sheet3

May 16, 2017

1 Exercise Sheet 3: Sampling, Simulation

In this exercise sheet, we will simulate a Markov chain. In the first part, we will consider a pure Python based implementation where a single particle jumps from one position to another of the lattice, where all transitions to neighboring states have the same probability. Then, we will add probability weightings for the transitions. Finally, the implementation will be parallelized to run many chains in parallel.

```
In [1]: %matplotlib inline
```

1.1 Exercise 1: Random moves in a lattice (20 P)

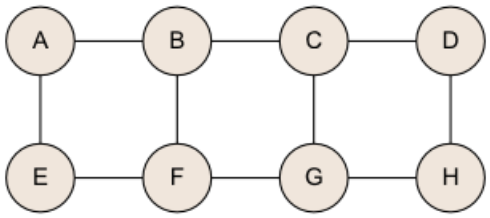
In this exercise, we will simulate the propagation of particles in a graph composed of 8 states (denoted by letters A-H) and stored in the variable `S` defined in the cell below. The lattice is the following:

The particle starts in state `A` and then jumps randomly from its current state to one of its neighbors, all with same probability. Note that it cannot stay at the current position. The dictionary `T` defined in the cell below encode such transition behavior.

```
In [2]: # List of states
        S = 'ABCDEFGH'

        # Set of transitions
        T = {'A':'BE','B':'AFC','C':'BGD','D':'CH','E':'AF','F':'EBG','G':'FCH','H'
```

Using pure Python, simulate the experiment below and run it for 1999 iterations. Print the sequence of first 400 states visited by the particle. To obtain the same results as in `pdf` solution file, you should initialize the seed of the module `random` to value `123` using the function `random.seed` before starting the simulation.



```
In [3]: import random as rng
        rng.seed(123)
        def simulpy(init='A',n=1999,trans=T):
            state,path = init,init
            for i in range(1999):
                state=rng.choice(trans[state])
                path += state
            return path
        path=simulpy()
        for i in range(5):
            print(path[i*80:(i+1)*80])
```

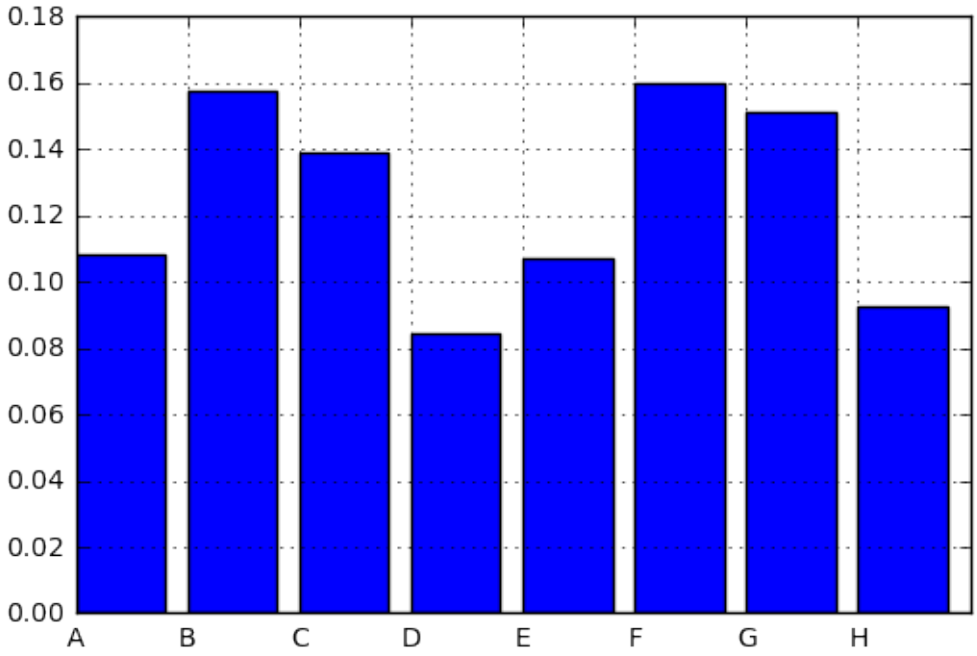
```
ABABAEAEAEABFEABAEABFGFEFEFBAEFBCBFBCBFGCDHDHGHGFGFGHDCGCBFBFEAEABAEFGCDHDCGFGCD
CBABFEABABFEABFEFBCBFGCGCDHGHDCBAEAEFEFGHGFGCGHGFEFBCDHDCDCGHDCGCGHGFBABFGFEAEFE
FEABFGHDHGHDCDHGCBFEFGHDCDCBCGHGCDCDHGHGCGHGCGHGFGFGFBAEFEFEAEAEFGFGFEFGCBCBCDCD
CDHDCGHGCDCBABFBFEAEFGHGHGFEAEABCBCGHGCDCGCBABAEAEAEFEFGHGFEAEFEABABAEAEAEABAEFE
AEFEFEAEAEFBABCDCBFGFGFEABCGFEABCGHDHGCDCBABCDCBCBAEFBCDHGCDCGHDHGHGCDHGHDCBFGFB
```

