User manual of LidarSafeGuard

Yaoyu Hu

huyaoyu@sjtu.edu.cn

2017-12-18

Table of contents

1 Introduction ... 1
2 Get the source code ... 2
3 Compiling and linking ... 2
4 Namespace ... 2
5 Local frame and *LidarMask* ... 3
6 Class *LidarSafeGuard* ... 5
7 Class *InCylinderSafeGuard* ... 5
8 Class *InCylRgrSafeGuard* ... 7
9 Test cases ... 10
  9.1 test_naive_situation() ... 11
  9.2 test_perfect_circle_with_noise() ... 11
  9.3 test_data_on_launchrig_no_mask() ... 12
  9.4 test_data_on_launchrig_mask() ... 13
  9.5 test_data_on_launchrig_mask_regression() ... 13
10 Acknowledgement ... 13
11 References ... 13

1 Introduction

A LIDAR detector is used in a new kind of robot aiming at in-pipe inspection as the safeguard component. The LIDAR will constantly make measurement while the robot is operating. As for safeguarding purpose, there should be a code which could tell whether the robot is inside the pipe or not, based on the range data obtained by the LIDAR.

For the robot here in question, a specific LIDAR, the RPLIDAR, will be used as the safeguard. This piece of code is designed to analyze the range data measured by an RPLIDAR to determine the in/out pipe condition.

The RPLIDAR will be installed on the back of the robot with no obstacles in all the detectable zone. This is shown in Fig. 1. Some assumptions are made to the scenarios:

(1) The perfect pipe has a cylinder or tube shape. The cross-section of the pipe is a circle.
(2) The RPLIDAR is installed perfectly perpendicular to the pipe wall.

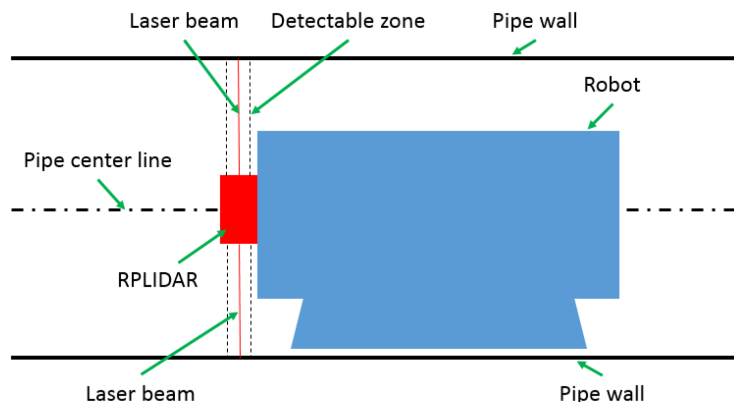

Fig. 1 Installation of the RPLIDAR.

The code is designed to figure out the most possible radius based on the assumption that the cross-section is a circle. Then the user could compare the analyzed radius with the actual average radius of the actual pipe. A standard deviation will be also obtained by the analysis. The user could determine whether the robot is inside a pipe or not based on the absolute differences between the analyzed radius and the actual average radius, considering the calculated deviation. To deal with some special cases, like the launch positioning, there is a mask mechanism to facilitate the radius analysis. The user could specify some angle regions that should be deliberately ignored when performing the analysis. This code is now shipped with two methods to analyze the radius, one based on simple statistical process and the other one adopts a regression model. Test cases will be given to compare both accuracy and performance.

The code is developed in C++ since the target system runs C++ developed software to control the robot. The target operating system is assumed to be Ubuntu 16.04 or above. Some third-party packages are needed. The third-party packages are the Boost library and the Eigen package. They are both freely available and extremely easy to deploy.

This project will be called “LidarSafeGuard” from now on.

This user manual will discuss the underlying method of the radius analysis together with sample codes and test cases.

