# The xHMM Standardization Procedure Resolved

The xHMM paper has imprecise notation, text and figures, which makes it hard to determine exactly what the program is doing. This difficulty is compounded because the program is written in obfuscated C++. To obtain unambiguous clarity about what the program is doing, this document indepenedently reproduces the calculations performed by their example workflow using simple R commands instead of xHMM's C++ program and compares the results to verify they are the same.

The clarifications achieved are:

- The PCA done by xHMM does dimension reduction of the Exon Ttargets, not the Subjects.
- xHMM frequently refers to "Principal Components" (which is the data after transformation) when they actually mean "Principal Directions/Eigenvectors" (which is the rotation used to obtain the principal components).
- The PCA is performed after centering the data (subtract mean), but does not scale the data (divide std. dev.)
- Their equation 1 is not sensible, but the procedure it is meant to describe is.
- The PCA decomposition performed by xHMM is equivalent to the SVD decomposition performed by CoNIFER, only the later scales the data by the standard deviation while the former does not.

---

### Reproducing the xHMM Transformations

To verify equivalence with xHMM and reproduce their transformations, I recreate each step in their pipeline with some simple R code in what follows.

### Filtering Data

The xHMM program has a tutorial which describes how to run a sample dataset starting with a depth of coverage matrix and performing a series of commands.

The first command loads the coverage array and filters it out removing a few exon targets for low coverage and for being $< 10$ bp in size. The xHMM command to do this is:

```
# Filters samples and targets and then mean-centers the targets:
xhmm --matrix -r ./DATA.RD.txt --centerData --centerType target \
-o ./DATA.filtered_centered.RD.txt \
--outputExcludedTargets ./DATA.filtered_centered.RD.txt.filtered_targets.txt \
--outputExcludedSamples ./DATA.filtered_centered.RD.txt.filtered_samples.txt \
--minTargetSize 10 --maxTargetSize 10000 \
--minMeanTargetRD 10 --maxMeanTargetRD 500 \
--minMeanSampleRD 25 --maxMeanSampleRD 200 \
--maxSdSampleRD 150
```

The equivalent R code is:

```
# Load up the coverage data (30 samples by 301 exon targets)
d = read.delim("DATA.RD.txt")
# Trim off the sample name
d2 = d[,-1]
# Remove intervals < 10 in size
intervalLength = function(x) {
  interval = strsplit(x, "\\.")[[1]]
  as.integer(interval[3]) - as.integer(interval[2]) + 1
```

```
}
lens = sapply(colnames(d2), intervalLength)
d2 = d2[, lens >= 10]
# Remove those with mean < 10
d2 = d2[, apply(d2, 2, mean) >= 10]
```

### Centering data

Next xHMM centers the data by subtracting the mean value of target coverage from each variable. It is not clear why they don't also scale the variables by the variance here as is typical in PCA. Coverage should be roughly Poisson and so the variance of coverage should scale with the mean coverage so this transformation would seem relevant.

```
xhmm --matrix -r ./DATA.RD.txt --centerData --centerType target \
    -o ./DATA.filtered_centered.RD.txt --outputExcludedTargets \
    ./DATA.filtered_centered.RD.txt.filtered_targets.txt \
    --outputExcludedSamples ./DATA.filtered_centered.RD.txt.filtered_samples.txt \
    --minTargetSize 10 --maxTargetSize 10000 --minMeanTargetRD 10 \
    --maxMeanTargetRD 500 --minMeanSampleRD 25 --maxMeanSampleRD 200 --maxSdSampleRD 150
```

In the R code below we center the data in the same way and show our results are essentially equivalent to theirs.

```
# Subtracting mean by not dividing by SD
d3 = apply(d2, 2, scale, scale = FALSE)
# Load the xhmm produced centering and remove sample name
hcd = read.delim("DATA.filtered_centered.RD.txt")
hcd = hcd[,-1]
# Now show that the maximum difference is a small epsilon, we have
# the same values here
max(d3 - hcd)
```

```
## [1] 3.333383e-09
```

