Tricks and Traps for Young Players

Ray D Brownrigg

Statistical Computing Manager
School of Mathematics, Statistics and Computer Science
Victoria University of Wellington

Wellington, New Zealand

ray@mcs.vuw.ac.nz

UseR! 2008

Dortmund, August 2008

CONTENTS

CONTENTS (continued)

1. Background

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

2. Introduction

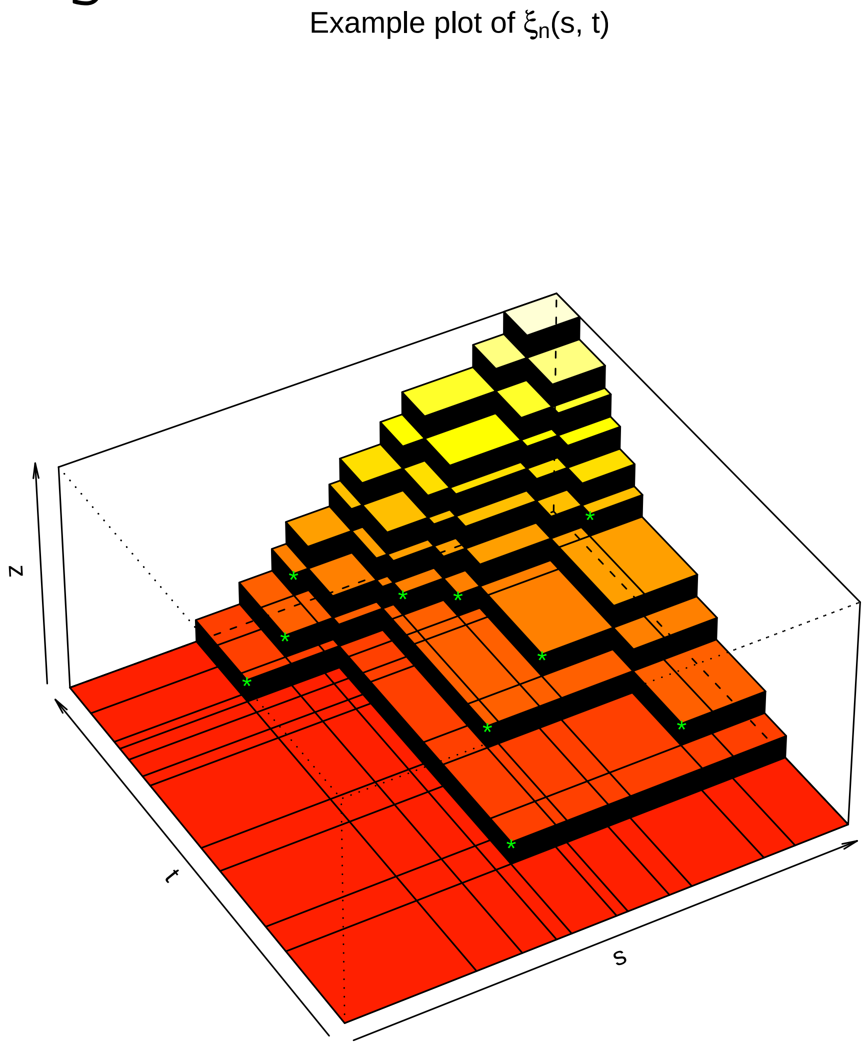
   Calculate the distribution function of the supremum of a normalised two-dimensional independent poisson process. This simulates Brownian Motion, which appears as a limiting process in goodness-of-fit studies.

