Singular Value Decomposition and its numerical computations

Wen Zhang, Anastasios Arvanitis and Asif Al-Rasheed

ABSTRACT

The Singular Value Decomposition (SVD) is widely used in many engineering fields. Due to the important role that the SVD plays in real-time computations, we try to study its numerical characteristics and implement the numerical methods for calculating it. Generally speaking, there are two approaches to get the SVD of a matrix, i.e., direct method and indirect method. The first one is to transform the original matrix to a bidiagonal matrix and then compute the SVD of this resulting matrix. The second method is to obtain the SVD through the eigen pairs of another square matrix. In this project, we implement these two kinds of methods and develop the combined methods for computing the SVD. Finally we compare these methods with the built-in function in Matlab (svd) regarding timings and accuracy.

1. INTRODUCTION

The singular value decomposition is a factorization of a real or complex matrix and it is used in many applications. Let A be a real or a complex matrix with m by n dimension. Then the SVD of A is: $A = U\Sigma V^T$ where U is an m by m orthogonal matrix, Σ is an m by n rectangular diagonal matrix and V^T is the transpose of V n × n matrix. The diagonal entries of Σ are known as the singular values of A. The m columns of U and the n columns of V are called the left singular vectors and right singular vectors of A, respectively. Both U and V are orthogonal matrices. In this project we assume that the matrix A is real. There are a few of methods that can be used to compute the SVD of a matrix and they will be discussed and analyzed.

APPLICATIONS

As we discussed before the singular value decomposition is very useful and can be used in many application areas. We will only mention a few.

- Digital Signal Processing:

  The SVD has applications in digital signal processing, as a method for noise reduction. The central idea is to let a matrix A represent the noisy signal, compute the SVD, and then discard small singular values of A. It can be shown that the small singular values mainly represent the noise, and thus the rank-k matrix A_k represents a filtered signal with less noise.

- Image Processing:

The SVD has also applications in image processing and specifically in image compression. Computer technology these days is most focused on storage space and speed. One way to help cure this problem is Singular Value Decomposition. Singular Value Decomposition can be used in order to reduce the space required to store images. Image compression deals with the problem of reducing the amount of data required to represent a digital image. Compression is achieved by the removal of three basic data redundancies:

1) coding redundancy, which is present when less than optimal;
2) interpixel redundancy, which results from correlations between the pixels;
3) psychovisual redundancies, which is due to data that is ignored by the human visual.

When an image is SVD transformed, it is not compressed, but the data take a form in which the first singular value has a great amount of the image information. With this, we can use only a few singular values to represent the image with little differences from the original. With this method we save valuable disc space.

- Mechanical Vibrations

The Frequency Response Function of a mechanical system can be decomposed using SVD at a specific frequency. The plot of the log-magnitude of the singular values as a function of frequency is called Complex Mode Indicator Function (CMIF) and is very helpful in locating the modal parameters of the system (natural frequencies, mode shapes, modal participation factors).

$$[H(j\omega_k)] = [U_k][S_k][V_k]^H$$

The first CMIF is considered the plot of the largest singular values in each frequency. Distinct peaks in the first CMIF indicate the existence of a mode. The orthonormal columns of $[U_k]$ are the left singular vectors of $[H(j\omega_k)]$ and represent the mode shape vectors. The orthonormal columns of $[V_k]$ are the right singular vectors and represent the modal participation factors.

2. IMPLEMENTATIONS OF DIFFERENT METHODS

2.1 INDIRECT METHOD

Suppose A is a $m \times n$ matrix, the SVD of A is $A = U\Sigma V^T$ where U and V are orthogonal matrices. We know that if $m < n$ then after getting the eigen values and eigen vectors of A^TA, the eigen vectors can be orthogonalized to form V and it is straightforward to get U through the formula: $U_i = AV_i/\sigma_i$ where σ_i's are the singular values of A and the square roots of eigen values of A^TA. Using the Matlab command 'eig' to get the eigen values, they are listed in ascending order and the SVDecom function we programmed will list the singular values in descending order, which is the same to that obtained by the built-in function ‘svd’. Without loss of generality, we implemented the function in both A^TA and AA^T ways. The second method is preferred when $m > n$. Another key point in the implementation of SVDecom is that, if A is rank deficient, which usually happens in case $m > n$, there are not enough V_i to get U. The strategy we used is to set the elements in the rest $m - n$ columns of U all ones and then orthonormalize it via Gram-Schmidt algorithm. All in all, different strategies can be combined to treat variant cases in programming the related subroutines.

The Matlab code for calculating the SVD via the A^TA/AA^T eigenvalue decomposition is in Table 2.1 where the function is named SVDecom. As we discussed, when $m > n$, the AA^T approach is employed in this function. The eigen vectors of this matrix are orthogonalized to form the orthogonal matrix U. The first n columns of U are utilized to obtain matrix V. Certainly, if $m < n$, we can also use this approximation. The first m columns of V are got via $V_i = A^TU_i/\sigma_i$. The left $n - m$ columns of V are first set to all-ones vectors, then we use Gram-Schmidt process to orthonormalize them in order to get orthogonal matrix V. If there are zero singular values, we will first set the corresponding columns all-one vectors, and then use double Gram-Schmidt process to get the orthogonal matrix.

Table 2.1 SVDecom Function

```
Function [u,d,v]=SVDecom(A)
[m,n]=size(A); sinflag=0;
if (m>n)
   [u,d]=eig(A*A’);          u=GSO(u);u=fliplr(u);
   d1(1:n,n+1:m)=zeros(n,m-n);
   dd=fliplr(diag(d.^(0.5))’)’;
   d1(1:n,1:n)=diag(dd(1:n));d=d1;
   for i=1:n
      if (d(i,i)~=0)
         v(:,i)=A’*u(:,i)/d(i,i);
      else
         sinflag=1;   v(:,i)=ones(n,1);
      end
   end
   v=GSO(v);
   if (sinflag==1)
      v=GSO(v);
   end
   d=d’;
end
```

```
 if (m<n||m==n)
   [v,d]=eig(A'*A);
    v=GSO(v);                v=fliplr(v);
   d1=zeros(m,n);
   dd=fliplr(diag(d.^(0.5))');
   d1(1:m,1:m)=diag(dd(1:m));
   d=d1;
   for i=1:m
      if(d(i,i)~=0)
         u(:,i)=A*v(:,i)/d(i,i);
      else
         sinflag=1;
         u(:,i)=ones(m,1);
      end
   end
   u=GSO(u);
   if (sinflag==1)
      u=GSO(u);
   end
end
```

The double Gram-Schmidt process is implemented in function ‘GSO’ which is listed in Table 2.2. It first orthogonalize the vectors twice and then normalize them.

2.2 DIRECT METHOD

2.2.1 BIDIAGONALIZATION AND CALCULATION OF THE SVD

The direct method is to transform the matrix A to a bidiagonal matrix first. Then through getting the SVD of the resulting bidiagonal matrix, we find the SVD of A. The code for reducing the matrix to the bidiagonal form is in Table 2.4. The corresponding function 'BiDiag' is programmed according to the instructions in [2] from Page 394 to Page 400. It employs the Householder function which gets the Householder matrix of the related vector and is programmed as in Table 2.3.

Table 2.2 GSO Function

```
function [Q]=GSO(A)
[n,m]=size(A);
Q1(:,1)=A(:,1);
for j=2:m
   Q1(:,j)=A(:,j);
   for i=1:j-1
      x=Q1(:,i); a=A(:,j); qnorm=x'*x;
      if (qnorm~=0)
        Q1(:,j)=Q1(:,j)-(a'*x)/qnorm*x;
      end
   end
end
Q(:,1)=Q1(:,1);
for j=2:m
   Q(:,j)=Q1(:,j);
   for i=1:j-1
      x=Q(:,i); a=Q1(:,j); qnorm=x'*x;
      if(qnorm~=0)
         Q(:,j)=Q(:,j)-(a'*x)/qnorm*x;
      end
   end
end
for i=1:m
   qnorm=Q(:,i);x=(qnorm'*qnorm);
   if(x~=0)
      Q(:,i)=Q(:,i)/(x^(0.5));
   end
end
```

Table 2.3 Housholder

```
function Q=Householder(x)
[m,n]=size(x);  tau=sign(x(1))*norm(x,2);
u=x;u(1)=u(1)+tau;  y=2/(norm(u,2)^2);
Q=eye(m,m)-y*u*u';
```

Note that if $m > n$, there will be one more Householder transforms on the column of A. If $m < n$, there are two more right transformation matrices which are used to make the Householder transforms on rows. Generally, it is more convenient to reduce a matrix with dimension $m > n$ to a bidiagonal matrix just as that introduced in the textbook. By doing so, it will not lose generality because if $m < n$ the corresponding transforms can be utilized to A^T and get the same factorizations by transposing the resulting matrices. After using function BiDiag to matrix A, it is factorized to $A = UBV^T$ where U and V are orthogonal matrices and B is a bidiagonal matrix. Take the $m > n$ case for example, we get $A = UBV^T$ where B is also m by n. The singular values of B is same to A. And if the SVD of B is gained: $B = P\Sigma Q^T$, then we have $A = UP\Sigma Q^T V^T$. Therefore the SVD of A is: $A = U_1 \Sigma {V_1}^T$, where $U_1 = UP$, $V_1 = VQ$. So the main task is focus on finding the SVD of B. We know B has the form $\begin{bmatrix} \hat{B} \\ 0_d \end{bmatrix}$ where $\hat{B}$ is a square n by n bidiagonal matrix and 0_d is a $(m-n)$ by n zero matrix. If we know $\hat{B} = \hat{P}\hat{\Sigma}\hat{Q}^T$ then what is the relation between Σ and $\hat{\Sigma}$, P and $\hat{P}$, Q and $\hat{Q}$? The relations are as follows: $Q = \hat{Q}$, $\Sigma = \begin{bmatrix} \hat{\Sigma} \\ 0_d \end{bmatrix}$, $P = \begin{bmatrix} \hat{P} & 0_1 \\ 0_2 & I_{m-n} \end{bmatrix}$. Here 0_1 is a $n \times (m-n)$ zero matrix and 0_2 is a $(m-n) \times n$

zero matrix. I_{m-n} is $(m-n)$ by $(m-n)$ identity matrix. There are many optional methods to get the SVD of $\hat{B}$. We can calculate it by the built-in function or by the SVDecom function we programmed. Iterative algorithms such as Francis and Jacobi methods can also be employed. From the literature [1], we know that if $B_{n\times n}$ has entries $\begin{bmatrix} b_1 & b_2 & & & O \\ & b_3 & b_4 & & \\ & & \ddots & & \\ & & & b_{2n-3} & b_{2n-2} \\ O & & & & b_{2n-1} \end{bmatrix}$ and SVD of B is $B = U\Sigma V^T$ with $\Sigma = diag\{\sigma_1, \sigma_2, \dots, \sigma_n\}$, $V = [v_1, v_2, \dots, v_n]$, and $U = [u_1, u_2, \dots, u_n]$. Then the symmetric matrix

Table 2.4 Bidiagonalization

function [u,b,v]=BiDiag(A)
[m,n]=size(A); n1=min(m,n);
u1=Householder(A(:,1));
b=u1*A;A1=b';a=A1(2:n,1);
v2=Householder(a); v1(2:n,2:n)=v2';
v1(1,1)=1; b=b*v1; u=u1;v=v1;
for i=2:n1-2
a=b(i:m,i); clear u1;
u1(1:i-1,1:i-1)=eye(i-1);
u1(i:m,i:m)=Householder(a);
b=u1*b; A1=b'; a1=A1(i+1:n,i);
v2=Householder(a1); clear v1;
v1(1:i,1:i)=eye(i); v1(i+1:n,i+1:n)=v2';
b=b*v1; u=u*u1; v=v*v1;
end
a=b(n1-1:m,n1-1); clear u1;
u1(1:n1-2,1:n1-2)=eye(n1-2);
u1(n1-1:m,n1-1:m)=Householder(a);
b=u1*b; u=u*u1;
	if (m>n)
	a=b(n:m,n); clear u1;
	u1(1:n-1,1:n-1)=eye(n-1);
	u1(n:m,n:m)=Householder(a);
	b=u1*b; u=u*u1;
	end
	if(m<n)
	A1=b';a1=A1(n1:n,n1-1);
	v2=Householder(a1);
	clear v1;
	v1(1:n1-1,1:n1-1)=eye(n1-1);
	v1(n1:n,n1:n)=v2'; b=b*v1; v=v*v1;
	A1=b';a1=A1(n1+1:n,n1);
	v2=Householder(a1);
	clear v1;
	v1(1:n1,1:n1)=eye(n1);
	v1(n1+1:n,n1+1:n)=v2';
	b=b*v1;v=v*v1;
	end