### Performing PCA

xHMM performs PCA on the centered read depth matrix $\mathbf{X}$ using an SVD transform, $\mathbf{X} = \mathbf{UDV^T}$, where $\mathbf{X}$ in this case in the samples (rows) by exon targets (columns) data matrix.

```
xhmm --PCA -r ./DATA.filtered_centered.RD.txt --PCAfiles ./DATA.RD_PCA
```

It then outputs each of the matrices from the decomposition into a separate file.

| Matrix | File Suffix |
|---|---|
| $\mathbf{U^T}$ | PC_LOADINGS.txt |
| $\mathbf{D}$ | PC_SD.txt |
| $\mathbf{V^T}$ | PC.txt |

Note that what they call loadings is a bit of a misnomer as the loadings are usually the principal axes scaled by the sqrt of the eigenvector, e.g. $\mathbf{VD^{1/2}}$, per discussion here. They also output $\mathbf{V^T}$ directly instead of $\mathbf{V}$. xhmm uses the LAPACK's `dgesdd` function under the hood, and then transposes the $\mathbf{V^t}$ matrix before returning it.

Also, note that because the number of principal components is limited by the rows and we have mean centered

each column, we only have $NUMROWS - 1$ eigenvalues $> 0$, though they write the paper as though all $NUMROW$ eignvectors could be valid.

To verify we can reproduce their decomposition we will show their output files are equivalent to the matrices obtained in R:

```
# La.svd() uses dgesdd and returns V^T, while svd() calls zgesdd and returns v
r_decomp = La.svd(d3)

# Let's check V matrix matches
xmm_v = read.delim("DATA.RD_PCA.PC.txt")[,-1]
# Note that because the data is mean-centered, we only have (Samples - 1)
# meaningful eigenvectors
difs = sapply(1:(nrow(xmm_v)-1), function(i) max(xmm_v[i,] - r_decomp$vt[i,]))
max(xmm_v[-nrow(xmm_v), ] - r_decomp$vt[-nrow(r_decomp$vt),])
```

```
## [1] 4.994011e-07
```

```
# Check that D matches
xmm_d = read.delim("DATA.RD_PCA.PC_SD.txt")[,-1]
max(xmm_d - r_decomp$d)
```

```
## [1] 3.771004e-07
```

```
# And that the U matrix matches
xmm_u = read.delim("DATA.RD_PCA.PC_LOADINGS.txt")[,-1]
max(xmm_u - t(r_decomp$u))
```

```
## [1] 4.986854e-07
```

We can also confirm that the principal components returned match what would be obtained by running a more standard PC analysis. Note however, that you can't actually do "R-mode" PCA with the `princomp` PCA function in R (which requires more rows than columns), whereas you can do the so called "Q-mode"" (more columns than rows).

```
# Note that princomp will not work
#princomp(d3) <- throws error "`princomp` can only be used with more units than variables"

# But we can use prcomp
pc = prcomp(d3)

# Let's compare our rotation matrix to the rotation matrix xHMM found
max(pc$rotation[,-ncol(pc$rotation)] - t(xmm_v[-nrow(xmm_v),]))
```

```
## [1] 4.997539e-07
```

### Variable standardization

Now how does xHMM standarize the read depth values? According to the paper, it first selects $K$ principal components to remove. At the command line this is accomplished as:

```
# Normalizes mean-centered data using PCA information:
xhmm --normalize -r ./DATA.filtered_centered.RD.txt --PCAfiles ./DATA.RD_PCA \
--normalizeOutput ./DATA.PCA_normalized.txt \
--PCnormalizeMethod PVE_mean --PVE_mean_factor 0.7
```

And the procedure taking place is described in the manuscript as:

To remove these $K$ components, we subtract them out from the matrix of all samples' read-depth matrix, $\mathbf{R}^*$:

$$\mathbf{R}^* = \mathbf{R} - \sum_{\mathbf{i=1}}^{\mathbf{K}} \mathbf{c_i c_i^T R}$$

where $c_i$ is the ith principal component of $\mathbf{R}$ to be normalized out of the depth signal.

