E c o l e C e n t r a l e d e N a n t e s
European Master on Advanced Robotics (EMARO M1)
Control and Robotics (CORO M1)

Calibration and development of a multi-robot localization system

Submitted by
Hryshaienko Olena
Ramachandran Ragesh

Supervisors
Garcia Gaëtan
Dominguez Salvador

TABLE OF CONTENTS:

Introduction

0.1. Project definition

Movement and control of mobile robots in closed workspaces depends on global knowledge of the environment and exact knowledge of robot position at all times (localization). Odometry, which is a popular relative localization method, is prone to systemic and non-systemic errors and therefore, results in the inaccurate estimation of the position of the robot. Consequently, localization systems that rely on sensing systems like vision cameras, range are gaining importance in the domain. This project deals with implementing one such system that uses infrared cameras to capture data of the position of several robots in a pre-defined workspace. This is an absolute localization for several robots equipped with IR LEDs, using multiple infrared cameras.

Prior information

The "Wiimote" game controllers (*Figure 0.1)* contains a vision sensor, which can track the positions of several IR sources in real time. The hardware part was developed for a multi-robot localization system based on these sensors.

A prototype of the system can be seen in the project room of building D, which is made of four IR sensors, allowing covering an area of approximately 2x2 meters (*Figure 0.2).*

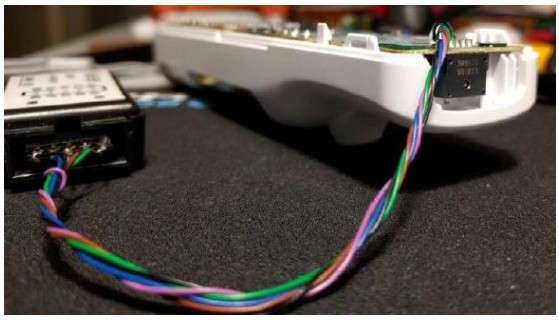

Figure 0.1: "Wiimote", with the camera sensor in its original location, attached to a logical analyzer

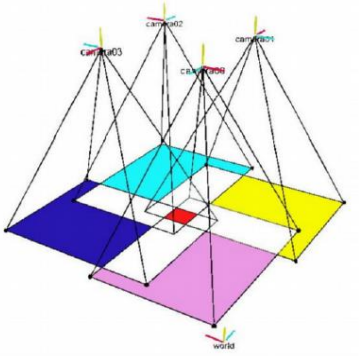

Figure 0.2: The relative position of the camera

Workspace

In the workspace of the Multi-Robot Localization, system has four cameras mounted on the ceiling of the project room and they are placed on the 4 corners of a ($\approx$1x1m^2). The relative positions of the four cameras are shown on *Figure 0.2*. Each camera has a triangular pyramid detection range. These ranges are not independent; they have overlapping space in which the robots (LEDs) can be detected by more than one camera at the same time. The base surface of each camera's detecting range is a rectangle.

The cameras are placed at different yaw angles (rotation around Z-axes) with respect to the world frame. Camera frame of the camera 2 is aligned in the same direction as the world frame. The remaining camera frames are fixed by a rotation of 90 degrees in the counter clock-wise direction progressively. In this way, the detecting range of four cameras forms a near square workspace of the robots, on which the LEDs will be placed.

0.2. Recall

Previous work is done

There were 2 teams, who already worked on given project:

- *The first team* developed the hardware and basic interface to read data from sensors though CAN-USB protocol. A ROS package */pixart_can_reader* was developed to read CAN bus data, convert the hexadecimal frame data from CAN bus into pixel coordinates.
- *Second* - made a calibration and localization part using ROS. A new node called *pixart_localization* has been created to subscribe to *pixart_world* and store data as desired.

First Team

To be more consistent, the first team started with a clean field and had nothing more than the idea of how it should look like. Along with the professor Dominguez Salvador, they have constructed the hardware of the system: took apart a sensor from the game console, established 4 sensors on a ceiling, wired them and connected to the microcontroller, is connected to a single CAN bus to carry all point data.

A single CAN bus is used for data received from the four sensors. Two CAN IDs are assigned to each camera, that is, 8 IDs per microcontroller box. Each CAN frame holds 8 bytes of data, 4 bytes per detected point. Hence, each CAN frame can hold two points (*Figure 0.3):*

```
[ INFO] [1525255426.863506408]: Received frame: 68 # 03 31 02 77 01 b2 02 73
[ INFO] [1525255426.863553038]: Decoded as: Camera 02 from box 00, Point0( 817,  631) and Point1( 434,
 627)
[ INFO] [1525255426.863657899]: Received frame: 69 # 02 ee 02 32 00 ff 03 ff
[ INFO] [1525255426.863705399]: Decoded as: Camera 02 from box 00, Point2( 750,  562) and Point3( 255,
1023)
```

Figure 0.3: Detected point data received by CAN bus

CAN bus is interfaced with a PC using a CAN-USB adaptor (*Figure 0.4)*. This enables live streaming of point data into the user PC terminal. The same PC holds the ROS network where the first team had started developing nodes in order to use data in means of localization.

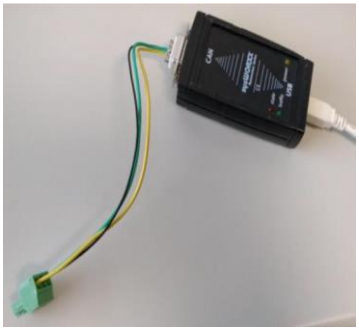

Figure 0.4: CAN to USB adaptor

They created the first ROS node. It publishes the *pixart_raw*, of type *raw_point*, which contains the information about: values of *x* and *y* coordinates, *camera_id* and *point_id*, as well as a time of

detection. All this data is published in a structure defined by *raw_point.msg.*

After, they made one ROS node per camera, which takes parameters defining the *camera_ID*, matrix and pose in the world frame, and subscribes to *pixart_raw*. Publishes: *world_point messages*. This was made in order to receive points projected back from camera frame to the world frame.

Second Team

This team had made a representation of the system in Rviz (*Figure 0.5-0.6)*. Therefore, we could actually see "what camera see": how points are moving with respect to the world frame in different camera frames.

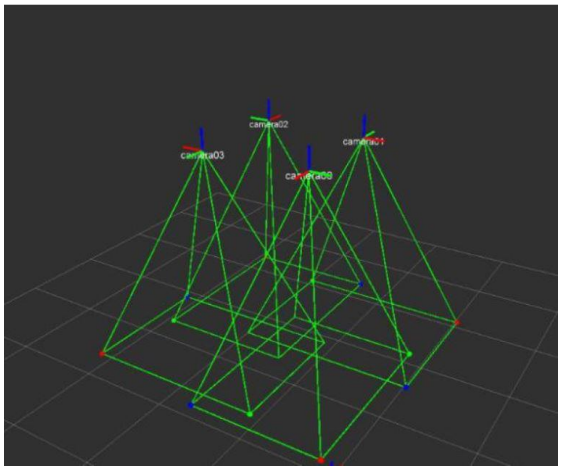

Figure 0.5: Workspace of the application.

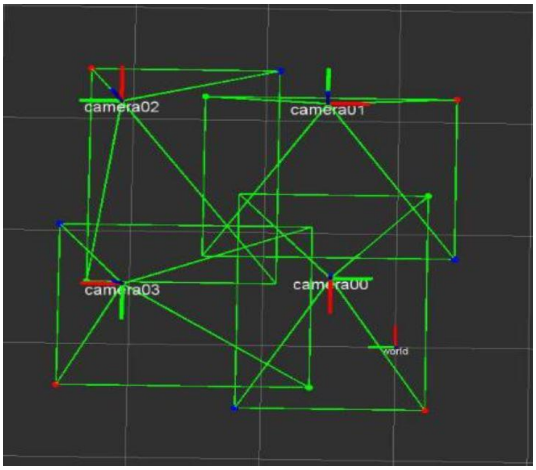

Figure 0.6: Top view of the workspace

This team had conducted a theoretical research and implemented two different Calibration Methods: Homography Matrix Method and Partial Calibration Method. They made a conclusion based on their results that for our system the Partial Calibration Method is given better results (by means of giving a smaller reprojection error).
They have built a localization system, which was good, but not robust enough due to imperfection in the Calibration procedure. In the *Appendix* part of the report, you can find a brief overview of the last year reports.

