SBF Math vignette

Amal Thomas

Contents

1 Background

Joint matrix factorization facilitates the comparison of expression profiles from different species without using gene mapping. Transforming gene expression profiles into reduced eigengene space using singular value decomposition (SVD) has been shown to capture meaningful biological information (Alter, Brown, and Botstein 2000). Tamayo et al. (2007) used a non-negative matrix factorization approach to learn a low-dimensional approximation of the microarray expression datasets and used the reduced space for comparisons. Matrix factorization-based methods are commonly used for gene expression analysis (Alter, Brown, and Botstein 2000; Tamayo et al. 2007). An orthology independent matrix factorization framework based on generalized singular value decomposition [GSVD; Van Loan (1976)] was used by Alter, Brown, and Botstein (2003) to compare gene-expression profiles from two species. This framework was later extended to develop higher-order generalized singular value decomposition (HO GSVD) to analyze data from more than two species (Ponnapalli et al. 2011). Using cell-cycle gene expression datasets, these approaches have shown examples of genes with highly conserved sequences across species but with significantly different cell-cycle peak times. Although these methods have shown the potential advantages of orthology-independent comparisons, the steps involved in estimating the shared factor and comparing the expression profiles using these methods require complex procedures. When estimating the shared factor, the pairwise quotients and their arithmetic mean involve the computation of inverses. As a result, the biological interpretation of the shared factor is difficult in HO GSVD. Similarly, to place new datasets to the space defined by the shared factor requires the computation of generalized inverses. Moreover, the independence of the columns of the species-specific factors and the shared factor is not guaranteed, making it challenging to differentiate the contribution of genes/features across different dimensions of the shared factor. These limitations restrict the application of these methods in cross-species studies.

This study developed a joint diagonalization approach called Orthogonal Shared Basis Factorization (OSBF) for cross-species expression comparisons. This approach extends the exact factorization approach we developed called shared basis factorization (SBF). We implemented both algorithms in the SBF package. The details and example cases of the two methods are shown in the following sections.

2 Shared basis factorization

Consider a set of real matrices $D_i \in \mathbb{R}^{m_i \times n}$ $(i = 1, \ldots, k)$ with full column rank. We define shared basis factorization (SBF) as

$$\begin{aligned} D_1 &= U_1 \Delta_1 V^T, \\ D_2 &= U_2 \Delta_2 V^T, \\ &\vdots \\ D_k &= U_N \Delta_k V^T. \end{aligned}$$

Here each $U_i \in \mathbb{R}^{m_i \times n}$ is a dataset-specific left basis matrix, each $\Delta_i \in \mathbb{R}^{n \times n}$ is a diagonal matrix with positive values δ_{ij}, and V is a a shared orthogonal matrix.

2.1 Estimating the shared right basis matrix

Let M be the scaled sum of the $D_i^T D_i$. We define M is defined as

$$M = \frac{\sum_{i=1}^{k} D_i^T D_i / w_i}{\alpha}.$$

The scaling factor w_i is the total variance explained by the column vectors of D_i, and α is the inverse sum of the total variance of D_i, for $i = 1, \ldots, k$. The weights w_i and α are defined as

$$\begin{aligned} w_i &= \sum_{j=1}^{n} {\sigma_{jj}^2}^{(i)} \text{ and} \\ \alpha &= \sum_{i=1}^{k} \frac{1}{\sum_{j=1}^{n} {\sigma_{jj}^2}^{(i)}}. \end{aligned}$$

Here $\sum_{j=1}^{n} {\sigma_{jj}^2}^{(i)} = \mathrm{tr}(D_i^T D_i)$. Using the w_i and α, individual $D_i^T D_i$ are standardized. If all the variances are equal, M becomes the arithmetic mean of the sum of $D_i^T D_i$. The shared right basis matrix V is then determined from the eigenvalue decomposition of M, where $M = V \Theta V^T$. The shared right basis matrix V is an orthogonal matrix as M is symmetric. Given V, we compute U_i and Δ_i by solving the linear system $D_i V = U_i \Delta_i = L_i$. By normalizing the columns of L_i, we have $\delta_{ij} = \|l_{ij}\|$ and $\Delta_i = \mathrm{diag}(\delta_{i1}, \ldots, \delta_{in})$.

