APTS Statistical Computing: Practical Lab 1 (Tuesday)

Here are some practical problems which aim to explore and reinforce some of the course material. They all use R. This is just for convenience: many real statistical numerical analysis tasks are approached with a mixture of compiled code, using a statically typed language (like C, C++, Fortran, Java, Scala, ...), together with code in a high level language such as R, Python or Matlab. But for exploratory and pedagogic purposes, high level languages are often convenient. Some languages (such as Scala and Julia) attempt avoid the "two language problem" by being both high level and easy to develop in interactively, as well as fast and efficient.

Several problems here use simulated data: when developing statistical modelling code, it is often best to start out with data where you know what the truth is (and can generate further replicates). Do not consult the solutions[1] until you've made good attempts.

1. Download the example code extracted from the course notes, from the course website. Load this code into R-Studio (or your favourite R IDE), and execute some code blocks to make sure that you can reproduce some of the examples from the lecture notes.

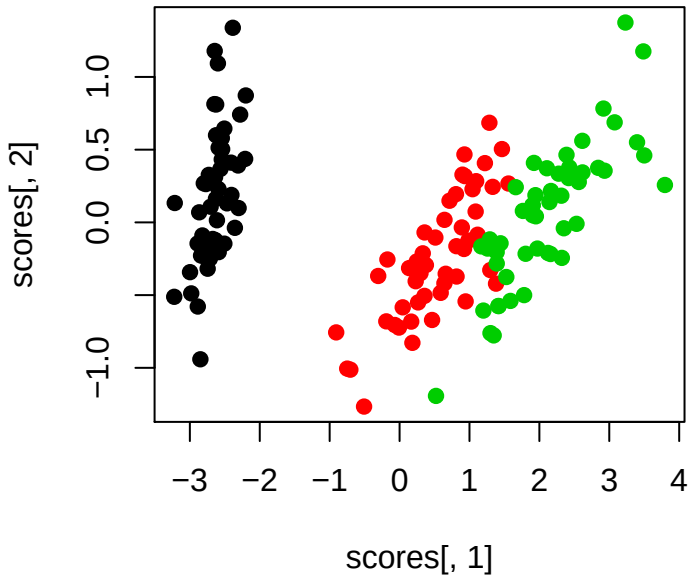
2. *Principal components analysis* (PCA) of a multivariate data set was traditionally based on the eigen-decomposition of the sample covariance matrix of the data. The matrix of eigenvectors can be used to rotate the (centred) data observations to a set of uncorrelated random quantities ordered by decreasing variance. These rotated data are often known as the *scores*. We can write a small function to implement this as follows.

```
pcScoresEig = function(X) {
  eig = eigen(var(X), symmetric=TRUE)
  Xc = sweep(as.matrix(X), 2, colMeans(X))
  Xc %*% eig$vectors
}
```

   We can test it on the infamous iris data, as follows.

```
Xi = iris[,-5]
scores = pcScoresEig(Xi)
plot(scores[,1], scores[,2], col=iris[,5], pch=19)
```

[1] Solutions will be made available at `https://www.staff.ncl.ac.uk/d.j.wilkinson/teaching/apts-sc/` before the end of the session.



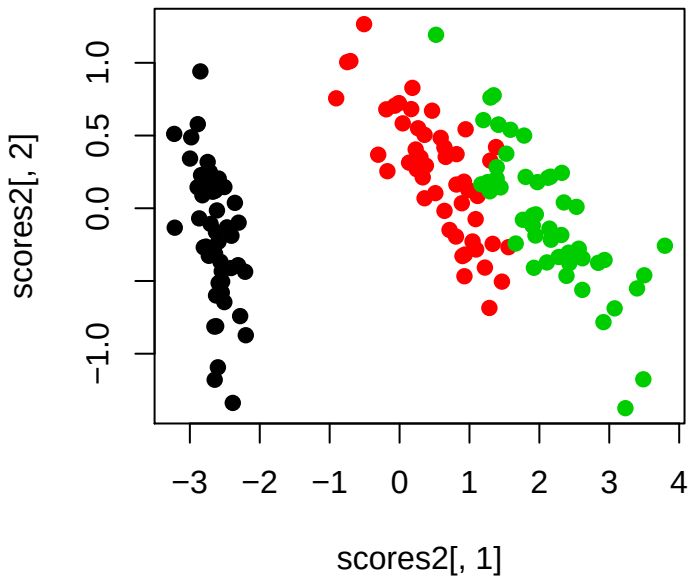
This is essentially how the `princomp` function in R is implemented, and we can verify this.

```
head(scores, 3)

##           [,1]       [,2]        [,3]        [,4]
## [1,] -2.684126  0.3193972  0.02791483 0.002262437
## [2,] -2.714142 -0.1770012  0.21046427 0.099026550
## [3,] -2.888991 -0.1449494 -0.01790026 0.019968390

head(princomp(Xi)$scores, 3)

##         Comp.1     Comp.2      Comp.3      Comp.4
## [1,] -2.684126  0.3193972  0.02791483 0.002262437
## [2,] -2.714142 -0.1770012  0.21046427 0.099026550
## [3,] -2.888991 -0.1449494 -0.01790026 0.019968390
```

(a) It turns out that the singular value decomposition of the (centred) data matrix can be used to construct the scores directly as $\mathbf{UD}$. Think about why this is true, and write an R function, `pcScoresSvd` to implement this. Test it on the iris data, and don't worry about a sign flip.

```
## Y = XV = UDV'V = UD

pcScoresSvd = function(X) {
    Xc = sweep(as.matrix(X), 2, colMeans(X))
    SVD = svd(Xc)
```

```
    SVD$u %*% diag(SVD$d)
}
scores2 = pcScoresSvd(Xi)
plot(scores2[,1], scores2[,2], col=iris[,5], pch=19)
```



