Python Introduction and Setup

Original Author(s): Koen Koning & Michael Lees

During the lab exercises of this course you will implement several models, and use them to perform experiments. The first half of the course will focus on Cellular Automata (CA), whereas the second half will focus on networking models. This first introductional mini-exercise involves setting up the environment in which you will work for the rest of the course.

We'll be using Python for the lab exercises of this course. There are currently two major versions of Python available, Python 2 and Python 3. You are free to use either one, but we suggest you stick to Python 2 if you are not familiar with Python.

There are two important differences to bear in mind during this course: one is that integer division (returning the floor of the result, i.e. discarding any fractional component) is performed with the // operator in Python 3. The other is that `print` is a statement in Python 2, but a function in Python 3 (and thus requires parentheses when calling it, just like any other function).

A number of libraries will be used during this course:

- **matplotlib**: An extensive visualization library. Can be used to plot graphs, images, etc. We will primarily use the pyplot interface, which offers a number of functions which each modify some aspect of an internal figure (global state-machine).
  `http://matplotlib.org/`
- **NumPy**: Provides efficient matrices and other mathematical operations. Since Python is an interpreted language, it cannot efficiently handle big arrays or matrices. NumPy will therefore be used for the CA models to represent the 2D grid as as 2D matrix. This matrix can be directly drawn using matplotlib.
  `http://www.numpy.org/`
- **NetworkX**: Allows for easy creation and modification of networks, which will be the focus of the second part of this course. NetworkX features matplotlib compatibility, allowing networks to be shown by matplotlib.
  `http://networkx.github.io/`
- **PyICS**: A custom framework built for this course, making it easy to create, visualize and test models. For an overview of this library, refer to Section . For more details the documentation can be viewed using Python's help function, e.g. `help(pyics.Model)`.[1]
  See Blackboard

PyICS should be downloaded from Blackboard and extracted into some directory of your choice.

These libraries should all hopefully be easy to set up on your own machine. While there should be no problems using Windows or OS X, we only officially support Linux (and we'll be testing your submitted code on Linux), so we recommend using it.

[1]The GUI class is a modified version of the PyCX simulator found at `http://pycx.sourceforge.net/`

For Linux, the libraries should be available via the package manager. E.g. on Ubuntu one can execute:

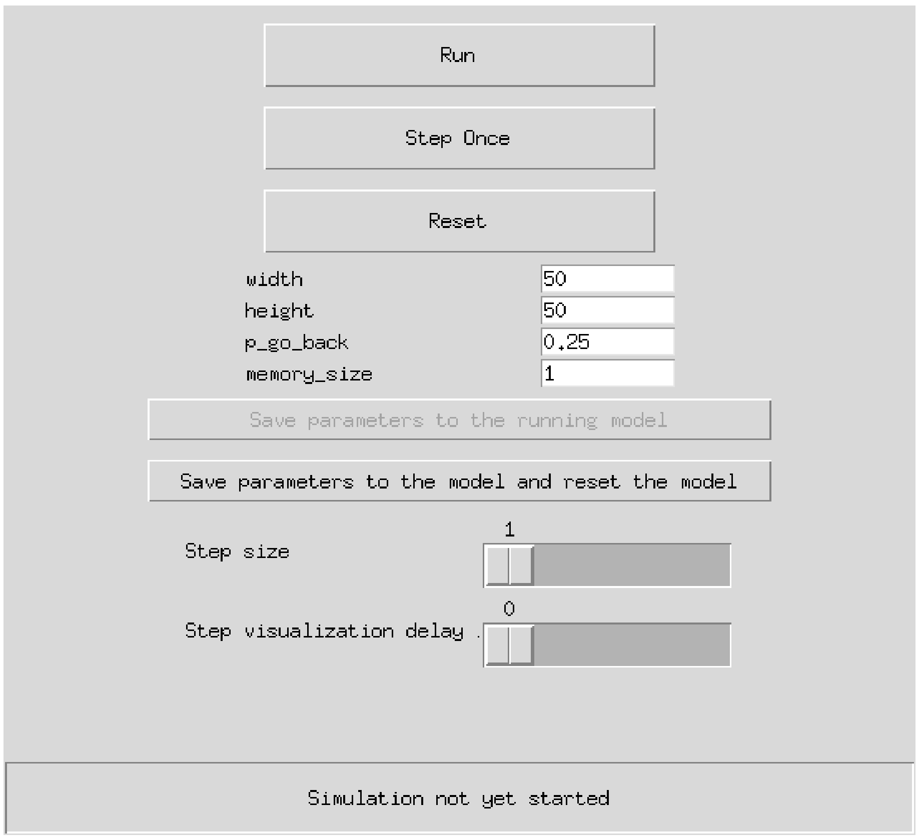
```
$ sudo apt-get install python-matplotlib python-numpy python-networkx
```

On OS X, the libraries can easily be installed using `pip`:

```
$ pip install <LIBRARY NAME>
```

If you experience a laggy PyICS GUI you need to upgrade your Python/Tkinter version (or use Linux e.g. on a virtual machine).

Finally, on Windows you can either download the installers from their websites manually, or install pythonxy: `https://code.google.com/p/pythonxy/`.


Run
Step Once
Reset
Save parameters to the running model
Save parameters to the model and reset the model

```
width          50
height         50
p_go_back      0.25
memory_size    1
Step size      1
Step visualization delay   0
Simulation not yet started
```

Figure 1: The GUI of the interactive simulator provided with the framework.

The PyICS framework

The framework available on Blackboard makes it easier to create and test models. A **model** is a class which must contain at least three methods: `reset`, `step` and `draw`. The **reset** method is called every time the simulation should be (re)initialized. In the case of CA, this would include resetting the state of the grid to an empty one. The **step** method should simulate a single step of the simulation. This method can also detect certain end-conditions for the simulation, and can tell the caller to stop further simulation by returning the value `True`. Finally, the **draw** method informs the simulation to draw its current state (e.g. with matplotlib). The `step` and `draw` methods are separate for two reasons: In some cases the drawing might take some time, so it may be more logical to only draw the state every 500 steps or so. In some cases you might not want a visualization at all, as we will see later.

To look at your model in an interactive way, a **GUI** is provided. All model files in the framework are set up in such a way that if the file is executed directly (and not imported from another file), it will start this GUI. The interface, as shown in Fig. 1, contains three buttons to start/stop the simulation, and to manually execute the step method a single time. The reset button will call the `reset` method.

By default, the GUI will call the `draw` method after every step, although this behavior can be changed via the slides on the bottom labeled "Step size". Setting this to a value of 100 means after every 100 steps the current state is drawn. The other slides labeled "Step visualization delay" controls the pause between each call to the step function, which allows you to more closely observe every frame.

Finally, the interface allows the modification of several **parameters** without restarting the application or modifying the code. Parameters are defined in the constructor of the model, which in Python is called `__init__`. Parameters are defined and used in the code as follows:

```
class MyModel(Model):
    def __init__(self):
        Model.__init__(self)  # Call parent constructor

        self.make_param('foo', 10)  # Registers foo accessible via 'self.foo'

        # For example:
        print self.foo  # Prints 10
```

After the call to `make_param` the parameter is accessible as any other variable, and will automatically show up in the GUI. All parameters should have a default value, which will also determine the type by default. For more details about the parameters, refer to the documentation in model.py.