3 Orthogonal shared basis factorization

Consider a set of matrices $D_i \in \mathbb{R}^{m_i \times n}$ $(i = 1, \ldots, k)$, each with full column rank. We define orthogonal shared basis factorization (OSBF) as

$$\begin{aligned} D_1 &= U_1 \Delta_1 V^T + \epsilon_1, \\ D_2 &= U_2 \Delta_2 V^T + \epsilon_2, \\ &\vdots \\ D_k &= U_k \Delta_k V^T + \epsilon_k. \end{aligned}$$

Each $U_i \in \mathbb{R}^{m_i \times n}$ is a D_i specific left basis matrix with **orthonormal columns** ($U_i^T U_i = I$) and $\Delta_i \in \mathbb{R}^{n \times n}$ is a diagonal matrix with positive values. The right basis matrix $V \in \mathbb{R}^{n \times n}$ is an orthogonal matrix and identical in all the k matrix factorizations. We use an alternate least square algorithm to minimize the total factorization error: $\sum_{i=1}^{k} \|\epsilon_i\|^2{}_F = \sum_{i=1}^{k} \|D_i - U_i \Delta_i V^T\|^2{}_F$ The estimation of the common space in cross-species gene expression analysis is explained in the `GeneExpressionAnalysis` vignette.

4 Use cases

4.1 SBF examples

```
# load SBF package
library(SBF)
```

Let us create some random matrices using the `createRandomMatrices` function from the SBF package. We will create four matrices, each with three columns with rows varying from 4 to 6.

```
set.seed(1231)
mymat <- createRandomMatrices(n = 4, ncols = 3, nrows = 4:6)
sapply(mymat, dim)
#>      mat1 mat2 mat3 mat4
#> [1,]    5    6    4    5
#> [2,]    3    3    3    3
```

The rank of each of these matrices:

```
sapply(mymat, function(x) {
  qr(x)$rank
  })
#> mat1 mat2 mat3 mat4
#>    3    3    3    3
```

Let us compute SBF using different approaches:

- Estimate V using the sum of $D_i^T D_i / k$
- Estimate V using the sum of $D_i^T D_i / k$ with inverse variance weighting
- Estimate V using the inter-sample correlation

```
sbf <- SBF(matrix_list = mymat)
sbf_inv <- SBF(matrix_list = mymat, weighted = TRUE)
sbf_cor <- SBF(matrix_list = mymat, transform_matrix = TRUE)
```

When the D_i matrices are transformed to compute inter-sample correlation, we do not need to scale it using inverse-variance weighting anymore. We recommend using inverse variance weights, giving a more robust estimate of V when noisy datasets are present. We estimate V using inter-sample correlation when dealing with gene expression data sets.

The `?SBF` help function shows all arguments for the SBF function. Let us inspect the output of the `SBF` call.

```
names(sbf)
#> [1] "v"      "lambda" "u"      "delta"  "m"
```

`sbf$u`, `sbf$v`, and `sbf$delta` correspond to the estimated left basis matrix, shared right basis matrix, and diagonal matrices.

The estimated V has a dimension of $n \times n$, where n is the number of columns in D_i.

```
sbf$v
#>           [,1]       [,2]       [,3]
```

```
#> [1,] 0.4793022  0.8669998  0.1363110
#> [2,] 0.7027353 -0.2860780 -0.6514004
#> [3,] 0.5257684 -0.4080082  0.7463892
```

The delta values for each matrix for the three cases are shown below.

```
printDelta <- function(l) {
  for (eachmat in names(l$delta)) {
  cat(eachmat, ":", l$delta[[eachmat]], "\n")
  }
}
cat("sbf\n");printDelta(sbf)
#> sbf
#> mat1 : 205.4915 29.6746 71.43295
#> mat2 : 206.5816 71.72548 55.682
#> mat3 : 189.9136 52.6758 42.36825
#> mat4 : 192.6911 80.22868 58.57913
cat("sbf_inv\n");printDelta(sbf_inv)
#> sbf_inv
#> mat1 : 205.5109 22.4888 73.95623
#> mat2 : 206.5963 77.00352 48.05638
#> mat3 : 189.8719 58.60394 33.92988
#> mat4 : 192.6942 72.41955 67.98802
cat("sbf_cor\n");printDelta(sbf_cor)
#> sbf_cor
#> mat1 : 200.4197 44.25134 77.99852
#> mat2 : 199.8621 80.34494 67.23723
#> mat3 : 176.1738 76.95449 60.64485
#> mat4 : 185.4579 67.51833 89.69184
```

The $V \in R^{n \times n}$ estimated in SBF is orthogonal. So $V^T V = V V^T = I$.

```
zapsmall(t(sbf$v) %*% sbf$v)
#>      [,1] [,2] [,3]
#> [1,]    1    0    0
#> [2,]    0    1    0
#> [3,]    0    0    1
```

The estimated V is an invertible matrix.

```
qr(sbf$v)$rank
#> [1] 3
```

The U_i matrices estimated in the SBF do not have orthonormal columns. Let us explore that.

```
sapply(sbf$u, dim)
#>      mat1 mat2 mat3 mat4
#> [1,]    5    6    4    5
#> [2,]    3    3    3    3
```

