SOFTWARE DOCUMENTATION

Team 14

Sebastian Andrade
Alexander Bratyshkin
Matthew Lesko
Ece Pidik
Felix Weiping Ren
Jeffrey Tsui
Stone Yun
Edward Zhao

Version 2.0
December 1, 2016

1.0 Table of Contents

2.0 Software Description

This section of the Software Documentation contains a brief explanation of all the important classes used in our software. For more ***detailed*** *explanations of the classes themselves, please refer to the* ***Javadoc API*** *documentation.*

2.1 StartRobot

This is the class which contains the *main()* method, therefore it is from here that we are going to execute all subsequent classes and methods needed to run our robot.

2.2 RobotMovement

This class essentially coordinates all of the robot's operations. It executes our search algorithm to find an object in its path, from which we then snap into the Navigation class once the maximum amount of blue blocks has been picked up to travel to the designated zone and then restart the algorithm again. The differentiation between the type of blocks (see object detection algorithm) will also be done in this class, as well as the control of the pulley motor and the claw (see object grabbing algorithm).

2.3 Odometer

This class will calculate the robot's X and Y coordinates and angle from its location relative to the starting point (see odometer algorithm).

2.4 OdometerCorrection

This class will correct any inaccuracy in the odometer caused by factors such as wheel slippage and others, with the help of a light sensor. This method attempts to correct the odometer values based on the occurrence of lines on the board (see odometer correction algorithm).

2.5 LSPoller

This class polls data from the light sensor.

2.6 USPoller

This class polls data from the ultrasonic sensors independently of each other.

2.7 Navigation

This class allows the robot to travel to specific X and Y coordinates in the arena, and also to turn to a desired angle from 0 to 360 degrees. It also grants the robot the ability to approach an object by simply moving forward towards it (see turn to angle algorithm and travel to location algorithms).

2.8 USLocalizer

This class is responsible for performing the ultrasonic localization (see localization algorithm). It can perform two types of localization, falling edge and rising edge, which both depend on two consecutive readings of the walls at the X1-X4 starting corners (see Requirements Document section 2.2)

2.9 LightLocalizer

This class is responsible for performing the light sensor localization after the ultrasonic localization is completed by reading four consecutive black grid lines on the floor while rotation around a desired (0, 0) point (see localization algorithm).

2.10 Vector

This class is simply defines a template for a vector, which includes a magnitude, an angle, and an initial X and initial Y readings to know from which point the vector was measured.

3.0 Class Diagram

This section of the Software Documentation contains a Unified Modeling Language (UML) class diagram. It describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects. It also displays how small software blocks come together to make the system a whole.

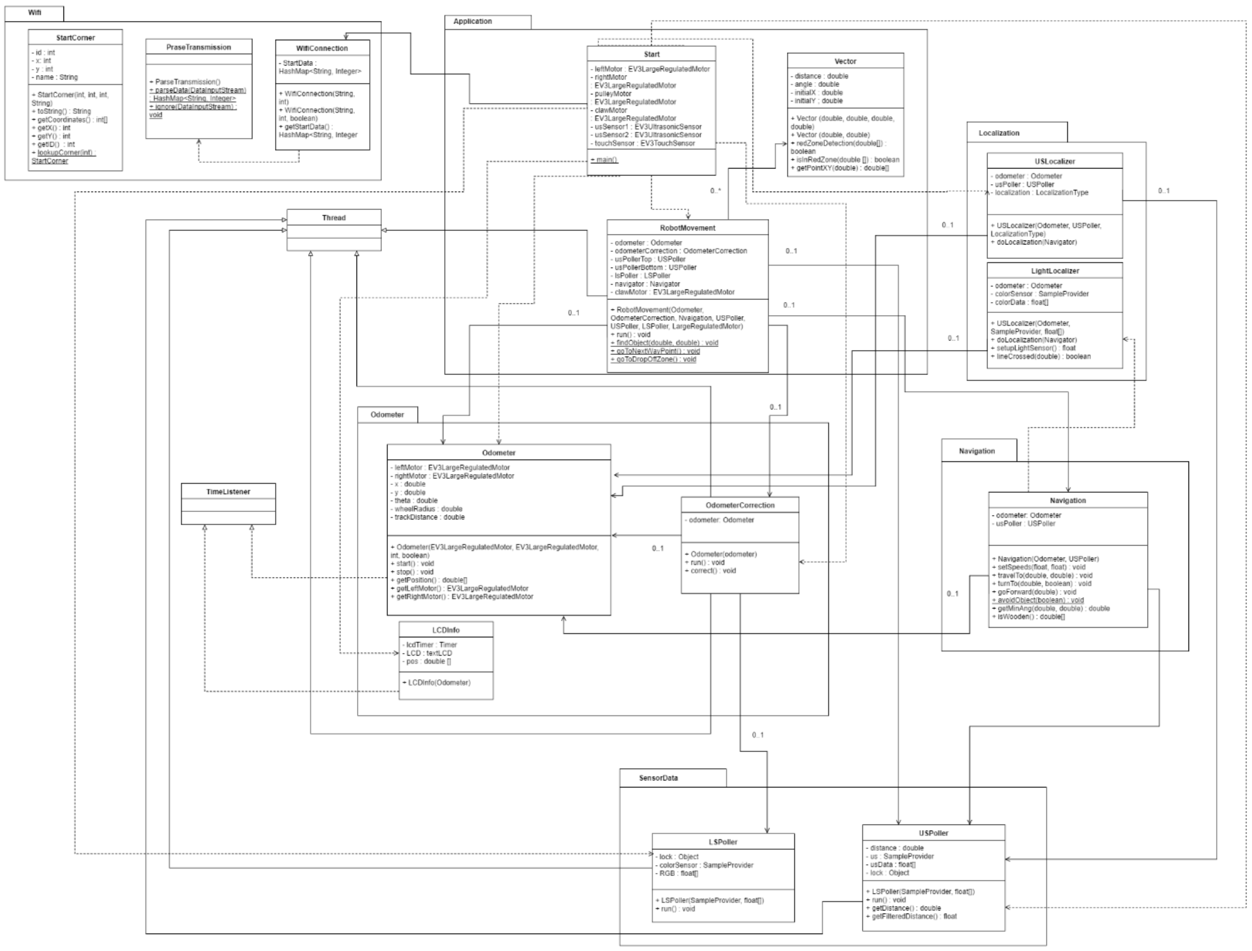

Wifi
StartCorner
- id : int
- x: int
- y : int
- name : String
+ StartCorner(int, int, int, String)
+ toString() : String
+ getCoordinates() : int[]
+ getX() : int
+ getY() : int
+ getID() : int
+ lookupCorner(int) : StartCorner
PraseTransmission
+ ParseTransmission()
+ parseData(DataInputStream) : HashMap<String, Integer>
+ ignore(DataInputStream) : void
WifiConnection
- StartData : HashMap<String, Integer>
+ WifiConnection(String, int)
+ WifiConnection(String, int, boolean)
+ getStartData() : HashMap<String, Integer
Application
Start
- leftMotor : EV3LargeRegulatedMotor
- rightMotor : EV3LargeRegulatedMotor
- pulleyMotor : EV3LargeRegulatedMotor
- clawMotor : EV3LargeRegulatedMotor
- usSensor1 : EV3UltrasonicSensor
- usSensor2 : EV3UltrasonicSensor
- touchSensor : EV3TouchSensor
+ main()
Vector
- distance : double
- angle : double
- initialX : double
- initialY ; double
+ Vector (double, double, double, double)
+ Vector (double, double)
+ redZoneDetection(double[]) : boolean
+ isInRedZone(double []) : boolean
+ getPointXY(double) : double[]
RobotMovement
- odometer : Odometer
- odometerCorrection : OdometerCorrection
- usPollerTop : USPoller
- usPollerBottom : USPoller
- lsPoller : LSPoller
- navigator : Navigator
- clawMotor : EV3LargeRegulatedMotor
+ RobotMovement(Odometer, OdometerCorrection, Nvaigation, USPoller, USPoller, LSPoller, LargeRegulatedMotor)
+ run() : void
+ findObject(double, double) : void
+ goToNextWayPoint() : void
+ goToDropOffZone() : void
0..*
0..1
Thread
Localization
USLocalizer
- odometer : Odometer
- usPoller : USPoller
- localization : LocalizationType
+ USLocalizer(Odometer, USPoller, LocalizationType)
+ doLocalization(Navigator)
LightLocalizer
- odometer : Odometer
- colorSensor : SampleProvider
- colorData : float[]
+ USLocalizer(Odometer, SampleProvider, float[])
+ doLocalization(Navigator)
+ setupLightSensor() : float
+ lineCrossed(double) : boolean
Odometer
Odometer
- leftMotor : EV3LargeRegulatedMotor
- rightMotor : EV3LargeRegulatedMotor
- x : double
- y : double
- theta : double
- wheelRadius : double
- trackDistance : double
+ Odometer(EV3LargeRegulatedMotor, EV3LargeRegulatedMotor, int, boolean)
+ start() : void
+ stop() : void
+ getPosition() : double[]
+ getLeftMotor() : EV3LargeRegulatedMotor
+ getRightMotor() : EV3LargeRegulatedMotor
OdometerCorrection
- odometer: Odometer
+ Odometer(odometer)
+ run() : void
+ correct() : void
TimeListener
LCDInfo
- lcdTimer : Timer
- LCD : textLCD
- pos : double []
+ LCDInfo(Odometer)
Navigation
Navigation
- odometer: Odometer
- usPoller : USPoller
+ Navigation(Odometer, USPoller)
+ setSpeeds(float, float) : void
+ travelTo(double, double) : void
+ turnTo(double, boolean) : void
+ goForward(double) : void
+ avoidObject(boolean) : void
+ getMinAng(double, double) : double
+ isWooden() : double[]
SensorData
LSPoller
- lock : Object
- colorSensor : SampleProvider
- RGB : float[]
+ LSPoller(SampleProvider, float[])
+ run() : void
USPoller
- distance : double
- us : SampleProvider
- usData : float[]
- lock : Object
+ LSPoller(SampleProvider, float[])
+ run() : void
+ getDistance() : double
+ getFilteredDistance() : float

Figure 3.0.1 – UML Class Diagram

4.0 Software Architecture

This section of the Software Documentation includes a diagram and explanation of the software architecture. It defines how the system is organized by breaking down the system into smaller modules that communicate with each other.

