Lidar Simulation

Horst Dumcke

Aug 14, 2023

CONTENTS

This repository contains a Lidar simulator that generates Lidar scans in different environments either based on 3D models or mazes in ASCII format.

We use these Lidar scans to test different algorithm for perception, localization and path finding.

Known Execution Errors

As generator for STL files we will use <a href=”https://build123d.readthedocs.io/en/latest/” target=”_blank”>Build123d This tool is not installed when generation the jupyter book, hence Square, Race Track and Box will show execution errors in the jupyter notebook

CHAPTER

ONE

OBJECTIVE

I was looking for a simulation environment to generate Lidar output to test different navigation and localization algorithm. Gazebo does not work for my purposes as it does not run on a Mac and I was not able to find a suitable replacement that meets m needs.

The original idea was to use a bitmap image, convert this into a point cloud and transform the cloud map to a polar coordinate system to generate a Lidar scan.

Turns out that there are always corner cases where a ray through the point cloud that represents a wall will not hit any of the points, the lases “sees through the wall”

To be able to do the calculation required for a simulated lases scan we need a line to represent the wall. A 3D mesh is a collection of triangles and there are many software tools available to generate such a 3d mesh, we will therefore use the stl file format as input.

For now we restrict our studies to two dimensional lidar scans. When dealing with a 3D STL file we will filter all the faces on the ground to have a 2D representation.

It is also straight forward to read a file with a bitmap image like pgm and convert each pixel to two triangles but this will lead to an unnecessary mount of trianles, we will only implement the import of pgm files when we have a specific need for it.

An other input file is using ASCII art as this is used to stare maze layouts for different micromouse competition. We will develop a converted for this file format.

As generator for STL files we will use Build123d To run the jupyter notebooks that uses Build123d you need an environment with Build123d installed and that will require the use of conda. However to use STL files we can work with an environment where all dependencies can be installed with pip.

CHAPTER

TWO

CREATING 3D MODELS

This section contains a few example of generated environments using different tools.

CHAPTER

THREE

SQUARE

We use Build123d to create squae with four walls around

```
from jupyter_cadquery import show, open_viewer, set_defaults
import cadquery as cq
from build123d import *
cv = open_viewer("Build123d", cad_width=770, glass=True)
set_defaults(edge_accuracy=0.0001)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 from jupyter_cadquery import show, open_viewer, set_defaults
      2 import cadquery as cq
      3 from build123d import *

ModuleNotFoundError: No module named 'jupyter_cadquery'
```

```
length = 1000
width = 551

wall_hight = 20
wall_thick = 2

with BuildPart() as p:
    Box(length, width, wall_hight)
    Box(length - wall_thick, width - wall_thick, wall_hight, mode=Mode.SUBTRACT)
```

```
assembly = Compound(children=[p.part])
```

```
assembly
```

```
assembly.export_stl('rectangle.stl')
```

```
True
```

CHAPTER

FOUR

RACE TRACK

We use Build123d to create a 3D model of a race track. We export the 3D model as a stl file.

We also export a 2D projection as svg file. We use image magic to convert the svg file to a pgn file and then to convert the png file to a pgm file. Adding some meta data will provide us with a map that we can use in Nav2

```
from jupyter_cadquery import show, open_viewer, set_defaults
import cadquery as cq
from build123d import *
cv = open_viewer("Build123d", cad_width=770, glass=True)
set_defaults(edge_accuracy=0.0001)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 from jupyter_cadquery import show, open_viewer, set_defaults
      2 import cadquery as cq
      3 from build123d import *

ModuleNotFoundError: No module named 'jupyter_cadquery'
```

```
class SideWall():

    def __init__(self, whichWall):
        ext_r = 150
        int_r = 50
        length = 1000
        width = 551

        len1 = 700
        arc1 = 180
        len2 = 150
        arc2 = -90
        len3 = 50
        arc3 = 360 - 168.2
        len3 = 50
        len4 = 58.78
        arc4 = -122.2
        len5 = 103.05
        arc5 = 360 - 159.6

        wall_hight = 20
        wall_thick = 2
```

(continues on next page)

(continued from previous page)

```
        if whichWall == 'ext':
            r1 = ext_r
            r2 = int_r
        else:
            r1 = int_r
            r2 = ext_r

        with BuildPart() as self.p:
            with BuildLine(mode=Mode.PRIVATE) as l:
                base_line = Line((0,0),(len1,0))
                rigth_curve = JernArc(start=base_line @ 1, tangent=base_line % 1,␣
↪radius=r1, arc_size=arc1)
                line2 = PolarLine(rigth_curve @ 1, len2, direction=rigth_curve % 1)
                rigth_upper_curve = JernArc(start=line2 @ 1, tangent=line2 % 1,␣
↪radius=r2, arc_size=arc2)
                line3 = PolarLine(rigth_upper_curve @ 1, len3, direction=rigth_upper_
↪curve%1)
                upper_curve = JernArc(start=line3 @ 1, tangent=line3 % 1, radius=r1,␣
↪arc_size=arc3)
                line4 = PolarLine(upper_curve @ 1, len4, direction=upper_curve%1)
                rigth_upper_curve = JernArc(start=line4 @ 1, tangent=line4%1,␣
↪radius=r2, arc_size=arc4)
                line5 = PolarLine(rigth_upper_curve @ 1, len5, direction=rigth_upper_
↪curve%1)
                left_curve = JernArc(start=line5 @ 1, tangent=line5%1, radius=r1, arc_
↪size=arc5)
            with BuildSketch(Plane.YZ) as s:
                Rectangle(wall_thick, wall_hight)
            sweep(path=l.wires()[0])
```

```
p_int = SideWall('int')
p_ext = SideWall('ext')
```

```
p_int.p.part.label = "internal wall"
p_int.p.part.location = Location((0, 100, 0))
p_ext.p.part.label = "external wall"

racetrack_assembly = Compound(label="racetrack", children=[p_int.p.part, p_ext.p.
↪part])
```

```
racetrack_assembly
```

```
100% ⁝――――――――――――――――――――――――――――――――――――――――――――――――――――――――――⁝ (2/2)  0.06s
```

```
racetrack_assembly.export_stl('racetrack.stl')
```

```
True
```

```
racetrack_assembly.export_svg('racetrack.svg', (0, 0, 1000000), (0, 1, 0),
                          svg_opts={"pixel_scale": 1, "margin_left": 0, "margin_top":␣
↪0,"show_axes": False, "show_hidden": False})
```

(continues on next page)

(continued from previous page)

```python
import xml.etree.ElementTree as ET
tree = ET.parse('racetrack.svg')
root = tree.getroot()
dim = root.attrib
```

```
!convert racetrack.svg racetrack.png
```

```
!convert racetrack.png -flatten racetrack.pgm
```

```python
metadata = {
    "image": "racetrack.pgm",
    "resolution": 0.01,
    "origin": [0.0, 0.0, 0.0],
    "occupied_thresh": 0.65,
    "free_thresh": 0.196,
    "negate": 0
}
```

```python
metadata
```

```
{'image': 'racetrack.pgm',
 'resolution': 0.01,
 'origin': [0.0, 0.0, 0.0],
 'occupied_thresh': 0.65,
 'free_thresh': 0.196,
 'negate': 0}
```

```python
import yaml
with open("racetrack.yaml", "w") as fh:
    yaml.dump(metadata, fh)
```

CHAPTER

FIVE

BOX

We build a simple box that we can use as obstacle

```
from jupyter_cadquery import show, open_viewer, set_defaults
import cadquery as cq
from build123d import *
cv = open_viewer("Build123d", cad_width=770, glass=True)
set_defaults(edge_accuracy=0.0001)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 from jupyter_cadquery import show, open_viewer, set_defaults
      2 import cadquery as cq
      3 from build123d import *

ModuleNotFoundError: No module named 'jupyter_cadquery'
```

```
length = 30
width = 10
hight = 20

with BuildPart() as p:
    Box(length, width, hight)
```

```
assembly = Compound(children=[p.part])
```

```
assembly
```

```
assembly.export_stl('box.stl')
```

```
True
```

CHAPTER

SIX

MAZE GENERATION

We will convert ASCII files as used in mazefiles into a suitable format

```
%matplotlib inline
from stl import mesh
import numpy as np
import matplotlib.pyplot as plt
```

```
def read_maze(file_name):
    WH_BIT = 1
    WV_BIT = 2

    with open(file_name, 'r') as fh:
        txt = fh.read()

    txt = txt.splitlines()
    post_char = txt[0][0]

    wall_horizontal = txt[::2]
    wall_horizontal = [
        [
            1 if wall == '---' else 0
            for wall in row.lstrip(post_char).split(post_char)
        ]
        for row in wall_horizontal
    ]
    wall_horizontal = np.array(wall_horizontal).astype('uint8')

    wall_vertical = txt[1::2]
    wall_vertical = [
        [1 if wall == '|' else 0 for wall in column[::4]]
        for column in wall_vertical
    ]
    wall_vertical.append([0] * 17)
    wall_vertical = np.array(wall_vertical).astype('uint8')

    wall_horizontal *= WH_BIT
    wall_vertical *= WV_BIT

    return wall_horizontal + wall_vertical
```

```
m = read_maze('maze_in/maze.txt')
```

```
for line in m:
    l1 = ''
    l2 = ''
    for l in line:
        if l & 1:
            l1 += 'o---'
        else:
            l1 += 'o   '
        if l & 2:
            l2 += '|   '
        else:
            l2 += '    '
    print(l1)
    print(l2)
```

```
o---o---o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
|                                               |               |
o   o   o   o---o   o---o   o---o   o---o---o   o---o   o   o   o
|   |   |   |       |           |       |           |   |   |   |
o   o   o   o   o---o   o---o---o---o   o   o   o   o   o---o   o
|   |   |       |               |   |   |   |   |               |
o   o   o   o   o   o---o   o   o   o   o   o   o   o   o---o   o
|           |       |       |   |           |   |   |       |   |
o---o   o---o   o---o   o   o   o   o   o---o   o   o   o   o   o
|   |   |               |   |   |   |   |   |   |   |   |       |
o   o   o   o---o---o---o   o   o   o   o   o   o   o   o   o   o
|   |       |               |   |   |       |   |           |   |
o   o   o   o   o---o---o---o   o---o   o   o   o---o   o---o   o
|       |                   |       |       |                   |
o   o---o---o---o---o   o---o---o---o   o---o---o---o   o---o   o
|   |                       |       |       |                   |
o   o---o   o---o---o   o   o   o---o   o   o   o---o   o---o   o
|                   |   |   |       |       |   |           |   |
o   o---o   o---o   o   o   o---o   o---o---o   o---o   o---o   o
|   |           |   |   |       |           |   |           |   |
o---o   o   o---o   o   o   o   o   o---o   o   o---o   o---o   o
|       |   |       |   |   |   |       |   |   |           |   |
o   o---o   o---o   o   o---o   o---o   o   o   o---o   o---o   o
|       |           |           |   |       |       |   |       |
o---o   o   o   o   o---o   o---o   o---o   o   o---o   o---o   o
|       |   |   |           |           |                   |   |
o   o---o   o   o   o---o---o   o   o   o---o---o---o---o---o   o
|   |       |   |   |       |   |   |           |   |   |   |   |
o   o   o   o   o   o   o   o   o   o---o---o   o   o   o   o   o
|   |   |           |   |   |               |       |       |   |
o   o   o   o   o---o   o   o---o---o---o   o   o   o   o   o   o
|   |       |           |                   |   |       |       |
o---o---o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
```

```
def find_walls(maze):
    index_y = 0
    is_wall_y = [False] * maze.shape[0]
    wall_y_start = [None] * maze.shape[0]
    walls = []
```

(continues on next page)

(continued from previous page)

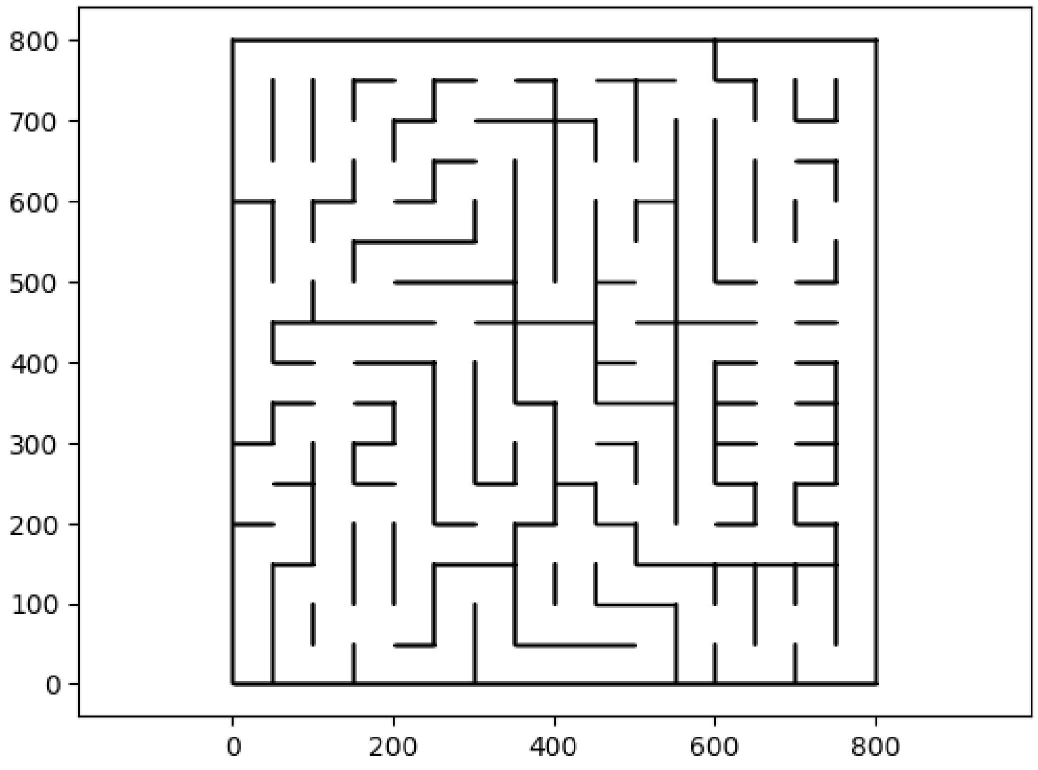
```
    for line in maze:
        is_wall_x = False
        index_x = 0
        for l in line:
            if l & 1:
                if not is_wall_x:
                    wall_x_start = [index_x, index_y]
                    is_wall_x = True
            else:
                if is_wall_x:
                    wall_x_end = [index_x, index_y]
                    walls.append([wall_x_start, wall_x_end])
                    is_wall_x = False
            if l & 2:
                if not is_wall_y[index_x]:
                    wall_y_start[index_x] = [index_x, index_y]
                    is_wall_y[index_x] = True
            else:
                if is_wall_y[index_x]:
                    wall_y_end = [index_x, index_y]
                    walls.append([wall_y_start[index_x], wall_y_end])
                    is_wall_y[index_x] = False
            index_x += 1
        index_y += 1
    return np.array(walls)
```

```
walls = find_walls(m)
```

```
data = np.zeros(2*walls.shape[0], dtype=mesh.Mesh.dtype)
index = 0
for wall in walls:
    y_dim = 16
    space = 50
    if wall[0, 0] == wall[1, 0]:
        p0 = [space * wall[0, 0], space * (y_dim - wall[0, 1]), 0]
        p1 = [space * wall[1, 0], space * (y_dim - wall[1, 1]), 0]
        p2 = [space * wall[0, 0], space * (y_dim - wall[0, 1]) + 2, 0]
        p3 = [space * wall[1, 0], space * (y_dim - wall[1, 1]) + 2, 0]
    else:
        p0 = [space * wall[0, 0], space * (y_dim - wall[0, 1]), 0]
        p1 = [space * wall[1, 0], space * (y_dim - wall[1, 1]), 0]
        p2 = [space * wall[0, 0] + 2, space * (y_dim - wall[0, 1]), 0]
        p3 = [space * wall[1, 0] + 2, space * (y_dim - wall[1, 1]), 0]
    data['vectors'][index] = np.array([p0, p1, p2])
    data['vectors'][index + 1] = np.array([p1, p2, p3])
    index += 2

env = mesh.Mesh(data, remove_empty_areas=False)
env.update_units()
```

```
triangles = []
for t in env.vectors:
    triangles.append(t)
```

```
fig, ax = plt.subplots(1, 1)
ax.axis('equal')
for t in triangles:
    ax.fill(t[:, 0],t[:, 1],fill=False)
```



```
data = np.zeros(12*walls.shape[0], dtype=mesh.Mesh.dtype)
index = 0
for wall in walls:
    y_dim = 16
    space = 50
    if wall[0, 0] == wall[1, 0]:
        p0 = [space * wall[0, 0], space * (y_dim - wall[0, 1]), 0]
        p1 = [space * wall[1, 0], space * (y_dim - wall[1, 1]), 0]
        p2 = [space * wall[0, 0] + 2, space * (y_dim - wall[0, 1]), 0]
        p3 = [space * wall[1, 0] + 2, space * (y_dim - wall[1, 1]), 0]
        p4 = [space * wall[0, 0], space * (y_dim - wall[0, 1]), 20]
        p5 = [space * wall[1, 0], space * (y_dim - wall[1, 1]), 20]
        p6 = [space * wall[0, 0] + 2, space * (y_dim - wall[0, 1]), 20]
        p7 = [space * wall[1, 0] + 2, space * (y_dim - wall[1, 1]), 20]
    else:
        p0 = [space * wall[0, 0], space * (y_dim - wall[0, 1]), 0]
        p1 = [space * wall[1, 0], space * (y_dim - wall[1, 1]), 0]
        p2 = [space * wall[0, 0], space * (y_dim - wall[0, 1]) + 2, 0]
        p3 = [space * wall[1, 0], space * (y_dim - wall[1, 1]) + 2, 0]
        p4 = [space * wall[0, 0], space * (y_dim - wall[0, 1]), 20]
        p5 = [space * wall[1, 0], space * (y_dim - wall[1, 1]), 20]
        p6 = [space * wall[0, 0], space * (y_dim - wall[0, 1]) + 2, 20]
        p7 = [space * wall[1, 0], space * (y_dim - wall[1, 1]) + 2, 20]
```

(continues on next page)

(continued from previous page)

```
    data['vectors'][index] = np.array([p1, p0, p2])
    data['vectors'][index + 1] = np.array([p1, p2, p3])
    data['vectors'][index + 2] = np.array([p4, p5, p6])
    data['vectors'][index + 3] = np.array([p6, p5, p7])
    data['vectors'][index + 4] = np.array([p0, p1, p4])
    data['vectors'][index + 5] = np.array([p4, p1, p5])
    data['vectors'][index + 6] = np.array([p3, p2, p6])
    data['vectors'][index + 7] = np.array([p3, p6, p7])
    data['vectors'][index + 8] = np.array([p2, p0, p4])
    data['vectors'][index + 9] = np.array([p2, p4, p6])
    data['vectors'][index + 10] = np.array([p1, p3, p5])
    data['vectors'][index + 11] = np.array([p5, p3, p7])
    index += 12

env = mesh.Mesh(data, remove_empty_areas=False)
env.update_units()
```

```
env.save('maze.stl')
```

CHAPTER

SEVEN

SHOW ALL MODELS

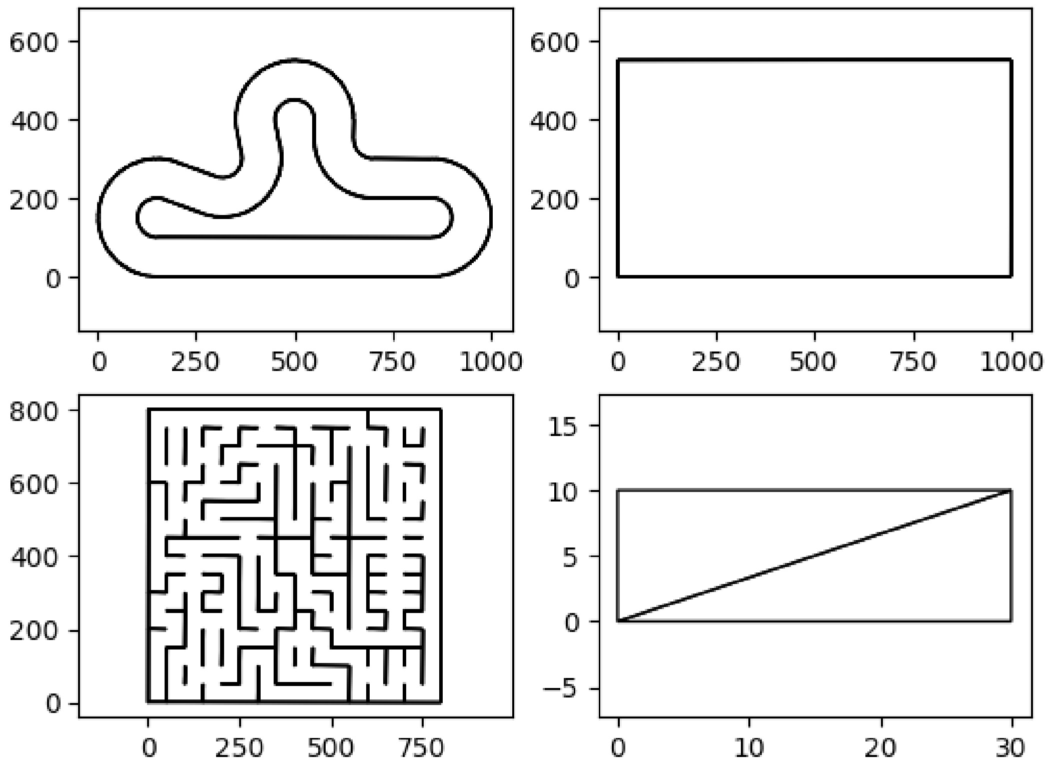
This requires that the python module LidarSim is installed

```
from LidarSim.lidar_sim import LidarSimulator
import matplotlib.pyplot as plt
%matplotlib inline
```

```
import glob
all_models = glob.glob("*.stl")
```

```
columns = 2
rows = int(len(all_models) / columns)
if len(all_models) % columns:
    rows += 1
```

```
fig, axs = plt.subplots(rows, columns)
for i in range(len(all_models)):
    row_index = int(i / columns)
    column_index = i % columns
    lidar = LidarSimulator(all_models[i])
    triangles = lidar.get_map_triangles()
    axs[row_index, column_index].axis('equal')
    for t in triangles:
        axs[row_index, column_index].fill(t[:, 0],t[:, 1],fill=False)
```



600
400
200
0
0
250
500
750
1000
600
400
200
0
0
250
500
750
1000
800
600
400
200
0
0
250
500
750
15
10
5
0
−5
0
10
20
30

CHAPTER

EIGHT

SIMULATING A LIDAR

We read our stl file using numpy-stl.

