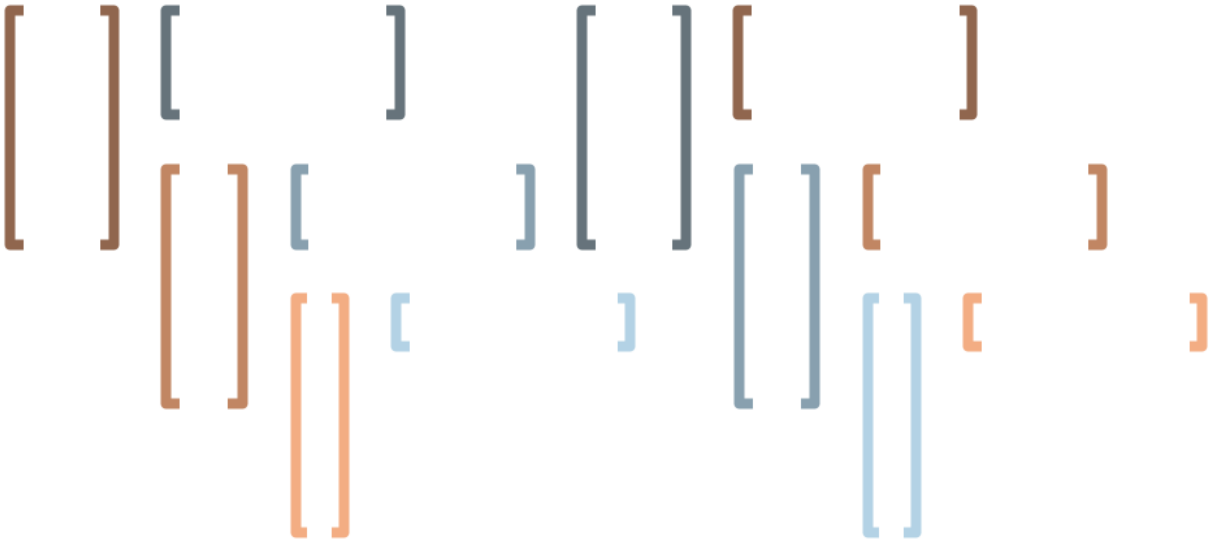


Distance to Singularity for Skew-Symmetric Matrix Pencils

A Numerical Method for Determining the Distance to the Nearest Skew-Symmetric Matrix Pencil of Given Maximal Rank

Master's Thesis in Engineering Mathematics and Computational Science

Rakel Hellberg

DEPARTMENT OF MATHEMATICAL SCIENCES

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

Master's thesis 2025

Distance to Singularity for Skew-Symmetric Matrix Pencils

A Numerical Method for Determining the Distance to the Nearest Skew-Symmetric Matrix Pencil of Given Maximal Rank

RAKEL HELLBERG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Chalmers University of Technology
Gothenburg, Sweden 2025

Distance to Singularity for Skew-Symmetric Matrix Pencils
A Numerical Method for Determining the Distance to the Nearest Skew-Symmetric Matrix Pencil of Given Maximal Rank
RAKEL HELLBERG

Supervisor: Andrii Dmytryshyn, Department of Mathematical Sciences
Examiner: Andrii Dmytryshyn, Department of Mathematical Sciences

Master's Thesis 2025
Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Abstract drawing inspired by the parametrization of the space of rank $\leq r$ skew-symmetric matrix pencils presented in section 2.1.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2025

Distance to Singularity for Skew-Symmetric Matrix Pencils
A Numerical Method for Determining the Distance to the Nearest Skew-Symmetric Matrix Pencil of Given Maximal Rank
RAKEL HELLBERG
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

The singularity of a matrix or matrix pencil is easily affected by small errors in numerical calculations. Therefore, it is motivated to consider the *distance* to singularity, rather than whether or not the matrix (pencil) is singular. In the case of unstructured matrices, the distance to singularity is well known. For structured matrices and matrix pencils, however, the question is more complex. In this thesis, a numerical method for determining the distance to the nearest skew-symmetric matrix pencil of given maximal rank is presented. The task is formulated as a minimization problem using the skew-symmetric Kronecker Canonical form and a rank-1 decomposition of skew-symmetric matrix pencils. Four different algorithms for solving the minimization problem are proposed, using the so-called vec-trick, QR-decomposition, singular value decomposition, and the GUPTRI form [5, 6]. These algorithms perform well compared to state of the art methods.

Acknowledgements

I would like to direct a big thank you to my supervisor Andrii Dmytryshyn, both for proposing this thesis project and for all of his support throughout the semester. I would also like to thank my opponents Markus Utterström and Nima Salmanpour, as well as my friend Filip Westberg, for their insightful questions which helped me understand my own work better.

Rakel Hellberg, Gothenburg, June 2025

Contents

List of Figures

1

Introduction

When performing numerical calculations, small errors from, for instance, round-off or measurements, are often present. As a small perturbation of a singular matrix is very likely to create a regular matrix, the question whether a given matrix is singular or not is ill-conditioned with respect to these errors. A more robust alternative is to consider the distance to singularity. For a given matrix, the nearest singular unstructured matrix was found already in 1907 to be given by truncating the smallest singular value in the singular value decomposition of the given matrix [18]. However, in the case of structured matrices and matrix pencils, the task of finding the nearest singularity is more complex.

In this thesis, a numerical method for finding the nearest singular skew-symmetric pencil, or, more appropriately, the nearest skew-symmetric pencil of given maximal rank, is developed. Three methods already exist that can be used to find the nearest singular skew-symmetric matrix pencil [10, 9, 8]. The first two can solve a larger class of problems; the method presented in [10] can be used to solve a variety of matrix nearness problems, and [9] presents a method for finding the nearest singular matrix valued function. The last one, [8], solves a slightly different problem which sometimes coincides with ours. In all three methods, structures such as skew-symmetry can be imposed.

The method presented here builds off of a method for finding the nearest (unstructured) matrix pencil of a given maximal rank presented in [3]. They parametrize the space of matrix pencils of rank at most r using the Kronecker canonical form and a rank-1 decomposition of matrix pencils. Using this parametrization, they formulate a minimization problem to find the nearest pencil of rank $\leq r$ to a given pencil. The minimization problem is solved using alternating least squares. In this thesis, the skew-symmetric Kronecker canonical form described in [7] and the rank-1 decomposition presented in [4] are used to parametrize the space of skew-symmetric pencils of rank at most r. Then, a minimization problem similar to the one in [3] is formulated. Four algorithms for solving the minimization problem are presented. All four use the Kronecker product and the so called vec-trick. Other main ingredients are the QR- and singular value decompositions [12], and the GUPTRI (generalized upper-triangular) form presented in [5, 6]. One algorithm was found to perform best in both speed and accuracy for small matrix pencils, while another one was faster for large pencils but less accurate. Compared to already existing methods, the algorithms presented here are fast for small matrix pencils, and at least equally

accurate. We lack the resources to compare the methods for large pencils.

An implementation of the method presented in this thesis can be found on GitHub, `https://github.com/rakeljh/nearest_singular_skew-symmetric_pencil`, as well as in appendix A.

2

Theory

Given any square matrix pencil, we wish to find the nearest skew-symmetric matrix pencil of a given maximal rank. At the very beginning of section 2.1, a few definitions are stated. Then, a metric space of of skew-symmetric matrix pencils of size $n \times n$ and rank at most r is constructed. Next, a parametrization of this space, given in [7], is presented. Using this parametrization, the task is formulated as a minimization problem in section 2.2.

2.1 The Space of Skew-Symmetric Matrix Pencils of Given Maximal Rank

A **matrix pencil** is a first degree polynomial $A - \lambda B$ with coefficients $A, B \in \mathbb{C}^{m \times n}$ and variable $\lambda \in \mathbb{C}$. A matrix pencil is **skew symmetric** if and only if $(A - \lambda B)^T = -(A - \lambda B)$. Clearly, a skew-symmetric matrix pencil is necessarily square.

A **metric space** is a set equipped with a metric, a function that measures distance from one point to another. A **metric** $d(x, y)$ must satisfy the following conditions [17].

i. $d(x, x) = 0$

ii. $d(x, y) = d(y, x)$

iii. $x \neq y \implies d(x, y) > 0$, and

iv. $d(x, y) + d(y, z) \geq d(x, z)$.

That is, the distance from any point to itself must be 0, the function must be symmetric, the distance between two distinct points must be positive, and the triangle inequality must hold.

Denote the set of all $n \times n$ skew-symmetric matrix pencils by $\mathcal{P}_n^{ss}$. Define the distance between two matrix pencils as

$$\operatorname{dist}(A - \lambda B, C - \lambda D) := \sqrt{\|A - C\|_F^2 + \|B - D\|_F^2},$$

where $\| \cdot \|_F$ denotes the Frobenius norm. It is easy to see that dist qualifies as a metric on $\mathcal{P}_n^{ss}$. Hence, $(\mathcal{P}_n^{ss}, \operatorname{dist})$ defines a metric space.

Any matrix pencil $A-\lambda B$ can be interpreted as a matrix of (first degree) polynomials in λ. Since polynomials are rational functions, matrix pencils are matrices over the field of rational functions. We denote the space of (complex) rational functions of λ by $\mathbb{C}(\lambda)$. The **rank** of an $n \times n$ matrix pencil is defined as its rank when considered as a matrix in $\mathbb{C}^{n\times n}(\lambda)$, which in turn is defined in the usual way (the dimension of the space spanned by its columns). This is sometimes called the *normal rank* [4].

Example 2.1.1 *Take, for example, the skew-symmetric matrix pencil*

$$A - \lambda B = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix} - \lambda \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -\lambda & 1 \\ \lambda & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}.$$

The rank of this pencil is 2, since the first column is linearly independent of the next two, but the second column may be multiplied by a rational function $f(\lambda) = -\frac{1}{\lambda}$ to get the third column.

A square matrix pencil $A - \lambda B$ is said to be **congruent** to another matrix pencil $C - \lambda D$ if there exists a non-singular matrix S such that $S^T AS = C$ and $S^T BS = D$ [7] (or, equivalently, $S^T(A - \lambda B)S = C - \lambda D, \quad \forall \lambda \in \mathbb{C}$). Each skew-symmetric matrix pencil is congruent to its skew-symmetric Kronecker canonical form (SS-KCF), a description of which will follow. Since congruence is an equivalence relation, two congruent matrix pencils necessarily have the same SS-KCF, and two matrix pencils with the same SS-KCF are necessarily congruent.

We will borrow the notation from [7] to build the SS-KCF. Denote by $J_k(\mu)$ the $k \times k$ Jordan block

$$J_k(\mu) := \begin{pmatrix} \mu & 1 & & \\ & \mu & \ddots & \\ & & \ddots & 1 \\ & & & \mu \end{pmatrix},$$

and by I_k the $k \times k$ identity matrix. By F_k and G_k, we denote the $k \times k+1$ matrices

$$F_k := \begin{pmatrix} 0 & 1 & & \\ & \ddots & \ddots & \\ & & 0 & 1 \end{pmatrix} \quad \text{and} \quad G_k := \begin{pmatrix} 1 & 0 & & \\ & \ddots & \ddots & \\ & & 1 & 0 \end{pmatrix}.$$

Theorem 2.1.1 (Skew-Symmetric Kronecker Canonical Form [7]) *Each skew-symmetric matrix pencil $A - \lambda B$ is congruent to a direct sum of pencils on the form*

$$\mathcal{H}_h(\mu) := \begin{pmatrix} 0 & J_h(\mu) \\ -J_h(\mu)^T & 0 \end{pmatrix} - \lambda \begin{pmatrix} 0 & I_h \\ -I_h & 0 \end{pmatrix},$$
$$\mathcal{K}_k := \begin{pmatrix} 0 & I_k \\ -I_k & 0 \end{pmatrix} - \lambda \begin{pmatrix} 0 & J_k(0) \\ -J_k(0)^T & 0 \end{pmatrix}, \quad \textit{and}$$
$$\mathcal{M}_m := \begin{pmatrix} 0 & F_m \\ -F_m^T & 0 \end{pmatrix} - \lambda \begin{pmatrix} 0 & G_m \\ -G_m^T & 0 \end{pmatrix}.$$

This sum is called the skew-symmetric Kronecker canonical form of $A - \lambda B$ and is unique up to permutation of summands.

The blocks $\mathcal{H}_h(\mu)$ and $\mathcal{K}_k$ constitute the regular part of the matrix pencil and correspond to finite and infinite[1] eigenvalues, respectively. The $\mathcal{M}_m$-blocks constitute the singular part of the pencil and correspond to the left and right minimal indices[2], which are equal for skew-symmetric matrices [7]. The number and size of regular blocks corresponding to one eigenvalue $\mu \in \overline{\mathbb{C}}$ is determined by its geometric multiplicity and the length of the Jordan chains associated to that eigenvalue. The size of a singular block is determined by the minimal index it represents, and the number of such blocks by how many times that minimal index appears.

Congruence classes of skew-symmetric matrix pencils were touched upon previously. Using the group[3] $GL_n(\mathbb{C})$, we define the class of pencils congruent to $A - \lambda B \in \mathcal{P}_n^{ss}$ as the orbit[4] [7]

$$\mathrm{O}(A - \lambda B) = \{S^T(A - \lambda B)S : S \in \mathbb{C}^{n\times n} \text{ non-singular}\}.$$

Since a skew-symmetric matrix pencil can only be congruent to other skew-symmetric matrix pencils,

$$A^T = -A \quad \Longrightarrow \quad (S^T A S)^T = S^T A^T S = -S^T A S,$$

the orbit of a matrix pencil in $\mathcal{P}_n^{ss}$ is contained in $\mathcal{P}_n^{ss}$, on which we have a metric. Thus, we may take the closure of the orbit.

Theorem 2.1.2 ([2]) *If $\mathcal{P}_1$ and $\mathcal{P}_2$ are two skew-symmetric matrix pencils in their skew-symmetric Kronecker canonical form, $\mathrm{O}(\mathcal{P}_2) \subset \overline{\mathrm{O}(\mathcal{P})}_1$ iff $\mathcal{P}_1$ can be obtained by applying the following rules to $\mathcal{P}_2$.*

i. $\mathcal{M}_{j-1} \oplus \mathcal{M}_{k+1} \mapsto \mathcal{M}_j \oplus \mathcal{M}_k$, *if* $1 \le j \le k$.

ii. $\mathcal{M}_j \oplus \mathcal{H}_{k+1}(\mu) \mapsto \mathcal{M}_{j+1} \oplus \mathcal{H}_k(\mu)$, *if* $j, k \ge 0$.

iii. $\mathcal{H}_j(\mu) \oplus \mathcal{H}_k(\mu) \mapsto \mathcal{H}_{j-1}(\mu) \oplus \mathcal{H}_{k+1}(\mu)$, *if* $1 \le j \le k$.

iv. $\mathcal{M}_p \oplus \mathcal{M}_q \mapsto \bigoplus_{i=1}^t \mathcal{H}_{k_i}(\mu_i)$ *if* $p + q + 1 = \sum_{i=1}^t k_i$ *and* $\mu_i \neq \mu_j$ *if* $i \neq j$.

Where, for ease of notation, $\mathcal{H}_k(\infty) := \mathcal{K}_k$ and, for all rules, $\mu \in \overline{\mathbb{C}}$. By the block $\mathcal{H}_0(\mu)$, we denote the empty matrix.

[1] An $n \times n$ matrix pencil is said to have eigenvalues at infinity if and only if the characteristic polynomial $\det(A - \lambda B)$ is not identically zero and has degree less than n. These eigenvalues correspond to 0 eigenvalues of the pencil $B - \lambda A$.

[2] The right (left) minimal indices of an $n \times n$ matrix pencil $A - \lambda B$ of rank $r < n$ are the degrees of the vector polynomials in the minimal polynomial basis of its right (left) null space $\mathcal{N}(A - \lambda B)$ ($\mathcal{N}\left((A - \lambda B)^T\right)$).

[3] The general linear group of degree n over $\mathbb{C}$, $GL_n(\mathbb{C})$, is the set of $n \times n$ non-singular matrices, together with matrix multiplication.

[4] Given a set X, the orbit of $x \in X$ under the action of the group G is the subset of X that can be attained by G acting on x.

If $\mathrm{O}(\mathcal{P}_2) \subset \overline{\mathrm{O}(\mathcal{P})_1}$, we say that $\mathcal{P}_1$ more **generic** than $\mathcal{P}_2$.

Recalling that skew-symmetric matrices always have even rank, we can now express the set of all skew-symmetric matrix pencils of a given size and (maximal) rank.

Theorem 2.1.3 ([7]) *Given that $2 \leq 2s < n$, with $s, n \in \mathbb{N}$, the set of $n \times n$ skew-symmetric matrix pencils of rank $\leq 2s$ is the set $\overline{\mathrm{O}(\mathcal{W}_{2s}^n)}$, where*

$$\mathcal{W}_{2s}^n = \mathrm{diag}(\underbrace{M_{\alpha+1}, \ldots, M_{\alpha+1}}_{\beta}, \underbrace{M_\alpha, \ldots, M_\alpha}_{n-2s-\beta}),$$

with $\alpha = \lfloor \frac{s}{n-2s} \rfloor$ and $\beta = s \mod (n-2s)$.

For a proof, see [7, Theorem 3.1].

Example 2.1.2 *Take the matrix pencil $\mathcal{H}_1(3) \oplus \mathcal{M}_0 \oplus \mathcal{M}_0$. It has $n = 4$ and $r = 2$. By theorem 2.1.3, we get $s = 1$, $\alpha = 0$, $\beta = 1$ and $n - 2s - \beta = 1$, which means the pencil should be contained in $\overline{\mathrm{O}(\mathcal{M}_1 \oplus \mathcal{M}_0)}$. If we apply the second rule in theorem 2.1.2, we get that $\mathcal{H}_1(3) \oplus \mathcal{M}_0 \mapsto \mathcal{H}_0(3) \oplus \mathcal{M}_1 = \mathcal{M}_1$, and we do indeed get the more generic pencil $\mathcal{M}_1 \oplus \mathcal{M}_0$.*

Any skew-symmetric matrix pencil can be written as a sum of pencils with rank 1. This will be used when formulating the minimization problem, and when solving it.

Theorem 2.1.4 (Rank-1 decomposition [4]) *Let $A - \lambda B \in \mathcal{P}_n^{ss}$ with rank r. Then, r is even and*

$$A - \lambda B = \sum_{i=1}^{s} v_i w_i^T - w_i v_i^T,$$

where $s = \frac{r}{2}$, for some vectors $v_1, \ldots, v_s \in \mathbb{C}^n$ and polynomial vectors $w_1, \ldots, w_s \in \mathbb{C}^n(\lambda)$ of degree ≤ 1.

This form is achieved by decomposing each block in the SS-KCF into a sum of rank-1 matrix pencils. For a proof, see [4, Theorem 5]. Note that for skew-symmetric pencils, which always have even rank, the rank-1 pencils come in pairs ($v_i w_i^T$ and $w_i v_i^T$) that form skew-symmetric pencils ($v_i w_i^T - w_i v_i^T$) of rank 2. Also note that for a pencil $A - \lambda B$ *not* in SS-KCF, this decomposition is still possible. Let S is be a matrix such that $S^T(A - \lambda B)S$ is in SS-KCF. Then, a rank-1 decomposition of $A - \lambda B$ is done by taking $v_i = S\tilde{v}_i$, and $w_i = S\tilde{w}_i$, where $\tilde{v}_i$ and $\tilde{w}_i$ are the vectors in the decomposition of the matrix pencil in SS-KCF.