2 Get the source code

The LidarSafeGuard project is now hosted on Github. The user could get access to the code by visiting:

https://github.com/huyaoyu/LidarSafeGuard

3 Compiling and linking

First of all, for any C++ project, it is possible that the user will experience a painful process to get everything properly compiled and linked. To compile and link against LidarSafeGuard, one need Boost and Eigen.

Boost is shipped with Ubuntu operating system and that version suffices. Eigen could be obtained from its official website. Eigen is extremely easy to work with since it only contains C++ header files, no additional linking effort will be needed. So after the user deployed Eigen system-widely (unzip the achieve file and make a symbolic link to /usr/local/includes, maybe), the compiling recipe of LidarSafeGuard needs no additional options. To summarize, if the user is going to only compile and link LidarSafeGuard, we need no additional modifications to the compiling and linking commands.

However, if the user would like to compile the test cases provided together with the project, she/he may need to specify *–lboost_chrono* and *–lboost_system* in their linking command.

4 Namespace

A namespace called *RP* is defined by LidarSafeGuard. And the *double* type is wrapped by the *RP::real* type in case the user would like to use *float* type to represent floating point values. All the classes are defined under the namespace *RP*.

5 Local frame and *LidarMask*

By default, the RPLIDAR returns a round of range measurements covering 2 π of angle range in the anti-clockwise direction, in the local frame (coordinate system) attached to its center. The local frame is illustrated in Fig. 2. The x'-axis is along the cable hole of the outer shell and the y'-axis is vertical to it obeying the right-hand rule. The data returned by RPLIDAR always start from angle position near $-\pi$ all the way to π.

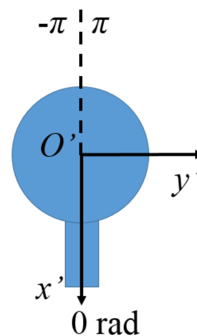

Fig. 2 The local frame of RPLIDAR.

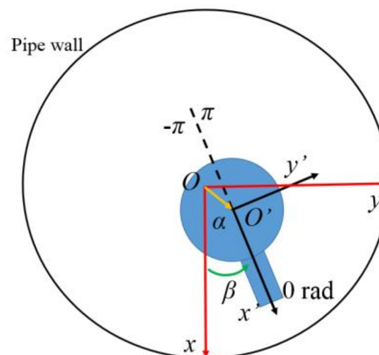

Fig. 3 The local frame and global frame.

The user should figure out how the RPLIDAR is installed in reality, especially the relative phase angle, β, between the x'-axis of the local frame and the x-axis of the global frame defined with respect to the pipe, as illustrated in Fig. 3. Knowing the relative phase angle is important since the user may want to deliberately ignore some range of angles when trying to analyze the radius. The reason for ignoring angle range is that for some special case the robot may operate in a pipe segment which has only a portion of a full circle as its cross-section, e.g. the launch rig. Only the lower half of a circle may be detected when the robot is sitting on the launch rig. If the user needs LidarSafeGuard to give a good prediction of the radius, the upper half must be ignored in the analysis. When it is the case the user would like to work on, the *LidarMask* class will come into play.

The user uses the *LidarMask* to define a mask of angle positions that will be

ignored in the future analysis process. A *LidarMask* object could contain several angle segments described by the *AngleSegment_t* struct. The user specifies an angle segment in an *AngleSegment_t* struct variable, then this variable can be added or registered to a *LidarMask* object. The user could add multiple *AngleSegment_t* structs to *LidarMask* object as long as the structs represent proper angle ranges. For “proper”, it means no cross-covered angle ranges and all the structs are added in an angle-increasing manner. List. 1 shows a sample code for creating a *LidarMask* object and populates it with two *AngleSegment_t* structs.

