**LeJOS Study:**

**Clear the Table**

**CS491**

**Special Topics (AI)**

March 31, 2015

By Cody Kubicek

And Tyler Rusch

**Abstract:**

This study serves as a look into full implementation of an algorithm. When merely testing code, all inputs and outputs are consistent and perfect—if no errors exist within the code, the result will be predictable and accurate every time. However, when switching to physical implementation, this is not always the case. Inputs may be flawed and execution may be less than optimal, and it is important to see how obtaining a degree of robustness can require a different approach.

Contents

[1. Introduction 1](#_Toc412577616)

[2. Background 1](#_Toc412577617)

[3. Approach 2](#_Toc412577618)

[4. Results 7](#_Toc412577619)

[5. Conclusion 8](#_Toc412577620)

[6. Appendices 9](#_Toc412577621)

List of Figures

Figure 1. Simple Reflex Agents…………….……………………………………………………….…1

Figure 2. The Robot……………………………………………………………………………………..3

Figure 3. Robot Scanning Path……………….………………………………………………………..4

Figure 4. The Moderately Populated Table….………………………………………………………..6

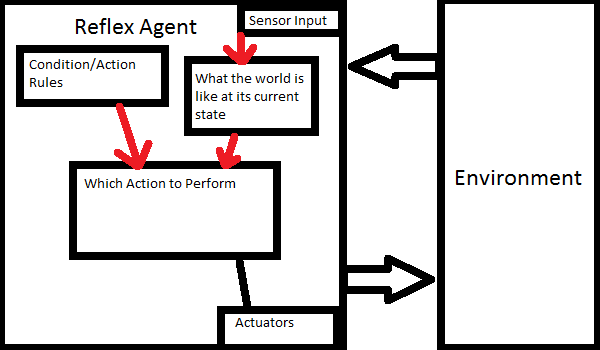
Figure 5. The Sparsely Populated Table.….………………………………………………………….7

# Introduction

When working with only code, programmers can take inputs and outputs for granted, as all sample runs and testing sessions are consistent and repeatable. This is not the case when working with a physical robot, and the effects of flawed I/O make for a valuable shift of focus from being solely on the design of the code to the actual application.

To gain familiarity with the LeJOS robots, a seemingly simple task was given: find and push objects off of a table. The goal was to research what functions the robot was capable of while working within the limitations of the sensors provided. The difficulty of the task lied not necessarily with the design of the algorithm itself, but the design of the robot and how it cooperated (or did not cooperate) with the algorithm.

# Background

This robot is a simple reflex agent as shown in Figure A below, meaning the agent/robot takes in information from the environment, and then looks to its established concepts and rules to make a decision. After the decision is made, the agent will then make a change to the environment followed by a repeat of the cycle until a terminating condition has been met.

**Figure 1: Reflex Agent Diagram**

# Approach

**3.1 The Sensors**

**3.1.1 Touch Sensors**

The first sensor put to use were the two touch sensors. These are simple binary on/off switches that check to see whether the button has been compressed or not.

Unfortunately, though this was the most obvious way to implement edge detection, the fact that the switches are not analog causes a problem when the buttons are not compressed enough.

Because of that, appropriate force is required which caused the buttons to be positioned in a peculiar way. Since there was no way to generate the force when the arms went off of the table, the force had to be generated while on the table then checked for a release. Initially, their positioning was simply at the ends of the arms, but because the force required was higher than expected, not only was the robot being lifted and losing enough traction to make turning near impossible, but even when it could the friction would work against it.

By redesigning the arms and repositioning the touch sensors, some of the force was able to be directed in a slightly different direction, and the arms were able to switch to a smoother surface to help with friction. Though the lift problem was not solved entirely, it was able to be mitigated by adding weight to the robot via the cup.

