COMPUTING MATRIX FUNCTIONS IN ARBITRARY PRECISION ARITHMETIC

A thesis submitted to The University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

2022

Xiaobo Liu
School of Mathematics

CONTENTS

Word count: 48 023

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Functions of matrices play an important role in many applications in science and engineering. Their reliable computation has been a topic of interest in numerical linear algebra over the decades, and a wide variety of methods for computing different functions have been studied. Meanwhile, the interest in multiple precision computing environments has been growing in recent years and nowadays there is an explosion of floating-point arithmetics beyond the most widely used standard IEEE binary32 and binary64 formats. Under such background, there are several works dedicated to the computation of matrix functions in arbitrary precision arithmetic, but overall this research topic has not yet attracted much attention. In this thesis, we study methods for computing functions of matrices in arbitrary precision arithmetic. Unlike many existing algorithms that are tightly coupled to a specific precision of floating-point arithmetic, the algorithms we develop take the unit roundoff of the working precision as an input argument since this is known only at runtime, and so works in an arbitrary precision.

First, we provide a version of the Schur–Parlett algorithm that requires only function values and not derivatives. The algorithm requires access to arithmetic of a matrix-dependent precision at least double the working precision, which is used to evaluate the function on the diagonal blocks of order greater than 2 (if there are any) of the reordered and blocked Schur form. The key idea is to compute by diagonalization the function of a small random diagonal perturbation of each diagonal block, where the perturbation ensures that diagonalization will succeed. The algorithm is inspired by Davies's randomized approximate diagonalization method, but we explain why that is not a reliable numerical method for computing matrix functions. This multiprecision Schur–Parlett algorithm is applicable to arbitrary analytic functions f and, like the original Schur–Parlett algorithm, it generally behaves in a numerically stable fashion. The algorithm is especially useful when the derivatives of f are not readily available or accurately computable. We apply our algorithm to the matrix Mittag–Leffler function and show that it yields results of accuracy similar to, and in some cases much greater than, the state-of-the-art algorithm for this function.

Second, we develop an algorithm for computing the matrix cosine in arbitrary precision. The algorithm employs a Taylor approximation with scaling and recovering and it can be used with a Schur decomposition or in a decomposition-free manner. We also derive a framework for computing the Fréchet derivative, construct an efficient evaluation scheme for computing the cosine and its Fréchet derivative simultaneously in arbitrary precision, and show how this scheme can be extended to compute the matrix sine, cosine, and their Fréchet derivatives all together. Numerical experiments show that the new algorithms behave in a forward stable way over a wide range of precisions. The transformation-free version of the algorithm for computing the cosine is competitive in accuracy with the state-of-the-art algorithms in double precision and surpasses existing alternatives in both speed and accuracy in working precisions higher than double.

Finally, we consider the problem of computing the square root of a perturbation of the scaled identity matrix, $A = \alpha I_n + UV^*$, where U and V are $n \times k$ matrices with $k \leqslant n$. This problem arises in various applications, including computer vision and

optimization methods for machine learning. We derive a new formula for the pth root of A that involves a weighted sum of powers of the pth root of the $k \times k$ matrix $\alpha I_k + V^*U$. This formula is particularly attractive for the square root, since the sum has just one term when $p = 2$. We derive a new class of Newton iterations for computing the square root that exploit the low-rank structure. Theses methods can be employed in arbitrary precision by simply executing all elementary scalar operations in arbitrary precision, and, additionally, for the iterative algorithms, by properly adjusting the internal tolerance that is used as stopping criterion. We also proposed several Schur-based methods that can utilize the structure of A. In particular, we established a scheme to obtain the Schur decomposition of the $n \times n$ matrix UV^* from the Schur decomposition of the $k \times k$ matrix V^*U. We test these new methods on random matrices and on positive definite matrices arising in applications. Numerical experiments show that the new approaches can yield much smaller residual than existing alternatives and can be significantly faster when the perturbation UV^* has low rank.

DECLARATION

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

COPYRIGHT STATEMENT

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420`), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see `http://www.library.manchester.ac.uk/about/regulations`) and in The University's Policy on Presentation of Theses.

ACKNOWLEDGEMENTS

First, I would like to express my most sincere gratitude to my supervisor, Prof. Nicholas J. Higham, for his expert guidance and unfailing support over the past three-and-a-half years during my doctoral studies. I could not have wished for a more professional and knowledgeable mentor like Nick, who always carefully reads manuscripts and constantly offers insightful feedback. Without his efforts this thesis would not have been possible.

I would like to thank Prof. Françoise Tisseur, Prof. Stefan Güttel, Dr. Marcus Webb, and all the other members of the Manchester Numerical Linear Algebra group, a truly exceptional research group, where there are friendly scholars and engaging research environment; it has been my privilege to be a part of it.

Many thanks to Prof. Awad H. Al-Mohy and Dr. Massimiliano Fasi for useful discussions that makes our collaboration a success.

Thanks should also go to my fantastic officemates of Office 2.111. Thank you Michael P. Connolly, Xinye Chen, Thomas McSweeney, and Gian Maria Negri Porzio for our interesting discussions and distractions. A special thank you goes to Dr. Massimiliano Fasi, who was a role model postgraduate student to me and generously shared his knowledge and experience with me. I still recall the many discussions and meals we had together.

There are many other fellow students and friends that I would like to thank. Thank you Lingshu Lei, Dr. Xiannan Meng, Mansell Lin, Ying Liu, Jiaming Shen, and Chunyu Wang for all the enjoyable time we spent together.

I gratefully acknowledge the funding I received from the Department of Mathematics at The University of Manchester for my postgraduate studies.

Last but not least, I thank my parents, Zhengliang Liu and Dongju Huang, for their unconditional love and unwavering support to me. Without them I would by no means have made it this far.

PUBLICATIONS

- Chapter 3 is based on the journal article: Nicholas J. Higham and Xiaobo Liu. A Multiprecision Derivative-Free Schur–Parlett Algorithm for Computing Matrix Functions. *SIAM J. Matrix Anal. Appl.* 42.3 (2021), pp. 1401–1422.
- Chapter 4 is based on the journal article: A. H. Al-Mohy, Nicholas J. Higham, and Xiaobo Liu. Arbitrary Precision Algorithms for Computing the Matrix Cosine and its Fréchet Derivative. *SIAM J. Matrix Anal. Appl.* 43.1 (2022), pp. 233–256.
- Chapter 5 is based on the preprint: Massimiliano Fasi, Nicholas J. Higham, and Xiaobo Liu. Computing the square root of a low-rank perturbation of the scaled identity matrix. MIMS EPrint 2022.1. UK: Manchester Institute for Mathematical Sciences, The University of Manchester, Jan. 2022. Revised May 2022, p. 19. Submitted to *SIAM J. Matrix Anal. Appl.*

1 INTRODUCTION

The term "matrix" was coined by James Joseph Sylvester in 1850 [77], but, long before that, the mathematical object, in its guise of arrays, had been used in solving system of linear equations. *The Nine Chapters on the Mathematical Arts*,[1] a Chinese mathematics book written from the 10th to the 2nd century B.C.E., first used a method that is similar to Gaussian elimination to find the solution of linear equations [75]. Arthur Cayley, who was the pioneer to study matrices in their own right, investigated algebraic properties of matrices and also began the study of functions of matrices in his 1858 paper "A Memoir on the Theory of Matrices" [13], where he treated the square roots of 2×2 and 3×3 matrices.

The study of theory of matrix functions thrived over the succeeding hundred years and many mathematicians contributed to this research field. Laguerre first defined in 1867 [50] via the power series the exponential of a square matrix, which was subsequently shown by Peano [64] to be capable of representing the solution of systems of homogeneous differential equations. Metzler [55] defined the matrix exponential, the matrix sine, and their inverse functions all via power series. Weyr [80] was the first to give a convergence criterion for matrix functions defined by a matrix power series. In addition to the power series definition, functions of matrices can be defined via many other equivalent means. In 1883 Sylvester [76] derived an interpolating polynomial formula for functions of square matrices that have distinct eigenvalues; the same formula was also derived [11] by Buchheim who later generalized it to multiple eigenvalues using Hermite interpolation [10]. The Cauchy integral representation of functions of matrices is due to Frobenius [21] and Giorgi [22], and the latter also stated the Jordan canonical form definition of matrix functions [22]. The equivalence of all the above definitions of matrix functions was then proved by Rinehart [66]

1 《九章算術》.

in 1955. See [31, sect. 1.10] for an informative and more thorough survey of the historical development of matrix functions.

Matrix functions were purely of theoretical interest in their early stage of development, and the theory is treated in a number of books. The first research monograph written on matrix functions, by Schwerdtfeger [72], was published in 1938, though there were many books with substantial materials on the topic before. The landmark of matrix functions being utilized in practical applications is the book *Elementary Matrices and Some Applications to Dynamics and Differential Equations* [20] published in the same year by Frazer, Duncan, and Collar, which accentuates the importance of matrix exponential in solving differential equations. Since then matrix functions have gradually broadened into a flourishing subject of study in applied mathematics from their root in pure mathematics, with new applications constantly being found in science and engineering. Examples include the matrix exponential and the matrix trigonometric sine and cosine, whose archetypal application is in expressing the solution of matrix differential equations [23]; the matrix logarithm, which arises in computer graphics [68], [70], image recognition [5], [38], and optical systems representation [28]; the matrix roots, which are frequently used in machine learning [27], [65], [74]; and other matrix functions; we refer the readers to [31, Chap. 2] for many more applications of various matrix functions.

Driven by the constantly emerging and widespread new applications of functions of matrices, the interest in the numerical computation of matrix functions is growing rapidly over the decades. Various methods have been developed for evaluating matrix functions, and there is now a tremendous literature on them, for example, the scaling and squaring algorithm for the matrix exponential [1], [57], the Schur methods [9], [30] and Newton's method [31, sect. 6.3] for the matrix square root, the Schur–Padé algorithm for the real matrix power [37], the inverse scaling and squaring algorithm for the matrix logarithm [2]; and routines of many functions have been implemented in prevalent languages and libraries including the NAG library [60], The Matrix Function Toolbox [33] of MATLAB, Python's SymPy [56], and Armadillo [71] for C++. A catalogue of software for various matrix functions available in different languages and packages is given in the recent work [36].

Nowadays, we are seeing increasing use of binary floating-point arithmetics beyond the 32-bit IEEE single precision and the 64-bit IEEE double precision arithmetics [42], where most floating-point calculations in scientific computing have been carried out since their launch in 1985. The interest in low precision arithmetic (and in particular, the 16-bit half precision, also called binary16) has exploded in recent years due to its satisfactory performance and energy efficiency in climate modelling [62], [61] and in training and running neural networks in deep learning [14]. On the other hand, the need for higher precisions has emerged in various area including analytic number theory [7], cryptography [19], high-performance computing [29], optimization [52], and physics applications [6], [49]. The 2008 IEEE standard revision [43] added a 128-bit quadruple precision floating-point format and a 16-bit half precision format, the latter defined as a storage format only rather than for computation. The former floating-point format is now supported on the IBM z13 processor [51] and the IBM Power9 processor [79] and available in software, while the latter has been adopted by various manufacturers for computation. For example, binary16 is supported by the NVIDIA V100 and A100 GPUs, and the AMD Radeon Instinct MI250X GPU, as well as the A64FX Arm processor that powers the top-ranked[2] Fujitsu Post-K exascale computer. Worth mentioning, the newly released NVIDIA H100 GPU even supports calculations in an 8-bit quarter precision format. *Arbitrary* precision arithmetic is available in a wide range of software, including Maple [53], Mathematica [54], PARI/GP [63], Sage [69], Python's mpmath [45] and SymPy [56], Julia [8], [46] through its built-in data type `BigFloat`, and MATLAB with the Symbolic Math Toolbox [78] or the Multiprecision Computing Toolbox [58].

Associated with the broadening of the precision landscape, there is a growing amount of research literature focusing on the development and efficient implementation of linear algebra subroutines for arbitrary precision arithmetic, including those on the computation of functions of matrices. Given that the computation of matrix functions in double precision has been well-studied, it might be plausible to think if

2 This is from the 58th TOP500 list https://www.top500.org/lists/top500/2021/11/.

we can modify existing algorithms to arbitrary precision environments with little or no modifications. At least two classes of algorithms fall into this category. Iterative methods can often be run in arbitrary precision by simply perform all elementary scaler operations in arbitrary precision and adjusting the prescribed tolerance which is used as stopping criterion; for instance, the Newton's method for the matrix sign function [48], [47], [67], the matrix square root [32], [34], [40], the matrix *p*th root [25], [26], [39], and the matrix Lambert W function [18]. Another typical example is substitution methods, such as the algorithms for the matrix square root [9], [30] and the matrix *p*th root [24], [41], [73], whose mechanism resembles that of the forward and backward substitution for the solution of linear systems.

Many advanced algorithms, howbeit, cannot conveniently extend to an arbitrary precision environment. These algorithms typically approximate the matrix function by a polynomial or rational approximant at a matrix argument, and require precomputing symbolically or in high precision a set of precision-dependent constants that are crucial for selecting algorithmic parameters since they appear in the truncation error bounds or as the coefficients of the approximating functions. This strategy, proposed by Higham [35] for computing the matrix exponential in double precision, proves very efficient and is adopted in the state-of-the-art algorithms for various matrix functions including the matrix exponential [1], the matrix logarithm [2], the matrix fractional powers [37], the matrix sine and cosine [3], the matrix inverse trigonometric and inverse hyperbolic functions [4], and the wave-kernel matrix functions [59]. The algorithmic design of these algorithms significantly depends on the knowledge of the working precision which is only to be known at runtime and therefore is impractical to be carried out in arbitrary precision environments.

Several algorithms for computing matrix functions that work in arbitrary precision have been developed, inclusive of the ones for matrix exponential [12], [16] and the matrix logarithm [17], which take the working precision at which the algorithm is to be executed as an input argument to the algorithm, to minimize the impact of the working precision on the design stage. Besides, there are many computer algebra systems that offer functions for evaluating in arbitrary precision a wide range of matrix functions, including Maple [53], Mathematica [54], Python's mpmath library [45], MATLAB with the Symbolic Math Toolbox [78] and the Multiprecision Computing Toolbox [58], and the `ArbFloats` package, a wrapper to the C library Arb [44]. However, we are not aware of the underlying algorithms implemented in the above software as well as details of the implementations, which in some cases may involve symbolic arithmetic.

This thesis mainly explores the computation of matrix functions in arbitrary precision and possesses both theoretical and computational contributions to the area of matrix functions. Firstly, we build a multiprecision derivative-free version of the Schur–Parlett algorithm that greatly expands the class of readily computable matrix functions; we explain why Davies's randomized approximate diagonalization method [15], which is widely used for calculating reference solutions in higher precision when the forward error of algorithms for matrix functions is estimated, is not reliable and show how to estimate the condition number of the eigenvector matrix of a triangular matrix based on its elements, so increase accordingly the precision at which to carry out the diagonalization to guarantee accuracy. Secondly, we develop an arbitrary precision algorithm for computing the matrix cosine and its Fréchet derivative simultaneously. We also generalize the algorithmic framework to evaluate the matrix the matrix sine, cosine, and their Fréchet derivatives all together. Thirdly, we derive a new formula for the pth root of a perturbation of the scaled identity matrix, and focus on the case of $p = 2$ when the formula is of most computational interest and derive a new class of Newton iterations for computing the square root that exploit the possible low-rank structure.

In the next chapter we revise basic definitions and preliminary results in general matrix theory, functions of matrices, and floating-point arithmetic that build the foundation of our research. The thesis is in the journal format, with Chapters 3, 4, and 5 presented in a format suitable for publication and based on the preprints and journal papers listed on page 12; the authors of the papers on which these chapters are based contributed equally to the final manuscripts, and therefore it is not necessary to further discriminate their specific contribution. Conclusions and remarks on future work are offered in Chapter 6.

REFERENCES

[1] A. H. Al-Mohy and N. J. Higham. "A new scaling and squaring algorithm for the matrix exponential." *SIAM J. Matrix Anal. Appl.* 31.3 (2009), pp. 970–989 (cited on pp. 14, 16).

[2] A. H. Al-Mohy and N. J. Higham. "Improved inverse scaling and squaring algorithms for the matrix logarithm." *SIAM J. Sci. Comput.* 34.4 (2012), pp. C153–C169 (cited on pp. 14, 16).

[3] A. H. Al-Mohy, N. J. Higham, and S. D. Relton. "New algorithms for computing the matrix sine and cosine separately or simultaneously." *SIAM J. Sci. Comput.* 37.1 (2015), A456–A487 (cited on p. 16).

[4] M. Aprahamian and N. J. Higham. "Matrix inverse trigonometric and inverse hyperbolic functions: Theory and algorithms." *SIAM J. Matrix Anal. Appl.* 37.4 (2016), pp. 1453–1477 (cited on p. 16).

[5] V. Arsigny, O. Commowick, N. Ayache, and X. Pennec. "A fast and log-euclidean polyaffine framework for locally linear registration." *J. Math. Imaging Vis.* 33 (2009), pp. 222–238 (cited on p. 14).

[6] D. H. Bailey, R. Barrio, and J. M. Borwein. "High-precision computation: mathematical physics and dynamics." *Appl. Math. Comput.* 218.20 (2012), pp. 10106–10121 (cited on p. 15).

[7] G. Beliakov and Y. Matiyasevich. "A parallel algorithm for calculation of determinants and minors using arbitrary precision arithmetic." *BIT Numer. Math.* 56 (2016), pp. 33–50 (cited on p. 15).

[8] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. "Julia: a fresh approach to numerical computing." *SIAM Rev.* 59.1 (Jan. 2017), pp. 65–98 (cited on p. 15).

[9] Å. Björck and S. Hammarling. "A Schur method for the square root of a matrix." *Linear Algebra Appl.* 52/53 (1983), pp. 127–140 (cited on pp. 14, 16).

[10] A. Buchheim. "An extension of a theorem of Professor Sylvester's relating to matrices." *London, Edinburgh, Dublin Philos. Mag. J. Sci.* 22.135 (1886), pp. 173–174 (cited on p. 13).

[11] A. Buchheim. "On the theory of matrics." *Proc. London Math. Soc.* 16 (1884), pp. 63–82 (cited on p. 13).

[12] M. Caliari and F. Zivcovich. "On-the-fly backward error estimate for matrix exponential approximation by Taylor algorithm." *J. Comput. Appl. Math.* 346 (2019), pp. 532–548 (cited on p. 16).

[13] A. Cayley. "A memoir on the theory of matrices." *Philos. Trans. Roy. Soc. London* 148 (Jan. 1858), pp. 17–37 (cited on p. 13).

[14] M. Courbariaux, Y. Bengio, and J.-P. David. *Training Deep Neural Networks with Low Precision Multiplications*. 2015. ArXiv preprint 1412.7024v5 (cited on p. 15).

[15] E. B. Davies. "Approximate diagonalization." *SIAM J. Matrix Anal. Appl.* 29.4 (2008), pp. 1051–1064 (cited on p. 17).

[16] M. Fasi and N. J. Higham. "An arbitrary precision scaling and squaring algorithm for the matrix exponential." *SIAM J. Matrix Anal. Appl.* 40.4 (2019), pp. 1233–1256 (cited on p. 16).

[17] M. Fasi and N. J. Higham. "Multiprecision algorithms for computing the matrix logarithm." *SIAM J. Matrix Anal. Appl.* 39.1 (2018), pp. 472–491 (cited on p. 16).

[18] M. Fasi, N. J. Higham, and B. Iannazzo. "An algorithm for the matrix Lambert *W* function." *SIAM J. Matrix Anal. Appl.* 36.2 (2015), pp. 669–685 (cited on p. 16).

[19] A. Flores-Vergara, E. E. García-Guerrero, E. Inzunza-González, O. R. López-Bonilla, E. Rodríguez-Orozco, J. R. Cárdenas-Valdez, and E. Tlelo-Cuautle. "Implementing a chaotic cryptosystem in a 64-bit embedded system by using multiple-precision arithmetic." *Nonlinear Dyn.* 96 (2019), pp. 497–516 (cited on p. 15).

[20] R. A. Frazer, W. J. Duncan, and A. R. Collar. *Elementary Matrices and Some Applications to Dynamics and Differential Equations*. Cambridge University Press, 1938, pp. xviii+416. 1963 printing (cited on p. 14).

[21] G. Frobenius. "Über die cogredienten transformationen der bilinearen formen". *Sitzungsber K. Preuss. Akad. Wiss. Berlin* 16 (1896), pp. 7–16 (cited on p. 13).

[22] G. Giorgi. "Nuove osservazioni sulle funzioni delle matrici". *Atti Accad. Lincei Rend.* 6.8 (1928), pp. 3–8 (cited on p. 13).

[23] S. K. Godunov. *Ordinary differential equations with constant coefficient*. Americal Mathematical Society, Providence, RI, USA: volume 169 of *Translations of Mathematical Monographs*, 1997, pp. ix+282 (cited on p. 14).

[24] F. Greco and B. Iannazzo. "A binary powering Schur algorithm for computing primary matrix roots." *Numer. Algorithms* 55.1 (2010), pp. 59–78 (cited on p. 16).

[25] C.-H. Guo. "On Newton's method and Halley's method for the principal *p*th root of a matrix." *Linear Algebra Appl.* 432.8 (2010), pp. 1905–1922 (cited on p. 16).

[26] C.-H. Guo and N. J. Higham. "A Schur–Newton method for the matrix *p*th root and its inverse." *SIAM J. Matrix Anal. Appl.* 28.3 (2006), pp. 788–804 (cited on p. 16).

[27] V. Gupta, T. Koren, and Y. Singer. "Shampoo: Preconditioned Stochastic Tensor Optimization." In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by J. Dy and A. Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden, 2018, pp. 1842–1850 (cited on p. 14).

[28] W. F. Harris. "The average eye." *Opthal. Physiol. Opt.* 24.6 (2004), pp. 580–585 (cited on p. 14).

[29] Y. He and C. H. Q. Ding. "Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications." *The Journal of Supercomputing* 18 (2001), pp. 259–277 (cited on p. 15).

[30] N. J. Higham. "Computing real square roots of a real matrix." *Linear Algebra Appl.* 88/89 (1987), pp. 405–430 (cited on pp. 14, 16).

[31] N. J. Higham. *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008, pp. xx+425 (cited on p. 14).

[32] N. J. Higham. "Newton's method for the matrix square root." *Math. Comp.* 46.174 (Apr. 1986), pp. 537–549 (cited on p. 16).

[33] N. J. Higham. *The Matrix Function Toolbox*. `http://www.maths.manchester.ac.uk/~higham/mftoolbox` (cited on p. 14).

[34] N. J. Higham. "The matrix sign decomposition and its relation to the polar decomposition." *Linear Algebra Appl.* 212/213 (1994), pp. 3–20 (cited on p. 16).

[35] N. J. Higham. "The scaling and squaring method for the matrix exponential revisited." *SIAM J. Matrix Anal. Appl.* 26.4 (2005), pp. 1179–1193 (cited on p. 16).

[36] N. J. Higham and E. Hopkins. *A Catalogue of Software for Matrix Functions. Version 3.0*. MIMS EPrint 2020.7. UK: Manchester Institute for Mathematical Sciences, The University of Manchester, Mar. 2020, p. 24 (cited on p. 15).

[37] N. J. Higham and L. Lin. "An improved Schur–Padé algorithm for fractional powers of a matrix and their Fréchet derivatives." *SIAM J. Matrix Anal. Appl.* 34.3 (2013), pp. 1341–1360 (cited on pp. 14, 16).

[38] W. Hu, H. Zuo, O. Wu, Y. Chen, Z. Zhang, and D. Suter. "Recognition of adult images, videos, and web page bags." *ACM Trans. Multimedia Comput. Commun. Appl.* 7S (2011), 28:1–28:24 (cited on p. 14).

[39] B. Iannazzo. "A family of rational iterations and its application to the computation of the matrix pth root." *SIAM J. Matrix Anal. Appl.* 30.4 (Jan. 2009), pp. 1445–1462 (cited on p. 16).

[40] B. Iannazzo. "A note on computing the matrix square root." *Calcolo* 40.4 (2003), pp. 273–283 (cited on p. 16).

[41] B. Iannazzo and C. Manasse. "A Schur logarithmic algorithm for fractional powers of matrices." *SIAM J. Matrix Anal. Appl.* 34.2 (2013), pp. 794–813 (cited on p. 16).

[42] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. IEEE, 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987 (cited on p. 15).

[43] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (revision of IEEE Std 754-2008)*. IEEE, 2019, p. 84 (cited on p. 15).

[44] F. Johansson. "Arb: efficient arbitrary-precision midpoint-radius interval arithmetic." *IEEE Trans. Comput.* 66.8 (2017), pp. 1281–1292 (cited on p. 17).

[45] F. Johansson et al. *Mpmath: A Python Library for Arbitrary-Precision Floating-Point Arithmetic*. 2013. `http://mpmath.org` (cited on pp. 15, 17).

[46] *Julia*. `http://julialang.org` (cited on p. 15).

[47] C. Kenney and A. J. Laub. "On scaling Newton's method for polar decomposition and the matrix sign function." *SIAM J. Matrix Anal. Appl.* 13.3 (July 1992), pp. 688–706 (cited on p. 16).

[48] C. Kenney and A. J. Laub. "Rational iterative methods for the matrix sign function." *SIAM J. Matrix Anal. Appl.* 12.2 (Apr. 1991), pp. 273–291 (cited on p. 16).

[49] G. Khanna. "High-precision numerical simulations on a CUDA GPU: Kerr black hole tails." *J. Sci. Comput.* 56 (2013), pp. 366–380 (cited on p. 15).

[50] E. N. Laguerre. "Le Calcul des Systèmes Linéaires, Extrait d'Une Lettre Adressé à M. Hermite." In: *Oeuvres de Laguerre*. Ed. by C. Hermite, H. Poincaré, and E. Rouché. Vol. 1. Gauthier–Villars, Paris, 1898, pp. 221–267. The article is dated 1867 and is "Extrait du Journal de l'École Polytechnique, LXII^{e} Cahier" (cited on p. 13).

[51] C. Lichtenau, S. Carlough, and S. M. Mueller. "Quad Precision Floating Point on the IBM z13." In: *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. 2016, pp. 87–94 (cited on p. 15).

[52] D. Ma and M. A. Saunders. "Solving Multiscale Linear Programs Using the Simplex Method in Quadruple Precision." In: *Numerical Analysis and Optimization*. Ed. by M. Al-Baali, L. Grandinetti, and A. Purnama. Vol. 134. Springer Proceedings in Mathematics & Statistics. Springer, Cham, 2015, pp. 223–235 (cited on p. 15).

[53] *Maple*. Waterloo Maple Inc., Waterloo, Ontario, Canada. `http://www.maplesoft.com` (cited on pp. 15, 17).

[54] *Mathematica*. Wolfram Research, Inc., Champaign, IL, USA. `http://www.wolfram.com` (cited on pp. 15, 17).

[55] W. H. Metzler. *On the roots of matrices*. Friedenwald, Baltimore, 1892 (cited on p. 13).

[56] A. Meurer, C. P. Smith, M. Paprocki, et al. "SymPy: Symbolic computing in Python." *PeerJ Comput. Sci.* 3 (Jan. 2017), e103 (cited on pp. 14, 15).

[57] C. B. Moler and C. F. Van Loan. "Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later." *SIAM Rev.* 45.1 (2003), pp. 3–49 (cited on p. 14).

[58] *Multiprecision Computing Toolbox*. Advanpix, Tokyo, Japan. `http://www.advanpix.com` (cited on pp. 15, 17).

[59] P. Nadukandi and N. J. Higham. "Computing the wave-kernel matrix functions." *SIAM J. Sci. Comput.* 40.6 (2018), A4060–A4082 (cited on p. 16).

[60] *NAG Library*. NAG Ltd., Oxford, UK. `https://www.nag.co.uk` (cited on p. 14).

[61] T. N. Palmer. "Modelling: build imprecise supercomputers." *Nature* 526 (2015), pp. 32–33 (cited on p. 15).

[62] T. N. Palmer. "More reliable forecasts with less precise computations: a fast-track route to cloud-resolved weather and climate simulators?" *Philos. Trans. Roy. Soc.* A372.2018 (2014) (cited on p. 15).

[63] *PARI/GP*. The PARI Group, Bordeaux. `http://pari.math.u-bordeaux.fr` (cited on p. 15).

[64] G. Peano. "Intégration par séries des équations différentielles linéaires." *Math. Annalen* 32.3 (Sept. 1888), pp. 450–456 (cited on p. 13).

[65] G. Pleiss, M. Jankowiak, D. Eriksson, A. Damle, and J. Gardner. "Fast matrix square roots with applications to Gaussian processes and Bayesian optimization." In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin. Vol. 33. Stockholmsmässan, Stockholm Sweden, 2020, pp. 22268–22281 (cited on p. 14).

[66] R. F. Rinehart. "The equivalence of definitions of a matric function". *Amer. Math. Monthly* 62.6 (1955), pp. 395–414 (cited on p. 13).

[67] J. D. Roberts. "Linear model reduction and solution of the algebraic Riccatiq equation by use of the sign function." *Internat. J. Control* 32.4 (Oct. 1980), pp. 677–687 (cited on p. 16).

[68] J. Rossignac and Á. Vinacua. "Steady affine motions and morphs." *ACM Trans. Graph.* 30.5 (2011), 116:1–116:16 (cited on p. 14).

[69] The Sage Developers. *Sage Mathematics Software*. http://www.sagemath.org (cited on p. 15).

[70] P. Sanan. "Geometric elasticity for graphics, simulation, and computation." PhD thesis. Pasadena, California, USA: California Institute of Technology, 2014 (cited on p. 14).

[71] C. Sanderson and R. Curtin. "Armadillo: a template-based C++ library for linear algebra." *Journal of Open Source software* 1.2 (2016). 26 pp (cited on p. 14).

[72] H. Schwerdtfeger. *Les Fonctions de Matrices. I. Les Fonctions Univalentes*. Hermann, Paris, France: Number 649 in *Actualités Scientifiques et Industrielles*, 1938, p. 58 (cited on p. 14).

[73] M. I. Smith. "A Schur algorithm for computing matrix *p*th roots." *SIAM J. Matrix Anal. Appl.* 24.4 (2003), pp. 971–989 (cited on p. 16).

[74] Y. Song, N. Sebe, and W. Wang. *Fast Differentiable Matrix Square Root*. ArXiv:1209.5145. 2022 (cited on p. 14).

[75] P. D. Straffin. "Liu Hui and the first golden age of Chinese mathematics." *Math. Mag.* 71.3 (June 1998), p. 163 (cited on p. 13).

[76] J. J. Sylvester. "On the equation to the secular inequalities in the planetary theory." *Phil. Mag. J. Sci.* 16.100 (1883), pp. 267–269 (cited on p. 13).

[77] J. J. Sylvester. "XLVII. Additions to the articles in the September number of this journal, "On a new class of theorems," and on Pascal's theorem." *London, Edinburgh Dublin Philos. Mag. J. Sci.* 37.251 (Nov. 1850), pp. 363–370 (cited on p. 13).

[78] *Symbolic Math Toolbox*. The MathWorks, Inc., Natick, MA, USA. `http://www.mathworks.co.uk/products/symbolic/` (cited on pp. 15, 17).

[79] T. Trader. *IBM advances against x86 with Power9*. `https://www.hpcwire.com/2016/08/30/ibm-unveils-power9-details/`. Aug. 2016. Accessed March 14, 2022 (cited on p. 15).

[80] E. Weyr. "Note sur la théorie de quantités complexes formées avec n unités principales". *Bull. Sci. Math. II* 11 (1887), pp. 205–215 (cited on p. 13).

2 | BACKGROUND MATERIAL

This chapter summarizes many useful definitions and facts in matrix theory, functions of matrices, and floating-point arithmetic. The review only contains a minimal set of fundamental results that are most relevant to the material in the subsequent chapters, and by no means thoroughly covers all important aspects. General sources of complementary references and a more detailed coverage include [2], [5] for general matrix theory, [4] for functions of matrices, and [3], [9] for floating-point arthmetic.

2.1 MATRIX THEORY

We assume that the reader is familiar with the fundamental concepts of linear algebra and with basic matrix operations, such as matrix transpose, matrix multiplication, and matrix inverse.

Matrices and vectors. Let $\mathbb{F}$ be a field [1, Def. 3.2.2] and m and n be positive integers. A *matrix* is an m-by-n array of scalars from $\mathbb{F}$. If $m = n$, the matrix is said to be *square*. The set of all m-by-n matrices over $\mathbb{F}$ is denoted by $\mathbb{F}^{m\times n}$. Matrices in $\mathbb{F}^{1\times n}$ and $\mathbb{F}^{m\times 1}$ are called *row* and *column vectors*, respectively. In the latter case we often write $\mathbb{F}^{m\times 1}$ as $\mathbb{F}^{m}$ as their are identical. Throughout the thesis, we focus on square matrices with the underlying field being either the real numbers $\mathbb{R}$ or the complex numbers $\mathbb{C}$. We typically denote matrices by capital letters, and their elements by doubly subscripted lowercase letters.

Block matrices. A *block matrix* (or *partitioned matrix*) is a matrix whose elements are themselves matrices, which are called *submatrices*. There are different ways of partitioning a matrix, but we are most interested in the case where the rows and columns follow the same partitioning so the submatrices along the diagonal are square.

Special matrices. A *zero matrix* is a matrix all of whose entries are zero, and the zero matrix of size $m \times n$ is denoted by $0_{m\times n}$ or 0 if the dimension is clear from the context. A matrix $A \in \mathbb{F}^{n\times n}$ is a *diagonal matrix* if $a_{ij} = 0$ when $i \neq j$, and, if, additionally, $a_{ii} = 1$ for $i = 1, 2, \dots, n$, the matrix is the *identity matrix* of size n, denoted by I_n or just I if the dimension is obvious. If $a_{ij} = 0$ when $i > j$ or $i < j$, the matrix A is *upper triangular* or *lower triangular*, respectively; the definitions extend to non-square matrix $A \in \mathbb{F}^{m\times n}$, and if $a_{ij} = 0$ when $i > j$ or $i < j$, the matrix A is *upper trapezoidal* or *lower trapezoidal*, respectively. Block diagonal, block triangular, and block trapezoidal matrices can be defined analogously, by replacing elements with blocks in the definitions above. A block triangular matrix $A \in \mathbb{F}^{n\times n}$ is *quasi-triangular* if its diagonal blocks have size at most 2.

A matrix $A \in \mathbb{R}^{n\times n}$ is *symmetric* if $A^T = A$ and *skew-symmetric* if $A^T = -A$, and likewise, a matrix $A \in \mathbb{C}^{n\times n}$ is *Hermitian* if $A^* = A$ and *skew-Hermitian* if $A^* = -A$. A matrix $A \in \mathbb{C}^{n\times n}$ is *normal* if $AA^* = A^*A$. An *orthogonal* matrix $Q \in \mathbb{R}^{n\times n}$ satisfies $QQ^T = Q^TQ = I$, and likewise, a *unitary* matrix $U \in \mathbb{C}^{n\times n}$ satisfies $UU^* = U^*U = I$. Note that, among real matrices, all symmetric, skew-symmetric, and orthogonal matrices are normal, and likewise, among complex matrices, all Hermitian, skew-Hermitian, and unitary matrices are normal.

A Hermitian matrix $A \in \mathbb{C}^{n\times n}$ is *positive definite* if $x^*Ax > 0$ for all $x \in \mathbb{C}^n\backslash\{0\}$ and *positive semidefinite* if $x^*Ax \geqslant 0$ for all $x \in \mathbb{C}^n$. Definitions for a symmetric matrix to be positive definite and positive semidefinite follow similarly.

Eigenvalues and eigenvectors. Let $A \in \mathbb{C}^{n\times n}$. If there is a $\lambda \in \mathbb{C}$ such that

$$Ax = \lambda x, \quad x \in \mathbb{C}^n\backslash\{0\}, \tag{2.1}$$

then λ is called an *eigenvalue* of A, x is an *eigenvector* of A associated with λ, and the pair (λ, x) is an *eigenpair* of A. The set of eigenvalues of A, denoted by $\Lambda(A)$, is called the *spectrum* of A. The *spectral radius* of A is $\rho(A) = \max\{|\lambda| : \lambda \in \Lambda(A)\}$.

We can rewrite (2.1) in the form $(\lambda I - A)x = 0$, $x \neq 0$ showing that $\lambda I - A$ is a singular matrix and hence any eigenvalue λ must satisfy $\det(\lambda I - A) = 0$, which is called the *characteristic equation*. The left-hand side $\det(\lambda I - A) =: p(\lambda)$ is a polynomial of degree n in λ and is known as the *characteristic polynomial* of A. The Cayley–Hamilton theorem states that every complex square matrix satisfies its own characteristic equation, that is, $p(A) = 0$. However, there can be some other polynomial q of lower degree such that $q(A) = 0$; we call the unique monic polynomial ψ of lowest degree such that $\psi(A) = 0$ the *minimal polynomial* of A.

