Stat 3701 Lecture Notes: Matrices, Arrays, and Data Frames in R

Charles J. Geyer

August 12, 2020

1 License

2 R

- The version of R used to make this document is 4.0.2.
- The version of the `rmarkdown` package used to make this document is 2.3.
- The version of the `Matrix` package used to make this document is 1.2.18.

3 Matrices

The plural of *matrix* is *matrices* (it's Latin originally; this is like the plurals of *alumnus* and *alumna* being *alumni* and *alumnae*, respectively, or the plural of *datum* being *data*).

R, like MATLAB, can be considered a matrix language. It understands matrices better than most other computer languages. It has syntax especially for matrices and a lot of functions that understand matrices. The recommended package `Matrix` (recommended packages come with every R installation) understands even more about matrices.

Here is a matrix

```
fred <- matrix(1:6, nrow = 3, ncol = 2)
fred
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

From this example we see one very important thing to know about R. In R matrices, the components are stored in FORTRAN order (first index goes the fastest, rather than C or C++ order). This is the order R stuffs a vector into a matrix (as we see above).

If you don't want that order, there is an optional argument for that.

```
matrix(1:6, ncol = 2, byrow = TRUE)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
```

(we omitted the argument `nrow` because R can figure out that it must be the length of the first argument divided by `ncol`).

Also, as we have seen before, R uses one-origin indexing like FORTRAN rather than zero-origin indexing like C and C++.

Storage order is important when you want to run a matrix through a function that doesn't respect matrices.

```
as.character(fred)
```

```
## [1] "1" "2" "3" "4" "5" "6"
```

Oops! Lost our matrix information.

```
frank <- matrix(as.character(fred),
    nrow = nrow(fred), ncol = ncol(fred))
frank
```

```
##      [,1] [,2]
## [1,] "1"  "4"
## [2,] "2"  "5"
## [3,] "3"  "6"
```

But there is actually a better, less obvious way

```
storage.mode(fred) <- "character"
fred
```

```
##      [,1] [,2]
## [1,] "1"  "4"
## [2,] "2"  "5"
## [3,] "3"  "6"
```

4 Matrix Operations

4.1 Multiplication

4.1.1 Matrix times Matrix

Matrix multiplication is different from ordinary multiplication. If A and B are matrices, then $C = AB$ is only defined if the column dimension of A is the same as the row dimension of B, in which case if we denote the components of A by a_{ij} and similarly for B and C

$$c_{ij} = \sum_j a_{ij} b_{jk},$$

where we deliberately leave the bounds of the summation on j unspecified, the intention being that it does over all possible values, which, since j is the column index of A and the row index of B must be the column dimension of A and the row dimension of B, which hence must be the same (as we already said above).

4.1.2 Non-Commutative

The commutative law of multiplication says $xy = yx$ for all numbers x and y. It does not apply to matrix multiplication.

If A and B are matrices, then

- AB is only defined when the column dimension of A is the same as the row dimension of B, and
- BA is only defined when the column dimension of B is the same as the row dimension of A.

So it may be that either, both, or neither of AB and BA are defined.

And, even when both are defined, it may be that $AB \neq BA$.

4.1.3 Matrix times Vector

Vectors are considered as matrices with one column in order that multiplication of matrices and vectors make sense. This is a stupid mathematician's trick. Vectors are not really matrices. We only consider them as one-column matrices (called *column vectors* to emphasize we are doing this trick) in order to be able to use our previous definition of matrix multiplication here too.

4.1.4 Linear Functions

Abstractly, matrix multiplication corresponds to vector-to-vector linear functions. If A is a matrix with r rows and c columns, then the function f defined by

$$f(x) = Ax, \qquad \text{for all vectors } x \text{ of dimension } c,$$

maps vectors whose dimension is c to vectors whose dimension is r. A function like this is called a *linear function* by mathematicians.

4.1.5 Affine Functions

Again suppose A is a matrix with r rows and c columns. And suppose b is a vector of dimension r. The function g defined by

$$g(x) = b + Ax, \qquad \text{for all vectors } x \text{ of dimension } c,$$

also maps vectors whose dimension is c to vectors whose dimension is r. A function like this is called an *affine function* by mathematicians.

Clearly every linear function is an affine function, but an affine function defined by the displayed equation in this section is a linear function if and only if $b = 0$.

4.1.6 Mathematicians versus Everybody Else

Everybody besides mathematicians, and even mathematics teachers teaching courses below the level of the course called *linear algebra* (which is not a pre-requisite for this course) use different terminology.

- When mathematicians say *affine function* everybody else says *linear function*.
- When mathematicians say *linear function* everybody else says *linear function* too.

So everybody else does not distinguish between the two kinds of functions.

If we specialize to scalar-to-scalar functions (so A is a one-by-one matrix and b is a vector of length one) we get

$$f(x) = ax$$
$$g(x) = b + ax$$

and it is the latter that everybody calls *linear function* with *slope* a and *intercept* b. And the former is just the case when the intercept is zero, and that gets no special terminology.

4.1.7 Linear Models

Except everybody else is sloppy. In the term *linear models*, the statistical models fit by the R function `lm`, the "linear" means the mathematicians linear rather than everybody else's linear. These are models where the mean vector of the response vector has the form

$$\mu = M\beta,$$

where M is the so-called *model matrix* (a function of predictor data) and β is the so-called *coefficients* vector. And this is a linear function in the mathematician's sense.

4.1.8 Composition

Matrix multiplication corresponds to the *composition* of linear (in the mathematician's sense) functions.

If

$$f(x) = Ax$$
$$g(x) = Bx$$

the the *composition* of f and g, written $g \circ f$, is the function h defined by

$$h(x) = g(f(x)) = BAx,$$

that is,

- A is the matrix that corresponds to the linear function f,
- B is the matrix that corresponds to the linear function g, and
- BA is the matrix that corresponds to the linear function $h = g \circ f$.

Of course, the composition is only defined when the matrix multiplication is (when the column dimension of B is the same as the row dimension of A).

4.1.9 R

After that long theoretical digression we can get back to computing. The R operator for matrix multiplication is `%*%`.

