Computational Statistics

Lecture 2: Introduction to statistical computing

Raymond Bisdorff

University of Luxembourg

5 décembre 2022

Part I : Introduction to R

Generating simulation data with Python

```
#!/usr/bin/env python3
# Generate simulation data
# by using the csv module
# R. Bisdorff October 2019
########################################
from random import seed, random, gauss, triangular
import csv
seed(1)
sampleSize = 1000
dataFileName = 'testData.csv'
fo = open(dataFileName,'w')
csvwriter = csv.writer(fo, quoting=csv.QUOTE_NONNUMERIC)
# fo.write('"uniform","gaussian","triangular"\n')
csvwriter.writerow(["uniform","gaussian","triangular"])
for i in range(sampleSize):
    csvwriter.writerow([random(), gauss(0.0,1.0),\
                        triangular(0.0,1.0, 0.5)])
    # fo.write('%f,%f,%f\n'\
    #    % (random(), gauss(0.0,1.0),triangular(0.0,1.0, 0.5)))
fo.close()
print('Successfully generated simulation data.\
       See %s file!' % dataFileName)
```

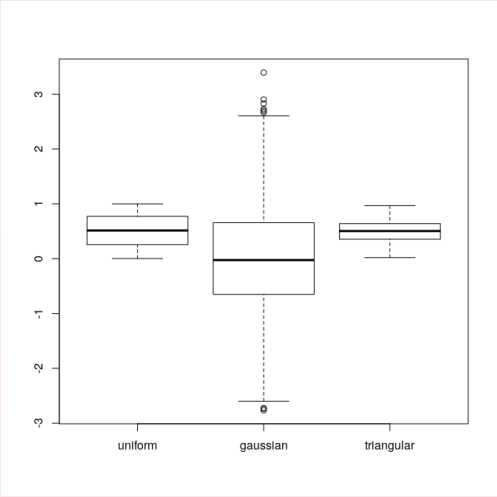

Comma Separated Variables (cvs) data files

Content of file "testData.cvs" previously written :

```
"uniform","gaussian","triangular"
0.411850,0.132471,0.101166
0.595201,-0.608468,0.346641
0.489850,-0.090474,0.431335
0.734018,-1.482315,0.687852
0.026291,2.070926,0.515696
0.036253,0.166679,0.541732
0.972565,-0.858055,0.563673
0.473698,-2.264553,0.560888
0.845316,0.862148,0.576463
0.653825,-1.817857,0.133604
0.929349,-0.949603,0.315628
0.663872,-0.204988,0.064443
...
...
```

Exploring cvs data files with gretl

```
>....$ gretl testData.csv
get_gretl_charset: using UTF-8
parsing testData.csv...
using delimiter ','
   longest line: 33 characters
   first field: 'uniform'
   number of columns = 3
   number of variables: 3
   number of non-blank lines: 1001
scanning for variable names...
   line: uniform,gaussian,triangular
scanning for row labels and data...
treating these as undated data
```



Getting started with R

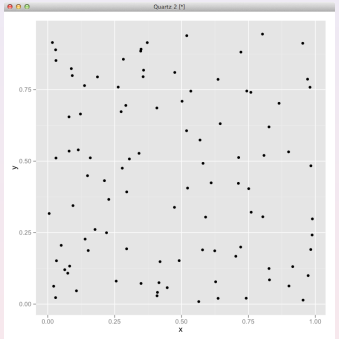
```
.....~$ R
R version 3.0.1 (2013-05-16) -- "Good Sport"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin10.8.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```


Getting started with R

Stopping the R interactive session :

```
> ...
> ...
> q()
Save workspace image? [y/n/c]:
```

Installing and loading libraries for instance :



```
> install.packages("ggplot2")
> library(ggplot2)   # advanced plotting utilities
> help(qplot)        # main plot function
> set.seed(1)        # seeding the random generator
> x = runif(100)     # 100 uniform random numbers
> y = runif(100)     # 100 uniform random numbers
> qplot(x,y)         # see picture above
                     # compare with plot(x,y)
```


R objects : vectors

The vector class :

```
> x = c(2,6,-4,9,18)
> y = c(3,4)
> z = c(x,y)  # c() concatenator for vectors
> z
[1]  2  6 -4  9 18  3  4
>
```

Slicing vectors :