We used layered architecture. The software system was broken down into two layers containing modules similar in functionality. The platform layer accesses and controls the hardware. It receives information from the sensors: it uses the light sensor for color detection and ultrasonic sensor for distance detection. Motor control is also handled in this layer. Manipulating the motors connected to the wheels changes the speed and the direction of the robot. Motors connected to the pulley has to be manipulated to raise or lower the Styrofoam blocks. Motors connected to the claw has to be manipulated for the pick-up mechanism to work.

Second layer is the algorithm layer. This part of the software system contains the algorithms that make the robot do certain tasks. Localization, search and path planning, object detection and avoidance (both forbidden zone and object), navigation, odometry correction, and object pick up make up this layer.

In layered software architecture, one layer can only interact with the layer below and above. Because our system only has two layers, the difficulty of establishing communication between a higher-level layer and a lower-level layer is eliminated. Both layers are always interacting with each other as the hardware has to constantly be accessed and controlled for the modules in the algorithm layer to complete tasks.

Additionally, we were required to implement, by the constraints of the project (see Requirements Document section 2.2), a client-server architecture. The client-server architecture model distinguishes, via network communication, between two types of computers or processes: a client and a server. Servers are generally used for storage of data, while clients are used for manipulation of data. An application which implements the client-server architecture is a distributed system which is composed of both, client and server software. In our case, the EV3 brick is going to be acting as a client, while the server is going to be set up by the organizers of the final competition. Through a common router connection, we will have to, from the server, retrieve the parameters specified in the Requirements Document, section 2.2, which will let the robot (client) know what its predetermined role is, to consequently be aware of its drop-off and danger zones.

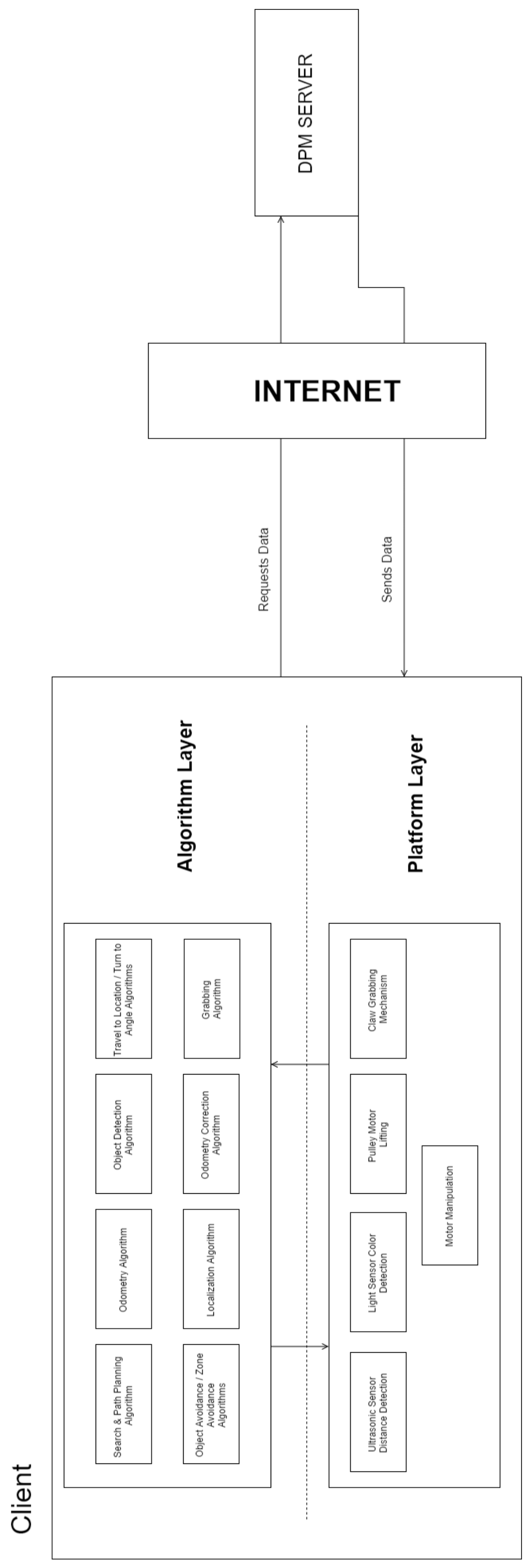

DPM SERVER
INTERNET
Requests Data
Sends Data
Algorithm Layer
Platform Layer
Travel to Location / Turn to Angle Algorithms
Object Detection Algorithm
Odometry Algorithm
Search & Path Planning Algorithm
Grabbing Algorithm
Odometry Correction Algorithm
Localization Algorithm
Object Avoidance / Zone Avoidance Algorithms
Claw Grabbing Mechanism
Pulley Motor Lifting
Light Sensor Color Detection
Ultrasonic Sensor Distance Detection
Motor Manipulation
Client

Figure 4.0.1 – System Architecture Diagram

5.0 Behavioral Flowchart

This section of the Software Documentation includes a diagram and explanation of the overall behaviour of our robot, i.e. the functionalities that the robot is supposed to perform to comply with the Requirements Document and their interactions with each other.

The flowchart below demonstrates the behavior of the robot. The robot is fully automated once the user starts it. It will begin with the falling edge localization. When the localization is complete, the robot goes into the search algorithm loop. At this point, it will search the grid to find blue Styrofoam blocks. If it finds one while already holding a block or blocks, the current ones will be lifted and the Styrofoam block on the grid will be picked up. Otherwise the robot will simply pick up the Styrofoam block. Odometry correction is performed at every step of the process.

The robot also has to perform object avoidance. The object avoidance starts when the robot is searching for Styrofoam blocks and comes across a wooden block and/or a forbidden area. Once the object avoidance is completed, the robot will proceed with the search algorithm.

The last part of the search algorithm is navigating to the designated zone. This part starts when the robot has collected enough Styrofoam blocks and the block tower has reached its max capacity. At this point robot has to do one of two things depending on its current role. If the robot's current role is 'garbage collector', then it navigates to the red zone. If the robot's current role is 'builder, then it navigates to the green zone. When the navigation is complete the robot once again goes to the beginning of the search algorithm.

References to localization, odometry correction, search and path planning, object detection, object avoidance, forbidden zone avoidance and object grabbing algorithms are available in the diagram.

