Vision System for the Autonomous Ecocar

31910 Synthesis in electrotechnology

Tomasz Jerzy Firynowicz (s171810)

21.09.2018

Contents

1 Introduction

The main goal of the project was to propose, implement and test computer vision system for autonomous vehicle control for Shell Eco-marathon 2018 competition. Primary requirement for the developed system was to achieve as good as possible performance and robustness in the challenges. Secondary requirement was to keep the system general and flexible enough so that it can be built upon in upcoming years for different challenges.

1.1 Description of the challenges

In initial version of Shell Eco-marathon rules included two parking challenges that could be solved using computer vision.

First one, that was also kept in the final competition, was about detecting a single parking spot on the straight part of the track and parking there autonomously. Additional points were awarded for parking as close as possible to the wall at the end of the spot. Track layout is shown in fig. 1.

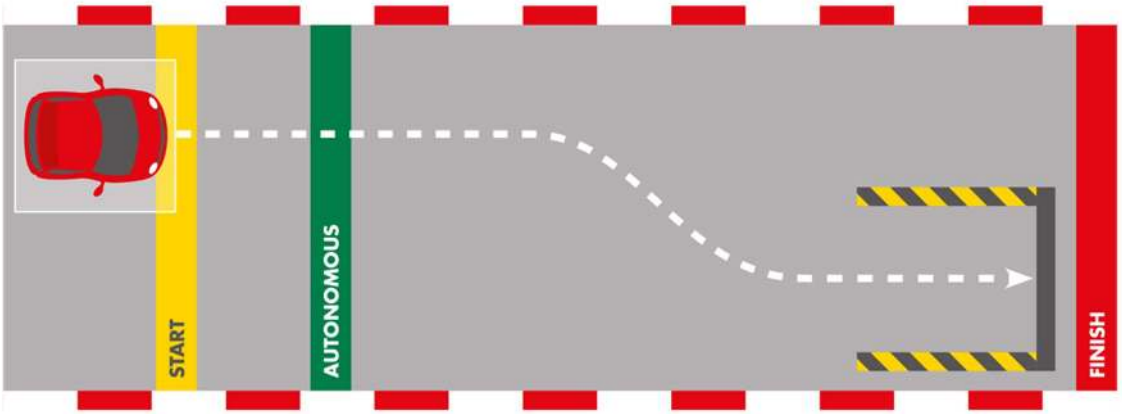

Figure 1: Track layout in the first parking challenge. [1]

In the second challenge, four parking spots were marked on the track. The task was to park autonomously on one of them. Which one it should be was to be announced just before the start of the challenge. This challenge was set up in qualification round in Paris. Only our team managed to successfully complete it. For this reason it did not appear again in the final competition in London. Track layout for this challegne is shown in fig. 2.

For complete description of the challenges please refer to official rules of the competition in [1].

1.2 My contribution

During the development of the computer vision system I worked together with Mikołaj Patalan. My contribution was camera intrinsic calibration and inverse perspective transform and its calibration. I have also written functions for transforming coordinates of points on the image from the camera to global reference frame. Another thing that I