Example 2.1.3 *Take the $\mathcal{M}_1$ block*

$$\mathcal{M}_1 = \begin{pmatrix} 0 & -\lambda & 1 \\ \lambda & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}.$$

Let

$$v = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad \textit{and} \quad w = \begin{pmatrix} 0 \\ -\lambda \\ 1 \end{pmatrix}$$

and note that

$$\mathcal{M}_1 = vw^T - wv^T.$$

Both terms is are rank 1 pencils, as

$$vw^T = \begin{pmatrix} 0 & -\lambda & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

which has rank 1 since the third column can be multiplied by $-\lambda \in \mathbb{C}(\lambda)$ *to get the second column.*

Denote by C_{2s}^n the set

$$C_{2s}^n = \left\{ \sum_{i=1}^{s} v_i w_i^T - w_i v_i^T : \begin{array}{l} v_i \in \mathbb{C}^n \text{ for } i = 1, 2, \dots \\ w_i \text{ polynomial vectors} \\ \quad \in \mathbb{C}^n(\lambda) \text{ of deg} \leq 1 \\ \quad \text{for } i = 1, 2, \dots \end{array} \right\}.$$

Clearly, C_{2s}^n only contains $n \times n$ skew-symmetric matrix pencils, due to its form, and clearly these pencils are of rank $\leq 2s$, since it is constructed from a maximum of $2s$ linearly independent vectors in $\mathbb{C}^n(\lambda)$. That is, $\overline{\mathrm{O}(\mathcal{W}_{2s}^n)} \supseteq C_{2s}^n$. By theorem 2.1.4, we also have that $\overline{\mathrm{O}(\mathcal{W}_{2s}^n)} \subseteq C_{2s}^n$, since every pencil in $\overline{\mathrm{O}(\mathcal{W}_{2s}^n)}$ can be written on the form of the elements in C_{2s}^n.

2.2 Minimization Problem

The problem of finding the nearest $n \times n$ skew-symmetric matrix pencil of rank $\leq r$ to a given $n \times n$ matrix pencil $A - \lambda B$ may be expressed as

$$\begin{array}{ll} \min & \mathrm{dist}(A - \lambda B, C - \lambda D) \\ \mathrm{St} & C - \lambda D \in C_r^n. \end{array} \tag{2.1}$$

By theorem 2.1.4, we have that

$$C - \lambda D = \sum_{i=1}^{\lfloor r/2 \rfloor} v_i w_i^T - w_i v_i^T$$

for some 0-degree polynomial vectors v_i and 1-degree (or less) polynomial vectors w_i in $\mathbb{C}^n(\lambda)$. We split each w_i into its 0-degree part and its 1-degree part,

$$w_i = w_i^0 - \lambda w_i^1,$$

and let $s = \left\lfloor \frac{r}{2} \right\rfloor$ to get that

$$\begin{aligned} C - \lambda D &= \sum_{i=1}^{s} \left(v_i (w_i^0 - \lambda w_i^1)^T - (w_i^0 - \lambda w_i^1) v_i^T \right) = \\ &= \sum_{i}^{s} (v_i (w_i^0)^T - w_i^0 v_i^T) - \lambda \sum_{i}^{s} (v_i (w_i^1)^T - w_i^1 v_i^T). \end{aligned}$$

Here, we can identify the first sum with C and the second one with D. Finally, let

$$V = \begin{pmatrix} | & & | \\ v_1 & \dots & v_s \\ | & & | \end{pmatrix}, \quad W_0 = \begin{pmatrix} | & & | \\ w_1^0 & \dots & w_s^0 \\ | & & | \end{pmatrix}, \quad \text{and} \quad W_1 = \begin{pmatrix} | & & | \\ w_1^1 & \dots & w_s^1 \\ | & & | \end{pmatrix},$$

and use the fact that

$$\begin{aligned} &\|A - VW_0^T + W_0V^T\|_F^2 + \|B - VW_1^T + W_1V^T\|_F^2 = \\ =&\left\| \begin{pmatrix} A \\ B \end{pmatrix} - \begin{pmatrix} VW_0^T - W_0V^T \\ VW_1^T - W_1V^T \end{pmatrix} \right\|_F^2, \end{aligned}$$

to formulate the minimization problem (2.1) as follows.

Given an $n \times n$ complex matrix pencil $A - \lambda B$ and $r \in [2, n)$, the nearest skew-symmetric matrix pencil $C - \lambda D$ of rank $\leq r$ is given by

$$C = VW_0^T - W_0V^T \quad \text{and} \quad D = VW_1^T - W_1V^T,$$

where V, W_0 and W_1 are a solution to

$$\begin{aligned} \min \quad & \left\| \begin{pmatrix} A \\ B \end{pmatrix} - \begin{pmatrix} VW_0^T - W_0V^T \\ VW_1^T - W_1V^T \end{pmatrix} \right\|_F \\ \text{St} \quad & V, W_0, W_1 \in \mathbb{C}^{n \times \lfloor r/2 \rfloor}. \end{aligned} \tag{2.2}$$

3

Method

In [3], the unstructured problem (without the skew-symmetry) is solved by writing the quantity to be minimized on the form $\|R - ST\|$ and using alternating least squares. Here, since W_0 and W_1 are multiplied by V from both left and right, and vice versa, we cannot get equation 2.2 on the form $\|R - ST\|$, but we can solve the system of equations

$$\begin{cases} VW_0^T - W_0V^T = A & \text{(3.1a)} \\ VW_1^T - W_1V^T = B & \text{(3.1b)} \end{cases}$$

for V and $W := \begin{pmatrix} W_0 \\ W_1 \end{pmatrix}$ alternatingly. A starting guess for V is chosen randomly.

In this chapter, we first consider the most straight-forward option, using the Kronecker product and the so-called vec-trick. Then, in sections 3.2 and 3.3, faster options are investigated, using the assumption that the input pencil is skew symmetric. In section 3.4, algorithms for improving the starting guess for V are discussed, and, lastly, in section 3.5, the termination criteria are presented.

3.1 Kronecker Product and the Vec-Trick

The **Kronecker product** ($\otimes$) of two matrices $X \in \mathbb{C}^{m\times n}$ and $Y \in \mathbb{C}^{k\times l}$, is the matrix

$$X \otimes Y = \begin{pmatrix} x_{11}Y & x_{12}Y & \dots & x_{1n}Y \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1}Y & x_{m2}Y & \dots & x_{mn}Y \end{pmatrix} \in \mathbb{C}^{mk\times nl}.$$

The **vectorization** of a matrix X, $\text{vec}(X)$, is the vector consisting of the columns of X, stacked on top of each other.

Here, we will use the fact that

$$\begin{aligned} &\left\| \begin{pmatrix} A \\ B \end{pmatrix} - \begin{pmatrix} VW_0^T - W_0V^T \\ VW_1^T - W_1V^T \end{pmatrix} \right\|_F = \\ =&\left\| \begin{pmatrix} \text{vec}(A) \\ \text{vec}(B) \end{pmatrix} - \begin{pmatrix} \text{vec}\left(VW_0^T - W_0V^T\right) \\ \text{vec}\left(VW_1^T - W_1V^T\right) \end{pmatrix} \right\|, \end{aligned} \tag{3.2}$$

and the so-called **vec-trick**,

$$\operatorname{vec}(XYZ) = (Z^T \otimes X)\operatorname{vec}(Y),$$

which holds for matrices of compatible sizes.

The second term in (3.2) can be simplified using

$$\begin{aligned} &\operatorname{vec}\left(VW_i^T - W_iV^T\right) = \operatorname{vec}\left(VW_i^T\right) - \operatorname{vec}\left(W_iV^T\right) = \\ &= \operatorname{vec}\left(VW_i^TI\right) - \operatorname{vec}\left(IW_iV^T\right) = \\ &= \left(I^T \otimes V\right)\operatorname{vec}\left(W_i^T\right) - (V \otimes I)\operatorname{vec}\left(W_i\right) = \\ &= \left((I \otimes V)\,P - (V \otimes I)\right)\operatorname{vec}(W_i) =: \widetilde{V}\operatorname{vec}(W_i), \end{aligned} \tag{3.3}$$

for $i \in \{0, 1\}$, where P is a permutation matrix such that $P \operatorname{vec} W_i = \operatorname{vec} W_i^T$. The same term can also be simplified using

$$\begin{aligned} &\operatorname{vec}\left(VW_i^T - W_iV^T\right) = \operatorname{vec}\left(VW_i^T\right) - \operatorname{vec}\left(W_iV^T\right) = \\ &= \operatorname{vec}\left(IVW_i^T\right) - \operatorname{vec}\left(W_iV^TI\right) = \\ &= (W_i \otimes I)\operatorname{vec}\left(V\right) - \left(I^T \otimes W_i\right)\operatorname{vec}\left(V^T\right) = \\ &= \left((W_i \otimes I) - (I \otimes W_i)\,P\right)\operatorname{vec}(V) =: \widetilde{W_i}\operatorname{vec}(V). \end{aligned} \tag{3.4}$$

Alternating between the two forms (3.3) and (3.4), we may find the V, W_0 and W_1 that minimize equation 2.2. First, fix V at some initial value and let

$$\operatorname{vec}\left(W_0\right) = \widetilde{V} \backslash \operatorname{vec}(A), \quad \text{and} \quad \operatorname{vec}\left(W_1\right) = \widetilde{V} \backslash \operatorname{vec}(B), \tag{3.5}$$

where $\backslash$ denotes MATLAB's backslash. Then, fixing W_0 and W_1 to be the solutions in (3.5), let

$$\operatorname{vec}(V) = \begin{pmatrix} \widetilde{W_0} \\ \widetilde{W_1} \end{pmatrix} \backslash \begin{pmatrix} \operatorname{vec}(A) \\ \operatorname{vec}(B) \end{pmatrix}. \tag{3.6}$$

Repeating this until some termination criteria is reached gives rise to algorithm 3.1.1.

Algorithm 3.1.1: vecvec

$V \leftarrow$ random starting guess
while *termination criteria not reached* **do**
  Solve for W using the vec-trick
  Solve for V using the vec-trick

3.2 Solving for W

In this section, we will consider solving (3.1a) separately from (3.1b). Naturally, everything applies to (3.1b) as well. Here, we assume the input pencil to be skew symmetric.

We note that (3.1a) is a T-congruence Sylvester equation (T-Sylvester for short). A well-known algorithm for solving Sylvester equations (on the form $AX - XB = C$) with square matrices is the Bartels-Stewart algorithm. It uses the Schur form of the coefficients (A and B) to get them on upper triangular form. Though equation 3.1a is a T-Sylvester equation, it may be possible to do something similar. However, I have not found a method for calculating the Schur form of a non-square matrix, and therefore have used the QR-decomposition and the singular value decomposition instead.

3.2.1 QR-Decomposition

Inspired by [1, Section 4.3.3], we simplify

$$VW_0^T - W_0V^T = A$$

by a QR-decomposition of V,

$$V = QR = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R_1 \\ 0 \end{pmatrix}.$$

Here, Q is an $n \times n$ unitary matrix, Q_1 is $n \times s$ with orthogonal columns, and R_1 is $s \times s$, upper triangular, and non-singular [1]. We get the equation

$$QRW_0^T - W_0R^TQ^T = A,$$

and by multiplying by Q^H from the left and $\overline{Q}$ from the right, we get

$$RW_0^T\overline{Q} - Q^HW_0R^T = Q^HA\overline{Q}.$$

Let $X := Q^HW_0$ and $E := Q^HA\overline{Q}$ to get the simplified equation

$$RX^T - XR^T = E.$$

Using the form of R and partitioning X and E analogously, we get that

$$\begin{cases} R_1X_1^T - X_1R_1^T = E_{11} & \text{(3.7a)} \\ R_1X_2^T = E_{12}. & \text{(3.7b)} \end{cases}$$

From this, we may solve for X_2 using backslash in MATLAB[1], and X_1 using back substitution as follows.

For readability, let $\mathcal{R} := R_1$, $\mathcal{X} := X_1$ and $\mathcal{E} := E_{11}$. Recalling that $\mathcal{R}$ is upper triangular and $\mathcal{E}$ is skew-symmetric, we have an equation on the form

$$\begin{pmatrix} \mathcal{R}_{11} & \mathbf{r}_1 \\ \mathbf{0}^T & r_{ss} \end{pmatrix} \begin{pmatrix} \mathcal{X}_{11}^T & \mathbf{x}_{s1} \\ \mathbf{x}_{1s}^T & x_{ss} \end{pmatrix} - \begin{pmatrix} \mathcal{X}_{11} & \mathbf{x}_{1s} \\ \mathbf{x}_{s1}^T & x_{ss} \end{pmatrix} \begin{pmatrix} \mathcal{R}_{11}^T & \mathbf{0} \\ \mathbf{r}_1^T & r_{ss} \end{pmatrix} = \begin{pmatrix} \mathcal{E}_{11} & \mathbf{e}_1 \\ -\mathbf{e}_1^T & 0 \end{pmatrix},$$

[1]Since the coefficient R_1 is upper triangular, the system can be solved for X_2 using back substitution. This can easily be solved without using MATLAB's backslash, but MATLAB's built in functions are generally faster than a simple user implementation.

where $\mathbf{x}_{s1}^T = (x_{s,1}, \ldots, x_{s,s-1})$, is the sth row of $\mathcal{X}$ except for the last element, $\mathcal{X}_{11}$ is $\mathcal{X}$ without the last row and column, and the analogous partition is made for the other vectors and matrices. This gives rise to the following three equations (not four, due to symmetry).

$$\begin{cases} \mathcal{R}_{11}\mathcal{X}_{11}^T - \mathcal{X}_{11}\mathcal{R}_{11}^T = \mathcal{E}_{11} - \mathbf{r}_1\mathbf{x}_{1s}^T + \mathbf{x}_{1s}\mathbf{r}_1^T =: \widetilde{\mathcal{E}}, & \text{(3.8a)} \\ \mathcal{R}_{11}\mathbf{x}_{s1} + \mathbf{r}_1 x_{ss} - \mathbf{x}_{1s} r_{ss} = \mathbf{e}_1, & \text{(3.8b)} \\ r_{ss}x_{ss} - x_{ss}r_{ss} = 0. & \text{(3.8c)} \end{cases}$$

We note that x_{ss} is free and choose it to be zero. Equation 3.8b simplifies to

$$\mathcal{R}_{11}\mathbf{x}_{s1} - \mathbf{x}_{1s} r_{ss} = \mathbf{e}_{1s}. \tag{3.9}$$

We choose $\mathcal{X}$ (that is, X_1) to be upper triangular as well, and get that

$$\mathbf{x}_{1s} = -\frac{1}{r_{ss}}\mathbf{e}_1.$$

Substituting $\mathbf{x}_{1s}$ and applying the choice of upper triangular $\mathcal{X}$, the same procedure can be applied to (3.8a) with $\widetilde{\mathcal{E}}$ on the right hand side. We repeat this until we have X_1 $(= \mathcal{X})$ and then set $W_0 = QX$.

Solving for W_0 and W_1 using the QR-decomposition of V and for V using the method described in section 3.1 gives rise to algorithm 3.2.1.

Algorithm 3.2.1: qrvec

$V \leftarrow$ random starting guess
while *termination criteria not reached* **do**
    Solve for W using the QR-decomposition of V
    Solve for V using the vec-trick

3.2.2 Singular Value Decomposition

Instead of the QR-decomposition, we may take the SVD of V,

$$V = RST^H.$$

Here, R is an $n \times n$ unitary matrix, T is an $s \times s$ unitary matrix, and S is an $n \times s$ matrix with the singular values of V on the diagonal starting in the top left corner, and zeros elsewhere. Inserting this into (3.1a), we get the equation

$$\begin{aligned} &RST^H W_0^T - W_0\overline{T}S^T R^T = A \\ &ST^H W_0^T \overline{R} - R^H W_0 \overline{T} S^T = R^H A \overline{R} \\ &S(R^H W_0 \overline{T})^T - (R^H W_0 \overline{T}) S^T = R^H A \overline{R}. \end{aligned} \tag{3.10}$$

Let $X := R^H W_0 \overline{T}$ and $E := R^H A \overline{R}$. Denote by S_1 the $s \times s$ diagonal block of S, that is,

$$S = \begin{pmatrix} S_1 \\ 0 \end{pmatrix},$$

and by X_1 and X_2 the s first and $n-s$ last rows of X, respectively. Make the analogous partition of E and write (3.10) as

$$SX^T - XS^T = \begin{pmatrix} S_1X_1^T - X_1S_1^T & S_1X_2^T \\ -X_2S_1^T & 0 \end{pmatrix} = \begin{pmatrix} E_{11} & E_{12} \\ E_{21} & E_{22} \end{pmatrix} = E. \tag{3.11}$$

Now, we can solve for X_2 by using backslash in MATLAB on one of the two (equivalent) equations that contain it[2]. As for X_1, we note that since S_1 is diagonal, each element in $S_1X_1^T - X_1S_1^T$ is $s_{ii}x_{ji} - s_{jj}x_{ij}$. If X_1 is chosen to be skew-symmetric, these equations simplify to

$$x_{ji} = \frac{e_{ij}}{s_{ii} + s_{jj}}.$$

Lastly, we get that

$$W_0 = R \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} T^T.$$

Solving for W_0 and W_1 using the SVD of V gives rise to algorithm 3.2.2.

Algorithm 3.2.2: svdvec

$V \leftarrow$ random starting guess
while *termination criteria not reached* **do**
  Solve for W using the SVD of V
  Solve for V using the vec-trick

3.3 Solving for V

In this section, we will consider solving the system of equations (3.1) for V. In order to do so, we wish to factorize W_0 and W_1 in a way that simplifies the equations. We need to factorize W_0 and W_1 simultaneously in order to change the variable V. We assume the input pencil to be skew symmetric.

The simultaneous SVD [14] factorizes W_0 and W_1 into

$$W_0 = PSQ^H, \quad \text{and} \quad W_1 = PTQ^H,$$

where $P \in \mathbb{C}^{n\times n}$ and $Q \in \mathbb{C}^{s\times s}$, and S and T are in block diagonal form. This gives us smaller decoupled equations that can be solved using the vec-trick, similarly to section 3.1. This should be faster than using the vec-trick method on the original system, as this method scales badly with n. For more details on that, see section 5.1 in the discussion. However, while algorithms for computing P and Q are described in [14], implementing them is out of the scope of this thesis project. Instead, we will consider another factorization of W_0 and W_1.

[2]Since the coefficient S_1 is diagonal, the system consists of separate and simple equations for each element in X_2. This can easily be solved without using MATLAB's backslash, but, as mentioned previously, MATLAB's built in functions are generally faster than a simple user implementation.

3.3.1 GUPTRI

GUPTRI [5, 6] is short for generalized upper-triangular (form), and is a generalized Schur decomposition. Given an arbitrary matrix pencil $X(\lambda) = X_0 - \lambda X_1$, GUPTRI provides a decomposition $P^H X Q$ on the form [5]

$$P^H X Q = \begin{pmatrix} X_r & * & * & * & * \\ & X_z & * & * & * \\ & & X_f & * & * \\ & & & X_i & * \\ & & & & X_l \end{pmatrix},$$

where the empty parts are zero and the asterisks denote arbitrary submatrices. The X_r and X_l blocks correspond to right and left minimal indices of X, and the X_z, X_f, and X_i blocks correspond to zero, finite, and infinite eigenvalues, respectively. We denote by X_{reg} the submatrix that includes all three regular blocks (X_z, X_f, and X_i), but excludes both singular blocks (X_r and X_l). The matrices P and Q are unitary.

Assumption 3.3.1 *For the rest of section 3.3.1, we assume that W_0 and W_1 have full rank.*

Since the set of rank deficient $n \times s$ matrices has (Lebesgue) measure zero in the space of $n \times s$ matrices, we can safely assume that a randomly sampled $n \times s$ matrix has full rank. Moreover, by the Eckart-Young-Mirsky theorem, the best rank $\leq 2s < n$ approximation C of the full rank matrix $A \in \mathbb{C}^{n \times n}$ has rank $2s$. Thus, the W_0 that minimizes the distance from $C = VW_0^T - W_0 V^T$ to A, given $V \in \mathbb{C}^{n \times s}$, is not rank deficient. Naturally, the analogous statement holds for W_1. This motivates assumption 3.3.1.

