STAT 37810 HW1

Boxin

10/11/2018

Note: In this homework, all the exercises are from second edition, except for section 4.4.1 exercise 1, 2. Sorry for the trouble this may cause to you.

section 4.1.1, exercises 2 (second edition)

(a)

```
# Construct a Fibonacci sequence first
Fibonacci <- numeric(30)
Fibonacci[1] <- Fibonacci[2] <- 1
for (i in 3:30){
  Fibonacci[i] <- Fibonacci[i - 2] + Fibonacci[i - 1]
}


# Use the Fibonacci sequence to constructe the new sequence
new.sequence <- numeric(30)
new.sequence[1] <- 1
for (i in 2:30) {
  new.sequence[i] <- Fibonacci[i] / Fibonacci[i - 1]
}
new.sequence
```

```
##  [1] 1.000000 1.000000 2.000000 1.500000 1.666667 1.600000 1.625000
##  [8] 1.615385 1.619048 1.617647 1.618182 1.617978 1.618056 1.618026
## [15] 1.618037 1.618033 1.618034 1.618034 1.618034 1.618034 1.618034
## [22] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
## [29] 1.618034 1.618034
```

The sequence apperas to be converging

(b)

```
(1 + sqrt(5)) / 2
```

```
## [1] 1.618034
```

It seems like the sequence is converging to golden ratio

Simple Proof: Let

$$a_n = f_n / f_{n-1}$$

The we can show that an satifies

$$a_{n+1} = 1 + 1/a_n$$

Assume that the limit of an is x, then x must satisfy

$$x = 1 + 1/x$$

Solve this equation and abandon the negative solution, we have

$$x = (1 + \sqrt{5})/2$$

section 4.1.1, exercises 3 (second edition)

(a)

```
answer <- 0
for (j in 1:5) answer <- answer + j
answer
```

```
## [1] 15
```

(b)

```
answer <- NULL
for (j in 1:5) answer <- c(answer, j)
answer
```

```
## [1] 1 2 3 4 5
```

(c)

```
answer <- 0
for (j in 1:5) answer <- c(answer, j)
answer
```

```
## [1] 0 1 2 3 4 5
```

(d)

```
answer <- 1
for (j in 1:5) answer <- answer * j
answer
```

```
## [1] 120
```

(e)

```
answer <- 3
for (j in 1:15) answer <- c(answer, (7 * answer[j]) %% 31)
answer
```

```
##  [1]  3 21 23  6 11 15 12 22 30 24 13 29 17 26 27  3
```

section 4.1.2, exercises 4 (second edition)

```
InterestEarnedByGIC <- function(P, n){
  if (n <= 3){
    total <- P * ((1 + 0.04) ^ n)
  }
  else{
```

```
    total <- P * ((1 + 0.05) ^ n)
  }
  return (total - P)
}
```

section 4.1.2, exercises 5 (second edition)

```
MonthlyMortgagePayment <- function(P, n, open){

  if (open == TRUE){
    i <- 0.005
  }
  else {
    i <- 0.004
  }

  R <- (P * i) / (1 - (1 + i) ^ (-n))

  return (R)
}
```

section 4.1.3, exercises 2 (second edition)

```
Fibonacci <- numeric(3)
Fibonacci[1] <- 1
Fibonacci[2] <- 1
Fibonacci[3] <- Fibonacci[1] + Fibonacci[2]
while (Fibonacci[length(Fibonacci)] < 300){
  Fibonacci[(length(Fibonacci) + 1)] <- Fibonacci[length(Fibonacci)] + Fibonacci[(length(Fibonacci) - 1
}
Fibonacci <- Fibonacci[1:(length(Fibonacci) - 1)]
Fibonacci
```

```
##  [1]   1   1   2   3   5   8  13  21  34  55  89 144 233
```

section 4.1.3, exercises 4 (second edition)

```
i1 <- 0.006
i2 <- (1 - (1 + i1) ^ (-20)) / 19
while (abs(i1 - i2) >= 0.000001){
  itemp <- (1 - (1 + i2) ^ (-20)) / 19
  i1 <- i2
  i2 <- itemp
}
i2
```

```
## [1] 0.004954139
```

```
i1 <- 1
i2 <- (1 - (1 + i1) ^ (-20)) / 19
while (abs(i1 - i2) >= 0.000001){
```

```r
  itemp <- (1 - (1 + i2) ^ (-20)) / 19
  i1 <- i2
  i2 <- itemp
}
i2
```

```
## [1] 0.004953779
```

```r
i1 <- 10
i2 <- (1 - (1 + i1) ^ (-20)) / 19
while (abs(i1 - i2) >= 0.000001){
  itemp <- (1 - (1 + i2) ^ (-20)) / 19
  i1 <- i2
  i2 <- itemp
}
i2
```

```
## [1] 0.004953779
```

