R PROGRAMMING LAB MANNUAL

1. **Write R code to perform Different arithmetic operations on two vectors**

Source Code:

```
x<-c(7,5,3,6)
y<-c(1,4,3,5)
x+y
x-y
x*y
x/y
x%%y
```

Output:

```
>x+y
[1]  8  9  6 11
> x-y
[1] 6 1 0 1
> x*y
[1]  7 20  9 30
> x/y
[1] 7.00 1.25 1.00 1.20
> x%%y
[1] 0 1 0 1
```

2. Write R code to find a)Mean b)Median of a vector

Source Code:

```
x<-c(1,3,5)
mean(X)
median(x)
```

Output:

```
> mean(X)
[1] 4
> median(x)
[1] 3
```

3. Write R code, Creating a Function that gives Oddcount of a vector.

Source Code:

```
oddcount<-function(x){
 k<-0
for(n in x)
if(n %% 2 == 1)   k<-k+1
   return(k)
}
x<-c(1,3,24,5,4,2,9,6,5)
oddcount(x)
```

Output:

```
> x<-c(1,3,24,5,4,2,9,6,5)
>oddcount(x)
[1] 5
```

4. Write a R code to print Fibonacci Series.

Source Code:

```
# take input from the user
n terms = as. integer (read line(prompt="How many terms? "))
# first two terms
n1 = 0
n2 = 1
count = 2
# check if the number of terms is valid
if(n terms<= 0) {
print("Please enter a positive integer")
} else {
if(n terms == 1) {
print("Fibonacci sequence:")
   print(n1)
  } else {
print("Fibonacci sequence:")
   print(n1)
   print(n2)
while(count <n terms) {
     nth = n1 + n2
     print(nth)
```

```
    # update values
    n1 = n2
    n2 = nth
    count = count + 1
   }
  }
}
```

Output:

How many terms? 9

[1] "Fibonacci sequence:"

[1] 0

[1] 1

[1] 1

[1] 2

[1] 3

[1] 5

[1] 8

[1] 13

[1] 21

5. Write R code to create a function that print factors of a given number.

Source Code:

```
print_factors<- function(x) {
print(paste("The factors of",x,"are:"))
for(i in 1:x) {
if((x %% i) == 0) {
      print(i)
    }
  }
}
```

Output:

```
>print_factors(125)

[1] "The factors of 125 are:"

[1] 1

[1] 5

[1] 25

[1] 125
```

6. Write R code to check whether the given number is Armstrong or not.

An Armstrong number, also known as narcissistic number, is a number that is equal to the sum of the cubes of its own digits.

For example, 370 is an Armstrong number since 370 = 3*3*3 + 7*7*7 + 0*0*0.

Source Code:

```
# take input from the user
num = as. integer (read line(prompt="Enter a number: "))
# initialize sum
sum = 0
# find the sum of the cube of each digit
temp = num
while(temp > 0) {
  digit = temp %% 10
  sum = sum + (digit ^ 3)
  temp = floor(temp / 10)
}
# display the result
if(num == sum) {
print(paste(num, "is an Armstrong number"))
} else {
print(paste(num, "is not an Armstrong number"))
}
```

Output:

```
>num = as. Integer (read line(prompt="Enter a number: "))
Enter a number: 370
[1] "370 is an Armstrong number"
>num = as. integer(read line(prompt="Enter a number: "))
Enter a number: 254
[1] "254 is not an Armstrong number"
```

Simple Graphs:

7.Consider x and y vectors having set of values as (1,2,3,4,5,5,5,6) and (9,4,6,3,8,4,2,1) respectively. Write a R Code to Plot the following types of for above vectors x,y:

a) Line Graph
b) Points Graph
c) Histograms
d) Stair steps Graph

Source code:

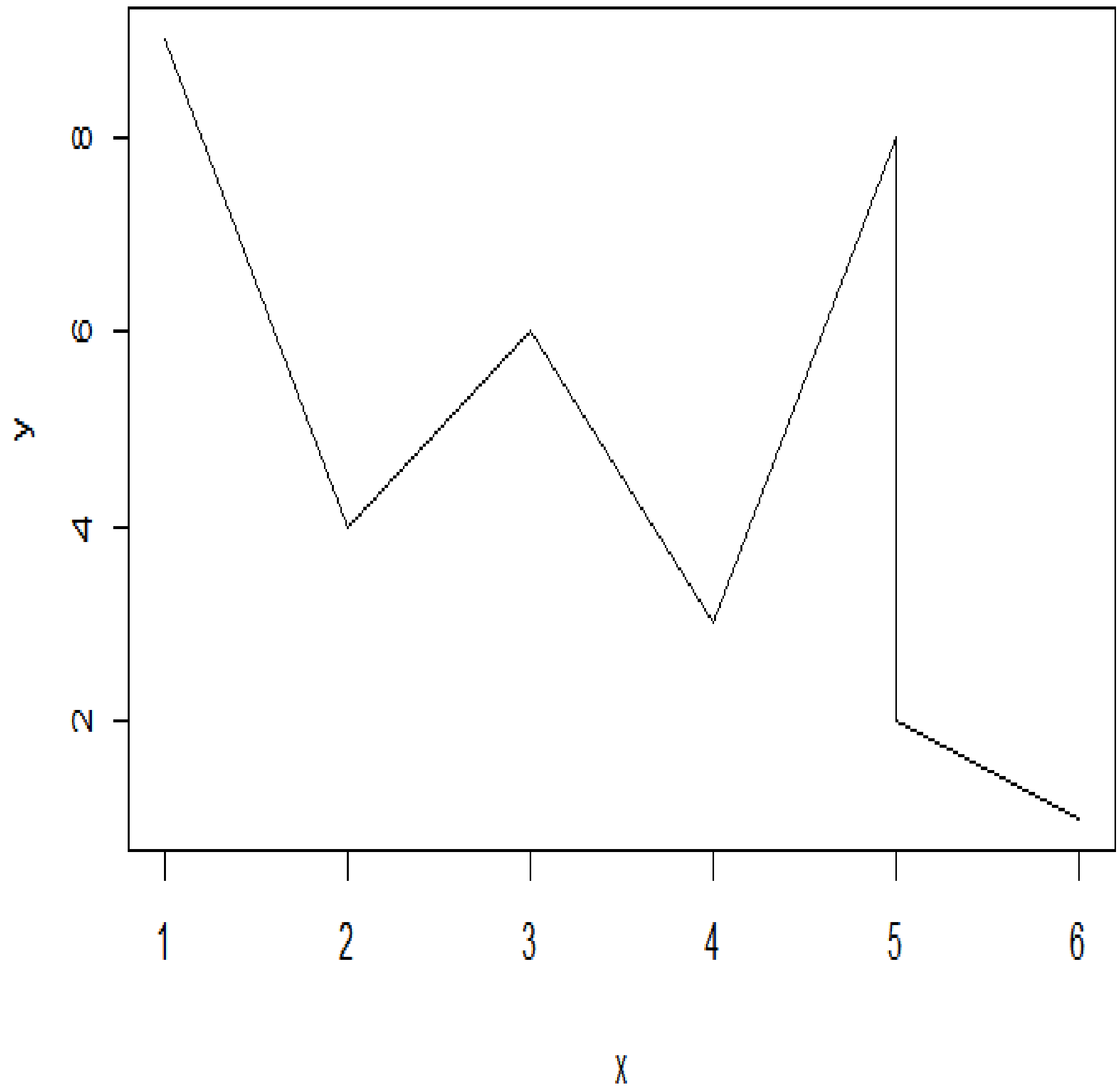
a. Line Graph

```
x<-c(1,2,3,4,5,5,5,6)

y<-c(9,4,6,3,8,4,2,1)

plot(x,y,"l")
```