**3.1.2 Ultrasonic Sensor**

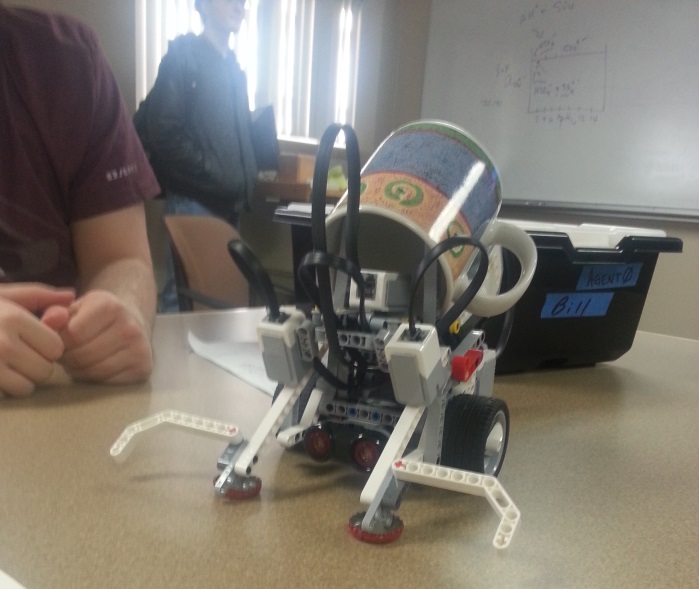
The ultrasonic sensor served as the eyes of the robot. In reality, it sends out a “ping” and checks to see how quickly that ping returns. If readings are registered under a certain distance, the robot confirms that it “sees” an object and will carry out procedures to push it off of the table.

**3.1.3 Gyroscope**

A later addition to the robot, the gyroscope tracks how far the robot has rotated. This was discovered late into the design process, but was included to try and increase the accuracy of the robot’s turns. For each turn, the gyroscope is reset, then queried to see how far it has turned from that reset position.

**3.2 The Robot**

The robot we worked with was the LeJOS EV3 model. In addition to the sensors above, the kit contained Legos pieces for us to build a base for the brick (the brains of the robot) to reside in.



**Figure 2. Final Robot Design**

Though not entirely visible, the robot is balanced with its two forward arms that served both as levers for the touch sensors as well as catches for the objects, two wheels, and a supporting ball-joint in the rear. Staying with two wheels as opposed to four allowed for more simplistic point turns, but also meant that the robot was vulnerable if any of those points were to work less than optimally.

However, extra pieces were added to attempt to reverse those problems. The cup placed on top of the robot, as well as the bottle caps attached to the feet, assisted with the friction/traction problems caused by the forces required by the touch sensors. The weight of the cup allowed the wheels to gain some traction, and the bottle caps provided a smoother, rounder surface than the Lego arm ends.

Additionally, extensions were added to the arms in order to cover for robot stopping “early” when spotting an object. This guaranteed that if something was within the robots scan range, it would be captured by the extensions and carried along.

**3.3 Classes**

**3.1.1 Cups**

Cups.java is the main class for the project, and as such combines and calls the other classes. The Cups class initializes all variables, then sets up the robot’s looping operation. In order to keep code clean, the looping operation was moved from the main class to the sweep() function. That way, the main function only needs to loop calls of sweep() until the terminating conditions have been met. In order to ensure the entire table had been appropriately scanned, it was decided that the program would end if the robot performed two full sweep() functions without finding any objects. If an object was found, the robot returns to a starting position and resets the counter.

**3.2.2 Eyes**

Eyes.java is the main implementation for all the Ultra-sonic sensors. It simplifies the control for the ultrasonic sensors. It contains all the distance methods, which measure the distance between the sensor and the object being scanned.

This class exists merely to help keep the code readable, as it is much easier to call its only function getDistance() rather than repeatedly assign the variables within.

**3.2.3 Gyro**

Gyro.java is the implementation for all the gyroscopic sensor control. It simplifies controlling turning, and contains a method for getting the turning distance that the agent has turned.

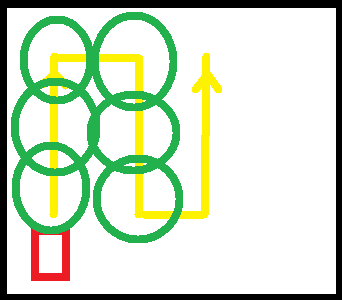
This class exists merely to help keep the code readable, as it is much easier to call the function getDistance() rather than repeatedly assign the variables within. Gyro contains one extra reset() function, which allows it to be set back to zero before counting the degrees turned.

**3.2.4 TouchSensors**

TouchSensors.java is the implementation for the two touch sensors that ensures that the robot does not go off the edge of the table. It contains the methods for determining which, if any, touch sensors are compressed.

**3.4 Helper Functions**

**3.4.1 sweep(pilot, eyes, touch, gyro)**

Since sweep() is the primary function of the robot, it requires information on all of the controllers available—the motors, eyes, touch sensors, and the gyro sensor. The sweeping motion consists of two movements: a full cross of the table followed by a sideways adjustment. The robot moves forward, scanning occasionally, until an edge is hit. The robot then adjusts, turns, and performs the sideways adjustment. After both moves, a global variable DIRECTION is flipped by multiplying it by -1 so that the robot can continue moving across the table. This is looped until the robot hits an edge *during the sideways adjustment*.