0.3. Objective

In the frame of the given project, we will need to develop a system, which will localize a robot, which is equipped with an infrared light-emitting diode (IR LEDs) using overhead IR sensors. In order to do so, 4-IR sensors were used which is mounted on a ceiling of the lab D103.

Pitfalls

As we want our localization to to be a robust system, before starting the process of determining the pose of the robot, we will need to calibrate cameras. Without it, the determined coordinates had to be considered with a threshold (*[x, y]* $\pm$ *[threshold_x, threshold_y]*).

Making a robust system

During the calibration of cameras, we will take into account the lens distortion and the fact that cameras are placed not perfectly with respect to each other.

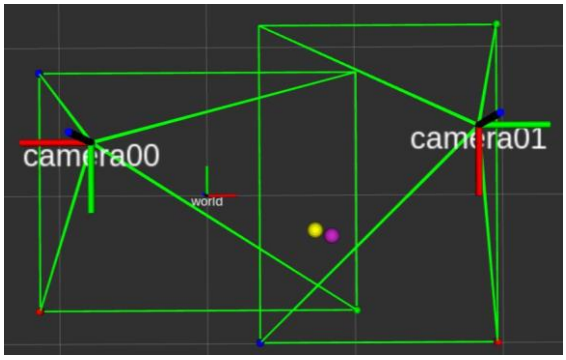

Figure 0.7: The overlapping area between camera_00 and 01 with detecting the same LED

At the end of the calibrating procedure, we want that all of our four cameras will give the same coordinates of the point in the image frames. As now, when testing the system, we can clearly see four different points next to each other (in the overlapping area, where the cameras' ranges are meeting together and sharing the field of view) - *Figure 0.7*.

PART 1 – Before Implementation

1) Localization

Localization is a process of determination of some or all variables of the pose (usually only those necessary to execute a given task). Making it simple, localization is a procedure that will tell us where our robot is at that point in time. In the frame of our project, our objective will be to localize the Turtlebot. For this, we can use as many (well, up to 4) LEDs, as we consider as an officiant amount. Thinking about it we considered that the pattern or 3 LEDs placed in a line in a different distance – will be sufficient.

The idea is following – the middle LED is constantly switched-on – this LED is giving us [x, y] coordinates of the robot. If the robot is moving - it is also possible to compute its θ value (simply using the estimation formula, having the current pose and knowing what the position a moment ago was). However, if the robot is in a hold mode and not moving, the LEDs from aside will be fleshing. As they are placed on the different distances from the centre – it will be more than reachable to establish where robot's front and back sides are and consequently compute θ value.

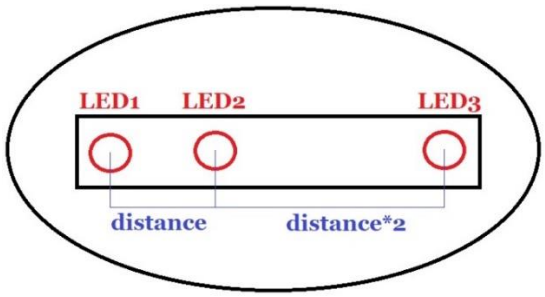

Figure 1.1: The schematic view of a top of a Turtlebot, equipped with 3 LEDs

2) Calibration

2.1. Need for calibration

As the calibration method developed by the previous teams does not take into account the fact that:

- Cameras are not perfectly placed.
- The optical centre of the cameras perfectly in the middle of the image – which is not the case in real world.

Therefore, after developing the localization part, they were obtaining considerably big errors. Hence, we need to develop a new calibration procedure for the cameras that should be able to localize robot accurately. Moreover, make it as flexible to expansion as it can be. for the reason that if the idea of localizing the robot by means of IR sensors will show itself good – a stand will grow from 4 to 20 cameras.

2.2. The theory behind the calibration process

Camera calibration is about finding the internal and external parameters of the camera and use these parameters to correct the lens distortion, measure the size of an object in world units, or determine the location of the camera in the scene.

Here are some of the factors that will be taken care of, indicating whether this parameter is considered as intrinsic or extrinsic one:

Intrinsic *([K] – matrix)*	
Image centre:	We need to find the position of the image centre in the image. *Wait a minute, isn't the image centre located at (width/2, height/2)?* Well, not really! Unless we calibrate the camera, the image will usually appear to be off-centre.
Focal length:	Remember, how people using DSLR cameras tend to "focus" on things before capturing the image? This parameter is directly related to the "focus" of the camera and it is very critical.
Skew factor:	This refers to shearing. The image will look like a parallelogram otherwise!
Lens distortion:	This refers to the pseudo-zoom effect that we see near the centre of any image.
Scaling factors:	The scaling factors for row pixels and column pixels might be different. If we do not take care of this thing, the image will look stretched (either horizontally or vertically).

Extrinsic *([R T] – matrix)*
Rotation and **Translation** of the calibration board with respect to the camera frame

Camera calibration matrix transforms a 3D point in the real world to a 2D point on the image plane (*Figure 2.1*), considering all the things like focal length of the camera, distortion, resolution, shifting of origin, etc. This matrix consists of parameters that are intrinsic as well as extrinsic to the camera and can be seen on *Figure 2.2*.

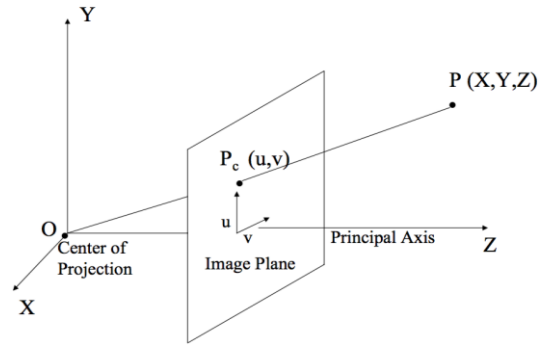

Figure 2.1: Image formation parameterized scheme

$$z_c \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} R & T \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

[2D] = Intrinsic [Extrinsic] [3D]

Figure 2.2: Equation of transition from world frame (3D) to image frame (2D)

In addition, it means that if we will find those matrices, that contains the intrinsic parameters, as well the transformation vectors (ξ-ksi) between camera_00 and all other cameras in a system (*Figure 2.3*) - we can do a reverse procedure and retrieve a 3D pose, knowing the corresponding point in the image frame. It is interesting that in this case, we don't need to know all 12 vectors, 3 is more than enough.

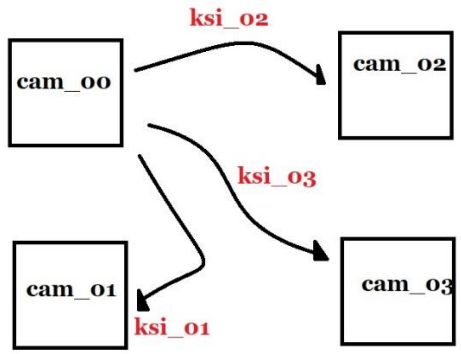

Figure 2.3: Schematic representation of a related transformation vectors, needed for a pose estimation

For example:
If the point was detected in between camera_02 and camer_03 we need to know the relations between them in order to obtain the right value. In this case, we will us a ksi-vector between camera_00 and camera_02, and inverse ksi-vector between camera_00 and camera_01. Multiplying them, we will get a relation between camera_02 and camera_03.