$B_1 = \begin{bmatrix} 0 & b_1 & & & & O \\ b_1 & 0 & b_2 & & & \\ & b_2 & 0 & & & \\ & & \ddots & & \ddots & \\ & & & & & b_{2n-1} \\ O & & & & b_{2n-1} & 0 \end{bmatrix}$ has eigenvalues $\pm\sigma_i$ *with* normalized associated eigenvectors $h_i^{\pm} = \frac{1}{\sqrt{2}}(v_{i1}, \pm u_{i1}, v_{i2}, \pm u_{i2}, \dots, v_{in}, \pm u_{in})^T$. Considering this property, we first transform $\hat{B}$ to the matrix B_1 which is a tridiagonal matrix and compute the eigen pairs of B_1. Then we get the SVD of $\hat{B}$. Once this SVD is obtained, the SVD of B and hence of A is calculated.The procedure that left is to transform $\hat{B}$ *to* the matrix B_1. To operate a shuffle matrix [2] on the revised matrix $[\begin{matrix} 0 & \hat{B}^T \\ \hat{B} & 0 \end{matrix}]$ can get B_1: $B_1 = P^T \begin{bmatrix} 0 & \hat{B}^T \\ \hat{B} & 0 \end{bmatrix} P$. The code for generating the shuffle matrix with respect to the dimension of $\hat{B}$ is given in Table 2.5. The input of function GenerateShuff is the dimension of matrix $\hat{B}$. So far we can give the code of calculating the SVD of the original bidiagonal matrix B, which is shown in Table 2.6.

Table 2.5 Shuffle matrix

```
function p=GenerateShuff(m)
for i=1:2:2*m-1
   p(:,i)=geneVector(2*m,(i+1)/2);
end
for i=2:2:2*m
   p(:,i)=geneVector(2*m,i/2+m);
end
function p=geneVector(m,n)
p=zeros(m,1);  p(n)=1;
```

Table 2.6 SVDUpBidiag

```
function [u,d,v]=SVDUpBidiag(B)
[m,n]=size(B); B1=B(1:n,1:n);
C(1:n,n+1:n+n)=B1';C(1:n,1:n)=zeros(n,n);
C(n+1:2*n,1:n)=B1;C(n+1:2*n,n+1:2*n)=zeros(n,n);
p=GenerateShuff(n); c1=p'*C*p; [x,d]=eig(c1);
x1=fliplr(x);d1=fliplr(flipud(d));  d=d1(1:n,1:n);
x1=x1(:,1:n);  x1=x1*2^(.5);
for i=1:n
   v(i,:)=x1(2*i-1,:);  u(i,:)=x1(2*i,:);
end
d(n+1:m,1:n)=zeros(m-n,n);u(n+1:m,1:n)=zeros(m-n,n);
u(:,n+1:m)=zeros(m,m-n);u(n+1:m,n+1:m)=eye(m-n);
```

Table 2.7 SVDDirect

```
function [u,d,v]= SVDDirect (A)
[m,n]=size(A);
if (m>n|m==n)
   [p,j,q]=BiDiag(A);[u1,d,v1]=SVDUpBidiag(j);
   u=p*u1;     v=q*v1;
end
if (m<n)
   [p,j,q]=BiDiag(A');[u1,d1,v1]=SVDUpBidiag(j);
   v=p*u1;    u=q*v1;     d=d1';
end
```

Combine the function SVDUpBidiag and BiDiag, we get the function which computes SVD of any matrix. We call this function ‘SVDDirect’ although it is actually an ‘indirect’ iterative way to get SVD. The code of it is shown in Table 2.7.

2.2.2 COMBINED METHODS

We see in the function SVDDirect, the line '[u1,d,v1]=SVDUpBidiag(j);' is used to get the SVD of the bidiagonal matrix B. This line can be substituted by other functions that implement the SVD process. One option is the 'SVDecom' function which we made at the beginning. Certainly, the built-in functions can be used here. Another choice is to use Jacobi iterative method to get SVD and replace 'SVDUpBidiag' by 'SVDBiDiaJacob' which is a function that we programmed based on Jacobi iterative method. It is given in Table 2.8. The function 'jacobi_svd' is based on Jacobi iteration. Replacing 'SVDUpBidiag' in SVDDirect by 'SVDBiDiaJacob', we give the function 'SVDJacob' which is listed in Table 2.9.

Table 2.8 SVDBiDiaJacob

```
function [u,d,v]=SVDBiDiaJacob(B)
[m,n]=size(B);  B1=B(1:n,1:n);      [u,d,v]=jacobi_svd(B1);
d(n+1:m,1:n)=zeros(m-n,n); u(n+1:m,1:n)=zeros(m-n,n);
u(:,n+1:m)=zeros(m,m-n);    u(n+1:m,n+1:m)=eye(m-n);

function [U,S,V]= jacobi_svd(A)
TOL=1.e-8; n=size(A,1);U=A; V=eye(n);converge=TOL+1;
while converge>TOL
  converge=0;
  for j=2:n
    for i=1:j-1
      alpha=U(:,i)'*U(:,i); beta=U(:,j)'*U(:,j);  gamma=U(:,i)'*U(:,j);
      converge=max(converge,abs(gamma)/sqrt(alpha*beta));
      zeta=(beta-alpha)/(2*gamma);
      t=sign(zeta)/(abs(zeta)+sqrt(1+zeta^2));
      c=1/((1+t^2)^(.5));s=c*t;t=U(:,i);
       U(:,i)=c*t-s*U(:,j);     U(:,j)=s*t+c*U(:,j);
       t=V(:,i);  V(:,i)=c*t-s*V(:,j);  V(:,j)=s*t+c*V(:,j);
    end
  end
end
for j=1:n
    singvals(j)=norm(U(:,j));   U(:,j)=U(:,j)/singvals(j);
end
S=diag(singvals);
```

We call the SVD function that combines the functions 'SVDecom' and 'BiDiag', SVDComb which is shown in Table 2.10. As an extended work, we implemented the QR decomposition using Householder transformations. The code to implement the QR is in Table 2.11.

Table 2.9 SVDJacob

```
function [u,d,v]=SVDJacob(A)
[m,n]=size(A);
if (m>n|m==n)
   [p,j,q]=BiDiag(A);    [u1,d,v1]=SVDBiDiaJacob(j);    u=p*u1;    v=q*v1;
end
if (m<n)
   [p,j,q]=BiDiag(A'); [u1,d1,v1]=SVDBiDiaJacob(j);    v=p*u1;    u=q*v1;    d=d1';
End
```

Table 2.10 SVDComb

```
function [u,d,v]=SVDComb(A)
[m,n]=size(A);
if (m>n|m==n)
   [p,j,q]=BiDiag(A);
    [u1,d,v1]=SVDecom(j);
   u=p*u1;    v=q*v1;
end
if (m<n)
   [p,j,q]=BiDiag(A');
   [u1,d1,v1]=SVDecom(j);
   v=p*u1;    u=q*v1;    d=d1';
end
```

Note that if we use ‘Q=HouseholderQR(A)’, an orthogonal matrix Q will be generated. This function can be used to generate an arbitrary matrix with known singular values. Actually, it can be used to generate an orthogonal matrix, too.

Table 2.11 Housholder QR

```
function [Q,r]=HouseholderQR(A)
[m,n]=size(A);m1=min(m,n);
if m>n
   m1=m1+1;
end
x=A(:,1);Q1=Householder(x); A1=Q1*A;Q=Q1;
for i=2:m1-1
   x=A1(i:m,i);
   Q1(i:m,i:m)=Householder(x);
   Q1(i:m,1:i-1)=zeros(m-i+1,i-1);
   Q1(1:i-1,1:i-1)=eye(i-1);
   Q1(1:i-1,i:m)=zeros(i-1,m-i+1);
   A1=Q1*A1;    Q=Q*Q1;
end
r=Q'*A;
```

2.2.3 TESTING

Figure 2.1 is given to show the comparisons of the timing used by different SVD methods. In this experiment, the square matrices are treated. We give the log-log plot of the time cost against the dimension.

The indirect method cost less time than the direct method. Obviously, the implemented numerical methods will cost more CPU time than the built-in function.

The accuracy comparison is shown in Figure 2.2. We see the direct method achieves higher accuracy than the indirect method. The combined squaring method gets lowest precision.