List. 1 *LidarMask*.

```
1  RP::LidarMask LM(-MY_PI, MY_PI);
2  RP::LidarMask::AngleSegment_t AS;
3  AS.angle0 = -MY_PI;
4  AS.angle1 = -MY_PI / 2.0;
5  LM.copy_push_segment(AS);
6
7  AS.angle0 = MY_PI / 2.0;
8  AS.angle1 = MY_PI;
9  LM.copy_push_segment(AS);
```

The maximum and minimum allowed angles should be specified when the user creates a *LidarMask* object LM (Line 1). Here LidarSafeGuard defined a macro, MY_PI, to represent the value of π. An *AngleSegment_t* struct variable AS is created at Line 2. The user fills AS with lower and upper angles (Line 3, 4). Then AS is added into LM at Line 5. At Line 7 and 8, AS is reused to describe a new angle segment. Then AS is used again to register a new angle segment into LM (Line 9). Now, LM has two angle segments registered. LM could be used in the later process of radius analysis. Fig. 4 shows the masked range of List. 1.

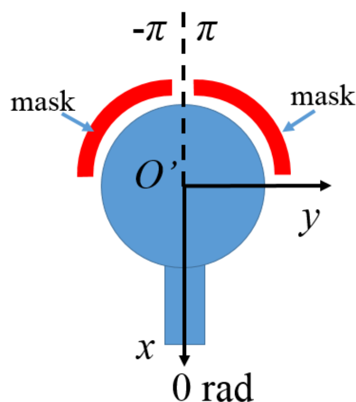

Fig. 4 Masked range of List. 1.

LidarMask will produce a raw C++ array of integers to represent the mask. In that array, 1 means mask 0 means not mask. The array has the same length of the length of

the data array returned from the RPLIDAR. If the length of the data array of RPLIDAR changes dynamically, the length of the mask array will change accordingly and automatically.

The user could always choose not to use a *LidarMask* or even to use a *LidarMask* with no *AngleSegment_t* struct registered. The radius analysis will work in these two cases. However, if an empty *LidarMask* (*LidarMask* object with on *AngleSegment_t* struct registered) is used, a mask array will still be generated with all possible angle positions marked as “do not mask”. This will cause a little waste of system computing resource. But it could be ignored since the performance overhead is negligible in reality.

6 Class *LidarSafeGuard*

LidarSafeGuard has a class called *LidarSafeGuard*. It sounds a little bit confusing, though. Class *LidarSafeGuard* is a base class which provides the basic functionalities of all safeguards. The user could perform the following operations with *LidarSafeGuard*’s member functions.

(1) Set and get the name of a *LidarSafeGuard* object.
(2) Set and get the *LidarMask* object referenced by a *LidarSafeGuard* object.
(3) Check the safety status of the robot.
(4) Examine the ratio of masked regions specified by the referenced *LidarMask* object.
(5) Examine the ratio of Infs in the supplied data measured by the RPLIDAR.
(6) Copy the raw data resulting from the RPLIDAR.

The most important function of the above is the *copy_data()* function which copies data from the raw data of RPLIDAR measurement. The user is encouraged to read the source code of *LidarSafeGuard.hpp* to inspect the argument specifications of *copy_data()*.

The most effective way to show how the *LidarSafeGuard* class works is reading the test case codes. However, each class derived from *LidarSafeGuard* will use the member functions of *LidarSafeGuard* to perform common operations. So we can just refer to the sample codes for the derived classes or the test cases to investigate the usage of class *LidarSafeGuard*.

7 Class *InCylinderSafeGuard*

Class *InCylinderSafeGuard* provides the simplest method to analyze the radius of the cylinder (pipe). The method is to use the average value calculated from the range data returned from the RPLIDAR.

For *InCylinderSafeGuard* to work properly, the user has to provide a reasonably accurate approximation of the eccentricity, *ecc*, between the center of the RPLIDAR and the center of the pipe. There are two more parameters needs to be specified, as shown in Fig. 3, the angle α and β. An *InCylinderSafeGuard* object will use the above three parameters to translate all the ranges points back to the global frame defined with respect to the pipe.

