Programming Assignment 2: Lexical Scoping

Caching The Inverse of a Matrix

M. G. Ahsan | August 20, 2015

Key Words: Control Structure, Dynamic Scoping, Global Environment, Inverse Matrix, Lexical scoping, Parent Environment, Parent Frame, R, Square Matrix, Singular Matrix, Scoping Rules, Static Scoping

Introduction:

This programming exercise constructs an R function system that is able to cache potentially time consuming computations. Matrix inversion is usually a costly computation given that inverse of a matrix needs to be used multiple times when solving for a linear system or even when using it in a loop. Taking advantage of the scoping rules of the R language, this programming exercise creates a preserve state inside of an R object, which computes the inverse of a matrix and can cache it in the memory for future use.

This short note begins with introducing the mathematical properties of invertible matrices as well as how an inverse matrix is computed in R. It then presents the core mechanism of the R function-pair that are constructed. The body of the note ends by discussing the implication of scoping rules in this sort of a project.

The document also includes two detailed appendices that demonstrate step by step evaluation of the functions (**Appendix A**) and reports some test results (**Appendix B**)using different variations of square matrices.

Inverse Matrix

If A is a square $(n \times n)$ and non-singular $(\det(A) \neq 0)$ matrix then an inverse matrix of A is denoted by A^{-1} such that $AA^{-1} = A^{-1}A = I$; where I is the identity matrix.

As only non-zero real numbers can have an inverse, in matrix algebra only non-singular square matrices have an inverse. Therefore, for A to be invertible a matrix A^{-1} must exist.

Consider the following 2 by 2 matrix:

$$X = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$X^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

For the above square matrix X, an inverse X^{-1} exists, only if $ad - bc \neq 0$. Also note that $(ad - bc)$ is the determinant of X.

One simple but necessary point to consider for this particular assignment is that the inverse of an invertible numeric matrix is as simple as dividing the identity matrix with the matrix itself, i.e., $X^{-1}X = I$ or, $X^{-1} = \frac{I}{X}$.

Inverse Matrix in R

There are several methods that can be used to calculate inverse of a matrix in R. For this assignment though the scope is confined to the `solve` function.

The `solve` function in R is typically used to solve a linear system. Following are the default arguments used:

```
solve(a, b, ...)

a : A square numeric or complex matrix containing the coefficients of the linear system.
b : A numeric or complex vector or matrix giving the right-hand side of the linear system.
... : Further arguments passed to or from other methods
```

This function basically solves the linear equation $a \times x = b$ for x; where b can be a matrix or a vector. The argument b is also defined such that if it is not included as an argument in the function, it is assumed to be an identity matrix by R. This characteristic of b allows the calculation on an inverse of a by using the `solve(a)` function.

When b is missing in the function the generic equation becomes $a \times x = I$ or, $x = \frac{I}{a} = a^{-1}$. Therefore, given that a is an invertible square matrix, x is simply the inverse of that matrix.

The Cache Matrix

The `makeCacheMatrix` function creates a special "matrix' object that can cache its own inverse matrix. The function takes a matrix input 'x' as an argument and then creates an environment that can both 'produce and cache the inverse of that matrix' and 'retrieve it from memory' if it is already cached.

The list of functions created in the Global Environment' provides the following functionality:

1. Set the value of the matrix - `set`
2. Get the value of the matrix - `get`
3. Set the value of the inverse matrix - `setinv`
4. Get the value of the inverse matrix - `getinv`

```r
makeCacheMatrix <- function(x = matrix()) {
        IM <- NULL
        set <- function(y) {
                x <<- y
                IM <<- NULL
        }
        get <- function() x
        setinv <- function(inv) IM <<- inv
        getinv <- function() IM
        list(set = set, get = get, setinv = setinv, getinv = getinv)
}
```

Solving for the Inverse Matrix

The `cacheSolve` function solves for the inverse of the 'special matrix' returned by `makeCacheMatrix` function above. Using an if-else control structure, it first checks to see if the inverse matrix has already been produced. If so, then, it skips the computation and gets the inverse matrix from the cache via the `getinv` function. Otherwise, it computes the inverse of the original matrix and sets the output in the cache via the `setinv` function.

```
cacheSolve <- function(x, ...) {
        IM <- x$getinv()
        if(!is.null(IM)) {
                message("getting cached data (inverse matrix)")
                return (IM)
        }
        data <- x$get()
        IM <- solve(data, ...)
        x$setinv(IM)
        IM
}
```

Appendix A discusses how these system of functions work step by step through different environments abiding R's scoping rules. Test outputs using matrices of different dimensions have been presented in **Appendix B**.