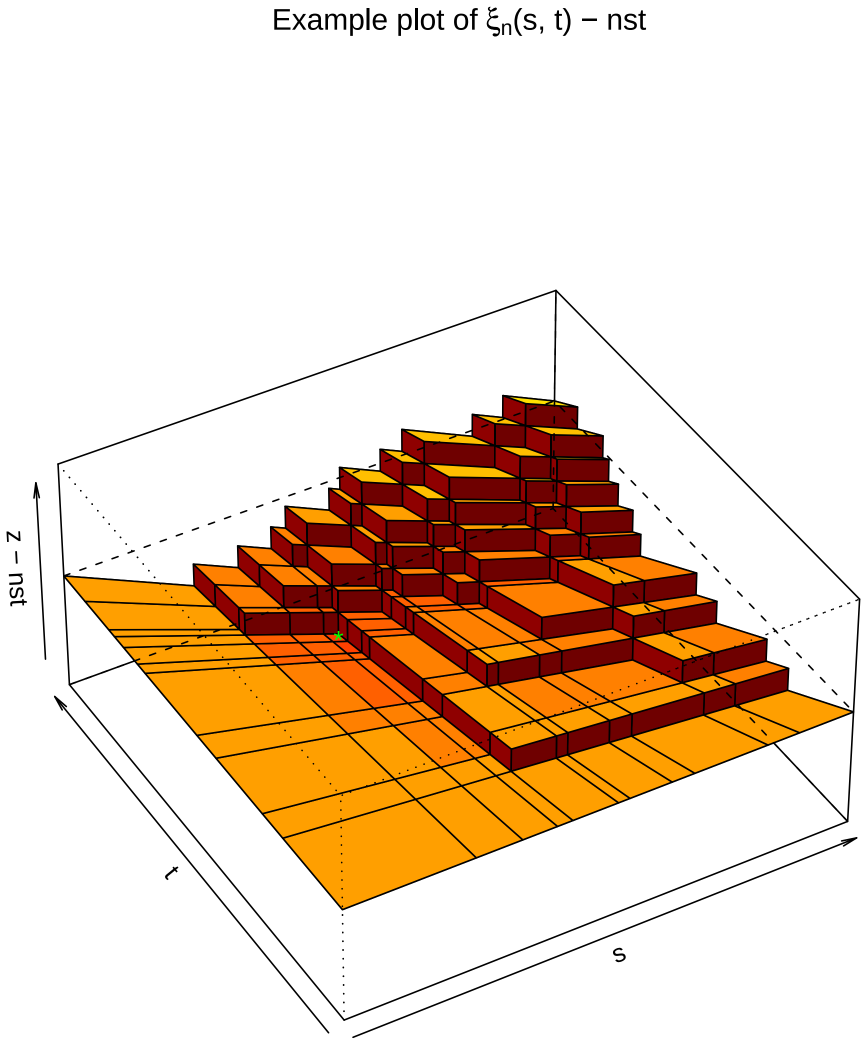
   – throw down N points on unit square

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

   – find supremum

E.g.:





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

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

– Fortran gives >1 order of magnitude improvement (12-40x)

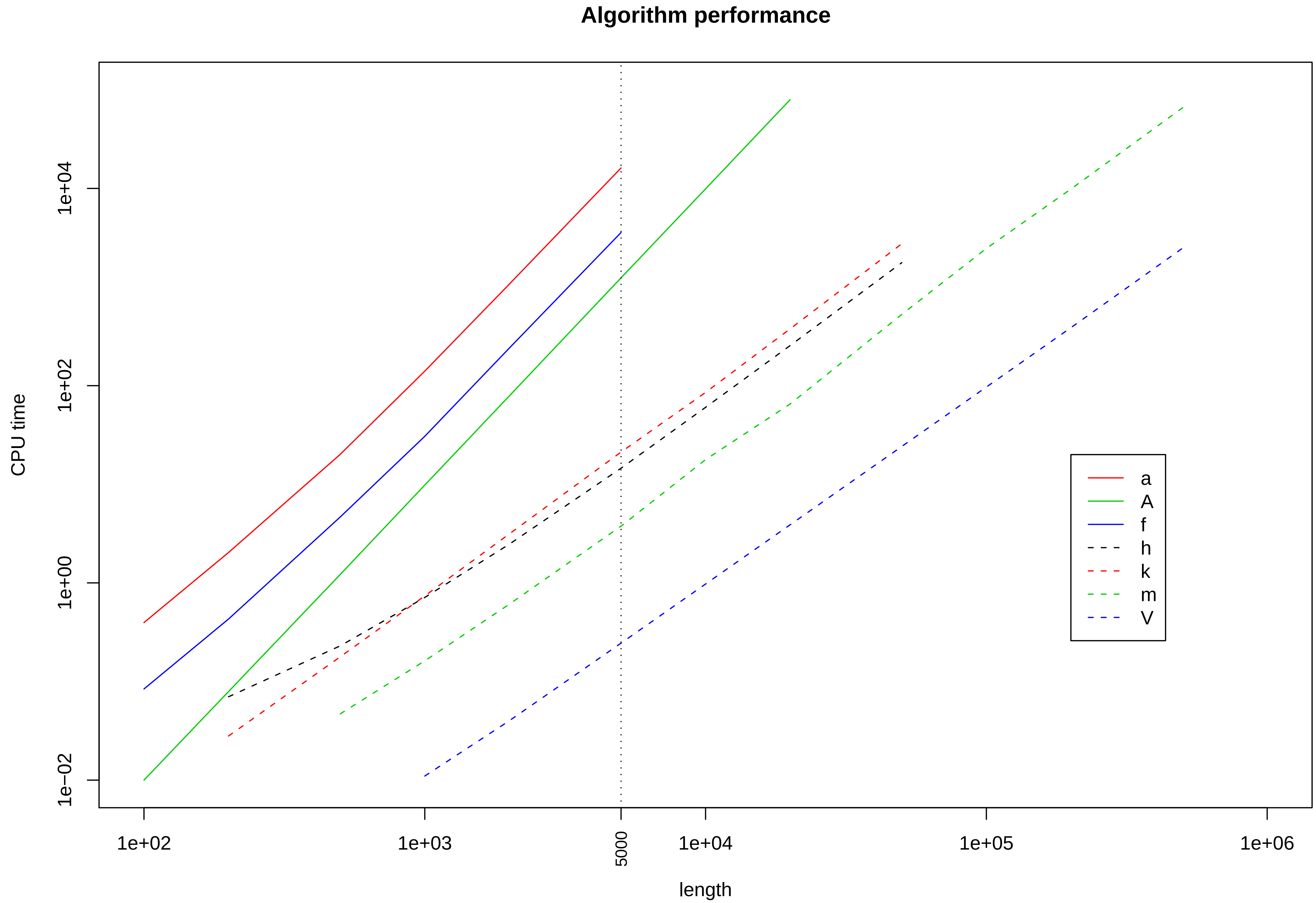
– restructuring to single loop using cumsum() and order() is generally faster than the initial Fortran

