



15,840,620 members

Sign in

[home](#) [articles](#) [quick answers](#) [discussions](#) [features](#) [community](#) [help](#)

Search for articles, questions, tips

[Articles / General Programming / Algorithms](#)[C++](#) [algorithm](#) [text](#) [C++11](#)

Singular Values Decomposition (SVD) in C++11 by an Example



Arthur V. Ratz

Rate me: ★★★★★ 5.00/5 (15 votes)

30 Dec 2018

CPOL

22 min read

👁 49.9K

📄 2.1K

🔖 14

💬 14

SVD in C++11 explained with an example

In this article, we will demonstrate how to compute full SVD of a given matrix A and discuss about the code in C++11 implementing the full SVD computation by using simple iteration and Jordan-Gaussian methods.

[Download SVD_MP-EXE - 176.3 KB](#)[Download SVD-MP - 6.9 KB](#)[Download SVD-EXE - 78.1 KB](#)[Download SVD - 6.7 KB](#)

Introduction

Singular value decomposition (Singular Value Decomposition, SVD) is the decomposition of a real matrix in order to bring it to a canonical form. Singular decomposition is a convenient method when working with matrices. It shows the geometric structure of the matrix and allows you to visualize the available data. Singular decomposition is used in solving various problems - from approximation by the method of least squares and solving systems of equations to image compression. At the same time, different properties of singular decomposition are used, for example, the ability to show the rank of a matrix, to approximate matrices of a given rank. SVD allows you to calculate inverse and pseudoinverse matrices of large size, which makes it a useful tool for solving regression analysis problems.

Singular value decomposition was originally invented and proposed by mathematicians to determine whether there's another natural bilinear form of a given matrix obtained as the result of performing various independent orthogonal transformations of two spaces.

In 1873 and 1874, Eugenio Beltrami and Camille Jordan revealed that the singular values of the bilinear forms, represented as a matrix, produce a complete set of invariants for bilinear forms while performing orthogonal substitutions. James Joseph Sylvester also invented and proposed the singular-value decomposition for real square matrices in 1889.

The fourth mathematician who founded the singular value decomposition independently is Autonne in 1915, who was able to compute SVD via the polar decomposition. The first proof of the singular value decomposition for rectangular and complex matrices was made by Carl Eckart and Gale Young in 1936. They observed it as a generalization of the principal axis transformation for Hermitian matrices.

In 1907, Erhard Schmidt defined an analog of singular values for integral operators (which are compact, under some weak technical assumptions); it seems he was not familiar with the parallel work on singular values of finite matrices. This theory was further proposed by Émile Picard in 1910, who was the first mathematician who called the computed numerical values as "singular values".

Practical methods for computing the SVD date back to Kogbetliantz in 1954, 1955 and Hestenes in 1958, resembling closely the Jacobi eigenvalue algorithm, which uses plane rotations or Givens rotations. However, these were replaced by the method of Gene Golub and William Kahan published in 1965, which uses Householder transformations or reflections. In 1970, Jacques Golub and Christian Reinsch published a variant of the Golub/Kahan algorithm that is still the one most-used today.

In this article, we will discuss about method of simple iterations and Jordan-Gaussian transformation used to compute the full SVD of a given real or unitary matrix. Also, we will demonstrate the code in C++11 implementing the SVD computational algorithm thoroughly discussed. Specifically, we will provide a step-by-step tutorial for computing the full SVD, based on the example of finding singular values decomposition for a given integral matrix.

Background

Computing Singular Value Decomposition (SVD)

Singular values decomposition (SVD) of matrix A is an algorithm that allows us to find a decomposition of a given real or complex matrix A into a set of singular values, as well as its left and right singular vectors. Algebraically, singular value decomposition can be formulated as:

$$A = U * S * V^T$$

where A - is a given real or unitary matrix, U - an orthogonal matrix of left singular vectors, S - is a symmetric diagonal matrix of singular values, V^T - is a transpose orthogonal matrix of right singular vectors, respectively.

As we can see from formula (1) above, a decomposition of given matrix (A) is a product of a certain orthogonal matrix of left singular vectors U , symmetric diagonal singular values matrix S and transpose orthogonal matrix of right singular vectors V^T :

$$A = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} \\ u_{2,1} & u_{2,2} & u_{2,3} \\ u_{3,1} & u_{3,2} & u_{3,3} \end{bmatrix} * \begin{bmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & s_3 \end{bmatrix} * \begin{bmatrix} v_{1,1} & v_{2,1} & v_{3,1} \\ v_{1,2} & v_{2,2} & v_{3,2} \\ v_{1,3} & v_{2,3} & v_{3,3} \end{bmatrix}$$

According to the formula listed above, each singular value of $\forall s_i (s_1 \geq s_2 \geq s_3 \geq \dots \geq s_{\min(m,n)})$ exactly corresponds to a pair of singular vectors of either $U_i = \{u_{1,1} \ u_{1,2} \ u_{1,3} \dots u_{1,(n-1)} \ u_{1,n}\}$ or $V_i = \{v_{1,1} \ v_{2,1} \ v_{3,1} \dots v_{(m-1),1} \ v_{m,1}\}$. Formally, the SVD decomposition of a given matrix A must satisfy the following criteria so that:

$$A * \bar{v}_i = s_i * \bar{u}_i, \quad A^T * \bar{u}_i = s_i * \bar{v}_i$$

where A - is a given matrix, U_i and V_i - left and right orthogonal singular vectors, s_i - a singular value. By applying the second formula to the first one listed above, we can prove that i - th vectors U_i and V_i are actually the eigenvectors and the i - th value of s_i - the eigenvalue, exactly corresponding to both of these vectors.

Step 1: Computing Symmetric Factorization Matrix $A^T A$

To compute the full SVD of matrix A , we must first compute the eigenvalues and eigenvectors of a certain symmetric real or unitary matrix, obtained as a product of matrices A^T and A . The product of these two matrices actually gives us a symmetric matrix, for which the eigenvalues σ are easily computed.

For example, suppose we're given a matrix A :

$$A = \begin{bmatrix} 3 & 1 & 0 \\ 1 & 2 & 2 \\ 0 & 1 & 1 \end{bmatrix}$$

Let's find a symmetric factorization matrix $A^T A$:

$$A^T A = \begin{vmatrix} 3 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 2 & 1 \end{vmatrix} * \begin{vmatrix} 3 & 1 & 0 \\ 1 & 2 & 2 \\ 0 & 1 & 1 \end{vmatrix} = \begin{vmatrix} 10 & 5 & 2 \\ 5 & 6 & 5 \\ 2 & 5 & 5 \end{vmatrix},$$

After we've successfully computed $A^T A$, let's find the eigenvalues for this symmetric factorization matrix. Obviously, that, the factorization matrix will have the following eigenvalues: $\sigma_1 = 15.4310$, $\sigma_2 = 5.5573$, $\sigma_3 = 0.0116$.

