SOFTWARE DOCUMENT:

Project: DPM Final Design Project
Task: Describe current and previous iterations of the software, as well as explaining design decisions.

Document Version Number: 4.0
Date: November 28^{th} 2019
Author: Pierre Robert-Michon, Isaac Di Francesco

Edit History:
Isaac Di Francesco - 2019/10/26 - Filled in preliminary class layout & description of steps from a software POV.
Pierre Robert-Michon - 2019/10/27 - Reviewed preliminary class layout & description of steps from a software POV.
Pierre Robert-Michon - 2019/10/27 - Added preliminary software architecture description & flowchart.
Pierre Robert-Michon - 2019/10/27 - Added preliminary class hierarchy & description.
Brendan Marks - 2019/11/03 - Added JavaDoc API.
Aurelia Haas - 2019/11/23 - Added sections 6.0 and 7.0
Isaac Di Francesco - 2019/11/24 - Added flowchart for final demonstration
Aurelia Haas - 2019/11/27 - Detailed the final demonstration
Aurelia Haas - 2019/11/28 - Revision of the entire document

1.0 TABLE OF CONTENTS

2.0 SOFTWARE DESIGN

The software design for the final project will be a combination of the classes used for the first five labs leading up to the project.

2.1 CLASS LAYOUT

Preliminary class layout:

- ***Main.java*** → Used to dictate what the robot should do and run the various components of the final project competition. Defined information to display on the EV3 brick's LCD screen depending on the current status of the robot or which step of the competition it is currently performing.
- ***Resources.java*** → As in the first labs, this class will be used to contain various resources used in the other classes of the project (variables, instances of motors/sensors, etc.)
- ***Display.java*** → Used to display the odometer and readings obtained from the sensors on the EV3 brick's LCD screen. This class will be used mostly for testing/debugging purposes throughout the design weeks.
- ***UltrasonicPoller.java*** → Used to poll the ultrasonic sensor values, runs in a thread.
- ***LightsensorPoller.java***→ Used to poll the light sensor values, runs in a thread.
- ***UltrasonicLocalizer.java*** → Used to reorient the robot toward the 0-degree direction. Two methods can be used: falling edge (detects dips below a distance threshold) or rising edge (detects rises above a distance threshold).
- ***LightLocalization.java*** → Used to replace the robot at the point (1, 1) and the 0-degree orientation using Navigation and the light sensor.
- ***Navigation.java*** → Used to run the robot through different waypoints with minimal turn angle and distance.
- ***Odometer.java*** → Used to calculate the robots X and Y positions, as well as its orientation. These values are calculated through trigonometric calculations.

2.2 SOFTWARE PLAN

Given the current requirements for the project, the initial software plan consists of the following steps:

- Use ultrasonic localization (*UltrasonicLocalizer.java*) and light localization (*LightLocalizer.java*) to localize the robot in the starting corner (either red or green). By the end of this step, the robot will have its current X and Y positions as well as its orientation.
- With the values obtained in the previous step, the robot will travel (*Navigation.java*) to the entrance of the tunnel and orient itself to be directly facing it. If tests conducted during the design weeks demonstrate that the robot has difficulty traveling through the tunnel without touching the walls, a wall following class will be implemented similar to that of the first lab.

- Once the robot has made it to the island, it will localize (*LightLocalizer.java*) to the corner of a tile and orient itself to face the direction of the bin in which the ball must land. The robot will then travel (*Navigation.java)* to the specified launch point which will be calculated using the launch distance (from testing) and the bin location (BIN_x & BIN_y)
- Once at the launch point, the robot will launch a ball (or several if a reload mechanism is implemented).
- After the launching is complete, the robot will travel (*Navigation.java*) back to the tunnel and cross back to the starting corner the same way it first crossed this tunnel.
- The polling classes and the odometer class will be used throughout all of these steps for various tasks.

The following flowchart describes broadly what will be achieved during the final demo and the order in which is will be done.

