Performance assessment of *vsn* with simulated data

Wolfgang Huber

November 30, 2008

Contents

1 Overview

The purpose of this vignette is to assess that the software in *vsn* does what it is intended do, and in particular, to assess the performance of the parameter estimation on simulated data where the true parameters are known.

There are two functions `sagmbSimulateData` and `sagmbAssess` that can be used to generate simulated data and assess the difference between the 'true' and 'estimated' data calibration and transformation by *vsn*. This vignette demonstrates some examples. Please refer to reference [1] for more detail on the simulation model, the assessment strategy and a comprehensive suite of assessments with respect to the number of features `n`, the number of arrays `d`, the fraction of differentially expressed genes `de`, and the fraction of up-regulated genes `up`.

2 Helper functions used in this document

This section is given just for completeness – for the results, you can skip this and go to Section 3.

```
> library("vsn")
> set.seed(0xabcd)
```

The function `sim` computes simulated data using the function `sagmbSimulateData` from the *vsn* package, calls `vsn2` to fit the VSN model, and assesses model fit using the function `sagmbSimulateData`.

```
> sim = function(..., lts.quantile=1, nrrep=30L) {
+   callpar = list(...)
+   ll      = listLen(callpar)
+   stopifnot(ll[1]>=1, all(ll[-1]==1))
+   res  = matrix(1, nrow=nrrep, ncol=ll[1])
+
+   ## default parameters
+   simpar = append(callpar,
+             list(n=4096L, d=2L, de=0, up=0.5, nrstrata=1L, miss=0, log2scale=TRUE))
+   simpar = simpar[!duplicated(names(simpar))]
+
+   for (i in 1:ll[1]) {
+     simpar[[1]] = callpar[[1]][i]
+     for (r in 1:nrrep) {
+       sim = do.call("sagmbSimulateData", simpar)
+       ny  = vsn2(sim$y, strata=factor(sim$strata), lts.quantile=lts.quantile, verbose=!TRUE)
+       res[r, i] = sagmbAssess(exprs(ny), sim)
+       if(!TRUE){
+         cat(paste(sprintf("%6g", signif(sim$coefficients, 3), collapse=" ")), "")
+         cat(sprintf("i=%d, r=%d: %g\n", i, r, signif(res[r,i], 3)))
+         plot(sim$y, pch="."); abline(a=0, b=1, col="orange", main=paste(i,r))
+         if(res[r, i]>0.02) browser()
+       }
+    } ## for r
+   } ## for i
+   return(res)
+ }
```

Here some functions to automate the formating of the plots that are used in the following.

```
> onePlot = function(n, res, log="xy", ...) {
+   matplot(n, t(res), pch=20, log=log, ylab='r.m.s. error', col="orange",
+           xlab=deparse(substitute(n)), ...)
+   lines(n, colMeans(res), col="blue")
+ }
> twoPlot = function(n, rl) {
+   par(mfrow=c(1,2))
+   ylim=range(unlist(rl))
+   for(i in seq(along=rl)) {
+     x = if(is.list(n)) n[[i]] else n
+     matplot(x, t(rl[[i]]), pch=20, ylab='r.m.s. error', xlab=deparse(substitute(n)),
+             log="y", ylim=ylim, col="orange", main=names(rl)[i])
+     lines(x, colMeans(rl[[i]]), col="blue")
+   }
+ }
> makeFig = function(name, width, height, expr) {
+   pdfname = paste(name, "pdf", sep=".")
+   pdf(file=pdfname, width=4*width, height=4*height)
+   expr
+   dev.off()
```

