

# L6b Singular Value Decomposition - Jacobi and Lanczos Methods

Ivan Slapničar

May 2, 2018

## 1 Singular Value Decomposition - Jacobi and Lanczos Methods

Since computing the SVD of  $A$  can be seen as computing the EVD of the symmetric matrices  $A^*A$ ,  $AA^*$ , or  $\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}$ , simple modifications of the corresponding EVD algorithms yield version for computing the SVD.

For more details on one-sided Jacobi method, see [Drm14] and the references therein.

## References

[Drm14] Z. Drmač, Computing Eigenvalues and Singular Values to High Relative Accuracy, in L. Hogben, ed., 'Handbook of Linear Algebra', pp. 59.1-59.21, CRC Press, Boca Raton, 2014.

### 1.1 Prerequisites

The reader should be familiar with concepts of singular values and vectors, related perturbation theory, and algorithms, and Jacobi and Lanczos methods for the symmetric eigenvalue decomposition.

### 1.2 Competences

The reader should be able to recognise matrices which warrant high relative accuracy and to apply Jacobi method to them. The reader should be able to recognise matrices to which Lanczos method can be efficiently applied and do so.

### 1.3 One-sided Jacobi method

Let  $A \in \mathbb{R}^{m \times n}$  with  $\text{rank}(A) = n$  (therefore,  $m \geq n$ ) and  $A = U\Sigma V^T$  its thin SVD.

#### 1.3.1 Definition

Let  $A = BD$ , where  $D = \text{diag}(\|A_{:,1}\|_2, \dots, \|A_{:,n}\|_2)$  is a **diagonal scaling**, and  $B$  is the **scaled matrix** of  $A$  from the right. Then  $[B^T B]_{i,i} = 1$ .

### 1.3.2 Facts

1. Let  $\tilde{U}$ ,  $\tilde{V}$  and  $\tilde{\Sigma}$  be the approximations of  $U$ ,  $V$  and  $\Sigma$ , respectively, computed by a backward stable algorithm as  $A + \Delta A = \tilde{U}\tilde{\Sigma}\tilde{V}^T$ . Since the orthogonality of  $\tilde{U}$  and  $\tilde{V}$  cannot be guaranteed, this product in general does not represent an SVD. There exist nearby orthogonal matrices  $\hat{U}$  and  $\hat{V}$  such that  $(I + E_1)(A + \Delta A)(I + E_2) = \hat{U}\hat{\Sigma}\hat{V}^T$ , where departures from orthogonality,  $E_1$  and  $E_2$ , are small in norm.
2. Standard algorithms compute the singular values with backward error  $\|\Delta A\| \leq \phi\epsilon\|A\|_2$ , where  $\epsilon$  is machine precision and  $\phi$  is a slowly growing function of  $n$ . The best error bound for the singular values is  $|\sigma_j - \tilde{\sigma}_j| \leq \|\Delta A\|_2$ , and the best relative error bound is

$$\max_j \frac{|\sigma_j - \tilde{\sigma}_j|}{\sigma_j} \leq \frac{\|\Delta A\|_2}{\sigma_j} \leq \phi\epsilon\kappa_2(A).$$

3. Let  $\|[\Delta A]_{:,j}\|_2 \leq \epsilon\|A_{:,j}\|_2$  for all  $j$ . Then  $A + \Delta A = (B + \Delta B)D$  and  $\|\Delta B\|_F \leq \sqrt{n}\epsilon$ , and

$$\max_j \frac{|\sigma_j - \tilde{\sigma}_j|}{\sigma_j} \leq \|(\Delta B)B^\dagger\|_2 \leq \sqrt{n}\epsilon\|B^\dagger\|_2.$$

This is Fact 3 from the [Relative perturbation theory](#).

4. It holds

$$\|B^\dagger\| \leq \kappa_2(B) \leq \sqrt{n} \min_{S=\text{diag}} \kappa_2(A \cdot S) \leq \sqrt{n} \kappa_2(A).$$

Therefore, numerical algorithm with column-wise small backward error computes singular values more accurately than an algorithm with small norm-wise backward error.

