Here we want to give some brief introduction to R. Due to its popularity, there are plenty of good materials online that can be used as tutorials. However, the best way to learn R according my personal experience is by using it to do your own analysis by trials and errors, and directly use Google to search when you have any question. In addition, the helper documentation installed in R is one of the most useful ways to know exactly how R works. Having said that, it actually means that I myself never used any R tutorial. However, on the other hand, R programing is something you might want to think deeply about, if you would like to do an expert in R. For that purpose, one recommended reference is John Chambers’ book - Software for Data Analysis. It doesn’t give detailed discussion on how to use R in data analysis. Instead, it focuses on using R as a programing language, how to create your own R package and some fundamental concept of R programing, such as functional programing. Some text books contain nice examples of R. One recent book is An Introduction to Statistical Learning with Applications in R, from G. James, D. Witten, T. Hastie, R. Tibshirani (2013).

Below we simply give some illustration on basic R functions.

### 0.1 Data manipulation

> ```
> ## most important, find the function documentation
> ?svd
> ```

What if you don’t know what function to use? Google it!

Now let’s try some real examples.

> ```
> data(iris)
> class(iris)
> ```

[1] "data.frame"

> ```
> ## data.frame is a very special data structure in R.
> dim(iris)
> ```
\begin{verbatim}
> nrow(iris)
[1] 150

> ncol(iris)
[1] 5

> summary(iris)

  Sepal.Length  Sepal.Width  Petal.Length  Petal.Width
   Min.    :4.300  Min.    :2.000  Min.    :1.000  Min.    :0.100
  1st Qu.  :5.100  1st Qu.  :2.800  1st Qu.  :1.600  1st Qu.  :0.300
  Median   :5.800  Median   :3.000  Median   :4.350  Median   :1.300
  Mean     :5.843  Mean     :3.057  Mean     :3.758  Mean     :1.199
  3rd Qu.  :6.400  3rd Qu.  :3.300  3rd Qu.  :5.100  3rd Qu.  :1.800

Species
  setosa :50
  versicolor:50
  virginica:50

> table(iris$Species)

    setosa versicolor virginica
   setosa    :50       :50       :50

> unique(iris$Species)

[1] setosa  versicolor virginica

Levels: setosa versicolor virginica

In the case when one wants to have a look at the first a few rows or last a few rows, the function \texttt{head()} and \texttt{tail()} can be used.

> head(iris)

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
  1       5.1     3.5          1.4       0.2       setosa
  2       4.9     3.0          1.4       0.2       setosa
  3       4.7     3.2          1.3       0.2       setosa
  4       4.6     3.1          1.5       0.2       setosa
  5       5.0     3.6          1.4       0.2       setosa
  6       5.4     3.9          1.7       0.4       setosa

In data frame, the elements can be called as usually in matrix or by attribute name.
\end{verbatim}
> iris[1,2]
[1] 3.5

> iris$Sepal.Length[2]
[1] 4.9

Basic operation functions

> max(iris$Sepal.Length)
[1] 7.9

> which.max(iris$Sepal.Length)
[1] 132

> mean(iris$Sepal.Length)
[1] 5.843333

> median(iris$Sepal.Length)
[1] 5.8

> log(iris$Sepal.Length)

