Chapter 10

Interactive Simulation of Complex Systems

10.1 Simulation of Systems with a Large Number of Variables

We are finally moving into the modeling and analysis of *complex systems.* The number of variables involved in a model will jump drastically from just a few to tens of thousands! What happens if you have so many dynamical components, and moreover, if those components interact with each other in nontrivial ways? This is the core question of complex systems. Key concepts of complex systems, such as emergence and self-organization, all stem from the fact that a system is made of a massive amount of interactive components, which allows us to study its properties at various scales and how those properties are linked across scales.

Modeling and simulating systems made of a large number of variables pose some practical challenges. First, we need to know how to specify the dynamical states of so many variables and their interaction pathways, and how those interactions affect the states of the variables over time. If you have empirical data for all of these aspects, lucky you—you could just use them to build a fairly detailed model (which might not be so useful without proper abstraction, by the way). However, such detailed information may not be readily available, and if that is the case, you have to come up with some reasonable assumptions to make your modeling effort feasible. The modeling frameworks we will discuss in the following chapters (cellular automata, continuous field models, network models, and agent-based models) are, in some sense, the fruit that came out of researchers' collective effort to come up with "best practices" in modeling complex systems, especially

with the lack of detailed information available (at least at the time when those frameworks were developed). It is therefore important for you to know explicit/implicit model assumptions and limitations of each modeling framework and how you can go beyond them to develop your own modeling framework in both critical and creative ways.

Another practical challenge in complex systems modeling and simulation is visualization of the simulation results. For systems made of a few variables, there are straightforward ways to visualize their dynamical behaviors, such as simple time series plots, phase space plots, cobweb plots, etc., which we discussed in the earlier chapters. When the number of variables is far greater, however, the same approaches won't work. You can't discern thousands of time series plots, or you can't draw a phase space of one thousand dimensions. A typical way to address this difficulty is to define and use a metric of some global characteristics of the system, such as the average state of the system, and then plot its behavior. This is a reasonable approach by all means, but it loses a lot of information about the system's actual state.

An alternative approach is to visualize the system's state at each time point in detail, and then *animate* it over time, so that you can see the behavior of the system without losing information about the details of its states. This approach is particularly effective if the simulation is *interactive*, i.e., if the simulation results are visualized on the fly as you operate the simulator. In fact, most complex systems simulation tools (e.g., NetLogo, Repast) adopt such *interactive simulation* as their default mode of operation. It is a great way to explore the system's behaviors and become "experienced" with various dynamics of complex systems.

10.2 Interactive Simulation with PyCX

We can build an interactive, dynamic simulation model in Python relatively easily, using *PyCX*'s "pycxsimulator.py" file, which is available from `http://sourceforge.net/projects/pycx/files/` (it is also directly linked from the file name above if you are reading this electronically). This file implements a simple interactive graphical user interface (GUI) for your own dynamic simulation model, which is still structured in the three essential components—initialization, observation, and updating—just like we did in the earlier chapters.

To use pycxsimulator.py, you need to place that file in the directory where your simulation code is located. Your simulation code should be structured as follows:

Code 10.1: interactive-template.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *

# import necessary modules
# define model parameters

def initialize():
    global # list global variables
    # initialize system states

def observe():
    global # list global variables
    cla() # to clear the visualization space
    # visualize system states

def update():
    global # list global variables
    # update system states for one discrete time step

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

The first three lines and the last two lines should always be in place; no modification is needed.

Let's work on some simple example to learn how to use pycxsimulator.py. Here we build a model of a bunch of particles that are moving around randomly in a two-dimensional space. We will go through the process of implementing this simulation model step by step.

First, we need to import the necessary modules and define the parameters. In this particular example, a Gaussian random number generator is useful to simulate the random motion of the particles. It is available in the `random` module of Python. There are various parameters that are conceivable for this example. As an example, let's consider the number of particles and the standard deviation of Gaussian noise used for random motion of particles, as model parameters. We write these in the beginning of the code as follows:

Code 10.2:

```
import random as rd
n = 1000 # number of particles
sd = 0.1 # standard deviation of Gaussian noise
```

Next is the initialization function. In this model, the variables needed to describe the state of the system are the positions of particles. We can store their x- and y-coordinates in two separate lists, as follows:

Code 10.3:

```
def initialize():
    global xlist, ylist
    xlist = []
    ylist = []
    for i in xrange(n):
        xlist.append(rd.gauss(0, 1))
        ylist.append(rd.gauss(0, 1))
```

Here we generate `n` particles whose initial positions are sampled from a Gaussian distribution with mean 0 and standard deviation 1.

