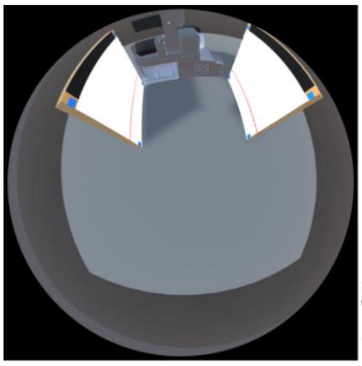
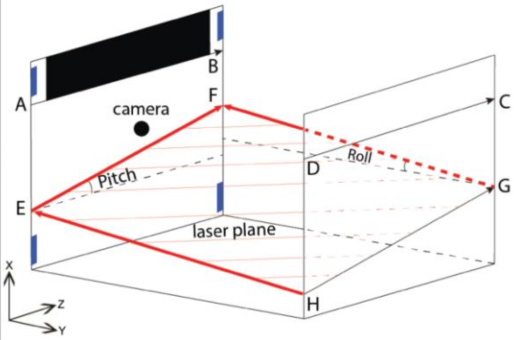
Extrinsic System Calibration

Camera Rotation matrix

Target has white and black regions, we are interested in the border extraction between them, which is parallel to the floor. Moreover, target is located in such a way that sides of it are parallel to the corresponding walls of the indoor environment. Target is presented below.





First of all, let's make an initialization process:

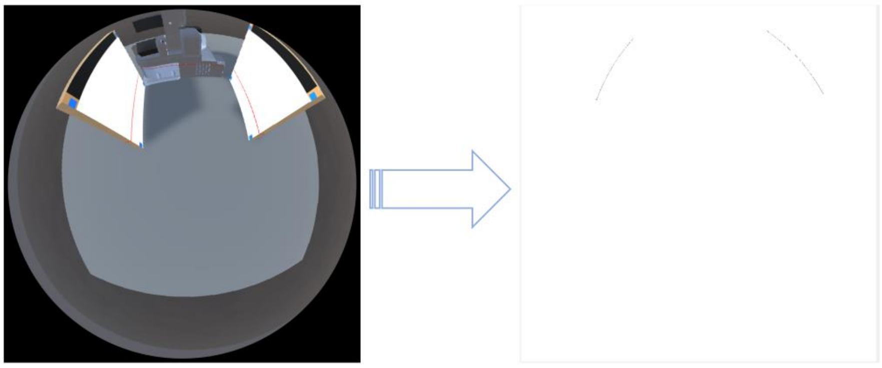
```
% Initialize
clc
clear all
tic;
global ocam_model;
global Cent;
global Cent1;
global cam_pitch_opt;
global cam_roll_opt;
global cam_yaw_opt;
global las_pitch_opt;
global las_roll_opt;
global Cube_width;
load('Omni_Calib_Results_9sim.mat');
ocam_model = calib_data.ocam_model;
```

We defined global variables here, in order to use them in the objective functions, it'll be clearer later.

After that we can load our image and extract border between black and white regions.

```
%% Border extraction
I1_ = imread('images/11.png'); % read image
BW = BWBorder_(I1_); % extract border between Black & White regions
figure
imshow(~BW);
```

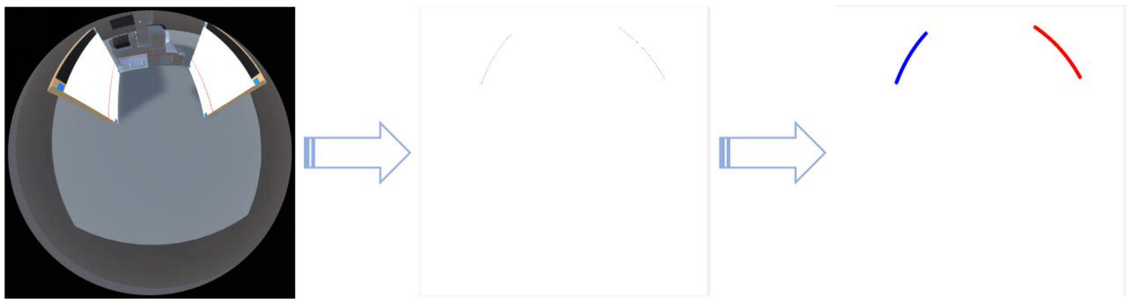
As a result we will get the following image:



