MATH0154 Computational Statistics HW #1

Franklin Marsh 9/12/16 Prof Gabe Chandler

Problem 1

Recall the formula:

$V = 1.M \times 2^{(E-127)} \times (sign)$

If the value is -19.9375, this is [1011.1111] in binary. In binary scientific notation this value is:

$[1] * [1.00111111] \times 2^4$

So, we know $E - 127 = 4 \rightarrow E = 131$

$131 = [10000011]$ in binary. Thus:

$V = 110000011001111110...0 \rightarrow [1][10000011][001111110...0]$

Problem 2

In [8]:
```
FindMachineE <- function(init) {
  final <- 1 # the final value that we will arrive at is 1

  while (final + init > final) { #while 1 plus the macihne-e test valu
e is still greater than 1
    init <- init/2.0 #divide our initial guess in half
  }
return(init)
}

FindMachineE(0.5)
```

1.11022302462516e-16

Problem 3

In [115]:
```
G.Prime <- function(x) {
  #returns the value of the function G.prime evaluated at x.
  #
  # Args:
  #   x: the value at which to evaluate the function
  #
  # Returns:
  #   g'(x)
  return((1 + 1/x - log(x)) / ((1+x) ^ 2))
}


Bisector <- function(ifunc, boundlist) {
    # finds the root of a function (inputfunc) using the bisection met
hod
    #
    # Args:
    #   ifunc: the function that we wish to find the root of
    #   boundlist: a list of length 2 with the left-bound first, and t
hen the right bound
    # Returns:
    #   outs: a list of length 2 with the updated boundaries.
    #the initial guess, constructed out of the left and right bounds
    iters = 0

    a0 = as.numeric(boundlist[1])
    b0 = as.numeric(boundlist[2])

    if (ifunc(a0)*ifunc(b0) >= 0){
        stop("Root not in Bounds")
    }

    else {

        x0 <- (a0 + b0)/2.0

        if (ifunc(a0)*ifunc(x0) < 0.0) {
            at <- a0
            bt <- x0

        }

        else if (ifunc(a0)*ifunc(x0) > 0.0) {
            at <- x0
            bt <- b0
```

```
        }

        }
    return (list(at,bt))
}

GeneralIterRecord <- function(ifunc, ifunc2, seedlist, n_iter) {

  # iterate a recursive function over a list of numbers, recording eac
h input and output.
  #
  # Args:
  #   ifunc: a function with
  #   ifunc2:
  #   seedlist: a list of the inputs for the first run of inputfunctio
n
  #   n_iter: the number of iterations to be performed
  #
  #
  #
  # Returns:
  #   results: list of function outputs.

  results <- list(seedlist)

  for (i in 2:n_iter) {
    results[[i]] <- ifunc(ifunc2, results[[i-1]])
  }

  return(results)
}
```

In [186]:
```
GeneralIterRecord(Bisector, G.Prime, list(0,100), 30)
```

1. A. 0
   B. 100
2. A. 0
   B. 50
3. A. 0
   B. 25
4. A. 0
   B. 12.5
5. A. 0
   B. 6.25
6. A. 3.125
   B. 6.25
7. A. 3.125

B. 4.6875
8. A. 3.125
B. 3.90625
9. A. 3.515625
B. 3.90625
10. A. 3.515625
B. 3.7109375
11. A. 3.515625
B. 3.61328125
12. A. 3.564453125
B. 3.61328125
13. A. 3.5888671875
B. 3.61328125
14. A. 3.5888671875
B. 3.60107421875
15. A. 3.5888671875
B. 3.594970703125
16. A. 3.5888671875
B. 3.5919189453125
17. A. 3.59039306640625
B. 3.5919189453125
18. A. 3.59039306640625
B. 3.59115600585938
19. A. 3.59077453613281
B. 3.59115600585938
20. A. 3.59096527099609
B. 3.59115600585938
21. A. 3.59106063842773
B. 3.59115600585938
22. A. 3.59110832214355
B. 3.59115600585938
23. A. 3.59110832214355
B. 3.59113216400146
24. A. 3.59112024307251
B. 3.59113216400146
25. A. 3.59112024307251
B. 3.59112620353699
26. A. 3.59112024307251
B. 3.59112322330475
27. A. 3.59112024307251
B. 3.59112173318863
28. A. 3.59112098813057

B. 3.59112173318863
29. A. 3.5911213606596
B. 3.59112173318863
30. A. 3.5911213606596
B. 3.59112154692411

It appears that the root of the function to five decimal places is 3.49112

Problem 4

In [8]:
```
G.1 <- function(x) {
  # returns the value of the function $G_{1}$ in problem 4.
  #
  # Args:
  #   x: the value at which to evaluate the function
  #
  # Returns:
  #   $G_{1}(x)$

  e <- exp(1)

  return((x + e^(-x))/2.0)
}

G.2 <- function(x) {
  # returns the value of the function $G_{2}$ in problem 4.
  #
  # Args:
  #   x: the value at which to evaluate the function
  #
  # Returns:
  #   $G_{2}(x)$

  e <- exp(1)

  return(e^(-x))
}

G.3 <- function(x) {
  # returns the value of the function $G_{3}$ in problem 4.
  #
  # Args:
  #   x: the value at which to evaluate the function
  #
  # Returns:
```

```
  #   $G_{3}(x)$

  e <- exp(1)

  return(-log(x))
}

G2.prime <- function(x) {
  # returns the value of the function $g'_{x}$ in problem 4.
  #
  # Args:
  #   x: the value at which to evaluate the function
  #
  # Returns:
  #   $g'(x)$

  return(x + log(x))
}

IterRecord <- function(inputfunction, seed, n_iter) {

  # iterate a recursive function over a list of numbers, recording eac
h input and output.
  #
  # Args:
  #   inputfunction: the "seed" or $X_{0}$, initial value
  #   seed: the input for the first run of inputfunction
  #   n_iter: the number of iterations to be performed
  #
  #
  #
  # Returns:
  #   results: list of function outputs.

  results <- list(seed)

  for (i in 2:n_iter) {
    results[i] <- inputfunction(as.numeric(results[i-1]))
  }

  return(as.numeric(results))
}
```

We will find the root of $g'(x) = x + \log(x)$:

$0 = x + \log(x) \rightarrow x = -\log(x)$

So, that's the root. Let's make the substitution into our three equations:

1

$G_1(x) = \frac{x+e^{-x}}{2}$

$G_1(-\log(x)) = \frac{x+e^{-(-\log(x))}}{2}$

$G_1(-\log(x)) = \frac{x+e^{\log(x)}}{2} = \frac{x+x}{2} = x = -\log(x)$

2

$G_2(x) = e^{-x}$

$G_2(-\log(x)) = e^{-(-(\log(x)))} = e^{\log(x)} = x = -\log(x)$

3

$G_3(x) = -\log(x) = -\log(-\log(x)) = -\log(-\log(-\log(x)))\ldots$

We can show that G_3 is not contractive.

