University of Girona - MIRS

Lab2 – Laser Scanner

3D Perception and Sensor Fusion

Authors: Ricard Campos and Josep Forest

Sensor: camera + laser (active)

Programming language: Python

1 INTRODUCTION

In this session, you will build a laser scanner. We will identify the different steps required for both **calibrating the laser** scanner and also for **reconstructing** the **3D** data using this type of scanner.

First and foremost, please be aware that we will be **using a laser that it is NOT eye safe**. Therefore, **AVOID ANY DIRECT EYE EXPOSURE!**

That said, you will find a code base in the following GIT repository:

https://bitbucket.org/ricardcd/3dpsf_lab_2_laser/

However, in this case, it is not complete. If you open the different codes, you will see that, in most cases, they depend on functions defined on the "my_library.py" file. However, most functions in this file are empty, and you need to implement them as you proceed with this practical activity. You are assumed to have some knowledge about NumPy arrays and the basic operations you can do with them. Otherwise, please review its documentation before this lab session.

As in the last lab, the idea is for you to follow this document and implement/run all the steps first with the sample data we provide. Then you are supposed to **capture some data yourself**, both for the calibration and for reconstruction, and re-execute the code you developed on this new data. You will have two days for this assignment. Ideally, you should work on the lab the first day, continue working at home, and leave the second session just for last-minute doubts and for capturing data and check that your pipeline works.

2 PREPARING THE ENVIRONMENT

You will find the code and sample data to work in this assignment in the Moodle page of the course. Please download the sample data and go to the link provided above to gather the base sources.

We use the same framework as in Lab 1, so please *refer to the first lab in case you do not remember how to prepare your environment.*

As a summary:

- Create a virtual environment:

```
python -m venv venv
```

- Activate it:

```
(windows) .\venv\Scripts\Activate.ps1
```

```
(linux) source ./venv/bin/activate
```

- Install requirements:

```
pip install -r requirements.txt
```

- To use any of the scripts, from the folder containing the codes:

```
python <script_name.py>
```

3 THEORETICAL BACKGROUND

Our simple laser scanner is a regular (monocular) camera rigidly coupled with a laser projector. The laser projects a laser stripe into the scene, and the idea is to detect the deformation that this laser presents on the image in order to recover its 3D shape.

There are two separate steps in the calibration of the scanner. First, we need to **calibrate the scanner**, which requires **calibrating the camera**, and then **calibrating the laser plane** with respect to the camera. Once calibrated, by detecting the laser stripe in an image, we can **reconstruct the 3D shape of the object where the laser projects**.

We present in this section a brief overview of the steps required for both calibrating and reconstructing 3D measurements using our laser scanner.

3.1 CAMERA CALIBRATION

The camera calibration is exactly the same as in the first lab of this course. However, to make it a little different, in this case we use another type of calibration patterns named ChAruco patterns. The ChAruco pattern mixes makers normally used for Augmented Reality (that is, fast and easy to detect) with the chessboard pattern we saw in the first lab. The white squares of the chessboard patters are now filled with markers. Since the *markers* are located in a pre-defined position on the pattern, by finding one or more of the *markers*, we have a clue for where the different *corners* of the chessboard are (if you recall, this is the info we used for calibrating in the first session).

Once calibrated, these patterns also help in determining the pose of the camera (we will use this later on). If you know the 3D of an object (the world coordinates of the pattern), and its 2D projections (the detected corners on the image) on a calibrated camera, you can obtain the 3D pose (rotation and translation) of this camera. This is called the Perspective-n-Point problem.

3.2 LASER PLANE CALIBRATION

As shown in Figure 1, the laser stripe we see in the scene when turning on the laser is the intersection between the “plane of light” emitted by the laser and the scene. In order to be able to reconstruct the laser, we need to find the plane of the laser with respect to the camera.

