APTS Statistical Computing: Practical Lab 2 (Thursday)

N.B. *If at all possible, please do 1.(a) (installing an R package) in advance of the practical session, to avoid over-loading the Wi-Fi.*

1. (a) For this exercise, we will use the `FLtools` package from `https://bitbucket.org/finnlindgren/FLtools/`, developed by Finn Lindgren (the previous lecturer for this course). There are a couple of different ways that you can install this. If you have the `devtools` package installed, you can install it with:

```
devtools::install_bitbucket("finnlindgren/FLtools")
```

Alternatively, if you have the `remotes` package installed, you can install it with:

```
remotes::install_bitbucket("finnlindgren/FLtools")
```

If neither of these work, then you probably don't have either `devtools` or `remotes` installed. In that case, first install the `remotes` package, with:

```
install.packages("remotes")
```

Although the `devtools` package is useful, it is a big install, especially on Windows, so it's probably best *not* to install this during the practical session if possible.
However you install the `FLtools` package, you should be able to load it with

```
library(FLtools)
```

Make sure you have this package installed before proceeding to the next step.

(b) Start the `optimisation` shiny app:

```
FLtools::optimisation()
```

This should start a Shiny web application. It will also attempt to start up a tab in your browser connected to the session. If this doesn't work, just connect your browser to the URL of the Shiny app. Make sure the Shiny app is running in a browser window before proceeding to the next step.

(c) For the "Simple (1D)" and "Simple (2D)" functions, familiarise yourself with the "Step", "Converge", and "Reset" buttons.

(d) Choose different optimisation starting points by clicking in the figure.

(e) Explore the different optimisation methods and what they display in the figure for each optimisation step[1][2]. Also observe the diagnostic output box and how the number of function, gradient, and Hessian evaluations differ between the methods.

[1]The simplex/triangle shapes are shown for each "Simplex" method step in blue. The "best" points for each simplex are connected (magenta).

[2]The Newton methods display the true quadratic Taylor approximations (red) as well as the approximations used to find the proposed steps (blue).

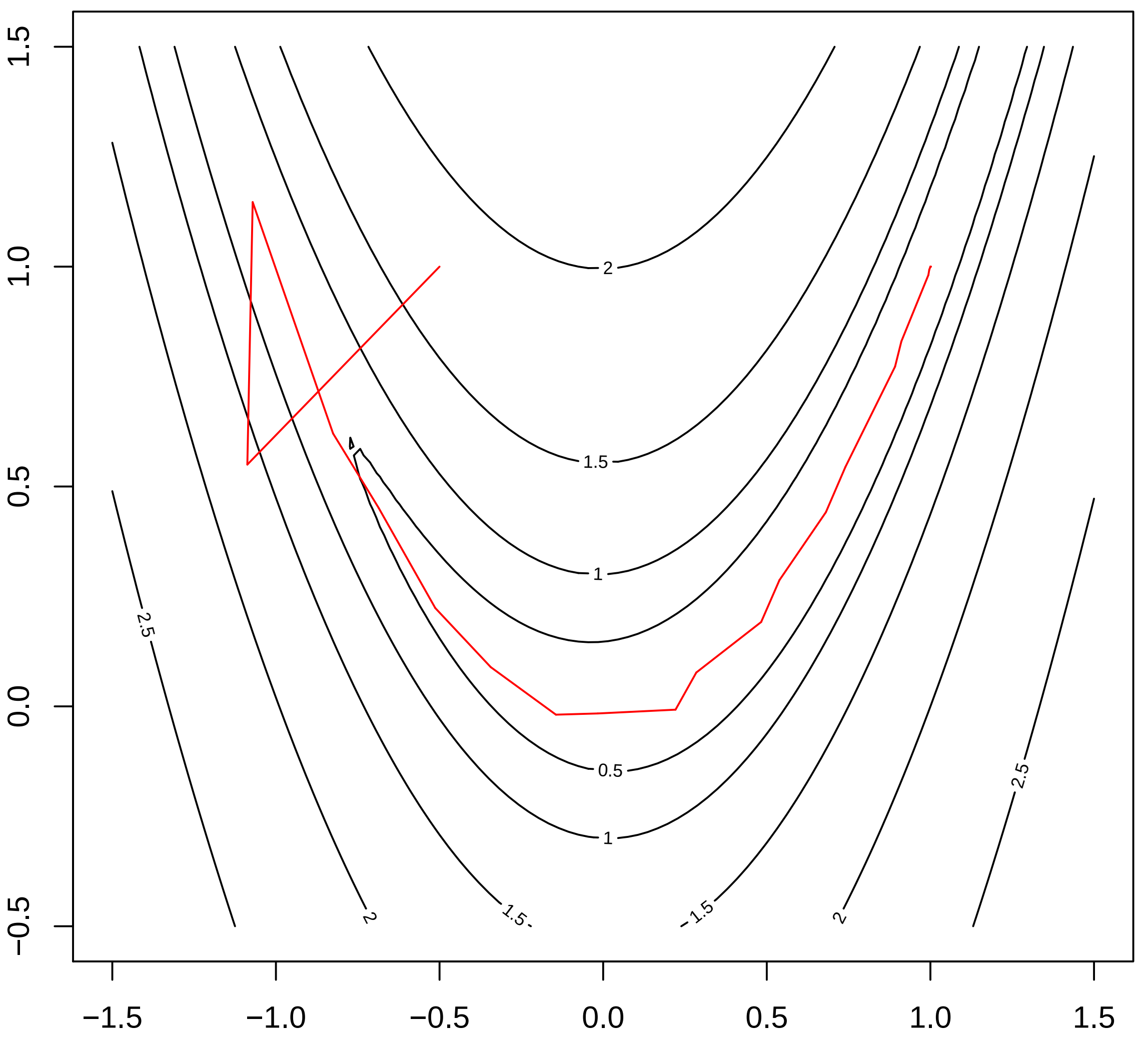
(f) For the "Rosenbrock (2D)" function, observe the differences in convergence behaviour for the four different optimisation methods.

(g) For the "Multimodal" functions, explore how the optimisation methods behave for different starting points.

(h) How far out can the optimisation start for the "Spiral" function? E.g., try the "Newton" method, starting in the top right corner of the figure.

