COMP105P
2. Introducing the Simulator

Version: 2015-01

Mark Handley

M.Handley@cs.ucl.ac.uk

Contents

1 Introduction

To allow you to work on your robot programs in your own time we have written a robot simulator, which currently goes by the imaginative name of *RoboSim*[1]. The simulator supports the same synchronous ASCII API as our robots - the API is described in more detail in the document "*1. Introducing the Robots*". In this document we will describe how to use the simulator and the ways we know that it differs from the actual robot behaviour.

2 Running the simulator

The simulator is written in *Python*, which is an interpreted language well suited to rapid prototyping. Python also provides a cross-platform environment - the same code can be run on Linux, MacOS and Windows. We're using the pygame graphical toolkit, which provides fairly high performance graphics in a native manner on each operating system.

The simulator has been developed using Python 2.7 and pygame 1.9.1. Pygame supports most 2.x variants of python. There is supposed to be support for Python 3.x, but I have not tested this. Pygame only supports 32-bit installations of Python, so it is possible you may need to install a different version of Python if you've got a 64-bit version installed.

To run the simulator, first download the code from Moodle. It is supplied as a zip file, as a tarball, and (for MacOS) as an app file. Unpack the code into the directory where you wish to install RoboSim.

In the top level directory, you will find the following:

- `README` - instructions for starting RoboSim.
- `RoboSim` - the script to start the simulator.
- `scenery` - a directory containing a number of scenery files.
- `src` - a directory containing the RoboSim source files.

You may need to tailor RoboSim for your system.

2.1 Linux

Find out where your system has installed python. Typically you can do this by running the following from the command line in a terminal window:

```
which python
```

The response should say something like `/usr/bin/python`. If so, then you should not need to tailor RoboSim.

Check pygame is installed. Run python from the command line, and type "import pygame". If this doesn't return any errors, you're good to go.

[1]Suggestions for better names gratefully appreciated

Just run RoboSim from the command line by cd'ing to the directory you have installed RoboSim in, and running:

```
./RoboSim
```

If, when you typed `which python`, there was no response, then either python is not installed on your system, or it is not in your default command path. If it's installed, but not in your default command path, you can either edit the path (usually specified in your `.bashrc` file in your home directory) or change the first line of `RoboSim` to something like:

```
#!/path/to/where/you/installed/python
```

Your system may have more than one version of python installed. You can edit the first line of `RoboSim` to specify a particular version of python if you wish:

```
#! /usr/bin/env python2.7
```

You shouldn't need to do this unless the default version of python is really old. You can find out the version of python by running:

```
python --version
```

2.2 MacOS

Python is usually installed on MacOS by default, and is installed in `/usr/bin/python`. Unfortunately the version supplied with MacOS is unlikely to be compatible with pygame. Follow the instructions on the pygame website to install the correct version of python and pygame, then edit the first line of RoboSim (as in the Linux directions above) to point to the correct version of Python.

On the machines in Lab 4.06, the correct version of python is installed as `python2.7-32`**. You should edit the first line of RoboSim to specify this.**

You can also download the `Robosim.app` version of the simulator, but it's a much larger download, so it's better to install Python and pygame properly if you can.

You should be able to run RoboSim by double-clicking on it from the finder. You can also run RoboSim from the command line in a Terminal window. `cd` to the directory you have installed RoboSim in, and running:

```
./RoboSim
```

It's probably a good ideal to get used to using the Unix command line interface on MacOS, as you'll be using it a lot to run your programs on the robots.

2.3 Windows

Python is not installed on Windows by default. Download the MSI installer from `http://www.python.org/download/` and install Python. The default choices from the installer are OK. We have tested with Python 2.7, but other 2.x versions should also work. Follow the instructions on the pygame website to install pygame.

Download the zipfile version of RoboSim and extract the contents of the zipfile. In the toplevel directory you extracted, find the RoboSim file and rename it RoboSim.py (Windows is fussy about filename extensions, whereas Linux and MacOS are not). Double-click on RoboSim.py to start the simulator.

