Solutions to Exercises from Chapter 04

Contents

1 Exercises on systems of linear equations **1**
- 1.1 Exercise 01 . . . 1
- 1.2 Exercise 02 . . . 2
- 1.3 Exercise 03 . . . 3
- 1.4 Exercise 04 . . . 5
- 1.5 Exercise 05 . . . 7

2 Exercises on matrix decomposition **9**
- 2.1 Exercise 06 . . . 9
- 2.2 Exercise 07 . . . 10
- 2.3 Exercise 08 . . . 12
- 2.4 Exercise 09 . . . 12
- 2.5 Exercise 10 . . . 14
- 2.6 Exercise 11 . . . 16
- 2.7 Exercise 12 . . . 19
- 2.8 Exercise 13 . . . 21
- 2.9 Exercise 14 . . . 23
- 2.10 Exercise 15 . . . 24
- 2.11 Exercise 16 . . . 27
- 2.12 Exercise 17 . . . 30
- 2.13 Exercise 18 . . . 34
- 2.14 Exercise 19 . . . 36

The `comphy` package is loaded once at the beginning so to make all its functions available to this exercises session.

```
library(comphy)
```

1 Exercises on systems of linear equations

1.1 Exercise 01

Find the solution of the following system of five equations and five unknowns:

$$\left\{\begin{array}{rcr} 2x_1 + x_2 - 3x_3 - 5x_4 + x_5 & = & -4 \\ 7x_1 - x_2 - 5x_5 & = & 3 \\ -6x_1 + 8x_2 - 2x_4 - 4x_5 & = & -2 \\ -3x_1 + 8x_2 + x_3 + 8x_4 + 6x_5 & = & 0 \\ 5x_1 + 2x_2 + 7x_3 + 8x_4 - 2x_5 & = & -2 \end{array}\right.$$

using the function `gauss_elim`. Verify the solution found by substitution in the original system (use R code).

SOLUTION

The system in matrix form,

$$A\mathbf{x} = \mathbf{b},$$

yields,

$$A = \begin{pmatrix} 2 & 1 & -3 & -5 & 1 \\ 7 & -1 & 0 & 0 & -5 \\ -6 & 8 & 0 & -2 & -4 \\ -3 & 8 & 1 & 8 & 6 \\ 5 & 2 & 7 & 8 & -2 \end{pmatrix} \quad , \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \quad , \quad \mathbf{b} = \begin{pmatrix} -4 \\ 3 \\ -2 \\ 0 \\ -2 \end{pmatrix}$$

To use Gaussian elimination we will have to create the augmented matrix,

$$M = (A|\mathbf{b}) = \begin{pmatrix} 2 & 1 & -3 & -5 & 1 & -4 \\ 7 & -1 & 0 & 0 & -5 & 3 \\ -6 & 8 & 0 & -2 & -4 & -2 \\ -3 & 8 & 1 & 8 & 6 & 0 \\ 5 & 2 & 7 & 8 & -2 & -2 \end{pmatrix}$$

The function then returns the solution straight away in terms of components of a vector.

```
# Augmented matrix
raw_data <- c( 2, 1,-3,-5, 1,-4,
               7,-1, 0, 0,-5, 3,
              -6, 8, 0,-2,-4,-2,
              -3, 8, 1, 8, 6, 0,
               5, 2, 7, 8,-2,-2)
M <- matrix(raw_data,ncol=6,byrow=TRUE) # Data above input
                                        # row by row

# Gaussian elimination
sols <- gauss_elim(M)
print(sols) # Numbers displayed with a default precision
#> [1] -0.3293013 -0.6129265 -1.7584402  1.4130710 -0.9384365
```

The solution found, if correct, should make the product $A\mathbf{x}$ as close as possible to $\mathbf{b}$.

```
# Solution as a column vector
x <- matrix(sols,ncol=1)

# Ax should be equal to b
print(M[,6])
#> [1] -4  3 -2  0 -2
print(M[,1:5] %*% x)
#>      [,1]
#> [1,]   -4
#> [2,]    3
#> [3,]   -2
#> [4,]    0
#> [5,]   -2
```

$A\mathbf{x}$ is indeed equal to $\mathbf{b}$.

1.2 Exercise 02

Use `gauss_elim` to verify that the system,

$$\left\{ \begin{array}{rcr} 3x_1 + 2x_2 - x_3 - 4x_4 & = & 10 \\ x_1 - x_2 + 3x_3 - x_4 & = & -4 \\ 2x_1 + x_2 - 3x_3 & = & 16 \\ -x_2 + 8x_3 - 5x_4 & = & 3 \end{array} \right.$$

has either no solution, or an infinite number of solutions. Use next the function `transform_upper` and quantitative reasoning to prove that the system has, in fact, no solution.

SOLUTION

All we need to carry out Gaussian elimination is the augmented matrix which is, in this case,

$$M = (A|\mathbf{b}) = \begin{pmatrix} 3 & 2 & -1 & -4 & 10 \\ 1 & -1 & 3 & -1 & -4 \\ 2 & 1 & -3 & 0 & 16 \\ 0 & -1 & 8 & -5 & 3 \end{pmatrix}$$

```
# Augmented matrix
raw_data <- c( 3, 2,-1,-4, 10,
               1,-1, 3,-1, -4,
               2, 1,-3, 0, 16,
               0,-1, 8,-5,  3)
M <- matrix(raw_data,ncol=5,byrow=TRUE)

# Gaussian elimination
sols <- gauss_elim(M)
#> This system has no solution or infinite solutions.
```

Clearly the system has either no solution or an infinite number of solutions, as commented in the output. In order to find out which alternative is true, let us write the system in upper diagonal form, using the function `transform_upper`.

```
transform_upper(M) # Upper diagonal equivalent system
#>      [,1]      [,2]      [,3]          [,4]      [,5]
#> [1,]    3  2.000000 -1.000000 -4.000000e+00 10.000000
#> [2,]    0 -1.666667  3.333333  3.333333e-01 -7.333333
#> [3,]    0  0.000000  6.000000 -5.200000e+00  7.400000
#> [4,]    0  0.000000  0.000000 -4.440892e-16 14.500000
```

Considering the finite precision with which the above numbers are expressed, it is not difficult to re-write the reduced upper triangular system as follows:

$$\begin{cases} 3x_1 + 2x_2 - x_3 - 4x_4 & = & 10 \\ -(5/3)x_2 + (10/3)x_3 + (1/3)x_4 & = & -22/3 \\ 6x_3 - (26/5)x_4 & = & 37/5 \\ 0 & = & 29/2 \end{cases}$$

The last equation is never true, therefore the system has no solution. If, on the other hand, the last equation turned out to be $0 = 0$ then we would have been left with three equations and four unknowns. One of the unknowns could have been chosen as a parameter and the other three unknowns expressed as function of that parameter. That would have meant that the system had an infinite number of solutions.

1.3 Exercise 03

The LU decomposition of a square matrix A can be used to solve an algebraic system of linear equations. Starting from the system in matrix form,

$$A\mathbf{x} = \mathbf{b}$$

and using the LU decomposition of A we get:

$$L\,U\,\mathbf{x} = \mathbf{b}$$

If a new set of unknowns, $\mathbf{y}$, defined as

$$\mathbf{y} = U\mathbf{x},$$

is introduced, then the original system becomes a lower triangular system,

$$L\mathbf{y} = \mathbf{b}$$

which can be quickly solved using back-substitution. Eventually, also the solution $\mathbf{x}$ can be found quickly by solving the upper triangular system,

$$U\mathbf{x} = \mathbf{y}$$

as the previously unknown $\mathbf{y}$ have now a numeric value.

Write a program that makes use of the LU decomposition and of the considerations written above, to solve the system presented in Exercise 01. Compare the solution found with the one found when solving Exercise 01.

SOLUTION

The program needs the augmented matrix, $M = (A|\mathbf{b})$, as input. Then it will have to call the function `LUdeco` to find the lower and upper triangular matrices for the decomposition of A. The back substitution can be implemented using the second part of the code in the function `gauss_elim`, as this consisted of the back-substitution part of the algorithm. The same code will have to be implemented twice, the first time to solve $L\mathbf{y} = \mathbf{b}$ and the second time to solve $U\mathbf{x} = \mathbf{y}$. The procedure is included in the following function, called `LUsolve`.

```
LUsolve <- function(M) {
  # Size of the augmented matrix
  tmp <- dim(M)
  n <- tmp[1]

  # Divide the augmented matrix into A and b
  A <- M[,1:n]
  b <- M[,n+1]

  # LU decomposition
  ltmp <- LUdeco(A,"doolittle")

  # L and U matrices
  L <- ltmp$L
  U <- ltmp$U

  # New permuted constants
  # (no permutation in this specific case)
  newb <- b[ltmp$ord]

  # Back-substitution (y)
  y <- rep(0,length=n)
  y[1] <- newb[1]
  for (i in 2:n) {
    y[i] <- newb[i]-sum(y[1:(i-1)]*L[i,1:(i-1)]) # No division
                                                 # by L[i,i]
                                                 # because here
                                                 # all L[i,i]=1
  }

  # Back-substitution (x)
  x <- rep(0,length=n)
  x[n] <- y[n]/U[n,n]
  for (i in (n-1):1) {
```

```
    x[i] <- (y[i]-sum(x[(i+1):n]*U[i,(i+1):n]))/U[i,i]
  }

  return(x)
}

# Apply function to case in Exercise 01
raw_data <- c( 2, 1,-3,-5, 1,-4,
               7,-1, 0, 0,-5, 3,
              -6, 8, 0,-2,-4,-2,
              -3, 8, 1, 8, 6, 0,
               5, 2, 7, 8,-2,-2)
M <- matrix(raw_data,ncol=6,byrow=TRUE)
x1 <- LUsolve(M)
print(x1)
#> [1] -0.3293013 -0.6129265 -1.7584402  1.4130710 -0.9384365

# Comparison with gaussian elimination
x2 <- gauss_elim(M)
print(x2)
#> [1] -0.3293013 -0.6129265 -1.7584402  1.4130710 -0.9384365
```

The function returns the same result as the one found with Gaussian elimination, therefore it is an appropriate function for the task required. There are a couple of observation in place here:

1. The LU decomposition in the code has used the Doolittle method that returned the lower triangular matrix with ones along the diagonal. For this reason, there was no need to divide by `L[i,i]` in the back-substitution section. The function would have equally worked if the Crout method had been used. The only difference is that in that case it is the `U[i,i]` to have ones along the diagonals and this has to be considered for a correct back-substitution.

2. The first back-substitution is, in fact, a forward-substitution because the reduction produces a lower triangular system.

