# computational details of low-rank matrix completion via adaptive-impute

*Alex Hayes*

*2019-05-21*

## Abstract

TODO: motivate ugh this is a show

This paper is a tutorial covering the computational details associated with computing the low-rank rank adaptive imputations for matrices described in (Cho, Kim, and Rohe 2015, 2018). This work extends previous data adaptive matrix imputation strategies such as that of (Mazumder, Hastie, and Tibshirani 2010), and has better performance while eliminating tuning parameters.

The tutorial proceeds in three parts. First, we introduce the imputation algorithms, useful tidbits of linear algebra and the R package `Matrix`, which we will use to illustrate computations. After a naive initial implementation which eats up lots of memory, we demonstrate a memory-efficient implementation. Finally, we extend this memory-efficient implementation to the partially observed matrices that possibly have a large number of observed zeros.

(Bates 2005; Cho, Kim, and Rohe 2015, 2018; Maechler and Bates 2006; Mazumder, Hastie, and Tibshirani 2010; Bro, Acar, and Kolda 2007)

## Notation & Algorithm

There are two steps to computing the low rank approximation of (Cho, Kim, and Rohe 2018). First we use compute an initial low rank estimate with the `AdaptiveInitialize` algorithm. This step is essentially a debiased SVD. We then use the initial solution as a seed for the `AdaptiveImpute` algorithm, a form of iterative SVD thresholding with a data adaptive thresholding parameter.

The input to these algorithms is a partially observed matrix $M$, and $r$, the desired rank of the low-rank approximation. We define $Y$ to be an indicator matrix that tells us whether or not we observed a particular value of $M$.

Here $\mathbf{u}_i, \boldsymbol{\lambda}_i, \mathbf{v}_i$ are functions that return the $i^{th}$ left singular vector, singular value, and right singular value, respectively. $\tilde{\alpha}$ is the data adaptive thresholding parameter. Note that it is the average of the uncalculated singular values, and is thus positive.

In line 8, $\langle , \rangle$ denotes the Frobenius inner norm. That is, for vectors $x, y$ and matrices $A, B$:

**Algorithm 1:** `AdapativeInitialize`

**Input:** $M, Y$ and $r$

1  $\hat{p} \leftarrow \frac{1}{nd} \sum_{i=1}^{n} \sum_{j=1}^{d} Y_{ij}$

2  $\Sigma_{\hat{p}} \leftarrow M^T M - (1 - \hat{p}) \operatorname{diag}(M^T M)$

3  $\Sigma_{t\hat{p}} \leftarrow M M^T - (1 - \hat{p}) \operatorname{diag}(M M^T)$

4  $\hat{V}_i \leftarrow \mathbf{v}_i(\Sigma_{\hat{p}})$ for $i = 1, ..., r$

5  $\hat{U}_i \leftarrow \mathbf{u}_i(\Sigma_{t\hat{p}})$ for $i = 1, ..., r$

6  $\tilde{\alpha} \leftarrow \frac{1}{d-r} \sum_{i=r+1}^{d} \boldsymbol{\lambda}_i(\Sigma_{\hat{p}})$

7  $\hat{\lambda}_i \leftarrow \frac{1}{\hat{p}} \sqrt{\boldsymbol{\lambda}_i(\Sigma_{\hat{p}}) - \tilde{\alpha}}$ for $i = 1, ..., r$

8  $\hat{s}_i \leftarrow \operatorname{sign}(\langle \hat{V}_i, \mathbf{v}_i(M) \rangle) \operatorname{sign}(\langle \hat{U}_i, \mathbf{u}_i(M) \rangle)$ for $i = 1, ..., r$

9  $\hat{\lambda}_i \leftarrow \hat{s}_i \cdot \hat{\lambda}_i$

10 **return** $\hat{\lambda}_i, \hat{U}_i, \hat{V}_i$ *for* $i = 1, ..., r$

$$\langle x, y \rangle = x^T y = \sum_i x_i y_i \tag{1}$$

$$\langle A, B \rangle = \sum_{i,j} A_{ij} B_{ij} = \sum_{ij} A \odot B \tag{2}$$

$$\tag{3}$$

We use $A \odot B$ to mean the elementwise (Hadamard) product of matrices $A$ and $B$.

We can use the output of `AdaptiveInitialize` to construct a low rank approximation to $M$ via:

$$\hat{M} = \sum_{i=1}^{r} \hat{\lambda}_i \hat{U}_i \hat{V}_i^T \tag{4}$$

Next up is `AdaptiveImpute`:

Here we again have some new notation. First we define

$$P_\Omega(A) = A \odot Y \tag{5}$$

$$P_\Omega^\perp(A) = A \odot (1 - Y) \tag{6}$$

These are the projecttion of a matrix $A$ onto the observed of $M$ and the unobserved elements of $M$, respectively. Similar $\Omega$ is the set of all pairs $(i, j)$ such that $M_{i,j}$ is observed. $\boldsymbol{\lambda}_i^2(A)$ is a function that returns the $i^{th}$ *squared* singular value of $A$ (i.e. $\boldsymbol{\lambda}_i^2(A) = (\boldsymbol{\lambda}_i(A))^2$). Finally $\| \cdot \|_F$ is the Frobenius norm of a matrix.

**Algorithm 2:** `AdaptiveImpute`

**Input:** $M, Y, r$ and $\varepsilon > 0$

**1** $Z^{(1)} \leftarrow \texttt{AdaptiveInitialize}(M, Y, r)$

**2 repeat**

**3** $\quad \tilde{M}^{(t)} \leftarrow P_\Omega(M) + P_\Omega^\perp(Z_t)$

**4** $\quad \hat{V}_i^{(t)} \leftarrow \mathbf{v}_i(\tilde{M}^{(t)})$ for $i = 1, ..., r$

**5** $\quad \hat{U}_i^{(t)} \leftarrow \mathbf{u}_i(\tilde{M}^{(t)})$ for $i = 1, ..., r$

**6** $\quad \tilde{\alpha}^{(t)} \leftarrow \frac{1}{d-r} \sum_{i=r+1}^{d} \boldsymbol{\lambda}_i^2(\tilde{M}^{(t)})$

**7** $\quad \hat{\lambda}_i^{(t)} \leftarrow \sqrt{\boldsymbol{\lambda}_i^2(\tilde{M}^{(t)}) - \tilde{\alpha}^{(t)}}$ for $i = 1, ..., r$

**8** $\quad Z^{(t+1)} \leftarrow \sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T}$

**9** $\quad t \leftarrow t + 1$

**10 until** $\|Z_{t+1} - Z_t\|_F^2 / \|Z_{t+1}\|_F$

**11 return** $\hat{\lambda}_i^{(t)}, \hat{U}_i^{(t)}, \hat{V}_i^{(t)}$ *for* $i = 1, ..., r$

## Pre-requisites

### The Matrix package

If you haven't used the `Matrix` package before, we recommend reading this introduction as well as this 2nd introduction.

```
library(Matrix)


set.seed(27)


# create a random 8 x 12 sparse matrix with 30 nonzero entries
M <- rsparsematrix(8, 12, nnz = 30)
M
```

```
summary(M)
```

