Computational Physics

Topic 02 : Computational Problems involving Marko Chains

Lecture 01 : Review of Markov Chains

Dr Kieran Murphy
Computing and Mathematics, SETU (Waterford).
(kieran.murphy@setu.ie)

Autumn Semester, 2025/26

Outline

- Some simple models
- Terminology and definitions

Example (Land of Oz)

Description

The Land of Oz is blessed by many things, but not by good weather. They never have two nice days in a row. If they have a nice day, they are just as likely to have snow as rain the next day. If they have snow or rain, they have an even chance of having the same the next day. If there is change from snow or rain, only half of the time is this a change to a nice day.

Example (Land of Oz)

Description

The Land of Oz is blessed by many things, but not by good weather. They never have two nice days in a row. If they have a nice day, they are just as likely to have snow as rain the next day. If they have snow or rain, they have an even chance of having the same the next day. If there is change from snow or rain, only half of the time is this a change to a nice day.

Formalisation . . . as a transition matrix

		. . . to . . .		
		Rain	Nice	Snow
. . . from . . .	Rain	1/2	1/4	1/4
	Nice	1/2	0	1/2
	Snow	1/4	1/4	1/2

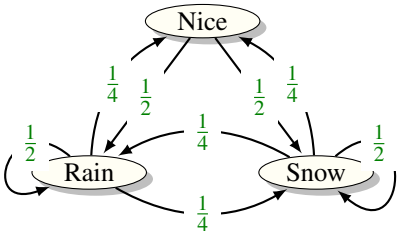
Example (Land of Oz)

Description

The Land of Oz is blessed by many things, but not by good weather. They never have two nice days in a row. If they have a nice day, they are just as likely to have snow as rain the next day. If they have snow or rain, they have an even chance of having the same the next day. If there is change from snow or rain, only half of the time is this a change to a nice day.

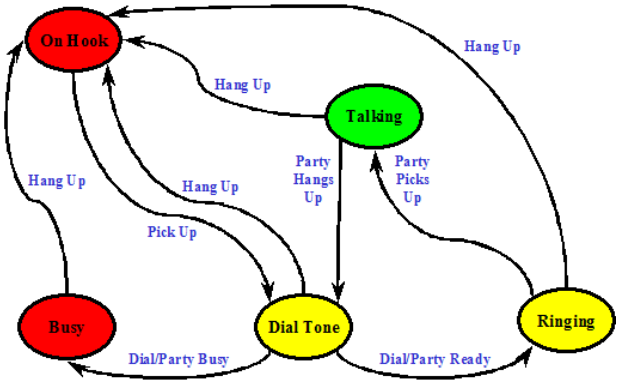
Formalisation . . . as a transition matrix . . . and finite state machine

$$
\begin{array}{cc}
 & \\
 & \text{Rain} \\
\text{from} \ldots & \text{Nice} \\
 & \text{Snow}
\end{array}
\begin{array}{c}
\ldots \text{ to } \ldots \\
\begin{array}{ccc}
\text{Rain} & \text{Nice} & \text{Snow}
\end{array} \\
\begin{pmatrix}
1/2 & 1/4 & 1/4 \\
1/2 & 0 & 1/2 \\
1/4 & 1/4 & 1/2
\end{pmatrix}
\end{array}
$$



Example (Model Based Testing)

Model-Based Testing*is the automatic generation of efficient test procedure sequences — using finite state machines or Markov chains – to represent a system's requirements and specified functionality.



*Google "Model Based Testing" or read Harry Robinson's (Microsoft) paper
http://www.ecs.csun.edu/ rlingard/COMP595VAV/GraphTheory Techniques In Model-Based Testing.pdf

Example (Worker Employment Model) I

Example 1

Consider a worker who, at any given month, t, is either unemployed (state 0) or employed (state 1). Suppose that, over a one month period:

- An unemployed worker finds a job with probability $\alpha \in (0, 1)$.
- An employed worker loses their job and becomes unemployed with probability $\beta \in (0, 1)$.

