Tricks and Traps for Young Players

Ray D Brownrigg

Statistical Computing Manager
School of Engineering and Computer Science
Victoria University of Wellington

Wellington, New Zealand

ray@ecs.vuw.ac.nz

UseR! 2011

Coventry, August 2011

CONTENTS

CONTENTS (continued)

1. Background

   Items encountered during a simulation research project using a computation grid of approximately 150 unix workstations.

2. Introduction

   Calculate the distribution function of the supremum of a normalised two-dimensional independent poisson process. This simulates Brownian Motion, which appears as a limiting process in goodness-of-fit studies.

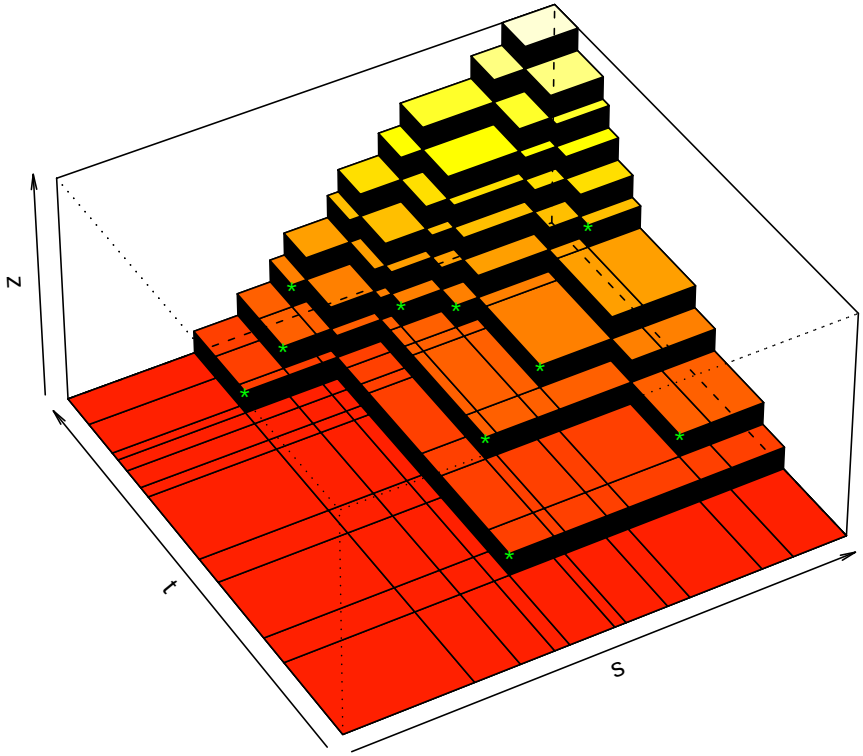
   – throw down N points on unit square

   – calculate difference between density and expected density at every point on the square

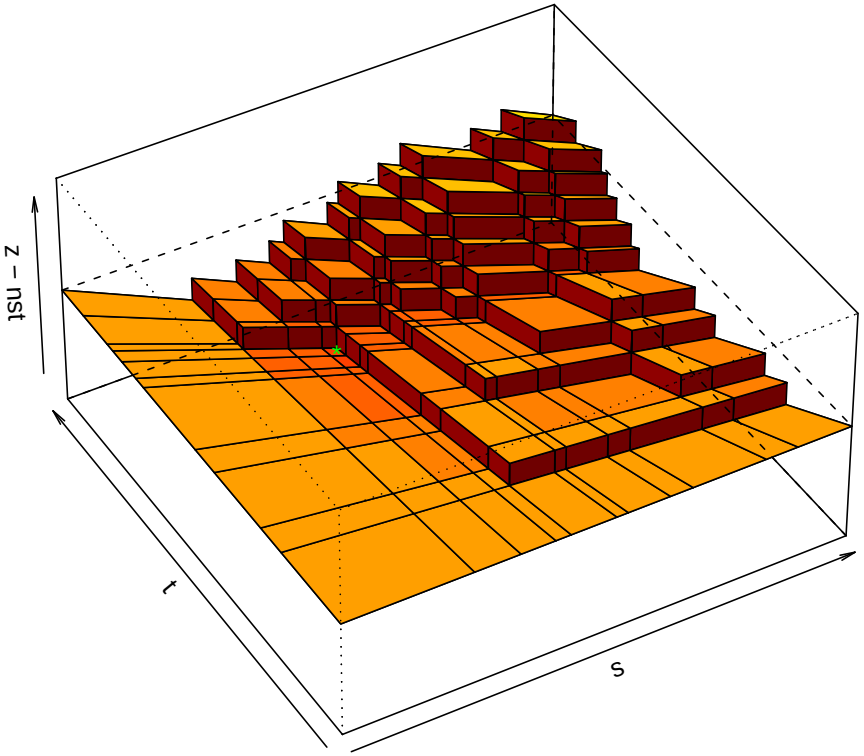
   – find supremum

E.g.:

