COMP105P Lab Tasks Week 1-2

Tasks

Here are the tasks for this week (and first lab next week):

- Task 1.0. In the simulator, get the robot to go spin in a circle on the spot.
- Task 1.1. In the simulator, get the robot to go in a straight line and then stop.
- Task 1.2. In the simulator, using just the motors speed commands, get the robot to draw a square using timing to determine the length of the sides and the angles turned.
- Task 1.2b. When you've got task 1.2 working in the simulator, try it on a real robot.
- Task 1.3. In the simulator, get the robot to draw a 1-metre square using the motor encoders to measure the length of the sides and the angles turned.
- Task 1.3b. When you've got task 1.3 working in the simulator, try it on a real robot.

Hints and code

Since this is the first set of tasks we will help you out a bit and give you several hints plus a really simple source listing to get you going.

- Put each task (or subtask) in its own self-contained C file and name it appropriately. You will end up with four files, each of which should compile and execute cleanly when ran against the simulator. We will only test these simple tasks with the simulator. When submitting bundle all the tasks in a .zip and name it appropriately. There are a lot of students in the class and we do not want to have to guess which group originated which file and which zip archive - please take care to adhere to a sensible naming convention.
- Keep your code neat and tidy. Make sure you do everything as obviously as possible. Please add comments where it may not be immediately apparent what you are trying to do. We will actually be looking at the code you submit, so do not produce a jumbled mess.
- Make things as modular as possible - as mentioned in the first lecture your later tasks will build on earlier ones and the level of complexity will increase gradually. This usually means you will have to do the same basic things over and over again (e.g. parsing the readings of a certain type of sensor) - why not put those things in separate functions (and files later on when the tasks get big) that you can reuse.

- The teaching assistants are
  - Group A:

    Lynne Salameh L.Salameh@cs.ucl.ac.uk
    Ye-Sheen Lim Y.Lim@cs.ucl.ac.uk
  - Group B:

    Cosmin Cocora C.Cocora@cs.ucl.ac.uk
    Daniel Worrall D.Worrall@cs.ucl.ac.uk
  - Group C:

    Sebastian Friston sebastian.friston.12@ucl.ac.uk
    Andrew Macquarrie andrew.macquarrie.13@ucl.ac.uk,
  - Group D:

    Nicola Gvozdiev nikola.gvozdiev.10@ucl.ac.uk,
    Joseph Jacobs joseph.jacobs.12@ucl.ac.uk

  If you have any questions about the assignments feel free to email us. Please note that we will not do the tasks for you or code anything for you.

Here is one way to make the robot turn left. This is a complete code snippet and you can just paste it into a file called something.c and compile it with gcc like you would any other C program. Note that you will need to compile and execute this on a UNIX machine - Windows has its own TCP API, which is a bit different from the UNIX one. You also will, obviously, need to have the simulator running and listening when executing the code. Both the robot and the simulator listen for incoming TCP connections, so the routine should be mostly similar when programming the actual robot.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/socket.h>
6  #include <arpa/inet.h>
7
8  int main() {
9    char buf[80];
10   struct sockaddr_in s_addr;
11   int i, sock;
12
13   if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
14     fprintf(stderr, "Failed to create socket\n");
15     exit(1);
16     }
17
18   s_addr.sin_family = AF_INET;
19   s_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
20   s_addr.sin_port = htons(55443);
21
22   if (connect(sock, (struct sockaddr *) &s_addr, sizeof(s_addr)) < 0) {
23     fprintf(stderr, "Failed to connect socket\n");
24     exit(1);
25     }
26
27   while (1) {
28     sprintf(buf, "M L 10\n");
29     write(sock, buf, strlen(buf));
30     memset(buf, 0, 80);
31     read(sock, buf, 80);
32     }
33 }
```

And here is a breakdown of what the code does:

Lines 1-6 In order to have the program do useful things we need to import several files. The first four ones are pretty standard and you should have seen them in previous C courses. The last two have to do with opening and managing a socket (a TCP connection). You are likely to need the same set of imports for all the code you write.

Lines 9-11 Since this is a very simple program all the action will take place in main(). Before we get started we need a bunch of local variables. Among other things we need a string which we will use to send commands to the robot and a handle to the socket (which is just an integer which tells the operating system which socket we are operating on).

Lines 13-16 We create a TCP socket.

Lines 18-25 We populate the `s_addr` variable with relevant information for the socket and we connect to it (after this we can start using the socket). Line 20 specifies the port number we will try to connect to. The simulator and the robot run on the same port so you do not have to change that.

Lines 27-32 Up so far we have only written boilerplate code which opens a TCP connection to the simulator (or the robot if you want to run the code on the actual hardware). Here is where we program the robot. We have an endless loop (1 always evaluates to true) where we put the string `"M L 10"` (followed by a UNIX-style newline) into the string buffer, we send the buffer to the socket, we clear the buffer and we read from the socket to get the results. Note that we need to read the socket back - if you do not the robot will not move. For the right command strings look up the manual. In (slightly) more advanced tasks you will also need to parse the output.

Using picomms

Manually constructing commands and interpreting the responses can be somewhat tedious, though it does let you have complete control over what your code does. To make your code more readable, you may wish to use our picomms library. The program above, written using picomms, reduces to the following:

```
1 #include <stdio.h>
2 #include ``picomms.h''
3
4 int main() {
5   int speed = 10;
6   connect_to_robot();
7   initialize_robot();
8   set_motor(LEFT, speed);
9 }
```

The condition for using `picomms` is that you do not edit `picomms.c` or `picomms.h`, but separately compile them and link with them. This allows us to roll out changes or bugfixes to picomms during term, and you can easily adopt them. There is no documentation for `picomms`. This is not a bug, but deliberate. You should read `picomms.h` to figure out the available commands, then read `picomms.c` to understand what those functions actually do.