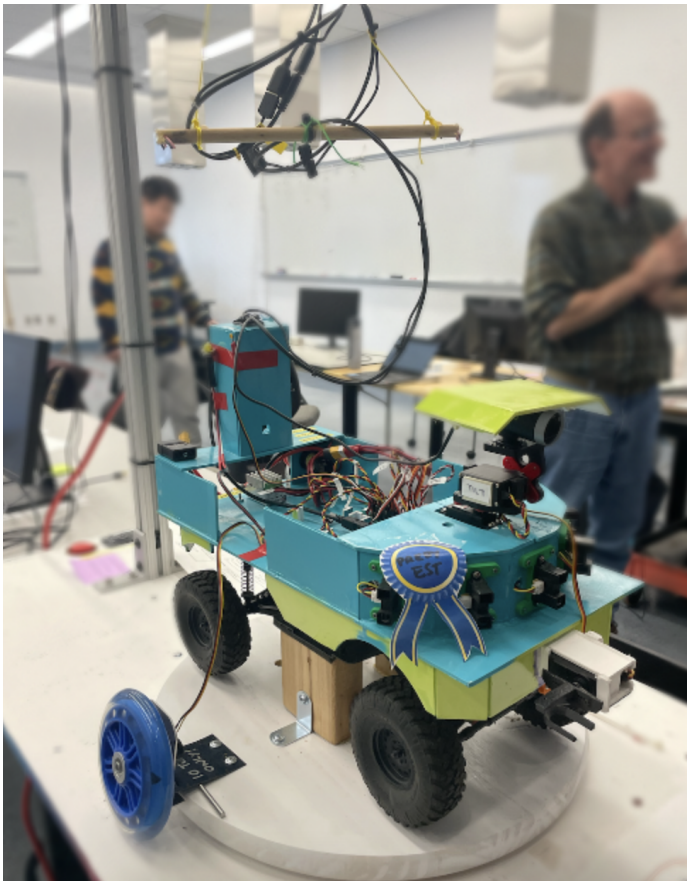
ENGR3390: Fundamentals of Robotics Report 3
THINK Lab



Think Lab Rover Hardware

Octopus Team: Alexis Wu, Braden Vaccari, Dowling Wong, Eliyahu Suskind, and Mahderekal Regassa

Report Author: Mahderekal Regassa

April 8, 2023

Table of Contents **Page**

Theory

Subsumption Architecture

In this lab, the main focus was to build THINK behavior functions and arbiters that command various actuators by adding onto the SENSE functions given in the tutorial. What makes robots carry out complex tasks in modern times, similar to biological systems in nature, is the **subsumption architecture** that allows high and low level behaviors to run at the same time. This important insight was forwarded by Prof. Rod Brook and it revolutionized how robots work. The main idea here is that low level actuation happens until/unless subsumed by higher-level behaviors. This “frees” the robot to reason and make decisions in the world for example while it is still able to avoid obstacles because both behaviors run in parallel.

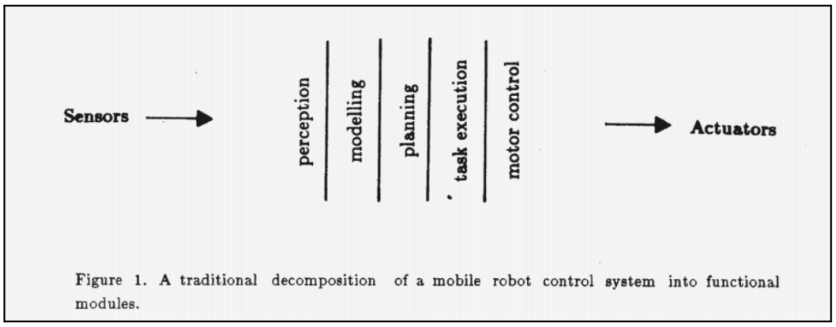


Figure 1. A traditional decomposition of a mobile robot control system into functional modules.