$G_3(x) = -\log(x)$, which has the derivative $-\frac{1}{x}$. For $0 < x < 1$, this derivative is greater than 1, which is not allowed by the condition $0 < \lambda < 1$. For $[1, \infty)$, $G_3(x)$ maps the value to a negative, which is not in the set $[1, \infty)$. Thus $G_3(x)$ is not contractive over the reals.

In [188]:
```
G.1_solution <- IterRecord(G.1, 2, 100) # run the iterative solver and record all values
G.2_solution <- IterRecord(G.2, 2, 100) # run the iterative solver and record all values
G.3_solution <- IterRecord(G.3, 2, 100) # run the iterative solver and record all values
```

```
Warning message:
In log(x): NaNs produced
```

In [189]:
```
print(G.1_solution) #show me the values
print(G.2_solution)
print(G.3_solution)
```

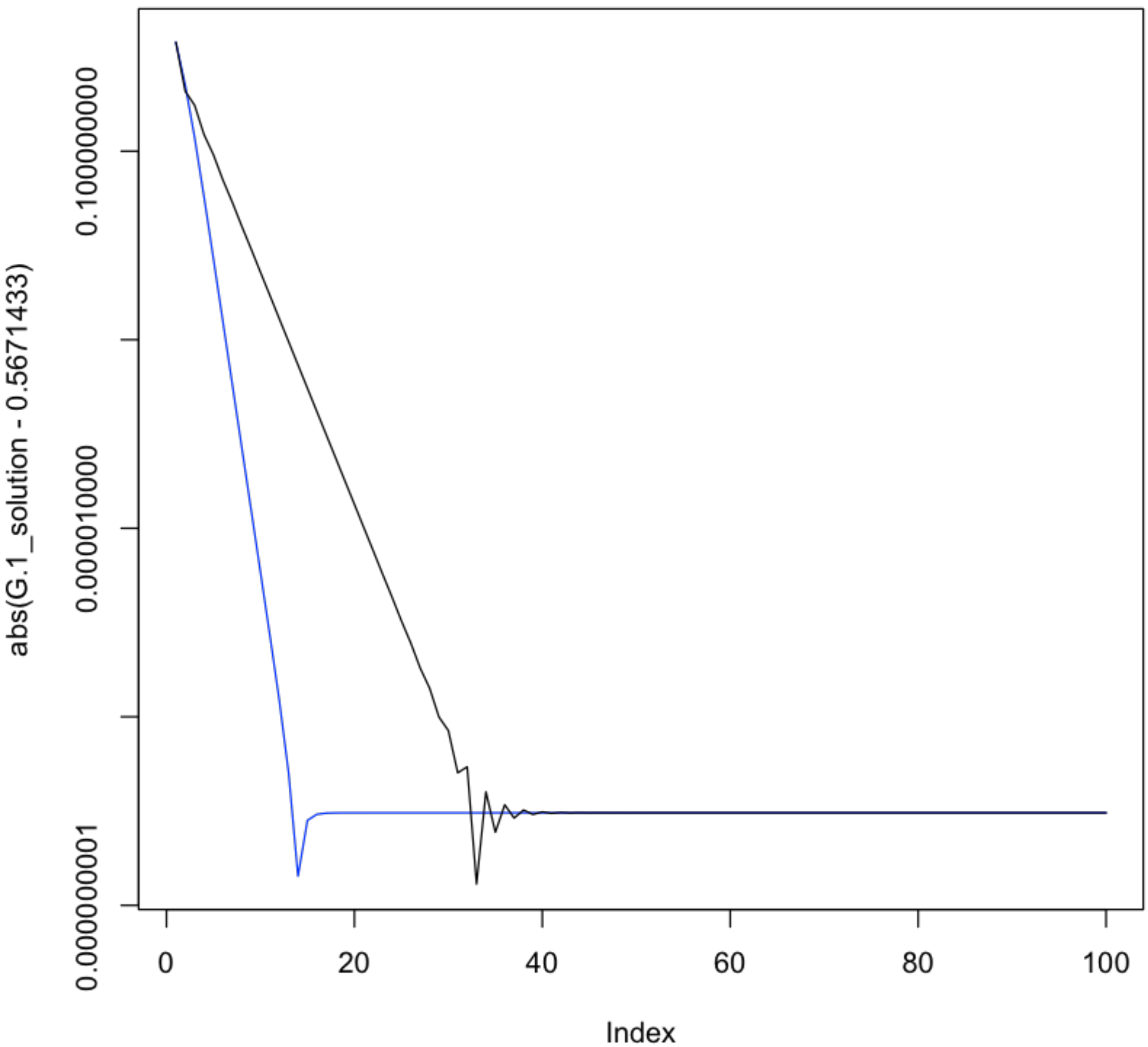
```
  [1] 2.0000 1.0677 0.7057 0.5997 0.5743 0.5687 0.5675 0.5672 0.5672 0
.5671
 [11] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
```

```
.5671
 [21] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [31] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [41] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [51] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [61] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [71] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [81] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [91] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
  [1] 2.0000 0.1353 0.8734 0.4175 0.6587 0.5175 0.5960 0.5510 0.5764 0
.5619
 [11] 0.5701 0.5655 0.5681 0.5666 0.5674 0.5670 0.5672 0.5671 0.5672 0
.5671
 [21] 0.5672 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [31] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [41] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [51] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [61] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [71] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [81] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
 [91] 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0.5671 0
.5671
  [1]  2.0000 -0.6931     NaN     NaN     NaN     NaN     NaN     NaN
NaN
 [10]     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
NaN
 [19]     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
NaN
 [28]     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
NaN
 [37]     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
NaN
 [46]     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
NaN
 [55]     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
```

```
NaN
 [64]     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
NaN
 [73]     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
NaN
 [82]     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
NaN
 [91]     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
NaN
[100]     NaN
```

In [190]:

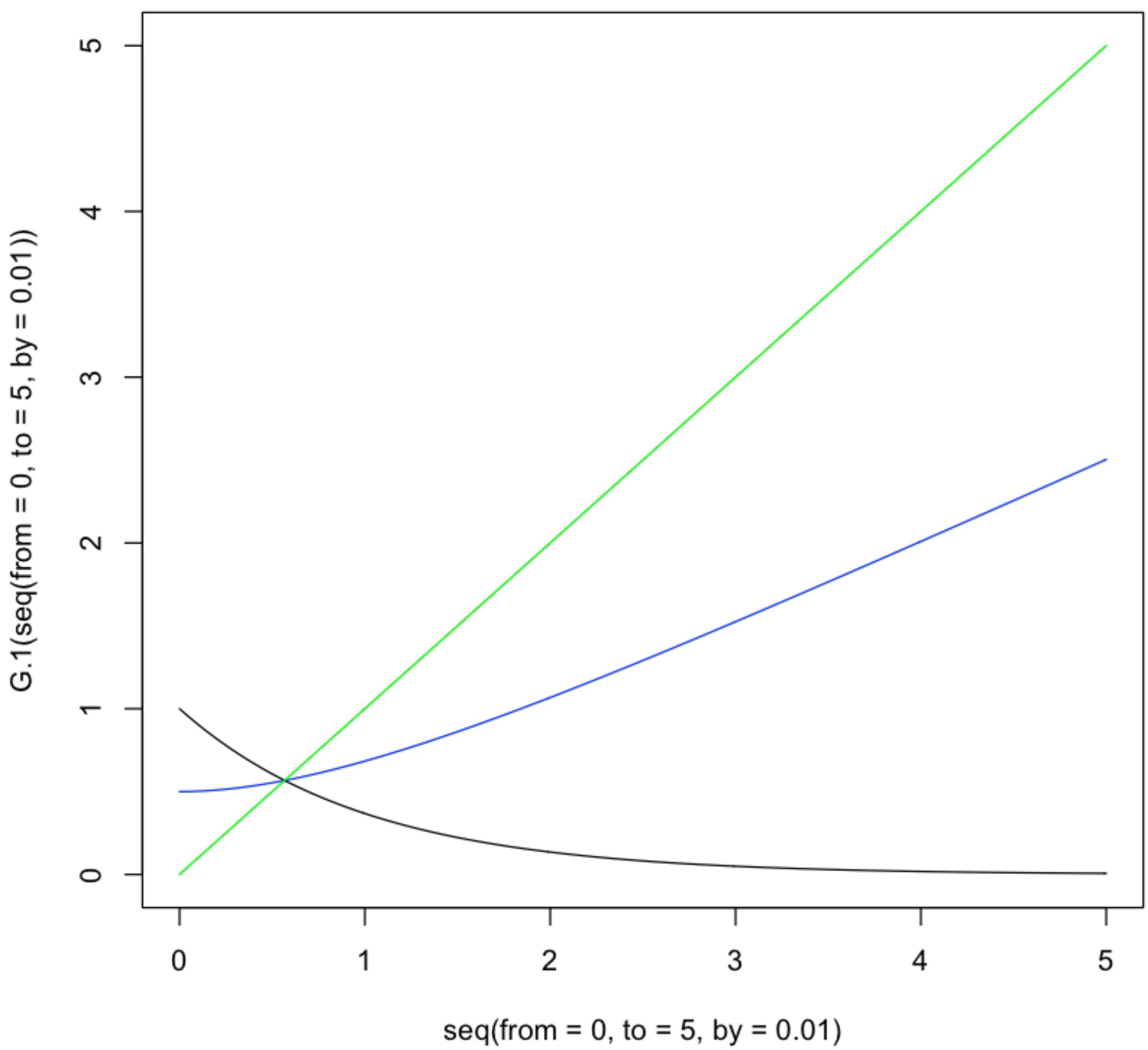
```
plot(abs(G.1_solution - 0.5671433),type = 'l', log = 'y', col = 'blue'
)
par(new=T)
lines(abs(G.2_solution - 0.5671433), type = 'l')
par(new=F)
```



Apparently G_1 (plotted in blue) converges faster, in about 17 iterations as opposed to about 40.

This is perhaps because the slope of G_1 near the root is closer to 1 than the slope of G_2

In [152]:
```
plot(seq(from = 0, to = 5, by = 0.01), G.1(seq(from = 0, to = 5, by =
0.01)), type = 'l', col = 'blue',xlim = c(0,5), ylim = c(0,5))
par(new=T)
lines(seq(from = 0, to = 5, by = 0.01), G.2(seq(from = 0, to = 5, by =
0.01)), type = 'l')
par(new=T)
lines(seq(from = 0, to = 5, by = 0.01), seq(from = 0, to = 5, by = 0.0
1), type = 'l', col = 'green')
par(new=F)
```