Two matrices $A, B \in \mathbb{C}^{n\times n}$ are *similar* if there exists a nonsingular matrix $P \in \mathbb{C}^{n\times n}$ such that $B = P^{-1}AP$, where P is the *transforming matrix*, and it can be shown that similar matrices have the same spectrum. If a matrix $A \in \mathbb{C}^{n\times n}$ is similar to a diagonal matrix then A is said to be *diagonalizable*. It can be shown that $A \in \mathbb{C}^{n\times n}$ is diagonalizable if and only if A has n linearly independent eigenvectors. This implies that a matrix with distinct eigenvalues is diagonalizable because eigenvectors associated with different eigenvalues are linearly independent.

Matrix norms. A *norm* is a function $\|\cdot\|\colon \mathbb{C}^{m\times n} \to \mathbb{R}$ satisfying the following conditions:

(i) $\|A\| \geqslant 0$ for all $A \in \mathbb{C}^{m\times n}$, with $\|A\| = 0$ if and only if $A = 0$;

(ii) $\|\alpha A\| = |\alpha|\|A\|$ for all $\alpha \in \mathbb{C}$, $A \in \mathbb{C}^{m\times n}$;

(iii) $\|A + B\| \leqslant \|A\| + \|B\|$ for all $A, B \in \mathbb{C}^{m\times n}$.

We refer to a *vector norm* if the argument of the norm is a vector, and to a *matrix norm* if the argument is a matrix. For the case $n = 1$, the most relevant is the *vector p-norm*, which is defined, for $p \geqslant 1$, by

$$\|x\|_p = \left(\sum_{i=1}^{m} |x_i|^p\right)^{1/p}, \quad x \in \mathbb{C}^m,$$

and, in particular, $\|x\|_\infty = \max_{1\leqslant i\leqslant m} |x_i|$.

An important class of matrix norms is the *operator norms* (often called *subordinate* or *induced matrix norms*); given a vector norm $\|\cdot\|\colon \mathbb{C}^m \to \mathbb{R}$, the corresponding operator norm on $\mathbb{C}^{m\times n}$ is defined by

$$\|A\| = \max_{x\in\mathbb{C}^m\setminus\{0\}} \frac{\|Ax\|}{\|x\|} = \max_{x\in\mathbb{C}^m, \|x\|=1} \|Ax\|.$$

When the vector norm is the p-norm then the subordinate matrix norm $\|A\|_p = \max_{x\in\mathbb{C}^m\setminus\{0\}} \|Ax\|_p/\|x\|_p$ is called the *matrix p-norm*. For the 1, 2, and ∞ vector norms it can be shown that

$$\begin{aligned}
\|A\|_1 &= \max_{1\leqslant j\leqslant n} \sum_{i=1}^{m} |a_{ij}|,\\
\|A\|_2 &= \rho(A^*A)^{1/2},\\
\|A\|_\infty &= \max_{1\leqslant i\leqslant m} \sum_{j=1}^{n} |a_{ij}|.
\end{aligned}$$

An example of matrix norm that is not induced by a vector norm yet very useful in numerical linear algebra is the *Frobenius norm*, which is defined, for $A \in \mathbb{C}^{m\times n}$, by

$$\|A\|_F = \left(\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2\right)^{1/2}.$$

All norms on $\mathbb{C}^{m\times n}$ are *equivalent*: for any two norms $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$, we have that $\nu_1\|A\|_\alpha \leqslant \|A\|_\beta \leqslant \nu_2\|A\|_\alpha$ for some $\nu_1, \nu_2 > 0$, for all matrices $A \in \mathbb{C}^{m\times n}$. A norm is *consistent* if it has the property of sub-multiplicativity, namely, it satisfies $\|AB\| \leqslant \|A\|\|B\|$ for all $A \in \mathbb{C}^{m\times n}$ and $B \in \mathbb{C}^{n\times p}$. The Frobenius norm and all subordinate norms are consistent. It is easy to show that, for $A \in \mathbb{C}^{n\times n}$, one has that $\rho(A) \leqslant \|A\|$ for any consistent norm.

For nonsingular $A \in \mathbb{C}^{n\times n}$ and any matrix norm, the quantity $\kappa(A) = \|A\|\|A^{-1}\|$ is the *normwise condition number* (with respect to inversion) of A; for any subordinate matrix norm $\|\cdot\|$ we have $\|A\|\|A^{-1}\| \geqslant \|AA^{-1}\| = \|I\| = 1$ and hence $\kappa(A) \geqslant 1$. The condition number can be arbitrarily large, and if A is singular we regard the condition number as infinite. We denote by $\kappa_F(A)$ the condition number with respect to the Frobenius norm, and by $\kappa_p(A)$ that with respect to the operator norm induced by the vector p-norm. It is not hard to see from the definitions that if $A \in \mathbb{C}^{n\times n}$ is unitary then $\kappa_2(A) = 1$, which shows that unitary matrices are a class of extremely well-conditioned matrix.

Jordan canonical form. Any matrix $A \in \mathbb{C}^{n\times n}$ with p linearly independent eigenvectors is similar to a block diagonal matrix

$$J = X^{-1}AX = \operatorname{diag}(J_1, J_2, \dots, J_p), \tag{2.2}$$

where $X \in \mathbb{C}^{n\times n}$ is nonsingular, the diagonal blocks are called *Jordan blocks* and are upper triangular matrix of the form

$$J_k := J_k(\lambda_k) := \begin{bmatrix} \lambda_k & 1 & & \\ & \lambda_k & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_k \end{bmatrix} \in \mathbb{C}^{m_k\times m_k},$$

and $m_1 + m_2 + \cdots + m_p = n$. The matrix J is called the *Jordan canonical form* of A and is unique up to the ordering of the blocks J_k, but the transforming matrix X is not unique. If A is diagonalizable then the Jordan canonical form reduces to an *eigendecomposition* $A = XDX^{-1}$, with $D = \operatorname{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ and the columns of X being eigenvectors of A.

The Jordan canonical form is an invaluable tool from a theoretical standpoint despite the fact that it cannot be reliably computed in finite precision arithmetic except in special cases such as when A is Hermitian or normal. It is the basis for an elegant definition of matrix functions and provides a concrete way to prove and understand many results.

QR factorization. For every $A \in \mathbb{C}^{m\times n}$, there exist a unitary matrix $Q \in \mathbb{C}^{m\times m}$ and an upper trapezoidal matrix $R \in \mathbb{C}^{m\times n}$ such that $A = QR$. This factorization is referred to as the *QR factorization* of A, and if $m \geqslant n$ and A has full column rank, by partitioning

$$\begin{aligned} Q &= \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}, & Q_1 &\in \mathbb{C}^{m\times n}, & Q_2 &\in \mathbb{C}^{m\times(m-n)}, \\ R &= \begin{bmatrix} R_1 \\ 0 \end{bmatrix}, & R_1 &\in \mathbb{C}^{n\times n}, & 0 &\in \mathbb{C}^{(m-n)\times n}, \end{aligned}$$

then $A = Q_1R_1$ is called the *reduced QR factorizatoin* or the *thin QR factorization*. If A is real, then the factors Q and R may be taken to be real.

There are three standard means of computing a QR factorization, which are via Householder reflections, Givens rotations, and Gram–Schmidt orthogonalization; see [2, sect. 5.2] for details.

Schur decomposition. Let $A \in \mathbb{C}^{n\times n}$. Then there exists a unitary matrix $U \in \mathbb{C}^{n\times n}$ and an upper triangular matrix $T \in \mathbb{C}^{n\times n}$ such that

$$T = U^{-1}AU = U^*AU,$$

that is, $A = UTU^*$, which is called a *Schur decomposition* of A. The Schur decomposition is not unique as the eigenvalues of T can be made to appear in any order on the diagonal by choosing different transforming matrix U.

For $A \in \mathbb{R}^{n\times n}$ we can restrict the transforming matrix to be real and obtain a real analogue of the Schur decomposition, the *real Schur decomposition* $A = QTQ^T$, where $Q \in \mathbb{R}^{n\times n}$ is orthogonal and $T \in \mathbb{R}^{n\times n}$ is upper quasi-triangular with each of the diagonal blocks being either 1×1 or 2×2 with complex conjugate eigenvalues. The matrices A and T have the same spectrum as they are similar; for the Schur decomposition the diagonal elements of T are the eigenvalues of A, and, for the real Schur decomposition, any diagonal block of size 1×1 is a real eigenvalue of A and the eigenvalues of a 2×2 diagonal block of T coincide with a complex conjugate pair of eigenvalues of A.

For normal matrices Schur decomposition reduces to *spectral decomposition*, derived from the spectral theorem: $A \in \mathbb{C}^{n\times n}$ is normal if and only if there exists a unitary matrix $U \in \mathbb{C}^{n\times n}$ and a diagonal matrix $D \in \mathbb{C}^{n\times n}$ such that $A = UDU^{-1} = UDU^*$. It follows immediately that a normal matrix has n orthonormal eigenvectors so its spectral decomposition is exceedingly well conditioned.

The Schur decomposition of a dense general matrix can be computed with perfect backward stability via the QR algorithm and its variants [2, sect. 7.5], and hence it is a standard tool in numerical linear algebra.

SVD decomposition. For any $A \in \mathbb{C}^{m\times n}$ there exists unitary matrices $U \in \mathbb{C}^{m\times m}$ and $V \in \mathbb{C}^{n\times n}$ such that

$$A = U\Sigma V^*, \quad \Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \dots, \sigma_p) \in \mathbb{R}^{m\times n}, \quad p = \min(m,n),$$

where $\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_p \geqslant 0$. This is called the *SVD decomposition* of the matrix A. The σ_i are the *singular values* of A, and the columns of U and V are called the left and right *singular vectors* of A, respectively. If A is real, U and V can be taken to be real.

By partitioning, for $m \geqslant n$,

$$\begin{aligned}
U &= \begin{bmatrix} U_1 \; U_2 \end{bmatrix}, && U_1 \in \mathbb{C}^{m\times n}, \quad U_2 \in \mathbb{C}^{m\times(m-n)},\\
\Sigma &= \begin{bmatrix} \Sigma_1 \\ 0 \end{bmatrix}, && \Sigma_1 \in \mathbb{R}^{n\times n}, \quad 0 \in \mathbb{R}^{(m-n)\times n},\\
V &= V_1,
\end{aligned}$$

and for $m < n$,

$$\begin{aligned}
U &= U_1,\\
\Sigma &= \begin{bmatrix} \Sigma_1 \; 0 \end{bmatrix}, && \Sigma_1 \in \mathbb{R}^{m\times m}, \quad 0 \in \mathbb{R}^{m\times(n-m)},\\
V &= \begin{bmatrix} V_1 \; V_2 \end{bmatrix}, && V_1 \in \mathbb{C}^{m\times n}, \quad V_2 \in \mathbb{C}^{(n-m)\times n},
\end{aligned}$$

we obtain an abbreviated version of the SVD, $A = U_1\Sigma_1 V_1^*$, which is referred to as the *thin SVD decomposition*. The rank of A is equal to the number of its nonzero singular values. If the matrix A is not full rank, the thin SVD decomposition can be further reduced by dropping the left and right singular vectors corresponding to zero singular values and we arrive at the *compact SVD decomposition* which contains the essential SVD information: $A = U\Sigma V^*$, where $U \in \mathbb{C}^{m\times r}$, $\Sigma = \operatorname{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r) \in \mathbb{R}^{r\times r}$, $V \in \mathbb{C}^{n\times r}$, and $r = \operatorname{rank}(A)$.

Algorithms for computing the SVD decomposition are discussed in [2, sect. 8.6].

Sherman–Morrison–Woodbury formula. For any nonsingular matrix $A \in \mathbb{C}^{n\times n}$ if $U, V \in \mathbb{C}^{n\times k}$ and $I + V^*A^{-1}U$ is nonsingular then $A + UV^*$ is nonsingular and

$$(A + UV^*)^{-1} = A^{-1} - A^{-1}U(I + V^*A^{-1}U)^{-1}V^*A^{-1},$$

which is the *Sherman–Morrison–Woodbury* formula. The matrix $I + V^*A^{-1}U$ is $k \times k$, so if $k \ll n$ and A^{-1} is already known then this formula provides a cheap way to evaluate the inverse of A corrected by the matrix UV^*, which has rank at most k.

The *Sherman–Morrison–Woodbury* formula generalizes the well-known *Sherman–Morrison* formula, which computes the inverse of A perturbed by a rank-1 matrix (the case $k = 1$).

2.2 FUNCTIONS OF MATRICES

The concept of *functions of matrices* can have different meaning in the literature. Common functions with a matrix input include the rank, the determinant, the spectral radius, and the norms of a matrix. Within the scope of this thesis we study functions of matrices that stem from a scalar function and map $\mathbb{C}^{n\times n}$ to $\mathbb{C}^{n\times n}$.

Matrix function via Jordan canonical form. Let A have the Jordan canonical form (2.2) and let f be defined on the spectrum of $A \in \mathbb{C}^{n\times n}$, that is, for distinct eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_s$ of A the values $f^{(j)}(\lambda_i), j = 0 \colon n_i - 1, i = 1 \colon s$ exist, where

n_i, which is called the *index* of λ_i, is the order of the largest Jordan block in which λ_i appears. Then

$$f(A) := Zf(J)Z^{-1} = Z\,\mathrm{diag}(f(J_1), f(J_2), \dots, f(J_p)))Z^{-1},$$

where

$$f(J_k) := \begin{bmatrix} f(\lambda_k) & f'(\lambda_k) & \dots & \frac{f^{(m_k-1)}(\lambda_k)}{(m_k-1)!} \\ & f(\lambda_k) & \ddots & \vdots \\ & & \ddots & f'(\lambda_k) \\ & & & f(\lambda_k) \end{bmatrix}. \tag{2.3}$$

For multivalued functions such as the square root and logarithm it is implicit in the above definition that a single branch has been chosen in (2.3). Furthermore, if A has a repeated eigenvalue occurring in more than one Jordan block, we will choose the same branch in each block; the resulting function is called a *primary* matrix functrion, which is the one that appears in most applications and is what the thesis is exclusively concerned with. If a different choice of branch is made for the same eigenvalue in two different Jordan blocks then a *nonprimary* matrix function is obtained; see [4, sect. 1.4] for more on nonprimary matrix functions.

The definition yields an $f(A)$ that is independent of the particular Jordan canonical form that is chosen. An intuition we get from the Jordan canonical form definition of $f(A)$ is that f needs to be sufficiently differentiable on the spectrum of A and the differentiability requirement relies on the Jordan structure of A: an eigenvalue that appears in a larger Jordan block needs higher derivatives of f to be defined at the point.

Matrix function via Hermite interpolation. Let f be defined on the spectrum of $A \in \mathbb{C}^{n\times n}$ and let ψ be the minimal polynomial of A. Then $f(A) := p(A)$, where p is the polynomial of degree less than $\sum_{k=1}^{s} n_i$ (namely, the degree of the minimal polynomial) that satisfies the interpolation conditions

$$p^{(j)}(\lambda_i) = f^{(j)}(\lambda_i), \quad j = 0 : n_i - 1, \quad i = 1 : s.$$

There is a unique such p and it is known as the *Hermite interpolating polynomial*.

It follows immediately from the definition that $f(A)$ is a polynomial in A that is completely determined by the spectrum of A. It is worth noting, however, that the polynomial *p depends on A*, so we do not have $f(A) \equiv q(A)$ for some fixed polynomial q in dependent of A.

There are alternative means of defining $f(A)$ in addition to the two mentioned above, such as the Cauchy integral definition, and it has been shown that these definitions are equivalent, modulo the requirement on analyticity of f in the Cauchy integral definition; we refer the readers to [4, sect. 1.2] for details.

Matrix Taylor series. Suppose f has a Taylor series expansion

$$f(z) = \sum_{k=0}^{\infty} a_k (z-\alpha)^k, \quad a_k = \frac{f^{(k)}(\alpha)}{k!}$$

with radius of convergence r. If $A \in \mathbb{C}^{n\times n}$ then $f(A)$ is defined and is given by

$$f(A) = \sum_{k=0}^{\infty} a_k (A-\alpha I)^k$$

if and only if each of the distinct eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_s$ of A satisfies one of the conditions [4, Thm. 4.7]

(i) $|\lambda_i - \alpha| < r$,

(ii) $|\lambda_i - \alpha| = r$ and the series for $f^{(n_i-1)}(\lambda)$ (where n_i is the index of λ_i) is convergent at the point $\lambda = \lambda_i$, $i = 1\colon s$.

Functions of matrices therefore can be defined everywhere from the Taylor series if the respective functions have a Taylor series with an infinite radius of convergence. Hence we can define, for example, the matrix cosine and sine by

$$\cos A = I - \frac{A^2}{2!} + \frac{A^4}{4!} - \frac{A^6}{6!} + \cdots,$$

$$\sin A = A - \frac{A^3}{3!} + \frac{A^5}{5!} - \frac{A^7}{7!} + \cdots.$$

The Taylor series representation of matrix functions is a useful tool for approximating matrix functions applicable to general functions, by summing a suitable finite number of terms.

Basic properties of matrix functions. Let $A \in \mathbb{C}^{n\times n}$ and let f be defined on the spectrum of A. Then [4, Thm. 1.13]

(i) $f(A)$ commutes with A;

(ii) $f(A^T)$=$f(A)^T$;

(iii) $f(XAX^{-1}) = Xf(A)X^{-1}$;

(iv) the eigenvalues of $f(A)$ are $f(\lambda_i)$, where the λ_i are the eigenvalues of A;

(v) if X commutes with A then X commutes with $f(A)$;

(vi) if $A = (A_{ij})$ is block triangular then $F = f(A)$ is block triangular with the same block structure as A, and $F_{ii} = f(A_{ii})$;

(vii) if $A = \operatorname{diag}(A_{11}, A_{22}, \ldots, A_{mm})$ is block diagonal then

$$f(A) = \operatorname{diag}(f(A_{11}), f(A_{22}), \ldots, f(A_{mm})).$$

The above properties follow from the definitions of $f(A)$ or can be easily proved, and will be employed repeatedly throughout the thesis.

Fréchet derivatives. The *Fréchet derivative* of a matrix function f at $A \in \mathbb{C}^{n\times n}$ is a linear mapping $L_f(A, \cdot)\colon \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ such that

$$f(A+E) - f(A) - L_f(A,E) = o(\|E\|) \tag{2.4}$$

for all $E \in \mathbb{C}^{n\times n}$. The notation $L_f(A,E)$ can be read as "the Fréchet derivative of f at A in the direction E". The Fréchet derivative may not exist, but if it does then it is unique and we say f is *Fréchet differentiable*.

Like the scalar derivative, the Fréchet derivative satisfies sum, product, and chain rules. If g and h are Fréchet differentiable at A then [4, Thm. 3.2–3.4]

(i) $L_{\alpha g+\beta h}(A,E) = \alpha L_g(A,E) + \beta L_h(A,E)$ for all $\alpha,\beta \in \mathbb{C}$;

(ii) $L_{gh}(A,E) = L_g(A,E)h(A) + g(A)L_h(A,E)$;

(iii) $L_{g\circ h}(A,E) = L_g(h(A), L_h(A,E))$ if g is Fréchet differentiable at $h(A)$.

We also point out that if $A = UTU^*$ is a Schur decomposition then $L_f(A,E) = UL_f(T,U^*EU)U^*$ [4, Prob. 3.2].

Condition numbers. Let $A \in \mathbb{C}^{n\times n}$ and let f be defined in a neighbourhood of A. Then the *relative condition number* is defined by, for any matrix norm,[1]

$$\operatorname{cond}(f,A) := \lim_{\epsilon\to 0} \sup_{\|E\|\leqslant\epsilon\|A\|} \frac{\|f(A+E)-f(A)\|}{\epsilon\|f(A)\|}.$$

This definition implies that

$$\frac{\|f(A+E)-f(A)\|}{\|f(A)\|} \leqslant \operatorname{cond}(f,A)\frac{\|E\|}{\|A\|} + o(\|E\|),$$

which is valid for $A+E$ in the neighbourhood where f is defined, and so provides a first order perturbation bound for small perturbations E. An *absolute condition number*, in which the change in the data and the function are measured in an absolute sense, can be defined correspondingly, but often we are more interested in measuring the sensitivity in the relative sense.

The condition number of f is essentially the norm of the Fréchet derivative (when it exists) as it can be expressed as [4, Thm. 3.1]

$$\operatorname{cond}(f,A) = \frac{\|L_f(A)\|\|A\|}{\|f(A)\|},$$

where

$$\|L_f(A)\| := \max_{Z\in\mathbb{C}^{n\times n}\setminus\{0\}} \frac{\|L_f(A,Z)\|}{\|Z\|} = \max_{Z\in\mathbb{C}^{n\times n}, \|Z\|=1} \|L_f(A,Z)\|.$$

1 The definition follows the style of Rice [11].

2.3 FLOATING-POINT ARITHMETIC

Floating-point arithmetic is thus far the most widely used way of approximately representing real-number arithmetic for performing numerical computations in modern computers. In this section we recall the floating-point number system and the basic model of arithmetic that underlies rounding error analysis, with particular attention drawn to the IEEE arithmetics and arbitrary precision arithmetic.

Floating-point number system. A floating-point number system $\mathcal{F}$ is a finite subset of the real numbers $\mathbb{R}$ comprising numbers of the form

$$y = (-1)^s \cdot m \cdot \beta^{e-p+1},$$

where $\beta \in \mathbb{N}\backslash\{0, 1\}$ is the *base* (or *radix*), $p \in \mathbb{N}\backslash\{0, 1\}$ is the *precision*, $m \in \mathbb{N}$ is the *significand* satisfying $0 \leqslant m \leqslant \beta^p - 1$, $e \in \mathbb{N}$ is the *exponent*, which lies in the range $e_{\min} \leqslant e \leqslant e_{\max}$, and the *sign bit* s is 0 for $y \geqslant 0$ or 1 for $y < 0$. The system is fully characterized by the four integer parameters β, p, $e_{\min}$, and $e_{\max}$.

To ensure a unique representation for each nonzero $y \in \mathcal{F}$, it is assumed that if $y > \beta^{e_{\min}-p+1}$ then $\beta^{p-1} \leqslant m \leqslant \beta^p - 1$, so that the system is *normalized*. In such systems, a nonzero number $y \in \mathcal{F}$ is *normal* if $m \geqslant \beta^{p-1}$, and those with $0 < m < \beta^{p-1}$ and $e = e_{\min}$ are called *subnormal numbers* (or *denormalized numbers*), which have the minimal exponent and fewer than p digits of precision. The number 0 is special in that it does not have a normalized representation. We also note that any nonzero $y \in \mathcal{F}$ can be alternatively expressed as

$$y = (-1)^s \cdot \left(d_0 + \frac{d_1}{\beta} + \cdots + \frac{d_{p-1}}{\beta^{p-1}}\right) \cdot \beta^e,$$

where each *digit* d_i satisfies $0 \leqslant d_i \leqslant \beta - 1$ and $d_0 \neq 0$ for normalized numbers.

The largest positive number in the system is $y_{\max} = \beta^{e_{\max}}(\beta - \beta^{1-p})$, while the smallest positive normalized and subnormal numbers are $y_{\min} = \beta^{e_{\min}}$ and $y^s_{\min} = \beta^{e_{\min}-p+1}$, respectively, the latter being the smallest nonzero *representable* number. Note that $\mathcal{F}$

is a subset of the closed interval $[-y_{\max}, y_{\max}]$, which is called the *range* of $\mathcal{F}$. Two other important quantities are $u = \frac{1}{2}\beta^{1-p}$, the *unit roundoff*, and $\epsilon = \beta^{1-p}$, the *machine epsilon*, which is the distance from 1 to the next larger floating-point number.

It is in general highly desirable to have a *closed* floating-point number system, by the inclusion of two special floating-point numbers: infinity (∞) and NaN (Not a Number), so every arithmetic operation is well defined in the system; any number lies out of the range of the system is represented by a signed infinity, and NaN is used for representing the result of invalid operations over the real numbers, such as dividing any numbers by 0.

Rounding. Any real number x can be mapped into $\mathcal{F}$ by *rounding*, and the operator that performs this mapping is denoted by $\mathrm{fl}(\cdot)$. We say that $\mathrm{fl}(x)$ *overflows* if $|\mathrm{fl}(x)| > y_{\max}$ and *underflows* if $x \neq 0$ and $\mathrm{fl}(x) = 0$. When subnormal numbers are included in $\mathcal{F}$, underflow is said to be *gradual*.

We are exclusively interested in the cases where $\mathrm{fl}(x)$ does not overflow or underflow for the input x. If $x \in \mathcal{F}$ then $\mathrm{fl}(x) = x$; otherwise, x lies between two floating-point numbers and there are many possible rounding functions. It is customary to take $\mathrm{fl}(x)$ to be the element in $\mathcal{F}$ nearest to $x \in \mathbb{R}$ in absolute value, and this rule is known as *round to nearest*. If x is equidistant from two floating-point numbers, there are several usual ways to break ties, including rounding to the number with an even last digit; see [3, sect. 2.12] for more discussion on tie-breaking strategies. Note that $\mathrm{fl}(x)$ with round to nearest is monotonic: $x_1 \geqslant x_2$ implies $\mathrm{fl}(x_1) \geqslant \mathrm{fl}(x_2)$.

For round to nearest it can be shown that if $x \in \mathbb{R}$ lies in the range of $\mathcal{F}$ then [3, Thm. 2.2]

$$\mathrm{fl}(x) = x(1+\delta), \quad |\delta| < u. \tag{2.5}$$

This result shows that, for every real number x lying in the range of $\mathcal{F}$, round to nearest introduces a relative error no larger than the unit roundoff.

Model of arithmetic. The standard model of floating-point arithmetic [3, sect. 2.2] states that, for $x, y \in \mathcal{F}$,

$$\mathrm{fl}(x \text{ op } y) = (x \text{ op } y)(1+\delta), \quad |\delta| \leqslant u, \quad \text{op} = +, -, *, /. \tag{2.6}$$

It is normal to assume that (2.6) holds also for the square root operation. The four elementary operations of addition, subtraction, multiplication, and division of floating-point numbers are known as *floating-point operations* (flops). We will use the total number of flops required by a numerical algorithm to measure its algorithmic complexity.

Errors caused by the potentially inexact representation of real numbers in $\mathcal{F}$ and the arithmetic operations performed with floating-point numbers are referred to as *rounding errors*. The accumulation of these errors in the standard floating-point model (2.6) is unavoidably due to the use of *finite* precision arithmetic, and is the subject of research in *rounding error analysis*.

IEEE arithmetics. The IEEE Standard 754, published in 1985 [6] and revised in 2008 [7] and 2019 [8], is a standard for binary and decimal floating-point arithmetics, which specifies floating-point number formats and precise rules for carrying out arithmetic on them. The standard for decimal arithmetic was included with the binary standard from the 2008 revision, but here we focus on the binary part as decimal formats are currently of limited practical interest.

The widely used IEEE standard binary arithmetic has $\beta = 2$ and now prescribes four precisions. The 32-bit *single precision* format (binary32) has $p = 24$, $e_{\min} = -126$, and $e_{\max} = 127$. The 64-bit *double precision* format (binary64) has $p = 53$, $e_{\min} = -1022$, and $e_{\max} = 1023$. The 128-bit *quadruple precision* format (binary128) and the 16-bit *half precision* format (binary16), which are introduced in the 2008 revision of the standard, have $p = 113$, $e_{\min} = -16382$, and $e_{\max} = 16383$ and $p = 11$, $e_{\min} = -14$, and $e_{\max} = 15$, respectively, the latter defined only as a storage format despite its broad use for computation. The key parameters for these formats are summarized in Table 2.1.

Table 2.1: Parameters for four IEEE floating-point arithmetics: number of bits in significand (including implicit most significant bit) and exponent (sig, exp); unit roundoff u; smallest positive (subnormal) number $y^s_{\min}$; smallest positive normalized number $y_{\min}$; and largest finite number $y_{\max}$. The last four columns are given to three significant figures.

	(sig, exp)	u	$y^s_{\min}$	$y_{\min}$	$y_{\max}$
binary16	$(11, 5)$	4.88×10^{-4}	5.96×10^{-8}	6.10×10^{-5}	6.55×10^{4}
binary32	$(24, 8)$	5.96×10^{-8}	1.40×10^{-45}	1.11×10^{-38}	3.40×10^{38}
binary64	$(53, 11)$	1.11×10^{-16}	4.94×10^{-324}	2.22×10^{-308}	1.80×10^{308}
binary128	$(113, 15)$	9.63×10^{-35}	6.48×10^{-4966}	3.36×10^{-4932}	1.19×10^{4932}

These IEEE floating-point formats have fixed parameters and are highly standardized, which enables highly optimized hardware implementations of the logic circuits that operate on these numbers as well as fine-tuned numerical algorithms to be designed for a specific working precision. This feature, however, can be undesirable in some cases when a flexible computational environment is needed.

Arbitrary precision arithmetic. In *arbitrary* precision arithmetic the user is allowed to prescribe the precision of each floating-point operation, where floating-point calculations are performed on numbers whose digits of precision are limited only by the available memory of the host system. This is made possible through software libraries and contrasts with the potentially much faster fixed precision floating-point arithmetics that are usually implemented in hardware. It is for this reason that arbitrary precision arithmetic is mainly of interest in applications where one is willing to sacrifice some computational efficiency in exchange for the ability to compute with any number of digits.

In the following chapters, we simulate arbitrary precision floating-point arithmetic by MATLAB with the Multiprecision Computing Toolbox [10], where the required precision is specified by the user in terms of decimal digits.

REFERENCES

[1] M. Artin. *Algebra*. Upper Saddle River, NJ, USA: Prentice-Hall, 1991 (cited on p. 26).

[2] G. H. Golub and C. F. Van Loan. *Matrix Computations*. 4th ed., Baltimore, MD, USA: Johns Hopkins University Press, 2013, pp. xxi+756 (cited on pp. 26, 31–33).

[3] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. 2nd ed., Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. xxx+680 (cited on pp. 26, 39, 40).

[4] N. J. Higham. *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008, pp. xx+425 (cited on pp. 26, 34–37).

[5] R. A. Horn and C. R. Johnson. *Matrix Analysis*. 2nd ed., Cambridge University Press, 2012, pp. xiii+662 (cited on p. 26).

[6] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. IEEE, 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987 (cited on p. 40).

[7] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (revision of IEEE Std 754-1985)*. IEEE, 2008, p. 58 (cited on p. 40).

[8] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (revision of IEEE Std 754-2008)*. IEEE, 2019, p. 84 (cited on p. 40).

[9] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic*. 2nd ed., Boston, MA, USA: Birkhäuser, 2018, pp. xxv+627 (cited on p. 26).

[10] *Multiprecision Computing Toolbox*. Advanpix, Tokyo, Japan. `http://www.advanpix.com` (cited on p. 41).

[11] J. R. Rice. "A theory of condition." *SIAM J. Numer. Anal.* 3.2 (1966), pp. 287–310 (cited on p. 37).

3 A MULTIPRECISION DERIVATIVE-FREE SCHUR–PARLETT ALGORITHM FOR COMPUTING MATRIX FUNCTIONS

Abstract. The Schur–Parlett algorithm, implemented in MATLAB as `funm`, evaluates an analytic function f at an $n \times n$ matrix argument by using the Schur decomposition and a block recurrence of Parlett. The algorithm requires the ability to compute f and its derivatives, and it requires that f has a Taylor series expansion with a suitably large radius of convergence. We develop a version of the Schur–Parlett algorithm that requires only function values and not derivatives. The algorithm requires access to arithmetic of a matrix-dependent precision at least double the working precision, which is used to evaluate f on the diagonal blocks of order greater than 2 (if there are any) of the reordered and blocked Schur form. The key idea is to compute by diagonalization the function of a small random diagonal perturbation of each diagonal block, where the perturbation ensures that diagonalization will succeed. Our algorithm is inspired by Davies's randomized approximate diagonalization method, but we explain why that is not a reliable numerical method for computing matrix functions. This multiprecision Schur–Parlett algorithm is applicable to arbitrary analytic functions f and, like the original Schur–Parlett algorithm, it generally behaves in a numerically stable fashion. The algorithm is especially useful when the derivatives of f are not readily available or accurately computable. We apply our algorithm to the matrix Mittag–Leffler function and show that it yields results of accuracy similar to, and in some cases much greater than, the state-of-the-art algorithm for this function.

Keywords: multiprecision algorithm, multiprecision arithmetic, matrix function, Schur decomposition, Schur–Parlett algorithm, Parlett recurrence, randomized approximate diagonalization, matrix Mittag–Leffler function.

2010 MSC: 65F60.

3.1 INTRODUCTION

In this work we are concerned with functions mapping $\mathbb{C}^{n\times n}$ to $\mathbb{C}^{n\times n}$ that are defined in terms of an underlying scalar function f and a Cauchy integral formula, a Hermite interpolating polynomial, or the Jordan canonical form (see, for example, [24, Chap. 1] for details). We assume that f is analytic on a closed convex set whose interior contains the eigenvalues of the matrix argument. The need to compute such matrix functions arises in numerous applications in science and engineering. Specialized methods exist for evaluating particular matrix functions, including the scaling and squaring algorithm for the matrix exponential [1], [34], Newton's method for matrix sign function [24, Chap. 5], [39], and the inverse scaling and squaring method for the matrix logarithm [2], [32]. See [28] for a survey and [29] for links to software for these and other methods. For some functions a specialized method is not available, in which case a general purpose algorithm is needed. The Schur–Parlett algorithm [11] computes a general function f of a matrix, with the function dependence restricted to the evaluation of f on the diagonal blocks of the reordered and blocked Schur form. It evaluates f on the nontrivial diagonal blocks via a Taylor series, so it requires the derivatives of f and it also requires the Taylor series to have a sufficiently large radius of convergence. However, the derivatives are not always available or accurately computable.

We develop a new version of the Schur–Parlett algorithm that requires only the ability to evaluate f itself and can be used whatever the distribution of the eigenvalues. Our algorithm handles close or repeated eigenvalues by an idea inspired by Davies's idea of randomized approximate diagonalization [10] together with higher precision arithmetic. We therefore assume that we can compute not only at the working precision, with unit roundoff u, but also at a higher precision with unit roundoff $u_h < u$, where u_h can be arbitrarily chosen. Higher precisions will necessarily be implemented in software, and so will be expensive, but we aim to use them as little as possible.

We note that multiprecision algorithms have already been developed for the matrix exponential [14] and the matrix logarithm [15]. Those algorithms are tightly coupled to the functions in question, whereas here we place no restrictions on the function. Indeed the new algorithm greatly expands the range of functions f for which we can reliably compute $f(A)$. A numerically stable algorithm for evaluating the Lambert W function of a matrix was only recently developed [16]. Our algorithm can readily compute this function, as well as other special functions and multivalued functions for which the Schur–Parlett algorithm is not readily applicable.

In section 3.2 we review the Schur–Parlett algorithm. In section 3.3 we describe Davies's randomized approximate diagonalization and explain why it cannot be the basis of a reliable numerical algorithm. In section 3.4 we describe our new algorithm for evaluating a function of a triangular matrix using only function values. In section 3.5 we use this algorithm to build a new Schur–Parlett algorithm that requires only function values and we illustrate its performance on a variety of test problems. We apply the algorithm to the matrix Mittag–Leffler function in section 3.6 and compare it with a special purpose algorithm for this function. Conclusions are given in section 3.7.

We will write "standard Gaussian matrix" to mean a random matrix with entries independently drawn from a standard normal distribution (mean 0, variance 1). A diagonal or triangular standard Gaussian matrix has the entries on the diagonal or in the triangle drawn in the same way. We will use the Frobenius norm, $\|A\|_F = (\sum_{i,j} |a_{ij}|^2)^{1/2}$, and the p-norms $\|A\|_p = \max\{\, \|Ax\|_p : \|x\|_p = 1\,\}$, where $\|x\|_p = (\sum_i |x_i|^p)^{1/p}$. We will also need the condition number $\kappa(A) = \|A\|\|A^{-1}\|$, and we indicate the norm with a subscript on $\kappa(A)$.