```
fred <- matrix(1:6, ncol = 2)
sally <- matrix(1:8, nrow = 2)
fred
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
sally
```

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

```
fred %*% sally
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    9   19   29   39
## [2,]   12   26   40   54
## [3,]   15   33   51   69
```

```
sally %*% fred
```

```
## Error in sally %*% fred: non-conformable arguments
```

If you check it out, you will find that the R results agree with the mathematical definition.

R vectors are not matrices.

```
lou <- 1:2
is.matrix(lou)
```

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

but

```
fred %*% lou
```

```
##      [,1]
## [1,]    9
## [2,]   12
## [3,]   15
```

works.

If it bothers you that R is doing something tricky here, guessing that you want `lou` to be treated as a column vector (a one-column matrix) even though it really isn't, you can say

```
fred %*% cbind(lou)
```

```
##      lou
## [1,]   9
## [2,]  12
## [3,]  15
```

instead (more on the R function `cbind` below).

4.2 Addition

If A and B are matrices having the same dimensions, then $A + B$ is defined by componentwise addition. Here mathematics and R agree.

```
herman <- matrix(-(1:6), ncol = 2)
herman
```

```
##      [,1] [,2]
## [1,]   -1   -4
## [2,]   -2   -5
## [3,]   -3   -6
```

```
herman + fred
```

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

But what does R do when the dimensions are not the same?

```
fred + sally
```

```
## Error in fred + sally: non-conformable arrays
```

Oh! It does the right thing. R and math agree here, as they did for matrix multiplication.

4.3 Scalar Multiplication

This too is defined componentwise.

```
3 * fred
```

```
##      [,1] [,2]
## [1,]    3   12
## [2,]    6   15
## [3,]    9   18
```

4.4 Matrices are Really Vectors

Mathematically, any thingummies that can be added and multiplied by scalars *are* vectors (provided that they satisfy the axioms for vector spaces, which are found near the beginning of any linear algebra book, but which we don't need to fuss about here or anywhere in this course).

Of course, matrices are rather special vectors because, unlike other vectors, matrices can also be multiplied (sometimes, depending on dimensions).

The collection of all square matrices (meaning row and column dimensions are the same) of the same dimension is called a *linear algebra*. It is the mathematical structure that gives courses called "linear algebra" their name. These objects can be added, multiplied by scalars, and also multiplied by each other.

5 Special Binary Operators

We take a break from discussing matrices to discuss "special" binary operators. In R these are kind of like operator overloading in C++ but much easier to use, much easier to create, and much more powerful.

In C++ you can only overload the operators that C++ already has. This leads to highly unintuitive overloadings like overloading the shift operators `>>` and `<<` to mean something in the I/O (input-output) context.

In R you can define new binary operators: `%foo%` where `foo` is replaced by any valid R name, is a new operator. You define it this way

```
"%foo%" <- function(x, y) 42 # or whatever
fred %foo% sally
```

```
## [1] 42
```

The quotation marks in the function definition are necessary, because without them R would treat `%foo%` as a binary operator (that has already been defined), which would mean that it would have to have arguments on either side, and it doesn't. So without quotation marks this would be an error.

There are a bunch of these built into R of which we have already met one: `%*%` the matrix multiplication operator. Here are all in the R core.

function	package	what it does	other functions
`%/%`	Primitive	integer division	
`%%`	Primitive	remainder from integer division	
`%*%`	Primitive	matrix multiplication	`crossprod` and `tcrossprod`
`%in%`	base	set-theoretic element of operator	`is.element`
`%o%`	base	outer product of vectors	`outer`
`%x%`	base	Kronecker product of matrices	`kronecker`

"Primitive" isn't an R package, but rather lower level. They are the basic functions on which all other R functions are built. Their R definitions look like

```
get("%*%")
```

```
## function (x, y)  .Primitive("%*%")
```

"Other functions" refers to other functions that do exactly the same thing or almost the same thing.

```
c("red", "white", "blue", "chartreuse", "puce", "aqua") %in% colors()
```

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

```
is.element(c("red", "white", "blue", "chartreuse", "puce", "aqua"), colors())
```

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

Since the other "other functions" involve matrices, we are back to talking about matrices.

6 More on Matrices

6.1 Transpose

The *transpose* of a matrix having components a_{ij} is the matrix having components a_{ji}, that is, transposing a matrix turns rows into columns and vice versa. The R function that does this is called `t`.

```
fred
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
t(fred)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

The mathematical notation for the transpose of the matrix A is A^T or A' (some people like one, some the other).

