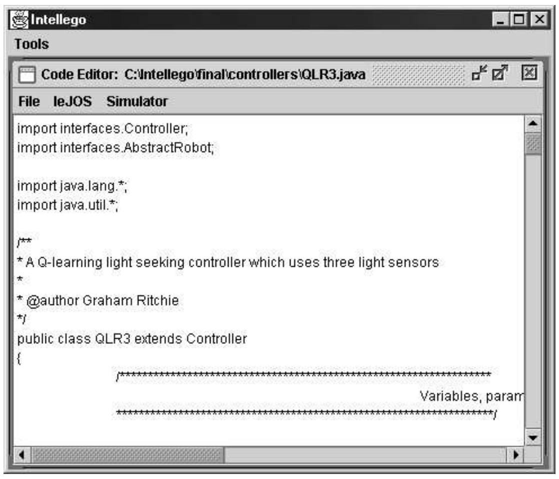
Intellego is a system designed for robotics researchers, and to use it to its full potential some experience of programming in Java is necessary. However the simulator component can be used without programming knowledge to either run existing controllers in simulation, or download them to the RCX. This document provides both an explanation of the main features of the user interface, and a developer's guide on how to implement extra components such as controllers and simworlds.

Start up:

Instructions as to how to set your system up and run Intellego are provided in the README.html file in the main Intellego directory. You need the Java development kit (jdk) and a leJOS distribution to run Intellego to its full potential. You can run the existing code with the Java runtime environment (jre), but you won't be able to download controllers to the RCX. You must set your PATHs and CLASSPATHs to the appropriate values, as outlined in README.html, and then run the correct script or batch file to start the system.

Once you have started the system the main graphical user interface screen will be displayed, and you can then access the Code Editor and Simulator components from the 'Tools' menu in the top left hand corner of the window.

Code Editor:

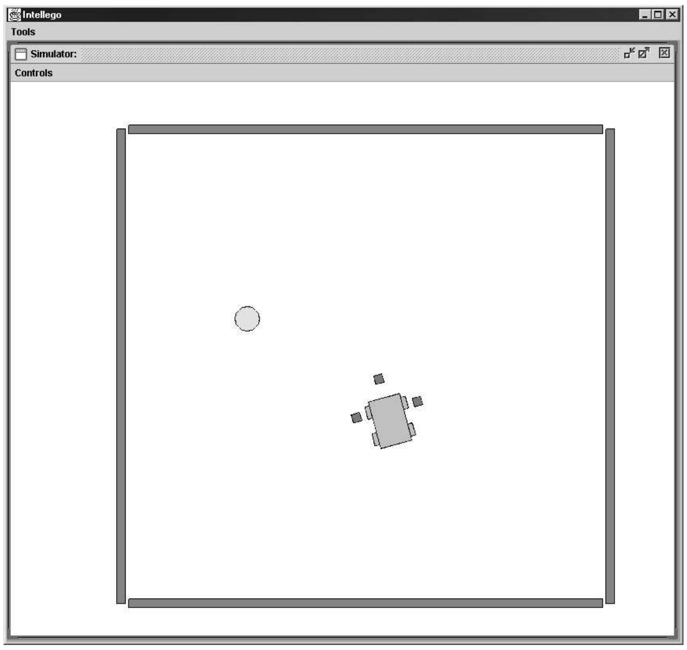


```
import interfaces.Controller;
import interfaces.AbstractRobot;

import java.lang.*;
import java.util.*;

/**
* A Q-learning light seeking controller which uses three light sensors
*
* @author Graham Ritchie
*/
public class QLR3 extends Controller
{
                /*******************************************************************
                                                                        Variables, param
                *******************************************************************/
```

The code editor window

The code editor component is a simple text editor which allows you to open text files and edit them from within the main GUI. It is intended for controller Java files, but will serve as an editor for any text file. To start a new code editor select 'Code Editor' from the 'Tools' menu. The 'File' menu deals with the opening, closing and saving of files just like any other text editor. The 'leJOS' menu allows you to attempt to compile your code, and subsequently download it to the robot. If you just want to try and compile your file (with lejosc) select 'Compile', if you want to download the controller to the real RCX, and link it with all other necessary files, select 'Link & Download'. These commands are performe d by making the appropriate calls to the leJOS executables, any messages returned from these programs will be displayed to you in a dialog box within Intellego. The 'Simulator' menu allows you to attempt to run the current controller file in the simulator. You can compile the file (with javac) by selecting 'Compile' from the Simulator menu, and then run it the simulator with 'Open controller in simulator' menu item.