1.4 Exercise 04

Computer time to solve a linear system is very short with modern processors. It is normally less than a second when the system size includes 100 or less equations. As it is very tedious and time-consuming to fill matrices of size 100 or more, we can test solution time of the `gauss_elim` function using matrices with elements generated randomly. Generate a random matrix of size $n = 100$, using the sampling of integers between -5 and +5 and using a fixed random seed for generation, in order to compare your results with those of the solution presented here. Use the seed 7821 for matrix A and the seed 7659 for the constant vector, $\mathbf{b}$. Use the R function `Sys.time` to find out the execution time. Try your procedure with various values of n, say $n = 100, 500, 1000$.

SOLUTION

To fill an $n \times n$ square matrix we need n^2 integers (between -5 and +5). We can use the function `sample` with the parameter `replace=TRUE` as the number of integers to be sampled is larger than the number of integers between -5 and 5. Then matrix A is easily created. A similar procedure goes towards the creation of $\mathbf{b}$; here we only need n random integers, and the vector is formed as a $n \times 1$ matrix.

```
# Size of system
n <- 100

# A and b
```

```r
set.seed(7821) # To reproduce same numbers
itmp <- sample(-5:5,size=n*n,replace=TRUE)
A <- matrix(itmp,ncol=n)
set.seed(7659) # To reproduce same numbers
b <- matrix(sample(-5:5,size=n,replace=TRUE),ncol=1)
M <- cbind(A,b)

## Start time
st <- Sys.time()

# Gaussian elimination
v <- gauss_elim(M)

## End time
et <- Sys.time()

# Duration
print(et-st)
#> Time difference of 0.02603507 secs

# Repeat with n=500
n <- 500
set.seed(7821)
itmp <- sample(-5:5,size=n*n,replace=TRUE)
A <- matrix(itmp,ncol=n)
set.seed(7659)
b <- matrix(sample(-5:5,size=n,replace=TRUE),ncol=1)
M <- cbind(A,b)
st <- Sys.time()
v <- gauss_elim(M)
et <- Sys.time()
print(et-st)
#> Time difference of 1.471656 secs

# Repeat with n=1000
n <- 1000
set.seed(7821)
itmp <- sample(-5:5,size=n*n,replace=TRUE)
A <- matrix(itmp,ncol=n)
set.seed(7659)
b <- matrix(sample(-5:5,size=n,replace=TRUE),ncol=1)
M <- cbind(A,b)
st <- Sys.time()
v <- gauss_elim(M)
et <- Sys.time()
print(et-st)
#> Time difference of 13.23327 secs
```

Note how the augmented matrix is formed from A and $\mathbf{b}$ using the function `cbind` which creates a matrix out of two or more matrices, columnwise. A similar function to combine matrices rowwise is called `rbind`.

1.5 Exercise 05

Write a function that returns coefficients and constants for a tridiagonal system $A\mathbf{x} = \mathbf{b}$. A is a square tridiagonal matrix of size n and $\mathbf{b}$ a vector of length n. The non-zero elements of A and the elements of $\mathbf{b}$ have to be sampled randomly (with repetition) among the numbers,

$$-5, -4, -3, -2, -1, 1, 2, 3, 4, 5$$

Use the integer seed 1243 for the random sampling of matrix A and the integer seed 8731 for the random sampling of vector $\mathbf{b}$.

Using $n = 100, 500, 1000$, compare the execution time to find the solution using the functions `gauss_elim` and `solve_tridiag`. As the last function is supposed to exploit the special structure of a tridiagonal system, execution times for it are expected to be shorter than for `gauss_elim`.

SOLUTION

It is here important to know in advance how many non-zero elements have to be sampled. For matrix A the counting goes as follows. There are n elements along the main diagonal and $n - 1$ elements along each adjacent diagonal. Therefore the number of non-zero elements for A is,

$$n + 2(n - 1) = 3n - 2$$

The vector $\mathbf{b}$ has simply n elements. Thus a function that provides A and $\mathbf{b}$, given the size n, can be written as in the following code.

```
random_tridiag <- function(n,iseed1=1243,iseed2=8731) {
  # Work out number of non-zero elements
  m <- 3*n-2

  # Random integers between -5 and 5 (excluding 0)
  set.seed(iseed1)
  raw_data <- sample(c(-5:-1,1:5),size=m,replace=TRUE)

  # Fill in matrix
  A <- matrix(rep(0,length=n*n),ncol=n)
  A[1,1] <- raw_data[1]
  A[1,2] <- raw_data[2]
  j <- 2
  for (i in 2:(n-1)) {
    A[i,i-1] <- raw_data[j+1]
    A[i,i] <- raw_data[j+2]
    A[i,i+1] <- raw_data[j+3]
    j <- j+3
  }
  A[n,n-1] <- raw_data[j+1]
  A[n,n] <- raw_data[j+2]

  # Fill in vector of constants
  set.seed(iseed2)
  b <- matrix(sample(c(-5:-1,1:5),size=n,replace=TRUE),ncol=1)

  return(list(A=A,b=b))
}
```

We can next use `random_tridiag` to generate A and $\mathbf{b}$, form the augmented matrix M and find out the time to find the solution of the related tridiagonal system using `gauss_elim` and `solve_tridiag`. We can use the three suggested values for n, 100, 500 and 1000.

```r
# n = 100
ltmp <- random_tridiag(n)
M <- cbind(ltmp$A,ltmp$b)
st <- Sys.time()
x1 <- gauss_elim(M)
et <- Sys.time()
print(paste("Time for Gaussian elimination: ",et-st))
#> [1] "Time for Gaussian elimination:  0.151159048080444"
st <- Sys.time()
x2 <- solve_tridiag(M)
et <- Sys.time()
print(paste("Time for Thomas algorithm:     ",et-st))
#> [1] "Time for Thomas algorithm:      0.00800490379333496"

# The two numerical solutions are very close
print(sum(abs(x1-x2)))
#> [1] 1.224499e-11

# n = 500
ltmp <- random_tridiag(n)
M <- cbind(ltmp$A,ltmp$b)
st <- Sys.time()
x1 <- gauss_elim(M)
et <- Sys.time()
print(paste("Time for Gaussian elimination: ",et-st))
#> [1] "Time for Gaussian elimination:  0.15387201309204l"
st <- Sys.time()
x2 <- solve_tridiag(M)
et <- Sys.time()
print(paste("Time for Thomas algorithm:     ",et-st))
#> [1] "Time for Thomas algorithm:      0.00528097152709961"

# The two numerical solutions are very close
print(sum(abs(x1-x2)))
#> [1] 1.224499e-11

# n = 1000
ltmp <- random_tridiag(n)
M <- cbind(ltmp$A,ltmp$b)
st <- Sys.time()
x1 <- gauss_elim(M)
et <- Sys.time()
print(paste("Time for Gaussian elimination: ",et-st))
#> [1] "Time for Gaussian elimination:  0.149728059768677"
st <- Sys.time()
x2 <- solve_tridiag(M)
et <- Sys.time()
print(paste("Time for Thomas algorithm:     ",et-st))
#> [1] "Time for Thomas algorithm:      0.00548291206359863"

# The two numerical solutions are very close
print(sum(abs(x1-x2)))
#> [1] 1.224499e-11
```

Clearly the solution is found much faster using Thomas' algorithm. The reason is that this algorithm involves a much smaller number of numerical operations (additions/subtractions and multiplications/divisions). Whenever the system is known to be tridiagonal, a fast algorithm like Thomas' algorithm should be used to find the solution.

2 Exercises on matrix decomposition

2.1 Exercise 06

Oneway to create arbitrary symmetric matrices is to add a generic matrix to its transpose. Prove that the resulting matrix is, indeed, symmetric and create a function called `symmat` that take the matrix size, n, a set of numbers (not necessarily n) as input and returns a symmetric $n \times n$ matrix, with elements derived from the input numbers.

SOLUTION

Consider the matrix M resulting from the sum of a generic matrix A and its transpose:

$$M = A + A^T$$

It is easy to show that M is symmetric because the transpose of a transpose is the matrix itself:

$$M^T = (A + A^T)^T = A^T + (A^T)^T = A^T + A = A + A^T = M \;\Rightarrow\; M^T = M$$

The R function requested, `symmat`, is relatively straightforward to implement. First a random set of n^2 integers is selected (with the possibility of sampling the same integer more than once) using `sample`. Then the numbers obtained are arranged, column by column, in a generic matrix A. Finally, a symmetric matrix, M, is obtained as sum of A and its transpose.

```
symmat <- function(rrr,n) {
  # rrr is a vector of numbers from which to sample
  #   n is the generic square matrix size

  # Elements to fill generic matrix A
  rtmp <- sample(rrr,size=n*n,replace=TRUE)
  A <- matrix(rtmp,ncol=n)

  # Symmetric matrix
  M <- A+t(A)

  return(M)
}
```

The function just created can now be tested. As no specific size and or type of input numbers were specified, the results shown here are in general going to be different, depending on what input was provided to the function.

```
# First set of numbers (just two values!)
rrr <- c(0,1)

# Test function. Matrix size is n=4
M <- symmat(rrr,4)
print(M)
#>      [,1] [,2] [,3] [,4]
#> [1,]    0    0    1    2
#> [2,]    0    0    0    2
#> [3,]    1    0    2    1
```

```
#> [4,]    2    2    1    2

# Second set of numbers (real between 0 and 1)
rrr <- seq(0,1,length.out=11)

# Test function. Matrix size is n=5
M <- symmat(rrr,5)
print(M)
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]  1.8  0.4  1.8  0.8  1.6
#> [2,]  0.4  1.6  1.4  0.4  0.1
#> [3,]  1.8  1.4  1.0  1.2  0.6
#> [4,]  0.8  0.4  1.2  1.4  0.9
#> [5,]  1.6  0.1  0.6  0.9  0.8
```

The matrices obtained are indeed symmetric. The function created works as expected.

2.2 Exercise 07

Apply the Cholesky decomposition to the following symmetric matrix:

$$A = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 2 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Calculate the determinant of A using the result of the decomposition and verify that the result is correct with both `condet` and `det`.

