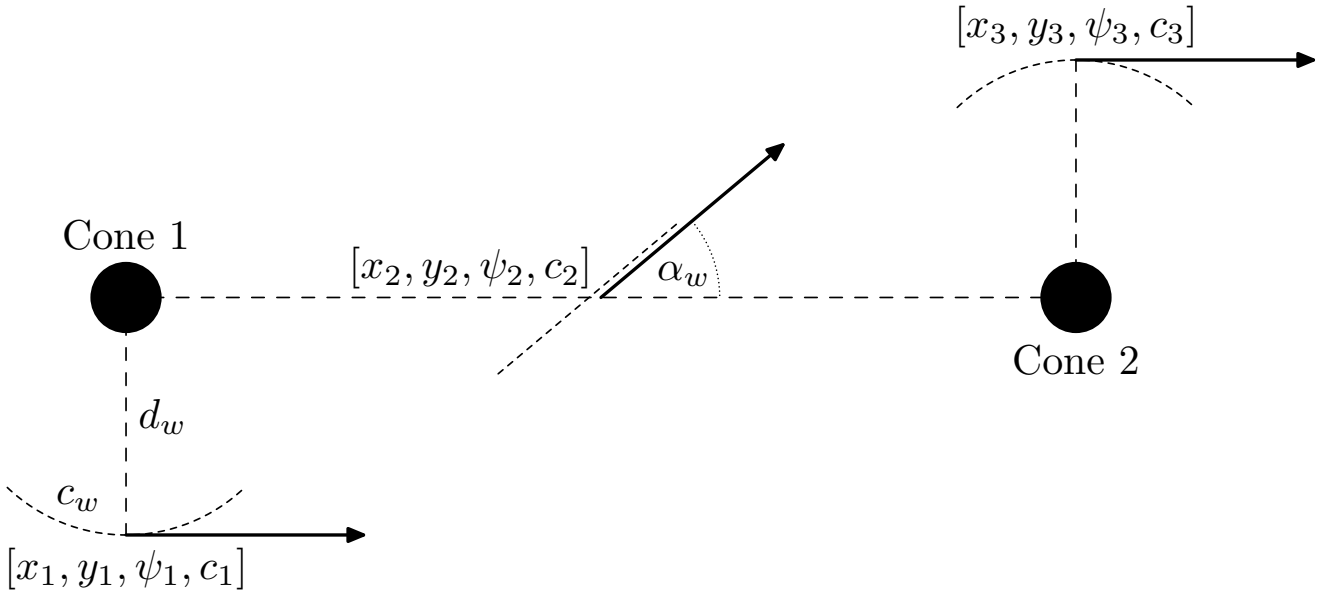



Figure 8. Planning the slalom waypoints

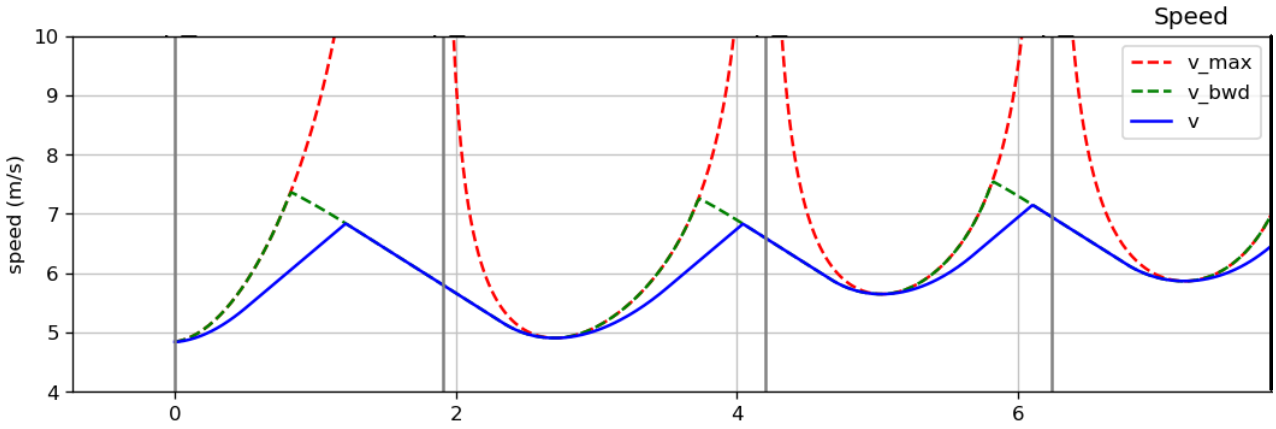


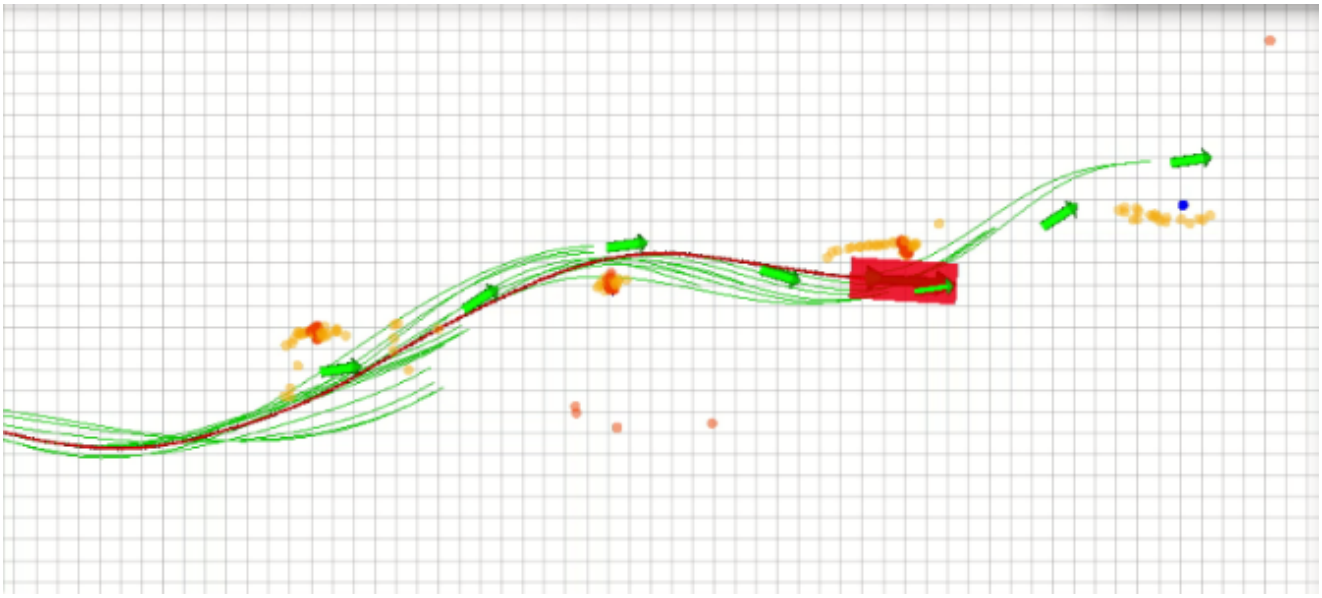

Figure 9. Speed profile

next to each cone and one in between two consecutive cones. This is illustrated in Fig. 8. The exact location depends on the following parameters:

- Curvature $c_w$ when driving next to the cone,
- distance $d_w$ from the cone, and
- angle $\alpha_w$ between the car heading and the line connecting the cones.

The local trajectory planner then takes the waypoints and creates a smooth trajectory for the slalom part. At the ends of the slalom, a circular trajectory around the last cone is appended to make the U-turns.

Once the smooth trajectory is planned, we compute the speed profile to fulfill constraints of maximum acceleration. We adapt the two-pass algorithm presented in **speed_profiling** The algorithm consists of the following steps. Find the maximum velocity at each of the points along the trajectory. Then the velocity profile is computed by applying maximal allowed acceleration and deceleration. We compute the velocity profile for acceleration by starting at the first point of the trajectory and applying acceleration until we reach the velocity limit. The same procedure is then executed in the backward direction, starting from the end of the trajectory, going back to the start, and applying the maximally allowed deceleration. An example of speed profile is shown in Fig. 9.