We select all triangles in the xy plane to get a 2D projection

We then place the robot with the lidar on the map using xy coordinates and yaw as the orientation. We transform the points of the map to the lidar coordinate system and then convert the carthesian coordinate system to a polar coordinate system. We use a simplified robot that consists of only a point

We sweep a full circle around the Lidar in segments which size depends on the Lidar resolution and we retain the closest point to construct our lidar scan.

```
%matplotlib inline
import numpy as np
from stl import mesh
import matplotlib.pyplot as plt
```

```
min_range = 2.0
max_range = 12000.0
resolution = 1 # in degrees
```

```
#simulated_environment = 'racetrack'
simulated_environment = 'square'
#simulated_environment = 'test'
```

```
if simulated_environment == 'test':
    data = np.zeros(3, dtype=mesh.Mesh.dtype)
    data['vectors'][0] = np.array([[100., 500., 0],
                                   [100., 100., 0],
                                   [  0.,   0., 0]])
    data['vectors'][1] = np.array([[ 900., 100., 0],
                                   [1000.,   0., 0],
                                   [   0.,   0., 0]])
    data['vectors'][2] = np.array([[1000., 500., 0],
                                   [1000., 200., 0],
                                   [ 100., 500., 0]])
    env = mesh.Mesh(data, remove_empty_areas=False)
    env.update_units()
else:
    env = mesh.Mesh.from_file('%s.stl' % simulated_environment)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[4], line 15
     13     env.update_units()
     14 else:
---> 15     env = mesh.Mesh.from_file('%s.stl' % simulated_environment)

File ~/.virtualenvs/lidar_sim/lib/python3.11/site-packages/stl/stl.py:389, in␣
 ↪BaseStl.from_file(cls, filename, calculate_normals, fh, mode, speedups, **kwargs)
    385     name, data = cls.load(
    386         fh, mode=mode, speedups=speedups
    387     )
    388 else:
--> 389     with open(filename, 'rb') as fh:
    390         name, data = cls.load(
    391             fh, mode=mode, speedups=speedups
    392         )
    394 return cls(
    395     data, calculate_normals, name=name,
    396     speedups=speedups, **kwargs
    397 )

FileNotFoundError: [Errno 2] No such file or directory: 'square.stl'
```

8.1 Adjust coordinate system

```
env.x = env.x - env.x.min()
env.y = env.y - env.y.min()
```

8.2 Select bottom Face

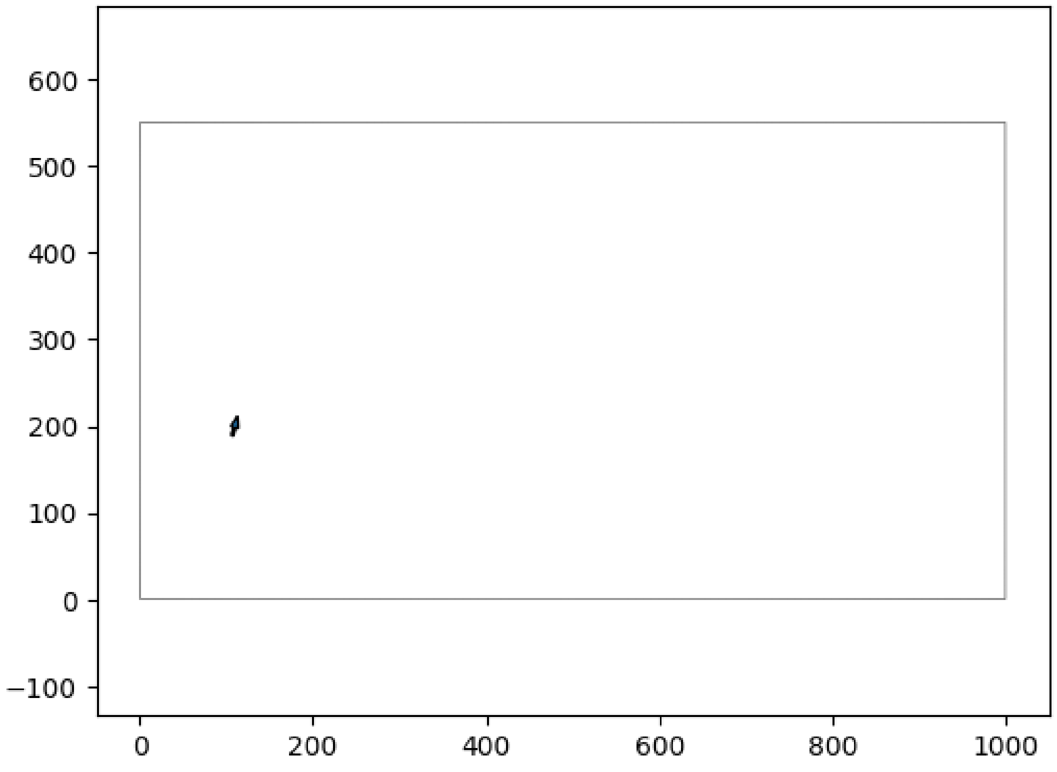
```
subset = env.vectors[(env.normals[:, 0] == 0.0) & (env.normals[:, 1] == 0.0) & (env.
 ↪normals[:, 2] < 0.0)]
```

8.3 Set initial Pose

```
pose = [900, 50, np.pi]
pose = [500, 50, np.radians(-45)]
#pose = [500, 50, 0]
#pose = [850, 52, 3.0892327760299634]
#pose = [753, 77, 2.792526803190927]
#pose = [ 166.648113,  213.283612,   0.20943951]
#pose = [ 215.555493,  223.679196,   0.20943951]
#pose = [ 260.735529,  234.518542,   0.2268928 ]
#pose = [107.199835, 189.184783, 1.30899694]
pose = [107, 189, 1.30899694]
```

```
from matplotlib.patches import Polygon

fig,ax = plt.subplots()
for y in subset[:,:, 0:2]:
    p = Polygon(y, facecolor = 'k')
    ax.add_patch(p)
ax.axis('equal')
plt.arrow(pose[0], pose[1], 10 * np.cos(pose[2]), 10 * np.sin(pose[2]), width=3.0)
```

```
<matplotlib.patches.FancyArrow at 0x1140a6bd0>
```



8.4 Transform to Lidar Frame

```
env.x = env.x - pose[0]
env.y = env.y - pose[1]
env.rotate([0.0, 0.0, 1.0], pose[2])
```

```
subset = env.vectors[(env.normals[:, 0] == 0.0) & (env.normals[:, 1] == 0.0) & (env.
↪normals[:, 2] < 0.0)]
```

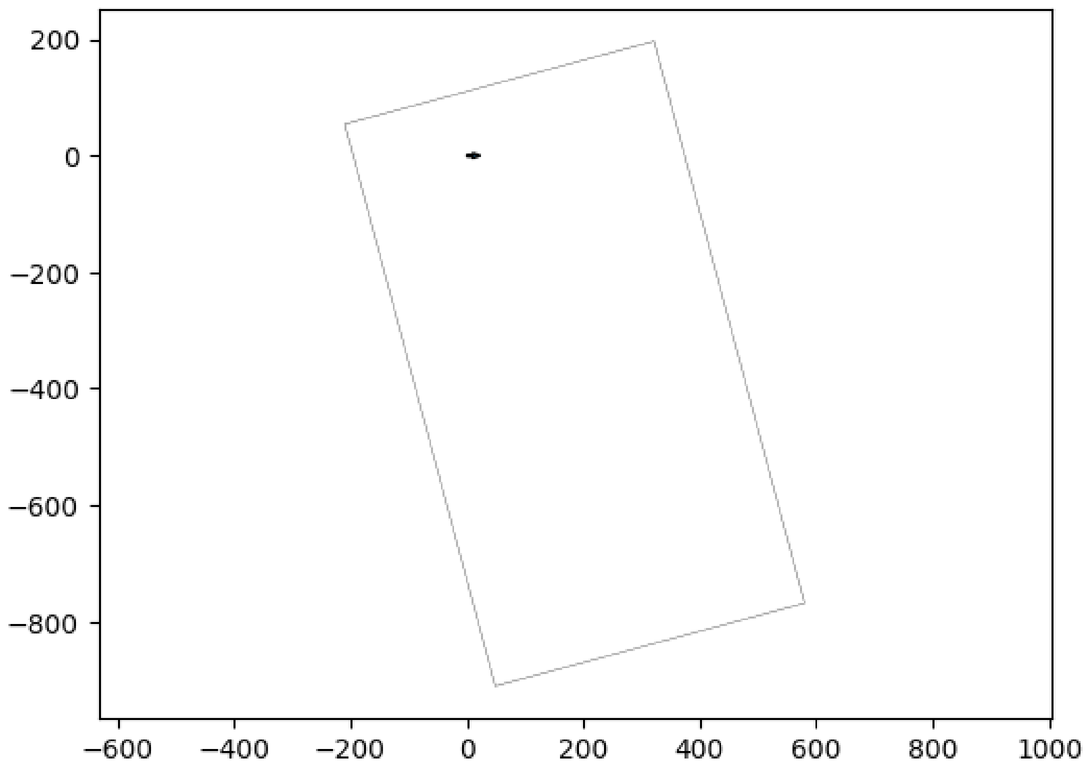
```
fig,ax = plt.subplots()
for y in subset[:,:, 0:2]:
    p = Polygon(y, facecolor = 'k')
    ax.add_patch(p)
```

(continues on next page)

(continued from previous page)

```
ax.axis('equal')
plt.arrow(0, 0, 10, 0, width=3.0)
```

```
<matplotlib.patches.FancyArrow at 0x114323350>
```



8.5 Convert to polar coordinate system

```
import cmath
def cart2polC(xyz):
    x, y, z = xyz
    return(cmath.polar(complex(x, y))) # rho, phi
def convert_array(arr):
    theta = []
    r = []
    for x in arr:
        rho, phi = cart2polC(x)
        theta.append(phi)
        r.append(rho)
    return theta, r
```

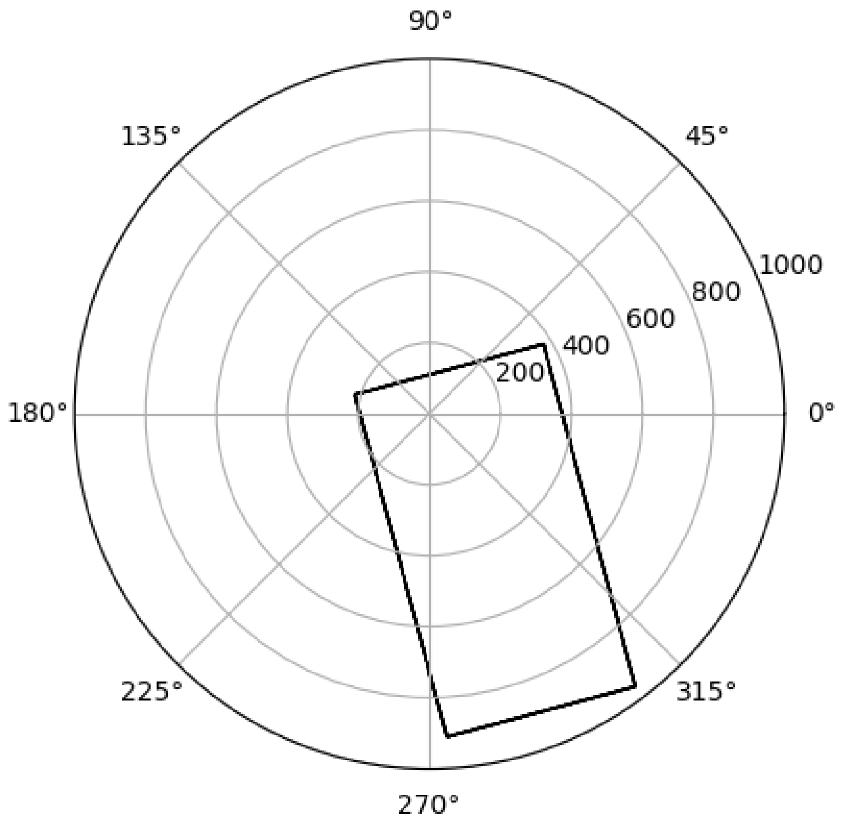
```
triangles = []
for t in subset[:,:, :]:
    a = np.array(convert_array(t)).transpose()
```

(continues on next page)

(continued from previous page)

```
    triangles.append(a[a[:, 0].argsort()])
triangles = np.array(triangles)
```

```
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
for t in triangles:
    ax.fill(t[:, 0],t[:, 1],fill=False)
```



8.6 Filter closest points

8.6.1 Lines in poloar coordinates

A line in carthesian coordonate system is described as

$$y = m * x + b$$

We make the following substitution:

$$x = r * cos(\theta)$$

$$y = r * sin(\theta)$$

Given two points in polar coordinates

$$P_1 = (\theta_1, r_1)$$

$$P_2 = (\theta_2, r_2)$$

we can calculate m and b. Finally we can calculate the intersection of the line connecting P_1 and P_2 with the line from the origin and an agle of θ

```
import sympy as sp
from sympy.abc import theta
```

```
x,y = sp.symbols("x,y")
m,b,r,r1, r2 = sp.symbols("m,b,r,r1,r2", real=True)
theta_1, theta_2 = sp.symbols('theta_1,theta_2')
```

```
expr = y - m*x - b
expr = expr.subs(x, r * sp.cos(theta))
expr = expr.subs(y, r * sp.sin(theta))
expr
```

$$-b - mr\cos(\theta) + r\sin(\theta)$$

```
p1 = expr.subs(r, r1).subs(theta, theta_1)
p2 = expr.subs(r, r2).subs(theta, theta_2)
```

```
p2.subs(b, ⍰1 * sp.sin(theta_1)) - m * r1 * sp.cos(theta_1)
```

$$-mr_1\cos(\theta_1) - mr_2\cos(\theta_2) - r_1\sin(\theta_1) + r_2\sin(\theta_2)$$

```
def get_distance(p1, p2, theta):
    theta_plus = theta
    if(p1[0] > np.pi) or (p2[0] > np.pi):
        # tringles are transformed in filter_triangles
        if theta < 0:
            theta_plus = theta + 2 * np.pi
    p = np.array([p1[0], p2[0]])
    if not (p.min() <= theta_plus) & (theta_plus <= p.max()):
        return max_range + 1.0
    r_s_1 = np.sin(p1[0]) * p1[1]
    r_c_1 = np.cos(p1[0]) * p1[1]
    r_s_2 = np.sin(p2[0]) * p2[1]
    r_c_2 = np.cos(p2[0]) * p2[1]
    m = (r_s_2 - r_s_1) / (r_c_2 - r_c_1)
    b = r_s_1 - m * r_c_1
    dist = b / (np.sin(theta_plus) - m * np.cos(theta_plus))
    return dist
```

8.7 Select Triangles hit by Ray

Must have vertices on both sides of the ray

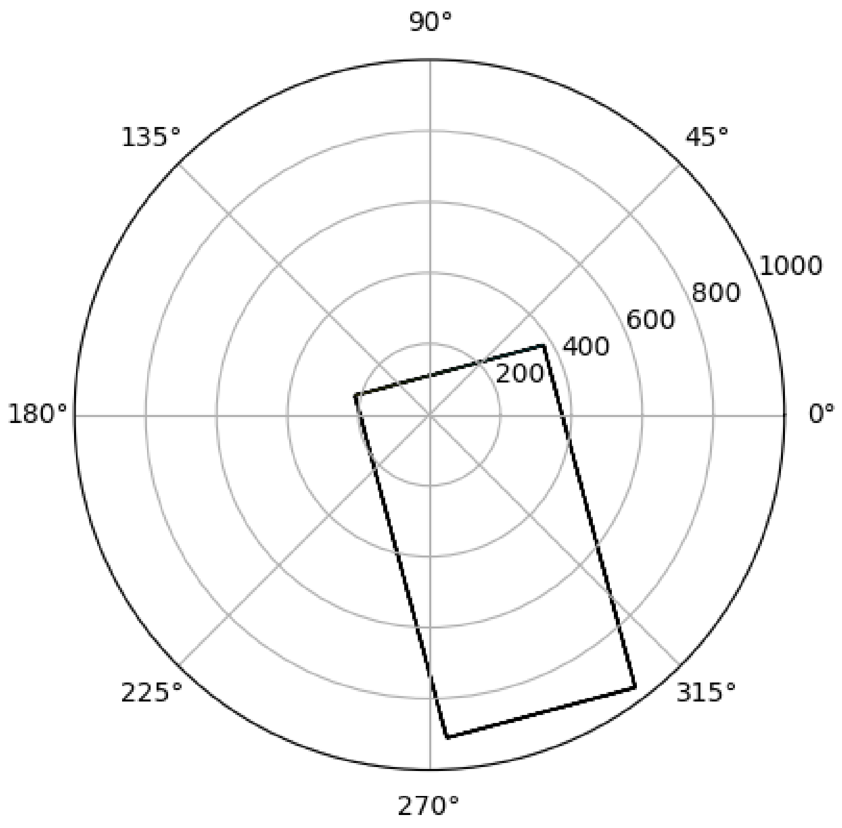
Handle case where triangle is on both sides of the x axis

Handle case where triangle crosses +/- np.pi line

```
def filter_triangles(triangles, theta):
    special_cases = triangles[np.any(triangles >= 0, axis=1)[:,0] & np.any(triangles
↪<0, axis=1)[:, 0]]
    other_cases = triangles[np.invert(np.any(triangles >= 0, axis=1)[:,0] & np.
↪any(triangles <0, axis=1)[:, 0]))]
    # verticies on both sides
    triangles_hit = other_cases[np.any(other_cases >= theta, axis=1)[:,0] & np.
↪any(other_cases <= theta, axis=1)[:, 0]]
    # handle special cases:
    sc = []
    for t in special_cases:
        if ((t[:, 0].max() - t[:, 0].min()) < np.pi):
            # not so special after all
            if (theta <= t[:, 0].max()) & (t[:, 0].min() <= theta):
                sc.append(t)
        else:
            for e in t:
                if e[0] < 0:
                    e[0] += 2* np.pi
            if (theta + 2* np.pi <= t[:, 0].max()) & (t[:, 0].min() <= theta + 2* np.
↪pi):
                sc.append(t)
            if (theta <= t[:, 0].max()) & (t[:, 0].min() <= theta):
                sc.append(t)

    result = []
    for t in triangles_hit:
        result.append(t)
    for t in sc:
        result.append(t)
    return np.array(result)
```

```
#theta = np.radians(0) # OK
theta = np.radians(90) # OK
#theta = np.radians(-90) # OK
#theta = np.radians(-80) # OK
#theta = np.radians(180) # OK
#theta = np.radians(-135) # OK

triangles_hit = filter_triangles(triangles, theta)
```

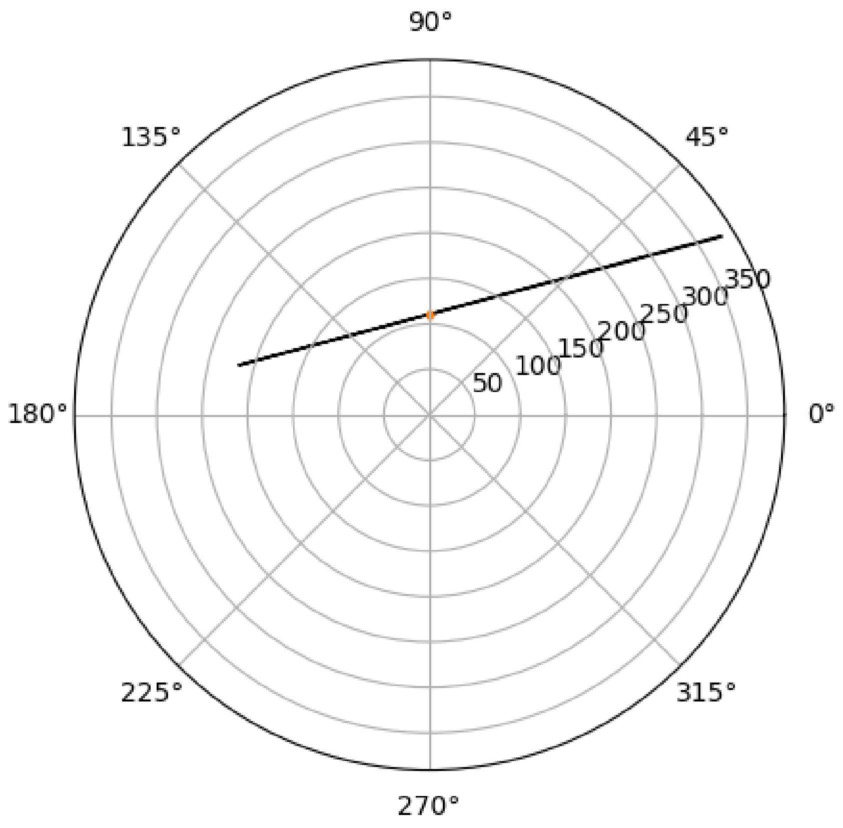
```
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
for t in triangles:
    ax.fill(t[:, 0],t[:, 1],fill=False)
for t in triangles_hit:
    ax.fill(t[:, 0],t[:, 1],fill=True)
```



```
intersection_points = np.zeros((2, 2))
intersection_points[:, 0] = theta
points = []
dist = get_distance(t[0], t[1], theta)
if dist < max_range:
    points.append(dist)
dist = get_distance(t[0], t[2], theta)
if dist < max_range:
    points.append(dist)
dist = get_distance(t[1], t[2], theta)
if dist < max_range:
    points.append(dist)
if len(points) > 0:
     intersection_points[0, 1] = points[0]
     intersection_points[1, 1] = points[1]
else:
     intersection_points[:, 1] = max_range + 1.0
     t = np.zeros((3, 2))
```

```
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
ax.fill(t[:, 0],t[:, 1],fill=False)
ax.scatter(intersection_points[:, 0], intersection_points[:, 1], s=4.0)
```

```
<matplotlib.collections.PathCollection at 0x127a1d790>
```



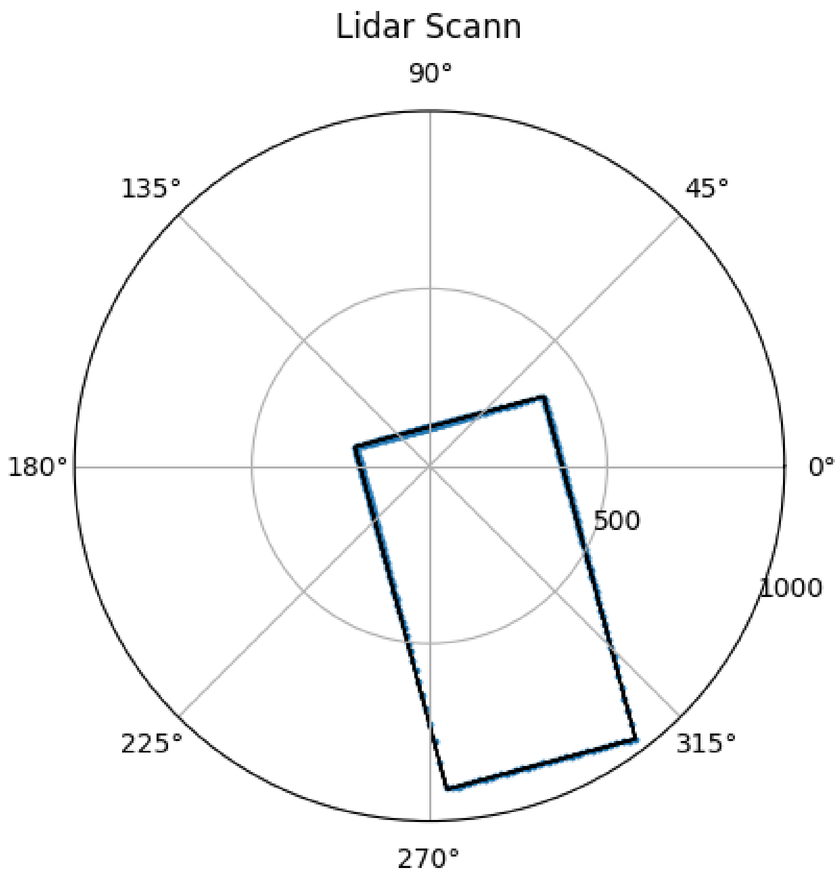
```
def lidar_filter(triangles):
    scan = []
    samples = np.arange(-np.pi, np.pi, np.radians(resolution))
    for sample in samples:
        #start with out of range
        dist = max_range + 1.0
        # select all triangles hit by the ray
        triangles_hit = filter_triangles(triangles, sample)
        for t in triangles_hit:
            dist_t = np.empty(3)
            dist_t[0] = get_distance(t[0], t[1], sample)
            dist_t[1] = get_distance(t[0], t[2], sample)
            dist_t[2] = get_distance(t[1], t[2], sample)
            dist = min(dist_t.min(), dist)
        scan.append(dist)
        if dist > max_range:
            scan[-1] = None
        if dist < min_range:
            scan[-1] = None
    return np.roll(np.array(scan), int(np.pi / np.radians(resolution)))
```

```
lidar_scan = lidar_filter(triangles)
```

```
plot_scan = np.stack((np.arange(0, 2 * np.pi, np.radians(resolution)), lidar_scan),␣
↪axis=1)
plot_scan = plot_scan[plot_scan[:, 1] != np.array(None)]
```

```
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
ax.scatter(plot_scan[:, 0], plot_scan[:, 1], s=3.0)
for t in triangles:
    ax.fill(t[:, 0],t[:, 1],fill=False)
ax.set_rmax(1000)
ax.set_rticks([500, 1000])  # Less radial ticks
ax.set_rlabel_position(-22.5)  # Move radial labels away from plotted line
ax.grid(True)

ax.set_title("Lidar Scann", va='bottom')
```

```
Text(0.5, 1.0, 'Lidar Scann')
```



```
from LidarSim.lidar_sim import LidarSimulator
```

```
test_lidar = LidarSimulator("racetrack.stl")
point = [107, 189]
yaw = np.radians(45)
plot_scan = test_lidar.get_lidar_points(point[0], point[1], yaw)
triangles = test_lidar.get_env_triangles(point[0], point[1], yaw)
```

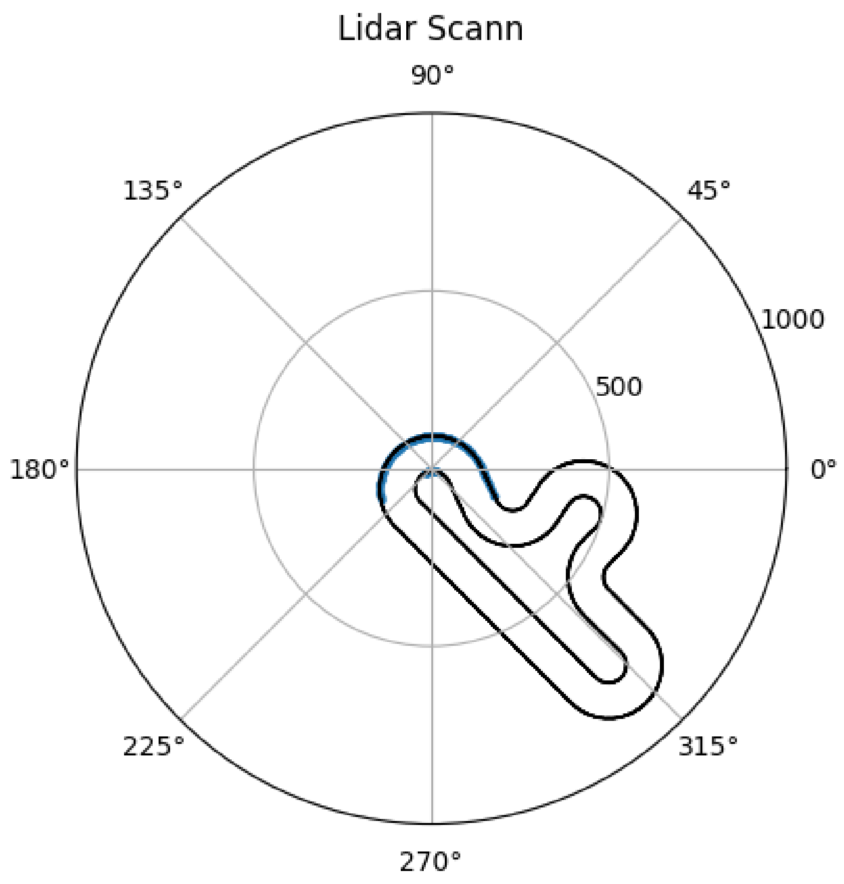
```
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
ax.scatter(plot_scan[:, 0], plot_scan[:, 1], s=3.0)
```