When you try different starting guess, the answer has little change

section 4.1.3, exercises 5 (second edition)

```r
i1 <- 0.006
i2 <- (1 - (1 + i1) ^ (-20)) / 19
k <- 1
while (abs(i1 - i2) >= 0.000001){
  itemp <- (1 - (1 + i2) ^ (-20)) / 19
  i1 <- i2
  i2 <- itemp
  k <- k + 1
}
i2
```

```
## [1] 0.004954139
```

```r
k
```

```
## [1] 74
```

```r
i1 <- 0.1
i2 <- (1 - (1 + i1) ^ (-20)) / 19
k <- 1
while (abs(i1 - i2) >= 0.000001){
  itemp <- (1 - (1 + i2) ^ (-20)) / 19
  i1 <- i2
  i2 <- itemp
  k <- k + 1
}
i2
```

```
## [1] 0.004953499
```

```r
k
```

```
## [1] 106
```

```r
i1 <- 10
i2 <- (1 - (1 + i1) ^ (-20)) / 19
k <- 1
while (abs(i1 - i2) >= 0.000001){
  itemp <- (1 - (1 + i2) ^ (-20)) / 19
  i1 <- i2
  i2 <- itemp
  k <- k + 1
}
i2
```

```
## [1] 0.004953779
```

```r
k
```

```
## [1] 106
```

section 4.1.5, exercise 2 (second edition)

(b)

Suppose that there is a m, which is not prime. By the algorithm, m must can be represented as:

$$m = k_1 \times k_2 \quad k_1, k_2 > p$$

Thus

$$m > p^2 \geq n$$

Which is a contradiction

(c)

```r
Eratosthenes <- function(n) {
  # Print prime numbers up to n (based on the sieve of Eratosthenes)
  if (n >= 2) {
    sieve <- seq(2, n)
    primes <- c()
    while (length(sieve) > 0) {
      p <- sieve[1]
      if (p >= sqrt(n)) {
        primes <- c(primes, sieve)
        break
      }
      primes <- c(primes, p)
      sieve <- sieve[(sieve %% p) != 0]
    }
    return(primes)
  } else {
    stop("Input value of n should be at least 2.")
  }
}
```

section 4.2.1, exercises 2 (second edition)

(a)

```r
compound.interest <- function(P, i.r, n){
  return (P * ((1 + i.r) ^ n))
}
```

(b)

```r
compound.interest(1000, 0.01, 30)
```

```
## [1] 1347.849
```

section 4.2.1, exercises 3 (second edition)

```r
TestFunction <- function(x){
  return (x + 1)
}

threshold <- 0.01

CalculateZeroPoint <- function(f){
  for (i in c(-20:20)){
    if (f(i) < 0){
      low.bound <- i
      break
    }
  }
  for (i in c(-20:20)){
    if (f(i) > 0){
      up.bound <- i
      break
    }
  }
  while (abs(low.bound - up.bound) > threshold){
    mid <- (low.bound + up.bound) / 2
    if (f(mid) < 0){
      low.bound <- mid
    } else if (f(mid) > 0){
      up.bound <- mid
    } else {
      return (mid)
    }
  }
  return (mid)
}

CalculateZeroPoint(TestFunction)
```

```
## [1] -1.005859
```

section 4.4.1 exercise 1 (note: this is from the first edition)

```
# 1: Use a merge sort to sort a vector
mergesort <- function (x, decreasing=FALSE) {
  # Check for a vector that doesn't need sorting
  len <-length(x)
  if (len < 2) {
    result <- x
    return (result)
  }
  # 2: Check if the decreasing is true
  if (decreasing==TRUE){
    # 3a: sort x into result
    # 3.1a: split x in half
    y <- x[1:(len / 2)]
    z <- x[(len / 2 + 1):len]
    # 3.2a: sort y and z
    y <- mergesort(y, TRUE)
    z <- mergesort(z, TRUE)
    # 3.3a: merge y and z into a sorted result
    result <- c()
    # 3.3.1a: while (some are left in both piles)
    # 3.3.2a: put the biggest first element on the end
    # 3.3.3a: remove it from y or z
    while (min(length(y), length(z)) > 0) {
      if (y[1] > z[1]) {
        result <- c(result, y[1])
        y <- y[-1]
      } else {
        result <- c(result, z[1])
        z <- z[-1]
      }
    }
    # 3.3.4a: put the leftovers onto the end of result
    if (length(y) > 0) {
      result <- c(result, y)
    } else {
      result <- c(result, z)
    }
    return(result)
  } else {
    # 3b: sort x into result
    # 3.1b: split x in half
    y <- x[1:(len / 2)]
    z <- x[(len / 2 + 1):len]
    # 3.2b sort y and z
    y <- mergesort(y, FALSE)
    z <- mergesort(z, FALSE)
    # 3.3b merge y and z into a sorted result
    result <- c()
    # 3.3.1b while (some are left in both piles)
    # 3.3.2b put the biggest first element on the end
    # 3.3.3b: remove it from y or z
```

```r
    while (min(length(y), length(z)) > 0) {
      if (y[1] < z[1]) {
        result <- c(result, y[1])
        y <- y[-1]
      } else {
        result <- c(result, z[1])
        z <- z[-1]
      }
    }
    # 3.3.4b: put the leftovers onto the end of result
    if (length(y) > 0) {
      result <- c(result, y)
    } else {
      result <- c(result, z)
    }
    return(result)
  }
}

vector.test <- c(2, 3, 1, 4, 5, 6, 8, 9)
mergesort(vector.test)
```

```
## [1] 1 2 3 4 5 6 8 9
```

```r
mergesort(vector.test, TRUE)
```

```
## [1] 9 8 6 5 4 3 2 1
```

section 4.4.1 exercise 2 (note: this is from the first edition)

