The markovchain Package: A Package for Easily Handling Discrete Markov Chains in R

Giorgio Alfredo Spedicato, Tae Seung Kang, Sai Bhargav Yalamanchi, Deepak Yadav, Ignacio Cordón

Abstract

The **markovchain** package aims to fill a gap within the R framework providing S4 classes and methods for easily handling discrete time Markov chains, homogeneous and simple inhomogeneous ones as well as continuous time Markov chains. The S4 classes for handling and analysing discrete and continuous time Markov chains are presented, as well as functions and method for performing probabilistic and statistical analysis. Finally, some examples in which the package's functions are applied to Economics, Finance and Natural Sciences topics are shown.

Keywords: discrete time Markov chains, continuous time Markov chains, transition matrices, communicating classes, periodicity, first passage time, stationary distributions.

1. Introduction

Markov chains represent a class of stochastic processes of great interest for the wide spectrum of practical applications. In particular, discrete time Markov chains (DTMC) make it possible to model the transition probabilities between discrete states by means of matrices. Various R packages deal with models that are based on Markov chains:

- **msm** (Jackson 2011) handles multi-state models for panel data.
- **mcmcR** (Geyer and Johnson 2013) implements Markov chain Monte Carlo methods.
- **HMM** (Himmelmann and www.linhi.com 2010) fits hidden Markov models with covariates.
- **mstate** (de Wreede, Fiocco, and Putter 2011) fits multi-state models based on Markov chains for survival analysis.

Nevertheless, the R statistical environment (R Core Team 2013) seems to lack a simple package that coherently defines S4 classes for discrete Markov chains and allows one to perform probabilistic analysis, statistical inference and applications. **DTMCPack** (Nicholson 2013) was the first package specifically dedicated to DTMC analysis and **markovchain** (Spedicato 2017) the second. The latter aims to offer more flexibility in handling DTMC than other existing solutions: it provides S4 classes for both homogeneous and semi-homogeneous Markov chains, together with methods to perform probabilistic and statistical analysis, to simulate and to aggregate chains, and to import and export them in several formats.

The **markovchain** package relies on the following R packages: **Matrix** to store large transition matrices in sparse form; **expm** (Goulet, Dutang, Maechler, Firth, Shapira, Stadelmann,

and expm-developers@lists.R-forge.R-project.org 2013) to compute matrix powers efficiently; **igraph** (Csardi and Nepusz 2006) to plot `markovchain` objects; and **Rcpp** (Eddelbuettel 2013) together with **RcppArmadillo** and **RcppParallel** to run the most demanding computations in compiled C++ code, optionally on several threads. Other scientific software also provides functions specifically designed to analyze DTMC, such as Mathematica 9 (Wolfram Research 2013a).

The paper is structured as follows. Section 2 briefly reviews the mathematics and the definitions regarding DTMC. Section 3 shows how to create and manage Markov chain objects within the package. Section 4 and Section 5 show how to perform probabilistic and statistical analysis: simulation, aggregation of states, serialization, estimation, prediction, statistical tests and Bayesian estimation. Section 6 is devoted to continuous time Markov chains, and Section 7 presents applied examples from various fields, analyzed by means of the **markovchain** package. Section 8 concludes. Higher order and multivariate Markov chains, and the functions added during the Google Summer of Code 2017, are described in two companion vignettes ("Higher order Markov chains" and "Google Summer of Code 2017 Additions").

2. Review of core mathematical concepts

2.1. General Definitions

A DTMC is a sequence of random variables X_1, $X_2, \ldots, X_n, \ldots$ characterized by the Markov property (also known as memoryless property, see Equation 1). The Markov property states that the distribution of the forthcoming state X_{n+1} depends only on the current state X_n and doesn't depend on the previous ones X_{n-1}, $X_{n-2}, \ldots,$ X_1.

$$Pr\left(X_{n+1} = x_{n+1} \,|X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n\right) = Pr\left(X_{n+1} = x_{n+1} \,|X_n = x_n\right). \quad (1)$$

The set of possible states $S = \{s_1, s_2, ..., s_r\}$ of X_n can be finite or countable and it is named the state space of the chain.

The chain moves from one state to another (this change is named either 'transition' or 'step') and the probability p_{ij} to move from state s_i to state s_j in one step is named transition probability:

$$p_{ij} = Pr\left(X_1 = s_j \,|X_0 = s_i\right). \quad (2)$$

The probability of moving from state i to j in n steps is denoted by

$$p_{ij}^{(n)} = Pr\left(X_n = s_j \,|X_0 = s_i\right).$$

A DTMC is called time-homogeneous if the property shown in Equation 3 holds. Time homogeneity implies no change in the underlying transition probabilities as time goes on.

$$Pr\left(X_{n+1} = s_j \,|X_n = s_i\right) = Pr\left(X_n = s_j \,|X_{n-1} = s_i\right). \quad (3)$$

If the Markov chain is time-homogeneous, then $p_{ij} = Pr\left(X_{k+1} = s_j \,|X_k = s_i\right)$ and $p_{ij}^{(n)} = Pr\left(X_{n+k} = s_j \,|X_k = s_i\right)$, where $k > 0$.

The probability distribution of transitions from one state to another can be represented into a transition matrix $P = (p_{ij})_{i,j}$, where each element of position (i, j) represents the transition probability p_{ij}. E.g., if $r = 3$ the transition matrix P is shown in Equation 4

$$P = \begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{bmatrix}. \tag{4}$$

The distribution over the states can be written in the form of a stochastic row vector x (the term stochastic means that $\sum_i x_i = 1, x_i \geq 0$): e.g., if the current state of x is s_2, $x = (0\,1\,0)$. As a consequence, the relation between $x^{(1)}$ and $x^{(0)}$ is $x^{(1)} = x^{(0)}P$ and, recursively, we get $x^{(2)} = x^{(0)}P^2$ and $x^{(n)} = x^{(0)}P^n$, $n > 0$.

DTMC are explained in most theory books on stochastic processes, see Brémaud (1999) and Dobrow (2016) for example. Valuable references online available are: Konstantopoulos (2009), Snell (1999) and Bard (2000).

2.2. Properties and classification of states

A state s_j is said accessible from state s_i (written $s_i \to s_j$) if a system starting in state s_i has a positive probability to reach the state s_j at a certain point, i.e., $\exists n > 0 : p_{ij}^{(n)} > 0$. If both $s_i \to s_j$ and $s_j \to s_i$, then s_i and s_j are said to communicate.

A communicating class is defined to be a set of states that communicate. A DTMC can be composed of one or more communicating classes. If the DTMC is composed of only one communicating class (i.e., if all states in the chain communicate), then it is said to be irreducible. A communicating class is said to be closed if no states outside of the class can be reached from any state inside it.

If $p_{ii} = 1$, s_i is defined as absorbing state: an absorbing state corresponds to a closed communicating class composed by one state only.

The canonical form of a DTMC transition matrix is a matrix having a block form, where the closed communicating classes are shown at the beginning of the diagonal matrix.

A state s_i has period k_i if any return to state s_i must occur in multiples of k_i steps, that is $k_i = gcd\{n : Pr(X_n = s_i \,|X_0 = s_i) > 0\}$, where gcd is the greatest common divisor. If $k_i = 1$ the state s_i is said to be aperiodic, else if $k_i > 1$ the state s_i is periodic with period k_i. Loosely speaking, s_i is periodic if it can only return to itself after a fixed number of transitions $k_i > 1$ (or multiple of k_i), else it is aperiodic.

If states s_i and s_j belong to the same communicating class, then they have the same period k_i. As a consequence, each of the states of an irreducible DTMC share the same periodicity. This periodicity is also considered the DTMC periodicity. States can also be classified according to how often the chain returns to them. Let $T^{x \to x}$ be the number of steps needed to go back to state x, knowing that the chain starts in x.

- A state x is recurrent if $P(T^{x \to x} < +\infty) = 1$ (equivalently $P(T^{x \to x} = +\infty) = 0$). In addition:
  1. A state x is null recurrent if in addition $E(T^{x \to x}) = +\infty$.
  2. A state x is positive recurrent if in addition $E(T^{x \to x}) < +\infty$.

3. A state x is absorbing if in addition $P(T^{x\to x}=1)=1$.

- A state x is transient if $P(T^{x\to x}<+\infty)<1$ (equivalently $P(T^{x\to x}=+\infty)>0$).

It is possible to analyze the timing to reach a certain state. The first passage time (or hitting time) from state s_i to state s_j is the number T_{ij} of steps taken by the chain until it arrives for the first time to state s_j, given that $X_0=s_i$. The probability distribution of T_{ij} is defined by Equation 5

$$h_{ij}{}^{(n)}=Pr\left(T_{ij}=n\right)=Pr\left(X_n=s_j,X_{n-1}\neq s_j,\ldots,X_1\neq s_j|X_0=s_i\right) \tag{5}$$

and can be found recursively using Equation 6, given that $h_{ij}{}^{(n)}=p_{ij}$.

$$h_{ij}{}^{(n)}=\sum_{k\in S-\{s_j\}}p_{ik}h_{kj}{}^{(n-1)}. \tag{6}$$

A commonly used quantity related to h is its average value, i.e. the *mean first passage time* (also expected hitting time), namely $\bar{h}_{ij}=\sum_{n=1}^{\infty}n\,h_{ij}^{(n)}$.

If in the definition of the first passage time we let $s_i=s_j$, we obtain the first recurrence time $T_i=\inf\{n\geq 1:X_n=s_i|X_0=s_i\}$. The *mean recurrence time* of state s_i is the expected value of T_i:

$$r_i=\sum_{k=1}^{\infty}k\cdot P(T_i=k).$$

Revisiting the definition of recurrence and transience: a state s_i is said to be recurrent if it is visited infinitely often, i.e., $Pr(T_i<+\infty|X_0=s_i)=1$. On the opposite, s_i is called transient if there is a positive probability that the chain will never return to s_i, i.e., $Pr(T_i=+\infty|X_0=s_i)>0$.

Given a time homogeneous Markov chain with transition matrix P, a stationary distribution z is a stochastic row vector such that $z=z\cdot P$, where $0\leq z_j\leq 1\,\forall j$ and $\sum_j z_j=1$.

If a DTMC $\{X_n\}$ is irreducible and aperiodic, then it has a limit distribution and this distribution is stationary. As a consequence, if P is the $k\times k$ transition matrix of the chain and $z=(z_1,...,z_k)$ is the unique eigenvector of P such that $\sum_{i=1}^{k}z_i=1$, then we get

$$\lim_{n\to\infty}P^n=Z, \tag{7}$$

where Z is the matrix having all rows equal to z. The stationary distribution of $\{X_n\}$ is represented by z.

A matrix A is called positive if all of its entries are strictly positive, and we write $A>0$. If some power of the transition matrix P of a DTMC is positive, i.e. there exists $m>0$ such that $P^m>0$, then P is called primitive and the DTMC is said to be regular. For a finite state space, being regular is equivalent to being irreducible and aperiodic. All regular DTMCs are irreducible, but the converse is not true.

Given two disjoint sets of states A and B, the *committor probability* $q_j^{(AB)}$ is the probability that a chain starting in state s_j reaches A before B (Noé, Schütte, Vanden-Eijnden, Reich, and Weikl 2009). For $j\notin A\cup B$ it solves

$$q_j^{(AB)} = \sum_{k \in S} P_{jk} q_k^{(AB)} \quad \text{with} \quad q_k^{(AB)} = 1 \text{ for } k \in A \quad \text{and} \quad q_k^{(AB)} = 0 \text{ for } k \in B. \tag{8}$$

The *hitting probability* from i to j is the probability of ever reaching the state j if the initial state is i:

$$h_{i,j} = Pr(T_{ij} < \infty) = \sum_{n=0}^{\infty} h_{ij}^{(n)} \tag{9}$$

In a DTMC with a finite set of states, every transient state can reach at least one recurrent state. If the chain starts in a transient state, once it hits a recurrent state it is caught in the corresponding recurrent class and never goes back to the initial state. Given a transient state i, the *absorption probability* f_{ij}^* into the recurrent state j is the probability that j is the first recurrent state visited by the chain (which is then absorbed by the recurrent class of j). The *mean absorption time* b_i is the mean number of steps the chain takes, starting from the transient state i, until it hits any recurrent state..

2.3. A short example

Consider the following numerical example. Suppose we have a DTMC with a set of 3 possible states $S = \{s_1, s_2, s_3\}$. Let the transition matrix be:

$$P = \begin{bmatrix} 0.5 & 0.2 & 0.3 \\ 0.15 & 0.45 & 0.4 \\ 0.25 & 0.35 & 0.4 \end{bmatrix}. \tag{10}$$

In P, $p_{11} = 0.5$ is the probability that $X_1 = s_1$ given that $X_0 = s_1$, and so on. It is easy to see that the chain is irreducible since all the states communicate (it is made by one communicating class only).

Suppose that the current state of the chain is $X_0 = s_2$, i.e., $x^{(0)} = (0\,1\,0)$, then the probability distribution of states after 1 and 2 steps can be computed as shown in Equations (11) and (12).

$$x^{(1)} = (0\,1\,0) \begin{bmatrix} 0.5 & 0.2 & 0.3 \\ 0.15 & 0.45 & 0.4 \\ 0.25 & 0.35 & 0.4 \end{bmatrix} = (0.15\,0.45\,0.4)\,. \tag{11}$$

$$x^{(n)} = x^{(n-1)} P \to (0.15\,0.45\,0.4) \begin{bmatrix} 0.5 & 0.2 & 0.3 \\ 0.15 & 0.45 & 0.4 \\ 0.25 & 0.35 & 0.4 \end{bmatrix} = (0.2425\,0.3725\,0.385)\,. \tag{12}$$

If we were interested in the probability of being in the state s_3 in the second step, then $Pr\,(X_2 = s_3\,|X_0 = s_2\,) = 0.385$.

3. The structure of the package

3.1. Creating markovchain objects

The package is loaded within the R command line as follows:

```
R> library("markovchain")
```

The `markovchain` and `markovchainList` S4 classes (Chambers 2008) are defined within the **markovchain** package as displayed:

```
Class "markovchain" [package "markovchain"]

Slots:

Name:            states            byrow transitionMatrix             name
Class:        character           logical            matrix        character

Class "markovchainList" [package "markovchain"]

Slots:

Name:  markovchains         name
Class:         list    character
```

The first class has been designed to handle homogeneous Markov chain processes, while the latter (which is itself a list of `markovchain` objects) has been designed to handle semi-homogeneous Markov chains processes.

Every `markovchain` object has the following slots:

1. `states`: a character vector, listing the states for which transition probabilities are defined.
2. `byrow`: a logical element, indicating whether transition probabilities are shown by row or by column.
3. `transitionMatrix`: the probabilities of the transition matrix.
4. `name`: optional character element to name the DTMC.

`markovchainList` objects have the following slots:

1. `markovchains`: a list of `markovchain` objects.
2. `name`: optional character element to name the DTMC.

A `markovchain` object can be created either in the long way, as the following code shows,

```
R> weatherStates <- c("sunny", "cloudy", "rain")
R> byRow <- TRUE
R> weatherMatrix <- matrix(data = c(0.70, 0.2, 0.1,
+                         0.3, 0.4, 0.3,
+                         0.2, 0.45, 0.35), byrow = byRow, nrow = 3,
+                       dimnames = list(weatherStates, weatherStates))
R> mcWeather <- new("markovchain", states = weatherStates, byrow = byRow,
+                 transitionMatrix = weatherMatrix, name = "Weather")
```

or in a shorter way, as displayed below

```
R> mcWeather <- new("markovchain", states = c("sunny", "cloudy", "rain"),
+                   transitionMatrix = matrix(data = c(0.70, 0.2, 0.1,
+                         0.3, 0.4, 0.3,
+                         0.2, 0.45, 0.35), byrow = byRow, nrow = 3),
+                   name = "Weather")
```

When `new("markovchain")` is called alone, a default Markov chain is created.

```
R> defaultMc <- new("markovchain")
```

The shorter way is made possible by the `initialize` S4 method, which checks that:

- the `transitionMatrix`, of class `matrix` or `Matrix`, is a transition matrix, i.e., all its entries are probabilities and either all rows or all columns sum up to one;
- the row and column names of `transitionMatrix`, if present, coincide with the `states` slot; if they are missing, they are taken from `states`.

`markovchain` objects can be collected in a `markovchainList` S4 object, as the following example shows.

```
R> mcList <- new("markovchainList", markovchains = list(mcWeather, defaultMc),
+                  name = "A list of Markov chains")
```

3.2. Handling markovchain objects

Table 1 lists which methods handle and manipulate `markovchain` objects.

The examples that follow show how operations on `markovchain` objects can be easily performed. For instance, using the previously defined chain we can find the probability distribution of the weather states in two and in seven days, given that today is cloudy.

```
R> initialState <- c(0, 1, 0)
R> after2Days <- initialState * (mcWeather * mcWeather)
R> after7Days <- initialState * (mcWeather ^ 7)
R> after2Days
```

Method	Purpose
`*`	Direct multiplication for transition matrices.
`^`	Compute the power `markovchain` of a given one.
`[`	Direct access to the elements of the transition matrix.
`==`	Equality operator between two transition matrices.
`!=`	Inequality operator between two transition matrices.
`as`	Operator to convert `markovchain` objects into `data.frame` and `table` object.
`dim`	Dimension of the transition matrix.
`names`	Equal to `states`.
`names<-`	Change the `states` name.
`name`	Get the name of `markovchain object`.
`name<-`	Change the name of `markovchain object`.
`plot`	`plot` method for `markovchain` objects.
`print`	`print` method for `markovchain` objects.
`show`	`show` method for `markovchain` objects.
`sort`	`sort` method for `markovchain` objects, in terms of their states.
`states`	Name of the transition states.
`t`	Transposition operator (which switches `byrow` 'slot value and modifies the transition matrix coherently).

Table 1: **markovchain** methods for handling `markovchain` objects.

```
     sunny cloudy  rain
[1,]  0.39  0.355 0.255
```

```
R> round(after7Days, 3)
```

```
     sunny cloudy  rain
[1,] 0.462  0.319 0.219
```

The same answer can be obtained by defining the vector of probabilities as a column vector. A column-defined transition matrix can be set up either by creating a new matrix or by transposing an existing `markovchain` object with the `t` method.

```
R> initialState <- c(0, 1, 0)
R> after2Days <- (t(mcWeather) * t(mcWeather)) * initialState
R> after7Days <- (t(mcWeather) ^ 7) * initialState
R> after2Days
```

```
        [,1]
sunny  0.390
cloudy 0.355
rain   0.255
```

```
R> round(after7Days, 3)
```

```
        [,1]
sunny  0.462
cloudy 0.319
rain   0.219
```

The initial state vector does not need to be a probability vector, as the code that follows shows, where it holds the number of individuals in each state:

```
R> fvals<-function(mchain,initialstate,n) {
+   out<-data.frame()
+   names(initialstate)<-names(mchain)
+   for (i in 0:n)
+   {
+     iteration<-initialstate*mchain^(i)
+     out<-rbind(out,iteration)
+   }
+   out<-cbind(out, i=seq(0,n))
+   out<-out[,c(4,1:3)]
+   return(out)
+ }
R> fvals(mchain=mcWeather,initialstate=c(90,5,5),n=4)

  i    sunny   cloudy     rain
1 0 90.00000  5.00000  5.00000
2 1 65.50000 22.25000 12.25000
3 2 54.97500 27.51250 17.51250
4 3 50.23875 29.88063 19.88062
5 4 48.10744 30.94628 20.94628
```

Basic methods return the states and the dimension of the transition matrix.

```
R> states(mcWeather)

[1] "sunny"  "cloudy" "rain"

R> names(mcWeather)

[1] "sunny"  "cloudy" "rain"

R> dim(mcWeather)

[1] 3
```

The name of a `markovchain` object can be read and changed with `name` and `name<-`.

```
R> name(mcWeather)
```

```
[1] "Weather"
```

```
R> name(mcWeather) <- "New Name"
R> name(mcWeather)
```

```
[1] "New Name"
```

The states, and the transition matrix accordingly, can be sorted alphabetically:

```
R> markovchain:::sort(mcWeather)
```

```
New Name
 A  3 - dimensional discrete Markov Chain defined by the following states:
 cloudy, rain, sunny
 The transition matrix  (by rows)  is defined as follows:
       cloudy rain sunny
cloudy   0.40 0.30   0.3
rain     0.45 0.35   0.2
sunny    0.20 0.10   0.7
```

Transition probabilities can be accessed directly through the `transitionProbability` method and the `"["` method.

```
R> transitionProbability(mcWeather, "cloudy", "rain")
```

```
[1] 0.3
```

```
R> mcWeather[2,3]
```

```
[1] 0.3
```

The transition matrix of a `markovchain` object can be displayed with the `print` or `show` methods (the latter being less verbose). The transition diagram can be plotted with the `plot` method (Figure 1), a wrapper of `plot.igraph` from the **igraph** package (Csardi and Nepusz 2006); additional parameters can be passed to `plot` to control the layout of the graph. The **diagram** and **DiagrammeR** packages can be used instead (Figure 2). The `plot` method uses `communicatingClasses` to separate the states of different communicating classes, and all the states of a class have the same color. The graph can also be exported as text for Graphviz or Mermaid, as shown in Section 5.3.

```
R> print(mcWeather)
```

```
       sunny cloudy rain
sunny    0.7   0.20 0.10
cloudy   0.3   0.40 0.30
rain     0.2   0.45 0.35
```

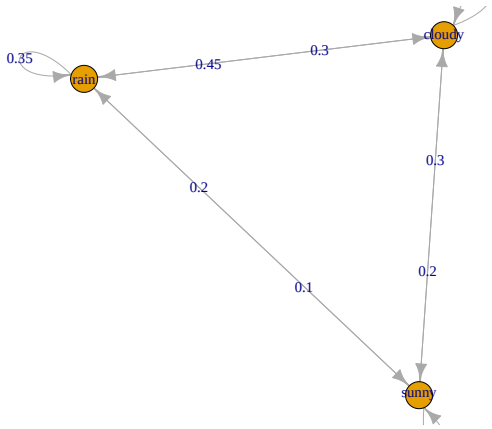


Figure 1: Weather example. Markov chain plot

```
R> show(mcWeather)

New Name
 A  3 - dimensional discrete Markov Chain defined by the following states:
 sunny, cloudy, rain
 The transition matrix  (by rows)  is defined as follows:
       sunny cloudy rain
sunny    0.7   0.20 0.10
cloudy   0.3   0.40 0.30
rain     0.2   0.45 0.35


Caricamento pacchetto: 'igraph'

I seguenti oggetti sono mascherati da 'package:stats':

    decompose, spectrum

Il seguente oggetto è mascherato da 'package:base':

    union


Caricamento del pacchetto richiesto: shape
```

Coercion from and to some other classes is possible, as shown in Figure 3 and in the following code.

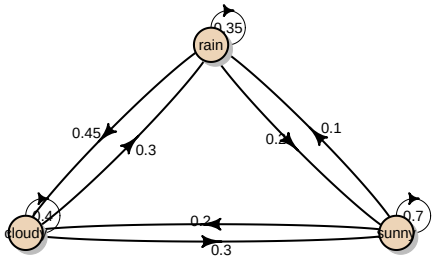


Figure 2: Weather example. Markov chain plot with diagram

```
R> mcDf <- as(mcWeather, "data.frame")
R> mcNew <- as(mcDf, "markovchain")
R> mcDf
```

```
      t0     t1 prob
1  sunny  sunny 0.70
2  sunny cloudy 0.20
3  sunny   rain 0.10
4 cloudy  sunny 0.30
5 cloudy cloudy 0.40
6 cloudy   rain 0.30
7   rain  sunny 0.20
8   rain cloudy 0.45
9   rain   rain 0.35
```

```
R> mcIgraph <- as(mcWeather, "igraph")
```

```
R> if (requireNamespace("msm", quietly = TRUE)) {
+ require(msm)
+ Q <- rbind ( c(0, 0.25, 0, 0.25),
+              c(0.166, 0, 0.166, 0.166),
+              c(0, 0.25, 0, 0.25),
+              c(0, 0, 0, 0) )
+ cavmsm <- msm(state ~ years, subject = PTNUM, data = cav, qmatrix = Q, death = 4)
+ msmMc <- as(cavmsm, "markovchain")
+ msmMc
+   } else {
```

```
+   message("msm unavailable")
+   }

Caricamento del pacchetto richiesto: msm

Unnamed Markov chain
 A  4 - dimensional discrete Markov Chain defined by the following states:
 State 1, State 2, State 3, State 4
 The transition matrix  (by rows)  is defined as follows:
           State 1    State 2    State 3    State 4
State 1 0.853958721 0.08836953 0.01475543 0.04291632
State 2 0.155576908 0.56663284 0.20599563 0.07179462
State 3 0.009903994 0.07853691 0.65965727 0.25190183
State 4 0.000000000 0.00000000 0.00000000 1.00000000
```

The following code coerces an object of the **etm** package into a `markovchain` (**etm** was archived from CRAN in September 2020, so the example runs only if the package is installed):

```
R> if (requireNamespace("etm", quietly = TRUE)) {
+ library(etm)
+ data(sir.cont)
+ sir.cont <- sir.cont[order(sir.cont$id, sir.cont$time), ]
+ for (i in 2:nrow(sir.cont)) {
+   if (sir.cont$id[i]==sir.cont$id[i-1]) {
+     if (sir.cont$time[i]==sir.cont$time[i-1]) {
+       sir.cont$time[i-1] <- sir.cont$time[i-1] - 0.5
+     }
+   }
+ }
+ tra <- matrix(ncol=3,nrow=3,FALSE)
+ tra[1, 2:3] <- TRUE
+ tra[2, c(1, 3)] <- TRUE
+ tr.prob <- etm::etm(sir.cont, c("0", "1", "2"), tra, "cens", 1)
+ tr.prob
+ etm2mc<-as(tr.prob, "markovchain")
+ etm2mc
+   } else {
+   message("etm unavailable")
+ }

Unnamed Markov chain
 A  3 - dimensional discrete Markov Chain defined by the following states:
 0, 1, 2
 The transition matrix  (by rows)  is defined as follows:
          0         1         2
0 0.0000000 0.5000000 0.5000000
1 0.5000000 0.0000000 0.5000000
2 0.3333333 0.3333333 0.3333333
```

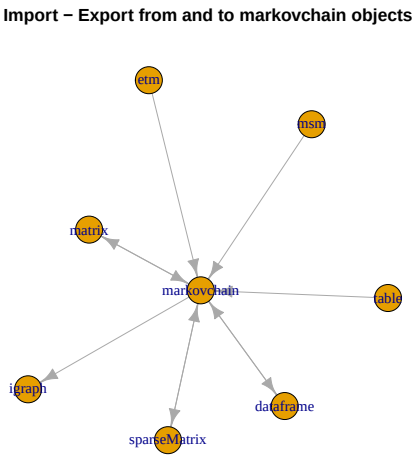


Figure 3: The markovchain methods for import and export

A square probability matrix can also be coerced into a `markovchain` object, which is another way to create one, as the code below shows.

```
R> myMatr<-matrix(c(.1,.8,.1,.2,.6,.2,.3,.4,.3), byrow=TRUE, ncol=3)
R> myMc<-as(myMatr, "markovchain")
R> myMc

Unnamed Markov chain
 A  3 - dimensional discrete Markov Chain defined by the following states:
 s1, s2, s3
 The transition matrix  (by rows)  is defined as follows:
    s1  s2  s3
s1 0.1 0.8 0.1
s2 0.2 0.6 0.2
s3 0.3 0.4 0.3
```

Semi-homogeneous Markov chains are handled by `markovchainList` objects. The example that follows arises from health insurance, where the costs associated with the patients of a Continuous Care Health Community (CCHC) are modeled by a semi-homogeneous Markov chain, since the transition probabilities change every year. The methods explicitly written for `markovchainList` objects are `print`, `show`, `dim` and `[`.

```
R> stateNames = c("H", "I", "D")
R> Q0 <- new("markovchain", states = stateNames,
+         transitionMatrix =matrix(c(0.7, 0.2, 0.1,0.1, 0.6, 0.3,0, 0, 1),
+         byrow = TRUE, nrow = 3), name = "state t0")
R> Q1 <- new("markovchain", states = stateNames,
```

```
+         transitionMatrix = matrix(c(0.5, 0.3, 0.2,0, 0.4, 0.6,0, 0, 1),
+         byrow = TRUE, nrow = 3), name = "state t1")
R> Q2 <- new("markovchain", states = stateNames,
+         transitionMatrix = matrix(c(0.3, 0.2, 0.5,0, 0.2, 0.8,0, 0, 1),
+         byrow = TRUE,nrow = 3), name = "state t2")
R> Q3 <- new("markovchain", states = stateNames,
+           transitionMatrix = matrix(c(0, 0, 1, 0, 0, 1, 0, 0, 1),
+         byrow = TRUE, nrow = 3), name = "state t3")
R> mcCCRC <- new("markovchainList",markovchains = list(Q0,Q1,Q2,Q3),
+       name = "Continuous Care Health Community")
R> print(mcCCRC)

Continuous Care Health Community  list of Markov chain(s)
Markovchain  1
state t0
 A  3 - dimensional discrete Markov Chain defined by the following states:
 H, I, D
 The transition matrix  (by rows)  is defined as follows:
    H   I   D
H 0.7 0.2 0.1
I 0.1 0.6 0.3
D 0.0 0.0 1.0

Markovchain  2
state t1
 A  3 - dimensional discrete Markov Chain defined by the following states:
 H, I, D
 The transition matrix  (by rows)  is defined as follows:
    H   I   D
H 0.5 0.3 0.2
I 0.0 0.4 0.6
D 0.0 0.0 1.0

Markovchain  3
state t2
 A  3 - dimensional discrete Markov Chain defined by the following states:
 H, I, D
 The transition matrix  (by rows)  is defined as follows:
    H   I   D
H 0.3 0.2 0.5
I 0.0 0.2 0.8
D 0.0 0.0 1.0

Markovchain  4
state t3
 A  3 - dimensional discrete Markov Chain defined by the following states:
 H, I, D
```

```
 The transition matrix  (by rows)  is defined as follows:
  H I D
H 0 0 1
I 0 0 1
D 0 0 1
```

The elements of a `markovchainList` can be accessed directly, and `dim` returns the number of `markovchain` objects it contains.

```
R> mcCCRC[[1]]

state t0
 A  3 - dimensional discrete Markov Chain defined by the following states:
 H, I, D
 The transition matrix  (by rows)  is defined as follows:
    H   I   D
H 0.7 0.2 0.1
I 0.1 0.6 0.3
D 0.0 0.0 1.0

R> dim(mcCCRC)

[1] 4
```

The package contains some data found in the literature on DTMC models (see Section 7). Table 2 lists the datasets included in the current release.

