Introduction

The goal of this assignment is to familiarize you with creating and manipulating objects in R. The assignment will also introduce you very handy base R functions that will make both of these tasks more efficient (less typing by hand!).

Concepts covered in this assignment include:

1. Creating objects in R
2. Indexing values in objects
3. Applying operators to objects

Resources/Files needed for this assignment:

- None

****** Note: For the questions below, there may be multiple correct answers. The solutions presented are just one option. *****

1. Run the R code given below and consider the vector and the two matrices:

```
a <- seq(2,12,by=2); a
A <- matrix(a,4,2); A
B <- matrix(a,3,4); B
```

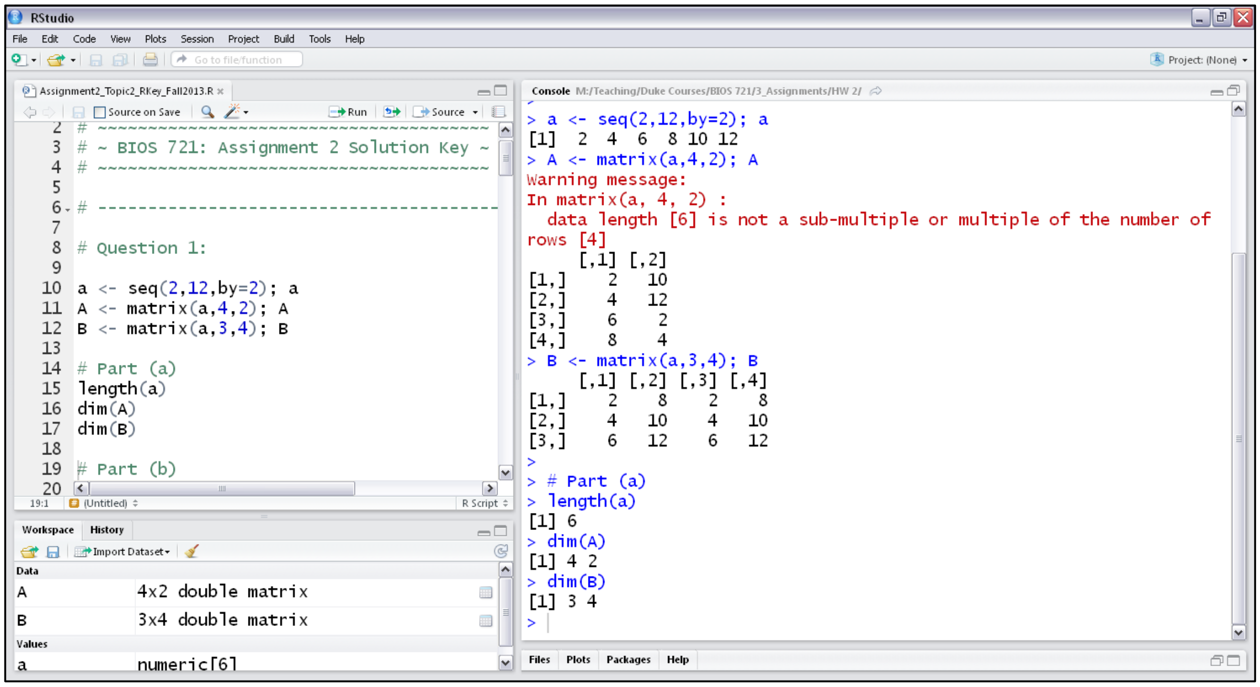
   a. What is the length of the vector? How does the length of the vector compare to the dimensions of the two matrices? How did R handle any discrepancies? Did it differ between the two matrices – if so, how?

      R Code:

```
length(a)
dim(A)
dim(B)
```

      The length of vector a is 6 (i.e. it has 6 elements). The dimensions of matrix A are 4 by 2 (i.e. it has 8 elements). The dimensions of matrix B are 3 by 4 (i.e. it has 12 elements). Thus, both matrices have MORE elements that the vector used to create them. R reconciles this by REPEATING the elements of the vector when creating the matrices. However, the number of elements in vector a is a multiple of the number of elements in matrix B, but not matrix A. As a result, R produces a warning when creating matrix A and repeat of vector a is truncated when creating this matrix (but is not when creating the matrix B).

R Screenshot:



```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~ BIOS 721: Assignment 2 Solution Key ~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# -----------------------------------------

# Question 1:

a <- seq(2,12,by=2); a
A <- matrix(a,4,2); A
B <- matrix(a,3,4); B

# Part (a)
length(a)
dim(A)
dim(B)

# Part (b)
```

```
Data
A            4x2 double matrix
B            3x4 double matrix
Values
a            numeric[6]
```

```
> a <- seq(2,12,by=2); a
[1]  2  4  6  8 10 12
> A <- matrix(a,4,2); A
Warning message:
In matrix(a, 4, 2) :
  data length [6] is not a sub-multiple or multiple of the number of
rows [4]
     [,1] [,2]
[1,]    2   10
[2,]    4   12
[3,]    6    2
[4,]    8    4
> B <- matrix(a,3,4); B
     [,1] [,2] [,3] [,4]
[1,]    2    8    2    8
[2,]    4   10    4   10
[3,]    6   12    6   12
>
> # Part (a)
> length(a)
[1] 6
> dim(A)
[1] 4 2
> dim(B)
[1] 3 4
>
```

b. Using the information obtained in part (a) provide the R code to create the matrix given below using the following approaches:

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

i. Using the c() and matrix() functions (i.e. the long way).

R Code:

```
i <- c(-2:2,-2:2,-2:2)
I <- matrix(i,3,5,byrow=TRUE); I
```

ii. Using the cbind() or the rbind() function.

R Code:

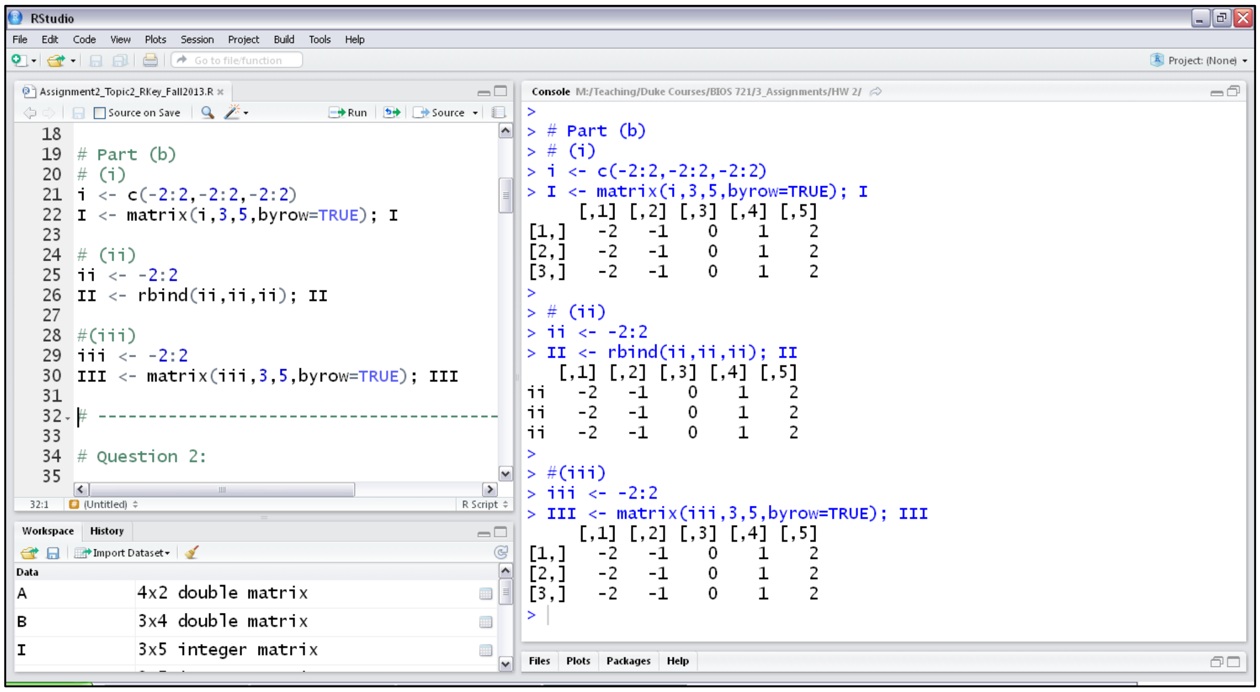
```
ii <- -2:2
II <- rbind(ii,ii,ii); II
```

iii. Using vectorized programming with the matrix() function (i.e. using information obtained in part (a)).

R Code:

```
iii <- -2:2
III <- matrix(iii,3,5,byrow=TRUE); III
```

R Screenshot:



```
# Part (b)
# (i)
i <- c(-2:2,-2:2,-2:2)
I <- matrix(i,3,5,byrow=TRUE); I

# (ii)
ii <- -2:2
II <- rbind(ii,ii,ii); II

#(iii)
iii <- -2:2
III <- matrix(iii,3,5,byrow=TRUE); III

# ----------------------------------------

# Question 2:
```

```
>
> # Part (b)
> # (i)
> i <- c(-2:2,-2:2,-2:2)
> I <- matrix(i,3,5,byrow=TRUE); I
     [,1] [,2] [,3] [,4] [,5]
[1,]   -2   -1    0    1    2
[2,]   -2   -1    0    1    2
[3,]   -2   -1    0    1    2
>
> # (ii)
> ii <- -2:2
> II <- rbind(ii,ii,ii); II
   [,1] [,2] [,3] [,4] [,5]
ii   -2   -1    0    1    2
ii   -2   -1    0    1    2
ii   -2   -1    0    1    2
>
> #(iii)
> iii <- -2:2
> III <- matrix(iii,3,5,byrow=TRUE); III
     [,1] [,2] [,3] [,4] [,5]
[1,]   -2   -1    0    1    2
[2,]   -2   -1    0    1    2
[3,]   -2   -1    0    1    2
>
```

Data	
A	4x2 double matrix
B	3x4 double matrix
I	3x5 integer matrix

2. Run the R code given below and consider the two matrices:

```
A <- matrix(2,2,2); A
B <- matrix(1:4,2,2); B
```

a. Submit the following commands to R: A*B and A%*%B. Explain the difference between the two commands (i.e. * vs. %*%).

R Code:
```
A*B
A%*%B
```

The * command performs ELEMENT-WISE matrix multiplication (i.e. multiplies the (i,j)th element in A by the (i,j)th element in B). However, the %*% command performs TRUE/ALGEBRAIC matrix multiplication (i.e. finds the cross product of each row in the first matrix with each column of the second matrix).

b. Should the output of A*B equal the output of B*A? Explain.

R Code:
```
A*B
B*A
```

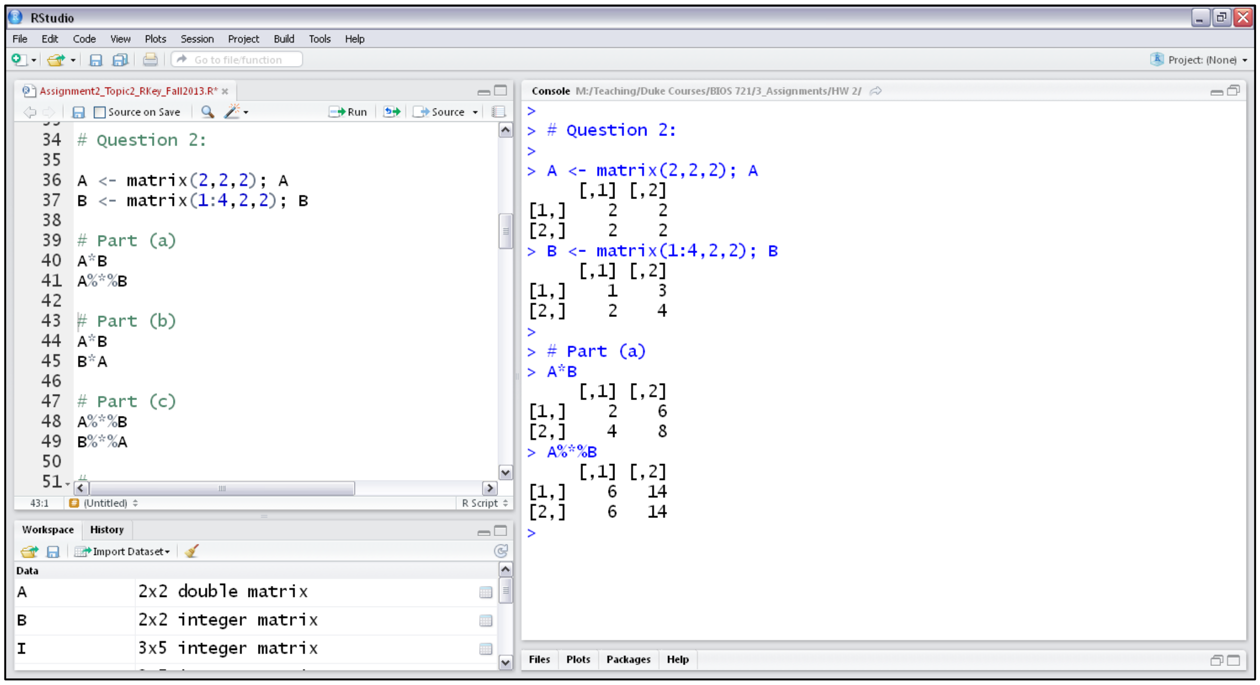
Yes – because element-wise multiplication is commutative.

c. Should the output of A%*%B equal the output of B%*%A? Explain.