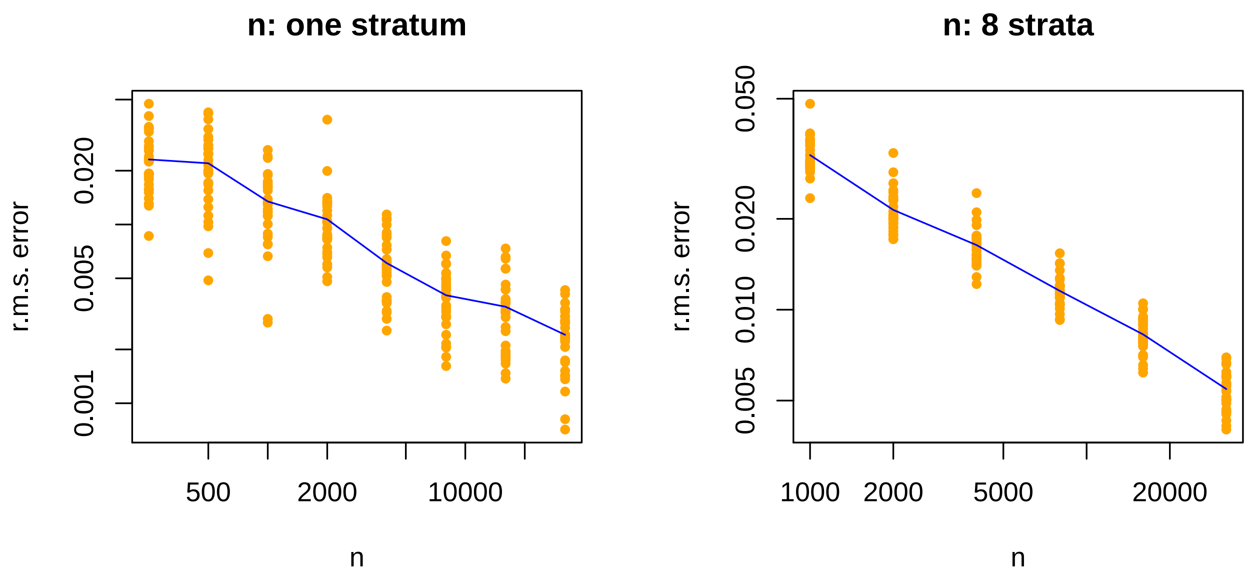

Figure 1: Estimation error as a function of the number of features n. If `vsn` works correctly, the estimation error should decrease roughly as $n^{-1/2}$.

```
+   invisible(pdfname)
+ }
```

3 Number of features n

Fig. 1 shows the estimation error for the transformation (i. e. the root mean squared difference between true and estimated transformed data) as a function of the number of features n. If `vsn` works correctly, the estimation error should decrease roughly as $n^{-1/2}$.

```
> n = 1000*2^seq(-2, 5)
> makeFig("fign1", 1, 1, {
+   res = sim(n=n)
+   onePlot(n, res, main="n: one stratum")
+ })

> n = 1000*2^seq(0, 5)
> makeFig("fign2", 1, 1, {
+   res = sim(n=n, nrstrata=8)
+   onePlot(n, res, main="n: 8 strata")
+ })
```

4 Number of samples d

Fig. 2a shows the estimation error as a function of the number of samples d. This curve is essentially flat. This is because the number of parameters that need to be estimated is proportional to d, so the ”number of data points per parameter” is constant in this plot (in contrast to Fig. 1).

