Portfolio Report 8: Matrices

Jake Spiteri

2019

Matrices

In this report we will look at how `R` treats matrices.

We will explore the `Matrix` package, which extends the basic `R` functionality for matrices. For example, it seems odd that base `R` does not have a method of determining the rank of a matrix up to a certain tolerance — the `Matrix` package adds this functionality.

Dense matrices

A matrix is a two dimensional data structure in `R`. We specify a matrix by its columns, and we can change this default functionality by setting `byrow=T`. We can assign names to the rows and columns of a matrix, and easily change these names by redefining `colnames(<matrix>)` and `rownames(<matrix>)`. This is shown below.

```
# matrices are defined by columns
m <- matrix(1:4, 2, 2)
m
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
# print the class of the matrix object
class(m)
```

```
## [1] "matrix"
```

```
# we can observe attributes
attributes(m)
```

```
## $dim
## [1] 2 2
```

```
# we can print this attribute
dim(m)
```

```
## [1] 2 2
```

```
# we can assign row and column names when creating the matrix
m <- matrix(c(rnorm(2,0,1), rnorm(2, 1, 1)), 2, 2,
            dimnames = list(c("obs 1", "obs 2"), c("r.v. 1", "r.v. 2")))
m
```

```
##           r.v. 1    r.v. 2
## obs 1 -0.8890308 1.2196586
```

```
## obs 2  0.2697414 0.7219776
```

```
# print colnames
colnames(m)
```

```
## [1] "r.v. 1" "r.v. 2"
```

```
# print rownames
rownames(m)
```

```
## [1] "obs 1" "obs 2"
```

```
# redefine colnames
colnames(m) <- c("new col name 1", "new col name 2")

# redefine rownames
rownames(m) <- c("new row name 1", "new row name 2")

# print new matrix
m
```

```
##                new col name 1 new col name 2
## new row name 1     -0.8890308      1.2196586
## new row name 2      0.2697414      0.7219776
```

It's quite odd but when we select a row or column of a matrix, `R` returns a vector and loses the matrix dimensions. When selecting a row of a 2×2 matrix we would expect a vector of dimension 1×2.

```
# select row
m[1,]
```

```
## new col name 1 new col name 2
##     -0.8890308      1.2196586
```

```
# check dimensions of row
dim(m[1,])
```

```
## NULL
```

In order to change this behavior, we can specify `drop=F`.

```
# select row
m[1,,drop=F]
```

```
##                new col name 1 new col name 2
## new row name 1     -0.8890308       1.219659
```

```
# check dimensions
dim(m[1,,drop=F])
```

```
## [1] 1 2
```

This is inconvenient but it allows us to maintain the dimensions of a matrix when indexing. When selecting a column of a matrix we would expect to obtain a column vector, but this is also not the case unless we specify `drop=F`.

`R` also has higher-dimensional data structures called arrays. We can think of an array as stacked matrices. For example, specifying dimension `(4,5,6)` creates 6 matrices with 4 rows and 5 columns.

Below we will specify an array of dimension `(2,2,3)`. Recall that `R` specifies a matrix by its columns — we will demonstrate this.

```r
arr <- array(1:3, c(2,2,3))

dim(arr)
```

```
## [1] 2 2 3
```

```r
arr[,,1]
```

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

```r
arr[,,2]
```

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

```r
arr[,,3]
```

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

```r
# We can also select the rows or columns
arr[1,,]
```

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

Note in the above that when we do `arr[1,,]` this selects the first row of each matrix, which are $(1, 3)$, $(2, 1)$, and $(3, 2)$, and puts them into a matrix.