2. Write your own code to optimise Rosenbrock's function by Newton's method. Ensure that you have implemented it correctly by comparing your output (and implementation) with that of the Shiny app from the first exercise. For this question you will want to make use of the preliminary material for the course (and the solutions).

```
## One step of a Newton method
newton_step <- function(x0, f, gf, hf, mh=5, me=0.0001) {
    h0 = hf(x0) # raw hessian
    eig = eigen(h0, symmetric=TRUE)
    e0 = eig$values # raw e-vals
    e1 = abs(e0) # flip negative e-vals
    e1[e1 < me] = me # inflate small e-vals
    ## since we have eigendecomposition, directly compute Newton step
    ei = 1/e1 # eigenvalues of inverse
    delta = -(eig$vectors %*% (ei * (t(eig$vectors) %*% gf(x0))))
    x1 = x0 + delta
    if (f(x1) > f(x0)) {
        for (i in 1:mh) {
            delta = delta / 2 # halve step length
            x1 = x0 + delta
            if (f(x1) < f(x0)) return(x1)
        }
        warning("Newton step didn't descend.")
    }
    x1
}

## Newton iteration
newton <- function(x0, f, gf, hf, mi=100, eps=1e-10, plt=FALSE, ...) {
    x = x0
    for (i in 1:mi) {
        x = newton_step(x, f, gf, hf, ...)
        if (plt) {
            lines(c(x0[1],x[1]), c(x0[2],x[2]), col=2)
            message(paste(i, " "), appendLF=FALSE)
            print(x)
        }
        ## mixed relative/absolute convergence check
        if (sum(abs(x-x0)) < eps*sum(abs(x0)) + eps) return(x)
        x0=x
    }
    warning("Max iterations exceeded before convergence.")
    x
}

## Application to the Rosenbrock function

## Define function
rb <- function(x,z) {
  100*(z-x^2)^2 + (1-x)^2
}
## Clearly has min value of 0 at (1,1)

## Plot contours of the function (logged)
n <- 100
x <- seq(-1.5,1.5,length=n)
z <- seq(-.5,1.5,length=n)
f <- outer(x,z,rb)
contour(x,z,matrix(log10(f),n,n),levels=(1:10/2))

## Gradient
rb.grad <- function(x,z) {
  g <- rep(NA,2)
  g[1] <- 400*(x^3-z*x) + 2*(x-1)
  g[2] <- 200*(z-x^2)
  g
}

## Hessian
rb.hess <- function(x,z) {
  H <- matrix(NA,2,2)
  H[1,1] <- 1200*x^2 - 400*z + 2
  H[2,1] <- H[1,2] <- -400*x
  H[2,2] <- 200
  H
}

## Versions with a single vector argument:
rbv <- function(x) rb(x[1],x[2])
rbg <- function(x) rb.grad(x[1],x[2])
rbh <- function(x) rb.hess(x[1],x[2])

xx <- newton(c(-.5,1), rbv, rbg, rbh, plt=TRUE)

## 1

##            [,1]
## [1,] -1.0873529
## [2,]  0.5498683
## 2

##            [,1]
## [1,] -1.070981
## [2,]  1.146732

## 3

##             [,1]
## [1,] -0.8252801
## [2,]  0.6204839

## 4

##             [,1]
## [1,] -0.6861652
## [2,]  0.4514698

## 5

##             [,1]
## [1,] -0.5130686
## [2,]  0.2236005

## 6

##              [,1]
## [1,] -0.34358995
## [2,]  0.08933102

## 7

##              [,1]
## [1,] -0.14438037
## [2,] -0.01883876

## 8

##              [,1]
## [1,] -0.01632909
## [2,] -0.01613049

## 9

##               [,1]
## [1,]  0.221162804
## [2,] -0.007489412

## 10
##             [,1]
## [1,] 0.28458355
## [2,] 0.07696561

## 11

##           [,1]
## [1,] 0.4828215
## [2,] 0.1918072

## 12

##           [,1]
## [1,] 0.5386610
## [2,] 0.2870376

## 13

##           [,1]
## [1,] 0.6807331
## [2,] 0.4416540

## 14

##           [,1]
## [1,] 0.7404237
## [2,] 0.5446642

## 15

##           [,1]
## [1,] 0.8919928
## [2,] 0.7726780

## 16

##           [,1]
## [1,] 0.9112983
## [2,] 0.8300919

## 17

##           [,1]
## [1,] 0.9938468
## [2,] 0.9809172

## 18
##              [,1]
## [1,] 0.9964510
## [2,] 0.9929077

## 19

##              [,1]
## [1,] 0.9999952
## [2,] 0.9999778

## 20

##      [,1]
## [1,]    1
## [2,]    1

## 21

##      [,1]
## [1,]    1
## [2,]    1

## 22
```



```
##      [,1]
## [1,]    1
## [2,]    1

print(xx)

##      [,1]
## [1,]    1
## [2,]    1

print(rbv(xx))
```

```
## [1] 0
```

3. Consider the linear mixed model for a response vector $\mathbf{y}$:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \mathbf{Z}\mathbf{b} + \boldsymbol{\epsilon}, \quad \mathbf{b} \sim N(\mathbf{0}, \mathbf{I}\sigma_b^2), \quad \boldsymbol{\epsilon} \sim N(\mathbf{0}, \mathbf{I}\sigma^2)$$

$\mathbf{X}$ and $\mathbf{Z}$ are (fixed) model matrices, $\boldsymbol{\beta}$, σ_b^2 and σ^2 are parameters, and $\mathbf{b}$ and $\boldsymbol{\epsilon}$ are independent.

(a) First simulate some data from a model of this sort, taking care to relate the code back to the mathematical statement of the model...

```r
set.seed(10)
n <- 100;n.b <- 10;n.beta <- 5
## X and Z are fixed in the model, not random. Random numbers
## used only to generate arbitrary examples, here....
X <- cbind(1,matrix(runif(n*n.beta-n),n,n.beta-1))
Z <- matrix(runif(n*n.b),n,n.b)
beta <- rep(1,n.beta)
b <- rnorm(n.b)
y <- X%*%beta + Z%*%b + rnorm(n)
```

You'll use the data, $\mathbf{y}$, simulated here, along with the corresponding `X` and `Z`, to experiment with *fitting* linear mixed models (so from now on pretend that you don't know what values $\boldsymbol{\beta}$, σ_b and σ had).