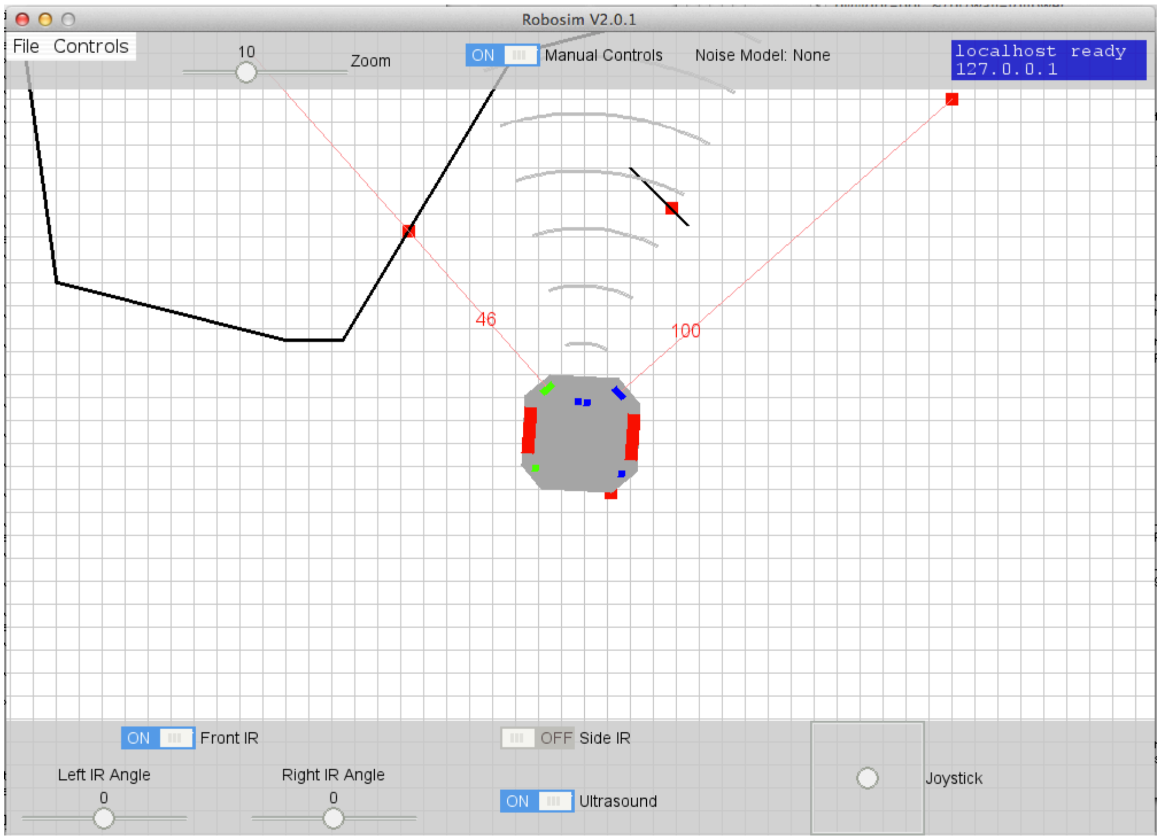


Figure 1: RoboSim GUI on MacOS

3 The RoboSim GUI

Figure 1 shows the RoboSim user interface, as it appears on MacOS. It is not very complicated. Along the top are a set of basic controls and displays. Below them is a simplistic view down on the top of the simulated robot.

The octagonal shape is the robot. In the figure it is facing up the page. The two large red rectangles represent the driving wheels. The robot's sensors are also shown:

- In the centre are two blue squares. These show the location of the ultrasound emitter and sensor.
- To either side of the front, there are two longer rectangles. These are the front infrared rangefinder sensors. The one on the right is blue and the one on the left is green.
- At the rear, behind the driving wheels are two small squares representing the side infrared rangefinders. Again, the right one is blue and the left one is green.

When a sensor is activated, its beam pattern is shown. In Fig. 1 the front infrared rangefinders and the ultrasound have been activated, but the side infrared rangefinders have not been activated.

The IR rangefinder beams are shown as lines and the ultrasound is shown as a series of arcs. Neither of these is quite a correct representation of the real beam patterns, but the model is good enough for most purposes and it is easy to visualize.

Fig. 1 also shows some walls, drawn as bold black lines. The sensors will detect these walls, and the robot can crash into them if your code doesn't command it to avoid them. Where a sensor beam detects a target in range, a red blob is shown on the target, and the range to the target (in cm) is shown next to the line. If there is no target in range, the red blob is placed at the maximum range of the beam. The left IR sees a wall at 46cm and the right does not see any wall, and is reporting 100cm (don't assume the robots will always report 100cm for maximum range though - they won't - but they will be very unreliable above 100cm).

The display of the sensors can be manually enabled using the switches in the manual controls panel at the bottom of the screen (you need to first display the manual panel using the switch at the top of the screen). If you leave these switches off, the sensor beams will be shown automatically when your code polls the sensor value. When your code stops polling, the beam will stop being shown.