Implication of Scoping Rules

Scoping rules followed by R are crucial to the design of the mechanism discussed so far. Scoping rules determine how a value is bound to a free variable (not a formal argument) in a function. The scoping rules followed by R are known as 'lexical scoping' which is also known as 'static scoping'.

Under lexical scoping, values for free variables are primarily searched in the environment in which the function was defined. Such an environment is is typically the 'global environment' or the user's workspace.

An exact opposite to lexical scoping is dynamic scoping. Under dynamic scoping, values for free variables are primarily searched in the environment from which the function was called. In R such an environment is known as the 'parent frame'. Example of a parent frame can be an R-package.

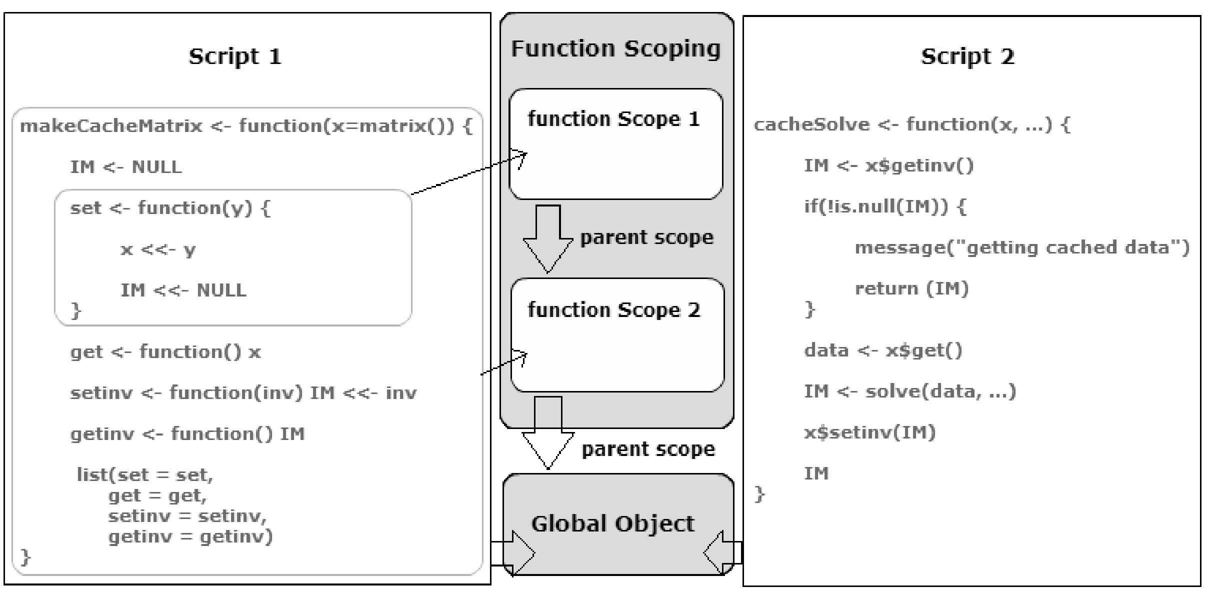

Figure: Lexical Scoping Rules in R

Coming back to lexical scoping, if a value cannot be found in the environment in which the function was defined then R will look for it in its parent environment and then the parent of the parent environment and so on until a match is found. If a function is defined inside the global environment (workspace) then the top level environment is the global environment itself. However if the function is defined in a package then the top-level environment is the namespace of that package. After the top level environment has been searched, R will look for the value in the search list until it hits the 'empty environment' which is the parent to the base package. R will come up with an error message if the value cannot be found after reaching the empty environment. The empty environment does not have a parent environment.

An interesting feature of R is that a function can be defined within a function. In other words, a function can return another function as an output. For the function which is defined within another function, the primary environment is the inside of the host function. R usually defines a temporary environment inside the host whenever such a system of function is sourced. R will first search for a value in the host function, then in the global environment, and then across the search list until a match is found. This feature of R functions combined with the manipulation of scoping rules makes it possible to undertake the project in hand and many other complex programming projects.

The figure above provides an sketch as to how the scoping rules would treat the assignment in hand. Both `makeCacheMatrix` and `cacheSolve` functions are global objects as for both of them the 'defining environment' is the global environment. For the `set`, `get`, `setinv` and `getinv` functions the defining environment is the inside of the `makeCacheMatrix` function. To bind a value to the `set` function, R will first look inside the the host function `makeCacheMatri` and then the global environment. Similarly, for a free variable inside `set`, R will first look for a value inside the function itself, then in its parent function and finally in the global environment.

