Operations Research

Laboratory Session 3: Solving linear models with R

by Josu Ceberio and Ana Zelaia

The aim of this laboratory session is to implement some functions in R to solve linear models.

Solving linear models

The graphical solution of linear models shows in a very intuitive way that the optimal solution of a linear model is an extreme point of the convex set of feasible solutions. If the problem has multiple optimal solutions, at least one of them is an extreme point of the set. However, the graphical solution cannot be used to solve linear models with more than three variables and linear models normally have a large number of variables.

In the process of developing an algebraic method to solve linear models, we analyzed two theorems that demonstrate that, if a linear model is feasible, it is possible to solve it in an algebraic way. They both consider a linear model in maximization standard form ($\mathbf{b} \geq \mathbf{0}$):

$$\begin{aligned} \max \quad z = \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \\ & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

and state the following:

- An optimal solution to a linear model is an extreme point of the feasible region.
- Every extreme point corresponds to a basic feasible solution, and conversely, every basic feasible solution corresponds to an extreme point.

Linear models have a finite number of basic solutions at most and, therefore, it is possible to compute all of them, select only the feasible ones and check the objective function to see which is the optimal. However, it is clear that this is not an efficient method. Moreover, the analysis of all the basic feasible solutions does not allow to detect unbounded problems.

In 1947 George Dantzig, the "Father of Linear Programming" developed the simplex algorithm to solve linear problems. It has been declared to be one of the Top 10 algorithms of the 20th century, because of its great influence on the development and practice of science and engineering. The simplex algorithm computes basic feasible solutions and finds the optimal in a very efficient way. Moreover, it handles unbounded problems appropriately.

In this 3rd lab session, we will implement R functions to solve linear models as if the simplex algorithm did not exist: given a linear model written in maximization standard form, we will compute all the basic solutions. Checking their feasibility and choosing the optimal one will be the tasks to work in the 4th lab session. In both lab sessions, we will not consider unbounded problems.

Some interesting R built-in functions

Given the $Ax = b$ system of linear equations, the R functions below will be useful, during the implementation process, to verify whether a set of columns of the matrix form a basis, to calculate a basic solution or to find linear combinations of the columns in the matrix.

Function `det(B)`. A system of linear equations may be inconsistent. To check if some columns in matrix **A** form a basis, function `det` can be used. If the system is consistent, a solution can be computed.

```
det(A[,c(1,2)])!=0
```

Function `solve(B,b)`. This function can be used to solve systems of linear equations that are consistent.

```
solve(A[,c(1,2)], b)
```

Function `combn(x,m)`. Given a system of linear equations, to compute all the basic solutions, we will have to analyze all the combinations of columns of matrix **A** that may form a basis. Function `combn(x,m)` generates all combinations of the elements of vector x taking m at a time and returns a matrix with a column for each combination generated.

```
x <- c(1:4)
x
# [1] 1 2 3 4
combn(x, 2)
#     [,1] [,2] [,3] [,4] [,5] [,6]
# [1,]    1    1    1    2    2    3
# [2,]    2    3    4    3    4    4
combn(x, 2)[,1]
# [1] 1 2
A[,combn(x, 2)[,1]]
solve(A[,combn(x, 2)[,1]], b)
```

Exercises

Exercise 1. The `basic.solution` function.

Given a matrix **A**, a vector $\mathbf{b}(\mathbf{b} \geq \mathbf{0})$ and a vector of column-indices for **A**, define a function that extracts those columns from **A** and checks if they form a basis. If they do, the function returns the corresponding basic solution: a vector with the values for all the variables of the system. If they don't, the function returns a vector with the values for all the variables of the system set to -1.

To check the correctness of the functions implemented, let us consider the following linear model:

$$\begin{aligned} \max\ & z = 3x_1 + 4x_2 + 5x_3 + 6x_4 \\ \text{subject to} & \\ & 2x_1 + x_2 + x_3 + 8x_4 = 6 \\ & x_1 + x_2 + 2x_3 + x_4 = 4 \\ & x_1, x_2, x_3, x_4 \geq 0 \end{aligned}$$

```
A <- matrix(c(2, 1, 1, 8, 1, 1, 2, 1), nrow=2, byrow=TRUE)
b <- c(6, 4)
c <- c(3, 4, 5, 6)
```

```
rm(list=ls())
basic.solution <- function(A, b, column.ind.vector){
# Implement here.
  if(!det(A[,column.ind.vector])){
```

```
    return(A[,-column.ind.vector])
  }
  allVariables <- rep(0,ncol(A))


  basicSolutions<-solve(A[,column.ind.vector],b)
  allVariables[column.ind.vector]<-basicSolutions
  return(allVariables)
}

A <- matrix(c(2, 1, 1, 8, 1, 1, 2, 1), nrow=2, byrow=TRUE)
b <- c(6, 4)

basic.solution(A, b, c(1,4))
# x1=0.0000000, x2=0.0000000, x3=1.7333333, x4=0.5333333
```

Exercise 2. The `all.basic_solutions` functions: with `for` and `apply` loops (two versions).