Solving linear systems

A common problem encountered in linear algebra is solving a system of linear equations. That is, finding x that solves $Ax = b$. On paper we would simply write $x = A^{-1}b$. There are two ways of implementing this solution in `R`: we could use `x <- solve(A) %*% b`; we could directly solve the system using `x = solve(A, b)`. Naturally we would expect these solutions to be equal, but this is not necessarily the case in `R`. We shall show this by an example with Hilbert matrices which are well known to be close to singular.

```r
A <- as.matrix(Hilbert(9))

# finding the rank is inconsistent
rankMatrix(A)
```

```
## [1] 9
## attr(,"method")
## [1] "tolNorm2"
## attr(,"useGrad")
## [1] FALSE
## attr(,"tol")
## [1] 1.998401e-15
```

```r
rankMatrix(A, tol = 1e-9)
```

```
## [1] 7
```

```
## attr(,"method")
## [1] "tolNorm2"
## attr(,"useGrad")
## [1] FALSE
## attr(,"tol")
## [1] 1e-09
```

```
# show nearly singular
det(A)
```

```
## [1] 9.720256e-43
```

```
# create a linear system
x <- matrix(rnorm(9), 9, 1)
b <- A %*% x

# solve
x1 <- solve(A) %*% b
x2 <- solve(A, b)

# compute some metric of the error
norm(x-x1)
```

```
## [1] 0.0001425924
```

```
norm(x-x2)
```

```
## [1] 3.41572e-05
```

```
# compare the residuals
norm(b - A %*% x1)
```

```
## [1] 1.532503e-06
```

```
norm(b - A %*% x2)
```

```
## [1] 1.665335e-16
```

We see that solving the system of equations directly is far superior, both in terms of accuracy and speed. In R we should always avoid directly computing the inverse of a matrix, unless the inverted matrix will be used multiple times to solve linear systems. Even then, there are better ways to approach the problem. We can always look at different decompositions of the matrix and find a more numerically stable method of solving the linear system. This is what functions such as `lm()` do — they do not directly solve for predicted coefficients using $\hat{\beta} = (X^T X)^{-1} X^T Y$, they rely on the QR decomposition. Before we compute the inverse of a matrix we should try to find a more elegant solution by decomposing the matrices involved.

Numerical stability and finite precision arithmetic

In most programming packages, a floating point number is stored as a 'double' precision number. According to the IEEE754 standard a double number consists of 64 binary bits arranged as follows:

- sign bit: 1 bit
- exponent bit: 11 bits
- significant precision: 52 bits

Thus, we are limited in the length of the numbers we can store. This means there exist smallest and largest numbers in R. We can see this demonstrated below.

```
c(2^(-1075), 2^(-1074), 2^1023, 2^1024, 2^1024 / 2^1024)
```

```
## [1]  0.000000e+00 4.940656e-324 8.988466e+307           Inf           NaN
```

We see that 2^{1023} is the largest number that can be stored. Any higher and `R` returns `Inf`. Note that we can no longer do arithmetic with `Inf` — even a simple operation such as $2^{1024}/2^{1024}$ cannot be computed.

We have 52 precision bits, so we can expect an error rate of approximately 10^{-16} ($2^{-52} \approx 2.22 \times 10^{-16}$) in numerical computations of double numbers. This is shown below. The `R` documentation mentions that we should avoid using more than 15 digits.

```
# R will not suggest digits >= 16
# so if we add a very small number to 1, it will not print
print(1 + 1e-14)
```

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

```
# specify digits
print(1 + 1e-14, digits=15)
```

```
## [1] 1.00000000000001
```

These numerical characteristics of the machine are also stored in `R`. We can access them using `.Machine`. For example, we can find the smallest x such that $1 + x \neq 1$ (the machine epsilon) using `.Machine$double.eps`. `.Machine` stores many interesting characteristics and is definitely worth exploring.

```
#
.Machine$double.eps
```

```
## [1] 2.220446e-16
```

