Exercise 3: How to Build a Range Finder using an Omnidirectional Camera

Dr. Davide Scaramuzza, CLA E14.3, 044 632 50 66, davide.scaramuzza @ieee.org
Stefan Gächter, CLA E11.2, 044 632 73 95, stefan.gaechter@ mavt.ethz.ch

Version 1.4, June 12, 2008

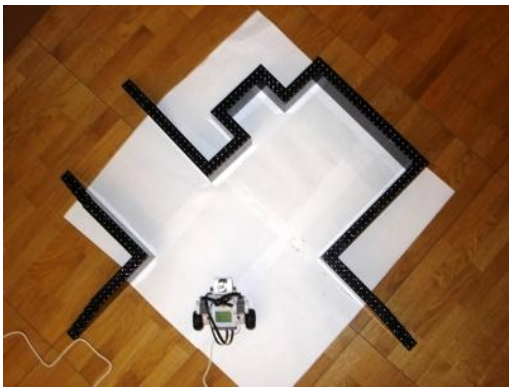


Fig. 1

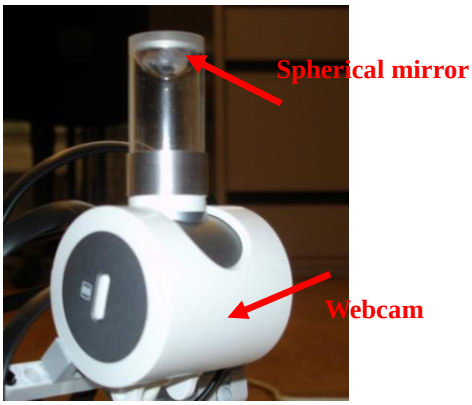


Fig. 2

Fig. 3

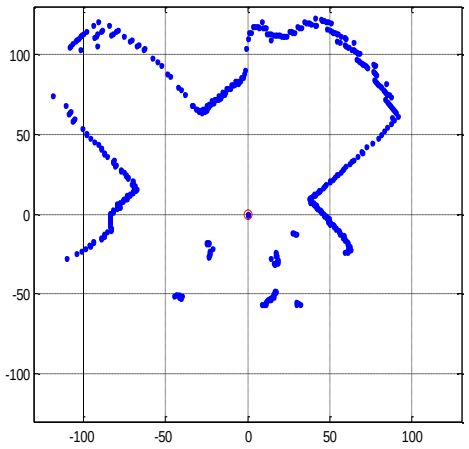


Fig. 4

1. Introduction

Suppose your LEGO-Robot is moving in an indoor structured environment and needs to detect the distances to the walls around it (Fig. 1 and Fig. 4). In mobile robotics, the devices that are usually used to accomplish this task are time of flight sensors like laser range finders and ultrasonic detectors. These sensors operate by emitting a pulse (i.e. laser or acoustic wave) towards the objects and measuring the time taken by the pulse to

be reflected off the target and returned to the sender. By changing the direction of the emitter, one can measure the distance in a plane in all directions. In this exercise, you will have to build a range sensor that, instead of using a laser beam or acoustic wave, uses an omnidirectional camera to measure the distance to the objects.
The omnidirectional camera you will use consists of a standard webcam and a spherical mirror (Fig. 2). Because the camera is looking upward at the mirror, a 360° field of view of the surrounding environment can be obtained. An example of omnidirectional image is depicted in Fig. 3.
Suppose the robot moves along a corridor and registers images. The robot has to detect the wall contours in the images, that is, the points separating the floor from the wall itself. To facilitate the detection of the wall contours, the environment is made on purpose of black walls and white floor.
The goal of this exercise is to find the wall contours and then measure the distances from the omnidirectional camera to the walls. The main steps of the exercise are as follows:

1. Acquiring an image using the Matlab Image Acquisition Toolbox.
2. Calibrating the camera-mirror system. This means detecting the center of the image and specifying internal and external boundaries of the omnidirectional image.
3. Detecting the wall contours using an appropriate image processing technique.
4. Calculating the distance to the walls by using the knowledge of the optics of the camera-mirror system.
5. Propagating the uncertainties from the pixel points to the distances and illustrating them by uncertainty ellipses.

The exercise can be solved by carefully following the instructions given in this document. The implementation is done in MATLAB using some additional proprietary functions that are included in the package of this exercise.
In the package there is a file called **`main.m`** that already contains some code. To complete the exercise you will need to add the functions given in this tutorial to this file.

