Documentation

ECSE 211 – Design Principals and Methods

Group 03

Garret Holt, Sung Hong, Nawras Rabbani,
Matthew Rodin, Tiffany Wang & Troy Yang

April 15th, 2016

TABLE OF CONTENTS

Requirements Document

Team 03
Project: Neymar
Task: Soccer - Playing Robot

Document version: v.5.0
Date: Tuesday March 22nd, 2016
Author: Matthew Rodin

Edit History: Matthew Rodin fixed the formatting of the document. He also added, clarified and deleted information from sections 2.1-3.2 in order to make sure all the specifications are up to date. Finally, he added a "glossary of terms" section

The purpose of this document is to ensure that the requirements of the client are well understood.

1.0 - TABLE OF CONTENTS

2.0 – CAPABILITIES

2.1 – PURPOSE

The purpose is to construct an autonomous, one-on-one soccer-playing robot capable of operating within a 12'x12' enclosure. It is expected that the robot will be able to navigate on the playing board autonomously while avoiding obstacles. As well, the product must be able to play offense (collect correct colored balls and project them into a net) and defense (stop randomly projected balls directed towards the robot).

2.2 – SCOPE

- **Range of capabilities**

Each team member has knowledge or expertise in specific areas, which should be taken into account for the distribution of tasks (see "Capabilities Document").

- **Resources**

- Six team members
- Three DPM kits
    a) 3 EV3 bricks (12 motor ports and sensor ports)
    b) 12 motors
    c) 3 ultrasonic sensors
    d) 3 light sensors
    e) 6 touch sensors
- DPM Lab (Trottier 0110) equipped with electrical outlets, computers (loaded with appropriate software and sample game board (for calibration and testing).

- **Project Limitations**

There is only 6 weeks between the release of the project description and the competition day. Therefore the budget is the biggest limitation. No member should exceed 9 hours per week (total of 54 hours for the project). Every team member has extra-curricular activities: manage the meeting and the lab times accordingly.

- **Competition limitations**

-The competition will consist of 2 rounds. In each round, each player will get the opportunity to play both as forward and defender (for a total of 4 runs)
- The floor is comprised of nine 4'x4' hardwood-covered metal panels that lock together.
- Obstacles will consist of wooden blocks (found in the lab) and can be placed anywhere.
- Once both players arrive at their starting positions, only the forward will be permitted to use its ultrasonic sensor.
- Localization must be completed in 30 seconds or less.
- The robot must be able to receive coordinates via WiFi
- Game play proceeds until time (5 minutes) runs out.

2.3 – CONSTRAINTS

- 3 Lego Mindstorm Kits are allowed to construct the design.
- The number of sensor and motor ports on one EV3 brick is limited (maximum of 4 sensors and 4 motors).
- Java using LeJOS on Eclipse is the software that will be used to program the robot.
- The robot has to be sufficiently small to fit within a 30cm by 30cm tile (for the purpose of localization)
- The robot has to function fast enough in order to accomplish all the tasks within the time limit (4 rounds of 5 minutes each)
- The battery has to be charged enough to be able to run the robot for at least 20 minutes → The weight of the robot will need to be kept to a minimum.

2.4 – USER FUNCTIONS

- The robot will receive a set of instructions via WiFi before beginning its operations (see Section 2.2)
- The user can control the robot before the operation (i.e. place it on the board and tell the robot whether he plays as a defender or an attacker).
- In order to begin the robot's operation, the center button will be pressed. In other words, the user can interact with the device before it operates by pushing a button (interface) on the robot that will make it run (complete the pre-determined set of tasks in the code). Therefore, this is set up in a "batch" mode since the set of tasks run to completion without human intervention.

2.5 – OPERATING ENVIRONMENT

- The robot is going to navigate on a 12' by 12' board (comprised of 9 4' by 4' wooden tiles. There will be black lines intersecting each of these nine tiles.
- On the board, there is to be an opponent robot as well as randomly placed obstacles.
- The competition will be held on the 2nd floor of the Trottier Building. It is to be noted that this floor has considerably more light exposure (due to numerous windows) than the DPM testing lab. We expect the intensity of the sunlight to play a big role in the design of the robot (i.e. the light sensors and ultrasonic sensor have to be adjusted accordingly).
- There will be many people assisting in the competition itself. Therefore, we must calibrate the ultrasonic sensor to filter out unwanted noise.

2.6 – PERFORMANCE

Please refer to sections 2.2 ("Scope") and 2.3 ("Constraints").

3.0 – COMPATIBILITY

3.1 – COMPONENT RE-USE

Existing robot designs and code written for the five labs may be used. For example:

- *Lab 2: Odometer* - correct the readings of the robot on its current position, this will allow it to navigate more accurately
    - Group 22's lab 2 used for the project
- *Lab3: Navigation* - avoid the obstacles during the course. (Note: Lab1 - WallFollower is implemented in this lab)
    - Group 22's lab 3 used for the project

Labs that are not reused:

- *Lab4: Localisation* - the robot can localize itself to its initial position prior to the beginning of the game.
    - Sung Hong will be writing the code from scratch.
- *Lab5: Ballistics* - launch the pallet to the goal.
    - Software team will write the code according to the final design of the robot.

As well, code and algorithms may also be pulled from the Internet, providing credit is properly given.

3.2 – COMPATIBILITY WITH THIRD PARTY PRODUCT

- Seeing that the robot will receive a set of instructions via WiFi before beginning its operations (see Section 2.2), the product must be able to use its built-in WiFi feature.
- There is no need for our product to interact with non-Lego components.
- During preliminary meeting, Professor Ferrie strongly suggested the implemented code to be in Java with imported LeJOS package. Thus far, the team has decided to implement a single EV3 brick as to limit the complication and increase in error cause by the cross communication of multiple bricks.

4.0 – GLOSSARY OF TERMS

- **LeJOS:** a firmware replacement for Lego Mindstorms programmable bricks.
- **Java Virtual Machine:** allows the robots to be programmed in Java
- **Odometry:** the use of data from motion sensors to estimate change in position over time
- **Navigation:** process of monitoring and controlling the movement of the robot from one place to another.
- **Localization:** the construction of a map/floor plan in order to localize oneself.

- **Ballistics:** mechanical science that deals with launching, flight, behavior, and effects of projectiles
- **Ultrasonic sensor:** use sound waves rather than light, making them ideal for stable detection of uneven surfaces, liquids, clear objects, and objects in dirty environments.
- **Light sensor:** measures how much light is shining on it. It has two modes: “light” and “dark.”
- **WiFi:** technology that allows electronic devices to connect to a wireless LAN (WLAN) network
- **EV3 Brick:** EV3 programmable brick. *EV* refers to the "evolution" of the Mindstorms product line and *3* refers to the fact that it is the third generation of computer modules.

Top

Capabilities Document

Team 03
Project: Neymar
Task: Soccer - Playing Robot

Document version: v.4.0
Date: Tuesday March 22nd, 2016
Author: Matthew Rodin

Edit History: Matthew Rodin fixed the formatting of the document and added a "glossary of terms" section.

The purpose of this document is to identify the skill base and resources available for solving the design problem.

1.0 - TABLE OF CONTENTS

2.0 - TEAM MEMBERS

Computer Engineering:

- Matthew Rodin (Documentation)
(514) 662- 2179
matthew.rodin@mail.mcgill.ca

Electrical Engineering:

- Garret Holt (Development)
(514) 577-9810
garret.holt@mail.mcgill.ca

- Sung Hong (Testing)
(514) 992-3280
sung.hong2@mail.mcgill.ca

- Tiffany Wang (Project Manager)
(514) 993-7710
tiffany.wang@mail.mcgill.ca

- Troy Yang (Mechanical)
438-880-4321
chenxi.yang@mail.mcgill.ca

Software Engineering:

- Nawras Rabbani (Development)
514 572 3857
nawras.rabbani@mail.mcgill.ca

3.0 - CAPABILITIES

	Software Ability (on 10)	Mechanical Ability (on 10)	Experience in Documentation	Experience in Managing
Garret Holt	9	8	Yes	Yes
Sung Hong	7	9	Yes	No
Nawras Rabbani	10	5	Yes	No
Matthew Rodin	8	9	Yes documentation experience in internship	Yes
Tiffany Wang	7	9	No	Yes. Mcgill Taiwanese Student Association
Troy Yang	6	10	Yes documentation experience through doing lab reports from DPM Lab 1-5	Yes McGill ACCE team facilitator in CLE&E

4.0 – POSSIBLE APPLICATION AREAS (RELEVANT EXPERIENCE)

- *Nawras* has extensive experience programming in Java. In addition, he teaches EV3 programming in elementary schools, making him familiar with the Lego hardware/software.
- *Matthew* had extensive documentation experience during his last summer internship.
- *Garrett* has experience developing (Java) due to experience gained at prior internships.
- *Tiffany* has experience managing and organizing large events within McGill.
- *Troy* was in charge of preparing the mechanical design of his robot for the first 5 DPM labs.
- *Sung* was in charge of testing/troubleshooting his robot for the first 5 DPM labs.

5.0 – AVAILABILITY

- *Garret*
  - EPTS tutor
  - Member of MFE Formula Electric Racing Team

- *Matthew*
  - Member of Engineering Investment Group and EUS Sports
  - Not available Monday and Tuesday from 5:30 pm to 8:00 pm

- *Tiffany*
  - Member of McGill Taiwanese Student Association
  - Member of McGill Women in Computer Science
  - Not available Saturday from 10:30am to 1:30 pm, Tuesday 1:00pm to 2:30 pm and Friday 3:00pm to 5:00 pm
  - Not available March 19-20th (RoboHacks)

- *Troy*
  - Plays Basketball (No fixed schedule for practice)

- *Sung*
  - Does Boxing (5-6 practices per week, unfixed time)

- *Nawras*
  - Does kickboxing (2 practices per week, fixed time)
  - Part-time job
  - Not available Tuesday evening, Thursday evening, Friday afternoon

6.0 – GLOSSARY OF TERMS

- **Java:** a general-purpose computer programming language that is class-based and object-oriented, which is specifically designed to have as few implementation dependencies as possible.

Top

Constraints Document

Team 03
Project: Neymar
Task: Soccer - Playing Robot

Document version: v.4.1
Date: Saturday April 2nd, 2016
Author: Matthew Rodin

Edit History: Matthew Rodin added a battery note (offense vs. defense) to the "Hardware Constraints" section.

This purpose of this document is to list all the constraints on the design solution.

1.0 - TABLE OF CONTENTS

2.0 – ENVIRONMENTAL CONSTRAINTS

- The dimensions of the playing field are pre-determined and described in section 2.2 "Scope" of the "Requirements Document"
- The overall dimensions of the robot cannot exceed the size of a single tile square (30.48 cm x 30.48 cm). However there is no constraint on the maximum height of the robot
- A time constraint of 5 minutes will be placed on the entire runtime of the task(s).

3.0 – HARDWARE CONSTRAINTS

- Please see section 2.2 "Scope" of the "Requirements Document" for all the hardware used.
- Additional hardware items not included in the Lego kit may be used. We used:
    - Various elastic bands in order to improve stability of multiple robot components
- There is also a limit to the power level of the robot (rechargeable battery), which will affect its performance.
    - If our robot is summoned to play defense first, we must attempt to minimize its movements in order to conserve battery for the offense portion of the demo.
- The sensors and motors must be connected to the EV3 brick using registered jack (RJ) connector wires, which place many motion constraints on our robot.

4.0 – SOFTWARE CONSTRAINTS

- The software component is limited to the leJOS EV3 0.9.1 and its Java based firmware API. Note: this does not include the entire Java™ Platform, Standard Edition 7 API Specification. This is important because the queue interface – a collection with specific operations – added to the Java™ Platform in version 1.5, cannot be applied to the LinkedList or ArrayList classes.

5.0 – AVAILABILITY OF RESOURCES

- Meeting Times
    - Professor Meeting: Mondays 11:50 AM - 12:10 PM (Trottier 4104)
    - TA Meeting: Fridays 11:30 AM - 12:00 PM (DPM Lab)
    - Team Meeting: Sundays 5:30 PM - 6:00 PM (DPM Lab) → In order to review/make final changes to presentation.

- Note: Software and hardware teams meet at non-fixed times during the week in order to complete assigned tasks and progress through the project. The team is in constant remote contact using the Slack application as to make sure every member is up to date with the current state of the project.
- Please refer to the "Capabilities Document" for more information.

6.0 – BUDGET

6.1 – TIME BUDGET

- Each member will put in no more than 9 hours of work per week (6 total weeks) until the final day of the competition on April 14th. No additional budget will be used in terms of money.
- A total of 324 hours cumulatively.

	Week 1	**Week 2**	**Week 3**	**Week 4**	**Week 5**	**Week 6**	**Total per Person**	**Average per Week (per Person)**
Holt, Garret	7.5	8	7	9	10	10	51.5	8.58
Hong, Sung	6	9	6	8	9	9.5	47.5	7.92
Rabanni, Nawras	6.5	9	8	7	8	9	47.5	7.92
Rodin, Matthew	6.5	6	8.5	9	10	10	50	8.33
Wang, Tiffany	7.5	8	10	9	11	12	57.5	9.58
Yang, Troy	5.5	8	7	8	8	9	45.5	7.58
Total per week	39.5	48	46.5	50	56	59.5	299.5	8.32

6.2 – MONEY BUDGET

- A budget of 40$ CAD is expected to be spent over the entire course of the project.
- This budget will be designed for miscellaneous extra resources (i.e. spoons, paper, etc.) as well as for one (1) poster and two (2) USB keys.

7.0 – GLOSSARY OF TERMS

- **Registered Jack (RJ):** a standardized telecommunication network interface for connecting voice and data equipment to a service provided by a local exchange carrier or long distance carrier
- **Java™ Platform:** a set of computer software and specifications owned by Oracle Corporation, that provides a system for developing application software and deploying it in a cross-platform computing environment.

Top

System Document

Team 03
Project: Neymar
Task: Soccer - Playing Robot

Document version: v.4.0
Date: Tuesday March 22nd, 2016
Author: Matthew Rodin

Edit History: Matthew Rodin fixed the formatting of the document. He also added, clarified and deleted information from sections 2.0 - 9.0 in order to make sure all the specifications are up to date. Finally, he added a "glossary of terms" section

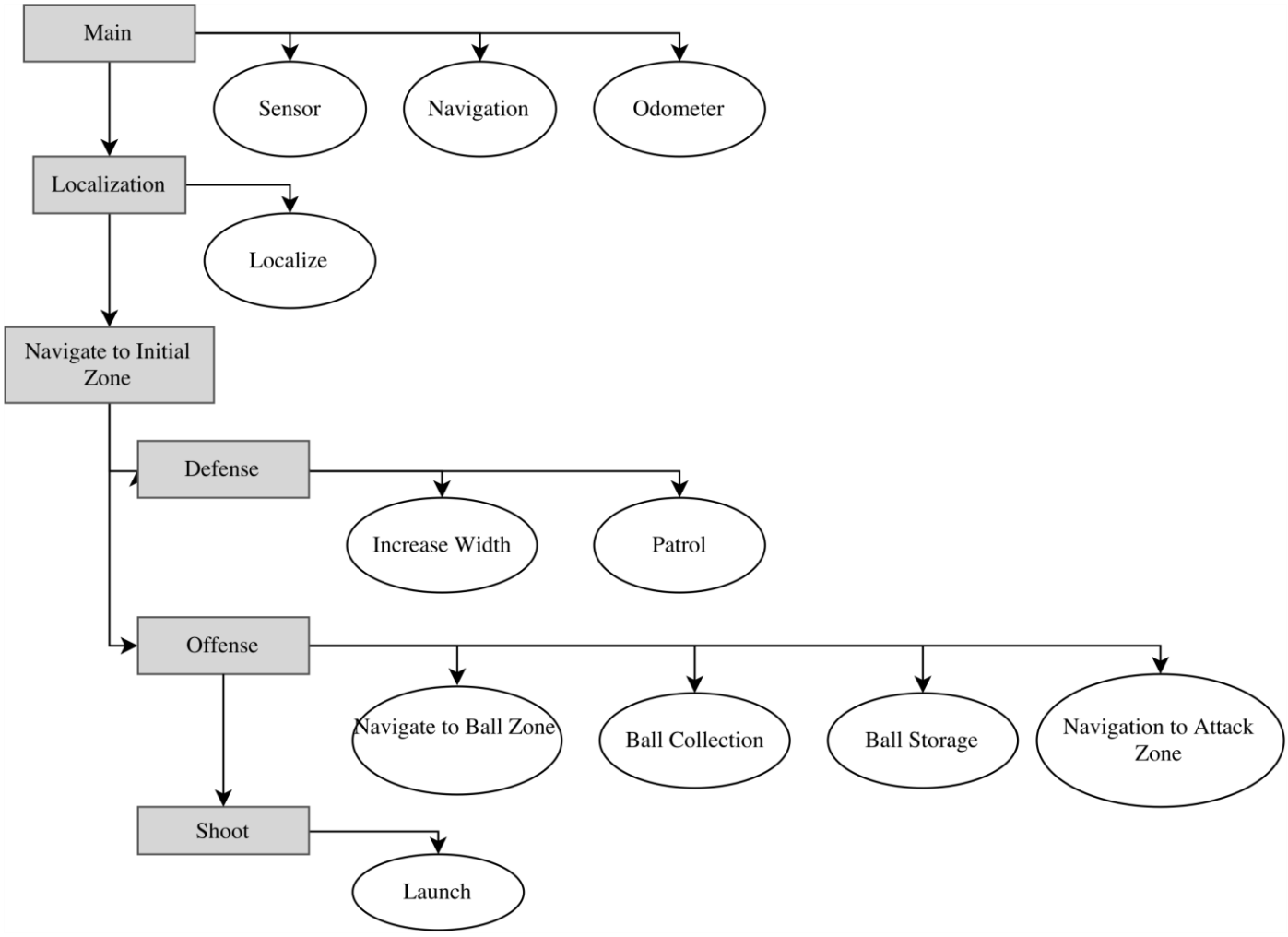
This purpose of this document is to define the solution environment and to understand the system to be used for implementing the solution to the design problem.

1.0 - TABLE OF CONTENTS

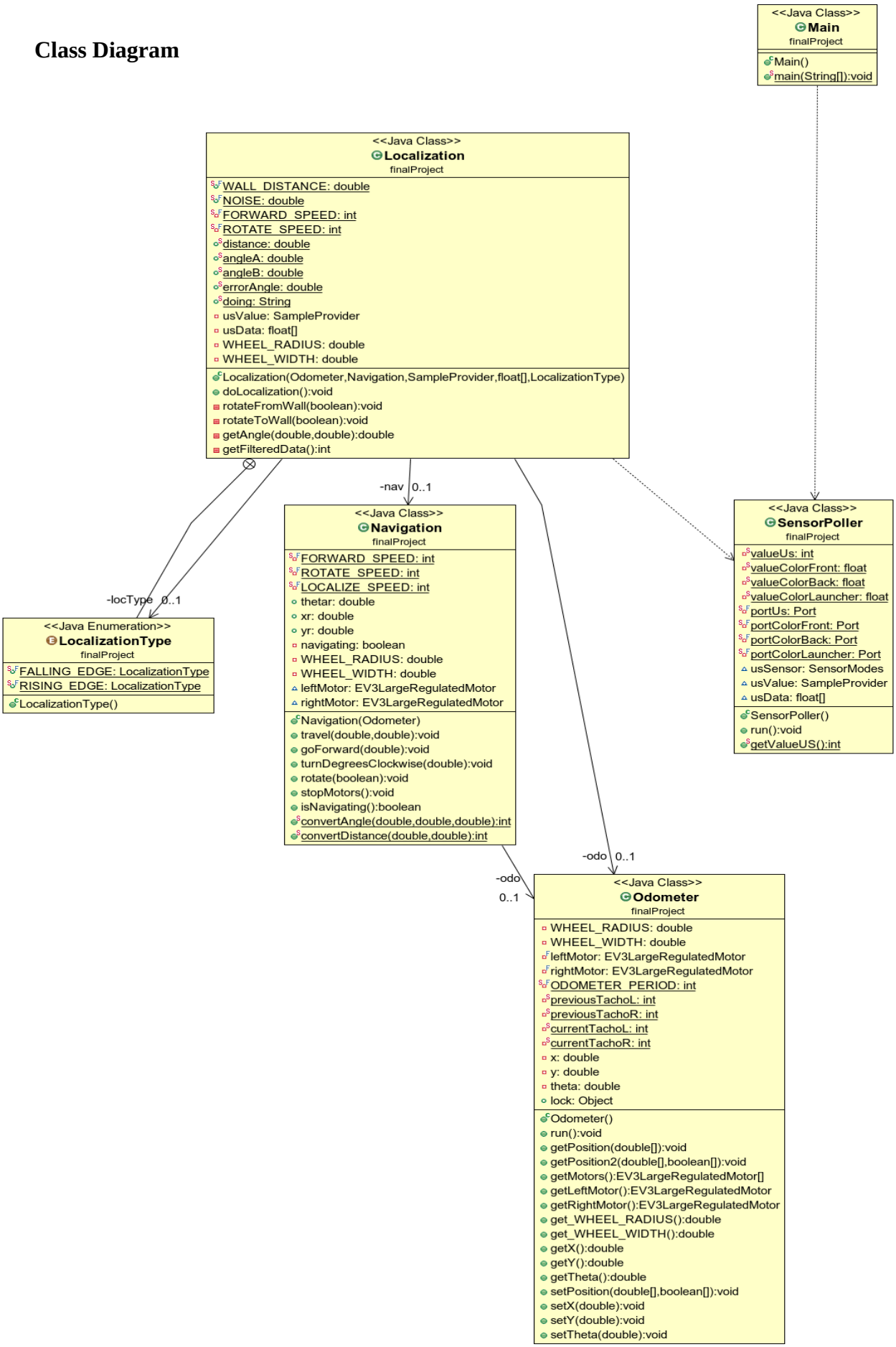
2.0 – SYSTEM MODEL

A robot design, which can localize, navigate, avoid (blocks), identify (ball), and shoot/defend.

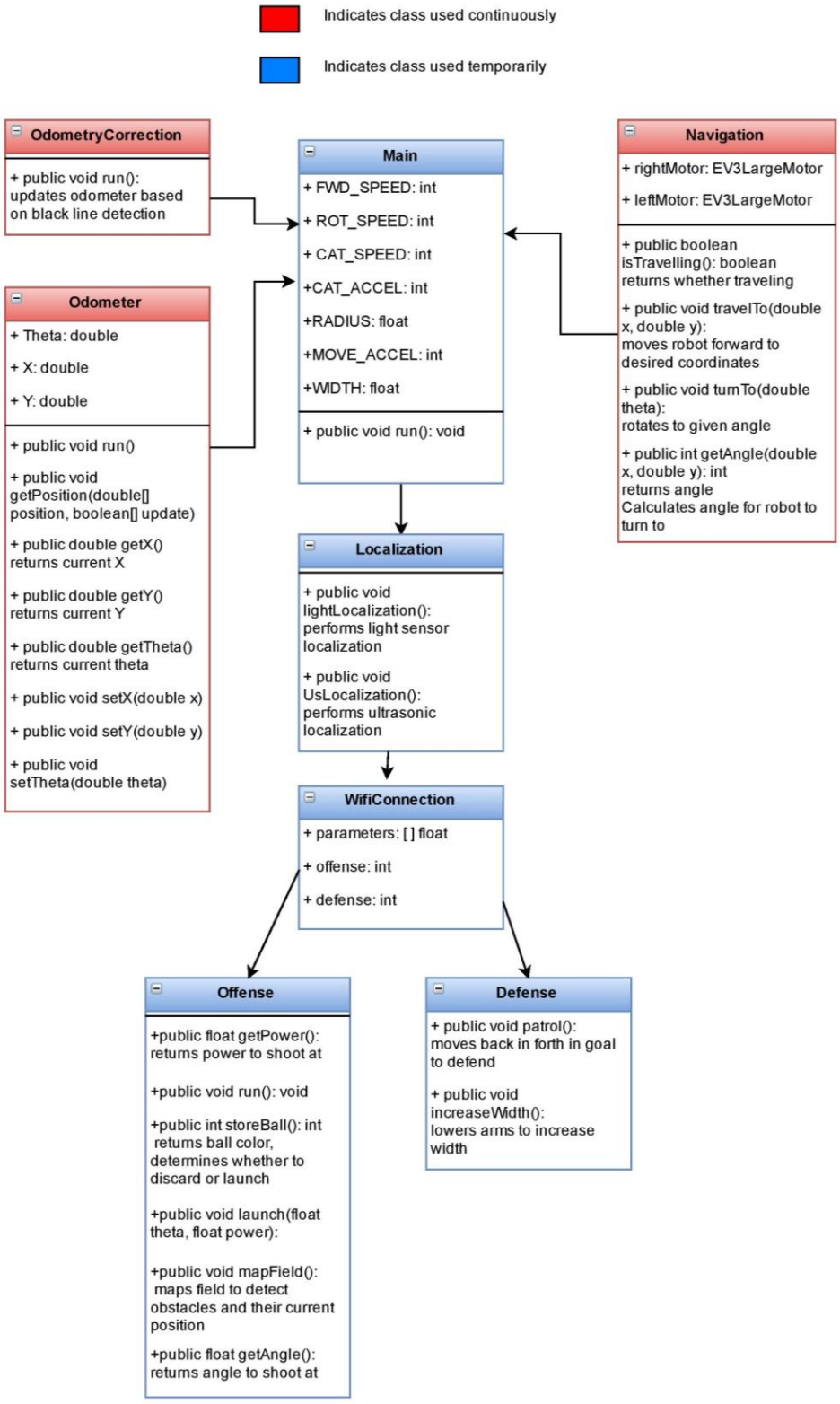
Block Diagram



Class Diagram



Flow Chart


Indicates class used continuously
Indicates class used temporarily
OdometryCorrection
+ public void run(): updates odometer based on black line detection
Main
+ FWD_SPEED: int
+ ROT_SPEED: int
+ CAT_SPEED: int
+CAT_ACCEL: int
+RADIUS: float
+MOVE_ACCEL: int
+WIDTH: float
+ public void run(): void
Navigation
+ rightMotor: EV3LargeMotor
+ leftMotor: EV3LargeMotor
+ public boolean isTravelling(): boolean returns whether traveling
+ public void travelTo(double x, double y): moves robot forward to desired coordinates
+ public void turnTo(double theta): rotates to given angle
+ public int getAngle(double x, double y): int returns angle Calculates angle for robot to turn to
Odometer
+ Theta: double
+ X: double
+ Y: double
+ public void run()
+ public void getPosition(double[] position, boolean[] update)
+ public double getX() returns current X
+ public double getY() returns current Y
+ public double getTheta() returns current theta
+ public void setX(double x)
+ public void setY(double y)
+ public void setTheta(double theta)
Localization
+ public void lightLocalization(): performs light sensor localization
+ public void UsLocalization(): performs ultrasonic localization
WifiConnection
+ parameters: [ ] float
+ offense: int
+ defense: int
Offense
+public float getPower(): returns power to shoot at
+public void run(): void
+public int storeBall(): int returns ball color, determines whether to discard or launch
+public void launch(float theta, float power):
+public void mapField(): maps field to detect obstacles and their current position
+public float getAngle(): returns angle to shoot at
Defense
+ public void patrol(): moves back in forth in goal to defend
+ public void increaseWidth(): lowers arms to increase width

3.0 – HARDWARE AVAILABLE AND CAPABILITIES

- As mentioned in the "Requirements Document", there There is a fixed number of components which can be used to construct the solution (3 Lego Mindstorm Kits)
- The electromechanical segment of the solution is limited to four (4) motors and four (4) sensors due to the EV3's available connection ports.
    - Two motors for mobility
    - One motor for picking up the ball
    - One motor for shooting
- The EV3 uses a TI Sitara AM1808 (ARM926EJ-S core) processor with a clock speed of 300 MHz. Therefore, this microprocessor can execute 300 million instructions per second.

4.0 – SOFTWARE AVAILABLE AND CAPABILITIES

- The tools used for designing the product are Git, Eclipse and leJOS. It is to be noted that leJOS API is open source, which allows the user to see how the given functions and classes are implemented.
- leJOS uses the Java language and consequently, this will be the language of choice for our software system.