Output:



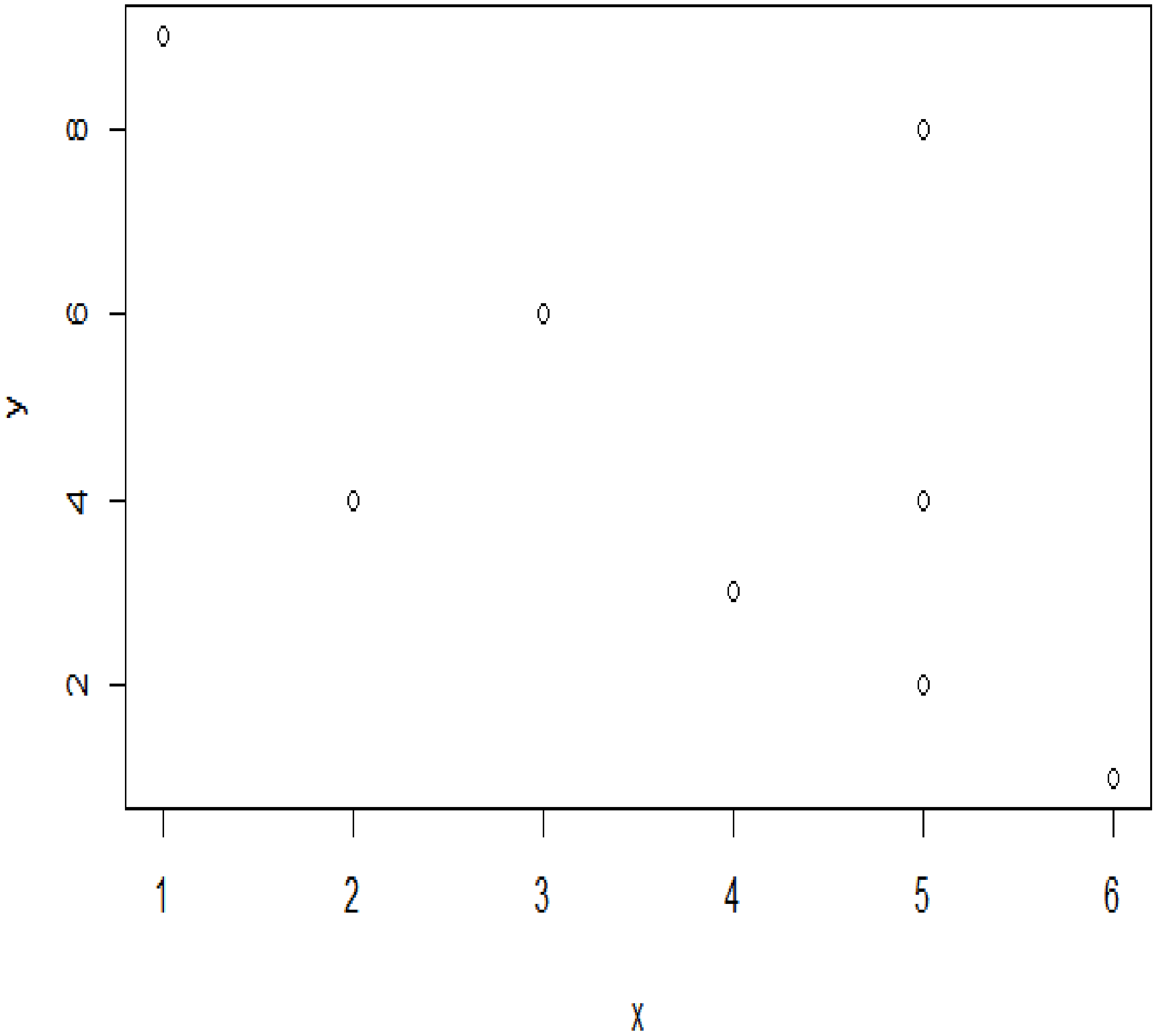
b. Points Graphs

x<-c(1,2,3,4,5,5,5,6)

y<-c(9,4,6,3,8,4,2,1)

plot(x,y,"p")

Output:

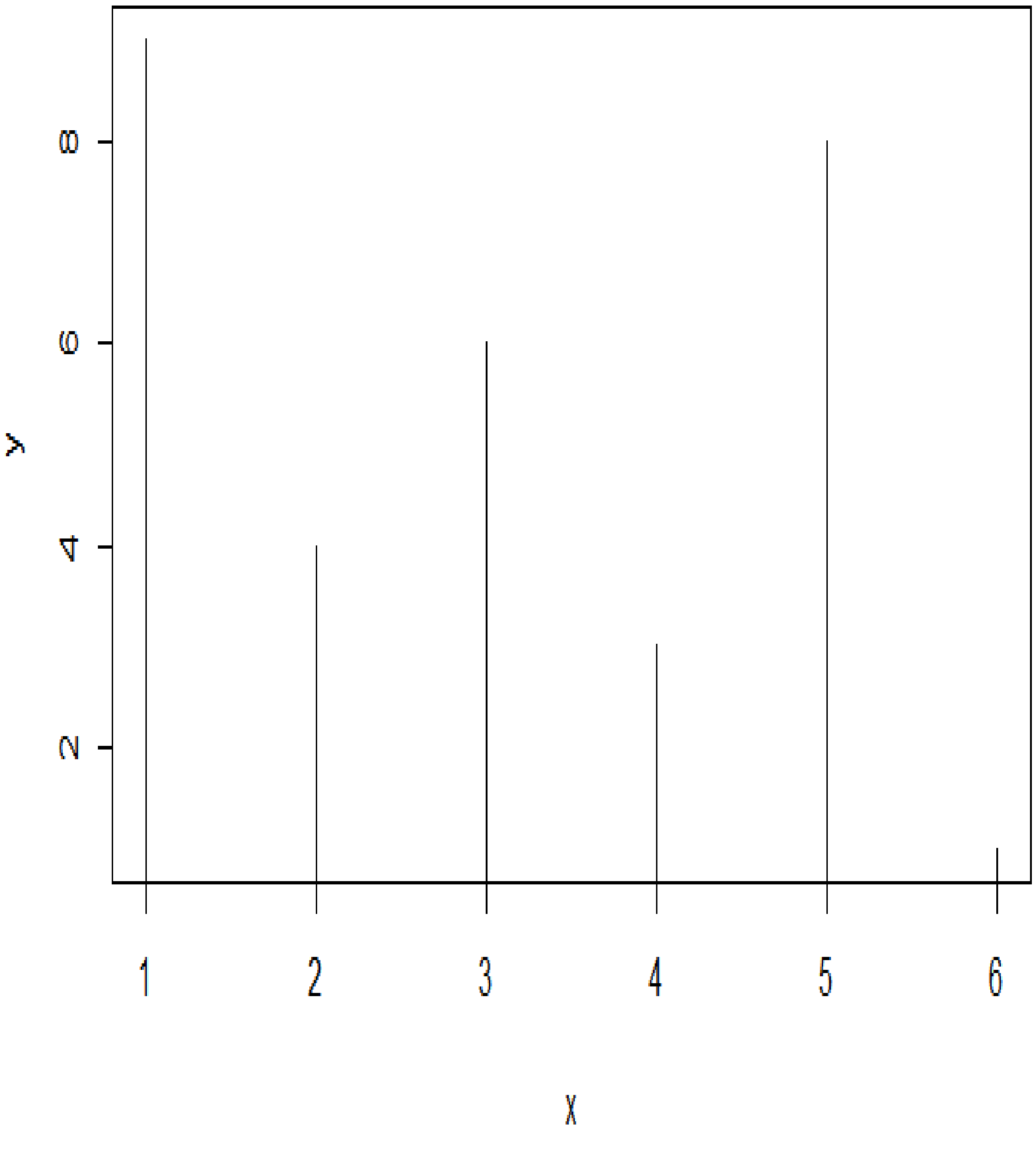


c. Histographs

x<-c(1,2,3,4,5,5,5,6)

```
y<-c(9,4,6,3,8,4,2,1)
plot(x,y,"h")
```

Output:



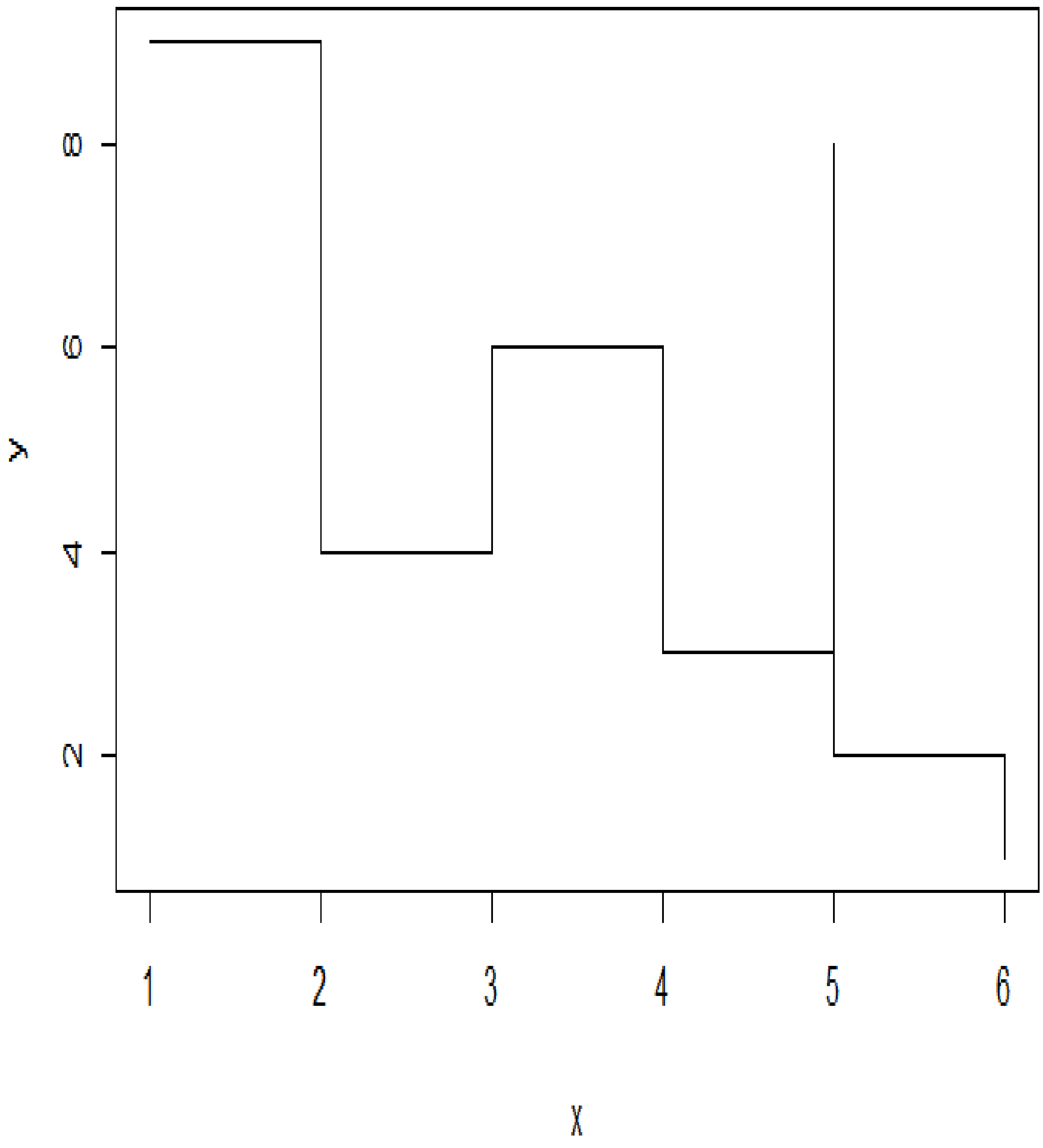
d. Stairsteps Graphs

```
x<-c(1,2,3,4,5,5,5,6)

y<-c(9,4,6,3,8,4,2,1)

plot(x,y,"s")
```

Output:



Labelled Graphs with Colours:

8.Write a R Code to Plot the following types of for given strengths and weights

a) Line Graph
b) Points Graph
c) Histograms
d) Stair steps Graph

Strengths	1	2	3	4	5	5	5	6
Weights	9	4	6	3	8	4	2	1

Source code:

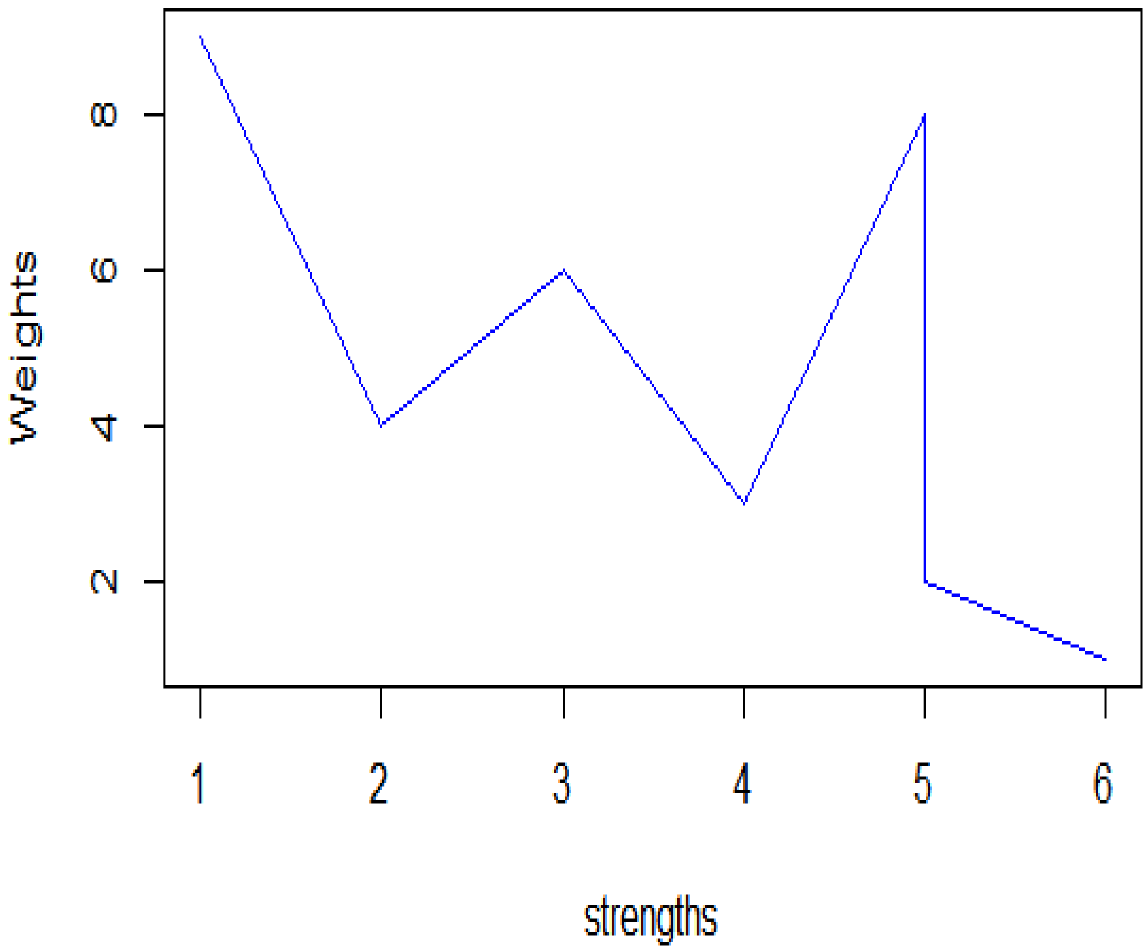
a. Line Graph

```
x<-c(1,2,3,4,5,5,5,6)

y<-c(9,4,6,3,8,4,2,1)

plot(x,y,"l",col="blue",xlab="strengths",ylab = "Weights")
```

