Software document

Author: Maxime Cardinal
Date: October 30th, 2018
Version: 5.0
Edit history: [Version system is WEEK#.EDIT#]
October 30, 2018 - Maxime Cardinal - Creation of the document, preliminary content - 2.1
October 30, 2018 - Spencer Handfield - Adjusted formatting for consistency with other docs and added table of contents - 2.2
November 5, 2018 - Maxime Cardinal - Modification of TravelToTree class description, Addition of the TravelToTree and travelToBridge flowcharts - 3.1
November 5, 2018 - Spencer Handfield - Elaboration of findings and design flow/logic of certain classes - 3.2
November 5, 2018 - Irmak Pakis - Added to OdometerCorrection - 3.3
November 6, 2018 - Spencer Handfield - Reformatted certain section to attempt to better illustrate the week by week design process logic - 3.4
November 11, 2018 - Spencer Handfield - Addition of week 4 implementation/modification of software - 4.1
November 13, 2018 - Spencer Handfield - modified section to reflect project description v2.0 - 4.2
November 15, 2018 – Maxime Cardinal – Modification of classes and update of the software architecture due to beta demo results - 5.0

1.0 Table of contents

2.0 Functionality

The robot must execute many independent tasks to reach its goal. First, the robot must receive the competition parameters via Wifi and localize itself in the grid. Then, it must travel to the tunnel, cross that tunnel and reach for its team corresponding tree, while correcting its position. Afterward, it must retrieve one or many rings and identify their corresponding color. Finally, the robot must travel back to its original position via the tunnel and unload the ring(s) it retrieved (see Figure 1).

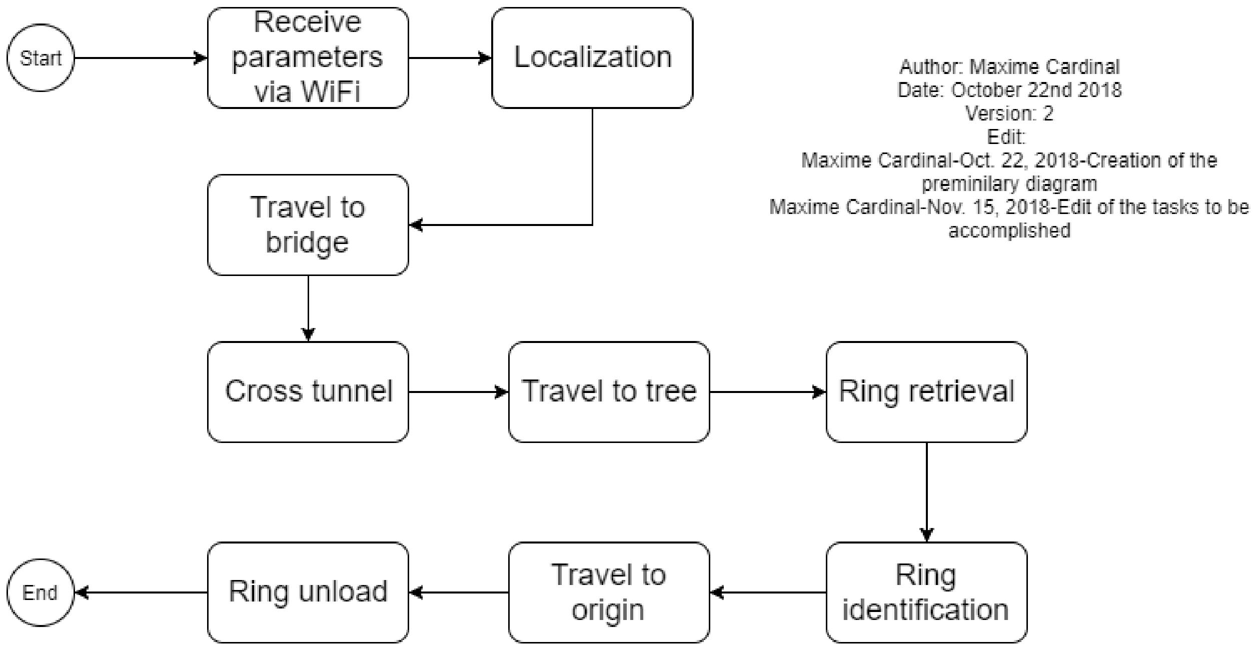

Figure 1 - Functionality flow chart

The following pages further explain how those tasks are executed by the software and which classes are responsible for handling those tasks.