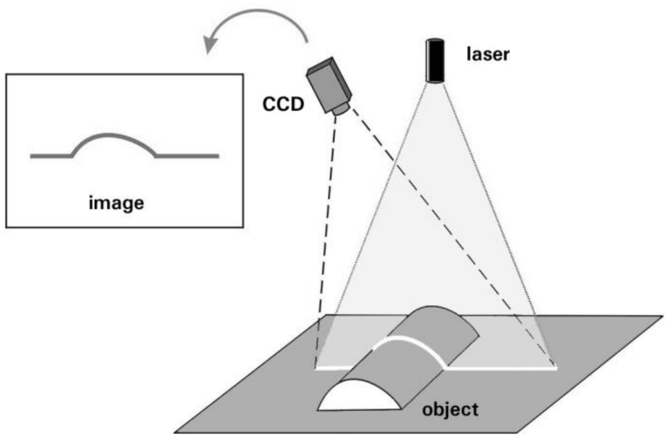


Figure 1. Laser scanner principle: the laser plane intersects the scene, and the camera captures the laser stripe. Image extracted from: Raffaella Fontana, Maria Chiara Gambino, Marinella Greco, Enrico Pampaloni, Luca Pezzati, Roberto Scopigno, "High-resolution 3D digital models of artworks," Proc. SPIE 5146, Optical Metrology for Arts and Multimedia, (9 October 2003)

How can we get some samples on the laser plane in order to reconstruct it? We have just seen that we can obtain the pose of the pattern using its known geometry and the camera calibration. If we project the laser to the same plane as the calibration pattern, and we know the pose of this pattern, we can obtain the 3D points of the laser at that pose. (see images in “sample_data/calibration/laser” for an example).

To summarize, for calibrating the laser we need the following procedure:

- Given a set of images seeing the laser projected on the same plane as the calibration pattern at different depths/poses:
    - **Compute the pose of the calibration pattern** and get its XY plane in 3D. This is the same plane as the laser stripe.
    - **Detect the laser stripe** on the image.
    - **Back-project** the pixels corresponding to the laser stripe in the image, the result is a set of rays (also known as “lines of sight”) originating from the camera center.
    - Compute the **intersection between these rays and the XY plane** of the calibration pattern. Collect these 3D samples.
- **Fit a plane to the set of 3D** samples at different (pattern) planes to extract the laser plane.

3.3 RECONSTRUCTING THE LASER

The procedure for reconstructing the laser in the camera plane is quite similar to the one used for calibration. Now that we have the laser plane, we can compute the intersection between the laser plane and the lines of sight originating from the camera and corresponding to the detection of the laser stripe in the image.

However, our laser scanner is static. Therefore, if we put it to observe a static scene, it will always see the same stripe of 3D points. Either the scene or the scanner must move in order to get a new part of the scene under the laser. Moreover, we need to know how the camera or the object moved with respect to each other, in order to be able to put the reconstructed profiles in the same global frame.

In industrial settings, the laser + camera are kept fixed and the object is moved below the camera by a conveyor belt. Since they can know, mechanically, how much the object moved between captures, they can reconstruct the shape of the object. In LIDAR lasers, the laser plane is swept along the scene with a mechanical mirror, and different profiles can be taken from a given position of the scanner (however, calibration is more involved).

In this lab, we will use another approach. Remember that we saw that we can estimate the pose of the camera from the ChAruco pattern? We will use the pose computed from the pattern to know where our laser scanner is. Therefore, for reconstructing the shape of the object, we will transform the 3D laser points reconstructed in the camera frame to be in the pattern frame. That is, we will consider the origin of the pattern as being the global or "world" reference frame. As long as we see (even if partially) the ChAruco pattern as well as the laser on the image, we will be able to reconstruct the laser points in a common global frame.

It may not be obvious at a first glance but, in fact, it does not matter if you move the object+pattern rigidly and leave the camera static, or you move the camera and leave the object+pattern static: the procedure is the same!

Thus, the method for reconstructing the laser is the following:

- Given a set of images seeing the laser and the ChAruco pattern in the same 3D plane:
    - **Detect the laser stripe** on the image.
    - **Back-project** the pixels corresponding to the laser stripe in the image, the result is a set of rays (also known as "lines of sight") originating from the camera center.
    - Compute the **intersection between these rays and the laser plane**. The results are the reconstructed 3D points of the laser in the camera frame.
    - **Compute the pose of the calibration pattern**.
    - Transform the 3D points in the camera frame to a common global frame via the pose of the calibration pattern.

4 EXERCISES

In the provided code, you will find scripts for:

- Calibrating the camera: "monocular_calibration_charuco.py"
- Calibrating the laser plane: "laser_plane_calibration.py"
- Reconstruct the 3D of the scene given the two calibrations above: "laser_reconstruction.py" (incomplete!)