(b) This SVD-based method is a bit cheaper, and much more numerically stable than the eigen-decomposition method, and works for data that is not of full rank, including wide data ($p > n$). It also gives some additional insight into what the SVD "means". This is essentially how the `prcomp` function in R is implemented (which is almost always preferred to the `princomp` function). Compare your function with this.

```
head(scores2, 3)
```

```
##           [,1]       [,2]        [,3]        [,4]
## [1,] -2.684126 -0.3193972  0.02791483 0.002262437
## [2,] -2.714142  0.1770012  0.21046427 0.099026550
## [3,] -2.888991  0.1449494 -0.01790026 0.019968390
```

```
head(prcomp(Xi)$x, 3)
```

```
##           PC1        PC2         PC3         PC4
## [1,] -2.684126 -0.3193972  0.02791483 0.002262437
## [2,] -2.714142  0.1770012  0.21046427 0.099026550
## [3,] -2.888991  0.1449494 -0.01790026 0.019968390
```

(c) Simulate some random (eg.) $5,000 \times 1,000$ test data, and time your two implementations. You should find that the SVD method is quicker, though that isn't really the point — the point is that it is more numerically stable.

```
X = matrix(rnorm(5000*1000),ncol=1000)
system.time(pcScoresEig(X))

##    user  system elapsed
##   5.091   1.534   3.409

system.time(pcScoresSvd(X))

##    user  system elapsed
##   5.908   1.722   1.100
```

(d) Think about how to use the SVD to compute the variances or standard deviations of the principal components. The variances would be given by the diagonal of the $\mathbf{D}$ matrix for the eigen-decomposition approach, but this doesn't quite work for the SVD approach. Modify your function to compute them. Check it against `prcomp` for the iris data to make sure you've done it correctly.

```
## S = X'X/(n-1) = (UDV')'UDV'/(n-1) = VDDV'/(n-1) = V[D^2/(n-1)]V'
## So D basically gives the standard deviations, rather than the variances,
## but need to divide by (n-1) somewhere

pcSdsSvd = function(X) {
    Xc = sweep(as.matrix(X), 2, colMeans(X))
    SVD = svd(Xc)
    SVD$d / sqrt(dim(Xc)[1] - 1)
}
pcSdsSvd(Xi)

## [1] 2.0562689 0.4926162 0.2796596 0.1543862

prcomp(Xi)$sd

## [1] 2.0562689 0.4926162 0.2796596 0.1543862
```

3. For a linear regression model,

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

we know that the quadratic loss $L_0(\boldsymbol{\beta}) = \|\boldsymbol{\epsilon}\|^2$ is minimised wrt $\boldsymbol{\beta}$ when $\boldsymbol{\beta}$ is a solution to the normal equations,

$$\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^\mathsf{T}\mathbf{y}.$$

In *ridge regression*, the slightly modified quadratic loss function $L_\lambda(\boldsymbol{\beta}) = \|\boldsymbol{\epsilon}\|^2 + \lambda\|\boldsymbol{\beta}\|^2$ is used, for some ridge penalty $\lambda > 0$, which encourages shrinkage of the regression coeffiecients towards zero.

(a) Show that for a given fixed $\lambda > 0$, the loss $L_\lambda(\boldsymbol{\beta})$ is minimised when $\boldsymbol{\beta}$ is a solution to

$$(\mathbf{X}^\mathsf{T}\mathbf{X} + \lambda\mathbf{I})\boldsymbol{\beta} = \mathbf{X}^\mathsf{T}\mathbf{y}.$$

```
## L = e'e + lb'b = (y-Xb)'(y-Xb) + lb'b = y'y - 2b'X'y + b'X'Xb + lb'b
## grad L = -2X'y + 2X'Xb + 2lb = -2X'y + 2(X'X + lI)b
## So grad L = 0 => (X'X + lI)b = X'y
```

(b) Starting from the singular value decomposition, $\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathsf{T}}$, show that the optimal $\widehat{\boldsymbol{\beta}}_\lambda$ can be written as

$$\widehat{\boldsymbol{\beta}}_\lambda = \mathbf{V}\mathbf{D}_\lambda\mathbf{U}^{\mathsf{T}}\mathbf{y},$$

where $\mathbf{D}_\lambda$ is a diagonal matrix with entries $d_i^\lambda = d_i/(d_i^2 + \lambda)$. Note that this means $\widehat{\boldsymbol{\beta}}_\lambda$ can be computed for as many different λ as desired, all for the cost of one single expensive SVD operation.

```
##     (X'X + lI)b = X'y
## => (VDU'UDV' + lI)b = VDU'y
## => (VDDV' + lI)b = VDU'y
## => V(DD + lI)V'b = VDU'y
## => b = V(DD + lI)^{-1}DU'y
## => b = V[(DD + lI)^{-1}D]U'y
## => b = VEU'y, where diagonal E = (DD + lI)^{-1}D
```

(c) In practice, both the data, $\mathbf{y}$, and the covariate matrix $\mathbf{X}$ are centred before ridge regression is applied, since then the model can be fit without an intercept, and typically you would not want to shrink the intercept. Write a function,

```
ridge(y, X, lambda)
```

which expects an n-vector `y`, an $n \times p$ matrix `X`, and a q-vector of λ values where the ridge solution is required. The function should return a $p \times q$ matrix of ridge regression parameters, with each column representing a solution for a given λ.

```
ridge = function(y, X, lambda) {
  y = y - mean(y)
  X = sweep(as.matrix(X), 2, colMeans(X))
  SVD = svd(X)
  uty = as.vector(t(SVD$u) %*% y)
  D = outer(SVD$d, lambda, function(d,l){d/(d*d+l)})
  SVD$v %*% (D * uty) # 1st product is matrix, 2nd is elementwise
}
```

(d) For the `trees` dataset, regress volume on the other two variables for a range of shrinkage parameters.

```
ridge(trees[,3], trees[,1:2], c(0,exp(0:5)))
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 4.7081605 4.6868475 4.6507090 4.5554535 4.3166299 3.7858307 2.8648009
## [2,] 0.3392512 0.3444192 0.3531561 0.3760244 0.4322559 0.5504595 0.7226325
```