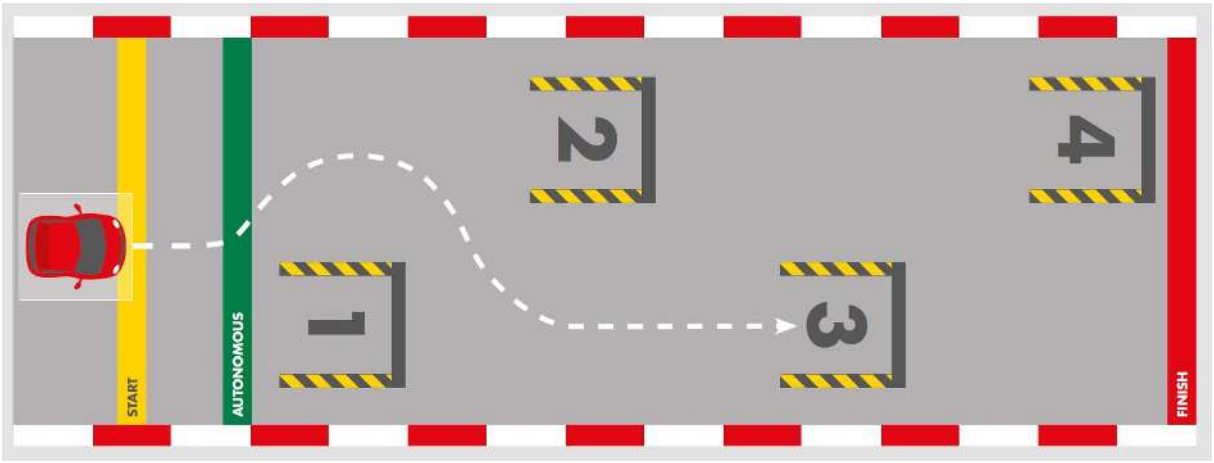

Figure 2: Track layout in the second parking challenge. [1]

worked on was a sound indicator that warns the user if camera is not connected properly or not detected by the operating system. I was also responsible for integration of our work with the rest of the car's system in ROS.

Mikołaj Patalan focused mostly on detection of the parking spot rectangles on the images after the inverse perspective transformation was applied. He also developed statistical analysis algorithm that rejects false positive detections and ensures that coordinates published by the package are correct, based on multiple detections of the same spot. For more information about those parts, please refer to his report [2].

1.3 Sections overview

The report covers the motivation of the chosen approach, detailed description of the solution and analysis of possible improvements for the upcoming editions of the competition. The main focus is put on the principle of operation of different modules of the project and their interaction. The details of implementation and operation of the developed package were intentionally omitted in the main body of the report. Instead, all the practical details necessary to use (but not necessarily understand) the developed package were collected in the appended manual.

2 Discussion of viable approaches

Both autonomous parking challenges initially proposed for Shell Eco-marathon 2018 can be solved using computer vision. Markings on the road that are clearly visible and distinguishable for human eye can be also detected by processing digital images captured by a camera. First parking challenge, with single parking spot, could also be solved using lidar data only. It is possible because the block at the far end of the parking spot had well defined dimensions, so it could be detected in the point cloud of lidar measurements. Following sections describe in more detail the advantages and disadvantages of both approaches.

2.1 Computer vision

Using vision for detection and determining the position of a parking spot is exactly what human driver does when faced with a parking task. It proves that vision based approach is in principle viable. It comes with the advantage of ability to detect markings on the road that do not have any distinctive three dimensional geometric features, like markings on the road surface or road signs. Solving general autonomous driving in the traffic with the current road infrastructure without computer vision is impossible.

2.2 Lidar

Lidar based approach could yield more accurate measurements than those obtained through image analysis. First parking challenge could be solved using lidar only. It would come with a cost of lower flexibility of the solution. Second parking challenge with multiple spots would be impossible to solve without utilization of visual information.

2.3 Chosen approach

In our solution we decided to use computer vision for detection and estimation of the position of the parking spot. It was supplemented by lidar measurements in the final seconds of the run, for precise braking as close as possible to the block at the end of the parking spot. Using computer vision for the most part allowed for more general solution that could be successfully applied to both challenges without large modifications to the core parts of parking spot detection code. During preparation we did not know that the second challenge would be canceled in the final competition and we were ready to complete it.

3 Solution

In the following sections the developed solution was described in detail. It contains description of used hardware and software, as well as description of the software modules developed specifically for the challenges.

3.1 Hardware

The most important pieces of hardware that affect the developed computer vision pipeline are camera and the on-board computer.

3.1.1 Camera

During the competition we used camera model ELP-USB8MP02G-L75 shown in fig. 3. It was connected to the computer using regular USB interface and worked out of the box

in Ubuntu just like any other USB webcam. It's maximum resolution is 3264x2448 at 15 FPS. Setting lower resolution allows for frame rate up to 30 FPS.

Figure 3: ELP-USB8MP02G-L75 USB camera used during the competition.

During the challenge we run the camera with resolution of 2048x1536 at 15 FPS. We found that with our camera, increasing the resolution beyond this level did not increase the image quality significantly. Increasing the resolution also increases the time needed to process the images, which is undesirable. Frame rate 15 FPS is the maximum available for the chosen resolution. It was sufficient for the application.

The camera was mounted on the top of the shell. It was facing forward and adjusted so that the car's shell is not present in the field of view. Any car parts visible in the field of view would have to be cut out in the preprocessing step to not interfere with the parking spot detection.