2.3. Calibration Algorithm for our system

Definition of a camera calibrating procedure

Calibration is defined as an optimization problem, the aim of which is to reduce the distance between desired pixel coordinates of the interesting point and the obtained coordinates.

Number of measurements to be made

By saying "*measurement*" we mean: how many valid data sets do we need to have in our data file. Data set can be named valid if at least one camera will record all 4 points from the calibrating board.
We can write an equation to express the number of unknown and known parameters.

Prior knowledge

From each data set, we will receive 8 values (4 points with both *x* and *y* coordinate).

Unknown variables

From each measurement, the *extrinsic parameters* have to be recalculated, as the relative position of the board with respect to the camera frame will change. Nevertheless, the *intrinsic parameters* are something that will not change with measurements. Once defined– they will be settled.

Equation will be then as follows:

$6N + 6 \leq 8N \rightarrow N \geq 3$, where N is an unknown number of measurements.

It means that for each camera we need to have at least three valid data sets, for the algorithm to work. However, to make it robust this number should be at least 3 times greater. Which means, that we will make around 10-15 measurements for each camera.

Algorithm for calibrating process

The method is similar to the stereo calibration using a chessboard pattern and Open Source Computer Vision (Open CV) Library.

1. Declare all necessary **vectors** to stor**e** the image points and the object points.
2. Read the input file, where the alleged **transformation vectors** between camera_00 and all other cameras are defined as well as the **initial intrinsic parameters**.

    - To find the right transformation vectors we had firstly to measure a distance than to define a rotation around *x*, *y* and *z* coordinate.
      ξ = [transl_x, transl_y, transl_z, roll, pitch, yaw]

    - This is a so-called initial guess. We do not put the exact value (for instance, the rotation angle around z can be 92.54^{o}), but this value is important, as the wrong definition of it will lead in finding the local minimum of the optimization function, instead of searching – global.

3. To use a function, that estimates the **object pose** given a set of object points, their corresponding image projections, as well as the camera matrix and the distortion coefficients.

4. Run the global Levenberg-Marquardt **optimization algorithm** to minimize the reprojection error, that is, the total sum of squared distances between the observed feature points imagePoints and the projected (using the current estimates for camera parameters and the poses) object points objectPoints.

The function computes Jacobians - matrices of partial derivatives of image points coordinates (as functions of all the input parameters) with respect to the particular parameters, intrinsic and/or extrinsic. The Jacobians are used during the global optimization

3) Calibration board selection

Why can’t we use chessboards?

Because our system will simply not see it. In order to make the calibration of a system, we have constructed a new calibration board that was mounted on a top frame of a Turtlebot. It was a good move, as this localization system was targeting on Turtlebots – the heights from the ground was considered to be as 0.45 m. Therefore, we will calibrate a system for this type of robot, using this robot.

Alternate calibration board

In order to retrieve points from the chessboards we need to run a function, which reads an image, process it to detect edges and then corners, after which it is giving us the found coordinates. In our case, there is no need in this step, as the only thing we are going to receive is the coordinate of a LED placed on a corner which reduces the computation cost.

Storage of datasets

In our case, we will need to read and store the data received from cameras separately, as then use it as an input to a calibration program. We have decided, to have a final storage in a dictionary-structured .json file.

JSON - JavaScript Object Notation is a way to store information in an organized, easy-to-access manner. In a nutshell, it gives us a human-readable collection of data that we can access in a really logical manner. With it, data can be load into file quickly and asynchronously.

4) Strategies

4.1. First strategy

After reading the previous reports and all necessary documentation about calibration, we have contacted Mr Bogdan Khomutenko, as he is one of the specialist in ECN for Camera Calibration. Following his advises, we had formulated the next steps to be done:

1) Modify the already existing board, so it would have constantly switched-on pins in the pattern, which you can see in *Figure 4.1*.
2) Create a file which will make it easier to communicate with the CAN-USB reader.
3) Create a ROS node to collect the data required for calibration.
4) Make a program to store needed data points in pre-defined structure, using JavaScript Object Notation (JSON);
5) Use received data to calibrate cameras and compute the errors.

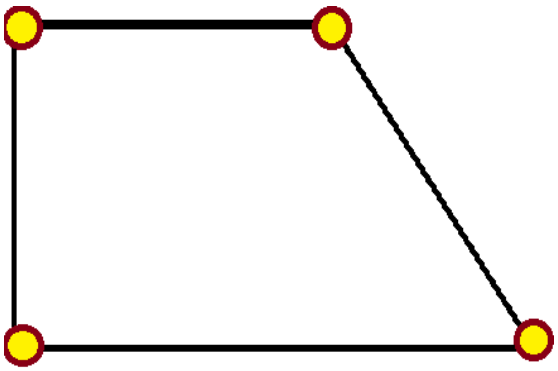

Figure 4.1: Pattern for the future calibration board v2.0

The idea was trivial enough in implementation: having a new calibration board with known parameters and a program, which stores points in a form presented on *Figure 4.2*, we could follow the same procedure as with normal cameras and a chessboard calibration board.

Serial Number	Cam00				Cam01				Cam02				Cam03			
1	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4
2	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4
3	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4
4	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4
5	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4	x1, y1	x2, y2	x3, y3	x4, y4

Figure 4.2: Structure of a form needed for storing 4 points from each camera

Finding the camera parameters include intrinsic, extrinsic, and distortion coefficient. To estimate the camera parameters, we need to have 3-D world points and their corresponding 2-D image points. You can get these correspondences using multiple images of a known calibration pattern, such as a board with known distances between each LED.

4.2. Problems we faced

We started our work by marking a floor, so it would be easier to test a floor knowing when the point is out of a camera's view range. Doing that we have noticed that camer_03 is not giving any values along OY axes. We went through all existing code and came to conclusion, that the problem must be in a hardware of the sensor as such, and the problem is not in CAN reader or wires. In a few weeks after this unfortunate lost we, we have camera_02 went out of order. It has just shown any values, giving no signal to us.

After changing the board and making a new program for storing the needed data we have found a glitch, which disabled our entire plan in making a calibration with a method defined above. The problem was that after bringing the board under the cameras and making translation-rotation of it: all points shifted chaotically in a camera frames. It means, that they were "jumping" all around (*Figure 4.3*) for no obvious reason for that. We believe that the problem is laying in a hardware of the sensors themselves. We have tested the behaviour of the system on a board with 3 and 2 points but the problem didn't disappear.

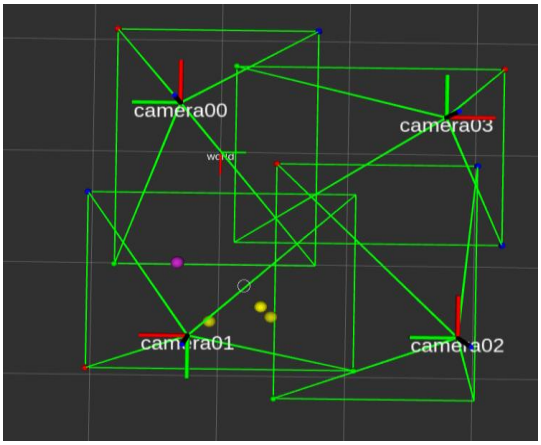

Figure 4.3: Jumping of LED positions

4.3. Second and final strategy

We had decided to mount 4 LEDs on a back of the Turtlebot, those LEDs will be switched ON/OFF in a given sequence: that's how we could still leave a possibility to match a 3D point on a board and a 2D point on the image and have only one point turned on at a time. Otherwise, working with one single LED would lead to the errors greater than the one presented by a previous team. The new outlook of the board can be seen in *Figure 4.4*.

