Computer Practical: Markov Chains

In this computer practical, you can use either Python or R. There is a list of useful Python and R functions at the end of this handout.

m-step transition kernel

In the lectures we have seen that the m-step transition kernel $\mathbf{K}^{(m)} = (\mathbb{P}(X_{t+m} = j | X_t = i))_{(i,j \in S)}$ is the m-th power of the transition kernel $\mathbf{K}$.

Task 1. Write a function which, given the transition kernel $\mathbf{K}$ and $m \in \mathbb{N}$, computes the m-step transition kernel $\mathbf{K}^{(m)}$.

Try out your function for the phone example from the lectures, i.e. $\mathbf{K} = \begin{pmatrix} 0.9 & 0.1 \\ 0.3 & 0.7 \end{pmatrix}$ (or any other example of your choice).

Distribution of states at time t

We have shown that if X_0 is drawn from the distribution $\boldsymbol{\lambda}_0$, then the distribution of X_t is given by the vector

$$\boldsymbol{\lambda}_0' \mathbf{K}^{(t)}$$

Task 2. Write a function which, given the transition kernel $\mathbf{K}$, initial distribution $\boldsymbol{\lambda}_0$, and $t \in \mathbb{N}$, computes the distribution of X_t. (You can call your function from task 1.)

Try out your function for the phone example from the lectures (or any other example of your choice).

Invariant distribution

The invariant distribution (given by the vector $\boldsymbol{\mu}$) satisfies

$$\boldsymbol{\mu}' = \boldsymbol{\mu}' \mathbf{K} \tag{1}$$

Thus we can use the following two approaches to compute $\boldsymbol{\mu}$:

1. Equation (1) is equivalent to
$$(\mathbf{K}' - \mathbf{I})\boldsymbol{\mu} = \mathbf{0}.$$
This system of linear equations is under-determined, so the solution is a one-dimensional linear subspace. However with the additional constraint that $\sum_{i \in S} \mu_i = 1$ we obtain a unique solution.

   Most numerical algorithms for solving linear systems of equations only work if there is a unique solution, so it is easiest to solve the system of equations
$$\begin{pmatrix} \mathbf{K}' - \mathbf{I} \\ 1 \ \dots \ 1 \end{pmatrix} \boldsymbol{\mu} = \begin{pmatrix} \mathbf{0} \\ 1 \end{pmatrix}.$$

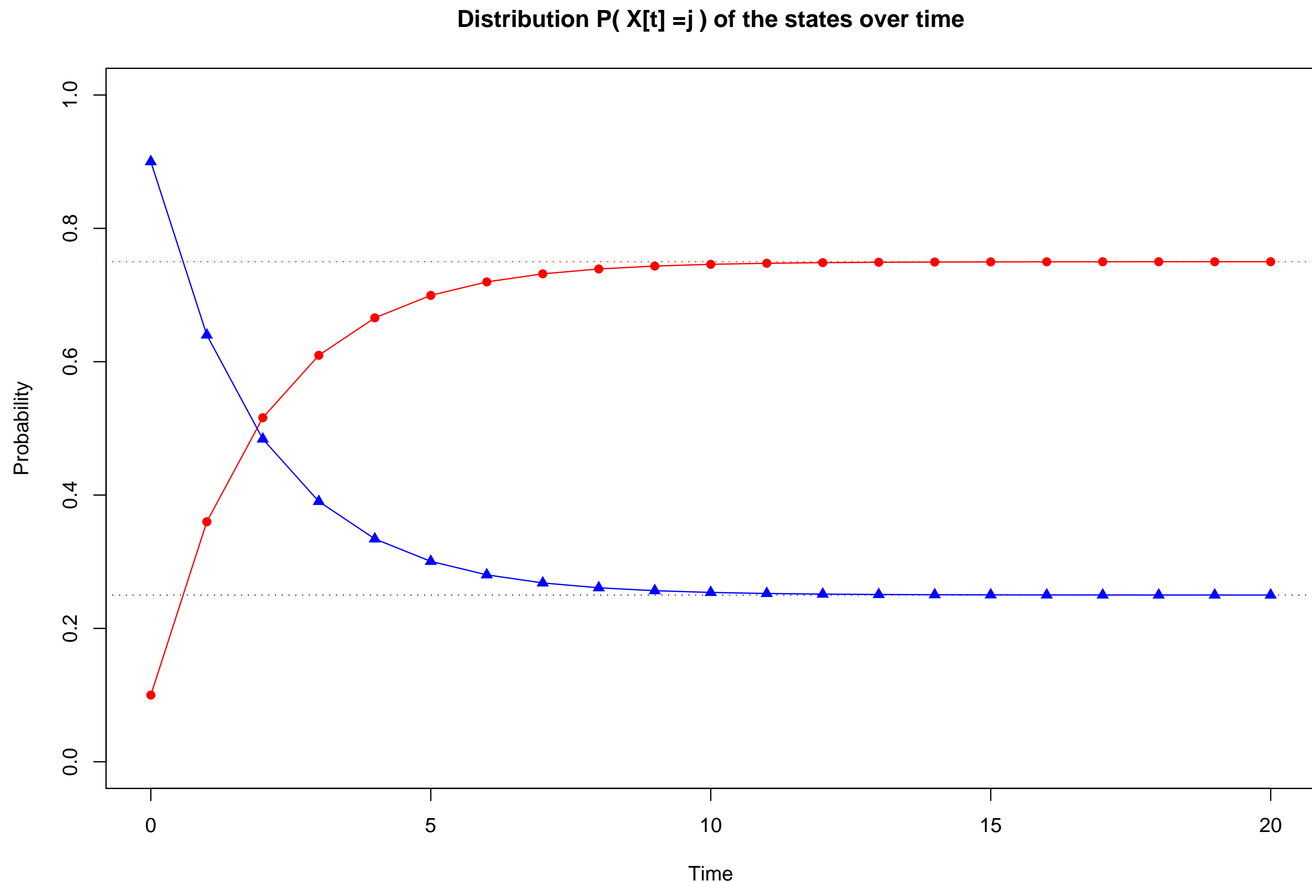
2. Equation (1) is equivalent to $\boldsymbol{\mu}$ being left eigenvector of $\mathbf{K}$ corresponding to the eigenvalue 1 (all other eigenvalues of $\mathbf{K}$ being less than 1), i.e. $\boldsymbol{\mu}$ is the (right) eigenvector of $\mathbf{K}'$ corresponding to an eigenvalue of 1.

