THE NEOTIA UNIVERSITY

ज्ञानम् आत्म प्रदीपाय

DEPARTMENT OF ROBOTICS & AUTOMATION

Localization Techniques

LAB MANUAL

THE NEOTIA UNIVERSITY
DEPARTMENT OF ROBOTICS & AUTOMATION

EXPERIMENT NO.: 1

NAME OF THE EXPERIMENT: Two Programming Exercises for Robots.

OBJECTIVE: To perform the Robot programming exercise for Pick and Place operation.

THEORY:

Most controllers for industrial robots provide a method of dividing a program into one or more branches. Branching allows the robot program to be subdivided into convenient segments that can be executed during the program. A branch can be thought of as a sub routine that is called one or more times during the program. The subroutine can be executed either by branching to it at a particular place in the program or by testing an input signal line to branch to it. The amount of decision logic that can be incorporated into a program varies widely with controllers. They permit the use of an incoming signal to invoke a branch. Most controllers allow the user to specify whether the signal should be interrupt the program branch currently being executed, or wait until the current branch completes. The interrupt capability is typically used for error branches. An error branch is invoked when an incoming signal indicates that some abnormal event has occurred. Depending on the event and the design of the error branch, the robot will either take some corrective action or simply terminate the robot motion and signal for human assistance.

A frequent use of the branch capability is when the robot has been programmed to perform more than one task. In this case, separate branches are used for indicating which branch of the program must be executed and when it must be executed. A common way of accomplishing this is to mate use of external signals which are activated by sensors or other interlocks. The device recognizes which took must be performed, and provide the appropriate signal to all that branch. This method is frequently used or spray painting robot which have been programmed to paint a limited variety of parts moving post the workstation of a conveyor photoelectric cells are frequently employed to identify the part of to be sprayed by distinguishing between the geometric features of different parts. The photoelectric cells are used

to generate the signal to the robot to call the spray painting sub routine corresponding to the particular part.

Robot programs have thus far been discussed as consisting of a series of points in space, where each point is designed as a set of joint coordinate corresponding to the number of degree of freedom of robot. These points are specified as in absolute coordinates. That is when the robot executes program; each point is visited at exactly the same location every time. The new concept involves the use of a reloatable branch.

A relocated branch allows the programmer to specify a branch involving a set of internal points in space that are performed relative to some defined starting point for the branch. This would permit the same motion subroutine to be performed at various locations in the workspace of the robot. Many industrial robot shave the capacity to accept reload able branches as a part of program. The programmer indicates that a relocatable branch will be defined and the controller records relative or Incremental motion points rather than absolute points.

PROGRAM 1:

POINT NAME	EXPLANATION
SAFE	Safe location to start and stop
PICK UP	Location of part pick-up and of chute
INTER	Intermediate point above chute to pass through.
Loc 1	Location of first pallet position
Loc 2	
……Loc 24	Location of 24^{th} pallet position
ABOVE 1	Location above 1^{st} pallet position
…..ABOVE 24	Location above 24^{th} pallet position.

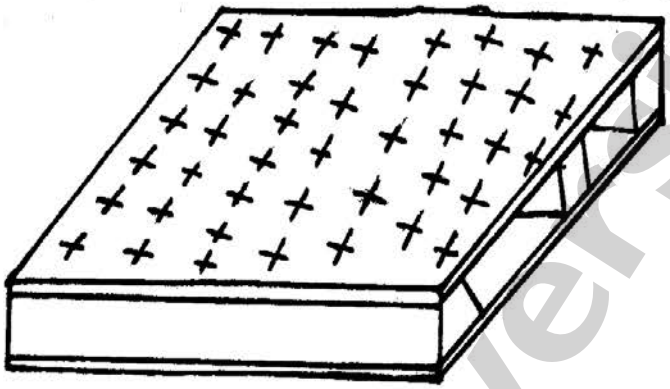


fig:- Pallet with 24 positions used to Illustrate branching in robot program.

Suppose that the operation required the robot to pick up parts from an input chute and place them on a pallet with 24 positions. When a start signal is given, the robot must begin picking up parts and loading them into the pallet, continuing until all 24 positions on the pallet is filled. The robot must then generate a signal to indicate that the pallet is full, and wait for the start signal to being the next cycle. When the robot is directed to go to the point name in the program, it goes to the associated joint coordinates. In creating robot programs for palletizing operations of this type, the robot is programmed to approach a given part from a direction choose to avoid interference with the other parts.

The speed at which the program is executed should be varied during the program when the gripper is approaching a pick up of drop off point, the speed setting should be at a relatively slow value. When the robot moves larger distance the chute and the pallet, higher speed would be programmed.

PROGRAM 2:

Program for Pick and Place activity:

STATEMENT	STATEMENT DESCRIPTION
BRANCH PICK	The branch of program indicating part picks.
MOVE INTER	Move to an intermediate position chute.
WAIT 12	Wait for an incoming part to chute.
SIGNAL 5	Open gripper fingers (Sensor control)
MOVE PICK-UP	Move gripper and Pick-up the object.
SIGNAL 6	Close the gripper to grasp the object.
MOVE INTER	Depart to intermediate position above chute.
END BRANCH	End of pick-up activity.
BRANCH PLACE	Start of placing activity.
MOVE Z (-50)	Position part and gripper above the pallet .
SIGNAL 5	Open gripper to release the part.
MOVE Z (50)	Depart from the place point.
END BRANCH	End of place activity.

CONCLUSION: Thus, we have studied how to perform Pick and Place operation.

THE NEOTIA UNIVERSITY
DEPARTMENT OF ROBOTICS & AUTOMATION

EXPERIMENT NO.: 2

NAME OF THE EXPERIMENT: Exercise on Robotic Simulation Software.

OBJECTIVE: To study the Robot path planning using Robotic simulation software.

THEORY:

The locus of points along the path defines the sequence of position through which the robot will move its wrist. In most applications, an end effector is attached to the wrist and program can be considered to be the path in space through which the end effector is to be moved by the robot.

Since, the robot consists of several joint (axes) linked together, the definition of the path in space in effect requires that the robot move its axes through various positions in order to follow that path for a robot with six axes, each point in the path consists of six coordinates value corresponds to the position of one joint. There are basic robot anatomies; Polar, Cylindrical, Cartesian and Jointed Arm.

Each one of three axes associated with the arm and body configuration and two or three additional joints are associated with wrist. The arm and body joint determines the general position in space of the end effector and the wrist determines its orientation. If we think of a joint in space in the robot program as a position and orientation of the end effector, there is usually more than one possible set of joint coordinate values that can be used for the robot to reach that point.

For example, there are two alternative axis configurations that can be used by the jointed arm shown in figure to achieve the target point indicated.

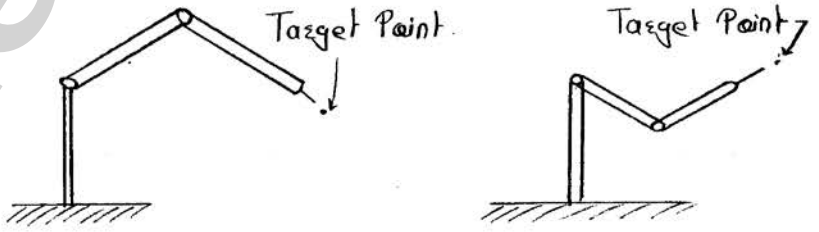


fig @:→ Two alternative axis configurations with end effector located at desired target point.

As shown in figure (a) that; although the target point has been reached by both of alternative axis configurations, there is a difference in the orientation of the wrist with respect to the point. We must conclude from this that the specification of the joint coordinates of the robot does define only one point in a space that orresponds to that set of coordinate values. Point specified in this fashion are said to be joint coordinates. Accordingly, an advantage of defining robot program in this way is that is simultaneously specifies the position and orientation of the end effector at each point in the path.