However, this equation is clearly incorrect and not what is occurring. What they appear to be doing is similar to be mocking the equation for subtracting the eigenvector of a matrix, as described in section 11.1.3 of Mining of Massived Datasets.

However, several things seem wrong in the above equation. I have tried to implement it from several possible perspectives, and none of them holds. I am not sure what they were up to.

Here is a more sensible intrepretation of what they are doing: They are using an SVD decomposition to select $K$ principal components and principal directions which will be used to create the best possible reconstruction of the original data matrix that is of rank $K$. They then subtract the original matrix from this lower dimensional reconstruction to create a matrix of "residual" data for analysis.

More formally, multiplying the first $k$ PCs by the corresponding principal axes $\mathbf{V}_k^T$ yields $\mathbf{R_k} = \mathbf{R V_k V_k^T}$, a lower dimensional approximation of $R$ which can be subtracted to yield $\mathbf{R}^* = \mathbf{R} - \mathbf{R_K}$, which is how they should have written their equation in my mind.

I'll now demonstrate this is equivalent to what they do.

```
# Load in their post transformation data.
rn = read.delim("DATA.PCA_normalized.txt")[,-1]
r_rn = d3 # This will be our transformed data
# In the C++ code they do not do matrix operations, so let's first copy their procedure here to verify
for (j in 1:nrow(d3)) {
  data = as.vector(d3[j,]) # This is jth sample subject data, 1 x Target number in size
  xmm_rn = as.numeric(rn[j,])
  for(i in 1:3) {
    pc = as.numeric(xmm_v[i,]) # This is the projection/eigenvector to take the condense the ith exon t
    dl = sum(pc*data) # Principal component for this sample when projected down (not scaled by the eige
    data = data - pc * dl
  }
  r_rn[j,] = data
}
# verify that our code matches theirs
max(r_rn - rn)
```

```
## [1] 8.686546e-09
```

```
# Now let's do that with matrix operations instead of a nasty for loop.
pcs = d3 %*% t(r_decomp$vt[1:3,]) # Get first three principal components
recon = pcs %*% r_decomp$vt[1:3,] # Now reconstruct the full matrix using the 3 lower dimensions
r_rn2 = d3 - recon
# Verify equivalence
max(r_rn2 - rn) # Some small numeric issues.
```

```
## [1] 0.0008039004
```

So we can regenerate their result. However, going through their original equation though, we can see that it doesn't match the results produced by comparing their output values to the values calculated by their equation.

```
# Verify the equation as given doesn't work
rdrop = matrix(0, ncol=266, nrow = 30) # matrix to hold summation of c_i * c_I^t * R
for (j in 1:3) {
```

```
  pc = matrix(pcs[,j], nrow = 30)
  rdrop = rdrop + pc %*% t(pc) %*% d3
}
f = 1:4
head(rn[f,f])
```

```
##   X22.16449025.16449223 X22.16449225.16449423 X22.16449425.16449804
## 1             12.056110             2.9491336             24.256702
## 2              2.130515             0.5548125             -5.504716
## 3             -6.324626            -7.6459230            -10.956043
## 4             -5.403961            -8.9739065              2.959190
##   X22.17071768.17071966
## 1            12.9745072
## 2             0.8531382
## 3             3.0999912
## 4           -10.8352379
```

```
head((d3 - rdrop)[f,f])
```

```
##      X22.16449025.16449223 X22.16449225.16449423 X22.16449425.16449804
## [1,]              -51662101              16325794             -30806232
## [2,]              -50656334              12112716             -32885952
## [3,]               68625015             -64751880              19828525
## [4,]               66891160             -67084022              16263450
##      X22.17071768.17071966
## [1,]             359047065
## [2,]             355116704
## [3,]            -598798069
## [4,]            -574447130
```

```
# And similarly an iterative version doesn't work.
rn4 = d3 # matrix to hold summation of c_i * c_I^t * R
for (j in 1:3) {
  pc = matrix(pcs[,j], nrow = 30)
  rn4 = rn4 - pc %*% t(pc) %*% rn4
}
head(rn[f,f])
```

```
##   X22.16449025.16449223 X22.16449225.16449423 X22.16449425.16449804
## 1             12.056110             2.9491336             24.256702
## 2              2.130515             0.5548125             -5.504716
## 3             -6.324626            -7.6459230            -10.956043
## 4             -5.403961            -8.9739065              2.959190
##   X22.17071768.17071966
## 1            12.9745072
## 2             0.8531382
## 3             3.0999912
## 4           -10.8352379
```

```
head((d3 - rdrop)[f,f])
```

```
##      X22.16449025.16449223 X22.16449225.16449423 X22.16449425.16449804
## [1,]              -51662101              16325794             -30806232
## [2,]              -50656334              12112716             -32885952
## [3,]               68625015             -64751880              19828525
## [4,]               66891160             -67084022              16263450
```

```
##      X22.17071768.17071966
## [1,]             359047065
## [2,]             355116704
## [3,]            -598798069
## [4,]            -574447130
```