Dataset	Description
`blanden`	Mobility across income quartiles, Jo Blanden and Machin (2005).
`craigsendi`	CD4 cells, B. A. Craig and A. A. Sendi (2002).
`kullback`	raw transition matrices for testing homogeneity, Kullback, Kupperman, and Ku (1962).
`preproglucacon`	Preproglucacon DNA basis, P. J. Avery and D. A. Henderson (1999).
`rain`	Alofi Island rains, P. J. Avery and D. A. Henderson (1999).
`holson`	Individual states trajectories.
`sales`	Sales of six beverages in Hong Kong, Ching, Ng, and Fung (2008).
`tm_abs`	Frequencies of the 2000 corporate credit rating transitions of Standard and Poor's, from ESMA.

Table 2: Datasets included in the **markovchain** package.

Finally, Table 3 lists the demos included in the `demo` directory of the package.

4. Probability with markovchain objects

The **markovchain** package contains functions to analyse DTMC from a probabilistic perspective. For example, the package provides methods to find stationary distributions, to identify

R Code File	Description
`quickStart.R`	Quick start for the package.
`examples.R`	Notable Markov chains, e.g., the gambler's ruin chain.
`bard.R`	Structural analysis of Markov chains from Bard PPT.
`extractMatrices.R`	Canonical form and fundamental matrix of a given transition matrix.
`mathematica.R`	Markov chains examples from Mathematica.
`reliability.R`	Simulation of reliability in inverting a Markov chain.
`quasiBayesian.R`	Quasi-Bayesian estimator function.
`computationTime.R`	Computation time by size of the character sequence.

Table 3: The **markovchain** demos.

absorbing and transient states, to compute passage and absorption times, and to study how fast a chain approaches its stationary distribution. Many of these methods come from MATLAB listings that have been ported into R: the interested reader can consult the original listings, Feres (2007) and Montgomery (2009), for a full description of the underlying theory and algorithms.

Table 4 shows the methods that can be applied to `markovchain` objects to perform probabilistic analysis.

4.1. Conditional distributions

The conditional distribution of the weather states, given that today is sunny, is computed by the following code.

```
R> conditionalDistribution(mcWeather, "sunny")
```

```
 sunny cloudy   rain
   0.7    0.2    0.1
```

4.2. Stationary states

A stationary (steady state, or equilibrium) vector is a probability vector such that Equation 13 holds

$$\begin{array}{c} 0 \leq \pi_j \leq 1 \\ \sum_{j \in S} \pi_j = 1 \\ \pi \cdot P = \pi \end{array} \tag{13}$$

Steady states are associated with the eigenvalues of P equal to one. One could be tempted to compute them by solving the eigenproblem of the matrix, taking the real parts of the eigenvectors (if $u+iv$ is an eigenvector of P for the eigenvalue one, then so are $u = Re(u+iv)$ and $v = Im(u+iv)$) and normalizing them by their sum. This approach, however, raises some concerns:

1. If $u, v \in \mathbb{R}^n$ are linearly independent eigenvectors associated with the eigenvalue 1, $u+iv$ and $u-iv$ are also linearly independent eigenvectors, and their real parts coincide. Taking real parts, we would lose an eigenvector, because we cannot know in advance whether the algorithm that computes the eigenvectors will return something similar to what we described. We should be agnostic to the underlying eigenvector computation algorithm.

2. Imagine the identity P of dimensions 2×2. Its eigenvectors associated to the 1 eigen value are $u = (1, 0)$ and $v = (0, 1)$. However, the underlying algorithm to compute eigenvectors could return $(1, -2)$ and $(-2, 1)$ instead, that are linear combinations of the aforementioned ones, and therefore eigenvectors. Normalizing by their sum, we would get: $(-1, 2)$ and $(2, -1)$, which obviously are not probability measures. Again, we should be agnostic to the underlying eigen computation algorithm.

3. Algorithms to compute eigenvalues and eigenvectors are computationally expensive: they are iterative, and we cannot predict a fixed number of iterations for them. Moreover, each iteration takes $\mathcal{O}(m^2)$ or $\mathcal{O}(m^3)$ algorithmic complexity, with m the number of states.

We use the fact that every irreducible DTMC has a unique steady state: if M is the transition matrix of an irreducible DTMC (all states communicate with each other), then there exists a unique $v \in \mathbb{R}^m$ such that

$$v \cdot M = v, \qquad \sum_{i=1}^{m} v_i = 1$$

We also use the fact that a steady state of a DTMC assigns 0 to the transient states. The canonical form of a row-stochastic matrix looks as follows:

$$\left(\begin{array}{c|c|c|c|c} M_1 & 0 & 0 & \dots & 0 \\ \hline 0 & M_2 & 0 & \dots & 0 \\ \hline 0 & 0 & M_3 & \dots & 0 \\ \hline \vdots & \vdots & \vdots & \ddots & \vdots \\ \hline A_1 & A_2 & A_3 & \dots & R \end{array}\right)$$

where M_i corresponds to irreducible sub-chains, the blocks A_i correspond to the transitions from transient states to each of the recurrent classes and R are the transitions from the transient states to themselves.

Moreover, a Markov chain has exactly as many steady states as recurrent classes. Therefore, we have coded the following algorithm [1]:

1. Identify the recurrent classes $[C_1, \dots, C_l]$ with the `recurrentClasses` function.
2. For each class C_i, extract the corresponding sub-matrix M_i.
3. Solve the system $v \cdot C_i = v$, $\sum_{j=1}^{|C_i|} v_j = 1$ which has a unique solution, for each $i = 1, \dots, l$.

[1] We would like to thank Prof. Christophe Dutang for his contributions to the development of this method. He coded a first improvement of the original `steadyStates` method and we could not have reached the current correctness without his previous work

4. Map each state v_i back to the original order in P and assign 0 to the slots corresponding to transient states.

The result is returned in matrix form, one steady state per row.

```
R> steadyStates(mcWeather)

          sunny    cloudy      rain
[1,] 0.4636364 0.3181818 0.2181818
```

A Markov chain can have more than one stationary distribution, as the gambler's ruin example shows.

```
R> gamblerRuinMarkovChain <- function(moneyMax, prob = 0.5) {
+   m <- markovchain:::zeros(moneyMax + 1)
+   m[1,1] <- m[moneyMax + 1,moneyMax + 1] <- 1
+   states <- as.character(0:moneyMax)
+   rownames(m) <- colnames(m) <- states
+
+   for(i in 2:moneyMax){
+     m[i,i-1] <- 1 - prob
+     m[i, i + 1] <- prob
+   }
+
+   new("markovchain", transitionMatrix = m,
+       name = paste("Gambler ruin", moneyMax, "dim", sep = " "))
+ }
R>
R> mcGR4 <- gamblerRuinMarkovChain(moneyMax = 4, prob = 0.5)
R> steadyStates(mcGR4)

     0 1 2 3 4
[1,] 0 0 0 0 1
[2,] 1 0 0 0 0
```

4.3. Entropy rate

For a stationary first-order Markov chain, the entropy rate is the conditional entropy of the next state given the current state:

$$H(X) = H(X_{t+1} \mid X_t) = -\sum_i \pi_i \sum_j p_{ij} \log_b p_{ij},$$

where π is the stationary distribution and $0 \log(0)$ is interpreted as zero. With $b = 2$ the result is measured in bits per transition; with $b = e$ it is measured in nats per transition (Cover and Thomas 2006). The method `entropyRate` uses the unique stationary distribution of a

finite irreducible chain. Aperiodicity is not required; reducible chains are rejected because the result can depend on the chosen stationary distribution (Strelioff, Crutchfield, and H"ubler 2007).

```
R> entropyRate(mcWeather)

[1] 1.366258

R> entropyRate(mcWeather, base = exp(1))

[1] 0.9470176
```

A deterministic chain has entropy rate zero, whereas a chain whose rows are all uniform on n states has the maximum value $\log_b(n)$.

Topological entropy and normalized entropy rate

The entropy rate is bounded above by the *topological entropy* $\log_b \rho(A)$, where A is the 0/1 adjacency matrix of the transitions with positive probability and $\rho(A)$ its Perron root (its spectral radius): the topological entropy is the growth rate of the number of admissible paths, regardless of their probabilities. The ratio of the two, the normalized entropy rate, lies in $[0, 1]$, and measures how close the chain is to the "maximally random" chain on the same set of allowed transitions. It is set to zero when either quantity is zero (for example for a deterministic cycle).

```
R> topologicalEntropy(mcWeather)

[1] 1.584963

R> normalizedEntropyRate(mcWeather)

[1] 0.8620126
```

4.4. Classification of states

The absorbing states are returned by the `absorbingStates` method.

```
R> absorbingStates(mcGR4)

[1] "0" "4"

R> absorbingStates(mcWeather)

character(0)
```

The methods that need to know the communicating classes, the recurrent states or the transient states rely on the internal function `.commclassKernel`, a modification of Tarjan's algorithm Tarjan (1972). It takes a transition matrix of dimension n and returns a list of two items:

1. `classes`, a matrix whose (i, j) entry is `TRUE` if s_i and s_j are in the same communicating class;
2. `closed`, a vector whose i-th entry indicates whether the communicating class to which s_i belongs is closed.

The `summary` method for `markovchain` objects is built on two internal functions that use it. The example matrix of Feres (2007) illustrates the purpose of the function.

```
R> P <- markovchain:::zeros(10)
R> P[1, c(1, 3)] <- 1/2;
R> P[2, 2] <- 1/3; P[2,7] <- 2/3;
R> P[3, 1] <- 1;
R> P[4, 5] <- 1;
R> P[5, c(4, 5, 9)] <- 1/3;
R> P[6, 6] <- 1;
R> P[7, 7] <- 1/4; P[7,9] <- 3/4;
R> P[8, c(3, 4, 8, 10)] <- 1/4;
R> P[9, 2] <- 1;
R> P[10, c(2, 5, 10)] <- 1/3;
R> rownames(P) <- letters[1:10]
R> colnames(P) <- letters[1:10]
R> probMc <- new("markovchain", transitionMatrix = P,
+               name = "Probability MC")
R> summary(probMc)

Probability MC  Markov chain that is composed by:
Closed classes:
a c
b g i
f
Recurrent classes:
{a,c},{b,g,i},{f}
Transient classes:
{d,e},{h},{j}
The Markov chain is not irreducible
The absorbing states are: f
```

With `details = TRUE`, `summary` also prints, after the output above, a block of further properties in the spirit of the printout of a PyDTMC chain: size and rank of the transition matrix, classes, irreducibility, period, regularity, whether the chain is absorbing, reversible, stochastically monotone or symmetric, and, when they are defined, entropy rate, SLEM, spectral gap and Kemeny's constant. The values are also returned, invisibly, in the `details` element of the result.

```
R> summary(mcWeather, details = TRUE)

New Name  Markov chain that is composed by:
Closed classes:
sunny cloudy rain
Recurrent classes:
{sunny,cloudy,rain}
Transient classes:
NONE
The Markov chain is irreducible
The absorbing states are: NONE
Further properties:
  Size                    : 3
  Rank                    : 2
  Communicating classes   : 1 (1 recurrent, 0 transient)
  Irreducible             : yes
  Period                  : 1
  Regular (ergodic)       : yes
  Absorbing chain         : no
  Reversible              : no
  Stochastically monotone : yes
  Symmetric               : no
  Entropy rate (bits)     : 1.36626
  SLEM                    : 0.45
  Spectral gap            : 0.55
  Kemeny constant         : 2.81818
```

The states that belong to a transient class are called transient, and the `transientStates` method returns them.

```
R> transientStates(probMc)

[1] "d" "e" "h" "j"
```

The `canonicForm` method turns a Markov chain into its canonical form, reordering the states so that the recurrent classes come first and the transient states last.

```
R> probMcCanonic <- canonicForm(probMc)
R> probMc

Probability MC
 A  10 - dimensional discrete Markov Chain defined by the following states:
 a, b, c, d, e, f, g, h, i, j
 The transition matrix  (by rows)  is defined as follows:
    a         b    c         d         e f         g    h         i         j
a 0.5 0.0000000 0.50 0.0000000 0.0000000 0 0.0000000 0.00 0.0000000 0.0000000
```

```
b 0.0 0.3333333 0.00 0.0000000 0.0000000 0 0.6666667 0.00 0.0000000 0.0000000
c 1.0 0.0000000 0.00 0.0000000 0.0000000 0 0.0000000 0.00 0.0000000 0.0000000
d 0.0 0.0000000 0.00 0.0000000 1.0000000 0 0.0000000 0.00 0.0000000 0.0000000
e 0.0 0.0000000 0.00 0.3333333 0.3333333 0 0.0000000 0.00 0.3333333 0.0000000
f 0.0 0.0000000 0.00 0.0000000 0.0000000 1 0.0000000 0.00 0.0000000 0.0000000
g 0.0 0.0000000 0.00 0.0000000 0.0000000 0 0.2500000 0.00 0.7500000 0.0000000
h 0.0 0.0000000 0.25 0.2500000 0.0000000 0 0.0000000 0.25 0.0000000 0.2500000
i 0.0 1.0000000 0.00 0.0000000 0.0000000 0 0.0000000 0.00 0.0000000 0.0000000
j 0.0 0.3333333 0.00 0.0000000 0.3333333 0 0.0000000 0.00 0.0000000 0.3333333
```

```
R> probMcCanonic
```

```
Probability MC
 A  10 - dimensional discrete Markov Chain defined by the following states:
 a, c, b, g, i, f, d, e, h, j
 The transition matrix  (by rows)  is defined as follows:
    a    c         b         g         i f         d         e    h         j
a 0.5 0.50 0.0000000 0.0000000 0.0000000 0 0.0000000 0.0000000 0.00 0.0000000
c 1.0 0.00 0.0000000 0.0000000 0.0000000 0 0.0000000 0.0000000 0.00 0.0000000
b 0.0 0.00 0.3333333 0.6666667 0.0000000 0 0.0000000 0.0000000 0.00 0.0000000
g 0.0 0.00 0.0000000 0.2500000 0.7500000 0 0.0000000 0.0000000 0.00 0.0000000
i 0.0 0.00 1.0000000 0.0000000 0.0000000 0 0.0000000 0.0000000 0.00 0.0000000
f 0.0 0.00 0.0000000 0.0000000 0.0000000 1 0.0000000 0.0000000 0.00 0.0000000
d 0.0 0.00 0.0000000 0.0000000 0.0000000 0 0.0000000 1.0000000 0.00 0.0000000
e 0.0 0.00 0.0000000 0.0000000 0.3333333 0 0.3333333 0.3333333 0.00 0.0000000
h 0.0 0.25 0.0000000 0.0000000 0.0000000 0 0.2500000 0.0000000 0.25 0.2500000
j 0.0 0.00 0.3333333 0.0000000 0.0000000 0 0.0000000 0.3333333 0.00 0.3333333
```

The function `is.accessible` checks whether a state s_j is accessible from state s_i, that is whether the probability of eventually reaching s_j starting from s_i is greater than zero.

```
R> is.accessible(object = probMc, from = "a", to = "c")
```

```
[1] TRUE
```

```
R> is.accessible(object = probMc, from = "g", to = "c")
```

```
[1] FALSE
```

In Section 2.2 we observed that, if a DTMC is irreducible, all its states share the same periodicity. The `period` function therefore returns the period of a DTMC, provided that it is irreducible. The example that follows checks with `is.irreducible` whether a DTMC is irreducible and, if so, computes its period with `period`.

```
R> E <- matrix(0, nrow = 4, ncol = 4)
R> E[1, 2] <- 1
```

```
R> E[2, 1] <- 1/3; E[2, 3] <- 2/3
R> E[3,2] <- 1/4; E[3, 4] <- 3/4
R> E[4, 3] <- 1
R>
R> mcE <- new("markovchain", states = c("a", "b", "c", "d"),
+       transitionMatrix = E,
+       name = "E")
R> is.irreducible(mcE)

[1] TRUE

R> period(mcE)

[1] 2
```

The next example is the Markov chain from the Mathematica website (Wolfram Research 2013b), plotted in Figure 4.

```
R> mathematicaMatr <- markovchain:::zeros(5)
R> mathematicaMatr[1,] <- c(0, 1/3, 0, 2/3, 0)
R> mathematicaMatr[2,] <- c(1/2, 0, 0, 0, 1/2)
R> mathematicaMatr[3,] <- c(0, 0, 1/2, 1/2, 0)
R> mathematicaMatr[4,] <- c(0, 0, 1/2, 1/2, 0)
R> mathematicaMatr[5,] <- c(0, 0, 0, 0, 1)
R> statesNames <- letters[1:5]
R> mathematicaMc <- new("markovchain", transitionMatrix = mathematicaMatr,
+                    name = "Mathematica MC", states = statesNames)


Mathematica MC  Markov chain that is composed by:
Closed classes:
c d
e
Recurrent classes:
{c,d},{e}
Transient classes:
{a,b}
The Markov chain is not irreducible
The absorbing states are: e
```

4.5. First passage time distributions and means

Feres (2007) provides code to compute first passage time (within $1, 2, \ldots, n$ steps) given the initial state to be i. The MATLAB listings translated into R on which the `firstPassage` function is based are:

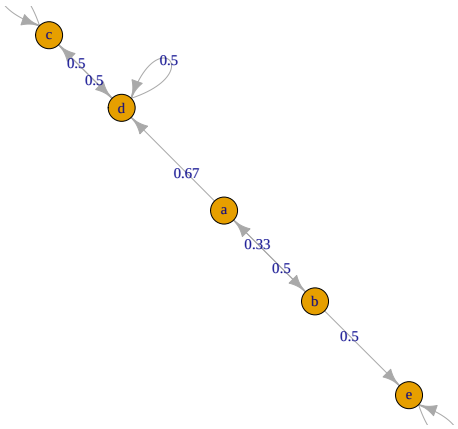


Figure 4: Mathematica 9 example. Markov chain plot.

```
R> .firstpassageKernel <- function(P, i, n){
+   G <- P
+   H <- P[i,]
+   E <- 1 - diag(size(P)[2])
+   for (m in 2:n) {
+     G <- P %*% (G * E)
+     H <- rbind(H, G[i,])
+   }
+   return(H)
+ }
```

We conclude that the probability for the *first* rainy day to be the third one, given that the current state is sunny, is given by:

```
R> firstPassagePdF <- firstPassage(object = mcWeather, state = "sunny",
+                                  n = 10)
R> firstPassagePdF[3, 3]

[1] 0.121
```

To compute the *mean* first passage times, i.e. the expected number of days before it rains given that today is sunny, we can use the `meanFirstPassageTime` function:

```
R> meanFirstPassageTime(mcWeather)

          sunny   cloudy     rain
sunny  0.000000 4.285714 6.666667
cloudy 3.725490 0.000000 5.000000
rain   4.117647 2.857143 0.000000
```

indicating e.g. that the average number of days of sun or cloud before rain is 6.67 if we start counting from a sunny day, and 5 if we start from a cloudy day. Note that we can also specify one or more destination states:

```
R> meanFirstPassageTime(mcWeather,"rain")

   sunny   cloudy
6.666667 5.000000
```

The implementation follows the matrix solutions by (Grinstead and Snell 2006). We can check the result by averaging the first passage probability density function:

```
R> firstPassagePdF.long <- firstPassage(object = mcWeather, state = "sunny",  n = 100)
R> sum(firstPassagePdF.long[,"rain"] * 1:100)

[1] 6.666664
```

4.6. Kemeny's constant

For a finite irreducible chain with stationary distribution π, let m_{ij} denote the mean time to hit state j from state i, using $m_{ii} = 0$. Kemeny's constant is

$$K = \sum_j \pi_j m_{ij}.$$

Remarkably, K does not depend on the starting state i. Equivalently, if $Z = (I - P + \mathbf{1}\pi^T)^{-1}$, then $K = \mathrm{tr}(Z) - 1$ (Kemeny and Snell 1960). The function `kemenyConstant` uses this formula and requires irreducibility, but not aperiodicity. If mean *return* times $m_{jj} = 1/\pi_j$ are placed on the diagonal instead, the weighted sum is $K + 1$.

```
R> kemenyConstant(mcWeather)

[1] 2.818182

R> piWeather <- as.numeric(steadyStates(mcWeather))
R> meanFirstPassageTime(mcWeather) %*% piWeather

           [,1]
sunny  2.818182
cloudy 2.818182
rain   2.818182
```

The second result has one entry for each possible starting state; apart from numerical rounding, all entries equal Kemeny's constant.

4.7. Sensitivity of the stationary distribution

Kemeny's constant uses the fundamental matrix $Z = (I - P + \mathbf{1}\pi^T)^{-1}$; the same matrix also gives the first-order sensitivity of the stationary distribution π to a small change in the chain's transition probabilities. If row k of P is perturbed by a zero-sum vector d (so that $P + e_k d^T$ is still row-stochastic to first order), the resulting change in π is, to first order,

$$\Delta\pi \approx d^T S, \qquad S_{lj} = \pi_k (Z_{lj} - \pi_j)$$

(Schweitzer 1968; Meyer 1980; Cho and Meyer 2001). The function `sensitivity(object, state)` returns the matrix S for a single perturbed state `k = state`; it does not itself pick a direction d, since that depends on which two transition probabilities are being traded off.

```
R> S <- sensitivity(mcWeather, "sunny")
R> S
```

```
            sunny     cloudy        rain
sunny   0.4479264 -0.2471450 -0.20078137
cloudy -0.3529001  0.3850864 -0.03218633
rain   -0.4371976 -0.0364012  0.47359880
```

For example, moving probability mass out of $P(\text{sunny} \to \text{sunny})$ and into $P(\text{sunny} \to \text{rain})$ (so $d = (-\varepsilon, 0, \varepsilon)$) changes π, to first order, by ε times the difference of the `"rain"` and `"sunny"` rows of S:

```
R> eps <- 1e-4
R> predictedChange <- eps * (S["rain", ] - S["sunny", ])
R> predictedChange
```

```
        sunny        cloudy          rain
-8.851240e-05  2.107438e-05  6.743802e-05
```

```
R> perturbedMatrix <- mcWeather@transitionMatrix
R> perturbedMatrix["sunny", "sunny"] <- perturbedMatrix["sunny", "sunny"] - eps
R> perturbedMatrix["sunny", "rain"] <- perturbedMatrix["sunny", "rain"] + eps
R> mcWeatherPerturbed <- new("markovchain", states = states(mcWeather),
+                            transitionMatrix = perturbedMatrix)
R>
R> as.numeric(steadyStates(mcWeatherPerturbed)) - as.numeric(steadyStates(mcWeather))
```

```
[1] -8.849550e-05  2.107036e-05  6.742514e-05
```

The two agree to $O(\varepsilon^2)$, as the first-order theory predicts. Unlike a positional, entry-by-entry sensitivity, `S` is built so that this holds for *any* zero-sum perturbation direction of row `k`, not only the one shown here.

4.8. Spectral diagnostics: SLEM, spectral gap and implied timescales

For a finite, irreducible chain, the Perron-Frobenius theorem guarantees a unique eigenvalue $\lambda_1 = 1$; every other eigenvalue satisfies $|\lambda_k| \leq 1$. The *second largest eigenvalue modulus* (SLEM) is

$$\text{SLEM} = \max_{k>1} |\lambda_k|,$$

and the *spectral gap* is $1 - \text{SLEM}$. Both are lightweight, eigenvalue-based diagnostics of how quickly a chain approaches its stationary distribution: a spectral gap close to 1 indicates fast mixing, while a spectral gap close to 0 indicates slow mixing (Levin and Peres 2017). Only irreducibility is required; unlike some implementations, `slem` and `spectralGap` do not require aperiodicity, and correctly report SLEM = 1 (spectral gap 0) for a periodic chain, reflecting that such a chain never contracts towards its stationary distribution.

```
R> slem(mcWeather)

[1] 0.45

R> spectralGap(mcWeather)

[1] 0.55
```

Each non-trivial eigenvalue λ_k, $k > 1$, also has its own *implied timescale* $\tau_k = -1/\log|\lambda_k|$, measured in number of steps: it is the time the corresponding mode of the chain takes to decay by a factor of $1/e$. The largest such value, τ_2, corresponds to the SLEM and is the slowest, dominant relaxation mode. The term comes from the Markov state model literature in molecular kinetics (Swope, Pitera, and Suits 2004; Prinz, Wu, Sarich, Keller, Senne, Held, Chodera, Schütte, and Noé 2011), where implied timescales are additionally compared across chains fitted at increasing lag times as a check of the Markov approximation; `impliedTimescales` evaluates a single already-fitted chain, leaving that comparison to the user.

```
R> impliedTimescales(mcWeather)

    tau2     tau3
1.252336 0.000000
```

As a textbook illustration with a known closed form, consider simple random walk on the cycle with n states, which moves to either neighbour with probability 1/2 (Levin and Peres 2017, Example 12.9). Its eigenvalues are $\lambda_k = \cos(2\pi k/n)$, $k = 0, \ldots, n-1$, so the SLEM and spectral gap can be checked in closed form. For an *odd* n the walk is aperiodic. Since cos is largest in modulus near 0 and near π, the SLEM is the larger of the moduli attained near $k = 1$ (angle near 0) and near $k = n/2$ (angle near π); for $n = 7$ the latter dominates, at $k = 3$ (and, symmetrically, $k = 4$):

```
R> randomWalkOnCycle <- function(n) {
+   cycleP <- matrix(0, n, n)
```

```
+   for (i in seq_len(n)) {
+     cycleP[i, ((i - 2) %% n) + 1] <- 1 / 2
+     cycleP[i, (i %% n) + 1] <- 1 / 2
+   }
+   new("markovchain", transitionMatrix = cycleP,
+       states = as.character(seq_len(n)),
+       name = paste("Random walk on the", n, "-cycle"))
+ }
R>
R> mcCycle7 <- randomWalkOnCycle(7)
R> slem(mcCycle7)

[1] 0.9009689

R> abs(cos(2 * pi * 3 / 7)) # closed-form SLEM, attained at k = 3 and k = 4

[1] 0.9009689

R> spectralGap(mcCycle7)

[1] 0.09903113

R> impliedTimescales(mcCycle7)

     tau2      tau3      tau4      tau5      tau6      tau7
9.5891459 9.5891459 2.1167476 2.1167476 0.6654537 0.6654537
```

For an *even* n the cycle graph is bipartite and the walk is periodic (period 2): it can only return to its starting state after an even number of steps. Correspondingly λ_k reaches -1 at $k = n/2$, so the SLEM is 1 regardless of n:

```
R> mcCycle8 <- randomWalkOnCycle(8)
R> slem(mcCycle8) # 1, attained at k = 4, where cos(2*pi*4/8) = cos(pi) = -1

[1] 1

R> round(spectralGap(mcCycle8), 12) # numerically 0, up to floating-point error

[1] 0

R> impliedTimescales(mcCycle8)["tau2"] # the k = 4 mode, sorted first, never decays

tau2
 Inf
```

`slem`, `spectralGap` and `impliedTimescales` handle this periodic case directly, rather than treating it as an error: a spectral gap of (numerically) 0 and an infinite implied timescale are the mathematically correct description of a chain that does not converge pointwise to its stationary distribution. The tiny residual before rounding is ordinary floating-point error from `eigen()`, of the same order as `.Machine$double.eps`, not a sign that the chain is "almost" aperiodic.

4.9. Reversibility

A chain is *reversible* with respect to its stationary distribution π if the detailed balance equations hold:

$$\pi_i P_{ij} = \pi_j P_{ji} \quad \text{for every } i, j.$$

Intuitively, a long stationary run of a reversible chain looks statistically the same whether played forwards or backwards in time: the flow of probability from i to j exactly balances the flow back (Norris 1998; Levin and Peres 2017). `is.reversible` checks this directly, and only requires irreducibility, not aperiodicity – reversibility and periodicity are independent properties.

A canonical family of reversible chains is simple random walk on an undirected graph (move to a uniformly random neighbour at each step): the symmetry of "being neighbours" is exactly what makes detailed balance hold. The random walk on a cycle from the previous section, whether the aperiodic `mcCycle7` or the periodic `mcCycle8`, is reversible either way:

```
R> is.reversible(mcCycle7)

[1] TRUE

R> is.reversible(mcCycle8)

[1] TRUE
```

By contrast, a directed cycle – where probability only ever flows "one way around" – is not reversible, since there is a net clockwise flow at stationarity with nothing flowing back:

```
R> states3 <- c("a", "b", "c")
R> directedCycle <- new("markovchain", states = states3,
+   transitionMatrix = matrix(c(0, 1, 0,
+                               0, 0, 1,
+                               1, 0, 0), byrow = TRUE, nrow = 3,
+                             dimnames = list(states3, states3)))
R> is.reversible(directedCycle)

[1] FALSE
```

The weather chain `mcWeather` is a more realistic example of a non-reversible chain: nothing forces the flow from "sunny" to "rain" to balance the flow back, and indeed it does not:

```
R> is.reversible(mcWeather)

[1] FALSE
```

Closest reversible approximation

Many results about Markov chains – spectral bounds, much of MCMC theory – assume reversibility, so when a fitted chain turns out not to be reversible it is natural to ask which reversible chain is closest to it. Write the *time reversal* of P with respect to π as

$$P^*_{ij} = \frac{\pi_j P_{ji}}{\pi_i},$$

the chain run backwards in time; P is reversible exactly when $P = P^*$. `closestReversible` returns the *additive reversibilization* $R = \frac{1}{2}(P + P^*)$ (Fill 1991), which is automatically a valid transition matrix – the rows of P^* sum to $\sum_j \pi_j P_{ji}/\pi_i = (\pi P)_i/\pi_i = 1$ because π is stationary – and is reversible with respect to the same π.

