6/25/2023

Προγραμμαρισμός Ευφυών Συστημάτων

Εργασία: αλγόριθμος iBug

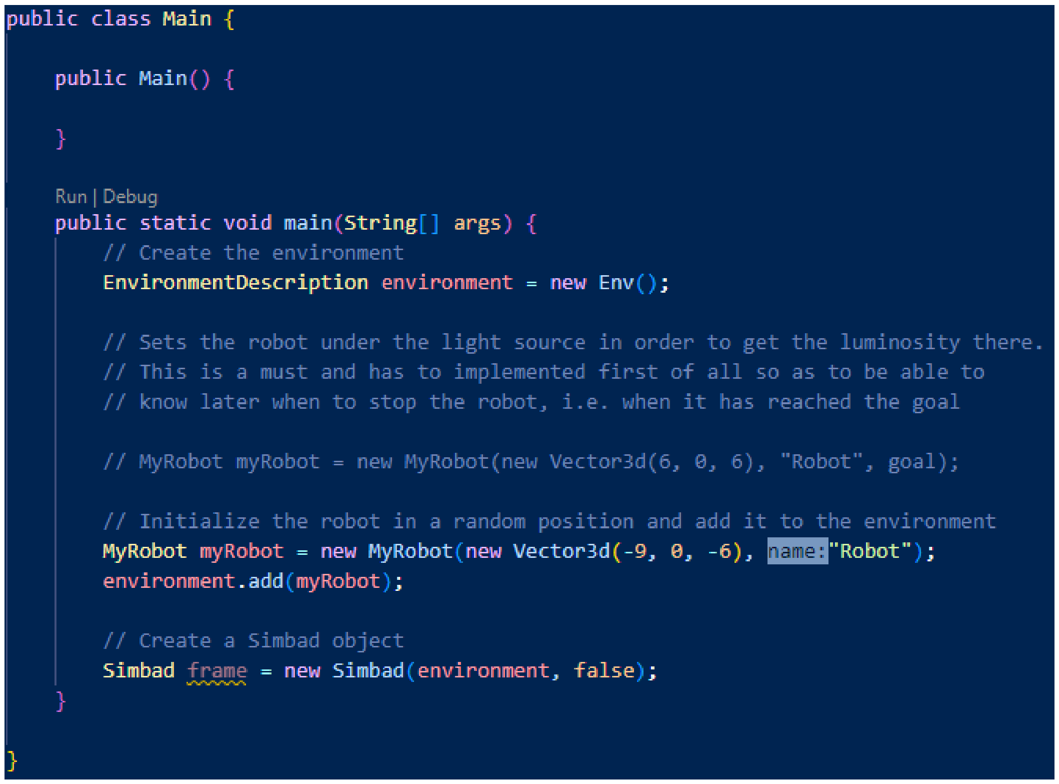
Παλάσκος Αχιλλέας (113)
ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ
ΠΜΣ ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ

Contents

1. Main.java

As we can see from figure 1, the main.java file implements the “**Main**” class, in which only the “main” function is included. This function:

- Creates a new environment instance
- Initializes the robot in a random position and adds it to the environment
- Finally, creates a Simbad object



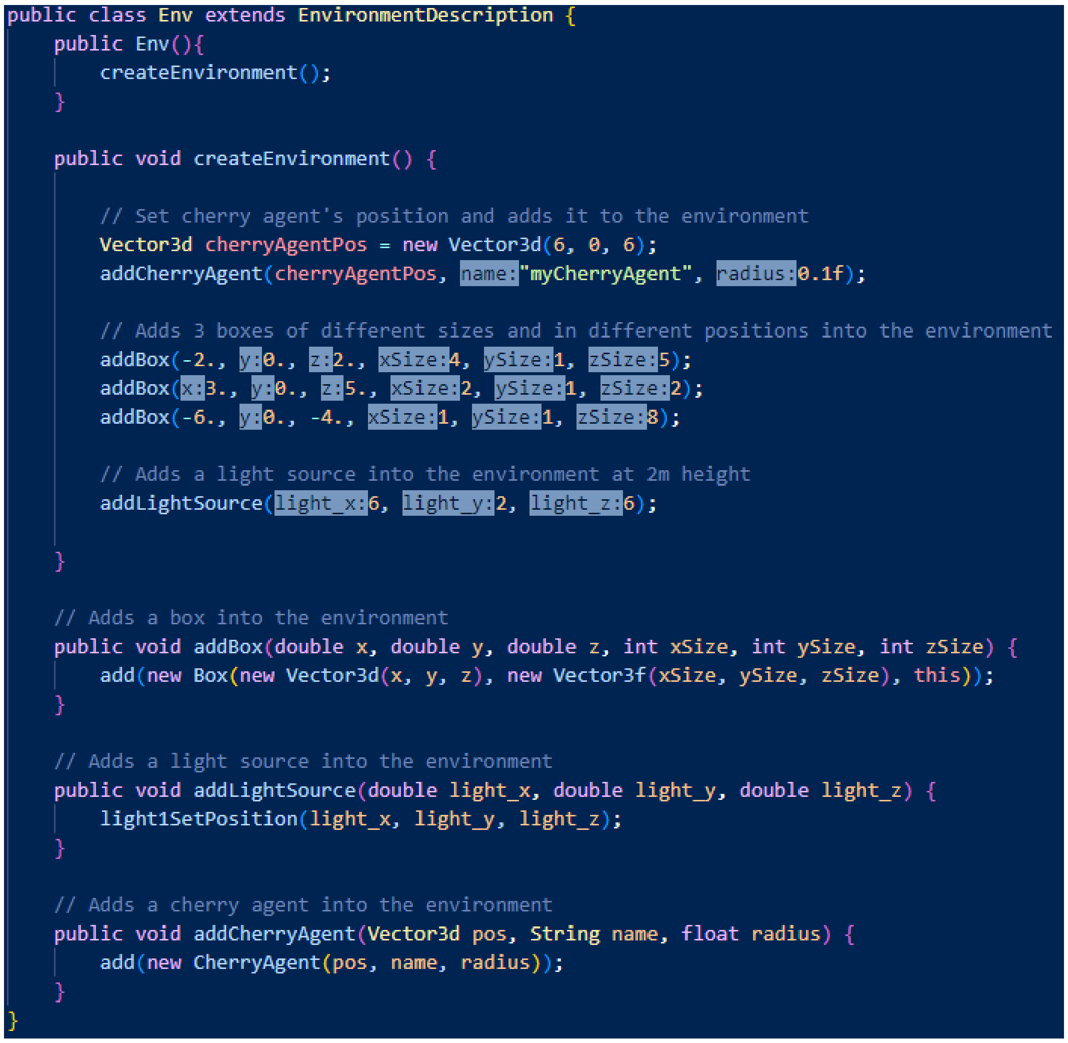
```java
public class Main {

    public Main() {

    }

    Run | Debug
    public static void main(String[] args) {
        // Create the environment
        EnvironmentDescription environment = new Env();

        // Sets the robot under the light source in order to get the luminosity there.
        // This is a must and has to implemented first of all so as to be able to
        // know later when to stop the robot, i.e. when it has reached the goal

        // MyRobot myRobot = new MyRobot(new Vector3d(6, 0, 6), "Robot", goal);

        // Initialize the robot in a random position and add it to the environment
        MyRobot myRobot = new MyRobot(new Vector3d(-9, 0, -6), name:"Robot");
        environment.add(myRobot);

        // Create a Simbad object
        Simbad frame = new Simbad(environment, false);
    }

}
```

Figure 1

2. Env.java

In figure 2 is presented the implementation of the "**Env**" class, which inherits from "EnvironmentDescription" and includes 4 methods:

- The "**addBox**" method, which adds a new Box into the environment.
- The "**addLightSource**" method, which adds a light source into the environment.
- The "**addCherryAgent**" method, which adds a cherry agent into the environment.
- The main method of this class is the "**createEnvironment**", which does the following:
  - Sets cherry agent's position and adds it to the environment.
  - Adds 3 boxes of different sizes and in different positions into the environment.
  - Finally, it adds a light source into the environment in **2m height.**



