Introduction

The goal of the project is to create a system to help a rather simple robot explore a map, find a few balls, and move those balls to a light source. This is all done inside a simulator.

The main challenge is creating a system with a suited and efficient architecture. The simulator itself is based on the Khepera-robot which is frequently used for robot experiments within artificial intelligence. The robot has 8 distance sensors, 8 light sensors and two wheels.

The only information the robot got about its surroundings, is the distance each wheel has moved, and values from the sensors. Apart from this, everything has to be created by you, in software.

The exercises themselves are written in a different document. The manual of the simulator is also recommended reading. Links to these can be found on it's learning.

The Robot

As mentioned in the introduction, the robot provides a limited set of resources. The controller has as its task to interpret the available data, and control the robot.

Moving around

Related code examples:
`TurnLeft.java, TurnDegrees.java, MoveForward.java`

The robot's wheels are controlled individually. In order to turn on the spot, set the wheel speed of the left and right wheel to (x, -x) (turn right).

With some thought it is possible to make the robot turn an exact number of degrees. When the robot turns on the spot like above, there is an *direct relation between the wheel counters and the angle of the robot's turn*. A change for each 3 ticks of the wheel counter, the robot has turned one degree.

An example: The robot starts, and the left counter is 0. It turns to the right until left counter is 270 (and the right i -270). At this point the robot has turned 90 degrees to the right.

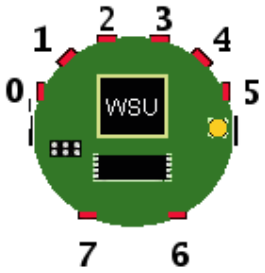
The same method is useful for controlling the distance in which a robot moves forward. For knowing where you are, watching the wheel counters is the key activity.

The wheel counters can be abused in various manners, and of course, you can turn by setting the wheels to different speeds, it just becomes a bit more complicated...

The sensors

Related code examples:
These are included in the robot extras zipped archive.
`MoveToWall.java, WallFollower.java (with the simulator)`

As indicated in the image, the sensors are numbered from 0 to 7. A few useful constants are found in the examples.



There are two different types of sensors: distance and light value. Both provides a value, but are subject to noise. To reduce the influence of noise, reading a sensor multiple times, and then averaging, may be a good idea.

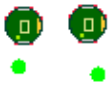
The distance sensors give a reading from 0 (far away from an object/wall) to 1023 (very close to the object or wall). The light sensors provide the values ranging from 0 (strong light) to 500 (no light).

It is hard to know which readings should be used to decide whether and object is a wall or not. This depends on your implementation of the robot's understanding of the surrounding.

One oddity to the light sensor is that it gives a lower reading when the robot is right at the light source, than if there is a small gap between.

Due to the crudeness of the sensors, it is not possible to decide whether and object is straight ahead, or to an angle compared to a sensor. This means that a sensory reading of for instance 250 could be a wall at a fair distance, or a object much closer but at an angle to the sensor.

The example situations below would give a similar reading from the right sensor:

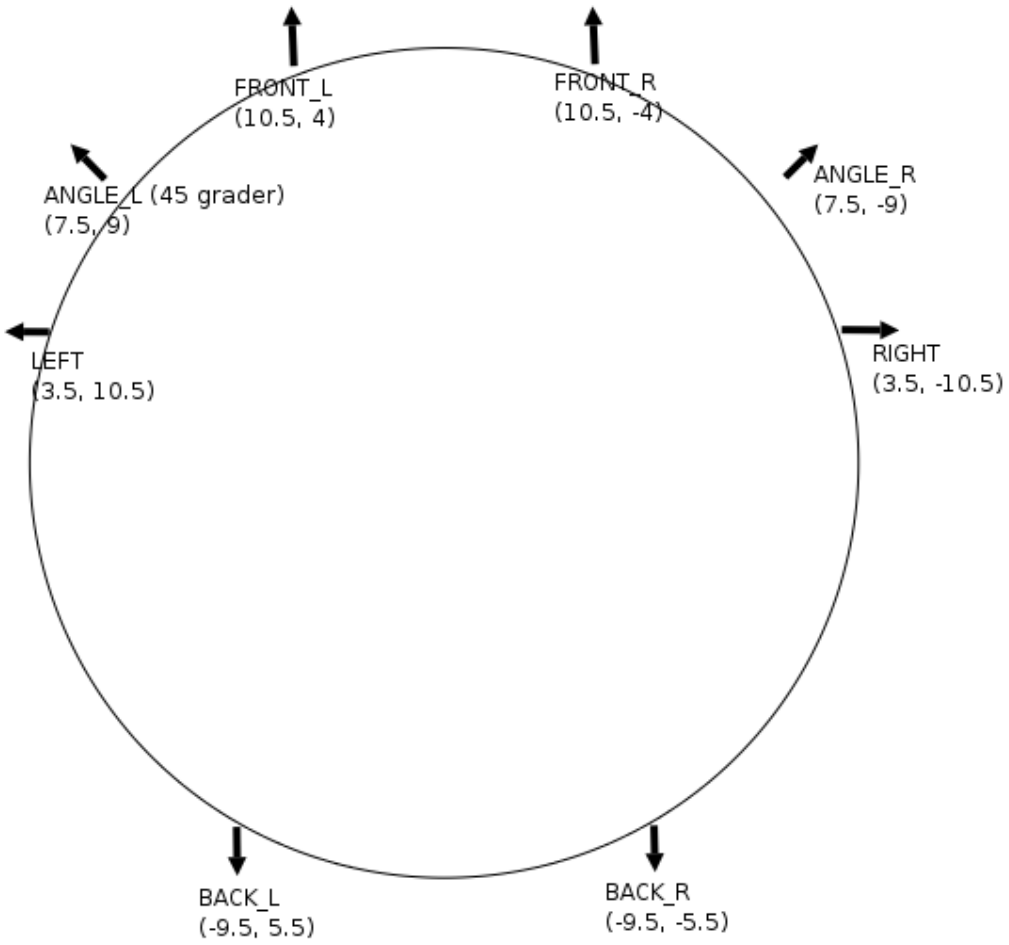


Interpreting the sensory values

The way of getting good knowledge from the sensor except for the general "Help, wall ahead!" is to know (a) where each sensor is located on the robot, and (b) in which direction each sensor is pointing.

The map of the robot simulator consists of a grid of 500 x 500

The placement on sensors on the robot is as follow (the numbers indicates position compared to the robot's center.



The challenge in exploiting this information lies in translating the sensory inputs into an internal representation of the world. How to do this requires experimentation, and the solution depends on each particular implementation. It is helpful to simplify the data collection somewhat, for instance by dividing the area around the robot in squares of, for instance, 3 x 3 pixels. Experimentation is required to find the optimal solution for *your* situation.

Compiling

In order to run a controller, you have to

- Unzip the simulator
- Have your controller inherit the RobotController class
- Position the controller in the controller/ directory

- Have the rest of your classes in classpath, the easiest solution being dumping all .class files in the controllers/ directory.

While it is possible to use simple tools like TextPad + the javac command, a proper IDE like JBuilder makes life easier. These two options are explained below.

The command line, how to compile and run the simulator

Create a directory for the project. Download the "Robot simulator zip" file from it's learning and unzip it in this directory.

In order to compile, for instance, `WallFollower.java` putting the class file in the correct directory, enter the src directory and use the following command: `javac -d ../controllers -sourcepath ../src WallFollower.java`

In order to run the directory: `java -jar WSU_KSim.jar`

It is not required to restart the simulator each time you recompile the controller.