We will be using the GUPTRI form of the pencil $W(\lambda) := W_0 - \lambda W_1$. Recall that W has full rank both if W_0 and W_1 do. Since $n > s$ and W is of full rank, the GUPTRI form of W has no right singular block. It may include a regular block, which is square and at most $s \times s$. The rest consists of a left singular block. That is, $P^H W Q$ is on the form

$$P^H W Q = \begin{pmatrix} W_{\text{reg}} & * \\ 0 & W_l \end{pmatrix}.$$

The left singular block W_l corresponding to left minimal indices has a certain structure [6]. If W_{reg} is of size $\rho \times \rho$, then W_l has $\xi := s - \rho$ columns. Clearly, $\xi \leq s$. Since $n > 2s$, W_l has $2\xi + \zeta$ columns, where $\zeta = n - 2\xi - \rho = n - \xi - s \geq n - 2s > 0$. By [6, page 184], the top $\xi \times \xi$ block of W_l contains a matrix pencil with both constant and first degree part. The second $\xi \times \xi$ block has zero constant term, and the bottom ζ rows are 0. The structure of $P^H W Q$ in full is illustrated in figure 3.1.

Hence, for the matrix pencil $W(\lambda) := W_0 - \lambda W_1$, GUPTRI provides a decomposition on the form

$$W_0 = PSQ^H \quad \text{and} \quad W_1 = PTQ^H$$

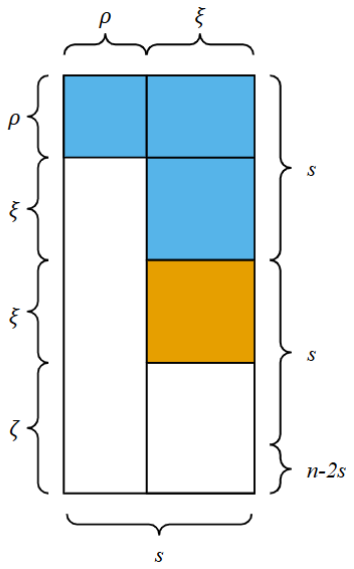


Figure 3.1: Blocks of the GUPTRI form of $W(\lambda)$. The blocks of the GUPTRI form of the full rank pencil $W(\lambda) = W_0 - \lambda W_1$. Blue blocks correspond to matrix pencils with both constant and first degree parts; the top left block is the $\rho \times \rho$ regular block W_{reg} and the blue $\xi \times \xi$ block on the second row is part of the left singular block W_l. The orange $\xi \times \xi$ block corresponds to a matrix pencil with zero constant part, which is also part of the left singular block W_l. White blocks are zero, the bottom right block making up the ζ zero rows in W_l.

where the matrices $P \in \mathbb{C}^{n \times n}$ and $Q \in \mathbb{C}^{s \times s}$ are both unitary, and S and T are on the form

$$S = \begin{pmatrix} S_1 \\ 0 \\ 0 \end{pmatrix} \quad \text{and} \quad T = \begin{pmatrix} T_1 \\ T_2 \\ 0 \end{pmatrix},$$

where S_1, T_1 and T_2 are $s \times s$ blocks. The block sizes are explained in figure 3.1.

We get the system of equations

$$\begin{cases} V\overline{Q}S^T P^T - PSQ^H V^T = A \\ V\overline{Q}T^T P^T - PTQ^H V^T = B \end{cases},$$

which we multiply by P^H from the left and by $\overline{P}$ from the right. Then, by letting $X := P^H V\overline{Q}$, $E := P^H A\overline{P}$, and $F := P^H B\overline{P}$, and partitioning the matrices into

$s \times s$ blocks as far as possible, we get the following system of equations.

$$\begin{cases} X_1 S_1^T - S_1 X_1^T = E_{11} & \text{(3.12a)} \\ -S_1 X_2^T = E_{12} & \text{(3.12b)} \\ -S_1 X_3^T = E_{13} & \text{(3.12c)} \\ X_1 T_1^T - T_1 X_1^T = F_{11} & \text{(3.12d)} \\ X_1 T_2^T - T_1 X_2^T = F_{12} & \text{(3.12e)} \\ X_2 T_2^T - T_2 X_2^T = F_{22} & \text{(3.12f)} \\ -T_1 X_3^T = F_{13} & \text{(3.12g)} \\ -T_2 X_3^T = F_{23} & \text{(3.12h)} \end{cases}$$

Note that equations 3.12a-3.12c are from equation 3.1a, with W_0 and A, and equations 3.12d-3.12h are from equation 3.1b, with W_1 and B. Also note that the equations are not fully decoupled, as 3.12e contains both X_1 and X_2.

Now, we may solve for X_3 using least squares (MATLAB's backslash) on equations 3.12c, 3.12g, and 3.12h. Then, we may solve for X_1 and X_2 using the Kronecker product and vec-trick on the remaining equations.

In more detail, let P be the permutation matrix such that $P\,\mathrm{vec}(M) = \mathrm{vec}(M^T)$ for $M \in \mathbb{C}^{s \times s}$. Then, we have the equations

$$\begin{pmatrix} S_1 \\ T_1 \\ T_2 \end{pmatrix} X_3^T = - \begin{pmatrix} E_{13} \\ F_{13} \\ F_{23} \end{pmatrix}, \tag{3.13}$$

$$\begin{pmatrix} S_1 \otimes I \\ T_2 \otimes I - (I \otimes T_2)P \end{pmatrix} \mathrm{vec}(X_2) = \begin{pmatrix} \mathrm{vec}(E_{21}) \\ \mathrm{vec}(F_{22}) \end{pmatrix},$$

and

$$\begin{pmatrix} S_1 \otimes I - (I \otimes S_1)P \\ T_1 \otimes I - (I \otimes T_1)P \\ T_1 \otimes I \end{pmatrix} \mathrm{vec}(X_1) = \begin{pmatrix} \mathrm{vec}(E_{11}) \\ \mathrm{vec}(F_{11}) \\ \mathrm{vec}(F_{12}) + (I \otimes T_2)P\,\mathrm{vec}(X_2) \end{pmatrix}. \tag{3.14}$$

These can be solved using backslash in MATLAB and we get algorithm 3.3.1, where we kept the SVD method to solve for W since it proved best (see section 4.1.1). Again, note that since the equations are not fully decoupled, we may not achieve the best least squares solution to the full system, but an approximation instead. This will yield an upper bound for the distance.

Algorithm 3.3.1: svdgup

$V \leftarrow$ random starting guess
while *termination criteria not reached* **do**
    Solve for W using the SVD of V
    Solve for V using GUPTRI of W

3.4 Phase 1 (Improving the Starting Guess)

In order to speed up the calculations, we may choose to improve the starting guess by a number of iterations where the equations are solved faster (see sections 4.1.2 and 5.1) but approximately. We call this phase 1. When this is used, the full algorithm will have the structure described in algorithm 3.4.1.

Algorithm 3.4.1: general algorithm with phase 1

$V \leftarrow$ random starting guess
while *in phase 1* **do**
    Solve approximately for W
    Solve approximately for V
while *termination criteria not reached* **do**
    Solve for W
    Solve for V

3.4.1 Singular Value Decomposition

As described in section 3.2.2, we can solve equation 3.1a for W_0 and equation 3.1b for W_1 using the SVD of V. Since solving for V requires solving the system of these two equations together, we cannot do the same for V. However, we can get an approximate solution by solving either equation 3.1a *or* equation 3.1b for V, using the SVD of W_0 or W_1. This gives rise to algorithm 3.4.2 for phase 1.

Algorithm 3.4.2: svd (phase 1)

$V \leftarrow$ random starting guess
while *in phase 1* **do**
    Solve for W_0 using the SVD of V
    Solve approximately for V using the SVD of W_0
    Solve for W_1 using the SVD of V
    Solve approximately for V using the SVD of W_1

3.4.2 GUPTRI

As described in section 3.3.1, GUPTRI gives rise to a system of equations that are solved for the variable $X = (X_1, X_2, X_3)^T$ in order to get a solution for V. Solving for X_3 is relatively fast, using backslash on equation 3.13. Solving for X_2 and X_1, however, is more involved. In this approximate method, we let equation 3.12b determine X_2 and equation 3.12e determine X_1.

Since the SVD-method of solving for W is relatively fast, we use it here as well, and get algorithm 3.4.3.

Algorithm 3.4.3: gup (phase 1)

$V \leftarrow$ random starting guess
while *in phase 1* **do**
  Solve for W using the SVD of V
  Solve approximately for V using GUPTRI:
    X_3 using (3.13)
    X_2 using (3.12b)
    X_1 using (3.12e)
    $V \leftarrow PXQ^T$

3.4.3 Triangle Inequality

By the triangle inequality,

$$\left\| \begin{pmatrix} A \\ B \end{pmatrix} - \begin{pmatrix} VW_0^T - W_0V^T \\ VW_1^T - W_1V^T \end{pmatrix} \right\|_F \leq$$
$$\leq \left\| \frac{1}{2} \begin{pmatrix} A \\ B \end{pmatrix} - \begin{pmatrix} VW_0^T \\ VW_1^T \end{pmatrix} \right\|_F + \left\| \frac{1}{2} \begin{pmatrix} A \\ B \end{pmatrix} + \begin{pmatrix} W_0 \\ W_1 \end{pmatrix} V^T \right\|_F .$$

From that, we derive the equations

$$VW_0^T = \frac{1}{2}A \tag{3.15a}$$
$$VW_1^T = \frac{1}{2}B \tag{3.15b}$$
$$\begin{pmatrix} W_0 \\ W_1 \end{pmatrix} V^T = -\frac{1}{2} \begin{pmatrix} A \\ B \end{pmatrix}, \tag{3.15c}$$

which give rise to algorithm 3.4.4.

Algorithm 3.4.4: tri (phase 1)

$V \leftarrow$ random starting guess
while *in phase 1* **do**
  Solve approximately for W_0 using (3.15a)
  Solve approximately for W_1 using (3.15b)
  Solve approximately for V using (3.15c)

3.5 Termination Criteria

Denote by d_i the distance

$$\operatorname{dist}(A - \lambda B, C - \lambda D),$$

where

$$C := VW_0^T - W_0V^T \text{ and } D := VW_1^T - W_1V^T,$$

in the ith iteration of the algorithm. The calculations finish when

$$|d_{i-1} - d_i| < \varepsilon\sqrt{\|A\|^2 + \|B\|^2},$$

for some ε, i.e., when the improvement from one iteration to the next is small, or when the maximal number of iterations N is reached.

If used, phase 1 terminates for a softer version of the first termination criterion, that is, when

$$|d_{i-1} - d_i| < \varepsilon_1\sqrt{\|A\|^2 + \|B\|^2}, \tag{3.16}$$

for some $\varepsilon_1 > \varepsilon$.

4

Results

In this chapter, all computational results will be presented. First, in section 4.1, the results of the algorithms presented in the previous chapter are compared briefly to determine which algorithm is best, i.e., fastest and most accurate. Then, in section 4.2, the results of our final algorithm are shown together with the results of other already existing methods. For interpretations of the results, see the discussion chapter.

In all computations, the input pencils were full rank, skew symmetric and random, where the elements in both matrices were generated by uniform sampling in $\{x+yi : x, y \in [-1, 1]\}$ or, in the real case, in $[-1, 1]$. The termination criteria used for our method were as described in section 3.5, with $\varepsilon = 10^{-7}$ and $N = 3000$.

4.1 Algorithm Comparison

We wish to find an algorithm that produces good results, that is, singular skew-symmetric pencils close to the original (input) pencil. We also wish for it to be fast. In the first part of section 4.1.1, the algorithms presented in sections 3.1 and 3.2, with different methods of solving for W, are compared. This is followed by a comparison of the algorithm presented in section 3.3, with a different method for solving for V, to one of the previous ones. Lastly, in section 4.1.2, we compare the phase 1 algorithms from section 3.4. While the choice of algorithm may depend on the size of the pencil, it has not been possible to run the method for very large pencils during this work, due to limited resources.

Since algorithm 3.1.1 (vecvec) is the only method that can handle non skew-symmetric input pencils, all comparisons were done for skew-symmetric input.

4.1.1 Main Algorithm

First, we will compare the different methods for solving for W, that is, algorithms 3.1.1 (vecvec), 3.2.1 (qrvec) and 3.2.2 (svdvec). The distance $\text{dist}(A - \lambda B, C - \lambda D)$ in each iteration, when searching for the nearest singular skew-symmetric pencil to a complex pencil of size 20×20, for four representative runs, is shown in figure 4.1. The time they take to find the nearest singular pencil and the nearest pencil of rank $\leq \frac{n}{4}$, for complex input pencils of size $n \times n$ with $n \in \{6, 12, ..., 30\}$, is displayed in

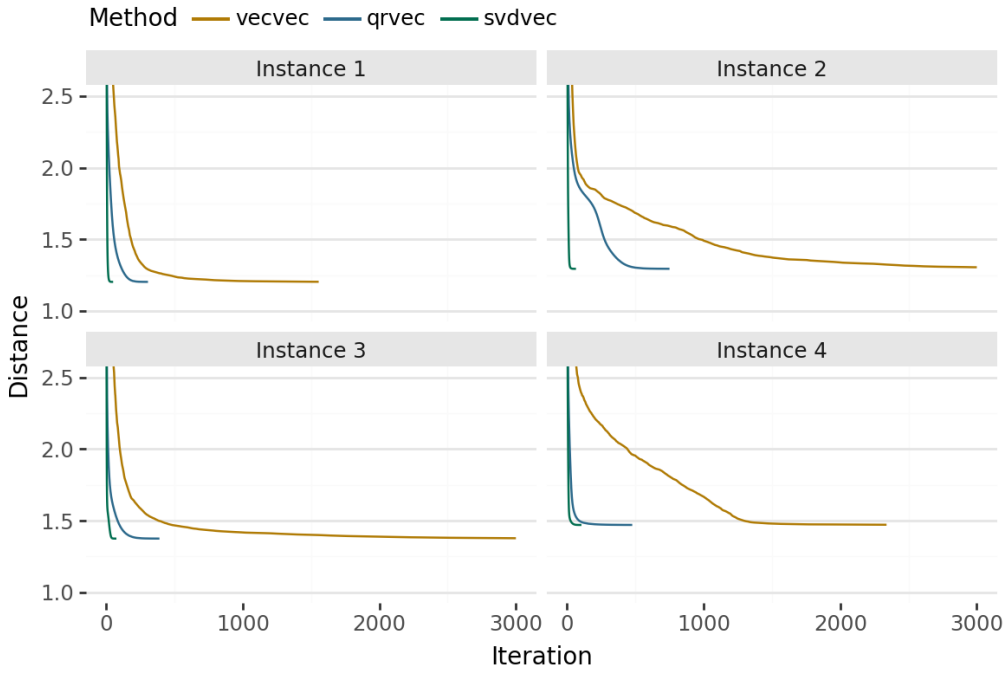


Figure 4.1: Distance per iteration using different algorithms for solving for W. Distance per iteration in four representative runs of algorithms 3.1.1 (vecvec), 3.2.1 (qrvec) and 3.2.2 (svdvec). Four random complex pencils of size 20×20 were used as input and the same initial guess V was used for all methods.

figure 4.2. The distance results for the same pencils are displayed in figure 4.3. It is clear that svdvec performs best out of these three, in both time and accuracy.

Second, we compare different methods to solve for V. Since the SVD method proved best for solving for W, we keep that when comparing the vec-trick and GUPTRI methods for solving for V. The distance per iteration, when searching for the nearest singular pencil, of the algorithms 3.2.2 (svdvec) and 3.3.1 (svdgup) are shown in figure 4.4. The average times and distances are shown in figures 4.5 and 4.6, respectively. While svdgup is faster for pencils larger than about 20×20, the accuracy is not on par with svdvec.

4.1.2 Phase 1

In this section, we will compare the phase 1 methods 3.4.2 (svd), 3.4.3 (gup), and 3.4.4 (tri). For comparison, we include svdvec (algorithm 3.2.2), which proved best for small input pencils. First, we will have a look at the behavior per iteration of the different methods. The distance $\text{dist}(A - \lambda B, C - \lambda D)$ in each iteration of four representative runs for complex 40×40 input pencils is shown in figure 4.7. The GUPTRI method, algorithm 3.4.3, behaves differently from the others. Both algorithms 3.4.4 and 3.4.2 behave as expected, with algorithm 3.4.2 (svd) giving a better estimate.

Next, we compare the time per iteration for algorithms 3.4.4 (tri) and 3.4.2 (svd) to

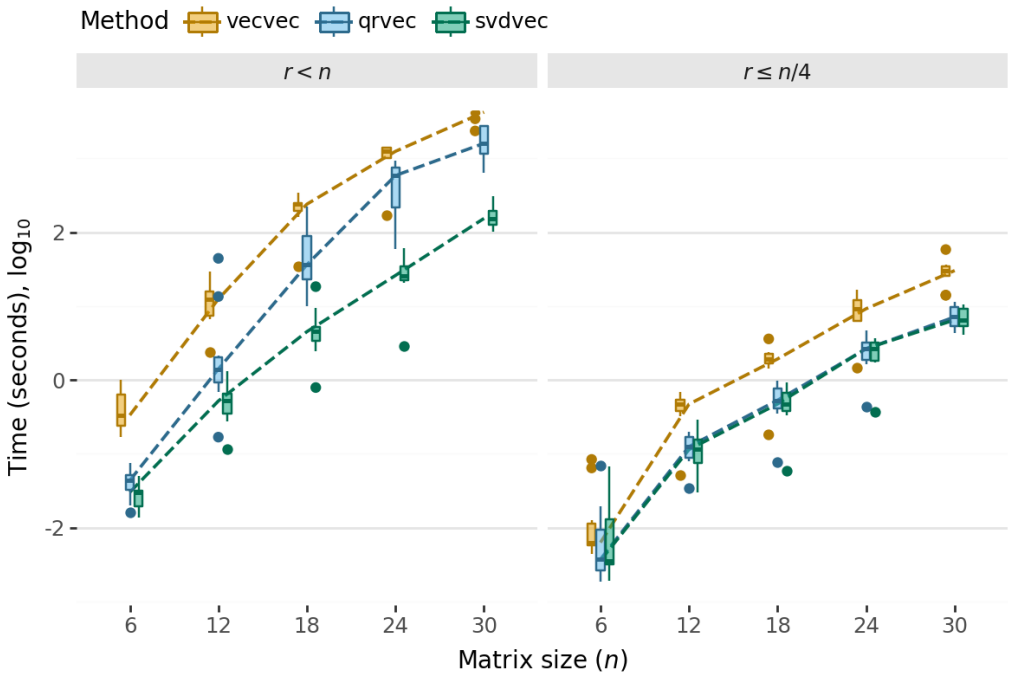


Figure 4.2: Solver time using different algorithms for solving for W. The time, on logarithmic scale, that algorithms 3.1.1, 3.2.1, and 3.2.2 take to find the nearest pencil with rank $r < n$ and $r \leq n/4$. For each matrix size, 10 random input pencils were used. The same initial guess for V was used for all methods.

Figure 4.3: Distance results of different algorithms for solving for W. The distance to the nearest pencil of rank $r < n$ and $r \leq n/4$ found by algorithms 3.1.1, 3.2.1, and 3.2.2. For each matrix size, 10 random input pencils were used. The same initial guess for V was used for all methods.