5. In each step, one-sided Jacobi method computes the Jacobi rotation matrix from the pivot submatrix of the current Gram matrix  $A^T A$ . Afterwards,  $A$  is multiplied with the computed rotation matrix from the right (only two columns are affected). Convergence of the Jacobi method for the symmetric matrix  $A^T A$  to a diagonal matrix, implies that the matrix  $A$  converges to the matrix  $AV$  with orthogonal columns and  $V^T V = I$ . Then  $AV = U\Sigma$ ,  $\Sigma = \text{diag}(\|A_{:,1}\|_2, \dots, \|A_{:,n}\|_2)$ ,  $U = AV\Sigma^{-1}$ , and  $A = U\Sigma V^T$  is the SVD of  $A$ .
6. One-sided Jacobi method computes the SVD with error bound from Facts 2 and 3, provided that the condition of the intermittent scaled matrices does not grow much. There is overwhelming numerical evidence for this. Alternatively, if  $A$  is square, the one-sided Jacobi method can be applied to the transposed matrix  $A^T = DB^T$  and the same error bounds apply, but the condition of the scaled matrix (*this time from the left*) does not change. This approach is slower.
7. One-sided Jacobi method can be preconditioned by applying one QR factorization with full pivoting and one QR factorization without pivoting to  $A$ , to obtain faster convergence, without sacrificing accuracy. This method is implemented in the LAPACK routine [DGESVJ](#). *Writing the wrapper for DGESVJ is a tutorial assignment.*

### 1.3.3 Example - Standard matrix

```
In [1]: function myJacobiR(A1::Array)
    A=deepcopy(A1)
    m,n=size(A)
    T=typeof(A[1,1])
    V=eye(T,n,n)
    # Tolerance for rotation
    tol=sqrt(map(T,n))*eps(T)
    # Counters
    p=n*(n-1)/2
    sweep=0
    pcurrent=0
    # First criterion is for standard accuracy, second one is for relative accuracy
    # while sweep<30 && vecnorm(A-diag(diag(A)))>tol
    while sweep<30 && pcurrent<p
        sweep+=1
        # Row-cyclic strategy
        for i = 1 : n-1
            for j = i+1 : n
                # Compute the 2 x 2 submatrix of A'*A
                F=A[:,[i,j]]'*A[:,[i,j]]
                # Check the tolerance - the first criterion is standard,
                # the second one is for relative accuracy
                # if A[i,j]!=zero(T)
                #
                if abs(F[1,2])>tol*sqrt(F[1,1]*F[2,2])
                    # Compute c and s
                     $\tau = (F[1,1] - F[2,2]) / (2 * F[1,2])$ 
                    t=sign( $\tau$ )/(abs( $\tau$ )+sqrt(1+ $\tau^2$ ))
                    c=1/sqrt(1+t^2)
                    s=c*t
                    G=LinAlg.Givens(i,j,c,s)
                    # A*=G'
                    # In-place multiplication
                    A_mul_Bc!(A,G)
                    # V*=G'
                    A_mul_Bc!(V,G)
                    pcurrent=0
                else
                    pcurrent+=1
                end
            end
        end
    end
     $\sigma = [\text{vecnorm}(A[:,k]) \text{ for } k=1:n]$ 
    for k=1:n
        A[:,k] ./= $\sigma[k]$ 
    end
end
```

```

        end
    A,  $\sigma$ , V
end

```

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

```

In [2]: m=8
        n=5
        s=srand(432)
        A=map(Float64,rand(-9:9,m,n))

```

```

Out[2]: 8×5 Array{Float64,2}:
  5.0  3.0  7.0 -4.0  7.0
  9.0 -4.0  5.0 -7.0 -4.0
  2.0  8.0  6.0 -2.0  1.0
 -3.0  8.0  6.0  8.0  0.0
  3.0 -8.0  5.0 -9.0  7.0
 -5.0 -4.0  1.0  6.0 -5.0
  3.0  0.0 -3.0  1.0  2.0
 -1.0 -9.0  6.0 -1.0  4.0

```

```
In [3]: U, $\sigma$ ,V=myJacobiR(A)
```

```
Out[3]: ([0.268498 -0.344584 ... -0.175395 0.129451; 0.21878 -0.449702 ... 0.790299 -0.309888; .
```

```

In [4]: # Residual
        A*V-U*diagm( $\sigma$ )