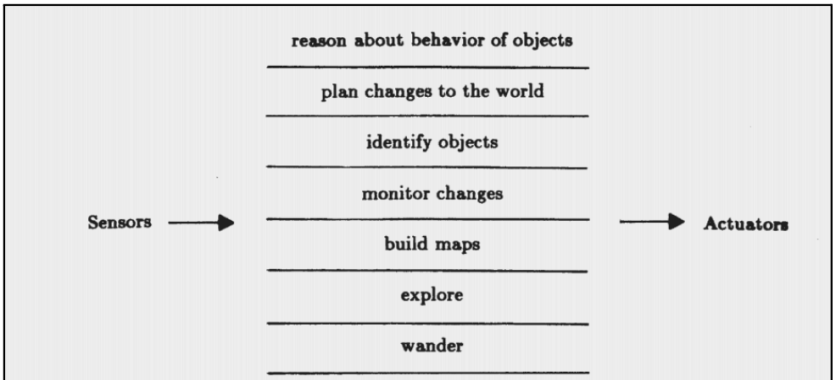


Figure 2. A decomposition of a mobile robot control system based on task achieving behaviors.

Fig.1.0 Linear execution of behaviors in old robots vs parallel behavior execution in modern robots

Using this principle, we use **finite state machines** to hold the *explore*, *avoid*, and *follow* behaviors for our robot. As the names suggest, these behaviors will have the rover explore to see what’s in the environment, avoid running into obstacles, and follow a target object. The two arbiters in this lab are heading and speed which are commanded steering angle and drive wheel speed, respectively.

Sensor Suits

Our robot is a rover mounted on a driven turntable and it has a webcam (for finding target), 6 sharp IR sensors (for obstacle detection), embedded servo controls (for steering), and a potentiometer (serving as a compass for heading).

