The Golub-Kahan method of computing a Singular Value Decomposition

Evelien Bogers

Bachelor's thesis
2023:K5

Lund University

Faculty of Science
Centre for Mathematical Sciences
Numerical Analysis

Centrum Scientiarum Mathematicarum

Abstract

This BSc thesis looks into the Golub-Kahan method for computing Singular Value Decompositions (SVD). Computing an SVD is expensive but can be worth it in some cases. This decomposition shows any near rank deficiency that might cause problems when solving Least Squares problems. The Golub-Kahan algorithm exploits the implicit symmetric QR algorithm to efficiently compute an SVD. A matrix is first bidiagonalized by Householder reflections, after which the Golub-Kahan SVD step is applied. This step implicitly applies the symmetric QR algorithm to the previously bidiagonalized matrix. The Golub-Kahan algorithm is more efficient and preferable to the Jacobi method for reducing a symmetric matrix to diagonal form. In this thesis, we present a theoretical analysis and a programming part regarding this process.

Popular Science Article

There is so much data in our modern world. We usually store it in big blocks of data, which we call matrices. When these blocks get very big, they are difficult to handle. The Singular Value Decomposition is a method we can apply to such matrices to make it easier to handle the big amount of data. In this decomposition method we rewrite our original matrix into three new ones that have some nice properties that we can exploit. The middle of these three matrices has characteristic values of our original block of data. This makes this middle block usually the most interesting and useful to look at. Using this method can for example help solving some mathematical problems. Another application is for improving image quality of PET scans in hospitals. Such a scan takes time, in which a body moves because of breathing for example. To reduce the blur caused by this movement, the characteristic values in the middle matrix of this decomposition are used to efficiently improve image quality.

Contents

Chapter 1

Introduction

This thesis is mostly based on the book 'Matrix Computations' by G. Golub and C. Van Loan [1]. The full Singular Value Decomposition (SVD) factors out a matrix $A \in \mathbb{R}^{m \times n}$ into an orthonormal matrix $U \in \mathbb{R}^{m \times m}$, a diagonal matrix $\Sigma \in \mathbb{R}^{m \times n}$ and another orthonormal matrix $V \in \mathbb{R}^{n \times n}$. The diagonal of Σ holds the singular values of the matrix A. The singular values of a matrix A are the non-negative square roots of the eigenvalues of $A^{\mathrm{T}}A$ or AA^{T}, whichever has the least rows.

One case in which the SVD is used is when the matrix is rank deficient. This rank deficiency causes issues when finding a solution in the classical sense to Least Squares problems. The rank of a matrix is defined by the dimension of the span of the columns of A. This means that the rank is the number of linearly independent column vectors of A. Therefore, a rank deficiency means that not all column vectors of A are linearly independent. The issue that arises when solving a Least Squares problem with a rank deficient matrix is that there will in that case be an infinite number of solutions. An SVD shows (near) rank deficiency and can solve Least Squares problems. A decomposition similar to the SVD that can solve eigenvalue problems is the eigenvalue decomposition. Such a decomposition factors a matrix $A \in \mathbb{R}^{m \times m}$ into an orthogonal matrix $Q \in \mathbb{R}^{m \times m}$, a diagonal matrix $\Lambda^{m \times m}$ with eigenvalues on the diagonal, and the transpose of Q, as such $A = Q\Lambda Q^{\mathrm{T}}$. An important difference however is that eigenvalue decompositions can only be used for square matrices whereas SVDs can be computed for any matrix $A \in \mathbb{R}^{m \times n}$. Even though the SVD is expensive to compute, it is worth the cost, since it is a decomposition and gives an arbitrarily close approximation.

The SVD can be a powerful tool. The reason for this is that it can be used for any matrix, even if it is rank deficient as stated before. Many mathematical methods of solving problems require matrices to be of full rank. Also, since an SVD is safe in the face of rank deficiency, it is also robust when faced with round off errors. Previously the conventional way to find the rank of a matrix would be to use Gaussian elimination to reduce the matrix to row echelon form. This means that all rows with only zeroes in them are at the bottom and the non zero entries are in a upper triangular shape, for example

$$\begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

In this example the matrix would be of rank 3. Using Gaussian Elimination to determine rank like this however, is not very efficient and has stability issues. If one had computed an SVD from the start, we would know about the rank deficiency and can also solve problems easily with the decomposition.

Another use of the SVD is in machine learning for extracting important data from the raw data that has been collected [2]. A process commonly used in machine learning called Principal Component Analysis builds upon the SVD, using the parts significant for this method. Principal Component Analysis is used to improve processing time for Positron emission tomography (PET) data in hospitals. Movement, such as breathing, while the scan is made will degrade image quality. Hence, a correction of the image must be made. Principal Component Analysis uses the singular values of an SVD to make such a correction [3].

There are several methods of computing an SVD, one of the earlier methods was the Jacobi method. It reduces a symmetric matrix to diagonal form by using rotations. The symmetry is exploited with multiplications from both sides. However, in many cases we need to deal with matrices that are not symmetric so another method is needed.

We will focus in this thesis on the iterative Golub-Kahan SVD algorithm. This algorithm is based upon an idea exploited by Lanczos [4]. The Golub-Kahan SVD algorithm first bidiagonalizes matrix A with the use of Householder reflections. Then it applies a Golub-Kahan SVD Step. Such a step involves multiplications with Givens rotations, and one full step is completed once the iteration has moved down the diagonal once. After such a step, the superdiagonal of the matrix is checked to see if it is zero. If it is not, meaning the matrix is not a diagonal matrix yet, another Golub-Kahan SVD Step is applied. This process continues until the matrix is diagonal. All the multiplication from the left make up our matrix U, and all multiplication from the right are V^{T}. The diagonal matrix is Σ. This method will be looked at in detail in this thesis.

In chapter 2, we shall look at the theoretical background to provide the building blocks for understanding the Golub-Kahan SVD algorithm. First the SVD is explained as well as its uses. An example of the use of an SVD will be given. Householder reflections are introduced. The implicit symmetric QR algorithm will be explained, which is the main idea behind the Golub-Kahan SVD step, which is used in the algorithm. There will also be pseudocode presented to further clarify how the algorithms work. Furthermore, a proof of the ability of the SVD to detect rank deficiency as well as its ability to represent a close approximation will be presented.

In chapter 3, we will look more closely at the Golub-Kahan SVD step and all it entails. The complete algorithm for computing the SVD is also looked at step by step in its entirety. The SVD suffers from a problem called 'sign ambiguity' or 'sign indeterminacy'. We will take a closer look at this issue and discuss a suggested solution to this.

In chapter 4, some tests that were performed will be presented and the results will also be explained in more detail. The program used for these tests takes a matrix as input and has an SVD as output. It computes the SVD by using the Golub-Kahan method. The program language is Python.

Chapter 2

Theoretical Background

2.1 Least Squares Problem

A very common problem to solve is the Least Squares problem. In such a problem we wish to find a vector $x \in \mathbb{R}^n$ which solves $Ax = b$, where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ are given and $m \geq n$ [1]. The system $Ax = b$ is overdetermined when $m > n$. For a solution in the classical sense we need b to be an element of the range(A), which it usually is not. Then we want the next best solution which is to minimize the residual $||Ax - b||$. We define a Least Squares problem as follows:

$$\text{Find } x \text{ s.t. } \min_x ||Ax - b||_2, \qquad A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m.$$

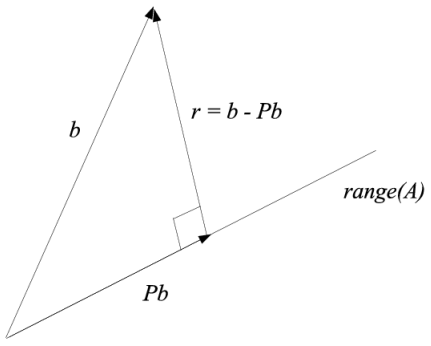

Figure 2.1: Geometric representation of the Least Squares problem

One example of a method to minimize the residual and solve a Least Squares problem is a QR decomposition. The QR decomposition rewrites a matrix as such $A = QR$, with $A \in \mathbb{R}^{m \times n}$, Q an orthogonal matrix and R an upper triangular matrix. There is however a problem when the matrix A is rank deficient, meaning some columns of A are not linearly independent. In such a case we cannot have that range(A)=span($q_1, ..q_n$) with the q_i's being the column vectors of Q. This does not hold since A is not of rank n, this is however needed for Q to be orthogonal. Furthermore, rank deficiency amounts to an infinite number of solutions to the Least Squares problem. Now we wish to overcome three issues:

1) Finding any solution to the Least Squares problem.