TODO: different storage formats for sparse matrices. CSC, triplet, symmetric. for the most part `Matrix` does the right thing for you. The triplet form will be most important for us later on when we right out some matrix multiplications by hand.

```
# note that Matrix objects are S4 classes so we access their
# slots using the @ symbol
class(M)


M@x   # vector of values in M
M@i   # corresponding row indices
```

```
# if you want column indices you need a dgTMatrix
M2 <- as(M, "dgTMatrix")
M2@j  # corresponding column indices // only for dgCMatrix object
```

We will repeatedly calculate squared Frobenious norms throughout this tutorial. It's important to know that there are *many, many* ways to calculate this norm. We will almost always calculate the squared Frobenius norm of a sparse Matrix M via `sum(M@x^2)`. That said, these are all equivalent:

```
M^2    # square each element in M elementwise, return as sparse matrix
M@x^2  # square each element in M elementwise, return as vector of nonzeros

# the second version is much faster
bench::mark(
  sum(M@x^2),
  sum(M^2),
  sum(colSums(M^2)),
  norm(M, type = "F")^2,
  sum(M * M),
  iterations = 20
)
```

The projections $P_\Omega(A)$ and $P_\Omega^\perp(A)$ where $\Omega$ indicates the observed elements of a matrix $M$ and $A$ is another matrix with the same dimensions as $M$.

```
y <- as(M, "lgCMatrix")  # indicator matrix only
all.equal(y * M, M)  # don't lose anything multiplying by indicators


A <- matrix(1:(8*12), 8, 12)


all.equal(dim(A), dim(M))  # appropriate to practice projections with


# Omega indicates whether an entry of M was observed


# P_Omega (A)
A * y


!y


# P_Omega^perp (A): NOTE: this results in a *dense* matrix
A * (1 - y)


all(A * y + A * (1 - y) == A)  # can recover A from both projections together
```

matrix-matrix crossproducts

```
bench::mark(
  crossprod(M),       # dsCMatrix -- most specialized class, want this
  crossprod(M, M),    # dgCMatrix
  t(M) %*% M,         # dgCMatrix
  check = FALSE
)
```

matrix-vector crossproducts

```
# TODO
```

the `drop()` function helps us manage dimensions

```
one_col <- matrix(1:4)
one_row <- matrix(5:8, nrow = 1)

drop(one_col)
drop(one_row)

c(one_col)  # same thing, less explicit. use drop to be explicit
```

```
# TODO: drop vs drop0 vs drop1
```

diagonal of crossproduct

```
v_sign == colSums(svd_M$v * v_hat)
diag(t(svd_M$v) %*% v_hat)
diag(crossprod(svd_M$v, v_hat))

bench::mark(
  colSums(svd_M$v * v_hat),
  crossprod(rep(1, d), svd_M$v * v_hat),
  iterations = 50,
  check = FALSE
)


# write a diag_crossprod helper
```

```
rhos <- matrix(1:12, ncol = 4, byrow = TRUE)

bench::mark(
  diag(crossprod(rhos)),
  diag(t(rhos) %*% rhos),
  colSums(rhos * rhos),
  crossprod(rep(1, nrow(rhos)), rhos^2),
  check = FALSE
)
```

```r
rhos <- matrix(1:12, ncol = 4, byrow = TRUE)

bench::mark(
  diag(tcrossprod(rhos)),
  diag(crossprod(t(rhos))),
  diag(rhos %*% t(rhos)),
  rowSums(rhos * rhos),
  check = FALSE
)
```

what we get from `eigen()` and `svd()`: slightly different stuff: `u`, `d` and `v` versus `values` and `vectors`

NOTE: RSpectra *only* does truncated decompositions. if you want the full decomposition, you have to use base R stuff. different algos.

## Brief aside in sign ambiguity

yada yada yada the signs of the left and right singular vectors are not identified in SVD

A more elegant solution as proposed in Bro, Acar, and Kolda (2007) and used in Karl's paper is to take inner products

NOTE TO SELF: identifying the signs of a single SVD is a much harder task than comparing two SVDs and seeing if they are the same up to sign differences. we only need to check if they are the same up to sign differences.

```r
set.seed(17)
M <- rsparsematrix(8, 12, nnz = 30) # small example, not very sparse

# number of singular vectors to compute
k <- 4

s <- svd(M, k, k)
s2 <- svds(M, k, k)

# irritating: svd() always gives you all the singular values even if you
# only request the first K singular vectors
s$u %*% diag(s$d[1:k]) %*% t(s$v)

# based on the flip_signs function of
# https://stats.stackexchange.com/questions/134282/relationship-between-svd-and-pca-how-to-use-svd-to-p
equal_svds <- function(s, s2) {
```

```r
  # svd() always gives you all the singular values, but we only
  # want to compare the first k
  k <- ncol(s$u)

  # the term sign(s$u) * sign(s2$u) performs a sign correction

  # isTRUE because output of all.equal is not a boolean, it's something
  # weird when the inputs aren't equal. lol why

  u_ok <- isTRUE(
    all.equal(s$u, s2$u * sign(s$u) * sign(s2$u), check.attributes = FALSE)
  )

  v_ok <- isTRUE(
    all.equal(s$v, s2$v * sign(s$v) * sign(s2$v), check.attributes = FALSE)
  )

  d_ok <- isTRUE(all.equal(s$d[1:k], s2$d[1:k], check.attributes = FALSE))

  u_ok && v_ok && d_ok
}
```

**Linear algebra facts**

Throughout these computations, we will repeatedly use several key facts about eigendecompositions, singular value decompositions (SVD) and the relationship between the two.

https://en.wikipedia.org/wiki/Gramian_matrix X'X – positive semi-def, so the singular values of M'M are the same as the eigenvalues

question: if A, B positive, is sum(svd(A - B)$d$) == $sum(svd(A)$d) - sum(svd(B)$d)

answer: NO! can't split into two easy computations and then combine them possibly use this to get some sort of bound?

think about what happens as p_hat -> 0

A key observation here is that $M^T M$ and

Fact: sum of squared singular values is `trace(A^T A)` https://math.stackexchange.com/questions/2281721/sum-of-singular-values-of-a-matrix

Fact: for symmetric positive definite matrices the eigendecomp is equal to the singular value decomp

Fact: sum of eigenvalues of M is equal to trace(M)

Consequence: for pos def symmetric M the sum of the singular values is trace(M) as well

Again computing `alpha` deserves some explanation.