5.0 – COMPATIBILITY

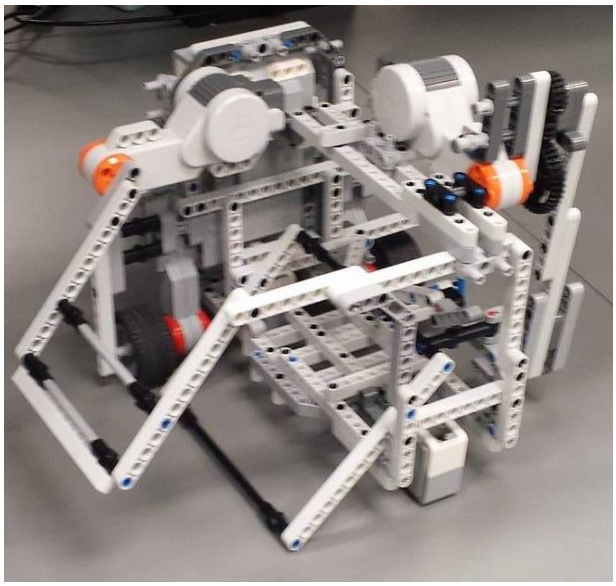
- The software system will be developed in order to be compatible with leJOS' existing structure and provided code.
- Additional code has been developed in order to adhere to the specific requirements of the project description.
- As mentioned in section 3.1 "Component Re-Use" of the "Requirements Document", past code for odometry and navigation used in the first five labs, will be used to speed up development time. However, since different teams developed the code, certain adjustments must be made in order for the different code to work together and for a new mechanical system (i.e. one team may have chosen to write the odometer class using radians for the angle while another team implemented the same class using degrees instead).

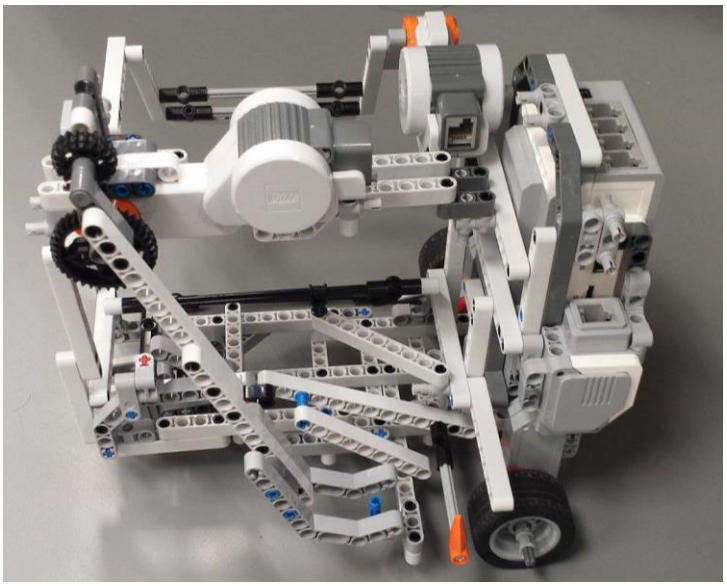
6.0 – REUSABILITY

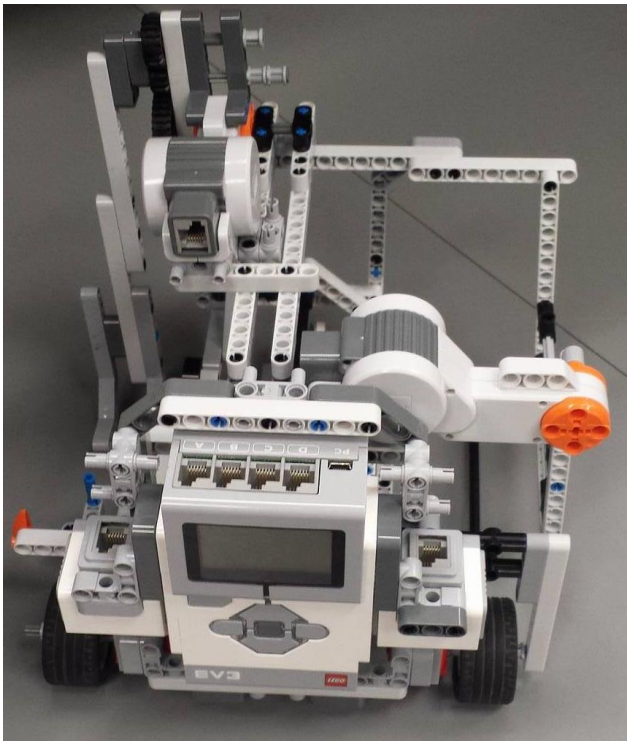
Please refer to sections 5.0 ("Compatibility") and 3.1 of the "Requirements Document" ("Component Re-Use").

7.0 – STRUCTURES

After testing, the group decided that none of the previous two (2) hardware versions were viable to complete the required tasks. Therefore, this is the final design/structure







The newest hardware version (V.3.0) has extensive changes relative to V.2.2. We newest changes include:

- Ball collector is now at the front of the robot and parallel to its motion.
- Ultrasonic sensor is placed at the front of the robot and at ball level.
- Ball collection arm is wider to account for the increase ball size (we were informed that the balls would be ping-pong balls)
- New funnel mechanism that ensures the ping pong balls will be properly directed to the back of the robot and ready for launch

These additions were able to solve the following problems:

- Perpendicular ball retrieval mechanism
- No strategic spot for the ultrasonic sensor
- Multiple balls not being properly directed to the launching mechanism area from the ball retrieval mechanism.

8.0 – METHODOLOGIES

Offense

Using the light sensor in the front for localization and odomertry correction (used for navigation) as well as the two ultrasonic sensors (object detection/distance estimation) the robot will be able to move along the board to area where the balls will be stored. Using its collection arm, the robot will load only the correct color balls (using a light sensor for verification) onto a ramp and then into the funnel. Next, the launching area will receive one (1) ball at a time from the funnel. After the ball has landed on the scoop, the launch arm will rotate completely in order to propel the ball into the net. The robot will repeat this procedure until there are no more balls remaining in the funnel.

Defense

While on defense, the robot will extend its launching arm outwards to 180 degrees (parallel with the ground) and rotate horizontally (with respect to the net) in a random fashion. Given the large width and the random nature of the robot's defensive movements, we believe this gives us a sufficient strategy to defend against the opponent's shots.

9.0 – TOOLS

- USB Key
- 1x Roll of Scotch Tape
- Elastic Bands

10.0 – GLOSSARY OF TERMS

- **Block Diagram:** a diagram of a system in which the principal parts or functions are represented by blocks connected by lines that show the relationships of the blocks
- **Class Diagram:** a diagram in the Unified Modeling Language (UML) that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.
- **Flow Chart:** a diagram that represents an algorithm, workflow or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows.
- **Processor:** the electronic circuitry that carries out the instructions of a computer program.
- **Eclipse:** a software application that provides comprehensive facilities to computer programmers for software development.
- **Git:** source code management system for software development
- **LeJOS:** a firmware replacement for Lego Mindstorms programmable bricks.

Top

Adopted Mechanical Design

Team 03
Project: Neymar
Task: Soccer - Playing Robot

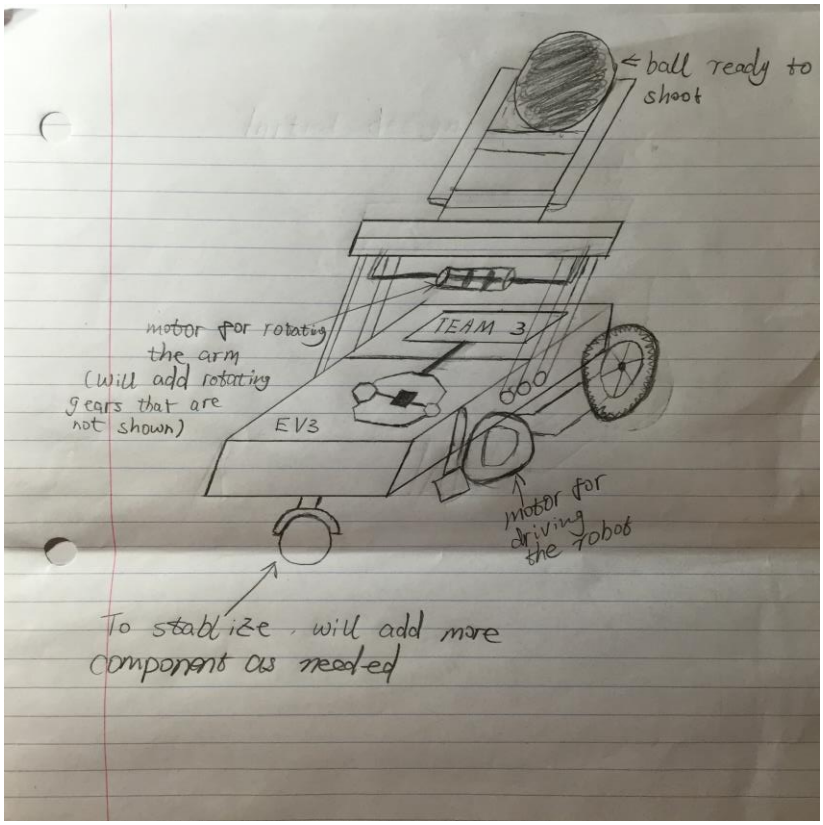
Document version: v.3.0
Date: Tuesday March 26nd, 2016
Author: Matthew Rodin

Edit History: Matthew Rodin fixed the formatting (added table of contents) and added V.3.0 to the document.

This purpose of this document is to provide a complete overview of the progression of the hardware architecture.

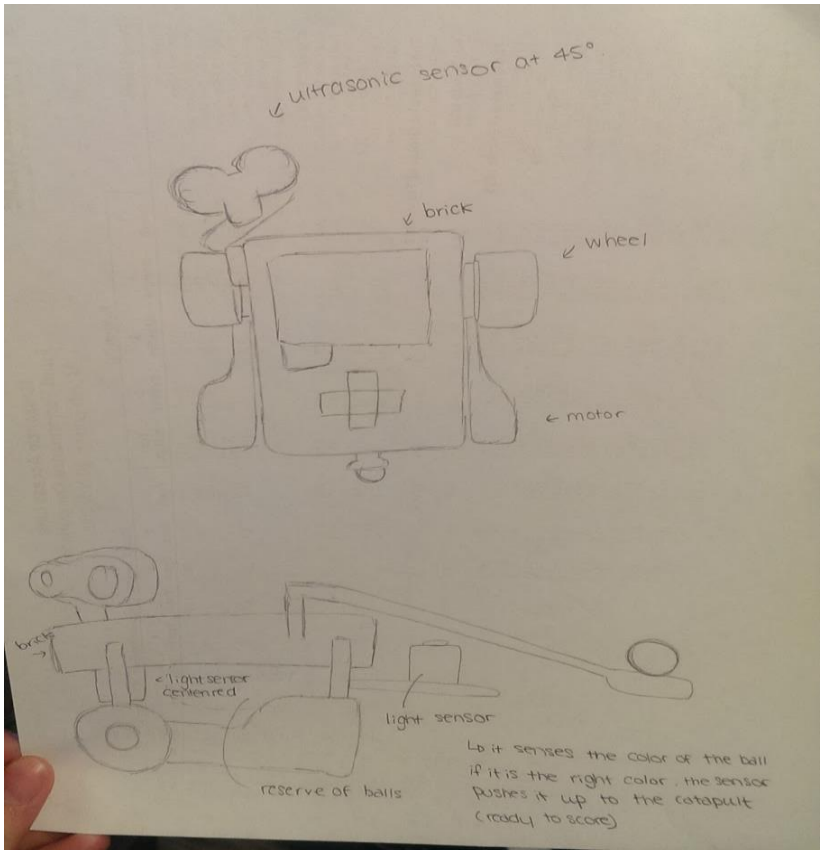
TABLE OF CONTENTS

INITIAL SKETCH



3 PROTOTYPES

PROTOTYPE DESIGN 1



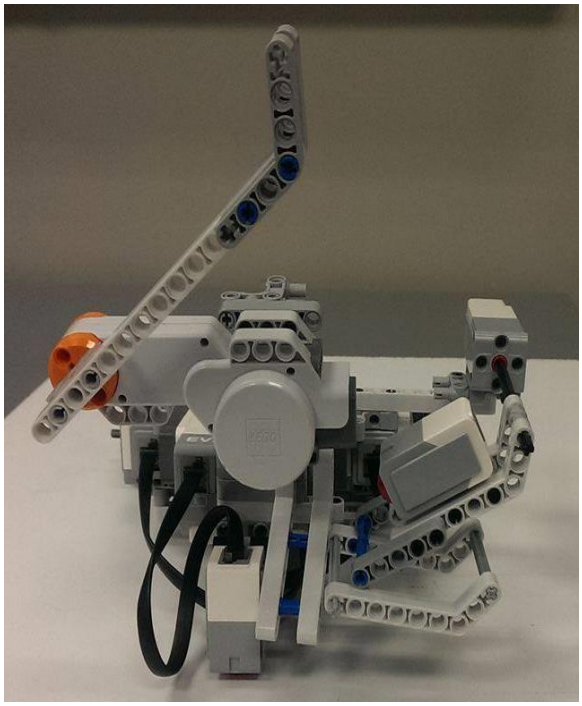
Pros:

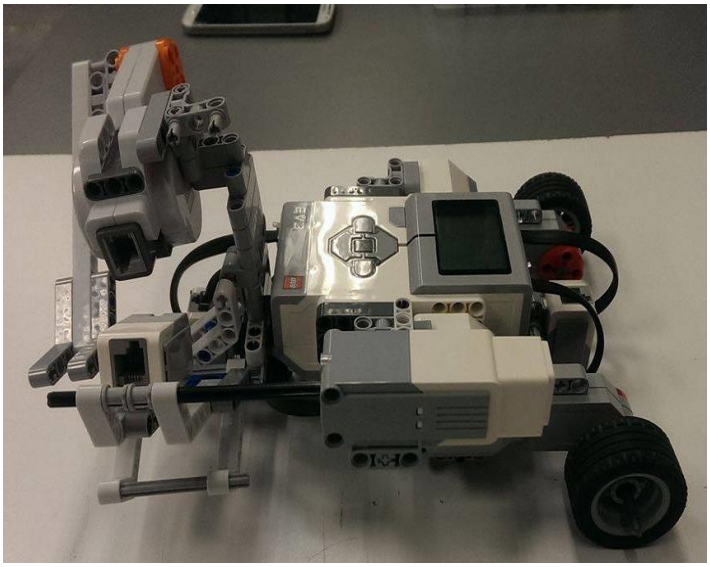
- reserve of balls → no need to travel back and forth
- only one brick : easier coding
- Light sensor aligned with the wheels → easier for navigation

Cons:

- Unstable
- Too heavy for the wheels (they start to compress)
- Hard to pick up balls
- Fixed ultrasonic sensor

PROTOTYPE DESIGN #2





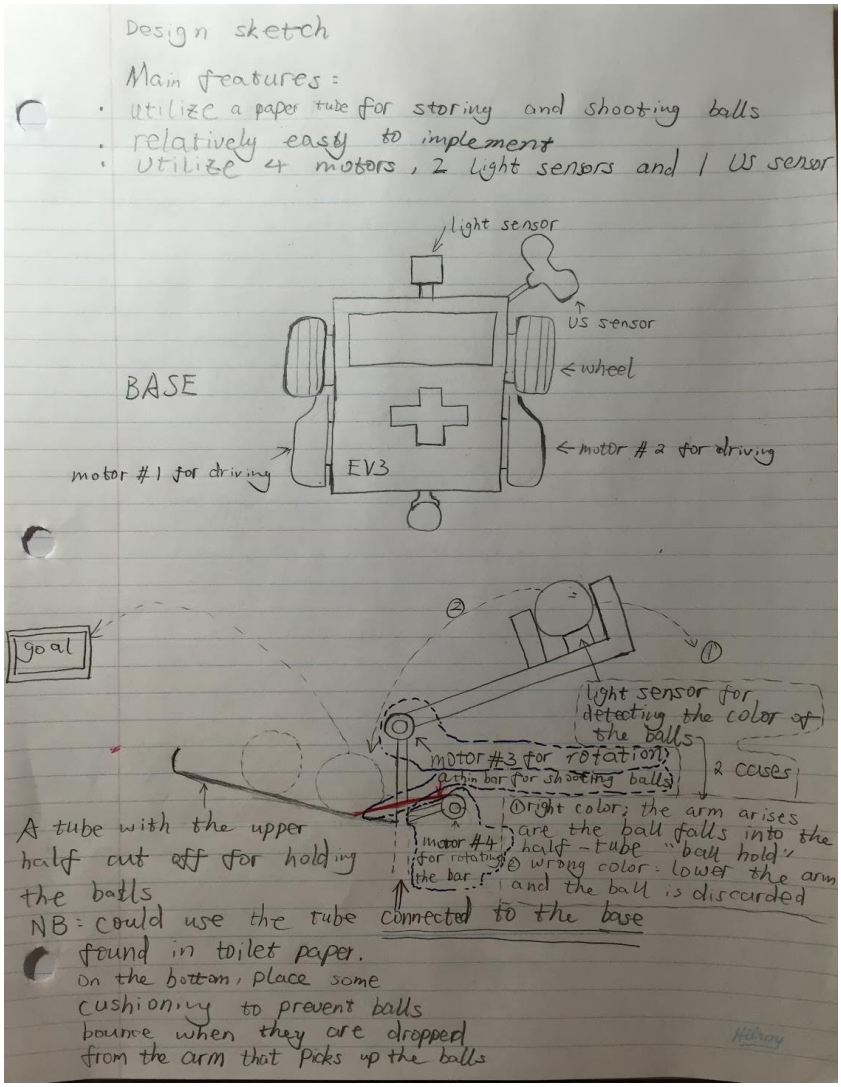
Pros:

- reserve of balls → no need to travel back and forth
- only one brick : easier coding
- Efficient launching

Cons:

- Robot too wide → higher possibility of error when rotating
- Uneven weight distribution
- Too heavy for wheels to handle

PROTOTYPE DESIGN #3



Pros:

- relatively easy to build
- the shooting mechanism is light, so the robot is kept stable
- ability to hold multiple balls at the same time to save time going back and forth to pick up the balls

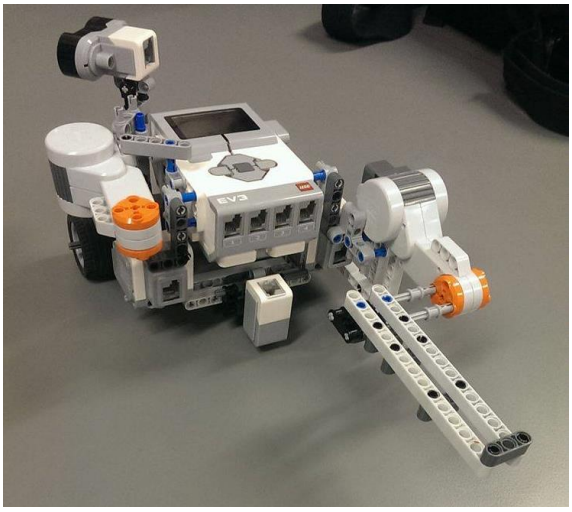
Cons:

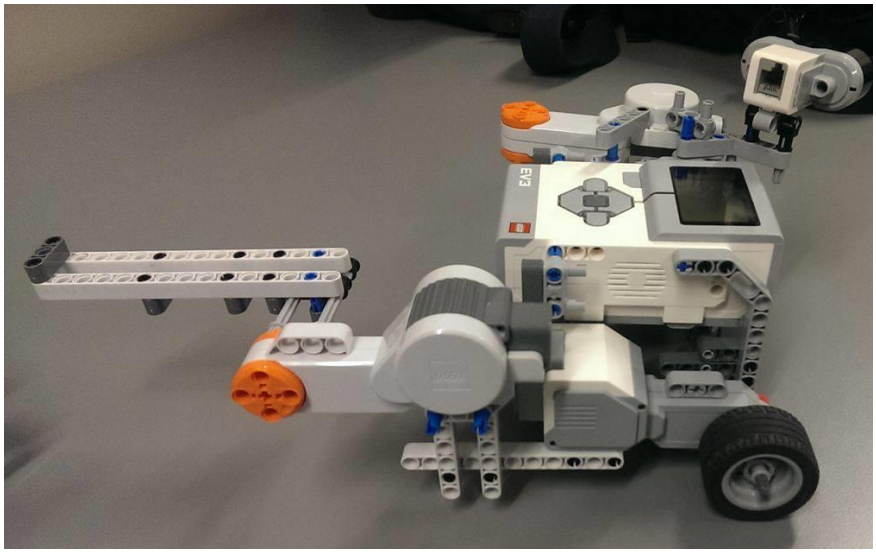
- the thin bar might not hit the ball on the tube accurately, the ball might not go in the direction desired.
- when dropping the balls from the arm that picks up the balls, it might bounce, making it possible for the balls to fall out of the tube

After testing, the group decided that none of the previous 3 options were viable to complete the required tasks. Therefore, we were forced to completely change the design of our robot…

DESIGN VERSIONS

DESIGN (V.1)





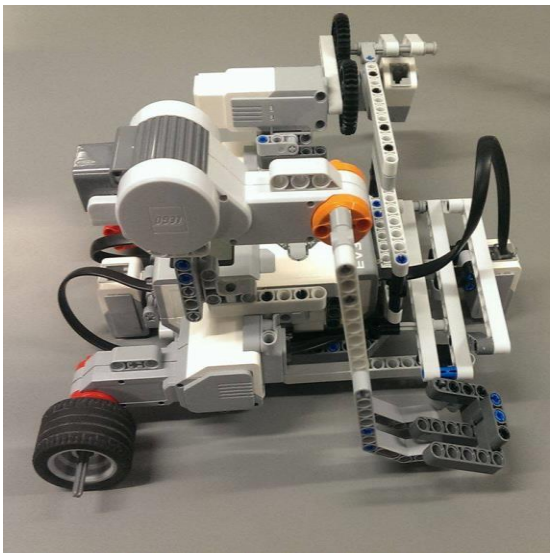
This is our first design prototype. A little arm should be attached to the motor on the left. It swipes the ball horizontally onto a slanted paper that will bring it to catapult. We decided to attach a paper to the arm to create a sort of half tunnel to enclose the ball when it slides towards the catapult's ball bucket. This tunnel will facilitate storing the ball in the arm. There will be a ball-wide hole right at the bucket. Thus, when launching the catapult, only the last ball will be shot.

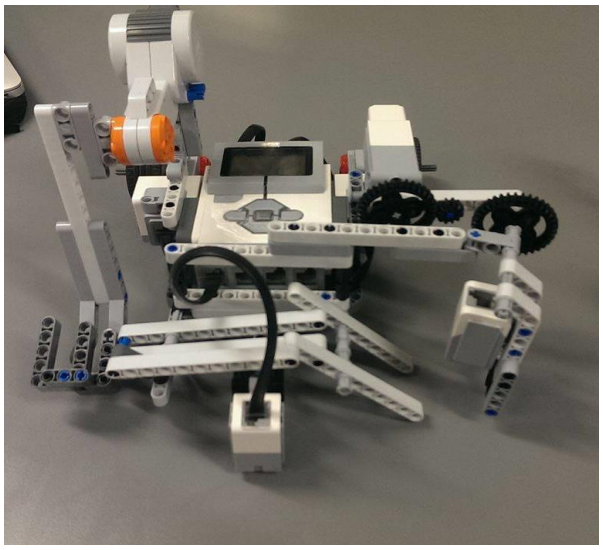
Current Problem:

- Unstable design: the back support of the weight of the robot is too close to the front leaving the base of support too narrow for a stable design.
- Non practical: Some components tend to block the ports.
- The launcher is not solid enough.

Conclusion:
This design is not practically compatible with the project.
The robot cannot finish the required tasks.

DESIGN (V.2)





The long arm that is placed on the side of the robot is used to pick up the balls from the ball pool. The balls are then lifted up along the slant on the side of the robot and lined up for the catapult. The ball that can pass through the triangle is then held at the end of the robot with a light sensor sensing the color of the ball. The ball is the picked up by the catapult. Depending on the color of the ball, the catapult would apply different forces (by varying the speed of the catapult) so that the ball can either be discarded (if the color is wrong) or shot to the destination (if the color is right).

What changed from the last version?
- Complete redesign of robot.
- Strategic placement of components in order to have adequate weight distribution.
- Implementation of light sensor, which verifies the correct color of the ball before launch.
- Implementation of shorter but more stable launcher (includes “scoop” in order to avoid ball slipping off launcher)

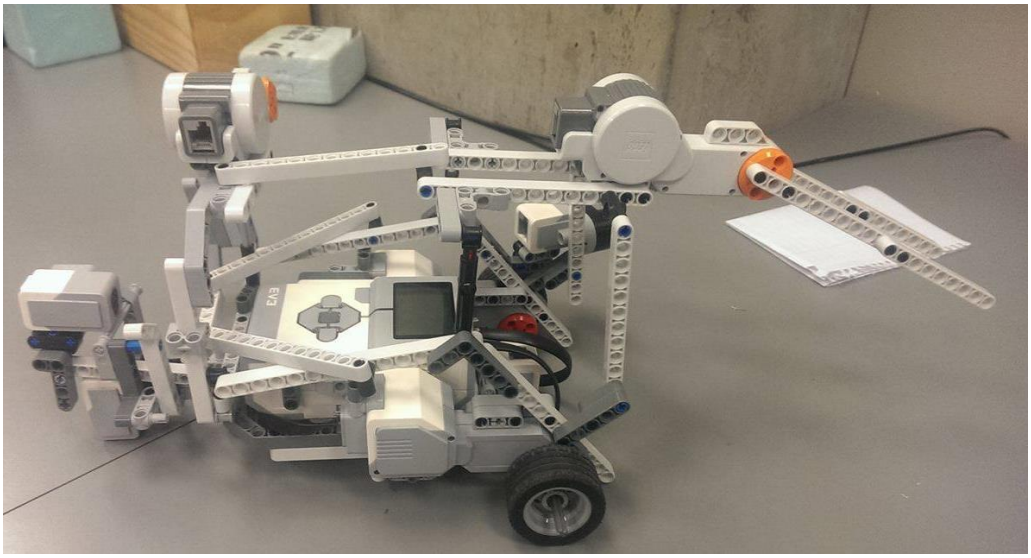
Why did we change:
We needed to completely redesign to account for the multitude of design flaws with version 1.0 (i.e. unstable weight distribution, non-practicality of certain components blocking ports and unstable launcher).

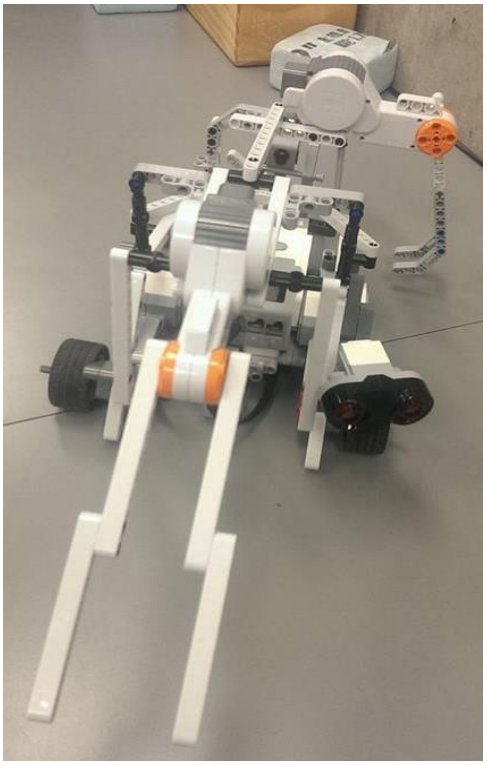
Problem(s) solved:
-Unstable weight distribution
-Unstable launcher
-Blocked ports.

Current Problem (if any):
- No practical location on the robot to place the ultrasonic sensor in order to obtain the readings necessary.
- The ball collector is perpendicular to the robot's motion (and detection). This would require the robot to rotate to its side after it detects the ball area and drastically decrease the success rate of ball retrieval.
-Some balls are not being properly directed to the back of the robot (launching area) after being retrieved.