```
> a = z[1]
> b = z[2:4]
> d = z[c(1,3,5,7)]
> d
[1]  2 -4 18  4
```

Vectorwise computing :

```
> 2*z # elementwise multiplication
> z%%2 # elementwise mudulo 2
[1] 0 0 0 0
> length(z)
> sum(z)
[38]
```

R objects : matrices

The matrix class :

```
> vec = 1:20
> n = 4
> p = 5
> x = matrix(vec,nrow=n,ncol=p)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> x = matrix(vec,nrow=n,byrow=T)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
```


R objects : lists

The list class (named variables like a dictionary in Python) :

```
> sample = list(values=runif(20),author="RB")
> sample$author
[1] "RB"
> sample$values
 [1] 0.963579189 0.480940525 0.346466211 0.415405289 0.233362204 0.606959191
 [7] 0.588871247 0.173322686 0.248154716 0.457274632 0.427662024 0.585808923
[13] 0.116761743 0.413169441 0.009225232 0.405227793 0.350169643 0.143638819
[19] 0.666597490 0.798963208
> summary(sample$values)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
0.009225 0.244500 0.414300 0.421600 0.586600 0.963600
```


R objects : data.frames

The data.frame class (list of matrix type) :

```
> v1 = sample(1:5,5,rep=T)
> v2 = sample(LETTERS,5,rep=T)
> v3 = rnorm(5)
> x = data.frame(v1,v2,v3)
> x
  v1 v2         v3
1  2  E -1.6925376
2  1  M  0.3488328
3  2  D  1.5406901
4  5  I  0.8202620
5  5  D -1.4280548
> x$v1
[1] 2 1 2 5 5
> x$v2
[1] E M D I D
Levels: D E I M
> x$v3
[1] -1.6925376  0.3488328  1.5406901  0.8202620 -1.4280548
```


R : read csv data files into data.frames

```
> x = read.csv("testData.csv")
> x[1:5,]
    uniform  gaussian triangular
1  0.250740 -0.859572   0.396068
2  0.384398 -1.193267   0.404702
3  0.818547 -0.749454   0.436957
4  0.063539  0.194047   0.528703
5  0.337983 -0.494701   0.587533

> summary(x)
    uniform           gaussian          triangular
 Min.   :0.001463   Min.   :-2.76517   Min.   :0.02055
 1st Qu.:0.255122   1st Qu.:-0.65045   1st Qu.:0.35510
 Median :0.513780   Median :-0.02534   Median :0.50388
 Mean   :0.507929   Mean   :-0.01684   Mean   :0.49497
 3rd Qu.:0.771545   3rd Qu.: 0.65354   3rd Qu.:0.63928
 Max.   :0.998774   Max.   : 3.39440   Max.   :0.96674
 Max.   :0.99992    Max.   : 3.23964   Max.   :0.98879
>
```

Part II : Doing linear algebra in R

Constructing matrix objects

In R, matrices can be constructed using functions matrix(data,nrow,ncol), cbind(cv1,cv2,...) or rbind(rv1, rv2, ...). Here an example of a Hilbert matrix where entry (i, j) equals $1/(i + j - 1)$.

```
> H3 = matrix(c(1   ,1/2,1/3,
+               1/2,1/3,1/4,
+               1/3,1/4,1/5),
+             nrow=3)
> H3
          [,1]      [,2]      [,3]
[1,] 1.0000000 0.5000000 0.3333333
[2,] 0.5000000 0.3333333 0.2500000
[3,] 0.3333333 0.2500000 0.2000000
```

Similarly :