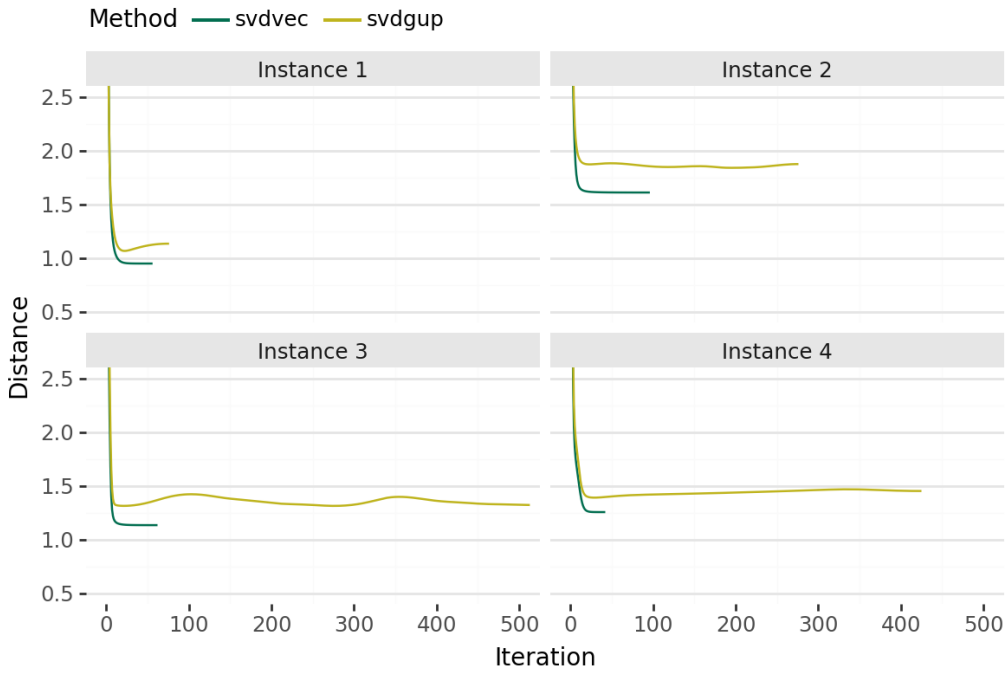


Figure 4.4: Distance per iteration using different algorithms for solving for V. Distance per iteration in four representative runs of algorithms 3.2.2 (svdvec) and 3.3.1 (svdgup), with random complex 20×20 input pencils and using same initial guess for V for all methods.

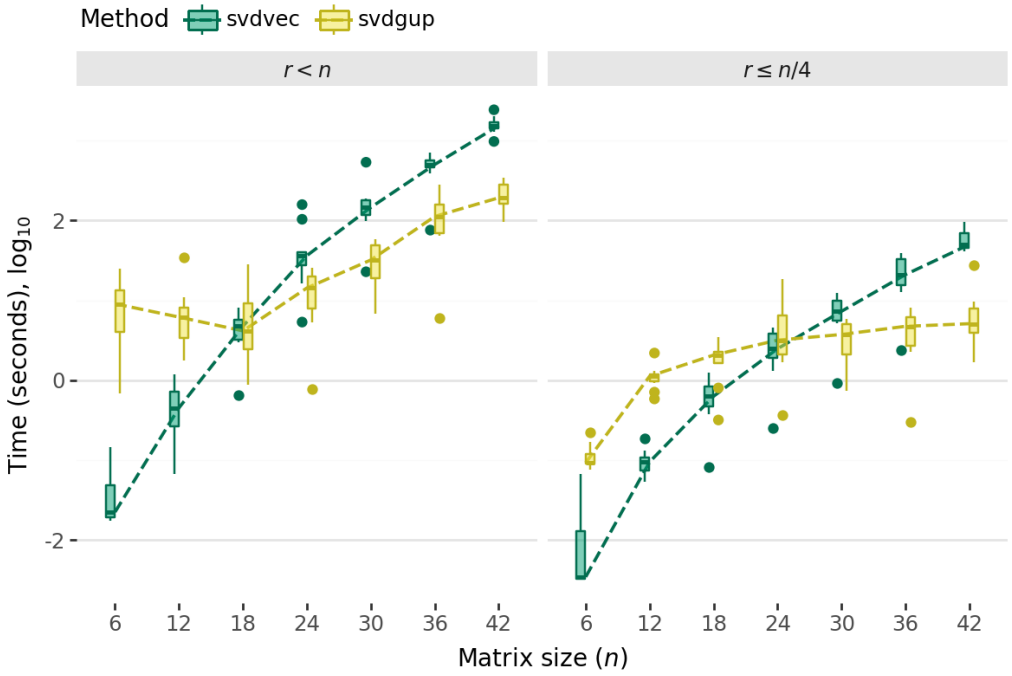


Figure 4.5: Solver time using different algorithms for solving for V. The time, on logarithmic scale, that algorithms 3.2.2 and 3.3.1 take to find the nearest pencil with rank $r < n$ and $r \leq n/4$. For each matrix size, 10 random input pencils were used. The same initial guess for V was used for all methods.

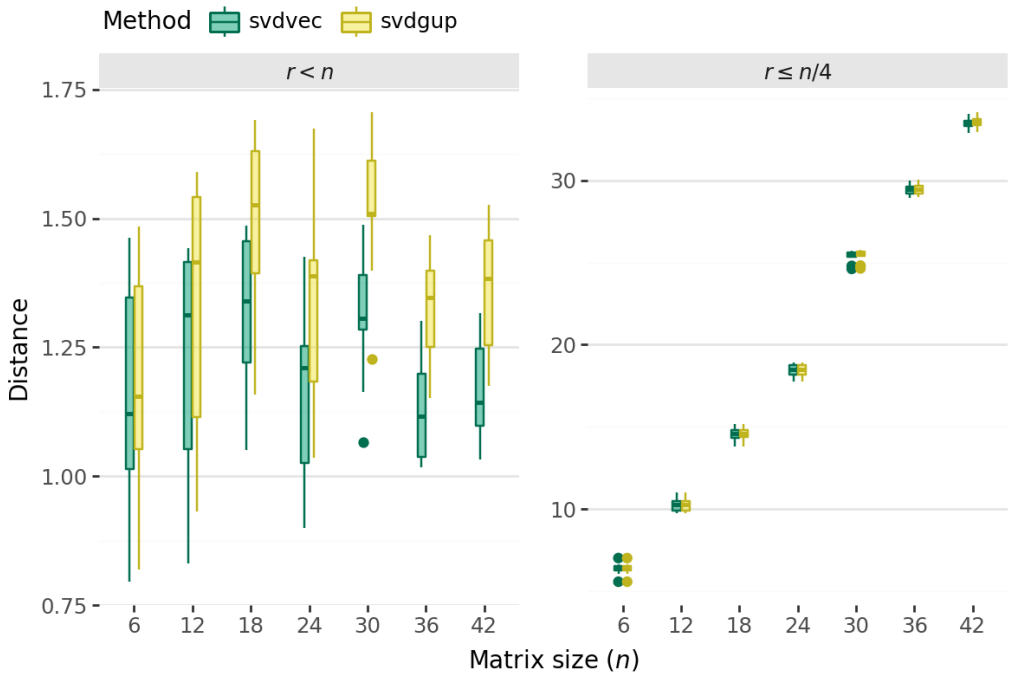


Figure 4.6: Distance results of different algorithms for solving for V. The distance to the nearest pencil of rank $r < n$ and $r \leq n/4$ found by algorithms 3.2.2 and 3.3.1. For each matrix size, 10 random input pencils were used. The same initial guess for V was used for all methods.

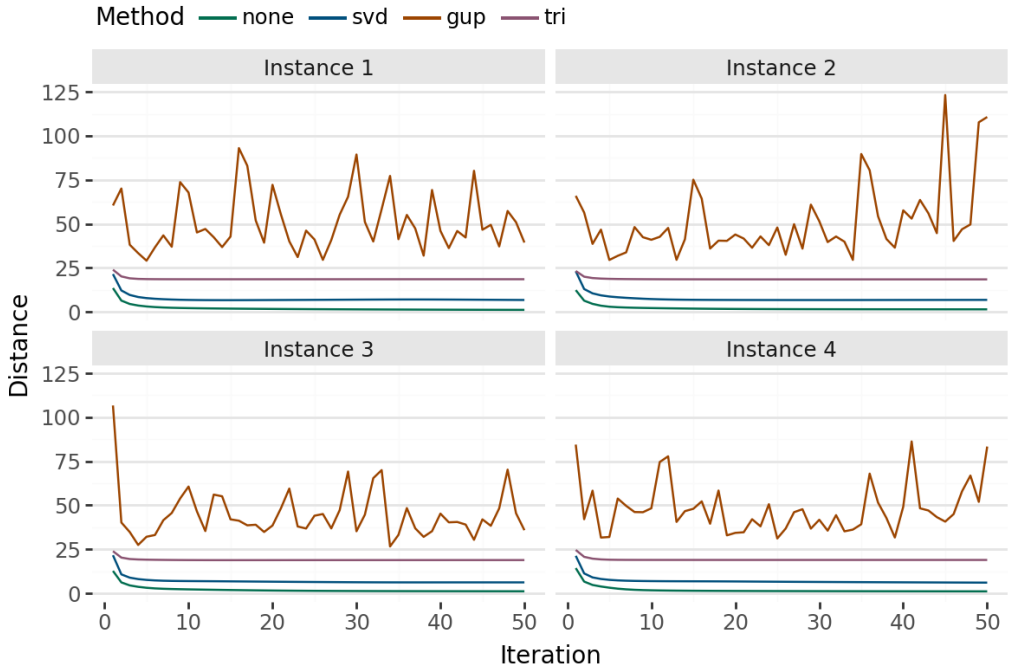


Figure 4.7: Distance per iteration of different versions of phase 1. Distance per iteration of four representative runs (four random input pencils) of algorithms 3.4.2, 3.4.3 and 3.4.4 on a 40×40 pencil, with algorithm 3.2.2 (svdvec) for reference. The same initial guess for V was used for all methods.

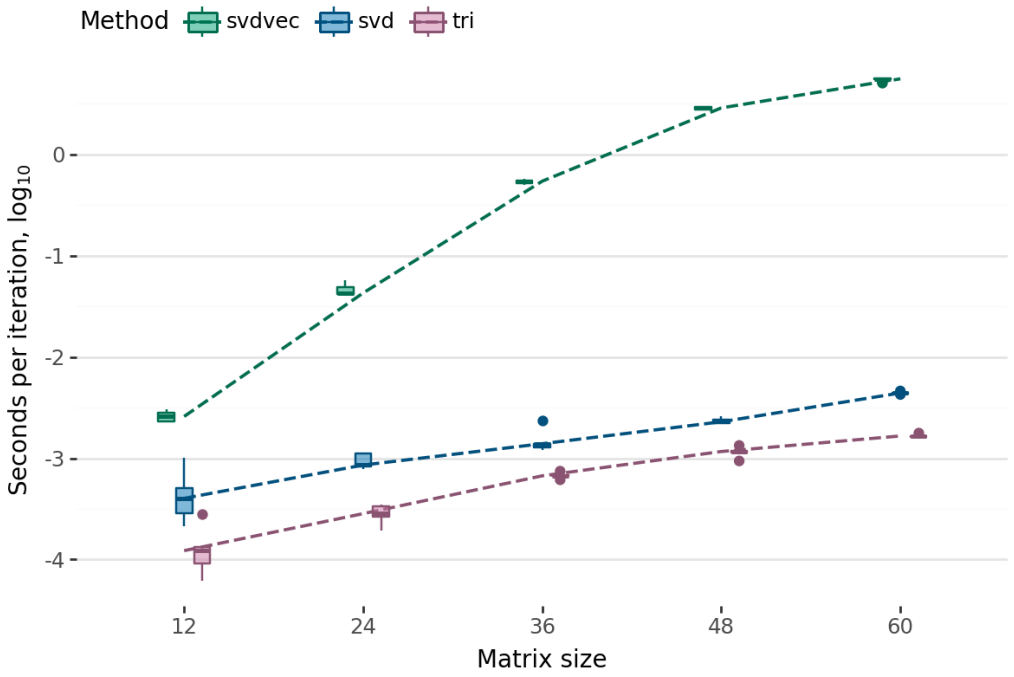


Figure 4.8: Time per iteration for different versions of phase 1. Time per iteration of algorithms 3.4.2 and 3.4.4, with algorithm 3.2.2 (svdvec) for reference. Five runs per matrix size. The same initial guess for V was used for all methods.

svdvec (algorithm 3.2.2). While algorithm 3.4.4 is faster per iteration than algorithm 3.4.2, both are much faster than svdvec. Seeing as svd (algorithm 3.4.2) yields a better starting guess, we will choose this method for phase 1.

Lastly, we want to know for which matrix sizes phase 1 could be useful, and for how long it should run. Figure 4.9 shows the total solver time for different tolerances ε_1 in the termination criterion (3.16) for phase 1, for different matrix sizes. Unfortunately, I do not have the resources to run the algorithms for larger pencils.

4.2 Comparison to State of the Art

Here, we will compare algorithm 3.2.2 (svdvec) to existing methods. To our knowledge, there are three methods that can handle this type of structured distance problems. One is called Riemann-Oracle [10], here sometimes abbreviated to RO. One we will call nearest singular Matrix Valued Function, or MVF, after the title of [9]. The last, which we will call nearest Common Kernel, or CK for short, [8], handles a slightly different type of problem; it can be used to find the distance to the nearest pencil whose coefficients have a common nontrivial kernel. Having a common kernel is a sufficient but not necessary condition for singularity of matrix pencils. The problem of finding the nearest common kernel is equivalent to the problem of finding the nearest singular pencil under certain circumstances, see section 4.2.1.3 for more details.

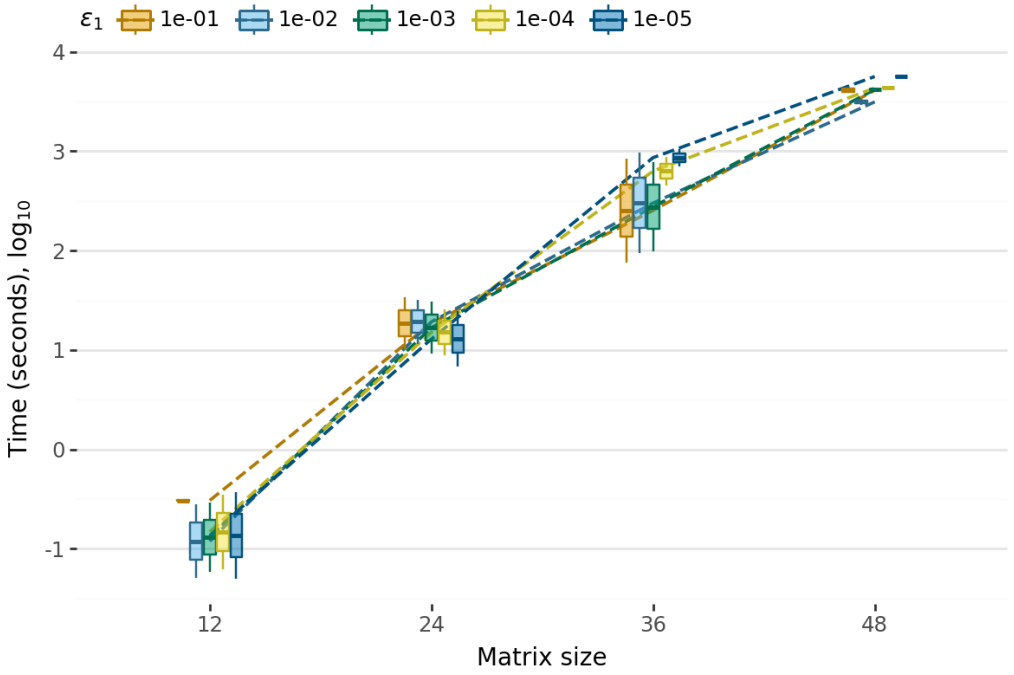


Figure 4.9: Solver time for different phase 1 termination criteria. The time algorithm 3.2.2 with phase 1 algorithm 3.4.2 takes to find the nearest singular matrix pencil, for different values of ε_1 in the termination criterion (3.16) for phase 1. The solver was run for five random matrix pencils per matrix size.

4.2.1 Background

In this section the theoretical background needed for using the methods from [10], [9], and [8] is presented. Because of the way that the structure is imposed in these methods, we only consider cases where the input pencil is also skew symmetric.

4.2.1.1 Riemann-Oracle

Given a matrix polynomial, the Riemann-Oracle method [10] involves representing this polynomial as a matrix, and finding the nearest singular matrix with a certain structure. We begin with a brief description of the method for finding the nearest structured singular matrix.

Let A be a matrix in $\mathbb{C}^{m\times n}$ and $\mathcal{S}$ be the space of perturbations with a certain structure. We wish to find the $\Delta A \in \mathcal{S}$ with smallest norm, such that $A + \Delta A$ is singular. Let $\left\{B^{(i)}\right\}_{i=1}^{b}$ be an orthonormal basis of $\mathcal{S}$ with respect to the Frobenius inner product, and write ΔA as

$$\Delta A = \sum_{i=1}^{b} \delta_i B^{(i)}.$$

If $(\operatorname{vec} B^{(1)}, \ldots, \operatorname{vec} B^{(b)})$ is orthogonal, then $\|\Delta A\|_F = \|\delta\|$, where $\delta := (\delta_1, \ldots, \delta_b)^T$. Henceforth, this is assumed to be true. The resulting matrix $A + \Delta A$ is singular if

and only if there exists a nonzero vector x such that $(A + \Delta A)x = 0$. The vector x may of course be normalized, yielding the problem to

$$\begin{aligned} \min_{\Delta A \in \mathcal{S}} \quad & \|\Delta A\|_F \\ \text{St} \quad & \exists x : \|x\| = 1,\ (A + \Delta A)x = 0. \end{aligned}$$

This is solved by splitting the problem into two nested optimization problems. The outer one is solved using an augmented Lagrangian method, while the inner one is solved using Riemannian optimization.

Now, consider the matrix polynomial $P(\lambda) = \lambda^0 A_0 + \cdots + \lambda^d A_d$ with coefficients in $\mathbb{C}^{n\times n}$. The polynomial P is singular if and only if any of the following equivalent statements hold [10][1].

i. The determinant of P is identically equal to zero.

ii. There exists a nonzero vector polynomial $x(\lambda)$ of degree $k \leq d(n-1)$ such that $P(\lambda)x(\lambda) \equiv 0$.

iii. There exists a nonzero vector polynomial $x(\lambda)$ of degree $k \leq \lfloor \frac{d(n-1)}{2} \rfloor$ such that either $P(\lambda)x(\lambda) \equiv 0$ or $x(\lambda)^H P(\lambda) \equiv 0$.

For a fixed k, define

$$\mathcal{T}_k(P) := \begin{pmatrix} A_0 & & \\ \vdots & \ddots & \\ A_d & \ddots & A_0 \\ & \ddots & \vdots \\ & & A_d \end{pmatrix} \in \mathbb{C}^{n(k+d+1)\times n(k+1)}.$$

Define also

$$\operatorname{vec} P(\lambda) := \begin{pmatrix} \operatorname{vec}(A_0) \\ \vdots \\ \operatorname{vec}(A_d) \end{pmatrix} \quad \text{and} \quad \operatorname{vec} x := \begin{pmatrix} x_0 \\ \vdots \\ x_k \end{pmatrix}.$$

Then, $\operatorname{vec}(P(\lambda)x(\lambda)) = \mathcal{T}_k(P) \operatorname{vec} x$, and thus

$$\operatorname{vec}(P(\lambda)x(\lambda)) = 0 \iff \mathcal{T}_k(P) \operatorname{vec} x = 0.$$

Hence, the problem of finding the nearest singular matrix polynomial can be solved using the method for finding the nearest singular matrix to $\mathcal{T}_k(P)$, with the same block Toeplitz structure[2] and, possibly, additional structure.

[1] For consistency with other sections, notation is different from the original article. In particular, the letters d and k are switched, so that d corresponds to the degree of the polynomial P, as opposed to the degree of the vector x.

[2] A Toeplitz matrix is a matrix where each diagonal is constant. That is, the element in position i, j is equal to the element in position $i + 1, j + 1$, if it exists.