Conclusion:
We need an updated version which implements a ball collector mechanism which is parallel to the robot's motion and which is able to have sufficient space to implement an ultrasonic sensor in a convenient location. As well, we will add a "funnel" which will ensure the proper path of the ball from retrieval area to launching area.

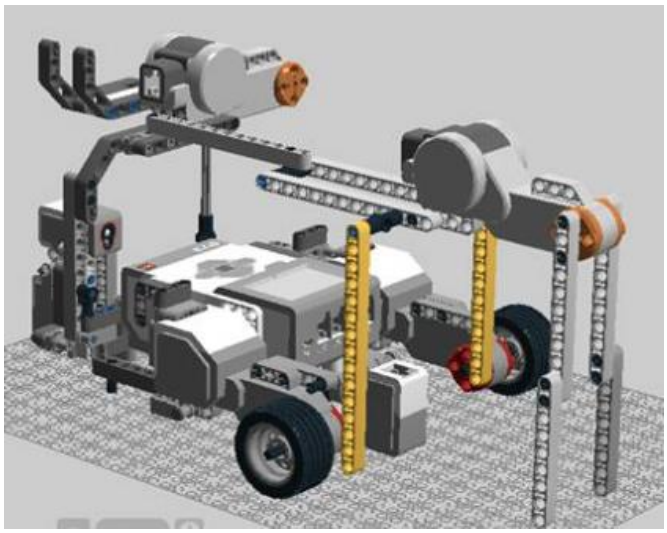
DESIGN (V.2.1)

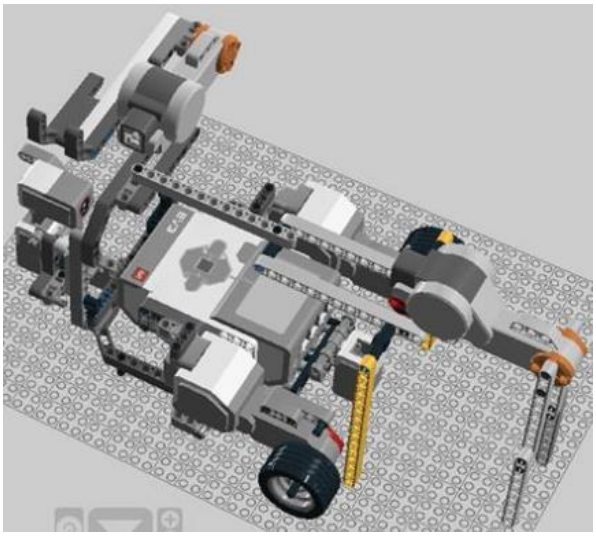




The long arm that is placed in front of the robot is used to pick up the balls from the ball pool. The balls are then lifted up to the top of the robot then they would roll down to the end of the robot, where a triangular shaped mechanism (used to store multiple balls) is placed such that only one ball is allowed to pass at a time. The ball that can pass through the triangle is then held at the end of the robot with a light sensor sensing the color of the ball. The ball is the picked up by the catapult. Depending on the color of the ball, the catapult would apply different forces (by varying the speed of the catapult) so that the ball can either be discarded (if the color is wrong) or shot to the destination (if the color is right).

There is a LDD provided for this design.
However, we could not draw the special triangular shaped mechanism that is placed on top of the robot because LDD cannot seem to be able to bend the components 45 degrees.





What changed from the last version:
- Ball collector is now at the front of the robot and parallel to its motion.
- Ultrasonic sensor is placed at the front of the robot and at ball level.
- Ball collection arm is wider to account for the increase ball size (we were informed that the balls would be ping-pong balls)
-New funnel mechanism that ensures the ping pong balls will be properly directed to the back of the robot and ready for launch.

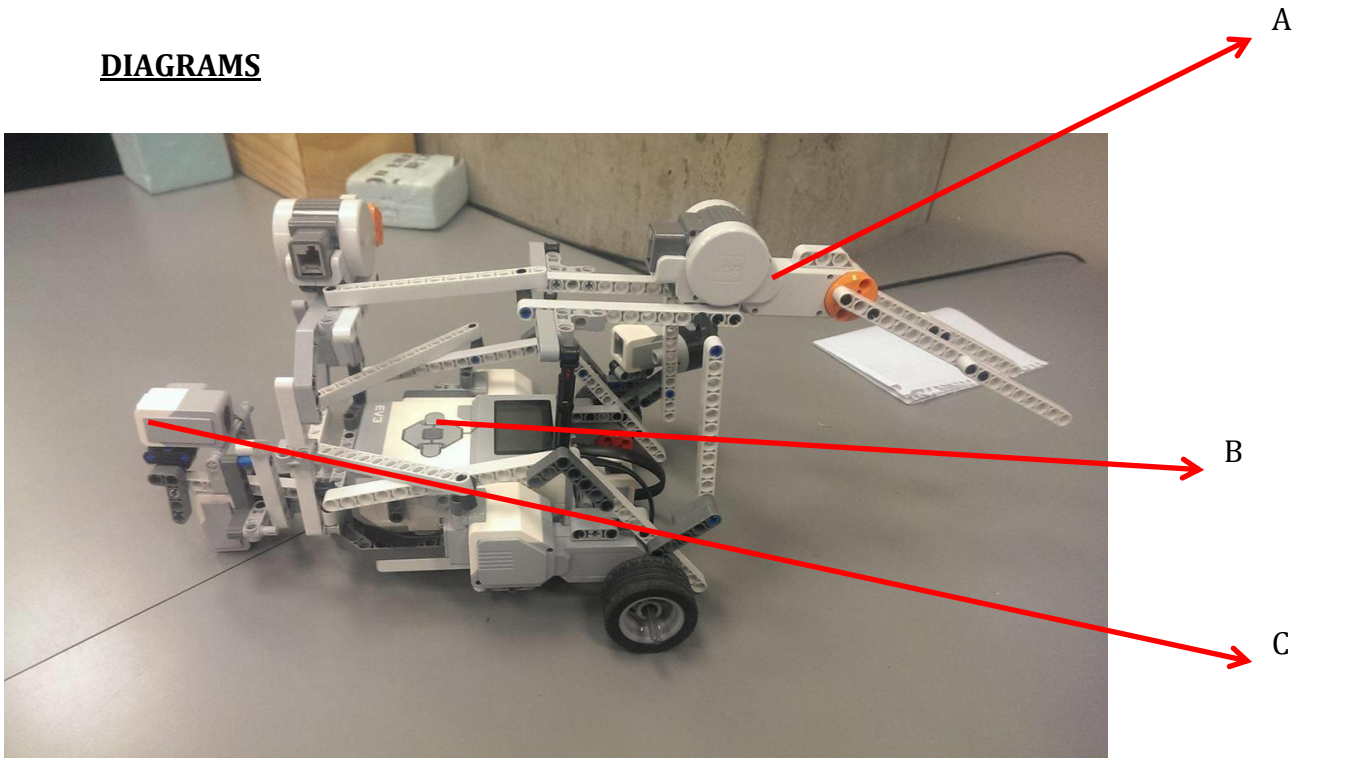
Why did we change:
This version update was necessary because it solves the issues of version 2.0 (perpendicularity of the ball collector resulted in rotation and error increase in ball collection and the ultrasonic sensor could not strategically be placed on the robot).

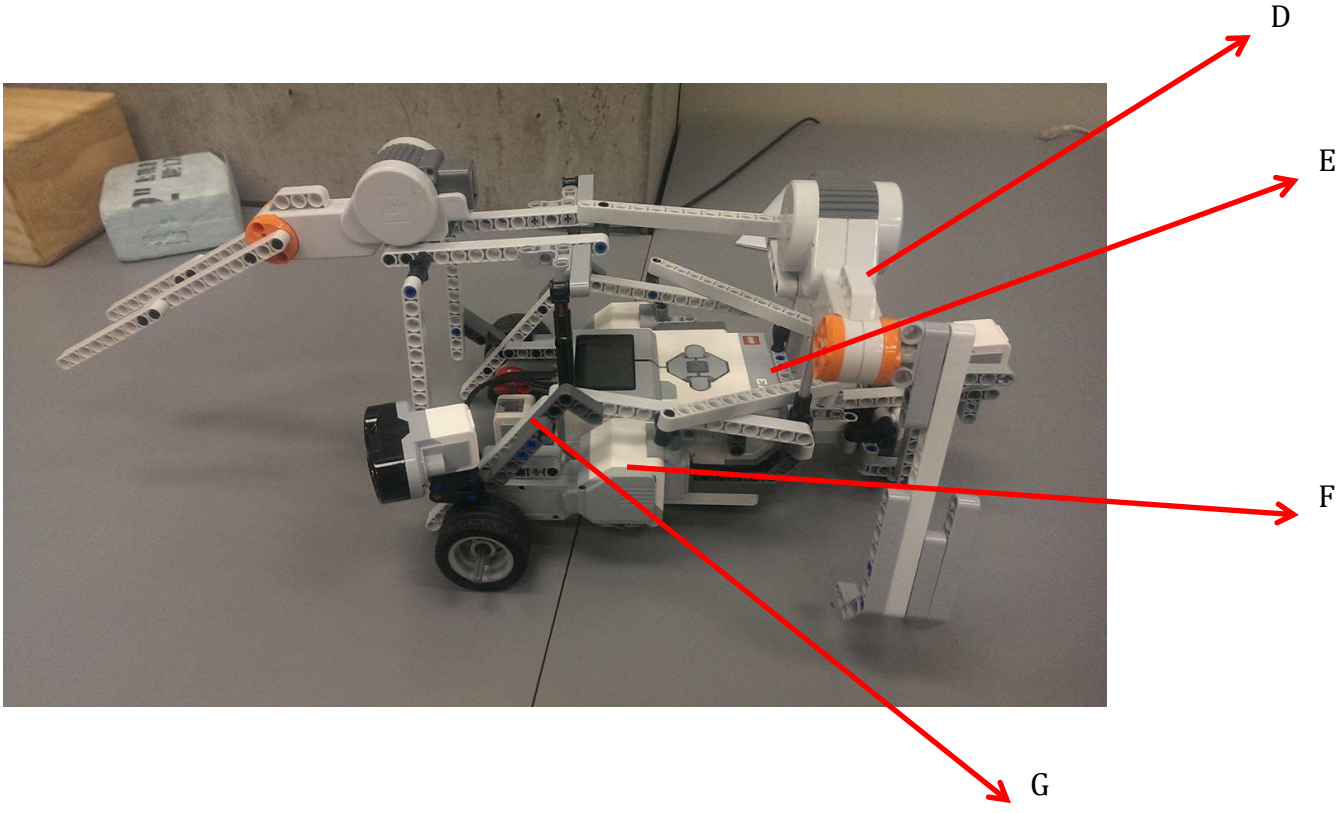
Problem(s) solved:
-Perpendicular ball retrieval mechanism
-No strategic spot for the ultrasonic sensor
-Multiple balls not being properly directed to the launching mechanism area from the ball retrieval mechanism.

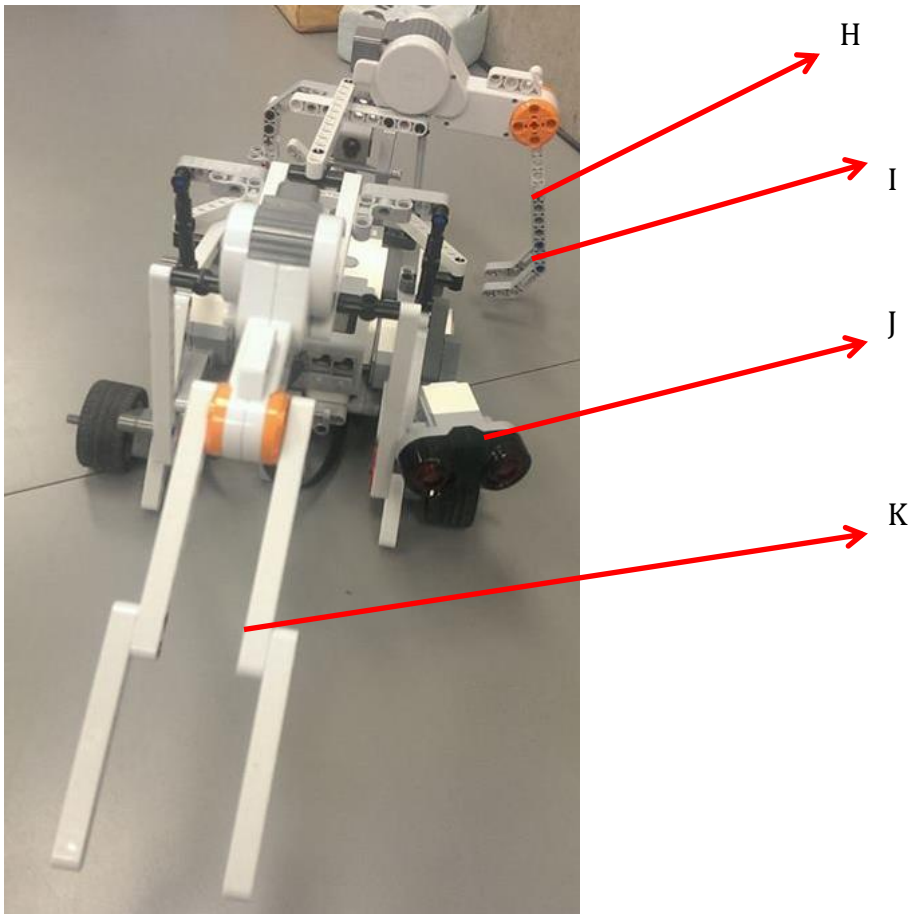
Current Problem (if any):
-No current problems

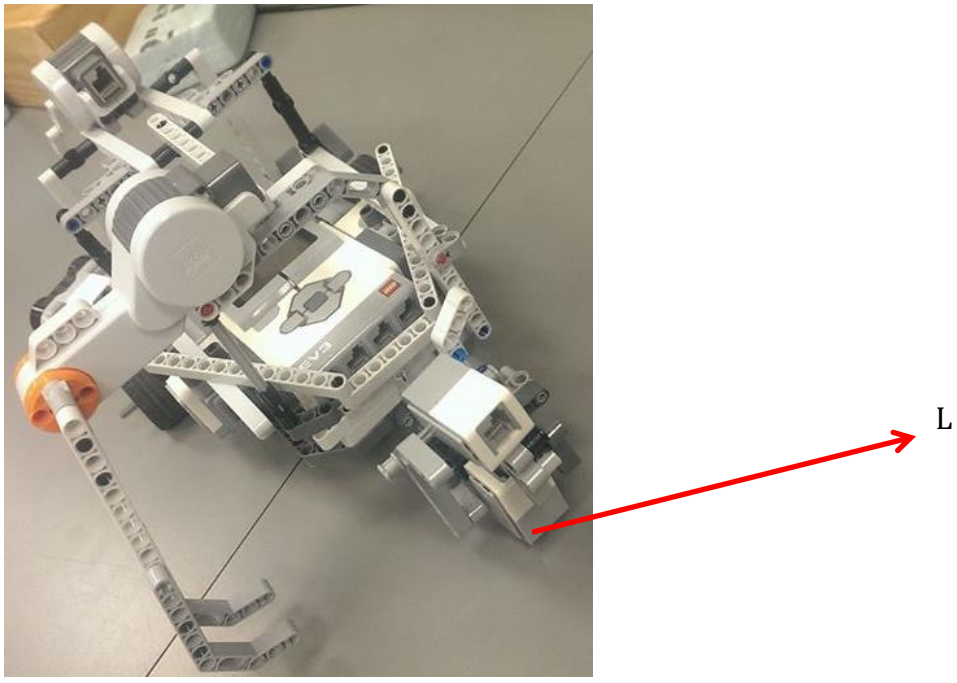
Conclusion:
With no current problems to our newest prototype, we are ready to test our robot and verify if it can properly complete the required tasks.

DIAGRAMS






H
I
J
K


L

FEATURES

A. **Motor #1:** Responsible for the rotation of the ball collector

B. **EV3 Brick:** programmable with the LeJOS firmware and the Java programming language

C. **Sensor #1:** Light sensor responsible for verifying the correct color of the ball before launch

D. **Motor #2:** Responsible for the rotation of the launching arm.

E. **Funnel:** Arm extending at an angle of each side of the EV3 brick which ensures the ball will be properly directed into the launching area

F. **Motor #3 & Motor #4:** One on each side of the robot responsible for the rotation of the robots wheels.

G. **Sensor #2:** Light sensor responsible for odometry correction which will be used for navigation

H. **Launching Arm:** Responsible for projecting the ball into the net.

I. **Scoop:** Responsible for ensuring the ball will not slip off the arm while being projected.

J. **Sensor #3:** Ultrasonic Sensor responsible for recognizing object shapes and estimating their distance from the robot.

K. **Ball Collector:** Responsible for collecting balls from the designated area and directing them to the back of the robot (launching area).

L. **Sensor #4:** Light sensor responsible for localization

GENERAL PROCEDURE

Offense

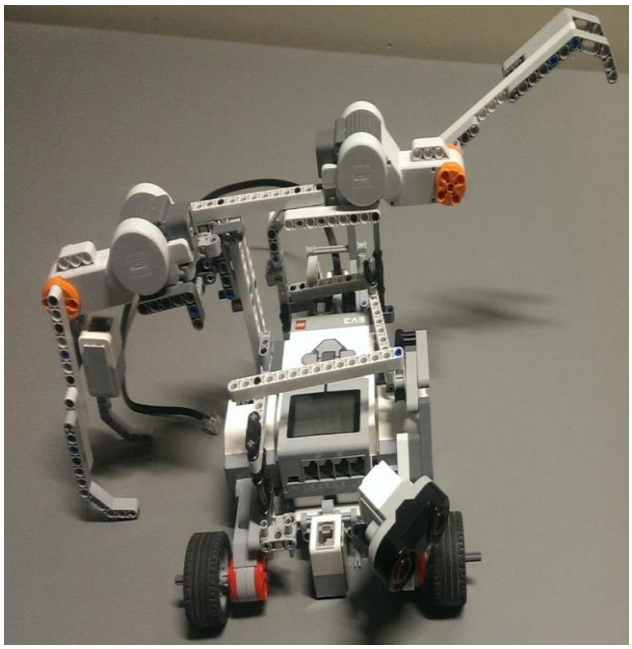
Using the light sensors in the front (navigation) and in the back (localization) as well as the ultrasonic sensor (object detection/distance estimation) the robot will be able to move along the board to area where the balls will be stored. Using its collection arm, the robot will load every ball (regardless of its color) into the funnel. Next the launching area will receive one (1) ball at a time from the funnel. Each ball will pass through a light sensor to detect whether it is the correct color before

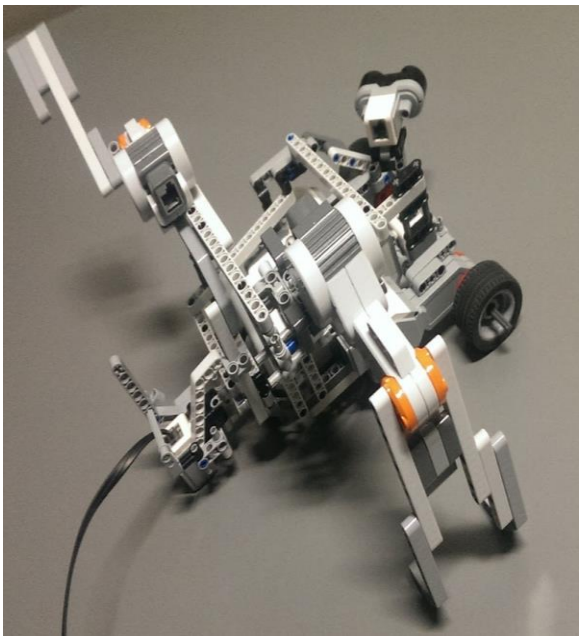
entering the scoop. If the ball is not the correct color, the launch arm will rotate upwards *slightly* in order to "let go" of the ball and cancel the launch. If the ball is the correct color, the launch arm will rotate *completely* in order to launch the ball into the net. The robot will repeat this procedure until the allotted amount of correct-colored balls are launched towards the net.

Defense

While on defense, the robot will extend its arm outwards to 180 degrees (parallel with the ground) and rotate horizontally (with respect to the net) in a random fashion. Given the large width and the random nature of the robot's defensive movements, we believe this gives us a sufficient strategy to defend against the opponents shots.

DESIGN (V.2.2)





What changed from the last version:

- We changed the placement of the ball collecting arm from the front to the side of the robot
- We changed the size of the robot and the launcher to fit the bigger ball. (We elevated the ball storage area
- We added a light sensor on the ball collector.

Why did you change:

- The professor recently specified that the balls will be placed upon a wooden rack during the competition. Having the arm at the front of the robot demands a lot of precision in navigation since the robot will have to rotate and navigate back and forth in order to collect all four balls.
- The light sensor is used to read the color of the ball. Since the balls are bigger, it would not be as practical to store them all onto the robot. Therefore, we thought that only collecting the balls of the correct colors will facilitate the task of the robot.

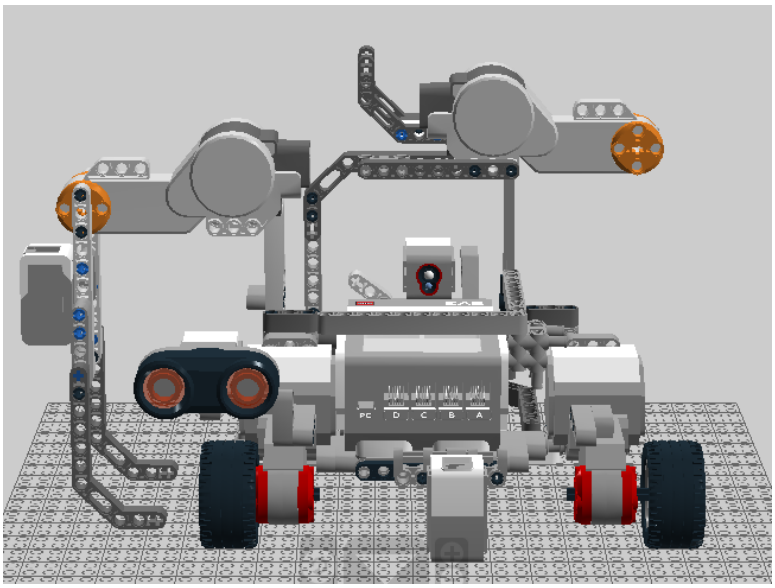
Problem solved:

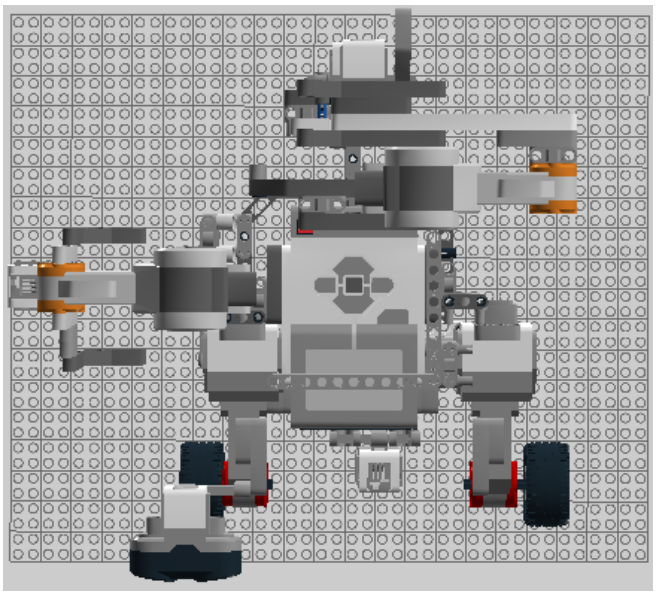
- Direct ball retrieval mechanism: the robot can just collect all the balls while travelling in a straight line by the wooden rack
- Ball storage big enough for the new ball.

Current Problem (if any):
No current problem

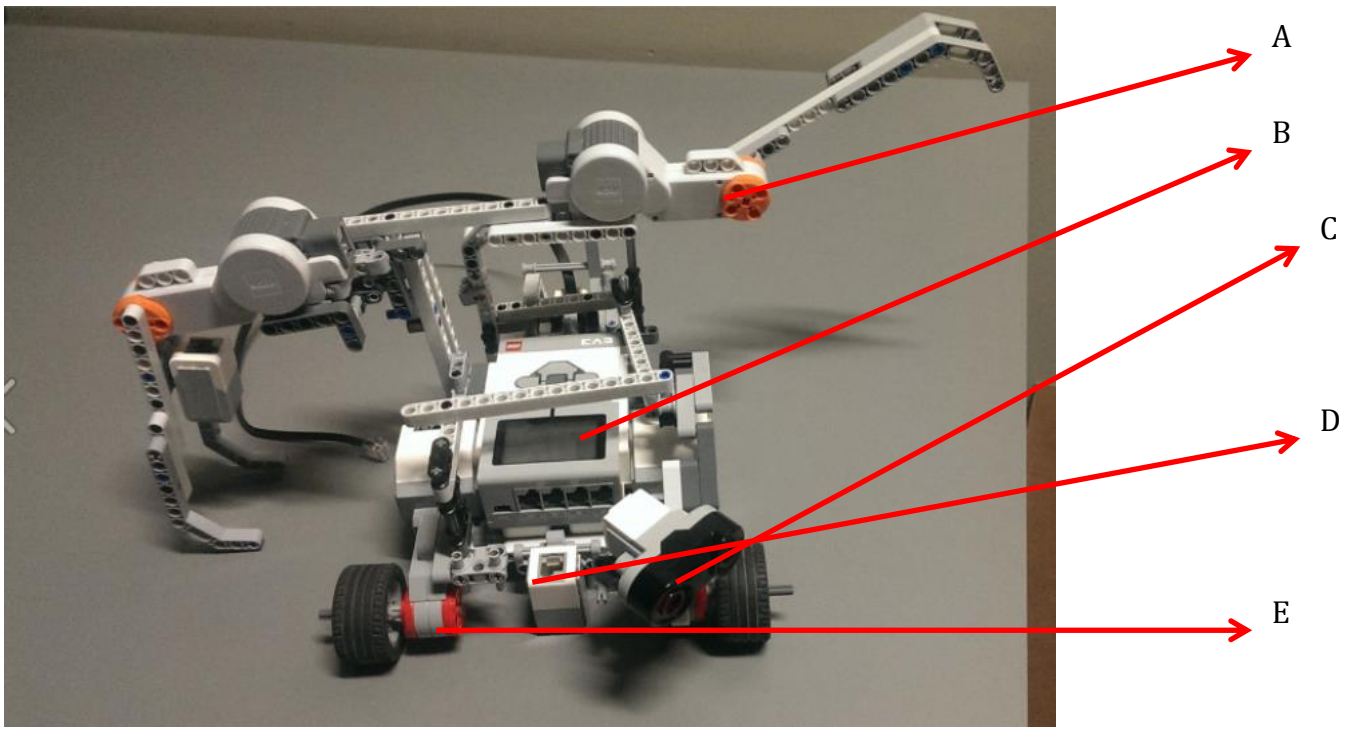
Conclusion:
With no current problems to our newest prototype, we are ready to calibrate the robot to the tasks it is assigned to.

