MINISTRY OF EDUCATION AND TRAINING  
HO CHI MINH CITY OPEN UNIVERSITY

REPORT

ROBOCODE

Team : Chấm

Faculty of Information Technology

Nguyễn Thành Nam (1951052126 - IT93)  
Nguyễn Trung Kiên (1951052091 - IT93)

Ho Chi Minh City, May 2021

Table of Contents

[I. INTRODUCTION ABOUT ROBOCODE 2](#_Toc70764973)

[II. ROBOCODE ACTUALIZATION 4](#_Toc70764974)

[1. METHODS FROM ROBOCODE SOURCE CODE 4](#_Toc70764975)

[2. INTRODUCTION ABOUT TEAM’S ROBOT 6](#_Toc70764976)

[2.1 Concept about robot 6](#_Toc70764977)

[3. IMPLEMENTATION OF THE IDEA 6](#_Toc70764978)

[3.1 GuessFactor Targeting Algorithm 6](#_Toc70764979)

[3.2 Melee Strategy Algorithm 8](#_Toc70764980)

[3.3 Robot class 14](#_Toc70764981)

[3.4 Utility class 15](#_Toc70764982)

[3.5 Movement\_1VS1 class 16](#_Toc70764983)

[3.6 Wave class 18](#_Toc70764984)

[3.7 The methods in Wave class 19](#_Toc70764985)

[3.8 Attributes required of the robot to execute the methods 22](#_Toc70764986)

[3.9 shooting method 23](#_Toc70764987)

[3.10 updateListLocation method 25](#_Toc70764988)

[3.11 evaluatePoint method 26](#_Toc70764989)

[3.12 movement method 29](#_Toc70764990)

[3.13 onRobotDeath method 30](#_Toc70764991)

[3.14 onScannedRobot method 30](#_Toc70764992)

[3.15 run method 34](#_Toc70764993)

[3.16 changeColor method 36](#_Toc70764994)

[III. REFERENCES 37](#_Toc70764995)

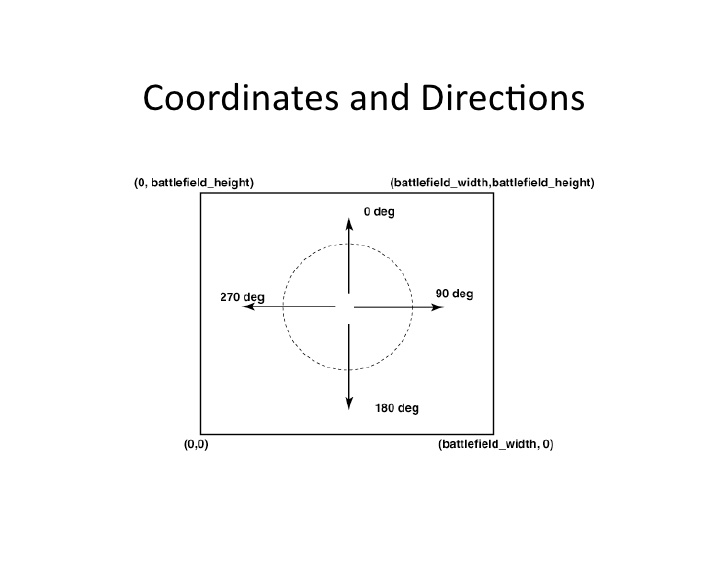
# INTRODUCTION ABOUT ROBOCODE

According to Wikipedia (<http://vi.wikipedia.org/wiki/Robocode>), Robocode is a programming game where the goal is to code a robot to compete against other robots in a battle area. The player is the programmer of the robot, who will have no direct influence on the game. Instead, the player must write the AI of the robot telling it how to behave and react to events occurring in the battle arena. So the name Robocode is short for "Robot code".

The game is designed to help you learn Java, and have fun doing it. Robots are written in the Java programming language, and the Robocode game can run on any operating system supported by the Java Platform, which includes all common operating systems like Windows, macOS, Linux, etc.

Robocode's battles take place on a battlefield, where small automated 6-wheeled robots fight it out until only one is left. Please notice that Robocode contains no gore, no blood, no people, and no politics. The battles are simply for the excitement of competition that we love so much. There are explosions, however, but these can be turned off if they are offending.

**Robocode Interface**

****

**Coordinates and Directions of the battlefield**

# ROBOCODE ACTUALIZATION

Our robot is inherited from the class **AdvancedRobot**.

## METHODS FROM ROBOCODE SOURCE CODE

* **ahead(distance):** The robot moves ahead specific distance in pixels.
* **back(distance):** The robot moves back a specific distance is a pixels.
* **fire(power):** The robot fires a bullet with assigned energy.
* **setAdjustGunForRobotTurn(independent):** Set the gun to turn independently with the body of the robot or not. If the parameter equals ‘true’, the gun will turn independently with the gun and vice versa.
* **setAdjustRadarForGunTurn(independent):** Set the radar to turn independently with the gun of the robot or not. If the parameter equals ‘true’, the radar will turn independently with the gun and vice versa. **setAdjustRadarForRobotTurn(independent):** Set the radar to turn independently with the body of the robot or not. If the parameter equals ‘true’, the radar will turn independently with the body and vice versa.
* **setColors(bodyColor, gunColor, radarColor, bulletColor, scanArcColor):** Set the color for all parts of the robot.
* **getX():** Return the current X position of the robot.
* **getY():** Return the current Y position of the robot. **getBattleFieldWidth():** Return the width of the battle field.
* **getBattleFieldHeight():** Return the height of the battle field.  
  **Note:** The default range of the battlefield is 800x600.
* **setTurnRadarRight(degree):** Set the radar to turn right an angle . If the passing angle is a negative number, the setTurnRadarRight method will turn into the setTurnRadarLeft method.
* **getRadarTurnRemaining():** Check if the radar is turning, unless the action can be performed as shown above.
* **execute():** Execute above commands in the while(true) loop.
* **getEnergy():** Return the current energy of the robot.
* **get heading():** Return the heading angle of the body of the robot . The value of the angle is followed clockwise.
* **getGunHeading():** Return the heading angle of the gun of the robot . The value of the angle is followed clockwise **getRadarHeading():** Return the heading angle of the body of the robot . The value of the angle is followed clockwise.
* **getVelocity():** Return the current velocity of the robot.
* **setAhead(distance):** The robot moves ahead of a specific distance in pixels. If the parameter is a negative number, setAhead method will turn into setBack method.
* **setBack(distance):** The robot moves back a specific distance in pixels. If the parameter is a negative number, setBack method will turn into setAhead method.
* **setTurnRight(rotate):** Control the body of the robot to turn right at a specific angle.
* **setTurnGunRight(rotate):** Control the gun of the robot to turn right at a specific angle.
* **setFire(power):** Set the energy of the bullet when the robot fires. The energy is in the range
* **normalRelativeAngle(angle):** Normalize a relative angle. The method will return an angle which value is in the range
* **In scannedRobotEvent e**:
  + **e.getBearing():** Return an angle in range

If the enemy robot locates on the left side of our robot, the return value will be in the range [0, -180]. Otherwise, this method will return a positive number in the range [0, 180]

* + **Note**: Comparing the location of the opponent robot is on the left side or right side depends on the heading angle of our robot. That means the return value is a relative angle**.**
  + **e.getHeading():** Get the current heading of the opponent robot.
  + **e.getDistance():** Get the distance from the enemy’s location to our robot’s location.
  + **e.getVelocity():** Get the current velocity of the enemy.
* **onScannedRobot(ScannedRobotEvent event)** this method helps us perform some actions when our robot sees an enemy in the scan area
* **onHitWall(HitWallEvent event)** this method helps us perform some actions when our robot hits the wall.
* **onWin(WinEvent event)** This method helps us perform some actions when our robot is the last on the battlefield.
* Other methods can refer [here](https://robocode.sourceforge.io/docs/robocode/index.html?robocode/Robot.html).

## INTRODUCTION ABOUT TEAM’S ROBOT

### Concept about robot

In the 1v1 mode, a robot can chase an opponent but retains a fixed distance to avoid danger (analysis of danger - rick), thereby giving a suitable distance, capable of computing the enemy's projectile trajectory, speed, movement, and self-adjust the projectile to suit your energy.