Simulator:



The main simulator window: displaying the LightWorld SimWorld, and a single SimRCX

To run the simulator select 'Simulator' from the 'Tools' menu. An empty simulator window will then be displayed. To run a simulation you must first select a SimWorld, select 'Pick SimWorld' from the 'Controls' menu, and a dialog box will pop up showing compiled class files in Intellego's 'simworlds' directory. If the world you want to use is not in this directory, then navigate to the appropriate directory (and make sure that your CLASSPATH environment variable is set to the appropriate class path - NB it is probably simplest to keep all your SimWorlds in the simworlds directory). Select the required SimWorld and click on 'Open', if the world is a valid Intellego SimWorld it will then be displayed in the main simulator window. If your file is not a valid SimWorld you will be informed and you will need to open a valid SimWorld before a run can begin.

Once you have opened your world, you can then add robots to it. Select 'Add Robot' from the 'Controls' menu, and once again select the required Controller class file. (by default Intellego starts in its 'controllers' directory, and it is probably easiest to keep Controller files here.) If the class is a valid Controller, a dialog box will pop up asking for the initial parameters of the robot – an x coordinate, a z coordinate and a bearing. Enter these details, click 'OK' and a robot icon will be displayed in the main window. More robots can be added in the same way.

To start a simulator run, select 'Start Simulation' from the 'Controls' menu. You can stop the simulation at any time, by selecting 'Stop Simulation' from the 'Tools' menu.

Creating a Controller:

A controller is the program of the robot, it always has robot associated with it, and can interact with the robot in certain ways. In Intellego controllers must be written in Java. The robots that an Intellego Controller can deal must all meet the AbstractRobot interface (see source file: interfaces/AbstractRobot.java) which defines all the methods available to a controller. These methods are:

```
public abstract void forward();
public abstract void forward(int time);
public abstract void backward();
public abstract void backward(int time);
public abstract void right();
public abstract void right(int time);
public abstract void left();
public abstract void left(int time);
public abstract void stopMoving();
public abstract void beep();
```

The function of each of these methods is fairly clear from their names, for example a call to `forward()` will set the robot moving forward, and a call to `forward(100)` will set the robot moving forward for 100 milliseconds, and then stop the robot moving. Calls to `right()` and `left()` will set the robot turning within its footprint the appropriate direction. `stopMoving()` will stop the robot moving immediately, and `beep()` causes the robot to emit a beep sound (in simulation this beep will come from the host computer).

The input methods are:

```
public abstract int getSensor1();
public abstract int getSensor2();
public abstract int getSensor3();
```

These will return the value of the appropriate sensor. What value these functions will return is based on the type of the sensor, which is set in the Controller code. An Intellego Controller must therefore tell the system the type of sensors it needs. Currently two sensor types are supported, touch and light. To set the sensors the controller must include the following line in the main body of the class (i.e. not inside a method):

```
private int[] sensors={Controller.SENSOR_TYPE_VALUE,
                       Controller.SENSOR_TYPE_VALUE,
                       Controller.SENSOR_TYPE_VALUE};
```

Where the `Controller.SENSOR_TYPE_VALUE variable` can be one of:

- `Controller.SENSOR_TYPE_LIGHT`
- `Controller.SENSOR_TYPE_TOUCH`

All controllers that are intended to run in Intellego must also meet the Controller interface (see source file: interfaces/Controller.java) and therefore must implement the following methods:

```java
public void initController(AbstractRobot r);
public int[] getSensors();
public void run();
public void halt();
public AbstractRobot getRobot();
```

The Controller interface is a 'program by contract' interface, as long as a controller implements all of these methods in some way it will be accepted by the system, and will run. However each of the methods has a 'contract' associated with it, and the developer must obey these contracts, or system behaviour is undefined. Outlined below are the contracts for each method. A controller runs in a single thread (this is mainly to allow the controller to be run in the simulation) it can start its own threads, but it must take responsibility for cleaning them up when it is finished, or again system behaviour is undefined. A controller is initialised with a call to `initController()`, and actually set running with a call to `run()`, and can be subsequently stopped with a call to `halt()`.

`public void initController(AbstractRobot r);`

This method initialises the controller, and will only ever be called once in the lifetime of the controller, whereas `run()` can be called many times, so any variables or data structures etc. that are meant to persist in between stops and starts of the controller' s thread, or that should only be initialised once should be set up from within this method, not in `run()`.