You should open these scripts and check what they do. You will notice that all of them rely on functions defined on a file called "my_library.py". However, in this file, you will see that we just provide the signature for most of the functions, and their actual implementation is empty. In fact, the task in this practical session is to **implement all the functions missing in "my_library.py"**. Note that the functions to implement have a direct match with the key steps marked in bold in Section 3. In the following exercises, we will implement the required functions one by one. To ease your implementation, we provide individual test scripts that allow you to try each function and check its correctness/behaviour. Once all the functions are implemented, we will be able to calibrate the laser scanner. Additionally, these functions will allow you to implement the laser reconstruction itself, by **filling the missing part of "laser_reconstruction.py"**.

4.1.1 EXERCISE 1

Calibrate the camera using the "monocular_calibration_charuco.py" script. As mentioned in Section 3.1, we now use a different pattern. In order to calibrate using this ChAruco pattern, you need to know the number of "squares" (black+white) in the pattern, in the X and Y direction. You also need to know the ID of the *dictionary* from which we generated the markers, which in this case is dictionary *1* (you can leave out this argument if you want, as 1 is the default parameter). A dictionary is a set of pre-computed markers, you can find more information about this topic on the original article, and on the OpenCV docs.

Finally, you need to specify the size of the squares and the markers, which is 0.02m and 0.018m respectively.

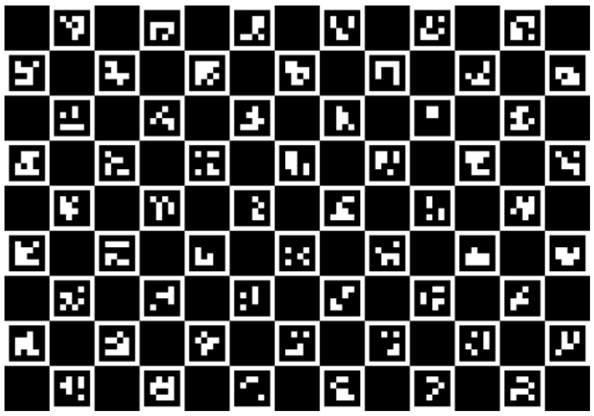
As in the previous lab activity, you can run the code with the "-h" option to get a list of required parameters. You may execute again the command with the "--debug_dir" option and comment on the calibration results. What are the advantages of using this type of pattern instead of the chessboard pattern of the first assignment?

4.1.2 EXERCISE 2

Use the calibration of the camera found in the previous exercise, and use it to detect the pose of the camera. This is one of the two functions already defined in "my_library.py", and it returns a flag indicating if it was possible to compute the pose, and the pose in the affirmative case. The returned pose is a transformation ${}^{C}T_{P}$ that transforms points from the pattern coordinate system (P) to the camera coordinate system (C).

Open the code and check how it works. We provide a script "test_camera_pose_charuco.py" for you to test this pose estimation. You should take the calibration of the previous exercise and execute it with any of the sample images to see its behaviour. You will see that we use the computed transform to draw the reference frame of the pattern as seen from the camera.

If the printed pattern is the following:



- Where is its origin?
- If we were to draw in 3D the frame of the pattern in this page, would the Z be pointing "outside" or "inside" of this page?

4.1.3 EXERCISE 3

We also provide in "my_library.py" the function "extract_xy_plane_from_charuco_pose", which takes the pose of the pattern and extracts the XY plane of the pattern from ${}^{C}T_{P}$ (we need this step for calibrating the laser plane).

Explain in your own words what this function is doing to extract the plane equation out of ${}^{C}T_{P}$.

4.1.4 EXERCISE 4

Figure 2 shows a synthetic image of a laser stripe projected onto a flat surface. As can be seen, there is a maximum of red concentration at the center of the stripe, vanishing outwards of the center. This is consistent with the light energy distribution of the laser stripe: maximum power at the centre of the stripe, gradually decreasing outwards off the stripe center.