(a)

```r
# 1. Use Newton's method to solve the equations
NewtonMethod <- function(f, g, x0, y0){

  x1 <- x0
  x.origin <- x0
  x2 <- x1 + 1
  y1 <- y0
  y.origin <- y0
  y2 <- y1 + 1

  # Set threshold
  thresh <- 1e-7
  step.len <- 1e-14

  # 2: Keep iteration until the function value is close enough
  while (min(abs(f(x2, y2) - f(x.origin, y.origin)), abs(g(x2, y2) - g(x.origin, y.origin))) > thresh){

    # 3: Calculate the gradients and other numbers

    ## 3.1: Calculate the numerical differential
    gradient.fx.x1.y1 = (f(x1 + step.len, y1) - f(x1, y1)) / step.len
    gradient.fy.x1.y1 = (f(x1, y1 + step.len) - f(x1, y1)) / step.len
```

```r
    gradient.gx.x1.y1 = (g(x1 + step.len, y1) - g(x1, y1)) / step.len
    gradient.gy.x1.y1 = (g(x1, y1 + step.len) - g(x1, y1)) / step.len

    ## 3.2 Calculate other numbers
    f1 <- f(x1, y1)
    g1 <- g(x1, y1)
    d <- gradient.fx.x1.y1 * gradient.gy.x1.y1 - gradient.fy.x1.y1 * gradient.gx.x1.y1

    # 4: Compute next step
    x2 <- x1 - (gradient.gy.x1.y1 * f1 - gradient.fy.x1.y1 * g1) / d
    y2 <- y1 - (gradient.fx.x1.y1 * g1 - gradient.gx.x1.y1 * f1) / d

    # 5: Replace x1, y1 with x2, y2, and save the original x1
    #    y1 to compute the function value
    x.origin <- x1
    y.origin <- y1

    x1 <- x2
    y1 <- y2
  }

  # Return the answer
  return (c(x2, y2))
}
```

(b)

```r
test.fucntion1 <- function(x, y){
  return (x + y)
}

test.function2 <- function(x, y){
  return (x ^ 2 + 2 * (y ^ 2) - 2)
}

NewtonMethod(test.fucntion1, test.function2, 1, -1)
```

```
## [1]  0.8332 -0.8332
```

```r
NewtonMethod(test.fucntion1, test.function2, -1, 1)
```

```
## [1] -0.8332  0.8332
```

The analytical solutions are

```r
c(sqrt(2/3), -sqrt(2/3))
```

```
## [1]  0.8164966 -0.8164966
```

```r
c(-sqrt(2/3), sqrt(2/3))
```

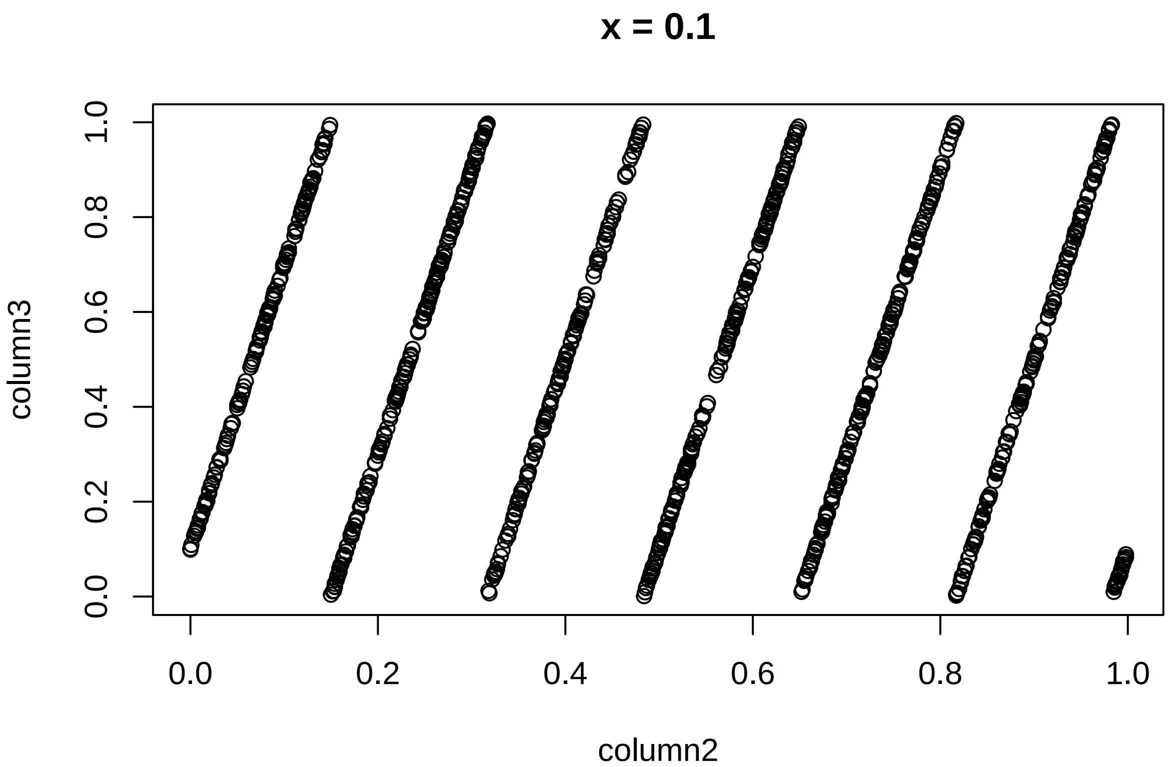
```
## [1] -0.8164966  0.8164966
```

Chapter 4 exercises 1 (second edition)