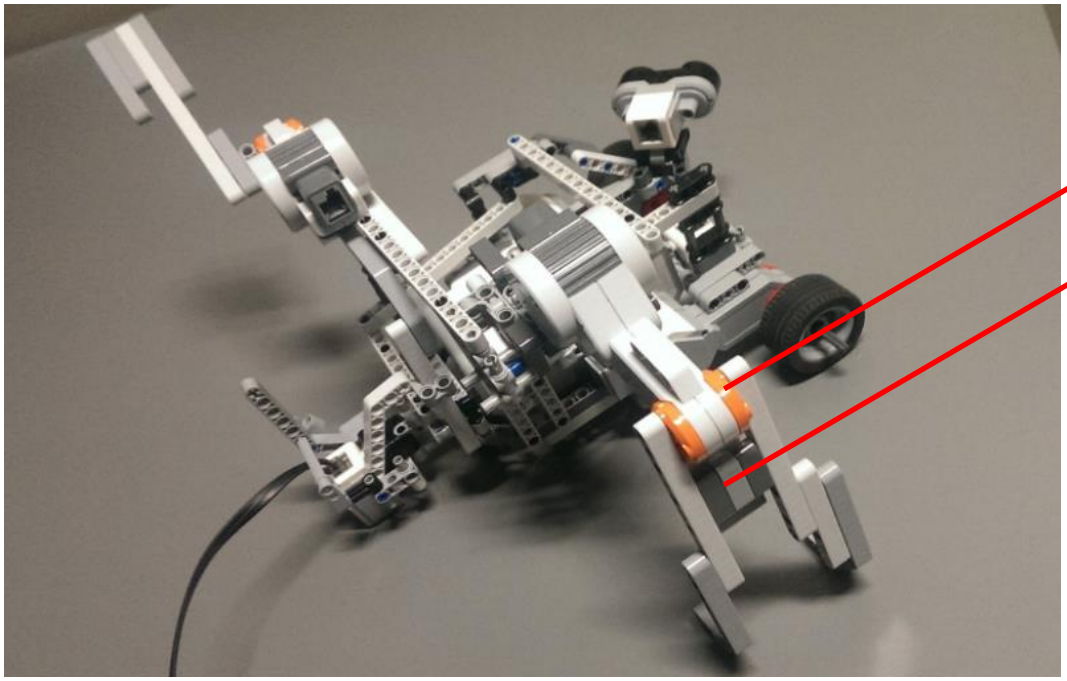
LDD DESIGN

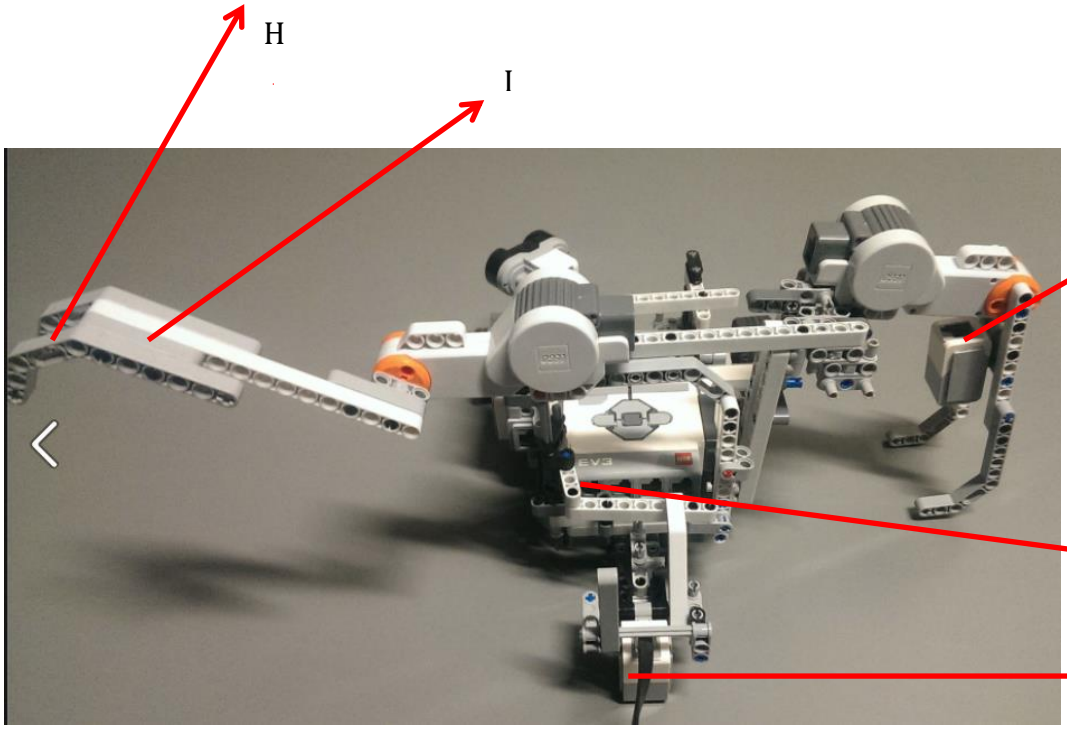




DIAGRAMS




F
G


H
I
J
K
L

FEATURES

A. **Motor #1:** Responsible for the rotation of the launching arm.

B. **EV3 Brick:** programmable with the LeJOS firmware and the Java programming language

C. **Sensor #1:** Ultrasonic Sensor responsible for recognizing object shapes and estimating their distance from the robot.

D. **Sensor #2:** Light sensor responsible for odometry correction which will be used for navigation

E. **Motor #2 & Motor #3:** One on each side of the robot responsible for the rotation of the robots wheels.

F. **Motor #4:** Responsible for the rotation of the ball collector.

G. **Ball Collector:** Responsible for collecting balls from the designated area and directing them to the back of the robot (launching area).

H. **Scoop:** Responsible for ensuring the ball will not slip off the arm while being projected.

I. **Launching Arm:** Responsible for projecting the ball into the net.

J. **Sensor #3:** Light sensor responsible for verifying the correct color of the ball before collection

K. **Funnel:** Arm extending at an angle of each side of the EV3 brick which ensures the ball will be properly directed into the launching area

L. **Sensor #4:** Light sensor responsible for localization

GENERAL PROCEDURE

Offense

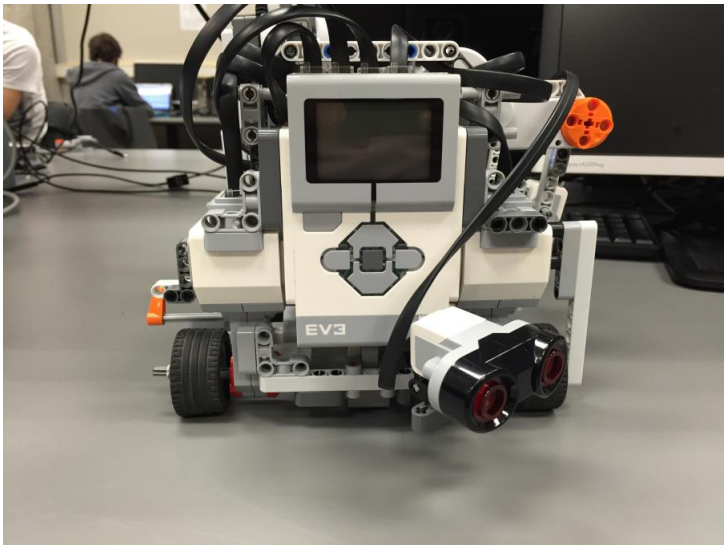
Using the light sensors in the front (navigation) and in the back (localization) as well as the ultrasonic sensor (object detection/distance estimation) the robot will be able to move along the board to area where the balls will be stored. Using its collection arm, the robot will load only the correct color balls (using a light sensor) into the funnel. Next the launching area will receive one (1) ball at a time from the funnel. After the ball has landed on the scoop, the launch arm will rotate completely

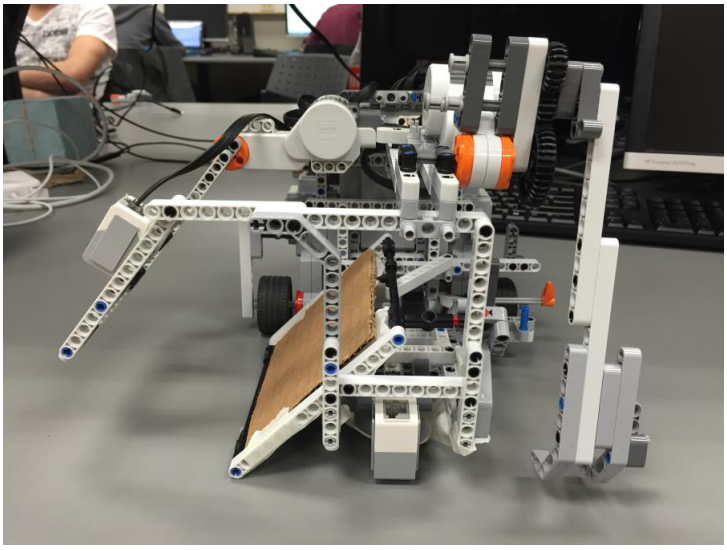
in order to launch the ball into the net. The robot will repeat this procedure until there are no more balls waiting in the funnel.

Defense

While on defense, the robot will extend its arm outwards to 180 degrees (parallel with the ground) and rotate horizontally (with respect to the net) in a random fashion. Given the large width and the random nature of the robot's defensive movements, we believe this gives us a sufficient strategy to defend against the opponent's shots.

DESIGN (V.3.0)





What changed from the last version:

- Ev3 Brick vertical and placed at the front of robot
- Spoon head used instead of marble
- Changed direction of launching arm
- Added gears to the launching arm
- Increased size of collection arm
- Piece of cardboard to act as ramp
- Added Lego pieces to funnel

Why did you change:

These changes were made for various reasons. Being that the playing board is made up of three (3) individual 4x4 boards, a separation gap is created at their intersection. Therefore, our robot was getting stuck in this "crack". The spoon was used because Professor Ferrie mentioned that when navigating between these tiles, the contact point should be enlarged as to minimize the chances of getting stuck. As well, the collection arm was enlarged as to increase the chances of physically collecting the ball and to not rely solely on the accuracy of the localization. The cardboard ramp was added to properly direct the collected balls towards the funnel. Finally, the brick was placed in a vertical direction and at the front of the robot in order to minimize the interference of the connection wires. Previously, the wires were limiting the range of our collection arm as well as the launching arm.

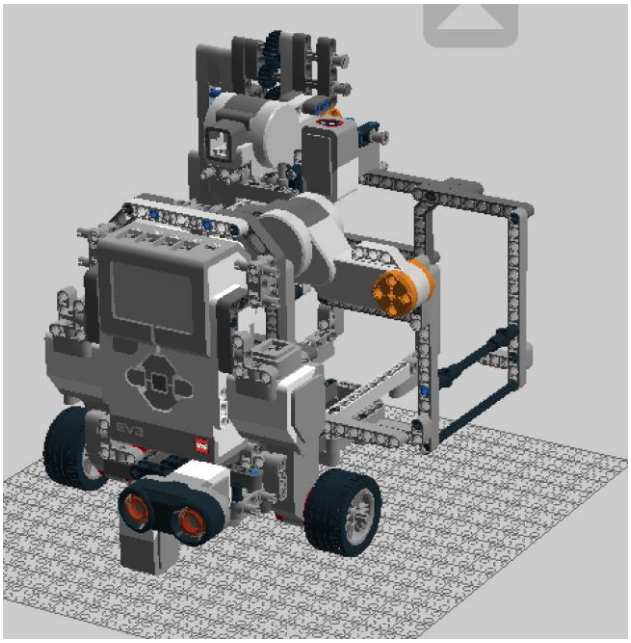
Problem solved:

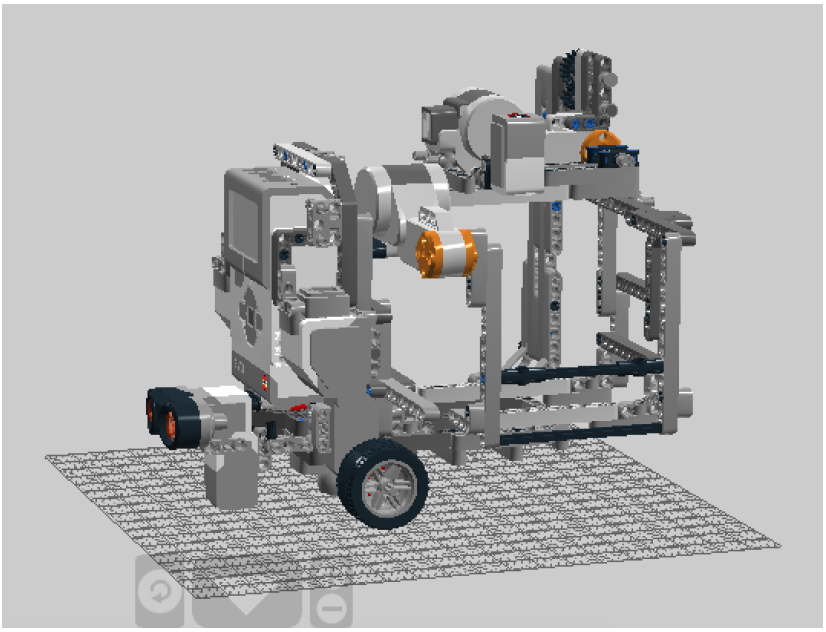
- Interference of wires
- Getting stuck in between tiles
- Poor ball collection system

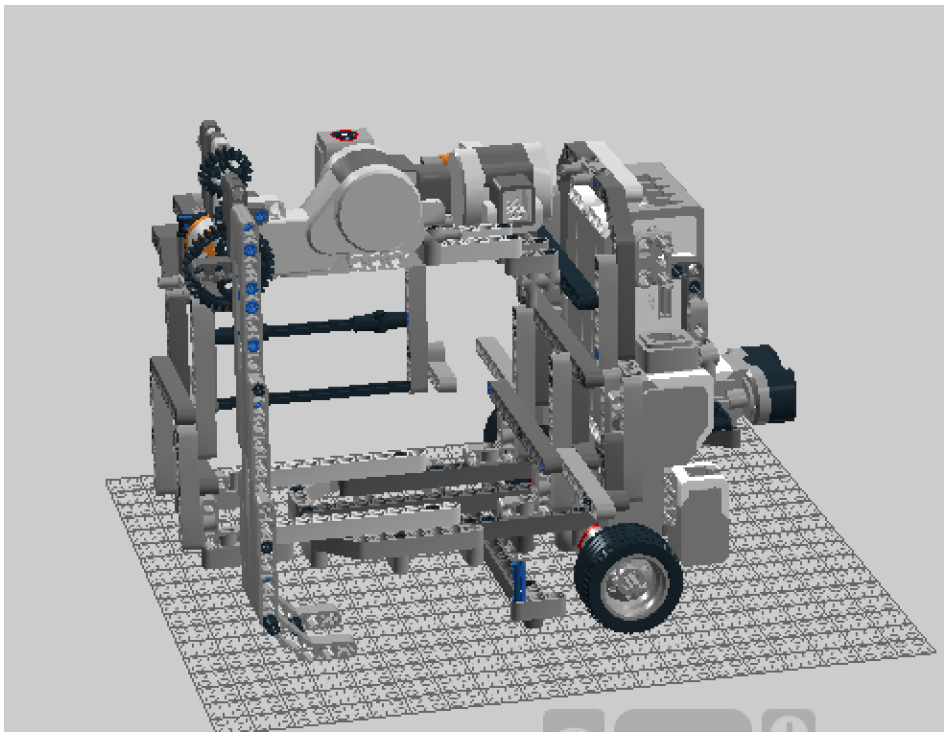
Current Problem (if any):
No current problem

Conclusion:
With no current problems to our newest prototype, we are ready to calibrate the robot to the tasks it is assigned to.

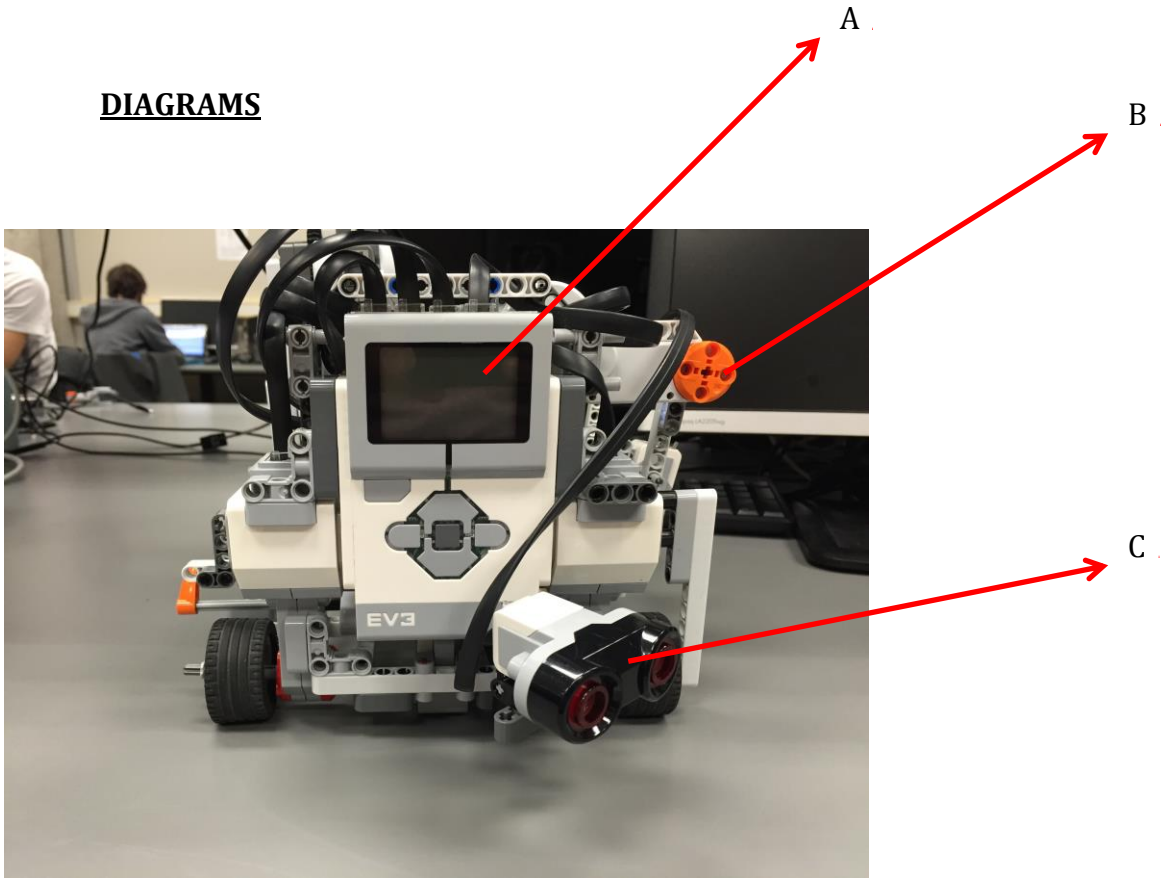
LDD DESIGN

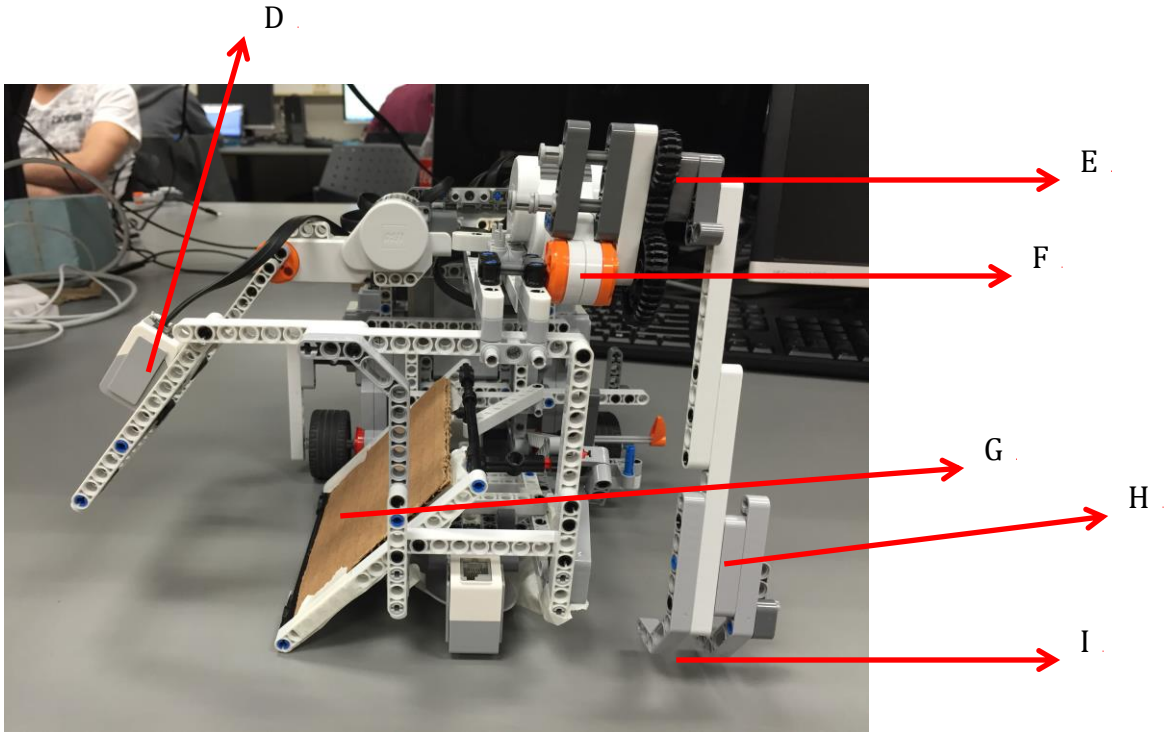


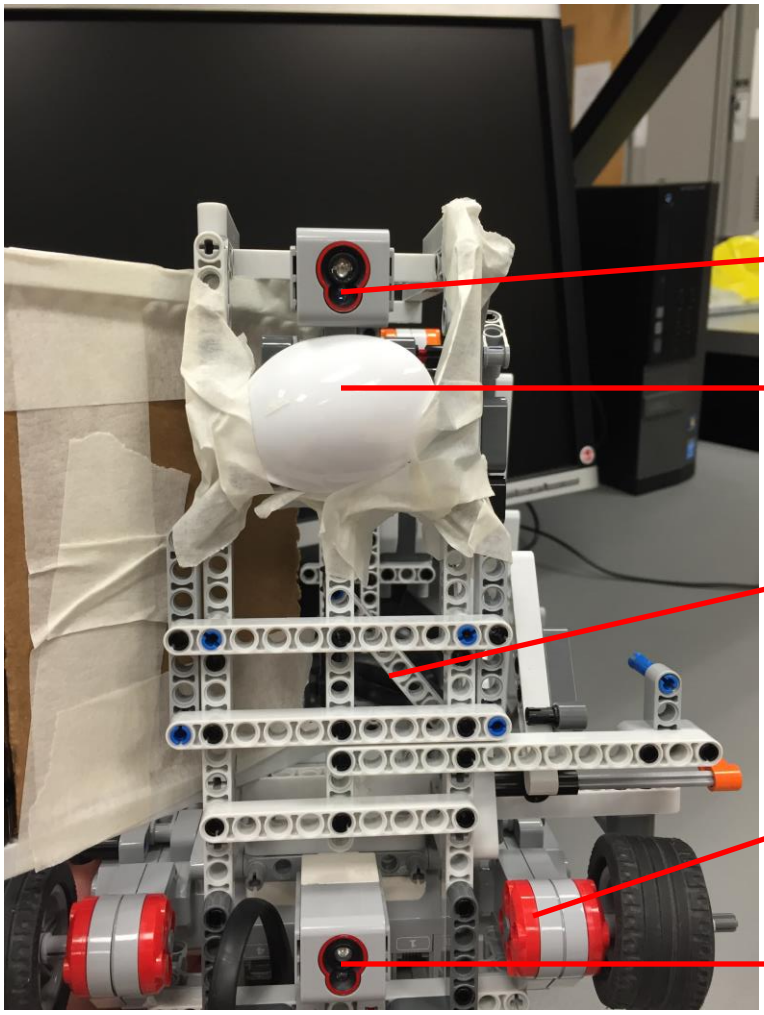


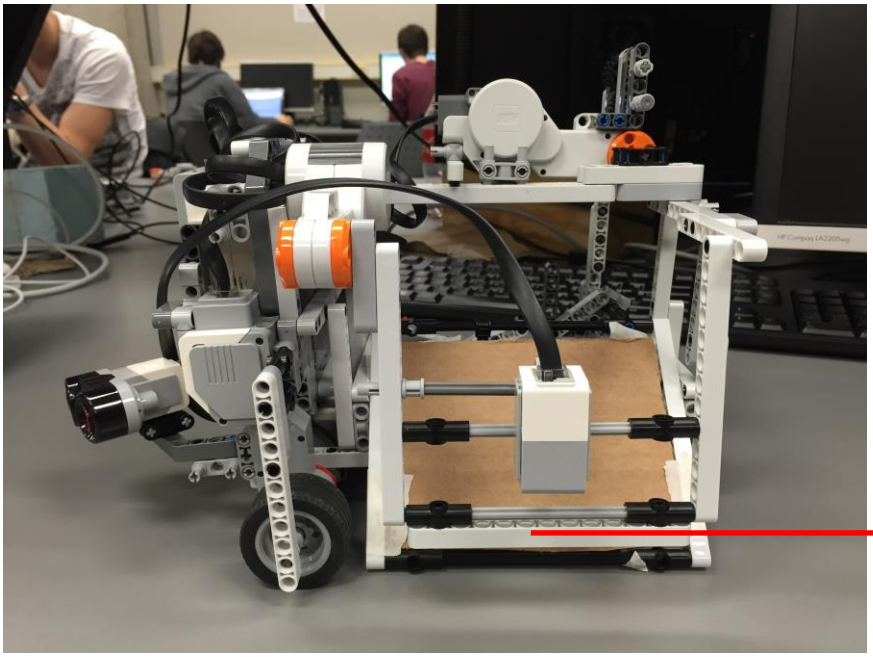


DIAGRAMS






J
K
L
M
N


O

FEATURES

A. **EV3 Brick:** Programmable with the LeJOS firmware and the Java programming language

B. **Motor #1:** Responsible for the rotation of the ball collector.

C. **Sensor #1:** Ultrasonic Sensor responsible for recognizing object shapes and estimating their distance from the robot.

D. **Sensor #2:** Light sensor responsible for verifying the correct color of the ball before collection

E. **Gears:** Responsible for stabilizing the launching arm motion as well as increasing its power.

F. **Motor #2:** Responsible for the rotation of the launching arm.

G. **Ramp:** Cardboard ramp responsible for directing the collected balls towards the funnel.

H. **Launching Arm:** Responsible for projecting the ball into the net.

I. **Scoop:** Responsible for ensuring the ball will not slip off the arm while being projected.

J. **Sensor #3:** Light sensor responsible for localization

K. **Spoon:** Responsible for increasing the contact point between the robot and the playing board

L. **Funnel:** Ensures the ball will be properly directed from the ramp to the launching area.

M. **Motor #3 & Motor #4:** One on each side of the robot responsible for the rotation of the robots wheels.

N. **Sensor #4:** Light sensor responsible for odometry correction which will be used for navigation

O. **Ball Collector:** Responsible for collecting balls from the designated area and directing them to the back of the robot (launching area).

GENERAL PROCEDURE

Offense

Using the light sensor in the front for localization and odomertry correction (used for navigation) as well as the two ultrasonic sensors (object detection/distance estimation) the robot will be able to move along the board to area where the balls will be stored. Using its collection arm, the robot will load only the correct color balls (using a light sensor for verification) onto a ramp and then into the funnel. Next, the launching area will receive one (1) ball at a time from the funnel. After the ball has landed on the scoop, the launch arm will rotate completely in order to propel the ball into the net. The robot will repeat this procedure until there are no more balls remaining in the funnel.

Defense

While on defense, the robot will extend its launching arm outwards to 180 degrees (parallel with the ground) and rotate horizontally (with respect to the net) in a random fashion. Given the large width and the random nature of the robot's defensive movements, we believe this gives us a sufficient strategy to defend against the opponent's shots.

Top

Software Design Document

Team 03
Project: Neymar
Task: Soccer - Playing Robot

Document version: v.1.0
Date: Sunday April 10th, 2016
Author: Matthew Rodin

Edit History: Matthew Rodin created the document by compiling the seven individual software design documents.

This purpose of this document is to provide a complete overview of the progression of the software design.

TABLE OF CONTENTS

Software Version 1.0

Date: 3/11/2016

Author(s): Garret Holt

Software Version: 1.0

What changed from the last version: We started off the software by integrating the labs into one project. We evaluated what worked and what didn't for each lab from each group. We then chose one from each group to use. We took Tiff and Matthew's odometry, navigation, wall follower because their navigation was good for obstacle avoidance. Odometry and wall follower are involved with navigation so we used theirs as well. Garret and Troy's localization worked well so we used theirs.

Why did you change: This is the beginning of the software for our project with the foundation being the integration of the most successful of our labs. This is a good starting point and is fundamental for the success of this project.

Problem solved: We now have code for the labs relevant to the project integrated into one project.

Current Problem (if any): We need to test the code to see that they work as an integrated unit as intended. In addition, we need to cater them for the specifics of this project as well.

Conclusion: This is a good start to the software. We will have to see if it works. If so then we will be able to advance the project further.

