Student ID: R05944004

Name: Cheng-Wei Wu

# Assignment 1
# Pioneer 3-DX Simulation

## Report

## • Part C

For part C, I try to use the built-in function “getKey()”of the class <ArKeyHandler> to fetch the standard keyboard control input signal from the user. However, I seemed to forget use while loop to detect the input so that my original program only fetched the first keyboard control input, and ignored all the following control input. After referencing to the “Aria Reference Manual ver 1.1.7” and the book “Programming Mobile Robots with Aria and Player” published from Springer, I finally adopted the method of using the “Functor”, registering the callback function to the new “KeyHandler” and adding the created “KeyHandler” to the robot. I created the new class called “CallbackContainer” which was demonstrated the usage in the previous book. The following processes are the basic concepts:

1. Declare the class <CallbackContainer>
2. Declare the callback functions for detecting the keyboard control input from user in the class <CallbackContainer>
3. Setup both the velocity and angular velocity in each callback function according to their own features
4. Declare the <ArFunctor1C> in the main function for new keyHandlers for each control, and also register the callback function of each functor for the KeyHandlers
5. Add the new KeyHandlers to the original KeyHandler

The reason why I use Functor to design the keyboard control is that I didn't need to fetch the keyboard input in the infinite while loop, I can just register the callback function to let the system detect the keyboard interrupt automatically. Another reason is that the previous book suggest users implementing the keyboard control via this way.

And my design for the keyboard control is list as follows:

| | |
|---|---|
| • Up(↑): | forward at the speed 0.5m/s |
| • Down(↓): | backward at the speed 0.5m/s |
| • Left(←): | rotate at the angular velocity 2.5°/s |
| • Right(→): | rotate at the angular velocity -2.5°/s |

- If user doesn't hit any key, both the velocity and angular velocity will decrease to 0

## • Part D

I created a function called "isObstacleBySonar" which will get the distance data from the front sonars which indexes from "2" to "5". Since there are small angles between each sonar data, I defined danger distance to be "333" for the sonar 3 and sonar 4 while the sonar 2 and sonar 5 are "666" due to the real situation that Pioneer 3-DX moves along the direction of its heading angle and the sonar 3 and sonar 4 are closer to the front of the Pioneer 3-DX than sonar 2 and sonar 5. Then I call this function to check if there is an obstacle in front of the robot before setting the velocity and angular velocity while the user trying to press the keyboard to control the robot. In conclusion, I stopped the robot if there is an obstacle in front of it before setting the velocity of the robot.

At first, I only adopted the readings from the sonar 3 and sonar 4, but in some cases, the robot may bump into the obstacles. So I also read the data from the sonar 2 and sonar 5 to avoid such cases.

## • Part E

According to the values shown on the MobileSim, the true pose and the odometric pose are not the same, their meanings are different. The true pose means the real position of the robot on the map, so it will be changed when the robot is moving. On the other hand, the odometric pose will be reset to zero if I close the terminal then reconnect my program to the MobileSim. It's similar with the robot's "own" feelings, which means that the values of odometric pose are determined from robot's point of view. For each turn of execution, robot may feel like it starts without any previous rotation or translation which means that it seems to be on the original point no matter where is it. And I think that's the main difference between their physical meanings.

I design my moving process to be:

1. Compare the robot pose and the target pose, and calculate the angle between them
2. Rotate the robot to fit the correct angle for the next moving process
3. Moving the robot to the target pose via setting the velocity
4. Rotate the robot to fit the target pose's angle

To reach the goal on the spec, I use the simple and intuitive way to conduct the whole process. In the rotate phase, I set high angular velocity to let the robot rotate fast to save time; however, since high angular velocity cannot decrease to zero immediately, I set several thresholds to bound the angular velocity stage by stage to decrease it smoothly for buffer space to reach the exact angle for moving the robot. (to my surprise, I tried to set negative angular velocity after it and let the robot sleep for a while, and finally set the angular velocity to be zero, this method can increase the whole efficiency via setting higher angular velocity at first and decrease to zero in less time)

