

uRProgramming

By

Zekai Otles

**Created on Tue Feb 23 22:46:16
2021**

Table of Contents

1 Introduction.....	4
2 Environment Setup for R programming Developers.....	4
2.1 Environment setup.....	4
2.2 Using Version Control System.....	4
2.3 Using IDE to develop R scripts.....	4
2.4 Writing functions.....	4
2.5 Create R package System.....	4
3 Simple Data Types and Objects.....	5
4 Control Structures.....	7
4.1 If statements.....	7
4.2 If then Else statements.....	9
4.3 Switch statements.....	9
5 Loop Structures.....	11
5.1 For loop.....	11
5.2 While loop.....	11
6 Data Structures.....	13
6.1 Data Frame.....	15
6.2 Vector.....	16
6.3 Matrix.....	17
6.4 List.....	17
7 Simple Statistical Analysis.....	18
8 Graphic Outputs.....	20
9 Simple Table Outputs.....	20
10 Reference.....	20

Acknowledgements

I would love thank my wife Sevtap Otles putting up with me while I was working on this book.

It would have been impossible to reach wherever I am at without my mom's trust and support. She (Zulfinaz Otles) influenced me and others to reach our education goals to be successful in life. She constantly worries the lack of educational opportunities for the people and children of Turkey, she is so concerned about education cost and how it is financially stressing the families.

My grand father (Ethem Demirkol) has taught me to work hard, be honest and treat all the people equal.

Finally, I would love thank to both of my sons (Erkin and Arel) for their sincerity, honesty, life choices and support to my wife and myself.

Thanks to all supporters, readers of this book and contributors to society.

1 Introduction

The goal of this document is to create a simple R-book utilizing software engineering development tools. Our vision for the readers of this book is to demonstrate how to develop R scripts and manage their scripts using currently available tools. Many people don't have much time to start learning R programming and don't have much experience to integrate their script with the existing software engineering tools. Here in this book, our aim is to start simple and built R software scripts using some software technologies.

In the next section of this book, we will inform the readers how to set up environment for writing R scripts. Also, in this section, we will introduce reader how to develop/write their first R program. We will briefly explain version control system which is required for any software development. Regardless of the project size, the version control system will greatly benefit any of the software developer. The writing R scripts is not an exception, that's why we would start integrating our software with a version control.

We will start explaining how and where to develop R scripts. The modern days software development requires fast development cycle. The fast development cycle brought us an Integrated Development Environments (IDEs), we could develop software checkout to or from version control system and run the code to see the immediate results. Here, we will use one of the widely used IDE

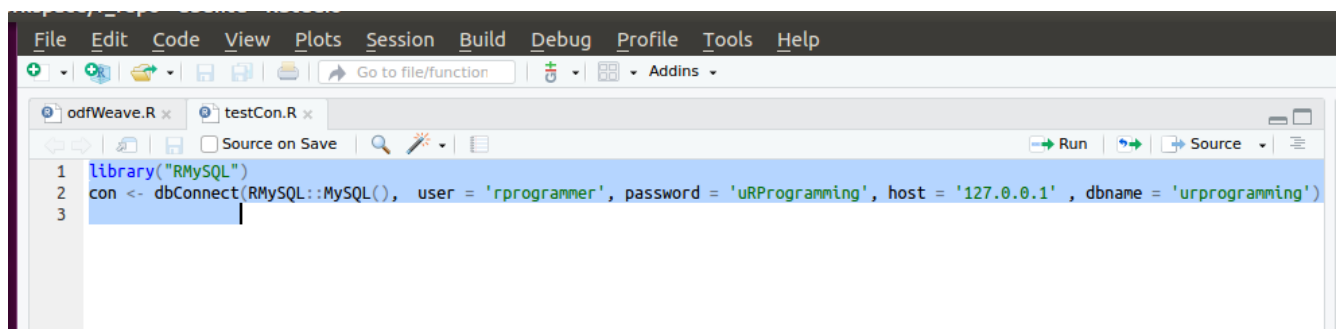
environment Rstudio to develop and maintain R-scripts. The Rstudio will help us to develop scripts faster and allow us to integrate the code development with version control system such as “git”. The git version control is a widely used version control system for the software development. The Rstudio will also give us options to run, debug and build options. Those are a few of the benefits of developing R scripts in an IDE.

2 Environment Setup for R programming Developers

2.1 Environment setup

The modern software developments are done by using integrated development environment (IDE). The development environment increases the productivity and integrates software building with other auxiliary tools. In order to develop R-scripts, one needs to have tools like Rstudio which can be downloaded from the website (<https://rstudio.com>). The R scripts can be written in Rstudio, the scripts can be run or debug from the source. The R scripts can be integrated with version control using IDE available plugin.

The portion of code snippets can be selected and run as shown here.



The results of the R scripts can be seen in the console panel of the R studio.

```
3:17 | (Top Level) | R Script
Console Terminal x
~/R_workspace/r_repo/
Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/R_workspace/r_repo/.RData]

Loading required package: RMySQL
Loading required package: DBI
> library("RMySQL")
> con <- dbConnect(RMySQL::MySQL(), user = 'rprogrammer', password = 'uRProgramming', host = '127.0.0.1', dbname = 'urprogramming')
> con
```

The “*help*” command is a useful keyword in R language. For example, if you want learn what is “*ls()*” command, you can type “*help(ls)*”. The *ls()* and *objects()* commands list the existing objects in the environment. A few commands will help reader to navigate in R environment, *dir()* lists the files and directories, *setwd(“dirName”)* will set the current working directory, *getwd()* gets the name of the current directory. Currently used packages and versions, can be listed with the following command *pkgVersions()*. Here is list of the packages currently in use.

	Package	Version
1	gridExtra	2.3
2	uRProgramming	1.1.1
3	tcltk	3.2.3
4	readr	1.1.1
5	log4r	0.2
6	odfWeave	0.8.4
7	XML	3.98-1.9
8	lattice	0.20-33
9	chron	2.3-51
10	RMySQL	0.10.13
11	DBI	0.7
12	stats	3.2.3
13	graphics	3.2.3
14	grDevices	3.2.3
15	utils	3.2.3
16	datasets	3.2.3

	Package	Version
17	methods	3.2.3
18	base	3.2.3

A simple function demonstrates the usage of `ls`, and next line shows how to get current directories and files.

```
> environment("ls")
usage of ls()
funcNameusage of ls(all=TRUE)
funcName
> cat("Current Working Dir \n", getwd())
Current Working Dir
/home/zekai/R_workspace/odfWeaveTmp
```

2.2 Using Version Control System

Writing any software program will also require management of changes and modifications of the codes. The person develops software continuously make modification and store changes; the version control systems specifically developed for these kinds of needs. The R script development is not different from the other software developments and requires constant modification and updates of the scripts. Utilizing version control system will immensely help the scripts development process.

There are so many different version control systems available such as CVS, SVN, Mercurial and git. Here, we decided to use one of the widely used version control systems **git**. The *git* is chosen because of the simplicity and it is also distributed version control system. In the following sections, we will demonstrate how to use *git* to manage R script development.

We can keep our codes in the clouds such as github, gitlab, bitbucket or Microsoft team foundation services. We can also deploy gitlab or git servers on the premise. Since git distributed, the developer also can keep their repository on their workstation. This infrastructure offers developers great flexibility to manage their codes locally and centrally. Having the codes in the cloud, frees user to owning and managing their infrastructure and offers great flexibility to collaborate with their peers.

Once the code repository is available, we can clone repository similar to the command “*git clone https://otles@bitbucket.org/otles/r_repo*”.

This command will clone “*r_repo*” into the current directory.

```
$ git clone https://otles@bitbucket.org/otles/r_repo.git
Cloning into 'r_repo'...
Password for 'https://otles@bitbucket.org':
remote: Counting objects: 1318, done.
remote: Compressing objects: 100% (1165/1165), done.
```

```
remote: Total 1318 (delta 785), reused 147 (delta 64)
Receiving objects: 100% (1318/1318), 9.59 MiB | 8.73 MiB/s, done.
Resolving deltas: 100% (785/785), done.
Checking connectivity... done.
zekai@zoubuntu:~$ cd r_repo
```

One can see, the checked-out version of the files by going into the folder “*cd r_repo*”. Type following command “*git checkout*” to see the directory is up to date or not.

```
$ git checkout
Your branch is up to date with 'origin/master'.
```

We can also checkout different branch by defining the branch name “*git clone -b branchName repoName*”.

```
$ git clone -b ubuntu https://otles@bitbucket.org/otles/r_repo.git
```

We could checkout the git repository from the R-studio, figure 2.3 shows how to check out the codes for the new project.

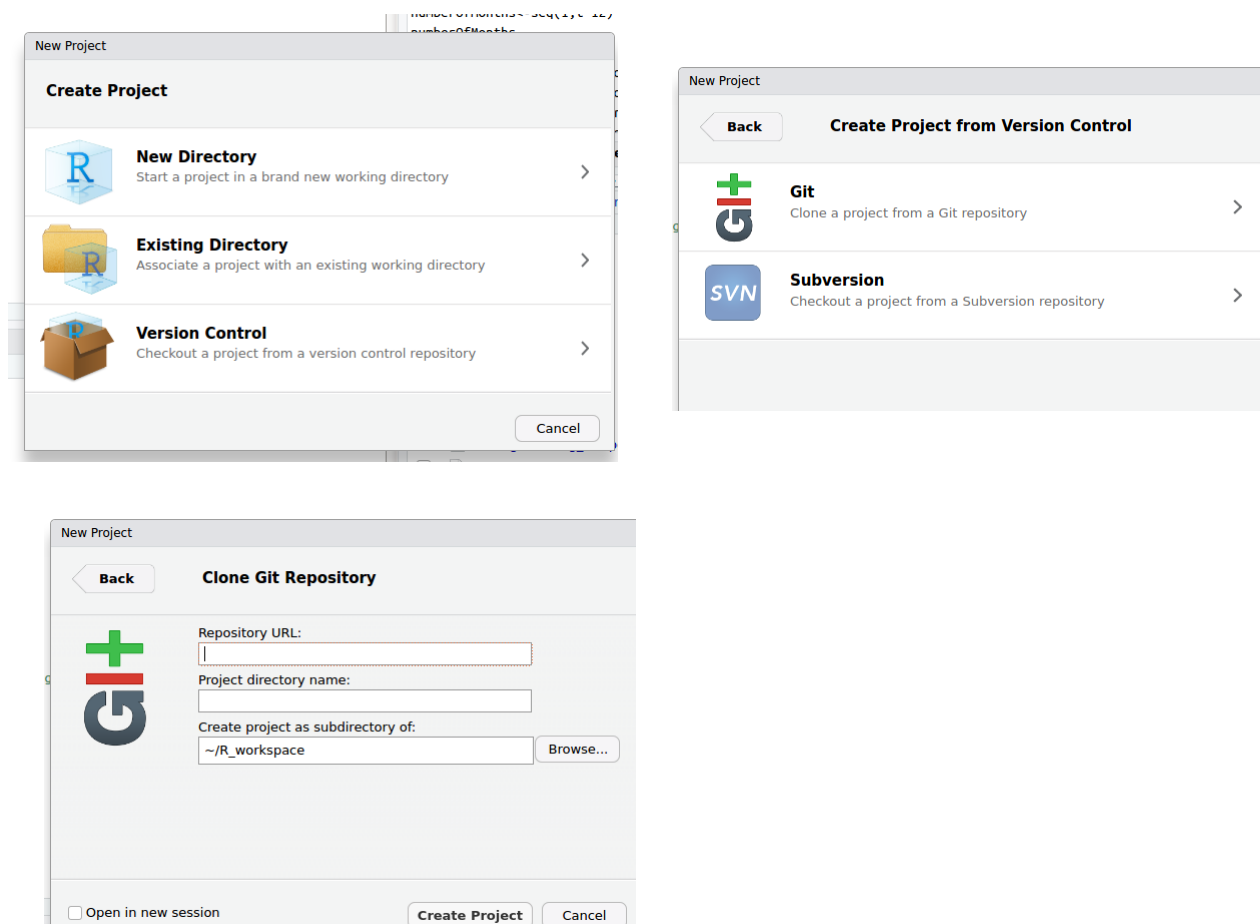


Figure 2.3 Creating new project in R-studio.