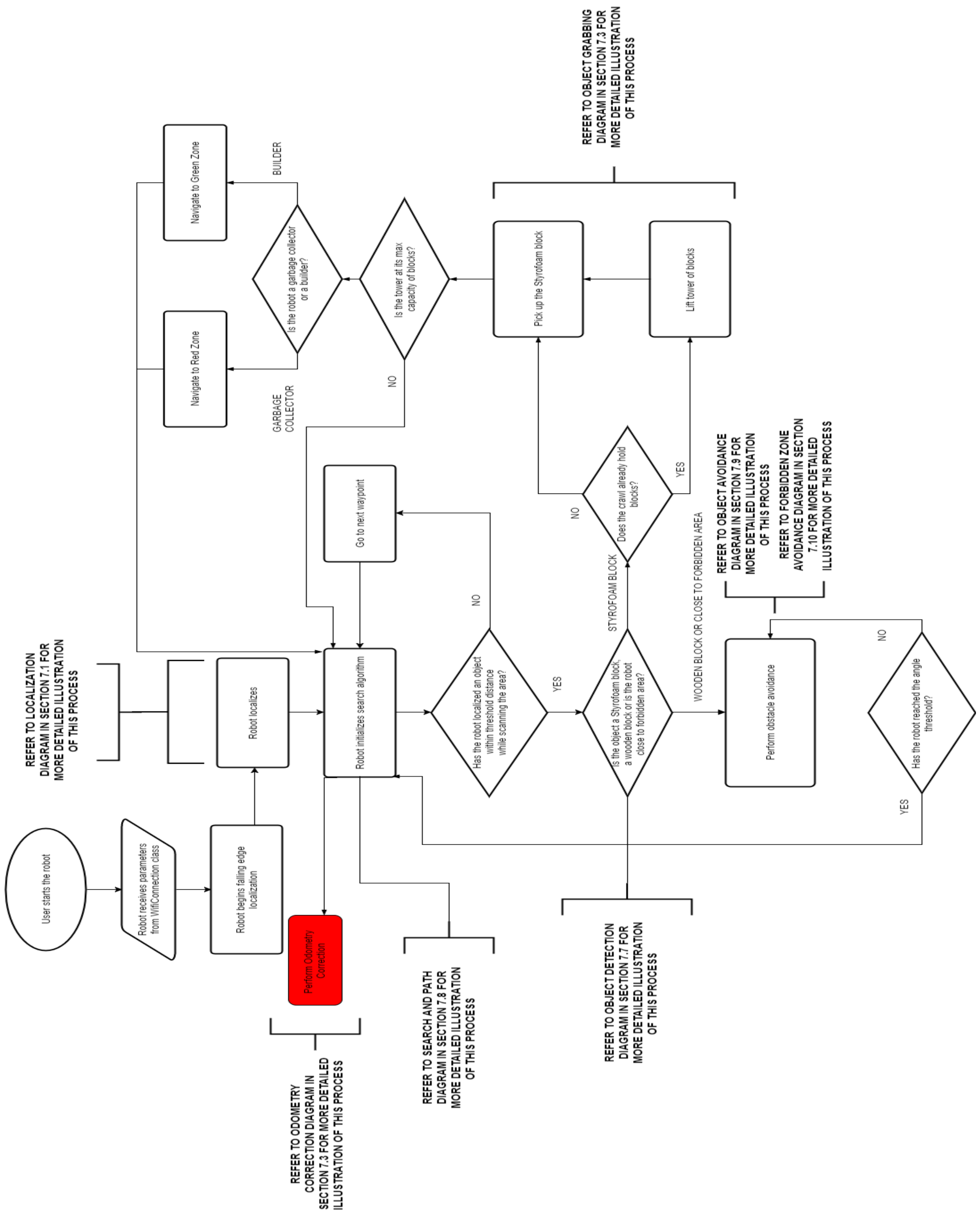

User starts the robot
Robot receives parameters from WifiConnection class
Robot begins falling edge localization
Robot localizes
REFER TO LOCALIZATION DIAGRAM IN SECTION 7.1 FOR MORE DETAILED ILLUSTRATION OF THIS PROCESS
Perform Odometry Correction
REFER TO ODOMETRY CORRECTION DIAGRAM IN SECTION 7.3 FOR MORE DETAILED ILLUSTRATION OF THIS PROCESS
Robot initializes search algorithm
REFER TO SEARCH AND PATH DIAGRAM IN SECTION 7.8 FOR MORE DETAILED ILLUSTRATION OF THIS PROCESS
Go to next waypoint
Has the robot localized an object within threshold distance while scanning the area?
NO
YES
Is the object a Styrofoam block, a wooden block or is the robot close to forbidden area?
REFER TO OBJECT DETECTION DIAGRAM IN SECTION 7.7 FOR MORE DETAILED ILLUSTRATION OF THIS PROCESS
STYROFOAM BLOCK
WOODEN BLOCK OR CLOSE TO FORBIDDEN AREA
Does the crawl already hold blocks?
NO
YES
Pick up the Styrofoam block
Lift tower of blocks
REFER TO OBJECT GRABBING DIAGRAM IN SECTION 7.3 FOR MORE DETAILED ILLUSTRATION OF THIS PROCESS
Is the tower at its max capacity of blocks?
NO
Is the robot a garbage collector or a builder?
GARBAGE COLLECTOR
BUILDER
Navigate to Red Zone
Navigate to Green Zone
Perform obstacle avoidance
REFER TO OBJECT AVOIDANCE DIAGRAM IN SECTION 7.9 FOR MORE DETAILED ILLUSTRATION OF THIS PROCESS
REFER TO FORBIDDEN ZONE AVOIDANCE DIAGRAM IN SECTION 7.10 FOR MORE DETAILED ILLUSTRATION OF THIS PROCESS
Has the robot reached the angle threshold?
NO
YES

Figure 5.0.1 – Behavioural Flowchart

6.0 Sequence Diagram

This section of the Software Documentation includes the sequence diagram, which simply extends the behavioural flowchart by showing how objects operate with one another and in what order.

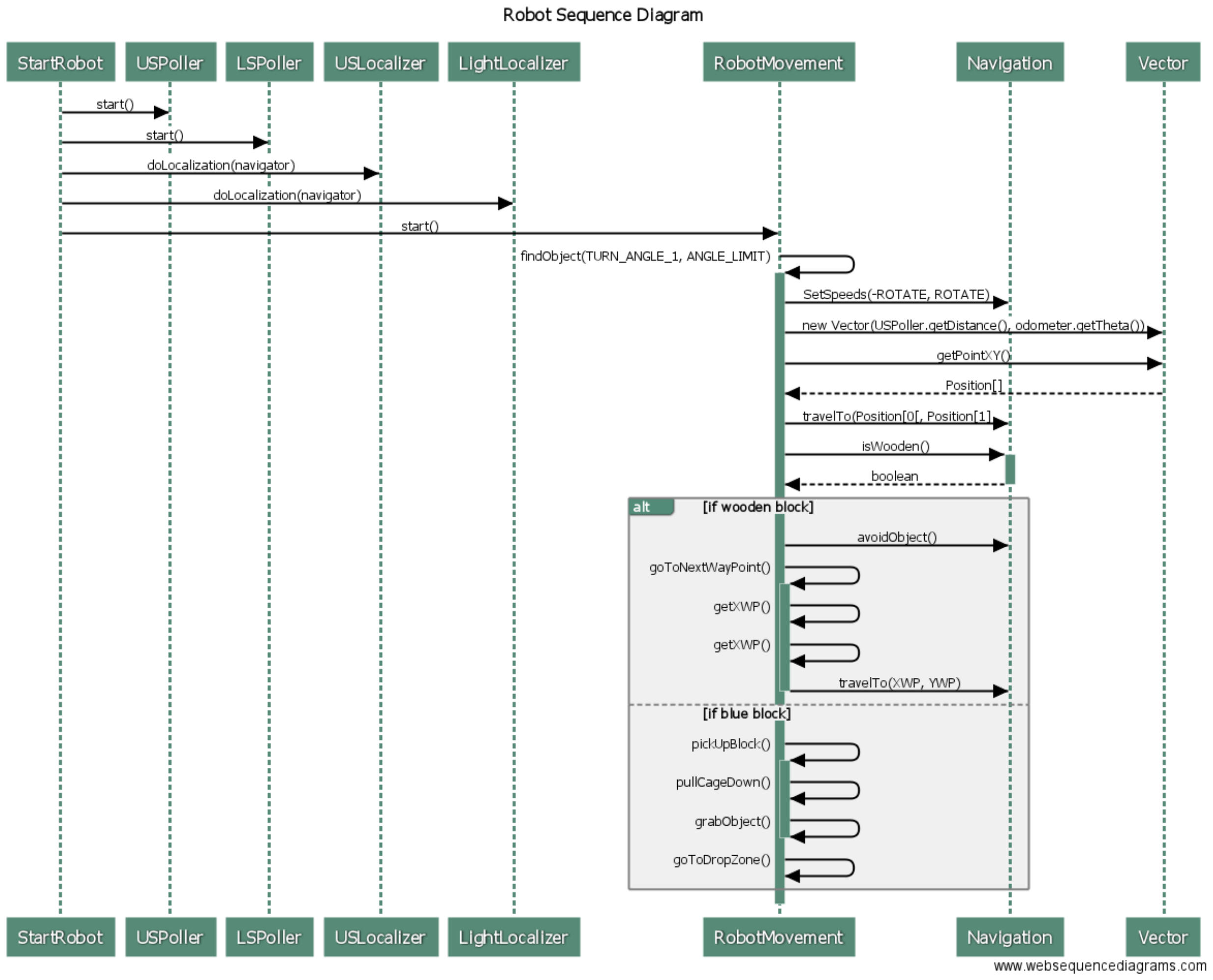


Figure 6.0.1 – Sequence Diagram of the Robot

7.0 Algorithms

This section of the Software Documentation gives an overview of the processes and functionalities which we implemented to solve the project's problems.

7.1 Localization Algorithm

Since the robot will be starting at one of the designated four corners of the arena (see Requirements Document section 2.2), and that the desired origin is assumed to be the point of intersection between the horizontal and vertical lines composing the starting corner tile, it is safe to assume that we can implement the localization techniques from lab four. For the ultrasonic localization, the robot is going to use the falling edge method, which means that it will record the first angle at the heading of the robot at the moment it first encounters the back wall (assuming the ultrasonic sensor is not facing the wall initially), and then the second angle as the robot's heading at the moment it encounters the left wall. Both of the angles are retrieved with the *getFilteredDistance()* method from the USPoller class to avoid any false negative readings in case the robot starts by facing the wall.

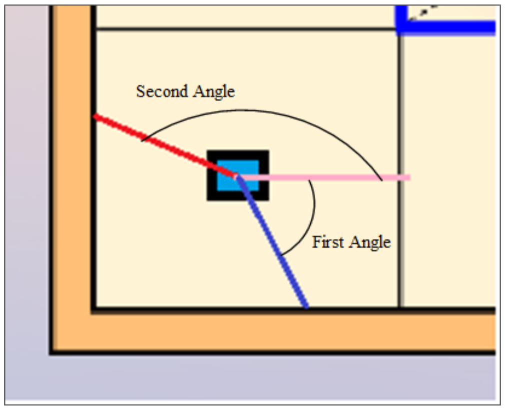


Figure 7.1.1 – Visualization of Angles Recorded by Ultrasonic Sensor

From these two angles, we can compute the angle to be added to the heading reported by the odometer in order to update the robot's current heading with the odometer's *setPosition()* method (see figure 7.1.1).

$$\text{Correction Angle} = \begin{cases} \textit{If First Angle Is Smaller Than Second Angle} \\ 45 - (\text{First Angle} + \text{Second Angle}) \div 2 \\ \textit{If First Angle Is Bigger Than Second Angle} \\ 225 - (\text{First Angle} + \text{Second Angle}) \div 2 \end{cases}$$

Figure 7.1.2 – Formula for Correcting Heading

Once the robot's current heading is set via ultrasonic localization, we can determine the robot's position through light localization. To perform light localization, the robot will proceed to move to the desired (0, 0) point, and from then it will start rotating until it reads four consecutive black lines. Each time a black line is detected, the robot records the heading at which the line is detected. These headings are used to calculate the respective x and y distance of the robot from the origin.