The clearest example is a chain that is as far from reversible as possible: a directed cycle, which only ever moves one way round. Its closest reversible approximation is the undirected random walk on the same triangle:

```
R> approximation <- closestReversible(directedCycle)
R> approximation$chain

Unnamed Markov chain (closest reversible)
 A  3 - dimensional discrete Markov Chain defined by the following states:
 a, b, c
 The transition matrix  (by rows)  is defined as follows:
    a   b   c
a 0.0 0.5 0.5
b 0.5 0.0 0.5
c 0.5 0.5 0.0

R> is.reversible(approximation$chain)

[1] TRUE
```

Applied to the weather chain, the approximation keeps the same stationary distribution and reports how far it had to move to become reversible:

```
R> weatherReversible <- closestReversible(mcWeather)
R> weatherReversible$chain

New Name (closest reversible)
 A  3 - dimensional discrete Markov Chain defined by the following states:
 sunny, cloudy, rain
```

```
 The transition matrix  (by rows)  is defined as follows:
          sunny    cloudy       rain
sunny  0.7000000 0.2029412 0.09705882
cloudy 0.2957143 0.4000000 0.30428571
rain   0.2062500 0.4437500 0.35000000

R> weatherReversible$distance

[1] 0.01074924

R> steadyStates(mcWeather)

         sunny    cloudy      rain
[1,] 0.4636364 0.3181818 0.2181818

R> steadyStates(weatherReversible$chain) # unchanged

         sunny    cloudy      rain
[1,] 0.4636364 0.3181818 0.2181818
```

In what sense is this the *closest* chain? Reversible chains are exactly the self-adjoint operators on $\ell^2(\pi)$, so the natural distance is the Hilbert–Schmidt norm of that space,

$$\|A\|_\pi^2 = \sum_{i,j} \frac{\pi_i}{\pi_j} A_{ij}^2,$$

for which $A \mapsto A^*$ is an isometric involution: R is then the orthogonal projection of P onto the reversible subspace, and the stochasticity and non-negativity constraints are satisfied automatically, so no constrained optimization is needed. The price is that R does *not* minimize the plain Frobenius distance $\|P - R\|_F$, which the return value reports separately.

That distance is the one used by Nielsen and Weber (2015), who show that, for a fixed π, the reversible chain nearest to P in the Frobenius norm solves a convex quadratic program. `closestReversible(norm = "frobenius")` solves it exactly with **quadprog**, taking the entries R_{ij}, $i < j$, as unknowns, $R_{ji} = \pi_i R_{ij}/\pi_j$ by detailed balance and the diagonal as the remainder of each row. Here π need not be stationary for P: any positive distribution can be imposed, and the result is reversible, and therefore stationary, with respect to it. With `stationaryDistribution = "free"` the stationary distribution is optimized too. Every reversible chain is $R_{ij} = S_{ij}/\sum_k S_{ik}$ for a symmetric non-negative matrix of flows S, so the problem becomes a smooth minimization over S, solved by L-BFGS-B from several starting points and then polished by the convex problem at the π found. The set of reversible chains is not convex, so this is a local optimum. Nielsen and Weber keep π fixed: this extension is specific to the package.

For the weather chain the three answers differ:

```
R> weatherFrobenius <- closestReversible(mcWeather, norm = "frobenius")
R> set.seed(2)
```

```
R> weatherFree <- closestReversible(mcWeather, "free", norm = "frobenius")
R> c(additive = weatherReversible$frobeniusDistance,
+   fixedPi = weatherFrobenius$distance,
+   freePi = weatherFree$distance)

   additive     fixedPi      freePi
0.011496118 0.010500198 0.009021996

R> rbind(original = steadyStates(mcWeather)[1, ],
+       freePi = weatherFree$stationaryDistribution)

             sunny    cloudy      rain
original 0.4636364 0.3181818 0.2181818
freePi   0.4648462 0.3195120 0.2156418

R> weatherFree$chain

New Name (closest reversible)
 A  3 - dimensional discrete Markov Chain defined by the following states:
 sunny, cloudy, rain
 The transition matrix  (by rows)  is defined as follows:
           sunny    cloudy       rain
sunny  0.7013060 0.2046385 0.09405552
cloudy 0.2977209 0.4000115 0.30226760
rain   0.2027499 0.4478637 0.34938639
```

The weather chain is close to reversible: the free solution is 22% closer to it than the additive reversibilization, and it moves the stationary probability of each state by less than 0.003.

No published example of the Frobenius problem with its data was found to reproduce. The fixed-π solutions agree with those of `closest_reversible` in **PyDTMC**, which uses an iterative solver, to about 10^{-5}, with a distance that is never larger. For the free problem the tests check that the distance found is not larger than the best fixed-π solution on a fine grid of stationary distributions.

4.10. Mixing time

Where `spectralGap` summarises convergence to stationarity through the transition matrix's spectrum, `mixingTime` answers a more operational question directly: after how many steps is the chain's distribution close to π, no matter where it started? Formally, with total variation distance $d(t) = \max_i \frac{1}{2} \sum_j |P^t_{ij} - \pi_j|$, the mixing time is the smallest t with $d(t) \leq \varepsilon$, for a threshold ε (conventionally 0.25) (Levin and Peres 2017).

```
R> mixingTime(mcWeather)

[1] 2

R> mixingTime(mcWeather, epsilon = 0.01) # a tighter threshold needs more steps
```

```
[1] 6
```

Spectral gap and mixing time are connected by a standard bound (Levin and Peres 2017, Theorem 12.3):

$$t_{\text{mix}}(\varepsilon) \leq \frac{1}{\text{gap}} \log\left(\frac{1}{\varepsilon\, \pi_{\min}}\right),$$

which we can check directly against the values already computed for `mcWeather` in the previous section:

```
R> piWeather <- as.numeric(steadyStates(mcWeather))
R> upperBound <- log(1 / (0.25 * min(piWeather))) / spectralGap(mcWeather)
R> upperBound
```

```
[1] 5.288583
```

```
R> mixingTime(mcWeather) <= upperBound
```

```
[1] TRUE
```

`mixingTime` requires aperiodicity in addition to irreducibility, unlike `slem`, `spectralGap` and `impliedTimescales`: a periodic chain's distribution never settles down at all (it keeps cycling), so "the number of steps until it is close to π" is not a meaningful question to ask of it. Calling it on the periodic `mcCycle8` therefore raises an informative error rather than returning a misleading number:

```
R> mixingTime(mcCycle8)
```

```
Error in ‘mixingTime()‘:
! mixingTime is defined here only for aperiodic (ergodic) Markov chains, because a periodi
```

4.11. Lazy chains

The standard fix for periodicity is to make the chain *lazy*: stay in place with probability α at every step, and otherwise take a step of the original chain (Levin and Peres 2017). For transition matrix P, the lazy chain's transition matrix is

$$L = \alpha I + (1 - \alpha)P.$$

Adding positive probability of staying in every state rules out any period greater than 1, while leaving the stationary distribution and reachability structure unchanged. Applying it to the periodic `mcCycle8` from before:

```
R> lazyCycle8 <- lazyChain(mcCycle8, alpha = 0.5)
R> period(mcCycle8)
```

```
[1] 2
```

```
R> period(lazyCycle8)
```

```
[1] 1
```

```
R> steadyStates(mcCycle8)
```

```
         1     2     3     4     5     6     7     8
[1,] 0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125
```

```
R> steadyStates(lazyCycle8) # unchanged by laziness
```

```
         1     2     3     4     5     6     7     8
[1,] 0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125
```

Now that the chain is aperiodic, `mixingTime` is well-defined for it:

```
R> mixingTime(lazyCycle8)
```

```
[1] 6
```

Laziness is not free, however: every non-trivial eigenvalue of L equals $\alpha + (1-\alpha)\lambda$ for the corresponding eigenvalue λ of P, so the whole non-trivial spectrum is pulled towards α and mixing generally gets *slower* as α grows:

```
R> spectralGap(mcCycle7) # aperiodic already, no laziness needed
```

```
[1] 0.09903113
```

```
R> spectralGap(lazyChain(mcCycle7, alpha = 0.5))
```

```
[1] 0.1882551
```

```
R> spectralGap(lazyChain(mcCycle7, alpha = 0.9))
```

```
[1] 0.03765102
```

4.12. Evolution of a distribution

`redistribute` propagates an initial distribution through a given number of steps and returns the whole trajectory, one row per time point (including the initial time 0). The initial distribution can be omitted (uniform), be a single state name (a point mass) or a numeric vector. `relaxationTime` is the reciprocal of the spectral gap: the characteristic number of steps needed for the distribution to approach stationarity.

```
R> redistribute(mcWeather, steps = 5, initial = "rain")
```

```
      sunny    cloudy      rain
0 0.0000000 0.0000000 1.0000000
1 0.2000000 0.4500000 0.3500000
2 0.3450000 0.3775000 0.2775000
3 0.4102500 0.3448750 0.2448750
4 0.4396125 0.3301938 0.2301938
5 0.4528256 0.3235872 0.2235872
```

```
R> relaxationTime(mcWeather)
```

```
[1] 1.818182
```

An observed sequence of states defines an observable f on the states, f_j being the number of times state j occurs in it. `timeRelaxations` returns $\mu P^t f$, the expected value of f after t steps from the initial distribution μ, and `timeCorrelations` returns $\sum_i \pi_i f_i (P^t g)_i$, the stationary correlation between f at time 0 and the observable g of a second sequence (by default the same) at time t (Noé, Doose, Daidone, Löllmann, Sauer, Chodera, and Smith 2011). Both converge at the speed set by the second largest eigenvalue modulus, and they correspond to the functions of the same name in PyDTMC.

```
R> days <- c("sunny", "sunny", "rain", "cloudy", "sunny")
R> timeCorrelations(mcWeather, days, timePoints = c(0, 1, 2, 10))
```

```
       0        1        2       10
4.709091 4.152727 3.911636 3.714712
```

```
R> timeRelaxations(mcWeather, days, initial = "rain",
+                  timePoints = c(0, 1, 2, 10))
```

```
       0        1        2       10
1.000000 1.400000 1.690000 1.926874
```

The trajectory and the eigenvalues of the transition matrix can also be plotted with the `type` argument of `autoplot` (Figure 5).

```
R> ggplot2::autoplot(mcWeather, type = "flow", steps = 10, initial = "rain")
```

With `type = "comparison"`, `autoplot` places two or more chains defined on the same states side by side, with the states matched by name. By default the transition matrices are drawn as heatmaps on a common probability scale; with `what = "stationary"` the stationary distributions are compared instead, which requires irreducible chains. As an example, a lazy version of the weather chain built with `lazyChain` has exactly the same stationary distribution but visibly different, "stickier" transitions (Figure 6).

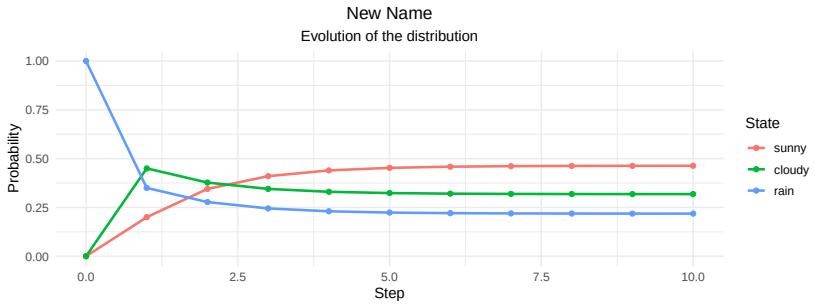


Figure 5: Evolution of the distribution of the weather chain, starting from rain.

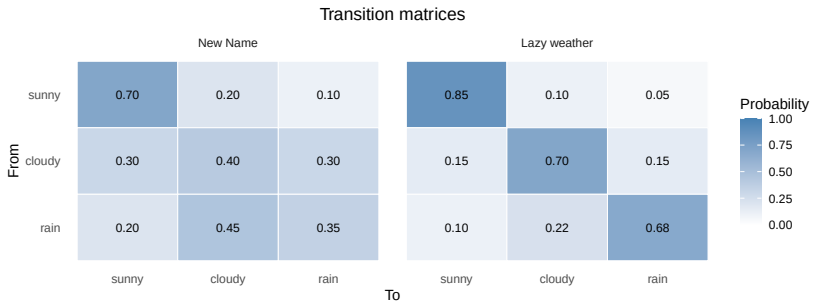


Figure 6: Transition matrices of the weather chain and of its lazy version with alpha = 0.5.

```
R> mcLazyWeather <- lazyChain(mcWeather, alpha = 0.5)
R> mcLazyWeather@name <- "Lazy weather"
R> ggplot2::autoplot(mcWeather, type = "comparison", other = mcLazyWeather)
```

4.13. Mean recurrence time

The `meanRecurrenceTime` method gives the first mean recurrence time (expected number of steps to go back to a state if it was the initial one) for each recurrent state in the transition probabilities matrix for a DTMC. Let's see an example:

```
R> meanRecurrenceTime(mcWeather)

   sunny   cloudy     rain
2.156863 3.142857 4.583333
```

Another example, with not all of its states being recurrent:

```
R> recurrentStates(probMc)

[1] "a" "b" "c" "f" "g" "i"

R> meanRecurrenceTime(probMc)
```

```
       f        b        g        i        a        c
1.000000 2.555556 2.875000 3.833333 1.500000 3.000000
```

4.14. Absorption probabilities and mean absorption time

We are going to use the Drunkard's random walk from (Grinstead and Snell 2006). We have a drunk person walking through the street. Each move the person does, if they have not arrived to either home (corner 1) or to the bar (corner 5) could be to the left corner or to the right one, with equal probability. In case of arrival to the bar or to home, the person stays there.

```
R> drunkProbs <- markovchain:::zeros(5)
R> drunkProbs[1,1] <- drunkProbs[5,5] <- 1
R> drunkProbs[2,1] <- drunkProbs[2,3] <- 1/2
R> drunkProbs[3,2] <- drunkProbs[3,4] <- 1/2
R> drunkProbs[4,3] <- drunkProbs[4,5] <- 1/2
R>
R> drunkMc <- new("markovchain", transitionMatrix = drunkProbs)
R> drunkMc

Unnamed Markov chain
 A  5 - dimensional discrete Markov Chain defined by the following states:
 1, 2, 3, 4, 5
 The transition matrix  (by rows)  is defined as follows:
    1   2   3   4   5
1 1.0 0.0 0.0 0.0 0.0
2 0.5 0.0 0.5 0.0 0.0
3 0.0 0.5 0.0 0.5 0.0
4 0.0 0.0 0.5 0.0 0.5
5 0.0 0.0 0.0 0.0 1.0
```

Recurrent (in fact absorbing states) are:

```
R> recurrentStates(drunkMc)

[1] "1" "5"
```

Transient states are the rest:

```
R> transientStates(drunkMc)

[1] "2" "3" "4"
```

The probability of either being absorbed by the bar or by the sofa at home are:

```
R> absorptionProbabilities(drunkMc)
```

```
     1    5
2 0.75 0.25
3 0.50 0.50
4 0.25 0.75
```

which means that the probability of arriving home / bar is inversely proportional to the distance to each one.

But we also would like to know how much time does the person take to arrive there, which can be done with `meanAbsorptionTime`:

```
R> meanAbsorptionTime(drunkMc)

2 3 4
3 4 3
```

Thus, the expected time to absorption is three steps from states 2 or 4 and four steps from state 3.

The drunkard's walk is a special case of the classical gambler's ruin chain (Grinstead and Snell 2006): a walk on $\{0, 1, \ldots, n\}$ that is absorbed at both ends and, at every interior state, moves up with probability p and down with probability $1 - p$. `gamblersRuin(upperBound, prob, states)` builds exactly this chain, so it reproduces `drunkMc` above without having to write out the transition matrix by hand ($n = 4$, $p = 1/2$, relabelled to states `"1"`, ..., `"5"`):

```
R> ruinMc <- gamblersRuin(upperBound = 4, prob = 0.5, states = as.character(1:5))
R> identical(unclass(ruinMc@transitionMatrix), unclass(drunkMc@transitionMatrix))

[1] TRUE
```

For a general (asymmetric) gambler's ruin with $p \neq 1/2$, the probability of ruin (absorption at 0) starting from state i has the closed form

$$\Pr(\text{ruin} \mid i) = \frac{r^i - r^n}{1 - r^n}, \qquad r = \frac{1 - p}{p},$$

(Grinstead and Snell 2006), which `absorptionProbabilities` reproduces numerically:

```
R> upperBound <- 6
R> prob <- 0.35
R> ruinAsymmetric <- gamblersRuin(upperBound = upperBound, prob = prob)
R> ap <- absorptionProbabilities(ruinAsymmetric)
R>
R> r <- (1 - prob) / prob
R> i <- seq_len(upperBound - 1)
R> theoreticalRuin <- (r^i - r^upperBound) / (1 - r^upperBound)
R>
R> cbind(state = i, package = as.numeric(ap[as.character(i), "0"]),
+       formula = theoreticalRuin)
```

```
     state   package   formula
[1,]     1 0.9785860 0.9785860
[2,]     2 0.9388171 0.9388171
[3,]     3 0.8649606 0.8649606
[4,]     4 0.7277986 0.7277986
[5,]     5 0.4730691 0.4730691
```

Fundamental matrix

For an absorbing chain, let Q be the transition submatrix among transient states. The fundamental matrix is

$$N = (I - Q)^{-1} = I + Q + Q^2 + \cdots . \tag{14}$$

Its (i, j) entry is the expected number of visits to transient state j before absorption when starting from transient state i, including the initial visit. The expected absorption-time vector is $N\mathbf{1}$, while absorption probabilities are obtained from NR, where R contains transitions from transient to absorbing states (Grinstead and Snell 2006).

```
R> fundamentalMatrix(drunkMc)
```

```
    2 3   4
2 1.5 1 0.5
3 1.0 2 1.0
4 0.5 1 1.5
```

Restricting a chain to a subset of states

The matrix Q above – transitions among transient states only – is exactly what `subchain` returns with `method = "submatrix"` when restricted to the transient states: the raw principal submatrix, with no adjustment. Its rows generally sum to *less than* one, because probability that leaves the subset (here, to the absorbing corners) is simply dropped, not redistributed:

```
R> Q <- subchain(drunkMc, transientStates(drunkMc), method = "submatrix")
R> Q
```

```
    2   3   4
2 0.0 0.5 0.0
3 0.5 0.0 0.5
4 0.0 0.5 0.0
```

```
R> rowSums(Q)
```

```
  2   3   4
0.5 1.0 0.5
```

Feeding this Q into $(I - Q)^{-1}$ by hand reproduces `fundamentalMatrix` exactly, confirming the two are the same object:

```
R> solve(diag(nrow(Q)) - Q)
```

```
    2 3   4
2 1.5 1 0.5
3 1.0 2 1.0
4 0.5 1 1.5
```

The second option, `method = "renormalize"`, instead divides each row by its own sum, giving a genuine `markovchain` object: the walk's position *conditioned on never having reached home or the bar* (Norris 1998). This requires every retained state to have some positive probability of staying inside the subset – true here, since every transient corner can move to another transient corner:

```
R> watchedDrunk <- subchain(drunkMc, transientStates(drunkMc), method = "renormalize")
R> watchedDrunk
```

```
Unnamed Markov chain (subchain)
 A  3 - dimensional discrete Markov Chain defined by the following states:
 2, 3, 4
 The transition matrix  (by rows)  is defined as follows:
    2 3   4
2 0.0 1 0.0
3 0.5 0 0.5
4 0.0 1 0.0
```

```
R> rowSums(watchedDrunk@transitionMatrix)
```

```
2 3 4
1 1 1
```

The two methods answer different questions and are not interchangeable: use `"submatrix"` as a building block for further linear algebra (as `fundamentalMatrix` does internally), and `"renormalize"` when you want an actual Markov chain describing conditional behaviour on the subset.

4.15. Committor probability

The committor probability is the probability of reaching a given set of states before another one. Suppose that we start on a cloudy day: the probability of experiencing a rainy day before a sunny one is 0.5, as the output below shows (the function takes the target set A first):

```
R> committorAB(mcWeather,3,1)
```

```
sunny cloudy   rain
  0.0    0.5    1.0
```

4.16. Hitting probabilities

For states i and j, the hitting probability h_{ij} is the probability of ever visiting j when the chain starts in i. For $i = j$, the package uses the return probability after at least one transition, rather than the trivial time-zero probability. The values are the minimal non-negative solution of the first-step equations

$$h_{ij} = p_{ij} + \sum_{k \neq j} p_{ik} h_{kj}. \tag{15}$$

`hittingProbabilities` first analyses the directed graph of positive transitions. This identifies probabilities that are structurally zero or one, without discarding very small positive transitions. This is important for nearly absorbing chains: a small transition probability may imply a long waiting time, but the probability of eventually taking that transition can still be exactly one. The remaining probabilities solve the linear system $(I-Q)h = r$ restricted to the undecided states. The argument `solver` chooses how: `"direct"`, the default, is an LU factorisation, the most accurate choice; `"bicgstab"` is an iterative Krylov method (van der Vorst 1992) working on the sparse matrix, the fastest choice on large sparse chains; `"doubling"` is the doubled Neumann series of earlier versions of the package, kept for reproducibility. The iterative solvers stop at the relative residual `tol` or after `maxIter` iterations, with a warning if `tol` is not reached.

```
R> M <- markovchain:::zeros(5)
R> M[1,1] <- M[5,5] <- 1
R> M[2,1] <- M[2,3] <- 1/2
R> M[3,2] <- M[3,4] <- 1/2
R> M[4,2] <- M[4,5] <- 1/2
R>
R> hittingTest <- new("markovchain", transitionMatrix = M)
R> hittingProbabilities(hittingTest)
```

```
    1     2     3         4   5
1 1.0 0.000 0.000 0.0000000 0.0
2 0.8 0.375 0.500 0.3333333 0.2
3 0.6 0.750 0.375 0.6666667 0.4
4 0.4 0.500 0.250 0.1666667 0.6
5 0.0 0.000 0.000 0.0000000 1.0
```

When only some target states matter, `targets` restricts the computation to the corresponding columns, which coincide with those of the full matrix. On this small chain the three solvers agree to rounding:

```
R> hittingProbabilities(hittingTest, targets = c("1", "5"))
```

```
    1   5
1 1.0 0.0
2 0.8 0.2
3 0.6 0.4
4 0.4 0.6
5 0.0 1.0
```

```
R> all.equal(hittingProbabilities(hittingTest, solver = "direct"),
+            hittingProbabilities(hittingTest, solver = "bicgstab"))
```

```
[1] TRUE
```

For the irreducible finite chain `mcWeather`, every state is eventually visited with probability one:

```
R> hittingProbabilities(mcWeather)
```

```
       sunny cloudy rain
sunny      1      1    1
cloudy     1      1    1
rain       1      1    1
```

4.17. Stochastic monotonicity

A transition matrix P is stochastically monotone if the probability of moving to higher states is non-decreasing with respect to the starting state. In other words, row i is stochastically dominated by row j whenever $i < j$. This property is fundamental in many applied models, such as credit ratings or social mobility, because it guarantees that the Markov chain preserves the initial ordering of states over time.

For instance, in an income mobility model, if you start in a higher income class, your probability of moving to or staying in a high income class should ideally be greater than or equal to someone starting in a lower income class.

The `is.stochasticallyMonotone` method verifies this property for any `markovchain` object:

```
R> # Define the state space for income classes
R> incomeStates <- c("low", "mid", "high")
R>
R> # Example 1: A stochastically monotone transition matrix
R> # Higher starting income leads to higher probabilities of reaching/staying in higher br
R> monotoneMatrix <- matrix(c(0.6, 0.3, 0.1,
+                             0.3, 0.5, 0.2,
+                             0.1, 0.4, 0.5),
+                           byrow = TRUE, nrow = 3,
+                           dimnames = list(incomeStates, incomeStates))
R>
```

```
R> mcIncomeMonotone <- new("markovchain", states = incomeStates,
+                          transitionMatrix = monotoneMatrix,
+                          name = "Monotone Income MC")
R>
R> is.stochasticallyMonotone(mcIncomeMonotone)

[1] TRUE

R> # Example 2: A non-monotone transition matrix
R> # Here, a person in the "mid" class has an unusually high chance of dropping
R> # to "low" (0.7), which is worse than a person already in the "low" class (0.5).
R> # This violates the stochastic monotonicity property.
R> notMonotoneMatrix <- matrix(c(0.5, 0.3, 0.2,
+                                0.7, 0.2, 0.1,
+                                0.1, 0.4, 0.5),
+                              byrow = TRUE, nrow = 3,
+                              dimnames = list(incomeStates, incomeStates))
R>
R> mcIncomeNotMonotone <- new("markovchain", states = incomeStates,
+                             transitionMatrix = notMonotoneMatrix,
+                             name = "Non-Monotone Income MC")
R>
R> is.stochasticallyMonotone(mcIncomeNotMonotone)

[1] FALSE
```

4.18. Classical Markov chain constructors

The package also offers ready-made constructors for a few textbook Markov chains that are frequently the starting point of an applied model, rather than requiring every state and transition of, e.g., a birth-death or diffusion chain to be typed in by hand as in Section 3.

Birth-death chains

A birth-death chain moves, from every interior state, either one step up (with probability p_i) or one step down (with probability q_i), and otherwise stays put; the two absorbing-chain examples above (`drunkMc`/`gamblersRuin()`) are the special case with no "stay" probability at all. `birthDeath(p, q)` builds a general birth-death chain from the $n-1$ up- and down-probabilities of its n states, deliberately taking `p` and `q` to be vectors of length $n-1$ (one per adjacent pair of states) rather than length n: this leaves no leftover boundary probability that would otherwise have to be silently redistributed, and the diagonal ("stay") probability of every interior state is simply what remains, $1 - p_i - q_{i-1}$:

```
R> bd <- birthDeath(p = c(0.3, 0.4, 0.5), q = c(0.2, 0.3, 0.1))
R> bd
```

```
Birth-Death Chain
 A  4 - dimensional discrete Markov Chain defined by the following states:
 1, 2, 3, 4
 The transition matrix  (by rows)  is defined as follows:
    1   2   3   4
1 0.7 0.3 0.0 0.0
2 0.2 0.4 0.4 0.0
3 0.0 0.3 0.2 0.5
4 0.0 0.0 0.1 0.9
```

By construction `birthDeath()` has no special behaviour at its two boundary states beyond "no further up/down move exists there", so state `"1"` and state `"4"` above are themselves absorbing. `toBoundedChain(object, boundaryCondition)` replaces the first and last transition rows of *any* existing chain with an absorbing, reflecting, or "semi-reflecting" rule, so a reflecting boundary can be added to the same chain instead:

```
R> reflected <- toBoundedChain(bd, "reflecting")
R> reflected@transitionMatrix[c(1, 4), ]
```

```
  1 2 3 4
1 0 1 0 0
4 0 0 1 0
```

The Ehrenfest urn model

`urnModel(balls)` builds the Ehrenfest urn diffusion chain (Ehrenfest and Ehrenfest 1907), a classical model of the approach to statistical equilibrium: $m =$ `balls` balls are distributed between two urns, the state is the number of balls in the first urn, and at each step one ball, chosen uniformly at random among all m, is moved to the other urn. From state k this means moving down to $k-1$ with probability k/m and up to $k+1$ with probability $(m-k)/m$ – reflecting, not absorbing, at both $k = 0$ and $k = m$. Its stationary distribution is exactly Binomial$(m, 1/2)$ (Grinstead and Snell 2006):

```
R> ehrenfest <- urnModel(balls = 6)
R> ehrenfest@transitionMatrix
```

```
          0         1         2         3         4         5         6
0 0.0000000 1.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
1 0.1666667 0.0000000 0.8333333 0.0000000 0.0000000 0.0000000 0.0000000
2 0.0000000 0.3333333 0.0000000 0.6666667 0.0000000 0.0000000 0.0000000
3 0.0000000 0.0000000 0.5000000 0.0000000 0.5000000 0.0000000 0.0000000
4 0.0000000 0.0000000 0.0000000 0.6666667 0.0000000 0.3333333 0.0000000
5 0.0000000 0.0000000 0.0000000 0.0000000 0.8333333 0.0000000 0.1666667
6 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 1.0000000 0.0000000
```

```
R> piEhrenfest <- as.numeric(steadyStates(ehrenfest))
R> cbind(package = piEhrenfest, binomial = dbinom(0:6, 6, 0.5))
```

```
      package binomial
[1,] 0.015625 0.015625
[2,] 0.093750 0.093750
[3,] 0.234375 0.234375
[4,] 0.312500 0.312500
[5,] 0.234375 0.234375
[6,] 0.093750 0.093750
[7,] 0.015625 0.015625
```

Combining two chains

`mergeWith(object, other, gamma)` returns the convex combination $(1-\gamma)P_1 + \gamma P_2$ of the transition matrices of two chains defined on the same state space, e.g. to interpolate between two seasonal or regime-specific regimes. States are matched *by name*, not by matrix position, so `other` need not list its states in the same order as `object`; chains on different state-name sets are rejected with an error rather than being combined positionally.

```
R> drySeason <- new("markovchain", states = c("sunny", "rain"),
+                   transitionMatrix = matrix(c(0.9, 0.1, 0.2, 0.8), byrow = TRUE,
+                     nrow = 2, dimnames = list(c("sunny", "rain"), c("sunny", "rain"))),
+                   name = "Dry season")
R> wetSeason <- new("markovchain", states = c("sunny", "rain"),
+                   transitionMatrix = matrix(c(0.5, 0.5, 0.6, 0.4), byrow = TRUE,
+                     nrow = 2, dimnames = list(c("sunny", "rain"), c("sunny", "rain"))),
+                   name = "Wet season")
R>
R> mergeWith(drySeason, wetSeason, gamma = 0.3)
```

```
Dry season + Wet season (merged, gamma = 0.3)
 A  2 - dimensional discrete Markov Chain defined by the following states:
 sunny, rain
 The transition matrix  (by rows)  is defined as follows:
      sunny rain
sunny  0.78 0.22
rain   0.32 0.68
```

The n-step chain

`toNthOrder(object, order)` returns the chain whose one-step transition probabilities are `object`'s `order`-step transition probabilities, i.e. $P^{\texttt{order}}$. It is a thin, discoverability-only wrapper around the `^` operator already shown in Section 3: the two give identical results, and `^` already uses repeated squaring ($O(n^3 \log(\texttt{order}))$) rather than a naive repeated product.

```
R> identical(unclass(toNthOrder(mcWeather, 5)@transitionMatrix),
+            unclass((mcWeather ^ 5)@transitionMatrix))
```

```
[1] TRUE
```

Population genetics: the Moran and Wright-Fisher models

`populationGeneticsModel(model, n, s, u, v)` builds the Markov chain tracking the number of copies of a mutant allele, $0, 1, \ldots, n$, in a population of n individuals, under mutation (backward rate u, mutant to wild-type; forward rate v, wild-type to mutant) and viability selection (mutant fitness $1 + s$ relative to the wild-type's 1), for either the Moran or the Wright-Fisher reproduction scheme. States `"0"` and `as.character(n)` – loss and fixation of the mutant allele – are absorbing, as is standard for both models.