In battle royal mode, the idea is like PvP but will be upgraded in terms of scanning, not targeting a fixed target to avoid shooting 1 target at a long distance while. There is another target near me, when the enemies are almost destroyed, the state switches to focus (targeting 1 target). At the same time always choose good positions such as corners on the wall, always avoid other opponents and limit movement to the center of the ring.

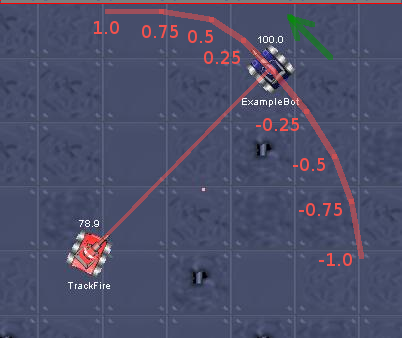
## IMPLEMENTATION OF THE IDEA

### GuessFactor Targeting[[1]](https://robowiki.net/wiki/GuessFactor_Targeting_Tutorial) Algorithm

GuessFactor (GF) is a way of measuring firing angles which, unlike a raw [bearing offset](https://robowiki.net/wiki/Bearing_Offset), takes into account the enemy's relative direction at fire time and the [maximum escape angle](https://robowiki.net/wiki/Maximum_Escape_Angle) for the specific firing situation.

A GuessFactor of 1.0 represents the bearing offset that the enemy would reach if it maximized its escape angle while moving in its current direction relative to the firing bot (i.e., clockwise or counter-clockwise), -1.0 represents the bearing offset that the enemy would reach if it maximized his escape angle after reversing directions, and 0.0 represents the bearing offset that points directly at the enemy (as in [head-on targeting](https://robowiki.net/wiki/Head-on_targeting)).

GuessFactors are essentially normalized angles. They are relevant for [Wave Surfing](https://robowiki.net/wiki/Wave_Surfing)[[2]](https://robowiki.net/wiki/Wave_Surfing/GoTo_Surfing), in which all GuessFactors of each [Wave](https://robowiki.net/wiki/Wave) are assigned a predicted danger, allowing the robot to move in the direction ([True Surfing](https://robowiki.net/wiki/Wave_Surfing/True_Surfing))[[3]](https://robowiki.net/wiki/Wave_Surfing/True_Surfing) or to the point ([GoTo Surfing](https://robowiki.net/wiki/Wave_Surfing/GoTo_Surfing))[[4]](https://robowiki.net/wiki/Wave_Surfing/GoTo_Surfing) of least danger. Some [guns](https://robowiki.net/wiki/Targeting) also use GuessFactors, in that they pick a GF and shoot at it. GuessFactors are also viable in [Melee](https://robowiki.net/wiki/Melee), but since in melee, bots are not typically moving perpendicular to you, GuessFactors are generally less useful.



GuessFactor Targeting Terminology:

* [Movement Profile[5]](https://robowiki.net/wiki/Movement_Profile): The combination of directions we could fire and the frequency that each direction is correct is what people refer to as the bot's "profile". You can view the profile of a given bot from the saved data of other robots.
* [Rolling Averages[6]](https://robowiki.net/wiki/Rolling_Averages): Also called moving averages and some other things. A way of favoring newer data over older data. A [guess-factor](https://robowiki.net/wiki/GuessFactor_Targeting_(traditional)) gun can be awful against a robot that changes its movement unless it uses rolling averages, however, for simplicity, We probably won't implement them in this tutorial.
* [Segmentation[7]](https://robowiki.net/wiki/Segmentation): Sometimes we can get more relevant data by splitting it up on some parameters, rather than just looking at the general profile.
* [Virtual Bullets](https://robowiki.net/wiki/Virtual_Bullets)[[8]](https://robowiki.net/wiki/Virtual_Bullet): An object that tracks where a bullet would be if it were fired along a certain trajectory and finds out if it would hit at that trajectory or not. These were used in all of the earliest guess factor guns to analyze the bot's [MovementProfile](https://robowiki.net/w/index.php?title=MovementProfile&action=edit&redlink=1).
* [Wave](https://robowiki.net/wiki/Wave)[[9]](https://robowiki.net/wiki/Waves): Most current robots that use [Guess Factor Targeting](https://robowiki.net/wiki/GuessFactor_Targeting_(traditional)) use [Waves](https://robowiki.net/wiki/Wave) to find the bearing that they should have shot at, as an alternative to using several [Virtual Bullets](https://robowiki.net/wiki/Virtual_Bullets). These are a little more efficient in general, and can also be modified slightly to give the same results as [Virtual Bullets](https://robowiki.net/wiki/Virtual_Bullets) (tracking ALL angles that would have hit, rather than just the median).

### Melee Strategy Algorithm[[10]](https://robowiki.net/wiki/Melee_Strategy#Evaluating_your_Melee_bot)

#### Introduction about melee

Melee competitions are your bot versus at least 2 other bots (and usually 9 others for a total of 10 bots.) These bots need to manage radar, energy consumption, and field location much more carefully than their [1-vs-1](https://robowiki.net/wiki/1-vs-1) counterparts. A good melee bot is usually a good 1-vs-1 bot, but the converse is not necessarily true. Field sizes for these battles are almost always larger than the default and usually are around 1000x1000 - though larger fields can make interesting battles when other bots are so far apart that they can't see each other. These bots are a good challenge after 1-vs-1 due to the new variables that are added to the battle.

#### The timeline of a melee battle

|  |  |  |
| --- | --- | --- |
| Bots on the field | Target | Description |
| 10 | The bots in bad starting positions die right away. Hope you're not one of them. | If there is someone close to you, it had better die before you. Most good bots will turn and move away from other bots or the center in general right away, so if you have some kind of targeting that shoots at the path of least resistance, this is where you'll notice it, but a semi-linear targeting will probably work for that first few seconds in the match. |
| 6 – 7 | Fighting over corners and trying not to get pushed into crossfires. | You'll be happier if you're targeting one bot and no bot is targeting you. You should be able to hit most of the bots you'd reasonably want to target at this point, and you won't be successful with a hit rate of less than maybe 25-30% at this point. |
| 3 - 4 | Everyone stays in their corners and snipes at each other. Try to get the other guys to target each other instead of you. | This is where you'll be the happiest with solid targeting. Here's why: when you're down to 4 bots, there's a really good chance that you're all in corners. If the next bot who dies is the one across from you (the one you probably weren't targeting), you are probably next, because you're the most obvious target for both of the remaining enemies. |
| Duel | The [1 vs 1](https://robowiki.net/wiki/1-vs-1) fight at the end of a melee battle. | It should be obvious why targeting is important here. It's your gun against their movement. |

In reality, these stages aren't something you normally put into code when making a melee bot, they're observations about what a battle will look like between several good melee bots. The fact that these stages are defined in terms of position and movement says a lot about what is important in melee. The reason these stages are important to understand is really so that you can analyze your melee bot's performance if you put it in a battle with other bots of a similar level.

#### How to put together a good melee bot

The best way is moving continuously without being targeted by the enemy, but not moving linearly because it is easy to guess the direction. At the same time always choose the best angle you can shoot.

#### The theory of melee movement

You'll dodge every bullet if no one is firing at you, and so most melee movement is centered in one way or another around not getting attacked. This is what makes melee movement so dang interesting. If it is inevitable that you will be targeted, then you can start worrying about just not being hit.

Two movement goals combine into the balance between "being in the right place" (not being shot at) and "moving in the right direction" (not being hit). Since you can only move in one direction, sometimes you have to choose to either move toward the right place or move in the right direction. Since different bots prioritize these differently, some will be extremely predictable and easy to hit, but it doesn't matter until someone shoots at them. In fact, in theory, as long as no one is shooting at you, the only reason you have to keep moving is making sure they keep not shooting at you (until they're shooting at you again, which would be nice to be able to magically determine).