Normally, black lines are detected by comparing the data from the colour sensor to a threshold value obtained via testing. Each time the colour sensor values drop below that threshold the robot knows that a black line has been detected. However, due to the fact that, for the final project, we are required to localize within 30 seconds, we cannot simply keep the same slow speed of rotation which allowed us to retrieve accurate threshold readings from the colour sensor for the detection black lines. Therefore, we have decided to implement a line detection system which is based on the difference between two consecutive readings from the light sensor (i.e drop from the reading of the wooden part of the arena to the reading of the black line), instead of a detection based on the concrete infrared reading corresponding to the black colour. This will give us the opportunity to execute the light localization in a short amount of time while not compromising its overall accuracy. After four consecutive lines have been recognized, we can correctly reposition the robot at its new origin.

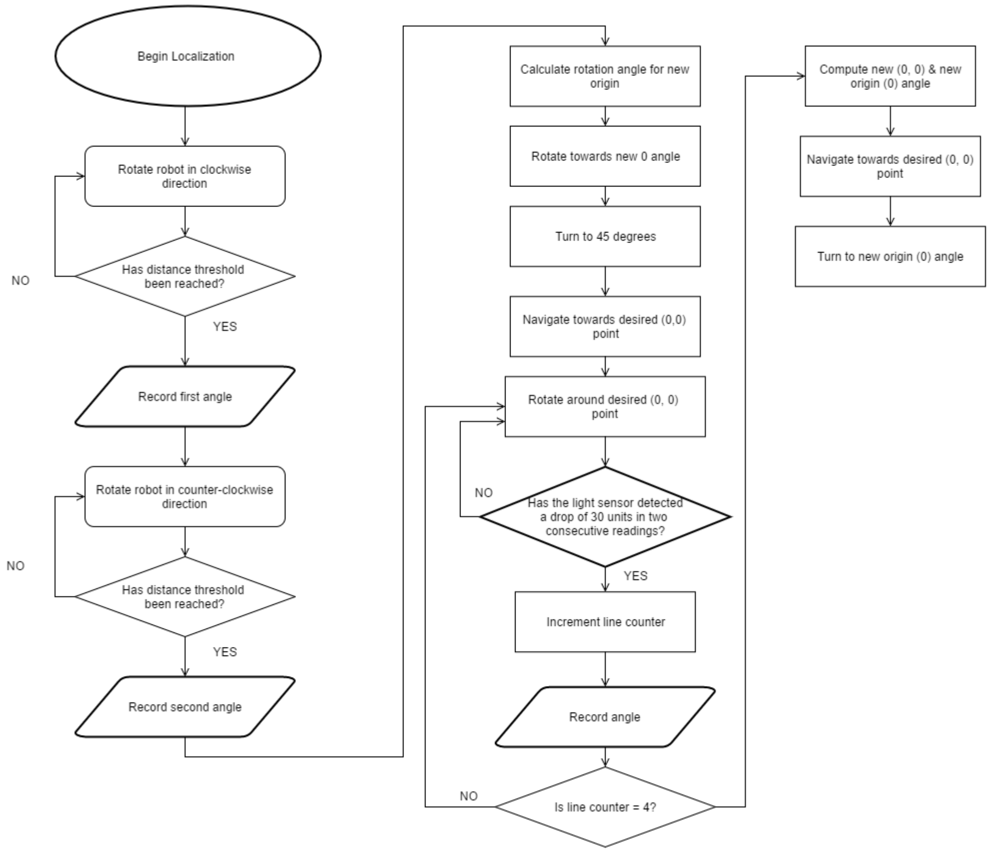

Begin Localization
Rotate robot in clockwise direction
Has distance threshold been reached?
NO
YES
Record first angle
Rotate robot in counter-clockwise direction
Has distance threshold been reached?
NO
YES
Record second angle
Calculate rotation angle for new origin
Rotate towards new 0 angle
Turn to 45 degrees
Navigate towards desired (0,0) point
Rotate around desired (0, 0) point
Has the light sensor detected a drop of 30 units in two consecutive readings?
NO
YES
Increment line counter
Record angle
Is line counter = 4?
NO
Compute new (0, 0) & new origin (0) angle
Navigate towards desired (0, 0) point
Turn to new origin (0) angle

Figure 7.1.3 – Flowchart for the Entire Localization Process

7.2 Odometry Algorithm

The odometry algorithm acts as an odometer for the robot such that it can track the position and heading of the robot as it traverses the field. For the odometer implemented by our team, the positive x-axis is set as the 0-degree heading and the positive rotational direction is assumed to be counter clockwise. Thus the 90-degree heading is the positive y-axis. In the implementation we are assuming that over a short time interval the angular velocity of the wheels is constant, that there is minimal slippage of the wheels, that the wheels are parallel to each other, that the radius of the wheels is constant, and that the distance from each wheel to the robot's center point is also constant. The radius of the wheels (*RADIUS*) and their respective distance to the robot *(TRAC*K) are constants that are measured and adjusted based on testing such that the odometer is as accurate as possible. While the odometer is running, the EV3 takes regular samples of the tachometers of each motor to calculate their respective changes in angle over time with the help of the *getTachoCount()* from Lejos' Motor API. These changes in tachometer values are then used along with the *RADIUS* and *TRACK* to calculate the distance the robot has travelled in the x-direction, the distance the robot has travelled in the y-direction, and the angle that the robot's change in heading subtends. The calculated values are then used to update the odometer's data with regards to the robot's current x-position, current y-position, and the heading that the front of the robot is facing.

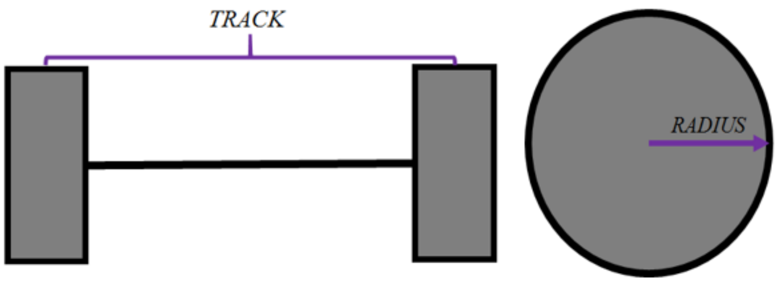


Figure 7.2.1 – Visualization of Track on and Radius Variables on 2D Wheel Representation

7.3 Odometry Correction Algorithm

The ideal odometry correction we intended to implement was composed of two light sensors mounted on the bottom of the robot and the correction would be performed measuring the time difference between the two rising edges of the different light sensors when encountering a grid line. Unfortunately, we did not have enough time to properly develop and test this technique, which forced us to implement a simplified algorithm for the correction.

The odometry correction that was finally implemented only corrects the odometer when the robot goes from one 4' by 4' board to another one. The rationale for this was that the odometer only loses precision when the robot goes through the gaps between boards. Once the robot finds itself in a new board, it performs light localization once again to calibrate the X and Y position of the odometer. This approach is less reliable and more time consuming than our initial method, but given the time constraints, it became the best option available.

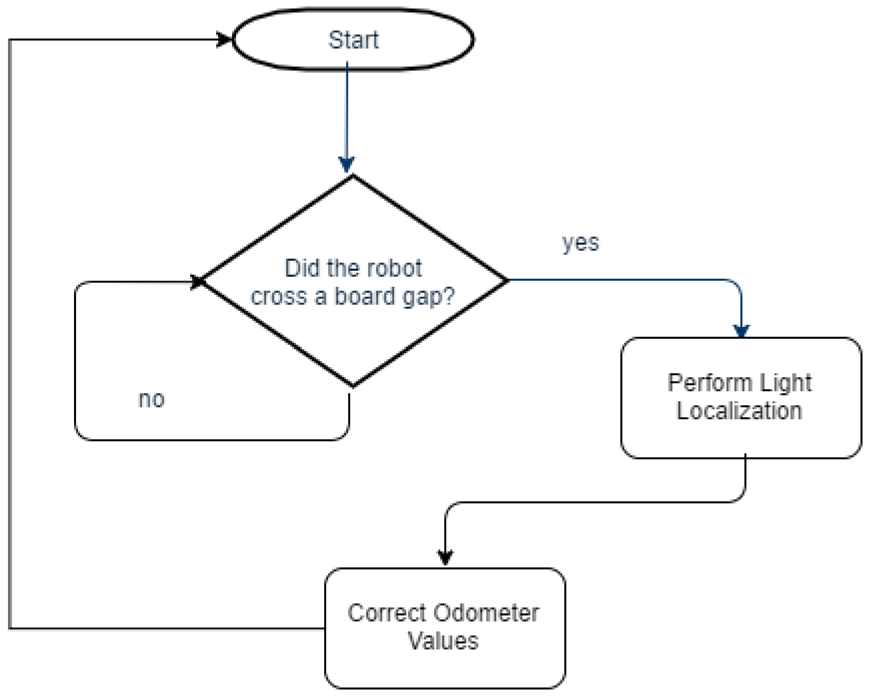


Figure 7.3.1 – Flowchart for Odometry Correction Algorithm

7.4 Turn to Angle Algorithm

To turn to a specific heading, the robot first calculates the difference between its current heading and the desired heading. The angle of the heading is calculated by using the method *getMinAng()* which takes the arctan of the desired Y coordinate minus the odometer's current Y coordinate over the desired X coordinate minus the odometer's current X coordinate. This method is once again contained in the Navigation class, and the arctan is calculated thanks to the *arctan2()* function provided by Java's Math API which wraps the angle between a -180 to 180 degree range. Once it calculates this difference, it determines the fastest way of reaching that heading by turning in its place. The wheels turn at the same speed in opposite directions thanks to the *setSpeeds()* method of the Navigation class until the heading reported by the odometer minus the calculated difference between heading is smaller than the constant *DEG_ERR*.