Next, I calculated the distance between the current robot position and the target point, and move the robot straightforward. Different from the previous method, I set the velocity to be its limit speed at first, and then set a small distance threshold to change its velocity to be negative. Similarly, I let the robot sleep for a while and set the velocity to be zero finally. Thus, I wrote total 3 functions to manipulate these processes as follows:

- void roateRobotToMove(ArRobot *robot, ArPose *target)

  → rotate the robot to the correct angle to move the robot at first
- void rotateRobotToFinish(ArRobot *robot, ArPose *target)

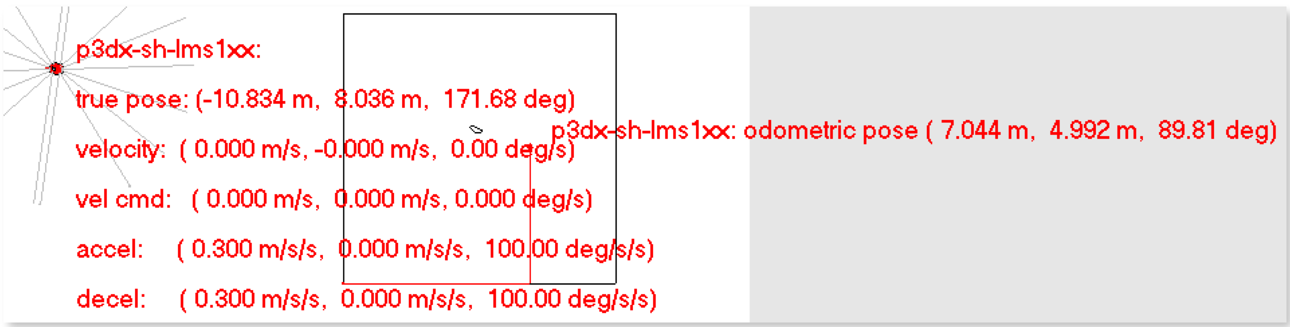
  → rotate the robot to fit the final requirement
- void moveRobot(ArRobot *robot, ArPose *target)

  → move the robot fast at the most of the distance between robot and target point

## • Results

The following figures are the results of the Part E (note that the odometric pose is different from the true pose in MobileSim):

• Input: (7, 5, pi/2)   Time: 17.38 sec   Distance Error: 0.044 m   Degree Error: 0.09º

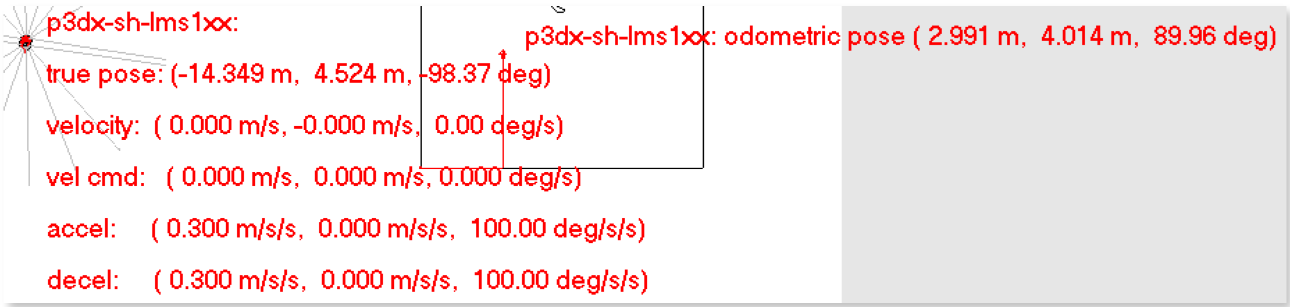


```
p3dx-sh-lms1xx:
true pose: (-10.834 m,  8.036 m,  171.68 deg)
velocity:  ( 0.000 m/s, -0.000 m/s,  0.00 deg/s)
vel cmd:   ( 0.000 m/s,  0.000 m/s, 0.000 deg/s)
accel:     ( 0.300 m/s/s,  0.000 m/s/s,  100.00 deg/s/s)
decel:     ( 0.300 m/s/s,  0.000 m/s/s,  100.00 deg/s/s)
p3dx-sh-lms1xx: odometric pose ( 7.044 m,  4.992 m,  89.81 deg)
```