```
> H3 = 1/cbind(seq(1,3), seq(2,4),
+              seq(3,5))
> H3
          [,1]      [,2]      [,3]
[1,] 1.0000000 0.5000000 0.3333333
[2,] 0.5000000 0.3333333 0.2500000
[3,] 0.3333333 0.2500000 0.2000000
```

Here rbind() would give the same result due to the symmetry of H3.

Accessing matrix elements

Indexing of matrix elements is the same as for R data frames : the (i, j) element of a matrix object M is accessed with $M[i, j]$.
The ith row is accessed with $M[i,]$ and the jth column with $M[, j]$.
The optional argument `drop=False` is used to keep the original row or column orientation of the resulting vector. It is possible to add row and/or column names with the commands `colnames(M) = c("x1", ...)` or `rownames(M) = c("a",...)`.
Internally R stores a matrix object differently than the corresponding data frame object. The latter are stored as lists of columns, whereas the matrix object is stored as a linear vector of values with dimension attribute. Therefore no M$x1 access is for instance given for accessing the column "$x1$". But row and column names may be used as index values like $M[,"x1"]$ for instance.

Matrix properties

1. The dimension of a matrix : `dim(M)`,
2. The transpose of a matrix : `t(M)`,
3. The determinant of a matrix : `det(M)`,
4. The diagonal elements of a matrix : `diag(M)` ; `diag(diag(M))` renders a corresponding diagonal matrix,
5. The trace of a matrix my be computed as `sum(diag(M))`,
6. The lower or upper triangular parts of a matrix : `M[lower.tri(M)]` or `M[upper.tri(M,diag=T)]` where the optional argument `diag=T` includes the diagonal elements.

Elementwise operations

1. `2 * M` multiplies each element of M by two,
2. If M and N are two matrices of same dimensions, `M + N` renders an elementwise addition of M and N,
3. If M and N are of different dimensions, an error "`non-conformable arrays`" occurs.
4. Notice that `M * N` elementwise multiplies each element of M by the corresponding element of N.

Matrix multiplication

Usual matrix multiplication is operated via the `%*%` and requires conformable matrix operands. To multiply a matrix X with a transposed matrix $t(X)$, the special `crossprod()` function is more efficient. It avoids the creation of a transpose.

```
> X = cbind(runif(3),runif(3),runif(3))
> t(X) %*% X
          [,1]      [,2]      [,3]
[1,] 1.0112340 0.8828675 0.8156668
[2,] 0.8828675 1.0706355 0.7575319
[3,] 0.8156668 0.7575319 1.0875304
```

Similarly :

```
> crossprod(X,X)
          [,1]      [,2]      [,3]
[1,] 1.0112340 0.8828675 0.8156668
[2,] 0.8828675 1.0706355 0.7575319
[3,] 0.8156668 0.7575319 1.0875304
```


Matrix inversion

The inverse of a square $n \times n$ matrix A, denoted A^{-1}, is the solution of the matrix equation : $AA^{-1} = I$, where I is the $n \times n$ diagonal identity matrix.
In R, this equation is solved using a lower and upper triangular LU decomposition or an orthogonal and upper triangular QR decomposition that minimizes the rounding error occurrances. The R commands are `solve()` respectively `qr.solve()` :

```
> H3
          [,1]      [,2]      [,3]
[1,] 1.0000000 0.5000000 0.3333333
[2,] 0.5000000 0.3333333 0.2500000
[3,] 0.3333333 0.2500000 0.2000000
> H3inv = solve(H3)
> H3inv
     [,1] [,2] [,3]
[1,]    9  -36   30
[2,]  -36  192 -180
[3,]   30 -180  180
> H3 %*% H3inv
              [,1]         [,2]          [,3]
[1,]  1.000000e+00 3.330669e-15 -9.610368e-16
[2,]  0.000000e+00 1.000000e+00  0.000000e+00
[3,] -3.773024e-16 3.684553e-15  1.000000e+00
```

Solving linear systems

The function `solve(A,b)` gives the solution to the linear system of equations of the $Ax = b$. For instance, let us find a vector x such that $H_3 x = b$, where H_3 is the Hilbert matrix of dimension 3×3 and $b = [1, 2, 3]^T$.

```
> b = matrix(c(1,2,3),ncol=1)
> x = solve(H3,b)
> t(x)
     [,1] [,2] [,3]
[1,]   27 -192  210
```

In other words, the solution vector is $x = [27, -192, 210]^T$, as we may easily verify :

```
> t(x) %*% H3
     [,1] [,2] [,3]
[1,]    1    2    3
> t(b)
     [,1] [,2] [,3]
[1,]    1    2    3
```

Eigenvalues and -vectors

The eigenvectors of a square matrix are the non-zero vectors that, after being multiplied by the matrix, remain parallel to the original vector. If A is a square matrix, a non-zero vetor x is an eigenvector of A if there is a scalar λ such that $Ax = \lambda x$. The R function `eigen(A)` gives the eigenvalues `$values` and the eigenvectors `$vectors` of a matrix A. Let x_1 denote the first column of `$vectors` output. This is the first eigenvector of H_3 and it corresponds to the first eigenvalue λ_1. Hence, $H_3 x_1 = \lambda_1 x_1$. For example :