6.2 Outer Product

Now we can explain the outer product of vectors or matrices. The outer product of x and y is xy^T (in math) or `x %*% t(y)` (in R). This always is defined for numeric vectors, but may or may not be defined for matrices (it depends on their dimensions).

```
1:3 %o% 1:5
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    2    4    6    8   10
## [3,]    3    6    9   12   15
```

```
outer(1:3, 1:5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    2    4    6    8   10
## [3,]    3    6    9   12   15
```

```
1:3 %*% t(1:5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    2    4    6    8   10
## [3,]    3    6    9   12   15
```

But, unlike `%o%` the function `outer` has an optional argument that allows one to change the arithmetic operation. The outer product of the vector having components x_i and the vector having components y_j is the matrix having components $x_i y_j$. But the R function `outer` allows you to change the multiplication to any binary operation.

```
outer(1:3, 1:5, "+")
```

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

```
outer(1:3, 1:5, max)
```

```
## Error in dim(robj) <- c(dX, dY): dims [product 15] do not match the length of object [1]
```

Ouch! That is a particularly user-hostile error message. Usually I can understand R error messages, but not that one. Looking at the examples for the help page for `outer` shows we need quotation marks.

```
outer(1:3, 1:5, "max")
```

```
## Error in dim(robj) <- c(dX, dY): dims [product 15] do not match the length of object [1]
```

Oops! That's not the issue. Try three.

```
outer(1:3, 1:5, "pmax")
```

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

Oh! The functions have to be vectorizing, so `pmax` instead of `max`. (See their help pages for the difference.)

So the big difference between `%o%` and `outer` is that the latter is a higher-order function, and hence way more powerful.

6.3 Kronecker Product

We're not even going to explain this, beyond the scope of this course.

6.4 Dimension

R matrices, like other R objects know what they are

```
is.matrix(fred)
```

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

```
dim(fred)
```

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

```
nrow(fred)
```

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

```
ncol(fred)
```

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

So `nrow(fred)` is short for `dim(fred)[1]` and `ncol(fred)` is short for `dim(fred)[2]`.

Rows and columns can also have names.

```
rownames(fred) <- c("red", "white", "blue")
colnames(fred) <- c("car", "truck")
fred
```

```
##       car truck
## red     1     4
## white   2     5
## blue    3     6
```

6.5 Indexing

6.5.1 Two Indices

The plural of *index* is *indices* (it's Latin originally; this is like *matrix* and *matrices*).

Since matrices have two indices they have two slots for subscripts

```
fred
```

```
##       car truck
## red     1     4
## white   2     5
## blue    3     6
```

```
fred[1, 1]
```

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

```
fred[1, ]
```

```
##   car truck
##     1     4
```

```
fred[ , 1]
```

```
##   red white  blue
##     1     2     3
```

```
fred[ , ]
```

```
##       car truck
## red     1     4
## white   2     5
## blue    3     6
```

But of course, all of these vectorize, and can be any of the four types of indexing that R understands (positive integers, negative integers, logical, or character).

```
fred[1:2, 1]
```

```
##   red white
##     1     2
```

```
fred[-1, -1]
```

```
## white  blue
##     5     6
```

```
fred[1:nrow(fred) <= 2, 1]
```

```
##   red white
##     1     2
```

```
fred["red", "car"]
```

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

6.5.2 The Square Brackets Function is a Footgun

The R function named `"["` is a footgun when applied to matrices (and arrays, which we haven't covered yet) because sometimes it produces a vector and sometimes a matrix (and sometimes an array), and sometimes you know what it is going to do and sometimes you don't.

It is almost always always wrong, when `fred` is a matrix whose dimensions you do not know and `i` and `j` are vectors specifying indices, to say `fred[i, j]` because you do not know whether it will return a vector or a matrix.

If it doesn't matter whether the result is a vector or a matrix, then maybe you're OK. But how do you know the result doesn't matter? How do you know everything you may ever do with the result?

Especially when you are writing functions that others may use, the square brackets function is a footgun. Two-thirds of the time or more, when I say `fred[i, j]` inside a function I am writing, it is a source of bugs. So you can do it if you want, but don't say I didn't warn you.

The way to get a result that you can depend on is to say `fred[i, j, drop = FALSE]` which does not "drop" the matrixness when the result has one of the dimensions equal to one.

```
fred[1:2, 1]
```

```
##   red white
##     1     2
```

```
fred[1:2, 1, drop = FALSE]
```

```
##       car
## red     1
## white   2
```

(the default value of `drop` is `TRUE`).

Summary: Always use `drop = FALSE` when you are inside a function or whenever you are not *sure* it makes no difference in what you are doing.

6.5.3 One Index

You can always use one index with matrices. But it may or may not do what you want.

```
fred[1:4]
```

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

But sometimes it is useful. The R functions `row` and `col` tell us the row and column indices of each position in a matrix.

```
row(fred)
```

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

```
col(fred)
```

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

and they allow us to do rather complicated things.

```
alfie <- fred %*% sally
alfie
```

```
##       [,1] [,2] [,3] [,4]
## red      9   19   29   39
## white   12   26   40   54
## blue    15   33   51   69
```

Note that we have row names inherited from `fred` but no column names (they would have been inherited from `sally` if it had had any, but it didn't).

```
alfie[abs(row(alfie) - col(alfie)) > 1] <- 0
alfie
```

```
##       [,1] [,2] [,3] [,4]
## red      9   19    0    0
## white   12   26   40    0
## blue     0   33   51   69
```

Here we use the square brackets function on the left-hand side of an assignment (so this is really using the function called `"[<-"`) and we are also using logical indexing.