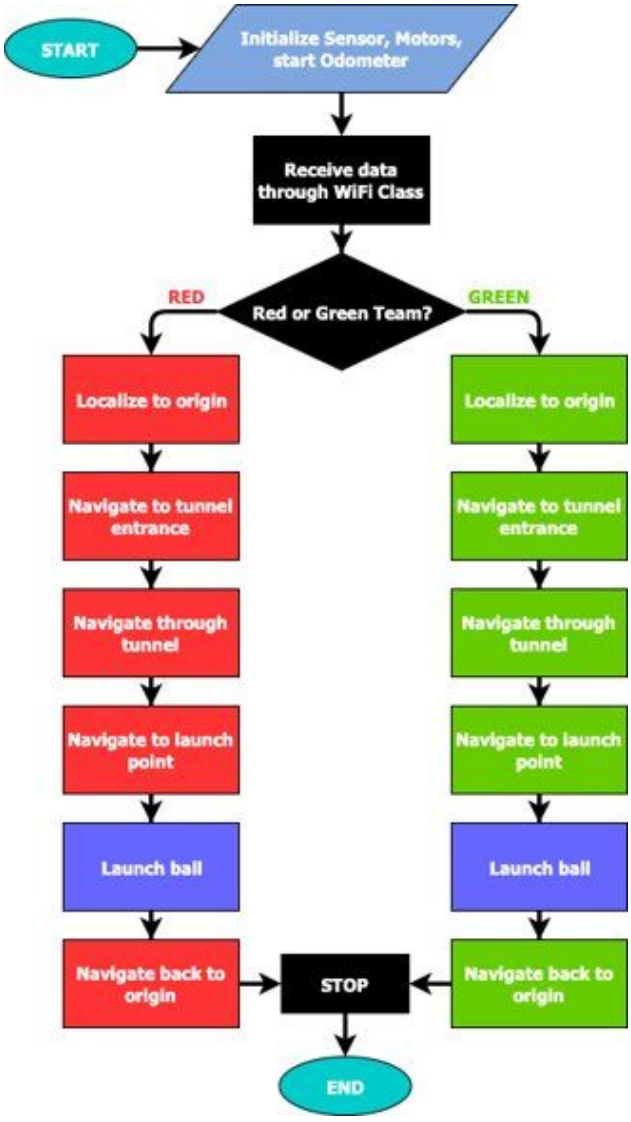

Fig. 2.2.1. Software plan for final demo

3.0 SOFTWARE ARCHITECTURE

The system's software architecture can be divided into two main sections. Firstly, there is the section responsible for completing the objectives and tasks that must be tackled by the robot in the chronological order detailed by the competition guidelines. Secondly, there is the section responsible for gathering the information in real time that is necessary to guide and correct the robot's actions as it completes these objectives and tasks. These can be referred to as the "sequential" and "ongoing" actions.

The flowchart below illustrates the software architecture of the robot in an exhaustive, task-focused and chronological manner, expressing both ongoing and sequential actions. From this flowchart, we can have an overview of the entirety of functionalities to implement, along with their chronological location and their interactions.