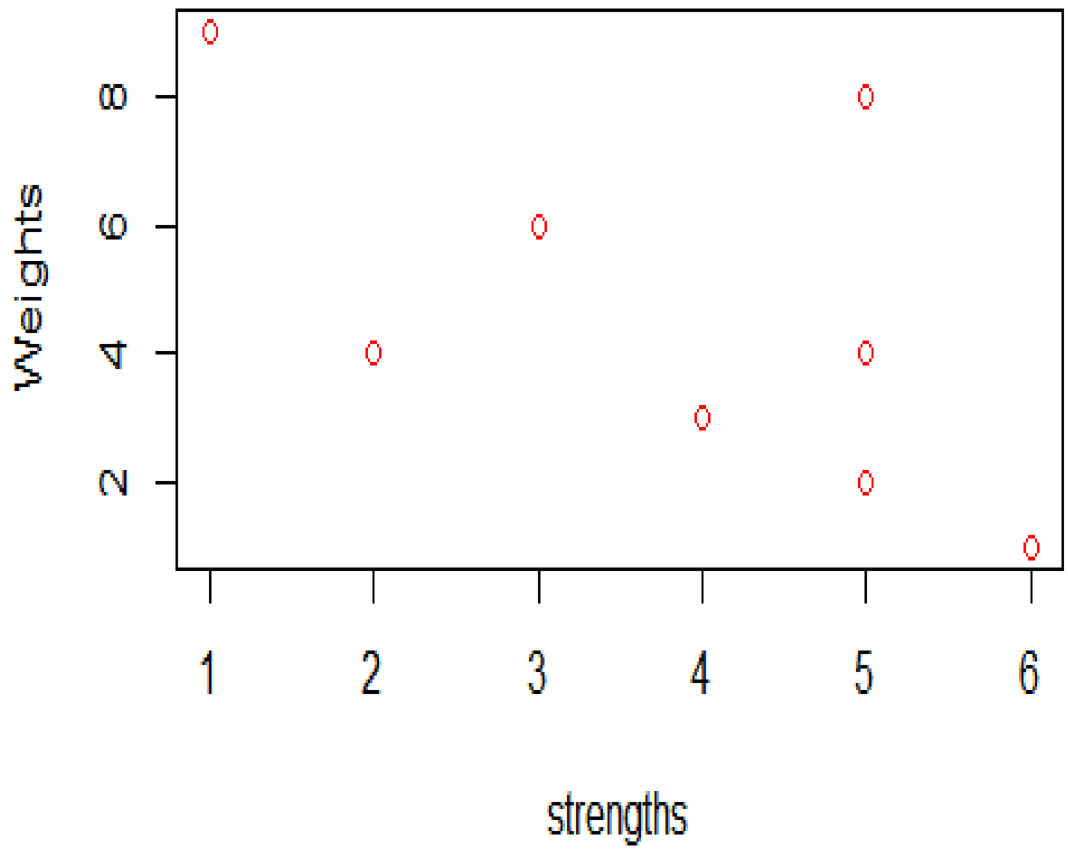
Output:



b. Points Graph

```
x<-c(1,2,3,4,5,5,5,6)
y<-c(9,4,6,3,8,4,2,1)
plot(x,y,"p",col="red",xlab="strengths",ylab = "Weights")
```

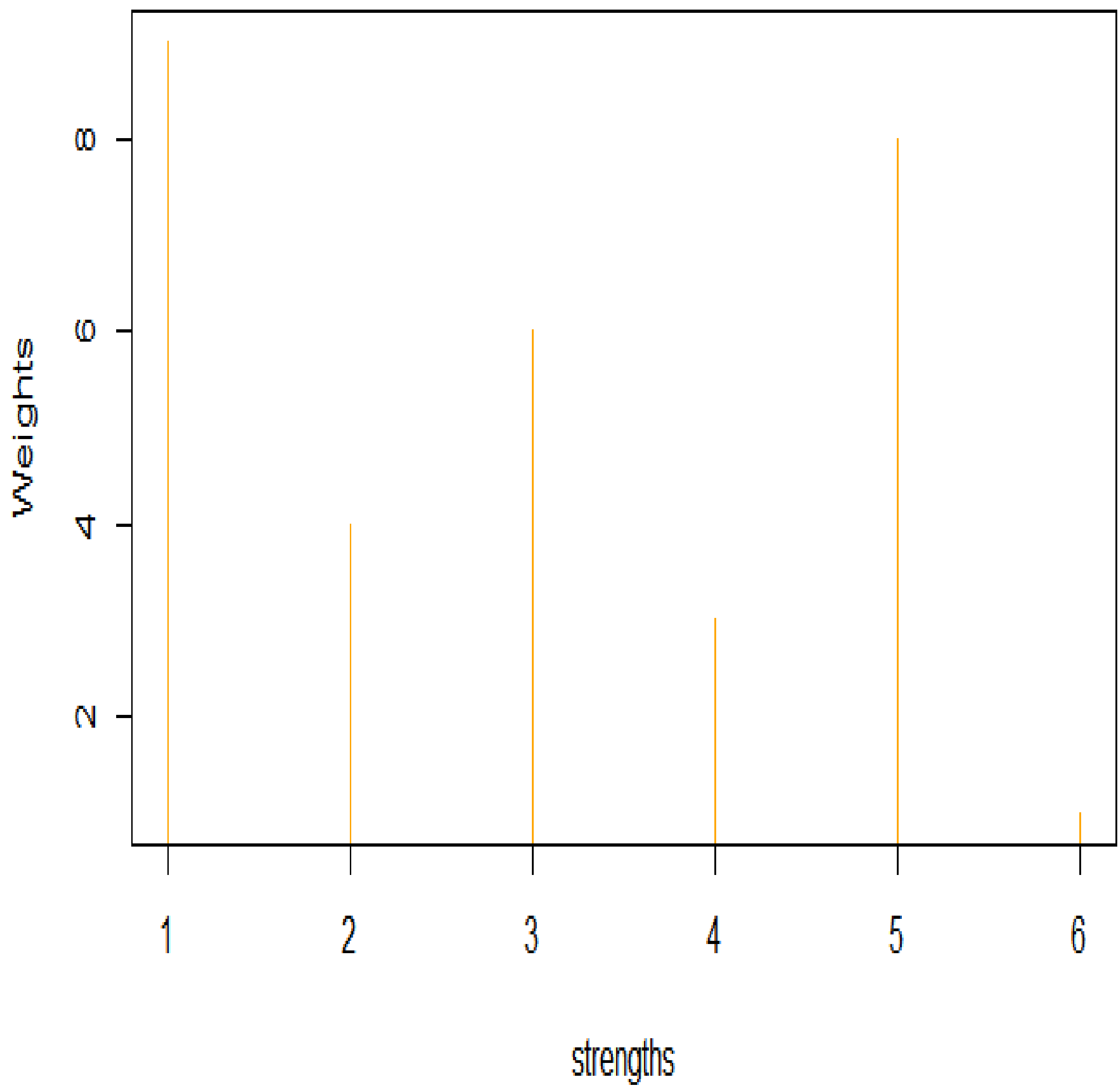
Output:



c. Histographs

```
x<-c(1,2,3,4,5,5,5,6)
y<-c(9,4,6,3,8,4,2,1)
plot(x,y,"h",col="orange",xlab="strengths",ylab = "Weights")
```