Software Version 2.0

Date: 3/18/2016

Author(s): Garret Holt

Software Version: 2.0

What changed from the last version: We replaced Tiff's Odometry, navigation because they were not working as intended. We replaced them with code from Nawras and Sung's labs. In addition, we added a class called sensor poller which extends thread. It is intended to continuously read the values of each of the sensors. It has return methods so that the value from any sensor can be easily accessed from any class. In addition, it is all in one class which makes it very easy and consolidates of the sensors to one class.

Why did you change: We were having problems with the integration of Tiff's code and were unsure of why it wasn't working. In addition, we thought it would be easier to use Nawras' code since he is working with me on programming and has a better understanding of his own code. We added the sensor poller out of convenience to make it much easier to read from the sensors.

Problem solved: We were able to get the labs integrated successfully with the adaptation of Nawras' code. Odometry and navigation are working successfully. We developed a good solution to reading sensors.

Current Problem (if any): Need to test and improve localization, as well as run lots of tests on the other components.

Conclusion: We redid the integration of labs with success. We are ready to start perfecting each component and to tailor the code to the specifications of the project.

Software Version 2.1

Date: 3/23/2016

Author(s): Garret Holt

Software Version: 2.1

What changed from the last version: We implemented our obstacle avoidance but it needs to be tested. We designed our navigation to cause the robot to travel around the outside of the track always hugging the wall. Although this method takes longer to travel, it significantly reduces the number of problems that we can encounter. We also improved localization and calibration. In addition, more code specific to the project. After localization odometer updates based off of starting corner. Navigation chooses closest path to offensive zone.

Why did you change: Our previous code used wallfollower which caused a lot of error especially when traveling over longer distances. We thought that our navigation was a novel idea and will be successful by reducing the amount of variation and error that could possibly cause us to go wrong.

Problem solved: Our navigation reduces the amount of error and we now have a way of obstacle avoidance that is more reliable than wallfollower.

Current Problem (if any): We need to continue to test our obstacle avoidance and have still not tested or made improvements to localization.

Conclusion: Our robot may take longer to navigate to certain areas; however, our method should reduce the possible sources of error and thus be successful in completing the objective.

Software Version 2.2

Date: 3/25/2016

Author(s): Garret Holt

Software Version: 2.2

What changed from the last version: Lot of work on offense class. Code implemented to sweep ball, launch ball, detect color of ball and then to discard or store.

Why did you change: Further progression of code.

Problem solved: Solutions to pick up, store, color identify and launch balls.

Current Problem (if any): Need to test all of these functions and need lot of work on navigation.

Conclusion: Biggest priority right now in terms of software is navigation and obstacle avoidance. All of these functions must be tested so that changes can be made accordingly.

Software Version 2.3

Date: 3/28/2016

Author(s): Garret Holt

Software Version: 2.3

What changed from the last version: We made changes to improve localization so that it was perfected and worked with our hardware. We added another function to improve the accuracy of localization. After performing ultrasonic and light localization, the robot will travel to 0,0 and face 0 degrees. Afterwards the robot will sweep in each direction so that the back light sensor detects a black line. The black line indicates the location of the y axis. If the robot stops when the light sensor is over this line the robot will be facing exactly 0 degrees as intended.

Why did you change: These changes fixed ultrasonic and light localization for our specific hardware as well as added another function that fixes the angle. This is necessary because we need a very accurate localization because with such a big game board even slight inaccuracy can have devastating effects.

Problem solved: Our localization now works very well.

Current Problem (if any): Now that all of the components work individually we need to ensure that they all work when being run together.

Conclusion: We are in a good place for the upcoming demo. Each part works individually, if they work together then our demo should work well.

Software Version 2.4

Date: 4/2/2016

Author(s): Garret Holt

Software Version: 2.4

What changed from the last version: We did a lot of work on the offense class. We developed code to sweep, launch and detect balls. The arm will sweep the ball up and hold the ball against the sensor, if it is the desired ball the arm will push further and the ball will fall into the storage bin. If it not the right color the arm will move back to the initial position and the ball will roll out and be discarded. Our code for launching is similar to lab 5. We first scoop up the ball by rotating the arm and then set a very high acceleration and speed, and launch the ball.

Why did you change: This code enables us to perform offense.

Problem solved: The code provides solutions for

- Ball pick up
- Ball storage
- Color detection
- launching

Current Problem (if any): Need to test all of these functions and need lot of work on navigation. In addition, we need to make sure that the color detection works under various lighting conditions. Also we need to experimentally determine which values and angles work best for this process.

Conclusion: Now that the offense function has been implemented our biggest priority right now in terms of software is navigation and obstacle avoidance. All of these functions must be tested so that changes can be made accordingly.

Software Version 2.5

Date: 4/5/2016

Author(s): Garret Holt

Software Version: 2.5

What changed from the last version: We implemented the wifi and defense classes successfully. The wifi class is able to correctly receive the parameters. The defense class navigates the robot to the defensive zone and it increases its width by raising the launching arm to the horizontal position. Then it patrols in front of the goal to block shooters.

Why did you change: These are specific to the project and had not yet been completed.

Problem solved: Can now remotely receive the parameters and perform defense.

Current Problem (if any): Need to run a full test to see what works. Also need to make sure that the navigation is accurate enough to be able to navigate to the desired spot to pick up the balls from the rack.

Conclusion: We are close to being complete. We are in a pretty good position, we need to run a lot of tests and assess what is functioning properly and what needs to be adjusted.

Top

Software Summary

Team 03
Project: Neymar
Task: Soccer - Playing Robot

Document version: v.1.0
Date: Tuesday March 26nd, 2016
Author: Matthew Rodin

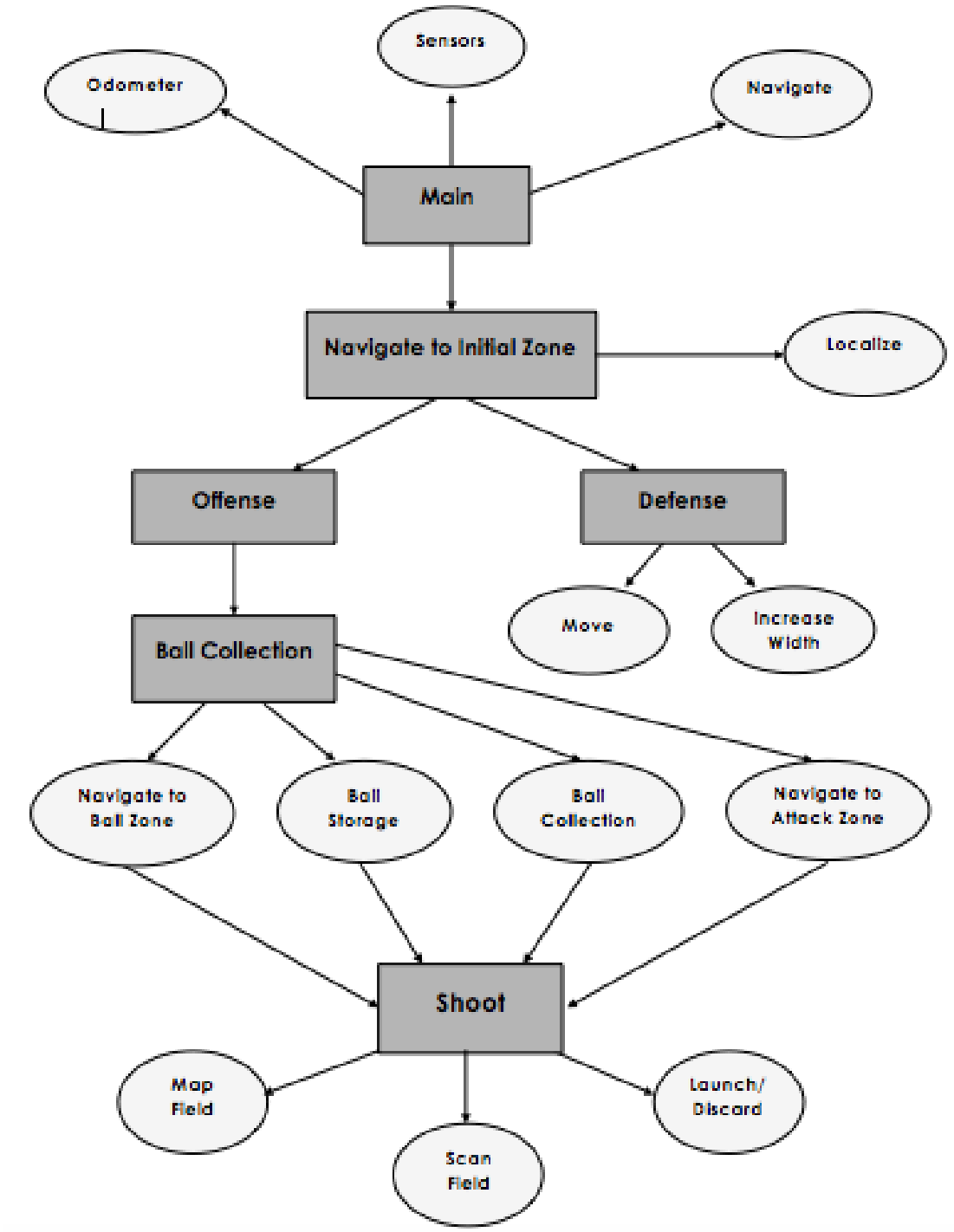
Edit History: Matthew Rodin created the document with five different sections depicting several types of software diagrams.

This purpose of this document is to provide a complete overview of the progression of the software architecture.

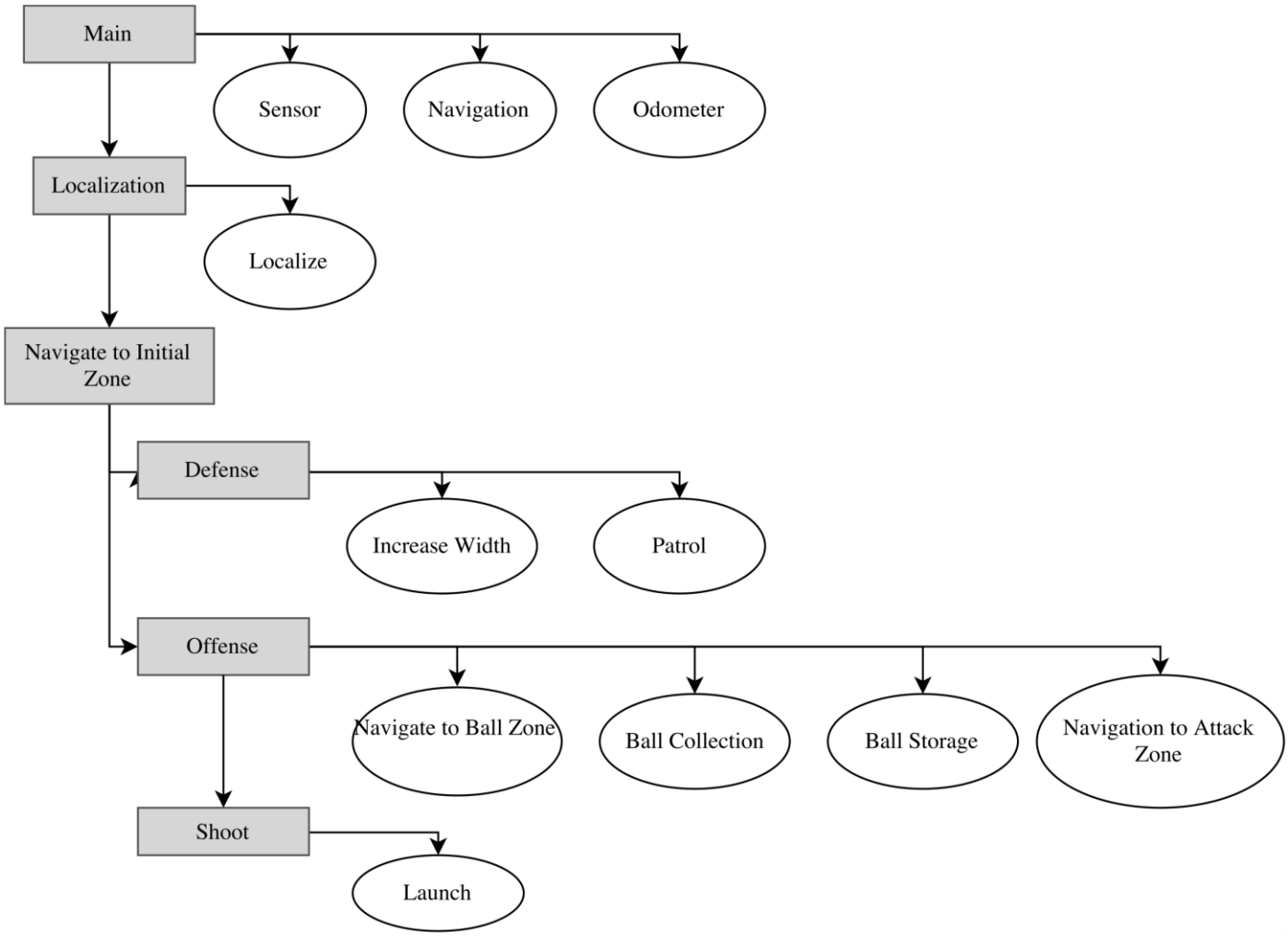
TABLE OF CONTENTS

BLOCK DIAGRAMS

BLOCK DIAGRAM (V.1.0)

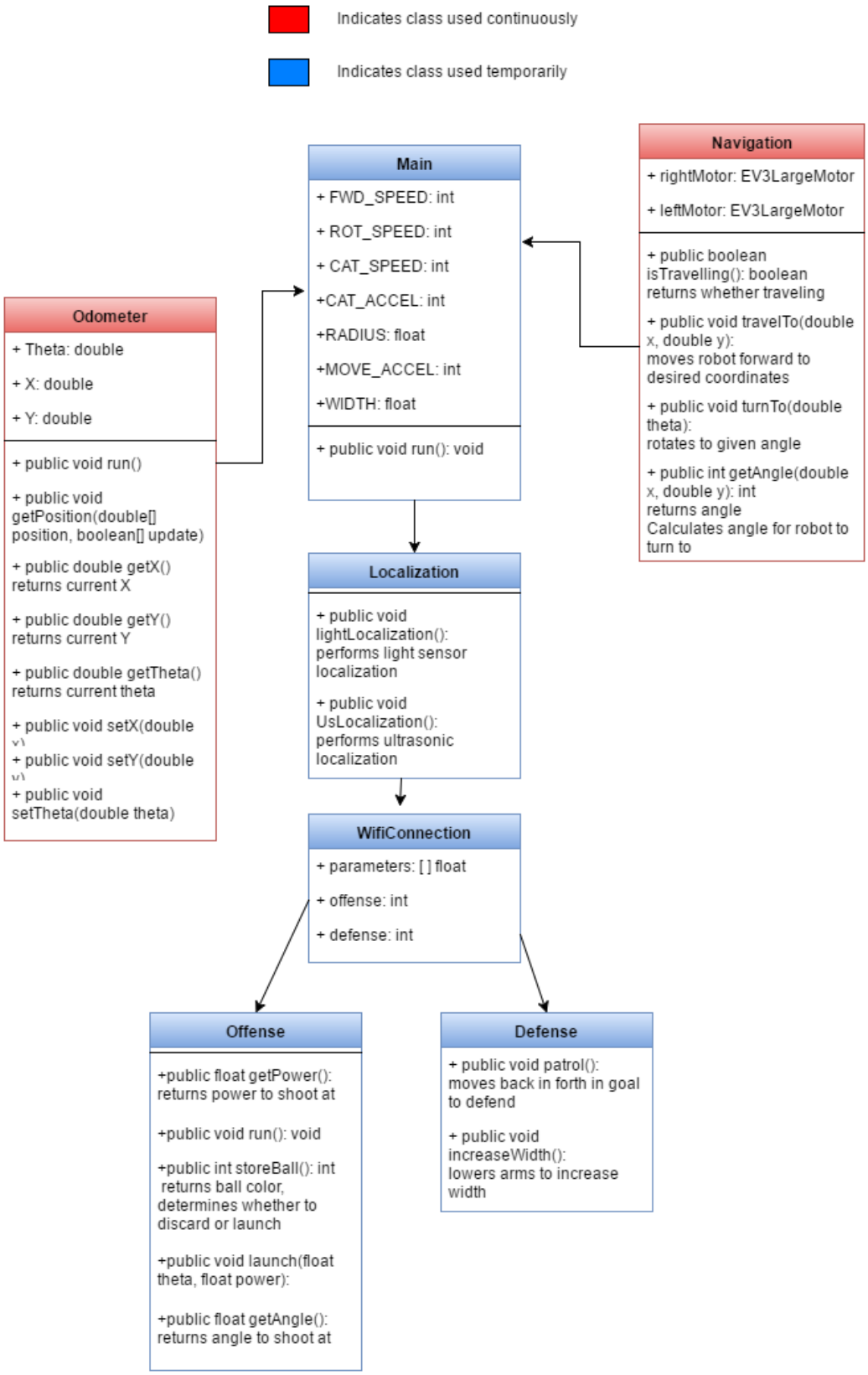


BLOCK DIAGRAM (V.1.1)



The role of a block diagram is to illustrate the principal parts or functions of the system. As well, it ensures that they are represented by blocks connected by lines that show the relationships of the blocks. In our preliminary block diagram, our goal was simply to map out the general pieces of the software architecture in order to gain a perspective on what needs to be accomplished. The square blocks represent the *major* blocks in the diagram (general concepts) while the circle blocks represent the *minor* blocks (specific concepts).

FLOW CHART

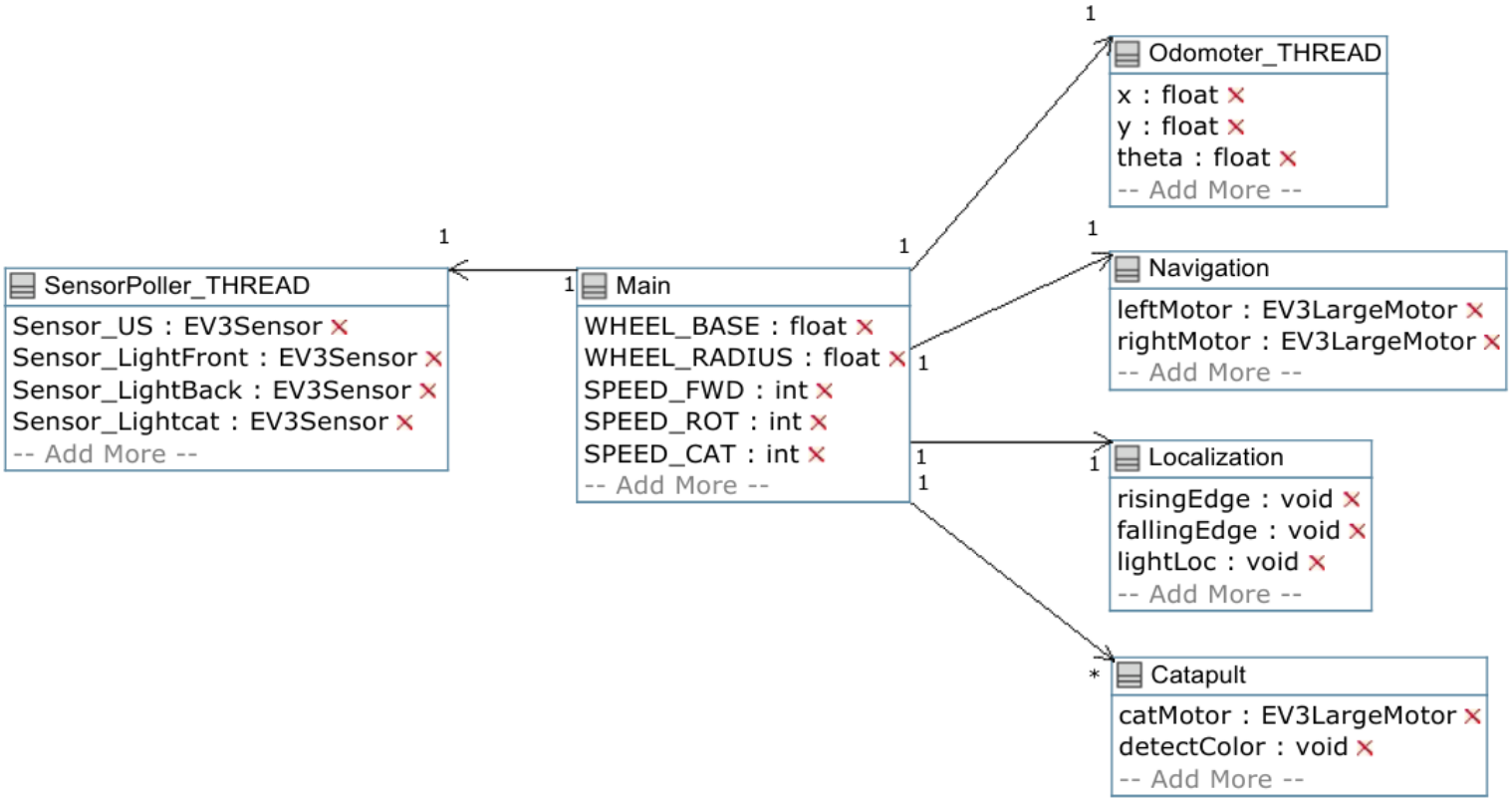

Indicates class used continuously
Indicates class used temporarily
Main
+ FWD_SPEED: int
+ ROT_SPEED: int
+ CAT_SPEED: int
+CAT_ACCEL: int
+RADIUS: float
+MOVE_ACCEL: int
+WIDTH: float
+ public void run(): void
Navigation
+ rightMotor: EV3LargeMotor
+ leftMotor: EV3LargeMotor
+ public boolean isTravelling(): boolean returns whether traveling
+ public void travelTo(double x, double y): moves robot forward to desired coordinates
+ public void turnTo(double theta): rotates to given angle
+ public int getAngle(double x, double y): int returns angle Calculates angle for robot to turn to
Odometer
+ Theta: double
+ X: double
+ Y: double
+ public void run()
+ public void getPosition(double[] position, boolean[] update)
+ public double getX() returns current X
+ public double getY() returns current Y
+ public double getTheta() returns current theta
+ public void setX(double
+ public void setY(double
+ public void setTheta(double theta)
Localization
+ public void lightLocalization(): performs light sensor localization
+ public void UsLocalization(): performs ultrasonic localization
WifiConnection
+ parameters: [ ] float
+ offense: int
+ defense: int
Offense
+public float getPower(): returns power to shoot at
+public void run(): void
+public int storeBall(): int returns ball color, determines whether to discard or launch
+public void launch(float theta, float power):
+public float getAngle(): returns angle to shoot at
Defense
+ public void patrol(): moves back in forth in goal to defend
+ public void increaseWidth(): lowers arms to increase width

A Flow Chart is a diagram that represents an algorithm, workflow or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows. In version 1.0, the goal was to elaborate on the block diagram and provide more detail about the interaction between prospective classes. Above, we have separated the classes that will only be used temporary (in blue) from the classes that will continuously be used (in red). The arrows between them provide more detail about the manner in which they interact with each other. Finally, each class contains a summary of its variables and methods (if applicable) to clarify their function.

CLASS DIAGRAMS

CLASS DIAGRAM (V.1.0)

The following figure has been created on Umple Online (link):



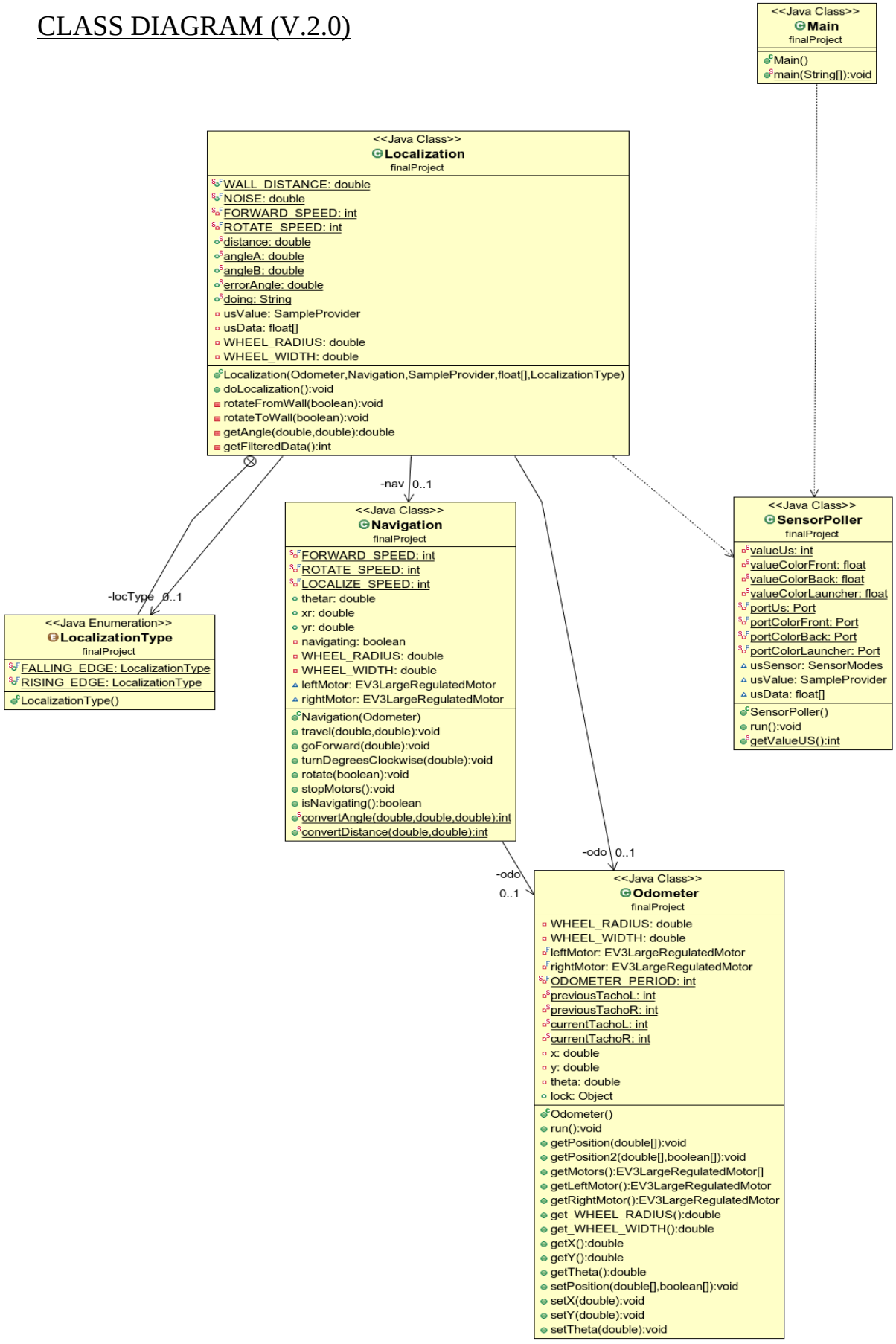
The goal of version 1.0 of the class diagram was to understand that big picture of the system's software architecture. The graphic above achieved that goal in representing the interaction between the multiple classes in the simplest of terms. The Main class will call SensorPoller, Odometer, Navigation, Localization and Catapult. The main attributes of each class, such as the X and Y position for the Odometer, are outlined in the figure. The SensorPoller and Odometer classes are threads that are synchronized by a lock. The arrows that link the classes together are called associations, and can be interpreted in the following way:

1 Main [launches →] * Catapult

1 Main [moves with →] 1 Navigation

1 Main [reads →] 1 SensorPoller

CLASS DIAGRAM (V.2.0)

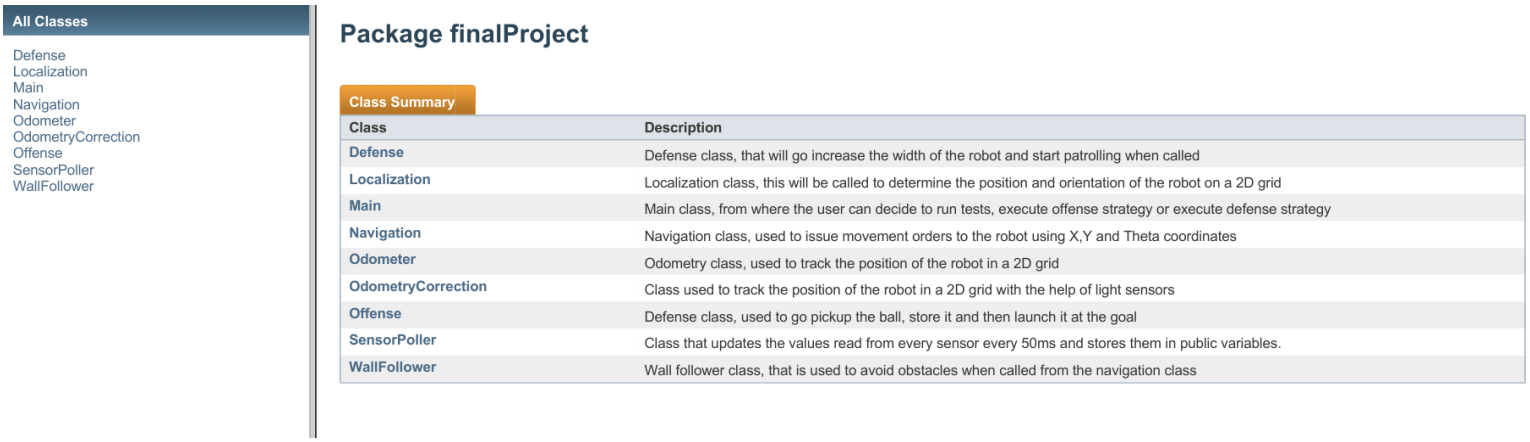


The class diagram, represented in the Unified Modeling Language (UML), is a graphic that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects. Version 2.0 of the class diagram contains far more specificity than its previous version. The structure contains six classes with specific interaction between them (represented by the direction of the connection lines). As well, each class now contains a summary of its variables as well as its methods. The idea of this diagram is to demonstrate to an outside developer how the system’s software architecture functions in terms of its classes.