#### Common ways of implementing melee movement

There are lots of ways people accomplish the goals of not being targeted and not being hit. Some common examples:

* Anti-Gravity Movement[[11]](https://robowiki.net/wiki/Anti-Gravity_Movement)
* Corner Movement[[12]](https://robowiki.net/wiki/Corner_Movement)
* Pattern Movement[[13]](https://robowiki.net/wiki/Pattern_Movement)

#### Minimum Risk Movement[[14]](https://robowiki.net/wiki/Minimum_Risk_Movement)

On the surface, this looks like just an implementation of the [Anti-Gravity Movement](https://robowiki.net/wiki/AntiGravityMovement), and maybe the difference is just nitpicky semantics to some, but I think the way it is implemented is different enough that the only similarity is the fundamentals of melee movement. It also seems much more versatile than the [Anti-Gravity Movement](https://robowiki.net/wiki/AntiGravityMovement) and is also less likely to converge at some "happy" local minimum.

There are two basic parts to [Minimum Risk Movement](https://robowiki.net/wiki/MinimumRiskMovement):

* + A Risk Function - Determines how good or bad a point is for moving.
  + A Point-generating function - generates candidates to try in the risk function.

Then you just go to the candidate point with the lowest risk (or highest benefit, although MaximumBenefitMovement probably has more place in [Teams](https://robowiki.net/wiki/Teams)). Randomness can assist in either function just as effectively. Also, as an interesting tweak, you need to decide to look for a point to go to constantly rather than only when you reach the last point you were going to. Both strategies have merits - you might get into a rut if you keep changing your mind on points, but you can also more quickly react to changes on the battlefield.

#### Things to think about in your movement

|  |  |
| --- | --- |
| Things | Description |
| Being far away from other bots | This one of the easier things to test for, of course, once bots are so far away, it doesn't matter as much if they get further. That's why using the inverse square of the distance to each robot works so well (the threat approaches zero at some point). This will likely keep you out of the action. |
| Moving perpendicularly/one-on-one-like to other bots | Especially bots that are close to you. |
| Don't hit walls or other bots too much | It might be tempting. This is possibly a bigger deal in melee, just because you're going to be close to other bots and the walls a lot more than in one-on-one. |
| Stay out of the middle | People always say this, but in reality, it's ok to be in the middle of the battle is still big and all the other bots are on the outside. However, if you're still in the middle with 4 or 5 bots left, you won't likely last long. |
| Don't stay in the same place for too long | Just as in 1 vs 1, this can be a quick death. I don't care how great that spot is, someone will get close enough to it to put you in danger in not too much time |
| Battle dynamics change constantly | Be capable of adjusting and changing quickly, and don't try to plan too far into the future. |
| Guess who will fire at you | You want to be at the point where the fewest robots will target you. You only have to worry about being hard to hit the robots that are likely to target you (in other words, the direction you move in to get to a given point only matters relative to bots that are likely to target either that point or your current location) |

#### Little melee tweaking notes

**Melee radar**: What you do with your radar depends on what your targeting strategy is, and whether or not you want to fire blind (firing blind means you don't fire when you can see your enemy, but rather in your general run loop or something). If you just spin your radar constantly, you almost need to fire blind, otherwise, your gun won't have time to turn before you fire. However, I find (again, if your gun doesn't depend on a spinning radar, as some melee pattern-matches do) that firing blindly reduces my hit rate by a little. I suggest (particularly if you're using a guess-factor gun in melee) using a melee radar lock. A melee radar lock is spinning your radar until you're getting close to firing and then lock on your target as if they were the only bot on the field for a couple of ticks until you fire. Plus, if you have a radar lock in your code, you can also use it for 1 vs 1.

**Melee firepower**: A lot of hitting happens early in a melee match, but by the end, you can be pretty well drained of energy. Some people take the survivalist element of melee to mean they should always fire low-powered bullets. This is only partly true - you will likely start the melee battle close to other bots. So close, that you are very likely to rack up a lot of hits (either for or against you) in the first 5 seconds of a match. Don't be shy about firing bullets that are more than powerful enough to finish a bot off if they're close to you (because you still want to finish them off if they hit you or someone else before your bullet reaches them, and if you're close, that's likely to happen).

**Picking a target**: There are so many heuristics that can be dreamed up for this, but one of the easiest and most effective is to simply target the closest enemy. While this is the best general guideline, it might be beneficial to consider a few other things:

* Who has the lowest energy? There are a few reasons you might want to target them:
* To get the kill bonus.
* If they are at least sort of close to you, and they are in a "safer" part of the battlefield, you might be able to take their place if they die :-) If your movement allows you to get closer to robots with lower energy, you might be more likely to do this.
* So you can outsurvive them! You get something like 100 points for every bot you outlast in around. You'd hate to see a robot that had 5 energy when you had 60 ends up dying after you because no one shot at it when you could have.
* If you have a good risk function your bot will attack a low energy bot in case it has a good position on the field so that your bot will get there (important if there are around 4 to 6 bots left).
* Who keeps shooting at you? There are two ways to avoid getting shot at - get them to shoot at someone else, or take them out. Yes, killing someone can help you conserve energy!
* Try not to thrash between enemies too much, especially when you're ready to fire. If you're only using distance to pick a target, try only switching targets if the new target is 100 pixels closer, or 10% closer, or something. If you have some other kind of "score" technique for picking a target, something similar could be used. Thrashing can cost you, especially early in a battle. Another good technique is to not switch targets if your gun is almost cool (unless your target is dead, of course).

#### Enemy management

If you're trying to convert a 1-on-1 bot to a melee bot, you'll find out quickly that all the little assumptions you used to make when you coded your 1-on-1 bot are wrong. It can be a big challenge to effectively store volatile information about your enemy (like current position, heading, velocity, etc) as well as persistent information about your enemy (like targeting information). One of the big challenges is how to specify that a given enemy is dead. I stumbled on a great way to do this when I was working on [Coriantumr](https://robowiki.net/wiki/Coriantumr), which was to have a volatile object (class EnemyInfo) in a Map that has a position, heading, velocity, and all the obvious stuff in it, as well as a list of waves (note - all these things should go away between rounds). This map doesn't have to be static (although it probably is for code size reasons) because it's recreated every round. In onScannedRobot, I check to see if the robot I'm scanning is on the map, and if it isn't, I put it in there. Then I get the EnemyInfo object and operate on it all over onScannedRobot. onRobotDeath removes that robot's object from the map if they exist (so the robot no longer exists when I don't care about it anymore). Meanwhile, I have a different map that maps the name to the guess factor stats which is static and isn't reinitialized between rounds.

#### Targeting algorithms in melee

* Head On Targeting[[15]](https://robowiki.net/wiki/Head-On_Targeting) [in melee.](https://robowiki.net/wiki/Head-On_Targeting)
* [LinearTargeting](https://robowiki.net/wiki/LinearTargeting)[[16]](https://robowiki.net/wiki/Linear_Targeting) / [CircularTargeting](https://robowiki.net/wiki/CircularTargeting)[[17]](https://robowiki.net/wiki/Circular_Targeting)and variants.
* Pattern Matching[[18]](https://robowiki.net/wiki/Pattern_Matching) in melee.
* Guess Fator Targeting[[19]](https://robowiki.net/wiki/GuessFactor_Targeting_Tutorial) in melee.

#### Evaluating your Melee bot

Before I get too far into this, I want to emphasize that the number one way of figuring out where you can improve is by critically watching the battles. The first thing to do is to create a testbed. Find a bunch of bots that are at least in the same neighborhood as you as far as ability, some better and some worse. I also make a text file where I log battles and what tweaks I made between them. Of course, don't give too much credit to the scores, since they often vary wildly. If you start to beat your testbed consistently, upgrade it.