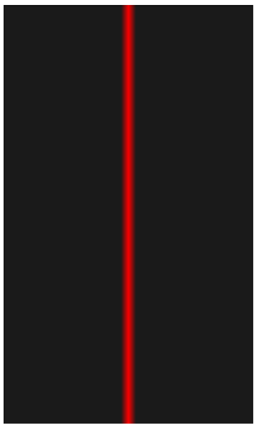


Figure 2. Laser stripe projected onto a flat surface, as seen by a camera.

Taking a horizontal cross-section of the image above, the laser light energy looks like a Gaussian bell, as shown in Figure 3.

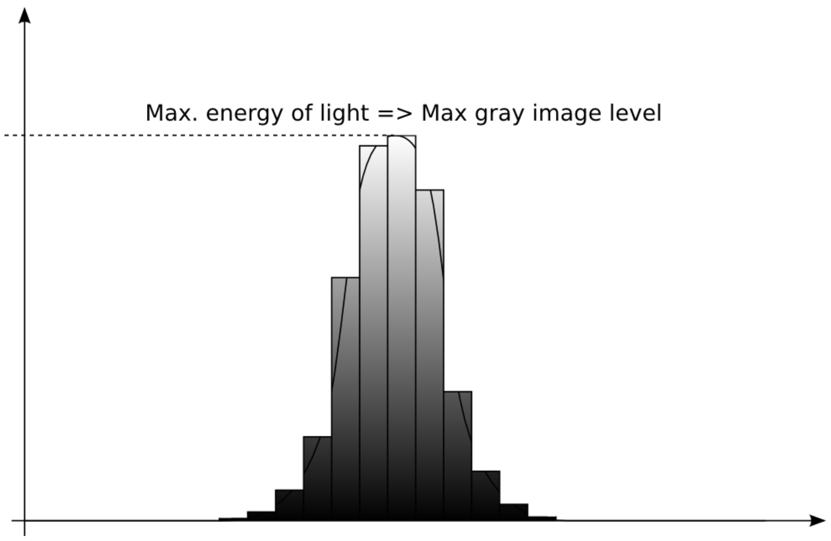


Figure 3. Quantized cross-section of the laser stripe as seen by the camera.

Detecting the laser means calculating an estimate of what is the vertical coordinate across the laser stripe that corresponds to the maximum energy or maximum gray level. Repeating the calculation for each row of the image, a set of laser coordinates are obtained. We will call this set a “profile”.

One possible way of obtaining the estimate of the maximum location can be using the weighted centre of gravity computation, or COG:

$$v_{COG} = \frac{\sum_i^N v_i g_i}{\sum_i^N g_i}$$

where $\mathbf{v_i}$ is the vertical pixel coordinate, starting from the top of the image till the bottom of it, $\mathbf{g_i}$ is the gray level corresponding to the pixel $\mathbf{v_i}$, and finally **N** is the number of pixels under the Gaussian curve.

To be practical, a threshold value should be defined in order to filter the background of the image so only the laser stripe is visible. The thresholding operation should leave the values above the threshold as they are, while the ones below should be set to zero. Since the laser is red, you only need to look for intense red on the images. A simple thresholding on the red channel of the image will be sufficient for our purposes.

This approach is shown in Figure 4, so that:

$$v_{COG} = (9\cdot g_9+10\cdot g_{10}+11\cdot g_{11}+12\cdot g_{12}+13\cdot g_{13}+14\cdot g_{14}+15\cdot g_{15}+16\cdot g_{16}) / (g_9+g_{10}+g_{11}+g_{12}+g_{13}+g_{14}+g_{15}+g_{16})$$