3.1.2 On-board computer

As the on-board computer we used Intel NUC Board NUC7i7DNBE with Intel Core i7-8650U quad core processor. This CPU performed all of the computing tasks during the competition, including image processing. The achieved image processing frequency was in the range between 3 Hz and 5 Hz and was sufficient to solve the challenge. More powerful CPU could improve the performance of the program. Using a weaker CPU without lowering the input image resolution or modifying the program could make the detection frequency too low to work.

3.2 Software

The image processing pipeline used in the system consists of several ROS packages. Some of them were already available and had to be put together to achieve desired effect. Some crucial parts of the pipeline were developed by us. In those parts, OpenCV library was leveraged as much as possible to get desired implementation faster and assure good software quality.

The overview of the pipeline is shown on fig. 4. All of the software was developed and tested on Ubuntu 16.04 with ROS Kinetic Kame installed. For installation, launch and use instructions please refer to the appended manual.

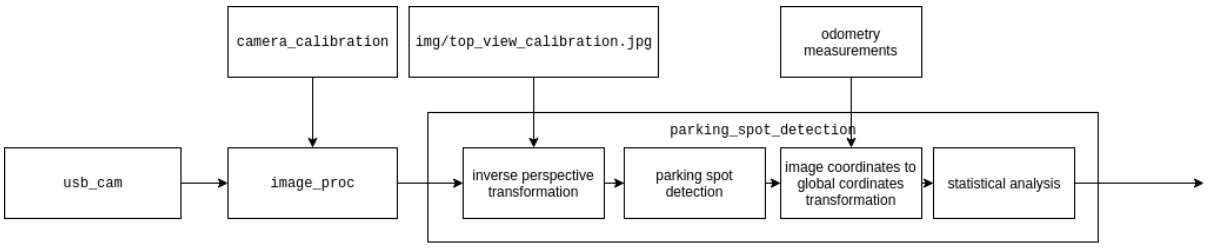

Figure 4: Image processing pipeline.

3.2.1 `usb_cam` package

This package is responsible for acquisition of video stream from USB camera and publishing each frame in a ROS topic. The package can publish images in multiple different compression levels. The most important topic for our application is `/usb_cam/image_raw`, which contains uncompressed images.

3.2.2 `image_proc` package

This package is responsible for applying the intrinsic calibration. The images published by `usb_cam` are ingested by `image_proc`, the geometry correcting transform is applied to each frame after which it is republished in many different topics. New topics published by `image_proc` offer different compression levels or color and gray-scale images. The topics that are most important in our application are `/usb_cam/image_rect_color` for real-time image analysis and `/usb_cam/image_rect_color/compressed` for logging purposes.

Intrinsic calibration is necessary to eliminate various kinds of geometric distortion caused by optics of the camera. It's primary goal is to ensure that real-world's straight lines are also straight in the images acquired by the camera. It is important to enable geometric measurements of the surroundings using the camera images. Example of an image before and after calibration is shown in fig. 5.

The calibration needs to be performed separately for each camera and for each resolution setting. For more specific instructions on calibration process, please refer to the appended manual.

3.2.3 `parking_spot_detection` package

This package contains the rest of the code necessary to perform parking spot detection. It was custom written for this application and makes extensive use of OpenCV library. It accepts calibrated images published by `image_proc` as input and outputs coordinates of the detected and validated parking spots.

The image processing routine within the package consists of several steps outlined in the following sections.

