

L6a Singular Value Decomposition - Algorithms and Error Analysis

April 19, 2018

1 Singular Value Decomposition - Algorithms and Error Analysis

We study only algorithms for real matrices, which are most commonly used in the applications described in this course.

For more details, see Section ?? and the references therein.

1.1 Prerequisites

The reader should be familiar with facts about the singular value decomposition and perturbation theory and algorithms for the symmetric eigenvalue decomposition.

1.2 Competences

The reader should be able to apply an adequate algorithm to a given problem, and to assess the accuracy of the solution.

1.3 Basics

1.3.1 Definitions

The **singular value decomposition** (SVD) of $A \in \mathbb{R}^{m \times n}$ is $A = U\Sigma V^T$, where $U \in \mathbb{R}^{m \times m}$ is orthogonal, $U^T U = U U^T = I_m$, $V \in \mathbb{R}^{n \times n}$ is orthogonal, $V^T V = V V^T = I_n$, and $\Sigma \in \mathbb{R}^{m \times n}$ is diagonal with singular values $\sigma_1, \dots, \sigma_{\min\{m,n\}}$ on the diagonal.

If $m > n$, the **thin SVD** of A is $A = U_{1:m,1:n} \Sigma_{1:n,1:n} V^T$.

1.3.2 Facts

1. Algorithms for computing SVD of A are modifications of algorithms for the symmetric eigenvalue decomposition of the matrices AA^T , $A^T A$ and $\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$.
2. Most commonly used approach is the three-step algorithm:
 1. Reduce A to bidiagonal matrix B by orthogonal transformations, $X^T A Y = B$.
 2. Compute the SVD of B with QR iterations, $B = W \Sigma Z^T$.
 3. Multiply $U = XW$ and $V = YZ$.

3. If $m \geq n$, the overall operation count for this algorithm is $O(mn^2)$ operations.
4. **Error bounds:** Let $U\Sigma V^T$ and $\tilde{U}\tilde{\Sigma}\tilde{V}^T$ be the exact and the computed SVDs of A , respectively. The algorithms generally compute the SVD with errors bounded by

$$|\sigma_i - \tilde{\sigma}_i| \leq \phi\epsilon\|A\|_2, \quad \|u_i - \tilde{u}_i\|_2, \|v_i - \tilde{v}_i\|_2 \leq \psi\epsilon \frac{\|A\|_2}{\min_{j \neq i} |\sigma_i - \tilde{\sigma}_j|},$$

where ϵ is machine precision and ϕ and ψ are slowly growing polynomial functions of n which depend upon the algorithm used (typically $O(n)$ or $O(n^2)$). These bounds are obtained by combining perturbation bounds with the floating-point error analysis of the algorithms.