2. Acquiring an Image

Acquiring a snapshot or a video in Matlab is very straightforward with the Image acquisition Toolbox. Run the script **`test_camera.m`**. This script checks the connected

video device, creates a video object that is used to capture images, and finally displays a real-time video preview. You have to run this script again every time you restart Matlab. Have a look at the script.

The main functions for accessing a video device with Matlab are the functions **`videoinput`** and **`getsnapshot`** :

```
vid = videoinput( AdaptorName, DeviceID, format );
snapshot = getsnapshot( vid );
```

`videoinput` creates a video **object** **`vid`**. The input arguments are **`AdaptorName`** , the name of the video library - in windows usually **`winvideo`**, **`DeviceID`** , the address of the video device - usually 1, and `format` , the image size - in our case **`RGB32_640x480`**.

`getsnapshot` captures and returns an image. You can use **`imagesc(snapshot)`** to visualize the acquired image. Here, **`snapshot`** is a RGB image. If the image is an indexed image that stores colors as an array of indices into a colormap, you can use **`colormap(gray)`** to display a binary or gray scale image as will result later in the exercise.

ATTENTION

Because our omnidirectional camera is composed of a mirror, in the reminder of this exercise you will have to flip every acquired frame. For doing this, use function **`imflipud`**:

```
snapshot = imflipud( snapshot );
```

This function automatically flips your image up-down. In the reminder of this exercise, it will be assumed that the image has been already flipped. However, in the next section on camera-mirror system calibration the image is automatically flipped by the script.