Now, we will consider using this method to find the distance to singularity for a skew-symmetric matrix pencil. Consider the polynomial $P(\lambda) = A - \lambda B$ with A and B skew symmetric and in $\mathbb{C}^{n\times n}$. In the case of skew-symmetric pencils, we have that

$$x^H P = 0 \iff P^H x = 0 \iff P^T \overline{x} = 0 \iff$$
$$-P\overline{x} = 0 \iff P\overline{x} = 0$$

and, thus,

$$\exists x(\lambda) \not\equiv 0 : \deg(x) = k, P(\lambda)x(\lambda) = 0 \iff$$
$$\exists x(\lambda) \not\equiv 0 : \deg(x) = k, x(\lambda)^H P(\lambda) = 0.$$

Hence, statement iii in the list on page 28 reduces to

$$\exists x(\lambda) \not\equiv 0 : \deg(x) = k \leq \left\lfloor \frac{d(n-1)}{2} \right\rfloor \text{ and } P(\lambda)x(\lambda) \equiv 0.$$

Noting that the degree d of a matrix pencil is 1, we get $k \leq \lfloor \frac{n-1}{2} \rfloor$ when reformulating the problem of finding the nearest structured matrix pencil as a problem of finding the nearest structured matrix. Set $k = \lfloor \frac{n-1}{2} \rfloor$ and let

$$\widetilde{A} := \mathcal{T}_k(P) = \begin{pmatrix} A & & \\ -B & \ddots & \\ & \ddots & A \\ & & -B \end{pmatrix} \in \mathbb{C}^{n(k+2)\times n(k+1)}.$$

Also let $\widetilde{B}^{(i)} = \mathcal{T}_k(B^{(i)}(\lambda))$, where $\left\{B^{(i)}\right\}_{i=1}^{2b}$ is the basis of the space of complex skew-symmetric matrix pencils of size $n \times n$. That is,

$$B^{(1)}(\lambda) = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ -1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}, \quad B^{(2)}(\lambda) = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & 0 & 1 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ -1 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix},$$

and, where $b = \frac{n(n-1)}{2}$,

$$B^{(b)}(\lambda) = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & \cdots & 0 & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & 0 & 0 \\ 0 & \cdots & 0 & 0 & 1 \\ 0 & \cdots & 0 & -1 & 0 \end{pmatrix},$$

followed by

$$B^{(b+j)}(\lambda) = \lambda B^{(j)}(\lambda)$$

for $j \in \{1, ..., b\}$.

Then, use the method for finding the nearest structured singular matrix to the matrix $\widetilde{A}$, with $\mathcal{S}$ being the space spanned by $\{\widetilde{B}^{(i)}\}_{i=1}^{2b}$.

4.2.1.2 Nearest Singular Matrix Valued Function

The method described in [9] can be used for finding the nearest singular matrix valued function, to a given (non-singular) matrix valued function by perturbing the coefficients.

Consider the matrix valued function

$$F(\lambda) = \sum_{i=0}^{d} f_i(\lambda) A_i$$

with $A_i \in \mathbb{C}^{n\times n}$ and $f_i : \mathbb{C} \to \mathbb{C}$ entire, for $i = 0, \ldots, d$, and with $A_d \neq 0$ and $f_d \not\equiv 0$. Let

$$\mathbf{A} := \begin{pmatrix} A_0 \\ \vdots \\ A_d \end{pmatrix}$$

and define the norm $\|F(\lambda)\| := \|\mathbf{A}\|_F$. Denote by ΔF the perturbation

$$\Delta F(\lambda) = \sum_{i=0}^{d} f_i(\lambda) \Delta A_i$$

with $\Delta A_i \in \mathbb{C}^{n\times n}$, and define

$$\|\Delta F\| := \|\mathbf{\Delta A}\|_F, \quad \text{where} \quad \mathbf{\Delta A} := \begin{pmatrix} \Delta A_0 \\ \vdots \\ \Delta A_d \end{pmatrix}.$$

Recall that the matrix valued function $(F + \Delta F)(\lambda)$ is singular if and only if

$$\det(F + \Delta F)(\lambda) \equiv 0.$$

We denote the determinant $\det(F + \Delta F)(\lambda)$ by $f(\lambda)$ and note that if $f_0, \ldots f_d$ are entire, then so is f.

As previously mentioned, the objective is to find the smallest perturbation ΔF of F, such that $F + \Delta F$ is singular. That is, we wish to

$$\begin{aligned} &\min \|\Delta F\| \\ &\text{St } f(\lambda) \equiv 0. \end{aligned}$$

In order to apply the constraint $f(\lambda) \equiv 0$ numerically, it must be discretized. Since $f(\lambda)$ is entire, it is holomorphic in any subset of $\mathbb{C}$, for instance in $D := \{\lambda \in \mathbb{C} : |\lambda| \leq 1\}$. By [9, Theorem 3.2], the maximum of $|f|$ in D is obtained on the boundary ∂D of D. Thus, we want to find ΔF such that

$$\max_{\lambda \in \partial D} |f(\lambda)| = 0.$$

If the above holds, we have that $f(\lambda) \equiv 0$ on D and, by Taylor expansion, on all of $\mathbb{C}$. Further, we approximate $f(\lambda)$ by a polynomial interpolant $p(\lambda)$. If the degree of

$p(\lambda)$ is k, then it suffices to check that $p(\mu_j) = 0$ in $m \geq k$ distinct points $\mu_1, \ldots, \mu_m$. In this method, however, the number of evaluation points m is chosen in a more sophisticated manner; see [9] for more detail.

The (unstructured) minimization problem we have arrived at is the following.

$$\begin{aligned} &\min \|\Delta F\| \\ &\text{St } f(\mu_j) = 0 \quad \text{for} \quad j = 1, \ldots, m. \end{aligned}$$

In order to find the structured distance to singularity, let $\mathcal{S} \in \mathbb{C}^{(d+1)n \times n}$ be the space of structured perturbations $\mathbf{\Delta A}$. Then, the minimization problem reads

$$\begin{aligned} &\min_{\mathbf{\Delta A} \in \mathcal{S}} \|\Delta F\| \\ &\text{St } f(\mu_j) = 0 \quad \text{for} \quad j = 1, \ldots, m. \end{aligned} \tag{4.1}$$

The structure is imposed by an orthogonal projection $\Pi_{\mathcal{S}}$ of $\mathbf{\Delta A}$ onto $\mathcal{S}$, with respect to the Frobenius inner product.

This problem is solved using a two level iterative method. The inner iteration involves fixing the size of the perturbation and finding the direction which minimizes the smallest singular values of the perturbed function in each point μ_j. The outer iteration consists of fixing the direction of the perturbation and finding the smallest size of the perturbation such that the smallest singular values of the perturbed function in each point μ_j is zero.

An assumption [9, Assumption 4.9] used in the outer iteration is that the smallest singular value of $(F + \Delta F)(\mu_j)$ is simple and non-zero for $j = 1, \ldots, m$. In the case of skew-symmetric matrix pencils, the singular values come in pairs and hence the smallest singular value is only simple if n is odd, in which case it is zero. This may affect the performance of this method when used to find the nearest singular skew-symmetric matrix pencil.

In the implementation of this method, the input function $F(\lambda)$ is assumed to be normalized, in the sense that

$$\max_{\lambda \in \Xi} |\det F(\lambda)| = 1,$$

where $\Xi \subset \partial D$ is a discrete set of points. The points should be sufficiently many that, in practice,

$$\max_{\lambda \in \partial D} |\det F(\lambda)| = 1.$$

When using this method to find the nearest singular skew-symmetric matrix pencil to the pencil $A - \lambda B$, the normalization was done by choosing 1000 equidistant points $\lambda_1, \ldots, \lambda_{1000}$ on ∂D and setting

$$\begin{aligned} \alpha &= \max_{j \in \{1,\ldots,1000\}} |\det(A - \lambda_j B)|, \\ A_{\text{normalized}} &= \frac{1}{\alpha} A, \quad B_{\text{normalized}} = \frac{1}{\alpha} B. \end{aligned}$$

The function input was $A_0 := A_{\text{normalized}}$, $A_1 := B_{\text{normalized}}$, $f_0 := 1$, and $f_1 := -\lambda$. The orthogonal projection $\Pi_{\mathcal{S}}$ used was

$$\Pi_{\mathcal{S}} \begin{pmatrix} A_0 \\ A_1 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} A_0 - A_0^T \\ A_1 - A_1^T \end{pmatrix}. \tag{4.2}$$

To get the (not normalized) distance from $A - \lambda B$ to the nearest singular skew-symmetric matrix pencil, the distance output is multiplied by α [9, section 7].

4.2.1.3 Nearest Common Kernel

In [8], two methods are proposed, one for finding the nearest singular matrix polynomial, and one for finding the nearest matrix polynomial with coefficients with a common nontrivial kernel. The method presented in the previous section, (MVF, from [9]) is similar to the former, but extended from matrix polynomials to more general matrix valued functions, and further improved[3]. Therefore, we will consider only the latter here. We recall that having a common kernel is a sufficient but not necessary condition for a matrix pencil to be singular.

Consider the matrix polynomial $P(\lambda) = \lambda^0 A_0 + \cdots + \lambda^d A_d$ with coefficients in $\mathbb{C}^{n\times n}$. Its coefficients having a common nontrivial right kernel means that there exists a nonzero vector $x \in \mathbb{C}^n$ such that x belongs to the right nullspace of each coefficient, $x \in \mathcal{N}(A_i)$ for $i = 0, \ldots, d$.

Let $\mathbf{A}$ denote the coefficient block matrix corresponding to $P(\lambda)$,

$$\mathbf{A} := \begin{pmatrix} A_0 \\ \vdots \\ A_d \end{pmatrix}.$$

By [8, Lemma 5.3], the coefficients of $P(\lambda)$ have a common nontrivial kernel if and only if the smallest singular value of $\mathbf{A}$ is 0, that is,

$$\exists x \neq 0 : x \in \bigcap_{i=0}^{d} \mathcal{N}(A_i) \iff \sigma_{\min}(\mathbf{A}) = 0.$$

Denote by ΔP a perturbation of P, and by $\mathbf{\Delta A}$ the coefficient block matrix

$$\mathbf{\Delta A} = \begin{pmatrix} \Delta A_0 \\ \vdots \\ \Delta A_d \end{pmatrix}$$

corresponding to ΔP. By $\mathcal{S} \subseteq \mathbb{C}^{(d+1)n\times n}$, denote the space of perturbations with a certain structure. The minimization problem formulated in [8] reads

$$\begin{aligned} &\min_{\mathbf{\Delta A}\in\mathcal{S}} \|\mathbf{\Delta A}\|_F \\ &\text{St } \sigma_{\min}(\mathbf{A} + \mathbf{\Delta A}) = 0. \end{aligned}$$

[3] According to the authors of [8, 9], the number of evaluation points μ_j used is generally lower in [9] than in [8], which may make the former faster than the latter.

This is solved, similarly to (4.1), by nested iterations. The inner iteration consists of finding a unit norm matrix $\mathbf{\Delta}$ that minimizes $\sigma_{\min}(\mathbf{A} + \varepsilon\mathbf{\Delta})$, while the outer iteration consists of finding the smallest positive ε such that $\sigma_{\min}(\mathbf{A} + \varepsilon\mathbf{\Delta}) = 0$.

Now, consider using CK to find the nearest singular skew-symmetric pencil to a given (full rank) skew-symmetric pencil. In order to find a perturbation $\Delta\mathbf{A}$ in the space $\mathcal{S}$ corresponding to skew-symmetric pencils, we need to provide an orthogonal projection $\Pi_{\mathcal{S}}$ onto $\mathcal{S}$, with respect to the Frobenius inner product. Naturally, we use the same projection $\Pi_{\mathcal{S}}$ as described in equation 4.2.

As previously mentioned, the problem of finding the nearest skew-symmetric pencil with common kernel coincides with the problem of finding the nearest singular skew-symmetric pencil under certain circumstances. The skew-symmetric pencil $C - \lambda D$ has an $\mathcal{M}_0$ block in its SS-KCF if and only if there is a 0-degree vector in the minimal polynomial basis of its nullspace, that is,

$$\exists x \neq 0 \in \mathbb{C}^n : (C - \lambda D)x = 0 \ \ \forall\lambda.$$

This is equivalent to C and D having a common nonzero vector in their kernel,

$$\exists x \neq 0 \in \mathbb{C}^n : Cx = Dx = 0.$$

Thus, the task of finding the nearest skew-symmetric pencil with common kernel to a given pencil is equivalent to finding the nearest skew-symmetric pencil with an $\mathcal{M}_0$-block in its SS-KCF. The problem of finding the nearest singular skew-symmetric pencil coincides with the problem of finding the nearest skew-symmetric pencil with common kernel if and only if the nearest singular skew-symmetric pencil has an $\mathcal{M}_0$-block in its SS-KCF.

From theorem 2.1.2 we can see that no pencil with an $\mathcal{M}_0$-block in its SS-KCF is more generic than a pencil without an $\mathcal{M}_0$-block in its SS-KCF. That is, if the generic pencil whose closed orbit is the space we are minimizing over has an $\mathcal{M}_0$-block in its SS-KCF, the resulting pencil is guaranteed to also have one. From theorem 2.1.3, we can tell that this only happens when $\alpha = 0$. Moreover, we have to choose s to be as large as possible, since otherwise, the nearest pencil with common kernel may be different than the nearest pencil of rank $\leq 2s$. For an odd n and skew-symmetric input pencil, the distance to the nearest singular skew-symmetric pencil is 0, since the rank of a skew-symmetric pencil is always even. Thus, n must be even. We get that $s = \frac{n-2}{2}$ and

$$0 = \alpha = \left\lfloor \frac{s}{n-2s} \right\rfloor = \left\lfloor \frac{(n-2)/2}{n-(n-2)} \right\rfloor = \left\lfloor \frac{n-2}{4} \right\rfloor,$$

and thus $\frac{n-2}{4} < 1$, which is true for $n < 6$. Since in theorem 2.1.3, n is assumed to be greater than 2, and n must be even, we arrive at $n = 4$ and $s = 1$ (that is, the maximal rank $r = 2s = 2$) as the only case when the two problems coincide. Note that the nearest singular pencil to a pencil of any size may have a common kernel, but the only case when that is guaranteed to happen for *any* input pencil is when $n = 4$ and $r = 2$.

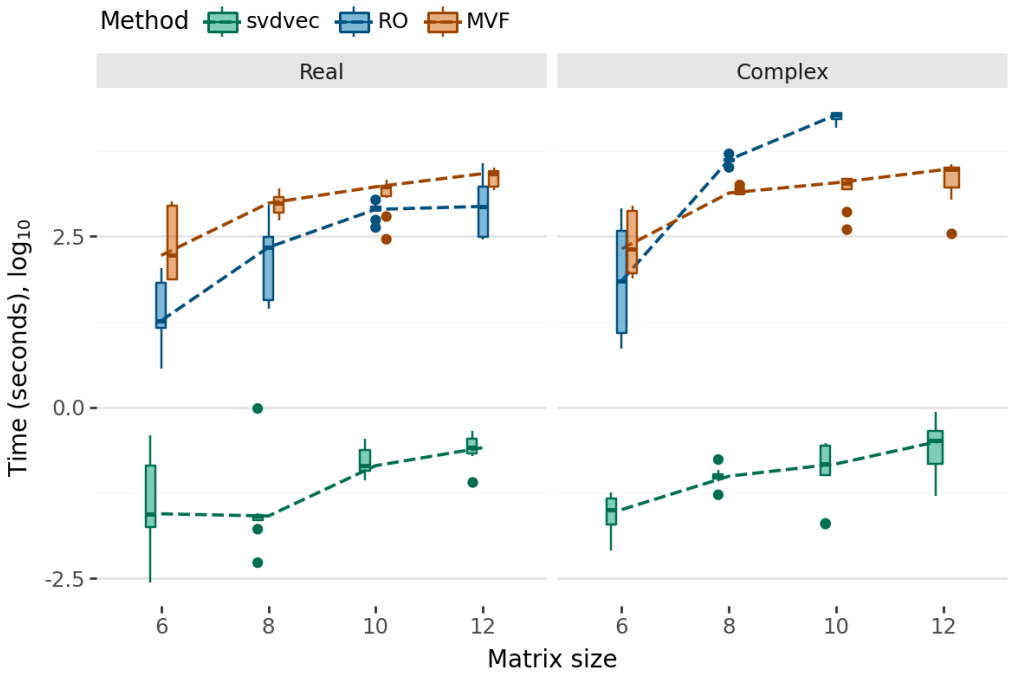


Figure 4.10: Solver time of our method and other existing methods. The time, on logarithmic scale, that algorithm 3.2.2, Riemann-Oracle [10], and nearest singular matrix valued function [9] take to find the nearest singular pencil. For each matrix size, 10 real and 10 complex random input pencils were used.

4.2.2 Results

When comparing to other methods, the values compared were time, distance to, and quality of singularity. The quality of the singularity was measured by the largest and smallest of the smallest singular values of the resulting pencil $C - \lambda D$ for $\lambda \in \Lambda$, where $\Lambda := \{x + iy : x, y \in \{-999, -777, \ldots, 999\}\}$. That is,

$$\max_{\lambda \in \Lambda} \sigma_{\min}(C - \lambda D) \quad \text{and} \quad \min_{\lambda \in \Lambda} \sigma_{\min}(C - \lambda D).$$

The methods are compared for 10 random real and complex input pencils per matrix size. In figure 4.10, the solver time for RO and MVF are compared to that of svdvec (algorithm 3.2.2). Their distance results are shown in figure 4.11, and their smallest singular values are shown in figure 4.12. The solver time, distance results and smallest singular values of CK are compared to those of svdvec in figure 4.13. As explained in section 4.2.1.3, the problem that our method solves only coincides with the problem that CK solves for matrix pencils of size 4×4.

The smallest singular value, $\sigma_{\min}$, of a matrix pencil is identically equal to zero (for all λ in $\mathbb{C}$, that is) if and only if the pencil is singular. In the case of numerical computations, a very small $\sigma_{\min}$ may be sufficient to call the pencil *numerically singular*. The smallest singular value of the result of each method, for a random 6×6 input pencil, is shown in figure 4.14.

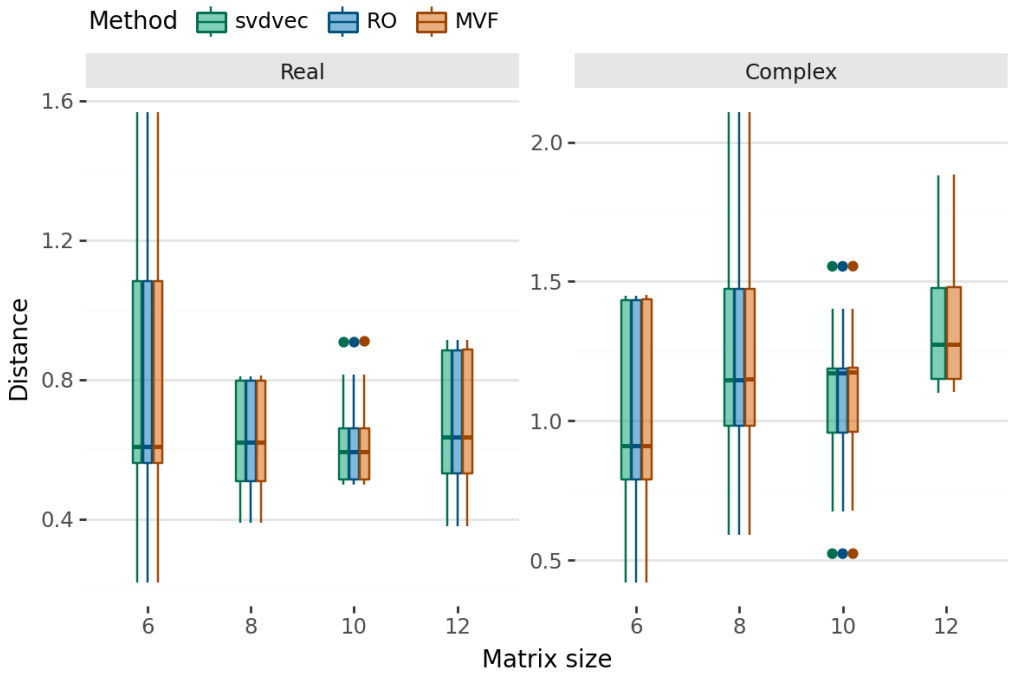


Figure 4.11: Distance results of our method and other existing methods. Distance to the nearest singular pencils found by algorithm 3.2.2, Riemann-Oracle [10], and nearest singular matrix valued function [9]. For each matrix size, 10 real and 10 complex random input pencils were used.

