Project 4: Mapping

Implement the project by modifying *project_4* package provided to you and changing the name to look like ***<my_directory_id>_project_4*** (the package is further referred as 'project_4' for simplicity). Hand in this project by uploading the package via the ELMS website. Generally, the only three things you need to modify to change the package name - the name of the folder, *package.xml* and *CMakeLists.txt*

Most of the guidelines (as well as starter code) are designed for Python. C++ developers will get some additional extra credit (+20%, as usual) for their implementations.

DELIVERABLES:

- ***<my_directory_id>_project_4*** - package with your code
- A screenshot of your map built in rviz with resolution of 0.05

This time you will have to build a map of the robot's environment using laser scanner data collected from a moving robot.

1. Coordinate conversion

Our map discretizes the world into a grid of cells. The relation between the grid and the continuous world is defined by five parameters:

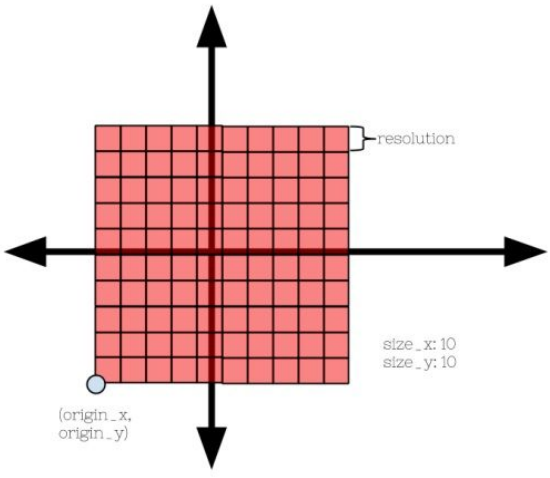


Figure 1: Map parameters

- *origin_x* and *origin_y* - defining the bottom left corner of the grid (in world coordinates)

- *resolution* - the (square) size of each grid cell (world units)
- *size_x* and *size_y* - the number of grid cells in each dimension.

In order to create the map of the environment we need to know how to convert between world coordinates and grid coordinates. In *project_4/src/project_4/geometry.py* write the following two functions:

- Convert the coordinate (x, y) (a pair of floats in world coordinates) by returning a pair of integer coordinates in the grid. If the coordinates are outside the defined grid, the function should return *None*. You can use *int(x)* to convert a float to an integer.

  def to_grid(x, y, origin_x, origin_y, size_x, size_y, resolution)

- Convert a pair of integers in grid coordinates *(gx, gy)* by returning a pair of float coordinates representing the center of the grid cell in world coordinates.

  def to_world(gx, gy, origin_x, origin_y, size_x, size_y, resolution)

Note that if your coordinate conversion functions don't work correctly you won't be able to complete the second part of the project. You can test your functions using the following script:

rosrun project_4 coordinate_test.py

2. Map building

To build the map you will use data from a scanning laser range finder sensor or just laser scanner for short. This sensor measures the distance to the objects in the environment by sweeping a laser in a circular arc. The sweeping plane is parallel to the ground. Range measurements are made at constant angular steps. The properties of the scanner are defined by the following parameters:

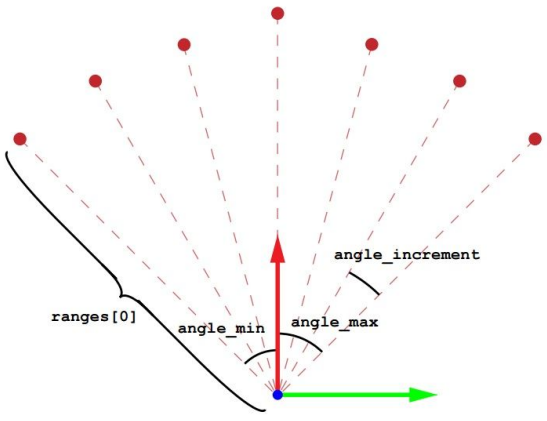
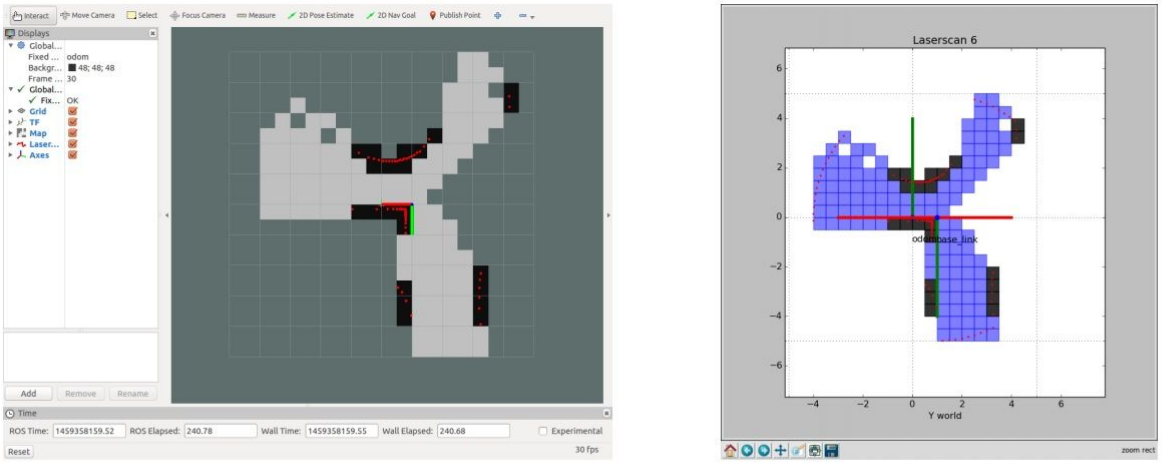


Figure 2: Parameters of a laser scanner. Red and green arrows represent the X and Y axes of the coordinate frame of the laser scanner respectively. *ranges* is an array of distance measurements returned by the scanner.

- *angle_min* and *angle_max* - defining the circular arc
- *angle_increment* - the angular step between consecutive laser rays
- *range_max* - maximum distance from the scanner to an obstacle

In ROS these parameters along with the distance readings are stored in the *LaserScan* messages. Data from the laserscanner allows us to divide the space around the robot into occupied space (endpoints of the rays) and free space (space along each ray). Since the scans are collected from a moving robot, in order to fuse them into a map they need to be oriented correctly relative to each other. To do this you can use the odometry information from the robot. Although rarely true in real applications, for this project you can assume that the origin of the laser is at the coordinates indicated by the odometry.



(a) (b)

Figure 3: Visualization of the mapping process (a) in `rviz` and (b) using `visualize_scans` method.

Fill in the details of the *MapMaker* class in *project_4/src/project_4/map_maker.py* to create a map from *Odometry* and *LaserScan* messages. Whenever you get a laser reading, use the most recent odometry message (and the coordinate conversion methods from *part 1*) to clear out the empty space in the map (by changing the cell value to *0*) and mark the occupied cells (with *100*). All cells are initially *-1* i.e. unknown space. Note that you should not mark cells as occupied if the range is greater than or equal to the *range_max* of the laser scanner. The map will be built in the *OccupancyGrid* message. Instead of a two dimensional grid, the data is stored in a one-dimensional vector (*OccupancyGrid.data*).

The following functions are already provided for you in *geometry.py* and *map_maker.py*

- *to_index(x, y, size_x)* converts a pair of integer grid coordinates to grid index.
- *bresenham* finds coordinates of grid cells occupied by a line in the map.
- *visualize_scans* visualizes robot pose, laser rays and cell occupancy information. Use it for debugging your code.

To test your code use the prerecorded bag file *project_4/Mapping1.bag* containing odometry and laserscan messages collected from a robot. To test your code use *project_4/src/ros_map.py* script which automatically reads the bag file and calls your functions from *map_maker.py* to process the messages and build the map. You can also change the resolution of the map with the *-r* argument. By default it is *0.5* meters. Try *0.25* and *0.05*.

rosrun project_4 ros_map.py Mapping1.bag -r 0.05

You can visualize the map building process using

run rosrun rviz rviz -d project_4/map.rviz

which will display the laser data as well as the map as you build it. Note that rviz will show laserscans that are already aligned to the robot pose. However, to build the map you need to do the alignment yourself in your *process_scan* method. Generate a map at resolution of 0.05 and submit the screenshot of the map from rviz along with your *project_4* package.

TIP: use the following code to convert the orientation component of an *Odometry* message to Euler angles:

```
orientation = msg.pose.pose.orientation
orientation = (orientation.x, orientation.y, orientation.z, orientation.w)
theta = euler_from_quaternion(orientation)[2]
```

3. Grading

to_grid	20 points
to_world	20 points
Scan alignment	30 points
Map building	30 points