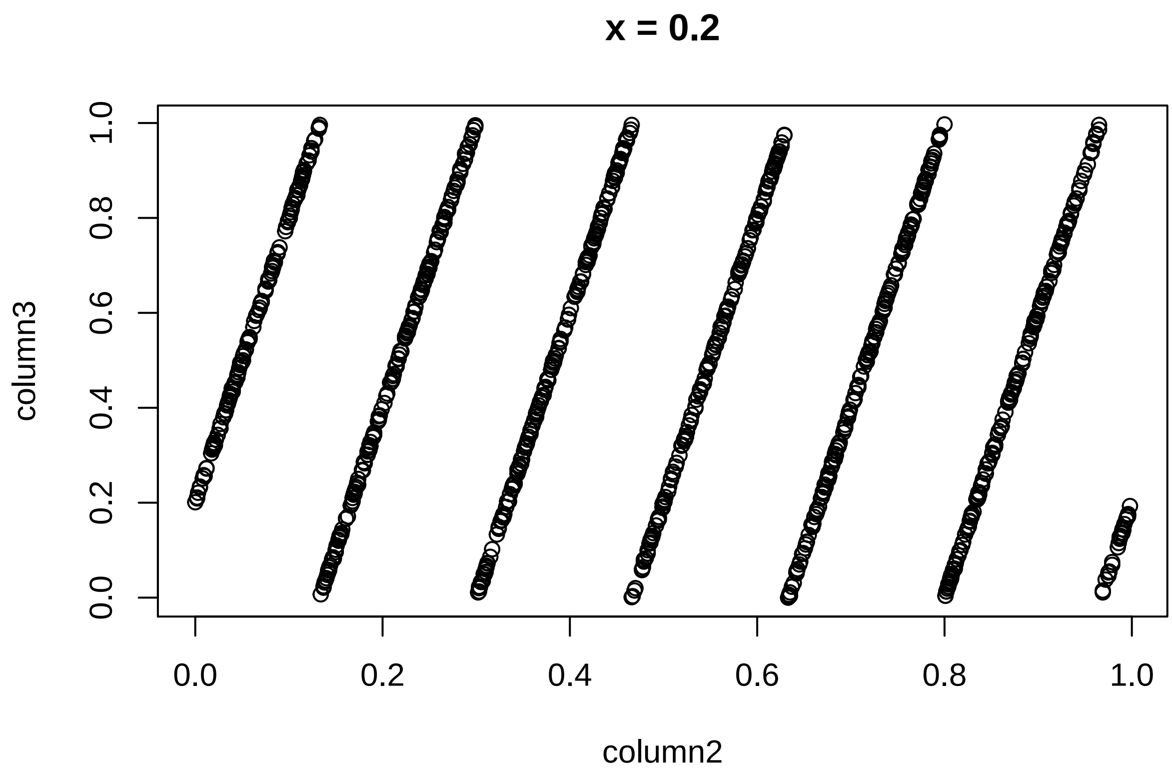
```r
results <- numeric(3000000)
x <- 123
for (i in 1:3000000) {
  x <- (65539 * x) %% (2 ^ 31)
  results[i] <- x / (2 ^ 31)
}
results <- round(results, 3)

m <- matrix(results, nrow=1000000, ncol=3, byrow=TRUE)
```

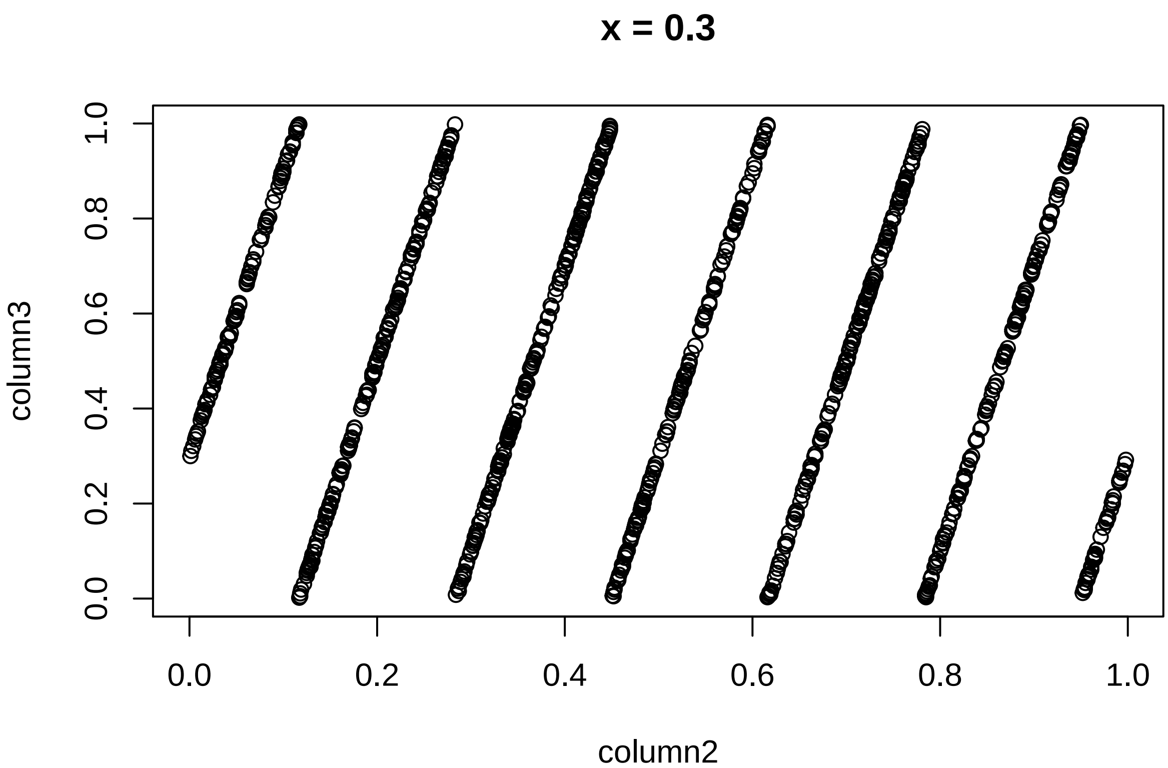
```r
x <- 0.1
m.sub <- m[(m[, 1] == x), ]
plot(m.sub[, 2], m.sub[, 3], main="x = 0.1", xlab="column2", ylab="column3")
```