An alternative: JBuilder

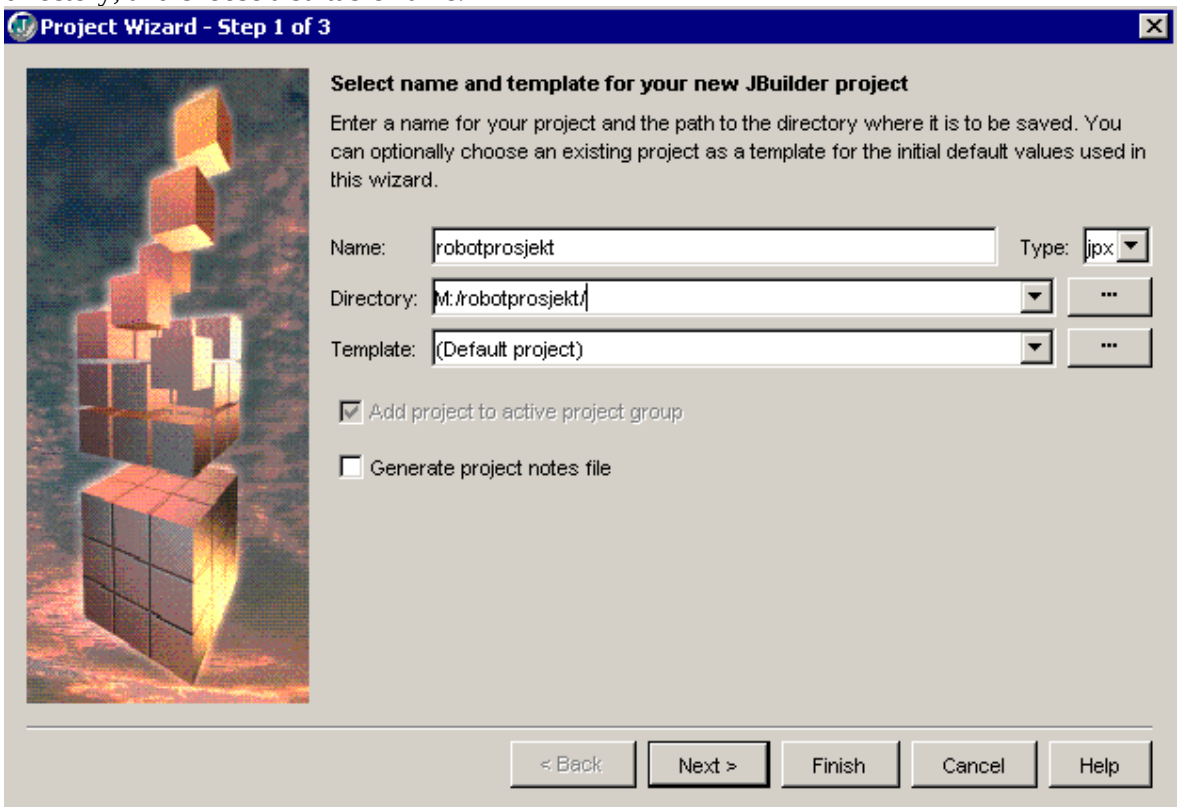
JBuilder (or any other Java-IDE like Eclipse) will make your life much happier compare to messing around with Textpad. Textpad may be an ok editor, but lacks features like debugging compared to JBuilder.

JBuilder is installed on the terminal servers, but unless you have done so previously, you have to download a license file from Borland. This is free for JBuilder Personal (version 9 is the one installed at NTNU). Go to http://www.borland.com/ (unfortunately you have to provide name, etc). You can also download JBuilder for use at home.