Example plot of $\xi_n(s, t)$



Example plot of $\xi_n(s, t) - nst$



– goal is to have N as large as computationally possible, given we need a large number of repetitions

– basic exhaustive search algorithm is $O(N^3)$ **(a)**

– Fortran gives >1 order of magnitude speedup (12-40x) **(f)**

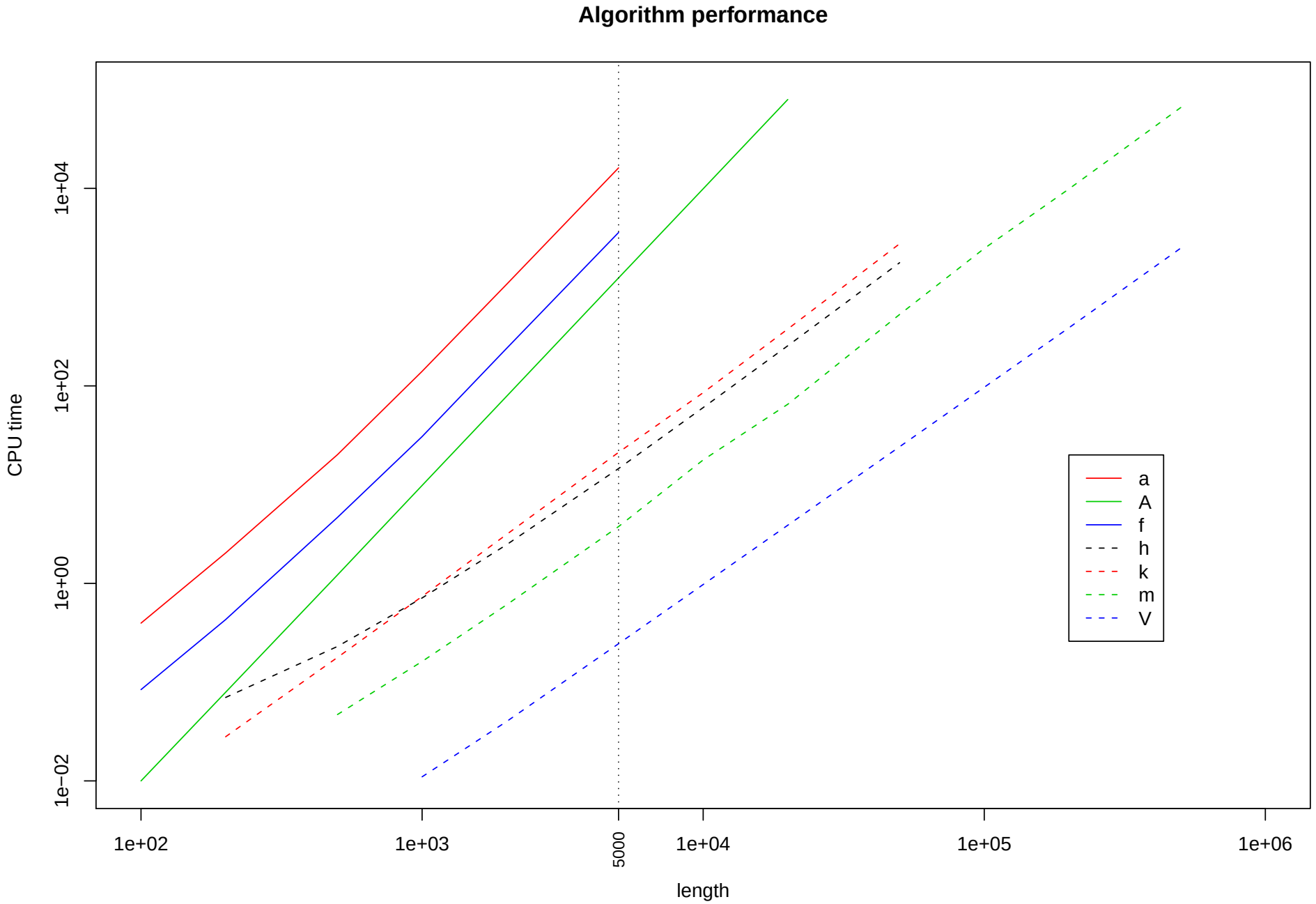
– restructuring to single loop using `cumsum()` and `order()` is generally faster than the initial Fortran **(A), (h)**