JAVADOC

Javadoc is a documentation generator created to generate API documentation in HTML format from Java source code. The format used by Javadoc is the de facto industry standard for documenting Java classes.

This is a snapshot of the API index.html:



All Classes

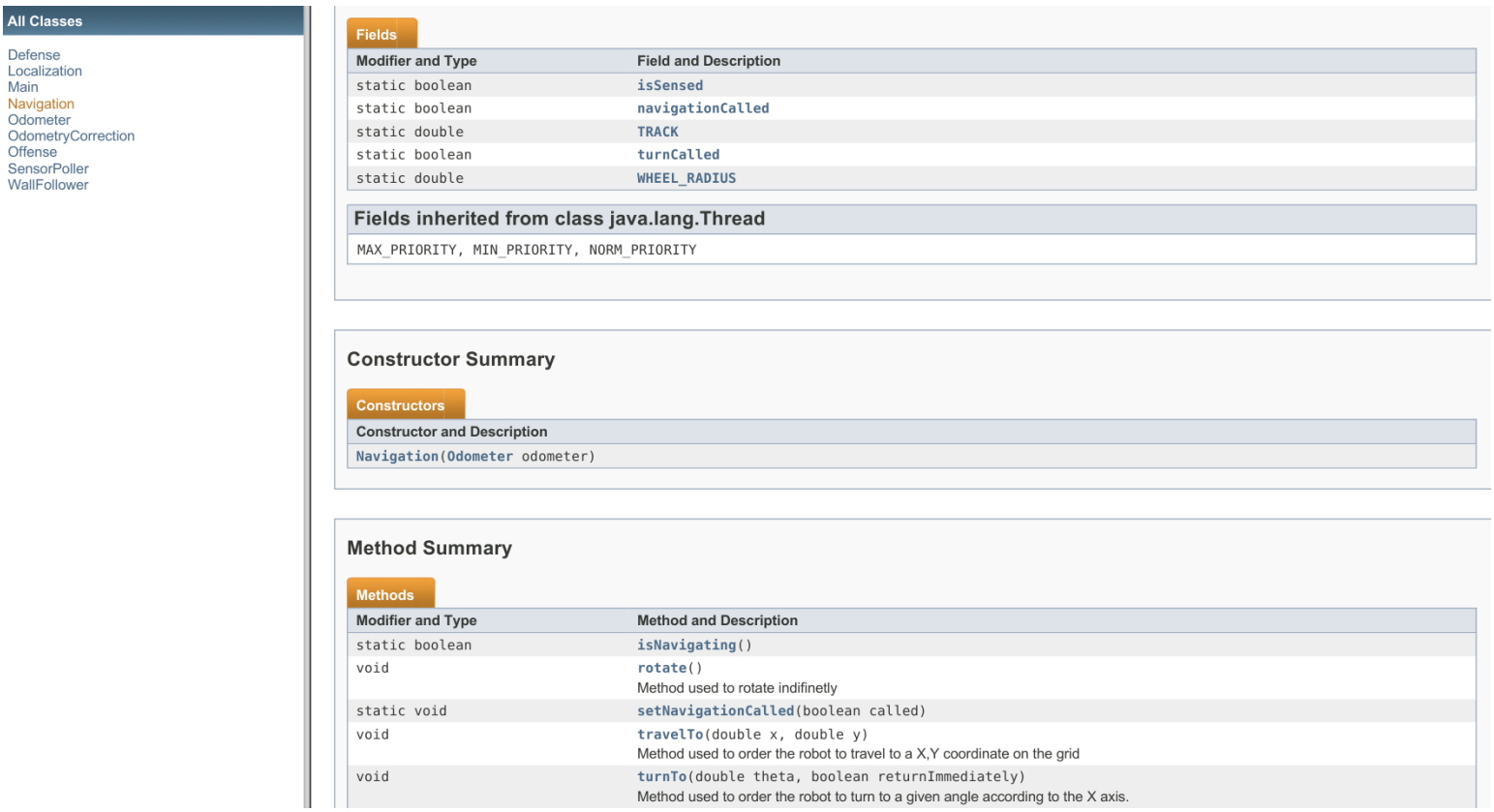
Defense
Localization
Main
Navigation
Odometer
OdometryCorrection
Offense
SensorPoller
WallFollower

Package finalProject

Class Summary

Class	Description
Defense	Defense class, that will go increase the width of the robot and start patrolling when called
Localization	Localization class, this will be called to determine the position and orientation of the robot on a 2D grid
Main	Main class, from where the user can decide to run tests, execute offense strategy or execute defense strategy
Navigation	Navigation class, used to issue movement orders to the robot using X,Y and Theta coordinates
Odometer	Odometry class, used to track the position of the robot in a 2D grid
OdometryCorrection	Class used to track the position of the robot in a 2D grid with the help of light sensors
Offense	Defense class, used to go pickup the ball, store it and then launch it at the goal
SensorPoller	Class that updates the values read from every sensor every 50ms and stores them in public variables.
WallFollower	Wall follower class, that is used to avoid obstacles when called from the navigation class

As well, below is an example of the API for a specific class (Navigation):



All Classes

Defense
Localization
Main
Navigation
Odometer
OdometryCorrection
Offense
SensorPoller
WallFollower

Fields

Modifier and Type	Field and Description
static boolean	isSensed
static boolean	navigationCalled
static double	TRACK
static boolean	turnCalled
static double	WHEEL_RADIUS

Fields inherited from class java.lang.Thread

MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY

Constructor Summary

Constructors

Constructor and Description
Navigation(Odometer odometer)

Method Summary

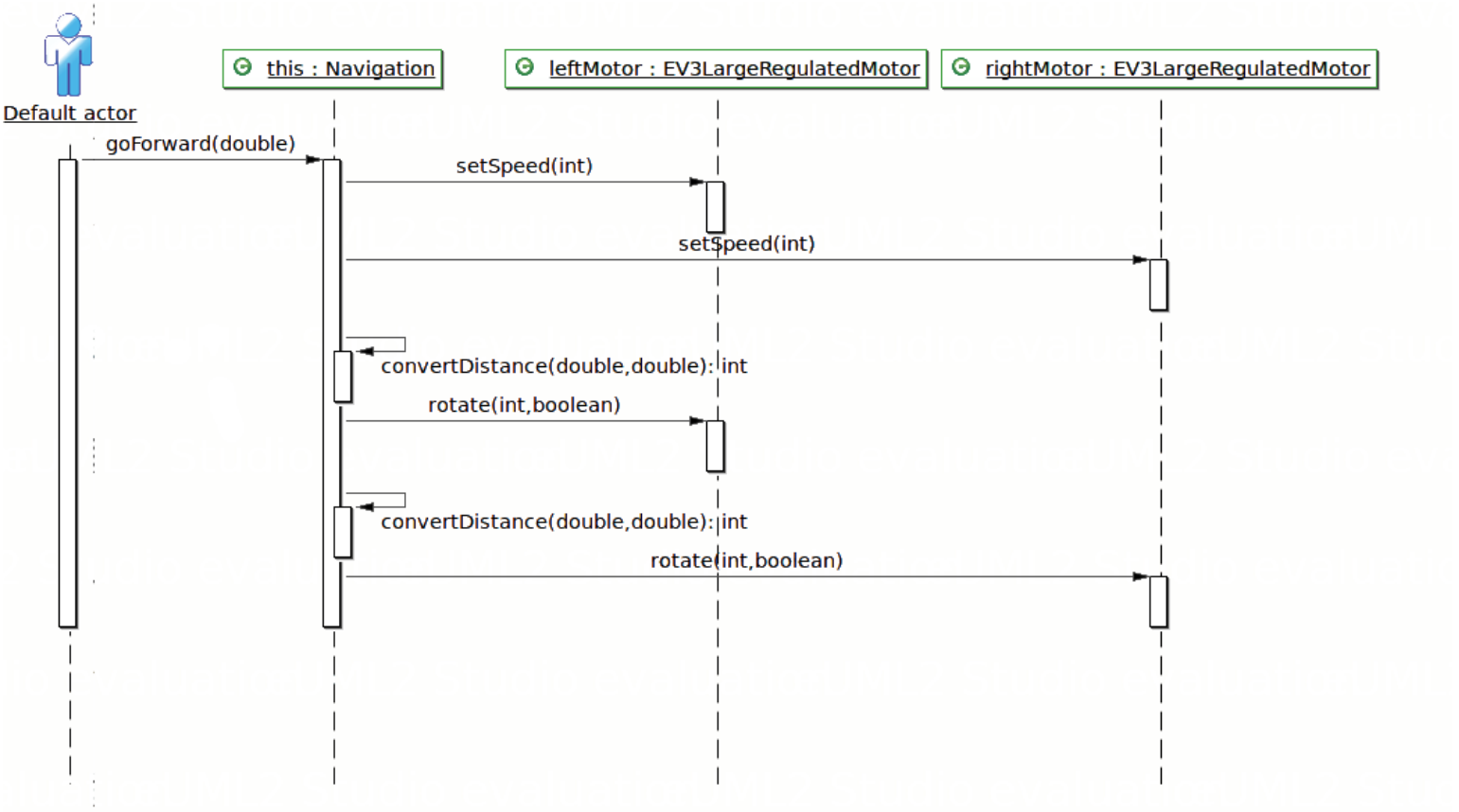
Methods

Modifier and Type	Method and Description
static boolean	isNavigating()
void	rotate() Method used to rotate indifinetly
static void	setNavigationCalled(boolean called)
void	travelTo(double x, double y) Method used to order the robot to travel to a X,Y coordinate on the grid
void	turnTo(double theta, boolean returnImmediately) Method used to order the robot to turn to a given angle according to the X axis.

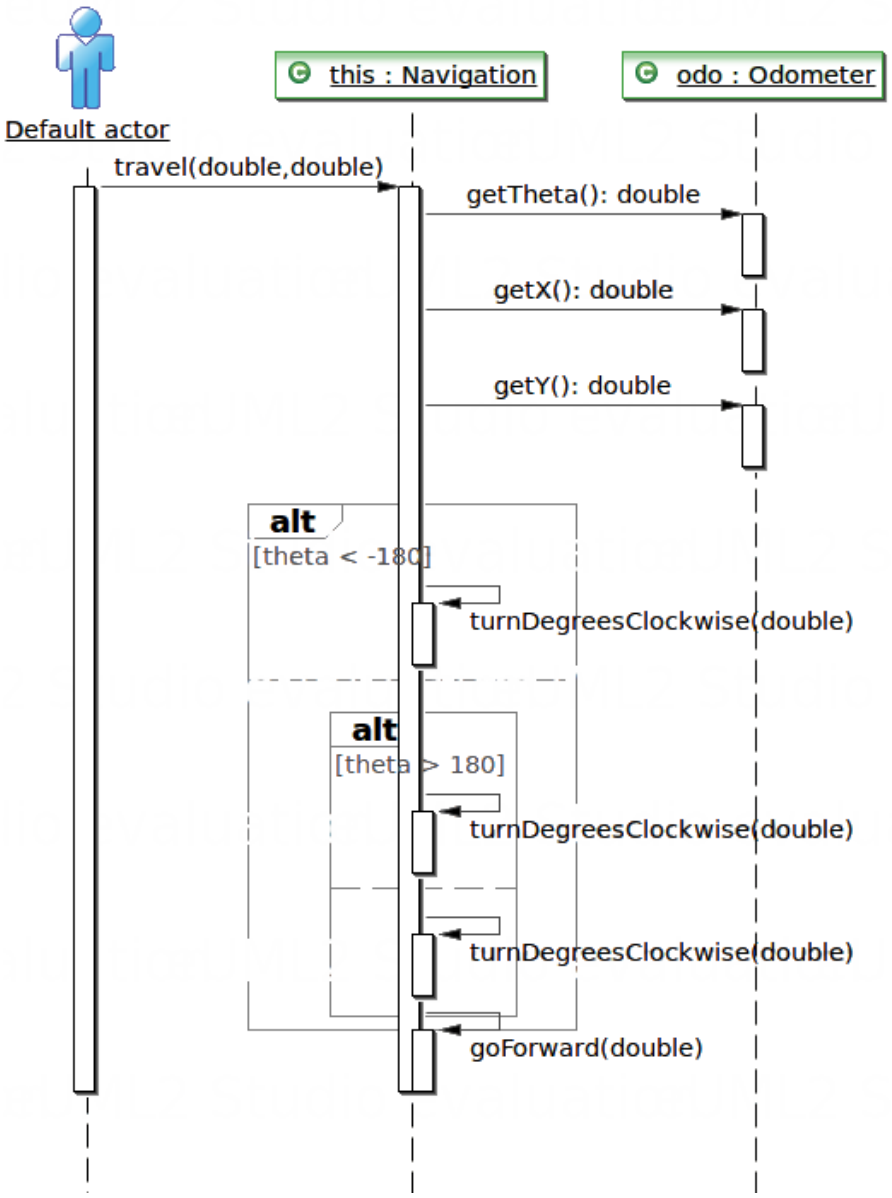
SEQUENCE DIAGRAMS

A Sequence diagram is an interaction diagram that shows how processes operate with one another and in what order. In other words, it shows object interactions arranged in time sequence. The goal of each of our sequence diagrams is to be able to represent a simple runtime scenario in graphical form in order to simply its understanding. We are able to do so by representing different processes or objects that live simultaneously with parallel vertical lines. As well, we were able to show the messages exchanged between as horizontal arrows. It is nearly impossible to create a sequence diagram for each specific process, however, here are a few examples:

GoForward()

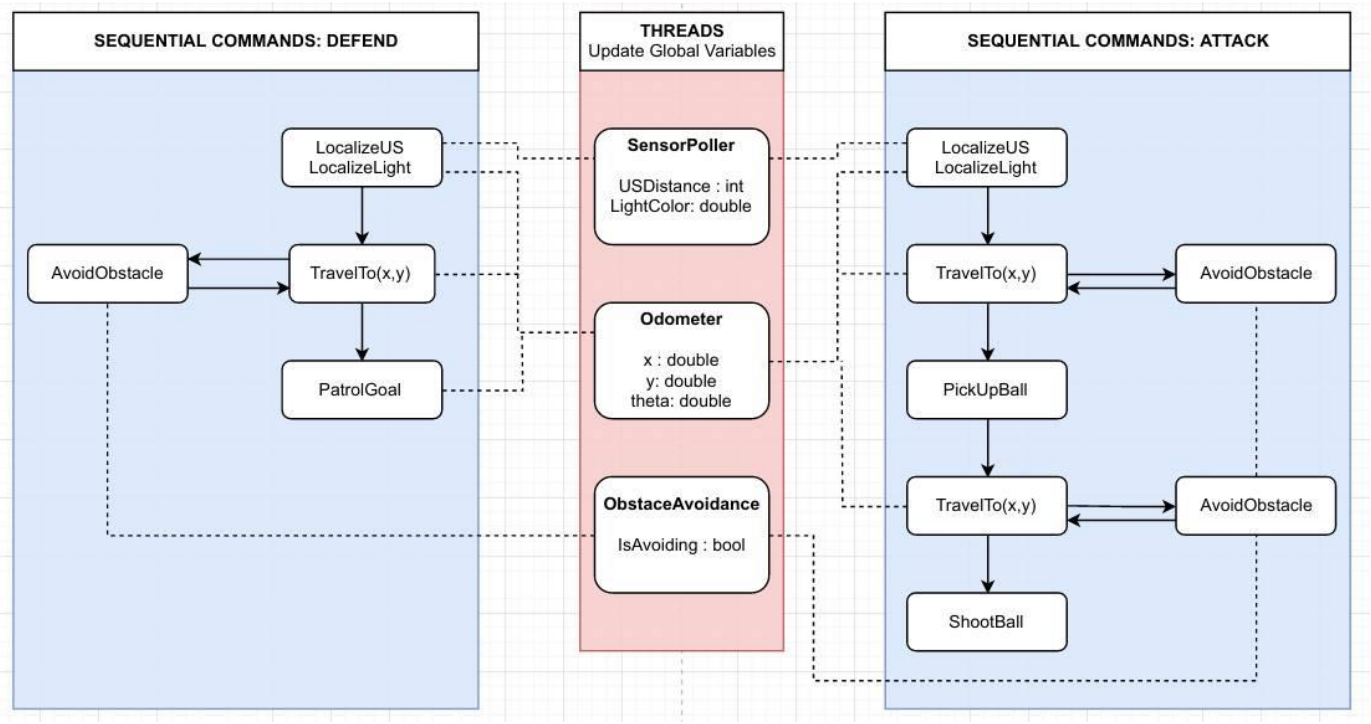


TravelTo()


Default actor
this : Navigation
odo : Odometer
travel(double,double)
getTheta(): double
getX(): double
getY(): double
alt
[theta < -180]
turnDegreesClockwise(double)
alt
[theta > 180]
turnDegreesClockwise(double)
turnDegreesClockwise(double)
goForward(double)

MULTITHREADING

Multi threading enables the software developer to write in a way where multiple activities can proceed concurrently in the same program. Each part can handle a different task at the same time making optimal use of the available resources. In blue are the sequential commands given to the robot one by one - always beginning with localization. To execute properly, these commands need to evaluate global variables that are dependent on the robot's position (x, y, theta) and sensor readings. The threads are started at the beginning of the source code and their role is to constantly update these global variables. An example of this would be localization, which constantly needs to poll the ultrasonic sensor through a thread in order to set up the robot's starting position.



Top

Testing Document

Team 03
Project: Neymar
Task: Soccer - Playing Robot

Document version: v.1.0
Date: Sunday April 3rd, 2016
Author: Matthew Rodin

Edit History: Matthew Rodin created the document to compile each individual testing document into one source file

The purpose of this document is to provide a complete overview of the testing plan as well as explain each individual test

TABLE OF CONTENTS

Test #1: Ultrasonic Sensor Full Characterization

Date: March 13th, 2016

Tester(s): Tiffany Wang

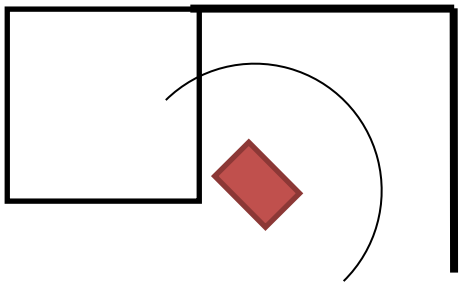
Author(s): Tiffany Wang

Hardware Version: Robot Version 2.1

Software Version: Software Version 2

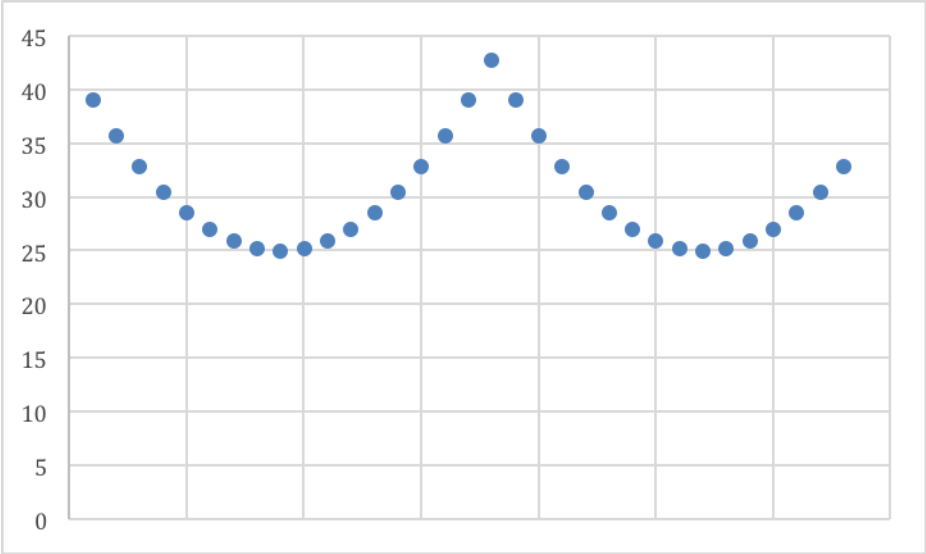
Goal: Complete characterization of US Sensor

Procedure:



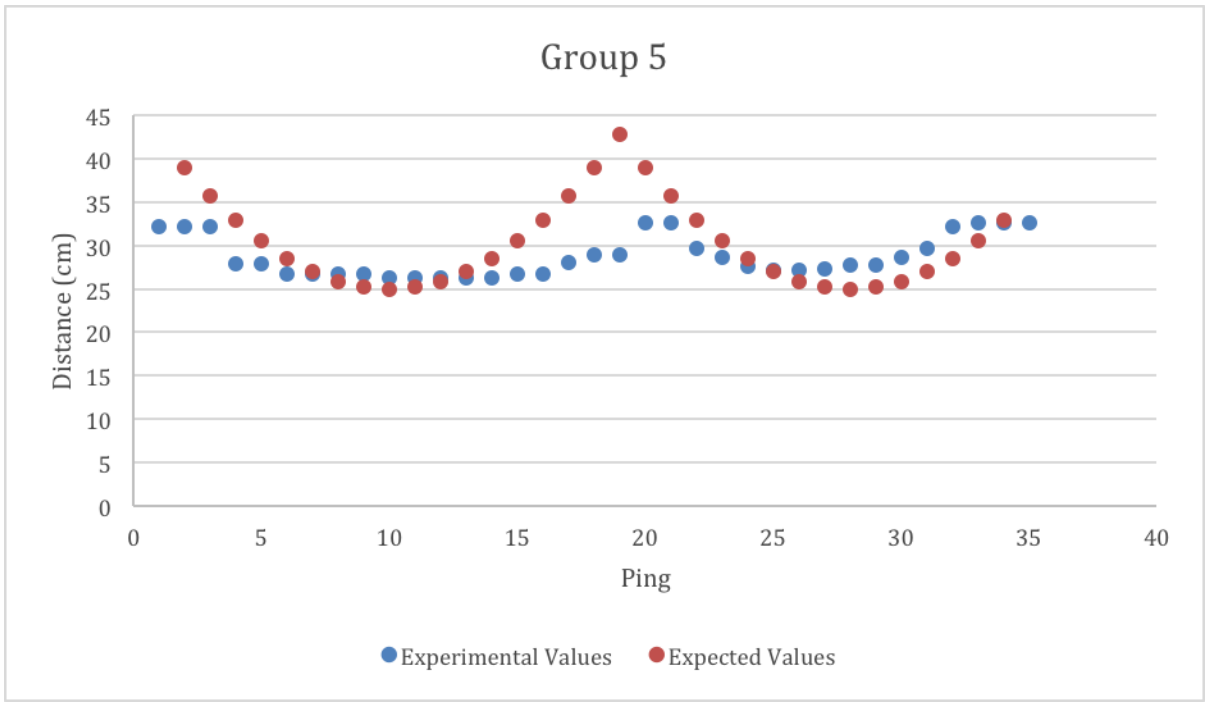
The placed in placed at the diagonally from the corner of the wall facing 45 degrees from the wall (as shown in the diagram). The robot will rotate 90 degrees while reading its distance from the wall.

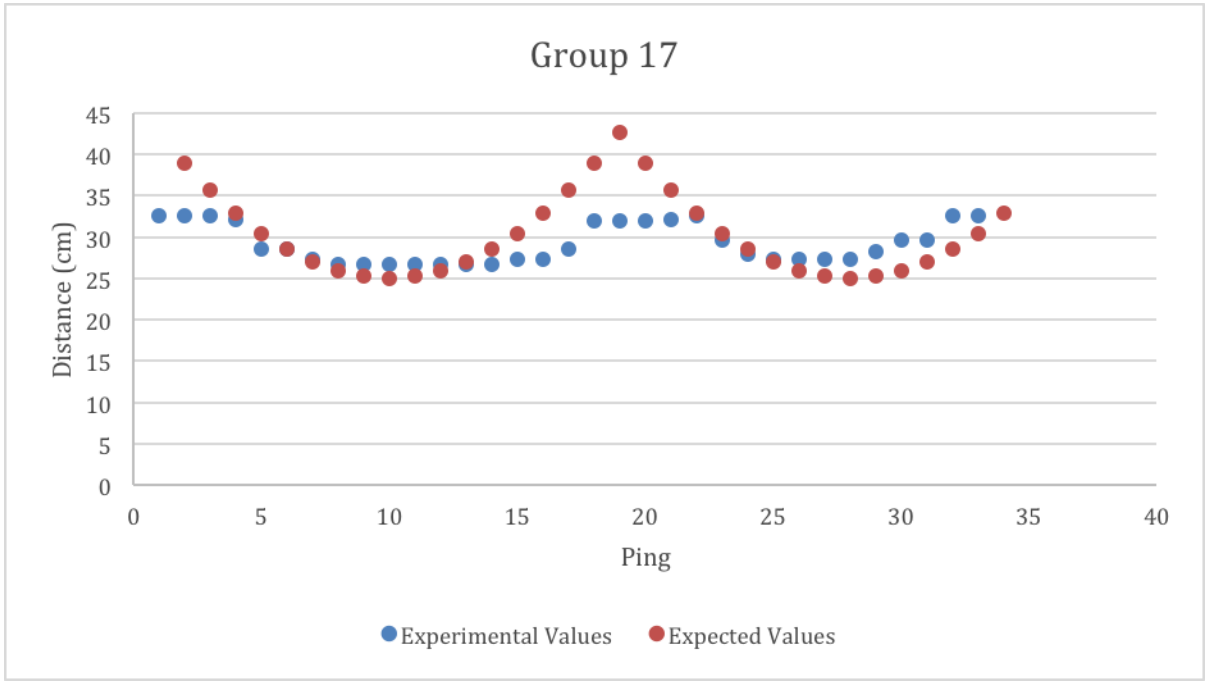
Expected results:

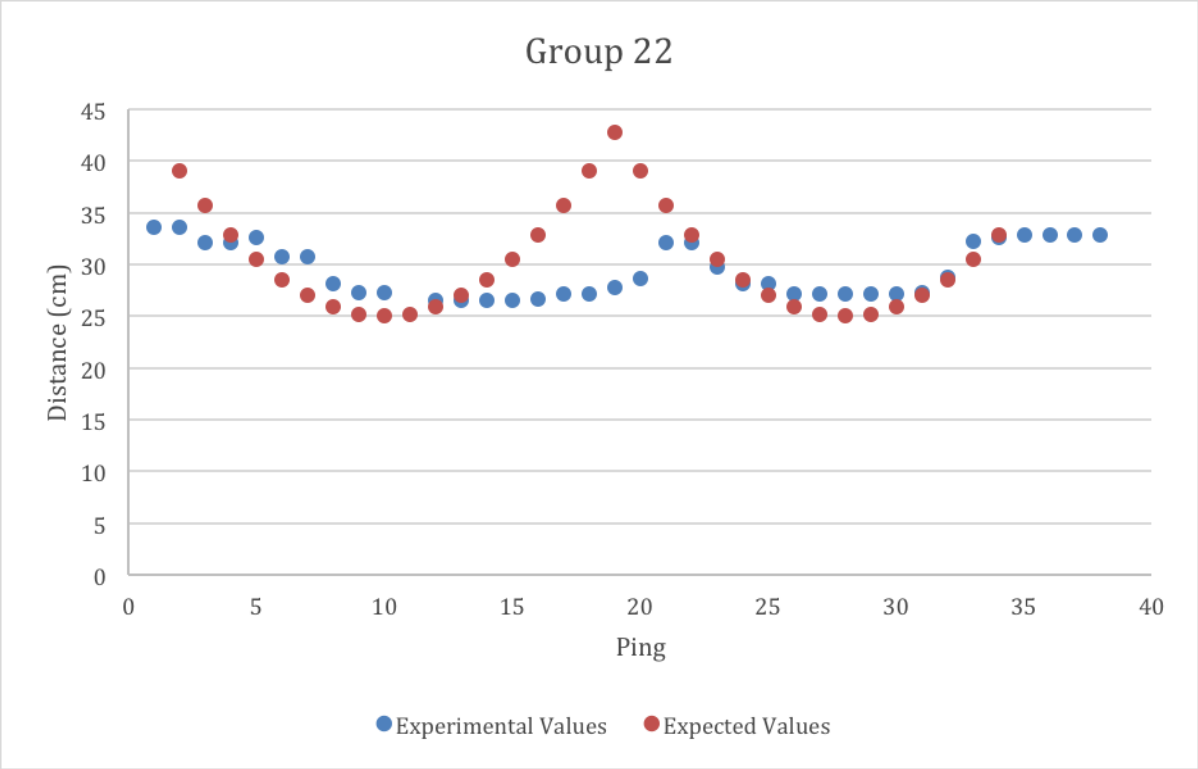


The data collected by the ultrasonic sensor should result a two perfect parabolas. The pick corresponds to the distance from the robot to the corner of the wall.