(continues on next page)

(continued from previous page)

```
for t in triangles:
    ax.fill(t[:, 0],t[:, 1],fill=False)
ax.set_rmax(1000)
ax.set_rticks([500, 1000])  # Less radial ticks
ax.grid(True)

ax.set_title("Lidar Scann", va='bottom')
```

```
Text(0.5, 1.0, 'Lidar Scann')
```



```
test_lidar = LidarSimulator("maze.stl")
point = [25, 25]
yaw = np.radians(90)
plot_scan = test_lidar.get_lidar_points(point[0], point[1], yaw)
triangles = test_lidar.get_env_triangles(point[0], point[1], yaw)
```

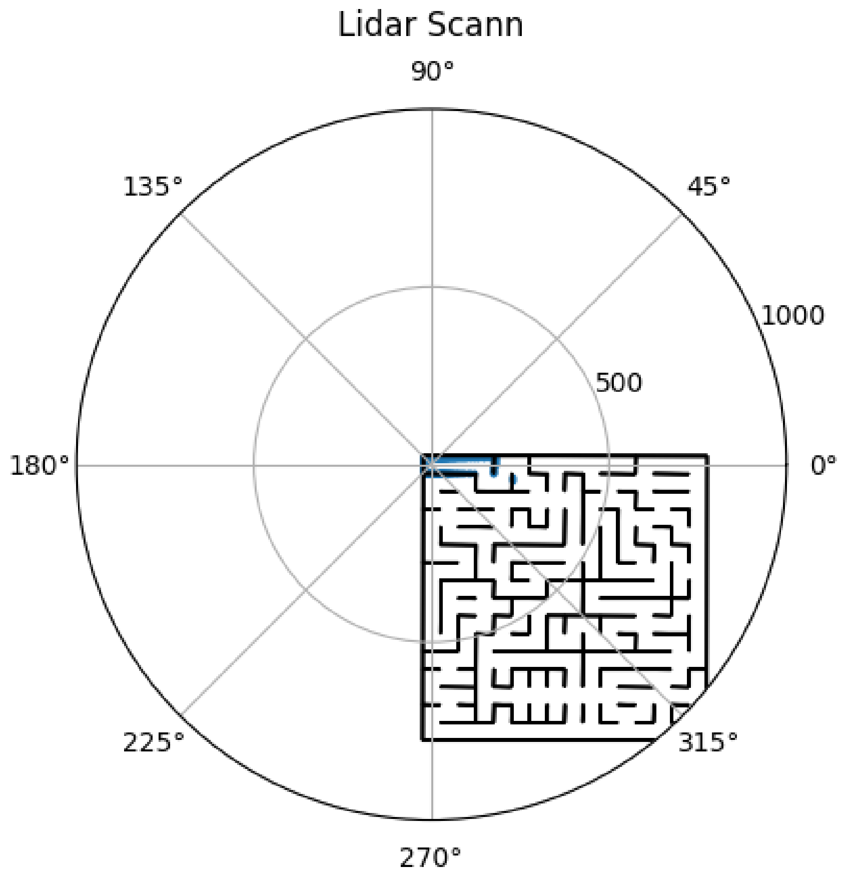
```
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
ax.scatter(plot_scan[:, 0], plot_scan[:, 1], s=3.0)
for t in triangles:
    ax.fill(t[:, 0],t[:, 1],fill=False)
ax.set_rmax(1000)
ax.set_rticks([500, 1000])  # Less radial ticks
ax.grid(True)
```

(continues on next page)

(continued from previous page)

```
ax.set_title("Lidar Scann", va='bottom')
```

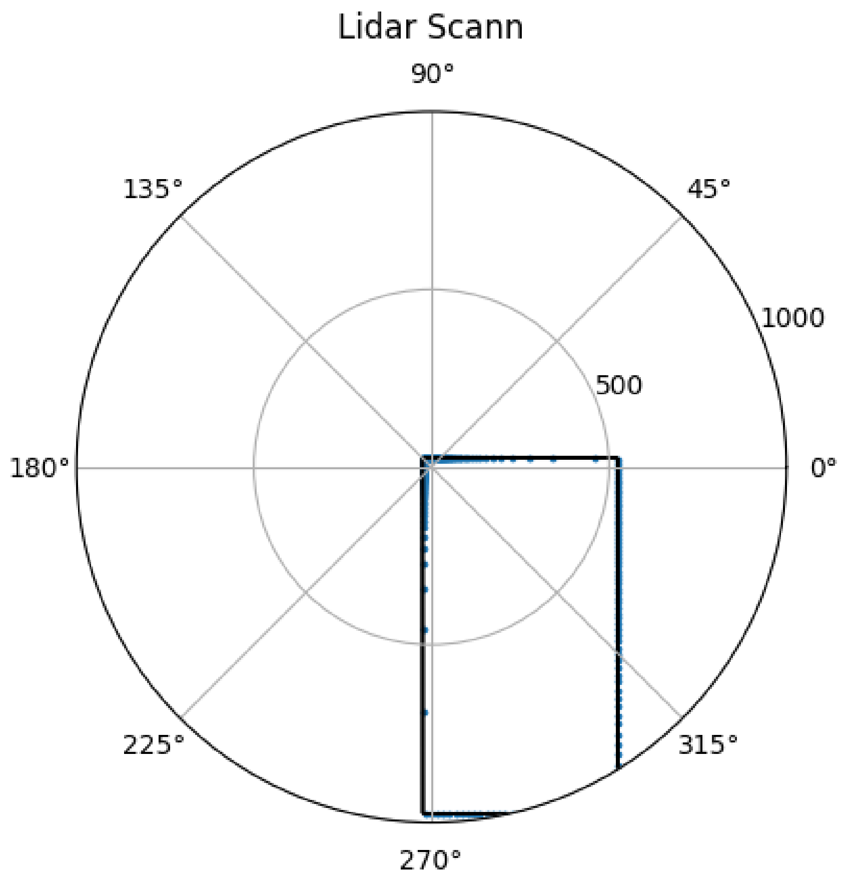
```
Text(0.5, 1.0, 'Lidar Scann')
```



```
test_lidar = LidarSimulator("rectangle.stl")
point = [25, 25]
yaw = np.radians(90)
plot_scan = test_lidar.get_lidar_points(point[0], point[1], yaw)
triangles = test_lidar.get_env_triangles(point[0], point[1], yaw)
```

```
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
ax.scatter(plot_scan[:, 0], plot_scan[:, 1], s=3.0)
for t in triangles:
    ax.fill(t[:, 0],t[:, 1],fill=False)
ax.set_rmax(1000)
ax.set_rticks([500, 1000])  # Less radial ticks
ax.grid(True)

ax.set_title("Lidar Scann", va='bottom')
```

```
Text(0.5, 1.0, 'Lidar Scann')
```


Lidar Scann
90°
45°
0°
315°
270°
225°
180°
135°
500
1000

CHAPTER

NINE

ADDING STATISTICAL ERRORS

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

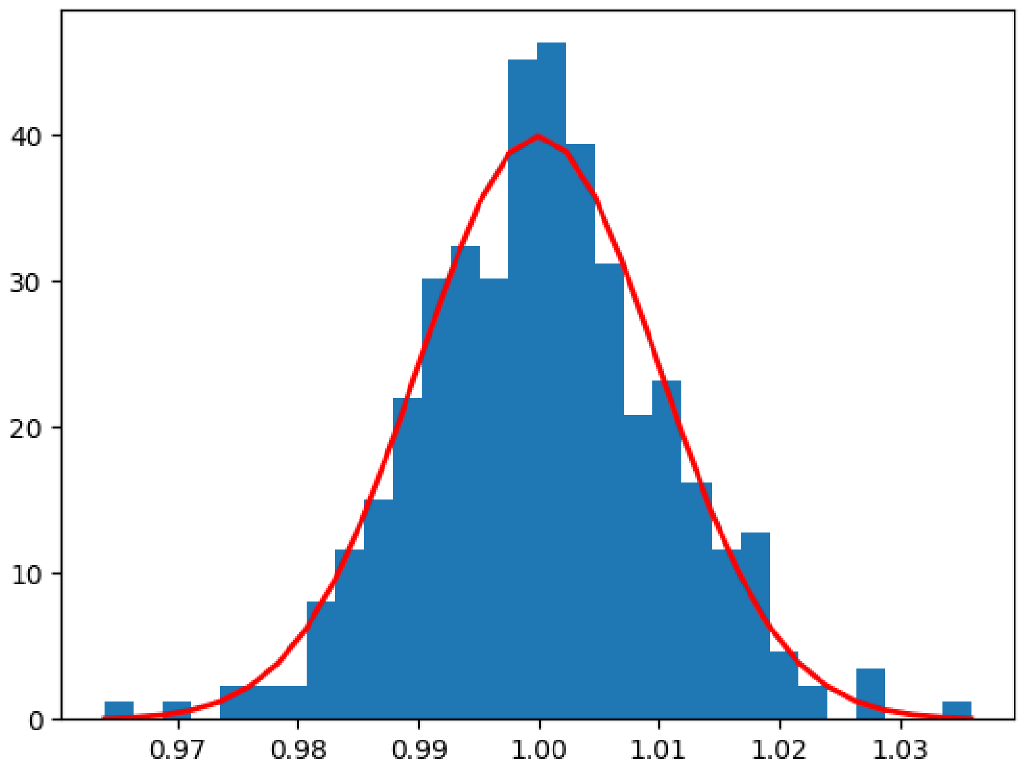
```
from LidarSim.lidar_sim import LidarSimulator
```

```
test_lidar = LidarSimulator("rectangle.stl")
point = [25, 25]
yaw = np.radians(90)
plot_scan = test_lidar.get_lidar_points(point[0], point[1], yaw)
triangles = test_lidar.get_env_triangles(point[0], point[1], yaw)
```

```
mu, sigma = 1.0, 0.01 # mean and standard deviation
s = np.random.normal(mu, sigma, plot_scan.shape[0])
```

```
count, bins, ignored = plt.hist(s, 30, density=True)
plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
               np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
         linewidth=2, color='r')
```

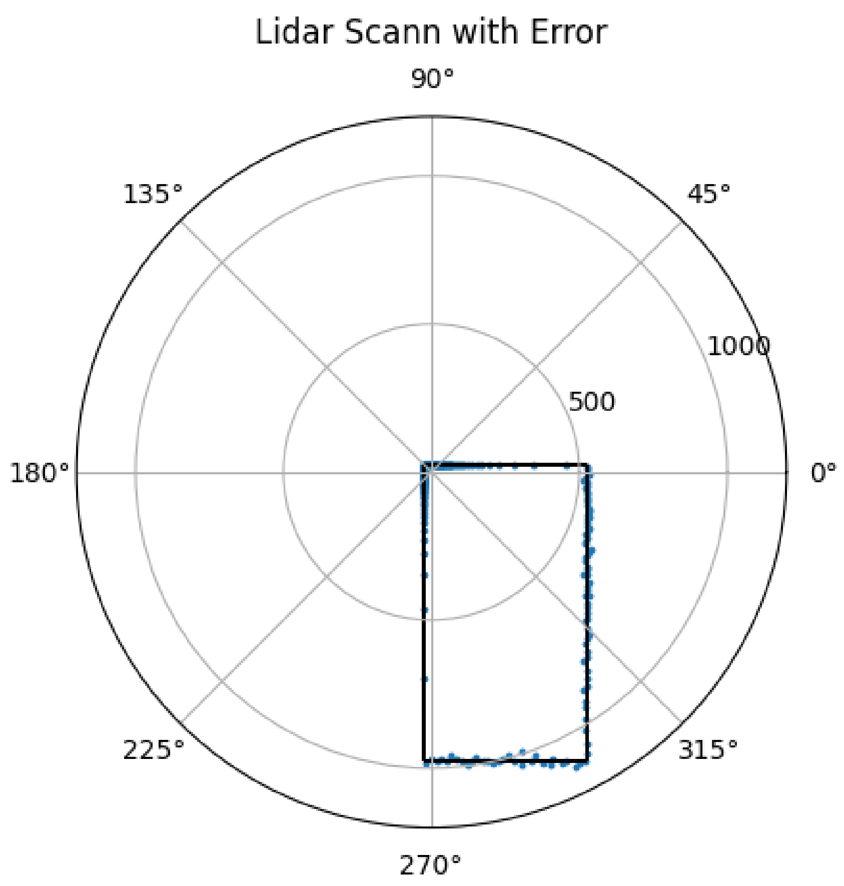
```
[<matplotlib.lines.Line2D at 0x109942050>]
```



```
plot_scan[:,1] = plot_scan[:,1] * s
```

```
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
ax.scatter(plot_scan[:, 0], plot_scan[:, 1], s=3.0)
for t in triangles:
    ax.fill(t[:, 0],t[:, 1],fill=False)
ax.set_rmax(1200)
ax.set_rticks([500, 1000])  # Less radial ticks
ax.grid(True)

ax.set_title("Lidar Scann with Error", va='bottom')
```

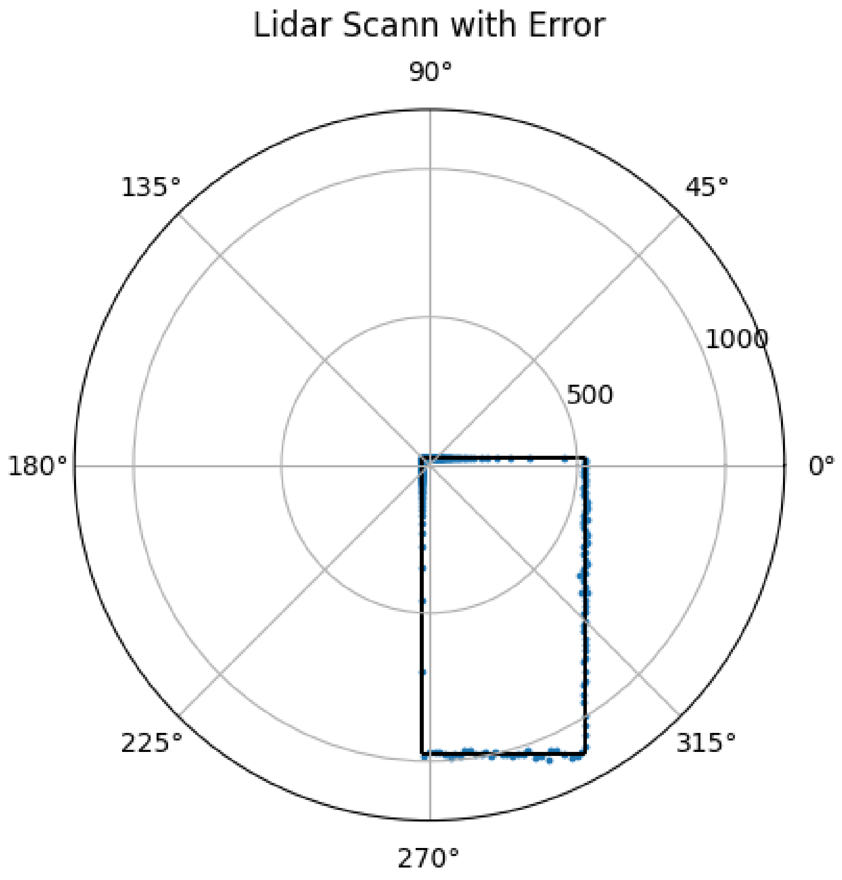
```
Text(0.5, 1.0, 'Lidar Scann with Error')
```



```
test_lidar = LidarSimulator("rectangle.stl", error=0.01)
point = [25, 25]
yaw = np.radians(90)
plot_scan = test_lidar.get_lidar_points(point[0], point[1], yaw)
triangles = test_lidar.get_env_triangles(point[0], point[1], yaw)
```

```
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
ax.scatter(plot_scan[:, 0], plot_scan[:, 1], s=3.0)
for t in triangles:
    ax.fill(t[:, 0],t[:, 1],fill=False)
ax.set_rmax(1200)
ax.set_rticks([500, 1000])  # Less radial ticks
ax.grid(True)

ax.set_title("Lidar Scann with Error", va='bottom')
```

```
Text(0.5, 1.0, 'Lidar Scann with Error')
```


Lidar Scann with Error
90°
45°
0°
315°
270°
225°
180°
135°
500
1000

CHAPTER

TEN

FEATURE EXTRACTION FROM LIDAR

We analyze the data from the Lidar to find simple geometric features like lines or circles.

We must answer questions like:

- how many lines are there
- which point belongs to which line
- what are the characteristics of this line

10.1 Field-of-View (FoV)

Our simulated Lidar provides a view of 360 degrees, to extract fearures we will restrict or fied of view to a direction θ and a range specified in degrees. Our FoV if then θ plus/minus the range

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from LidarSim.lidar_sim import LidarSimulator
```

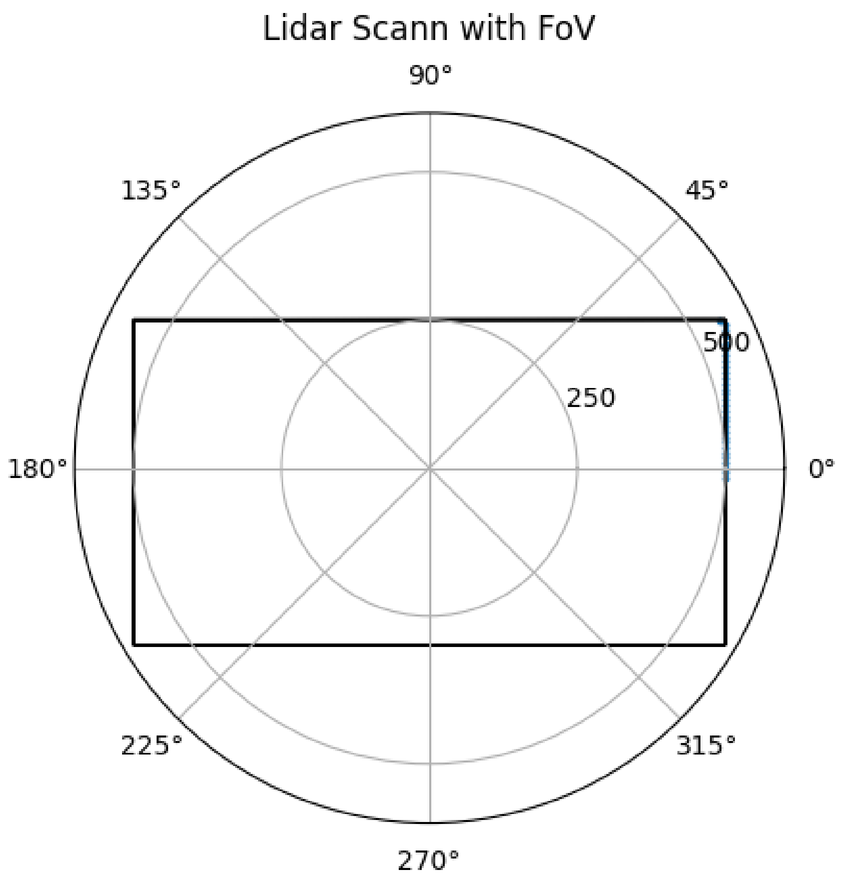
```
lidar = LidarSimulator("rectangle.stl")
point = [500, 300]
yaw = np.radians(0)
plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
triangles = lidar.get_env_triangles(point[0], point[1], yaw)
```

```
def get_fov(theta, view_range, scan, resolution):
    idx = np.searchsorted(scan[:, 0], theta) - 1
    idx_max = int(idx + view_range/resolution)
    idx_min = int(idx - view_range/resolution)
    if idx_max <= scan.shape[0] and idx_min >=0:
        return scan[idx_min:idx_max]
    if idx_min < 0:
        return np.roll(scan, -idx_min, axis = 0)[:2*int(view_range/resolution)]
    if idx_max > scan.shape[0]:
        return np.roll(scan, -(idx_max - scan.shape[0]), axis = 0)[-2*int(view_range/
↪resolution):]
```

```
fov = get_fov(np.radians(14), 15, plot_scan, lidar.resolution)
```

```
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
ax.scatter(fov[:, 0], fov[:, 1], s=3.0)
for t in triangles:
    ax.fill(t[:, 0],t[:, 1],fill=False)
ax.set_rmax(600)
ax.set_rticks([250, 500])  # Less radial ticks
ax.grid(True)

ax.set_title("Lidar Scann with FoV", va='bottom')
```

```
Text(0.5, 1.0, 'Lidar Scann with FoV')
```



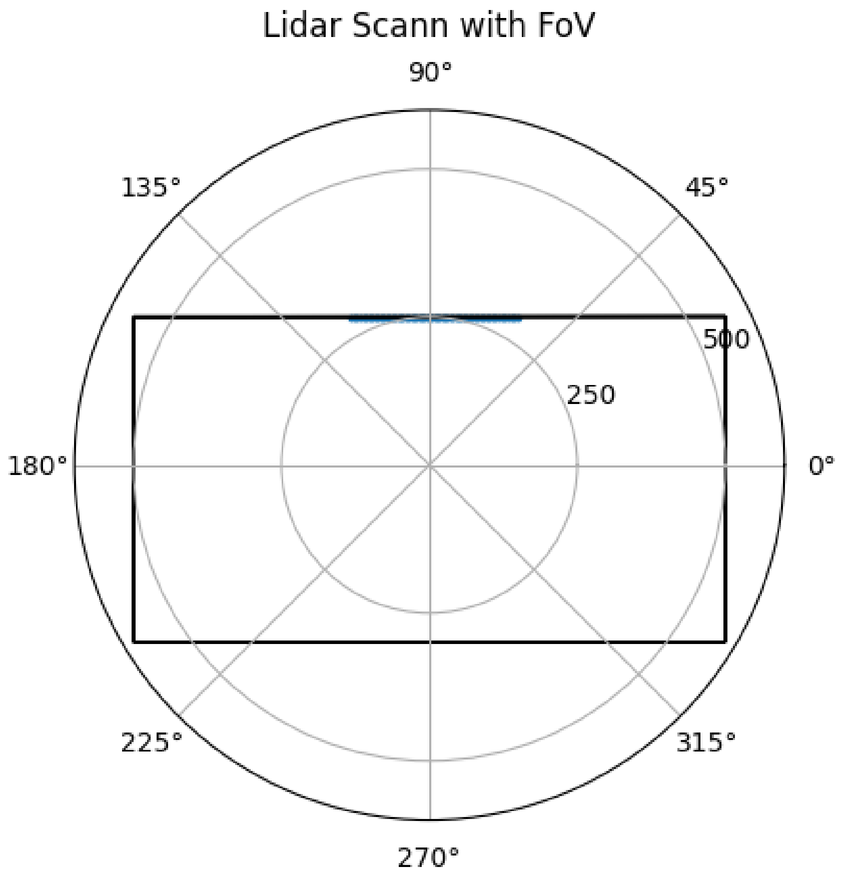
10.2 FoV and LidarSimulator module

This is an example how to set the Fow when calling the LidarSimulator module

```
lidar = LidarSimulator("rectangle.stl")
point = [500, 300]
yaw = np.radians(0)
plot_scan = lidar.get_lidar_points(point[0], point[1], yaw, theta=np.radians(90),␣
↪view_range=30)
triangles = lidar.get_env_triangles(point[0], point[1], yaw)
```