R Code:
```
A%*%B
B%*%A
```

Not necessarily – in general, true matrix multiplication is not commutative. (This is only guaranteed to be true if the matrices were inverses of each other.)

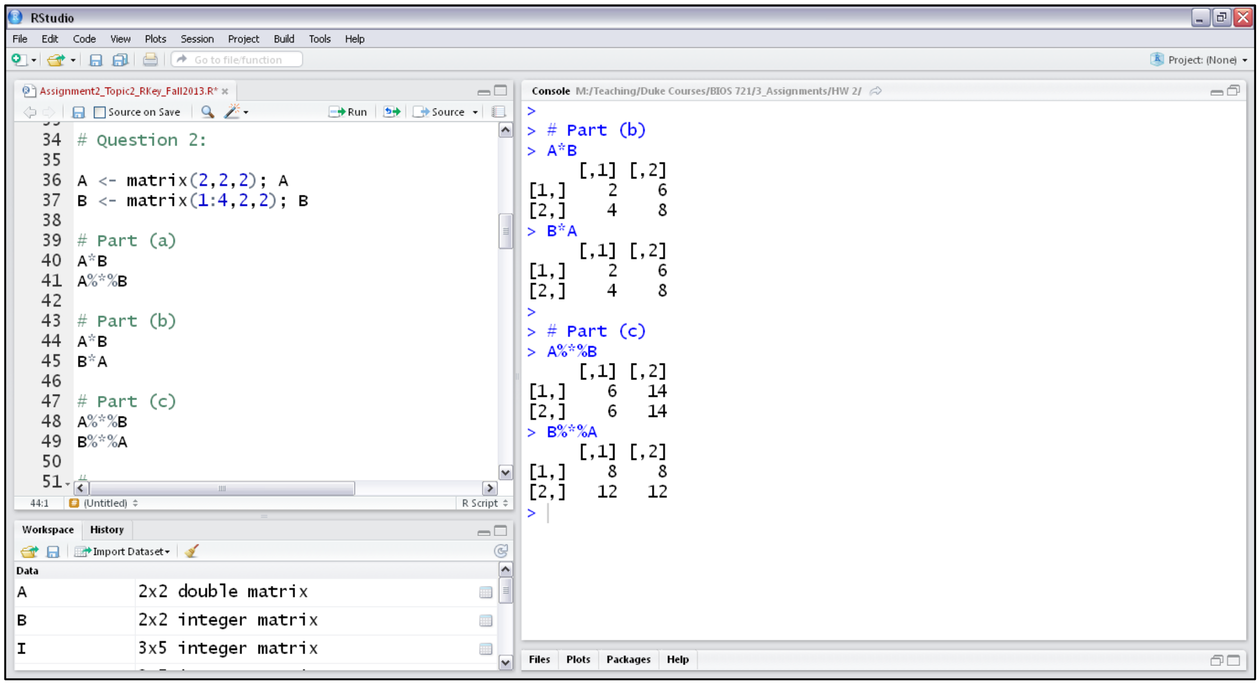
R Screenshots:


RStudio
File Edit Code View Plots Session Project Build Tools Help
Go to file/function
Project: (None)
Assignment2_Topic2_RKey_Fall2013.R*
Source on Save
Run
Source
43:1
(Untitled)
R Script
Workspace History
Import Dataset
Console M:/Teaching/Duke Courses/BIOS 721/3_Assignments/HW 2/
Files Plots Packages Help

```
# Question 2:

A <- matrix(2,2,2); A
B <- matrix(1:4,2,2); B

# Part (a)
A*B
A%*%B

# Part (b)
A*B
B*A

# Part (c)
A%*%B
B%*%A
```

```
>
> # Question 2:
>
> A <- matrix(2,2,2); A
     [,1] [,2]
[1,]    2    2
[2,]    2    2
> B <- matrix(1:4,2,2); B
     [,1] [,2]
[1,]    1    3
[2,]    2    4
>
> # Part (a)
> A*B
     [,1] [,2]
[1,]    2    6
[2,]    4    8
> A%*%B
     [,1] [,2]
[1,]    6   14
[2,]    6   14
>
```

```
Data
A          2x2 double matrix
B          2x2 integer matrix
I          3x5 integer matrix
```


RStudio
File Edit Code View Plots Session Project Build Tools Help
Go to file/function
Project: (None)
Assignment2_Topic2_RKey_Fall2013.R*
Source on Save
Run
Source
44:1
(Untitled)
R Script
Workspace History
Import Dataset
Console M:/Teaching/Duke Courses/BIOS 721/3_Assignments/HW 2/
Files Plots Packages Help

```
# Question 2:

A <- matrix(2,2,2); A
B <- matrix(1:4,2,2); B

# Part (a)
A*B
A%*%B

# Part (b)
A*B
B*A

# Part (c)
A%*%B
B%*%A
```

```
>
> # Part (b)
> A*B
     [,1] [,2]
[1,]    2    6
[2,]    4    8
> B*A
     [,1] [,2]
[1,]    2    6
[2,]    4    8
>
> # Part (c)
> A%*%B
     [,1] [,2]
[1,]    6   14
[2,]    6   14
> B%*%A
     [,1] [,2]
[1,]    8    8
[2,]   12   12
>
```

```
Data
A          2x2 double matrix
B          2x2 integer matrix
I          3x5 integer matrix
```

3. Consider the data set given below. It lists 5 students' scores on three tests and on a final exam in a course:

```
         ID  Test1  Test2  Test3  Final
Student1      20     23     18     48
Student2      16     15     18     36
Student3      25     20     22     40
Student4      14     19     18     42
Student5      10     15     14     30
```

   a. Create a data object that contains the same information as the data set shown above. Call the data object grades. Make sure that the mode of each data element is appropriate. Print the grades object.

      R Code:

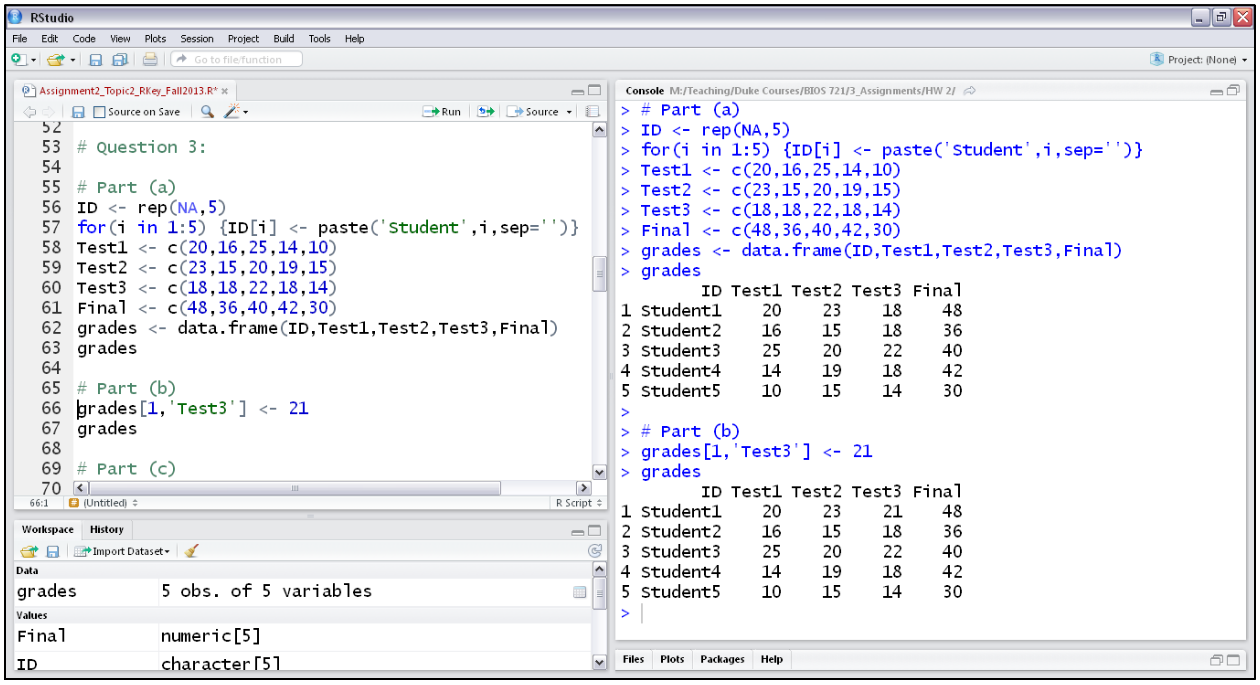
```
ID <- rep(NA,5)
for(i in 1:5) {ID[i] <- paste('Student',i,sep='')}
Test1 <- c(20,16,25,14,10)
Test2 <- c(23,15,20,19,15)
Test3 <- c(18,18,22,18,14)
Final <- c(48,36,40,42,30)
grades <- data.frame(ID,Test1,Test2,Test3,Final)
grades
```

   b. Suppose that Student 1 actually received a 21 on Test 3. Update the grades data object to reflect the correct score. Print the updated grades object.

      R Code:

```
grades[1,'Test3'] <- 21
grades
```

R Screenshot:


RStudio
File Edit Code View Plots Session Project Build Tools Help
Assignment2_Topic2_RKey_Fall2013.R*
Source on Save
Run
Source
Console M:/Teaching/Duke Courses/BIOS 721/3_Assignments/HW 2/
Workspace History
Import Dataset
Files Plots Packages Help
Project: (None)
R Script

```r
# Question 3:

# Part (a)
ID <- rep(NA,5)
for(i in 1:5) {ID[i] <- paste('Student',i,sep='')}
Test1 <- c(20,16,25,14,10)
Test2 <- c(23,15,20,19,15)
Test3 <- c(18,18,22,18,14)
Final <- c(48,36,40,42,30)
grades <- data.frame(ID,Test1,Test2,Test3,Final)
grades

# Part (b)
grades[1,'Test3'] <- 21
grades

# Part (c)
```

```
> # Part (a)
> ID <- rep(NA,5)
> for(i in 1:5) {ID[i] <- paste('Student',i,sep='')}
> Test1 <- c(20,16,25,14,10)
> Test2 <- c(23,15,20,19,15)
> Test3 <- c(18,18,22,18,14)
> Final <- c(48,36,40,42,30)
> grades <- data.frame(ID,Test1,Test2,Test3,Final)
> grades
        ID Test1 Test2 Test3 Final
1 Student1    20    23    18    48
2 Student2    16    15    18    36
3 Student3    25    20    22    40
4 Student4    14    19    18    42
5 Student5    10    15    14    30
>
> # Part (b)
> grades[1,'Test3'] <- 21
> grades
        ID Test1 Test2 Test3 Final
1 Student1    20    23    21    48
2 Student2    16    15    18    36
3 Student3    25    20    22    40
4 Student4    14    19    18    42
5 Student5    10    15    14    30
>
```

```
Data
grades          5 obs. of 5 variables
Values
Final           numeric[5]
ID              character[5]
```

c. Print all the grades for ...