With no selection ($s = 0$) and negligible mutation, a classical result (going back to Wright 1931 and Moran 1958; see Ewens 2004 for a modern treatment) gives the probability that the mutant allele is eventually *fixed* (reaches state n) starting from i copies as simply i/n – the same value for both models, despite their very different reproduction mechanics:

```
R> n <- 8
R> neutralMoran <- populationGeneticsModel(model = "moran", n = n, s = 0)
R> neutralWF <- populationGeneticsModel(model = "wright-fisher", n = n, s = 0)
R>
R> i <- 1:(n - 1)
R> fixation <- function(mc) {
+   B <- absorptionProbabilities(mc)
+   as.numeric(B[as.character(i), as.character(n)])
+ }
R> cbind(i = i, moran = fixation(neutralMoran),
+       wrightFisher = fixation(neutralWF), theoretical = i / n)
```

```
     i moran wrightFisher theoretical
[1,] 1 0.125        0.125       0.125
[2,] 2 0.250        0.250       0.250
[3,] 3 0.375        0.375       0.375
[4,] 4 0.500        0.500       0.500
[5,] 5 0.625        0.625       0.625
[6,] 6 0.750        0.750       0.750
[7,] 7 0.875        0.875       0.875
```

Introducing positive selection ($s > 0$) raises the fixation probability of the mutant allele above this neutral baseline, from every interior starting count, in both models:

```
R> favouredMoran <- populationGeneticsModel(model = "moran", n = n, s = 0.5)
R> favouredWF <- populationGeneticsModel(model = "wright-fisher", n = n, s = 0.5)
R>
R> # fixation probability starting from 4 copies out of 8, neutral vs favoured
R> c(neutral = as.numeric(absorptionProbabilities(neutralMoran)["4", "8"]),
+   favoured = as.numeric(absorptionProbabilities(favouredMoran)["4", "8"]))
```

```
  neutral  favoured
0.5000000 0.8350515
```

Random and special chains

The function `randomMarkovChain` draws a chain with random transition probabilities, as the function `random` of PyDTMC does. Its argument `zeros` sets the exact number of zero probabilities, while `mask` fixes some probabilities in advance: the entries to draw are marked by `NA`, and in each row the free entries share the remaining probability. `dirichletChain` draws every row from a truncated Dirichlet process with concentration `diffusion`, optionally with a bias towards the diagonal; `identityChain` returns the chain that never moves. The `seed` argument makes a draw reproducible without touching the random number stream of the session.

```
R> fixed <- matrix(NA, 3, 3)
R> fixed[1, 1] <- 0.5
R> randomMarkovChain(3, zeros = 2, mask = fixed, seed = 1)@transitionMatrix

          1         2          3
1 0.5000000 0.4572463 0.04275369
2 0.0000000 0.0000000 1.00000000
3 0.5397077 0.3594227 0.10086963

R> dirichletChain(4, diffusion = 1, diagonalBias = 2,
+                 seed = 1)@transitionMatrix

          1          2            3           4
1 0.8533391 0.07032067 0.0752879483 0.001052291
2 0.2089573 0.75664931 0.0228492982 0.011544060
3 0.1796169 0.28299040 0.4809133706 0.056479302
4 0.3989103 0.14352165 0.0002538273 0.457314239

R> identityChain(states = c("a", "b"))@transitionMatrix

  a b
a 1 0
b 0 1
```

5. Statistical analysis

Table 5 lists the functions and methods of the package that help to simulate, aggregate, fit and predict DTMC, and to test hypotheses about them.

5.1. Simulation

A random sequence from a DTMC is simulated with the function `rmarkovchain`. The following code generates 220 days of weather states from the process underlying `mcWeather`.

```
R> weathersOfDays <- rmarkovchain(n = 220, object = mcWeather, t0 = "sunny")
R> weathersOfDays[1:30]

 [1] "cloudy" "rain"   "cloudy" "sunny"  "sunny"  "sunny"  "sunny"  "cloudy"
 [9] "cloudy" "cloudy" "sunny"  "sunny"  "sunny"  "rain"   "cloudy" "cloudy"
[17] "cloudy" "rain"   "sunny"  "sunny"  "sunny"  "cloudy" "cloudy" "sunny"
[25] "sunny"  "rain"   "cloudy" "cloudy" "cloudy" "rain"
```

Similarly, one or more sequences can be simulated from a semi-homogeneous Markov chain, as the following code (applied to the CCHC example) shows.

```
R> patientStates <- rmarkovchain(n = 5, object = mcCCRC, t0 = "H",
+                                include.t0 = TRUE)
R> patientStates[1:10,]

   iteration values
1          1      H
2          1      H
3          1      I
4          1      D
5          1      D
6          2      H
7          2      H
8          2      D
9          2      D
10         2      D
```

Two advanced parameters of `rmarkovchain` select the implementation, among four options: R, R in parallel, C++ and C++ in parallel. The boolean parameters `useRcpp` and `parallel` choose the implementation; the default is `useRcpp = TRUE` and `parallel = FALSE`, i.e. the sequential C++ implementation, which is generally faster than the R one. On multicore processors, setting `parallel = TRUE` takes advantage of several cores; when both `useRcpp = TRUE` and `parallel = TRUE`, the parallelization relies on the **RcppParallel** package (Allaire, Francois, Ushey, Vandenbrouck, Geelnard, and Intel 2016). Every simulated sequence draws from its own random number generator, seeded from R's generator, so that `set.seed()` gives the same result whatever the number of threads. That number is set by the argument `num.cores`; when it is `NULL`, the default, the options `RcppParallel.numThreads` and `Ncpus` and the environment variables `RCPP_PARALLEL_NUM_THREADS` and `OMP_NUM_THREADS` are honoured, in this order, and otherwise at most two threads are used, as required by the CRAN policy.

Plotting sequences

`plotSequence()` draws simulated or observed sequences in the two basic graphs of sequence analysis (Gabadinho, Ritschard, Müller, and Studer 2011): the *sequence index plot*, in which each sequence is a row of cells coloured by state, and the *state distribution plot*, which shows the proportion of sequences in each state at each time point. Given a `markovchain`, it

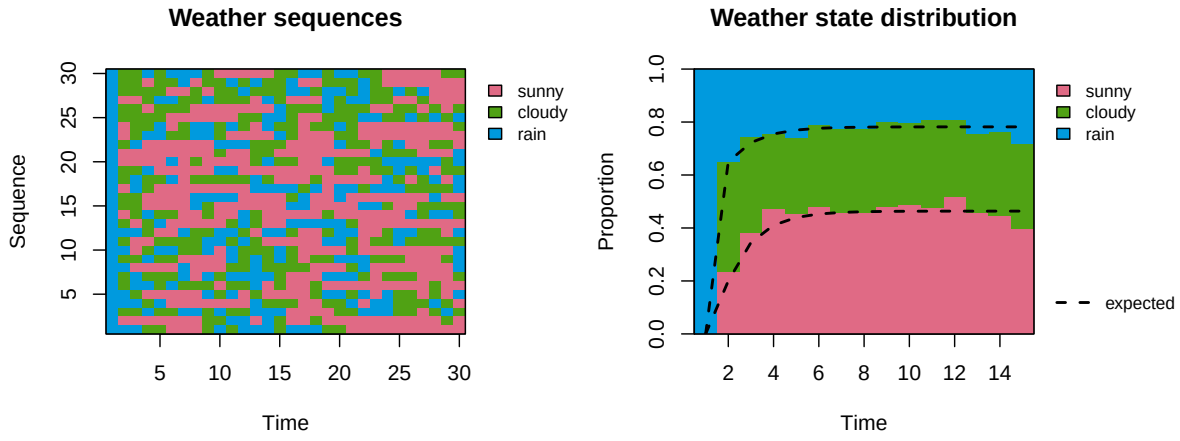


Figure 7: Weather chain: thirty simulated sequences started from rain (left) and the state distribution of three hundred, with the distribution expected from the chain (right).

simulates `nSequences` sequences of `steps` time points from an initial distribution given as in `redistribute()`; given observed data (a sequence, a list of sequences, or a matrix or data frame with one sequence per row), it plots them, and a `chain` can be supplied to overlay the state distribution it predicts from the observed one at the first time point. Figure 7 shows thirty weather sequences started from rain and the distribution of three hundred of them, which approaches the stationary distribution in about five days, as the expected distribution (dashed) does.

```
R> par(mfrow = c(1, 2))
R> plotSequence(mcWeather, nSequences = 30, steps = 30, initial = "rain",
+               seed = 1, main = "Weather sequences")
R> plotSequence(mcWeather, type = "distribution", nSequences = 300, steps = 15,
+               initial = "rain", seed = 1, main = "Weather state distribution")
```

With observed data the second graph is a check of time-homogeneity. The `holson` data contain the trajectories of 1000 individuals over eleven time points; Figure 8 sorts sixty of them by their first state and compares the distribution of all of them with the one predicted by a single time-homogeneous chain fitted to the whole data set.

```
R> data(holson)
R> holsonChain <- markovchainFit(holson[, 2:12])$estimate
R> par(mfrow = c(1, 2))
R> plotSequence(holson[1:60, 2:12], sort = "first", main = "Holson trajectories")
R> holsonPlot <- plotSequence(holson[, 2:12], type = "distribution",
+                             chain = holsonChain, main = "Holson distribution")


R> gap <- holsonPlot$proportions - holsonPlot$expected
R> round(gap[, c(2, 6, 9, 11)], 3)
```

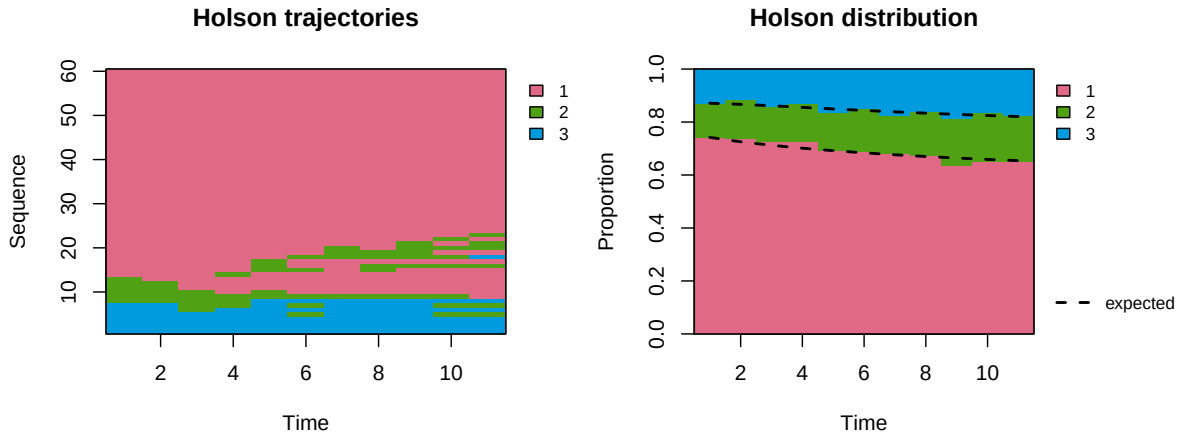


Figure 8: Holson data: sixty trajectories sorted by their first state (left) and the state distribution of the 1000 trajectories against a time-homogeneous chain fitted to them (right).

```
    [,1]   [,2]   [,3]   [,4]
1  0.014  0.005 -0.029 -0.005
2  0.004  0.001  0.013  0.007
3 -0.017 -0.006  0.016 -0.002
```

Most individuals stay in state `1` for the whole period. The proportions observed at each time point stay within three percentage points of those predicted by the fitted chain (the largest gap, 2.9 points for state `1`, is at the ninth time point), so the plot gives no strong sign that the transition probabilities change over time; `assessStationarity()` tests it formally. The values returned invisibly (the sequences as plotted and the observed and expected proportions) allow such comparisons to be made numerically.

5.2. Lumpability and aggregation of states

In some applications the original state space is too detailed, and it is useful to aggregate several micro-states into a smaller number of macro-states. A Markov chain is said to be strongly lumpable with respect to a partition of the state space if, for every pair of macro-states, the total probability of moving from a micro-state in the source block to the destination block is the same for all micro-states in the source block (Kemeny and Snell 1960). Under this condition, the aggregated process is again Markovian and its transition matrix is well defined.

The function `is.lumpable` checks exact strong lumpability for a user-supplied partition. The function `lump` returns the aggregated Markov chain. By default, `lump` requires exact lumpability; setting `force = TRUE` performs an approximate weighted aggregation.

The following example is the classical Land of Oz weather chain. We aggregate `rainy` and `snowy` into a single `Bad_Weather` macro-state, leaving `nice` as a separate macro-state.

```
R> states_oz <- c("rainy", "nice", "snowy")
R>
```

```
R> P_oz <- matrix(
+   c(0.5,  0.25, 0.25,
+     0.5,  0.00, 0.5,
+     0.25, 0.25, 0.5),
+   byrow = TRUE,
+   nrow = 3,
+   dimnames = list(states_oz, states_oz)
+ )
R>
R> mcOz <- new(
+   "markovchain",
+   states = states_oz,
+   transitionMatrix = P_oz,
+   name = "Land of Oz"
+ )
R>
R> partition_oz <- list(
+   Bad_Weather = c("rainy", "snowy"),
+   Nice_Weather = c("nice")
+ )
R>
R> is.lumpable(mcOz, partition_oz)

[1] TRUE

R> lumped_oz <- lump(mcOz, partition_oz)
R> lumped_oz

Land of Oz (Lumped)
 A  2 - dimensional discrete Markov Chain defined by the following states:
 Bad_Weather, Nice_Weather
 The transition matrix  (by rows)  is defined as follows:
             Bad_Weather Nice_Weather
Bad_Weather         0.75         0.25
Nice_Weather        1.00         0.00
```

The lumped chain has two states. From either `rainy` or `snowy`, the probability of moving to `nice` is 0.25, so the transition probability from `Bad_Weather` to `Nice_Weather` is also 0.25. The remaining probability, 0.75, stays inside the `Bad_Weather` macro-state.

If the exact lumpability condition is not satisfied, `lump` stops by default:

```
R> P_bad <- matrix(
+   c(0.5, 0.3, 0.2,
+     0.7, 0.2, 0.1,
+     0.1, 0.4, 0.5),
+   byrow = TRUE,
```

```
+   nrow = 3,
+   dimnames = list(c("A1", "A2", "B"), c("A1", "A2", "B"))
+ )
R>
R> mcBad <- new(
+   "markovchain",
+   states = c("A1", "A2", "B"),
+   transitionMatrix = P_bad
+ )
R>
R> partition_bad <- list(
+   Group_A = c("A1", "A2"),
+   Group_B = c("B")
+ )
R>
R> is.lumpable(mcBad, partition_bad)

[1] FALSE

R> try(lump(mcBad, partition_bad))

Error in lump(mcBad, partition_bad) :
  The Markov chain is not exactly lumpable. Use force = TRUE to perform an approximate wei
```

For exploratory analyses, approximate lumping can be forced:

```
R> lump(mcBad, partition_bad, force = TRUE)

Unnamed Markov chain (Lumped)
 A  2 - dimensional discrete Markov Chain defined by the following states:
 Group_A, Group_B
 The transition matrix  (by rows)  is defined as follows:
          Group_A   Group_B
Group_A 0.8389831 0.1610169
Group_B 0.5000000 0.5000000
```

When `force = TRUE`, the aggregated transition probabilities are computed using stationary weights when available. If a macro-state has zero stationary mass, an arithmetic average over its micro-states is used as a fallback. This produces a valid coarse transition matrix, but it should be interpreted as an approximation: the aggregated process is not guaranteed to be exactly Markovian unless the strong lumpability condition holds.

The helper `autoLump` provides a heuristic way to discover macro-states using spectral clustering. It should be regarded as an exploratory approximation, not as a test of exact lumpability.

```
R> rc <- c("AAA", "AA", "A", "BBB", "BB", "B", "CCC", "D")
R>
```

```
R> creditMatrix <- matrix(
+   c(90.81, 8.33, 0.68, 0.06, 0.08, 0.02, 0.01, 0.01,
+     0.70, 90.65, 7.79, 0.64, 0.06, 0.13, 0.02, 0.01,
+     0.09, 2.27, 91.05, 5.52, 0.74, 0.26, 0.01, 0.06,
+     0.02, 0.33, 5.95, 85.93, 5.30, 1.17, 1.12, 0.18,
+     0.03, 0.14, 0.67, 7.73, 80.53, 8.84, 1.00, 1.06,
+     0.01, 0.11, 0.24, 0.43, 6.48, 83.46, 4.07, 5.20,
+     0.21, 0.00, 0.22, 1.30, 2.38, 11.24, 64.86, 19.79,
+     0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 100.00),
+   nrow = 8,
+   byrow = TRUE,
+   dimnames = list(rc, rc)
+ ) / 100
R>
R> creditMc <- new(
+   "markovchain",
+   states = rc,
+   transitionMatrix = creditMatrix,
+   name = "Credit ratings"
+ )
R>
R> set.seed(123)
R> auto_lumped <- autoLump(creditMc, k = 3)
R>
R> auto_lumped$partition
R> auto_lumped$lumped_chain
```

To keep the vignette fast, the example uses a small transition matrix. In larger applications, the automatically discovered partition should be inspected and validated substantively before being used for modelling or forecasting.

Spectral Kullback-Leibler aggregation

`aggregateStates(object, k, method)` targets the same problem as `autoLump` from a different angle: instead of clustering eigenvectors with k-means, it greedily builds the partition into `k` macro-states that minimizes the Kullback-Leibler (KL) divergence rate between the original chain and the "lifted" macro-chain (Deng, Mehta, and Meyn 2011). Two heuristics are available: `spectral-bottom-up`, which grows the partition one split at a time and suits a large reduction, and `spectral-top-down`, which starts from every state on its own and repeatedly merges the least costly pair, suiting a small reduction; `method = "adaptive"` (the default) picks between them using the same rule of thumb as PyDTMC. Unlike PyDTMC, `k` itself is optional: left unset, it is chosen automatically from the eigengap of the transition matrix's eigenvalues.

Applied to the Land of Oz chain, `aggregateStates` correctly finds the same `Bad_Weather` and `Nice_Weather` partition used above and reproduces `lump`'s aggregated transition matrix exactly (up to the macro-state labels, which `aggregateStates` always names generically):

```
R> agg_oz <- aggregateStates(mcOz, k = 2)
```

```
R> agg_oz$partition

$Macro_1
[1] "rainy" "snowy"

$Macro_2
[1] "nice"

R> isTRUE(all.equal(unname(agg_oz$aggregatedChain@transitionMatrix),
+                   unname(lumped_oz@transitionMatrix)))

[1] TRUE
```

The KL divergence it reports, however, is *not* zero:

```
R> agg_oz$klDivergence

[1] 0.0490225
```

This is not a discrepancy: strong lumpability (what `is.lumpable` checks) only requires that `rainy` and `snowy` send the *same total* probability into `Bad_Weather`, not that they split it the same way *within* the block. Here `rainy` moves to `rainy` with probability 0.5 and to `snowy` with 0.25, while `snowy` moves to `rainy` with 0.25 and to `snowy` with 0.5 – the totals into `Bad_Weather` agree (0.75 each), but the internal split does not. `aggregateStates`'s divergence is sensitive to exactly that within-block heterogeneity, so it is only zero for the stronger property illustrated in `?aggregateStates`'s own example, where the split is also identical. In other words: a zero divergence always implies strong lumpability, but strong lumpability does not always give a zero divergence.

For a chain with no exact lumpability at all, the KL divergence measures how much dynamical information the aggregation discards more generally. The following is the illustrative 4-state example used by Deng, Mehta and Meyn themselves to introduce the method (Deng, Mehta, and Meyn 2009): a doubly stochastic chain built from two tightly coupled pairs of states, `\{1,2\}` and `\{3,4\}`, joined by two weak links (`1-4` and `2-3`):

```
R> states_dmm <- as.character(1:4)
R> P_dmm <- matrix(
+   c(0.5, 0.4, 0.0, 0.1,
+     0.4, 0.5, 0.1, 0.0,
+     0.0, 0.1, 0.5, 0.4,
+     0.1, 0.0, 0.4, 0.5),
+   byrow = TRUE, nrow = 4,
+   dimnames = list(states_dmm, states_dmm)
+ )
R> mcDMM <- new("markovchain", states = states_dmm, transitionMatrix = P_dmm,
+              name = "Deng-Mehta-Meyn (2009) example")
R>
R> agg_dmm <- aggregateStates(mcDMM, k = 2)
R> agg_dmm$partition
```

```
$Macro_1
[1] "1" "2"

$Macro_2
[1] "3" "4"
```

```
R> agg_dmm$aggregatedChain@transitionMatrix
```

```
        Macro_1 Macro_2
Macro_1     0.9     0.1
Macro_2     0.1     0.9
```

`aggregateStates` recovers exactly the bi-partition `\{1,2\}/\{3,4\}` and the aggregated matrix $\begin{pmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{pmatrix}$ reported in the original paper – the two tightly coupled pairs become the two macro-states, and the weak cross-links are what is left over as a 0.1 leakage probability between them. The reported divergence,

```
R> agg_dmm$klDivergence
```

```
[1] 0.1080315
```

also matches the original $\lambda^\star \approx 0.0749$ once the difference in logarithm base is accounted for ($0.0749/\ln 2 \approx 0.1081$): the divergence rate here is reported in bits (base-2 logarithm, matching PyDTMC's own convention), while Deng *et al.* (2009) report it in nats (natural logarithm).

As with `lump`'s output, `aggregateStates`'s `partition` traces every macro-state back to the original state names, so it can be fed directly into `is.lumpable()` or `lump(..., force = TRUE)` for comparison.

5.3. Serialization and export

A `markovchain` object is often built once and then reused elsewhere: shared with a colleague, stored alongside a dataset, or handed to a program written in another language. `toDictionary` and `fromDictionary`, together with `toFile` and `fromFile`, give the object a plain, language-neutral representation for exactly this purpose, mirroring PyDTMC's own `to_dictionary`, `from_dictionary`, `to_file` and `from_file` – with one deliberate difference: PyDTMC represents a chain as a *flat* mapping from every `(from_state, to_state)` pair to its probability (n^2 entries with no state grouping), whereas here the representation is nested by source state, which is both more compact to read and round-trips through R's own list-of-lists idiom without any special tuple-key handling.

We reuse `mcOz`, the Land of Oz chain introduced in the previous section, a well-known textbook example (J. G. Kemeny, J. L.Snell, and G. L. Thompson 1974): unlike a synthetic chain built only to demonstrate file input and output, it is worth saving and reloading.

```
R> oz_dict <- toDictionary(mcOz)
R> str(oz_dict)
```

```
List of 4
 $ name            : chr "Land of Oz"
 $ states          : chr [1:3] "rainy" "nice" "snowy"
 $ byrow           : logi TRUE
 $ transitionMatrix:List of 3
  ..$ rainy:List of 3
  .. ..$ rainy: num 0.5
  .. ..$ nice : num 0.25
  .. ..$ snowy: num 0.25
  ..$ nice :List of 3
  .. ..$ rainy: num 0.5
  .. ..$ nice : num 0
  .. ..$ snowy: num 0.5
  ..$ snowy:List of 3
  .. ..$ rainy: num 0.25
  .. ..$ nice : num 0.25
  .. ..$ snowy: num 0.5
```

`toDictionary` always normalizes to a row-stochastic representation, so it faithfully captures `mcOz` regardless of the chain's own `byrow` storage convention. Reversing it exactly recovers the original transition matrix:

```
R> identical(fromDictionary(oz_dict)@transitionMatrix, mcOz@transitionMatrix)

[1] TRUE
```

`toFile`/`fromFile` write this same representation to disk, inferring the format (JSON, YAML or CSV) from the file extension:

```
R> oz_json <- tempfile(fileext = ".json")
R> toFile(mcOz, oz_json)
R> cat(readLines(oz_json), sep = "\n")

{
  "name": "Land of Oz",
  "states": ["rainy", "nice", "snowy"],
  "byrow": true,
  "transitionMatrix": {
    "rainy": {
      "rainy": 0.5,
      "nice": 0.25,
      "snowy": 0.25
    },
    "nice": {
      "rainy": 0.5,
      "nice": 0,
```

```
      "snowy": 0.5
    },
    "snowy": {
      "rainy": 0.25,
      "nice": 0.25,
      "snowy": 0.5
    }
  }
}
```

```
R> identical(fromFile(oz_json)@transitionMatrix, mcOz@transitionMatrix)

[1] TRUE
```

JSON and YAML preserve every field, including the chain's `name`, at full numeric precision. CSV, which has no natural place for a `name`, only stores the transition matrix itself – the same limitation PyDTMC's own CSV format has:

```
R> oz_csv <- tempfile(fileext = ".csv")
R> toFile(mcOz, oz_csv)
R> cat(readLines(oz_csv), sep = "\n")

"","rainy","nice","snowy"
"rainy",0.5,0.25,0.25
"nice",0.5,0,0.5
"snowy",0.25,0.25,0.5

R> fromFile(oz_csv)@name # the package's own default, not "Land of Oz"

[1] "Unnamed Markov chain"
```

The XML format is PyDTMC's own, so a chain can travel between the two packages in both directions; the `name` is kept as an attribute that PyDTMC ignores. Reading XML requires the **xml2** package.

```
R> oz_xml <- tempfile(fileext = ".xml")
R> toFile(mcOz, oz_xml)
R> cat(readLines(oz_xml)[1:4], sep = "\n")

<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<MarkovChain name="Land of Oz">
    <Item state_from="rainy" state_to="rainy" probability="0.5"/>
    <Item state_from="rainy" state_to="nice" probability="0.25"/>
```

Finally, `toDot` and `toMermaid` export the transition graph as Graphviz DOT or Mermaid text, to be rendered outside R or pasted into a Markdown document; transitions below `minProbability` can be left out.

```
R> cat(toMermaid(mcOz, digits = 2), "\n")

flowchart LR
  s1(("rainy"))
  s2(("nice"))
  s3(("snowy"))
  s1 -->|"0.5"| s1
  s1 -->|"0.25"| s2
  s1 -->|"0.25"| s3
  s2 -->|"0.5"| s1
  s2 -->|"0.5"| s3
  s3 -->|"0.25"| s1
  s3 -->|"0.25"| s2
  s3 -->|"0.5"| s3
```

5.4. Estimation

A time-homogeneous Markov chain can be fitted from observed state sequences. The current version of **markovchain** implements four fitting methods: maximum likelihood (MLE), MLE with Laplace smoothing, bootstrap estimation and maximum a posteriori (MAP) estimation.