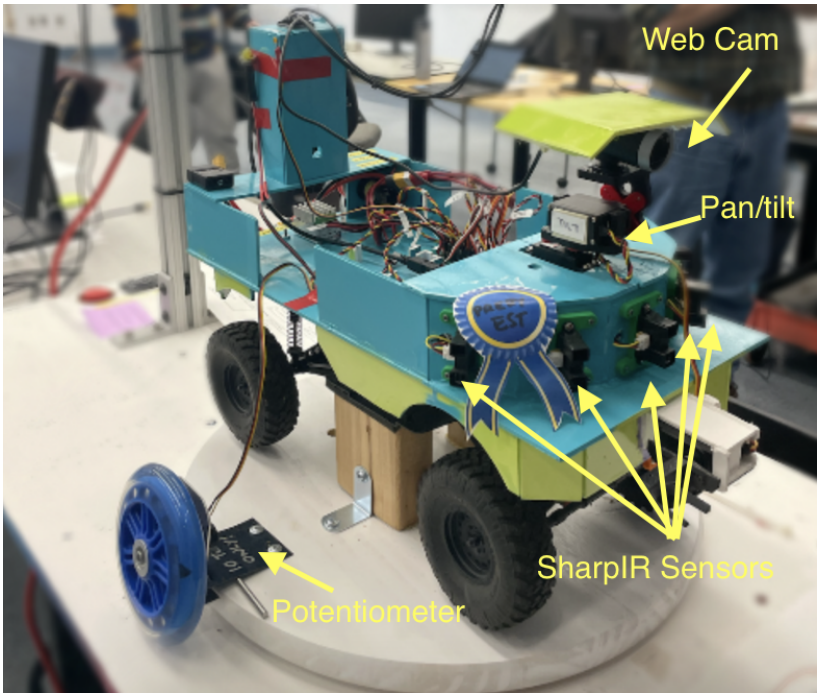


Fig. 1.1 The rover, atop a turntable

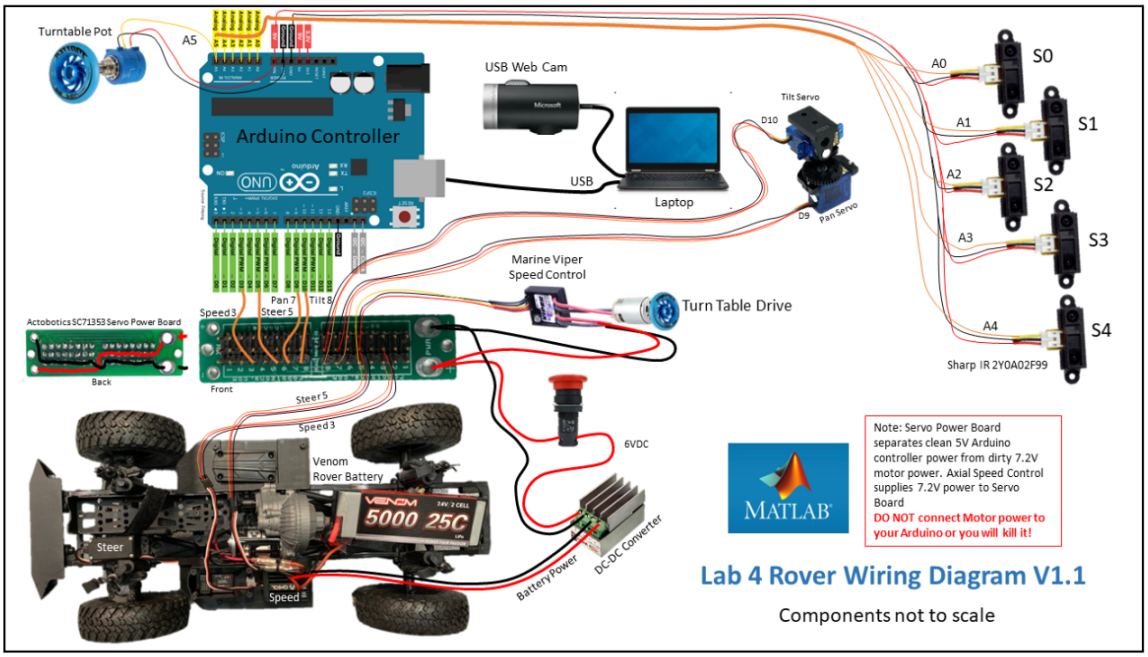


Fig. 1.2 Wiring diagram for the rover

Control Program for Simulated Environment

For the purposes of getting first hand experience in seeing the difference between a control system and robot (or robot controller), the lab is divided into two sections. For the first half we were made to write a waypoint pursuit code for the rover to follow in a simulated environment. For the second part, the goal is to add a thinking or behavior based robot brain to avoid obstacles, explore the environment, and follow the orange cone. In the former, we only have a control system that is hard wired into the rover, there is no thinking component. Such control systems prevent robots from reacting to unforeseen changes and interacting with the dynamic world freely. Since I was on spring break for the second half of the lab, I will only cover the first half (simulated rover) for the remainder of the report. I will be showing the results along with the code instead of having all the results in a different section in the end to avoid confusion of which code block gives the shown result. Before going over the code, I will briefly explain the theory behind the robot control code we used for this lab.

To simulate a simplified vehicle dynamics for a differential-drive vehicle (wheels that can be independently driven), we used a built-in Matlab function: **differentialDriveKinematics**. This model simplifies the rover to a fixed axle and two wheels separated by specific track width. The vehicle speed and heading can be defined from the center with global xy-position in meters and theta in radians (for the heading). As you will see in the code section, we gave the differentialDriveKinematics function two inputs: VehicleInputs VehicleSpeedHeadingRate. The former specifies the format of input commands for the model when using the derivative function. The latter specifies the vehicle speed and heading rate in m/s and rad/s, respectively. See image below to visualize the vehicle model.

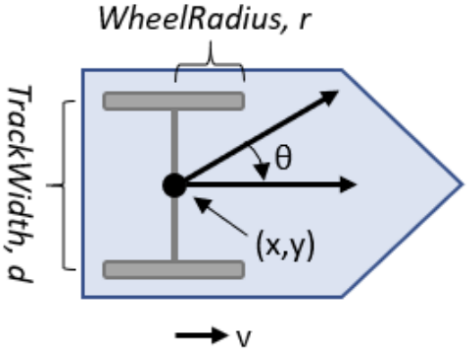


Fig. 1.3 Two wheel differential drive vehicle model

For our rover to follow the desired path, we generate robot speed and heading using another Matlab built-in object: **controllerPurePursuit**. Given the current pose (position and orientation of a vehicle), the object computes the linear and angular velocities for the rover. It takes a Name, Value pair as inputs.

Code

StepUp

- Since we started the lab with the simulated environment, we skipped the first setup chunk of the code in the tutorial. We clear and close all open figures starting off, then set up the oval test track image.

OvalRoverSimulation.mlx is a simulation of a planetary rover

This code simulates a wheeled autonomous planetary rover performing multiple missionas on the olin roval.