```
> eigH3 = eigen(H3)
> eigH3
$values
[1] 1.408318927 0.122327066 0.002687340
$vectors
           [,1]       [,2]       [,3]
[1,] -0.8270449  0.5474484  0.1276593
[2,] -0.4598639 -0.5282902 -0.7137469
[3,] -0.3232984 -0.6490067  0.6886715
> x1 = eigH3$vectors[,1,drop=F]
> t(H3 %*% x1)
          [,1]      [,2]       [,3]
[1,] 1.164743 0.647635 0.4553073
> lambda1 = eigH3$values[1]
> t(lambda1 * x1)
[1] 1.1647430 0.6476350 0.4553073
```

The singular value decomposition

The singular value decomposition of a matrix A consists of three square matrices : U, D and V such that $A = UDV^T$, where D is a diagonal matrix and U and V are orthogonal, that is their inverses correspond to their transposes. The diagonal elements of D are called the singular values of A.

```
> H3svd = svd(H3)
> H3svd
$d
[1] 1.408318927 0.122327066 0.002687340
$u
           [,1]       [,2]       [,3]
[1,] -0.8270449  0.5474484  0.1276593
[2,] -0.4598639 -0.5282902 -0.7137469
[3,] -0.3232984 -0.6490067  0.6886715
$v
           [,1]       [,2]       [,3]
[1,] -0.8270449  0.5474484  0.1276593
[2,] -0.4598639 -0.5282902 -0.7137469
[3,] -0.3232984 -0.6490067  0.6886715
> H3svd$u %*% diag(H3svd$d) %*% t(H3svd$v)
          [,1]      [,2]      [,3]
[1,] 1.0000000 0.5000000 0.3333333
[2,] 0.5000000 0.3333333 0.2500000
[3,] 0.3333333 0.2500000 0.2000000
```


The singular value decomposition – continue

Note that $A^T A = V^{-1} D^2 V$. This is a "similarity transformation" which tells us that the squares of the singular values of A are the eigenvalues of $A^T A$.

```
> H3svd$d * H3svd$d
[1] 1.983362e+00 1.496391e-02 7.221798e-06
> eigen(t(H3) %*% H3 )$values
[1] 1.983362e+00 1.496391e-02 7.221798e-06
```

Because of the properties of the U, D and V matrices, the singular value decomposition provides as well a a simple way to compute the inverse of a square matrix : $A^{-1} = VD^{-1}U^T$.

```
> H3svd$v %*% diag(1/H3svd$d) %*% t(H3svd$u)
     [,1] [,2] [,3]
[1,]    9  -36   30
[2,]  -36  192 -180
[3,]   30 -180  180
```

The Choleski decomposition of positive definite matrices

An $n \times n$ real matrix M is positive definite if $z^T Mz > 0$ for all non-zero vectors z with real entries. All covariance matrices in statistics are positive definite. Now, if a matrix A is positive definite, it possesses a square root and all eigenvalues are positive. In fact there are generally several matrices B such that $A = B^2$. The Choleski decomposition computes in fact a special square root U of A which is an upper triangular matrix such that $U^T U = A$. Note that $A^{-1} = U^{-1}(U^{-1})^T$ gives the inverse of A via the inverse of U. The R functions `chol()` and `chol2inv()` do that :

```
> H3chol = chol(H3)
> H3chol
     [,1]      [,2]      [,3]
[1,]    1 0.5000000 0.3333333
[2,]    0 0.2886751 0.2886751
[3,]    0 0.0000000 0.0745356

> crossprod(H3chol,H3chol)
          [,1]      [,2]      [,3]
[1,] 1.0000000 0.5000000 0.3333333
[2,] 0.5000000 0.3333333 0.2500000
[3,] 0.3333333 0.2500000 0.2000000

> chol2inv(H3chol)
     [,1] [,2] [,3]
[1,]    9  -36   30
[2,]  -36  192 -180
[3,]   30 -180  180
```

The Choleski decomposition for solving linear systems

The Choleski decomposition may be used for solving the linear system $Ax = b$. If $A = U^T U$, then we see that :

$$U^T Ux = b \quad \equiv \quad \begin{cases} U^T y = b & (1) \\ Ux = y & (2) \end{cases}$$

System (1) is lower triangular and a forward elimination and the R method `forwardsolve()` can be used to solve it. Whereas System (2) is upper triangular and a back substitution with R function `backsolve()` can be used.