Legend:
- Direct cone localizations.
- Approximate cone localizations (interpolation between two LiDAR beams).
- Cone cluster centers used for the planning.
- Actual car position.
- Waypoints from the global planner which determine the slalom path.
- Trajectory from the local planner.
- Real car trajectory (GPS position).
- 1 meter grid.

Figure 10. Automated slalom map

#### 3.5.2. Trajectory tracking

The global and local planners are called periodically every 2 seconds to update the planned trajectory according to the new measurements. The car does not follow the trajectory perfectly because of the difference in car dynamics for different surfaces and steering angles. Therefore the new trajectory may start at a small distance from the old one. We linearly interpolate between the old and new trajectory for one second after the switch to avoid discontinuities in the controller steering output when switching between the plans.

The steering angle is computed from the trajectory curvature. A simple non-linear P-controller tries to minimize the deviation from the planned trajectory by computing the steering angle from the current car position, heading, and the closest point of the planned trajectory. We do not implement any sophisticated trajectory tracking algorithm, an example of a better method can be found in **trajectory_tracking**

We have to compute the action values ahead of time (by approximately 0.1 s) to compensate for delay caused by the communication chain and the car ECU response time, i.e., time between sending the request for steering angle position and reception of this value.

## 4. Experiments

We have performed many test drives with the whole setup described in the previous sections. This section presents the data from some of these real-world experiments.

### 4.1. Slalom drives

The overview visualization of the data from one slalom drive is shown in Fig. 10 (corresponding drone photo is in Fig. 13). As noted above, the approximate cone positions (orange dots) are used only at the beginning of the

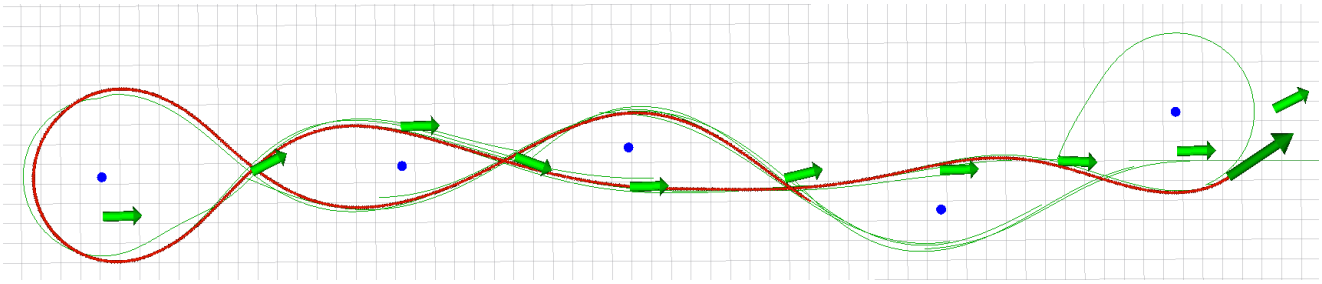


Figure 11. Slalom between scattered cones.