i. Test 3

R Code:
```r
grades[,4]
grades[,'Test3']
```

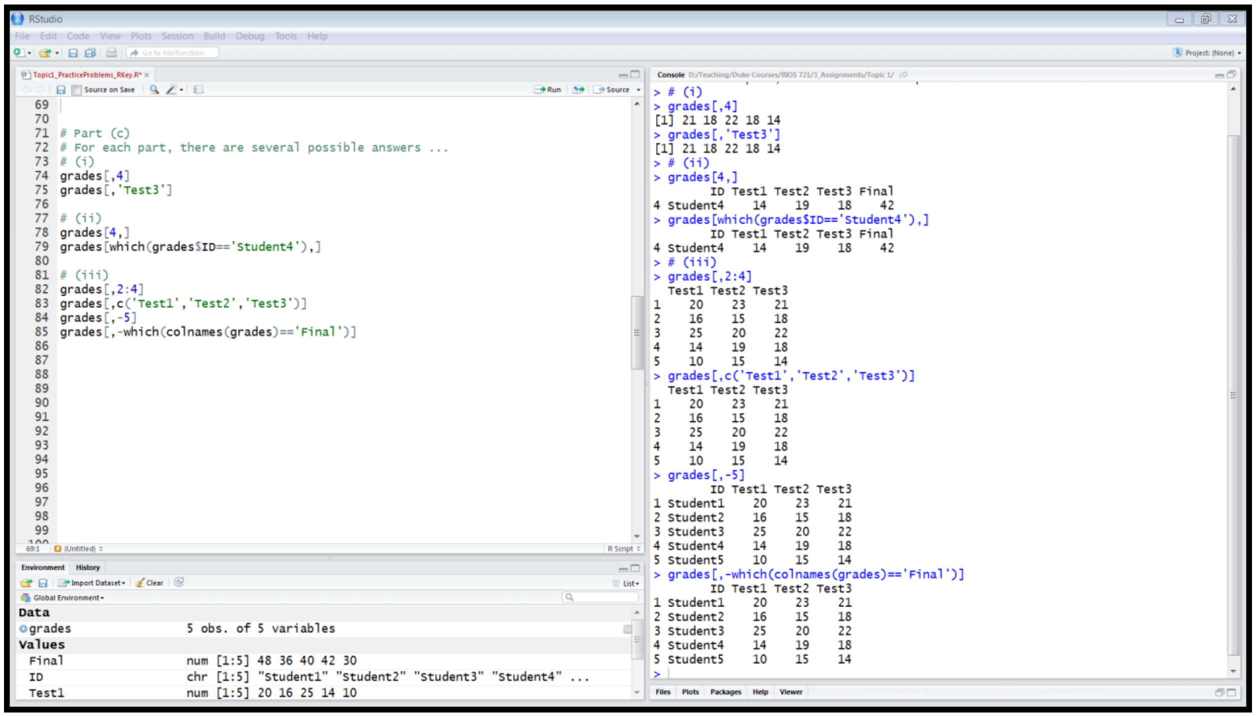
ii. Student 4

R Code:
```r
grades[4,]
grades[which(grades$ID=='Student4'),]
```

iii. Test 1 – 3 but not for the final exam.

R Code:
```r
grades[,2:4]
grades[,c('Test1','Test2','Test3')]
grades[,-5]
grades[,-which(colnames(grades)=='Final')]
```

R Screenshot:



```
# Part (c)
# For each part, there are several possible answers ...
# (i)
grades[,4]
grades[,'Test3']

# (ii)
grades[4,]
grades[which(grades$ID=='Student4'),]

# (iii)
grades[,2:4]
grades[,c('Test1','Test2','Test3')]
grades[,-5]
grades[,-which(colnames(grades)=='Final')]
```

```
Data
grades                5 obs. of 5 variables
Values
 Final                num [1:5] 48 36 40 42 30
 ID                   chr [1:5] "Student1" "Student2" "Student3" "Student4" ...
 Test1                num [1:5] 20 16 25 14 10
```

```
> # (i)
> grades[,4]
[1] 21 18 22 18 14
> grades[,'Test3']
[1] 21 18 22 18 14
> # (ii)
> grades[4,]
        ID Test1 Test2 Test3 Final
4 Student4    14    19    18    42
> grades[which(grades$ID=='Student4'),]
        ID Test1 Test2 Test3 Final
4 Student4    14    19    18    42
> # (iii)
> grades[,2:4]
  Test1 Test2 Test3
1    20    23    21
2    16    15    18
3    25    20    22
4    14    19    18
5    10    15    14
> grades[,c('Test1','Test2','Test3')]
  Test1 Test2 Test3
1    20    23    21
2    16    15    18
3    25    20    22
4    14    19    18
5    10    15    14
> grades[,-5]
        ID Test1 Test2 Test3
1 Student1    20    23    21
2 Student2    16    15    18
3 Student3    25    20    22
4 Student4    14    19    18
5 Student5    10    15    14
> grades[,-which(colnames(grades)=='Final')]
        ID Test1 Test2 Test3
1 Student1    20    23    21
2 Student2    16    15    18
3 Student3    25    20    22
4 Student4    14    19    18
5 Student5    10    15    14
>
```

d. Print the test scores for students who scored above 16 on Test 1.

R Code:
```
grades[which(grades$Test1>16),]
```

e. Find the mean score for each student across all tests (including the final).

R Code:
```
apply(grades[,2:5],1,mean)
```

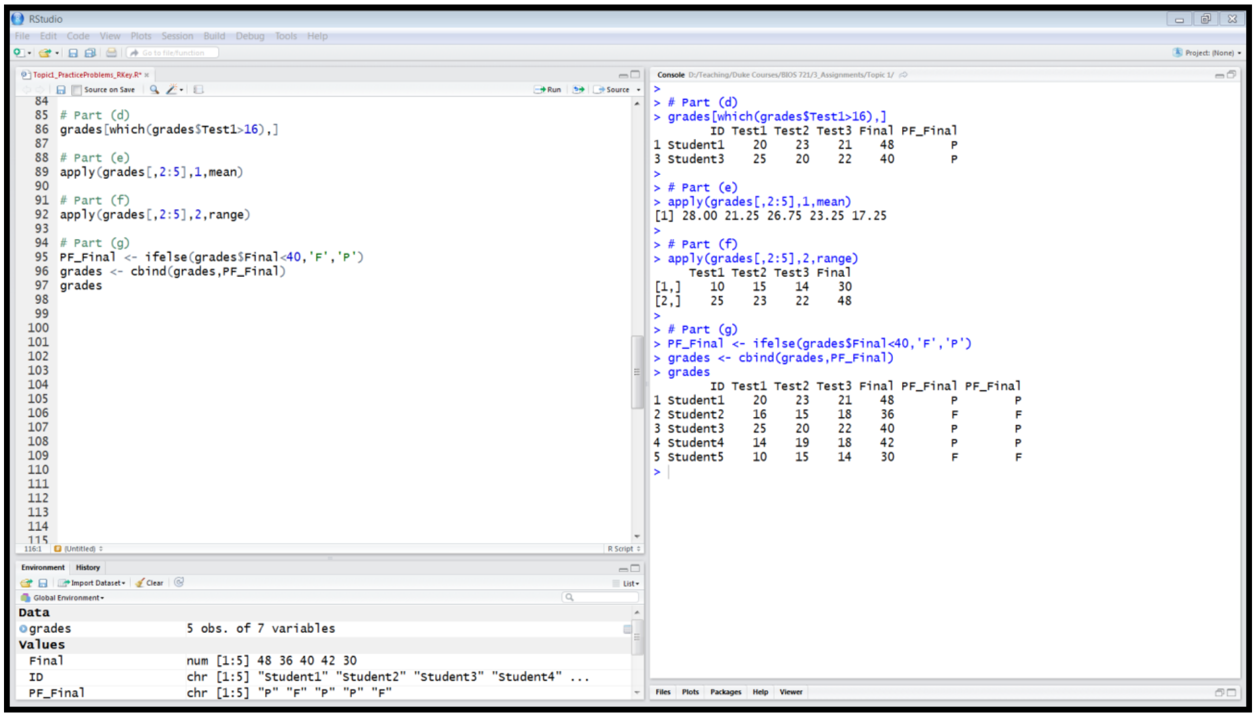
f. Find the range of scores for each test (including the final).

R Code:
```
apply(grades[,2:5],2,range)
```

g. Add a variable that denotes whether a student passed or failed the final. A fail is defined as a score below 40. Indicate a pass with a P and a fail with an F. Name the new variable PF_Final. Print the updated grades object.

R Code:
```
PF_Final <- ifelse(grades$Final<40,'F','P')
grades <- cbind(grades,PF_Final)
grades
```

R Screenshot:



```
# Part (d)
grades[which(grades$Test1>16),]

# Part (e)
apply(grades[,2:5],1,mean)

# Part (f)
apply(grades[,2:5],2,range)

# Part (g)
PF_Final <- ifelse(grades$Final<40,'F','P')
grades <- cbind(grades,PF_Final)
grades
```

Console D:/Teaching/Duke Courses/BIOS 721/3_Assignments/Topic 1/

```
>
> # Part (d)
> grades[which(grades$Test1>16),]
        ID Test1 Test2 Test3 Final PF_Final
1 Student1    20    23    21    48        P
3 Student3    25    20    22    40        P
>
> # Part (e)
> apply(grades[,2:5],1,mean)
[1] 28.00 21.25 26.75 23.25 17.25
>
> # Part (f)
> apply(grades[,2:5],2,range)
     Test1 Test2 Test3 Final
[1,]    10    15    14    30
[2,]    25    23    22    48
>
> # Part (g)
> PF_Final <- ifelse(grades$Final<40,'F','P')
> grades <- cbind(grades,PF_Final)
> grades
        ID Test1 Test2 Test3 Final PF_Final PF_Final
1 Student1    20    23    21    48        P        P
2 Student2    16    15    18    36        F        F
3 Student3    25    20    22    40        P        P
4 Student4    14    19    18    42        P        P
5 Student5    10    15    14    30        F        F
>
```

```
Data
grades       5 obs. of 7 variables
Values
Final        num [1:5] 48 36 40 42 30
ID           chr [1:5] "Student1" "Student2" "Student3" "Student4" ...
PF_Final     chr [1:5] "P" "F" "P" "P" "F"
```

4. Run the R code given below and consider the vector and the two matrices:

```
a <- c(-2,4,-3,0); a
A <- matrix(c(2,5,3,0,9,-1,-4,-7,6),3,3); A
B <- matrix(c(2,5,-3,10,9,-1,9,-7,6),3,3); B
```

a. Apply the length() function to the vector and the first matrix. Explain what the function returns for each data object.

R Code:

```
length(a)
length(A)
```

For each data object, the length() function returns the NUMBER of ELEMENTS in each data object. The length() function 'vectorizes' the matrix and then applies the action of the function.

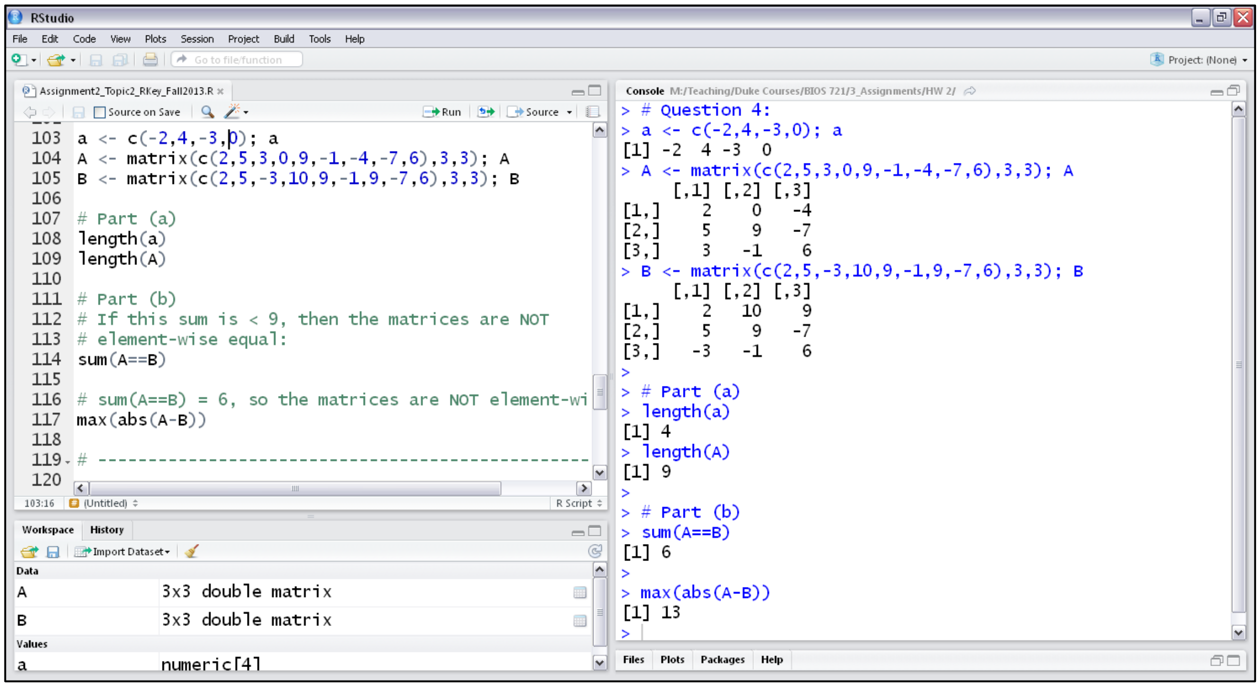
b. Using the information obtained in part (a), determine if the two matrices are element-wise equal. If not, find the largest absolute element-wise difference between the two matrices.