Once the developer start working on the codes and making modifications, the git repository plugin will mark the changes on the code. The figure 2.4 demonstrates how the modified files marked (notice the files are marked with “M”) and needs to be committed to the repository.

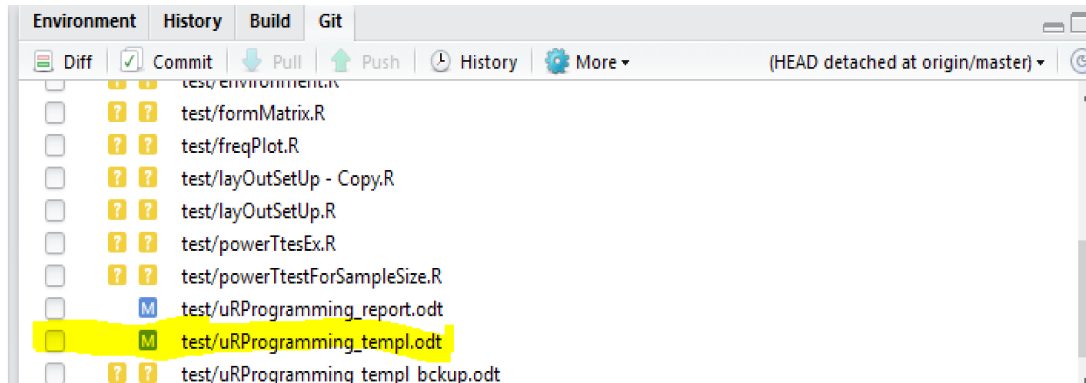


Figure 2.4: Git from Rstudio shows which files are modified recently.

The following screen from Rstudio shows how to commit changes and adding message “Modified after adding bar plot example to the document.”. After one makes changes and commit then code needs to be pushed into the repository.

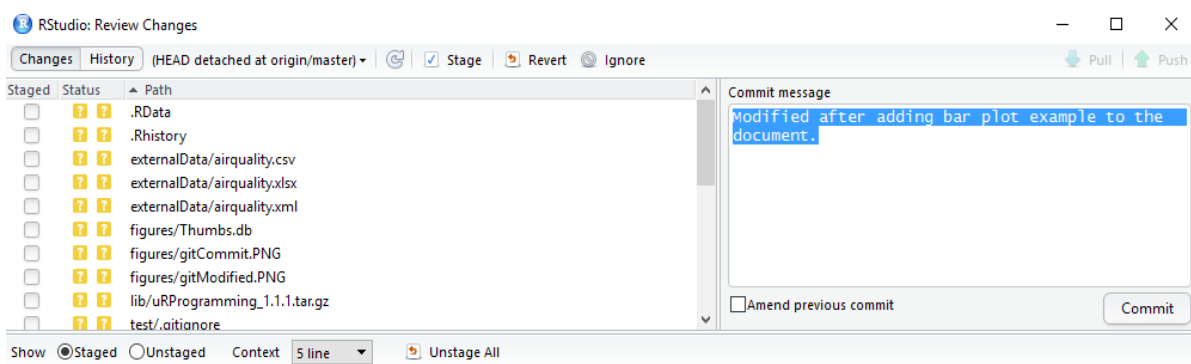


Figure 2.5: Git from Rstudio shows how to commit the modified file.

The source code branching with Git is created either using command line or git GUI like source tree can be used for creating branch and other git functionality.

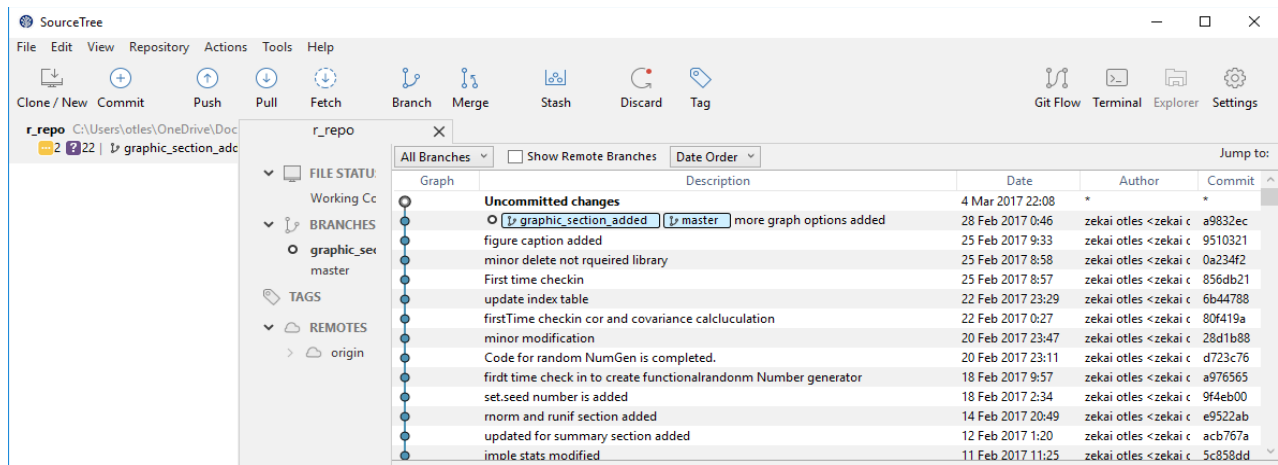


Figure 2.6: Source tree Git shows the created branches

When another branch checked out from R studio, it shows what files are checkout from the branch.



Figure 2.7: Git from Rstudio shows the file from the branch

We can check out a branch and worked on the branch independently. For example, we checkout the codes on ubuntu machine and name that branch as ubuntu. Let's assume, we are working on the codes to make them compatible with ubuntu. The "graphic_section_added" branch pulled from remote repository and created new branch called as "ubuntu" from that branch. After modification made in the repository, the "ubuntu" branch is also *pushed* to the remote branch with

the same name. Once the local branches pushed into the remote repository, they could be used by other people but also can be checkout from another system.

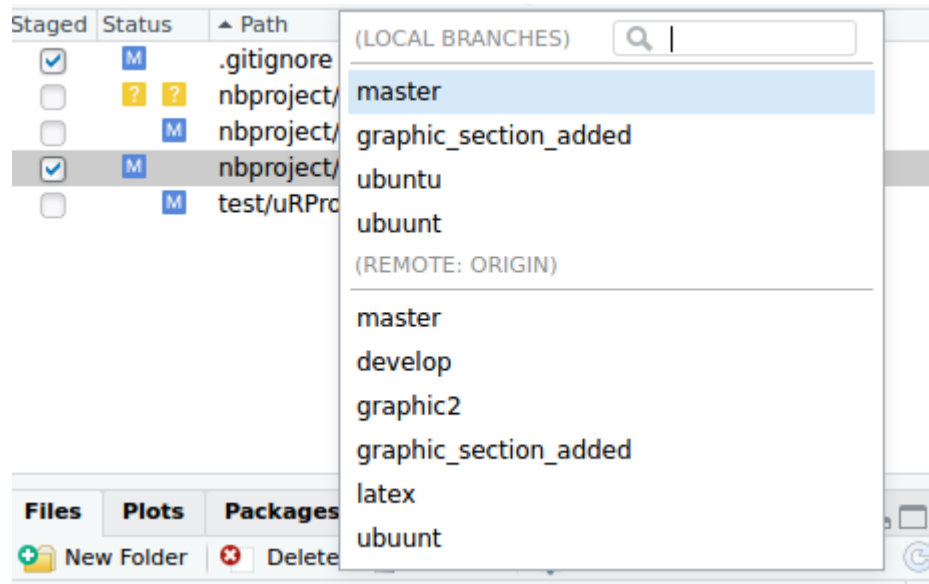


Figure 2.8: Git from Rstudio shows the local and remote branches

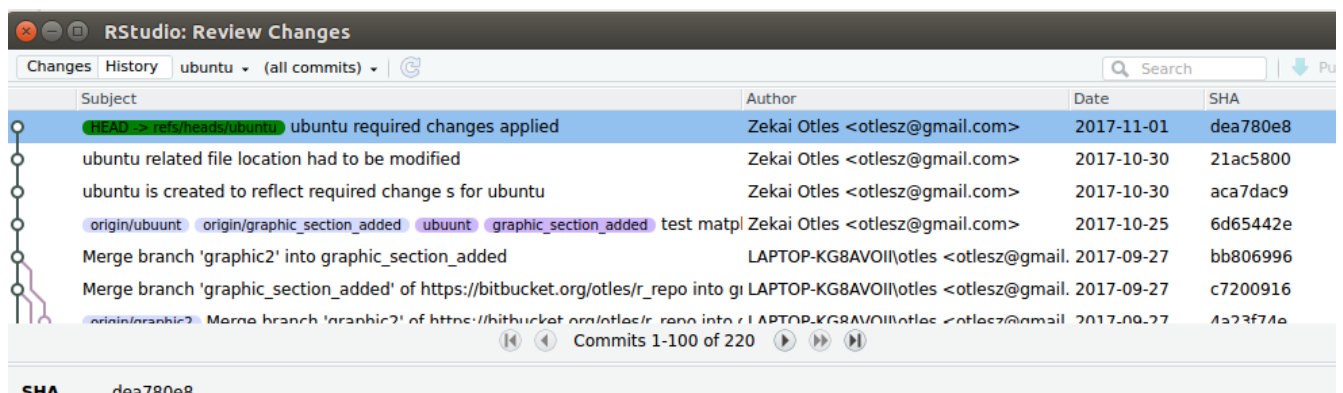


Figure 2.9: Git from Rstudio shows the ubuntu branch

Git also allows us to merge different branches within a local or remote repository. Merging allow us to combine different branches, permit developer freely develop codes in a separate branches until they are ready to finalize the code.

2.3 Using IDE to develop R scripts

Using IDE like R studio or other tools to develop R-script applications allow us to increase our productivity, Here, we decide to use R-studio to demonstrate to develop R-scripts. The integrated environment R-studio gives developer to write

their software, manage versions and build and deploy as an application. We utilized open office and relevant odfeave package to make it continuous integration.

In the following sections, we will demonstrate how we we utilized R-studio in terms of writing R codes, running the R-scripts, creating, and building packages.

2.4 Writing functions

Following R function is a simple example. It calculates air density from ideal gas law (<https://sciencing.com/calculate-air-density-5149935.html>) for given temperature (as Kelvin) at the observed pressure (as mb).

#this function computes air density air based on Temperature (K), Pressure (mb)

#using ideal gas law

#

airdensity<-function(temp=temp, pres=pres) {

#using hydrostatic equation we can approximate the thicknes of the

#atmosphere as m

#temp as K and pressure as mb

Rd=287.04 # Gas constant for dry air m²/s² K

*pres<-pres*100 #converts from mb to pascal*

*airdensity<-pres/(Rd*temp)*

}

Here in this function, air temperature (as degree Kelvin) and air pressure (as mb) are inputs to the function. The function name is defined as air density, if one uses following command to source the function in the R environment. The function will be available for the user until this object explicitly deleted. The air density function for given temperature 313K (29.84 degree Celcius) at 1001 mb calculates the air density as 1.14159(kg/m³).