1.4 Bidiagonalization

1.4.1 Facts

1. The reduction of A to bidiagonal matrix can be performed by applying $\min\{m-1, n\}$ Householder reflections H_L from the left and $n-2$ Householder reflections H_R from the right. In the first step, H_L is chosen to annihilate all elements of the first column below the diagonal, and H_R is chosen to annihilate all elements of the first row right of the first super-diagonal. Applying this procedure recursively yields the bidiagonal matrix.
2. H_L and H_R do not depend on the normalization of the respective Householder vectors v_L and v_R . With the normalization $[v_L]_1 = [v_R]_1 = 1$, the vectors v_L are stored in the lower-triangular part of A , and the vectors v_R are stored in the upper-triangular part of A above the super-diagonal.
3. The matrices H_L and H_R are not formed explicitly - given v_L and v_R , A is overwritten with $H_L A H_R$ in $O(mn)$ operations by using matrix-vector multiplication and rank-one updates.
4. Instead of performing rank-one updates, p transformations can be accumulated, and then applied. This **block algorithm** is rich in matrix-matrix multiplications (roughly one half of the operations is performed using BLAS 3 routines), but it requires extra workspace.
5. If the matrices X and Y are needed explicitly, they can be computed from the stored Householder vectors. In order to minimize the operation count, the computation starts from the smallest matrix and the size is gradually increased.
6. The backward error bounds for the bidiagonalization are as follows: The computed matrix \tilde{B} is equal to the matrix which would be obtained by exact bidiagonalization of some perturbed matrix $A + \Delta A$, where $\|\Delta A\|_2 \leq \psi\epsilon\|A\|_2$ and ψ is a slowly increasing function of n . The computed matrices \tilde{X} and \tilde{Y} satisfy $\tilde{X} = X + \Delta X$ and $\tilde{Y} = Y + \Delta Y$, where $\|\Delta X\|_2, \|\Delta Y\|_2 \leq \phi\epsilon$ and ϕ is a slowly increasing function of n .
7. The bidiagonal reduction is implemented in the [LAPACK](#) subroutine [DGEHRD](#). The computation of X and Y is implemented in the subroutine [DORGBR](#), which is not yet wrapped in Julia.
8. Bidiagonalization can also be performed using Givens rotations. Givens rotations act more selectively than Householder reflectors, and are useful if A has some special structure, for example, if A is a banded matrix. Error bounds for function `myBidiagG()` are the same as above, but with slightly different functions ψ and ϕ .

```
In [1]: m=8
        n=5
        s=srand(421)
        A=map(Float64,rand(-9:9,m,n))
```

```
Out[1]: 8x5 Array{Float64,2}:
 -8.0  -1.0  -2.0   2.0  -9.0
 -6.0   7.0   1.0   1.0   5.0
  3.0   2.0   9.0  -3.0   1.0
  7.0  -4.0  -8.0   0.0  -5.0
 -7.0   1.0  -4.0   3.0  -1.0
  0.0   0.0  -2.0   3.0  -8.0
  2.0   2.0  -2.0  -7.0  -6.0
 -7.0  -3.0  -8.0  -8.0  -5.0
```

```
In [2]: ?LAPACK.gebrd!
```

```
Out[2]:
```

```
gebrd!(A) -> (A, d, e, tauq, taup)
```

Reduce A in-place to bidiagonal form $A = QBP'$. Returns A, containing the bidiagonal matrix B; d, containing the diagonal elements of B; e, containing the off-diagonal elements of B; tauq, containing the elementary reflectors representing Q; and taup, containing the elementary reflectors representing P.

```
In [3]: # We need copy()
        Outg=LAPACK.gebrd!(copy(A))
```

```
Out[3]: ([16.1245 5.13847 0.0827436 -0.330974; 0.24871 -13.9844 0.106485 0.452059; ; -0.08290
```

```
In [4]: B=Bidiagonal(Outg[2],Outg[3][1:end-1],true)
```

```
Out[4]: 5x5 Bidiagonal{Float64}:
 16.1245    5.13847
          -13.9844    13.1844
                   -9.58104    -3.25817
                          -11.0389    -0.512951
                               -10.2308
```

```
In [5]: svdvals(A), svdvals(B)
```

```
Out[5]: ([21.167, 16.1921, 11.6027, 10.1944, 6.01866], [21.167, 16.1921, 11.6027, 10.1944, 6.018
```

```
In [6]: # Extract X
        function myBidiagX(H::Array)
            m,n=size(H)
            T=Float64
            X = eye(T,m,n)
            v=Array{T}(m)
```

```

    for j = n : -1 : 1
        v[j] = one(T)
        v[j+1:m] = H[j+1:m, j]
            = -2 / (v[j:m]v[j:m])
        w = * X[j:m, j:n]'*v[j:m]
        X[j:m, j:n] = X[j:m, j:n] + v[j:m]*w'
    end
    X
end

# Extract Y
function myBidiagY(H::Array)
    n,m=size(H)
    T=Float64
    Y = eye(T,n)
    v=Array{T}(n)
    for j = n-2 : -1 : 1
        v[j+1] = one(T)
        v[j+2:n] = H[j+2:n, j]
            = -2 / (v[j+1:n]v[j+1:n])
        w = * Y[j+1:n, j+1:n]'*v[j+1:n]
        Y[j+1:n, j+1:n] = Y[j+1:n, j+1:n] + v[j+1:n]*w'
    end
    Y
end