– now $O(N^2)$

– further improvements save another factor of 3 **(k), (m)**

– now Fortran gives 1.5 orders of magnitude (i.e. 30x) **(V)**

– overall 5 orders of magnitude speed improvement


Algorithm performance
CPU time
1e−02
1e+00
1e+02
1e+04
1e+02
1e+03
5000
1e+04
1e+05
1e+06
length
a
A
f
h
k
m
V

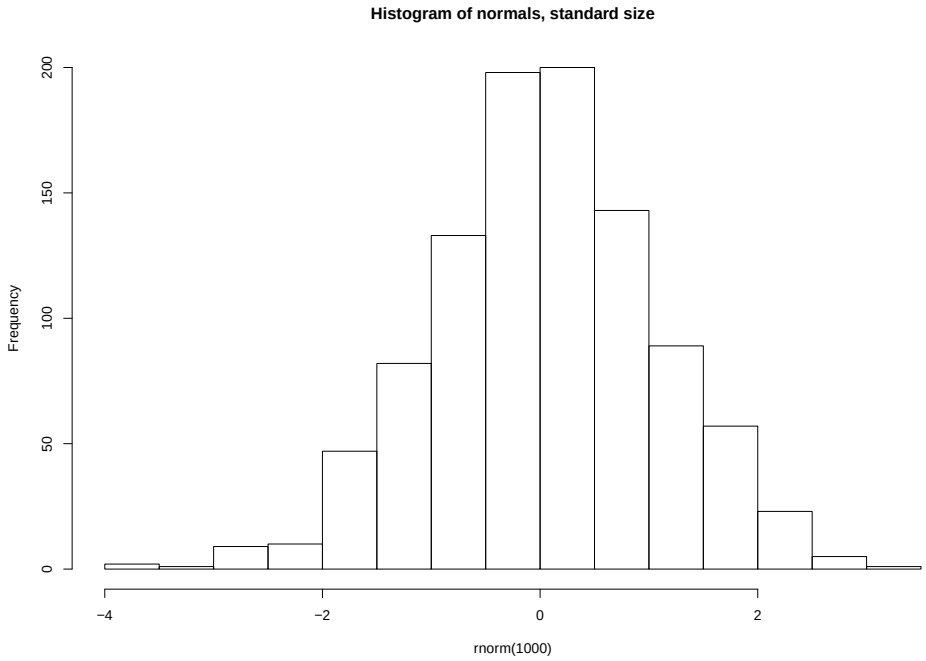
3. `sort()`, `order()` and `rank()`
   - $sort(x) == x[order(x)]$
     - in fact it is defined that way
   - $rank(x) == order(order(x))$
   - $order(order(x))$ is generally faster than $rank(x)$
   - for small vector lengths $x[order(x)]$ can be faster than $sort(x)$
     - but see later

4. Reproducible random numbers for grid computing
   - generally need to be able to rerun a task
   - can generate .Random.seed for each task, keep in table, lookup table when required
   - **or** generate random sequence 'on the fly'
     - don't need to pass R data to each task
     - each task can be 'text only'
     - **but** do need to know how many random numbers are used for each task