A standard procedure of using InCylinderSafeGuard may look like the following:

(1) Create a new *InCylinderSafeGuard* object icsg with a specified name, eccentricity, and phase angles (α and β).
(2) Set a reference to a *LidarMask* object by *set_mask()* if a mask is needed.
(3) Copy the raw data of the RPLIDAR by *copy_data()*.
(4) Get information of the raw data and the mask by *infRatio = get_inf_ratio()* and *maskRatio = get_mask_ratio()*.
(5) Analysis the radius with *verify()* and obtain a *LidarSafeGuard::SafetyFalg_t* typed value, *flag*.
(6) Retrieve analyzed radius and the associated standard deviation by *radiusMean = get_radius_mean()* and *radiusStd = get_radius_std()*.
(7) Determine the actual safety status from the above five values: *flag*, *infRatio*, *maskRatio*, *radiusMean*, and *radiusStd*.

A practical strategy is that it should be safe if: *flag* == *LidarSafeGuard::FLAG_SAFE*, *infRatio* < some kind of limit with user-specified *maskRatio* < some kind of limit.

Here *infRatio* is calculated by Eq. (1).

$$infRatio = \frac{\text{infs in the unmasked positions}}{\text{all unmasked positions}} \tag{1}$$

and *maskRatio* is calculated by Eq. (2).

$$maskRatio = \frac{\text{masked positions}}{\text{all possible positions}} \tag{2}$$

So for the above mentioned practical strategy, the user has to specify a reasonable limit for *infRatio* considering the actual *maskRatio*. That is to say, *infRatio* should be lower than a certain limit with this limit defined by a known *maskRatio*.

The *verify()* function analyzes the radius by a simple averaging process. Considering a non-zero eccentricity of the center of the RPLIDAR, the averaging process is described in Fig. 5. Initially, the range data is measured in the local frame *x'O'y'*, as illustrated in (1) of Fig. 5. The *InCylinderSafeGuard* object first converts the range data from the "angle-range" pairs to the "*x'* and *y'* coordinates" pairs. Then all

the coordinates (x_i', y_i') are transferred from $x'O'y'$ to xOy, the global frame, as shown in (2) of Fig. 5 and Eq. (3). Finally, the radius (amplitude) of every point, r_i, is evaluated and the mean radius and the standard deviation are calculated using all r_i values, as described in (3) of Fig. 5.

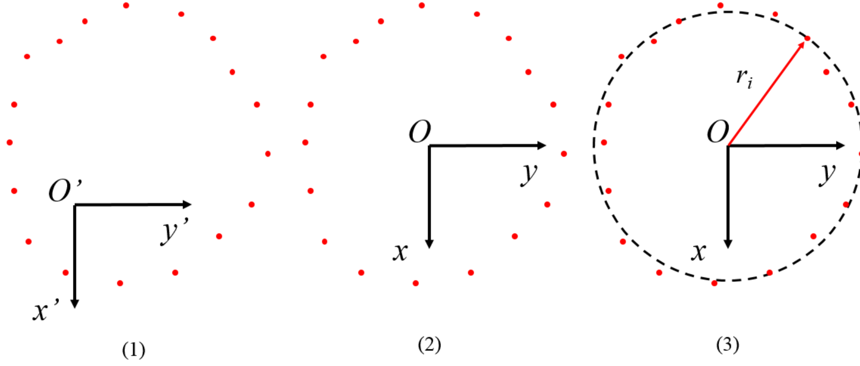

Fig. 5 The averaging procedures.

$$\begin{Bmatrix} x \\ y \end{Bmatrix} = \begin{bmatrix} \cos(\beta) & -\sin(\beta) \\ \sin(\beta) & \cos(\beta) \end{bmatrix} \begin{Bmatrix} x' \\ y' \end{Bmatrix} + ecc \begin{Bmatrix} \cos(\alpha) \\ \sin(\alpha) \end{Bmatrix} \tag{3}$$

After the radius is analyzed, the user could compare it with the real radius of the cylinder and take the standard deviation into consideration. This process in fast. However, the accuracy is heavily based on the eccentricity, α, and β. It is encouraged that the end user makes some additional tests on real cases and to investigate the reasonable limits for (1) the difference between the analyzed radius and the real radius and (2) the standard deviation. The *verify()* function takes these two limits as the input argument in order to give back a flag of safety.