As readers can see here, using R functions makes the programming more modular and readable.

```
source('~/.R_workspace/r_repo/test/airdensity.R')
```

```
> airdensity
function(temp=temp, pres=pres) {
#using hydrostatic equation we can approximate the
thicknes of the
#atmosphere as m
#temp as K and pressure as mb
Rd=287.04 # Gas constant for dry air m2/s2 K
```

```
pres<-pres*100    #converts to pascal

airdensity<-pres/(Rd*temp)

}
> airdensity(313, 1001)
> rho<-airdensity(313, 1001)
> rho
[1] 1.114159
>
```

2.5 Create R package System

In order to build R package, Rtools needs to be installed (Reference) for windows and Linux. Windows installation requires downloading Rtools.exe for the R version. The determining version of R can be found by typing "version" in R shell window.

In order to build package, following folders needed with necessary files.

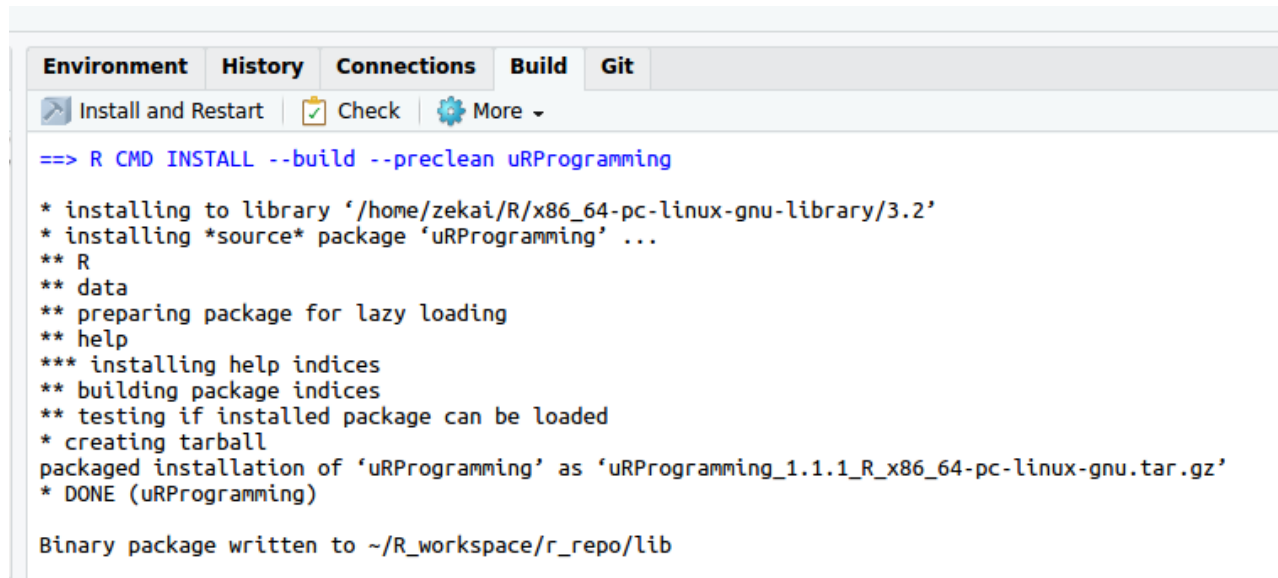
```
0 directories, 0 files
zekai@zoubuntu:~/r_repo/lib/uRProgramming$ tree
.
├── data
│   ├── demData.RData
│   ├── heightData.RData
│   └── synDat.RData
├── DESCRIPTION
├── INDEX
├── man
│   ├── conditionalStatements.Rd
│   ├── createDataFile.Rd
│   ├── dataStructures.Rd
│   ├── dataTypes.Rd
│   ├── environment.Rd
│   ├── legendM.Rd
│   ├── loopStructures.Rd
│   ├── manipulatingData.Rd
│   ├── matPlotM.Rd
│   ├── object.Rd
│   ├── packageBuild.Rd
│   ├── RGUI.rd
│   ├── simpleGraphs.Rd
│   └── simpleStats.Rd
├── NAMESPACE
└── R
    ├── conditionalStatements.R
    ├── corTest.R
    ├── createDataFile.R
    ├── dataStructures.R
    ├── dataTypes.R
    ├── environment.R
    ├── legendM.R
    ├── loopStructures.R
    ├── manipulatingData.R
    ├── matPlotM.R
    ├── object.R
    ├── packageBuild.R
    ├── RGUI.R
    ├── simpleGraphs.R
    └── simpleStats.R

3 directories, 35 files
zekai@zoubuntu:~/r_repo/lib/uRProgramming$
```

Fig. 2.9 *The folder structure and the necessary files to build URProgramming package.*

The uRProgramming package can be build using following command

“R CMD build uRProgramming” in the Linux/UNIX type system, and windows command line is pretty similar. One can build the package using Rstudio see Fig. 2.10.



```
Environment History Connections Build Git
Install and Restart Check More ▾

==> R CMD INSTALL --build --preclean uRProgramming

* installing to library ‘/home/zekai/R/x86_64-pc-linux-gnu-library/3.2’
* installing *source* package ‘uRProgramming’ ...
** R
** data
** preparing package for lazy loading
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded
* creating tarball
packaged installation of ‘uRProgramming’ as ‘uRProgramming_1.1.1_R_x86_64-pc-linux-gnu.tar.gz’
* DONE (uRProgramming)

Binary package written to ~/R_workspace/r_repo/lib
```

Figure 2.10. *The package is built option in R Studio*

Show them how to build package from GUI.

As it is shown here, building R-package is not trivial, it requires package functions and corresponding document files.

In the next section, we will go into **R** language details.

3 Simple Data Types and Objects

R language has following defined data types, Double, Character, Logical and Complex. Numeric variable as a default defined as double, simple text string defined as Character, single character [T and F] is defined as Logical and complex numbers are defined as Complex.

Integer objects are created with integer, as.integer keywords. Similarly numeric objects are created with numeric and as.numeric keywords. Boolean, Double, Character, Logical, Complex objects are created by their types [boolean, double, character, ..] and as.types [boolean, double,..], The variables or vector also can be casted from one type to another type using similar keywords.

```
library(uRProgramming)
dataTypes(24)
dataTypes('ali')
dataTypes(T)
a<-as.integer(24)
dataTypes(a)
```

```
standard output from above lines
[1] "double"
[1] "character"
[1] "logical"
[1] "integer"
```

The following examples show how casting works. Following snippets can be copied and saved as `testDataTypes.R`. This R script should be sourced in the R shell in order to have function `testDataTypes()`, and run the program. The x vector created as sequence of variables between 1 and 100 increment by 10.011. The x is created as default double variable. The x vector is casted to integer y vector. The later part of the example demonstrates, casting x vector into z character vector.

```
testDataTypes<-function(){
#create logical variable
iTest<-T
is.logical(iTest)
if(iTest == is.logical(iTest)) cat('Logical ')
#create simple double vector
x<-seq(1,100,by=10.011)
is.double(x)
if(iTest == is.double(x)) cat('x is double Vector ')
#cast double vector to integer
y<-as.integer(x)
is.integer(y)
if(iTest == is.integer(y)) cat('y is integer Vector ')
#cast to character vector
z<-as.character(x)
is.character(z)
if(iTest == is.character(z)) cat('z is character Vector ')
}
```

Source the above program, and run the program (`testDataTypes()`)

```
TRUE
Logical
1 11.011 21.022 31.033 41.044 51.055 61.066 71.077 81.088 91.099
x is double Vector
1 11 21 31 41 51 61 71 81 91
y is integer Vector
1 11.011 21.022 31.033 41.044 51.055 61.066 71.077 81.088 91.099
z is character Vector
```

4 Control Structures

Control structures are used to change a flow or follow a logic. The control structures easily could make any program more followable for maintenance and understanding. Control structures are very straight forward when used in a modular way in a smaller number of statements or calling other reusable

functions.

In the following sections, the usage of if block will be explained with examples. The samples should help reader to solidify the understand **if** control structures.

4.1 If statements

Let's use simple vector which contains only numbers, check if numbers is equal to specific value assign iFound as True.

If (Statements) { Assignment}

```
x<-c(0,3,6,9,12,15,18,21,24)
if(x[which(x==6)]==6){iFound<-TRUE}
print(iFound)
[1] TRUE
```

In the above example, which finds the index number for the value. The usage of **which** in vector is pretty powerful as shown in the example. The *which* statement is handy determining the index of vector or array. When the vector with right index with exact value found, iFound assigned to be *true*.

Logical operators in R are

Operator	Description
==	Equal to
!=	Not equal to
<	Less than
<=	Less or equal to
>	Greater than
>=	Greater or equal to
&	And
	Or

Following code snippet is simple example of how to use logical operators. The first example shows *equal* operator. The second example demonstrates *less than or equal* to operator and assigning the value based on the result. The third example shows *greater than* comparison. The last example is combination *greater than, and less than or equal* to.

```
Ex. 1
if(demDat[indx,1] ==1 ) {demDat[indx,1]<- 'M'}
```

```
Ex.2
if(demDat[indx,2] <=25 )demDat[indx,2]<- 'Y'
```

```
Ex.3
```



```
if(demDat[indx,2] > 50 ) {demDat[indx,2]<-'0'}
```

Ex.4

```
if(demDat[indx,2] > 25 & demDat[indx,2] <=50 )demDat[indx,2]<-'M'
```

4.2 If then Else statements

The “If then else block” is used when comparison expected to yield either true or false result. It gives the programmer, the power to control logical flow. In R programming, if then else blocks are in the following format

```
If (Statements) { Assignment}
```

```
else (statements) {Assignment}
```

Here is a simple code snippet demonstrates **if conditional** block.

```
if(onePlot)
{
    iSelect=1
else{
    iSelect=2
}
```

In the above code snippet, if the value of onePlot is true, the variable **iSelect** will be equal to 1. The **iSelect** will be equal to 2 when the value of onePlot is false.

4.3 Switch statements

One can use switch statement for multiple option conditional statements. The switch statement evaluates according to the first argument.

```
switch(Expression,List)
```

The first argument determines how the results will be evaluated. The first line shows a number in the first argument will select the corresponding list element, in the second line of code snippet shows the the corresponding character strings of list element.

```
> switch(3, 1.5, 2, 300, 4)
[1] 300
> switch("iki", bir = "bir", iki = "Two", uc = "uc")
[1] "Two"
```

Following function demonstrates how simple “if then else block” could be replaced by modular switch statements, this type of example can be extended to more modular functionality.

```
#Simple example for data types casting using swith statement
```

```

usingSwitch<-function(x,dataTypes){
  switch (dataTypes,
    integer = {x<-as.integer(x)},
    double = {x<-as.double(x)},
    character = {x<-as.character(x)}
  )
}
> x <- seq(1, 100, by = 10.011)
> x
[1] 1.000 11.011 21.022 31.033 41.044 51.055 61.066 71.077 81.088 91.099
> xC <- usingSwitch(x, "character")
> xC
[1] "1"      "11.011" "21.022" "31.033" "41.044" "51.055" "61.066" "71.077"
"81.088" "91.099"
> xI <- usingSwitch(x, "integer")
> xI
[1] 1 11 21 31 41 51 61 71 81 91

```

5 Loop Structures

The loop structures are needed to access elements of vectors, matrix, list, and data frame. The access of the data element is crucial to manipulate the elements. For and *while* loop are used to reach data elements.