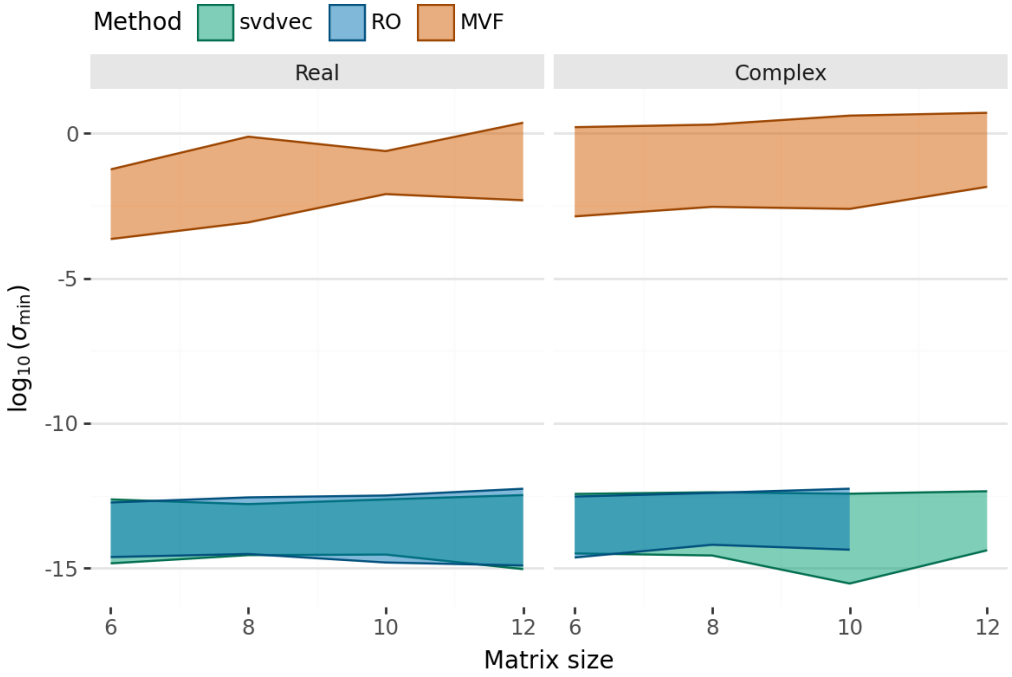
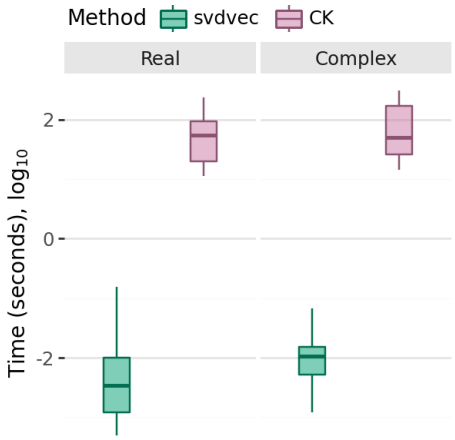
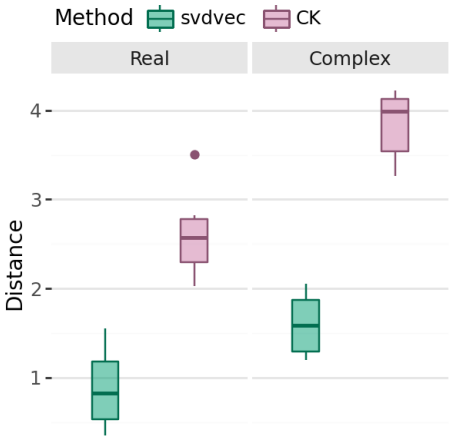


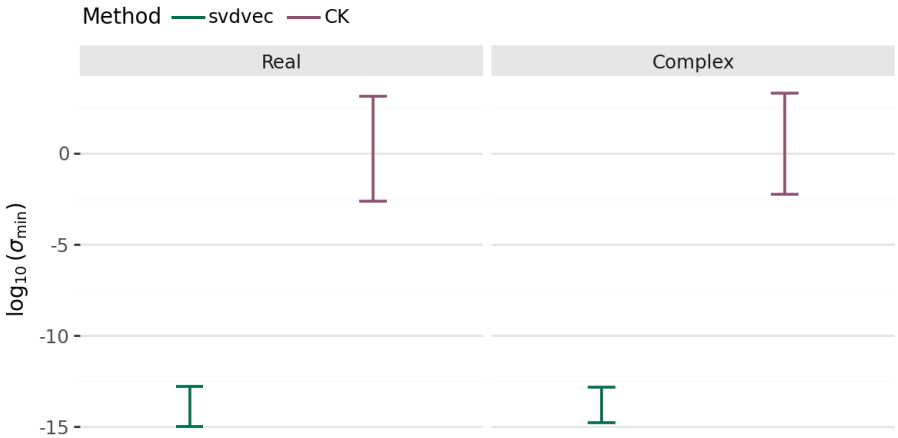
Figure 4.12: Quality of singularity of solution using our method and other existing methods. Smallest singular value of the pencils found by algorithm 3.2.2, Riemann-Oracle [10], and nearest singular matrix valued function [9], in Λ. For each matrix size, 10 real and 10 complex random input pencils were used.



(a) Time. The time, on logarithmic scale, that svdvec and CK take to find the nearest singular pencil.

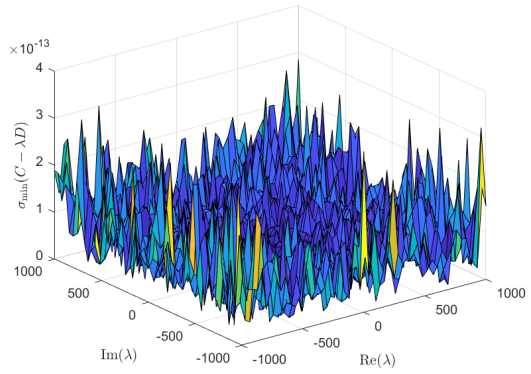


(b) Distance. Distance to the nearest singular pencils found by svdvec and CK.

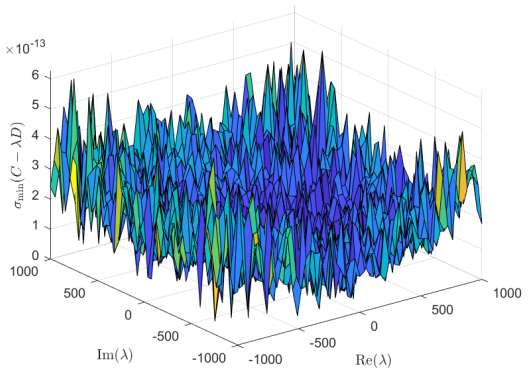


(c) Quality of singularity. Smallest singular value of the nearest singular pencils found by svdvec and CK, in the points $\lambda \in \{x+iy : x, y \in \{-999, -777, \ldots, 999\}\}$.

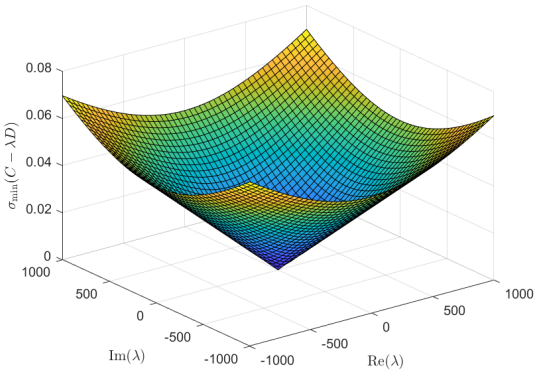
Figure 4.13: Time, distance, and singularity results of our method and an already existing method for finding the nearest common kernel. Algorithm 3.2.2 (svdvec) compared to the nearest common kernel [8] (CK). Results computed in the same way as in figures 4.10 – 4.12, but only for pencils of size 4×4, since this is the only size for which our method finds the nearest common kernel. For each figure, 10 real and 10 complex random input pencils were used.



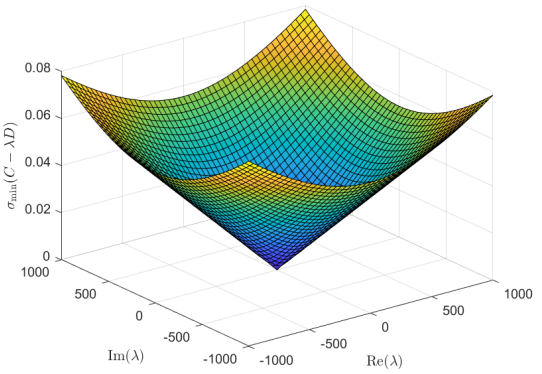
(a) svdvec. Smallest singular value in the order of 10^{-13}.



(b) RO. Smallest singular value in the order of 10^{-13}.



(c) MVF. Smallest singular value in the order of 10^{-2}.



(d) CK. Smallest singular value in the order of 10^{-2}.

Figure 4.14: The behavior of the smallest singular value of solutions using our method and other existing methods. The smallest singular value of the solution of methods svdvec (algorithm 3.2.2), RO [10], MVF [9], and CK [8], using a random 6×6 input pencil. The smallest singular value has been calculated on the square grid $\{x + yi : x, y \in \{-1000, -960, \ldots, 1000\}\}$

5

Discussion

In this chapter we will discuss the results presented in the previous chapter. As a tool for discussing these, we will first have a look at the time complexity of each of the presented algorithms. Then, in section 5.2, we will discuss the results plot by plot, and in section 5.3, some aspects are discussed in more detail.

5.1 Algorithm Complexity

The first, "naive", algorithm presented was algorithm 3.1.1, vecvec. It uses the Kronecker product and the vec-trick to get the system of equations on a form that can be solved using MATLAB's backslash. This approach creates a large matrix equation, which takes time to solve.

We recall that vecvec uses the Kronecker product and vec-trick to solve for both V and W. When solving for V using the vec-trick, we are solving the system

$$\begin{pmatrix} \widetilde{W}_0 \\ \widetilde{W}_1 \end{pmatrix} \operatorname{vec} V = \begin{pmatrix} \operatorname{vec} A \\ \operatorname{vec} B \end{pmatrix}$$

(see equation 3.6) using backslash in MATLAB. The size of the coefficient is $2n^2 \times ns$. Since it is non-square, MATLAB backslash uses a QR solver [15]. The most computationally expensive step in the QR solver [16] is the QR decomposition of the coefficient, which requires $O((2n^2)^3) = O(n^6)$ flops (floating point operations) [11]. When solving for W_0 or W_1 (equation 3.5), the coefficient matrix $\widetilde{V}$ is of the size $n^2 \times ns$, and hence the complexity of this step is also $O(n^6)$. As seen in figure 5.1, the time grows proportionally to n^6. The expected time being lower than the recorded median for some sizes is due to it being fitted for $n = 30$, where constant costs make up a relatively small proportion of the total time; this is also the reason the fitting was done for the largest n.

The complexity of algorithms 3.2.1, qrvec, and 3.2.2, svdvec, is, of course, also $O(n^6)$, since they use the same method to solve for V. However, in solving for W, the complexity drops from $O(n^6)$ to $O(n^3)$. This is because the most expensive step is the decomposition (QR or SVD), which is done for the $n \times s$ matrix V, and has a cost of at most $O(n^3)$ [11, 13]. In qrvec, MATLAB's backslash is only used with triangular coefficient matrices. While such systems are simpler than the general

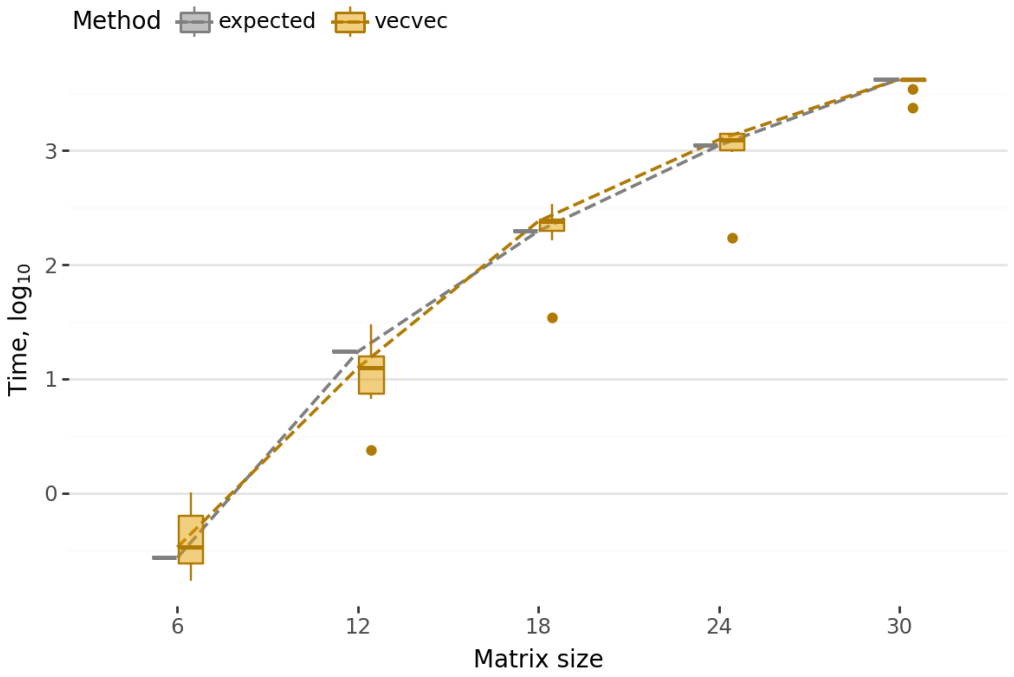


Figure 5.1: Solver time for vecvec compared to expected time. Expected time for vecvec (algorithm 3.1.1), if complexity is in the order of $O(n^6)$. The expected time is $t(n) = Cn^6$, where C is such that $t(30)$ agrees with the median time of vecvec for $n = 30$.

case, solving them still requires $O(n^3)$ flops[1]. In svdvec, backslash is only used for systems with diagonal coefficient matrices, which are solved in $O(n^2)$ flops[2].

Algorithm 3.3.1, svdgup, uses a different method to solve for V. First, it calculates the GUPTRI form of W_0 and W_1 using the function *pguptri*. This step has complexity $O(n^3)$, as reported in [6]. After the decomposition, the most expensive step is solving for X_1 in equation 3.14. The coefficient side in (3.14) is of the size $3s^2 \times 3s^2$. For square coefficients, MATLAB's backslash uses an LU solver. The complexity of the LU-factorization is in the order of $O((3s^2)^3) = O(s^6) = O(n^6)$ [11]. Hence, the solver time is expected to grow with n at the same rate as that of vecvec, qrvec and svdvec. This agrees with the results, see figure 4.5 where the time of svdgup grows similarly to that of svdvec. However, since $s < \frac{n}{2}$, svdgup can be expected to solve for V faster than svdvec for any given n. This is, however, not true; for small n the cost of calling GUPTRI makes svdgup slower than svdvec. Additionally, as seen in figure 4.4, svdgup generally uses more iterations than svdvec.

If we had used simultaneous SVD to solve for V, as mentioned in section 3.3, the complexity would probably have been similar to that of svdgup. Calculating the

[1] A matrix equation with an $n \times n$ triangular coefficient matrix and an $n \times n$ unknown matrix X requires one subtraction, one division and $n - i$ multiplications per element on row i in X to be solved. This sums to $\frac{1}{2}(n^3 + 3n^2)$ flops.

[2] A matrix equation with an $n \times n$ diagonal coefficient matrix and an $n \times n$ unknown matrix X requires only one division per element in X to be solved.

simultaneous SVD of W_0 and W_1 would likely take $O(n^3)$ time. After the change of variables we would have smaller equations, similar to what we get when using GUPTRI but (hopefully) fully decoupled. If the blocks in the simultaneous SVD were of size $\mu \times \mu$, the decoupled equations would be solved in $O(\mu^6)$ time using the Kronecker product and the vec-trick. If the block size depended linearly on n, as it does for GUPTRI, the complexity would be $O(n^6)$, as for GUPTRI. However, if the block size did not increase linearly with n, the order of the complexity could have been reduced. Moreover, even if μ depended linearly on n, an algorithm using the simultaneous SVD might be faster than svdgup if $\mu < s$.

The phase 1 algorithms 3.4.2, svd, and 3.4.4, tri, both have the same complexity. When solving for both V and W in svd, the most expensive step is calculating the SVD. In tri, the decomposition done in MATLAB's backslash function is the most expensive step both when solving for both V and W. Both of these calculations are done in $O(n^3)$ time, as previously mentioned. However, while tri only uses backslash three times per iteration, svd uses four calculations of the SVD. That explains why svd is slower than tri, as seen in figure 4.8.

5.2 Interpretation of Results

Figures 4.1 and 4.4 show that all main algorithms except svdgup (algorithm 3.3.1), that is, algorithms 3.1.1 (vecvec), 3.2.1 (qrvec), and 3.2.2 (svdvec), typically yield a distance that decreases in each iteration. As seen in figure 4.4, the distance when using svdgup does not only fluctuate, but it also never gets as small as when using svdvec. This is due to X_2 being determined from equations 3.12b and 3.12e only, and then used in equation 3.12f when determining X_1.

In figure 4.2, we saw that vecvec is slower than qrvec and svdvec. While qrvec and svdvec use matrix decompositions to solve for W, vecvec uses the Kronecker product. As explained above, in section 5.1, the size of the equations increase greatly when the Kronecker product is used, why the (time) complexity of solving the equations is relatively large. As also explained above, the complexity of certain steps of svdvec is lower than that of qrvec.

Figure 4.3 shows that the accuracy of vecvec, and to a some degree qrvec as well, is worse than that of svdvec. This is likely due to the maximal number of iterations N being reached, as seen in figure 4.1. We can see that svdvec usually converges faster than qrvec, and qrvec faster than vecvec. As seen in figure 5.2, the relative error in A when calculating W_0 using the vec-trick is higher than when using the QR- and singular value decompositions. Why svdvec converges faster than qrvec, however, is not obvious. It may depend on the choices made for X_1 in (3.7a) and (3.11), respectively, namely that X_1 is chosen to be upper triangular in qrvec and skew-symmetric in svdvec. Naturally, there are mutliple equally good solutions for W_0 and W_1, and it seems that svdvec finds a W_0 and W_1 that "fit" better together in the whole system (3.1).