- reference: https://math.stackexchange.com/questions/1463269/how-to-obtain-sum-of-square-of-eigenvalues-without-fir
- Frobenius norm (A) = trace(crossprod(A))
- TODO: how we know this thing is strictly positive to prevent sqrt() from exploding

```
## STOPPED HERE: WHY ARE the following not the same?
  isSymmetric(sigma_p)
  eigen(sigma_p)$values
  sum(diag(sigma_p))
  sum(svd(sigma_p)$d)
  sum(eigen(sigma_p)$values)


  # let's think just about the first term for a moment
  sum(diag(MtM / p_hat^2))
  sum(svd(MtM / p_hat^2)$d)


  sum(colSums(M^2 / p_hat^2))


  # has some negative eigenvalues


  # Fact: for a symmetric matrix, the singular values are the *absolute* values
  # of the eigenvalues


  # https://www.mathworks.com/content/dam/mathworks/mathworks-dot-com/moler/eigs.pdf


  eigen(sigma_p)$values
  sum(eigen(sigma_p)$values)
  sum(abs(eigen(sigma_p)$values))
  sum(svd(sigma_p)$d)
  sum(abs(diag(sigma_p)))


  # those agree so what about the second term


  # note to self: alpha should be positive
  # issue karl ran into:
  # https://math.stackexchange.com/questions/381808/sum-of-eigenvalues-and-singular-values
  # how to get the sum of singular values itself (start):
  # https://math.stackexchange.com/questions/569989/sum-of-singular-values-of-ab


  # options when sigma_p is not positive definite:
  # - calculate the full SVD
  # - set alpha to zero (don't truncate the singular values)
  # - this lower bounds the average of the remaining singular values
```

```
  # -

  # positive semi-definite is enough since symmetric and eigen/singular values
  # of zero don't matter

  # this is only an issue in the initialization. in the iterative updates
  # we use the squared singular values, which we can more easily calculate
  # the sum of

  # ask Karl what he wants to do about this: computing a full SVD is gonna be really expensive.
```

FACT: sum(diag(crossprod(M))) == sum(M^2)

## Reference implementation

```r
library(RSpectra)
library(Matrix)
```

```r
adaptive_initialize <- function(M, r) {

  # TODO: ignores observed zeros!
  p_hat <- nnzero(M) / prod(dim(M))  # line 1

  MtM <- crossprod(M)
  MMt <- tcrossprod(M)

  # need to divide by p~2 from Cho et al 2016 to get the "right"
  # singular values / singular values on a comparable scale

  # both of these matrices are symmetric, but not necessarily positive
  # this has important implications for the SVD / eigendecomp relationship

  sigma_p <- MtM / p_hat^2 - (1 - p_hat) * diag(diag(MtM))  # line 2
  sigma_t <- MMt / p_hat^2 - (1 - p_hat) * diag(diag(MMt))  # line 3

  # crossprod() and tcrossprod() return dsCMatrix objects,
  # sparse matrix objects that know they are symmetric

  # unfortunately, RSpectra doesn't support dsCMatrix objects,
  # but does support dgCMatrix objects, a class representing sparse
  # but not symmetric matrices

  # support for dsCMatrix objects in RSpectra is on the way,
```

```
  # which will eliminate the need for the following coercions.
  # see: https://github.com/yixuan/RSpectra/issues/15

  sigma_p <- as(sigma_p, "dgCMatrix")
  sigma_t <- as(sigma_t, "dgCMatrix")

  svd_p <- svds(sigma_p, r)  # TODO: is eigs_sym() faster?
  svd_t <- svds(sigma_t, r)

  v_hat <- svd_p$v  # line 4
  u_hat <- svd_t$u  # line 5

  n <- nrow(M)
  d <- ncol(M)

  # NOTE: alpha is incorrect due to singular values and eigenvalues
  # being different when sigma_p is not positive

  alpha <- (sum(diag(sigma_p)) - sum(svd_p$d)) / (d - r)  # line 6
  lambda_hat <- sqrt(svd_p$d - alpha) / p_hat             # line 7

  svd_M <- svds(M, r)

  v_sign <- crossprod(rep(1, d), svd_M$v * v_hat)
  u_sign <- crossprod(rep(1, n), svd_M$u * u_hat)
  s_hat <- drop(sign(v_sign * u_sign))

  lambda_hat <- lambda_hat * s_hat  # line 8

  list(u = u_hat, d = lambda_hat, v = v_hat)
}
```

It's worth commenting on computation of `alpha` and `s_hat`.

When we compute `alpha` in line 20 `adaptive_initialize()`, we don't want to do the full eigendecomposition of $\Sigma_{\hat{p}}$ since that could take a long time, so we use trick and recall that the trace of a matrix (the sum of it's diagonal elements) equals the sum of all the eigenvalues. Then we subtract off the first $r$ eigenvalues, which we do compute, and are left with $\sum_{i=r+1}^{d} \lambda_i(\Sigma_{\hat{p}})$.

```
adaptive_impute <- function(M, r, epsilon = 1e-7) {

  s <- adaptive_initialize(M, r)
  Z <- s$u %*% diag(s$d) %*% t(s$v)  # line 1
  delta <- Inf
```

```r
  while (delta > epsilon) {

    y <- as(M, "lgCMatrix")  # indicator if entry of M observed
    M_tilde <- M + Z * (1 - y)  # line 3

    svd_M <- svds(M_tilde, r)

    u_hat <- svd_M$u  # line 4
    v_hat <- svd_M$v  # line 5

    d <- ncol(M)

    alpha <- (sum(M_tilde^2) - sum(svd_M$d^2)) / (d - r)   # line 6

    lambda_hat <- sqrt(svd_M$d^2 - alpha)  # line 7

    Z_new <- u_hat %*% diag(lambda_hat) %*% t(v_hat)

    delta <- sum((Z_new - Z)^2) / sum(Z^2)
    Z <- Z_new

    print(glue::glue("delta: {round(delta, 8)}, alpha: {round(alpha, 3)}"))
  }


  Z
}
```

Finally we can do a minimal sanity check and see if this code even runs, and see if we are recovering something close-ish to implanted low-rank structure.

```r
n <- 500
d <- 100
r <- 5

A <- matrix(runif(n * r, -5, 5), n, r)
B <- matrix(runif(d * r, -5, 5), d, r)
M0 <- A %*% t(B)

err <- matrix(rnorm(n * d), n, d)
Mf <- M0 + err

p <- 0.3
y <- matrix(rbinom(n * d, 1, p), n, d)
```

```r
dat <- Mf * y

init <- adaptive_initialize(dat, r)
filled <- adaptive_impute(dat, r)
```

## Low-rank implementation

The reference implementation has some problems. As our data matrix $M$ gets larger, we can no longer fit the dense representation of $\hat{M}$ and $Z^{(t)}$ into memory. Instead, we need to work with just the low rank components $\hat{\lambda}, \hat{U}$ and $\hat{V}$.