5.1 For loop

The for loop is used for sequentially access the data elements. Each element of data structures can be reached, used, or manipulated within the *for* loop. The *for* loop is expressed in the following format.

```

for (values in sequence) {
  statement
}

```

The following code snippets show how *for* loop is used. In the following example, 100 number with specified seed assign to vector. This vector has only integers for demonstration purpose.

```

set.seed(312)
#create 100 random number between 0 and 1000
xInt<-floor(runif(100,0,1000))
for(indx in 0:length(xInt) ){
  if(indx %%10 ==0){print(xInt[indx])}
}
numeric(0)
[1] 450
[1] 261
[1] 405
[1] 295
[1] 763
[1] 356
[1] 833

```

```
[1] 329
[1] 419
[1] 948
```

5.2 While loop

The access of certain elements could be controlled by using while loop, the expression determines executing statements or not within while loop block.

```
while(expression){
  statement
}
```

In the following example, the elements between two indices are assigned as either *even* or *odd*. Execution will stop when expression result is false, otherwise statements within the block is executed. The while block will stop when index reaches the value length of xInt vector which is 100 in this example. The index value has initial value 80, then index incrementally increased in the while block.

```
set.seed(312)
#create 100 random number between 0 and 1000
xInt<-floor(runif(100,0,1000))
index<-80
while(index<=length(xInt)){
  xInt[index]<-ifelse(as.integer(xInt[index]) %%2==0, "Even", "Odd")
  if(index %% 4 == 0 ){print(xInt[index])}
  index<-index+1
}
[1] "Odd"
[1] "Even"
[1] "Odd"
[1] "Odd"
[1] "Odd"
[1] "Even"
```

6 Data Structures

In my experience as software engineer and data analyst, introduction to database and utilizing with the data structures would be valuable experience in terms of programming, specifically storing and retrieving data from database is a great experience for the structured programming. Thus, we introduce the database in this section. The MySQL was chosen because of simplicity to installation and free to use. The community version is freely available to use, it can be downloaded from oracle. The RMySQL package allows us to utilize some of database functionality within R scripts. Available datasets from basic R installation will be loaded to the urprogramming schema into MySql database. Later those dataset tables will be used to demonstrate to basic data structures like vector, list, and matrix.

Insert air quality data from R datasets into air quality table into the schema. Later, selected data from mysql table will be converted into vector and list, etc.

```
#insert data into "airquality" table
```

```

insertData<-function(){
library("RMySQL")
#if(con)dbDisconnect(con)

numR<-dim(airquality)[1]
#con <- dbConnect(RMySQL::MySQL(), dbname = "urprogramming",password="urprogram-
ming")

con <- dbConnect(RMySQL::MySQL(), user = 'rprogrammer', password = 'uRProgram-
ming', host = '127.0.0.1' , dbname = 'urprogramming')

results<-dbSendQuery(con, "SELECT * FROM airquality")
dataFrm<-dbFetch(results)

#dbFetch(results)
print(dim(dataFrm)[1])
#print(dataFrm[1:5,])
iSkip<-FALSE
if(dim(dataFrm)[[1]] > 1){iSkip<-FALSE
}else {
  iSkip<-TRUE
}

if(iSkip){
  for (iRow in 1:numR) {

# print(iRow)
data<-airquality[iRow,]

data[is.na(data)]<-999999
statement<-paste("INSERT INTO ", "airquality")

# print(statement)
insertQuery <- paste(paste(statement,"VALUES("), paste(data, collapse = ", "),
")")
#
print(insertQuery)
dbGetQuery(con,insertQuery)
}
}#end of iSkip
dbDisconnect(con)

}

```

In order to load air quality data to the table in MySql database, connection to database will open with following statement.

```

con <- dbConnect(RMySQL::MySQL(), user = 'rprogrammer', password =
'uRProgramming', host = '127.0.0.1' , dbname = 'urprogramming')

```

The dbConnect statement in RMySQL library requires name of database, schema, username and password. Once the connection is established, the connection

object is used to populate the rows in table using the statements starting with "INSERT INTO airquality VALUES(,,,,)".

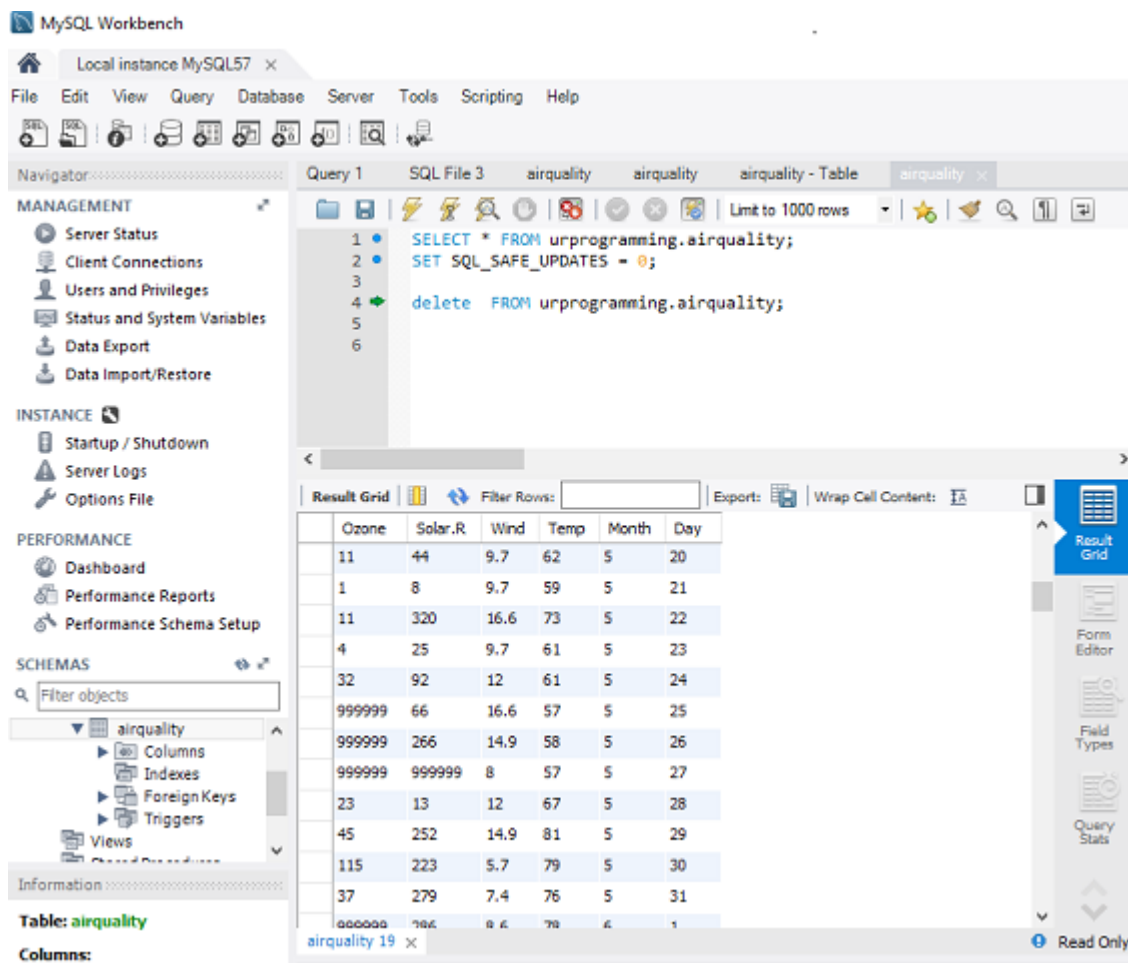
```
> insertData()  
[1] 153  
[1] TRUE  
> results <- dbSendQuery(con, "SELECT * FROM airquality")  
> dataFrm <- dbFetch(results, )  
> print(dim(dataFrm))  
[1] 153 6  
> print(dataFrm[1:5, ])  
   ozone  solar wind temp Month day  
1    41   190  7.4   67     5    1  
2    36   118  8.0   72     5    2  
3    12   149 12.6   74     5    3  
4    18   313 11.5   62     5    4  
5 999999 999999 14.3   56     5    5
```

The following statement will select each row and form a result set.

```
results<-dbSendQuery(con, "SELECT * FROM airquality")
```

The result set will be loaded to dataframe using *dbFetch(results,)* function from the RMySQL library. The *insertData* function is configured so the records will only be inserted when there is no data in the table. Here, we introduced the basic functionality of database in R. Similar to *insert and select*, *update*, *truncate* could be used to manipulate data in tables. There are many options for SQL statements to create complex joins or selecting the subsets of the data. This kind of SQL statements can be found in MySQL documentations, that's why we did not want to go beyond the basic of SQL statements.

The following figure shows the instance from MySQL Workbench. The tool can be used to execute SQL statements.



Similar to data stored in databases, one could easily store data as flat ascii files. Those simple flat data files can be created from either simple editor or spreadsheet programs. R has extremely easy interfaces to read flat files or spreadsheet data. The flat file stored in `"/home/zekai/R_workspace/r_repo/externalData/airquality.csv"` can be read with following statement

```
read.csv(file="/home/zekai/R_workspace/r_repo/externalData/airquality.csv",header=TRUE,sep=",")
```

If data is in spreadsheet, it can be saved as flat file from spreadsheet then may be read data from flat file using similar statement. The data file could be read using read.csv function from R util package. There are other available packages for reading spreadsheet data to R.

6.1 Data Frame

Data frame is one of the available data structures in R. The data frame is similar to the spreadsheet data, one can easily see the structure of the data. Once the data saved in data frame structure, the structure of data, dimensions and contents of data is readily available in R. The structure of data is determined from **str(name of data frame object)** as shown in the following example

str(dataFrm).

The statement gives the structure of data, name of data variables and type of variable stored. As seen in the example, air quality data store Ozone, Solar.R, Temp, Month, Day variables as **int** and wind as **num**. These variables can be changed using casting as we shown earlier in Simple Data Type section. We can figure out how data stored in a data frame using existing function **dim** in R base functions, **colnames** (again from R base functions) refers the variable names stored in data structures. The **View** function in R util package can display the data frame contents so we can have some idea about the data. The following section

View(dataFrm[1:4,])

is displaying the data from the first through fourth row in dataFrm.

```
results<-dbSendQuery(con, "SELECT * FROM airquality")
dataFrm<-dbFetch(results)
the results from select assigned to dataFrm. The dataFrm has following structure
str(dataFrm)
'data.frame': 153 obs. of 6 variables:
 $ ozone: num 41 36 12 18 999999 ...
 $ solar: num 190 118 149 313 999999 ...
 $ wind : num 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ temp : num 67 72 74 62 56 66 65 59 61 69 ...
 $ Month: int 5 5 5 5 5 5 5 5 5 5 ...
 $ day : int 1 2 3 4 5 6 7 8 9 10 ...
dim(dataFrm)
[1] 153 6
colnames(dataFrm)
[1] "ozone" "solar" "wind" "temp" "Month" "day"
View(dataFrm[1:4,])
  ozone solar wind temp Month day
1    41   190  7.4   67     5    1
2    36   118  8.0   72     5    2
3    12   149 12.6   74     5    3
```

6.2 Vector

Vector is used for atomic data types (integer, numeric, complex, and character) and list storage. It is a powerful way to store data and its usage in the computations extremely simplifies the programming. The examples will demonstrate the power of vector calculations. Temperature from Fahrenheit is easily converted to Celsius with following statement.

tempAsCels<-5*(dataFrm\$Temp-32)/9.0

```
> tempAsCels <- 5 * (dataFrm$Temp - 32)/9
> cat("is.vector(tempAsCels)")
is.vector(tempAsCels)
> print(is.vector(tempAsCels))
[1] TRUE
> cat("View(tempAsCels(1:5))")
View(tempAsCels(1:5))
> print(tempAsCels[1:5])
[1] NA NA NA NA NA
```

It is also easy to dissect and examine data stored in vector format. The following simple snippet will demonstrate how to retrieve the month data from the dataFrm object. We have determined in a month which days are 80 degree Fahrenheit. It also shown number of 80-degree days for our data.

```
dataFrm$Month[dataFrm$Temp==80]
length(dataFrm$Month[dataFrm$Temp==80])
integer(0)
[1] 0
```

The **which**, function is available from the R base package, is used to determine the indices of the vector when the condition meets up the criteria. The following snippets show which element of the temperature vector was exactly 80 degrees Fahrenheit.

```
which(dataFrm$Temp==80)
integer(0)
```

6.3 Matrix

The matrix is available in R to store two-dimensional **array** in a matrix format. The discussion of matrix without **array** will not be complete to understand data structures in **R** programming. The array can store vector data in one, two or dimensional format. The vector data is stored without array bound information, but **array** and **matrix** need complete information about dimensions.