SOLUTION

The matrix is symmetric, but we have to see whether it is positive definite. The leading principal minors are, proceding along the diagonal of A,

$$|2| = 2 \quad , \quad \begin{vmatrix} 2 & 1 \\ 1 & 2 \end{vmatrix} = 3 \quad , \quad \begin{vmatrix} 2 & 1 & 0 \\ 1 & 2 & 0 \\ 0 & 1 & 1 \end{vmatrix} = 3$$

This can also be worked out using R.

```
# Matrix A
A <- matrix(c(2,1,0,1,2,1,0,0,1),ncol=3)
print(A)
#>      [,1] [,2] [,3]
#> [1,]    2    1    0
#> [2,]    1    2    0
#> [3,]    0    1    1

# Minors
for (i in 1:3) {
  mm <- matrix(A[1:i,1:i],ncol=i)
  print(mm)
  print(det(mm))
}
#>      [,1]
#> [1,]    2
#> [1] 2
#>      [,1] [,2]
```

```
#> [1,]    2    1
#> [2,]    1    2
#> [1] 3
#>      [,1] [,2] [,3]
#> [1,]    2    1    0
#> [2,]    1    2    0
#> [3,]    0    1    1
#> [1] 3
```

In any case, all the minors of A are positive and therefore A is positive definite. Being also symmetric, we can apply Cholesky decomposition and find the lower triangular matrix L and its transpose upper triangular, L^T. Using `chol` one has to remember that the function returns the upper triangular, rather than the lower triangular.

```
# Cholesky decomposition
L <- chol(A)
print(L) # Upper triangular
#>          [,1]      [,2] [,3]
#> [1,] 1.414214 0.7071068    0
#> [2,] 0.000000 1.2247449    0
#> [3,] 0.000000 0.0000000    1
print(t(L)) # Lower triangular
#>           [,1]     [,2] [,3]
#> [1,] 1.4142136 0.000000    0
#> [2,] 0.7071068 1.224745    0
#> [3,] 0.0000000 0.000000    1

# Their product yields A
print(t(L) %*% L)
#>      [,1] [,2] [,3]
#> [1,]    2    1    0
#> [2,]    1    2    0
#> [3,]    0    0    1
```

The determinant of A can be calculated easily using the decomposition because:

1. the determinant of the product of two matrices is the product of their determinants;
2. the determinant of a triangular matrix is the product of the elements of its diagonal;
3. the determinant of a matrix transpose is equal to the determinant of the matrix.

Therefore we can compute the determinant of A using the L derived from its Cholesky decomposition, multiply the elements of its diagonal and squaring the result.

```
# Product of the elements along the diagonal of L
detA <- prod(diag(L))

# Determinant of A
detA <- detA*detA
print(detA)
#> [1] 3

# Compare with the determinant computed using condet and det
print(condet(A))
#> [1] 3
print(det(A))
```

```
#> [1] 3
```

The three results coincide, as expected.

2.3 Exercise 08

Apply the Cholesky decomposition to the following symmetric matrix:

$$A = \begin{pmatrix} 2 & 1 & -3 \\ 1 & 2 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Does the `chol` function return a numeric result? Why?

SOLUTION

Let us work out, first of all, if the symmetric matrix is positive definite, using R.

```
# Matrix A
A <- matrix(c(2,1,0,1,2,1,-3,0,1),ncol=3)
print(A)
#>      [,1] [,2] [,3]
#> [1,]    2    1   -3
#> [2,]    1    2    0
#> [3,]    0    1    1

# Minors
for (i in 1:3) {
  mm <- matrix(A[1:i,1:i],ncol=i)
  print(mm)
  print(det(mm))
}
#>      [,1]
#> [1,]    2
#> [1] 2
#>      [,1] [,2]
#> [1,]    2    1
#> [2,]    1    2
#> [1] 3
#>      [,1] [,2] [,3]
#> [1,]    2    1   -3
#> [2,]    1    2    0
#> [3,]    0    1    1
#> [1] 0
```

Not all the leading principal minors are positive. We should expect something unusual happening when the function `chol` is applied to A.

```
# Is Cholesky valid?
L <- chol(A)
#> Error in chol.default(A): the leading minor of order 3 is not positive
```

It is indeed true that the leading principal minor of order 3 is not a positive number, as it is zero.

2.4 Exercise 09

The QR decomposition is at the foundation of most methods to find the eigenvalues of a matrix. The algorithm is a cyclical repetition of the following steps, provided that the matrix whose eigenvalues have to

be found is the square matrix of size n, A:

1. Find the QR decomposition of A. This is cycle 1 of the algorithm, and A is called A_0. The QR decomposition yields,
$$A_0 = Q_0 \, R_0$$
2. Build the new matrix, A_1, for cycle 1,
$$A_1 = R_0 \, Q_0$$
3. Check that all the elements in the lower triangular part of A_1 are close to 0 (practically below a fixed threshold `zero_cut`). If they are all close to zero the algorithm has terminated successfully and the eigenvalues of A are the elements on the diagonal of A_1. Otherwise, go back to step 1, where in cycle 2 A_1 replaces A_0. In general, in cycle i, A_{i-1} replaces A_{i-2}.
4. If, after a pre-established number of cycles, `nmax`, not all elements of the lower triangular part of A_i, at cycle i, fall below the threshold `zero_cut`, the algorithm has not achieved convergence and it is not possible to find the eigenvalues.

Write a function, called `eigenQR`, that implements the steps above to find the eigenvalues of an input matrix, A. Besides the matrix, the function takes in the threshold, `zero_cut`, basically a small number (default is `1e-6`) and the maximum number of cycles, `nmax` (default is 1000). Apply the function to find the eigenvalues of,

$$A = \begin{pmatrix} -2 & -4 & 2 \\ -2 & 1 & 2 \\ 4 & 2 & 5 \end{pmatrix}$$

SOLUTION

A possible implementation of the function `eigenQR` is presented here.

```
eigenQR <- function(A,zero_cut=1e-6,nmax=1000) {
  # Size of A
  n <- dim(A)[1]

  # Loop
  izero <- 100
  icycle <- 1
  for (icycle in 1:nmax) {
    QR <- qr(A)
    Q <- qr.Q(QR)
    R <- qr.R(QR)
    A <- R %*% Q
    eigs <- diag(A)
    s <- 0
    for (i in 2:n) {
      s <- s+sum(abs(A[i,1:(i-1)]))
    }
    if (s < izero) izero <- s
    if (izero < zero_cut) {
      return(list(eigs=eigs,res=s))
    }
  }

  # Algorithm has not converged
  cat("The algorithm has not converged")
```

```
  return(list(eigs=diag(A),res=s))
}
```

The function can be written in any other way, as long as it delivers what asked. In fact, let us apply it to the matrix provided.

```
# Matrix provided
A <- matrix(c(-2,-2,4,-4,1,2,2,2,5),ncol=3)
print(A)
#>      [,1] [,2] [,3]
#> [1,]   -2   -4    2
#> [2,]   -2    1    2
#> [3,]    4    2    5

# Apply function to matrix
lEigens <- eigenQR(A)

# Eigenvalues
print(lEigens$eigs)
#> [1]  6 -5  3

# What is the sum of the absolute values of all elements
# in the lower triangular part of the final A(n) matrix?
print(lEigens$res)
#> [1] 8.423575e-07

# Check with built in function
lambdas <- eigen(A)
print(lambdas$values)
#> [1]  6 -5  3
```

In this case, the algorithm suggested works.

2.5 Exercise 10

What would be the result of applying `eigenQR` to the following matrix,

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}?$$

Try to understand what the issue is in this case, by tracing it back, through inspection of the function, to what is causing it. Verify that the eigenvalues are found with the default R function, `eigen`.

SOLUTION

Let us apply `eigenQR` to the matrix provided, without worrying to much of what the result can be.

```
# Matrix provided
A <- matrix(c(0,1,1,0),ncol=2)
print(A)
#>      [,1] [,2]
#> [1,]    0    1
#> [2,]    1    0

# Attempt at obtaining eigenvalues
lEigens <- eigenQR(A)
#> The algorithm has not converged
```

```r
print(lEigens$eigs)
#> [1] 0 0
print(lEigens$res)
#> [1] 1
```

The algorithm has clearly not converged. The residual is relatively large (`res = 1`). The default number of cycles is 1000; was the residual larger, say at cycle 10, or 5, or 2? We can check that by changing the `nmax` parameter.

```r
# eigenQR again, with nmax=10,5,2
lEigens <- eigenQR(A,nmax=10)
#> The algorithm has not converged
print(lEigens$eigs)
#> [1] 0 0
print(lEigens$res)
#> [1] 1
```

So, the residual is 1 already at the first few cycles. It seems that the algorithm gets *trapped* immediately in some sort of fixed matrix configuration. To explore this further, we will need to implement the algorithm one step at a time. It is simply a matter of carrying out the QR decompositionn manually and build the new matrix $R\,Q$, to see what that entails.

```r
# First QR decomposition
QR <- qr(A)
Q <- qr.Q(QR)
print(Q)
#>      [,1] [,2]
#> [1,]    0   -1
#> [2,]   -1    0
R <- qr.R(QR)
print(R)
#>      [,1] [,2]
#> [1,]   -1    0
#> [2,]    0   -1

# New matrix
Anew <- R %*% Q
print(Anew)
#>      [,1] [,2]
#> [1,]    0    1
#> [2,]    1    0
```

With the above simple steps we have revealed what went wrong with the matrix A provided, because the new matrix is exactly equal to A. As the lower-triangular only element of the new matrix is 1, so is the residual which is, therefore, going to be always 1. Essentially, the algorithm gets trapped in a never-ending loop where the residual does not get smaller beyond a given value (1 in this case). This is the risk associated with using the algorithm above. The algorithm is, in fact, known as the *unshifted QR algorithm* because a modification of its steps that, among other things, shifts the elements of the matrix of a given quantity, either speeds up its convergence rate, or avoids situations in which convergence is never achieved. Improvements of the QR algorithm have been investigated and implemented and modern functions in general avoid the type of problems met with ths specific example. The eigenvalues are for instance readily found with `eigen`.

```r
lEigens <- eigen(A)
print(lEigens)
#> eigen() decomposition
#> $values
```

```
#> [1]  1 -1
#>
#> $vectors
#>           [,1]       [,2]
#> [1,] 0.7071068 -0.7071068
#> [2,] 0.7071068  0.7071068
```

2.6 Exercise 11

The requirement for the algorithm of Exercise 09 did not included the eigenvectors as part of the output. In fact, obtaining the eigenvectors in general implies using additional algorithmical steps that use matrices in special forms. Rather than considering the general case, we will study the special case in which the starting matrix is symmetric. As explained in the theory of the QR algorithm, the similarity transformation between A and the A_k matrix obtained at cycle k of the algorithm is of the form,

$$A = (Q_0\, Q_1\, \cdots\, Q_k)\, A_k\, (Q_0\, Q_1\, \cdots\, Q_k)^T$$

The quantities in parenthesis can be indicated simply with the letter Q, as they are single matrices. The expression of the algorithm after k iteration is thus

$$A = QA_kQ^T$$

In general, A_k is an upper triangular matrix and the columns of Q are not the eigenvectors of A. The only thing that can be stated with accuracy is that the elements on the diagonal of A_k are good approximations of A's eigenvalues. If A is symmetric, though, A_k is a diagonal matrix and the columns of Q are the orthonormal eigenvectors of A. To see this let's consider that for a symmetric matrix $A^T = A$. Therefore, taking the transpose of the expression above we get,

$$A^T = QA_k^TQ^T = A = QA_kQ^T \quad \Rightarrow \quad A_k^T = A_k$$

But A_k is an upper triangular matrix and this can be at the same time symmetric only if the off-diagonal elements are all zero. In conclusion, A_k is a diagonal matrix containing the eigenvalues of A. And, accordingly, the columns of Q will be its ordered eigenvectors.