```

```

Out[4]: 8×5 Array{Float64,2}:
 2.22045e-16  1.77636e-15  1.77636e-15  2.88658e-15  1.11022e-15
-2.55351e-15  0.0        -2.22045e-16 -8.88178e-16  2.22045e-15
 0.0         1.88738e-15 -3.55271e-15  4.44089e-16  4.44089e-16
 3.33067e-15  4.44089e-15 -1.77636e-15  6.66134e-16 -1.77636e-15
-2.22045e-15  3.55271e-15 -4.44089e-16  3.9968e-15  0.0
 3.46251e-15 -8.88178e-16  2.66454e-15 -1.77636e-15 -1.77636e-15
 4.44089e-16  2.22045e-16 -1.11022e-16  8.88178e-16  0.0
 6.66134e-16 -8.88178e-16  0.0         1.83187e-15 -1.77636e-15

```

### 1.3.4 Example - Strongly scaled matrix

```

In [5]: m=20
        n=15
        B=rand(m,n)
        D=exp.(50*(rand(n)-0.5))
        A=B*diagm(D)

```

```

Out[5]: 20×15 Array{Float64,2}:
 0.114904  1.06011e-11  92.0619  ...  1.21854e-11  1.28563e9  0.0517864
 0.145037  7.74744e-12  199.615  ...  1.07725e-12  1.27903e9  0.0344775

```

```

0.145173      8.04352e-12  250.722      2.47004e-11  1.01152e9  0.128363
0.127954      1.08418e-11  148.993      1.98467e-11  8.2121e8   0.149898
0.0979719     7.87014e-12  196.877      3.90962e-11  1.07317e9  0.153036
0.0346202     1.36199e-11  88.2634     ...  4.28074e-11  3.39398e8   0.127692
0.140697      1.47439e-11  222.475      1.95024e-11  1.98827e9  0.0523626
0.114083      3.133e-12    91.2772      1.3705e-11   4.88712e9  0.152737
0.00877594    7.73339e-12  113.058      1.72984e-11  3.54388e9  0.148548
0.0175307     1.8273e-12   19.4549      3.34049e-11  1.26015e9  0.00107817
0.114658      1.70102e-11  152.776     ...  2.76547e-12  2.28663e9   0.175011
0.165126      1.51486e-11  188.014      1.41324e-11  3.53435e8  0.149874
0.096255      1.56129e-11  252.27       2.22214e-11  5.25828e9  0.116115
0.122679      9.60674e-12  155.68       1.37609e-11  3.36784e9  0.092775
0.104545      1.0295e-11   208.408      2.74716e-11  2.91041e8  0.102976
0.0867603     5.60991e-12  20.2328     ...  4.05889e-11  1.52022e9   0.0745066
0.0652479     6.03934e-12  19.2932      3.96623e-11  3.35883e9  0.0929536
0.0767892     3.58796e-12  97.1746      1.89704e-11  9.56002e8  0.072967
0.0209979     1.52085e-11  206.467      2.34509e-11  5.93668e9  0.0254649
0.0416082     9.10183e-13  81.8504      1.33351e-13  4.94899e9  0.0836844

```

```
In [6]: cond(B), cond(A)
```

```
Out[6]: (30.21929975818701, 1.9307387276084254e21)
```

```
In [7]: U, $\sigma$ ,V=myJacobiR(A);
```

```
In [8]: [sort( $\sigma$ ,rev=True) svdvals(A)]
```

```
Out[8]: 15×2 Array{Float64,2}:
 8.66752e10  8.66752e10
 1.04921e10  1.04921e10
 2.43562e9   2.43562e9
 1.5945e8    1.5945e8
325.304     325.304
 4.41691     4.41691
 0.212232    0.212232
 0.13615     0.13615
 0.000636519 0.00063652
 4.42412e-6  4.42545e-6
 2.24838e-6  2.74444e-6
 3.36285e-7  2.17949e-6
 9.4011e-9   3.36274e-7
 4.49969e-11 9.08092e-9
 8.0436e-12  4.48923e-11