Let us take the first matrix $U_i \in R^{m_i \times n}$ to check this. For this matrix, $U_i^T U_i$ will be $n \times n$ matrix where $n = 3$.

```
t(sbf$u[[names(sbf$u)[1]]]) %*% sbf$u[[names(sbf$u)[1]]]
#>              [,1]        [,2]       [,3]
#> [1,]  1.00000000 -0.07071457  0.1405487
#> [2,] -0.07071457  1.00000000 -0.6201468
```

```
#> [3,]  0.14054867 -0.62014676  1.0000000
```

The estimated M matrix is stored `sbf$m` and `sbf$lambda` gives the eigenvalues in the eigenvalue decomposition ($M = V\Theta V^T$).

```
sbf$lambda
#> [1] 39524.940  3809.127  3357.434
```

SBF is an exact factorization. Let compute the factorization error for the three cases using `calcDecompError` function.

```
calcDecompError(mymat, sbf$u, sbf$delta, sbf$v)
#> [1] 1.851693e-26
calcDecompError(mymat, sbf_inv$u, sbf_inv$delta, sbf_inv$v)
#> [1] 1.894292e-26
calcDecompError(mymat, sbf_cor$u, sbf_cor$delta, sbf_cor$v)
#> [1] 2.835482e-26
```

The errors are close to zero in all three cases.

4.1.1 Adding new dataset

The total column variance of matrix 1-4 in `mymat` is nearly in the same range.

```
sapply(mymat, function(x) sum(diag(cov(x))))
#>    mat1    mat2    mat3    mat4
#> 2076.80 2273.50 2375.25 2860.40
```

Now, let us create two new matrix lists containing the `mymat`. We will add a dataset with a similar variance to the first list and a high variance to the second.

```
mat5 <- matrix(c(130, 183, 62, 97, 147, 94, 102, 192, 19), byrow = TRUE,
                     nrow = 3, ncol = 3)
mat5_highvar <- matrix(c(406, 319, 388, 292, 473, 287, 390, 533, 452),
                       byrow = TRUE, nrow = 3, ncol = 3)

mymat_new <- mymat
mymat_new[["mat5"]] <- mat5
sapply(mymat_new, function(x) sum(diag(cov(x))))
#>     mat1     mat2     mat3     mat4     mat5
#> 2076.800 2273.500 2375.250 2860.400 2299.667
mymat_new_noisy <- mymat
mymat_new_noisy[["mat5"]] <- mat5_highvar
sapply(mymat_new_noisy, function(x) sum(diag(cov(x))))
#>     mat1     mat2     mat3     mat4     mat5
#>  2076.80  2273.50  2375.25  2860.40 22915.00
```

Let us compute SBF with the new datasets.

```
sbf_new <- SBF(matrix_list = mymat_new)
sbf_inv_new <- SBF(matrix_list = mymat_new, weighted = TRUE)


sbf_new_noisy <- SBF(matrix_list = mymat_new_noisy)
sbf_inv_new_noisy <- SBF(matrix_list = mymat_new_noisy, weighted = TRUE)
```

Let us take the newly estimated values U_i, Δ_i, and V for the four initial matrices in `mymat`. We will then compare the decomposition error for the two cases with and without inverse variance weighting.

```
e1 <- calcDecompError(mymat, sbf_new$u[1:4], sbf_new$delta[1:4], sbf_new$v)
e2 <- calcDecompError(mymat, sbf_new_noisy$u[1:4], sbf_new_noisy$delta[1:4],
                      sbf_new_noisy$v)
e2 / e1
#> [1] 3.887268
```

```
e3 <- calcDecompError(mymat, sbf_inv_new$u[1:4], sbf_inv_new$delta[1:4],
                      sbf_inv_new$v)
e4 <- calcDecompError(mymat, sbf_inv_new_noisy$u[1:4],
                      sbf_inv_new_noisy$delta[1:4], sbf_inv_new_noisy$v)
e4 / e3
#> [1] 1.657059
```

With inverse variance weighting, the deviation is smaller.

4.2 OSBF examples

```
set.seed(1231)
mymat <- createRandomMatrices(n = 4, ncols = 3, nrows = 4:6)
sapply(mymat, dim)
#>      mat1 mat2 mat3 mat4
#> [1,]    5    6    4    5
#> [2,]    3    3    3    3
```

Now let us compute Orthogonal-SBF for the same datasets for the following three cases.