Modify the algorithm created for Exercise 09 so that the matrix of eigenvectors forms part of its output, and apply it to the symmetric matrix, $B = A + A^T$, where A was introduced in that exercise. Find an effective way to show that the eigenvectors found do correspond to the eigenvalues obtained.

SOLUTION

The modified algorithm just have to keep track of each matrix Q of the QR decomposition in order to form the product,

$$Q_0\, Q_1\, \cdots\, Q_k$$

Then an initial matrix that starts off the multiplication chain, $Q_0\, Q_1\, \cdots\, Q_k$, is needed. Subsequent matrices will build up in the loop through matrix multiplication. Therefore the initial matrix will have to be the identity matrix so that the first matrix multiplication will yield Q_0 (indeed it's $Q_0\, I = Q_0$). The algorithm is here illustrated in a modified `eigenQR` function called `eigenQR2`.

```
eigenQR2 <- function(A,zero_cut=1e-6,nmax=1000) {
  # Size of A
  n <- dim(A)[1]

  # Eventually, V will be the matrix with eigenvectors.
  # Initially, V is the identity matrix
  V <- diag(n)
```

```
  # Loop
  izero <- 100
  icycle <- 1
  for (icycle in 1:nmax) {
    QR <- qr(A)
    Q <- qr.Q(QR)
    R <- qr.R(QR)
    V <- V %*% Q
    A <- R %*% Q
    eigs <- diag(A)
    s <- 0
    for (i in 2:n) {
      s <- s+sum(abs(A[i,1:(i-1)]))
    }
    if (s < izero) izero <- s
    if (izero < zero_cut) {
      return(list(eigs=eigs,V=V,A=A,res=s))
    }
  }

  # Algorithm has not converged
  cat("The algorithm has not converged")

  return(list(eigs=diag(A),V=V,A=A,res=s))
}
```

Let us apply this new function to the suggested matrix.

```
# Suggested matrix
A <- matrix(c(-2,-2,4,-4,1,2,2,2,5),ncol=3) # Old matrix
B <- A + t(A) # Symmetric matrix

# Apply the new function and find eigenvalues and eigenvectors
lEigens <- eigenQR2(B)

# Eigenvalues
lbda <- lEigens$eigs

# Matrix whose columns are the eigenvectors
V <- lEigens$V

# Display values
print(lbda)
#> [1]  12.570240 -10.270905   5.700665
print(V)
#>            [,1]       [,2]        [,3]
#> [1,] -0.2667871  0.7987391  0.53929624
#> [2,] -0.2049283  0.4997661 -0.84156892
#> [3,] -0.9417160 -0.3350368  0.03035313
```

We can verify that the obtained scalars `lbda` and vectors `V` are eigenvalues and eigenvectors through the definition,

$$B\mathbf{v} = \lambda\mathbf{v},$$

where the eigenvector $\mathbf{v}$ corresponding to the eigenvalue λ is a 3×1 matrix. Considering that in R the

division of two vectors is done in a pointwise fashion, we should have,

$$B\mathbf{v}/\mathbf{v} = \begin{pmatrix} \lambda \\ \lambda \\ \lambda \end{pmatrix}.$$

This can be applied to the three columns at the same time if the single eigenvector $\mathbf{v}$ is replaced by the matrix V. In this case we should have,

$$BV/V = \begin{pmatrix} \lambda_1 & \lambda_2 & \lambda_3 \\ \lambda_1 & \lambda_2 & \lambda_3 \\ \lambda_1 & \lambda_2 & \lambda_3 \end{pmatrix}$$

And indeed,

```
# Result from the application of QR
print(B %*% V / V)
#>          [,1]      [,2]     [,3]
#> [1,] 12.57024 -10.27091 5.700665
#> [2,] 12.57024 -10.27091 5.700665
#> [3,] 12.57024 -10.27090 5.700665

# Eigenvalues
print(lbda)
#> [1]  12.570240 -10.270905   5.700665

# Eigenvalues with a different method
print(eigen(B))
#> eigen() decomposition
#> $values
#> [1]  12.570240   5.700665 -10.270905
#>
#> $vectors
#>           [,1]        [,2]       [,3]
#> [1,] 0.2667871  0.53929624  0.7987391
#> [2,] 0.2049283 -0.84156892  0.4997661
#> [3,] 0.9417160  0.03035313 -0.3350368
```

It is also possible to verify that the three column vectors are orthonormal, using the inner product.

```
for (i in 1:3) {
  for (j in 1:3) {
    print(sum(V[,i]*V[,j]))
  }
}
#> [1] 1
#> [1] 5.967449e-16
#> [1] 6.245005e-17
#> [1] 5.967449e-16
#> [1] 1
#> [1] 1.752071e-16
#> [1] 6.245005e-17
#> [1] 1.752071e-16
#> [1] 1
```

Equivalently, the property of an orthogonal matrix can be exploited with a single matrix product.

```r
# This should be a 3X3 identity matrix
print(V %*% t(V))
#>               [,1]          [,2]          [,3]
#> [1,]  1.000000e+00 -1.110223e-16 -3.504141e-16
#> [2,] -1.110223e-16  1.000000e+00 -9.714451e-17
#> [3,] -3.504141e-16 -9.714451e-17  1.000000e+00
```

2.7 Exercise 12

A square matrix A can be *diagonalised* if a diagonal matrix D and another, invertible, square matrix P can be found such that the following relation holds:

$$A = PDP^{-1} \tag{1}$$

When this happens, D has the eigenvalues of A along its diagonal, while the columns of P are, in order, the corresponding eigenvectors. This can be seen if the above equation is multiplied, on the left, by P,

$$AP = PD,$$

and if P is re-written in terms of its column vectors, $\mathbf{v}_i$,

$$A(\mathbf{v}_1 \ \ldots \ \mathbf{v}_n) = (\mathbf{v}_1 \ \ldots \ \mathbf{v}_n) \begin{pmatrix} \lambda_1 & \ldots & 0 \\ \ldots & \ldots & \ldots \\ 0 & \ldots & \lambda_n \end{pmatrix}$$

$$\Downarrow$$

$$(A\mathbf{v}_1 \ \ldots \ A\mathbf{v}_n) = (\lambda_1\mathbf{v}_1 \ \ldots \ \lambda_n\mathbf{v}_n)$$

Clearly, the expression just derived is a set of n eigenvalue equations for n eigenvectors of A.

Using the function `eigen`, write the diagonal matrix D corresponding to the 4×4 matrix A generated by sampling randomly the integers $-1, 0, 1$ (use seed 1188). Find also the expression of P and, using matrix multiplication in R, show that $AP = PD = DP$.

SOLUTION

The matrix is generated using `sample`.

```r
# Matrix A
set.seed(1188)
A <- matrix(sample(-1:1,size=16,replace=TRUE),ncol=4)
print(A)
#>      [,1] [,2] [,3] [,4]
#> [1,]    1   -1   -1    0
#> [2,]    0    1    0    0
#> [3,]    1    1   -1   -1
#> [4,]   -1    0    1    0
```

The eigenvalues and eigenvectors can be found using `eigen`. The printout of these objects can be difficult to read, as in general the result will include complex numbers (characterised by the presence of the letter `i`, representing the imaginary unit).

```r
# Eigenvalues and eigenvectors in the named list lEigen
lEigen <- eigen(A)

# Eigenvalues
lbda <- lEigen$values
print(lbda)
```

```
#> [1]  2.220446e-16+1i  2.220446e-16-1i  1.000000e+00+0i -3.045239e-16+0i

# Corresponding eigenvectors
P <- lEigen$vectors
print(P)
#>                       [,1]                    [,2]          [,3]             [,4]
#> [1,]  0.3535534+0.3535534i  0.3535534-0.3535534i -0.7559289+0i  7.071068e-01+0i
#> [2,]  0.0000000+0.0000000i  0.0000000+0.0000000i  0.3779645+0i  0.000000e+00+0i
#> [3,]  0.7071068+0.0000000i  0.7071068+0.0000000i -0.3779645+0i  7.071068e-01+0i
#> [4,] -0.3535534-0.3535534i -0.3535534+0.3535534i  0.3779645+0i -2.220446e-16+0i
```

The four eigenvalues are,

$$\lambda_1 = \mathrm{i} \quad , \quad \lambda_2 = -\mathrm{i} \quad , \quad \lambda_3 = 1 \quad , \quad \lambda_4 = 0$$

To verify that thesecorrespond to the columns of P we should form both expressions,

$$A\mathbf{v}_i \quad \text{and} \quad \lambda_i \mathbf{v}_i$$

and verify that they coincide. The function `cbind` can be used for a better visual comparison.

```
# Verify eigen-equations for all eigenvalues
for (i in 1:4) {
  cat(paste("Eigenvalue",i,"\n"))
  print(cbind(A %*% P[,i],lbda[i]*P[,i]))
  cat("\n")
}
#> Eigenvalue 1
#>                         [,1]                      [,2]
#> [1,] -3.535534e-01+0.3535534i -3.535534e-01+0.3535534i
#> [2,]  0.000000e+00+0.0000000i  0.000000e+00+0.0000000i
#> [3,]  4.440892e-16+0.7071068i  1.570092e-16+0.7071068i
#> [4,]  3.535534e-01-0.3535534i  3.535534e-01-0.3535534i
#>
#> Eigenvalue 2
#>                         [,1]                      [,2]
#> [1,] -3.535534e-01-0.3535534i -3.535534e-01-0.3535534i
#> [2,]  0.000000e+00+0.0000000i  0.000000e+00+0.0000000i
#> [3,]  4.440892e-16-0.7071068i  1.570092e-16-0.7071068i
#> [4,]  3.535534e-01+0.3535534i  3.535534e-01+0.3535534i
#>
#> Eigenvalue 3
#>               [,1]          [,2]
#> [1,] -0.7559289+0i -0.7559289+0i
#> [2,]  0.3779645+0i  0.3779645+0i
#> [3,] -0.3779645+0i -0.3779645+0i
#> [4,]  0.3779645+0i  0.3779645+0i
#>
#> Eigenvalue 4
#>                  [,1]             [,2]
#> [1,] -1.110223e-16+0i -2.153309e-16+0i
#> [2,]  0.000000e+00+0i  0.000000e+00+0i
#> [3,]  1.110223e-16+0i -2.153309e-16+0i
#> [4,]  1.110223e-16+0i  6.761790e-32+0i

# In relation to the last eigenvalue, consider that
```

```
# the corresponding eigenvector is not a null vector
print(P[,4])
#> [1]  7.071068e-01+0i  0.000000e+00+0i  7.071068e-01+0i -2.220446e-16+0i
print(sum(Conj(P[,4])*P[,4]))
#> [1] 1+0i
```