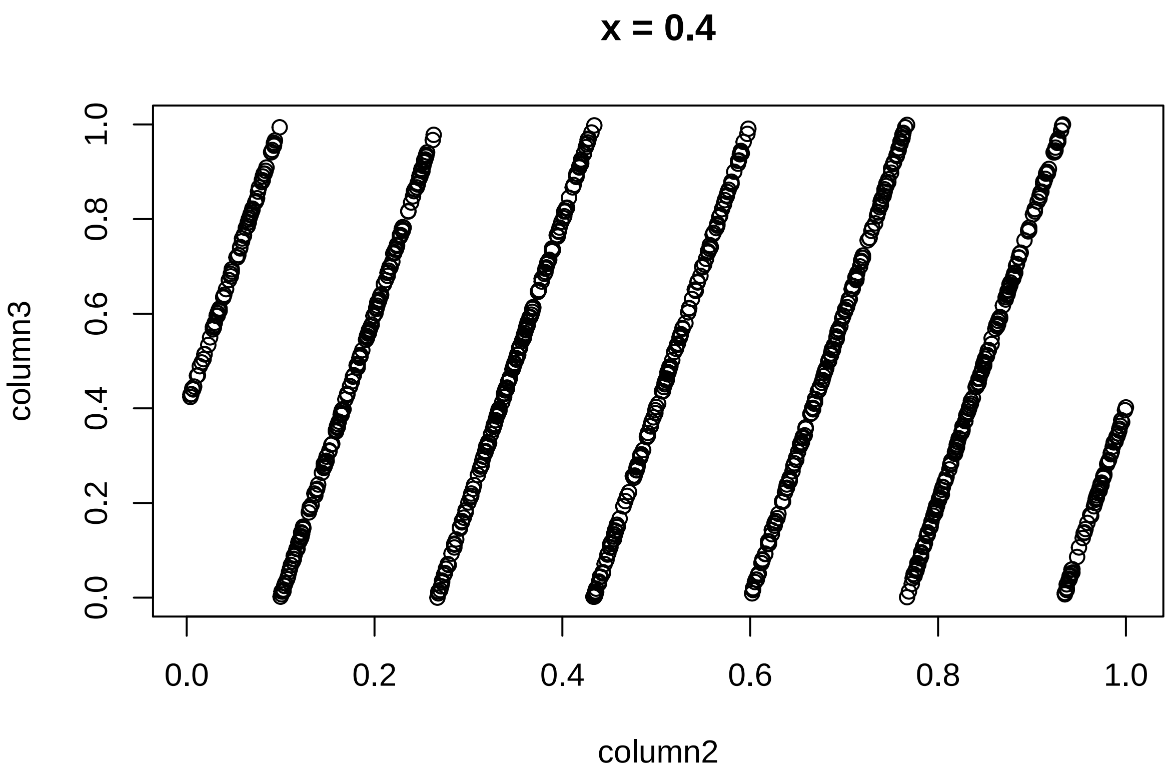

```r
x <- 0.2
m.sub <- m[(m[, 1] == x), ]
plot(m.sub[, 2], m.sub[, 3], main="x = 0.2", xlab="column2", ylab="column3")
```