## The equivalence of xHMM's "PCA" technique with CoNIFER's "SVD" technique

Confusingly, the CNV Review Paper describes xHMM as using a "Using singular value decomposition to normalize copy number and avoiding batch bias by integrating multiple samples" whereas it describes xHMM as "Uses principal component analysis to normalize copy number", which makes it sound as though they are doing different things.

This is a false distinction. Both CoNIFER and xHMM use SVD (which is one form of PCA) to normalize the data, though how they describe the mathematically equivalent results in a rather different fashion. Starting with the exon-by-sample matrix ($\mathbf{X}$), CoNIFER describes its procedure as.

> Using SVD, we decomposed $\mathbf{X}$ into three matrices:
>
> $\mathbf{X} = \mathbf{USV^T}$
>
> The values in the $\mathbf{S}$ matrix, known as "singular values", can be used to examine the relative amount of contributed variance from each component. We used a plot of these singular values, known as a "scree plot" to identify this experimental noise. Our analysis reveals that the first 10–15 components disproportionally contribute to the variance of the data (Fig. S2). Given that we expect biological variation, in the form of rare CNVs as well as common CNPs, to be a minor contributor to the overall variance of the exon-by-sample matrix X, we formulated the basis of our algorithm by eliminating these strongest components. We selected the number of components for elimination based on the inflection point of the scree plot. Algorithmically, in order to remove the strongest k components, we set $S_1 ... S_k$ to zero to form $\mathbf{S'}$,and then recalculate $\mathbf{X}$ as the dot product of $\mathbf{U}$, $\mathbf{S'}$ and $\mathbf{V^T}$.

So both methods are looking for a matrix of residuals formed after subtracting out the projection of a lower order approximation of the full matrix. xHMM does it by calculating subtracting the matrix formed by creating an approximation matrix from the first $K$ components from the full matrix to get the "residuals", while CoNIFER does it by directly calculating these "residuals" - which are equivalent to an approximate matrix formed using only using the last $N - K$ components. The result is exactly the same as is shown below:

```
# Calculate approximate matrix with top three and subtract from full
pcs = d3 %*% t(r_decomp$vt[1:3,]) # Get first three principal components
recon = pcs %*% r_decomp$vt[1:3,] # Now reconstruct the full matrix using the 3 lower dimensions
xHMM = d3 - recon # full matrix minus 3 dimensions

# Calculate approximate matrix with bottom 27 and get the same result.
totRows = 4:nrow(r_decomp$vt)
pcs = d3 %*% t(r_decomp$vt[totRows,]) # Get last 27 principal components
CoNIFER = pcs %*% r_decomp$vt[totRows,] # Reconstruction using the bottom 27 dimensions
max(xHMM - CoNIFER)
```

```
## [1] 8.90843e-13
```

Although the matrix conifer uses is the transpose of the one xHMM uses, as one standardizes the columns and the other standardizes the rows, the manipulation becomes equivalent. The only meaningful distinction is that xHMM centers the data, while CoNIFER both centers and scales the data. Additionally, CoNIFER does not use read depths directly, but rather RPKM values (very similar to read depths) as described in the paper.