The **array** needs the information about data types and dimensions of the array. The **array** stores one to many dimensional data. In the following example, row_names stored in a one dimensional **array**.


```
row_Names<-array("",dim=length(unique(dataFrm$Month)))
```

Structure of **array** can be found using **str** from util package in R. The **str** function will give data types and dimension of the **array**. The following demonstrates how to use **str** function with **array**.

```
str(row_Names)
```

Another useful function to check whether data stored in array or not. If the data stored in an array, `is.array(row_Names)` will return *TRUE*.

The **matrix** is a special case of **array** which is a two dimensional data structure. The **matrix** needs data types, row and column dimensions. The tempAsMatrix is defined as **matrix** in the following example.

```
tempAsMatrix<-matrix("",nrow=length(unique(dataFrm$Month)),ncol=30)
```

```
#form a Matrix from dataFrm data
#the purpose of this function to ceate Temp matrix
formTempMatrix<-function(dataFrm)
{
  monthAsNum<-unique(dataFrm$Month)
  row_Names<-array("",dim=length(unique(dataFrm$Month)))
  tempAsMatrix<-matrix("",nrow=length(unique(dataFrm$Month)),ncol=30)

  for(mIndx in 1:length(monthAsNum)){
    row_Names[mIndx]<-
switch(as.character(monthAsNum[mIndx]), "5"="May", "6"="June", "7"="July", "8"="Aug", "
9"="Sep")
  }
  rownames(tempAsMatrix)<-row_Names

  length(unique(dataFrm$Month))
  rowIndex<-unique(dataFrm$Month)
  for(index in 1:length(rowIndex)){

    x<-dataFrm$temp[dataFrm$Month==rowIndex[index]]
    tempAsMatrix[index,]<-x[1:30]
  }

  return(tempAsMatrix)
}
```

The **str** function for matrix type will return data types and bounds of each dimensions. The base package has a function to determine whether data structure is matrix or not. In the following example,

```
is.matrix(tempAsMatrix)
```

the return value will be *TRUE*.

6.4 List

The list is another very powerful data structure, heterogeneous data can be stored in a list. The list can contain list, data frame, array and matrix in a structure. The *dataFrm*, *row_Names* and *tempAsMatrix* data structures are all stored in a **list** called *aList*. The data structure can be retrieved using the *list\$name* structure, such as *aList\$dataFrm*.

```
#demonstrate the list
useList<-function(dataFrm,row_Names,tempAsMatrix){
  aList<-list(dataFrm=dataFrm,rowNames=row_Names,matrix=tempAsMatrix)
  str(aList)
  return(aList)
}

str(aList)
List of 3
 $ dataFrm : 'data.frame': 153 obs. of 6 variables:
  ..$ Ozone   : int [1:153] 41 36 12 18 999999 28 23 19 8 999999 ...
  ..$ Solar.R: int [1:153] 190 118 149 313 999999 999999 299 99 19 194 ...
  ..$ Wind    : num [1:153] 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
  ..$ Temp    : int [1:153] 67 72 74 62 56 66 65 59 61 69 ...
  ..$ Month   : int [1:153] 5 5 5 5 5 5 5 5 5 5 ...
  ..$ Day     : int [1:153] 1 2 3 4 5 6 7 8 9 10 ...
 $ rowNames: chr [1:5(1d)] "" "" "" "" ...
 $ matrix  : chr [1:5, 1:30] "" "" "" "" ...
```

As seen above, the list data structure is very convenient way to store multiple data structures into a data structure.

7 Simple Statistical Analysis

This section is simple introduction to statistical analysis. There will be brief explanation about creating normally distributed variables used for statistical analysis, creating simple summary mean, standard deviation. The simple statistical functions are available on stats, base, time functional packages. The functional examples will show how to calculate simple statistics using already defined data structures. Introduction how to use simple statistics will give foundation to extend the learners ability to do more complex statistical analysis. This will also help us utilizing the other packages in available by R community.

```
#demonstrate simple distribution and some statistics
simpleStats<-function(n,mean,sd){
  x<-rnorm(n,mean=mean,sd)
  #mean of variable
  mean=mean(x)
  ave=ave(x)
  max=max(x)
  min=min(x)
  median=median(x)
  sd=sd(x)
  sum=sum(x)
```

```
summary=summary(x)
return(list(mean=mean,ave=ave,max=max,min=min,median=mean,sd=sd,summary=summary,sum=sum))
}
```

In the example, normally distributed random variable created using **rnorm** function from **stats** package. The **mean**, **summary** and **sd** are some of the available functions from **base** and **stats** packages. The **mean** from **base** package gives the arithmetic mean of the values. The **min** and **max** returns minimum and maximum values for the input values. The **sd** is standard deviation of the input values. The summary function from base package returns **min** and **max** values, the first and third quartiles, **median** and **mean** of input values.

```
xStats<-simpleStats(100,273.16,15)
str(xStats)
List of 8
 $ mean      : num 273
 $ ave       : num [1:100] 273 273 273 273 273 273 ...
 $ max       : num 304
 $ min       : num 232
 $ median    : num 273
 $ sd        : num 14.4
 $ summary:Classes 'summaryDefault', 'table' Named num [1:6] 232 264 273 273 283 ...
 .. ..- attr(*, "names")= chr [1:6] "Min." "1st Qu." "Median" "Mean" ...
 $ sum       : num 27295
```

The **runif** generates uniformly distributed random numbers. The random number generators ran four times in the example and selected the fifth element of the storage vectors (**xTempNorm** and **xTempUni**). Since the script is not using fixed seed number for the random number generators, the storage vectors will consist of different elements as we shown here with the fifth elements of the vectors.

```
#Create randomly number generated vectors
#simple statistics
randomNumGen<-function(seed=FALSE){
print("Normally distributed Random Number")
for(indx in 0:3){
  print(paste("Iteration ",indx,sep= ""))
  if(seed) set.seed(6543)
  xTempNorm<-rnorm(100,273.16,15)

  print(paste("xTempNorm[5]=",xTempNorm[5]))
  cat('summary(xTempNorm)\n')
  print(summary(xTempNorm))
}

print("Uniformly distributed Random Number")
for(indx in 0:3){
  if(seed)set.seed(6543)
```

```

xTempUni<-runif(100,240,320)
print(paste("Iteration ",indx,sep= ""))
print(paste("xTempUni[5]=",xTempUni[5]))
cat('summary(xTempUni)\n')
#

print(summary(xTempUni))

}

return(list(xTempNorm=xTempNorm,xTempUni=xTempUni))
}

```

```

[1] "Normally distributed Random Number"
[1] "Iteration 0"
[1] "xTempNorm[5]= 282.506660883275"
summary(xTempNorm)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 241.3  261.6   274.4   273.5   284.2   321.4
[1] "Iteration 1"
[1] "xTempNorm[5]= 266.941109367306"
summary(xTempNorm)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 236.3  260.4   274.3   272.8   284.6   314.1
[1] "Iteration 2"
[1] "xTempNorm[5]= 261.022663275379"
summary(xTempNorm)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 235.5  261.8   271.0   271.1   280.9   305.0
[1] "Iteration 3"
[1] "xTempNorm[5]= 279.732265995178"
summary(xTempNorm)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 235.7  262.1   274.4   273.2   282.8   304.0
[1] "Uniformly distributed Random Number"
[1] "Iteration 0"
[1] "xTempUni[5]= 245.185738392174"
summary(xTempUni)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 240.4  256.4   274.5   277.0   299.3   319.5
[1] "Iteration 1"
[1] "xTempUni[5]= 302.323486860842"
summary(xTempUni)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 240.6  255.8   271.9   277.4   299.8   319.9
[1] "Iteration 2"
[1] "xTempUni[5]= 266.91300611943"
summary(xTempUni)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 240.3  258.8   278.9   279.3   297.9   319.0
[1] "Iteration 3"
[1] "xTempUni[5]= 242.070628292859"
summary(xTempUni)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 240.4  256.3   277.7   277.9   301.1   319.9

```

The same example will be repeated for fixed seed numbers for the random number generation. When we set the seed the numbers to produce random number vectors using **rnorm** and **runif**, they will always contains same elements. The fifth elements for the **xTempNorm** and **xTempUni** vectors is always same for the given seed number 6543. The **summary** of the storage **xTempNorm** and **xTempUni** vector will yield same results because of the same seed number.

```
[1] "Normally distributed Random Number"
[1] "Iteration 0"
[1] "xTempNorm[5]= 288.495445822198"
summary(xTempNorm)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
221.5  259.9  273.3   270.8  281.0   304.0
[1] "Iteration 1"
[1] "xTempNorm[5]= 288.495445822198"
summary(xTempNorm)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
221.5  259.9  273.3   270.8  281.0   304.0
[1] "Iteration 2"
[1] "xTempNorm[5]= 288.495445822198"
summary(xTempNorm)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
221.5  259.9  273.3   270.8  281.0   304.0
[1] "Iteration 3"
[1] "xTempNorm[5]= 288.495445822198"
summary(xTempNorm)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
221.5  259.9  273.3   270.8  281.0   304.0
[1] "Uniformly distributed Random Number"
[1] "Iteration 0"
[1] "xTempUni[5]= 272.028888594359"
summary(xTempUni)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
240.0  262.8  281.6   280.5  297.3   318.4
[1] "Iteration 1"
[1] "xTempUni[5]= 272.028888594359"
summary(xTempUni)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
240.0  262.8  281.6   280.5  297.3   318.4
[1] "Iteration 2"
[1] "xTempUni[5]= 272.028888594359"
summary(xTempUni)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
240.0  262.8  281.6   280.5  297.3   318.4
[1] "Iteration 3"
[1] "xTempUni[5]= 272.028888594359"
summary(xTempUni)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
240.0  262.8  281.6   280.5  297.3   318.4
```

The xRandGenT list is used to sample two vector datasets. The sample method from base R package used to select 15 random elements from vector datasets . Once the samples assigned the correlations values computed by using different methods available from **stats** package. The covariance between

xTempNorm and **xTempUni** vector datasets computed using **cov** function from **stats** package. The standard deviations calculated using **sd** function from the **stats** package.

```
corTest<-function(xVals=xVals,yVals=yVals){
  x<-sample(xVals,size=15)
  y<-sample(yVals,size=15)
  #cor(x,y,use="everything",method=c("pearson","kendall","spearman"))
  cor.test(x,y,method="pearson")
  print(cor.test(x,y,method="pearson"))
  sdXsdY=sd(x)*sd(y)
  pearsonCor=cov(x,y,method="pearson")/sdXsdY
  print(paste("Pearson=",pearsonCor))

  cor.test(x,y,method="spearman")
  print(cor.test(x,y,method="spearman"))

  cor.test(x,y,method="kendall")
  print(cor.test(x,y,method="kendall"))

}
```

```
corTest(xRandGenT$xTempNorm,xRandGenT$xTempUni)
```

Pearson's product-moment correlation

```
data:  x and y
t = -0.7144, df = 13, p-value = 0.4876
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
-0.6426395  0.3530540
sample estimates:
      cor
-0.1943601
```

```
[1] "Pearson= -0.194360080385567"
```

Spearman's rank correlation rho

```
data:  x and y
S = 766, p-value = 0.1779
alternative hypothesis: true rho is not equal to 0
sample estimates:
      rho
-0.3678571
```

Kendall's rank correlation tau

```
data:  x and y
T = 40, p-value = 0.2395
alternative hypothesis: true tau is not equal to 0
sample estimates:
      tau
```

-0.2380952

8 Graphic Outputs

This section is designed to be how to represent data output in a graphical format.

The visual presentation of data will help audience form an opinion much faster in human's brain. This might be very well represented by the commonly known phrase "A picture is worth a thousand words"

(<http://www.phrases.org.uk/meanings/a-picture-is-worth-a-thousand-words.html>).