*Figure 3: Robot scanning path.*

**3.4.2 flush(pilot, touch)**

Flush() is a small but important adjustment called any time the robot reaches an edge. It needs control of the motors and touch sensors in order to shift the robot so that it is directly facing the edge it hits in the event that it approaches any edge at an angle. This is required because the sweeping motion operates off of 90 degree angles, and without it would follow the path appropriately.

Depending on which touch sensor is no longer compressed, flush() rotates the robot in the appropriate direction until the other touch sensor is released. At this point, an extra adjustment is made to face the edge, and the robot is moved backwards to safety.

**3.4.3 findCorner(pilot, touch, gyro, eyes)**

findCorner() is the beginning of the setup for the robot. In order to ensure that the sweep goes from one end of the table to another, the robot must begin from a consistent position. Thus, findCorner() moves forward until an edge is found, calls flush() to face it, then turns right 90 degrees and repeats.

In order to speed up the cleaning process, findCorner() was given scan functionalities, so that if any object were found during this process, it would be pushed off and the process would restart. In order to make sure findCorner() did not attempt to pick up where it left off after finding an object, the RETURNINGFROMCOP conditional was added to force the function to start from the beginning when called.

**3.4.4 findEdge(pilot, touch)**

findEdge() is a minor function that moves the robot forward until one of the touch sensors detects that there is no longer a surface in front of it, then stops. It then calls flush() in order to face the edge directly.

**3.4.5 testEyes(eyes)**

testEyes() is a debugging method, used in calibrating the Ultrasonic sensor. It prints out the range that an object is detected at intervals so that an optimal range could be found.

**3.4.6 rotate(pilot, gyro, angle, eyes, touch)**

Rotate() is a combined function—initially, turning and scanning were separate actions, but it made more sense to simply scan every time the robot did any turn.

To do this, the robot turns slowly while the gyroscope tracks the difference in the starting and current angle, moving until either the ultrasonic sensor detects something within a certain scan range (.4 meters, though in practice this turned out to be slightly less than that due to the ultrasonic sensor’s variance) or the gyroscope detected that the robot had turned an amount equal to 5 degrees less than the desired angle. This 5 degree adjustment accounted for the gyroscope’s natural error as well as the stopping time required by the robot.

If an object was found, the robot calls findEdge(), moving forward until there is no surface left in front of it and effectively pushing it off.

**3.5 Running the Program**

The execution of the code was as follows:

1) Find a corner to begin from

2) Begin first sweep across the table

2.1) If an object is found, push it to the edge and repeat from step 1

3) If nothing is found, sweep back across the table to double check

3.1) If an object is found, push it to the edge and repeat from step 1

4) If nothing is found again, terminate

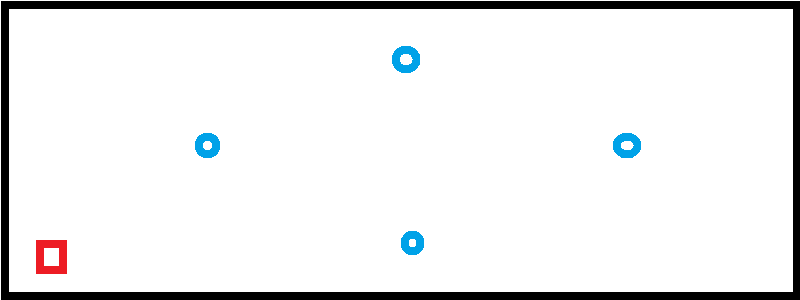
**3.6 Testing the Program**

Testing the robot’s function came down to repeated, smaller tests rather than full tests of the cleaning function due to time constraints. The touch sensors, ultrasonic, and gyroscopic sensors all require error correction. The small tests consisted of turning a small amount, then double checking if the amount was accurate. Small adjustments were made until the accuracy desired was obtained.

During tests, it was found that the design of the robot had the flaws previously mentioned (traction, etc.), and the added weight and foot-pads were introduced to help with consistency.