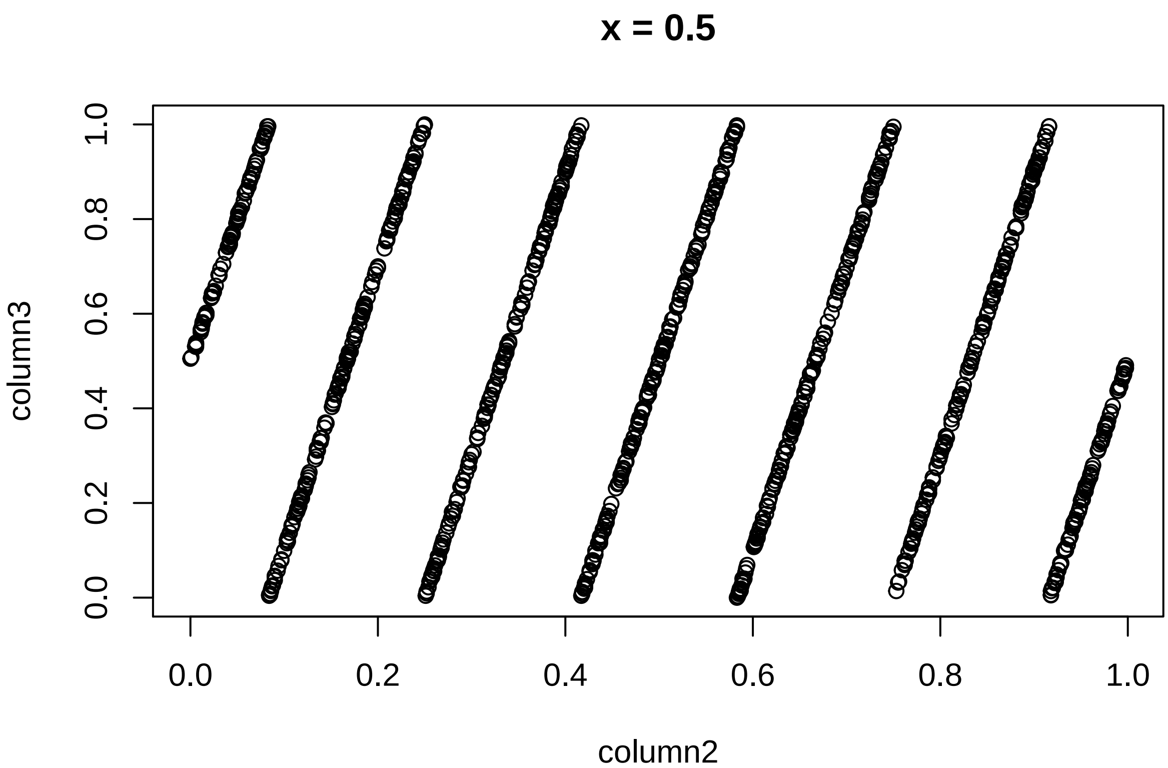

```
x <- 0.3
m.sub <- m[(m[, 1] == x), ]
plot(m.sub[, 2], m.sub[, 3], main="x = 0.3", xlab="column2", ylab="column3")
```



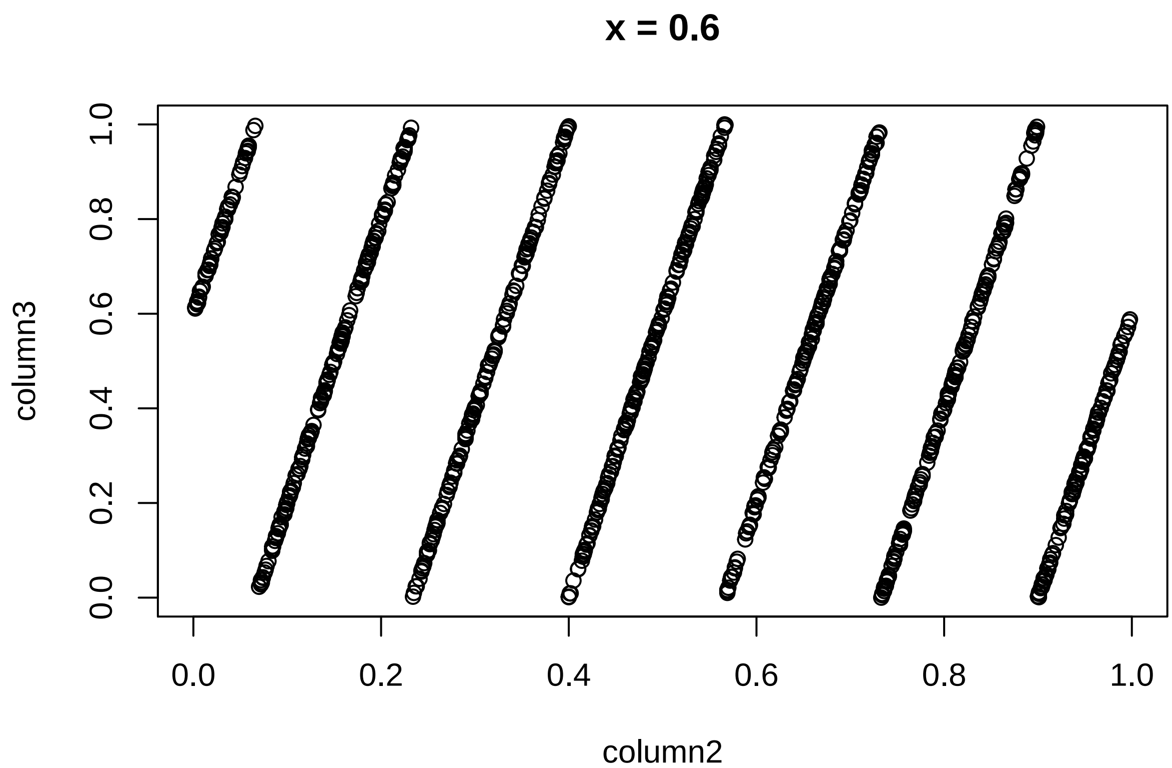

```
x <- 0.4
m.sub <- m[(m[, 1] == x), ]
plot(m.sub[, 2], m.sub[, 3], main="x = 0.4", xlab="column2", ylab="column3")
```



