Getting started

Installation of the main packages

In [1]:
```
library(igraph)
library(ggplot2)
library("igraphdata")
data("karate")
data(UKfaculty)
data(USairports)
data(enron)
data(foodwebs)
data(immuno)
data(kite)
data(macaque)
data(rfid)
data(yeast)
```

```
Attaching package: ‘igraph’


The following objects are masked from ‘package:stats’:

    decompose, spectrum


The following object is masked from ‘package:base’:

    union
```

Spectral analysis of a graph with full observation

1/ Create the function *Spectrum.Laplacian(graph)* which takes as an input a graph object and will return the eigenvalues and the associated eigenvectors of the unnormalized graph Laplacian.

Reminder: the unnormalized graph Laplacian is equal to $L = D - A$, where D is the degree matrix and A is the adjacency matrix.

In [2]:
```
Spectrum.Laplacian<-function(graph){
    graph.1<-as.undirected(graph)
    A<-  as_adjacency_matrix(graph.1) #Adjacency matrix
    D <-  diag(degree(graph.1)) #Degree matrix
    L <- D - A #Laplacian matrix
    return(eigen(L, symmetric=TRUE))
}
```

2/ Create the function *Spectrum.Laplacian.Random.attack(graph,number.of.edges.attacked)* which:

- removes *number.of.edges.attacked* number of edges in the graph, where the edges are selected randomly (Hint: use the functions *delete_edges()* and *sample()*)
- computes and returns the eigenvalues and the associated eigenvectors of the unnormalized Laplacian associated with the graph after the attack.

In [3]:

```
Spectrum.Laplacian.Random.attack <- function(graph, number.of.edges.attacked
    graph.1 <- as.undirected(graph)

    # Remove random edges
    edges_to_attack <- sample(E(graph.1), number.of.edges.attacked, replace
    graph.1 <- delete_edges(graph.1, edges_to_attack)

    # Obtain new laplacian after the attack
    A <- as_adjacency_matrix(graph.1)
    D <- diag(degree(graph.1))
    L <- D - A

    # Return eigenvalues and eigenvectors
    return(eigen(L, symmetric = TRUE))
}
```

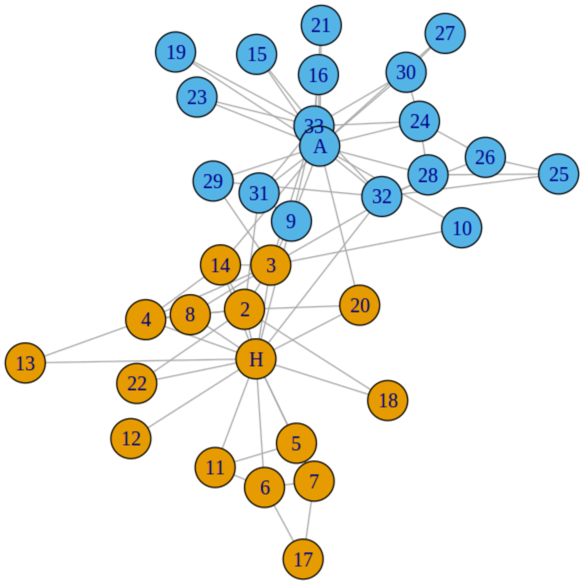
3/ Compare the evolution of the spectrum of the Laplacian before and after an attack (You can try with a removal of 4, 10 and 15 edges). How evolve the smallest and the second smallest eigenvalues of the unnormalized graph Laplacian matrix? Do the experiments on the *karate* network.