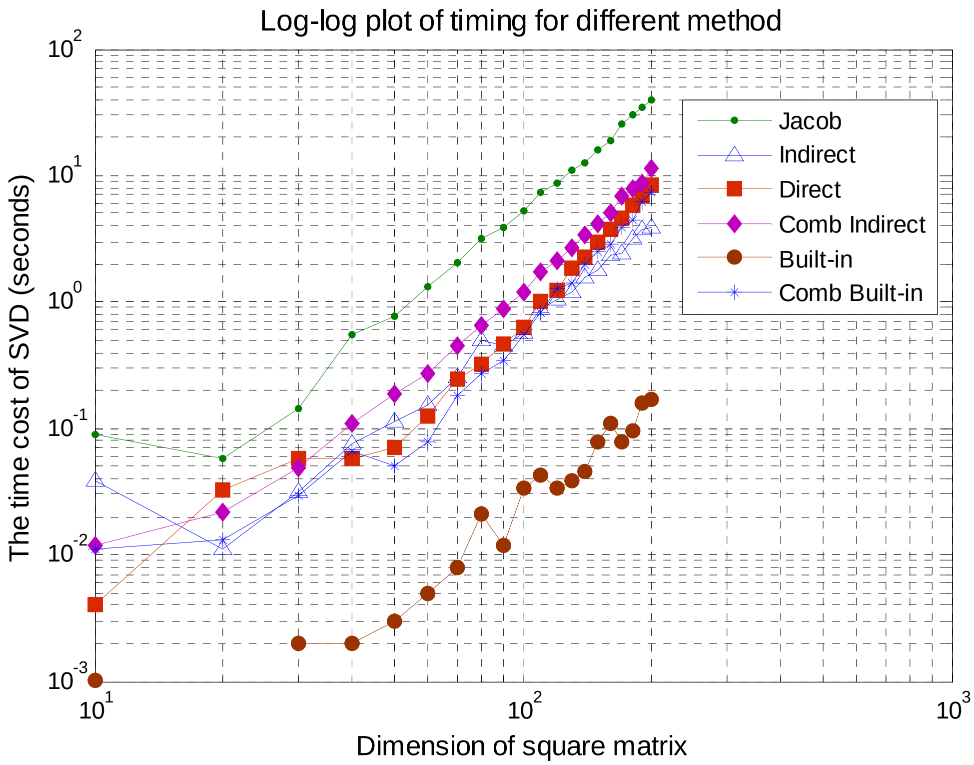

Figure 2.1 Timings

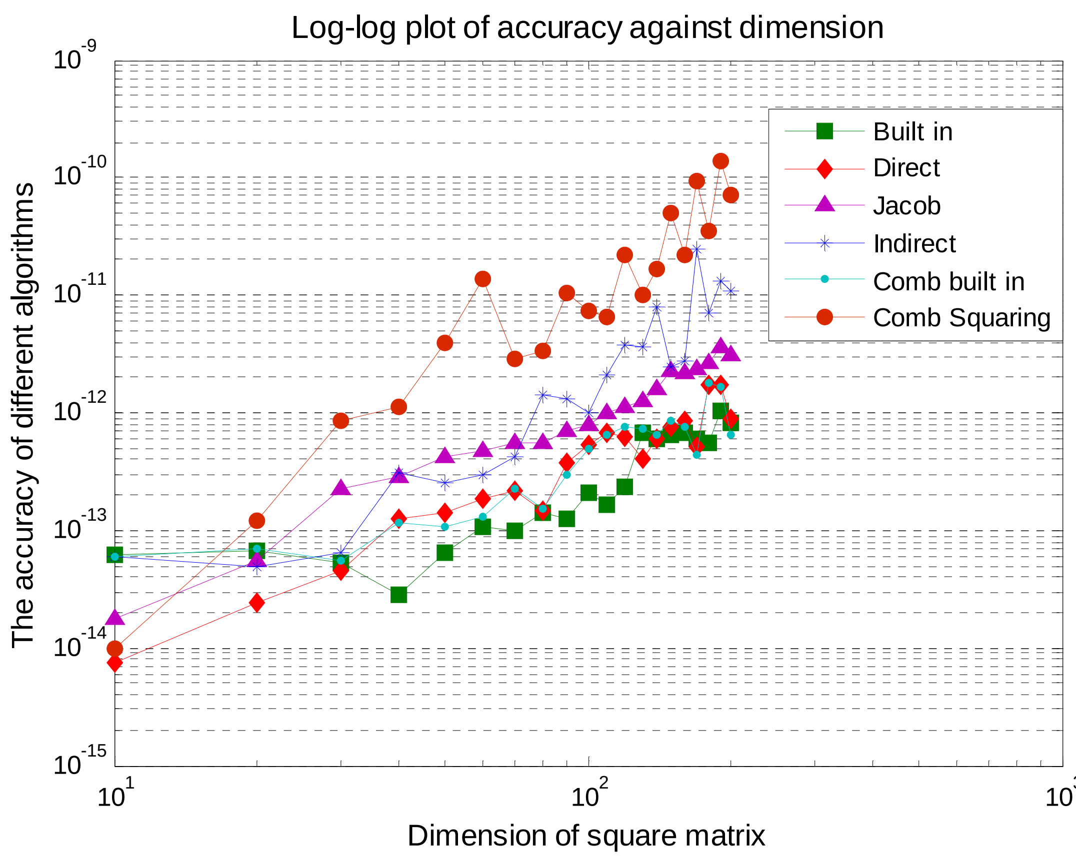

Figure 2.2 Acuraccy

The description of each method:

- Built-in method is using the svd function in Matlab to get the SVD.
- Direct method is first transforming the original matrix to the bidiagonal matrix and then getting its SVD via the eigen pairs of the tridiagonal matrix.
- The Jacob method is to get the SVD of the bidiagonal matrix through Jacobi rotation.
- Indirect method is the $A^T A$ and AA^T approach.
- Combined built in method is to get the SVD of bidiagonal matrix via the built in svd function and combined squaring method is employing the squaring/indirect method to get the SVD of bidiagonal matrix and finally get the SVD of original matrix.

The following lines are to test the orthogonality of the matrices got by the direct method. The last two lines are to compare the singular values got by the direct method and the built in function.

Table 2.12 – Orthogonality and singular value check

>> A=rand(150,40); >> [u,d,v]=SVDDirect(A); >> norm(u'*u-eye(150),inf) ans = 4.9280e-014 >> norm(v'*v-eye(40),inf) ans = 1.5504e-014 >> [u1,d1,v1]=svd(A); >> norm(d-d1,1) ans = 7.1054e-014	>> B=randn(120,230); >> [u,d,v]=SVDDirect(B); >> norm(u*u'-eye(120),inf) ans = 5.9718e-014 >> norm(v*v'-eye(230),1) ans = 8.5688e-014 >> [u1,d1,v1]=svd(B); >> norm(d1-d,inf) ans = 9.9476e-014

2.3 GOLUB – REINSCH ALGORITHM

This last method of computing the SVD of a matrix is based on the Golub-Reinsch algorithm which is a variant of the Francis algorithm. This time we will not compute the SVD of the matrix using the eig function of matlab but we will implement the Golub-Reinsch algorithm .

The following steps are followed in order to compute the SVD of a rectangular matrix A. We assume that the number of rows (m) is greater than the number of columns(n).

- Bidiagonalization of matrix A.

In this step the matrix is decomposed in $A = QBP^T$ where B is an upper Bidiagonal matrix by applying a series of Householder transformation as we described in the previous chapter .

- $B = \begin{bmatrix} \tilde{B} \\ 0 \end{bmatrix}$
- $B = \tilde{B}$

- Diagonalization of the bidiagonal matrix B.

The Bidiagonal matrix B can be reduced to a diagonal matrix by iteratively applying the implicitly shifted QR algorithm (Francis). The matrix B is decomposed as $\Sigma = X^T BY$ where Σ is a diagonal matrix, X and Y are orthogonal unitary matrices.

- $U = QX$
- $V^T = (PY)^T$
- $A = U\Sigma V^T$ Singular value decomposition of A.

2.3.1 IMPLICITLY SHIFTED QR ALGORITHM

The algorithm computes a sequence B of bidiagonal matrices starting from $B_0 = B$ as follows. From B_i the algorithm computes a shift μ^2, which is usually taken to be the smallest eigenvalue of the bottom 2 by 2 block of $B_i{B_i}^T$. Then the algorithm does an implicit QR factorization of the shifted matrix $B_i{B_i}^T - \mu^2 I = QR$, where Q is orthogonal and R upper triangular, from which it computes a bidiagonal B_{i+1} such that ${B_{i+1}}^T B_{i+1} = RQ + \mu^2 I$. As i increases, B converges to a diagonal matrix with the singular values on the diagonal.

Steps: $B = \begin{pmatrix} d_1 & f_2 & & \\ & d_2 & \ddots & \\ & & \ddots & f_n \\ & & & d_n \end{pmatrix}$

- Determine the shift μ which is called the Wilkinson shift. This is the smallest eigenvalue of the bottom 2 by 2 block of $B_i{B_i}^T$. $\begin{pmatrix} {d^2}_{n-1} + {f^2}_{n-1} & d_{n-1}f_n \\ d_{n-1}f_n & {d^2}_n + {f^2}_n \end{pmatrix}$
- Find the Givens matrix $G_1 = G(1,2;\theta)$ such that $\begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \cdot \begin{pmatrix} {d^2}_1 - \mu \\ d_1 f_2 \end{pmatrix} = \begin{pmatrix} * \\ 0 \end{pmatrix}$. Compute BG_1.
- We have: $BG_1 = \begin{pmatrix} * & * & & \\ * & * & \ddots & \\ & & \ddots & * \\ & & & * \end{pmatrix}$ So we should zero out the * term. We want to find P_2 and G_2 such that $P_2(BG_1)G_2$ is bidiagonal.
- We can find P_2 and G_2 by Givens transformation and we have $P_2(BG_1)G_2 = \begin{pmatrix} * & * & & \\ & * & \ddots & \\ & * & \ddots & * \\ & & & * \end{pmatrix}$

So we should zero out the * term. We want to find P_3 and G_3 such that $P_3P_2BG_1G_2G_3$ is bidiagonal. We repeat these steps until BG_1 is bidiagonal.

- Finally we have $P_{n-1} \dots P_2 B G_1 \dots G_{n-1} = \begin{pmatrix} * & * & & \\ & * & \ddots & \\ & & \ddots & * \\ & & & * \end{pmatrix}$. Iterate until the off-diagonal entries converge to 0, and the diagonal entries converge to singular values.

2.3.2 IMPLICITLY ZERO SHIFTED QR ALGORITHM

The roundoff errors in this algorithm are generally on the order of eB, where e is the precision of the floating point arithmetic used. We would expect absolute errors in the computed singular values of the same order. In particular, tiny singular values of *B* could be changed completely. In order to avoid these errors and to increase precision it is introduced a variation of this algorithm. It is called the implicit zero shift QR algorithm and it corresponds to the algorithm above when μ=0. Comparing to the standard algorithm we see that the (1,2) entry is zero instead of nonzero. This zero will propagate through the rest of the algorithm and is the key to its effectiveness. We decided to implement this algorithm instead of the standard. Below you can see both the steps and the pseudo code of the algorithm.

Steps: $B = \begin{pmatrix} d_1 & f_2 & & \\ & d_2 & \ddots & \\ & & \ddots & f_n \\ & & & d_n \end{pmatrix}$

- We have $BG_1 = \begin{pmatrix} * & 0 & & \\ * & * & \ddots & \\ & & \ddots & * \\ & & & * \end{pmatrix}$

- We find P_2 and G_2, such that $P_2(BG_1)G_2 = \begin{pmatrix} * & * & & & \\ 0 & * & 0 & & \\ & * & * & \ddots & \\ & & & \ddots & * \\ & & & & * \end{pmatrix}$

- Finally $P_{n-1} \dots P_2 B G_1 \dots G_{n-1} = \begin{pmatrix} * & * & & \\ & * & \ddots & \\ & & \ddots & * \\ & & & * \end{pmatrix}$.

Let B be an n by n bidiagonal matrix with diagonal entries $d_1 \dots d_n$ and superdiagonal entries $f_1 \dots f_{n-1}$. The following algorithm replace $d_i$and f_i by new values corresponding to one step of the QR iteration with zero shift:

$oldc = 1;\ g = d_1;\ p = f_1$
$for\ i = 1\ to\ n-1$
  $[c, s, r] = ROT(g, p)$
  $if\ (i \neq 1)\ then$
    $f_{i-1} = olds * r$
  $end\ if$

$g = oldc * r$; $p = d_{i+1} * s$; $h = d_{i+1} * c$;
$[c, s, r] = ROT(g, p)$; $d_i = r$
$if\ (i \neq n-1)\ then$
  $p = f_{i+1}$
$end\ if$
$oldc = c$; $olds = s$
$end\ for$
$f_{n-1} = h * s$; $d_{n-1} = h * c$

The algorithm above uses a subroutine ROT which actually is the Givens rotation and takes g and p as inputs and returns $c = cos\theta$ and $s = sin\theta$ such that: $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}\begin{bmatrix} g \\ p \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$

<u>Stopping criterion</u>

The algorithm below decides when an offdiagonal entry f_i can be set to zero. $0 < tol < 1$ is a relative error tolerance. The value of tol we used in the numerical tests was $tol = 100 \cdot \varepsilon$. The value 100 was chosen empirically to make the algorithm run fast but it could be easily be set as large as 1000 or as small as 10.

$\mu_1 = d_1$
$for\ i = 1{:}n-1$
  $if\ |f_i| < tol * \mu_i, set\ f_i = 0$; $\mu_{i+1} = |d_{i+1}| * \left(\frac{\mu_i}{\mu_i + |f_i|}\right)$
$end\ for$

2.3.3 MATLAB FUNCTIONS

Followings are the Matlab codes regarding the above iterative method of computing the singular values.