3.2 SCHUR–PARLETT ALGORITHM

The Schur–Parlett algorithm [11] for computing a general matrix function $f(A)$ is based on the Schur decomposition $A = QTQ^* \in \mathbb{C}^{n\times n}$, with $Q \in \mathbb{C}^{n\times n}$ unitary and $T \in \mathbb{C}^{n\times n}$ upper triangular. Since $f(A) = Qf(T)Q^*$, computing $f(A)$ reduces to computing $f(T)$, the same function evaluated at a triangular matrix. Let $T = (T_{ij})$ be partitioned to be block upper triangular with square diagonal blocks, possibly of different sizes. Then $F = f(T)$ has the same block structure and if the function of the square diagonal blocks $F_{ii} = f(T_{ii})$ can be computed then the off-diagonal blocks F_{ij} can be obtained using the block form of Parlett's recurrence [37]. To be more specific, by equating (i, j) blocks in $TF = FT$, we obtain a triangular Sylvester equation for F_{ij},

$$T_{ii}F_{ij} - F_{ij}T_{jj} = F_{ii}T_{ij} - T_{ij}F_{jj} + \sum_{k=i+1}^{j-1} (F_{ik}T_{kj} - T_{ik}F_{kj}), \quad i < j, \tag{3.1}$$

from which F_{ij} can be computed either a block superdiagonal at a time or a block row or block column at a time. To address the potential problems caused by close or equal eigenvalues in two diagonal blocks of T, Davies and Higham [11] devised a scheme with a blocking parameter $\delta > 0$ to reorder T into a partitioned upper triangular matrix $\widetilde{T} = U^*TU = (\widetilde{T}_{ij})$ by a unitary similarity transformation such that

- eigenvalues λ and μ from any two distinct diagonal blocks $\widetilde{T}_{ii}$ and $\widetilde{T}_{jj}$ satisfy $|\lambda - \mu| > \delta$, and
- the eigenvalues of every block $\widetilde{T}_{ii}$ of size larger than 1 are well clustered in the sense that either all the eigenvalues of $\widetilde{T}_{ii}$ are equal or for every eigenvalue λ_1 of $\widetilde{T}_{ii}$ there is an eigenvalue λ_2 of $\widetilde{T}_{ii}$ with $\lambda_1 \neq \lambda_2$ such that $|\lambda_1 - \lambda_2| \leqslant \delta$.

To evaluate $f(\widetilde{T}_{ii})$, the Schur–Parlett algorithm expands f in a Taylor series about $\sigma = \text{trace}(\widetilde{T}_{ii})/m_i$, the mean of the eigenvalues of $\widetilde{T}_{ii} \in \mathbb{C}^{m_i\times m_i}$,

$$f(\widetilde{T}_{ii}) = \sum_{k=0}^{\infty} \frac{f^{(k)}(\sigma)}{k!}(\widetilde{T}_{ii} - \sigma I)^k, \tag{3.2}$$

truncating the series after an appropriate number of terms. All the derivatives of f up to a certain order are required in (3.2), where that order depends on how quickly the powers of $\widetilde{T}_{ii} - \sigma I$ decay. Moreover, for the series (3.2) to converge we need $\lambda - \sigma$ to lie in the radius of convergence of the series for every eigenvalue λ of $\widetilde{T}_{ii}$. Obviously, this procedure for evaluating $f(\widetilde{T})$ may not be appropriate if it is difficult or expensive to accurately evaluate the derivatives of f or if the Taylor series has a finite radius of convergence.

3.3 APPROXIMATE DIAGONALIZATION

If $A \in \mathbb{C}^{n\times n}$ is diagonalizable then $A = VDV^{-1}$, where $D = \operatorname{diag}(d_i)$ is diagonal and V is nonsingular, so $f(A) = Vf(D)V^{-1} = V\operatorname{diag}(f(d_i))V^{-1}$ is trivially obtained. For normal matrices, V can be chosen to be unitary and this approach is an excellent way to compute $f(A)$. However, for nonnormal A the eigenvector matrix V can be ill-conditioned, in which case an inaccurate computed $f(A)$ can be expected in floating-point arithmetic [24, sect. 4.5].

A way to handle a nonnormal matrix is to perturb it before diagonalizing it. Davies [10] suggested perturbing A to $\widetilde{A} = A + E$, computing the diagonalization $\widetilde{A} = VDV^{-1}$, and approximating $f(A)$ by $f(\widetilde{A}) = Vf(D)V^{-1}$. This approach relies on the fact that even if A is defective, $A + E$ is likely to be diagonalizable because the diagonalizable matrices are dense in $\mathbb{C}^{n\times n}$. Davies measured the quality of the approximate diagonalization by the quantity

$$\sigma(A, V, E, \epsilon) = \kappa_2(V)\epsilon + \|E\|_2, \tag{3.3}$$

where ϵ measures the accuracy of the finite precision computations and so can be taken as the unit roundoff. Minimizing over E and V (since V is not unique) gives

$$\sigma(A, \epsilon) = \inf_{E,V} \sigma(A, V, E, \epsilon),$$

which is a measure of the best approximate diagonalization that this approach can achieve. Davies conjectured that

$$\sigma(A,\epsilon) \leqslant c_n \epsilon^{1/2} \tag{3.4}$$

for some constant c_n, where $\|A\|_2 \leqslant 1$ is assumed, and he proved the conjecture for Jordan blocks and triangular Toeplitz matrices (both with $c_n = 2$) and for arbitrary 3×3 matrices (with $c_3 = 4$). Davies's conjecture was recently proved by Banks, Kulkarni, Mukherjee, and Srivastava [7, Thm. 1.1] with $c_n = 4n^{3/2} + 4n^{3/4} \leqslant 8n^{3/2}$. Building on the solution of Davies' conjecture a randomized algorithm with low computational complexity is developed in [5], [6] for approximately computing the eigensystem. Note that (3.4) suggests it is sufficient to choose E such that $\|E\|_2 \approx \epsilon^{1/2}$ in order to obtain $\sigma(A,\epsilon)$ of order $\epsilon^{1/2}$.

As we have stated it, the conjecture is over $\mathbb{C}^{n\times n}$. Davies's proofs of the conjecture for Jordan blocks and triangular Toeplitz matrices have E real when A is real, which is desirable. In the proof in [7], E is not necessarily real when A is real. However, Jain, Sah, and Sawhney [31, Thm 1.1] have proved the conjecture for real A and real perturbations E up to an extra factor of $(\log(1/\epsilon))^{1/4}$, while a weaker bound for which the extra factor is $\epsilon^{1/4}$ was independently derived by Banks et al. [8, Thm. 1.7G].

The matrix E can be thought of as a regularizing perturbation for the diagonalization. For computing matrix functions, Davies suggests taking E as a random matrix and gives empirical evidence that standard Gaussian matrices E scaled so that $\|E\|_2 \approx u^{1/2}$ are effective at delivering a computed result with error of order $u^{1/2}$ when $\|A\|_2 \leqslant 1$. Nick Higham published a short MATLAB code to implement this idea [25],[1] as a way of computing $f(A)$ with error of order $u^{1/2}$. However, this approach does not give a reliable numerical method for approximating matrix functions. The reason is that (3.3) does not correctly measure the effect on $f(A)$ of perturbing A by E. For small E, for any matrix norm we have

$$\|f(A+E) - f(A)\| \lesssim \|L_f(A,E)\| \leqslant \|L_f(A)\|\|E\|, \tag{3.5}$$

1 https://gist.github.com/higham/6c00f62e48c1b0116f2e9a8f43f2e02a

Table 3.1: Relative errors $\|f(\widetilde{A}) - f(A)\|_F / \|f(A)\|_F$ for approximation from randomized approximate diagonalization to the square root of the Jordan block $J(\lambda) \in \mathbb{R}^{n\times n}$, where $\widetilde{A} = A + E$ and $\|E\|_F = u^{1/2}\|A\|_F$.

λ	$n = 10$	$n = 20$	$n = 30$
1.0	7.46×10^{-9}	7.22×10^{-9}	9.45×10^{-9}
0.5	1.22×10^{-7}	3.42×10^{-4}	1.44
0.1	1.14	1.00	1.00

Table 3.2: Values of $\|L_f(A)\|_F$ corresponding to the results in Table 3.1.

λ	$n = 10$	$n = 20$	$n = 30$
1.0	1.41	2.01	2.46
0.5	2.62×10^{3}	8.55×10^{8}	4.75×10^{14}
0.1	1.13×10^{16}	4.99×10^{30}	3.24×10^{54}

where $L_f(A, E)$ is the Fréchet derivative of f at A in the direction E and $\|L_f(A)\| = \max\{\, \|L_f(A, E)\| : \|E\| = 1\,\}$ [24, sect. 3.1]. Hence while σ in (3.3) includes $\|E\|_2$, the change in f induced by E is as much as $\|L_f(A)\|_2\|E\|_2$, and the factor $\|L_f(A)\|_2$ can greatly exceed 1.

A simple experiment with $\epsilon = u$ illustrates this point. Unless otherwise stated, all the experiments in this paper are carried out in MATLAB R2020b with a working precision of double ($u \approx 1.1 \times 10^{-16}$). We take A to be an $n \times n$ Jordan block with eigenvalue λ and $f(A) = A^{1/2}$ (the principal matrix square root), for which $\|L_f(A)\|_F = \|(I \otimes A^{1/2} + (A^{1/2})^T \otimes I)^{-1}\|_2$ [23]. The diagonalization and evaluation of $f(\widetilde{A})$ is done at the working precision. In Table 3.1 we show the relative errors $\|f(A) - f(\widetilde{A})\|_F / \|f(A)\|_F$, where E is a (full) standard Gaussian matrix scaled so that $\|E\|_F = u^{1/2}\|A\|_F$ and the reference solution $f(A)$ is computed in 200 digit precision using the function `sqrtm` from the Multiprecision Computing Toolbox [36]. For $\lambda = 1$ we obtain an error of order $u^{1/2}$, but the errors grow as λ decreases and we achieve no correct digits for $\lambda = 0.1$. The reason is clear from Table 3.2, which shows the values of the term that multiplies $\|E\|_F$ in (3.5), which are very large for small λ. We stress that increasing the precision at which $f(\widetilde{A})$ is evaluated does not reduce the errors; the damage done by the perturbation E cannot be recovered.

In this work we adapt the idea of diagonalizing after a regularizing perturbation, but we take a new approach that does not depend on Davies's theory.

3.4 EVALUATING A FUNCTION OF A TRIANGULAR MATRIX

Our new algorithm uses the same blocked and reordered Schur form as the Schur–Parlett algorithm. The key difference from that algorithm is how it evaluates f on the (upper triangular) diagonal blocks. Given a diagonal block $T \in \mathbb{C}^{m\times m}$ of the blocked and reordered Schur form and an arbitrary function f we apply a regularizing perturbation with norm of order u and evaluate $f(T)$ at precision $u_h < u$. We expect m generally to be small, in which case the overhead of using higher precision arithmetic is small. In the worst case this approach should be competitive with the worst case for the Schur–Parlett algorithm [11, Alg. 2.6], since (3.2) requires up to $O(m^4)$ (working precision) flops.

We will consider two different approaches.

3.4.1 Approximate diagonalization with full perturbation

Our first approach is a direct application of approximate diagonalization, with $\epsilon = u^2$. Here, E is a multiple of a standard Gaussian matrix with norm of order $\epsilon^{1/2} = u$. Whereas Davies considered only matrices A of 2-norm 1, we wish to allow any norm, and the norm of E should scale with that of A. We will scale E so that

$$\|E\|_F = u \max_{i,j} |t_{ij}|. \tag{3.6}$$

We evaluate $f(T+E)$ by diagonalization at precision $u_h = u^2$ and hope to obtain a computed result with relative error of order u. Diagonalization requires us to compute the Schur decomposition of a full matrix $T + E$, and it costs about $28\frac{2}{3}m^3$ flops in precision u_h.

Although we do not expect this approach to provide a numerical method that works well for all problems, in view of the discussion and example in section 3.3, it is a useful basis for comparison with the new method in the next section.

3.4.2 Approximate diagonalization with triangular perturbation

Instead of regularizing by a full perturbation, we now take the perturbation E to be an upper triangular standard Gaussian matrix, normalized by (3.6). An obvious advantage of taking E triangular is that $\widetilde{T} = T + E$ is triangular and we can compute the eigenvectors (needed for diagonalization) by substitution, which is substantially more efficient than computing the complete eigensystem of a full matrix. Note that the diagonal entries of $\widetilde{T}$ are distinct with probability 1, albeit perhaps differing by as little as order $\|E\|_F$.

This approach can be thought of as indirectly approximating the derivatives by finite differences. Indeed for $m = 2$ we have

$$f(T) = \begin{bmatrix} f(t_{11}) & t_{12}f[t_{11}, t_{22}] \\ 0 & f(t_{22}) \end{bmatrix}, \quad f[t_{11}, t_{22}] = \begin{cases} \dfrac{f(t_{22}) - f(t_{11})}{t_{22} - t_{11}}, & t_{11} \neq t_{22}, \\ f'(t_{11}), & t_{11} = t_{22}, \end{cases} \tag{3.7}$$

so when $t_{11} = t_{22}$, perturbing to $\widetilde{t}_{11} \neq \widetilde{t}_{22}$ results in a first order finite difference approximation to $f'(t_{11})$. If $m = 2$ we can simply use (3.7) at the working precision, so the rest of this section is aimed at the case $m \geqslant 3$.

In order to find the eigenvector matrix V of the perturbed triangular matrix $\widetilde{T} = T + E$ we need to compute a set of m linearly independent eigenvectors v_i, $i = 1 : m$. This can be done by solving at precision u_h the m triangular systems

$$(\widetilde{T} - \widetilde{t}_{ii}I)v_i = 0, \quad i = 1 : m, \tag{3.8}$$

where we set v_i to be 1 in its ith component, zero in components $i+1\colon m$, and solve for the first $i-1$ components by substitution. Thus the matrix V is upper triangular.

To summarize, we compute in precision u_h the diagonalization

$$\widetilde{T} = VDV^{-1}, \quad D = \operatorname{diag}(\lambda_i), \tag{3.9}$$

where in practice the λ_i will be distinct. We then form $f(\widetilde{T}) = Vf(D)V^{-1}$ in precision u_h, which involves solving a multiple right-hand side triangular system with a triangular right-hand side. The cost of the computation is $\sum_{k=1}^{m} k^2 + m^3/3 = 2m^3/3 + O(m^2)$ flops in precision u_h.

3.4.2.1 *Determining the precision*

Now we determine the precision at which to carry out the diagonalization.

We expect the error in the computed approximation $\widehat{F}$ to $F = f(\widetilde{T})$ to be bounded approximately by (cf. [24, p. 82])

$$\frac{\|F - \widehat{F}\|_1}{\|F\|_1} \lesssim \kappa_1(V)\frac{\|f(D)\|_1}{\|f(\widetilde{T})\|_1}u_h. \tag{3.10}$$

(The choice of norm is not crucial; the 1-norm is convenient here.) We will use this bound to determine u_h. We do not know $\|f(\widetilde{T})\|_1$ a priori, but we have the bound

$$\frac{\|f(D)\|_1}{\|f(\widetilde{T})\|_1} \leqslant 1, \tag{3.11}$$

which follows from the fact that the spectral radius of a matrix is bounded above by any norm. Hence we can certainly expect that

$$\frac{\|F - \widehat{F}\|_1}{\|F\|_1} \lesssim \kappa_1(V)u_h. \tag{3.12}$$

Since we need to know how to choose u_h before we compute V, we need an estimate of $\kappa(V)$ based only on $\widetilde{T}$. Since we are using a triangular perturbation its regularizing effect will be less than that of a full perturbation, so we expect that we may need a precision higher than double the working precision.

Demmel [4, sect. 5.3], [12] showed that $\kappa_2(V)$ is within a factor m of $\max_i \|P_i\|_2$, where P_i is the spectral projector corresponding to the eigenvalue λ_i. Writing

$$\widetilde{T} = \begin{bmatrix} \widetilde{t}_{11} & \widetilde{t}_{12}^* \\ 0 & \widetilde{T}_{22} \end{bmatrix},$$

the spectral projector for the eigenvalue $\lambda_1 = \widetilde{t}_{11}$ is, with the same partitioning,

$$P_1 = \begin{bmatrix} 1 & p^* \\ 0 & 0 \end{bmatrix}, \quad p^* = \widetilde{t}_{12}^*(\widetilde{t}_{11}I - \widetilde{T}_{22})^{-1}. \tag{3.13}$$

From (3.13) we have

$$\|P_1\|_1 = \max(1, \|p\|_\infty) \leqslant \max\big(1, \|\widetilde{t}_{12}\|_\infty \|(\widetilde{t}_{11}I - \widetilde{T}_{22})^{-1}\|_1\big).$$

Now for any $m \times m$ upper triangular matrix U we have the bound [22, Thm. 8.12, Prob. 8.5]

$$\|U^{-1}\|_1 \leqslant \frac{1}{\alpha}\left(\frac{\beta}{\alpha} + 1\right)^{m-1}, \quad \alpha = \min_i |u_{ii}|, \quad \beta = \max_{i<j} |u_{ij}|. \tag{3.14}$$

This bound will be very pessimistic if we apply it to $\widetilde{t}_{11}I - \widetilde{T}_{22}$, because for the bound to be a good approximation it is necessary that many diagonal elements of U are of order α, yet $\widetilde{t}_{11}I - \widetilde{T}_{22}$ will typically have only a few (if any) small elements. Let us group the $\widetilde{t}_{ii}$ according to the Schur–Parlett blocking criteria described in section 3.2, with blocking parameter $\delta = \delta_1$. Suppose the largest block has size $k \geqslant 2$ and suppose, without loss of generality, that it comprises the first k diagonal elements of $\widetilde{T}$. Then we will approximate $\|(\widetilde{t}_{11}I - \widetilde{T}_{22})^{-1}\|_1$ by $\|(\widetilde{t}_{11}I - \widetilde{T}_{22}(1\colon k-1, 1\colon k-1))^{-1}\|_1$, and bound it by (3.14), leading to the approximation

$$\max_i \|P_i\|_1 \approx \frac{\max_{i<j} |\widetilde{t}_{ij}|}{c_m u}\left(\frac{\max_{i<j} |\widetilde{t}_{ij}|}{c_m u} + 1\right)^{k-2},$$

where the parameter c_m is such that $c_m u \approx \min_i |w_{ii}|$, where the w_{ii} are the diagonal elements of $\widetilde{t}_{11}I - \widetilde{T}_{22}(1\colon k-1, 1\colon k-1)$, and hence this is an estimate of $\kappa_1(V)$ by Demmel's result.

We are aiming for an error of order u, so from (3.12) we need $\kappa_1(V)u_h \lesssim u$, which gives the requirement

$$u_h \lesssim \frac{c_m u^2}{\max_{i<j} |\widetilde{t}_{ij}| \left(\frac{\max_{i<j} |\widetilde{t}_{ij}|}{c_m u} + 1 \right)^{k-2}}, \quad k \geqslant 2. \tag{3.15}$$

In the case $k = 2$, the bound (3.15) is $u_h \lesssim c_m u^2 / \max_{i<j} |\widetilde{t}_{ij}| = O(u^2)$. If the largest block size is $k = 1$, we use $u_h = u^2$ (corresponding to Davies's conjecture (3.4)) since we do not expect $\kappa(V)$ to be so large that a precision higher than double the working precision is required.

3.4.2.2 *The algorithm*

We summarize this algorithm, based on a triangular perturbation, in Algorithm 3.1, where we have

$$u_h = \begin{cases} u^2, & k = 1, \\ \min\bigl(u^2, \text{right-hand side of (3.15)}\bigr), & k \geqslant 2. \end{cases} \tag{3.16}$$

In summary, we make an upper triangular perturbation E of norm $O(u\|T\|_1)$ to T and evaluate $f(T+E)$. We choose the precision of the evaluation in order to be sure of obtaining an accurate evaluation of $f(T+E)$. Our approach differs fundamentally from Davies's randomized approximate diagonalization. Our triangular E has a lesser regularizing effect (on the condition number of the eigenvector matrices) than a full one, so it results in a potentially larger $\kappa(V)$, but our choice of u_h takes this into account. On the other hand, since our perturbation E is upper triangular and of order $u\|T\|_1$, it corresponds to a backward error of order u and so is harmless. A full perturbation E cannot be interpreted as a backward error for the $f(T)$ evaluation as it perturbs the zeros in the lower triangle of T.

The analysis above is unchanged if E is diagonal, so we allow E to be chosen as diagonal or upper triangular in Algorithm 3.1 and will compare the two choices experimentally in later section.

Algorithm 3.1 Multiprecision algorithm for function of a triangular matrix.

Given a triangular matrix $T \in \mathbb{C}^{m\times m}$ and a function f, this algorithm computes $F = f(T)$. It uses arithmetics of unit roundoff u (the working precision), u^2, and possibly a higher precision $u_h \leqslant u^2$. Lines 11–13 are to be executed at precision u^2 and lines 14–18 are to be executed at precision u_h, as indicated to the right of the line numbers.

```
1          if $m = 1$
2             $f_{11} = f(t_{11})$, quit
3          end
4          if $m = 2$ and $t_{11} \neq t_{22}$
5             $f_{11} = f(t_{11})$, $f_{22} = f(t_{22})$
6             $f_{12} = t_{12}(f_{22} - f_{11})/(t_{22} - t_{11})$
7             quit
8          end
9          Form an $m \times m$ diagonal or upper triangular standard Gaussian matrix $N$.
10         $E = u(\max_{i,j} |t_{ij}|/\|N\|_F)N$
11  $u^2$   $\widetilde{T} = T + E$
12  $u^2$   $D = \mathrm{diag}(\widetilde{T})$
13  $u^2$   Evaluate $u_h$ by (3.16).
14  $u_h$   if $u_h < u^2$, convert $\widetilde{T}$ and $D$ to precision $u_h$, end
15         for $i = 1\colon m$
16  $u_h$      Set $(v_i)_i = 1$ and $(v_i)_k = 0$ for $k > i$ and solve the triangular system
              $(\widetilde{T} - \widetilde{t}_{ii}I)v_i = 0$ for the first $i-1$ components of $v_i$.
17         end
18  $u_h$   Form $F = Vf(D)V^{-1}$, where $V = [v_1, \dots, v_m]$.
19         Round $F$ to precision $u$.
```

In practice, overflow could occur when solving the triangular systems (3.8), especially when there is a large cluster of eigenvalues. However, we solve (3.8) in high precision arithmetic, and these arithmetics have a very large exponent range, which reduces the likelihood of overflow. For example, IEEE quadruple precision arithmetic has largest element of order 10^{4932} [30]. In any case, scaling techniques are available that avoid overflow [3], [42] (and likewise for the Sylvester equations (3.1) in the Schur–Parlett algorithm [41]).

3.4.2.3 *Specifying the parameters*

We now discuss the choice of the parameters δ_1 and c_m in Algorithm 3.1. The parameter c_m is such that $c_m u \approx \min_i |w_{ii}|$, where the w_{ii} are the diagonal elements of $\widetilde{t}_{11}I - \widetilde{T}_{22}(1\colon k-1, 1\colon k-1)$. We will determine c_m by considering the extreme case when $\min_i |w_{ii}|$ is extremely small, which is when all $\widetilde{t}_{ii}$ in the largest block of size $k \leqslant m$ differ only by the perturbation we added (in which case the t_{ii} are exactly repeated) and thus $\widetilde{t}_{11}I - \widetilde{T}_{22}(1\colon k-1, 1\colon k-1)$ is extremely ill-conditioned. This choice of c_m makes the chosen higher precision $u_h < u^2$ pessimistic when some of the $\widetilde{t}_{ii}$ are close in the sense they are partitioned in the same block by $\delta = \delta_1$ but not all of them are exactly repeated, but it helps to ensure the accuracy of the algorithm in all cases. However, since the algorithm is to be employed in the next section for computing a function of $T \in \mathbb{C}^{m\times m}$ where generally m (and hence k) is expected to be small, we do not expect this approach to seriously affect the efficiency of the overall algorithm. In the case we are considering we have $|w_{ii}| = |\widetilde{t}_{11} - \widetilde{t}_{ii}| = |e_{11} - e_{ii}|$. The matrix E on line 10 of Algorithm 3.1 has entries $u(\max_{i,j}|t_{ij}|)|\widetilde{n}_{ij}|$, where $\|\widetilde{N}\|_F = 1$, and we expect that $|\widetilde{n}_{ij}| \approx 1/\sqrt{m}$ if N is chosen to be diagonal, or $|\widetilde{n}_{ij}| \approx \sqrt{2}/m$ if N is triangular [33]. This suggests taking

$$c_m = \begin{cases} \theta_d \max_{i,j}|t_{ij}|/\sqrt{m}, & \text{if } N \text{ is diagonal,} \\ \theta_t \max_{i,j}|t_{ij}|/m, & \text{if } N \text{ is triangular,} \end{cases}$$

for some constants θ_d and θ_t. The constants are introduced to empirically quantify the uncertainty caused by the stochasticity of the left-hand side in the approximation $\min_i |e_{11} - e_{ii}| \approx c_m u$. In our experiments with different choices of θ_d and θ_t we found $\theta_d = 0.4$ and $\theta_t = 0.5$ to be good choices.

The blocking parameter $\delta = \delta_1$ is important in determining the largest group size k in (3.15). A smaller δ can potentially group fewer eigenvalues and decrease k, causing a larger u_h to be used. Yet too large a δ can result in a u_h that is much smaller than necessary to achieve the desired accuracy. We have found experimentally that $\delta_1 = 5 \times 10^{-3}$ is a good choice in a working precision of double, which worked well for the experiments presented in this work. In general there is no optimal (fixed) value for δ_1 as this blocking parameter controls the size k in the approximation $\|(\widetilde{t}_{11}I - \widetilde{T}_{22})^{-1}\|_1 \approx \|(\widetilde{t}_{11}I - \widetilde{T}_{22}(1\colon k-1, 1\colon k-1))^{-1}\|_1$ used in section 3.4.2.1 for determining the precision u_h, and a suitable value for k that results in a good approximation is clearly problem-dependent. In general a larger δ_1 is more likely to produce a larger k and thus result in a higher precision to be used, so an overestimated δ_1 does no harm to accuracy but may impair the efficiency.

3.4.2.4 *The parameters in arbitrary precision*

In this section we investigate how to choose in arbitrary precision the two parameters δ_1 and c_m, which may depend on the working precision u so should be considered as some functions of u. The general framework for selecting δ_1 and c_m remains the same as in double precision. We will test the diagonal perturbation case since it requires slightly less computation, and for the case of triangular perturbation the same procedure can be used.

We set different values to θ_d and run the algorithm 10 times in different working precisions for computing $f(T)$, where $f = \exp$, and T has only one eigenvalue of multiplicity m:

- $T_1 =$ `gallery('jordbloc',m,5)`,
- $T_2 =$ `gallery('triw',m)`.

The maximal error over the 10 executions is reported in Figure 3.1. We repeated this experiment with $f = \log$ and the behaviour of errors were similar.

Figure 3.1 shows that $\theta_d \leqslant 0.4$ effectively enables Algorithm 3.1 to produce an error of order u in all tested cases, independent of the working precision. Then we can take $\theta_d = 0.4$ as the experiment suggests that this choice is small enough for the algorithm to provide an error of $O(u)$ for matrices whose multiplicity of eigenvalues is up to 100. In fact, we have found that the chosen digits of precision u_h is not sensitive to changes in θ_d, for example, in a working precision of 64 digits for $m = 100$ Algorithm 3.1 with $\theta_d = 0.4$ employs 6442 digits, and the digits used will increase by only 10 if $\theta_d = 0.3$ is used. The choice is also confirmed by Figure 3.2, which shows the errors delivered by Algorithm 3.1 with $\theta_d = 0.4$ for T of different size m in various working precisions.

On the other hand, the blocking parameter δ_1 is important in determining the largest group size k in (3.15). As discussed in the previous section, a smaller δ_1 can potentially group fewer eigenvalues and decrease k, causing a larger u_h to be used

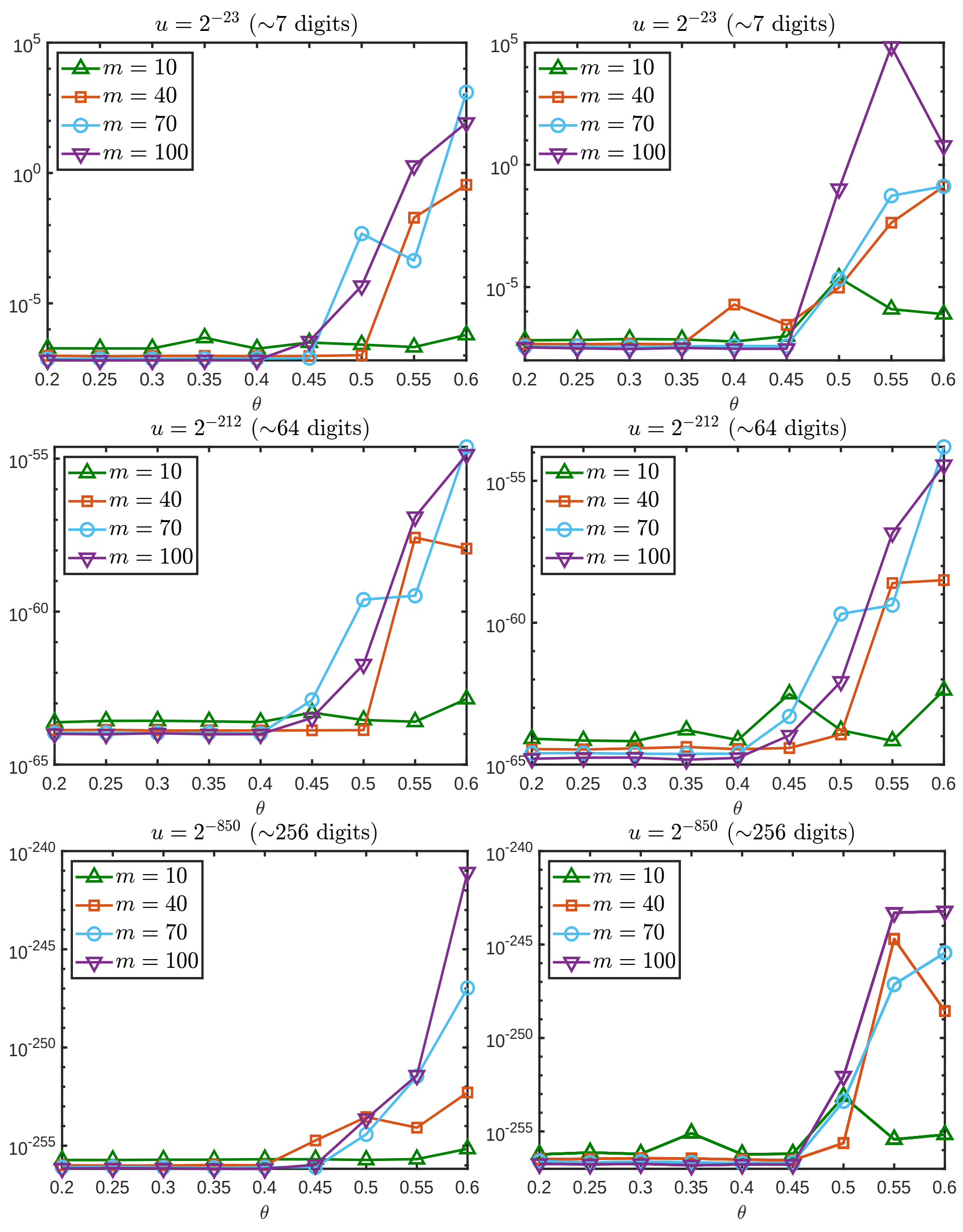


Figure 3.1: Maximal normwise relative errors for Algorithm 3.1 with different θ_d over 10 executions, the working precisions are 7, 64, and 256 decimal digits. Left: $T = T_1$; right: $T = T_2$. $f = \exp$.

(for a fixed working precision u). It can happen that when T has close eigenvalues too small a δ_1 will fail to choose a sufficiently small u_h for the large $\kappa(V)$, and thus cause loss of accuracy. Yet too large a δ_1 can result in a u_h that is much smaller than necessary to achieve the desired accuracy. We note that it is not guaranteed that the u_h chosen by the algorithm is always exactly as small as it needs to just cancel the loss of digits caused by large $\kappa(V)$, and this is due to our pessimistic approach of determining c_m, but we aims to reduce the number of unnecessary digits as much

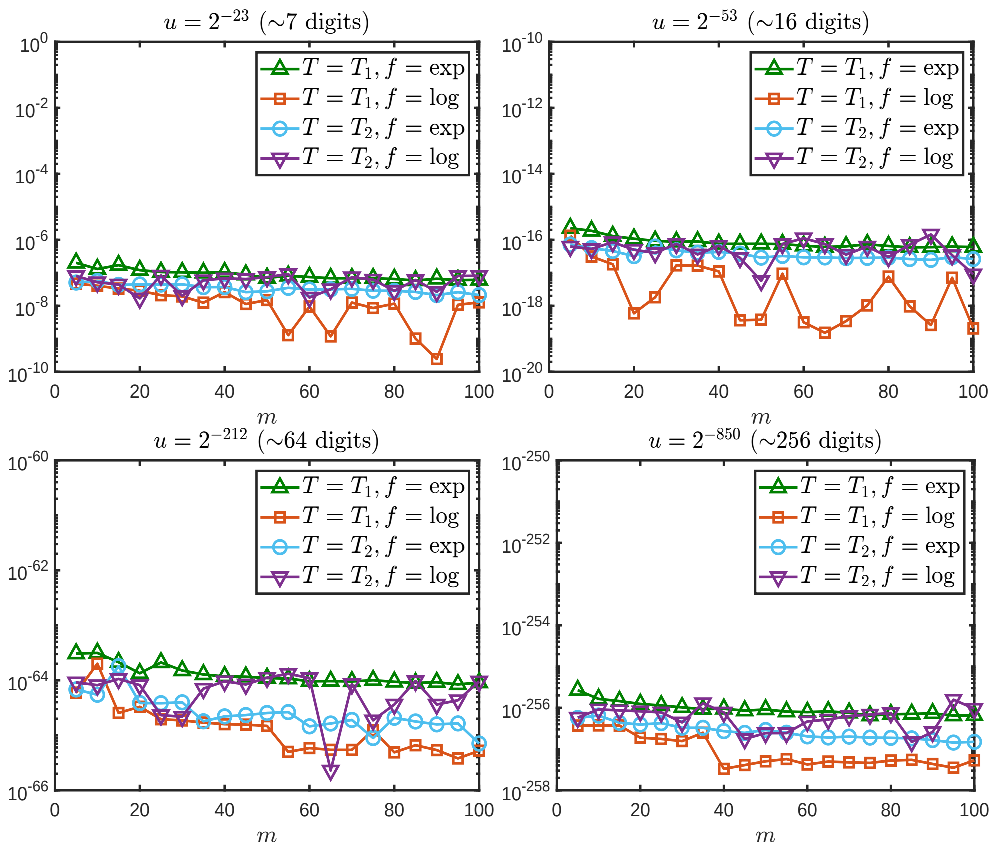


Figure 3.2: Normwise relative errors for Algorithm 3.1 with $\theta_d = 0.4$ for T with different size m, the working precisions are 7, 16, 64, and 256 decimal digits.

as possible. Following this commitment, for a given matrix there must exist a *feasible interval* $[a, b)$ in the sense that $\delta_1 \in [a, b)$ enables the algorithm to provide an error of order u, and any $\delta_1 < a$ will reduce the largest group size k and ruin the desired accuracy, and any $\delta_1 \geqslant b$ will increase k while the algorithm will give the same level of accuracy. For a triangular matrix whose spectra are available we still cannot tell whether a chosen δ_1 is in its feasible interval $[a, b)$ without knowing $\kappa(V)$ (the order of the resulting error will be predictable if $\kappa(V)$ is known), or actually computing the error. In general such an feasible interval $[a, b)$ for δ_1 is different for different matrices, and this is illustrated by the simple experiments reported in Table 3.3, where we also report the quantity $\kappa_{\exp}(T)u$, where $\kappa_{\exp}(T)$ is the Frobenius norm condition number [24, Chap. 3] of the matrix exponential at T, which we estimate using the `funm_condest1` function provided by [27].