Equation 16 shows the maximum likelihood estimator of the transition probability p_{ij}. Here n_{ij} is the number of observed transitions $X_t = s_i, X_{t+1} = s_j$ and $n_i = \sum_{u=1}^{k} n_{iu}$ is the number of observed outgoing transitions from state s_i:

$$\hat{p}_{ij}^{MLE} = \frac{n_{ij}}{n_i}. \tag{16}$$

For a fixed origin state s_i, the counts $(n_{i1}, \ldots, n_{ik})$ are treated as a multinomial sample with row total $n_i = \sum_{u=1}^{k} n_{iu}$. The `standardError` returned for each marginal transition probability is therefore the usual binomial standard error in Equation (17).

$$SE_{ij} = \sqrt{\frac{\hat{p}_{ij}^{MLE}(1 - \hat{p}_{ij}^{MLE})}{n_i}} \tag{17}$$

```
R> weatherFittedMLE <- markovchainFit(data = weathersOfDays, method = "mle",name = "Weathe
R> weatherFittedMLE$estimate

Weather MLE
 A  3 - dimensional discrete Markov Chain defined by the following states:
 cloudy, rain, sunny
 The transition matrix  (by rows)  is defined as follows:
          cloudy      rain     sunny
cloudy 0.4285714 0.2987013 0.2727273
rain   0.5813953 0.1627907 0.2558140
sunny  0.1919192 0.1313131 0.6767677

R> weatherFittedMLE$standardError
```

```
           cloudy       rain      sunny
cloudy 0.05639586 0.05215845 0.05075368
rain   0.07523217 0.05629860 0.06653783
sunny  0.03957938 0.03394440 0.04700668
```

The Laplace smoothing approach is a smoothed version of the MLE in which each count n_{ij} is replaced by $n_{ij} + \alpha$ (Equation 18). The smoothing parameter α must be non-negative; $\alpha = 0$ gives the unsmoothed MLE.

$$\hat{p}_{ij}^{LS} = \frac{n_{ij} + \alpha}{\sum_{u=1}^{k} (n_{iu} + \alpha)} \tag{18}$$

```
R> weatherFittedLAPLACE <- markovchainFit(data = weathersOfDays,
+                                         method = "laplace", laplacian = 0.01,
+                                         name = "Weather LAPLACE")
R> weatherFittedLAPLACE$estimate

Weather LAPLACE
 A  3 - dimensional discrete Markov Chain defined by the following states:
 cloudy, rain, sunny
 The transition matrix  (by rows)  is defined as follows:
          cloudy      rain     sunny
cloudy 0.4285343 0.2987148 0.2727509
rain   0.5812224 0.1629096 0.2558680
sunny  0.1919620 0.1313743 0.6766636
```

The confidence interval option is enabled by default; set `confint = FALSE` to reduce computation time when intervals are not needed. Both MLE and Laplace fitting are based on `createSequenceMatrix`, which returns the raw transition-count matrix.

```
R> createSequenceMatrix(stringchar = weathersOfDays)

       cloudy rain sunny
cloudy     33   23    21
rain       25    7    11
sunny      19   13    67
```

`stringchar` could contain `NA` values, and the transitions containing `NA` would be ignored.

If a state is observed only as the last element of a sequence, or is supplied through the argument `possibleStates` but has no observed outgoing transition, its row has zero total count. In this case the point estimate follows the package convention of assigning a uniform row, $p_{\beta j} = 1/k$. For MLE confidence intervals, such a row carries no empirical information, so the corresponding cell-wise intervals are uninformative.

The uniform row is an assumption about the data rather than a consequence of them. The argument `sanitize` controls it: `TRUE` (the default of `markovchainFit`), or equivalently

"uniform", gives the uniform row; "absorbing" makes every such state absorbing instead, adding no transition that was never observed; FALSE leaves the row at zero, which gives a matrix that is not stochastic.

```
R> partial <- c("a", "b", "a", "b", "a")
R> markovchainFit(partial, possibleStates = c("a", "b", "c"),
+                 sanitize = "uniform")$estimate@transitionMatrix

          a         b         c
a 0.0000000 1.0000000 0.0000000
b 1.0000000 0.0000000 0.0000000
c 0.3333333 0.3333333 0.3333333

R> markovchainFit(partial, possibleStates = c("a", "b", "c"),
+                 sanitize = "absorbing")$estimate@transitionMatrix

  a b c
a 0 1 0
b 1 0 0
c 0 0 1
```

createSequenceMatrix can also be used to obtain raw transition counts from a given $n \times 2$ matrix of (from, to) pairs, as the following example shows:

```
R> myMatr<-matrix(c("a","b","b","a","a","b","b","b","b","a","a","a","b","a"),ncol=2)
R> createSequenceMatrix(stringchar = myMatr,toRowProbs = TRUE)

          a         b
a 0.6666667 0.3333333
b 0.5000000 0.5000000
```

A bootstrap estimation approach is also available to assess the variability of $\hat{p}_{ij}$. The implementation in **markovchain** proceeds as follows:

1. generate bootstrap state sequences using the transition probabilities estimated from the original data. The number of bootstrap samples is controlled by the nboot argument of markovchainFit;
2. apply MLE estimation to each bootstrap sequence; the bootstrap estimates are saved in the bootStrapSamples component of the returned list;
3. compute the bootstrap transition estimate as the row-normalized average of the bootstrap MLE estimates. The reported standardError is the empirical standard deviation of the bootstrap estimates for each transition probability.

```
R> weatherFittedBOOT <- markovchainFit(data = weathersOfDays,
+                                      method = "bootstrap", nboot = 10)
R> weatherFittedBOOT$estimate
```

```
BootStrap Estimate
 A  3 - dimensional discrete Markov Chain defined by the following states:
 cloudy, rain, sunny
 The transition matrix  (by rows)  is defined as follows:
          cloudy      rain     sunny
cloudy 0.4281249 0.2784138 0.2934612
rain   0.5826604 0.1538263 0.2635133
sunny  0.1765214 0.1437399 0.6797386
```

```
R> weatherFittedBOOT$standardError
```

```
           cloudy       rain      sunny
cloudy 0.05504898 0.05025193 0.05483873
rain   0.05661726 0.05881230 0.06499345
sunny  0.03585856 0.03563507 0.03333761
```

The bootstrapping process can be done in parallel thanks to **RcppParallel** package (Allaire *et al.* 2016). The parallelized implementation is recommended when the data sample size or the required number of bootstrap runs is high.

```
R> weatherFittedBOOTParallel <- markovchainFit(data = weathersOfDays,
+                                              method = "bootstrap", nboot = 200,
+                                              parallel = TRUE)
R> weatherFittedBOOTParallel$estimate
R> weatherFittedBOOTParallel$standardError
```

By default the parallel bootstrapping uses at most two threads, as required by the CRAN policy, unless the options or environment variables listed above for `rmarkovchain` ask for more. The argument `num.cores` sets the number of threads explicitly; for example, the following code uses four threads. With the same seed, the bootstrap samples are the same whatever the number of threads.

```
R> markovchainFit(data = weathersOfDays, method = "bootstrap", nboot = 200,
+                 parallel = TRUE, num.cores = 4)
```

For more details, please refer to **RcppParallel** web site.

For all the fitting methods, the `logLikelihood` (Skuriat-Olechnowska 2005) denoted in Equation 19 is provided.

$$LLH = \sum_{i,j} n_{ij} * log(p_{ij}) \tag{19}$$

where n_{ij} is the entry of the frequency matrix and p_{ij} is the entry of the transition probability matrix.

```
R> weatherFittedMLE$logLikelihood
```

```
[1] -208.2122
```

```
R> weatherFittedBOOT$logLikelihood
```

```
[1] -208.4616
```

Confidence matrices of estimated parameters are available as well. The `confidenceInterval` component contains two matrices, `lowerEndpointMatrix` and `upperEndpointMatrix`. The confidence level is 0.95 by default and can be changed through the `confidencelevel` argument of `markovchainFit`.

For MLE fits, **markovchain** reports cell-wise Wilson score intervals for the marginal transition probabilities (Brown, Cai, and DasGupta 2001). Compared with the simple Wald interval $\hat{p}_{ij} \pm z SE_{ij}$, Wilson intervals have better small-sample behaviour and do not collapse to a zero-width interval when a transition is never observed or always observed. With $z = \Phi^{-1}(1-\alpha/2)$, $n_i = \sum_u n_{iu}$, $\hat{p}_{ij} = n_{ij}/n_i$, and confidence level $1-\alpha$, the interval is

$$LowerEndpoint_{ij} = \frac{\hat{p}_{ij} + z^2/(2n_i) - z\sqrt{\hat{p}_{ij}(1-\hat{p}_{ij})/n_i + z^2/(4n_i^2)}}{1 + z^2/n_i}, \tag{20}$$

$$UpperEndpoint_{ij} = \frac{\hat{p}_{ij} + z^2/(2n_i) + z\sqrt{\hat{p}_{ij}(1-\hat{p}_{ij})/n_i + z^2/(4n_i^2)}}{1 + z^2/n_i}. \tag{21}$$

If no outgoing transition is available for a state, the point estimate follows the package convention of a uniform row, while the interval is left maximally uninformative. These intervals are cell-wise marginal intervals, not simultaneous multinomial confidence regions for a full row or matrix. For simultaneous multinomial intervals, use `multinomialConfidenceIntervals`. For bootstrap fits, intervals use the empirical standard deviation of the bootstrap transition-probability estimates and the two-sided normal quantile corresponding to `confidencelevel`. For MAP fits, credible intervals are equal-tailed beta posterior intervals, obtained from the inverse beta distribution (Strelioff *et al.* 2007).

```
R> weatherFittedMLE$confidenceInterval
```

```
NULL
```

```
R> weatherFittedBOOT$confidenceInterval
```

```
$confidenceLevel
[1] 0.95

$lowerEndpointMatrix
          cloudy       rain     sunny
cloudy 0.3202309 0.17992186 0.1859793
rain   0.4716926 0.03855633 0.1361285
sunny  0.1062399 0.07389647 0.6143981
```

```
$upperEndpointMatrix
          cloudy      rain     sunny
cloudy 0.5360190 0.3769058 0.4009431
rain   0.6936282 0.2690963 0.3908981
sunny  0.2468029 0.2135834 0.7450791
```

The function `multinomialConfidenceIntervals` computes simultaneous multinomial confidence intervals. It is an **Rcpp** translation of the functions of the **MultinomialCI** package (Villacorta 2012), which are in turn based on Sison and Glaz (1995).

```
R> multinomialConfidenceIntervals(transitionMatrix =
+         weatherFittedMLE$estimate@transitionMatrix,
+         countsTransitionMatrix = createSequenceMatrix(weathersOfDays))

$confidenceLevel
[1] 0.95

$lowerEndpointMatrix
          cloudy       rain     sunny
cloudy 0.3116883 0.18181818 0.1558442
rain   0.4418605 0.02325581 0.1162791
sunny  0.1111111 0.05050505 0.5959596

$upperEndpointMatrix
          cloudy      rain     sunny
cloudy 0.5464247 0.4165545 0.3905805
rain   0.7298932 0.3112885 0.4043118
sunny  0.2895186 0.2289126 0.7743671
```

Most of the functions for fitting DTMC have been rewritten in C++ using **Rcpp** (Eddelbuettel 2013) since version 0.2.

A DTMC can also be fitted from `matrix` or `data.frame` objects, as shown in the following code.

```
R> data(holson)
R> singleMc<-markovchainFit(data=holson[,2:12],name="holson")
```

The same applies for `markovchainList` (output length has been limited).

```
R> mcListFit<-markovchainListFit(data=holson[,2:6],name="holson")
R> mcListFit$estimate

holson  list of Markov chain(s)
Markovchain  1
Unnamed Markov chain
```

```
 A  3 - dimensional discrete Markov Chain defined by the following states:
 1, 2, 3
 The transition matrix  (by rows)  is defined as follows:
           1         2         3
1 0.94609164 0.05390836 0.0000000
2 0.26356589 0.62790698 0.1085271
3 0.02325581 0.18604651 0.7906977

Markovchain  2
Unnamed Markov chain
 A  3 - dimensional discrete Markov Chain defined by the following states:
 1, 2, 3
 The transition matrix  (by rows)  is defined as follows:
          1         2         3
1 0.9323410 0.0676590 0.0000000
2 0.2551724 0.5103448 0.2344828
3 0.0000000 0.0862069 0.9137931
...
```

Finally, given a `list` object, it is possible to fit a `markovchain` object or to obtain the raw transition matrix.

```
R> c1<-c("a","b","a","a","c","c","a")
R> c2<-c("b")
R> c3<-c("c","a","a","c")
R> c4<-c("b","a","b","a","a","c","b")
R> c5<-c("a","a","c",NA)
R> c6<-c("b","c","b","c","a")
R> mylist<-list(c1,c2,c3,c4,c5,c6)
R> mylistMc<-markovchainFit(data=mylist)
R> mylistMc

$estimate
MLE Fit
 A  3 - dimensional discrete Markov Chain defined by the following states:
 a, b, c
 The transition matrix  (by rows)  is defined as follows:
   a         b         c
a 0.4 0.2000000 0.4000000
b 0.6 0.0000000 0.4000000
c 0.5 0.3333333 0.1666667


$standardError
          a         b         c
a 0.1549193 0.1264911 0.1549193
b 0.2190890 0.0000000 0.2190890
```

```
c 0.2041241 0.1924501 0.1521452

$confidenceLevel
[1] 0.95

$lowerEndpointMatrix
          a          b          c
a 0.1681803 0.05668215 0.16818033
b 0.2307243 0.00000000 0.11762077
c 0.1876163 0.09677141 0.03005337

$upperEndpointMatrix
          a         b         c
a 0.6873262 0.5098375 0.6873262
b 0.8823792 0.4344825 0.7692757
c 0.8123837 0.7000067 0.5635028
```

The same works for `markovchainFitList`.

```
R> markovchainListFit(data=mylist)

$estimate
  list of Markov chain(s)
Markovchain  1
Unnamed Markov chain
 A  3 - dimensional discrete Markov Chain defined by the following states:
 a, b, c
 The transition matrix  (by rows)  is defined as follows:
    a   b   c
a 0.5 0.5 0.0
b 0.5 0.0 0.5
c 1.0 0.0 0.0

Markovchain  2
Unnamed Markov chain
 A  3 - dimensional discrete Markov Chain defined by the following states:
...
```

If any transition contains `NA`, it will be ignored in the results as the above example showed.

Uncertainty of derived properties

The standard errors above refer to the transition probabilities. Properties of the chain computed from them, such as the SLEM, the spectral gap, the relaxation time or the mixing time, are returned by the corresponding functions as single numbers, although they inherit the sampling error of the estimated matrix through non-linear, and sometimes non-smooth, transformations. `bootstrapProperties()` estimates the chain by maximum likelihood from

one or more sequences (or from transition counts) and gives bootstrap standard errors and percentile or basic intervals for any such property. With `method = "multinomial"`, the default, each replicate redraws the departures of every state from its estimated row, keeping the number of visits fixed, which is the conditional bootstrap of Basawa, Green, McCormick, and Taylor (1990); `method = "parametric"` simulates new sequences of the observed lengths from the estimated chain (Kulperger and Prakasa Rao 1989). Besides the four default properties, the entropy rate, Kemeny's constant, the stationary distribution and any function of a `markovchain` returning numbers can be bootstrapped. For the Alofi rainfall data:

```
R> data(rain)
R> rainBoot <- bootstrapProperties(rain$rain, nboot = 1000, seed = 1)
R> rainBoot

Bootstrap of a Markov chain on 3 states (multinomial, 1000 replicates)

               estimate      se   2.5%  97.5%
slem             0.4106 0.02982 0.3533 0.4664
spectralGap      0.5894 0.02982 0.5336 0.6467
relaxationTime   1.6970 0.08648 1.5460 1.8740
mixingTime       2.0000 0.10440 2.0000 2.0000

Percentile 95% intervals.

R> bootstrapProperties(rain$rain, nboot = 1000, seed = 1,
+   statistics = list(dryDays = function(mc) steadyStates(mc)[1, "0"],
+                     "kemenyConstant"))

Bootstrap of a Markov chain on 3 states (multinomial, 1000 replicates)

               estimate      se   2.5%  97.5%
dryDays          0.5009 0.02161 0.4582 0.5427
kemenyConstant   2.7450 0.09742 2.5790 2.9510

Percentile 95% intervals.
```

The rainfall chain forgets its initial state quickly: its SLEM is 0.41 with a standard error of 0.03, and the mixing time is two days in practically every replicate, so its interval collapses to a single value. No published example of bootstrap intervals for these properties with available data was found; the asymptotic theory of the spectral gap of reversible chains estimated from one path is developed by Hsu, Kontorovich, Levin, Peres, Szepesvári, and Wolfer (2019). The coverage of the intervals was therefore checked by simulation. For each of two chains and two methods, 300 sequences were simulated, a 95% percentile interval with 499 replicates was computed from each, and the proportion of intervals containing the true value was recorded (the whole study takes about half a minute, since the replicates of the built-in statistics are computed in C++):

```
R> coverage <- function(truth, n, method, reps = 300) {
+   theta <- c(slem(truth), relaxationTime(truth), mixingTime(truth))
+   hits <- replicate(reps, {
+     x <- rmarkovchain(n - 1, truth, t0 = states(truth)[1], include.t0 = TRUE)
+     ci <- bootstrapProperties(x, nboot = 499, method = method,
+       statistics = c("slem", "relaxationTime", "mixingTime"))$ci
+     ci[, 1] <= theta & theta <= ci[, 2]
+   })
+   rowMeans(hits)
+ }
```

Chain	Method	n	SLEM	Relaxation time	Mixing time
Alofi rainfall (SLEM 0.41)	multinomial	1096	0.950	0.950	1.000
	parametric	1096	0.943	0.943	1.000
	multinomial	200	0.943	0.943	0.997
	parametric	200	0.963	0.963	1.000
Slowly mixing (SLEM 0.88)	multinomial	300	0.950	0.950	0.950
	parametric	300	0.977	0.977	0.983

The first chain is the one estimated from the rainfall data; the second has 0.9 on the diagonal and rare transitions of probability 0.02, so that its relaxation time is 8.3 steps. With 300 simulations the Monte Carlo standard error of a coverage of 0.95 is about 0.013, so the observed coverage of the spectral properties agrees with the nominal level; the spectral gap, equal to one minus the SLEM for these chains, has the same coverage as the SLEM. The intervals of the mixing time, an integer, are conservative when the chain mixes fast. For the slowly mixing chain the intervals are wide (on average 12.8 steps for a relaxation time of 8.3) and skewed, and in two or three replicates out of 499 the rare transitions vanish, the replicate is reducible and the spectral properties cannot be computed: such replicates are counted in `failures` and left out. Wide intervals of this kind are an accurate description of how little 300 observations reveal about a slowly mixing chain, not a failure of the method.

5.5. Prediction

The n-step forward predictions can be obtained using the `predict` methods explicitly written for `markovchain` and `markovchainList` objects. The prediction is the mode of the conditional distribution of X_{t+1} given $X_t = s_j$, being s_j the last realization of the DTMC (homogeneous or semi-homogeneous).

Predicting from a markovchain object

The 3-days forward predictions from `markovchain` object can be generated as follows, assuming that the last two days were respectively "cloudy" and "sunny".

```
R> predict(object = weatherFittedMLE$estimate, newdata = c("cloudy", "sunny"),
+         n.ahead = 3)

[1] "sunny" "sunny" "sunny"
```

Predicting from a markovchainList object

Given an initial two years health status, the 5-year ahead prediction of any CCRC guest is

```
R> predict(mcCCRC, newdata = c("H", "H"), n.ahead = 5)

[1] "H" "D" "D"
```

The prediction has stopped at time sequence since the underlying semi-homogeneous Markov chain has a length of four. In order to continue five years ahead, the argument `continue = TRUE` makes the `predict` method keep using the last `markovchain` of the list.

```
R> predict(mcCCRC, newdata = c("H", "H"), n.ahead = 5, continue = TRUE)

[1] "H" "D" "D" "D" "D"
```

5.6. Statistical tests

In this section, we describe the statistical tests: assessing the independence of consecutive states (`assessIndependence`), the Markov property (`verifyMarkovProperty`), the order (`assessOrder`), the time-homogeneity (`assessStationarity`) of a Markov chain sequence, and the divergence tests (`verifyEmpiricalToTheoretical`, `verifyHomogeneity`) for empirically estimated transition matrices. Most of these tests are based on the χ^2 statistics; the Pearson and likelihood-ratio (G) versions of each test are provided, together with a Monte Carlo option (`method = "simulation"`) that avoids the asymptotic χ^2 approximation altogether by simulating the null distribution of the G statistic directly from the fitted model. Test results are returned as objects of class `htest`, following R's standard convention for hypothesis tests, so that they print, and can be manipulated, like the output of `stats::chisq.test`. Relevant references are Kullback *et al.* (1962) and Anderson and Goodman (1957).

As the sample size increases, even slight departures from the Markov property are easily detected, so these tests are most informative on samples of small or moderate size. The Monte Carlo option is particularly useful in this regime, since the asymptotic χ^2 approximation can be unreliable when expected counts are small; the simulated p-value is computed as $(1 + \#\{G^* \geq G\})/(B + 1)$ over B parametric replicates drawn under the null hypothesis, which is a finite-sample-valid Monte Carlo test (Davison and Hinkley 1997). The accuracy of the statistical inference functions for very small samples is nonetheless an area of ongoing work.

Assessing the independence of consecutive states

The `assessIndependence` function tests whether consecutive observations are independent, $P(X_{t+1} = j \mid X_t = i) = P(X_{t+1} = j)$ for all i, j, against the alternative of a first-order Markov chain. It applies the classical test of Anderson and Goodman (1957) to the table of one-step transition counts n_{ij}. Denoting by $n_{i\cdot}$ and $n_{\cdot j}$ the row and column totals and by $N = \sum_{i,j} n_{ij}$ their sum, the counts expected under independence are $n_{i\cdot} n_{\cdot j}/N$, and the Pearson and likelihood-ratio statistics are

$$X^2 = \sum_i \sum_j \frac{(n_{ij} - n_{i\cdot} n_{\cdot j}/N)^2}{n_{i\cdot} n_{\cdot j}/N}, \qquad G^2 = 2 \sum_i \sum_j n_{ij} \log \frac{n_{ij}\, N}{n_{i\cdot} n_{\cdot j}},$$

both asymptotically χ^2 with $(r-1)(c-1)$ degrees of freedom under the null hypothesis, where r and c are the numbers of states observed as a departure and as an arrival state. When r or c equals one, for instance for a constant sequence, the test is not defined: the degrees of freedom are zero and the p-value is `NA`. As for any χ^2 test on a table of counts, the approximation is unreliable when many expected counts are small. The test is complementary to `verifyMarkovProperty` and `assessOrder`, which compare the first-order model with higher orders rather than with independence.

The following examples use two sequences distributed with the package that were analysed by P. J. Avery and D. A. Henderson (1999): the daily rainfall recorded at Alofi Island, in three classes, and the preproglucacon DNA sequence. Independence is strongly rejected for both. Since the Pearson statistic is that of the table of transition counts, it coincides with the output of `chisq.test` applied to `createSequenceMatrix`. The values shown are computed by the package; they are not claimed to reproduce the figures of the original paper.

```
R> data(rain)
R> assessIndependence(rain$rain)


    Pearson's Chi-squared test for independence of consecutive states

data:  rain$rain
X-squared = 190.61, df = 4, p-value < 2.2e-16

R> data(preproglucacon)
R> assessIndependence(preproglucacon$preproglucacon, method = "G")


    Likelihood-ratio test for independence of consecutive states

data:  preproglucacon$preproglucacon
G-squared = 64.455, df = 9, p-value = 1.841e-10

R> chisq.test(createSequenceMatrix(rain$rain), correct = FALSE)$statistic

X-squared
 190.6124
```

A published benchmark is available for the likelihood-ratio version. Berchtold and Raftery (2002) condition their log-likelihoods on the first 14 observations of the Koeberg wind-direction series (744 hourly observations in four directions, provided by the authors and distributed in `inst/extdata`) and report -954.8 for independence and -413.3 for a first-order chain. Twice their difference, 1083.0, is exactly the likelihood-ratio statistic of the independence test applied to the transitions entering those likelihoods, that is from the 15th observation onwards, and is reproduced by `assessIndependence`:

```
R> koeberg <- as.character(read.csv(system.file("extdata", "koeberg_wind.csv",
+                                                package = "markovchain"))$state)
R> assessIndependence(koeberg[14:length(koeberg)], method = "G")
```

```
    Likelihood-ratio test for independence of consecutive states

data:  koeberg[14:length(koeberg)]
G-squared = 1083.1, df = 9, p-value < 2.2e-16
```

The same check on the epileptic seizure series of the paper gives $G^2 \approx 13.5$ with one degree of freedom, against 13.4 implied by the published log-likelihoods (-129.3 and -122.6); the difference is due to the rounding of the published values.

Assessing the Markov property of a Markov chain sequence

The `verifyMarkovProperty` function verifies whether the Markov property holds for the given chain. The test implemented in the package looks at triplets of successive observations. If $x_1, x_2, \ldots, x_N$ is a set of observations and n_{ijk} is the number of times t $(1 \leq t \leq N-2)$ such that $x_t = i, x_{t+1} = j, x_{t+2} = k$, then, if the Markov property holds, n_{ijk} is expected to be close to $n_{ij}\hat{p}_{jk} = n_{ij}\, n_{jk}/n_j$, where $n_{ij} = \sum_k n_{ijk}$, $n_{jk} = \sum_i n_{ijk}$ and $n_j = \sum_k n_{jk}$. The corresponding Pearson and likelihood-ratio statistics are

$$X^2 = \sum_i \sum_j \sum_k \frac{(n_{ijk} - n_{ij}\hat{p}_{jk})^2}{n_{ij}\hat{p}_{jk}}, \qquad G^2 = 2 \sum_i \sum_j \sum_k n_{ijk} \log \frac{n_{ijk}}{n_{ij}\hat{p}_{jk}},$$

both asymptotically $\chi^2(q)$ under the null hypothesis (Anderson and Goodman 1957). The number of degrees of freedom q is obtained by summing, over each present state j, the degrees of freedom of the corresponding (past $\times$ future) contingency table restricted to the past and future states that were actually observed together with j: $q = \sum_j (s_j - 1)(r_j - 1)$, where s_j is the number of past states i with $n_{ij} > 0$ and r_j is the number of future states k with $n_{jk} > 0$.

```
R> sample_sequence <- c("a", "b", "a", "a", "a", "a", "b", "a", "b", "a",
+                       "b", "a", "a", "b", "b", "b", "a")
R> verifyMarkovProperty(sample_sequence, method = "G")
```

```
    Likelihood-ratio test

data:  sample_sequence
G-squared = 0.59916, df = 2, p-value = 0.7411
```

Burkett, Scherer, and Todd (2019) study recurrent and persistent financial-market states. One of their published tests considers the transition into state 1 and reports the following counts, for which a standard Pearson test of independence gives a non-significant result:

```
R> burkett_counts <- matrix(
+   c(127, 59,
+      11,  7,
+      36, 20,
+       9,  4),
+   nrow = 4,
+   byrow = TRUE,
+   dimnames = list(
+     c("previous state 1", "previous state 2",
+       "previous state 4", "previous state 5"),
+     c("transition to state 1", "transition elsewhere")
+   )
+ )
R> chisq.test(burkett_counts, correct = FALSE)
```

```
Warning in chisq.test(burkett_counts, correct = FALSE): L'approssimazione al
Chi-quadrato potrebbe essere inesatta


    Pearson's Chi-squared test

data:  burkett_counts
X-squared = 0.63611, df = 3, p-value = 0.8881
```

The published contingency table gives a non-significant Pearson test ($\chi^2 \approx 0.64$, 3 degrees of freedom, $p \approx 0.89$). It is included as an independent literature example of a non-rejection; it is not claimed to be numerically identical to `verifyMarkovProperty`, since the published test is a cell-wise test for a particular transition rather than the package's global first-order Markov-property test.

Assessing the order of a Markov chain sequence

The `assessOrder` function checks whether the given chain is of first order or of second order. For each possible present state, we construct a contingency table of the frequency of the future state for each past-to-present state transition, as shown in Table 6, and performs a Pearson χ^2 test of independence between past and future conditionally on the present state, summing the resulting statistics over all present states.

As for `verifyMarkovProperty`, degrees of freedom are summed over present states using only the past and future states actually observed together with that present state, $q = \sum_j (s_j - 1)(r_j - 1)$, rather than a fixed $k(k-1)^2$; the two coincide whenever every state is observed as both a past and a future state for every present state, and only differ – correctly reducing q – when some transitions are unobserved. If the p-value is greater than the given significance level, we cannot reject the hypothesis that the sequence is of first order.

```
R> data(rain)
R> assessOrder(rain$rain)
```

```
    Pearson's Chi-squared test for Markov order

data:  rain$rain
X-squared = 26.096, df = 12, p-value = 0.0104
```

Assessing the stationarity and time-homogeneity of a Markov chain sequence

The `assessStationarity` function assesses whether the transition probabilities of the given chain change over time, i.e. whether

$$p_{ij}(t) = p_{ij} \quad \text{for all} \quad t. \tag{22}$$

For each state, the observed sequence is split into `nblocks` contiguous blocks and a (block $\times$ destination state) contingency table of one-step-ahead transition counts is built, as shown in Table 7; a Pearson χ^2 test of homogeneity across blocks is then performed and the resulting statistics summed across states, with degrees of freedom adapted, state by state, to the blocks and destinations that were actually observed.

```
R> assessStationarity(rain$rain, 10)


    Pearson's Chi-squared test for time-homogeneity

data:  rain$rain
X-squared = 110.02, df = 54, p-value = 1.05e-05
```

Tran and Carmichael (2012) apply likelihood-ratio tests to transition probabilities in a real contractor-payment application, and provide an example where the null hypothesis of time-homogeneous transition probabilities is rejected. Their application also motivates explicit handling of *structural zeros*, i.e. transitions that are impossible under the state-space model, as opposed to *sampling zeros*, i.e. transitions that are possible but were simply not observed. The package allows structural zeros to be specified through the `structural.zeros` argument, so that they are excluded from the contingency tables rather than treated as evidence of homogeneity:

```
R> structural.zeros <- matrix(FALSE, 3, 3,
+                             dimnames = list(c("a", "b", "c"),
+                                             c("a", "b", "c")))
R> structural.zeros["a", "a"] <- TRUE
R>
R> structural_sequence <- c(
+   "a", "b", "a", "c", "a", "b", "c", "a",
+   "b", "c", "b", "a", "c", "b", "a", "c",
+   "a", "b", "c", "a", "b", "c", "b", "a"
+ )
```

```
R>
R> assessStationarity(structural_sequence, nblocks = 4,
+                     structural.zeros = structural.zeros)


    Pearson's Chi-squared test for time-homogeneity

data:  structural_sequence
X-squared = 1.7639, df = 15, p-value = 1
```

Divergence tests for empirically estimated transition matrices

This section discusses tests developed to verify whether:

1. An empirical transition matrix is consistent with a theoretical one.

2. Two or more empirical transition matrices belong to the same DTMC.

The first test is implemented by the `verifyEmpiricalToTheoretical` function. Denoting by f_{ij} the raw empirical transition count and by P_{ij} the theoretical (fully specified) transition probability, expected counts are constructed row-wise as $E_{ij} = f_{i+}P_{ij}$, where $f_{i+} = \sum_j f_{ij}$, and

$$G^2 = 2\sum_{i=1}^{r}\sum_{j=1}^{r} f_{ij} \ln \frac{f_{ij}}{f_{i+}P_{ij}} \sim \chi^2\left(\sum_i (k_i - 1)\right),$$

where k_i is the number of states reachable with positive probability from i under the theoretical matrix; since P is fully specified rather than estimated, no degrees of freedom are lost to parameter estimation. The following example is taken from Kullback *et al.* (1962):

```
R> kullback_counts <- matrix(
+   c(51, 11,  8,
+     12, 31,  9,
+      6, 11, 10),
+   nrow = 3,
+   byrow = TRUE,
+   dimnames = list(c("a", "b", "c"), c("a", "b", "c"))
+ )
R>
R> kullback_P <- matrix(
+   c(5/8, 1/4, 1/8,
+     1/4, 1/2, 1/4,
+     1/4, 3/8, 3/8),
+   nrow = 3,
+   byrow = TRUE,
+   dimnames = list(c("a", "b", "c"), c("a", "b", "c"))
```

```
+ )
R>
R> kullback_mc <- new("markovchain", states = c("a", "b", "c"),
+                     transitionMatrix = kullback_P)
R>
R> verifyEmpiricalToTheoretical(kullback_counts, kullback_mc, method = "G")


    Likelihood-ratio test

data:  kullback_counts
G-squared = 6.5184, df = 6, p-value = 0.3677
```

Kullback et al. report $G^2 = 6.551795$, $df = 6$ and $p = 0.3642899$ for this table. The historical `markovchain` implementation reproduced this published value by using column totals when constructing expected counts. For the row-wise multinomial null implemented by the current package – the standard construction for this kind of test, since each row of a transition matrix is itself a separate multinomial sample – expected counts are $E_{ij} = f_{i+}P_{ij}$, which gives $G^2 = 6.518384$ with the same $df = 6$ and the same non-rejection of the null hypothesis. The small numerical discrepancy with the published value is therefore a correction of the historical implementation rather than a change in the inferential conclusion.