2) Finding the unique solution we want in the minimal 2-norm.

3) Performing reliable calculations when there are an infinite number of solutions.

One way to detect rank deficiency of a matrix A is to compute a QR decomposition with a permutation Π and some column pivoting, which gives

$$A\Pi = QR \qquad R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix} \qquad R_{11} \in \mathbb{R}^{r\times r}, R_{12} \in \mathbb{R}^{(n-r)\times r}, R_{22} \in \mathbb{R}^{(n-r)\times(n-r)}$$

with $r = \text{rank}(A)$, Q orthogonal and R_{11} upper triangular. In the case of rank deficiency, R_{22} is sufficiently small s.t.

$$||R_{22}||_2 \leq \epsilon ||A||_2,$$

for some small machine dependent parameter ϵ.

Sadly this is not always the case [1]. A matrix can be (nearly) rank deficient without R_{22} being sufficiently small. Thus QR decomposition with a permutation is not enough to determine rank deficiency reliably.

We shall look instead at the Singular Value decomposition, which is reliable when dealing with rank deficiency as we will show.

2.2 Singular Value Decomposition

The Singular Value Decomposition is a very useful tool in detecting (near) rank deficiency, as well as solving a Least Squares problem.
We define the SVD as such:

Theorem 2.2.1 (Singular value Decomposition (SVD) [1]). *If $A \in \mathbb{R}^{m\times n}$ then there exist orthogonal matrices*

$$U = [u_1, ..., u_m] \in \mathbb{R}^{m\times m}$$

and

$$V = [v_1, ..., v_n] \in \mathbb{R}^{n\times n}$$

such that

$$U^{\mathrm{T}} AV = \text{diag}(\sigma_1, ..., \sigma_p) \qquad p = \min\{m, n\}$$

where

$$\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_p \geq 0.$$

The σ_i are the singular values of matrix A. The u_i and v_i are the i-th left and right singular vectors respectively. The singular values exposed by the SVD reveal the rank of the matrix A, since the rank of a matrix is equal to the number of non-zero singular values. Many theorems in linear algebra hold only if a matrix has full rank. This makes it problematic when dealing with rounding errors and imprecise data, since they can be the cause of (near) rank deficiency.

The singular values of a matrix A are lengths of the semi-axis of the hyperellipsoid associated with the decomposition $A = U\Sigma V^{\mathrm{T}}$. To give a geometric interpretation, we have figure 2.2

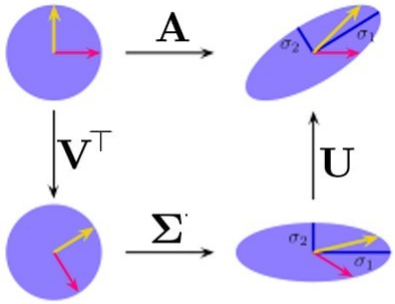

Figure 2.2: Geometric representation of the SVD [5]

Applying the matrix A to the unit circle (the pre-image) will give a hyperellipsoid (image). We can reach the same hyperellipsoid by first a rotation from orthogonal matrix V^{T}, followed by a scaling with the singular values in Σ. Finally, another rotation by matrix U to come to the same result as when A is applied to the unit circle. Once again we can see that the semi-axis of the hyperellipsoid are the singular values of A.

2.3 Householder Bidiagonalization

Many algorithms to compute the SVD first bidiagonalize a given matrix $A \in \mathbb{R}^{m\times n}$. Examples of such algorithms are due to Golub and Reinsch (1970)[6], Chan [7] and Golub-Kahan[8]. The bidiagonalization we will look at is done by using Householder reflections.

Given a matrix $A \in \mathbb{R}^{m\times n}$ with $m > n$ we wish to determine Householder matrices

$$U_B = U_1 ... U_n \in \mathbb{R}^{m\times m}$$

and

$$V_B = V_1 ... V_{n-2} \in \mathbb{R}^{n\times n},$$

such that we acquire

$$U_B^{\mathrm{T}} A V_B = B \qquad B \in \mathbb{R}^{m\times n},$$

where U_B, V_B are orthogonal and B is upper bidiagonal.

The Householder matrices each 'target' a column of matrix A. The matrices are of the form

$$U_k = I_k - \frac{2vv^{\mathrm{T}}}{v^{\mathrm{T}}v}, \qquad I_k \in \mathbb{R}^{m-k},$$

with $k = 0, ..., n-2$ our iterations. The matrices U_i upper triangularize A from the left. Followed by $V_j \in \mathbb{R}^{n-k}$ from the right to bidiagonalize A. Our vectors v are chosen depending on which column needs elements eliminated. In general we have $v = a - \hat{a}$, with $a = (a_{kk}, ..., a_{mk})$ part of columns of A, and $\hat{a}$ is a vector of zeroes

of the same length as a with the norm of a as the first entry.

$$v = \begin{bmatrix} a_{kk} \\ \vdots \\ a_{mk} \end{bmatrix} - \begin{bmatrix} ||a|| \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Then we normalize v, because it is the direction of the vector that is important for the reflection, not the size. After doing this it follows that the inner product $v^{\mathrm{T}}v = 1$. Now we want the Householder matrix U_B to be of size m and V_B of size n for the matrix multiplication. At every step after $k = 0$, we move down the diagonal, meaning that our v's become smaller in dimension and no longer match the size needed. Since we only wish to alter the elements on the diagonal and below, we take sub-matrices. To get to the correct size, we fill the sub-matrices into an identity matrix of size n and the Householder matrix becomes, for example at k=3:

$$U_3 = I - 2vv^{\mathrm{T}} = \left[\begin{array}{cc|ccc} 1 & 0 & \dots & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \hline \vdots & 0 & & & \\ \vdots & \vdots & & I - 2v_3v_3^{\mathrm{T}} & \\ 0 & 0 & & & \end{array}\right]$$

For V_j, we take rows instead of columns so $a = (a_{kk+1}, ..., a_{kn})$, $\hat{a} = (||a||, 0..., 0)$. The V_j's we apply to A as transpose since they work on the rows instead of columns. We alter the superdiagonal entries and everything of the submatrix. To obtain the Householder matrices we need to compute the relevant submatrix corresponding to the iteration. A pseudo-code is shown below.

Algorithm 1 Algorithm to acquire the Householder submatrix

```
def HouselholderSubMatrix(a):
      """
      This take a vector a and computes Beta for the Householder
      matrices. Beta is the submatrix of the Householder matrix
      H[:k,:k]= I-2vv^{^\mathrm{T}}.
      """
      ahat=(norm(a),0,0,...)
      v=a-ahat
      if norm(v)=0 then v=vector with zeroes
      else v=v/norm(v)
      return I-2*vv^{^\mathrm{T}}
```

A vector $a = (a_{k,k}, ..., a_{m,k})$ of matrix A is taken as input. First, $\hat{a}$ is computed, followed by v. Then v is normalized. An if statement is used in the case that norm(v) is zero, then v is a zero vector. This is to prevent a division by zero in the normalization process. Out of the algorithm comes the submatrix of the Householder reflection $I - 2vv^{\mathrm{T}}$ of size $m - k$ when computing the U's and $n - k - 1$ for the V's. The submatrix is then filled into an identity matrix of the correct size, m for U and n for V.