Table 3.3: The largest group size k, digits of the higher precision u_h, and relative error $\|\exp(T) - \widehat{F}\|_F / \|\exp(T)\|_F$ for Algorithm 3.1 with $\theta_d = 0.4$ and different δ_1. The working precision is double ($u = 2^{-53}$).

matrix	δ_1	k	digits	error	$\kappa_{\exp}(T)u$
`triu(randn(100))`	2×10^{-3}	2	34	2.3e-5	8.8e-15
	5×10^{-3}	3	51	5.7e-17	8.8e-15
`triu(rand(100))`	2×10^{-3}	5	86	5.7e-17	6.2e-14
`gallery('kahan',100)`	1×10^{-5}	1	32	2.6e-16	3.1e-16
	5×10^{-3}	62	1048	6.6e-17	3.1e-16
`gallery('kahan',20)`	0.01	1	32	6.0e-17	1.6e-16

For $T =$ `triu(randn(100))` the precision chosen by $\delta_1 \leqslant 2 \times 10^{-3}$ is not sufficiently high, which leads to loss of accuracy, while if δ_1 is increased to 5×10^{-3}, the largest group size k will increase by 1 and the algorithm can provide the desired level of accuracy. Corresponding to our definition of the feasible interval it is obvious that $2 \times 10^{-3} < a \leqslant 5 \times 10^{-3} < b$ for this matrix. But for $T =$ `triu(rand(100))` we can deduce that $\delta_1 = 2 \times 10^{-3} \geqslant a$. For $T =$ `gallery('kahan',100)` whose eigenvalues are distinct and on the interval $(0, 1]$ using the default precision with 32 digits achieves the targeted level of accuracy, and a suitable $\delta_1 > 0$ can be chosen arbitrarily close to 0 such that no eigenvalues are grouped together; $\delta_1 = 5 \times 10^{-3}$ results in a much higher precision being used to deliver the same level of accuracy, so it is larger than necessary. Hence, for $T =$ `gallery('kahan',100)` we have $a = 0$ and $1 \times 10^{-5} < b < 5 \times 10^{-3}$. Intuitively, it is not surprising to see that, for the same class of matrices the upper bound b to δ_1 will in general increase as size of the matrix decrease, for example, for $T =$ `gallery('kahan',20)` we have $a = 0$ and $b > 0.01$.

We have shown that it is not possible to choose an optimal δ_1 for general matrices, and in general its choice is a balance between accuracy and efficiency: a smaller δ_1 can reduce the chance or extent of unnecessarily high digits being used but the algorithm is more likely to be inaccurate, while a larger δ_1 enables the algorithm to produce accurate results for more cases but it is more often to have some digits wasted. For the sake of accuracy it is inevitable that in some cases a fixed δ_1 will be overestimated, especially for matrices of large size. However, this should not be too problematic since Algorithm 3.1 will be employed in next section for computing the reordered

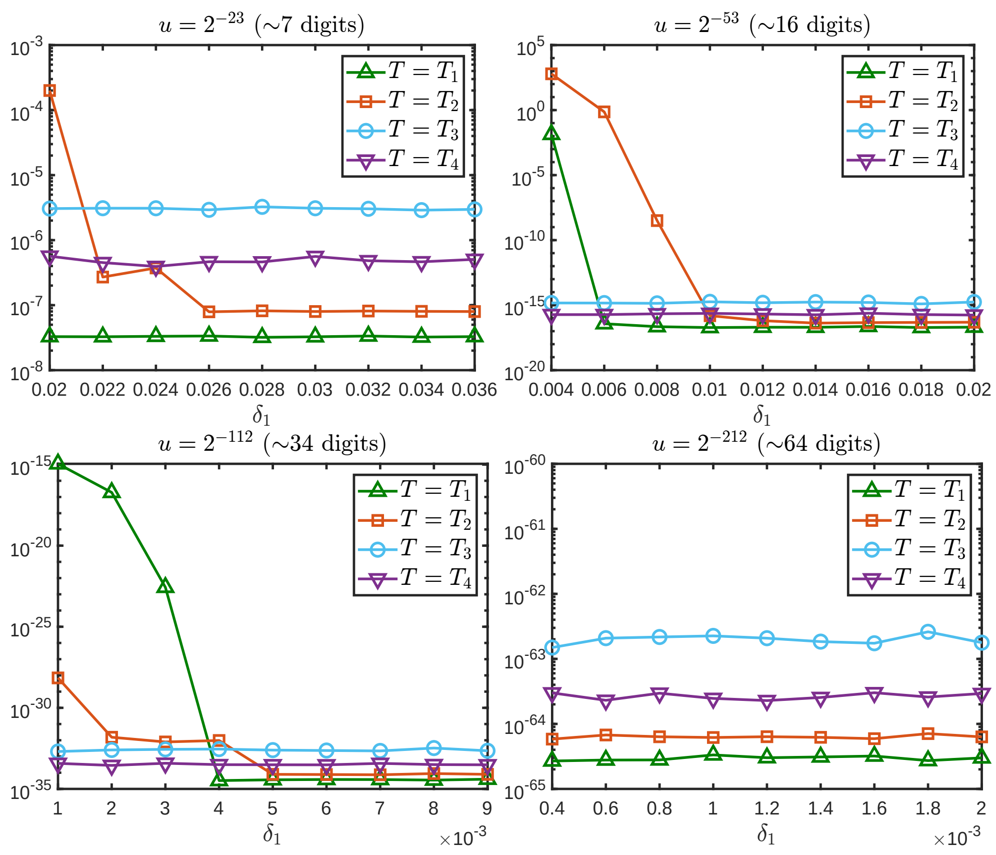


Figure 3.3: Maximal relative errors for Algorithm 3.1 with $\theta_d = 0.4$ and different δ_1 among 100 different matrices of each class; $f = \exp$, the working precisions are 7, 16, 34, and 64 decimal digits.

Schur matrix on the diagonal whose size is generally small, and computation in precision higher than double is done in software so its speed is not massively affected if only few more digits are used. Next, we will seek suitable values for δ_1 based on experiments with 100 different random matrices of the following kinds:

- T_1 = `triu(rand(m))`.
- T_2 = `triu(randn(m))`.
- T_3 = `schur(rand(m),'complex')`.
- T_4 = `schur(randn(m),'complex')`.

Then we set $f = \exp$ (we found that the choice of f is not crucial here), and use Algorithm 3.1 with $\theta = 0.4$ and different δ_1 for computing f of above matrices of size 100×100 in different working precisions, and show in Figure 3.3 the maximal error among the 100 different matrices of each class.

From Figure 3.3 the 'optimal' values for the blocking parameter, denoted as δ_1^*, are obvious, which are the smallest δ_1 such that the algorithm gives an error of order u for all tested matrices, and we observe that δ_1^* is distinct in different working precisions, for example, $\delta_1^* = 0.022$ in single precision ($u = 2^{-23}$), $\delta_1^* = 0.010$ in double precision ($u = 2^{-53}$), and $\delta_1^* = 0.004$ in quadruple precision ($u = 2^{-112}$). We see that δ_1^* is decreasing as u becomes smaller, and this is because the way we determine c_m in the bound (3.15) becomes more pessimistic as the working precision increases: for a matrix T the higher precision u_h being used will have one more factor of the working precision u if the largest block size k increases by 1, so for the same matrix in a higher working precision $\widetilde{u} < u$ the algorithm might tolerate a smaller δ_1 (and hence smaller k) to give an error of order $\widetilde{u}$. Indeed, we have found that for a working precision of 64 digits or higher, a higher precision $u_h = u^2$ ensures targeted accuracy for all the matrices tested in the experiments, in which case we have $\delta_1^* = 0$. Due to this property of the blocking parameter δ_1 we could use $\delta_1 = 0.022$ for any working precisions higher than single, but this choice is increasingly pessimistic as the working precision increases. Instead, in arbitrary precision one can use

$$\delta_1 = \frac{c}{\lceil \log_{10}(1/u) \rceil}, \qquad c = 0.16,$$

where the denominator is the approximated decimal digits of the working precision u, and $c \approx 0.16$ is obtained from linear interpolation of the values of δ_1^* and $\lceil \log_{10}(1/u) \rceil^{-1}$ in single, double, and quadruple precisions.

3.4.3 Numerical experiments

In this section we describe a numerical experiment with the methods of sections 3.4.1 and 3.4.2 for computing a function of a triangular matrix. Precisions higher than double precision are implemented with the Multiprecision Computing Toolbox [36].

We set the function f to be the exponential, the square root, the sign function, the logarithm, the cosine, and the sine. The algorithms for computing $f(T)$ to be tested are

Table 3.4: Equivalent number of decimal digits for the higher precision u_h used by Algorithm 3.1 in the computation. 32 digits corresponds to $u_h = u^2$.

	$m = 35$	$m = 75$
$T_1 =$ `gallery('kahan',m)`	32	623
$T_2 =$ `schur(gallery('smoke',m),'complex')`	32	32
$T_3 =$ `schur(randn(m),'complex')`	32	32
$T_4 =$ `schur(rand(m),'complex')`	32	32
$T_5 =$ `triu(randn(m))`	34	68
$T_6 =$ `triu(rand(m))`	51	68
$T_7 =$ `gallery('jordbloc',m,0.5)`	599	1296

- `Alg_full`: approximate diagonalization with a full perturbation and $u_h = u^2$, as described in section 3.4.1,
- `Alg_diag`: Algorithm 3.1 with diagonal E, $c_m = 0.4\max_{i,j}|t_{ij}|/\sqrt{m}$, and $\delta_1 = 5 \times 10^{-3}$.

We use the following matrices, generated from built-in MATLAB functions.

- $T_1 =$ `gallery('kahan',m)`: upper triangular with distinct diagonal elements on the interval $(0, 1]$.
- $T_2 =$ `schur(gallery('smoke',m),'complex')`: Schur factor of the complex matrix whose eigenvalues are the mth roots of unity times $2^{1/m}$.
- $T_3 =$ `schur(randn(m),'complex')`.
- $T_4 =$ `schur(rand(m),'complex')`.
- $T_5 =$ `triu(randn(m))`.
- $T_6 =$ `triu(rand(m))`.
- $T_7 =$ `gallery('jordbloc',m,0.5)`: a Jordan block with eigenvalue 0.5.

Since we are computing the principal matrix square root and the principal logarithm we multiply matrices T_3, T_4, and T_5 by $1 + \mathrm{i}$ for these functions to avoid their eigenvalues being on the negative real axis, where i is the imaginary unit.

We report the equivalent number of decimal digits for the higher precision u_h used by Algorithm 3.1 for each test matrix in the computation in Table 3.4. Since

Table 3.5: Maximal normwise relative errors for Algorithm 3.1 with a diagonal E (`Alg_diag`) and the method of approximate diagonalization with full perturbation (`Alg_full`). Exceptionally large errors are highlighted in bold red text.

	$f = \exp$			$f = \mathrm{sqrt}$		
	`Alg_diag`	`Alg_full`	$\kappa_f(A)u$	`Alg_diag`	`Alg_full`	$\kappa_f(A)u$
$T_1, m = 35$	5.6e-17	3.3e-17	6.4e-15	2.7e-16	3.1e-13	5.4e-11
$T_1, m = 75$	5.4e-17	2.5e-17	1.2e-14	2.1e-15	**8.2e-7**	3.2e-11
$T_2, m = 35$	5.2e-17	4.2e-17	1.9e-15	5.9e-16	3.9e-13	5.6e-11
$T_2, m = 75$	3.0e-17	2.2e-17	3.6e-15	5.5e-16	**1.0e-7**	4.5e-12
$T_3, m = 35$	1.7e-16	9.2e-17	1.9e-14	1.4e-16	1.4e-16	3.3e-14
$T_3, m = 75$	1.1e-16	6.2e-17	1.1e-13	1.8e-16	1.3e-16	1.5e-13
$T_4, m = 35$	6.6e-16	2.2e-16	3.2e-15	1.4e-15	1.3e-15	4.9e-14
$T_4, m = 75$	1.1e-15	2.1e-17	2.6e-14	1.4e-15	1.7e-15	1.5e-13
$T_5, m = 35$	8.9e-17	7.7e-17	1.0e-14	1.5e-15	**9.7e-9**	3.8e-10
$T_5, m = 75$	5.5e-17	6.7e-17	5.6e-14	2.5e-14	**1.0**	4.3e-22
$T_6, m = 35$	4.6e-17	4.8e-17	3.7e-14	1.0e-15	6.7e-12	5.6e-9
$T_6, m = 75$	2.8e-17	5.4e-17	2.6e-13	1.9e-15	**1.0**	2.7e-14
$T_7, m = 35$	5.8e-17	1.8e-16	5.7e-15	4.1e-16	**8.5e-8**	3.9e-12
$T_7, m = 75$	1.1e-19	3.6e-16	1.2e-14	3.4e-16	**1.0**	6.6e-18

	$f = \mathrm{sign}$			$f = \log$		
	`Alg_diag`	`Alg_full`	$\kappa_f(A)u$	`Alg_diag`	`Alg_full`	$\kappa_f(A)u$
$T_1, m = 35$	0	1.6e-25	7.7e-22	2.3e-16	5.1e-13	7.9e-11
$T_1, m = 75$	0	5.2e-20	1.3e2	4.1e-15	**1.1e-6**	3.1e-12
$T_2, m = 35$	3.6e-16	3.8e-13	4.2e-11	5.1e-16	6.3e-13	5.7e-11
$T_2, m = 75$	5.1e-16	**1.2e-7**	5.6e-12	8.8e-16	**1.7e-7**	2.5e-12
$T_3, m = 35$	1.6e-16	1.4e-16	4.1e-14	2.0e-16	1.9e-16	3.2e-14
$T_3, m = 75$	8.6e-17	9.1e-17	4.2e-14	1.9e-16	1.8e-16	1.2e-13
$T_4, m = 35$	9.4e-16	9.2e-16	3.7e-14	1.3e-15	2.2e-15	6.1e-14
$T_4, m = 75$	1.2e-15	2.2e-15	9.8e-14	1.5e-15	3.1e-15	1.5e-13
$T_5, m = 35$	1.8e-15	**7.2e-9**	8.9e-11	1.6e-15	**1.1e-8**	1.0e-10
$T_5, m = 75$	1.8e-14	**1.0**	5.8e-23	3.6e-14	**1.0**	6.0e-23
$T_6, m = 35$	0	3.0e-17	4.1e-21	1.9e-15	8.8e-12	2.1e-9
$T_6, m = 75$	0	**1.0**	3.8e3	3.0e-15	**1.0**	3.4e-15
$T_7, m = 35$	0	1.1e-16	2.3e1	2.3e-16	**1.4e-7**	7.2e-13
$T_7, m = 75$	0	**1.0**	1.4e2	7.1e-16	**1.0**	1.0e-18

	$f = \cos$			$f = \sin$		
	`Alg_diag`	`Alg_full`	$\kappa_f(A)u$	`Alg_diag`	`Alg_full`	$\kappa_f(A)u$
$T_1, m = 35$	3.8e-17	3.5e-17	7.9e-15	4.6e-17	4.7e-17	8.3e-15
$T_1, m = 75$	3.2e-17	2.6e-17	1.7e-14	3.6e-17	4.4e-17	3.7e-14
$T_2, m = 35$	5.4e-17	4.4e-17	3.5e-15	4.1e-17	3.4e-17	4.3e-15
$T_2, m = 75$	4.1e-17	3.0e-17	7.1e-15	3.8e-17	1.7e-17	8.3e-15
$T_3, m = 35$	1.8e-16	8.6e-17	1.6e-14	1.4e-16	9.7e-17	1.6e-14
$T_3, m = 75$	1.3e-16	6.8e-17	6.6e-14	1.7e-16	6.6e-17	5.6e-14
$T_4, m = 35$	3.4e-16	2.5e-16	8.7e-15	3.3e-16	2.4e-16	8.1e-15
$T_4, m = 75$	4.8e-16	3.0e-16	2.1e-14	4.5e-16	2.9e-16	1.9e-14
$T_5, m = 35$	7.0e-17	7.5e-17	1.3e-14	7.0e-17	7.4e-17	1.5e-14
$T_5, m = 75$	5.1e-17	6.1e-17	5.2e-9	4.9e-17	5.9e-17	4.0e-9
$T_6, m = 35$	3.8e-17	4.1e-17	2.1e-11	3.8e-17	4.8e-17	8.4e-11
$T_6, m = 75$	4.0e-17	7.2e-17	1.1e-13	2.2e-17	3.5e-17	1.1e-13
$T_7, m = 35$	4.0e-17	4.8e-16	3.0e-15	3.7e-17	4.7e-16	2.9e-15
$T_7, m = 75$	2.0e-17	9.3e-16	6.5e-15	1.5e-17	1.4e-15	6.3e-15

the outputs of `Alg_full` and `Alg_diag` depend on the random perturbation E, we compute the function of each matrix 10 times and report in Table 3.5 the maximum relative error $\|F - \widehat{F}\|_F / \|F\|_F$, where F is a reference solution computed by the functions `expm`, `sqrtm`, `logm`, `cosm`, and `sinm` provided by the Multiprecision Computing Toolbox running at 200 digit precision, and rounded back to double precision. For the reference solution of the matrix sign function, we run `signm` from the Matrix Function Toolbox [27] at 200 digit precision, and round back to double precision. The same procedure is followed in the experiments in the following sections.

We also show in Table 3.5 the quantity $\kappa_f(A)u$, where $\kappa_f(A)$ is the 1-norm condition number [24, Chap. 3] of f at A, which we estimate using the `funm_condest1` function provided by [27]. A numerically stable algorithm will produce forward errors bounded by a modest multiple of $\kappa_f(A)u$.

The results show that Algorithm 3.1 behaves in a numerically stable fashion in every case, typically requiring a higher precision with unit roundoff u_h equal to or not much smaller than u^2. We see that for the same class of matrices the number of digits of precision used is nondecreasing with the matrix size m, which is to be expected since we expect a larger maximum block size (equal to k in (3.15)) for a larger matrix.

On the other hand, as expected in view of the discussion in section 3.3, the randomized approximate diagonalization method `Alg_full` is less reliable and sometimes not accurate at all. The failure of the method occurs for the square root, sign, and logarithm functions, all of which have singularities.

Note that our test matrices here are more general than will arise in the algorithm of the next section, for which the diagonal blocks will have clustered eigenvalues.

We repeated this experiment with an upper triangular E in Algorithm 3.1. The errors were of the same order of magnitude as for diagonal E. Since a diagonal E requires slightly less computation, we will take E diagonal in the rest of this paper.

Algorithm 3.2 Multiprecision Schur–Parlett algorithm for function of a full matrix.

Given $A \in \mathbb{C}^{n\times n}$ and a function f this algorithm computes $F = f(A)$. It uses arithmetics of unit roundoff u (the working precision), u^2, and possibly higher precisions $u_h \leqslant u^2$ (chosen in Algorithm 3.1). It requires only function values, not derivatives.

```
1  Compute the Schur decomposition of $A = QTQ^*$.
2  if $T$ is diagonal, $F = Qf(T)Q^*$, quit, end
3  Use Algorithms 4.1 and 4.2 in [11, sect. 4] with $\delta > 0$ to reorder $T$ into
   a block $m \times m$ upper triangular matrix $\widetilde{T} = U^*TU$.
4  for $i = 1\colon m$
5      Use Algorithm 3.1 (with a diagonal $E$) to evaluate $F_{ii} = f(\widetilde{T}_{ii})$.
6      for $j = i-1\colon -1\colon 1$
7          Solve the Sylvester equation (3.1) for $F_{ij}$.
8      end
9  end
10 $F = QUFU^*Q^*$
```

3.5 OVERALL ALGORITHM FOR COMPUTING $f(A)$

Our algorithm for computing $f(A)$ follows the framework of the Schur–Parlett algorithm [11]. First the Schur decomposition $A = QTQ^*$ is computed. Then the triangular matrix T is reordered to a partitioned upper triangular matrix $\widetilde{T}$ by a unitary similarity transformation, which is achieved by Algorithms 4.1 and 4.2 in [11, sect. 4]. The function is evaluated on the diagonal blocks $\widetilde{T}_{ii}$ by Algorithm 3.1 instead of by a Taylor expansion as in the Schur–Parlett algorithm, and the precision u_h used in Algorithm 3.1 is potentially different for each diagonal block. The off-diagonal blocks of $f(\widetilde{T})$ are computed using the block form (3.1) of the Parlett recurrence. Finally, we undo the unitary similarity transformations from the Schur decomposition and the reordering. This gives Algorithm 3.2.

In Algorithm 3.2 we distinguish a special case: if A is normal, the Schur decomposition becomes $A = QDQ^*$ with D diagonal, and the algorithm simply computes $f(A) = Qf(D)Q^*$. We note that the algorithm preserves the advantages of the Schur–Parlett algorithm that if one wants to compute $f(A) = \sum_i f_i(A)$ then it is not necessary to compute each $f_i(A)$ separately because the Schur decomposition and its reordering can be reused.

In the reordering and blocking of the Schur–Parlett framework the blocking parameter $\delta > 0$, described in section 3.2, needs to be specified. A large δ leads to greater separation of the eigenvalues of the diagonal blocks, which improves the accuracy of the solutions to the Sylvester equations. In this respect, there is a significant difference between Algorithm 3.2 and the standard Schur–Parlett algorithm: the latter algorithm cannot tolerate too large a δ because it slows down convergence of the Taylor series expansion, meaning that more terms may be needed (or the series may simply not converge). Since Algorithm 3.1 performs well irrespective of the eigenvalue distribution we can choose δ without consideration of the accuracy of the evaluation of f on the diagonal blocks and larger δ will in general do no harm to accuracy. In the extreme case where δ is so large that one block is employed, Algorithm 3.2 does not solve Sylvester equations and thus avoids the potential error incurred in the process, and in general this is when our algorithm attains its optimal accuracy, but the price to pay is that it becomes very expensive because higher precision arithmetic is being used on an $n \times n$ matrix. We investigate the choice of δ experimentally in the next subsection.

3.5.1 Numerical experiments

In the Schur–Parlett algorithm [11] the blocking parameter $\delta = 0.1$ is chosen, which is shown there to perform well most of the time. In order to investigate a suitable value for δ in Algorithm 3.2, we compare the following four algorithms, where "nd" stands for "no derivative".

- `funm_nd_0.1`, Algorithm 3.2 with $\delta = 0.1$;
- `funm_nd_0.2`, Algorithm 3.2 with $\delta = 0.2$;
- `funm_nd_norm`, Algorithm 3.2 with $\delta = 0.1 \max_i |t_{ii}|$; and
- `funm_nd_∞`, Algorithm 3.2 with $\delta = \infty$ (no blocking, so the whole Schur factor T is computed by Algorithm 3.1).

The 35 tested matrices are nonnormal taken from

- the MATLAB `gallery`;
- the Matrix Computation Toolbox [26];
- other MATLAB matrices: `magic`, `rand`, and `randn`.

We set their size to be 32×32, and we also test the above matrices multiplied by $10^{\pm 2}$ to examine the robustness of the algorithms under scaling. We set the function f to be the matrix sine; similar results were obtained with the other functions. Figure 3.4, in which the solid line is $\kappa_f(A)u$, shows that Algorithm 3.2 with a constant δ is fairly stable under scaling while using a δ that scales with the matrix A (`funm_nd_norm`) can produce large errors when $\|A\|$ is small. This is not unexpected since a smaller δ results in a smaller separation of the blocks and more ill-conditioned Sylvester equations.

In most cases there is no difference in accuracy between the algorithms. The results show no significant benefit of $\delta = 0.2$ over $\delta = 0.1$, and the former produces larger blocks in general, so it increases the cost.

In general, the choice δ in Algorithm 3.2 must be a balance between speed and accuracy, and the optimal choice of δ will be problem dependent. We suggest taking $\delta = 0.1$ as the default blocking parameter in Algorithm 3.2.

Next we set the function f to the sine, the cosine, the hyperbolic sine, and the hyperbolic cosine and use the same set of 35 test matrices as in the previous experiment. We compare the following three algorithms:

- `funm`, the built-in MATLAB function implementing the standard Schur–Parlett algorithm [11] with $\delta = 0.1$;
- `funm_nd`, Algorithm 3.2 with $\delta = 0.1$.
- `funm_nd_`∞, Algorithm 3.2 with $\delta = \infty$ (no blocking, so the whole Schur factor T is computed by Algorithm 3.1).

Note that since we are comparing with the Schur–Parlett algorithm `funm` we are restricted to functions f having a Taylor expansion with an infinite radius of convergence and for which derivatives of all orders can be computed. Also, we exclude the

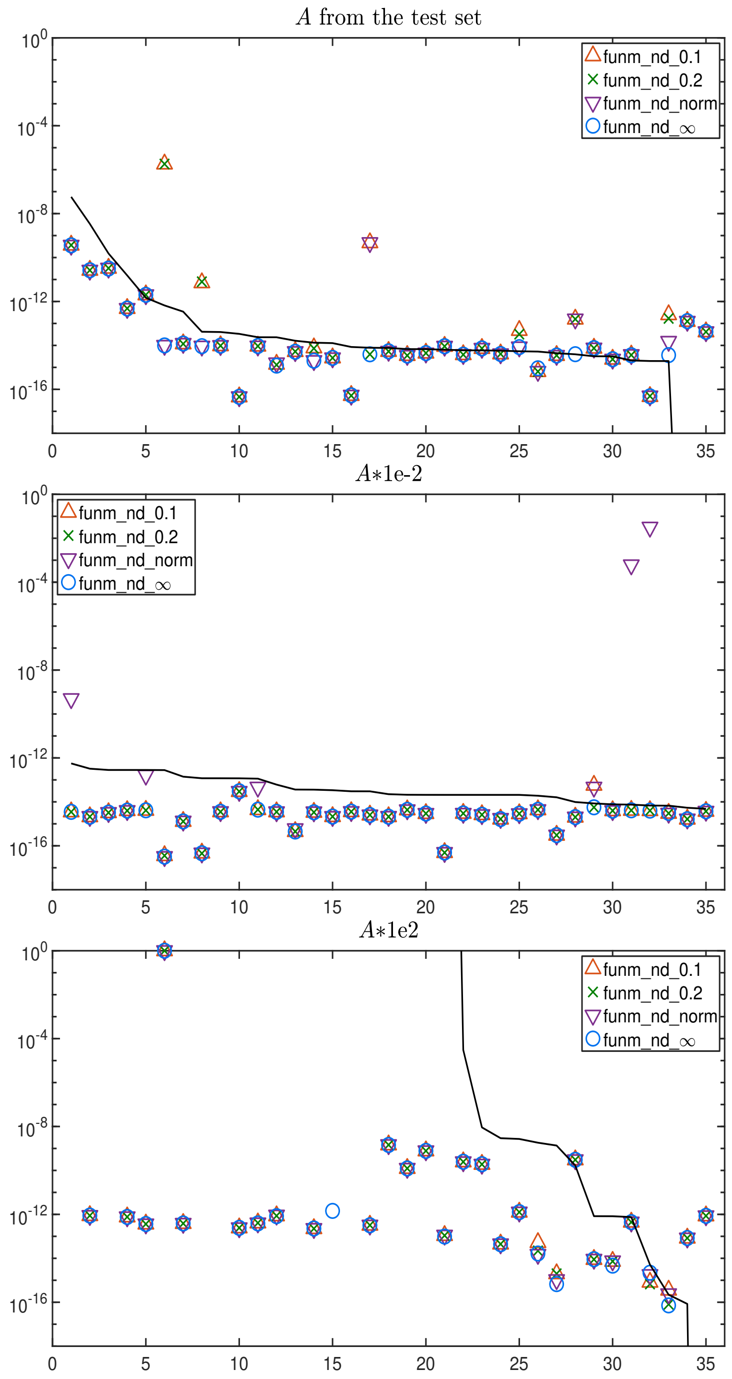


Figure 3.4: Forward normwise relative errors for `funm_nd_0.1`, `funm_nd_0.2`, `funm_nd_norm`, and `funm_nd_∞` on the test set of 35 matrices, for the matrix sine. The solid line is $\kappa_{\sin}(A)u$.

exponential, square root, and logarithm because for these functions the specialized MATLAB codes `expm`, `sqrtm`, and `logm` are preferred to `funm`.

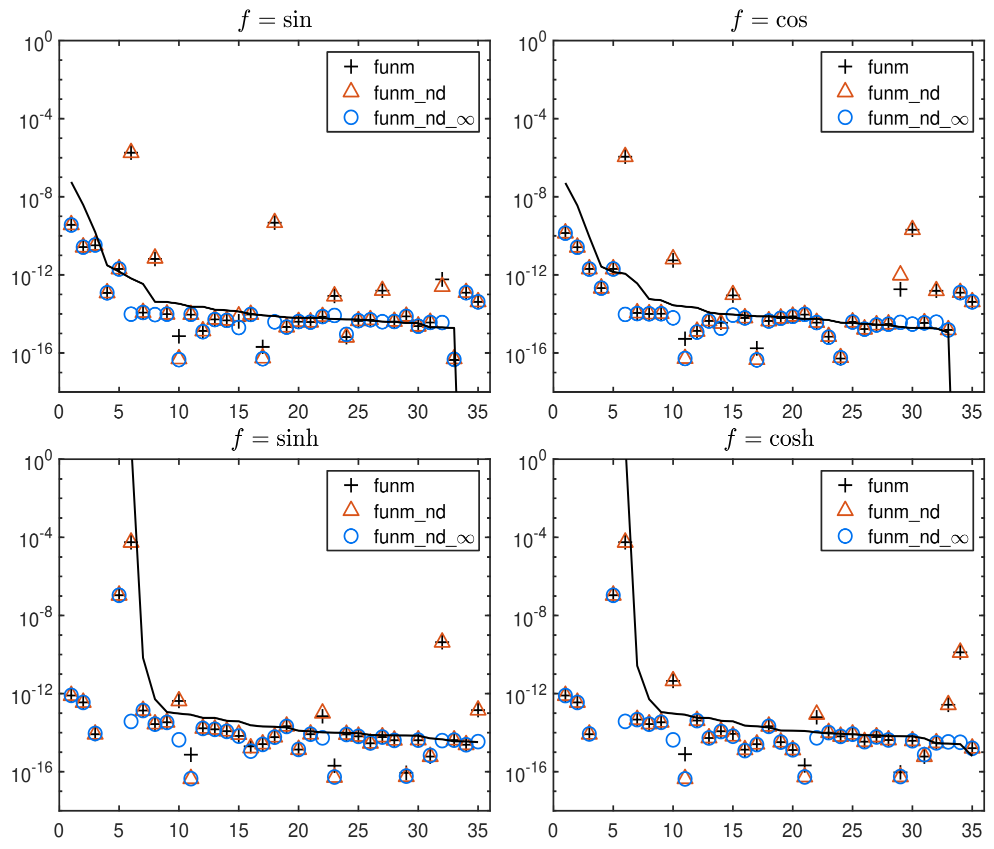


Figure 3.5: Forward normwise relative errors for `funm`, `funm_nd_∞`, and `funm_nd` on the test set of 35 matrices. The solid line is $\kappa_f(A)u$.

Table 3.6: Asymptotic costs in flops of `funm`, `funm_nd`, and `funm_nd_∞`. Here, $n = \sum_{i=1}^{s} m_i$ is the size of the original matrix A, s is the number of diagonal blocks in the Schur form after reordering and blocking, and m_i is the size of the ith block.

	`funm`	`funm_nd`		`funm_nd_∞`	
Precision	u	u	u_h	u	u_h
Flops	$28n^3$ to $n^4/3$	$28n^3$	$2/3\sum_{i=1}^{s} m_i^3$	$28n^3$	$2n^3/3$

From Figure 3.5 we observe that, overall, there is no significant difference between `funm_nd` and `funm` in accuracy, and `funm_nd_∞` is superior to the other algorithms in accuracy, as expected.

We list the computational cost of the three algorithms in flops in Table 3.6. We note that the cost of reordering and blocking, and solving the Sylvester equations that are executed in precision u, is usually negligible compared with the overall cost. For more details of the reordering and partitioning processes of T and evaluating the upper triangular part of $f(A)$ via the block Parlett recurrence, see [11]. In most cases

Table 3.7: Mean execution times (in seconds) and the maximal normwise relative errors over ten runs for `funm`, `funm_nd`, and `funm_nd_∞`, and the maximal block size and the maximal number of equivalent decimal digits used by `funm_nd`.

	Maximal relative error			Mean execution time (secs)				
$f = \sin$	`funm`	`funm_nd`	`funm_nd_∞`	`funm`	`funm_nd`	`funm_nd_∞`	size	digits
A_1, $n = 40$	4.6e-15	4.6e-15	4.6e-15	2.1e-2	4.5e-2	1.4e-1	8	32
A_2, $n = 40$	4.0e-15	4.0e-15	3.9e-15	2.2e-2	2.4e-2	1.4e-1	3	32
A_3, $n = 40$	1.5e-14	7.1e-17	6.8e-17	1.8e-3	4.1e-2	4.1e-2	40	685
A_1, $n = 100$	6.7e-15	6.7e-15	6.7e-15	6.6e-2	1.6e-1	9.7e-1	13	32
A_2, $n = 100$	6.3e-15	6.4e-15	6.3e-15	1.9e-1	1.9e-1	1.0	4	32
A_3, $n = 100$	1.0e-12	5.8e-17	5.8e-17	2.7e-2	7.3e-1	7.4e-1	100	1734
$f = \cosh$	`funm`	`funm_nd`	`funm_nd_∞`	`funm`	`funm_nd`	`funm_nd_∞`	size	digits
A_1, $n = 40$	7.1e-15	7.1e-15	7.1e-15	2.1e-2	4.6e-2	1.4e-1	8	32
A_2, $n = 40$	3.0e-15	2.9e-15	2.9e-15	2.2e-2	2.3e-2	1.4e-1	3	32
A_3, $n = 40$	8.3e-16	9.0e-17	8.0e-17	2.0e-3	4.3e-2	4.3e-2	40	685
A_1, $n = 100$	1.4e-14	1.4e-14	1.4e-14	6.8e-2	1.6e-1	9.9e-1	13	32
A_2, $n = 100$	5.9e-15	5.9e-15	5.9e-15	2.0e-1	2.0e-1	1.0	4	32
A_3, $n = 100$	8.0e-16	5.7e-17	5.8e-17	2.9e-2	7.6e-1	7.6e-1	100	1734

the blocks are expected to be of much smaller dimension than A, especially when n is large. Obviously, `funm_nd` is not more expensive than `funm_nd_∞` and it can be substantially cheaper; indeed `funm_nd` requires no higher than the working precision to evaluate the function on the 1×1 and 2×2 diagonal blocks in the Schur form.

Table 3.7 compares in a working precision of double the mean execution times in seconds and the maximal normwise relative errors of `funm`, `funm_nd`, and `funm_nd_∞` over ten runs, and reports the maximal block size in the reordered and blocked Schur form for each matrix and the maximal number of equivalent decimal digits used by `funm_nd`. We choose $f = \sin$ and $f = \cosh$ and consider the following matrices, generated from built-in MATLAB functions and scaled to different degrees to have nontrivial blocks of the reordered and blocked Schur form in the Schur–Parlett algorithms.

- A_1 = `rand(n)`$/5$.
- A_2 = `randn(n)`$/10$.
- A_3 = `gallery('triw',n,-5)`: upper triangular with 1s on the diagonal and -5s off the diagonal.

We see from Table 3.7 that `funm`, `funm_nd`, and `funm_nd_`∞ provide the same level of accuracy except for one case: $f = \sin$ and A_3. In this case `funm` requires about n Taylor series terms and produces an error several orders of magnitude larger than that of other algorithms. For the matrix A_3 with repeated eigenvalues, `funm_nd` is much slower than `funm` due to the use of higher precision arithmetic in a large block, and in this case there is no noticeable difference in execution time between `funm_nd` and `funm_nd_`∞, which confirms that the cost of the reordering and blocking in `funm_nd` is negligible. For the randomly generated matrices (A_1 and A_2) `funm` can be up to about 2.4 times faster than `funm_nd` ($f = \sin$ and A_1 with $n = 100$), but in some cases when the block size is small `funm_nd` is competitive with `funm` in speed. For these matrices, `funm_nd` is much faster than `funm_nd_`∞.

Finally, we note that Algorithm 3.2 is not restricted only to a working precision of double since its framework is precision independent. For other working precisions suitable values for the parameters c_m, δ_1 and δ may be different, but they can be determined in an approach similar to the one used in this work. The reason for developing Algorithm 3.2 is that it requires only accurate function values and not derivative values. In the next section we consider a function for which accurate derivative values are not easy to compute.

3.6 AN APPLICATION TO THE MATRIX MITTAG–LEFFLER FUNCTION

For $A \in \mathbb{C}^{n\times n}$, the matrix Mittag–Leffler with two parameters $\alpha, \beta \in \mathbb{C}$, $\mathrm{Re}(\alpha) > 0$, is defined by the convergent series

$$E_{\alpha,\beta}(A) = \sum_{k=0}^{\infty} \frac{A^k}{\Gamma(\alpha k + \beta)},$$

where $\Gamma(\gamma), \gamma \in \mathbb{C}\backslash\{0, -1, -2, \dots\}$ is the Euler gamma function. Analogously to the matrix exponential in the solution of systems of linear differential equations, the

Mittag–Leffler function plays an important role in the solution of linear systems of fractional differential equations [21], [40], including time-fractional Schrödinger equations [19], [18] and multiterm fractional differential equations [38]. Despite the importance of the matrix Mittag–Leffler function, little work has been devoted to its numerical computation. In [35], the computation of the action of matrix Mittag–Leffler functions based on Krylov methods is analyzed. The Jordan canonical form and minimal polynomial or characteristic polynomial are considered in [13] for computing the matrix Mittag–Leffler function, but this approach is unstable in floating-point arithmetic.