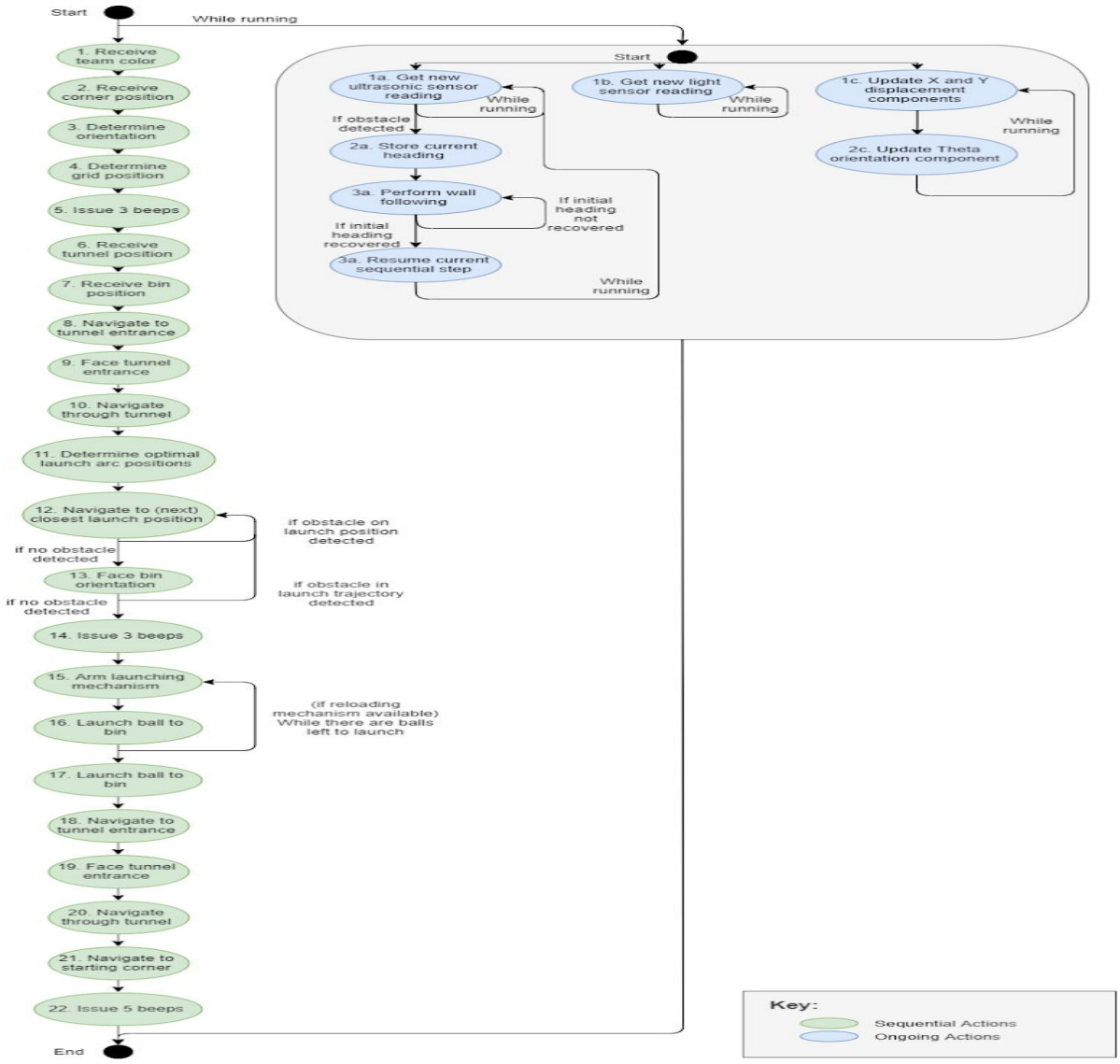

Fig 3.0.1. Preliminary Software Architecture Flowchart

4.0 SOFTWARE CLASS HIERARCHY

There are a number of observations to be made from the above flowchart. The primary observation is the distinction between the actions that must be performed continually, and those that must be performed sequentially. To do so, we must use java threads to implement continuous actions, such as sensor polling, displacement/orientation tracking (odometry) and obstacle avoidance. As for sequential actions, these will be implemented through classes and methods which will be called chronologically while the robot progresses through the course, from the main methods.

We can already derive the number of threads and classes to be implemented from the above observations. Indeed, we will be needing a thread for odometry, a thread for the polling of each sensor and potentially a thread for obstacle avoidance - although this could be done by directly using the ultrasonic poller thread for obstacle avoidance by triggering an obstacle avoidance/wall following class or method below a certain distance threshold. The total number of threads can be estimated to be at 1 (odometry) + 1 * #ultrasonicSensors + 1 * #lightSensors + 1 * threadedObstacleAvoidance, where #ultrasonicSensors and #lightSensors are integer valued, representing the number of implemented sensors of each type, and threadedObstacleAvoidance is boolean valued, representing the presence of an independent obstacle avoidance thread (as opposed to combined ultrasonic polling threads + obstacle avoidance/wall following class/methods).

For the necessary classes, building upon section 2.1, we will need a main method to orchestrate all actions and their chronology (and for actions 5, 14, 22), a resource class to store parameters (excluded from software architecture because it is not directly linked to the competition guidelines and requirements), instances and variables, the provided wifi class (for actions 1, 2, 6, 7), the ultrasonic localizer class (for action 3), the light localizer class (for action 4 and possible relocalizations during the course - prior to entering the tunnel, after exiting the tunnel, before launching), the navigation class for actions (8, 9, 10, 12, 13, 18, 19, 20, 21), a launcher class (for actions 11, 15, 16), an ultrasonic sensor polling class (for action 1a), a light sensor polling class (for action 1b), an odometer class (for actions 1c and 2c), and a wall following/obstacle avoidance class (for actions 2a, 3a). There is also the display class but it will be excluded from the software architecture since its function is to help in debugging and testing, and it is not relevant to the competition requirements. This brings us to a total of 10 classes, excluding the display and resource classes.