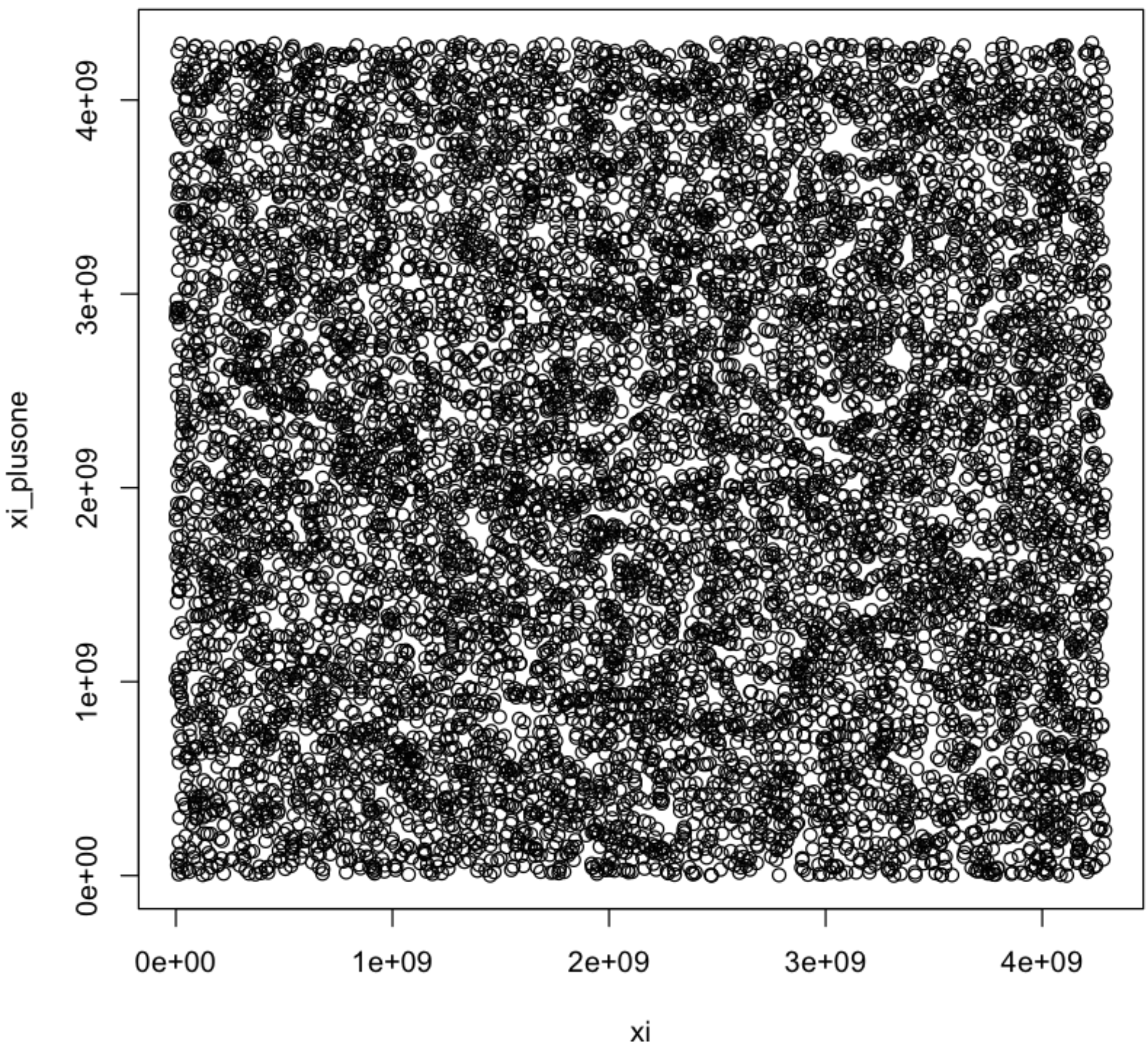


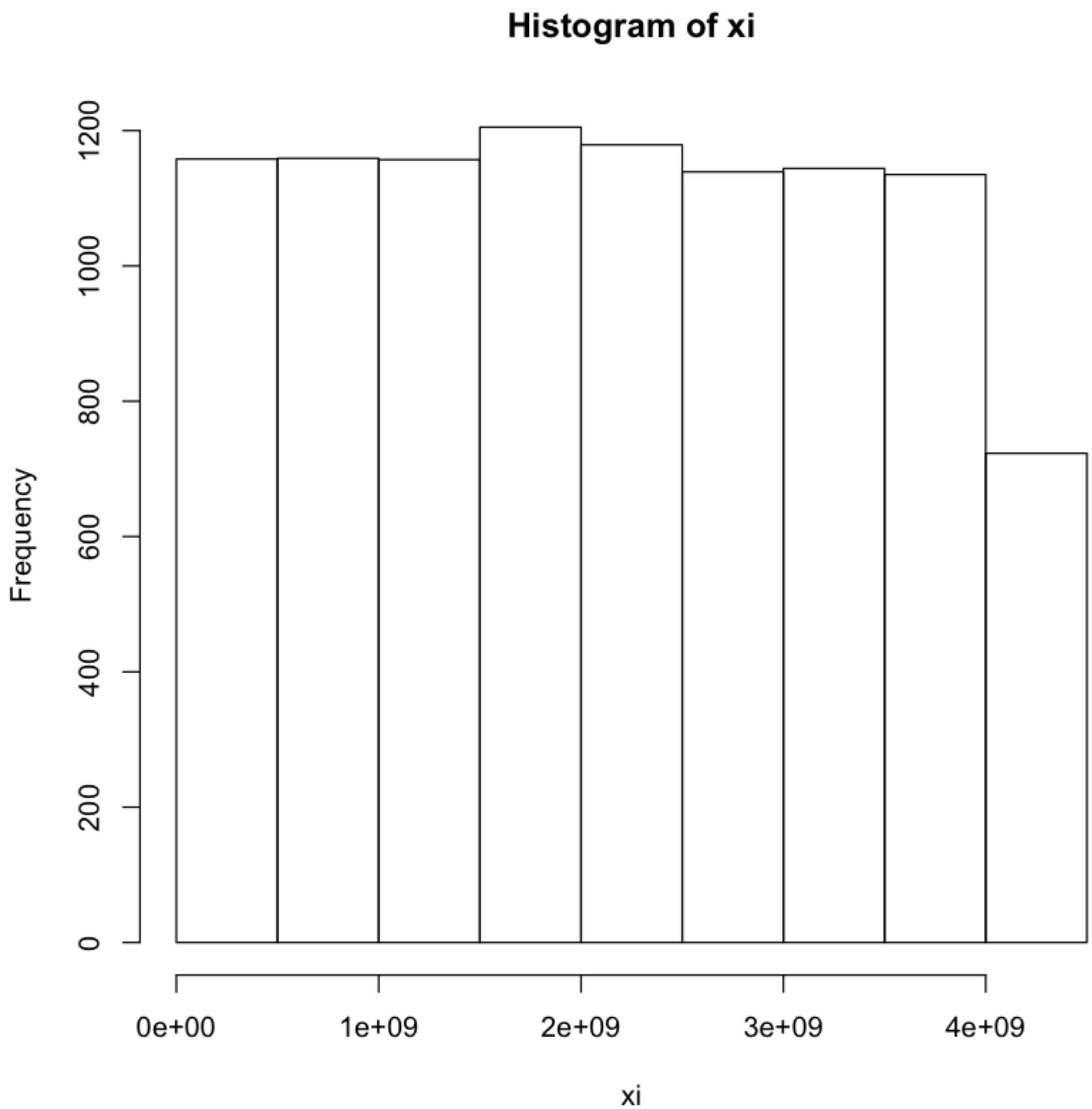
Problem 5

In [125]:
```
MCG_nr <- function(x, a = 1664525,c = 1013904223 ,m = 2^32) {
  # mixed congruential generator function as recommended by Numerical 
Recipies
  #
  # Args:
  #   x: the "seed" or $X_{0}$, initial value
  #   a: the multiplier, 0 < a < m
  #   c: the increment, 0 <= c < m
  #   m: the modulus
  #
  # Returns:
  #   x_1: the next number in the sequence.

  return(((a*x + c) %% m))
}

IterRecord <- function(inputfunction, seed, n_iter) {

  # iterate a recursive function over a list of numbers, recording eac
h input and output.
  #
  # Args:
  #   inputfunction: the "seed" or $X_{0}$, initial value
  #   seed: the input for the first run of inputfunction
  #   n_iter: the number of iterations to be performed
  #
  #
  #
  # Returns:
  #   results: list of function outputs.

  results <- list(seed)

  for (i in 2:n_iter) {
    results[i] <- inputfunction(as.numeric(results[i-1]))
  }

  return(as.numeric(results))
}
```

Part (a)

In [128]:
```
rand_numbers = IterRecord(MCG_nr, 2, 10000)
xi <- rand_numbers[1:length(rand_numbers) - 1]
xi_plusone <- rand_numbers[2:length(rand_numbers)]

plot(xi, xi_plusone)
hist(xi)
```