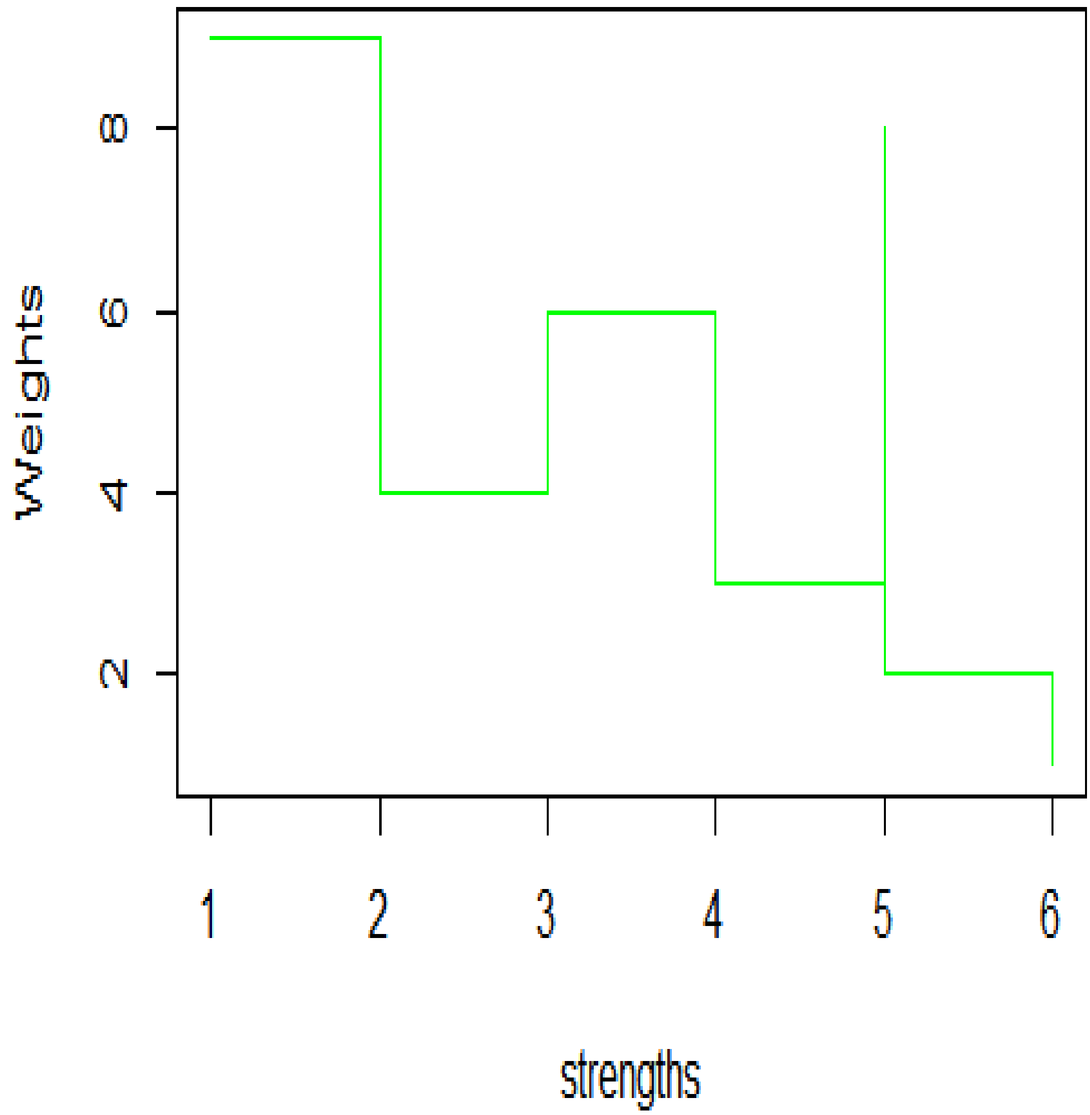
Output:



d. Stairsteps Graph

```
x<-c(1,2,3,4,5,5,5,6)
y<-c(9,4,6,3,8,4,2,1)
plot(x,y,"s",col="green",xlab="strengths",ylab = "Weights")
```

Output:


Weights
2
4
6
8
1
2
3
4
5
6
strengths

Minima and Maxima- Calculus, Functions Fir Statistical Distribution, Sorting, Linear Algebra Operation on Vectors and Matrices, Extended Example: Vector cross Product- Extended Example: Finding Stationary Distribution of Markov Chains, Set Operation, Input /output, Accessing the Keyboard and Monitor, Reading and writer Files,

Math Functions

R includes an extensive set of built-in math functions. Here is a partial list:

- exp(): Exponential function, base e
- log(): Natural logarithm
- log10(): Logarithm base 10
- sqrt(): Square root
- abs(): Absolute value
- sin(), cos(), and so on: Trig functions
- min() and max(): Minimum value and maximum value within a vector
- which.min() and which.max(): Index of the minimal element and maximal element of a vector
- pmin() and pmax(): Element-wise minima and maxima of several vectors
- sum() and prod(): Sum and product of the elements of a vector cumsum() and cumprod(): Cumulative sum and product of the elements of a vector
- round(), floor(), and ceiling(): Round to the closest integer, to the closest integer below, and to the closest integer above
- factorial(): Factorial function

Extended Example Calculating Probability

Calculating a probability using the prod() function. Suppose we have n independent events, and the i th event has the probability pi of occurring. What is the probability of exactly one of these events occurring?

Suppose first that n = 3 and our events are named A, B, and C. Then we break down the computation as follows:

P(exactly one event occurs) =
P(A and not B and not C) +
P(not A and B and not C) +
P(not A and not B and C)

P(A and not B and not C) would be pA(1 − pB)(1 − pC), and so on. For general n, that is calculated as follows:

(The ith term inside the sum is the probability that event i occurs and all the others do not occur.)

Here's code to compute this, with our probabilities pi contained in the vector p:

```
exactlyone <- function(p) {
notp <- 1 - p
tot <- 0.0
for (i in 1:length(p))
tot <- tot + p[i] * prod(notp[-i])
return(tot)
}
```

Cumulative Sums and Products

The functions cumsum() and cumprod() return cumulative sums and products.

```
> x <- c(12,5,13)
> cumsum(x)
[1] 12 17 30
> cumprod(x)
[1] 12 60 780
```

In x, the sum of the first element is 12, the sum of the first two elements is 17, and the sum of the first three elements is 30.

The function cumprod() works the same way as cumsum(), but with the product instead of the sum.

Minima and Maxima

There is quite a difference between min() and pmin(). The former simply combines all its arguments into one long vector and returns the minimum value in that vector. In contrast, if pmin() is applied to two or more vectors, it returns a vector of the pair-wise minima, hence the name pmin.

Here's an example:

```
z<-matrix(c(1,5,6,2,3,2),nrow=3,ncol=2)
> z
[,1] [,2]
[1,] 1 2
[2,] 5 3
 [3,] 6 2
> min(z[,1],z[,2])
[1] 1
> pmin(z[,1],z[,2])
[1] 1 3 2
```

In the first case, min() computed the smallest value in (1,5,6,2,3,2). But the call to pmin() computed the smaller of 1 and 2, yielding 1; then the smaller of 5 and 3, which is 3; then finally the minimum of 6 and 2, giving 2. Thus, the call returned the vector (1,3,2).

