


# Tarea 1 - Simulación Estocástica

Fabián Ramírez

Una versión ejecutable de los códigos los puede encontrar haciendo click en la siguiente [GitHub](#) 

## Problema 1

Escriba una rutina para resolver el sistema de ecuaciones lineales  $Ax = b$  con:

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 3 & 1 \\ 1 & -1 & -1 \end{pmatrix}, \quad b = \begin{pmatrix} 4 \\ 9 \\ -1 \end{pmatrix}$$

### Solución:

Procederemos a mostrar dos rutinas para resolver el sistema.

### Usando la descomposición LU

En primer lugar creamos una función que realice la descomposición de una matriz.

```
LU = function( A ){
  if ( dim(A)[1]!=dim(A)[2] ){
    stop( 'La matriz no es cuadrada' )
  }
  if ( !is.numeric( A ) ){
    stop( 'Argumentos Erroneos' )
  }
  n = dim(A)[1]
  L = matrix(0,n,n)
  U = diag(n)
  for ( i in 1:n ) {
    aux1 = i - 1
    for (j in 1:n){
      L[j,i] = A[j,i]
      if ( aux1 > 0 ) {
        for ( k in 1:aux1 ) {
          L[j,i] = L[j,i] -
-L[j,k] * U[k,i]
        }
      }
    }
  }
}
```

```
    aux2 = i+1
    if ( aux2 <= n ) {
      for ( j in aux2:n ) {
        U[i,j] = A[i,j]
        if ( aux1 > 0 ) {
          for ( k in 1:aux1 ) {
            U[i,j] = U[i,j] -
-L[i,k] * U[k,j]
          }
        }
        U[i,j] = U[i,j] / L[i,i]
      }
    }
  }
  return(list(L=L,U=U))
}
```

De esta forma obtenemos el sistema:

$$Ax = b \iff LUx = b$$

Donde  $L$  es triangular inferior y  $U$  es triangular superior. Ahora creamos dos rutinas, una para resolver un sistema triangular inferior y uno para resolver un sistema triangular superior.

```
rti = function (L,b){
  n = dim(L)[1]
  z = c()
  for (i in 1:n){
    sol = b[i]
    if ( length(z) > 0 ) {
      for ( k in 1:length(z) ) {
        sol = sol - L[i,k]*z[k]
      }
    }
    sol = sol/L[i,i]
    z = c(z,sol)
  }
  return(matrix(z,n,1))
}
```

```
rts = function (U,b){
  n = dim(U)[1]
  z = c()
  for (i in n:1){
    sol = b[i]
    if ( length(z) > 0 ) {
      for ( k in 1:length(z) ) {
        sol = sol -
-U[i,n-k+1]*z[k]
      }
    }
    sol = sol/U[i,i]
    z = c(sol,z)
  }
  return(matrix(z,n,1))
}
```

Finalmente para resolver el sistema de lineal debemos seguir los siguientes pasos:

```
resolver = function (A,b){
  L =LU(A)$L
  U =LU(A)$U
  z=rti(L,b)
  x = rts(U,z)
  return(x)
}
```

Y la solución del sistema viene dado por:

```
A = matrix(c(1,2,1,1,3,-1,1,1,-1),3,3)
b = matrix(c(4,9,-1),3,1)
resolver(A,b)
```

$$\begin{pmatrix} 1.50 \\ 1.75 \\ 0.75 \end{pmatrix}$$

## Usando Refinamiento Iterativo

El algoritmo para utilizar refinamiento iterativo es:

```
refinamiento_iterativo = function (kmax,tol,A,b,x_0){  
  k=0  
  while (k<kmax){  
    r = b - A%%x_0  
    delta = resolver(A,r)  
    x_0 = x_0 + delta  
    if (max(delta)<= tol*max(x_0)){  
      return(x_0)  
    }  
    k=k+1  
  }  
  stop(paste('El algoritmo no converge despues de',kmax,'iteraciones'))  
}
```

Luego la solución del sistema por método iterativo viene dado por:

```
refinamiento_iterativo(10,0.001,A,b,rnorm(3))
```

$$\begin{pmatrix} 1.50 \\ 1.75 \\ 0.75 \end{pmatrix}$$

**Observación:** Note que  $x_0$  lo tomo como tres números generados al azar por una ley normal 0-1

## Problema 2

Implementar el operador Sweep usando su lenguaje de programación favorito.