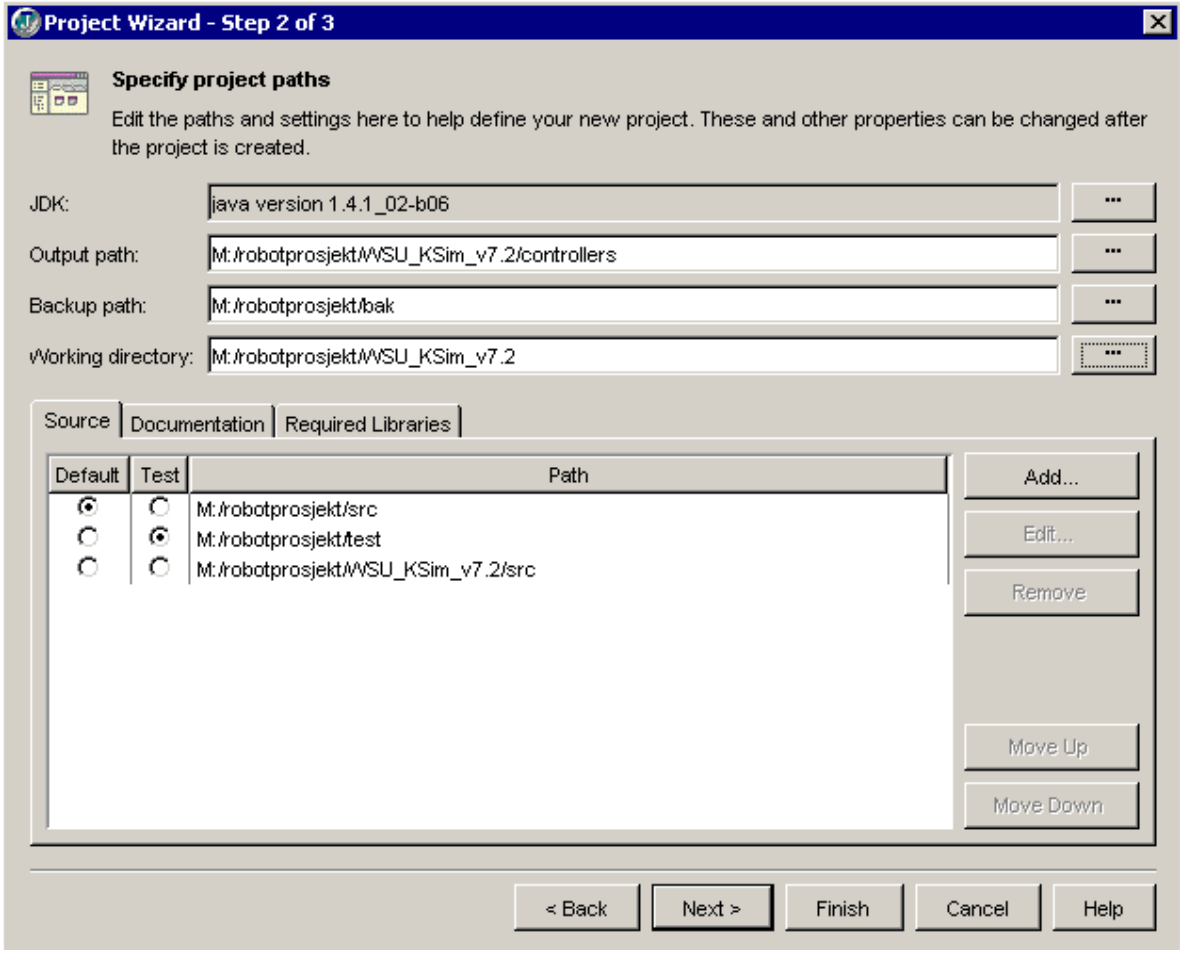
After getting JBuilder up and running, it is time to creating your project, and including the simulator.