This could also have been done (TIMTOWTDI)

```
library(Matrix)
alphie <- band(fred %*% sally, -1, 1)
alphie
```

```
## 3 x 4 Matrix of class "dgeMatrix"
##       [,1] [,2] [,3] [,4]
## red      9   19    0    0
## white   12   26   40    0
## blue     0   33   51   69
```

But I, being an old fart, who has been using R since long before the `Matrix` package existed, prefer the former, even though the former is *not functional programming* and the latter is. Also they are not quite the same.

```
class(alfie)
```

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

```
class(alphie)
```

```
## [1] "dgeMatrix"
## attr(,"package")
## [1] "Matrix"
```

```
identical(alfie, as.matrix(alphie))
```

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

6.5.4 One-Index Example: Hodges-Lehmann Estimator

A *Hodges-Lehmann* estimator associated with a procedure for making confidence intervals is the point one gets by letting the confidence level go to zero.

Sometimes this is already a well-known estimator. The Hodges-Lehmann estimator associated with t confidence intervals is the sample mean.

Sometimes it is something new. The Hodges-Lehmann estimator associated with the confidence intervals associated with the Wilcoxon signed rank test is the median of the Walsh averages, which are the averages of pairs

$$\frac{X_i + X_j}{2}, \qquad 1 \le i \le j \le n,$$

where the data points are X_i, $i = 1, \ldots, n$. Note that $i = j$ is allowed.

We can make these as follows. First we need some data to try it out on

```
set.seed(42)
x <- rcauchy(50)
```

and then we can make the Walsh averages as follows

```
o <- outer(x, x, "+") / 2
w <- o[row(o) <= col(o)]
```

there is another way (TIMTOWTDI)

```
w2 <- o[upper.tri(o, diag = TRUE)]
identical(w, w2)
```

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

(there is also a function `lower.tri` to extract lower triangles).

Then the Hodges-Lehmann estimator is

```
median(w)
```

```
## [1] -0.1410889
```

6.6 Solving Linear Equations

The equation $Ax = b$, where x and b are vectors (not necessarily of the same dimension) and A is a matrix, can also be thought of as a set of linear equations

$$\sum_j a_{ij} x_j = b_i, \qquad i = 1, \ldots, n.$$

Suppose A and b are known and x is the unknown to be solved for. Then we know from somewhere or other (high-school?) that if the number of unknowns is the same as the number of equations there is (maybe) a solution, and we have been taught how to find it by hand.

R can do that faster.

```
barbie <- matrix(1:9, 3)
ken <- 10:12
solve(barbie, ken)
```

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

```
eigen(barbie)
```

```
## eigen() decomposition
## $values
## [1]  1.611684e+01 -1.116844e+00 -5.700691e-16
##
## $vectors
##            [,1]       [,2]       [,3]
## [1,] -0.4645473 -0.8829060  0.4082483
## [2,] -0.5707955 -0.2395204 -0.8164966
## [3,] -0.6770438  0.4038651  0.4082483
```

Hmmmm. That example didn't show what we wanted it to show, but it does show that R is smart enough to detect when the equations *do not* have a solution.

We don't want to explain eigenvalues and eigenvectors here (more beyond the scope of this course), but they do show the reason that the equations have no solution is because

```
barbie[1, ] + barbie[3, ] - 2 * barbie[2, ]
```

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

is zero, so some of the equations are linear combinations of the others.

Try again.

```
barbie[3, 3] <- 20
barbie
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6   20
```

```
ken
```

```
## [1] 10 11 12
```

```
solve(barbie, ken)
```

```
## [1] -2  3  0
```

and this is indeed the solution

```
x <- solve(barbie, ken)
identical(barbie %*% x, ken)
```

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

Oops! That didn't work because the attributes are different. `barbie %*% x` is a matrix and `ken` is a vector.

```
identical(as.vector(barbie %*% x), ken)
```

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

It still didn't work. Why?

```
as.vector(barbie %*% x) - ken
```

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

```
typeof(barbie %*% x)
```

```
## [1] "double"
```

```
typeof(ken)
```

```
## [1] "integer"
```

Oh! They don't compare equal because the attributes don't match.

```
identical(as.numeric(barbie %*% x), as.numeric(ken))
```

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

Warning! Never compare numeric variables for equality and expect that you have to get the answer you expect. It would have been much safer to say

```
all.equal(as.numeric(barbie %*% x), as.numeric(ken))
```

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

The R function `identical` insists on exact bit-for-bit equality. This is unlikely to be the case when the objects are the results of complicated calculations. The R function `all.equal` compares with a tolerance. It says they are equal if they are *almost equal*. More on this later when we cover computer arithmetic.

6.6.1 Matrix inverse

Division of matrices is not defined, but some square matrices do have inverses. The inverse of the matrix A (if it exists) is written A^{-1}. and when it exists we have $Ax = b$ if and only if $x = A^{-1}b$.