### Solución:

Diseñamos la siguiente función `sweep`.

```
sweep = function(A, k) {
  B = matrix(0, dim(A)[1], dim(A)[1])
  B[k,k] = 1/A[k,k]
  for (i in 1:n){
    if (i != k){
      B[i,k] = -A[i,k]/A[k,k]
    }
  }
  for (j in 1:n){
    if (j != k){
      B[k,j] = A[k,j]/A[k,k]
    }
  }
  for (i in 1:n){
    for (j in 1:n){
      if (i != k & j != k){
        B[i,j] = A[i,j] - (A[i,k]*A[k,j])/A[k,k]
      }
    }
  }
  return(B)
}
```

Note que esta función cumple las tres propiedades de la función Sweep vista en clases. Probaremos el algoritmo para la matriz  $A$  del Problema 1.

```
sweep(sweep(A, 2), 2)
```

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

```
sweep(sweep(A, 1), 2)
sweep(sweep(A, 2), 1)
```

$$\begin{pmatrix} 3 & -1 & 2 \\ -2 & 1 & -1 \\ -5 & 2 & -4 \end{pmatrix} \text{ y } \begin{pmatrix} 3 & -1 & 2 \\ -2 & 1 & -1 \\ -5 & 2 & -4 \end{pmatrix} \text{ Ahora nos gustaría hacer una función que permita obtener la inversa de}$$

una matriz a partir del operador Sweep.

```
inv_sweep = function (A) {  
  n = dim(A)[1]  
  B = A  
  for (i in 1:n){  
    B = sweep(B,i)  
  }  
  return(B)  
}
```

```
inv_sweep(A)
```

$$\begin{pmatrix} 0.50 & 0.0 & 0.50 \\ -0.75 & 0.5 & -0.25 \\ 1.25 & -0.5 & -0.25 \end{pmatrix}$$

Finalmente para comparar veamos la matriz inversa de  $A$  mediante la función solve del R:

```
solve(A)
```

$$\begin{pmatrix} 0.50 & 0.0 & 0.50 \\ -0.75 & 0.5 & -0.25 \\ 1.25 & -0.5 & -0.25 \end{pmatrix}$$

### Problema 3

Pruebe su rutina para obtener la inversa de una matriz usando el operador Sweep con la siguiente matriz:

$$B = \begin{pmatrix} 30 & 16 & 46 \\ 16 & 10 & 26 \\ 46 & 26 & 72 \end{pmatrix}$$

#### Solución:

```
B = matrix(c(30,16,46,16,10,26,46,26,72),3,3)
```

Luego notemos que:

```
inv_sweep(B)
```

$$\begin{pmatrix} 1.8765e+14 & 1.8765e+14 & -1.8765e+14 \\ 1.8765e+14 & 1.8765e+14 & -1.8765e+14 \\ -1.8765e+14 & -1.8765e+14 & 1.8765e+14 \end{pmatrix}$$

Notemos que según el método Sweep para invertir matrices tenemos que la matriz inversa de  $B$  es prácticamente una matriz cuyas entradas son '*infinito*', lo que nos podría advertir que existe un problema con la matriz  $B$ . Por tanto podríamos inferir que la matriz  $B$  tiene alguno de los siguientes problemas.

1. Uno o más valores propios de la matriz  $B$  es 0.
2. Uno o más valores propios de la matriz  $B$  son muy cercanos a 0, por tanto se obtienen errores numéricos muy grandes dado el mal condicionamiento.