```
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
ax.scatter(plot_scan[:, 0], plot_scan[:, 1], s=3.0)
for t in triangles:
    ax.fill(t[:, 0],t[:, 1],fill=False)
ax.set_rmax(600)
ax.set_rticks([250, 500])  # Less radial ticks
ax.grid(True)

ax.set_title("Lidar Scann with FoV", va='bottom')
```

```
Text(0.5, 1.0, 'Lidar Scann with FoV')
```



CHAPTER

ELEVEN

PROBLEM STATEMENT

Assuming we have a set of points that all belong to a single line. By choosing two points we can calculate the characteristics of the line and assuming we have no statistical error we can prove that all other points are part of the line

However we will always have measurement errors with Lidar, therefore we need to use statistical methods to fit a line through the points that minimizez the error

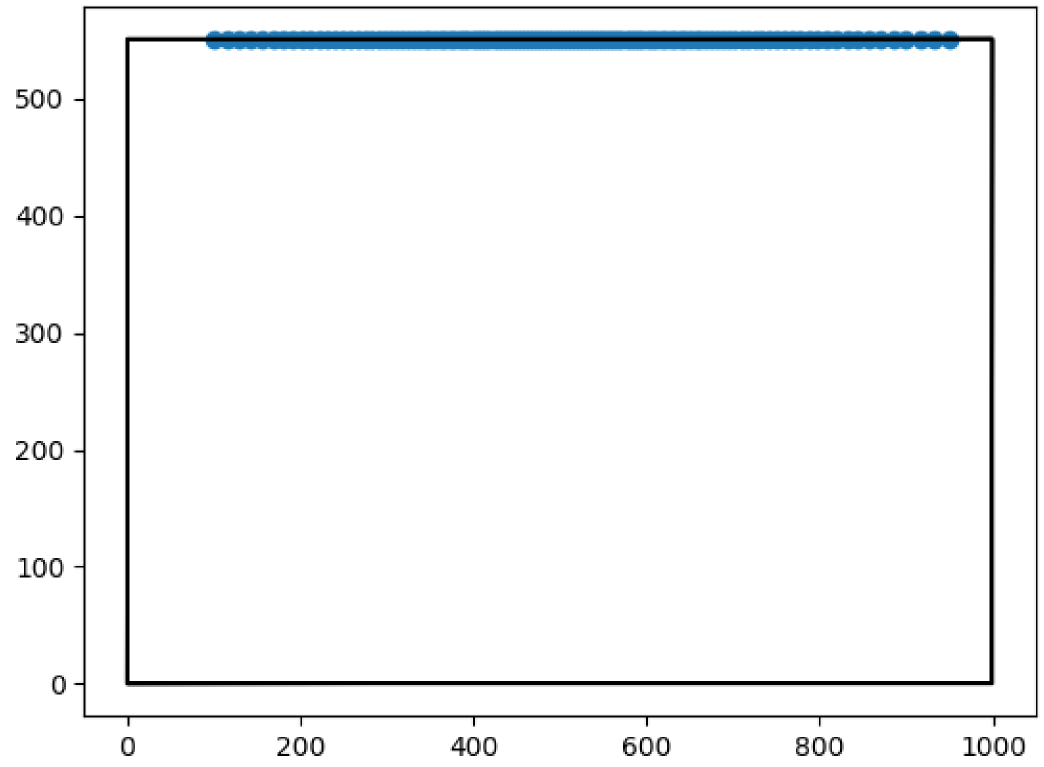
11.1 Ideal Situation

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from LidarSim.lidar_sim import LidarSimulator
```

```
lidar = LidarSimulator("rectangle.stl")
point = [500, 300]
yaw = np.radians(0)
plot_scan = lidar.get_lidar_points(point[0], point[1], yaw, theta=np.radians(90),␣
↪view_range=60)
triangles = lidar.get_map_triangles()
```

```
# get carthesian coordinates
x = []
y = []
for alpha, r in plot_scan:
    x.append(r * np.cos(alpha)+ point[0])
    y.append(r * np.sin(alpha) + point[1])
```

```
fig,ax = plt.subplots()
plt.scatter (x, y)
for t in triangles:
    ax.fill(t[:, 0],t[:, 1],fill=False)
```



```
m = (y[-1] - y[0]) / (x[-1] - x[0])
b = y[-1] - m * x[-1]
y_err = []
for i in range(len(x)):
    y_err.append(y[i] - m * x[i] - b)

y_err = np.array(y_err)
print("mean: %0.2f, median: %0.2f, standard deviation: %0.2f" % (np.mean(y_err), np.
 ↪median(y_err), np.std(y_err)))
```

```
mean: 0.00, median: 0.00, standard deviation: 0.00
```

CHAPTER

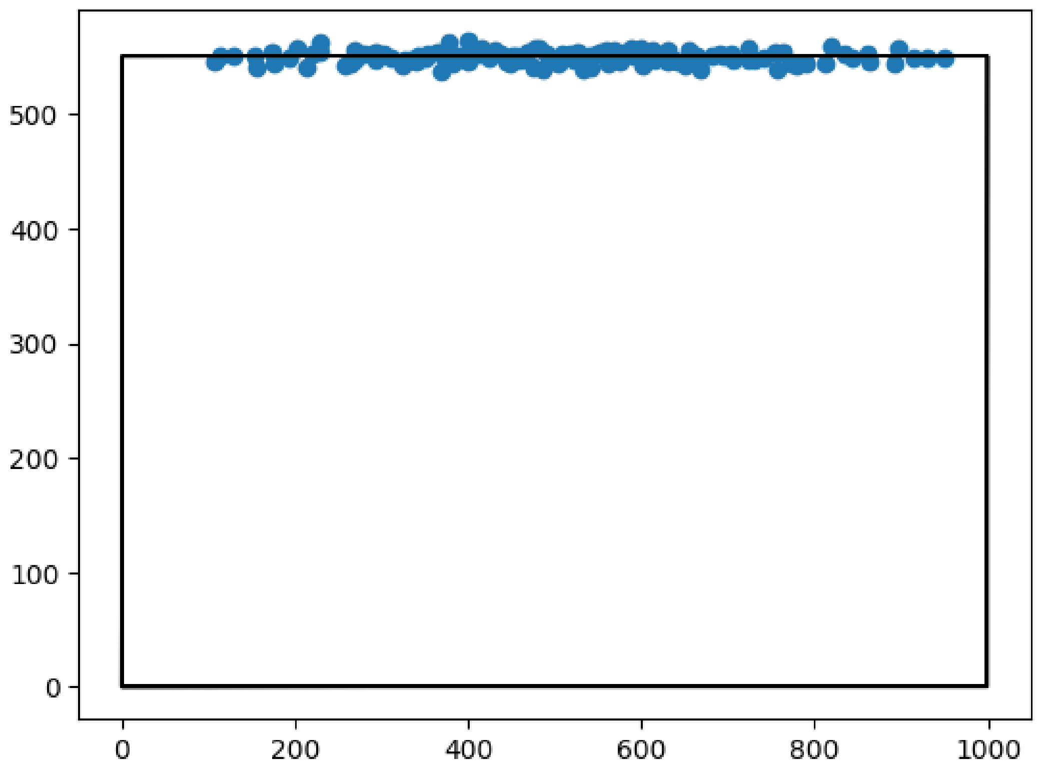
TWELVE

WITH STATISTICAL ERROR

```
lidar = LidarSimulator("rectangle.stl", error=0.02)
point = [500, 300]
yaw = np.radians(0)
plot_scan = lidar.get_lidar_points(point[0], point[1], yaw, theta=np.radians(90),␣
 ↪view_range=60)
triangles = lidar.get_map_triangles()
```

```
# get carthesian coordinates
x = []
y = []
for alpha, r in plot_scan:
    x.append(r * np.cos(alpha)+ point[0])
    y.append(r * np.sin(alpha) + point[1])
```

```
fig,ax = plt.subplots()
plt.scatter (x, y)
for t in triangles:
    ax.fill(t[:, 0],t[:, 1],fill=False)
```



```
m = (y[-1] - y[0]) / (x[-1] - x[0])
b = y[-1] - m * x[-1]
y_err = []
for i in range(len(x)):
    y_err.append(y[i] - m * x[i] - b)

y_err = np.array(y_err)
print("mean: %0.2f, median: %0.2f, standard deviation: %0.2f" % (np.mean(y_err), np.
 ↪median(y_err), np.std(y_err)))
```

```
mean: 2.50, median: 2.93, standard deviation: 5.41
```

12.1 Statistical Line Fitting

```
model = np.polyfit (x, y, 1)
```

```
y_err = []
for i in range(len(x)):
    y_err.append(y[i] - model[0] * x[i] - model[1])

y_err = np.array(y_err)
print("mean: %0.2f, median: %0.2f, standard deviation: %0.2f" % (np.mean(y_err), np.
 ↪median(y_err), np.std(y_err)))
```

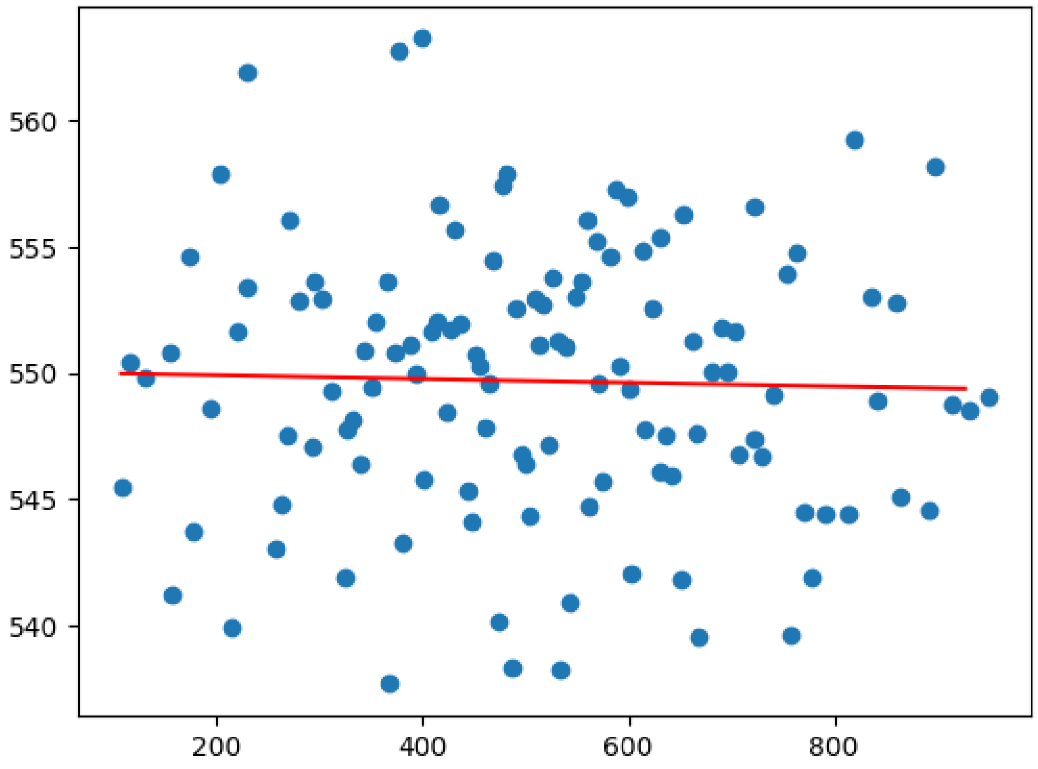
```
mean: 0.00, median: 0.35, standard deviation: 5.32
```

```
x[-1]
```

```
107.13131145805664
```

```
x_lin_reg = np.arange(x[-1], x[1], 10)
predict = np.poly1d(model)
y_lin_reg = predict(x_lin_reg)

fig,ax = plt.subplots()
plt.scatter (x, y)
plt.plot (x_lin_reg, y_lin_reg, c = 'r')
```

```
[<matplotlib.lines.Line2D at 0x110d3ae50>]
```



12.2 Fitting a line with Least Square Method

Reference: https://www.varsitytutors.com/hotmath/hotmath_help/topics/line-of-best-fit

Step 1: Calculate the mean of the x -values and the mean of the y -values.

$$\overline{X} = \frac{\sum_{i=1}^{n} x_i}{n}$$

$$\overline{Y} = \frac{\sum_{i=1}^{n} y_i}{n}$$

Step 2: The following formula gives the slope of the line of best fit:

$$m = \frac{\sum_{i=1}^{n} (x_i - \overline{X})(y_i - \overline{Y})}{\sum_{i=1}^{n} (x_i - \overline{X})^2}$$

Step 3: Compute the y-intercept of the line by using the formula:

$$b = \overline{Y} - m\overline{X}$$

Step 4: Use the slope m and the y -intercept b to form the equation of the line.

To contruct a perpendicular line we have to solve the equations

$$y = mx + b$$

$$y = -\frac{1}{m}x$$

```
angles = np.arange(0, 85, 5)
values = [0.5197, 0.4404, 0.4850, 0.4222, 0.4132, 0.4371, 0.3912, 0.3949, 0.3919, 0.
↪4276, 0.4075, 0.3956, 0.4053, 0.4752, 0.5032, 0.5273, 0.4879]
```

```
# get carthesian coordinates
x_cart = []
y_cart = []
for i in range(len(values)):
    r = values[i]
    alpha = np.radians(angles[i])
    x_cart.append(r * np.cos(alpha))
    y_cart.append(r * np.sin(alpha))
```

```
# Calculate centroid
X = np.average(x_cart)
Y = np.average(y_cart)
```

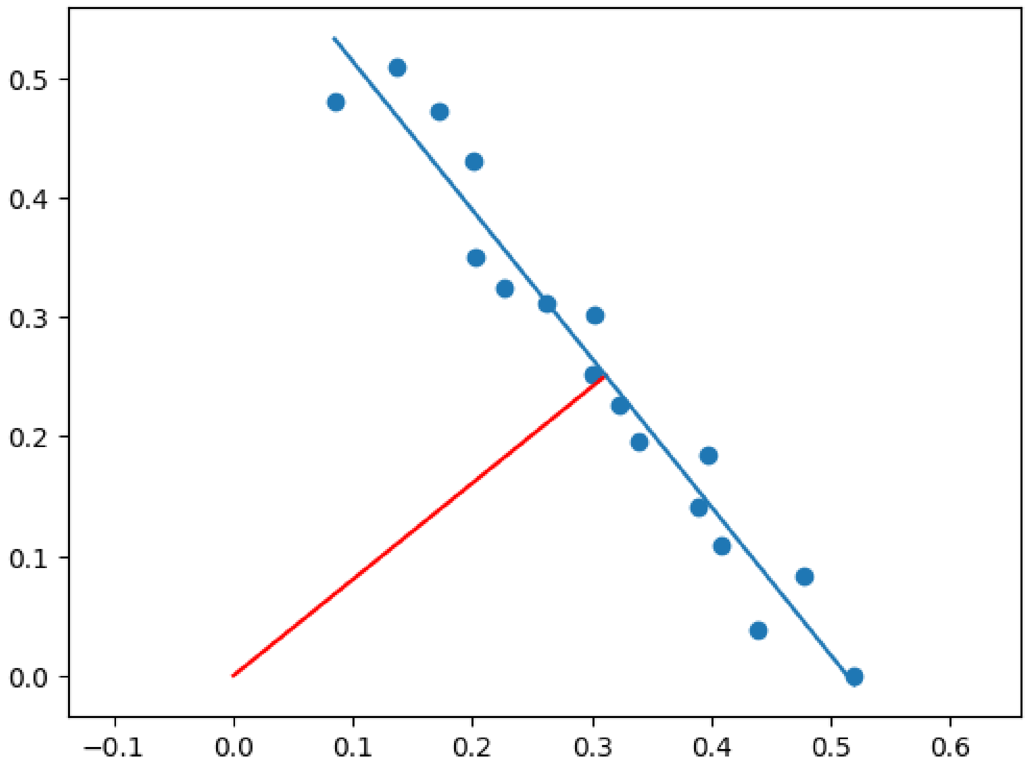
```
# calculate line parameters
X_d = (np.array(x_cart) - X)
Y_d = (np.array(y_cart) - Y)
X_d_2 = X_d * X_d
m = (X_d * Y_d).sum() / X_d_2.sum()
b = Y - m * X
```

```
# calculate line

line = np.empty((2,2))
line[0,0] = np.array(x_cart).min()
line[1,0] = np.array(x_cart).max()
line[0,1] = m * line[0,0] + b
line[1,1] = m * line[1,0] + b
```

```
# caclulate perpendicular line
x = -b*m/(m*m+1)
y = -x/m
```

```
fig,ax = plt.subplots()
plt.plot([0, x], [0, y], c = 'r')
plt.plot(line[:, 0], line[:, 1])
plt.scatter (x_cart, y_cart)
ax.axis('equal')
```

```
(-0.025985000000000005,
 0.5456850000000001,
 -0.034309997394643715,
 0.5597197002622666)
```



```
alpha = np.arctan2(y, x)
r = np.sqrt(x**2 + y**2)
print("alpha = %.2f r = %.2f" % (np.degrees(alpha), r))
```

```
alpha = 38.85 r = 0.40
```

```
def fitLine(x, y):
    X = np.average(x)
    Y = np.average(y)
    X_d = (np.array(x) - X)
    Y_d = (np.array(y) - Y)
    X_d_2 = X_d * X_d
    X_d_2_sum = X_d_2.sum()
    if np.isclose([X_d_2_sum], [0.0])[0]:
        if X >= 0:
            alpha = 0.0
            r = X
        else:
            alpha = np.pi
            r = -X
        return None, None, alpha, r
    m = (X_d * Y_d).sum() / X_d_2.sum()
    b = Y - m * X
    # caclulate perpendicular line
    x_p = -b*m/(m*m+1)
    if np.isclose([m], [0.0])[0]:
        if Y >= 0:
            alpha = np.pi / 2
            r = Y
        else:
            alpha = 3 * np.pi / 2
            r = -Y
        return None, None, alpha, r
    y_p = -x_p/m
    alpha = np.arctan2(y_p, x_p)
    r = np.sqrt(x_p**2 + y_p**2)
    return m, b, alpha, r
```

```
x_cart = -1 * np.array(x_cart)
y_cart = -1 * np.array(y_cart)
```

```
m, b, alpha, r = fitLine(x_cart, y_cart)
print("alpha = %.2f r = %.2f" % (np.degrees(alpha), r))
```

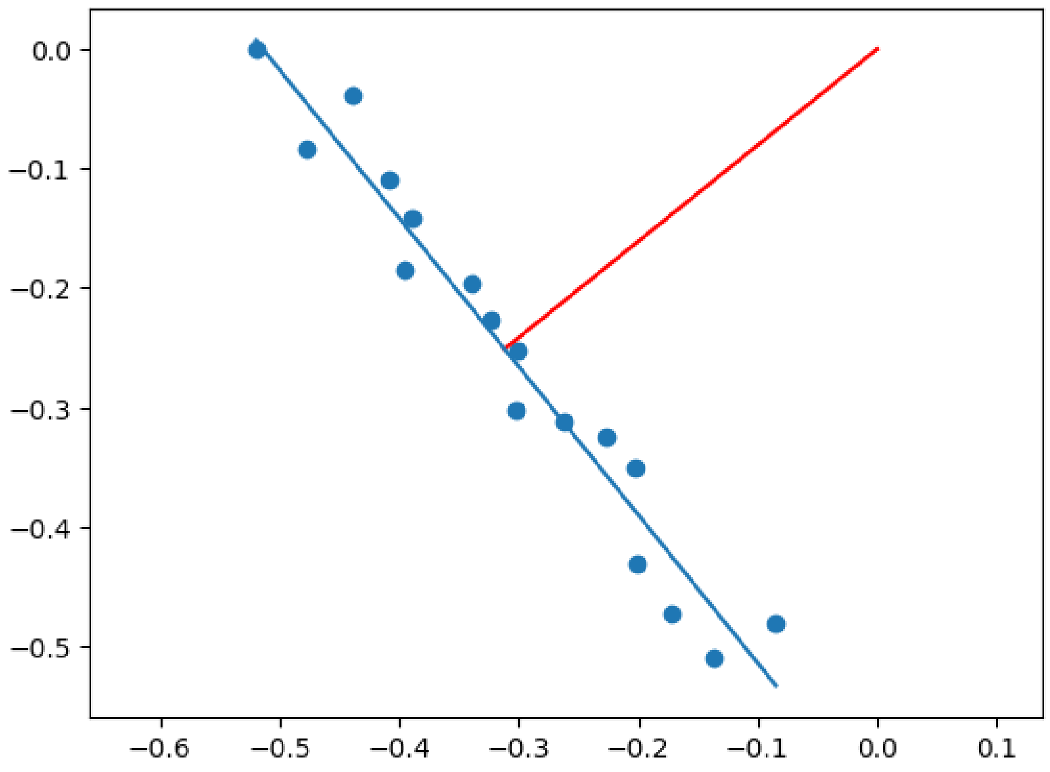
```
alpha = -141.15 r = 0.40
```

```
# calculate line

line = np.empty((2,2))
line[0,0] = np.array(x_cart).min()
line[1,0] = np.array(x_cart).max()
line[0,1] = m * line[0,0] + b
line[1,1] = m * line[1,0] + b
```

```
# caclulate perpendicular line
x = -b*m/(m*m+1)
y = -x/m
```

```
fig,ax = plt.subplots()
plt.plot([0, x], [0, y], c = 'r')
plt.plot(line[:, 0], line[:, 1])
plt.scatter (x_cart, y_cart)
ax.axis('equal')
```

```
(-0.5456850000000001,
 0.025985000000000005,
 -0.5597197002622666,
 0.034309997394643715)
```



```
cosA = np.cos(alpha);
sinA = np.sin(alpha);

xcosA = x_cart * cosA;
ysinA = y_cart * sinA;
```

```
m, b, alpha, r = fitLine([0, 1], [1, 0])
np.testing.assert_allclose([alpha, r], [np.pi / 4, np.sqrt(0.5)])
```

```
m, b, alpha, r = fitLine([-1, 0], [0, 1])
np.testing.assert_allclose([alpha, r], [3 * np.pi / 4, np.sqrt(0.5)])
```

```
m, b, alpha, r = fitLine([-1, 1], [1, 1])
np.testing.assert_allclose([alpha, r], [np.pi/2, 1.0])
```

```
m, b, alpha, r = fitLine([-1, 1], [-1, -1])
np.testing.assert_allclose([alpha, r], [3 * np.pi / 2, 1.0])
```

```
m, b, alpha, r = fitLine([1, 1], [1, -1])
np.testing.assert_allclose([alpha, r], [0.0, 1.0])
```

```
m, b, alpha, r = fitLine([-1, -1], [1, -1])
np.testing.assert_allclose([alpha, r], [np.pi, 1.0])
```

```
rng = np.random.default_rng(seed=43)
for i in range(30):
    alpha_expected = rng.uniform(low=-np.pi, high=np.pi)
    r_expected = rng.uniform(low=1.0, high=2.0)
    ts = np.arange(-1.0, 1.1, 0.1)
    p = [r_expected * np.cos(alpha_expected), r_expected * np.sin(alpha_expected)]
    v = [ -np.sin(alpha_expected), np.cos(alpha_expected)]
    x_cart = p[0] + v[0] * ts
    y_cart = p[1] + v[1] * ts
    m, b, alpha, r = fitLine(x_cart, y_cart)
    np.testing.assert_allclose([alpha, r], [alpha_expected, r_expected])
    error = rng.uniform(low=-0.01, high=0.01, size=(2, len(ts)))
    x_cart += error[0]
    y_cart += error[1]
    m, b, alpha, r = fitLine(x_cart, y_cart)
    np.testing.assert_allclose([alpha, r], [alpha_expected, r_expected], atol=0.01)
```

12.3 Fitting a line with Polar Coordinates

σ_i^2 is the variance that models the uncertainty regarding distance ρ_i of a particular sensor measurement

$$w_i = \frac{1}{\sigma_i^2}$$

$$\alpha = \frac{1}{2} atan \left(\frac{\sum w_i \rho_i^2 sin 2\theta_i - \frac{2}{\sum_{w_i}} \sum\sum w_i w_j \rho_i \rho_j cos\theta_i sin\theta_j}{\sum w_i \rho_i^2 cos 2\theta_i - \frac{1}{\sum_{w_i}} \sum\sum w_i w_j \rho_i \rho_j cos(\theta_i + \theta_j)} \right)$$

$$r = \frac{\sum w_i \theta_i cos(\theta_i - \alpha)}{\sum w_i}$$

12.4 TODO

c = np.cos(angles) c2 = np.cos(2 * angles) s = np.sin(angles) s2 = np.sin(2 * angles) r_square = np.array(values)**2 N = angles.shape[0] y = r_square * s2 - 2/N * np.array(values) * c * np.array(values) * s x = r_square * c2 - csIJ / N; alpha = 0.5 * (atan2(y, x) + pi); r = rho * cos(theta - ones(size(theta)) * alpha)’ / N;

CHAPTER

THIRTEEN

ALGORITHM 1: SPLIT-AND-MERGE

Data: Set S consisting of all N points, a distance threshold d > 0

Result: L, a list of sets of points each resembling a line

```
L ← (S), i ← 1;
while i ≤ len(L) do
  fit a line (r,α) to the set Li;
  detect the point P ∈ Li with the maximum distance D to the line (r, α);
  if D < d then
    i←i+1
  else
    split Li at P into S1 and S2;
    Li ← S1; Li+1 ← S2;
  end
end

Merge collinear sets in L;
```

13.1 Reference

Roland Siegwart, Illah Nourbakhsh, and Davide Scaramuzza. Introduction to Autonomous Mobile Robots. MIT Press, 2nd edition, 2011.