This is mind, we will show examples of histogram, bar plot, plot, matplot is readily available R **graphics** package.

In the following example, we used normal distribution function (**rnorm(num,mean=p,sd=sd)** calculated set of numbers (100) for different prescribed mean and standard deviation values. The values plotted using R hist function (**hist(x,probability=T,main=titleOfPlot)**). The calculated density (dnorm) using curve function is plotted on top of the histogram plots. The following sections with mean value (50) and four different standard deviations values shown here.

```
#The purpose of this test to see effect of number of Normal distribution
#Normal distribution is back bone statistical theory
#to run this
#testNormal(numSet=100,mean=c(5,15,50),sd=c(10,5,3,1))
#num number of random observations
normalDist<-function(numSet=numSet,meanSet=meanSet,sdSet=sdSet){

  #opar<-par(ask=interactive())
  opar<-par(font.main=1,ps=9)
  layout(matrix(c(1,2,3,4),2,2,byrow=TRUE))
  for(numIndx in 1:length(numSet)){
    num<-numSet[numIndx]
    for(pIndx in 1:length(meanSet)){
      p=meanSet[pIndx]
      for(i in 1:length(sdSet)){

        x<-rnorm(num,mean=p,sd=sdSet[i])
        iqr<-(quantile(x)[4]-quantile(x)[2])
        psd<-iqr/1.35

        if(sd(x)<psd){
          tail="light"
        }else{
          tail="heavy"
        }
        titleOfPlot<-paste(num,"obs ",sep=" ")
        titleOfPlot<-
        paste(titleOfPlot,paste("mean",sprintf("%4.2f",p),sep="="),sep=" ")
        titleOfPlot<-
        paste(titleOfPlot,paste("sd",sprintf("%4.2f",sdSet[i]),sep="="),sep=" ")
        titleOfPlot<-paste(titleOfPlot,paste("tail",tail,sep="="),sep=" ")
      }
    }
  }
}
```

```

hist(x,probability=T,main=titleOfPlot)
#xVals<-0:nset[i]
#points(xVals,dbinom(xVals,nset[i],p),type="h",lwd="3")
#points(xVals,dbinom(xVals,nset[i],p),type="p",lwd="3")

      curve(dnorm(x,mean=p,sd=sdSet[i]),add=T)
      points(quantile(x)[3],dnorm(quantile(x)
[3],mean=p,sd=sdSet[i]),type="h",lwd="3")
      points(quantile(x)[2],dnorm(quantile(x)
[2],mean=p,sd=sdSet[i]),type="h",lwd="2")
      points(quantile(x)[4],dnorm(quantile(x)
[4],mean=p,sd=sdSet[i]),type="h",lwd="2")
    }
  }
  #frame()

}
par(opar)

  #print(x)
  return(as.numeric(x))
}

```

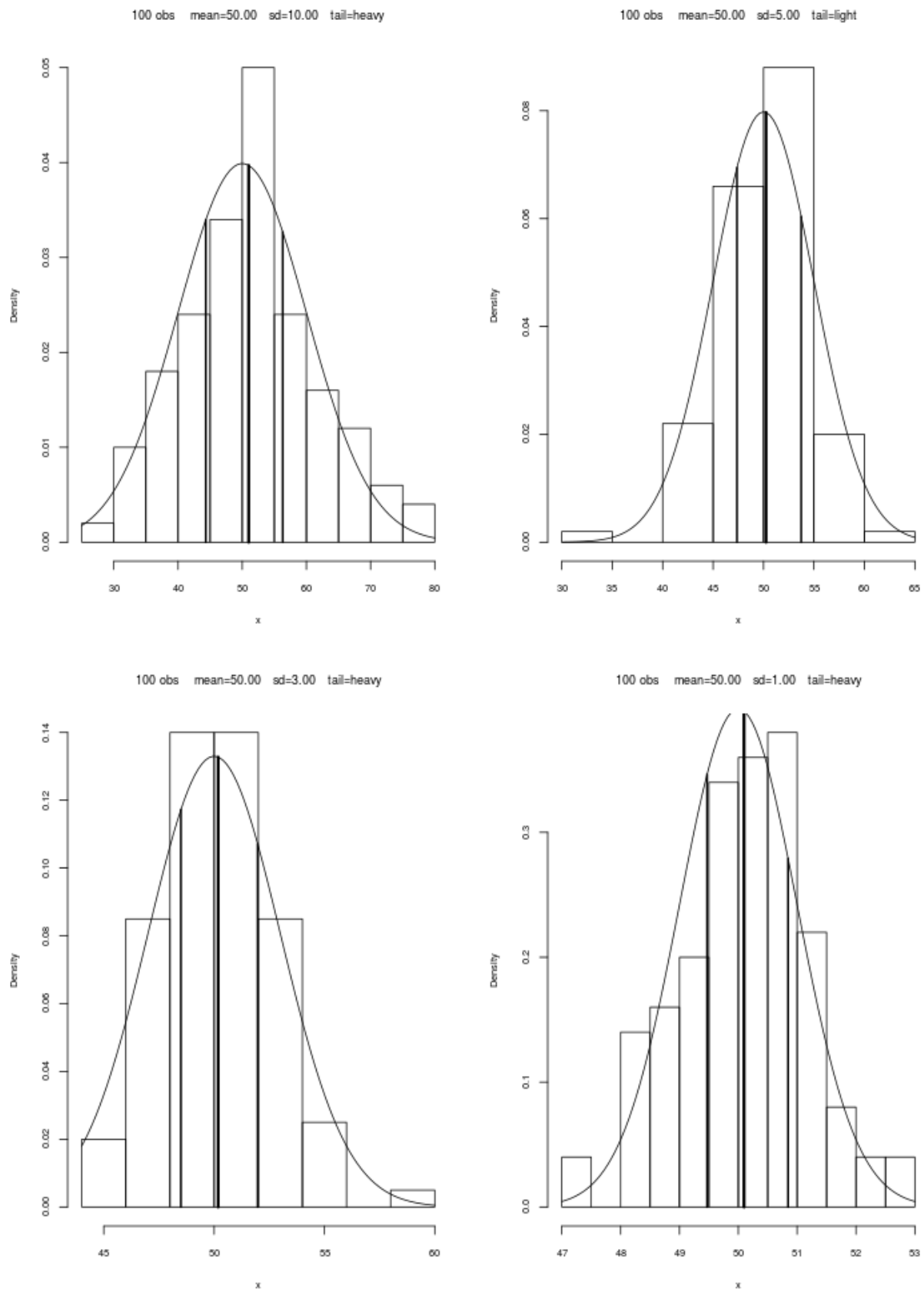



Figure 0 : Example Histogram

In the next section, we utilized some sample temperature data to demonstrate barplots. The temperature occurrence within each of 10 categories ("55-60","60-65","65-70","70-75","75-80","80-85","85-90","90-95","95-100","100-105") counted during a month (May, June, July, August and September) and plotted as barplot. We also plotted the temperature range occurrence for combined month values. The last barplot example demonstrates each temperature range occurrence for all of the months combined. The first bar shows May, and the last bar shows the September temperature range occurrence (the temperature range labeled under bar plots).

```
#####
# Function freqPlot
#####
#The purpose of these function to teach some statistics and demonstrate R
#functionality
#This function plots barplots for for given matrix data,
#with bar stacked on each other or bars side of each other
#different pattern style filling is also an option
#This function written by Zekai Otles otlesz@gmail.com

freqPlot<-function(mat=freqData,xlabels=NULL,ylabels=NULL,onePlot=FALSE,beside-
OrNot=TRUE,isCol=TRUE,titleOfPlot=NULL){

#num of Data Columns besides the first column with identifies common data variable
#in this case is plate number
numRows<-dim(mat)[1]
#layout for the plots
layOutSetUp(1)
if(!onePlot)
{
  layOutSetUp(numRows-1)}
else{
  layOutSetUp(2)
}

colArray<-array("",dim=numRows)
if(isCol & onePlot){
  colArray<-c("#00007F", "blue", "#007FFF", "cyan","#7FFF7F", "yellow", "#FF7F00",
"red", "#7F0000")
}else{
  colArray="gray"
}
titleOfP<-paste(titleOfPlot, "Months")
if(!onePlot){

  for(index in 2:numRows){

    titleOfP<-paste(titleOfPlot,rownames(mat)[index])

    bp<-barplot(as.numeric(mat[index,]),xlab=xlabels,ylab=ylabels,col=colArray)
    axis(side=1, at=bp, labels=names(mat[index,]))
    title(titleOfP)
    box()
  }#end of for loop for index
}else {
  #barplot(mat,xlab=xlabels,ylab=ylabels,beside=besideOrNot,col=colArray[1:num-
Rows])
}
```

```

barplot(mat,beside=besideOrNot,xlab=xlabels,ylab=ylabels,col=colArray)
  title(titleOfP)

  box()
}
}#end of function

```

	55-60	60-65	65-70	70-75	75-80	80-85	85-90	90-95	95-100	100-105
May	8	7	9	4	1	1	0	0	0	0
June	0	1	1	5	13	5	3	2	0	0
July	0	0	0	2	1	17	7	3	0	0
Aug	0	0	0	2	9	6	9	2	2	0
Sep	0	2	5	6	8	4	1	4	0	0