The user could further examine the test cases shipped with LidarSafeGuard to see the overall performance.

8 Class *InCylRgrSafeGuard*

To eliminate the dependence on the input of the eccentricity and the phase angles of class *InCylinderSafeGuard*, LidarSafeGuard provides another class. This is the *InCylRgrSafeGuard*.

When working with *InCylRgrSafeGuard*, the user no longer need to specify the eccentricity and the phase angles. *InCylRgrSafeGuard* could figure out the radius and the center coordinate with respect to the local frame by a regression method. The radius is analyzed in a best-fit sense by an optimization procedure in a least-squares manner. The underlying mathematics will be discussed as follows. It should be mentioned that this method is inspired by the work of Gander, et al.[1]

In a 2D Cartesian coordinate system, xOy, a circle could be represented by Eq. (4).

$$(x-x_c)^2+(y-y_c)^2=r^2 \tag{4}$$

where x_c and y_c are the center coordinates and r is the radius. Let's re-write Eq. (4) in Eq. (5).

$$g(x,y)=(x-x_c)^2+(y-y_c)^2-r^2 \tag{5}$$

For a circle $g(x, y) = 0$.

Now we have a set of range data, (x_i, y_i), $i = 1, \dots N$, that comes from the RPLIDAR. Suppose we would like to find a circle with specific radius, r_0, and center coordinate (x_0, y_0) such that this circle best fits into (x_i, y_i). In other words, this is equivalent to solve the optimization problem listed as Eq. (6).

$$\begin{aligned}&\text{Find } x_c = x_0, y_c = y_0, \text{ and } r = r_0 \text{ to give}\\ &\min\left(F=\sum_{i=1}^{N} f_i\right), f_i=(g_i)^2=\left(g(x_i,y_i)\right)^2\end{aligned} \tag{6}$$

For F, the minimum possible value is 0 and it means all (x_i, y_i) points are located exactly on a circle. The Eq. (6) turns to a new problem that how close we could get F to 0. Let's start with an initial guess of

$$\begin{aligned}x_0 &= x_t+h_x\\ y_0 &= y_t+h_y\\ r_0 &= r_t+h_r\end{aligned} \tag{7}$$

Where we presume that x_0, y_0, and r_0 will make $F = 0$. Now, we should take F as to be a function of x_0, y_0, and r_0. Then we could approximate $0 = F(x_0, y_0, r_0)$ at point (x_t, y_t, r_t) with a first order Taylor series, Eq. (8).

$$\begin{aligned}&F(x_0,y_0,r_0)\\ &=F(x_t,y_t,r_t)+\left.\frac{\partial F}{\partial x_c}\right|_t\left((x_t+h_x)-x_t\right)+\left.\frac{\partial F}{\partial y_c}\right|_t\left((x_t+h_y)-x_t\right)+\left.\frac{\partial F}{\partial r_c}\right|_t\left((x_t+h_r)-x_t\right)\\ &=F(x_t,y_t,r_t)+h_x\left.\frac{\partial F}{\partial x_c}\right|_t+h_y\left.\frac{\partial F}{\partial y_c}\right|_t+h_r\left.\frac{\partial F}{\partial r_c}\right|_t\end{aligned} \tag{8}$$