```
> H3
          [,1]      [,2]      [,3]
[1,] 1.0000000 0.5000000 0.3333333
[2,] 0.5000000 0.3333333 0.2500000
[3,] 0.3333333 0.2500000 0.2000000
> b
[1] 1 2 3
> solve(H3,b)
[1]   27 -192  210
> U = chol(H3)
> y = forwardsolve( t(U), b )
> backsolve( U, y )
[1]   27 -192  210
```


The QR decomposition

Another way of decomposing a matrix A is via the $A = QR$ decomposition, where Q is an orthogonal matrix, and R is an upper triangular matrix. The corresponding R function is `qr()`.

```
> H3qr = qr(H3)
> H3qr
$qr
          [,1]       [,2]         [,3]
[1,] -1.1666667 -0.6428571 -0.450000000
[2,]  0.4285714 -0.1017143 -0.105337032
[3,]  0.2857143  0.7292564  0.003901372

$rank
[1] 3

$qraux
[1] 1.857142857 1.684240553 0.003901372

$pivot
[1] 1 2 3

attr(,"class")
[1] "qr"
```

```
> Q = qr.Q(H3qr)
> Q
          [,1]       [,2]       [,3]
[1,] -0.8571429  0.5016049  0.1170411
[2,] -0.4285714 -0.5684856 -0.7022469
[3,] -0.2857143 -0.6520864  0.7022469

> R = qr.R(H3qr)
> R
          [,1]       [,2]         [,3]
[1,] -1.166667 -0.6428571 -0.450000000
[2,]  0.000000 -0.1017143 -0.105337032
[3,]  0.000000  0.0000000  0.003901372

> Q %*% R
          [,1]      [,2]      [,3]
[1,] 1.0000000 0.5000000 0.3333333
[2,] 0.5000000 0.3333333 0.2500000
[3,] 0.3333333 0.2500000 0.2000000
```


Part III : Applications of linear algebra

Principal Component Analysis - PCA

The singular value decomposition of a square symetric matrix A of dimension $n \times n$, delivers also an orthogonal linear transformation ordered by decreasing variance. Indeed, if $A = UDV^T$, U gives the eigenvectors of AA^T and V gives the eigenvectors of $A^T A$. Hence, $B = U^T A = U^T UDV^T = DV^T$, and each column of B renders a rotation of the corresponding column in A that captures a maximal part of the remaining variance of matrix A.

Example R session :

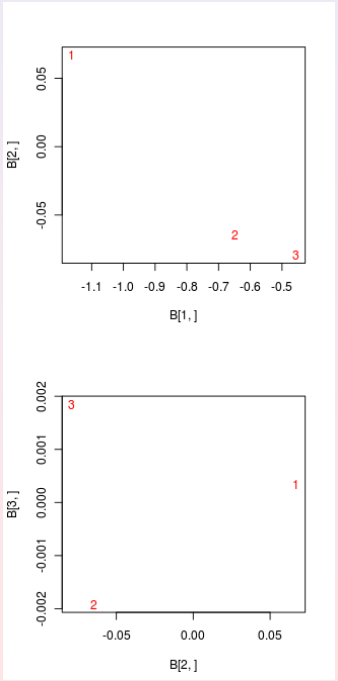
```
> H3svd = svd(H3)
> H3svd
$d
[1] 1.40831 0.12232 0.00268
$u
          [,1]      [,2]      [,3]
[1,] -0.82704  0.54744  0.12765
[2,] -0.45986 -0.52829 -0.71374
[3,] -0.32329 -0.64900  0.68867
$v
          [,1]      [,2]      [,3]
[1,] -0.82704  0.54744  0.12765
[2,] -0.45986 -0.52829 -0.71374
[3,] -0.32329 -0.64900  0.68867
```