Class Name	Required Actions
Main	5, 14, 22 + orchestration
Wifi	1, 2, 6, 7
Light Localization	4 + potential relocalizations
Ultrasonic Localization	3

Navigation	8, 9, 10, 12, 13, 18, 19, 20, 21
Launcher	11, 15, 16
Ultrasonic Poller	1a
Light Poller	1b
Odometer	1c, 2c
Obstacle Avoidance	2a, 3a
Display	Debugging, Testing
Resource	Store variables, instances and parameters

Tab 4.0.1. Class layout and respective responsible actions

From the observations, constraints and requirements we have gathered, we can formulate the following hierarchy of classes, from which the software development will be driven. This class hierarchy expresses which classes should run in threads, and illustrates the interactions and dependencies between the different classes.

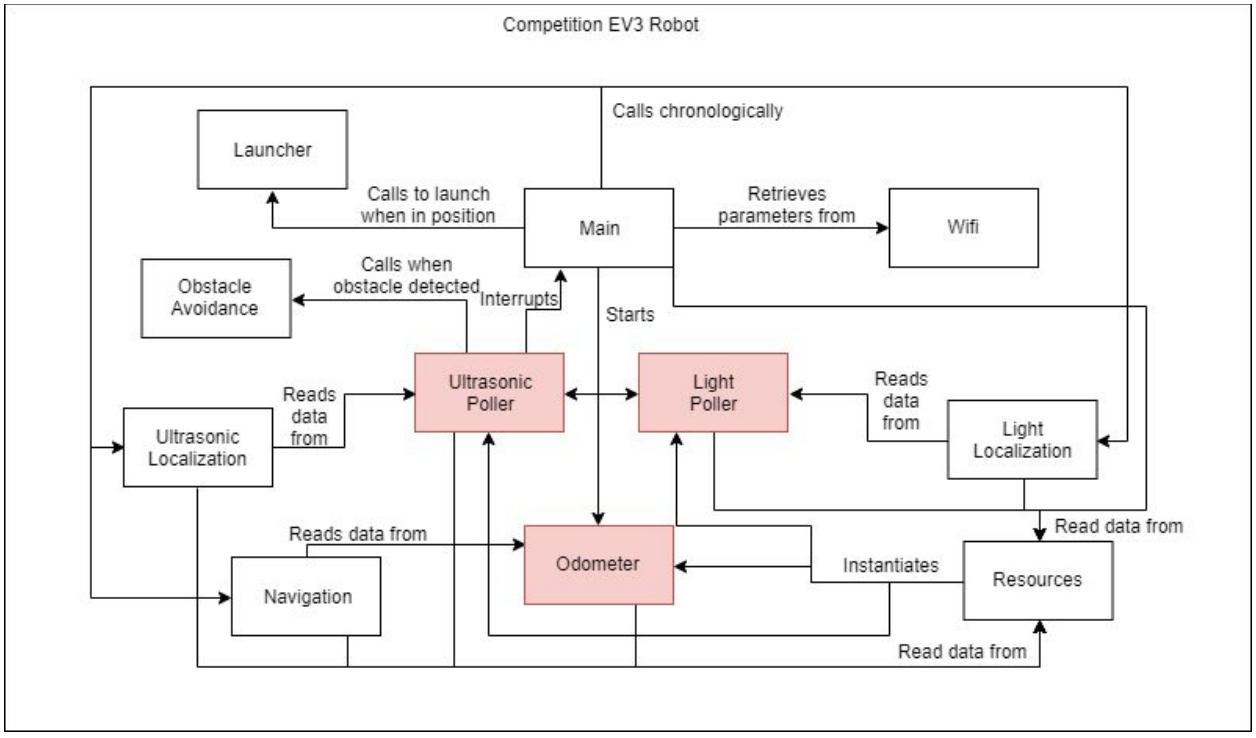

Fig 4.0.2. Preliminary Class Hierarchy and Interactions (classes in red run in threads)

5.0 PRELIMINARY JAVADOC API

Main.java:

The main driver class for the localization lab.