3. Calibrating the Camera-Mirror System

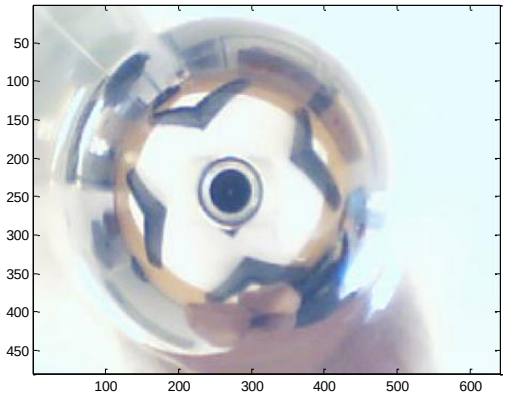


Fig. 5

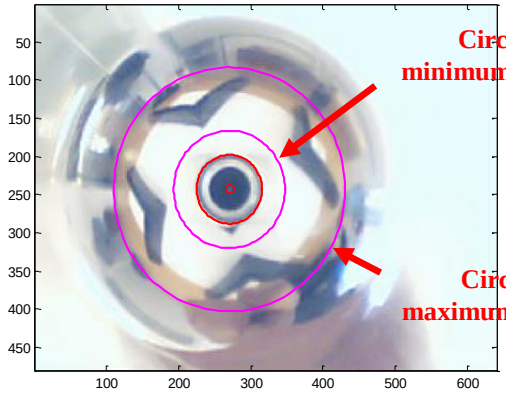


Fig. 6

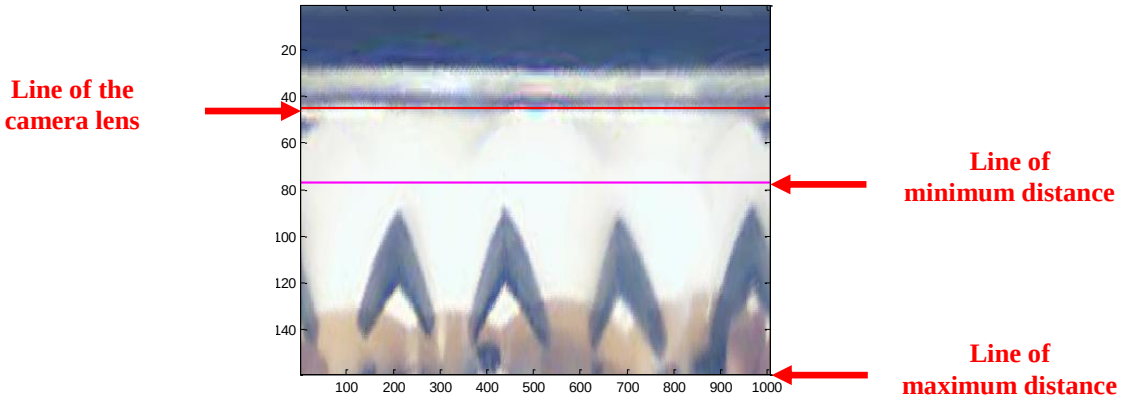


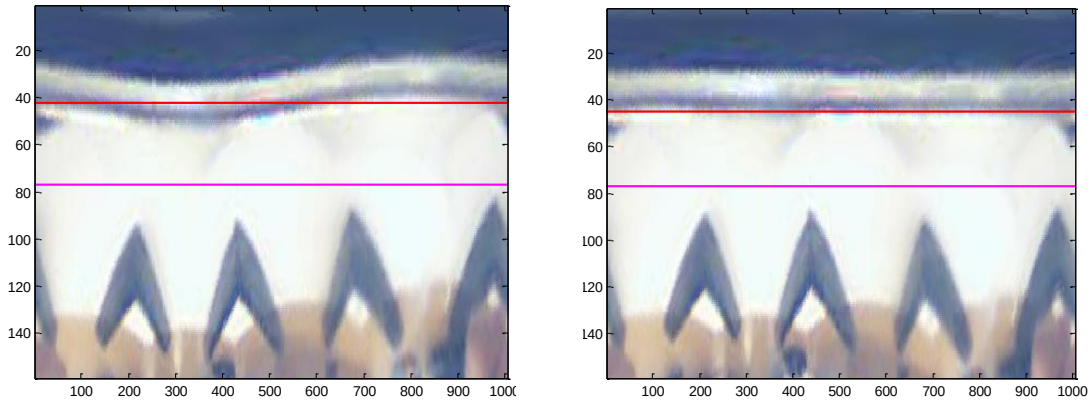
Fig. 7

Calibrating the camera-mirror system means to find the center of the omnidirectional image. The knowledge of the position of the center is necessary, because the distance to the wall is referred to this center.

To automatically detect the center of the image run the script **`calibrate_camera.m`**. The program will acquire a new snapshot from the camera and will ask you to manually click on two different points. The first one is the point where the center is supposed to be located. The second one can be any point along the perimeter of the first circle around the center (in our setting this circle is the border of the camera lens). The result of the detection of the center is depicted in Fig 6. In this case the snapshot will be flipped automatically by the script.

As observed in Fig. 6, there are two main circles called “maximum distance” and “minimum distance”. These two circles delimit the region of interest where we want to detect the walls. The minimum distance circle is used here to ignore the image region covered by the robot itself.

Fig. 7 depicts the unwrapped version of Fig. 6, that is, the image that would be generated by rotating a standard camera about its center. To generate this undistorted image, each radial line of the original image is mapped into a vertical line. Fig. 7 is very useful to check the quality of the calibration. Indeed, if the calibration is correct, the border of the camera lens should appear parallel to image border. If this parallelism is not verified (Fig. 8), then you will have to run again **`calibrate_camera.m`**.



(a) (b)

Fig. 8. a) Camera not well calibrated . b) Camera well calibrated.

Important variables that are generated by the calibration script are:

`center`	Coordinates (row, column) of the image center
`Rmin`	Minimum distance radius
`Rmax`	Maximum distance radius