Notemos que si hacemos la descomposición LU de  $B$  tenemos que:

```
LU(B)
```

$$\$L = \begin{pmatrix} 30 & 0.000000 & 0.000000e+00 \\ 16 & 1.466667 & 0.000000e+00 \\ 46 & 1.466667 & 1.776357e-15 \end{pmatrix}$$

$$\$U = \begin{pmatrix} 1 & 0.533333 & 1.533333 \\ 0 & 1.000000 & 1.000000 \\ 0 & 0.000000 & 1.000000 \end{pmatrix}$$

De donde podemos obtener que el determinante de  $B$  es  $30 \cdot 1,466667 \cdot 1,776357e-15 \cdot 1 \cdot 1 \cdot 1 \approx 0$  por lo tanto el determinante de  $B$  es aproximadamente 0. Por lo tanto invertir una matriz mal condicionada como lo es  $B$  tiene mucho error numérico, lo cual explica los valores que se obtienen al tratar de invertir  $B$ .

**Observación:** Al calcular los valores propios de  $B$  a mano nos podemos dar cuenta que uno de los valores propios es 0, y por tanto no es invertible, pero no siempre es simple calcular explícitamente los valores propios sin un método iterativo, por tanto decidí argumentar este problema únicamente desde el punto de vista numérico.

## Problema 4

Verifique que  $B$  es matriz semi definida positiva usando la factorización Cholesky. ¿En cuál etapa del algoritmo el procedimiento falla?

### Solución:

En primer lugar plantearemos un algoritmo para obtener la descomposición Cholesky de una matriz.

```
choleski = function(A){  
  n = dim(A)[1]  
  T = matrix(0,n,n)  
  T[1,1] = sqrt(A[1,1])  
  for (j in 2:n){  
    T[1,j] = A[1,j]/T[1,1]  
  }  
  for (i in 2:n){  
    res = A[i,i]  
    for (k in 1:(i-1)){  
      res = res - (T[k,i])^2  
    }  
    T[i,i] = sqrt(res)  
    res=0  
    if ((i+1)<=n){  
      for (j in (i+1):n){  
        res = A[i,j]  
        for (k in 1:(i-1)){  
          res = res - T[k,i]*T[k,j]  
        }  
        T[i,j] = res/T[i,i]  
      }  
    }  
  }  
  return(T)  
}
```

Ahora procedemos a aplicar nuestro algoritmo a la matriz  $B$  (que ya sabemos por **Pregunta 3** que es una matriz mal condicionada con algún valor propio muy cercano a 0).

```
choleski(B)
```

```
Warning message in sqrt(res):  
"NaNs produced"
```

$$\begin{pmatrix} 5.477226 & 2.921187 & 8.398413 \\ 0.000000 & 1.211060 & 1.211060 \\ 0.000000 & 0.000000 & \text{NaN} \end{pmatrix}$$

Ahora bien realizaremos un seguimiento del código para saber cuando se genera el error.

### Paso 1

```
n = dim(B)[1]
T = matrix(0,n,n)
T[1,1] = sqrt(A[1,1])
```

No hay error.

### Paso 2

Realizamos el ciclo `for` para `i=2`

```
for (j in 2:n){
  T[1,j] = A[1,j]/T[1,1]
}
```

```
i=2
```

```
res = A[i,i]
for (k in 1:(i-1)){
  res = res - (T[k,i])^2
}
T[i,i] = sqrt(res)
res=0
if ((i+1)<=n){
  for (j in (i+1):n){
    res = A[i,j]
    for (k in 1:(i-1)){
      res = res - T[k,i]*T[k,j]
    }
    T[i,j] = res/T[i,i]
  }
}
```

No hay error.