Constructor Summary

Constructors

Constructor	Description
`Main()`	

Method Summary

All Methods | **Static Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`private static int`	`chooseFallingOrRisingEdge()`	Asks the user whether the robot should using falling or rising edge.
`static void`	`main(java.lang.String[] args)`	The main entry point.
`static void`	`sleepFor(long duration)`	Sleeps current thread for the specified duration.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Display.java:

This class is used to display the content of the odometer variables (x, y, Theta)

Field Summary

Fields

Modifier and Type	Field	Desc
`private long`	`DISPLAY_PERIOD`	
`private double[]`	`position`	
`private long`	`timeout`	

Constructor Summary

Constructors

Constructor	Description
`Display()`	

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`void`	`run()`	
`void`	`setTimeout(long timeout)`	Sets the timeout in ms.
`static void`	`showText(java.lang.String... strings)`	Shows the text on the LCD, line by line.

LightLocalization.java:

This class is used to localize the robot using the light sensor and make it navigate to the origin (0,0)

Constructor Summary

Constructors

Constructor	Description
`LightLocalization()`	

Method Summary

All Methods | **Static Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`static int`	`convertAngle(double angle)`	Converts input angle to the total rotation of each wheel needed to rotate the robot by that angle.
`static int`	`convertDistance(double distance)`	Converts input distance to the total rotation of each wheel needed to cover that distance.
`static void`	`findOrigin()`	Corrects the X, Y and theta of the robot then navigates to origin

LightsensorPoller.java:

This class samples the light sensor and periodically updates the red light value.

Field Summary

Fields

Modifier and Type	Field	Description
`private float[]`	`csData`	
`private int`	`redValue`	

Constructor Summary

Constructors

Constructor	Description
`LightsensorPoller()`	Constructs an LightsensorPoller object.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`int`	`getRedValue()`	Get the current red value measured by the Light Sensor Accessible from outside of class
`void`	`run()`	

Navigation.java:

This class is used to drive the robot on the demo floor through the map's waypoints.

Field Summary

Fields

Modifier and Type	Field	Desc
private static double	currentTheta	
private static boolean	navigating	
private static double	xPosition	
private static double	yPosition	

Constructor Summary

Constructors

Constructor	Description
Navigation()	

Method Summary

All Methods | Static Methods | Concrete Methods

Modifier and Type	Method	Description
static void	avoidObstacle()	Avoid the obstacle ahead (a 14.3 by 23 cm block)
static int	convertAngle(double angle)	Converts input angle to the total rotation of each wheel needed to rotate the robot by that angle.
static int	convertDistance(double distance)	Converts input distance to the total rotation of each wheel needed to cover that distance.
static boolean	isNavigating()	Signal whether robot is currently mvoing or not
static void	travelTo(double x, double y)	Navigate to the given X and Y coordinates
static void	turnTo(double theta)	Change the current heading according to specified heading change
private static void	updateCoordinates()	Update the current odometer data.
static void	updateTurningCorrection(int direction)	Converts input angle to the total rotation of each wheel needed to rotate the robot by that angle.

Odometer.java:

Fields

Modifier and Type	Field	Description
private java.util.concurrent.locks.Condition	doneResetting	Lets other threads know that a reset operation is over.
private boolean	isResetting	Indicates if a thread is trying to reset any position parameters
private static java.util.concurrent.locks.Lock	lock	Fair lock for concurrent writing
private double	theta	The orientation in degrees.
private static double	turningCorrection	
private double	x	The x-axis position in cm.
private double	y	The y-axis position in cm.

Constructor Summary

Constructors

Constructor	Description
Odometer()	Constructs an Odometer object.