```
> B = t(H3svd$u) %*% H3
           [,1]      [,2]      [,3]
[1,] -1.1647430 -0.647635 -0.455307
[2,]  0.0669677 -0.064624 -0.079391
[3,]  0.0003430 -0.001918  0.001850
> D = diag(H3svd$d)
          [,1]      [,2]         [,3]
[1,] 1.408319 0.0000000 0.000000000
[2,] 0.000000 0.1223271 0.000000000
[3,] 0.000000 0.0000000 0.002687340
> D %*% t(H3svd$v)
           [,1]      [,2]      [,3]
[1,] -1.1647430 -0.647635 -0.455307
[2,]  0.0669677 -0.064624 -0.079391
[3,]  0.0003430 -0.001918  0.001850
```

PCA with the eigen decomposition

The same orthogonal linear rotation $B = AX = \lambda \cdot X$ of a square symmetric matrix A may be achieved by directly computing its eigen values and vectors. Using the eigen function in R may illustrate this :

```
> eigH3 = eigen(H3)
> eigH3
$values
[1] 1.408318 0.122327 0.002687
$vectors
          [,1]     [,2]     [,3]
[1,] -0.82704  0.54744  0.12765
[2,] -0.45986 -0.52829 -0.71374
[3,] -0.32329 -0.64900  0.68867

> B = eigH3$values * t(eigH3$vectors)
> #   B = t(eigH3$vectors) %*% H3
           [,1]     [,2]     [,3]
[1,] -1.164743 -0.64763 -0.45530
[2,]  0.066967 -0.06462 -0.07939
[3,]  0.000343 -0.00191  0.00185

> plot(B[1,],B[2,],"n")
> text(B[1,],B[2,],c("1","2","3"))

> plot(B[2,],B[3,],"n")
> text(B[2,],B[3,],c("1","2","3"))
```



Discrete Markov chains

A discrete Markov chain describes probabilistic movements between a finite number of states. Suppose the given potential states are numbered s_1 to s_6. If a person is located in state s_t at time t, the probability that she moves to another state s_{t+1} in time $t+1$ is given by a probability transition matrix T shown in the R session (next column). If we start our random walk in state s_3, we can simulate 10000 moves using a simple Gibbs sampler. The relative frequency of visiting each individual state may be estimated with the table function.

Example R session :

```
> T = read.csv('chain.csv')
> rownames(T) = colnames(T)
   s1   s2   s3   s4   s5   s6
s1 0.50 0.50 0.00 0.00 0.00 0.00
s2 0.25 0.50 0.25 0.00 0.00 0.00
s3 0.00 0.25 0.50 0.25 0.00 0.00
s4 0.00 0.00 0.25 0.50 0.25 0.00
s5 0.00 0.00 0.00 0.25 0.50 0.25
s6 0.00 0.00 0.00 0.00 0.50 0.50
> sampleSize = 10000
> mc = rep(0,sampleSize)
> mc["s1"] = 3   # initial state
> for (j in 2:sampleSize){
+     mc[j] = sample(1:6,size=1,\
                    prob=T[mc[j-1],])}
> table(mc)/sampleSize
  s1     s2     s3     s4     s5     s6
0.0931 0.1876 0.2027 0.2013 0.2099 0.1054
```

Stationary probability distribution

The Markov chain illustrated before has two important properties : (1) it is irreducible, i.e. every state is reachable from every state ; (2) it is aperiodic, i.e. that states are not revisited within constant time periods. Now, the relative frequency of visiting a state, i.e. the stationary probability distribution p (column vector) of an irreducible and aperiodic Markov chain, with transition matrix T, is unique and may be computed by solving the eigen system :

$$\boxed{T^t \cdot p = \lambda p}$$

```
> ex = eigen(t(T))
$values
[1]  1.000000e+00  9.045085e-01  ...
$vectors
          [,1]       [,2]       [,3]  ...
[1,] 0.2357023 -0.3535534  0.3535534  ...
[2,] 0.4714045 -0.5720614  0.2185080  ...
[3,] 0.4714045 -0.2185080 -0.5720614  ...
[4,] 0.4714045  0.2185080 -0.5720614  ...
[5,] 0.4714045  0.5720614  0.2185080  ...
[6,] 0.2357023  0.3535534  0.3535534  ...
> p = ex$vectors[,1]/sum(ex$vectors[,1])
> p = as.matrix(p)
     [,1]
[1,]  0.1
[2,]  0.2
[3,]  0.2
[4,]  0.2
[5,]  0.2
[6,]  0.1
> X = as.matrix(T)
> t(X) %*% p
     s1  s2  s3  s4  s5  s6
[1,] 0.1 0.2 0.2 0.2 0.2 0.1
```