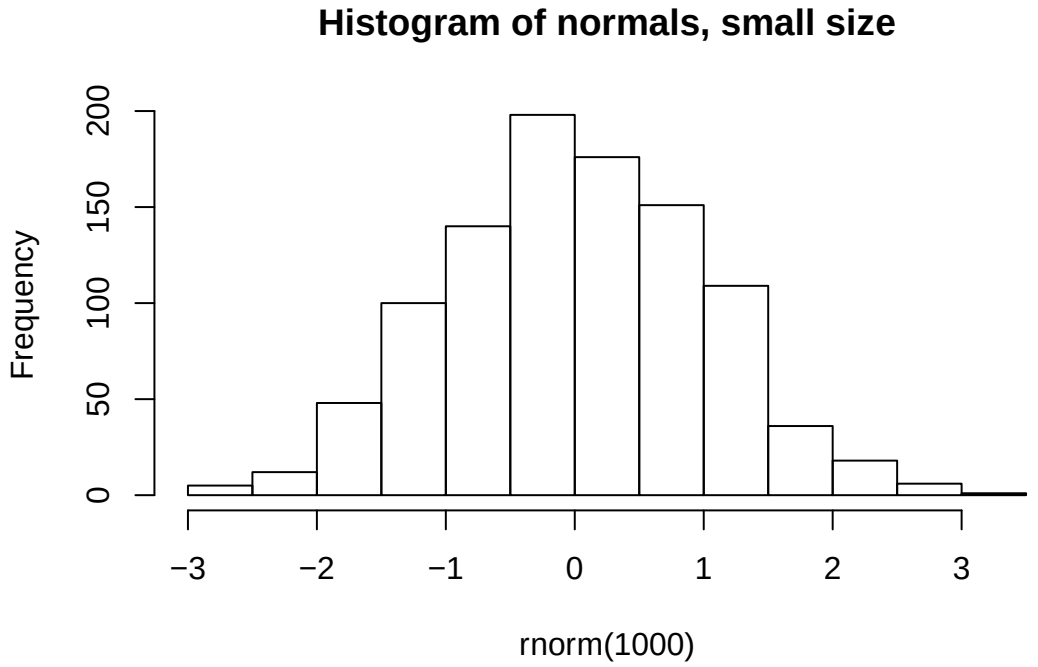
5. Resolution of pdf graphs

- specify width= and height= to suit eventual size
- e.g. small diagram in paper

`postscript()`



`postscript(width=6, height=4)`



6. Local versions of standard functions

- once algorithm and data are known to be 'clean'
- extract just the 'active' part of primary function
- savings are dependent on the format of the data
- e.g. 1: `rank()`

```
> x <- runif(50000)
> system.time(for(i in 1:1000) rank(x))
   user  system elapsed
  22.46    0.05   22.51
> system.time(for(i in 1:1000) .Internal(rank(x, "min")))
   user  system elapsed
  19.52    0.02   19.53
>
```

– e.g. 2: sort()

```
> system.time(for(i in 1:1000) sort(x))
   user  system elapsed
  12.27    0.00   12.28
> system.time(for(i in 1:1000) .Internal(qsort(x, FALSE)))
   user  system elapsed
   7.74    0.00    7.74
> all.equal(sort(x), .Internal(qsort(x, FALSE)))
[1] TRUE
>
```

– e.g. 3: order()

```
> system.time(for(i in 1:1000) order(x))
   user  system elapsed
  18.80    0.00   18.81
> system.time(for(i in 1:1000) .Internal(qsort(x, TRUE))$ix)
   user  system elapsed
   8.21    0.00    8.20
> all.equal(order(x), .Internal(qsort(x, TRUE))$ix)
[1] TRUE
>
```

7. Vectorisation

- user-defined functions using `curve()`
  - `curve()` requires a vectorised expression
  - e.g.

$$G1(x) = \int_{-\infty}^{x} g1(y)\,dy$$

where $g1(x) = a(x)/(1 + x.a(x) - a^2(x))$

and $a(x) = \phi(x)/(1 - \Phi(x))$

ϕ is standard normal density

Φ is standard normal df

  - want `G1()` to be vectorised

```
‘G0‘ <-
function(z) return(integrate(g1, -Inf, z)$value)
‘G1‘ <-
function(z) {
  lz <- length(z)
  oz <- order(z)
  z <- c(-Inf, z[oz])
  result <- rep(NA, lz)
  for (i in 1:lz)
    result[i] <- integrate(g1, z[i], z[i + 1])$value
  return(cumsum(result)[order(oz)])
}
```

– check vectorisation:

```
> print(x <- rnorm(10))
 [1]  0.3135143  0.5262350 -1.1798969 -1.6283480 -0.0983911
 [6]  0.9134180 -2.8988797 -0.0213748 -0.9831606  0.1166303
> for (i in 1:10) res[i] <- G0(x[i]); res
 [1] 2.22925554 3.04868106 0.16027705 0.06039301 1.16851162
 [6] 5.21374042 0.00189237 1.32401479 0.23838252 1.64819056
> print(G1(x))
 [1] 2.22925554 3.04868106 0.16027705 0.06039301 1.16851163
 [6] 5.21374042 0.00189237 1.32401479 0.23838253 1.64819056
>
```

– check accuracy of G1()

```
> g1 <- dnorm # so now G0() is like pnorm()
> res0 <- numeric(length(x))
> system.time(for (i in 1:length(x)) res0[i] <- G0(x[i]))
   user  system elapsed
  14.32    0.00   14.32
> system.time(res1 <- G1(x))
   user  system elapsed
  10.28    0.00   10.29
> max(abs(res0 - pnorm(x)))
[1] 0.0001182122
> max(abs(res1 - pnorm(x)))
[1] 3.166911e-14
```

– check timing:

```
> x <- rnorm(100000)
> system.time(for (i in 1:length(x)) G0(x[i]))
   user  system elapsed
  21.74    0.05   24.49
> system.time(G1(x))
   user  system elapsed
  12.53    0.00   12.53
>
```