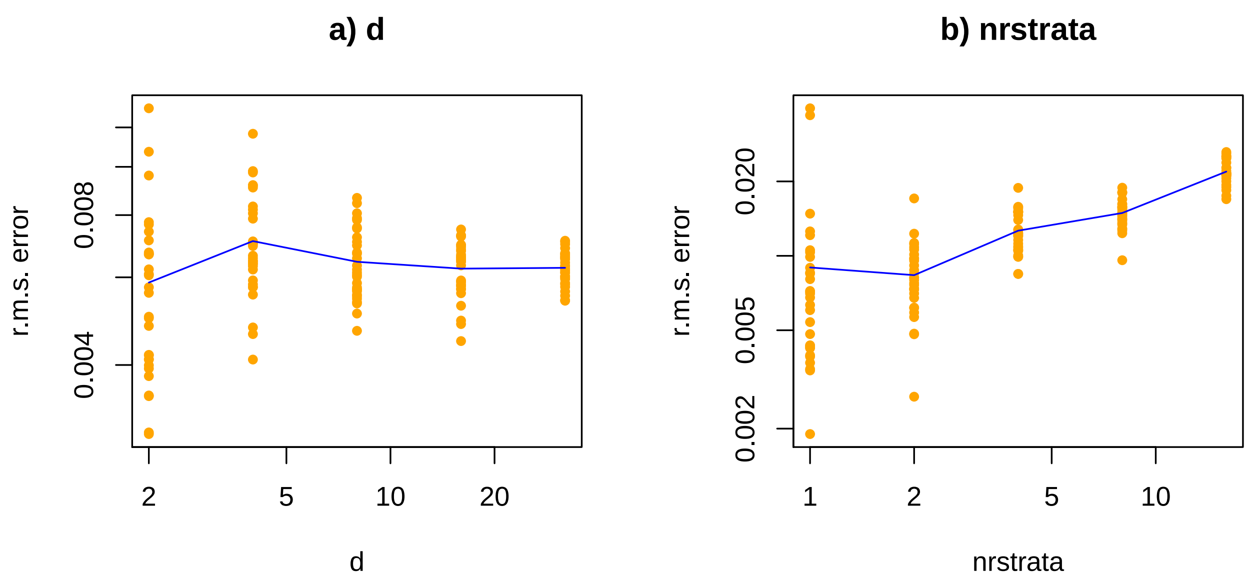

Figure 2: Estimation error as a function of (a) the number of samples and (b) the number of strata. See Sections 4 and 5.

```
> makeFig("figd", 1, 1, {
+   d = 2^seq(1, 5)
+   res = sim(d=d)
+   onePlot(d, res, main="a) d")
+ })
```

5 Number of strata

In Fig. 2b, we see the estimation error as a function of the number of strata. It should increase, since for each stratum, we need to estimate separate parameters, and if the overall number of features does not change, more strata means less data per parameter.

```
> makeFig("fignrstrata", 1, 1, {
+ nrstrata = 2^seq(0, 4)
+ res = sim(nrstrata=nrstrata)
+ onePlot(nrstrata, res, main="b) nrstrata")
+ })
```

6 Differentially expressed genes

In the following code, `de` is the fraction of differentially expressed genes. We run the simulation both with `vsn2`'s default setting `lts.quantile=0.9` and the more robust `lts.quantile=0.5`. The reason why `lts.quantile=0.5` is not the default is that the estimator with `lts.quantile=0.9` is more efficient (more precise with less data) *if* the fraction of differentially expressed genes is not that large. See Figure 3.

```
> makeFig("figdiff", 2, 1, {
+ de  = (0:6)/10
```