Let's consider the problem of defining a sequence of points in space. We will assume that these points are defined by specifying the joint coordinates as described above. Although, this method of specification will not affect the issue we are discussing here for a sake of simplicity, lets assume that we are programming a point-to-point Cartesian robot with only two axes and only two addressable point is one of the available points (as determine by the control revolution) that can be commended to go to that point. Figure (b) shows the four points (possible points) in the robot's rectangular space. A program of this robot to start in lower left hand corner and traverse the perimeter of the rectangle could be written as follows;

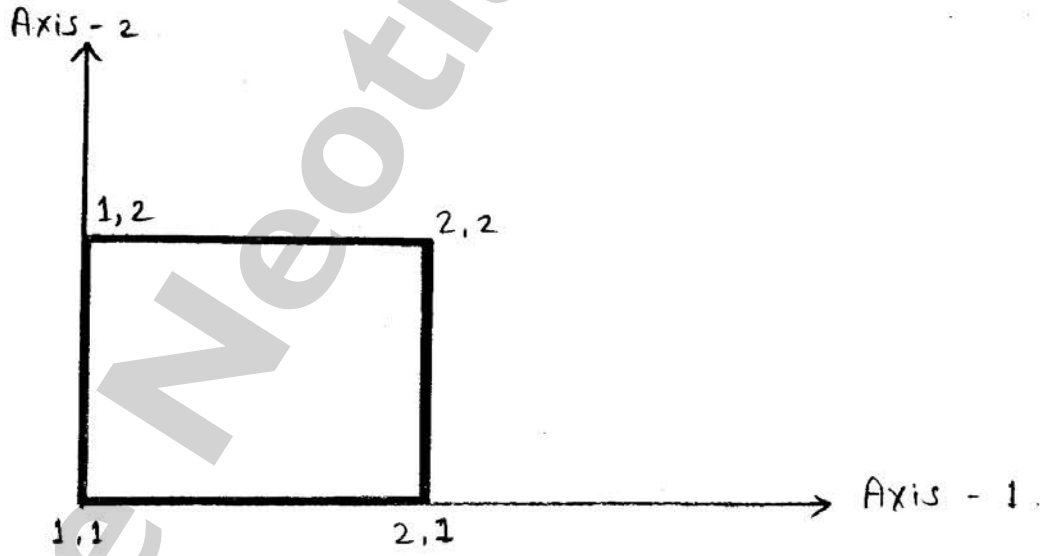


fig (b) - Robot Workspace.

STEP	MOVE	COMMENTS
1	1, 1	Move to lower left corner.
2	2, 1	Move to lower right corner.
3	2, 2	Move to upper right corner.
4	1, 2	Move to upper left corner.
5	1, 1	Move back to start position.

The point designation corresponds to the x, y- coordinates positions in the Cartesian axis system. In this example, using a robot with two orthogonal slides and only two addressable points per axis, the definition of points in space corresponds exactly with joint coordinate's values.

CONCLUSION: Thus, we have studied the robot path planning using simulation control software.

THE NEOTIA UNIVERSITY
DEPARTMENT OF ROBOTICS & AUTOMATION

EXPERIMENT NO.: 3

NAME OF THE EXPERIMENT: Implement Simultaneous Localization and Mapping (SLAM) with LIDAR Scans

OBJECTIVE: To Implement Simultaneous Localization and Mapping (SLAM) with LIDAR Scans.

THEORY:

This example demonstrates how to implement the Simultaneous Localization And Mapping (SLAM) algorithm on a collected series of lidar scans using pose graph optimization. The goal of this example is to build a map of the environment using the lidar scans and retrieve the trajectory of the robot.

To build the map of the environment, the SLAM algorithm incrementally processes the lidar scans and builds a pose graph that links these scans. The robot recognizes a previously-visited place through scan matching and may establish one or more loop closures along its moving path. The SLAM algorithm utilizes the loop closure information to update the map and adjust the estimated robot trajectory.

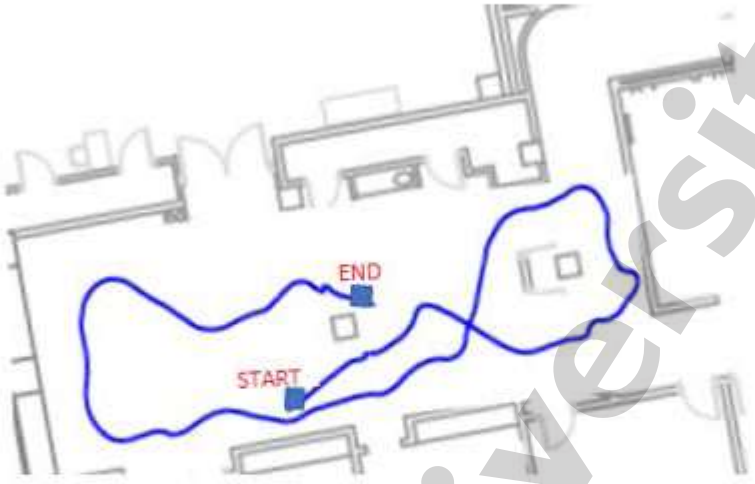
Load Laser Scan Data from File:

Load a down-sampled data set consisting of laser scans collected from a mobile robot in an indoor environment. The average displacement between every two scans is around 0.6 meters.

The offlineSlamData.mat file contains the scans variable, which contains all the laser scans used in this example:

```
load('offlineSlamData.mat');
```

A floor plan and approximate path of the robot are provided for illustrative purposes. This image shows the relative environment being mapped and the approximate trajectory of the robot.



Run SLAM Algorithm, Construct Optimized Map and Plot Trajectory of the Robot

Create a lidarSLAM object and set the map resolution and the max lidar range. This example uses a Jackal™ robot from Clearpath Robotics™. The robot is equipped with a SICK™ TiM-511 laser scanner with a max range of 10 meters. Set the max lidar range slightly smaller than the max scan range (8m), as the laser readings are less accurate near max range. Set the grid map resolution to 20 cells per meter, which gives a 5cm precision.

```
maxLidarRange = 8;
mapResolution = 20;
slamAlg = lidarSLAM(mapResolution, maxLidarRange);
```

The following loop closure parameters are set empirically. Using higher loop closure threshold helps reject false positives in loop closure identification process. However, keep in mind that a high-score match may still be a bad match. For example, scans collected in an environment that has similar or repeated features are more likely to produce false positives. Using a higher loop closure search radius allows the algorithm to search a wider range of the map around current pose estimate for loop closures.

```
slamAlg.LoopClosureThreshold = 210;
slamAlg.LoopClosureSearchRadius = 8;
```

Observe the Map Building Process with Initial 10 Scans

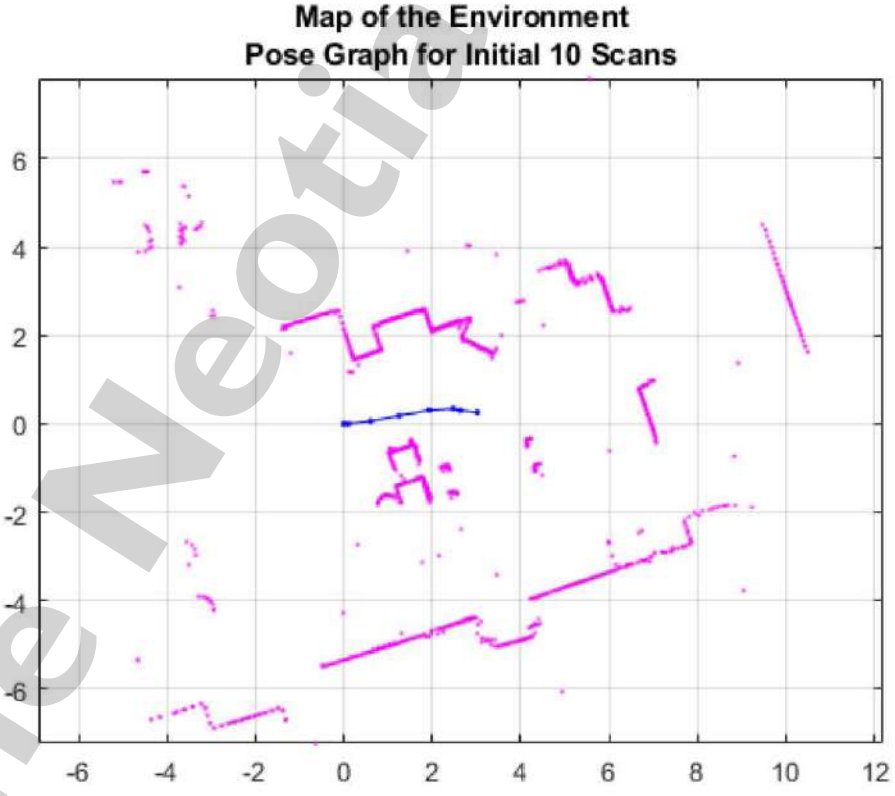
Incrementally add scans to the slamAlg object. Scan numbers are printed if added to the map. The object rejects scans if the distance between scans is too small. Add the first 10 scans first to test your algorithm.

```
for i=1:10
    [isScanAccepted, loopClosureInfo, optimizationInfo] = addScan(slamAlg, scans{i});
    if isScanAccepted
        fprintf('Added scan %d \n', i);
    end
end
```

```
Added scan 1
Added scan 2
Added scan 3
Added scan 4
Added scan 5
Added scan 6
Added scan 7
Added scan 8
Added scan 9
Added scan 10
```

Reconstruct the scene by plotting the scans and poses tracked by the slamAlg.

```
figure;
show(slamAlg);
title({'Map of the Environment','Pose Graph for Initial 10 Scans'});
```