### Robot class

*class* Robot *extends* Point2D.Double {  
 *public long* scanTime;  
 *public boolean* alive = *true*;  
 *public double* energy;  
 *public* String name;  
 *public double* gunHeadingRadians;  
 *public double* absoluteBearingRadians;  
 *public double* velocity;  
 *public double* heading;  
 *public double* lastHeading;  
 *public double* shootAbleScore;  
 *public double* dist;  
}

* This Robot class extends from Point2D because we want to store the location of the robot and use some methods in the Point2D class (getDistance(), getX(), getY()).
* Attributes of the class:
  + scanTime: represents the scan time of the robot.
  + alive: check the status of the robot on the battlefield.
  + energy: energy of the robot.
  + name: name of the robot.
  + gunHeadingRadians: the heading angle of the gun.
  + absoluteBearingRadians: the absolute bearing angle.
  + velocity: velocity of robot.
  + heading: heading angle of the robot.
  + lastHeading: last heading angle of the robot.
  + shootAbleScore: represents for the shoot score of the robot based on the remaining energy. The smaller the robot’s score, the closer it is to deadth
  + dist: distance of robot.

### Utility class

*public static class* Utility {  
 *static double* clamp(*double* value, *double* min, *double* max) {  
 *return* Math.*max*(min, Math.*min*(max, value));  
 }  
   
 *static* doublerandomBetween(*double* min, *double* max) {  
 *return* min + Math.*random*() \* (max - min);  
 }  
  *static* Point2D project(Point2D sourceLocation, *double* angle, *double* length) {  
 *return new* Point2D.Double(sourceLocation.getX() + Math.*sin*(angle) \* length,  
 sourceLocation.getY() + Math.*cos*(angle) \* length);  
 }  
  *static double* absoluteBearing(Point2D source, Point2D target) {  
 *return* Math.*atan2*(target.getX() - source.getX(), target.getY() - source.getY());  
 }

*static int* sign(*double* v) {  
 *return* v < 0 ? -1 : 1;  
 }  
}

* The methods of the Utility class:
* **clamp(value, min, max):** Returns the value between the three were passed.
* **randomBetween(min, max):** Returns the random value which is in range (min, max).
* **project(sourceLocation, angle, length):** Returns the position of the enemy robot.
* **absoluteBearing(p1, p2):** Calculates the angle of the point in the polar coordinate system which was created from two other points in the rectangular coordinate system.

x = target.getX() – source.getX()

y = target.getY() – source.getY()

* + Forms of the coordinate system were used:
  + Rectangular coordinates system (x, y)[[20]](https://en.wikipedia.org/wiki/Cartesian_coordinate_system)
  + Polar coordinate system (r, theta)[[21]](https://en.wikipedia.org/wiki/Polar_coordinate_system)
* **sign(v):** reverse direction.

### Movement\_1VS1 class

*private static final double BATTLE\_FIELD\_WIDTH = 800;  
private static final double BATTLE\_FIELD\_HEIGHT = 600;  
private static final double MAX\_TRY\_TIME = 125;  
private static final double REVERSE\_TUNER = 0.421075;  
private static final double DEFAULT\_EVASION = 1.2;  
private static final double WALL\_BOUNCE\_TUNER = 0.699484;  
private final AdvancedRobot robot;  
private final Rectangle2D fireField = new Rectangle2D.Double(WALL\_MARGIN, WALL\_MARGIN,  
 BATTLE\_FIELD\_WIDTH - WALL\_MARGIN \* 2, BATTLE\_FIELD\_HEIGHT - WALL\_MARGIN \* 2);  
private double direction = 0.4;*

* Movement\_1VS1 class is created to advanced portability to fit solo mode.
* Attributes of the class:
  + BATTLE\_FIELD\_WIDTH represents the width of the field robot can move.
  + BATTLE\_FIELD\_HEIGHT: represents the height of the field robot can move.
  + MAX\_TRY\_TIME: the amount of the time robot tries to move
  + REVERSE\_TUNER: a constant number to tune the direction of the robot.
  + DEFAULT\_EVASION: a constant number represents the evasion ratio of the robot.
  + WALL\_BOUNCE\_TUNER: a constant number represents the wall-avoiding ratio of the robot.
  + robot: the robot wants to create the wave.
  + fireField: represents a fire area.
* onScannedRobot() method:
  + Create a robot that symbolizes the enemy robot and calculate values such as absolute bearing angle, distance. Then calculate the coordinates 1 point representing the current location of our robot, a representative for the current enemy robot position. Create 1 more point to store the destination of the robot (used for the rear), 1 tryTime value = 0.

Robot enemy = new Robot();  
enemy.absoluteBearingRadians = robot.getHeadingRadians() + e.getBearingRadians();  
enemy.dist = e.getDistance();  
Point2D robotLocation = new Point2D.Double(robot.getX(), robot.getY());  
Point2D enemyLocation = Utility.project(robotLocation, enemy.absoluteBearingRadians, enemy.dist);

* + Migration algorithm
    - Step 1: repeat up to MAX\_TRY\_TIME = 125 to find the new destination of the robot both of us (the condition is that that point must be in the firing area of the robot (fireField))

New destination coordinates

* + - * + X= e.x + sin(e.absBearing + PI + direction ) \* e.distance \* (1.2 - tryTime /100) )
        + Y= e.y + cos(e.absBearing + PI + direction ) \* e.distance \* (1.2 - tryTime /100) )

while (!fireField.contains(robotDestination = Utility.project(enemyLocation, enemy.absoluteBearingRadians + Math.PI + direction,  
 enemy.dist \* (DEFAULT\_EVASION - tryTime / 100.0))) && tryTime < MAX\_TRY\_TIME)  
 tryTime++;

* + - Step 2: Change the direction of movement in 2 cases. When taking a random number in the range [0,1) < bullet velocity /(0.421075 \* enemy distance) or the number of attempts (tryTime) > enemy distance / (bullet velocity \* 0.699484)

if ((Math.random() < (Rules.getBulletSpeed(enemyFirePower) / REVERSE\_TUNER) / enemy.dist ||  
 tryTime > (enemy.dist / Rules.getBulletSpeed(enemyFirePower) / WALL\_BOUNCE\_TUNER)))  
 direction = -direction;

* + - Step 3: calculate the angle to move. The moving angle is the angle of the point created by 2 points (the current robot position, the new counting point position) in the polar coordinate system – the current angle of movement of the robot

double angle = Utility.absoluteBearing(robotLocation, robotDestination) - robot.getHeadingRadians();

* + - Step 4: Set up movement and radar rotation for the robot based on the angle just found

robot.setAhead(Math.cos(angle) \* 100);  
robot.setTurnRightRadians(Math.tan(angle));

### Wave class

static class Wave extends Condition {  
 static Point2D targetLocation;  
 double bulletPower;  
 Point2D gunLocation;  
 double bearing;  
 double lateralDirection;  
 private static final double MAX\_DISTANCE = 900;  
 private static final int DISTANCE\_INDEXES = 5;  
 private static final int VELOCITY\_INDEXES = 5;  
 private static final int BINS = 25;  
 private static final int MIDDLE\_BIN = (BINS - 1) / 2;  
 private static final double MAX\_ESCAPE\_ANGLE = 0.7;  
 private static final double BIN\_WIDTH = MAX\_ESCAPE\_ANGLE / (double) MIDDLE\_BIN;  
 private static final int[][][][] statBuffers = new int[DISTANCE\_INDEXES][VELOCITY\_INDEXES][VELOCITY\_INDEXES][BINS];  
 private int[] buffer;  
 private final AdvancedRobot robot;  
 private double distanceTraveled;  
}

* Wave class is created to use for collecting data to control movement, target, and shoot enemy, Wave class is inherited from Condition class to check condition when the robot should shoot the enemy.
* A wave contains source position (position of the shooting robot), velocity (based on the power of the fired bullet), travel time of the wave.
* A wave "breaks" or "hits" when the distance the wave has traveled (wave\_velocity \* (time\_now - time\_fired)) is greater than the distance from the wave source to the enemy. You can imagine the wave as a circle that radiates out from the point from which it was fired (which is where the name came from). Once the wave breaks, the firing angle - the offset relative to head-on targeting - that would have hit the enemy can easily be deduced.
* GuessFactor Targeting[[1]](#_[1]__https://robowiki.net/wiki/Gues) is used in solo mode.
* Attributes in this class:
  + targetLocation: represents the location robot is aiming at.
  + bulletPower: represents the power of the bullet.
  + gunLocation: represents the location of the gun when robot fires.
  + bearing: the bearing angle.
  + lateralDirection: direction of the robot.
  + MAX\_DISTANCE: maximum distance between our robot and enemy robot.
  + DISTANCE\_INDEXES: indexes of the segments that stored distant values of the enemy robot were collected.
  + VELOCITY\_INDEXES: indexes of the segments that stored velocity values of the enemy robot were collected.
  + BINS: the number of bins uses to stored data.