Using `matplotlib`, produce a bar plot (`matplotlib.pyplot.bar`) showing the fraction of the time the particle is found in a given state, averaged over the whole simulation.

```
In [4]: import matplotlib.pyplot as plt
        from collections import Counter

        def plotbars(path):
            c = Counter(path)
            p = [float(c[i])/len(path) for i in sorted(c)]
            bars = plt.bar(range(len(S)),p, color='b')
            plt.xticks(range(len(S)), S)
            plt.grid(color='k', linestyle=':')
            plt.show()

        plotbars(path)
```

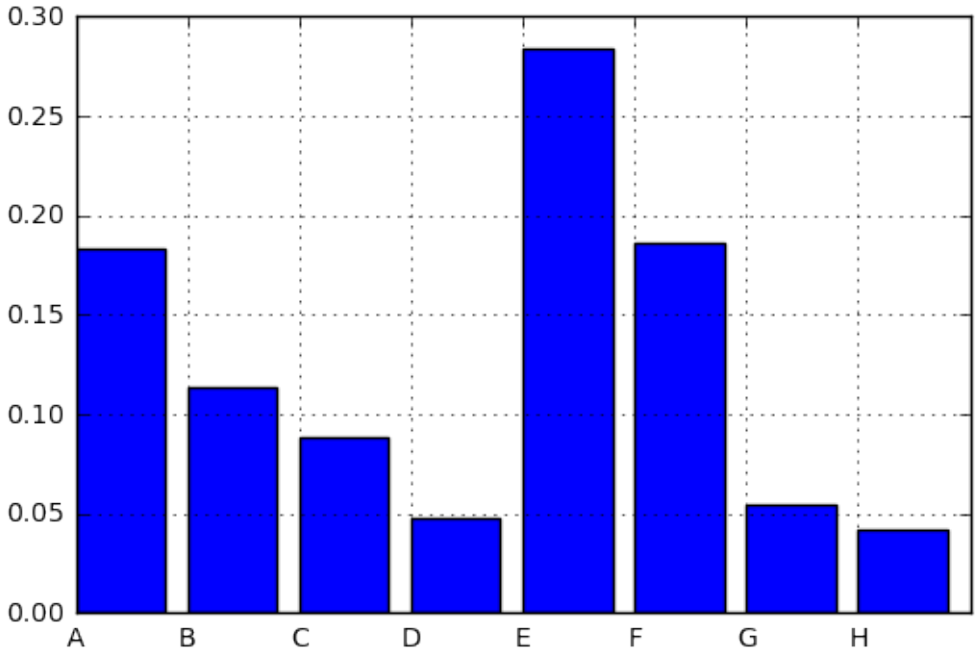


1.2 Exercise 2: Adding a special state (20 P)

Suppose now that the rule is modified such that everytime the particle is in state `F`, it always moves to `E` in the next step.