The simulator also handles collision detection when your robot crashes into objects. When this occurs, the red blob on the back edge of the robot will move to show the location of the collision. Just like the real robot, only collisions on the front will be detected by the bump sensors.

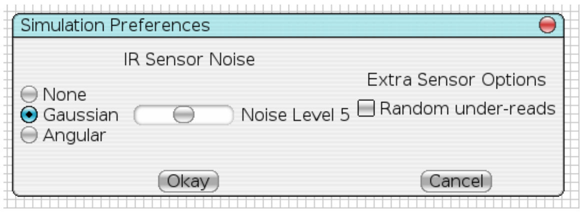
At the top of the screen a zoom control is provided, and at the top right the blue "LCD" shows messages that would be shown on the LCD panel on the top of the robot.

Finally on the manual control panel at the bottom right is a manual joystick control you can use to move the robot around the simulator.

On the menubar there are two menus.

- **File** allows you to load scenery files and quit the simulator.
- **Controls** allows you to select simulation preferences, and reset the robot's position to its starting point.

The simulation preferences panel allows you to set noise models for the IR sensors. You should aim for your code to work with no noise and both the two noise models with a range of noise levels - this will make your code more robust to sensor noise on the real robots.



None
[✓ Gaussian] Noise Level 5
Angular
Random under-reads

Figure 2: RoboSim Sensor Preferences

3.1 Scenery Files

In the `scenery` directory, there are a number of sample scenery files. There are currently five commands used to describe the scenery:

```
robot <xpos> <ypos> <angle>
```

This sets the initial robot position and direction in the file. The robot will be positioned at coordinates `<xpos>,<ypos>` and point `<angle>` degrees from north. Coordinates are integers in the range 0 to 2000.

```
wall <x1> <y1> <x2> <y2>
```

This positions a wall from coordinates `<x1>,<y1>` to coordinates `<x2>,<y2>`.

```
line <x1> <y1> <x2> <y2>
```

This draws a line on the floor from coordinates `<x1>,<y1>` to coordinates `<x2>,<y2>`.

Figure 3 shows RoboSim following a line on the floor.

```
cone <x1> <y1> <radius>
```

This draws a cone at coordinates `<x1>,<y1>` with radius `<radius>`.

```
target <x1> <y1> <radius> <text>
```

This draws a target at coordinates `<x1>,<y1>` with radius `<radius>` and containing text `<text>`. Targets may be used to check the robot has traversed a valid path for a task.

A very simple scenery file might look like:

```
robot 800 700 30
wall 5 50 50 5
wall 50 5 300 5
wall 300 5 350 20
wall 350 20 400 400
wall 400 400 600 450
wall 600 450 650 450
```