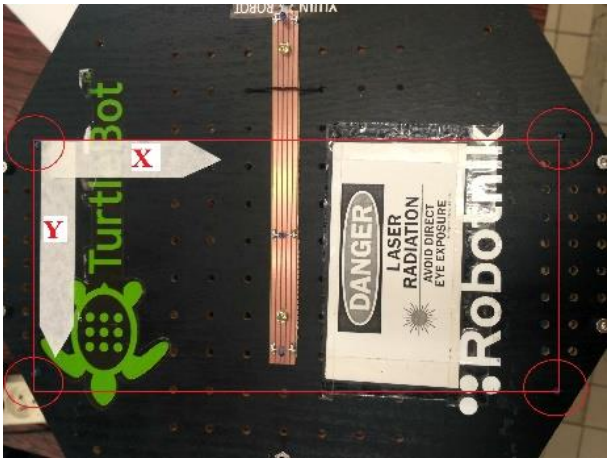

Figure 4.4: final calibration board imbedded on the planar back of the Turtlebot

PART 2 – Work Done

5) Calibration board

5.1. Description of the Board

Geometry

LEDs are placed in the corners of a perfect rectangle, with a longer and shorter side to be equal to 0.262 and 0.12 respectively (dimensions are given in meters).

IR LED

Note: Small view angle of the LED made the tilting of calibration board more difficult but since the proper LEDs were not available to us, we proceeded to calibrate our system using these:

Lens type	Blue transparent
Spectral wavelength	940nm
Spectral bandwidth	50nm
Radiant intensity	8mW/sr
Forward current	50mA
Forward voltage typ.	1.2V
Forward voltage max.	1.6V
Power dissipation	80mW
View angle	34°

Figure 5.1: photo of the described LED mounted on a battery, in order to test a work of a system

Power Supply

The LEDs and Arduino are powered using the onboard 12 Volts, 1.5-Amps power supply of turtle bot.

The frequency of LEDs' blinking

We have implemented a small program on Arduino microcontroller, which will sequentially switch ON-OFF LEDs with a cycle time = 100 milliseconds, on the press of toggle switch.

5.2. Embedding

Our calibration board was mounted on a top frame of a Turtlebot (*Figure 5.2*) in order to automate the process of calibration. It is a tedious task to walk around the arena carrying the calibration board to collect datasets so we came up with an idea of placing the calibration board on top of Turtlebot and making it go around collecting data. This will be helpful when the camera array expands to a larger size in the future.

In addition, this localization system was targeting on Turtlebots whose heights from the ground were considered to be as 0.45 m. Therefore, we will calibrate a system for this type of robot using this robot.

Figure 5.2:The image of the "Kabuki" Turtlebot

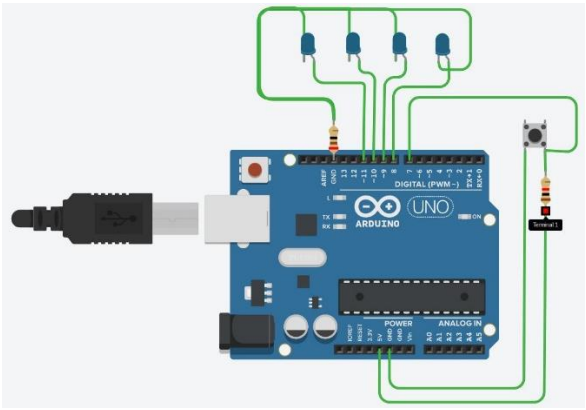

Figure 5.3: Circuit diagram

5.3 Electronics

Board was made using the Arduino microprocessor. LEDs were connected in the sequence depicted on Figure 5.3. There will be 4 LEDs – 4 corners. Each of the LED will be turned ON by a switcher for 200 milliseconds and then OFF for the same amount of time.

We have tested a system when the ON time was 200 milliseconds and OFF time was 100, but we have received a few a data points from sensors as if they were flashed at the same time. That is why we have decided to increase the time twice.

```
while (pressed)
  {
    delay(200);
    for (int i = 8; i < 12; i++)
    {
      digitalWrite(i, HIGH);
      delay(100);
      digitalWrite(i, LOW);
      delay(100);
    }
    digitalWrite(13, HIGH);
    pressed = LOW;
  }
```

Code 5.1: A snippet of code from *calibration.ino* file

6) Configuration File

6.1. Old method to run the process

Last year, there were defined the steps to be followed in order to correctly connect the localization system to the user PC, which was:

1. This package is found to work seamlessly on Ubuntu equipped with ROS Indigo.
2. Download and install can-utils software in Ubuntu environment.
3. Plug in the power supply cable into mains (large white adaptor).
4. Connect the CAN-USB adaptor into one of the USB ports of the user PC.
5. Enter can-utils folder and type: *./configure*
6. Type the following command to check the sl-device that is assigned to the user PC: *ls /dev/ttyACM**. This should return one of the values: *ttyACM0* or *ttyACM1*.
7. Depending on the index in the previous step, type this command: *./slcan0_up.sh* or *./slcan1_up.sh.* This should give a message that the *slcan0* or *1* is attached.
8. One can now stream the CAN bus data into the terminal by typing: *candump slcan0* (or *slcan1* as applicable).
9. In a separate terminal, enter the *src* folder of the catkin workspace where the package is to be unzipped: *pixart*
10. This package has its own launch folder and corresponding launch file that can be launched.

We found repeating these actions everyday time-consuming. Consequently, with a great help of Dominguez Salvador and professor Gaetan Garcia, we have created a file *init.sh* in order to make the sl-device that is assigned to the user PC, have one and only "0" index.

6.2. A new method to run the process

It is accessing the can-utils folder and with all necessary permission changes, it uses a file which blocks the receiving of a "1" index, always enabling the "0" one.
Then, it is going into the */catkin_ws* directory. Where it:

- Compiles all existing packages;
- Adding environment variables to your path to allow ros to function;
- Launches a *pixart* package.

Below, in a *Code 6.1*, you can find a full script within the *init.sh* file:

```
0.  #! /bin/bash
1.  # this file we use to read data from CAN-USB reader
2.  chmod a+x init.sh
3.  chmod a+x slcan0_up.sh
4.  chmod a+x slcand
5.  chmod +x ./slcan_attach
6.  cd /home/ragesh/catkin_ws/src/pixart/src/can-utils
7.  bash -c './configure'
8.  ls /dev/ttyACM0
9.  bash -c 'sh ./slcan0_up.sh'
```

```
# this file we use to build and set the environment
cd ..
cd ..
cd ..
cd ..
bash -c 'catkin_make'
bash -c 'source devel/setup.bash'
bash -c 'roslaunch pixart pixart.launch'
```

Code 6.1: code of *init.sh* used to read a data from CAN-USB and setting the environment

6.3. Steps to be followed

1. This package is found to work seamlessly on Ubuntu equipped with ROS Kinetic.
2. Download and install can-utils software in Ubuntu environment.
3. Plug in the power supply cable into mains (large white adaptor).
4. Connect the CAN-USB adaptor into one of the USB ports of the user PC.
5. Unzip an archive with all provided code into the desired folder
6. Before the first run: in the root of *catkin_ws* folder there will be a file called *init.sh*. Open it and modify a path to the to the *./configure* file:
   - line 8: in between cd ---------- /catkin_ws/.. .
7. Type in a command line *./init.sh* to execute a file (you do it after you are already in *catkin_ws* folder)

6.4. Conclusion

There is always a way to make your life easier and not to do all over again the same actions which can be done automatically.

7) Data Storage

7.1. Preconditions

After running the program, it can be seen that the output is being printed in a line and not very easy in being read. We decided to make a separate program, which will extract the data in a list form defined by a *raw_point.msg*. Consequently, import it into a given file.

