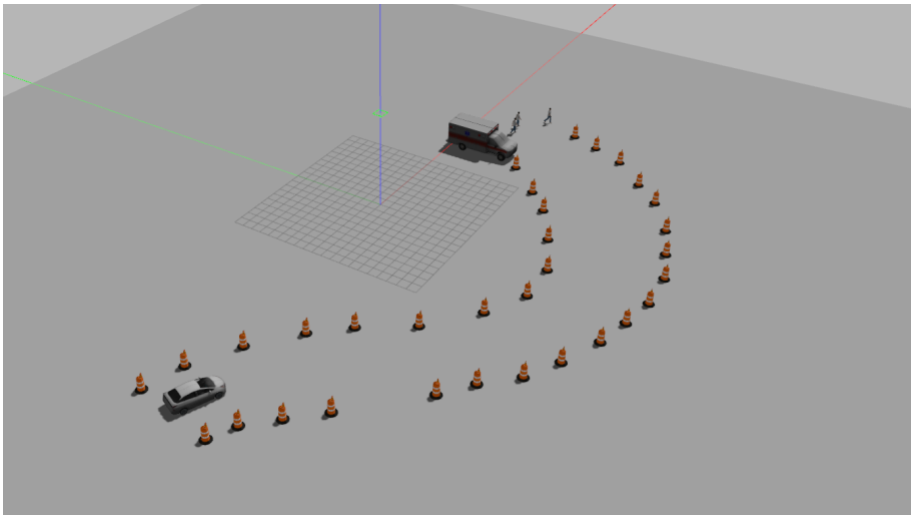
Robot Software Practicals - RO47003
2021/2022
manual version 38

# Final Assignment: Autonomous Driving



Ronald Ensing, Julian Kooij

# 1 Foreword

This is the final assignment of the Robot Software Practicals course. You will develop software for autonomous driving in a simulated test track, for which you will have to apply many things that you have learned during the course. So if you don't know how to proceed with some parts of the assignment, check with the previous lab manuals and lecture slides.

The purpose of this assignment is to assess your skills developed during this course on using Linux, Git, C++, ROS, and on performing software development tasks, reading documentation, etc. You will need to solve practical tasks by reusing the skills learned in the previous assignments, which should now give you a sufficient foundation to integrate functionality from OpenCV and PCL into your project. We will review your solution, resulting in a pass/fail for this final assignment.

Good luck and have fun with the assignment!

## 1.1 Notation

Something about the notational conventions used in this manual (and which should be familiar by now): anything you should see on your screen is shown in `Courier`. All commands in Linux are *case sensitive*, which means that there is a difference between a small "a" and a capital "A". This also holds for file names. Anything you should type into your terminal is shown in Courier too, but it's `underlined` as well. Keys you should press are typed in boxes, for example `Enter`.

## 1.2 Getting help

The practicum session is the main opportunity to get help. Outside practicum hours, please use the Brightspace forum to ask any question regarding the assignment. We will not answer questions via email. This way, everybody will have access to the same clarifications. Do not publish your own solutions to answer others, or ask the instructors if this is ok before doing that. If you find errors, ambiguously phrased exercises, or have another question about this assignment, please use the Brightspace lab support forum.

## 1.3 Submission

You will make this practicum assignment in pairs. and you will have to submit your solution using git. To submit, do not send anything to the TAs or lecturers, but ensure that you and your partner's work is in the *master* branch in your group's private repository.

Like in the previous lab assignments, the submission criteria listed apply here too:

1. Your group's work must have been placed in the group's private repository on the TU Delft Gitlab server, provided by us. So, do *not* use an account on gitlab.com, github.com nor bitbucket.com or any other hosting solution.

2. You will be given a *new repository* for this lab, so do not reuse the repository from the previous lab.

3. All files you created for the assignment must be included in the `master` branch on Gitlab, which is the only branch that will be checked. While previous labs you could still push to other branches after the deadline and ask your peers, **we will *not* check any other branches, so do not ask us. Upload your work in time, and *check* if your master branch on Gitlab is correct.**

4. Use the functions and parameters that are provided in this manual. These functions and parameters are **not** a suggestions, these are **requirements**. It is **not** allowed to use alternative functions or parameters.