Then visualization. This is fairly easy. We can simply plot the positions of the particles as a scatter plot. Here is an example:

Code 10.4:

```
def observe():
    global xlist, ylist
    cla()
    plot(xlist, ylist, '.')
```

The last option, `'.'`, in the `plot` function specifies that it draws a scatter plot instead of a curve.

Finally, we need to implement the updating function that simulates the random motion of the particles. There is no interaction between the particles in this particular simulation model, so we can directly update `xlist` and `ylist` without using things like `nextxlist` or `nextylist`:

Code 10.5:

```
def update():
    global xlist, ylist
```

```
    for i in xrange(n):
        xlist[i] += rd.gauss(0, sd)
        ylist[i] += rd.gauss(0, sd)
```

Here, a small Gaussian noise with mean 0 and standard deviation `sd` is added to the x- and y-coordinates of each particle in every time step.

Combining these components, the completed code looks like this:

Code 10.6: random-walk-2D.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *

import random as rd
n = 1000 # number of particles
sd = 0.1 # standard deviation of Gaussian noise

def initialize():
    global xlist, ylist
    xlist = []
    ylist = []
    for i in xrange(n):
        xlist.append(rd.gauss(0, 1))
        ylist.append(rd.gauss(0, 1))

def observe():
    global xlist, ylist
    cla()
    plot(xlist, ylist, '.')

def update():
    global xlist, ylist
    for i in xrange(n):
        xlist[i] += rd.gauss(0, sd)
        ylist[i] += rd.gauss(0, sd)

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

Run this code, and you will find two windows popping up (Fig. 10.1; they may appear overlapped or they may be hidden under other windows on your desktop)[1][2].

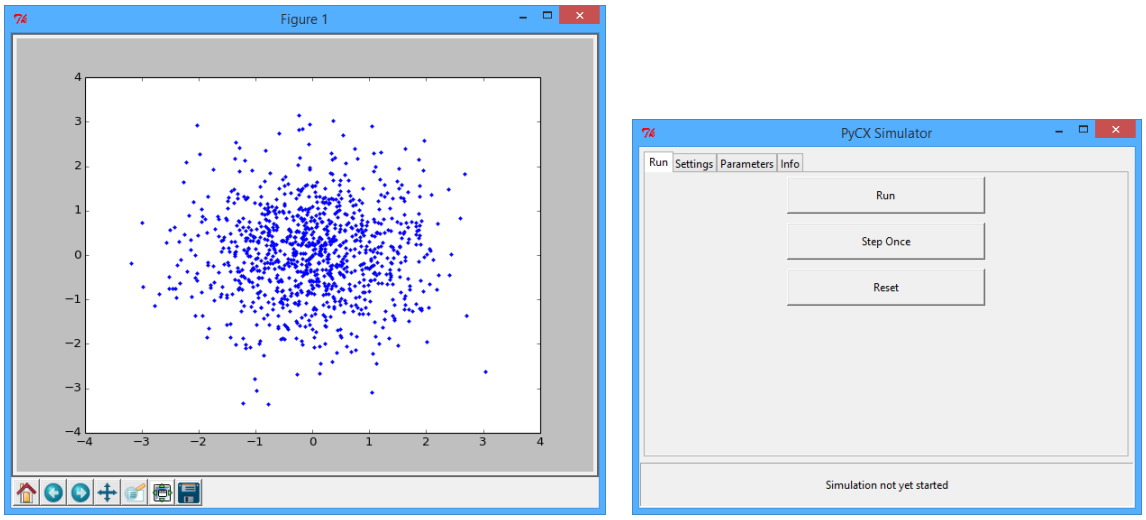

Figure 10.1: Graphical user interface (GUI) generated by pycxsimulator.py. Left: Visualization window. Right: Control window.

This interface is very minimalistic compared to other software tools, but you can still do basic interactive operations. Under the "Run" tab of the control window (which shows up by default), there are three self-explanatory buttons to run/pause the simulation, update the system just for one step, and reset the system's state. When you run the simulation, the system's state will be updated dynamically and continuously in the other visualization window. To close the simulator, close the control window.