You can use more than two arguments in pmin(), like this:

```
> pmin(z[1,],z[2,],z[3,])
[1] 1 2
```

The 1 in the output is the minimum of 1, 5, and 6, with a similar computation leading to the 2.

The max() and pmax() functions act analogously to min() and pmin(). Function minimization/maximization can be done via nlm() and optim(). For example, let's find the smallest value of f(x) = x2 − sin x .

```
>nlm(function(x) return(x^2-sin(x)),8)
$minimum
[1] -0.2324656
$estimate
[1] 0.4501831
$gradient
[1] 4.024558e-09
$code
[1] 1
$iterations
```

[1] 5

Here, the minimum value was found to be approximately −0.   , occurring at x = 0.45. A Newton-Raphson method (a technique from numerical analysis for approximating roots) is used, running five iterations in this case.

Calculus

R also has some calculus capabilities, including symbolic differentiation and numerical integration, as you can see in the following example

R also has some calculus capabilities, including symbolic differentiation and numerical integration, as you can see in the following example.

```
> D(expression(exp(x^2)),"x") # derivative
exp(x^2) * (2 * x)
> integrate(function(x) x^2,0,1)
0.3333333 with absolute error < 3.7e-15
```

Functions Fir Statistical Distribution

R has functions available for most of the famous statistical distributions.

Prefix the name as follows:

• With d for the density or probability mass function (pmf)
• With p for the cumulative distribution function (cdf)
• With q for quantiles
• With r for random number generation

The rest of the name indicates the distribution. as shown in below Table lists some common statistical distribution functions.

Distribution Density/pmf cdf Quantiles Random Numbers
Normal dnorm() pnorm() qnorm() rnorm()
Chi square dchisq() pchisq() qchisq() rchisq()
Binomial dbinom() pbinom() qbinom() rbinom()

Table: Common R Statistical Distribution Functions

As an example, let’s simulate 1,000 chi-square variates with 2 degrees of freedom and find their mean.

```
> mean(rchisq(1000,df=2))
[1] 1.938179
```

The r in rchisq specifies that we wish to generate random numbers in this case, from the chi-square distribution. As seen in this example, the first argument in the r-series functions is the number of random variates to generate.

In our example, we use the df argument for the chi-square family, indicating the number of degrees of freedom.

Let’s also compute the 95th percentile of the chi-square distribution with two degrees of freedom:

```
> qchisq(0.95,2)
[1] 5.991465
```

Here, we used q to indicate quantile—in this case, the 0.95 quantile, or the 95th percentile. The first argument in the d, p, and q series is actually a vector so that we can evaluate the density/pmf, cdf, or quantile function at multiple points.

Let’s find both the 50th and 95th percentiles of the chi-square distribution with 2 degrees of freedom.

qchisq(c(0.5,0.95),df=2)
[1] 1.386294 5.991465

Sorting

numerical sorting of a vector can be done with the sort() function, as in this example:

```
> x <- c(13,5,12,5)
> sort(x)
[1] 5 5 12 13
> x
[1] 13 5 12 5
```

If you want the indices of the sorted values in the original vector, use the order() function. Here's an example:

```
> order(x)
[1] 2 4 3 1
```

This means that x[2] is the smallest value in x, x[4] is the second smallest, x[3] is the third smallest, and so on.

You can use order(), together with indexing, to sort data frames, like this:

```
> v1<-c("def","ab","zzzz")
> v2<-c(2,5,1)
> y<-data.frame(v1,v2)
> y
v1 v2
1 def 2
2 ab 5
3 zzzz 1
> r <- order(y$v2)
> r
[1] 3 1 2
> z <- y[r,]
> z
V1 V2
3 zzzz 1
1 def 2
2 ab 5
```

order() on the second column of y, yielding a vector r, telling us where numbers should go if we want to sort them. The 3 in this vector tells us that x[3,2] is the smallest number in x[,2]; the 1 tells us that x[1,2] is the second smallest; and the 2 tells us that x[2,2] is the third smallest. We then use indexing to produce the frame sorted by column 2, storing it in z.

You can use order() to sort according to character variables as well as numeric ones, as

follows:
> d
kids ages
1 Jack 12
2 Jill 10
3 Billy 13
> d[order(d$kids),]
kids ages
3 Billy 13
1 Jack 12
2 Jill 10
> d[order(d$ages),]
kids ages
2 Jill 10
1 Jack 12
3 Billy 13
A related function is rank(), which reports the rank of each element of a vector.
> x <- c(13,5,12,5)
> rank(x)
[1] 4.0 1.5 3.0 1.5
This says that 13 had rank 4 in x; that is, it is the fourth smallest. The value 5 appears twice in x, with those two being the first and second smallest, so the rank 1.5 is assigned to both.

Linear Algebra Operation on Vectors and Matrices

Multiplying a vector by a scalar works directly
> y
[1] 1 3 4 10
> 2*y
[1] 2 6 8 20
If you wish to compute the inner product (or dot product) of two vectors, use crossprod(), like this:
> crossprod(1:3,c(5,12,13))
[,1]
[1,] 68
The function computed 1 · 5 + 2 · 12 + 3 · 13 = 68.
Note that the name crossprod() is a misnomer, as the function does not compute the vector cross product.
For matrix multiplication in the mathematical sense, the operator to use is %*%, not *. For instance, here we compute the matrix product:
a<-matrix(c(1,2,3,4),nrow=2,ncol=2,byrow=TRUE)
> a
[,1] [,2]
[1,] 1 2
[2,] 3 4
b<-matrix(c(1,-1,0,1),nrow=2,ncol=2,byrow=TRUE)
> b
[,1] [,2]
[1,] 1 -1
[2,] 0 1
> a %*% b
[,1] [,2]

[1,] 1 1
[2,] 3 1
The function solve() will solve systems of linear equations and even find
matrix inverses. For example, let's solve this system:
x1 - x2 = 2
x1 + x2 = 4

```
> a <- matrix(c(1,1,-1,1),nrow=2,ncol=2)
> a
[,1] [,2]
[1,] 1 -1
[2,] 1 1
> b <- c(2,4)
> solve(a,b)
[1] 3 1
> solve(a)
[,1] [,2]
[1,] 0.5 0.5
[2,] -0.5 0.5
```

that we simply wish to compute the inverse of the matrix. Here are a few other linear algebra functions:

- t(): Matrix transpose
- qr(): QR decomposition
- chol(): Cholesky decomposition
- det(): Determinant
- eigen(): Eigenvalues/eigenvectors
- diag(): Extracts the diagonal of a square matrix (useful for obtaining
- variances from a covariance matrix and for constructing a diagonal
- matrix).
- sweep(): Numerical analysis sweep operations

nature of diag(): If its argument is a matrix, it returns a vector, and vice versa. Also, if the argument is a scalar, the function returns the identity matrix of the specified size.

```
> m<-matrix(c(1,2,7,8),nrow=2,ncol=2,byrow=TRUE)
> m
[,1] [,2]
[1,] 1 2
[2,] 7 8
> dm <- diag(m)
> dm
[1] 1 8
> diag(dm)
[,1] [,2]
[1,] 1 0
[2,] 0 8
> diag(3)
[,1] [,2] [,3]
[1,] 1 0 0
[2,] 0 1 0
[3,] 0 0 1
```

The sweep() function is capable of fairly complex operations. As a simple example, let's take a 3-by-3 matrix and add 1 to row 1, 4 to row 2, and 7 to row 3.

```
> m
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 4 5 6
[3,] 7 8 9
> sweep(m,1,c(1,4,7),"+")
[,1] [,2] [,3]
[1,] 2 3 4
[2,] 8 9 10
[3,] 14 15 16
```

The first two arguments to sweep() are like those of apply(): the array and the margin, which is 1 for rows in this case. The fourth argument is a function to be applied, and the third is an argument to that function (to the "+" function).