Arithmetic with double floating point numbers is not so easy in `R`. This is because of numerical errors — this is a problem for even trivial calculations as shown below. These errors occur because floating points can only represent the dyadic rationals, which are numbers with denominators which are powers of 2. Hence 0.1 is simply approximated as a dyadic rational. This leads to unexpected results, for example, `0.1+0.2 == 0.3` is `FALSE`!

```
# not zero
0.1+0.2-0.3
```

```
## [1] 5.551115e-17
```

```
# looks okay
0.1 + 0.2
```

```
## [1] 0.3
```

```
# check
0.1 + 0.2 == 0.3
```

```
## [1] FALSE
```

```
# all.equal uses a tolerance parameter
all.equal(0.1 + 0.2, 0.3)
```

```
## [1] TRUE
```

We see that the equality comparison `==` often does not behave the way we expect it to. When working with double floating point numbers we should avoid testing equality using `==` and instead use `all.equal()` which ignores machine errors using a tolerance parameter.

`R` also has an integer data type 'long'.

```r
# 1L seems to be the same as 1
1L
```

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

```r
# test equality
1 == 1L
```

```
## [1] TRUE
```

```r
identical(1, 1L)
```

```
## [1] FALSE
```

Given that error is introduced with arithmetic operations, the errors produced by matrix multiplication are much larger due to the numerous operations of addition and scalar multiplication.

Suppose we have matrices $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, and $C \in \mathbb{R}^{m\times 1}$. Then the flop cost of computing AB is of the order $\mathcal{O}(n^2m)$ as it requires n^2m multiplications. The flop cost of computing BC is of the order $\mathcal{O}(nm)$. Let $m = n$, then to compute ABC has cost of the order $\mathcal{O}(n^3)$ and to compute $A(BC)$ has cost of the order $\mathcal{O}(n^2)$. Thus, whilst these quantities are the same algebraically, the order of multiplications impacts the number of multiplications and thus impacts the error produced. In the following example we expect computing ABC to have an error approximately 10 times larger than computing $A(BC)$.

```r
# check error for matrix addition and subtraction
A <- matrix(rnorm(100),10,10); B <- matrix(rnorm(100),10,10); C <- rnorm(10)
summary(c((A + B) - A - B))
```

```
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## -4.441e-16  0.000e+00  0.000e+00 -1.023e-18  0.000e+00  4.441e-16
```

```r
# check error for matrix multiplication
summary(c(A%*%B%*%C - A%*%(B%*%C)))
```

```
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## -7.105e-15 -8.882e-16  6.661e-16 -8.882e-17  2.109e-15  3.553e-15
```

```r
# check computation times
n <- 5000
A <- matrix(rnorm(n^2), n, n); B <- matrix(rnorm(n^2), n, n); C <- rnorm(n)
c(system.time(A%*%B%*%C), system.time(A%*%(B%*%C)))
```

```
##  user.self   sys.self     elapsed user.child  sys.child  user.self   sys.self
##     33.961      3.784      6.741      0.000      0.000      0.506      0.179
##    elapsed user.child  sys.child
##      0.143      0.000      0.000
```

When working with matrices we must think carefully about what we are trying to do, in order to avoid unnecessary multiplications which can introduce numerical errors.

Suppose we want to sum the diagonal entries of the matrix product AB, for $A \in \mathbb{R}^{n\times m}$, $B \in \mathbb{R}^{m\times n}$ where $m << n$. Note the costs of forming AB compared to BA ($\mathcal{O}(n^2m)$ compared to $\mathcal{O}(m^2n)$). In R there are multiple ways we can do this.