**Test Situation 1**

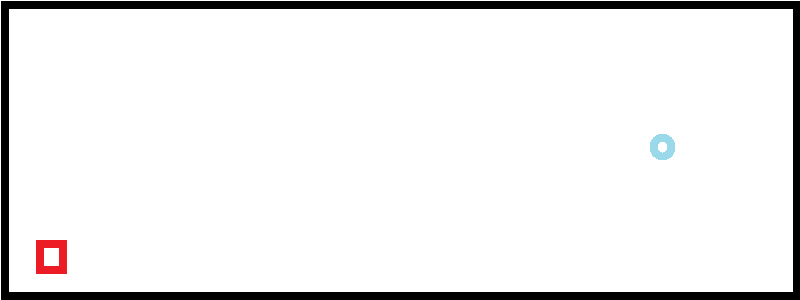
This situation had four cups, placed in a diamond shape as shown below in Figure 3.



**Figure 4. Moderately Populated Table**

The robot can successfully clear the table. The only problem that occasionally occurs is when the robot attempts to push an object off at very shallow angle. The shallow angle moves the wheel base off the table, followed by the touch sensor decompressing. Once the touch sensor is decompressed, the robot will turn to face the edge. Since the wheel base is off, the robot topples off the edge. This does not happen on every run through the program, and after several trials, this could be considered a partial success.

**Test Situation 2**



**Figure 5. Sparsely Populated Table**

The second situation shown above in *Figure 4* was to test the Robots competence in a low populated environment. After several scans, and completing half a sweep of the table, the cup was removed. After several completions without error, it could safely be said that this challenge is a success.

# Results

During the robot’s demonstration, though it was able to find and clear all cups, it did begin to fall off of the table at one point. Additionally, there were some extra unnecessary movements that came about from attempts to prevent the very case where the robot failed, but these were only extra quirks that neither helped nor hurt the objective.

**4.1 What Went Wrong**

The case in which the robot failed, although a rare case, was a result of a slight flaw in design that was not protected by the code, nor caught by the design placement of the touch sensors. Though the arms did accurately detect when there was no space for the robot to move forward, it did not detect movement to the side or backwards. The code does contain a slight movement backwards after reaching an edge and facing it, but since it cannot check to the sides, if the robot were ever to push an object parallel to an edge of the table, the rear ball-joint will fall off the table while rotating.

Additionally, the factors such as friction and acceleration made it so that the robot did not accurately turn the amount desired every time. Often, it would turn just slightly more than 90 degrees, resulting in the algorithm taking slightly longer and the robot lingered backwards. If the robot did reach an edge before being led too far off track, the flush() function would correct it again, but if the turns were too extreme, there is risk of the robot thinking it has completed a sweep and turning around.

After demonstrating, it was realized that the algorithm contained an unnecessary “return trip” that caused it to sweep back over the table to make sure all objects were cleared. In theory, this was to make sure that the robot did not clear only half of the table when set to begin in the center, but because of the decision to force the robot to start in a corner every time, such a scenario would never occur and a single completed sweep would be sufficient.

**4.2 What Would Be Changed**

Working with the touch sensors was problematic and not recommended for edge detection in the future. Instead, color or light sensors, thanks to their limited range, could serve a similar purpose by searching for reads that return no value instead. Since the light sensors do not have to be in physical contact with the surface, the problems caused by friction would not exist making movement simpler and more accurate.

The arms containing the light sensors would also have to be extended outwards. The scenario in which the robot fell off the table was caused because the arms did not reach far enough, and it was possible that the arms remained in contact with the surface while the ball-joint did not. Extending the arms would create a larger “safety perimeter” in which the robot would not approach an edge any closer than it would be safe to.

# Conclusion

Though the algorithm for scanning then entire table was quite simple by itself, the challenge of the project came from the variance brought about through imperfect input (ultrasonic sensor fails to detect, gyroscope fails to accurately detect number of degrees, etc.) and imperfect execution (wheels failing to gain traction, motors turning too far). This called for an increased degree of robustness, but even with careful precautions, the robot was still vulnerable to failure. It was decided that the original design of the robot would be scrapped in favor of one that had less of its own pieces working against it, and a sufficient understanding of the code behind the LeJOS EV3 robot was gained through this introductory study.