– `curve()` is extremely useful when tracking down numerical instability

– e.g.

```
> G1(7)
Error in integrate(g1, z[i], z[i + 1]) :
  maximum number of subdivisions reached
>
```

– remembering ...

$$G1(x) = \int_{-\infty}^{x} g1(y)\, dy$$

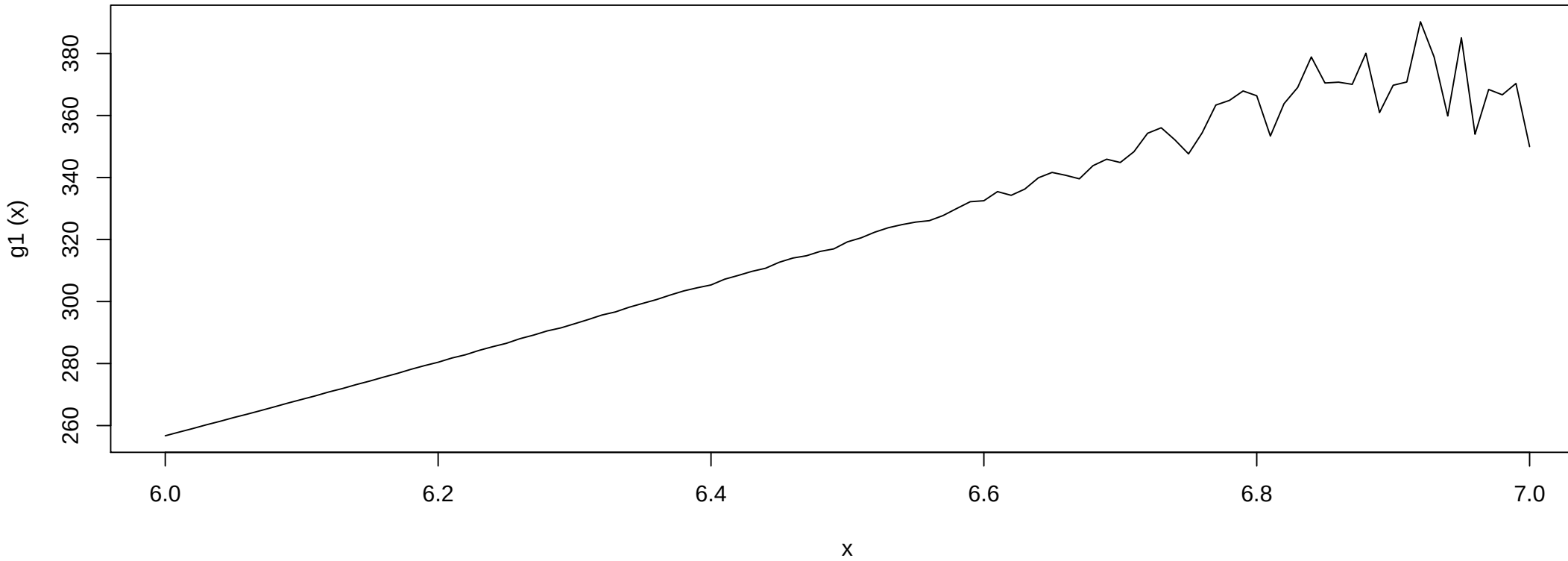
where $g1(x) = a(x)/(1 + x.a(x) - a^2(x))$

