Simulation – Experiments and Trials

15-110 – Monday 04/18

Announcements

- Check6-1 grades released
  - **Make sure to view your feedback on the programming part especially!** You'll want to fix errors now so that they don't impact your work on Check6-2 and Hw6
- Check6-1 Revisions on Tue 4/22
- Check 6-2 due Tue 4/22

Learning Goals

- Update a model after **events** (mouse-based and keyboard-based) based on **rules**

- Use **Monte Carlo methods** to estimate the answer to a question

Interaction Events

Interaction Events

In the previous Simulation lecture, we learned about how to use controllers that change a model over **time**. The second kind of controller is one that captures **events**.

An event represents a single user interaction with the computer system. Events come in many forms: **keyboard presses**, **mouse clicks**, touchpad gestures, button presses, touchscreen presses, etc...

When you take an action on your computer, a **signal** is sent from the computer hardware to any programs that are currently running. That signal has information about the type of the event (key press vs. mouse click), plus any additional information that might be useful (which key was pressed).

Sidebar: Controller Functions – Event Loop

The event controller runs an **event loop** to capture the signals that the computer sends out, similar to the time loop discussed before. However, events occur **irregularly,** unlike regularly-timed rules.

To implement this event loop, we'll have our simulation system constantly **listen** for events. When an event occurs, the controller will catch it and send the event data on to the correct rule function; then it will tell the view to update. This is done with a special kind of Tkinter function called `bind` and is provided in the starter code.

With Tkinter we can listen for and bind functions to lots of different event types. We'll care about just two: `<Key>`, a key press, and `<Button-1>`, a left mouse click. There are lots of other Tkinter events we can implement if we want them:

https://web.archive.org/web/20190512164300id_/http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/event-types.html

Event Rules

To deal with Key and Mouse events, we'll introduce two new rule functions to our simulation framework:

- `keyPressed(data, event)`
- `mousePressed(data, event)`

Each of these takes `data` (our components data structure) and `event`, an **event object** that contains the information about the event.

These work like `runRules(data, call)` – we update `data`, then the controller refreshes the view immediately afterwards. This lets us make visible changes to the model.

`keyPressed` Events

In `keyPressed`, the `event` parameter contains two values we can access with a `.` (like string or list methods):

- `event.char` is a string that holds the character pressed
- `event.keysym` is a string that holds the 'name' of the character, for characters we can't show in a string (e.g., Enter or BackSpace)

If we want to draw the last-pressed character in the middle of the screen, for example, we would store that character in `data`, then draw it in `makeView`:

```
def keyPressed(data, event):
    data["text"] = event.char
```

Example Key Event

```
def makeModel(data):
    data["color"] = "red"
    data["tmp"] = "" # need to hold partial strings

def makeView(data, canvas):
    canvas.create_oval(200 - 50, 200 - 50, 200 + 50, 200 + 50,
                       fill=data["color"])

def keyPressed(data, event):
     # build up a color string one char at a time until user presses Return
     if event.keysym != "Return":
        data["tmp"] += event.char
     else:
        # move the color into data["color"]
        data["color"] = data["tmp"]
        data["tmp"] = ""
```

`mousePressed` Events

In `mousePressed`, the `event` parameter holds the pixel location where the user clicked on the canvas.

- `event.x` is the x location
- `event.y` is the y location

If we want to move a circle around the canvas to be centered wherever you click, we'd need to store the center location and draw the circle based on the model location in `makeView`:

```
def mousePressed(data, event):
    data["cx"] = event.x
    data["cy"] = event.y
```

Example Mouse Event

```
def makeModel(data):
    data["color"] = "red"

def makeView(data, canvas):
    canvas.create_oval(200 - 50, 200 - 50, 200 + 50, 200 + 50,
                       fill=data["color"])

def mousePressed(data, event):
    import random
    newColor = random.choice(["red", "orange", "yellow",
                              "green", "blue", "purple"])
    # Check if the user clicked inside the circle
    # Is the distance between the center and the click less than the radius?
    if ((event.x - 200)**2 + (event.y - 200)**2)**0.5 <= 50:
        data["color"] = newColor
```

Randomness

Random Functions

Most simulations use randomness in some way; otherwise, every run of the simulation will produce the same result.

Recall the random library, which we learned about early in the semester. This module included several useful functions we can use:

```
random.random() # pick a random float between 0-1

random.randint(x, y) # pick a random number in a range

random.choice(lst) # chooses an element randomly
random.shuffle(lst) # destructively shuffles the list
```

Computing Randomness

How is it possible for us to generate random numbers this way?

Randomness is difficult to define, either philosophically or mathematically. Here is a practical definition: given a truly random sequence, there is **no gambling strategy possible** that allows a winner in the long run.

But computers are **deterministic** – given an input, a function should always return the same output. Circuits should not behave differently at different points in time. So how does the random library work?

True Randomness

To implement truly random behavior, we can't use an algorithm. Instead, we must gather data from **physical phenomena** that can't be predicted.

Common examples are atmospheric noise, radioactive decay, or thermal noise from a transistor.

This kind of data is impossible to predict, but it's also slow and expensive to measure.

Pseudo-Randomness

Most programs instead use **pseudo-random numbers** for casual purposes. A **pseudo-random number generator** is an algorithm that produces numbers which look 'random enough'. Each number the algorithm generates acts as a starting place to generate the next one.

By calling the function repeatedly, the algorithm generates a **sequence** of numbers that appear to be random to the casual observer.

The number sequence generated by a pseudo-random number generator isn't *truly* random; if someone figures out the algorithm, they can predict the results. But it is random enough to use for casual purposes.

Monte Carlo Methods

Randomness in Simulation

Using randomness in a simulation means that the same simulation might have multiple different outcomes on the same input model. A single run of a simulation is not a good estimate of the true average outcome.