# Appendices

**6.1 Cups.java**

package cups;

import java.util.concurrent.TimeUnit;

import lejos.hardware.Brick;

import lejos.hardware.BrickFinder;

import lejos.hardware.Button;

import lejos.hardware.motor.EV3LargeRegulatedMotor;

import lejos.robotics.RegulatedMotor;

import lejos.robotics.navigation.DifferentialPilot;

import lejos.hardware.Sound;

public class Cups {

// http://lejos.sourceforge.net/tools/eclipse/plugin/ev3

public static int DIRECTION = 1;

public static int BIG = 100;

public static boolean RETURNINGFROMCUP = true;

public static void main(String[] args) {

Brick brick=BrickFinder.getDefault();

RegulatedMotor leftMotor=new EV3LargeRegulatedMotor(brick.getPort("A"));

RegulatedMotor rightMotor=new EV3LargeRegulatedMotor(brick.getPort("D"));

TouchSensors touch = new TouchSensors(brick);

Eyes eyes = new Eyes();

Gyro gyro = new Gyro();

DifferentialPilot pilot=new DifferentialPilot(5.24, 10.5, leftMotor, rightMotor);

pilot.setRotateSpeed(30);

pilot.setTravelSpeed(8);

Button.waitForAnyPress();

int fail = 0;

//Begin by finding a corner so we have a consistent starting position

findCorner(pilot, touch, gyro, eyes);

//Exit condition: Robot makes two full sweeps of the table without finding any cups

while(fail < 2) {

//Cut-off condition (for testing purposes:

//if both sensors are deactivated (robot is flipped upside-down), quit early

if(!(touch.leftIsPressed() && touch.rightIsPressed())) {

Sound.beep();

Sound.beep();

break;

}

//Sweep in one direction, returns true if something is found

if(sweep(pilot, eyes, touch, gyro)) {

//If a cup is found, reset the robot (# of fails, direction, realign to corner starting position)

fail = 0;

DIRECTION = 1;

findCorner(pilot, touch, gyro, eyes);

} else {

//Sweep did not find a cup,

//Prepare to sweep in the other direction

rotate(pilot, gyro, 90 \* DIRECTION, eyes, touch);

fail++;

}

}

}

public static boolean sweep(DifferentialPilot pilot, Eyes eyes, TouchSensors touch, Gyro gyro) {

//Begin the sweep by scanning in the current position

if(rotate(pilot, gyro, 360, eyes, touch)) {

return true;

}

//Loop until something is found or the sweep has failed

while(true) {

//move forward to reposition

pilot.travel(40, true);

//Check condition to prevent running off the table

while(pilot.isMoving()) {

//if you hit an edge, turn 90 degrees and travel forward, then turn 90 more degrees

//This pattern creates the "zig-zag" across the table

if(!(touch.leftIsPressed() && touch.rightIsPressed())) {

pilot.stop();

flush(pilot, touch);

if(rotate(pilot, gyro, 90 \* DIRECTION, eyes, touch)){

return true;

}

pilot.travel(40, true);

//if you hit an edge during this move, you're at the end of the sweep

while(pilot.isMoving()) {

if(!(touch.leftIsPressed() && touch.rightIsPressed())) {

pilot.stop();

return false;

}

}

//Adjust slightly backwards for safety

pilot.travel(-5);

//turn 90 degrees again

if(rotate(pilot, gyro, 90 \* DIRECTION, eyes, touch)) {

return true;

}

//Change the direction of our turns

DIRECTION = DIRECTION \* -1;

}

}

//Full scan

if(rotate(pilot, gyro, 360, eyes, touch)) {

return true;

}

}

}

//Flush is an attempt to get the robot to face the edge directly for orientation

public static void flush(DifferentialPilot pilot, TouchSensors touch) {

//if left is off the edge, turn counter clockwise to get right off the edge

if(touch.leftIsPressed()) {

while(touch.leftIsPressed()) {

if(!pilot.isMoving()) {

pilot.rotate(20, true);

}

if(!touch.leftIsPressed()) {

pilot.stop();

}

}

//if right is off the edge, turn clockwise to get left off the edge

} else {

while(touch.rightIsPressed()) {

if(!pilot.isMoving()) {

pilot.rotate(-20, true);

}

if(!touch.rightIsPressed()) {

pilot.stop();

}

}

}

//Move away from the edge

pilot.travel(-20);

}

//find an edge, face that edge, rotate at a right angle, then find another edge.