4. Detecting the Wall Contours

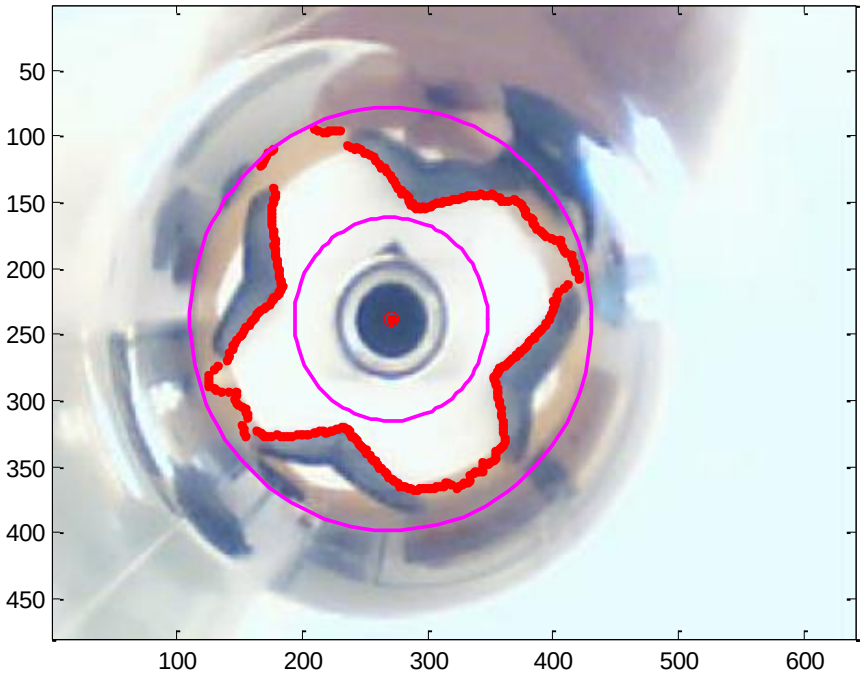


Fig. 9

In this section, we propose an approach to detect the wall contours (in red in Fig. 9).
Our idea consists in scanning each radial line departing from the image center and searching for the first intensity step between white and black. This can be seen as a simplified edge detection!
To do this, we have to unwrap the omnidirectional image into a rectangular image:

```
[undistortedimg, theta] = imunwrap(snapshot, center, angstep, Rmax, Rmin);
```

where **snapshot** is the acquired and flipped image, **center** are the coordinates of the image center, **angstep** is the angular step between two successive radial lines, and , **Rmax** and **Rmin** are the maximum and minimum distance radii. **undistortedimg** is the unwrapped image (Fig. 10).
If you want to display this image, use this command:

```
figure;
imagesc(undistortedimg);
colormap(gray); %This line is to visualize the image in grayscale
```

The undistorted image is similar to Fig. 7. The only two differences are that the image is rotated by 90° and that the region within **`Rmin`** has been set to white, meaning that we want to ignore the internal part. Each horizontal line of Fig. 10 corresponds to a certain orientation angle **`theta`** of the original radial line (Fig. 11).

Use **`angstep`** to change the angular step between two successive radial lines. By changing **`angstep`**, you actually change the angular resolution of our simulated laser range finder sensor (see Fig. 11)!