Formulate model as a Markov chain[†]. Then:

1. What is the average duration of unemployment?
2. Over the long-run, what fraction of time does a worker find themself unemployed?
3. Conditional on employment, what is the probability of becoming unemployed at least once over the next 12 months?

[†]Since this model has only two states we could model this using the geometric distribution.

Specifying a Markov Chain I

Any Markov process/chain can be described as follows:

- We have a set of states, $S = \{s_1, s_2, \ldots, s_r\}$.

  For the *Land of Oz* example the states were

  $$S = \{\underbrace{\text{Rain}}_{s_1}, \underbrace{\text{Nice}}_{s_2}, \underbrace{\text{Snow}}_{s_3}\}$$

- The process starts (at step/stage 0) in one of these states and moves successively from one state to another, one step/stage at a time.
- Each move is called a step, and after n steps the process is at stage n, generating a run.

  For the *Land of Oz* example some possible runs are:

  - Rain, Rain, Rain, Nice, Snow, Rain, Snow, Snow, Nice, ...
  - Rain, Rain, Rain, Rain, Rain, Rain, Rain, Rain, ...

  while the following sequence is not possible (Why?)

  - Snow, Snow Snow, Nice, Nice, Snow, Snow, Snow, ...

Specifying a Markov Chain II

- If the chain is currently in state s_i, then it moves to state s_j at the next step/stage with a transition probability denoted by p_{ij}.

  The probability does not depend upon which states the chain was in before the current state $\Longrightarrow$ The next state only depends on the current state and not the past states (Markov memory less property).

For the *Land of Oz* example the matrix of transition probabilities is

$$\begin{array}{cc} & \begin{array}{ccc} \multicolumn{3}{c}{\ldots \text{ to } \ldots} \\ \text{Rain} & \text{Nice} & \text{Snow} \end{array} \\ \begin{array}{c} \ldots \text{ from } \ldots \\ \text{Rain} \\ \text{Nice} \\ \text{Snow} \end{array} & \begin{pmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/2 \end{pmatrix} \end{array} = P$$

- The Markov chain can remain in the state it is in, and this occurs with probability p_{ii}.

Specifying a Markov Chain III

- The random variable X_n represents the state of the Markov chain at stage n. It assumes values $S = \{s_1, s_2, \ldots, s_r\}$ with probability distribution

$$\boldsymbol{u}_n = (u_{n1}, u_{n2}, \ldots, u_{nr})$$

- The Markov memoryless property can be expressed in terms of conditional probability as follows

$$Pr(X_{n+1} = s | X_n, X_{n-1}, \ldots, X_0) = Pr(X_{n+1} = s | X_n)$$

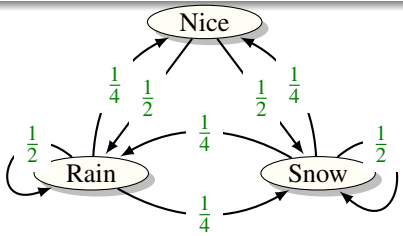
(No extra uncertainty is removed by knowing what has happened in the past.)

Of interest are:

- Given the initial state (i.e. given $\boldsymbol{u}_0$) what is the probability distribution for the state at some later stage n?
- What is the long term behaviour of the chain?

Transition Matrix

$$P = \begin{array}{c} \\ \text{from} \\ \vdots \end{array} \begin{array}{c} \\ \text{Rain} \\ \text{Nice} \\ \text{Snow} \end{array} \begin{array}{c} \overset{\text{... to ...}}{\text{Rain}} \\ \\ \end{array} \begin{array}{c} \text{Nice} \\ \\ \end{array} \begin{array}{c} \text{Snow} \\ \\ \end{array} \begin{pmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/2 \end{pmatrix}$$



- From the total law of probability the **sum of the entries along every row equals one**. (Sum of probabilities of outgoing arcs on each node equals one).‡
- If given the probability distribution for some stage n, then the probability distribution for some later stage $n + m$, is

$$\boldsymbol{u}_{n+m} = \boldsymbol{u}_n P^m$$

In other words, to move the distribution forward m units of time, we post-multiply by P^m.