//sets up starting position

public static void findCorner(DifferentialPilot pilot, TouchSensors touch, Gyro gyro, Eyes eyes) {

//RETURNINGFROMCUP simply makes sure that the robot does not get "interrupted"

//on its journey to a corner. Otherwise, the robot can find a cup halfway through, then

//return to only complete half of the findCorner function

while(RETURNINGFROMCUP) {

RETURNINGFROMCUP = false;

findEdge(pilot, touch);

while(pilot.isMoving()) {} //wait

rotate(pilot, gyro, 90, eyes, touch);

if(!RETURNINGFROMCUP) {

findEdge(pilot, touch);

while(pilot.isMoving()) {} //wait

rotate(pilot, gyro, 90, eyes, touch);

}

}

}

//find an edge then face it

public static void findEdge(DifferentialPilot pilot, TouchSensors touch) {

pilot.forward();

while(pilot.isMoving()) {

//if either sensor falls off an edge

if(!(touch.leftIsPressed() && touch.rightIsPressed())) {

//stop, then face that edge

pilot.stop();

flush(pilot, touch);

}

}

}

//prints out the distance an object is from the ultrasonic sensor

//used for debugging and scan range calibration

public static void testEyes(Eyes eyes) {

for(int i = 0; i < 10; i++) {

try {

TimeUnit.SECONDS.sleep(2);

float distance = eyes.getDistance();

System.out.println(distance);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

//rotates the robot, scanning for cups in the process

//robot will rotate a large amount, then stop once a certain distance has been travelled

//or if a cup has been found

public static boolean rotate(DifferentialPilot pilot, Gyro gyro, double angle, Eyes eyes, TouchSensors touch) {

gyro.reset();

float start = gyro.getDistance();

pilot.rotate(BIG\*angle, true);

float distance = eyes.getDistance();

//use absolute value to check the angle traveled

//stop when angle-5 is approached (acceleration and gyroscope error accounts for the other 5 degrees

//OR when the eyes see something within .4 units

while(Math.abs((gyro.getDistance() - start)) < Math.abs(angle)-5 && !(distance > 0 && distance < .4)) {

distance = eyes.getDistance();

System.out.println(Math.abs((gyro.getDistance() - start)));

};

pilot.stop();

//if a cup is within scanning range

if(distance > 0 && distance < .4) {

//push it off the edge!

Sound.beep();

RETURNINGFROMCUP = true;

findEdge(pilot, touch);

pilot.travel(-25);

return true;

}

return false;

}

}

**6.2 Eyes.java**

Eyes.java

package cups;

import lejos.robotics.SampleProvider;

import lejos.robotics.filter.MeanFilter;

import lejos.hardware.ev3.LocalEV3;

import lejos.hardware.port.Port;

import lejos.hardware.sensor.EV3UltrasonicSensor;

import lejos.hardware.sensor.SensorModes;

public class Eyes {

Port port;

SensorModes sensor;

public Eyes() {

port = LocalEV3.get().getPort("S2");

sensor = new EV3UltrasonicSensor(port);

}

//simplified function to retrieved how far away an object is

public float getDistance() {

SampleProvider distance= sensor.getMode("Distance");

SampleProvider average = new MeanFilter(distance, 5);

float[] sample = new float[distance.sampleSize()];

average.fetchSample(sample, 0);

return sample[0];

}

}

**6.3 Gyro.java**

Gyro.java

package cups;

import lejos.hardware.ev3.LocalEV3;

import lejos.hardware.port.Port;

import lejos.hardware.sensor.EV3GyroSensor;

import lejos.robotics.SampleProvider;

public class Gyro {

Port port;

EV3GyroSensor sensor;

public Gyro() {

port = LocalEV3.get().getPort("S3");

sensor = new EV3GyroSensor(port);

}

//simplified function to retrieve how far the gyro has rotated

public float getDistance() {

SampleProvider distance= sensor.getAngleMode();

float[] sample = new float[1];

distance.fetchSample(sample, 0);

return sample[0];

}

public void reset() {

sensor.reset();

}

}

**6.1 TouchSensors.java**

TouchSensors.java

package cups;

import lejos.hardware.Brick;

import lejos.hardware.sensor.EV3TouchSensor;

import lejos.robotics.SampleProvider;

public class TouchSensors {

EV3TouchSensor leftHand, rightHand;

SampleProvider touchLeft, touchRight;

float[] leftSample, rightSample;

public TouchSensors(Brick brick) {

this.leftHand = new EV3TouchSensor(brick.getPort("S4"));

this.rightHand = new EV3TouchSensor(brick.getPort("S1"));

this.touchLeft = leftHand.getMode("Touch");

this.touchRight = rightHand.getMode("Touch");

this.leftSample = new float[touchLeft.sampleSize()];

this.rightSample = new float[touchRight.sampleSize()];

}

public boolean leftIsPressed() {

touchLeft.fetchSample(leftSample, 0);

if(leftSample[0] == 1) {

return true;

} else {

return false;

}

}

public boolean rightIsPressed() {

touchRight.fetchSample(rightSample, 0);

if(rightSample[0] == 1) {

return true;

} else {

return false;

}

}

}