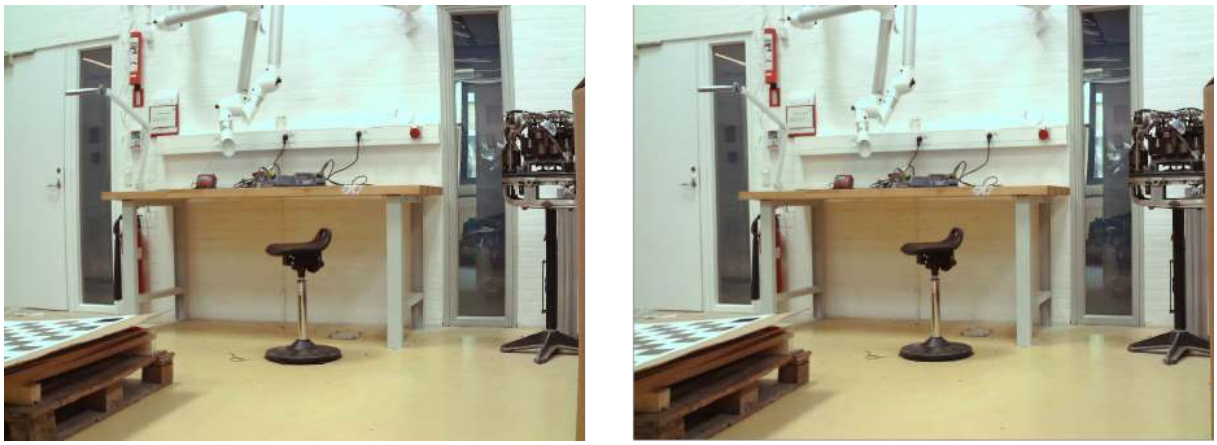

Figure 5: Example of uncalibrated and calibrated camera image. The distortion of straight lines in the peripheral parts of the field of view is visible in the left image.

3.2.3.1 Inverse perspective transformation

The goal of inverse perspective transformation is to take image from the front facing camera mounted on the car and transform it so that part of track in the field of view appears as if it was seen from top. Example of image before and after such transformation is shown in fig. 6. After transformation the rectangle of a parking spot is really a rectangle with its sides parallel and square angles preserved. This representation makes it particularly easy to detect parking spots.

Figure 6: Image before and after inverse perspective transformation.

The key information needed to perform the transformation is the homography matrix. Its values are dependent on the relative position of camera mounted on the car and the surface of the track. The values of homography matrix can be estimated using `findHomography()` function from OpenCV. It takes two vectors of points as output. First vector contains points from the image before the transformation. The second vector contains corresponding points, which positions define where the points from the first vector should be mapped to by obtained homography matrix. The theoretical minimum of points

needed to calculate a homography is 4, but providing more points will result in better homography estimation in case of inaccuracies in points placement. `findHomography()` function employs optimization techniques to get the best homography matrix estimation.

The `findHomography()` function was used to calibrate the inverse perspective transformation. Step by step instructions on how to perform it are available in the appended manual. The basis for the calibration is a calibration image taken from the car's camera. Camera first must be calibrated to remove intrinsic distortion of the image. In the calibration image calibration checkerboard of specified dimensions must be visible and placed in a well defined position relative to the car. Example of calibration image before and after applying the transformation is shown in fig. 7. The corners of the checkerboard in the calibration image can be easily detected using OpenCV functions. Detected points are marked in fig. 7 in the top right image. Then a new set of points can be generated. Those points should be evenly spaced on a rectangular grid, as shown in lower left image of fig. 7. This will be second set of points passed to the `findHomography()` function. In this way, it will return a homography that will transform the checkerboard image affected by perspective into a "flat", rectangular checkerboard image, exactly as if it was viewed from the top, along with the horizontal planes parallel to the checkerboard surface, most importantly the surface of the track. Homography obtained using this procedure can be used to transform new frames as long as the track is approximately planar and the position of the camera on the car is not changed.