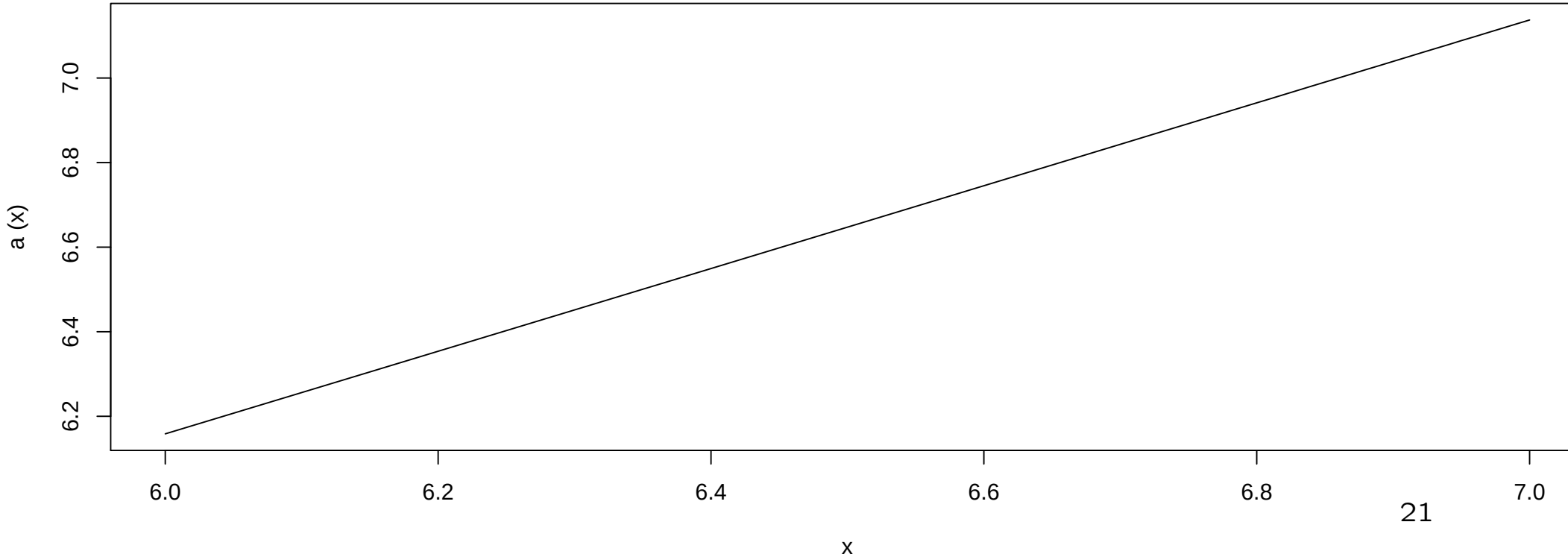
and $a(x) = \phi(x)/(1 - \Phi(x))$

ϕ is standard normal density

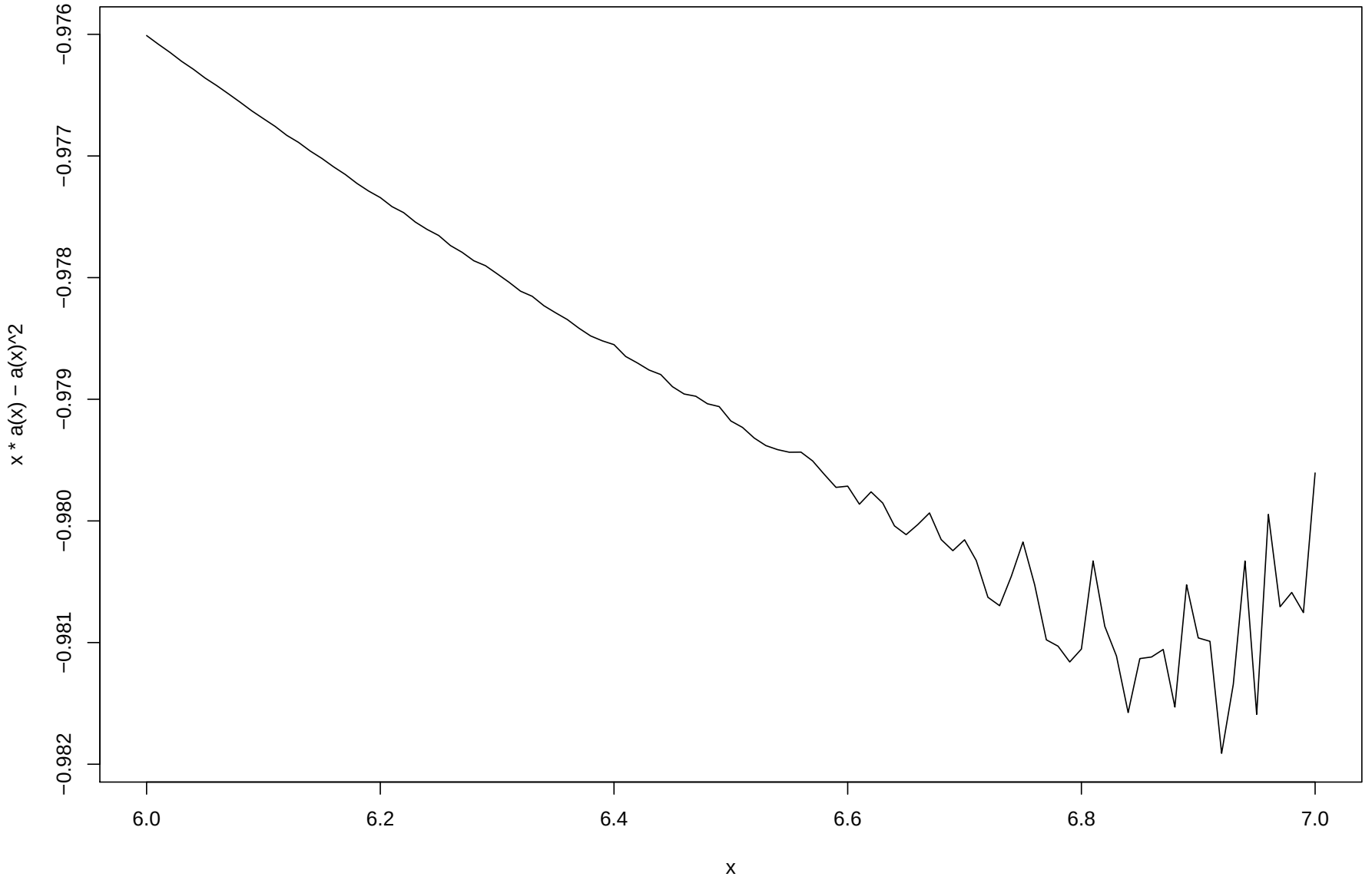
Φ is standard normal df

```
> g1
function(y) {
  ay <- a(y)
  return(ay/(1 + y*ay - ay^2))
}
> a
function(y)
return(dnorm(y)/(1 - pnorm(y)))
>
> curve(g1, 6, 7)        # shows the problem
> curve(a, 6, 7)         # doesn't
>
```