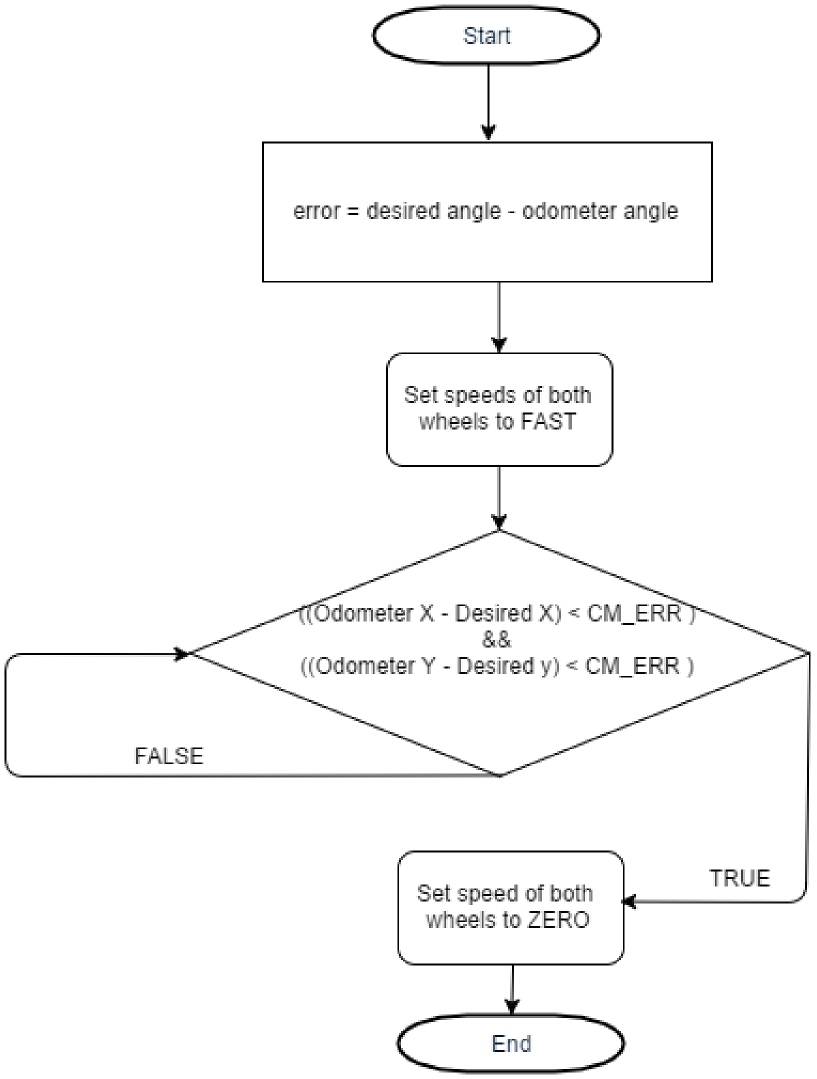

Start
error = desired angle - odometer angle
Set speeds of both wheels to FAST
((Odometer X - Desired X) < CM_ERR)
&&
((Odometer Y - Desired y) < CM_ERR)
FALSE
TRUE
Set speed of both wheels to ZERO
End

Figure 7.4.1 – Flowchart of Turn to Angle Algorithm

7.5 Travel to Location Algorithm

To go to a specific position on the XY plane, the robot first calculates and turns, thanks to the *turnTo()* method of the Navigation class, to the ideal heading to reach the desired position (see turn to angle algorithm). Then, the speeds of both wheels are set to *FAST* thanks to the *setSpeeds()* method contained within the Navigation class. The robot will only stop once both the odometer's X axis and the Y axis are within a small error (*CM_ERR*) from the desired XY position.

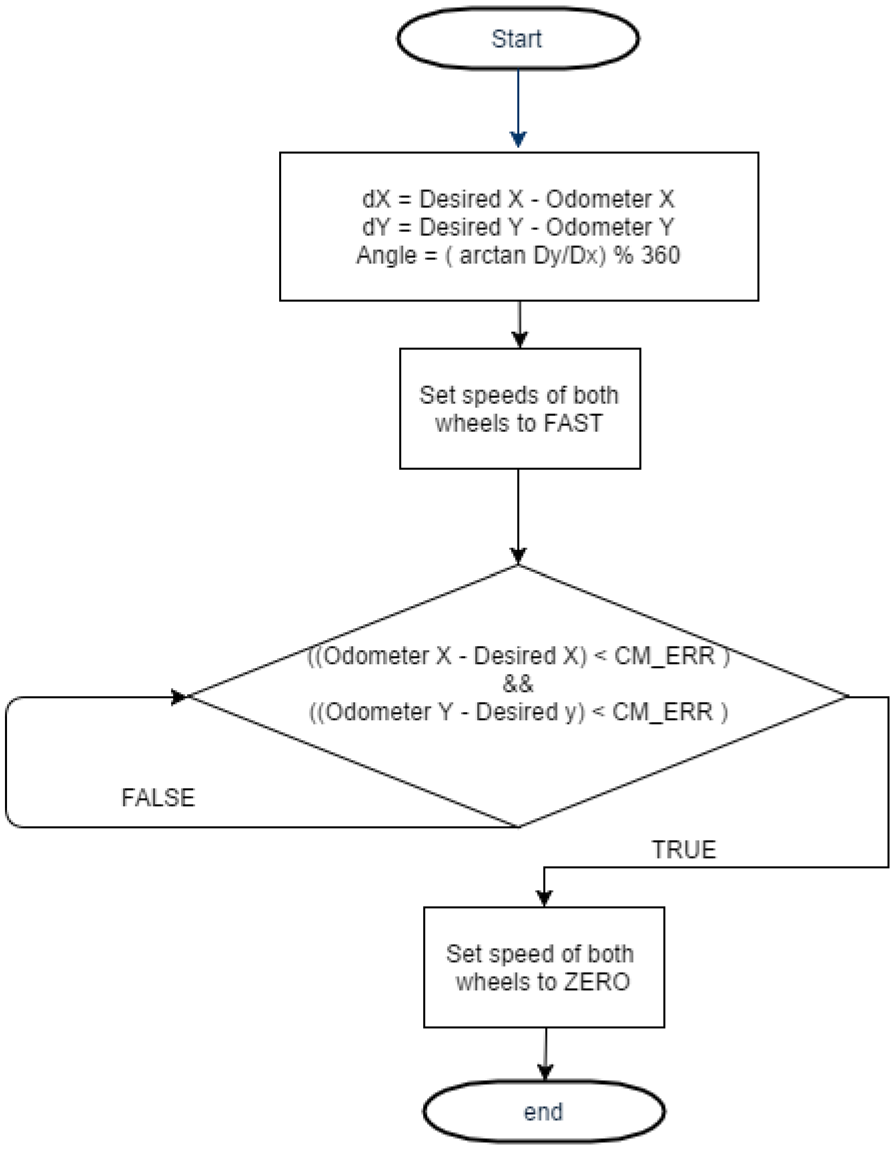

Start
dX = Desired X - Odometer X
dY = Desired Y - Odometer Y
Angle = (arctan Dy/Dx) % 360
Set speeds of both wheels to FAST
((Odometer X - Desired X) < CM_ERR)
&&
((Odometer Y - Desired y) < CM_ERR)
FALSE
TRUE
Set speed of both wheels to ZERO
end

Figure 7.5.1 – Flowchart of Travel to Location Algorithm

7.6 Object Grabbing Algorithm

Since we decided to go for an approach of building the tower on the go, i.e. as the robot picks up blocks, our object grabbing mechanism required some thought. After the robot has detected a Styrofoam block (see object detection algorithm), a counter variable named *blue_counter* (which is incremented every time the robot picks up a blue block) defined within the *run()* of the RobotMovement class is checked to see whether or not the block encountered is the first one in the stack or not. If *blue_counter* is equal to zero, the cage is simply brought down, through a rotation of the pulley's motor with the help of the *rotate()* method defined in the Motor API to the Styrofoam block's level, and then the claw is opened (again, by using the *rotate()* method on the motor that is responsible for the control of the claw) to trap the block. Afterwards, the pulley's motor brings the cage up to its normal level. If, however, upon detection of a block, the robot had already previously detected Styrofoam blocks (meaning that the value of *blue_counter* is not equal to zero), the pulley's motor rotates by a predetermined angle which levels the robot's cage with the newfound block, after which the robot drops the currently trapped block[s] on top of it.

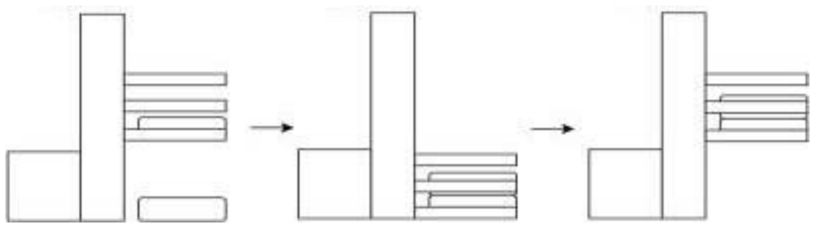


Figure 7.6.1 – Illustration of Object Grabbing Algorithm

Then, the cage is brought down down to the ground, and from thereupon the claw's motor is rotated so as to grab the newly built stack of blocks, after which the cage is raised back to the tower of the robot's tower structure.