[1] If you are using Anaconda Spyder, make sure you run the code in a plain Python console (not an IPython console). You can open a plain Python console from the "Consoles" menu.

[2] If you are using Enthought Canopy and can't run the simulation, try the following:

1. Go to "Edit" → "Preferences" → "Python" tab in Enthought Canopy.
2. Uncheck the "Use PyLab" check box, and click "OK."
3. Choose "Run" → "Restart kernel."
4. Run your code. If it still doesn't work, re-check the "Use PyLab" check box, and try again.

Under the “Settings” tab of the control window, you can change how many times the system is updated before each visualization (default = 1) and the length of waiting time between the visualizations (default = 0 ms). The other two tabs (“Parameters” and “Info”) are initially blank. You can include information to be displayed under the “Info” tab as a “docstring” (string placed as a first statement) in the `initialize` function. For example:

Code 10.7:

```
def initialize():
    '''
This is my first PyCX simulator code.
It simulates random motion of n particles.
Copyright 2014 John Doe
    '''
    global xlist, ylist
    ...
```

This additional docstring appears under the “Info” tab as shown in Fig. 10.2.

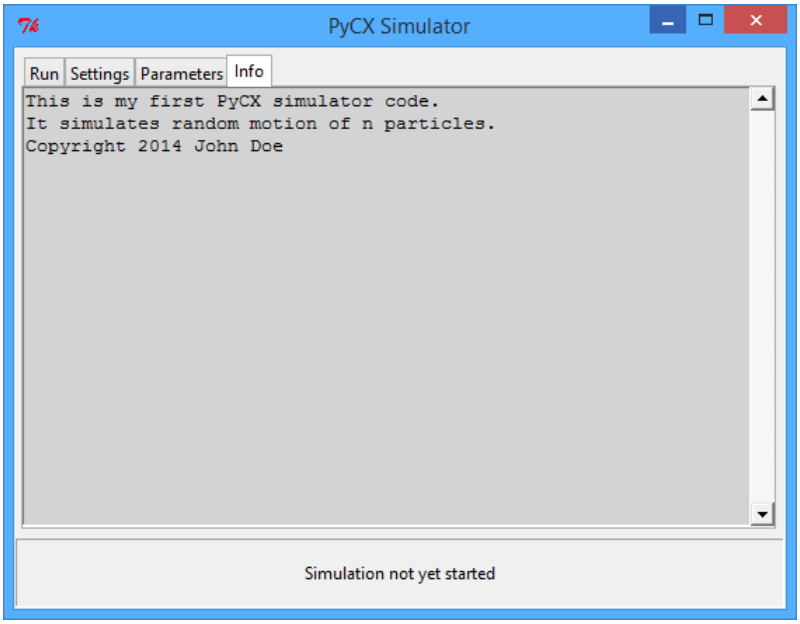

Figure 10.2: Information about the simulation added as a “docstring” in the initialization function.

Exercise 10.1 The animation generated by Code 10.6 frequently readjusts the plot range so that the result is rather hard to watch. Also, the aspect ratio of the plot is not 1, which makes it hard to see the actual shape of the particles' distribution. Search matplotlib's online documentation (`http://matplotlib.org/`) to find out how to fix these problems.

10.3 Interactive Parameter Control in PyCX

In Code 10.6, the parameter values are all directly given in the code and are not changeable from the control window. PyCX has a feature, however, by which you can create interactive parameter setters (thanks to Przemysław Szufel and Bogumił Kamiński at the Warsaw School of Economics who developed this nice feature!). A parameter setter should be defined as a function in the following format:

Code 10.8:

```
def <parameter setter name> (val = <parameter name>):
    '''
    <explanation of parameter>
    <this part will be displayed when you mouse-over on parameter setter>
    '''
    global <parameter name>
    <parameter name> = int(val) # or float(val), str(val), etc.
    return val
```

This may look a bit confusing, but all you need to do is to fill in the <. . .> parts. Note that the `int` function in the code above may need to be changed according to the type of the parameter (`float` for real-valued parameters, `str` for string-valued ones, etc.). Once you define your own parameter setter functions, you can include them as an option when you call the `pycxsimulator.GUI()` function as follows:

Code 10.9:

```
pycxsimulator.GUI(parameterSetters=<list of parameter setters>).start...
```

Here is an example of a parameter setter implementation that allows you to interactively change the number of particles:

Code 10.10: random-walk-2D-pSetter.py

```
def num_particles (val = n):
    '''
    Number of particles.
    Make sure you change this parameter while the simulation is not running,
    and reset the simulation before running it. Otherwise it causes an error!
    '''
    global n
    n = int(val)
    return val

import pycxsimulator
pycxsimulator.GUI(parameterSetters = [num_particles]).start(func=[initialize
, observe, update])
```

Once you apply this modification to Code 10.6 and run it, the new parameter setter appears under the "Parameters" tab of the control window (Fig. 10.3). You can enter a new value into the input box, and then click either "Save parameters to the running model" or "Save parameters to the model and reset the model," and the new parameter value is reflected in the model immediately.

> *Exercise 10.2* To the code developed above, add one more parameter setter for `sd` (standard deviation of Gaussian noise for random motion of particles).

10.4 Simulation without PyCX

Finally, I would like to emphasize an important fact: The PyCX simulator file used in this chapter was used only for creating a GUI, while the core simulation model was still fully implemented in your own code. This means that your simulation model is completely independent of PyCX, and once the interactive exploration and model verification is over, your simulator can "graduate" from PyCX and run on its own.

For example, here is a revised version of Code 10.6, which automatically generates a series of image files *without using PyCX at all.* You can generate an animated movie from the saved image files using, e.g., Windows Movie Maker. This is a nice example that illustrates the main purpose of PyCX—to serve as a stepping stone for students and

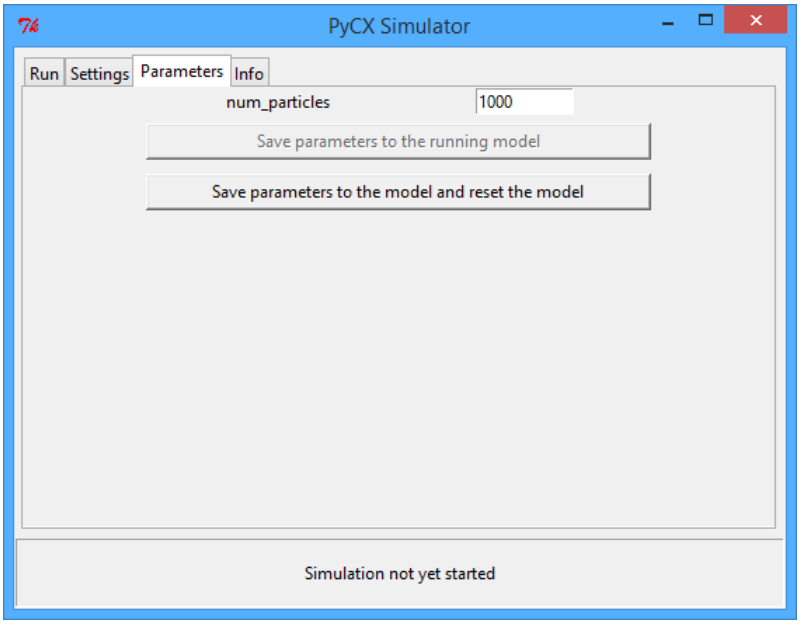

Figure 10.3: New parameter setter for the number of particles implemented in Code 10.10.

researchers in learning complex systems modeling and simulation, so that it eventually becomes unnecessary once they have acquired sufficient programming skills.

Code 10.11: random-walk-2D-standalone.py

```
from pylab import *

import random as rd
n = 1000 # number of particles
sd = 0.1 # standard deviation of Gaussian noise

def initialize():
    global xlist, ylist
    xlist = []
    ylist = []
    for i in xrange(n):
        xlist.append(rd.gauss(0, 1))
        ylist.append(rd.gauss(0, 1))

def observe():
    global xlist, ylist
    cla()
    plot(xlist, ylist, '.')
    savefig(str(t) + '.png')

def update():
    global xlist, ylist
    for i in xrange(n):
        xlist[i] += rd.gauss(0, sd)
        ylist[i] += rd.gauss(0, sd)

t = 0
initialize()
observe()
for t in xrange(1, 100):
    update()
    observe()
```

Exercise 10.3 Develop your own interactive simulation code. Explore various features of the plotting functions and the PyCX simulator's GUI. Then revise your code so that it automatically generates a series of image files without using PyCX. Finally, create an animated movie file using the image files generated by your own code. Enjoy!