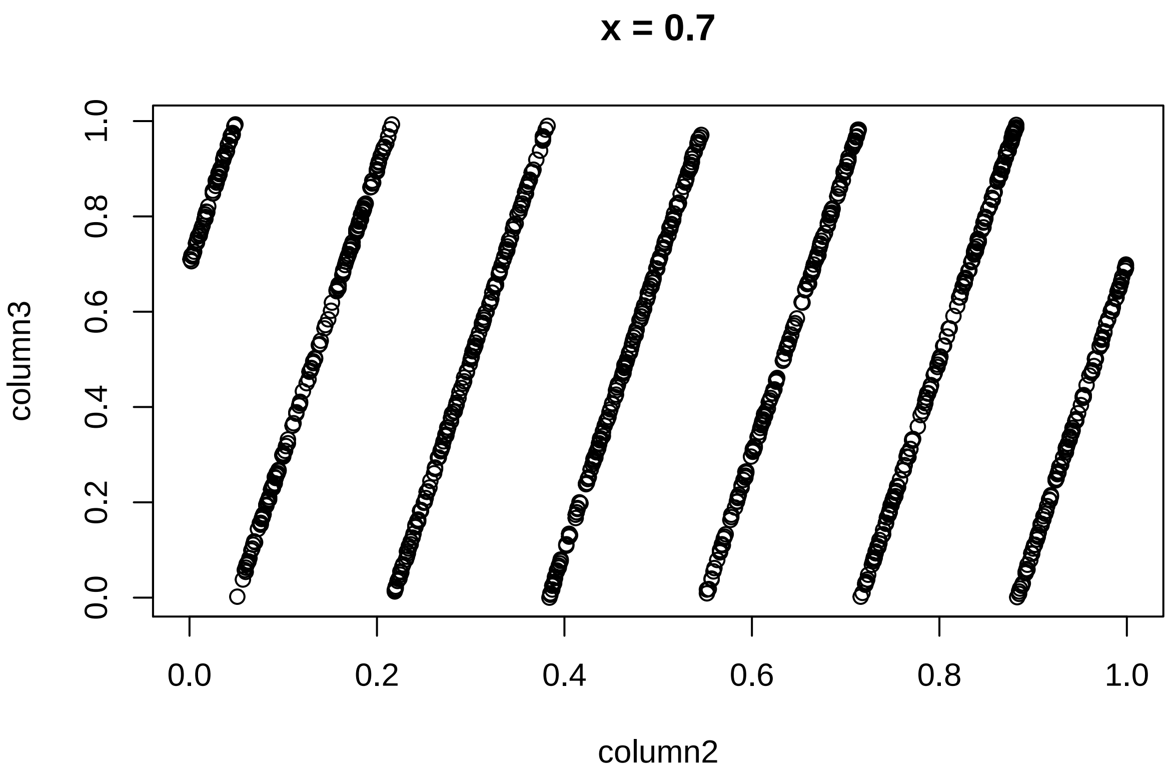

```r
x <- 0.5
m.sub <- m[(m[, 1] == x), ]
plot(m.sub[, 2], m.sub[, 3], main="x = 0.5", xlab="column2", ylab="column3")
```



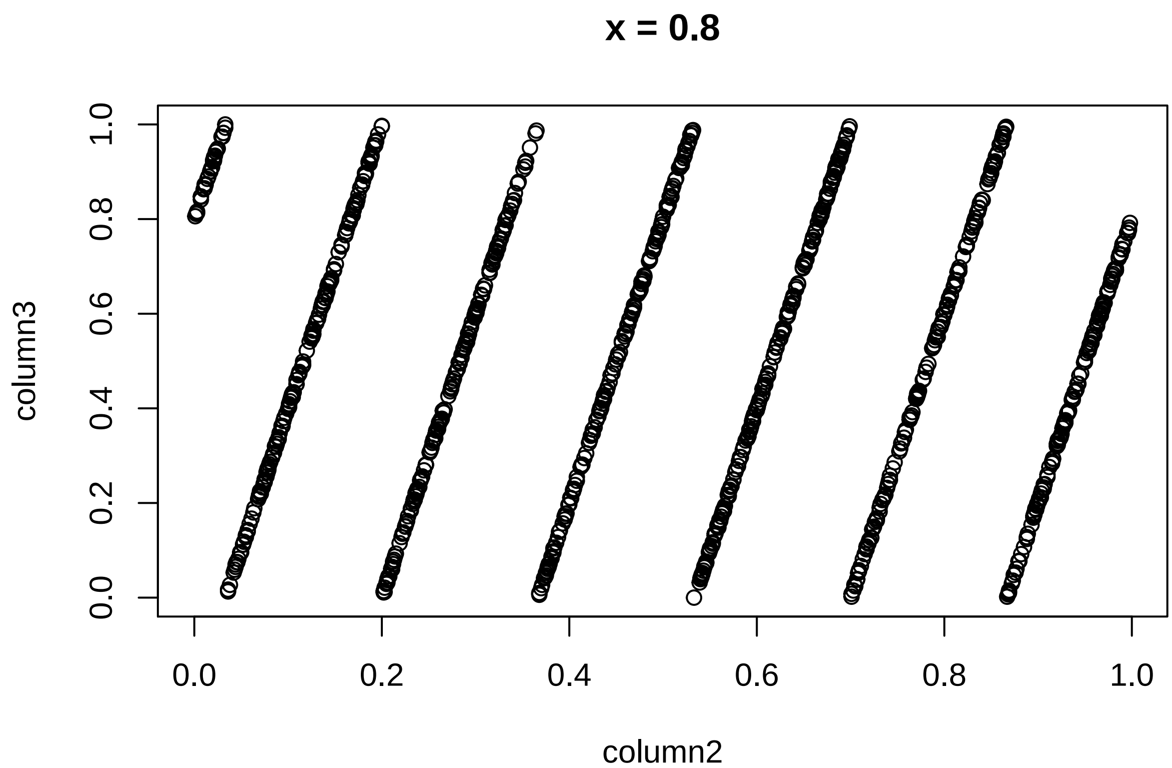

```r
x <- 0.6
m.sub <- m[(m[, 1] == x), ]
plot(m.sub[, 2], m.sub[, 3], main="x = 0.6", xlab="column2", ylab="column3")
```