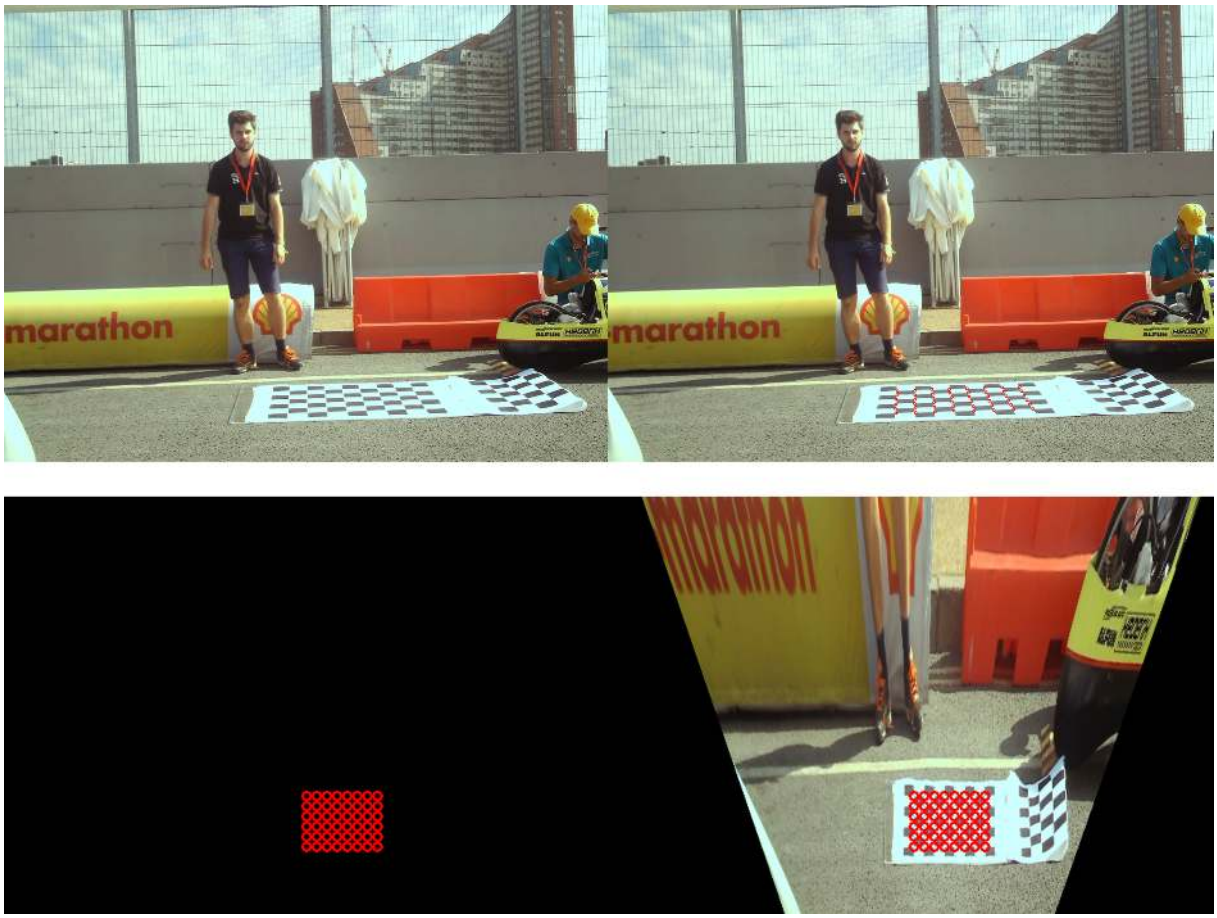

Figure 7: Inverse perspective transformation calibration procedure.

3.2.3.2 Parking spot detection and statistical analysis

Those crucial parts of the pipeline were developed by Mikołaj Patalan. For more information about them please refer to his report. [2]

3.2.3.3 Image coordinates to global coordinates transformation

When a parking spot is detected in the image, its position is known in the top-view image coordinates. It needs to be transformed to global coordinate system to compare it with previous detections and use it for navigation. There are two steps involved in the transformation.

First the coordinates from top-view image need to be transformed into car's local coordinate system. Assuming perfect calibration, top-view image plane is coplanar with XY plane of the car's local coordinate system. The checkerboard position in the car's local coordinate system was known during calibration. The pixel/m ratio was one of the parameters defined and used during the inverse perspective transform calibration. Using this information it is possible to calculate position of every pixel in the image relative to the car. Car's local coordinate system is also rotated in relation to the image coordinates. With all this information it is possible to derive the geometric transformation between top-view image coordinates and car's local coordinates.

Second step is to convert those into global coordinates using the current car pose read from `/car_pose_estimate` topic. The topic is updated based on the odometry of the car.

4 Possible improvements

Extensive testing and analysis of the performance on the competition provided enough insight to see which parts of the system need improvement the most in next iterations of the project. The most promising directions are listed below.

4.1 Camera mounting and calibration

A lot of time could be saved by mounting the camera to a fixed part of the car. This time it was rigidly mounted on top of the car's shell. Unfortunately the shell itself is not rigidly mounted to the frame of the car and is removed every time someone gets in or out of the driver seat. Every remounting of the top shell causes the position of the camera relative to the car to change. This makes inverse perspective calibration invalid. The algorithm may still work but the quality of measurements will be worse. It means that for perfect calibration quality, it needs to be performed after someone gets in or out of the car and every time when the camera mount is removed from the shell and remounted again.