3.0 Classes

3.1 DesignProjectMain

The “DesignProjectMain” class is the main class of the project. It is responsible for launching the program, initializing all the sensors, all the motors and the important constants. Also, this class is responsible for starting required threads and calling required method from other classes. We decided to initialize all sensors and motors in this class to minimize our time consumption when starting new threads. Furthermore, by calling methods from other classes instead of creating threads, we reduced the robot time consumption and faster the software.

//To be added: flowchart of the class -> will be added once software is completed

3.2 OdometerData

The “OdometerData” class is responsible of keeping track of the robot’s location and orientation. It stores and provides a save access to the odometer data. It contains methods such as “getXYT()”, “update(double dx, double dy, double dtheta)”, “setXYT(double x, double y, double theta)”, “setX(double x)”, “setY(double y)” and “setTheta(double theta)”, which can be used to access odometer data easily. This class has been reused from previous lab.

3.3 Odometer

The “Odometer” class is responsible of updating the odometer data according to the robot’s wheels displacement. This class extends the “OdometerData” class and has been reused from the previous lab we did. The “Odometer” class is running as a thread by the main class “DesignProjectMain” and runs until the end of the whole program.

3.4 OdometerExceptions

The “OdometerExceptions” class is used to handle errors regarding the singleton pattern used for the odometer and “OdometerData”. This class has been reused from the previous lab and was provided to us.

3.5 Wifi

The “Wifi” class is responsible for connecting the EV3 brick to a server, receive the game parameters and assign the required game parameters depending on the team’s color. The “Wifi” class make use of an imported library to create a WifiConnection and receive the data from the server. Both the class and the library were provided, but we had to change the Wifi class so that is assign only the desired parameters to the project. The Wifi class first retrieves the data from the server, then compares the team number to each team number (green/red) and assigns the parameters according to the team color.

3.6 Localization

The “Localization” class is responsible of correcting the initial position and the orientation of the robot to be (0,0) and 0-degree respectively. This class has been implemented using the class “UltrasonicLocalizer” and “LightLocalizer” from Lab5. The “UltrasonicLocalizer” class was responsible of correcting the robot orientation, making use of an ultrasonic sensor and the “LightLocalizer” class was responsible of correcting the initial position of the robot making use of a light sensor. We decided to merge these two classes into one to minimize the time consumption of the process. By merging these classes, we reduce the number of classes needed by one, thus reducing the time needed to initialize sensors, variables and constants. The code has been further simplified to increase its readability by making use of multiple methods. This class contains is separated into two main methods: usLocalization() and lsLocalization().

3.6.1 usLocalization()

The “usLocalization()” method is responsible for correcting the initial angle of the robot in the grid. To do so, this method makes use of an ultrasonic sensor to detect its position relative to the walls. Using a falling edge algorithm, the robot will compute the angle difference between the left and right falling edge to determine its orientation in the grid (see Figure 2).