From the figure above it can be seen that it has 4 belonging to the side's regions. The code bellow is simply verifying these number of regions and also fitting curves to every region and extracts points from them [Cent, Cent1]. In order to eliminate noise and also sort them in the following order: Cent - right side, Cent1 - left side:

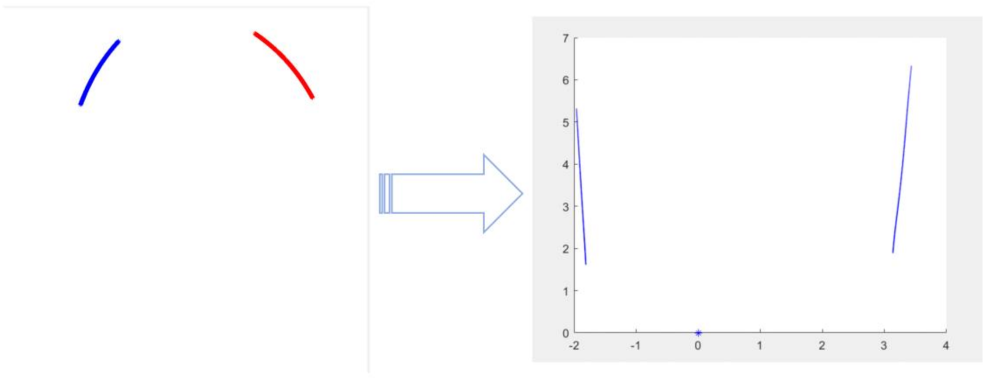
```
%% Border extraction
I1_ = imread('images/11.png'); % read image
BW = BWBorder_(I1_); % extract border between Black & White regions
figure
imshow(~BW);
% extract border for each side of the box
[Cent,Cent1,idx]=border_blobs(BW);
% check whether number of blobs correct or not
if (idx ~= 2)
    clear;
end
```

We've got fitted points for each of the side in arrays [Cent, Cent1] and these points are also shown in the figure below:



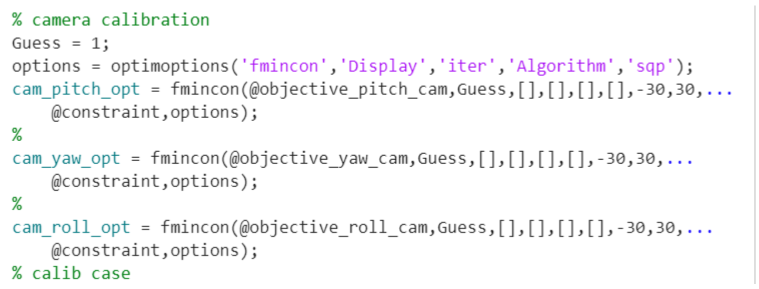
About Once border points were extracted, we can move to the camera calibration itself. Our target has known rectangular shape, the goal of calibration is to obtain parameters by which projected border will have a right shape. Firstly, in order to show that projected rectangular will have a different shape where even small camera orientation presents, we set up orientation parameters of camera orientations are equal. Camera calibration does not change when we are only interested in the orientation parameters, distance between camera and border can be set up randomly (geometry is not changed), here it's set up equal to '1':

```
% calib case
[x,y] = mapping_points(Cent,cam_roll_opt,cam_pitch_opt,cam_yaw_opt,...
    0,0,1,ocam_model);
[x1,y1] = mapping_points(Cent1,cam_roll_opt,cam_pitch_opt,cam_yaw_opt,...
    0,0,1,ocam_model);
scatter(x,y,5,'b.'); % border intersections
hold on
scatter(x1,y1,5,'b.'); % border intersections
scatter(0,0,'b*'); % Robot location
```



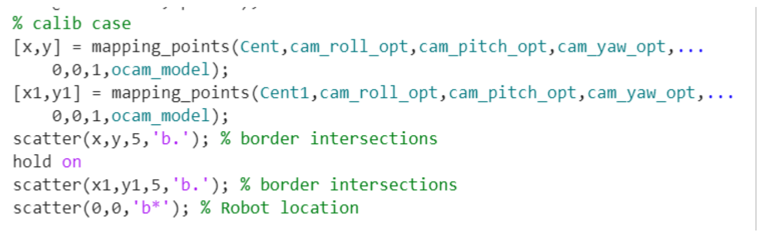
By this figure it can be seen that projected rectangular doesn't have a desirable shape and orientation, because parameters of the camera orientation (pitch, roll, yaw) are different from the written previously zeros in the camera orientation matrix. Pitch, roll and yaw can be found with the aid of frmincon function, which can find minimum of constrained nonlinear functions. The desirable shape is the following:

- Vectors AB and DC are collinear;
- Vectors DA and CB are collinear;
- Vectors DA and DC are orthogonal.