### The methods in Wave class

* + - * **test():** use for testing the conditions to calculate whether the bullet can hit the target or not.

public boolean test() {  
 advance();  
 if (hasArrived()) {  
 buffer[currentBin()]++;  
 robot.removeCustomEvent(this);  
 }  
 return false;  
}

* Step 1: Calculate the travel distance of the bullet (distance += velocity).
* Step 2: Check if the distance value was found in step 1 is greater than the distance from our robot position to the point we are aiming at. If so, increase the index of the bin by 1 and delete the previously registered shot event.
* Step 3: Return the false value after check condition.
  + - * **mostVisitedBearingOffset():** calculate offset of the bearing angle = direction \* 0.05 - ( index of the highest data bin -14).

double mostVisitedBearingOffset() {  
 return (lateralDirection \* BIN\_WIDTH) \* (mostVisitedBin() - MIDDLE\_BIN);  
}

* + - * **setSegmentation():** get data for all bins from data aggregated and analyzed.

void setSegmentations(double distance, double velocity, double lastVelocity) {  
 int distanceIndex = (int) (distance / (MAX\_DISTANCE / DISTANCE\_INDEXES));  
 int velocityIndex = (int) Math.abs(velocity / 2);  
 int lastVelocityIndex = (int) Math.abs(lastVelocity / 2);  
 buffer = statBuffers[distanceIndex][velocityIndex][lastVelocityIndex];  
}

* + - * **advance():** calculate the travel distance of the bullet.

private void advance() {  
 distanceTraveled += Rules.getBulletSpeed(bulletPower);  
}

* + - * **hasArrived():** check to see if the fired bullet reaches the target.

private boolean hasArrived() {  
 return distanceTraveled > gunLocation.distance(targetLocation) - WALL\_MARGIN;  
}

* + - * **currentBin():** return the current data bin.

private int currentBin() {  
 int bin = (int) Math.round(((Utils.normalRelativeAngle  
 (Cham.Utility.absoluteBearing(gunLocation, targetLocation) - bearing)) /  
 (lateralDirection \* BIN\_WIDTH)) + MIDDLE\_BIN);  
 return (int) Utility.clamp(bin, 0, BINS - 1);  
}

* + - * **mostVisitedBin():** return the index of highest data bin.

private int mostVisitedBin() {  
 int mostVisited = MIDDLE\_BIN;  
 for (int i = 0; i < BINS; i++)  
 if (buffer[i] > buffer[mostVisited])  
 mostVisited = i;  
 return mostVisited;  
}

* + - * **onScannedRobot()**
  + Create a robot to represent the enemy robot and perform calculations such as absolute bearing angle and distance. Then calculate the coordinates of one point that represents the current position of our robot, one point represents the current position of the enemy robot. Create one more point used to store the destination of our robot, one try time (tryTime) = 0

Robot enemy = new Robot();  
enemy.absoluteBearingRadians = robot.getHeadingRadians() + e.getBearingRadians();  
enemy.dist = e.getDistance();  
Point2D robotLocation = new Point2D.Double(robot.getX(), robot.getY());  
Point2D enemyLocation = Utility.project(robotLocation, enemy.absoluteBearingRadians, enemy.dist);

* + Movement algorithm
    - * + Step 1: repeat up to MAX\_TRY\_TIME = 125 to find out a new possible destination for our robot (the condition is the point must be in the fired field of the robot (fireField)).

New coordinates of the destination

while (!fireField.contains(robotDestination = Utility.project(enemyLocation, enemy.absoluteBearingRadians + Math.PI + direction,  
 enemy.dist \* (DEFAULT\_EVASION - tryTime / 100.0))) && tryTime < MAX\_TRY\_TIME)  
 tryTime++;

* + - * + Step 2: Change direction of the movement in 2 case:

a random value is in range [0, 1) < velocity of the bullet /(0.421075 \* enemy distance)

or

try time > enemy distance / (velocity of the bullet \* 0.699484).

if ((Math.random() < (Rules.getBulletSpeed(enemyFirePower) / REVERSE\_TUNER) / enemy.dist ||  
 tryTime > (enemy.dist / Rules.getBulletSpeed(enemyFirePower) / WALL\_BOUNCE\_TUNER)))  
 direction = -direction;

* + - * + Step 3: Calculate travel angle is the angle of the point created by 2 points (current robot position, new destination) in polar coordinates system– the robot’s current travel angle.

double angle = Utility.absoluteBearing(robotLocation, robotDestination) - robot.getHeadingRadians();

* + - * + Bước 4: Set movement and radar rotation for robot based on an angle has just been found.

robot.setAhead(Math.cos(angle) \* 100);  
robot.setTurnRightRadians(Math.tan(angle));

### Attributes required of the robot to execute the methods

*static final int* AMOUNT\_PREDICTED\_POINTS = 150;  
static final int WALL\_MARGIN = 18;  
HashMap<String, Robot> enemyList = *new* HashMap<>();  
Robot myRobot = *new* Robot();  
Robot targetRobot;  
*List*<Point2D.Double> possibleLocations = *new* ArrayList<>();  
Point2D.Double targetPoint = *new* Point2D.Double(60, 60);  
Rectangle2D.Double battleField = *new* Rectangle2D.Double();  
*int* idleTime = 30;  
private static double lateralDirection;  
private static double preEnemyVelocity;  
private static Movement\_1VS1 movement1VS1;

* AMOUNT\_PREDICTED\_POINTS: Represents the amount of the possible location which the robot can reach. In this case, we need 150 points.
* WALL\_MARGIN: Represents for the least distance from the robot to the wall for avoiding wall
* enemyList: Use HashMap[[22]](https://www.w3schools.com/java/java_hashmap.asp) to a stored list of the opponent robot (each enemy is unique ) to calculate reasonable movement, target location,...
* myRobot: Represents our robot to store some necessary data.
* targetRobot: Represents an opponent robot that we are targeting.
* possibleLocations: Uses to stored a list of points to which our robot can move to.
* targetPoint: Represents the location of the point which has the lowest risk.
* battleField: Uses to reduce call methods get information of the battlefield (getBattleFieldWidth và getBattleFieldHefight). At the same time, it determines the size of the battlefield.
* idleTime: Represents the amount of time that the robot needs to reach the lowest risk location. If the robot needs a too long time, the robot will recalculate the destination.
* lateralDireciton: lateral direction of the robot.
* preEnemyVelocity: previous lateral velocity.
* movement1VS1: object performs how the robot moves in solo mode.

### shooting method

Checks the status of the robot we are aiming for. If it is alive, takes the following actions: *if* (targetRobot != *null* && targetRobot.alive)

* Calculates the energy of the bullets we will fire with the target achieves the greatest amount of damage with the least energy.
  + Step 1: Calculating energy bases on distance.
  + Step 2: Calculating energy bases on the enemy’s energy and our robot’s energy.
  + Step 3: Gets the value between three (power has just been calculated in step 2, the lowest power of the bullet, the greatest power of the bullet).