The work by Garrappa and Popolizio [20] employs the Schur–Parlett algorithm to compute the matrix Mittag–Leffler function. For $z \in \mathbb{C}$, the derivatives of the scalar Mittag–Leffler function are given by

$$E^{(k)}_{\alpha,\beta}(z) = \sum_{j=k}^{\infty} \frac{(j)_k}{\Gamma(\alpha j + \beta)} z^{j-k}, \quad k \in \mathbb{N},$$

where $(j)_k = j(j-1)\cdots(j-k+1)$ is the falling factorial, and are difficult to compute accurately. Garrappa and Popolizio use three approaches, based on series expansion, numerical inversion of the Laplace transform, and summation formulas to compute the derivatives. They exploit certain identities [20, Props. 3–4] to express high-order derivatives in terms of lower order ones, since they observe that all three methods tend to have reduced accuracy for high order derivatives. In fact, almost all of [20] is devoted to the computation of the derivatives. By combining derivative balancing techniques with algorithms for computing the derivatives the authors show in their experiments that the computed $\widehat{E}^{(k)}_{\alpha,\beta}(z)$ have errors

$$\frac{|E^{(k)}_{\alpha,\beta}(z) - \widehat{E}^{(k)}_{\alpha,\beta}(z)|}{1 + |E^{(k)}_{\alpha,\beta}(z)|}$$

that lie "in a range $10^{-13} \sim 10^{-15}$" [20, p. 146]. Now if

$$\frac{|E^{(k)}_{\alpha,\beta}(z) - \widehat{E}^{(k)}_{\alpha,\beta}(z)|}{1 + |E^{(k)}_{\alpha,\beta}(z)|} = \epsilon, \tag{3.17}$$

then the relative error

$$\phi = \frac{|E^{(k)}_{\alpha,\beta}(z) - \widehat{E}^{(k)}_{\alpha,\beta}(z)|}{|E^{(k)}_{\alpha,\beta}(z)|} = \frac{\epsilon}{|E^{(k)}_{\alpha,\beta}(z)|} + \epsilon,$$

so ϵ approximates the relative error for large function values $|E^{(k)}_{\alpha,\beta}(z)|$ and the absolute error when $|E^{(k)}_{\alpha,\beta}(z)|$ is small. However, in floating-point arithmetic it is preferred to use the relative error ϕ to quantify the quality of an approximation. Because they only satisfy (3.17), derivatives computed by the methods of [20] can have large relative errors when $|E^{(k)}_{\alpha,\beta}(z)| \ll 1$. It is hard to identify the range of z, α, β, and k for which $|E^{(k)}_{\alpha,\beta}(z)| < 1$, but intuitively we expect that the kth order derivatives $|E^{(k)}_{\alpha,\beta}(z)|$ will generally decrease with decreasing $|z|$ or increasing β. Since the algorithm of [20] is so far the most practical algorithm for computing the matrix Mittag–Leffler function, we use it as a comparison in testing Algorithm 3.2.

In order to compute a matrix function by Algorithm 3.2 it is necessary to be able to accurately evaluate its corresponding scalar function. For the Mittag–Leffler function, the state-of-the-art algorithm `ml_opc` proposed by Garrappa [17] for computing the scalar function aims to achieve

$$\frac{|E_{\alpha,\beta}(z) - \widehat{E}_{\alpha,\beta}(z)|}{1 + |E_{\alpha,\beta}(z)|} \leqslant 10^{-15}.$$

Hence, in view of the discussion above `ml_opc` can produce large relative errors when $|E_{\alpha,\beta}(z)| \ll 1$ and we do not expect it to provide small relative errors for all arguments.

3.6.1 Numerical experiments

In this section we present numerical tests of Algorithm 3.2 (`funm_nd`). In `funm_nd` the ability to accurately evaluate the scalar Mittag–Leffler function in precisions beyond the working precision is required. We evaluate the scalar Mittag–Leffler function by truncating the series definition and we use a precision a few digits more than the highest precision required by the algorithms for the evaluation of the diagonal blocks.

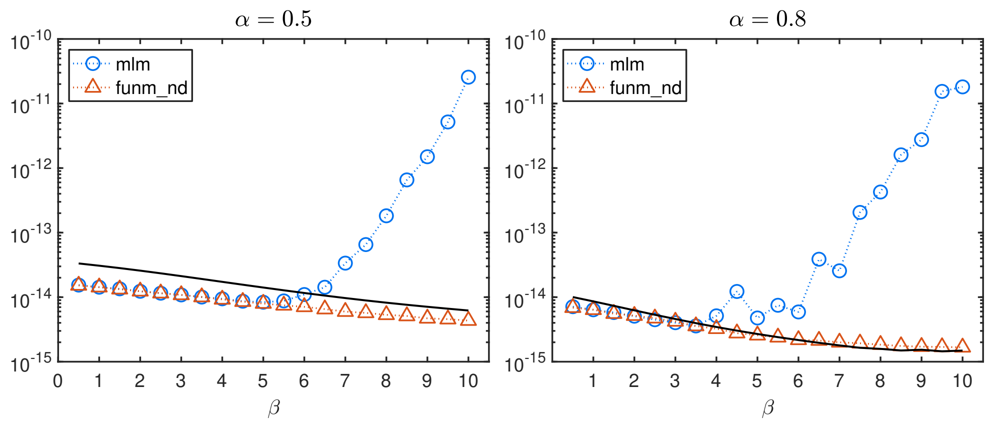


Figure 3.6: Normwise relative errors in the computed $E_{\alpha,\beta}(-R)$ for the Redheffer matrix R and different α and β. The solid lines are $\kappa_{\mathrm{ML}}(R)u$.

In the literature particular attention has been paid to the Mittag–Leffler functions with $0 < \alpha < 1$ and $\beta > 0$ as this is the case that occurs most frequently in applications [18], [35]. In addition to the Mittag–Leffler functions with $\beta \approx 1$ that are often tested in the literature, we will also investigate the cases when β takes other positive values that appear in actual applications. For example, in linear multiterm fractional differential equations the source term can often be approximated by a polynomial, say, $p(t) = \sum_{i=0}^{s} c_i t^i$, and then the solution involves evaluating the matrix Mittag–Leffler functions with $\beta = \alpha + \ell$ for $\ell = 1, 2, \dots, s+1$ [20].

We compare the accuracy of our algorithm `funm_nd` with that of `mlm`, the numerical scheme proposed by Garrappa and Popolizio [20]. The normwise relative error $\|\widehat{X} - E_{\alpha,\beta}(A)\|_F / \|E_{\alpha,\beta}(A)\|_F$ of the computed $\widehat{X}$ is reported, where the reference solution $E_{\alpha,\beta}(A)$ is computed by randomized approximate diagonalization at 200 digit precision. In the plots we also show $\kappa_{\mathrm{ML}}(A)u$, where $\kappa_{\mathrm{ML}}(A)$ is an estimate of the 1-norm condition number of the matrix Mittag–Leffler function.

Example 1*: the Redheffer matrix.* We first use the Redheffer matrix, which is generated by the MATLAB function `gallery('redheff')` and has been used for test purposes in [20]. It is a square matrix R with $r_{ij} = 1$ if i divides j or if $j = 1$ and otherwise $r_{ij} = 0$. The Redheffer matrix has $n - \lfloor \log_2 n \rfloor - 1$ eigenvalues equal to 1 [9], which makes it necessary to evaluate high order derivatives in computing $E_{\alpha,\beta}(-R)$ by means of the standard Schur–Parlett algorithm. The dimension of the matrix is set to $n = 20$.

Table 3.8: Eigenvalues (with multiplicities/numbers) for the matrices in Example 2. Here, $[\ell, r](k)$ means that we take k eigenvalues from the uniform distribution on the interval $[\ell, r]$.

Matrix	Eigenvalues (multiplicities/numbers)	Size
A_{21}	$0(3)$, $\pm 1.0(6)$, $\pm 5(6)$, $-10(3)$	30×30
A_{22}	$\pm[0.9, 1.0](5)$, $\pm[1.2, 1.3](4)$, $\pm[1.4, 1.5](3)$, $\pm[0.9, 1.0] \pm 1i(4)$	40×40

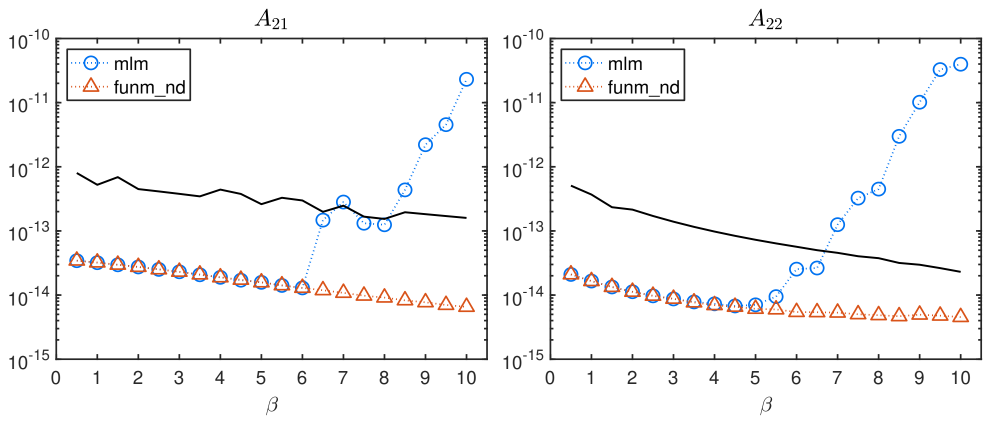


Figure 3.7: Normwise relative errors in the computed $E_{\alpha,\beta}(A)$ for $\alpha = 0.8$ and different β for the matrices in Table 3.8. The solid lines are $\kappa_{\mathrm{ML}}(A)u$.

In this case the Schur–Parlett algorithm `funm_nd` chooses five blocks: one 16×16 block and four 1×1 blocks to compute the matrix Mittag–Leffler functions with $\alpha = 0.5$ or $\alpha = 0.8$ and β starting from 0.5 to 10 with increment 0.5. Figure 3.6 shows that the errors for `funm_nd` are all $O(10^{-14})$ and are below $\kappa_{\mathrm{ML}}(A)u$ for all tested α and β, showing the forward stability of `funm_nd`. On the other hand, for $\beta \geqslant 6.5$, `mlm` produces errors that in general grow with β and become much larger than $\kappa_{\mathrm{ML}}(A)u$, so it is behaving numerically unstably. It is not surprising to see that `mlm` becomes numerically unstable when $\beta = 10$, as it aims to achieve (3.17) and $|E_{\alpha,\beta}(z)|$ decays to 0 when β increases; for example, $|E_{0.5,10}(1)| \approx 4.0 \times 10^{-6}$.

Example 2: matrices with clustered eigenvalues. In the second experiment we test two matrices A_1 and A_2 of size 30×30 and 40×40 with both fixed and randomly generated eigenvalues that are clustered to different degrees, as explained in Table 3.8.

The test matrices were designed to have nontrivial diagonal blocks in the reordered and blocked Schur form. We assigned the specified values to the diagonal matrices and performed similarity transformations with random matrices having a condition

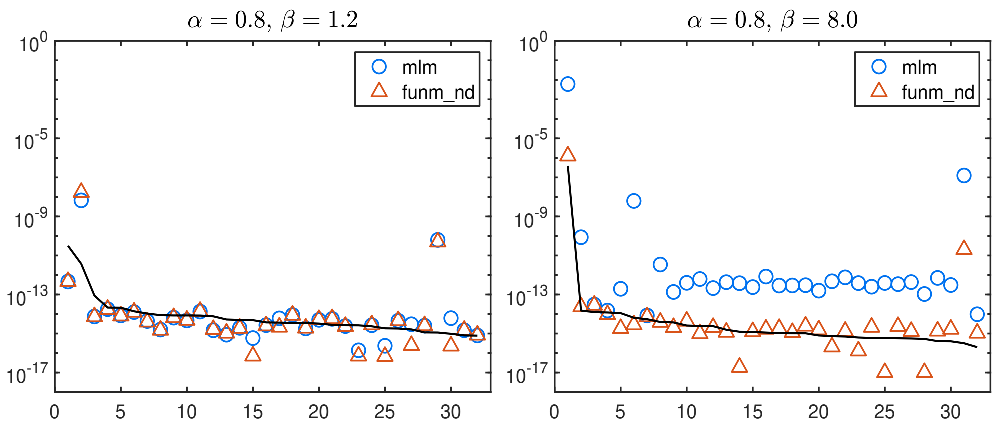


Figure 3.8: Normwise relative errors in the computed $E_{\alpha,\beta}(A)$ for A of size 10×10 from the set of 32 matrices from the MATLAB `gallery`. The solid lines are $\kappa_{\mathrm{ML}}(A)u$.

number of order the matrix size to obtain the full matrices A_{21} and A_{22} with the desired spectrum.

In this example, `funm_nd` chooses six blocks for A_{21} and ten blocks for A_{22}. Figure 3.7 shows that for these matrices `funm_nd` performs in a numerically stable fashion, whereas `mlm` does not for $\beta \geqslant 7$.

Example 3: *matrices from the MATLAB* `gallery`. Now we take 32 matrices of size 10×10 from the MATLAB `gallery` and test the algorithm using the matrix Mittag–Leffler functions with $\alpha = 0.8$ and $\beta = 1.2$ or $\beta = 8.0$. The errors are shown in Figure 3.8. We see that `mlm` is mostly numerically unstable for $\beta = 8$ while `funm_nd` remains largely numerically stable.

One conclusion from these experiments is that by exploiting higher precision arithmetic it is possible to evaluate the Mittag–Leffler function with small relative error even when the function has small norm.

3.7 CONCLUDING REMARKS

We have built a multiprecision algorithm for evaluating analytic matrix functions $f(A)$ that requires only function values and not derivatives. By contrast, the standard Schur–Parlett algorithm, implemented as `funm` in MATLAB, requires derivatives and is applicable only to functions that have a Taylor series with a sufficiently large radius of convergence. Our algorithm needs arithmetic of precision at least double the working precision to evaluate f on the diagonal blocks of order greater than 2 (if there are any) of the reordered and blocked Schur form.

The inspiration for our algorithm is Davies's randomized approximate diagonalization. We have shown that the measure of error that underlies randomized approximate diagonalization makes it unsuitable as a practical means for computing matrix functions. Nevertheless, we have exploited the approximate diagonalization idea within the Schur–Parlett algorithm by making random diagonal perturbations to the nontrivial blocks of order greater than 2 in the Schur form and then diagonalizing the perturbed blocks in higher precision.

Numerical experiments show similar accuracy of our algorithm to `funm`. We found that when applied to the Mittag–Leffler function $E_{\alpha,\beta}$ our algorithm provides results of accuracy at least as good as, and systematically for $\beta \geqslant 6$ much greater than, the special-purpose algorithm `mlm` of [20].

Our multiprecision Schur–Parlett algorithm requires at most $2n^3/3$ flops to be carried out in higher precisions in addition to the approximately $28n^3$ flops at the working precision, and the amount of higher precision arithmetic needed depends on the eigenvalue distribution of the matrix. When there are only 1×1 and 2×2 blocks on the diagonal of the reordered and blocked triangular Schur factor no higher precision arithmetic is required.

Our new algorithm is a useful companion to `funm` that greatly expands the class of readily computable matrix functions. Our MATLAB code `funm_nd` is available on GitHub.[2]

2 https://github.com/Xiaobo-Liu/mp-spalg

REFERENCES

[1] A. H. Al-Mohy and N. J. Higham. "A new scaling and squaring algorithm for the matrix exponential." *SIAM J. Matrix Anal. Appl.* 31.3 (2009), pp. 970–989 (cited on p. 44).

[2] A. H. Al-Mohy and N. J. Higham. "Improved inverse scaling and squaring algorithms for the matrix logarithm." *SIAM J. Sci. Comput.* 34.4 (2012), pp. C153–C169 (cited on p. 44).

[3] E. Anderson. *Robust Triangular Solves for Use in Condition Estimation*. Technical Report CS-91-142. Knoxville, TN, USA: Department of Computer Science, University of Tennessee, Aug. 1991, p. 35. LAPACK Working Note 36 (cited on p. 55).

[4] Z. Bai, J. W. Demmel, and A. McKenney. "On computing condition numbers for the nonsymmetric eigenproblem." *ACM Trans. Math. Software* 19.2 (1993), pp. 202–223 (cited on p. 53).

[5] J. Banks, J. Garza-Vargas, A. Kulkarni, and N. Srivastava. *Pseudospectral Shattering, the Sign Function, and Diagonalization in Nearly Matrix Multiplication Time*. ArXiv:1912.08805. 2019. Revised September 2020 (cited on p. 48).

[6] J. Banks, J. Garza-Vargas, A. Kulkarni, and N. Srivastava. "Pseudospectral Shattering, the Sign Function, and Diagonalization in Nearly Matrix Multiplication Time." In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. 2020, pp. 529–540 (cited on p. 48).

[7] J. Banks, A. Kulkarni, S. Mukherjee, and N. Srivastava. *Gaussian Regularization of the Pseudospectrum and Davies' Conjecture*. ArXiv:1906.11819. 2019. Revised April 2020 (cited on p. 48).

[8] J. Banks, J. G. Vargas, A. Kulkarni, and N. Srivastava. *Overlaps, Eigenvalue Gaps, and Pseudospectrum Under Real Ginibre and Absolutely Continuous Perturbations*. ArXiv:2005.08930. May 2020 (cited on p. 48).

[9] W. W. Barrett and T. J. Jarvis. "Spectral properties of a matrix of Redheffer." *Linear Algebra Appl.* 162-164 (1992), pp. 673–683 (cited on p. 75).

[10] E. B. Davies. "Approximate diagonalization." *SIAM J. Matrix Anal. Appl.* 29.4 (2007), pp. 1051–1064 (cited on pp. 44, 47).

[11] P. I. Davies and N. J. Higham. "A Schur–Parlett algorithm for computing matrix functions." *SIAM J. Matrix Anal. Appl.* 25.2 (2003), pp. 464–485 (cited on pp. 44, 46, 50, 66–68, 70).

[12] J. W. Demmel. "The condition number of equivalence transformations that block diagonalize matrix pencils." *SIAM J. Numer. Anal.* 20.3 (1983), pp. 599–610 (cited on p. 53).

[13] J. Duan and L. Chen. "Solution of fractional differential equation systems and computation of matrix Mittag–Leffler functions." *Symmetry* 10.10 (2018), p. 503 (cited on p. 73).

[14] M. Fasi and N. J. Higham. "An arbitrary precision scaling and squaring algorithm for the matrix exponential." *SIAM J. Matrix Anal. Appl.* 40.4 (2019), pp. 1233–1256 (cited on p. 45).

[15] M. Fasi and N. J. Higham. "Multiprecision algorithms for computing the matrix logarithm." *SIAM J. Matrix Anal. Appl.* 39.1 (2018), pp. 472–491 (cited on p. 45).

[16] M. Fasi, N. J. Higham, and B. Iannazzo. "An algorithm for the matrix Lambert W function." *SIAM J. Matrix Anal. Appl.* 36.2 (2015), pp. 669–685 (cited on p. 45).

[17] R. Garrappa. "Numerical evaluation of two and three parameter Mittag-Leffler functions." *SIAM J. Numer. Anal.* 53.3 (2015), pp. 1350–1369 (cited on p. 74).

[18] R. Garrappa, I. Moret, and M. Popolizio. "On the time-fractional Schrödinger equation: theoretical analysis and numerical solution by matrix Mittag-Leffler functions." *Comput. Math. Applic.* 74.5 (2017), pp. 977–992 (cited on pp. 73, 75).

[19] R. Garrappa, I. Moret, and M. Popolizio. "Solving the time-fractional Schrödinger equation by Krylov projection methods." *J. Comp. Phys.* 293 (2015), pp. 115–134 (cited on p. 73).

[20] R. Garrappa and M. Popolizio. "Computing the matrix Mittag-Leffler function with applications to fractional calculus." *J. Sci. Comput.* 77.1 (2018), pp. 129–153 (cited on pp. 73–75, 78).

[21] R. Garrappa and M. Popolizio. "On the use of matrix functions for fractional partial differential equations." *Math. Comput. Simulation* C-25.81 (2011), pp. 1045–1056 (cited on p. 73).

[22] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. 2nd ed., Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. xxx+680 (cited on p. 53).

[23] N. J. Higham. "Computing real square roots of a real matrix." *Linear Algebra Appl.* 88/89 (1987), pp. 405–430 (cited on p. 49).

[24] N. J. Higham. *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008, pp. xx+425 (cited on pp. 44, 47, 49, 52, 59, 65).

[25] N. J. Higham. "Short codes can be long on insight." *SIAM News* 50.3 (Apr. 2017), pp. 2–3 (cited on p. 48).

[26] N. J. Higham. *The Matrix Computation Toolbox*. `http://www.maths.manchester.ac.uk/~higham/mctoolbox` (cited on p. 68).

[27] N. J. Higham. *The Matrix Function Toolbox*. `http://www.maths.manchester.ac.uk/~higham/mftoolbox` (cited on pp. 59, 65).

[28] N. J. Higham and A. H. Al-Mohy. "Computing matrix functions." *Acta Numerica* 19 (2010), pp. 159–208 (cited on p. 44).

[29] N. J. Higham and E. Hopkins. *A Catalogue of Software for Matrix Functions. Version 3.0*. MIMS EPrint 2020.7. UK: Manchester Institute for Mathematical Sciences, The University of Manchester, Mar. 2020, p. 24 (cited on p. 44).

[30] *IEEE Standard for Floating-Point Arithmetic, IEEE Std* 754-2019 (*Revision of IEEE* 754-2008). New York, USA: The Institute of Electrical and Electronics Engineers, 2019, p. 82 (cited on p. 55).

[31] V. Jain, A. Sah, and M. Sawhney. *On the Real Davies' Conjecture*. ArXiv:2005.08908v2. May 2020 (cited on p. 48).

[32] C. S. Kenney and A. J. Laub. "Condition estimates for matrix functions." *SIAM J. Matrix Anal. Appl.* 10.2 (1989), pp. 191–209 (cited on p. 44).

[33] C. S. Kenney and A. J. Laub. "Small-sample statistical condition estimates for general matrix functions." *SIAM J. Sci. Comput.* 15.1 (1994), pp. 36–61 (cited on p. 56).

[34] C. B. Moler and C. F. Van Loan. "Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later." *SIAM Rev.* 45.1 (2003), pp. 3–49 (cited on p. 44).

[35] I. Moret and P. Novati. "On the convergence of Krylov subspace methods for matrix Mittag–Leffler functions." *SIAM J. Numer. Anal.* 49.5 (Oct. 2011), pp. 2144–2164 (cited on pp. 73, 75).

[36] *Multiprecision Computing Toolbox*. Advanpix, Tokyo, Japan. `http://www.advanpix.com` (cited on pp. 49, 62).

[37] B. N. Parlett. *Computation of Functions of Triangular Matrices*. Memorandum ERL-M481. Berkeley: Electronics Research Laboratory, College of Engineering, University of California, Nov. 1974, p. 18 (cited on p. 46).

[38] M. Popolizio. "Numerical solution of multiterm fractional differential equations using the matrix Mittag–Leffler functions." *Mathematics* 6.1 (2018), p. 7 (cited on p. 73).

[39] J. D. Roberts. "Linear model reduction and solution of the algebraic Riccatiq equation by use of the sign function." *Internat. J. Control* 32.4 (Oct. 1980), pp. 677–687 (cited on p. 44).

[40] M. R. Rodrigo. "On fractional matrix exponentials and their explicit calculation." *J. Differential Equations* 261.7 (2016), pp. 4223–4243 (cited on p. 73).

[41] A. Schwarz and C. C. K. Mikkelsen. "Robust Task-Parallel Solution of the Triangular Sylvester Equation." In: *Parallel Processing and Applied Mathematics*. Ed. by

R. Wyrzykowski, E. Deelman, J. Dongarra, and K. Karczewski. Springer, Cham, Switzerland, 2020, pp. 82–92 (cited on p. 55).

[42] A. Schwarz, C. C. K. Mikkelsen, and L. Karlsson. "Robust parallel eigenvector computation for the non-symmetric eigenvalue problem." *Parallel Comput.* 100 (Dec. 2020), p. 102707 (cited on p. 55).

4 ARBITRARY PRECISION ALGORITHMS FOR COMPUTING THE MATRIX COSINE AND ITS FRÉCHET DERIVATIVE

Abstract. Existing algorithms for computing the matrix cosine are tightly coupled to a specific precision of floating-point arithmetic for optimal efficiency so they do not conveniently extend to an arbitrary precision environment. We develop an algorithm for computing the matrix cosine that takes the unit roundoff of the working precision as input, and so works in an arbitrary precision. The algorithm employs a Taylor approximation with scaling and recovering and it can be used with a Schur decomposition or in a decomposition-free manner. We also derive a framework for computing the Fréchet derivative, construct an efficient evaluation scheme for computing the cosine and its Fréchet derivative simultaneously in arbitrary precision, and show how this scheme can be extended to compute the matrix sine, cosine, and their Fréchet derivatives all together. Numerical experiments show that the new algorithms behave in a forward stable way over a wide range of precisions. The transformation-free version of the algorithm for computing the cosine is competitive in accuracy with the state-of-the-art algorithms in double precision and surpasses existing alternatives in both speed and accuracy in working precisions higher than double.

Keywords: multiprecision algorithm, multiprecision arithmetic, matrix cosine, matrix exponential, matrix function, Fréchet derivative, double angle formula, Taylor approximation, forward error analysis, MATLAB.

2010 MSC: 15A16, 65F30, 65F60.

4.1 INTRODUCTION

Matrix functions have been the subject of much research because of their many applications in science and engineering. The matrix exponential is the most studied function thanks to its crucial role in representing the solutions of linear first order differential equations. The matrix sine and cosine, which can be defined for $A \in \mathbb{C}^{n\times n}$ by their Maclaurin series

$$\cos A = I - \frac{A^2}{2!} + \frac{A^4}{4!} - \frac{A^6}{6!} + \cdots,$$

$$\sin A = A - \frac{A^3}{3!} + \frac{A^5}{5!} - \frac{A^7}{7!} + \cdots,$$

where I denotes the identity matrix of order n, play an analogous role for second order differential equations. For example, the second order system

$$y''(t) + Ay(t) = g(t), \quad y(0) = y_0, \quad y'(0) = y'_0, \tag{4.1}$$

which appears in finite element semidiscretization of the wave equation, has the solution

$$y(t) = \cos(\sqrt{A}t)y_0 + (\sqrt{A})^{-1}\sin(\sqrt{A}t)y'_0 + \int_0^t (\sqrt{A})^{-1}\sin\big(\sqrt{A}(t-s)\big)g(s)\mathrm{d}s,$$

where $\sqrt{A}$ denotes any square root of A [38]. For generalizations of the system (4.1) and other applications see [5] and the references therein.

In recent years there has been a growing interest in multiprecision algorithms for computing matrix functions. Several algorithms that work in arbitrary precision have been developed, including algorithms for the matrix exponential [8], [15], the matrix logarithm [16], and general matrix functions [23]. We note that the cosine and sine of a matrix can be computed via the the matrix exponential by exploiting the matrix analogue of Euler's formula, $\mathrm{e}^{\mathrm{i}A} = \cos A + \mathrm{i}\sin A$, and this idea is implemented in the mpmath library [29], but complex arithmetic needs to be used even for a real A. The Multiprecision Computing Toolbox [32] offers functions that can evaluate in arbitrary precision the sine and cosine of a matrix using the Schur–Parlett algorithm [9]. We are not aware of any specialized algorithm for computing the matrix cosine in arbitrary precision.

The need for arbitrary precision algorithms for matrix trigonometric functions arises from their inclusion in many languages and libraries that attempt to offer arbitrary precision implementations of various functions with both scalar and matrix arguments, including the software mentioned above, as well as the Julia language [7] and Python's SymPy [31], for example. Furthermore, from the aspect of algorithm development, a reference solution computed in higher precision is required to estimate the forward error of algorithms for these matrix functions.

In this work we develop a new arbitrary precision algorithm for computing the matrix cosine. The algorithm uses a Taylor approximant to $\cos(2^{-s}A)$ in conjunction with the double angle recurrence $\cos(2A) = 2\cos^2 A - I$, and we refer to this process as scaling and recovering. The algorithmic parameters s and the degree of the approximant are determined from a relative forward error bound for the approximant. The algorithm takes the working precision as an input argument and can compute the Fréchet derivative $L_{\cos}(A, E)$ (defined in section 4.5) simultaneously.

We begin in section 4.2 by reviewing previous work on computing the matrix cosine and explaining why existing algorithms are not suitable for arbitrary precision arithmetic. In section 4.3 we derive a bound on the norm of the forward error of a Taylor approximant to the matrix cosine. Based on this error bound we develop an algorithm for evaluating the matrix cosine in arbitrary precision in section 4.4. In section 4.5 we derive a framework for computing $L_{\cos}(A, E)$ by Fréchet differentiating our algorithm for $\cos A$, and we construct an efficient evaluation scheme for computing $\cos A$ and $L_{\cos}(A, E)$ simultaneously. An algorithm for computing $\cos A$ and $L_{\cos}(A, E)$ in arbitrary precision is developed. We also discuss an extension of the evaluation scheme to the matrix sine and its Fréchet derivative. We then test the algorithms developed in the previous sections experimentally and compare their performance against alternative approaches in section 4.6. Conclusions are drawn in section 4.7.

Throughout this work we denote by $\|\cdot\|$ any consistent matrix norm, by $\mathbb{N}$ the set of nonnegative integers, and by $\mathbb{N}^+$ the set of positive integers. We denote by u the unit roundoff of the floating-point arithmetic.

4.2 PREVIOUS WORK

The focus in the literature has been on computing the matrix cosine rather than the matrix sine, as the sine can be obtained with a cosine algorithm by using the identity $\sin A = \cos(A - \frac{\pi}{2}I)$. The most popular and successful method for computing the cosine of a matrix is the scaling and recovering algorithm. It uses a rational or polynomial approximation to $\cos(2^{-s}A)$ in conjunction with scaling and recovering [20, Thm. 12.1]. The algorithm was first suggested by Serbin and Blalock [39], though they did not propose a concrete scheme for choosing the algorithmic parameters.

Higham and Smith [27] develop an algorithm that scales the matrix such that $\|2^{-s}A\|_\infty \leqslant 1$ and employs a diagonal Padé approximant of fixed degree 8, where ad hoc analysis shows that this choice provides full normwise relative accuracy in IEEE double precision arithmetic. Diagonal Padé approximants are preferred over non-diagonal ones, as symmetries in the coefficients of the numerator and denominator can be utilized for efficient evaluation of the approximant. Hargreaves and Higham [19] derived an algorithm that chooses the degree of the diagonal Padé approximant adaptively to minimize the computational cost subject to achieving a desired absolute error bound. Since then the strategy of using variable degree approximants has been widely adopted. Sastre et al. [35] propose an algorithm that uses Taylor series approximations with sharper absolute error bounds derived using ideas similar to those in [2, sect. 4]. The derivation of these algorithms is based on forward error bounds. Al-Mohy, Higham, and Relton [5] develop algorithms that are based on backward error analysis and Padé approximants to $\sin x$ and e^x, and they can compute the matrix sine and cosine separately or simultaneously. Another algorithm that can calculate the two functions simultaneously is proposed by Seydaoglu, Bader, Blanes, and Casas [40]; it chooses from some Taylor polynomial approximations of fixed degree and relies on precomputed constants. Other algorithms have been developed for computing the matrix cosine based on Taylor series [6], [36], [37], with improvements on the error bounds or the cost of evaluation of the approximating polynomials. There are also algorithms for evaluating the matrix cosine based on approximating functions other than Taylor and Padé approximants, for example, algorithms based on Bernoulli matrix polynomials [10] and Hermite matrix polynomials [11].

The algorithms mentioned above require computing symbolically in high precision certain constants that depend on the working precision, and these constants are crucial for selecting algorithmic parameters since they appear in the truncation error bounds or are the coefficients of the approximating functions. For example, the algorithm of [19, sect. 3] is based on the absolute forward error bound

$$\|\cos A - r_{2m}(A)\|_\infty = \left\| \sum_{i=2m+1}^{\infty} g_{2i} A^{2i} \right\|_\infty \leqslant \sum_{i=2m+1}^{\infty} |g_{2i}| \|A^2\|_\infty^i, \tag{4.2}$$

where r_{2m} is the diagonal Padé approximant of degree $2m$ to the cosine. Then for some chosen values of m, symbolic and high precision computation are used, respectively, in computing the coefficients g_{2i} and the quantity

$$\theta_{2m} = \max \left\{ \theta : \sum_{i=2m+1}^{\infty} |g_{2i}| \theta^{2i} \leqslant \tau \right\},$$

where $\tau = 2^{-53}$ is the unit roundoff of double precision, so that (4.2) ensures an error not exceeding τ as long as $\|A^2\|_\infty^{1/2} \leqslant \theta_{2m}$. The same mechanism is used in other existing algorithms, based on either forward or backward error analysis, for computing the matrix cosine. Therefore, none of these algorithms conveniently extends to an arbitrary precision environment since it is impractical to carry out this procedure when the accuracy at which the function should be evaluated is known only at run time. Hence a new approach is required for computing the matrix cosine in arbitrary precision.

4.3 FORWARD ERROR ANALYSIS FOR THE MATRIX COSINE

Padé approximation has been widely adopted in algorithms, especially arbitrary precision algorithms, for evaluating matrix functions, including the matrix logarithm [16] and the matrix exponential [15]. In comparison with the exponential and logarithm functions, relatively few results are available concerning Padé approximants of the cosine function. In particular, we are not aware of a proof of existence of the Padé approximants for arbitrary degrees. Magnus and Wynn [30] give the coefficients of the Padé approximants of the scalar cosine function in terms of determinants of matrices whose entries are binomial coefficients, but these expressions are not useful for deriving a general error bound. For this reason, we employ the scaling and recovering idea and bound the relative forward error of the truncated Taylor approximant to the cosine. The techniques used in [15, sect. 3] for bounding the forward error of a Taylor approximant as an approximation to the matrix exponential do not generalize to the matrix cosine, because the terms in its Taylor expansion alternate in sign, but we can derive computable error bounds by using the hyperbolic cosine.

Let

$$t_{2m}^{c}(A) := \sum_{i=0}^{m} \frac{(-1)^i}{(2i)!} A^{2i}, \qquad t_{2m}^{ch}(A) := \sum_{i=0}^{m} \frac{1}{(2i)!} A^{2i} \tag{4.3}$$

denote the Taylor approximants of order $2m$ to $\cos A$ and $\cosh A$, respectively and let $B = A^2$. Then we have

$$\begin{aligned} \|\cos A - t_{2m}^{c}(A)\| &= \left\| \sum_{i=m+1}^{\infty} \frac{(-1)^i}{(2i)!} A^{2i} \right\| = \left\| \sum_{i=m+1}^{\infty} \frac{(-1)^i}{(2i)!} B^{i} \right\| \leqslant \sum_{i=m+1}^{\infty} \frac{1}{(2i)!} \alpha_m(B)^i \\ &= \sum_{i=m+1}^{\infty} \frac{1}{(2i)!} \big(\sqrt{\alpha_m(B)}\big)^{2i} = \cosh\big(\sqrt{\alpha_m(B)}\big) - t_{2m}^{ch}\big(\sqrt{\alpha_m(B)}\big) \end{aligned} \tag{4.4}$$

by [2, Thm. 4.2(a)], where

$$\begin{aligned}\alpha_m(B) &\in \mathcal{A}_m(B) \\ &:= \big\{\max\big(\|B^d\|^{1/d}, \|B^{d+1}\|^{1/(d+1)}\big) : d \in \mathbb{N}^+, d(d-1) \leqslant m+1\big\}.\end{aligned} \tag{4.5}$$

For nonnormal matrices this bound can be much smaller than a simpler bound based on a single power of A, such as [20, eq. (4.9)]. The main concern with the latter bound is that it can be arbitrarily loose for nonnormal A due to the use of the potentially arbitrarily weak inequality $\|A^k\| \leqslant \|A\|^k$ for $k \in \mathbb{N}^+$ in its derivation, as discussed in [2, sect. 1], [20, p. 288].

We note that elements in $\mathcal{A}_m(B)$ are of the form $\|B^d\|^{1/d}$ for some $d \in \mathbb{N}^+$, and the size of $\mathcal{A}_m(B)$ depends on m. In fact, we could instead apply [2, Thm. 4.2(b)] in (4.4), as Al-Mohy does in [1, eq. (3.2)], and this would lead to exactly the same bound. Nadukandi and Higham [33] show that the use of

$$\widetilde{\alpha}_m(B) := \min\big\{\max\big(\|B^a\|^{1/a}, \|B^b\|^{1/b}\big) : a, b \in \mathbb{N}^+,\ \gcd(a,b) = 1,\ ab - a - b < m+1\big\}$$

in place of $\alpha_m(B)$ results in a more refined bound, but it requires considerably more computation, which can be undesirable in high precision, as discussed in [15, sect. 3.1]. For this reason we will choose an $\alpha_m(B)$ from $\mathcal{A}_m(B)$ defined in (4.5), but it should be noted that all the results in this section remain true with $\alpha_m(B)$ replaced by $\widetilde{\alpha}_m(B)$.