Step 2: Computing Singular Values Of Matrix $A^T A$

Since a set of eigenvalues for the factorization matrix $A^T A$ is already computed, let's now find a diagonal matrix of singular values S . To do this, we must compute a square-root of each of these eigenvalues and place them along the diagonal of matrix S :

$$S = \begin{vmatrix} \sqrt{\sigma_1} & 0 & 0 \\ 0 & \sqrt{\sigma_2} & 0 \\ 0 & 0 & \sqrt{\sigma_3} \end{vmatrix}$$

After that, we also must compute an inverse matrix S^{-1} . As we've already found all singular values must be arranged onto the diagonal of matrix S in descending order, starting at the first maximum singular value s_{max} .

To find the inverse diagonal matrix S^{-1} , all we have to do is to divide value of 1.0 by each of the singular values in the diagonal of matrix S :

$$S^{-1} = \begin{vmatrix} \frac{1}{s_1} & 0 & 0 \\ 0 & \frac{1}{s_2} & 0 \\ 0 & 0 & \frac{1}{s_3} \end{vmatrix}$$

For example, suppose we already have those eigenvalues computed during step 1. Let's now compute the matrix of singular values S and its inverse S^{-1} :

$$S = \begin{vmatrix} \sqrt{\sigma_1} & 0 & 0 \\ 0 & \sqrt{\sigma_2} & 0 \\ 0 & 0 & \sqrt{\sigma_3} \end{vmatrix} = \begin{vmatrix} \sqrt{15.4310} & 0 & 0 \\ 0 & \sqrt{5.5573} & 0 \\ 0 & 0 & \sqrt{0.0116} \end{vmatrix} = \begin{vmatrix} 3.9282 & 0 & 0 \\ 0 & 2.3573 & 0 \\ 0 & 0 & 0.1079 \end{vmatrix}$$

$$S^{-1} = \begin{vmatrix} \frac{1}{3.9282} & 0 & 0 \\ 0 & \frac{1}{2.3573} & 0 \\ 0 & 0 & \frac{1}{0.1079} \end{vmatrix} = \begin{vmatrix} 0.2545 & 0 & 0 \\ 0 & 0.4241 & 0 \\ 0 & 0 & 9.2604 \end{vmatrix}$$

Step 3: Computing Right Singular Vectors of Matrix $A^T A$

Since we've already computed the diagonal matrix of singular values and found its inverse, now let's obtain the right eigenvectors of $A^T A$ factorization matrix and find the right singular vectors V of matrix A , dividing each component of those right eigenvectors by each vector's absolute scalar length L :

$$L = \sqrt{\sum_{i=1}^m v_i^2}, \bar{V}_i = \left\{ \frac{v_1}{L} \quad \frac{v_2}{L} \quad \frac{v_3}{L} \quad \dots \quad \frac{v_{m-1}}{L} \quad \frac{v_m}{L} \right\}$$

Additionally, to compute the right singular vectors of matrix A , we must also find a transpose of the obtained matrix V right after performing those computations.

For example, suppose we've already computed right eigenvectors of the factorization matrix $A^T A$ by using formula $(A^T A - \sigma I) = 0$:

$$\sigma_1 = 15.4310, \quad A^T A = \begin{vmatrix} 10 & 5 & 2 \\ 5 & 6 & 5 \\ 2 & 5 & 5 \end{vmatrix},$$

Article

[View Code](#)

[Stats](#)

[Revisions \(100\)](#)

$$\begin{aligned}
 A^T A - \sigma_1 I &= \begin{vmatrix} 10 - 15.4310 & 5 & 2 \\ 5 & 6 - 15.4310 & 5 \\ 2 & 5 & 5 - 15.4310 \end{vmatrix} = \\
 &= \begin{vmatrix} -5.4310 & 5 & 2 \\ 5 & -9.4310 & 5 \\ 2 & 5 & -10.4310 \end{vmatrix} = 0 \\
 \sigma_2 &= 5.5573, \quad A^T A = \begin{vmatrix} 10 & 5 & 2 \\ 5 & 6 & 5 \\ 2 & 5 & 5 \end{vmatrix}, \\
 A^T A - \sigma_2 I &= \begin{vmatrix} 10 - 5.5573 & 5 & 2 \\ 5 & 6 - 5.5573 & 5 \\ 2 & 5 & 5 - 5.5573 \end{vmatrix} = \\
 &= \begin{vmatrix} 4.4427 & 5 & 2 \\ 5 & 1.5573 & 5 \\ 2 & 5 & -0.5573 \end{vmatrix} = 0 \\
 \sigma_3 &= 0.0116, \quad A^T A = \begin{vmatrix} 10 & 5 & 2 \\ 5 & 6 & 5 \\ 2 & 5 & 5 \end{vmatrix}, \\
 A^T A - \sigma_3 I &= \begin{vmatrix} 10 - 0.0116 & 5 & 2 \\ 5 & 6 - 0.0116 & 5 \\ 2 & 5 & 5 - 0.0116 \end{vmatrix} = \\
 &= \begin{vmatrix} 9.9884 & 5 & 2 \\ 5 & 5.9884 & 5 \\ 2 & 5 & 4.9884 \end{vmatrix} = 0
 \end{aligned}$$

Each of these three right eigenvectors V can be obtained by finding a non-trivial solution for the given system of linear equations, solving it by using Jordan-Gaussian method that allows us to transform the given matrices into a reduced row echelon form, discussed, below, in the next paragraph of this article.

Apparently, the factorization matrix $A^T A$ will have the following right eigenvectors V :

$$\begin{aligned}
 \sigma_1 &= 15.4310, \quad V_1 = \{1.6728, 1.4170, 1.0000\}, \\
 \sigma_2 &= 5.5573, \quad V_2 = \{-1.0469, 0.5302, 1.0000\}, \\
 \sigma_3 &= 0.0116, \quad V_3 = \{0.3740, -1.1473, 1.0000\}
 \end{aligned}$$