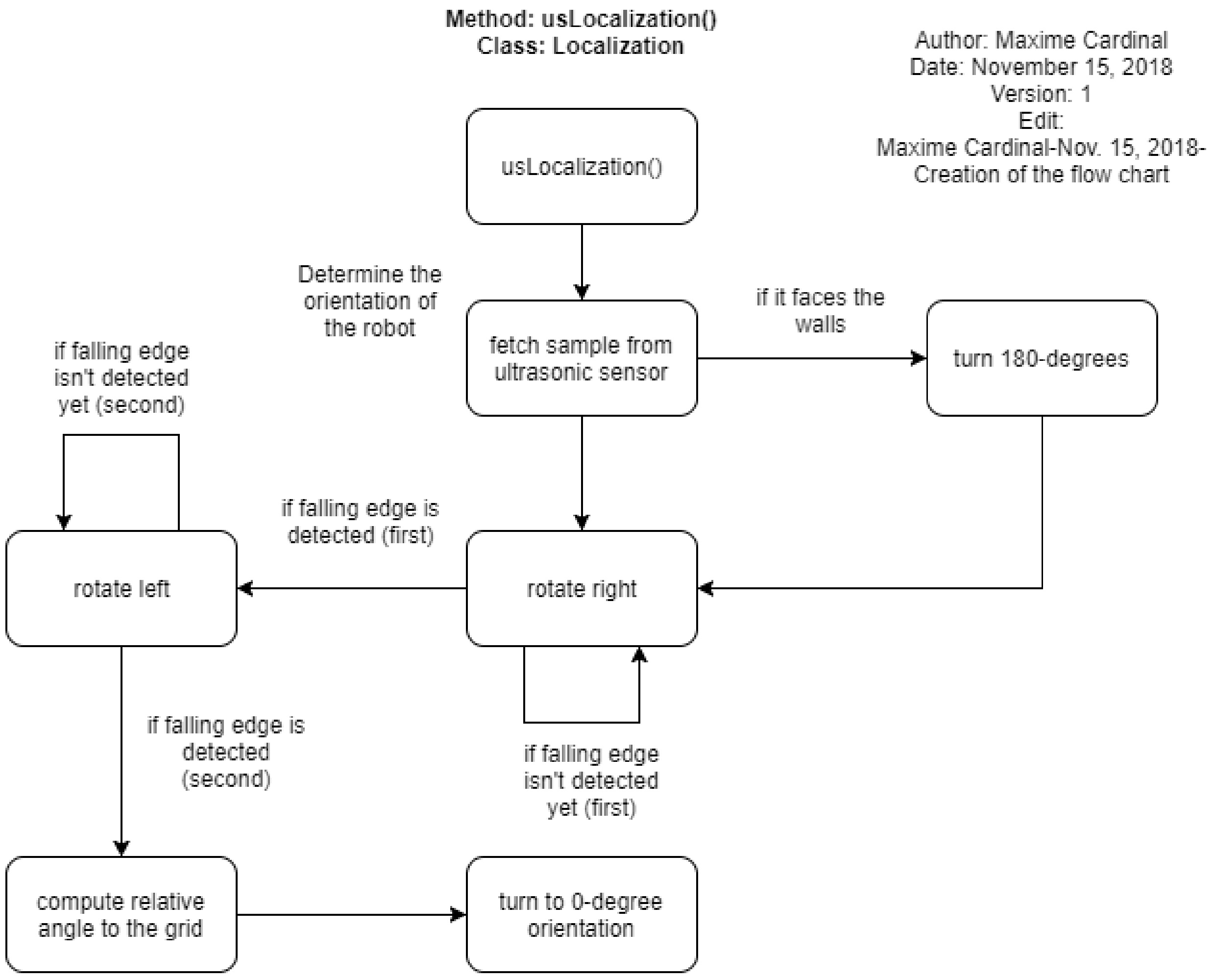
Method: usLocalization()
Class: Localization
Author: Maxime Cardinal
Date: November 15, 2018
Version: 1
Edit:
Maxime Cardinal-Nov. 15, 2018-
Creation of the flow chart
usLocalization()
Determine the orientation of the robot
fetch sample from ultrasonic sensor
if it faces the walls
turn 180-degrees
if falling edge isn't detected yet (second)
if falling edge is detected (first)
rotate left
rotate right
if falling edge is detected (second)
if falling edge isn't detected yet (first)
compute relative angle to the grid
turn to 0-degree orientation

Figure 2 - usLocalization Flow Chart

To filter the sensor samples, we restrained the falling edge detection to be from 0 to 40 cm, thus reducing the probability of detecting false falling edge at long distances. Since falling edge implementation requires the robot to be facing the outside of its starting corner, the sensor first detects if it is facing the good direction or not. If not, it will perform a 180-degree rotation before starting the falling edge detection (see Figure 3).