We cannot use Householder reflections to diagonalize the matrix A completely since the multiplication on both sides changing the diagonal entries will ruin the

factorization. Say have our first Householder matrix and we multiply our matrix to zero our first column under the diagonal. Then we continue and find the Householder matrix to zero the row next to the diagonal and multiply our matrix with this reflection. This multiplication will then alter the entries in the first column we have zeroed before, thus ruining the multiplication. The factorization looks as such, with H a Householder matrix and the ' indicating a changed entry,

$$H_1A = \begin{bmatrix} * & *' & *' & *' & *' \\ 0 & *' & *' & *' & *' \\ 0 & *' & *' & *' & *' \\ 0 & *' & *' & *' & *' \\ 0 & *' & *' & *' & *' \end{bmatrix} \rightarrow H_1AH_2^{\mathrm{T}} = \begin{bmatrix} *'' & 0 & 0 & 0 & 0 \\ *' & *'' & *'' & *'' & *'' \\ *' & *'' & *'' & *'' & *'' \\ *' & *'' & *'' & *'' & *'' \\ *' & *'' & *'' & *'' & *'' \end{bmatrix}.$$

Applying the Householder matrices to our given matrix A, we get the following computations, with the b and c elements to indicate that the entry was altered and the c elements will be further altered in next steps.

$$U_1A = \begin{bmatrix} b_{11} & c_{12} & \dots & c_{1n} \\ 0 & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ 0 & c_{m1} & \dots & c_{mn} \end{bmatrix} \quad \Rightarrow \quad U_1AV_1^{\mathrm{T}} = \begin{bmatrix} b_{11} & b_{12} & 0 & \dots & 0 \\ 0 & c_{22} & c_{23} & & \vdots \\ \vdots & c_{32} & \ddots & & \vdots \\ \vdots & \vdots & & & \vdots \\ 0 & c_{m2} & \dots & \dots & c_{mn} \end{bmatrix}$$

$$\Rightarrow U_2U_1AV_1^{\mathrm{T}} = \begin{bmatrix} b_{11} & b_{12} & 0 & \dots & \dots & 0 \\ 0 & b_{22} & c_{23} & c_{23} & & \vdots \\ \vdots & 0 & c_{33} & \ddots & & \vdots \\ \vdots & \vdots & c_{43} & & & \vdots \\ \vdots & \vdots & \vdots & & & \vdots \\ 0 & 0 & c_{m3} & \dots & \dots & c_{mn} \end{bmatrix}$$

$$\Rightarrow U_2U_1AV_1^{\mathrm{T}}V_2^{\mathrm{T}} = \begin{bmatrix} b_{11} & b_{12} & 0 & \dots & \dots & 0 \\ 0 & b_{22} & b_{23} & 0 & \dots & 0 \\ \vdots & 0 & c_{33} & c_{34} & & \vdots \\ \vdots & \vdots & c_{43} & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & & \vdots \\ 0 & 0 & c_{m3} & \dots & \dots & c_{mn} \end{bmatrix}.$$

We continue this until we have $U_BAV_B^{\mathrm{T}} = B$ with B upper bidiagonal. Notice how V_B only consists of $n-2$ Householder matrices as opposed to the $n-1$ matrices in U_B.

For this whole bidiagonalization process we generalize to the following algorithm which uses algorithm 1.

The algorithm to achieve the bidiagonalization overwrites A with $U_B^TAV_B = B$, with B upper bidiagonal. The input is any matrix $A \in \mathbb{R}^{m\times n}$ with $m \geq n$. The U_k is computed first by taking the vector $a = (a_{k,k}, ..., a_{m,k})$ from the k-th column of A. The Householder matrix is formed with the help of the previous algorithm. When it is filled into the proper sized identity matrix, A is multiplied on the left

Algorithm 2 Algorithm to Bidiagonalize a matrix using Householder reflections

```
def HouseholderBidiagonalization(A):
    for k in range(n):
        a=A[k:,k]
        Beta=HouseholderSubMatrix(a)
        U=I(m)
        U[k:,k:]=Beta
        A=U*A
        if k<n-1:
            b=A[k,k+1:]
            Alpha=HouseholderSubMatrix(b)
            V=I(n)
            V[k+1:,k+1:]=Alpha
            A=A*V^{^\mathrm{T}}
    return A
```

with U_k. Then V_k is computed. The vector taken is now a row vector of A instead of a column. The Householder matrix is computed and A is multiplied on the right with V_k^{T}. In this process, the A get overwritten by UAV^{T} every loop. The result of this algorithm is an upper bidiagonalized matrix as mentioned before. Every loop, U and V^{T} are also multiplied and updated.

If the Least Squares problem is of full rank, one can at this point solve it easily with U_B and V_B. If it is not full rank we continue to compute the SVD. To do this, since A has been bidiagonalized, we continue with the Golub-Kahan Algorithm that is discussed later.

2.4 The symmetric implicit QR iteration

The QR iteration finds eigenvalues. Since the singular values are related to the eigenvalues, this is a useful algorithm to use to help compute an SVD.

We will first have a look at the Implicit QR algorithm with a shift, also called 'Francis's Algorithm', where we use the Wilkinson shift.

Given a symmetric tridiagonal matrix $T \in \mathbb{R}^{n\times n}$

$$T = \begin{bmatrix} a_1 & b_1 & & 0 \\ b_1 & a_2 & b_2 & \\ & \ddots & \ddots & b_n \\ 0 & & b_n & a_n \end{bmatrix}.$$

Since T is symmetric, we have $T = T^{\mathrm{T}}$. Now we wish to find Q and R such that $T = QR$, with Q orthogonal, and R upper triangular. We wish to use a shift of the diagonal, this is to improve the converge rate. The rate of convergence depends on the ratio $\frac{\lambda_1}{\lambda_2}$ so when we shift the diagonal by λ_1 it improves the convergence of the iteration greatly[9].

The eigenvalues are not given however so we use a shift known as the Wilkinson shift:

$$\mu = a_n + d - \mathrm{sign}(d)\sqrt{d^2 + b_n^2},$$

with $d = \frac{a_{n-1}=a_n}{2}$. We will continue with this shift later in the paragraph.

Since we are dealing with a symmetric upper tridiagonal matrix, we can more efficiently eliminate the off diagonal entries with Givens rotations, which we will now look at.

From our matrix we pick a diagonal entry $a_{i,i} = y$ and an off diagonal entry that is unwanted $a_{i,k} = z$. Geometrically speaking, a Givens rotation rotates in a plane onto the y-axis by θ degrees such that the z-element of a (y, z) vector is zeroed. We can determine $c = \cos(\theta)$ and $s = \sin(\theta)$ for this rotation such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix}.$$

Then our rotation matrix becomes

$$G = \begin{vmatrix} c & s \\ -s & c \end{vmatrix}.$$

This can be adjusted to a more general case

$$G(i,k,\theta) = \begin{vmatrix} 1 & \vdots & & \vdots & & \\ \dots & c & \dots & s & \dots & i \\ & \vdots & & \vdots & & \\ \dots & -s & \dots & c & \dots & k \\ & \vdots & & \vdots & 1 & \end{vmatrix}.$$

Computing c and s like this, without computing θ, is computationally more attractive. We choose a entry on the diagonal to use as rotational pivot, from here on called y. We also choose the unwanted off-diagonal element that we want to zero, from here on called z. By choosing these entries we can very specifically zero entries.

We use the following algorithm [1] to compute our c and s. The extra if else

Algorithm 3 Algorithm to acquire the c and s for Givens rotations

```
1  def GivensCoefficients(y,z):
2    if z==0:
3      c=1
4      s=0
5    else:
6      if abs(z)>=abs(y):
7        t=y/z
8        s=1/((1+t**2)**(1/2))
9        c=s*t
10     else:
11       t=z/y
12       c=1/((1+t**2)**(1/2))
13       s=c*t
14   return c,s
```

condition on line 6 makes sure that c and s do not exceed abs(1). This is because cos and sin are always between -1 and 1.

We can save in computation time for the matrix multiplication by only multiplying the effected rows or columns with the c and s coefficients in a clever way as shown in algorithm 4. The c and s computed with algorithm 3 are taken as input.

