Simulation - Modeling a Bike Share System

September 21, 2023

Contents

1 README

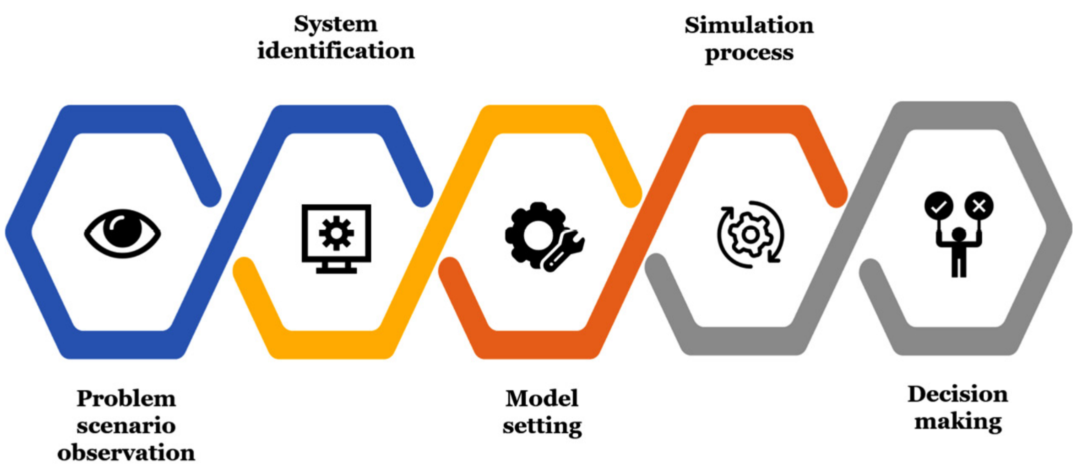
- Decision making workflow model
- Differences between modeling and simulation
- Simulation modeling terminology
- Simple example: bike sharing simulation

- Python support for simulations
- Based on Ciaburrio (2022), and Downey (2023).

2 Decision making workflow model

In decision-making processes, the starting point is identifying the problem that requires a change and therefore a decision.

A formal model is built that allows the simulation of the system to understand its behavior and identify decisions to be made.



3 Comparing modeling and simulation

A model is a representation of a physical system. Modeling is a design process.

Simulation is the process of seeing how a model-based system would work under certain conditions. Simulation is an operative process.

4 Pros and cons of simulation modeling

Pros:

- System behavior that cannot be directly experienced is reproduced
- Real complex systems are represented with sources of uncertainty

- Limited data resources are required
- Experimentation in limited time is possible
- Resulting models are easily interpretable

Cons:

- Simulation provides indications but not exact results
- Analysis of the output can be complex
- Implementation of a simulation model can be laborious
- Results that are returned depend on the input data quality
- Simulation complexity depends on system complexity

5 Simulation modeling terminology

1. System: a set of interacting elements with a boundary.
2. State variables: e.g. for weather, the temperature.
3. Events: causes state variables to change instantaneously.
4. Parameters: adjustable values that determine model behavior.
5. Calibration: adjusting paramters to obtain maximum accuracy.
6. Accuracy: degree of correspondence of simulation and actual data.
7. Sensitivity: degree to which model outputs are affected by input.
8. Validation: verification of the accuracy of the model.

6 Exiting gracefully from an error

You can make Python exit gracefully if you prepare for well-known errors, e.g. an `ImportError` if a library is not installed. The code below will only work in IPython - otherwise you need a

```
# Install pint if necessary
try:
    import pint
except ImportError:
    !pip install pint
```

7 Download and install `modsim`

The `download` function loads a Python file from `url`. It also will exit gracefully if there is no file at the URL. We then call the function on the location of `modsim` and import its functions:

```
# download modsim.py if necessary
from os.path import basename, exists

def download(url):
    filename = basename(url)
    if not exists(filename):
        from urllib.request import urlretrieve
        local, _ = urlretrieve(url, filename)
        print('Downloaded ' + local)

# call function for download
download('https://raw.githubusercontent.com/AllenDowney/' +
         'ModSimPy/master/modsim.py')

# import functions from modsim
from modsim import *
```