*double* dist = myRobot.distance(targetRobot);  
*double* power = (dist > 850 ? 0.1 :  
 (dist > 700 ? 0.5 : (dist > 250 ? 2.0 : 3.0)));  
power = Math.*min*(myRobot.energy / 4d,  
 Math.*min*(targetRobot.energy / 3d, power));  
power = Utility.*clamp*(power, 0.1, 3.0);

* Targeting: Uses the algorithm of Circular Targeting method[[17]](#_[17]__https://robowiki.net/wiki/Cir)
  + Step 1: Get the current location of our targeting robot to set the current targeting location, time needs to hit the enemy is 0, deltaHead (angle deviation between new direction and old direction) of opponent robot = present heading– last heading.

long deltaHitTime;  
Point2D.Double shootAt = new Point2D.Double();  
double head, deltaHead, bulletSpeed;  
double predictX, predictY;  
predictX = targetRobot.getX();  
predictY = targetRobot.getY();  
head = targetRobot.heading;  
deltaHead = head - targetRobot.lastHeading;  
shootAt.setLocation(predictX, predictY);  
deltaHitTime = 0;

* + Step 2: Loop until the distance between our robot and aiming position – WALL\_MARGIN) / velocity of the bullet with specific power <= amount of time needs to hit ()

*do* {

*…condition*  
} *while* ((*int*) Math.*round*((shootAt.distance(myRobot) – WALL\_MARGIN) / Rules.*getBulletSpeed*(power)) > deltaHitTime);

* + - Predict the next position of the opponent robot bases on velocity and direction (lateral velocity method)

* + - * Inside:
        + head : angle of direction of enemy
        + velocity: velocity of enemy

predictX += Math.*sin*(head) \* targetRobot.velocity;  
predictY += Math.*cos*(head) \* targetRobot.velocity;

* + - Update direction of the enemy by adding an angle ( deltaHeading).
    - Increase delta hit time by 1.
    - Create a fired field which has size is calculated by the following formulary at the point has location is (x=18, y=18).

head += deltaHead;  
deltaHitTime++;  
Rectangle2D.Double fireField = *new* Rectangle2D.Double(WALL\_MARGIN, WALL\_MARGIN,  
 battleField.width - 36, battleField.height - 36);

* + - Check the predicted coordinates if it in the advantage fire area. If not we calculate the velocity of the bullet bases on the distance between the last targeting location and hit time . After that, we need to set fit power for the bullet and break the loop. Otherwise, change the predicted coordinate into aiming location

*if* (!fireField.contains(predictX, predictY)) {  
 bulletSpeed = shootAt.distance(myRobot) / deltaHitTime;  
 power = Utility.*clamp*((20 - bulletSpeed) / 3.0, 0.1, 3.0);  
 *break*;  
}

* + Step 3: Set new value for the targeting location based on predict coordinates and size of the battle field to ensure it is always correct

shootAt.setLocation(Utility.*clamp*(predictX, 34, getBattleFieldWidth() - 34), Utility.*clamp*(predictY, 34, getBattleFieldHeight() - 34));

* + Step 4: Set condition to fire (with purpose is saving as much energy as possible)

*if* ((getGunHeat() == 0.0) && (getGunTurnRemaining() ==0.0) && (power > 0.0) && (myRobot.energy > 0.1))  
 setFire(power);

* + Step 5: Turn radar:
    - Change rectangle coordinate system (shootAt.x – getX(), shootAt.y – getY()) into polar coordinate system to get angle.
    - Then get value from formulary

to ensure radar turning angle has value no more than 90 degrees. Get the above result subtract present direction of the gun then normalize to get an absolute angle in range .

setTurnGunRightRadians(Utils.*normalRelativeAngle*((  
(Math.PI / 2) - Math.*atan2*(shootAt.y - myRobot.getY(),shootAt.x - myRobot.getX())) - getGunHeadingRadians()));

### updateListLocation method

* Update list of possible positions of the robot with a specific amount
  + Step 1: Clear all old positions in the list.
  + Step 2: Set the range of x coordinate for the robot’s movement by using 125 \* 1.5. The value will be rounded down to the nearest whole number.
  + Step 3: Use the loop with the maximum time as the amount of the predicted locations.
    - Create a random value (randXMod) of the robot in range (-xRange, xRange)
    - Calculate the range of the y coordinate depends on a range of the x coordinate, yRange =
    - Create a random value (randYMod) of the robot in range (-yRange, -yRange)
    - Set the coordinate which robot can move with the condition is avoiding (wall, enemy)
    - x equals value between three (current x coordinate + randXMod, 75, battleFieldHeight - 75)
    - y equals value between three (current y coordinate + randYMod, 75, battleFieldHeight – 75)
    - Add the location has just found above to the list for movement later.

*public void* updateListLocations(*int* n) {  
 possibleLocations.clear();  
 *final int* radius = (*int*) (125 \* 1.5);  
 *for* (*int* i = 0; i < n; i++) {  
 *double* randXMod = Utility.*randomBetween*(-radius, radius); *double* yRange = Math.*sqrt*(radius \* radius - randXMod \* randXMod);  
 *double* randYMod = Utility.*randomBetween*(-yRange, yRange);  
 *double* y = Utility.*clamp*(myRb.y + randYMod, 75, battleField.height - 75);  
 *double* x = Utility.*clamp*(myRb.x + randXMod, 75, battleField.width - 75);  
 possibleLocations.add(*new* Point2D.Double(x, y));  
 }  
}

### evaluatePoint method

Uses to evaluate the hazard of a specific location aim to decide whether the robot should move there or not.

First of all, let's learn Anti – Gravity Movement technique [[2]](#_[11]__https://robowiki.net/wiki/Ant): This is a medium movement method that is often used in melee. Like most melee movements, it targets and holds our robot as far as possible from other robots. This technique relates to using the law of gravity, but vice versa. For example, when our robot moves to the wall closely, it will be pushed our robot out a specific distance.

Formulary

* Step 1: Initialize a risk evaluated value (rickValue) by a number in range (0,75, 2) \* gravity

*double* rickValue = Utility.*randomBetween*(0.75, 2) / p.distanceSq(myRobot);

* Step 2: In the case of the battlefield there has only one enemy, our robot can move in a center area of the battlefield. Otherwise, the robot just moves to a position where the robot is least likely to be shot by an enemy robot (move to one of four corners of the battlefield) by getting risk value plus another value which equals to (The distance in the anti-gravity formulary is the distance between the evaluated point and the center point of the battlefield)

rickValue += (6 \* (getOthers() - 1)) /   
 p.distanceSq(battleField.width /2, battleField.height/2);

* Step 3: It is the same with step 2 but uses to avoid collision with the other robots and four corners of the battlefield: rickValue will plus a value which equals to conerFactor (with cornerFactor is one value of association {0.25, 0.5, 1} based on the number of the enemies remaining in the battlefield \* gravity force (The distance in the gravity formulary is the distance between the evaluated point and four corners of the battlefield)
  + Case 1: The bottom – left corner has coordinate is (0,0)
  + Case 2: The bottom – right corner has coordinate is (width of the battle field, 0)
  + Case 3: The top – left corner has coordinate is (0, height of the battle field)
  + Case 4: The top – right corner has coordinate is (width of the battle field, height of the battle field)

*double* cornerFactor = getOthers() <= 5 ? getOthers() == 1 ? 0.25 : 0.5 : 1;  
rickValue += cornerFactor / p.distanceSq(0, 0);  
rickValue += cornerFactor / p.distanceSq(battleField.width, 0);  
rickValue += cornerFactor / p.distanceSq(0, battleField.height);  
rickValue += cornerFactor / p.distanceSq(battleField.width, battleField.height);

* Step 4: Check the status of the targeting robot is alive or dead. If the status is dead, moves over to step 5. Otherwise, performs the following actions then move over to step 7.