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pickle
from SplitAndMerge.split_and_merge import SplitAndMerge
```

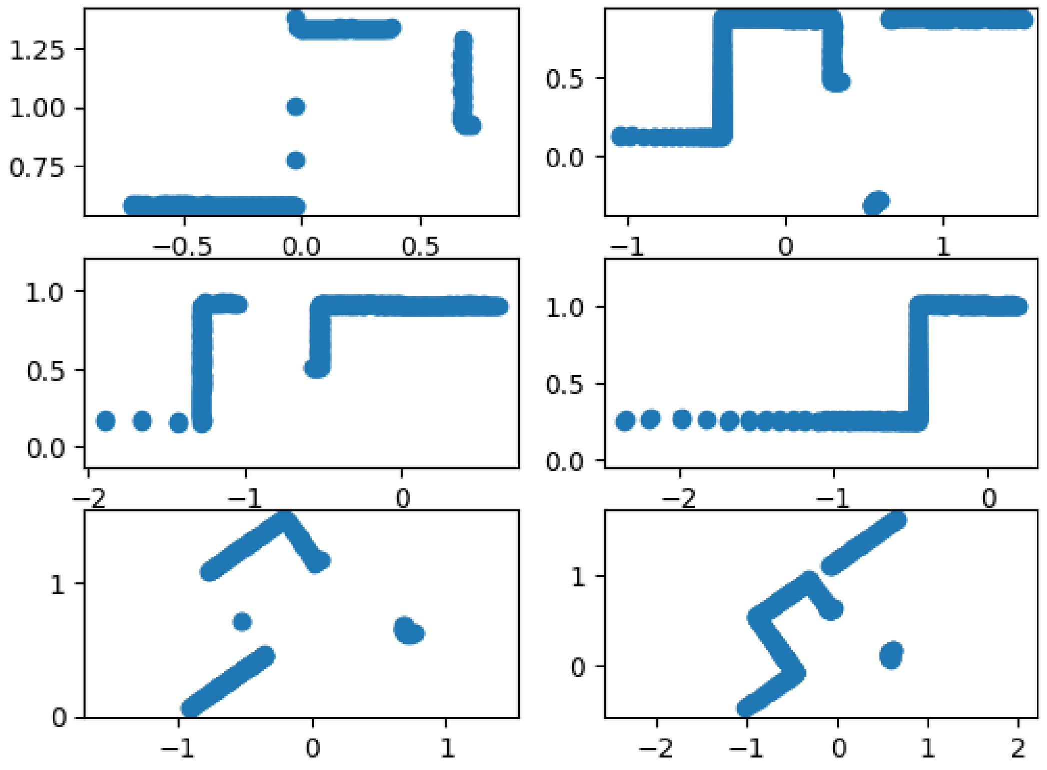
```
testdata = []
for i in range(6):
    with open('/Users/hdumcke/git/lidar-simulator/jupyternb/data/testLineExtraction%s.
↪mat.pickle' % str(i+1), 'rb') as f:
        testdata.append(pickle.load(f))
```

```
fig, axs = plt.subplots(3, 2)
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
```

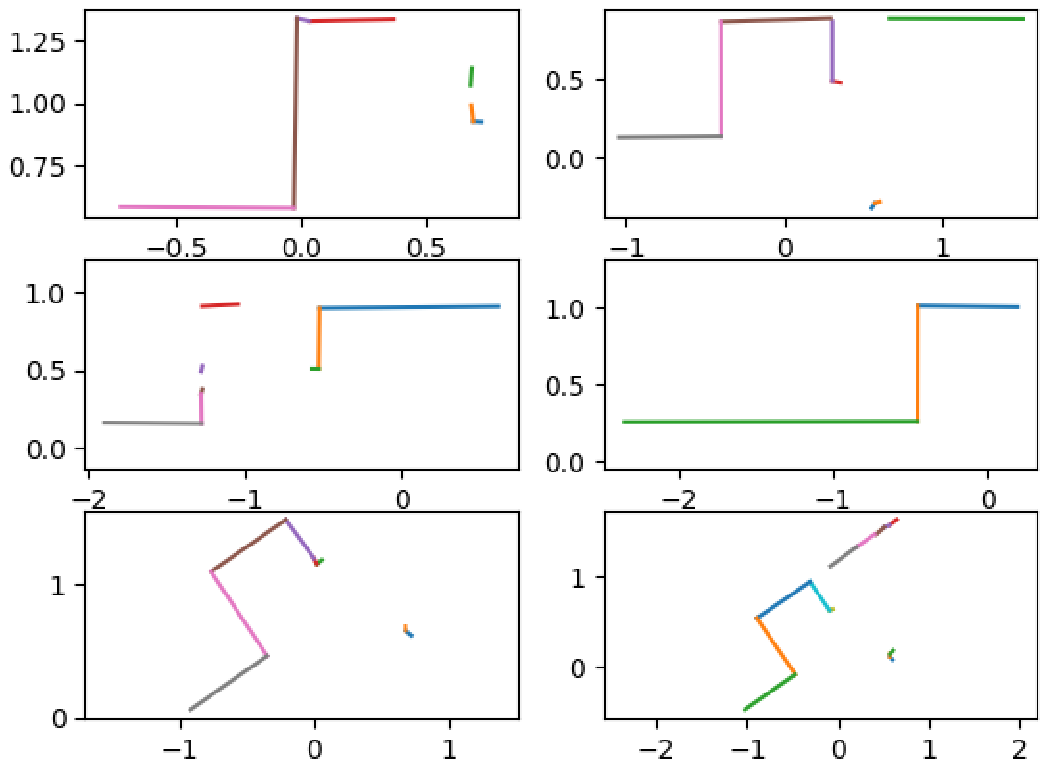
(continues on next page)

(continued from previous page)

```
    axs[row_index, column_index].axis('equal')
    x = testdata[i]['rho'] * np.cos(testdata[i]['theta'])
    y = testdata[i]['rho'] * np.sin(testdata[i]['theta'])
    axs[row_index, column_index].scatter(x, y)
```



```
sam = SplitAndMerge(line_point_dist_threshold=0.004, min_points_per_segment=4, min_
↪seg_length=0.01)

fig, axs = plt.subplots(3, 2)
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    x = testdata[i]['rho'] * np.cos(testdata[i]['theta'])
    y = testdata[i]['rho'] * np.sin(testdata[i]['theta'])
    alpha_a, r_a, segend, seglen, pointIdx_a = sam.extractLines(x[0], y[0])
    for j in range(segend.shape[0]):
        axs[row_index, column_index].plot([segend[j,0], segend[j,2]], [segend[j,1],␣
↪segend[j,3]])
```

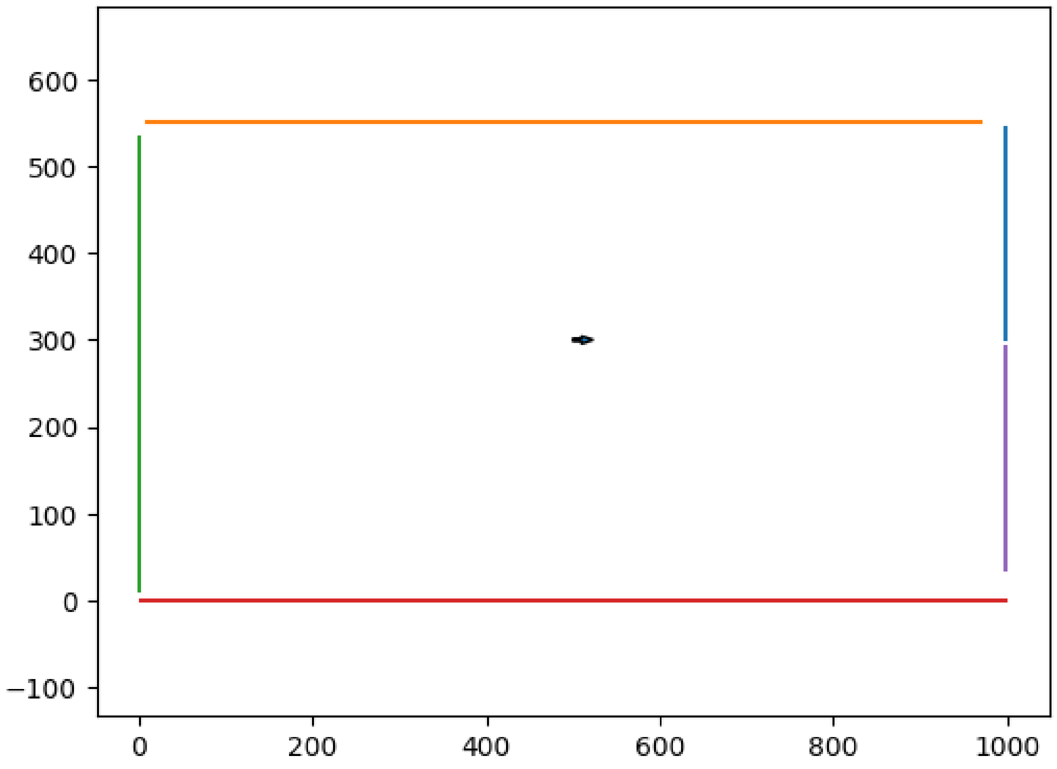


```
from LidarSim.lidar_sim import LidarSimulator
lidar = LidarSimulator("rectangle.stl")
```

```
point = [500, 300]
yaw = np.radians(0)
#plot_scan = lidar.get_lidar_points(point[0], point[1], yaw, theta=0, view_range=25)
plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
```

```
# get carthesian coordinates
x = []
y = []
for alpha, r in plot_scan:
    x.append(r * np.cos(alpha)+ point[0])
    y.append(r * np.sin(alpha) + point[1])
```

```
sam = SplitAndMerge(line_point_dist_threshold=0.005, min_points_per_segment=5)

fig,ax = plt.subplots()
ax.axis('equal')
plt.arrow(point[0], point[1], 10 * np.cos(yaw), 10 * np.sin(yaw), width=3.0)
#plt.scatter (x[211:329], y[211:329])
alpha_a, r_a, segend, seglen, pointIdx_a = sam.extractLines(x, y)
for j in range(segend.shape[0]):
    ax.plot([segend[j,0], segend[j,2]], [segend[j,1], segend[j,3]])
```



```
import Utilities.utilities as utilities
```

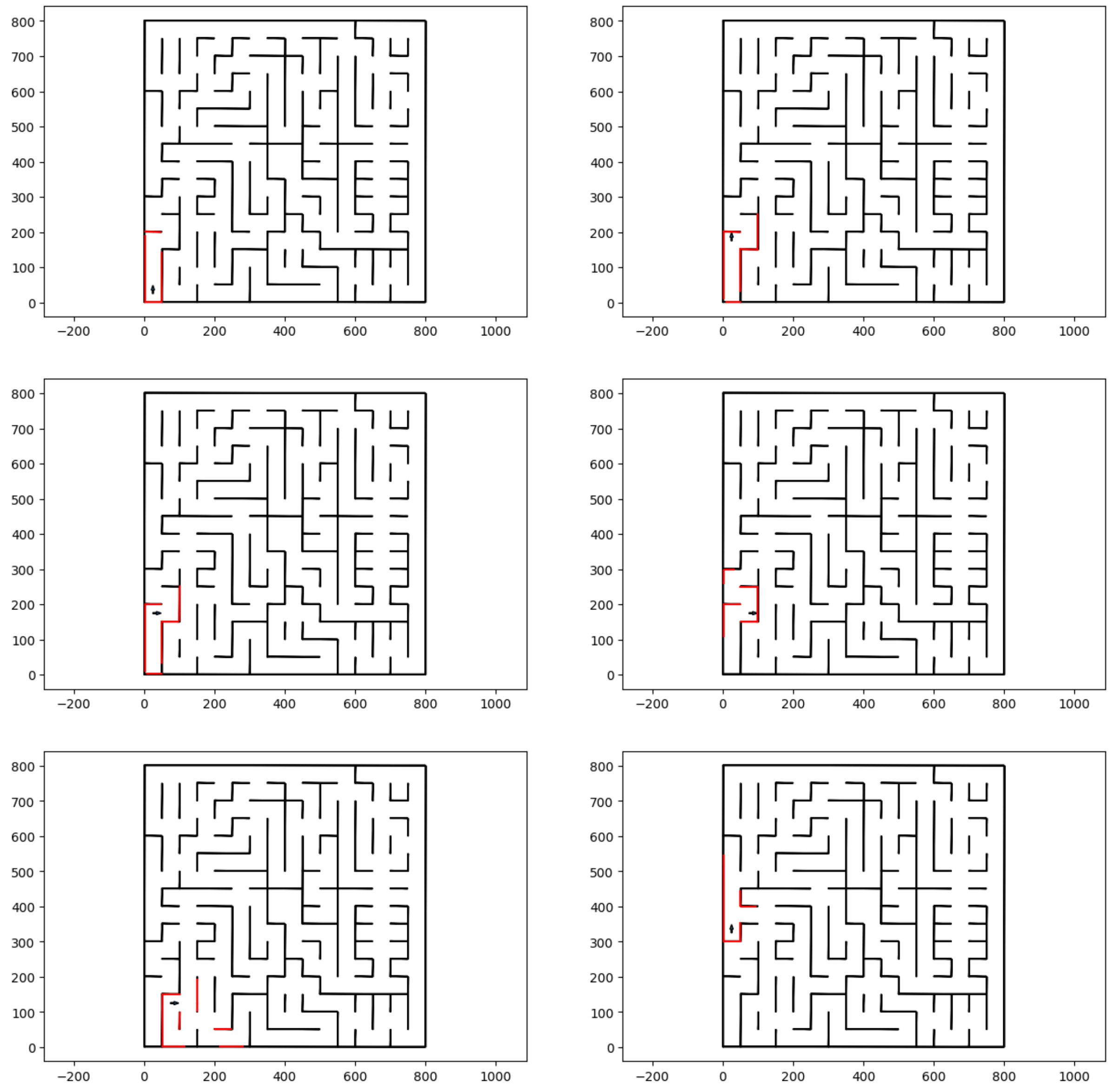
```
from LidarSim.lidar_sim import LidarSimulator
lidar = LidarSimulator("maze.stl")
triangles = lidar.get_map_triangles()
point = [25, 25]
yaw = np.radians(90)
```

```
pose = [[25, 25, 90 ],
        [25, 175, 90 ],
        [25, 175, 0 ],
        [75, 175, 0 ],
        [75, 125, 0 ],
        [25, 325, 90 ]]

fig, axs = plt.subplots(3, 2, figsize=(15, 15))
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    point = pose[i][0:2]
    yaw = np.radians(pose[i][2])
    plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
    # get carthesian coordinates
    x = []
    y = []
    for alpha, r in plot_scan:
        x.append(r * np.cos(alpha)+ point[0])
```

(continues on next page)

(continued from previous page)

```
        y.append(r * np.sin(alpha) + point[1])

    sam = SplitAndMerge(line_point_dist_threshold=0.005, min_points_per_segment=5)

    axs[row_index, column_index].arrow(point[0], point[1], 10 * np.cos(yaw), 10 * np.
↪sin(yaw), width=3.0)
    alpha_a, r_a, segend, seglen, pointIdx_a = sam.extractLines(x, y)
    for j in range(segend.shape[0]):
        x_p, y_p = utilities.rotate_segend(segend[j], point, yaw)
        axs[row_index, column_index].plot(x_p, y_p, 'r')
    for t in triangles:
        axs[row_index, column_index].fill(t[:, 0],t[:, 1],fill=False)
```



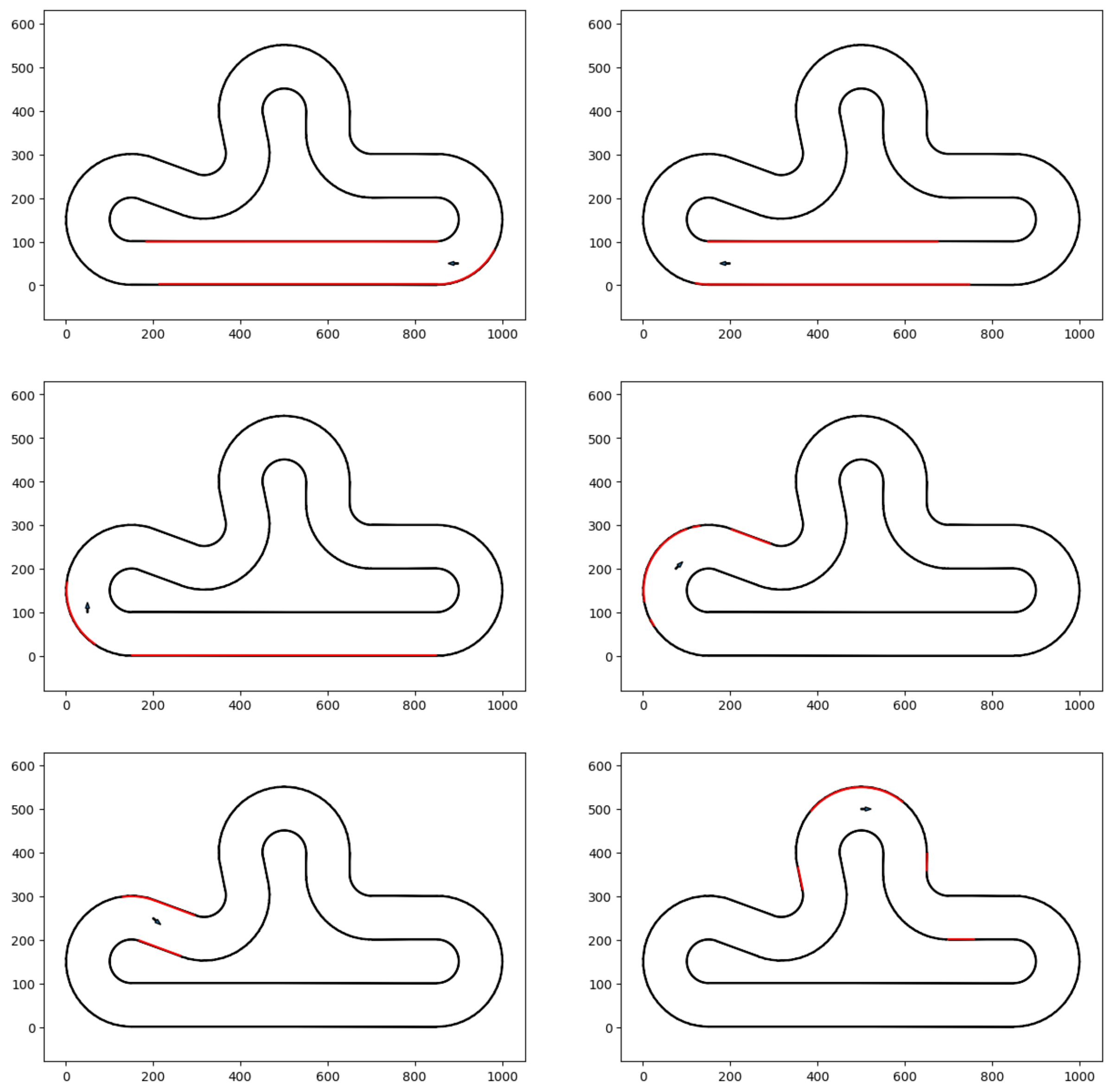
```
from LidarSim.lidar_sim import LidarSimulator
lidar = LidarSimulator("racetrack.stl")
```

(continues on next page)

(continued from previous page)

```
triangles = lidar.get_map_triangles()
yaw = np.radians(180)
```

```
pose = [[900, 50, 180 ],
        [200, 50, 180 ],
        [50, 100, 90 ],
        [75, 200, 45 ],
        [200, 250, -40 ],
        [500, 500, 0 ]]

fig, axs = plt.subplots(3, 2, figsize=(15, 15))
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    point = pose[i][0:2]
    yaw = np.radians(pose[i][2])
    plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
    # get carthesian coordinates
    x = []
    y = []
    for alpha, r in plot_scan:
        x.append(r * np.cos(alpha)+ point[0])
        y.append(r * np.sin(alpha) + point[1])

    sam = SplitAndMerge(line_point_dist_threshold=0.005, min_points_per_segment=5)

    axs[row_index, column_index].arrow(point[0], point[1], 10 * np.cos(yaw), 10 * np.
↪sin(yaw), width=3.0)
    alpha_a, r_a, segend, seglen, pointIdx_a = sam.extractLines(x, y)
    for j in range(segend.shape[0]):
        x_p, y_p = utilities.rotate_segend(segend[j], point, yaw)
        axs[row_index, column_index].plot(x_p, y_p, 'r')
    for t in triangles:
        axs[row_index, column_index].fill(t[:, 0],t[:, 1],fill=False)
```


600
500
400
300
200
100
0
0
200
400
600
800
1000

CHAPTER

FOURTEEN

ALGORITHM 2: LINE-REGRESSION

```
1. Initialize sliding window size Nf
2. Fit a line to every Nf consecutive points
3. Compute a line fidelity array. Each element of the array contains the sum of␣
↪Mahalanobis distances between every three adjacent windows
4. Construct line segments by scanning the fidelity array for consecutive elements␣
↪having values less than a threshold
5. Merge overlapped line segments and recompute line parameters for each segment
```

14.1 Reference

Roland Siegwart, Illah Nourbakhsh, and Davide Scaramuzza. Introduction to Autonomous Mobile Robots. MIT Press, 2nd edition, 2011.

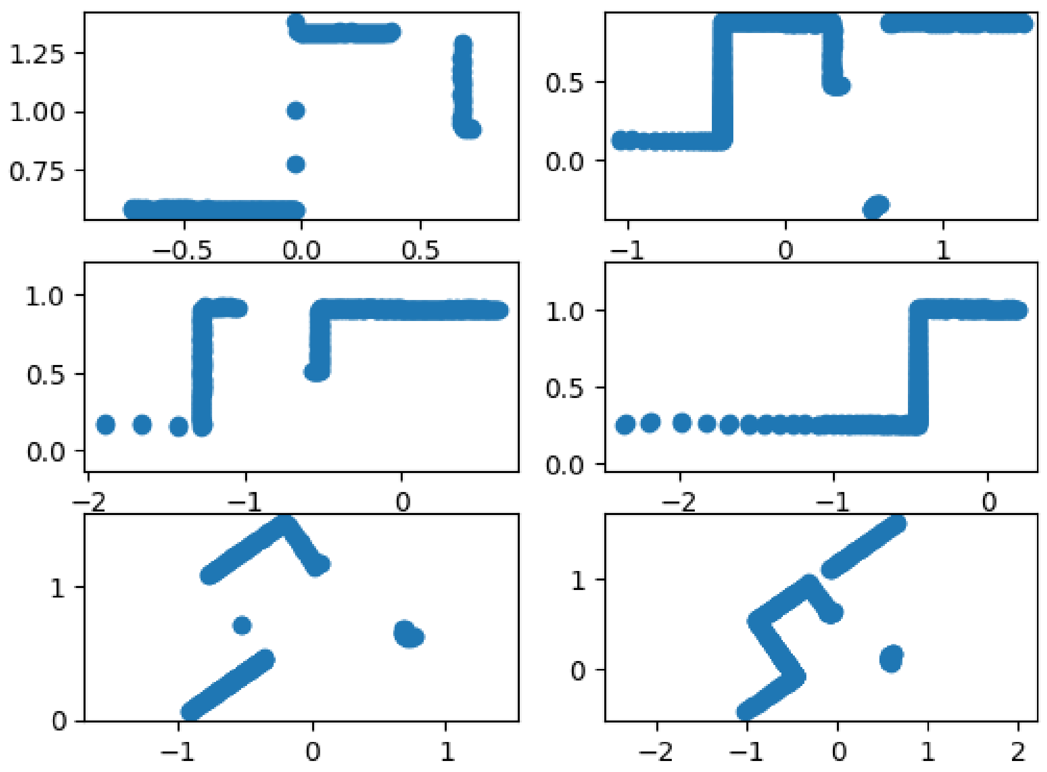
```
## ToDo
```

```
## Work in progress
```

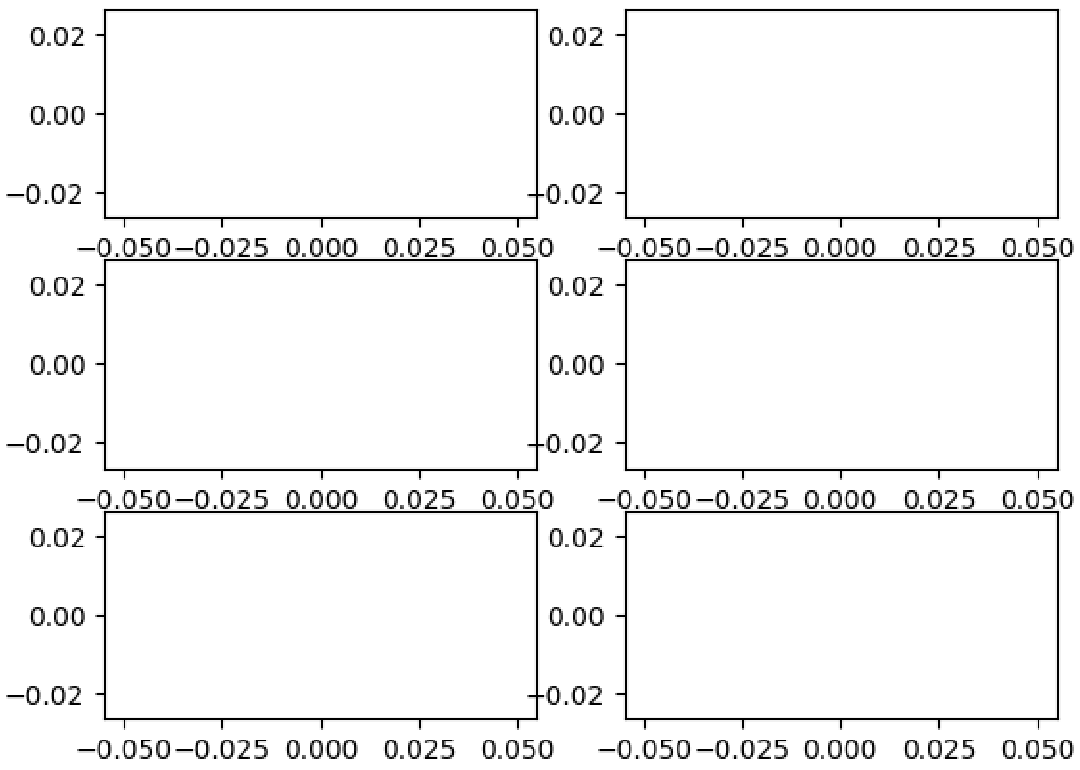
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pickle
from LineRegression.lineregression import LineRegression
```

```
testdata = []
for i in range(6):
    with open('/Users/hdumcke/git/lidar-simulator/jupyternb/data/testLineExtraction%s.
↪mat.pickle' % str(i+1), 'rb') as f:
        testdata.append(pickle.load(f))
```

```
fig, axs = plt.subplots(3, 2)
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    x = testdata[i]['rho'] * np.cos(testdata[i]['theta'])
    y = testdata[i]['rho'] * np.sin(testdata[i]['theta'])
    axs[row_index, column_index].scatter(x, y)
```



```
lr = LineRegression()

fig, axs = plt.subplots(3, 2)
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    x = testdata[i]['rho'] * np.cos(testdata[i]['theta'])
    y = testdata[i]['rho'] * np.sin(testdata[i]['theta'])
    segend, seglen, pointIdx_a = lr.extractLines(x[0], y[0])
    for j in range(segend.shape[0]):
        axs[row_index, column_index].plot([segend[j,0], segend[j,2]], [segend[j,1],
↪segend[j,3]])
```