Sample shows how freqplot

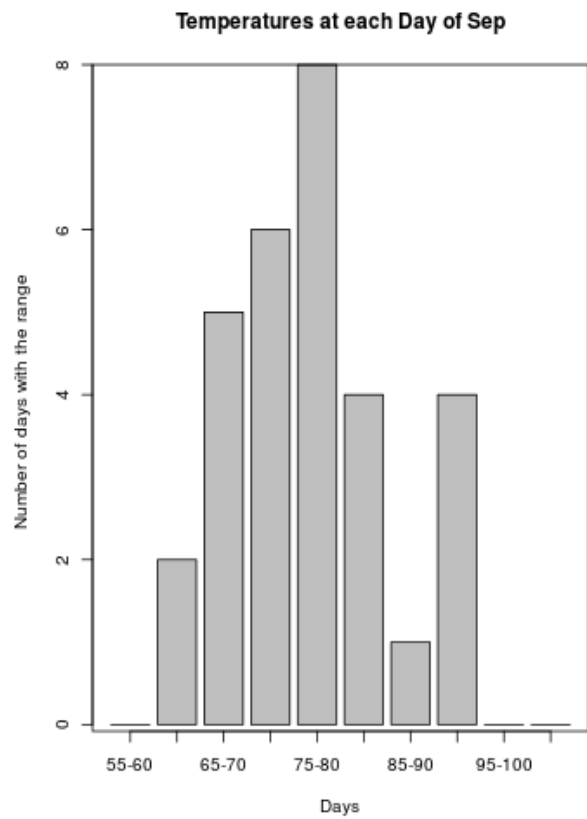
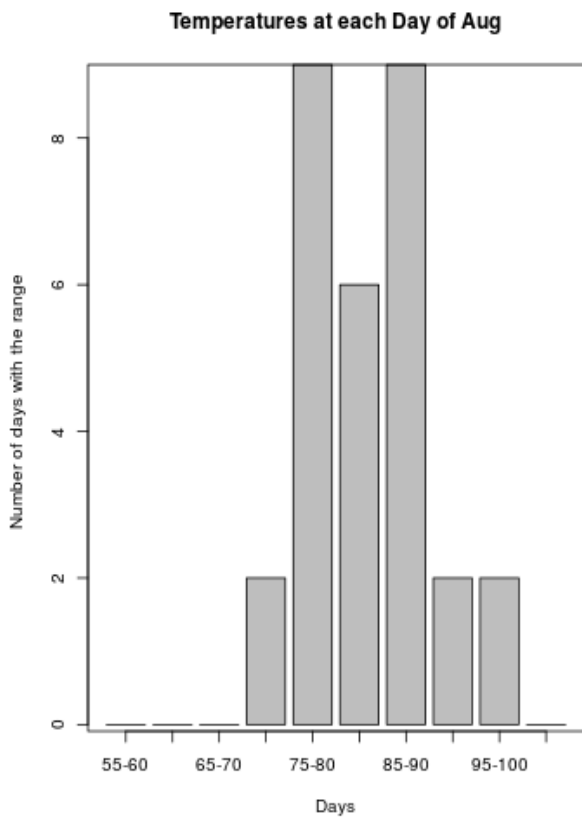
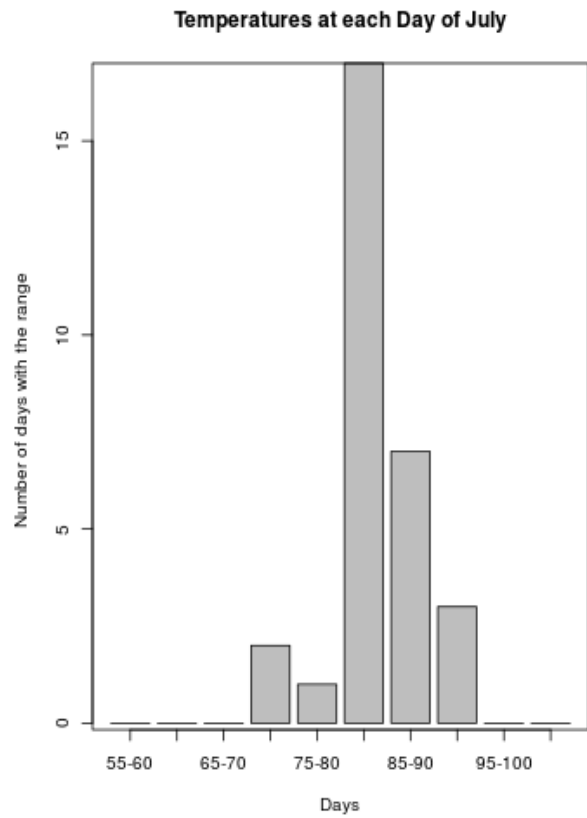
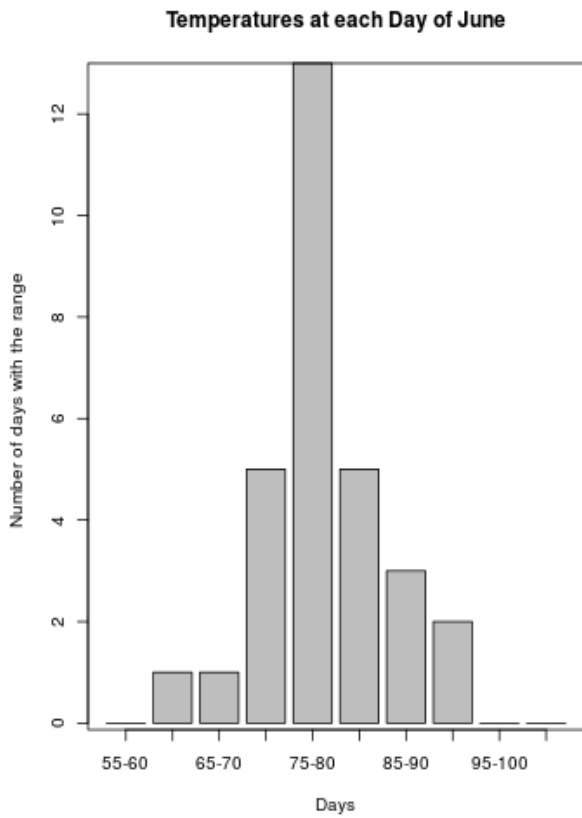