Ideally, in designing an algorithm for computing the matrix cosine we would like to use in the bound (4.4) the quantity

$$\alpha_m^{\mathsf{opt}}(B) = \min\{\, \alpha_m : \alpha_m \in \mathcal{A}_m(B)\,\},$$

in order to obtain the sharpest bounds, since these bounds are obviously increasing in $\alpha_m(B)$. However, to find $\alpha_m^{\mathsf{opt}}(B)$ we would need to search over $\mathcal{D} := \{d \in \mathbb{N}^+ : d(d-1) \leqslant m+1\}$, and this set has $\lfloor (1 + \sqrt{4m+5})/2 \rfloor$ elements. This makes computation of $\alpha_m^{\mathsf{opt}}(B)$ impractical since the value of m can be large for an algorithm aiming for an arbitrary precision environment. More importantly, it has been observed [2] that for nonnormal matrices the sequence $\{\|B^k\|^{1/k}\}$ is typically roughly decreasing despite possible considerably nonmonotonic behavior, so it is reasonable and effective to employ the considerably cheaper approximation to $\alpha_m^{\mathtt{opt}}(B)$,

$$\alpha_m^*(B) = \max\left(\|B^{d^*}\|^{1/d^*}, \|B^{d^*+1}\|^{1/(d^*+1)}\right),\ d^* = \max_d \mathcal{D} = \left\lfloor \frac{1+\sqrt{4m+5}}{2} \right\rfloor, \tag{4.6}$$

and this strategy has been shown to be effective for computing the matrix exponential in arbitrary precision by Fasi and Higham [15].

4.4 A MULTIPRECISION ALGORITHM FOR THE MATRIX COSINE

In this section we build a novel algorithm for computing the matrix cosine in arbitrary precision floating-point arithmetic based upon the Taylor approximant t_{2m}^c of (4.3) together with the scaling and recovering idea. There are two algorithmic parameters: m, which relates to the order of approximation, and s, the number of scalings in $X = 2^{-s}A$ (or $Y = 4^{-s}B$, as we work with $B = A^2$), to be determined in order to guarantee that

$$\|\cos X - t_{2m}^c(X)\| \lesssim u\|\cos X\|. \tag{4.7}$$

By (4.4), a sufficient condition for (4.7) to hold is

$$\cosh\left(\sqrt{\alpha_m^*(4^{-s}B)}\right) - t_{2m}^{ch}\left(\sqrt{\alpha_m^*(4^{-s}B)}\right) \lesssim u\|\cos(2^{-s}A)\|. \tag{4.8}$$

We employ the Paterson–Stockmeyer method [34], which is the customary choice in the literature, to evaluate

$$t_{2m}^c(X) = \sum_{i=0}^{m} \frac{(-1)^i}{(2i)!} X^{2i} = \sum_{i=0}^{m} \frac{(-1)^i}{(2i)!} (4^{-s}B)^i =: p_m^c(4^{-s}B), \tag{4.9}$$

which is a polynomial (in $B = A^2$) of degree m. It is important to note that, in choosing the degree m and hence the corresponding approximants, only those that maximize the approximation degree for a given number of matrix multiplications are worth considering. For evaluating polynomial approximants p_m^c to the cosine by means of the Paterson–Stockmeyer method, the sequence of optimal degrees is [14, eq. (14)]

$$\mathtt{m}_i := \left\lfloor \frac{(i+2)^2}{4} \right\rfloor, \quad i \in \mathbb{N}.$$

To evaluate the approximant $p_{\mathtt{m}_i}^c(B)$, it is known that at least the first $\nu = \lfloor\sqrt{\mathtt{m}_i}\rfloor$ powers of B will be required [18, Thm. 1.7.4]. Hence we can form the first ν powers of B immediately after the degree m is chosen, which can be used to reduce the computational cost of evaluating $\alpha_m^*(4^{-s}B)$. Note that $\|(4^{-s}B)^d\|^{1/d} = 4^{-s}\|B^d\|^{1/d}$ and thus $\alpha_m^*(4^{-s}B) = 4^{-s}\alpha_m^*(B)$, so we could compute the norms of powers of B and then perform scaling as required. More computational effort can be saved in finding $\alpha_m^*(B)$ by estimating the 1-norms of powers of B since it is enough to evaluate accurately the order of magnitude of $\alpha_m^*(B)$. We adapt the numerical scheme used by Fasi and Higham [15, Frag. 4.5] for estimating $\|B^d\|_1$, $d \in \mathbb{N}^+$. The algorithm efficiently computes B^dW using available powers of B, where $W \in \mathbb{C}^{n\times t}$, with $t \ll n$, and integrates this process with the block 1-norm estimation algorithm NORMEST1 proposed by Higham and Tisseur [28] that repeatedly computes the action of B on W, without explicitly forming any powers of B. This algorithm requires only $O(n^2)$ flops.

The technique proposed by Fasi and Higham [15, sect. 4.1] can be exploited to obtain a sharper bound at almost no extra cost, by reusing quantities computed during previous steps of the algorithm. Since the algorithm considers the approximants in nondecreasing order of cost, the value of $d^*(\mathtt{m}_i)$ in (4.6) is nondecreasing in i. Hence, in the process of seeking suitable a degree parameter we can use a variable $\alpha_{\min}$ to keep track of the smallest value of $\alpha_{\mathtt{m}_i}^*(B)$ computed up to now, and update it when a new value $\alpha_{\mathtt{m}_j}^*(B) < \alpha_{\min}$ is found for some $j > i$, and in practical calculation we use this $\alpha_{\min}$ to replace $\alpha_m^*(B)$.

On the other hand, we do not know $\|\cos(2^{-s}A)\| = \|\cos X\|$ a priori, so we could use a lower bound for the norm of $\cos X$, such as those presented in [20, Thm. 12.3]. In fact, we can even derive a sharper bound by exploiting the result in [2, Thm. 4.2(a)]: with $Y = X^2$,

$$\begin{aligned}\|\cos X\| &= \left\| I + \sum_{i=1}^{\infty} \frac{(-1)^i}{(2i)!} X^{2i} \right\| \geqslant 1 - \left\| \sum_{i=1}^{\infty} \frac{(-1)^i}{(2i)!} Y^i \right\| \\ &\geqslant 1 - \sum_{i=1}^{\infty} \frac{\left(\sqrt{\alpha_m(Y)}\right)^{2i}}{(2i)!} = 2 - \cosh\left(\sqrt{\alpha_m(Y)}\right).\end{aligned}$$

However, to use this bound or those in [20, Thm. 12.3] it is required that $\theta < \cosh^{-1}(2)$, for $\theta = \sqrt{\alpha_m(Y)}$ or $\theta = \sqrt{\|Y\|}$. This condition on the norm of the scaled matrix $Y = 4^{-s}A^2$ can require a very large s (when $\|A\|$ is large), which means a large number of double angle recurrence steps. This potentially rigid restriction on s is undesirable, especially for an algorithm aiming for arbitrary precision. Alternatively, an absolute bound can be used in developing the algorithm, for example [19], [27], [35], and clearly this is reasonable if $\|Y\|$ is not too large. In our algorithm we truncate the Taylor series to obtain the practical approximation

$$\cos(2^{-s}A) \approx \sum_{i=0}^{\ell} \frac{(-1)^i}{(2i)!} (2^{-s}A)^{2i} = \sum_{i=0}^{\ell} \frac{(-4^{-s})^i}{(2i)!} \mathcal{B}_i, \quad \ell = \mathrm{length}(\mathcal{B}), \tag{4.10}$$

where $\mathcal{B} = \{I, B, B^2, \dots\}$ is an array that stores the powers of $B = A^2$, and $\mathrm{length}(\mathcal{B})$ is the number powers in $\mathcal{B}$ with positive exponents. This is the best approximation we currently have to $\cos(2^{-s}A)$. In practice, we can evaluate (4.10) in a lower precision (for example, single or double precision if the working precision is higher than double) since it suffices to obtain the correct order of magnitude of $\|\cos(2^{-s}A)\|$ in the bound (4.7), and in fact this is necessary for better efficiency considering that we have to recompute the coefficients in (4.10) when s is changed. We update $\mathcal{B}$ when it does not contain the first $\lfloor\sqrt{m}\rfloor$ powers of B, so the value of $\mathrm{length}(\mathcal{B})$ varies with the degree m. Since the estimate (4.10) uses only the powers of B that are available in $\mathcal{B}$, it requires only $O(n^2)$ flops.

Fragment 4.1: Error bound checking for the matrix cosine.

```
1  function EvalBound($B \in \mathbb{C}^{n\times n}$, $m, s \in \mathbb{N}$)
   ▷ Check (4.8) using elements in $\mathcal{B}$. Lines 2–3 are executed in precision $u$, line 10 is
     executed in precision $u^{1.2}$, and the other lines can be executed in a precision lower
     than precision $u$.
2  for $i \leftarrow \mathrm{length}(\mathcal{B}) + 1$ to $\lfloor\sqrt{m}\rfloor$ do
3      $\mathcal{B}_i = \mathcal{B}_{i-1}B$
4  $d^* \leftarrow \lfloor\frac{1+\sqrt{4m+5}}{2}\rfloor$
5  if $b_{d^*} = -\infty$ then
6      $b_{d^*} \leftarrow$ normest1$(\lambda x.\textsc{EvalPowVec}(d^*, x))^{1/d^*}$
7  if $b_{d^*+1} = -\infty$ then
8      $b_{d^*} \leftarrow$ normest1$(\lambda x.\textsc{EvalPowVec}(d^*+1, x))^{1/(d^*+1)}$
9  $\alpha_{\min} \leftarrow \min\{\max\{b_{d^*}, b_{d^*+1}\}, \alpha_{\min}\}$
10 $\delta_{\mathrm{nxt}} \leftarrow \cosh(\sqrt{4^{-s}\alpha_{\min}}) - t^{ch}_{2\mathtt{m}_i}(\sqrt{4^{-s}\alpha_{\min}})$
11 $M \leftarrow \sum_{i=0}^{\mathrm{length}(\mathcal{B})} \frac{(-4^{-s})^i}{(2i)!}\mathcal{B}_i$
12 $\phi \leftarrow$ normest1$(\lambda x.Mx)$
13 return $\delta_{\mathrm{nxt}}$, $\phi$
```

```
14 function EvalPowVec($W \in \mathbb{C}^{n\times t}$, $d \in \mathbb{N}$)
   ▷ Compute $B^dW$ using elements in $\mathcal{B}$.
15 $\ell \leftarrow \mathrm{length}(\mathcal{B})$
16 while $d > 0$ do
17     for $i \leftarrow 1$ to $\lfloor d/\ell\rfloor$ do
18         $W \leftarrow \mathcal{B}_\ell W$
19     $d \leftarrow d \mod \ell$
20     $\ell \leftarrow \min\{\ell - 1, d\}$
21 return $W$
```

The function EvalBound in Fragment 4.1 shows how the bound (4.8) can be evaluated efficiently using the techniques discussed above. We use some extra precision in forming the sum in the t^{ch}_{2m} term in the error bound (4.8) to guarantee sufficient accuracy, and we found this strategy makes no noticeable difference to the speed. In our implementation we only compute and store these scalar coefficients at run time when the order increases from $\mathtt{m}_i$ to $\mathtt{m}_{i+1}$, so each of the coefficients is calculated at most once. Within the 1-norm estimating function normest1 in EvalBound, we have used the lambda syntax from lambda calculus for an anonymous function: $\lambda x.f(x)$ denotes a function that replaces all the occurrences of x in the body of f with the value of its input argument.

Fragment 4.2: Modified Paterson–Stockmeyer algorithm for the cosine.

```
1  function PSEvalCos($B \in \mathbb{C}^{n\times n}$, $m, s \in \mathbb{N}$)
   ▷ Evaluate $\sum_{i=0}^{m} c_i(4^{-s}B)^i$ using elements of $\mathcal{B}$.
2  for $i \leftarrow 0$ to $m$ do
3      $c_i \leftarrow (-1)^i/(2i)!$
4  $\nu \leftarrow \lfloor\sqrt{m}\rfloor$
5  $\mu \leftarrow \lfloor m/\nu\rfloor$
6  for $i \leftarrow \text{length}(\mathcal{B})+1$ to $\nu$ do
7      $\mathcal{B}_i \leftarrow \mathcal{B}_{i-1}B$
8  $C \leftarrow \sum_{j=0}^{m-\mu\nu} c_{\mu\nu+j}4^{-sj}\mathcal{B}_j$
9  for $i \leftarrow \mu-1$ down to $0$ do
10     $C \leftarrow 4^{-s\nu}C\mathcal{B}_\nu + \sum_{j=0}^{\nu-1} c_{\nu i+j}4^{-sj}\mathcal{B}_j$
11 return $C$
```

For a chosen combination of s and m, if the bound (4.8) is not satisfied we can either increase m from $\mathtt{m}_i$ to $\mathtt{m}_{i+1}$ to use a Taylor approximant of higher order or increment the scaling parameter s, to reduce the truncation error of approximation. Both options will increase the dominant part of the computational cost by one matrix multiplication. Although increasing s will increase the number of matrix squarings that will occur during the recovering phase of the algorithm, which is a potentially significant source of rounding errors for the algorithm, we still need to choose s such that the norm of $X = 2^{-s}A$ is sufficiently small in order for the Taylor approximation of $\cos X$ to be computed stably and accurately. On the other hand, when $\|2^{-s}A\| \gg 1$ both the actual error and the bound (4.4) can decrease extremely slowly as m increases, leading to the use of an approximant of degree much higher then necessary, which in turn results in loss of accuracy in floating-point arithmetic and unnecessary computation. It sometimes can be cheaper (and even more accurate) to perform a stronger scaling on A and use a lower order approximant.

Facing this flexibility in selecting the algorithmic parameters, algorithms aiming for a fixed precision environment usually choose to consider the approximants only up to a certain order, or set a scaling threshold $\eta > 0$ and keep increasing s until $\|2^{-s}A\| \leqslant \eta$ is satisfied. The multiprecision algorithm for the matrix exponential [15] determines both parameters at run time by monitoring the decay rate of the bound

Fragment 4.3: Recomputation of the diagonals.

```
1  function RecompDiags($A, C \in \mathbb{C}^{n\times n}$)
     ▷ Compute main diagonal and first superdiagonal of $C \approx \cos A$ for upper triangular
       or real upper quasi-triangular $A$.
2  for $i = 1$ to $n$ do
3      if $i = n-1$ or $i \leqslant n-2$ and $a_{i+2,i+1} = 0$ then
4          if $a_{i+1,i} = 0$ then
5              Recompute $c_{ii}$, $c_{i,i+1}$, $c_{i+1,i+1}$ using [5, eqs. (3.1), (3.3)].
6          else
7              Recompute $c_{ii}$, $c_{i,i+1}$, $c_{i+1,i}$, $c_{i+1,i+1}$ using [5, eq. (3.6)].
8          $i \leftarrow i + 1$
9      else
10         $c_{ii} \leftarrow \cos a_{ii}$
11 return $C$
```

on the truncation error of the approximant as m increases, and this heuristic proves to be effective. Let us denote the truncation error bound associated with the Taylor approximant of order $2\mathtt{m}_i$ of this algorithm by

$$\delta_i = \cosh\Big(\sqrt{\alpha^*_{\mathtt{m}_i}(4^{-s}B)}\Big) - t^{ch}_{2\mathtt{m}_i}\Big(\sqrt{\alpha^*_{\mathtt{m}_i}(4^{-s}B)}\Big). \tag{4.11}$$

In the algorithm we increment s when $\delta_{i-1} < \delta_i^k$, $k \in \mathbb{N}^+$, that is, when the bound on the absolute error does not decay at least at the order k as m increases from $\mathtt{m}_{i-1}$ to $\mathtt{m}_i$. We found in practice that $k = 3$ is the best choice for accuracy.

Once a combination of Taylor approximant of order m and scaling parameter s is found, the algorithm computes an approximation to $\cos(2^{-s}A)$ by evaluating the polynomial $p^c_m(4^{-s}B)$ using the modified Paterson–Stockmeyer method PSEvalCos given in Fragment 4.2, and finally recovers $\cos A$ by applying s steps of the double angle recurrence. If A is upper quasi-triangular, then in order to reduce the rounding errors introduced during the recovering phase and improve accuracy of the final result, the algorithm recomputes the diagonal and first superdiagonal of the intermediate matrices in the recovering stage from the elements of A, as discussed in [5], and this is accomplished by RecompDiags in Fragment 4.3. The algorithm can optionally make use of preprocessing (and postprocessing) techniques as discussed in [19], [27].

Algorithm 4.4: Multiprecision algorithm for the matrix cosine.

Given $A \in \mathbb{C}^{n\times n}$ this algorithm computes an approximation C to $\cos A$ in floating-point arithmetic with unit roundoff u using a scaling and recovering method based on Taylor approximants. The pseudocode of EvalBound is given in Fragment 4.1, that of PSEvalCos in Fragment 4.2, and that of RecompDiags in Fragment 4.3. The function isSchurForm returns true if A is upper triangular or real and upper quasi-triangular, and otherwise false.

```
1  $B \leftarrow A^2$
2  $\mathcal{B}_0 \leftarrow I$
3  $\mathcal{B}_1 \leftarrow B$
4  $\alpha_{\min} \leftarrow \infty$
5  $\delta_{\text{pre}} \leftarrow \infty$
6  $b \leftarrow [-\infty, -\infty, \dots]$
7  $s \leftarrow 0$
8  $i \leftarrow 1$
9  $[\delta_{\text{nxt}}, \phi] \leftarrow$ EvalBound$(B, \mathtt{m}_i, s)$
10 while $\delta_{\text{nxt}} > u\phi$ and $i < N$ do
11     if $\delta_{\text{pre}} < \delta^k_{\text{nxt}}$ then
12         $s \leftarrow s + 1$
13     else
14         $i \leftarrow i + 1$
15     $\delta_{\text{pre}} \leftarrow \delta_{\text{nxt}}$
16     $[\delta_{\text{nxt}}, \phi] \leftarrow$ EvalBound$(B, \mathtt{m}_i, s)$
17 $C \leftarrow$ PSEvalCos$(B, \mathtt{m}_i, s)$
18 if isSchurForm$(A)$ then
19     $C \leftarrow$ RecompDiags$(2^{-s}A, C)$
20 for $j \leftarrow 1$ to $s$ do
21     $C \leftarrow 2C^2 - I$
22     if isSchurForm$(A)$ then
23         $C \leftarrow$ RecompDiags$(2^{-s+j}A, C)$
24 return $C$
```

We now present the complete precision-independent scaling and recovering algorithm for the matrix cosine, which is given in Algorithm 4.4. In addition to the matrix $A \in \mathbb{C}^{n\times n}$, the algorithm takes the following input arguments.

- The arbitrary precision floating-point parameter $u > 0$ that specifies the unit roundoff of the working precision of the algorithm.
- The positive integer $m_{\max}$ determines the maximum order of the approximants p^c_m in (4.9) that the algorithm can consider. The algorithm will try the orders $m = \mathtt{m}_i$ ascendingly for $i = 1\colon N$ such that $\mathtt{m}_N \leqslant m_{\max} < \mathtt{m}_{N+1}$.

In the algorithm the variables $\mathcal{B}$, b, and $\alpha_{\min}$ are assumed to be available within all the code fragments (that is, their scope is global in the codes). We use the notation $[x, x, \dots]$ to denote a vector whose elements are all initialized to x and whose length is unimportant, and such a vector b is defined to store the approximated values of $\|B^d\|_1^{1/d}$, $d \in \mathbb{N}^+$, so the 1-norm of each power of B is estimated at most once.

Overall, Algorithm 4.4 requires about $2\sqrt{\mathtt{m}_i} + s$ matrix multiplications, or a total of $(4\sqrt{\mathtt{m}_i} + 2s)n^3$ flops in the highest order in precision u, where $2\sqrt{\mathtt{m}_i}$ multiplications are for forming the required powers of B and evaluating the polynomial $p_m^c(4^{-s}B)$, and s multiplications are for performing the final recovering phase.

4.4.1 Schur variant

If A is normal ($A^*A = AA^*$) and a multiprecision implementation of the QR algorithm [17, sect. 7.5] is available, then we should simply diagonalize A in precision u to obtain $A = QDQ^*$ with Q unitary and D diagonal and then compute $\cos A = Q\cos(D)Q^*$. More generally, for nonnormal A a (real) Schur decomposition can be computed before invoking our multiprecision algorithm. More specifically, we compute $A = QTQ^*$, where Q and T are, respectively, unitary and upper triangular if A has complex entries and orthogonal and upper quasi-triangular if A has real entries; then we compute $\cos A = Q\cos(T)Q^*$. This Schur variant of the algorithm requires $\big(28 + (4\sqrt{\mathtt{m}_i} + 2s)/3\big)n^3$ flops in precision u.

4.5 COMPUTING THE FRÉCHET DERIVATIVE

The Fréchet derivative of a matrix function f at $A \in \mathbb{C}^{n\times n}$ is a linear operator $L_f(A, \cdot)$ satisfying

$$f(A+E) - f(A) - L_f(A,E) = o(\|E\|)$$

for all $E \in \mathbb{C}^{n\times n}$. It appears in an expression for the condition number [20, sect. 3.1]:

$$\operatorname{cond}(f,A) := \lim_{\epsilon\to 0} \sup_{\|E\|\leqslant\epsilon\|A\|} \frac{\|f(A+E)-f(A)\|}{\epsilon\|f(A)\|} = \frac{\|L_f(A)\|\,\|A\|}{\|f(A)\|},$$

where

$$\|L_f(A)\| := \max_{G\neq 0} \frac{\|L_f(A,G)\|}{\|G\|}.$$

The condition number measures the first order sensitivity of $f(A)$ to small perturbations in A.

Recall that the truncation error for a Taylor approximant of degree $2m$ to $\cos X$ is

$$\cos X - t_{2m}^c(X) = \sum_{i=m+1}^{\infty} c_i X^{2i}, \quad c_i := \frac{(-1)^i}{(2i)!}. \tag{4.12}$$

Fréchet differentiating both sides of (4.12) at $X = 2^{-s}A$ in the direction $E_s := 2^{-s}E$ gives the truncation error for an approximation to the Fréchet derivative:

$$L_{\cos}(X,E_s) - L_{t_{2m}^c}(X,E_s) = \sum_{i=m+1}^{\infty} c_i L_{x^{2i}}(X,E_s). \tag{4.13}$$

From (4.13) we can approximate $L_{\cos}(X,E_s)$ by $L_{t_{2m}^c}(X,E_s)$ with a controllable truncation error. We will discuss in detail the computation of $L_{t_{2m}^c}(X,E_s)$, the Fréchet derivative of a power series, in the next subsection.

Now we derive the basic framework for computing $\cos A$ and $L_{\cos}(A,E)$ simultaneously given that the approximated values of $\cos X \equiv \cos(2^{-s}A)$ and $L_{\cos}(X,E_s) \equiv L_{\cos}(2^{-s}A, 2^{-s}E)$ are available. Fréchet differentiating the double angle formula $\cos(2A) = 2\cos^2 A - I$ and employing the chain rule, we have the relation

$$L_{\cos}(2A,2E) = L_{2x^2-1}\big(\cos A, L_{\cos}(A,E)\big).$$

Then using the linearity of the Fréchet derivative and the sum and product rules [20, Sec. 3.2], we obtain

$$L_{\cos}(2A, 2E) = 2\big(\cos A L_{\cos}(A, E) + L_{\cos}(A, E)\cos A\big).$$

Using this relation we can construct the following recurrence relation, which yields $C_0 := \cos A$ and $L_0 := L_{\cos}(A, E)$ simultaneously:

$$\begin{aligned} L_s &= L_{\cos}(2^{-s}A, 2^{-s}E), \\ C_s &= \cos(2^{-s}A), \\ &\left.\begin{aligned} L_{k-1} &= 2(C_k L_k + L_k C_k) \\ C_{k-1} &= 2C_k^2 - I \end{aligned}\right\} k = s : -1 : 1. \end{aligned}$$

4.5.1 Error analysis and evaluation scheme

We can derive an error bound for the approximation $L_{\cos}(X, E_s) \approx L_{t_{2m}^c}(X, E_s)$. Taking norms on both sides of (4.13) gives

$$\begin{aligned} \|L_{\cos}(X, E_s) - L_{t_{2m}^c}(X, E_s)\| &\leqslant \sum_{i=m+1}^{\infty} 2i|c_i|\|E_s\|\|X\|^{2i-1} = \|E_s\| \sum_{i=m}^{\infty} \frac{\|X\|^{2i+1}}{(2i+1)!} \\ &= \|E_s\| \Big(\sinh(\|X\|) - \sum_{i=0}^{m-1} \frac{\|X\|^{2i+1}}{(2i+1)!}\Big), \end{aligned} \tag{4.14}$$

where we have used the result $\|L_{x^i}(X, E_s)\| \leqslant i\|E_s\|\|X\|^{i-1}$ [3, Thm. 3.2]. One advantage of this forward error bound for the Fréchet derivative of t_{2m}^c is that it can be used in a multiprecision environment. Note that the error bound is based on $\|X\|$, whereas the error bound for t_{2m}^c itself is based on $\alpha_m^*(X^2)$. Since the bound based on $\|X\|$ can be arbitrarily weak (see the discussion in section 4.3) we will base our algorithm for computing $\cos A$ and $L_{\cos}(A, E)$ on the α_m-based bound (4.8). We will test experimentally whether this produces an accurate Fréchet derivative. A similar situation holds in the works [4] for the matrix logarithm and [22] for the matrix fractional powers, where the algorithms (designed for double precision) are based

Fragment 4.5: Modified Paterson–Stockmeyer scheme for the matrix cosine and its Fréchet derivative.

```
1  function PSEvalCosLc($A, B, E \in \mathbb{C}^{n\times n}$, $m, s \in \mathbb{N}$)
     ▷ Compute simultaneously $C \approx \cos(2^{-s}A)$ and $L \approx L_{\cos}(2^{-s}A, 2^{-s}E)$.
2  for $i = 0$ to $m$ do
3      $c_i \leftarrow (-1)^i/(2i)!$
4  $\nu \leftarrow \lfloor\sqrt{m}\rfloor$
5  $\mu \leftarrow \lfloor m/\nu\rfloor$
6  for $i \leftarrow \text{length}(\mathcal{B}) + 1$ to $\nu$ do
7      $\mathcal{B}_i \leftarrow \mathcal{B}_{i-1}B$
8  for $i \leftarrow 0$ to $\mu - 1$ do
9      $\mathbb{Z}_i \leftarrow \sum_{j=0}^{\nu-1} c_{\nu i+j}4^{-sj}\mathcal{B}_j$
10 $\mathbb{Z}_\mu \leftarrow \sum_{j=0}^{m-\mu\nu} c_{\nu i+j}4^{-sj}\mathcal{B}_j$
11 $\mathcal{M}_1 \leftarrow 4^{-s}(AE + EA)$
12 for $j \leftarrow 2$ to $\nu$ do
13     $\mathcal{M}_j \leftarrow 4^{-s}\mathcal{M}_{j-1}B + 4^{-s(j-1)}\mathcal{B}_{j-1}\mathcal{M}_1$
14 $\mathbb{N}_1 \leftarrow \mathcal{M}_\nu$
15 $P \leftarrow \mathbb{N}_1$
16 for $i \leftarrow 2$ to $\mu$ do
17     $P \leftarrow 4^{-s\nu}\mathcal{B}_\nu P$
18     $\mathbb{N}_i \leftarrow 4^{-s\nu}\mathbb{N}_{i-1}\mathcal{B}_\nu + P$
19 $C \leftarrow \mathbb{Z}_\mu$
20 $L \leftarrow \sum_{j=1}^{m-\mu\nu} c_{\nu i+j}\mathcal{M}_j$
21 for $i \leftarrow \mu - 1$ down to $0$ do
22     $C \leftarrow 4^{-s\nu}C\mathcal{B}_\nu + \mathbb{Z}_i$
23     $L \leftarrow 4^{-s\nu}L\mathcal{B}_\nu + \sum_{j=1}^{\nu-1} c_{\nu i+j}\mathcal{M}_j$
24 $L \leftarrow L + \sum_{i=1}^{\mu} \mathbb{Z}_i\mathbb{N}_i$
25 return $C$, $L$
```

on backward α_m-based error bounds for the functions themselves. Of course, if the Fréchet derivatives are being used for condition number estimation then an accurate derivative is not required.

Now we derive an evaluation scheme for computing $\cos X$ and $L_{\cos}(X, E_s)$ in a way that reuses matrix operations from the computation of $\cos X$ in the computation of $L_{\cos}(X, E_s)$. Fréchet differentiating both sides of $t^c_{2m}(X) = p^c_m(Y)$ from (4.9), where $Y = X^2$, at X in direction E_s and using the chain rule, we have

$$L_{t^c_{2m}}(X, E_s) = L_{p^c_m}\big(X^2, L_{x^2}(X, E_s)\big) = L_{p^c_m}(Y, XE_s + E_sX).$$

Recall that $p_m^c(Y)$ is evaluated by the Paterson–Stockmeyer method. To be more specific, we rewrite the polynomial as

$$p_m^c(Y) = \sum_{i=0}^{\mu} Z_i (Y^\nu)^i, \quad \mu = \lfloor m/\nu \rfloor, \tag{4.15}$$

where

$$Z_i = \begin{cases} \sum_{j=0}^{\nu-1} c_{\nu i+j} Y^j, & i = 0, \dots, \mu-1, \\ \sum_{j=0}^{m-\mu\nu} c_{\nu i+j} Y^j, & i = \mu. \end{cases}$$

Note that the powers of $Y = 4^{-s}B$ up to the νth are available by ν matrix scalings given that we have formed those powers of B. Fréchet differentiating both sides of (4.15) at Y in direction $\widetilde{E}_s := XE_s + E_sX$ and employing the product rule, we have

$$L_{p_m^c}(Y, \widetilde{E}_s) = \sum_{i=0}^{\mu} L_{z_i}(Y, \widetilde{E}_s)(Y^\nu)^i + \sum_{i=1}^{\mu} Z_i M_{\nu i},$$

where

$$L_{z_i}(Y, \widetilde{E}_s) = \begin{cases} \sum_{j=1}^{\nu-1} c_{\nu i+j} M_j, & i = 0, \dots, \mu-1, \\ \sum_{j=1}^{m-\mu\nu} c_{\nu i+j} M_j, & i = \mu, \end{cases}$$

and $M_j := L_{y^i}(Y, \widetilde{E}_s)$ satisfies the recurrence relation

$$M_j = M_{\ell_1} Y^{\ell_2} + Y^{\ell_1} M_{\ell_2}, \quad M_1 = \widetilde{E}_s, \tag{4.16}$$

where $j = \ell_1 + \ell_2$ with positive integers ℓ_1 and ℓ_2 [3, Thm. 3.2]. Hence, it is efficient to compute explicitly and store Z_i in computing $p_m^c(Y) \approx \cos(2^{-s}A)$ as we can reuse these coefficient matrices for computing $L_{p_m^c}(Y, \widetilde{E}_s)$. In addition, M_j for $j = 1, 2, \dots, \nu-1$ and $j = \nu, 2\nu, \dots, \mu\nu$ are needed. Using (4.16) we can compute

$$\begin{aligned} M_j &= M_{j-1} Y + Y^{j-1} M_1, & j &= 2:\nu, \\ M_{i\nu} &= M_{(i-1)\nu} Y^\nu + Y^{(i-1)\nu} M_\nu, & i &= 2:\mu, \end{aligned} \tag{4.17}$$

where all the required powers of Y are available if both the right-hand sides of (4.15) and (4.17) are evaluated via explicit powers. However, we found in practice that

Algorithm 4.6: Multiprecision algorithm for the matrix cosine and its Fréchet derivative.

Given $A \in \mathbb{C}^{n\times n}$ and $E \in \mathbb{C}^{n\times n}$ this algorithm computes simultaneously $C \approx \cos A$ and $L \approx L_{\cos}(A,E)$ in floating-point arithmetic with unit roundoff u using a scaling and recovering method based on Taylor approximants. The pseudocode of PSEvalCosLc is given in Fragment 4.5, and that of RecompDiags in Fragment 4.3.

```
1  Execute lines 1–16 in Algorithm 4.4.
2  $[C, L] \leftarrow$ PSEvalCosLc$(A, B, E, \mathtt{m}_i, s)$
3  if isSchurForm$(A)$ then
4      $C \leftarrow$ RecompDiags$(2^{-s}A, C)$
5  for $j \leftarrow 1$ to $s$ do
6      $L \leftarrow 2(CL + LC)$
7      $C \leftarrow 2C^2 - I$
8      if isSchurForm$(A)$ then
9          $C \leftarrow$ RecompDiags$(2^{-s+j}A, C)$
10 return $C$, $L$
```

evaluating the right-hand side of (4.15) via explicit powers produces a less accurate approximation for $\cos(2^{-s}A)$ than using Horner's method. We hence use Horner's method and form the extra powers $Y^{i\nu}$, $i = 2, \dots, \mu - 1$ implicitly when required in (4.17).

We summarize in Fragment 4.5 our scheme for computing simultaneously $\cos(2^{-s}A)$ and $L_{\cos}(2^{-s}A, 2^{-s}E)$ where we have introduced the arrays $\mathbb{Z}_i = Z_i$, $i = 0, \dots, \mu$, $\mathcal{M}_j = M_j$, $j = 1, \dots, \nu$, and $\mathbb{N}_i = M_{i\nu}$, $i = 1, \dots, \mu$.

Exploiting Fragment 4.5, we obtain Algorithm 4.6, the overall algorithm for computing $\cos A$ and $L_{\cos}(A, E)$ simultaneously. The total cost of Algorithm 4.6 in precision u in the highest order is the $(4\sqrt{\mathtt{m}_i} + 2s)n^3$ flops cost of Algorithm 4.4 plus the extra cost for computing $L_{\cos}(A, E)$, which consists of about $6\sqrt{\mathtt{m}_i}$ matrix multiplications for computing the required coefficient matrices M_j and forming the approximated $L_{\cos}(2^{-s}A, 2^{-s}E)$, and $2s$ matrix multiplications in the recovering recurrence for $L_{\cos}(A, E)$, namely an extra cost of $(12\sqrt{\mathtt{m}_i} + 4s)n^3$ flops. This algorithm also requires about $3\sqrt{\mathtt{m}_i}n^2$ additional memory locations for the storage of the Z_i and M_i for the computation of $L_{\cos}(A, E)$.

If a Schur decomposition $T = Q^*AQ$ is computed before invoking Algorithm 4.6, then for the Fréchet derivative we need apply to the direction E the same transformation and undo the transformation at the end [20, Prob. 3.2], arriving at $L_{\cos}(A, E) = QL_{\cos}(T, Q^*EQ)Q^*$. If a real Schur decomposition is computed and E is full, the Schur variant of Algorithm 4.6 requires an extra cost of $\bigl(8 + (12\sqrt{\mathtt{m}_i} + 4s)/2\bigr)n^3$ flops in precision u for computing the Fréchet derivative. For normal A we can simply employ an explicit formula obtained from the Daleckiĭ–Kreĭn theorem [20, Thm. 3.11] for computing the Fréchet derivative.

In some situations, such as in condition estimation, when several Fréchet derivatives $L_{\cos}(A, E)$ are needed at a fixed A and different direction we need only compute the parameters s and m once since they depend only on A.

4.5.2 Extension to the sine and its Fréchet derivative

It is straightforward to bound the truncation error of a Taylor approximant to the matrix sine function in a similar way to (4.4), by employing the hyperbolic sine. We have

$$\begin{aligned}\|\sin A - t^{s}_{2m+1}(A)\| &\leqslant \|A\| \left\| \sum_{i=m+1}^{\infty} \frac{(-1)^i}{(2i+1)!} B^i \right\| \leqslant \frac{\|A\|}{\sqrt{\alpha_m(B)}} \sum_{i=m+1}^{\infty} \frac{\bigl(\sqrt{\alpha_m(B)}\bigr)^{2i+1}}{(2i+1)!} \\ &= \frac{\|A\|}{\sqrt{\alpha_m(B)}} \Bigl(\sinh\bigl(\sqrt{\alpha_m(B)}\bigr) - t^{sh}_{2m+1}\bigl(\sqrt{\alpha_m(B)}\bigr)\Bigr), \qquad (4.18)\end{aligned}$$

where

$$t^{s}_{2m+1}(A) := \sum_{i=0}^{m} \frac{(-1)^i}{(2i+1)!} A^{2i+1}, \qquad t^{sh}_{2m+1}(A) := \sum_{i=0}^{m} \frac{1}{(2i+1)!} A^{2i+1} \qquad (4.19)$$

are the Taylor approximants of order $2m + 1$ to $\sin A$ and $\sinh A$, respectively. Based on the t^{s}_{2m+1} of (4.19) together with the triple angle recurrence $\sin(3A) = 3\sin A - 4\sin^3 A$ we can design a multiprecision algorithm for the matrix sine similarly to that in section 4.4. Moreover, an algorithm for computing the matrix sine and cosine at the same time can be easily developed exploiting the idea in [5], where computational savings are possible by reusing the powers of $B = A^2$ in the matrix array $\mathcal{B}$.

