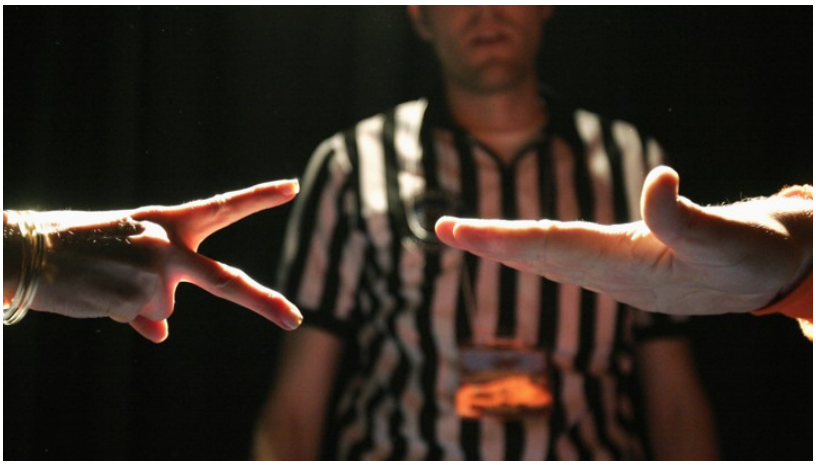
Laboratory 10: Simulator Supplement

Bryan Ruddy

29 May 2022

The music fades. The lights flash. The crowds roar. It's the most anticipated game of the year. Rock Paper Scissors. ...but what if there are no crowds, and you have to play the game from afar? Enter the simulator.



The ultimate battle. (Image from The Atlantic, `https://www.theatlantic.com/entertainment/archive/2015/12/how-rock-paper-scissors-went-viral/418455/`.)

This document describes the use of a micro:bit simulator for testing your radio code, if you don't have access to a physical micro:bit. There will be a few things you have to change about your code to make it work in the simulator, but it will allow you to run your code on two simulated micro:bits, side-by-side, so you can play a game against yourself. You have been provided with:

- This document (`commlab_simulator.pdf`)
- A code template (`commlab_simulator.py`)
- A simulation framework (the remaining files and directories in the .zip file)

We recommend you complete tasks 0-4 following the original lab instructions and using the test framework we provided, then translate your code to a version that works with the simulator for task 5.

Using the simulator

To run the simulator, first open a command prompt with access to Python. (Most of you will need the Anaconda Prompt.) Next, navigate to the folder containing the simulator. (You don't have to keep your own code in this same folder. Then use the following command to run the simulator:

In keeping with tradition, this lab uses yet a different method for accessing and using the simulator.

```
python server.py
```

You can also just run `server.py` in PyCharm or your other favourite development environment.

Next, open a web browser and go to `http://localhost:8000/`. You should see two simulated micro:bits, side by side. Each will allow you to load and edit code separately, and can run independently. However, they will act as though they can see each other on the radio link. There is also a "noise" feature that helps you ensure that your code works even when some data get lost.

We have tested the simulator successfully on current versions of Firefox and Chrome on Windows, and found that it does not work with Microsoft Edge.

Adapting your code

We have provided a fresh code template that has been modified to work better in the simulator. You should copy your code from the functions in tasks 1-4 into the corresponding functions in this new template. Once you have done this, make sure you put the `await` keyword before any function calls to the following:

- `microbit.display.show()`
- `microbit.display.scroll()`
- `microbit.sleep()`
- `radio.on()`
- `radio.send_bytes()`
- `radio.send()`
- `radio.receive_bytes()`
- `radio.receive()`
- `utime.sleep()`
- `utime.sleep_ms()`
- `utime.sleep_us()`
- `choose_opponent()`
- `choose_play()`
- `send_choice()`
- `send_acknowledgement()`
- `parse_message()`
- `resolve()`
- `display_score()`

Note that you definitely won't use all of these functions, and your code probably only uses a small number of them. The full set of asynchronous functions supported by the simulator has been listed for completeness.

You can then write your code for the `main()` function. If you have already written this code, look carefully at how the function is arranged in the new template, and ensure you move lines of code to the right spots instead of copying and pasting over the whole thing.

Properly emulating the behaviour of the display functions is difficult in the simulator, so we can't use all their normal features and have to mimic the behaviour in the `main()` function.

Testing your code

To test your code, run the simulator, then use the *Choose File* button above each micro:bit to load your adapted code. **IMPORTANT: You must change the value in MYID on at least one micro:bit, or you will not be able to communicate.** Click the *Run* button on each one to start the code, and you should see the game start with opponent selection on each. Have fun!

We have provided a button named *Noise On* to use in testing the robustness of your code. When activated, it causes that micro:bit to randomly lose 25% of the messages it sends and 25% of the messages it receives. The text box to the right of the micro:bit will show a message each time a message is lost. Because our code can't multi-task, it is possible for one micro:bit to be asking the user for a new play while the other is stuck waiting for a missed acknowledgement message. If your code is working correctly, the "stuck" micro:bit should get un-stuck once you select a new play on the other one.

Note that code robustness is part of the marking rubric!

This "sticking" effect can also happen on the physical micro:bits.

Once you have adapted your code to work in the simulator, the test framework we provided won't work any more. Thus, we don't recommend starting to use the simulator until you are ready for Task 5. (Don't worry, we will make our automated marking scripts work correctly regardless of whether or not you use the simulator.)

Finishing the lab

Please see the main lab document for full instructions for Task 5 and for questions and submission procedures.

Optional: How it works

The simulator consists of a simple web server, plus a webpage that runs Python code. The web server is written in Python, and uses HTTP POST requests to imitate the radio system. It tracks each instance of the simulator using a unique ID, and keeps track of which messages each one has seen so that every simulator has the chance to see every message sent while its radio is on. To make your life easier, we have provided a webpage that loads two simulators side-by-side, but if you wish you can open `ubit.html` in multiple browser tabs instead. Note that, just like on the real micro:bits, the radio starts over and forgets all past messages each time you call `radio.on()`. Feel free to have a look through `server.py` and the code in the `lib` folder that implements the micro:bit functions to see how it all works.

The webpages run Python code using a library called Pyodide, which compiles it to a form of assembly language that can be executed by web browsers. Pyodide has some limitations, which don't strongly affect our code, but it also enforces standard Python coding conventions that the version on the micro:bit does not. Mostly,

this is around the issue of asynchronous programming. On the real micro:bit, the Python interpreter actually is the operating system, and directly controls timed tasks and access to the hardware. Thus, you can write normal code, and the interpeter will keep the system running behind the scenes during your infinite loops. However, Pyodide does not have such access, and we have had to implement the operating system functions in Python using asynchronous programming in order to keep the browser from freezing. In asynchronous programming, multiple functions can run simultaneously, but Python must be told about each such function. In a function definition, we use the `async` keyword to do this. When calling a function, the `await` keyword tells Python that you want to wait for the function to finish before proceeding. The asynchronous `sleep()` function we use is special, and allows the web browser to complete its other tasks (like updating the display and reacting to buttons) while we wait.