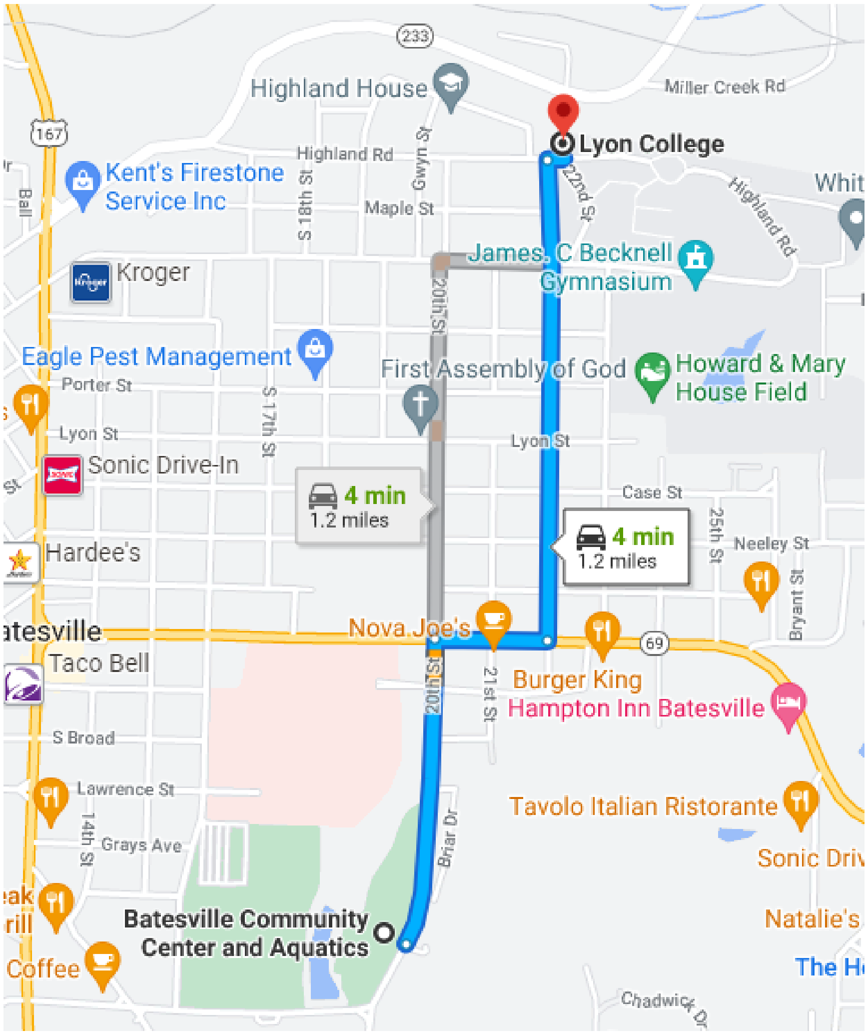
Modify this last command so that Python exits in case `modsim` is not installed:

```
# Install pint if necessary
try:
    import modsim
except ImportError:
    print("modsim is not installed.")
```

8 Bike sharing model

The simulation sandbox

Consider a bike sharing system for students traveling between two sites, LEAP on Lyon's campus, and the Batesville City Community Centre on 20th Street.



System description

The **system** contains 12 bikes as **elements** and 2 bike racks each with a capacity to hold 12 bikes.

State changes in either location are caused by students checking out bikes at one and riding to the other location.

In the **simulation**, we keep track on where the bikes are using the `modsim.State` function:

1. import the `modsim` library

2. look at the `help` for `State`

```
import modsim
# help(State)
```

More about the `State` function

`State` is defined with `**variables` as argument, which means that any keyword arguments passed to that function will be collected into a dictionary called `variables`.

This means that you can initialize `State` with any number of keyword variables. For example, you could use the function to represent a simple bank account:

```
from modsim import State
bank_account = State(balance=100,interest_rate=0.05)
print(f'You have ${bank_account.balance} in your \
{int(bank_account.interest_rate*100)}% interest bank account.')
```

When you check the `type` of `modsim.State()`, you can see that it is based on a `pandas Series` object, or one-dimensional `numpy` array, or a vector (see doc):

```
print(type(bank_account))
```

You can also use `source_code` to see the code for the function:

```
modsim.source_code(modsim.State)
```

`Series` objects provide their own `plot` function, `Series.plot()`.