On the other hand, we can evaluate the matrix sine and its Fréchet derivative simultaneously without computing $\cos A$. Fréchet differentiating the triple angle formula $\sin(3A) = \sin A(3I - 4\sin^2 A)$ and employing the product rule, we arrive at

$$\begin{aligned} L_{\sin}(3A, 3E) = L_{\sin}(A, E)\big(3I - 4\sin^2 A\big) \\ - 4\sin A\big(\sin A\, L_{\sin}(A, E) + L_{\sin}(A, E)\sin A\big). \end{aligned} \tag{4.20}$$

If we scale A by 3^s for some $s \in \mathbb{N}$, using (4.20) we obtain the following relation which produces $S_0 := \sin A$ and $\widetilde{L}_0 := L_{\sin}(A, E)$ simultaneously:

$$\widetilde{L}_s = L_{\sin}(3^{-s}A, 3^{-s}E), \quad S_s = \sin(3^{-s}A),$$
$$\left.\begin{aligned} \widetilde{L}_{k-1} &= \widetilde{L}_k(3I - 4S_k^2) - 4S_k(S_k\widetilde{L}_k + \widetilde{L}_kS_k) \\ S_{k-1} &= S_k(3I - 4S_k^2) \end{aligned}\right\} k = s : -1 : 1,$$

where S_k can be evaluated by a Taylor approximant with truncation error bounded above by (4.18), and an approximated $\widetilde{L}_k$ is obtainable by Fréchet differentiating S_k in the direction $3^{-s}E$.

Ultimately, it is possible to construct an algorithm to compute efficiently the matrix cosine and sine functions and their Fréchet derivatives all together. Fréchet differentiating both sides of the double angle formulae

$$\cos(2A) = I - 2\sin^2 A, \qquad \sin(2A) = 2\sin A\cos A$$

gives

$$\begin{aligned} L_{\cos}(2A, 2E) &= -2\big(\sin A\, L_{\sin}(A, E) + L_{\sin}(A, E)\sin A\big), \\ L_{\sin}(2A, 2E) &= 2\big(\sin A\, L_{\cos}(A, E) + L_{\sin}(A, E)\cos A\big), \end{aligned}$$

from which we can obtain the following recurrence for computing $C_0 = \cos A$, $S_0 = \sin A$, $L_0 = L_{\cos}(A, E)$, and $\widetilde{L}_0 = L_{\sin}(A, E)$ all together.

$$
\begin{aligned}
&L_s = L_{\sin}(2^{-s}A, 2^{-s}E), \quad \widetilde{L}_s = L_{\sin}(2^{-s}A, 2^{-s}E),\\
&C_s = \cos(2^{-s}A), \quad S_s = \sin(2^{-s}A),
\end{aligned}
$$

$$
\left.\begin{aligned}
L_{k-1} &= -2(S_k\widetilde{L}_k + \widetilde{L}_k S_k)\\
\widetilde{L}_{k-1} &= 2(S_k L_k + \widetilde{L}_k C_k)\\
S_{k-1} &= 2S_k C_k\\
C_{k-1} &= I - 2S_k^2
\end{aligned}\right\} k = s : -1 : 1.
$$

4.6 NUMERICAL EXPERIMENTS

All our experiments were performed using the 64-bit version of MATLAB 2021a on a laptop equipped with an Intel i7-6700HQ processor running at 2.60GHz and with 16GB of RAM. The code uses the Advanpix Multiprecision Computing Toolbox (version 4.8.3.14463) [32], which allows the user to specify the number of *decimal* digits d of working precision by using the command `mp.Digits(d)`.

The test matrices, whose size ranges between 4 and 41, are nonnormal and are selected from Anymatrix [25], [24] and the literature of matrix functions [5], [15], [27]; those from the matrix function literature are collected in an Anymatrix group that is available on GitHub.[1] Normal matrices are excluded since they can be easily handled by diagonalization, as we have discussed in the previous sections. Most of test matrices have only real elements and are set to be of size 16×16. To examine the algorithms for complex matrices we also test the above matrices multiplied by the imaginary unit i. In total 198 matrices are used in the experiments, and we denote by $\mathcal{F}$ the set containing these matrices. The MATLAB code for our algorithms and experiments is available on GitHub.[2]

1 https://github.com/Xiaobo-Liu/matrices-mp-cosm
2 https://github.com/Xiaobo-Liu/mp-cosm

We compare the following codes for computing the cosine. The first three codes are for double precision only, and are used to test whether our algorithm is competitive in double precision.

- `cosm`, the algorithm by Al-Mohy, Higham, and Relton [5, Alg. 4.1], which is intended for double precision only.
- `cosm_tay`, the algorithm by Sastre et al. [36], which uses the scaling and recovering method based on truncated Taylor series, and is intended for double precision only.
- `cosm_pol`, the algorithm by Sastre et al. [37], which uses Taylor polynomial approximations of fixed degree with precomputed coefficients, and is intended for double precision only.

The next four codes are for arbitrary precision.

- `cosm_adv`, the (overloaded) `cosm` function provided by the Advanpix Multiprecision Computing Toolbox [32].[3]
- `cosm_exp`, the algorithm [20, Alg. 12.7] that computes the cosine via the identity involving the exponential

$$\cos A = \begin{cases} \operatorname{Re}(e^{iA}), & \text{if } A \text{ is real,} \\ \frac{1}{2}(e^{iA} + e^{-iA}), & \text{if } A \text{ is complex,} \end{cases} \tag{4.21}$$

  where the exponential is computed by `expm` [2] and the multiprecision algorithm for the matrix exponential [15], respectively, in double precision and other precisions.
- `cosm_mp`, our implementation of Algorithm 4.4 with $m_{\max} = 500$.
- `cosm_mp_s`, our implementation of the Schur variant of Algorithm 4.4 with $m_{\max} = 500$ (the real Schur decomposition is used where possible).

3 This function in fact uses a multiprecision Schur–Parlett algorithm in computation, according to a private communication with Pavel Holoborodko, the author of the toolbox.

We found in practice that the preprocessing techniques in general made little difference to accuracy of our algorithm (this is also found in [19], for example). Therefore, we did not perform preprocessing in the tests. We also compared `cosm_mp` with its counterpart based on an absolute error bound and found that the former is faster and more accurate in practice. In our implementation of Algorithm 4.4 we set $k = 3$, which controls the switch between increasing m and s as the truncation error decays, and we chose the lower precision in Fragment 4.1 to be double precision.

We assess the quality of a computed solution $\widetilde{X}$ by an algorithm running with d digits of precision in terms of the 1-norm relative forward error $\|X - \widetilde{X}\|_1/\|X\|_1$, where the reference solution X is computed in $2d$ digits of precision using `cosm_mp` with $m_{\max} = 2500$. We gauge the forward stability of the algorithms by comparing the forward error with $\kappa_{\cos}(A)u$, where $\kappa_{\cos}(A)$ is the 1-norm condition number [20, Chap. 3] of the matrix cosine of A. We estimate it in double precision by applying the `funm_condest1` function provided by the Matrix Function Toolbox [21] to `cosm`.

To improve the plots of forward error, we map any errors outside the displayed range onto the nearest edge (top or bottom) of the plot. We also present the results in the form of performance profiles [13], and use the technique of Dingle and Higham [12] to rescale errors smaller than u.

4.6.1 Accuracy of the computed cosine in double precision

Our first experiments compare the accuracy of `cosm_mp` and `cosm_mp_s` with `cosm`, `cosm_exp`, `cosm_tay`, `cosm_pol`, and `cosm_adv` in IEEE double precision, with `cosm_adv` running with 16 decimal digits of precision simulated by Advanpix [32].

Figure 4.1 presents the comparison in accuracy between the algorithms on the test matrices, sorted by decreasing condition number. In the performance profiles, the y-coordinates of a given method represents the frequency of matrices for which its relative error is within a factor θ of the error of the algorithm that produces the most accurate result. We observe that our implementation of `cosm_mp` in double precision is competitive in accuracy with the most accurate algorithms that are optimized for

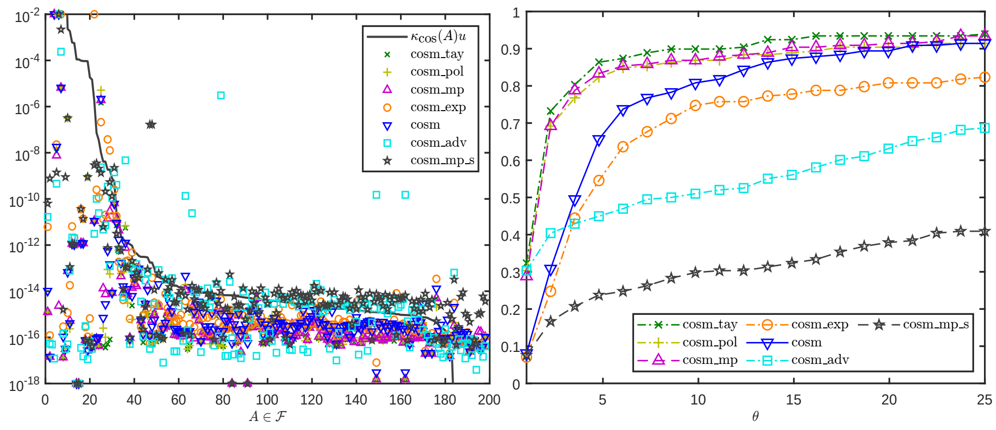


Figure 4.1: Left: forward errors of the algorithms on the matrices in $\mathcal{F}$ in double precision, where the solid line is $\kappa_{\cos}(A)u$. Right: corresponding performance profiles.

double precision. We also note that among the algorithms `cosm_adv` is overall the least accurate and can be unstable, as it sometimes provides a forward error far above $\kappa_{\cos}(A)u$.

4.6.2 Accuracy in higher precision

Now we examine the accuracy of our algorithms in higher precision. We compare the relative forward errors of `cosm_adv`, `cosm_exp`, `cosm_mp`, and `cosm_mp_s` running at 256 and 1024 decimal digits of precision on the test matrices, and report the same data in the form of performance profiles.

As reported in Figure 4.2, `cosm_mp` delivers superior accuracy to its counterparts and gives the best accuracy in more than 60 percent of the cases. The exponential-based algorithm `cosm_exp` is only slightly less accurate than `cosm_mp`. The Schur-based algorithm `cosm_mp_s` is distinctively less accurate than its Schur-free counterpart. We also note that `cosm_adv` achieves the worst overall accuracy in the experiments and shows signs of forward instability, especially when the number of decimal digits is increased from 256 to 1024, as it gives errors much larger than $\kappa_{\cos}(A)u$ in many cases.

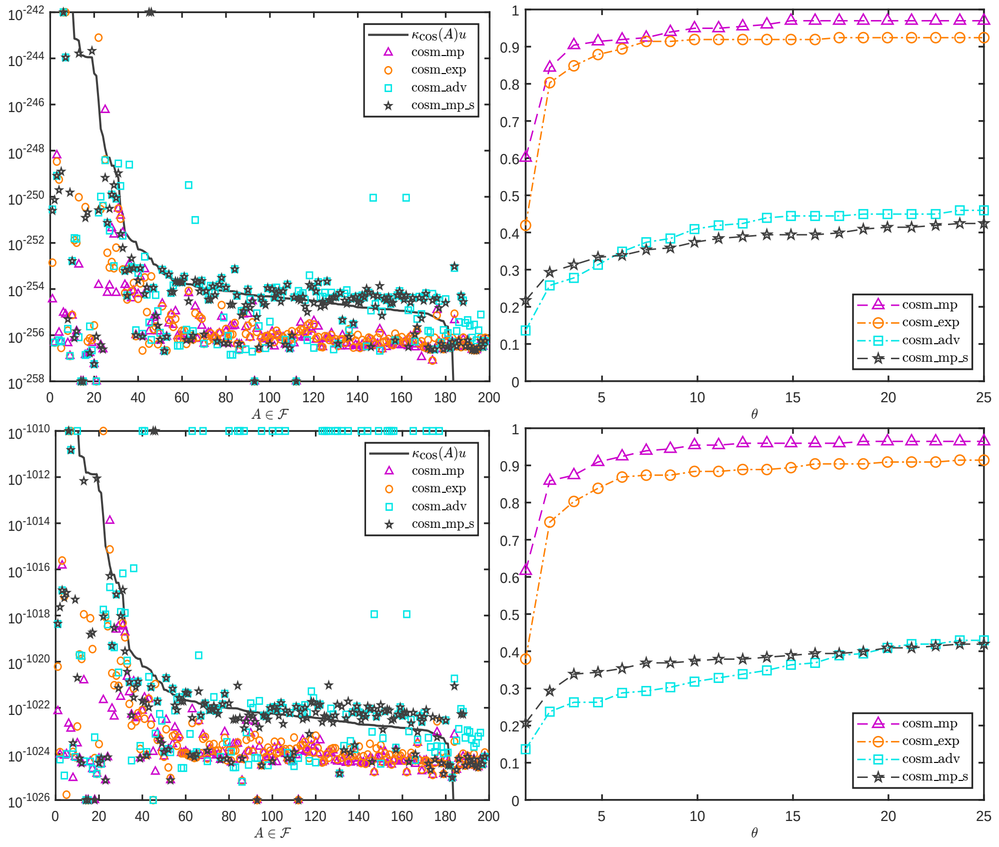


Figure 4.2: Left: forward errors of the algorithms on the matrices in $\mathcal{F}$ in d digits of precision, where the solid line is $\kappa_{\cos}(A)u$. Right: corresponding performance profiles. Top: $d = 256$. Bottom: $d = 1024$.

4.6.3 Speed comparison for computing the cosine

We also compared the execution times of our implementations of `cosm_mp` and `cosm_mp_s` with the other algorithms in double and higher precisions. For this purpose it is sensible to test the algorithms on matrices of different sizes, so we take from $\mathcal{F}$ the matrices whose size is variable in the tests. We denote the new set of 104 matrices by $\mathcal{V}$.

Figure 4.3 shows that, in double precision, `cosm_tay`, `cosm_pol`, `cosm_exp`, and `cosm` are the fastest algorithms and have close performance in computation time. These double-precision-oriented algorithms employ a rational approximant of degree chosen from a fixed set based on error bounds with precomputed coefficients and are highly optimized in selecting the degree and scaling parameter, so in general `cosm_mp` and `cosm_mp_s` are not expected to be as efficient. However, from the performance

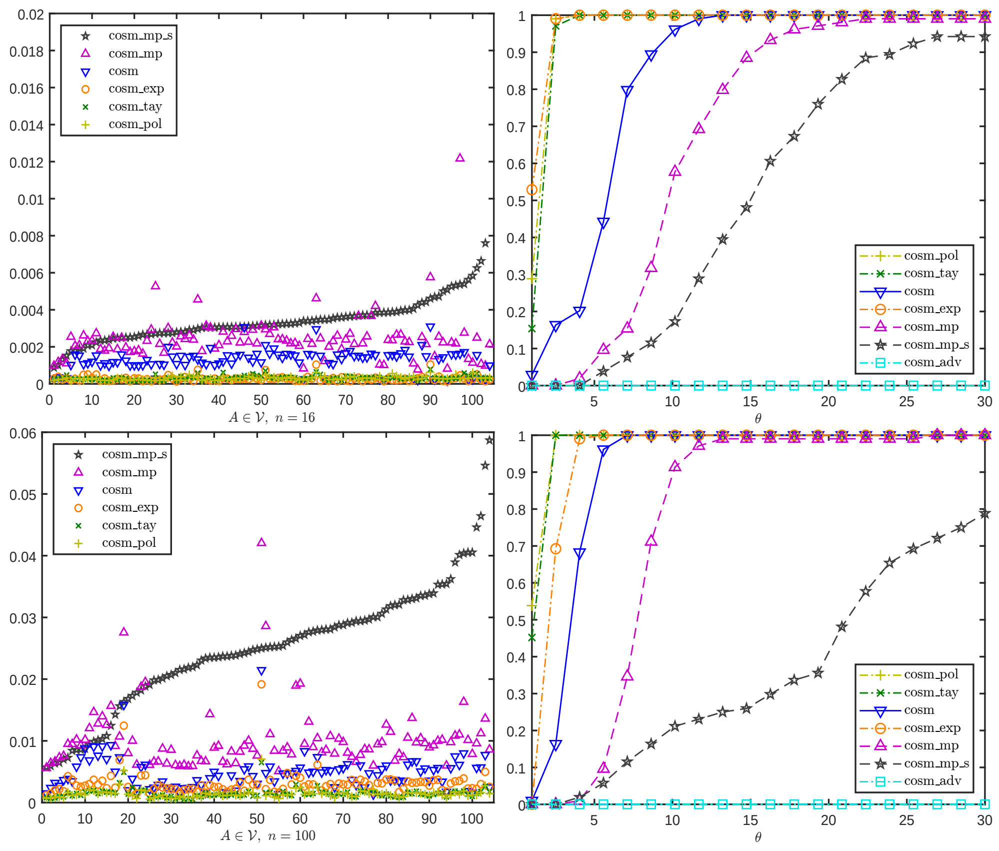


Figure 4.3: Execution times (in seconds) and the corresponding performance profile of the algorithms for matrices of different size in $\mathcal{V}$ in double precision. The execution times for `cosm_adv` are significantly larger and are not plotted.

profiles we observe that `cosm_mp` has relatively better performance as n increases from 16 to 100. This is because the $O(n^2)$ flops required by `cosm_mp` in evaluating the error bound, which is extra compared with the above double-precision-oriented algorithms, are expensive for small matrices but become negligible for large n. The Schur-based algorithm `cosm_mp_s` becomes relatively slower as n grows. We also note that `cosm_adv` is appreciably slower than the rest of the algorithms in both cases.

Then we compare the execution times of `cosm_adv`, `cosm_exp`, `cosm_mp`, and `cosm_mp_s` in precisions higher than double. Figure 4.4 reports the execution times and corresponding performance profiles of these algorithms in 256 digits of precision, where matrices of size $n = 16$ and $n = 100$ are used. It is observed that `cosm_mp` is substantially faster than the other algorithms, being the fastest algorithm on about 80 percent of the matrices in both sets. The Schur-based algorithm `cosm_mp_s` is in general faster than `cosm_exp` for $n = 16$, but its performance deteriorates for $n = 100$. `cosm_adv`

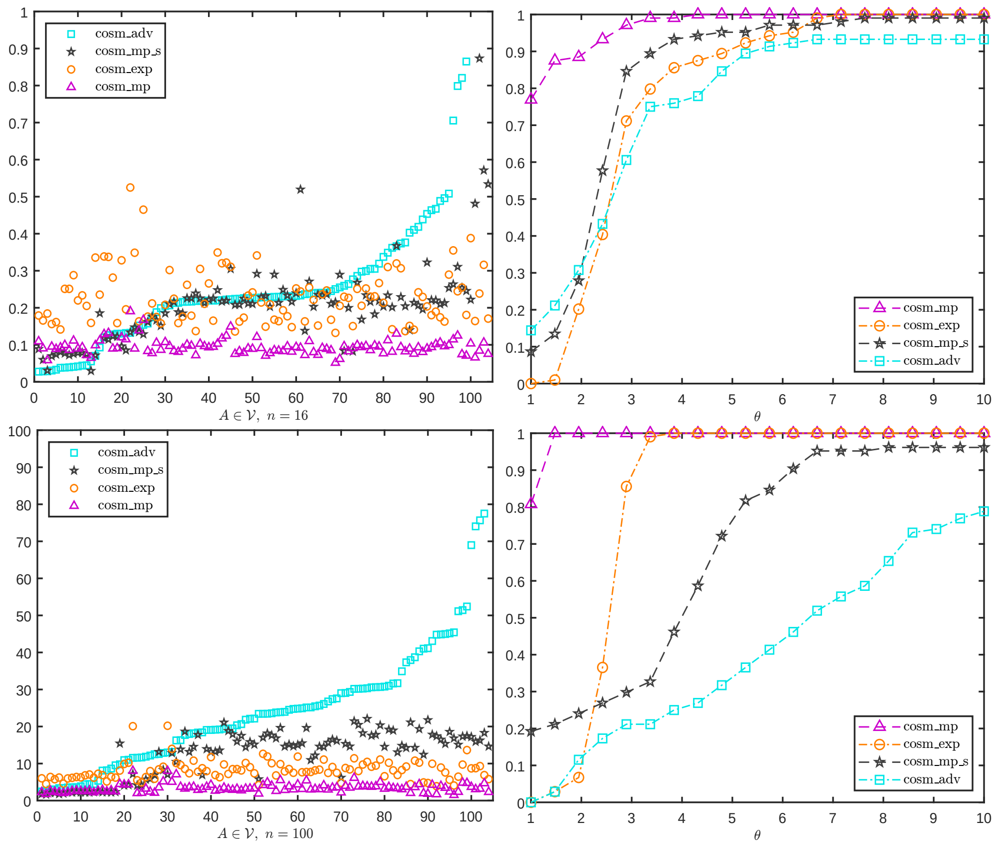


Figure 4.4: Execution times (in seconds) and corresponding performance profiles of the algorithms in 256 digits of precision on matrices of different sizes.

is the least efficient algorithm and its behavior is unsteady as it can be much slower than other algorithms on certain matrices. We repeated the above experiments in a working precision of 1024 digits, finding similar behavior of the algorithms.

4.6.4 Accuracy of the computed Fréchet derivative

In this section we examine the accuracy of Algorithm 4.6 for computing the Fréchet derivative in multiprecision arithmetic on the 198 matrices in set $\mathcal{F}$. For each A, we used a different E, generated to have pseudorandom elements drawn from the standard normal distribution and normalised such that $\|E\|_1 = 1$. We evaluate in the 1-norm the relative forward error of the computed Fréchet derivative. To obtain the reference solution $L_{\cos}(A, E)$ we apply Algorithm 4.4 with $m_{\max} = 2500$ in twice the

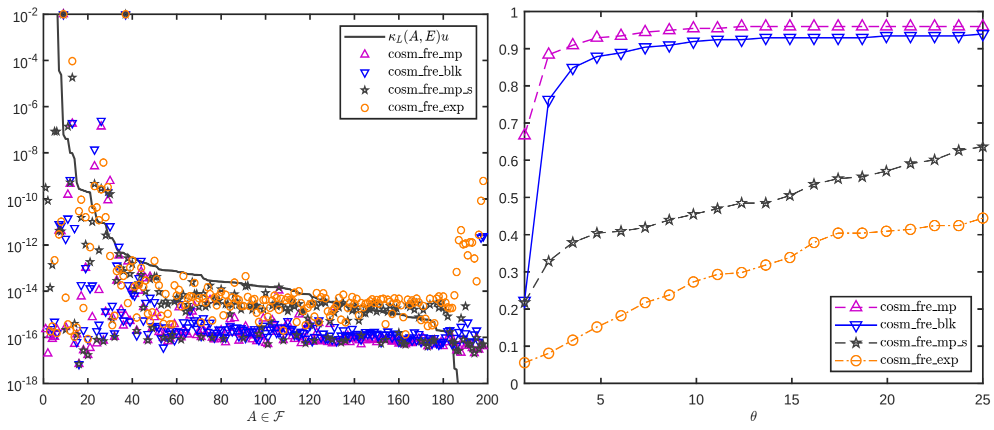


Figure 4.5: Left: forward errors in $L_{\cos}(A, E)$ on the matrices in $\mathcal{F}$ in double precision, where the solid line is $\kappa_L(A, E)u$. Right: corresponding performance profiles.

working precision to the $2n \times 2n$ matrix $\left[\begin{smallmatrix} A & E \\ 0 & A \end{smallmatrix}\right]$ and exploit the property, for arbitrary f [20, eq. (3.16)],

$$f\left(\begin{bmatrix} A & E \\ 0 & A \end{bmatrix}\right) = \begin{bmatrix} f(A) & L_f(A,E) \\ 0 & f(A) \end{bmatrix}. \tag{4.22}$$

We tested the following four schemes for computing the Fréchet derivative:

- `cosm_fre_blk`, Algorithm 4.4 with $m_{\max} = 500$ applied to the block 2×2 matrix in (4.22).
- `cosm_fre_mp`, our implementation of Algorithm 4.6 with $m_{\max} = 500$.
- `cosm_fre_mp_s`, our implementation of the Schur variant of Algorithm 4.6 with $m_{\max} = 500$.
- `cosm_fre_exp`, which computes $L_{\cos}(A, E) = \frac{i}{2}(L_{\exp}(iA, E) - L_{\exp}(-iA, E))$, which is obtained by applying the chain rule to the complex case of (4.21), by invoking the algorithm [3] for computing the Fréchet derivative of the matrix exponential, and is intended for double precision only.

As in the previous experiments in the implementation of Algorithms 4.4 and 4.6 we set $k = 3$, which controls the switch between increasing m and s as the truncation error decays. We also measure the forward stability of these algorithms by compar-

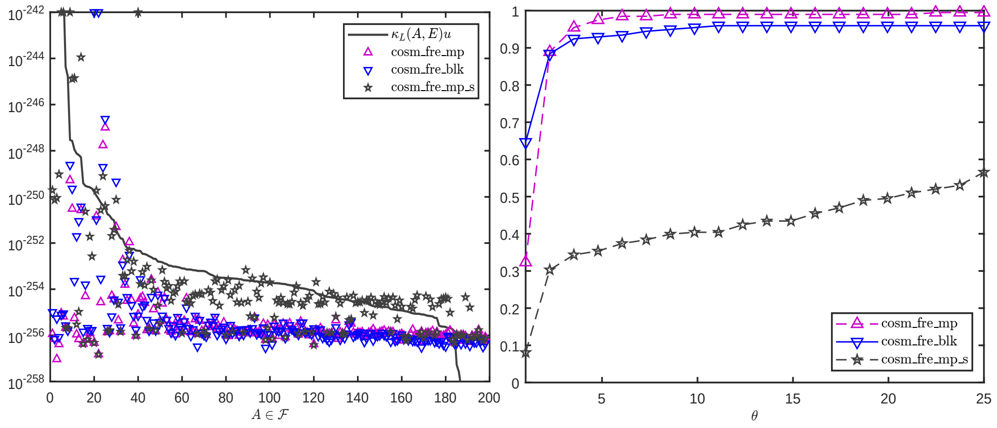


Figure 4.6: Left: forward errors in $L_{\cos}(A,E)$ on the matrices in $\mathcal{F}$ in 256 digits of precision, where the solid line is $\kappa_L(A,E)u$. Right: corresponding performance profiles.

ing the error with $\operatorname{cond}_L(A,E)u$, where $\operatorname{cond}_L(A,E)$ is the condition number of the Fréchet derivative, defined as

$$\operatorname{cond}_L(A,E) = \lim_{\epsilon\to 0} \sup_{\substack{\|\Delta A\|\leqslant\epsilon\|A\| \\ \|\Delta E\|\leqslant\epsilon\|E\|}} \frac{\|L_{\cos}(A+\Delta A, E+\Delta E) - L_{\cos}(A,E)\|}{\epsilon\|L_{\cos}(A,E)\|}.$$

We estimate $\operatorname{cond}_L(A,E)$ using an algorithm of Higham and Relton [26].

We observe from Figure 4.5 that `cosm_fre_mp` and `cosm_fre_blk` are competitive in terms of accuracy. This also reflects the robustness of Algorithm 4.4 for computing the matrix cosine. However, `cosm_fre_blk` has eight times the cost and four times the storage requirement of `cosm_fre_mp`, and its performance may depend on the scaling of the perturbation E, which is undesirable [22, sect. 4.3]. All the algorithms except `cosm_fre_exp` behave in a forward stable manner in most of cases. The exponential-based algorithm `cosm_fre_exp` is, in general, the least accurate and can be unstable on some very well-conditioned problems.

Finally, we examine the accuracy of the algorithms in precisions higher than double. Figure 4.6 shows a similar trend to that in the double precision. The Schur-free algorithms `cosm_fre_mp` and `cosm_fre_blk` are most accurate and have close performance, and all three algorithms are reasonably forward stable. We repeated the above experiments in a working precision of 1024 digits, finding similar behavior of the algorithms.

4.7 CONCLUDING REMARKS

Existing algorithms for computing the matrix cosine are all designed for double precision arithmetic and typically require certain precomputed constants that are specific to double precision arithmetic, so they do not conveniently extend to an arbitrary precision environment. In this work we have developed multiprecision algorithms that take the unit roundoff u and matrices A and E as input and compute $\cos A$ and the Fréchet derivative $L_{\cos}(A, E)$. The algorithms employ a forward error bound on the Taylor approximant to $\cos A$ that combines the hyperbolic cosine function with the quantity $\sqrt{\alpha_m(A^2)}$. We have also derived a framework for computing the Fréchet derivative, constructed an efficient evaluation scheme for computing the cosine and its Fréchet derivative simultaneously in arbitrary precision, and shown how this scheme can be extended to compute the matrix sine, cosine, and their Fréchet derivatives all together.

Experiments show that our new algorithms behave in a forward stable manner in floating-point arithmetic. The transformation-free version of the new algorithm for computing the cosine is competitive in accuracy with the state-of-the-art algorithms in double precision and is the fastest and most accurate among all candidates in working precisions higher than double. The fact that the Fréchet derivative computation in Algorithm 4.6 is based on an error bound that is valid only for the $\cos A$ computation does not appear to affect the accuracy of the computed Fréchet derivative. The new algorithms have been shown to have excellent accuracy on various test matrices as well as their variants multiplied by the imaginary unit i, so the algorithms are also good candidates for computing the matrix hyperbolic cosine function and its Fréchet derivative from the identity $\cosh A = \cos(\mathrm{i}A)$.

The analysis and techniques here can be adapted for evaluating other matrix trigonometric and hyperbolic functions in arbitrary precision arithmetic, such as those treated in [1] and the wave-kernel functions investigated in [33] and their Fréchet derivatives. Another possible future direction is to extend our algorithms to compute the action of these functions on a matrix in arbitrary precision as it is actually the matrix–vector products that are required in the solutions of wave equations.

REFERENCES

[1] A. H. Al-Mohy. "A truncated Taylor series algorithm for computing the action of trigonometric and hyperbolic matrix functions." *SIAM J. Sci. Comput.* 40.3 (2018), A1696–A1713 (cited on pp. 90, 115).

[2] A. H. Al-Mohy and N. J. Higham. "A new scaling and squaring algorithm for the matrix exponential." *SIAM J. Matrix Anal. Appl.* 31.3 (2009), pp. 970–989 (cited on pp. 87, 90, 93, 107).

[3] A. H. Al-Mohy and N. J. Higham. "Computing the Fréchet derivative of the matrix exponential, with an application to condition number estimation." *SIAM J. Matrix Anal. Appl.* 30.4 (2009), pp. 1639–1657 (cited on pp. 100, 102, 113).

[4] A. H. Al-Mohy, N. J. Higham, and S. D. Relton. "Computing the Fréchet derivative of the matrix logarithm and estimating the condition number." *SIAM J. Sci. Comput.* 35.4 (2013), pp. C394–C410 (cited on p. 100).

[5] A. H. Al-Mohy, N. J. Higham, and S. D. Relton. "New algorithms for computing the matrix sine and cosine separately or simultaneously." *SIAM J. Sci. Comput.* 37.1 (2015), A456–A487 (cited on pp. 85, 87, 96, 105–107).

[6] P. Alonso, J. Ibánez, J. Sastre, J. Peinado, and E. Defez. "Efficient and accurate algorithms for computing matrix trigonometric functions." *J. Comput. Appl. Math.* 309 (2017), pp. 325–332 (cited on p. 88).

[7] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. "Julia: A fresh approach to numerical computing." *SIAM Rev.* 59.1 (2017), pp. 65–98 (cited on p. 86).

[8] M. Caliari and F. Zivcovich. "On-the-fly backward error estimate for matrix exponential approximation by Taylor algorithm." *J. Comput. Appl. Math.* 346 (2019), pp. 532–548 (cited on p. 85).

[9] P. I. Davies and N. J. Higham. "A Schur–Parlett algorithm for computing matrix functions." *SIAM J. Matrix Anal. Appl.* 25.2 (2003), pp. 464–485 (cited on p. 86).

[10] E. Defez, J. Ibánez, J. M. Alonso, and P. Alonso-Jordá. "On Bernoulli series approximation for the matrix cosine." *Math. Meth. Appl. Sci.* (2020), pp. 1–15 (cited on p. 88).

[11] E. Defez, J. Ibánez, J. Peinado, J. Sastre, and P. Alonso-Jordá. "An efficient and accurate algorithm for computing the matrix cosine based on new Hermite approximations." *J. Comput. Appl. Math.* 348 (2019), pp. 1–13 (cited on p. 88).

[12] N. J. Dingle and N. J. Higham. "Reducing the influence of tiny normwise relative errors on performance profiles." *ACM Trans. Math. Software* 39.4 (2013), 24:1–24:11 (cited on p. 108).

[13] E. D. Dolan and J. J. Moré. "Benchmarking optimization software with performance profiles." *Math. Program.* 91 (2002), pp. 201–213 (cited on p. 108).

[14] M. Fasi. "Optimality of the Paterson–Stockmeyer method for evaluating matrix polynomials and rational matrix functions." *Linear Algebra Appl.* 574 (2019), pp. 182–200 (cited on p. 92).

[15] M. Fasi and N. J. Higham. "An arbitrary precision scaling and squaring algorithm for the matrix exponential." *SIAM J. Matrix Anal. Appl.* 40.4 (2019), pp. 1233–1256 (cited on pp. 85, 89–92, 95, 106, 107).

[16] M. Fasi and N. J. Higham. "Multiprecision algorithms for computing the matrix logarithm." *SIAM J. Matrix Anal. Appl.* 39.1 (2018), pp. 472–491 (cited on pp. 85, 89).

[17] G. H. Golub and C. F. Van Loan. *Matrix Computations*. 4th ed., Baltimore, MD, USA: Johns Hopkins University Press, 2013, pp. xxi+756 (cited on p. 98).

[18] G. Hargreaves. "Topics in Matrix Computations: Stability and Efficiency of Algorithms." PhD thesis. Manchester, England: University of Manchester, 2005, p. 204 (cited on p. 92).

[19] G. I. Hargreaves and N. J. Higham. "Efficient algorithms for the matrix cosine and sine." *Numer. Algorithms* 40.4 (2005), pp. 383–400 (cited on pp. 87, 88, 93, 96, 108).

[20] N. J. Higham. *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008, pp. xx+425 (cited on pp. 87, 90, 93, 99, 100, 104, 107, 108, 113).

[21] N. J. Higham. *The Matrix Computation Toolbox*. `http://www.maths.manchester.ac.uk/~higham/mctoolbox` (cited on p. 108).

[22] N. J. Higham and L. Lin. "An improved Schur–Padé algorithm for fractional powers of a matrix and their Fréchet derivatives." *SIAM J. Matrix Anal. Appl.* 34.3 (2013), pp. 1341–1360 (cited on pp. 100, 114).

[23] N. J. Higham and X. Liu. "A multiprecision derivative-free Schur–Parlett algorithm for computing matrix functions." *SIAM J. Matrix Anal. Appl.* 42.3 (2021), pp. 1401–1422 (cited on p. 85).

[24] N. J. Higham and M. Mikaitis. "Anymatrix: An extensible MATLAB matrix collection." *Numer. Algorithms* (2021) (cited on p. 106).

[25] N. J. Higham and M. Mikaitis. *Anymatrix: An Extensible MATLAB Matrix Collection*. `https://github.com/mmikaitis/anymatrix` (cited on p. 106).

[26] N. J. Higham and S. D. Relton. "Estimating the condition number of the Fréchet derivative of a matrix function." *SIAM J. Sci. Comput.* 36.6 (2014), pp. C617–C634 (cited on p. 114).

[27] N. J. Higham and M. I. Smith. "Computing the matrix cosine." *Numer. Algorithms* 34 (2003), pp. 13–26 (cited on pp. 87, 93, 96, 106).

[28] N. J. Higham and F. Tisseur. "A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra." *SIAM J. Matrix Anal. Appl.* 21.4 (2000), pp. 1185–1201 (cited on p. 92).