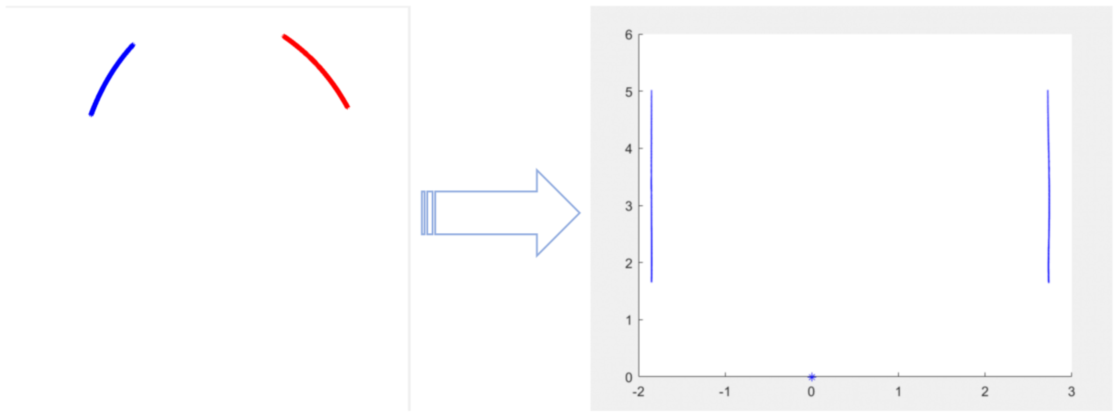


```
% camera calibration
Guess = 1;
options = optimoptions('fmincon','Display','iter','Algorithm','sqp');
cam_pitch_opt = fmincon(@objective_pitch_cam,Guess,[],[],[],[],-30,30,...
    @constraint,options);
%
cam_yaw_opt = fmincon(@objective_yaw_cam,Guess,[],[],[],[],-30,30,...
    @constraint,options);
%
cam_roll_opt = fmincon(@objective_roll_cam,Guess,[],[],[],[],-30,30,...
    @constraint,options);
% calib case
```

Now we can project border to the world coordinates with the calculated pitch, roll, and yaw:



```
% calib case
[x,y] = mapping_points(Cent,cam_roll_opt,cam_pitch_opt,cam_yaw_opt,...
    0,0,1,ocam_model);
[x1,y1] = mapping_points(Cent1,cam_roll_opt,cam_pitch_opt,cam_yaw_opt,...
    0,0,1,ocam_model);
scatter(x,y,5,'b.'); % border intersections
hold on
scatter(x1,y1,5,'b.'); % border intersections
scatter(0,0,'b*'); % Robot location
```



After that we can compare experiment data with the real one:

	Pitch,degrees	Roll,degrees	Yaw,degrees
Real values	-3	-4.5	5
Experiment values	-2.78	-4.26	4.93
Error(Real-Experiment)	0.22	0.24	0.07

From the table it can be seen that orientation parameters were accurately found. Camera orientation parameters will be used during the next step, which is related with the laser plane calibration.

Laser plane calibration

We are going to understand orientation of the laser plane with regard to the camera as well as distance between camera and laser plane. In the ideal case we need to get our setup angles in degrees: pitch = -1, roll = -2.

First of all, we have to extract laser pixels from the target by thresholding, for that procedure 8 blue regions were added to the target's corners, so once they were detected we can create a mask with the aid of them and find laser points belonging to every side.

```
% for las
Cent = [];
Cent1 = [];
I_ = las_segm(I1_,-80);
[xx_,zz_,idx]=blue_blobs(I_,I1_);
% check whether number of blobs correct or not
if (idx ~= 8)
    clear;
end
```

Laser extraction for each side of the target and fitting curves for eliminating noise:

```
% fitting laser points for each side of the target
img1 = las_segm(I1_,138);
img1 = bwmorph(img1,'skel',Inf);
[Cent, Cent1]=intersect_blobs(img1,zz_,xx_); % Cent1,2-h, Cent2,3-v
% laser plane calibration
```