Python vs C++

There are the numpy and matplotlib libraries that give a solid foundation for all numerical, statistical, matrix computations and graphical representations, similar to what Matlab offer. There is just NO comparable library of this kind and scale in C/C++. Since we were working more on datasets and storage, we decided to proceed with python which made the programming faster, easier and more maintainable**.**

JSON

JSON - JavaScript Object Notation is a way to store information in an organized, easy-to-access manner. In a nutshell, it gives us a human-readable collection of data that we can access in a really logical manner. With it, data can be load into file quickly and asynchronously.

7.2. Development of ROS node

A new ROS node *Calibration_data* was developed using Python, which stores the incoming raw data into *JSON* file. This node subscribes from *pixart_can_reader* that published the raw data as *pixart_raw* from the IR cameras through the USB-CAN controller. This stored data is later used for data processing for the calibration of the IR cameras.

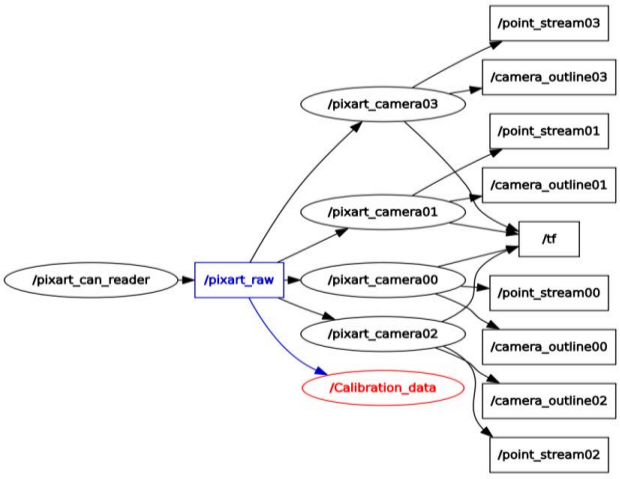

Figure 7.1: ROS node for data storage

```
[
   [
      [
         'time_ms',
         'time_ns'
      ],
      [
         'y',
         'x'
      ]
   ]
]
```

Code 7.1: Structure of raw data stored in the *json*

7.3. The algorithm for data processing

1. **Single point data structure.**
2. **Storing the raw data in separate files for each camera.**
3. **Filter files by taking the average value for all points.**
4. **Grouping of filtered points into a single file based on the *time stamp* and *camera_id***

1. The defined **structure for a single point** detected in a camera frame is as follows:

```
{ 'x':data.x, 'y':data.y, 'cameraID':data.camera_id,
'time ms':data.stamp.secs, 'time ns':data.stamp.nsecs}
```

Code 7.2: A snippet of code from *raw_data_store.py* to define a structure inherited by receiving a *raw_point* subscribed message

Essentially, we just built a structure for carrying an information, received from *raw_data* into a JSON file format.

2. In order to begin the procedure of camera calibration, as such, we need to distinguish points that are seen in **different camera frames** – save the data into 4 separate files, filtering by averaging the datasets to remove the error which may occur when capturing the calibration board.
3. We need to make a test to know which dataset belong to which point, and which one is the completely different point (as for one blinking LED we will not receive one and only measurement in a camera, it's going to be a continuing flow of points).

   To do so, we are checking if next point is close enough for being considered as the **same point** as the previous or not (within a threshold to take into account oscillation of a calibration board). In our case the flashing of LED takes 800 milliseconds so we took a tolerance of 2 seconds. If it is being the same, we add it is parameters to the instance called "*accumulator*" and incrementing the "*counter*". After a data set for the point was checked, we will divide *accumulator* with the *counter* value, which gives the average for each point.

```
1. n = len(data_points)
2. for i in range(1,n):
3.       cur = itemgetter('x', 'y')(data_points[i])
4.       cur_time = itemgetter('time ms','time ns')(data_points[i])
5.       diff = np.abs(tuple(map(sub, prev, cur)))
6.       res = np.any(diff <= tol_pix)
7.       if (res):
8.            acc = tuple(map(add, cur, acc))
9.            count = count + 1.
10.           prev = cur
```

```
    else:
        filtered_time.append(save_time)
        save_data = [x / count for x in acc]
        save_time = cur_time
        prev = cur
        acc = cur
        count = 1
        filtered_data.append(save_data)
```

Code 7.3: a snippet of code from *raw_data_store.py* to filter an average the value of the same coordinates

4. After distinguishing our points in each file, we are creating a new file; where we will group those data into different data sets in depend on camera that received the points.

   In addition, we have to make a function, which shows how many cameras have seen **points at the same time** (we consider only those sets where one camera see all 4 points, leaving behind those, which have a lack of it) – it means distinguish data from overlapping zone.

```
for i in data:
        pt = [i[2], i[3]]
        cur_time = i[0]           #sec
        time_diff = np.abs(cur_time - prev_time)
        if count == 4:
            time_points = [cur_time, temp]
            points.append(time_points)
            count = 0
            temp = []

            if(time_diff < tol):
                count = count + 1

            else:
                temp = []
                count = 1
         temp.append(pt)
         prev_time = cur_time
       return points
```

Code 7.4: The snippet of code from *merged_data.py* to save a data only from those measurements, which have all 4 points

After grouping and merging the data in a predefined pattern, we have created a dictionary in order to be able to use this file as an input to the calibration function. In case, if we will get a measurement from an overlapping between to cameras.

```
[
 [{"Cam_id": "cam0",
  "points":[
          [ x1,y1],
          [ x2,y2],
          [ x3,y3],
          [ x4,y4] ]
 }
 ],
 [
 {"Cam_id": "cam1",
  "points":[
          [ x1,y1],
          [ x2,y2],
          [ x3,y3],
          [ x4,y4] ]
 }
 ]
]
```

Dictionary 7.1: The structure defined for importing data into *Merged_file.json*, where *x* and *y* are numerical value for the corresponding point coordinates

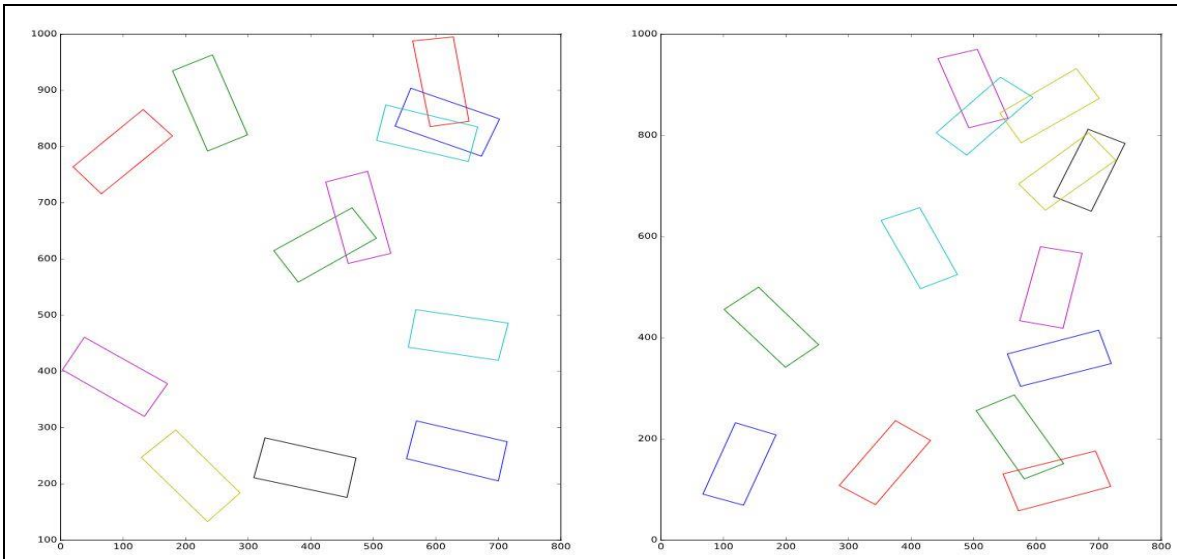

Figure 7.2: plots of the measurements made in camera_00 and camera_01 frames

In the same file *Merged_file.json*, we have printed the collected data, which can be seen in *Figure 7.1*, where each rectangle represents one measurement of a calibration board.