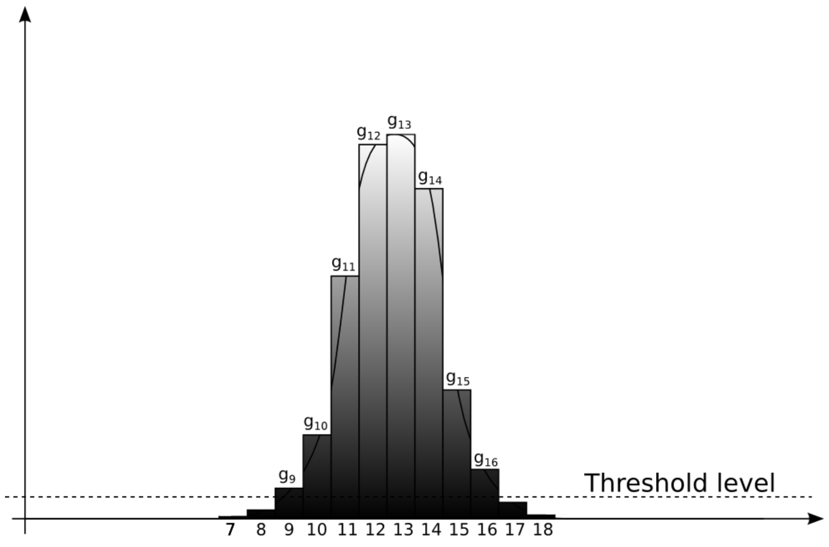


Figure 4. Laser stripe energy profile with quantized levels and threshold.

Considered this way, after thresholding, **N** turns to be the number of pixels of the stripe whose value is not zero.

With this information, **implement the laser detection using the COG method**. Your code must be implemented in function “detect_laser” of “my_library.py”, and you should adhere to the expected input and output as presented in the comments of the function. *Beware that OpenCV images are in BGR format (not in RGB!).* We provide the script “test_laser_detection.py” for you to check your implementation.

4.1.5 EXERCISE 5

The perspective projection describes how a 3D point is projected into the image plane. Since we calibrated our camera, in our case we have modelled this projection using a *pinhole* + *plumb bob* distortion model. However, now we detected the laser on the 2D image, and we want to compute its 3D position. Therefore, we want to *invert the projection model*, a procedure also known as *back-projection*.

However, as you know, depth information of a pixel is lost when projected to an image. Given a point p in an image, there is a collection of points that project on that same point. These points are placed along a ray connecting the camera center $C = (C_1, C_2, C_3)$ and the point $p = (x, y, 1)$. This homogeneous point $p = (x, y, 1)$ is the result of inverting the projection model. It is a homogeneous point in camera coordinates, that does not depend on the camera matrix. We call this point to be in *normalized coordinates*. Moreover, remember that homogeneous points are, in fact, rays in projective space.

Therefore, by computing *normalized coordinates*, we are mainly converting the points detected in an image into vectors originating from the camera center. You can scale these vectors by any factor, and they will always result in the same 2D projection on the image plane. Therefore, we can think of them as rays originating from the camera center (also known as *lines of sight*).

In OpenCV, the function used to invert the projection model is implemented in the cv2.undistortPoints function. Use the cv2.undistortPoints function to fill the function "get_normalized_coordinates" of "my_library.py" and **compute the direction of rays corresponding to a set of image points** (i.e., get their *normalized coordinates*). Beware of the shapes of the input/output matrices (you need to filter the output of the cv2.undistortPoints function with the squeeze function). Also, note that cv2.undistortPoints returns 2D *normalized* coordinates in 2D. You need to use the np.append or np.concatenate method to add a column of *ones* to the returned matrix in order to return the actual homogeneous coordinates we need. Use the provided "test_normalized_coordinates.py" script to check your implementation.

4.1.6 EXERCISE 6

Now that we can transform the detected pixels of the laser in an image into rays originating from the camera, we need to find the 3D locations of the laser along those rays in order to reconstruct the 3D of the scene:

- For the case of laser calibration, we will take advantage of the calibration pattern and the laser projection being at the same plane in order to compute the depth of the laser by intersecting the lines of sight of the laser and the plane of the calibration pattern.
- Once we have the laser plane calibrated, we will reconstruct the 3D position of the laser by finding the intersection between the laser plane and the camera rays of the laser image points.

Therefore, in both cases, we need to **implement a ray-plane intersection method**.

The parametric equation of a ray is:

$$p = p_O + td$$

Where p_O is the origin of the ray, and d its direction. Modifying the parameter t allows us to move along the ray.

And a plane, in normal form, is given by the following equation: $Ax + By + Cz + D = 0$. Note that the three elements of the normal of the plane $n = [n_1, n_2, n_3]$ correspond to the first three parameters of the plane equation in normal form, that is: $A = n_1$, B$= n_2$, $C = n_3$. Therefore, the equation can be rewritten as:

$$n \cdot p + D = 0$$

Where $n \cdot p$ is the dot product between the normal and a point. Substituting the first equation on the second, we have:

$$n \cdot (p_O + td) + D = 0$$

Solving for t, we have:

$$t = -(p_O \cdot n + D)/(d \cdot n)$$

Implement the ray-plane intersection equation above in the function "rays_plane_intersection" of "my_library.py". The function should accept multiple rays at the same time. We provide the script "test_ray_plane_intersection.py" for you to check your implementation.