```

Out [6]: myBidiagY (generic function with 1 method)

In [7]: X=myBidiagX(Outg[1])

Out [7]: 8E5 Array{Float64,2}:

-0.496139	0.218152	-0.228942	0.139865	0.548494
-0.372104	-0.123781	0.721083	-0.269817	-0.14492
0.186052	-0.400271	-0.186877	-0.381146	0.418489
0.434122	0.622108	0.0728932	0.142778	-0.19147
-0.434122	0.144278	0.211698	0.20254	-0.0701254
0.0	0.407358	0.191111	0.085435	0.527199
0.124035	0.363177	0.146036	-0.748373	0.143439
-0.434122	0.266831	-0.533594	-0.365505	-0.403642

In [8]: Y=myBidiagY(Outg[1]')

Out [8]: 5E5 Array{Float64,2}:

1.0	0.0	0.0	0.0	0.0
0.0	-0.458631	-0.812867	0.329131	-0.143449
0.0	0.736223	-0.234864	0.180671	-0.608418
0.0	-0.120692	-0.242442	-0.907227	-0.321859
0.0	0.482769	-0.474666	-0.189655	0.711096

```
In [9]: # Orthogonality
```

```
vecnorm(X'*X-I), vecnorm(Y'*Y-I)
```

```
Out[9]: (1.4131996528429256e-15, 4.933896999410976e-16)
```

```
In [10]: # Error
```

```
X'*A*Y-B
```

```
Out[10]: 5E5 Array{Float64,2}:
```

```
-1.06581e-14 -3.55271e-15 1.72724e-15 3.84939e-16 -1.41118e-15
6.10623e-16 0.0 0.0 4.49386e-16 6.59131e-16
-1.11022e-15 2.27043e-15 5.32907e-15 0.0 -1.09631e-15
1.4988e-15 -3.29434e-16 1.34701e-15 3.55271e-15 5.55112e-16
2.55351e-15 1.62984e-15 -2.28131e-15 -1.94344e-15 1.77636e-15
```

```
In [11]: # Bidiagonalization using Givens rotations
```

```
function myBidiagG{T}(A::Array{T})
    m,n=size(A)
    X=eye(T,m,m)
    Y=eye(T,n,n)
    for j = 1 : n
        for i = j+1 : m
            G,r=givens(A,j,i,j)
            A=G*A
            X=G*X
        end
        for i=j+2:n
            G,r=givens(A',j+1,i,j)
            A=A*G'
            Y*=G'
        end
    end
    X',Bidiagonal(diag(A),diag(A,1),true), Y
end
```

```
Out[11]: myBidiagG (generic function with 1 method)
```

```
In [12]: X1, B1, Y1 = myBidiagG(A)
```

```
Out[12]: ([0.496139 0.218152 0.461552 -0.308972; 0.372104 -0.123781 0.388273 0.248552; ; -0.1
diag: -16.124515496597102 13.984444568826952 10.23083099487791
super: 5.138467296173651 -13.184384910674824 -0.5129509442085607, [1.0 0.0 0.0 0.
```

```
In [13]: # Orthogonality
```

```
vecnorm(X1'*X1-I), vecnorm(Y1'*Y1-I)
```

```
Out[13]: (8.706420678153364e-16, 4.728461161860565e-16)
```

```
In [14]: # Diagonalization
```

```
X1'*A*Y1
```

```
Out[14]: 8E5 Array{Float64,2}:
-16.1245      5.13847      -2.21042e-15   3.20822e-16  -7.96603e-16
 8.88178e-16  13.9844      -13.1844      -2.65924e-16  -2.41072e-15
-1.22125e-15  -8.7425e-16   9.58104      3.25817      3.08144e-15
 1.11022e-15  -1.90283e-16  -1.56095e-15  11.0389      -0.512951
-1.22125e-15  3.29889e-16   1.41e-15     1.19112e-15  10.2308
 0.0          -1.54899e-15  7.40357e-16  -6.82381e-16  -7.39309e-16
 4.44089e-16  -3.08189e-15  -6.94843e-16  -6.544e-16   -1.45398e-15
 1.16573e-15  -7.67794e-16  -7.35941e-17  6.07855e-16  -9.74301e-18
```

```
In [15]: # X, Y and B are not unique
B
```

```
Out[15]: 5E5 Bidiagonal{Float64}:
 16.1245      5.13847
      -13.9844   13.1844
              -9.58104   -3.25817
                  -11.0389   -0.512951
                      -10.2308
```

```
In [16]: B1
```

```
Out[16]: 5E5 Bidiagonal{Float64}:
-16.1245      5.13847
      13.9844   -13.1844
              9.58104   3.25817
                  11.0389   -0.512951
                      10.2308
```