Table 2.13 Given's Rotations

```
function [cs,sn,r]=rot(f,g)
x=[f g];[G,r]=planerot(x');
cs=G(1,1);sn=G(1,2);r=r(1);
```

Table 2.14 Implicit QR zero shift

```
function [d,e]=IQR(d,e)
n=length(d);oldc=1;c=1;
for k=1:n-1
   [c,s,r]=ROT(c*d(k),e(k));
   if k~=1
      e(k-1)=r*olds;
   end
   [oldc,olds,d(k)]=ROT(oldc*r,d(k+1)*s);
end
h=c*d(n);e(n-1)=h*olds;d(n)=h*oldc;
```

Table 2.15 Iterations to get singular values

```
function d=svdFrancis(A)
 TOL=100*eps;  d=diag(A);  n=length(d);
for i=1:n-1
    e(i)=A(i,i+1);
end
maxit=500*n^2; lambda(n)=abs(d(n));
for j=n-1:-1:1
  lambda(j)=abs(d(j))*lambda(j+1)/(lambda(j+1)+abs(e(j)));
end
mu(1)=abs(d(1));
for j=1:n-1
  mu(j+1)=abs(d(j+1))*mu(j)/(mu(j)+abs(e(j)));
end
sigmaLower=min(min(lambda),min(mu));
thresh=max(TOL*sigmaLower,maxit*realmin); iUpper=n-1; iLower=1;
for iterations=1:maxit
    iUpper=i;
    if abs(e(i))>thresh
      break;
    end
  end
  j=iUpper;
  for i=iLower:iUpper
    if abs(e(i))>thresh
      j=i;    break;
    end
  end
  iLower=j;
   if (iUpper==iLower & abs(e(iUpper))<=thresh) |(iUpper<iLower)
        d=sort(abs(d));    d(end:-1:1)=d;
    return
  end
[d(iLower:iUpper+1),e(iLower:iUpper)]=IQR(d(iLower:iUpper+1),e(iLower:iUpper));
End
```

Table 2.16 SVD Function

```
function s=SVDFranIt(A);
[n,m]=size(A);
if n>m
    B=BiDiag(A);    Bhat=B(1:m,1:m);
else
    A=A'; B=BiDiag(A);    Bhat=B(1:n,1:n);
end
d=svdFrancis(Bhat); s=d;
end
```

3. NUMERICAL EXPERIMENTS

We conducted some experiments on the following matrices:

- Dense rectangular matrix.

- Dense symmetric matrix.
- Sparse symmetric matrix.

The dense matrices were generated using predefined singular values and the sparse matrices were generated using Matlab's built-in function.

The SVD decomposition methods we tested are as follows:

- **Built-in SVD**: Matlab's built-in svd function.
- **Comb. Built in**: We first bidiagonalize the matrix then use the built-in function to get the decomposition.
- **Comb. Square**: Use indirect method to get the SVD of the bidiagonalized matrix.
- **Indirect method**: We use the squaring method to get the SVD.
- **Direct method**: We tridiagonalize the bidiagonalized matrix, then use the relationship between eigen pairs of tridiagonal matrix and SVD to get the decomposition.
- **Francis**: We bidiagonalize the matrix, then use the Golub-Reinsch algorithm to get SVD.

We evaluated the following parameters:

- **Timing**: the time it takes to compute the decomposition.
- **Accuracy**: Accuracy of the decomposition is found using $\left\| U\Sigma V^T - A \right\|_\infty$.
- **Orthogonality Check**: Orthogonality of the left and right orthogonal matrices was checked via $\left\| UU^T - I_d \right\|_\infty$, $\left\| U^T U - I_d \right\|_\infty$, $\left\| VV^T - I_d \right\|_\infty$ and $\left\| V^T V - I_d \right\|_\infty$.
- **Error in singular values**: Found using $\left\| SV_{computed} - SV_{actual} \right\|_\infty$ where the actual singular values were generated previously for dense matrices and for sparse matrices they were generated using the built- in function.

3.1 Dense Rectangular Matrix

We tested for two types of rectangular matrices: vary number of columns keeping the number of rows fixed and the inverse case, where the numbers of rows are varied keeping the number of columns fixed. The results are tabulated below.

Table 3.1: Timings when n is fixed and m is varied (seconds).

n	m	Built in SVD	Comb. Built in	Comb. Square	Indirect Method	Direct Method	Francis
	100	0.0643	0.1742	0.6436	0.4331	0.2001	1.0876
	200	0.0133	0.5130	1.5780	1.1453	0.4999	2.0906
	300	0.0189	1.2288	3.6087	2.3616	1.2544	3.1980
	400	0.0365	2.5293	6.7061	4.2062	2.5645	4.8086
	500	0.0355	4.9097	11.5758	6.7367	4.9373	7.9302
	600	0.0487	7.8638	17.9327	10.1238	7.8953	9.4502
	700	0.0553	11.6358	25.6041	14.3858	11.6814	12.2837
100	800	0.0710	16.5239	35.2338	19.3894	16.5601	15.4566

	900	0.0854	22.6020	47.3990	25.6453	22.4533	19.1506
	1000	0.1057	31.7354	64.9555	33.1568	31.6759	23.2127
	1100	0.1411	41.2789	83.6196	40.9601	40.7136	27.6247
	1200	0.1617	50.7881	100.2060	49.6997	51.6991	32.6406
	1300	1.0426	63.5952	122.9314	60.2307	61.9592	37.8728
	1400	1.1965	77.8064	147.5340	72.3586	75.6486	43.8433
	1500	1.3826	95.3954	180.9166	88.9260	95.1457	49.7953
200	100	0.0178	0.4646	1.6561	1.1513	0.5551	1.3880
	200	0.0334	1.9439	3.8301	1.8352	1.9785	5.4959
	300	0.0613	3.7402	6.6063	3.0762	3.6905	9.0105
	400	0.0717	6.7568	11.7872	4.8840	6.8905	12.9847
	500	0.0823	11.9713	19.3053	7.6183	12.0306	17.3449
	600	0.1200	18.6601	29.1580	10.7631	18.4879	22.7105
	700	0.1332	26.3469	40.9780	14.8118	26.2718	29.1993
	800	0.1621	36.7793	56.2104	19.7015	36.7853	35.7350
	900	0.1928	49.7251	74.9258	25.7169	49.7240	43.5936
	1000	0.2483	69.3476	101.6042	35.1367	69.2113	52.0930
	1100	0.2704	88.8150	130.0029	41.4801	88.8953	61.2714
	1200	0.3322	110.0937	160.8240	50.8083	110.5251	72.3280
	1300	0.3265	133.9495	194.5232	61.9938	133.7772	82.2232
	1400	0.4057	162.4365	234.0571	74.4330	162.3709	94.1530
	1500	0.4107	201.6069	286.7923	89.4480	200.9714	106.6020

From the Table 3.1, we see that the built-in Matlab function is the fastest. For the methods that we have coded, the indirect method is the fastest when n<m, followed by the Francis algorithm, direct and combined built-in method. From Figure 3.1, we know when n>m the Francis algorithm gives the fastest result followed by the rest. The combination of square method is the slowest of them all in both cases, because it involves bidiagonalization and squaring.

Table 3.2: Accuracies when m is fixed and n is varied.

n	m	Built in SVD(10^{-12})	Comb. Built in(10^{-11})	Comb. Square(10^{-10})	Indirect Method(10^{-11})	Direct Method(10^{-11})
100	100	0.2662	0.0370	0.0082	0.0592	0.0580
200		0.3044	0.0526	0.0048	0.0155	0.0604
300		0.3710	0.0535	0.0047	0.0160	0.0572
400		0.2831	0.0564	0.0053	0.0183	0.0692
500		0.3181	0.0679	0.0068	0.0149	0.0758
600		0.3468	0.0817	0.0082	0.0166	0.0736
700		0.2938	0.0815	0.0082	0.0178	0.0787
800		0.3970	0.0980	0.0096	0.0199	0.0745
900		0.3123	0.1170	0.0114	0.0234	0.0900
1000		0.2936	0.1128	0.0113	0.0235	0.0896
1100		0.3266	0.1264	0.0126	0.0226	0.0946
1200		0.3402	0.1261	0.0123	0.0244	0.0961
1300		0.3923	0.0958	0.0101	0.0212	0.0786
1400		0.3512	0.1234	0.0120	0.0239	0.0897
1500		0.3203	0.1757	0.0171	0.0231	0.1059
100	200	0.6043	0.0921	0.0081	0.0317	0.1407
200		0.5283	0.1153	0.2477	0.1671	0.1768
300		0.5575	0.1360	0.0140	0.0511	0.1814
400		0.5982	0.1348	0.0135	0.0371	0.1894
500		0.7264	0.1729	0.0162	0.0358	0.1904
600		0.6554	0.2310	0.0229	0.0373	0.2442
700		0.6513	0.1867	0.0184	0.0405	0.2009
800		0.6208	0.1955	0.0194	0.0403	0.1889
900		0.6462	0.1803	0.0177	0.0412	0.2065
1000		0.6525	0.1998	0.0201	0.0456	0.2129
1100		0.6955	0.1994	0.0204	0.0407	0.2328

1200		0.5719	0.2344	0.0230	0.0461	0.2468
1300		0.6046	0.2178	0.0221	0.0453	0.2660
1400		0.5862	0.2150	0.0215	0.0438	0.2115
1500		0.6172	0.2702	0.0265	0.0490	0.2403

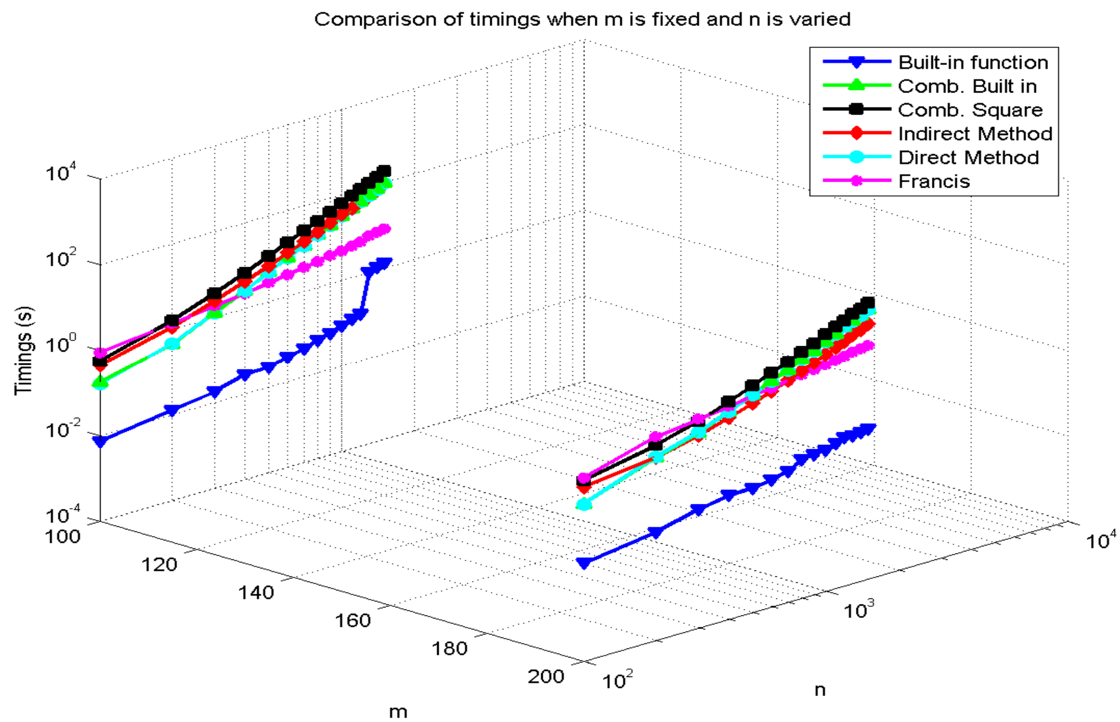

Fig. 3.1. Comparison of timings when m is fixed and n is varied.