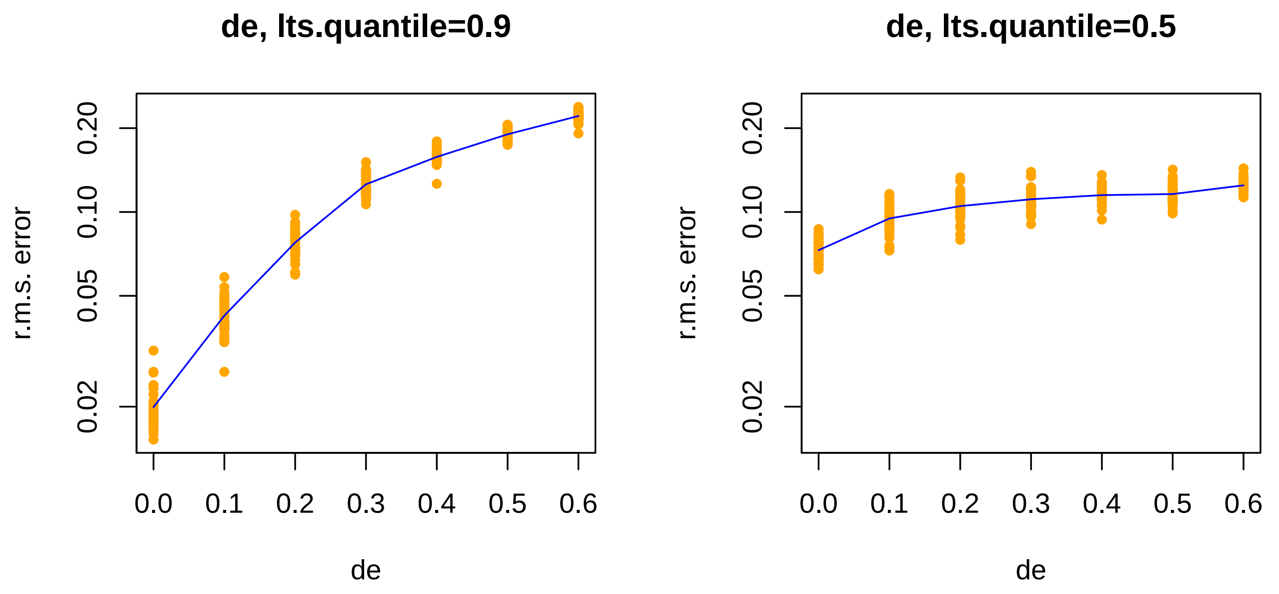

Figure 3: Estimation error as a function of the number of differentially expressed genes, for two different settings of `lts.quantile`. Note how a) is better for small values auf `de`, but becomes worse for larger values of `de`. See Section 6.

```
+ res1 = sim(de=de, nrstrata=2, lts.quantile=0.9)
+ res2 = sim(de=de, nrstrata=2, lts.quantile=0.5)
+ twoPlot(de, list("de, lts.quantile=0.9"=res1, "de, lts.quantile=0.5"=res2))
+ })
```

In the next code chunk, `up` is the fraction of up-regulated genes among the differentially expressed genes.

```
> makeFig("figup", 2, 1, {
+ up  = (0:8)/8
+ res1 = sim(up=up, nrstrata=2, de=0.2)
+ res2 = sim(up=up, nrstrata=2, de=0.2, lts.quantile=0.5)
+ twoPlot(up, list("a) up, lts.quantile=0.9"=res1, "b) up, lts.quantile=0.5"=res2))
+ })
```

7 Missing values

In this Section, we check the impact of missing values on the performance of the estimator. `miss` is the fraction of missing values in the overall

```
> makeFig("figmiss", 2, 1, {
+ miss1 = seq(0, 0.5, length=6)
+ res1 = sim(miss=miss1, d=8)
+ miss2 = seq(0, 0.1, length=6)
+ res2 = sim(miss=miss2, d=2)
+ twoPlot(list(miss1, miss2), list("fraction NA (d=8)"=res1, "b) fraction NA (d=2)"=res2))
+ })
```