1.5 Bidiagonal QR method

Let B be a real upper-bidiagonal matrix of order n and let $B = W\Sigma Z^T$ be its SVD.

All methods for computing the SVD of bidiagonal matrix are derived from the methods for computing the EVD of the tridiagonal matrix $T = B^T B$.

1.5.1 Facts

1. The shift μ is the eigenvalue of the 2×2 matrix $T_{n-1:n, n-1:n}$ which is closer to $T_{n,n}$. The first Givens rotation from the right is the one which annihilates the element $(1, 2)$ of the shifted 2×2 matrix $T_{1:2, 1:2} - \mu I$. Applying this rotation to B creates the bulge at the element $B_{2,1}$. This bulge is subsequently chased out by applying adequate Givens rotations alternating from the left and from the right. This is the **Golub-Kahan algorithm**.
2. The computed SVD satisfies error bounds from the Fact 4 above.
3. The special variant of zero-shift QR algorithm (the **Demmel-Kahan algorithm**) computes the singular values with high relative accuracy.
4. The tridiagonal divide-and-conquer method, bisection and inverse iteration, and MRRR method can also be adapted for bidiagonal matrices.

5. Zero shift QR algorithm for bidiagonal matrices is implemented in the LAPACK routine [DBDSQR](#). It is also used in the function `svdvals()`. Divide-and-conquer algorithm for bidiagonal matrices is implemented in the LAPACK routine [DBDSDC](#). However, this algorithm also calls zero-shift QR to compute singular values.

1.5.2 Examples

```
In [17]: W,,Z=svd(B)
```

```
Out[17]: ([0.38462 0.9133 -0.0153442 0.0997382; -0.865161 0.307806 -0.0389653 0.347244; ; 0.0
```

```
In [18]: @which svd(B)
```

```
Out[18]: svd(A::Union{AbstractArray, Number}) in Base.LinAlg at linalg\svd.jl:106
```

```
In [19]: 1=svdvals(B)
```

```
Out[19]: 5-element Array{Float64,1}:
 21.167
 16.1921
 11.6027
 10.1944
  6.01866
```

```
In [20]: @which svdvals(B)
```

```
Out[20]: svdvals(A::AbstractArray{#s268,2} where #s268<:Union{Complex{Float32}, Complex{Float64}}
```

```
In [21]: -1
```

```
Out[21]: 5-element Array{Float64,1}:
-3.55271e-15
 3.55271e-15
-1.77636e-15
-3.55271e-15
-1.77636e-15
```

```
In [22]: ?LAPACK.bdsqr!
```

```
Out[22]:
```

```
bdsqr!(uplo, d, e_, Vt, U, C) -> (d, Vt, U, C)
```

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e_` on the off-diagonal. If `uplo = U`, `e_` is the superdiagonal. If `uplo = L`, `e_` is the subdiagonal. Can optionally also compute the product $Q' * C$.