In [4]:

```
plot(karate)

# Number of edges to be attacked
edges_to_attack <- c(4, 10, 15)

# Compare Laplacian spectrum before and after attack
for (edges_attacked in edges_to_attack) {
    cat("\nNumber of edges attacked:", edges_attacked)

    # Laplacian spectrum before attack
    spectrum_before <- Spectrum.Laplacian(karate)

    # Laplacian spectrum after attack
    spectrum_after <- Spectrum.Laplacian.Random.attack(karate, edges_attacke

    cat("\nBefore Attack - Smallest eigenvalue:",sort(unlist(spectrum_before
    cat("\nBefore Attack - Second smallest eigenvalue:", sort(unlist(spectru

    cat("\nAfter Attack - Smallest eigenvalue:", sort(unlist(spectrum_after[
    cat("\nAfter Attack - Second smallest eigenvalue:", sort(unlist(spectrum
}
```

```
This graph was created by an old(er) igraph version.
  Call upgrade_graph() on it to use with the current igraph version
  For now we convert it on the fly...
```

```
Number of edges attacked: 4
Before Attack - Smallest eigenvalue: 4.724264e-16
Before Attack - Second smallest eigenvalue: 0.4685252
After Attack - Smallest eigenvalue: -6.853032e-16
After Attack - Second smallest eigenvalue: 0.460053
Number of edges attacked: 10
Before Attack - Smallest eigenvalue: 4.724264e-16
Before Attack - Second smallest eigenvalue: 0.4685252
After Attack - Smallest eigenvalue: 7.316647e-16
After Attack - Second smallest eigenvalue: 0.3050862
Number of edges attacked: 15
Before Attack - Smallest eigenvalue: 4.724264e-16
Before Attack - Second smallest eigenvalue: 0.4685252
After Attack - Smallest eigenvalue: -4.429652e-16
After Attack - Second smallest eigenvalue: -4.037613e-16
```



The smallest eigenvalue is always equal to 0. This happens because the laplacian is calculated as

$$L = D - A$$

so that the sum of every row is equal to 0. This implies that an all 1 vector v_1 satisfies

$$L\ v_1 = 0$$

so that $\lambda_0 = 0$ is always an eigenvalue with v_1 its associated eigenvector.

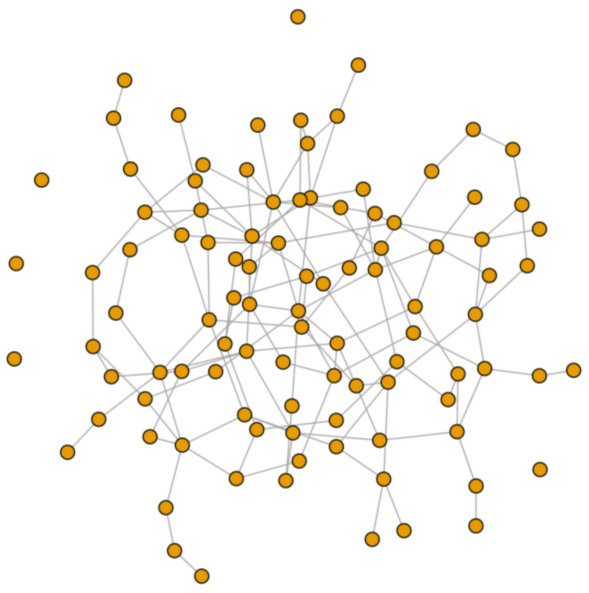
On the other hand, the second smallest eigenvalue does indeed change as edges are attacked. In fact, this number, also called algebraic connectivity, is a measure of the robustness of the network. It can be proven that is value is equal to:

$$\min_{\substack{x \\ \sum x_i = 0}} \sum_{(i,j)\in E} (x_i - x_j)^2$$

so that it measures the connectivity between two classes in a graph.

For higher values of λ_2, the more connected the two subgraphs will be. Logically, as more and more edges are attacked, the value of λ_2 decreases.

In [5]:
```
graph <- erdos.renyi.game(100, 0.028, type = "gnp")
plot(graph, vertex.label= NA, edge.arrow.size=0.02,vertex.size = 5, xlab = "
```



Random Network: G(N,p) model

4/ What is the multiplicity of the smallest eigenvalue of the unnormalized Laplacian associated with the graph defined above? Is it related to a specific quantity in the graph (Look at the graph above)? Warning: you have to round all the eigenvalues with the function *round(,3)*.