Now let's find an absolute scalar length of each of these vectors:

$$V_1 = \{1.6728, 1.4170, 1.0000\}, \quad L_1 = \sqrt{1.6728^2 + 1.4170^2 + 1.0000^2} = \sqrt{5.8065} = 2.4096$$

$$V_1 = \left\{ \frac{1.6728}{2.4096}, \frac{1.4170}{2.4096}, \frac{1.0000}{2.4096} \right\} = \{0.6942, 0.5880, 0.4149\}$$

$$V_2 = \{-1.0469, 0.5302, 1.0000\}, \quad L_2 = \sqrt{(-1.0469)^2 + 0.5302^2 + 1.0000^2} = \sqrt{2.3772} = 1.5418$$

$$V_2 = \left\{ \frac{-1.0469}{1.5418}, \frac{0.5302}{1.5418}, \frac{1.0000}{1.5418} \right\} = \{-0.6790, 0.3439, 0.6485\}$$

$$V_3 = \{0.3740, -1.1473, 1.0000\}, \quad L_3 = \sqrt{0.3740^2 + (-1.1473)^2 + 1.0000^2} = \sqrt{2.4562} = 1.5672$$

$$V_3 = \left\{ \frac{0.3740}{1.5672}, \frac{-1.1473}{1.5672}, \frac{1.0000}{1.5672} \right\} = \{0.2386, -0.7320, 0.6380\}$$

Finally, let's place these three vectors along rows in matrix V and transpose the resultant matrix:

$$V = \begin{bmatrix} 0.6942 & 0.5880 & 0.4149 \\ -0.6790 & 0.3439 & 0.6485 \\ 0.2386 & -0.7320 & 0.6380 \end{bmatrix}, V^T = \begin{bmatrix} 0.6942 & -0.6790 & 0.2386 \\ 0.5880 & 0.3439 & -0.7320 \\ 0.4149 & 0.6485 & 0.6380 \end{bmatrix}$$

Obviously, that, the transpose matrix V^T – the orthogonal matrix of right singular vectors of matrix A .

Step 4: Computing Left Singular Vectors of Matrix $A^T A$

Finally, since we've already obtained an orthogonal matrix of right singular values, we can easily compute the specific matrix of left singular values U as a product of the given matrix A , transposed matrix of right singular values V^T and the inverse diagonal matrix S^{-1} , previously obtained. This is typically done by using the following formula:

$$U = A * V^T * S^{-1}$$

After applying the specific transformation listed above, each row of matrix U will contain the computed left singular vectors. For example, let's find the orthogonal matrix U of left singular vectors of matrix A by using formula listed above:

$$U = \begin{bmatrix} 3 & 1 & 0 \\ 1 & 2 & 2 \\ 0 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 0.6942 & -0.6790 & 0.2386 \\ 0.5880 & 0.3439 & -0.7320 \\ 0.4149 & 0.6485 & 0.6380 \end{bmatrix} * \begin{bmatrix} 0.2545 & 0 & 0 \\ 0 & 0.4241 & 0 \\ 0 & 0 & 9.2604 \end{bmatrix}$$

$$U = \begin{bmatrix} 0.6798 & -0.7182 & -0.1479 \\ 0.6874 & 0.5539 & 0.4696 \\ 0.2553 & 0.4210 & -0.8703 \end{bmatrix}$$

As the result of performing all those computations, we will obtain the following full SVD decomposition of matrix A :

$$\begin{bmatrix} 3 & 1 & 0 \\ 1 & 2 & 2 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0.6798 & -0.7182 & -0.1479 \\ 0.6874 & 0.5539 & 0.4696 \\ 0.2553 & 0.4210 & -0.8703 \end{bmatrix} * \begin{bmatrix} 3.9282 & 0 & 0 \\ 0 & 2.3573 & 0 \\ 0 & 0 & 0.1079 \end{bmatrix} =$$

$$\begin{bmatrix} 0.6942 & -0.6790 & 0.2386 \\ 0.5880 & 0.3439 & -0.7320 \\ 0.4149 & 0.6485 & 0.6380 \end{bmatrix}$$

In the next paragraph of this article, we will discuss specifically how to compute eigenvalues and eigenvectors of a given symmetric factorization matrix $A^T A$ and provide a detailed example for performing of such computations.

Eigenvalues and Eigenvectors (EVD)

As we've already discussed, the process of full SVD composition for a given matrix A is mainly based on eigenvalues and eigenvectors computation. In fact, to compute those eigenvectors for the given matrix A , we must first compute a set of all eigenvalues for this matrix.

In linear algebra, there's the number of methods that allow us to find the eigenvalues for different matrices of small sizes, such as 2x2, 3x3 and more. For example, to find all eigenvalues of a given matrix, we can compute its determinant. The result of finding the matrix A determinant is an equation of power N , the solutions of which are actually those eigenvalues we need to find. However, the most of existing methods for finding matrix A eigenvalues are not computational and either cannot be formulated as a computer algorithm.

In this paragraph, we will discuss about an approach, called method of simple iterations, that will allow us to find all eigenvalues for a factorization matrix $A^T A$, and, then, find each eigenvector of the following matrix, corresponding exactly to each eigenvalue obtained. Unlike the other existing methods, recalled above, the following method can be easily and conveniently formulated as a computational algorithm.

In general, an idea of using simple iterations method is illustrated in the figure shown below:

Step 1: Computing A Maximum Eigenvalue Of Matrix A^*

The first thing that we need to do is to find the first maximum eigenvalue for the given matrix $A^* = A^T A$. To do this, we must use the method of simple iterations that can be formulated as follows:

1. $A^*_{(mxm)} = A^T A$ – is a factorization matrix and $R = \{1, 1, 1 \dots, 1\}$ – an initial unit vector. Also, let $i = 1, t$ – a loop counter variable and $\varepsilon = 10e - 6$ – a constant value of accuracy error, V^* – vector that holds the results obtained during each iteration, $M_{(mxt)}$ – resultant matrix, the columns of which are values of vector V^* , σ and σ_{old} – eigenvalues for the current and previous iteration, respectively ($\sigma = 0, \sigma_{old} = \sigma$);
2. Find a product of matrix A^* and specific vector R , and obtain vector $V^* = A^* * R$;
3. Append vector V^* to the resultant matrix M by placing its values along its i – th column;
4. Update vector R by assigning the values of resultant vector V^* to vector $R (R = V^*)$;
5. If this is not the first iteration ($i > 0$), compute the value of σ by performing the division of i – th and $(i - 1)$ – th value in the first row of the resultant matrix $M (\sigma = \frac{M_{(1,i)}}{M_{(1,i-1)}})$;
6. If this is not the first iteration ($i > 0$), compute the difference between σ and σ_{old} as $\Delta = \sigma - \sigma_{old}$;
7. If the value of delta is greater than the value of accuracy error $\varepsilon = 10e - 6$ (e.g. $\Delta > \varepsilon$), then return and proceed with step 2, otherwise terminate the computations and go to step 8;
8. At the end of performing the computations listed above, we will obtain the maximum eigenvalue for the given matrix $A^*_{(mxm)} : \sigma_{max} \leftarrow \sigma$;

The example below illustrates the following computations:

Suppose we're given a factorization matrix $A^*_{(mxm)} = A^T A$ and unit vector $R = \{1, 1, 1\}$:

$$A^* = A^T A = \begin{bmatrix} 10 & 5 & 2 \\ 5 & 6 & 5 \\ 2 & 5 & 5 \end{bmatrix}, R = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Let's find a product of the following matrix and vector by performing the following computations:

$$V^* = A^* * R = \begin{bmatrix} 17 \\ 16 \\ 12 \end{bmatrix}$$

Since, we're performing the first iteration and $i = 1$, we perform no other computations, but appending the resultant vector obtained to the matrix M .