Result:







Remarks:

- There is some discontinuity in the graph.
    - false negative: really high numbers sometimes
      Group 22: the sensor detected a null when reading the wall
    - jumps
      In every graph, there's a jump when reading the corner (not smooth)

Solution: noise filter

When there is a sudden inconsistence in the reading, make sure if the robot is sensing an object or is it just a false negative.

- The graph of the two parabolas are remarkably differently
  Group 22:
  The distances to the twos walls and the motion of rotation are the same, yet the ultrasonic pings differently.

Conclusion:

The sensor of group 17 reads the most accurately.

Test #2: Light Sensor Full Characterization

Date: March 13th, 2016

Tester(s): Tiffany Wang

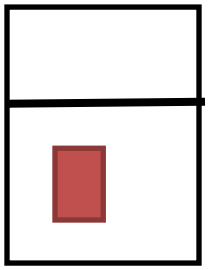
Author(s): Tiffany Wang

Hardware Version: Robot Version 2.1

Software Version: Software Version 2.0

Goal: Complete characterization of Light Sensor

Procedure:

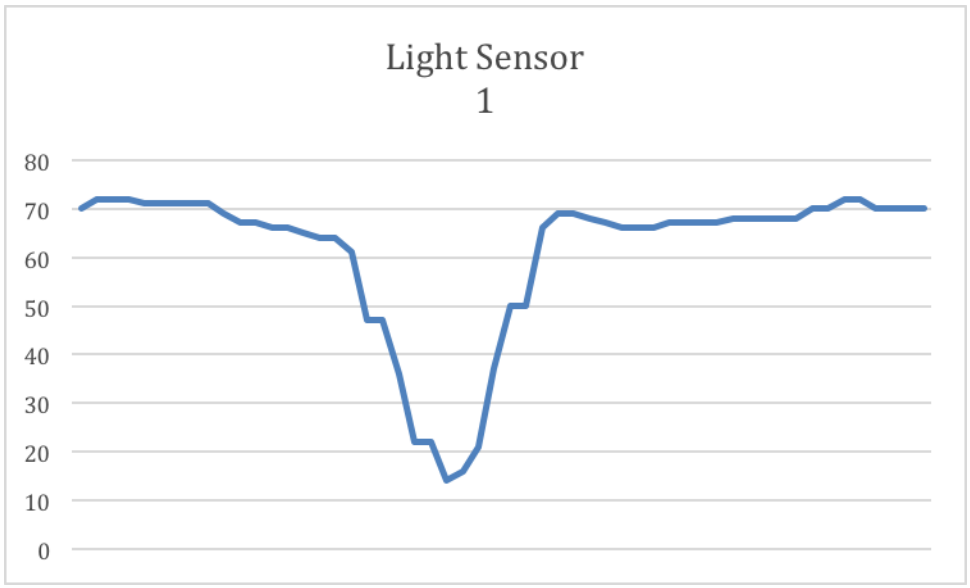


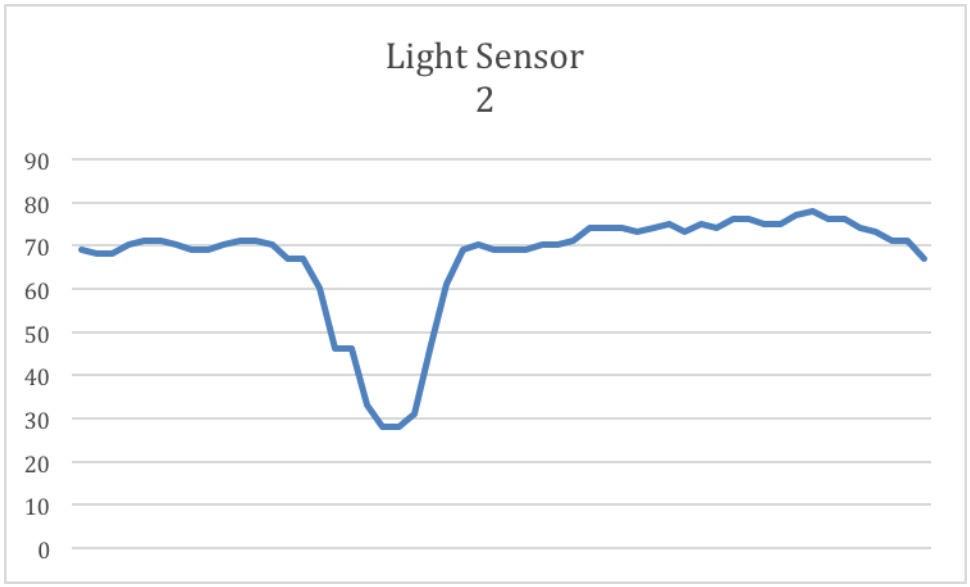
The robot is placed in one of the tile and moves forward to the next tile. By moving along, the robot retrieves data from the Light Sensor.

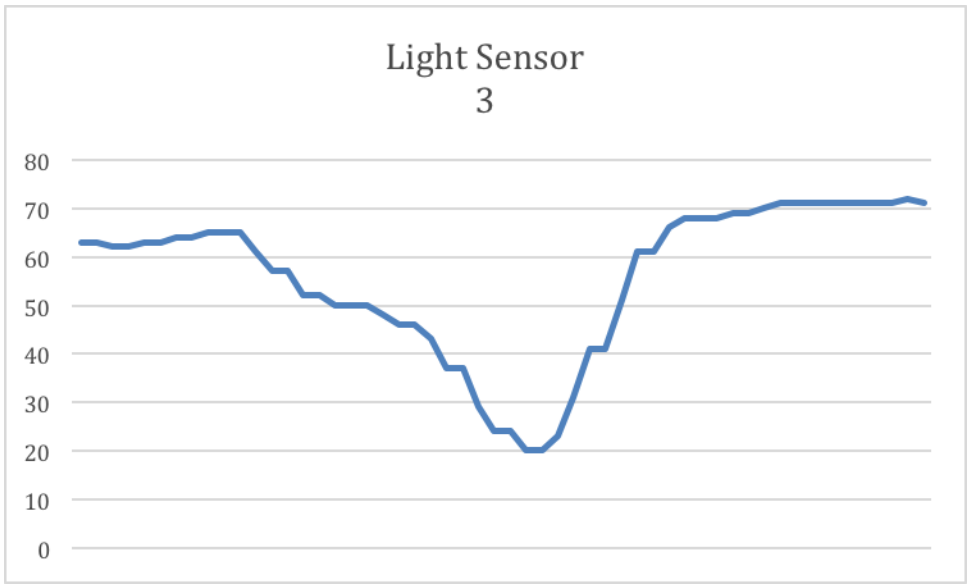
Expected results:

The data collected by the Light Sensor should result one peak downwards. The value correspond to the amount of light that is reflected back into the sensor.

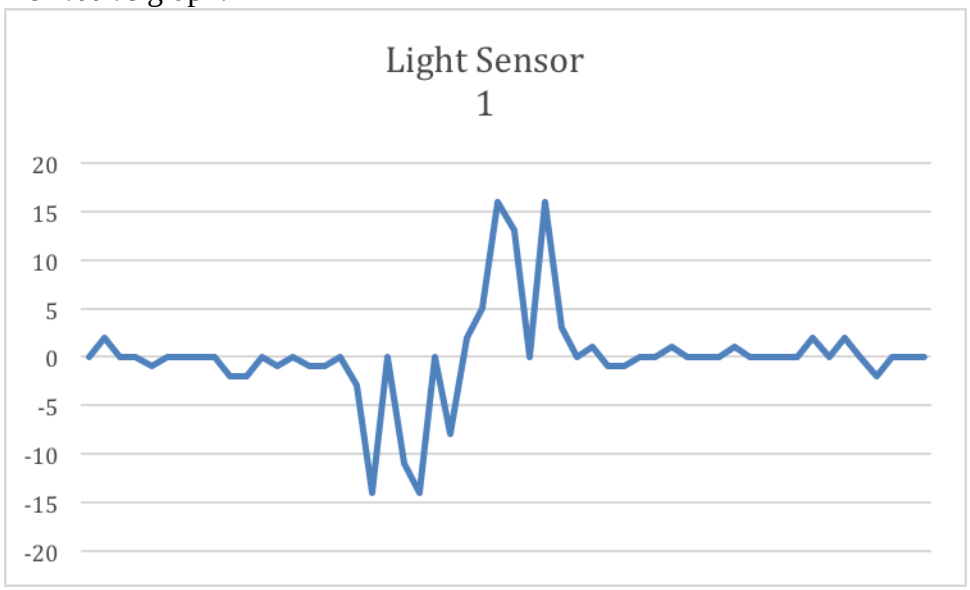
Result:

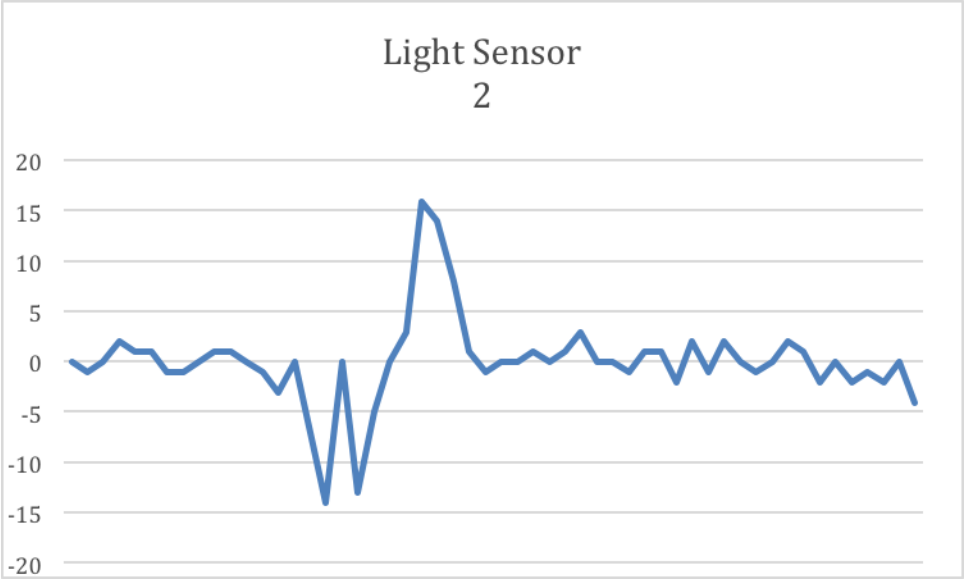


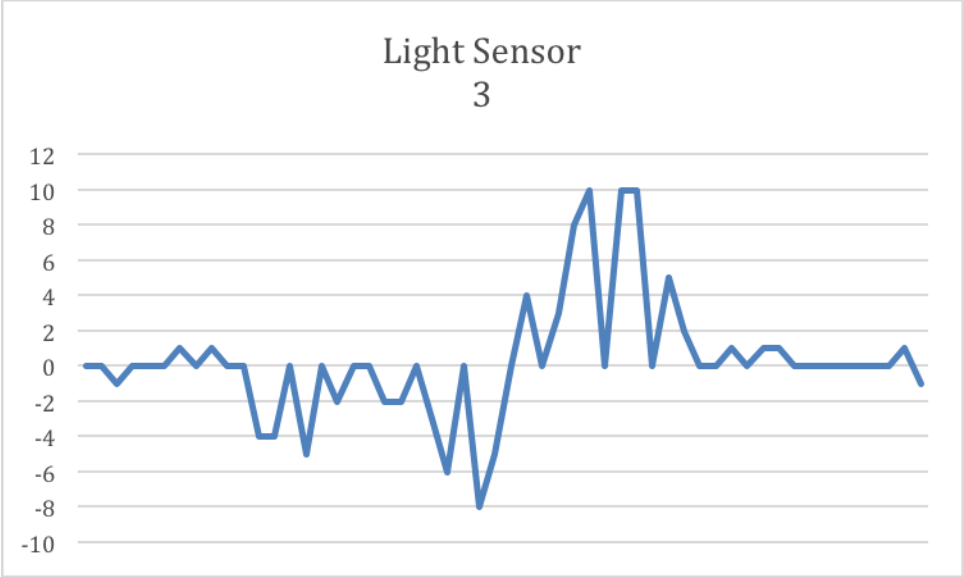




Derivative graph:







Remarks:

- The collected data did not have a smooth line. The lines were oscillating up and down (As it was detecting the same color)

Solution:

Filter by using a range within the values we get. For example if the value is greater than 60 and less than 80, value equal to 70

- The values each senor gets from the same color was different

- As the color sensor was detecting the boundary between the line and the tile, the value ranged approximately between the 70 and 20

Difference between experimental results and expected results:

Compared to the expected results, our graph shows oscillation within the values and different readings for the different light sensors, although reading the same color.

Conclusion:
The Light Sensor 1 seems to be the most accurate sensor of all 3, the values shown are reasonably consistent and the values ranged in a larger scale when comparing the retrieved values from the tile and the line. We can use the Light Sensor 1 for the localization process.

Test #3: Robot Calibration - goForward()

Date: March 17th, 2016

Tester(s): Matthew Rodin

Author(s): Matthew Rodin

Hardware Version: Robot Version 2.1

Software Version: Software Version 2

Goal: Calibrate the goForward() method.

Procedure: Make the robot travel forwards 40 cm and calculate the difference in expected distance and actual distance. Repeat 3 times.

Expected Result: The expected result would be a delta distance of 0 - meaning that the function is well calibrated.

Test Result: The difference in distance was 5 cm. The robot was not moving in a straight line (veering to the right).

Conclusion: The weight distribution of the robot was not ideal.

Action: I added weight on the right hand side of the robot to fix the weight distribution problem

Distribution: Software development, Project management, Hardware

Solution: The robot is now moving in a straight line.

Test #4: Robot Calibration - goForward()

Date: March 17th, 2016

Tester(s): Matthew Rodin

Author(s): Matthew Rodin

Hardware Version: Robot Version 2.1

Software Version: Software Version 2

Goal: Calibrate the goForward() method.

Procedure: Make the robot travel forwards 40 cm and calculate the difference in expected distance and actual distance. Repeat 3 times.

Expected Result: The expected result would be a delta distance of 0 - meaning that the function is well calibrated.

Test Result: The difference in distance was 2 cm. The robot was not moving in a straight line (veering to the right).

Conclusion: We need to create a correction constant.

Action: I set the DISTANCE_CORRECTION constant to 3.5 in order to correct the angle.

Distribution: Software development, Project management

Solution: After many trials, the DISTANCE_CORRECTION variable was set to 2 in order to properly calibrate the goForward() method.

Test #5: Robot Calibration - goForward()

Date: March 20th, 2016

Tester(s): Matthew Rodin

Author(s): Matthew Rodin

Hardware Version: Robot Version 2.2

Software Version: Software Version 2.0

Goal: Calibrate the goForward() method.

Procedure: Make the robot travel forwards 40 cm and calculate the difference in expected distance and actual distance. Repeat 3 times.

Expected Result: The expected result would be a delta distance of 0 - meaning that the function is well calibrated.

Test Result: The difference in distance was 0 cm centimeters. The robot was moving

Conclusion: The goForward() method is properly calibrated

Action: No actions taken

Distribution: Software development, Project management

Solution: N/A

Test #6: Robot Calibration - goForward()

Date: March 23rd, 2016

Tester: Tiffany Wang

Author(s): Tiffany Wang

Hardware Version: v.3.0

Software Version: v.2.1 (with WHEEL_WIDTH = 16.9 and WHEEL_RADIUS = 2.07)

Goal: Calibrate the robot so move straight.

Procedure: Align the robot with a black line on the board and make the robot traverse the fours boards. Observe if it deviates from its straight path.

Expected Result: After travelling through the four boards, it does not diverge away from the black line.

Test Result:

	physical x position (cm)
1	10.9
2	12.2
3	9.7
4	8.6
5	10.7
6	14.2
7	12
8	10.7
9	11.1
10	11.7

Average $= \frac{\sum_{k=1}^{10} angle_k}{10} = 11.80$

Standard Deviation = $\sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 1.5$

where x_i is each value of the data, and μ is the median.

Conclusion: The final position of the robot is on average 11.80 cm to the right of the black line and this error is rather consistent as the standard deviation is small (1.5). This means that the robot's weight is unevenly distributed: in this case, the right side of the robot is too heavy in comparison to the left side.

Action: Add more weight onto the left side.
Adjust the counterweight using clay after each run until the result is satisfying.

Distribution: Calibration, Mechanical Design

Test #7: Robot Calibration - goForward()

Date: March 23rd, 2016

Tester: Tiffany Wang

Author(s): Tiffany Wang

Hardware Version: v.3.0

Software Version: v.2.1 (with WHEEL_WIDTH = 16.9 and WHEEL_RADIUS = 2.07)

Goal: Calibrate the robot so move straight.

Procedure: After the calibration test taken for Calibration1 – goFoward(), I have run the robot multiple times with varying amount of clay as counterweight.

Align the robot with a black line on the board and make the robot traverse the fours boards. Observe if it deviates from its straight path.

Expected Result: After travelling through the four boards, it does not diverge away from the black line.

Test Result:

	physical x position (cm)
1	0.5
2	0
3	0.2
4	-0.4
5	0
6	0.3
7	0.6
8	1
9	0.5
10	0.2

Average $= \frac{\sum_{k=1}^{10} angle_k}{10} = 0.29$

Standard Deviation = $\sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 0.387$

where x_i is each value of the data, and μ is the median.

Conclusion: The final position of the robot is on average 0.29 cm to the right of the black

line and this result is rather consistent as the standard deviation is small (0.387). This result is satisfying as the error is considerably minimal on such a large scaled navigation. The robot is calibrated.

Action: the robot is calibrated: no further action needed.

Distribution: Calibration, Mechanical Design

Test #8: Robot Calibration - travelTo()

Date: March 17th, 2016

Tester(s): Matthew Rodin

Author(s): Matthew Rodin

Hardware Version: Robot Version 2.1

Software Version: Software Version 2

Goal: Calibrate the travelTo() method.

Procedure: Start the robot at (0,0) and have it travel to each of the following point individually: (60.96, 60.96), (-60.96, 60.96), (60.96, -60.96), (-60.96, -60.96). Then, calculate the difference in expected (x, y) coordinates and actual (x, y) coordinates for each case. Repeat the procedure 3 times for each case.

Expected Result: The expected result would be a delta x and delta y of 0 for each case - meaning that the method is well calibrated.

Test Result:
The difference for (60.96, 60.96) = (5.5, 4.0)
The difference for (-60.96, 60.96) = (-6.0, 3.0)
The difference for (60.96, -60.96) = (4.5, -5.0)
The difference for (-60.96, -60.96) = (-5.5, -5.5)

Conclusion: We need to create a correction constant.

Action: I set the ANGLE_CORRECTION constant to 1.5 in order to correct the angle the robot turns to (which will ultimately fix the accuracy of the travelTo method).

Distribution: Software development, Project management

Solution: After many trials, the ANGLE_CORRECTION variable was set to 0.7 in order to properly calibrate the travelTo() method.

Test #9: Robot Calibration - travelTo()

Date: March 20th, 2016

Tester(s): Matthew Rodin

Author(s): Matthew Rodin

Hardware Version: Robot Version 2.2

Software Version: Software Version 2

Goal: Calibrate the travelTo() method.

Procedure: Start the robot at (0,0) and have it travel to each of the following point individually: (60.96, 60.96), (-60.96, 60.96), (60.96, -60.96), (-60.96, -60.96). Then, calculate the difference in expected (x, y) coordinates and actual (x, y) coordinates for each case. Repeat the procedure 3 times for each case.

Expected Result: The expected result would be a delta x and delta y of 0 for each case - meaning that the method is well calibrated.

Test Result:
The difference for (60.96, 60.96) = (0.5, 0.0)
The difference for (-60.96, 60.96) = (-1.0, 0.0)
The difference for (60.96, -60.96) = (1.0, 0.0)
The difference for (-60.96, -60.96) = (-1.0, 0.0)

Conclusion: The travelTo() method is properly calibrated.

Action: No actions taken.

Distribution: Software development, Project management

Solution: N/A

Test #10: Robot Calibration - turnDegreesClockwise()

Date: March 17th, 2016

Tester(s): Matthew Rodin

Author(s): Matthew Rodin

Hardware Version: Robot Version 2.1

Software Version: Software Version 2

Goal: Calibrate the turnDegreesClockwise() method.

Procedure: Rotate robot 1800 degrees (5 turns) and calculate the difference in expected angle and actual angle. Repeat 3 times.

Expected Result: The expected result would be a delta angle of 0 - meaning that the function is well calibrated.

Test Result: The difference in angle was 25 degrees.

Conclusion: We need to create a correction constant.

Action: I set the WHEEL_WIDTH_CORRECTION constant to 2.0 in order to correct the angle.

Distribution: Software development, Project management

Solution: After many trials, the WHEEL_WIDTH_CORRECTION variable was set to 1.36 in order to properly calibrate the turnDegreesClockwise() method.

Test #11: Robot Calibration - turnDegreesClockwise()

Date: March 20th, 2016

Tester(s): Matthew Rodin

Author(s): Matthew Rodin

Hardware Version: Robot Version 2.2

Software Version: Software Version 2

Goal: Calibrate the turnDegreesClockwise() method.

Procedure: Rotate robot 1800 degrees (5 turns) and calculate the difference in expected angle and actual angle. Repeat 3 times.

Expected Result: The expected result would be a delta angle of 0 - meaning that the function is well calibrated.

Test Result: The difference in angle was 20 degrees.

Conclusion: We need to increase WHEEL_WIDTH constant.

Action: I increased WHEEL_WIDTH from 17.1 to 17.4

Distribution: Software development, Project management

Solution: After many trials, the WHEEL_WIDTH variable was set to 17.6 in order to properly calibrate the turnDegreesClockwise() method.

Test #12: Robot Calibration – 2 Revolutions

Date: March 23rd, 2016

Tester: Tiffany Wang

Author(s): Tiffany Wang

Hardware Version: v.3.0

Software Version: v.2.1 (with WHEEL_WIDTH = 17.4 and WHEEL_RADIUS = 2.07)

Goal: Calibrate the robot so it can rotate properly without shifting from its center

Procedure: Make the robot turn two full revolutions (720 degrees) without shifting away from its center of rotation

Expected Result: After two full revolutions, the robot returns to its original position.

Test Result:

	Angles away from 0
1	27
2	30
3	20
4	38
5	38
6	28
7	41
8	32
9	36
10	23

Average $= \frac{\sum_{k=1}^{10} angle_k}{10} = 31$

Standard Deviation = $\sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 6.9$

where x_i is each value of the data, and μ is the median.

Conclusion: The robot turns 31 degrees over what is expected. The robot is not calibrated.

Action: Since the calculation of *angle rotate* is:

```
public void rotateDegreeCW(int degree){
```

```
            leftMotor.rotate(convertAngle(WHEEL_RADIUS, WHEEL_WIDTH,
degree), true);
            rightMotor.rotate(-convertAngle(WHEEL_RADIUS, WHEEL_WIDTH,
degree), false);
```

where

```
    public static int convertAngle(double radius, double width, double angle) {
        return convertDistance(radius, Math.PI * width * angle / 360.0);
    }

    public static int convertDistance(double radius, double distance) {
        return (int) ((180.0 * distance) / (Math.PI * radius));
    }
```

The rotated angle depends on the WHEEL_RADIUS and a WHEEL_WIDTH.
Since varying the WHEEL_RADIUS would change a lot of factors in other methods, changing the WHEEL_WIDTH would be the solution.
We want the robot to rotate less. Therefore, we reduce the WHEEL_WIDTH from 17.4 to 16.8.

Distribution: Calibration, Mechanical Design, Software

Test #13: Robot Calibration – 2 Revolutions

Date: March 23rd, 2016

Tester: Tiffany Wang

Author(s): Tiffany Wang

Hardware Version: v.3.0

Software Version: v.2.1 (with WHEEL_WIDTH = 16.8 and WHEEL_RADIUS = 2.07)

Goal: Calibrate the robot so it can rotate properly without shifting from its center

Procedure: Make the robot turn two full revolutions (720 degrees) without shifting away from its center of rotation

Expected Result: After two full revolutions, the robot returns to its original position.

Test Result:

	Angles away from 0
1	-6
2	-9
3	-2
4	-5
5	-7
6	-6
7	-7
8	-8
9	-3
10	-4

Average $= \frac{\sum_{k=1}^{10} angle_k}{10} = -5.65$

Standard Deviation = $\sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 2.21$

where x_i is each value of the data, and μ is the median.

Conclusion: The robot turns on average 5.65 degrees under what is expected. The robot is not calibrated.

Action: Since the calculation of *angle rotate* is :

```
public void rotateDegreeCW(int degree){
```

```
        leftMotor.rotate(convertAngle(WHEEL_RADIUS, WHEEL_WIDTH,
degree), true);
        rightMotor.rotate(-convertAngle(WHEEL_RADIUS, WHEEL_WIDTH,
degree), false);
```

where

```
    public static int convertAngle(double radius, double width, double angle) {
        return convertDistance(radius, Math.PI * width * angle / 360.0);
    }

    public static int convertDistance(double radius, double distance) {
        return (int) ((180.0 * distance) / (Math.PI * radius));
    }
```

The rotated angle depends on the WHEEL_RADIUS and a WHEEL_WIDTH.
Since varying the WHEEL_RADIUS would change a lot of factors in other methods, changing the WHEEL_WIDTH would be the solution.
We want the robot to rotate less. Therefore, we reduce the WHEEL_WIDTH from 16.8 to 16.9

Distribution: Calibration, Mechanical Design, Software

Test #14: Robot Calibration – 2 Revolutions

Date: March 23rd, 2016

Tester: Tiffany Wang

Author(s): Tiffany Wang

Hardware Version: v.3.0

Software Version: v.2.1 (with WHEEL_WIDTH = 16.9 and WHEEL_RADIUS = 2.07)

Goal: Calibrate the robot so it can rotate properly without shifting from its center

Procedure: Make the robot turn two full revolutions (720 degrees) without shifting away from its center of rotation

Expected Result: After two full revolutions, the robot returns to its original position.