In [6]:
```
spectrum <-Spectrum.Laplacian(graph)
rounded_spectrum <- unname(sapply(sort(unlist(spectrum[1])), round, digits =
cat("\n Rounded smallest eigenvalue:", rounded_spectrum[1])
cat("\n Multiplicity of smallest eigenvalue:", sum(rounded_spectrum == rounc
Rounded smallest eigenvalue: 0
Multiplicity of smallest eigenvalue: 6
```

The multiplicity of the smallest eigenvalue is equal to 1 plus to the number of isolated nodes in the graph. This can easily be understood through the laplacian L. Indeed, let's suppose that there is only one isolated node. Then,

$$L = D - A = \begin{bmatrix} d_1 & -a_{12} & \cdots & \cdots & 0 \\ -a_{21} & d_2 & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -a_{n1} & -a_{n2} & \cdots & d_{n-1} & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}$$

so that both $v_1 = (1, 1, \ldots, 1, 1)$ and $v_1 = (1, 1, \ldots, 1, \alpha)$ satisfy

$$Lv_1 = Lv_2 = 0 \Rightarrow \lambda_1 = \lambda_2 = 0$$

proving that in that case the multiplicity of the 0 eigenvalue is 2. This result easily generalizes for more isolated nodes.

Spectral analysis with partially observed network

In this section we will implement the algorithm seen during the class to estimate, in an online manner the largest eigenvalue of a symmetric matrix.

5/ Complete the function *Numerical.integrator(A,x.initial,nb.iterations)* which implements the following recursive algorithm:

$$y(k+1) = \frac{y(k) + 0.1(A - y^T(k)Ay(k) * I)y(k)}{\|y(k) + 0.1(A - y^T(k)Ay(k) * I)y(k)\|_2}$$

where:

- A: matrix of size $n \times n$.
- $y(k)$: vector of size n. $y(k)$ can be seen as an estimator of the eigenvector associated with the largest eigenvalue. $y^T(k)$ is the transpose of vector $y(k)$.
- I: the identity matrix of size $n \times n$.
- $\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$.

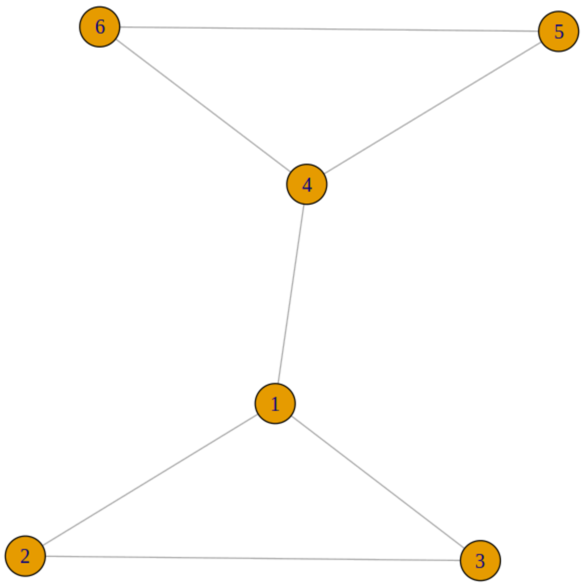
In [7]:
```
Numerical.integrator<-function(A,x.initial,nb.iterations){
    I<-dim(A)[1]
    x<-array(0,c(I,nb.iterations))
    x.cum<-array(0,c(I,nb.iterations))
    y1<-array(0,c(I,nb.iterations))
    Lambda<-array(0,c(nb.iterations))
    Lambda.1<-array(0,c(nb.iterations))
    Lambda[1]<-0
    x[,1]<-x.initial
    for(n in 1:(nb.iterations-1)){
        z2 <- t(x[, n]) %*% A %*% x[, n]  # x^TAx
        y1[,n+1]<- x[, n] + 0.1 * (A - z2[1] * diag(I)) %*% x[, n] # numerat
        x[,n+1]<- y1[, n + 1] / sqrt(sum(y1[, n + 1]^2)) # normalization
        Lambda.1[n+1]<- t(x[, n + 1]) %*% A %*% x[, n + 1] # Estimate of the
    }
    return(list(x,Lambda.1))
}
```

5/ Look at the function *Stochastic.approximation.scheme(A,x.initial,nb.iterations,nb.,nb.iterations.1)* which implements the stochastic approximation algorithm seen during the class. Please explain each line of the code and its purpose.