NOTE: in our program, we merged only data from camera_00 and camera_01 as during our work cameras_02 & 03 went out of order, which made it impossible to collect needed data.

Conclusion

For the tasks defined in this chapter, we have created next scripts:

raw_data_store.py	program wrote in Python, created in order to subscribe for /*pixart_raw* (published by *pixart_can_reader* topic), to average values received for a single point and to store those values in a defined structure in 4 separate .json files (each for every camera);
merge_data.py	program, for combining those 4 files, created after using the previous script, in order to give it the right dictionary-structure and consider only those data-sets, which has all 4 points from the calibration board.

8) Calibration procedure

In order to implement this procedure, the software developed by Bogdan Khomutenko was modified and embedded into the main system. The most difficult part was to adapt our data, so this software will accept our system.

In the case of stereo calibration, we will use two datasets – it combines a transformation between camera and the board dataset and one stereo calibration dataset.

8.1. Declaration of Transformations

In this case, we have two transformations: in between camera and board as well as between available cameras.

```
"transformations" : [
    {
        "name" : "xiCamBoard",
        "global" : false,
        "constant" : false,
        "prior" : false
    },
    {
        "name" : "xi12",
        "global" : true,
        "constant" : false,
        "prior" : true,
        "value" : [1, 0, 0, 0, 0, 1.57]
    } ]
```

Code 8.1: a snippet of data from *calib_ir.json*, used as an input file to the calibration program

- **xiCamBoard** is used to calibrate a single camera with respect to the board.
- **xi12** is the same for all the stereo images and defines the transformation between the two cameras. It is global and has a prior.

8.2. Transformation Prior Format

The following formats are supported to define a transformation:

• 3 values [x, y, theta]	2D posture. z-coordinate, x- and y-rotation are 0
• 6 values [x, y, z, rx, ry, rz]	full 3D parametrized by translation and rotation vectors (see Rodriguez rotation formula)
• 7 values [x, y, z, qx, qy, qz, qw]	3D parametrized by a translation vector and a normalized quaternion
• 12 values [r11, r12, r13, t1, r21, r22, r23, t2, r31, r32, r33, t3]	homogeneous transformation matrix, stored row-wise: [r11 r12 r13 t1] [r21 r22 r23 t2] [r31 r32 r33 t3] [0 0 0 1]

8.3. Declaration of Cameras

In the case of stereo calibration, we'll have two cameras.

```
"cameras": [
    {
       "name" : "cam0",
       "type" : "eucm",
       "constant" : false,
       "value" : [0.01, 1, 1175, 1175, 512, 384]
    },
    {
       "name" : "cam1",
       "type" : "eucm",
       "constant" : false,
       "value" : [0.01, 1, 1175, 1175, 512, 384]
    }]
```

Code 8.2: a snippet of data from *calib_ir.json*, used as an input file to the calibration program

8.4. Data Definition

```
"transform_chain" : [
            {"name" : "xiCamBoard", "direct" : true}
```

Code 8.3: a snippet of data from *calib_ir.json*, used as an input file to the calibration program

- *transform_chain* - describes the sequence of transformations between the camera and the calibration board.

The transformations are combined in the same order before being applied to the points of the calibration board.

```
 "points" : [[0, 0.12, 0], [0, 0, 0], [0.262, 0, 0], [0.262, 0.12, 0]],
 "_points" : [[0, 0, 0], [0.262, 0, 0], [0.262, 0.12, 0], [0, 0.12, 0]]
```

Code 8.3: a snippet of data from *calib_ir.json*, used as an input file to the calibration program, where "points" represents the *x* coordinate and "_points" - *y*

We make a double set of values inside of "data" structure, as both camera_00 and camera_01 will see the same calibration board and are of the same type.

Results

We tested the calibration software with a set of 12 data sets for *camera_00* and *camera_01*. The stereo calibration was done using a set of 3 data sets. We obtained the following results for the datasets we used.

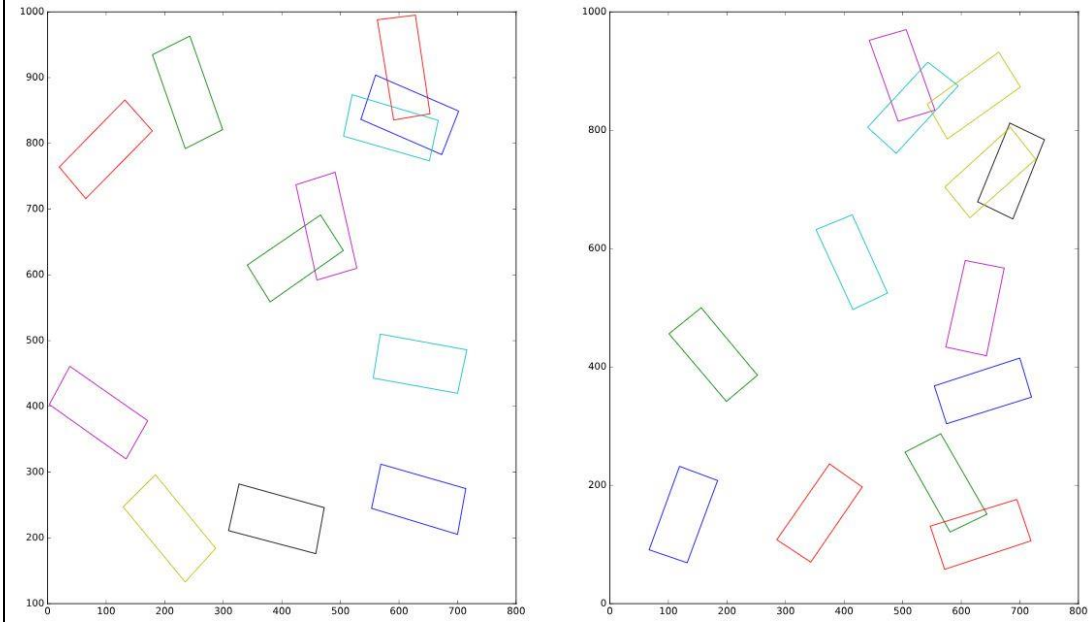

Figure 9.1: Calibration board detected by camera_00 and 01 for calibration

Found intrinsic parameters are:

```
cam0 : 0  0.36035  1203.23  1204.24  526.616  389.011
cam1 : 0.108349  1.60597  1220.57  1216.91  522.642  393.629
```

Changing the files, defined inside of the *calib* folder, we have inserted new values instead of the one used by the previous team. As we can see on the Rviz, now points almost completely coincide, especially if to compare with the plots presented in *Figure 0.7*.

Future groups will use global extrinsic parameters in order to make a localization system more robust:

```
xi12 :    0.997239 -0.00124434  -0.0429873 0.0130096 0.0136953   1.50725
```

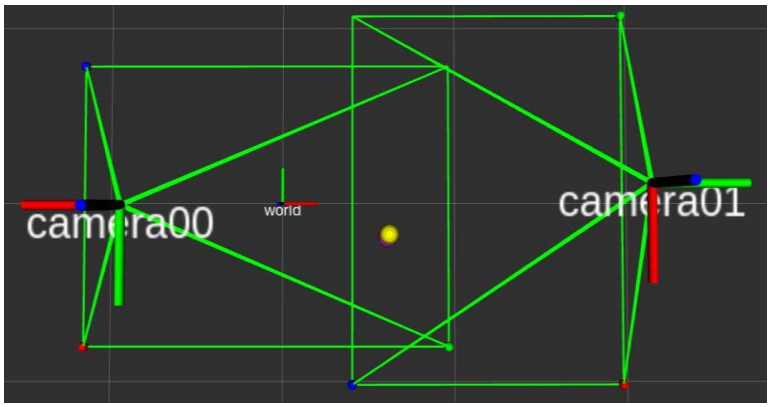

Figure 9.2: The overlapping area between camera_00 and 01 with detecting the same LED with the new intrinsic parameters

Conclusion

The idea of using IR-sensors to localize a Robot is very good: we do not need to waste operational time and use a big data server to store images (as in the case with a localization using usual cameras). It could work if the hardware was developed specifically for these purposes.