– now $O(N^2)$

– further improvements save another factor of 3

– now Fortran saves another 1.5 orders of magnitude (i.e. 30x)

– overall 5 orders of magnitude speed improvement


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

3. sort(), order() and rank()

– $sort(x) == x[order(x)]$

• in fact it is defined that way (for “objects” - with class)

– $rank(x) == order(order(x))$

– $order(order(x))$ 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'

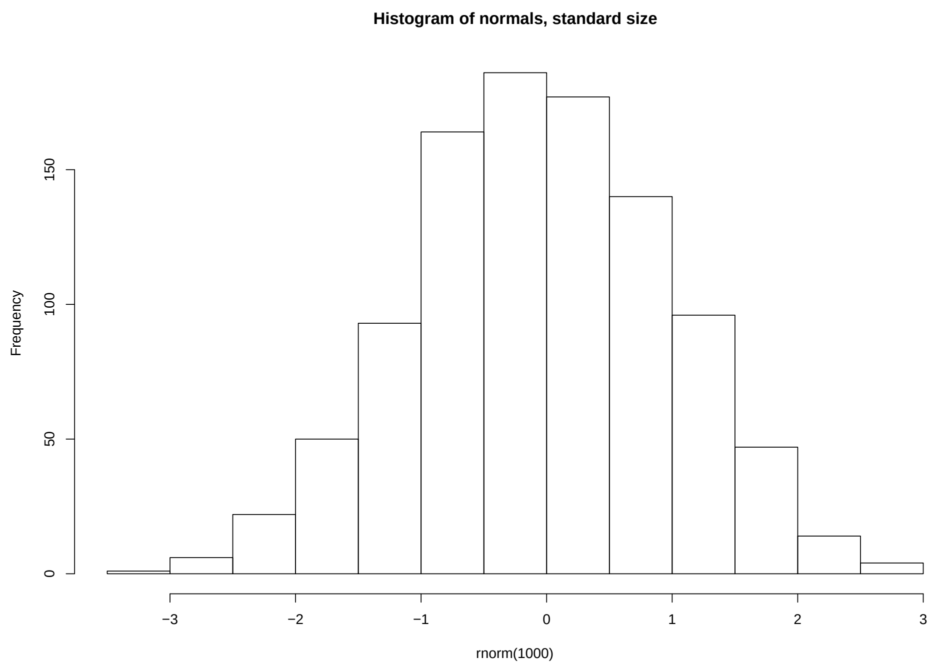
- don't need to pass R data to each task
- each task can be 'text only'
- **but** do need to know how many random numbers are used for each task