[1] 1.629241 1.589235 1.547563 1.526056 1.609438 1.686399 1.526056 1.609438
[9] 1.481605 1.589235 1.686399 1.568616 1.458615 1.757858 1.740466
[17] 1.686399 1.629241 1.740466 1.629241 1.686399 1.629241 1.526056 1.629241
[25] 1.568616 1.609438 1.609438 1.648659 1.648659 1.547563 1.568616 1.686399
[33] 1.648659 1.704748 1.589235 1.609438 1.704748 1.589235 1.481605 1.629241
[41] 1.609438 1.504077 1.481605 1.609438 1.629241 1.568616 1.629241 1.526056
[49] 1.667707 1.609438 1.945910 1.856298 1.931521 1.704748 1.871802 1.740466
[57] 1.840550 1.589235 1.887070 1.648659 1.609438 1.774952 1.791759 1.808289
[65] 1.722767 1.902108 1.722767 1.757858 1.824549 1.722767 1.774952 1.808289
[73] 1.840550 1.808289 1.856298 1.887070 1.916923 1.902108 1.791759 1.740466
[81] 1.704748 1.704748 1.757858 1.791759 1.686399 1.791759 1.902108 1.840550
[89] 1.722767 1.704748 1.704748 1.808289 1.757858 1.609438 1.722767 1.740466
[97] 1.740466 1.824549 1.629241 1.740466 1.840550 1.757858 1.960095 1.840550
[105] 1.871802 2.028148 1.589235 1.987874 1.902108 1.974081 1.974081 1.856298
[113] 1.916923 1.740466 1.757858 1.856298 1.871802 2.041220 2.041220 1.791759
[121] 1.931521 1.722767 2.041220 1.840550 1.902108 1.974081 1.824549 1.808289
[129] 1.856298 1.974081 2.001480 2.06863 1.856298 1.840550 1.808289 2.041220
[137] 1.840550 1.856298 1.791759 1.931521 1.902108 1.931521 1.757858 1.916923
[145] 1.902108 1.902108 1.840550 1.871802 1.824549 1.774952
### functions are operation by element

```r
> x <- iris$Sepal.Length[1:5]
> y <- iris$Sepal.Width[1:5]
> x
[1] 5.1 4.9 4.7 4.6 5.0
> y
[1] 3.5 3.0 3.2 3.1 3.6
> x*y
[1] 17.85 14.70 15.04 14.26 18.00
```

Some commonly used random number and vector generator:

```r
> set.seed(100) ## Important if you want reproducible results
> rnorm(1) ## standard normal
[1] -0.5021924
```

```r
> x <- rnorm(10) ## generate 10 of them as a vector or array
> u <- runif(10) ## uniform(0,1)
> y <- seq(1,10)
> y <- seq(1,10,by=2)
> t <- seq(1,10,length.out=3)
```

Matrices operations

```r
> X <- matrix(x,nrow=5,ncol=2)
> Y <- matrix(y,nrow=2,ncol=5)
> t(X) ## transpose
[1,] 0.1315312 -0.07891709 0.8867848 0.1169713 0.31863009
[2,] -0.5817907 0.71453271 -0.8252594 -0.3598621 0.08988614
> solve(t(X)%*%X)
```

```r
> X%*%Y ## multiplication
[1,] -1.6138409 -3.414879  0.601989803 -2.514360 -4.315398
[2,]  2.0646810  4.607144  0.004278903  3.335912  5.878375
[3,] -1.5889935 -1.342892  7.165803859 -1.465943 -1.219841
[4,] -0.9626151 -1.934179  0.692879303 -1.448397 -2.419960
[5,]  0.5882885  2.222353  2.957556932  1.405321  3.039386
> solve(t(X)%*%X)
```
## Try to directly use the linear equations operation directly with another matrix

\[ \text{svd}(X) \]

\[ d \\
\begin{bmatrix}
1.4999491 \\
0.5856165
\end{bmatrix}
\]

\[ u \\
\begin{bmatrix}
-0.37173995 & 0.3617262 \\
0.42612004 & -0.5618543 \\
-0.78524426 & -0.4834820 \\
-0.24305596 & 0.1730553 \\
-0.06742681 & -0.5383038
\end{bmatrix}
\]

\[ v \\
\begin{bmatrix}
-0.5525394 & -0.8334868 \\
0.8334868 & -0.5525394
\end{bmatrix}
\]

Indexing of matrix (and dataframe): we can use single index to call elements. In a matrix or dataframe, the broadcasting is going across rows along each column. We operate a matrix with a vector, the same rule applies.

\[ X \\
\begin{bmatrix}
0.13153117 & -0.58179068 \\
-0.07891709 & 0.71453271 \\
0.88678481 & -0.82525943 \\
0.11697127 & -0.35986213 \\
0.31863009 & 0.08988614
\end{bmatrix}
\]

\[ X[2] \quad \text{this is } X[2,1] \]

\[ -0.07891709 \]

\[ X[6] \quad \text{this is } X[1,2] \]

\[ -0.5817907 \]

\[ Y+y \]
0.2 Basic loops

As in usual programming languages, R supports the basic for loop, while loop, if-else statements etc. The logic operation rule is also the same.

```r
> x <- rep(0,20)
> for(k in 1:10){
+   x[k] <- k
+ }
> x
[1] 1 2 3 4 5 6 7 8 9 10 0 0 0 0 0 0 0 0 0 0
> while(k<20){
+   k <- k+1
+   x[k] <- k
+ }
> x
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
> TRUE&&FALSE
[1] FALSE
> TRUE||FALSE
[1] TRUE
```