- It takes a desired location waypoints as inputs
- It delivers a moving map simulation of rover transersing the Oval as an output

Octopus Team and 21/03/2023

```
close all;
clc        % clear commmand window
clear      % clear MATLAB workspace
```

Set up robot control system (code that runs once)

```
img = imread('TestPool.png'); % load oval image
grayimage = im2gray(img);       % convert to grayscale
bwimage = grayimage < 0.5;      % convert to black and white
                                % anything black is an obstacle in this case
MapOfOval = binaryOccupancyMap(bwimage,10); % convert to a binary occupancy
                                            % map in meters
show(MapOfOval);

% Inflate the map with the robot rover size, so robot can be treated as a
% point. Assume robot rover is a 0.25m long bow to stern
MapOfOvalInflated= copy(MapOfOval); % make a copy to save original data
inflate(MapOfOvalInflated,0.25); % dilate map to accomendate robot size

% plot a standalone figure to view the testtrack
testTrack = figure('name','testTrackMap','NumberTitle','off','Visible','on')
figure(testTrack)
    show(MapOfOvalInflated);
    grid on;
    grid minor;
```

- We then create an empty figure to be used later in mapping the virtual IR sensor readings.

Create a IR based map of world (to be filled in as Rober drives around Oval)

```
% Create an empty map of the same dimensions as the test track map

[mapdimx, mapdimy] = size(bwimage)
sharpIRMap = binaryOccupancyMap(mapdimx, mapdimy, 10, 'grid')

% Create a free floating figure for map to deal with livescript update
% problem
roversharpIRScan = figure('name','RoverSharpIRScan','NumberTitle', ...
                          'off','Visible','on');
figure(roversharpIRScan)
    show(sharpIRMap);
    grid on;
    grid minor;
```

- The test track map has two obstacles (shown in black). See images below:

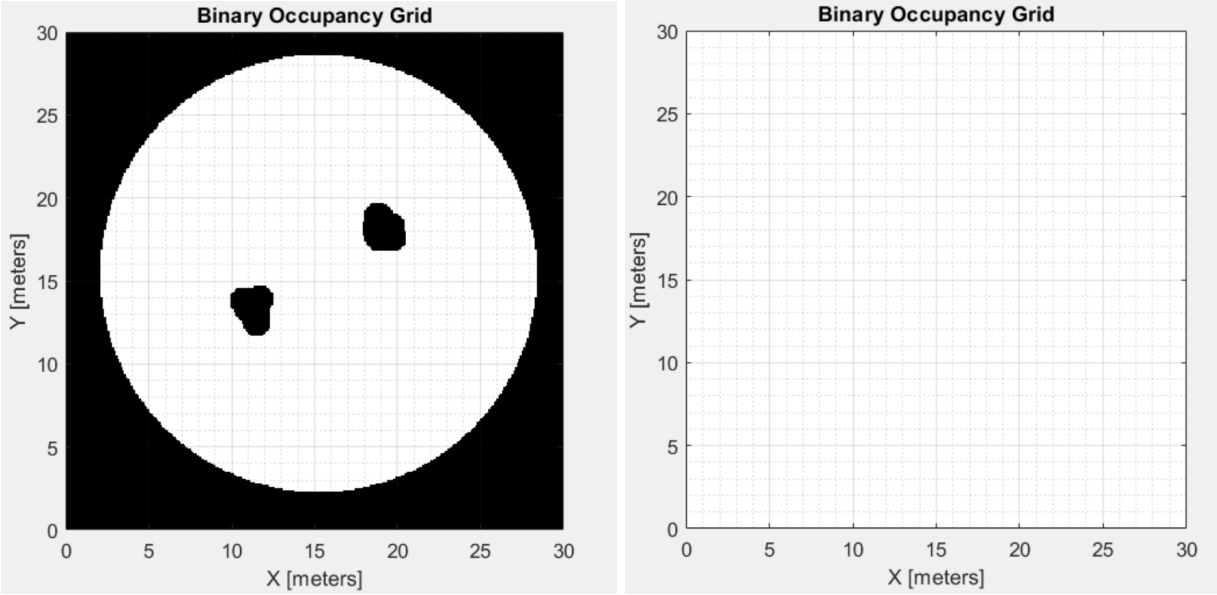


Fig. 1.4 Standalone figure of the test track map for our virtual rover (to the left) and a free floating empty figure for IR sensor readings (to the right)

- Here is the robot control code.

Create a differential driver robot rover (thus model taks robot vehicle speed and heading for inputs)

```
diffDriveRover = differentialDriveKinematics('VehicleInputs', ...
                                             'VehicleSpeedHeadingRate');
```

Create a pure pursuit rover controller (this controller generates the robots speed and hearing needed to follow a desired path, set desired linear velocity and max angular velocity in m/s and rad/s.

```
RoverController = controllerPurePursuit('DesiredLinearVelocity',2, ...
                                        'MaxAngularVelocity',3);
```

- Matlab has a virtual 360° scanning LiDAR, so for our purposes we converted that into 6 180° SharpIR sensors similar to the ones we have on the rover. The **rangeSensor**

object outputs range and angle measurements based on the given pose and test track map.