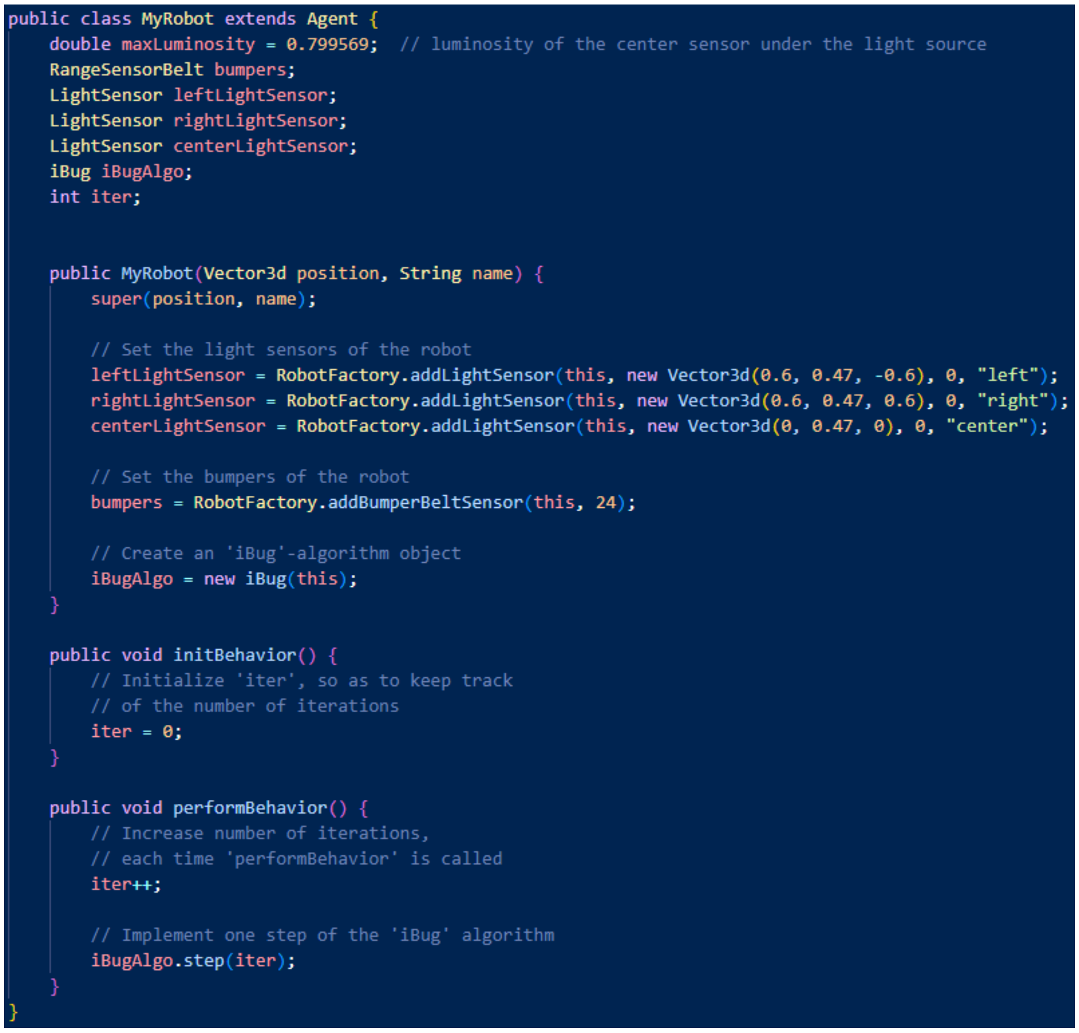
```java
public class Env extends EnvironmentDescription {
    public Env(){
        createEnvironment();
    }

    public void createEnvironment() {

        // Set cherry agent's position and adds it to the environment
        Vector3d cherryAgentPos = new Vector3d(6, 0, 6);
        addCherryAgent(cherryAgentPos, name:"myCherryAgent", radius:0.1f);

        // Adds 3 boxes of different sizes and in different positions into the environment
        addBox(-2., y:0., z:2., xSize:4, ySize:1, zSize:5);
        addBox(x:3., y:0., z:5., xSize:2, ySize:1, zSize:2);
        addBox(-6., y:0., -4., xSize:1, ySize:1, zSize:8);

        // Adds a light source into the environment at 2m height
        addLightSource(light_x:6, light_y:2, light_z:6);

    }

    // Adds a box into the environment
    public void addBox(double x, double y, double z, int xSize, int ySize, int zSize) {
        add(new Box(new Vector3d(x, y, z), new Vector3f(xSize, ySize, zSize), this));
    }

    // Adds a light source into the environment
    public void addLightSource(double light_x, double light_y, double light_z) {
        light1SetPosition(light_x, light_y, light_z);
    }

    // Adds a cherry agent into the environment
    public void addCherryAgent(Vector3d pos, String name, float radius) {
        add(new CherryAgent(pos, name, radius));
    }
}
```

Figure 2

3. MyRobot.java

In figure 3 is presented the implementation of the “**MyRobot**” class, which inherits from “Agent”. Initially, in this class, some variables are declared. How the value for “maxLuminosity” is calculated, will be mentioned later in this assignment. This class includes 3 methods:

- The “**MyRobot**” method (constructor), which:
    - Sets the left, right and center light sensors on the robot **based on the code you gave us**.
    - Sets the bumpers of the robot.
    - Creates an iBug object.
- The “**initBehavior**” method, which only initializes the “iter” variable that keeps track of the iterations of the that the iBug algorithm has run. **It was used only for debugging purposes**.
- The “**performBehavior**” method, increases the “iter” variable by one and the runs one step of the iBug algorithm.