- OSBF
- OSBF with inverse variance weighting
- OSBF with inter-sample correlation

Here the OSBF is invoked without minimizing the factorization error (`minimizeError=FALSE`).

```
osbf <- SBF(matrix_list = mymat, orthogonal = TRUE,
            minimizeError = FALSE)
#>
#> OSBF with no optimization
osbf_inv <- SBF(matrix_list = mymat, weighted = TRUE, orthogonal = TRUE,
                minimizeError = FALSE)
#>
#> OSBF with no optimization
osbf_cor <- SBF(matrix_list = mymat, orthogonal = TRUE,
                transform_matrix = TRUE, minimizeError = FALSE)
#>
#> OSBF with no optimization
```

```
names(osbf)
#> [1] "v"        "lambda"   "u"        "u_ortho" "delta"    "m"        "error"
```

OSBF is not an exact factorization and has decomposition error.

```
# decomposition error
osbf$error
#> [1] 2329.73
osbf_inv$error
#> [1] 1651.901
osbf_cor$error
#> [1] 14045.99
```

- `osbf$u_ortho` is the matrix with orthonormal columns that is closet to the exact U in SBF
- `osbf$v` is orthogonal.

```
zapsmall(t(osbf$u_ortho[[names(osbf$u_ortho)[1]]]) %*%
           osbf$u_ortho[[names(osbf$u_ortho)[1]]])
#>      [,1] [,2] [,3]
#> [1,]    1    0    0
#> [2,]    0    1    0
#> [3,]    0    0    1
```

```
zapsmall(t(osbf$v) %*% osbf$v)
#>      [,1] [,2] [,3]
#> [1,]    1    0    0
#> [2,]    0    1    0
#> [3,]    0    0    1
```

5 Minimizing OSBF error

We use an alternate least square algorithm to minimize the total factorization error: $\sum_{i=1}^{k} \|\epsilon_i\|^2{}_F = \sum_{i=1}^{k} \|D_i - U_i \Delta_i V^T\|^2{}_F$. Our algorithm determines the optimal learning rate in each update step and converges to a local optimum (see additional file 2 in manuscript).

5.1 Examples

5.1.1 Minimizing error

Let us optimize the factorization error using the `optimizeFactorization` function for the three cases of OSBF computation. The `optimizeFactorization` is called setting `orthogonal = TRUE` and `minimizeError = TRUE` in the SBF function. The argument `minimizeError` is set to be TRUE by default. Depending upon the data matrix and initial values of U_i, Δ_i, and V, optimization could take some time.

```
set.seed(1231)
mymat <- createRandomMatrices(n = 4, ncols = 3, nrows = 4:6)
osbf <- SBF(matrix_list = mymat, orthogonal = TRUE)
#>
#> OSBF optimizing factorization error
osbf_inv <- SBF(matrix_list = mymat, weighted = TRUE, orthogonal = TRUE)
#>
#> OSBF optimizing factorization error
osbf_cor <- SBF(matrix_list = mymat, orthogonal = TRUE, transform_matrix = TRUE)
#>
#> OSBF optimizing factorization error
```

```
names(osbf)
#>  [1] "v"             "u"             "delta"         "error"
#>  [5] "error_pos"     "error_vec"     "v_start"       "lambda_start"
#>  [9] "u_start"       "u_ortho_start" "delta_start"   "m"
#> [13] "error_start"
```

- `osbf$u` is the optimized left basis matrices with orthonormal columns
- `osbf$v` is the optimized shared right basis matrix
- `osbf$delta` is the optimized delta matrices
- `osbf$error` gives the final decomposition error

```
# initial decomposition error
osbf$error_start
#> [1] 2329.73
osbf_inv$error_start
#> [1] 1651.901
osbf_cor$error_start
#> [1] 14045.99
```

This is the same error (OSBF with no optimization) we showed previously in the OSBF examples section. Now let us check the final decomposition error after optimization.

```
# final decomposition error
osbf$error
#> [1] 1411.555
osbf_inv$error
#> [1] 1411.555
osbf_cor$error
#> [1] 1411.555
```

After optimization, for all three OSBF factorizations, the final error is

```
#> same (up to 2 decimals). The final error is 1411.56
```

Independent of the initial values, if the optimization converges, we achieve the same decomposition error.

