Final Exam: SVD and the Netflix Problem

Gregory Smetana
ID 1917370
ACM 106a

December 12, 2013

1 Recommender systems and SVD

Recommender systems or recommendation systems are broadly defined as algorithms used to predict the rating an individual would assign to an item. For example, an online vendor may want to determine the preferences of its users in order to recommend new products (and increase its sales). These preferences can be thought of as being stored in a matrix with rows corresponding to items in the vendor's catalogue and columns corresponding to each user; the (i, j)th entry of this matrix would be the rating given to item i by user j. For example, the Netflix data set consists of a matrix with rows corresponding to movies in Netflix's catalogue, columns indexed by users and entries corresponding to a user rating of a particular movie on an integer scale from one to five.

Unfortunately, most entries of this preference matrix are unknown in general; many, if not all, users will not have rated every item in the catalogue. The goal of a recommender system is to obtain a good estimate of the unknown entries of this preference matrix. This is a special case of the matrix completion problem: given partially known matrix M , compute the unknown entries of M . Clearly, this problem is absurdly underdetermined; for example, without any additional assumptions, any value between one and five can be assigned to each unknown entry of the Netflix matrix. One classical approach to the matrix completion problem is to place some additional assumption on the matrix M to be completed, e.g., that M is low-rank, and perform regularization to choose a particular solution. This is not an unreasonable assumption regarding the structure of preference matrices. For example, it is widely theorized and/or assumed that individual users in the Netflix matrix can be represented as a linear combination of a relatively small number of eigencustomers, whose behaviour represent canonical preferences. For example, one eigencustomer may be a user who only watches documentaries, another is someone who only enjoys comedies, etc. Similarly, any movie can be represented as a combination of eigenmovies. Although there may be many canonical users or catalogue items, it is highly likely that the number of such users or items should be much smaller than the dimensions

of the preference matrix (on the order of dozens versus tens of thousand of movies and millions of users in the case of the Netflix matrix). This corresponds to the preference matrix having very small rank (relative to its dimension).

This assumption leads to a natural formulation of the problem as an underdetermined linear system with matrix rank used for regularization. Unfortunately, this requires the solution of an intractable optimization problem and the methods for approximately solving it are well beyond the scope of this course; however, if you are interested this may be covered in ACM 113: Introduction to Optimization. Instead, we will use a classical technique based on singular value decomposition of the preference matrix to obtain our predicted user ratings. This heuristic follows two steps. In the first, values are assigned to the unknown entries of the preference matrix. For example, if the (i, j) entry is unknown, then we may assign a value by taking a combination of the average rating for movie i or the average rating by user j. Once the unknown entries have been assigned a value, we obtain a rank-k prediction matrix using the singular value decomposition of the preference matrix (with filled in entries). The use of low-rank matrices to identify a low-dimensional set of latent variables explaining variance also plays a significant role in the analysis of (incomplete) scientific data, especially that arising from applications in geophysics, and is closely related to principal component analysis; this will likely be discussed at length in ACM/ESE 118: Methods in Applied Statistics and Data Analysis.

2 Description of partial SVD algorithm

The outline of the partial SVD algorithm is similar to the full SVD algorithm

1. Form matrix $A^T A$
2. Compute partial eigendecomposition $A^T A \approx V_k D_k V_k^T$
3. Let $\Sigma_k = D_k^{1/2}$
4. Solve the system $U_k \Sigma_k = A_k V_k$ for the matrix of left singular vectors U_k.

a) Partial eigendecomposition of $A^T A$

To compute the partial eigendecomposition of the symmetric matrix $A^T A$ in Step 2, the Lanczos algorithm in floating point arithmetic of Demmel Algorithm 7.2 is used:

$$
\begin{aligned}
&q_1 = b/\|b\|_2, \beta_0 = 0, q_0 = 0 \\
&\text{for } j = 1 \text{ to } k \\
&\quad z = Aq_k \\
&\quad \alpha_j = q_j^T z \\
&\quad z = z - \sum_{i=1}^{j-1} (z^T q_i) q_i \\
&\quad z = z - \sum_{i=1}^{j-1} (z^T q_i) q_i \\
&\quad \beta_j = \|z\|_2 \\
&\quad \text{if } \beta_j = 0, \text{quit} \\
&\quad q_{j+1} = z/\beta_j \\
&\quad \text{Compute eigenvalues, eigenvectors, and error bounds of } T_k \\
&\text{end}
\end{aligned}
\tag{1}
$$

The $z = z - \sum_{i=1}^{j-1} (z^T q_i) q_i$ steps are to reorthogonalize after loss of orthogonality in floating point arithmetic. If the Lanczos vectors are not kept orthogonal, there may be multiple copies of converged Ritz values.