Moving on to figure 4.5, we could tell that svdgup (algorithm 3.3.1) was faster than

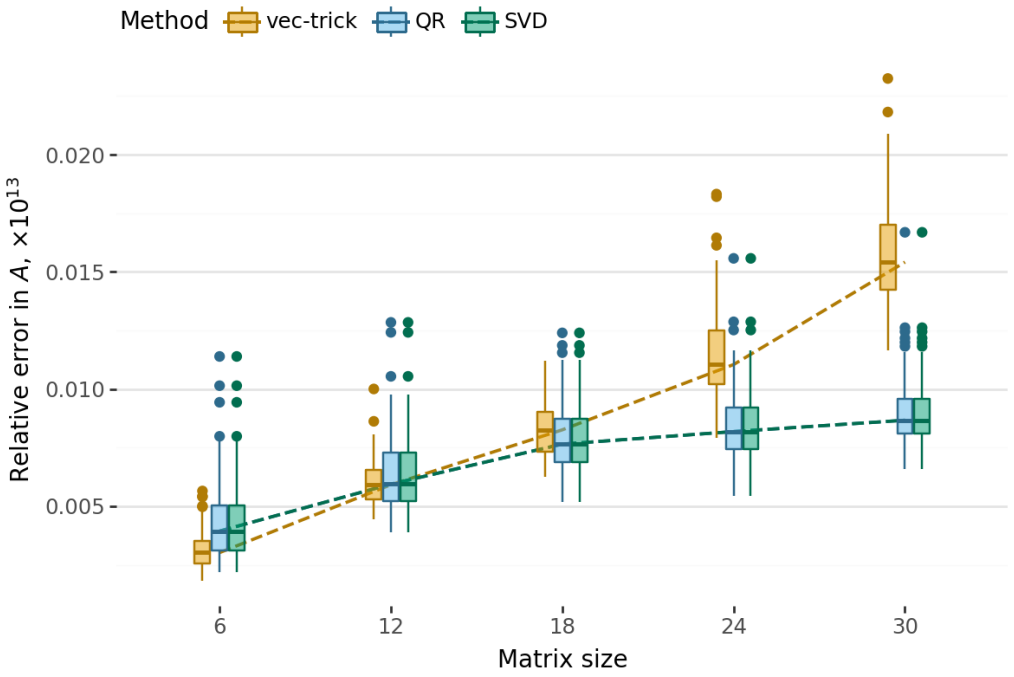


Figure 5.2: Error when calculating W_0 using QR-decomposition, SVD and the vec-trick. The relative error in A when calculating W_0 using the vec-trick (as in vecvec), QR-factorization (as in qrvec), and SVD (as in svdvec), for 100 random skew-symmetric matrix pencils per size.

svdvec for pencils larger than 20×20. This is due svdgup creating smaller equations, such that they do not grow as much when the Kronecker product is applied. For lower rank results ($r \leq \frac{n}{4}$), the resulting equations are smaller than those for $r < n$, since the number of columns in W and V is $s = \lfloor \frac{r}{2} \rfloor$. Thus, all equations are cheaper to solve, and the initial cost of calling GUPTRI in svdgup is not compensated for until $n \approx 24$. Regarding the distance results in figure 4.6, we saw that svdgup is less accurate than svdvec. It is worth noting that while the larger span of the y-axis makes it difficult to see, the difference is similar in the $r \leq \frac{n}{4}$ case. As mentioned earlier, svdgup is expected to find an upper bound rather than the optimal solution.

Figure 4.7 shows the distance per iteration when using the phase 1 algorithms 3.4.2 (svd), 3.4.3 (gup), and 3.4.4 (tri). We noticed that gup does not behave as expected; the distance does not decrease over time. This is most likely because not enough equations are used in the computation, i.e., the approximation being too rough. We also noticed that tri does not produce as good of an estimate as svd does; this is because it solves other equations (see equation 3.15). Then, in figure 4.8, we saw that tri is faster than svd, which in turn is faster than svdvec. As explained in the previous section, tri and svd have complexity of lower order than svdvec, and tri uses fewer expensive operations per iteration than svd. Lastly, figure 4.9 is less informative. We would need data from runs with larger pencils to determine what ε_1 in the termination criterion (3.16) for phase 1 should be, if used at all.

While RO and MVF are slower than svdvec for pencils of size 12×12 or smaller,

as seen in figure 4.10, it is possible that they scale better with the matrix size. Unfortunately, I do not have the resources to investigate this question further. The results reported in [10] about the Riemann-Oracle method show much shorter running times, albeit for different examples and on a different machine. The distance results in figure 4.11 are almost identical for the three methods, for these small pencils. This could indicate that all methods found the global minimum.

From figures 4.12 and 4.14, we saw that the MVF and CK results were not "as singular" as the results from RO and svdvec. The authors state in [9] that MVF can be used to find *numerically singular* polynomials (or matrix valued functions), meaning that the smallest singular value is small but not necessarily zero. Such results may well be useful in certain applications, especially if λ is known to be close to 0. Moreover, the assumption that the smallest singular value of the perturbed function is simple does not hold for skew-symmetric pencils (with an even number of rows and columns), which may affect the result. In [8], a similar note about numerical singularity of the result of CK is made.

It is also worth noting that some changes seem to have been made to the MVF code that I downloaded from GitHub since the article [9] was published. Among other things, some tolerances that are described as constant in the paper are updated in each iteration while running the GitHub code. For all results reported here, I have used the code from GitHub.

5.3 Further Discussion

As mentioned a few times already, svdgup (algorithm 3.3.1) does not find the best solution. In figure 5.3, we can see that the distance increases drastically when switching to svdgup from svdvec (algorithm 3.2.2). When solving for V in svdgup, we do not (necessarily) get best the least squares solution, as explained in section 3.3.1. If the algorithm is adjusted so that we solve for X_1 and X_2 in (3.12) together, the behavior (in terms of distance per iteration) is equal to that of svdvec. However, by doing so, we lose the effects on speed.

The usefulness of phase 1 has not been established. Since the distance decreases rapidly in the first few iterations, as seen in figures 4.1 and 4.4, phase 1 may not be useful even for large pencils. As seen in figure 5.4, the starting guess has some impact on the number of iterations needed, and therefore also on the time. However, the nearest starting guess (in terms of the Frobenius norm) does not necessarily correspond to the smallest number of iterations. Thus, a phase 1 algorithm that quickly improves the starting guess should reduce the runtime, but it is not obvious what a good starting guess is.

From figure 5.4, together with figure 4.11, we can also tell that svdvec seems to find a global minimum. If it were to get stuck in a local minimum, it would be very unlikely that it found the same local minimum for five different initial values, and it would also be unlikely that both RO and MVF had the same distance result.

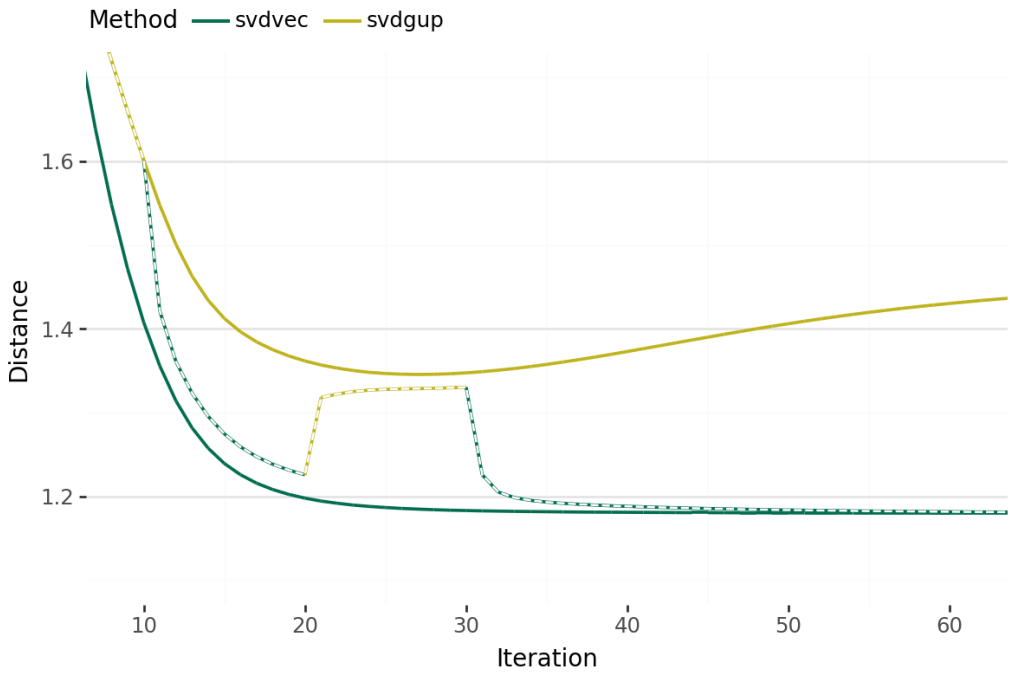
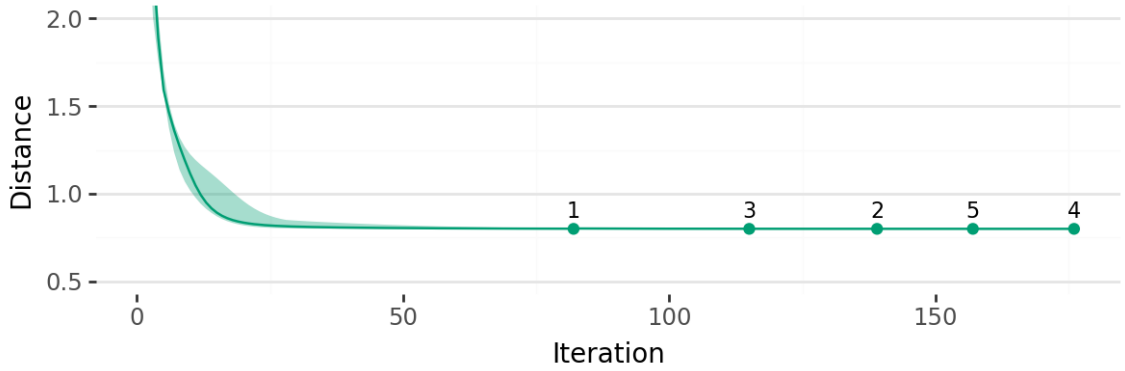
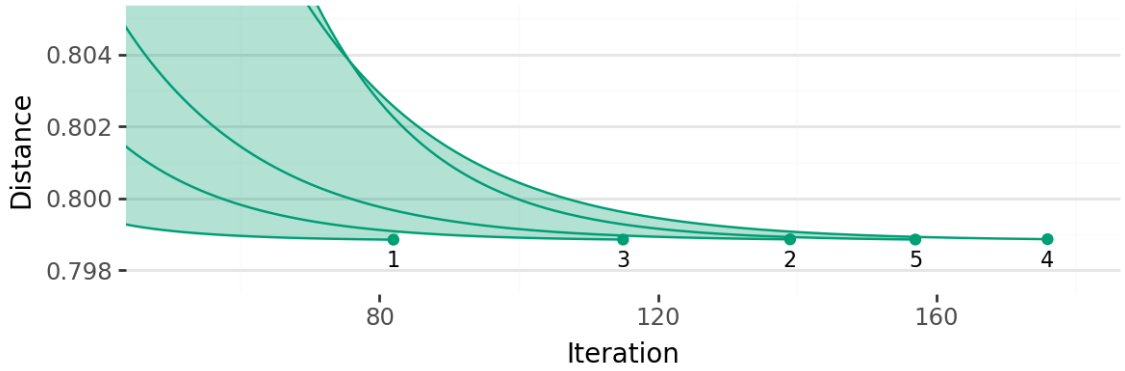


Figure 5.3: Distance per iteration when switching between svdgup and svdvec. The dashed line represents one run, where the first 10 iterations were done using svdgup (algorithm 3.3.1), followed by 10 using svdvec (algorithm 3.2.2), 10 more using svdgup and then svdvec until termination.



(a) Overall behavior. The line corresponds to the median distance.



(b) Zoomed in. The lines correspond to each starting guess.

Figure 5.4: Distance per iteration for different starting guesses. Distance per iteration for algorithm 3.2.2 (svdvec) using different initial guesses for V. The points mark where each run ended and are numbered by the distance after the first iteration; the point marked with 1 shows where the run with the smallest distance after the first iteration ended, etc.

6

Conclusions

In this thesis, four algorithms for finding the nearest skew-symmetric matrix pencil of a given maximal rank were presented. The skew-symmetric Kronecker canonical form and its parametrization were used to express a generic skew-symmetric pencil of rank at most r (given that $2 \leq r < n$ for pencils of size $n \times n$). A rank-1 decomposition of skew-symmetric matrix pencils provided a parametrization of the space of rank $\leq r$ skew-symmetric pencils. Using this parametrization, the task of finding the nearest skew-symmetric matrix pencil of rank $\leq r$ to a given (input) pencil was formulated as a minimization problem. Four algorithms for solving it were presented. Algorithm 3.1.1, vecvec, makes use of the Kronecker product and the so called vec-trick. Algorithm 3.2.1, qrvec, and 3.2.2, svdvec, also use the QR- and singular value decomposition, respectively. Algorithm 3.3.1, svdgup, uses the SVD and the generalized upper-triangular (GUPTRI) form [5], in addition to the Kronecker product and vec-trick, to find an upper bound of the distance. The latter three of these algorithms require the input pencil to be skew-symmetric.

Out of these algorithms, algorithm 3.2.2, svdvec, performed best in terms of accuracy. It also performed best in speed for pencils smaller than 20×20. For larger pencils, algorithm 3.3.1, svdgup, proved to be faster.

Further, the idea of using a "phase 1" algorithm to find a better starting guess in relatively short time was presented. The benefit of phase 1 could not be established due to limited resources; for the matrix sizes we were able to run the algorithms with, a random starting guess sufficed.

For small pencils, svdvec turned out to be faster than already existing methods. It was also equal or better in terms of accuracy, finding pencils with smallest singular value close to zero. However, the other methods can solve a much larger class of problems. For pencils larger than 12×12, we have not had the resources to compare.

While the algorithms presented here performed well compared to existing methods, further improvements could be done. Some ideas for future work are presented in the next section.

6.1 Future Work

While svdgup (algorithm 3.3.1) was faster than the other algorithms, it was less accurate. As discussed in section 5.2, this was due to the equations not being fully decoupled. It is possible that using the simultaneous SVD, as mentioned in sections 3.3 and 5.2, instead of the GUPTRI form would provide both faster an more accurate results.

It is possible that the time of qrvec (algorithm 3.2.1) and svdvec (algorithm 3.2.2) could be improved by a better assumption on X_1 in equations 3.7a and 3.11 respectively.

As for phase 1, it would be interesting to see whether it actually reduces the time for large matrices. With more resources, one could investigate which sizes it is useful for, if any, and what a reasonable default value for ε_1 in its termination criteria (3.16) could be.

Finally, the possibility of extending this method to skew-symmetric matrix polynomials of degree > 1 could be explored.

Bibliography

[1] Chun-Yueh Chiang, Eric King-Wah Chu, and Wen-Wei Lin. "On the (star)-Sylvester equation AX±X(star)B(star)=C". In: *Applied Mathematics and Computation* 218.17 (2012), pp. 8393–8407. ISSN: 0096-3003. DOI: 10.1016/j.amc.2012.01.065.

[2] Sweta Das and Andrii Dmytryshyn. *Minimal degenerations of orbits of skew-symmetric matrix pencils*. Submitted. 2025.

[3] Sweta Das et al. *Computing nearest rank deficient matrix polynomials using generic sets*. In progress. 2025.

[4] Fernando De Terán, Christian Mehl, and Volker Mehrmann. "Low-Rank Perturbation of Regular Matrix Pencils with Symmetry Structures". In: *Foundations of Computational Mathematics* 22.1 (Mar. 2021), pp. 257–311. ISSN: 1615-3383. DOI: 10.1007/s10208-021-09500-4.

[5] James Demmel and Bo Kågström. "The generalized Schur decomposition of an arbitrary pencil $A - \lambda B$ – robust software with error bounds and applications. Part I: theory and algorithms". In: *ACM Transactions on Mathematical Software* 19.2 (June 1993), pp. 160–174. ISSN: 1557-7295. DOI: 10.1145/152613.152615.

[6] James Demmel and Bo Kågström. "The generalized Schur decomposition of an arbitrary pencil $A - \lambda B$ – robust software with error bounds and applications. Part II: software and applications". In: *ACM Transactions on Mathematical Software* 19.2 (June 1993), pp. 175–201. ISSN: 1557-7295. DOI: 10.1145/152613.152616.

[7] Andrii Dmytryshyn and Froilán M. Dopico. "Generic skew-symmetric matrix polynomials with fixed rank and fixed odd grade". In: *Linear Algebra and its Applications* 536 (2018), pp. 1–18. ISSN: 0024-3795. DOI: https://doi.org/10.1016/j.laa.2017.09.006.

[8] Miryam Gnazzo and Nicola Guglielmi. *Approximating the closest structured singular matrix polynomial*. 2023. DOI: 10.48550/ARXIV.2301.06335.

[9] Miryam Gnazzo and Nicola Guglielmi. *On the numerical approximation of the distance to singularity for matrix-valued functions*. 2023. DOI: 10.48550/ARXIV.2309.01220.

[10] Miryam Gnazzo et al. *Riemann-Oracle: A general-purpose Riemannian optimizer to solve nearness problems in matrix theory*. 2024. DOI: 10.48550/ARXIV.2407.03957.

[11] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Second. Society for Industrial and Applied Mathematics, 2002. DOI: 10.1137/1.9780898718027.

[12] Kjell Holmåker and Ivar Gustafsson. *Linjär algebra: Fortsättningskurs*. Liber, 2016.

[13] Xiaocan Li, Shuo Wang, and Yinghao Cai. *Tutorial: Complexity analysis of Singular Value Decomposition and its variants*. 2019. eprint: arXiv:1906.12085.

[14] Takanori Maehara and Kazuo Murota. "Simultaneous singular value decomposition". In: *Linear Algebra and its Applications* 435.1 (2011), pp. 106–116. ISSN: 0024-3795. DOI: https://doi.org/10.1016/j.laa.2011.01.007.

[15] MathWorks. *mldivide, *. https://se.mathworks.com/help/matlab/ref/double.mldivide.html. [Accessed 22-05-2025].

[16] MathWorks. *QR Solver*. https://se.mathworks.com/help/dsp/ref/qrsolver.html. [Accessed 22-05-2025].

[17] Wolfram MathWorld. *Metric*. https://mathworld.wolfram.com/Metric.html. [Accessed 22-01-2025].

[18] Wikipedia. *Low-rank approximation*. https://en.wikipedia.org/wiki/Low-rank_approximation. [Accessed 26-06-2025].

A

Appendix

This appendix contains the final implementation of the algorithms presented in the thesis. The main method, which is called by the user, is shown in listing A.1. This method includes the implementation of algorithm 3.4.2 (svd). Listings A.2 - A.4 show the implementation of algorithms 3.1.1 (vecvec), 3.2.2 (svdvec) and 3.3.1 (svdgup), which are called by the main method. The code can also be found on GitHub, `https://github.com/rakeljh/nearest_singular_skew-symmetric_pencil`.