We can also compute the same optimization by independently calling the `optimizeFactorization`. For example,

```
myopt <- optimizeFactorization(mymat, osbf$u_ortho_start, osbf$delta_start,
                               osbf$v_start)
names(myopt)
#> [1] "u"         "v"         "d"         "error"     "error_pos" "error_vec"
```

```
myopt$error
#> [1] 1411.555
```

The number of iteration taken for optimizing and new factorization error:

```
cat("For osbf, # iteration =", osbf$error_pos, "final error =", osbf$error)
#> For osbf, # iteration = 220 final error = 1411.555
cat("For osbf inv, # iteration =", osbf_inv$error_pos, "final error =",
    osbf_inv$error)
#> For osbf inv, # iteration = 202 final error = 1411.555
cat("For osbf cor, # iteration =", osbf_cor$error_pos, "final error =",
    osbf_cor$error)
#> For osbf cor, # iteration = 196 final error = 1411.555
```

5.1.2 Using different initial values

```
set.seed(1231)
mymat <- createRandomMatrices(n = 4, ncols = 3, nrows = 4:6)
```

1. Let us initialize the `optimizeFactorization` function with a random orthogonal matrix and check the final optimization error. The V matrix estimated from the `mymat` matrix has a dimension of 3×3. First, we will create a random 3×3 matrix and obtain an orthogonal matrix based on this.

```
set.seed(111)
rand_mat <- createRandomMatrices(n = 1, ncols = 3, nrows = 3)
```

```
cat("\nRank is:", qr(rand_mat[[1]])$rank, "\n")
#>
#> Rank is: 3
dim(rand_mat[[1]])
#> [1] 3 3
```

Get an orthogonal V matrix using SVD. We will set V as the right basis matrix from the SVD.

```
mysvd <- svd(rand_mat[[1]])
randV <- mysvd$v
```

Now for this V, we will first compute U_i's and Δ_i for different D_i matrices in the `mymat`. We achieve this by solving the linear equations: $D_i = U_i \Delta_i V^T$ for $i = 1, \ldots, 4$. We then orthonormalize the columns of U_i using Proposition I.

```
# get Ui and Delta for this newV
out <- computeUDelta(mymat, randV)
names(out)
#> [1] "u"       "u_ortho" "d"       "d_ortho" "error"
```

The initial decomposition error is :

```
calcDecompError(mymat, out$u_ortho, out$d, randV)
#> [1] 22879.08
```

Now we will try to optimize using the new random V and corresponding U_i's and Δ_i's.

```
newopt <- optimizeFactorization(mymat, out$u_ortho, out$d, randV)
# Number of updates taken
newopt$error_pos
#> [1] 220
# New error
newopt$error
#> [1] 1411.555
```

We achieve the same factorization error (`1411.5550218`) after the `optimizeFactorization` function call.

2. Now, instead of the right basis matrix from the SVD, we will set V as the left basis matrix.

```
mysvd <- svd(rand_mat[[1]])
randV <- mysvd$u
dim(randV)
#> [1] 3 3
```

```
# get Ui and Delta for this newV
out <- computeUDelta(mymat, randV)
calcDecompError(mymat, out$u_ortho, out$d, randV)
#> [1] 13903.45
```

Now we will try to optimize with these matrices as our initial values.

```
newopt <- optimizeFactorization(mymat, out$u_ortho, out$d, randV)
# Number of updates taken
newopt$error_pos
#> [1] 283
# New error
newopt$error
#> [1] 1411.555
```

Again we get the same decomposition error after optimizing.

3. Instead of the initial value being an orthogonal matrix, we will initialize U_i's, Δ_i, and V with random matrices such that it does not guarantee

- orthogonal property for V and
- orthonormal columns for U_i's.

```
set.seed(111)
# new random v
newv <- createRandomMatrices(n = 1, ncols = 3, nrows = 3)[[1]]
# seed value
k <- 2392
newu <- newd <- list()
for (i in names(mymat)) {
  myrow <- nrow(mymat[[i]])
  mycol <- ncol(mymat[[i]])
  set.seed(k)
  # new random u_i
  newu[[i]] <- createRandomMatrices(n = 1, ncols = mycol, nrows = myrow)[[1]]
  set.seed(k * 2)
  # new random d_i
  newd[[i]] <- sample(1:1000, size = mycol)
  newmat <- newu[[i]] %*% diag(newd[[i]]) %*% t(newv)
  if (!qr(newmat)$rank == mycol)
    cat("\nNew matrix does not have full column rank")
  k <- k + 1
}
error <- calcDecompError(mymat, newu, newd, newv)
cat("\nInitial error = ", error, "\n")
#>
#> Initial error =  2.062531e+15
```

We see a very high factorization error because of the random initialization.

```
newopt <- optimizeFactorization(mymat, newu, newd, newv)
newopt$error_pos
#> [1] 142
newopt$error
#> [1] 1411.555
```

Again, we get the same factorization error after optimizing. Try changing the seed value and compare the results.

This shows that the iterative update procedure converges and achieves the same decomposition error regardless of the initial values.