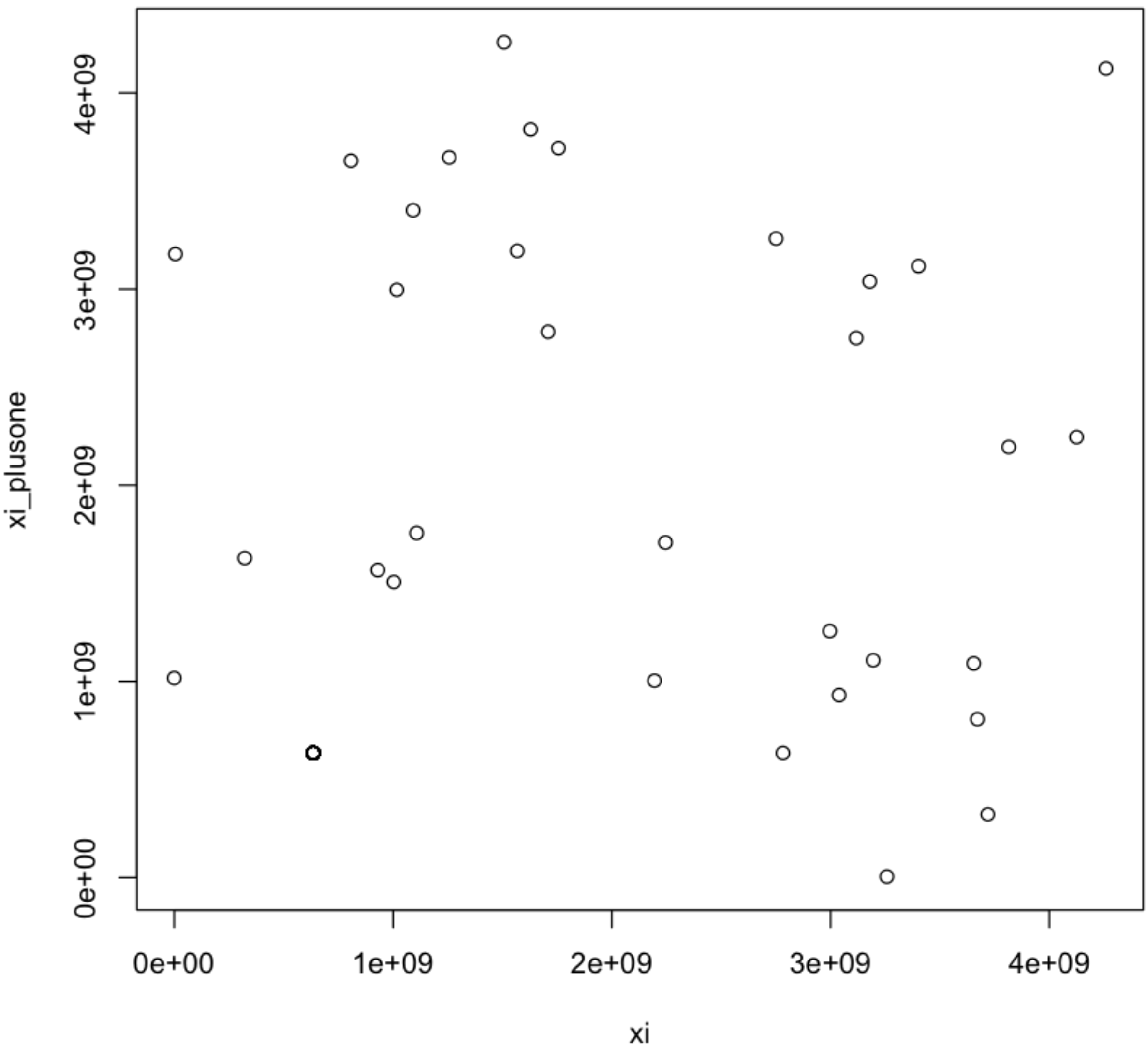


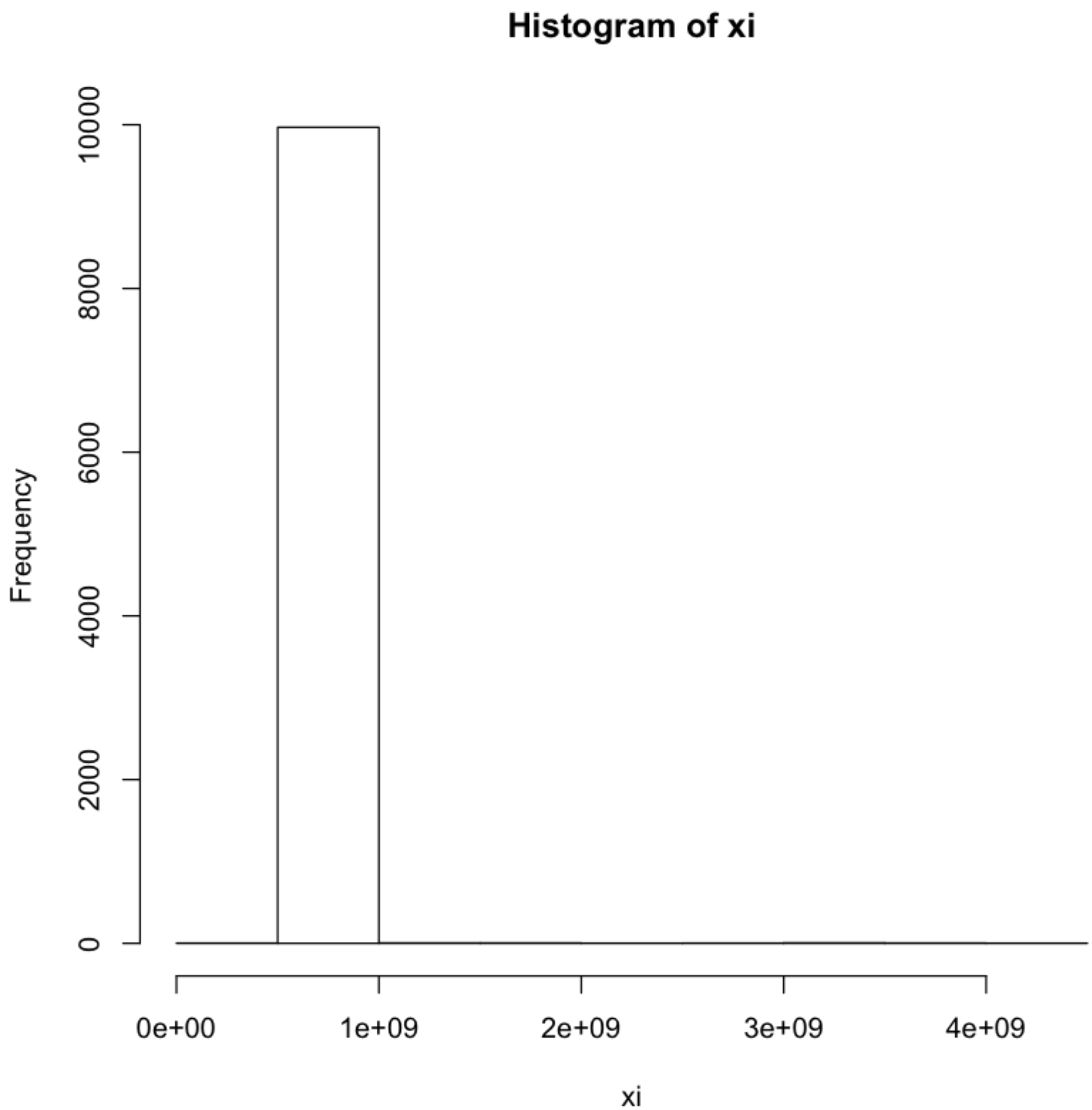
The distribution is uniform, except for the last bine (which straddles the edge of the range). There does not appear to be any correlation between x_i and x_{i+1}