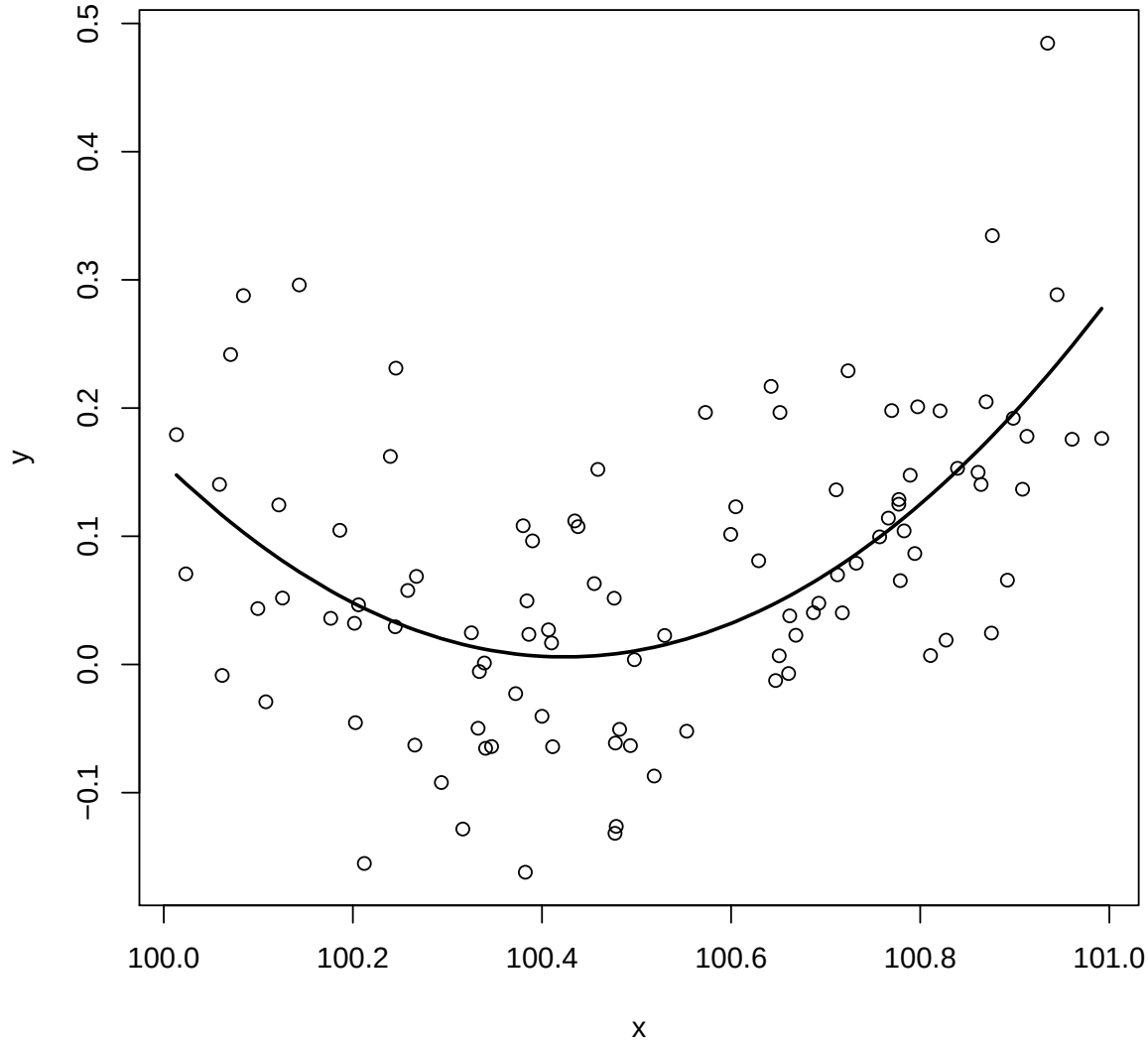
Ensure that your solution matches up with that of `lm` in the case $\lambda = 0$.

```
lm(as.vector(trees[,3]) ~ as.matrix(trees[,1:2]))$coefficients

##                    (Intercept)  as.matrix(trees[, 1:2])Girth
##                     -57.9876589                     4.7081605
## as.matrix(trees[, 1:2])Height
##                      0.3392512
```

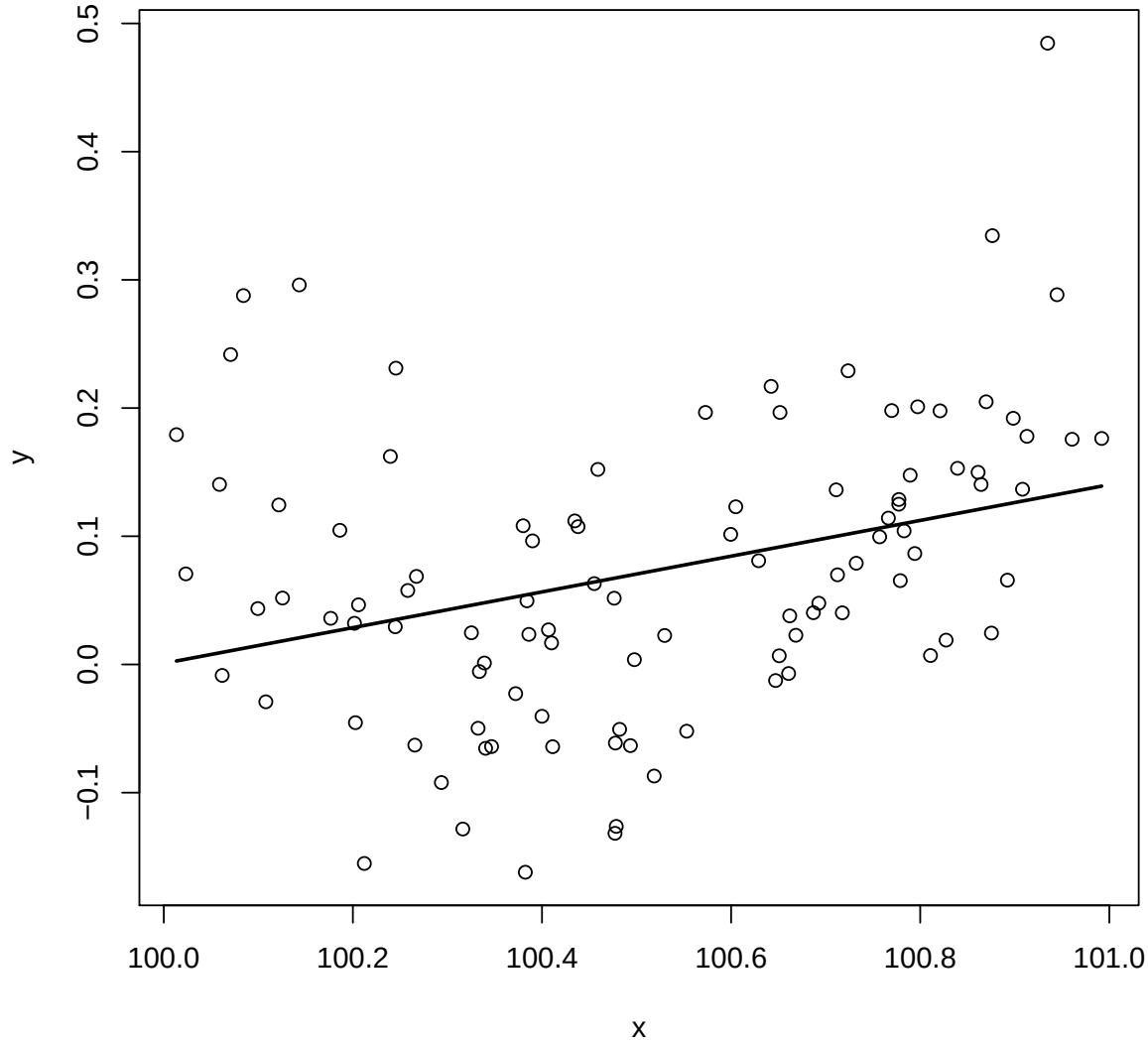
4. This question revisits the third example in section 1.1 of the notes.