Figure 0 : Example Bar Plot

bar plot

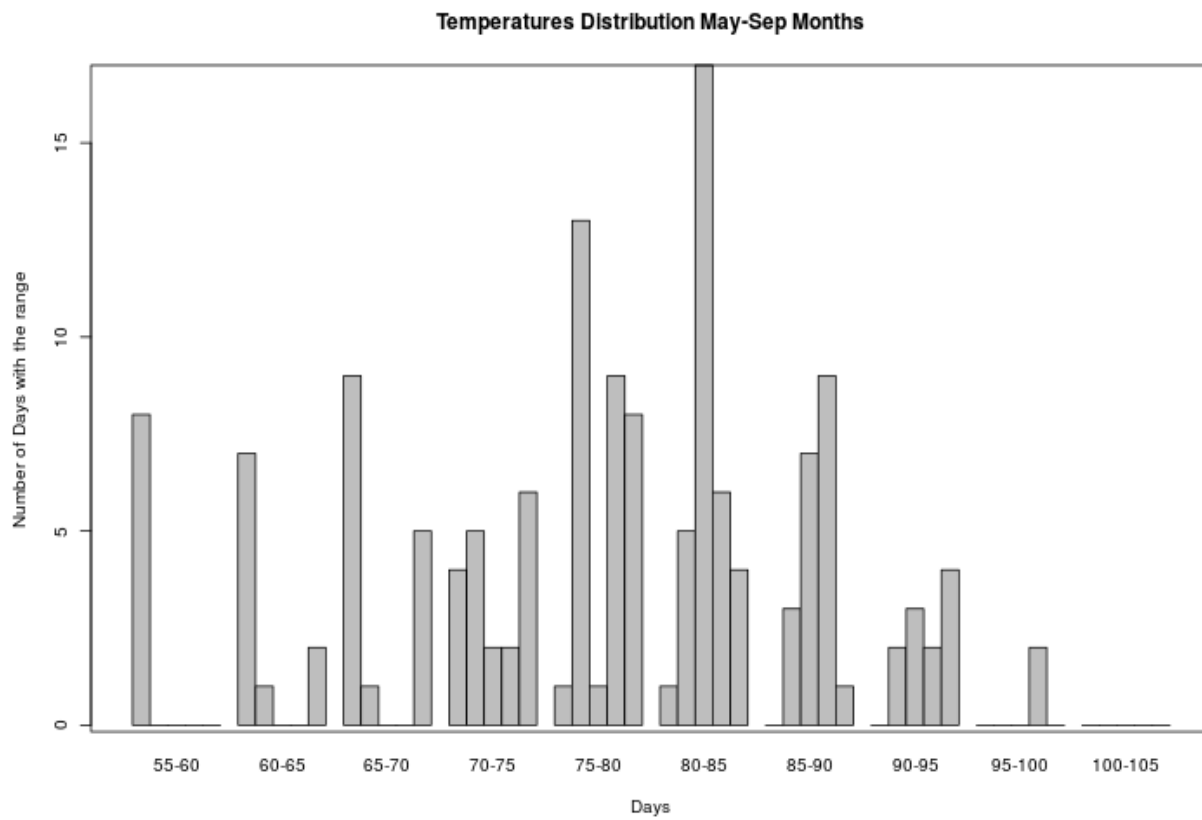


Figure 0 : Example Bar Plot with combined data

stackbar plot

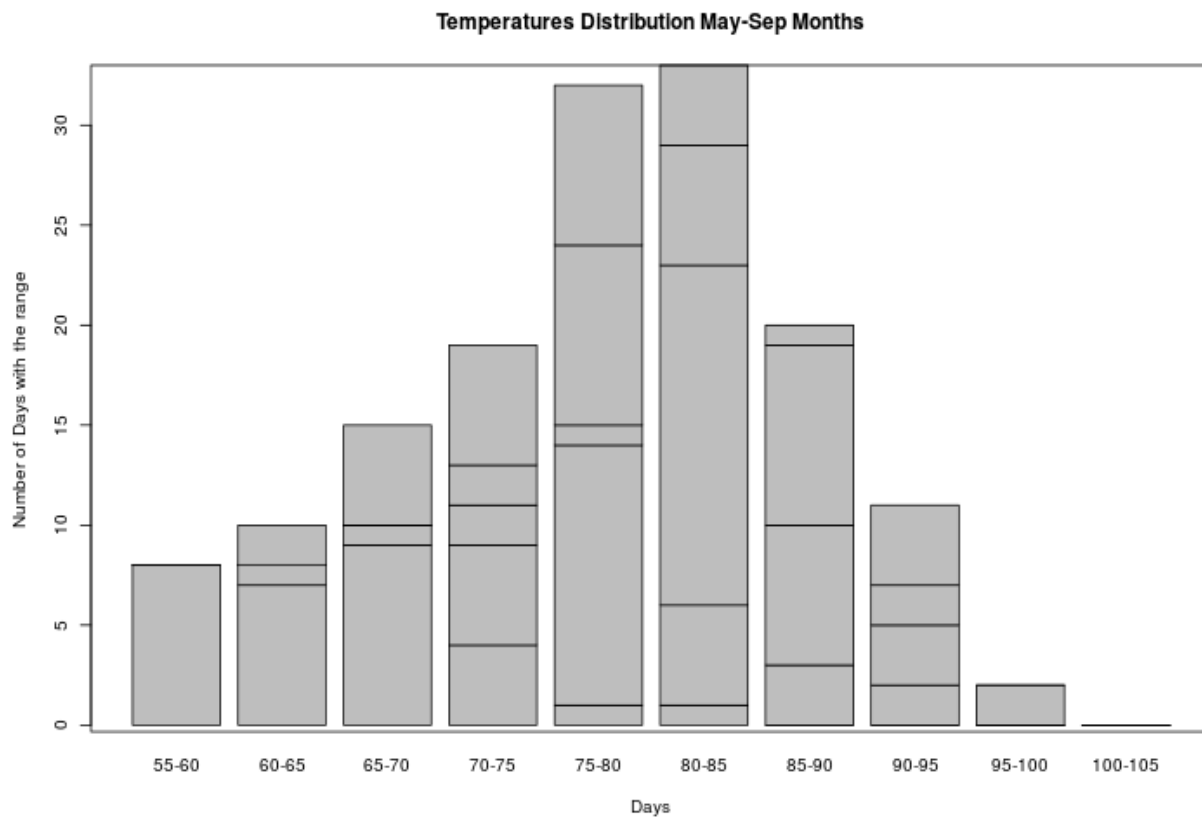


Figure 0 : Example Bar Plot with combined data

Multi line plots

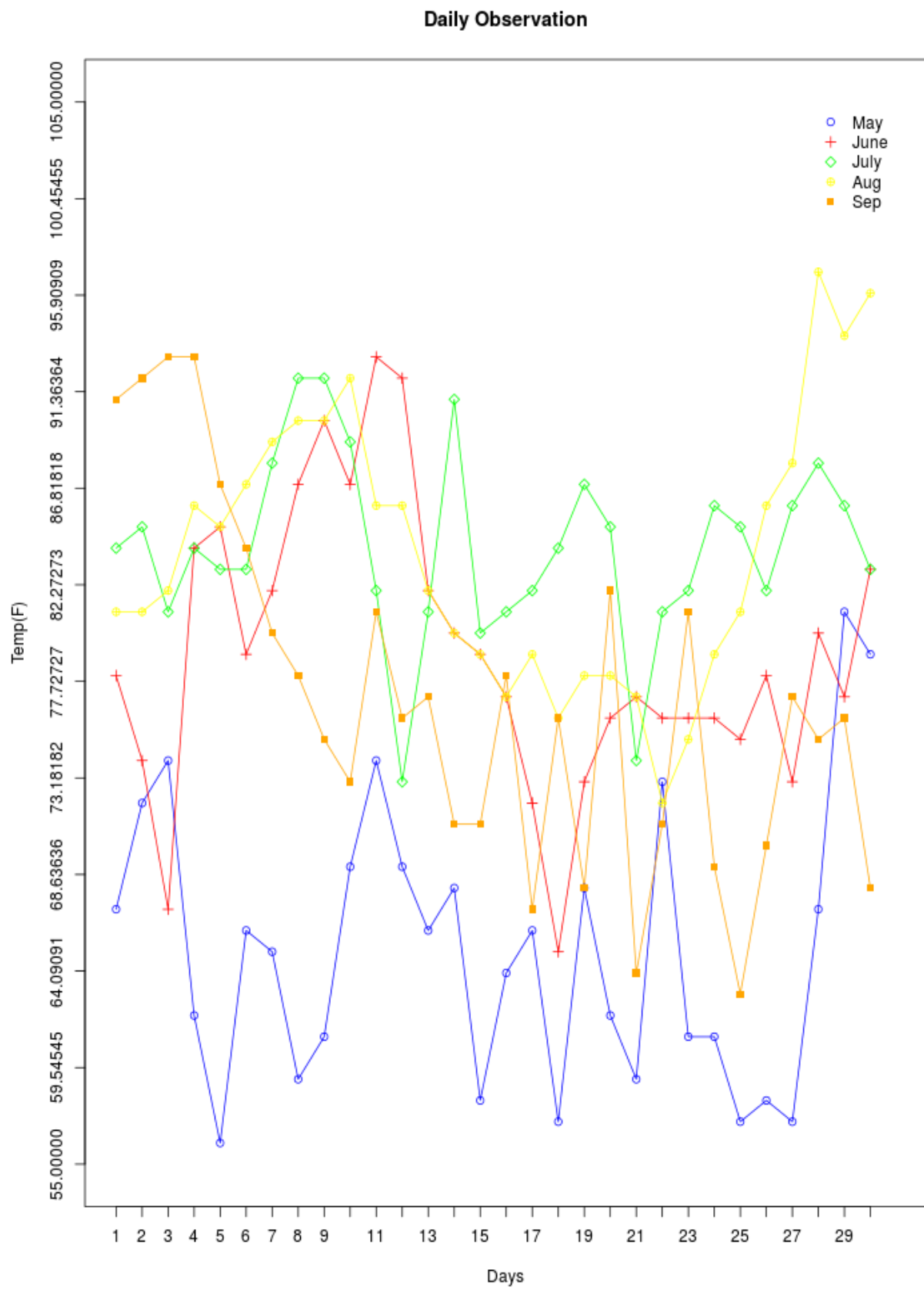


Figure 0 : Observation

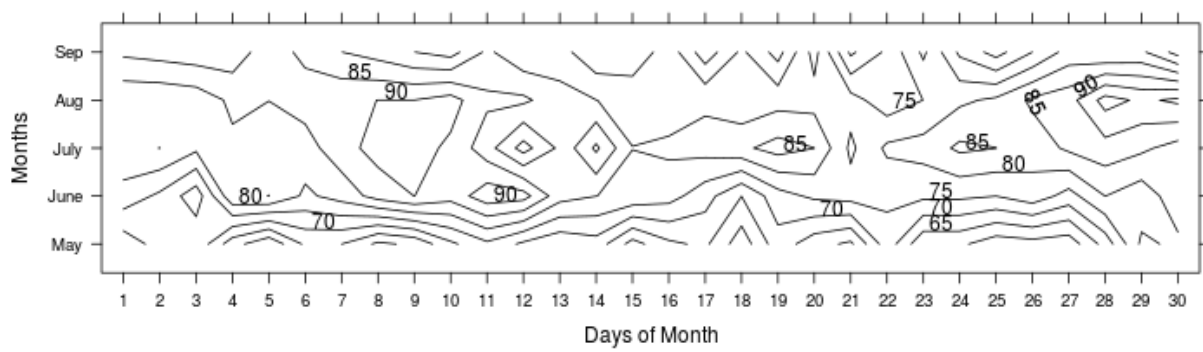


Figure 0 : Daily Temperature Values Plotted as Contours

9 Simple Table Outputs

	May	June	July	Aug	Sep
1	67	78	84	81	91
2	72	74	85	81	92
3	74	67	81	82	93
4	62	84	84	86	93
5	56	85	83	85	87
6	66	79	83	87	84
7	65	82	88	89	80
8	59	87	92	90	78
9	61	90	92	90	75
10	69	87	89	92	73
11	74	93	82	86	81
12	69	92	73	86	76
13	66	82	81	82	77
14	68	80	91	80	71
15	58	79	80	79	71
16	64	77	81	77	78
17	66	72	82	79	67
18	57	65	84	76	76
19	68	73	87	78	68
20	62	76	85	78	82
21	59	77	74	77	64
22	73	76	81	72	71
23	61	76	82	75	81
24	61	76	86	79	69
25	57	75	85	81	63
26	58	78	82	86	70
27	57	73	86	88	77
28	67	80	88	97	75
29	81	77	86	94	76
30	79	83	83	96	68

May be grid table from grid Extra package reference

	May	June	July	Aug	Sep
1	67	78	84	81	91
2	72	74	85	81	92
3	74	67	81	82	93
4	62	84	84	86	93
5	56	85	83	85	87
6	66	79	83	87	84
7	65	82	88	89	80
8	59	87	92	90	78
9	61	90	92	90	75
10	69	87	89	92	73
11	74	93	82	86	81
12	69	92	73	86	76
13	66	82	81	82	77
14	68	80	91	80	71
15	58	79	80	79	71
16	64	77	81	77	78
17	66	72	82	79	67
18	57	65	84	76	76
19	68	73	87	78	68
20	62	76	85	78	82
21	59	77	74	77	64
22	73	76	81	72	71
23	61	76	82	75	81
24	61	76	86	79	69
25	57	75	85	81	63
26	58	78	82	86	70
27	57	73	86	88	77
28	67	80	88	97	75
29	81	77	86	94	76
30	79	83	83	96	68

summary of statistics in table format

```

paired_tTest<-function(){

library('uRProgramming')
#heightData<-data.frame(CreateDataFile())
load("/home/zekai/R_workspace/r_repo/lib/uRProgramming/data/heightData.RData")
df<-heightData
t.test(df$Kadin,df$Erkek,paired=TRUE, conf.level=0.95)

#Do it for the chickwts (not equal number of sample)
#data("chickwts")

colNames<-levels(chickwts$feed)
xMatr<-array(NA,c(14,6))

maxLen<-dim(xMatr)[2]-1
tvalues<-array("",c(maxLen,dim(xMatr)[2]))

xStats<-array("",c(length(colNames),7))
tValues<-array("",c(6,3))

pValues<-array("",c(5,6))

colnames(xMatr)<-colNames
for (colIndx in 1:length(colNames)){
  x<-
chickwts[which(as.character(chickwts$feed)==as.character(colNames[colIndx])),]
  x1<-x$weight
  #print("x",x1)
  for (rowIndx in 1:length(x1)){

    xMatr[rowIndx,colIndx]<-x1[rowIndx]
  }
}

for (rrowIndx in 1:maxLen){
  iStart<-rrowIndx+1

  for (ccolIndx in iStart:dim(xMatr)[2]){

    # tvalues[rowIndx,ccolIndx]
    tValue<-t.test(xMatr[,rrowIndx],xMatr[,ccolIndx],paired=TRUE, conf.level=0.95)
    #pvals<-tvalues[rowIndx,ccolIndx]$p.value
    pvals<-tValue$p.value
    pValues[rrowIndx,ccolIndx]<-format(pvals,digits = 1 ,nsmall = 5, justify =
"left")

    rowValues<-paste("rrowIndx=",rrowIndx,sep=" ")
    colValues<-paste("ccolIndx=",ccolIndx,sep=" ")
    print (rowValues)
    print (pValues[rrowIndx,ccolIndx])
  }
  tValues[iStart,1]<-format(tValue$parameter,digits = 2 ,nsmall = 0, justify =
"left")
  tValues[iStart,2]<-format(tValue$statistic,digits = 2 ,nsmall = 4, justify =

```

```

"left")
  tValues[iStart,3]<-format(tValue$p.value,digits = 1 ,nsmall = 4, justify =
"left")
}

for (columnIndx in 1:dim(xMatr)[2]){
#   print(c("columnIndx=",columnIndx))
  rr2owIndx=columnIndx
#   print(xStats)
  xStats[rr2owIndx,1]<-length(xMatr[,columnIndx])
  xStats[rr2owIndx,2]<-summary(xMatr[,columnIndx])[4]
  xStats[rr2owIndx,3]<-summary(xMatr[,columnIndx])[7]
  xStats[rr2owIndx,4]<-format(sd(xMatr[,columnIndx],na.rm=TRUE),digits = 2 ,ns-
mall = 2, justify = "left")
  if(rr2owIndx >=2) {
    xStats[rr2owIndx,5]<-tValues[rr2owIndx,1]
    xStats[rr2owIndx,6]<-tValues[rr2owIndx,2]
    xStats[rr2owIndx,7]<-tValues[rr2owIndx,3]
  }
#   print(xStats)
  rownames(xStats)<-colNames
  colnames(xStats)<-c("n", "Mean", "num Of NA", "StdDev", "df", "t", "p-values")
} #columnIndx

return(list(xMatr=xMatr,pValues=pValues,xStats=xStats))
}
[1] "rrowIndx= 1"
[1] "9e-05"
[1] "rrowIndx= 1"
[1] "8e-04"
[1] "rrowIndx= 1"
[1] "0.09916"
[1] "rrowIndx= 1"
[1] "0.02294"
[1] "rrowIndx= 1"
[1] "0.79590"
[1] "rrowIndx= 2"
[1] "0.02223"
[1] "rrowIndx= 2"
[1] "2e-04"
[1] "rrowIndx= 2"
[1] "4e-06"
[1] "rrowIndx= 2"
[1] "9e-06"
[1] "rrowIndx= 3"
[1] "0.01167"
[1] "rrowIndx= 3"
[1] "0.16020"
[1] "rrowIndx= 3"
[1] "6e-05"
[1] "rrowIndx= 4"

```

```

[1] "0.46506"
[1] "rrowIdx= 4"
[1] "0.00584"
[1] "rrowIdx= 5"
[1] "0.00252"

```

source paired_test and modify heightData input it will be read

	casein	horsebean	linseed	meatmeal	soybean	sunflower
1	368	179	309	325	243	423
2	390	160	229	257	230	340
3	379	136	181	303	248	392
4	260	227	141	315	327	339
5	404	217	260	380	329	341
6	318	168	203	153	250	226
7	352	108	148	263	193	320
8	359	124	169	242	271	295
9	216	143	213	206	316	334
10	222	140	257	344	267	322
11	283	NA	244	258	199	297
12	332	NA	271	NA	171	318
13	NA	NA	NA	NA	158	NA
14	NA	NA	NA	NA	248	NA

t -test p-values compare to each other

Feed Category	Statistics				pair t test results		
	n	Mean	num Of NA	StdDev	df	t	p-values
casein	14	323.6	2	64.43			
horse-bean	14	160.2	4	38.63	11	-0.2650	0.7959
linseed	14	218.8	2	52.24	9	-8.9998	9e-06
meatmeal	14	276.9	3	64.90	11	-6.3416	6e-05
soybean	14	246.4	NA	54.13	10	-3.4887	0.0058
sunflower	14	328.9	2	48.84	11	-3.8892	0.0025

	casein	horsebean	linseed	meatmeal	soybean	sunflower
casein		9e-05	8e-04	0.09916	0.02294	0.79590
horsebean			0.02223	2e-04	4e-06	9e-06
linseed				0.01167	0.16020	6e-05
meatmeal					0.46506	0.00584
soybean						0.00252

Car Model	Features										
	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3

Car Model	Features										
	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
450SE											
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SL C	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4

Car Model	Features										
	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Z28											
Pon-tiac Fire-bird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsc he 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Eu-ropa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pan-tera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Fer-rari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maser ati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

started here one

1.00
11.01
21.02
31.03
41.04
51.05
61.07
71.08
81.09

Please start here

10 Reference

- git (first in page5)
- Rstudio(first in page 5)
- MySQL reference
- RMySQL