(b) With pencil and paper, find the (marginal) expectation, $\boldsymbol{\mu}$, and covariance matrix, $\mathbf{V}$, of $\mathbf{y}$. State the (marginal) distribution of $\mathbf{y}$. Here we marginalise over the random effects, $\mathbf{b}$, and the errors, but still condition on everything else.

```r
## E(y) = X \beta
## V(y) = I \sigma^2 + ZZ'\sigma^2_b
## N( E(y) , V(y) )
```

(c) The following R function evaluates the log likelihood of $\boldsymbol{\theta}^\mathsf{T} = (\boldsymbol{\beta}^\mathsf{T}, \sigma_b^2, \sigma^2)$ given data $\mathbf{y}$. Note that $\boldsymbol{\theta}$ is the first argument of the function.

```r
logLik <- function(theta,y,X,Z) {
## somewhat plodding linear mixed model log
## likelihood with theta partitioned
## [beta,sig2.b,sig2]
   n <- length(y)
   beta <- theta[1:ncol(X)]
   theta <- theta[-(1:ncol(X))]
   V <- diag(n)*theta[2] + Z %*% t(Z)*theta[1]
   R <- chol(V)
   z <- forwardsolve(t(R), y-X %*% beta)
   ll <- -n*log(2*pi)/2 - sum(log(diag(R))) - sum(z*z)/2
   ll
}
```

To maximise the log likelihood of the model using unconstrained methods, it is better to use a parameterization that guarantees positive variances.[3] Modify the function to accept a parameter vector $\boldsymbol{\theta}^{\mathsf{T}} = (\boldsymbol{\beta}^{\mathsf{T}}, \rho_b, \rho)$ where $\rho = \log(\sigma)$ and $\rho_b = \log(\sigma_b)$.

```
logLik <- function(theta,y,X,Z) {
## somewhat plodding linear mixed model log
## likelihood with theta partitioned
## [beta,log(sig.b),log(sig)]
   n <- length(y)
   beta <- theta[1:ncol(X)]
   theta <- theta[-(1:ncol(X))]
   V <- diag(n)*exp(theta[2]*2) + Z %*% t(Z)*exp(theta[1]*2)
   R <- chol(V)
   z <- forwardsolve(t(R), y-X %*% beta)
   ll <- -n*log(2*pi)/2 - sum(log(diag(R))) - sum(z*z)/2
   ll
}
```

(d) Use `optim` to maximise your likelihood (note that `optim` *minimizes* by default; see the documentation for how to do maximisation, and for how to choose optimisation method).

```
theta.start <- c(beta,0,0)
fit <- optim(theta.start, logLik, method="BFGS", y=y, X=X, Z=Z,
             control=list(fnscale=-1))
logLik(fit$par,y,X,Z)

## [1] -143.2712

fit$par

## [1] -0.78525229  1.26234051  1.37798218  1.21289665  1.66304133 -0.47014505
## [7] -0.08006465
```

(e) In fact, using general purpose optimisation methods to find the optimising $\boldsymbol{\beta}$ is a bit wasteful. Given the variance parameters, closed form expressions for the $\boldsymbol{\beta}$ maximising the likelihood are available, and might as well be used. Then it is only necessary to use general methods for the variance parameters. The likelihood considered only as a function of the variance parameters, with the corresponding MLEs of $\boldsymbol{\beta}$ 'plugged in' is termed a 'profile likelihood'. Show that, given the variance parameters, the log-likelihood is maximised by the $\boldsymbol{\beta}$ minimising

$$(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^{\mathsf{T}}\mathbf{V}^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) = \|\mathbf{R}^{-\mathsf{T}}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})\|^2$$

where $\mathbf{R}^{\mathsf{T}}\mathbf{R} = \mathbf{V}$. ($\|\mathbf{x}\|^2 = \mathbf{x}^{\mathsf{T}}\mathbf{x}$ here.) Hence, produce a 'profile log likelihood' function equivalent to your previous log likelihood function. Your function should accept a vector of variance parameters as its first argument, and should return the corresponding profile log likelihood value. You might want to return the corresponding $\boldsymbol{\beta}$ values as an attribute of the return value, e.g.

[3]Such reparameterisation can often have the added benefit of leading posterior distributions closer to Gaussian, enabling accurate and precise Bayesian approximations not relying on Monte Carlo simulations.

```
  .
  .
  attr(ll,"beta") <- beta
  ll
}
```

```
logLikp <- function(theta,y,X,Z) {
## somewhat plodding linear mixed model log
## *profile* likelihood with theta partitioned
## [log(sig.b),log(sig)]
   n <- length(y)
   V <- diag(n)*exp(theta[2]*2) + Z%*%t(Z)*exp(theta[1]*2)
   R <- chol(V)
   yt <- forwardsolve(t(R),y)
   Xt <- forwardsolve(t(R),X)
   beta <- lm(yt~Xt-1)$coef
   z <- forwardsolve(t(R),y-X%*%beta)
   ll <- -n*log(2*pi)/2 - sum(log(diag(R))) - sum(z*z)/2
   attr(ll,"beta") <- beta
   ll
}
```

(f) Use `optim` to maximise your profiled log likelihood function (or copy the code from the online solutions!), and confirm that you get near identical parameter estimates to those from part (d).

```
theta.start <- c(0,0)
fitp <- optim(theta.start, logLikp, method="BFGS", y=y, X=X, Z=Z,
              control=list(fnscale=-1))
logLikp(fitp$par,y,X,Z)

## [1] -143.2712
## attr(,"beta")
##        Xt1        Xt2        Xt3        Xt4        Xt5
## -0.7852214  1.2623269  1.3779723  1.2128958  1.6630199

fitp$par

## [1] -0.47009527 -0.08005542
```

4. Note: `base::chol()` always returns a matrix in dense storage format, so use `Matrix::chol()` instead, to obtain sparse storage output for sparse storage input.