R Code:

```
# If this sum is < 9, then the matrices are NOT
# element-wise equal:
sum(A==B)

# sum(A==B) = 6, so the matrices are NOT
# element-wise equal
max(abs(A-B))
```

To determine if the matrices are element-wise equal, first use the == command to determine if each element of the matrices are equal. This will return a 3 by 3 matrix of logicals (TRUE/FALSE). Then can use the sum() function to vectorize the this logical matrix and count the number of TRUEs. If that count is 9, then the matrices are element-wise equal. If not, we can use the same logic (but different functions) to find the largest absolute element-wise difference between the two matrices.

R Screenshot:


RStudio
File Edit Code View Plots Session Project Build Tools Help
Go to file/function
Project: (None)
Assignment2_Topic2_RKey_Fall2013.R
Source on Save
Run
Source
103:16
(Untitled)
R Script
Workspace
History
Import Dataset
Console M:/Teaching/Duke Courses/BIOS 721/3_Assignments/HW 2/
Files
Plots
Packages
Help

```
a <- c(-2,4,-3,0); a
A <- matrix(c(2,5,3,0,9,-1,-4,-7,6),3,3); A
B <- matrix(c(2,5,-3,10,9,-1,9,-7,6),3,3); B

# Part (a)
length(a)
length(A)

# Part (b)
# If this sum is < 9, then the matrices are NOT
# element-wise equal:
sum(A==B)

# sum(A==B) = 6, so the matrices are NOT element-wi
max(abs(A-B))

# ------------------------------------------------
```

Data

A	3x3 double matrix
B	3x3 double matrix

Values

a	numeric[4]

```
> # Question 4:
> a <- c(-2,4,-3,0); a
[1] -2  4 -3  0
> A <- matrix(c(2,5,3,0,9,-1,-4,-7,6),3,3); A
     [,1] [,2] [,3]
[1,]    2    0   -4
[2,]    5    9   -7
[3,]    3   -1    6
> B <- matrix(c(2,5,-3,10,9,-1,9,-7,6),3,3); B
     [,1] [,2] [,3]
[1,]    2   10    9
[2,]    5    9   -7
[3,]   -3   -1    6
>
> # Part (a)
> length(a)
[1] 4
> length(A)
[1] 9
>
> # Part (b)
> sum(A==B)
[1] 6
>
> max(abs(A-B))
[1] 13
>
```

5. The diag() function is very flexible and can be a useful tool with working the matrices. Run the R code below:

```
a <- 1:5; a
diag(a)

A <- matrix(c(-5,9,-14,1,3,-12,6,-6,13),3,3); A
diag(A)
```

   a. Explain the difference between applying the diag() function to vector vs. applying the diag() function to a matrix.

      When the diag() function is applied to a vector, it creates a square matrix with the vector as the diagonal elements. When the diag() function is applied to a square matrix, it creates a vector containing the diagonal elements of the matrix.

b. Using the information obtained in part (a) …

i. Find the smallest on diagonal element of matrix A.

R Code:

```
min(diag(A))
```

First, pull of the diagonal elements using the diag() function. Then, apply the min() function to that vector.

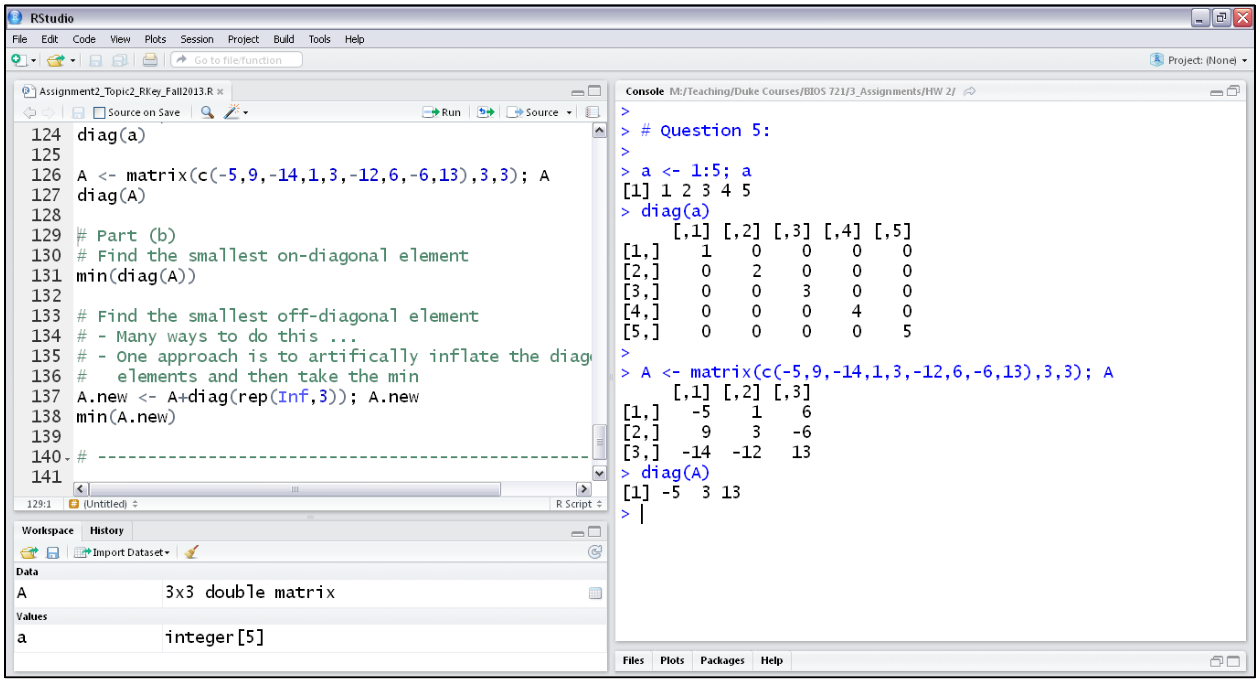
ii. Find the smallest off diagonal element of matrix A.

R Code:

```
A.new <- A+diag(rep(Inf,3)); A.new
min(A.new)
```

First, artificially inflate the magnitude of the diagonal elements by creating a diagonal matrix of large values (here infinity) and adding it to the original matrix (creates A.new in the code above). Then, find the min of all elements in the new matrix.

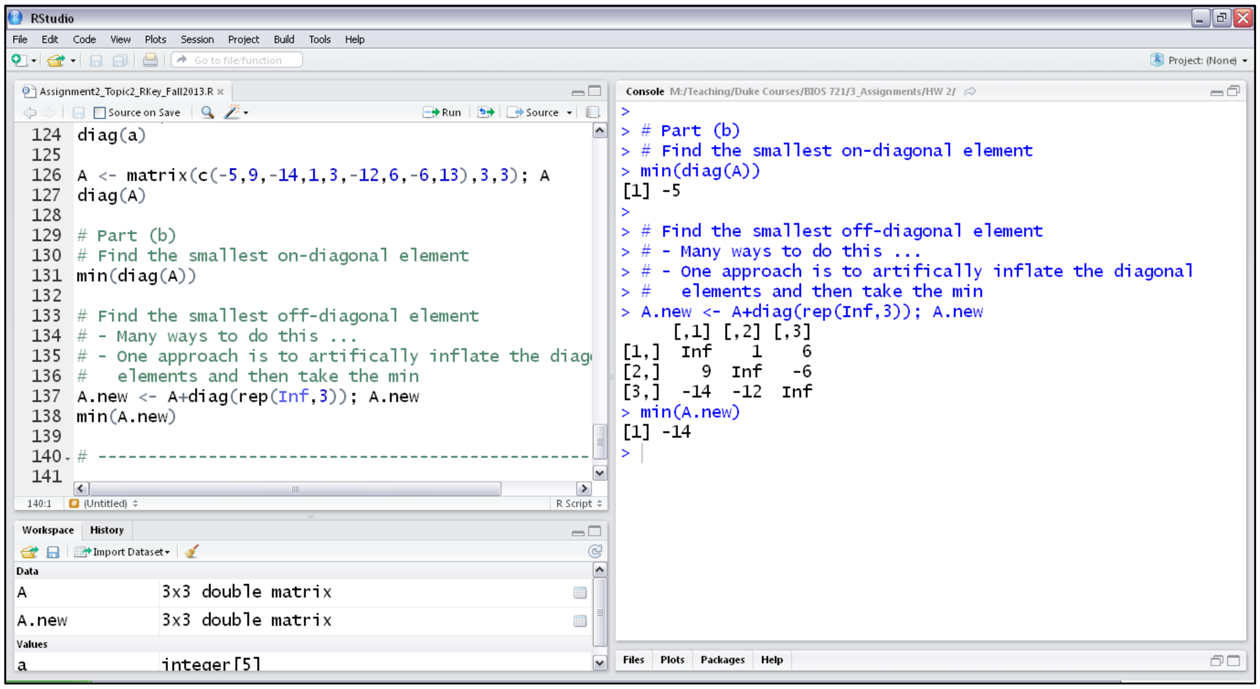
R Screenshots:



```
diag(a)

A <- matrix(c(-5,9,-14,1,3,-12,6,-6,13),3,3); A
diag(A)

# Part (b)
# Find the smallest on-diagonal element
min(diag(A))

# Find the smallest off-diagonal element
# - Many ways to do this ...
# - One approach is to artifically inflate the diag
#   elements and then take the min
A.new <- A+diag(rep(Inf,3)); A.new
min(A.new)

# -------------------------------------------------
```

```
>
> # Question 5:
>
> a <- 1:5; a
[1] 1 2 3 4 5
> diag(a)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    2    0    0    0
[3,]    0    0    3    0    0
[4,]    0    0    0    4    0
[5,]    0    0    0    0    5
>
> A <- matrix(c(-5,9,-14,1,3,-12,6,-6,13),3,3); A
     [,1] [,2] [,3]
[1,]   -5    1    6
[2,]    9    3   -6
[3,]  -14  -12   13
> diag(A)
[1] -5  3 13
> 
```

Data	
A	3x3 double matrix
Values	
a	integer[5]


RStudio
File Edit Code View Plots Session Project Build Tools Help
Go to file/function
Project: (None)
Assignment2_Topic2_RKey_Fall2013.R
Source on Save
Run
Source
140:1
(Untitled)
R Script
Console M:/Teaching/Duke Courses/BIOS 721/3_Assignments/HW 2/
Workspace
History
Import Dataset
Files
Plots
Packages
Help

```
diag(a)

A <- matrix(c(-5,9,-14,1,3,-12,6,-6,13),3,3); A
diag(A)

# Part (b)
# Find the smallest on-diagonal element
min(diag(A))

# Find the smallest off-diagonal element
# - Many ways to do this ...
# - One approach is to artifically inflate the diag
#   elements and then take the min
A.new <- A+diag(rep(Inf,3)); A.new
min(A.new)

# --------------------------------------------------
```

```
>
> # Part (b)
> # Find the smallest on-diagonal element
> min(diag(A))
[1] -5
>
> # Find the smallest off-diagonal element
> # - Many ways to do this ...
> # - One approach is to artifically inflate the diagonal
> #   elements and then take the min
> A.new <- A+diag(rep(Inf,3)); A.new
     [,1] [,2] [,3]
[1,]  Inf    1    6
[2,]    9  Inf   -6
[3,]  -14  -12  Inf
> min(A.new)
[1] -14
>
```