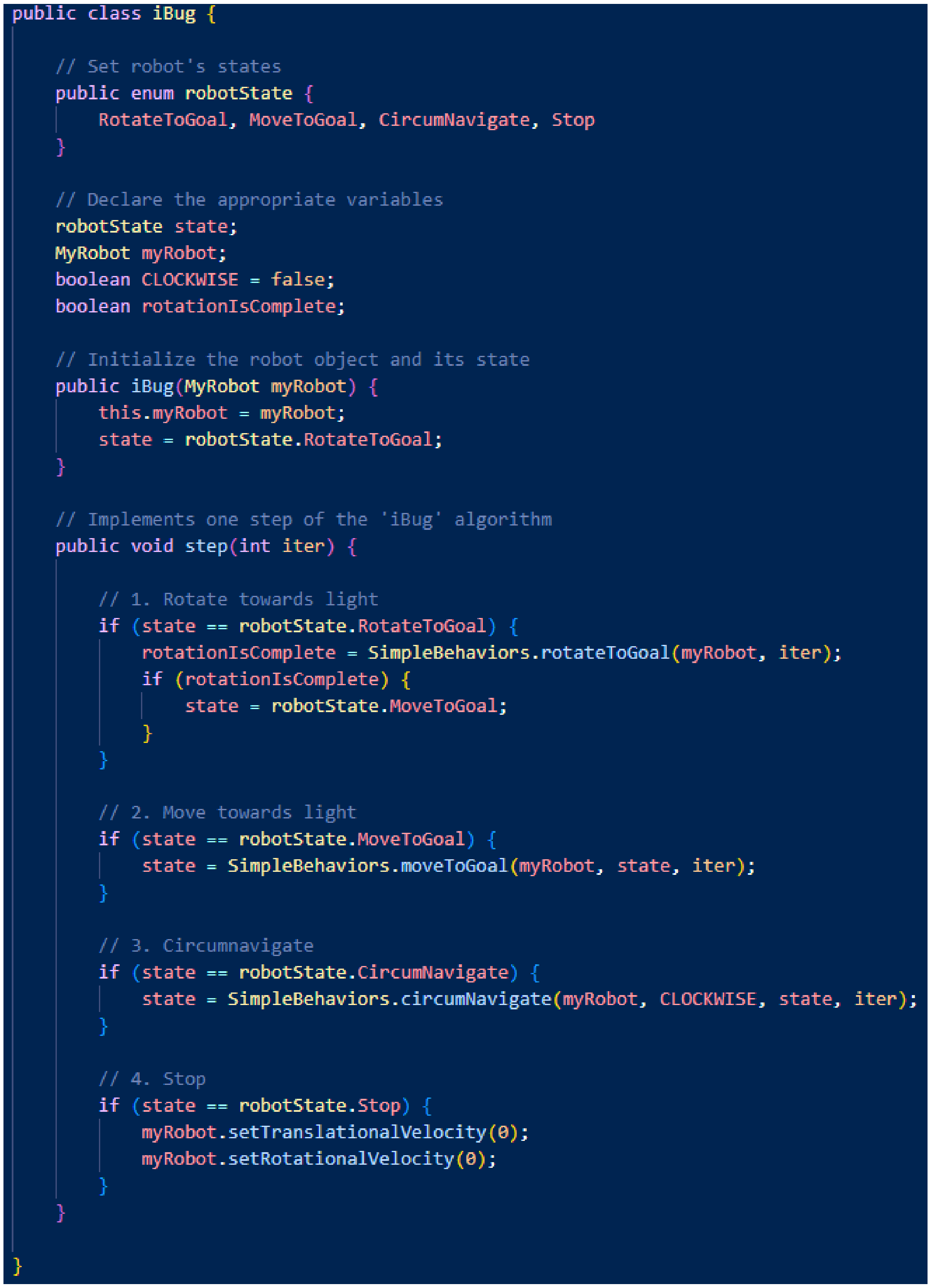
```java
public class MyRobot extends Agent {
    double maxLuminosity = 0.799569;  // luminosity of the center sensor under the light source
    RangeSensorBelt bumpers;
    LightSensor leftLightSensor;
    LightSensor rightLightSensor;
    LightSensor centerLightSensor;
    iBug iBugAlgo;
    int iter;


    public MyRobot(Vector3d position, String name) {
        super(position, name);

        // Set the light sensors of the robot
        leftLightSensor = RobotFactory.addLightSensor(this, new Vector3d(0.6, 0.47, -0.6), 0, "left");
        rightLightSensor = RobotFactory.addLightSensor(this, new Vector3d(0.6, 0.47, 0.6), 0, "right");
        centerLightSensor = RobotFactory.addLightSensor(this, new Vector3d(0, 0.47, 0), 0, "center");

        // Set the bumpers of the robot
        bumpers = RobotFactory.addBumperBeltSensor(this, 24);

        // Create an 'iBug'-algorithm object
        iBugAlgo = new iBug(this);
    }

    public void initBehavior() {
        // Initialize 'iter', so as to keep track
        // of the number of iterations
        iter = 0;
    }

    public void performBehavior() {
        // Increase number of iterations,
        // each time 'performBehavior' is called
        iter++;

        // Implement one step of the 'iBug' algorithm
        iBugAlgo.step(iter);
    }
}
```

Figure 3

4. iBug.java

In figure 4 is presented the implementation of the “**iBug**” class. Initially, in this class, some variables are declared, among which is the “roboState” that all the possible states of the robot. This class includes 2 methods:

- The “**iBug**” method (constructor), which:
    - Initializes the state of the robot in “**RotateToGoal**”. This is because, the first step the robot has to do is to rotate facing the light source.
    - Creates the “**myRobot**” object.
- The “**step**” method, which changes the state of the robot based on its current state. In particular:
    - If the current state of the robot is “**RotateToGoal**”, the “**rotateToGoal**” function is called and if its rotation is complete, then its state changes to “**MoveToGoal**”, so as to move forward towards the goal.
    - If the current state of the robot is “**MoveToGoal**”, the “**moveToGoal**” function is called and its state changes accordingly.
    - If the current state of the robot is “**CircumNavigate**”, the “**circumNavigate**” function is called and its state changes accordingly.
    - If the current state of the robot is “Stop”, then both the translational and rotational velocities are set to 0.



```java
public class iBug {

    // Set robot's states
    public enum robotState {
        RotateToGoal, MoveToGoal, CircumNavigate, Stop
    }

    // Declare the appropriate variables
    robotState state;
    MyRobot myRobot;
    boolean CLOCKWISE = false;
    boolean rotationIsComplete;

    // Initialize the robot object and its state
    public iBug(MyRobot myRobot) {
        this.myRobot = myRobot;
        state = robotState.RotateToGoal;
    }

    // Implements one step of the 'iBug' algorithm
    public void step(int iter) {

        // 1. Rotate towards light
        if (state == robotState.RotateToGoal) {
            rotationIsComplete = SimpleBehaviors.rotateToGoal(myRobot, iter);
            if (rotationIsComplete) {
                state = robotState.MoveToGoal;
            }
        }

        // 2. Move towards light
        if (state == robotState.MoveToGoal) {
            state = SimpleBehaviors.moveToGoal(myRobot, state, iter);
        }

        // 3. Circumnavigate
        if (state == robotState.CircumNavigate) {
            state = SimpleBehaviors.circumNavigate(myRobot, CLOCKWISE, state, iter);
        }

        // 4. Stop
        if (state == robotState.Stop) {
            myRobot.setTranslationalVelocity(0);
            myRobot.setRotationalVelocity(0);
        }
    }

}
```

Figure 4

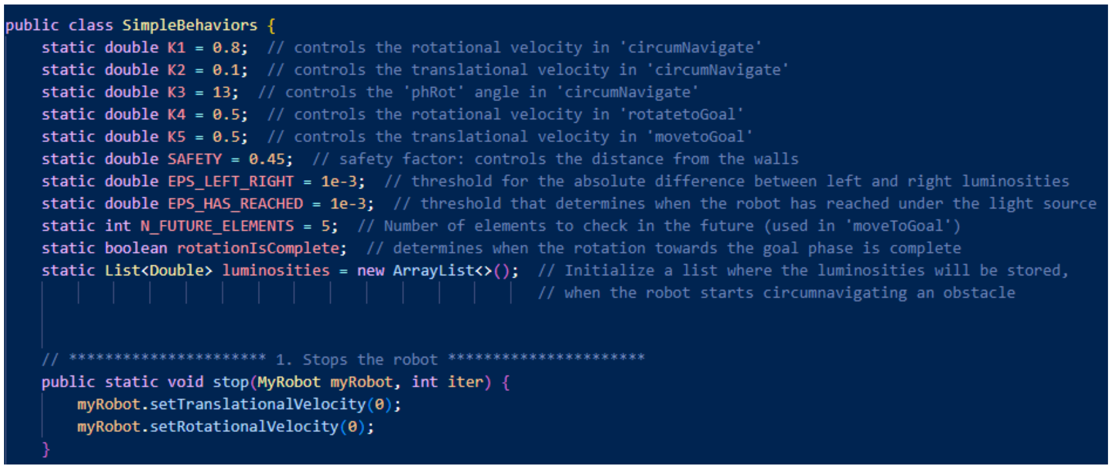
5. SimpleBehaviors.java

a. Variable declaration & initialization and “**stop**” method

In figure 5 is presented the first part of the implementation of the “**SimpleBehaviors**” class. Initially, in this class, the following variables are declared and initialized as shown in figure 5:

- “**K1**”, which controls the rotational velocity in ‘**circumNavigate’**.
- “**K2**”, which controls the translational velocity in ‘**circumNavigate’**.
- “**K3**”, which controls the ‘**phRot’** angle in ‘**circumnavigate’**.
- “**K4**”, which controls the rotational velocity ‘**rotateToGoal’**.
- “**K5**”, which controls the translational velocity in ‘**moveToGoal’**.
- “**SAFETY**” is a safety factor that controls the minimum distance from the walls.
- “**EPS_HAS_REACHED**” is a threshold that determines when the robot has reached under the light source.
- “**EPS_LEFT_RIGHT**” is a threshold for the absolute difference between left and right luminosities.
- “**N_FUTURE_ELEMENTS**” is the number of future elements to be checked in the future (used in ‘**moveToGoal**’).
- “**rotationIsComplete** determines when the rotation, towards the goal, phase is complete.
- “**luminosities**” is a list where the luminosities will be stored, when the robot starts circumnavigating an obstacle.