Extended Example: Vector cross Product- Extended Example: Finding
Stationary Distribution of Markov Chains

A Markov chain is a random process in which we move among various states, in a memoryless fashion, whose definition need not concern us here. The state could be the number of jobs in a queue, the number of items stored in inventory, and so on. We will assume the number of states to be finite.

As a simple example, consider a game in which we toss a coin repeatedly and win a dollar whenever we accumulate three consecutive heads. Our state at any time i will the number of consecutive heads we have so far, so our state can be 0, 1, or 2. (When we get three heads in a row, our state reverts to 0.)

Markov modeling is usually the long-run state distribution, meaning the long-run proportions of the time we are in each state. In our coin-toss game, we can use the code we'll develop here to calculate that distribution, which turns out to have us at states 0, 1, and 2 in proportions 57.1%, 28.6%, and 14.3% of the time. Note that we win our dollar if we are in state 2 and toss a head, so $0.143 \times 0.5 = 0.071$ of our tosses will result in wins.

Since R vector and matrix indices start at 1 rather than 0, it will be convenient to relabel our states here as 1, 2, and 3 rather than 0, 1, and 2. For example, state 3 now means that we currently have two consecutive heads.

Let pij denote the transition probability of moving from state i to state j during a time step. In the game example, for instance, p23 = 0.5, reflecting the fact that with probability 1/2, we will toss a head and thus move from having one consecutive head to two. On the other hand, if we toss a tail while we are in state 2, we go to state 1, meaning 0 consecutive heads; thus p21 = 0.5.

Set Operation

R includes some handy set operations, including these:

- union(x,y): Union of the sets x and y
- intersect(x,y): Intersection of the sets x and y
- setdiff(x,y): Set difference between x and y, consisting of all elements of x that are not in y
- setequal(x,y): Test for equality between x and y
- c %in% y: Membership, testing whether c is an element of the set y
- choose(n,k): Number of possible subsets of size k chosen from a set of size n

Here are some simple examples of using these functions:

```
> x <- c(1,2,5)
> y <- c(5,1,8,9)
> union(x,y)
[1] 1 2 5 8 9
> intersect(x,y)
[1] 1 5
> setdiff(x,y)
[1] 2
> setdiff(y,x)
[1] 8 9
> setequal(x,y)
[1] FALSE
> setequal(x,c(1,2,5))
[1] TRUE
> 2 %in% x
[1] TRUE
> 2 %in% y
[1] FALSE
> choose(5,2)
[1] 10
```

Input /output

I/O capabilities, the basics of access to the keyboard and monitor, and then go into considerable detail on reading and writing files, including the navigation of file directories.

Accessing the Keyboard and Monitor

R provides several functions for accessing the keyboard and monitor. Here, we'll look at the scan(), readline(), print(), and cat() functions.

Using the scan() Function

You can use scan() to read in a vector, whether numeric or character, from a file or the keyboard. With a little extra work, you can even read in data to form a list.

Suppose we have files named z1.txt, z2.txt, z3.txt, and z4.txt. The z1.txt file contains the following:

```
--------------------------------------------------------------------------------
123
4 5
6
--------------------------------------------------------------------------------
```

The z2.txt file contents are as follows:

```
--------------------------------------------------------------------------------
123
4.2 5
6
```

The z3.txt file contains this:

---

abc
de f
g

And finally, the z4.txt file has these contents:

---

abc
123 6
y

---

we can do with these files using the scan() function.
> scan("z1.txt")
Read 4 items
[1] 123 4 5 6
> scan("z2.txt")
Read 4 items
[1] 123.0 4.2 5.0 6.0
> scan("z3.txt")
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings, :
scan() expected 'a real', got 'abc'
> scan("z3.txt",what="")
Read 4 items
[1] "abc" "de" "f" "g"
> scan("z4.txt",what="")
Read 4 items
[1] "abc" "123" "6" "y"
In the **first call,** we got a vector of four integers (though the mode is numeric).
The **second time**, since one number was non-integral, the others were shown as floatingpoint numbers, too.
In the **third case,** we got an error. The scan() function has an optional argument named what, which specifies mode, defaulting to double mode. So, the nonnumeric contents of the file z3 produced an error. But we then tried again, with what="". This assigns a character string to what, indicating that we want character mode.

The **last call** worked the same way. The first item was a character string, so it treated all the items that followed as strings too.
Here's an example:
> v <- scan("z1.txt")
By default, scan() assumes that the items of the vector are separated by whitespace, which includes blanks, carriage return/line feeds, and horizontal tabs. You can use the optional sep argument for other situations. As example, we can set sep to the newline character to read in each line as a string, as follows:

```
> x1 <- scan("z3.txt",what="")
Read 4 items
> x2 <- scan("z3.txt",what="",sep="\n")
Read 3 items
> x1
[1] "abc" "de" "f" "g"
> x2
[1] "abc" "de f" "g"
> x1[2]
[1] "de"
> x2[2]
[1] "de f"
```

In the first case, the strings "de" and "f" were assigned to separate elements of x1. But in the second case, we specified that elements of x2 were to be delineated by end-of-line characters, not spaces. Since "de" and "f" are on the same line, they are assigned together to x[2].

Using the readline() Function

If you want to read in a single line from the keyboard, readline() is very handy.

```
> w <- readline()
abc de f
> w
[1] "abc de f"
```

Typically, readline() is called with its optional prompt, as follows:

```
> inits <- readline("type your initials: ")
type your initials: NM
> inits
[1] "NM"
```

Printing to the Screen

At the top level of interactive mode, you can print the value of a variable or expression by simply typing the variable name or expression. This won't work if you need to print from within the body of a function. In that case, you can use the print() function, like this:

```
> x <- 1:3
> print(x^2)
 [1] 1 4 9
```

Recall that print() is a generic function, so the actual function called will depend on the class of the object that is printed. If, for example, the argument is of class "table", then the print.table() function will be called.

To use cat() instead of print(), as the latter can print

only one expression and its output is numbered, which may be a nuisance. Compare the results of the functions:

```
> print("abc")
[1] "abc"
> cat("abc\n")
abc
```

Note that we needed to supply our own end-of-line character, "\n", in the call to cat(). Without it, our next call would continue to write to the same line.

The arguments to cat() will be printed out with intervening spaces:

:

```
> x
[1] 1 2 3
```

```
> cat(x,"abc","de\n")
1 2 3 abc de
```
If you don’t want the spaces, set sep to the empty string "", as follows:
```
> cat(x,"abc","de\n",sep="")
123abcde
```
Any string can be used for sep. Here, we use the newline character:
```
> cat(x,"abc","de\n",sep="\n")
1
2
3
abc
de
```
You can even set sep to be a vector of strings, like this:
```
> x <- c(5,12,13,8,88)
> cat(x,sep=c(".",".",".","\n","\n"))
5.12.13.8
88
```

Reading and writer Files

Working with text files, accessing files on remote machines, and getting file and directory information.

The function read.table() to read in a data frame. As a quick review, suppose the file z looks like this:
```
name age
John 25
Mary 28
Jim 19
```
The first line contains an optional header, specifying column names. We could read the file this way:
```
> z <- read.table("z",header=TRUE)
> z
name age
1 John 25
2 Mary 28
3 Jim 19
```
Note that scan() would not work here, because our file has a mixture of numeric and character data (and a header).