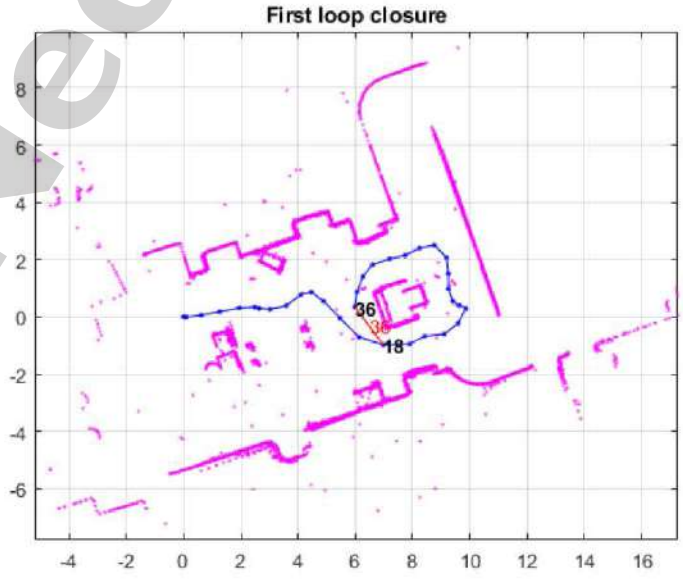


Observe the Effect of Loop Closures and the Optimization Process

Continue to add scans in a loop. Loop closures should be automatically detected as the robot moves. Pose graph optimization is performed whenever a loop closure is identified. The output optimization Info has a field, Is Performed, that indicates when pose graph optimization occurs..

Plot the scans and poses whenever a loop closure is identified and verify the results visually. This plot shows overlaid scans and an optimized pose graph for the first loop closure. A loop closure edge is added as a red link.

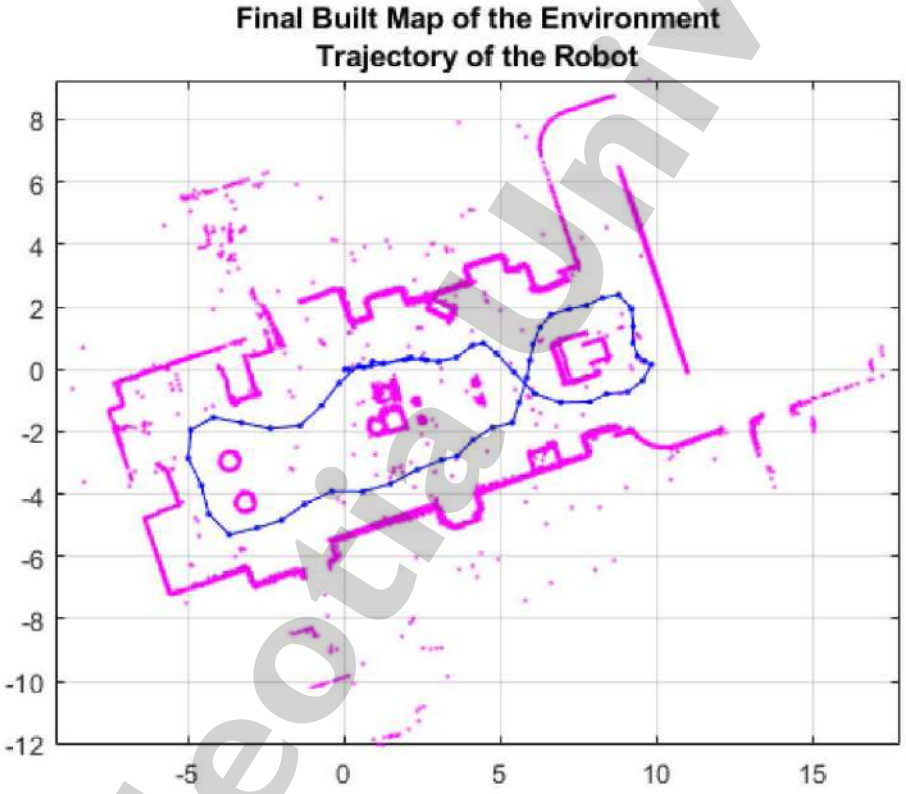
```
firstTimeLCDetected = false;

figure;
for i=10:length(scans)
    [isScanAccepted, loopClosureInfo, optimizationInfo] = addScan(slamAlg, scans{i});
    if ~isScanAccepted
        continue;
    end
    % visualize the first detected loop closure, if you want to see the
    % complete map building process, remove the if condition below
    if optimizationInfo.IsPerformed && ~firstTimeLCDetected
        show(slamAlg, 'Poses', 'off');
        hold on;
        show(slamAlg.PoseGraph);
        hold off;
        firstTimeLCDetected = true;
        drawnow
    end
end
title('First loop closure');
```



Visualize the Constructed Map and Trajectory of the Robot

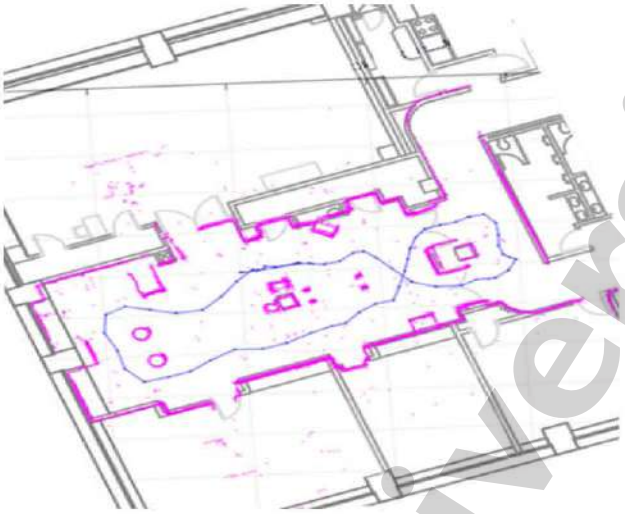
Plot the final built map after all scans are added to the slamAlg object. The previous for loop should have added all the scans despite only plotting the initial loop closure.

```
figure
show(slamAlg);
title({'Final Built Map of the Environment', 'Trajectory of the Robot'});
```



Visually Inspect the Built Map Compared to the Original Floor Plan

An image of the scans and pose graph is overlaid on the original floorplan. You can see that the map matches the original floor plan well after adding all the scans and optimizing the pose graph.



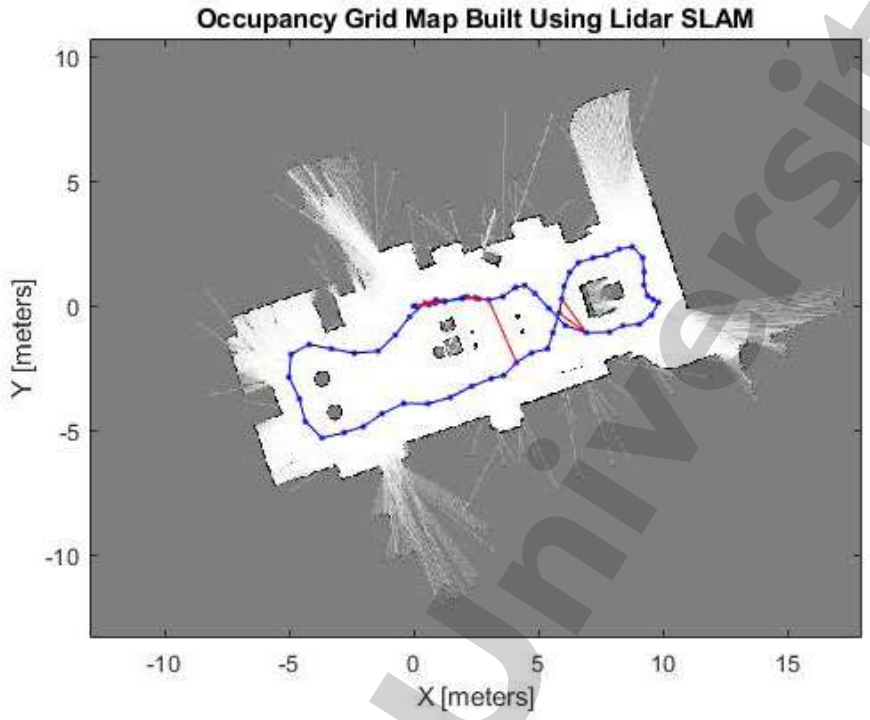
Build Occupancy Grid Map

The optimized scans and poses can be used to generate a occupancyMap, which represents the environment as a probabilistic occupancy grid.

```
[scans, optimizedPoses]  = scansAndPoses(slamAlg);
map = buildMap(scans, optimizedPoses, mapResolution, maxLidarRange);
```

Visualize the occupancy grid map populated with the laser scans and the optimized pose graph.

```
figure;
show(map);
hold on
show(slamAlg.PoseGraph, 'IDs', 'off');
hold off
title('Occupancy Grid Map Built Using Lidar SLAM');
```



CONCLUSION: After completion the experiment, students are able implement Simultaneous Localization and Mapping (SLAM) with LIDAR Scans

THE NEOTIA UNIVERSITY
DEPARTMENT OF ROBOTICS & AUTOMATION

EXPERIMENT NO.: 4

NAME OF THE EXPERIMENT: Localize TurtleBot Using Monte Carlo Localization.

