Homework 6

Math 87

1 Strolling down an infinite street

1. Your position at time t is (randomly chosen to be) one step to the left or to the right of your previous position, and hence is only dependent on your position at time $t-1$. It doesn't matter (for instance) what sequence of steps took you to get to your position at time $t-1$, and hence this is a Markov chain.

2. Similar to part 1, your distance at time t is one greater than or one less than your distance at time $t-1$.

3. By the same reasoning as before, this is a Markov chain, there are just now more states you may transition to from a given position x at time t. To compare this to the process in part 2, you might have said that your position as a function of time should vary more slowly than the processes in part 1 and 2, since there is now a chancne that the distance doesn't change at any given time. To make this precise one would need to define variance for Markov chains.

4. For the first experiment, there is a 100% probability of being at an odd number at odd times, i.e. times 1 and 3, and 0 otherwise. For experiment 2, this amounts to a somewhat tedious calculation. It is made easier when one notices that the probability distributions obtained this way are symmetric, in the sense that the probability of being at a number k is the same as the probability of being at $-k$. The probability of being at an odd number at time 0 is 0, at time 1 is 0.5, at time 2 is 0.5, at time 3 is 0.71875 and at time 4 is 0.49609.

5. It is a fact that two random walkers will eventually meet regardless of where they start.

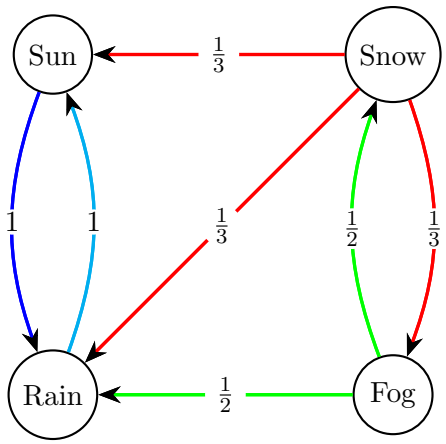
2 Rain or shine

On Planet X, the weather is strangely predictable: The weather is always either sunny, rainy, foggy or snowy. If it rains today, its sunny tomorrow. If it is sunny today, its rainy tomorrow. If its foggy today, its not sunny tomorrow. Finally, the weather is never the same two days in a row. Apart from these rules, the weather is completely random, in that if e.g. its foggy today it is equally likely to be either rainy or snowy tomorrow. You live on Planet X and are trying to figure out what to wear this week, so you'd like to develop a model for the weather.

1. The weather the following day is a probability distribution based on the previous day's weather. Hence it can be modeled by a Markov chain. The transition matrix of this Markov chain is as follows:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1/3 & 1/3 & 0 & 1/3 \\ 0 & 1/2 & 1/2 & 0 \end{bmatrix}$$

   where the first row and column is indexed by the state "Sunny", the second row/column by "Rainy", the third row/column by "Snowy" and the fourth row/column by "Foggy." The associated automata is:



2. The Perron-Frobenius theorem is not satisfied. Recall that strongly connected means that there is a path from each node to every other node, and aperiodic means that all cycles have length that is not divisible by a common factor. First, it's clear that the graph is not strongly connected since rainy and sunny only connect to each other. There is no path from either rainy or sunny to snowy or foggy. The transition graph is not aperiodic since the only two cycles are between sunny and rainy, and snowy and foggy. Therefore, all cycles have lengths of multiples of two.

3. Since P-F is not satisfied, we are not guaranteed that the power method converges. Since P-F is not satisfied, there may not be a unique dominant eigenvalue equal to one for the transition matrix T. Since there may not be a dominant (largest magnitude) eigenvalue, we cannot guarantee convergence of the power method.

4. (With reindexed columns) the code to compute eigenvectors via the power method is as follows:

Eigenvalue Decomposition

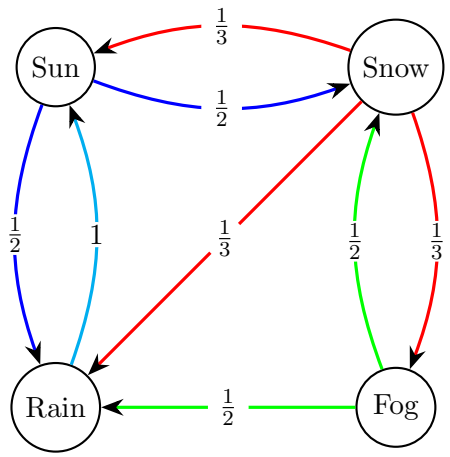
```
import numpy as np
from numpy import linalg as LA
T= np.array([[0, 1/3, 0, 1], [0, 0, 1/2, 0],
             [0, 1/3, 0, 0], [1, 1/3, 1/2, 0]])
D, V = LA.eig(T)
print('Eigenvalues:')
print(D)
print('\n')
print('Eigenvectors:')
print(V)
```

```
Eigenvalues:
[ 1.         -1.          0.40824829 -0.40824829]


Eigenvectors:
[[ 0.70710678  0.70710678 -0.57469305 -0.18265886]
 [ 0.          0.          0.54566516 -0.72140928]
 [ 0.          0.          0.44553374  0.58902821]
 [ 0.70710678 -0.70710678 -0.41650584  0.31503993]]
```

We can again see that P-F cannot be satisfied since we have a 1 and -1 eigenvalue. Since we have two dominant eigenvalues with the same magnitude, we expect that the power method does not converge.