‡The sum down the columns need not be one (sum of probabilities of incoming arcs), but if the column totals are also one then this special type of process is called a doubly stochastic (more later).

Example (Land of Oz, cont)

Question

Today it is raining in the Land of Oz. What is the probability distribution for tomorrow?

Solution

Since $s_1 = \text{Rain}$, we have $\boldsymbol{u}_0 = (1, 0, 0)$ and we want $\boldsymbol{u}_1$.

Using $\boldsymbol{u}_1 = \boldsymbol{u}_0 P$ we have

$$(1,0,0)\begin{pmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/2 \end{pmatrix} = (1/2, 1/4, 1/4)$$

So the probability of rain tomorrow is 1/2, nice tomorrow is 1/4, and snow tomorrow is 1/4.

Example (Land of Oz, cont)

Question

Today it is nice in the Land of Oz. What is the probability of it raining in two days time?

Solution

Since $s_2 = \text{Nice}$, we have $\boldsymbol{u}_0 = (0, 1, 0)$ and we want the first component of $\boldsymbol{u}_2$.

Using $\boldsymbol{u}_2 = \boldsymbol{u}_0 P^2$ we have

$$(0,1,0)\begin{pmatrix} 0.5 & 0.25 & 0.25 \\ 0.5 & 0 & 0.5 \\ 0.25 & 0.25 & 0.5 \end{pmatrix}^2 = (0,1,0)\begin{pmatrix} 0.438 & 0.188 & 0.375 \\ 0.375 & 0.250 & 0.375 \\ 0.375 & 0.188 & 0.438 \end{pmatrix}$$
$$= (0.375, 0.250, 0.375)$$

So the probability of rain in two days time is 0.375, probability of being nice is 0.250 and probability of snow is 0.375.

Python Implementation I

Python setup

```
1 import numpy as np
  import matplotlib.pyplot as plt
  from numpy.random import default_rng
  rng = default_rng(12)
```

Define states

```
2 # label states for output
  labels = np.array(["Rain","Nice","Show"])
  states = np.arange(0, len(labels))
  print("Labels:", labels)
  print("States:", states)
```

```
Labels: ['Rain' 'Nice' 'Show']
States: [0 1 2]
```

Python Implementation II

Define transition matrix, *P*

```
# transition probability matrix
P = np.array([[1/2, 1/4, 1/4], [1/2, 0, 1/2], [1/4, 1/4, 1/2] ])
print("Transition probabilities:\n", P)
```

Compute distributions

```
u0 = np.array([1,0,0])
print("Stage 0:", u0)
print("Stage 1:", np.dot(u0,P))

u0 = np.array([0,1,0])
u2 = np.dot( np.dot(u0,P), P)
print("\nStage 2:", u2)
u2 = np.dot(u0, np.linalg.matrix_power(P,2))
print("or directly by P squared:", u2)
```

Python Implementation III

Rather than computing the probability distribution of each state we can simulate individual runs. To simplify things we write a generic function:

```
[0 2 1 2 2 0]
['Rain', 'Show', 'Nice', 'Show', 'Show', 'Rain']
```

```
5 def simulate_run(t_n, P, s_0, rng):

    # preallocate space for history
    history = np.zeros(t_n+1, dtype=int)

    # initial state
    history[0] = s_0

    for k in range(t_n):
        p = P[history[k]]
        history[k+1] = rng.choice(states, p=p)

    return history

history = simulate_run(5, P, 0, default_rng(42) )
print(history)
print([str(labels[k]) for k in history])
```

Python Implementation IV

Now we can use our function `simulate_run` to generate a few sample runs — each run is called a trajectory.