Data

A	3x3 double matrix
A.new	3x3 double matrix

Values

a	integer[5]

6. Consider the vector v given in the R code below:

```
> v <- 1:10; v
[1] 1  2  3  4  5  6  7  8  9 10
```

R is a vector-based or “vectorized” language. As such, vectors are very flexible data objects in R. We have seen an example of this in the lecture slides where vectors were re-shaped to create matrices and arrays. This works because R views vectors as “dimension-less” objects; instead vectors have length. To see an example of this, run the R code given below:

```
length(v)
dim(v)
```

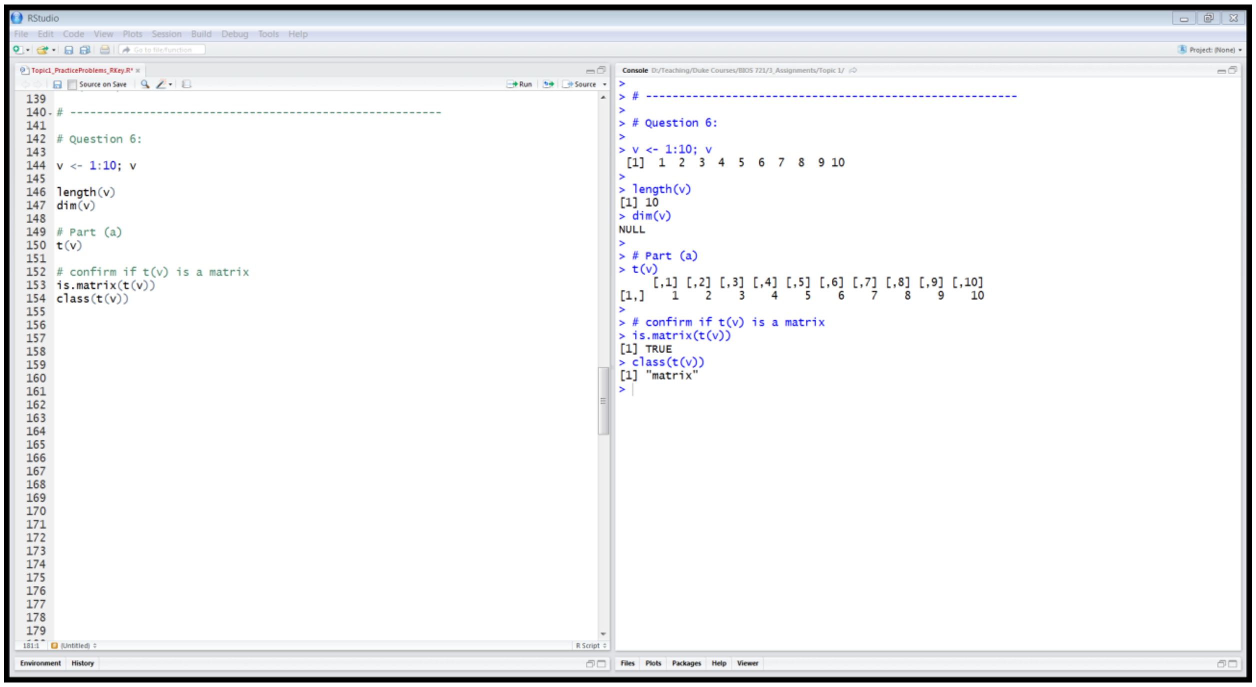
a. The function t() finds the transpose of a matrix. What happens when this function is applied to the data object called v? Does it fail to execute? If so, why? Does it execute? If so, does it return a matrix? If so, why does this make sense? Find an R function that confirms whether the output of t(v) is actually viewed as a matrix by R.
   i. Hint: Google is your friend!

R Code:

```
t(v)

# confirm if t(v) is a matrix
is.matrix(t(v))
class(t(v))
```

The R code t(v) executes – which seems surprising at first because t() is a matrix operation and v is not stored as a matrix. When applying the t() function to v, R internally coerces v to a matrix and then transposes it. As such, the output of t(v) is matrix. This can be confirmed using either the is.matrix() function which returns a logical for whether or not the data object is a matrix or the class() function which returns a character string listing object type.

R Screenshot:



```
# ---------------------------------------------------------

# Question 6:

v <- 1:10; v

length(v)
dim(v)

# Part (a)
t(v)

# confirm if t(v) is a matrix
is.matrix(t(v))
class(t(v))
```

```
>
> # ---------------------------------------------------------
>
> # Question 6:
>
> v <- 1:10; v
 [1]  1  2  3  4  5  6  7  8  9 10
>
> length(v)
[1] 10
> dim(v)
NULL
>
> # Part (a)
> t(v)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9    10
>
> # confirm if t(v) is a matrix
> is.matrix(t(v))
[1] TRUE
> class(t(v))
[1] "matrix"
>
```

b. What type of object does the R code t(v)%*%v return – a scalar, vector, or matrix? (Test yourself – try to determine BEFORE running the code.) Confirm your answer. What is the name given to this object in linear algebra?

R Code:
```
# Part (b)
t(v)%*%v

# determine type of object
class(t(v)%*%v)
```

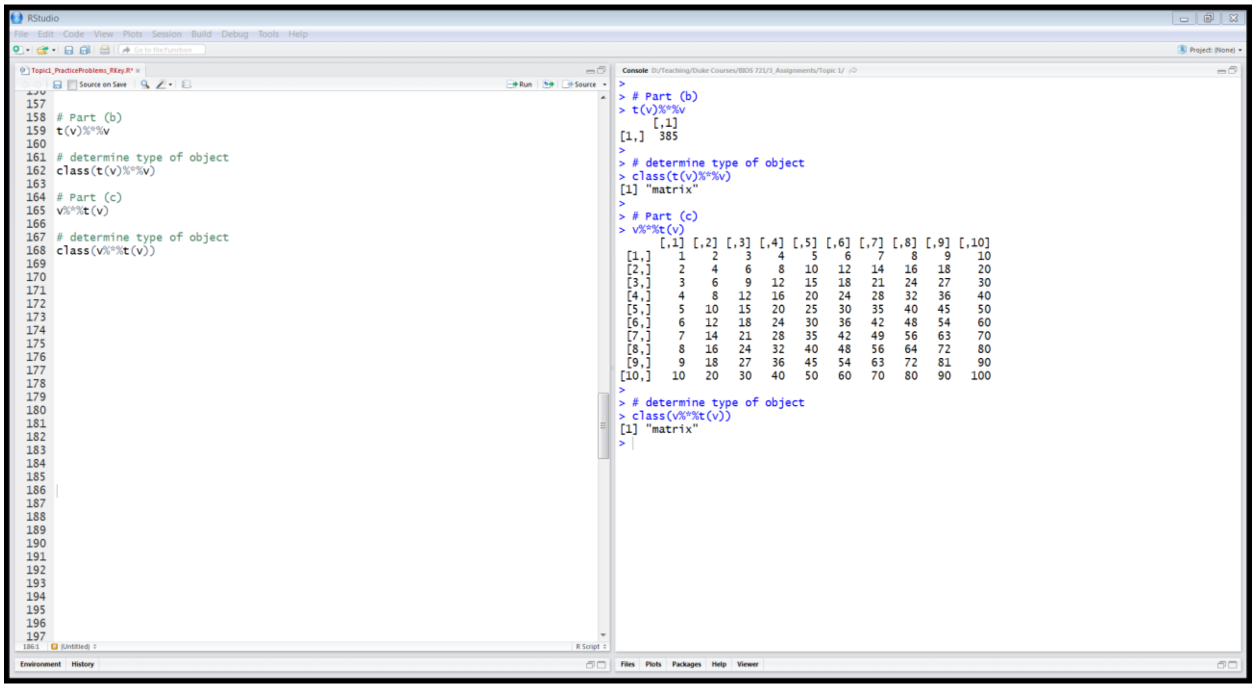
This R code returns a matrix because t() and %*% are matrix operations in R. This quantity is usually referred to as the inner product in linear algebra and is considered to be a scalar.

c. What type of object does the R code v%*%t(v) return – a scalar, vector, or matrix? (Test yourself – try to determine BEFORE running the code.) What is the name given to this object in linear algebra?

R Code:
```
# Part (c)
v%*%t(v)

# determine type of object
class(v%*%t(v))
```

This R code returns a matrix because t() and %*% are matrix operations in R. This quantity is usually referred to as the outer product in linear algebra.

R Screenshot:



```
# Part (b)
t(v)%*%v

# determine type of object
class(t(v)%*%v)

# Part (c)
v%*%t(v)

# determine type of object
class(v%*%t(v))
```

```
>
> # Part (b)
> t(v)%*%v
     [,1]
[1,]  385
>
> # determine type of object
> class(t(v)%*%v)
[1] "matrix"
>
> # Part (c)
> v%*%t(v)
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    1    2    3    4    5    6    7    8    9    10
 [2,]    2    4    6    8   10   12   14   16   18    20
 [3,]    3    6    9   12   15   18   21   24   27    30
 [4,]    4    8   12   16   20   24   28   32   36    40
 [5,]    5   10   15   20   25   30   35   40   45    50
 [6,]    6   12   18   24   30   36   42   48   54    60
 [7,]    7   14   21   28   35   42   49   56   63    70
 [8,]    8   16   24   32   40   48   56   64   72    80
 [9,]    9   18   27   36   45   54   63   72   81    90
[10,]   10   20   30   40   50   60   70   80   90   100
>
> # determine type of object
> class(v%*%t(v))
[1] "matrix"
>
```

d. Using vectors and matrix algebra, find the mean of the elements in the data object called v. Make sure your code is automated and would work properly for ANY vector v. Confirm your answer using a function available in R.
    i. Hint: You may have to create additional vectors to answer this question.

R Code:

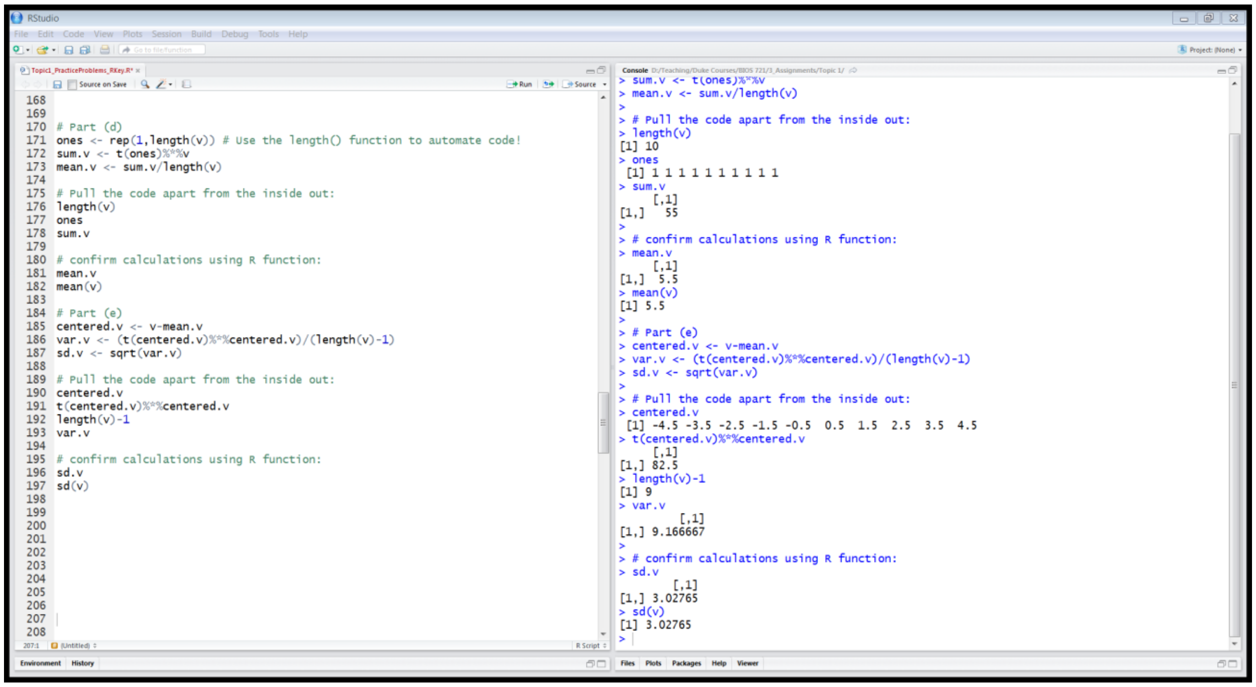
```
# Part (d)
# Use the length() function to automate code!
ones <- rep(1,length(v))
sum.v <- t(ones)%*%v
mean.v <- sum.v/length(v)

# Pull the code apart from the inside out:
length(v)
ones
sum.v

# confirm calculations using R function:
mean.v
mean(v)
```

e. Using vectors and matrix algebra, find the standard deviation of the elements in the data object called v. Make sure your code is automated and would work properly for ANY vector v. Confirm your answer using a function available in R.
    i. Hint: You may have to create additional vectors to answer this question.