We can force

$$F\left(x_0, y_0, r_0\right) = F\left(x_{\mathrm{t}}, y_{\mathrm{t}}, r_{\mathrm{t}}\right) + h_x \left.\frac{\partial F}{\partial x_{\mathrm{c}}}\right|_{\mathrm{t}} + h_y \left.\frac{\partial F}{\partial y_{\mathrm{c}}}\right|_{\mathrm{t}} + h_r \left.\frac{\partial F}{\partial r_{\mathrm{c}}}\right|_{\mathrm{t}} = 0 \tag{9}$$

Eq. (9) is satisfied by all the (x_i, y_i) points. Because $f_i >= 0$, so $F = 0$ means $f_i = 0$. Then Eq. (9) could be written in a matrix form

$$\{\mathbf{F}\} + [\mathbf{A}]\{\mathbf{h}\} = \{\mathbf{0}\} \tag{10}$$

where

$$\{\mathbf{F}\} = \left.\begin{Bmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{Bmatrix}\right|_{\mathrm{t}} \tag{11}$$

$$[\mathbf{A}] = \left.\begin{bmatrix} \frac{\partial f_1}{\partial x_{\mathrm{c}}} & \frac{\partial f_1}{\partial y_{\mathrm{c}}} & \frac{\partial f_1}{\partial r_{\mathrm{c}}} \\ \frac{\partial f_2}{\partial x_{\mathrm{c}}} & \frac{\partial f_2}{\partial y_{\mathrm{c}}} & \frac{\partial f_2}{\partial r_{\mathrm{c}}} \\ \vdots & \vdots & \vdots \\ \frac{\partial f_N}{\partial x_{\mathrm{c}}} & \frac{\partial f_N}{\partial y_{\mathrm{c}}} & \frac{\partial f_N}{\partial r_{\mathrm{c}}} \end{bmatrix}\right|_{\mathrm{t}} \tag{12}$$

$$\{\mathbf{h}\} = \begin{Bmatrix} h_x \\ h_y \\ h_r \end{Bmatrix} \tag{13}$$

From Eq. (10) we can solve the linear equation set

$$[\mathbf{A}]\{\mathbf{h}\} = -\{\mathbf{F}\} \tag{14}$$

Since point (x_{t}, y_{t}, r_{t}) is only a guess and Eq (8) is just a first-order approximation of Eq. (6), so Eq. (7) may not necessarily satisfy Eq. (6). But we could take the updated Eq. (7) as a closer guess of the real (x_0, y_0, r_0). All this means that we could refine our guess of the real (x_0, y_0, r_0) by using Eq. (14) and Eq. (7) iteratively. Hopefully, the iterative process will converge to an (x_0, y_0, r_0) which is close enough to the real (x_0, y_0, r_0). To detect a convergence, a residual is defined as

$$\varepsilon = \mathrm{norm}\left(\{\mathbf{h}\}^{n+1} - \{\mathbf{h}\}^{n} \right) \tag{15}$$

where n is the current number of iteration and norm() is the L-2 norm. A convergence could be achieved with sufficiently small ε.

The above procedure includes solving the over-determined linear equation set, Eq. (14). This equation is also solved by the least-squares method. In the code of *InCylRgrSafeGurad*, it is achieved thanks to Eigen. In the actual code of *InCylRgrSafeGuard*, the above iterative process is controlled by a maximum residual limit and a maximum number of iterations. The user could specify these two parameters by *set_max_residual()* and *set_max_iters()* functions. To further enhance the stability of the iteration, a relaxation factor, λ, could be set to alter Eq. (7) into Eq. (16).

$$\begin{aligned} x_0 &= x_\mathrm{t} + \lambda h_x \\ y_0 &= y_\mathrm{t} + \lambda h_y \\ r_0 &= r_\mathrm{t} + \lambda h_r \end{aligned} \tag{16}$$

where λ should be a positive number. If the user chooses a λ to be $0 < \lambda < 1$, then the under-relaxation effect could be achieved. Otherwise, if $\lambda > 1$, we will have an over-relaxation effect. The under-relaxation is preferable than the other one. The user could set λ by *set_relax()* function.