   (a) Run the following code that reproduces the good and bad fits produced by calls to `lm` in the example.

```
## First, simulate the data
set.seed(1)
x <- sort(runif(100)) + 100
y <- .2*(x-100 -.5)+(x-100 -.5)^2 + rnorm(100)*.1

## lm manages to estimate the regression
plot(x,y)
b <- lm(y~x+I(x^2))
lines(x,fitted(b),lwd=2)
```




```
## direct solution of the normal equations doesn't work
X <- model.matrix(b)
beta.hat <- solve(t(X)%*%X, t(X)%*%y)

## Error in solve.default(t(X) %*% X, t(X) %*% y):  system is computationally
singular:  reciprocal condition number = 3.98647e-19

## What if the x-values were even further away from zero?
## In theory, this should still be able to represent the quadratic function.
x1 <- x+1000
plot(x,y)
b1 <- lm(y~x1+I(x1^2))
lines(x,fitted(b1),lwd=2)
```




```
## lm fails without warning!
```

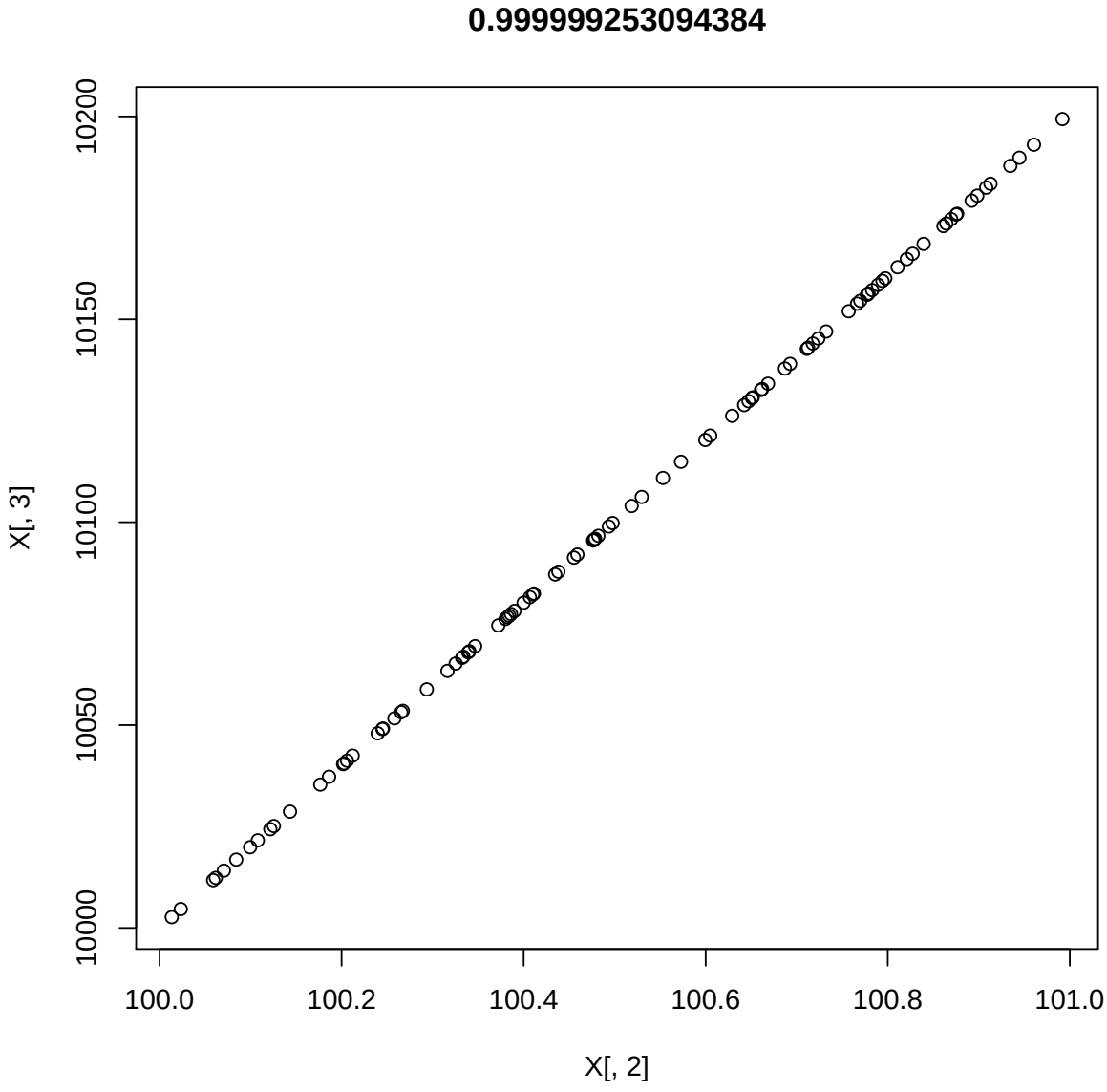
(b) Examine the condition numbers of the model matrices in the two cases (`lm` computes the fit using the QR decomposition approach, not by direct solution of the normal equations). Why was the second `lm` fit so bad?

```
d <- svd(X)$d
d[1]/d[3] ## manageable

## [1] 1560769713

X1 <- model.matrix(b1)
d <- svd(X1)$d
d[1]/d[3] ## very high
```