5. Do *not* add code to your repository that has been provided by us (e.g. the simulator).

6. Make sure you only commit **your own code**. And *your own code* means two different (but related) things here: 1) You are (of course) **not allowed** to commit something you copied from another (current or past) student. This goes without saying. 2) It also means: make sure you **do not commit code written by your team mate**. In conclusion, *all* code in your commits, and *only* code in your commits, should be yours.

7. Unlike the previous lab assignments, most of the work in this final assignments is done independently by each lab partner to demonstrate that each has individually passed the learning objectives, and also to ensure that the pass/fail of one lab partner does not affect the other. Therefore, there you should be a clear separation of concerns, and **you should agree on a division of work before you start working**: one team member works on the PCL perception package, the other on the OpenCV perception package. Both team members are allowed to work on the control package. **Clearly state who worked on which packages in the README.** This will assure that both lab partners implemented and committed a significant amount of C++ code that is part of your completed solution on *master* branch.

8. As a consequence of the previous two points, **only your commits may create or update your perception package**, and you are not allowed to alter the other perception package. You *can* help *test building and running* your lab partner's package, and for instance raise issues for them to fix on Gitlab. Still, they should fix any problems in their code, and you in your own code. We use the commit history of your Lab 4 repository to verify who worked on what, which should be in agreement with what

you state in the README. This directly influences how your contributions are graded. As an extreme example: If one team member commits everything, we don't see any work done by the other team member, and **consequently, we'll grade it as if the non-committing team member has done nothing (i.e. fail)**. This also means that you should not be over-eager: doing your lab partner's work will *not help them, and can prevent them from passing the course*. Once you have committed code for them (even if you later revert it), we would not be able to verify that your partner was able to solve the problems his or herself. For this reason, again, committing code on the other perception package is explicitly not allowed, and can also prevent *you* from passing the course.

9. Your solution must be able to work in the provided Singularity container, just like last week with the ROS assignment. See Section 3 for the exact commands that we will run to build and test your solution, and which should work without any further interaction. Singularity and the image is also installed on the lab computers on the Cornelis Drebbelweg. Thus, if you are working on any other computer, make sure to compile and test your solution from scratch on the lab computers before the deadline.

10. You may only hand in your own work. You are responsible for *not* sharing your code with other students outside your group. Only use the Gitlab repository that we will provide to collaborate with your partner. If we believe that you have used material from other groups, or that you have submitted material that is not yours, it will be reported to the exam committee. This may ultimately result in a failing grade or an expulsion.

11. If code is submitted that was written with ill intent, e.g. to manipulate files in the user's home directory that were not specified by the task, or which collects user passwords, you will immediately fail the course.

## 1.4 Deadline

The deadline for this assignment is **Sunday, October 24, 23:59**. **This is a hard deadline. We strongly recommend that you start in time.**

We will automatically restrict access to the *master* branch of your Gitlab repository after the deadline, which means that you cannot push any new changes anymore! Only the quality of the code in the last version on *master* branch will be reviewed, not in earlier commits or other branches.

## 1.5 Evaluation

In this final assignment you will need to demonstrate the practical skills that you have learned. The evaluation procedure is as follows:

- This assignment will be graded *pass* or *fail*.

- Your solution to this assignment will be reviewed within the weeks after the deadline.
- In case we find problems with your solution during the evaluation, we will add issues to the issue tracker, you then receive additional time to correct the issues. Pushing to *master* branch will be restricted after the deadline. You will have to create a new branch with the name *fixes-after-review* and push your fixes to that branch.
- If your solution is not fulfilling the minimum submission requirements from Section 1.3, or is so incomplete that we cannot create clear specific issues required to make it work, **you will fail the course immediately**. **You** are responsible for complying, do double check Section 1.3, and that your code was properly uploaded to your Gitlab repository.

To pass the course, possibly after the correction round, your solution must fulfill the following reviewing criteria:

- Your code should be of good quality. All nodes should be implemented in C++. It should not be overly complicated, be well organized, and easy to follow. It should not contain compile-time, link-time or run-time bugs.
- Your solution should work as described in the manual. You are free to add additional functionality (which will not count in your grade), as long as it at least contains the required functionality.
- Your solution should have good documentation, and clear comments where necessary. The README.md should be well formatted and explain who worked on what packages, how to build your code, and how to run it. Each package, node, class and function should have a short description of its intended function.

# 2 Task description

In this assignment we will work again with a virtual controllable Prius vehicle, driving through a virtual environment. The vehicle must use its sensors to detect obstacles and pedestrians, and use these detections to generate control instructions. This way, the vehicle should drive automatically between a path outlined by cones/barrels, without hitting any obstacle or pedestrian.

The virtual vehicle has two main sensors: a front-facing camera behind the windscreen, and a 360 degree top-mounted lidar. The vehicle receives camera images and lidar pointclouds as ROS topics, and can also be controlled by publishing control messages to a specified topic.

The simple control module makes the car move away from construction barrels in the simulated environment, by steering right if there is a barrel to the left, and vice versa. Whenever there is a person in right in front of the car and close by, brake and stop driving. Thus, these are the conditions:

- Drive forward if no objects or persons are detected.
- If an object is detected close-by and to the left-front of the car, and no persons are detected: Drive forward and steer to the right.
- If an object is detected close-by and to the right-front of the car, and no persons are detected: Drive forward and steer to the left.
- Brake in case a person is detected close to the car (e.g. the area of the detection is large).

For your solution, you will have to implement three ROS packages:

- An `opencv_person_detector` package, containing the node to detect 2D bounding boxes of all persons in a camera image using OpenCV. Details are discussed in Section 2.2.
- A `pcl_obstacle_detector` package, containing the node to detect 3D bounding boxes of all barrels in the lidar pointcloud using PCL. Details of this package are found in Section 2.3.
- A `control_barrel_world` package, containing the node to control the vehicle, by subscribing to 3D barrel detections and 2D person detections and publishing control messages. This package is discussed in Section 2.4.

All nodes should be implemented in C++. The relation between these nodes is visualized schematically in Figure 1. To express 2D detections of pedestrians in the image, and 3D detections of barrels in the pointcloud, we will use the quite new standard ROS package `vision_msgs`[1], which contains message types for 2D/3D bounding box detections.

[1] `http://wiki.ros.org/vision_msgs`

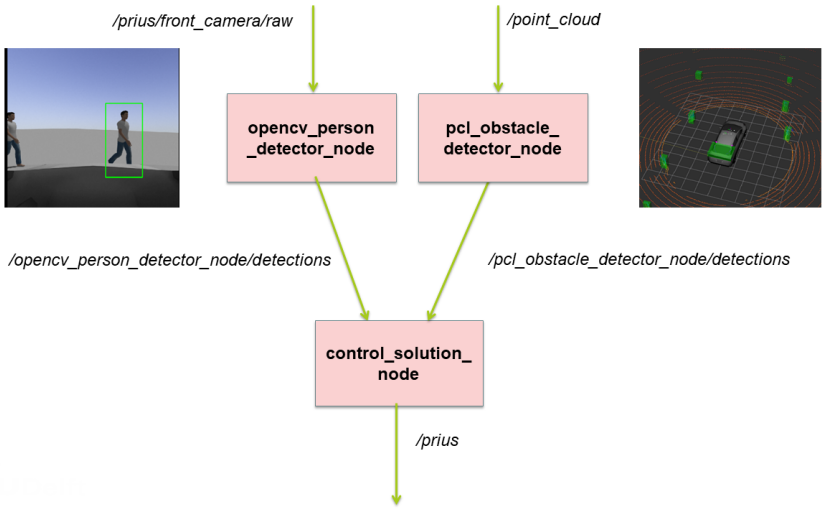



Figure 1: Schematic overview of ROS nodes (rectangles), publishing and subscribing to ROS topics (arrows).

**Tip** Unlike the previous manuals, this manual contains *no exercises*, but only lists the requirements of your solution. Therefore, you do not have to implement the ROS nodes in the order that they are listed here.

## 2.1 Getting started

Create a new ROS workspace. Clone the simulator repository, which was also used in the previous ROS assignment,

`git@gitlab.ro47003.3me.tudelft.nl:students-2122/ro47003_simulator.git`

.

**Git pull** the simulator repository before you get started on this assignment to get the latest changes.

Compile the simulator packages with `catkin_make`.

Launch the `simulation_barrel_world.launch` file in the provided `car_simulation` package. This launch file starts Gazebo, a simulator, which is running headless. If everything works out, you will see the car and the environment in Rviz. Check that topics are being published with `rostopic list`. Make sure that this works and that there are no errors before proceeding to the next sections.

It is also possible to run gazebo with a Graphical User Interface (GUI), by explicitly setting the `gazebo_gui` parameter:

```
roslaunch car_simulation simulation_barrel_world.launch gazebo_gui:=true
```

Running the Gazebo simulator with its GUI enabled is typically slower then when running Gazebo headless. However, if the Gazebo GUI is focused (e.g. after you clicked on its window with the mouse), you can

- Reset the simulation using CTRL-R to avoid stopping and relaunching the simulator.
- Interact, add, move or remove the obstacles around the Prius.

Alternatively, you can test in different worlds. This may be convenient while working on specific parts of the assignment. Pass the `world_name` argument to `simulation_barrel_world.launch`, like one of these:

- `world_name:=person.world`
- `world_name:=barrels.world`
- `world_name:=barrel_left.world`
- `world_name:=barrel_right.world`

## 2.2 Person detection package

Create a new package with the name `opencv_person_detector`. Setup the dependencies, e.g. on OpenCV in the `CMakeLists.txt` and `package.xml`. You will have to create a ROS node with the name `opencv_person_detector` that:

1. subscribes to the image topic `/prius/front_camera/image_raw`
2. processes the received image to find persons
3. publishes the detections as a ROS topic

**Subscribing to the image topic** Run the simulation using car_simulation `roslaunch car_simulation simulation_barrel_world.launch` .

Check that the `/prius/front_camera/image_raw` topic is published using `rostopic list`. The image should also be visible in Rviz, in the `front_camera` panel on the right, as in Figure 2. Your ROS node will have to subscribe to this topic[2].

**Detecting persons in the images** You will use the HOG detector to detect persons in the `front_camera` images. The OpenCV documentation for the HOG detector for person detection[3] shows the available functions. The referenced example[4] can be used to understand how to create, initialize and use the `HOGDescriptor` object to detect humans in a 2D image. It uses the `defaultPeopleDetector` retrieved using the `getDefaultPeopleDetector()` function.

In your implementation, use the following parameters in your call to the `detectMultiScale` function:

[2]See `http://wiki.ros.org/image_transport/Tutorials/SubscribingToImages`
[3]See `https://docs.opencv.org/3.2.0/d5/d33/structcv_1_1HOGDescriptor.html`
[4]See `https://github.com/opencv/opencv/blob/3.2.0/samples/cpp/peopledetect.cpp`

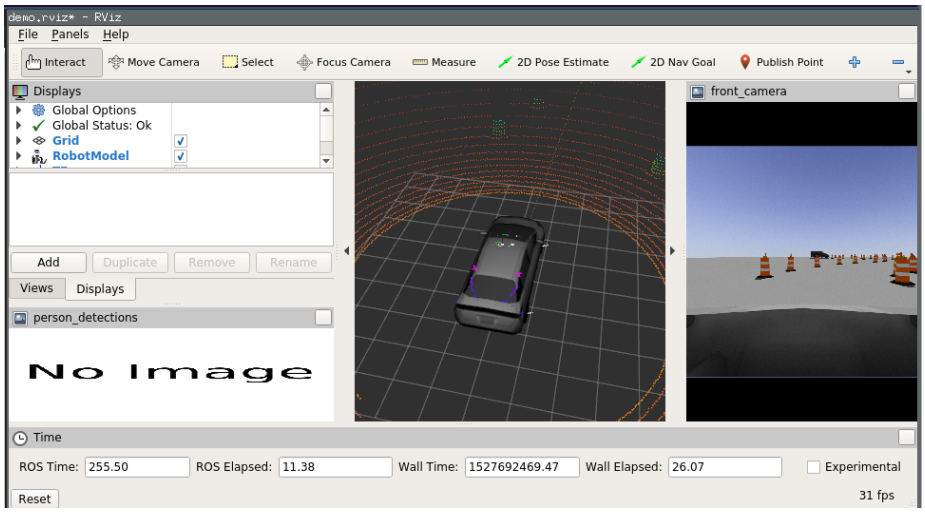



Figure 2: Screenshot of Rviz with the image visible on the right panel with the name 'front_camera'.

- A hit threshold of 0
- A win stride of (8, 8)
- Leave the other parameters to their default values

**Publishing a visualization of the detections** Draw green rectangles on your input image where persons are detected. Use the `cv::rectangle` function[5]. Publish the resulting images with the green rectangles on the topic `/opencv_person_detector_node/visual`[6]. Rviz is already configured to subscribe to the `/opencv_person_detector_node/visual` topic. Once you publish your drawn detections, they should appear in Rviz like in Figure 3.

**Publishing the detections** A detection is of the OpenCV type `cv::Rect`. Convert your detections to the more general `vision_msgs/Detection2DArray`[7], and publish the detections on the topic `/opencv_person_detector_node/detections`. Note that the `detections` field may contain fields such as `results` and `source_image` that you do not need. You can leave these fields empty.

[5] See `https://docs.opencv.org/3.2.0/d6/d6e/group__imgproc__draw.html#ga346ac30b5c74e9b5137576c9ee9e0e8c` and the people detector HOG example linked earlier

[6] See `http://wiki.ros.org/image_transport/Tutorials/PublishingImages`

[7] See `http://docs.ros.org/api/vision_msgs/html/msg/Detection2DArray.html`

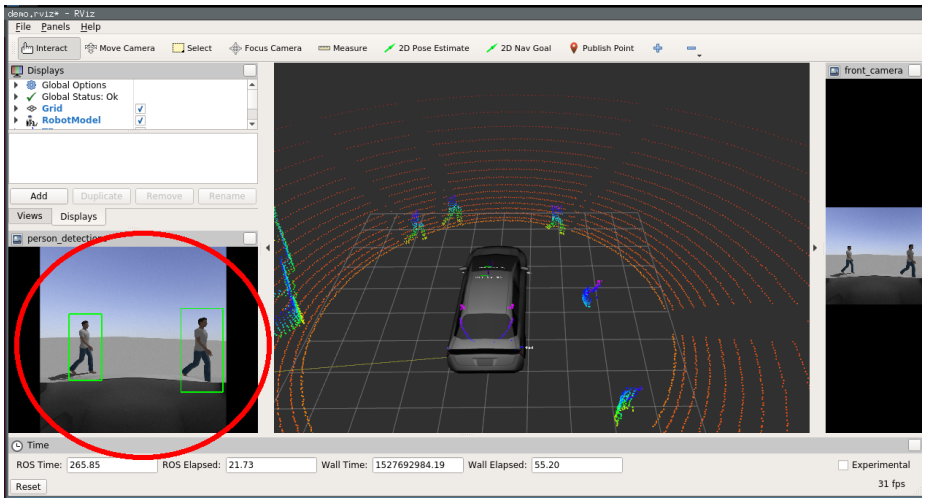



Figure 3: Screenshot of Rviz with a visualization of the detected persons in the image.

## 2.3 Obstacle detection package

Create a new package with the name `pcl_obstacle_detector` . Setup the dependencies in the `CMakeLists.txt` and `package.xml`. You will have to create a ROS node with the name `pcl_obstacle_detector_node` that:

1. Subscribes to the point cloud topic `/point_cloud`
2. Processes the received point cloud to find clusters (mainly barrels)
3. Publishes the clusters as a ROS topic

**Subscribing to the point cloud topic** Run the simulation using `roslaunch car_simulation simulation_barrel_world.launch`.

Check that the `/point_cloud` topic is published using `rostopic list`. The point cloud should also be visible in the main window in Rviz, as in Figure 2. Your ROS node will have to subscribe to this topic.

**Ground plane removal** Use PCL to segment the ground plane[8]. Filter the point cloud to only keep objects that are not part of the ground plane. Regarding the parameters for your algorithm, in the `SACSegmentation`, set the distance threshold to 0.3. The remaining points can be used for cluster extraction.

[8] See `https://pcl.readthedocs.io/projects/tutorials/en/pcl-1.12.0/planar_segmentation.html`. Note: this is the documentation for PCL version 1.12, while version 1.8 is used by ROS Melodic. The example shown on this page also works for version 1.8 of PCL.

**Euclidean cluster extraction** You will use Euclidean cluster extraction from PCL to detect clusters (mainly barrels) in the filtered point cloud after ground plane removal. Read how the example application is implemented using PCL[9], and implement it in your node. Use the following parameters in your algorithm:

- Set your cluster tolerance to 0.5.
- Set your min cluster size to 10.
- Set your max cluster size to 25000.

**Publishing the clusters** A pointcloud cluster in PCL is of the datatype `pcl::PointCloud<pcl::PointXYZ>`. Each cluster must be represented as a 3D bounding boxes. You can convert the clusters to bounding boxes as follows:

1. Use the PCL function `pcl::compute3DCentroid` to calculate the center of the cluster, and assign this value to the `center.position.center` field of the message.
2. Use the PCL function `pcl::getMinMax3D` to calculate the extrema of the cluster, and use it to fill the `bbox.size` field of the message.

Publish the detected bounding boxes as `vision_msgs/Detection3DArray`[10] on the topic `/pcl_obstacle_detector_node/detections`. Note that the `detections` field may contain fields such as `results` and `source_cloud` that you do not need. You can leave these fields empty.

The algorithm actually detects any kind of separated objects, not only barrels. The trunk of the car also generates some points in the lidar scan, which are being detected as a cluster. However, these clusters will be ignored by the `control_barrel_world` package.

**Tip** The provided `detection_3d_to_markers_node` can subscribe to the 3D detections that you publish, and visualize that as green boxes in Rviz, as can be seen in Figure 4. Use it to debug your solution!

## 2.4 Prius control package

Create a new package with the name `control_barrel_world`, and setup its dependencies in `CMakeLists.txt` and `package.xml`. Create a node containing a class that will control the vehicle. Subscribe to both the `/opencv_person_detector_node/detections` and `/pcl_obstacle_detector/detections`. Create a publisher on the topic `/prius` that sends `prius_msgs/Control` messages. The next section describes how to you should fill the content of the control message.

[9] See `https://pcl.readthedocs.io/projects/tutorials/en/pcl-1.12.0/cluster_extraction.html`. Note: this is the documentation for PCL version 1.12, while version 1.8 is used by ROS Melodic. The example shown on this page also works for version 1.8 of PCL.

[10] See `http://docs.ros.org/api/vision_msgs/html/msg/Detection3DArray.html`

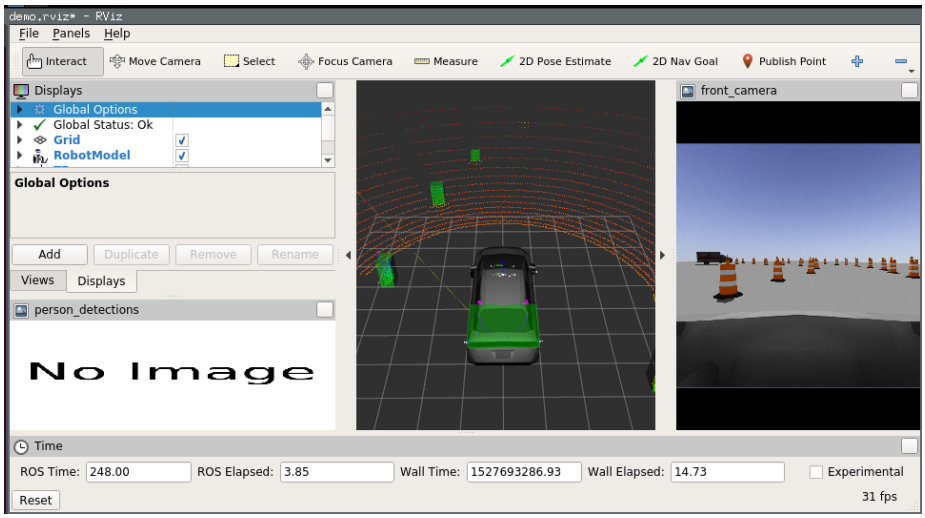



Figure 4: Screenshot of Rviz with a visualization of the clusters as green boxes in the main panel.