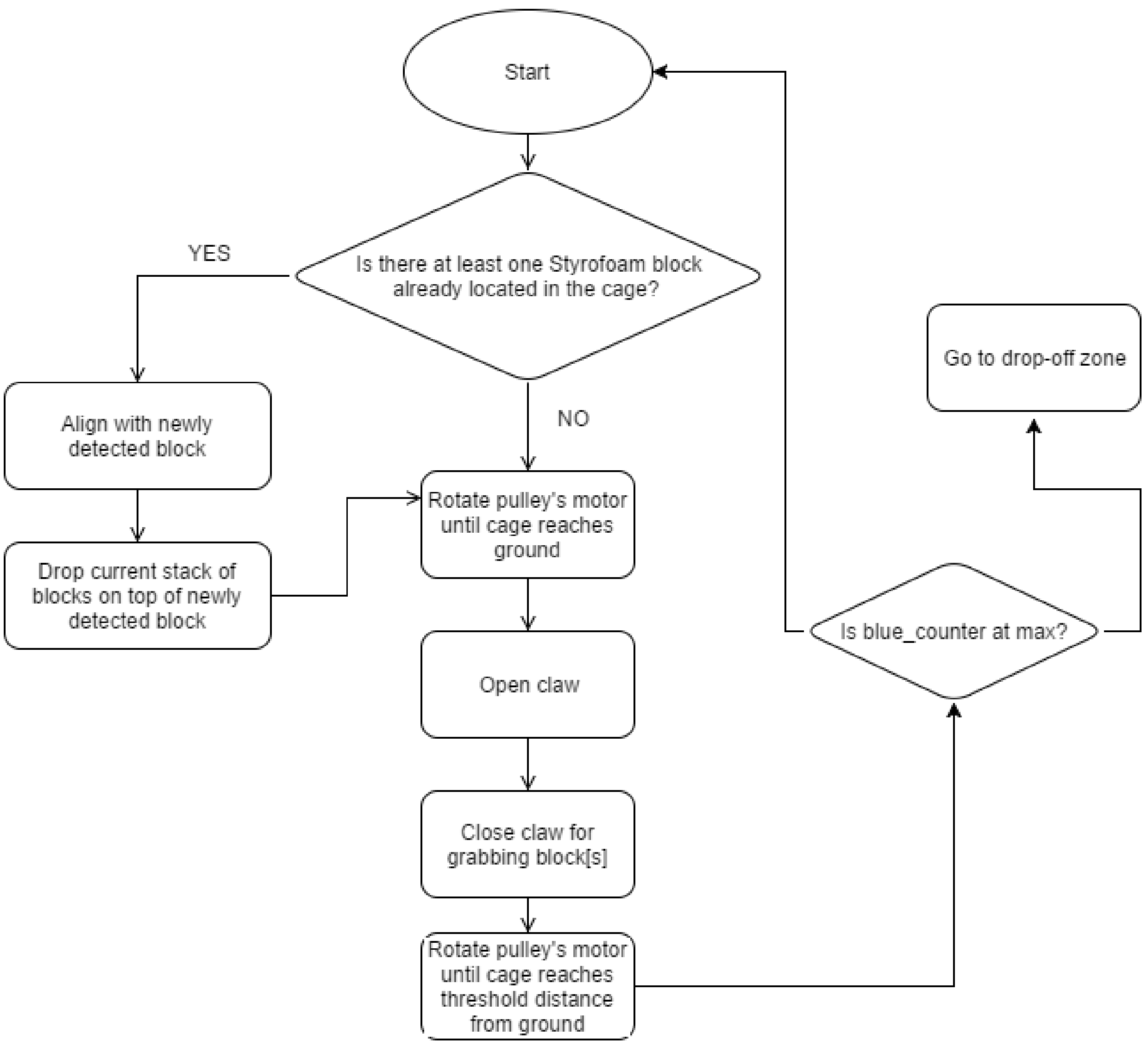

Start
Is there at least one Styrofoam block already located in the cage?
YES
NO
Align with newly detected block
Drop current stack of blocks on top of newly detected block
Rotate pulley's motor until cage reaches ground
Open claw
Close claw for grabbing block[s]
Rotate pulley's motor until cage reaches threshold distance from ground
Is blue_counter at max?
Go to drop-off zone

Figure 7.6.2 – Flowchart of Object Grabbing Algorithm

7.7 Object Detection Algorithm

The object detection works with the help of two ultrasonic sensors which are aligned in parallel, one of top of the other. One of the sensors is almost at ground-level, and the other sensor is above it, separated by a distance which is just slightly over the height of the Styrofoam block. By placing the two ultrasonic sensors in such manner, we can efficiently perform object detection with the following logic: if the lower ultrasonic sensor's reading is below a given threshold distance, but the upper ultrasonic sensor's reading isn't, we know that the robot has encountered a Styrofoam block. However, if both ultrasonic sensor's readings are below the same threshold, we can safely assume that the robot is facing a wooden block.

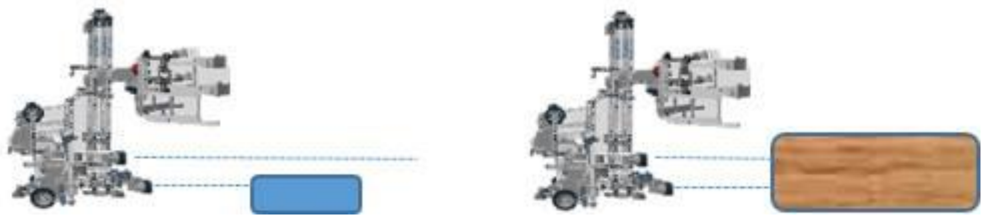
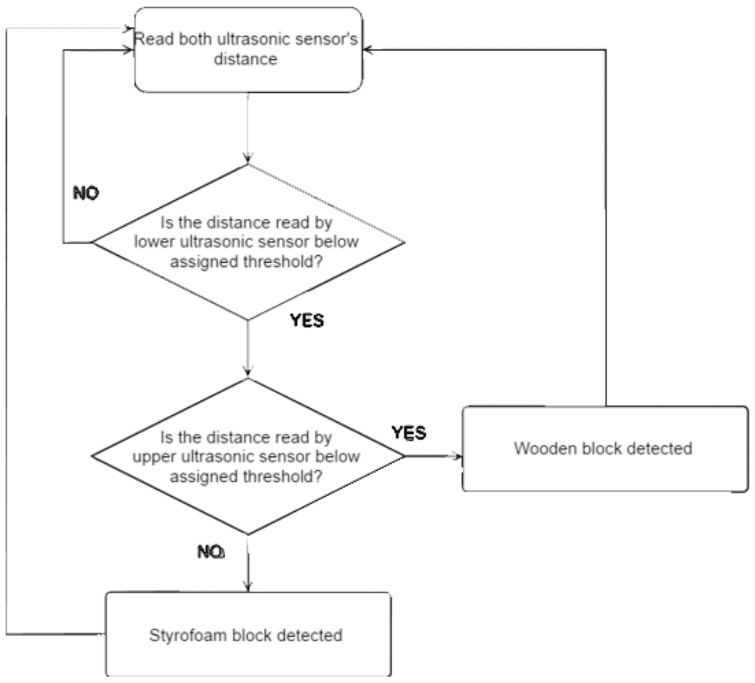


Figure 7.7.1 – Illustration of Object Detection Algorithm

This method allows us to distinguish objects fastly and accurately, without the need of a colour sensor, which implies that we do not have to deal with the pain of optimally approaching and aligning the robot with a detected object to retrieve precise RGB readings. Both of the readings from the ultrasonic sensors are retrieved by calling the method *getDistance()*, which is located in the USPoller class.


Read both ultrasonic sensor's distance
NO
Is the distance read by lower ultrasonic sensor below assigned threshold?
YES
Is the distance read by upper ultrasonic sensor below assigned threshold?
YES
Wooden block detected
NO
Styrofoam block detected
Figure 7.7.2 – Flowchart of Object Detection Algorithm

7.8 Search & Path Algorithm

The search algorithm essentially scans the grid board and maps it into an ArrayList. When the localization is complete robot starts to rotate at a 75 degree angle (relative to the 90 degree angle of the Y-axis, see odometry algorithm for definition of robot's axis) and continues until it reaches the *LIMIT_ANGLE*. During the rotation, the ultrasonic sensor repeatedly returns the distance values it reads (thanks to the *getDistance()* method of the USPoller class) and the odometer returns the robot's angle at the time of the reading. Both of these values are used to create, every given interval of time, a Vector object which gets stored in an ArrayList of type Vector. The delay between each consecutive addition of an element to the array is handled by the *Thread.sleep()* function provided in the Thread API. Since the search algorithm is contained within the *run()* method of the RobotMovement class (which extends Thread), we simply call the *Thread.sleep()* function to ensure the filtering of most garbage values provided by the ultrasonic sensor, which increases the accuracy at which the robot approaches the object at the end of the algorithm's execution. When the *LIMIT_ANGLE* of rotation is reached, the array is searched for the vector object with the smallest distance value thanks to the *sort()* function offered by the ArrayList API.

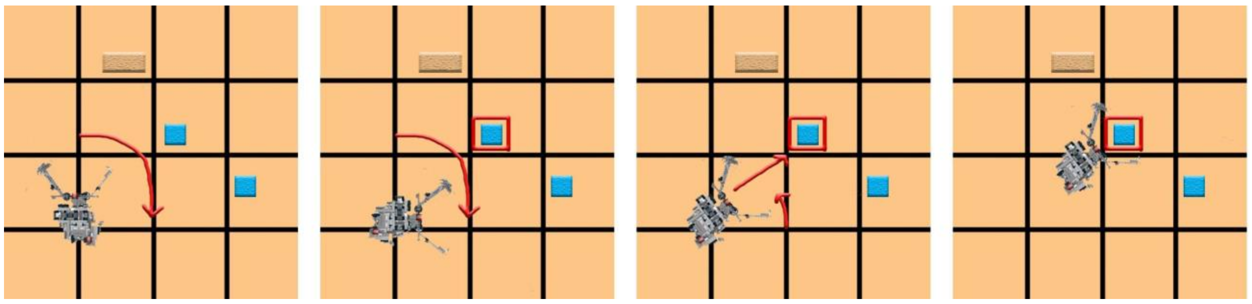


Figure 7.8.1 – Illustration of Search Algorithm

If the closest object falls within a given threshold distance (implemented to account for the limitations of the ultrasonic sensor's accuracy), the robot rotates to the angle corresponding to the smallest distance value and navigates forward by that distance. Else, if a robot finds the an object but it is above the given threshold distance, the robot will travel to a point that is approximately within a threshold distance from the detected object, and from that point it will begin scanning again to approach said object with more accuracy afterwards. If a scan does not detect anything, the robot travels to its next waypoint, which is simply located along the robot's path to its designated drop-off zone (see Requirements Document section 2.2).