After that, we proceed with the next iteration $i = 2$. During this iteration, we're similarly performing the following computation:

$$A^* = A^T A = \begin{bmatrix} 10 & 5 & 2 \\ 5 & 6 & 5 \\ 2 & 5 & 5 \end{bmatrix}, R = V^* = \begin{bmatrix} 17 \\ 16 \\ 12 \end{bmatrix}, V^* = A^* * R = \begin{bmatrix} 274 \\ 241 \\ 174 \end{bmatrix}$$

Similarly, we're appending the intermediate result to matrix M , and, since it's the not first iteration, performing the so far eigenvalue \sigma computation: $\sigma = \frac{M_{(1,i)}}{M_{(1,i-1)}} = \frac{274}{17} = 16.11$. Then, we're computing value of $\Delta = \sigma - \sigma_{old} = 16.11 - 0.00 = 16.11$. Obviously, that, the value of Δ is much greater than the value of accuracy error ε . That's actually why, we're proceeding with the next iteration, during which we will obtain the following result:

$$V^* = A^* * R = \begin{bmatrix} 4293 \\ 3686 \\ 2623 \end{bmatrix}$$

After that, we're computing the value of Δ over again and proceed with the next computation phase until the value of Δ becomes actually smaller than the given accuracy error value ε . At the end of performing those computation listed above, we will obtain the following resultant matrix M and the first maximum eigenvalue σ_{max} of the factorization matrix A^* :

At the end of performing those computations listed above, we will obtain the following resultant matrix M and the first maximum eigenvalue σ_{max} of the factorization matrix A^* :

```
17   274   4293   ..   ..   ..   ..   ..   3.9148848553274702e+28  6.0410605508676628e+29
16   241   3686   ..   ..   ..   ..   ..   3.3162556201511577e+28  5.1173155134159651e+29
12   174   2623   ..   ..   ..   ..   ..   2.3402394273230705e+28  3.6112244948026083e+29
```

In this particular case, to find the first maximum eigenvalue of the given factorization matrix, we're performing $t = 25$ iterations and finally end up with the following result:

$$\sigma_{max} = \frac{6.0410605508676628e + 29}{3.9148848553274702e + 28} = 15.4310 \dots$$

Step 2: Computing An Eigenvector of Matrix A^*

Since we've successfully computed the first maximum eigen value σ_{max} , let's now find a specific eigenvector that corresponds the following eigenvalue. To do this, we must subtract this eigenvalue from each diagonal element of the given matrix $A^* = A^T A$, and then find a non-trivial solution for the system of linear equations by performing Jordan-Gaussian transformation to the row echelon form for the given matrix:

$$A^* = A^T A = \begin{bmatrix} 10 & 5 & 2 \\ 5 & 6 & 5 \\ 2 & 5 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 10 - \sigma_{max} & 5 & 2 \\ 5 & 6 - \sigma_{max} & 5 \\ 2 & 5 & 5 - \sigma_{max} \end{bmatrix} = 0$$

$$\begin{bmatrix} 10 - 15.4310 & 5 & 2 \\ 5 & 6 - 15.4310 & 5 \\ 2 & 5 & 5 - 15.4310 \end{bmatrix} = \begin{bmatrix} -5.4310 & 5 & 2 \\ 5 & -9.4310 & 5 \\ 2 & 5 & -10.4310 \end{bmatrix} = 0$$

$$\begin{aligned} -5.4310x_1 + 5x_2 + 2x_3 &= 0 \\ 5x_1 - 9.4310x_2 + 5x_3 &= 0 \\ 2x_1 + 5x_2 - 10.4310x_3 &= 0 \end{aligned}$$

At this point, let's discuss how to solve the following system of linear equations by using Jordan-Gaussian transformation, which is the computational algorithm that can be formulated as follows:

1. Let $A^*_{(m \times m)} = (A^T A - \sigma I) = 0$ – the system of linear equations we're about to solve and $i = 1, \bar{m} - a$ loop counter variable;

2. Take the i - th row of the given matrix as the leading row and obtain the value of basis element $\alpha = a_{i,i}$;
3. Check if a given basis element α is equal to 0 or 1. If not, proceeding with step 4, otherwise go to step 5;
4. Divide each element in the i - th leading row by the value of basis element α ;
5. Check if the value of α is equal to 0. If so, interchange the i - th and $(i + 1)$ - th rows, otherwise go to the next step;
6. For each row $r = 1, \bar{m}$, other than the leading row i , perform an update by re-computing values in each row. To do this, obtain the value of each row's basis element as $\gamma = a_{r,i}$ and then use the following formula to update these values $k = 1, \bar{m}$: $a_{r,k} = a_{r,k} - a_{i,k} * \gamma$;
7. Perform a check if the next basis element $\alpha_{i,i}$ is equal to 0 or $i = m - 1$ (e.g., this is the last column of matrix A^*). If so, jump to Step 8, otherwise increment the value of $i = i + 1$ and return to Step 2 to proceed with the next iteration.
8. Extract each value from the $i + 1$ - th column of the given matrix and assign it to vector X , which is a non-trivial solution for the given system of linear equations;
9. Negate a value of each component of vector X ;
10. Assign the last component of vector X to the value of 1 to obtain the eigenvector that exactly corresponds to the eigenvalue σ ;

Here's an example for performing such computations:

Suppose we're already given a matrix A^* computed at the beginning of step 2:

$$A^* = \left| \begin{array}{ccc} -5.4310 & 5 & 2 \\ 5 & -9.4310 & 5 \\ 2 & 5 & -10.4310 \end{array} \right| = 0$$

Let's compute a non-trivial solution for this system of linear equations using the algorithm listed above. To do this, we're taking the first row as the leading row and obtain its basis element $\alpha = a_{1,1} = -5.4310$. At this point, all we have to do is to check if this value is not either 0 or 1, and if so, divide each element in the first row by the value of α :

$$A^* = \left| \begin{array}{ccc} \frac{-5.4310}{-5.4310} & \frac{5}{-5.4310} & \frac{2}{-5.4310} \\ 5 & -9.4310 & 5 \\ 2 & 5 & -10.4310 \end{array} \right| = \left| \begin{array}{ccc} 1 & -0.9206 & -0.3682 \\ 5 & -9.4310 & 5 \\ 2 & 5 & -10.4310 \end{array} \right|$$