The previous set of eigen-equations can be verified just with one matrix operation, once the diagonal matrix D of eigenvalues is formed.

```
# All eigen-equations true when AP=PD (=DP)
D <- diag(lbda,nrow=4,ncol=4)
print(D)
#>                  [,1]             [,2] [,3]             [,4]
#> [1,] 2.220446e-16+1i 0.000000e+00+0i 0+0i  0.000000e+00+0i
#> [2,] 0.000000e+00+0i 2.220446e-16-1i 0+0i  0.000000e+00+0i
#> [3,] 0.000000e+00+0i 0.000000e+00+0i 1+0i  0.000000e+00+0i
#> [4,] 0.000000e+00+0i 0.000000e+00+0i 0+0i -3.045239e-16+0i
print(cbind(A %*% P,P %*% D))
#>                           [,1]                       [,2]          [,3]
#> [1,] -3.535534e-01+0.3535534i -3.535534e-01-0.3535534i -0.7559289+0i
#> [2,]  0.000000e+00+0.0000000i  0.000000e+00+0.0000000i  0.3779645+0i
#> [3,]  4.440892e-16+0.7071068i  4.440892e-16-0.7071068i -0.3779645+0i
#> [4,]  3.535534e-01-0.3535534i  3.535534e-01+0.3535534i  0.3779645+0i
#>                  [,4]                       [,5]                       [,6]
#> [1,] -1.110223e-16+0i -3.535534e-01+0.3535534i -3.535534e-01-0.3535534i
#> [2,]  0.000000e+00+0i  0.000000e+00+0.0000000i  0.000000e+00+0.0000000i
#> [3,]  1.110223e-16+0i  1.570092e-16+0.7071068i  1.570092e-16-0.7071068i
#> [4,]  1.110223e-16+0i  3.535534e-01-0.3535534i  3.535534e-01+0.3535534i
#>                [,7]             [,8]
#> [1,] -0.7559289+0i -2.153309e-16+0i
#> [2,]  0.3779645+0i  0.000000e+00+0i
#> [3,] -0.3779645+0i -2.153309e-16+0i
#> [4,]  0.3779645+0i  6.761790e-32+0i
```

2.8 Exercise 13

Consider the full set of *Pauli matrices*,

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad , \quad \sigma_y = \begin{pmatrix} 0 & -\mathrm{i} \\ \mathrm{i} & 0 \end{pmatrix} \quad , \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Using the function `eigen`, verify that the eigenvalues of each matrix are $+1$ and -1 and that the corresponding eigenvectors are,

$$\psi_{x+} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad , \quad \psi_{x-} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ -1 \end{pmatrix}$$
$$\psi_{y+} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ \mathrm{i} \end{pmatrix} \quad , \quad \psi_{y-} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ -\mathrm{i} \end{pmatrix}$$
$$\psi_{z+} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad , \quad \psi_{z-} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

SOLUTION

First we have to define the three Pauli matrices. The only difficulty with that could be the definition in R of a complex number. Using the syntax, $a + bi$, where a and b are numeric variables, any complex number can be defined.

```r
# Pauli sigma_x
sigx <- matrix(c(0,1,1,0),ncol=2)

# Pauli sigma_y
sigy <- matrix(c(0,0+1i,0-1i,0),ncol=2)

# Pauli sigma_z
sigz <- matrix(c(1,0,0,-1),ncol=2)

print(sigx)
#>      [,1] [,2]
#> [1,]    0    1
#> [2,]    1    0
print(sigy)
#>      [,1] [,2]
#> [1,] 0+0i 0-1i
#> [2,] 0+1i 0+0i
print(sigz)
#>      [,1] [,2]
#> [1,]    1    0
#> [2,]    0   -1
```

Then the calculation of the eigenvectors and eigenvalues is readily done using `eigen`.

```r
# Eigenvalues and eigenvectors of sigma_x
print(eigen(sigx))
#> eigen() decomposition
#> $values
#> [1]  1 -1
#>
#> $vectors
#>           [,1]       [,2]
#> [1,] 0.7071068 -0.7071068
#> [2,] 0.7071068  0.7071068

# Eigenvalues and eigenvectors of sigma_y
print(eigen(sigy))
#> eigen() decomposition
#> $values
#> [1]  1 -1
#>
#> $vectors
#>                       [,1]                  [,2]
#> [1,] -0.7071068+0.0000000i -0.7071068+0.0000000i
#> [2,]  0.0000000-0.7071068i  0.0000000+0.7071068i

# Eigenvalues and eigenvectors of sigma_z
print(eigen(sigz))
#> eigen() decomposition
#> $values
#> [1]  1 -1
#>
#> $vectors
#>      [,1] [,2]
```

```
#> [1,]   -1    0
#> [2,]    0   -1
```

The eigenvalues are ± 1, as expected, but the eigenvectors are not exactly what was defined in the problem. The signs of some components can differ. But this is coherent with the scaling arbitrariety in the definition of eigenvectors. The only important feature of the values found is that the eigenvectors of each matrix are orthonormal. In terms of the diagonalising matrix P, this means that P is a unitary matrix:

$$P^\dagger P = PP^\dagger = I$$

```
# The matrices formed by the eigenvectors
# of each Pauli matrix, are unitary
ltmp <- eigen(sigx)
Px <- ltmp$vectors
print(t(Px) %*% Conj(Px))
#>      [,1] [,2]
#> [1,]    1    0
#> [2,]    0    1
ltmp <- eigen(sigy)
Py <- ltmp$vectors
print(t(Py) %*% Conj(Py))
#>      [,1] [,2]
#> [1,] 1+0i 0+0i
#> [2,] 0+0i 1+0i
ltmp <- eigen(sigz)
Pz <- ltmp$vectors
print(t(Pz) %*% Conj(Pz))
#>      [,1] [,2]
#> [1,]    1    0
#> [2,]    0    1
```

2.9 Exercise 14

After having generated a 12×10 random matrix M with the function `sample` and starting from the set $\{-2, -1, 0, 1, 2\}$ and with seed 2673, find its singular values without using the function `svd`. Verified then that the values found are correct by using the function `svd`.

SOLUTION

In Appendix B it is explained that the singular values can be found as square roots of the eigenvalues of the matrix $M_u = MM^\dagger$ or the matrix $M_v = M^\dagger M$. Mu is a 12×12 symmetric (Hermitian) matrix and has 12 eigenvalues, the two smallest being zero. Mv is a 10×10 symmetric matrix with 10 eigenvalues, exactly equal to the non-zero eigenvalues of Mu.

Let's first generate the random matrix M.

```
# Fixed starting seed
set.seed(2673)

# Pool for sampling
x <- -2:2

# Random 12 X 10 matrix
M <- matrix(sample(x,size=120,replace=TRUE),ncol=10)
print(dim(M))
#> [1] 12 10
```

Then let us find the eigenvalues of both Mu and Mv. The square root of the 10 largest ones are the requested singular values. Let us save them in a variable for later comparison.

```
# Mu: 12X12 symmetric matrix
Mu <- M %*% t(M)

# The singular values are square root of the
# first 10 (non-zero) eigenvalues (the last two
# are approximations to zero and, as such, can be also
# negative numbers)
lEigen <- eigen(Mu)
print(sqrt(lEigen$values[1:10]))
#>  [1] 8.098100 6.933402 5.906714 5.467179 5.177016 4.587080 3.964595 2.951754
#>  [9] 2.171782 1.256623

# Mv: 10X10 symmetric matrix
Mv <- t(M) %*% M

# The singular values are square root of the
# 10 eigenvalues. No zeros involved here
lEigen <- eigen(Mv)
sigmas <- sqrt(lEigen$values) # Store for later comparison
print(sigmas)
#>  [1] 8.098100 6.933402 5.906714 5.467179 5.177016 4.587080 3.964595 2.951754
#>  [9] 2.171782 1.256623
```

Finally, let us calculate the singular values using `svd`.

```
# normal SVD of the original matrix
lSVD <- svd(M,nu=12,nv=10)
print(lSVD$d)
#>  [1] 8.098100 6.933402 5.906714 5.467179 5.177016 4.587080 3.964595 2.951754
#>  [9] 2.171782 1.256623
print(sigmas) # Compare with previous result
#>  [1] 8.098100 6.933402 5.906714 5.467179 5.177016 4.587080 3.964595 2.951754
#>  [9] 2.171782 1.256623
```

The two sets of values, calculated with different functions, do coincide.

2.10 Exercise 15

Create a 20×20 real random matrix, A, using numbers generated from a normal distribution with mean 2 and standard deviation 0.5, and making sure to start the random generation with integer seed 3367 (fill the matrix column by column). Next, create a 20×1 real column vector, $\mathbf{b}$, filled with a random sample with repetitions from the set $\{-1, 1\}$, and making sure to start the sampling with integer seed 4189.

(a) Attempt to solve the linear system $A\mathbf{x} = \mathbf{b}$ using the functions `PJacobi` and `GSeidel` in the `comphy` package. Notice that the matrix A is not diagonally dominant.

(b) Turn A into a diagonally dominant matrix by adding a same positive integer to all the elements along its diagonal. What is the smallest positive integer that makes A diagonally dominant?

(c) With the diagonally dominant matrix found in part b, find the numerical solution of the system, using both `PJacobi` and `GSeidel`.