OBJECTIVE: To Localize TurtleBot Using Monte Carlo Localization.

THEORY:

This example demonstrates an application of the Monte Carlo Localization (MCL) algorithm on TurtleBot® in simulated Gazebo® environment.

Monte Carlo Localization (MCL) is an algorithm to localize a robot using a particle filter. The algorithm requires a known map and the task is to estimate the pose (position and orientation) of the robot within the map based on the motion and sensing of the robot. The algorithm starts with an initial belief of the robot pose's probability distribution, which is represented by particles distributed according to such belief. These particles are propagated following the robot's motion model each time the robot's pose changes. Upon receiving new sensor readings, each particle will evaluate its accuracy by checking how likely it would receive such sensor readings at its current pose. Next the algorithm will redistribute (resample) particles to bias particles that are more accurate. Keep iterating these moving, sensing and re sampling steps, and all particles should converge to a single cluster near the true pose of robot if localization is successful.

Adaptive Monte Carlo Localization (AMCL) is the variant of MCL implemented in monteCarloLocalization. AMCL dynamically adjusts the number of particles based on KL-distance [1] to ensure that the particle distribution converge to the true distribution of robot state based on all past sensor and motion measurements with high probability.

The current MATLAB® AMCL implementation can be applied to any differential drive robot equipped with a range finder.

The Gazebo TurtleBot simulation must be running for this example to work.

Prerequisites: Get Started with Gazebo and a Simulated TurtleBot (ROS Toolbox), Access the tf Transformation Tree in ROS (ROS Toolbox), Exchange Data with ROS Publishers and Subscribers (ROS Toolbox).

Note: Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

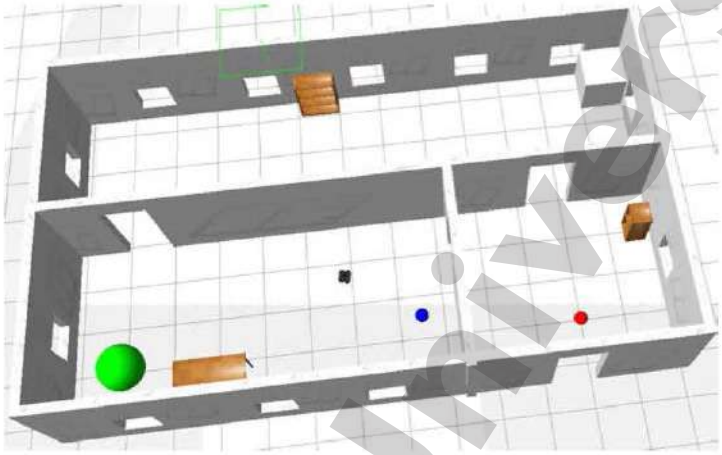
Connect to the TurtleBot in Gazebo

First, spawn a simulated TurtleBot inside an office environment in a virtual machine by following steps in the Get Started with Gazebo and a Simulated TurtleBot (ROS Toolbox) to launch the Gazebo Office World from the desktop, as shown below.

In your MATLAB instance on the host computer, run the following commands to initialize ROS global node in MATLAB and connect to the ROS master in the virtual machine through its IP address ipaddress. Replace ipaddress with the IP address of your TurtleBot in virtual machine.

```
ipaddress = '192.168.2.150';
rosinit(ipaddress,11311);
```

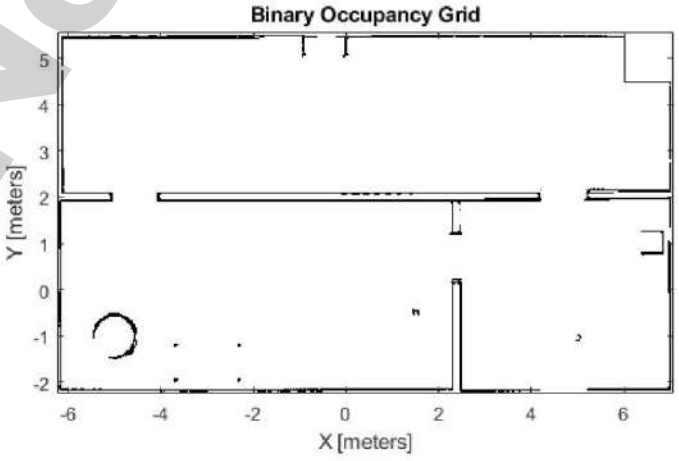
The layout of simulated office environment:



Load the Map of the Simulation World

Load a binary occupancy grid of the office environment in Gazebo. The map is generated by driving TurtleBot inside the office environment. The map is constructed using range-bearing readings from Kinect® and ground truth poses from gazebo/model_states topic.

```
load officemap.mat
show(map)
```



Setup the Laser Sensor Model and TurtleBot Motion Model:

TurtleBot can be modeled as a differential drive robot and its motion can be estimated using odometry data. The Noise property defines the uncertainty in robot's rotational and linear motion. Increasing the odometryModel.Noise property will allow more spread when propagating particles using odometry measurements. Refer to odometry Motion Model for property details.

```
odometryModel = odometryMotionModel;
odometryModel.Noise = [0.2 0.2 0.2 0.2];
```

The sensor on TurtleBot is a simulated range finder converted from Kinect readings. The likelihood field method is used to compute the probability of perceiving a set of measurements by comparing the end points of the range finder measurements to the occupancy map. If the end points match the occupied points in occupancy map, the probability of perceiving such measurements is high. The sensor model should be tuned to match the actual sensor property to achieve better test results. The property SensorLimits defines the minimum and maximum range of sensor readings. The property Map defines the occupancy map used for computing likelihood field. Please refer to likelihoodFieldSensorModel for property details.

```
rangeFinderModel = likelihoodFieldSensorModel;
rangeFinderModel.SensorLimits = [0.45 8];
rangeFinderModel.Map = map;
```

Set rangeFinderModel.SensorPose to the coordinate transform of the fixed camera with respect to the robot base. This is used to transform the laser readings from camera frame to the base frame of TurtleBot. Please refer to Access the tf Transformation Tree in ROS (ROS Toolbox) for details on coordinate transformations.

Note that currently SensorModel is only compatible with sensors that are fixed on the robot's frame, which means the sensor transform is constant.

```
% Query the Transformation Tree (tf tree) in ROS.
tftree = rostf;
waitForTransform(tftree,'/base_link','/base_scan');
sensorTransform = getTransform(tftree,'/base_link', '/base_scan');
```

```
% Get the euler rotation angles.
laserQuat = [sensorTransform.Transform.Rotation.W
sensorTransform.Transform.Rotation.X ...
    sensorTransform.Transform.Rotation.Y sensorTransform.Transform.Rotation.Z];
laserRotation = quat2eul(laserQuat, 'ZYX');

% Setup the |SensorPose|, which includes the translation along base_link's
% +X, +Y direction in meters and rotation angle along base_link's +Z axis
% in radians.
rangeFinderModel.SensorPose = ...
    [sensorTransform.Transform.Translation.X sensorTransform.Transform.Translation.Y
laserRotation(1)];
```

Receiving Sensor Measurements and Sending Velocity Commands

Create ROS subscribers for retrieving sensor and odometry measurements from TurtleBot.

```
laserSub = rossubscriber('/scan');
odomSub = rossubscriber('/odom');
```

Create ROS publisher for sending out velocity commands to TurtleBot. TurtleBot subscribes to '/mobile_base/commands/velocity' for velocity commands.

```
[velPub,velMsg] = ...
    rospublisher('/cmd_vel','geometry_msgs/Twist');
```

Initialize AMCL Object

Instantiate an AMCL object amcl. See monteCarloLocalization for more information on the class.

```
amcl = monteCarloLocalization;
amcl.UseLidarScan = true;
```

Assign the MotionModel and SensorModel properties in the amcl object.

```
amcl.MotionModel = odometryModel;
amcl.SensorModel = rangeFinderModel;
```

The particle filter only updates the particles when the robot's movement exceeds the UpdateThresholds, which defines minimum displacement in [x, y, yaw] to trigger filter update. This prevents too frequent updates due to sensor noise. Particle resampling happens after the amcl.ResamplingInterval filter updates. Using larger numbers leads to slower particle depletion at the price of slower particle convergence as well.

```
amcl.UpdateThresholds = [0.2,0.2,0.2];
amcl.ResamplingInterval = 1;
```

Configure AMCL Object for Localization with Initial Pose Estimate.

amcl.ParticleLimits defines the lower and upper bound on the number of particles that will be generated during the resampling process. Allowing more particles to be generated may improve the chance of converging to the true robot pose, but has an impact on computation speed and particles may take longer time or even fail to converge. Please refer to the 'KL-D Sampling' section in [1] for computing a reasonable bound value on the number of particles. Note that global localization may need significantly more particles compared to localization with an initial pose estimate. If the robot knows its initial pose with some uncertainty, such additional information can help AMCL localize robots faster with a less number of particles, i.e. you can use a smaller value of upper bound in amcl.ParticleLimits.