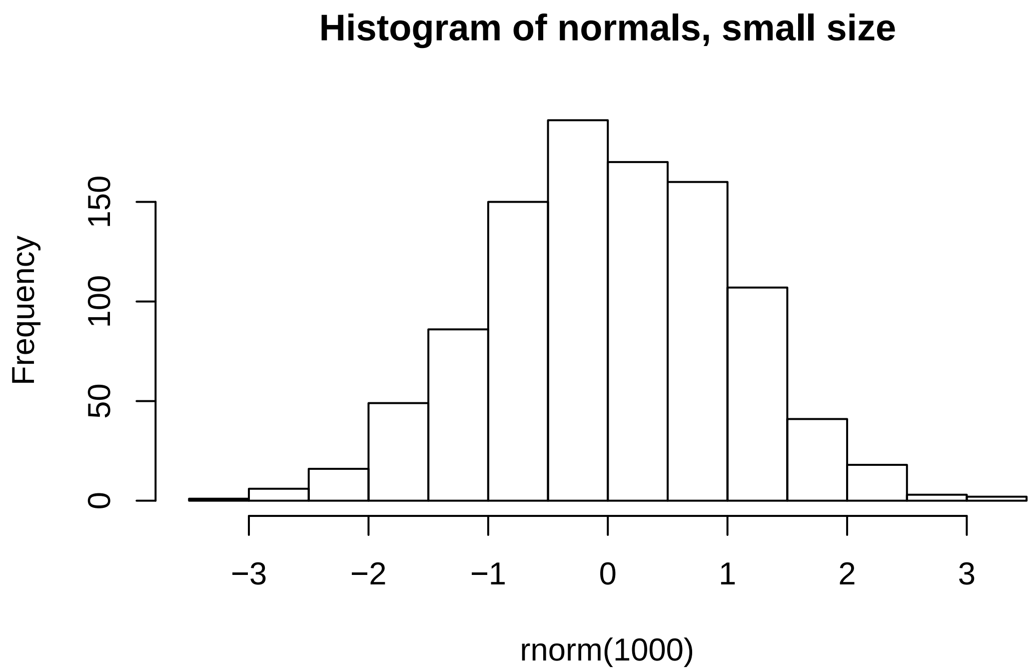
5. Resolution of pdf graphs

– specify width= and height= to suit eventual size

– e.g. small diagram in paper

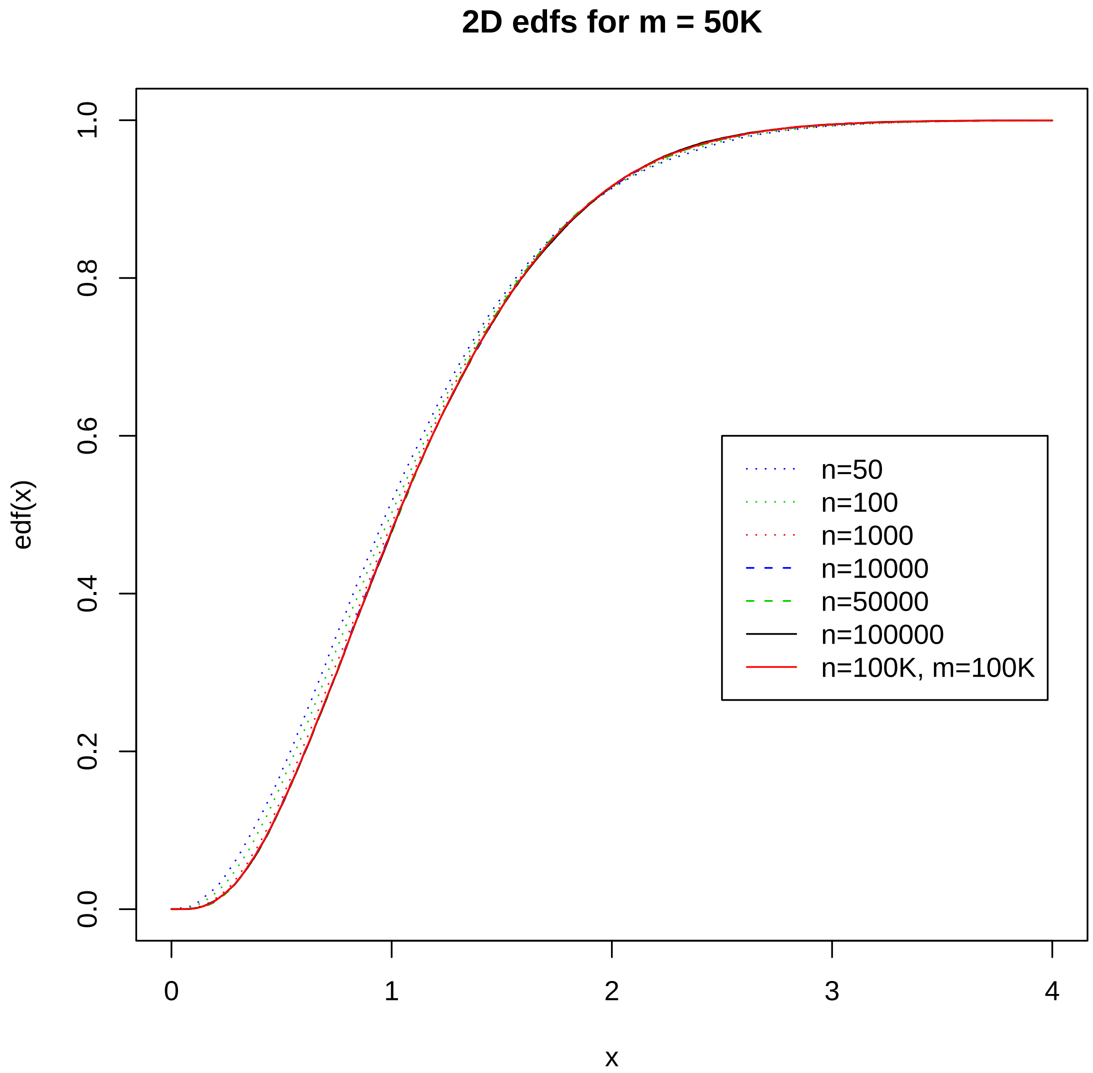
`postscript()` `postscript(width=6, height=4)`