0.02
0.00
−0.02
−0.050
−0.025
0.000
0.025
0.050

CHAPTER

FIFTEEN

ALGORITHM 3: INCREMENTAL

```
1. Start by the first 2 points, construct a line
2. Add the next point to the current line model
3. Recompute the line parameters by line fitting
4. If it satisfies the line condition, continue (go to step 2)
5. Otherwise, put back the last point, recompute the line parameters, return the line
6. Continue with the next two points, go to step 2
```

15.1 Reference

Roland Siegwart, Illah Nourbakhsh, and Davide Scaramuzza. Introduction to Autonomous Mobile Robots. MIT Press, 2nd edition, 2011.

```
Put all points on curve list, in order along the curve
Empty the line point list
Empty the line list
Until there are too few points on the curve
  Transfer first few points on the curve to the line point list Fit line to line␣
↪point list
  While fitted line is good enough
    Transfer the next point on the curve to the line point list and refit the line
  end
  Transfer last point(s) back to curve
  Refit line
  Attach line to line list
end
```

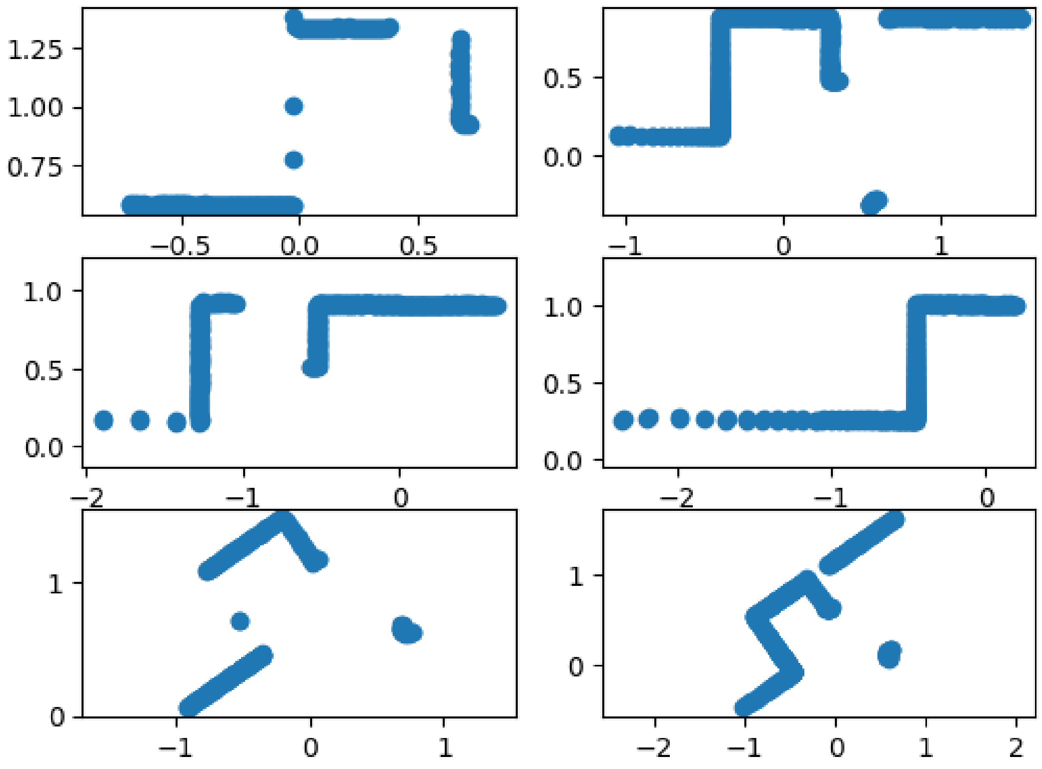
15.2 Reference

Forsyth, D. A., Ponce, J., Computer Vision: A Modern Approach. Upper Saddle River, NJ, Prentice Hall, 2003.

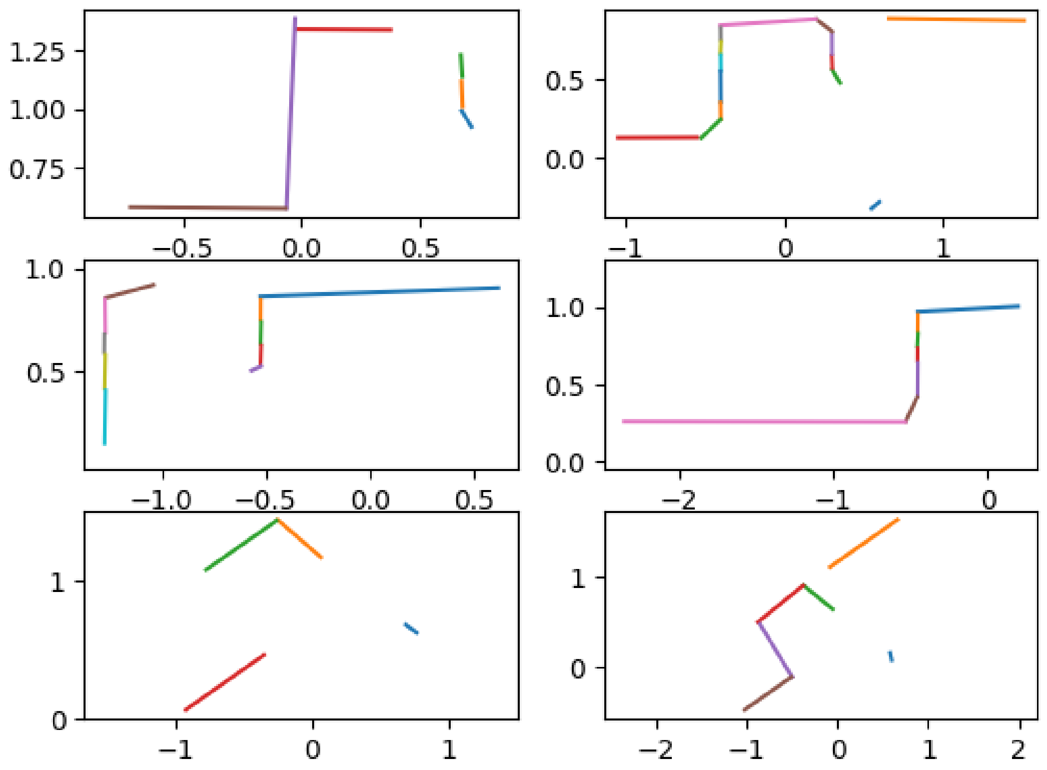
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pickle
from Incremental.incremental import Incremental
```

```
testdata = []
for i in range(6):
    with open('/Users/hdumcke/git/lidar-simulator/jupyternb/data/testLineExtraction%s.
↪mat.pickle' % str(i+1), 'rb') as f:
        testdata.append(pickle.load(f))
```

```
fig, axs = plt.subplots(3, 2)
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    x = testdata[i]['rho'] * np.cos(testdata[i]['theta'])
    y = testdata[i]['rho'] * np.sin(testdata[i]['theta'])
    axs[row_index, column_index].scatter(x, y)
```



```
inc = Incremental(dist_threshold=0.05, min_points_per_segment=5)

fig, axs = plt.subplots(3, 2)
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    x = testdata[i]['rho'] * np.cos(testdata[i]['theta'])
    y = testdata[i]['rho'] * np.sin(testdata[i]['theta'])
    segend, seglen, pointIdx_a = inc.extractLines(x[0], y[0])
    for j in range(segend.shape[0]):
        axs[row_index, column_index].plot([segend[j,0], segend[j,2]], [segend[j,1],␣
↪segend[j,3]])
```



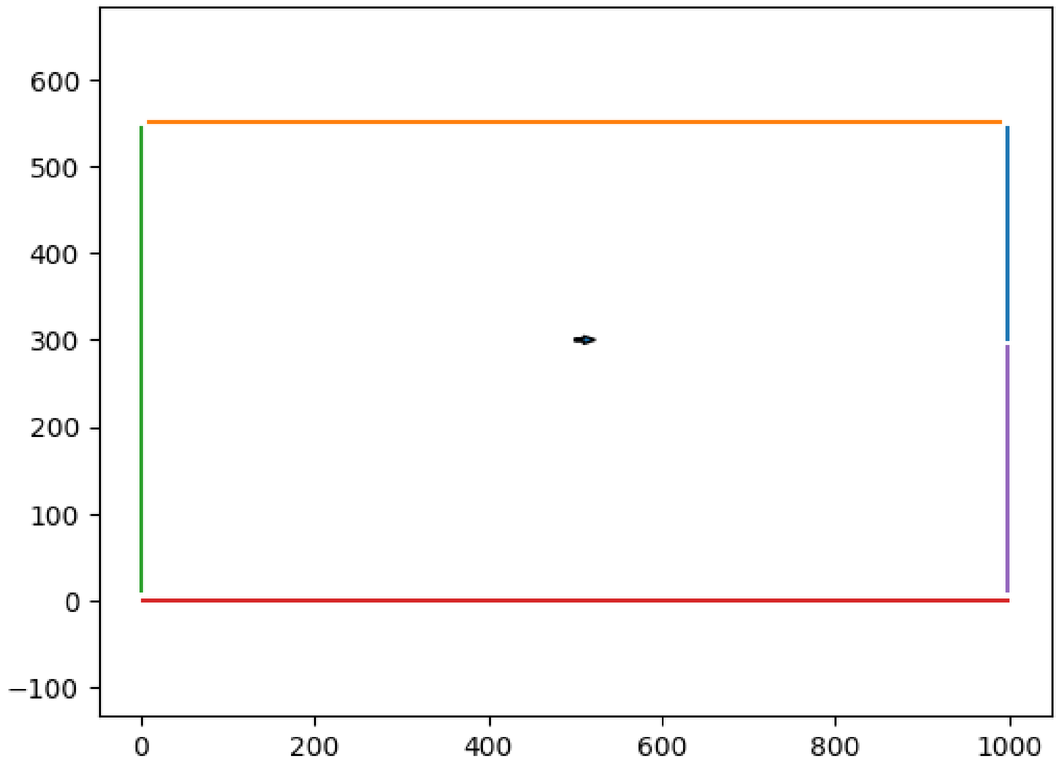
```
from LidarSim.lidar_sim import LidarSimulator
lidar = LidarSimulator("rectangle.stl")
```

```
point = [500, 300]
yaw = np.radians(0)
plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
```

```
# get carthesian coordinates
x = []
y = []
for alpha, r in plot_scan:
    x.append(r * np.cos(alpha)+ point[0])
    y.append(r * np.sin(alpha) + point[1])
```

```
inc = Incremental()

fig,ax = plt.subplots()
ax.axis('equal')
plt.arrow(point[0], point[1], 10 * np.cos(yaw), 10 * np.sin(yaw), width=3.0)
#plt.scatter (x[212:330], y[212:330])
segend, seglen, pointIdx_a = inc.extractLines(x, y)
for j in range(segend.shape[0]):
    ax.plot([segend[j,0], segend[j,2]], [segend[j,1], segend[j,3]])
```

```
/Users/hdumcke/git/lidar-simulator/python/incremental/src/Incremental/incremental.
↪py:20: RuntimeWarning: invalid value encountered in scalar divide
  m = (X_d * Y_d).sum() / X_d_2.sum()
```



```
import Utilities.utilities as utilities
```

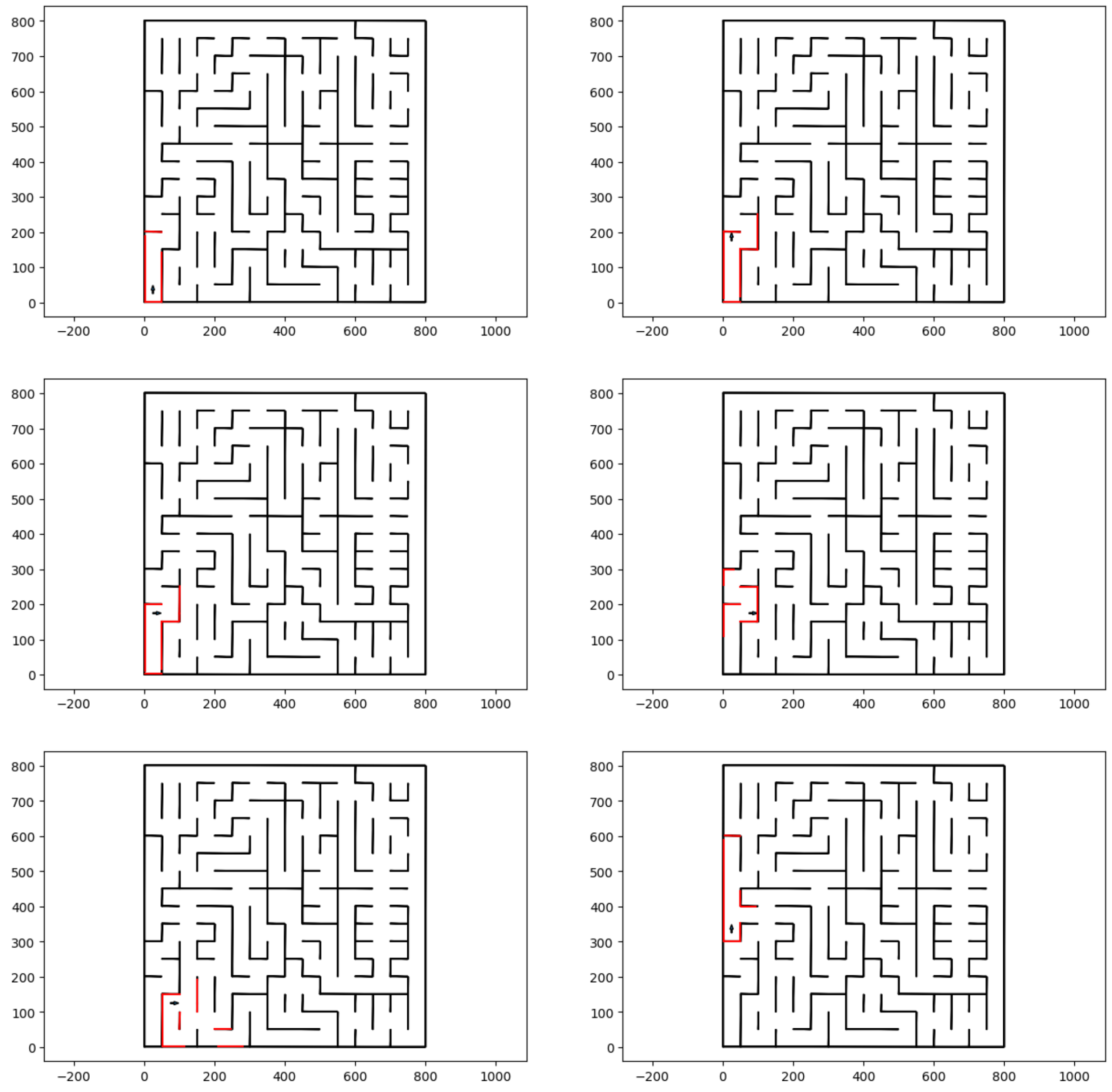
```
from LidarSim.lidar_sim import LidarSimulator
lidar = LidarSimulator("maze.stl")
triangles = lidar.get_map_triangles()
point = [25, 25]
yaw = np.radians(90)
```

```
pose = [[25, 25, 90 ],
        [25, 175, 90 ],
        [25, 175, 0 ],
        [75, 175, 0 ],
        [75, 125, 0 ],
        [25, 325, 90 ]]

fig, axs = plt.subplots(3, 2, figsize=(15, 15))
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    point = pose[i][0:2]
    yaw = np.radians(pose[i][2])
    plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
    # get carthesian coordinates
    x = []
    y = []
    for alpha, r in plot_scan:
        x.append(r * np.cos(alpha)+ point[0])
```

(continues on next page)

(continued from previous page)

```
        y.append(r * np.sin(alpha) + point[1])

    inc = Incremental()

    axs[row_index, column_index].arrow(point[0], point[1], 10 * np.cos(yaw), 10 * np.
↪sin(yaw), width=3.0)
    segend, seglen, pointIdx_a = inc.extractLines(x, y)
    for j in range(segend.shape[0]):
        x_p, y_p = utilities.rotate_segend(segend[j], point, yaw)
        axs[row_index, column_index].plot(x_p, y_p, 'r')
    for t in triangles:
        axs[row_index, column_index].fill(t[:, 0],t[:, 1],fill=False)
```



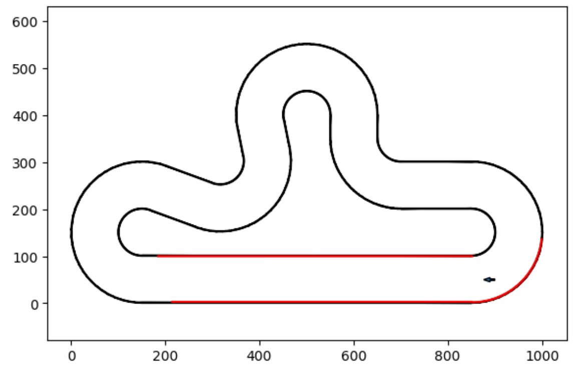
```
from LidarSim.lidar_sim import LidarSimulator
lidar = LidarSimulator("racetrack.stl")
```

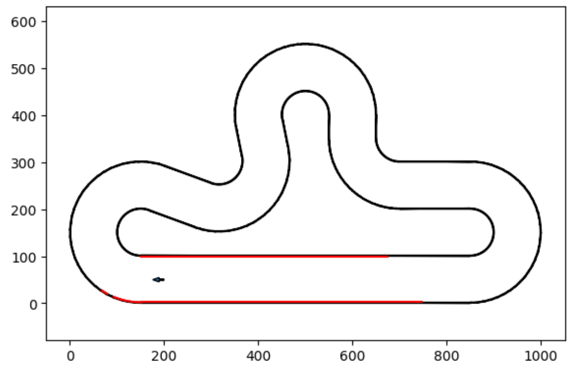
(continues on next page)

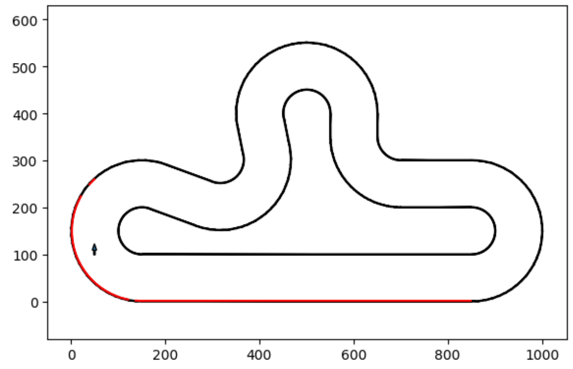
(continued from previous page)

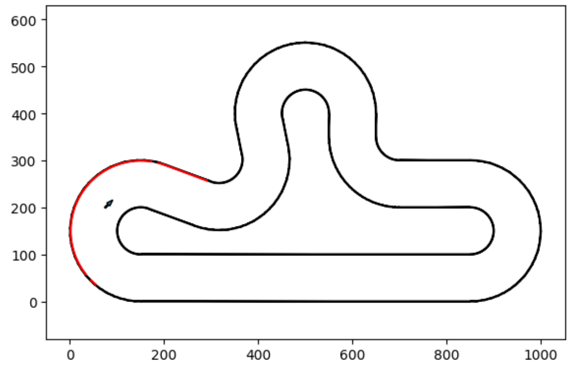
```
triangles = lidar.get_map_triangles()
yaw = np.radians(180)
```

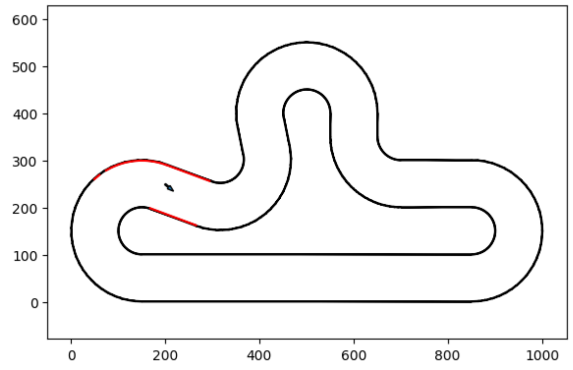
```
pose = [[900, 50, 180 ],
        [200, 50, 180 ],
        [50, 100, 90 ],
        [75, 200, 45 ],
        [200, 250, -40 ],
        [500, 500, 0 ]]

fig, axs = plt.subplots(3, 2, figsize=(15, 15))
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    point = pose[i][0:2]
    yaw = np.radians(pose[i][2])
    plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
    # get carthesian coordinates
    x = []
    y = []
    for alpha, r in plot_scan:
        x.append(r * np.cos(alpha)+ point[0])
        y.append(r * np.sin(alpha) + point[1])

    inc = Incremental()

    axs[row_index, column_index].arrow(point[0], point[1], 10 * np.cos(yaw), 10 * np.
 ↪sin(yaw), width=3.0)
    segend, seglen, pointIdx_a = inc.extractLines(x, y)
    for j in range(segend.shape[0]):
        x_p, y_p = utilities.rotate_segend(segend[j], point, yaw)
        axs[row_index, column_index].plot(x_p, y_p, 'r')
    for t in triangles:
        axs[row_index, column_index].fill(t[:, 0],t[:, 1],fill=False)
```

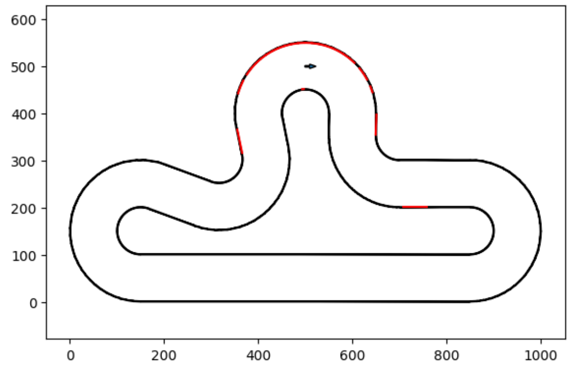

600
500
400
300
200
100
0
0
200
400
600
800
1000


600
500
400
300
200
100
0
0
200
400
600
800
1000


600
500
400
300
200
100
0
0
200
400
600
800
1000


600
500
400
300
200
100
0
0
200
400
600
800
1000


600
500
400
300
200
100
0
0
200
400
600
800
1000


600
500
400
300
200
100
0
0
200
400
600
800
1000

CHAPTER SIXTEEN

ALGORITHM 4: RANSAC