After that, we must update each row, other than the leading row, by performing the following computations. For each specific row, we must obtain the value of basis element $\gamma = a_{r,i}$. For the second row, this value is equal to $\gamma = 5$. Let's apply the value of γ and each value in the leading first row to the other rows of matrix A^* :

$$A^* = \left| \begin{array}{ccc} 1 & -0.9206 & -0.3682 \\ 5 & -9.4310 & 5 \\ 2 & 5 & -10.4310 \end{array} \right|$$

$$A^* = \left| \begin{array}{ccc} 1 & -0.9206 & -0.3682 \\ 5 - 5 * 1 & -9.4310 - 5 * (-0.9206) & 5 - 5 * (-0.3682) \\ 2 - 2 * 1 & 5 - 2 * (-0.9206) & -10.4310 - 2 * (-0.3682) \end{array} \right|$$

$$A^* = \left| \begin{array}{ccc} 1 & -0.9206 & -0.3682 \\ 0 & -4.8278 & 6.8412 \\ 0 & 6.8412 & -9.6944 \end{array} \right|$$

Since we've obtained values for the first leading row and update values in each row other than the first row, let's proceed the computations with the second leading row. To do this, we need again to obtain the value of basic element $\alpha = a_{2,2} = -4.8278$ and perform similar computation as we've already done above:

$$A^* = \begin{vmatrix} 1 & -0.9206 & -0.3682 \\ 0 & -4.8278 & 6.8412 \\ 0 & 6.8412 & -9.6944 \end{vmatrix} = \begin{vmatrix} 1 & -0.9206 & -0.3682 \\ 0 & \frac{-4.8278}{-4.8278} & \frac{6.8412}{-4.8278} \\ 0 & 6.8412 & -9.6944 \end{vmatrix}$$

$$A^* = \begin{vmatrix} 1 & -0.9206 & -0.3682 \\ 0 & 1 & -1.4170 \\ 0 & 6.8412 & -9.6944 \end{vmatrix}$$

Similarly, to the previous phase, we also need to update values in all other rows. To do this, we must obtain the value of $\gamma = -0.9206$ and perform the following computations:

$$A^* = \begin{vmatrix} 1 & -0.9206 & -0.3682 \\ 0 & 1 & -1.4170 \\ 0 & 6.8412 & -9.6944 \end{vmatrix}$$

$$A^* = \begin{vmatrix} 1 & -0.9206 - (-0.9206) * 1 & -0.3682 - (-0.9206) * (-1.4170) \\ 0 & 1 & -1.4170 \\ 0 & 6.8412 - 6.8412 * 1 & -9.6944 - 6.8412 * (-1.4170) \end{vmatrix}$$

$$A^* = \begin{vmatrix} 1 & 0 & -1.6728 \\ 0 & 1 & -1.4170 \\ 0 & 0 & 0 \end{vmatrix}$$

After we've computed the second leading row and updated the rest of other rows, we might notice that all diagonal elements of matrix A^* are equal to 1, except for the last diagonal element $a_{2,2}$, which is equal to 0. This actually means that all elements of column $i=2$ are the non-trivial solution for this system of equations. At this point, we must extract all elements of the column $i = 2$ and place them into a vector X , negating each element. Also, we must assign the last element in the column (i.e. vector X) to 1:

$$A^* = \begin{vmatrix} 1 & 0 & -1.6728 \\ 0 & 1 & -1.4170 \\ 0 & 0 & 0 \end{vmatrix}, X = \begin{vmatrix} 1.6728 \\ 1.4170 \\ 1 \end{vmatrix}$$

Obviously, that, the following vector X is an eigenvector that exactly corresponds to the given eigenvalue $\sigma = 15.4310$:

$$A^* = A^T A = \begin{vmatrix} 10 & 5 & 2 \\ 5 & 6 & 5 \\ 2 & 5 & 5 \end{vmatrix}, \sigma_1 = 15.4310, X_1 = \begin{vmatrix} 1.6728 \\ 1.4170 \\ 1 \end{vmatrix}$$

Step 3: Finding An Equivalent Matrix B of matrix A^*

Since we've already computed first maximum eigenvalue σ_1 and eigenvector X_1 for the factorization matrix A^* , let's now find the second eigenvalue and eigenvector of the following matrix. This is typically done by computing a matrix B of reduced size, equivalent to matrix A^* , and then perform the same computations as we've already done during step 1 and 2 to find the second pair of eigenvalue and eigenvector of the given matrix A^* . In order to find an equivalent matrix B , we normally use the following formula:

$$H * A^* * H^{-1} = \begin{vmatrix} \sigma_1 & b'_1 \\ 0 & B \end{vmatrix},$$

where H – Hermitian conjugate of matrix A^* , H^{-1} – Hermitian conjugate inverse of matrix A^* , B – equivalent of matrix A^* , for which the second eigenvalue and eigenvector are computed;

To find an equivalent matrix B we must first find Hermitian conjugate matrix H and its inverse H^{-1} . Hermitian conjugate matrix H is a special case of a diagonal identity matrix that can be represented as follows:

$$H = \begin{bmatrix} \frac{1}{x_1}, 0, 0, 0, 0 \\ -\frac{x_2}{x_1}, 1, 0, 0, 0 \\ -\frac{x_3}{x_1}, 0, 1, 0, 0 \\ \dots \\ -\frac{x_m}{x_1}, 0, 0, 0, 1 \end{bmatrix}$$

Each element in first column of Hermitian matrix H is assigned to the fractional value of each component of eigenvector x_i divided by the first component x_1 , except for the first element which is the division of $1/x_1$.

Also, to compute matrix $H_{(mxm)}^{-1}$ we actually don't have a need to find the inverse of Hermitian matrix H . The Hermitian conjugate inverse matrix $H_{(mxm)}^{-1}$ can be represented as follows:

$$H^{-1} = \begin{bmatrix} x_1, 0, 0, 0, 0 \\ -x_2, 1, 0, 0, 0 \\ -x_3, 0, 1, 0, 0 \\ \dots \\ -x_m, 0, 0, 0, 1 \end{bmatrix}$$

The first column of this matrix is much similar to the Hermitian matrix H with only one difference that we place the components of vector X along the first column, negating each value except the first one, without dividing it by the first component x_1 .