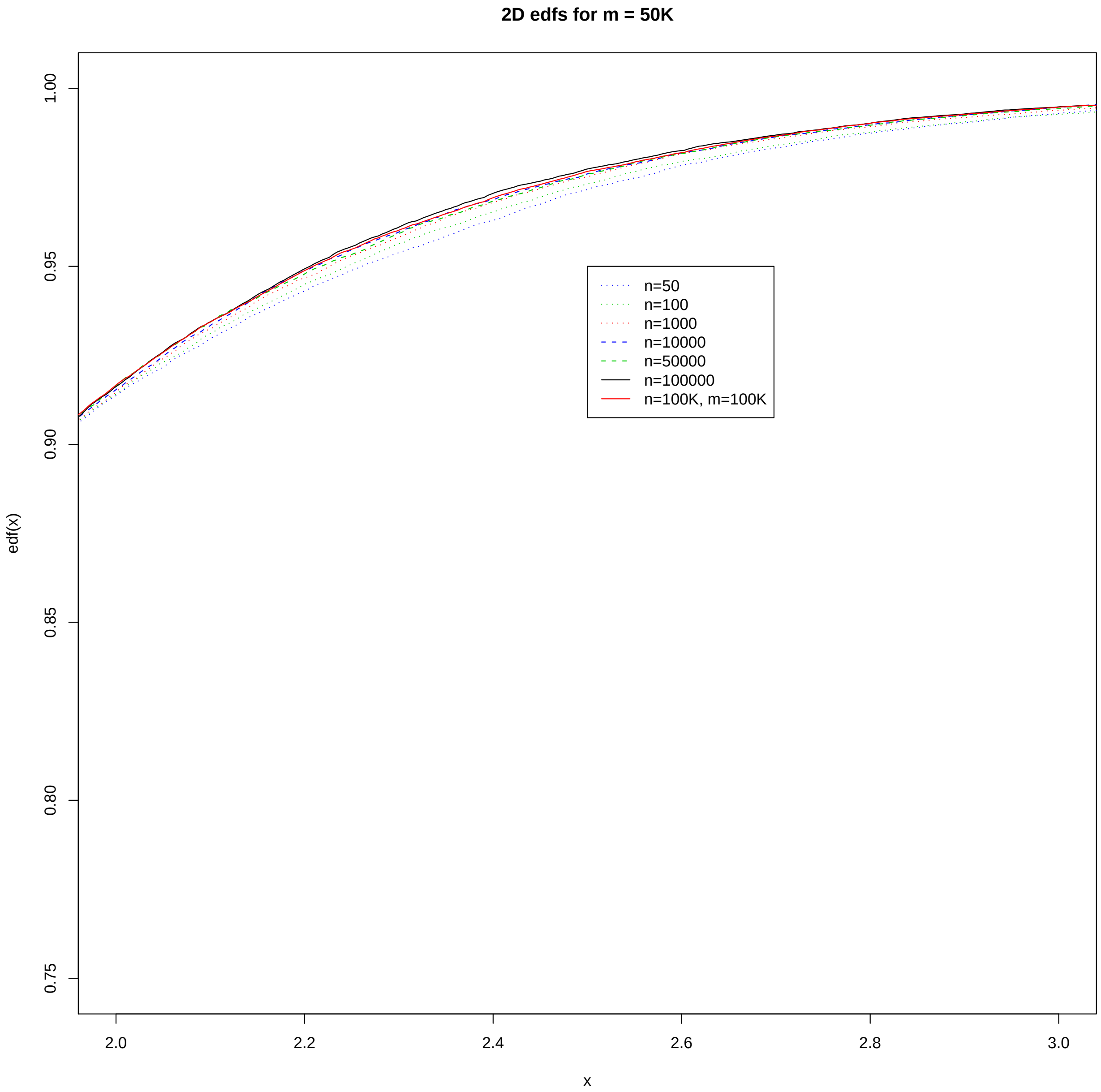


– e.g. fine detail in downloadable file

`pdf()`



pdf(width=12, height=12)



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. rank()

```
> x <- runif(50000)
> system.time(for(i in 1:1000) rank(x))
   user  system elapsed
 22.698   0.550  23.257
> system.time(for(i in 1:1000) .Internal(rank(x, "min")))
   user  system elapsed
 20.356   0.160  20.575
>
```

– e.g. sort()

```
> system.time(for(i in 1:1000) sort(x))
   user  system elapsed
 11.189   0.119  11.349
> system.time(for(i in 1:1000) .Internal(qsort(x, FALSE)))
   user  system elapsed
  5.237   0.070   5.321
> all.equal(sort(x), .Internal(qsort(x, FALSE)))
[1] TRUE
>
```

– e.g. order()

```
> system.time(for(i in 1:1000) order(x))
   user  system elapsed
 18.948   0.010  18.986
> system.time(for(i in 1:1000) .Internal(qsort(x, TRUE))$ix)
   user  system elapsed
  7.105   0.050   7.170
> all.equal(order(x), .Internal(qsort(x, TRUE))$ix)
[1] TRUE
>
```

7. Vectorisation

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

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

    ϕ is standard normal density

    Φ is standard normal df

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

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

```
`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: ...