Now set amcl.GlobalLocalization to false and provide an estimated initial pose to AMCL. By doing so, AMCL holds the initial belief that robot's true pose follows a Gaussian distribution with a mean equal to amcl.InitialPose and a covariance matrix equal to amcl.InitialCovariance. Initial pose estimate should be obtained according to your setup. This example helper retrieves the robot's current true pose from Gazebo.

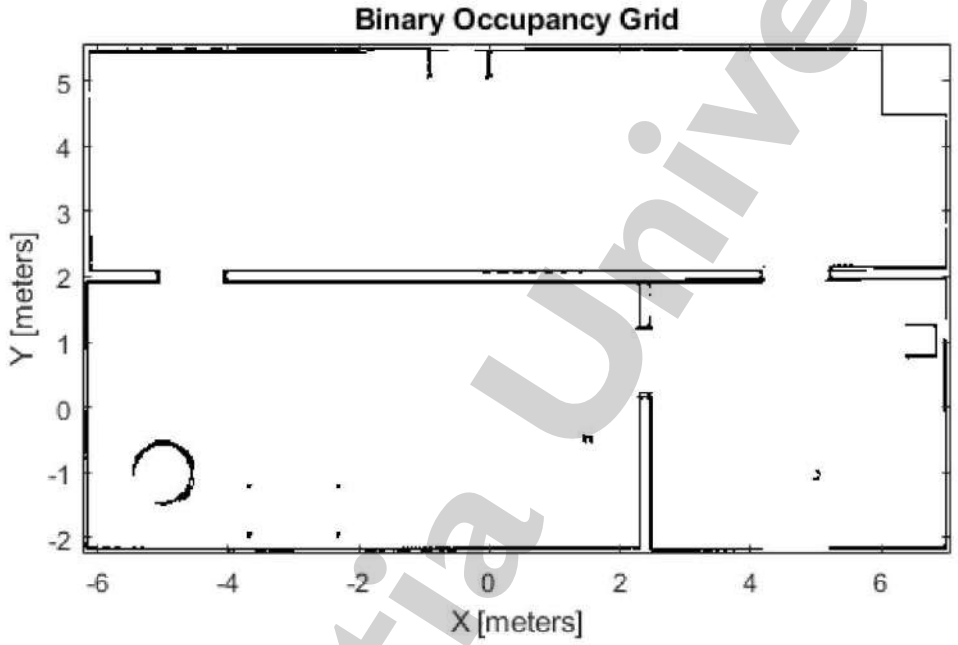
Please refer to section Configure AMCL object for global localization for an example on using global localization.

```
amcl.ParticleLimits = [500 5000];
amcl.GlobalLocalization = false;
amcl.InitialPose = ExampleHelperAMCLGazeboTruePose;
amcl.InitialCovariance = eye(3)*0.5;
```

Setup Helper for Visualization and Driving TurtleBot.

Setup ExampleHelperAMCLVisualization to plot the map and update robot's estimated pose, particles, and laser scan readings on the map.

```
visualizationHelper = ExampleHelperAMCLVisualization(map);
```



Robot motion is essential for the AMCL algorithm. In this example, we drive TurtleBot randomly using the ExampleHelperAMCLWanderer class, which drives the robot inside the environment while avoiding obstacles using the controllerVFH class.

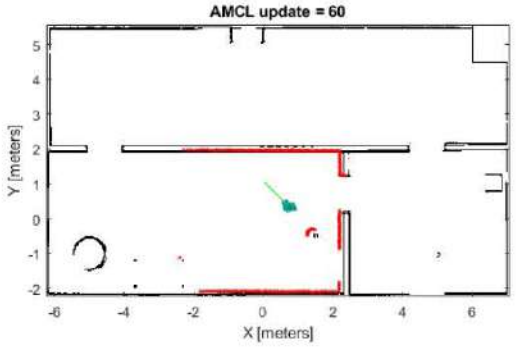
```
wanderHelper = ...
    ExampleHelperAMCLWanderer(laserSub, sensorTransform, velPub, velMsg);
```

Localization Procedure

The AMCL algorithm is updated with odometry and sensor readings at each time step when the robot is moving around. Please allow a few seconds before particles are initialized and plotted in the figure. In this example we will run numUpdates

AMCL updates. If the robot doesn't converge to the correct robot pose, consider using a larger numUpdates.

```
numUpdates = 60;
i = 0;
while i < numUpdates
    % Receive laser scan and odometry message.
    scanMsg = receive(laserSub);
    odompose = odomSub.LatestMessage;

    % Create lidarScan object to pass to the AMCL object.
    scan = lidarScan(scanMsg);

    % For sensors that are mounted upside down, you need to reverse the
    % order of scan angle readings using 'flip' function.

    % Compute robot's pose [x,y,yaw] from odometry message.
    odomQuat = [odompose.Pose.Pose.Orientation.W, odompose.Pose.Pose.Orientation.X,
...
        odompose.Pose.Pose.Orientation.Y, odompose.Pose.Pose.Orientation.Z];
    odomRotation = quat2eul(odomQuat);
    pose = [odompose.Pose.Pose.Position.X, odompose.Pose.Pose.Position.Y
odomRotation(1)];

    % Update estimated robot's pose and covariance using new odometry and
    % sensor readings.
    [isUpdated,estimatedPose, estimatedCovariance] = amcl(pose, scan);

    % Drive robot to next pose.
    wander(wanderHelper);

    % Plot the robot's estimated pose, particles and laser scans on the map.
    if isUpdated
        i = i + 1;
        plotStep(visualizationHelper, amcl, estimatedPose, scan, i)
    end

end
```



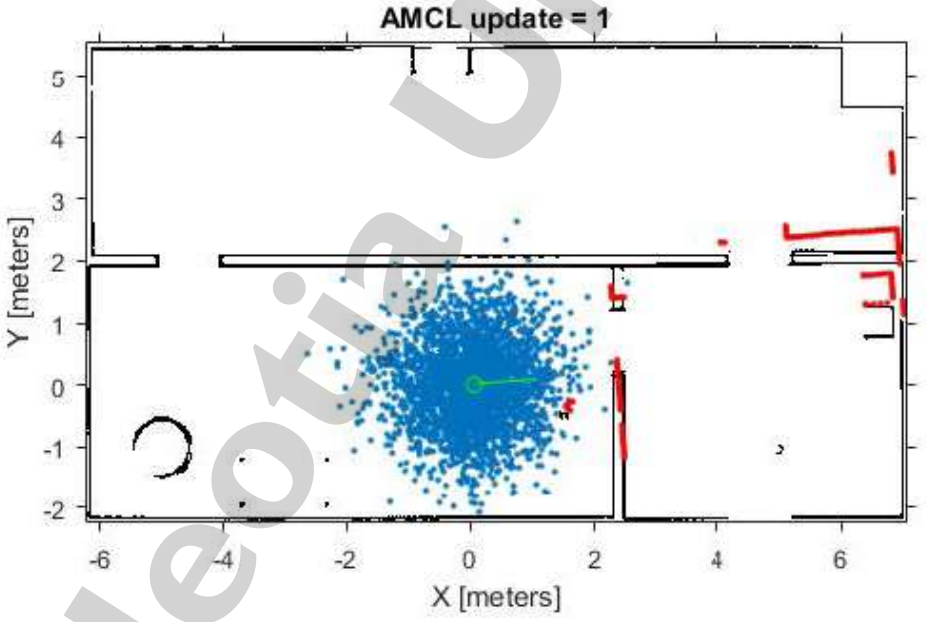
Stop the TurtleBot and Shutdown ROS in MATLAB

```
stop(wanderHelper);
rosshutdown
```