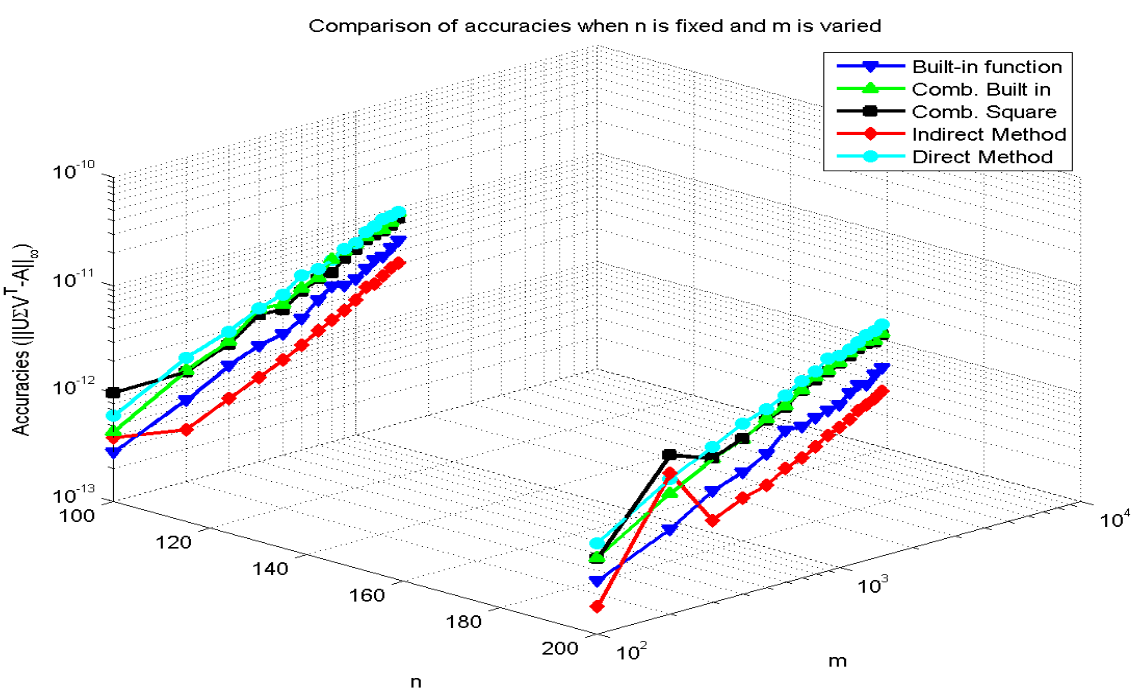

Fig. 3.2: Comparison of accuracies when n is fixed and m is varied.

Then we provide the accuracy comparisons in these two cases. Table 3.2 gives the accuracy comparisons when m is fixed and n is varied. Figure 3.2 plots comparison of accuracies when n is fixed and m is varied.

From Table 3.2 and Fig. 3.2, we see that using the methods that we have coded, for both n<m and n>m, indirect method gets the best results, even better than the built-in function. The reason behind this is that the built in svd decomposition of Matlab uses 75 QR step iterations, which cannot be changed, but the indirect method keeps on computing further, so it gives better results. The accuracy loss in direct method is due to precision loss during orthogonalization. The combined methods obtain accuracies which are between those of direct and indirect methods.

As for the orthogonality testing experiments, we tested $\|UU^T - I_d\|_\infty$, $\|U^TU - I_d\|_\infty$, $\|VV^T - I_d\|_\infty$ and $\|V^TV - I_d\|_\infty$ in both cases, i.e., either *n* varied or *m* varied with the other dimension fixed. In Figure 3.3 to Figure 3.6, we give the orthogonality comparisons via different methods. We see the indirect methods get better results than the direct methods. The indirect method achieves better orthogonality than the built-in function and so on due to its increased accuracy over the others.

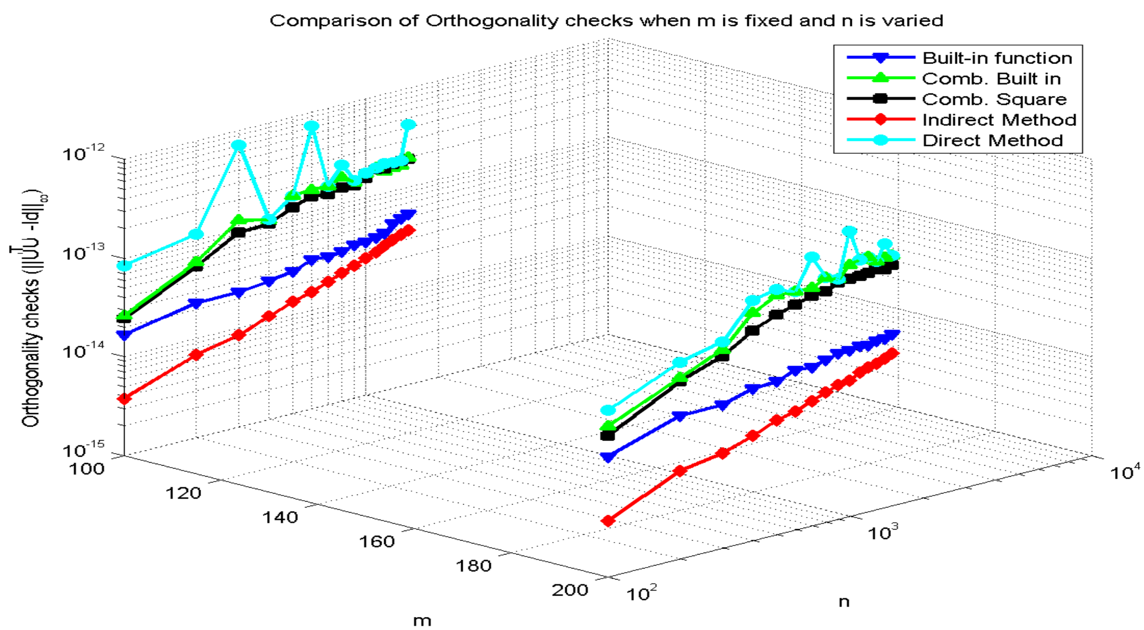

Fig. 3.3: Orthogonality checks $\|U^TU - I_d\|_\infty$ when m is fixed and n is varied.

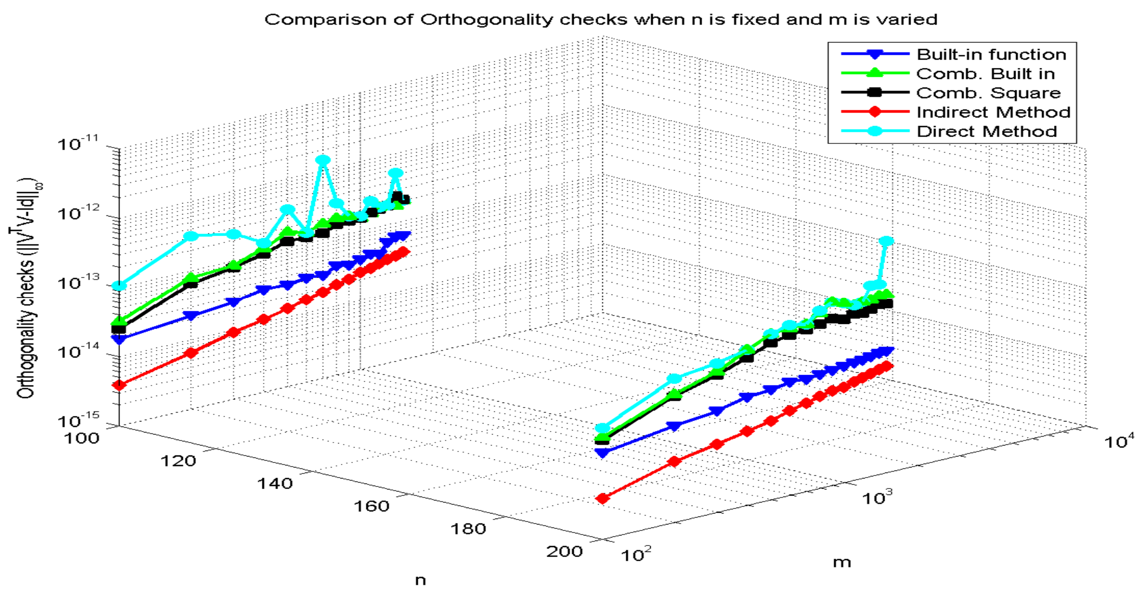

Fig. 3.4: Orthogonality checks $\left\|V^T V - I_d\right\|_\infty$ when n is fixed and m is varied.

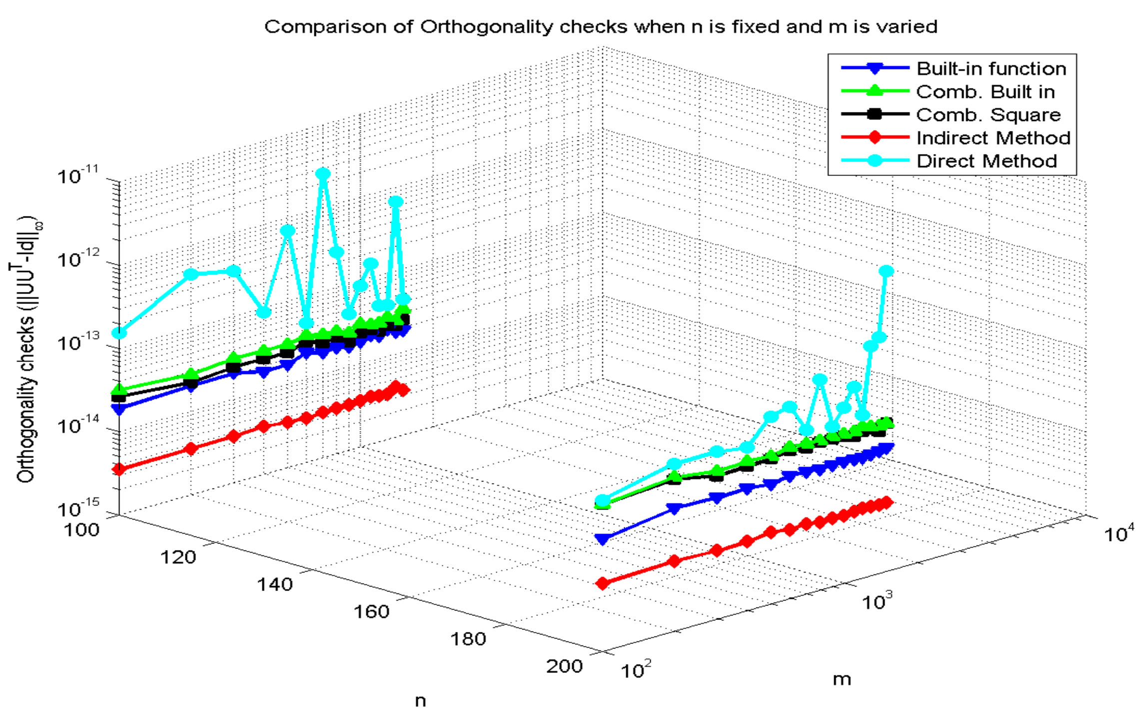

Fig. 3.5: Orthogonality checks $\left\|UU^T - I_d\right\|_\infty$ when n is fixed and m is varied.