The example below demonstrates how to find Hermitian matrix H and its inverse H^{-1} :

$$H_{(mxm)} = \begin{bmatrix} \frac{1}{1.6728} & 0 & 0 \\ -\frac{1.4170}{1.6728} & 1 & 0 \\ -\frac{1}{1.6728} & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.5978 & 0 & 0 \\ -0.8470 & 1 & 0 \\ -0.5978 & 0 & 1 \end{bmatrix}, \quad H_{(mxm)}^{-1} = \begin{bmatrix} 1.6728 & 0 & 0 \\ -1.4170 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

Since we've already found Hermitian matrix and its inverse, let's now compute the equivalent matrix B by using formula listed above:

$$H * A^* * H^{-1} = \begin{bmatrix} 0.5978 & 0 & 0 \\ -0.8470 & 1 & 0 \\ -0.5978 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 10 & 5 & 2 \\ 5 & 6 & 5 \\ 2 & 5 & 5 \end{bmatrix} * \begin{bmatrix} 1.6728 & 0 & 0 \\ -1.4170 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

$$H * A^* * H^{-1} = \begin{bmatrix} 4.5689 & 2.9888 & 1.1955 \\ -11.6125 & 1.7645 & 3.3058 \\ -13.3085 & 2.0111 & 3.8044 \end{bmatrix}$$

After we've performing the computations listed above, let's find an equivalent matrix B :

$$B = \begin{bmatrix} 1.7645 & 3.3058 \\ 2.0111 & 3.8044 \end{bmatrix}$$

Finally, since we've found an equivalent matrix B , let's compute the second eigenvalue of matrix $A^* \leftarrow B$ by performing the same computations, as we've already done during step 1 and 2.

Note: Each eigenvector computed during step 2 is actually *NOT* an eigenvector of our input factorization matrix $A^T A$. We purposely compute these eigenvectors to be able to find an equivalent matrix B via Hermitian H and its inverse H^{-1} matrix computation. For each equivalent matrix B , during each iteration of steps 1 and 2, we're recursively aiming to compute a maximum eigenvalue σ_{max} , which is the next eigenvalue of the factorization matrix $A^T A$. The eigenvectors of the factorization matrix $A^T A$ are computed separately during the next step 4, since all eigenvalues of $A^T A$ matrix are already found.

While performing the computations during Steps 1 and 2, we're actually finding each eigenvalue of the factorization matrix $A^T A$. We're performing the following computation until we've found all specific eigenvalues and then proceed with step 4 to find all eigenvectors for the factorization matrix $A^T A$.

That's why, all what have to do at this point is to get back to Step 1 and proceed the computation of the next

eigenvalue for the factorization matrix $A^T A$.

Step 4: Computing Eigenvectors of Matrix $A^T A$:

Since we've computed all eigenvalues of $A^T A$ matrix ($\sigma_1 = 15.4310$, $\sigma_2 = 5.5573$, $\sigma_3 = 0.0116$), now we can use the Jordan-Gaussian transformation performed during step 2, to find an eigenvector for each eigenvalue of the factorization matrix $A^T A$. As the result of performing those computations, we will obtain the following eigenvectors:

$$\sigma_1 = 15.4310, V_1 = \{1.6728, 1.4170, 1.0000\},$$

$$\sigma_2 = 5.5573, V_2 = \{-1.0469, 0.5302, 1.0000\},$$

$$\sigma_3 = 0.0116, V_3 = \{0.3740, -1.1473, 1.0000\}.$$

Using the Code

This is the main function for computing SVD:

C++

Shrink ▲

```
template<typename arg="long" double="">
void svd(std::vector<std::vector<arg>> matrix, std::vector<std::vector<arg>>& s,
std::vector<std::vector<arg>>& u, std::vector<std::vector<arg>>& v)
{
    std::vector<std::vector<arg>> matrix_t;
    matrix_transpose(matrix, matrix_t);

    std::vector<std::vector<arg>> matrix_product1;
    matrix_by_matrix(matrix, matrix_t, matrix_product1);

    std::vector<std::vector<arg>> matrix_product2;
    matrix_by_matrix(matrix_t, matrix, matrix_product2);

    std::vector<std::vector<arg>> u_1;
    std::vector<std::vector<arg>> v_1;

    std::vector<arg> eigenvalues;
    compute_evd(matrix_product2, eigenvalues, v_1, 0);

    matrix_transpose(v_1, v);

    s.resize(matrix.size());
    for (std::uint32_t index = 0; index < eigenvalues.size(); index++)
    {
        s[index].resize(eigenvalues.size());
        s[index][index] = eigenvalues[index];
    }

    std::vector<std::vector<arg>> s_inverse;
    get_inverse_diagonal_matrix(s, s_inverse);

    std::vector<std::vector<arg>> av_matrix;
    matrix_by_matrix(matrix, v, av_matrix);
    matrix_by_matrix(av_matrix, s_inverse, u);
}
```

This function performs the computation of eigenvalues and eigenvectors of a given factorization matrix:

C++

Shrink ▲

```
template<typename arg="long" double="">
void compute_evd(std::vector<std::vector<arg>> matrix,
std::vector<arg>& eigenvalues, std::vector<std::vector<arg>>& eigenvectors,
std::size_t eig_count)
{
    std::size_t m_size = matrix.size();
    std::vector<arg> vec; vec.resize(m_size);
    std::fill_n(vec.begin(), m_size, 1);

    static std::vector<std::vector<arg>> matrix_i;

    if (eigenvalues.size() == 0 && eigenvectors.size() == 0)
    {
        eigenvalues.resize(m_size);
        eigenvectors.resize(eigenvalues.size());
    }
```

```

    matrix_i = matrix;
}

std::vector<std::vector<arg>> m; m.resize(m_size);
for (std::uint32_t row = 0; row < m_size; row++)
    m[row].resize(100);

Arg lambda_old = 0;

std::uint32_t index = 0; bool is_eval = false;
while (is_eval == false)
{
    for (std::uint32_t row = 0; row < m_size && (index % 100) == 0; row++)
        m[row].resize(m[row].size() + 100);

    for (std::uint32_t row = 0; row < m_size; row++)
    {
        m[row][index] = 0;
        for (std::uint32_t col = 0; col < m_size; col++)
            m[row][index] += matrix[row][col] * vec[col];
    }

    for (std::uint32_t col = 0; col < m_size; col++)
        vec[col] = m[col][index];

    if (index > 0)
    {
        Arg lambda = (m[0][index - 1] != 0) ? \
            (m[0][index] / m[0][index - 1]) : m[0][index];
        is_eval = (std::fabs(lambda - lambda_old) < 0.000000001) ? true : false;

        lambda = (std::fabs(lambda) >= 10e-6) ? lambda : 0;
        eigenvalues[eig_count] = lambda; lambda_old = lambda;
    }

    index++;
}

std::vector<std::vector<arg>> matrix_new;

if (m_size > 1)
{
    std::vector<std::vector<arg>> matrix_target;
    matrix_target.resize(m_size);

    for (std::uint32_t row = 0; row < m_size; row++)
    {
        matrix_target[row].resize(m_size);
        for (std::uint32_t col = 0; col < m_size; col++)
            matrix_target[row][col] = (row == col) ? \
                (matrix[row][col] - eigenvalues[eig_count]) : matrix[row][col];
    }

    std::vector<arg> eigenvector;
    jordan_gaussian_transform(matrix_target, eigenvector);

    std::vector<std::vector<arg>> hermitian_matrix;
    get_hermitian_matrix(eigenvector, hermitian_matrix);

    std::vector<std::vector<arg>> ha_matrix_product;
    matrix_by_matrix(hermitian_matrix, matrix, ha_matrix_product);

    std::vector<std::vector<arg>> inverse_hermitian_matrix;
    get_hermitian_matrix_inverse(eigenvector, inverse_hermitian_matrix);

    std::vector<std::vector<arg>> iha_matrix_product;
    matrix_by_matrix(ha_matrix_product,
        inverse_hermitian_matrix, iha_matrix_product);

    get_reduced_matrix(iha_matrix_product, matrix_new, m_size - 1);
}

if (m_size <= 1)
{
    for (std::uint32_t index = 0; index < eigenvalues.size(); index++)
    {
        Arg lambda = eigenvalues[index];
        std::vector<std::vector<arg>> matrix_target;
        matrix_target.resize(matrix_i.size());

        for (std::uint32_t row = 0; row < matrix_i.size(); row++)
        {
            matrix_target[row].resize(matrix_i.size());
            for (std::uint32_t col = 0; col < matrix_i.size(); col++)
                matrix_target[row][col] = (row == col) ? \