This leads us to following implementation:

```r
low_rank_adaptive_initialize <- function(M, r) {

  M <- as(M, "dgCMatrix")

  p_hat <- nnzero(M) / prod(dim(M))   # line 1

  # NOTE: skip explicit computation of line 2
  # NOTE: skip explicit computation of line 3

  eig_p <- eigen_helper(M, r)
  eig_t <- eigen_helper(t(M), r)

  lr_v_hat <- eig_p$vectors   # line 4
  lr_u_hat <- eig_t$vectors   # line 5

  d <- ncol(M)
  n <- nrow(M)

  # NOTE: alpha is again incorrect since we work with eigenvalues
  # rather than singular values here
  sum_eigen_values <- sum(M@x^2) / p^2 - (1 - p) * sum(colSums(M^2))
  lr_alpha <- (sum_eigen_values - sum(eig_p$values)) / (d - r)   # line 6

  lr_lambda_hat <- sqrt(eig_p$values - lr_alpha) / p_hat   # line 7

  # TODO: Karl had another sign computation here that he said was faster
  # but it wasn't documented anywhere, so I'm going with what was in the
  # paper

  lr_svd_M <- svds(M, r)
```

```r
  # v_hat is d by r
  lr_v_sign <- crossprod(rep(1, d), lr_svd_M$v * lr_v_hat)
  lr_u_sign <- crossprod(rep(1, n), lr_svd_M$u * lr_u_hat)
  lr_s_hat <- c(sign(lr_v_sign * lr_u_sign))  # line 8

  lr_lambda_hat <- lr_lambda_hat * lr_s_hat

  list(u = lr_u_hat, d = lr_lambda_hat, v = lr_v_hat)
}
```

What does `eigen_helper()` do? Describe the return object (also do this for svds)

```r
# Take the eigendecomposition of t(M) %*% M - (1 - p) * diag(t(M) %*% M)
# using sparse computations only
eigen_helper <- function(M, r) {
  eigs_sym(
    Mx, r,
    n = ncol(M),
    args = list(
      M = M,
      p = nnzero(M) / prod(dim(M))
    )
  )
}


# compute (t(M) %*% M / p^2 - (1 - p) * diag(diag(t(M) %*% M))) %*% x
# using sparse operations

# TODO: divide the second term by p^2 like in the reference implementatio
Mx <- function(x, args) {
  drop(
    crossprod(args$M, args$M %*% x) / args$p^2 - (1 - args$p) * Diagonal(ncol(args$M), colSums(args$M^2)
  )
}
```

Now we check that `Mx` works

```r
x <- rnorm(12)
p <- 0.3
out <- (t(M) %*% M / p^2 - (1 - p) * diag(diag(t(M) %*% M))) %*% x
out2 <- Mx(x, args = list(M = M, p = p))
all.equal(as.matrix(out), as.matrix(out))
```

TODO: update alg description to divide by $p^2$ to get the right singular values

Quickly check that the components works before we try the code that integrates them all together

Finally, sanity check this by comparing to the reference implementation. These don't agree, which isn't great:

```
lr_init <- low_rank_adaptive_initialize(dat, r)

# some weird stuff is happening with the singular values but I'm
# going to not worry about it for the time being

equal_svds(init, lr_init)
```

## Space-efficient adaptive impute

TODO: figure out the actual space complexity

Recall the algorithm looks like

$$\hat{M} = \sum_{i=1}^{r} \hat{s}_i \hat{\lambda}_i \hat{U}_i \hat{V}_i^T \tag{7}$$

---

**Algorithm 3:** AdapativeImpute

**Input:** $M, y, r$ and $\varepsilon > 0$

1   $Z^{(1)} \leftarrow$ AdaptiveInitialize$(M, y, r)$

2 **repeat**

3     $\tilde{M}^{(t)} \leftarrow P_\Omega(M) + P_\Omega^\perp(Z_t)$

4     $\hat{V}_i^{(t)} \leftarrow \mathbf{v}_i(\tilde{M}^{(t)})$ for $i = 1, ..., r$

5     $\hat{U}_i^{(t)} \leftarrow \mathbf{u}_i(\tilde{M}^{(t)})$ for $i = 1, ..., r$

6     $\tilde{\alpha}^{(t)} \leftarrow \frac{1}{d-r} \sum_{i=r+1}^{d} \boldsymbol{\lambda}_i^2(\tilde{M}^{(t)})$

7     $\hat{\lambda}_i^{(t)} \leftarrow \sqrt{\boldsymbol{\lambda}_i^2(\tilde{M}^{(t)}) - \tilde{\alpha}^{(t)}}$ for $i = 1, ..., r$

8     $Z^{(t+1)} \leftarrow \sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T}$

9     $t \leftarrow t + 1$

10 **until** $\|Z_{t+1} - Z_t\|_F^2 / \|Z_{t+1}\|_F$

11 **return** $\hat{\lambda}_i^{(t)}, \hat{U}_i^{(t)}, \hat{V}_i^{(t)}$ *for* $i = 1, ..., r$

---

Now we need two things:

1. The SVD of $\tilde{M}^{(t)}$
2. (Certain sums of) the squared singular values.

**Sums of squared singular values**

For a matrix $A$, the sum of squared singular values (denoted by $\lambda_i$) equals the squared frobenius norm:

$$\sum_{i=1}^{\min(n,d)} \lambda_i^2 = ||A||_F^2 = \text{trace}(A^T A) \tag{8}$$

Also note that

$$||A + B||_F^2 = ||A||_F^2 + ||B||_F^2 + 2 \cdot \langle A, B \rangle_F \tag{9}$$

Now we consider $\tilde{M}^{(t)}$. Suppose that unobserved values of $M$ are set to zero, as is the case for $M$ stored in a sparse matrix representation

$$\tilde{M}^{(t)} = P_\Omega(M) + P_\Omega^\perp(Z_t) \tag{10}$$

$$= P_\Omega(M) + P_\Omega^\perp \left( \sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T} \right) \tag{11}$$

Now we need

$$||\tilde{M}^{(t)}||_F^2 = \left\| P_\Omega(M) + P_\Omega^\perp \left( \sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T} \right) \right\|_F^2 \tag{12}$$

$$= ||P_\Omega(M)||_F^2 + \left\| P_\Omega^\perp \left( \sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T} \right) \right\|_F^2 + 2 \cdot \left\langle P_\Omega(M), P_\Omega^\perp \left( \sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T} \right) \right\rangle_F \tag{13}$$

$$= ||P_\Omega(M)||_F^2 + \left\| P_\Omega^\perp \left( \sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T} \right) \right\|_F^2 \tag{14}$$

Where the cancellation in the final line follows because

$$\left\langle P_\Omega(M), P_\Omega^\perp \left( \sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T} \right) \right\rangle_F = \sum_{i,j} P_\Omega(M)_{ij} \cdot P_\Omega^\perp(Z_t)_{ij} = \sum_{i,j} 0 = 0 \tag{15}$$