The second test is implemented by the `verifyHomogeneity` function, inspired by (Kullback *et al.* 1962, section 9). Assuming that $i = 1, 2, \ldots, s$ DTMC samples are available and that the cardinality of the state space is r, it verifies whether the s chains belong to the same unknown one. Pooling counts across samples for each state i and comparing each sample's destination distribution to the pooled one, the resulting statistic is asymptotically χ^2 with degrees of freedom summed, state by state, over the samples and destinations actually observed for that state. The following example generates two independent sequences from the same chain, for which the homogeneity hypothesis should not be rejected:

```
R> homogeneity_P <- matrix(
+   c(.70, .20, .10,
+     .20, .50, .30,
+     .10, .30, .60),
+   nrow = 3,
+   byrow = TRUE,
+   dimnames = list(c("a", "b", "c"), c("a", "b", "c"))
+ )
R>
R> homogeneity_mc <- new("markovchain", states = c("a", "b", "c"),
+                        transitionMatrix = homogeneity_P)
R>
R> set.seed(123)
R> x1 <- rmarkovchain(500, homogeneity_mc, t0 = "a")
R> x2 <- rmarkovchain(500, homogeneity_mc, t0 = "a")
R>
R> verifyHomogeneity(list(x1, x2), method = "G")
```

```
    Likelihood-ratio test

data:  list(x1, x2)
G-squared = 1.8881, df = 6, p-value = 0.9297
```

For sparse data, the asymptotic χ^2 approximation underlying any of the tests above should be interpreted with care; the `method = "simulation"` option provides a Monte Carlo alternative that does not rely on the asymptotic approximation.

Distances and divergences between two chains

A test says whether two estimated chains differ by more than sampling noise; it does not say by how much. `compareChains()` reports both kinds of quantities for two chains on the same states, matched by name. The row-wise distances compare the next-step distributions from each state: the largest total variation distance $\max_i \frac{1}{2}\sum_j |P_{ij} - Q_{ij}|$, the same distances averaged with the stationary distribution of the first chain, the largest Hellinger distance, the Frobenius and maximum entrywise distances, and the total variation distance between the two stationary distributions. These follow the definitions of **PyDTMC** and give the same values.

The divergences include the Kullback-Leibler divergence *rate*

$$D(P \,\|\, Q) = \sum_i \pi_i \sum_j P_{ij} \log \frac{P_{ij}}{Q_{ij}},$$

also available on its own as `divergenceRate()`. For irreducible chains it is the limit of $\frac{1}{n} D(X_1, \ldots, X_n \,\|\, Y_1, \ldots, Y_n)$ (Rached, Alajaji, and Campbell 2004), and therefore the average information per step for discriminating the first chain from the second; it is not symmetric, and it is infinite when the first chain makes a transition that the second forbids. It also connects the distances to the tests above. With the departure frequencies of a sample as weights, $2ND(\hat{P} \,\|\, Q)$, N being the number of transitions, is the likelihood-ratio statistic of the hypothesis that the sample comes from Q, and summing it over the samples of a homogeneity test, with Q the pooled estimate, gives Kullback's information statistic. For the two six-state samples of the `kullback` dataset, Table 9.2 of Kullback *et al.* (1962) reports $2\hat{I} = 160.961$:

```
R> data(kullback)
R> pooled <- kullback[[1]] + kullback[[2]]
R> pooledChain <- new("markovchain", states = rownames(pooled),
+                     transitionMatrix = pooled / rowSums(pooled))
R> sum(sapply(kullback, function(counts) {
+   sampleChain <- new("markovchain", states = rownames(counts),
+                      transitionMatrix = counts / rowSums(counts))
+   2 * sum(counts) *
+     divergenceRate(sampleChain, pooledChain, weights = rowSums(counts))
+ }))

[1] 160.9606
```

```
R> verifyHomogeneity(kullback, verbose = FALSE)$statistic

G-squared
 160.9606
```

Both computations reproduce the published value. The degrees of freedom of `verifyHomogeneity()` are 29, not the 30 of the paper's formula $r(s-1)(r-1)$, because no transition from `d` to `a` is observed in either sample and that cell is not counted.

Given two sequences, `compareChains()` estimates both chains on the union of their states and adds the test of equality of `verifyHomogeneity()`. Here the sequences `x1` and `x2` simulated above from the same chain are compared with one simulated from a chain with a different first row:

```
R> other_P <- homogeneity_P
R> other_P["a", ] <- c(.50, .30, .20)
R> other_mc <- new("markovchain", states = c("a", "b", "c"),
+                  transitionMatrix = other_P)
R> set.seed(321)
R> x3 <- rmarkovchain(500, other_mc, t0 = "a")
R> compareChains(x1, x2)

Comparison of two Markov chains on 3 states

Distances:
        totalVariation weightedTotalVariation             hellinger
               0.05407                0.02475               0.04958
             frobenius                maximum             stationary
               0.07660                0.05407               0.03439

Divergences (rates weighted by the stationary distributions):
       rateXY        rateYX      maxRowXY jensenShannon
     0.003850      0.003793      0.009558      0.002455

Test that both sequences come from the same chain:
  G-squared = 1.888, df = 6, p-value = 0.9297

R> compareChains(x1, x3)

Comparison of two Markov chains on 3 states

Distances:
        totalVariation weightedTotalVariation             hellinger
               0.15680                0.09775               0.11700
             frobenius                maximum             stationary
               0.23820                0.15680               0.07174
```

```
Divergences (rates weighted by the stationary distributions):
       rateXY        rateYX      maxRowXY jensenShannon
      0.02686       0.02448       0.05299       0.01363

Test that both sequences come from the same chain:
  G-squared = 12.68, df = 6, p-value = 0.04832
```

The two samples of the same chain are at a weighted total variation distance of a few hundredths, which the test attributes to sampling noise. For the second pair the largest row-wise distance, 0.16, is that of the row of `a`, an estimate of the 0.2 by which the two chains differ there; but sampling noise alone produces distances of 0.11 in the row of `b`, and the test rejects equality at the 5% level only just ($p \approx 0.05$). With 500 observations a difference of this size is close to the limit of what can be detected.

5.7. Pseudo-Bayesian estimation

Hu, Kiesel, and Perraudin (2002) show an empirical quasi-Bayesian method to estimate transition matrices, given an empirical $\hat{P}$ transition matrix (estimated using the classical approach) and an a priori estimate Q. In particular, each row of the matrix is estimated using the linear combination $\alpha \cdot Q + (1 - \alpha) \cdot P$, where α is defined for each row as Equation 23 shows

$$\begin{cases} \hat{\alpha}_i = \frac{\hat{K}_i}{v(i)+\hat{K}_i} \\ \hat{K}_i = \frac{v(i)^2 - \sum_j Y_{ij}^2}{\sum_j (Y_{ij} - v(i) * q_{ij})^2} \end{cases} \tag{23}$$

The following code returns the pseudo Bayesian estimate of the transition matrix:

```
R> pseudoBayesEstimator <- function(raw, apriori){
+   v_i <- rowSums(raw)
+   K_i <- numeric(nrow(raw))
+   sumSquaredY <- rowSums(raw^2)
+   #get numerator
+   K_i_num <- v_i^2-sumSquaredY
+   #get denominator
+   VQ <- matrix(0,nrow= nrow(apriori),ncol=ncol(apriori))
+   for (i in 1:nrow(VQ)) {
+     VQ[i,]<-v_i[i]*apriori[i,]
+   }
+
+   K_i_den<-rowSums((raw - VQ)^2)
+
+   K_i <- K_i_num/K_i_den
+
+   #get the alpha vector
+   alpha <- K_i / (v_i+K_i)
+
+   #empirical transition matrix
```

```
+   Emp<-raw/rowSums(raw)
+
+   #get the estimate
+   out<-matrix(0, nrow= nrow(raw),ncol=ncol(raw))
+   for (i in 1:nrow(out)) {
+     out[i,]<-alpha[i]*apriori[i,]+(1-alpha[i])*Emp[i,]
+   }
+   return(out)
+ }
```

We then apply it to the weather example:

```
R> trueMc<-as(matrix(c(0.1, .9,.7,.3),nrow = 2, byrow = 2),"markovchain")
R> aprioriMc<-as(matrix(c(0.5, .5,.5,.5),nrow = 2, byrow = 2),"markovchain")
R>
R> smallSample<-rmarkovchain(n=20,object = trueMc)
R> smallSampleRawTransitions<-createSequenceMatrix(stringchar = smallSample)
R> pseudoBayesEstimator(
+   raw = smallSampleRawTransitions,
+   apriori = aprioriMc@transitionMatrix
+ ) - trueMc@transitionMatrix

           s1          s2
s1 0.03742072 -0.03742072
s2 0.05471698 -0.05471698

R> biggerSample<-rmarkovchain(n=60,object = trueMc)
R> biggerSampleRawTransitions<-createSequenceMatrix(stringchar = biggerSample)
R> pseudoBayesEstimator(
+   raw = biggerSampleRawTransitions,
+   apriori = aprioriMc@transitionMatrix
+ ) - trueMc@transitionMatrix

            s1          s2
s1 -0.01671885  0.01671885
s2  0.03764148 -0.03764148

R> bigSample<-rmarkovchain(n=100,object = trueMc)
R> bigSampleRawTransitions<-createSequenceMatrix(stringchar = bigSample)
R> pseudoBayesEstimator(
+   raw = bigSampleRawTransitions,
+   apriori = aprioriMc@transitionMatrix
+ ) - trueMc@transitionMatrix

           s1          s2
s1 0.05301222 -0.05301222
s2 0.07310699 -0.07310699
```

5.8. Bayesian estimation

The **markovchain** package provides functionality for maximum a posteriori (MAP) estimation of the chain parameters (at the time of writing this document, only first order models are supported) by Bayesian inference. It also computes the probability of observing a new data set, given a (different) data set. This section describes the mathematics of the methods employed by the package.

Notation and set-up

The data is denoted by D, the model parameters (transition matrix) by θ. The object of interest is $P(\theta|D)$ (posterior density). $\mathcal{A}$ represents an alphabet class, each of whose members represent a state of the chain. Therefore

$$D = s_0 s_1 ... s_{N-1}, s_t \in \mathcal{A}$$

where N is the length of the data set. Also,

$$\theta = \{p(s|u), s \in \mathcal{A}, u \in \mathcal{A}\}$$

where $\sum_{s\in\mathcal{A}} p(s|u) = 1$ for each $u \in \mathcal{A}$.
Our objective is to find θ which maximizes the posterior. That is, if our solution is denoted by $\hat{\theta}$, then

$$\hat{\theta} = \underset{\theta}{argmax} P(\theta|D)$$

where the search space is the set of right stochastic matrices of dimension $|\mathcal{A}| \times |\mathcal{A}|$.
$n(u, s)$ denotes the number of times the word us occurs in D and $n(u) = \sum_{s\in\mathcal{A}} n(u, s)$. The hyper-parameters are similarly denoted by $\alpha(u, s)$ and $\alpha(u)$ respectively.

Methods

Given D, its likelihood conditioned on the observed initial state in D is given by

$$P(D|\theta) = \prod_{s\in\mathcal{A}} \prod_{u\in\mathcal{A}} p(s|u)^{n(u,s)}$$

Conjugate priors are used to model the prior $P(\theta)$. The reasons are twofold:

1. exact expressions can be derived for the MAP estimates, expectations and even variances;
2. model order selection and comparison can be implemented easily (not yet available for the models of this section).

The hyper-parameters determine the form of the prior distribution, which is a product of Dirichlet distributions

$$P(\theta) = \prod_{u\in\mathcal{A}} \Big\{ \frac{\Gamma(\alpha(u))}{\prod_{s\in\mathcal{A}} \Gamma(\alpha(u, s))} \prod_{s\in\mathcal{A}} p(s|u)^{\alpha(u,s)-1} \Big\}$$

where $\Gamma(.)$ is the Gamma function. The hyper-parameters are specified using the `hyperparam` argument in the `markovchainFit` function. If this argument is not specified, then a default value of 1 is assigned to each hyper-parameter which gives a prior that is uniform over the set of transition probability vectors of each row.

Given the likelihood and the prior as described above, the evidence $P(D)$ is simply given by

$$P(D) = \int P(D|\theta)P(\theta)d\theta$$

which simplifies to

$$P(D) = \prod_{u\in\mathcal{A}} \Big\{ \frac{\Gamma(\alpha(u))}{\prod_{s\in\mathcal{A}}\Gamma(\alpha(u,s))} \frac{\prod_{s\in\mathcal{A}}\Gamma(n(u,s)+\alpha(u,s))}{\Gamma(\alpha(u)+n(u))} \Big\}$$

Using Bayes' theorem, the posterior now becomes (thanks to the choice of conjugate priors)

$$P(\theta|D) = \prod_{u\in\mathcal{A}} \Big\{ \frac{\Gamma(n(u)+\alpha(u))}{\prod_{s\in\mathcal{A}}\Gamma(n(u,s)+\alpha(u,s))} \prod_{s\in\mathcal{A}} p(s|u)^{n(u,s)+\alpha(u,s)-1} \Big\}$$

Since this is again a product of Dirichlet distributions, the marginal distribution of a particular parameter $P(s|u)$ of our chain is given by

$$P(s|u) \sim Beta(n(u,s)+\alpha(u,s), n(u)+\alpha(u)-n(u,s)-\alpha(u,s))$$

Thus, the MAP estimate $\hat{\theta}$ is given by

$$\hat{\theta} = \Big\{ \frac{n(u,s)+\alpha(u,s)-1}{n(u)+\alpha(u)-|\mathcal{A}|}, s\in\mathcal{A}, u\in\mathcal{A} \Big\}$$

The function also returns the expected value, given by

$$\mathrm{E}_{\mathrm{post}} p(s|u) = \Big\{ \frac{n(u,s)+\alpha(u,s)}{n(u)+\alpha(u)}, s\in\mathcal{A}, u\in\mathcal{A} \Big\}$$

The variance is given by

$$\mathrm{Var}_{\mathrm{post}} p(s|u) = \frac{n(u,s)+\alpha(u,s)}{(n(u)+\alpha(u))^2} \frac{n(u)+\alpha(u)-n(u,s)-\alpha(u,s)}{n(u)+\alpha(u)+1}$$

The square root of this quantity is the standard error, which is returned by the function.

The credible intervals are constructed as equal-tailed intervals by computing the inverse of the beta integral.

Predictive distribution

Given the old data set, the probability of observing new data is $P(D'|D)$ where D' is the new data set. Let $m(u,s), m(u)$ denote the corresponding counts for the new data. Then,

$$P(D'|D) = \int P(D'|\theta)P(\theta|D)d\theta$$

We already know the expressions for both quantities in the integral and it turns out to be similar to evaluating the evidence

$$P(D'|D) = \prod_{u \in \mathcal{A}} \Big\{ \frac{\Gamma(\alpha(u))}{\prod_{s \in \mathcal{A}} \Gamma(\alpha(u,s))} \frac{\prod_{s \in \mathcal{A}} \Gamma(n(u,s) + m(u,s) + \alpha(u,s))}{\Gamma(\alpha(u) + n(u) + m(u))} \Big\}$$

Choosing the hyper-parameters

The hyper-parameters model the shape of the prior distribution of the parameters. These must be provided by the user. The package offers functionality to translate a given prior belief transition matrix into the hyper-parameter matrix. It is assumed that this belief matrix corresponds to the mean value of the parameters. Since the relation

$$\mathrm{E}_{\mathrm{prior}} p(s|u) = \frac{\alpha(u,s)}{\alpha(u)}$$

holds, the function accepts as input the belief matrix as well as a scaling vector (serves as a proxy for $\alpha(.)$) and proceeds to compute $\alpha(.,.)$.

Alternatively, the function accepts a data sample and infers the hyper-parameters from it. Since the mode of a parameter (with respect to the prior distribution) is proportional to one less than the corresponding hyper-parameter, we set

$$\alpha(u,s) - 1 = m(u,s)$$

where $m(u,s)$ is the $u \to s$ transition count in the data sample. This is regarded as a 'fake count' which helps $\alpha(u,s)$ to reflect knowledge of the data sample.

Usage and examples

```
R> weatherStates <- c("sunny", "cloudy", "rain")
R> byRow <- TRUE
R> weatherMatrix <- matrix(data = c(0.7, 0.2, 0.1,
+                                   0.3, 0.4, 0.3,
+                                   0.2, 0.4, 0.4),
+                          byrow = byRow, nrow = 3,
+                          dimnames = list(weatherStates, weatherStates))
R> mcWeather <- new("markovchain", states = weatherStates,
+                   byrow = byRow, transitionMatrix = weatherMatrix,
+                   name = "Weather")
R> weathersOfDays <- rmarkovchain(n = 365, object = mcWeather, t0 = "sunny")
```

In this section we continue to use the weather example introduced earlier; the chain is defined again below for convenience.

Let us invoke the fit function to estimate the MAP parameters with 92% credible intervals and hyper-parameters as shown below, based on the first 200 days of the weather data. Additionally, let us find out what the probability is of observing the weather data for the next 165 days. The usage would be as follows

```
R> hyperMatrix<-matrix(c(1, 1, 2,
+                        3, 2, 1,
+                        2, 2, 3),
+                      nrow = 3, byrow = TRUE,
+                      dimnames = list(weatherStates,weatherStates))
R> markovchainFit(weathersOfDays[1:200], method = "map",
+                 confidencelevel = 0.92, hyperparam = hyperMatrix)

$estimate
Bayesian Fit
 A  3 - dimensional discrete Markov Chain defined by the following states:
 cloudy, rain, sunny
 The transition matrix  (by rows)  is defined as follows:
          cloudy      rain     sunny
cloudy 0.3750000 0.3437500 0.2812500
rain   0.2950820 0.3770492 0.3278689
sunny  0.2560976 0.1829268 0.5609756


$expectedValue
          cloudy      rain     sunny
cloudy 0.3731343 0.3432836 0.2835821
rain   0.2968750 0.3750000 0.3281250
sunny  0.2588235 0.1882353 0.5529412

$standardError
           cloudy       rain      sunny
cloudy 0.05864965 0.05757859 0.05465982
rain   0.05666911 0.06004806 0.05823810
sunny  0.04722951 0.04215183 0.05361330

$confidenceInterval
$confidenceInterval$confidenceLevel
[1] 0.92

$confidenceInterval$lowerEndpointMatrix
          cloudy      rain     sunny
cloudy 0.2731590 0.2458239 0.1925325
rain   0.2022249 0.2726645 0.2300146
sunny  0.1802077 0.1197595 0.4582499

$confidenceInterval$upperEndpointMatrix
          cloudy      rain     sunny
cloudy 0.4783496 0.4472152 0.3835650
rain   0.4003057 0.4827417 0.4336671
sunny  0.3452775 0.2668332 0.6459104
```

```
$logLikelihood
[1] -207.746

R> predictiveDistribution(weathersOfDays[1:200],
+                         weathersOfDays[201:365],hyperparam = hyperMatrix)

[1] -161.2824
```

The results should not change after permuting the dimensions of the matrix.

```
R> hyperMatrix2<- hyperMatrix[c(2,3,1), c(2,3,1)]
R> markovchainFit(weathersOfDays[1:200], method = "map",
+                 confidencelevel = 0.92, hyperparam = hyperMatrix2)

$estimate
Bayesian Fit
 A  3 - dimensional discrete Markov Chain defined by the following states:
 cloudy, rain, sunny
 The transition matrix  (by rows)  is defined as follows:
          cloudy      rain     sunny
cloudy 0.3750000 0.3437500 0.2812500
rain   0.2950820 0.3770492 0.3278689
sunny  0.2560976 0.1829268 0.5609756


$expectedValue
          cloudy      rain     sunny
cloudy 0.3731343 0.3432836 0.2835821
rain   0.2968750 0.3750000 0.3281250
sunny  0.2588235 0.1882353 0.5529412

$standardError
           cloudy       rain      sunny
cloudy 0.05864965 0.05757859 0.05465982
rain   0.05666911 0.06004806 0.05823810
sunny  0.04722951 0.04215183 0.05361330

$confidenceInterval
$confidenceInterval$confidenceLevel
[1] 0.92

$confidenceInterval$lowerEndpointMatrix
          cloudy      rain     sunny
cloudy 0.2731590 0.2458239 0.1925325
rain   0.2022249 0.2726645 0.2300146
```

```
sunny  0.1802077 0.1197595 0.4582499

$confidenceInterval$upperEndpointMatrix
          cloudy      rain     sunny
cloudy 0.4783496 0.4472152 0.3835650
rain   0.4003057 0.4827417 0.4336671
sunny  0.3452775 0.2668332 0.6459104


$logLikelihood
[1] -207.746

R> predictiveDistribution(weathersOfDays[1:200],
+                         weathersOfDays[201:365],hyperparam = hyperMatrix2)

[1] -161.2824
```

Note that the predictive probability is very small. However, this can be useful when comparing model orders. Suppose we have an idea of the (prior) transition matrix corresponding to the expected value of the parameters, and have a data set from which we want to deduce the MAP estimates. We can infer the hyper-parameters from this known transition matrix itself, and use this to obtain our MAP estimates.

```
R> inferHyperparam(transMatr = weatherMatrix, scale = c(10, 10, 10))

$scaledInference
       cloudy rain sunny
cloudy      4    3     3
rain        4    4     2
sunny       2    1     7
```

Alternatively, we can use a data sample to infer the hyper-parameters.

```
R> inferHyperparam(data = weathersOfDays[1:15])

$dataInference
       cloudy rain sunny
cloudy      2    2     2
rain        2    3     2
sunny       2    2     6
```

In order to use the inferred hyper-parameter matrices, we do

```
R> hyperMatrix3 <- inferHyperparam(transMatr = weatherMatrix,
+                                  scale = c(10, 10, 10))
R> hyperMatrix3 <- hyperMatrix3$scaledInference
R> hyperMatrix4 <- inferHyperparam(data = weathersOfDays[1:15])
R> hyperMatrix4 <- hyperMatrix4$dataInference
```

Now we can safely use `hyperMatrix3` and `hyperMatrix4` with `markovchainFit` (in the `hyperparam` argument).

Supposing we don't provide any hyper-parameters, then the prior is uniform. This is the same as maximum likelihood.

```
R> data(preproglucacon)
R> preproglucacon <- preproglucacon[[2]][1:200]
R> MLEest <- markovchainFit(preproglucacon, method = "mle")
R> MAPest <- markovchainFit(preproglucacon, method = "map")
R> MLEest$estimate
```

```
MLE Fit
 A  4 - dimensional discrete Markov Chain defined by the following states:
 A, C, G, T
 The transition matrix  (by rows)  is defined as follows:
          A          C         G         T
A 0.4084507 0.07042254 0.2112676 0.3098592
C 0.3809524 0.14285714 0.0952381 0.3809524
G 0.3414634 0.07317073 0.1707317 0.4146341
T 0.3181818 0.15151515 0.2424242 0.2878788
```

```
R> MAPest$estimate
```

```
Bayesian Fit
 A  4 - dimensional discrete Markov Chain defined by the following states:
 A, C, G, T
 The transition matrix  (by rows)  is defined as follows:
          A          C         G         T
A 0.4084507 0.07042254 0.2112676 0.3098592
C 0.3809524 0.14285714 0.0952381 0.3809524
G 0.3414634 0.07317073 0.1707317 0.4146341
T 0.3181818 0.15151515 0.2424242 0.2878788
```

6. Continuous time Markov chains

6.1. Definition and generator matrix

The **markovchain** package provides functionality for continuous time Markov chains (CTMCs). CTMCs are a generalization of discrete time Markov chains (DTMCs) in that we allow time to be continuous. We assume a finite state space S, so that the generator matrix introduced below can be stored in memory. We can think of CTMCs as Markov chains in which state transitions can happen at any time.

More formally, we require our CTMCs to satisfy the following two properties:

- The Markov property - let $F_{X(s)}$ denote the information about X up to time s. Let $j \in S$ and $s \leq t$. Then, $P(X(t) = j|F_{X(s)}) = P(X(t) = j|X(s))$.
- Time homogeneity - $P(X(t) = j|X(s) = k) = P(X(t-s) = j|X(0) = k)$.

If both the above properties are satisfied, it is referred to as a time-homogeneous CTMC. If a transition occurs at time t, then $X(t)$ denotes the new state and $X(t) \neq X(t-)$.

Now, let $X(0) = x$ and let T_x be the time a transition occurs from this state. We are interested in the distribution of T_x. For $s, t \geq 0$, it can be shown that $ P(T_x > s+t | T_x > s) = P(T_x > t) $

This is the memoryless property that only the exponential random variable exhibits. Therefore, the exponential distribution is the relevant distribution, and each state $s \in S$ has an exponential holding parameter $\lambda(s)$. Since $ET_x = \frac{1}{\lambda(x)}$, the higher the rate $\lambda(x)$, the smaller the expected time of transitioning out of the state x.

However, specifying this parameter alone for each state gives an incomplete picture of our CTMC. To see why, consider a state x that may transition to either state y or z. The holding parameter enables us to predict when a transition may occur if we start off in state x, but tells us nothing about which state will be next.

To this end, we also need transition probabilities associated with the process, defined as follows (for $y \neq x$) - $p_{xy} = P(X(T_x) = y|X(0) = x)$. Note that $\sum_{y \neq x} p_{xy} = 1$. Let Q denote this transition matrix ($Q_{ij} = p_{ij}$). What is key here is that T_x and the state y are independent random variables. We define $\lambda(x, y) = \lambda(x)p_{xy}$.

We now look at Kolmogorov's backward equation. Let's define $P_{ij}(t) = P(X(t) = j|X(0) = i)$ for $i, j \in S$. The backward equation is given by (it can be proved) $P_{ij}(t) = \delta_{ij}e^{-\lambda(i)t} + \int_0^t \lambda(i)e^{-\lambda(i)s} \sum_{k \neq i} Q_{ik}P_{kj}(t-s)ds$. The first term is non-zero if and only if $i = j$ and represents the probability that the first transition from state i occurs after time t, so that at time t the state is still i. The second term accounts for the transitions that occur before time t: it is the probability of being in state j at time t after a first transition at time $s \leq t$.

In compact form, the equation reads $P'(t) = AP(t)$, where A is the *generator* matrix:

$$A(i, j) = \begin{cases} \lambda(i, j) & \text{if } i \neq j \\ -\lambda(i) & \text{else.} \end{cases}$$

The entries of each row sum to zero, and a CTMC is completely specified by its generator matrix.

6.2. Stationary distributions

The following theorem concerns the existence of a stationary distribution for CTMCs. Note that $X(t)$ is irreducible and recurrent if and only if the embedded chain X_n is irreducible and recurrent.

Suppose that $X(t)$ is irreducible and recurrent. Then $X(t)$ has an invariant measure η, which is unique up to multiplicative factors. Moreover, for each $k \in S$, we have

$$\eta_k = \frac{\pi_k}{\lambda(k)}$$

where π is the invariant measure of the embedded discrete time Markov chain X_n, unique up to multiplicative factors. Finally, η satisfies

$$0 < \eta_j < \infty, \forall j \in S$$

and if $\sum_i \eta_i < \infty$ then η can be normalized to get a stationary distribution.

6.3. Estimation of the generator matrix

Let the data set be $D = \{(s_0, t_0), (s_1, t_1), ..., (s_{N-1}, t_{N-1})\}$ where $N = |D|$. Each s_i is a state from the state space S and during the time $[t_i, t_{i+1}]$ the chain is in state s_i. Let the parameters be represented by $\theta = \{\lambda, P\}$ where λ is the vector of holding parameters for each state and P the transition matrix of the embedded discrete time Markov chain.