R Code:

```
# Part (e)
centered.v <- v-mean.v
var.v <- (t(centered.v)%*%centered.v)/(length(v)-1)
sd.v <- sqrt(var.v)

# Pull the code apart from the inside out:
centered.v
t(centered.v)%*%centered.v
length(v)-1
var.v

# confirm calculations using R function:
sd.v
sd(v)
```

R Screenshot:



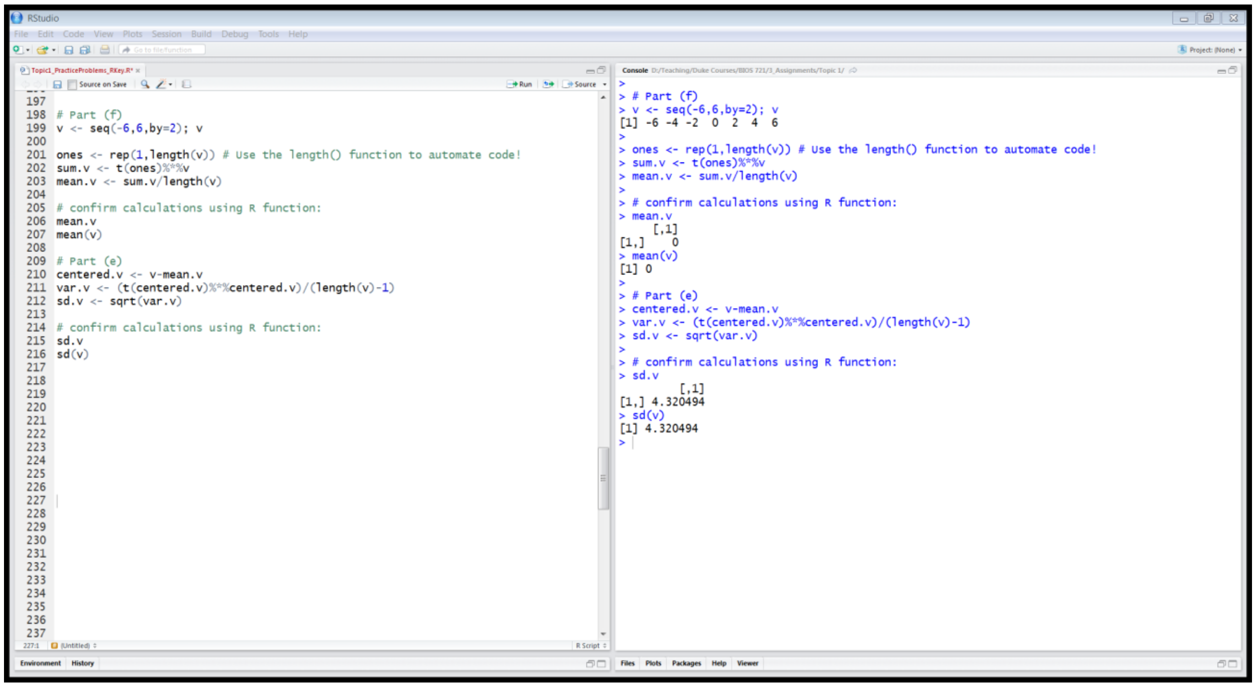
```
# Part (d)
ones <- rep(1,length(v)) # Use the length() function to automate code!
sum.v <- t(ones)%*%v
mean.v <- sum.v/length(v)

# Pull the code apart from the inside out:
length(v)
ones
sum.v

# confirm calculations using R function:
mean.v
mean(v)

# Part (e)
centered.v <- v-mean.v
var.v <- (t(centered.v)%*%centered.v)/(length(v)-1)
sd.v <- sqrt(var.v)

# Pull the code apart from the inside out:
centered.v
t(centered.v)%*%centered.v
length(v)-1
var.v

# confirm calculations using R function:
sd.v
sd(v)
```

```
> sum.v <- t(ones)%*%v
> mean.v <- sum.v/length(v)
>
> # Pull the code apart from the inside out:
> length(v)
[1] 10
> ones
 [1] 1 1 1 1 1 1 1 1 1 1
> sum.v
     [,1]
[1,]   55
>
> # confirm calculations using R function:
> mean.v
     [,1]
[1,]  5.5
> mean(v)
[1] 5.5
>
> # Part (e)
> centered.v <- v-mean.v
> var.v <- (t(centered.v)%*%centered.v)/(length(v)-1)
> sd.v <- sqrt(var.v)
>
> # Pull the code apart from the inside out:
> centered.v
 [1] -4.5 -3.5 -2.5 -1.5 -0.5  0.5  1.5  2.5  3.5  4.5
> t(centered.v)%*%centered.v
     [,1]
[1,] 82.5
> length(v)-1
[1] 9
> var.v
         [,1]
[1,] 9.166667
>
> # confirm calculations using R function:
> sd.v
        [,1]
[1,] 3.02765
> sd(v)
[1] 3.02765
>
```

f. Using the code developed in part (d) and part (e), find the mean and standard deviation of the following vector: v <- seq(-6,6,by=2). Confirm your answer using functions available in R.

R Code:
```
# Part (f)
v <- seq(-6,6,by=2); v

ones <- rep(1,length(v))
sum.v <- t(ones)%*%v
mean.v <- sum.v/length(v)

# confirm calculations using R function:
mean.v; mean(v)

# Part (e)
centered.v <- v-mean.v
var.v <- (t(centered.v)%*%centered.v)/(length(v)-1)
sd.v <- sqrt(var.v)

# confirm calculations using R function:
sd.v; sd(v)
```

R Screenshot:



```
# Part (f)
v <- seq(-6,6,by=2); v

ones <- rep(1,length(v)) # Use the length() function to automate code!
sum.v <- t(ones)%*%v
mean.v <- sum.v/length(v)

# confirm calculations using R function:
mean.v
mean(v)

# Part (e)
centered.v <- v-mean.v
var.v <- (t(centered.v)%*%centered.v)/(length(v)-1)
sd.v <- sqrt(var.v)

# confirm calculations using R function:
sd.v
sd(v)
```

```
>
> # Part (f)
> v <- seq(-6,6,by=2); v
[1] -6 -4 -2  0  2  4  6
>
> ones <- rep(1,length(v)) # Use the length() function to automate code!
> sum.v <- t(ones)%*%v
> mean.v <- sum.v/length(v)
>
> # confirm calculations using R function:
> mean.v
     [,1]
[1,]    0
> mean(v)
[1] 0
>
> # Part (e)
> centered.v <- v-mean.v
> var.v <- (t(centered.v)%*%centered.v)/(length(v)-1)
> sd.v <- sqrt(var.v)
>
> # confirm calculations using R function:
> sd.v
         [,1]
[1,] 4.320494
> sd(v)
[1] 4.320494
>
```

7. A student is tasked with simulating (generating data using R based on assumed probability distributions) the blood glucose levels of 15 patients with type 1 diabetes or type 2 diabetes. The following data was simulated: the glucose1 measurement is a fasting blood sugar measurement, the glucose2 measurement is a blood sugar measurement after a low carb lunch, and the glucose3 measurement is a blood sugar measurement after a high carb dinner. He uses the following code to generate the first test data set:

```
patientID <- 1:10
set.seed(15)
age <- runif(n=10,min=25,max=85)
type <- sample(x=c("Type1","Type2"),size=10,
              replace=TRUE,prob=c(0.5,0.5))
glucose1 <- round(runif(n=10,min=50,max=120))
glucose2 <- round(runif(n=10,min=100,max=250))
glucose3 <- round(runif(n=10,min=180,max=450))
sugar<-cbind(patientID,type,glucose1,glucose2,glucose3); sugar
```

Note: The sample() function randomly draws from vector x, to generate a vector the length specified with the size input, with or without replacement, and assigns the probabilities of drawing each value of in x, according to the vector, prob. The runif() function is used to generate a vector of n randomly drawn numbers from a uniform distribution with the provided minimum and maximum. The default of the round() function, rounds an numeric object (scalar, vector, matrix, etc), to the nearest whole number, and the set.seed() function ensures that the random numbers generated are the same each time the code is executed. For more details, remember you can access the R help pages!

a. The code above produces the following output:

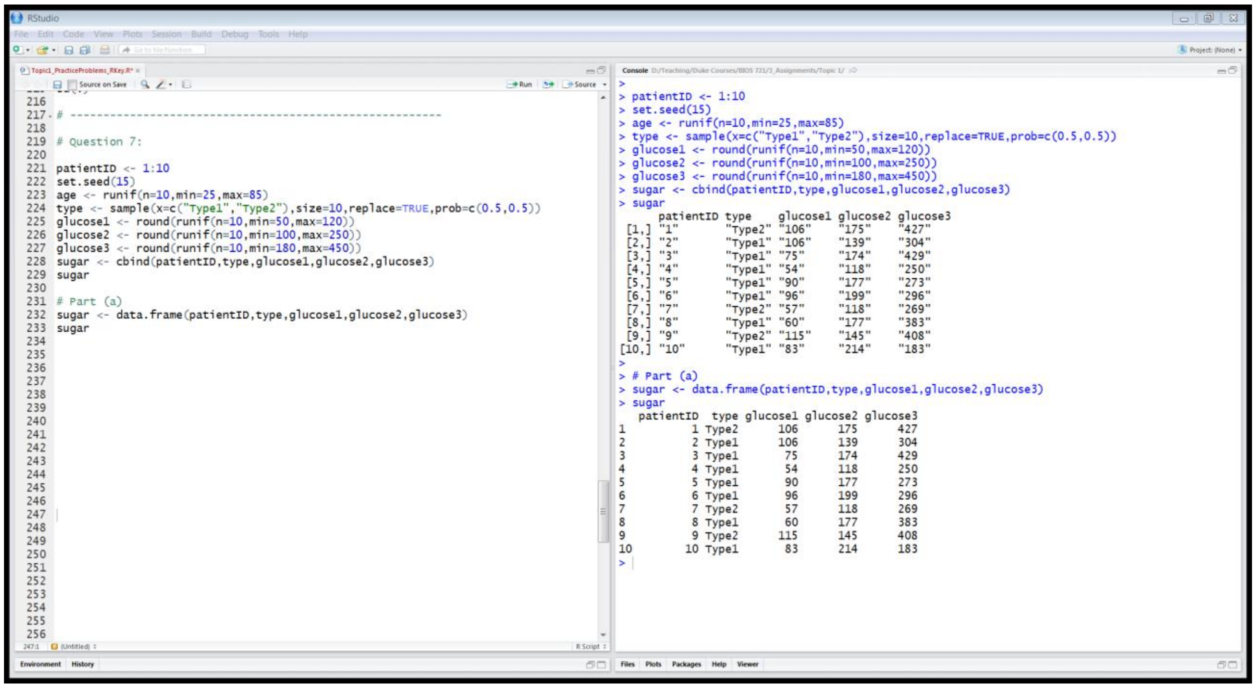
```
      patientID type    glucose1 glucose2 glucose3
 [1,] "1"       "Type2" "106"    "175"    "427"
 [2,] "2"       "Type1" "106"    "139"    "304"
 [3,] "3"       "Type1" "75"     "174"    "429"
 [4,] "4"       "Type1" "54"     "118"    "250"
 [5,] "5"       "Type1" "90"     "177"    "273"
 [6,] "6"       "Type1" "96"     "199"    "296"
 [7,] "7"       "Type2" "57"     "118"    "269"
 [8,] "8"       "Type1" "60"     "177"    "383"
 [9,] "9"       "Type2" "115"    "145"    "408"
[10,] "10"      "Type1" "83"     "214"    "183"
```

Will the student be able to perform numerical analyses on the glucose measurements using the object called sugar? If so, simply state so. If not, explain why. Could you modify the R code given above so that numerical analyses could be performed? If so, provide the modified R code.

