This section continues where we left off last week, introducing you to increasingly complex matrix manipulation in R. Next week we will be working with real data again, so don't despair!

But first, as promised, some answers to questions from last week:

## Questions from last week's section

**Supplementary materials for each section**: Last week we had some trouble accessing the dataset I used in section. Sorry about that. In this and all future sections, supplementary materials will be posted in the section notes folder.

**Row names**: Dataframe rows also have names in R. The preferred function to get and set those names is `row.names()`, which is the row analogue of `names()`. We can see how this function works using the dataset we loaded last week. Since there are 74 rows, I only show the first five, but you get the point.

```
row.names(data)[1:5]
row.names(data) <- 3:76
row.names(data)[1:5]
row.names(data)[1:2]  <- c("myrow1","myrow2")
row.names(data)[1:5]
```

```
[1] "1" "2" "3" "4" "5"
[1] "3" "4" "5" "6" "7"
[1] "myrow1" "myrow2" "5"      "6"      "7"
```

As you can see, you can set as many or as few of the row names as you like.

**Row means**: Here's how to calculate the mean of a row in R. This example isn't very meaningful, since we're averaging `price`, `mpg`, and `weight`, but it works as a proof of concept.

```
data$price[2] = NA
weirdmeans1 <- data.frame(Means=rowMeans(data))
weirdmeans2 <- data.frame(Means=rowMeans(data, na.rm=TRUE))
cbind(head(weirdmeans1),head(weirdmeans2))
```

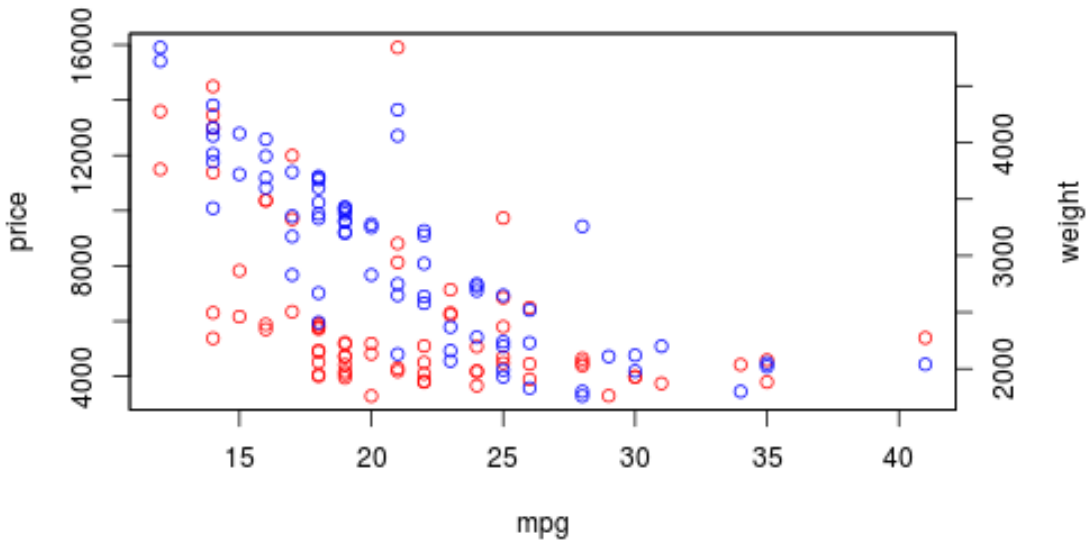
```
          Means    Means
myrow1 2350.333 2350.333
myrow2       NA 1683.500
5      2153.667 2153.667
6      2695.333 2695.333
7      3974.000 3974.000
8      3158.667 3158.667
```

There are a couple wrinkles in here. First, to make it more interesting, I set one of prices equal to NA, which is R-speak for "not available". The first time we take `rowMeans`, R basically tells us that

it can't take the mean of a set of numbers when one of them is not available[1]. But by passing the optional argument `na.rm=TRUE` we can tell `R` to recklessly take the mean of the remaining numbers. The second wrinkle is less interesting but also useful — I just used `cbind()` to bind the two means columns together in a data frame for ease of display.

**Double Y Axes**:
Overlaying two variables on the same axis is a convenient way to display the two relationships at the same time. In this case, let's say we want to visually compare how price and weight change with miles per gallon. We can do this by telling `R` to draw a second scatterplot directly on top of the first one.

```
par(mar=c(5.1,4.1,4.1,5.1)) # just so we can fit the label for the second y axis
plot(data$mpg,data$price,col="red",ylab="price",xlab="mpg")
par(new=TRUE)
plot(data$mpg,data$weight,col="blue",xaxt="n",yaxt="n",xlab="",ylab="")
axis(4)
mtext("weight",side=4,line=3)
```




There are a lot of new commands here[2]. `par()` lets us pre-set parameters for what we're about to graph. First, we set `mar` to increase the size of the right-hand side margin in order to fit the label for the second axis. Second, we set `new=TRUE`, which allows us to draw our second scatterplot on the same plotting device[3].
That's it for last week. And now for something completely different, matrix commands!