*if* (targetRobot.alive)

* + Get the angle of the targeting robot with evaluated point (botAngle)
  + Traversal list of enemy robots and compute the risk of the evaluated point with each enemy robot
  + New risk = old risk + danger level of the enemy robot \* gravity force \* danger level when moving perpendicular to the enemy robot \* danger level when moving closely with the enemy robot
  + Inside:
    - The danger level of the enemy robot = enemy.energy / me.energy
    - Danger level when moving perpendicular to the enemy robot = 1
    - Danger level when moving closely with enemy robot = 1 +
    - Danger level when moving closely = enemy.energy – me.energy

*double* robotAngle = Utils.*normalRelativeAngle*(  
 Utility.*calcAngle*(p, targetRobot) - Utility.*calcAngle*(myRobot, p));  
*Iterator*<Robot> enIterator = enemyList.values().iterator();  
*while* (enIterator.hasNext()) {  
 Robot en = enIterator.next();  
 rickValue += (en.energy / myRobot.energy) \* (1 / p.distanceSq(en)) \*  
 (1.0 + ((1 - (Math.*abs*(Math.*sin*(robotAngle)))) + Math.*abs*(Math.*cos*(robotAngle))) / 2) \*  
 (1 + Math.*abs*(Math.*cos*(Utility.*calcAngle*(myRobot, p) - Utility.*calcAngle*(en, p))));  
}

* Step 5: If there have no enemy robot on the list, move over to step 6. Otherwise, performs following actions then move over to step 7:
  + Get each enemy robot in the list to compute risk.
  + New risk = old risk + danger level of the enemy robot \* gravity force \* danger level when move closely enemy robot

*else if* (enemyList.values().size() >= 1) {  
 *Iterator*<Robot> enIterator = enemyList.values().iterator();  
 *while* (enIterator.hasNext()) {  
 Robot en = enIterator.next();  
 rickValue += (en.energy / myRobot.energy) \* (1 / p.distanceSq(en)) \*  
 (1 + Math.*abs*(Math.*cos*(Utility.*calcAngle*(myRobot, p) - Utility.*calcAngle*(en, p))));  
 }  
}

* Step 6: New risk = old risk + danger level of the point which our robot is aiming to.
  + The danger level of the point which our robot is aiming to = 1 +

rickValue += (1 + Math.*abs*(Utility.*calcAngle*(myRobot, targetPoint) - getHeadingRadians()));

* Step 7: Return the computed risk value of the point *return* rickValue;

### movement method

* Step 1: Check if the distance between the target point and current robot location is less than 15 or targeting time is too long (idleTime > 25 tick), perform the following actions. Otherwise, move over to step 2:
  + Initialize value for idleTime = 0 aim to new target point
  + Update list of possible points around the robot
  + Traversal list of points to find out the least risk point and change the new point which has just been found into the new target point of robot.

idleTime = 0;  
 updateListLocations(AMOUNT\_PREDICTED\_POINTS);  
 Point2D.Double lowRiskP = *null*;  
 *double* lowestRisk = Double.MAX\_VALUE;  
 *for* (Point2D.Double p : possibleLocations) {  
 *double* currentRisk = evaluatePoint(p);  
 *if* (currentRisk <= lowestRisk || lowRiskP == *null*) {  
 lowestRisk = currentRisk;  
 lowRiskP = p;  
 }  
 }  
 targetPoint = lowRiskP;

* Step 2: Keep moving to the target point and perform the following:
  + Increase idleTime by 1 unit
  + Compute the angle between our robot and a target point (alpha)
  + Set the direction as 1 (go ahead)
  + Check if cos(alpha) < 0, reverse direction because moving back and turning robot will be faster than moving ahead in this case. (Choose the best direction to move)
  + Set the fit velocity for the robot to the number of turns is the least

( )

* + Moving ahead or back a distance equals the distance between our robot and target point
  + Normalize alpha angle (for a range of alpha is in )
  + Turn radar right an angle equals to angle has just normalized

idleTime++;  
*double* angle = Utility.*calcAngle*(myRobot, targetPoint) - getHeadingRadians();  
*double* direction = 1;  
*if* (Math.*cos*(angle) < 0) {  
 angle += Math.PI;  
 direction \*= -1;  
}  
setMaxVelocity(10 - (4 \* Math.*abs*(getTurnRemainingRadians())));  
setAhead(myRobot.distance(targetPoint) \* direction);  
angle = Utils.*normalRelativeAngle*(angle);  
setTurnRightRadians(angle);

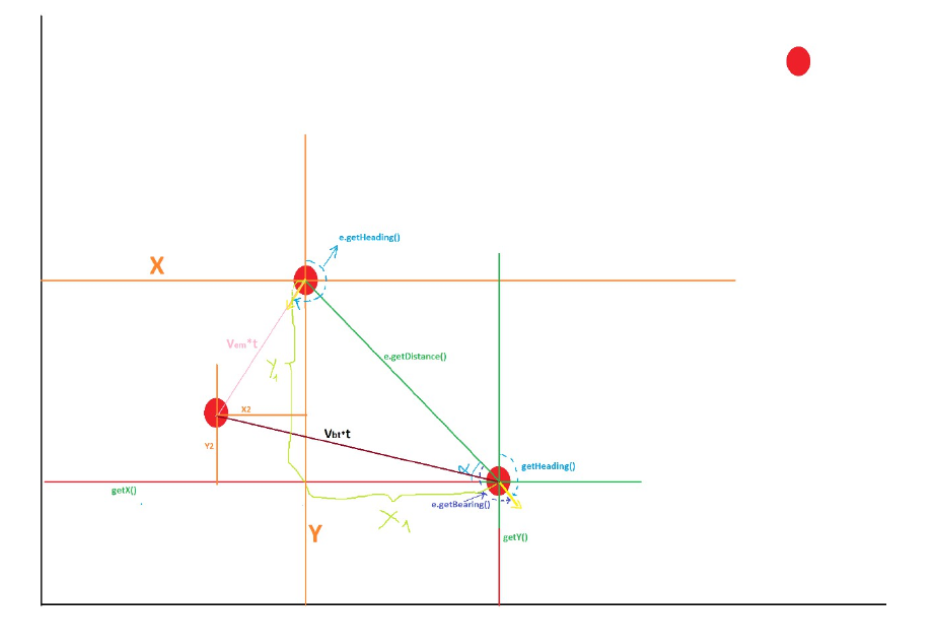
### onRobotDeath method

* Check if there have any robot in the list that has to name that is the same as the name of the dead robot, set status of that robot is dead.
* Check if our targeting robot has a name that is the same as the name of the dead robot, set status of that robot is dead

*public void* onRobotDeath(RobotDeathEvent event) {  
 *if* (enemyList.containsKey(event.getName()))  
 enemyList.get(event.getName()).alive = *false*;  
 *if* (event.getName().equals(targetRobot.name))  
 targetRobot.alive = *false*;  
}

### onScannedRobot method

* ***If the robot is in a chaos mode***
  + - * Firstly, we get information about the robot in the enemy robot list which has the name is the same as the name of the scanned robot. If there no have a robot in the list, add the scanned robot to the list.
      * Update necessary data for the enemy robot (bearing, heading, energy, position, shootAbleScore..)
      * In case there have only one enemy robot, we will lock radar with achieving always target enemy to shot
      * If the targeting robot has status is dead or has shootAbleScore is greater than shootAbleScore of our robot, we will convert the target into a robot that has just been scanned
      * The calculation methods used:
  + Calculate the coordinate of the enemy robot



According to the picture we have:

Với

According the nature we have

In conclusion, The coordinate of the enemy is

* + Calculate shootAbleScore of the enemy robot: Bases on energy and distance between enemy robot has just scanned and our robot. If energy is smaller and distance is closer, the smaller shootAbleScore.

Robot en = enemyList.get(e.getName());  
if (en == null) {  
 en = new Robot();  
 enemyList.put(e.getName(), en);  
}  
en.absoluteBearingRadians = e.getBearingRadians();  
en.setLocation(new Point2D.Double(myRobot.x + e.getDistance() \* Math.sin(getHeadingRadians() + en.absoluteBearingRadians),  
 myRobot.y + e.getDistance() \* Math.cos(getHeadingRadians() + en.absoluteBearingRadians)));  
en.lastHeading = en.heading;  
en.name = e.getName();  
en.energy = e.getEnergy();  
en.alive = true;  
en.scanTime = getTime();  
en.velocity = e.getVelocity();  
en.heading = e.getHeadingRadians();  
en.shootAbleScore = en.energy < 25 ? (en.energy < 5 ?  
 (en.energy == 0 ? Double.MIN\_VALUE : en.distance(myRobot) \* 0.1) :  
 en.distance(myRobot) \* 0.75) : en.distance(myRobot);  
if (getOthers() == 1) {  
 setTurnRadarLeftRadians(getRadarTurnRemainingRadians());  
}  
if (!targetBot.alive || en.shootAbleScore < targetBot.shootAbleScore)  
 targetBot = en;

* ***If the robot is in a solo mode:***
  + - * Create a new robot represent the enemy robot and set some information for it (absolute bearing angle, distance between that robot and our robot, current velocity of that robot)