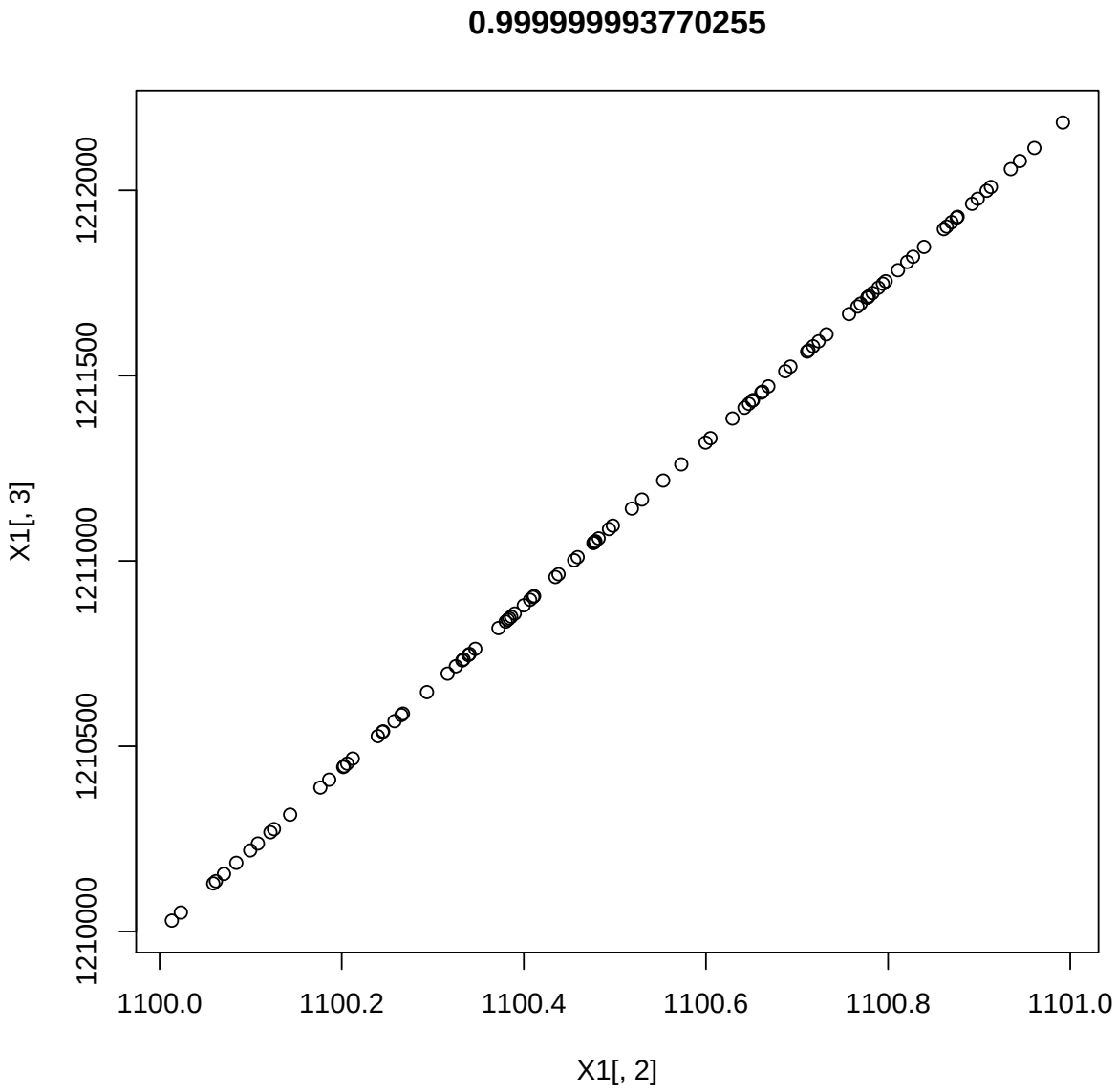
```
## [1] 2.242481e+13

## ...could also use function kappa, to *estimate* condition number.
```

(c) Plot the second and third columns of the model matrix against each other for the two cases, and use `cor` to examine their correlation. Why are the condition numbers so high here?

```
plot(X[,2],X[,3],main=cor(X[,2],X[,3]))      ## near rank deficient
```