Note that using loops in R can be slow. This is an inherent drawback of such high-level languages, compared to more basic languages such as C++. For small iterations, this might not be a big deal. But if your computation is going to involves a lot of for loops, you might want to think about some other ways for it. Next week, we will briefly talk about vectorized programing. In next section, we give give a few convinient functions as alternatives of loops for simple operations.
0.3 apply functions

The “apply” functions are a family of functions that are designed to accelerate operations along certain data structures. These are functions are included in R base package, so you can call them directly. Here we only illustrate these basic functions. There is another package that contains even more finer designed ones, called plyr.

```r
> X <- matrix(1:25,5,5)
> colSums(X) ## calculate sum() by columns
[1] 15 40 65 90 115
> rowMeans(X) ## calculate mean() by rows
[1] 11 12 13 14 15
> apply(X,2,sum) ## apply the sum function to each of the columns
[1] 15 40 65 90 115
> apply(X,1,mean) ## apply the mean function to each of the rows
[1] 11 12 13 14 15
> apply(X,1,function(x)x[2]^x[1]) ## you can also apply your own functions
[1] 6 49 512 6561 100000
```

## sapply and lapply are similar, but works on vectors and lists
## list is a very useful data structure as far as I see.
## It is able to take any type of element without any constraint on the lengths.
## The index can either by number or by string.
> set.seed(100)
> ## example: grades for different number of exams in each course
> Grades <- list()
> Grades["STATS600"] <- sample(100,3,replace=T)
> Grades["STATS601"] <- sample(100,4,replace=T)
> Grades["STATS610"] <- sample(100,2,replace=T)
> Grades["STATS611"] <- sample(100,2,replace=T)
> Grades["STATS620"] <- sample(100,2,replace=T)
> Grades["STATS621"] <- sample(100,2,replace=T)
> Grades[["STATS621"]]

```
[1] 40 77
```

> lapply(Grades,mean)
## Note that `lapply` returns a list
`unlist(lapply(Grades, mean))`

## you can use `unlist` to change a list to a vector

### 0.4 Packages

The packages are written by researcher all around the world for different data analysis needs. In most of the cases, the functions you need have been provided by some others. The only thing you have to do just to install the corresponding packages.

```r
# install.packages("survival")
library(survival)
require(survival)
#library(NoSuchPackage)
#require(NoSuchPackage)
```

Difference? `library` will load the package, and if it fails, an error will be thrown. The whole procedure will stop because of that. `require` will call the `library` and if it fails, a warning will be given. But it would not automatically stop everything. In most cases, these two are essentially the same, since if you do need package for some functions, your program will stop anyway. In very special circumstances, using `require` might have a little bit potential risk.
0.5 Plot

R is able to generate very beautiful figures, if you are good at it and patient enough. There are very nice plotting packages, such as \texttt{lattice}, \texttt{ggplot2}. Another interesting extension is \texttt{GGobi} (though I never used it myself). In most cases, the basic graphics functions in R is pretty enough.
> pairs(iris) ## Pairwise scatterplot

Figure 1: Pairwise scatterplot
```r
> hist(iris$Sepal.Width) ## histogram

Histogram of iris$Sepal.Width

Figure 2: Histogram
> boxplot(iris) ## boxplot
>

Figure 3: Boxplot
> set.seed(100)
> x <- runif(10)
> y <- runif(10)
> plot(x,y,type="p",main="Test plot",col="blue",
+     xlim=c(0,1),ylim=c(0,1),xlab="X-axis",
+     ylab="Y-axis",pch=20)
> points(y/2,x/2,col="red",pch=1:10)
> lines(y[1:2],x[1:2],col="purple")
> abline(v=0.5,col="green")
> abline(h=0.5,col="green",lty=2)
> abline(a=0,b=1,col="green",lwd=5)
>

![Test plot](image)

Figure 4: Generic plot function test