In [8]:
```
Stochastic.approximation.scheme<-function(A,x.initial,nb.iterations,nb.,nb.i
    I<-dim(A)[1]
    x<-array(0,c(I,nb.iterations)) #Eigenvector estimations
    x.cum<-array(0,c(I,nb.iterations))
    y1<-array(0,c(I,nb.iterations))
    Lambda<-array(0,c(nb.iterations))
    Lambda.1<-array(0,c(nb.iterations)) #List of estimated highest eigenvalu
    Lambda[1]<-0
    x[,1]<-x.initial #Initialize x
    #Recurrency relationship iterations
    for(n in 1:(nb.iterations-1)){
        Lambda<-c()
        Block.matrix<-array(0,c(I,nb.iterations.1))
        #nb.iterations.1 nodes are sampled in each iteration
        for(m in 1:nb.iterations.1){
            z<-sample(I)[1] #Sample a node
            Lambda<-c(Lambda,I*x[z,n]*(A[z,]%*%x[,n])) #Add new node informa
            Block.matrix[z,m]<-I*A[z,]%*%x[,n] #Store results
            Block.matrix[-z,m]<-0
        }
        z2<-mean(Lambda) #Unbiased estimator of r_A(x)
        z3<-A%*%x[,n] #Compute z3 term for the recursive algorithm
        y1[,n+1]<-y1[,n]+(0.1)*(z3-(z2*diag(I))%*%x[,n]) #Update new eigenve
        x[,n+1]<-y1[,n+1]/norm(y1[,n+1],"2") #Normalization
        Lambda.1[n+1]<-t(x[,n+1])%*%A%*%x[,n+1] #Estimate highest eigenvalue
        }
    return(list(x,Lambda.1))}
```

In [9]:
```
I<-6
A<-matrix(c(0,1,1,1,0,0,1,0,1,0,0,0,1,1,0,0,0,0,1,0,0,0,1,1,0,0,0,1,0,1,0,0,
g<-graph_from_adjacency_matrix(A,mode="undirected")
plot(g)
L<-diag(colSums(A))-A
epsilon<-0.001
W<-diag(array(1,c(I)))-epsilon*L
```

```
eig.W<-eigen(W)
B<-W-(1/I)*array(1,c(I))%*%t(array(1,c(I)))
```



6/ Run the cell above. Test the *Stochastic.approximation.scheme()* function and the *Numerical.integrator()* function with the matrix B defined in the cell above and with *nb.iterations=5000*. What do you observe? What is happening when you change *nb.iterations.1*? To compare the two algorithms, you can use the *matplot* function or use the library *ggplot2*.

```
In [10]: x.initial<-runif(I,0,1)
         x.initial<-x.initial/norm(x.initial,"2")
         nb.iterations<-5000
         nb.iterations.1<-3
         test.sa<-Stochastic.approximation.scheme(B,x.initial,nb.iterations,nb.,nb.it
         test.ni<-Numerical.integrator(B,x.initial,nb.iterations)
```

By considering that

$$W = I - \epsilon L - \frac{1}{N} 11^T$$

Then, if x is an eigenvector of W,

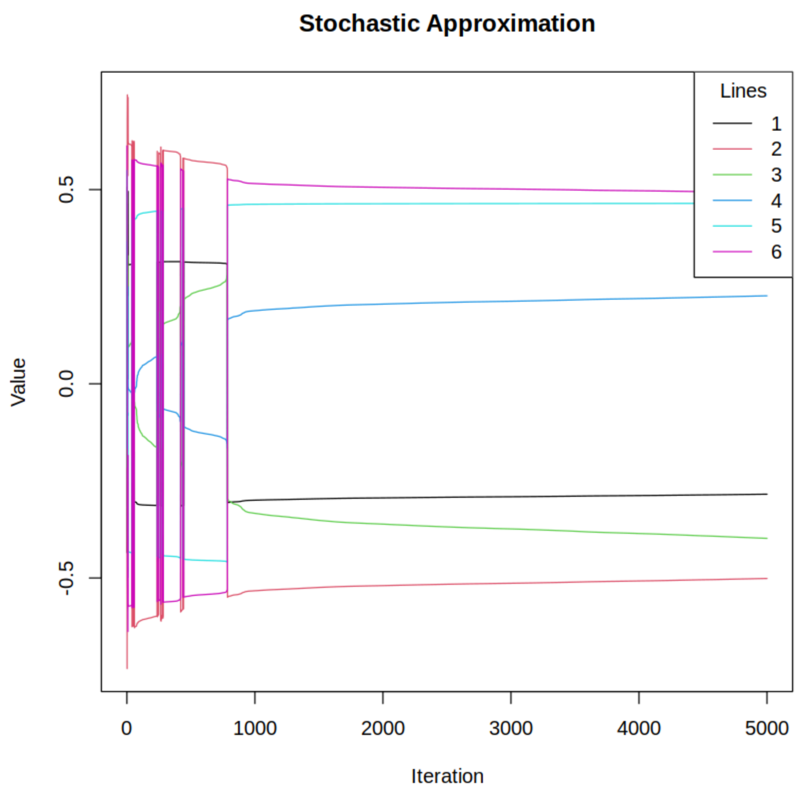
$$Wx = x - \epsilon Lx - \frac{1}{N} 11^T x = \lambda x$$