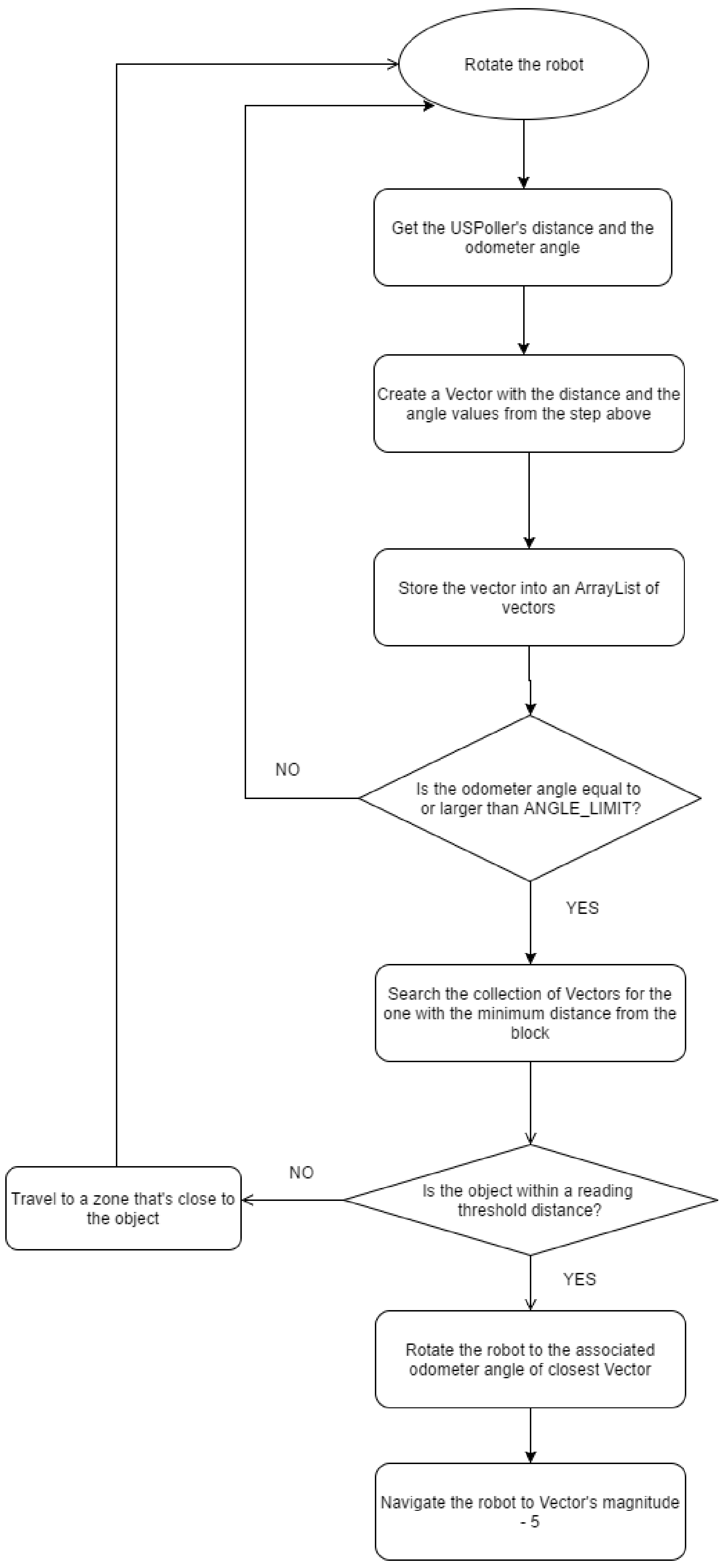

Rotate the robot
Get the USPoller's distance and the odometer angle
Create a Vector with the distance and the angle values from the step above
Store the vector into an ArrayList of vectors
Is the odometer angle equal to or larger than ANGLE_LIMIT?
NO
YES
Search the collection of Vectors for the one with the minimum distance from the block
Is the object within a reading threshold distance?
NO
Travel to a zone that's close to the object
YES
Rotate the robot to the associated odometer angle of closest Vector
Navigate the robot to Vector's magnitude - 5

Figure 7.8.2 – Flowchart for Search and Planning Algorithm

7.9 Object Avoidance Algorithm

Object avoidance algorithm starts when a wooden block is detected (see object detection algorithm). The ultrasonic sensor first reads the distance between the robot and the wooden block and assigns the value to a variable, *distanceFromSensorToBlock*. Once the robot is in front of the obstacle to avoid, it starts scanning the object until it sees the rightmost edge of the wooden block. After the *getDistance()* of the USPoller class returns a value that is considerably bigger than the distance from the block recorded previously, we know that we have encountered an edge, and therefore we add a given constant angle to the current angle read by the odometer so as to ensure that, no matter at which orientation the wooden block is placed, we always manage to move away from it.

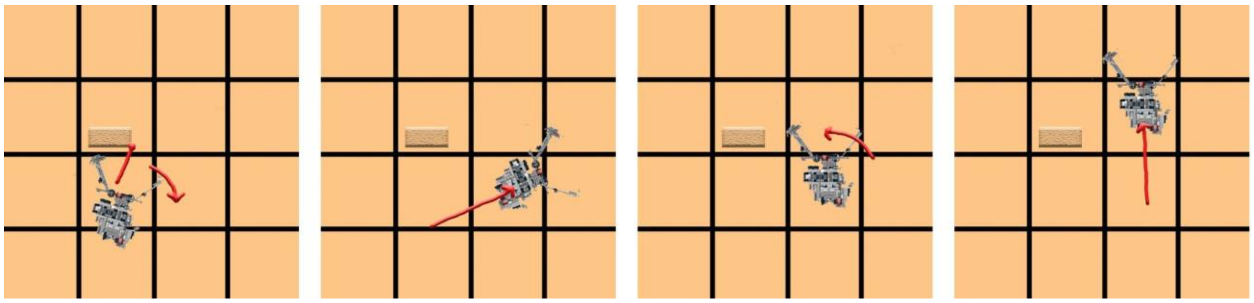


Figure 7.9.1 – Illustration of the Object Avoidance Algorithm

After turning by said angle, the robot moves forward by a constant distance parallel to the block, rotates back by an angle of -90 degrees and then checks whether or not the object has been avoided by comparing the current reading of the ultrasonic sensor with the recorded *distanceFromSensorToBlock*, from which the robot decides whether or not it should avoid again or simply move forward by another constant distance to get out of the way of the wooden block.