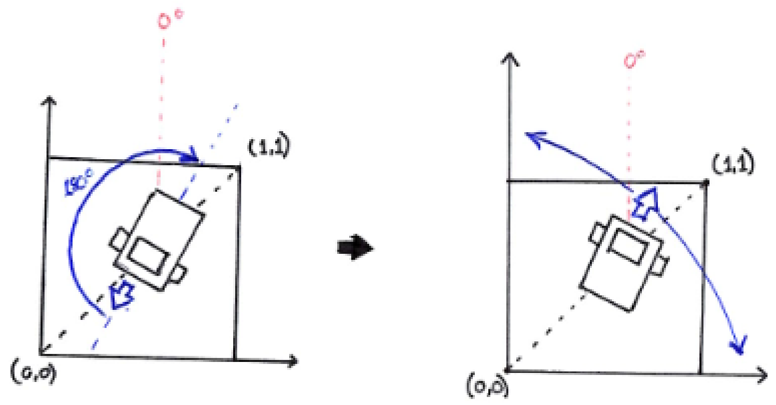

Figure 3 - Falling edge: Starting orientation correction

3.6.1 lsLocalization()

The "lsLocalization()" method is responsible for correcting the initial position of the robot to its closest grid intersection. To facilitate the understanding, the starting corner of the robot will be defined as 0 and its closest grid intersection will be defined as (1,1) for the following explanations. To correct the robot's position, we used to only use one light sensor. At first, the light sensor localization algorithm was simple: the robot moved straight forward until it detected a line with the left light sensor, then it turned 90-degrees right and repeated the same process to correct its position to be (1,1) (see Figure 4).

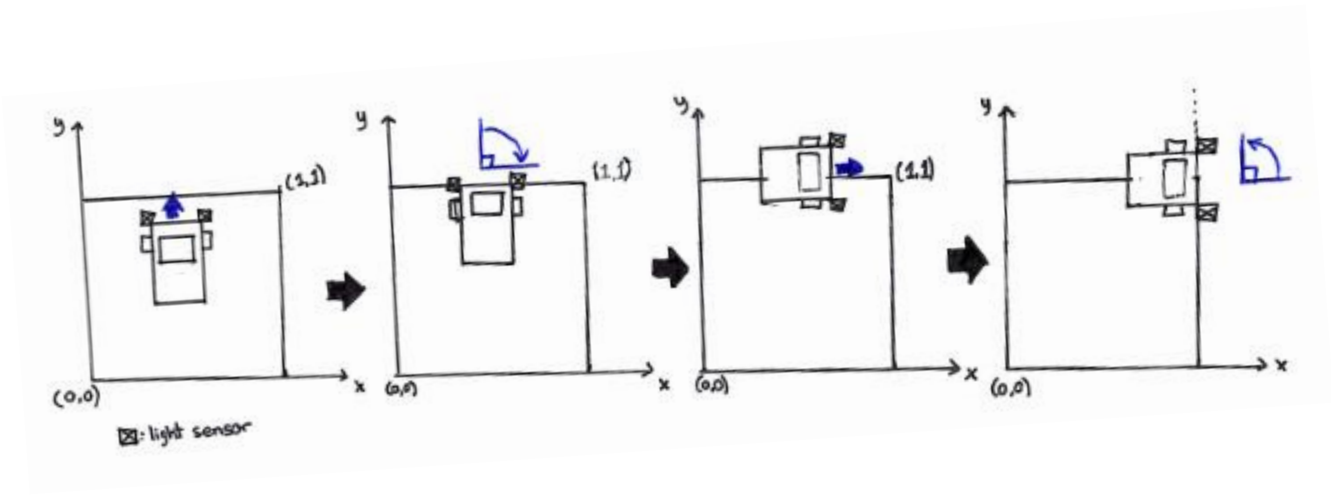

Figure 4 - initial light sensor localization algorithm

After review, it was concluded that this algorithm wasn't a reliable position correction, as the its totally depends on the ultrasonic localization accuracy. An offset in the ultrasonic localization would directly impacts the precision of this method, which is an undesirable effect and had to be corrected. Thereby, we came up with our second light sensor localization, which made use of two light sensors and a more complex algorithm.

Instead of assuming the orientation of the robot is good when crossing the lines, the robot makes use of the light sensors to correct its orientation again. The algorithm goes as follow: The robot moves forward until one of the sensors detects a line. Then, it stops the left or right sensor according to the one that detected the line. Finally, it turns until the second sensor detects the line (see Figure 5).