Some considerations:

- The dot operator in Numpy is the '*' operator or the "np.dot(x, y)" function.

- Note that we are working with rays, not lines, so if $t < 0$ we are *behind* the origin of the ray, and therefore intersection is not valid.
- Check before solving for t if $(d \cdot n)$ is zero (or some small value Ɛ), as this means that intersection does not exist (you would divide by 0 in the equation above!).

4.1.7 EXERCISE 7

During laser calibration, we estimate the 3D positions of some samples of the laser in the calibration plane. Given this set of samples, we need to extract the plane of the laser light. Therefore, you are required to **implement a plane fitting method**.

You can obtain the parameters A, B, C and D of the plane equation $Ax + By + Cz + D = 0$ (normal form) applying the following procedure:

a) Compute the centroid of the points p_i:

$$centroid = \frac{1}{N}\sum_{i=1}^{N} p_i$$

b) Configure a matrix A as:

$$A = [p_1 - centroid, p_2 - centroid, \dots, p_n - centroid]$$

In Numpy, if you have the points matrix in point-per-row form, and the centroid is a column 3-element vector, you can simply do: A = points-centroid.

c) Obtain the singular value decomposition of A (np.linalg.svd in Numpy):

$$[U\ S\ V_h] = svd(A)$$

d) The normal vector of the plane $n = [n_1, n_2, n_3]$ is the third row of V_h. In Python: n = Vh[-1, :]. Note that the three elements of the normal of the plane correspond to the first three parameters of the plane equation in normal form, that is: $A = n_1$, B= n_2, $C = n_3$.

e) We only miss the D parameter, which is the "distance of the plane from the origin". It can be computed by simply computing the dot product between the normal and the centroid:

$$D = -dot(n, centroid)$$

Implement this plane fitting method in the function "fit_plane" of "my_library.py". We provide the script "test_plane_fit.py" for you to check your implementation.

4.1.8 EXERCISE 8

Finally, you implemented all the ingredients required for calibrating and reconstructing the laser in different images. As above-mentioned, we provide the "laser_plane_calibration.py" script to calibrate the laser plane. Open this script and check its expected parameters and what steps are executed for each image. You will notice that all the functions you implemented in the exercises above are used at some point.

Execute the calibration script with the images in "sample_data/calibration/laser".

4.1.9 EXERCISE 9

The only thing that is missing is to reconstruct the laser. You should open the “laser_reconstruction.py” script and check the missing parts. You will see that we left comments on the different steps missing. Most of them can be filled directly with the functions you already implemented in “my_library.py”. Note also that all the functions you implemented are also used in the “laser_plane_calibration.py” script, so you can use this script to inspire you.

Implement the missing parts of the “laser_reconstruction.py” script, and reconstruct the two datasets provided in “sample_data/reconstruction/horse” and “sample_data/reconstruction/greek”. Comment the results for both datasets. Which is the best “--min_red” parameter for each of these datasets? Do you think the results can be improved? How?

5 COLLECTING NEW DATA

You will find a laser scanner in the lab, composed of the ZED camera from the first lab and a laser rigidly attached to it.

You can run the “capture_right_images_zed.py” script (it is the same as the one used in the first lab, but in this case it only captures images from the right camera).

As in the previous lab session, **you must collect new calibration images for the camera, calibration images for the laser, as well as new images for reconstruction**. Then, use this new data to re-execute exercises 1, 8 and 9 (i.e., calibrate the camera, the laser, and try to reconstruct a scene).

6 TO DELIVER

You are expected to deliver a report including the solution to the different exercises and questions listed in the sections above. Include results using both the sample data provided and the data you collected at the lab. You should list all the problems you encountered during the development of this lab session, and especially the differences you found between using our sample data and the data you collected**. In this case, send the “my_library.py” and “laser_reconstruction.py” files alongside your report**.

The report and code are to be delivered one week after the end of the second day in the lab (i.e., the day before starting the next practical session) via Moodle.