The R function `solve` also calculates matrix inverses, when the second argument is missing. (So here is another example, where R could not do this unless the R function named `missing` existed. It is also an example of R's rather bizarre use of what C++ would call method overloading. Why not just use a different name for the function that calculates inverses?)

```
barbie.inverse <- solve(barbie)
barbie.inverse
```

```
##             [,1]        [,2]        [,3]
## [1,] -1.57575758  1.15151515  0.09090909
## [2,]  0.48484848  0.03030303 -0.18181818
## [3,]  0.09090909 -0.18181818  0.09090909
```

```
barbie.inverse %*% ken
```

```
##               [,1]
## [1,] -2.000000e+00
## [2,]  3.000000e+00
## [3,] -2.220446e-16
```

```
solve(barbie, ken)
```

```
## [1] -2  3  0
```

You should rarely, if ever, calculate a matrix inverse. If you want to solve linear equations, use the R function `solve` with two arguments rather than `solve` with one argument followed by a matrix multiplication. The former is

1. more accurate (as we see above), and
2. faster.

Summary: don't use matrix inverse; use solution of linear equations instead.

6.6.2 Identity Matrix

The matrix that "does nothing" in matrix multiplication, like zero "does nothing" in addition of scalars and one "does nothing" in multiplication of scalars is called the *identity matrix*.

If A is a square matrix and I is the identity matrix of the same dimension, then $AI = IA = A$. And B is the matrix inverse of A if and only if $AB = BA = I$.

The identity matrix is created by the R function `diag`.

```
diag(3)
```

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

This is a highly unusual usage of this function. Usually it is used to extract or assign to the diagonal of a matrix (see the help page). (This is also an example where perhaps two different function names should have been used, but it's too late to fix that now. This remark is here because Ansu Chatterjee said this in casual conversation.)

6.7 R Package Matrix

This is a "recommended package" meaning it comes with every R installation. It is especially useful in dealing with *sparse* matrices (those with most components equal to zero). It can be vastly more efficient dealing with sparse matrices than the rest of R is.

But all of that is beyond the scope of this course.

6.8 More Higher-Order Functions: `apply` and `sweep`

6.8.1 Applying Functions that Return Scalars

The R function `apply` is a higher-order function that applies another function to the rows or columns of a matrix (or array, more on that later).

```
fred
```

```
##       car truck
## red     1     4
## white   2     5
## blue    3     6
```

```
apply(fred, 1, mean)
```

```
##   red white  blue
##   2.5   3.5   4.5
```

```
apply(fred, 2, mean)
```

```
##   car truck
##     2     5
```

```
rowMeans(fred)
```

```
##   red white  blue
##   2.5   3.5   4.5
```

```
colMeans(fred)
```

```
##   car truck
##     2     5
```

Since there are the convenience functions `rowMeans` and `colMeans`, we don't need to use `apply` with `mean` as its third argument. There are also the convenience functions `rowSums` and `colSums`, so we don't need to use `apply` with `sum` as its third argument. But that's it for the convenience functions. For every other function we want to apply we need `apply`.

```
fred
```

```
##       car truck
## red     1     4
## white   2     5
## blue    3     6
```

```
apply(fred, 1, max)
```

```
##   red white  blue
##     4     5     6
```

```
apply(fred, 2, max)
```

```
##   car truck
##     3     6
```

And

```
apply(fred, 1, function(x) quantile(x, 0.75))
```

```
##   red white  blue
##  3.25  4.25  5.25
```

```
apply(fred, 2, function(x) quantile(x, 0.75))
```

```
##   car truck
##   2.5   5.5
```

```
apply(fred, 1, quantile, probs = 0.75)
```

```
##   red white  blue
##  3.25  4.25  5.25
```

```
apply(fred, 2, quantile, probs = 0.75)
```

```
##   car truck
##   2.5   5.5
```

For those who have not seen the term *quantile* before, we explain that it is the same as a *percentile* except without the percent. The 0.75 *quantile* of a numeric vector is the same as the 75th *percentile* of a vector. In statistics and probability theory we never use percents because they yield much more confusion than clarity. Most of the public cannot do arithmetic on percentages correctly. So we always insist that probabilities are numbers between zero and one and never percentages. Similarly in probability and statistics we always use quantiles and never percentiles. Even the term *95% confidence interval* has a competing term that does not use percents: a confidence interval with *0.95 coverage probability*.

This example shows us two ways to apply a function that has more than one argument. We can use an anonymous function, or we can use the `...` argument of apply to pass other arguments to the function being applied.

6.8.2 Applying Functions that Return Vectors

Apply can also be used when the result of the function being applied is not a scalar.

```
apply(fred, 1, quantile, probs = (1:3)/4)
```

```
##      red white blue
## 25% 1.75  2.75 3.75
## 50% 2.50  3.50 4.50
## 75% 3.25  4.25 5.25
```

```
apply(fred, 2, quantile, probs = (1:3)/4)
```

```
##     car truck
## 25% 1.5   4.5
## 50% 2.0   5.0
## 75% 2.5   5.5
```

Sometimes this doesn't do what you think it should. The help for `apply` says

> If each call to `FUN` returns a vector of length `n`, then `apply` returns an array of dimension `c(n, dim(X)[MARGIN])` if `n > 1`. If `n` equals `1`, `apply` returns a vector if `MARGIN` has length 1 and an array of dimension `dim(X)[MARGIN]` otherwise. If `n` is `0`, the result has length 0 but not necessarily the `correct` dimension.

So the dimension of the result that has length `n` is alway the first one (the row dimension if the result is a matrix). Sometimes this is the transpose of you think it *ought* to do. But it is what it is documented to do.

6.8.3 Sweep Following Apply

The R function `sweep` can be used with `apply` to further process a matrix. Suppose we want to subtract means to make rows of a matrix have mean zero. That is done by

```
moo <- apply(fred, 1, mean)
fred.mean.free.rows <- sweep(fred, 1, moo)
rowMeans(fred.mean.free.rows)
```

```
##   red white  blue
##     0     0     0
```

But `sweep` is a higher-order function because its fourth optional argument is the function to be used instead of subtraction (see the help page).