[1] In high school, I had a friend named Jared who got his license a full year before the rest of us. Apparently he was told that if he had more than one person in the car he could get arrested, so if we ever wanted to get anywhere he had to shuttle us. `R` is kind of acting like Jared here.

[2] Credit to Professor Rob J Hyndman for this code. Original available here: `http://robjhyndman.com/hyndsight/r-graph-with-two-y-axes/`.

[3] No, the boolean choice here doesn't make sense to me either.

## Creating and testing matrices

There are a variety of data objects in R, including numbers, vectors, matrices, strings, and dataframes. We will mainly be working with vectors and matrices, which are quick to create and manipulate in R. The `matrix` function will create a matrix, according to the supplied arguments.

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

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

```
(B <- matrix(1:6, ncol=3, byrow=TRUE))
```

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

For convenience, we use `->` to assign the matrices to the variables `A` and `B` for use in subsequent manipulations. The = operator also assigns values, with a slightly different behavior[4] . The `ncol` option specifies the number of columns for the output matrix; and the default behavior of `matrix` is to cycle through by column. To cycle through by rows we set the optional argument `byrow=TRUE`.

Suppose we wanted to check to see if the first matrix was equal to the transpose of the second. This is clearly the case — we can see that it is. But when we're working with larger matrices it will be convenient to have a way to do this programmatically. The `==` comparison operator will yield `TRUE` or `FALSE`:

```
A == t(B)
```

```
     [,1] [,2]
[1,] TRUE TRUE
[2,] TRUE TRUE
[3,] TRUE TRUE
```

Note that `t()` will return the transpose of the supplied matrix. Each element is checked individually, and each is identical in matrix $\mathbf{A}$ and $\mathbf{B}'$. To check the truthiness of the statement that all elements are identical, we need only to employ the `all` function:

```
all(A == t(B))
```

```
[1] TRUE
```

Keeping track of your matrix dimensions is a Good Idea™. That's where the `dim()` command comes in handy:

```
dim(A)
dim(B)
```

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

With the dimensions of our matrices in mind, we'll move on to matrix operations.

[4] It is also common practice to use the = operator for function arguments.

## Matrix operations

Matrix muliplication in R is bound to %*%, whereas scalar multiplication is bound to *. Consider the product $\mathbf{BA}$:

```
B %*% A
```

```
     [,1] [,2]
[1,]   14   32
[2,]   32   77
```

The dimensions have to line up properly for matrix multiplication to be appropriately applied, otherwise R returns an error, as is the case with the product $\mathbf{BA}'$:

```
B %*% t(A)
```

```
Error in B %*% t(A) : non-conformable arguments
```

If scalar multiplication is applied to matrices of exactly the same dimensions, then the result is element-wise multiplication. This type of operation is sometimes called the Hadamard product, denoted $\mathbf{B} \circ \mathbf{A}'$:

```
B * t(A)
```

```
     [,1] [,2] [,3]
[1,]    1    4    9
[2,]   16   25   36
```

Suppose we want to scale all elements by a factor of two. This is similar, we just multiply the matrix by a scalar using the regular * operator.

```
A * 2
```

```
     [,1] [,2]
[1,]    2    8
[2,]    4   10
[3,]    6   12
```

Consider a more complicated operation, whereby each column of a matrix is multiplied element-wise by another, fixed column. Here, each column of a particular matrix is multiplied in-place by a fixed column of residuals. Let $\mathbf{e}$ be a vector defined as an increasing sequence of length three:

```
e <- matrix(1:3)
```

Note first that the default sequence in R is a column vector, and not a row vector. We would like to `apply` a function to each column of $\mathbf{A}$, specifically a function that multiplies each column in-place by $\mathbf{e}$. We must supply a 2 to ensure that the function is applied to the second dimension (columns) of $\mathbf{A}$:

```
apply(A, 2, function(x) {x * e})
```

```
     [,1] [,2]
[1,]    1    4
[2,]    4   10
[3,]    9   18
```

The function that is applied is anonymous, but it could also be bound to a variable – just as a matrix is bound to a variable:

```
whoop <- function(x) {x * e}
apply(A, 2, whoop)
```

```
     [,1] [,2]
[1,]    1    4
[2,]    4   10
[3,]    9   18
```

We will often need to define an identity matrix of dimension $n$, or $\mathbf{I}_n$. This is quick using `diag()`:

```
I <- diag(5)
```