Returns the singular values in `d`, and the matrix `C` overwritten with $Q' * C$.

```
In [23]: BV=eye(n)
          BU=eye(n)
          BC=eye(n)
          2,Z2,W2,C = LAPACK.bdsqr!('U',copy(B.dv),copy(B.ev),BV,BU,BC)
```

```
Out[23]: ([21.167, 16.1921, 11.6027, 10.1944, 6.01866], [0.292994 0.664957 -0.0676505 -0.001116
```

```
In [24]: W2'*full(B)*Z2'
```

```
Out[24]: 5E5 Array{Float64,2}:
 21.167      -1.12726e-15   5.49488e-16   2.9672e-17   -2.27569e-15
-1.09667e-16 16.1921      -1.45718e-14  -1.24393e-17  3.18775e-15
-3.03192e-17 -1.33137e-16  11.6027      -1.44369e-16  -8.44255e-16
-3.09388e-17 -5.93405e-18  -4.57894e-17  10.1944      -1.12256e-16
 2.91257e-17  4.6231e-16   -5.22312e-17  3.75542e-16   6.01866
```

```
In [25]: ?LAPACK.bdsdc!
```

```
Out[25]:
```

```
bdsdc!(uplo, compq, d, e_) -> (d, e, u, vt, q, iq)
```

Computes the singular value decomposition of a bidiagonal matrix with d on the diagonal and $e_$ on the off-diagonal using a divide and conquer method. If $uplo = U$, $e_$ is the superdiagonal. If $uplo = L$, $e_$ is the subdiagonal. If $compq = N$, only the singular values are found. If $compq = I$, the singular values and vectors are found. If $compq = P$, the singular values and vectors are found in compact form. Only works for real types.

Returns the singular values in d , and if $compq = P$, the compact singular vectors in iq .

```
In [26]: 3,ee,W3,Z3,rest=LAPACK.bdsdc!('U','I',copy(B.dv),copy(B.ev))
```

```
Out[26]: ([21.167, 16.1921, 11.6027, 10.1944, 6.01866], e = 2.7182818284590..., [0.38462 0.9133
```

```
In [27]: W3'*full(B)*Z3'
```

```
Out[27]: 5E5 Array{Float64,2}:
 21.167      -1.12726e-15   5.49488e-16   2.9672e-17   -2.27569e-15
-1.09667e-16 16.1921      -1.45718e-14  -1.24393e-17  3.18775e-15
-3.03192e-17 -1.33137e-16  11.6027      -1.44369e-16  -8.44255e-16
-3.09388e-17 -5.93405e-18  -4.57894e-17  10.1944      -1.12256e-16
 2.91257e-17  4.6231e-16   -5.22312e-17  3.75542e-16   6.01866
```

Functions `svd()`, `LAPACK.bdsqr!()` and `LAPACK.bdsdc!()` use the same algorithm to compute singular values.

```
In [28]: [3-2 3-]
```

```
Out[28]: 5E2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0
 0.0  0.0
 0.0  0.0
 0.0  0.0
```

Let us compute some timings. We observe $O(n^2)$ operations.


```
In [29]: n=1000
        Abig=Bidiagonal(rand(n),rand(n-1),true)
        Bbig=Bidiagonal(rand(2*n),rand(2*n-1),true)
        @time svdvals(Abig);
        @time svdvals(Bbig);
        @time LAPACK.bdsdc!('U','N',copy(Abig.dv),copy(Abig.ev));
        @time svd(Abig);
        @time svd(Bbig);

0.029130 seconds (98 allocations: 163.529 KiB)
0.100129 seconds (19 allocations: 313.844 KiB)
0.030543 seconds (16 allocations: 141.688 KiB)
0.101864 seconds (27 allocations: 45.900 MiB, 5.12% gc time)
0.549771 seconds (27 allocations: 183.351 MiB, 31.99% gc time)
```

1.6 QR method

Final algorithm is obtained by combining bidiagonalization and bidiagonal SVD methods. Standard method is implemented in the LAPACK routine [DGESVD](#). Divide-and-conquer method is implemented in the LAPACK routine [DGESDD](#).