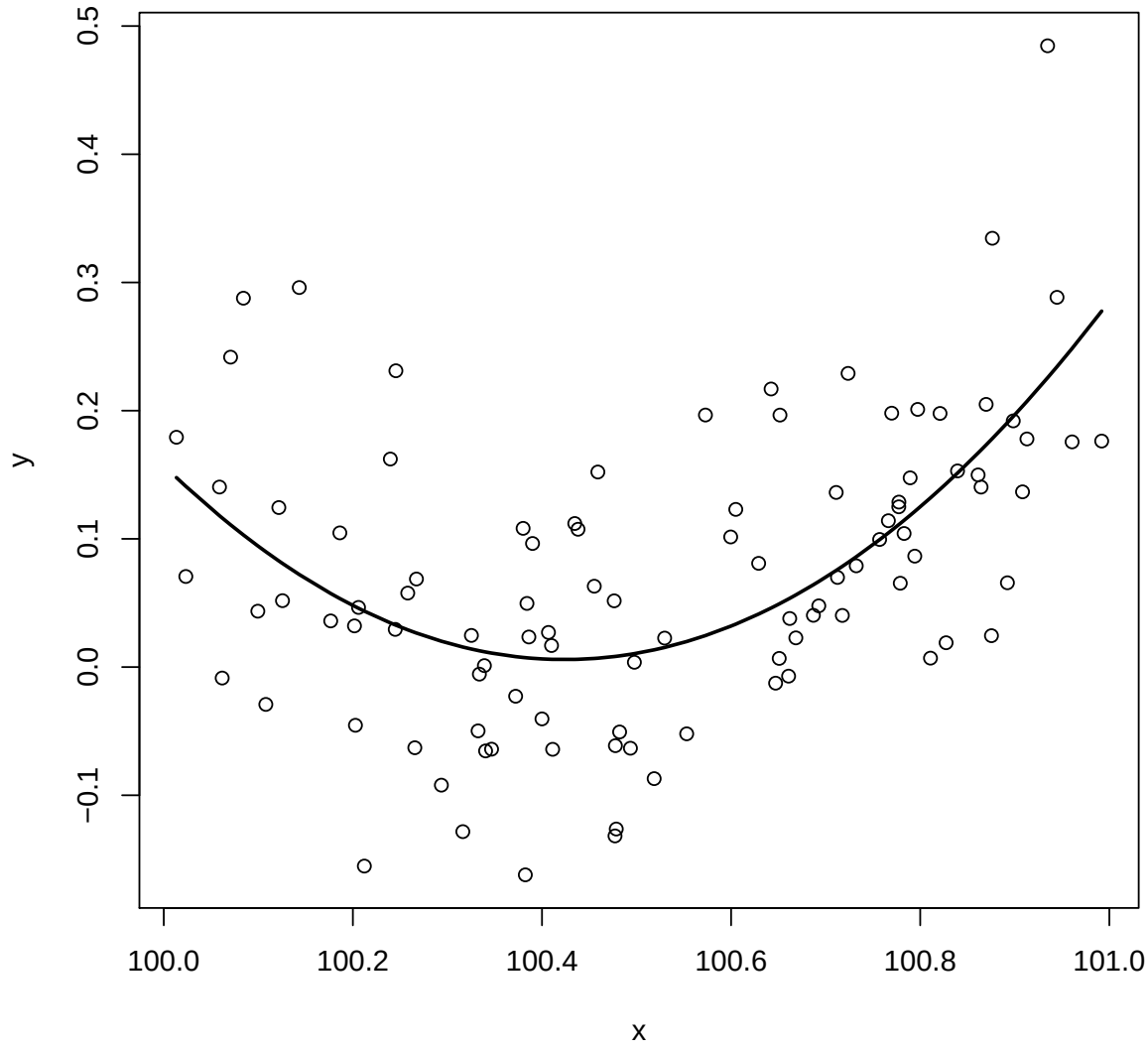
```r
plot(X1[,2],X1[,3],main=cor(X1[,2],X1[,3])) ## even nearer!
```



(d) Since the linear model says simply that the expected value vector $E(\mathbf{y})$ lies in the space spanned by the columns of $\mathbf{X}$, one possibility is to attempt to arrive at a better conditioned $\mathbf{X}$ by linear rescaling and/or recombination of its columns. This is always equivalent to a linear reparameterization. Try this on the model matrix of the model that causes `lm` to fail. In particular, for each column (except the intercept column) subtract the column mean. Then divide each column (except the intercept column) by its standard deviation. Find the condition number of the new model matrix. Fit the model with this model matrix using something like `lm(y ~ Xs - 1)` where `Xs` is the re-scaled model matrix. Produce a plot that confirms that the resulting fit is sensible now.

```r
Xs <- X1
Xs[,2:3] <- sweep(Xs[,2:3],2, colMeans(Xs[,2:3]))
```