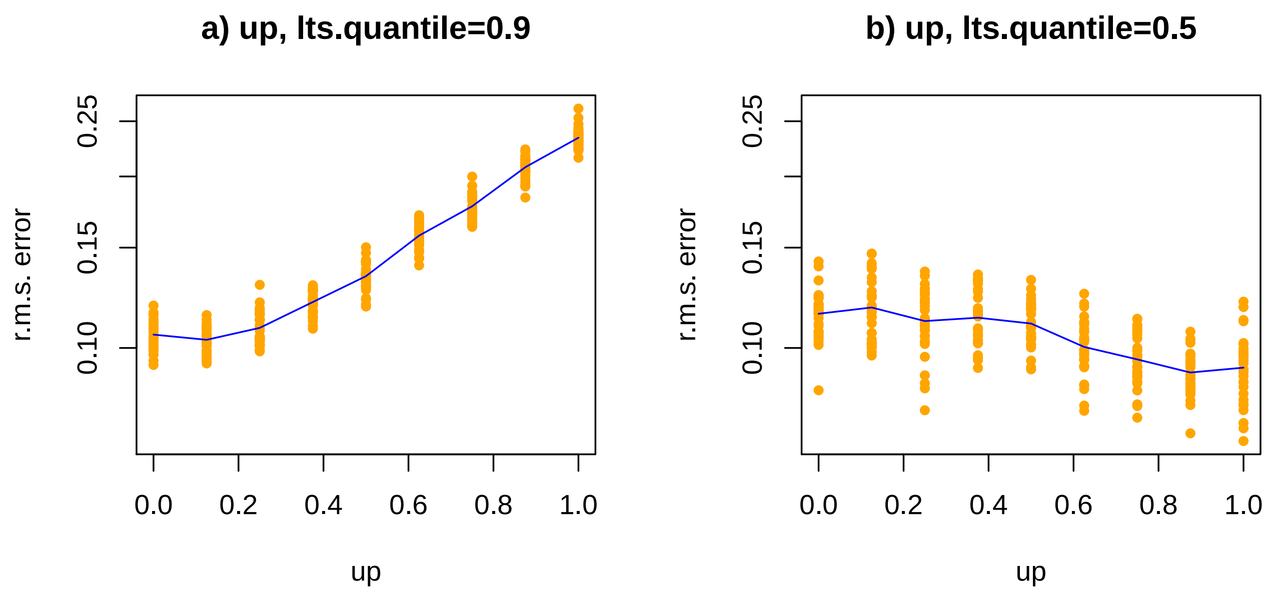

Figure 4: Estimation error as a function of the fraction of up-regulated genes, for two different settings of `lts.quantile`; see Section 6.