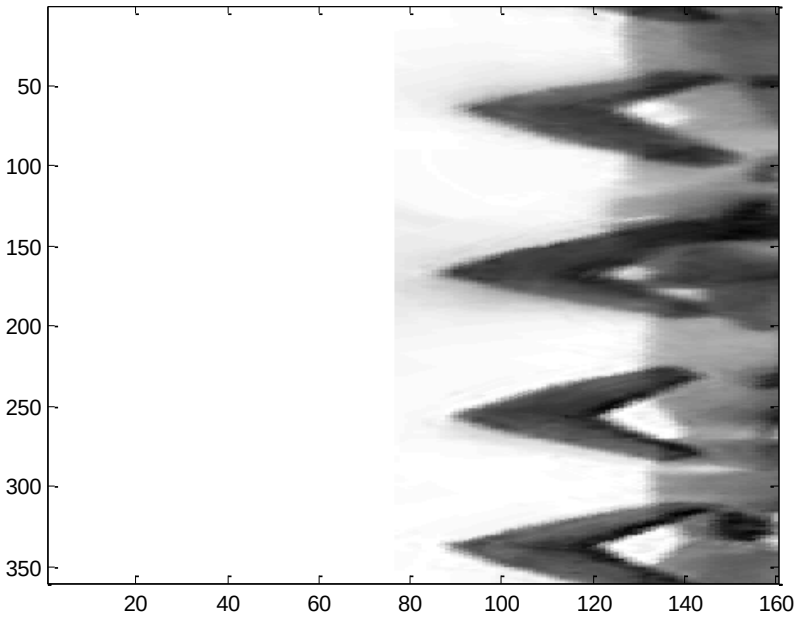


Fig. 10

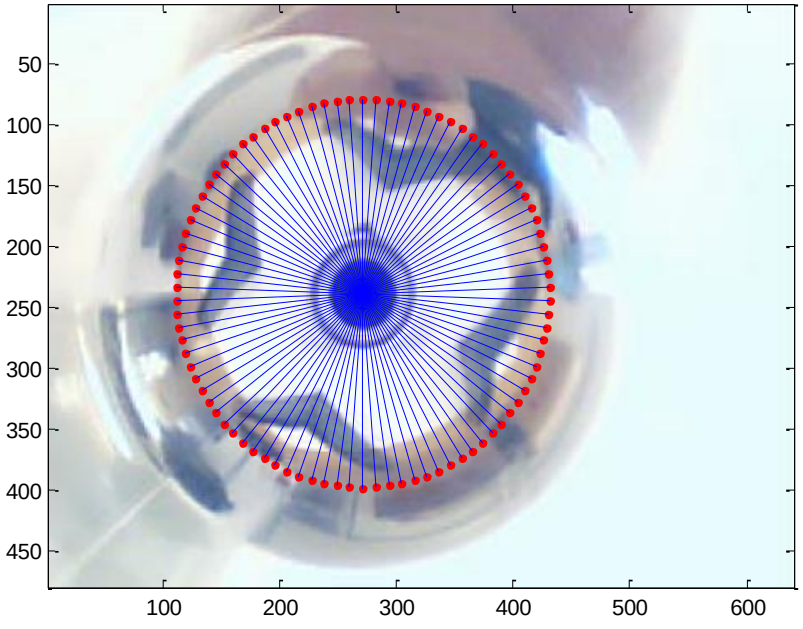


Fig. 11

At this point, you may want to segment the undistorted image into black and white:

```
BWimg = img2bw( undistortedimg, Bwthreshold );
```

What is **`Bwthreshold`** ? Try to determine a suitable value!
The binary image **`BWimg`** is depicted in Fig. 12.

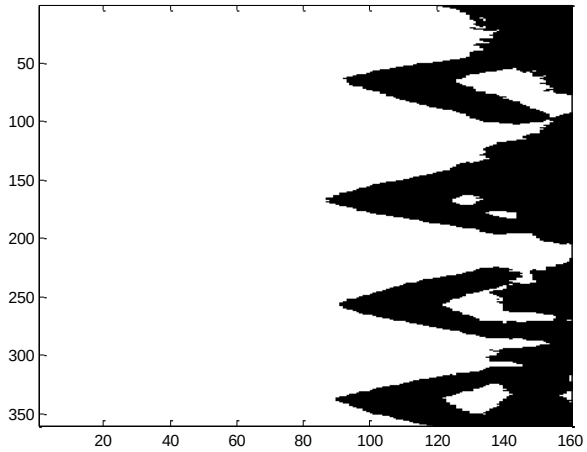


Fig. 12

The distance in pixel to the wall of each radial line in Fig. 11 corresponds to the number of pixels from left image border until the first black point in Fig 12. The following command returns all the distances:

```
rho = getpixeldistance( BWimg , Rmin );
```

You should be able to generate a similar as Fig. 9 with:

```
figure;
imagesc( snapshot );
hold on;
drawlaserbeam( center, theta, rho );
hold off;
```

5. Convert Pixel Distance to Metric Distance

The distance **`rho`** computed in the previous stage does not represent the real distance in

meters to the wall but only the image distances in pixels.
If one knows the optics of the camera-mirror system, it is however possible to recover the metric distance (Fig. 13).