```

```
In [9]: (sort( $\sigma$ ,rev=True)-svdvals(A))./(sort( $\sigma$ ,rev=True))
```

```
Out[9]: 15-element Array{Float64,1}:
 3.52091e-16
 3.63579e-16

```

```

5.8733e-16
1.86907e-16
8.95676e-11
-2.38847e-8
2.26975e-9
1.38959e-9
-7.52147e-7
-0.000300544
-0.220627
-5.48108
-34.7697
-200.812
-4.58111

```

```
In [10]: vecnorm(A*V-U*diag( $\sigma$ ))
```

```
Out[10]: 2.058172581009814e-5
```

```
In [11]: U'*A*V
```

```
Out[11]: 15×15 Array{Float64,2}:
 0.13615      -1.01529e-27   -1.40204e-14   ...   -1.54927e-6    2.31791e-17
-1.23337e-17   8.0436e-12    -2.55462e-14   ...   -4.44557e-8    7.69411e-17
-2.01654e-18   8.71967e-27   325.304        ...   -4.12943e-8    -1.38778e-17
 3.91245e-17   8.50136e-27   8.06744e-14   ...   -8.98037e-7    -2.77556e-17
-7.07781e-18  -1.15971e-27  -1.25372e-14   ...   -1.7425e-7     5.55112e-17
 5.13191e-17  -9.71558e-28   1.12367e-15   ...   4.12043e-7     9.02056e-17
 1.9631e-17   -4.61632e-27  -6.10407e-14   ...   6.76888e-7     1.2523e-17
 3.85089e-18   1.85254e-27  -1.47988e-14   ...   -1.0069e-6     -3.77978e-17
 2.59806e-17   4.97311e-28  -4.09113e-14   ...   -3.36875e-7    3.84483e-17
 2.2014e-18    3.21257e-27   4.81899e-14   ...   -2.80016e-8    -2.07353e-17
 6.10303e-17   5.87522e-27   8.76331e-14   ...   1.04923e-6     2.77556e-16
 1.47817e-17   6.85588e-27   1.4021e-14     ...   -1.66476e-7    -1.14734e-17
 2.23966e-17   3.69491e-27  -6.27239e-14   ...   -8.55755e-7    -8.77851e-19
-7.88884e-18  -2.60429e-27   8.12154e-14     ...   1.04921e10     8.58447e-17
-2.84073e-17  -6.62795e-28  -8.18586e-15    ...   -1.12441e-6     0.212232

```

In the alternative approach, we first apply QR factorization with column pivoting to obtain the square matrix.

```
In [12]: Q,R,p=qr(A, Val{true},thin=true)
```

```
Out[12]: ([-0.264274 -0.0702712 ... 0.190367 -0.418178; -0.362727 -0.143041 ... 0.255463 0.16941
```

```
In [13]: diag(R)
```

```
Out[13]: 15-element Array{Float64,1}:
 -8.62587e10
 1.04214e10

```

```

2.46348e9
-1.59482e8
325.303
4.41519
0.208183
-0.138853
0.00063652
3.8571e-6
-2.5744e-6
3.36873e-7
-9.40114e-9
-4.49919e-11
8.0445e-12

```

```
In [14]: UR,σR,VR=myJacobiR(R')
```

```
Out[14]: ([-0.995129 -0.0947108 ... -4.62875e-22 4.64305e-24; -0.089472 0.9842 ... -4.09528e-21
```

```
In [15]: (sort(σ)-sort(σR))./sort(σ)
```

```
Out[15]: 15-element Array{Float64,1}:
```

```

1.80768e-15
8.61706e-16
5.27926e-16
1.57425e-15
1.31855e-15
1.53166e-15
1.533e-15
4.0772e-16
1.30779e-16
6.03257e-16
0.0
1.86907e-16
1.95777e-16
0.0
3.52091e-16

```

```
In [16]: P=eye(15)
```

```
P=P[:,p];
```

Now  $QRP^T = A$  and  $R^T = U_R \Sigma_R V_R^T$ , so

$$A = (QV_R)\Sigma_R(U_R^T P^T)$$

is an SVD of  $A$ .

```
In [17]: # Check the residual
```

```
U1=Q*VR
```

```
V1=UR[invperm(p),:]
```

```
vecnorm(A*V1-U1*diagm(σR))
```

```
Out[17]: 4.3119609214613406e-5
```