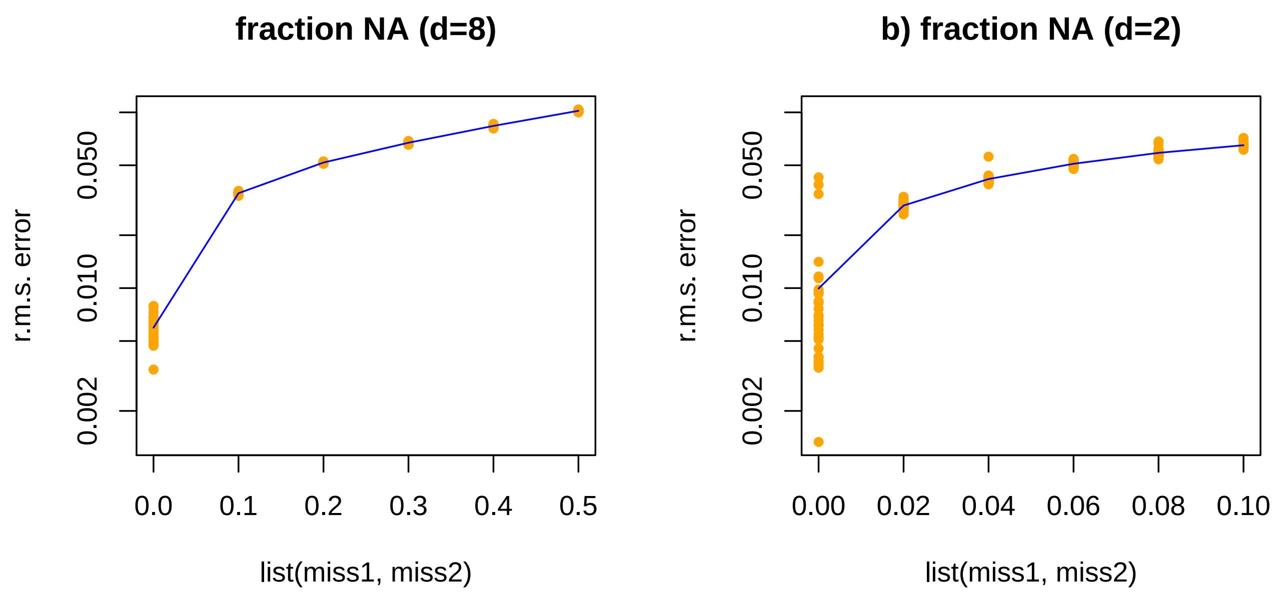

Figure 5: Estimation error as a function of the fraction of missing data points, see Section 7.

```
> toLatex(sessionInfo())
```

- R version 2.9.0 Under development (unstable) (2008-11-29 r47029), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=it_IT.UTF-8;LC_NUMERIC=C;LC_TIME=it_IT.UTF-8;LC_COLLATE=it_IT.UTF-8;LC_MONETARY=C;LC_MESSAGES=it_IT.UTF-8;LC_PAPER=it_IT.UTF-8;LC_NAME=C;LC_ADDRESS=C;LC_TELEPHONE=C;LC_MEASUREMENT=it_IT.UTF-8;LC_IDENTIFICATION=C`
- Base packages: base, datasets, graphics, grDevices, methods, stats, tools, utils
- Other packages: affy 1.21.0, Biobase 2.3.3, fortunes 1.3-5, lattice 0.17-17, limma 2.17.3, vsn 3.10.2
- Loaded via a namespace (and not attached): affyio 1.11.2, grid 2.9.0, preprocessCore 1.5.2

Table 1: The output of `sessionInfo` on the build system after running this vignette.

8 Incremental normalization

First, let's simulate a dataset with 10000 features, 12 arrays, and no differentially expressed genes (in order to be able to look at the ML estimates rather than their robustified modifications).

```
> dat = sagmbSimulateData(n=10000, d=12, de=0, nrstrata=1, miss=0, log2scale=TRUE)
> v  = new("vsn", mu=dat$mu, sigsq=dat$sigsq)
> fit = vsn2(dat$y, lts.quantile=1, verbose=FALSE)
```

`fit` contains the maximum profile likelihood estimate of the *vsn* model. Then we use the *incremental mode* of *vsn* to estimate, in turn for each array individually, the parameters. The results are shown in Figure 6.

```
> parRef = array(as.numeric(NA), dim=dim(coef(fit)))
> for(j in seq_len(ncol(dat$y))) {
+   vj = vsn2(dat$y[,j], reference=v, lts.quantile=1, verbose=FALSE)
+   parRef[,j,] = coef(vj)
+ }
```

```
> makeFig("figincr", 2, 2, {
+ par(mfcol=c(2,2))
+ for(k in 1:2) {
+   plot(dat$coefficients[1,,k], parRef[1,,k], pch=16, xlab="True", ylab="Reference fit",
+     main=c("offset", "factor")[k])
+   abline(a=0, b=1, col="orange")
+   plot(coefficients(fit)[1,,k], parRef[1,,k], pch=16, xlab="Profile Likelihood fit", ylab="Refe
+     main=c("offset", "factor")[k])
+   abline(a=0, b=1, col="orange")
+ }
+ })
```

References

[1] W. Huber, A. von Heydebreck, H. Sültmann, A. Poustka, and M. Vingron. Parameter estimation for the calibration and variance stabilization of microarray data. *Statistical Applications in Genetics and Molecular Biology*, Vol. 2: No. 1, Article 3, 2003. http://www.bepress.com/sagmb/vol2/iss1/art3