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
double	getTheta()	Returns the Odometer Theta data.
static double	getTurningCorrection()	Returns the current turning correction.
double	getX()	Returns the Odometer X data.
double[]	getXYT()	Returns the Odometer data.
double	getY()	Returns the Odometer Y data.
void	run()	This method is where the logic for the odometer will run.
void	setTheta(double theta)	Overwrites theta.
static void	setTurningCorrection(double correction)	Overwrites turning correction.
void	setX(double x)	Overwrites x.
void	setXYT(double x, double y, double theta)	Overrides the values of x, y and theta.
void	setY(double y)	Overwrites y.
void	update(double dx, double dy, double dtheta)	Adds dx, dy and dtheta to the current values of x, y and theta, respectively.

UltrasonicLocalizer.java:

This class is used to localize the robot on the demo floor using the ultrasonic sensor. It updates theta with an estimate (5 degrees precision) of the robots heading.

Field Summary

Fields

Modifier and Type	Field	Description
private static int	d	
private static double	dTheta	
private static int	k	
private static double	tempTheta1	
private static double	tempTheta2	
private static double	theta1	
private static double	theta2	

Constructor Summary

Constructors

Constructor	Description
UltrasonicLocalizer()	

Method Summary

All Methods | **Static Methods** | **Concrete Methods**

Modifier and Type	Method	Description
static int	convertAngle(double angle)	Converts input angle to the total rotation of each wheel needed to rotate the robot by that angle.
static int	convertDistance(double distance)	Converts input distance to the total rotation of each wheel needed to cover that distance.
static void	fallingEdge()	Performs falling edge localization: takes the average angle of detection of the headings when the falling edge threshold is reached (wall detected)
static void	risingEdge()	Performs rising edge localization: takes the average angle of detection of the headings when the rising edge threshold is reached (wall no longer detected)

UltrasonicPoller.java:

Samples the US sensor and invokes the selected controller on each cycle. Control of the wall follower is applied periodically by the UltrasonicPoller thread. The while loop at the bottom executes in a loop. Assuming that the us.fetchSample, and cont.processUSData methods operate in about 20ms, and that the thread sleeps for 50 ms at the end of each loop, then one cycle through the loop is approximately 70 ms. This corresponds to a sampling rate of 1/70ms or about 14 Hz.

Field Summary

Fields

Modifier and Type	Field	Description
private boolean	avoiding	
private int	distance	
private float[]	usData	

Constructor Summary

Constructors

Constructor	Description
UltrasonicPoller()	Constructs an UltrasonicPoller object.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
boolean	getAvoiding()	Signal whether robot is currently avoiding an obstacle or not
int	getDistance()	Get the current distance measured by the Ultrasonic Sensor Accessible from outside of class
void	run()	
void	setAvoiding(boolean newAvoiding)	Update the signal whether robot is currently avoiding an obstacle or not

6.0 BETA DEMONSTRATION SOFTWARE

During the beta demonstration the goal of the robot was to localize in less than 30 seconds, then navigate to a tunnel, go through it and then reach the bin position to launch a ball in the right direction. The main issue was to use the wifi class given and link it to the current code.
For the API, refer to the folder: Javadocs Beta Demo.

7.0 FINAL DEMONSTRATION SOFTWARE

The goal of the final software was to add a few components as ObstacleAvoidance, OdometryCorrection as well as be careful with the new parameter added to the WiFi class: 2 bins (1 for each team).
For the API, refer to the folder: Javadocs Final Demo.
For the final code, refer to the Code - src folder.

Realizing the final software was extremely technical. Indeed a lot of different actions were to be realized in a few different cases.

1. The robot, to properly localize, needs to use 2 US sensors and realizes falling edge. Then to realize odometry corrections, 2 light Sensors are used (this function is used a lot during the whole demo.
2. To reach the tunnel, the robot follows the lines to keep a pretty accurate odometry correction and faces the tunnel almost perfectly in the middle of the square before the tunnel.
3. After going through the tunnel, the robot localizes again and realizes odometry correction.
4. The code is created to calculate a few number of launch positions inside the island. The robot, by following the lines, tries to reach the closest launch position.
5. If any obstacle is in its way, the robot avoids it by trying to reach another launch position.
6. Once the launch position reached, the robot turns to the right angle to launch five balls into the bin.
7. The robot realizes the inverse path as it just did to reach the end of the tunnel.
8. The same format as the 2 point is used but this time to reach the corner.

8.0 GLOSSARY OF TERMS

API: Application programming interface
Odometer/Odometry: instrument for measuring the distance traveled by a vehicle