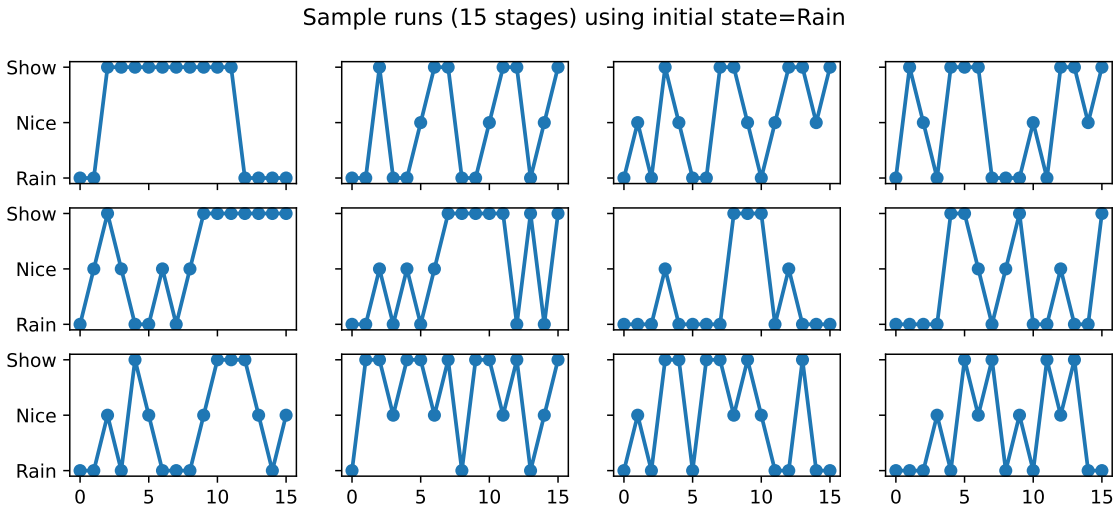
```
fig, axs = plt.subplots(3, 4, figsize=(10,4), sharey=True, sharex=True)

t_n = 15
rng = default_rng(667)

for ax in fig.axes:
    history = simulate_run(t_n, P, 0, rng)
    ax.plot(history, "o-",linewidth=2)

plt.yticks(states, labels)
plt.suptitle(f"Sample runs ({t_n} stages) using initial state={labels[history[0]]}")
plt.savefig("oz_runs.pdf", bbox_inches="tight")
plt.show()
```

Python Implementation V

Or better yet, we write a second generic function to generate a table of plots — each plot represent a single run of the Markov chain.

```
def plot_runs(t_n, P, s_0, labels, rng, filename=None, rows=3, cols=4):

    fig, axs = plt.subplots(rows, cols, figsize=(10,4), sharey=True, sharex=True)

    for ax in fig.axes:
        history = simulate_run(t_n, P, s_0, rng)
        ax.plot(history, "o-",linewidth=2)

        plt.yticks(states, labels)
    plt.suptitle(f"Sample runs ({t_n} stages) using initial state={labels[history[0]]}")

    if filename is not None: plt.savefig(filename, bbox_inches="tight")

    plt.show()

plot_runs(15, P, 0, labels, default_rng(667), filename="oz_runs.pdf" )
```

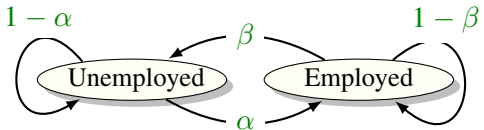
Land of Oz, Sample Runs I



Worker Employment Model, Representation

Formulating as a Markov chain we have:

$$P = \begin{array}{c} \\ \text{... from ...} \\ \end{array} \begin{array}{c} \\ \text{Unemployed} \\ \text{Employed} \end{array} \overset{\text{... to ...}}{\begin{array}{cc} \text{Unemployed} & \text{Employed} \\ \end{array}} \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix}$$