Part (b)

In [129]:
```
MCG_dirty <- function(x, a = 1664526, c = 1013904223, m = 2^32) {
  # mixed congruentieal generator, with a = 1664526
  #
  # Args:
  #   x: the "seed" or $X_{0}$, initial value
  #   a: the multiplier, 0 < a < m
  #   c: the increment, 0 <= c < m
  #   m: the modulus
  #
  # Returns:
  #   x_1: the next number in the sequence.

  return(((a*x + c) %% m))
}
```

In [130]:
```
rand_numbers = IterRecord(MCG_dirty, 2, 10000)
xi <- rand_numbers[1:length(rand_numbers) - 1]
xi_plusone <- rand_numbers[2:length(rand_numbers)]

plot(xi, xi_plusone)
hist(xi)
```


xi_plusone
0e+00
1e+09
2e+09
3e+09
4e+09
xi

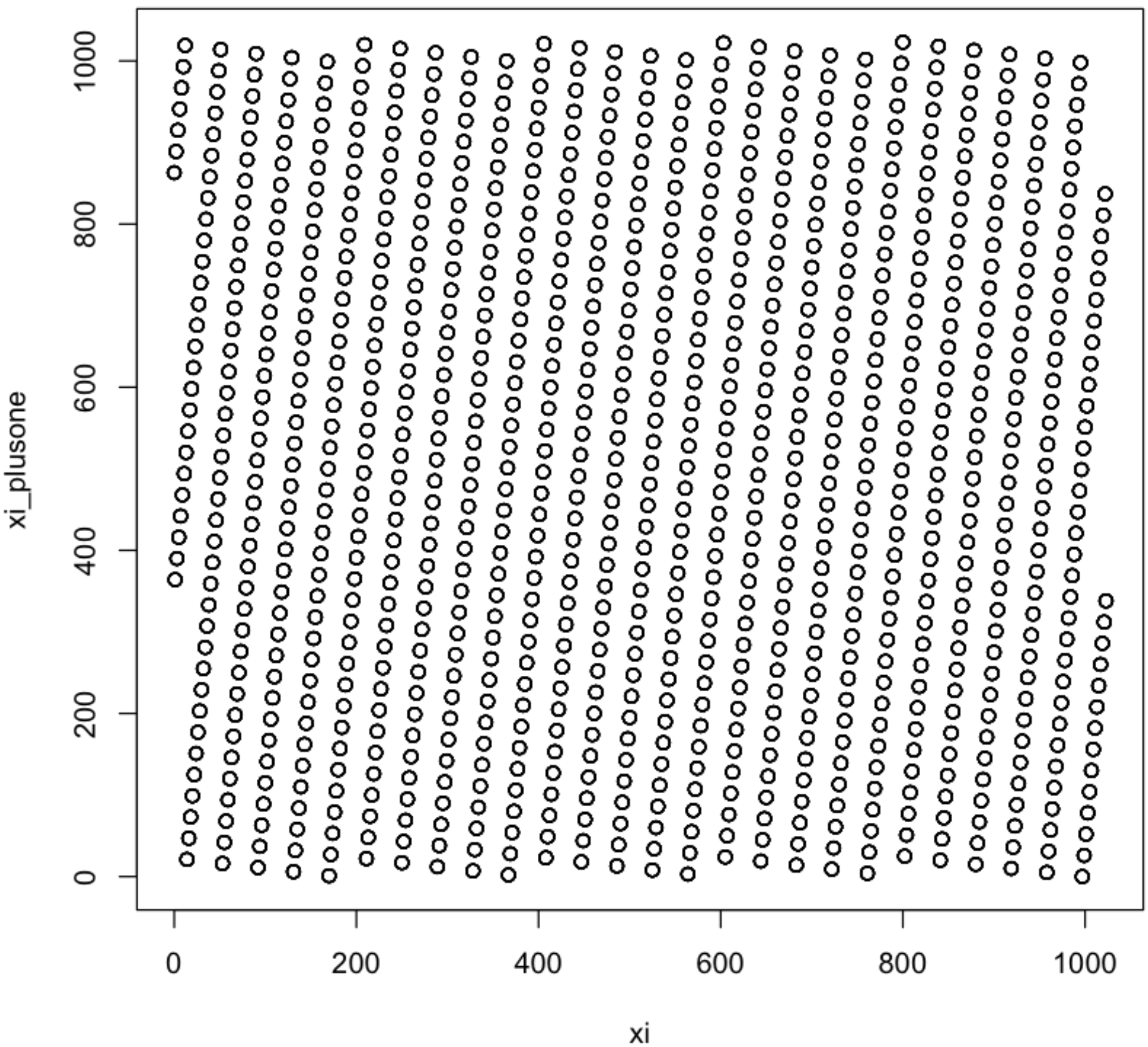


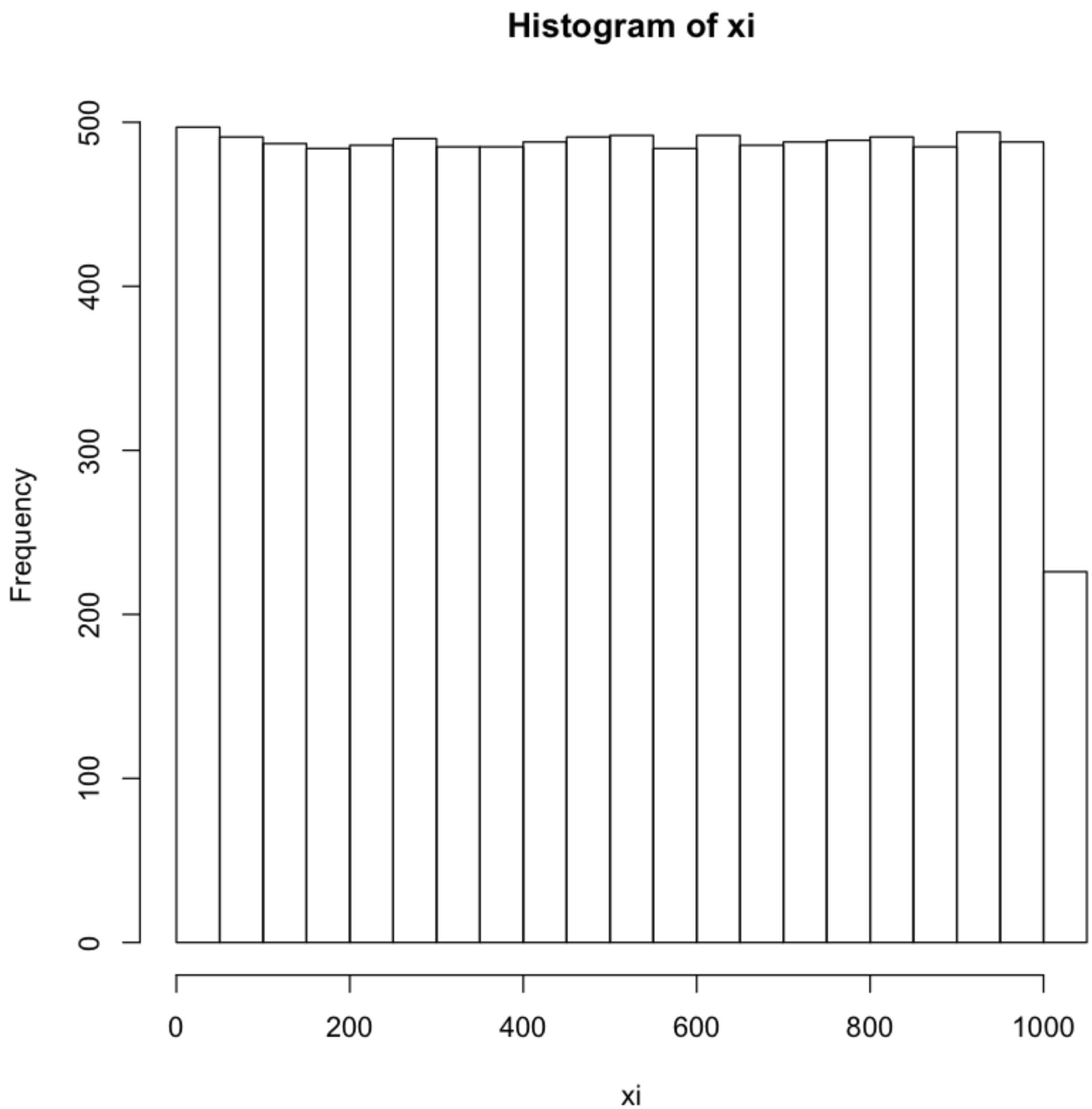
There is trouble because the MCG is looping through a very limited set of numbers. The histoggram shows that the vast majority of points occupy a small range (less than 10 percent) of the possible values. - this is not good for creating random numbers.