Now we need one more trick, which is that

$$\|Z_t\|_F^2 = \left\| P_\Omega(Z_t) + P_\Omega^\perp(Z_t) \right\|_F^2 = \|P_\Omega(Z_t)\|_F^2 + \left\| P_\Omega^\perp(Z_t) \right\|_F^2 \tag{16}$$

and then

$$\left\|P_\Omega^\perp(Z_t)\right\|_F^2 = \|Z_t\|_F^2 - \|P_\Omega(Z_t)\|_F^2 \tag{17}$$

$$= \sum_{i=1}^r \lambda_i^2 - \|Z_t \odot Y\|_F^2 \tag{18}$$

Putting it all together we see

$$||\tilde{M}^{(t)}||_F^2 = \|P_\Omega(M)\|_F^2 + \left\|P_\Omega^\perp\left(\sum_{i=1}^r \hat{\lambda}_i^{(t)}\hat{U}_i^{(t)}\hat{V}_i^{(t)^T}\right)\right\|_F^2 \tag{19}$$

$$= \|M\|_F^2 + \sum_{i=1}^r \lambda_i^2 - \|Z_t \odot Y\|_F^2 \tag{20}$$

In code we will have a sparse matrix `M` and a list `s` with elements of the SVD. The first Frobenious norm is quick to calculate, but I am not sure how to calculate the other two frobenius norms.

```r
# s is a matrix defined in terms of it's svd
# G is a sparse matrix
# compute only elements of U %*% diag(d) %*% t(V) only on non-zero elements of G
# G and U %*% t(V) must have same dimensions


# maybe call this svd_perp?
svd_perp <- function(s, mask) {

  # note: must be dgTMatrix to get column indexes j larger
  # what if we used dlTMatrix here?
  m <- as(mask, "dgTMatrix")

  # the indices for which we want to compute the matrix multiplication
  # turn zero based indices into one based indices
  i <- m@i + 1
  j <- m@j + 1

  # gets rows and columns of U and V to multiply, then multiply
  ud <- s$u %*% diag(s$d)
  left <- ud[i, ]
  right <- s$v[j, ]

  # compute inner products to get elements of U %*% t(V)
  uv <- rowSums(left * right)

  # NOTE: specify dimensions just in case
  sparseMatrix(i = i, j = j, x = uv, dims = dim(mask))
}
```

Test it

```
set.seed(17)

M <- rsparsematrix(8, 12, nnz = 30)
s <- svds(M, 5)

y <- as(M, "lgCMatrix")

Z <- s$u %*% diag(s$d) %*% t(s$v)

all.equal(
  svd_perp(s, M),
  Z * y
)
```

So, to take an eigendecomp you just need to be able to do $Mx$. To take an SVD, what do you need? matrix vector and matrix transpose vector multiplication

```
set.seed(17)
r <- 5

M <- rsparsematrix(8, 12, nnz = 30)
y <- as(M, "lgCMatrix")

s <- svds(M, r)
Z <- s$u %*% diag(s$d) %*% t(s$v)

M_tilde <- M + Z * (1 - y)  # dense!

Z_perp <- svd_perp(s, M)
sum_singular_squared <- sum(M@x^2) + sum(s$d^2) - sum(Z_perp@x^2)

all.equal(
  sum(svd(M_tilde)$d^2),
  sum_singular_squared
)
```

**SVD of M tilde**

```
set.seed(17)
r <- 5

M <- rsparsematrix(8, 12, nnz = 30)
```

17

```r
y <- as(M, "lgCMatrix")

s <- svds(M, r)
Z <- s$u %*% diag(s$d) %*% t(s$v)

M_tilde <- M + Z * (1 - y)   # dense!

svd_M_tilde <- svds(M_tilde, r)
svd_M_tilde
```

```r
Ax <- function(x, args) {
  drop(M_tilde %*% x)
}

Atx <- function(x, args) {
  drop(t(M_tilde) %*% x)
}

# is eigs_sym() with a two-sided multiply faster?
args <- list(u = s$u, d = s$d, v = s$v, m = M)
test1 <- svds(Ax, k = r, Atrans = Atx, dim = dim(M), args = args)

test1
svd_M_tilde

all.equal(
  svd_M_tilde,
  test1
)
```

So we're done our first sanity check of the function interface. Let $x$ be a vector. Now we want to calculate

$$\tilde{M}^{(t)}x = \left[P_\Omega(M) + P_\Omega^\perp(Z_t)\right] x \tag{21}$$
$$= \left[P_\Omega(M) - P_\Omega(Z_t) + P_\Omega(Z_t) + P_\Omega^\perp(Z_t)\right] x \tag{22}$$
$$= P_\Omega(M - Z_t)x + Z_t x \tag{23}$$

where we can think of $R_t \equiv P_\Omega(M - Z_t)$ as "residuals" of sorts. Crucially, $R_t$ is sparse, and

$$Z_t x = (\hat{U} \operatorname{diag}(\hat{\lambda}_1, ..., \hat{\lambda}_r)\hat{V}^t)x \tag{24}$$
$$= (\hat{U}(\operatorname{diag}(\hat{\lambda}_1, ..., \hat{\lambda}_r)(\hat{V}^t x))) \tag{25}$$

So now the memory requirement of the computation has been reduced to that of two sparse matrix vector multiplications, rather than that of fitting the dense matrix $P_\Omega^\perp(Z_t)$ into memory.

Similarly, for the transpose, we have

$$\tilde{M}^{(t)^T} x = \left[ P_\Omega(M) + P_\Omega^\perp(Z_t) \right]^T x \tag{26}$$

$$= \left[ P_\Omega(M) - P_\Omega(Z_t) + P_\Omega(Z_t) + P_\Omega^\perp(Z_t) \right]^T x \tag{27}$$

$$= P_\Omega(M - Z_t)^T x + Z_t^T x \tag{28}$$

This leads us to a second, less memory intensive implementation of `Ax()` and `Atx()`:

```r
# input: M, Z_t as a low-rank SVD list s

R <- M - svd_perp(s, M)   # residual matrix
args <- list(u = s$u, d = s$d, v = s$v, R = R)


Ax <- function(x, args) {
  drop(args$R %*% x + args$u %*% diag(args$d) %*% crossprod(args$v, x))
}


Atx <- function(x, args) {
  # TODO: can we use a crossprod() for the first multiplication here?
  drop(t(args$R) %*% x + args$v %*% diag(args$d) %*% crossprod(args$u, x))
}


# is eigs_sym() with a two-sided multiply faster?
test2 <- svds(Ax, k = r, Atrans = Atx, dim = dim(M), args = args)


all.equal(
  svd_M_tilde,
  test2
)

relative_f_norm_change <- function(s_new, s) {
  # TODO: don't do the dense calculation here

  Z_new <- s_new$u %*% diag(s_new$d) %*% t(s_new$v)
  Z_new <- s$u %*% diag(s$d) %*% t(s$v)


  sum((Z_new - Z)^2) / sum(Z^2)
}

low_rank_adaptive_impute <- function(M, r, epsilon = 1e-03) {
```

```r
  # coerce M to sparse matrix such that we use sparse operations
  M <- as(M, "dgCMatrix")

  # low rank svd-like object, s ~ Z_1
  s <- low_rank_adaptive_initialize(M, r)  # line 1
  delta <- Inf
  d <- ncol(M)
  norm_M <- sum(M@x^2)

  while (delta > epsilon) {

    # update s: lines 4 and 5
    # take the SVD of M-tilde

    R <- M - svd_perp(s, M)  # residual matrix
    args <- list(u = s$u, d = s$d, v = s$v, R = R)

    s_new <- svds(Ax, k = r, Atrans = Atx, dim = dim(M), args = args)

    MtM <- norm_M + sum(s_new$d^2) - sum(svd_perp(s_new, M)^2)
    alpha <- (sum(MtM) - sum(s_new$d^2)) / (d - r)  # line 6

    s_new$d <- sqrt(s_new$d^2 - alpha)  # line 7

    # NOTE: skip explicit computation of line 8
    delta <- relative_f_norm_change(s_new, s)

    s <- s_new

    print(glue::glue("delta: {round(delta, 8)}, alpha: {round(alpha, 3)}"))
  }

  s
}

out <- low_rank_adaptive_impute(M, r)
out
```