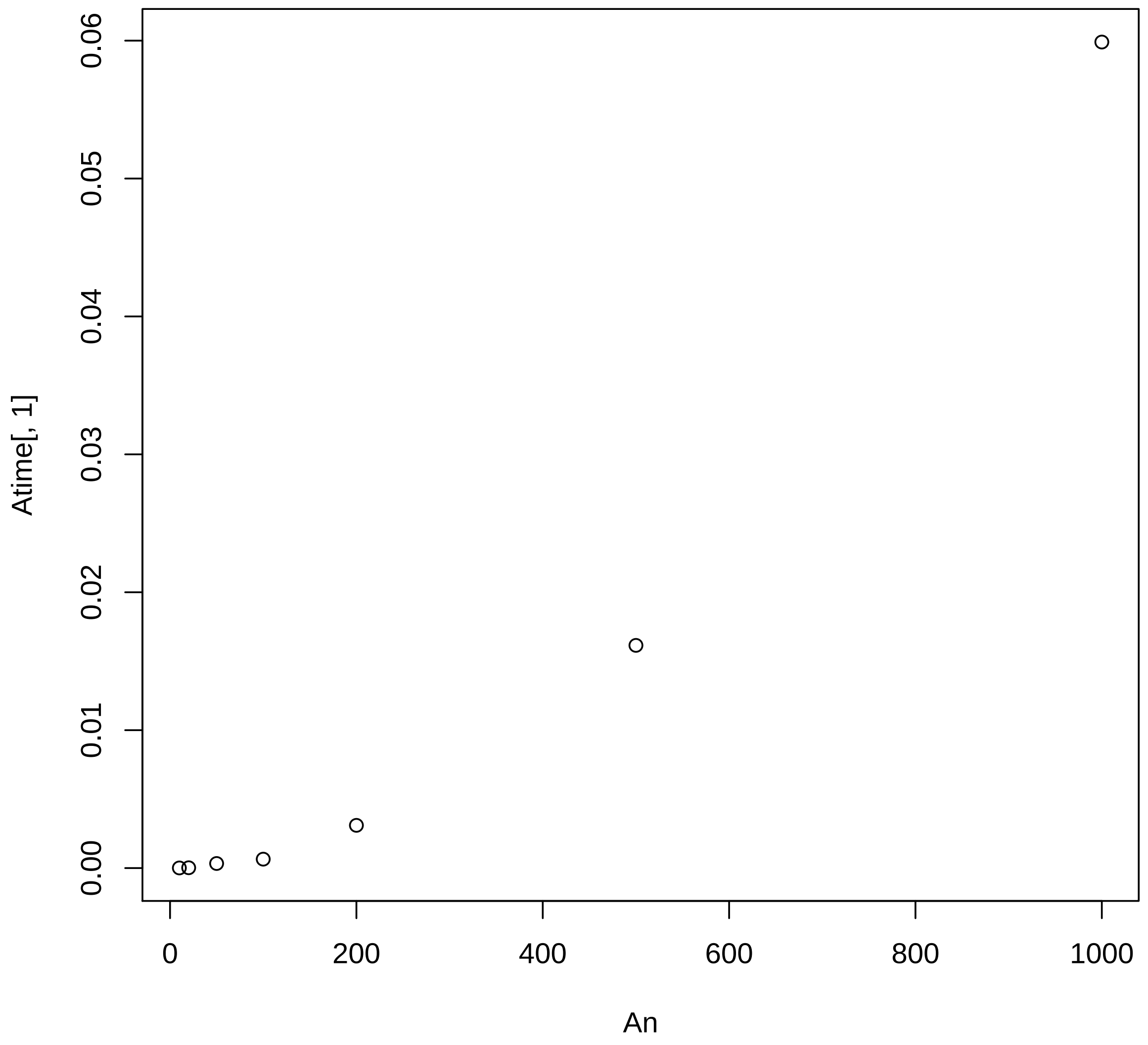
   (a) Run the following code that measures the time it takes to compute dense Cholesky factorisations for matrices of varying size (from 10 to 1000):

```
An <- c(10, 20, 50, 100, 200, 500, 1000)
Atime <- c()
for (n in An) {
  ## Use several repeated runs for small matrices:
  loop.max <- max(1, 10000/n)
```

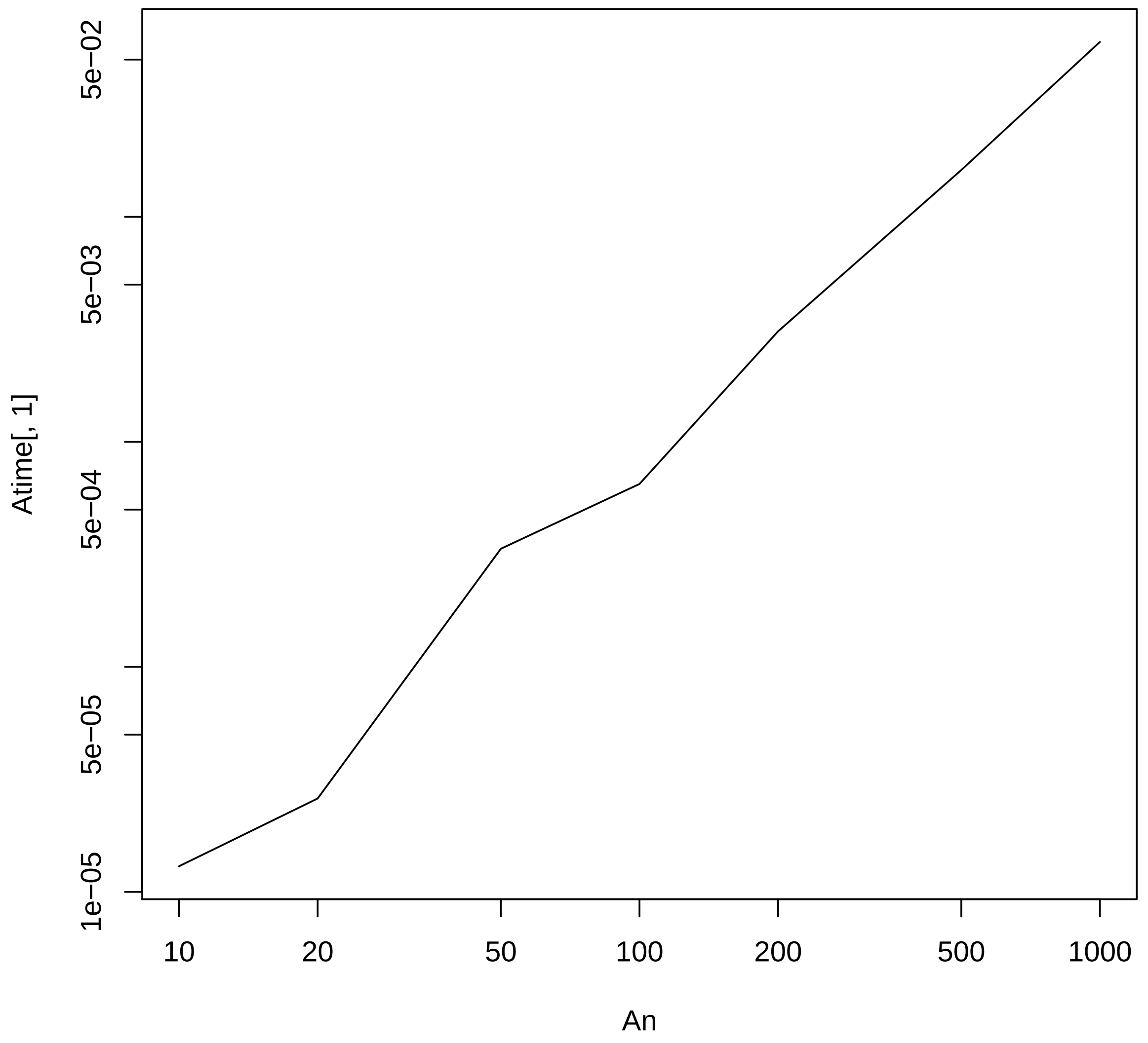
```
    ## Construct a random symmetric positive definite matrix:
    A <- matrix(rnorm(n^2), n, n); A <- t(A) %*% A
    ## Compute and time the Cholesky calculations.
    ## Use B <- A*1 to make sure R doesn't use any hidden
    ## precomputations
    Atime <- rbind(Atime,
                   system.time({
                     for (loop in 1:loop.max) {
                       B <- A*1
                       Matrix::chol(B)
                     }}) / loop.max)
}
Atime
```

```
##       user.self sys.self  elapsed user.child sys.child
## [1,]   0.000013  0.00000 0.000013          0         0
## [2,]   0.000026  0.00000 0.000024          0         0
## [3,]   0.000335  0.00041 0.000100          0         0
## [4,]   0.000650  0.00045 0.000130          0         0
## [5,]   0.003100  0.00204 0.000660          0         0
## [6,]   0.016150  0.00735 0.003000          0         0
## [7,]   0.059900  0.03690 0.012600          0         0
```

```
plot(An, Atime[,1], log="")
```



```
plot(An, Atime[,1], type="l", log="xy")
```