The user should set the maximum residual, maximum number of iterations, and the relaxation factor carefully. The goal is to achieve a satisfactory solution of (x_0, y_0, r_0) as fast as possible, with small number of iterations and acceptable ε value. If the user chooses not to set those three parameters, *InCylRgrSafeGuard* will use a set of default values. The default values are 1e-4 maximum residual, 20 iterations and 0.5 relaxation factor.

In the code of *InCylRgrSafeGuard* the standard deviation is evaluated after the radius is analyzed.

The usage of *InCylRgrSafeGuard* is best illustrated by the test cases.

9 Test cases

In the current source code of LidarSafeGuard project, there is a *main.cpp* file which now serves as the file contains all the test cases. The test cases are designed to cover as many aspects of LidarSafeGuard as possible.

No.	Test case	Description
1	test_naive_situation()	Perfect circle, mask. InCylinderSafeGuard.
2	test_perfect_circle_with_noise()	Perfect circle, noise, no mask.

		InCylinderSafeGuard.
3	test_data_on_launchrig_no_mask()	Real launch rig, no mask. InCylinderSafeGuard.
4	test_data_on_launchrig_mask()	Real launch rig, mask. InCylinderSafeGuard.
5	test_data_on_launchrig_mask_regression()	Real launch rig, mask. InCylRgrSafeGuard.

The test cases are run on a system of Ubuntu 17.04 64bit, which is hosted on a Windows 10 64bit PC. The hosting software is Oracle VirtualBox. The hardware includes an Intel Core i5-2400 CPU at 3.10 GHz. The time consumed by the *verify()* function in each case is measured by routines provided by the Boost library.

The input data of all the test cases are a bunch of CSV files (if an input file is needed). These input data are shipped with the LidarSafeGuard project.

9.1 test_naive_situation()

This test case sets up an extreme case with a perfect circle as the cylinder wall. The radius is 1 m. The RPLIDAR is located at $\varepsilon = 0.9$ m in the global frame, with $\alpha = 0$ and $\beta = 0$. A mask with two angle segments is set up and registered to an *InCylinderSafeGuard* object. The input data is synthetic and is created on the fly. There is no Inf in the input data. *InCylinderSafeGuard* should give very good analyzed results on the radius and standard deviation. And it does.

9.2 test_perfect_circle_with_noise()

This test case use $\varepsilon = 0.9$ m, $\alpha = 0$, and $\beta = 0$. The radius is 1 m. A random noise with peak-to-peak fluctuation about 0.1 m is imposed on the perfect circle. No mask is applied this time. The input data is shown in Fig. 6. An *InCylinderSafeGuard* is used to do the analysis.

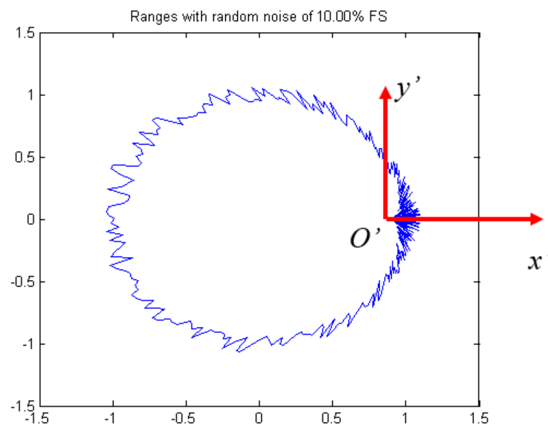

Fig. 6 A circle with noise as the cylinder wall.

9.3 test_data_on_launchrig_no_mask()

In this test, a real robot is put on a real launch rig with an actual RPLIDAR installed. The robot sits on the rig with no movements. The RPLIDAR is installed with $\varepsilon \approx 0$ m, $\alpha \approx 0$, and $\beta \approx 0$. The actual radius of the launch rig is about 0.521 m.