5.1.3 Estimating SVD

We will further demonstrate the case for $k = 1$ when we have just one matrix. The `optimizeFactorization` function gives U_i's with orthonormal column, Δ_i a diagonal matrix, and an orthogonal V. If the function converges, the results should be identical to a standard SVD, except for the sign changes corresponding to U and V columns. So we will compare the results from the `optimizeFactorization` function with the standard SVD output. Let us generate one example matrix say `newmat`.

```
set.seed(171)
newmat <- createRandomMatrices(n = 1, ncols = 3, nrows = 3)
newmat
```

```
#> $mat1
#>      [,1] [,2] [,3]
#> [1,]   41   10    6
#> [2,]   64   85    8
#> [3,]   82   87   57
```

1. We will estimate the SVD of `newmat` using our iterative update function by setting the initial values to be an identity matrix.

```
newu <- newd <- list()
newu[["mat1"]]  <- diag(3)
newd[["mat1"]] <- diag(newu[["mat1"]])
newu
#> $mat1
#>      [,1] [,2] [,3]
#> [1,]    1    0    0
#> [2,]    0    1    0
#> [3,]    0    0    1
newd
#> $mat1
#> [1] 1 1 1
```

The factorization error when initializing using an identity matrix:

```
calcDecompError(newmat, newu, newd, diag(3))
#> [1] 30381
```

Let us optimize.

```
opt_new <- optimizeFactorization(newmat, newu, newd, diag(3))
cat("\n # of updates:", opt_new$error_pos, "\n")
#>
#>  # of updates: 163
opt_new$error
#> [1] 0.0005937236
```

Error is close to zero. Let us compare the original matrix with the reconstructed matrix based on the estimated u, d and v using the `optimizeFactorization` function.

```
newmat
#> $mat1
#>      [,1] [,2] [,3]
#> [1,]   41   10    6
#> [2,]   64   85    8
#> [3,]   82   87   57
opt_new$u[[1]] %*% diag(opt_new$d[[1]]) %*% t(opt_new$v)
#>          [,1]     [,2]      [,3]
#> [1,] 40.99568 10.00850  5.989020
#> [2,] 64.01161 84.99197  7.993729
#> [3,] 81.99198 87.00407 57.007920
```

```
opt_new1 <- optimizeFactorization(newmat, newu, newd, diag(3), tol = 1e-21)
cat("\n # of updates:", opt_new1$error_pos, "\n")
#>
#>  # of updates: 559
opt_new1$error
#> [1] 7.402498e-13
```

```
newmat
#> $mat1
#>      [,1] [,2] [,3]
#> [1,]   41   10    6
#> [2,]   64   85    8
#> [3,]   82   87   57
opt_new1$u[[1]] %*% diag(opt_new1$d[[1]]) %*% t(opt_new1$v)
#>      [,1] [,2] [,3]
#> [1,]   41   10    6
#> [2,]   64   85    8
#> [3,]   82   87   57
```

The reconstructed matrix is the same as the original matrix. Let us compare the U and V with that from the standard SVD.

```
newmat_svd <- svd(newmat[[1]])
```

```
newmat_svd$d
#> [1] 170.70126  31.96746  24.14876
opt_new1$d
#> $mat1
#> [1]  24.14876 170.70126  31.96746
```

```
newmat_svd$u
#>            [,1]       [,2]        [,3]
#> [1,] -0.2067092  0.1766241 -0.96232802
#> [2,] -0.6071027 -0.7944740 -0.01541011
#> [3,] -0.7672664  0.5810465  0.27145407
opt_new1$u[[1]]
#>             [,1]      [,2]       [,3]
#> [1,]  0.96232803 0.2067092  0.1766241
#> [2,]  0.01541005 0.6071027 -0.7944740
#> [3,] -0.27145402 0.7672664  0.5810465
```

```
newmat_svd$v
#>            [,1]       [,2]       [,3]
#> [1,] -0.6458388  0.1264120 -0.7529358
#> [2,] -0.7054605 -0.4758901  0.5252181
#> [3,] -0.2919209  0.8703727  0.3965270
opt_new1$v
#>            [,1]      [,2]       [,3]
#> [1,]  0.7529358 0.6458388  0.1264119
#> [2,] -0.5252182 0.7054605 -0.4758901
#> [3,] -0.3965269 0.2919209  0.8703727
```

The results agree except for the sign and order of columns.