```
> x <- qnorm(runif(10))
> x
 [1]  1.2629543 -0.6264538 -0.3262334  0.1836433  1.3297993
 [6] -0.8356286  1.2724293  1.5952808  0.4146414  0.3295078
> for (i in 1:10) cat(G1(x[i]), "\t")
8.17856         0.4691605       0.7979545       1.829099
8.883072        0.3174469       8.27546         12.20594
2.591307        2.283419
> print(G1(x))
 [1]  8.1785600  0.4691605  0.7979545  1.8290994  8.8830715
 [6]  0.3174469  8.2754602 12.2059365  2.5913071  2.2834192
>
```

– check timing:

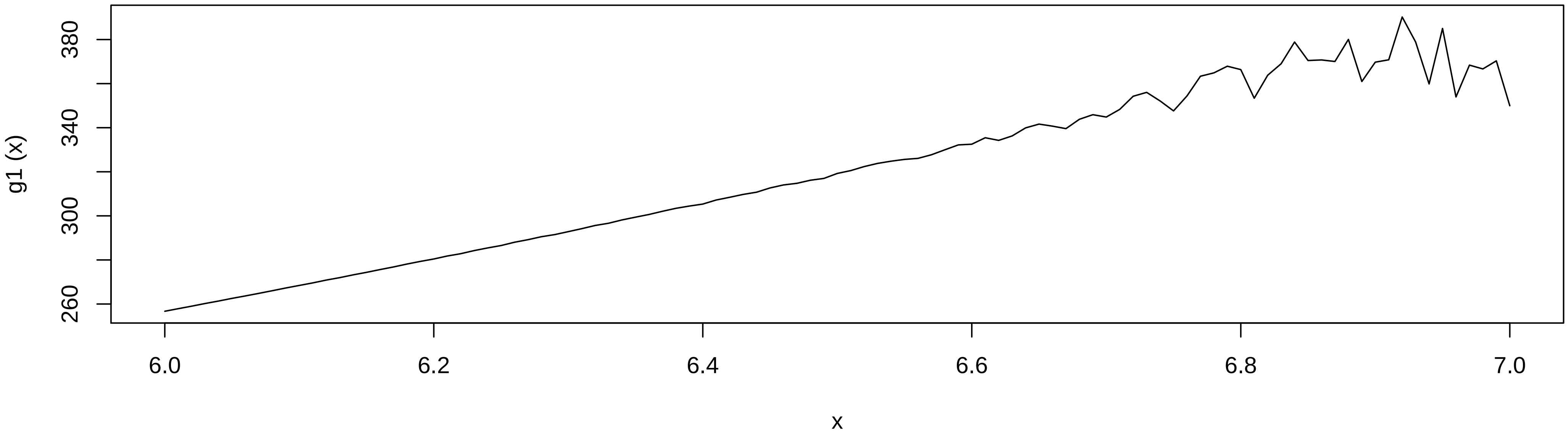
```
> x <- qnorm(runif(100000))
> system.time(for (i in 1:length(x)) G1(x[i]))
   user  system elapsed
 24.251  -0.001  24.270
> system.time(G1(x))
   user  system elapsed
 11.496   0.000  11.501
>
```

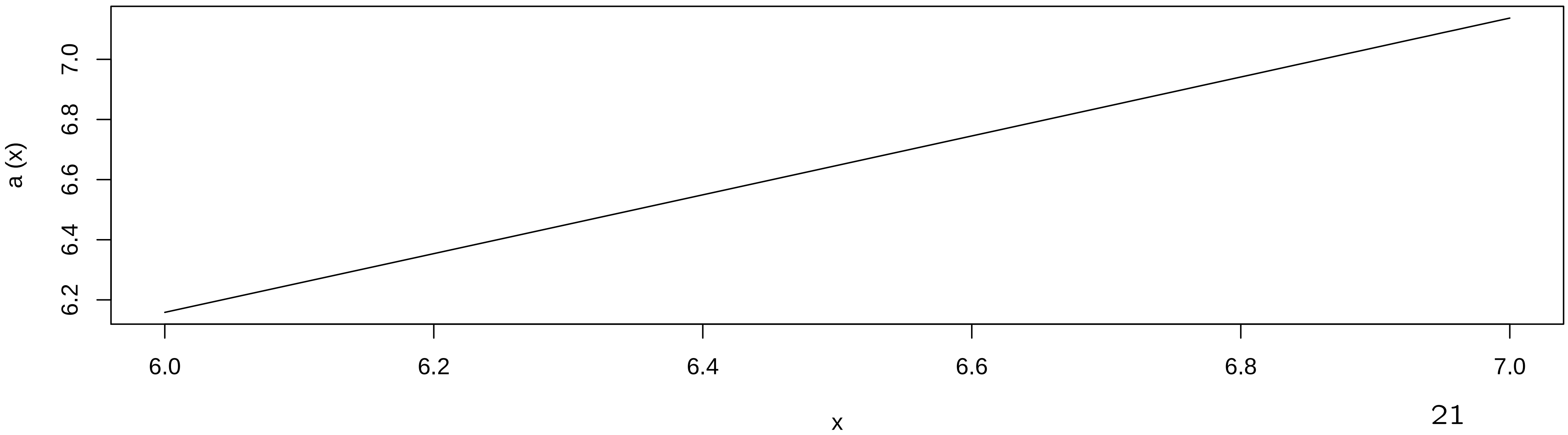
– curve() is extremely useful when tracking down numerical instability

– e.g.

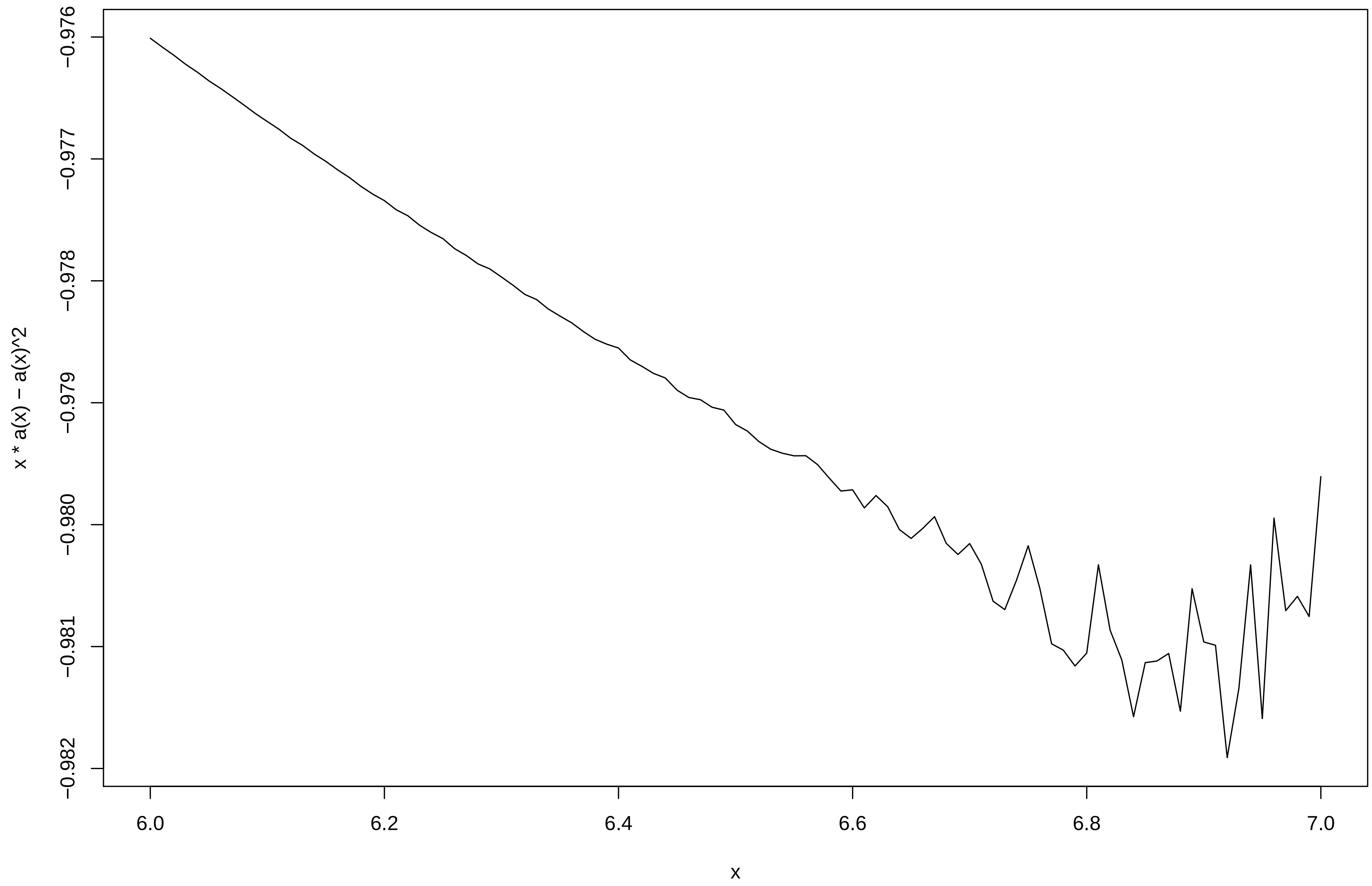
```
> G1(7)
Error in integrate(g1, z[i], z[i + 1]) :
  maximum number of subdivisions reached
>
```