The range data is recorded by using the bag file functionality provided by the ROS. Then the raw data is extracted from the recorded bag file. Only one frame of data is used. There are about 360 data points in the frame of data. As shown in Fig. 7, because of the geometry of the launch rig, there are about half the data points are Infs. However, the other half of the data points are quite good. For the real configuration, the RPLIDAR has its tailing cable pointing to the launch rig. So the x-axis in Fig. 7 is pointing downwards in the real world.

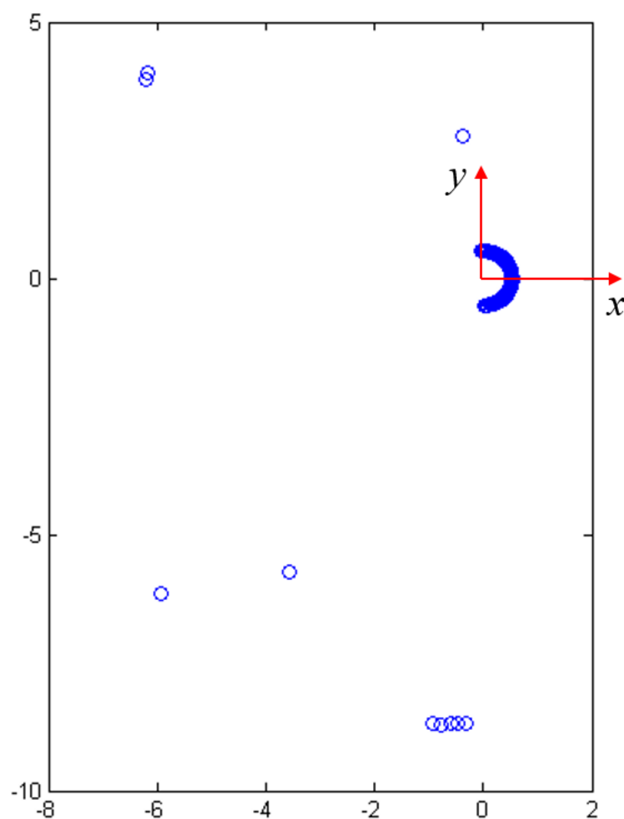

Fig. 7 Actual data from the RPLIDAR on the launch rig.

An *InCylinderSafeGuard* object with no mask is used. Since there is no mask to facilitate the analysis, *InCylinderSafeGuard* should give an unsafe flag, a wrong radius prediction, and a large standard deviation. Meanwhile, the *infRatio* is as high as 0.5. It is another sign of unsafe status.

9.4 test_data_on_launchrig_mask()

It is basically the same with test case *test_data_on_launchrig_no_mask()* with the exception that a proper mask is applied. In the real test, the space over the robot is open and the obstacles are all far away from the RPLIDAR, resulting lots of Inf values in these angle regions. A mask is placed on these regions. An *InCylinderSafeGuard* object is adopted. This time, the radius prediction is good with acceptable standard deviation. The *infRatio* should be very low since all the data points targeting the launch rig are in quite a good condition.

9.5 test_data_on_launchrig_mask_regression()

All the configurations are the same except that an *InCylRgrSafeGuard* object is used this time. This object successfully predicts the radius with good standard deviation. However, the time consumed by *InCylRgrSafeGuard* is about a magnitude higher than that of an *InCylinderSafeGuard* object. But the actual time length is about 3 ms on the test PC. So it may be acceptable on the real robot.

10 Acknowledgement

Special thanks to David Kohanbash at Carnegie Mellon University to provide me with such a good hardware and the valuable instructions on the development.

11 References

[1] Gander, W., Golub, G. H., and Strebel, R., 1994, “Least-Squares Fitting of Circles and Ellipses,” BIT Numer. Math., **34**(4), pp. 558–578.