To alleviate this problem it is recommended to mount the camera to a non-removable part of the car, so that its position relative to the track surface does not change.

4.2 GPU

Image processing takes a lot of computational power. The i7 CPU was powerful enough for all our computational needs after some optimization. Image processing performance could be improved by delegating some tasks to a separate GPU. Looking into this is strongly recommended for the next iteration of the project, especially if complexity of computer vision tasks would increase.

5 References

[1] O. H. (Shell), "2018 autonomous urbanconcept competition rules," 2017.

[2] M. Patalan, "Vision system for the autonomous shell eco-marathon car," 2018.

A Manual

This document is intended as a practical aid for setting up, operating and incorporating the `parking_spot_detection` package in a ROS project. It does not aim to explain how the package works internally or what was the motivation for particular approaches. For information of this kind please refer to the rest of the report. Following sections cover input and output of the package, software requirements and usage instructions.

A.1 Input

The package was developed to solve autonomous parking challenge of Shell Eco-marathon 2018 and is directly applicable only to this task. Examples of images that the package would work on are shown in fig. 8. For the definition of the challenge please refer to the Shell Eco-marathon rules [1].

Figure 8: Examples of valid input.

The package is designed to work with a front-facing USB camera with UVC interface (standard USB webcam interface).

A.2 Output

Detected parking spots are published to ROS topic `/parking_spot_detection/corners` as messages of type `geometry_msgs/PolygonStamped` structured as follows

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Polygon polygon
  geometry_msgs/Point32[] points
    float32 x
    float32 y
    float32 z
```

`polygon.points` is an array that contains two points defining the location of the parking spot in global coordinate system. `polygon.points[0]` is the center of the spot, while `polygon.points[1]` contains the center of the front line.

First message is published to `/parking_spot_detection/corners` topic only after it was validated by detection on multiple frames. After a spot is validated it is published every time it is detected on a new frame. The published coordinates are based on all previous detections of the same spot. For those reasons detections of false positives should appear very rarely and output of the package may be considered reliable.

Tests shown that the measurements of the y coordinate (how far to the left or right) of the spot were very accurate. Measurement of the x coordinate (how far down the track) was prone to errors and not reliable enough to use it for precise breaking close to the wall at the end of the spot. Lidar data was used instead to park as close to the wall as possible.

A.3 Installation

The developed ROS package was tested in ROS Kinetic Kame, running on Ubuntu 16.04. In following sections it is assumed that both are already installed.

A.3.1 Dependencies

First install `usb_cam` package missing from clean ROS installation by running