`public int[] getSensors();`

This method should return this controllers sensor array (as described above) as an array of three ints.

`public void run();`

This method will be called to set the controller running, and so all the actual controlling of the robot should be initialised from here.

`public void halt();`

This method will be called when the simulator requires the controller to stop its thread. This method must therefore allow `run()` to return, a simple way to do this is by making `halt()` set some running boolean flag to false, and have `run()` check this flag every so often. This method is used instead of simply killing the controller's thread directly to allow the controller to do some sensible house-keeping before it quits, e.g. saving internal data to a file etc.

```
public AbstractRobot getRobot();
```

This method must return the AbstractRobot associated with this controller, i.e. the one passed to it in `initController()`.

The easiest way to see how to implement a controller is probably to look at the source code of the controllers that have already been developed (see source files in the controllers directory.).

Creating a SimWorld:

All Intellego SimWorlds must adhere to the SimWorld interface, which defines the methods described below. Intellego SimWorlds must be implemented in Java. The most basic function of a SimWorld is to deal with SimObjects (see below for further details about SimObjects), and the way in which they do this is up to the developer.

```
public void tick();
```

Performs one update loop of the world, i.e. should update each SimObject in the world.

```
public long getTime();
```

Returns the number of ' ticks' since this world was started.

```
public int getBrightness(double x, double y, double z);
```

Returns the light level at the specified co-ordinate.

```
public boolean hasObstacle(double x, double y, double z);
```

Checks whether there is an obstacle in the specified co-ordinate

```
public LinkedList getObjectList();
```

Returns this SimWorld' s object list.

```
public void addObject(SimObject o);
```

Adds an object to this SimWorld

This is a fairly low level interface, which allows the developer to create a SimWorld completely from scratch, with arbitrary physical laws, e.g. how light is modelled. Details of how to implement such a world are therefore left up to the developer. However a basic implementation of SimWorld, BasicSimWorld (see source file: simworldobjects/BasicSimWorld.java) models basic physical laws (e.g. collision detection, light radiating from a source or sources – see javadoc generated documentation for low-level details). BasicSimWorld is an abstract class and it contains no artefacts (light sources, walls, obstacles etc.) and so the easiest way to create an individual SimWorld is simply to extend BasicSimWorld. Individual SimObjects can then be added as required. Again the easiest way to see how to implement your own world is to look at the prewritten worlds (see the source directory 'simworlds', e.g. simworlds/LightWorld.java which models a world with a single light source and a number of walls).

Creating a SimObject:

Every object in a SimWorld must be an Intellego SimObject. These include robots, walls, lights, etc. The only SimObjects that have been implemented are SimRCX, SimWall, SimLight, SimLightSensor and SimTouchSensor (see source files in the simworldobjects directory). The SimObject interface is rather complex, and so for low-level details you should look at the Javadoc generated documentation on SimObject. A basic implementation has been written which deals with the most common features of SimObjects, BasicSimObject (see simworldobjects/BasicSimObject.java) and immobile objects such as SimWall and SimLight simply extend this class and add a few methods, SimRCX is more complex as it also implements the AbstractRobot interface and so must be able to model controller commands such as `forward()` etc. The Javadoc generated documentation provides low-level details of this class. It is simple to create new instances of the various SimObjects which have already been implemented and add them to a new SimWorld. If you want to create new types of SimObject then, once again, it probably easiest to look at the implementations of SimWall etc. and deduce what different sort of attributes the new object needs from there.

SimSensors are rather more complex SimObjects. They do not extend the BasicSimObject class, but implement the SimObject interface themselves. This is because sensors are tightly coupled with a SimRCX, and therefore establish their co-ordinates in relation to the robot, again see the Javadocs. SimSensors are the only SimObjects which are able to directly 'talk' to the SimWorld. SimLightSensor calls SimWorld's `getBrightness()` method, and SimTouchSensor calls SimWorld's `hasObstacle()` method. To implement different sensor types would involve creating new SimWorld methods for the sensor to talk to. This would mean updating the SimWorld interface, and would be more complex that adding a new SimObject, but is still fairly straightforward.

In general the simulator component is designed to be extended and improved upon. It a very modular system so extension and evolution should be relatively easy. Refer to the Javadocs for detailed interface specifications, and the Project Details for an overview.