2. Now, we will estimate the SVD of `newmat` using our iterative update function from another random matrix with the same dimension.

```
set.seed(253)
randmat_new <- createRandomMatrices(n = 1, ncols = 3, nrows = 3)
randmat_new
#> $mat1
#>      [,1] [,2] [,3]
#> [1,]   94   30   77
```

```
#> [2,]   60   35  100
#> [3,]   67   84   58
newsvd <- svd(randmat_new[[1]])
```

Let us create a list for the u and δ matrices we just obtained from the SVD of the random matrix. This allows us to use these matrices as the initial values for the `optimizeFactorization` function.

```
newu <- newd <- list()
newu[[names(randmat_new)]] <- newsvd$u
newd[[names(randmat_new)]] <- newsvd$d
```

The factorization error

```
calcDecompError(newmat, newu, newd, newsvd$v)
#> [1] 19465
```

Let us optimize.

```
opt_new <- optimizeFactorization(newmat, newu, newd, newsvd$v)
cat("\n # of updates:", opt_new$error_pos, "\n")
#>
#>  # of updates: 235
opt_new$error
#> [1] 0.0006461125
```

Error is close to zero. Let us compare the original matrix with the reconstructed matrix based on the estimated u, d and v using the `optimizeFactorization` function.

```
newmat
#> $mat1
#>      [,1] [,2] [,3]
#> [1,]   41   10    6
#> [2,]   64   85    8
#> [3,]   82   87   57
opt_new$u[[1]] %*% diag(opt_new$d[[1]]) %*% t(opt_new$v)
#>          [,1]     [,2]      [,3]
#> [1,] 40.99550 10.00886  5.988544
#> [2,] 64.01211 84.99162  7.993461
#> [3,] 81.99163 87.00424 57.008260
```

The estimated value is very close.

We can further improve our estimate by decreasing the tolerance parameter (`tol`) in the optimization function.

```
opt_new1 <- optimizeFactorization(newmat, newu, newd, newsvd$v, tol = 1e-21)
cat("\n # of updates:", opt_new1$error_pos, "\n")
#>
#>  # of updates: 673
opt_new1$error
#> [1] 9.156226e-14
opt_new1$u[[1]] %*% diag(opt_new1$d[[1]]) %*% t(opt_new1$v)
#>      [,1] [,2] [,3]
#> [1,]   41   10    6
#> [2,]   64   85    8
#> [3,]   82   87   57
```

The reconstructed matrix is the same as the original matrix. Let us compare the U and V with that from the standard SVD.

```
newmat_svd <- svd(newmat[[1]])
```

```
newmat_svd$u
#>            [,1]       [,2]        [,3]
#> [1,] -0.2067092  0.1766241 -0.96232802
#> [2,] -0.6071027 -0.7944740 -0.01541011
#> [3,] -0.7672664  0.5810465  0.27145407
opt_new1$u[[1]]
#>            [,1]       [,2]        [,3]
#> [1,] -0.2067092 -0.1766241  0.96232803
#> [2,] -0.6071027  0.7944740  0.01541009
#> [3,] -0.7672664 -0.5810465 -0.27145405
```

```
newmat_svd$v
#>            [,1]       [,2]       [,3]
#> [1,] -0.6458388  0.1264120 -0.7529358
#> [2,] -0.7054605 -0.4758901  0.5252181
#> [3,] -0.2919209  0.8703727  0.3965270
opt_new1$v
#>            [,1]       [,2]       [,3]
#> [1,] -0.6458388 -0.1264120  0.7529358
#> [2,] -0.7054605  0.4758901 -0.5252181
#> [3,] -0.2919209 -0.8703727 -0.3965269
```

The results agree!