$$\texttt{MtM} = \tilde{M}^{(t)^T} \tilde{M}^{(t)} \tag{29}$$

## Extension to a mixture of observed and unobserved missingness

originally solving an optimization vaguely of the form

$$\left\| M - \hat{M} \right\|_F^2 \tag{30}$$

where $M$ is a partially observed matrix. now, we let $M' = YM$ where $Y$ is an indicator of whether or not $M$ was observed. Typically we have some setup like

$$M = \begin{bmatrix} \cdot & \cdot & 3 & 1 & \cdot \\ 3 & \cdot & \cdot & 8 & \cdot \\ \cdot & -1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 2 & \cdot & \cdot & \cdot \\ 5 & \cdot & 7 & \cdot & 4 \end{bmatrix}, \qquad Y = \begin{bmatrix} \cdot & \cdot & 1 & 1 & \cdot \\ 1 & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot \\ 1 & \cdot & 1 & \cdot & 1 \end{bmatrix} \tag{31}$$

Here the symbol $\cdot$ means that an entry of the matrix was unobserved.

but what we do now is, continuing to represent $M$ as a sparse matrix with no zero entries, is *observe* a bunch of zeros. So we might know that the upper triangle of $M$ has structurally missing zeros that we have observed are missing. These zeros are primarily important because they affect the residuals in our calculations. In this particular case, the take multiplication by $M$, a sparse operation, and make it into a dense operation.

At this point it becomes useful to introduce some additional notation. Let $\tilde{\Omega}$ be the set of indicies $(i, j)$ such that $M_{i,j}$ is non-zero. Observe that $\tilde{\Omega} \subset \Omega$. Then we have $P_{\tilde{\Omega}}(A) = P_\Omega(A)$.

$$M = \begin{bmatrix} 0 & 0 & 3 & 1 & 0 \\ 3 & 0 & 0 & 8 & 0 \\ \cdot & -1 & 0 & 0 & 0 \\ \cdot & \cdot & \cdot & 0 & 0 \\ \cdot & 2 & \cdot & \cdot & 0 \\ 5 & \cdot & 7 & \cdot & 4 \end{bmatrix}, \qquad Y = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ \cdot & 1 & 1 & 1 & 1 \\ \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & 1 & \cdot & \cdot & 1 \\ 1 & \cdot & 1 & \cdot & 1 \end{bmatrix}, \qquad M' = \begin{bmatrix} \cdot & \cdot & 3 & 1 & \cdot \\ 3 & \cdot & \cdot & 8 & \cdot \\ \cdot & -1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 2 & \cdot & \cdot & \cdot \\ 5 & \cdot & 7 & \cdot & 4 \end{bmatrix} \tag{32}$$

And now we need to figure out how to calculate

$$\tilde{M}^{(t)} x = \left[ P_\Omega(M) + P_{\tilde{\Omega}}^\perp(Z_t) \right] x \tag{33}$$

$$= \left[ P_\Omega(M) - P_\Omega(Z_t) + P_\Omega(Z_t) + P_{\tilde{\Omega}}^\perp(Z_t) \right] x \tag{34}$$

$$= P_\Omega(M) x - P_\Omega(Z_t) x + Z_t x \tag{35}$$

$$= P_{\tilde{\Omega}}(M) x - P_\Omega(Z_t) x + Z_t x \tag{36}$$

We already know how to calculate $P_{\tilde{\Omega}}(M)$ and $Z_t x$ using sparse operations, so we're left with $P_{\Omega}(Z_t)x$. Note that $P_{\Omega}(Z_t)x \neq P_{\tilde{\Omega}}(Z_t)x$ since $Z_t$ is not necessarily zero on $\tilde{\Omega}^{\perp}$. In other words $P_{\Omega}^{\perp}(Z_t) \neq Z_t$.

When $Y$ is dense, there is no way to avoid paying the computational cost of the dense or near dense computation. Our primary concern is fitting $Y$ into memory for large datasets. This is an issue when $Y$ is dense but with no discernible structure that permits a more compact representation.

Similarly we need to be able to calculate

$$\tilde{M}^{(t)^T}x = \left[P_{\Omega}(M) + P_{\Omega}^{\perp}(Z_t)\right]^T x \tag{37}$$
$$= \left[P_{\Omega}(M)^T + P_{\Omega}^{\perp}(Z_t)^T\right]x \tag{38}$$
$$= \left[P_{\Omega}(M)^T - P_{\Omega}(Z_t)^T + P_{\Omega}(Z_t)^T + P_{\Omega}^{\perp}(Z_t)^T\right]x \tag{39}$$
$$= P_{\Omega}(M)^T x - P_{\Omega}(Z_t)^T x + Z_t^T x \tag{40}$$
$$= P_{\tilde{\Omega}}(M)^T x - P_{\Omega}(Z_t)^T x + Z_t^T x \tag{41}$$

If we can fit $Y$ into memory, we can do a low-rank computation, only calculating elements $Z_{ij}^{(t)}$ when $Y_{ij} = 1$. When $Y$ is stored as a vector of row indices together with a vector of column indices (plus some information about the dimension), we can write the computation out:

```
M <- Matrix(
  rbind(
    c(0, 0, 3, 1, 0),
    c(3, 0, 0, 8, 0),
    c(0, -1, 0, 0, 0),
    c(0, 0, 0, 0, 0),
    c(0, 2, 0, 0, 0),
    c(5, 0, 7, 0, 4)
  )
)


Y <- rbind(
  c(1, 1, 1, 1, 1),
  c(1, 1, 1, 1, 1),
  c(0, 1, 1, 1, 1),
  c(0, 0, 1, 1, 1),
  c(0, 1, 0, 1, 1),
  c(1, 0, 1, 0, 1)
)


s <- svds(M, 2)


Y <- as(Y, "TsparseMatrix")
```

```
# triplet form
# compressed column matrix form even better but don't
# understand the format
Y <- as(Y, "lgCMatrix")
Y <- as(Y, "lgTMatrix")

# ugh: RcppArmadillo only supports dgTMatrix rather than
# lgTMatrix which is somewhat unfortunate

# link: https://cran.r-project.org/web/packages/RcppArmadillo/vignettes/RcppArmadillo-sparseMatrix.pdf

Y

x <- rnorm(5)

# want to calculate
Z <- s$u %*% diag(s$d) %*% t(s$v)
out <- drop((Z * Y) %*% x)
out
```

At this point it's worth writing out explicitly how calculate $Z_{ij}^{(t)}$.