Robot enemy = new Robot();  
enemy.absoluteBearingRadians = getHeadingRadians() + e.getBearingRadians();  
enemy.dist = e.getDistance();  
enemy.velocity = e.getVelocity();

* + - * Check if the robot is moving or not (based on speed). If it moves (velocity > 0 ), set its direction of movement by checking the calculated velocity value. If the value is negative, the robot will move backward, otherwise, the robot will move forward.

if (enemy.velocity != 0) {  
 lateralDirection = Utility.sign(enemy.velocity \* Math.sin(e.getHeadingRadians() - enemy.absoluteBearingRadians));  
}

* + - * Create a wave for our robot and set appropriate values for it (our shot position, the enemy robot’s coordinates, the enemy’s movement direction)

Wave wave = new Wave(this);  
wave.gunLocation = new Point2D.Double(getX(), getY());  
Wave.targetLocation = Utility.project(wave.gunLocation, enemy.absoluteBearingRadians, enemy.dist);  
wave.lateralDirection = lateralDirection;

* + - * Set segment values (enemy distance, current enemy velocity, previous enemy velocity)

wave.setSegmentations(enemy.dist, enemy.velocity, preEnemyVelocity);

* + - * Update the enemy robot’s previous velocity to its current velocity then set the wave bearing angle to be the absolute bearing angle of the enemy robot. We then set up the radar to turn an angle after it has been normalized within the range . Then the rotation angle o the radar = absolute bearing angle of the enemy robot – the angle of the robot’s gun + the offset angle calculated by the wave

setTurnGunRightRadians(Utils.normalRelativeAngle(enemy.absoluteBearingRadians - getGunHeadingRadians() + wave.mostVisitedBearingOffset()));

* + - * Set the energy of the bullets fired based o the energy of us an enemy to limit energy expenditure in a wasteful manner and shoot.

wave.bulletPower=Math.min(3, Math.min(this.getEnergy(), e.getEnergy()) / (double) 4);  
setFire(wave.bulletPower);

* + - * Reduce bullet damage if our energy is relatively small and the enemy distance is too far.

if (getEnergy() < 2 && e.getDistance() < 500)  
 wave.bulletPower = 0.1;  
else if (e.getDistance() >= 500)  
 wave.bulletPower = 1.1;  
setFire(wave.bulletPower);

* + - * Check the remaining energy of the robot. If there is still enough energy, continue adding an event to create waves for the next wave

if (getEnergy() >= wave.bulletPower)   
 addCustomEvent(wave);

* + - * Perform a scan of the enemy robot while moving, then reset the radar rotation to continue to adjust the scan angle for the next wave.

movement1VS1.onScannedRobot(e);  
setTurnRadarRightRadians(Utils.normalRelativeAngle(enemy.absoluteBearingRadians - getRadarHeadingRadians()) \* 2);

### run method

When start the battle, we initialize default value for some attributes:

* Size of the battle field assigns to battleField.

battleField.height = getBattleFieldHeight();  
battleField.width = getBattleFieldWidth();

* Coordinate, default energy assigns to variable which represents for our robot

myRobot.x = getX();  
myRobot.y = getY();  
myRobot.energy = getEnergy();

* Current coordinate of our robot assigns to lowest target point.

targetPoint.x = myRobot.x;  
targetPoint.y = myRobot.y;

* We have AMOUNT\_PREDICTED\_POINTS = 150. Take the positions that can be moved around our robot in a circle and put in the list of possible coordinates to calculate the risk of each location giving the direction of movement for our robot.

updateListLocations(AMOUNT\_PREDICTED\_POINTS);

* Create a robot to represent the enemy robot that we are aiming at. Since we have not yet targeted any specific object, keep its status is dead.

targetRobot = new Robot();  
targetRobot.alive = false;

* The gun turn setting doesn’t depend on the direction of the robot turn so that the robot can move in one direction and still shoot the target in the other direction.

setAdjustGunForRobotTurn(true);

* The radar turn setting doesn’t depend on the direction of the gun turn so that the robot can shoot in one direction and still scan the target in the other direction.

setAdjustRadarForGunTurn(true);

* Set the radar turn right infinitely.

setTurnRadarRightRadians(Double.POSITIVE\_INFINITY);

* In an infinite loop, with each round will perform the following actions:
  + Update data of our robot (location, previous direction, current direction, energy, current radar direction)
  + Check each robot in the list of the enemy robot, if the result when the current time subtracts scanned time is greater than 25, we know the data is because it isn’t updated after 25 ticks. We will assign dead to the status of that robot. After that, we check the targeting robot is that robot, we also assign dead to the status of the targeting robot
  + Setting for robot move
  + If targeting robot’s status is alive, shoot it

while (true) {  
 myRobot.lastHeading = myRobot.heading;  
 myRobot.heading = getHeadingRadians();  
 myRobot.x = getX();  
 myRobot.y = getY();  
 myRobot.energy = getEnergy();  
 myRobot.gunHeadingRadians = getGunHeadingRadians();  
 Iterator<Robot> enIterator = enemyList.values().iterator();  
   
 while (enIterator.hasNext()) {  
 Robot r = enIterator.next();  
 if (getTime() - r.scanTime > 25) {  
 r.alive = false;  
 if (targetRobot.name != null && r.name.equals(targetRobot.name))  
 targetRobot.alive = false;  
 }  
 }  
 movement();  
 if (targetRobot.alive)  
 shooting();  
 execute();  
}

* If the robot is in solo mode:
  + Set default lateral direction is 1 (go ahead)
  + Set the value of the variable which stored the previous velocity of the enemy robot as 0 (when starting the game, the enemy robot doesn’t have the previous velocity so we set it as 0)
  + Turn radar right infinitely to scan enemy robot

lateralDirection = 1;  
preEnemyVelocity = 0;  
do {  
 turnRadarRightRadians(Double.POSITIVE\_INFINITY);  
} while (true);

### changeColor method

* Create a static variable to change color with a random color for robot

The color uses RGB color values in rane đến (0, 255)

static Random random = new Random();

private void changeColor() {  
 setColors(new Color(random.nextInt(255), random.nextInt(255), random.nextInt(255)),  
 new Color(random.nextInt(255), random.nextInt(255), random.nextInt(255)),  
 new Color(random.nextInt(255), random.nextInt(255), random.nextInt(255)),  
 new Color(random.nextInt(255), random.nextInt(255), random.nextInt(255)),  
 new Color(random.nextInt(255), random.nextInt(255), random.nextInt(255)));  
}

# REFERENCES

##### [1] <https://robowiki.net/wiki/GuessFactor_Targeting_Tutorial>

##### [2] <https://robowiki.net/wiki/Wave_Surfing>

##### [3] <https://robowiki.net/wiki/Wave_Surfing/True_Surfing>

##### [4] <https://robowiki.net/wiki/Wave_Surfing/GoTo_Surfing>

##### [5] <https://robowiki.net/wiki/Movement_Profile>

##### [6] <https://robowiki.net/wiki/Rolling_Averages>

##### [7] <https://robowiki.net/wiki/Segmentation>

##### [8] <https://robowiki.net/wiki/Virtual_Bullet>

##### [9] <https://robowiki.net/wiki/Waves>

##### [10] <https://robowiki.net/wiki/Melee_Strategy#Evaluating_your_Melee_bot>

##### [11] <https://robowiki.net/wiki/Anti-Gravity_Movement>

##### [12] <https://robowiki.net/wiki/Corner_Movement>

##### [13] <https://robowiki.net/wiki/Pattern_Movement>

##### [14] <https://robowiki.net/wiki/Minimum_Risk_Movement>

##### [15] <https://robowiki.net/wiki/Head-On_Targeting>

##### [16] <https://robowiki.net/wiki/Linear_Targeting>

##### [17] <https://robowiki.net/wiki/Circular_Targeting>

##### [18] <https://robowiki.net/wiki/Pattern_Matching>

##### [19] <https://robowiki.net/wiki/GuessFactor_Targeting_Tutorial>

##### [20] <https://en.wikipedia.org/wiki/Cartesian_coordinate_system>

##### [21] <https://en.wikipedia.org/wiki/Polar_coordinate_system>

##### [22] <https://www.w3schools.com/java/java_hashmap.asp>