KIT103/KMA155 Practical 12: Modelling and Simulation

This week you'll play with a collection of modelling and simulation functions and, if you have time and interest, can also have a quick play with some of Python's array and matrix types.

Preparation

In the Labs section of MyLO download the file **modelling.py**. Although the file is long, much of the code within it is not overly complex. Each function also includes a substantial documentation comment to explain its behaviour.

Optional: Windowed Graphs

When creating charts and other graphs, Spyder's default behaviour is to render (draw) them inside the IPython terminal. This makes the chart completely non-interactive and precludes dynamically resizing it. If you'd like to be able to resize the charts generated then you have two alternatives:

1. Select *Consoles* | *Open a Python console* from the menu. Then run the file and enter the various commands into the Python console instead of the IPython one. *OR*
2. Close the IPython terminal in Spyder. Select *Tools* | *Preferences* from the menu. In the IPython section, select the *Graphics* tab, then change *Graphics Backend* from *Inline* to *Automatic*. Select *Consoles* | *Open an IPython console* for the changes to take effect. Now run the file. (Note that you need to change this back if you want the inline drawing behaviour restored.)

1 Modelling Coins, Dice and the Weather

The script modelling.py contains a large number of functions for simulating various stochastic ('random') behaviours and plotting the results. You'll work in the IPython terminal today, so run the script file and then work through the following exercises.

1.1 Tossing a Coin ¢

"Hmmm. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads." – Rosencrantz, from *Rosencrantz and Guildenstern are Dead*

In the script file are the following functions related to simulating the toss of a (biased) coin:

- **`coin_toss(n, p)`:** returns the outcomes of `n` coin tosses where the probability of coming up heads is `p` $\in [0,1]$.
- **`plot_coin_toss(n, p)`:** runs `coin_toss()` and plots the outcomes of each trial.
- **`coin_toss_average(p, upto)`:** performs `upto` independent `coin_toss()` runs, varying the number of trials between 1 and `upto`; returns lists of the proportion of heads in each run and expected probability.
- **`plot_coin_toss_average(p, upto)`:** runs `coin_toss_average()` and plots the result.

Experiment with the following:

- Call `plot_coin_toss()` with a few different values of `n` (e.g., 10, 50) and `p` (e.g., 0.1, 0.5, 0.7).
  - Perform the same run twice, for some combination of `n` and `p`. Are the outcomes identical?
  - Perform the same run twice but prior to each run call `reset_random_numbers()`. What do you observe now?
- Call `plot_coin_toss_average()` with various values of `p` (e.g., 0.1, 0.5, 0.7) and observe the results.

1.2 Rolling a Die ⚄

"God doesn't play dice with the world" – Albert Einstein

In the script file are the following functions related to simulating the roll of a (biased) die:

- **`die_roll(n, sides, probs)`:** returns the outcomes of `n` rolls of a `sides`-sided die with the probabilities of each side given by `probs`. `sides` and `probs` can be left out and default values assumed.
- **`plot_die_roll(n, sides, probs)`:** plots the outcomes from `die_roll()`.
- **`plot_fair_die_average(sides, upto)`:** runs `coin_toss_average()` with the (fair) probability of one side of a die with `sides` sides and plots the result.

Experiment with the following:

- Call `plot_die_roll()` with various values of `n` and `sides`. Is there any pattern in the output?
- Call `plot_die_roll()` with `probs=[0.1, 0.1, 0.1, 0.5, 0.1, 0.1]` as one of the arguments and observe the result. Try some different values of `probs` (note that if `len(probs)` is not six then you must set `size` to match).
- Call `plot_fair_die_average()` with for dice with various numbers of `sides` and observe the results.
  - Why does this function, relating to dice, call `coin_toss_average()`?
  - What changes would be required for this to model the average occurrences of a given side if the die was not fair?