No, the student will not be able to perform numerical analyses on the glucose measurements using the data object sugar because the elements are all characters. This happened because the student stored the data variables in a MATRIX which can only contain one type of element – here, the presence of character variables coerced the numeric variables to character when creating the sugar data object. To fix this error, the student should have created a data frame. See the updated R code below.

R Code:
```
# Part (a)
sugar <- data.frame(patientID,type,glucose1,glucose2,glucose3)
sugar
```

R Screenshot:



```
# ------------------------------------------------------------

# Question 7:

patientID <- 1:10
set.seed(15)
age <- runif(n=10,min=25,max=85)
type <- sample(x=c("Type1","Type2"),size=10,replace=TRUE,prob=c(0.5,0.5))
glucose1 <- round(runif(n=10,min=50,max=120))
glucose2 <- round(runif(n=10,min=100,max=250))
glucose3 <- round(runif(n=10,min=180,max=450))
sugar <- cbind(patientID,type,glucose1,glucose2,glucose3)
sugar

# Part (a)
sugar <- data.frame(patientID,type,glucose1,glucose2,glucose3)
sugar
```

```
>
> patientID <- 1:10
> set.seed(15)
> age <- runif(n=10,min=25,max=85)
> type <- sample(x=c("Type1","Type2"),size=10,replace=TRUE,prob=c(0.5,0.5))
> glucose1 <- round(runif(n=10,min=50,max=120))
> glucose2 <- round(runif(n=10,min=100,max=250))
> glucose3 <- round(runif(n=10,min=180,max=450))
> sugar <- cbind(patientID,type,glucose1,glucose2,glucose3)
> sugar
      patientID type    glucose1 glucose2 glucose3
 [1,] "1"       "Type2" "106"    "175"    "427"
 [2,] "2"       "Type1" "106"    "139"    "304"
 [3,] "3"       "Type1" "75"     "174"    "429"
 [4,] "4"       "Type1" "54"     "118"    "250"
 [5,] "5"       "Type1" "90"     "177"    "273"
 [6,] "6"       "Type1" "96"     "199"    "296"
 [7,] "7"       "Type2" "57"     "118"    "269"
 [8,] "8"       "Type1" "60"     "177"    "383"
 [9,] "9"       "Type2" "115"    "145"    "408"
[10,] "10"      "Type1" "83"     "214"    "183"
>
> # Part (a)
> sugar <- data.frame(patientID,type,glucose1,glucose2,glucose3)
> sugar
   patientID  type glucose1 glucose2 glucose3
1          1 Type2      106      175      427
2          2 Type1      106      139      304
3          3 Type1       75      174      429
4          4 Type1       54      118      250
5          5 Type1       90      177      273
6          6 Type1       96      199      296
7          7 Type2       57      118      269
8          8 Type1       60      177      383
9          9 Type2      115      145      408
10        10 Type1       83      214      183
>
```

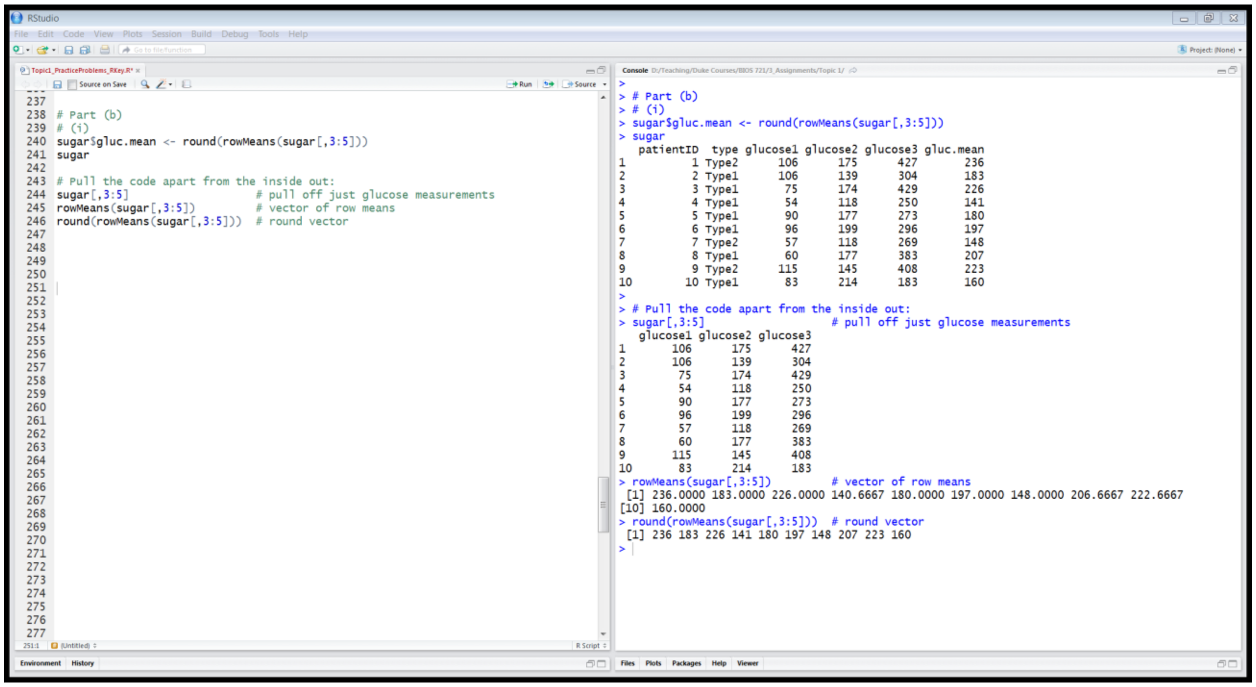
b. The student wants to compare the average glucose values across all fasting conditions (fasting, before meal, and after meal) between patients. Add a variable called gluc.mean to the data object sugar that contains this value. Round the answers to the nearest whole number.

  i. Perform this task using the rowMeans() function.

R Code:

```
# Part (b)
# (i)
sugar$gluc.mean <- round(rowMeans(sugar[,3:5]))
sugar

# Pull the code apart from the inside out:
sugar[,3:5]     # pull off just glucose measurements
rowMeans(sugar[,3:5])           # vector of row means
round(rowMeans(sugar[,3:5]))   # round vector
```

R Screenshot:



```
# Part (b)
# (i)
sugar$gluc.mean <- round(rowMeans(sugar[,3:5]))
sugar

# Pull the code apart from the inside out:
sugar[,3:5]                  # pull off just glucose measurements
rowMeans(sugar[,3:5])        # vector of row means
round(rowMeans(sugar[,3:5]))  # round vector
```

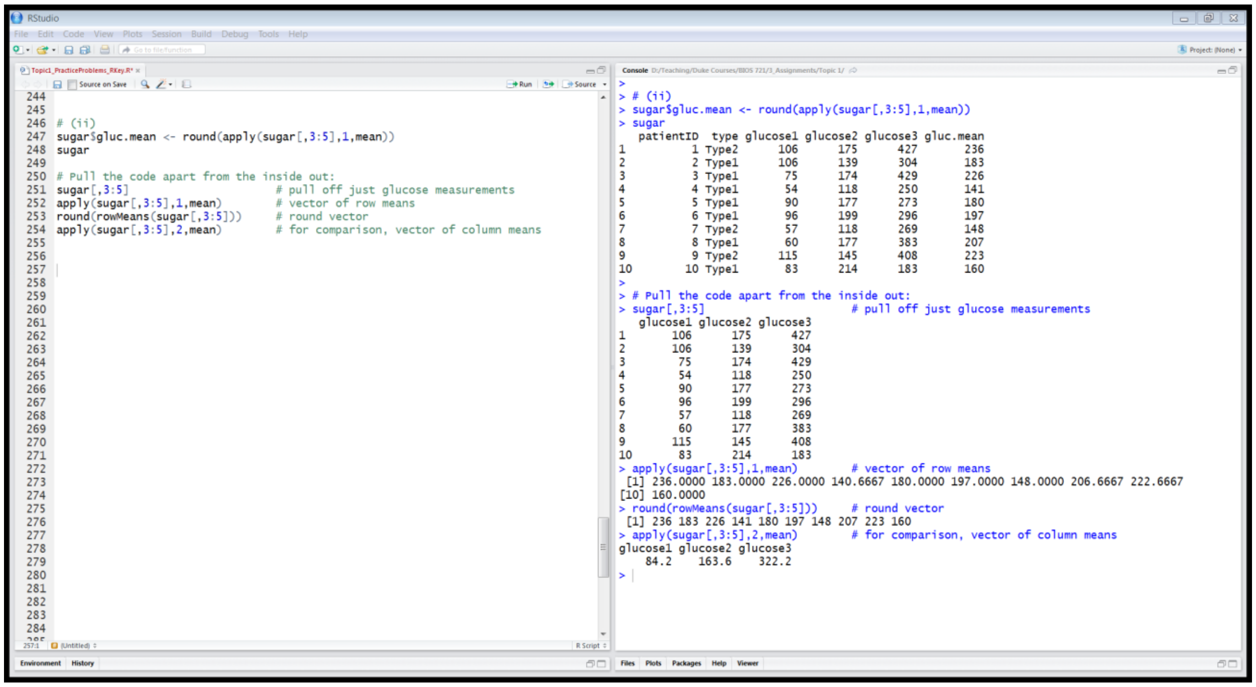
```
>
> # Part (b)
> # (i)
> sugar$gluc.mean <- round(rowMeans(sugar[,3:5]))
> sugar
   patientID  type glucose1 glucose2 glucose3 gluc.mean
1          1 Type2      106      175      427       236
2          2 Type1      106      139      304       183
3          3 Type1       75      174      429       226
4          4 Type1       54      118      250       141
5          5 Type1       90      177      273       180
6          6 Type1       96      199      296       197
7          7 Type2       57      118      269       148
8          8 Type1       60      177      383       207
9          9 Type2      115      145      408       223
10        10 Type1       83      214      183       160
>
> # Pull the code apart from the inside out:
> sugar[,3:5]                  # pull off just glucose measurements
   glucose1 glucose2 glucose3
1       106      175      427
2       106      139      304
3        75      174      429
4        54      118      250
5        90      177      273
6        96      199      296
7        57      118      269
8        60      177      383
9       115      145      408
10       83      214      183
> rowMeans(sugar[,3:5])        # vector of row means
 [1] 236.0000 183.0000 226.0000 140.6667 180.0000 197.0000 148.0000 206.6667 222.6667
[10] 160.0000
> round(rowMeans(sugar[,3:5]))  # round vector
 [1] 236 183 226 141 180 197 148 207 223 160
>
```

ii. Perform this task again using the apply() function.

R Code:

```
# (ii)
sugar$gluc.mean <- round(apply(sugar[,3:5],1,mean))
sugar

# Pull the code apart from the inside out:
sugar[,3:5]      # pull off just glucose measurements
apply(sugar[,3:5],1,mean)        # vector of row means
round(rowMeans(sugar[,3:5]))     # round vector
apply(sugar[,3:5],2,mean)        # for comparison,
                                 # vector of col means
```

R Screenshot:



```
# (ii)
sugar$gluc.mean <- round(apply(sugar[,3:5],1,mean))
sugar

# Pull the code apart from the inside out:
sugar[,3:5]                      # pull off just glucose measurements
apply(sugar[,3:5],1,mean)        # vector of row means
round(rowMeans(sugar[,3:5]))     # round vector
apply(sugar[,3:5],2,mean)        # for comparison, vector of column means
```