It also takes the affected rows or columns of the matrix to be multiplied as vectors a and b. The vectors given as output are then replacing the respective old rows or columns as new updated rows or columns respectively.

Algorithm 4 Algorithm to multiply the effected rows or columns with the Givens rotation

```
def GivensProduct(c,s,a,b):
  for j in range(n):
    v=a[j]
    w=b[j]
    a[j]=c*v+s*w
    b[j]=-s*v+c*w
  return a,b
```

Now coming back to the Implicit QR factorization, applying these Givens rotations to our tridiagonal matrix with shift, from both sides at once because of the symmetry, we take $t_{n+1,n}$ and $t_{n,n+1}$ as y for left and right respectively such that $A \mapsto Q_0^{-1}AQ_0 = Q_0AQ_0$. The last equality by orthogonality. This is a similarity transformation[9]. It creates an unwanted element which we will call a 'bulge' that needs to be chased away. More Givens rotations follow to 'chase the bulge' and return the matrix T to tridiagonal form. This is one iteration of the implicit QR iteration. The first step looks like this:

$$(T^{(k)} - \mu^{(k)}I) = \begin{bmatrix} t_{11} - \mu^{(k)} & t_{12} & 0 & \\ t_{21} & t_{22} - \mu^{(k)} & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots \\ & 0 & \ddots & \ddots \end{bmatrix}$$

$$\Rightarrow \quad G_0^{\mathrm{T}}(T^{(k)} - \mu^{(k)}I)G_0 = \begin{bmatrix} t_{11} - \mu^{(k)} & 0 & 0 & \\ 0 & t_{22} - \mu^{(k)} & t_{23} & 0 \\ 0 & t_{32} & \ddots & \ddots \\ & 0 & \ddots & \ddots \end{bmatrix} = T^{(k+1)}.$$

There is however an unwanted nonzero element that shows up noted below as a '+'. To get rid of this we do something called chasing the 'bulge' or sometimes 'zero-chasing'[10]. Let us look at the case where $T \in \mathbb{R}^{6\times 6}$. We continue after our first Givens rotation and we form $G_1^{\mathrm{T}}TG_1$

$$G_1^{\mathrm{T}} = \begin{bmatrix} c_1 & s_1 & & & & \\ -s_1 & c_1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix} \qquad T := G_1^{\mathrm{T}}TG_1 = \begin{bmatrix} x & x & + & & & 0 \\ x & x & x & & & \\ + & x & x & x & & \\ & & x & x & x & \\ & & & x & x & x \\ 0 & & & & x & x \end{bmatrix}$$

To get rid of these unwanted nonzeroes, '+', we simply continue with the Givens rotations. We target the bulge, so for G_2 we take t_{12} or t_{21} as our z for the rotation and the diagonal entry t_{11} for our y. Continue doing this for the diagonal and off

diagonal entries corresponding to the position of the 'bulge' while moving down the diagonal. It will looks as such:

$$T := G_2^{\mathrm{T}} T G_2 = \begin{bmatrix} x & x & 0 & & & 0 \\ x & x & x & + & & \\ 0 & x & x & x & & \\ & + & x & x & x & \\ & & & x & x & x \\ 0 & & & & x & x \end{bmatrix} \qquad T := G_3^{\mathrm{T}} T G_3 = \begin{bmatrix} x & x & 0 & & & 0 \\ x & x & x & 0 & & \\ 0 & x & x & x & + & \\ & 0 & x & x & x & \\ & & + & x & x & x \\ 0 & & & & x & x \end{bmatrix}$$

$$T := G_4^{\mathrm{T}} T G_4 = \begin{bmatrix} x & x & 0 & & & 0 \\ x & x & x & 0 & & \\ 0 & x & x & x & 0 & \\ & 0 & x & x & x & + \\ & & 0 & x & x & x \\ 0 & & & + & x & x \end{bmatrix} \qquad T := G_5^{\mathrm{T}} T G_5 = \begin{bmatrix} x & x & 0 & & & 0 \\ x & x & x & 0 & & \\ 0 & x & x & x & 0 & \\ & 0 & x & x & x & 0 \\ & & 0 & x & x & x \\ 0 & & & 0 & x & x \end{bmatrix}$$

In general, using Householder reflections are a lot more efficient than Givens rotations when more than one element in the same row or column must be zeroed. Givens 'targets' specific elements in a matrix A to eliminate them while a Householder reflection 'targets' the whole columns or row after the diagonal. It is because of this that Givens rotations take up more memory and needs more computations for large dense matrices. Thus we use Householder reflections to bidiagonalize and Givens rotations for the more specific element eliminations.

We use the implicit symmetric QR algorithm later for the Golub-Kahan Algorithm.

2.5 Ability of the SVD to detect rank deficiency

The ability of the SVD to detect (near) rank deficiency is very important which is why we will prove this. The statement we want to prove is that for an SVD $A = U\Sigma V^{\mathrm{T}}$, the computed $\hat{\sigma_k}$ is close to σ_k for $k = 1, 2, ..., n$ as such

$$|\sigma_k - \hat{\sigma_k}| \leq \epsilon ||A||_2 = \epsilon \sigma_1, \qquad k = 1, 2, ..., n.$$

We denote the computed U, V and $\Sigma = \mathrm{diag}(\sigma_i)$ by $\hat{U}, \hat{V}$ and $\hat{\Sigma} = \mathrm{diag}(\hat{\sigma_i})$. Now we define

$$(2.5-1) \qquad \hat{U} = W + \triangle U, \quad W^{\mathrm{T}} W = I_m, \quad ||\triangle U||_2 \leq \epsilon,$$

$$(2.5-2) \qquad \hat{V} = Z + \triangle V, \quad Z^{\mathrm{T}} Z = I_n, \quad ||\triangle V||_2 \leq \epsilon,$$

$$(2.5-3) \qquad \hat{\Sigma} = W^{\mathrm{T}}(A + \triangle A)Z, \quad ||\triangle A||_2 \leq \epsilon ||A||_2,$$

where ϵ is the machine precision.

The SVD algorithm computes the singular values of a 'nearby' matrix $A + \triangle A$. Even though $\hat{U}$ and $\hat{V}$ are not necessarily close to U and V respectively, $\hat{\sigma_i}$ is within ϵ range of σ_i[1].

First we will state and prove a useful theorem that we use to prove our statement:

Lemma 1. *Of all matrices of rank k, A_k is closest to the matrix A.*

To prove this let us have a look at the SVD $(A - A_k) = U(\Sigma - \Sigma_k)V^{\mathrm{T}}$. The largest singular value of $A - A_k$ is σ_{k+1}, so $||A - A_k||_2 = \sigma_{k+1}$. What remains to be shown is that for any matrix B of rank k we have $||A - B||_2 \geq \sigma_{k+1}$.

To do this, we look at the dimensions of the nullspace of B, the span of B and the column space:

$$\dim(\mathscr{N}(B)) = \dim(\mathbb{R}^m) - \dim(\mathscr{R}(B)) = m - k.$$

Then we consider the columns v_i in $B = U\Sigma V^{\mathrm{T}}$ $(v_1, ..., v_m)$. Take the orthonormal space $< v_1, ...v_{k+1} >$ with dimension $k + 1$.

This space and $\mathscr{N}(B)$ are both subspaces of $\mathbb{R}^m$. The sum of them is $\dim(\mathscr{N}(B))+ \dim(< v_1, ..., v_{k+1} >) = m - k + k + 1 = m + 1$. This exceeds the dimension of $\mathbb{R}^m$ so there must be a non trivial intersection.