**Figure 1.**

The result of Part E (7, 5, pi/2)

• Input: (3, 4, pi/2)   Time: 13.755 sec   Distance Error: 0.016 m   Degree Error: 0.04º

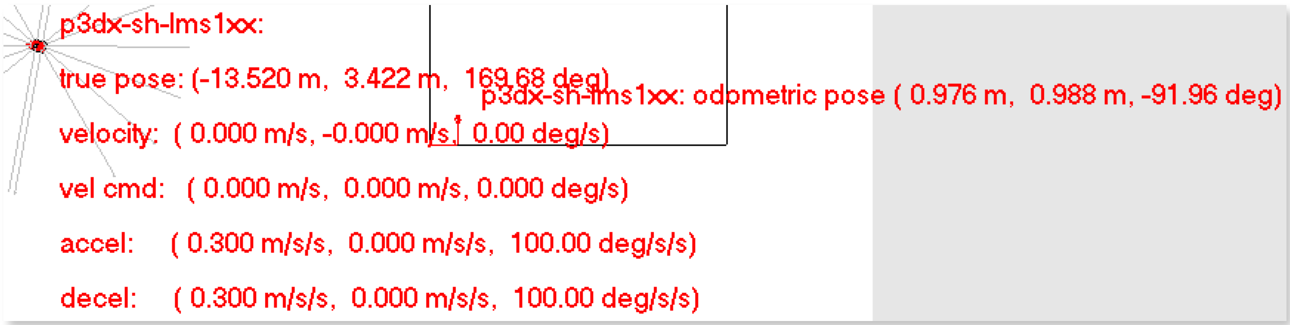


```
p3dx-sh-lms1xx:
true pose: (-14.349 m,  4.524 m, -98.37 deg)
velocity:  ( 0.000 m/s, -0.000 m/s,  0.00 deg/s)
vel cmd:   ( 0.000 m/s,  0.000 m/s, 0.000 deg/s)
accel:     ( 0.300 m/s/s,  0.000 m/s/s,  100.00 deg/s/s)
decel:     ( 0.300 m/s/s,  0.000 m/s/s,  100.00 deg/s/s)
p3dx-sh-lms1xx: odometric pose ( 2.991 m,  4.014 m,  89.96 deg)
```

**Figure 2.**

The result of Part E (3, 4, pi/2)

• Input: (1, 1, -pi)   Time: 12.013 sec   Distance Error: 0.026 m   Degree Error: 1.96º

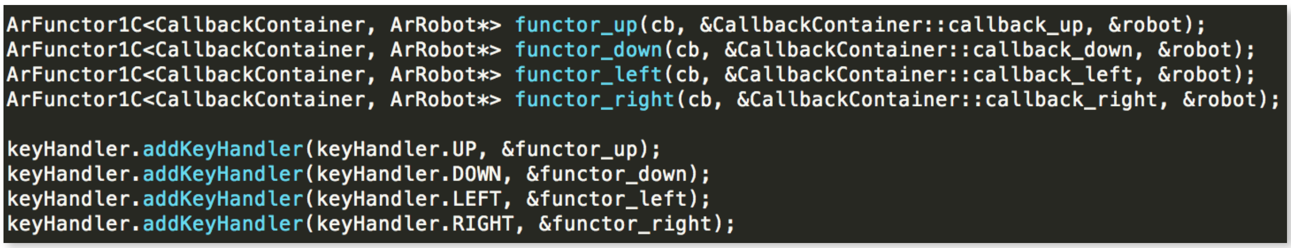


```
p3dx-sh-lms1xx:
true pose: (-13.520 m,  3.422 m,  169.68 deg)
velocity:  ( 0.000 m/s, -0.000 m/s,  0.00 deg/s)
vel cmd:   ( 0.000 m/s,  0.000 m/s, 0.000 deg/s)
accel:     ( 0.300 m/s/s,  0.000 m/s/s,  100.00 deg/s/s)
decel:     ( 0.300 m/s/s,  0.000 m/s/s,  100.00 deg/s/s)
p3dx-sh-lms1xx: odometric pose ( 0.976 m,  0.988 m, -91.96 deg)
```