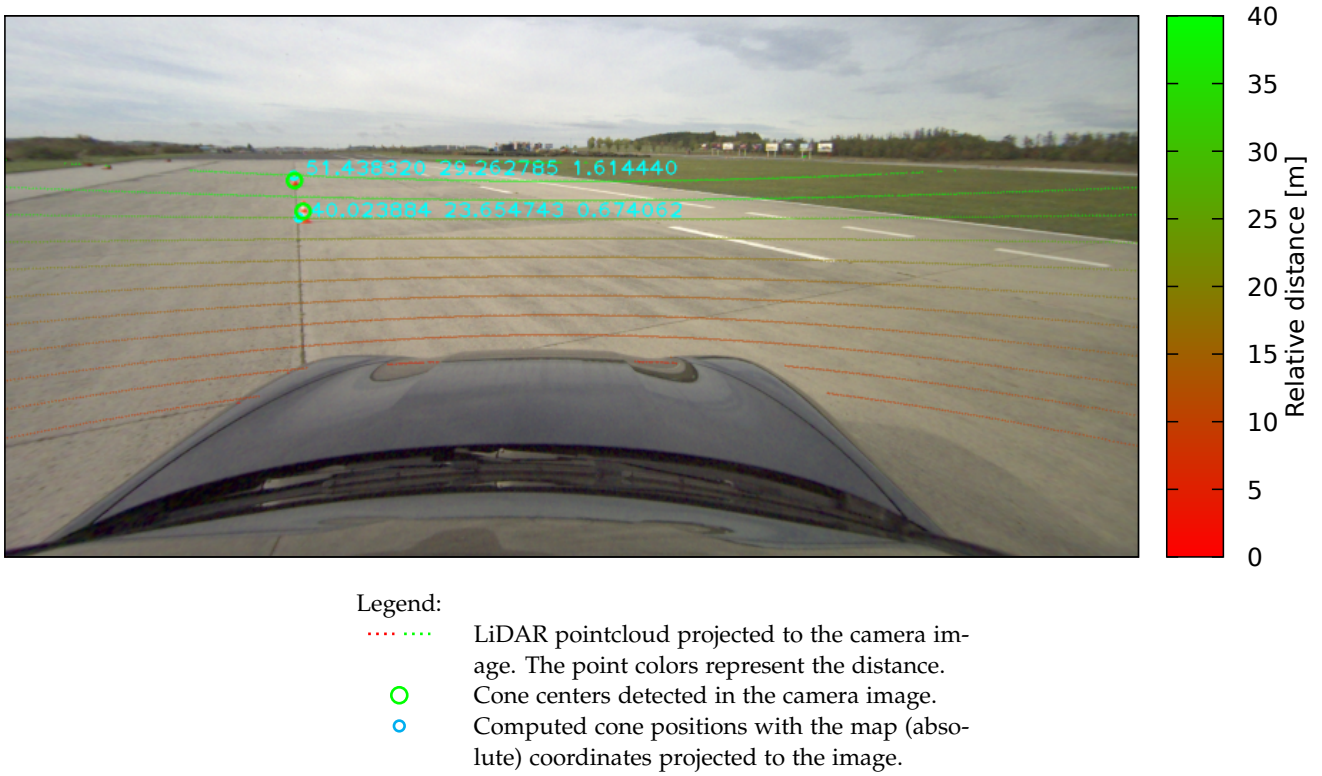



Figure 12. Camera view from an experimental drive

drive. After the car is close enough for the direct localization, more precise cone positions (red dots) are used. The approximate positions serve only to create the initial trajectory plan.

The cones do not necessarily need to be placed uniformly on a straight line. The slalom can be also driven between cones laid out as is shown in Fig. 11.

Figure 12 shows what the camera and LiDAR see when the car is in front of the cones.

## 5. Conclusion

This paper describes the architecture and algorithms used for the automated car prototype demonstrated in the automated slalom use case. This prototype is a reference solution based on a well-established modular architecture, enabling future improvements of individual components. A simple initial design, implementation, and early outdoor experiments have shown to be the right way to reveal bottlenecks and weaknesses of development versions of the system and improve individual components in an agile manner. For example, it turned out that the appropriate precision of one component (e.g., heading estimation) was essential for the correct behavior of another component (e.g., trajectory planning). Given the limited resources of the project, we have learned that outdoor experiments are more efficient than the complete system and environment simulation, which cannot model every detail of the real world. On the other hand, simulations of only a small part of the system and environment proved helpful for reproducing and fixing specific problems encountered in outdoor

Figure 13. Bird view of the experiment from Fig. 10

experiments. Due to their limited extent, such simulations were much easier to create. The significant success is that we were able to fit all the computations in the embedded NVIDIA TX2 computer and run all the algorithms at a sufficient rate while keeping the reliable operation of the system.

## Acknowledgments

Research leading to these results has received funding from the EU ECSEL Joint Undertaking and the Min- istry of Education of the Czech Republic under grant agreement 826452 and 8A19011 (project Arrowhead Tools).