Let $\hat{x}$ be a nonzero vector in this intersection with $||\hat{x}||_2 = 1$. There also exist some scalar such that $\hat{x} = c_1v_1 + ... + c_{k+1}v_{k+1}$ because $\hat{x} \in < v_1, ..., v_{k+1} >$. Since the v_i's are orthogonal we can say $|c_1|^2 + ... + |c_{k+1}|^2 = ||\hat{x}||_2^2 = 1$.
Because $\hat{x} \in \mathscr{N}(B)$ we have that $B\hat{x} = 0$.
Hence,

$$(A - B)\hat{x} = A\hat{x} = \sum_{i=1}^{k+1} u_i\sigma_i v_i \cdot \sum_{i=1}^{k+1} c_i v_i = \sum_{i=1}^{k+1} \sigma_i c_i u_i v_i^2 = \sum_{i=1}^{k+1} \sigma_i c_i u_i.$$

Since all u_i's are orthonormal as well, we can say

$$||(A - B)\hat{x}||_2^2 = \sum_{i=1}^{k+1} |\sigma_i c_i v_i|^2 = \sum_{i=1}^{k+1} |\sigma_i c_i|^2 \geq \sigma_{k+1}^2 \sum_{i=1}^{k+1} |c_i|^2 = \sigma_{k+1}^2.$$

Finally with Cauchy Schwartz we have

$$||A - B||_2 \geq \frac{||(A - B)\hat{x}||_2}{||\hat{x}||_2} \geq \sigma_{k+1}.[11]$$

This means that the smallest singular value of A is only a 2-norm distance away from the set of all rank deficient matrices. □

Now we continue with our original statement to be proven

$$|\sigma_k - \hat{\sigma_k}| \leq \epsilon||A||_2 = \epsilon\sigma_1, \qquad k = 1, 2, ..., n.$$

Then, using the SVD, we want to prove that for any matrix B of rank $k - 1$, there is a nearby full rank matrix A.

From Lemma 1 we know

$$\sigma_k = \min_{rank(B)=k-1} ||A - B||_2 = \min_{rank(B)=k-1} ||\Sigma - B||_2.$$

The singular values computed in the SVD of A are actually singular values of a 'nearby' matrix $A + \triangle A$[1], which gives us

$$\hat{\sigma_k} = \min_{rank(B)=k-1} ||(A + \triangle A) - B||_2 = \min_{rank(B)=k-1} ||\hat{\Sigma} - B||_2.$$

Now we use definition (2.5-3)

$$\min_{rank(B)=k-1} ||W^{\mathrm{T}}(A + \triangle A)Z - B||_2 = \min_{rank(B)=k-1} ||W^{\mathrm{T}}AZ + W^{\mathrm{T}}\triangle AZ - B||_2.$$

So we can say that

$$|\sigma_k - \hat{\sigma_k}| = \min_{rank(B)=k-1} ||\Sigma - W^{\mathrm{T}} AZ - W^{\mathrm{T}} \triangle AZ||_2 = ||\triangle A||_2$$

Then with (2.5-3) we can conclude that

$$||\triangle A||_2 \leq \epsilon ||A||_2 \implies |\sigma_k - \hat{\sigma_k}| \leq \epsilon ||A||_2. \quad \square$$

Chapter 3

The Golub-Kahan method of computing an SVD

3.1 The Golub-Kahan SVD step

The Golub-Kahan method to compute the SVD finds U and V simultaneously by implicitly applying the symmetric QR algorithm to $A^{\mathrm{T}}A$. We start by reducing the matrix A to a bidiagonal matrix by applying Householder matrices. We will get

$$U_B^{\mathrm{T}} A V_B = \begin{bmatrix} B \\ \hline 0 \end{bmatrix} = \left[\begin{array}{cccc} d_1 & f_2 & & 0 \\ & d_2 & f_3 & \\ & & \ddots & f_n \\ 0 & & & d_n \\ \hline & & 0 & \end{array}\right].$$

To do this we use algorithm 1 and algorithm 2 as explained previously in chapter 2.3.

There are a few cases to discuss with regards to the matrix upper bidiagonal B that is obtained:

1. There are no d_k's or f_k's that are zero. No extra steps need to be taken to continue.

2. There are f_k's that are zero but all d_k's are nonzero. No extra steps need to be taken to continue.

3. Some d_k's are zero and the corresponding f_k's are zero as well. This means we will have to split B into subproblems as such:

$$B = \begin{bmatrix} B_1 & 0 \\ 0 & B_2 \end{bmatrix}.$$

Then continue with B_1 and B_2 separately. It can also happen that there are more that 2 subproblems, of course.

4. There is a d_k that is zero, but the corresponding f_{k+1} is not zero. In this case we apply Givens rotations from the right to zero the superdiagonal and split the problem into sub problems as with case 3. The zeroing of this superdiagonal requires some special attention, since we have to be careful when choosing our y and

z for the rotation. For example suppose that

$$B = \begin{bmatrix} x_{11} & x_{12} & 0 & 0 & 0 & 0 \\ 0 & x_{22} & x_{23} & 0 & 0 & 0 \\ 0 & 0 & 0 & x_{34} & 0 & 0 \\ 0 & 0 & 0 & x_{44} & x_{45} & 0 \\ 0 & 0 & 0 & 0 & x_{55} & x_{56} \\ 0 & 0 & 0 & 0 & 0 & x_{66} \end{bmatrix}.$$

Here we want to rotate such that the unwanted x_{34} becomes zero. This is accomplished when rotating in plane 3 and 4, first. We choose $y = x_{44}$ as our pivot. Our straightforward choice of the unwanted element will be $z = x_{34}$[12]. We use algorithm 3 to compute c and s, the affected rows of B can now be multiplied with algorithm 4. These multiplications to zero the unwanted element are all from the right. More than one Givens rotation is needed because the unwanted elements will move as such

$$\xrightarrow{(3,4)} \begin{bmatrix} x_{11} & x_{12} & 0 & 0 & 0 & 0 \\ 0 & x_{22} & x_{23} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_{35} & 0 \\ 0 & 0 & 0 & x_{44} & x_{45} & 0 \\ 0 & 0 & 0 & 0 & x_{55} & x_{56} \\ 0 & 0 & 0 & 0 & 0 & x_{66} \end{bmatrix} \xrightarrow{(3,5)} \begin{bmatrix} x_{11} & x_{12} & 0 & 0 & 0 & 0 \\ 0 & x_{22} & x_{23} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x_{36} \\ 0 & 0 & 0 & x_{44} & x_{45} & 0 \\ 0 & 0 & 0 & 0 & x_{55} & x_{56} \\ 0 & 0 & 0 & 0 & 0 & x_{66} \end{bmatrix}$$

$$\xrightarrow{(3,6)} \begin{bmatrix} x_{11} & x_{12} & 0 & 0 & 0 & 0 \\ 0 & x_{22} & x_{23} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_{44} & x_{45} & 0 \\ 0 & 0 & 0 & 0 & x_{55} & x_{56} \\ 0 & 0 & 0 & 0 & 0 & x_{66} \end{bmatrix}.$$

Note that on every rotation the diagonal entry used as pivot, y, changes. The reason we cannot use the zero entry as a pivot is that the resulting c and s will be 0 and 1 respectively. Multiplying our rows with such a rotation will result in a swapping of rows, which is not what we want to achieve. After these rotations B can be separated in subproblems, all upper bidiagonal. These subproblems will individually go through the next steps. With these extra steps taken for cases 3 and 4, we can continue with our matrix our matrices B_n.

Next we will apply Givens rotations from the left and right. Given an upper bidiagonal matrix $B \in \mathbb{R}^{n \times n}$ with no zeroes on the diagonal or subdiagonal, the algorithm overwrites B with the bidiagonal matrix $\hat{B} = \hat{U}^{\mathrm{T}} B \hat{V}$, where $\hat{U}$ and $\hat{V}$ are orthogonal. We shall now use the implicit symmetric QR algorithm. First, to pick a suitable shift we want to determine the smallest eigenvalue of a related symmetric matrix such as $T = B^{\mathrm{T}} B$. However, we do not want to explicitly compute this completely since we do not need all entries to determine the last eigenvalue. We can pick 4 entries of B that we need and then use this to acquire a submatrix T as such:

$$\begin{bmatrix} d_m^2 + f_m^2 & d_m f_n \\ d_m f_n & d_n^2 + f_n^2 \end{bmatrix}.$$

Let μ be the eigenvalue of this 2-by-2 submatrix of $T = B^{\mathrm{T}} B$. From the eigenvalues calculated here we want to pick the closest to $d_n^2 + f_n^2$. This to use for the first shift.