Reading Text Files

A distinction made between text files and binary files. That distinction is somewhat misleading—every file is binary
in the sense that it consists of 0s and 1s. Let’s take the term text file to mean a file that consists mainly of ASCII characters or coding for some other human language
Non text files, such as JPEG images or executable program files, are generally called binary files.
You can use readLines() to read in a text file, either one line at a time or in a single operation. For example, suppose we have a file z1 with the following contents:
```
John 25
Mary 28
Jim 19
```
We can read the file all at once, like this:
```
> z1 <- readLines("z1")
```

> z1
[1] "John 25" "Mary 28" "Jim 19"
Since each line is treated as a string, the return value here is a vector of strings—that is, a vector of character mode. There is one vector element for each line read, thus three elements here.

Graphics, Creating Graphs, The Workhorse of R Base Graphics, the plot() Function Customizing Graphs, Saving Graphs to Files. writer Files,

Graphics is a great strength of R. The graphics package is part of the standard distribution and contains many useful functions for creating a variety of graphic displays.

Notes on Graphics Functions

high-level graphics function starts a new graph. It initializes the graphics window (creating it if necessary); sets the scale; maybe draws some adornments, such as a title and labels; and renders the graphic. Examples include:

plot
Generic plotting function
boxplot
Create a box plot
hist
Create a histogram
qqnorm
Create a quantile-quantile (Q-Q) plot
curve
Graph a function

A low-level graphics function cannot start a new graph. Rather, it adds something to an existing graph: points, lines, text, adornments, and so forth. Examples include:

points
Add points
lines
Add lines
abline
Add a straight line
segments
Add line segments
polygon
Add a closed polygon
text
Add text

Creating a Scatter Plot

You have paired observations: (x1, y1), (x2, y2), ..., (xn, yn). You want to create a scatter plot of the pairs.

Solution

If your data are held in two parallel vectors, x and y, then use them as arguments of plot:

> plot(x, y)

Creating the scatter plot is easy:

> plot(x, y)

The plot function does not return anything. Rather, its purpose is to draw a plot of the (x, y) pairs in the graphics window.

Life is even easier if your data is captured in a two-column data frame. If you plot a twocolumn data frame, the function assumes you want a scatter plot created from the two columns. The scatter plot shown in **Figure** was created by one call to plot:

> plot(cars)

The cars dataset contains two columns, speed and dist. The first column is speed, so that becomes the x-axis and dist becomes the y-axis.

Adding a Title and Labels

When calling plot:

• Use the main argument for a title.
• Use the xlab argument for an x-axis label.
• Use the ylab argument for a y-axis label.

> plot(x, main="The Title", xlab="X-axis Label", ylab="Y-axis Label")

We can add them via arguments to plot, yielding the graphic shown in **Figure**

> plot(cars,
+ main="cars: Speed vs. Stopping Distance (1920)",
+ xlab="Speed (MPH)",
+ ylab="Stopping Distance (ft)")

Figure: Adding a title and labels

This is a clear improvement over the defaults. Now the title tells us something about the plot, and the labels give the measurement units.

Adding a Grid

Call plot with type="n" to initialize the graphics frame without displaying the data.
• Call the grid function to draw the grid.
• Call low-level graphics functions, such as points and lines, to draw the graphics overlaid on the grid.

plot(x, y, type="n")
grid()
points(x, y)

We can add a grid to the graphic First, we draw the graphic but with type="n". That will create a new plot, including title and axes, but will not (yet) plot the data:

> plot(cars, main="cars: Speed vs. Stopping Distance (1920)", xlab="Speed (MPH)", ylab="Stopping Distance (ft)", type="n")

Next we draw the grid:

> grid()
Finally, we draw the scatter plot by calling points, a low-level graphics function, thus creating the graph shown in **Figure:**
> points(cars)

Figure Adding a grid
Plotting multiple groups in one scatter plot creates an uninformative mess unless we distinguish one group from another. The pch argument of plot lets us plot points using a different plotting character for each (x, y) pair.
The iris dataset contains paired measures of **Petal.Length** and **Petal.Width.** Each measurement also has a Species property indicating the species of the flower that was measured. If we plot all the data at once, we just get the scatter plot shown in the lefthand panel of **Figure**
> with(iris, plot(Petal.Length, Petal.Width))
The graphic would be far more informative if we distinguished the points by species.
The pch argument is a vector of integers, one integer for each (x, y) point, with values between 0 and 18. The point will be drawn according to its corresponding value in pch. This call to plot draws each point according to its group:
> with(iris, plot(Petal.Length, Petal.Width, pch=as.integer(Species)))
The result is displayed in the right hand panel of which clearly shows the points in groups according to their species.

Figure Scatter plots of multiple groups

Adding a Legend

You want your plot to include a legend, the little box that decodes the graphic for the viewer
After calling plot, call the legend function. The first three arguments are always the same, but the arguments after that vary as a function of what you are labelling. Here is how you create legends for points, lines, and colors:
Legend for points
legend(x, y, labels, pch=c(pointtype1, pointtype2, ...))
Legend for lines according to line type
legend(x, y, labels, lty=c(linetype1, linetype2, ...))
Legend for lines according to line width
legend(x, y, labels, lwd=c(width1, width2, ...))
Legend for colors
legend(x, y, labels, col=c(color1, color2, ...))
Here, x and y are the coordinates of the legend box, and labels is a vector of the character strings to appear in the legend. The pch, lty, lwd, and col arguments are vectors that parallel the labels. Each label will be shown next to its corresponding point type, line type, line width, or color according to which one you specified.
>legend(1.5, 2.4, c("setosa","versicolor","virginica"), pch=1:3)
The left hand panel of **Figure** shows a legend added to the scatter plot of. The legend tells the viewer which points are associated with which species.
The right hand panel of **Figure** shows a legend for lines based on the line type (solid, dashed, or dotted). It was created like this:
> legend(0.5, 95, c("Estimate","Lower conf lim","Upper conf lim"),
lty=c("solid","dashed","dotted"))

Figure Examples of legends

Bar Chart

You want to create a bar chart. Use the barplot function. The first argument is a vector of bar heights:
> barplot(c(height1, height2, ..., heightn))
The barplot function produces a simple bar chart. It assumes that the heights of your bars are conveniently stored in a vector. That is not always the case, however. Often you have a vector of numeric data and a parallel factor that groups the data, and you want to produce a bar chart of the group means or the group totals.
For example, the air quality dataset contains a numeric Temp column and a Month column. We can create a bar chart of the mean temperature by month in two steps. First, we compute the means:
> heights <- tapply(airquality$Temp, airquality$Month, mean)
That gives the heights of the bars, from which we create the bar chart:
> barplot(heights)
The result is shown in the left hand panel of Figure The result is pretty bland, as you can see, so it’s common to add some simple adornments: a title, labels for the bars, and a label for the y-axis:
> barplot(heights, main="Mean Temp. by Month", names.arg=c("May", "Jun", "Jul", "Aug", "Sep"), ylab="Temp (deg. F)")
The right hand panel of Figure shows the improved bar chart.
Figure Bar charts, plain and adorned

Writing Your Plot to a File

You want to save your graphics in a file, such as a PNG, JPEG, or PostScript file. Create the graphics on your screen; then call the savePlot function:
> savePlot(filename="filename.ext", type="type")
The type argument is platform specific. See the Devices help page for a list of types available on your machine:
> help(Devices)
Most platforms allows a type of "png" or "jpeg", but each platform has others, too.
• Call a function to open a new graphics file, such as png ... or jpeg ... .
• Call plot and its friends to generate the graphics image.
• Call dev.off to close the graphics file.
Functions for opening graphics files depend upon your platform and file format. A common function is png, which creates a PNG file; it is used like this:
> png("filename.png", width=w, height=h)
Here w is the desired width and h is the desired height, both in pixels. Using that function, the recipe works like this to create myPlot.png:
> png("myPlot.png", width=648, height=432) # Or whatever dimensions work for you
> plot(x, y, main="Scatterplot of X, Y")

```
> dev.off()
```