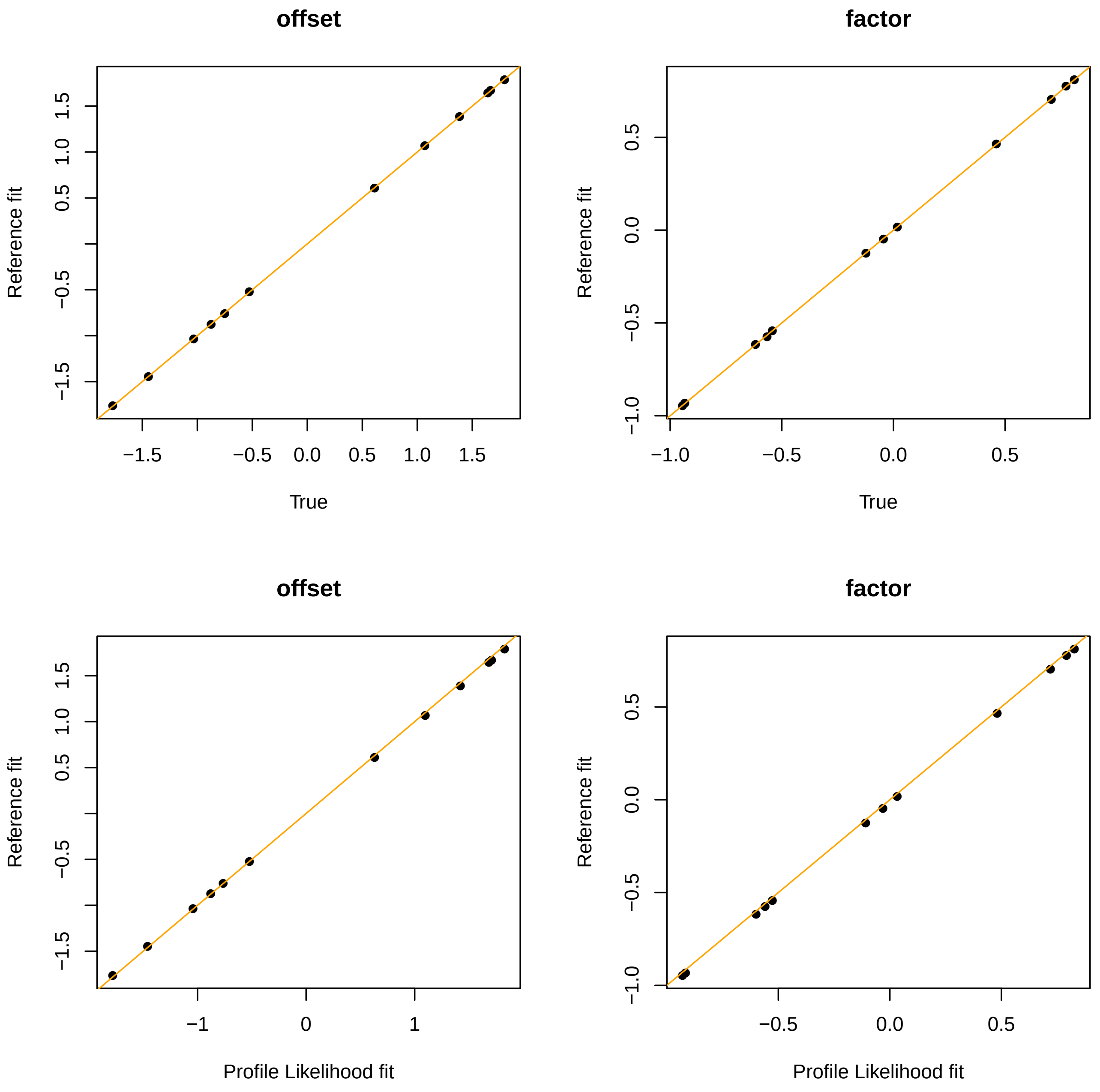

Figure 6: Comparison of parameters fitted from incremental normalisation (y-axis) with true parameters (x-axis, upper row) and with parameters fitted from joint profile-likelihood normalisation (x-axis, lower row); see Section 8.