To compute the first rotation matrix c and s are acquired with algorithm 3, using $y = d_1^2 - \mu$ and $z = d_1 f_2$.

After the first multiplication of the rows with the rotation, it continues to take a new y and z, to compute a new rotation to multiply B on the left this time.

Note that when multiplying on the right we take the rows and from the left we take columns that are affected. We continue to take a new y and z before going into the next iteration that moves over the diagonal. This is the chasing of the bulge mentioned before. Once the iteration is complete, we get a new B which is upper bidiagonal with the super diagonal entries getting smaller.

This iteration of rotations from both the left and right is called a Golub-Kahan SVD step for which we have the following pseudo code:

Algorithm 5 The Golub-Kahan SVD Step

```
def GKStep(B):
  T=[[d_m^2+f_m^2,d_m f_n],
     [d_m f_n,d_n^2+f_n^2]]
Find mu, the eigenvalue of T that is closest to d_n^2 + f_n^2
  y=b_11^2-mu
  z=b_12*b_11
  for k in range(n-1):
    c,s=GivensCoefficients(y,z)
    B[:,k],B[:,k+1]=GivensProd(c,s,B[:,k],B[:,k+1])
    y=B[k,k]
    z=B[k+1,k]
    c,s=GivensCoefficients(y,z)
    B[k,:],B[k+1,:]=GivensProd(c,s,B[k,:],B[k+1,:])
    if k<n-2:
      y=B[k,k+1]
      z=B[k,k+2]
  return B
```

Doing this iteration several times will eventually make the new B into a diagonal matrix, which is what we are aiming for with regards to computing the SVD.

3.2 The complete SVD algorithm

With everything put together the final completed algorithm will look as below. Given $A \in \mathbb{R}^{m \times n}$ (m≥n) and a small unit roundoff ϵ.

We bidiagonalize matrix A with Householder reflections as described in chapter 2.3. Then we have to look if there are zero elements on the diagonal and superdiagonal, starting at the last element in the matrix, a_{nn}, and moving up. If there are any zeroes on the diagonal with a corresponding nonzero superdiagonal, A has to be split into sub problems as discussed before in chapter 3.1. If the superdiagonal entries are zero we can move further over the diagonal. The process will look as follows.

$$A = \left|\begin{array}{ccc|c} A_{11} & 0 & 0 & p \\ 0 & A_{22} & 0 & n-p-q \\ 0 & 0 & A_{33} & q \\ 0 & 0 & 0 & m-n \end{array}\right| .$$

The goal is to diagonalize A, and we are working at this moment in the algorithm with the potentially split problems of B. Each subproblem goes through this algorithm. At each step, we overwrite A with $U^{\mathrm{T}}AV = D + E$. Here $U \in \mathbb{R}^{m\times n}$, $V \in \mathbb{R}^{n\times n}$ are orthogonal, $D \in \mathbb{R}^{m\times n}$ is diagonal, and E, some small error term, satisfies $||E||_2 \cong \epsilon||A||_2$. First A_{33} is checked and the diagonal entries are stored if the superdiagonal entries are zero. Moving on to checking A_{22}, this upper bidiagonal problem will get smaller and smaller each iteration of the Golub-Kahan SVD Step. This algorithm will zero superdiagonal entries and needs to be completed to chase away the resulting 'bulge'. Then we repeat the process of checking A_{33} and applying the Golub-Kahan SVD step to A_{22} until the matrix A is a diagonal matrix. Note that it can happen that a diagonal entry turns zero and the superdiagonal needs to be zeroed as well. This process is as discussed before with the splitting of B into subproblems. Ultimately the goal is to make q larger with each iteration until $q = n$, meaning A is diagonal.

The complete algorithm[1] will look like this

Algorithm 6 The SVD algorithm

```
def algorithm(A):
    B=HouseholderBidiagonalization(A)

    We set any element in $A$ to zero when it is small enough,
    |a_{i,i+1}|>= epsilon (|a_{i,i}| +|a_{i+1,i+1}|
    with i=1,...,n-1

Repeat
    Find the smallest p and largest q such that A_33 is diagonal
   and A_22 has a nonzero superdiagonal.
    if q=n:
        break
    if A_22 has a zero on the diagonal:
        Use Givens rotation to zero the superdiagonal
        Go back to Repeat
    else:
        A_22=GKStep(A_22)
        Go back to Repeat
    Return B
```

Finally, we have a diagonal matrix. This is the Σ of our SVD. Collecting all the U, V and G's and multiplying then on their respective sides we will acquire $A = U\Sigma V^{\mathrm{T}}$.

3.3 Sign Ambiguity

There is one last problem to discuss. The SVD suffers from a sign indeterminacy or often called sign ambiguity[13]. This is not often mentioned in discussions of the SVD, however it can significantly effect conclusions when it is not taken into account.

The orthogonal matrices U and V computed in the SVD are not unique. They hold the left and right singular vectors of a matrix A respectively. There are many linear combinations of U and V that can be made such that $A = U\Sigma V^{\mathrm{T}}$, with

$\Sigma = diag(\sigma_1, ..., \sigma_k)$. Meaning there is not a one specific sign that makes the correct decomposition. We do have that the singular values are non-negative, which restricts the number of options slightly. The sign ambiguity can cause problems when U or V is needed for drawing a conclusion and someone is not aware of this problem. To find to correct decomposition so that $A = U\Sigma V^{\mathrm{T}}$, one must determine the signs of the singular vectors in such a way that this factorization holds.

One suggestion for the determination of the sign of a singular vector is make it the same as that of the direction of the vector it represents[13]. This means that we look at the original matrix A and correct the sign of the singular vector according to the sign of the inner product of the singular vector and the corresponding vector in A.

To clarify this process, pseudo code[13] is presented below

Algorithm 7 Determination of the sign of a singular vector

First, we correct the left singular vectors:
for each left singular vector, $k = 1, 2, ..., K$ and for y_j being the jth column of Y **do**
  $Y = X - \sum_{m=1, m\neq k}^{K} \sigma_m u_m v_m^{\mathrm{T}}$
  Let $s_k^{left} = \sum_{j=1}^{J} \mathrm{sign}(u_k^{\mathrm{T}} y_j)(u_k^{\mathrm{T}} y_j)^2$
end for
Second, all the right singular vectors are corrected:
for each right singular vector, $k = 1, 2, ..., K$ and for y_i being the ith transposed row of Y **do**
  $Y = X - \sum_{m=1, m\neq k}^{K} \sigma_m u_m v_m^{\mathrm{T}}$
  Let $s_k^{right} = \sum_{i=1}^{I} \mathrm{sign}(v_k^{\mathrm{T}} y_i)(v_k^{\mathrm{T}} y_i)^2$
end for
for each singular vector, k=1,2,...,K **do**
  if $|s_k^{left}||s_k^{right}| < 0$ **then**
    if $|s_k^{left}| < |s_k^{right}|$ **then**
      $s_k^{left} = -s_k^{left}$
    else: $s_k^{right} = -s_k^{right}$
    end if
  end if
  $\hat{u}_k = sign(s_k^{left}) u_k$
  $\hat{v}_k = sign(s_k^{right}) v_k$
end for

Chapter 4

Testing

Multiple tests of the program computing an SVD using the Golub-Kahan method were performed. The program being tested was written in Python. It takes as input a matrix of type np.ndarray and the output are three np.arrays. The first is $U \in \mathbb{R}^{m \times m}$, the second is just the diagonal entries of Σ and the last is $V^{\mathrm{T}} \in \mathbb{R}^{n \times n}$. This means we computed a full SVD, not a reduced SVD.