The “**stop**” method sets both the translational and rotational velocities to 0.



```java
public class SimpleBehaviors {
    static double K1 = 0.8;  // controls the rotational velocity in 'circumNavigate'
    static double K2 = 0.1;  // controls the translational velocity in 'circumNavigate'
    static double K3 = 13;  // controls the 'phRot' angle in 'circumNavigate'
    static double K4 = 0.5;  // controls the rotational velocity in 'rotatetoGoal'
    static double K5 = 0.5;  // controls the translational velocity in 'movetoGoal'
    static double SAFETY = 0.45;  // safety factor: controls the distance from the walls
    static double EPS_LEFT_RIGHT = 1e-3;  // threshold for the absolute difference between left and right luminosities
    static double EPS_HAS_REACHED = 1e-3;  // threshold that determines when the robot has reached under the light source
    static int N_FUTURE_ELEMENTS = 5;  // Number of elements to check in the future (used in 'moveToGoal')
    static boolean rotationIsComplete;  // determines when the rotation towards the goal phase is complete
    static List<Double> luminosities = new ArrayList<>();  // Initialize a list where the luminosities will be stored,
                                                           // when the robot starts circumnavigating an obstacle


    // ********************** 1. Stops the robot **********************
    public static void stop(MyRobot myRobot, int iter) {
        myRobot.setTranslationalVelocity(0);
        myRobot.setRotationalVelocity(0);
    }
```

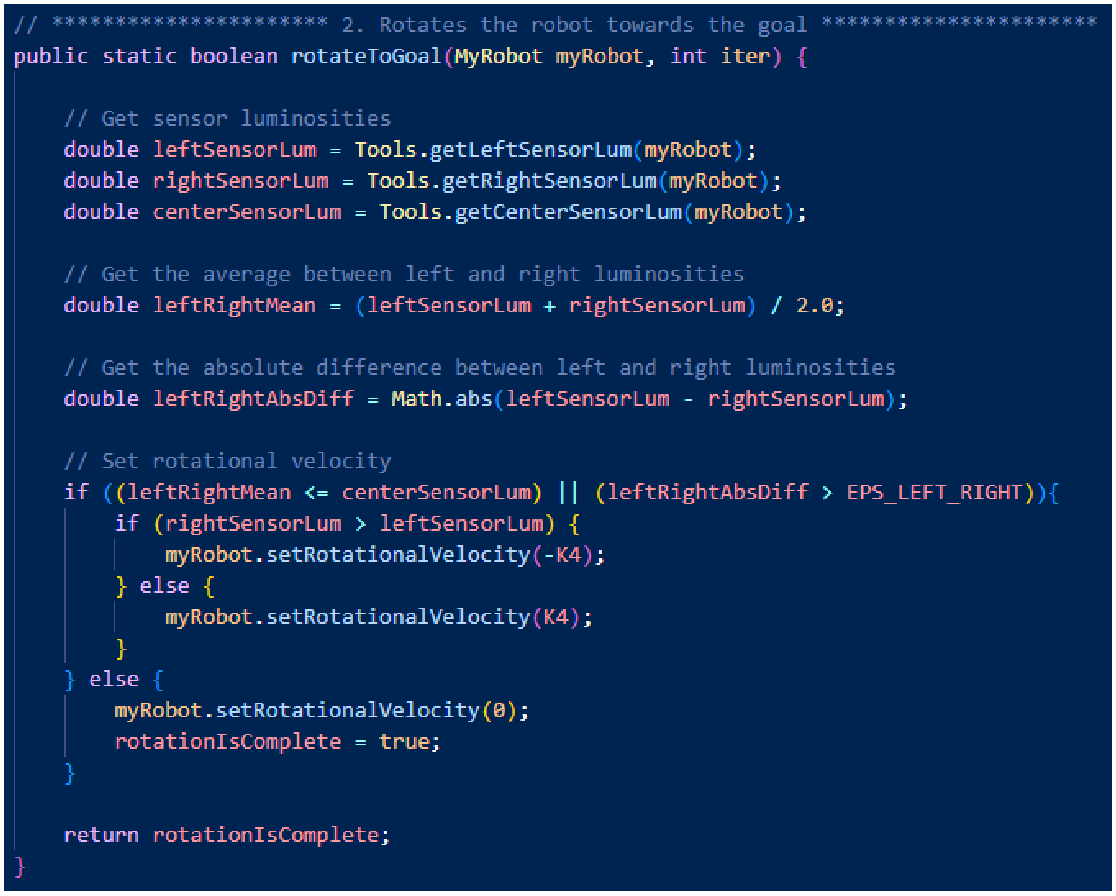
Figure 5

b. The "**rotateToGoal**" method

This method rotates the robot so as it 'look' towards the right direction, i.e. towards the light source (goal). First after it calculates the left, right and center luminosities, it finds the mean between the left and right luminosities as well as the absolute difference between them. **The key idea here is that when the robot faces the right direction the left and the right luminosities should be almost the same AND their mean should be less than the luminosity given by the center sensor (because the center sensor is nearer the light source).** This is checked in condition:

```
if ((leftRightMean <= centerSensorLum) || (leftRightAbsDiff > EPS_LEFT_RIGHT))
```

If the above condition is true, then, depending on which of the left and right luminosities is greater, I determine whether the robot should be rotated clockwise or counter-clockwise. If the condition is false, then this means that the rotation is complete and therefore the rotational velocity is set to zero and the Boolean variable 'rotationIsComplete' to true.



```java
// ********************** 2. Rotates the robot towards the goal **********************
public static boolean rotateToGoal(MyRobot myRobot, int iter) {

    // Get sensor luminosities
    double leftSensorLum = Tools.getLeftSensorLum(myRobot);
    double rightSensorLum = Tools.getRightSensorLum(myRobot);
    double centerSensorLum = Tools.getCenterSensorLum(myRobot);

    // Get the average between left and right luminosities
    double leftRightMean = (leftSensorLum + rightSensorLum) / 2.0;

    // Get the absolute difference between left and right luminosities
    double leftRightAbsDiff = Math.abs(leftSensorLum - rightSensorLum);

    // Set rotational velocity
    if ((leftRightMean <= centerSensorLum) || (leftRightAbsDiff > EPS_LEFT_RIGHT)){
        if (rightSensorLum > leftSensorLum) {
            myRobot.setRotationalVelocity(-K4);
        } else {
            myRobot.setRotationalVelocity(K4);
        }
    } else {
        myRobot.setRotationalVelocity(0);
        rotationIsComplete = true;
    }

    return rotationIsComplete;
}
```