```
sudo apt install ros-kinetic-usb-cam
```

Also make sure to add `dynamo_msgs` package and build it before building `parking_spot_detection`. It contains topic definitions used in the package.

A.3.2 'parking_spot_detection' package installation

Copy the `parking_spot_detection` folder into `src` folder in the catkin workspace. Build the package by running `catkin_make` command.

A.4 Intrinsic calibration

Intrinsic calibration removes several types of distortion from the images captured by the camera. It ensures that real-world's straight lines are actually straight in the camera image. Fig. 9 shows an example of uncalibrated (left) and calibrated (right) image.

A.4.1 Intrinsic calibration file

If you already have intrinsic calibration file, save it as `~/.ros/camera_info/head_camera.yaml`. Remember that it is camera- and resolution-specific. If it is not available for the desired device and resolution combination, it needs to be generated by following the intrinsic calibration procedure.

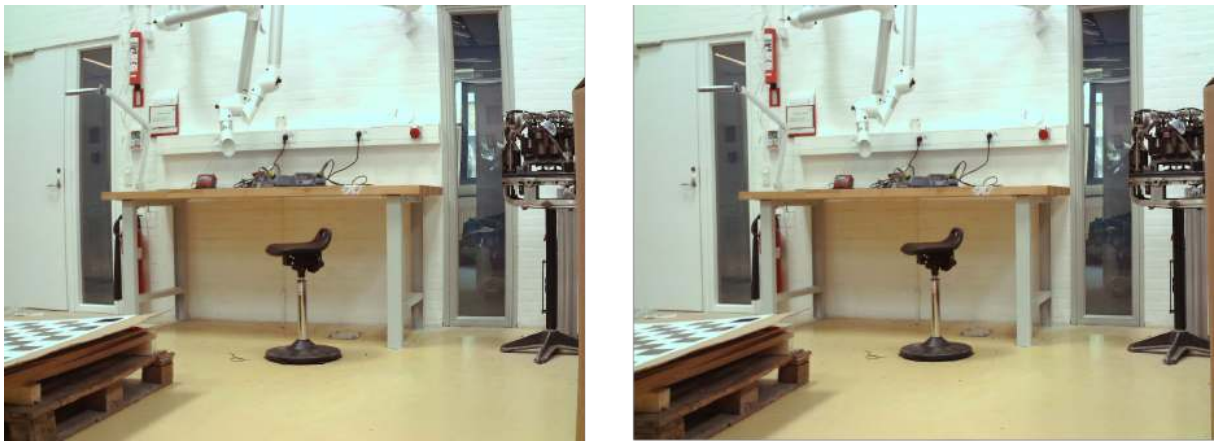

Figure 9: Example of uncalibrated and calibrated camera image. The distortion of straight lines in the peripheral parts of the field of view is visible in the left image.

A.4.2 Intrinsic calibration procedure

To perform intrinsic calibration you need to print calibration checkerboard. You can use 8x6 checkerboard with square size 108 mm provided with this manual. It should be printed on A0 format without any scaling. It is a good idea to print it on non-glossy paper to avoid reflections that will interfere with automatic checkerboard detection. Stick the printed checkerboard to any kind of light, rigid panel, that will be easy to move around and will keep the checkerboard flat.

Camera intrinsic calibration can be performed using `camera_calibration` ROS package. To perform the calibration, first make sure that `roscore` is running. Next, open a new terminal window and run

```
rosrun usb_cam usb_cam_node _image_width:=2048 _image_height:=1536
```

Replace 2048 and 1536 with resolution that you want to calibrate for. If there are multiple cameras connected to the computer you might need to specify additional argument `_video_device:="/dev/video0"` with path to the device that should be used. Then, in a new terminal window, run

```
rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.108 \
image:=/usb_cam/image_raw camera:=/usb_cam
```

where arguments are correct for calibration checkerboard provided with the manual. Next, follow the calibration wizard. Successful calibration will result in `~/.ros/camera_info/head_camera.yaml` file being generated.

A.4.3 Verification

To verify the quality of the intrinsic calibration you need to have `roscore` and `usb_cam usb_cam_node` running. Then run `image_proc` package that applies intrinsic calibration to raw camera images

```
ROS_NAMESPACE=usb_cam rosrun image_proc image_proc
```

To preview calibrated image run

```
rosrun image_view image_view image:=//usb_cam/image_rect_color
```

Inspect the calibrated image, especially the corners of the field of view, and take note of any distortion or curvature of lines that should be straight. If there is no visible distortion, the calibration was successful and can be used in parking spot detection pipeline.

A.5 Inverse perspective transform calibration

Inverse perspective transform calibration provides the system with information about the position of the camera relative to the track surface. This information is used to get "top view" of the track from the images taken with the front facing camera on the car. With perfect calibration, all rectangles on the road surface should appear as actual rectangles on the transformed image. Example of an image before and after transformation is shown in fig. 10.

Figure 10: Image before and after inverse perspective transformation.

A.5.1 Calibration file

The file that stores a calibration is an JPEG image file taken from the on-board camera. An example of such image is shown in fig. 11. The image should contain the calibration checkerboard in specific position relative to the car. Calibration file should be stored in the catkin workspace as `catkin_ws/img/top_view_calibration.jpg`. For the calibration to be correct, the image has to be taken with the same camera in the same position as in intended use. The resolution does not necessarily have to be the same, because the calibration image will be scaled to match it. It is still recommended to use the same resolution for both calibration image and the camera operation because it ensures the best calibration quality.

Figure 11: Example of inverse perspective transform calibration file.

A.5.2 Calibration procedure

1. To perform the inverse perspective transform you will need the same calibration checkerboard that was used for intrinsic calibration.
2. Park the car on a flat surface. The flat surface should extend about 4 m in front of the car and it should be free of any clutter. Remember to engage the handbrake to prevent car from moving.
3. Put the calibration checkerboard in front of the car as shown in fig. 12. The calibration will be correct only if 8x6 checkerboard with square size 0.108 m is used.

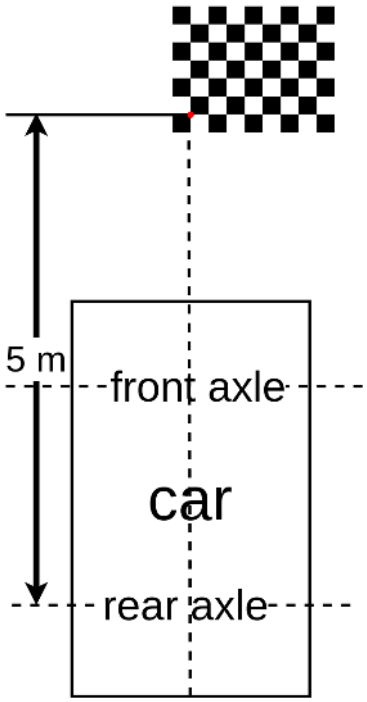

Figure 12: Position of checkerboard during calibration.

4. Make sure that calibration checkerboard is visible in the camera's field of view. You can use Cheese or