4.1 Test 1

Given a matrix $A \in \mathbb{R}^{3 \times 3}$ we shall calculate the SVD by hand and compare it to the output of the program. We have given matrix

$$A = \begin{bmatrix} 4 & 0 \\ 3 & -5 \end{bmatrix}.$$

Step 1 is to bidiagonalize A using Householder reflections. We take $a_1 = (4, 3)$ which gives $\hat{a_1} = (5, 0)$. Thus $v_1 = a_1 - \hat{a_1} = (-1, 3)$. v_1 then gets normalised, giving

$$v = \frac{v_1}{||v_1||} = \left(\frac{-1}{\sqrt{10}}, \frac{3}{\sqrt{10}} \right).$$

Everything needed for computing the first Householder matrix is available, this gives

$$H_1 = I - 2vv^{\mathrm{T}} = \begin{bmatrix} \frac{8}{10} & \frac{6}{10} \\ \frac{6}{10} & \frac{-8}{10} \end{bmatrix}.$$

Multiplying this with A from the left gives us

$$H_1 A = \begin{bmatrix} 5 & -3 \\ 0 & 4 \end{bmatrix} = B.$$

Now we have a upper bidiagonal matrix with no zeroes on the diagonal. This means that the Golub-Kahan SVD Step should be applied next.

A suitable shift is needed, μ, the eigenvalue of $T = B^{\mathrm{T}}B$ that is closest to t_{22}.

$$T = B^{\mathrm{T}}B = \begin{bmatrix} 25 & -15 \\ -15 & 25 \end{bmatrix}, \quad \mu = 10 \vee \mu = 40.$$

We shall take $\mu = 10$ as the shift. The first Givens rotation will thus be

$$\begin{bmatrix} 5^2 - 10 & -15 \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix}$$

with $c = \frac{1}{\sqrt{2}}$ and $s = \frac{-1}{\sqrt{2}}$. Multiplying this rotation transposed with B from the right gives

$$B := BG_1^{\mathrm{T}} = \begin{bmatrix} 5 & -3 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} c & -s \\ s & c \end{bmatrix} = \begin{bmatrix} \frac{8}{\sqrt{2}} & \frac{2}{\sqrt{2}} \\ \frac{-4}{\sqrt{2}} & \frac{4}{\sqrt{2}} \end{bmatrix}.$$

The G_1 matrix is transposed because the multiplication is from the right. Now the second part of the Golub-Kahan Step follows, this is a rotation which multiplies from the left such that

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} \frac{8}{\sqrt{2}} \\ \frac{-4}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix}.$$

This holds when

$$c = \frac{1}{\sqrt{1 + \frac{2 \cdot \sqrt{2}}{\sqrt{2} \cdot 4}}} = \frac{2}{\sqrt{5}} \text{ and } s = \frac{1}{-2\sqrt{1 + \frac{2 \cdot \sqrt{2}}{\sqrt{2} \cdot 4}}} = \frac{-1}{\sqrt{5}}.$$

Now we multiply from the left

$$B := G_2 B = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} \frac{8}{\sqrt{2}} & \frac{2}{\sqrt{2}} \\ \frac{-4}{\sqrt{2}} & \frac{4}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{20}{\sqrt{10}} & 0 \\ 0 & \frac{10}{\sqrt{10}} \end{bmatrix}.$$

The acquired diagonal matrix is Σ. To acquire our U and V^{T} we multiply our reflections and rotations. Putting it all together we have

$$A = \begin{bmatrix} 4 & 0 \\ 3 & -5 \end{bmatrix} = G_2 H_1 \Sigma G_1$$

$$= \begin{bmatrix} \frac{2}{\sqrt{5}} & \frac{-1}{\sqrt{5}} \\ \frac{1}{\sqrt{5}} & \frac{2}{\sqrt{5}} \end{bmatrix} \begin{bmatrix} \frac{8}{10} & \frac{6}{10} \\ \frac{6}{10} & \frac{-8}{10} \end{bmatrix} \begin{bmatrix} \frac{20}{\sqrt{10}} & 0 \\ 0 & \frac{10}{\sqrt{10}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$A = U\Sigma V^{\mathrm{T}} = \begin{bmatrix} \frac{1}{\sqrt{5}} & \frac{2}{\sqrt{5}} \\ \frac{2}{\sqrt{5}} & \frac{-1}{\sqrt{5}} \end{bmatrix} \begin{bmatrix} \frac{20}{\sqrt{10}} & 0 \\ 0 & \frac{10}{\sqrt{10}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}.$$

The result of the program being run to compute an SVD of A is as follows:

$$A = \begin{bmatrix} 0.4472136 & 0.89442719 \\ 0.89442719 & -0.4472136 \end{bmatrix} \begin{bmatrix} 6.32455532 & 0 \\ 0 & 3.16227766 \end{bmatrix} \begin{bmatrix} 0.70710678 & 0.70710678 \\ -0.70710678 & 0.70710678 \end{bmatrix}.$$

As we can see here, the SVD computed by the program is the same as the one computed by hand. To confirm if it is a correct decomposition, we multiply our result to hopefully obtain the original matrix A.

$$A = \begin{bmatrix} 4 & 0 \\ 3 & -5 \end{bmatrix} = U\Sigma V^{\mathrm{T}} = \begin{bmatrix} \frac{1}{\sqrt{5}} & \frac{2}{\sqrt{5}} \\ \frac{2}{\sqrt{5}} & \frac{-1}{\sqrt{5}} \end{bmatrix} \begin{bmatrix} \frac{20}{\sqrt{10}} & 0 \\ 0 & \frac{10}{\sqrt{10}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{20}{\sqrt{50}} & \frac{20}{\sqrt{50}} \\ \frac{40}{\sqrt{50}} & \frac{-10}{\sqrt{50}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{20}{\sqrt{100}} - \frac{20}{\sqrt{100}} & \frac{20}{\sqrt{100}} + \frac{20}{\sqrt{100}} \\ \frac{40}{\sqrt{100}} + \frac{10}{\sqrt{100}} & \frac{40}{\sqrt{100}} - \frac{10}{\sqrt{100}} \end{bmatrix} = \begin{bmatrix} 0 & 4 \\ 5 & 3 \end{bmatrix}.$$

We can see our entries of the original matrix A but this is clearly not our original matrix. Note that in the calculation process by hand and the program follow nearly

the same steps. The sign ambiguity algorithm has not successfully been integrated into the program.

Playing around with the signs, we can find our original matrix as follows.

$$A = \begin{bmatrix} 0.4472136 & 0.89442719 \\ 0.89442719 & -0.4472136 \end{bmatrix} \begin{bmatrix} 6.32455532 & 0 \\ 0 & 3.16227766 \end{bmatrix} \begin{bmatrix} 0.70710678 & -0.70710678 \\ 0.70710678 & 0.70710678 \end{bmatrix}$$

$$= \begin{bmatrix} 4.00000001e+00 & -2.23606797e-08 \\ 2.99999998e+00 & -5.00000000e+00 \end{bmatrix}.$$

This is indeed our original matrix and we can thus confirm the issue of sign ambiguity.

Thus this test is confirmed successful, except for the sign ambiguity issue, and we can move on to the next test.