```
> 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)
> curve(a, 6, 7)
>
```

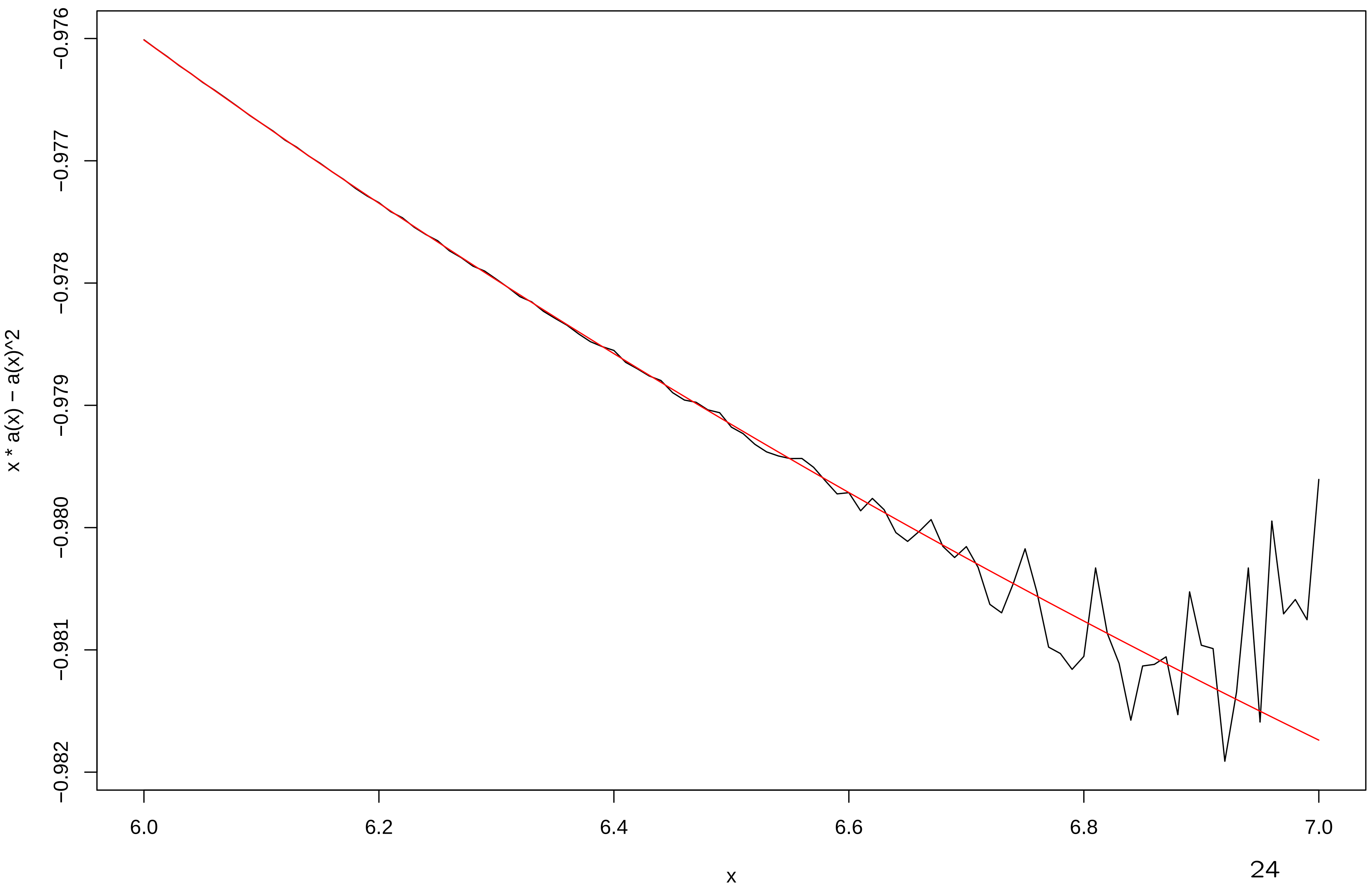

g1 (x)
260
300
340
380
6.0
6.2
6.4
6.6
6.8
7.0
x


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)
>
```