Listing A.1: Main method. Uses one of the algorithms 3.1.1, 3.2.2, and 3.3.1 depending on input. If specified, uses the phase 1 algorithm 3.4.2.

```
function [distance, C, D] = dist_to_sing_ss_pencil(A, B,
   r, options)
%dist_to_sing_ss_pencil Finds nearest skew-symmetric
   matrix pencil C-xD of
% rank <= r, with r in [2,n), to the n by n matrix pencil
    A-xB, using the
% Frobenius norm.
%
%   Input:
%       A                   :   First coefficient of
   matrix pencil A-xB.
%       B                   :   Second coefficient of
   matrix pencil A-xB.
%       r                   :   Rank of matrix to search
   for, in [2,n).
%       max_iter (optional) :   Maximum number of
   iterations, default 3000.
%       epsilon (optional)  :   Sensitivity of
   termination criteria,
%                               default 10^-7.
%       u_bound (optional)  :   If true, returns an upper
    bound in less
%                               time. Requires GUPTRI (
   MCS Toolbox) in
```

```
%                                   MATLAB path. Default
   false. Recommended
%                                   only for pencils of size
   20x20 or larger.
%       phase_1 (optional)   :      If true, improves the
   starting guess by
%                                   cheap iterations before
   entering the main
%                                   algorithm. ONLY for skew-
   symmetric pencils.
%                                   Default false.
%       epsilon_1 (optional):       Sensitivity of
   termination criteria for
%                                   phase 1. Default 10^-3.
%
%   Output:
%       distance     :    Distance between matrix pencils A
   -xB and C-xD, in
%                         the sense of (|A-C|^2 + |B-D|^2)
   ^(1/2) using the
%                         Frobenius norm.
%       C            :    First coefficient of matrix
   pencil C-xD.
%       D            :    Second coefficient of matrix
   pencil C-xD.

    arguments
        A double {mustBeNumeric, mustBeSquare(A)};
        B double {mustBeNumeric, mustBeSquare(B),
           mustBeEqualSize(A,B)};
        r double {mustBeNonnegative, mustBeWithin(r,A)} =
            0;
        options.max_iter double {mustBeNonnegative} =
           3000;
        options.epsilon double {mustBeNonnegative} =
           10^-7;
        options.u_bound logical = false;
        options.phase_1 logical = false;
        options.epsilon_1 double = 10^-3;
    end

    % Unpack options
    N = options.max_iter;
    eps = options.epsilon;
    GUPTRI = options.u_bound;
    ph_1 = options.phase_1;
```

```
n = length(A);
s = floor(r/2);
V = rand(n,s,like=[A;B]);

if ph_1
    if isequal(A,-A.') && isequal(B,-B.')
        term_crit_ph1 = options.epsilon_1 * norm([A;B
           ],"fro");

        prevdist = NaN;
        for i=1:100 % max iter for phase 1 is 100
            W0 = svd_solver(V,A);
            V = svd_solver(W0,-A);
            W1 = svd_solver(V,B);
            V = svd_solver(W1,-B);

            dist = norm([A;B] - [V*W0.'-W0*V.'; V*W1
               .'-W1*V.'], "fro");

            if abs(prevdist - dist) < term_crit_ph1
                break;
            end

            prevdist = dist;
        end
    else
        warning("Phase 1 not compatible with non skew
           -symmetric" + ...
            "pencils, phase 1 not used.");
    end
end

% Check skew symmetry and choose method.
if isequal(A,-A.') && isequal(B,-B.')
    if GUPTRI
        warning("off") % Produces warnings about rank
            deficiency.
        [distance, C, D] = svdgup(A, B, r, V, ...
            max_iter=N, epsilon=eps);
        warning("on")
    else
        [distance, C, D] = svdvec(A, B, r, V, ...
            max_iter=N, epsilon=eps);
    end
else
```

```
        warning("Input matrices not skew symmetric, " +
           ...
            "performance will be affected.")
        warning("off") % Produces warnings about rank
           deficiency.
        [distance, C, D] = vecvec(A, B, r, V,...
            max_iter=N, epsilon=eps);
        warning("on")
    end
end

function mustBeSquare(M)
    [m,n] = size(M);
    if m ~= n
        error("Matrices must be square.")
    end
end

function mustBeEqualSize(M,N)
    if ~isequal(size(M),size(N))
        error("Matrices must be of equal size.")
    end
    if any([size(M),size(N)] <= 2)
        error("Size of input pencil must be greater than
           2 by 2.")
    end
end

function mustBeWithin(r,M)
    if 2 > r || r >= length(M)
        error("Rank r must be in [2,n).")
    end
end

function X = svd_solver(V,C)
%svd_solver returns X as the solution to VX.' - XV.' = C.

    % V and X have the same dim, n by s
    [n,s] = size(V);

    % SVD of V
    [R,S,T] = svd(V);
    S1 = S(1:s,:);
    E = R'*C*conj(R);
    E11 = E(1:s,1:s);
    E12 = E(1:s,s+1:n);
```

```
    % Solve for Y = T'*X.'*conj(R) (s by n)
    Y1 = zeros(s);
    for k = 1:s
        for l = 1:s
            Y1(k,l) = E11(k,l)/(S1(k,k)+S1(l,l));
        end
    end

    Y2 = S1\E12;

    Y = [Y1 Y2];

    % Solve for X
    X = R*Y.'*T.';

end
```

Listing A.2: Implementation of algorithm 3.1.1.

```
function [distance, C, D] = vecvec(A, B, r, V_init,
   options)
%vecvec Finds nearest skew-symmetric matrix pencil C-xD
   of rank <= r to
% the matrix pencil A-xB, using the "vec-trick".
%
%   Input:
%       A                   :   First coefficient of
   matrix pencil A-xB.
%       B                   :   Second coefficient of
   matrix pencil A-xB.
%       r                   :   Rank of matrix to search
   for, in [2,n).
%       V_init              :   Initial guess for the n
   by s matrix V.
%       max_iter (optional) :   Maximum number of
   iterations, default 3000.
%       epsilon (optional)  :   Sensitivity of
   termination criteria,
%                               default 10^-7.
%
%   Output:
%       distance    :   Distance between matrix pencils A
   -xB and C-xD, in
%                       the sense of (|A-C|^2 + |B-D|^2)
   ^(1/2) using the
%                       Frobenius norm.
```

```
%       C           :   First coefficient of matrix
   pencil C-xD.
%       D           :   Second coefficient of matrix
   pencil C-xD.

    arguments
        A double {mustBeNumeric, mustBeSquare(A)};
        B double {mustBeNumeric, mustBeSquare(B),
           mustBeEqualSize(A,B)};
        r double {mustBeNonnegative, mustBeWithin(r,A)};
        V_init double;
        options.max_iter double {mustBeNonnegative} =
           3000
        options.epsilon double {mustBeNonnegative} =
           10^-7;
        options.phase_1 logical = false;
        options.epsilon_1 double = 10^-3;
    end

    % Constants
    [n,~] = size(A);
    s = floor(r/2);
    N = options.max_iter;
    term_crit = options.epsilon * norm([A;B],"fro");
    if options.phase_1
        phase = 1;
    else
        phase = 2;
    end
    term_crit_ph1 = options.epsilon_1 * norm([A;B],"fro")
       ;

    % Initial V
    V = rand(n,s,like=[A;B]);
    % V should be real if both A and B are, otherwise
       complex.

    % Vectorize
    A_vec = reshape(A,[],1);
    B_vec = reshape(B,[],1);

    % Permutation matrix (transpose)
    P = zeros(n*s,n*s);
    for i=1:n % ith row of blocks
        for j=1:s % jth column of blocks
            % This is the block p(i,j)
```

```
            % It should have a 1 in position (j,i) (
                inside the block)
            % The block contains P((i-1)*s+1:i*s, (j-1)*n
                +1:j*n)
            % The jth row of the block is the row (i-1)*s
                +j of P
            % and the ith column is (j-1)*n+i
            P((i-1)*s+j, (j-1)*n+i) = 1;
        end
    end

    I = eye(n);
    distance = NaN;
    prevdist = NaN;
    for i=1:N

        if phase == 1

            % Approximate V, solve for W1, approximate V,
                solve for W0, ...
            W0 = svd_solver(V,A);
            V = svd_solver(W0,-A);
            W1 = svd_solver(V,B);
            V = svd_solver(W1,-B);

        else
            % Solve for W
            V_tilde = kron(I, V)*P - kron(V, I);

            dV = decomposition(V_tilde); % saves time
            W0_vec = dV\A_vec;
            W1_vec = dV\B_vec;

            W0 = reshape(W0_vec,n,s);
            W1 = reshape(W1_vec,n,s);

            % Solve for V
            W0_tilde = kron(W0, I) - kron(I, W0)*P;
            W1_tilde = kron(W1, I) - kron(I, W1)*P;

            V_vec = [W0_tilde; W1_tilde]\[A_vec; B_vec];

            V = reshape(V_vec,n,s);
        end

        % Calc distance
```

```
        distance = norm([A;B] - [V*W0.'-W0*V.'; V*W1.'-W1
           *V.'], "fro");

        % Check termination criterion
        if abs(prevdist - distance) < term_crit
            break;
        elseif phase == 1 && abs(prevdist - distance) <
           term_crit_ph1
            phase = 2;
        end

        prevdist = distance;
    end


    C = V*W0.'-W0*V.';
    D = V*W1.'-W1*V.';
end

function mustBeSquare(M)
    [m,n] = size(M);
    if m ~= n
        error("Matrices must be square.")
    end
end

function mustBeEqualSize(M,N)
    if ~isequal(size(M),size(N))
        error("Matrices must be of equal size.")
    end
    if any([size(M),size(N)] <= 2)
        error("Size of input pencil must be greater than
           2 by 2.")
    end
end

function mustBeWithin(r,M)
    if r < 2 || r >= length(M)
        error("Rank r must be in [2,n).")
    end
end
```

Listing A.3: Implementation of algorithm 3.2.2.

```
function [distance, C, D] = svdvec(A, B, r, V_init,
   options)
%svdvec Finds nearest skew-symmetric matrix pencil C-xD
```

```
   of rank <= r to
% the skew-symmetric matrix pencil A-xB using the SVD and
    the "vec-trick".
%
%   Input:
%       A                   :   First coefficient of
   matrix pencil A-xB.
%       B                   :   Second coefficient of
   matrix pencil A-xB.
%       r                   :   Rank of matrix to search
   for, in [2,n).
%       V_init              :   Initial guess for the n
   by s matrix V.
%       max_iter (optional) :   Maximum number of
   iterations, default 3000.
%       epsilon (optional)  :   Sensitivity of
   termination criteria,
%                               default 10^-7.
%
%   Output:
%       distance    :   Distance between matrix pencils A
   -xB and C-xD, in
%                       the sense of (|A-C|^2 + |B-D|^2)
   ^(1/2) using the
%                       Frobenius norm.
%       C           :   First coefficient of matrix
   pencil C-xD.
%       D           :   Second coefficient of matrix
   pencil C-xD.

    arguments
        A double {mustBeNumeric, mustBeSkewSym(A)};
        B double {mustBeNumeric, mustBeSkewSym(B),
           mustBeEqualSize(A,B)};
        r double {mustBeNonnegative, mustBeWithin(r,A)};
        V_init double;
        options.max_iter double {mustBeNonnegative} =
           3000;
        options.epsilon double {mustBeNonnegative} =
           10^-7;
        options.phase_1 logical = false;
        options.epsilon_1 double = 10^-3;
    end

    % Constants
    [n,~] = size(A);
```

```
s = floor(r/2);
N = options.max_iter;
term_crit = options.epsilon * norm([A;B],"fro");
if options.phase_1
    phase = 1;
else
    phase = 2;
end
term_crit_ph1 = options.epsilon_1 * norm([A;B],"fro")
   ;

% Vectorize
A_vec = reshape(A,[],1);
B_vec = reshape(B,[],1);

% Permutation matrix (for transpose)
P = zeros(n*s,n*s);
for i=1:n % ith row of blocks
    for j=1:s % jth column of blocks
        % This is the block p(i,j)
        % It should have a 1 in position (j,i) (
           inside the block)
        % The block contains P((i-1)*s+1:i*s, (j-1)*n
           +1:j*n)
        % The jth row of the block is the row (i-1)*s
           +j of P
        % and the ith column is (j-1)*n+i
        P((i-1)*s+j, (j-1)*n+i) = 1;
    end
end

% Identity matrix
I = eye(n);

% Initial V
V = rand(n,s,like=[A;B]);
% V should be real if both A and B are, otherwise
   complex.

distance = NaN;
prevdist = NaN;
for i=1:N

    if phase == 1

        % Approximate V, solve for W1, approximate V,
```

```
        solve for W0, ...
    W0 = svd_solver(V,A);
    V = svd_solver(W0,-A);
    W1 = svd_solver(V,B);
    V = svd_solver(W1,-B);

else
    % Solve for W using SVD
    [R,S,T] = svd(V);
    S1 = S(1:s,:);

    % Solve for W0...
    E = R'*A*conj(R);
    E11 = E(1:s,1:s);
    E12 = E(1:s,s+1:n);

    % Y = T'*W0.'*conj(R)
    Y1 = zeros(s);
    for k = 1:s
        for l = 1:s
            Y1(k,l) = E11(k,l)/(S1(k,k)+S1(l,l));
        end
    end

    Y2 = S1\E12;
    Y = [Y1 Y2];
    W0 = R*Y.'*T.';

    % Solve for W1...
    E = R'*B*conj(R);
    E11 = E(1:s,1:s);
    E12 = E(1:s,s+1:n);

    % Y = T'*W1.'*conj(R) (s by n)
    Y1 = zeros(s);
    for k = 1:s
        for l = 1:s
            Y1(k,l) = E11(k,l)/(S1(k,k)+S1(l,l));
        end
    end

    Y2 = S1\E12;
    Y = [Y1 Y2];
    W1 = R*Y.'*T.';

    % Solve for V using vec
```

```
            W0_tilde = kron(W0, I) - kron(I, W0)*P;
            W1_tilde = kron(W1, I) - kron(I, W1)*P;

            V_vec = [W0_tilde; W1_tilde]\[A_vec; B_vec];
            V = reshape(V_vec,n,s);
        end

        % Calc distance
        distance = norm([A;B] - [V*W0.'-W0*V.'; V*W1.'-W1
           *V.'], "fro");

        % Check termination criterion
        if abs(prevdist - distance) < term_crit
            break;
        elseif phase == 1 && abs(prevdist - distance) <
           term_crit_ph1
            phase = 2;
        end

        prevdist = distance;
    end

    C = V*W0.'-W0*V.';
    D = V*W1.'-W1*V.';

end

function mustBeSkewSym(M)
    if ~isequal(M,-M.')
        error("Matrices must be skew symmetric.")
    end
end

function mustBeEqualSize(M,N)
    if ~isequal(size(M),size(N))
        error("Matrices must be of equal size.")
    end
    if any([size(M),size(N)] <= 2)
        error("Size of input pencil must be greater than
           2 by 2.")
    end
end

function mustBeWithin(r,M)
    if r < 2 || r >= length(M)
        error("Rank r must be in [2,n).")
```

```
    end
end
```

Listing A.4: Implementation of algorithm 3.3.1.

```
function [distance, C, D] = svdgup(A, B, r, V_init,
   options)
%svdgup Finds upper bound of the distance to the nearest
   skew-symmetric
% matrix pencil C-xD of rank <= r to the skew-symmetric
   matrix pencil A-xB
% using the SVD and GUPTRI.
%
%   Input:
%       A                    :   First coefficient of
   matrix pencil A-xB.
%       B                    :   Second coefficient of
   matrix pencil A-xB.
%       r                    :   Rank of matrix to search
   for, in [2,n).
%       V_init               :   Initial guess for the n
   by s matrix V.
%       max_iter (optional)  :   Maximum number of
   iterations, default 3000.
%       epsilon (optional)   :   Sensitivity of
   termination criteria,
%                                default 10^-7.
%
%   Output:
%       distance     :   Distance between matrix pencils A
   -xB and C-xD, in
%                        the sense of (|A-C|^2 + |B-D|^2)
   ^(1/2) using the
%                        Frobenius norm.
%       C            :   First coefficient of matrix
   pencil C-xD.
%       D            :   Second coefficient of matrix
   pencil C-xD.

    arguments
        A double {mustBeNumeric, mustBeSkewSym(A)};
        B double {mustBeNumeric, mustBeSkewSym(B),
           mustBeEqualSize(A,B)};
        r double {mustBeNonnegative, mustBeWithin(r,A)};
        V_init double;
        options.max_iter double {mustBeNonnegative} =
           3000;
```

```
        options.epsilon double {mustBeNonnegative} =
           10^-7;
        options.phase_1 logical = false;
        options.epsilon_1 double = 10^-3;
    end

    % Constants
    [n,~] = size(A);
    s = floor(r/2);
    N = options.max_iter;
    term_crit = options.epsilon * norm([A;B],"fro");

    % Permutation matrix (transpose)
    Perm = zeros(s*s,s*s);
    for i=1:s % ith row of blocks
        for j=1:s % jth column of blocks
            % This is the block p(i,j)
            % It should have a 1 in position (j,i) (
               inside the block)
            Perm((i-1)*s+j, (j-1)*s+i) = 1;
        end
    end
    I = eye(s);

    % Initial V
    V = V_init;

    distance = NaN;
    prevdist = NaN;
    mindist = NaN;
    C_min = zeros(n);
    D_min = zeros(n);
    for i=1:N

        % Solve for W using SVD
        [R,S,T] = svd(V);
        S1 = S(1:s,:);

        % Solve for W0...
        E = R'*A*conj(R);
        E11 = E(1:s,1:s);
        E12 = E(1:s,s+1:n);

        % Y = T'*W0.'*conj(R)
        Y1 = zeros(s);
        for k = 1:s
```

```
    for l = 1:s
        Y1(k,l) = E11(k,l)/(S1(k,k)+S1(l,l));
    end
end

Y2 = S1\E12;
Y = [Y1 Y2];
W0 = R*Y.'*T.';

% Solve for W1...
E = R'*B*conj(R);
E11 = E(1:s,1:s);
E12 = E(1:s,s+1:n);

% Y = T'*W1.'*conj(R) (s by n)
Y1 = zeros(s);
for k = 1:s
    for l = 1:s
        Y1(k,l) = E11(k,l)/(S1(k,k)+S1(l,l));
    end
end

Y2 = S1\E12;
Y = [Y1 Y2];
W1 = R*Y.'*T.';

% Solve for V using GUPTRI
[S,T,P,Q,~] = pguptri(W0,W1,zeros=true);

S1 = S(1:s,:);
T1 = T(1:s,:);
T2 = T(s+1:2*s,:);

S23 = S(s+1:n,:);
T3 = T(2*s+1:n,:);

% S2, S3 and T3 should be zero. If not, something
    is wrong.
assert(isequal(S23, zeros(n-s,s)), "S2-3 non-zero
   ")
assert(isequal(T3, zeros(n-2*s,s)), "T3 non-zero
   ")

% Calculate E and F
E = P'*A*conj(P);
F = P'*B*conj(P);
```

```
% Blocks of E and F
E11 = E(1:s, 1:s);
E13 = E(1:s, 2*s+1:n);
E21 = E(s+1:2*s, 1:s);
F11 = F(1:s, 1:s);
F12 = F(1:s, s+1:2*s);
F22 = F(s+1:2*s, s+1:2*s);
F13 = F(1:s, 2*s+1:n);
F23 = F(s+1:2*s, 2*s+1:n);

% Solve for X = [X1; X2; X3]; X = P'*V*conj(Q)
X3 = -([S1; T1; T2]\[E13; F13; F23]).';

% For X1 and X2 we will be using vec...

C1 = kron(S1,I);
RHS1 = reshape(E21,[],1); % This is vec(E21)
C2 = (kron(T2,I) - kron(I,T2)*Perm);
RHS2 = reshape(F22,[],1);

X2vec = [C1; C2]\[RHS1; RHS2];

C3 = (kron(S1,I) - kron(I,S1)*Perm);
RHS3 = reshape(E11,[],1);
C4 = (kron(T1,I) - kron(I,T1)*Perm);
RHS4 = reshape(F11,[],1);
C5 = kron(T2,I);
RHS5 = reshape(F12,[],1) + kron(I,T1)*Perm*X2vec;

X1vec = [C3; C4; C5]\[RHS3; RHS4; RHS5];

X1 = reshape(X1vec,s,s);
X2 = reshape(X2vec,s,s);

X = [X1; X2; X3];
V = P*X*Q.';

% Calc distance
distance = norm([A;B] - [V*W0.'-W0*V.'; V*W1.'-W1
   *V.'], "fro");

% In first iteration and whenever distance is
   less than recorded
% minimum, update minimum
if i == 1 || distance < mindist
```

```
            mindist = distance;
            C_min = V*W0.'-W0*V.';
            D_min = V*W1.'-W1*V.';
        end

        if isnan(distance)
            break;
        end

        % Check termination criterion
        if abs(prevdist - distance) < term_crit
            break;
        end

        prevdist = distance;
    end

    % return best solution found
    C = C_min;
    D = D_min;
    distance = mindist;

end

function mustBeSkewSym(M)
    if ~isequal(M,-M.')
        error("Matrices must be skew symmetric.")
    end
end

function mustBeEqualSize(M,N)
    if ~isequal(size(M),size(N))
        error("Matrices must be of equal size.")
    end
    if any([size(M),size(N)] <= 2)
        error("Size of input pencil must be greater than
           2 by 2.")
    end
end

function mustBeWithin(r,M)
    if r < 2 || r >= length(M)
        error("Rank r must be in [2,n).")
    end
end
```

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY