Åbo Akademi

Multidimensional Sensing Techniques

Assignment N°4: LIDAR points projection

Cédric Léonard, 21/10/2021, student ID: 2100775

Introduction

This 4th and last Multidimensional Sensing Techniques assignment aims to get familiarized with calibration data in order to project LIDAR points on a picture. This assignment uses python code and **OpenCV** to calibrate the point projection from lidar data onto the video camera image and overlay the two. You can find additional informational on the following GitHub repository.

What you did?

Set up my work environment

As always, the first steps have been about setting the environment in order to run the code. After downloading the code and the data from Moodle we could either work under *Google Colab* either under Anaconda and *Jupyter Lab*. If *Google Colab* was easier to set up and more user-friendly it was also much less appropriated for this assignment. I chose to use **Anaconda**, install the **OpenCV** package open *Jupyter Lab* in local, find the folder where all the code and data were stored and I was finally able to run the first 2 tabs of the code. As I summarized all these steps in one sentence it might sounds easy, but it wasn't and I had plenty of problems with **Anaconda**.

Code explanation

The provided code was divided into 3 tabs, which allows us to independently run each of them inside the *Jupyter Lab* environment. The first tab of code was only importing packages, defining functions to read and store file content or converting angles to a rotation matrix. At the end of this first tab the image and the LIDAR points were loaded inside the kernel.

The second tab of code was almost optional as it was only displaying different projection of the LIDAR point, particularly a 3D plot of the points which was really useful.

And finally, the last and third tab of code was the one I had to modify. It first includes some parameters initialization, such as the camera axes, its rotation and translation relative LIDAR axes, etc. Those parameters define the calibration between the camera and

the LIDAR and that's what we will have to modify in order to correctly calibrate these 2 entities. Below these parameters initialization, a function `createProjection()` creating a 2D projection of the LIDAR points onto the camera picture is defined using **OpenCV** features. Finally, a loop is running, constantly refreshing the window displayed by `createProjection()` but also listening to the keyboard allowing us to interact with the displayed window.

This feature, allowing us to interact with the displayed window, was really important as it allows us to slightly modify each parameter by mapping a key to increase or decrease a parameter:

```
key = cv2.waitKey(100)      # Wait for key event
if key == 97:               # If 'a' key is pressed
   rotation[1] += 0.002     # Increase the x axis rotation of the camera
elif key == 122:            # If 'z' key is pressed
   rotation[1] -= 0.002     # Decrease
```

That feature explains why I chose to use *Jupyter Lab* and not *Google Colab*. Indeed, in *Google Colab* it's impossible to have input in the running loop and therefore to modify in real time a parameter. Another solution was available, but was much more complicated.

Camera axes and basis

This assignment was pretty different from the previous ones as there was almost no coding but, a lot of thinking to understand what was happening and how to modify a parameter in order to get the desired behavior. The 3 first questions did not require any code modifications and are basically explained above. After running the second code tab I was able to visualize the LIDAR points in a 3D plot. It took me some times to realize that the points were showed from behind and that the camera and the LIDAR shared the "same" coordinates in our real-world base.

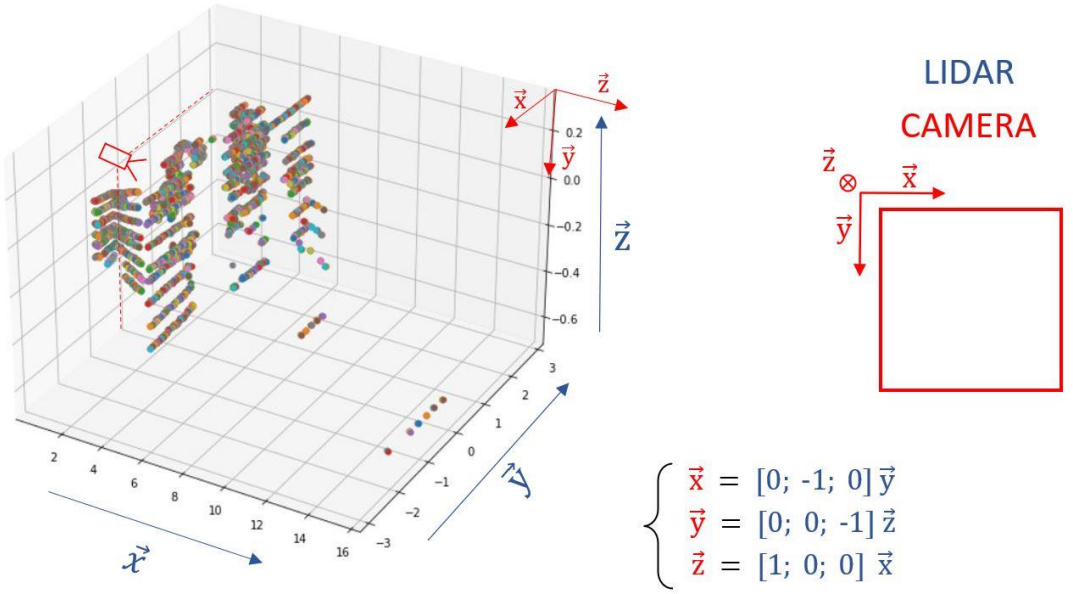

Figure N°1: Scheme explaining the rotation matrix and the change of basis between LIDAR and camera

We can see on Figure N°1 our real-world basis, in blue, corresponding to the basis in which the LIDAR points are located in absolute coordinates. And in red the camera basis as described in the *Assignment4_Instructions.pdf*, *"OpenCV assumes that coordinates from 3d points show depth at z-axis, right on the x-axis and downwards on the y axis."*.

From this information we can deduce the following rotation matrix:

```
camera_axis = np.float64([[0, -1, 0],  # x = -y
                          [0, 0, -1],  # y = -z
                          [1, 0, 0]]) # z = x
```

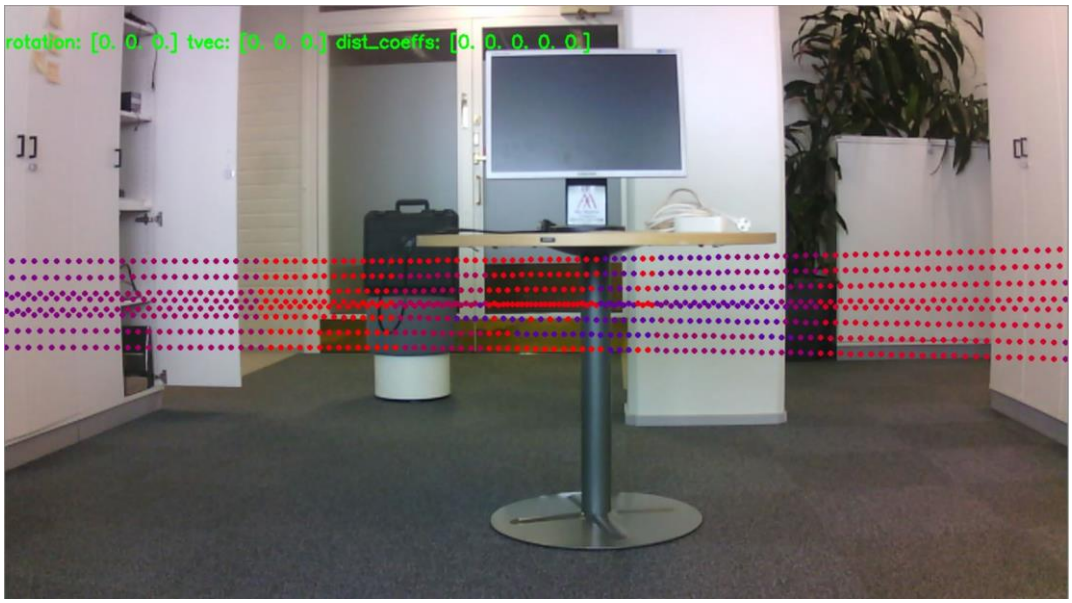

Figure N°2: Screenshot of output window after initializing the rotation matrix

Figure N°2 presents a screenshot of the projection of LIDAR points onto the camera image after initializing the rotation matrix. LIDAR points are projected as horizontal lines because we have not initialized any rotation, translation or distortion coefficient parameter yet. So, the image on Figure N°2 assumes that the LIDAR and the camera have the exact same coordinates.

Parameters adjustment

Then came the hardest part: finding proper values to `rotation`, `tvec` and `dist_coeffs` so that the points in the image are properly overlaid on top of the objects present in the image. Without waiting, Figure N°3 shows the best result I could get:

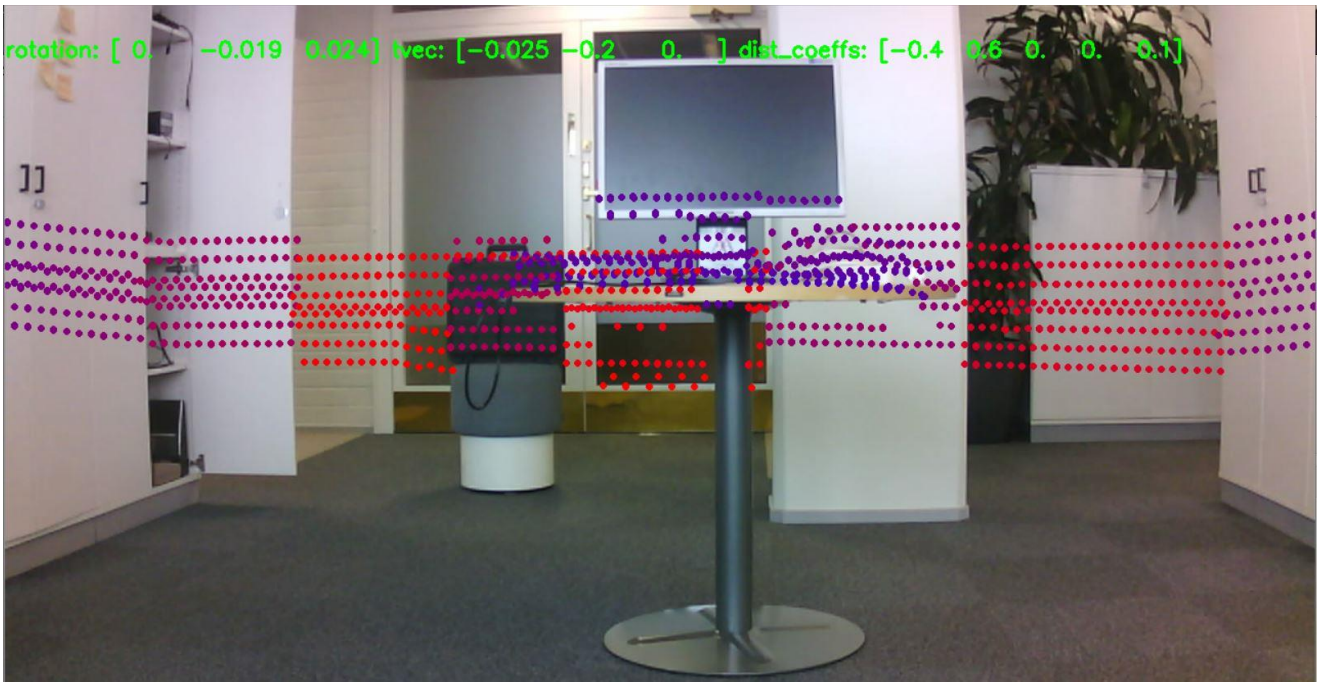

Figure N°3: Screenshot of the best overlay I could get between LIDAR points and camera image

The parameters used for such a calibration are the following:

- $rotation = [0, -0.019, 0.024]$;
- $tvec = [-0.025, -0.2, 0]$;
- $dist\_coef = [-0.4, 0.6, 0, 0, 0.1]$.

I had in *Assignment4_Instructions.pdf* the information that the camera is slightly tilted downwards and to the side and is placed slightly under the lidar. Therefore, I deduced that the rotation vector must have a negative value on the $\overrightarrow{y_L}$ axis and some value on the $\overrightarrow{z_L}$ axis (for the side tilt). It is really important to note that we are modifying `rotation` relative to the LIDAR basis and not to the camera basis because the rotation is first transformed in a matrix before getting transformed in the camera basis with a multiplication by `camera_axis` and then computed in `rvec` (rotation vector) with **OpenCV** `Rodrigues()` conversion function. This process is realized by the following code:

```
rvec = cv2.Rodrigues(camera_axis * eulerAnglesToRotationMatrix(rotation))[0]
```

About `tvec`, I know that the camera is under the lidar, around 10 or 20 centimeters, so $-0.1\overrightarrow{y_C}$ or $-0.2\overrightarrow{y_C}$ and by dint of testing I found `tvec` values. It's important to note that we are this time talking about the camera basis. For `dist_coef` different values were proposed in the subject, I selected the best fitting ones by testing.

Everything is not perfect because there is an inversion in the color scale about the points distance to the LIDAR. Indeed, it's stated in the subject that *"red corresponds to close distance and blue corresponds far away dustance"* but we can identify the exact inverse in Figure N°3. However, Figure N°2 (just before modifying the intrinsic matrix of the camera) matched what we were supposed to get in the subject.

How your solution can be improved/generalized?

This kind of calibration exercise can always be improved by spending more time adjusting parameters. As we can see on Figure N°3, most of the LIDAR points are in the good region of the camera image but some of them are not perfect.

In order to generalize this solution, we can think of a method to do step by step what I have done in this assignment. This kind of method may require more details about the real-world camera position or its angle compared to the LIDAR.

What did you learn? Did anything surprise you? Did you find anything challenging? Why?

In this assignment I discovered *Jupyter Lab* and more generally some features of **Anaconda** and it was surprisingly not that user friendly, but therefore challenging. I also "re-learn" to play with my fingers in order to perform basis changes and state about the validity of some potential basis, and I have to admit it was satisfying. In a more general way, this assignment was really about thinking how was the problem and its environment, understanding what we have to change in order to perform such or such action and then try it and adjust the "reality" of the overlay to our theory. This was interesting, especially the mathematical part, but in the realization that was more trying some approximation, failing, trying again, failing, trying something different, etc. satisfying at the end but frustrating during the process.