To find the truth in the randomness, we need to use probability!

Law of Large Numbers

The Law of Large Numbers states that if you perform an experiment multiple times, the average of the results will approach the **expected value** as the number of trials grows.

This law works for simulation as well! We can calculate the expected value of an event by simulating it a large number of times.

We call programs that repeat simulations this way **Monte Carlo methods**, after the famous gambling district in the French Riviera.

Monte Carlo Method Structure

If we put our simulation code in the function `runTrial()` and want to find the odds that a simulation 'succeeds', a Monte Carlo method might take the following format:

```
def getExpectedValue(numTrials):
    count = 0
    for trial in range(numTrials):
        result = runTrial() # run a new simulation
        if result == True:  # check the result
            count = count + 1
    return count / numTrials # return the probability
```

Monte Carlo Example

Every year, SCS holds the Random Distance Race. The length of this race is determined by rolling two dice. **What is the expected number of laps a runner will need to complete?**

```
import random
def runTrial():
    return random.randint(1, 6) + random.randint(1, 6)

def getExpectedValue(numTrials):
    lapCount = 0
    for trial in range(numTrials):
        lapCount += runTrial()
    return lapCount / numTrials
```

Activity: Monte Carlo Methods

You do: what are the odds that a runner in the Random Distance Race will need to run 10 or more laps?

Write the code to run the trial. You can modify the code from the previous slide.

Testing Simulations

Using Simulations

Once we've programmed a robust simulation, we can **change the starting state** to see how it changes the simulation. This is especially useful when we want to **predict** certain things about the world.

We can check predictions more quickly by making `timeRate` smaller (calling the simulation more often).

We've included on the course website a pre-written simulation that models a zombie apocalypse. Let's use this as an example of how to make predictions with simulations.

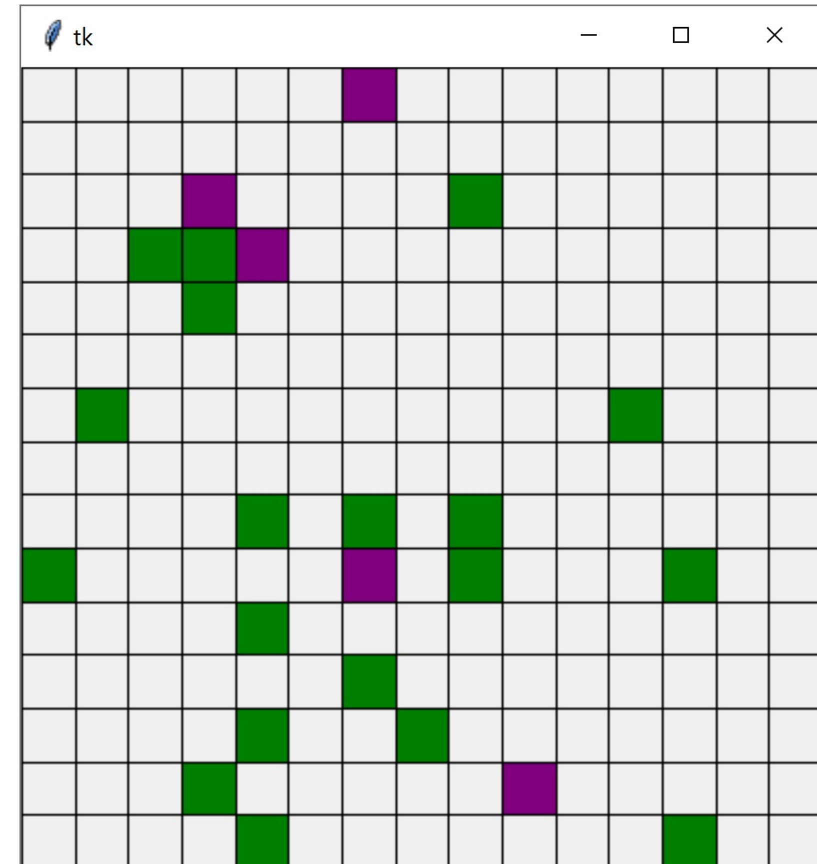
Zombie Simulation

This simulation models the world as a grid. Each cell of the grid can be empty, or can have a human (green) or zombie (purple) on it.

At every time step, the zombies move in a random direction while the humans stay still (they're hiding). If a zombie is bordering a human, there is an infection rate (a probability) for whether the human will turn into a zombie or not. The simulation is done when all entities are zombies.

Here are a few questions we can ask: how long will it take for the whole world to become zombies...

- In our current code?
- If we start with more or fewer humans?
- If we start with a higher infection rate?



Calculating Outcomes

If we want to explore the simulation, we can run it with the visualization on.

If we just want to find the **average results**, we can call the `makeModel` and `runRules` functions from a new function where the time loop becomes a while loop. Have that function return the number of days it takes to zombify all the humans.

When we run this function with `getExpectedValues` we find the expected amount of time left for the human race. Monte Carlo solves the problem!

Calculating Outcomes Code

```
def runTrial():
    data = { }
    makeModel(data) # initial setup
    daysPassed = 0
    while not allZombies(data["creatures"]): # while loop instead of time loop
        runRules(data, daysPassed)
        daysPassed += 1
    return daysPassed

def getExpectedValue(numTrials):
    dayCount = 0
    for trial in range(numTrials):
        dayCount += runTrial()
    return dayCount / numTrials

print(getExpectedValue(100))
```

Learning Goals

- Update a model after **events** (mouse-based and keyboard-based) based on **rules**

- Use **Monte Carlo methods** to estimate the answer to a question

Feedback: https://bit.ly/110-s22-feedback