As you know, $\mathbf{I}_n = \mathbf{I}_n^{-1}$. We can verify this with the `solve()` command, which will return the inverse of a square matrix[5].

```
all(solve(I) == I)
```

```
[1] TRUE
```

There are many ways to calculate the trace of $\mathbf{I}_5$. One method has been bundled into a function, called `tr()`, that is included in a package called `psych` which is not included in the base distribution of `R`. We will need to grab and call the library to have access to the function, installing it with the command `install.packages("psych")`. For this, you'll need an internet connection.

```
library(psych)
tr(I)
```

```
[1] 5
```

We can get a list of all the object currently available in memory with the `ls()` function, which is useful as the assignments begin to accumulate:

```
ls()
```

```
[1] "A"           "B"           "data"        "e"           "I"
[6] "weirdmeans1" "weirdmeans2" "whoop"
```

Note that the objects we did not explicitly assign, such the transpose of $\mathbf{B}$, `t(B)`, or the trace of $\mathbf{I}$, `tr(I)`, are created on the fly and not stored in memory.

When paired with the `rm()` function, we can use `ls()` to delete all of the objects in memory. This is similar to the command `clear` in Stata.

[5] Note that we can't use `solve()` on $\mathbf{A}$ or $\mathbf{B}$ since neither are square.

```
rm(list = ls())
```

What's going on here? `list` is actually the name of an argument built in to the `rm()` command. The default behavior of `rm` is to accept character strings; we could have alternatively specified `rm("A","B","data","e","I","weirdmeans1","weirdmeans2","whoop")` and the outcome would have been the same. But by passing it a list of all of the objects in memory, we are telling `rm()` to clear everything, not just the variables we name.

Next week we will leave the training wheels behind and dig into an example with real data. Now that we have all of the tools, our new best friend $(X'X)^{-1}X'y$ may even make an appearance. Hopefully you all have started work on the first problem set and are starting to feel at least somewhat comfortable in `R`.

## Linear algebra puzzles

1. Define vectors $\mathbf{x} = [1 \;\; 2 \;\; 3]'$, $\mathbf{y} = [2 \;\; 3 \;\; 4]'$, and $\mathbf{z} = [3 \;\; 5 \;\; 7]$. Define $\mathbf{W} = [\mathbf{x} \;\; \mathbf{y} \;\; \mathbf{z}]$. Calculate $\mathbf{W}^{-1}$. If you cannot take the inverse, explain why not and adjust $\mathbf{W}$ so that you *can* take the inverse. *Hint*: the `solve()` function will return the inverse of the supplied matrices.

2. Show, somehow, that $(\mathbf{X}')^{-1} = (\mathbf{X}^{-1})'$.

3. Generate a $3 \times 3$ matrix $\mathbf{X}$, where each element is drawn from a standard normal distribution. Let $\mathbf{A} = \mathbf{I}_3 - \frac{1}{3}\mathbf{B}$ be a demeaning matrix, with ß a $3 \times 3$ matrix of ones. First show that $\mathbf{A}$ is idempotent and symmetric. Next show that each row of the matrix $\mathbf{XA}$ is the deviation of each row in $\mathbf{X}$ from its mean. Finally, show that $(\mathbf{XA})(\mathbf{XA})' = \mathbf{XAX'}$, first through algebra and then `R` code.

4. Demonstrate from random matrices that $(\mathbf{XYZ})^{-1} = \mathbf{Z}^{-1}\mathbf{Y}^{-1}\mathbf{X}^{-1}$.

5. Let $\mathbf{X}$ and $\mathbf{Y}$ be square $20 \times 20$ matrices. Show that $tr(\mathbf{X} + \mathbf{Y}) = tr(\mathbf{X}) + tr(\mathbf{Y})$.

6. Generate a diagonal matrix $\mathbf{X}$, where each element on the diagnonal is drawn from $U[10, 20]$. Now generate a matrix $\mathbf{B}$ s.t. $\mathbf{X} = \mathbf{BB'}$. *Hint*: There is a method in `R` that makes this easy. Does the fact that you can generate $\mathbf{B}$ tell you anything about $\mathbf{X}$?

7. Demonstrate that for any scalar $c$ and any square matrix $\mathbf{X}$ of dimension $n$ that $\det(c\mathbf{X}) = c^n \det(\mathbf{X})$.

8. Demonstrate that for an $m \times m$ matrix $\mathbf{A}$ and a $p \times p$ matrix $\mathbf{B}$ that $\det(\mathbf{A} \otimes \mathbf{B}) = \det(\mathbf{A})^p \det(\mathbf{B})^m$. *Hint*: Note that $\otimes$ indicates the Kronecker product[6]. Google the appropriate `R` function.

[6]The Kronecker product is a useful mathemagical tool for econometricians, allowing us to more easily describe block-diagonal matricees for use in panel data settings. I wouldn't lose sleep over it, though.