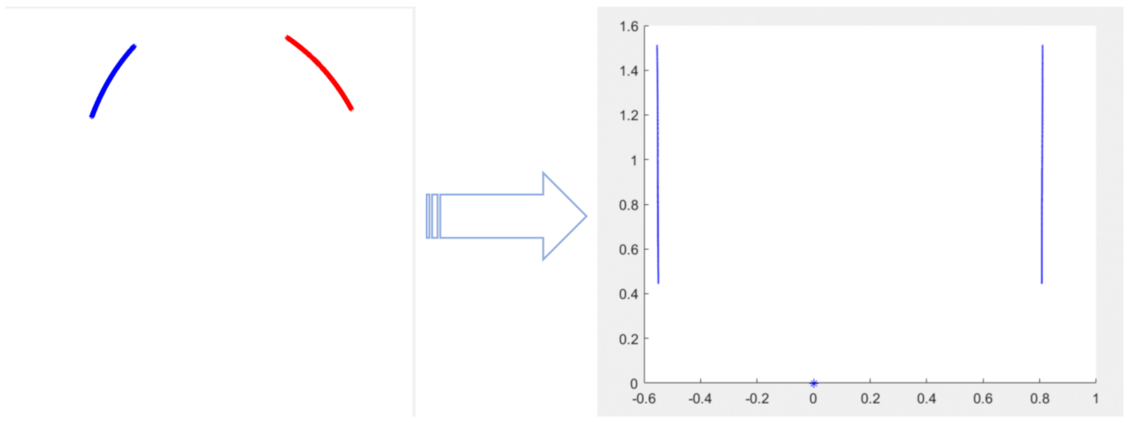
Laser plane doesn't have yaw, which means that once pitch, roll and distance from camera to the laser plane are found, vision system is calibrated. Pitch and roll here can be found by different from the camera minimization technique. As it doesn't have yaw, so we can calculate pitch and roll by minimization multiplication of parallel vectors, namely for pitch: EF and HG; for roll: HE and GF. When these vectors become parallel to each other (multiplication is close to zero) pitch and roll are found:

```
% laser plane calibration
las_pitch_opt = fmincon(@objective_pitch_las,Guess,[],[],[],[],-30,30,...
    @constraint,options);
las_roll_opt = fmincon(@objective_roll_las,Guess,[],[],[],[],-30,30,...
    @constraint,options);
```

Once this procedure has been done, we can project laser points to the world coordinates with the known calibration parameters:

```
[x,y] = mapping_points(Cent,cam_roll_opt,cam_pitch_opt,cam_yaw_opt,...
    las_roll_opt,las_pitch_opt,las_dist_opt,ocam_model);
[x1,y1] = mapping_points(Cent1,cam_roll_opt,cam_pitch_opt,cam_yaw_opt,...
    las_roll_opt,las_pitch_opt,las_dist_opt,ocam_model);
scatter(x,y,5,'b.'); % Laser intersections
hold on
scatter(x1,y1,5,'b.'); % Laser intersections
scatter(0,0,'b*'); % Robot location
```

Result is shown bellow:



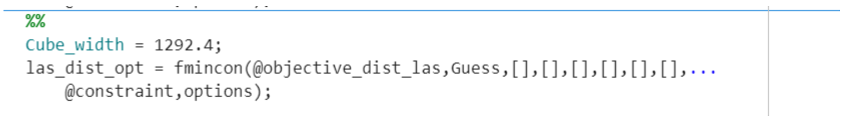
From that figure it can be seen that geometry is correct, moreover let's compare experiment values with the real ones:

	Pitch,degrees	Roll,degrees
Real values	-1	-2
Experiment values	-1.34	-1.89
Error(Real-Experiment)	0.34	0.11

From the table it can be seen that orientation parameters were accurately found. Camera and laser plane orientation parameters will be used during the next step, which is related with the distance between camera and laser plane.