−0.976
−0.977
−0.978
−0.979
−0.980
−0.981
−0.982
x * a(x) − a(x)^2
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)
}
```

- multi-dimensional
  - a function of two parameters may give the correct result when one of the parameters is supplied as a vector, but fail if both are
  - e.g. two-dimensional linear interpolation (achieves vectorisation through the use of recycling)

```
`jointpc0.5` <-
function(s, t) {          # Only one of s, t can be vector.
  DI <- DJ <- 0.001       # granularity of table
  i <- s/DI + 1
  j <- t/DJ + 1
```

```
  di <- i %% 1
  dj <- j %% 1
  i <- trunc(i)
  j <- trunc(j)
  res <- (1 - dj)*
    ((1 - di)*jpt0.5[i, j] + di*jpt0.5[i + 1, j]) + dj*
    ((1 - di)*jpt0.5[i, j + 1] + di*jpt0.5[i + 1, j + 1])
  return(res)
}
```

```
`jointpc0.5m` <-
function(s, t) {          # Either or both can be a vector.
  DI <- DJ <- 0.001       # granularity of table
  i <- s/DI + 1; j <- t/DJ + 1
  di <- i %% 1; dj <- j %% 1
  i <- trunc(i); j <- trunc(j)
  res <- t(
    (1 - dj)*
      t((1 - di)*jpt0.5[i, j] + di*jpt0.5[i + 1, j]) +
    dj*
      t((1 - di)*jpt0.5[i, j + 1] + di*jpt0.5[i + 1, j + 1]))
  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)


get("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(1:24)
> dim(arr) <- 4:2
> arr
, , 1

     [,1] [,2] [,3]
[1,]    5   14    7
[2,]   19   16    4
[3,]    8   24    1
[4,]   18   20    6

, , 2

     [,1] [,2] [,3]
[1,]   22   13   12
[2,]   15    9   21
[3,]   17   10   23
[4,]    2    3   11

>
```

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

     [,1] [,2] [,3]
[1,]    5   14    7
[2,]   19   16    4
[3,]    8   NA    1
[4,]   18   20    6

, , 2

     [,1] [,2] [,3]
[1,]   NA   13   12
[2,]   15    9   NA
[3,]   17   10   NA
[4,]    2    3   11

>
```

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
 16.502   3.180  19.712
> 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
  6.765   3.330  10.097
> 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?)