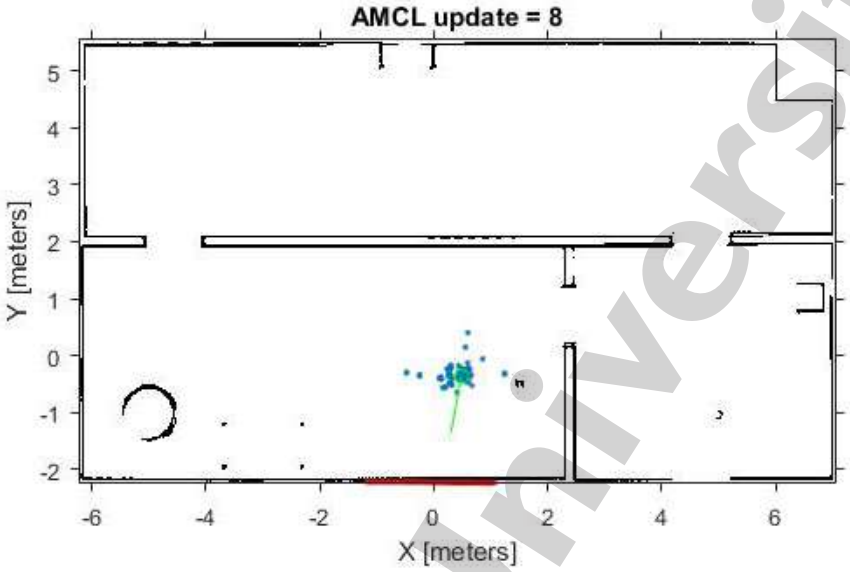
Sample Results for AMCL Localization with Initial Pose Estimate

AMCL is a probabilistic algorithm, the simulation result on your computer may be slightly different from the sample run shown here.

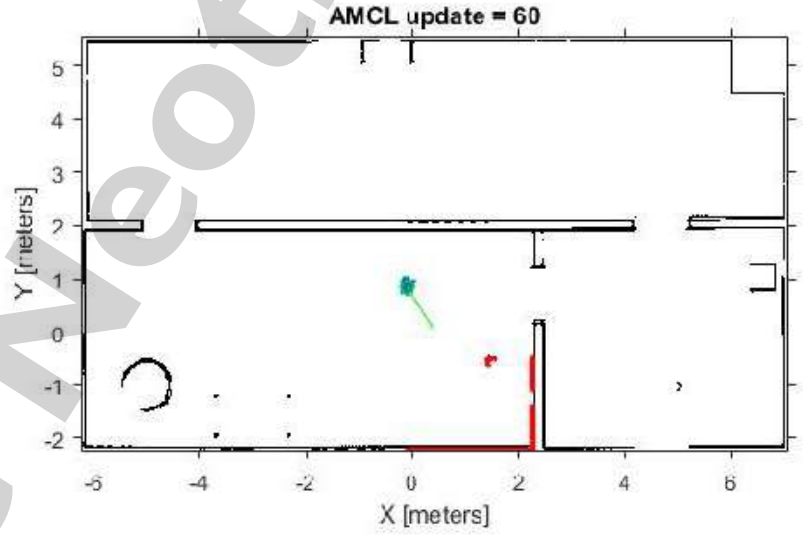
After first AMCL update, particles are generated by sampling Gaussian distribution with mean equal to amcl.InitialPose and covariance equal to amcl.InitialCovariance.



After 8 updates, the particles start converging to areas with higher likelihood:



After 60 updates, all particles should converge to the correct robot pose and the laser scans should closely align with the map outlines.



Configure AMCL Object for Global Localization.

In case no initial robot pose estimate is available, AMCL will try to localize robot without knowing the robot's initial position. The algorithm initially assumes that the robot has equal probability in being anywhere in the office's free space and generates uniformly distributed particles inside such space. Thus Global localization requires significantly more particles compared to localization with initial pose estimate.

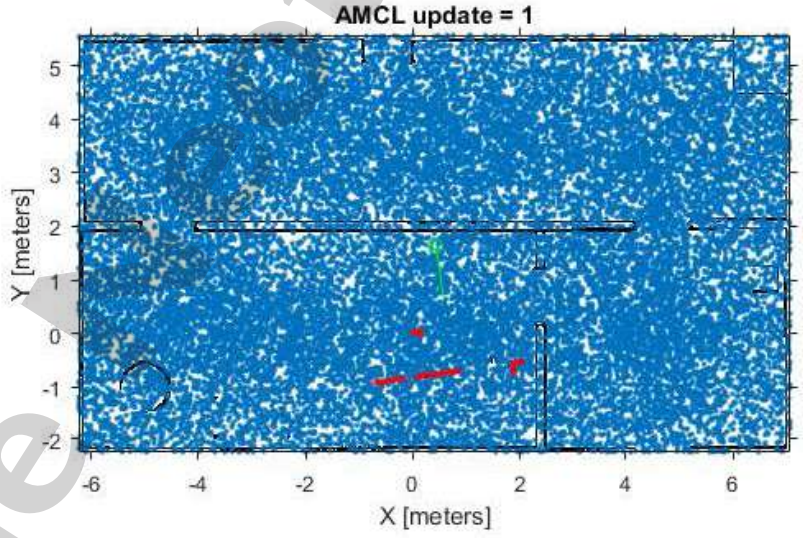
To enable AMCL global localization feature, replace the code sections in Configure AMCL object for localization with initial pose estimate with the code in this section.

```
amcl.GlobalLocalization = true;
amcl.ParticleLimits = [500 50000];
```

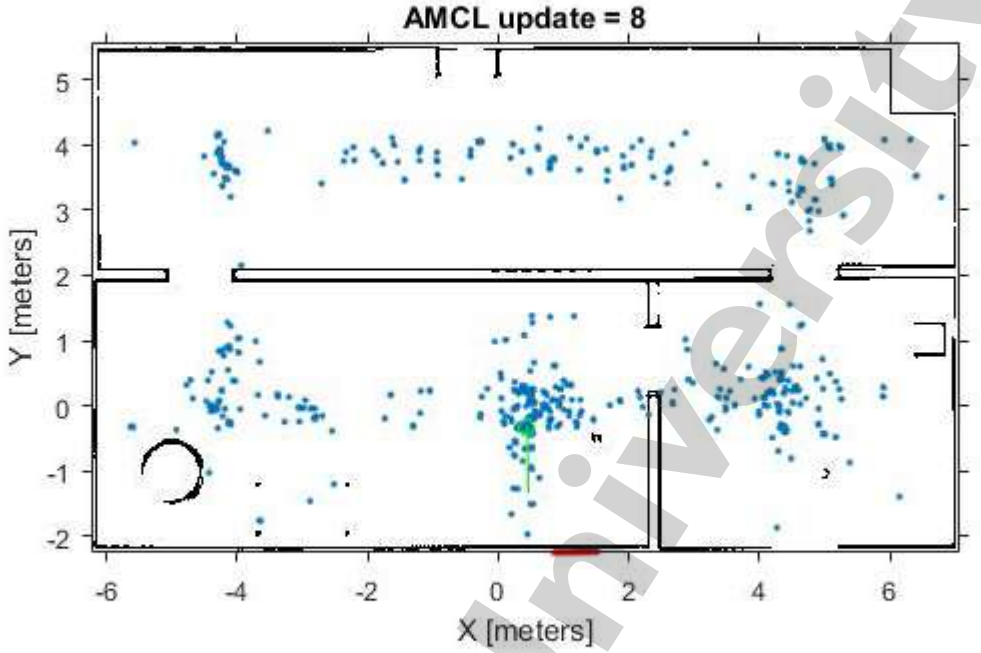
Sample Results for AMCL Global Localization

AMCL is a probabilistic algorithm, the simulation result on your computer may be slightly different from the sample run shown here.

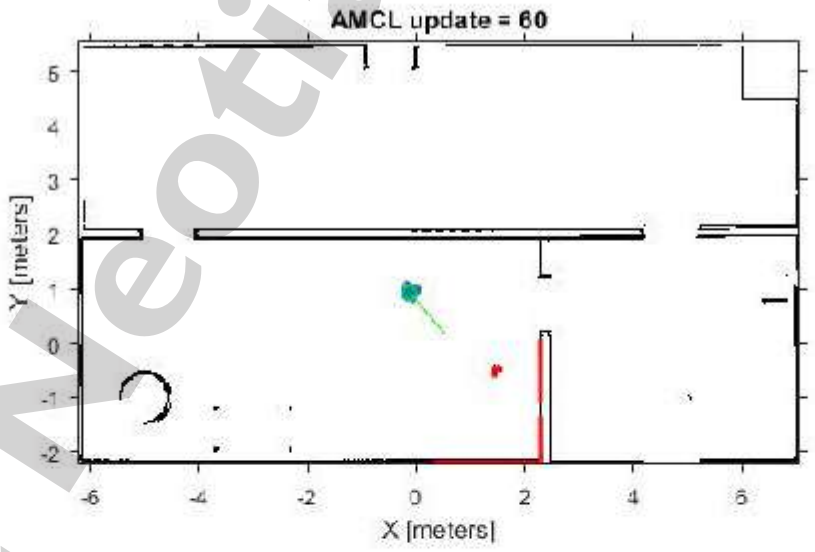
After first AMCL update, particles are uniformly distributed inside the free office space:



After 8 updates, the particles start converging to areas with higher likelihood:



After 60 updates, all particles should converge to the correct robot pose and the laser scans should closely align with the map outlines.