## 1.4 Lanczos method

The function `svds()` is based on the Lanczos method for symmetric matrices. Input can be matrix, but also an operator which defines the product of the given matrix with a vector.

```
In [18]: ?svds;
```

```
search: svds svdvals svdvals! svd svdfact svdfact! isvalid
```

```
In [19]: m=20
         n=15
         A=rand(m,n);
```

```
In [20]: U, $\sigma$ ,V=svd(A);
```

```
In [21]: # All singular values
          $\sigma$ L,rest=svds(A,nsv=15);
```

```
In [22]: typeof( $\sigma$ L)
```

```
Out [22]: Base.LinAlg.SVD{Float64,Float64,Array{Float64,2}}
```

```
In [23]:  $\sigma$ L[:U]
```

```
Out [23]: 20×15 Array{Float64,2}:
 0.176248  0.132782  0.0397018 ... 0.471799  0.0417031 -0.0237426
 0.21288  -0.419569 -0.113033  -0.12853  0.0292693  0.217011
 0.251677 -0.178385 -0.0869505 -0.0270492  0.101613  0.106015
 0.237157  0.116688 -0.139547  -0.105093  0.0222566 -0.0303586
 0.218226  0.193489  0.0954024 -0.34615 -0.225351  0.150867
 0.23564  0.167165 -0.261244 ... -0.224525 -0.214341  0.225831
 0.154493 -0.263709 -0.0405919  0.178305 -0.415037  0.213407
 0.264305 -0.208731  0.101172  -0.114691 -0.219622 -0.36142
 0.187772  0.386485  0.247946  0.0986992  0.299929  0.16475
 0.196871 -0.0577339 -0.345281  0.599963  0.166903 -0.0740634
 0.243507  0.0564267  0.549 ... 0.0547743 -0.126885  0.155771
 0.199568  0.121405 -0.150761  -0.038897  0.214828  0.198936
 0.258398  0.125219 -0.16626  -0.0463476  0.111254  0.299045
 0.19667  0.0508737 -0.233125  0.132561 -0.249989 -0.123458
 0.260888 -0.225213  0.283597  0.0900903  0.0272815 -0.211223
 0.214104 -0.157597  0.355541 ... 0.086989  0.0224978 -0.141842
 0.239158  0.477794  0.029849  0.0947451 -0.265977 -0.164247
 0.242139  0.0821147 -0.237784 -0.287902  0.153583 -0.611808
 0.16799  -0.0677703  0.113081 -0.170264  0.561068  0.00274507
 0.265744 -0.279866 -0.0742213 -0.058851  0.0636666  0.151952
```

```
In [24]:  $\sigma$ L[:S]
```



```
Out[24]: 15-element Array{Float64,1}:
 8.31279
 1.97266
 1.80958
 1.75437
 1.58679
 1.47587
 1.36461
 1.25851
 1.1822
 1.11861
 0.804881
 0.601466
 0.423178
 0.368888
 0.234277
```

```
In [25]: ( $\sigma$ - $\sigma$ L[:S])./ $\sigma$ 
```

```
Out[25]: 15-element Array{Float64,1}:
 4.27379e-16
-1.12561e-16
 1.22705e-16
-7.59398e-16
-1.11947e-15
-1.05315e-15
 3.25434e-16
-1.41148e-15
-5.63468e-16
-5.95499e-16
 6.89681e-16
-7.38344e-16
-3.9353e-16
-9.02893e-16
-1.18473e-15
```

```
In [26]: # Some largest singular values
 $\sigma$ p,rest=svds(A,nsv=5);
( $\sigma$ [1:5]- $\sigma$ p[:S])./ $\sigma$ [1:5]
```

```
Out[26]: 5-element Array{Float64,1}:
 0.0
-3.26427e-15
-1.47246e-15
-1.39223e-15
 1.39934e-16
```