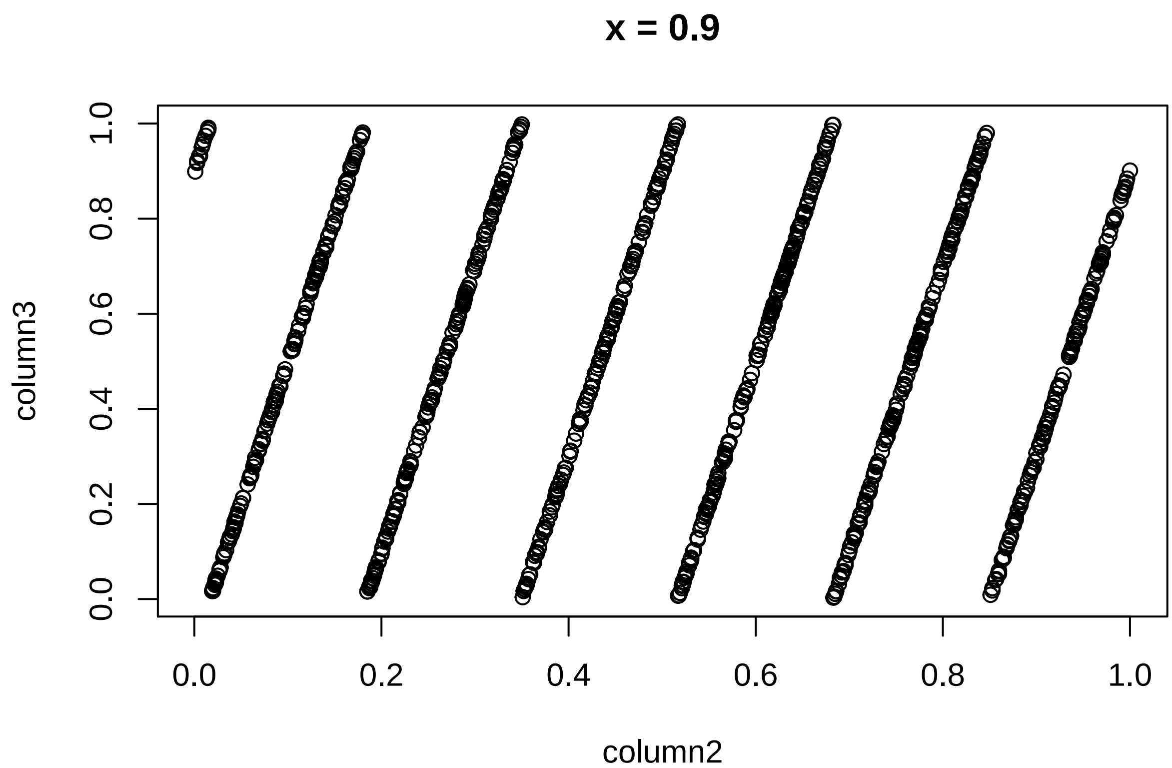

```
x <- 0.7
m.sub <- m[(m[, 1] == x), ]
plot(m.sub[, 2], m.sub[, 3], main="x = 0.7", xlab="column2", ylab="column3")
```




```r
x <- 0.8
m.sub <- m[(m[, 1] == x), ]
plot(m.sub[, 2], m.sub[, 3], main="x = 0.8", xlab="column2", ylab="column3")
```




```
x <- 0.9
m.sub <- m[(m[, 1] == x), ]
plot(m.sub[, 2], m.sub[, 3], main="x = 0.9", xlab="column2", ylab="column3")
```



Chapter 4 exercises 2 (second edition)

```
directpoly <- function(x, v){
  n <- length(v)
  result <- 0
  for (i in c(1:n)){
    result <- result + x ^ (i - 1) * v[i]
  }
  return (result)
}

directpoly(2, c(1, 2, 3))
```

```
## [1] 17
```

Chapter 4 exercises 3 (second edition)

```
hornerpoly <- function(y, c){
  result <- numeric(length(y))
  k <- 0
  for (x in y){
    n <- length(c)
    a <- numeric(n)
    a[n] <- c[n]
```

```r
    for (i in c((n - 1):1)){
      a[i] <- a[i+1] * x + c[i]
    }
    k <- k + 1
    result[k] <- a[1]
  }
  return (result)
}

hornerpoly(2, c(1, 2, 3))
```

```
## [1] 17
```

```r
hornerpoly(c(2, 3), c(1, 2, 3))
```

```
## [1] 17 34
```