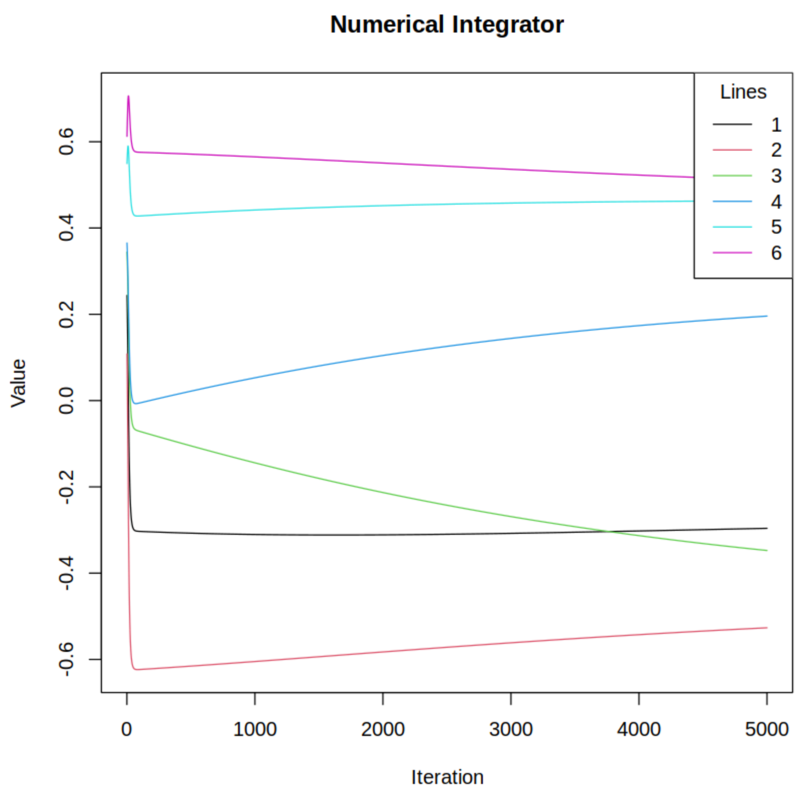
Where the last term is 0 because $\sum x_i = 0$. Then

$$Wx = Lx = (\frac{1}{\epsilon} - \lambda)x$$

Meaning that x is the corresponding eigenvector of x

In [11]:
```
matplot(t(test.sa[[1]]), type = "l", col = 1:6, lty = 1, xlab = "Iteration",
legend("topright", legend = c("1", "2", "3", "4", "5", "6"), col = 1:6, lty
matplot(t(test.ni[[1]]), type = "l", col = 1:6, lty = 1, xlab = "Iteration",
legend("topright", legend = c("1", "2", "3", "4", "5", "6"), col = 1:6, lty
```