4.2 Test 2

We do know about the sign ambiguity issue so simply multiplying our result will not be productive. Therefore we shall run the program and test if our resulting U and V^{T} are orthonormal matrices. For this test we let Python generate a random matrix $A \in \mathbb{R}^{6\times 6}$,

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

The output, which should be $U\Sigma V^{\mathrm{T}}$, after running the program gives

$$\begin{bmatrix} -0.33351717 & 0.1817529 & -0.01344654 & -0.07061632 & -0.74856017 & -0.53872292 \\ -0.29331515 & -0.40056124 & 0.07574729 & -0.80323403 & -0.10893395 & 0.30121047 \\ 0.6851945 & -0.13459746 & 0.27165799 & -0.37180802 & 0.08218539 & -0.54184713 \\ 0.11028259 & -0.16562363 & -0.94852665 & -0.13041141 & 0.07921172 & -0.19344801 \\ 0.55409585 & 0.27397586 & -0.12870201 & -0.07498138 & -0.55613866 & 0.5351994 \\ 0.11856033 & -0.82819451 & 0.06337561 & 0.43469285 & -0.32476441 & 0.03988777 \end{bmatrix}$$

$$\begin{bmatrix} 4.95115958 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.42422995 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2.98169733 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.17858772 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.33487483 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.10845986 \end{bmatrix}$$

$$\begin{bmatrix} 0.32103966 & -0.55922672 & 0.48695616 & -0.37941773 & -0.32886369 & -0.30816159 \\ 0.62483237 & 0.37056336 & -0.32684297 & 0.04259879 & 0.01166801 & -0.60290147 \\ 0.60432652 & 0.20238096 & 0.20233172 & -0.28228073 & 0.2839549 & 0.62656243 \\ -0.30225325 & 0.27975153 & 0.41073522 & -0.36959797 & 0.61814267 & -0.37812116 \\ 0.17100571 & 0.02694759 & 0.57595862 & 0.78718751 & 0.12240534 & -0.06045908 \\ -0.14390107 & 0.65574624 & 0.33864065 & -0.13519718 & -0.64342733 & 0.04832036 \end{bmatrix}.$$

Computing the multiplication $U^{\mathrm{T}}U$ and $V^{\mathrm{T}}V$, gives identity matrices. This means that U and V are indeed orthogonal matrices as we wish for with the decomposition. Finally, we need the norm of the column vectors of U and V to be 1 to be able to conclude that these matrices are orthonormal. Upon calculation, the norm of the column vectors are all 1, accurate to e^{-14}.

A good thing to check at this point however, is if at least Σ is correct. Therefore, we use the numpy function in Python to compute an SVD for A of this test 2. The numpy.linalg.svd function uses different methods of computing the SVD depending on the dimensions of A. When A has at least as many rows as columns and if A has sufficiently more rows than columns then A is first reduced using the QR decomposition (if sufficient workspace is available). From the computed $A = QR$, R is zeroed below the diagonal to give $A = Q\Sigma R$. When $m \geq n$, but not much larger, A is reduced to bidiagonal form without QR decomposition. So first A is bidiagonalized. Then the left singular vectors of bidiagonal matrix are computed and the right singular vectors of bidiagonal matrix are computed. The left singular vectors are stored in U and the right singular vectors are stored in V^{T}. The computed singular values will be stored in Σ and so the svd function gives an SVD of A [14].

The following singular values were computed with the numpy svd command

$$4.95115958, 3.42422995, 2.98169733, 2.17858772, 0.33487483, 0.10845986.$$

From this we see that the correct singular values are computed.

4.3 Test 3

Look us have a look at the computation time of our program versus that of the numpy svd function in Python. To do this we use the command *time.perf_ counter_ ns()* in Python, which gives the time elapsed in nanoseconds. We once again generate a random matrix $A \in \mathbb{R}^{6\times 6}$ for this test.

The *time.perf_ counter_ ns()* for the numpy function shows 7124704291910ns as opposed to 7124739245107ns for our program. The difference is 34953197 nanoseconds, which is 0.034953197 seconds. This shows the numpy function to be more efficient computationally, but only slightly.

The explanation for this is that there are ways to make our program numerically more attractive. One such things would be in our Golub-Kahan step to store our B bidiagonal matrix as vectors, one for the diagonal and one for the superdiagonal. Another way to improve numerical efficiency is for the Householder factorizations. The Householder matrix is $I - 2vv^{\mathrm{T}}$ so every step after the first has an identity part at the top left. Multiplying the whole matrix A is inefficient, since not all entries are affected by the multiplication. A lot more efficient would be to simply multiply the affected submatrix of A.

$$HA = (I - \beta vv^{\mathrm{T}})A = A - \beta v(A^{\mathrm{T}}v)^{\mathrm{T}}, \quad A \in \mathbb{R}^{n\times n}$$

where $\beta = \frac{2}{v^{\mathrm{T}}v}$. Thus, H does not need to be explicitly computed. To impliment this multiplication we have the following pseudo code

The input is a matrix $A \in \mathbb{R}^{n\times q}$, the v vectors are acquired as we do in algorithm 1. Implementing algorithm 8 with the inclusion of the v vectors would take the place of both algorithm 1 and 2.

Algorithm 8 Multiplication with a Householder matrix

```
def Householder multiplication(A):
For p in range(q):
    s=v_k a_kp+...+v_j a_jp
    s=beta*s
    for i=k,...,j:
        a_ip = a+ip -s*v_i
```

With both the Householder algorithm improvement and the storage of B in vectors for the Golub-Kahan step, our program should get closer to the computational time of the SVD by the numpy function in python.

Chapter 5

Conclusion

Computing an SVD is very useful when a matrix is rank deficient. It can be used to solve Least Squares problems or eigenvalue problems, it is also used in other fields such as image processing and machine learning. There are several methods to compute an SVD but in this thesis we look mainly at the Golub-Kahan method of computing an SVD. First Householder matrices are used to bidiagonalize the given matrix A. After which an iteration of rotations starts. We call this one Golub-Kahan SVD Step when we have iterated over the diagonal once. Golub-Kahan SVD Steps are repeated and between every step we check if the superdiagonal has become zero. Once the matrix has fully diagonalized, we stop repeating the Golub-Kahan steps, and we have our SVD.

The computational efficiency of the program written for the Golub-Kahan SVD method can be improved upon compared to the numpy SVD function. The improvement of the Householder multiplication and the storage of the bidiagonal matrix B in vectors for the Golub-Kahan SVD step, as described in section 4.3, will make the program much more efficient. Another thing to improve upon in our program is to get a working implementation of the sign ambiguity algorithm.

We can conclude that the program is faulty when U and V are needed. This means that a least squares problem cannot be solved with this program, however an eigenvalue problem can be solved. Image processing can also make use of this program since only the Σ is used for that. The program does produce the correct singular values but it does not produce an SVD. The factorization of the output $U\Sigma V^{\mathrm{T}}$ does not gives the original matrix A. When the sign ambiguity algorithm is successfully implemented it is likely to work, but that is speculation until proven of course.

The issue of sign ambiguity arose quite late in the process of writing this thesis and is therefore a good topic for further research. Especially because this problem is not commonly discussed for the SVD as most uses of the SVD care most for the Σ, not U and V. Having looked at this problem briefly, only one article [13] seems to show up, which is obviously not much.

Bibliography

[1] Golub G, Van Loan C. Matrix Computations; 1989.

[2] Hui J. Machine Learning — Singular Value Decomposition (SVD) Principal Component Analysis (PCA); 2019. Available from: https://jonathan-hui.medium.com/machine-learning-singular-value-decomposition-svd-principal-component-analysis-pca-1d45e885e491.

[3] Thielemans K, Rathore S, Engbrant F, Razifar P. Device-less gating for PET/CT using PCA; 2011.

[4] Lanczos C. Linear Differential Operators. London: Van Nostrand; 1961.

[5] Nimbalkar R. Singular Value Decomposition and its Application in AI; 2021. Available from: https://medium.com/appengine-ai/singular-value-decomposition-and-its-application-in-ai-120397aa2047.

[6] Golub G, Reinsch C. Singular value decomposition and least squares solutions. 1970.

[7] Chan T. An Improved Algorithm for Computing the Singular Value Decomposition. 1982.

[8] Golub G, Kahan W. Calculating the Singular Values and Pseudo-Inverse of a Matrix. 1965.

[9] Watkins D. Francis's algorithm. 2011.

[10] Demmel J. Applied numerical linear algebra; 1997.

[11] Watkins D. Fundamentals of matrix computations; 1991.

[12] Fenner M. Givens Rotations and the Case of the Blemished Bidiagonal Matrix; 2016. Available from: http://drsfenner.org/blog/2016/03/givens-rotations-and-the-case-of-the-blemished-bidiagonal-matrix/.

[13] Bro R, Acar E, Kolda T. Resolving the sign ambiguity in the singular value decomposition. 2008.

[14] Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, et al. LAPACK Users' Guide. 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics; 1999.

Bachelor's Theses in Mathematical Sciences 2023:K5
ISSN 1654-6229

LUNFNA-4043-2023

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lu.se/