The algorithm was said to be converged when

$$|\beta_n||v_k(n)| < tol \tag{2}$$

b) Eigenvalues and eigenvectors of T_k

In computing the eigenvalues and eigenvectors of T_k during the Lanczos iteration, QR iteration with a shift was used:

$$
\begin{aligned}
&i = 0 \\
&\text{repeat} \\
&\quad \text{Choose a shift } \sigma_i \\
&\quad \text{Factor } A_i - \sigma_i I = Q_i R_i \\
&\quad A_{i+1} = Q_i^T A_i Q_i \\
&\quad i = i + 1 \\
&\text{Until convergence}
\end{aligned}
\tag{3}
$$

Convergence was defined as the off-diagonal entries of A being smaller than tol.

b).1 QR iteration shift

The QR iteration used Wilkinson's shift. This shift is the eigenvalue of the lower right 2×2 submatrix of A_i

$$B = \begin{pmatrix} a_{n-1} & b_{n-1} \\ b_{n-1} & a_n \end{pmatrix} \tag{4}$$

that is closest to a_n. QR iteration with Wilkinson's shift is globally, and at least linearly convergent. It is asymptotically cubically convergent for almost all matrices.

b).2 Computational efficiency

We may take advantage of the tridiagonal structure of T_k to reduce the computational time during the QR iteration. Recall the Implicit Q Theorem:

- Suppose that $Q^T AQ = H$ is unreduced upper Hessenberg. Then columns 2 through n of Q are determined uniquely (up to signs) by the first column of Q

Therefore, to compute the first column of Q_i, we just normalize the first column of $A_i - \sigma_i I$ and choose other columns of Q_i so Q_i is orthogonal and A_{i+1} is unreduced Hessenberg

In practice, implicit single shift QR is known as "chasing the bulge." An example is detailed in Example 4.10 of Demmel. The matrices Q_i^T have the form

$$Q_1^T = \begin{pmatrix} c_1 & s_1 & & & \\ -s_1 & c_1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{pmatrix} \tag{5}$$

where the 2×2 submatrix is a Givens rotation matrix that eliminates the bulge created in the previous step. The cost of one implicit QR iteration for an n by n matrix is $6n^2 + O(n)$

c) Computation of left singular vectors

In Step 4, given the first k singular values of A and the first k right singular vectors, the algorithm computes the first k left singular vectors of A by solving the system

$$U_k \Sigma_k = A_k V_k \tag{6}$$

To show this is correct, start with

$$A = U \Sigma V^T = \sum_{i=1}^{n} \sigma_i u_i v_i^T \tag{7}$$

where

- $U \in \mathbf{R}^{m \times m}$ such that $U^T U = I$
- $V \in \mathbf{R}^{n \times n}$ such that $V^T V = I$
- $\Sigma = \text{Diag}(\sigma_1, \sigma_2, ..., \sigma_n)$ where $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_n \geq 0$.

multiplying by V_k on the right,

$$AV_k = U\Sigma V^T V_k \tag{8}$$

now note that $V^T V_k = 1$ for $i \leq k$ and $V^T V_k = 0$ for $i > k$, so

$$AV_k = U_k \Sigma_k \tag{9}$$

may be solved for the first k left singular vectors.

3 Data Analysis

The MovieLens data set is stored as an 800×500 real integer matrix containing the ratings (on a scale from 1 to 5) of a catalogue of 800 movies by 500 users. The data is highly incomplete (roughly 12% of all possible ratings are known). The SVD-based procedure described above was used to learn the unknown customer ratings (which in turn would be used to make recommendations to our customers).

a) Fill-in

Any unknown entries of the training data matrix M are replaced with either

(a) a random integer between 1 and 5

(b) average movie rating (average of known entries in the row)

(c) average customer rating (average of known entries in column)

(d) average of the two ($0.5\times$(column average + row average))

b) SVD parameters

The stopping criterion of the Lanczos iteration was

$$|\beta_n||v_k(n)| < tol \tag{10}$$

with $tol = 1E-6$ and a maximum of 500 iterations. The stopping criterion of the QR iteration was that the off diagonal entries were smaller than $tol = 1E-6$, or a maximum of 100 iterations. These levels of tolerance were found to produce results that coincided with matlab `svds` to at least the first 4 digits.

c) Evaluation

To evaluate performance, we compare the learned entries of M with those of the test set T. We use the root mean square error to measure accuracy of our predictions

$$RSME = \left(\frac{1}{N_{test}} \sum_{ij:T_{ij} \neq 0} (T_{ij} - [M_k]_{ij})^2 \right)^{1/2} \tag{11}$$

where $N_{test} = |\{ij : T_{ij} \neq 0\}|$

The relative error between each rank-k matrix and the true value of M with unknown entries filled in was also examined, along with the convergence of the singular values and the computational time

d) Results

The ratings were predicted for each value of k in $\{1, 2, 3, 4, 5, \ldots, 25\}$ and each fill-in method described above.