Then the probability is given by

$$Pr(D|\theta) \propto \lambda(s_0)e^{-\lambda(s_0)(t_1-t_0)}Pr(s_1|s_0) \cdot \ldots \cdot \lambda(s_{N-2})e^{-\lambda(s_{N-2})(t_{N-1}-t_{N-2})}Pr(s_{N-1}|s_{N-2})$$

Let $n(j|i)$ denote the number of $i \to j$ transitions in D, and $n(i)$ the number of times s_i occurs in D. Let $t(s_i)$ denote the total time the chain spends in state s_i.

Then the MLEs are given by

$$\hat{\lambda}(s) = \frac{n(s)}{t(s)}, \qquad \widehat{Pr}(j|i) = \frac{n(j|i)}{n(i)}.$$

6.4. Expected hitting time

The function `ExpectedTime` calculates the expected hitting time from one state to another. Let j be the target state and let q_{kl} be the entries of the generator matrix, with holding rate $q_k = -q_{kk} > 0$ for every $k \neq j$. Under this condition, the vector p of the expected hitting times of j is the minimal non-negative solution of the system of linear equations

$$\begin{cases} p_k = 0 & k = j \\ -\sum_{l \in S} q_{kl} p_l = 1 & k \neq j. \end{cases} \tag{24}$$

6.5. Transition probabilities over time

The function `probabilityatT` computes the probability of every state at time t for a `ctmc` object. It solves Kolmogorov's backward equation, whose solution is $P(t) = e^{tA}$ for $t \geq 0$ (with $P(0) = I$), where A is the generator matrix. The entry $P_{ij}(t)$ is the probability that the chain is in state j at time t given that it was in state i at time 0. The function also handles generators stored by columns. If the initial state is not provided, it returns the whole transition matrix $P(t)$.

6.6. Examples

To create a CTMC object, one provides a valid generator matrix. A `ctmc` object has the slots `states`, `generator`, `byrow` and `name` (see the documentation of the class for details). Consider the following example in which we aim to model the transition of a molecule from the σ state to the σ^* state. When in the former state, if it absorbs sufficient energy, it can make the jump to the latter state and remains there for some time before transitioning back to the original state. Let us model this by a CTMC:

```
R> energyStates <- c("sigma", "sigma_star")
R> byRow <- TRUE
R> gen <- matrix(data = c(-3, 3,
+                         1, -1), nrow = 2,
+               byrow = byRow, dimnames = list(energyStates, energyStates))
R> molecularCTMC <- new("ctmc", states = energyStates,
+                   byrow = byRow, generator = gen,
+                   name = "Molecular Transition Model")
```

To generate random CTMC transitions, we provide an initial distribution of the states. This must be in the same order as the dimnames of the generator. The output can be returned either as a list or a data frame.

```
R> statesDist <- c(0.8, 0.2)
R> rctmc(n = 3, ctmc = molecularCTMC, initDist = statesDist, out.type = "df", include.T0 =

      states               time
1 sigma_star 0.0610785912722349
2      sigma   1.85906361780109
3 sigma_star    2.0077206621969
```

The argument `n` is the number of transitions to generate. The optional argument `T` is the time at which the simulation ends; to use it, set `n` to `Inf` (the number of transitions is not known beforehand) and set `T` accordingly.

```
R> statesDist <- c(0.8, 0.2)
R> rctmc(n = Inf, ctmc = molecularCTMC, initDist = statesDist, T = 2)

[[1]]
[1] "sigma"      "sigma_star"

[[2]]
[1] 0.0000000 0.1090702
```

The stationary distribution is returned by the `steadyStates` function:

```
R> steadyStates(molecularCTMC)

     sigma sigma_star
[1,]  0.25       0.75
```

The expected time needed to move from the state `sigma` to the state `sigma_star` is 1/3 of a time unit, the inverse of the holding rate of `sigma`, since the only possible transition leads to the target. The probabilities of the two states half a time unit after the chain was in `sigma` follow from $P(t) = e^{tA}$:

```
R> ExpectedTime(molecularCTMC, 1, 2)

[1] 0.3333333

R> probabilityatT(molecularCTMC, 0.5, 1)

[1] 0.3515015 0.6484985
```

The `ctmcFit` function fits a CTMC from a sequence of states and the times at which they are observed. It returns the maximum likelihood estimates of the parameters along with confidence intervals.

```
R> data <- list(c("a", "b", "c", "a", "b", "a", "c", "b", "c"),
+               c(0, 0.8, 2.1, 2.4, 4, 5, 5.9, 8.2, 9))
R> ctmcFit(data)

$estimate
An object of class "ctmc"
Slot "states":
[1] "a" "b" "c"

Slot "byrow":
[1] TRUE

Slot "generator":
           a          b          c
a -0.9090909  0.6060606  0.3030303
b  0.3225806 -0.9677419  0.6451613
c  0.3846154  0.3846154 -0.7692308

Slot "name":
[1] ""


$errors
$errors$dtmcConfidenceInterval
$errors$dtmcConfidenceInterval$confidenceLevel
[1] 0.95

$errors$dtmcConfidenceInterval$lowerEndpointMatrix
           a          b          c
```

```
a 0.00000000 0.20765960 0.06149194
b 0.06149194 0.00000000 0.20765960
c 0.09453121 0.09453121 0.00000000

$errors$dtmcConfidenceInterval$upperEndpointMatrix
          a         b         c
a 0.5614970 0.9385081 0.7923404
b 0.7923404 0.5614970 0.9385081
c 0.9054688 0.9054688 0.6576198


$errors$lambdaConfidenceInterval
$errors$lambdaConfidenceInterval$lowerEndpointVector
[1] 0 0 0

$errors$lambdaConfidenceInterval$upperEndpointVector
[1] 1.937805 2.062825 1.835311
```

One approach to obtain the generator matrix is to apply the `logm` function from the **expm** package on a transition matrix. Numeric issues arise, see Israel, Rosenthal, and Wei (2001). For example, applying the standard `method` ('Higham08') on `mcWeather` raises an error, whilst the alternative method (eigenvalue decomposition) is OK. The following code estimates the generator matrix of the `mcWeather` transition matrix.

```
R> mcWeatherQ <- expm::logm(mcWeather@transitionMatrix,method='Eigen')
R> mcWeatherQ

            sunny     cloudy        rain
sunny  -0.4633266  0.4270614  0.03626515
cloudy  0.6405921 -1.8129810  1.17238885
rain    0.0725303  1.5631851 -1.63571543
```

The "half-day" transition probability of the `mcWeather` chain is therefore

```
R> mcWeatherHalfDayTM <- expm::expm(mcWeatherQ*.5)
R> mcWeatherHalfDay <- new("markovchain",transitionMatrix=mcWeatherHalfDayTM,name="Half Da
R> mcWeatherHalfDay

Half Day Weather Transition Matrix
 A  3 - dimensional discrete Markov Chain defined by the following states:
 sunny, cloudy, rain
 The transition matrix  (by rows)  is defined as follows:
            sunny    cloudy       rain
sunny  0.81711847 0.1367957 0.04608582
cloudy 0.20519357 0.5226767 0.27212969
rain   0.09217163 0.3628396 0.54498878
```

The **ctmcd** package (Pfeuffer 2017) provides various functions to estimate the generator matrix (GM) of a CTMC process using different methods. The following code provides a way to join **markovchain** and **ctmcd** computations.

```
R> if(requireNamespace(package='ctmcd', quietly = TRUE)) {
+ require(ctmcd)
+ require(expm)
+ #defines a function to transform a GM into a TM
+ gm_to_markovchain<-function(object, t=1) {
+   if(!inherits(object, c("gm","matrix","Matrix")))
+     stop("Error! Expecting either a matrix or a gm object")
+   if (inherits(object, c("matrix","Matrix"))) generator_matrix<-object else generator_ma
+   #must add importClassesFrom("markovchain",markovchain) in the NAMESPACE
+   #must add importFrom(expm, "expm")
+   transitionMatrix<-expm(generator_matrix*t)
+   out<-as(transitionMatrix,"markovchain")
+   return(out)
+ }
+ #loading ctmcd dataset
+ data(tm_abs)
+ gm0=matrix(1,8,8) #initializing
+ diag(gm0)=0
+ diag(gm0)=-rowSums(gm0)
+ gm0[8,]=0
+ gmem=gm(tm_abs,te=1,method="EM",gmguess=gm0) #estimating GM
+ mc_at_2=gm_to_markovchain(object=gmem, t=2) #converting to TM at time 2
+ } else {
+   warning('package ctmcd unavailable')
+ }

Caricamento del pacchetto richiesto: ctmcd

Caricamento del pacchetto richiesto: expm


Caricamento pacchetto: 'expm'

Il seguente oggetto è mascherato da 'package:Matrix':

    expm
```

7. Applications

This section shows applications of DTMC in various fields.

7.1. Weather forecasting

Markov chains provide a simple model to predict the next day's weather given the current meteorological condition. The first application herewith shown is the "Land of Oz example" from J. G. Kemeny *et al.* (1974), the second is the "Alofi Island Rainfall" from P. J. Avery and D. A. Henderson (1999).

Land of Oz

The Land of Oz is acknowledged not to have ideal weather conditions: the weather is snowy or rainy very often and there are never two nice days in a row. Consider three weather states: rainy, nice and snowy. Let the transition matrix be as in the following:

```
R> mcWP <- new("markovchain", states = c("rainy", "nice", "snowy"),
+          transitionMatrix = matrix(c(0.5, 0.25, 0.25,
+                                      0.5, 0, 0.5,
+                                      0.25,0.25,0.5), byrow = T, nrow = 3))
```

Given that today it is a nice day, the corresponding stochastic row vector is $w_0 = (0\,, 1\,, 0)$ and the forecast after 1, 2 and 3 days are given by

```
R> W0 <- t(as.matrix(c(0, 1, 0)))
R> W1 <- W0 * mcWP; W1

     rainy nice snowy
[1,]   0.5    0   0.5

R> W2 <- W0 * (mcWP ^ 2); W2

     rainy nice snowy
[1,] 0.375 0.25 0.375

R> W3 <- W0 * (mcWP ^ 3); W3

       rainy   nice   snowy
[1,] 0.40625 0.1875 0.40625
```

As can be seen from w_1, if in the Land of Oz today is a nice day, tomorrow it will rain or snow with probability 1. One week later, the prediction can be computed as

```
R> W7 <- W0 * (mcWP ^ 7)
R> W7

         rainy      nice     snowy
[1,] 0.4000244 0.1999512 0.4000244
```

The steady state of the chain can be computed by means of the `steadyStates` method.

```
R> q <- steadyStates(mcWP)
R> q

     rainy nice snowy
[1,]   0.4  0.2   0.4
```

Note that, from the seventh day on, the predicted probabilities are substantially equal to the steady state of the chain and do not depend on the starting point, as the following code shows.

```
R> R0 <- t(as.matrix(c(1, 0, 0)))
R> R7 <- R0 * (mcWP ^ 7); R7

         rainy      nice     snowy
[1,] 0.4000244 0.2000122 0.3999634

R> S0 <- t(as.matrix(c(0, 0, 1)))
R> S7 <- S0 * (mcWP ^ 7); S7

         rainy      nice     snowy
[1,] 0.3999634 0.2000122 0.4000244
```

Alofi Island Rainfall

Alofi Island daily rainfall data were recorded from January 1st, 1987 until December 31st, 1989 and classified into three states: "0" (no rain), "1-5" (from non zero until 5 mm) and "6+" (more than 5mm). The corresponding dataset is provided within the **markovchain** package.

```
R> data("rain", package = "markovchain")
R> table(rain$rain)


  0 1-5  6+
548 295 253
```

The underlying transition matrix is estimated as follows.

```
R> mcAlofi <- markovchainFit(data = rain$rain, name = "Alofi MC")$estimate
R> mcAlofi

Alofi MC
 A  3 - dimensional discrete Markov Chain defined by the following states:
 0, 1-5, 6+
 The transition matrix  (by rows)  is defined as follows:
            0       1-5        6+
0   0.6605839 0.2299270 0.1094891
1-5 0.4625850 0.3061224 0.2312925
6+  0.1976285 0.3122530 0.4901186
```

The long term daily rainfall distribution is obtained by means of the `steadyStates` method.

```
R> steadyStates(mcAlofi)
```

```
             0       1-5        6+
[1,] 0.5008871 0.2693656 0.2297473
```

7.2. Finance and Economics

Other relevant applications of DTMC can be found in Finance and Economics.

Finance

Credit ratings transitions have been successfully modeled with discrete time Markov chains. Some rating agencies publish transition matrices that show the empirical transition probabilities across credit ratings. The example that follows comes from **CreditMetrics** R package (Wittmann 2007), carrying Standard & Poor's published data.

```
R> rc <- c("AAA", "AA", "A", "BBB", "BB", "B", "CCC", "D")
R> creditMatrix <- matrix(
+   c(90.81, 8.33, 0.68, 0.06, 0.08, 0.02, 0.01, 0.01,
+     0.70, 90.65, 7.79, 0.64, 0.06, 0.13, 0.02, 0.01,
+     0.09, 2.27, 91.05, 5.52, 0.74, 0.26, 0.01, 0.06,
+     0.02, 0.33, 5.95, 85.93, 5.30, 1.17, 1.12, 0.18,
+     0.03, 0.14, 0.67, 7.73, 80.53, 8.84, 1.00, 1.06,
+     0.01, 0.11, 0.24, 0.43, 6.48, 83.46, 4.07, 5.20,
+     0.21, 0, 0.22, 1.30, 2.38, 11.24, 64.86, 19.79,
+     0, 0, 0, 0, 0, 0, 0, 100
+    )/100, 8, 8, dimnames = list(rc, rc), byrow = TRUE)
```

It is easy to convert such matrices into `markovchain` objects and to perform some analyses

```
R> creditMc <- new("markovchain", transitionMatrix = creditMatrix,
+                  name = "S&P Matrix")
R> absorbingStates(creditMc)
```

```
[1] "D"
```

Economics

For a recent application of **markovchain** in economics, see Jacob (2014).

A dynamic system generates two kinds of economic effects (Bard 2000):

1. those incurred when the system is in a specified state, and
2. those incurred when the system makes a transition from one state to another.

Let the monetary amount of being in a particular state be represented as an m-dimensional column vector c^{S}, while let the monetary amount of a transition be embodied in a C^R matrix in which each component specifies the monetary amount of going from state i to state j in a single step. Equation (25) then gives the expected monetary amount associated with being in state i.

$$c_i = c_i^{\mathrm{S}} + \sum_{j=1}^{m} C_{ij}^{\mathrm{R}} p_{ij}. \tag{25}$$

Let $\bar{c} = [c_i]$ and let e_i be the vector valued 1 in the initial state and 0 in all other, then, if f_n is the random variable representing the economic return associated with the stochastic process at time n, Equation (26) holds:

$$E\left[f_n\left(X_n\right)|X_0 = i\right] = e_i P^n \bar{c}. \tag{26}$$

The following example assumes that a telephone company models the transition probabilities between customer/non-customer status by matrix P and the cost associated to states by matrix M.

```
R> statesNames <- c("customer", "non customer")
R> P <- markovchain:::zeros(2); P[1, 1] <- .9; P[1, 2] <- .1; P[2, 2] <- .95; P[2, 1] <- .
R> rownames(P) <- statesNames; colnames(P) <- statesNames
R> mcP <- new("markovchain", transitionMatrix = P, name = "Telephone company")
R> M <- markovchain:::zeros(2); M[1, 1] <- -20; M[1, 2] <- -30; M[2, 1] <- -40; M[2, 2] <-
```

If the average revenue for existing customer is +100, the cost per state is computed as follows.

```
R> c1 <- 100 + conditionalDistribution(mcP, state = "customer") %*% M[1,]
R> c2 <- 0 + conditionalDistribution(mcP, state = "non customer") %*% M[2,]
```

For an existing customer, the expected gain (loss) at the fifth year is given by the following code.

```
R> as.numeric((c(1, 0)* mcP ^ 5) %*% (as.vector(c(c1, c2))))
```

```
[1] 48.96009
```

Discretizing autoregressive processes

Interest rate, inflation, or income processes used in finance, economics and actuarial reserving/ALM work are frequently modelled as a stationary AR(1),

$$y_t = (1-\rho)\alpha + \rho\, y_{t-1} + \varepsilon_t, \qquad \varepsilon_t \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2),$$

which is not itself a (finite-state) Markov chain. `tauchen()` and `rouwenhorst()` discretize such a process onto a finite grid, returning both the discretized `markovchain` object and the

numeric grid `states` itself (needed for any further numeric work, e.g. plugging the levels into a pricing formula). `tauchen()` (Tauchen 1986) uses an evenly spaced grid of fixed half-width $k\sigma_y$ ($\sigma_y = \sigma/\sqrt{1-\rho^2}$) and bins a Normal conditional distribution into it; `rouwenhorst()` (Rouwenhorst 1995) instead builds the transition matrix recursively from $\theta = (1+\rho)/2$, with no grid-width parameter to choose.

Kopecky and Suen (2010) show that for highly persistent processes – their example is a quarterly TFP growth process calibrated at $\rho = 0.979$, $\sigma_\varepsilon = 0.0072$ – Rouwenhorst's method matches the process's theoretical unconditional variance $\sigma_y^2 = \sigma^2/(1-\rho^2)$ almost exactly at every chain size they try, while Tauchen's method is markedly less accurate at the same sizes and only improves slowly as the chain grows. Both `tauchen()` and `rouwenhorst()` reproduce that comparison directly:

```
R> rho <- 0.979
R> sigmaEps <- 0.0072
R> theoreticalVar <- sigmaEps^2 / (1 - rho^2)
R> theoreticalVar

[1] 0.001247383

R> relativeError <- function(size) {
+   outT <- tauchen(alpha = 0, sigma = sigmaEps, rho = rho, size = size)
+   outR <- rouwenhorst(alpha = 0, sigma = sigmaEps, rho = rho, size = size)
+   piT <- as.numeric(steadyStates(outT$chain))
+   piR <- as.numeric(steadyStates(outR$chain))
+   varT <- sum(piT * (outT$states - sum(piT * outT$states)) ^ 2)
+   varR <- sum(piR * (outR$states - sum(piR * outR$states)) ^ 2)
+   c(tauchen = 100 * abs(varT - theoreticalVar) / theoreticalVar,
+     rouwenhorst = 100 * abs(varR - theoreticalVar) / theoreticalVar)
+ }
R>
R> sapply(c(5, 10, 25), relativeError)

                    [,1]         [,2]         [,3]
tauchen     8.139476e+01 4.979587e+01 9.328706e+00
rouwenhorst 1.738363e-13 1.408074e-12 7.301123e-13
```

For every chain size shown, `rouwenhorst()`'s discretized variance matches the theoretical AR(1) variance to numerical precision (percentage error effectively 0), while `tauchen()`'s fixed-width grid is off by tens of percent at small sizes and still off by nearly 10% at 25 states – worse, at 25 states, than `rouwenhorst()` already is at only 5 – matching Kopecky and Suen's own qualitative conclusion.

Quadrature-based discretizations A third family of methods replaces the integral that defines the next period's expectations with a quadrature rule. If the process moves from x to y with density $f(x, y)$, and $y_1, \ldots, y_n$ and $w_1, \ldots, w_n$ are the nodes and weights of a rule for $\int g(y)\, dy$, then

$$P_{ij} = \frac{w_j f(y_i, y_j)}{\sum_k w_k f(y_i, y_k)}$$

is a transition matrix whose conditional expectations $\sum_j P_{ij} h(y_j)$ approximate $E[h(Y_{t+1}) \mid Y_t = y_i]$ (Tauchen and Hussey 1991). `fitFunction()` builds this chain for any kernel f and any of seven rules with positive weights (Gauss-Legendre, Gauss-Chebyshev, Fejér, Gauss-Hermite, trapezoid, Simpson and midpoint); the Gauss rules are computed from the eigenvalues of the Jacobi matrix (Golub and Welsch 1969). It returns the chain, the nodes and the weights.

`tauchenHussey()` is the special case of an AR(1) with Gauss-Hermite nodes for a normal distribution of standard deviation $\hat{\sigma}$. Tauchen and Hussey take $\hat{\sigma} = \sigma$, the conditional standard deviation, which concentrates the nodes too much when the process is persistent; Flodén (2008) proposes the weighted choice $\hat{\sigma} = w\sigma + (1-w)\sigma_y$ with $w = 1/2 + \rho/4$ (`sigmaWeight = "floden"`), and also reports $\hat{\sigma} = \sigma_y$ (`sigmaWeight = "unconditional"`). His Table 1 gives, for the income process labelled HSZ ($\rho = 0.95$, $\sigma_\varepsilon^2 = 0.030$, so that $\sigma_\varepsilon = 0.1732$ and $\sigma_y = 0.5547$), the conditional and unconditional standard deviations implied by each chain. The chains built here reproduce all twelve values:

```
R> impliedSd <- function(out) {
+   P <- out$chain@transitionMatrix
+   z <- out$states
+   p <- as.numeric(steadyStates(out$chain))
+   condVar <- P %*% z^2 - (P %*% z)^2
+   c(sigmaEps = sqrt(sum(p * condVar)),
+     sigmaY = sqrt(sum(p * (z - sum(p * z))^2)))
+ }
R> designs <- expand.grid(size = c(5, 9),
+                         sigmaWeight = c("conditional", "unconditional", "floden"),
+                         stringsAsFactors = FALSE)
R> ours <- t(mapply(function(n, s)
+   impliedSd(tauchenHussey(alpha = 0, sigma = sqrt(0.03), rho = 0.95,
+                           size = n, sigmaWeight = s)),
+   designs$size, designs$sigmaWeight))
R> floden <- rbind(c(0.1576, 0.3275), c(0.1670, 0.4303),
+                  c(0.0101, 0.5622), c(0.0561, 0.5556),
+                  c(0.1410, 0.4792), c(0.1692, 0.5407))
R> colnames(floden) <- c("sigmaEps (Floden)", "sigmaY (Floden)")
R> cbind(designs, round(ours, 4), floden)
```

```
  size   sigmaWeight sigmaEps sigmaY sigmaEps (Floden) sigmaY (Floden)
1    5   conditional   0.1576 0.3275            0.1576          0.3275
2    9   conditional   0.1670 0.4303            0.1670          0.4303
3    5 unconditional   0.0101 0.5622            0.0101          0.5622
4    9 unconditional   0.0561 0.5556            0.0561          0.5556
5    5        floden   0.1410 0.4792            0.1410          0.4792
6    9        floden   0.1692 0.5407            0.1692          0.5407
```

The original choice understates the unconditional standard deviation by 41% with five nodes, the unconditional one makes the chain almost deterministic (a conditional standard deviation

of 0.0101 instead of 0.1732), and Flodén's weighting is the best compromise of the three. The same table also reports the autocorrelation implied by each chain, which is not reproduced here: the paper does not say how it is computed, and the lag-one autocorrelation under the stationary distribution of these chains does not match it when the approximation is coarse (0.8757 against 0.9073 for the original method with five nodes; 0.5976 against 0.5992 for the $\rho = 0.6$ process of the same table). The standard deviations of the $\rho = 0.6$ process are reproduced too, and both sets are in the package's tests.

`fitFunction()` is not restricted to Gaussian processes. A proportion, such as a loss ratio or an occupancy rate, can be modelled on $[0, 1]$ with a Beta transition density whose mean moves linearly with the current value, $Y_{t+1} \mid Y_t = x \sim \text{Beta}(\kappa m(x), \kappa(1-m(x)))$ with $m(x) = 0.2+0.6x$. Its stationary mean is 0.5 and, since $\text{Var}(Y_{t+1} \mid Y_t = x) = m(x)(1-m(x))/(\kappa+1)$, its stationary variance solves $v = 0.36v + (0.25 - 0.36v)/(\kappa + 1)$. The chains built with two rules approach it as the number of nodes grows:

```
R> kappa <- 20
R> m <- function(x) 0.2 + 0.6 * x
R> betaKernel <- function(x, y) dbeta(y, kappa * m(x), kappa * (1 - m(x)))
R> stationaryVariance <- function(out) {
+   p <- as.numeric(steadyStates(out$chain))
+   sum(p * (out$states - sum(p * out$states))^2)
+ }
R> (trueVariance <- (0.25 / (kappa + 1)) / (1 - 0.36 + 0.36 / (kappa + 1)))

[1] 0.01811594

R> sapply(c(5, 10, 20), function(n)
+   c(nodes = n,
+     gaussLegendre = stationaryVariance(fitFunction(betaKernel, n)),
+     trapezoid = stationaryVariance(fitFunction(betaKernel, n, "trapezoid"))))

                    [,1]        [,2]        [,3]
nodes         5.00000000 10.00000000 20.00000000
gaussLegendre 0.01245155  0.01811544  0.01811594
trapezoid     0.01771685  0.01811055  0.01811588
```

With five nodes Gauss-Legendre places no node close enough to the end points and misses almost a third of the variance, while the trapezoid rule is already within 3%; from ten nodes on both are within 0.03% of the exact value. For the same kernel, interval and number of nodes the Gauss-Legendre, Fejér, trapezoid and Simpson rules give the same matrices as `fit_function` in **PyDTMC**.

7.3. Actuarial science

Markov chains are widely applied in the field of actuarial science. Two classical applications are policyholders' distribution across Bonus Malus classes in Motor Third Party Liability (MTPL) insurance (Section 7.3.1) and health insurance pricing and reserving (Section 7.3.2).

MTPL Bonus Malus

Bonus Malus (BM) contracts grant the policyholder a discount (bonus) or impose a surcharge (malus) as a function of the number of claims in the experience period. The discount or surcharge is applied to a premium that already allows for known (a priori) policyholder characteristics (Denuit, Maréchal, Pitrebois, and Walhin 2007), which usually depend on the vehicle, the territory, the demographic profile of the policyholder, and the policy coverage (deductible and policy limits).\ Since the BM level depends only on the BM level and on the claims of the previous period, it can be modeled by a discrete Markov chain. A very simplified example follows. Assume a BM scale from 1 to 5, where 4 is the starting level. The evolution rules are shown in Equation 27:

$$bm_{t+1} = \max\left(1, bm_t - 1\right) * \left(\tilde{N} = 0\right) + \min\left(5, bm_t + 2 * \tilde{N}\right) * \left(\tilde{N} \geq 1\right). \tag{27}$$

The number of claims $\tilde{N}$ is a random variable that is assumed to be Poisson distributed.

```
R> getBonusMalusMarkovChain <- function(lambda) {
+   bmMatr <- markovchain:::zeros(5)
+   bmMatr[1, 1] <- dpois(x = 0, lambda)
+   bmMatr[1, 3] <- dpois(x = 1, lambda)
+   bmMatr[1, 5] <- 1 - ppois(q = 1, lambda)
+
+   bmMatr[2, 1] <- dpois(x = 0, lambda)
+   bmMatr[2, 4] <- dpois(x = 1, lambda)
+   bmMatr[2, 5] <- 1 - ppois(q = 1, lambda)
+
+   bmMatr[3, 2] <- dpois(x = 0, lambda)
+   bmMatr[3, 5] <- 1 - dpois(x=0, lambda)
+
+   bmMatr[4, 3] <- dpois(x = 0, lambda)
+   bmMatr[4, 5] <- 1 - dpois(x = 0, lambda)
+
+   bmMatr[5, 4] <- dpois(x = 0, lambda)
+   bmMatr[5, 5] <- 1 - dpois(x = 0, lambda)
+   stateNames <- as.character(1:5)
+   out <- new("markovchain", transitionMatrix = bmMatr,
+              states = stateNames, name = "BM Matrix")
+   return(out)
+ }
```

Assuming that the a-priori claim frequency per car-year is 0.05 in the class (being the class the group of policyholders that share the same common characteristics), the underlying BM transition matrix and its underlying steady state are as follows.

```
R> bmMc <- getBonusMalusMarkovChain(0.05)
R> as.numeric(steadyStates(bmMc))

[1] 0.895836079 0.045930498 0.048285405 0.005969247 0.003978772
```

If the underlying BM coefficients of the class are 0.5, 0.7, 0.9, 1.0, 1.25, this means that the average BM coefficient applied on the long run to the class is given by

```
R> sum(as.numeric(steadyStates(bmMc)) * c(0.5, 0.7, 0.9, 1, 1.25))

[1] 0.534469
```

This means that the average premium paid by policyholders in the portfolio almost halves in the long run.

Health insurance example

Actuaries quantify the risk inherent in insurance contracts evaluating the premium of insurance contract to be sold (therefore covering future risk) and evaluating the actuarial reserves of existing portfolios (the liabilities in terms of benefits or claims payments due to policyholder arising from previously sold contracts), see Deshmukh (2012) for details.