Given a matrix $\mathbf{A}$ and a vector $\mathbf{b}$ ($\mathbf{b} \geq \mathbf{0}$), compute all the basic feasible solutions. Propose two different implementations: using the `for` loop(s) and with the `apply` functions.

2.1. Using `for` loops. Try returning the solutions in a list. To test your implementation, you can use the same linear model as in Exercise 1.

```
all.basic_solutions_for <- function(A, b){
# Implement here.
  solutions <- list()
  bases <- combn(ncol(A),nrow(A))
  for (j in 1:ncol(bases)) {
    solutions[[length(solutions)+1]]<-basic.solution(A,b,bases[,j])
  }
  return(solutions)
}
# This function returns a list.
all.basic_solutions_for(A,b)
## [[1]]
## [1] 2 2 0 0
##
## [[2]]
## [1] 2.6666667 0.0000000 0.6666667 0.0000000
##
## [[3]]
## [1] 4.3333333  0.0000000  0.0000000 -0.3333333
##
## [[4]]
## [1] 0 8 -2 0
##
## [[5]]
## [1] 0.0000000 3.7142857 0.0000000 0.2857143
##
## [[6]]
## [1] 0.0000000 0.0000000 1.7333333 0.5333333
```

2.2. Using `apply` loops. Try returning the solutions in a matrix. To test your implementation, you can use the same linear model as in Exercise 1.

```r
all.basic_solutions_apply <- function(A,b){
  # Implement here.
  bases <- combn(ncol(A),nrow(A))
  solution<-apply(bases,2, function(base) basic.solution(A,b,base))
  return(solution)
}

# This function returns a matrix. Basic solutions are shown in columns
all.basic_solutions_apply(A,b)

##     [,1]      [,2]       [,3] [,4]      [,5]      [,6]
##[1,]    2 2.6666667  4.3333333    0 0.0000000 0.0000000
##[2,]    2 0.0000000  0.0000000    8 3.7142857 0.0000000
##[3,]    0 0.6666667  0.0000000   -2 0.0000000 1.7333333
##[4,]    0 0.0000000 -0.3333333    0 0.2857143 0.5333333
```

Exercise 3. The `basic.feasible.solutions` functions: with `for` and `apply` functions (two versions).

Adapt the implementations in exercise 2 to return only the basic solutions that are feasible.

3.1. Adaptation of the functions using `for` loops. Try returning the solutions in a list. To test your implementation, you can use the same linear model as in Exercise 1.

```r
basic.feasible.solutions_for <- function(A, b){
# Implement here.
  solutions <- list()
  bases <- combn(ncol(A),nrow(A))
  for (j in 1:ncol(bases)) {
    # solution<- list()
    # for (i in 1:length(bases[,j])){
    #   print(A[,bases[,j]]*)
    #
    solution<-basic.solution(A,b,bases[,j])
    if(all(A %*% solution == b)){
      if(any(solution<0)){
        next
      }
      solutions[[length(solutions)+1]]<-solution
    }


  }
  return(solutions)

}
# This function returns a list.
basic.feasible.solutions_for(A,b)
## [[1]]
## [1] 2 2 0 0
##
## [[2]]
## [1] 2.6666667 0.0000000 0.6666667 0.0000000
##
## [[3]]
```

```
## [1] 0.0000000 3.7142857 0.0000000 0.2857143
##
## [[4]]
## [1] 0.0000000 0.0000000 1.7333333 0.5333333
#
# Interpretation: There are four basic feasible solutions.
# x1=2, x2=2, x3=0, x4=0
# x1=2.6666667, x2=0.0000000, x3=0.6666667, x4=0.0000000
# x1=0.0000000, x2=3.7142857, x3=0.0000000, x4=0.2857143
# x1=0.0000000, x2=0.0000000, x3=1.7333333, x4=0.5333333
```