```
1. Initial: let A be a set of N points
2. repeat
3. Randomly select a sample of 2 points from A
4. Fit a line through the 2 points
5. Compute the distances of all other points to this line
6. Construct the inlier set (i.e. count the number of points with distance to the␣
↪line <d)
7. Store these inliers
8. until Maximum number of iterations k reached
9. The set with the maximum number of inliers is chosen as a solution to the problem
```

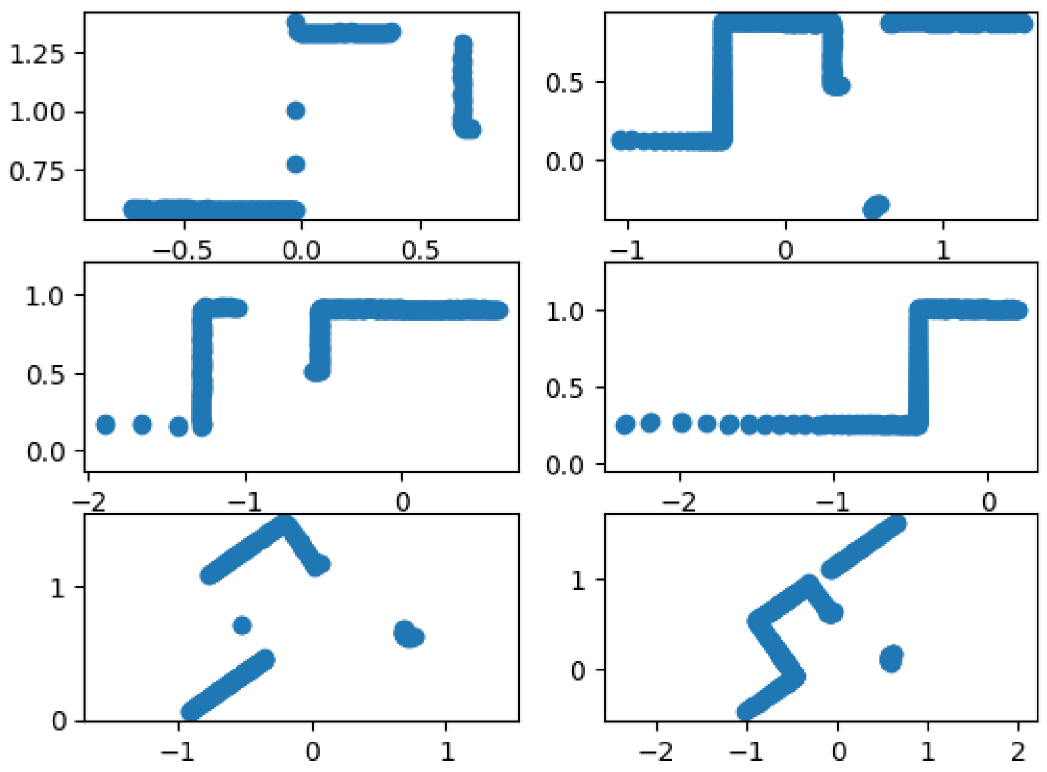
16.1 Reference

Roland Siegwart, Illah Nourbakhsh, and Davide Scaramuzza. Introduction to Autonomous Mobile Robots. MIT Press, 2nd edition, 2011.

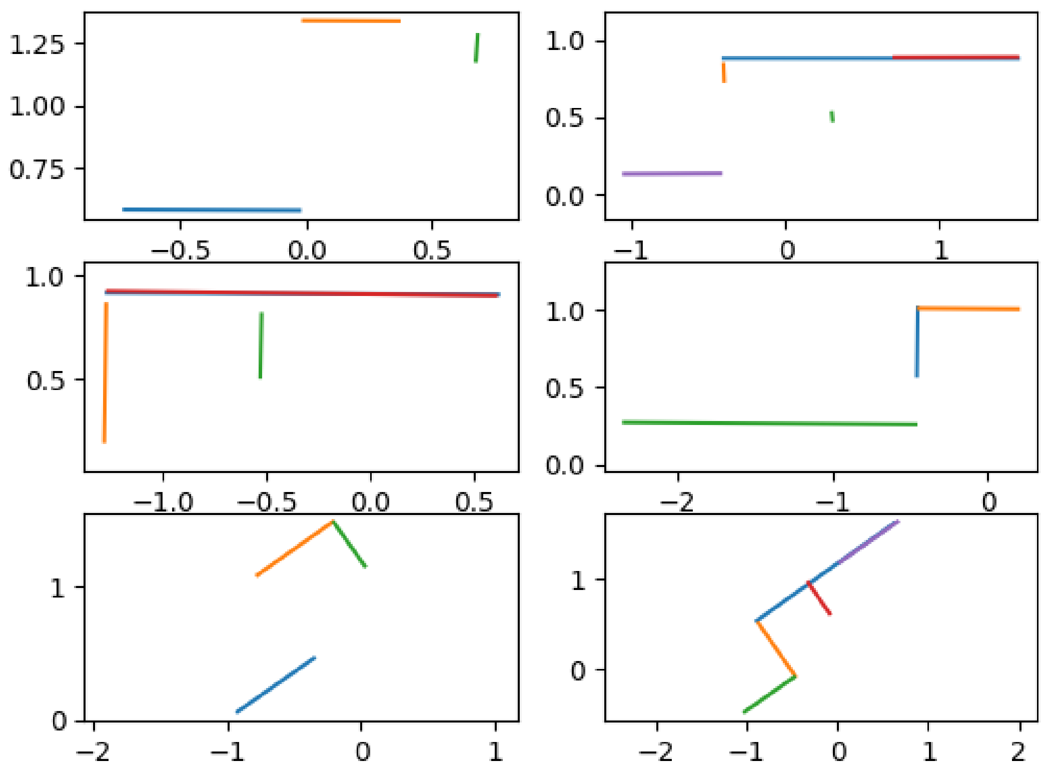
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pickle
from Ransac.ransac import Ransac
```

```
testdata = []
for i in range(6):
    with open('/Users/hdumcke/git/lidar-simulator/jupyternb/data/testLineExtraction%s.
↪mat.pickle' % str(i+1), 'rb') as f:
        testdata.append(pickle.load(f))
```

```
fig, axs = plt.subplots(3, 2)
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    x = testdata[i]['rho'] * np.cos(testdata[i]['theta'])
    y = testdata[i]['rho'] * np.sin(testdata[i]['theta'])
    axs[row_index, column_index].scatter(x, y)
```



```
rs = Ransac(num_iterations=20, dist_threshold=0.005, min_points_per_segment=15)

fig, axs = plt.subplots(3, 2)
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    x = testdata[i]['rho'] * np.cos(testdata[i]['theta'])
    y = testdata[i]['rho'] * np.sin(testdata[i]['theta'])
    segend, seglen, pointIdx_a = rs.extractLines(x[0], y[0])
    for j in range(segend.shape[0]):
        axs[row_index, column_index].plot([segend[j,0], segend[j,2]], [segend[j,1],␣
↪segend[j,3]])
```

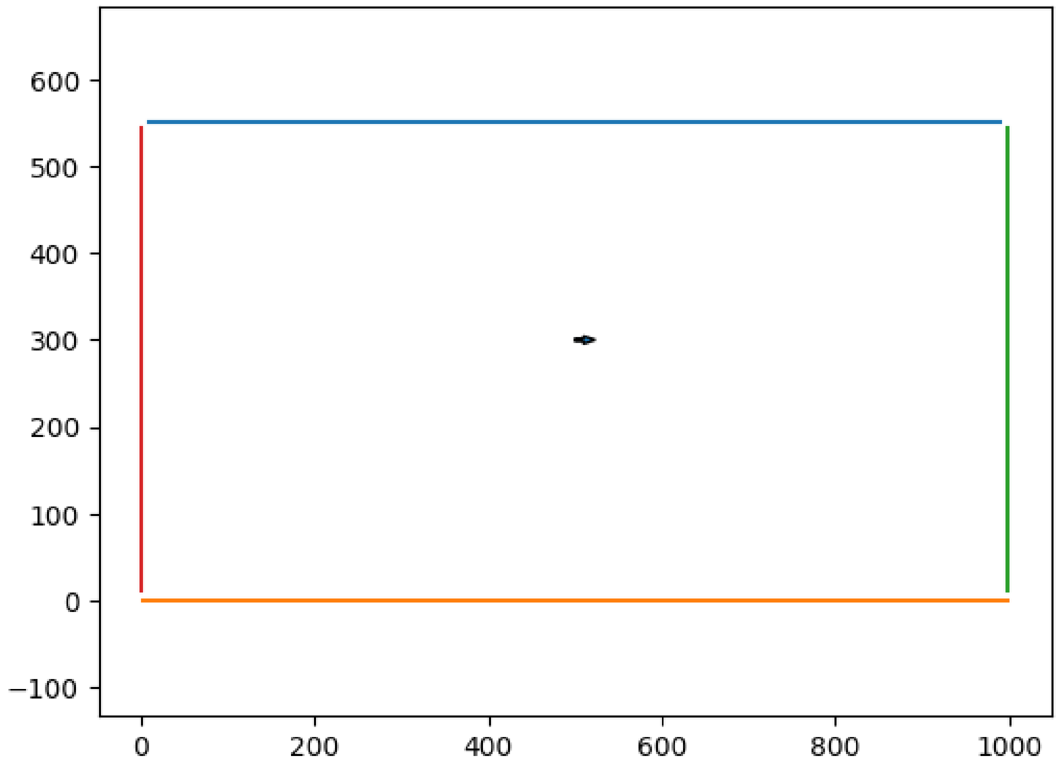


```
from LidarSim.lidar_sim import LidarSimulator
lidar = LidarSimulator("rectangle.stl")
```

```
point = [500, 300]
yaw = np.radians(0)
plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
```

```
# get carthesian coordinates
x = []
y = []
for alpha, r in plot_scan:
    x.append(r * np.cos(alpha)+ point[0])
    y.append(r * np.sin(alpha) + point[1])
```

```
rs = Ransac()

fig,ax = plt.subplots()
ax.axis('equal')
plt.arrow(point[0], point[1], 10 * np.cos(yaw), 10 * np.sin(yaw), width=3.0)
#plt.scatter (x[211:329], y[211:329])
segend, seglen, pointIdx_a = rs.extractLines(x, y)
for j in range(segend.shape[0]):
    ax.plot([segend[j,0], segend[j,2]], [segend[j,1], segend[j,3]])
```



```
import Utilities.utilities as utilities
```

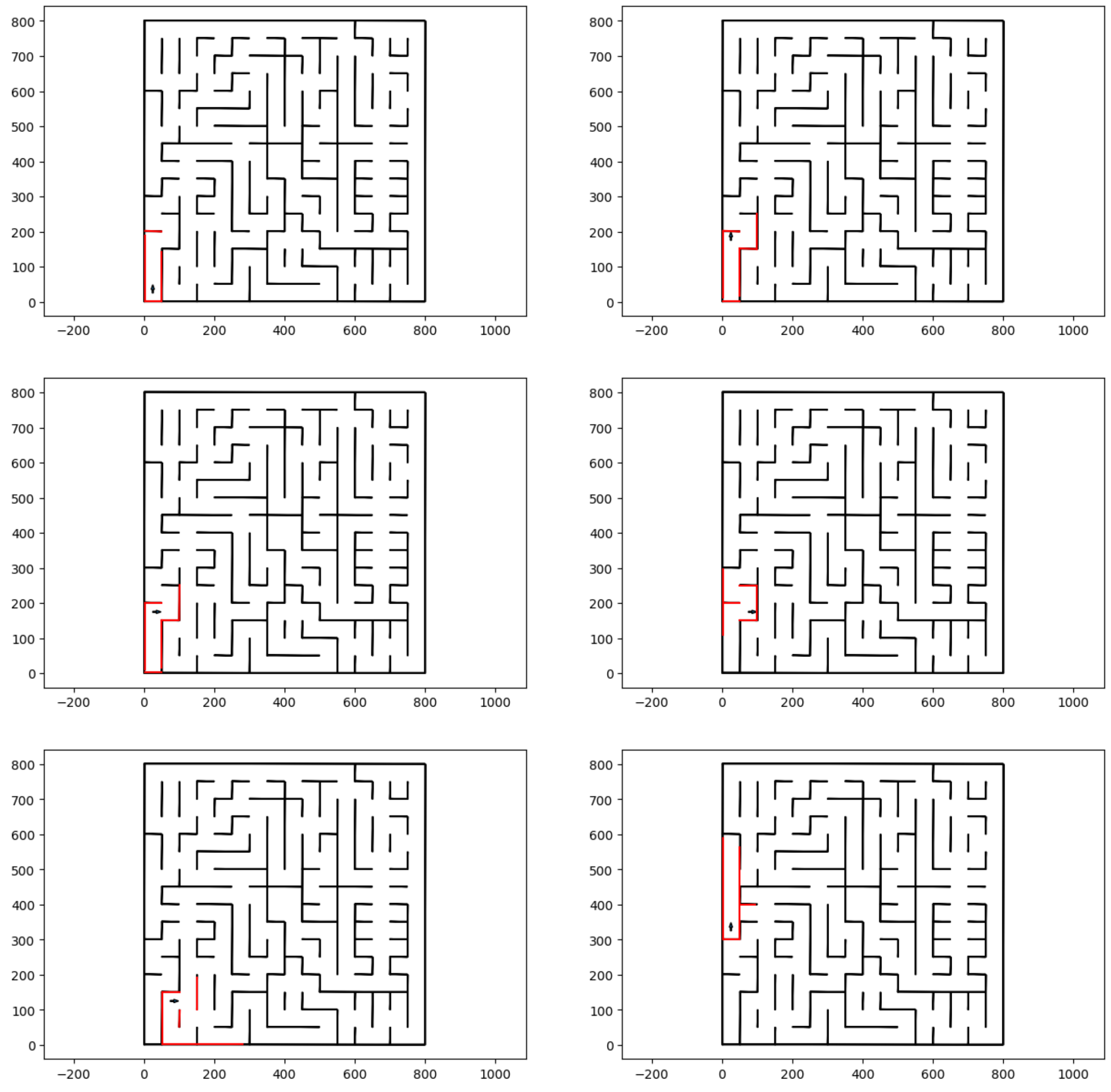
```
from LidarSim.lidar_sim import LidarSimulator
lidar = LidarSimulator("maze.stl")
triangles = lidar.get_map_triangles()
point = [25, 25]
yaw = np.radians(90)
```

```
pose = [[25, 25, 90 ],
        [25, 175, 90 ],
        [25, 175, 0 ],
        [75, 175, 0 ],
        [75, 125, 0 ],
        [25, 325, 90 ]]

fig, axs = plt.subplots(3, 2, figsize=(15, 15))
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    point = pose[i][0:2]
    yaw = np.radians(pose[i][2])
    plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
    # get carthesian coordinates
    x = []
    y = []
    for alpha, r in plot_scan:
        x.append(r * np.cos(alpha)+ point[0])
```

(continues on next page)

(continued from previous page)

```
        y.append(r * np.sin(alpha) + point[1])

    rs = Ransac()

    axs[row_index, column_index].arrow(point[0], point[1], 10 * np.cos(yaw), 10 * np.
↪sin(yaw), width=3.0)
    segend, seglen, pointIdx_a = rs.extractLines(x, y)
    for j in range(segend.shape[0]):
        x_p, y_p = utilities.rotate_segend(segend[j], point, yaw)
        axs[row_index, column_index].plot(x_p, y_p, 'r')
    for t in triangles:
        axs[row_index, column_index].fill(t[:, 0],t[:, 1],fill=False)
```



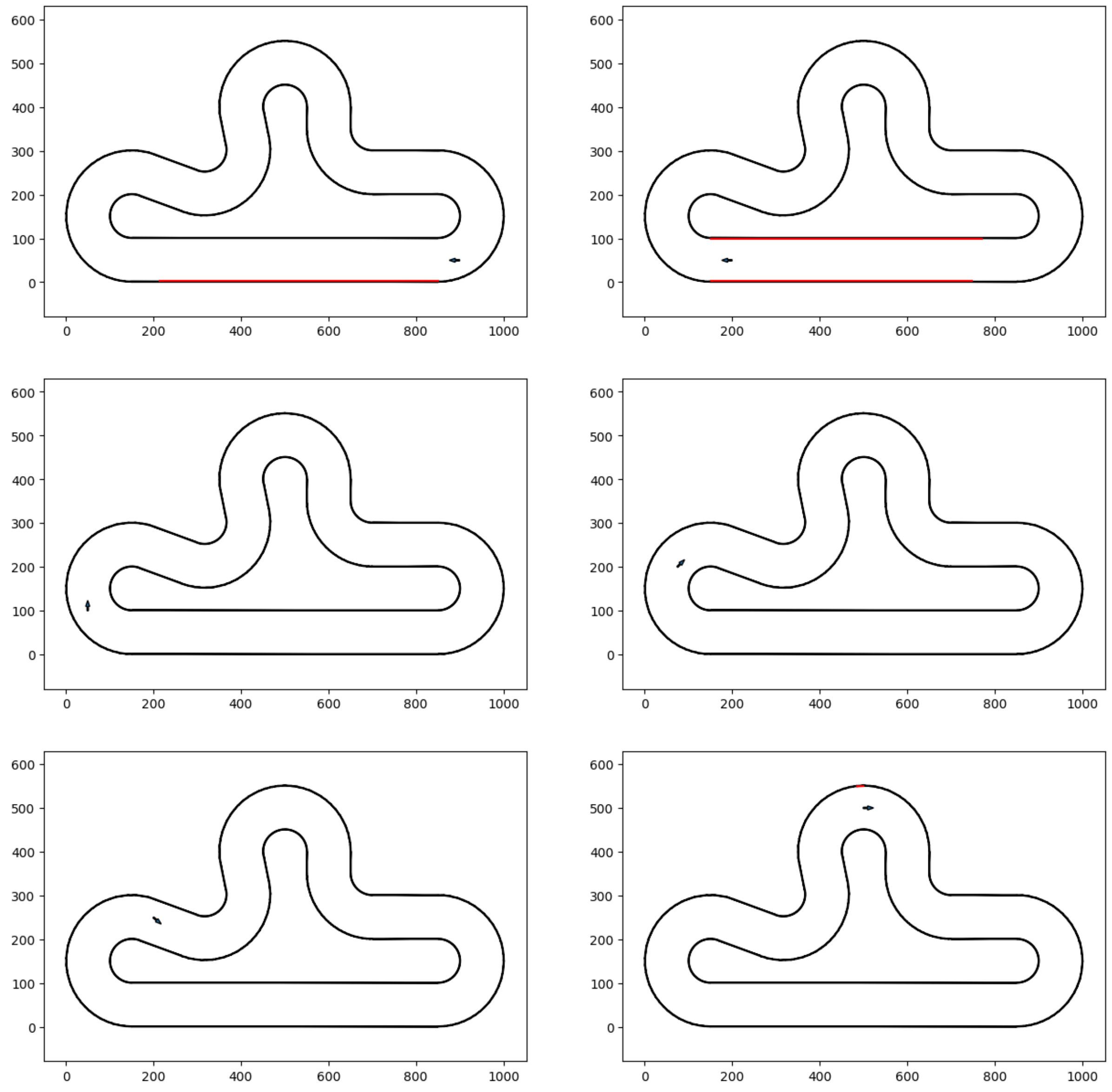
```
from LidarSim.lidar_sim import LidarSimulator
lidar = LidarSimulator("racetrack.stl")
```

(continues on next page)

(continued from previous page)

```
triangles = lidar.get_map_triangles()
yaw = np.radians(180)
```

```
pose = [[900, 50, 180 ],
        [200, 50, 180 ],
        [50, 100, 90 ],
        [75, 200, 45 ],
        [200, 250, -40 ],
        [500, 500, 0 ]]

fig, axs = plt.subplots(3, 2, figsize=(15, 15))
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    point = pose[i][0:2]
    yaw = np.radians(pose[i][2])
    plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
    # get carthesian coordinates
    x = []
    y = []
    for alpha, r in plot_scan:
        x.append(r * np.cos(alpha)+ point[0])
        y.append(r * np.sin(alpha) + point[1])

    rs = Ransac()

    axs[row_index, column_index].arrow(point[0], point[1], 10 * np.cos(yaw), 10 * np.
↪sin(yaw), width=3.0)
    segend, seglen, pointIdx_a = rs.extractLines(x, y)
    for j in range(segend.shape[0]):
        x_p, y_p = utilities.rotate_segend(segend[j], point, yaw)
        axs[row_index, column_index].plot(x_p, y_p, 'r')
    for t in triangles:
        axs[row_index, column_index].fill(t[:, 0],t[:, 1],fill=False)
```



16.2 Issues with Ransac

Ransac finds infinite lines. If two segments are on the same line they are considered the same as we can see with the maze example

Ransac is also non deterministic due to the random sample selection. Run the examples several times to see the changes

CHAPTER

SEVENTEEN

ALGORITHM 5: HOUGH TRANSFORM

```
1. Initial: let A be a set of N points
2. Initialize the accumulator array by setting all elements to 0
3. Construct values for the array
4. Choose the element with max. votes V_max
5. If V_max is less than a threshold,terminate
6. Otherwise, determine the inliers
7. Fit a line through the inliers and store the line
8. Remove the inliers from the set, go to step 2
```

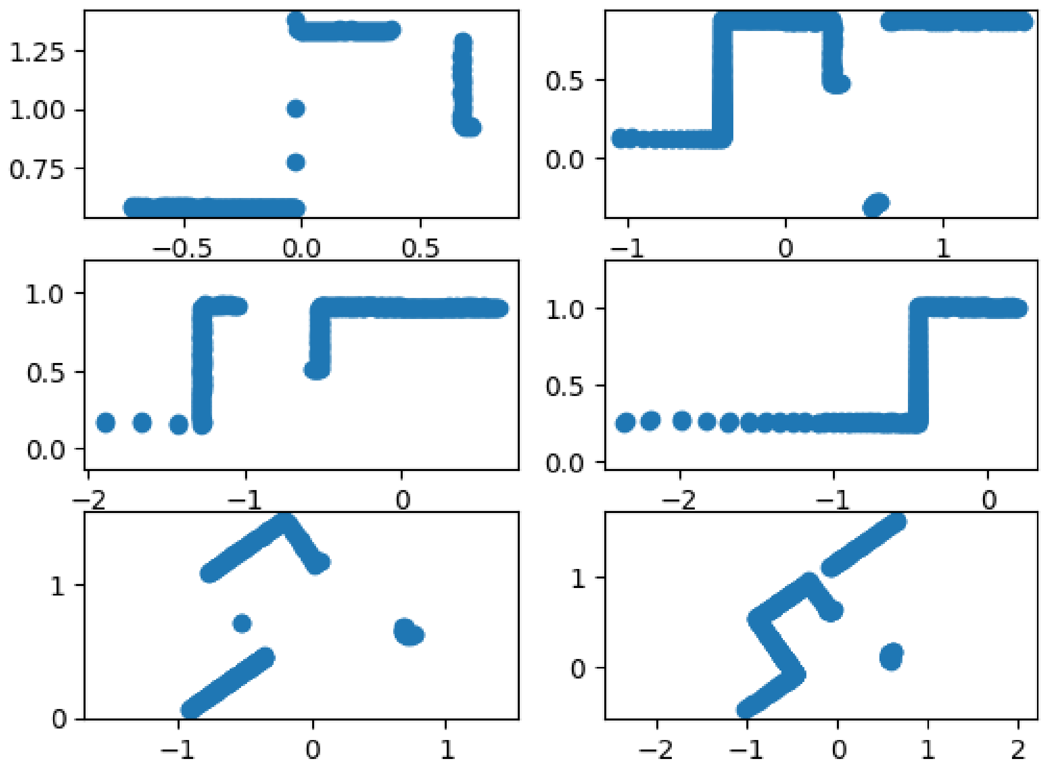
17.1 Reference

Roland Siegwart, Illah Nourbakhsh, and Davide Scaramuzza. Introduction to Autonomous Mobile Robots. MIT Press, 2nd edition, 2011.