Figure 6

c. The “**moveToGoal**” method

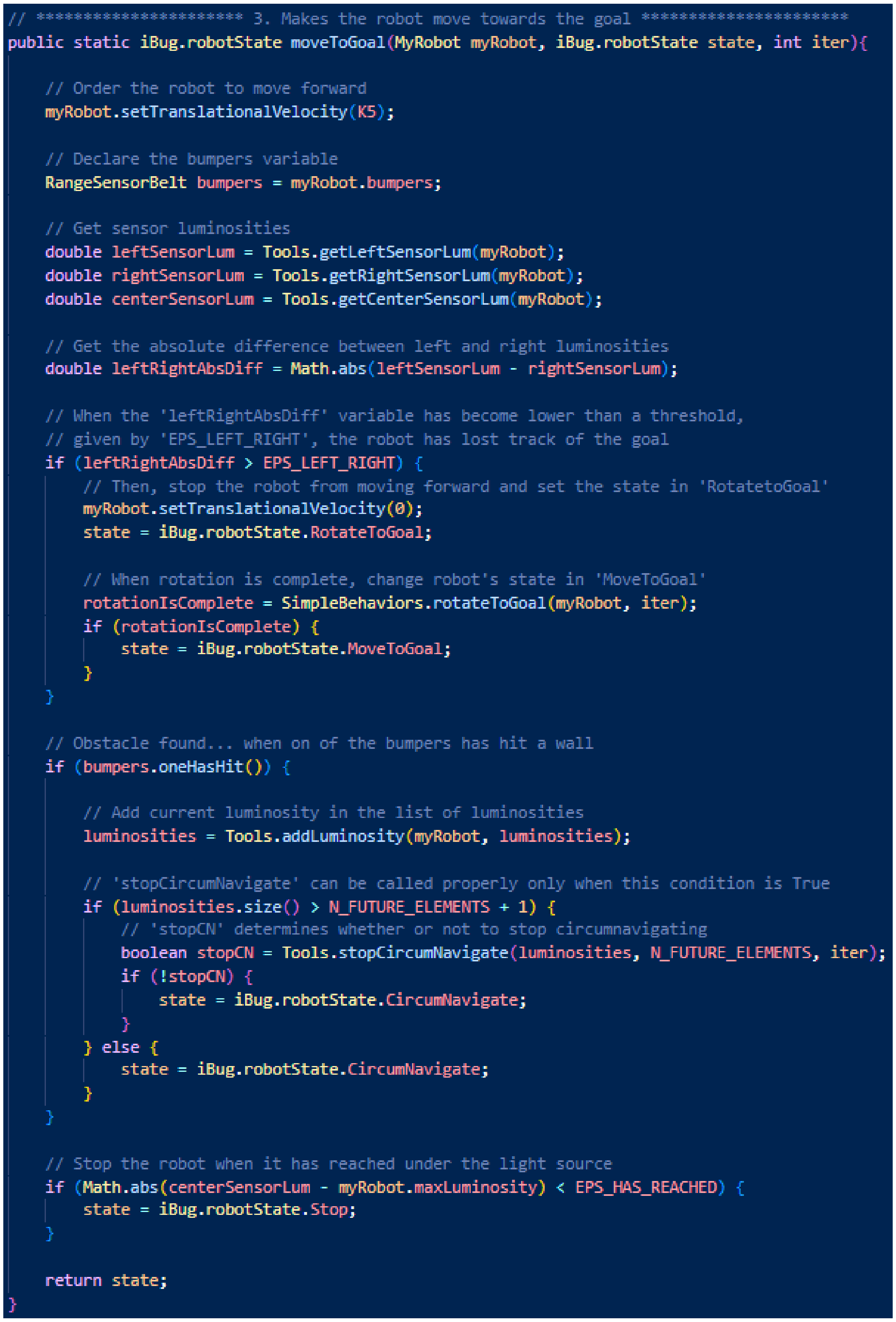
This method, which is shown in figure 7, is run only when the rotation towards the goal is finished. For this reason the first thing to do is to set the translational velocity. Then the bumpers object is created and the left, right and center luminosities are calculated. The absolute difference between the right and left luminosities will also be need later, therefore it is calculated as well.

The first thing to check is whether the robot has lost track of the goal. **This is true when the absolute difference between the right and left luminosities has become lower than a threshold** “**EPS_LEFT_RIGHT**”. In this case I set the translational velocity to 0 and the state to “**RotateToGoal**” and the “**rotateToGoal**” method is called. When the rotation is complete the state is again set to “**MoveToGoal**”.

The next thing to consider is whether an obstacle is found. **This happens when at least one of the bumpers has hit.** When that happens I add the current luminosity in the list of luminosities, which will be used by the “**stopCircumNavigate**” function, which is called next. This function determines when to stop circumnavigating the obstacle and will be explained later how it works. While the “**stopCN**” variable is False, the robot keeps circumnavigating.

Finally, it has also to be checked if the object has reached the goal. This happens when the luminosity given by the center sensor has come ‘close enough’ to the maximum possible luminosity. Here:

- ‘close enough’ means that the absolute difference between the 2 aforementioned quantities has dropped below a predetermined threshold, which is the “**EPS_HAS_REACHED**”.
- **the maximum possible luminosity was calculated by setting the robot (before the main simulation starts) under the light source and calculating the receiving luminosity there**



```java
// ********************** 3. Makes the robot move towards the goal **********************
public static iBug.robotState moveToGoal(MyRobot myRobot, iBug.robotState state, int iter){

    // Order the robot to move forward
    myRobot.setTranslationalVelocity(K5);

    // Declare the bumpers variable
    RangeSensorBelt bumpers = myRobot.bumpers;

    // Get sensor luminosities
    double leftSensorLum = Tools.getLeftSensorLum(myRobot);
    double rightSensorLum = Tools.getRightSensorLum(myRobot);
    double centerSensorLum = Tools.getCenterSensorLum(myRobot);

    // Get the absolute difference between left and right luminosities
    double leftRightAbsDiff = Math.abs(leftSensorLum - rightSensorLum);

    // When the 'leftRightAbsDiff' variable has become lower than a threshold,
    // given by 'EPS_LEFT_RIGHT', the robot has lost track of the goal
    if (leftRightAbsDiff > EPS_LEFT_RIGHT) {
        // Then, stop the robot from moving forward and set the state in 'RotatetoGoal'
        myRobot.setTranslationalVelocity(0);
        state = iBug.robotState.RotateToGoal;

        // When rotation is complete, change robot's state in 'MoveToGoal'
        rotationIsComplete = SimpleBehaviors.rotateToGoal(myRobot, iter);
        if (rotationIsComplete) {
            state = iBug.robotState.MoveToGoal;
        }
    }

    // Obstacle found... when on of the bumpers has hit a wall
    if (bumpers.oneHasHit()) {

        // Add current luminosity in the list of luminosities
        luminosities = Tools.addLuminosity(myRobot, luminosities);

        // 'stopCircumNavigate' can be called properly only when this condition is True
        if (luminosities.size() > N_FUTURE_ELEMENTS + 1) {
            // 'stopCN' determines whether or not to stop circumnavigating
            boolean stopCN = Tools.stopCircumNavigate(luminosities, N_FUTURE_ELEMENTS, iter);
            if (!stopCN) {
                state = iBug.robotState.CircumNavigate;
            }
        } else {
            state = iBug.robotState.CircumNavigate;
        }
    }

    // Stop the robot when it has reached under the light source
    if (Math.abs(centerSensorLum - myRobot.maxLuminosity) < EPS_HAS_REACHED) {
        state = iBug.robotState.Stop;
    }

    return state;
}
```