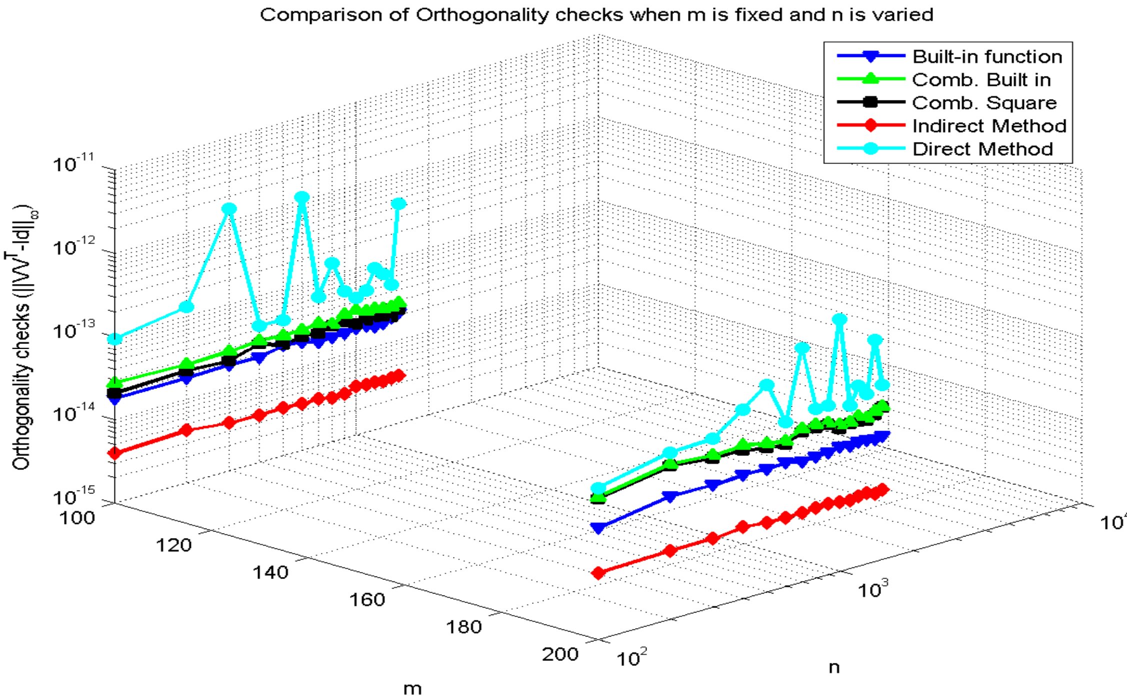

Fig. 3.6: Orthogonality checks $\left\|VV^T - I_d\right\|_\infty$ when m is fixed and n is varied.

Table 3.3: errors in singular values when n is fixed and m is varied.

n	M	Built in SVD (10^{-13})	Comb. Built in (10^{-14})	Comb. Square (10^{-12})	Indirect Method (10^{-12})	Direct Method (10^{-13})	Francis (10^{-11})
100	100	0.0311	0.2442	0.0151	0.0106	0.0422	0.0142
	200	0.0400	0.3997	0.0972	0.0161	0.0466	0.0266
	300	0.0400	0.2665	0.0211	0.0133	0.0444	0.0341
	400	0.0400	0.3997	0.0213	0.0036	0.0444	0.0263
	500	0.0799	0.2442	0.0522	0.0220	0.0533	0.0380
	600	0.0533	0.3997	0.0110	0.0050	0.0666	0.0369
	700	0.0222	0.3109	0.0174	0.0471	0.0444	0.0220
	800	0.0377	0.4663	0.0135	0.0043	0.0488	0.0547
	900	0.0488	0.3109	0.0096	0.0173	0.0533	0.0469
	1000	0.0311	0.3109	0.1272	0.0047	0.0711	0.0462
	1100	0.0311	0.3109	0.0024	0.0053	0.0622	0.0519
	1200	0.0355	0.4441	0.0046	0.0022	0.0555	0.0490
	1300	0.0444	0.3997	0.0019	0.0047	0.0666	0.0476
	1400	0.0355	0.3109	0.0143	0.0077	0.0366	0.0611
	1500	0.0355	0.3109	0.0050	0.0027	0.0355	0.0426
	100	0.0577	0.8882	0.0173	0.0085	0.0400	0.0338
	200	0.0533	0.5329	0.0141	0.0039	0.0600	0.0373
	300	0.0355	0.3553	0.1296	0.0111	0.0799	0.0380
	400	0.0444	0.5773	0.1719	0.0052	0.1332	0.0782
	500	0.0577	0.2665	0.0211	0.0308	0.0777	0.0725
	600	0.0444	0.5773	0.0134	0.0089	0.0688	0.1087
	700	0.0400	0.3109	0.0097	0.0249	0.0933	0.0682
	800	0.0355	0.4885	0.0201	0.0144	0.0644	0.1108

200	900	0.0488	0.2665	0.0237	0.0096	0.0666	0.0753
	1000	0.0355	0.4885	0.0226	0.0061	0.1288	0.1059
	1100	0.0311	0.3997	0.0231	0.0075	0.0666	0.0817
	1200	0.0355	0.3997	0.0092	0.0074	0.0600	0.1116
	1300	0.0444	0.5773	0.0195	0.0098	0.1199	0.1144
	1400	0.1110	0.4885	0.0073	0.0054	0.0755	0.1307
	1500	0.0444	0.3553	0.0553	0.1428	0.0933	0.0824

Table 3.4: Errors in singular values when m is fixed and n is varied.

n	M	Built in SVD (10^{-14})	Comb. Built in (10^{-14})	Comb. Square (10^{-12})	Indirect Method (10^{-13})	Direct Method (10^{-13})	Francis (10^{-11})
100	100	0.3109	0.3553	0.0665	0.0335	0.0488	0.0259
200		0.3997	0.3997	0.0339	0.1107	0.0600	0.0153
300		0.3553	0.2665	0.0274	0.2923	0.0488	0.0217
400		0.5551	0.4885	0.0104	0.0178	0.0488	0.0224
500		0.2665	0.5329	0.0022	0.0329	0.0600	0.0302
600		0.4219	0.3775	0.0183	0.0571	0.0600	0.0384
700		0.3553	0.3775	0.0091	0.0870	0.0666	0.0249
800		0.3775	0.3553	0.0073	0.1036	0.0444	0.0242
900		0.2220	0.4885	0.0039	0.0133	0.0622	0.0256
1000		0.6217	0.2887	0.0080	0.0720	0.0711	0.0348
1100		0.5773	0.3109	0.0048	0.0912	0.0533	0.0490
1200		0.4441	0.3109	0.0021	0.0345	0.0577	0.0632
1300		0.2220	0.3997	0.0186	0.0576	0.0755	0.0625
1400		0.3775	0.1998	0.0074	0.0455	0.0666	0.0298
1500		0.3331	0.3553	0.0141	0.0715	0.0444	0.0576
100	200	0.3553	0.3553	0.0563	0.0414	0.0488	0.0263
200		0.3997	0.2665	0.0460	0.0444	0.0822	0.0551
300		0.3997	0.3997	0.3152	0.1192	0.0733	0.0593
400		0.5773	0.3997	0.0901	0.3312	0.0799	0.0853
500		0.3997	0.5773	0.0098	0.1518	0.1066	0.1059
600		0.5773	0.3997	0.0134	0.0570	0.0711	0.0746
700		0.6661	0.3109	0.0275	0.2344	0.0622	0.0860
800		0.6217	0.6217	0.0065	0.0749	0.0977	0.1165
900		0.3997	0.4441	0.0082	0.0926	0.0644	0.1116
1000		0.3553	0.4441	0.0272	0.1977	0.1110	0.0938
1100		0.7550	0.3997	0.0330	0.1075	0.0622	0.1414
1200		0.3997	0.5773	0.0392	0.2010	0.0600	0.1251
1300		0.3997	0.3997	0.0293	0.0427	0.0844	0.1023
1400		0.8882	0.5773	0.0296	0.0984	0.0711	0.0973
1500		0.5107	0.3997	0.0151	0.0906	0.0511	0.1549

From tables 3.3 and 3.4, we see that when $n<m$ all of the methods give comparable results with the combined built-in method marginally the best. The accuracy loss in indirect method is due to the Gram-Schmidt used to orthogonalize the matrices and precision losses caused by truncation errors during the eigenvector computation. When $n>m$ the Francis algorithm gives the worst results and combined built-in method again gives the best results. All the other methods give comparable results. For both cases the built-in method gives good results.

Then we give the scatter plot of absolute value for error in singular values of a 50 by 100 dense rectangular matrix in Figure 3.7.

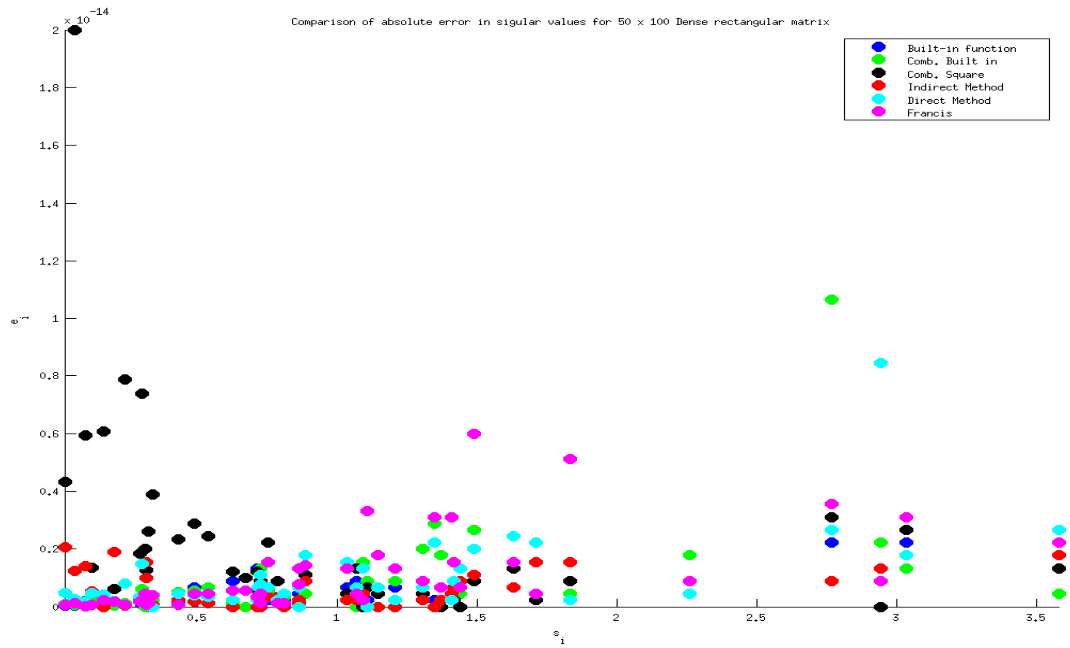

Fig. 3.7: Absolute value of error in singular values for a 50 x 100 dense rectangular matrix.

3.2 Dense Symmetric Matrix

We generated different dimensions of square dense symmetric matrices using the formula $A+A^T$. Then the comparisons of timing via different methods are tabulated below.

Table 3.5: Timings.

n	Built in SVD	Comb. Built in	Comb. Square	Indirect Method	Direct Method	Francis
100	0.0084	0.2367	1.3849	1.0093	0.2611	1.0950
200	0.0357	2.6565	6.8058	4.1679	2.9140	6.2764
300	0.0974	12.9809	22.4650	9.6180	13.6609	18.6827
400	0.1996	39.6636	56.9376	17.3091	41.9120	42.6178
500	0.3593	96.3270	123.4449	28.8710	101.4543	80.0743
600	0.6735	195.2862	236.4395	42.3036	204.3160	135.3093
700	1.0766	356.9621	412.5972	58.8551	373.1254	210.8308
800	1.6276	603.0270	677.8269	80.9572	650.2559	313.4111
900	2.2761	967.6153	1065.6365	104.0490	1004.3078	442.3627
1000	3.1573	1450.4959	1575.3845	132.3029	1517.4951	601.0980
1100	4.2206	2116.5901	2323. 8312	165.5579	2205.4484	793.6480
1200	5.5353	2992.6766	3176.0752	195.7879	3101.1535	1024.9213
1300	7.1699	4091.1788	4308.79466	235.7230	4240.1016	1301.1876
1400	8.9552	5518.5362	5768.1958	284.2645	5698.9596	1622.8149
1500	11.4804	7238.4487	7512.5604	333.7268	7443.6220	1988.7083

From table 3.5 we see that, the built-in function gives the fastest result. The indirect method is the fastest among our developed methods because it involves fewer computations. The Francis method is slower than the indirect method because it involves bidiagonalization and Givens rotation. The combined methods are slow because they involve bidiagonalization. The direct method involves bidiagonalization and tridiagonalization so it is also not fast.

Table 3.6: Accuracy.