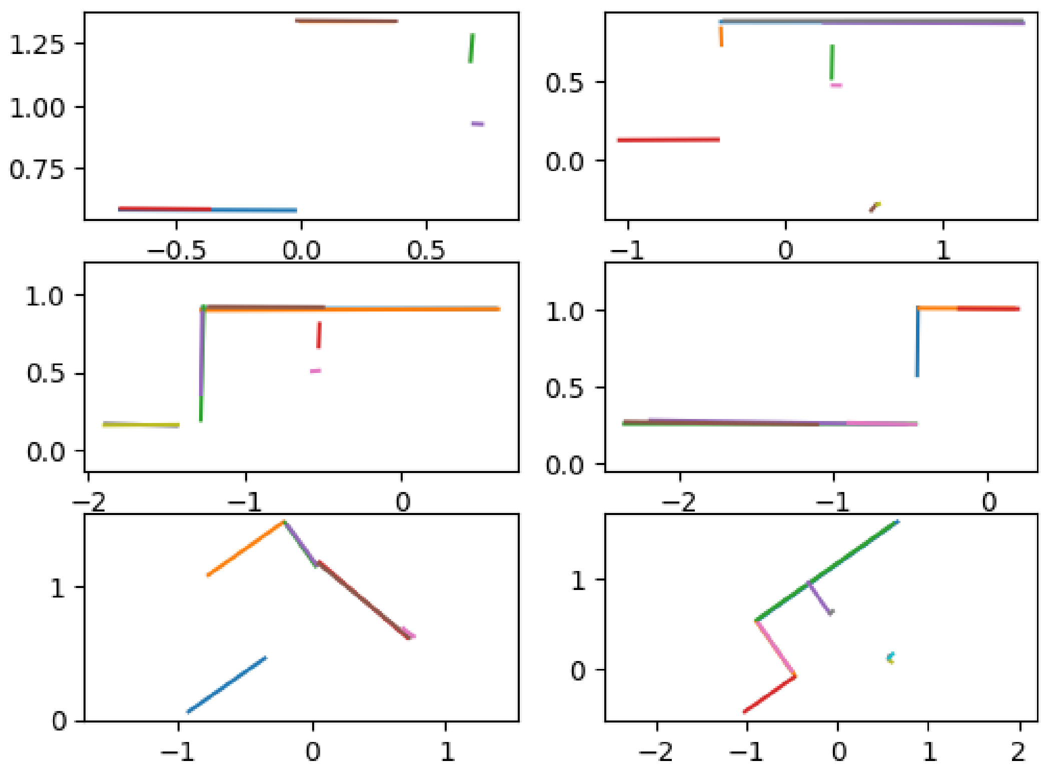
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pickle
from HoughTransform.hough_transform import HoughTransform
```

```
testdata = []
for i in range(6):
    with open('/Users/hdumcke/git/lidar-simulator/jupyternb/data/testLineExtraction%s.
↪mat.pickle' % str(i+1), 'rb') as f:
        testdata.append(pickle.load(f))
```

```
fig, axs = plt.subplots(3, 2)
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    x = testdata[i]['rho'] * np.cos(testdata[i]['theta'])
    y = testdata[i]['rho'] * np.sin(testdata[i]['theta'])
    axs[row_index, column_index].scatter(x, y)
```



```
ht = HoughTransform()

fig, axs = plt.subplots(3, 2)
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    x = testdata[i]['rho'] * np.cos(testdata[i]['theta'])
    y = testdata[i]['rho'] * np.sin(testdata[i]['theta'])
    segend, seglen, pointIdx_a = ht.extractLines(x[0], y[0])
    for j in range(len(segend)):
        axs[row_index, column_index].plot([segend[j][0], segend[j][2]], [segend[j][1],
 ↪ segend[j][3]])
```

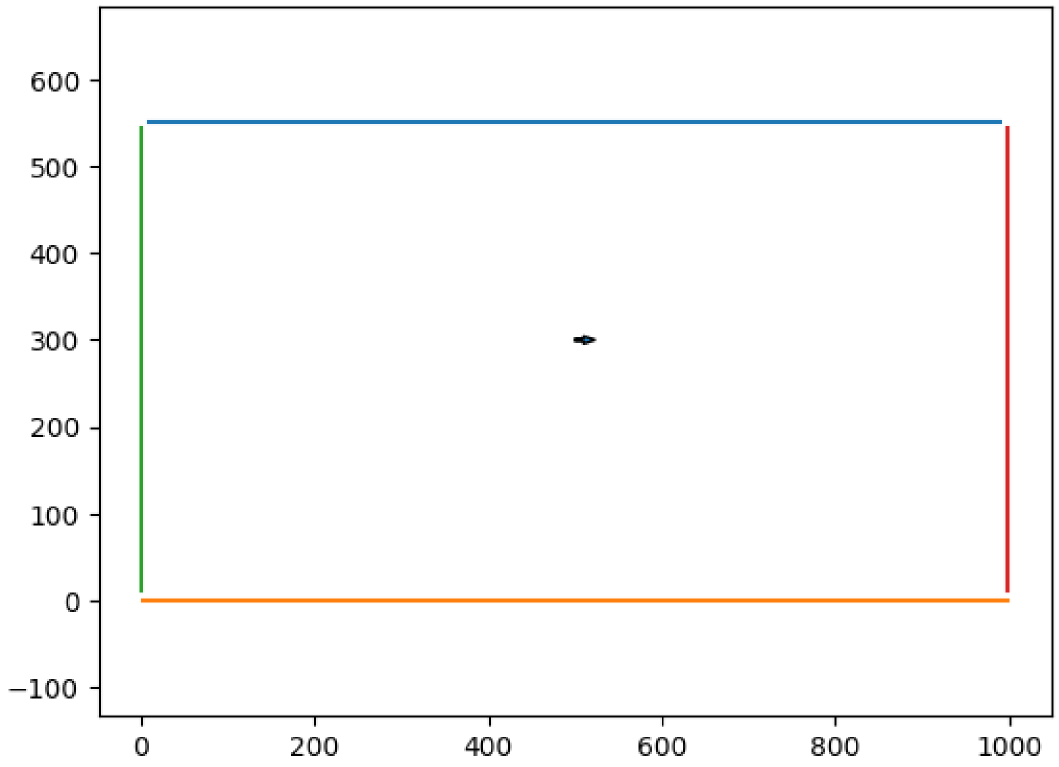


```
from LidarSim.lidar_sim import LidarSimulator
lidar = LidarSimulator("rectangle.stl")
```

```
point = [500, 300]
yaw = np.radians(0)
#plot_scan = lidar.get_lidar_points(point[0], point[1], yaw, theta=0, view_range=25)
plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
```

```
# get carthesian coordinates
x = []
y = []
for alpha, r in plot_scan:
    x.append(r * np.cos(alpha)+ point[0])
    y.append(r * np.sin(alpha) + point[1])
```

```
ht = HoughTransform(rho_samples=200000)

fig,ax = plt.subplots()
ax.axis('equal')
plt.arrow(point[0], point[1], 10 * np.cos(yaw), 10 * np.sin(yaw), width=3.0)
#plt.scatter (x[211:329], y[211:329])
segend, seglen, pointIdx_a = ht.extractLines(x, y)
for j in range(len(segend)):
    ax.plot([segend[j][0], segend[j][2]], [segend[j][1], segend[j][3]])
```



```
import Utilities.utilities as utilities
```

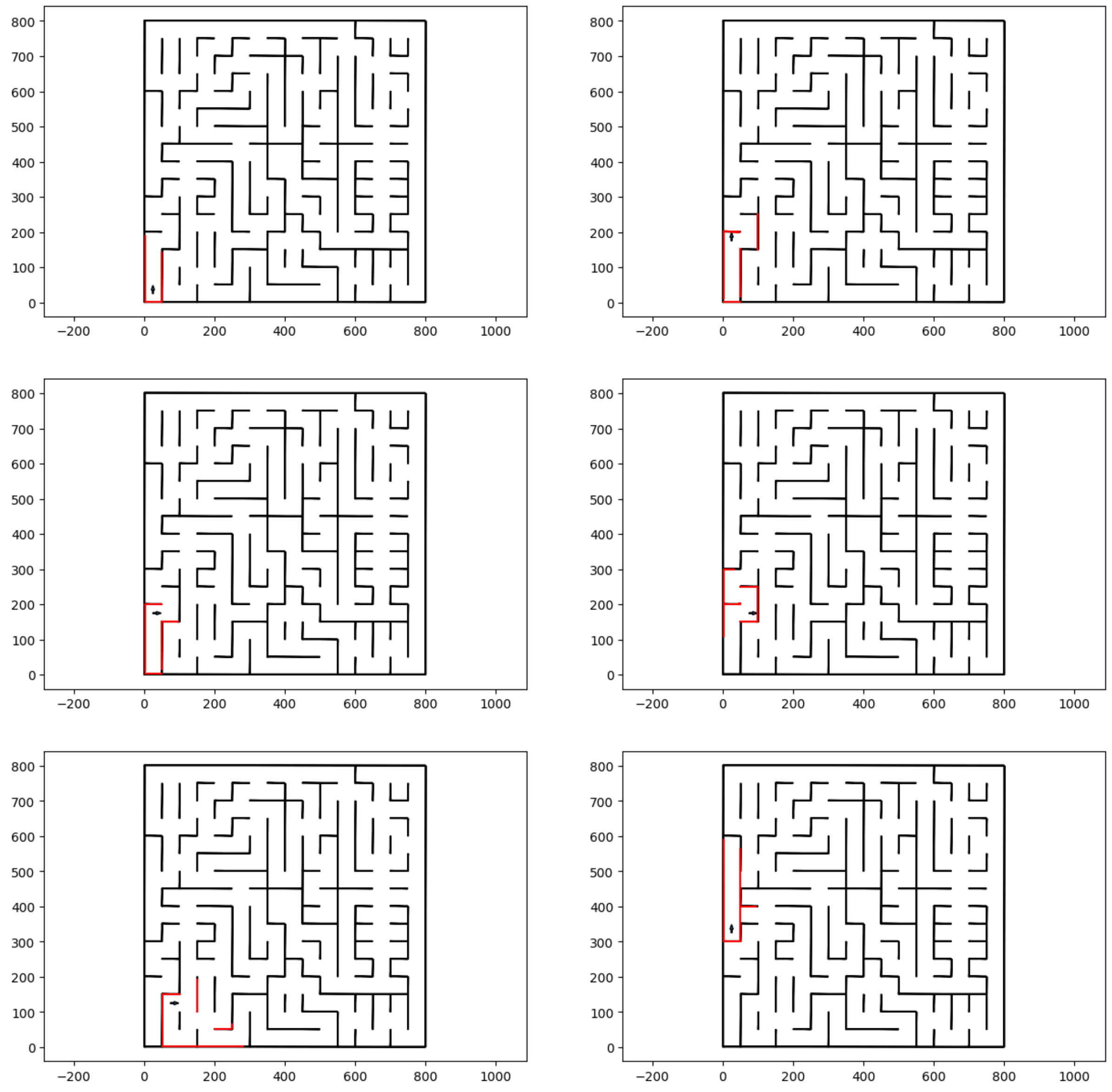
```
from LidarSim.lidar_sim import LidarSimulator
lidar = LidarSimulator("maze.stl")
triangles = lidar.get_map_triangles()
point = [25, 25]
yaw = np.radians(90)
```

```
pose = [[25, 25, 90 ],
        [25, 175, 90 ],
        [25, 175, 0 ],
        [75, 175, 0 ],
        [75, 125, 0 ],
        [25, 325, 90 ]]

fig, axs = plt.subplots(3, 2, figsize=(15, 15))
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    point = pose[i][0:2]
    yaw = np.radians(pose[i][2])
    plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
    # get carthesian coordinates
    x = []
    y = []
    for alpha, r in plot_scan:
        x.append(r * np.cos(alpha)+ point[0])
```

(continues on next page)

(continued from previous page)

```
        y.append(r * np.sin(alpha) + point[1])

    ht = HoughTransform(rho_samples=200000)

    axs[row_index, column_index].arrow(point[0], point[1], 10 * np.cos(yaw), 10 * np.
↪sin(yaw), width=3.0)
    segend, seglen, pointIdx_a = ht.extractLines(x, y)
    for j in range(len(segend)):
        x_p, y_p = utilities.rotate_segend(segend[j], point, yaw)
        axs[row_index, column_index].plot(x_p, y_p, 'r')
    for t in triangles:
        axs[row_index, column_index].fill(t[:, 0],t[:, 1],fill=False)
```



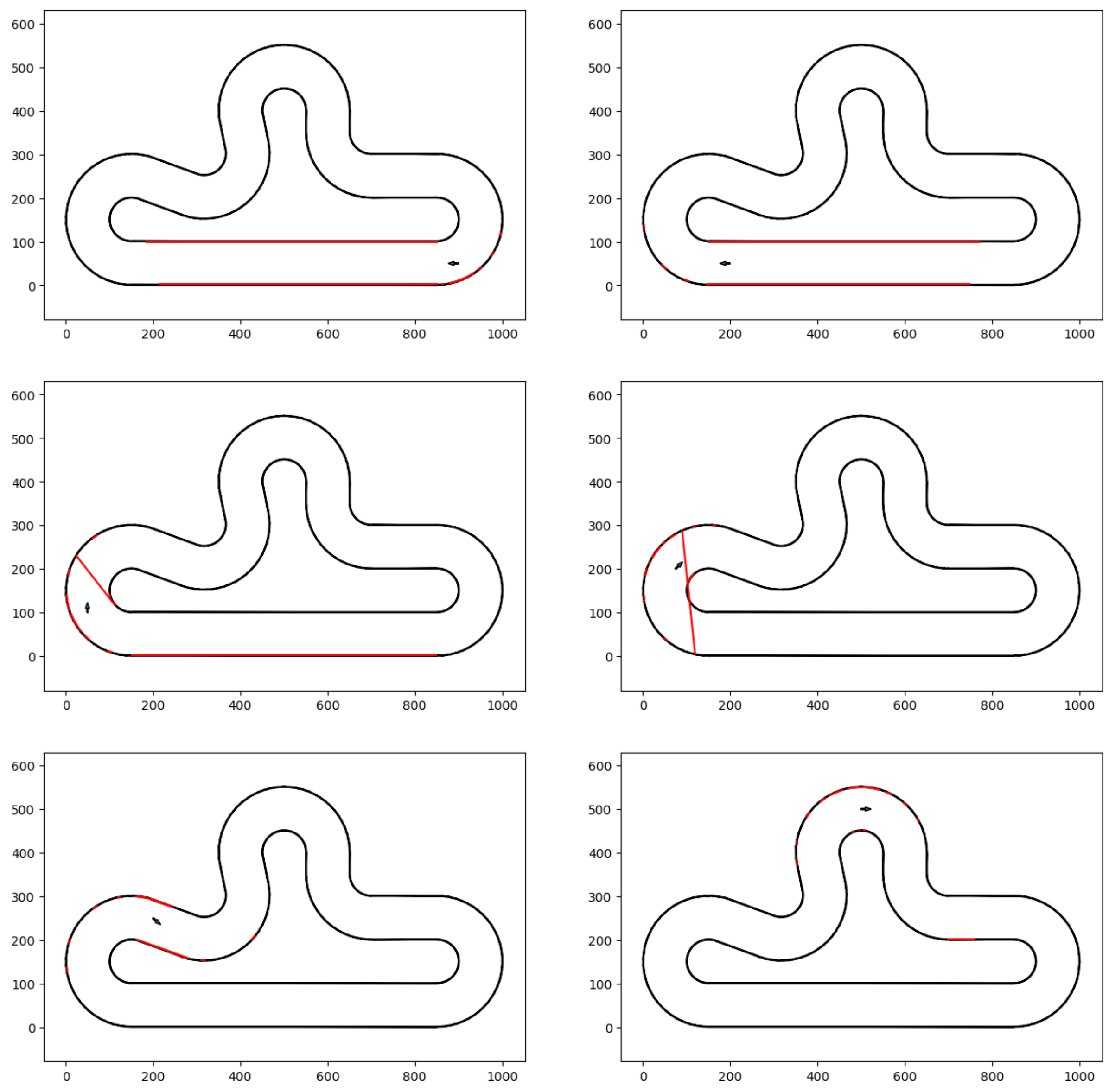
```
from LidarSim.lidar_sim import LidarSimulator
lidar = LidarSimulator("racetrack.stl")
```

(continues on next page)

(continued from previous page)

```
triangles = lidar.get_map_triangles()
yaw = np.radians(180)
```

```
pose = [[900, 50, 180 ],
        [200, 50, 180 ],
        [50, 100, 90 ],
        [75, 200, 45 ],
        [200, 250, -40 ],
        [500, 500, 0 ]]

fig, axs = plt.subplots(3, 2, figsize=(15, 15))
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    point = pose[i][0:2]
    yaw = np.radians(pose[i][2])
    plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
    # get carthesian coordinates
    x = []
    y = []
    for alpha, r in plot_scan:
        x.append(r * np.cos(alpha)+ point[0])
        y.append(r * np.sin(alpha) + point[1])

    segend, seglen, pointIdx_a = ht.extractLines(x, y)

    axs[row_index, column_index].arrow(point[0], point[1], 10 * np.cos(yaw), 10 * np.
↪sin(yaw), width=3.0)
    for j in range(len(segend)):
        x_p, y_p = utilities.rotate_segend(segend[j], point, yaw)
        axs[row_index, column_index].plot(x_p, y_p, 'r')
    for t in triangles:
        axs[row_index, column_index].fill(t[:, 0],t[:, 1],fill=False)

    #axs[row_index, column_index].scatter (x, y)
```



17.2 Issues with Hough Transform

The Hough Transform finds infinite lines. If two segments are on the same line they are considered the same as we can see with the maze example

The Hough Transform alows allows to detect circles. In the case of the race track we know that we are only looking for circles with a radius of either 50 or 150 which simplifies the parameter stace significantly.

For the circle detection I used https://github.com/PavanGJ/Circle-Hough-Transform.git as inspiration

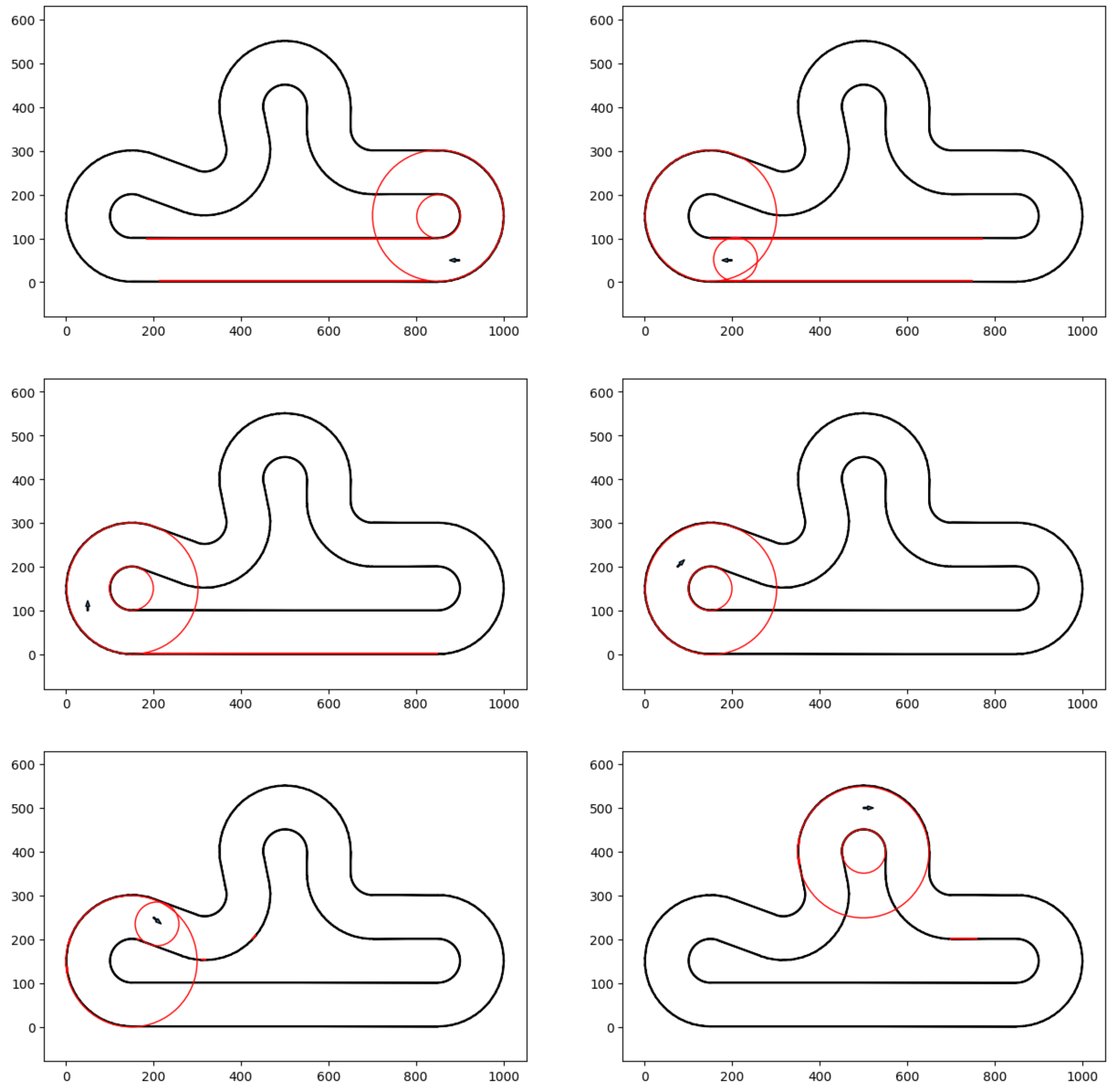
More work would be required to make this usefull like filtering relevant arc of the circles and detecting smaller lines when exiting a circle

```
pose = [[900, 50, 180 ],
        [200, 50, 180 ],
```

(continues on next page)

(continued from previous page)

```
        [50, 100, 90 ],
        [75, 200, 45 ],
        [200, 250, -40 ],
        [500, 500, 0 ]]

circles = [[], [], [], [], [], []]

fig, axs = plt.subplots(3, 2, figsize=(15, 15))
for i in range(6):
    row_index = int(i / 2)
    column_index = i % 2
    axs[row_index, column_index].axis('equal')
    point = pose[i][0:2]
    yaw = np.radians(pose[i][2])
    plot_scan = lidar.get_lidar_points(point[0], point[1], yaw)
    # get carthesian coordinates
    x = []
    y = []
    for alpha, r in plot_scan:
        x.append(r * np.cos(alpha)+ point[0])
        y.append(r * np.sin(alpha) + point[1])

    segend, seglen, pointIdx_a = ht.extractCircles(x, y)

    axs[row_index, column_index].arrow(point[0], point[1], 10 * np.cos(yaw), 10 * np.
↪sin(yaw), width=3.0)
    for j in range(len(segend)):
        if len(segend[j]) == 3:
            x_c, y_c = utilities.rotate_segend([segend[j][1], segend[j][2], 0, 0],␣
↪point, yaw)
            circles[i].append(plt.Circle((x_c[0],y_c[0]),segend[j][0],color=(1,0,0),
↪fill=False))
        else:
            x_p, y_p = utilities.rotate_segend(segend[j], point, yaw)
            axs[row_index, column_index].plot(x_p, y_p, 'r')
    for t in triangles:
        axs[row_index, column_index].fill(t[:, 0],t[:, 1],fill=False)
    for c in circles[i]:
        axs[row_index, column_index].add_patch(c)

    #axs[row_index, column_index].scatter (x, y)
```


600
500
400
300
200
100
0
0
200
400
600
800
1000

CHAPTER EIGHTEEN

ALGORITHM 6: EXPECTATION MAXIMIZATION

```
1.  Initial: let A be a set of N points
2.  repeat
3.    Randomly generate parameters for a line
4.    Initialize weights for remaining points
5.    repeat
6.      E-Step: Compute the weights of the points from the line model
7.      M-Step: Recompute the line model parameters
8.     until Maximum number of steps reached or convergence
9.  until Maximum number of trials reached or found a line
10. If found, store the line, remove the inliers, go to step 2
11  Otherwise, terminate
```

18.1 Reference

Forsyth, D. A., Ponce, J., Computer Vision: A Modern Approach. Upper Saddle River, NJ, Prentice Hall, 2003.

18.2 ToDO

CHAPTER

NINETEEN

RANGE HISTOGRAM FEATURES

19.1 ToDO

CHAPTER

TWENTY

LOCALIZATION

20.1 ToDo

CHAPTER

TWENTYONE

ODOMETRY SIMULATOR

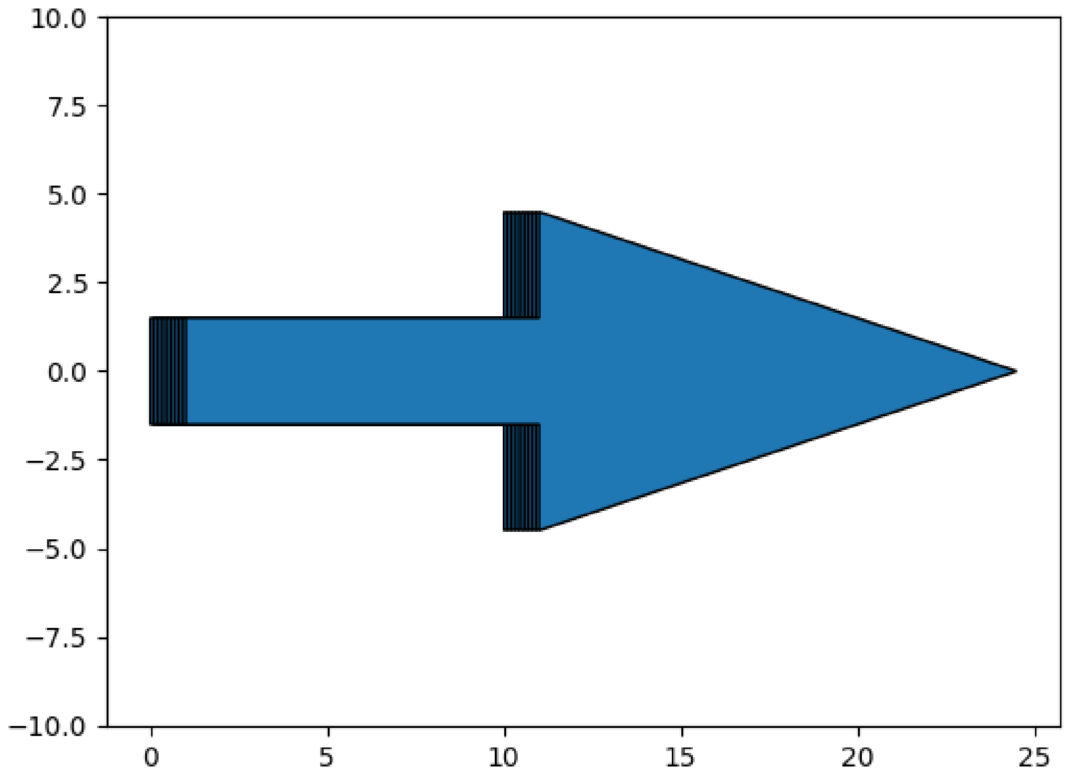
21.1 ToDo

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import Utilities.utilities as utilities
```

```
pose = [0.0, 0.0, 0.0]
cmd_vel = {}
cmd_vel["linear_x"] = 1.0
cmd_vel["linear_y"] = 1.0
cmd_vel["angular_z"] = 0.0
dt = 0.1
```

```
fig,ax = plt.subplots()
position = pose
ax.axis('equal')
plt.arrow(position[0], position[1], 10 * np.cos(position[2]), 10 * np.
 ↪sin(position[2]), width=3.0)
for i in range(10):
    position = utilities.get_odom(cmd_vel, position, dt)
    print(position)
    plt.arrow(position[0], position[1], 10 * np.cos(position[2]), 10 * np.
 ↪sin(position[2]), width=3.0)
```

```
[0.1, 0.0, 0.0]
[0.2, 0.0, 0.0]
[0.30000000000000004, 0.0, 0.0]
[0.4, 0.0, 0.0]
[0.5, 0.0, 0.0]
[0.6, 0.0, 0.0]
[0.7, 0.0, 0.0]
[0.7999999999999999, 0.0, 0.0]
[0.8999999999999999, 0.0, 0.0]
[0.9999999999999999, 0.0, 0.0]
```


10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
0
5
10
15
20
25