**Figure 3.**

The result of Part E (1, 1, -pi)

## • Codes

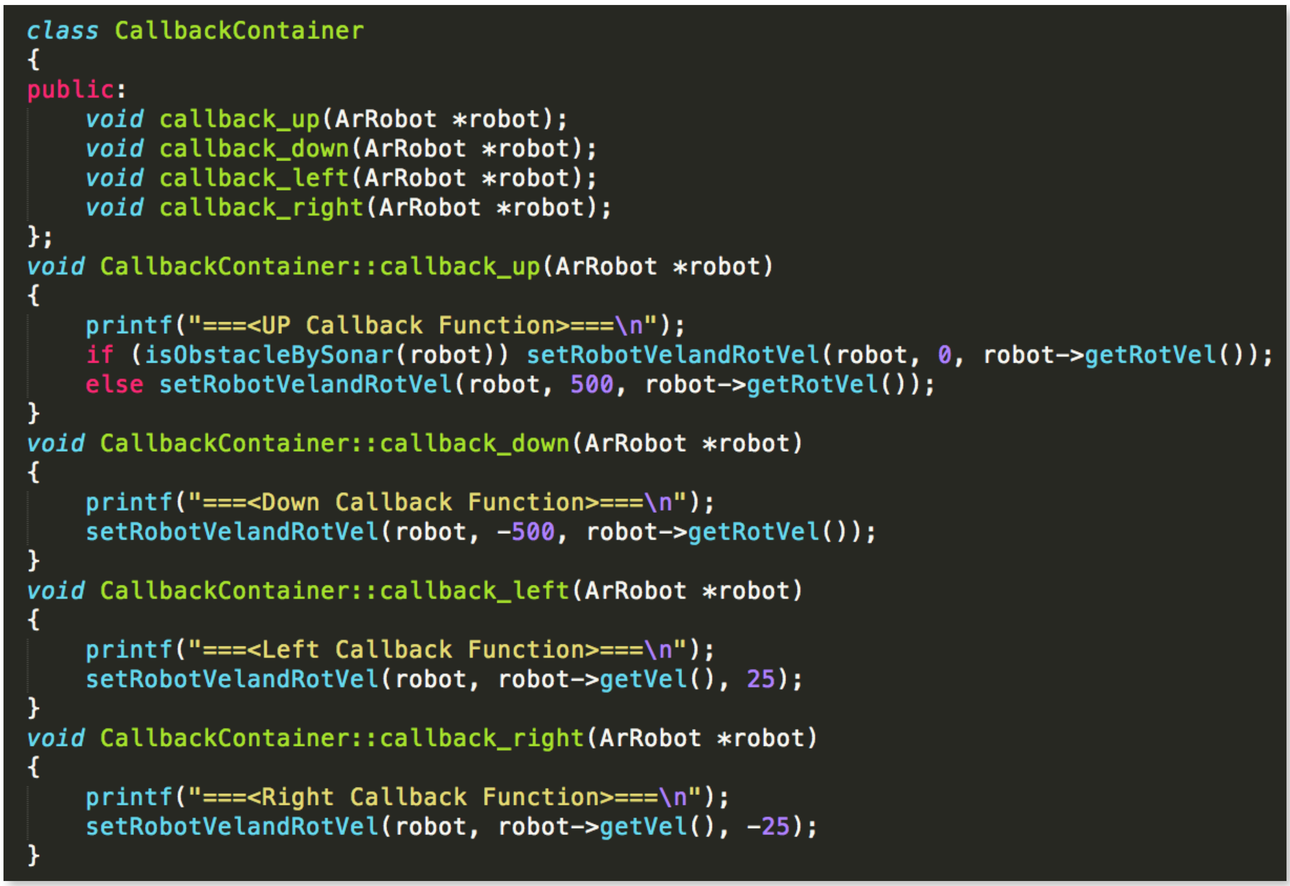


```
ArFunctor1C<CallbackContainer, ArRobot*> functor_up(cb, &CallbackContainer::callback_up, &robot);
ArFunctor1C<CallbackContainer, ArRobot*> functor_down(cb, &CallbackContainer::callback_down, &robot);
ArFunctor1C<CallbackContainer, ArRobot*> functor_left(cb, &CallbackContainer::callback_left, &robot);
ArFunctor1C<CallbackContainer, ArRobot*> functor_right(cb, &CallbackContainer::callback_right, &robot);

keyHandler.addKeyHandler(keyHandler.UP, &functor_up);
keyHandler.addKeyHandler(keyHandler.DOWN, &functor_down);
keyHandler.addKeyHandler(keyHandler.LEFT, &functor_left);
keyHandler.addKeyHandler(keyHandler.RIGHT, &functor_right);
```