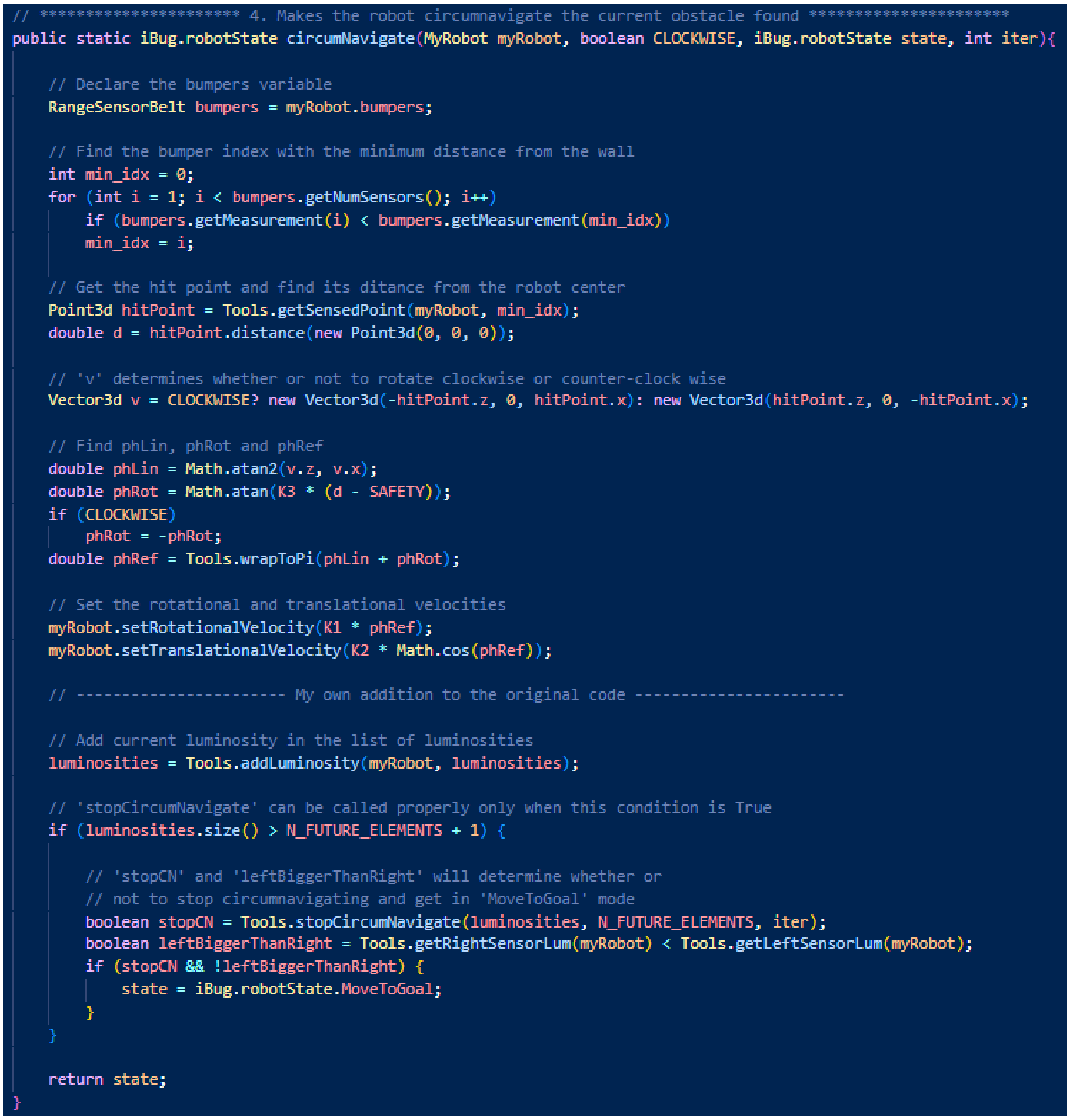
Figure 7

d. The “**circumNavigate**” method

This function is mainly the same with your own “circumNavigate” function. Only in the last part of it I have added my own code. At the beginning, the bumpers object is created and are found the bumper with the minimum distance from the obstacle, the coordinates of the hit point and its distance from the center of the robot. Then the v variable is calculated and determines whether or not to rotate clockwise or counter-clockwise. The final part of the original code calculates the φ_{lin}, φ_{rot} and φ_{ref} from which the final angle determines both the rotational and translational velocities that governs the circumnavigation.

After that, I added my own code, which determines when should the robot quit from circumnavigation. To implement this, first I add the current luminosity in the list of luminosities, which will be used by the “**stopCircumNavigate**” function, which is called next. This function determines when to stop circumnavigating the obstacle and will be explained later how it works. While the “**stopCN**” variable is False, the robot keeps circumnavigating. The robot stops circumnavigating when the following 2 conditions hold:

1. The “**stopCN**” is true
2. The luminosity o the right sensor is bigger the luminosity of the left sensor. This is because the robot circumnavigates the obstacle in counter-clockwise mode. In other words, I want the object to be “in front of” the object and not behind it.



```java
// ********************** 4. Makes the robot circumnavigate the current obstacle found **********************
public static iBug.robotState circumNavigate(MyRobot myRobot, boolean CLOCKWISE, iBug.robotState state, int iter){

    // Declare the bumpers variable
    RangeSensorBelt bumpers = myRobot.bumpers;

    // Find the bumper index with the minimum distance from the wall
    int min_idx = 0;
    for (int i = 1; i < bumpers.getNumSensors(); i++)
        if (bumpers.getMeasurement(i) < bumpers.getMeasurement(min_idx))
            min_idx = i;

    // Get the hit point and find its ditance from the robot center
    Point3d hitPoint = Tools.getSensedPoint(myRobot, min_idx);
    double d = hitPoint.distance(new Point3d(0, 0, 0));

    // 'v' determines whether or not to rotate clockwise or counter-clock wise
    Vector3d v = CLOCKWISE? new Vector3d(-hitPoint.z, 0, hitPoint.x): new Vector3d(hitPoint.z, 0, -hitPoint.x);

    // Find phLin, phRot and phRef
    double phLin = Math.atan2(v.z, v.x);
    double phRot = Math.atan(K3 * (d - SAFETY));
    if (CLOCKWISE)
        phRot = -phRot;
    double phRef = Tools.wrapToPi(phLin + phRot);

    // Set the rotational and translational velocities
    myRobot.setRotationalVelocity(K1 * phRef);
    myRobot.setTranslationalVelocity(K2 * Math.cos(phRef));

    // ---------------------- My own addition to the original code ----------------------

    // Add current luminosity in the list of luminosities
    luminosities = Tools.addLuminosity(myRobot, luminosities);

    // 'stopCircumNavigate' can be called properly only when this condition is True
    if (luminosities.size() > N_FUTURE_ELEMENTS + 1) {

        // 'stopCN' and 'leftBiggerThanRight' will determine whether or
        // not to stop circumnavigating and get in 'MoveToGoal' mode
        boolean stopCN = Tools.stopCircumNavigate(luminosities, N_FUTURE_ELEMENTS, iter);
        boolean leftBiggerThanRight = Tools.getRightSensorLum(myRobot) < Tools.getLeftSensorLum(myRobot);
        if (stopCN && !leftBiggerThanRight) {
            state = iBug.robotState.MoveToGoal;
        }
    }

    return state;
}
```

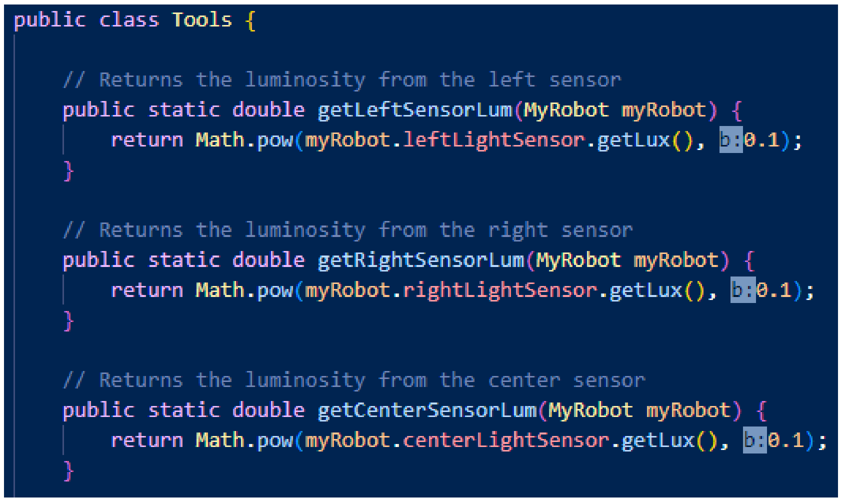
Figure 8

6. Tools.java

In the Tools class some useful functions are implemented.

6.1 The “getLeftSensorLum”, “getRightSensorLum” and “getCenterSensorLum” methods

These 3 methods of the Tools class return the luminosity of the left, right and center sensors of the robot respectively. The corresponding code is shown in figure 9.