Part (c)

In [131]:
```
MCG_smallm <- function(x, a = 1664525, c = 1013904223, m = 2^10) {
  # mixed congruentieal generator, with m set to the small values of 2
^10
  #
  # Args:
  #   x: the "seed" or $X_{0}$, initial value
  #   a: the multiplier, 0 < a < m
  #   c: the increment, 0 <= c < m
  #   m: the modulus
  #
  # Returns:
  #   x_1: the next number in the sequence.

  return(((a*x + c) %% m))
}
```

In [132]:
```
rand_numbers = IterRecord(MCG_smallm, 2, 10000)
xi <- rand_numbers[1:length(rand_numbers) - 1]
xi_plusone <- rand_numbers[2:length(rand_numbers)]

plot(xi, xi_plusone)
hist(xi)
```


xi_plusone
0
200
400
600
800
1000
xi
0
200
400
600
800
1000



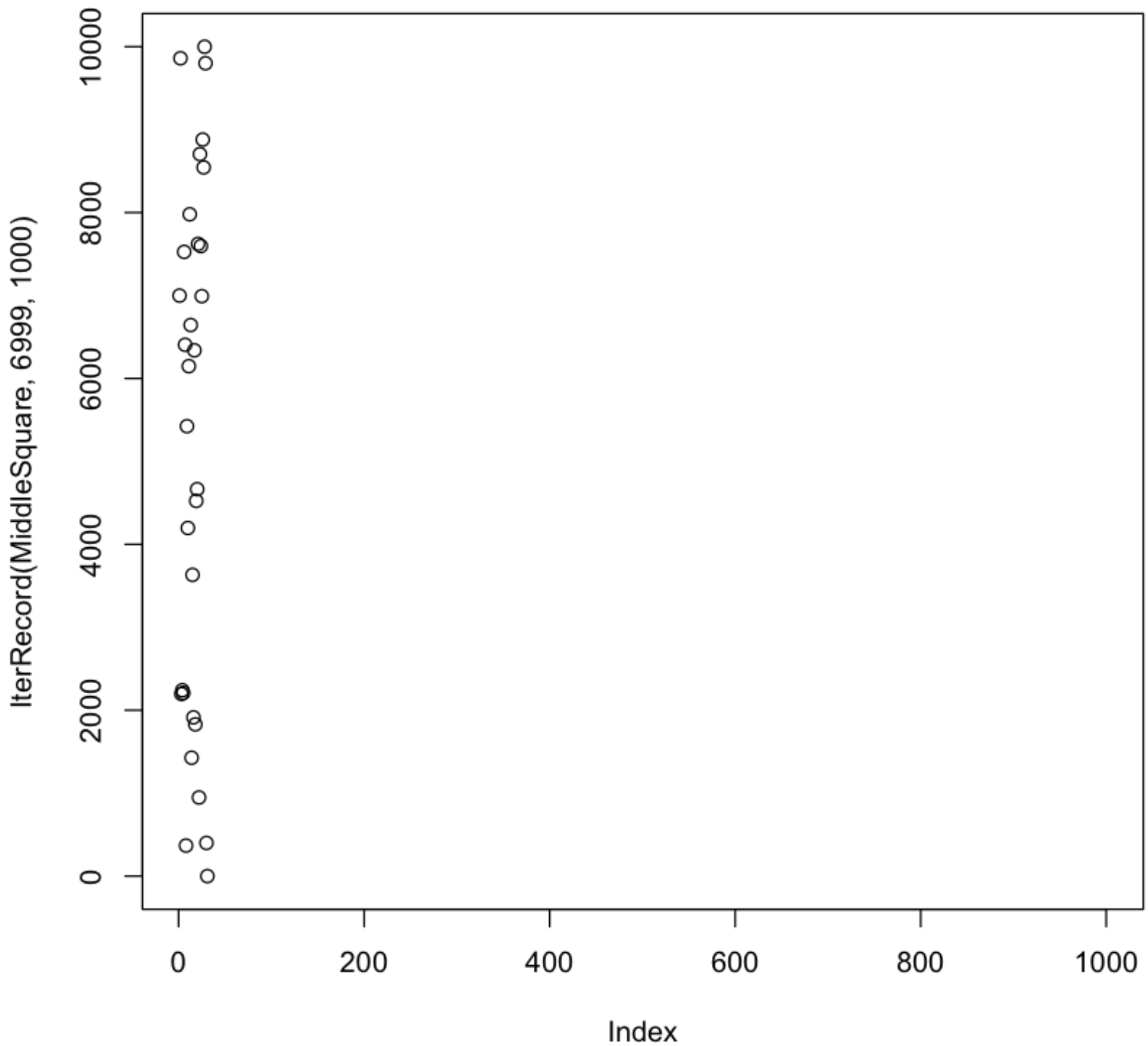
These do not seem like good values because only a small set of numbers are being created by the MCG. This gives the scatter plot a grid-like appearance.