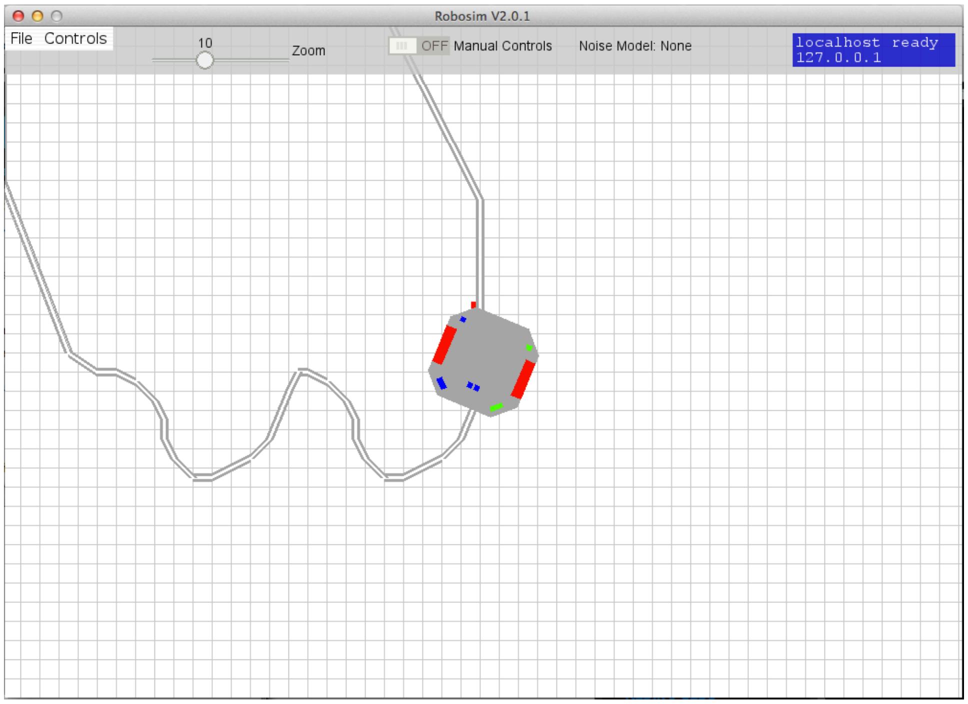


Figure 3: RoboSim following lines

4 Programming the Simulator

The API for programming the robots is described in "*1. Introducting the Robots*", and the simulator supports the same API. Your software should make a TCP connection to port 55443 on the computer the simulator is running on, and issue textual commands.

You can manually test this API using the telnet program. For example:

```
bash-3.2$ telnet localhost 55443
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
I L 45
.
I R -45
.
M LR 10 10
.
S IFLR
S IFLR 153 153
M LR 0 0
.
```

Commands you type are shown in bold, responses from RoboSim are not bold.

This sequence of commands points both front infrared rangefinders forwards, drives both motors forwards slowly, samples the range from the front infrared rangefinders and finally stops.

You code will need to issue commands like these, sending one at a time followed by a newline character, wait for a response, then send the next. In the case of commands that return no value, such as the motor speed commands above, only a "." is expected in response.

The Moodle page has the necessary C code functions to establish a local TCP connection to port 55443. You can use these as a starting point for your programs.

4.1 Compiling your code

When you compile code to run on the robots, you will use a C compiler that targets the ARM CPU architecture. The computer you are sitting at will[2] have an Intel x86 CPU, but the Raspberry Pi on the robots have an ARM CPU. When you compile C code, it compiles down to the native instruction set of the machine it is to run on. By default, this is the instruction set of the machine you're running the compiler on. To compile for the robots though you have two choices:

- you can copy your source code to the robot and compile it there.
- you can compile C code to run on an ARM CPU using a cross-compiler on your x86 CPU.

[2]almost certainly

We believe the former is easier, and suggest you use that approach. Don't forget to copy your source files back to your code repository afterwards though if you edit then on the robot.

When you compile your code to drive the simulator, you will be compiling your code to run on the machine the simulator runs on - usually the same machine you're sitting at.

Your C source code for driving the simulator and for driving the real robots via the Raspberry Pi may be identical (the robot and the simulator support the same API), but you'll need to compile it separately for the robot and the simulator. You cannot compile it for the simulator, and then copy the compiled code to the robot.

If you've got a compiled program, and you're not sure which architecture it was compiled for, you can use the *file* utility. For example:

```
bash-3.2$ file wall-follower
wall-follower: ELF 32-bit LSB executable, ARM, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
```

You can see that this particular executable program was compiled for an ARM CPU. If you try to run it on an x86 CPU, it will not work:

```
bash-3.2$ ./wall-follower
bash: ./wall-follower: cannot execute binary file
```

If you copy this file to the Raspberry Pi using *scp* and run it there, it will run normally. For the simulator though, we want a native x86 executable.

Once your code starts to get more complicated, you'll split it up into several source files. A common error when you're copying source files back and forwards is to forget to copy a particular source file. A revision control system like *subversion* or *git* can help a lot here. Last year a number of students used Dropbox for this - Dropbox is not really a good tool for this task because it's easy to get confused about different versions of files when you're working in a pair. It's better to learn to use a proper revision control system for this task, as you'll need one later in your course anyway.

It's also easy to forget to recompile a file you've copied. It's a good idea to use a utility called *make* to control the compilation process, so you don't get confused. Here's an example:

```
CC=gcc
CFLAGS=-g -Wall
all:    wall-follower
wall-follower:wall-follower.o motors.o sensors.o
        $(CC) $(CFLAGS) -o wall-follower wall-follower.o motors.o sensors.o
wall-follower.o: wall-follower.c wall-follower.h
        $(CC) $(CFLAGS) -c wall-follower.c
motors.o: motors.c motors.h
        $(CC) $(CFLAGS) -c motors.c
sensors.o: sensors.c sensors.h
        $(CC) $(CFLAGS) -c sensors.c
```

This code compiles the three source files wall-follower.c, sensors.c and motors.c separately to .o files, then links them together to create the wall-follower executable. If you change any of the .c or .h files, all the correct files will be recompiled just by taking `make`.

One gotcha with Makefiles is that the *make* utility treats the TAB character as being meaningful. In the 3rd and 4th lines, the whitespace after the "`:`" must be a TAB. Similarly the leading whitespace on the 5th and 7h lines must be a TAB, not space characters.