**Control algorithm** Your control class should send a `prius_msgs/Control` message on the `/prius` topic. Implement the control class in such a way that the node's behavior is the following:

- Filter the 3D obstacle detections, and keep only the detections with a center in front of the car, meaning `x > 0` and within a radius of four meters.
- If this filtered set of 3D detections is empty, drive forward, meaning `throttle = 1` and `steer = 0`.
- If this filtered set of 3D detections is equal to or larger than one, get the object closest to the car.
- If the center y-value of the obstacle that is closest to the car is positive, steer right and drive, meaning `steer = -1` and `throttle = 1`. Otherwise, steer left and drive.
- Once a person is detected, and the area of any of the detections is larger than 25000 pixels, the car should brake, not steer, and never start driving again.

# 3 Requirements for the solution

Recall that each lab partner should fully implement one of the perception packages (OpenCV or PCL), while you can collaborate on the control package (carefully check Section 1.3 for the details). As part of the task, you will each need to show that you can split your implementation over multiple source files.

Therefore, for each node that you implement:

1. One source file should only implement the main() function for your ROS node, and no other functions or classes.
2. All other functions and/or classes that you write should be placed in another (or if you want multiple) separate source file(s).
3. Create corresponding header files for each of these source file(s). You can include these in the source file with the main().

Your repository should have the directory tree as can be seen in Figure 5. All of a node's header files must be placed in its package's include/ directory, and all source files in its package's src/ directory. Take note of the fact that your repository *only* contains the packages you created, not the packages we have given you (the simulator). Also make sure you do *not* commit the `build` and `devel` directories in your workspace to your repository.

You will **not** have to hand in a report. However, you will have to write a README.md file containing clear instructions on how to compile and run your solution. The README.md will also mention who worked on what packages, and contain a short description of your approach, so that we can understand the reasons for the choices you made. Provide clear comments in your code.

Assuming that you do not have a directory ∼`/catkin_ws` yet, the following code should compile and run your solution:

```
source /opt/ros/melodic/setup.sh
cd
mkdir -p catkin_ws/src
cd catkin_ws/src
catkin_init_workspace
git clone <your repository url>
git clone git@gitlab.ro47003.3me.tudelft.nl:students-2122/ro47003_simulator.git
cd ..
catkin_make
source devel/setup.sh
roslaunch control_barrel_world solution.launch
```

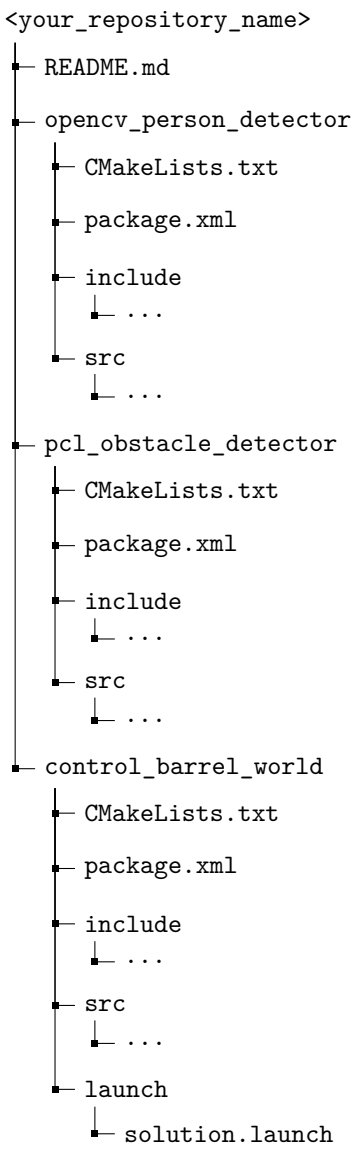



Figure 5: The expected directory tree of the repository that you will create for the assignment.