Instead, working on equipment developed as a game console – we found it very challenging and not sufficient enough in case of multiple robot localization in a one camera frame. As it was explained in an introduction part, it is possible for one camera to see multiple points (up to 4), but it cannot be sufficiently localized because of its unpredictable behaviour.

10.1. Future work

In our opinion, there are too many issues with a hardware to rely on the results obtained during the data processing. A robust hardware system is necessary for the better performance of the localization system. Hence, there is no point to continue developing software without solving the main obstacles in the hardware:

1) To find a reason for not receiving data from the camera02 and camera03.
2) To solve an issue with “jumping” points in case of their multiple appearances in any of camera frames.
3) Expand the camera array to more cameras for large area coverage.

10.2. Technical difficulties and Solutions

1. Unavailability of required IR LEDs for making calibration board delayed the calibration process. However, a new calibration board was made using the LEDs from the previous calibration board.
2. False detection of points when the workspace is illuminated by direct sunlight which can be avoided by using blinds or curtains for the windows in the project room.
3. Malfunctioning of cameras as we approached the deadline. We had decided to proceed with just two cameras at the end for calibration but the programs we developed can process the data for 4 cameras.
4. Calibration board was bulky and also requires an adaptor for power supply from mains which made the calibration procedure tedious. A new calibration board was made on the top of turtle bot which helped in the automation of calibration procedure.

Appendix

In this section, we will do a brief overview of the work done by other groups, mainly the information, that we had found useful during our work. In order to do so, we will have to omit a lot of details, so it's highly recommended for the future teams to go through all the reports before continuing working on a project.

11.1. Overview of work done by Henri Chain and Ghislain Rabin in March 2016

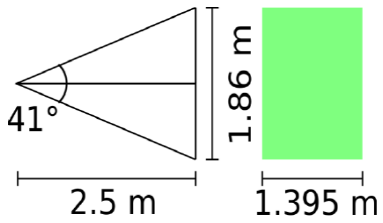

Figure 11.1: Sensor FOV and ground viewable area.

	Bit							
Byte	7	6	5	4	3	2	1	0
0	X1<7:0>							
1	Y1<7:0>							
2	Y1<9:8>		X1<9:8>		Y2<9:8>		X2<9:8>	
3	X2<7:0>							
4	Y2<7:0>							

Figure 11.2: data arrangement in basic mode.

uint16 x
uint16 y
uint16 camera_id
uint8 point_id
time stamp

Figure 11.3: Data defined in the *raw_point.msg.*

One sensor provides:

- The location of *four* dots;
- *Resolution:* 10 bit (*1024x768*);
- *Sample rate:* ≈*270 Hz*;
- *Field of view:* 41°.

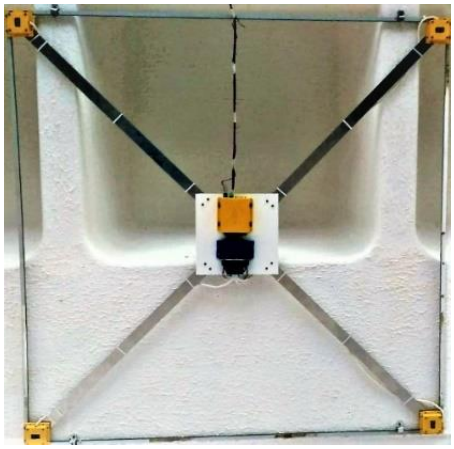

Figure 11.4: 4 cameras and electronic interface mounted on the ceiling of the project room

Communication protocol:

- *CAN* - between cameras;
- *USB* – between computer and cameras.
  ➔ CAN-USB adaptor

The cameras have three operating modes:

- *basic* (location of four points);
- *extended* (four points with dot sizes);
- *complete* (four points with bounding boxes and intensity).

For the purpose of localization and calibration, we work with the *basic* mode - provides location data up to four points (*Figure 11.2)* ➔ this application works under the constraint that each camera can detect only up to four under it.

The initial ROS node reads the CAN bus using the Linux Socket CAN API. It publishes data defined in a *raw_point.msg* and can be seen in *Figure 11.3*:

- ***camera_id*** is a unique identifier for each camera, combining the box ID and camera ID within the box;
- ***point_id*** is the ID of the point within the camera (0, 1, 2, 3).

11.2. Overview on a report made by Haorui Peng, Leonardo Stretti and Anusha Srihari Arva in June 2016

11.2.1 Determination of Viewing Range and Overlap

Camera ranges:

- Have a triangular pyramid detection;
- Are not independent - have overlapping space in which the robots (LEDs) can be detected by more than one camera at the same time.

Camera frames:

- The camera_02 is aligned in the same direction as the world frame;
- The remaining camera frames are fixed by a rotation of 90 degrees in the counter clock-wise direction progressively.
  → The detecting range of four cameras forms a near square workspace of the robots, on which the LEDs will be placed.

! IMPORTANT: The size of the detecting spaces is different at different heights**.** The higher the robot, the smaller is the detecting space. If the robot is too high, the workspace might not be a square any more. The detecting range of each camera will not be overlap, which will result in some spaces between the cameras where the LEDs cannot be detected.

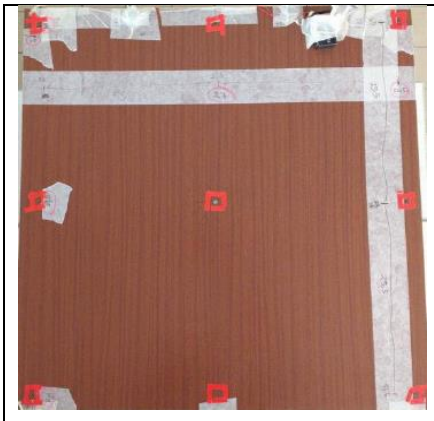

Figure 11.5: Calibration board

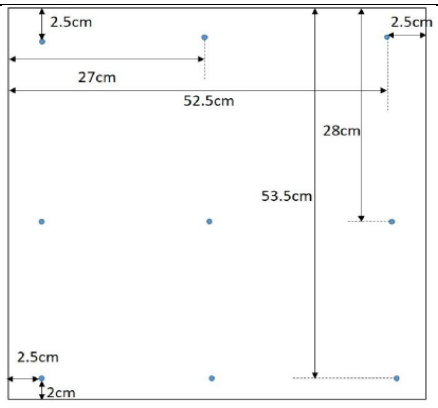

Figure 11.6: Dimensions of the calibration board

11.2.2. Determination of Nature of Calibration

Special "chess-board" like the pattern of LEDs was constructed shown in *Figure 11.5*. Its pattern contains nine LEDs on the corner of 2*2 squares with an edge length of 25cm. To be extremely accurate with measurements, the exact positions of the LEDs were calculated as shown in *Figure 11.6*.

11.2.3. Homography Matrix Method

$$s\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} \sim H\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Figure 11.7: Equation for Homography method.

In the equation presented in *Figure 11.7*:

- ***s*** is the homogenous factor of the pixel coordinates
- **H** is the homography matrix, in other words, the transformation matrix from the world plane to the pixel plane.