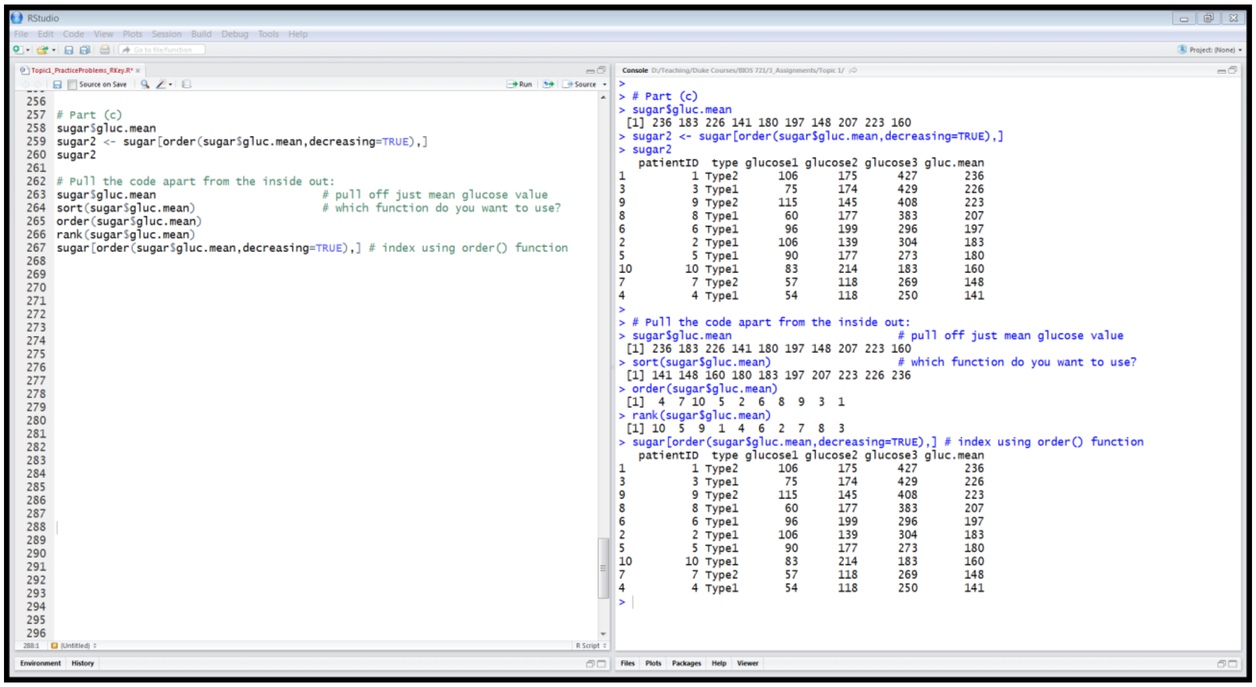
```
>
> # (ii)
> sugar$gluc.mean <- round(apply(sugar[,3:5],1,mean))
> sugar
  patientID  type glucose1 glucose2 glucose3 gluc.mean
1         1 Type2      106      175      427       236
2         2 Type1      106      139      304       183
3         3 Type1       75      174      429       226
4         4 Type1       54      118      250       141
5         5 Type1       90      177      273       180
6         6 Type1       96      199      296       197
7         7 Type2       57      118      269       148
8         8 Type1       60      177      383       207
9         9 Type2      115      145      408       223
10       10 Type1       83      214      183       160
>
> # Pull the code apart from the inside out:
> sugar[,3:5]                      # pull off just glucose measurements
  glucose1 glucose2 glucose3
1      106      175      427
2      106      139      304
3       75      174      429
4       54      118      250
5       90      177      273
6       96      199      296
7       57      118      269
8       60      177      383
9      115      145      408
10      83      214      183
> apply(sugar[,3:5],1,mean)        # vector of row means
 [1] 236.0000 183.0000 226.0000 140.6667 180.0000 197.0000 148.0000 206.6667 222.6667
[10] 160.0000
> round(rowMeans(sugar[,3:5]))     # round vector
 [1] 236 183 226 141 180 197 148 207 223 160
> apply(sugar[,3:5],2,mean)        # for comparison, vector of column means
glucose1 glucose2 glucose3
    84.2    163.6    322.2
>
```

c. The student thinks it might be useful to sort the data by the value of gluc.mean from highest to lowest to examine which patients tend to have poor blood glucose regulation. Perform this task using one of the following R functions: the sort(), order(), or rank() function. Make sure to not overwrite the data object sugar when performing this task; instead, create a new data object called sugar2.

R Code:

```
# Part (c)
sugar$gluc.mean
sugar2 <- sugar[order(sugar$gluc.mean,decreasing=TRUE),]
sugar2

# Pull the code apart from the inside out:
sugar$gluc.mean         # pull off just mean glucose value
sort(sugar$gluc.mean)   # which function to use?
order(sugar$gluc.mean)
rank(sugar$gluc.mean)
                        # index using order() function
sugar[order(sugar$gluc.mean,decreasing=TRUE),]
```

R Screenshot:



```
# Part (c)
sugar$gluc.mean
sugar2 <- sugar[order(sugar$gluc.mean,decreasing=TRUE),]
sugar2

# Pull the code apart from the inside out:
sugar$gluc.mean                              # pull off just mean glucose value
sort(sugar$gluc.mean)                        # which function do you want to use?
order(sugar$gluc.mean)
rank(sugar$gluc.mean)
sugar[order(sugar$gluc.mean,decreasing=TRUE),] # index using order() function
```

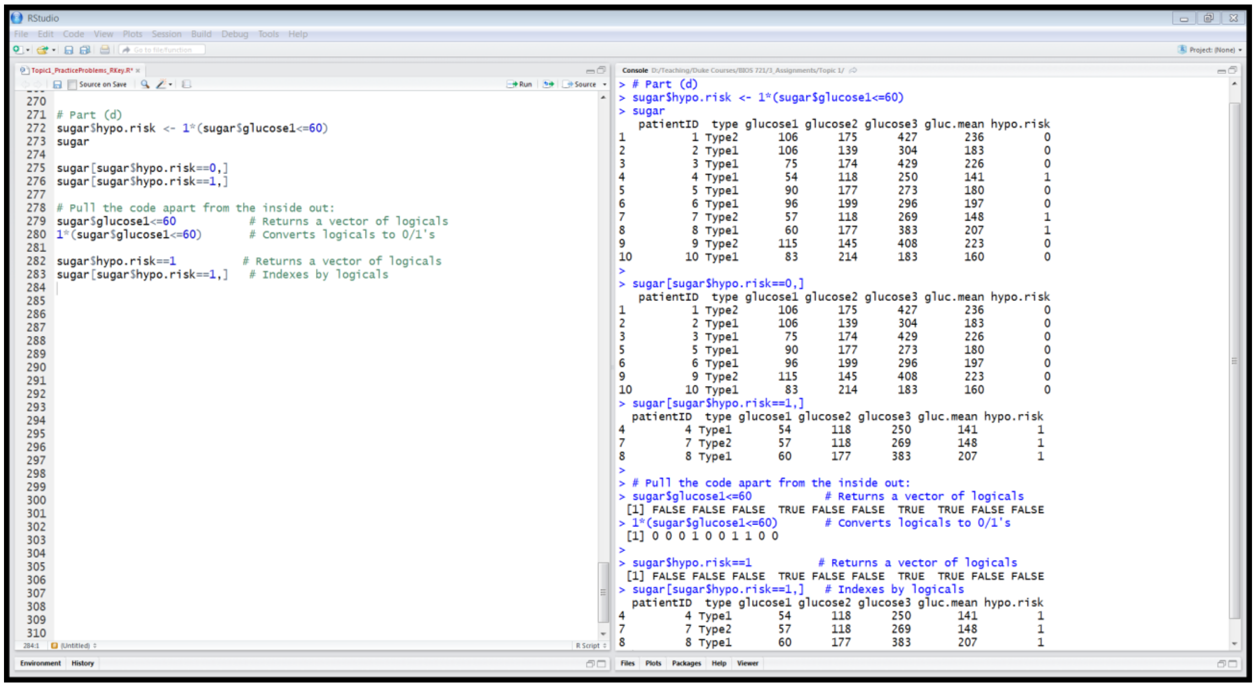
```
>
> # Part (c)
> sugar$gluc.mean
 [1] 236 183 226 141 180 197 148 207 223 160
> sugar2 <- sugar[order(sugar$gluc.mean,decreasing=TRUE),]
> sugar2
   patientID  type glucose1 glucose2 glucose3 gluc.mean
1          1 Type2      106      175      427       236
3          3 Type1       75      174      429       226
9          9 Type2      115      145      408       223
8          8 Type1       60      177      383       207
6          6 Type1       96      199      296       197
2          2 Type1      106      139      304       183
5          5 Type1       90      177      273       180
10        10 Type1       83      214      183       160
7          7 Type2       57      118      269       148
4          4 Type1       54      118      250       141
>
> # Pull the code apart from the inside out:
> sugar$gluc.mean                              # pull off just mean glucose value
 [1] 236 183 226 141 180 197 148 207 223 160
> sort(sugar$gluc.mean)                        # which function do you want to use?
 [1] 141 148 160 180 183 197 207 223 226 236
> order(sugar$gluc.mean)
 [1]  4  7 10  5  2  6  8  9  3  1
> rank(sugar$gluc.mean)
 [1] 10  5  9  1  4  6  2  7  8  3
> sugar[order(sugar$gluc.mean,decreasing=TRUE),] # index using order() function
   patientID  type glucose1 glucose2 glucose3 gluc.mean
1          1 Type2      106      175      427       236
3          3 Type1       75      174      429       226
9          9 Type2      115      145      408       223
8          8 Type1       60      177      383       207
6          6 Type1       96      199      296       197
2          2 Type1      106      139      304       183
5          5 Type1       90      177      273       180
10        10 Type1       83      214      183       160
7          7 Type2       57      118      269       148
4          4 Type1       54      118      250       141
> 
```

d. Ideally, fasting blood sugars should be above 60 in diabetics. Falling substantially below this cutoff puts a patient at risk for hypoglycemic shock and many diabetics have trouble avoiding this overnight if they take medications in the evening. The student wants to create a new variable called hypo.risk that is zero if the patient is not at risk and 1 if the patient is at risk. Provide code that creates this variable and adds it to the data object called sugar. To verify your code print all patients with glucose1 less than 60 and glucose1 greater than 60 separately.

R Code:

```
# Part (d)
sugar$hypo.risk <- 1*(sugar$glucose1<=60)
sugar

sugar[sugar$hypo.risk==0,]
sugar[sugar$hypo.risk==1,]

# Pull the code apart from the inside out:
sugar$glucose1<=60          # Returns a vector of logicals
1*(sugar$glucose1<=60)      # Converts logicals to 0/1's

sugar$hypo.risk==1          # Returns a vector of logicals
sugar[sugar$hypo.risk==1,]# Index using logicals
```

R Screenshot:



```
# Part (d)
sugar$hypo.risk <- 1*(sugar$glucose1<=60)
sugar

sugar[sugar$hypo.risk==0,]
sugar[sugar$hypo.risk==1,]

# Pull the code apart from the inside out:
sugar$glucose1<=60          # Returns a vector of logicals
1*(sugar$glucose1<=60)      # Converts logicals to 0/1's

sugar$hypo.risk==1          # Returns a vector of logicals
sugar[sugar$hypo.risk==1,]   # Indexes by logicals
```

```
> # Part (d)
> sugar$hypo.risk <- 1*(sugar$glucose1<=60)
> sugar
   patientID  type glucose1 glucose2 glucose3 gluc.mean hypo.risk
1          1 Type2      106      175      427       236         0
2          2 Type1      106      139      304       183         0
3          3 Type1       75      174      429       226         0
4          4 Type1       54      118      250       141         1
5          5 Type1       90      177      273       180         0
6          6 Type1       96      199      296       197         0
7          7 Type2       57      118      269       148         1
8          8 Type1       60      177      383       207         1
9          9 Type2      115      145      408       223         0
10        10 Type1       83      214      183       160         0
>
> sugar[sugar$hypo.risk==0,]
   patientID  type glucose1 glucose2 glucose3 gluc.mean hypo.risk
1          1 Type2      106      175      427       236         0
2          2 Type1      106      139      304       183         0
3          3 Type1       75      174      429       226         0
5          5 Type1       90      177      273       180         0
6          6 Type1       96      199      296       197         0
9          9 Type2      115      145      408       223         0
10        10 Type1       83      214      183       160         0
> sugar[sugar$hypo.risk==1,]
  patientID  type glucose1 glucose2 glucose3 gluc.mean hypo.risk
4         4 Type1       54      118      250       141         1
7         7 Type2       57      118      269       148         1
8         8 Type1       60      177      383       207         1
>
> # Pull the code apart from the inside out:
> sugar$glucose1<=60           # Returns a vector of logicals
 [1] FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE
> 1*(sugar$glucose1<=60)       # Converts logicals to 0/1's
 [1] 0 0 0 1 0 0 1 1 0 0
>
> sugar$hypo.risk==1          # Returns a vector of logicals
 [1] FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE
> sugar[sugar$hypo.risk==1,]   # Indexes by logicals
  patientID  type glucose1 glucose2 glucose3 gluc.mean hypo.risk
4         4 Type1       54      118      250       141         1
7         7 Type2       57      118      269       148         1
8         8 Type1       60      177      383       207         1
```