(b) Adapt the code to measure the time it takes to compute dense Cholesky factorisations of covariance matrices of an AR(1) process, for size $n = 10$ to 1000. Note that you can create such a matrix (for standard deviation `sd` and auto-regressive parameter `a` $\in [0, 1)$) with something like:

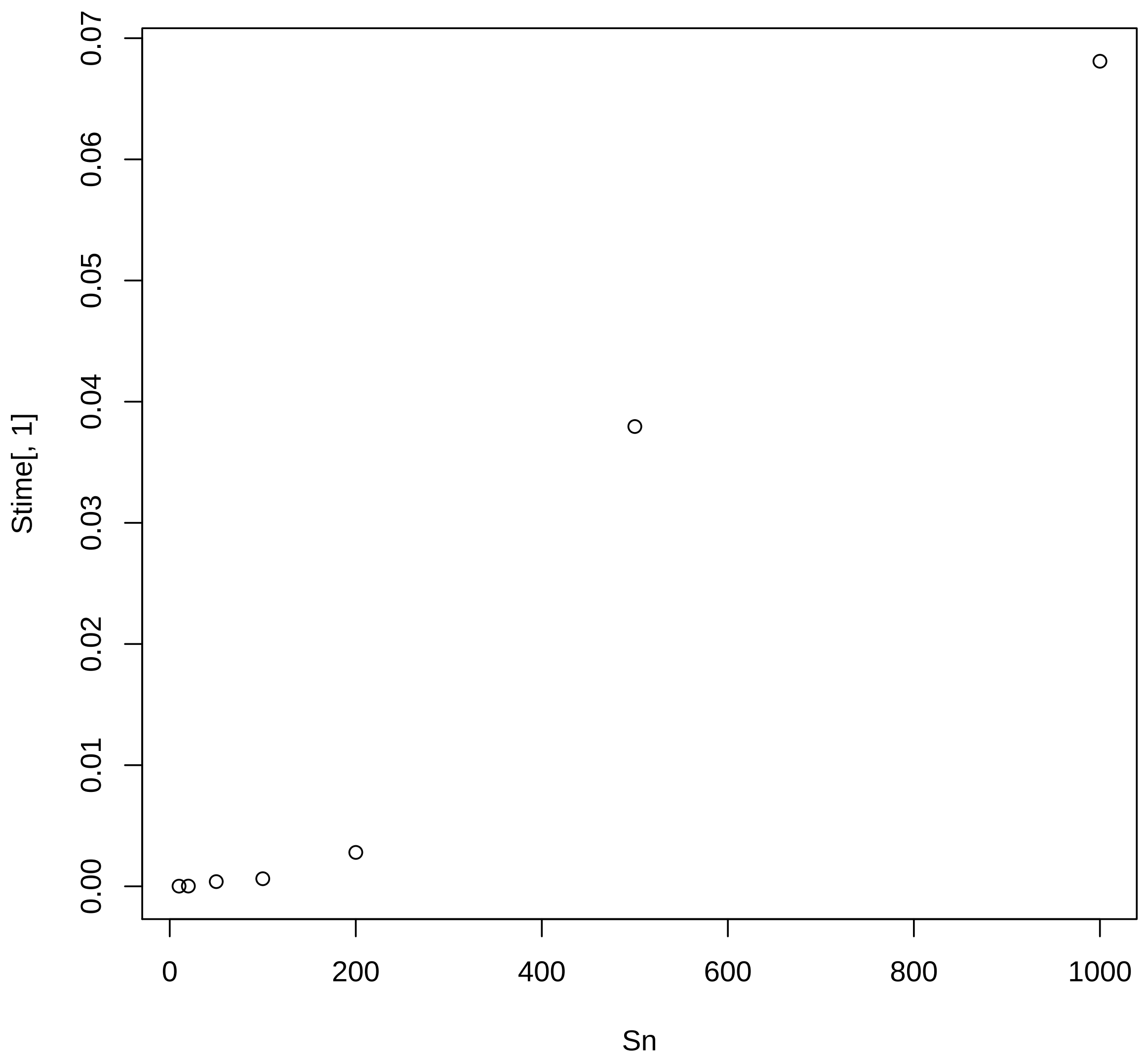
```r
S <- as.matrix(dist(0:(n - 1)))
S <- sd^2 * a^S
```

Choosing `sd=10` and `a=0.9` should be fine, but feel free to explore alternatives.