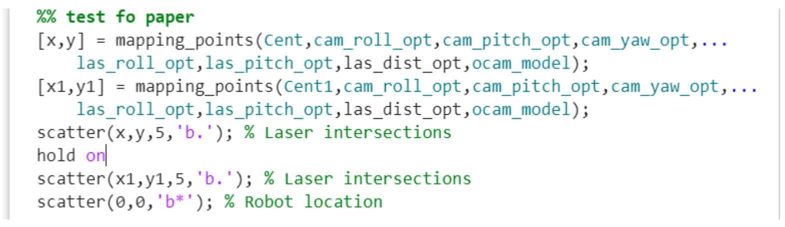
Distance between camera and laser plane

The last unknown parameter related with the calibration of our vision system is the distance between camera and laser plane. This distance can be found with the aid of the knowing size of the target. Surface can be calculated after measuring the known real distances between sides of the target – S1. It also can be calculated from the projected world coordinates of the laser beam – S2. World points are calculated with the known orientation parameters of camera and laser plane, which means that the S2 depends only on the distance between camera and laser plane. Thus, by minimizing difference between S1 and S2 in the objective function, it becomes possible to find the depending variable, which represents distance between camera and the laser plane. Aforementioned procedure has the following form:

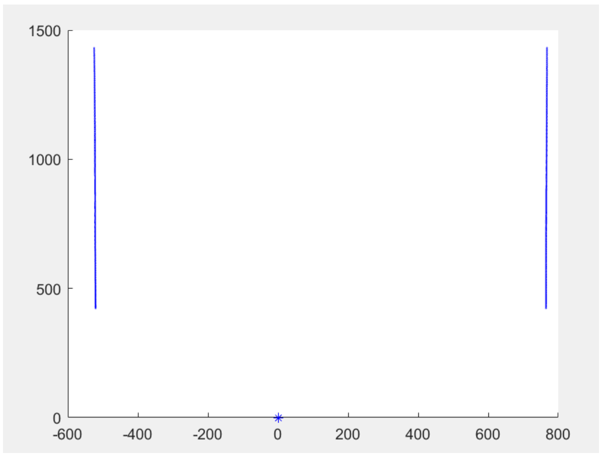


```
%%
Cube_width = 1292.4;
las_dist_opt = fmincon(@objective_dist_las,Guess,[],[],[],[],[],[],...
    @constraint,options);
```

Once this procedure has been done, we can project laser points to the world coordinates with the known calibration parameters:



```
%% test fo paper
[x,y] = mapping_points(Cent,cam_roll_opt,cam_pitch_opt,cam_yaw_opt,...
    las_roll_opt,las_pitch_opt,las_dist_opt,ocam_model);
[x1,y1] = mapping_points(Cent1,cam_roll_opt,cam_pitch_opt,cam_yaw_opt,...
    las_roll_opt,las_pitch_opt,las_dist_opt,ocam_model);
scatter(x,y,5,'b.'); % Laser intersections
hold on
scatter(x1,y1,5,'b.'); % Laser intersections
scatter(0,0,'b*'); % Robot location
```



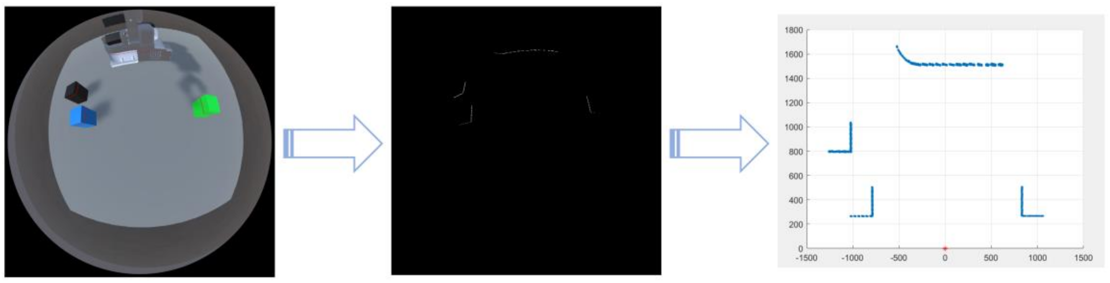
From that figure it can be seen that geometry is correct, as well as the distance between parallel sides, bellow we show comparison of real distance with the experiment one:

	Distance, mm
Real values	950.00
Experiment values	947.32
Error(Real-Experiment)	2.68

Once vision system was calibrated, we can place several obstacles to the simulation environment in order to conduct results validation.

Results

Now we can compare our input image with the output one. As you can see, output image contains only distance information of the laser strip intersection with the obstacles:



Next we are able to compare real distances with the experiment ones:

	Left Cube, mm	Front Cube, mm	Right Cube, mm
Real values	1000	270	-835
Experiment values	1023.5	269.2	-833.7
Error (Real-Experiment)	23.5	0.8	1.3

From the table above it can be seen that calibration results give us accurate distance measurements to the obstacles.