And in python (we have to pick a value for α and β, and typically we want to see the effect of different choices).

```
labels = np.array(["Unemployed","Employed"])
states = np.arange(0, len(labels))

alpha,beta = 0.1, 0.1
P = np.array([[1-alpha, alpha], [beta,1-beta] ])
```

Worker Employment Model, Implementation

Define model

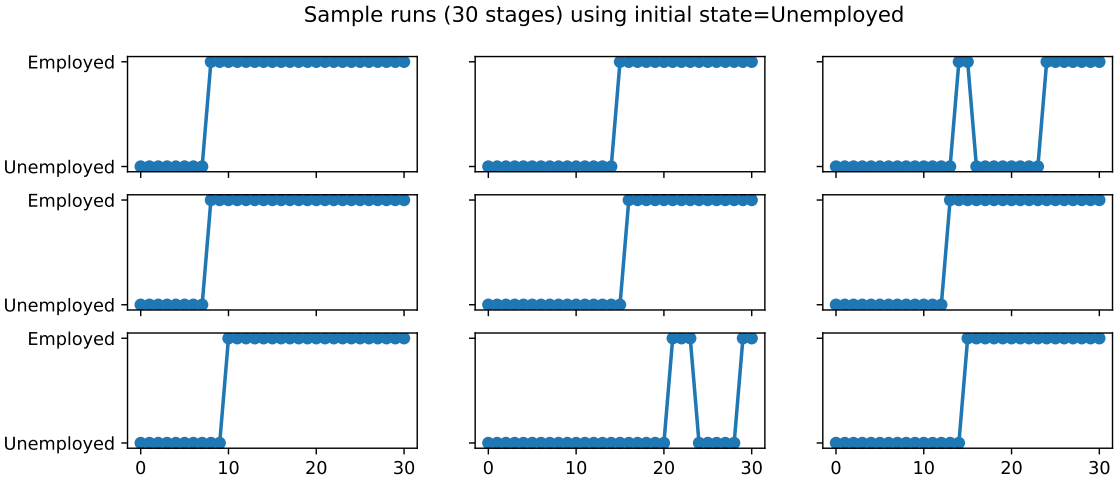
```
9 labels = np.array(["Unemployed","Employed"])
  states = np.arange(0, len(labels))

  alpha,beta = 0.1, 0.01
  P = np.array([[1-alpha, alpha], [beta,1-beta] ])
```

(I have picked the parameter values semi-randomly. They correspond to a 1/10 chance of getting a job in a month when unemployed and a 1/100 chance of losing their job in a month if employed.)

Generate plots of runs

```
10 plot_runs(30, P, 0, labels, default_rng(667), filename="worker_runs.pdf", cols=3 )
```

Worker Employment Model, Sample Runs ($\alpha = 0.1$, $\beta = 0.01$)



Computing Run Lengths I

11

```
t_n = 100_000
rng = default_rng(667)
history = simulate_run(t_n, P, 1, rng)

print(history[:100])
```

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

We want to know how a unemployed worker remains unemployed, or an employed worker remains employed. In terms of the model output this corresponds to computing run lengths of zero and ones in the history. General procedure is to get the difference of successive values using the `np.diff` function. Then

- `1` corresponds to switching start from ‘Unemployed’ to ‘Employed’.
- $-$`1` corresponds to switching start from ‘Employed’ to ‘Unemployed’.
- `0` corresponds to no change.