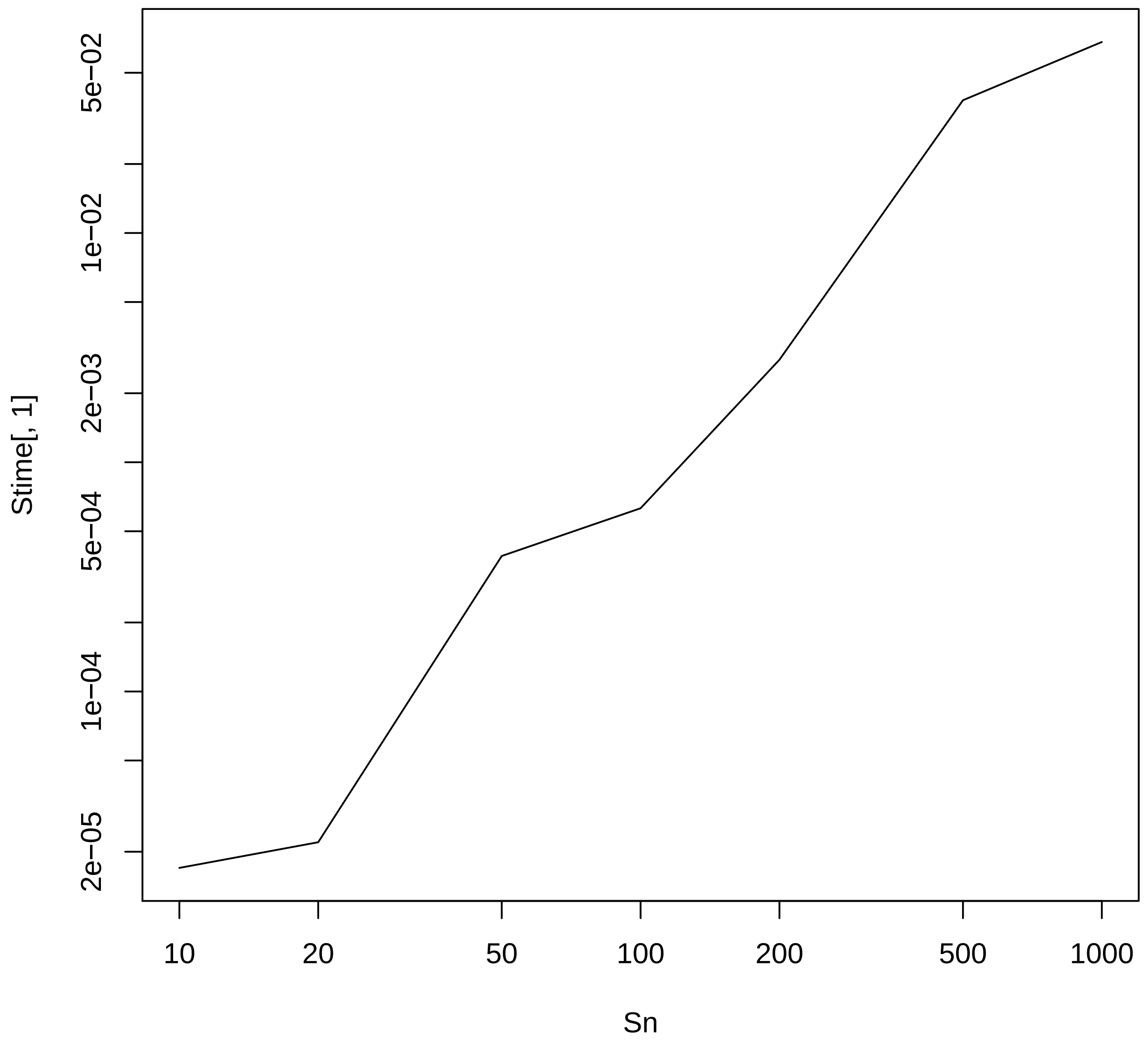
```r
Sn <- c(10, 20, 50, 100, 200, 500, 1000)
Stime <- c()
sd <- 10
a <- 0.9
for (n in Sn) {
  loop.max <- max(1, 10000/n)
  S <- as.matrix(dist(0:(n - 1)))
  S <- sd^2 * a^S
  Stime <- rbind(Stime,
                 system.time({
                   for (loop in 1:loop.max) {
                     B <- S*1
                     Matrix::chol(B)
                   }}) / loop.max)
}
Stime
```

```
##      user.self sys.self  elapsed user.child sys.child
## [1,]  0.000017 0.000000 0.000016          0         0
## [2,]  0.000022 0.000000 0.000020          0         0
## [3,]  0.000390 0.000355 0.000095          0         0
## [4,]  0.000630 0.000510 0.000140          0         0
## [5,]  0.002800 0.002340 0.000640          0         0
## [6,]  0.037950 0.027300 0.008350          0         0
## [7,]  0.068100 0.047100 0.014700          0         0
```

```
plot(Sn, Stime[,1], log="")
```



```r
plot(Sn, Stime[,1], type="l", log="xy")
```