- We can compute the matrix product AB, extract the diagonal entries, and sum them.
- We can instead compute the matrix product BA, extract the diagonal entries, and sum them. This uses the fact that $\text{tr}(AB) = \text{tr}(BA)$.
- We can sum the elements of $A * B^T$ (element-wise multiplication), since $\text{tr} = \sum_{i,j} A_{ij}B_{ji}$.

```r
n <- 10000; m <- 1000
A <- matrix(rnorm(n*m), n, m); B <- matrix(rnorm(n*m), m, n)

# first, second, and third method
first <- sum(diag(A %*% B))
second <- sum(diag(B %*% A))
third <- sum(A * t(B))

# compare errors
summary(first - second)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 1.819e-12 1.819e-12 1.819e-12 1.819e-12 1.819e-12 1.819e-12
```

```r
summary(first - third)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 1.819e-12 1.819e-12 1.819e-12 1.819e-12 1.819e-12 1.819e-12
```

```r
summary(second - third)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0       0       0       0       0       0
```

```r
# compare computation time
system.time(sum(diag(A %*% B))) # first
```

```
##    user  system elapsed
##  34.137   2.913   6.773
```

```r
system.time(sum(diag(B %*% A))) # second
```

```
##    user  system elapsed
##   2.656   0.687   0.549
```

```r
system.time(sum(A * t(B)))      # third
```

```
##    user  system elapsed
##    0.23    0.06    0.29
```

The second method is much better than the first both in terms of precision and run time. The third method is better than the second method.

Sparse Matrices

The R packages `Matrix` provides additional functionality for both dense matrices and sparse matrices. In the `Matrix` packages, dense matrices are stored as `dgeMatrix` objects, and sparse matrices are stored as `dgCMatrix` objects. A useful function we have already seen is `rankMatrix` which provides the rank of the input matrix up to a certain tolerance.

```r
library(Matrix)
A <- Matrix(c(1,1,2,2),2,2)
B <- Matrix(c(1,1,2,2) + rnorm(4)/1e10,2,2)

c(rankMatrix(A), rankMatrix(B), rankMatrix(B, tol=1e-7))
```

```
## [1] 1 2 1
```

A sparse matrix is one in which most entries are equal to zero. It's inefficient to store all of these zeros in memory, and so we only store the non-zero entries.

```
set.seed(25)
nrows <- ncols <- 500
entries <- sample(c(0,1,2),nrows*ncols,TRUE,c(0.98, 0.01, 0.01))
m1 <- matrix(entries, nrows, ncols)
m2 <- Matrix(entries, nrows, ncols, sparse = TRUE)
c(class(m1), class(m2))
```

```
## [1] "matrix"    "dgCMatrix"
```

```
m1[1:2,1:10]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    0    0    0    0    0    0    0    0    0     0
## [2,]    0    1    0    0    0    0    0    0    0     1
```

```
m2[1:2,1:10]
```

```
## 2 x 10 sparse Matrix of class "dgCMatrix"
##
## [1,] . . . . . . . . . .
## [2,] . 1 . . . . . . . 1
```

```
c(object.size(m1), object.size(m2))
```

```
## [1] 2000216   62904
```

We see that a sparse matrix defined using the `Matrix` package has class `dgCMatrix`. The 'd' stands for digit, the 'g' stands for general, and the 'C' stands for column. We can convert a `dgCMatrix` to a `dgeMatrix` — the standard class for dense matrices — but we will lose the memory improvements of the compressed sparse format. Alternatives to the `dgCMatrix` are the `dgRMatrix` and `dgTMatrix` classes. We can convert a `dgCMatrix` into a `dgTMatrix` (a triplet matrix), but not a `dgRMatrix`.

```
# display the structure
str(m2, max.level = 1)
```

```
## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots
```

```
str(m2)
```

```
## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots
##   ..@ i       : int [1:4950] 5 46 141 211 315 327 380 430 482 485 ...
##   ..@ p       : int [1:501] 0 10 21 29 42 47 54 58 67 76 ...
##   ..@ Dim     : int [1:2] 500 500
##   ..@ Dimnames:List of 2
##   .. ..$ : NULL
##   .. ..$ : NULL
##   ..@ x       : num [1:4950] 2 1 1 1 1 1 1 1 1 1 ...
##   ..@ factors : list()
```

```
which(m2[,1]>0)
```

```
##  [1]   6  47 142 212 316 328 381 431 483 486
```

```
# show difference in memory size
object.size(as(m2, 'dgeMatrix'))
```

```
## 2001176 bytes
```

```r
object.size(as(m2, 'dgTMatrix'))
```

```
## 80696 bytes
```

```r
object.size(m2) # dgCMatrix
```

```
## 62904 bytes
```

```r
# print the matrices before and after coercion
m2[1:2, 1:10]
```

```
## 2 x 10 sparse Matrix of class "dgCMatrix"
##
## [1,] . . . . . . . . . .
## [2,] . 1 . . . . . . . 1
```

```r
as(m2, 'dgTMatrix')[1:2, 1:10]
```

```
## 2 x 10 sparse Matrix of class "dgTMatrix"
##
## [1,] . . . . . . . . . .
## [2,] . 1 . . . . . . . 1
```