```
rosrun image_view image_view image:=//usb_cam/image_rect_color
```

rememeber that `roscore`, `usb_cam` and `image_proc` must be running.

5. Take a picture using command

```
rosrun image_view image_saver image:=/usb_cam/image_rect_color \
_filename_format:=$HOME/catkin_ws/img/top_view_calibration.jpg _sec_per_frame:=1
```

Change specified folder if different catkin workspace name is used. Wait until at least one frame will be saved and exit `image_saver` using Ctrl+C.

6. Check if the calibration image was saved in the specified folder.

A.5.3 Verification

1. Go to the `parking_spot_detection` folder and open file `src/parking_spot_detection.cpp`.
2. Change the line

```
bool debug = 0;
```

to

```
bool debug = 1;
```

3. Recompile the workspace usign `catkin_make`.
4. Run roscore, all necessary camera nodes and `rosrun parking_spot_detection parking_spot_detection`.
5. Two windows with live images should be displayed. The calibrated image from the camera and the 'top view' image after inverse perspective transform.
6. Sometimes the `parking_spot_detection` may only show a message `Top view calibration error!`, it most likely means that the checkerboard was not detected in the image and it should be taken again.
7. Move the car so that any kind of simple geometric feature on the road is visible in the camera's field of view. If the top view is not too much distorted, the calibration was successful. If the quality of the calibration is not satisfactory, repeat the procedure.

A.6 Launch

In order for `parking_spot_detection` package to work, all of the following commands must be executed in the following order in separate terminal windows

```
roscore
rosrun usb_cam usb_cam_node _image_width:=2048 _image_height:=1536 _framerate:=15
ROS_NAMESPACE=usb_cam rosrun image_proc image_proc
rosrun parking_spot_detection parking_spot_detection
```

It might be necessary to change values of some parameters to fit a specific application.

A.7 Image data and rosbag

Logging image data takes a lot of disk space. When running `usb_cam` and `image_proc` be sure not to log all published topics. Both of those packages publish multiple versions of the image topics, both compressed and uncompressed. It is advisable to log only the compressed images. There are two types of compression available `compressed`, which is much smaller in size than raw images but retains sufficient quality for image analysis tasks, and `theora` that brings size down even further, but with much greater negative impact on the image quality, it should be considered in cases when disk space is a constrain and image is logged only for preview purposes.

A.8 Frequently used commands

Run camera node

```
rosrun usb_cam usb_cam_node _image_width:=2048 _image_height:=1536 _framerate:=15
ROS_NAMESPACE=usb_cam rosrun image_proc image_proc
```

Calibrated image preview

```
rosrun image_view image_view image:=//usb_cam/image_rect_color compressed
```

Save single top-view calibration photo (be careful, it overwrites calibration photo that is already in the folder)

```
rosrun image_view image_saver image:=/usb_cam/image_rect_color _filename_format:=$HOM
```

Run `parking_spot_detection` node

```
rosrun parking_spot_detection parking_spot_detection
```

Save image data to rosbag

```
rosbag record /usb_cam/image_rect_color/compressed
```