A more technical demonstration of R environments has been presented in **Appendix A**, using the `environment` function in R and some of its applications. The `parent.env` function has been used to demonstrate the scoping hierarchy.

Appendix A: R-Environment and Scoping Hierarchy

The Construct of 'makeCacheMatrix'

```
source("cachematrix.R")
makeCacheMatrix()
```

```
## $set
## function (y)
## {
##     x <<- y
##     IM <<- NULL
## }
## <environment: 0x000000000a51d448>
##
## $get
## function ()
## x
## <environment: 0x000000000a51d448>
##
## $setinv
## function (inv)
## IM <<- inv
## <environment: 0x000000000a51d448>
##
## $getinv
```

```
## function ()
## IM
## <environment: 0x000000000a51d448>
```

Global Environment

```r
environment(makeCacheMatrix)
```

```
## <environment: R_GlobalEnv>
```

```r
environment(cacheSolve)
```

```
## <environment: R_GlobalEnv>
```

```r
ls(environment(makeCacheMatrix))
```

```
## [1] "cacheSolve"      "makeCacheMatrix"
```

```r
ls.str(environment(makeCacheMatrix))
```

```
## cacheSolve : function (x, ...)
## makeCacheMatrix : function (x = matrix())
```

How the System of Functions Work

Making the Special "Matrix"

```r
M <- makeCacheMatrix(matrix(c(13, 5, 18, 10), 2, 2))
ls(environment(makeCacheMatrix))
```

```
## [1] "cacheSolve"      "M"               "makeCacheMatrix"
```

```r
ls.str(environment(makeCacheMatrix))
```

```
## cacheSolve : function (x, ...)
## M : List of 4
##  $ set   :function (y)
##  $ get   :function ()
##  $ setinv:function (inv)
##  $ getinv:function ()
## makeCacheMatrix : function (x = matrix())
```

Environment of the Subfunctions Before Implementing 'cacheSolve'

```r
environment(M$set)
```

```
## <environment: 0x0000000007658c40>
```

```r
environment(M$get)
```

```
## <environment: 0x0000000007658c40>
```

```r
environment(M$setinv)
```

```
## <environment: 0x0000000007658c40>
```

```r
environment(M$getinv)
```

```
## <environment: 0x0000000007658c40>
```

```r
ls(environment(M$set))
```

```
## [1] "get"    "getinv" "IM"     "set"    "setinv" "x"
```

```r
ls.str(environment(M$set))
```

```
## get : function ()
## getinv : function ()
## IM :  NULL
## set : function (y)
## setinv : function (inv)
## x :  num [1:2, 1:2] 13 5 18 10
```

Extracting the Inputed Matrix and Its Inverse Before 'cacheSolve'

```r
M$get()
```

```
##      [,1] [,2]
## [1,]   13   18
## [2,]    5   10
```

```r
M$getinv()
```

```
## NULL
```

Implementing 'cacheSolve'

```r
cacheSolve(M)
```

```
##        [,1]   [,2]
## [1,]  0.250 -0.450
## [2,] -0.125  0.325
```

Environment of the Subfunctions After Implementing 'cacheSolve'

```r
ls(environment(M$set))
```

```
## [1] "get"    "getinv" "IM"     "set"    "setinv" "x"
```

```r
ls.str(environment(M$set))
```

```
## get : function ()
## getinv : function ()
## IM :  num [1:2, 1:2] 0.25 -0.125 -0.45 0.325
## set : function (y)
## setinv : function (inv)
## x :  num [1:2, 1:2] 13 5 18 10
```

Extracting the Inputed Matrix and its Inverse After 'cacheSolve'

```r
M$get()
```

```
##      [,1] [,2]
## [1,]   13   18
## [2,]    5   10
```

```r
M$getinv()
```

```
##        [,1]   [,2]
## [1,]  0.250 -0.450
## [2,] -0.125  0.325
```

Getting the Inverse From the Cache

```r
cacheSolve(M)
```

```
## getting cached data (inverse matrix)
```

```
##        [,1]   [,2]
## [1,]  0.250 -0.450
## [2,] -0.125  0.325
```