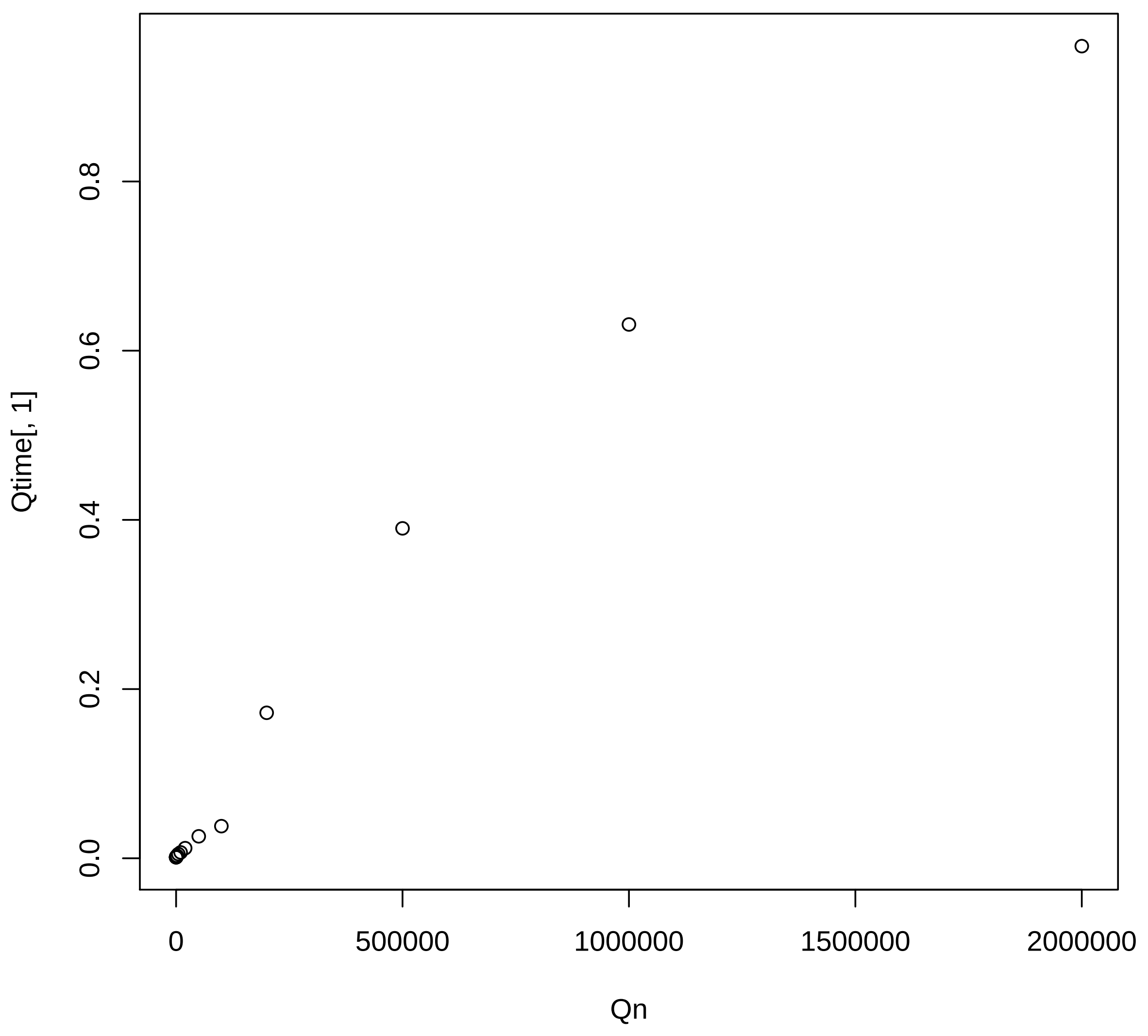
(c) Adapt the code to measure the time it takes to compute sparse Cholesky factorisations of precision matrices of an AR(1) process. Let n vary between 10 and 10^6. Note that you can create such a matrix with something like:

```r
Q <- Matrix::sparseMatrix(i = c(1:n, 2:n, 1:(n - 1)),
            j = c(1:n, 1:(n - 1), 2:n), x = rep(c(1, 1 + a^2,
                1, -a), c(1, n - 2, 1, 2 * (n - 1)))/(1 - a^2)/sd^2,
            dims = c(n, n))
```

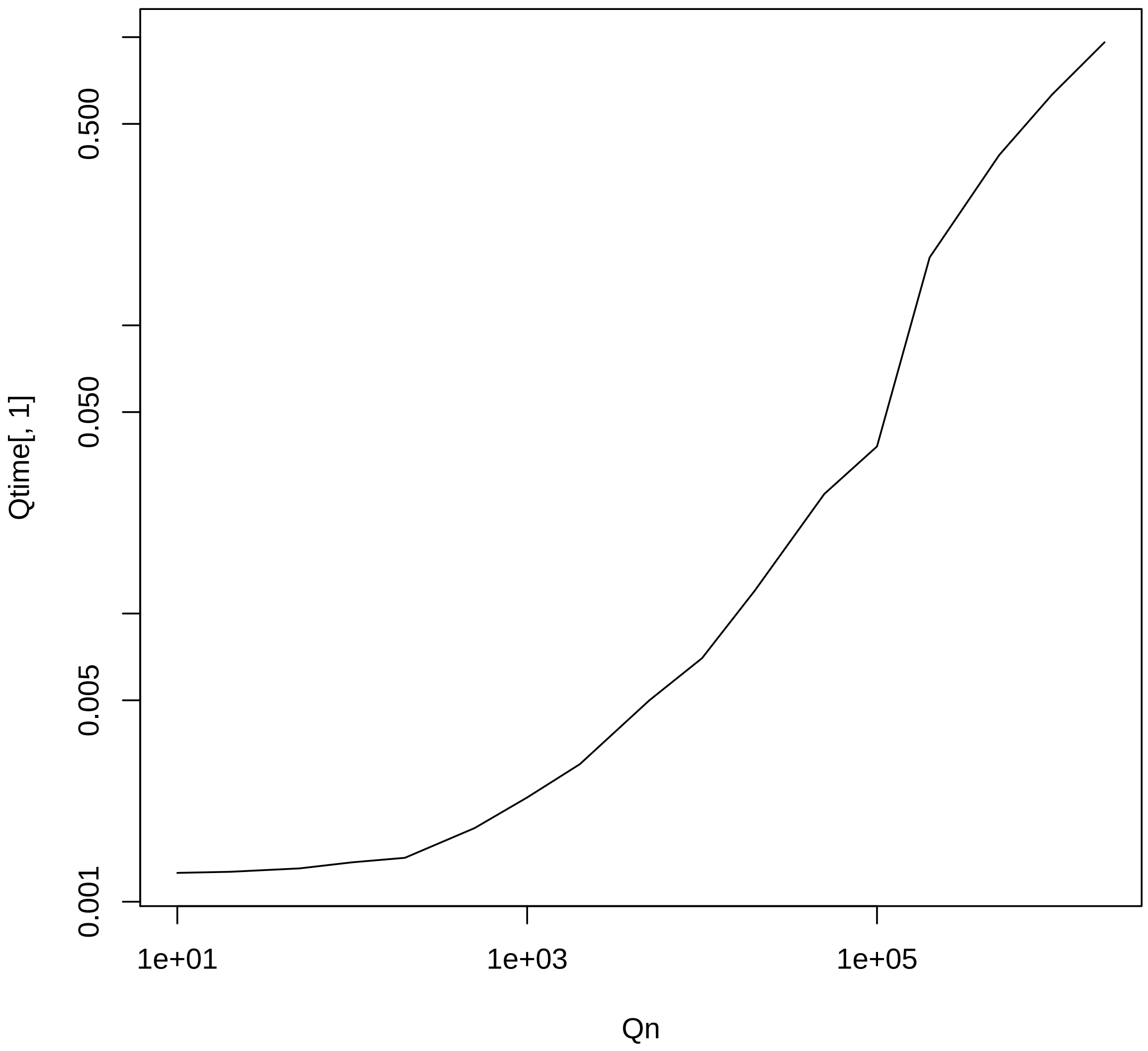
```r
Qn <- c(Sn, 2e3, 5e3, 1e4, 2e4, 5e4, 1e5, 2e5, 5e5, 1e6, 2e6)
Qtime <- c()
sd=10
a=0.9
for (n in Qn) {
  loop.max <- max(1, 10000/n)
  Q <- Matrix::sparseMatrix(i = c(1:n, 2:n, 1:(n - 1)),
            j = c(1:n, 1:(n - 1), 2:n), x = rep(c(1, 1 + a^2,
                1, -a), c(1, n - 2, 1, 2 * (n - 1)))/(1 - a^2)/sd^2,
            dims = c(n, n))
  Qtime <- rbind(Qtime,
                 system.time({
                   for (loop in 1:loop.max) {
                     B <- Q*1
                     Matrix::chol(B)
                   }}) / loop.max)
}
Qtime
```

```
##        user.self sys.self  elapsed user.child sys.child
##   [1,]  0.001259 0.000000 0.001259          0         0
##   [2,]  0.001270 0.000000 0.001270          0         0
##   [3,]  0.001305 0.000005 0.001305          0         0
##   [4,]  0.001370 0.000000 0.001370          0         0
##   [5,]  0.001420 0.000000 0.001420          0         0
##   [6,]  0.001800 0.000000 0.001750          0         0
##   [7,]  0.002300 0.000000 0.002300          0         0
##   [8,]  0.003000 0.000000 0.003000          0         0
##   [9,]  0.005000 0.000000 0.005000          0         0
##  [10,]  0.007000 0.000000 0.007000          0         0
##  [11,]  0.012000 0.001000 0.012000          0         0
##  [12,]  0.026000 0.000000 0.026000          0         0
##  [13,]  0.038000 0.003000 0.042000          0         0
##  [14,]  0.172000 0.000000 0.172000          0         0
##  [15,]  0.390000 0.005000 0.395000          0         0
##  [16,]  0.631000 0.020000 0.652000          0         0
##  [17,]  0.960000 0.128000 1.088000          0         0
```

```
plot(Qn, Qtime[,1], log="")
```