Test Result:

	Angles away from 0
1	1
2	-1
3	0
4	0
5	0
6	1
7	0
8	1
9	0
10	1

Average $= \frac{\sum_{k=1}^{10} angle_k}{10} = 0.30$

Standard Deviation = $\sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 0.67$

where x_i is each value of the data, and μ is the median.

Conclusion: The robot turns on average 0.30 degrees over what is expected. This is an expectable error. The robot is now calibrated.

Action: The robot is calibrated, no action to be taken.

Distribution: Calibration, Mechanical Design, Software

Test #15: Ball Recognition

Date: March 18th, 2016

Tester(s): Tiffany Wang

Author(s): Tiffany Wang

Hardware Version: 2.1

Software Version: Testing robot 1.0

Goal: Make the robot read the position of a ball and travel to it.

Procedure: As the robot advances, it finds the location at which the ultrasonic sensor's reading is the lowest in to localize the ball. Then, since the sensor is placed at 45 degrees, it calculates the corresponding coordinate system of the ball and travel there.

Expected Result: The robot navigates towards the ball and turns to its original direction.

Test Result: The robot stop at 3 cm away from the ball and when it turns, it is within 10 degrees error range.

Conclusion: The code functions well. Since this is not tested on the robot used for the final project, the results still have to be confirmed.

Action: Minimal calibration of the code.

Distribution: software development

Test #16: Ball Recognition

Date: 2 April 2016

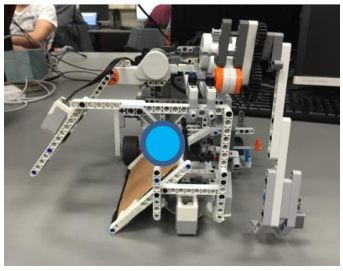
Tester(s): Sung Hong

Author(s): Sung Hong

Hardware Version: v3.0

Software Version: v2.2

Goal: Prior to the pickup, the tester enters the desired ball color. When picking up the balls, the robot successfully recognizes the color and uniquely picks up the balls of the right color. It discards the others.



Procedure: This test will only run the ball pick up mode. (off.testPickUp();). This will pick up the ball and lock it at the ball color reading position as shown in the picture. If the color of the ball is incorrect, the ball collector will release it, or else it would push the ball higher onto the storage track, where the balls will line up for the launch.

Expected Result: The robot will only pick up the ball with specific color choose by the tester and discard any other color balls.

Test Result:
Picking up the red ball

	1	2	3	4	5
Red ball	Yes	Yes	Yes	Yes	Yes
Blue ball	No	No	No	No	No

Picking up the blue ball

	1	2	3	4	5
Red ball	No	No	No	No	No
Blue ball	Yes	Yes	Yes	Yes	Yes

The robot only picked up the right colored balls

Conclusion: The robot was very accurate in determining which color ball it should pick up and should not. The test is successful.

Action: The test report will be sent to the software team to be reviewed

Distribution: Software development

Test #17: Odometry

Date: 3/16/2016

Tester(s): Garret, Nawras

Author(s): Garret

Hardware Version: V.2.1

Software Version: 2.0

Goal: get odometry to display (not working right now)

Procedure: rotate robot, see if odometer updates

Expected Result: updated odometer

Test Result: 0 degrees- odometer was not updated

Conclusion: There is a problem with either odometer, or odometry display

Action: Analyze classes to find solution

Distribution: Nawras look for solution

Solution: The odometer was correct. There was an error in the odometry display. The display was not getting the right values from the odometer.

Test #18: Odometry

Date: 2 April 2016

Tester(s): Tiffany Wang and Garret Holt

Author(s): Troy Yang

Hardware Version: v3.0

Software Version: v2.2

Goal: Test to see if the odometer can run properly. This includes properly reading the position of the robot and setting the coordinates to other values.

Purpose: From the localization test performed on April 1st, we realized that there is a problem with our odometer.

Procedure: First, we are going to float the motors with our hands and see whether or not the odometer updates its values. Second, we are going to manually push the robot on the board and see if the odometer correctly updates the values. Last, we are going to set the coordinates using the set methods.

Expected results:

1- When floating the motor with our hands
   a. if only one wheel moves or if the two wheels moves in opposite directions: the angle changes and not x and y.
   b. if both wheels move in the same direction: angle does not change, x and y changes according to the angles it is at.
2- When floating the motor on the board
   a. The rotating the robot: the angle should change accordingly.
   b. When moving backwards or forwards: the x and y coordinates should update accordingly
3- When setting the coordinate systems
   a. odo.setX(a), odo.setY(b), odo.setTheta(c) : (x, y, t) displayed on the screen should be (a, b, c)
   b. odo.setPosition(new double [] {a, b, c}, new boolean [] {true, true, true}); (x, y, t) displayed on the screen should be (a, b, c)

Test results:
Floating:

Tests	Success
Forward	yes (only y updates -- increasing)
Backward	yes (only y updates -- decreasing)
Rotate	yes
Moving in a certain direction	yes (x and y both updates)

Conclusion: the odometer is updating properly according to the motion of the motors.

On the board:

Tests	Success
Move from (0,0) to (0, 30.68) (we did not rotate the robot from 0^o to the direction of the destination).	yes – shows (3.0, 29.3, 278)
Move from (0,0) to (30.68, 61.36) (we did not rotate the robot from 0^o to the direction of the destination).	yes – shows (32.0, 58.34, 5.3)
Rotate 90^o clockwise	yes – shows (0.3, -0.2, 87)
Rotate 90^o counter clockwise	yes – shows (1.0, 0.5, 274)

Conclusion: the odometer is updating pretty accurately. The source of the problem should not be the imprecision of the odometer.

Setting:
3 different set of values:

1. Set x: we tried three times and it worked each time. The odometer displays x as the value we inputted. So the set method of x is not the problem that causes the inaccuracy in odometer.
2. Set y: we tried three times and it worked each time. The odometer displays y as the value we inputted. So the set method of y is not the problem that causes the inaccuracy in odometer.
3. Set theta: failed. No matter the value of theta we input, theta displayed on the screen remains the same as what it was previously. **Problem identified.**
4. Set position: we set the value of x, y and theta. The values of x and y update, however theta does not. This confirmed that it is indeed the values of theta that caused the problem with the inaccuracy in the odometer.

Improvement:
After we identified the problem, we started trying to figure out the solution to solve theta. It turns out that the problem lied in one line of the code that we wrote. In the code it states that theta = theta+deltaTheta. Which takes the previous value of theta and adds the change in theta to it to make up the new value for theta. However, the wheel did not move thus the value for theta would not change. After removing that line of code, the accuracy of the odometer improved drastically.

Division: Tester, Software Developer

Test #19: Navigation

Date: 3/17/2016

Tester(s): Garret, Nawras

Author(s): Garret

Hardware Version: 2.1

Software Version: 2.0

Goal: See if navigation working with accuracy in turning and movement

Procedure: place robot on ground, rotate 45 degrees. Travel 90 cm, rotate 180, travel 90 cm

Expected Result: Robot end up at initial position

Test Result: Robot ended up close to initial, off by a few cm and a few degrees

Conclusion: Navigation is working but additional calibration to wheel radius and width needed to improve accuracy.

Action: Further testing

Distribution: later this week Matthew do further testing

Test #20: Navigation

Date: 28 March 2016

Tester(s): Sung Hong

Author(s): Sung Hong

Hardware Version: v3.0

Software Version: v2.2

Goal: Accurately navigate to a final destination

Procedure: The robot will be placed at the (0, 0) facing 0 degrees, which corresponds to the original position of the robot. Then we will make the robot execute 6 different trajectories and see if it can travel to the destinations.

Expected Result: The robot should calculate the path to the final destination. Then, it should turn to the angle of the destination and travel to it accurately. The final physical location of the robot should be within a 2 cm range from the target.
The robot should calculate the path to the final destination and move accurately to the final location with error being smaller than 3cm.

Test Result:

Destination	(0, 60)	(30, 60)	(30, 90)	(0, 30)	(0, 60)	(30, 30)
Error X (cm)	-1.5	1.3	1.3	0.3	-0.9	1.1
Error Y (cm)	-0.7	-0.3	-1.2	-1.2	-1.5	-0.9

$\text{Average}_x = \frac{\sum error}{6} = 0.267$

$\text{Average}_y = \frac{\sum error}{6} = -0.967$

Total of 6 tests were performed. In all these tests, the robot's error in X and Y were within the accuracy of 3cm.

Conclusion: The robot meet the expectation however it may need further improvement because the error will accumulate as moving longer distance. As seen, the error of the Y component is all negatives. Which shows that perhaps there is a problem is the turnTo()

method. The robot might have turned to the wrong angle which increases the inaccuracy of the travelTo() method.

Action: Implementation of OdometryCorrector or more calibration and weight distribution will be made to reduce the error.

Distribution: Project manager and mechanical designer

Test #21: Localization

Date: 3/21/2016

Tester(s): Garret Holt, Nawras Rabbani

Author(s): Garret

Hardware Version: v3.0

Software Version: 2.1

Goal: Localize the robot on the intersection of two black lines

Procedure: place robot on ground run localization, pick up, repeat 5 times

Expected Result: Robot correctly localize, display correct X, Y and theta coordinates

Test Result:

Run #	X error(cm)	Y error(cm)	Theta error(degrees)
1	3	2	15
2	4	5	15
3	4	3	25
4	3	4	20
5	3	2	10

$\text{Average}_x = \frac{\sum_{k=1}^{10} angle_k}{10} = 3.4$

$\text{Average}_y = \frac{\sum_{k=1}^{10} angle_k}{10} = 3.2$

$\text{Average}_{theta} = \frac{\sum_{k=1}^{10} angle_k}{10} = 17$

$\text{Standard Deviation}_x = \sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 0.54$

$\text{Standard Deviation}_y = \sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 2.63$

$\text{Standard Deviation}_{theta} = \sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 1.30$

Conclusion: Light localization and ultrasonic localization are off. From the averages and the standard deviations, we can see that the error in the x-axis and the error of the angle

are consistent (the standard deviation is small). However, the final position of the robot in the y-axis varies remarkably. This might be a consequence of an erroneous US localization. We also noticed that the odometer shows that it is off in its rotations due to improper calibration. In order be able to reliably test the effectiveness the calibration must be better.

Action: Calibration, Tiffany Wang and Sung Hong will take care of the further calibrations. The USLocalization and the LightLocalization will be completed separately in order to find the reduce the errors accumulated in each part.

Distribution: Software, Calibrations

Test #22: Localization

Date: 1 April 2016

Tester(s): Garret Holt

Author(s): Troy Yang

Hardware Version: v3.0

Software Version: v2.2

Goal: Previously, we have tested the localization code and showed that it was successful. However, we want to test it again and verify its odometer readings.
Procedure: Place the robot at the corner of the board and run the localization test. At the end, take note of the displayed odometer values.

Expected result: After fully localized, the robot should display its physical location on the screen, i.e. x = 0, y = 0, t = 0. And the robot should be well positioned on the (0, 0, 0)

Test Result:

Trial(s)	1	2	3	4	5
Value(s)	255.1	254.9	93.5	8.7	22.5
Correctly positioned	yes	yes	yes	yes	yes

Conclusion: As can be seen from the chart, the values produced are rather random and they are really far off the values that we are supposed to get, which is 0. Although the robot is facing 0^o and theta has been set to zero, the odometer shows the angle it has rotated from its original position. Clearly, there is a problem with the odometer, it does not set the value angle of theta to zero.

Action: test the odometer individually in order to identify the problem within the odometer class.

Division: Tester, software developer

Test #23: Localization

Date: 2 April 2016

Tester(s): Garret Holt and Tiffany Wang

Author(s): Troy Yang

Hardware Version: v3.0

Software Version: v2.2

Goal: The robot can physically localize to the (0, 0, 0) coordinates, and update the odometer readings to (0, 0, 0)

Expected result: After fully localized, the robot should display its physical location on the screen, i.e. x = 0, y = 0, t = 0. And the robot should be well positioned on the (0, 0, 0)

Test Result:

Trial(s)	1	2	3	4	5
(x, y, z)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
Correctly positioned	yes	yes	yes	yes	yes

Conclusion: The test is successful, the robot can now localize properly and updates the odometer readings.

Action: The results of this test shows that the problem of the navigation should be solved. Our job now is the test the full test run again.

Division: Tester, software developer

Test #24: Localization (with light sensor angle correction)

Date: 4/06/2016

Tester(s): Tiffany Wang

Author(s): Tiffany Wang

Hardware Version: 3.0

Software Version: 2.1

General explanation: Although the previous tests of localization were satisfactory, we decided to add a little tweak at the end of the method. Our TA, Daniel Lima, suggested to correct the angle of the robot with the light sensor at the back of the robot. Precisely, aligning the back light sensor with the black line would perfectly orient the robot to 0^o. This would also solve the problem of any slight inaccuracy of the ultrasonic localization. We decided to integrate this little correction at to our localization to increase the accuracy.

Goal: Localize the robot to (0, 0, 0). The robot should precisely land at the intersection of black lines and face 0^o (+ y-axis). This should be completed within 30 seconds.

Procedure: Place the robot (center of rotation) within the first tile at the corner and run the localization test. Time the robot. At the end, note down the physical position of the robot. Repeat 10 times.

Expected Result: The robot should localize within 30 seconds. At the end, it should be perfectly centered at (0, 0) and facing 0^o. As well, the odometer should display (0, 0, 0)

Test Result:

Run #	Original orientation	within 30 seconds	X error(cm)	Y error(cm)	Theta error(degrees)
1	180^o	Yes	1	0	0
2	-90^o	Yes	0.5	0	0
3	-135^o	Yes	1	-0.5	5
4	45^o	Yes	1	0	4
5	135^o	Yes	1	0	0
6	-90^o	Yes	0	0	0
7	0^o	Yes	2	0	5
8	180^o	Yes	0	0	0
9	45^o	Yes	0.5	0	90
10	180^o	Yes	1	0	0

$\text{Average}_x = \frac{\sum_{k=1}^{10} angle_k}{10} = 0.8$

$\text{Average}_y = \frac{\sum_{k=1}^{10} angle_k}{10} = -0.05$

$\text{Average}_{theta} = \frac{\sum_{k=1}^{10} angle_k}{10} = 10.4$

Standard Deviation(x) = $\sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 0.58$

Standard Deviation(y) = $\sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 0.158$

Standard Deviation(theta) = $\sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 28.05$

Observation: In general, the localization results are really satisfactory. However, there is a run where the robot ended by facing 90^o instead of 0^o. In fact, during the run, the ultrasonic sensor experienced a false positive result during the UsLocalization. Consequently, the angle correction is wrong. At the end of the LightLocalization, the robot's orientation was actually closer to the 90^o line. As a result, when it was searching to align to the black line, it sensed the 90^o line and localized accordingly. If the results of this run were omitted, the results are remarkably satisfactory. The new calculation of the average and the standard deviation of theta are the following:

Average(theta) $= \frac{\sum_{k=1}^{9} angle_k}{9} = 1.55$

Standard Deviation(theta) = $\sqrt{\frac{1}{9}\sum_{i=1}^{9}(x_i - \mu)^2} \approx 2.35$

The final position of x, y and theta are close to 0. The low standard deviation of the three coordinate systems demonstrates the accuracy and the consistency of the result.

Conclusion: The test is successful. Statistically, the test has a 90% success rate.
The only worry for the implementation of the light sensor angle correction is that it senses the wrong line.

Action: No further test is required. Localization method works perfectly.

Distribution: Software, testing.

Test #25: Localization – Ultrasonic Sensor

Date: 3/26/2016

Tester(s): Tiffany Wang

Author(s): Tiffany Wang

Hardware Version: v3.0

Software Version: 2.2

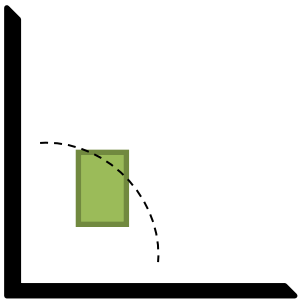
Goal: Localize the angle of the robot

Procedure: place robot on ground run localization, pick up, repeat 5 times

Expected Result: The robot orients towards 0^o (which is the y axis)

Test Result:

Run #	Initial orientation	Final angle
1	-90	0
2	45	1
3	180	0
4	-135	-5
5	-135	-10
6	-135	-1
7	-45	0
8	135	2
9	-135	5
10	180	0



Average $= \frac{\sum_{k=1}^{10} angle_k}{10} = -0.8$

Standard Deviation = $\sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 4.08$

Conclusion: According to the average, the overall accuracy of the Ultrasonic Localization is pretty high. However, the results vary depending on the initial position of the robot. For instance, when the robot is initially oriented towards either wall, it localizes perfectly to 0^o. However, when it is placed facing the corner of the wall, localization fails.

Action: Filtering – by implementing a filter can minimalize the erroneous readings of the ultrasonic sensor. Some further calibration could reduce the shift of the center of rotation

that could perhaps fix the logged angles of the walls. Should the center of rotation shift, these angles differ from the actual one which would result to an inaccurate localization.

Distribution: Software, Calibrations

Test #26: Light Lozalization

Date: 24 March 2016
Tester(s): Sung Hong

Author(s): Sung Hong

Hardware Version: v3.0

Software Version: v2.2 (with WHEEL_WIDTH = 16.9 and WHEEL_RADIUS = 2.07)

Goal: Use color sensor to localize the robot (x and y coordinates)

Procedure: The robot will be placed at the (-3, -3) facing 0 degrees. The robot will rotate itself to gather 4 dates based on the black grid lines on the tile. With the given dates, the robot will localize itself. The test will be performed 10 times

Expected Result: The robot will gather data and show its current x and y coordinates according to where it has finished which should be (-3, -3). The LCD display should be within the accuracy of 4cm.

Test Result:

	1	2	3	4	5	6	7	8	9	10
Odo X (cm)	-3.14	-3.56	-3.98	-2.53	-2.87	-3.13	-3.53	-2.56	-2.72	-3.95
Odo Y (cm)	-3.85	-2.52	-2.00	-3.73	-2.57	-2.18	-2.22	-2.67	3.86	-3.53
Actual X (cm)	-3.1	-3.7	-3.3	3.4	-3.6	-3.1	-3.5	-3.4	-3.4	4.1
Actual Y (cm)	-3.9	-4.3	-4.1	-4.4	4.3	3.8	4.0	3.7	-3.7	-4.7

Total of 10 tests were performed. In all these tests, the robot's LCD display was within the accuracy of 3cm.

Conclusion: The robot meets the expectation and had reliable light localization using the light sensor

Action: The test report will be sent to the software team to be reviewed and Gantt chart should be updated.

Distribution: Software development and project manager

Test #27: Ball Launch

Date: March 25th, 2016

Tester: Tiffany Wang and Troy Yang

Author(s): Tiffany Wang

Hardware Version: v.3.0

Software Version: v.2.2
with **acceleration** = 1500;
launchSpeed = 2000;
launchAngel = 75;
rotateSpeed = 50;
pickUpAngle = 43;
pickUpAcceleration = 100;

Goal: Shoot a ball to hit a specific target 2 meters away.

Procedure: Place the robot on a black line on an 8x8 board and along a wall. Then shoot the ball to the other end of the of the board. The black line is used as a reference for the accuracy of the shot.

Expected Result: The ball lands perfectly on the black line.

Test Result: (where the positive axis is at the right of the line)

	Distance from the black line (cm)
1	-1.5
2	6.5
3	8
4	6
5	10
6	8
7	6.5
8	7
9	5
10	1.5

Average $= \frac{\sum_{k=1}^{10} angle_k}{10} = 6.17$

Standard Deviation $= \sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 3.36$

where x_i is each value of the data, and μ is the median.

Conclusion: The standard deviation and the average shows that the trajectory of the ball is crooked. In general, except for the first case the ball landed to the right of the black line. However, from my observation, it was due to some instabilities of the catapult arm. The acceleration of the catapult was too high for the small angle of rotation. Thus, the came to a halt too fast and oscillated.

Action: I stabilized the arms by adding another bridge holding the motor to the body of the robot, and also decreased the lauchingAcceleration from 3000 to 2000.

Distribution: Mechanical Design, Software Design

Test #28: Ball Launch

Date: March 25th, 2016

Tester: Tiffany Wang and Troy Yang

Author(s): Troy Yang

Hardware Version: v.3.0

Software Version: v.2.2
with **acceleration** = 1500;
launchSpeed = 2000;
launchAngel = 75;
rotateSpeed = 50;
pickUpAngle = 43;
pickUpAcceleration = 100;

Goal: Shoot a ball to hit a specific target 2 meters away.

Procedure: Place the robot on a black line on an 8x8 board and along a wall. Then shoot the ball to the other end of the of the board. The black line is used as a reference for the accuracy of the shot.

Expected Result: The ball lands perfectly on the black line.

Test Result: (where the positive axis is at the right of the line)

	Distance from the black line (cm)
1	-2
2	2.5
3	5
4	6
5	6
6	3
7	0
8	1
9	2
10	1

Average $= \frac{\sum_{k=1}^{10} angle_k}{10} = 2.45$

Standard Deviation $= \sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \mu)^2} \approx 2.63$

where x_i is each value of the data, and μ is the median.

Conclusion: The average is within a 3 cm range from the black line, which is acceptable. The standard deviation shows that the results are pretty consistent. However, this result could be improved by a further calibration and perhaps fuller charged battery in order to increase the performance of the robot.

Action: The result of the ball launching is satisfying. As of now, we are not going to make further changes.

Distribution: Mechanical Design, Software Design

Test #29: Obstacle Avoidance

Date: 26 March 2016

Tester(s): Sung Hong

Author(s): Sung Hong

Hardware Version: v3.0

Software Version: v2.2 (with WHEEL_WIDTH = 16.9 and WHEEL_RADIUS = 2.07)

Goal: To travel from one place to another with avoiding an obstacle

Procedure: The robot will be placed at the (0, 0) facing 0 degrees. The robot will then travel 90 cm for
ward and in between its path, a wooden block will be placed (at (0, 45)).

Expected Result: The robot will travel to (0, 90) avoiding the wooden block on its path

Test Result:

	1	2	3	4	5	6	7	8	9	10
Avoided	No	No	No	No	No	No	No	No	No	No

The robot ran into the wooden block every time before it could make its turn to avoid it.

Conclusion: The robot did not meets the expectation. The robot made its turn after it has ran into the block. It was unable to avoid the obstacle and travel to the designated location

Action: The minimum value which allows the robot to detect the wooden block needs to be adjusted. Also the ultrasonic sensor could be placed at a better angle in which the blocks can be detected more efficiently.

Distribution: Software development and mechanical designer

Test #30: Full Run Test

Date: 28 March 2016

Tester(s): Tiffany Wang and Sung Hung

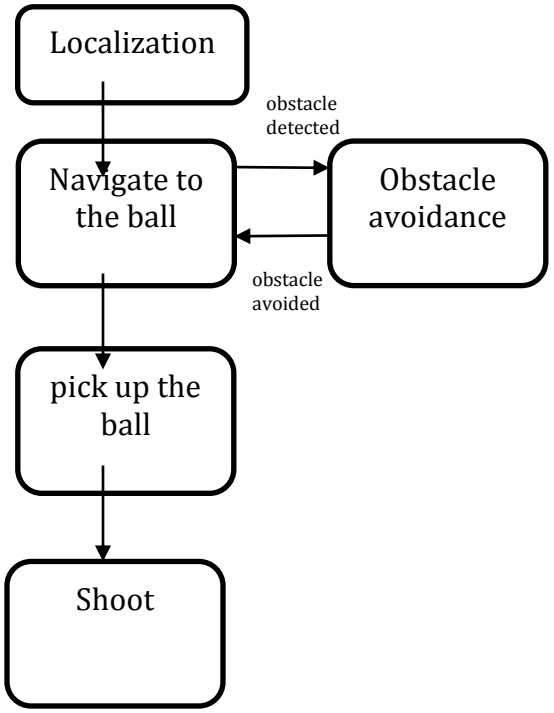
Author(s): Garret Holt

Hardware Version: v3.0

Software Version: v2.2

Goal: The robot can autonomously accomplish all the required tasks for the demo: localization, navigation, obstacle avoidance, ball pick up and ball launching (towards the right direction).

Procedure: We are going to load our main source code with the integration of every component. The steps of the code:



Then, we are going to place the robot within the first tile of the board, just like during the demo, and it will run on its own.

Expected Results: The robot runs through the code successfully.

Test Results:
Result table:

Trials	Success
1	no
2	no
3	no
4	no
5	no

Table of the angles the robot turns after localization

Trial(s)	Angle it should turn to	Angle Turned(deg)	Error(deg)
1	45	100	55
2	45	25	20
3	45	-68	113
4	45	10	35
5	45	-90	135

Means of the error in angle: (55 + 20 + 113 + 35 + 135) ÷ 5=71.6

Standard Deviation = $\sqrt{\frac{1}{5}\sum_{i=1}^{5}(x_i - \mu)^2} \approx 50.07$

Conclusion: The test has failed. We tested the robot in full run, however the result has not been really successful. After the robot finishes the localization, it was able to to turn to the right angle. The robot would turn to the wrong direction. As seen in the testing results, the robot turns to random directions after localization. Initially we thought it was an error, however after a few testing runs, we found out that this always happens and the robot cannot turn to the right direction after the localization. The standard deviation is really high. This shows that the error is not constant, which means that it should not be a problem with our navigation methods. Our guess is that the problem is in our odometer, for the navigation method works perfectly fine when tested alone.

Action: test localization once again and find out the problem.

Division: Tester, Software Developer

Top

Post-Milestone II Reflection

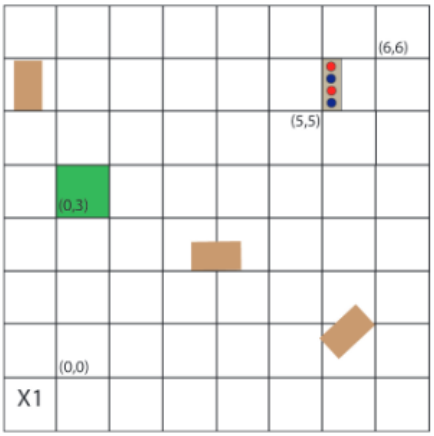
Team 03
Project: Neymar
Task: Soccer - Playing Robot

Document version: v.1.0
Date: Saturday April 2nd, 2016
Author: Matthew Rodin

This purpose of this document is to provide an insightful reflection regarding the outcome of Milestone II in order to correct outstanding faults.

Milestone II Specifications

"This demo will entail a subset of the competition on a reduced grid of 8' x 8'. Only one robot will be run at a time, originating from the corner shown in the diagram below. The demo will be started once a message has been received from the WiFi server class that will be posted shortly on myCourses. The corresponding client has everything necessary to return the full set of competition parameters. For this demo we will only be using the SC parameter (but in a different way), and the 4 parameters that define the location of the ball holder, ll-x, ll-y, ur-x, and ur-y."



Result

After the robot was placed in the starting tile and the jar file was launched, it began localizing. However, after a successful localization, the robot directed itself toward a boundary wall, instead of in the direction of the ball area, and then proceeded to navigate towards the wall. Before the robot made contact with the wall, a team member manually re-oriented the robot towards the ball pickup location in an attempt to showcase the robot's launching mechanism. Nevertheless, the robot was unable to avoid the obstacle in its path and we decided to terminate the demo due to too many complications.

Advice From Professors

After the software developers explained to the professors that they believed the problem to be caused by multi-threading, Prof. Ferrie explained that the multiple threads are running in parallel and not one after in sequence. Therefore, multiple threads were updating the same global variable at unpredictable times and forced an inaccuracy in the odometer readings. Derek (TA) suggested that we review the threading tutorial posted on myCourses as well as explore the possibility of implementing a finite state machine in order to properly control the sequence of the threads.

Conclusion

After getting over the disappointment of a complete demo failure, the software team decided to set up an appointment with our assigned teaching assistant (Daniel) in order to resolve the threading issue. Even after exploring the possibility of utilizing a finite state machine, Daniel suggested that we first attempt a simple “thread lock”, which would give us more control over the threads. Knowing that all the individual components of our software work very well (odometry, localization, navigation, ball collection & launch) and that the root of the issue is simply a threading problem, we are confidant that the robot will be in perfect working condition for the final demo!

Top