3.2. Adaptation of the functions using `apply` loops. Try returning the solutions in a matrix. To test your implementation, you can use the same linear model as in Exercise 1.

```
basic.feasible.solutions_apply <- function(A,b){
# Implement here (5-6 lines)
  bases <- combn(ncol(A),nrow(A))
  solutions<-apply(bases,2, function(base) basic.solution(A,b,base))

  f_s_with_null<-apply(solutions, 2, feasible <- function(solution, A, b) {
    if(all(A %*% solution == b)){
      if(!any(solution<0)){
        return(solution)
      }
    }
  }, A, b)
  return(f_s_with_null[!sapply(f_s_with_null, is.null)])
}
# This function returns a matrix. Basic solutions are shown in columns
basic.feasible.solutions_apply(A,b)
##      [,1]      [,2]      [,3]      [,4]
## [1,]    2 2.6666667 0.0000000 0.0000000
## [2,]    2 0.0000000 3.7142857 0.0000000
## [3,]    0 0.6666667 0.0000000 1.7333333
## [4,]    0 0.0000000 0.2857143 0.5333333

# Interpretation: There are four basic feasible solutions.
# x1=2, x2=2, x3=0, x4=0
# x1=2.6666667, x2=0.0000000, x3=0.6666667, x4=0.0000000
# x1=0.0000000, x2=3.7142857, x3=0.0000000, x4=0.2857143
# x1=0.0000000, x2=0.0000000, x3=1.7333333, x4=0.5333333
```

Linear problems for testing

Solve the following linear models using the functions defined, and check the solution.

System of linear equations from exercise 6 of the unit "Linear Algebra".

Let us consider the following system of linear equations:

$$\begin{aligned} 2x_1 + 3x_2 - x_3 &= 1 \\ x_1 + x_2 + x_4 &= 3 \\ -x_1 + 2x_2 + x_5 &= 5 \end{aligned}$$

Definition of matrix A and vector b:

```
A <- matrix(c(2, 3, -1, 0, 0, 1, 1, 0, 1, 0, -1, 2, 0, 0, 1), nrow=3, byrow=TRUE)
b <- c(1, 3, 5)
```

1. Calculate the basic solutions one by one using the function `basic.solution`.

```
# Implement here (10 lines)
```

2. Calculate all the basic solutions using the function `all.basic_solutions`.

2.1. Use the `for` version calling to the function `all.basic_solutions_for`:

```
# Implement here (1 line)
all.basic_solutions_for(A,b)
```

2.2. Use the `apply` version calling to the function `all.basic_solutions_apply`:

```
# Implement here (1 line)
all.basic_solutions_apply(A,b)
```

3. Return exclusively the basic solutions that are feasible. To that end, use the function `basic.feasible.solutions`.

3.1. Use the `for` version calling to the function `basic.feasible.solutions_for`:

```
# Implement here (1 line)
basic.feasible.solutions_for(A,b)
```

3.2. Use the `apply` version calling to the function `basic.feasible.solutions_applyr`:

```
# Implement here (1 line)
basic.feasible.solutions_apply(A,b)
```

System of linear equations from exercise 2 of the list of exercises of the unit "The simplex method".

Let us consider the following system of linear equations:

$$\begin{array}{rcrcrcr} -x_1 + & x_2 + x_3 & & & & = & 4 \\ 2x_1 + 5x_2 & & + x_4 & & & = & 20 \\ 2x_1 - & x_2 & & & + x_5 & = & 2 \end{array}$$

Definition of matrix A and vector b:

```
A <- matrix(c(-1, 1, 1, 0, 0, 2, 5, 0, 1, 0, 2, -1, 0, 0, 1), nrow=3, byrow=TRUE)
b <- c(4, 20, 2)
```

1. Calculate the basic solutions one by one using the function `basic.solution`..

```
# Implement here (10 lines)
```

2. Calculate all the basic solutions using the function `all.basic_solutions`.

2.1. Use the `for` version calling to the function `all.basic_solutions_for`:

```
# Implement here (1 line)
all.basic_solutions_for(A,b)
```

2.2. Use the `apply` version calling to the function `all.basic_solutions_apply`:

```
# Implement here (1 line)
all.basic_solutions_apply(A,b)
```

3. Return exclusively the basic solutions that are feasible. To that end, use the function `basic.feasible.solutions`.

3.1. Use the `for` version calling to the function `basic.feasible.solutions_for`:

```
# Implement here (1 line)
basic.feasible.solutions_for(A,b)
```

3.2. Use the `apply` version calling to the function `basic.feasible.solutions_apply`:

```
# Implement here (1 line)
basic.feasible.solutions_apply(A,b)
```