**Figure 4.**

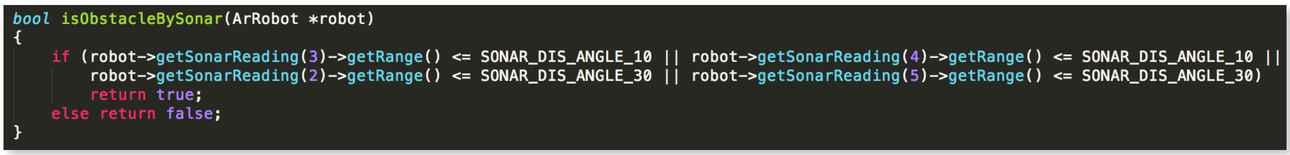
Declaration of ArFunctor1C for the part-C (keyboard control)
and the process of adding new key handler to the original one



```
class CallbackContainer
{
public:
    void callback_up(ArRobot *robot);
    void callback_down(ArRobot *robot);
    void callback_left(ArRobot *robot);
    void callback_right(ArRobot *robot);
};
void CallbackContainer::callback_up(ArRobot *robot)
{
    printf("===<UP Callback Function>===\n");
    if (isObstacleBySonar(robot)) setRobotVelandRotVel(robot, 0, robot->getRotVel());
    else setRobotVelandRotVel(robot, 500, robot->getRotVel());
}
void CallbackContainer::callback_down(ArRobot *robot)
{
    printf("===<Down Callback Function>===\n");
    setRobotVelandRotVel(robot, -500, robot->getRotVel());
}
void CallbackContainer::callback_left(ArRobot *robot)
{
    printf("===<Left Callback Function>===\n");
    setRobotVelandRotVel(robot, robot->getVel(), 25);
}
void CallbackContainer::callback_right(ArRobot *robot)
{
    printf("===<Right Callback Function>===\n");
    setRobotVelandRotVel(robot, robot->getVel(), -25);
}
```