1. Create a directory (i.e. `m:\robotproject`
2. Download the simulator, and unzip it to for instance `m:\robotproject\WSU_KSim`
3. Open JBuilder

4. Create a new project (File -> New Project). Choose your robotproject directory, and choose a suitable name.

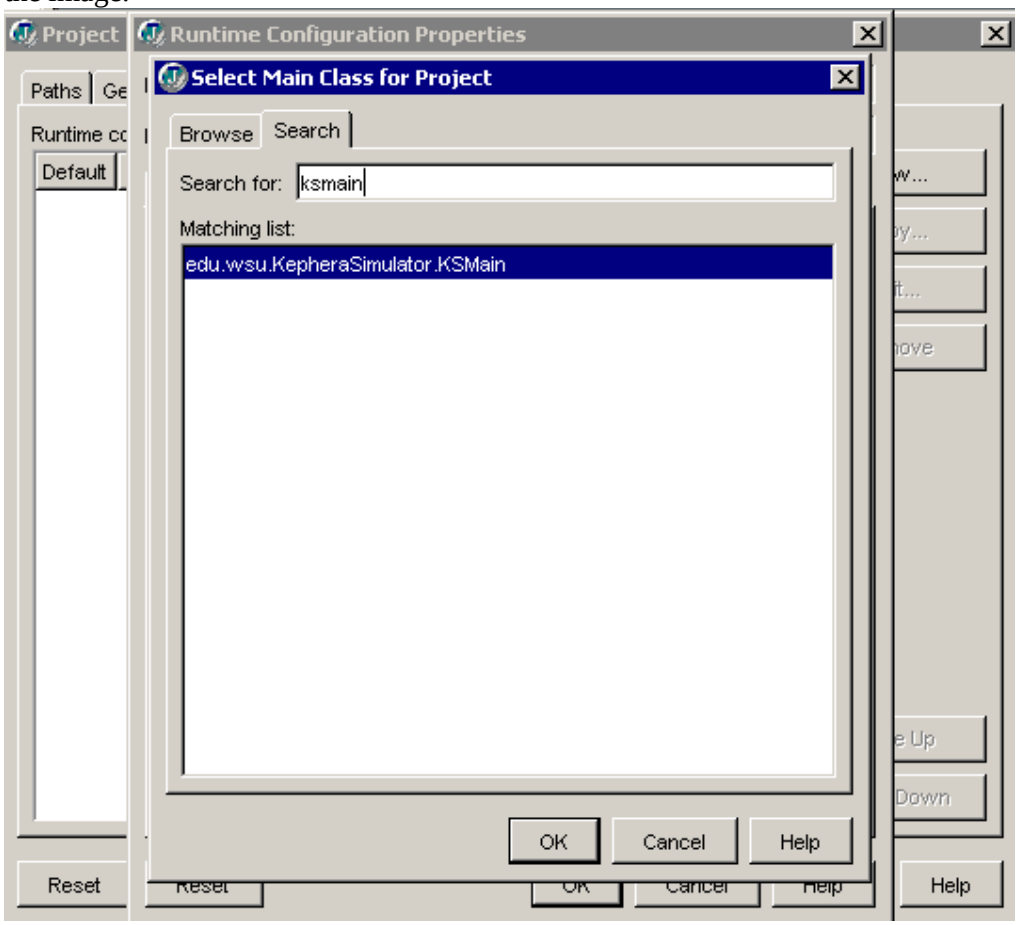


5. Choose your catalogs like shown in the image. Remember to create a link to the simulator (add)