The RMSE of the prediction error as a function of k for each fill-in method is plotted in Figure 1. A plot of the relative error between each rank-k matrix and the true value of M with unknown entries filled in is displayed in Figure 2.

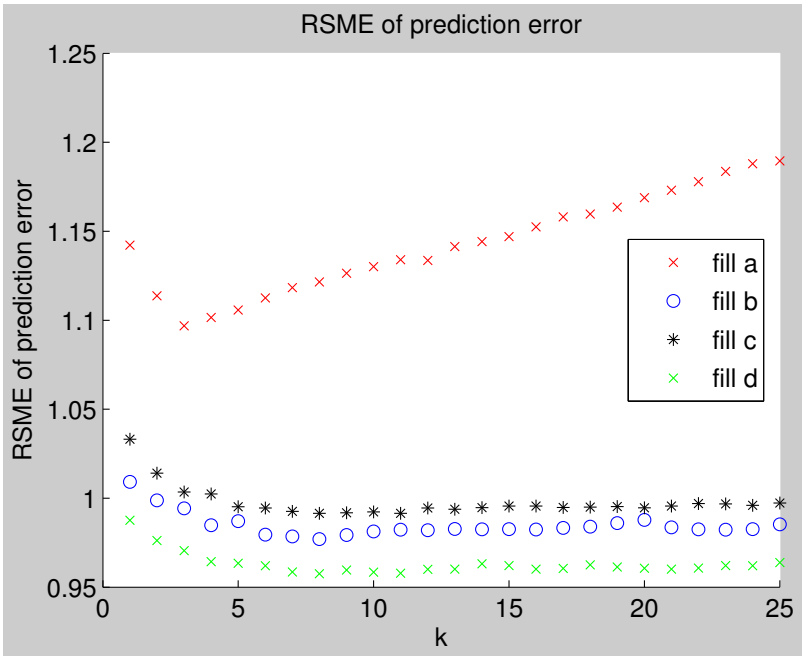

Figure 1

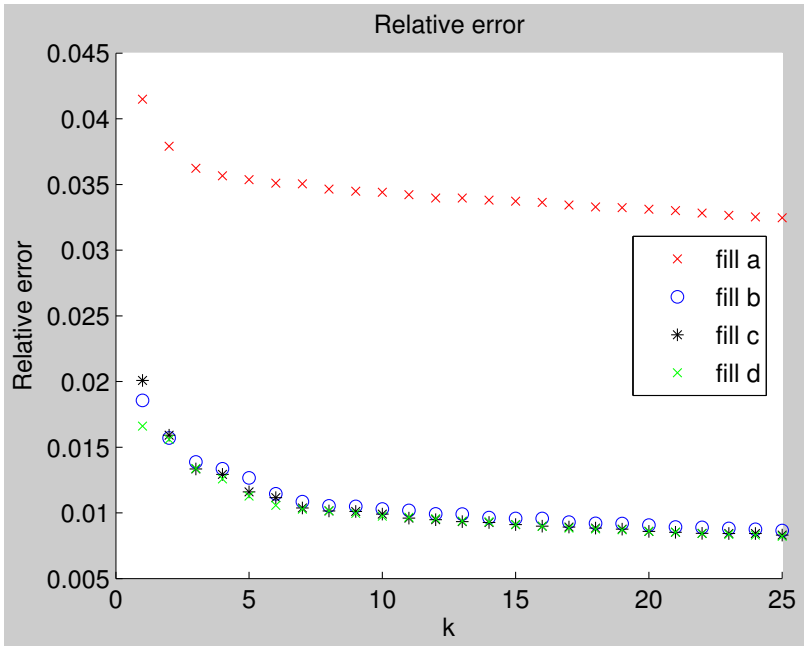

Figure 2

The value of each singular value at each step of the iterative singular value decomposition is plotted for each of the filling methods in Figures 3-6.

The time taken to compute the SVD is plotted vs k for each of the methods in Figure 7.