N	Built in SVD(10^{-12})	Comb. Built in(10^{-14})	Comb. Square(10^{-13})	Indirect Method(10^{-12})	Direct Method(10^{-13})
100	0.0381	6.5598	1.0359	0.0757	0.8580
200	0.0641	13.9198	4.3588	0.1186	1.9017
300	0.0808	17.4939	7.4608	0.2519	2.6218
400	0.1001	26.3540	8.0550	0.3714	3.5517
500	0.1115	31.1084	285.8391	11.0446	4.1857
600	0.1176	43.3248	26.5554	0.6932	5.1325
700	0.1517	47.4445	19.3878	0.6158	6.0305
800	0.1757	49.2307	32.0779	0.8446	6.6398
900	0.1589	60.5039	291.7658	3.6274	8.0712
1000	0.1888	65.6004	218.5062	1.4801	8.1734
1100	0.2158	75.9083	101.5951	1.2652	9.9128
1200	0.2295	84.8737	89.0598	0.8906	10.1115
1300	0.2284	84.2658	129.8546	13.6132	10.9471
1400	0.2890	100.0798	935.8007	3.6330	12.0308
1500	0.3093	99.1563	194.3723	6.3404	13.2002

From table 3.6 we see that the built-in one, comb. Built-in, Francis and the direct method give comparable results. The combined squaring method is the least accurate one. In Table 3.7, we give the orthogonality checks.

Table 3.7: Orthogonality check.

n	Built in SVD				Comb. Built in				Comb. Square			
	UU^T-I (10^{-12})	U^TU-I (10^{-12})	VV^T-I (10^{-12})	V^TV-I (10^{-12})	UU^T-I (10^{-12})	U^TU-I (10^{-12})	VV^T-I (10^{-12})	V^TV-I (10^{-12})	UU^T-I (10^{-12})	U^TU-I (10^{-12})	VV^T-I (10^{-12})	V^TV-I (10^{-12})
100	0.0204	0.0198	0.0198	0.0202	0.0337	0.0350	0.0312	0.0345	0.0301	0.0310	0.0284	0.0326
200	0.0329	0.0271	0.0309	0.0294	0.0845	0.0795	0.0682	0.0749	0.0793	0.0765	0.0669	0.0719
300	0.0424	0.0404	0.0415	0.0414	0.1216	0.1074	0.1167	0.1085	0.1172	0.1038	0.1032	0.1072
400	0.0534	0.0457	0.0532	0.0460	0.1463	0.1482	0.1350	0.1482	0.1397	0.1456	0.1270	0.1445
500	0.0614	0.0529	0.0630	0.0535	0.1968	0.1909	0.1842	0.1848	0.1916	0.1860	0.1737	0.1838
600	0.0749	0.0677	0.0717	0.0638	0.2710	0.2340	0.2416	0.2180	0.2645	0.2321	0.2325	0.2163
700	0.0803	0.0723	0.0831	0.0832	0.2802	0.2831	0.3003	0.2705	0.2714	0.2832	0.2944	0.2669
800	0.0945	0.0768	0.0989	0.0827	0.2999	0.3127	0.3825	0.3326	0.2898	0.3112	0.3724	0.3206
900	0.1049	0.0964	0.1098	0.0963	0.3710	0.3538	0.3228	0.3622	0.3583	0.3484	0.3093	0.3559
1000	0.1111	0.0854	0.1084	0.0849	0.3835	0.3987	0.4092	0.3905	0.3724	0.3950	0.4056	0.3842
1100	0.1241	0.1014	0.1184	0.0952	0.4936	0.4442	0.4809	0.4345	0.4803	0.4389	0.4667	0.4280
1200	0.1345	0.0998	0.1433	0.1041	0.4905	0.4891	0.4833	0.4678	0.4834	0.4786	0.4671	0.4592
1300	0.1448	0.1059	0.1463	0.1109	0.5222	0.5108	0.5178	0.4987	0.5139	0.5042	0.5088	0.4912
1400	0.1690	0.1219	0.1601	0.1227	0.5308	0.5446	0.5221	0.5436	0.5094	0.5407	0.5094	0.5387
1500	0.1752	0.1214	0.1605	0.1247	0.5960	0.5905	0.5866	0.5801	0.5771	0.5825	0.5673	0.5762

n	Indirect Method				Direct Method			
	UU^T-I (10^{-13})	U^TU-I (10^{-13})	VV^T-I (10^{-13})	V^TV-I (10^{-13})	UU^T-I (10^{-12})	U^TU-I (10^{-12})	VV^T-I (10^{-12})	V^TV-I (10^{-12})
100	0.0368	0.0379	0.0386	0.0400	0.2877	0.1978	0.2878	0.1900
200	0.0679	0.0799	0.0660	0.0674	0.1169	0.1334	0.1002	0.1149
300	0.0960	0.1119	0.0893	0.0934	0.1505	0.1811	0.1748	0.1952
400	0.1271	0.1358	0.1118	0.1163	0.2119	0.2652	0.2343	0.2566
500	0.1463	0.1726	0.1353	0.1500	29.4924	12.4785	31.1799	12.4597
600	0.1828	0.2156	0.1568	0.1735	0.3383	0.3438	0.3013	0.3409
700	0.1987	0.2334	0.1786	0.2058	1.2887	0.8127	1.2705	0.7813
800	0.2223	0.2721	0.1996	0.2251	0.3844	0.4247	0.4469	0.4281
900	0.2496	0.3039	0.2217	0.2456	2.0562	1.1971	1.9739	1.1683

1000	0.2733	0.3304	0.2423	0.2783	1.6912	1.3070	1.6830	1.2836
1100	0.2982	0.3502	0.2705	0.2960	1.2372	1.1390	1.4133	1.1229
1200	0.3283	0.3921	0.2872	0.3246	0.6063	0.6759	0.5827	0.6804
1300	0.3436	0.4173	0.3203	0.3519	1.0000	0.9620	1.0636	0.9434
1400	0.3618	0.4543	0.3333	0.3692	1.1033	1.1004	1.1794	0.9989
1500	0.3850	0.4783	0.3620	0.3907	0.7007	0.7327	0.7262	0.7487

Table 3.8 lists the errors in singular values using different methods. We see that the indirect method, combined built-in and squaring, Francis and the direct methods give comparable results.

Table 3.8: Error in singular values

n	Comb. Built in (10^{-12})	Comb. Square(10^{-11})	Indirect Method(10^{-11})	Direct Method(10^{-11})	Francis(10^{-12})
100	0.0391	0.0421	0.0046	0.0121	0.1315
200	0.0711	0.3000	0.0113	0.0256	0.7816
300	0.0639	0.4399	0.1118	0.0242	1.3571
400	0.0924	250.2830	180.1202	0.0341	2.1174
500	0.0995	5.2806	0.5130	0.0448	2.6148
600	0.0995	61.9305	4.6881	0.0625	3.1406
700	0.1563	2.7922	0.6248	0.0497	4.5901
800	0.1421	2.2875	0.6112	0.0767	8.2707
900	0.1066	12.6837	8.1206	0.0952	5.5422
1000	0.1208	2.7780	1.4873	0.0867	9.5497
1100	0.0995	5.1229	1.7121	0.1293	10.8002
1200	0.2416	41.3298	6.9733	0.0995	11.4824
1300	0.1563	10.8642	0.7514	0.1251	12.6761
1400	0.1847	22.6280	29.1753	0.1251	12.3208
1500	0.2274	3.4415	1.8547	0.1137	21.0036

3.3 Sparse Symmetric matrix

For completeness sake we test some of the methods on some randomly generated sparse matrices. We tested for three different cases of sparse matrices where the percentages of non-zero elements are 10%, 15% and 20% respectively. The sparsely generated matrices were converted to full matrices as our coded methods cannot operate on sparse matrices. Also, the built-in function for sparse matrix will only give the six biggest singular values for sparse matrices. As expected there is a drastic drop in performance when our tested methods are applied on sparse matrices.

From Fig. 3.8 we see that for all cases the built in method gives better results. The timings of direct and indirect method are comparable. As the matrices become denser via the increase of non-zero elements, the methods become faster.

Table 3.9 shows that the built-in method has the best accuracy. The direct and indirect methods get comparable accuracy. Table 3.10 shows the orthogonality check. Table 3.11 gives the error in singular values. During the computing of singular value errors, we find that the errors for smaller singular values are more obvious than those for the larger singular values. The built-in function gives better results than the other two. The direct method gives the lowest accurate results.

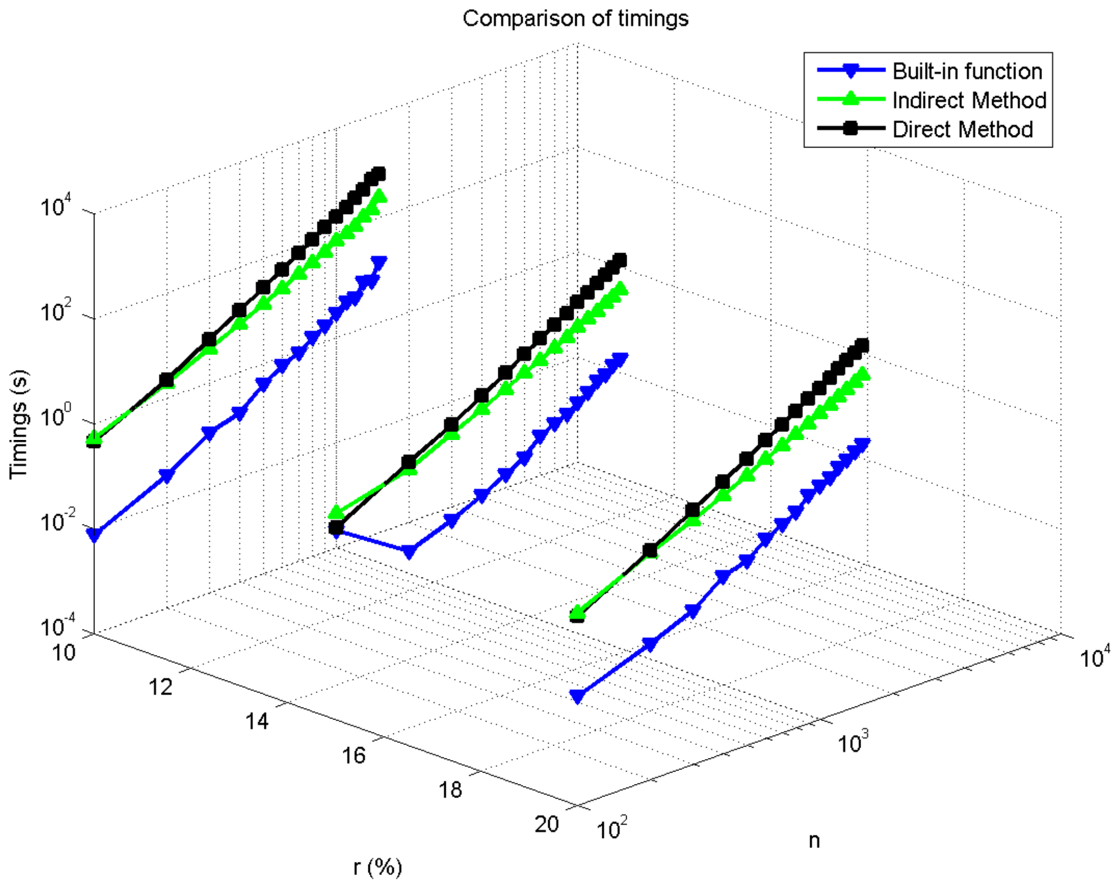

Fig. 3.8: Comparison of timings.