6.9 More on Quantiles

The qth quantile of a continuous random vector X is any number x (which need not be unique) such that

$$\Pr(X \le x) = q$$

The qth quantile of a discrete random vector X is any number x (which need not be unique) such that

$$\Pr(X \le x) \ge q$$
$$\Pr(X \ge x) \ge 1 - q$$

But the R function `quantile` doesn't use this definition (or doesn't *only* use this definition) because it is thinking of the vector whose quantiles it is calculating as a random sample from a population and it is trying to *estimate* the quantile of the *population*. And it has nine (!?) different ways to do that. The theoretically correct definition given above corresponds to `type = 1` or `type = 2`. More TIMTOWTDI. The default is (currently) `type = 7`.

6.10 What is a Matrix *Really?*

A matrix is just an atomic vector that has certain *attributes*.

```
mode(fred)
```

```
## [1] "numeric"
```

```
class(fred)
```

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

```
is.atomic(fred)
```

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

```
is.vector(fred)
```

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

```
is.matrix(fred)
```

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

```
attributes(fred)
```

```
## $dim
## [1] 3 2
##
## $dimnames
## $dimnames[[1]]
## [1] "red"   "white" "blue"
##
## $dimnames[[2]]
## [1] "car"   "truck"
```

I'm surprised. I thought `is.vector(fred)` was going to say `TRUE`. Shows what I know.

We can make a matrix without using any of the matrix-oriented functions.

```
fred.too <- 1:6
attributes(fred.too) <- list(dim = 3:2,
    dimnames = list(c("red", "white", "blue"), c("car", "truck")))
identical(fred, fred.too)
```

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

TIMTOWTDI. But you shouldn't do the latter in real life. Too tricky. It might confuse people looking at your code.

7 Arrays

Arrays are just like matrices but with more dimensions.

```
alice <- array(1:24, dim = 2:4)
alice
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
```

```
## [2,]   14   16   18
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
```

This is a little hard to read, and it gets even harder when there are more dimensions.

You can think of this array as being a three-dimensional table, and what R has printed as being four "slices" of the table, each of which is two-dimensional, hence a matrix. The printout is trying to say the same thing as

```
alice[ , , 1]
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
alice[ , , 2]
```

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

```
alice[ , , 3]
```

```
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
```

```
alice[ , , 4]
```

```
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
```

As was the case with matrices, R stuffs vectors into an array in FORTRAN order (first index changes the fastest, second index next fastest, and so forth) rather than C/C++ order.

According to their help pages, `apply`, `sweep`, and `outer` work on arrays. But I can never remember exactly how they work and always have to look up in the help or try little experiments to figure out what they do.

```
apply(alice, 1, mean)
```

```
## [1] 12 13
```

```
apply(alice, 1:2, mean)
```

```
##      [,1] [,2] [,3]
## [1,]   10   12   14
## [2,]   11   13   15
```

So the second argument of `apply`, which is named `MARGIN`, gives the indices of the array which are to be indices of the result.

In our first example (`MARGIN = 1`) the result is the average over the components of the array having specified values of the first index. It is the same as

```
mean(alice[1, , ])
```

```
## [1] 12
```

```
mean(alice[2, , ])
```

```
## [1] 13
```

In our second example (`MARGIN = 1:2`) the result is the average over the components of the array having specified values of the first and second indices. It is the same as

```
mean(alice[1, 1, ])
```

```
## [1] 10
```

```
mean(alice[2, 1, ])
```

```
## [1] 11
```

```
mean(alice[1, 2, ])
```

```
## [1] 12
```

```
mean(alice[2, 2, ])
```

```
## [1] 13
```

```
mean(alice[1, 3, ])
```

```
## [1] 14
```

```
mean(alice[2, 3, ])
```

```
## [1] 15
```

except that it stuffs these six numbers into a two by three matrix corresponding to the dimensions for these indices in the original matrix.

I hope it is clear from this example that

- `apply` is really powerful, and
- `apply` is tricky (use with care).

When you use `sweep` and `apply` together, you generally want their `MARGIN` arguments to agree.

```
foo <- sweep(alice, 1:2, apply(alice, 1:2, mean))
foo
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]   -9   -9   -9
## [2,]   -9   -9   -9
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   -3   -3   -3
## [2,]   -3   -3   -3
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]    3    3    3
## [2,]    3    3    3
```

```
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]    9    9    9
## [2,]    9    9    9
```

```
apply(foo, 1:2, mean)
```

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

That is all I have to tell you about `apply`, `sweep`, and `outer` applied to arrays. They are not easy to use. They are just at the edge of the level of expertise we want you to develop in this course. In thirty years of using R and its predecessor S, I have used `apply` on arrays a lot, `sweep` on arrays rarely (less than once a year), and `outer` on arrays never (but now that I know it works on arrays, I may try to use it in the future).

Unlike matrices, arrays do not have a counterpart in mathematics. but do have counterparts in FORTRAN, C, C++, and other computer languages. Every computer language I have ever heard of has arrays. They are a basic storage structure. But they are not used anywhere near as much in R as in other languages, because they don't work well with R models (which is another handout).