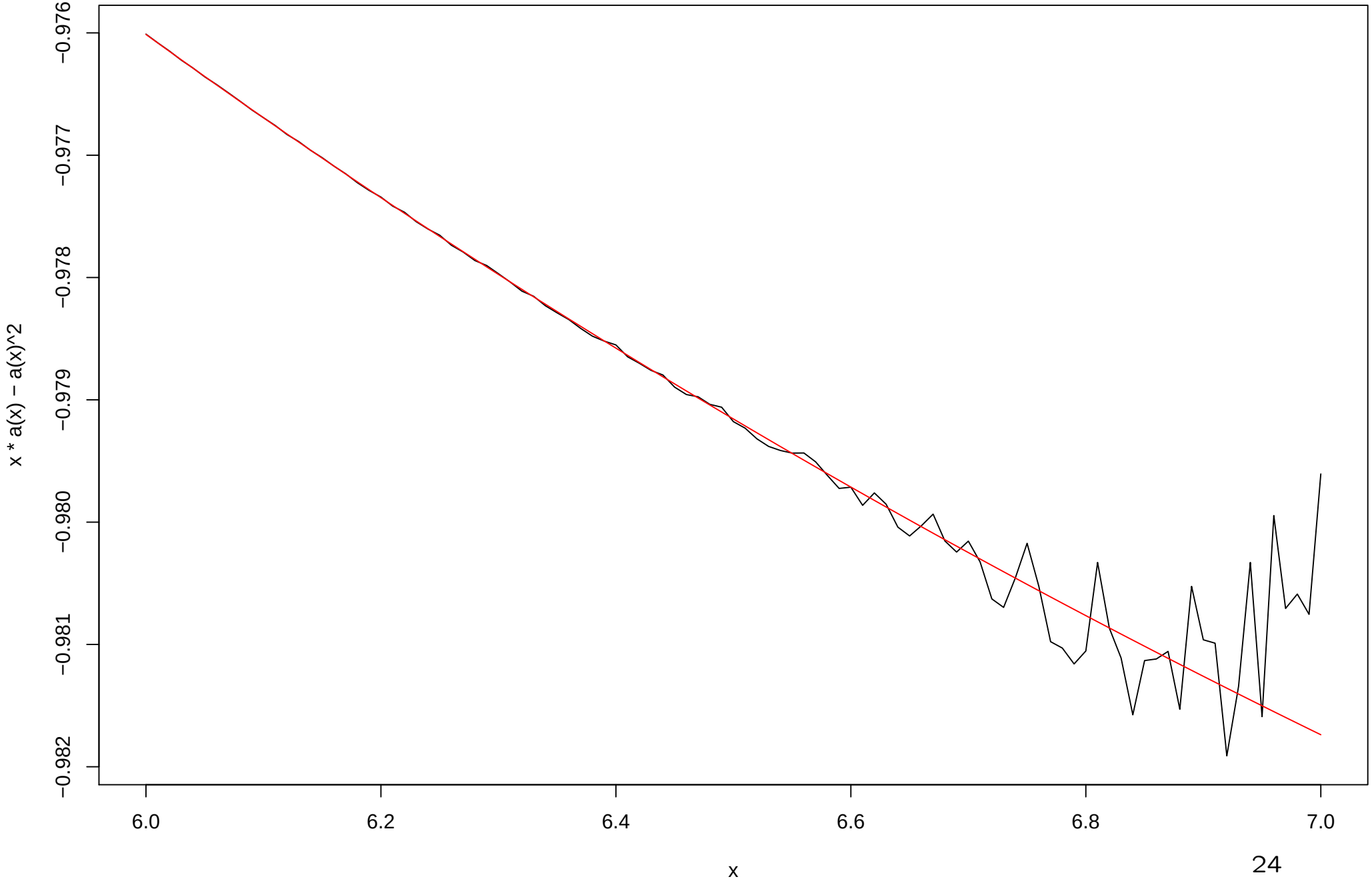

g1 (x)
260
280
300
320
340
360
380
6.0
6.2
6.4
6.6
6.8
7.0
x


a (x)
6.2
6.4
6.6
6.8
7.0
6.0
6.2
6.4
6.6
6.8
7.0
x

```
> curve(x*a(x) - a(x)**2, 6, 7)
```



```
> a
function(y)
return(dnorm(y)/(1 - pnorm(y)))
>
> a1
function(y)
return(dnorm(y)/pnorm(y, lower=FALSE))
>
> curve(x*a(x) - a(x)**2, 6, 7)
> curve(x*a1(x) - a1(x)**2, add=T, col=2)
>
```


x * a(x) − a(x)^2
−0.976
−0.977
−0.978
−0.979
−0.980
−0.981
−0.982
6.0
6.2
6.4
6.6
6.8
7.0
x

- pseudo vectorisation
  - if `val` is a scalar, then `val[1]` is defined
  - use a loop to generate a vector result
  - e.g.

```
‘stepfun2D‘ <-
function(u1, u2, s, t) { # vectorised in t
  res <- numeric(lt <- length(t))
  for (i in 1:lt)
    res[i] <- sum(u1 <= s & u2 <= t[i])
  return(res)
}
```

8. get()
   - useful when using paste to construct an object name
   - can be used as if it was an object of the type retrieved
   - e.g.

```
get("+")(3, 5)  # same as 3 + 5


get("x")[4]     # same as x[4]


thisobj <- get(paste("myobject", myval, sep=""))


for(i in ls())
  cat(i, "\t", object.size(get(i)), "\n")
```

9. Using a matrix to index an array
   - general format is m*n
     - m is the number of elements to be matched
     - n is the number of dimensions of the array
   - can generate the matrix using `matrix()`
   - or use `which(..., arr.ind = TRUE)`
   - e.g. ...

```
> arr <- sample(7:26)
> dim(arr) <-c(2, 5, 2)
> arr
, , 1

     [,1] [,2] [,3] [,4] [,5]
[1,]   20    8   22    7   24