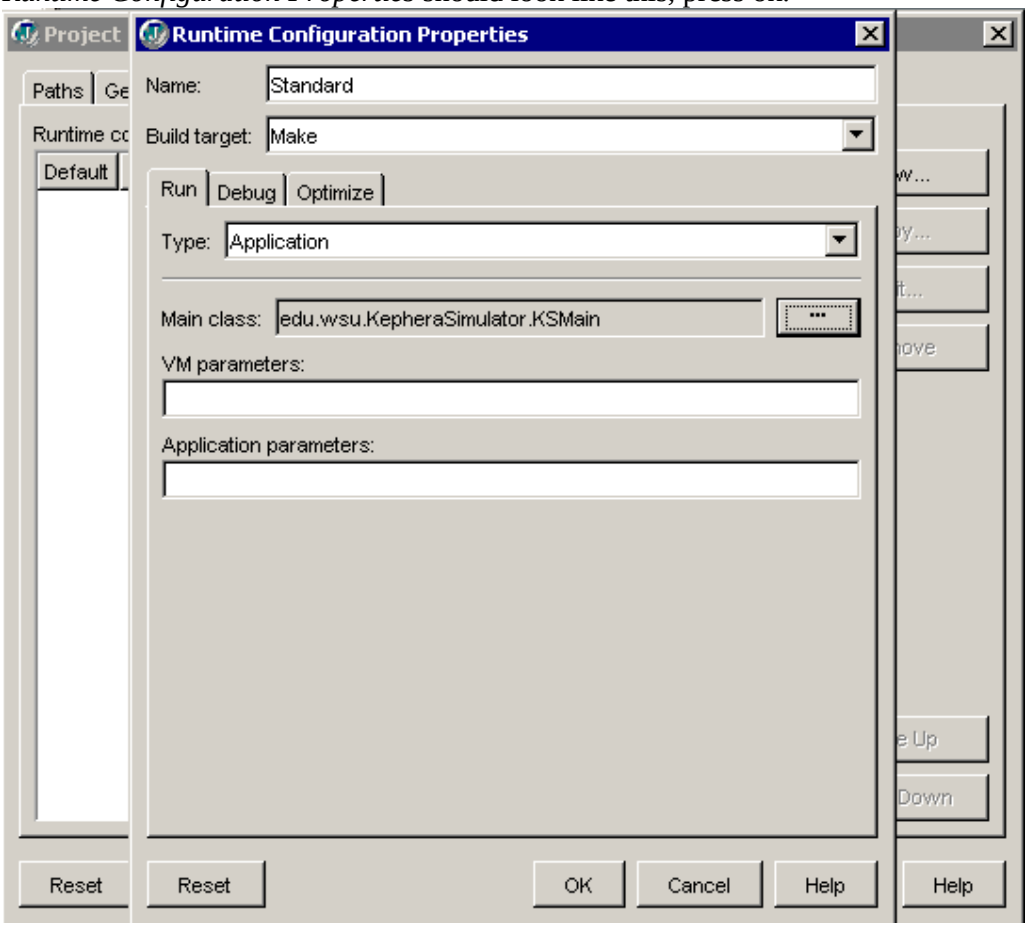


6. Push next, next, and finish. At this point the project is created. The next step is telling it which class contains the main() method. Go to *Run -> Configurations*. Press *New* and choose *Main Class*. Enter KSMain as shown in

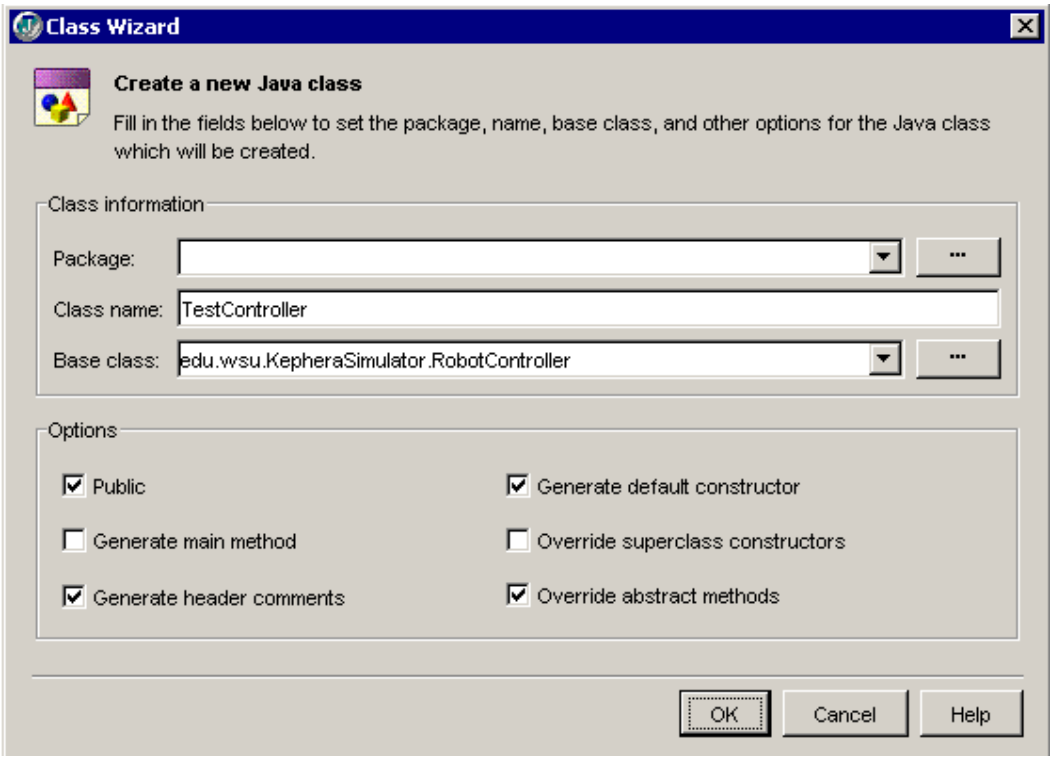
the image.