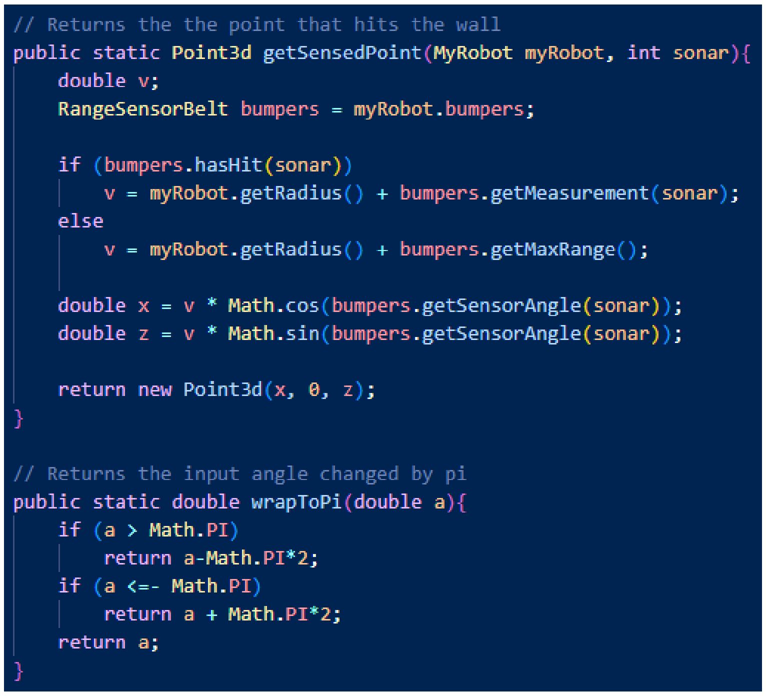


```java
public class Tools {

    // Returns the luminosity from the left sensor
    public static double getLeftSensorLum(MyRobot myRobot) {
        return Math.pow(myRobot.leftLightSensor.getLux(), b:0.1);
    }

    // Returns the luminosity from the right sensor
    public static double getRightSensorLum(MyRobot myRobot) {
        return Math.pow(myRobot.rightLightSensor.getLux(), b:0.1);
    }

    // Returns the luminosity from the center sensor
    public static double getCenterSensorLum(MyRobot myRobot) {
        return Math.pow(myRobot.centerLightSensor.getLux(), b:0.1);
    }
```

Figure 9

6.2 The “getSensedPoint” and “wrapToPi” methods

These methods of the Tools class are you own code (NOT implemented by me) and shown in figure 10.



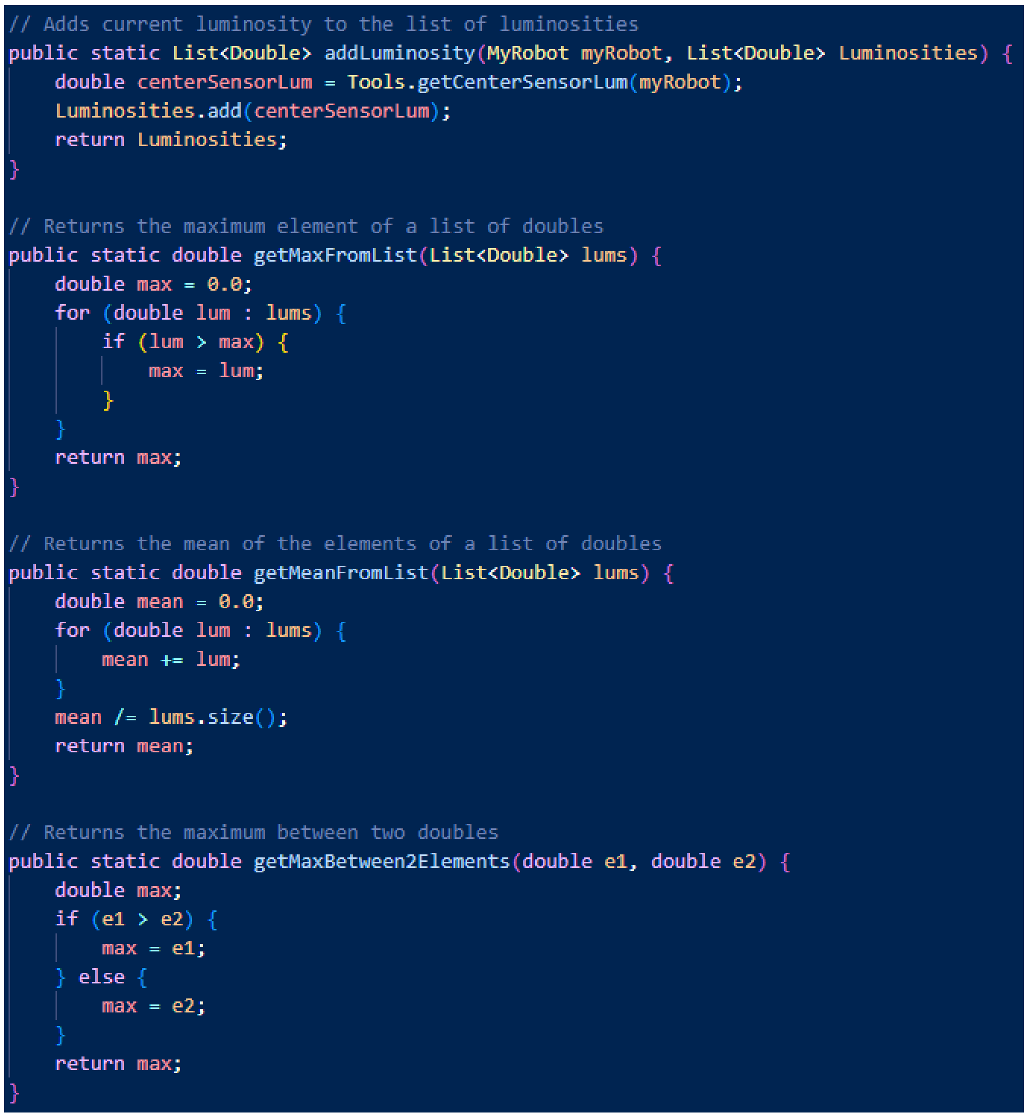
```java
// Returns the the point that hits the wall
public static Point3d getSensedPoint(MyRobot myRobot, int sonar){
    double v;
    RangeSensorBelt bumpers = myRobot.bumpers;

    if (bumpers.hasHit(sonar))
        v = myRobot.getRadius() + bumpers.getMeasurement(sonar);
    else
        v = myRobot.getRadius() + bumpers.getMaxRange();

    double x = v * Math.cos(bumpers.getSensorAngle(sonar));
    double z = v * Math.sin(bumpers.getSensorAngle(sonar));

    return new Point3d(x, 0, z);
}

// Returns the input angle changed by pi
public static double wrapToPi(double a){
    if (a > Math.PI)
        return a-Math.PI*2;
    if (a <=- Math.PI)
        return a + Math.PI*2;
    return a;
}
```

Figure 10

6.3 The “addLuminosity”, “getMaxFromList”, “getMeanFromList” and “getMaxBetween2Elements” methods

These 4 methods, shown in figure 11, implement the following:

- “**addLuminosity**” adds the luminosity of the center sensor of the robot to the list of luminosities.
- “**getMaxFromList**” returns the maximum element of a list of doubles.
- “**getMeanFromList**” returns the mean of the elements of a list of doubles.
- “**getMaxBetween3Elements**” returns the maximum between 2 doubles.



```java
// Adds current luminosity to the list of luminosities
public static List<Double> addLuminosity(MyRobot myRobot, List<Double> Luminosities) {
    double centerSensorLum = Tools.getCenterSensorLum(myRobot);
    Luminosities.add(centerSensorLum);
    return Luminosities;
}

// Returns the maximum element of a list of doubles
public static double getMaxFromList(List<Double> lums) {
    double max = 0.0;
    for (double lum : lums) {
        if (lum > max) {
            max = lum;
        }
    }
    return max;
}

// Returns the mean of the elements of a list of doubles
public static double getMeanFromList(List<Double> lums) {
    double mean = 0.0;
    for (double lum : lums) {
        mean += lum;
    }
    mean /= lums.size();
    return mean;
}

// Returns the maximum between two doubles
public static double getMaxBetween2Elements(double e1, double e2) {
    double max;
    if (e1 > e2) {
        max = e1;
    } else {
        max = e2;
    }
    return max;
}
```

Figure 11

6.4 The “**stopCircumNavigate**” method (**MOST IMPORTANT METHOD…!**)

This is perhaps the most difficult and most important function of the iBug algorithm as implemented by me. First of all, note that the “luminosities” list is always initialized to an empty list when the robot encounters a new object and therefore contains only the luminosities of the center sensor, while the robot is circumnavigating the obstacle. Then I do the following:

- First, I create 2 lists with the first ‘**lumsSize - N_FUTURE_ELEMENTS**’ and ‘**lumsSize - N_FUTURE_ELEMENTS – 1**’ elements respectively. After that, I get the maximum element from the ‘**pastLums1’** and ‘**pastLums2’** lists. Finally, I calculate the maximum between the two.
- Next, I create a list with the last ‘**N_FUTURE_ELEMENTS**’ elements and calculate the mean of the ‘**futureLums**’ list.
- Finally, the robot should stop circumnavigating if ‘**futureLumsMean**’ is less than ‘**pastLumsMax**’.