[2,]   14   25   23   26   10

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   13   11   17   19   16
[2,]   12   21   15   18    9

>
```

```
> toolarge <- which(arr > 20, arr.ind = TRUE)
> toolarge
     dim1 dim2 dim3
[1,]    2    2    1
[2,]    1    3    1
[3,]    2    3    1
[4,]    2    4    1
[5,]    1    5    1
[6,]    2    2    2
> arr[toolarge] <- NA
>
```

```
> arr
, , 1

     [,1] [,2] [,3] [,4] [,5]
[1,]   20    8   NA    7   NA
[2,]   14   NA   NA   NA   10

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   13   11   17   19   16
[2,]   12   NA   15   18    9

>
```

10. Matrices, lists and dataframes, which are more efficient?
    - In general, matrices are more efficient
    - but dataframes may be more useful
    - YMMV
    - e.g. creating a matrix of unknown size ...

```
> set.seed(0); x <- numeric()
> system.time({for (i in 1:10000) x <- rbind(x, runif(10))})
   user  system elapsed
   4.48    0.05    4.52
> set.seed(0); y <- numeric()
> system.time({for (i in 1:10000) y <- c(y, runif(10));
+   dim(y) <- c(10, 10000); y <- t(y)})
   user  system elapsed
   1.73    0.16    1.89
> all.equal(x, y)
[1] TRUE
>
```

11. Using and saving .Rhistory

    – in .Rprofile in home directory:

    ```
    .Last <- function() {if(interactive()) savehistory()}
    ```

    – saves history even if not saving image

12. [Windows] `file.choose()`

    – saves having to remember where all the quotes, colons, and backslashes go

    - (or should they be forward slashes?)