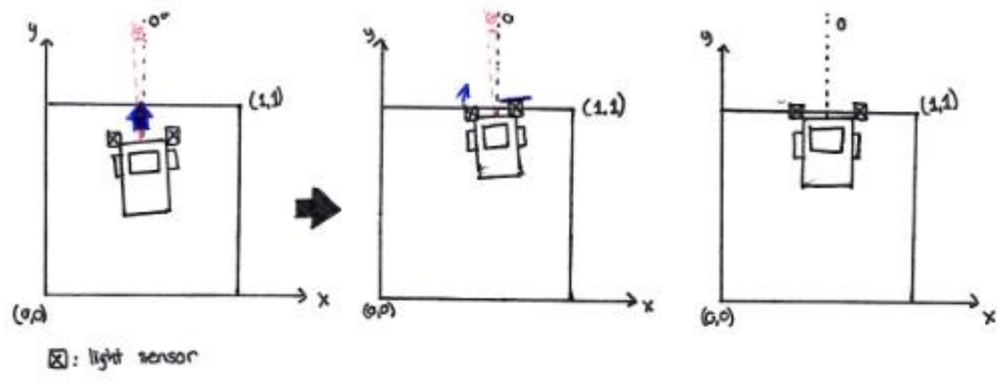

Figure 5 - Second light sensor localization algorithm

This light sensor localization ensures that the final position and orientation of the robot after executing the localization are respectively (1,1) and 0-degree (see Figure 6).

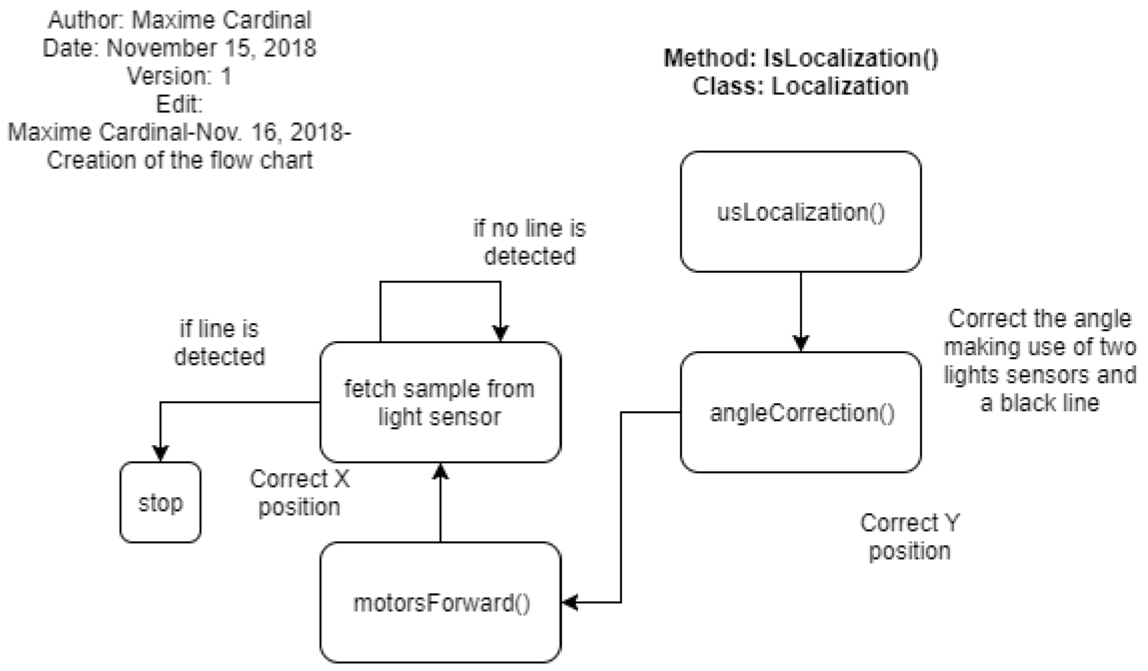

Figure 6 - lsLocalization Flow Chart