   Make sure you normalise $\boldsymbol{\mu}$, such that $\sum_{j \in S} \mu_j = 1$.

Task 3. Write a function which, given the transition kernel $\mathbf{K}$, computes the invariant distribution $\boldsymbol{\mu}$.

Try out your function for the phone example from the lectures (or any other example of your choice).

Task 4. Create a plot which illustrates that the distribution of X_t approaches the invariant distribution $\boldsymbol{\mu}$, as t becomes large. The plot could look similar the one below ...



Sampling from a Markov chain

To draw one sample path $(X_0(\omega), \ldots, X_\tau(\omega))$ of a Markov chain we have to first of all draw X_0 from the initial distribution $\boldsymbol{\lambda}_0$. Then for $t = 1, \ldots, \tau$ we have to draw $X_t | X_{t-1} = x_{t-1}$, i.e. draw X_t from the distribution given by the x_{t-1}-th row of $\mathbf{K}$.

Task 5. Write a function which, given the transition matrix $\mathbf{K}$, initial distribution $\boldsymbol{\lambda}_0$, and $\tau \in \mathbb{N}$ returns one random realisation of $(X_0(\omega), \ldots, X_\tau(\omega))$.

Plot your randomly drawn sample path for a simple example (e.g. phone line example).

Task 6. "Verify" the ergodic theorem (theorem 1.30) numerically by drawing one sample path from the Markov chain used in the phone example. Then compare the proportion of 0's and 1's to the probabilities given by the invariant distribution.

Useful Python functions

The functions listed below are available if you have imported the following libraries:

```
from scipy import *
from scipy import linalg
from numpy import random
import pylab
```

The following Python functions might be of use:

Matrix multiplication Two arrays are multiplied using the function `dot`. If `A` is a matrix (or array), then `transpose(A)` or `A.T` is its transpose.

`linalg.eig` computes the eigenvalues and eigenvectors of a matrix (or array).

```
A = array([[0.7, 0.2], [0.3, 0.8]])
linalg.eig(A)
```

```
(array([ 0.5+0.j,  1.0+0.j]), array([[-0.70710678, -0.5547002 ],
       [ 0.70710678, -0.83205029]]))
```

The eigenvalues are not necessarily ordered by their magnitude. The eigenvectors are the columns of the returned matrix of eigenvectors.

`linalg.solve` solves a system of of linear equations, provided the coefficient matrix is square and of full rank.

```
A = array([[0.7, 0.2], [1, 0]])
b = [1, 1]
linalg.solve(A, b)
```

```
array([ 1. ,  1.5])
```

`linalg.lstsq` solves a possibly over-determined system of linear equations using least squares. It thus also works for non-square matrices.

```
A = array([[0.7, 0.2], [-0.7, -0.2], [1, 0]])
b = [1, -1, 1]
linalg.lstsq(A, b)
```

```
(array([ 1. ,  1.5]), 1.4616612873612731e-31, 2, array([ 1.42140915,  0.19898754]))
```

The first entry is the solution to the least-squares problem.

Sampling from a discrete distribution The following function

```
def discrete_sample(probs):
    return sum(cumsum(probs)<random.random_sample())
```

draws one realisation from an arbitrary discrete distribution on $\{0, \ldots, n-1\}$ with probabilities given by `probs` (vector of length n).

```
p = [0.7, 0.2, 0.1]
discrete_sample(p)
```

```
1
```

Useful R functions

The following R functions might be of use:

Matrix multiplication Two matrices are multiplied using the operator `%*%`. If `A` is a matrix (or array), then `t(A)` is its transpose.

`eigen` computes the eigenvalues and eigenvectors of a matrix (or array).

```
A <- rbind(c(0.7, 0.2), c(0.3, 0.8))
eigen(A)
```

```
$values
[1] 1.0 0.5

$vectors
           [,1]       [,2]
[1,] -0.5547002 -0.7071068
[2,] -0.8320503  0.7071068
```

The eigenvalues are ordered in decreasing order. The eigenvectors are the columns of the returned matrix of eigenvectors.

`solve` solves a system of of linear equations, provided the coefficient matrix is square and of full rank.

```
A <- rbind(c(0.7, 0.2), c(1, 0))
b <- c(1, 1)
solve(A, b)
```

```
[1] 1.0 1.5
```

`qr.solve` solves a possibly over-determined system of linear equations using least squares. It thus also works for non-square matrices.

```
A <- rbind(c(0.7, 0.2), c(-0.7, -0.2), c(1, 0))
b <- c(1, -1, 1)
qr.solve(A, b)
```

```
[1] 1.0 1.5
```

`sample` The function `sample(n, 1, prob=prob)` draws one realisation from the discrete distribution on $\{1, \ldots, n\}$ given by the vector `probs` (of length `n`).

```
p <- c(0.7, 0.2, 0.1)
sample(3, 1, prob=p)
```

```
[1] 1
```