- Modify the code to handle this special case, and create a bar plot for the new states distribution.

```
In [5]: import copy
        T2=T.copy()
        T2['F']='E'
        path=simulpy(trans=T2)
        plotbars(path)
```



1.3 Exercise 3: Exact solution to the previous exercise (20 P)

For simple Markov chains, a number of statistics can be obtained analytically from the structure of the transition model, in particular, by analysis of the transition matrix.

- Compute the transition matrices associated to the models of exercise 1 and 2 (make sure that each row in these matrices sums to 1).
- Give the transition matrices as argument to the function `utils.getstationary(P)` and print their result.

This last function computes in closed form the stationary distribution associated to a given transition matrix `P` (i.e. the one we would get if running the simulation with such transition matrix for infinitely many time steps and looking at state frequencies).

```
In [6]: import numpy as np
        import utils

        def transitionmatrix(trans=T):
            n = len(S)
            P = np.zeros((n,n))
            for i in range(n):
                for j in range(n):
                    P[i,j] = 1.0 if S[j] in trans[S[i]] else 0.0
            P = P / P.sum(axis=1)[:,np.newaxis]
            return P
```

```
        P1 = transitionmatrix()
        P2 = transitionmatrix(T2)

        print("Exercise 1 {}".format(utils.getstationary(P1).round(decimals=2)))
        print("Exercise 2 {}".format(utils.getstationary(P2).round(decimals=2)))
```

```
Exercise 1 [ 0.1   0.15  0.15  0.1   0.1   0.15  0.15  0.1 ]
Exercise 2 [ 0.19  0.12  0.08  0.04  0.29  0.2   0.04  0.04]
```

1.4 Exercise 4: Adding non-uniform transition probabilities (20 P)

We consider the original lattice defined by the variable `T`. We set transition probabilities for each state to be such that: (1) the probability of moving left is always twice the probability of moving right when both moves are available. (2) The probability of moving vertically is the same as the probability of moving horizontally.

- Build the transition matrix `P` implementing the described behavior, and compute its stationary distribution using the function `utils.getstationary(P)`.

(Hints: You can notice that for each state, the transitions towards other states are always listed from left to right in the dictionary `T`. Also note that characters A-H can be mapped to integer values using the Python function ord(), thus, giving a direct relation between state names and indices of the transition matrix.)

```
In [7]: P=np.array([[0.,1.,0.,0.,1.,0.,0.,0.],
                    [2.,0.,1.,0.,0.,3.,0.,0.],
                    [0.,2.,0.,1.,0.,0.,3.,0.],
                    [0.,0.,1.,0.,0.,0.,0.,1.],
                    [1.,0.,0.,0.,0.,1.,0.,0.],
                    [0.,3.,0.,0.,2.,0.,1.,0.],
                    [0.,0.,3.,0.,0.,2.,0.,1.],
                    [0.,0.,0.,1.,0.,0.,1.,0.]])
        P = P / P.sum(axis=1)[:,np.newaxis]

        print(utils.getstationary(P).round(decimals=2))
```

```
[ 0.14  0.21  0.11  0.04  0.14  0.21  0.11  0.04]
```

1.5 Exercise 5: Simulation for multiple particles (20 P)

We let 1000 particles evolve simultaneously in the system described in Exercise 4. The initial state of these particles is pseudo-random and given by the function `utils.getinitialstate()`.

- Using the function `utils.mcstep()` that was introduced during the lecture, simulate this system for 500 time steps.
- Estimate the stationary distribution by looking at the distribution of these particles in state space after 500 time steps.

For reproducibility, give seed values to the function utils.mcstep corresponding to the current time step of the simulation (i.e. from 0 to 499).

```
In [8]: X = utils.getinitialstate()
        na = np.newaxis

        P5 = np.pad(P,1,mode='constant')[1:-1,:-1]

        for i in range(500):
            X=utils.mcstep(X,P5,i)

        print(X.mean(axis=0))
```

```
[ 0.145  0.238  0.096  0.03   0.139  0.214  0.096  0.042]
```