$$Z_{ij}^{(t)} = \left( \sum_{\ell=1}^{r} \hat{U}_\ell \hat{d}_\ell \hat{V}_\ell^T \right)_{ij} \tag{42}$$

For a visual reminder, this looks like (using $\mathrm{diag}(\hat{d})$ and $\hat{\Sigma}$ somewhat interchangeably here)

$$Z^{(t)} = \hat{U} \, \mathrm{diag}(\hat{d}) \, \hat{V}^T = \begin{bmatrix} U_1 & U_2 & \ldots & U_r \end{bmatrix} \begin{bmatrix} d_1 & 0 & \ldots & 0 \\ 0 & d_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & d_r \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \\ \vdots \\ V_r^T \end{bmatrix} \tag{43}$$

$$\begin{bmatrix} U_{11} & U_{12} & \ldots & U_{1r} \\ U_{21} & U_{22} & \ldots & U_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ U_{n1} & U_{n2} & \ldots & U_{nr} \end{bmatrix} \begin{bmatrix} d_1 & 0 & \ldots & 0 \\ 0 & d_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & d_r \end{bmatrix} \begin{bmatrix} V_{11} & V_{21} & \ldots & V_{d1} \\ V_{12} & V_{22} & \ldots & V_{d2} \\ \vdots & \vdots & \ddots & \vdots \\ V_{1r} & V_{2r} & \ldots & V_{dr} \end{bmatrix} \tag{44}$$

n x d = (n x r) x (r x r) x (r x d)

(r x d) is after the transpose

$$
\begin{bmatrix}
U_{11} & U_{12} & \dots & U_{1r} \\
U_{21} & U_{22} & \dots & U_{2r} \\
\vdots & \vdots & \ddots & \vdots \\
U_{n1} & U_{n2} & \dots & U_{nr}
\end{bmatrix}
\begin{bmatrix}
d_1 & 0 & \dots & 0 \\
0 & d_2 & \dots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \dots & d_r
\end{bmatrix}
\begin{bmatrix}
V_{11} & V_{21} & \dots & V_{d1} \\
V_{12} & V_{22} & \dots & V_{d2} \\
\vdots & \vdots & \ddots & \vdots \\
V_{1r} & V_{2r} & \dots & V_{dr}
\end{bmatrix}
\tag{45}
$$

Also because I never remember anything, recall that $Z$ is $n \times d$, $x$ is $d \times 1$ and $Zx$ is $n \times 1$:

$$
(Zx)_i = \sum_{j=1}^{d} Z_{ij} \cdot x_j
\tag{46}
$$

(recall that $X_i$ always refers to the $i^{th}$ column of $X$ in this document, and $X_{i\cdot}$ refers to the $i^{th}$ row of $X$)

Also note that

$$
Z_{ij} = \sum_{k=1}^{r} d_k U_{ik} V_{kj}
\tag{47}
$$

Putting these together we find

$$
(Zx)_i = \sum_{j=1}^{d} Z_{ij} \cdot x_j
\tag{48}
$$

$$
= \sum_{j=1}^{d} \sum_{k=1}^{r} d_k U_{ik} V_{kj} \cdot x_j
\tag{49}
$$

$$
= \sum_{j=1}^{d} U_{i\cdot}^{T} (DV)_{j\cdot} \cdot x_j
\tag{50}
$$

```r
# mask as a pair list
# L and Z / svd are both n x d matrices
# x is a d x 1 matrix / vector
masked_svd_times_x <- function(s, mask, x) {

  stopifnot(inherits(mask, "lgTMatrix"))

  u <- s$u
  d <- s$d
  v <- s$v

  zx <- numeric(nrow(u))
```

```r
  # lgTMatrix uses zero based indexing, add one
  row <- mask@i + 1
  col <- mask@j + 1

  # need to loop over index of indexes
  # double looping over i and j here feels intuitive
  # but is incorrect
  for (idx in seq_along(row)) {
    i <- row[idx]
    j <- col[idx]

    z_ij <- sum(u[i, ] * d * v[j, ])
    zx[i] <- zx[i] + x[j] * z_ij
  }

  zx
}

# how to calculate just one element of the reconstructed
# data using the SVD

i <- 6
j <- 4

sum(s$u[i, ] * s$d * s$v[j, ])
Z[i, j]

# the whole masked matrix multiply

Z <- s$u %*% diag(s$d) %*% t(s$v)
out <- drop((Z * Y) %*% x)

# check that we did this right
all.equal(
  masked_svd_times_x(s, Y, x),
  out
)
```

This is gonna be painfully slow in R so we rewrite in C++

```cpp
#include <RcppArmadillo.h>

using namespace arma;
```

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]

vec masked_svd_times_x_impl(
  const mat& U,
  const rowvec& d,
  const mat& V,
  const vec& row,
  const vec& col,
  const vec& x) {

  int i, j;
  double z_ij;

  vec zx = zeros<vec>(U.n_rows);

  for (int idx = 0; idx < row.n_elem; idx++) {

    i = row(idx);
    j = col(idx);

    // % does elementwise multiplication in Armadillo
    // accu() gives the sum of elements of resulting vector
    z_ij = accu(U.row(i) % d % V.row(j));

    zx(i) += x(j) * z_ij;
  }

  return zx;
}
```

```r
# wrap with slightly nicer interface
masked_svd_times_x_cpp <- function(s, mask, x) {
  drop(masked_svd_times_x_impl(s$u, s$d, s$v, mask@i, mask@j, x))
}
```

```r
bench::mark(
  masked_svd_times_x_cpp(s, Y, x),
  masked_svd_times_x(s, Y, x)
)
```

Now we also need to work out $(Z_t^T x)_i$, which I am hella blanking on.

```r
library(Matrix)
library(RSpectra)
```

```r
library(testthat)

set.seed(27)

# create a random 8 x 12 sparse matrix with 30 nonzero entries
M <- rsparsematrix(8, 11, nnz = 20)
M

s <- svds(M, 3)

Y <- M != 0
mask <- as(M, "TsparseMatrix")

u <- s$u
d <- s$d
v <- s$v


Z <- u %*% diag(d) %*% t(v)

Zt2 <- v %*% diag(d) %*% t(u)

expect_equal(Zt2, t(Z))

x <- rnorm(8)
Ztx <- t(Z * Y) %*% x

ztx <- numeric(nrow(v))

# lgTMatrix uses zero based indexing, add one
row <- mask@i + 1
col <- mask@j + 1

# need to loop over index of indexes
# double looping over i and j here feels intuitive
# but is incorrect
for (idx in seq_along(row)) {