Scoping Hierarchy

Scoping will continue till it reaches the 'base environment' unless a match is found on the way. The parent for the base is typically the empty environment. When scoping reaches the empty environment, the process stops. An empty environment does not have a parent. Base, Global and Empty are system environments.

```r
beta <- environment(M$set)
beta
```

```
## <environment: 0x0000000007658c40>
```

```r
alpha <- parent.env(beta)
alpha
```

```
## <environment: R_GlobalEnv>
```

```r
search1 <- parent.env(alpha)
search1
```

```
## <environment: package:stats>
## attr(,"name")
## [1] "package:stats"
## attr(,"path")
## [1] "C:/Program Files/R/R-3.2.1/library/stats"
```

```r
search2 <- parent.env (search1)
search2
```

```
## <environment: package:graphics>
## attr(,"name")
## [1] "package:graphics"
## attr(,"path")
## [1] "C:/Program Files/R/R-3.2.1/library/graphics"
```

```r
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

```r
baseenv()
```

```
## <environment: base>
```

```r
parent.env(baseenv())
```

```
## <environment: R_EmptyEnv>
```

Appendix B: Test Outputs

2 x 2 matrix

```r
source("cachematrix.R")
M <- makeCacheMatrix(matrix(10:13, 2, 2))
M$get()
```

```
##      [,1] [,2]
## [1,]   10   12
## [2,]   11   13
```

```r
M$getinv() # no data on inverse matrix is cached yet
```

```
## NULL
```

```r
cacheSolve(M)
```

```
##      [,1] [,2]
## [1,] -6.5    6
## [2,]  5.5   -5
```

```r
cacheSolve(M) # the data on inverse matrix is cached already
```

```
## getting cached data (inverse matrix)

##      [,1] [,2]
## [1,] -6.5    6
## [2,]  5.5   -5
```

3 x 3 matrix

```r
source("cachematrix.R")
M <- makeCacheMatrix(matrix(c(20, 13, 9, 51, 18, 33, 9, 101, 47), 3, 3))
M$get()
```

```
##      [,1] [,2] [,3]
## [1,]   20   51    9
## [2,]   13   18  101
## [3,]    9   33   47
```

```r
M$getinv() # no data on inverse matrix is cached yet
```

```
## NULL
```

```r
cacheSolve(M)
```

```
##              [,1]         [,2]         [,3]
## [1,]  0.077382619  0.065341174 -0.155231961
## [2,] -0.009272224 -0.026727652  0.059211550
## [3,] -0.008307664  0.006254084  0.009427798
```

```r
cacheSolve(M) # the data on inverse matrix is cached already
```

```
## getting cached data (inverse matrix)

##              [,1]         [,2]         [,3]
## [1,]  0.077382619  0.065341174 -0.155231961
## [2,] -0.009272224 -0.026727652  0.059211550
## [3,] -0.008307664  0.006254084  0.009427798
```

4 x 4 matrix

```
source("cachematrix.R")
M <- makeCacheMatrix(matrix(c(15, 11, 34, 50, 19, 68, 44, 12, 30, 11, 56, 98, 88, 1, 26, 10), 4, 4))
M$get()
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   15   19   30   88
## [2,]   11   68   11    1
## [3,]   34   44   56   26
## [4,]   50   12   98   10
```

```
M$getinv() # no data on inverse matrix is cached yet
```

```
## NULL
```

```
cacheSolve(M)
```

```
##              [,1]         [,2]         [,3]         [,4]
## [1,] -0.085666715 -0.179583600  0.357613044 -0.157968463
## [2,]  0.006943442  0.029508340 -0.028894926  0.011073682
## [3,]  0.041815295  0.088620384 -0.179581022  0.090074022
## [4,]  0.010211551 -0.005971773  0.006502711 -0.006171519
```

```
cacheSolve(M) # the data on inverse matrix is cached already
```

```
## getting cached data (inverse matrix)
```

```
##              [,1]         [,2]         [,3]         [,4]
## [1,] -0.085666715 -0.179583600  0.357613044 -0.157968463
## [2,]  0.006943442  0.029508340 -0.028894926  0.011073682
## [3,]  0.041815295  0.088620384 -0.179581022  0.090074022
## [4,]  0.010211551 -0.005971773  0.006502711 -0.006171519
```

A Technical Note:

The inverse of a singular matrix, a matrix that incurs a determinant of zero (0), cannot be calculated. Inverse of a singular matrix is not defined. R will produce the error, "system is exactly singular." in such cases. It is advisable to use a non-singular matrix for evaluating this exercise.

Example:

```
source("cachematrix.R")
M <- makeCacheMatrix(matrix(1:16, 4, 4))
M$get()
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

```r
det(M$get()) # determinant of the singular matrix
```

```
## [1] 0
```

The `cacheSolve` function will produce the following error message:

```
> cacheSolve(M)
```

```
 Error in solve.default(data, ...) :
  Lapack routine dgesv: system is exactly singular: U[3,3] = 0
```