Part (d)

In [133]:
```
MiddleSquare <- function(seed) {
  # compute a 'random' sequence of four-digit numbers using the Middle
Square method.
  #
  # Args:
  #   seed: the intial four digit numbeMir
  #
  # Returns:
  #   rand: new quasi-random number

  options("scipen" = 100, "digits" = 4) #disable scientific notation s
o the character operation works.
  seed_squared <- toString(seed^(2.0))
  rand <- as.numeric(substr(seed_squared, 3,6)) #take the middle slice
out of an 8 digit number and return to numeric
  options("scipen" = 8, "digits" = 4)

  return(rand)
}
```

As we can see in the next plot, for certain values, the MiddleSquare generator will run until it hits zero, and therefore is not a good pseudo-random number generator.

In [153]:
```
plot(IterRecord(MiddleSquare, 6999, 1000))
```

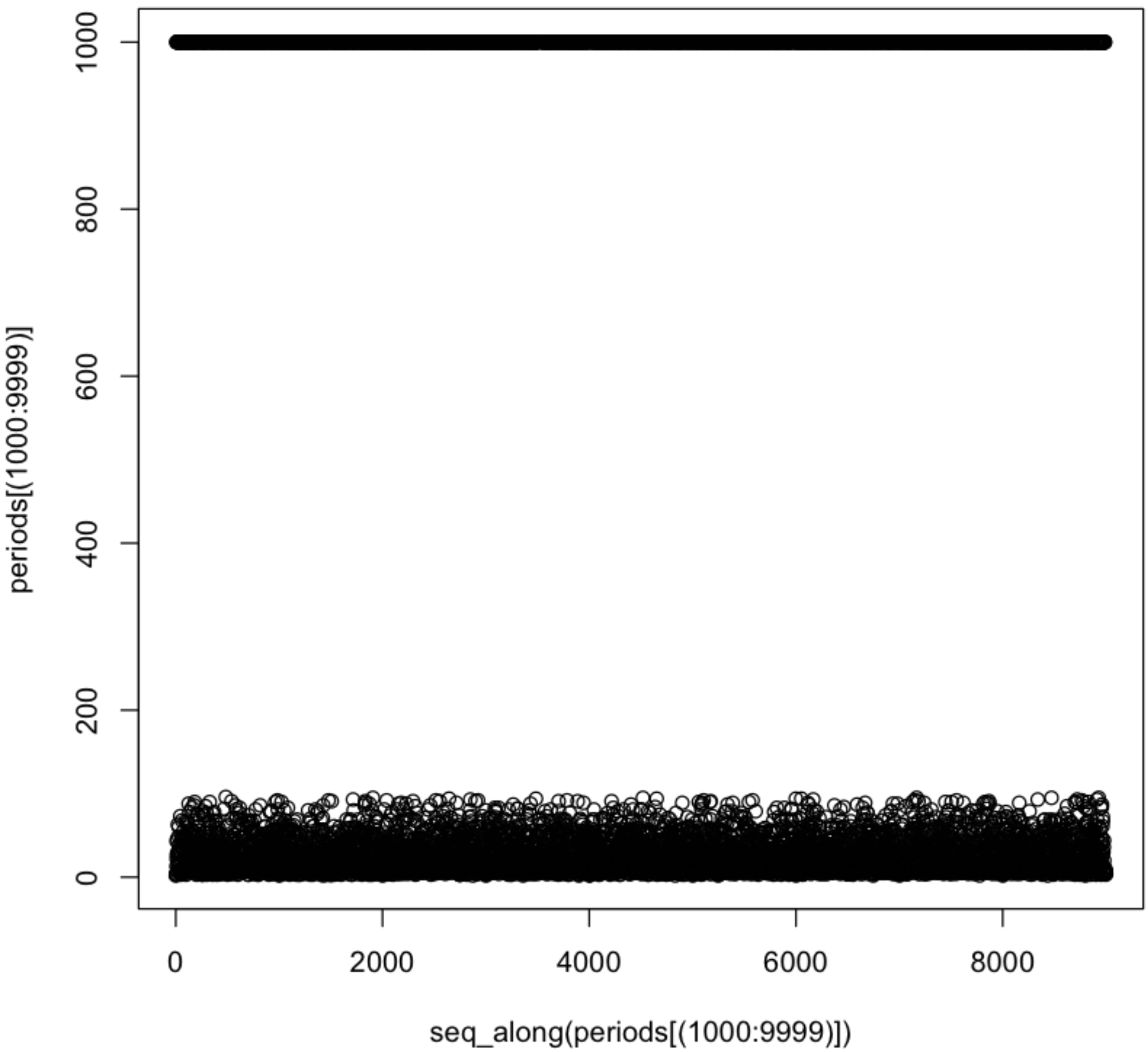


This next function returns how many iteration it takes for MiddleSquare to send the value to 0.

In [161]:
```
LenNotNA <- function(x) {
  # return length of the input vector after the NA values have been re
moved from it
  #
  # Args:
  #   x: input vector, as a list
  #
  # Returns:
  #   l: float of the length of the input vector

    l <- length(x[!is.na(x)])

    return(l)
}
```

In [167]:
```
periods = list() #initialize empty list to hold periods

for (i in 1000:9999) {
    periods[i] <- LenNotNA(IterRecord(MiddleSquare, i, 1000))
}
```

In [185]:
```
plot(seq_along(periods[(1000:9999)]), periods[(1000:9999)])
```



There seems to be quite a few values which we can't get more than 100 random numbers out of. Also, an obvious problem with this generator is that it maps 0000 to 0000.