  # here's where we do the row / column swap
  i <- row[idx]
  j <- col[idx]

  z_ij <- sum(u[i, ] * d * v[j, ])
```

```
  ztx[j] <- ztx[j] + x[i] * z_ij
}


Ztx
ztx


impl_result <- p_omega_ztx_impl(u, d, v, mask@i, mask@j, x)


testthat::expect_equivalent(
  drop(ztx),
  drop(Ztx)
)
```

```
# add the upper triangle back to Y since we're in the citation
# setting
Y <- Y | upper.tri(Y)


obs_Zt <- t(Z * Y)                        # project SVD onto observed matrix


Y <- as(Y, "TsparseMatrix")
impl_result <- p_omega_ztx_impl(s$u, s$d, s$v, Y@i, Y@j, x)


expect_equal(
  drop(impl_result),
  drop(obs_Zt %*% x)
)
```

START MISC

$Z$ is an $n \times d$ matrix. Thus we must have $x \in \mathbb{R}^n$, and then $Z^T x \in \mathbb{R}^d$.

$$(Z^T x)_i = \sum_{j=1}^{n} Z_{ij}^T \cdot x_j \tag{51}$$

Also note that

$$Z_{ij}^T = Z_{ji} \tag{52}$$

$$= \sum_{k=1}^{r} d_k U_{jk} V_{ki} \tag{53}$$

Putting these together we find

$$(Z^T x)_i = \sum_{j=1}^{n} Z_{ij}^T \cdot x_j \tag{54}$$

$$= \sum_{j=1}^{n} \sum_{k=1}^{r} d_k U_{jk} V_{ki} \cdot x_j \tag{55}$$

$$= \sum_{j=1}^{n} U_{j\cdot}^T (DV)_{i\cdot} \cdot x_j \tag{56}$$

END MISC

TODO:

- the tranpose multiplication
- the average singular value calculation

Recall that to calculate the average singular value we want

$$||\tilde{M}^{(t)}||_F^2 = ||M||_F^2 + \sum_{i=1}^{r} \lambda_i^2 - ||Z_t \odot Y||_F^2 \tag{57}$$

This is the other computationally intensive part so let's write it up in Armadillo as well

```cpp
#include <RcppArmadillo.h>

using namespace arma;

// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]

double p_omega_f_norm_impl(
    const mat& U,
    const rowvec& d,
    const mat& V,
    const vec& row,
    const vec& col) {

  int i, j;
  double total = 0;

  for (int idx = 0; idx < row.n_elem; idx++) {

    i = row(idx);
    j = col(idx);
```

```
    total += accu(U.row(i) % d % V.row(j));
  }


  return total;
}
```

```
# wrap with slightly nicer interface
p_omega_f_norm_cpp <- function(s, mask) {
  p_omega_f_norm_impl(s$u, s$d, s$v, mask@i, mask@j)
}
```

```
all.equal(
  sum(Z * Y),
  p_omega_f_norm_cpp(s, Y)
)
```

**A naive solution: the epsilon trick**

issue: we've moved back into dense computation land

**The memory efficient version**

asdf

TODO: don't coerce to explicit Matrix class as that is not recommended apparently. alternatives?

**Fully C++ implementation: Dense**

```
#include <RcppArmadillo.h>


using namespace arma;


// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]
Rcpp::List AdaptiveInitialize(const sp_mat& M, const int r) {

  // coerce to double to avoid integer division
  double p_hat = static_cast<double>(M.n_nonzero) / (M.n_cols * M.n_rows);


  sp_mat MtM = M.t() * M;
  sp_mat MMt = M * M.t();
```

```cpp
  sp_mat sigma_p = MtM / pow(p_hat, 2) - (1 - p_hat) * diagmat(MtM);
  sp_mat sigma_t = MMt / pow(p_hat, 2) - (1 - p_hat) * diagmat(MMt);

  mat U_p, V_p, U_t, V_t;
  vec s_p, s_t;

  svds(U_p, s_p, V_p, sigma_p, r);
  svds(U_t, s_t, V_t, sigma_t, r);

  // TODO: this is still the eigenvalue calculation
  double alpha = (sum(sigma_p.diag()) - sum(s_p)) / (M.n_cols - r);

  vec lambda_hat = sqrt(s_p - alpha) / p_hat;

  mat U_m, V_m;
  vec s_m;

  svds(U_m, s_m, V_m, M, r);

  // sum(A % B) finds the diag(A^T B) / the diagonal of the cross product
  // sum() does a *column-wise sum*, % an elementwise multiplication
  rowvec u_sign = sign(sum(U_m % U_t));
  rowvec v_sign = sign(sum(V_m % V_p));

  rowvec s_hat = u_sign % v_sign;
  lambda_hat = lambda_hat % conv_to< vec >::from(s_hat);

  return Rcpp::List::create(Rcpp::Named("u") = U_t,
                            Rcpp::Named("d") = lambda_hat,
                            Rcpp::Named("v") = V_p);
}
```

```cpp
#include <RcppArmadillo.h>

using namespace arma;

// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]
Rcpp::List AdaptiveImpute(const sp_mat& M, const int r) {

  const double EPSILON = 0.000001;
```

```r
  s <- adaptive_initialize(M, r)
  Z <- s$u %*% diag(s$d) %*% t(s$v)  # line 1
  delta <- Inf

  while (delta > epsilon) {

    y <- as(M, "lgCMatrix")  # indicator if entry of M observed
    M_tilde <- M + Z * (1 - y)  # line 3

    svd_M <- svds(M_tilde, r)

    u_hat <- svd_M$u  # line 4
    v_hat <- svd_M$v  # line 5

    d <- ncol(M)

    alpha <- (sum(M_tilde^2) - sum(svd_M$d^2)) / (d - r)  # line 6

    lambda_hat <- sqrt(svd_M$d^2 - alpha)  # line 7

    Z_new <- u_hat %*% diag(lambda_hat) %*% t(v_hat)

    delta <- sum((Z_new - Z)^2) / sum(Z^2)
    Z <- Z_new

    print(glue::glue("delta: {round(delta, 8)}, alpha: {round(alpha, 3)}"))
  }

  Z
}
```

## References

Bates, Douglas. 2005. "Introduction to the Matrix Package." https://cran.r-project.org/web/packages/Matrix/vignettes/Introduction.pdf.

Bro, Rasmus, Evrim Acar, and Tamara G. Kolda. 2007. "Resolving the Sign Ambiguity in the Singular Value Decomposition," 18.

Cho, Juhee, Donggyu Kim, and Karl Rohe. 2015. "Asymptotic Theory for Estimating the Singular Vectors and Values of a Partially-Observed Low Rank Matrix with Noise." *arXiv:1508.05431 [Stat]*, August. http://arxiv.org/abs/1508.05431.

———. 2018. "Intelligent Initialization and Adaptive Thresholding for Iterative Matrix Completion; Some Statistical and Algorithmic Theory for Adaptive-Impute." *Journal of Computational and Graphical Statistics*, September, 1–26. https://doi.org/10.1080/10618600.2018.1518238.

Maechler, Martin, and Douglas Bates. 2006. "2nd Introduction to the Matrix Package." https://cran.r-project.org/web/packages/Matrix/vignettes/Intro2Matrix.pdf.

Mazumder, Rahul, Trevor Hastie, and Robert Tibshirani. 2010. "Spectral Regularization Algorithms for Learning Large Incomplete Matrices." https://web.stanford.edu/~hastie/Papers/mazumder10a.pdf.