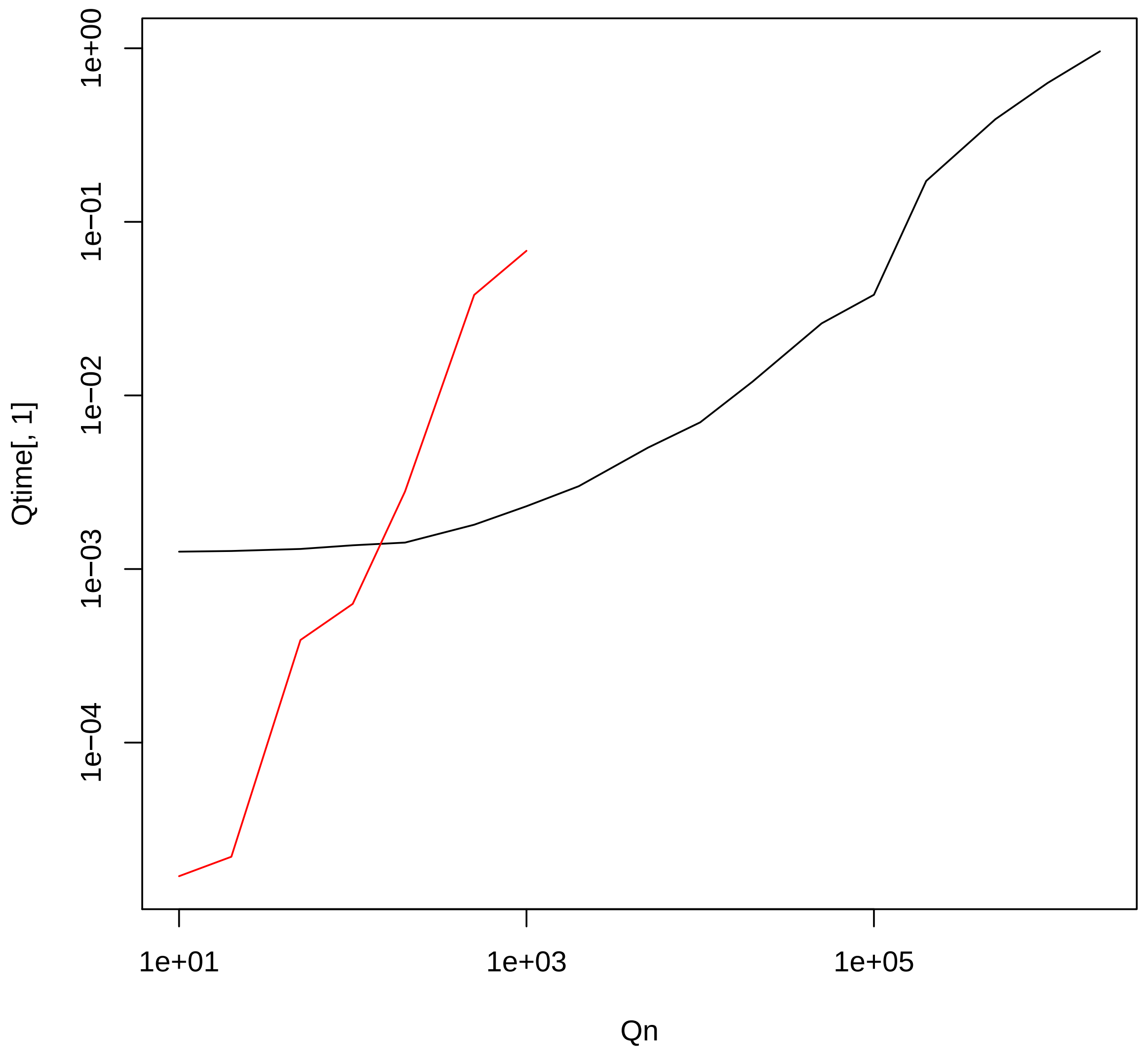


```r
plot(Qn, Qtime[,1], type="l", log="xy")
```



(d) Graphically compare the computational costs of dense and sparse Cholesky factorisations.

```r
plot(Qn, Qtime[,1], type="l", log="xy", ylim=range(Qtime[,1], Stime[,1]))
lines(Sn, Stime[,1], col=2)
```


1e+00
1e−01
1e−02
1e−03
1e−04
Qtime[, 1]
1e+01
1e+03
1e+05
Qn