Table 3.9: Accuracies

r (%)	m	Built in (10^{-11})	Indirect (10^{-8})	Direct (10^{-10})	r(%)	m	Built in(10^{-11})	Indirect (10^{-8})	Direct (10^{-10})
	100	0.0083	0.0001	0.0025		100	0.0135	0.0000	0.0051
	200	0.0253	0.0002	0.0123		200	0.0252	0.0222	0.0164
	300	0.0285	0.0001	0.0213		300	0.0427	0.0002	0.0292
	400	0.0401	0.0004	0.0537		400	0.0521	0.0019	0.0612
	500	0.0483	0.0003	0.0581		500	0.0793	0.0042	0.0522
	600	0.0597	0.0019	0.0860		600	0.0848	0.0020	0.1396
	700	0.0713	0.0012	0.0930		700	0.1038	0.0040	0.1485
10	800	0.0917	0.0049	0.1161	20	800	0.1230	0.0059	0.1764
	900	0.1056	0.0021	0.1739		900	0.1342	0.0099	0.2245
	1000	0.1066	0.0017	0.2143		1000	0.1626	0.0017	0.2595
	1100	0.1253	0.0018	0.2691		1100	0.1736	0.0020	0.3194
	1200	0.1386	0.0025	0.2956		1200	0.2046	0.0055	0.3127
	1300	0.1564	0.3657	0.2932		1300	0.2263	0.0094	0.3824
	1400	0.1782	0.0027	0.3414		1400	0.2436	0.0029	0.4027
	1500	0.1920	0.0037	0.3150		1500	0.2730	0.0032	0.5615
	100	0.0113	0.0000	0.0034					
	200	0.0222	0.0002	0.0138					
	300	0.0430	0.0003	0.0291					
	400	0.0464	0.0001	0.0498					
	500	0.0585	0.0004	0.0671					
	600	0.0720	0.0004	0.0854					

15	700	0.0972	0.0017	0.1301
	800	0.1039	0.0007	0.1617
	900	0.1224	0.0027	0.1682
	1000	0.1296	0.0021	0.2237
	1100	0. 1536	0.0015	0.2657
	1200	0.1717	0.0017	0.2902
	1300	0.1917	0.0019	0.3581
	1400	0.2190	0.0165	0.4453
	1500	0.2406	0.0105	0.4579

Table 3.10: Orthogonality checks

r (%)	m	Built in SVD				Indirect Method				Direct Method			
		UU^T-I (10^{-12})	U^TU-I (10^{-12})	VV^T-I (10^{-12})	V^TV-I (10^{-12})	UU^T-I (10^{-13})	U^TU-I (10^{-13})	VV^T-I (10^{-13})	V^TV-I (10^{-13})	UU^T-I (10^{-13})	U^TU-I (10^{-13})	VV^T-I (10^{-11})	V^TV-I (10^{-12})
10	100	0.0191	0.0202	0.0177	0.0181	0.0363	0.0389	0.0336	0.0342	0.0833	0.0773	0.0045	0.0257
	200	0.0298	0.0268	0.0279	0.0259	0.0645	0.0734	0.0701	0.0723	0.1449	0.1341	0.0155	0.0609
	300	0.0360	0.0339	0.0370	0.0388	0.0952	0.1023	0.0846	0.0856	0.1756	0.1719	0.0231	0.0915
	400	0.0475	0.0471	0.0498	0.0473	0.1211	0.1328	0.1090	0.1148	0.2237	0.2033	0.0335	0.1228
	500	0.0540	0.0548	0.0557	0.0558	0.1492	0.1721	0.1318	0.1465	0.2490	0.2434	0.0458	0.1816
	600	0.0631	0.0555	0.0640	0.0584	0.1788	0.2192	0.1601	0.1692	0.3049	0.2877	0.0603	0.1892
	700	0.0724	0.0631	0.0736	0.0637	0.1990	0.2344	0.1831	0.1905	0.3117	0.3138	0.0553	0.2098
	800	0.0795	0.0693	0.0840	0.0685	0.2234	0.2608	0.2041	0.2209	0.3522	0.3374	0.0844	0.2590
	900	0.0869	0.0777	0.0896	0.0773	0.2451	0.2947	0.2284	0.2428	0.3888	0.3650	0.0767	0.2680
	1000	0.0975	0.0858	0.1000	0.0900	0.2880	0.3337	0.2480	0.2701	0.4625	0.4003	0.0999	0.3280
	1100	0.1023	0.0843	0.1054	0.0862	0.2947	0.3682	0.2666	0.2917	0.4816	0.4276	0.1348	0.3897
	1200	0.1110	0.0902	0.1148	0.0933	0.3599	0.3984	0.2892	0.3177	0.4531	0.4651	0.1406	0.4091
	1300	0.1190	0.0937	0.1232	0.0982	0.3553	0.4143	0.3102	0.3460	0.5374	0.5161	0.1572	0.4149
	1400	0.1327	0.1112	0.1431	0.1106	0.3782	0.4575	0.3311	0.3637	0.3117	0.5295	0.1434	0.4790
	1500	0.1342	0.1124	0.1328	0.1115	0.4002	0.4793	0.3489	0.3874	0.3522	0.5480	0.1380	0.4730
15	100	0.0204	0.0169	0.0177	0.0179	0.0381	0.0370	0.0347	0.0315	0.0763	0.0737	0.0045	0.0303
	200	0.0272	0.0261	0.0283	0.0265	0.0665	0.0699	0.0677	0.0690	0.1364	0.1407	0.0132	0.0655
	300	0.0374	0.0370	0.0381	0.0367	0.0998	0.1049	0.0865	0.0960	0.1838	0.1772	0.0215	0.0941
	400	0.0472	0.0436	0.0467	0.0434	0.1248	0.1380	0.1106	0.1182	0.2117	0.2163	0.0278	0.1232
	500	0.0583	0.0510	0.0565	0.0492	0.1464	0.1681	0.1390	0.1535	0.2720	0.2408	0.0293	0.1470
	600	0.0644	0.0549	0.0655	0.0562	0.1736	0.2046	0.1566	0.1779	0.2904	0.2906	0.0464	0.1797
	700	0.0739	0.0658	0.0761	0.0668	0.2159	0.2643	0.1836	0.1926	0.3290	0.3125	0.0693	0.2303
	800	0.0811	0.0696	0.0814	0.0715	0.2240	0.2714	0.2052	0.2200	0.3507	0.3424	0.0754	0.2618
	900	0.0899	0.0770	0.0888	0.0785	0.2482	0.3004	0.2259	0.2461	0.4144	0.3773	0.0786	0.3139
	1000	0.0921	0.0959	0.0953	0.0926	0.2752	0.3365	0.2457	0.2697	0.4272	0.4398	0.0941	0.3304
	1100	0.1031	0.0840	0.1065	0.0867	0.2992	0.3593	0.2642	0.2885	0.4436	0.4171	0.1107	0.3705
	1200	0.1107	0.0891	0.1132	0.0952	0.3285	0.3998	0.2879	0.3180	0.4773	0.4327	0.1006	0.3914
	1300	0.1215	0.0961	0.1212	0.1040	0.3428	0.4085	0.3043	0.3405	0.5173	0.5083	0.1353	0.4640
	1400	0.1294	0.1053	0.1287	0.1066	0.3737	0.4606	0.3280	0.3665	0.5517	0.5274	0.1552	0.4412
	1500	0.1364	0.1106	0.1448	0.1131	0.3930	0.4909	0.3460	0.3985	0.6033	0.5628	0.1622	0.4875
20	100	0.0200	0.0182	0.0205	0.0208	0.0325	0.0364	0.0376	0.0383	0.0738	0.0731	0.0072	0.0313
	200	0.0297	0.0274	0.0293	0.0272	0.0657	0.0742	0.0673	0.0775	0.1245	0.1295	0.0128	0.0594
	300	0.0377	0.0357	0.0372	0.0370	0.0906	0.1048	0.0866	0.0931	0.1648	0.1752	0.0173	0.0864
	400	0.0462	0.0405	0.0456	0.0439	0.1186	0.1447	0.1152	0.1218	0.2024	0.2054	0.0295	0.1265
	500	0.0523	0.0528	0.0553	0.0590	0.1469	0.1682	0.1357	0.1433	0.2500	0.2385	0.0289	0.1437
	600	0.0643	0.0606	0.0640	0.0626	0.1749	0.2006	0.1582	0.1719	0.2924	0.3091	0.0569	0.2124
	700	0.0724	0.0612	0.0722	0.0625	0.1990	0.2412	0.1777	0.1990	0.3376	0.3172	0.0626	0.2316
	800	0.0810	0.0711	0.0820	0.0742	0.2455	0.3070	0.2043	0.2213	0.3797	0.3396	0.0638	0.2549
	900	0.0861	0.0799	0.0868	0.0790	0.2454	0.3123	0.2195	0.2452	0.3995	0.3728	0.0823	0.3110
	1000	0.0938	0.0815	0.0977	0.0859	0.2734	0.3351	0.2463	0.2731	0.4117	0.3980	0.1029	0.3148
	1100	0.1019	0.0889	0.1047	0.0927	0.2899	0.3632	0.2645	0.2959	0.4815	0.4376	0.0952	0.3655
	1200	0.1132	0.0929	0.1181	0.0946	0.3210	0.4137	0.2890	0.3139	0.5188	0.4399	0.1126	0.4022
	1300	0.1210	0.0976	0.1275	0.1022	0.3426	0.4166	0.3116	0.3393	0.5374	0.4963	0.1353	0.4343

	1400	0.1263	0.1024	0.1294	0.1032	0.3671	0.4591	0.3243	0.3629	0.5517	0.5210	0.1552	0.4710
	1500	0.1385	0.1049	0.1411	0.1118	0.3966	0.4805	0.3497	0.3916	0.5877	0.5569	0.1622	0.5375

Table 3.11: Error in singular values

r (%)	m	Indirect Method (10^{-11})	Direct Method(10^{-11})	r (%)	m	Indirect Method(10^{-11})	Direct Method (10^{-11})
10	100	0.0395	0.0045	20	100	0.0081	0.0082
	200	0.0763	0.0181		200	10.1266	0.0242
	300	0.0140	0.0306		300	0.0858	0.0485
	400	0.1739	0.0444		400	0.6645	0.0799
	500	0.0925	0.0723		500	1.7377	0.0700
	600	0.3237	0.0885		600	0.8445	0.1652
	700	0.4709	0.1030		700	1.1565	0.1513
	800	1.4763	0.1766		800	1.9781	0.2043
	900	0.3357	0.1521		900	3.4608	0.2427
	1000	0.3953	0.1869		1000	0.5732	0.2874
	1100	0.4629	0.2633		1100	0.4637	0.3705
	1200	0.8129	0.2530		1200	2.0077	0.4214
	1300	126.5751	0.2782		1300	2.1540	0.3858
	1400	0.6724	0.3798		1400	0.6398	0.4725
	1500	1.2641	0.2913		1500	0.6488	0.6629
15	100	0.0031	0.0094				
	200	0.0743	0.0220				
	300	0.1113	0.0352				
	400	0.0082	0.0538				
	500	0.0945	0.0689				
	600	0.1448	0.1023				
	700	0.2592	0.1286				
	800	0.2002	0.1485				
	900	0.6687	0.2320				
	1000	0.4829	0.2160				
	1100	0.2563	0.2959				
	1200	0.1536	0.3084				
	1300	0.4854	0.3691				
	1400	6.3058	0.3606				
	1500	3.1438	0.5226				

4. REFERENCES

[1] P. Deift, J. Demmel, C. Li and C. Tomei. The bidiagonal singular value decomposition and Hamiltonian mechanics. SIAM J. Numer. Anal., v. 18, n. 5, pp. 1463-1516, 1991.

[2] D. Watkins. Fundamentals of matrix computations. Third Edition. Wiley, 2010.

[3] V. Klema, A. Laub. The singular value decomposition: its computation and some applications. IEEE Trans. Auto. Cont., v. AC-25, n. 2, pp. 164-176, 1980.

[4] G. Golub, W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. J. SIAM. Numer. Anal. Ser. B, v. 2, n. 2, pp. 205-224, 1965.

[5] M. Gu, S. Eisenstat. A divide-and-conquer algorithm for the bidiagonal SVD. SIAM J. Matrix Anal. Appl., v. 16, n. 1, pp. 79-92, 1995.

[6] J. Demmel, W. KahanAccurate. Singular Values of Bidiagonal Matrices. SIAM J. Sci. Stat. Comput., v. 11, n. 5, pp. 873-912, 1990.