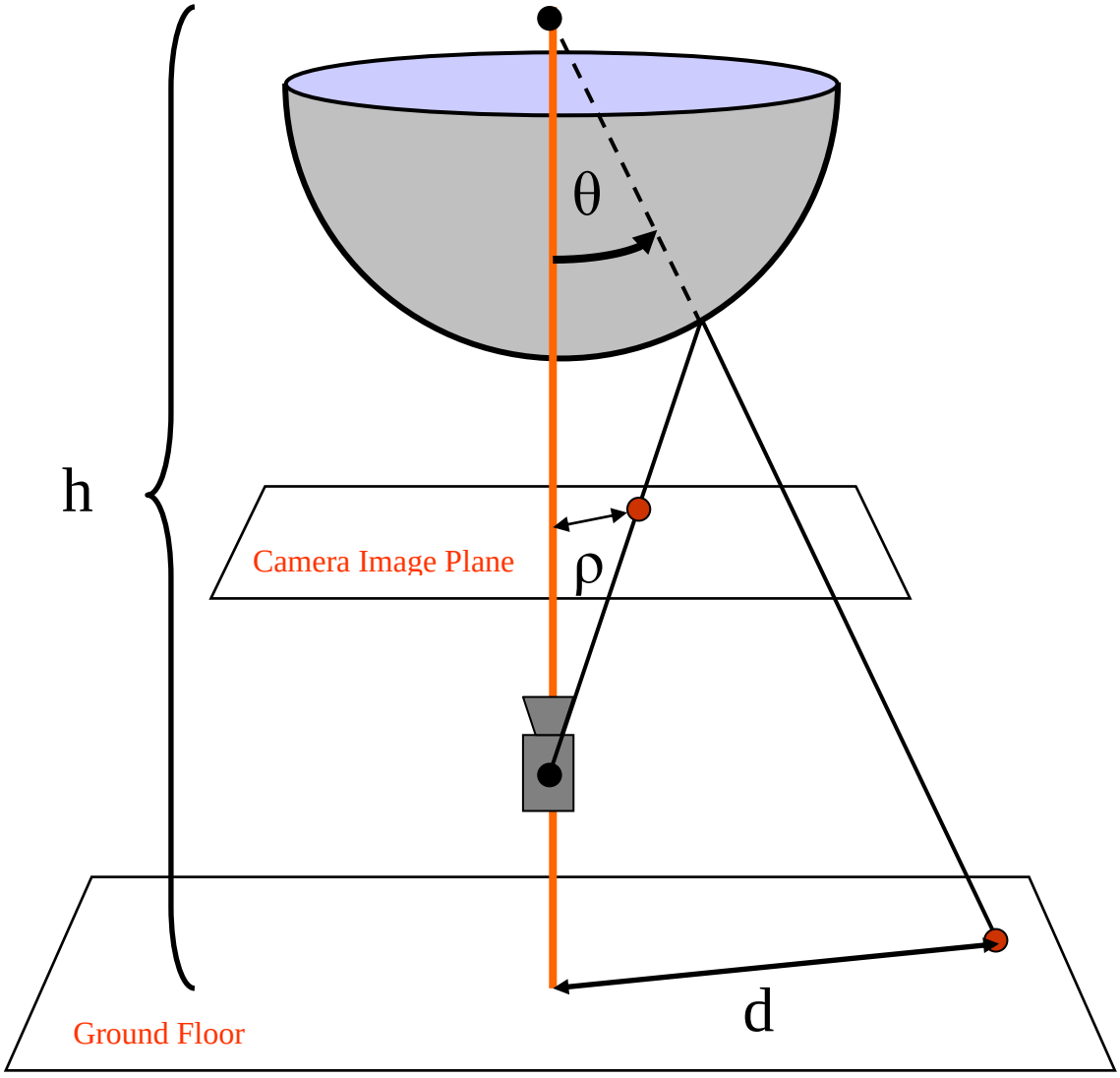


Fig. 13

Let d be the distance in meters on the ground floor and ρ the distance in pixels in the image plane. Then, we have:

$$d(\theta) = h\tan(\theta)\,. \qquad (1)$$

For most omnidirectional cameras, the relation between θ and ρ can be approximated by a first order Taylor expansion, that is:

$$\theta \approx \frac{1}{\alpha}\rho, \tag{2}$$

where α depends on the mirror shape and the camera-mirror distance. Then, the final relation becomes:

$$d(\rho) = h \tan\left(\frac{\rho}{\alpha}\right) \tag{3}$$

In our exercise, we use α = 95 pixel. You have to calibrate the height h in meter in order to get the correct distance measurements. Thus, use a reference distance on the ground floor measured by hand and compare it with result obtained by equation (3).

Use **`undistort_dist_points`** and **`draw_undistorted_beam`** to recover the real distances and display the metric map:

```
dist = undistort_dist_points( theta , rho , alpha , height );
figure;
draw_undistorted_beam( dist , theta , axislimit );
```

`axislimit` is a scalar that limits the plot axes to their minimum and maximum values. The result is depicted in Fig. 14.