4 Discussion

Looking at Figure 1, we see that the prediction error of the methods increases in the order: d), b), c), a). Methods b), c), and a) decrease from $k = 1$ to $k = 5$ and then approach a constant value. Method a) decreases from $k = 1$ to $k = 3$ and then increases with k. These results are not surprising. As k increases for the randomly filled matrix, it is using more randomly generated data which is not helpful in predicting unknown entries. The error of methods b), c), and d) include more useful information about the matrix, so they make better predictions. Method d), which uses the row and column average is the most accurate, but also requires the most computation.

Looking at Figure 2, we see that for all methods, the relative error between each rank-k matrix and the true value of M with unknown entries filled decreases with k. This matches the expectation that the quality of the approximation will increase with k. The relative error is highest for the fill method a) because the randomly filled data is not easily spanned by a set of low-rank eigenvectors.

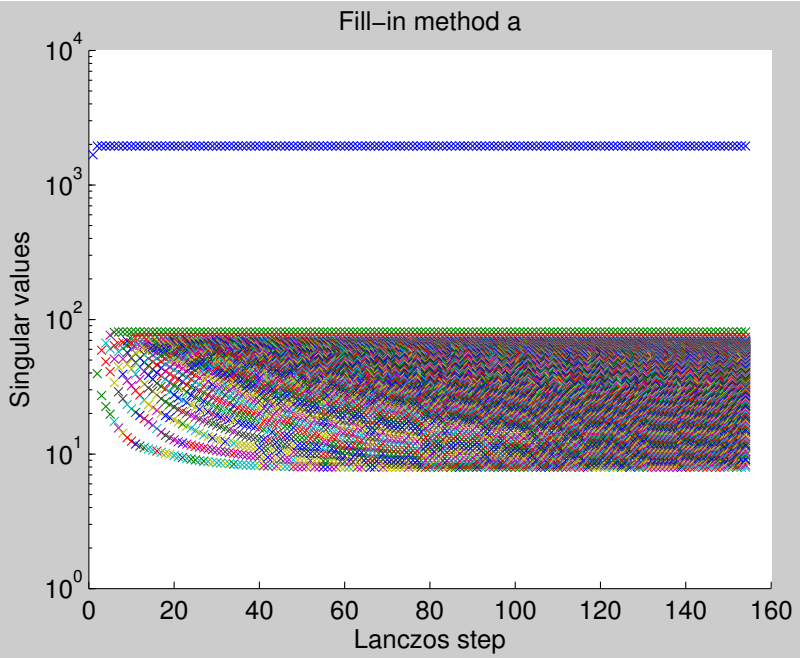

Figure 3

The convergence of each singular value for each fill in method is displayed in Figures 3-6. The singular values of filling methods b), c), and d) converge more quickly than those of method a) and take approximately 50% fewer iterations to meet the stopping criteria than method a). Most of the singular values for methods b), c), and d) are smaller than those of method a). All methods have one very large singular value that converges in only a few iterations.

Looking at Figure 7, we see that the time taken to compute the SVD has a linear relationship with k. This makes sense, because computing each additional singular value and set of singular vectors takes approximately the same number of operations. The filling method a) takes the most time, which is related to the slower rate of convergence discussed earlier. The speeds of methods b), c), and d) are all comparable.

5 Conclusions

By all measures, the random filling of method a) performed the worst. Although filling method d) took more operations to fill in the matrix, it was the most accurate at making predictions. It would depend on the size of your matrix and your accuracy requirements, but filling in the matrix using the average of the movie rating and customer rating is likely the best option because it includes the most useful information in creating a low-rank