The functions `svd()`, `svdvals()`, and `svdvecs()` use `DGESDD`. Wrappers for `DGESVD` and `DGESDD` give more control about output of eigenvectors.

```
In [30]: # The built-in algorithm
        U,A,V=svd(A)
```

```
Out[30]: ([-0.447559 -0.33734  0.537416 -0.205771; 0.185021 -0.545965  -0.147439 0.632901;  ; -0
```

```
In [31]: # With our building blocks
        U1=X*W
        V1=Y*Z
        U1'*A*V1
```

```
Out[31]: 5E5 Array{Float64,2}:
        21.167      -1.19942e-15   2.50215e-16  -7.44079e-16  -3.88974e-15
        -4.14376e-15  16.1921      -1.39619e-14  2.80928e-16  4.60623e-15
        1.77657e-15  -2.61637e-15  11.6027      -6.96939e-17  -7.90882e-16
        7.00688e-15   9.32618e-16  -1.96823e-15  10.1944      1.20843e-16
        5.31273e-17  -3.35754e-16  3.18964e-17  2.09432e-15  6.01866
```

```
In [32]: ?LAPACK.gesvd!
```

```
Out[32]:
```

```
gesvd!(jobu, jobvt, A) -> (U, S, VT)
```

Finds the singular value decomposition of A , $A = U * S * V'$. If `jobu = A`, all the columns of U are computed. If `jobvt = A` all the rows of V' are computed. If `jobu = N`, no columns of

U are computed. If `jobvt = N` no rows of V' are computed. If `jobu = 0`, A is overwritten with the columns of (thin) U. If `jobvt = 0`, A is overwritten with the rows of (thin) V' . If `jobu = S`, the columns of (thin) U are computed and returned separately. If `jobvt = S` the rows of (thin) V' are computed and returned separately. `jobu` and `jobvt` can't both be 0.

Returns U, S, and Vt, where S are the singular values of A.

```
In [33]: # DGESVD
```

```
LAPACK.gesvd!('A', 'A', copy(A))
```

```
Out[33]: ([-0.447559 0.33734 0.00795829 0.00793487; 0.185021 0.545965 -0.22584 0.368283; ; -0
```

```
In [34]: ?LAPACK.gesdd!
```

```
Out[34]:
```

```
gesdd!(job, A) -> (U, S, VT)
```

Finds the singular value decomposition of A, $A = U * S * V'$, using a divide and conquer approach. If `job = A`, all the columns of U and the rows of V' are computed. If `job = N`, no columns of U or rows of V' are computed. If `job = 0`, A is overwritten with the columns of (thin) U and the rows of (thin) V' . If `job = S`, the columns of (thin) U and the rows of (thin) V' are computed and returned separately.

```
In [35]: LAPACK.gesdd!('N', copy(A))
```

```
Out[35]: (Array{Float64}(8,0), [21.167, 16.1921, 11.6027, 10.1944, 6.01866], Array{Float64}(5,0))
```

Let us perform some timings. We observe $O(n^3)$ operations.

```
In [36]: n=1000
```

```
Abig=rand(n,n)
```

```
Bbig=rand(2*n,2*n)
```

```
@time Ubig,big,Vbig=svd(Abig);
```

```
@time svd(Bbig);
```

```
@time LAPACK.gesvd!('A', 'A', copy(Abig));
```

```
@time LAPACK.gesdd!('A', copy(Abig));
```

```
@time LAPACK.gesdd!('A', copy(Bbig));
```

```
1.306668 seconds (30 allocations: 53.529 MiB, 0.25% gc time)
```

```
8.383996 seconds (24 allocations: 213.868 MiB, 1.96% gc time)
```

```
5.788669 seconds (16 allocations: 23.408 MiB, 1.32% gc time)
```

```
0.969732 seconds (18 allocations: 45.899 MiB, 0.75% gc time)
```

```
7.144835 seconds (18 allocations: 183.350 MiB, 0.28% gc time)
```

```
In [37]: # Residual
```

```
vecnorm(Abig*Vbig-Ubig*diagm(big))
```

```
Out[37]: 1.0088443817107787e-12
```

```
In [ ]:
```