Let's look at this function `source_code`:

```
modsim.source_code(modsim.source_code)
```

Using `State` to describe the system

We store the state of the bike sharing system in a state `bikeshare`, with the number of bike in either location:

```
bikeshare = State(leap=10, city=2)
```

We can now get the value of the state variables `leap` and `city`:

```
print(f'Bikes at LEAP: {bikeshare.leap}')
print(f'Bikes at Community Center: {bikeshare.city}')
```

To see all state variables and their values, just enter the object's name (this is better formatted in IPython):

```
print(bikeshare)
```

Updating the state of the system

To update the system, we can either assign new values to the state variables, or we can use C-style update operators `+=` and `-=`:

```
bikeshare.leap = 9
bikeshare.city = 3
print(bikeshare)
```

We use the update operators to return the system to the previous state:

```
bikeshare.leap += 1
bikeshare.city -= 1
print(bikeshare)
```

The last line of the printout are `Series` metadata. To lose them but retain the tabular format, loop over the items:

```
print(f'     {bikeshare.name}')
for index, value in bikeshare.items():
    print(f'{index}     {value}')
```

The function `items` allows you to iterate over iterable tuples whose elements consist of an index and a value stored with that index:

```
print(modsim.source_code(bikeshare.items))
```

Defining functions

To be able to reuse code, we put it into functions. In Python, the template to create a function named `foo` (without arguments) that returns nothing looks like this:

```
def foo():
    # do something
    return #something
```

A simple example is this 'hello world' function:

```
# define function
def hello():
    print("Hello, world!")

# call function
hello()
```

As usual, functions are subroutines or encapsulated procedures: all variables inside the function are local, and if you want to return something to the calling routine, you need to add `return`.

The following function returns a string `msg`, which we can only access by saving the function result in a variable:

```
# define function with return value
def hello_again():
    msg = "Greeting complete."
    print("hello again")
    return msg

# call function and print return value
returned_msg = hello_again()
print(returned_msg)
try:
    print(msg)
except NameError:
    print("*** NameError: Cannot print local variable ***")
```

Add a statement at the very end to print `msg` itself. This will lead to a `NameError`. Fix this by wrapping `print(msg)` in a `try:...except NameError:` exception statement!

Finally, run the code through pythontutor.com to see what happens (solution).

Defining an updating function

Rather than repeat the update every time a bike moves, define a function that reflects a move of a bike from LEAP to the Community Center, `bike_to_city`:

```
def bike_to_city():
    bikeshare.leap -= 1
    bikeshare.city += 1
```

Now print the current state, then update it using the new function, then print the new state:

```
print(bikeshare)
bike_to_city()
print(bikeshare)
```

There's nothing that keeps our bike share state variables from going outside of the [0,12] range, which is a hard physical boundary. Let's fix this.

1. Write a function `reset` that restores a particular state, e.g. with 6 bikes in either location.
2. The function should print the old and the new state.
3. The function should announce itself "System reset".
4. Run `bike_to_city` a few times until the values are wrong.
5. Restore the steady state using your new function.

```
def reset():
    print("System reset. Old state:")
    print(bikeshare)
    bikeshare.leap = 6
    bikeshare.city = 6
    print("New state:")
    print(bikeshare)
```

Testing:

```
print(bikeshare)
bike_to_city()
bike_to_city()
bike_to_city()
bike_to_city()
bike_to_city()
reset()
```

Alter the `bike_to_city` function and print out "Moving bike to city" every time the function is called, test the function, and then move the system back to the steady state.

```
def bike_to_city():
    print("Moving bike to city.")
    bikeshare.leap -= 1
    bikeshare.city += 1

bike_to_city()
reset()
```

Pseudorandom number generator

As a simple model of customer behavior within the system, we use a *pseudo-random number generator* to determine when customers arrive at each bike station.

The function `modsim.flip` generates random coin tosses, i.e. it simulates tosses of a fair coin with default probability 0.5 for either side, and returns a Boolean value, `True` or `False`.

It is based on NumPy's `random` function:

```
from modsim import source_code,flip
print(source_code(flip))
```

The statement `np.random.random() < p` generates a Boolean value.

Call the function with a probability between 0 and 1, e.g. 70%. On average, it will return `True` with probability 70% or `False` with probability 30%:

```
for _ in range(10):
    print(flip(0.7),end=" ")
```

To control program behavior with Boolean values, we use conditional statements. The general form of such a statement is as follows:

```
if condition:
    # do something if condition is True
else:
    # do something else if condition is False
```

The following program simulates a fair coin: it prints "heads" if the `flip` results in `True`, and "tails" if it results in `False`.

```
if flip(0.5):
    print("heads")
else:
    print("tails")
```

For the particular argument 0.5 we could have left the argument out since `flip` is defined as `flip(p=0.5)` as we saw earlier, with `p=0.5` as the (named) default parameter.