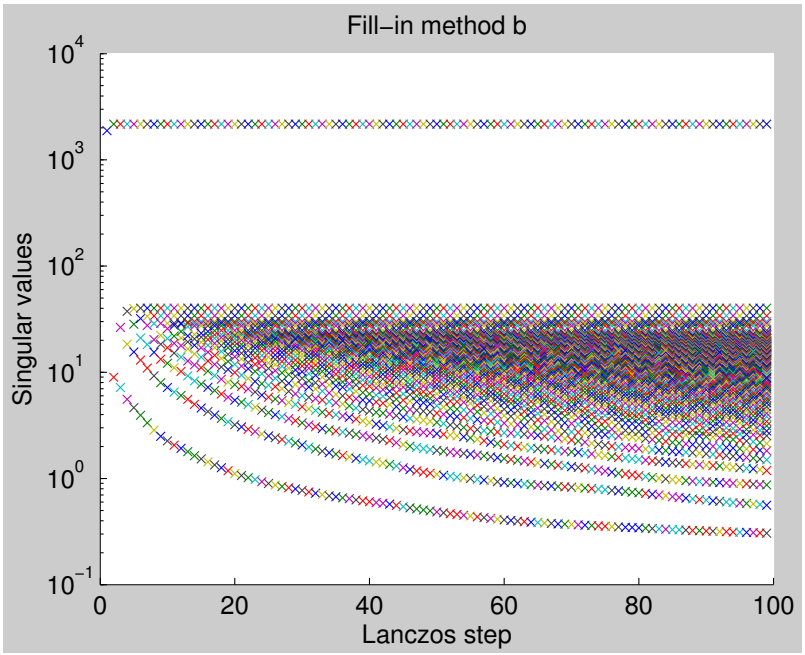

Figure 4

approximation. From Figure 1, we see that there is no point in creating an approximation of higher order than $k = 4$ or $k = 5$ because it will not be any more accurate at making predictions.