### Paso 3

Realizamos el ciclo `for` para `i=3`

```
i=3
```

```
res = A[i,i]
for (k in 1:(i-1)){
  res = res - (T[k,i])^2
}
T[i,i] = sqrt(res)
res=0
if ((i+1)<=n){
  for (j in (i+1):n){
    res = A[i,j]
    for (k in 1:(i-1)){
      res = res - T[k,i]*T[k,j]
    }
    T[i,j] = res/T[i,i]
  }
}
```

Warning message in `sqrt(res)`:  
"NaNs produced"

Hay error, por tanto veremos mas a detalle este ciclo `for`.

```
for (k in 1:(i-1)){
  res = res - (T[k,i])^2
}
```

No hay error.

### Paso 4

```
T[i,i] = sqrt(res)
```

Warning message in `sqrt(res)`:  
"NaNs produced"

Hay error. Por lo tanto debe ser un problema con `sqrt(res)`, por tanto veamos el valor de `res`

```
res
```

-1

Por tanto el error viene de que estoy calculando la raíz cuadrada de un número negativo, por tanto no se puede aplicar la factorización Cholesky. Note que  $B$  es simétrica y semi-definida positiva, por tanto el único argumento posible para que la matriz no admita una factorización Cholesky es que  $B$  no sea estrictamente definida positiva, es decir que tiene un valor propio 0.

## Comentarios

1. La descomposición LU se puede realizar para cualquier matriz cuadrada, pero si la matriz tiene rango completo entonces la descomposición LU es única. Por tanto la descomposición LU es muy útil para saber si una matriz esta mal condicionada o no puesto que nos entrega un método practico para calcular el determinante de una matriz.
2. En el **Problema 1** en la sección del método **Refinamiento Iterativo** realizo la siguiente linea de código:

```
r = b - A%*%x_0
```

en estricto rigor yo debería programar como hacer una multiplicación matricial. Por tanto dejo aquí una rutina para multiplicar matrices.

```
multiplicar_matrices = function (A,B){  
  if (dim(A)[2]!=dim(B)[1]){  
    stop('No coinciden las dimensiones')  
  }  
  m = dim(A)[1]  
  n = dim(A)[2]  
  p = dim(B)[2]  
  C = matrix(,m,p)  
  for (i in 1:m){  
    for (j in 1:p){  
      C[i,j]=0  
      for (k in 1:n){  
        C[i,j] = C[i,j]+ A[i,k]*B[k,j]  
      }  
    }  
  }  
  return(C)  
}
```

3. En **Problema 1** en la sección de **Descomposición LU** se tiene que la descomposición tiene la característica que los valores de la diagonal de  $L$  son todos 1 mientras que los de  $U$  son distintos de uno. Generaremos otra versión de la descomposición LU para que los valores distintos de 1 se encuentren en  $U$  tal como lo muestran las diapositivas vistas en clases.

```

LU_ver2 = function( A ){
  if ( dim(A)[1]!=dim(A)[2] ){
    stop( 'La matriz no es cuadrada' )
  }
  if ( !is.numeric( A ) ){
    stop( 'Argumentos Erroneos' )
  }
  n = dim(A)[1]
  L = matrix(0,n,n)
  U = diag(n)
  diag(L) = rep(1,n)
  for (i in 1:n) {
    aux1 = i - 1
    for (j in 1:n) {
      U[i,j] = A[i,j]
      if ( aux1 > 0 ) {
        for ( k in 1:aux1 ) {
          U[i,j] = U[i,j] - L[i,k] * U[k,j]
        }
      }
    }
    aux2 = i + 1
    if ( aux2 <= n ) {
      for ( j in aux2:n ) {
        L[j,i] = A[j,i]
        if ( aux1 > 0 ) {
          for ( k in 1:aux1 ) {
            L[j,i] = L[j,i] - L[j,k] * U[k,i]
          }
        }
        L[j,i] = L[j,i] / U[i,i]
      }
    }
  }
  return(list(L=L,U=U))
}

```

```
LU_ver2(A)
```

$$\$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -2 & 1 \end{pmatrix} \quad \$U = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & -4 \end{pmatrix}$$