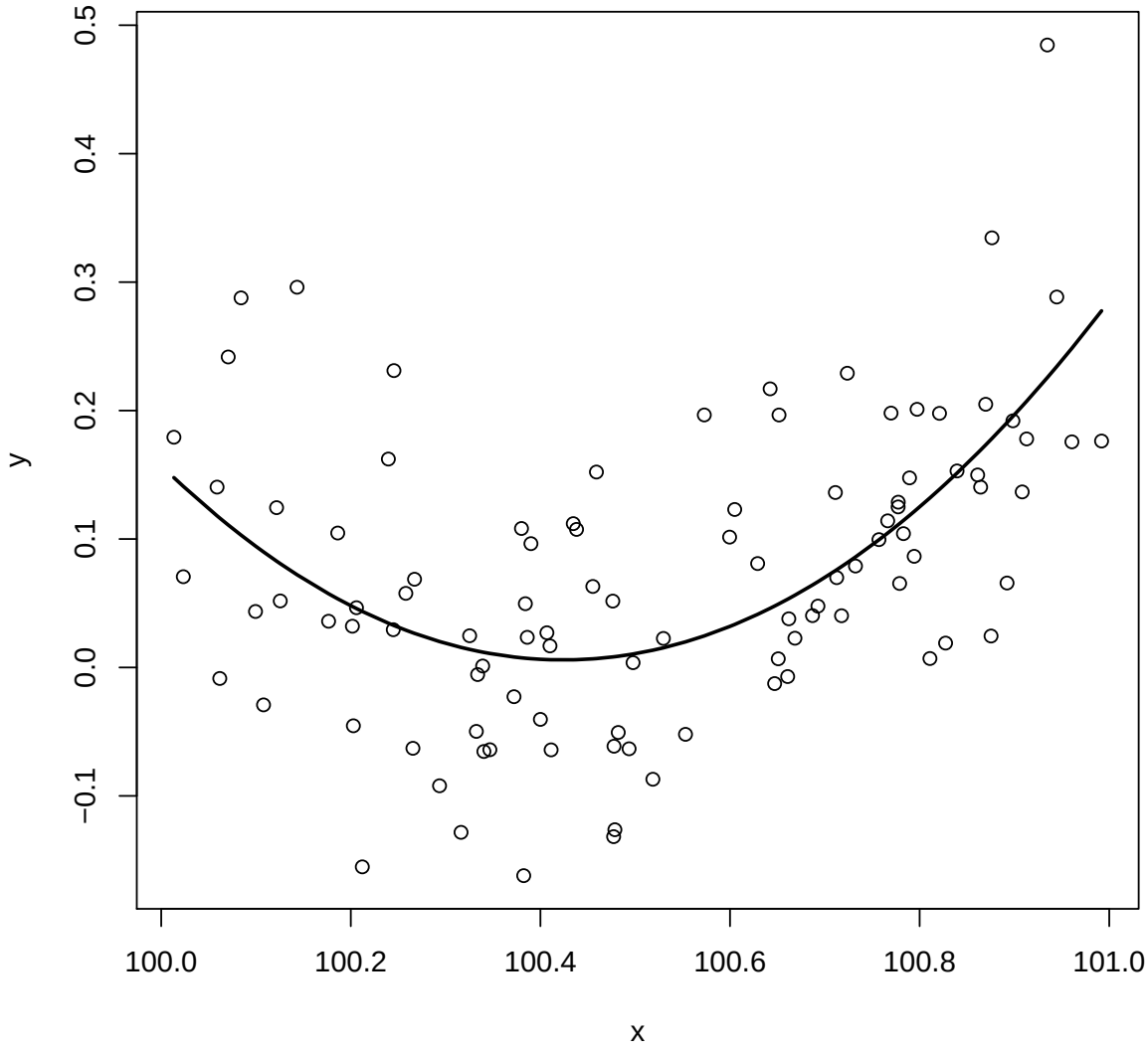
```
Xs[,2] <- Xs[,2]/sd(Xs[,2])
Xs[,3] <- Xs[,3]/sd(Xs[,3])

bs <- lm(y ~ Xs-1)
plot(x,y)
lines(x,fitted(bs),lwd=2)
```



(e) An alternative fix is to subtract the mean `x` value from the original `x` vector before fitting. Try this and see what happens to the condition number and column correlations of the model matrix now.

```
x2 <- x - mean(x)
b2 <- lm(y ~ x2 + I(x2^2))
plot(x,y)
```