6 Session info

```
sessionInfo()
#> R version 4.2.0 (2022-04-22)
#> Platform: x86_64-pc-linux-gnu (64-bit)
#> Running under: Ubuntu 20.04.4 LTS
#>
#> Matrix products: default
#> BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.9.0
#> LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.9.0
#>
#> locale:
#>  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
#>  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
#>  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
#>  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
#>  [9] LC_ADDRESS=C               LC_TELEPHONE=C
#> [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
#>
#> attached base packages:
#> [1] grid      stats     graphics  grDevices utils     datasets  methods
#> [8] base
#>
#> other attached packages:
#>  [1] gridExtra_2.3          goseq_1.48.0           geneLenDataBase_1.32.0
#>  [4] BiasedUrn_1.07         ggplot2_3.3.6          ggthemes_4.2.4
#>  [7] RColorBrewer_1.1-3     ComplexHeatmap_2.12.0  matrixStats_0.62.0
```

```
#> [10] dplyr_1.0.9            data.table_1.14.2      SBF_1.0.0.0
#>
#> loaded via a namespace (and not attached):
#>   [1] colorspace_2.0-3            rjson_0.2.21
#>   [3] ellipsis_0.3.2              rprojroot_2.0.3
#>   [5] circlize_0.4.15             XVector_0.36.0
#>   [7] GenomicRanges_1.48.0        GlobalOptions_0.1.2
#>   [9] fs_1.5.2                    clue_0.3-61
#>  [11] rstudioapi_0.13             farver_2.1.0
#>  [13] remotes_2.4.2               bit64_4.0.5
#>  [15] AnnotationDbi_1.58.0        fansi_1.0.3
#>  [17] xml2_1.3.3                  splines_4.2.0
#>  [19] codetools_0.2-18            doParallel_1.0.17
#>  [21] cachem_1.0.6                knitr_1.39
#>  [23] pkgload_1.2.4               Rsamtools_2.12.0
#>  [25] GO.db_3.15.0                dbplyr_2.1.1
#>  [27] cluster_2.1.3               png_0.1-7
#>  [29] compiler_4.2.0              httr_1.4.3
#>  [31] assertthat_0.2.1            Matrix_1.4-1
#>  [33] fastmap_1.1.0               cli_3.3.0
#>  [35] htmltools_0.5.2             prettyunits_1.1.1
#>  [37] tools_4.2.0                 gtable_0.3.0
#>  [39] glue_1.6.2                  GenomeInfoDbData_1.2.8
#>  [41] rappdirs_0.3.3              tinytex_0.39
#>  [43] Rcpp_1.0.8.3                Biobase_2.56.0
#>  [45] vctrs_0.4.1                 Biostrings_2.64.0
#>  [47] nlme_3.1-157                rtracklayer_1.56.0
#>  [49] iterators_1.0.14            xfun_0.31
#>  [51] stringr_1.4.0               ps_1.7.0
#>  [53] brio_1.1.3                  testthat_3.1.4
#>  [55] lifecycle_1.0.1             restfulr_0.0.13
#>  [57] devtools_2.4.3              XML_3.99-0.9
#>  [59] zlibbioc_1.42.0             scales_1.2.0
#>  [61] hms_1.1.1                   MatrixGenerics_1.8.0
#>  [63] parallel_4.2.0              SummarizedExperiment_1.26.1
#>  [65] curl_4.3.2                  yaml_2.3.5
#>  [67] memoise_2.0.1               biomaRt_2.52.0
#>  [69] stringi_1.7.6               RSQLite_2.2.14
#>  [71] highr_0.9                   S4Vectors_0.34.0
#>  [73] BiocIO_1.6.0                desc_1.4.1
#>  [75] foreach_1.5.2               filelock_1.0.2
#>  [77] GenomicFeatures_1.48.1      BiocGenerics_0.42.0
#>  [79] pkgbuild_1.3.1              BiocParallel_1.30.2
#>  [81] shape_1.4.6                 GenomeInfoDb_1.32.2
#>  [83] rlang_1.0.2                 pkgconfig_2.0.3
#>  [85] bitops_1.0-7                evaluate_0.15
#>  [87] lattice_0.20-45             purrr_0.3.4
#>  [89] GenomicAlignments_1.32.0    labeling_0.4.2
#>  [91] bit_4.0.4                   processx_3.5.3
#>  [93] tidyselect_1.1.2            magrittr_2.0.3
#>  [95] R6_2.5.1                    IRanges_2.30.0
#>  [97] generics_0.1.2              DelayedArray_0.22.0
#>  [99] DBI_1.1.2                   mgcv_1.8-40
```

```
#> [101] pillar_1.7.0               withr_2.5.0
#> [103] KEGGREST_1.36.0            RCurl_1.98-1.6
#> [105] tibble_3.1.7               crayon_1.5.1
#> [107] utf8_1.2.2                 BiocFileCache_2.4.0
#> [109] rmarkdown_2.14             GetoptLong_1.0.5
#> [111] progress_1.2.2             usethis_2.1.6
#> [113] blob_1.2.3                 callr_3.7.0
#> [115] digest_0.6.29              stats4_4.2.0
#> [117] munsell_0.5.0              sessioninfo_1.2.2
```

References

Alter, Orly, Patrick O Brown, and David Botstein. 2000. "Singular Value Decomposition for Genome-Wide Expression Data Processing and Modeling." *Proceedings of the National Academy of Sciences* 97 (18): 10101–6.

———. 2003. "Generalized singular value decomposition for comparative analysis of genome-scale expression data sets of two different organisms." *Proceedings of the National Academy of Sciences* 100 (6): 3351–56.

Ponnapalli, Sri Priya, Michael A Saunders, Charles F Van Loan, and Orly Alter. 2011. "A higher-order generalized singular value decomposition for comparison of global mRNA expression from multiple organisms." *PloS One* 6 (12): e28072.

Tamayo, Pablo, Daniel Scanfeld, Benjamin L Ebert, Michael A Gillette, Charles WM Roberts, and Jill P Mesirov. 2007. "Metagene projection for cross-platform, cross-species characterization of global transcriptional states." *Proceedings of the National Academy of Sciences* 104 (14): 5959–64.

Van Loan, Charles F. 1976. "Generalizing the Singular Value Decomposition." *SIAM Journal on Numerical Analysis* 13 (1): 76–83.