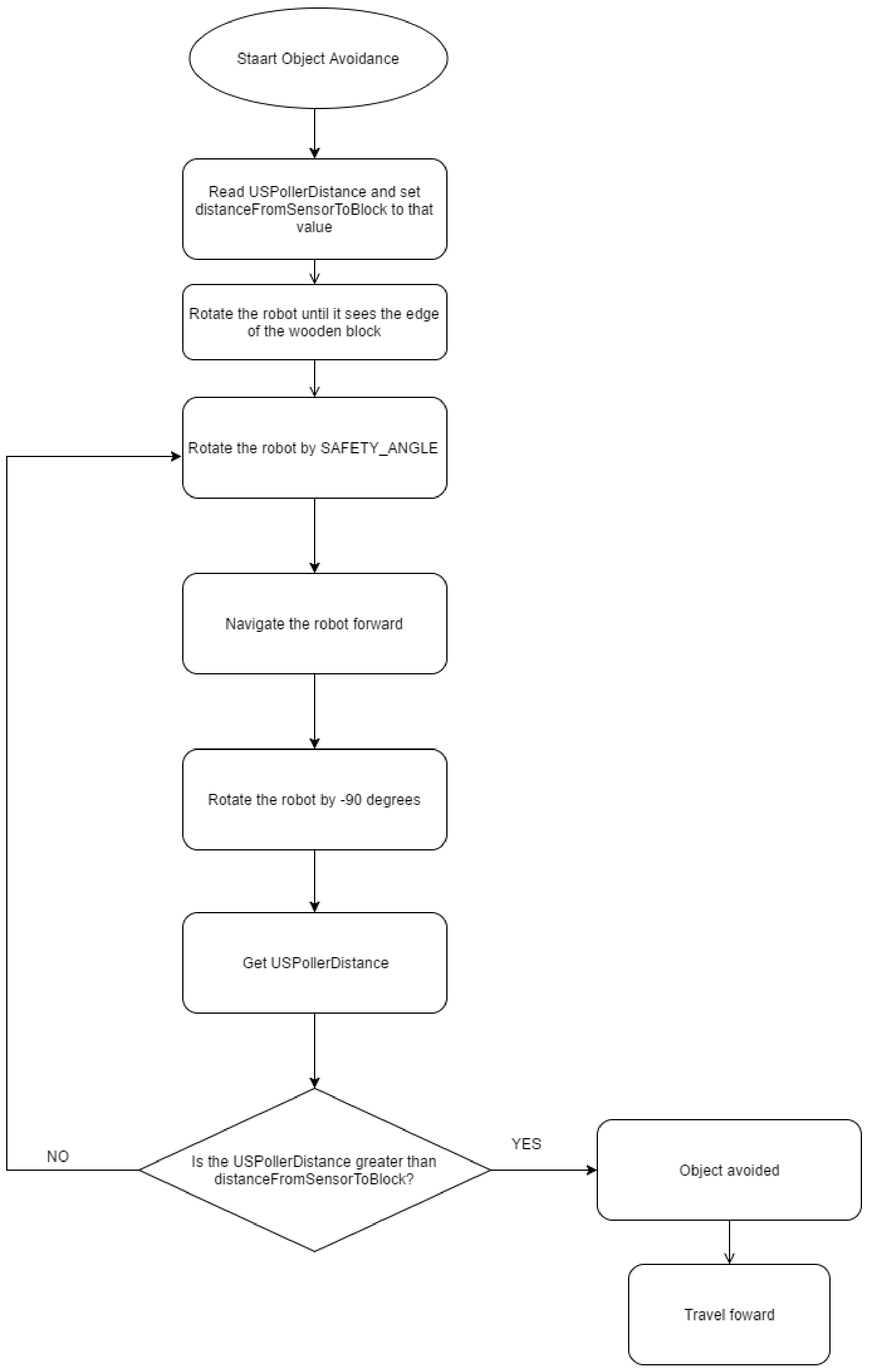
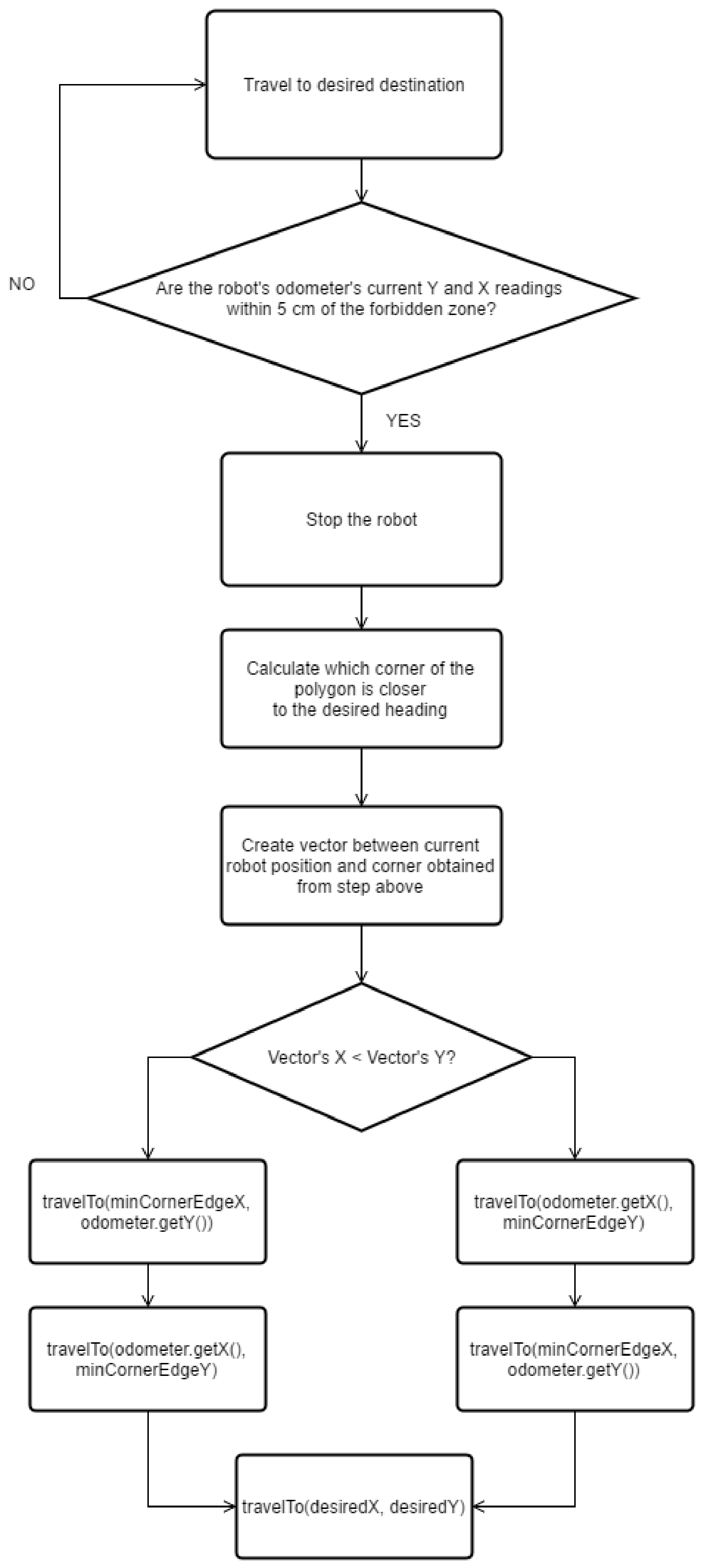

Staart Object Avoidance
Read USPollerDistance and set distanceFromSensorToBlock to that value
Rotate the robot until it sees the edge of the wooden block
Rotate the robot by SAFETY_ANGLE
Navigate the robot forward
Rotate the robot by -90 degrees
Get USPollerDistance
Is the USPollerDistance greater than distanceFromSensorToBlock?
NO
YES
Object avoided
Travel foward

Figure 7.9.2 – Flowchart of Object Avoidance Algorithm

7.10 Forbidden Zone Avoidance Algorithm

As specified per the Requirements Document's section 2.3, the robot cannot enter the opponent's zone. Since both of the drop off zone are either triangular or rectangular, we have implemented an algorithm which navigates around the edges of the polygon. In our *travelTo()* method located in the Navigation class, we constantly check whether the robot is near the forbidden zone. If the reported current position of the odometer sees that the robot is at a distance of 5 cm from the forbidden zone, it immediately stops the robot and snaps into the forbidden zone avoidance algorithm. A calculation is then performed to check which of the four edges is closest to the desired destination the robot was heading towards. Once we have found which of the polygon's corners is closest, we create a new Vector object which specifies the minimal angle and the distance between our robot and the closest edge to the desired destination. From these two parameters in polar coordinates, we can move on to finding the Cartesian coordinates of the Vector object, and once its X and Y values are calculated, we compare to see which one of the two is bigger. If X is larger than Y, the robot first moves to a point composed of the X parameter of the closest edge and the current Y reading of the odometer, then it moves towards a point composed of the odometer's current X reading and the Y parameter of the closest edge, and, finally, it travels to the originally desired point from thereupon. However, if X is smaller than Y, the robot first moves to a point composed of the Y parameter of the closest edge and the current X reading of the odometer, then it moves towards a point composed of the odometer's current Y reading and the X parameter of the closest edge, and, finally, it travels to the originally desired point from thereupon. By travelling in this specific way, we ensure that the robot perfectly contours the polygon representing the forbidden zone, and we also guarantee that it avoids the zone within a minimal path towards the desired heading.


Travel to desired destination
NO
Are the robot's odometer's current Y and X readings within 5 cm of the forbidden zone?
YES
Stop the robot
Calculate which corner of the polygon is closer to the desired heading
Create vector between current robot position and corner obtained from step above
Vector's X < Vector's Y?
travelTo(minCornerEdgeX, odometer.getY())
travelTo(odometer.getX(), minCornerEdgeY)
travelTo(odometer.getX(), minCornerEdgeY)
travelTo(minCornerEdgeX, odometer.getY())
travelTo(desiredX, desiredY)

7.10.1 – Flowchart of Forbidden Zone Avoidance Algorithm

7.11 Median Filter Algorithm

The ultrasonic sensor takes in data from a thirty degree window and records the minimum of those distances as the reported distance. This introduces a considerable amount of noise readings in the ultrasonic sensor data. Thus we implemented a median filter for the ultrasonic sensor's data to reduce the amount of noise and minimize the probability of a false negative. A median filter goes through the input data entry by entry and replaces each entry with the median of the neighbouring entries. The number of neighbours from which this median is taken is called the "window." This window slides over each entry and replaces the entries with the median of the window. In the implementation of our median filter *getFilteredDistance()* we passed the ultrasonic data to the *getMedian()* method in our USPoller class. In *getMedian()* the data is passed to an ArrayList using the ArrayList method *toArray()*. Next, *getMedian()* utilizes *sort()* to order the data entries in the window in ascending order and then returns the entry in the middle of the array. As a result after the window passes over an array of data received from the ultrasonic sensor, the extreme data points will have been filtered out and the median data entry remains. Once *getMedian()* returns the median value to *getFilteredDistance()*, the median is compared to the *distance* obtained from the ultrasonic sensor. If *distance* is less than the median, *getFilteredDistance()* returns the median as the reported distance. Else, *getFilteredDistance()* returns *distance* as the reported value.

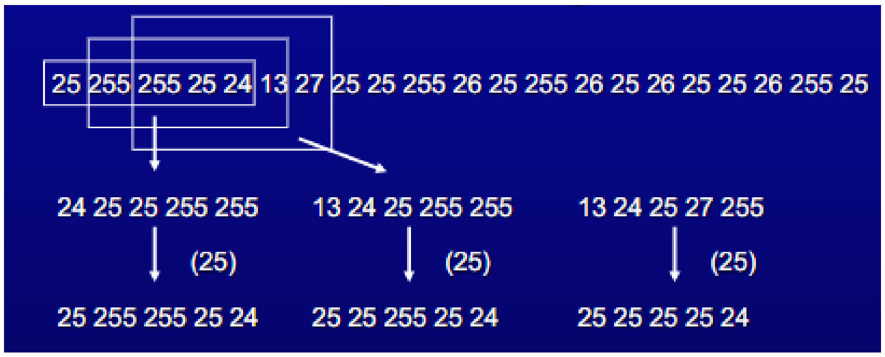


Figure 7.11.1 – Visualization of median filter from ECSE 211 Fall 2016 Sensors-DAQ Lecture Part 1

8.0 References

All theory and resources taken from ECSE 211 Fall 2016 lecture slides posted online on myCourses, authored by Professor David Alister Lowther and Professor Dennis Giannacopoulos