```

```

        (matrix_i[row][col] - lambda) : matrix_i[row][col];
    }

    eigenvectors.resize(matrix_i.size());
    jordan_gaussian_transform(matrix_target, eigenvectors[index]);

    Arg eigsum_sq = 0;
    for (std::uint32_t v = 0; v < eigenvectors[index].size(); v++)
        eigsum_sq += std::pow(eigenvectors[index][v], 2.0);

    for (std::uint32_t v = 0; v < eigenvectors[index].size(); v++)
        eigenvectors[index][v] /= sqrt(eigsum_sq);

    eigenvalues[index] = std::sqrt(eigenvalues[index]);
}

return;
}

compute_evd(matrix_new, eigenvalues, eigenvectors, eig_count + 1);

return;
}

```

These two functions allow us to find Hermitian matrix and its inverse:

C++ Shrink ▲

```

template<typename Arg="long" double="">
void get_hermitian_matrix(std::vector<Arg> eigenvector,
    std::vector<std::vector<Arg>>& h_matrix)
{
    h_matrix.resize(eigenvector.size());
    for (std::uint32_t row = 0; row < eigenvector.size(); row++)
        h_matrix[row].resize(eigenvector.size());

    h_matrix[0][0] = 1 / eigenvector[0];
    for (std::uint32_t row = 1; row < eigenvector.size(); row++)
        h_matrix[row][0] = -eigenvector[row] / eigenvector[0];

    for (std::uint32_t row = 1; row < eigenvector.size(); row++)
        h_matrix[row][row] = 1;
}

template<typename Arg="long" double="">
void get_hermitian_matrix_inverse(std::vector<Arg> eigenvector,
    std::vector<std::vector<Arg>>& ih_matrix)
{
    ih_matrix.resize(eigenvector.size());
    for (std::uint32_t row = 0; row < eigenvector.size(); row++)
        ih_matrix[row].resize(eigenvector.size());

    ih_matrix[0][0] = eigenvector[0];
    for (std::uint32_t row = 1; row < eigenvector.size(); row++)
        ih_matrix[row][0] = -eigenvector[row];

    for (std::uint32_t row = 1; row < eigenvector.size(); row++)
        ih_matrix[row][row] = 1;
}

```

The following function performs Jordan-Gaussian transformation of a given factorization matrix:

C++ Shrink ▲

```

template<typename Arg="long" double="">
void jordan_gaussian_transform(
    std::vector<std::vector<Arg>> matrix, std::vector<Arg>& eigenvector)
{
    const Arg eps = 0.000001; bool eigenv_found = false;
    for (std::uint32_t s = 0; s < matrix.size() - 1 && !eigenv_found; s++)
    {
        std::uint32_t col = s; Arg alpha = matrix[s][s];
        while (col < matrix[s].size() && alpha != 0 && alpha != 1)
            matrix[s][col++] /= alpha;

        for (std::uint32_t col = s; col < matrix[s].size() && !alpha; col++)
            std::swap(matrix[s][col], matrix[s + 1][col]);

        for (std::uint32_t row = 0; row < matrix.size(); row++)
        {
            Arg gamma = matrix[row][s];
            for (std::uint32_t col = s; col < matrix[row].size() && row != s; col++)

```

```


        matrix[row][col] = matrix[row][col] - matrix[s][col] * gamma;
    }

    std::uint32_t row = 0;
    while (row < matrix.size() &&
           (s == matrix.size() - 1 || std::fabs(matrix[s + 1][s + 1]) < eps))
        eigenvector.push_back(-matrix[row++][s + 1]);

    if (eigenvector.size() == matrix.size())
    {
        eigenv_found = true; eigenvector[s + 1] = 1;
        for (std::uint32_t index = s + 1; index < eigenvector.size(); index++)
            eigenvector[index] = (std::fabs(eigenvector[index]) >= eps) ?
                eigenvector[index] : 0;
    }
}
}
}

```

The following function is used to find an inverse diagonal matrix S:


C++ 

```

template<typename arg="long" double="">
void get_inverse_diagonal_matrix(std::vector<std::vector<arg>> matrix,
                                std::vector<std::vector<arg>>& inv_matrix)
{
    inv_matrix.resize(matrix.size());
    for (std::uint32_t index = 0; index < matrix.size(); index++)
    {
        inv_matrix[index].resize(matrix[index].size());
        inv_matrix[index][index] = 1.0 / matrix[index][index];
    }
}

```

This function allows us to compute an equivalent matrix B:

C++ 


```

template<typename arg="long" double="">
void get_reduced_matrix(std::vector<std::vector<arg>> matrix,
                        std::vector<std::vector<arg>>& r_matrix, std::size_t new_size)
{
    r_matrix.resize(new_size);
    std::size_t index_d = matrix.size() - new_size;
    std::uint32_t row = index_d, row_n = 0;
    while (row < matrix.size())
    {
        r_matrix[row_n].resize(new_size);
        std::uint32_t col = index_d, col_n = 0;
        while (col < matrix.size())
            r_matrix[row_n][col_n++] = matrix[row][col++];

        row++; row_n++;
    }
}

```

The following trivial function performs multiplication of two matrices:

C++ 

```

template<typename arg="long" double="">
void matrix_by_matrix(std::vector<std::vector<arg>> matrix1,
                      std::vector<std::vector<arg>>& matrix2, std::vector<std::vector<arg>>& matrix3)
{
    matrix3.resize(matrix1.size());
    for (std::uint32_t row = 0; row < matrix1.size(); row++)
    {
        matrix3[row].resize(matrix1[row].size());
        for (std::uint32_t col = 0; col < matrix1[row].size(); col++)
        {
            matrix3[row][col] = 0.00;
            for (std::uint32_t k = 0; k < matrix1[row].size(); k++)
                matrix3[row][col] += matrix1[row][k] * matrix2[k][col];
        }
    }
}

```

The following trivial function allows us to find a transpose of a given matrix:

C++



```
template<typename arg="long" double="">
void matrix_transpose(std::vector<std::vector<arg>> matrix1,
    std::vector<std::vector<arg>>& matrix2)
{
    matrix2.resize(matrix1.size());
    for (std::uint32_t row = 0; row < matrix1.size(); row++)
    {
        matrix2[row].resize(matrix1[row].size());
        for (std::uint32_t col = 0; col < matrix1[row].size(); col++)
            matrix2[row][col] = matrix1[col][row];
    }
}

</std::vector<arg>>/std::vector<arg>>/typename>
```

These two functions allow us to generate and print out matrices:

C++

Shrink ▲

```
template<typename arg="long" double="">
void generate_matrix(std::vector<std::vector<long double="">>& \
    matrix, std::size_t rows, std::size_t cols)
{
    std::srand((unsigned int)std::time(nullptr)); matrix.resize(rows);
    for (std::size_t row = 0; row < matrix.size(); row++)
    {
        matrix[row].resize(cols);
        for (std::size_t col = 0; col < matrix[row].size(); col++)
            matrix[row][col] = std::rand() % 20 - std::rand() % 20;
    }
}

template<typename arg="long" double="">
void print_matrix(std::vector<std::vector<long double="">> matrix)
{
    for (std::size_t row = 0; row < matrix.size(); row++)
    {
        for (std::size_t col = 0; col < matrix[row].size(); col++)
            std::cout << std::setprecision(5) << std::fixed << matrix[row][col] << " ";

        std::cout << "\n";
    }

    std::cout << "\n";
}

}
```

The following main function performs the demonstration of full SVD computation:

C++



```
int main(int argc, char* argv[])
{
    std::size_t matrix_size = 0;
    std::vector<std::vector<long double="">> u, v;
    std::vector<std::vector<long double="">> matrix, s;
    std::cout << "Singular Value Decomposition (SVD):\n\n";

    std::cout << "Enter size of matrix N = "; std::cin >> matrix_size;

    generate_matrix(matrix, matrix_size, matrix_size);

    std::cout << "\nA = \n"; print_matrix(matrix);

    svd(matrix, s, u, v);

    std::cout << "\nS = \n"; print_matrix(s);
    std::cout << "\nU = \n"; print_matrix(u);
    std::cout << "\nV = \n"; print_matrix(v);

    std::cin.get();
    std::cin.get();

    return 0;
}
```

Points of Interest

Nowadays, singular values decomposition is primarily used as a part of latent semantic analysis (LSA). The full SVD is used as a method that allows us to represent a given real or unitary matrix in the different orthogonal space (i.e., system of coordinates). The full SVD is the most applicable for performing natural language processing and distributional semantic, since it allows us to detect the principal components of a given incident matrix of frequencies, ignoring possible "noise". Specifically, the SVD approach is very useful if we have an incident matrix, each element of which is a frequency of occurrence of each phrase in a certain document. In this case, computing the full SVD allows us to arrange each phrase and documents in which this phrase occurs into small groups, and then perform a simple clustering to find all documents in which a certain phrase might occur.

History

- 30th November, 2018 - Published first revision of this article

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

Written By

Arthur V. Ratz

Software Developer (Senior) EpsilonDev

 Ukraine



I'm software developer, system analyst and network engineer, with over 20 years experience, graduated from L'viv State Polytechnic University and earned my computer science and information technology master's degree in January 2004. My professional career began as a financial and accounting software developer in EpsilonDev company, located at L'viv, Ukraine. My favorite programming languages - C/C++, C#.NET, Java, ASP.NET, Node.js/JavaScript, PHP, Perl, Python, SQL, HTML5, etc. While developing applications, I basically use various of IDE's and development tools, including Microsoft Visual Studio/Code, Eclipse IDE for Linux, IntelliJ/IDEA for writing code in Java. My professional interests basically include data processing and analysis algorithms, artificial intelligence and data mining, system analysis, modern high-performance computing (HPC), development of client-server web-applications using various of libraries, frameworks and tools. I'm also interested in cloud-computing, system security audit, IoT, networking architecture design, hardware engineering, technical writing, etc. Besides of software development, I also admire to write and compose technical articles, walkthroughs and reviews about the new IT- technological trends and industrial content. I published my first article at CodeProject in June 2015.





















































Comments and Discussions

You must [Sign in](#) to use this message board.

Search Comments

Spacing **(Relaxed)** Layout **(Normal)** Per page **(25)** [Update](#)

[First](#) [Prev](#) [Next](#)

 not able to get svd values greater than 3X3 dimensions 	 Arvind Kumar	19-Jun-22 17:59
 Re: not able to get svd values greater than 3X3 dimensions 	 Arthur V. Ratz	19-Jun-22 20:50
 Re: not able to get svd values greater than 3X3 dimensions 	 Arvind Kumar	19-Jun-22 22:52
 Re: not able to get svd values greater than 3X3 dimensions 	 Arthur V. Ratz	19-Jun-22 23:07
 Re: not able to get svd values greater than 3X3 dimensions 	 Arvind Kumar	20-Jun-22 0:12
 Re: not able to get svd values greater than 3X3 dimensions 	 Arthur V. Ratz	20-Jun-22 0:55
 Do not working 	 Member 14741318	11-Feb-20 1:14
 want to eigenvector calculation procedure during svd when last 2 rows are zeors after gauss reduction 	 shahpiyushn	24-Feb-19 20:23
 Please, add official references to SVD 	 Charlie1963	2-Jan-19 20:13
 My vote 	 User 11060979	27-Dec-18 2:08
 Size of matrix > 5 	 Wym2	4-Dec-18 14:40
 Re: Size of matrix > 5 	 Arthur V. Ratz	23-Dec-18 3:20
 Re: Size of matrix > 5 	 Arthur V. Ratz	27-Dec-18 0:50
 Eigenvalues 	 geoyar	4-Dec-18 9:35
Last Visit: 31-Dec-99 19:00 Last Update: 16-Feb-24 9:48 Refresh 1		
 General  News  Suggestion  Question  Bug  Answer  Joke  Praise  Rant  Admin		
Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.		