An applied example can be performed using the data from De Angelis, Paolo and Di Falco, L. (2016) that has been saved in the `exdata` folder.

```
R> ltcDemoPath<-system.file("extdata", "ltdItaData.txt",
+                           package = "markovchain")
R> ltcDemo<-read.table(file = ltcDemoPath, header=TRUE,
+                       sep = ";", dec = ".")
R> head(ltcDemo)

  age          pAD        pID          pAI       pAA
1  20 0.0004616002 0.01083364 0.0001762467 0.9993622
2  21 0.0004824888 0.01079719 0.0001710577 0.9993465
3  22 0.0004949938 0.01177076 0.0001592333 0.9993458
4  23 0.0005042935 0.01159394 0.0001605731 0.9993351
5  24 0.0005074193 0.01260574 0.0001606504 0.9993319
6  25 0.0005154267 0.01526364 0.0001643603 0.9993202
```

The data shows the probability of transition between the state of (A)ctive, to (I)ll and Dead. It is easy to complete the transition matrix.

```
R> ltcDemo<-transform(ltcDemo,
+                     pIA=0,
+                     pII=1-pID,
+                     pDD=1,
+                     pDA=0,
+                     pDI=0)
```

Now we build a function that returns the transition during the $t+1$ th year, assuming that the subject has attained year t.

```
R> possibleStates<-c("A","I","D")
R> getMc4Age<-function(age) {
+   transitionsAtAge<-ltcDemo[ltcDemo$age==age,]
+
+   myTransMatr<-matrix(0, nrow=3,ncol = 3,
+                       dimnames = list(possibleStates, possibleStates))
+   myTransMatr[1,1]<-transitionsAtAge$pAA[1]
+   myTransMatr[1,2]<-transitionsAtAge$pAI[1]
+   myTransMatr[1,3]<-transitionsAtAge$pAD[1]
+   myTransMatr[2,2]<-transitionsAtAge$pII[1]
+   myTransMatr[2,3]<-transitionsAtAge$pID[1]
+   myTransMatr[3,3]<-1
+
+   myMc<-new("markovchain", transitionMatrix = myTransMatr,
+             states = possibleStates,
+             name = paste("Age",age,"transition matrix"))
+
+   return(myMc)
+
+ }
```

Because transitions are not homogeneous across ages, we use a `markovchainList` object to describe the transition probabilities of an individual starting at age 100.

```
R> getFullTransitionTable<-function(age){
+   ageSequence<-seq(from=age, to=120)
+   k=1
+   myList=list()
+   for ( i in ageSequence) {
+     mc_age_i<-getMc4Age(age = i)
+     myList[[k]]<-mc_age_i
+     k=k+1
+   }
+   myMarkovChainList<-new("markovchainList", markovchains = myList,
+                          name = paste("TransitionsSinceAge", age, sep = ""))
+   return(myMarkovChainList)
+ }
R> transitionsSince100<-getFullTransitionTable(age=100)
```

We can use such transition for simulating ten life trajectories for an individual who begins "active" (A) at age 100:

```
R> rmarkovchain(n = 10, object = transitionsSince100,
+               what = "matrix", t0 = "A", include.t0 = TRUE)

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
 [1,] "A"  "A"  "A"  "A"  "D"  "D"  "D"  "D"  "D"  "D"   "D"   "D"   "D"
```

```
 [2,] "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "D"   "D"   "D"   "D"
 [3,] "A"  "A"  "D"  "D"  "D"  "D"  "D"  "D"  "D"  "D"   "D"   "D"   "D"
 [4,] "A"  "A"  "D"  "D"  "D"  "D"  "D"  "D"  "D"  "D"   "D"   "D"   "D"
 [5,] "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"   "A"   "A"   "D"
 [6,] "A"  "A"  "A"  "A"  "D"  "D"  "D"  "D"  "D"  "D"   "D"   "D"   "D"
 [7,] "A"  "A"  "A"  "A"  "A"  "D"  "D"  "D"  "D"  "D"   "D"   "D"   "D"
 [8,] "A"  "D"  "D"  "D"  "D"  "D"  "D"  "D"  "D"  "D"   "D"   "D"   "D"
 [9,] "A"  "D"  "D"  "D"  "D"  "D"  "D"  "D"  "D"  "D"   "D"   "D"   "D"
[10,] "A"  "A"  "A"  "D"  "D"  "D"  "D"  "D"  "D"  "D"   "D"   "D"   "D"
      [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22]
 [1,] "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"
 [2,] "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"
 [3,] "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"
 [4,] "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"
 [5,] "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"
 [6,] "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"
 [7,] "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"
 [8,] "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"
 [9,] "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"
[10,] "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"   "D"
```

Let us consider 100 simulated life trajectories for a healthy individual aged 80. We can compute the expected time the individual is disabled, starting from the active state at age 80.

```
R> transitionsSince80<-getFullTransitionTable(age=80)
R> lifeTrajectories<-rmarkovchain(n=1e2, object=transitionsSince80,
+                                 what="matrix",t0="A",include.t0=TRUE)
R> temp<-matrix(0,nrow=nrow(lifeTrajectories),ncol = ncol(lifeTrajectories))
R> temp[lifeTrajectories=="I"]<-1
R> expected_period_disabled<-mean(rowSums((temp)))
R> expected_period_disabled

[1] 0.86
```

Assuming that the health insurance will pay a benefit of 12000 per year disabled and that the real interest rate is 0.02, we can compute the lump-sum premium at 80.

```
R> mean(rowMeans(12000*temp%*%( matrix((1+0.02)^-seq(from=0, to=ncol(temp)-1)))))

[1] 8576.511
```

7.4. Sociology

Markov chains have been actively used to model progressions and regressions between social classes. The first study was performed by Glass and Hall (1954), while a more recent application can be found in Jo Blanden and Machin (2005). The table that follows shows the

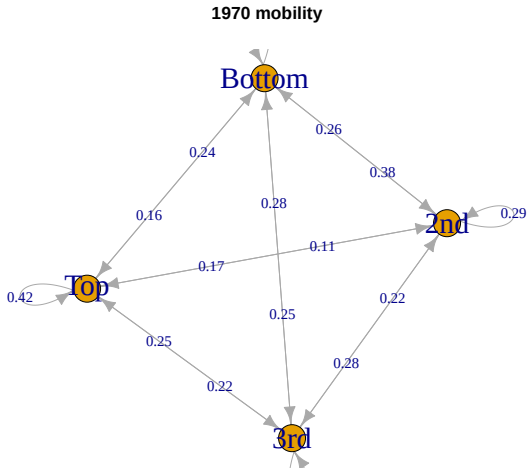


Figure 9: 1970 UK cohort mobility data.

income quartile of the father when the son was 16 (in 1984) and the income quartile of the son when aged 30 (in 2000) for the 1970 cohort.

```
R> data("blanden")
R> mobilityMc <- as(blanden, "markovchain")
R> mobilityMc
```

```
Unnamed Markov chain
 A  4 - dimensional discrete Markov Chain defined by the following states:
 Bottom, 2nd, 3rd, Top
 The transition matrix  (by rows)  is defined as follows:
            2nd       3rd    Bottom       Top
Bottom 0.2900000 0.2200000 0.3800000 0.1100000
2nd    0.2772277 0.2574257 0.2475248 0.2178218
3rd    0.2626263 0.2828283 0.2121212 0.2424242
Top    0.1700000 0.2500000 0.1600000 0.4200000
```

The underlying transition graph is plotted in Figure 9.

The steady state distribution is computed as follows. Since the states are income quartiles, the long-run distribution is (approximately) uniform: each quartile has probability about 0.25.

```
R> round(steadyStates(mobilityMc), 2)
```

```
     Bottom  2nd  3rd  Top
[1,]   0.25 0.25 0.25 0.25
```

7.5. Genetics and Medicine

This section contains two examples: the first shows the use of Markov chain models in genetics, the second shows an application of Markov chains in modelling diseases' dynamics.

Genetics

P. J. Avery and D. A. Henderson (1999) discuss the use of Markov chains to model the sequence of the bases of the preproglucacon gene. The `preproglucacon` dataset of the **markovchain** package contains this sequence.

```
R> data("preproglucacon", package = "markovchain")
```

It is possible to model the transition probabilities between bases as shown in the following code.

```
R> mcProtein <- markovchainFit(preproglucacon$preproglucacon,
+                              name = "Preproglucacon MC")$estimate
R> mcProtein

Preproglucacon MC
 A  4 - dimensional discrete Markov Chain defined by the following states:
 A, C, G, T
 The transition matrix  (by rows)  is defined as follows:
          A         C          G         T
A 0.3585271 0.1434109 0.16666667 0.3313953
C 0.3840304 0.1558935 0.02281369 0.4372624
G 0.3053097 0.1991150 0.15044248 0.3451327
T 0.2844523 0.1819788 0.17667845 0.3568905
```

Medicine

Discrete-time Markov chains are also employed to study the progression of chronic diseases. The following example is taken from B. A. Craig and A. A. Sendi (2002). Starting from six month follow-up data, the maximum likelihood estimation of the monthly transition matrix is obtained. This transition matrix aims to describe the monthly progression of CD4-cell counts of HIV infected subjects.

```
R> craigSendiMatr <- matrix(c(682, 33, 25,
+                154, 64, 47,
+                19, 19, 43), byrow = T, nrow = 3)
R> hivStates <- c("0-49", "50-74", "75-UP")
R> rownames(craigSendiMatr) <- hivStates
R> colnames(craigSendiMatr) <- hivStates
R> craigSendiTable <- as.table(craigSendiMatr)
R> mcM6 <- as(craigSendiTable, "markovchain")
R> mcM6@name <- "Zero-Six month CD4 cells transition"
R> mcM6
```

```
Zero-Six month CD4 cells transition
 A  3 - dimensional discrete Markov Chain defined by the following states:
 0-49, 50-74, 75-UP
 The transition matrix  (by rows)  is defined as follows:
           0-49      50-74      75-UP
0-49  0.9216216 0.04459459 0.03378378
50-74 0.5811321 0.24150943 0.17735849
75-UP 0.2345679 0.23456790 0.53086420
```

As shown in the paper, the second passage consists in the decomposition of $M_6 = V \cdot D \cdot V^{-1}$ in order to obtain M_1 as $M_1 = V \cdot D^{1/6} \cdot V^{-1}$.

```
R> eig <- eigen(mcM6@transitionMatrix)
R> D <- diag(eig$values)

R> V <- eig$vectors
R> V %*% D %*% solve(V)

          [,1]       [,2]       [,3]
[1,] 0.9216216 0.04459459 0.03378378
[2,] 0.5811321 0.24150943 0.17735849
[3,] 0.2345679 0.23456790 0.53086420

R> d <- D ^ (1/6)
R> M <- V %*% d %*% solve(V)
R> mcM1 <- new("markovchain", transitionMatrix = M, states = hivStates)
```

8. Discussion, issues and future plans

The **markovchain** package has been designed to make it easy to handle DTMC and to exchange them with other packages. Since its first release it has grown in several directions. The main computations are implemented in C++ through **Rcpp** (Eddelbuettel 2013), and the simulation and bootstrap routines can run on several threads with reproducible results. The probabilistic toolbox now includes hitting probabilities, with a choice of solvers for large chains, spectral diagnostics, mixing times, reversibility and time correlations. The statistical toolbox includes several estimation methods, the tests of independence, of the Markov property, of the order and of stationarity, the Bayesian estimators and the aggregation of states. Chains can be generated at random, saved to and read from JSON, YAML, CSV and XML files, and exported as Graphviz or Mermaid graphs. Higher order and multivariate Markov chains, and the other additions of the Google Summer of Code 2017, are described in the companion vignettes.

The package has several known limits. The transition matrices are stored in memory, so very large state spaces require sparse matrices and, for some functions, a specialised algorithm; the continuous time functions assume a finite state space; and the Bayesian estimation covers first order chains only. Hidden Markov chains are outside the scope of the package, for which

the packages mentioned in the introduction are available. Bug reports and suggestions are welcome on the project page, https://github.com/spedygiorgio/markovchain.

9. Acknowledgments

The package was selected for Google Summer of Code support in 2015 and 2017. The authors wish to thank Michael Cole, Tobi Gutman and Mildenberger Thoralf for their suggestions and bug checks. A final thanks also to Dr. Simona C. Minotti and Dr. Mirko Signorelli for their support in drafting this version of the vignettes.

Method	Returns
`absorbingStates`	the absorbing states of the transition matrix, if any.
`conditionalDistribution`	distribution of the next state given the current one.
`steadyStates`	the vector(s) of steady state(s) in matrix form.
`redistribute`	distribution of the chain after a number of steps.
`entropyRate`	entropy rate of an irreducible Markov chain.
`topologicalEntropy`	growth rate of the number of admissible paths.
`normalizedEntropyRate`	ratio of the entropy rate to the topological entropy.
`firstPassage`	distribution of the first passage time within n steps.
`meanFirstPassageTime`	matrix or vector of mean first passage times.
`kemenyConstant`	stationary-weighted mean hitting time for an irreducible chain.
`slem`	second largest eigenvalue modulus (irreducible).
`spectralGap`	one minus the SLEM, a lightweight mixing diagnostic.
`impliedTimescales`	per-eigenvalue relaxation timescales (irreducible).
`timeCorrelations`, `timeRelaxations`	time correlations and relaxations of observables defined on the states.
`sensitivity`	sensitivity of the stationary distribution to a perturbation of a row.
`is.reversible`	checks detailed balance w.r.t. the stationary distribution.
`closestReversible`	closest reversible chain for a fixed stationary law.
`mixingTime`	total-variation mixing time (irreducible, aperiodic).
`lazyChain`	the chain $\alpha I + (1-\alpha)P$, used to remove periodicity.
`subchain`	restriction to a subset of states (submatrix or renormalized).
`meanRecurrenceTime`	vector of mean number of steps to return to each recurrent state.
`hittingProbabilities`	matrix of hitting probabilities for a Markov chain.
`meanAbsorptionTime`	expected number of steps for a transient state to be absorbed by any recurrent class.
`fundamentalMatrix`	fundamental matrix of an absorbing chain.
`absorptionProbabilities`	probabilities of transient states of being absorbed by each recurrent state.
`committorAB`	committor probabilities.
`communicatingClasses`	list of communicating classes.
`canonicForm`	the transition matrix into canonic form.
`is.accessible`	checks whether a state j is reachable from state i.
`is.irreducible`	checks whether a DTMC is irreducible.
`is.regular`	checks whether a DTMC is regular.
`is.stochasticallyMonotone`	checks whether a DTMC (or a Matrix) is

Function	Purpose
`rmarkovchain`	samples from `markovchain` or `markovchainList` objects.
`randomMarkovChain`, `dirichletChain`, `identityChain`	generate random and special chains.
`is.lumpable`, `lump`, `aggregateStates`	lumpability and aggregation of states.
`toFile`, `fromFile`, `toDictionary`, `fromDictionary`	export and import of chains.
`toDot`, `toMermaid`	export of the transition graph as text.
`markovchainFit`	fits a Markov chain to a given sequence (maximum likelihood, Laplace, bootstrap, MAP).
`predict`	predictions from `markovchain` or `markovchainList` objects.
`assessIndependence`, `verifyMarkovProperty`, `assessOrder`, `assessStationarity`	tests on a sequence of states.
`verifyEmpiricalToTheoretical`, `verifyHomogeneity`	divergence tests for transition matrices.
`inferHyperparam`, `predictiveDistribution`	Bayesian estimation.

Table 5: The **markovchain** statistical functions.

past	present	future a	future b
a	a	2	2
b	a	2	2

Table 6: Contingency table to assess the order for the present state a.

time (t)	probability of transition to a	probability of transition to b
1	0	1
2	0	1
.	.	.
.	.	.
.	.	.
16	0.44	0.56

Table 7: Contingency table to assess the stationarity of the state a.

References

Allaire J, Francois R, Ushey K, Vandenbrouck G, Geelnard M, Intel (2016). *RcppParallel: Parallel Programming Tools for 'Rcpp'*. R package version 4.3.19, URL https://rcppcore.github.io/RcppParallel/.

Anderson TW, Goodman LA (1957). "Statistical inference about Markov chains." *The Annals of Mathematical Statistics*, pp. 89–110.

B A Craig, A A Sendi (2002). "Estimation of the Transition Matrix of a Discrete-Time Markov Chain." *Health Economics*, **11**, 33–42.

Bard JF (2000). "Lecture 12.5 - Additional Issues Concerning Discrete-Time Markov Chains."

Basawa IV, Green TA, McCormick WP, Taylor RL (1990). "Asymptotic Bootstrap Validity for Finite Markov Chains." *Communications in Statistics – Theory and Methods*, **19**(4), 1493–1510.

Berchtold A, Raftery AE (2002). "The Mixture Transition Distribution Model for High-Order Markov Chains and Non-Gaussian Time Series." *Statistical Science*, **17**(3), 328–356.

Brémaud P (1999). "Discrete-Time Markov Models." In *Markov Chains*, pp. 53–93. Springer.

Brown LD, Cai TT, DasGupta A (2001). "Interval Estimation for a Binomial Proportion." *Statistical Science*, **16**(2), 101–133. doi:10.1214/ss/1009213286.

Burkett MW, Scherer WT, Todd A (2019). "Are financial market states recurrent and persistent?" *Cogent Economics and Finance*, **7**(1), 1622171. doi:10.1080/23322039.2019.1622171.

Chambers J (2008). *Software for Data Analysis: Programming with R*. Statistics and computing. Springer-Verlag. ISBN 9780387759357.

Ching WK, Ng MK, Fung ES (2008). "Higher-order multivariate Markov chains and their applications." *Linear Algebra and its Applications*, **428**(2), 492–507.

Cho GE, Meyer CD (2001). "Comparison of perturbation bounds for the stationary distribution of a Markov chain." *Linear Algebra and its Applications*, **335**(1–3), 137–150. doi:10.1016/S0024-3795(01)00320-2.

Cover TM, Thomas JA (2006). *Elements of Information Theory*. 2 edition. Wiley-Interscience, Hoboken, NJ. doi:10.1002/047174882X.

Csardi G, Nepusz T (2006). "The **igraph** Software Package for Complex Network Research." *InterJournal*, **Complex Systems**, 1695.

Davison AC, Hinkley DV (1997). *Bootstrap Methods and Their Application*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press. doi:10.1017/CBO9780511802843.

De Angelis, Paolo, Di Falco, L (2016). *Assicurazioni sulla salute: caratteristiche, modelli attuariali e basi tecniche*. Il Mulino. Il Mulino. ISBN 9788815260840. URL https://books.google.it/books?id=D56bjgEACAAJ.

de Wreede LC, Fiocco M, Putter H (2011). "**mstate**: An R Package for the Analysis of Competing Risks and Multi-State Models." *Journal of Statistical Software*, **38**(7), 1–30. URL https://www.jstatsoft.org/v38/i07/.

Deng K, Mehta PG, Meyn SP (2009). "A Simulation-Based Method for Aggregating Markov Chains." In *Proceedings of the 48th IEEE Conference on Decision and Control (CDC)*.

Deng K, Mehta PG, Meyn SP (2011). "Optimal Kullback-Leibler Aggregation via Spectral Theory of Markov Chains." *IEEE Transactions on Automatic Control*, **56**(12), 2793–2808. doi:10.1109/TAC.2011.2141350.

Denuit M, Maréchal X, Pitrebois S, Walhin JF (2007). *Actuarial modelling of claim counts: Risk classification, credibility and bonus-malus systems*. Wiley.

Deshmukh S (2012). *Multiple Decrement Models in Insurance: An Introduction Using R*. SpringerLink : Bücher. Springer-Verlag. ISBN 9788132206590.

Dobrow RP (2016). *Introduction to Stochastic Processes with R*. John Wiley & Sons.

Eddelbuettel D (2013). *Seamless R and C++ Integration with **Rcpp***. Springer-Verlag, New York. ISBN 978-1-4614-6867-7.

Ehrenfest P, Ehrenfest T (1907). "Über zwei bekannte Einwände gegen das Boltzmannsche H-Theorem." *Physikalische Zeitschrift*, **8**, 311–314.

Ewens WJ (2004). *Mathematical Population Genetics I: Theoretical Introduction*. 2 edition. Springer, New York, NY.

Feres R (2007). "Notes for Math 450 MATLAB Listings for Markov Chains."

Fill JA (1991). "Eigenvalue bounds on convergence to stationarity for nonreversible Markov chains, with an application to the exclusion process." *The Annals of Applied Probability*, **1**(1). doi:10.1214/aoap/1177005981.

Flodén M (2008). "A Note on the Accuracy of Markov-Chain Approximations to Highly Persistent AR(1) Processes." *Economics Letters*, **99**(3), 516–520.

Gabadinho A, Ritschard G, Müller NS, Studer M (2011). "Analyzing and Visualizing State Sequences in R with TraMineR." *Journal of Statistical Software*, **40**(4), 1–37. doi:10.18637/jss.v040.i04.

Geyer CJ, Johnson LT (2013). *mcmc: Markov Chain Monte Carlo*. R package version 0.9-2, URL https://CRAN.R-project.org/package=mcmc.

Glass D, Hall JR (1954). "Social Mobility in Great Britain: A Study in Intergenerational Change in Status." In *Social Mobility in Great Britain*. Routledge and Kegan Paul.

Golub GH, Welsch JH (1969). "Calculation of Gauss Quadrature Rules." *Mathematics of Computation*, **23**(106), 221–230. doi:10.2307/2004418.

Goulet V, Dutang C, Maechler M, Firth D, Shapira M, Stadelmann M, expm-developers@listsR-forgeR-projectorg (2013). ***expm**: Matrix Exponential*. R package version 0.99-1, URL https://CRAN.R-project.org/package=expm.

Grinstead CM, Snell LJ (2006). *Grinstead and Snell's Introduction to Probability*. Version dated 4 july 2006 edition. American Mathematical Society.

Himmelmann SSDDL, wwwlinhicom (2010). ***HMM**: HMM - Hidden Markov Models*. R package version 1.0, URL https://CRAN.R-project.org/package=HMM.

Hsu D, Kontorovich A, Levin DA, Peres Y, Szepesvári C, Wolfer G (2019). "Mixing Time Estimation in Reversible Markov Chains from a Single Sample Path." *The Annals of Applied Probability*, **29**(4), 2439–2480. doi:10.1214/18-AAP1457.

Hu YT, Kiesel R, Perraudin W (2002). "The estimation of transition matrices for sovereign credit ratings." *Journal of Banking and Finance*, **26**(7), 1383–1406. ISSN 03784266. doi:10.1016/S0378-4266(02)00268-6.

Israel RB, Rosenthal JS, Wei JZ (2001). "Finding generators for Markov chains via empirical transition matrices, with applications to credit ratings." *Mathematical finance*, **11**(2), 245–265.

J G Kemeny, J LSnell, G L Thompson (1974). *Introduction to Finite Mathematics*. Prentice Hall.

Jackson CH (2011). "Multi-State Models for Panel Data: The **msm** Package for R." *Journal of Statistical Software*, **38**(8), 1–29. URL https://www.jstatsoft.org/v38/i08/.

Jacob I (2014). "Is R Cost Effective?" Electronic. Presented on Manchester R Meeting.

Jo Blanden PG, Machin S (2005). "Intergenerational Mobility in Europe and North America." *Technical report*, Center for Economic Performances.

Kemeny JG, Snell JL (1960). *Finite Markov Chains*. D. Van Nostrand, Princeton, NJ.

Konstantopoulos T (2009). "Markov Chains and Random Walks." Lecture notes.

Kopecky KA, Suen RMH (2010). "Finite state Markov-chain approximations to highly persistent processes." *Review of Economic Dynamics*, **13**(3), 701–714. doi:10.1016/j.red.2010.02.002.

Kullback S, Kupperman M, Ku H (1962). "Tests for Contingency Tables and Marltov Chains." *Technometrics*, **4**(4), 573–608.

Kulperger RJ, Prakasa Rao BLS (1989). "Bootstrapping a Finite State Markov Chain." *Sankhyā: The Indian Journal of Statistics, Series A*, **51**(2), 178–191.

Levin DA, Peres Y (2017). *Markov Chains and Mixing Times*. 2 edition. American Mathematical Society, Providence, RI. doi:10.1090/mbk/107.

Meyer CD (1980). "The condition of a finite Markov chain and perturbation bounds for the limiting probabilities." *SIAM Journal on Algebraic Discrete Methods*, **1**(3), 273–283. doi:10.1137/0601031.

Montgomery J (2009). "Communication Classes."

Moran PAP (1958). "Random processes in genetics." *Mathematical Proceedings of the Cambridge Philosophical Society*, **54**(1), 60–71. doi:10.1017/S0305004100033193.

Nicholson W (2013). ***DTMCPack**: Suite of Functions Related to Discrete-Time Discrete-State Markov Chains*. R package version 0.1-2, URL https://CRAN.R-project.org/package=DTMCPack.

Nielsen A, Weber M (2015). "Computing the nearest reversible Markov chain." *Numerical Linear Algebra with Applications*, **22**. doi:10.1002/nla.1967.

Noé F, Doose S, Daidone I, Löllmann M, Sauer M, Chodera JD, Smith JC (2011). "Dynamical Fingerprints for Probing Individual Relaxation Processes in Biomolecular Dynamics with Simulations and Kinetic Experiments." *Proceedings of the National Academy of Sciences*, **108**(12), 4822–4827. doi:10.1073/pnas.1004646108.

Norris JR (1998). *Markovchains*. Cambridge University Press.

Noé F, Schütte C, Vanden-Eijnden E, Reich L, Weikl TR (2009). "Constructing the equilibrium ensemble of folding pathways from short off-equilibrium simulations." *Proceedings of the National Academy of Sciences*, **106**(45), 19011–19016. ISSN 0027-8424, 1091-6490. doi:10.1073/pnas.0905466106.

P J Avery, D A Henderson (1999). "Fitting Markov Chain Models to Discrete State Series." *Applied Statistics*, **48**(1), 53–61.

Pfeuffer M (2017). *ctmcd: Estimating the Parameters of a Continuous-Time Markov Chain from Discrete-Time Data*. R package version 1.1, URL https://CRAN.R-project.org/package=ctmcd.

Prinz JH, Wu H, Sarich M, Keller B, Senne M, Held M, Chodera JD, Schütte C, Noé F (2011). "Markov models of molecular kinetics: Generation and validation." *The Journal of Chemical Physics*, **134**(17), 174105. doi:10.1063/1.3565032.

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rached Z, Alajaji F, Campbell LL (2004). "The Kullback-Leibler Divergence Rate Between Markov Sources." *IEEE Transactions on Information Theory*, **50**(5), 917–921. doi:10.1109/TIT.2004.826687.

Rouwenhorst KG (1995). "Asset pricing implications of equilibrium business cycle models." In TF Cooley (ed.), *Frontiers of Business Cycle Research*, pp. 294–330. Princeton University Press, Princeton, NJ.

Schweitzer PJ (1968). "Perturbation theory and finite Markov chains." *Journal of Applied Probability*, **5**(2), 401–413. doi:10.2307/3212261.

Sison CP, Glaz J (1995). "Simultaneous confidence intervals and sample size determination for multinomial proportions." *Journal of the American Statistical Association*, **90**(429), 366–369.

Skuriat-Olechnowska M (2005). *Statistical inference and hypothesis testing for Markov chains with Interval Censoring*. diploma thesis, Delft University of Technology.

Snell L (1999). "Probability Book: chapter 11." URL https://web.archive.org/web/20150721052835/http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/Chapter11.pdf.

Spedicato GA (2017). "Discrete Time Markov Chains with R." *The R Journal*. URL https://journal.r-project.org/articles/RJ-2017-036/index.html.

Strelioff CC, Crutchfield JP, H"ubler AW (2007). "Inferring Markov Chains: Bayesian Estimation, Model Comparison, Entropy Rate, and Out-of-Class Modeling." *Physical Review E*, **76**(1), 011106. doi:10.1103/PhysRevE.76.011106.

Swope WC, Pitera JW, Suits F (2004). "Describing protein folding kinetics by molecular dynamics simulations, 1: Theory." *The Journal of Physical Chemistry B*, **108**(21), 6571–6581. doi:10.1021/jp037421y.

Tarjan R (1972). "Depth-First Search and Linear Graph Algorithms." *SIAM Journal on Computing*, **1**(2), 146–160. doi:10.1137/0201010. URL https://doi.org/10.1137/0201010.

Tauchen G (1986). "Finite state Markov-chain approximations to univariate and vector autoregressions." *Economics Letters*, **20**(2), 177–181. doi:10.1016/0165-1765(86)90168-0.

Tauchen G, Hussey R (1991). "Quadrature-Based Methods for Obtaining Approximate Solutions to Nonlinear Asset Pricing Models." *Econometrica*, **59**(2), 371–396. doi:10.2307/2938261.

Tran H, Carmichael DG (2012). "Stationarity of the transition probabilities in the Markov chain formulation of owner payments on projects." *ANZIAM Journal*, **53**, C69–C89. doi:10.21914/anziamj.v53i0.5111.

van der Vorst HA (1992). "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems." *SIAM Journal on Scientific and Statistical Computing*, **13**(2), 631–644. doi:10.1137/0913035.

Villacorta PJ (2012). *MultinomialCI: Simultaneous confidence intervals for multinomial proportions according to the method by Sison and Glaz*. R package version 1.0, URL https://CRAN.R-project.org/package=MultinomialCI.

Wittmann A (2007). *CreditMetrics: Functions for Calculating the CreditMetrics Risk Model*. R package version 0.0-2.

Wolfram Research I (2013a). *Mathematica*. Wolfram Research, Inc., ninth edition.

Wolfram Research I (2013b). "Structural Properties of Finite Markov Processes." URL https://www.wolfram.com/mathematica/new-in-9/markov-chains-and-queues/structural-properties-of-finite-markov-processes.html.

Wright S (1931). "Evolution in Mendelian populations." *Genetics*, **16**(2), 97–159. doi:10.1093/genetics/16.2.97.

Affiliation:

Giorgio Alfredo Spedicato
Ph.D C.Stat FCAS, FSA, CSPA Unipol Group
Via Firenze 11 Paderno Dugnano 20037 Italy
E-mail: spedygiorgio@gmail.com
URL: www.statisticaladvisor.com

Tae Seung Kang
Ph.D student, Computer & Information Science & Engineering
University of Florida Gainesville, FL, USA
E-mail: tskang3@gmail.com

Sai Bhargav Yalamanchi
B-Tech student, Electrical Engineering
Indian Institute of Technology, Bombay Mumbai - 400 076, India
E-mail: bhargavcoolboy@gmail.com

Deepak Yadav
B-Tech student, Computer Science and Engineering
Indian Institute of Technology, Varanasi Uttar Pradesh - 221 005, India
E-mail: deepakyadav.iitbhu@gmail.com

Ignacio Cordón
Software Engineer
Madrid (Madrid), Spain
E-mail: nacho.cordon.castillo@gmail.com