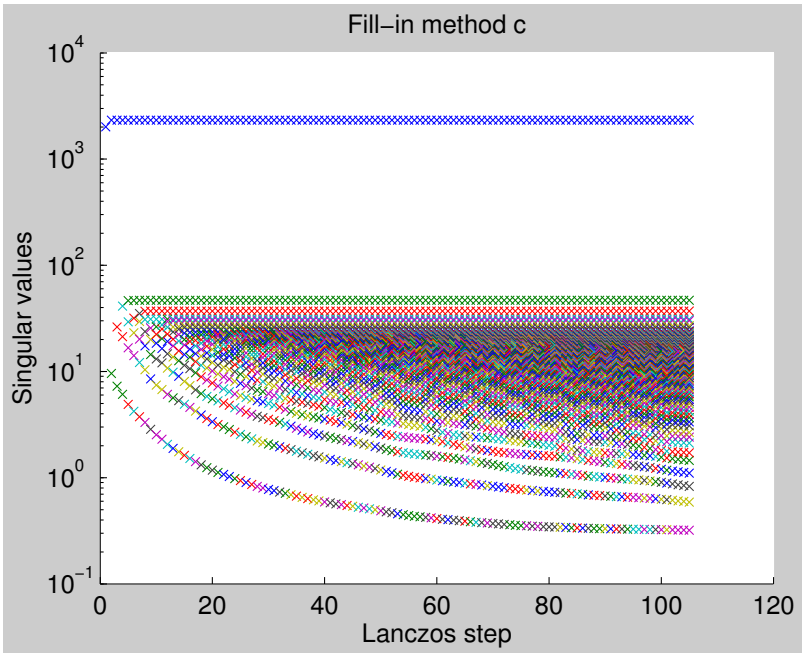

Figure 5

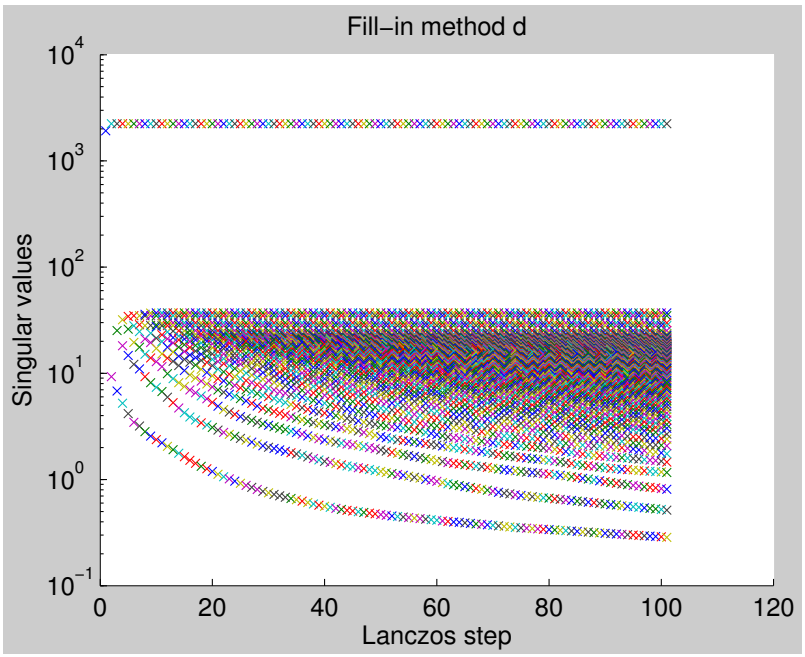

Figure 6

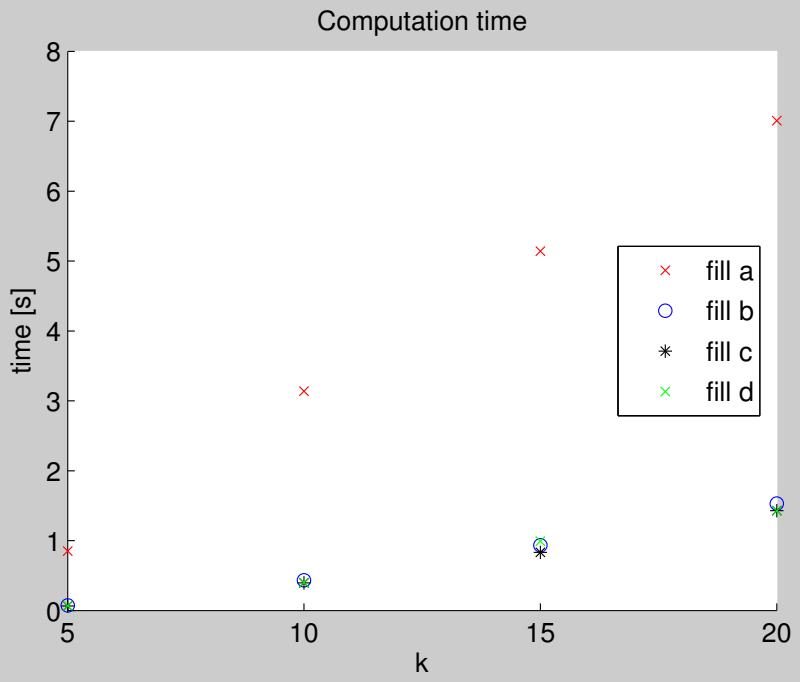
Computation time
time [s]
k
fill a
fill b
fill c
fill d
0
1
2
3
4
5
6
7
8
5
10
15
20

Figure 7

A Smetana_Gregory_1917370_FINAL_DIARY.txt

```
run('Smetana_Gregory_1917370_FINAL.m');
diary off
```