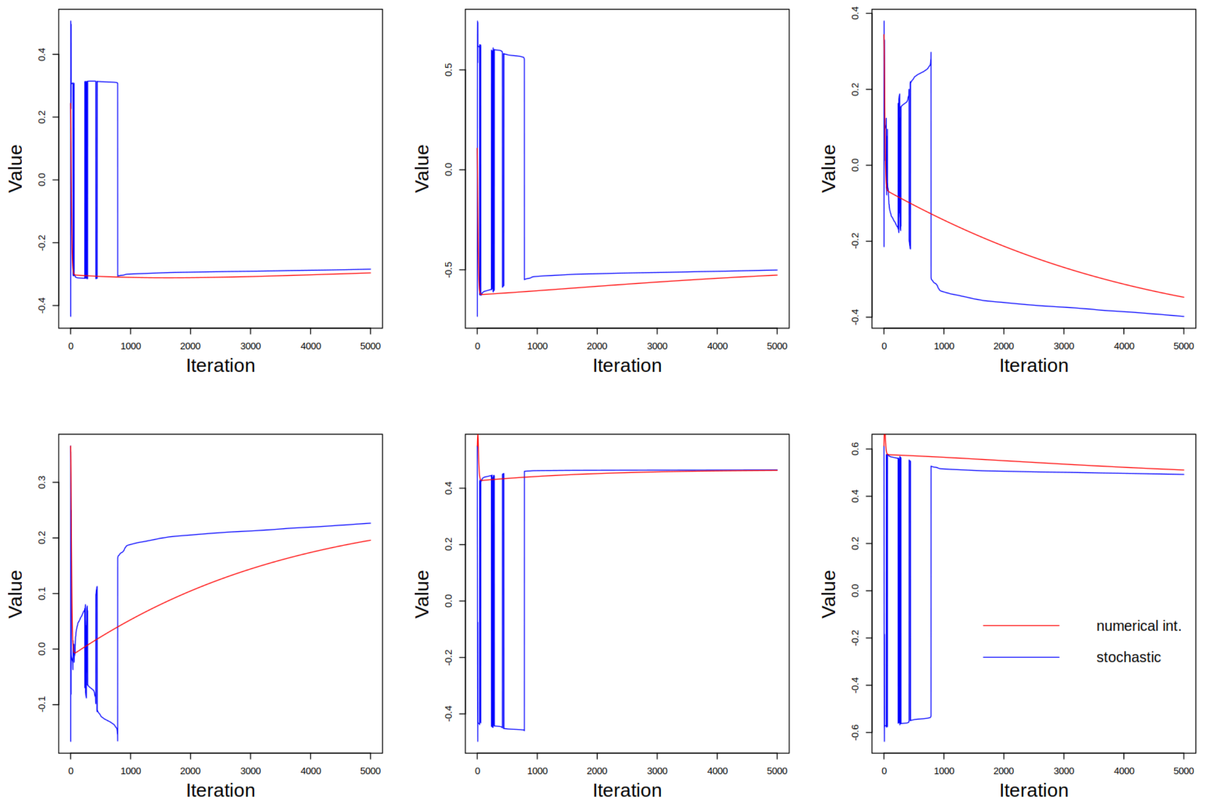


It is observed that both algorithms converge to the same eigenvector. As can be seen in the legend, classification based over positive and negative values implies that two classes are defined: $\{1,2,3\}$ and $\{4,5,6\}$. However, the results do not appear to be repeatable across different runs of the algorithms.

Overall, stochastic approximation has a more unstable behavior, mostly during the first iterations, which is logical given that the algorithm is learning node by node while the numerical integrator has already the whole information of the graph.

We now plot each component of the eigenvector in function of iterations.

In [12]:
```
# Install the necessary package if not already installed
if (!requireNamespace("repr", quietly = TRUE)) install.packages("repr")

# Load the repr package
library(repr)

# Set the figure size for Jupyter Notebook
options(repr.plot.width = 14, repr.plot.height = 10)

# Set up larger margins and overall size
par(mfrow = c(2, 3), mar = c(5, 5, 4, 2) + 0.1, oma = c(0, 0, 2, 0), cex.mai

for (i in 1:6) {
  # Plot for Stochastic Approximation
  matplot(test.sa[[1]][i, ], type = "l", col = "blue", lty = 1, xlab = "Iter

  # Plot for Numerical Integrator
  matplot(test.ni[[1]][i, ], type = "l", col = "red", lty = 1, xlab = "Itera


}
# Add legend
legend("topright", inset = c(-0.55,0.5), xpd = TRUE, legend = c("numerical i
# Reset the plotting grid to default
par(mfrow = c(1, 1), mar = c(5, 4, 4, 2) + 0.1, oma = c(0, 0, 0, 0))
```



As it can be seen in the plots, convergence is quite good in both algorithms. As nb.iterations.1 increases, the stochastic approximation algorithm is more precise, because it is somehow overfitting over the nodes, but at the same time it becomes more unstable (meaning that its values can jump almost in a discrete way from one iteration to the next one).

We verify that the sum of the components of the final eigenvector equals 0, as expected.

```
In [13]: sum(test.sa[[1]][,5000])
```

```
0
```