Create 180 degree sharp IR range sensor suite for your rover.

- Create a sensor with a max range of 10 meters. This sensor simulates range readings based on a given pose and map.
- The Test Track map is used with this range sensor to simulate collecting sensor readings in an unknown environment.
- Its a little hard to find documentation on range Sensor function, so right click on it and choose "help" or f1.

```
sharpIRPod = rangeSensor;        % create a rangeSensor system object
sharpIRPod.Range = [0.1,10];     % sets minimum and maximum range of sensor
sharpIRPod.HorizontalAngle =  [-pi/2, pi/2];
sharpIRPod.HorizontalAngleResolution = 0.628318;
testPose = [20 23 pi/2];    % sets an inital test pose for IR

% Plot the ttest spot for lidat scan on the reference Olin Oval Map
figure(testTrack)
hold on
plot(20,23,'r*');
hold off
```

```
% generate a IR test scan from test position
[ranges, angles] = sharpIRPod(testPose, MapOfOvalInflated);
scan = lidarScan(ranges,angles);


% Visualize the test lidat scan data in robot coordinate system.
% Create a new free-standing figure to display the local lidar data, rover
% at center

localSharpIRPlot = figure('Name','localSharpIRMap','NumberTitle','off', ...
                          'Visible','on')
figure(localSharpIRPlot)
plot(scan)
axis([-15 15 -15 15]);
hold on;
plot(0,0,'r*')
plot(-1,0,'r^')
plot(-2, 0, 'r^')
hold off;
```

- Following this code we get these figures:

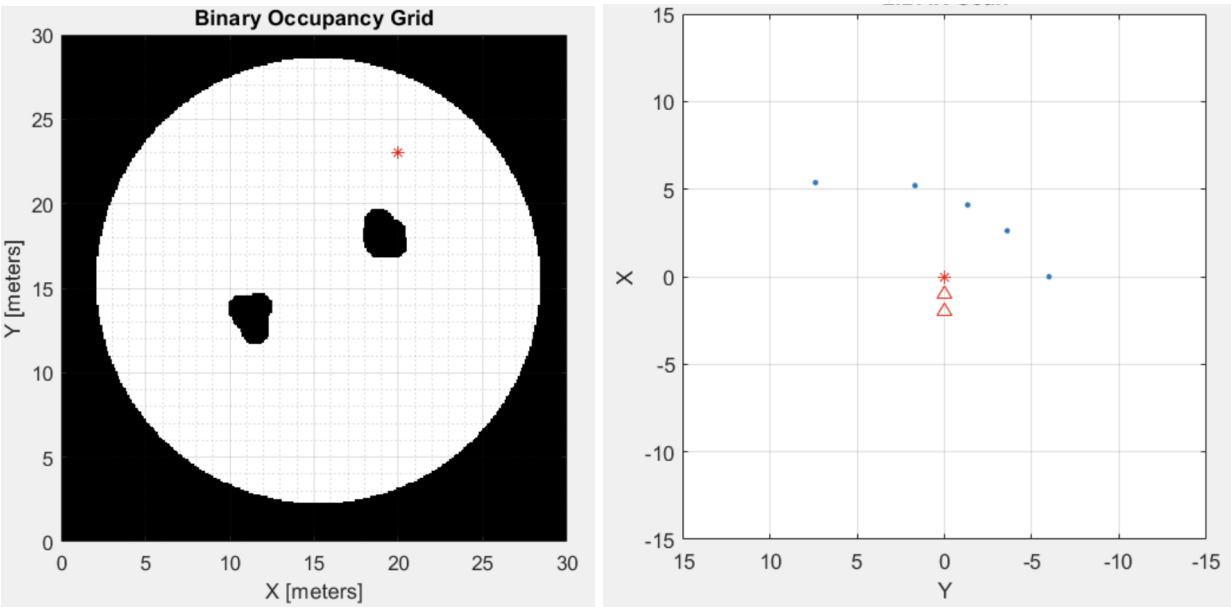


Fig. 1.5 Rover placed on the test track as a red star (to the left) and IR range readings from the *unknown* simulated environment of the test track.