SOLUTION

The matrix and the column vector are generated with the following code.

```r
# Fixed starting seed for A
set.seed(3367)
A <- matrix(rnorm(400,mean=2,sd=0.5),ncol=20)

# Fixed starting seed for b
set.seed(4189)
b <- matrix(sample(c(-1,1),size=20,replace=TRUE),ncol=1)
```

a As the matrix of the system is not diagonally dominant, convergence might not be obtained with either the Jacobi or the Gauss-Seidel method. In any case we will have to force both functions to attempt the solution.

```r
# Initially both functions do not proceed
# as the matrix is not diagonally dominant
xP <- PJacobi(A,b)
#> The matrix of coefficients is not diagonally dominant.
#> Not attempting solution.
xG <- GSeidel(A,b)
#> The matrix of coefficients is not diagonally dominant.
#> Not attempting solution.

#
# Then the keyword 'ddominant' is switched to
# 'FALSE' so to make the functions to attempt
# and find a solution
xP <- PJacobi(A,b,ddominant=FALSE)
#> The matrix of coefficients is not diagonally dominant.
#> Attempting solution anyway...
#>
#> The increment is getting larger and larger.
#> Max value of increment at cycle 90: 1.17745933879921e+100
xG <- GSeidel(A,b,ddominant=FALSE)
#> The matrix of coefficients is not diagonally dominant.
#> Attempting solution anyway...
#>
#> The increment is getting larger and larger.
#> Max value of Delta_X at cycle 463: 1.48722792038218e+100
```

b The matrix A is now changed with all elements on the diagonal being progressively increased by one unit, until A becomes diagonally dominant. A function to determine whether a matrix is diagonally dominant must be assembled first. A possibility to do this quickly is to copy, paste and modify the code used in `PJacobi`. The result is here presented in the following code.

```r
check_dominant <- function(a) {
 # To check that A is diagonally dominant

 # Size of matrix
 tmp <- dim(A)
 n <- tmp[1]

 # Pivoting if not correct order
 # Swap rows to have largest values on diagonal
 for (i in 1:(n-1)) {
   idx <- which(abs(A[i:n,i]) == max(abs(A[i:n,i])))
   idx <- idx[length(idx)] # If more than one, pick the last
   idx <- i+idx-1
```

```
    N <- A[i,]
    A[i,] <- A[idx,]
    A[idx,] <- N
    N <- b[i]
    b[i] <- b[idx]
    b[idx] <- N
  }

  # Each a_{ii} greater than sum of rest
  ans <- TRUE
  for (i in 1:n) {
    ff <- abs(A[i,i])
    ss <- sum(abs(A[1,]))-ff
    if (ff <= ss) ans <- FALSE
  }

  return(ans)
}
```

The function returns `TRUE` or `FALSE` depending on whether A is diagonally dominant or not. We know, for instance, that A is not diagonally dominant. Let us check that.

```
ans <- check_dominant(A)
print(ans)
#> [1] FALSE
```

Next, a loop can be set up in which the elements on the diagonal are increased of one unit at each cycle. At each cycle the new matrix is checked for diagonal dominance. The iterations are stopped when the matrix is found diagonally dominant.

```
ans <- TRUE
i <- 0
while (ans) {
  i <- i+1
  diag(A) <- diag(A)+i
  ans <- !(check_dominant(A))
}
print(i)
#> [1] 9
```

So the answer to this question is that the positive integer making A dominant is 9.

c

It is then easy to find the solution using the two methods, as the current A is diagonally dominant.

```
# A is diagonally dominant
ans <- check_dominant(A)
print(ans)
#> [1] TRUE

# Solution using Jacobi
xP <- PJacobi(A,b)
#> Number of cycles needed to converge: 36
#> Last relative increment: 9.39080491257904e-07

# Solution using Gauss-Seidel
```

```r
xG <- GSeidel(A,b)
#> Number of cycles needed to converge: 10
#> Last relative increment: 2.38275426389656e-07

# The two solutions are equal
absdif <- sum(abs(xP-xG))
print(absdif)
#> [1] 1.611361e-07
```

2.11 Exercise 16

To study the convergence of both the Jacobi and the Gauss-Seidel algorithm, the norm of the approximate solutions, $\mathbf{x}^{(i)}$, can be measured and plotted with respect to the iteration number. This is one of the many ways to study convergence, initially at a qualitative level.

In this exercise you should try and modify `PJacobi` to observe convergence qualitatively. Call the function you have modified, `MPJacobi`. Its input is the same as the one for `PJacobi`, but its output, besides the solution `x`, is the norm of all approximations from cycle 1 to the last cycle. Then plot the norms against the iteration number and verify that the values converge to a finite value. Try calculating the solution using different starting solutions and comment qualitatively on the character of the convergence.

Use the following non diagonally dominant matrix A and column vector $\mathbf{b}$

$$A = \begin{pmatrix} 3 & 4 & -1 & 1 & 5 \\ -1 & 1 & 0 & 4 & -1 \\ 1 & 2 & 3 & 5 & -1 \\ 1 & 4 & 0 & -1 & 0 \\ -1 & 2 & 0 & 3 & 4 \end{pmatrix} \quad , \quad \mathbf{b} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

for all calculations.

SOLUTION

The parts of `PJacobi` that are modified are shown in the following extract from the code of the functions `MPJacobi`.

```r
.........
.........
.........
# Iterations
  x <- x0

  # New vector containing the norm of x, cycle by cycle
  normx <- norm(as.matrix(x),type="2")

  for (icyc in 1:nmax) {
.........
.........
.........
    # Update values
    x <- x + Deltax

    # Norm of x
    normx <- c(normx,norm(as.matrix(x),type="2"))
.........
.........
.........
```

```
  }
.........
.........
.........
  # Modify output
  #return(x)
  ltmp <- list(x=x,norms=normx)
  return(ltmp)
}
```

The modified function, written in the file *modified_PJacobi.R*, is then loaded in memory with `source` (make sure to have this file, containing `MPJacobi`, available for the command `source`).

```
source("./modified_PJacobi.R")
```

To investigate the convergence to solution, we can apply `MPJacobi` to the linear system, starting from three different values of $\mathbf{x}^{(0)}$, chosen arbitrarily:

$$\mathbf{x}_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad , \quad \mathbf{x}_2 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \quad , \quad \mathbf{x}_3 = \begin{pmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 2 \end{pmatrix}$$

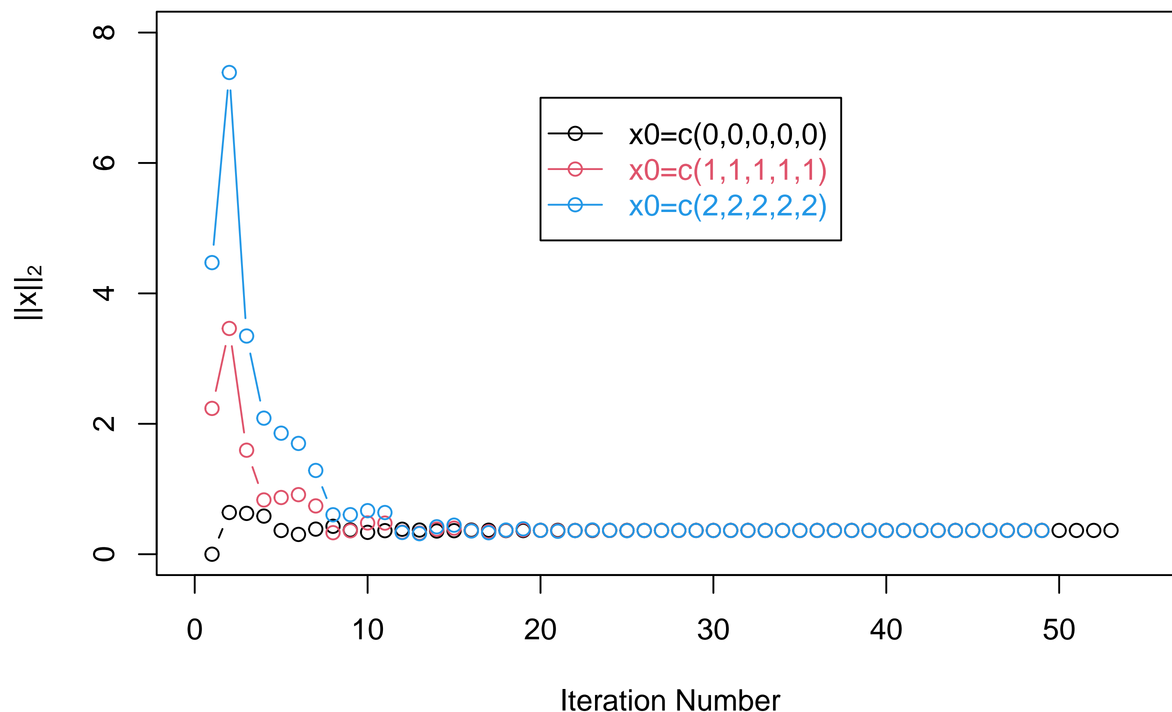
Other values can be chosen and even a systematic set of different starting solutions can be adopted. Here we will be happy just to show how a qualitative study of convergence via the norm of the successive solutions, can be implemented. The creation of all norms needed is obtained in the following passage.

```
# Create the A and b suggested
A <- matrix(c( 3, 4,-1, 1, 5,
              -1, 1, 0, 4,-1,
               1, 2, 3, 5,-1,
               1, 4, 0,-1, 0,
              -1, 2, 0, 3, 4),ncol=5,byrow=TRUE)
b <- matrix(c(1,1,1,1,1),ncol=1)

# Solution and sequence of norms when starting from (0,0,0,0,0)
lP1 <- MPJacobi(A,b,ddominant=FALSE)
#> The matrix of coefficients is not diagonally dominant.
#> Attempting solution anyway...
#>
#> Number of cycles needed to converge: 52
#> Last relative increment: 7.37413211826521e-07
print(lP1$x)
#> [1] -0.09745764  0.30932196 -0.08474564  0.13983061 -0.03389816

# Solution and sequence of norms when starting from (1,1,1,1,1)
lP2 <- MPJacobi(A,b,ddominant=FALSE,x0=c(1,1,1,1,1))
#> The matrix of coefficients is not diagonally dominant.
#> Attempting solution anyway...
#>
#> Number of cycles needed to converge: 48
#> Last relative increment: 6.99668518677754e-07
print(lP2$x)
#> [1] -0.09745850  0.30932323 -0.08474777  0.13982963 -0.03390047
```

```
# Solution and sequence of norms when starting from (2,2,2,2,2)
lP3 <- MPJacobi(A,b,ddominant=FALSE,x0=c(2,2,2,2,2))
#> The matrix of coefficients is not diagonally dominant.
#> Attempting solution anyway...
#>
#> Number of cycles needed to converge: 48
#> Last relative increment: 7.26060911282467e-07
print(lP3$x)
#> [1] -0.09745917  0.30932460 -0.08475005  0.13982839 -0.03390300
```