1.3 Predicting the Weather ☀↔☁

"Why does it always rain on me?" – Travis

The script file contains the following definitions and functions related to simulating the evolution of the weather using a three-state model of sunny, rainy and cloudy:

- **`Weather`:** an enumerated type that allows us to use more meaningful identifiers for the three states (which are otherwise 0, 1 and 2). You can refer to one of the enumeration's values by `Weather.sunny` and so on.
- **`P1`** and **`P2`:** matrices which include sample transition probabilities.
- **`simulate_weather(n, x0, P)`:** returns the predicted daily weather using the model for `n` days given an initial state in `x0` (which may be an integer or value of `Weather`) and transition probability matrix `P`.
- **`plot_weather_simulation(n, x0, P)`:** plots the predictions from `simulate_weather()`.
- **`simulate_weather_avgs(n, x0, P)`:** simulates the weather using `simulate_weather` then calculates the running averages for each weather state over the course of the simulation. Creates two plots: the day-to-day weather and the running averages.
- **`plot_weather_evolution(n, x0, P)`:** plots the outputs from `simulate_weather_avgs()`.

Experiment with the following:

- Call `plot_weather_simulation()` with `n` = 1000, `x0` any value of `Weather` (`sunny`, `rainy` or `cloudy`) and `P` = `P1` (one of the samples included in the script file).
  - Next, try it with transition probabilities `P2`
  - And *then* try it with a transition probability matrix of your own devising.
- Call `plot_weather_evolution()` with each of the matrices you tried above and observe not just the change day to day but also the running average proportion of times each weather state is observed.
- Do you remember that you can count the number of paths between nodes in a graph by multiplying the adjacency matrix by itself? Take any of the transition matrices (`P1`, `P2` or your own) and try the following:

  ```
  P1**10
  ```

  What do you notice about the values in the matrix? How do they compare to the final values in the plot produced by `plot_weather_evolution()`?

2 Optional after class challenge: Smooth running averages

The current coin toss and die roll average functions conduct `upto` independent experiments, which is why the lines jump around a lot. Another approach would be to conduct one ongoing experiment and revise the running average at each step, similar to the weather simulation. This results in a smooth line that will, over time, approach the value of `p`. Try implementing this.

3 After class: Matrix operations in Python

Python (through the add-ons NumPy and SciPy) has good facilities for representing and processing vectors and matrices. In this collection of tasks you'll get to experiment just a little with these.

3.1 numpy.array

The NumPy `array` type can be used for vectors (row and column vectors are interchangeable when represented as `arrays`) and matrices. One of the easiest ways to create a vector or matrix using the `array` data type is to define its contents in a Python list (or a list of lists). For instance, a vector $\mathbf{a} = \begin{bmatrix} 2 & 3 & 4 \end{bmatrix}$ can be declared using:

```
import numpy as np
a = np.array([2, 3, 4])
```

while the matrix $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ could be declared with `A = np.array([[1, 2], [3, 4]])`

Task: Declare another `array` to represent the vector $\mathbf{b} = \begin{bmatrix} 0 & 1 & 5 \end{bmatrix}$ and try out some of the example vector operations from the Apps 11 lecture.

3.2 numpy.matrix

Most operations on NumPy `arrays` are element-wise, meaning they apply the operation to each element within the array. To perform matrix multiplication (including raising a matrix to a power) more succinctly it is better to use NumPy's `matrix` type.

Task: Try the following, in order:

1. Declare the matrix $\mathbf{A}$ from above as an `array`.
2. Evaluate the Python expression `A**2`. **Before you do, what do you expect the result to be?**
3. Define a matrix $\mathbf{B}$ by converting `A` to a NumPy `matrix`, using `B = np.asmatrix(A)`. (A `matrix` can also be created directly using the same syntax as a two-dimensional `array` used to declare `A` above.)
4. Evaluate `B**2`. **Before you do, what do you expect the result to be?** (This is actually trickier if you have not had the opportunity to do any linear algebra before, so you don't have to get it correct right now.)

See also

For a more 'serious' application of vectors in visualisation take a look at these examples (each has downloadable code that will generate the plot; if viewing this on MyLO then right-click the link and open in a new tab):

Vectors to illustrate forces in a 2D space:
http://matplotlib.org/examples/pylab_examples/quiver_demo.html
Vectors illustrating forces in a 3D space:
http://matplotlib.org/examples/mplot3d/quiver3d_demo.html
Vector flow as (curved) streams:
http://matplotlib.org/examples/images_contours_and_fields/streamplot_demo_features.html

Useful references for the future

NumPy array
https://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html
NumPy matrix
https://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html