- Next, we hard code a list of sequential waypoints for the rover to follow and load these waypoints to the rover's controller. We then set initial pose and final waypoint goal based on the given path.

create a test path to drive through the map for gathering range sensor readings.

```
path = [ 20 23; 15 23; 10 23; 5 15; 10 10; 15 10; 25 15];

% Plot the path on the reference map figure
figure(testTrack) %select OvalTrackMap
hold on
plot(path(:,1),path(:,2),'o-');%plot path on it
hold off;
```

Use this test path as the set of waypoints the pure purtsuit controller will follow:

```
RoverController.Waypoints = path; % set rover waypoints
```

Set the initial pose and final goal WayPoint location based on the path. Create global variables for storing the current pose and an index for tracking the iterations.

```
initRoverPose = [path(1,1) path(1,2), pi/2];  % store initial location and
                                              % orieatation of the rover
goalWayPoint = [path(end,1) path(end,2)]';    % PATH END WAYPOINT
sampleTime = .1;                              % sample time (s)
t = 0:sampleTime:100;                         % time array
robotPoses = zeros(3,numel(t));
robotPoses(:,1) = initRoverPose';
r = rateControl(1/sampleTime);
reset(r)
```

Robot Control Loop

```
controlIndex = 1;    % create a robot loop control
while(controlIndex < numel(t))  %loop for a number of elements in t
    position = robotPoses(:,controlIndex)';
    roverLocation = position(1:2);
    % End loop if rover has reached goal waypoint within tolerance of 1.0m
        dist = norm(goalWayPoint'- roverLocation);
        if(dist<1.0)
            disp("Goal position reached!");
            break;
        end
    % SENSE: collect data from robot IR sensors
    [ranges, angles] = SENSE(position, MapOfOvalInflated, sharpIRMap, ...
                             roversharpIRScan , localSharpIRPlot, sharpIRPod);
    % THINK: compute what robots should do next
    [roverX, roverY, robotPoses] = THINK(robotPoses, diffDriveRover, ...
                                  RoverController, sampleTime, controlIndex);
    % ACT: command robot actuators
    ACT(roverX,roverY,testTrack);

    controlIndex = controlIndex+1;  % increment control loop index
    waitfor(r); % wait for loop cycle to complete
end
```

Robot Functions

```
function [ranges, angles] = SENSE(position, mapOfTrack, blankLidarMap, ...
                                  fig_lidarMap,fig_localLidarPlot,lidar)
% SENSE scans the reference map using the range sensor and the current
% pose.
% This simulates normal range readings for driving in an unknown
% environment
% Update the lidar map with the range readings
% inpus are rover position, mapOfTrack, lidarMap, figure to plot global
% lidar, figure to plot local lidar, sensor name outputs are lidar ranges
% and angles in local coordinate system

    [ranges, angles] = lidar(position, mapOfTrack);
    scan = lidarScan(ranges, angles);
    validScan = removeInvalidData(scan, "rangeLimits",[0,lidar.Range(2)]);
    insertRay(blankLidarMap, position, validScan, lidar.Range(2));

    figure(fig_lidarMap);
        hold on;
        show(blankLidarMap);
        grid on;
        grid minor;
        hold off;
```

```
    figure(fig_localLidarPlot);
    plot(scan);
    hold on;
    plot(0,0,'r*');
    hold off;
    workspace;
end
```

Think Functions (store all think related local funtions here)

```
function [roverX, roverY, poses] = THINK(poses, diffDrive, ppControl, ...
                                         sampleTime, loopIndex)
    % Run the Pure Pursuit controller and convert output to wheel speeds
    [vRef, wRef] = ppControl(poses(:,loopIndex));
    % Perform forward discrete integration step
    vel = derivative(diffDrive, poses(:,loopIndex), [vRef, wRef]);
    poses(:,loopIndex+1) = poses(:,loopIndex) + vel*sampleTime;

    %update rover location and pose
    roverX = poses(1,loopIndex+1);
    roverY = poses(2,loopIndex+1);
end
```

Act Functions (store all Act related functions here)

```
function ACT(roverX, roverY, fig_testTrack)
    figure(fig_testTrack)
    hold on
    plot(roverX,roverY,'b*');
     hold off
end
```

Clean and Shutdown

clean shut down

```
clc
disp("Rover control program has ended");
clear robotArduino
% clear all figures
close(testTrack);
close(roversharpIRScan);
close(localSharpIRPlot);

% play system sound when the program ends
beep
disp("All done!!");
```