Project
Runtime Configuration Properties
Select Main Class for Project
Paths
Runtime cc
Default
Browse
Search
Search for:
ksmain
Matching list:
edu.wsu.KepheraSimulator.KSMain
OK
Cancel
Help
Reset
Help

7. *Runtime Configuration Properties* should look like this, press ok.



You are now ready to create your first test controller. Create a new class through *File -> New Class*. And enter the data shown below:



In the `doWork()` method, enter `setMotorSpeeds(5, -5);`, and press the green arrow on the tool bar. (Or press `F9`) to run the simulator.

Example Code.

Included in the "Robot extras zip" are some example codebits. These examples introduce basic concepts like movement and measurement.

It may be worth noticing that while the examples are easy to understand, they are hardly suited as building blocks of a larger system.

Links

The simulator can be downloaded from it's learning or directly from the simulators official webpage.

NOTE: The simulator on it's learning is a slightly modified simulator which makes the sensor readings more accurate. It is reccomended that you use this modified version but feel free to use the official version if you want a harder challenge.

FAQ

1. **After running for a while, the robot suddenly stops, even though it has a motor speed set.**

The problem occurs once in a while. Make sure running the doWork() function does not take too long time each time it is called. If it is necessary to pause while inside doWork(), use the sleep() method of the RobotController class. Another option is doing setMotorSpeeds() each time the doWork() method is called.

2. **I turn exactly 90 degrees, but after a while the robot starts deviating from the planned course. Why?**
   There is no guarantee that the doWork() method is run at the exactly same intervals, or often enough. In addition to this, the robot may move 0.01 degrees to much, which accumulates over time. Check TurnDegrees.java on how to avoid this problem (included in the robot extras file).
3. **Do I have to restart the simulator each time I have recompiled the robot?**
   No.
4. **May I modify the simulator?**
   No.
5. **The sensors are useless, the readings flutters too much!**
   Make multiple calls to getDistanceValue() or getLightValue(). Each call will return a new value, which you can use when averaging.
6. **Weird things happens, and once in a while the robot fails at tasks it usually can handle just fine**
   Keep an eye open for synchronisation issues between the thread running the simulator, and the thread running your controller. This is a major issue on heavily loaded computers. Just because your dowork() function has been called again, does not mean that the robot has actually moved, or the world has been updated. Or the opposite, between each "tick" of the simulator, you may move more than one step forwards. Hardware with power to spare seems to cure this problem, unfortunately the terminal servers do not always have power to spare. Make some checks if this becomes an issue.

Last modified: Mon Jan 9 09:04:15 MET 2006