### 1.4.1 Example - Large matrix

```
In [27]: m=2000
```

```
         n=1500
```

```
         Ab=rand(m,n);
```

```
In [28]: @time Ub,σb,Vb=svd(Ab);
```

```
4.460463 seconds (109 allocations: 131.797 MiB, 2.18% gc time)
```

```
In [29]: # This is rather slow
```

```
         @time σl,rest=svds(Ab,nsv=10);
```

```
2.783898 seconds (6.42 k allocations: 2.777 MiB)
```

```
In [30]: (σb[1:10]-σl[:S])./σb[1:10]
```

```
Out[30]: 10-element Array{Float64,1}:
```

```
 -3.93633e-15
```

```
 -3.70705e-15
```

```
 -2.68072e-15
```

```
  1.34316e-15
```

```
 -3.29052e-15
```

```
 -6.89541e-15
```

```
 -1.20373e-15
```

```
 -9.50495e-15
```

```
 -6.95526e-15
```

```
 -2.42184e-15
```

### 1.4.2 Example - Very large sparse matrix

```
In [31]: ?sprand
```

```
search: sprand sprandn StepRange StepRangeLen
```

```
Out[31]:
```

```
sprand([rng],[type],m,[n],p::AbstractFloat,[rfn])
```

Create a random length  $m$  sparse vector or  $m$  by  $n$  sparse matrix, in which the probability of any element being nonzero is independently given by  $p$  (and hence the mean density of nonzeros is also exactly  $p$ ). Nonzero values are sampled from the distribution specified by  $rfn$  and have the type  $type$ . The uniform distribution is used in case  $rfn$  is not specified. The optional  $rng$  argument specifies a random number generator, see [Random Numbers](#).

## 2 Example

```
julia> rng = MersenneTwister(1234);
```

```
julia> sprand(rng, Bool, 2, 2, 0.5)
```

```
2×2 SparseMatrixCSC{Bool,Int64} with 2 stored entries:
```

```
 [1, 1] = true
```

```
 [2, 1] = true
```

```
julia> sprand(rng, Float64, 3, 0.75)
```

```
3-element SparseVector{Float64,Int64} with 1 stored entry:
```

```
 [3] = 0.298614
```

```
In [32]: m=10000
```

```
        n=3000
```

```
        A=sprand(m,n,0.05)
```

```
Out[32]: 10000×3000 SparseMatrixCSC{Float64,Int64} with 1500126 stored entries:
```

```
 [22 , 1] = 0.473301
```

```
 [24 , 1] = 0.744926
```

```
 [28 , 1] = 0.95938
```

```
 [37 , 1] = 0.748794
```

```
 [65 , 1] = 0.403014
```

```
 [73 , 1] = 0.794408
```

```
 [82 , 1] = 0.213179
```

```
 [131 , 1] = 0.966553
```

```
 [138 , 1] = 0.811639
```

```
 [149 , 1] = 0.105898
```

```
 ⋮
```

```
 [9671 , 3000] = 0.0225395
```

```
 [9706 , 3000] = 0.0505621
```

```
 [9748 , 3000] = 0.245875
```

```
 [9761 , 3000] = 0.170929
```

```
 [9816 , 3000] = 0.880533
```

```
 [9827 , 3000] = 0.867853
```

```
 [9831 , 3000] = 0.315429
```

```
 [9907 , 3000] = 0.259937
```

```
 [9918 , 3000] = 0.446506
```

```
 [9949 , 3000] = 0.687168
```

```
 [9954 , 3000] = 0.486146
```

```
In [34]: # No vectors, this takes about 30 sec.
```

```
        @time  $\sigma_1$ ,rest=svds(A,nsv=100,ritzvec=false)
```

```
36.528656 seconds (17.34 k allocations: 42.033 MiB, 0.02% gc time)
```

```
Out[34]: (Base.LinAlg.SVD{Float64,Float64,Array{Float64,2}}(Array{Float64}(3000,0), [137.677, 19
```

```
In [35]: @time  $\sigma_2$ =svdvals(full(A));
```

```
23.916113 seconds (4.38 k allocations: 459.733 MiB, 0.46% gc time)
```

```
In [38]: maximum(abs, ( $\sigma_1$ [:S]- $\sigma_2$ [1:100])./ $\sigma_2$ [1:100]))
```

```
Out[38]: 6.637629466276756e-15
```

```
In [ ]:
```