Computing Run Lengths II

12

```
t_n = 100_000
rng = default_rng(667)
history = simulate_run(t_n, P, 1, rng)

print(history[:100])
```

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

13

```
diff = np.diff(history)
print(diff[:100])
```

```
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0 -1  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0]
```

Computing Run Lengths III

14
```
run_starts = np.where(diff == -1)[0]
print(run_starts[:10])
```

```
[  75  233  276  530  670  871 1113 1287 1357 1452]
```

15
```
run_ends = np.where(diff == 1)[0]
print(run_ends[:10])
```

```
[  83  238  278  537  682  881 1115 1304 1358 1456]
```

16
```
n = min(len(run_ends), len(run_starts))
run_lens = run_ends[:n] - run_starts[:n]
print(run_lens[:10])
```

```
[ 8  5  2  7 12 10  2 17  1  4]
```

Computing Run Lengths IV

So to answer the questions

- What is the average duration of unemployment?

```
17 print(run_lens.mean())
```

```
10.150375939849624
```

- Over the long-run, what fraction of time does a worker find themself unemployed?

```
18 print((history==0).mean())
```

```
0.09449905500944991
```

- Conditional on employment, what is the probability of becoming unemployed at least once over the next 12 months?

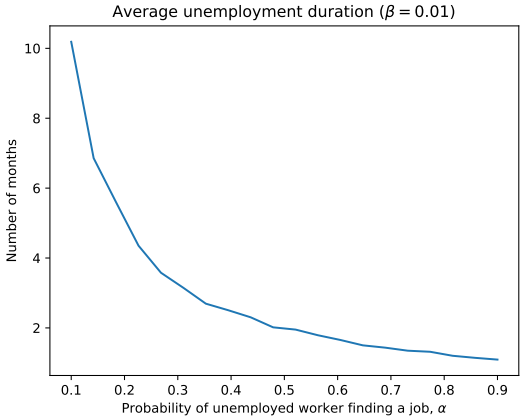
Left as exercise

Sensitivity Analysis for parameter α I

```
t_max = 100_000
beta = 0.01
alphaValues = np.linspace(0.1, 0.9, 20)
y = []
for alpha in alphaValues:

    P = np.array([[1-alpha, alpha], [beta,1-beta] ])

    history = simulate_run(t_n, P, 1, rng)

    diff = np.diff(history)
    run_starts = np.where(diff ==-1)[0]
    run_ends = np.where(diff == 1)[0]
    n = min(len(run_ends), len(run_starts))
    run_lens = run_ends[:n] - run_starts[:n]

    y.append(run_lens.mean())
```

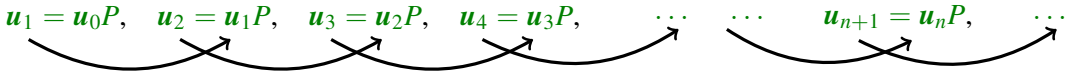
Sensitivity Analysis for parameter α II

```
20
plt.plot(alphaValues, y)
plt.title("Average unemployment duration ($\\beta=0.01$)")
plt.ylabel("Number of months")
plt.xlabel("Probability of unemployed worker finding a job, $\\alpha$")
plt.savefig("worker_pr_unemployed_wrt_alp
plt.show()
```



Equilibrium (Steady State) Distribution I

Consider the situation where we are given some starting probability distribution for the state and iteratively calculate the probability distribution for each successive stage. i.e.



Q: What happens to the sequence of generated probability distributions?
A: Under conditions:

- all states of the Markov chain communicate with each other (i.e., it is possible to go from each state, possibly in more than one step, to every other state),
- the Markov chain is not periodic (a periodic Markov chain is a chain in which, e.g., you can only return to a state in an even number of steps),

it is possible to show that the sequence of generated probability distributions converge to a finite probability distribution called the equilibrium distribution or steady state distribution.

Equilibrium (Steady State) Distribution II

Rather than going through the process of actually generating the sequence of probability distributions the steady state distribution can be calculated from the balance equation

$$\mu = \mu P \quad \text{subject to} \quad \sum_{i=1}^{r} \mu_i = 1 \tag{1}$$

- The equation is called a balance equation because it balances the probability of leaving and entering at each of the states. (This is analogous to the balance equations in the Network flow problems).
- The condition $\sum_{i=1}^{r} \mu_i = 1$ is required to ensure that the solution represents a probability distribution.