The main idea behind the previous calculations can be explained more clearly given the following figure (figure 12), which depicts how the luminosities of the center sensor of the robot change while it is circumnavigating the obstacle. As it is shown in figure 12, when the robot is in front of the obstacle (between the light source and the obstacle), the luminosity hits a peak. On the other hand, when the robot is behind the object (with respect to the light source) the corresponding luminosity declines. Therefore, the previous code tries to ‘catch’ when the robot is at the peak **but without making a full circle of the object**.

In order to achieve this, I create the variables “pastLumsMax” and “futureLumsMean”. Taking into account how these variables are calculated we can easily observe that:

- as far as we are on the side left to the peak (iterations 400 - 2600):
  $$\text{pastLumsMax} < \text{futureLumsMean} \quad (1)$$
  and therefore the robot should continue circumnavigating.
- But we have to be careful because in the first about 400 iterations (1) doesn’t hold, but the robot should also keep circling the object found. Here comes into play the “leftBiggerThanRight” variable in the “circumNavigate” method of the “SimpleBehaviors” class to fix this problem. How does it achieve that? By indicating whether the robot is in front or at the back of the object (with respect to the light source). In the first 400 iterations (1) violated, thus “stopCircumNavigating” returns true, but the “!leftBiggerThanRight” condition in the “circumNavigate” method ensures that the robot will keep circumnavigating.
- When the robot reaches to the left of the peak (iteration > 2600) then:
  $$\text{pastLumsMax} > \text{futureLumsMean} \quad (2)$$
  and therefore “stopCircumNavigating” returns true and the robot returns to “MoveToGoal” mode.

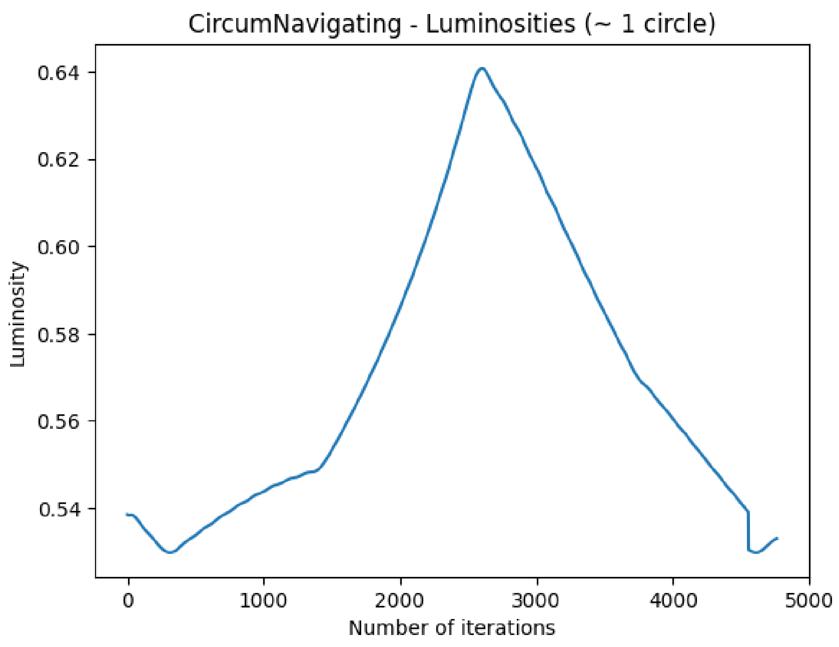
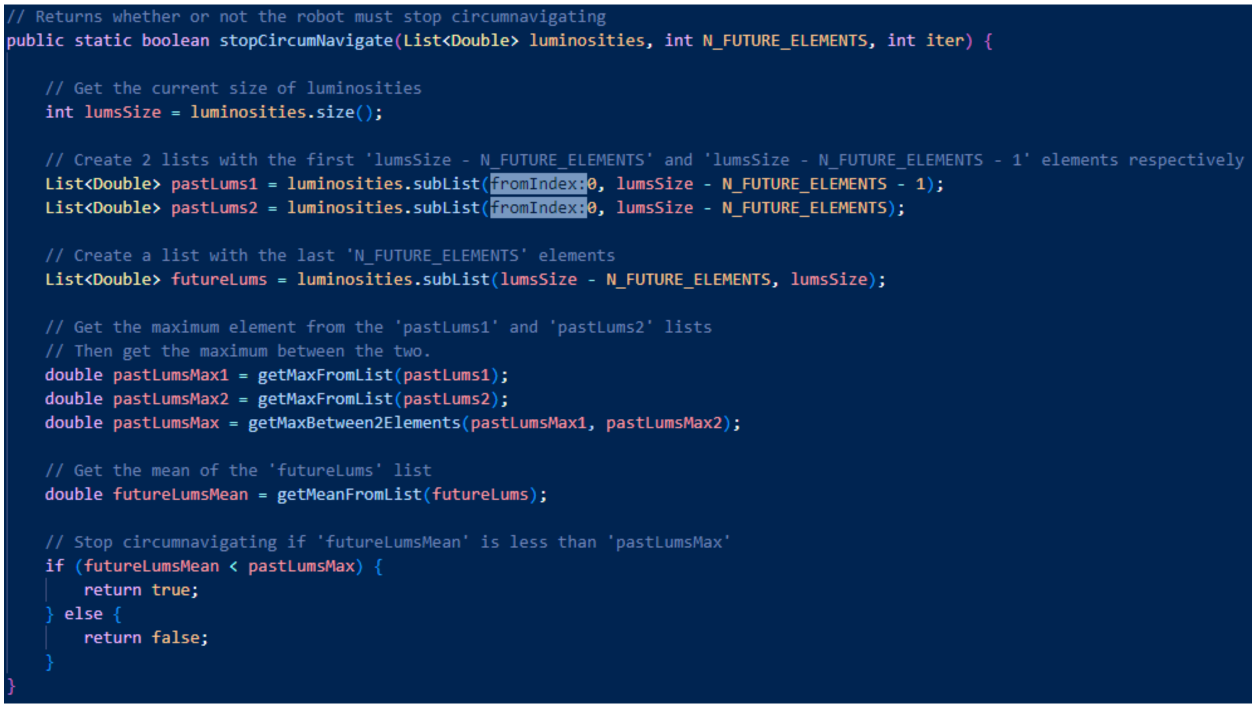


Figure 12



```
// Returns whether or not the robot must stop circumnavigating
public static boolean stopCircumNavigate(List<Double> luminosities, int N_FUTURE_ELEMENTS, int iter) {

    // Get the current size of luminosities
    int lumsSize = luminosities.size();

    // Create 2 lists with the first 'lumsSize - N_FUTURE_ELEMENTS' and 'lumsSize - N_FUTURE_ELEMENTS - 1' elements respectively
    List<Double> pastLums1 = luminosities.subList(fromIndex:0, lumsSize - N_FUTURE_ELEMENTS - 1);
    List<Double> pastLums2 = luminosities.subList(fromIndex:0, lumsSize - N_FUTURE_ELEMENTS);

    // Create a list with the last 'N_FUTURE_ELEMENTS' elements
    List<Double> futureLums = luminosities.subList(lumsSize - N_FUTURE_ELEMENTS, lumsSize);

    // Get the maximum element from the 'pastLums1' and 'pastLums2' lists
    // Then get the maximum between the two.
    double pastLumsMax1 = getMaxFromList(pastLums1);
    double pastLumsMax2 = getMaxFromList(pastLums2);
    double pastLumsMax = getMaxBetween2Elements(pastLumsMax1, pastLumsMax2);

    // Get the mean of the 'futureLums' list
    double futureLumsMean = getMeanFromList(futureLums);

    // Stop circumnavigating if 'futureLumsMean' is less than 'pastLumsMax'
    if (futureLumsMean < pastLumsMax) {
        return true;
    } else {
        return false;
    }
}
```

Figure 13