The output from the method*:* the errors are huge with this method, especially when the camera frame and the world frame are not in the same orientation.
The main reason for the huge error is the measurements of the world coordinates. Even though we used a board to define the world coordinates, the board is placed 45cm high upon the ground, there is no direct reference for the measurements of the world coordinates, and it will bring large error when we define the world coordinates.

11.2.4. Partial Calibration Method

$$K\begin{bmatrix} I_3 & 0_3 \end{bmatrix}\begin{bmatrix} R_{3\times3} & t_{3\times1} \end{bmatrix}\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = P\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Figure 11.8: Equation for Partial Calibration method.

In the equation presented on Figure 11.5, matrix ***K*** is the intrinsic matrix containing 5 parameters:

- Focal lengths - $\mathbf{f_u}$, $\mathbf{f_v}$;
- Center of image plane - $\mathbf{u_0}$,$\mathbf{v_0}$;
- Axis skew $\mathbf{f_{uv}}$ of the camera.

Matrix ***R*** and vector ***t***: form the extrinsic matrix of the camera depicting rotation and translation of camera with respect to the world frame.

It can be then assumed that the cameras are ideal and placed well - thus fixing the centre of the image frame at (512, 384).

→Relatively small errors compared with that of holography method.

11.2.5. Study of the Existing Software Package

For the current arrangement, Camera IDs range from 0 to 3 and are expected to go up to 15 when the whole array of cameras is assembled. Point_IDs range from 0 to 3 for each camera.

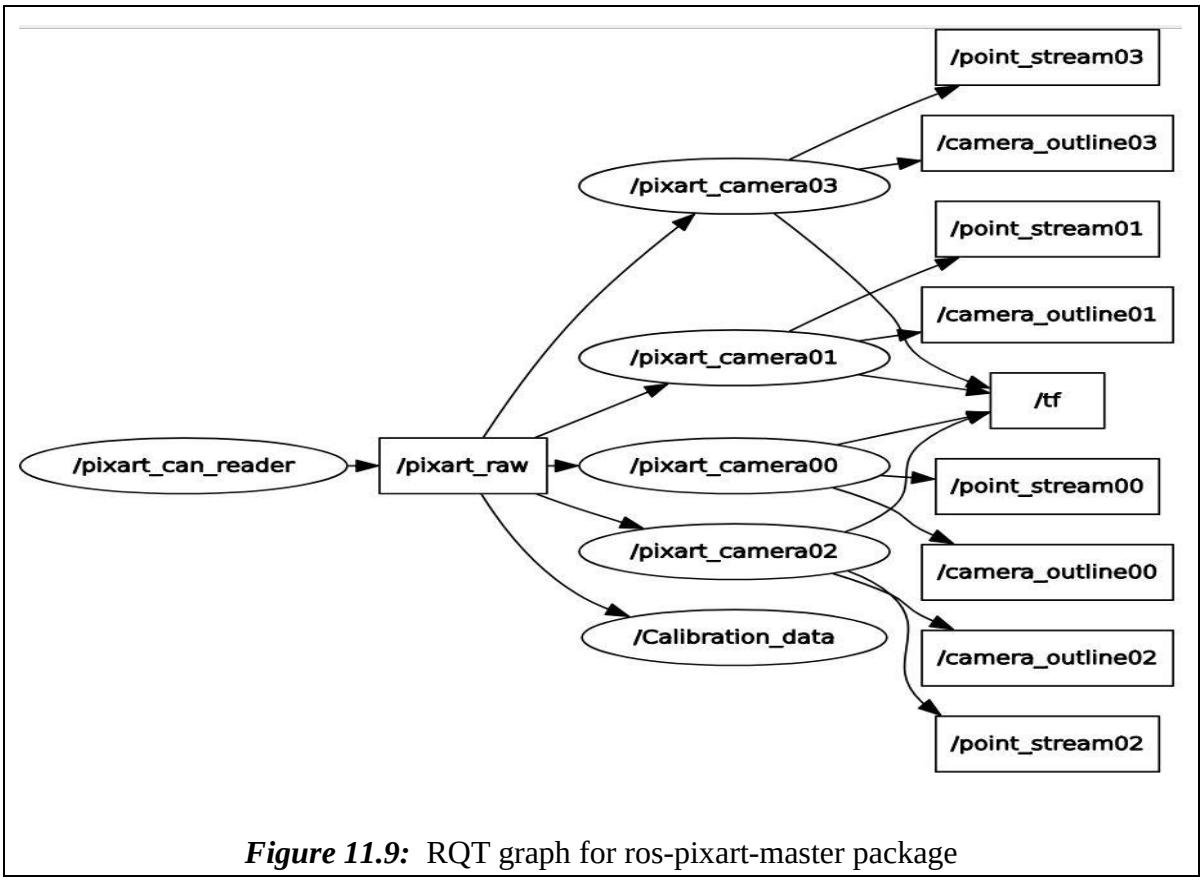

Figure 11.9: RQT graph for ros-pixart-master package

a. /pixart_can_reader

Reading the frame data coming from CAN bus which a hexadecimal value which contains:

- Camera ID
- Box ID
- Coordinates of the point in camera frame.

One message is published per detected point.
Publishes to: /pixart_raw

```
geometry_msgs/PointStamped pt
std_msgs/Header header
uint32 seq
time stamp
string frame_id
geometry_msgs/Point position
float64 x
float64 y
float64 z
uint16 camera_id
uint8 point_id
```

Figure 11.10: Message fields of */world_point*

```
std_msgs/Header header
 unit32 seq
 time stamp
 string framd_id
 geometry_msgs/Point position
  float64 x
  float64 y
  float64 z
```

Figure 11.11: Message fields of */point_stream0x*

b. /pixart_camera00 - /pixart_camera03

Reading pixel coordinates published in */pixart_raw* and project the same into world coordinates while creating visualization for the movement of LEDs.
The project function uses the camera matrix which is determined by the calibration of the cameras to identify the corresponding coordinates in world frame.

Node name	Subscribes to	Publishes to	Data published
/pixart_camera00 */pixart_camera01* */pixart_camera02* */pixart_camera03*	*/pixart_raw*	*/pixart_world*	Position data of all LEDs from all cameras
		/camera_outline (00-03)	Data of the workspace and frames for rviz
		/point_stream(00-03)	Position data of points under individual cameras for rviz

11.3. Quick overview of the updated structure

For the *Figure 7.9* we have made a new rqt-graph, where you can see a new additional subscriber to the */pixart_raw - /Calibration_data*.

This subscription is needed in order to obtain result described in a Chapter 3 – Subparagraph 3.2 of this report (JSON data storage - Algorithm of data processing).

References

1) Camera calibration - Theory:
- http://ksimek.github.io/2012/08/22/extrinsic/
- http://ksimek.github.io/2013/08/13/intrinsic/
  Good theoretical sites, with a nice interactive part to see how the change in different extrinsic-intrinsic parameters can influence the image you get at the output.
2) Calibration software, developed by Bogdan Khomutenko:
   https://github.com/BKhomutenko/visgeom
3) OpenCV site with documentation on a calibration method explained in the Chapter 4 of this report:
   https://docs.opencv.org/3.0-beta/modules/calib3d
4) In the report of the first team it was mentioned the usage of a LED array and some reflective tape, you can use the infrared camera in the Wii remote to track objects. Here is a good link:
   http://johnnylee.net/projects/wii/

Tips for the future team:

- Even though there was embedded a new file to block appearance of a “0” index of a CAN-USB reader, sometimes you can steal try to run the init.sh file and receive an error, finding out that somehow you do have a “1” index. In this case you will need to reboot your computer and re-run it.
- In contrast to the last year works, this year we were working on ROS-Kinetic instead of ROS-Indigo –except for the few new changes in a work with a command line (*catkin build vs catkin_make*) we did not find any big difference.
- If you will have any additional questions you can try to contact us via email or Facebook.