Operations for sparse matrices

```r
A <- B <- Matrix(c(1,1,0,0,0,1),3,2, sparse=TRUE)
A
```

```
## 3 x 2 sparse Matrix of class "dgCMatrix"
##
## [1,] 1 .
## [2,] 1 .
## [3,] . 1
```

```r
# division maintains class
A/10
```

```
## 3 x 2 sparse Matrix of class "dgCMatrix"
##
## [1,] 0.1 .
## [2,] 0.1 .
## [3,] .   0.1
```

```r
# addition and subtraction change class to dgeMatrix
A+1; A-1
```

```
## 3 x 2 Matrix of class "dgeMatrix"
##      [,1] [,2]
## [1,]    2    1
## [2,]    2    1
## [3,]    1    2
```

```
## 3 x 2 Matrix of class "dgeMatrix"
##      [,1] [,2]
## [1,]    0   -1
## [2,]    0   -1
## [3,]   -1    0
```

```r
# matrix multiplication changes class
A %*% c(1,1)
```

```
## 3 x 1 Matrix of class "dgeMatrix"
##      [,1]
## [1,]    1
## [2,]    1
## [3,]    1
```

```r
# addition and subtraction of two dgCMatrix objects produces an object of the same class
A+B; A-B
```

```
## 3 x 2 sparse Matrix of class "dgCMatrix"
##
## [1,] 2 .
## [2,] 2 .
## [3,] . 2
```

```
## 3 x 2 sparse Matrix of class "dgCMatrix"
##
## [1,] 0 .
## [2,] 0 .
## [3,] . 0
```

```r
# matrix multiplication of two dgCMatrix objects gives a dgCMatrix object
A %*% t(B)
```

```
## 3 x 3 sparse Matrix of class "dgCMatrix"
##
## [1,] 1 1 .
## [2,] 1 1 .
## [3,] . . 1
```

```r
# row or column binding dgCMatrix objects returns a dgCMatrix
cbind(A,A); rbind(A,A)
```

```
## 3 x 4 sparse Matrix of class "dgCMatrix"
##
## [1,] 1 . 1 .
## [2,] 1 . 1 .
## [3,] . 1 . 1
```

```
## 6 x 2 sparse Matrix of class "dgCMatrix"
##
## [1,] 1 .
## [2,] 1 .
## [3,] . 1
## [4,] 1 .
## [5,] 1 .
## [6,] . 1
```

Solving large linear systems

Solving linear systems for very large and sparse systems is very difficult, as the inverse of a sparse matrix is not necessarily sparse. To avoid these problems we should directly solve the linear system using `solve(A, b)`.

```r
nrow <- ncol <- 500
A <- Matrix(sample(c(0,1), nrow*ncol, TRUE, c(0.98, 0.02)), nrow, ncol, sparse=TRUE)
A.inv <- solve(A)

# object sizes
c(object.size(A), object.size(A.inv))
```

```
## [1] 1884440 2001176
```

```r
# possible elements
nrow*ncol
```

```
## [1] 250000
```

```r
# entries in A
nnzero(A)
```

```
## [1] 4897
```

```r
# entries in A.inv
nnzero(A.inv)
```

```
## [1] 250000
```