Like matrices, arrays are really just atomic vectors with a `dim` attribute that says what their dimensions are. They may also have a `dimnames` attribute that gives labels for their whatevers (I don't know what to call them; a matrix has rows and columns; an array has rows, columns, and whatever comes after rows and columns, but I don't think there is a word for that).

8 Data Frames

8.1 Introduction

A data frame is something that prints out like a matrix, but it isn't a matrix. It is best thought of as a collection of named vectors all of the same length. It was originally designed for data for R models (which is another handout). Typically the response and predictor vectors for a model must be the same length. Data frames are designed to hold such vectors.

8.2 Package `datasets`

There are many data frames that come with R and are used for examples.

```
foo <- eapply(as.environment("package:datasets"), is.data.frame)
foo <- unlist(foo)
length(foo)
```

```
## [1] 104
```

```
sum(foo)
```

```
## [1] 44
```

This says there are 104 datasets in the R package named `datasets`, which is always already attached when R starts (so there is no need to say `library(datasets)` before trying to access one), and 44 of these are objects of class `"data.frame"`.

The R function `eapply` used here applies R functions, here `is.data.frame` to all the objects stored in an R environment, which we haven't covered yet, so we shall say no more here about that.

8.3 Toy Data Frame

Here is an example data frame.

```
nam <- c("alice", "bob", "clara", "dan", "eve", "fred")
dat <- data.frame(name = nam, gpa = pmin(4.0, rnorm(length(nam), 3.0, 0.5)),
    class = sample(c("freshman", "sophomore", "junior", "senior"),
        length(nam), replace = TRUE))
dat
```

```
##    name      gpa     class
## 1 alice 2.784765 sophomore
## 2   bob 2.871365 sophomore
## 3 clara 2.118418 sophomore
## 4   dan 3.230049 sophomore
## 5   eve 2.680003    senior
## 6  fred 3.227725    junior
```

As we already said, data frames are especially designed to work with R functions that fit models and many such have an argument, usually named `data` that takes a data frame in which to find data. For example

```
summary(aov(gpa ~ class, data = dat))
```

```
##             Df Sum Sq Mean Sq F value Pr(>F)
## class        2 0.2048  0.1024   0.476  0.661
## Residuals    3 0.6453  0.2151
```

This does the analog of a two-sample, independent sample, t-test assuming equality of variances for the difference of means of two populations, when two is replaced by four (because there are four classes, except that how many classes we have in these data is random, so sometimes three classes).

But modeling is the subject of another handout, so back to data frames.

8.4 Indexing (Extracting, Subscripting)

Data frames are as we have already said, and will say some more, are really lists. But they look like matrices when printed. The reason is that data frames are not *just* lists.

```
class(dat)
```

```
## [1] "data.frame"
```

They know they are data frames, and R generic functions like print and the extractor functions (`"["`, `"[["`, `"[<-"`, `"[[<-"`, `"$"`) know how to deal especially with data frames. So the list extractors work on data frames as if they were lists and the matrix extractors work on data frames as if they were matrices.

```
dat$name
```

```
## [1] "alice" "bob"   "clara" "dan"   "eve"   "fred"
```

```
dat[["name"]]
```

```
## [1] "alice" "bob"   "clara" "dan"   "eve"   "fred"
```

```
dat[[1]]
```

```
## [1] "alice" "bob"   "clara" "dan"   "eve"   "fred"
```

```
dat[1, 1]
```

```
## [1] "alice"
```

```
dat[1, -1]
```

```
##        gpa     class
## 1 2.784765 sophomore
```

```
dat[dat$gpa > 3, ]
```

```
##   name      gpa     class
## 4  dan 3.230049 sophomore
## 6 fred 3.227725    junior
```

Great! But what was that nonsense about "levels".

8.5 Factors

For a bit we stop talking about data frames and start talking about factors, which is R's name for the kind of object it uses for categorical data. Where other people say "categorical data" (as in the title of STAT 5421 here at U of M) R says `factor`.

```
class(dat$class)
```

```
## [1] "character"
```

If we look up `??factor` we find that R has the following functions all documented on one help page

```
factor
ordered
is.factor
is.ordered
as.factor
as.ordered
addNA
```

that deal with factors. The categories of the categorical variable are called the "levels" of the factor in R-speak. Although the `See Also` section of `?factor` doesn't tell you this, the R functions

```
levels
nlevels
```

tell you what the levels of a factor are and how many of them there are. The `See Also` section of `?levels` tells us that there are also functions

```
relevel
reorder
```

(which I have never heard of before) that are useful in futzing with the levels of a factor.

The R function `ordered` makes ordered factors, which is R-speak for what other people call ordered categorical data. They are also beyond the scope of this course (should be mentioned at least briefly in STAT 5421).

8.5.1 What is a Factor *Really?*

```
dat.strip <- dat$class
attributes(dat.strip) <- NULL
dat.strip
```

```
## [1] "sophomore" "sophomore" "sophomore" "sophomore" "senior"    "junior"
```

```
typeof(dat.strip)
```

```
## [1] "character"
```

So an R factor object is *really* an integer vector. All the rest of the information is in the attributes

```
attributes(dat$class)
```

```
## NULL
```

The way one gets a class whose levels are character strings back to its original form is to run it through `as.character`.

```
as.character(dat$class)
```

```
## [1] "sophomore" "sophomore" "sophomore" "sophomore" "senior"    "junior"
```

You can never be really sure what a factor is going to be treated as (the integer vector it really is or the character vector you think it is) So running a factor through `as.character` is a good idea whenever you want the character vector that is how you think of the factor. (R might do this by some sort of default coercion, but it also might not.)