Example (Equilibrium distribution) I

Question

What is the steady state probability distribution for the weather in the Land of Oz?

Solution

Solving $\mu = \mu P$ subject to $\sum \mu_i = 1$ we have on setting $\mu = (\mu_1, \mu_2, \mu_3)$.

$$(\mu_1, \mu_2, \mu_3) = (\mu_1, \mu_2, \mu_3) \begin{pmatrix} \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \end{pmatrix} \implies \begin{aligned} 4\mu_1 &= 2\mu_1 + 2\mu_2 + \mu_3 \\ 4\mu_2 &= \mu_1 + \mu_3 \\ 4\mu_3 &= \mu_1 + 2\mu_2 + 2\mu_3 \end{aligned}$$

$$\sum \mu_i = 1 \implies 1 = \mu_1 + \mu_2 + \mu_3$$

We now appear to have four equations in three unknowns, but one of the equations is redundant — This is called an over-determined linear system.

Example (Equilibrium distribution) II

Simplifying we get

$$\begin{aligned} -2\mu_1 + 2\mu_2 + \mu_3 &= 0 \\ \mu_1 - 4\mu_2 + \mu_3 &= 0 \\ \mu_1 + 2\mu_2 - 2\mu_3 &= 0 \\ \mu_1 + \mu_2 + \mu_3 &= 1 \end{aligned} \implies \overbrace{\begin{pmatrix} -2 & 2 & 1 \\ 1 & -4 & 1 \\ 1 & 2 & -2 \\ 1 & 1 & 1 \end{pmatrix}}^{A} \overbrace{\begin{pmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \end{pmatrix}}^{\mu} = \overbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}}^{b}$$

$$\implies \mu_1 = \frac{2}{5}, \quad \mu_2 = \frac{1}{5}, \quad \mu_3 = \frac{2}{5}$$

Hence in the, long term, one-fifth of the days are nice and two-fifths are rain and two-fifths are snow.

For larger systems reproducing these step manually is painful. Instead we can use python, we can

- Construct the over-determined linear system $A\mu = b$ and solve for vector μ.
- Start with any initial distribution and iterate $\boldsymbol{u}_{n+1} = \boldsymbol{u}_n P$ until $\|\boldsymbol{u}_{n+1} - \boldsymbol{u}_n\|$ is small enough.

Python Implementation, solving linear system

```
# define Markov chain
P = np.array( [ [1/2,1/4,1/4], [1/2,0,1/2], [1/4,1/4,1/2] ])
n = P.shape[0]

# build matrix (note the transpose)
A = np.vstack( (P.T - np.eye(n), np.ones(n)) )
print("A =", A)

# build RHS vector
b = np.hstack( (np.zeros(n), [1]) )
print("\nb = ", b)

# solve the over-determined system
mu = np.linalg.lstsq(A, b, rcond=None)[0]
print("\nx = ", x[0])

# verify that we have a distribution: sum mu = 1
print("\n", np.isclose(mu.sum(),1))
# verify that we have a steady state distribution: mu = mu P
print(np.isclose( np.dot(mu, P), mu))
```

```
A = [[-0.5  0.5  0.25]
 [ 0.25 -1.   0.25]
 [ 0.25  0.5 -0.5 ]
 [ 1.    1.   1.  ]]

b =  [0. 0. 0. 1.]

x =  0.39999999999999974

 True
[ True  True  True]
```

Python Implementation, iterate system

```
22 tol = 1E-5

   # start with uniform distribution
   u = np.ones(n) / n

   # iterate till convergence
   for k in range(100):
       u_old = u
       u = np.dot(u_old, P)
       if np.linalg.norm(u - u_old) < tol: break
   else:
       print("Poor/no convergence. Exceeded max iterations")

   # output result
   print(u)
```

```
[0.40000025 0.19999949 0.40000025]
```