5. One path is added to the transition diagram. Dark blue edges now have probability $\frac{1}{2}$.



The new transition matrix is:

$$\begin{bmatrix} 0 & 1/2 & 1/2 & 0 \\ 1 & 0 & 0 & 0 \\ 1/3 & 1/3 & 0 & 1/3 \\ 0 & 1/2 & 1/2 & 0 \end{bmatrix}$$

Let's quickly note why this fixes the P-F theorem. The transition diagram is now strongly connected since the Sunny and rainy nodes are not a cycle anymore, meaning if we get the to the Sunny or Rainy nodes, we can now go from Sunny to Snowy with the new edge, and from Snowy to Foggy. Next, the transition diagram is now acyclic, since we have a cycle of length 3: Snowy $\to$ Rainy $\to$ Sunny $\to$ Snowy. The previous cycles between Rainy and Sunny, and Snowy and Foggy, are still there, both with length 2. However, 2 and 3 have no common factors, so this is enough to show that there are cycle lengths that do not have any common factors.

Now, run the power method again with the new matrix T.

Power Iteration

```
import numpy as np
from numpy import linalg as LA

#write your stochastic matrix here:
T= np.array([[0, 1, 1/3, 0], [1/2, 0, 1/3, 1/2],
                        [1/2, 0, 0, 1/2], [0, 0, 1/3, 0]])

#choose an initial vector: must be a probability vector
v1 = np.array([0.25, 0.25, 0.25, 0.25])

#update vector---make sure |v1-v0| > tol so it enters the loop
v0 = np.zeros((4))

#stop tolerance
tol = 1e-6

#iteration count and maxiter
iter = 0
maxiter = 100
```

```
#while |v1-v0| > tol and maxiters hasn't been reached
#do the power method
while LA.norm(v1-v0)>tol and iter < maxiter:
        v0 = v1
        v1 = T@v1
        iter+=1


#print stuff about the power method
if iter < maxiter:
  print('Power method converged!')
else:
  print('Power method did not converge :(')

print(f'Iterations: {iter}')
print(f'v = {v1}')
```

The power method now converges. We can confirm that the output is a probability by summing the entries:

$$0.38461513 + 0.23076945 + 0.07692298 + 0.30769244 = 0.9999999999999989$$

which is essentially 1, as required.

The new eigen-decomposition is given by:

```
Eigenvalues:
[ 1.00000000e+00 -7.88675135e-01 -2.11324865e-01  7.03757498e-17]


Eigenvectors:
[[ 7.00140042e-01  6.84550319e-01  5.08589803e-01  7.07106781e-01]
 [ 5.60112034e-01 -3.42275160e-01 -2.54294901e-01  4.53246652e-17]]
 [ 4.20084025e-01 -5.92837967e-01  4.40451689e-01 -2.26623326e-16]
 [ 1.40028008e-01  2.50562807e-01 -6.94746591e-01 -7.07106781e-01]
.
```

We now have only one eigenvalue with magnitude one. The corresponding eigenvector is a multiple of the output probability vector:

$$\begin{bmatrix} 0.700140042 \\ 0.560112034 \\ 0.420084025 \\ 0.140028008 \end{bmatrix} \approx 1.82 \begin{bmatrix} 0.38461513 \\ 0.30769244 \\ 0.23076945 \\ 0.07692298 \end{bmatrix}.$$

where the first/second/third/fourth entry corresponds to Sunny/Rainy/Snowy/Foggy respectively.

3 Geometry feat. Monte Carlo

Let C be the circle defined by $(x-1.25)^2 + (y-1.25)^2 = 0.4$. We know that its interior has area 0.4π, but suppose that we didn't. Your task is to compute this area by Monte Carlo simulation. Here is an outline: Define a region D containing C with known area (hint: for instance, take the square defined $0 \leq x \leq 2$, $0 \leq y \leq 2$). Write a program that uniformly samples from this region D (from class, you know how to uniformly sample an interval. If you chose D to be a square, this could come in handy...). Using this, and a Monte Carlo "rejection sampling" method, estimate the area within C. Your algorithm should let you choose the number of samples you take.

1. The true area of the circle is $0.4^2\pi$. Example function for computing samples that land within the circle:

Sampling

```
import numpy as np
import matplotlib.pyplot as plt
def sample(n):
    b=0
    for i in np.arange(n):
        x=np.random.uniform(0,2)
        y=np.random.uniform(0,2)
        if (x-1.25)**2+(y-1.25)**2<0.4
            b+=1
    d=4*b/n
    return d
```

2. Example code that computes the number of samples in the COMPLEMENT of the pair of circles in the square $[0,2] \times [0,2]$:

Sampling

```
import numpy as np
import matplotlib.pyplot as plt
def sample(n):
    b=0
    for i in np.arange(n):
        x=np.random.uniform(0,2)
        y=np.random.uniform(0,2)
        if (x-1.25)**2+(y-1.25)**2>0.4 or (x-.75)**2+(y-.75)**2>.3:
            b+=1
    d=4*b/n
    return 4-d
```