Portfolio 7

Sam Bowyer

Matrices

Matrices are an extremely important data structure in `R` so it will be instructive to examine their usage in some detail through this portfolio. In particular we will separately discuss dense matrices (which we are already familiar with) and sparse matrices.

Dense Matrices

As previously seen we can create a matrix by specifying its entries by columns as follows:

```
a = matrix(1:12, 4, 3)
a
```

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

However, we can also specify the values by row if we set `byrow=T`:

```
b = matrix(1:12, 4, 3, byrow=T)
b
```

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

The `4` and `3` in here specify the dimensions of the matrix and can be recovered using the `dim` function:

```
dim(a)
```

```
## [1] 4 3
```

We can also give the columns and rows names, either at creation of the matrix or afterwards:

```
c = matrix(1:15, 3, 5, dimnames=list(c('X', 'Y', 'Z'), c('1', '2', '3', '4', '5')))
c
```

```
##   1 2 3  4  5
## X 1 4 7 10 13
## Y 2 5 8 11 14
## Z 3 6 9 12 15
```

```
rownames(c)
```

```
## [1] "X" "Y" "Z"
```

```r
colnames(c)
```

```
## [1] "1" "2" "3" "4" "5"
```

```r
rownames(c) = c('A','B','C')
c
```

```
##   1 2 3  4  5
## A 1 4 7 10 13
## B 2 5 8 11 14
## C 3 6 9 12 15
```

Rows and columns can be obtained as follows:

```r
c[1,]
```

```
##  1  2  3  4  5
##  1  4  7 10 13
```

```r
c[,1]
```

```
## A B C
## 1 2 3
```

Note that by default this returns a vector for the column, we can use the `drop=F` indexing option to return a matrix instead:

```r
c[,1,drop=F]
```

```
##   1
## A 1
## B 2
## C 3
```

Matrices come with many features in base `R` including transposition (via `t(...)`) matrix multiplication (via `%*%`), inversion (via `solve(...)`) and eigen-decomposition (via `eigen(...)`), for example:

```r
d = t(a) %*% b
d
```

```
##      [,1] [,2] [,3]
## [1,]   70   80   90
## [2,]  158  184  210
## [3,]  246  288  330
```

```r
e = matrix(c(-1, 1.5, 1, -1), 2, 2)
e
```

```
##      [,1] [,2]
## [1,] -1.0    1
## [2,]  1.5   -1
```

```r
e_inverse = solve(e)
e_inverse
```

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

```r
eigen(e)
```

```
## eigen() decomposition
```

```
## $values
## [1] -2.2247449  0.2247449
##
## $vectors
##            [,1]      [,2]
## [1,] -0.6324555 0.6324555
## [2,]  0.7745967 0.7745967
```

Finally we also note that `R` has a `Matrix` package that greatly expands the functionality possible with matrices, for instance it allows us to easily get the rank of a matrix with `rankMatrix` (note that we are generating these matrices with the capitalised `Matrix` function and no longer the lowercase `matrix` function):

```
library(Matrix)
f = Matrix(c(2,4,2,4), nrow=2, ncol=2)
g = Matrix(c(2,4,2,5), nrow=2, ncol=2)
c(rankMatrix(f),rankMatrix(g))
```

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

Sparse Matrices

Matrix operations are relatively fast in `R` and can be sped up even further by using C/C++ code (with the `Rcpp` package) alongside careful memory management, however, for very large matrices computation time can easily become intractable regardless of implementation. It is here useful to distinguish the case in which a matrix is large but sparse (i.e. has mostly entries of 0); ordinarily large matrices would be difficult to work with efficiently but utilising the sparsity present we can improve the computational efficiency by not storing each of the 0 entries separtely. For this we'll use the `Matrix` package again, but this time with the argument `sparse=TRUE`.

```
h = matrix(c(1, rep(0,999)), 10, 100)
h_sparse = Matrix(c(1, rep(0,999)), nrow=10, ncol=100, sparse=TRUE)
h[1:5,1:5]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    0    0    0    0
## [3,]    0    0    0    0    0
## [4,]    0    0    0    0    0
## [5,]    0    0    0    0    0
```

```
h_sparse[1:5,1:5]
```

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

```
c(object.size(h), object.size(h_sparse))
```

```
## [1] 8216 1920
```

We can see here that the sparse matrix requires a lot less storage than the dense matrix, but note that there are storage costs involved in the sparse matrix that would result in higher storage requirements for small, sparse matrices:

```r
i = matrix(c(1, rep(0,24)), 5, 5)
i_sparse = Matrix(c(1, rep(0,24)), nrow=5, ncol=5, sparse=TRUE)
i
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    0    0    0    0
## [3,]    0    0    0    0    0
## [4,]    0    0    0    0    0
## [5,]    0    0    0    0    0
```

```r
i_sparse
```

```
## 5 x 5 diagonal matrix of class "ddiMatrix"
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    .    .    .    .
## [2,]    .    0    .    .    .
## [3,]    .    .    0    .    .
## [4,]    .    .    .    0    .
## [5,]    .    .    .    .    0
```

```r
c(object.size(i), object.size(i_sparse))
```

```
## [1]  416 1288
```

In this example we can also see that `i_sparse` is of class `ddiMatrix` whereas `h_sparse` was of class `dgCMatrix`. These refer to the different methods of compression used for storing these matrices–in particular, `dgC` refers to "digits" (rather than `l` for "logicals" for example) stored in a "general" (as opposed to e.g. triangular, symmetric or diagonal) sparse matrix with storage based on the "columns" (instead of, say, `r` for "rows" or `t` for "triplets"). The class `ddiMatrix` comes from the fact that `i_sparse` is a diagonal matrix (which extends from the sparse matrix class). It makes sense then that we can use a variety of different sparse matrix types, such as `dgRMatrix` for digit matrices stored in compressed sparse row (CSR)—rather than compressed sparse column (CSC)—format. Furthermore we can often (though not always) convert between these different formats: we can convert from `dgCMatrix` to `dgTMatrix` (and vice versa), but we cannot convert from `dgCMatrix` or `dgTMatrix` to `dgRMatrix`. Finally, note that we can perform regular matrix operations with sparse matrices and that these will (hopefully) result in a speed up of computation time (though be wary of taking inverses of sparse matrices as the inverses may not themselves be sparse).

```r
j_vals = sample(c(0,1), 100000, replace=TRUE, prob=c(0.95, 0.05))
j = matrix(j_vals, 1000, 1000)
j_sparse = Matrix(j_vals, nrow=1000, ncol=1000, sparse=TRUE)

k_vals = sample(c(0,-1,-2), 100000, replace=TRUE, prob=c(0.8, 0.05, 0.15))
k = matrix(k_vals, 1000, 1000)
k_sparse = Matrix(k_vals, nrow=1000, ncol=1000, sparse=TRUE)

j_sparse[1:5,1:5]
```

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

```r
k_sparse[1:5,1:5]
```

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

```r
system.time(j%*%t(k))
```

```
##    user  system elapsed
##   0.416   0.008   0.425
```

```r
system.time(j_sparse%*%t(k_sparse))
```

```
##    user  system elapsed
##   0.026   0.004   0.030
```