CONCLUSION: After completion the experiment, students are able to understand how to localize TurtleBot Using Monte Carlo Localization.

THE NEOTIA UNIVERSITY
DEPARTMENT OF ROBOTICS & AUTOMATION

EXPERIMENT NO.: 5

NAME OF THE EXPERIMENT: Estimate Position and Orientation of a Ground Vehicle.

OBJECTIVE: To Estimate Position and Orientation of a Ground Vehicle.

THEORY:

This example shows how to estimate the position and orientation of ground vehicles by fusing data from an inertial measurement unit (IMU) and a global positioning system (GPS) receiver.

SIMULATION SETUP:

Set the sampling rates. In a typical system, the accelerometer and gyroscope in the IMU run at relatively high sample rates. The complexity of processing data from those sensors in the fusion algorithm is relatively low. Conversely, the GPS runs at a relatively low sample rate and the complexity associated with processing it is high. In this fusion algorithm the GPS samples are processed at a low rate, and the accelerometer and gyroscope samples are processed together at the same high rate.

To simulate this configuration, the IMU (accelerometer and gyroscope) is sampled at 100 Hz, and the GPS is sampled at 10 Hz.

```
imuFs = 100;
gpsFs = 10;

% Define where on the Earth this simulation takes place using latitude,
% longitude, and altitude (LLA) coordinates.
localOrigin = [42.2825 -71.343 53.0352];

% Validate that the |gpsFs| divides |imuFs|. This allows the sensor sample
% rates to be simulated using a nested for loop without complex sample rate
% matching.

imuSamplesPerGPS = (imuFs/gpsFs);
assert(imuSamplesPerGPS == fix(imuSamplesPerGPS), ...
    'GPS sampling rate must be an integer factor of IMU sampling rate.');
```

FUSION FILTER:

Create the filter to fuse IMU + GPS measurements. The fusion filter uses an extended Kalman filter to track orientation (as a quaternion), position, velocity, and sensor biases.

The insfilterNonholonomic object that has two main methods: predict and fusegps. The predict method takes the accelerometer and gyroscope samples from the IMU as input. Call the predict method each time the accelerometer and gyroscope are sampled. This method predicts the states forward one time step based on the accelerometer and gyroscope. The error covariance of the extended Kalman filter is updated in this step.

The fusegps method takes the GPS samples as input. This method updates the filter states based on the GPS sample by computing a Kalman gain that weights the various sensor inputs according to their uncertainty. An error covariance is also updated in this step, this time using the Kalman gain as well.

The insfilterNonholonomic object has two main properties: IMUSampleRate and DecimationFactor. The ground vehicle has two velocity constraints that assume it does not bounce off the ground or slide on the ground. These constraints are applied using the extended Kalman filter update equations. These updates are applied to the filter states at a rate of IMUSampleRate/DecimationFactor Hz.

```
gndFusion = insfilterNonholonomic('ReferenceFrame', 'ENU', ...
    'IMUSampleRate', imuFs, ...
    'ReferenceLocation', localOrigin, ...
    'DecimationFactor', 2);
```

Create Ground Vehicle Trajectory

The waypoint Trajectory object calculates pose based on specified sampling rate, waypoints, times of arrival, and orientation. Specify the parameters of a circular trajectory for the ground vehicle.

```
% Trajectory parameters
r = 8.42; % (m)
speed = 2.50; % (m/s)
center = [0, 0]; % (m)
```

```
initialYaw = 90; % (degrees)
numRevs = 2;

% Define angles theta and corresponding times of arrival t.
revTime = 2*pi*r / speed;
theta = (0:pi/2:2*pi*numRevs).';
t = linspace(0, revTime*numRevs, numel(theta)).';

% Define position.
x = r .* cos(theta) + center(1);
y = r .* sin(theta) + center(2);
z = zeros(size(x));
position = [x, y, z];

% Define orientation.
yaw = theta + deg2rad(initialYaw);
yaw = mod(yaw, 2*pi);
pitch = zeros(size(yaw));
roll = zeros(size(yaw));
orientation = quaternion([yaw, pitch, roll], 'euler', ...
    'ZYX', 'frame');

% Generate trajectory.
groundTruth = waypointTrajectory('SampleRate', imuFs, ...
    'Waypoints', position, ...
    'TimeOfArrival', t, ...
    'Orientation', orientation);

% Initialize the random number generator used to simulate sensor noise.
rng('default');
```

GPS Receiver:

Set up the GPS at the specified sample rate and reference location. The other parameters control the nature of the noise in the output signal.

```
gps = gpsSensor('UpdateRate', gpsFs, 'ReferenceFrame', 'ENU');
gps.ReferenceLocation = localOrigin;
gps.DecayFactor = 0.5;                % Random walk noise parameter
gps.HorizontalPositionAccuracy = 1.0;
gps.VerticalPositionAccuracy =  1.0;
gps.VelocityAccuracy = 0.1;
```

IMU Sensors:

Typically, ground vehicles use a 6-axis IMU sensor for pose estimation. To model an IMU sensor, define an IMU sensor model containing an accelerometer and gyroscope. In a real-world application, the two sensors could come from a single

integrated circuit or separate ones. The property values set here are typical for low-cost MEMS sensors.

```
imu = imuSensor('accel-gyro', ...
    'ReferenceFrame', 'ENU', 'SampleRate', imuFs);

% Accelerometer
imu.Accelerometer.MeasurementRange =  19.6133;
imu.Accelerometer.Resolution = 0.0023928;
imu.Accelerometer.NoiseDensity = 0.0012356;

% Gyroscope
imu.Gyroscope.MeasurementRange = deg2rad(250);
imu.Gyroscope.Resolution = deg2rad(0.0625);
imu.Gyroscope.NoiseDensity = deg2rad(0.025);
```

Initialize the States of the insfilterNonholonomic:

The states are:

States	Units	Index
Orientation (quaternion parts)		1:4
Gyroscope Bias (XYZ)	rad/s	5:7
Position (NED)	m	8:10
Velocity (NED)	m/s	11:13
Accelerometer Bias (XYZ)	m/s^2	14:16

Ground truth is used to help initialize the filter states, so the filter converges to good answers quickly.

```
% Get the initial ground truth pose from the first sample of the trajectory
% and release the ground truth trajectory to ensure the first sample is not
% skipped during simulation.
[initialPos, initialAtt, initialVel] = groundTruth();
reset(groundTruth);

% Initialize the states of the filter
gndFusion.State(1:4) = compact(initialAtt).';
gndFusion.State(5:7) = imu.Gyroscope.ConstantBias;
gndFusion.State(8:10) = initialPos.';
gndFusion.State(11:13) = initialVel.';
gndFusion.State(14:16) = imu.Accelerometer.ConstantBias;
```

Initialize the Variances of the insfilterNonholonomic

The measurement noises describe how much noise is corrupting the GPS reading based on the GPS Sensor parameters and how much uncertainty is in the vehicle dynamic model.