B Smetana_Gregory_1917370_FINAL.m

```
%Smetana_Gregory_1917370_FINAL

clear;
clc;
close all;
path(path,'export_fig/');

%% Load data
M = load('movies.mat','movies_train');
M = full(M.movies_train);
T = load('movies.mat','movies_test');
T = full(T.movies_test);
[m,n]=size(M);

%% Fill in unknown entries
M_a = fill_a(M);
M_b = fill_b(M);
M_c = fill_c(M);
M_d = fill_d(M);

%% Compute SVD for k=25

[U_a, S_a, V_a, svs_a] = partial_svd(M_a,25);
[U_b, S_b, V_b, svs_b] = partial_svd(M_b,25);
[U_c, S_c, V_c, svs_c] = partial_svd(M_c,25);
[U_d, S_d, V_d, svs_d] = partial_svd(M_d,25);

%% Timing
kt = [5,10,15,20];
t_a = zeros(4,1);
t_b = zeros(4,1);
t_c = zeros(4,1);
t_d = zeros(4,1);
for i = 1:4
    tic
    partial_svd(M_a,kt(i));
    t_a(i) = toc;

    tic
```

```
    partial_svd(M_b,kt(i));
    t_b(i) = toc;

    tic
    partial_svd(M_c,kt(i));
    t_c(i) = toc;

    tic
    partial_svd(M_d,kt(i));
    t_d(i) = toc;
end

%% Plot
Mk_a = zeros(m,n);
Mk_b = zeros(m,n);
Mk_c = zeros(m,n);
Mk_d = zeros(m,n);

f1 = figure;
hold all;
f2 =figure;
hold all;
for k=1:25
    Mk_a = Mk_a + U_a(:,k) * S_a(k) * V_a(:,k)';
    Mk_b = Mk_b + U_b(:,k) * S_b(k) * V_b(:,k)';
    Mk_c = Mk_c + U_c(:,k) * S_c(k) * V_c(:,k)';
    Mk_d = Mk_d + U_d(:,k) * S_d(k) * V_d(:,k)';

    figure(f1);

    plot(k, compute_rsme(T,Mk_a),'rx')
    plot(k, compute_rsme(T,Mk_b),'bo')
    plot(k, compute_rsme(T,Mk_c),'k*')
    plot(k, compute_rsme(T,Mk_d),'gx')

    figure(f2);

    plot(k, norm(Mk_a - M_a)/ norm(M_a), 'rx');
    plot(k,  norm(Mk_b - M_b)/ norm(M_b),'bo')
    plot(k,  norm(Mk_c - M_c)/ norm(M_c),'k*')
    plot(k, norm(Mk_d - M_d)/ norm(M_d),'gx')

end

f3 = figure;
hold all;
for i=1:length(svs_a)
    plot(i, svs_a(:,i),'x')
end
set(gca,'yscale','log')
```

```
f4 = figure;
hold all;
for i=1:length(svs_b)
    plot(i, svs_b(:,i),'x')
end
set(gca,'yscale','log')

f5 = figure;
hold all;
for i=1:length(svs_c)
    plot(i, svs_c(:,i),'x')
end
set(gca,'yscale','log')

f6 = figure;
hold all;
for i=1:length(svs_d)
    plot(i, svs_d(:,i),'x')
end
set(gca,'yscale','log')
%%
f7 = figure;
hold all;
plot(kt, t_a,'rx')
plot(kt, t_b,'bo')
plot(kt, t_c,'k*')
plot(kt, t_d,'gx')

%% label plots
figure(f1)
h_legend = legend('fill a','fill b','fill c','fill d','Location','best');
set(h_legend,'FontSize',12);
set(gca,'FontSize',12);
xlabel('k','FontSize',12);
ylabel('RSME of prediction error','FontSize',12);
title('RSME of prediction error', 'FontSize',12);
filename = ['report/rsme.pdf'];
export_fig(filename)


figure(f2)
h_legend = legend('fill a','fill b','fill c','fill d','Location','best');
set(h_legend,'FontSize',12);
set(gca,'FontSize',12);
xlabel('k','FontSize',12);
ylabel('Relative error','FontSize',12);
title('Relative error', 'FontSize',12);
filename = ['report/relative.pdf'];
export_fig(filename)
```

```
figure(f3)
set(gca,'FontSize',12);
xlabel('Lanczos step','FontSize',12);
ylabel('Singular values','FontSize',12);
title('Fill-in method a', 'FontSize',12);
filename = ['report/sv_a.pdf'];
export_fig(filename)

figure(f4)
set(gca,'FontSize',12);
xlabel('Lanczos step','FontSize',12);
ylabel('Singular values','FontSize',12);
title('Fill-in method b', 'FontSize',12);
filename = ['report/sv_b.pdf'];
export_fig(filename)

figure(f5)
set(gca,'FontSize',12);
xlabel('Lanczos step','FontSize',12);
ylabel('Singular values','FontSize',12);
title('Fill-in method c', 'FontSize',12);
filename = ['report/sv_c.pdf'];
export_fig(filename)

figure(f6)
set(gca,'FontSize',12);
xlabel('Lanczos step','FontSize',12);
ylabel('Singular values','FontSize',12);
title('Fill-in method d', 'FontSize',12);
filename = ['report/sv_d.pdf'];
export_fig(filename)
%%
figure(f7)
h_legend = legend('fill a','fill b','fill c','fill d','Location','best');
set(h_legend,'FontSize',12);
set(gca,'FontSize',12);
xlabel('k','FontSize',12);
ylabel('time [s]','FontSize',12);
title('Computation time', 'FontSize',12);
filename = ['report/time.pdf'];
export_fig(filename)

%% save
save('workspace.mat','M_a', 'M_b', 'M_c', 'M_d',...
  'S_a', 'S_b', 'S_c', 'S_d', 'U_a', 'U_b', 'U_c',...
  'U_d', 'V_a', 'V_b', 'V_c', 'V_d');
```

C fill_a.m

```
function [ M ] = fill_a( M )
%FILL_A fills unknown entries of M with random integer between 1 & 5

z = find(M<1);
M(z) = ceil(5 * rand(length(z),1));

end
```

D fill_b.m

```
function [ M ] = fill_b( M )
%FILL_B fills unknown entries of M with row average

[m,n]=size(M);
rowavg = sum(M,2)./sum(M>0,2);
for i = 1:m
    M(i,M(i,:)<1) = rowavg(i);
end

end
```

E fill_c.m

```
function [ M ] = fill_c( M )
%FILL_C fills unknown entries of M with column average

[m,n]=size(M);
colavg = sum(M,1)./sum(M>0,1);
for i = 1:n
    M(M(:,i)<1,i) = colavg(i);
end

end
```