We can then plot the norms with respect to the iteration number, and see qualitatively how convergence to the solution is obtained by the algorithm. In the plots one has to consider that the range for each series varies and therefore a common range will have to be worked out using the function `range`.

```
# Work out range for the three convergence plots
print(range(lP1$norms,lP2$norms,lP3$norms))
#> [1] 0.000000 7.387583

# The range ylim=c(0,8) will then be chosen
# Also, xlim=c(0,55), as this includes the total
# number of cycles in the three cases (52,48,48)

# First plot (x0=c(0,0,0,0,0))
plot(lP1$norms,type="b",xlim=c(0,55),ylim=c(0,8),
     xlab="Iteration Number",
     ylab=expression(paste("||",x,"||"[2])))

# Second plot (x0=c(1,1,1,1,1))
points(lP2$norms,type="b",col=2)

# Third plot (x0=c(2,2,2,2,2))
points(lP3$norms,type="b",col=4)

# Add legend
legend(x=20,y=7,text.col=c(1,2,4),legend=c("x0=c(0,0,0,0,0)",
                                           "x0=c(1,1,1,1,1)",
                                           "x0=c(2,2,2,2,2)"),
       pch=c(1,1,1),lty=c(1,1,1),col=c(1,2,4))
```



From a quick observation of the plot, one can form the idea that convergence acts in two stages. First, the approximate solution is rapidly pushed in a close region around the final solution (here withinn the first 10-15 cycles). Then the algorithm takes longer to reach the required precision, corresponding to a more accurate convergence to the correct solution.

Several and varied ways to study convergence with norms are possible and in this exercise we have just touched lightly on one way to do that. When the code for the algorithm is available (most of the code is available in R), then it is important to know how to interact with it in order to extract the information needed to study convergence or other features of the algorithm.

2.12 Exercise 17

Carry out the calculations presented in Chapter 4 to introduce ill-conditioning. In the text we have two matrices. The first,

$$A_1 = \begin{pmatrix} 2 & 3 \\ 1 & 1 \end{pmatrix},$$

is well-conditioned, the second,

$$A_2 = \begin{pmatrix} 2 & 1.99 \\ 1 & 1.00 \end{pmatrix},$$

is ill-conditioned. They have been used to solve the two systems,

$$A_1\mathbf{x} = \mathbf{b}_1 \quad , \quad A_2\mathbf{x} = \mathbf{b}_2,$$

where

$$\mathbf{b}_1 = \begin{pmatrix} 7 \\ 3 \end{pmatrix} \quad , \quad \mathbf{b}_2 = \begin{pmatrix} 5.99 \\ 3 \end{pmatrix}.$$

Conditioning of the two systems is manifested when $\mathbf{b}_1$ and $\mathbf{b}_2$ are *perturbed*, i.e. slightly changed into the two vectors,

$$\mathbf{b}_1^{'} = \begin{pmatrix} 6.99 \\ 3.01 \end{pmatrix}, \qquad \mathbf{b}_2^{'} = \begin{pmatrix} 6.00 \\ 2.99 \end{pmatrix}.$$

These can be re-written as

$$\mathbf{b}_1^{'} = \mathbf{b}_1 - \Delta\mathbf{b}_1, \qquad \mathbf{b}_2^{'} = \mathbf{b}_2 - \Delta\mathbf{b}_2,$$

with

$$\Delta\mathbf{b}_1 = \begin{pmatrix} -0.01 \\ 0.01 \end{pmatrix}, \qquad \Delta\mathbf{b}_2 = \begin{pmatrix} 0.01 \\ -0.01 \end{pmatrix}.$$

Using the `norm` function in R, calculate the following quantities,

$$\|A_1\|,\ \|A_2\|,\ \|A_1^{-1}\|,\ \|A_2^{-1}\|,$$

$$\|\mathbf{b}_1\|,\ \|\mathbf{b}_2\|,$$

$$\|\Delta\mathbf{b}_1\|,\ \|\Delta\mathbf{b}_2\|.$$

Calculate also the solutions of the linear systems with the original and perturbed right hand sides, using the `solve` R function. You should verify that the relative error, $\|\Delta\mathbf{x}\|$, satisfies the inequality provided in the text:

$$\frac{1}{\|A^{-1}\|\|A\|}\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} \leq \frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \|A\|\|A^{-1}\|\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}.$$

For all calculations use both the infinity-norm (yielding the numbers in the text) and the Frobenius norm.

SOLUTION

Let us first create in memory all matrices and vectors needed.

```
# Well-conditioned matrix
A1 <- matrix(c(2,1,3,1),ncol=2)

# Its inverse
A1inv <- solve(A1)

# Ill-conditioned matrix
A2 <- matrix(c(2,1,1.99,1),ncol=2)

# Its inverse
A2inv <- solve(A2)

# b1 and b1'
b1 <- matrix(c(7,3),ncol=1)
bb1 <- matrix(c(6.99,3.01),ncol=1)

# b2 and b2'
b2 <- matrix(c(5.99,3.00),ncol=1)
bb2 <- matrix(c(6.00,2.99),ncol=1)
```

Next, the solutions of the four systems involved are calculated with `solve`.

```
# Solution A1*x=b1
x1 <- solve(A1,b1)

# Solution A1*x=bb1
xx1 <- solve(A1,bb1)

# The two solutions are not very different
```

```r
print(x1)
#>      [,1]
#> [1,]    2
#> [2,]    1
print(xx1)
#>      [,1]
#> [1,] 2.04
#> [2,] 0.97

# Solution A2*x=b2
x2 <- solve(A2,b2)

# Solution A2*x=bb2
xx2 <- solve(A2,bb2)

# The two solutions are very different
print(x2)
#>      [,1]
#> [1,]    2
#> [2,]    1
print(xx2)
#>       [,1]
#> [1,]  4.99
#> [2,] -2.00
```

Now we can calculate all the norms involved, using the infinity-norm version. The condition numbers are also given.

```r
# Norm for A1 and A1inv (infinity-norm)
nA1 <- norm(A1,"I")
nA1inv <- norm(A1inv,"I")
print(c(nA1,nA1inv))
#> [1] 5 4

# Condition number for A1
cn1 <- nA1*nA1inv
print(cn1)
#> [1] 20

# Norm for A2 and A2inv (infinity-norm)
nA2 <- norm(A2,"I")
nA2inv <- norm(A2inv,"I")
print(c(nA2,nA2inv))
#> [1]   3.99 300.00

# Condition number for A2
cn2 <- nA2*nA2inv
print(cn2)
#> [1] 1197

# Norm of b1 and Delta b1 (infinity-norm)
nb1 <- norm(b1,"I")
print(nb1)
#> [1] 7
```

```r
nDb1 <- norm(b1-bb1,"I")
print(nDb1)
#> [1] 0.01

# Ratio Delta b1 / b1
rb1 <- nDb1/nb1
print(rb1)
#> [1] 0.001428571

# Norm of b2 and Delta b2 (infinity-norm)
nb2 <- norm(b2,"I")
print(nb2)
#> [1] 5.99
nDb2 <- norm(b2-bb2,"I")
print(nDb2)
#> [1] 0.01

# Ratio Delta b2 / b2
rb2 <- nDb2/nb2
print(rb2)
#> [1] 0.001669449
```

With the above quantities calculated we can now calculate the interval for the relative errors.

```r
# Interval for the relative error of x1
lower1 <- rb1/cn1
upper1 <- cn1*rb1
print(c(lower1,upper1))
#> [1] 7.142857e-05 2.857143e-02

# Interval for the relative error of x2
lower2 <- rb2/cn2
upper2 <- cn2*rb2
print(c(lower2,upper2))
#> [1] 1.394694e-06 1.998331e+00
```

We should verify that the relative errors do fall inside the respective intervals. We will need to calculate the norm of the solutions and their difference.

```r
# Norm of Delta x1, x1 and their ratio (infinity-norm)
nx1 <- norm(x1,"I")
nDx1 <- norm(x1-xx1,"I")
rx1 <- nDx1/nx1
print(c(nDx1,nx1,rx1))
#> [1] 0.04 2.00 0.02

# Norm of Delta x2, x2 and their ratio (infinity-norm)
nx2 <- norm(x2,"I")
nDx2 <- norm(x2-xx2,"I")
rx2 <- nDx2/nx2
print(c(nDx2,nx2,rx2))
#> [1] 3.0 2.0 1.5

# The values are within the estimated range
```

```r
# Dx1/x1
print(c(lower1,rx1,upper1))
#> [1] 7.142857e-05 2.000000e-02 2.857143e-02

# Dx2/x2
print(c(lower2,rx2,upper2))
#> [1] 1.394694e-06 1.500000e+00 1.998331e+00
```

In both cases the relative errors are inside the predicted range. All the calculations involving the norms can be carried out, obviously with different numerical values, using other types of norm. But the predicted interval should still contain the relative error. For the Frobenius norm we have, in the ill-conditioned case:

```r
# Norm of A2, A2inv and condition number
cn2 <- norm(A2,"F")*norm(A2inv,"F")
print(cn2)
#> [1] 996.01

# Norm of Delta b2, b2 and ratio
rb2 <- norm(b2-bb2,"F")/norm(b2,"F")
print(rb2)
#> [1] 0.002110999

# Norm of Delta x2, x2 and ratio
rx2 <- norm(x2-xx2,"F")/norm(x2,"F")
print(rx2)
#> [1] 1.894207

# Interval
lower2 <- rb2/cn2
upper2 <- cn2*rb2
print(c(lower2,rx2,upper2))
#> [1] 2.119456e-06 1.894207e+00 2.102576e+00
```

The relative error is still within the new interval.

2.13 Exercise 18

Most of the time, values of $\|\Delta\mathbf{x}\|/\|\mathbf{x}\|$, where $\mathbf{x}$ is the solution of a linear system, are far from their upper limit. More specifically, when in the linear system,

$$A\mathbf{x} = \mathbf{b}$$

the right hand side is perturbed, the solution is changed so that the relative error acquires values in the following interval,

$$\frac{1}{\|A^{-1}\|\|A\|}\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} \leq \frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \|A\|\|A^{-1}\|\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|},$$

where $\|\Delta\mathbf{b}\|/\|\mathbf{b}\|$ measures the relative change in the right hand side of the linear system. Very often, $\|\Delta\mathbf{x}\|/\|\mathbf{x}\|$ is far from the upper limit of the inequality, $\|A\|\|A^{-1}\|\|\Delta\mathbf{b}\|/\|\mathbf{b}\|$, so that the system behaves as well-conditioned, even if the matrix condition number is high. In fact, both the right hand side of the system and its change, $\Delta\mathbf{b}$, are important quantities when the stability of the solution is contemplated.