**Figure 5.**

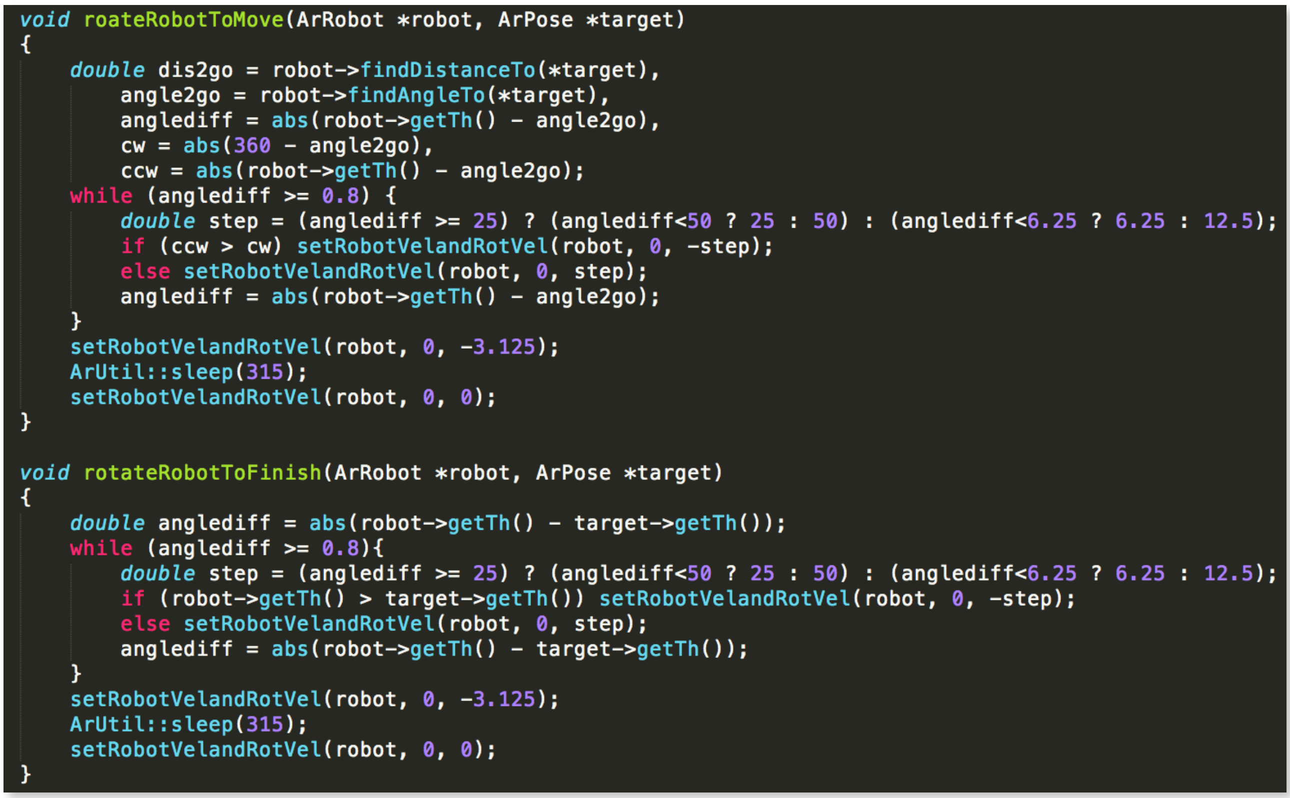
Declaration of the class <CallbackContainer>
and its callback functions



```
bool isObstacleBySonar(ArRobot *robot)
{
    if (robot->getSonarReading(3)->getRange() <= SONAR_DIS_ANGLE_10 || robot->getSonarReading(4)->getRange() <= SONAR_DIS_ANGLE_10 ||
        robot->getSonarReading(2)->getRange() <= SONAR_DIS_ANGLE_30 || robot->getSonarReading(5)->getRange() <= SONAR_DIS_ANGLE_30)
        return true;
    else return false;
}
```