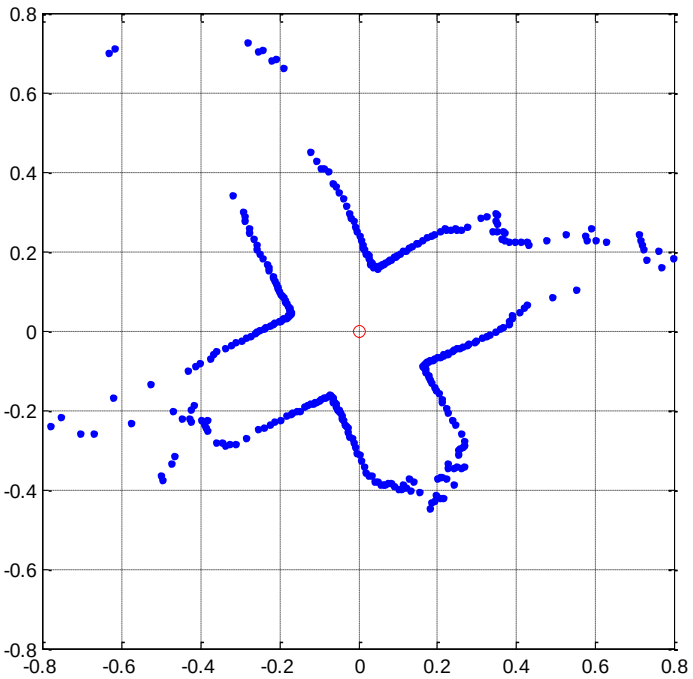


Fig. 14

6. Error Propagation

Because the image acquisition is affected by noise, the wall detection and, thus, the

distance measurement are affected by uncertainty. We want to quantify the error in the distance measurement. Therefore, we have to propagate the uncertainty of the distances in the image plane to the metric distances on the ground floor. The error propagation law uses a first order Taylor expansion of relation (3) :

$$\frac{\partial d(\rho)}{\partial \rho} = \frac{h}{\alpha}\left(1+\tan\left(\frac{\rho}{\alpha}\right)^2\right). \tag{4}$$

If we assume a pixel distance error with normal distribution of mean value ρ and standard deviation σ_ρ, then the metric distance error has a normal distribution of mean value d and standard deviation σ_d. According the error propagation law, the standard deviation is then

$$\sigma_d = \frac{\partial d(\rho)}{\partial \rho}\sigma_\rho . \tag{5}$$

Write yourself the function **`compute_uncertainty`** to compute the distance errors in (5) and use then **`draw_uncertainty`** to and display them in the metric map:

```
sigma_dist = compute_uncertainty( rho, sigma_rho, alpha, height );
hold on;
draw_uncertainty( dist, theta, sigma_dist );
```

The result is depicted in Fig. 15.

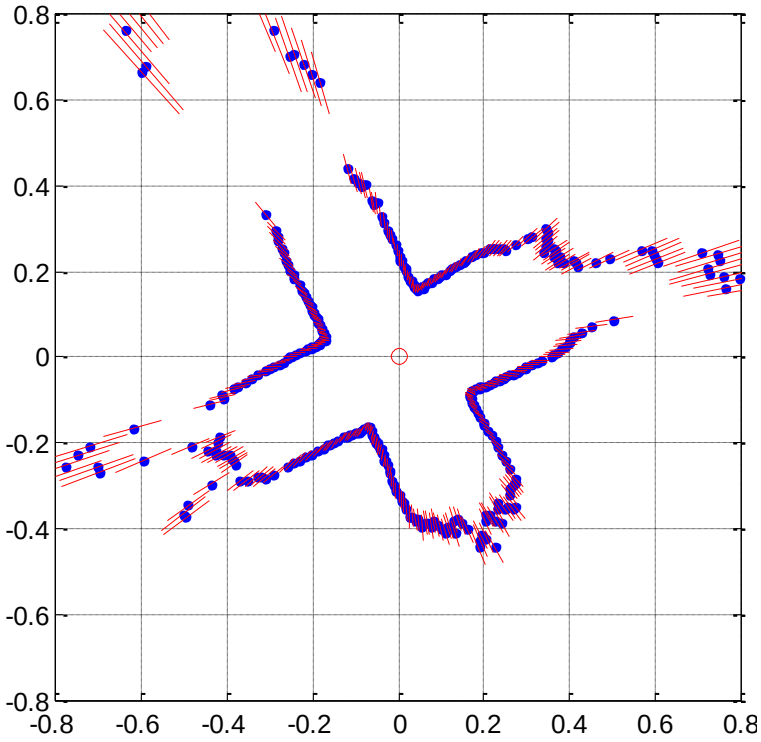


Fig. 15