This particular Makefile also specifies some flags for the C compiler to use.

- `-g` This includes debugging symbols in the executable, so that when it crashes we can debug it using the debugger *gdb*.
- `-Wall`. Nothing to do with wall following - this specifies that the compiler should switch on the generation of `all W`arnings. This is always a good idea when compiling C code, as without it the compiler won't warn you about many silly errors.

When you have compiled your code, first run the simulator, then run the compiled native executable. From the command line:

```
bash-3.2$ ./wall-follower
```

It should connect via TCP to the simulator and start controlling the simulated robot.

4.1.1 MacOS and Linux

The instructions above are for Linux and for MacOS using the Unix command line in a Terminal window.

Linux usually ships with the GNU compiler tools by default. MacOS does not - you'll need to install XCode, free from Apple[3]: `http://developer.apple.com/xcode/`. Only download the latest XCode if you're running the latest version of MacOS. Otherwise you'll need an earlier version. XCode is a full IDE, not just the compilers. We really want this just for the compilers.

You could use an IDE such as Eclipse on Linux or XCode on MacOS, but I would recommend not doing so for simple programs. While IDEs can help you track calls through complex programs, then tend to isolate you from what is actually going on with the result that you might not understand understand some of the basics. You should however use a text editor that understands C code and highlights syntax. On MacOS, XCode can be used as just a text editor rather than a project manager. On Linux there are many good text editors - *gedit* is quite a simple one. Personally, I use *emacs* for everything, but the other half of the Unix world uses *vim*, which is also a very good editor. *Nano* is also usually installed, and is simple to use, but lacks syntax highlighting which is a very useful feature when programming.

4.1.2 Windows

Using Windows to develop code to run on Android (which is essentially Linux) is making life hard for yourself. But it can be done, and if you want to use RoboSim on windows and use the same code on the

[3]you also need to join their free developer programme

robots, you'll have to do so. You'll either need to install *cygwin* on your Windows machine or run Linux in a virtual machine using *VMware* or similar to virtualize your machine. Cygwin is probably simpler.

You can get cygwin from `http://www.cygwin.com/`. Cygwin consists of common Unix command line utilities, plus the same GNU compiler tools used on MacOS and Linux and the necessary code libraries to use them to build code that can talk to the simulator. Once cygwin is installed, you can compile your code using a Makefile in the same was as on Linux or MacOS.

4.1.3 Robots

Whatever development environment you use for the simulator, unless you copy source files backwards and forwards all the time, you will still need to edit your files *on* the robots. You can use *vi* (actually vim), *emacs*, or *nano* to do this. Nano is simplest, but it will be worth the time investment to learn vim or emacs.

5 Differences between the robots and the simulator

Our aim is for the simulator to be a faithful representation of the behaviour of our robots. It's not perfect though - the physics model is very crude and the sensors are "perfect" - unlike those on the real robots. Some other differences:

- The robots have an emergency stop and a self-test button. When these are pressed the robot generates a `W STOP` or `W TEST` message that it sends to your code and them it ceases to respond to commands. When the stop or test is cancelled, the robot generates `W RUN`. The simulator does not model this behaviour but your code should check for these unsolicited responses.
- There is usually an error of a few degrees in the direction the robot's front infrared sensors point compared to what you commanded. This error varies from robot to robot. The RoboSim sensors point exactly where you command them to point.
- When the robot hits an object, it sometimes rebounds a little and sometimes does not. Thus after the robot has stopped, the bumper switch may or may not remain pressed. The simulator almost always rebounds, so the bumper switch response is more transient.
- If the side of the robot snags an object, the robot will usually slew around without detecting the collision. This may mess up distance readings from the motor encoders if the snagged wheel skids. In the simulator, the robot always stops moving when a collision is detected, though it doesn't shut down the "motors".
- The RoboSim world is two-dimensional. The real world is not - the real robots cannot see very low objects.
- None of the walls in RoboSim are transparent or reflective of infrared light. The windows of the MPEB lift lobbies are.
- The IR sensors on the robots can misread in various circumstances (described in the Robots documentation). The simulator lacks a good model for when they misread.
- Ultrasound can reflect off smooth walls if it hits them at a shallow angle (as described in the Robots documentation). The simulator does not model this effect.

These differences are usually not sufficient to cause problems with code you have developed in the simulator subsequently being run on the robots, so long as you are aware of the limitations of the sensors. Usually you will have to do a little tuning though, especially with motor speeds.