Intro to PyCX

CMPLXSYS 530 - Marisa Eisenberg

1/23/2020

What is PyCX?

- Developed by Hiroki Sayama
- A python library that provides a convenient way to visualize ABMs, cellular automata, etc.
  - Includes a GUI
  - 'Info' tab
  - Easy to add interactive parameter control (parameter sliders, etc.)
- Also includes a wide range of example scripts
  - Classic models of many kinds (not just ABMs—ODEs, networks, etc.)
  - These can be very useful as starting points for building your own models!

How to get PyCX

- Download from the PyCX GitHub: https://github.com/hsayama/PyCX
- Compatible with Python 2.7 or 3
- Several packages we will often want to use (be sure these are installed):
    - Numpy, scipy, matplotlib, random, math, and networkx

3/17

Using PyCX

- To use PyCX, make sure you put the file `pycxsimulator.py` in the directory where you have your model code (or wherever your working directory is)

- Try out the package: open and run `abm-segregation-discrete.py` to run the Schelling Model

PyCX Example Models

- The file names of sample codes use the following prefixes:
    - "ds-": for low-dimensional dynamical systems
    - "dynamic-": for demonstration of how to use pycxsimulator.py
    - "ca-": for cellular automata
    - "pde-": for partial differential equations
    - "net-": for network models
    - "abm-": for agent-based models

PyCX model template

Start by loading needed packages & defining model parameters

```
import pycxsimulator
from pylab import *  # imports numpy and pyplot

# import necessary modules
# define model parameters
```

PyCX model template

Next, build three functions we will need

```
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
```

PyCX model template

Lastly, run the model!

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

- Note that PyCX is very agnostic about how you code the model—it really just provides a nice simulation and visualization GUI

Let's build a simple model!

- Let's implement the voting model we built in the Emoji-simulator

- Simple voting model
  - 100 x 100 grid full of agents
    - Wrap the grid so that edge agents have neighbors on the opposite side
  - Each agent starts with an initial planned vote of "yes" or "no" (0 = no, 1 = yes)
    - Set each agent's initial vote with a 0.5 probability of yes
  - Each agent will change vote if more than half of queen-type neighbors vote the other way

Voting model in PyCX

Start by loading PyCX and setting parameters

```
import pycxsimulator
from pylab import *

n = 100 # size of space: n x n
p = 0.5 # initial agent probability of voting yes
```

Voting model in PyCX

Initialize the model

```
def initialize():
    # Things we need to access from different functions go here (discuss globals)
    global config, nextconfig

    # Build our grid of agents - fill with zeros for now
    config = zeros([n, n])

    # Set them to vote yes with probability p
    for y in range(n):
        for x in range(n):
            if random() < p: config[x, y] = 1

    # Set the next timestep's grid to zeros for now (we'll update in the update function)
    nextconfig = zeros([n, n])
```

Voting model in PyCX

Update the model at each time step

```
def update():
    global config, nextconfig

    # Go through each cell and check if they should change their vote in the next step
    for x in range(n):
        for y in range(n):
            count = 0  # variable to keep track of how many neighbors are voting yes

            for dx in [-1, 0, 1]:        # check the cell before/middle/after
                for dy in [-1, 0, 1]:    # check above/middle/below
                # discuss nesting for loops vs. not---what does this change?

                    # Add to count if neighbor is voting yes (note you also count yourself!)
                    count += config[(x + dx) % n, (y + dy) % n] # discuss
```

12/17

Voting model in PyCX

Update function continued

(This code goes outside the `for dx` and `for dy` loops but inside the `for x` and `for y` loops)

```
        # Now that we know how many neighbors are voting yes, decide what to do
        if config[x,y] == 0: # if this agent was going to vote no
            nextconfig[x, y] = 1 if count > 4 else 0
            # note we only change the vote for nextconfig, not config!

        else: # otherwise agent was going to vote yes (could also do elif)
            nextconfig[x, y] = 0 if (8 - (count-1)) > 4 else 1
            # note we reduced count by 1 since count included self

    # advance config forward one step and reset nextconfig
    config, nextconfig = nextconfig, zeros([n, n])
    # Can also be a little more efficient and do config, nextconfig = nextconfig, config
```

Voting model in PyCX

Observe function

```
def observe():
    global config, nextconfig
    cla() # clear visualization
    imshow(config, vmin = 0, vmax = 1, cmap = cm.binary) # display grid!
```

And let's run it!

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

Info

You can add text to the Info tab of the GUI by adding a comment to the initialize function:

```
def initialize():
'''
Information about my model goes here.
This is a voting model that does some neat stuff.
Copyright 2020 CSCS 530
'''
global # etc
```

15/17

Interactive parameters

- You can also add interactive parameters to the GUI by writing a "parameter setter" function
- For example, let's make one for the initial probability of voting yes:

```
def setvoteprob (val = p):
    '''
    Parameter info---this will be displayed when you mouse-over on parameter setter
    '''
    global p
    p = float(val) # or int(val), str(val), etc.
    return val
```

- Then, we pass this parameter setter to the `pycxsimulator.GUI()` when we run our model:

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

Simulating without PyCX

- One nice feature of the PyCX setup is that you can move away from it relatively easily when you want to do more complicated analyses
- Try running `initialize()` and then running `update()` a few times without using PyCX
- You can design your own visualizations, how to store results, etc., and then just run a loop over your timesteps
- The PyCX framework encourages organized functions for model setup, running, etc., but then you can move to your own system for final visualization and analysis