**Figure 6.**

Obstacle avoidance detection via sonar's data

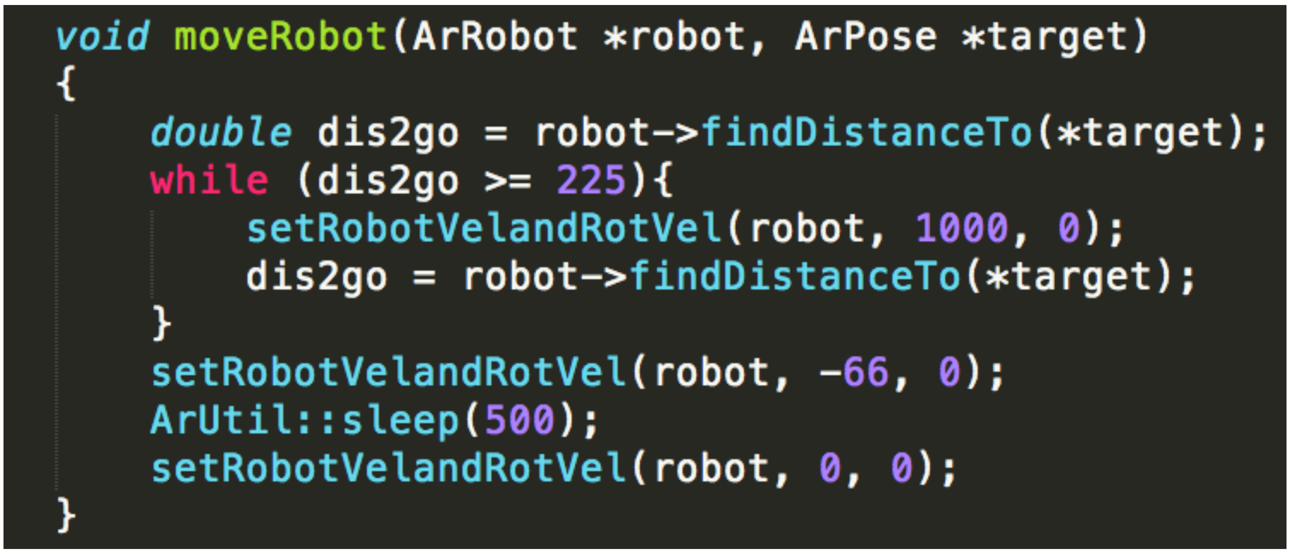


```
void roateRobotToMove(ArRobot *robot, ArPose *target)
{
    double dis2go = robot->findDistanceTo(*target),
        angle2go = robot->findAngleTo(*target),
        anglediff = abs(robot->getTh() - angle2go),
        cw = abs(360 - angle2go),
        ccw = abs(robot->getTh() - angle2go);
    while (anglediff >= 0.8) {
        double step = (anglediff >= 25) ? (anglediff<50 ? 25 : 50) : (anglediff<6.25 ? 6.25 : 12.5);
        if (ccw > cw) setRobotVelandRotVel(robot, 0, -step);
        else setRobotVelandRotVel(robot, 0, step);
        anglediff = abs(robot->getTh() - angle2go);
    }
    setRobotVelandRotVel(robot, 0, -3.125);
    ArUtil::sleep(315);
    setRobotVelandRotVel(robot, 0, 0);
}

void rotateRobotToFinish(ArRobot *robot, ArPose *target)
{
    double anglediff = abs(robot->getTh() - target->getTh());
    while (anglediff >= 0.8){
        double step = (anglediff >= 25) ? (anglediff<50 ? 25 : 50) : (anglediff<6.25 ? 6.25 : 12.5);
        if (robot->getTh() > target->getTh()) setRobotVelandRotVel(robot, 0, -step);
        else setRobotVelandRotVel(robot, 0, step);
        anglediff = abs(robot->getTh() - target->getTh());
    }
    setRobotVelandRotVel(robot, 0, -3.125);
    ArUtil::sleep(315);
    setRobotVelandRotVel(robot, 0, 0);
}
```

**Figure 7.**

Declaration of the functions for rotating the robot (before and after the translation)



```
void moveRobot(ArRobot *robot, ArPose *target)
{
    double dis2go = robot->findDistanceTo(*target);
    while (dis2go >= 225){
        setRobotVelandRotVel(robot, 1000, 0);
        dis2go = robot->findDistanceTo(*target);
    }
    setRobotVelandRotVel(robot, -66, 0);
    ArUtil::sleep(500);
    setRobotVelandRotVel(robot, 0, 0);
}
```

**Figure 8.**

Declaration of the functions for translating the the robot to the target point