8.5.2 Avoiding Factors

Factors are much less essential than they used to be. If a variable of type character is used as a predictor variable in an R model, then it is treated as if it had been converted to a `factor` variable. Moreover, R has had this behavior with a warning since version 2.4.0 (released October 2006) and the warning was removed in R 3.0.0 (released April 2013). Also character vectors used to take up a lot more space than the corresponding factors, but since version 2.6.0 (released October 2007) R has only one copy of every unique character string in any character vectors. All character vectors containing that string just point to it (in the C code implementing R).

Since version 2.4.0 (released October 2006) the R functions

```
as.data.frame
data.frame
expand.grid
rbind
read.table
read.csv
read.csv2
read.delim
read.delim2
```

have an argument `stringsAsFactors` which says whether the default behavior of converting character vectors to factors should be done when producing a data frame. The default for this is

```
args(data.frame)
```

```
## function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,
##     fix.empty.names = TRUE, stringsAsFactors = default.stringsAsFactors())
## NULL
```

and the documentation `?data.frame` and `?options` explains that to never have default conversion of strings to factors one does

```
options(stringsAsFactors = FALSE)
```

(except we did not actually do this for this document, that is, the preceding code chunk had `eval=FALSE` as an option). And if you want to have that whenever you start up R, you can put it in your `.Rprofile` file (`?Rprofile` explains that).

If this seems an incredible mess, it is. This shows the consequences of bad design decisions. They are very hard to partially undo and and impossible to completely undo (while maintaining backward compatibility so you do not break working code and have users hate you). Lest anyone think your humble author is being snooty here, I admit to having made bad design decisions myself.

As of R version 4.0.0 the default for

```
options("stringsAsFactors")
```

```
## $stringsAsFactors
## [1] FALSE
```

changes from `TRUE` to `FALSE`. As explained above, for the most part this is unnoticed by users but can be a pain. I spent two days debugging a problem that arose because I was using R-4.0.0 and a user I had given my code to was using an earlier version of R so we had different defaults of `stringsAsFactors`. But now that almost every R user is using R version 4.0.0 or later, this is less of an issue.

8.5.3 Needing Factors

But the whole notion of factors is not completely unnecessary. If someone insists on factor levels that are numeric or logical, then `data.frame` and friends cannot know that this variable is intended to be a factor. And one has to tell it that explicitly. If `fred` is a data frame containing a variable named `sally`, then there are a number of ways (TIMTOWTDI)

```
fred <- fred.save <- fred2 <- data.frame(sally = 1:10, herman = 11:20)
fred2$sally <- as.factor(fred2$sally)
fred3 <- transform(fred, sally = factor(sally))
fred4 <- within(fred, sally <- factor(sally))
identical(fred, fred.save) # have not clobbered original
```

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

```
identical(fred, fred2) # supposed to be FALSE
```

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

```
identical(fred2, fred3) # supposed to be TRUE
```

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

```
identical(fred3, fred4) # supposed to be TRUE
```

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

8.6 What is a Data Frame, *Really?*

```
dat.strip <- dat
attributes(dat.strip) <- NULL
dat.strip
```

```
## [[1]]
## [1] "alice" "bob"   "clara" "dan"   "eve"   "fred"
##
## [[2]]
## [1] 2.784765 2.871365 2.118418 3.230049 2.680003 3.227725
##
## [[3]]
## [1] "sophomore" "sophomore" "sophomore" "sophomore" "senior"    "junior"
```

```
typeof(dat.strip)
```

```
## [1] "list"
```

```
attributes(dat)
```

```
## $names
## [1] "name"  "gpa"   "class"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3 4 5 6
```

A data frame is *really* just a list having certain attributes and certain properties that are enforced by the function `data.frame` and friends.

8.7 Reading a Data Frame

By "reading" we mean doing I/O, getting stuff from outside of R into R and in this case, specifically as a data frame.

The standard methods for doing this are

```
read.DIF
read.fortran
read.fwf
read.table
read.csv
read.csv2
read.delim
read.delim2
```

which were mentioned above. There are even more methods in the recommended package `foreign` for reading even crazier formats (such as those from proprietary statistical packages).

These read from a file or from a "readable text-mode connection" which may be a URL (internet address). We saw the latter in action during the first quiz.

Here's another example

```
foo <- read.table("http://www.stat.umn.edu/geyer/3701/data/array-ex1.txt",
    header = TRUE)
head(foo)
```

```
##   x color      y
## 1 1   red 24.894
## 2 1  blue 12.323
```

```
## 3 1 green 16.645
## 4 2   red 25.231
## 5 2  blue 12.119
## 6 2 green 16.654
```

Looking at that URL shows what the data format is. It is a whitespace-separated plain text file with variable names in the first row and elements of the variables in the rest of the rows, each column being one variable (name and data).

Many users are in the (bad IMHO) habit of using spreadsheets (Microsoft Excel, for example) as data editors. So then how does one get the data out of the spreadsheet and into R? Spreadsheets can write CSV (comma separated values) format, and R can read that. (To make a CSV file, select "Save As" on the File menu, and then choose the output format to be CSV.)

```
foo.too <- read.csv("http://www.stat.umn.edu/geyer/3701/data/array-ex1.csv")
identical(foo, foo.too)
```

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