The process noises describe how well the filter equations describe the state evolution. Process noises are determined empirically using parameter sweeping to jointly optimize position and orientation estimates from the filter.

```
% Measurement noises
Rvel = gps.VelocityAccuracy.^2;
Rpos = gps.HorizontalPositionAccuracy.^2;

% The dynamic model of the ground vehicle for this filter assumes there is
% no side slip or skid during movement. This means that the velocity is
% constrained to only the forward body axis. The other two velocity axis
% readings are corrected with a zero measurement weighted by the
% |ZeroVelocityConstraintNoise| parameter.
gndFusion.ZeroVelocityConstraintNoise = 1e-2;

% Process noises
gndFusion.GyroscopeNoise = 4e-6;
gndFusion.GyroscopeBiasNoise = 4e-14;
gndFusion.AccelerometerNoise = 4.8e-2;
gndFusion.AccelerometerBiasNoise = 4e-14;

% Initial error covariance
gndFusion.StateCovariance = 1e-9*ones(16);
```

Initialize Scopes:

The HelperScrollingPlotter scope enables plotting of variables over time. It is used here to track errors in pose. The HelperPoseViewer scope allows 3-D visualization of the filter estimate and ground truth pose. The scopes can slow the simulation. To disable a scope, set the corresponding logical variable to false.

```
if useErrScope
    errscope = HelperScrollingPlotter( ...
            'NumInputs', 4, ...
            'TimeSpan', 10, ...
```

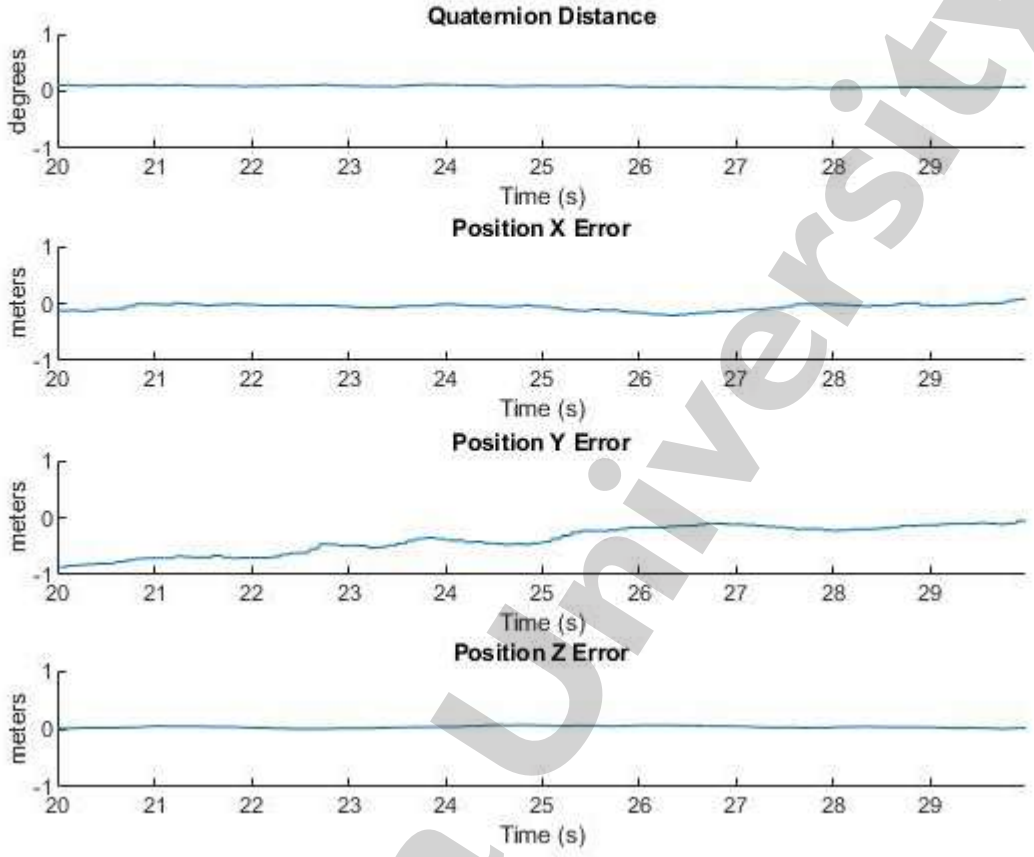
```
            'SampleRate', imuFs, ...
            'YLabel', {'degrees', ...
            'meters', ...
            'meters', ...
            'meters'}, ...
            'Title', {'Quaternion Distance', ...
            'Position X Error', ...
            'Position Y Error', ...
            'Position Z Error'}, ...
            'YLimits', ...
            [-1, 1
             -1, 1
             -1, 1
             -1, 1]);
end

if usePoseView
    viewer = HelperPoseViewer( ...
        'XPositionLimits', [-15, 15], ...
        'YPositionLimits', [-15, 15], ...
        'ZPositionLimits', [-5, 5], ...
        'ReferenceFrame', 'ENU');
end
```

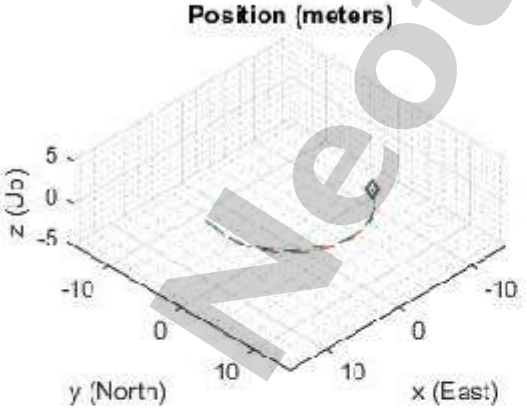
Simulation Loop:

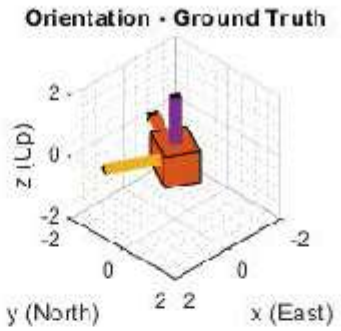
The main simulation loop is a while loop with a nested for loop. The while loop executes at the gpsFs, which is the GPS measurement rate. The nested for loop executes at the imuFs, which is the IMU sample rate. The scopes are updated at the IMU sample rate.

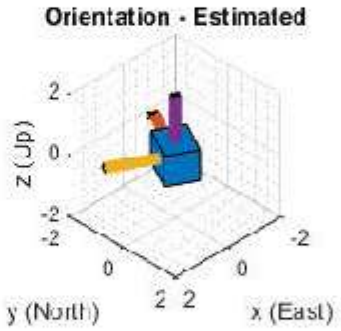
```
totalSimTime = 30; % seconds

% Log data for final metric computation.
numsamples = floor(min(t(end), totalSimTime) * gpsFs);
truePosition = zeros(numsamples,3);
trueOrientation = quaternion.zeros(numsamples,1);
estPosition = zeros(numsamples,3);
estOrientation = quaternion.zeros(numsamples,1);

idx = 0;

for sampleIdx = 1:numsamples
    % Predict loop at IMU update frequency.
    for i = 1:imuSamplesPerGPS
        if ~isDone(groundTruth)
```

```
            idx = idx + 1;

            % Simulate the IMU data from the current pose.
            [truePosition(idx,:), trueOrientation(idx,:), ...
                trueVel, trueAcc, trueAngVel] = groundTruth();
            [accelData, gyroData] = imu(trueAcc, trueAngVel, ...
                trueOrientation(idx,:));

            % Use the predict method to estimate the filter state based
            % on the accelData and gyroData arrays.
            predict(gndFusion, accelData, gyroData);

            % Log the estimated orientation and position.
            [estPosition(idx,:), estOrientation(idx,:)] = pose(gndFusion);

            % Compute the errors and plot.
            if useErrScope
                orientErr = rad2deg( ...
                    dist(estOrientation(idx,:), trueOrientation(idx,:)));
                posErr = estPosition(idx,:) - truePosition(idx,:);
                errscope(orientErr, posErr(1), posErr(2), posErr(3));
            end

            % Update the pose viewer.
            if usePoseView
                viewer(estPosition(idx,:), estOrientation(idx,:), ...
                    truePosition(idx,:), estOrientation(idx,:));
            end
        end
    end

    if ~isDone(groundTruth)
        % This next step happens at the GPS sample rate.
        % Simulate the GPS output based on the current pose.
        [lla, gpsVel] = gps(truePosition(idx,:), trueVel);

        % Update the filter states based on the GPS data.
        fusegps(gndFusion, lla, Rpos, gpsVel, Rvel);
    end
end
```









Error Metric Computation:

Position and orientation were logged throughout the simulation. Now compute an end-to-end root mean squared error for both position and orientation.

```
posd = estPosition - truePosition;

% For orientation, quaternion distance is a much better alternative to
% subtracting Euler angles, which have discontinuities. The quaternion
% distance can be computed with the |dist| function, which gives the
% angular difference in orientation in radians. Convert to degrees for
% display in the command window.

quatd = rad2deg(dist(estOrientation, trueOrientation));

% Display RMS errors in the command window.
fprintf('\n\nEnd-to-End Simulation Position RMS Error\n');
msep = sqrt(mean(posd.^2));
fprintf('\tX: %.2f , Y: %.2f, Z: %.2f   (meters)\n\n', msep(1), ...
    msep(2), msep(3));

fprintf('End-to-End Quaternion Distance RMS Error (degrees) \n');
fprintf('\t%.2f (degrees)\n\n', sqrt(mean(quatd.^2)));


End-to-End Simulation Position RMS Error
        X: 1.16 , Y: 0.99, Z: 0.03   (meters)
End-to-End Quaternion Distance RMS Error (degrees)
        0.09 (degrees)
```

CONCLUSION: After completion the experiment, students are able to understand how to estimate the Position and Orientation of a Ground Vehicle.

Courtesy: https://in.mathworks.com/& Google