It is possible to investigate the range of values of $\|\Delta\mathbf{x}\|/\|\mathbf{x}\|$, once A, $\mathbf{b}$ and $\Delta\mathbf{b}$ are given. In this exercise you are required to prove, using the Frobenius norm, that $\|\Delta\mathbf{x}\|/\|\mathbf{x}\|$ never reaches its upper limit, if A is a 2×2 matrix.

SOLUTION

The inequalities involving the upper limit are,

$$\|\mathbf{x}\| \geq \frac{1}{\|A\|}\|\mathbf{b}\| \qquad \text{and} \qquad \|\Delta\mathbf{x}\| \leq \|A^{-1}\|\|\Delta\mathbf{b}\|,$$

where the norms are all Frobenius norms. For example, given the dimensions of the matrix A and the column vector $\mathbf{x}$,

$$\|A\| = \sqrt{a_{11}^2 + a_{12}^2 + a_{21}^2 + a_{22}^2} \qquad , \qquad \|\mathbf{x}\| = \sqrt{x_1^2 + x_2^2}$$

The maximum value of $\|\Delta\mathbf{x}\|/\|\mathbf{x}\|$ is reached if and when the following two equations hold:

$$\|\mathbf{x}\| = \frac{1}{\|A\|}\|\mathbf{b}\| \quad \text{and} \quad \|\Delta\mathbf{x}\| = \|A^{-1}\|\|\Delta\mathbf{b}\|$$

Let us concentrate on the first of the two, as the demonstration for the second follows a similar argument.

The equation written using the Frobenius norm is:

$$\sqrt{x_1^2 + x_2^2} = \frac{1}{\|A\|}\sqrt{b_1^2 + b_2^2}$$

or, squaring both sides of the equation:

$$x_1^2 + x_2^2 = \frac{1}{\|A\|^2}(b_1^2 + b_2^2)$$

The components of $\mathbf{b}$, b_1, b_2 , are connected to the components of $\mathbf{x}$, x_1, x_2, via the linear system,

$$A\mathbf{x} = \mathbf{b} \quad \Leftrightarrow \quad \left\{ \begin{array}{lcl} a_{11}x_1 + a_{12}x_2 & = & b_1 \\ a_{21}x_1 + a_{22}x_2 & = & b_2 \end{array} \right.$$

Therefore the relation between norms becomes:

$$x_1^2 + x_2^2 = \frac{1}{\|A\|^2}\left((a_{11}x_1 + a_{12}x_2)^2 + (a_{21}x_1 + a_{22}x_2)^2\right)$$

$$\Downarrow$$

$$\left(1 - \frac{1}{\|A\|^2}(a_{11}^2 + a_{21}^2)\right)x_1^2 + \left(1 - \frac{1}{\|A\|^2}(a_{12}^2 + a_{22}^2)\right)x_2^2 - \frac{2}{\|A\|^2}(a_{11}a_{12} + a_{21}a_{22})x_1x_2 = 0$$

The above is one equations with two unknowns. The solution will therefore include a free parameter. Here we will simply fix the second component as the free parameter k:

$$\left(1 - \frac{1}{\|A\|^2}(a_{11}^2 + a_{21}^2)\right)x_1^2 - \frac{2k}{\|A\|^2}(a_{11}a_{12} + a_{21}a_{22})x_1 + k^2\left(1 - \frac{1}{\|A\|^2}(a_{12}^2 + a_{22}^2)\right) = 0$$

The equation in the unknown x_1 is the second-degree equation,

$$ax_1^2 - kbx_1 + k^2c = 0,$$

where,

$$a = 1 - \frac{1}{\|A\|^2}(a_{11}^2 + a_{21}^2) \quad , \quad b = \frac{2k}{\|A\|^2}(a_{11}a_{12} + a_{21}a_{22}) \quad , \quad c = 1 - \frac{1}{\|A\|^2}(a_{12}^2 + a_{22}^2)$$

The equation has one or two solutions if the discriminant, Δ is greater or equal than zero:

$$\Delta = k^2b^2 - 4k^2ac = k^2(b^2 - 4ac) \geq 0$$

k^2 is always non-negative. Therefore let us calculate $b^2 - 4ac$.

$$b^2 - 4ac = \frac{4}{\|A\|^4}(a_{11}a_{12} + a_{21}a_{22})^2 - 4\left(1 - \frac{1}{\|A\|^2}(a_{11}^2 + a_{21}^2)\right)\left(1 - \frac{1}{\|A\|^2}(a_{12}^2 + a_{22}^2)\right)$$

$$\Downarrow$$

$$\begin{aligned}
b^2 - 4ac &= \tfrac{4}{\|A\|^4}(a_{11}^2 a_{12}^2 + 2a_{11}a_{12}a_{21}a_{22} + a_{21}^2 a_{22}^2) \\
&\quad -4\left(1 - \tfrac{a_{11}^2+a_{21}^2+a_{12}^2+a_{22}^2}{\|A\|^2} + \tfrac{a_{11}^2a_{12}^2+a_{11}^2a_{22}^2+a_{21}^2a_{12}^2+a_{21}^2a_{22}^2}{\|A\|^4}\right) \\
&= \tfrac{4}{\|A\|^4}(a_{11}^2 a_{12}^2 + 2a_{11}a_{12}a_{21}a_{22} + a_{21}^2 a_{22}^2) \\
&\quad -4\left(1 - \tfrac{\|A\|^2}{\|A\|^2} + \tfrac{a_{11}^2a_{12}^2+a_{11}^2a_{22}^2+a_{21}^2a_{12}^2+a_{21}^2a_{22}^2}{\|A\|^4}\right) \\
&= \tfrac{4}{\|A\|^4}(a_{11}^2 a_{12}^2 + 2a_{11}a_{12}a_{21}a_{22} + a_{21}^2 a_{22}^2) \\
&\quad -\tfrac{4}{\|A\|^4}(a_{11}^2a_{12}^2+a_{11}^2a_{22}^2+a_{21}^2a_{12}^2+a_{21}^2a_{22}^2)
\end{aligned}$$

$$\Downarrow$$

$$b^2 - 4ac = -\frac{4}{\|A\|^4}(a_{11}^2a_{22}^2 + a_{21}^2a_{12}^2 - 2a_{11}a_{12}a_{21}a_{22})$$

$$\Downarrow$$

$$b^2 - 4ac = -\frac{4}{\|A\|^4}(a_{11}a_{22} + a_{12}a_{21})^2 < 0$$

Therefore, the discriminant is always negative, unless the matrix of coefficients is the trivial null matrix. This means that there are no solutions, $\mathbf{x}$, of the linear system that can cause the relative solution error to be equal to its upper limit.

2.14 Exercise 19

The function `illcond_sample` in `comphy` is a *toy* function that creates examples of linear systems that manifest ill-conditioning in a dramatic way. The highest theoretical limit for the solution relative error is often unreachable (see Exercise 18), but evident effects of ill-conditioning can be seen also without the relative error to reach such a limit. It is possible to find high values of the relative error, given the matrix A, if sampling of $\mathbf{b}$ and $\Delta\mathbf{b}$ is carried out in a statistically-meaningful way, i.e. with a high number of randomly generated values. This is what is achieved by `illcond_sample`.

1. Study the documentation and code of `illcond_sample` and try to understand how the function works.
2. The matrix,

$$A = \begin{pmatrix} 2 & 1.99 \\ 1 & 1.00 \end{pmatrix}$$

   was used in the text to demonstrate the effects of ill-conditioning. With,

$$\mathbf{b} = \begin{pmatrix} 5.99 \\ 3.00 \end{pmatrix}, \; \mathbf{b}' = \begin{pmatrix} 6.00 \\ 2.99 \end{pmatrix}$$

$$\Downarrow$$

$$\Delta\mathbf{b} \equiv \mathbf{b} - \mathbf{b}' = \begin{pmatrix} -0.01 \\ 0.01 \end{pmatrix}$$

   the relative error using the Frobenius norm turns out to be, for the above values of $\mathbf{b}$ and $\Delta\mathbf{b}$,

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} = 1.89421$$

   The upper limit for this case is,

$$\|A^{-1}\|\|A\|\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} = 2.10258$$

   Use `illcond_sample` to try and find a solution relative error greater than the one presented here. You will observe that it is relatively easy to reach a case with a relative solution error higher than the 1.89421 presented here, but this in general corresponds to an upper limit much higher than 2.10258. We can empirically observe that for the specific matrix presented, the ratio between the relative solution error and its upper limit struggles to reach the value $1.89421/2.10258 = 0.90090$ of the case treated here.

Note, some of the numbers here are different from those in the main text because there the norm were infinity-norms, while here they are Frobenius norms.

SOLUTION

The function takes other input besides the matrix. One possibility to increase sampling is to generate more random **b**'s, where the components of each random **b** are fished from the uniform distribution. By default the number of random generations is `ncyc=100000`. We can increase `ncyc` to 500000 and perhaps re-execute the function a few times with a different starting seed.

```
# Matrix
A <- matrix(c(2,1,1.99,1),ncol=2)

# First attempt (iseed=1786)
ltmp1 <- illcond_sample(A,ncyc=500000,iseed=1786)
#> Relative error: 413.685246696998 < 580.149547812422: upper limit.
#> Ratio: 0.713066567502959

# Second attempt (iseed=1787)
ltmp2 <- illcond_sample(A,ncyc=500000,iseed=1787)
#> Relative error: 648.925626820576 < 747.84595698469: upper limit.
#> Ratio: 0.867726328877995
```

Both scenarions don't reach the 0.90090 ratio previously found, but have relative errors much larger than the related 1.89421 value. This is reflected in a change in solution, $\Delta\mathbf{x}$, more dramatic than the change,

$$\Delta\mathbf{x} = \begin{pmatrix} -2.99 \\ 3.00 \end{pmatrix}$$

manifested in the scenario previously found.

```
# Try just the first smpled value
b <- ltmp1$b
Db <- ltmp1$Db
b2 <- b-Db
x <- solve(A,b)
print(x)
#>            [,1]
#> [1,] 0.67441924
#> [2,] 0.09327894
x2 <- solve(A,b2)
print(x2)
#>           [,1]
#> [1,]  199.3330
#> [2,] -199.5647
Dx <- x-x2
print(Dx)
#>           [,1]
#> [1,] -198.6586
#> [2,]  199.6580
```