Simulating customers as coin tosses

We can use `flip` to simulate the arrival of customers who want to borrow a bike: If customers arrive at the LEAP station every two minutes on average (that is with certainty, or 100%), then the chance of an arrival during any one-minute period is $100\%/2 = 50\%$:

```
if flip(0.5):
    bike_to_city()
```

If customers arrive at the Community Center station every three minutes on average, the chance of an arrival during any one-minute period is $100\%/3 = 33\%$:

```
if flip(0.33):
    bike_to_city()
```

Both of these snippets together with functions that change the state of the system can be used to simulate a time interval - in this case one minute:

```
def step():
    if flip(0.5):
        bike_to_city()
    if flip(0.33):
        bike_to_leap()
```

Depending on the random results from `flip`, a `step` moves a bike to the Community Centre or to the LEAP bike station, or neither, or both.

Before you can try it, you need to remember how to move a bike and create the function `bike_to_leap()` to move bikes back to LEAP:

```
def bike_to_leap():
    print("Moving bike to LEAP.")
    bikeshare.leap += 1
    bikeshare.city -= 1
```

Simulating customers and bikes:

```
step()
```

In reality, we'd need a smarter `reset()` function that is responsive to the fact that we only have 12 bikes and moves bikes automatically once the supply runs out at either end.

Adding simulation parameters

The previous version of `step` is fine if the arrival probabilities never change but in reality they vary over time.

To account for that, we can exchange the constant values by *parameters*:

```
def step(p1, p2):
    if flip(p1):
        bike_to_city()
    if flip(p2):
        bike_to_leap()
```

Now call the function with the previous values `p1=0.5` and `p2=0.33` as arguments:

```
step(0.5,0.33)
```

The parameters can be named or unnamed - if they're not named, you pass *positional* arguments relying on Python to know where to put them, but if you name them then you can decide the order. This is something to try in pythontutor.com (example).

You can now run the same function many times with different parameters each time - e.g. to distinguish different times of day.

Looping

To repeat a chunk of code, use a `for` loop:

```
for _ in range(3):
    print(_, end=": ")
    bike_to_city()
```

Here, `range` is used to control the number of times the loop runs, and `_` is a convenient choice for a dummy loop variable that serves no other purpose than counting.

`TimeSeries`

The `modsim` library provides a `TimeSeries` object to save results for later analysis: a `TimeSeries` is an event log. It contains a sequence of timestamps (labels) and their corresponding quantities (values).

1. Start a new `State` object (a system state).

2. Create a new, empty `TimeSeries` as `results`.
3. Print `results` (still empty).
4. Add a quantity (for example `bikeshare.leap`) to `results`.
5. Print `results` again (now containing a labelled quantity)

```
bikeshare = State(leap=10,city=2)
results = TimeSeries()
print(results)
results[0] = bikeshare.leap
print(results)
```

You can use `TimeSeries` in a loop to store the simulation results:

```
for i in range(3):
    print(i)
    step(0.6,0.6)
    results[i+1] = bikeshare.leap
```

We can display the `TimeSeries` with the `modsim.show` command:

```
from modsim import show
print(show(results))
```

Recap: What does `0.6` mean again in terms of customer arrivals?

Plotting simulation results

The `Series` objects in `pandas` provide a function called `plot` that we can use to plot the results. `modsim` provides `decorate` to minimally customize the plot with axis labels and title:

```
from modsim import decorate
results.plot()

decorate(title='LEAP-Community Center bikeshare',
         xlabel='Time step (min)',
         ylabel='Number of bikes [0,12]')
```

For three iterations, this is not very interesting - therefore next stop: iterative modeling!

9 Summary

- A system is constituted by elements and their relationships surrounded by a boundary.
- A simulation is the process of iterating over the State variables of a model-based system.
- Tools to develop and run a simulation include: State variables, update functions, random number generator and plots for visualization.
- Python tools include loops to repeat code, functions to write reusable procedures, and conditional statements to control flow.

10 References

Downey AB. Modeling and Simulation in Python. NoStarch Press; 2023. allendowney.github.io

Ciaburrio G. Hands-On Simulation Modeling with Python (2e). Packt; 2022. packtpub.com

Python Software Foundation. Python (Version 3.8.10). Python Software Foundation. Published 2021. Accessed August 19, 2023. python.org