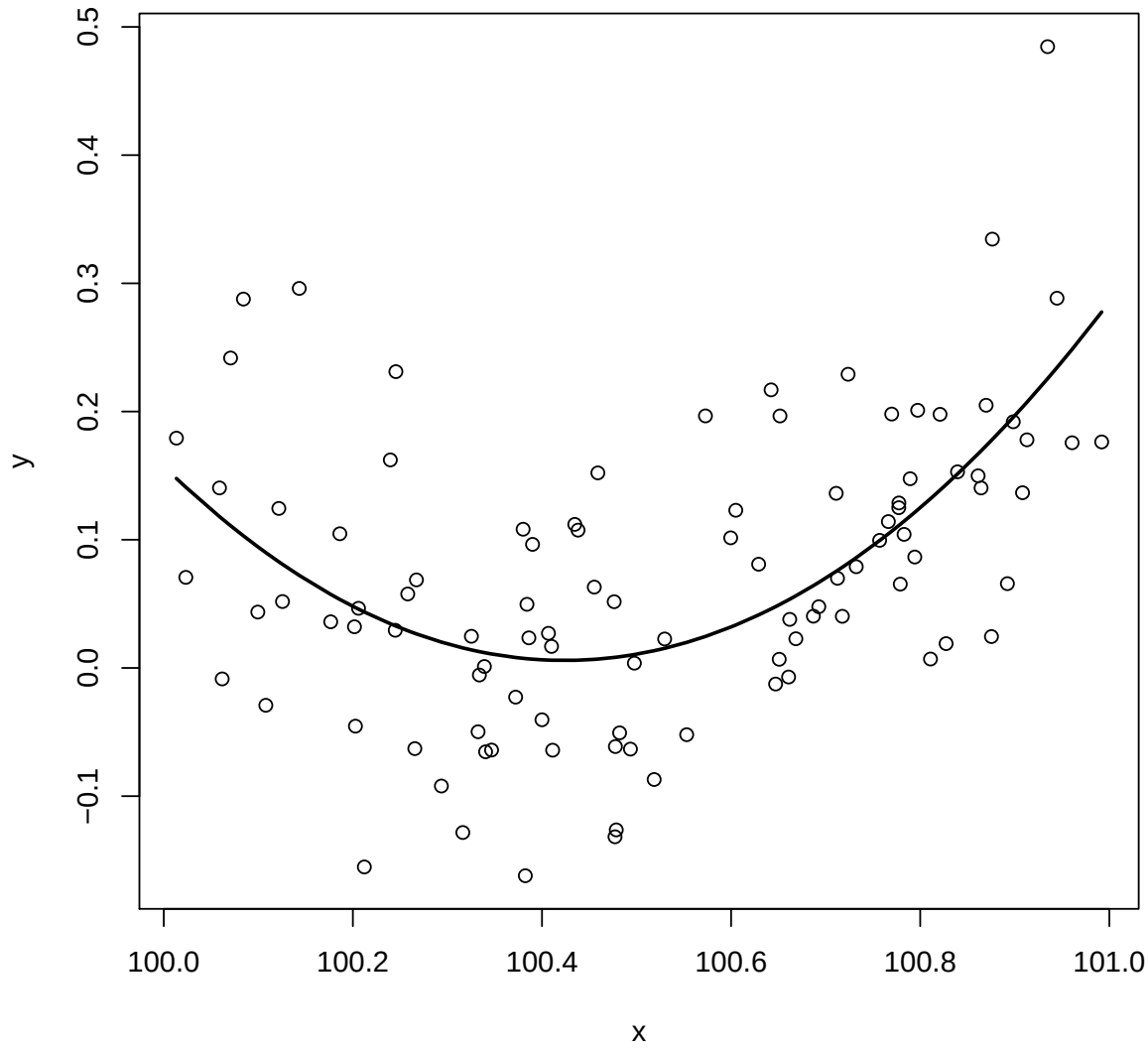
```
lines(x,fitted(b2),lwd=2)
```



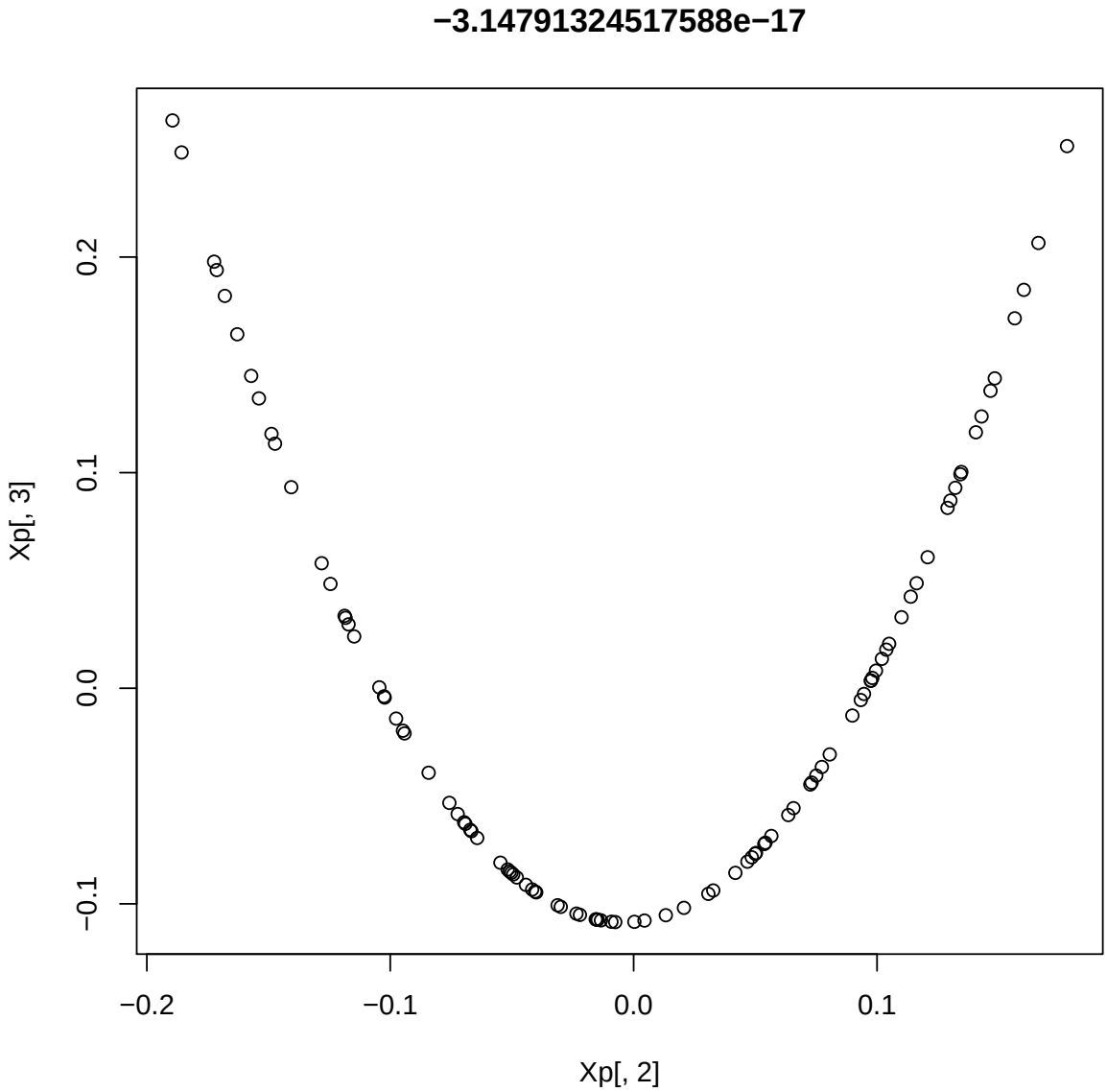

```
X2 <- model.matrix(b2)
d <- svd(X2)$d
d[1]/d[3]

## [1] 15.36915
```

(f) If we want good condition numbers then the best thing would be have a model matrix made up of columns from an orthogonal matrix. *Orthogonal polynomials* provide a way of achieving this. Try fitting with `lm(y~poly(x,2))` to see these in action. Look at the correlation between the model matrix columns now, and the condition number.

```r
bp <- lm(y~poly(x,2))
plot(x,y)
lines(x,fitted(bp),lwd=2)
```




```r
Xp <- model.matrix(bp)
d <- svd(Xp)$d
d[1]/d[3]

## [1] 10

## ...perhaps plot columns too, and look at correlation
plot(Xp[,2],Xp[,3],main=cor(Xp[,2],Xp[,3]))     ## orthogonal; no linear correl
```



(g) The model matrix produced in the last part is not quite (column) orthogonal (meaning orthonormal), but to correct this we could rescale and use a model matrix `X <- cbind(n^-.5,poly(x,2))` where `n` is the number of data. Without computing anything new, find a QR decomposition of this new model matrix? Show how the new model matrix can be used directly to find the fitted values in this case, without any need for an `lm` call. Check that it works.

```
## g) Then poly can produce perfectly conditioned model matrices
X <- cbind(length(x)^-.5,poly(x,2))
t(X) %*% X ## orthogonal

##                               1             2
##    1.000000e+00 -2.676717e-17 5.829481e-17
## 1 -2.676717e-17  1.000000e+00 9.590758e-17
## 2  5.829481e-17  9.590758e-17 1.000000e+00
```

```r
## So QR factorization is trivial! (It's Q=X R=I) =>
mu <- X %*% (t(X) %*% y)
plot(x,y)
lines(x,mu)
```