F fill_d.m

```
function [ M ] = fill_d( M )
%FILL_D fills unknown entries of M with average of column and row averages

rowavg = sum(M,2)./sum(M>0,2);
colavg = sum(M,1)./sum(M>0,1);
z = find(M<1);
[i,j] = ind2sub(size(M),z);
for k = 1:length(i)
```

```
    M(i(k), j(k)) = 1/2*(rowavg(i(k)) + colavg(j(k)));
end

end
```

G compute_rsme.m

```
function [ rsme ] = compute_rsme( T, M )
%RSME compute root mean square

Ntest = sum(sum(T > 0)); % non zero entries;

z = find(T);
[i,j] = ind2sub(size(T),z);
rsme = 0;
for k = 1:length(i)
    rsme = rsme + (T(i(k),j(k)) - M(i(k),j(k)))^2;
end

rsme = sqrt(1/Ntest *rsme);
end
```

H partial_svd.m

```
function [U,S,V, svs] = partial_svd( A, k )
%PARTIAL_SVD calculate rank-k approximation of A

[V,D, ev] = partial_eig(A'*A,k);
S = D.^.5;
svs = ev.^.5;
U = A* V;
for i = 1:k
    U(:,i) = U(:,i)/S(i);
end

end
```

I partial_eig.m

```
function [ Vk, Dk, evs ] = partial_eig( A, k, tol, maxiterations)
%PARTIAL_EIG iteratively computes the partial eigendecomposition of a
%symmetric matrix

if nargin < 4
```

```
    maxiterations = 500;
    tol = 1E-6;
end

[n,~]=size(A);
qj = rand(n,1);
qj = qj/norm(qj);
Q = qj;

i=0;
err = 1E9;
while i < maxiterations && err > tol
  i = i+1;

  z = A*qj;
  alpha(i) = qj'*z;
  % reorthogonalize twice
  z = z - Q*(Q'*z);
  z = z - Q*(Q'*z);
  beta(i) = norm(z);

  qj = z/beta(i);
  Q = [Q,qj];

  % compute ritz values/vectors
  [V,D] = symtri_eig(alpha, beta(1:i-1));
  if(nargout > 2 )
    evs(1:i,i) = D;
  end
  if( i > k)
    err = beta(i)*norm(V(i,k));
  end

end
Vk = Q(:,1:i) * V;
Vk = Vk(:,1:k);   % return approx
Dk = D(1:k);
end
```

J symtri_eig.m

```
function [ V, D ] = symtri_eig( a, b, tol, maxiterations )
%SYMTRI_EIG eigenvectors and eigenvalues of symmetric tridiagonal matrix
% variable a is on the diagonal, b is above/below

if nargin < 4
    maxiterations = 100;
    tol = 1E-6;
```

```
end

n = length(a);
c = zeros ( 1, n );
s = zeros ( 1, n );
V = eye(n);
shift =0;

for current = n:-1:2

    iteration = 0;
    while( abs(b(current-1)) > tol && iteration < maxiterations)

        % wilkinson shift (p213)
        d = (a(current-1)-a(current))/2;
        if( d >= 0)
            sigma = a(current) + d - sqrt(b(current-1)^2 + d^2);
        else
            sigma = a(current) + d + sqrt(b(current-1)^2 + d^2);
        end
        shift = shift + sigma;
        a(1:current) = a(1:current) - sigma;

        % implicit single shift QR (p169)
        b_old = b(1);
        for i = 2:current
            % givens rotation (p122)
            root = sqrt ( a(i-1)^2 + b_old^2 );
            c(i) = a(i-1) / root;
            s(i) = - b_old / root;
            product = [c(i) -s(i); s(i) c(i)] * [a(i-1) b(i-1); b(i-1) a(i)];
            a(i-1)=root;
            b(i-1)=product(1,2);
            a(i) = product(2,2);
            if ( i < current )
                b_old = b(i);
                b(i) = c(i)*b(i);
            end;
        end;

        a(1) = c(2)*a(1) - s(2)*b(1);
        b(1) = -s(2)*a(2);

        for i = 2:current-1
            a(i) = -s(i+1)*b(i) + c(i)*c(i+1)*a(i);
            b(i) = -s(i+1)*a(i+1);
        end;
        a(current) = c(current)*a(current);

       for i = 2 : current
```

```
            first = V(:,i-1) * c(i) - V(:,i) * s(i);
            V(:,i) = s(i) * V(:,i-1) + c(i) * V(:,i);
            V(:,i-1) = first;
        end;

        iteration = iteration + 1;
    end;
    D(current) = a(current)+ shift;
end
D(1) = a(1)+ shift;
```