[29] F. Johansson et al. *Mpmath: A Python Library for Arbitrary-Precision Floating-Point Arithmetic*. 2013. `http://mpmath.org` (cited on p. 85).

[30] A. Magnus and J. Wynn. "On the padé table of $\cos z$." *Proc. Amer. Math. Soc.* 47 (1975), pp. 361–367 (cited on p. 89).

[31] A. Meurer, C. P. Smith, M. Paprocki, et al. "SymPy: Symbolic computing in Python." *PeerJ Comput. Sci.* 3 (Jan. 2017), e103 (cited on p. 86).

[32] *Multiprecision Computing Toolbox*. Advanpix, Tokyo, Japan. `http://www.advanpix.com` (cited on pp. 85, 106–108).

[33] P. Nadukandi and N. J. Higham. "Computing the wave-kernel matrix functions." *SIAM J. Sci. Comput.* 40.6 (2018), A4060–A4082 (cited on pp. 90, 115).

[34] M. S. Paterson and L. J. Stockmeyer. "On the number of nonscalar multiplications necessary to evaluate polynomials." *SIAM J. Comput.* 2.1 (1973), pp. 60–66 (cited on p. 91).

[35] J. Sastre, J. Ibáñez, P. Ruiz, and E. Defez. "Accurate and efficient matrix exponential computation." *Internat. J. Comput. Math.* 91.1 (May 2013), pp. 97–112 (cited on pp. 87, 93).

[36] J. Sastre, J. Ibánez, P. Alonso, J. Peinado, and E. Defez. "Two algorithms for computing the matrix cosine function." *Appl. Math. Comput.* 312 (2017), pp. 66–77 (cited on pp. 88, 107).

[37] J. Sastre, J. Ibánez, P. Alonso-Jordá, J. Peinado, and E. Defez. "Fast Taylor polynomial evaluation for the computation of the matrix cosine." *J. Comput. Appl. Math.* 354 (2019), pp. 641–650 (cited on pp. 88, 107).

[38] S. M. Serbin. "Rational approximations of trigonometric matrices with application to second-order systems of differential equations." *Appl. Math. Comput.* 5 (1979), pp. 75–92 (cited on p. 85).

[39] S. M. Serbin and S. A. Blalock. "An algorithm for computing the matrix cosine." *SIAM J. Sci. Statist. Comput.* 1.2 (1980), pp. 198–204 (cited on p. 87).

[40] M. Seydaoğlu, P. Bader, S. Blanes, and F. Casas. "Computing the matrix sine and cosine simultaneously with a reduced number of products." *Appl. Numer. Math.* 163 (2021), pp. 96–107 (cited on p. 88).

5 COMPUTING THE SQUARE ROOT OF A LOW-RANK PERTURBATION OF THE SCALED IDENTITY MATRIX

Abstract. We consider the problem of computing the square root of a perturbation of the scaled identity matrix, $A = \alpha I_n + UV^*$, where U and V are $n \times k$ matrices with $k \leqslant n$. This problem arises in various applications, including computer vision and optimization methods for machine learning. We derive a new formula for the pth root of A that involves a weighted sum of powers of the pth root of the $k \times k$ matrix $\alpha I_k + V^*U$. This formula is particularly attractive for the square root, since the sum has just one term when $p = 2$. We also derive a new class of Newton iterations for computing the square root that exploit the low-rank structure. We test these new methods on random matrices and on positive definite matrices arising in applications. Numerical experiments show that the new approaches can yield a much smaller residual than existing alternatives and can be significantly faster when the perturbation UV^* has low rank.

Keywords: matrix pth root, matrix square root, low-rank update, matrix iteration, Newton iteration, MATLAB.
2010 MSC: 15A16, 65F60, 65F99.

5.1 INTRODUCTION

Any solution of the nonlinear equation $X^p = A$ is a pth root of the square matrix A. This matrix equation arises in many applications [18, sect. 2.14], and various methods for solving it numerically have been proposed in the literature. Particular attention has been devoted to the principal pth root $A^{1/p}$, which for a matrix with no eigenvalues on the closed negative real axis $\mathbb{R}^-$ is the unique pth root whose eigenvalues λ all satisfy $|\arg \lambda| < \pi/p$. For $p = 2$ one obtains the square root, which is the pth root most often needed in applications and most thoroughly investigated in the literature. Throughout this work, "pth root" refers to the principal pth root, and in particular "square root" refers to the principal square root, whose eigenvalues all lie in the open right half-plane.

The state-of-the-art methods for computing the matrix square root are based on the Schur decomposition [6], [8], [16], and can be extended to the computation of the pth root [12], [22], [31]. These methods have excellent numerical stability, in the sense that the computed solution satisfies essentially the same backward error bound as the rounded exact solution.

The Schur decomposition is typically computed using the QR algorithm [35], [36], [34], which is one of the most complex methods in matrix computations [11, sect. 7.5]. Implementing it in a robust and efficient way is a difficult task, so its low prevalence in libraries for matrix computations on custom hardware is not surprising. For example, a nonsymmetric dense eigensolver is not present in the NVIDIA cuSOLVER library.[1] Multiprecision environments often do not supply a routine for computing the Schur decomposition [10, sect. 5]; examples lacking one are the Julia language [5], which currently (version 1.7.3) does not provide it for its `BigFloat` data type,[2] and the MATLAB Symbolic Math Toolbox [33], where it is not available for the `sym` data type at the time of writing (version 9).

Matrix iterations for computing $A^{1/2}$ are an attractive alternative in these situations. Newton iterations converge quadratically in exact arithmetic and require only matrix multiplication and (in most cases) matrix inversion or the solution of multiple right-hand side linear systems. In deep learning, for example, the Newton–Schulz iteration [18, p. 153], [27] is widely used as an alternative to diagonalization when A is positive semidefinite. Being rich in matrix multiplication, it offers better performance on modern GPUs and it can speed up the forward propagation [32]. Tailored

1 `https://docs.nvidia.com/cuda/cusolver/`
2 A multiprecision Schur decomposition is available through the unofficial GenericSchur.jl package at `https://github.com/RalphAS/GenericSchur.jl`.

iterations are available for computing the square root of matrices with special structure or properties, including M-matrices, H-matrices, and Hermitian positive definite matrices; these are surveyed in [18, sect. 6.8].

Here we study methods for computing the square root of a matrix $A \in \mathbb{C}^{n\times n}$ of the form

$$A = \alpha I_n + UV^*, \quad \alpha \in \mathbb{C}, \quad U, V \in \mathbb{C}^{n\times k}, \quad k \leqslant n, \quad \Lambda(A) \cap \mathbb{R}^- = \varnothing, \tag{5.1}$$

where I_n is the identity matrix of order n and $\Lambda(A)$ denotes the spectrum of A. The condition $\Lambda(A) \cap \mathbb{R}^- = \varnothing$ implies that A is necessarily nonsingular, and if $k < n$, so that UV^* is rank deficient, it also implies that α lies off $\mathbb{R}^-$ (and in particular is nonzero). The same condition also requires that the $k \times k$ matrix $\alpha I_k + V^*U$ has no eigenvalues on $\mathbb{R}^-$, since the nonzero eigenvalues of BC and CB are the same for any two matrices B and C [18, Thm. 1.32], [20, Thm. 1.3.22].

An explicit expression for a function of a matrix in the form (5.1) is given by Higham in [18, Thm. 1.35] (and also by Harris in [15, Lem. 2] for $\alpha = 0$). The result allows us to evaluate $f(A)$ and only requires that f be defined on the spectrum of A. We recall this definition and give the corresponding theorem.

Definition 5.1 ([18, Def. 1.1]). Let $A \in \mathbb{C}^{n\times n}$, let $\lambda_1, \dots, \lambda_m$ be the distinct eigenvalues of A, and let $\zeta_1, \dots, \zeta_m$ be their respective indices (that is, ζ_i is the order of the largest Jordan block in which λ_i appears). A function f is defined on the spectrum of A if the values $f^{(j)}(\lambda_i)$ exist for $j = 0\colon \zeta_i - 1$ and $i = 1\colon m$.

Theorem 5.1 ([18, Thm. 1.35]). *Let $U, V \in \mathbb{C}^{n\times k}$ with $k \leqslant n$ and assume that V^*U is nonsingular. Let f be defined on the spectrum of $A = \alpha I_n + UV^*$, and if $k = n$ let f be defined at α. Then*

$$f(A) = f(\alpha)I_n + U(V^*U)^{-1}\big(f(\alpha I_k + V^*U) - f(\alpha)I_k\big)V^*. \tag{5.2}$$

The theorem says two things: that $f(A)$, like A, is a perturbation of rank at most k of the identity matrix and that $f(A)$ can be computed by evaluating f and the inverse at two $k \times k$ matrices. The formula (5.2) is of clear computational interest when $k \ll n$.

Note that if we take $f(x) = x^{-1}$ and write $A + UV^* = A(I_n + A^{-1}UV^*)$, then after a little manipulation we obtain as a special case of (5.2) the Sherman–Morrison–Woodbury formula, which says that if $I_k + V^*A^{-1}U$ is nonsingular then $A + UV^*$ is also nonsingular and

$$(A + UV^*)^{-1} = A^{-1} - A^{-1}U(I_k + V^*A^{-1}U)^{-1}V^*A^{-1}. \tag{5.3}$$

Taking for f the square root in (5.2) gives

$$A^{1/2} = \alpha^{1/2}I_n + U(V^*U)^{-1}\big((\alpha I_k + V^*U)^{1/2} - \alpha^{1/2}I_k\big)V^*. \tag{5.4}$$

This formula is valid only if V^*U is nonsingular, yet this condition is not required for $A^{1/2}$ to be defined. This is undesirable, since there is no guarantee that for a rank-k perturbation written as UV^* the matrix V^*U will be nonsingular. Consider the $k = 1$ case with $U = e_i$ and $V = e_j$ for $i \neq j$: formula (5.4) fails since $V^*U = 0$, and there is no alternative way of writing this perturbation. In general, we cannot avoid a singular V^*U given only the assumption that $A^{1/2}$ is well defined, and thus the formula cannot always be applicable.

Another problem with (5.2) is that it may not provide full accuracy when evaluated in floating-point arithmetic if the condition number of V^*U is large. This is illustrated in Figure 5.1, where we compare the accuracy of (5.4) and (5.9) (see below) on matrices of the form (5.1) for $n = 100$ and $\alpha = 1$. The square root of a $k \times k$ matrix is computed by the Schur method using the MATLAB function `sqrtm`. We gauge the accuracy by measuring the 2-norm relative residual of the equation that defines the square root, that is, the quantity

$$\frac{\|\widehat{X}^2 - A\|_2}{\|A\|_2}, \tag{5.5}$$

where $\widehat{X}$ is a computed approximation to the square root obtained in MATLAB using binary64 arithmetic with unit roundoff $u_{64} \approx 1.1 \times 10^{-16}$. In order to reduce the magnitude of possible roundoff errors in the computation of the relative residual,

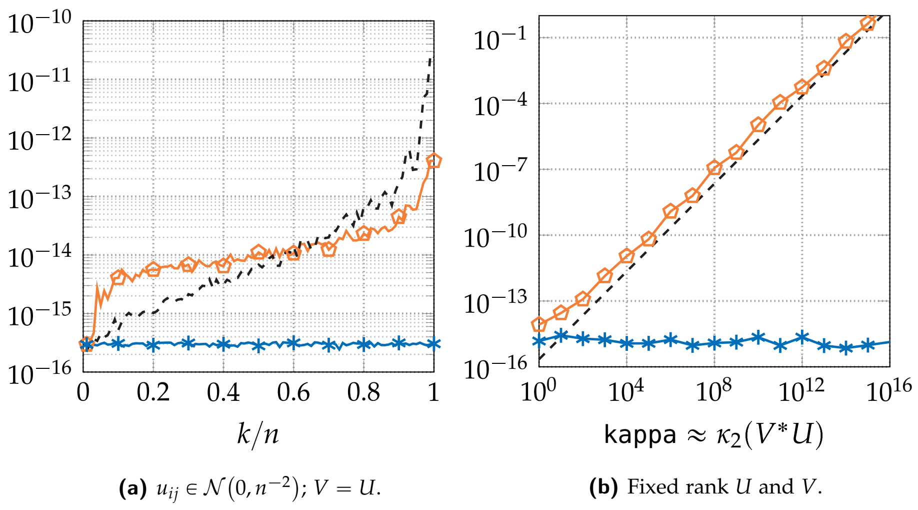


Figure 5.1: Relative residuals of (5.4) and (5.9) in the 2-norm. The matrix A is of the form (5.1) for $n = 100$, $\alpha = 1$, and $V = U$. The elements of U are drawn from different distributions in the two panels. Note that the y-axes have a different range.

(5.5) is evaluated in binary128 arithmetic by relying on the Multiprecision Computing Toolbox for MATLAB [25].

In Figure 5.1a, the matrix $U \in \mathbb{C}^{n\times k}$, for k varying between 1 and 100, has entries drawn from the normal distribution with mean 0 and variance n^{-2}, which we denote by $\mathcal{N}(0, n^{-2})$, and we set $V = U$ so that A is Hermitian. In Figure 5.1b, the parameter k is set to 10, and the matrices U and V are generated with the MATLAB code

```
S = logspace(-log10(kappa), 0, k);
U = orth(randn(n, k));
V = U .* S;
```

This ensures that V^*U has condition number approximately kappa, which in our experiment varies between 1 and 10^{16}. The relative residual of the solution computed using (5.4) deteriorates as the rank of UV^* or the condition number of V^*U increase.

Interestingly, the Sherman–Morrison–Woodbury formula (5.3) does not involve $(V^*U)^{-1}$, so for particular f this term does not necessarily have to appear in a formula for $f(A)$. We now derive a formula for the pth root of a matrix of the form (5.1) that does not have the restriction that V^*U be nonsingular.

Theorem 5.2. *Let $U, V \in \mathbb{C}^{n\times k}$ with $k \leqslant n$ have full rank and let the matrix $A = \alpha I_n + UV^*$ have no eigenvalues on $\mathbb{R}^-$. Then for any integer $p \geqslant 1$,*

$$A^{1/p} = \alpha^{1/p} I_n + U\left(\sum_{i=0}^{p-1} \alpha^{i/p} \cdot (\alpha I_k + V^*U)^{(p-i-1)/p}\right)^{-1} V^*. \tag{5.6}$$

Proof. Assume, first, that V^*U is nonsingular. Taking for f the pth root in Theorem 5.1 gives

$$A^{1/p} = \alpha^{1/p} I_n + U(V^*U)^{-1}\big((\alpha I_k + V^*U)^{1/p} - \alpha^{1/p} I_k\big)V^*. \tag{5.7}$$

On the other hand, from the identity $a^p - b^p = (a-b)\left(\sum_{i=0}^{p-1} a^{p-i-1}b^i\right)$ we have

$$(\alpha I_k + V^*U)^{1/p} - \alpha^{1/p} I_k = V^*U\left(\sum_{i=0}^{p-1} \alpha^{i/p} \cdot (\alpha I_k + V^*U)^{(p-i-1)/p}\right)^{-1}, \tag{5.8}$$

where the matrix in parentheses is nonsingular because $\alpha I_k + V^*U$ and αI_k have no eigenvalue in common, which means that the left-hand side of (5.8) is nonsingular. Using the identity (5.8) in (5.7) gives (5.6). If V^*U is singular, consider the matrix $A(t) = \alpha I_n + U(t)V^*$, where $U(t) = U + tV$ for $t \in \mathbb{R}$. Then $V^*U(t) = V^*U + tV^*V$ is nonsingular for sufficiently small t (specifically, for any $t > 0$ if V^*U has no negative real eigenvalues and otherwise for $t \in (0, |\lambda|)$, where λ is the algebraically largest negative real eigenvalue of V^*U). By (5.6) we have

$$A(t)^{1/p} = \alpha^{1/p} I_n + (U + tV)\left(\sum_{i=0}^{p-1} \alpha^{i/p} \cdot \big(\alpha I_k + V^*U + tV^*V\big)^{(p-i-1)/p}\right)^{-1} V^*,$$

and taking the limit as $t \to 0$ gives (5.6). □

Our main interest is in $p = 2$, for which we have the following corollary.

Corollary 5.3. *Let $U, V \in \mathbb{C}^{n\times k}$ with $k \leqslant n$ have full rank and let the matrix $A = \alpha I_n + UV^*$ have no eigenvalues on $\mathbb{R}^-$. Then*

$$A^{1/2} = \alpha^{1/2} I_n + U\big((\alpha I_k + V^*U)^{1/2} + \alpha^{1/2} I_k\big)^{-1} V^*. \tag{5.9}$$

The formula (5.9) in Corollary 5.3 is a significant improvement over (5.4), since it does not contain the factor $(V^*U)^{-1}$. Furthermore, in the experiments of Figure 5.1, formula (5.9) produces relative residuals of order u, unlike (5.4). Note that when $n = 1$, the difference between (5.4) and (5.9) boils down to the difference between expressions of the form $(\sqrt{1+x}-1)/x$ and $1/(\sqrt{1+x}+1)$, $x \in \mathbb{C}$; the latter does not require the inverse of x and is more stable when $|x|$ is small.

In section 5.2 we describe some applications that motivated this work. In section 5.3 we derive a Newton iteration that exploits the low-rank structure and provides an alternative to using (5.9); it is a structured variant of the Denman–Beavers iteration. We discuss Schur-based approaches for computing the square root in section 5.4, where we develop novel schemes that take advantage of the structure of the Schur decomposition. In section 5.5 we compare the computational cost of several methods applied to the explicitly formed A or to (5.9), and in section 5.6 we compare the methods in terms of numerical stability and speed on random matrices as well as on positive definite matrices arising in real applications. Concluding remarks are offered in section 5.7.

We note that similar problems have been addressed in the literature. Bernstein and Van Loan [4] proposed an algorithm for computing $f(X+uv^T)$ for $X \in \mathbb{R}^{n\times n}$ and $u, v \in \mathbb{R}^n$, where f is a rational function defined on the spectra of X and $X+uv^T$. Beckermann, Kressner, and Schweitzer [3] proposed a polynomial Krylov method for approximating $f(X+UV^*)$ for any $X \in \mathbb{C}^{n\times n}$ provided that UV^* has low rank and that f is analytic on some domain containing the spectra of X and $X+UV^*$. The algorithms are given and their convergence analyzed for the case of rank-1 perturbations, but the authors suggest two approaches to apply the proposed algorithms to higher rank. More recently, Beckermann, Cortinovis, Kressner, and Schweitzer [2] have developed a rational Krylov method to address the same problem. The proposed algorithm requires that the numerical range of X does not contain a singularity of f. For the matrix square root, the convergence of the algorithm may be slow when $\alpha \in \mathbb{C}$ in (5.1) is close to the origin in the complex plane. Being Krylov-based, these methods are necessarily iterative, and they aim to approximate the correction

$f(X+UV^*) - f(X)$ and compute $f(X+UV^*)$ as an update of $f(X)$, whereas the formula (5.9) is direct with a predictable cost and gives an explicit expression for $(\alpha I_n + UV^*)^{1/2}$.

Recently, Shumeli, Drineas, and Avron [30] developed a method to compute the quantity $(X \pm UU^T)^{\pm 1/2}$ for a symmetric positive semidefinite X. Their method is based on the approximate solution of an algebraic Riccati equation, and allows for either symmetric positive semidefinite or symmetric negative semidefinite perturbations.

5.2 APPLICATIONS

The need to compute roots of matrices of the form (5.1) arises in high-order optimization algorithms for machine learning [1], [14] and in machine vision [24].

The Shampoo technique, developed by Gupta, Koren, and Singer [14], is a preconditioned gradient method for second-order optimization. Computationally, the most expensive step of the algorithm is the evaluation of

$$L_t^{-1/2p} G_t R_t^{-1/2q}, \quad t = 1, \dots, \ell,$$

for some positive integers ℓ, p, and q where

$$L_t = \alpha I_n + \sum_{s=1}^{t} G_s G_s^*, \quad R_t = \alpha I_k + \sum_{s=1}^{t} G_s^* G_s, \quad \alpha > 0, \tag{5.10}$$

and $G_1, \dots, G_t \in \mathbb{R}^{n \times k}$ are of rank at most r. We note that the matrix $\sum_{s=1}^{t} G_s G_s^*$ can be written as UU^* where $U = [G_1 \dots G_t] \in \mathbb{R}^{n \times kt}$, which shows that L_t is of the form (5.1). The original implementation of Shampoo [14] used an SVD-based approach to compute the pth roots of L_t and G_t, but more recently Anil et al. [1] used the Schur–Newton algorithm of Guo and Higham [13] to compute the inverse pth roots. We note that the two algorithms are roughly equivalent for symmetric positive definite matrices such as L_t and R_t, as both the SVD and the Schur decomposition reduce to the eigendecomposition, and the only difference is in the way the pth roots of the eigenvalues are computed: the SVD-based algorithm computes the pth roots of the eigenvalues directly, whereas the Schur–Newton algorithm uses a scalar Newton iteration.

In representations for visual recognition [24, p. 39], the square root of a matrix of the form (5.1) is used in the spectral normalization of bilinear convolutional neural networks. This feature normalization technique runs an input image through a convolutional layer that extracts a set of k feature vectors $x_1, \dots, x_k \in \mathbb{R}^n$ with non-negative entries. These features are then aggregated via bilinear pooling, producing the matrix

$$A = \alpha I_n + \frac{1}{k}\sum_{i=1}^{k} x_i x_i^*, \quad \alpha > 0.$$

Since k depends on the size of the input image and of the convolutional filters, whereas n depends on the number of filters, these two numbers can be very different. Even when n and k are of similar magnitude, as often happens in state-of-the-art models [24, p. 52], many of the x_i may in principle be equal or very similar, producing a perturbation of rank much smaller than k. Because of the local nature of the convolutional filters, this is likely to happen when a large portion of the input image is filled by a homogeneous texture, as is the case for images with bursty features such as those considered in [24, Chap. 4]. The low-rank approximation can be obtained efficiently by using, for example, the randomized SVD algorithm recently developed by Nakatsukasa [26].

The need for computing the square root of a matrix of the form (5.1) also comes from numerical considerations in computing the square root of a singular or nearly singular matrix B. Rounding errors in floating-point arithmetic can displace small positive real eigenvalues of B to the negative real axis, where the principal square root is not well defined. In order to avoid potential issues, one can regularize B by adding the term αI for some small positive constant α: if B is factorized into a product of the form UV^* by truncating its singular value decomposition, then this diagonal shift produces a matrix of the form (5.1). This technique has been used, for example, to regularize some structured layers of deep neural networks [23]. The same regularization may be of interest when computing the inverse square roots of matrices of the form $\frac{1}{n}X^TX$, where X represents the data matrix and n is the number of samples. Matrices of this form arise in the training of deep neural networks [28], [37].

5.3 NEWTON ITERATIONS

An obvious approach for computing the square root is to apply any Newton iteration to A in (5.1) directly. For $k \ll n$, a more efficient strategy is to invoke (5.9) in Corollary 5.3 and apply the iteration to the $k \times k$ matrix $\alpha I_k + V^*U$. The standard Newton iteration is known to be numerically unstable [19], [18, sect. 6.4.1], so we focus on two of its numerically stable variants, namely the Denman–Beaver iteration (DB) and its product form.

The (scaled) DB iteration [9], [18, sect. 6.3] is

$$\begin{aligned} X_{i+1} &= \frac{1}{2}\left(\mu_i X_i + \mu_i^{-1} Y_i^{-1}\right), \quad X_0 = A, \\ Y_{i+1} &= \frac{1}{2}\left(\mu_i Y_i + \mu_i^{-1} X_i^{-1}\right), \quad Y_0 = I, \end{aligned} \tag{5.11}$$

where the positive scaling parameter $\mu_i \in \mathbb{R}$ can be used to accelerate the convergence of the method in its initial steps. The choice $\mu_i = 1$ yields the unscaled DB method, for which X_i and Y_i converge quadratically to $A^{1/2}$ and $A^{-1/2}$, respectively. An effective but possibly expensive technique for choosing the parameter μ_i is determinantal scaling, which we discuss later in this section.

We prove by induction that if A is of the form (5.1) then for $i \geqslant 0$ the iterates X_i and Y_i can be written in the form

$$X_i = \beta_i I_n + UB_iV^*, \quad \beta_i \in \mathbb{C}, \quad B_i \in \mathbb{C}^{k\times k}, \tag{5.12}$$

$$Y_i = \gamma_i I_n + UC_iV^*, \quad \gamma_i \in \mathbb{C}, \quad C_i \in \mathbb{C}^{k\times k}. \tag{5.13}$$

For $i = 0$, this follows from setting $\beta_0 = \alpha$, $B_0 = I_k$ and $\gamma_0 = 1$, $C_0 = 0$. For the inductive step, by using the Sherman–Morrison–Woodbury formula (5.3) for X_i^{-1} and Y_i^{-1} we obtain

$$\begin{aligned} X_{i+1} &= \frac{1}{2}\big(\mu_i X_i + \mu_i^{-1} Y_i^{-1}\big) \\ &= \frac{\mu_i}{2}(\beta_i I_n + U B_i V^*) + \frac{(\mu_i\gamma_i)^{-1}}{2}\big(I_n - U C_i(\gamma_i I_k + V^* U C_i)^{-1} V^*\big) \\ &= \frac{\mu_i\beta_i + (\mu_i\gamma_i)^{-1}}{2} I_n + \frac{1}{2} U\big(\mu_i B_i - (\mu_i\gamma_i)^{-1} C_i(\gamma_i I_k + V^* U C_i)^{-1}\big) V^*, \end{aligned}$$

and

$$\begin{aligned} Y_{i+1} &= \frac{1}{2}\big(\mu_i Y_i + \mu_i^{-1} X_i^{-1}\big) \\ &= \frac{\mu_i}{2}(\gamma_i I_n + U C_i V^*) + \frac{(\mu_i\beta_i)^{-1}}{2}\big(I_n - U B_i(\beta_i I_k + V^* U B_i)^{-1} V^*\big) \\ &= \frac{\mu_i\gamma_i + (\mu_i\beta_i)^{-1}}{2} I_n + \frac{1}{2} U\big(\mu_i C_i - (\mu_i\beta_i)^{-1} B_i(\beta_i I_k + V^* U B_i)^{-1}\big) V^*, \end{aligned}$$

so that

$$\beta_{i+1} = \frac{\mu_i\beta_i + (\mu_i\gamma_i)^{-1}}{2}, \tag{5.14a}$$

$$B_{i+1} = \frac{1}{2}\big(\mu_i B_i - (\mu_i\gamma_i)^{-1} C_i(\gamma_i I_k + V^* U C_i)^{-1}\big), \tag{5.14b}$$

$$\gamma_{i+1} = \frac{\mu_i\gamma_i + (\mu_i\beta_i)^{-1}}{2}, \tag{5.14c}$$

$$C_{i+1} = \frac{1}{2}\big(\mu_i C_i - (\mu_i\beta_i)^{-1} B_i(\beta_i I_k + V^* U B_i)^{-1}\big). \tag{5.14d}$$

With $W = V^* U \in \mathbb{C}^{k\times k}$ precomputed and stored, each step requires the solution of two $k \times k$ linear systems with k right-hand sides and two $k \times k$ matrix multiplications, for a total cost of $\frac{28}{3}k^3$ floating-point operations (flops).

Note that since $\beta_i \to \alpha^{1/2}$ and $\gamma_i \to \alpha^{-1/2}$, we might be tempted to remove the iterations for β_i and γ_i and replace β_i by $\alpha^{1/2}$ in (5.14d) and γ_i by $\alpha^{-1/2}$ in (5.14b). However, this choice changes the iteration, which is no longer convergent in general.

By introducing the product $M_i = X_i Y_i$ and rewriting (5.11), one of the inversions can be traded for a multiplication, giving the product form of the DB iteration [7], [18, sect. 6.3]

$$\begin{aligned} M_{i+1} &= \frac{1}{2}\left(I_n + \frac{\mu_i^2 M_i + \mu_i^{-2} M_i^{-1}}{2}\right), \quad & M_0 &= A, \\ X_{i+1} &= \frac{1}{2}\mu_i X_i \left(I_n + \mu_i^{-2} M_i^{-1}\right), & X_0 &= A. \end{aligned} \tag{5.15}$$

Here, $X_i \to A^{1/2}$ and $M_i \to I$ as $i \to \infty$.

Now we show that if A has the form (5.1) then for $i \geqslant 0$ the matrix M_i has the form

$$M_i = \nu_i I_n + U N_i V^*, \quad \nu_i \in \mathbb{C}, \quad N_i \in \mathbb{C}^{k\times k}, \tag{5.16}$$

and X_i has the form (5.12). For $i = 0$, this follows from setting $\nu_0 = \beta_0 = \alpha$ and $N_0 = B_0 = I_k$. For the inductive step, by using the Sherman–Morrison–Woodbury formula (5.3) for M_i^{-1} we obtain for M_{i+1} the expression

$$\begin{aligned} M_{i+1} &= \frac{1}{2}\left(I_n + \frac{\mu_i^2 M_i + \mu_i^{-2} M_i^{-1}}{2}\right) \\ &= \frac{1}{2}\left(I_n + \frac{\mu_i^2(\nu_i I_n + U N_i V^*) + \mu_i^{-2}\nu_i^{-1}(I_n - U N_i(\nu_i I_k + V^* U N_i)^{-1} V^*)}{2}\right) \\ &= \frac{2 + (\mu_i^2 \nu_i + \mu_i^{-2}\nu_i^{-1})}{4} I_n + \frac{1}{4} U\big(\mu_i^2 N_i - (\mu_i^2 \nu_i)^{-1} N_i (\nu_i I_k + V^* U N_i)^{-1}\big) V^* \\ &= \frac{2 + (\mu_i^2 \nu_i + \mu_i^{-2}\nu_i^{-1})}{4} I_n + \frac{1}{4} U\big(\mu_i^2 N_i - S_i\big) V^*, \end{aligned} \tag{5.17}$$

where

$$S_i = (\mu_i^2 \nu_i)^{-1} N_i (\nu_i I_k + V^* U N_i)^{-1}.$$

Similarly, for X_{i+1} we have

$$\begin{aligned} X_{i+1} &= \frac{\mu_i}{2} X_i\big(I_n + \mu_i^{-2} M_i^{-1}\big) \\ &= \frac{\mu_i}{2}(\beta_i I_n + UB_iV^*)\big(I_n + \mu_i^{-2}\nu_i^{-1}(I_n - UN_i(\nu_i I_k + V^*UN_i)^{-1}V^*)\big) \\ &= \frac{\mu_i}{2}\beta_i(1+\mu_i^{-2}\nu_i^{-1})I_n + \frac{1}{2}U\big((\mu_i + \mu_i^{-1}\nu_i^{-1})B_i - \mu_i\beta_i S_i - \mu_i B_i V^*US_i\big)V^*. \end{aligned} \tag{5.18}$$

From (5.17) and (5.18) we can read off formulas for ν_{i+1}, N_{i+1}, β_{i+1}, and B_{i+1} in terms of ν_i, N_i, β_i, and B_i.

With V^*U computed initially and stored, forming S_i requires one $k \times k$ matrix multiplication and one $k \times k$ linear system solve with k right-hand sides, and computing B_i takes two additional $k \times k$ matrix products. Therefore, each iteration entails three $k \times k$ matrix products and the solution of a $k \times k$ linear system with k right-hand sides, for a total cost of $\frac{26}{3}k^3$ flops.

An effective scaling is determinantal scaling, which is given for the DB iteration by

$$\mu_i = \left|\frac{\sqrt{\det(A)}}{\det(X_i)}\right|^{1/n} = \left|\frac{1}{\det(X_i)\det(Y_i)}\right|^{1/2n}$$

and for the product form of the DB iteration by

$$\mu_i = \left|\frac{1}{\det(M_i)}\right|^{1/2n}.$$

In order to perform this scaling efficiently, however, it is necessary to exploit the structure of the matrices A, X_i, Y_i, and M_i when computing their determinants. We explain how to compute the determinant of X_i in (5.12); those of A in (5.1), of Y_i in (5.13), and of M_i in (5.16) can be computed analogously. By exploiting the identity $\det(I + AB) = \det(I + BA)$, we obtain

$$\begin{aligned} \det(X_i) &= \det(\beta_i I_n + UB_iV^*) \\ &= \beta_i^n \det(I_k + \beta_i^{-1}(V^*U)B_i) \\ &= \beta_i^{n-k}\det(\beta_i I_k + (V^*U)B_i), \end{aligned}$$

where the last expression involves only $k \times k$ matrices. Since β_i can be small, to avoid underflow in forming β_i^{n-k} for large $n-k$ we should form directly

$$|\det(X_i)|^{1/n} = \beta_i^{1-k/n} \, |\det(\beta_i I_k + (V^*U)B_i)|^{1/n} \,,$$

rather than computing $\det(X_i)$ explicitly.

The det term itself is also prone to underflow, so care is needed in its evaluation.

If μ_i becomes an infinity, a NaN, or 0 we set $\mu_i = 1$. As is customary when using scaled iterations [17], we also set $\mu_i = 1$ when the relative difference between μ_{i-1} and μ_i becomes small.

5.4 SCHUR METHODS

In this section we consider Schur-based methods for computing the square root of A in (5.1). The most evident approach is employing the Schur method [6] directly on A, which costs roughly $28\frac{1}{3}n^3$ flops. A more efficient strategy when $k \ll n$ is invoking (5.9) in Corollary 5.3 and applying the Schur algorithm to the $k \times k$ matrix $\alpha I_k + V^*U$. For $k \ll n$ this reduces the cost to only $2kn^2$ in the leading order.

In addition to the above methods, we discuss how the structure of A in (5.1) can be exploited in several Schur-based methods.

5.4.1 Exploiting structure in Schur decomposition

For the Schur decomposition of $A = \alpha I_n + UV^*$ it is sufficient to consider the Schur decomposition of UV^*. Suppose

$$U = [Q_1 \quad Q_2] \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$$

is a (full) QR decomposition of U, where $Q_1 \in \mathbb{C}^{n\times k}$, $Q_2 \in \mathbb{C}^{n\times(n-k)}$, and $R_1 \in \mathbb{C}^{k\times k}$. Then we can write

$$UV^* = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} Z \\ 0 \end{bmatrix} =: \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} Z_1 & Z_2 \\ 0 & 0 \end{bmatrix},$$

where $Z = R_1 V^* \in \mathbb{C}^{k\times n}$, $Z_1 \in \mathbb{C}^{k\times k}$, and $Z_2 \in \mathbb{C}^{k\times(n-k)}$. Then the remaining task is to find a QR decomposition of the matrix $\widetilde{Z} := \left[\begin{smallmatrix} Z_1 & Z_2 \\ 0 & 0 \end{smallmatrix}\right]$. Suppose $Z_1 = PT$ is a (full) QR decomposition of Z_1, where $P \in \mathbb{C}^{k\times k}$ is unitary and $T \in \mathbb{C}^{k\times k}$ is upper triangular, then

$$\begin{bmatrix} Z_1 & Z_2 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} P & 0 \\ 0 & I_{n-k} \end{bmatrix} \begin{bmatrix} T & P^* Z_2 \\ 0 & 0 \end{bmatrix}$$

is a QR decomposition of $\widetilde{Z}$. So we have

$$UV^* = \begin{bmatrix} Q_1 P & Q_2 \end{bmatrix} \begin{bmatrix} T & P^* Z_2 \\ 0 & 0 \end{bmatrix} := QR.$$

If we repartition the unitary factor $Q \in \mathbb{C}^{n\times n}$ into

$$Q = \begin{bmatrix} Q_1 P & Q_2 \end{bmatrix} = \begin{bmatrix} Q_a \\ Q_b \end{bmatrix},$$

where $Q_a \in \mathbb{C}^{k\times n}$ and $Q_b \in \mathbb{C}^{(n-k)\times n}$, then we have

$$RQ = \begin{bmatrix} TQ_a + P^* Z_2 Q_b \\ 0 \end{bmatrix} \tag{5.19}$$

which can be partitioned in the same form as $\widetilde{Z}$, so we can compute its QR decomposition in the same fashion.

From the above discussion we can build a basic QR algorithm for reducing A to the Schur form. This algorithm avoids form the product UV^* explicitly and works with a $k \times n$ matrix in each iteration. For $k \ll n$ the algorithm costs $2kn^2 + O(k^2 n)$ for the initial calculation to obtain (5.19) plus $2k^2 n$ flops in the leading order in each iteration afterwards, so it can be very efficient in this case. However, the convergence of such a basic QR algorithm is not always guaranteed [11, sect. 7.3] and those advanced shifting techniques discussed in [11, sect. 7.5] for accelerating convergence do not appear applicable in our case. The iteration is thus of limited practical interest due to this potential issue.

5.4.2 Working with block-factorized matrices

Writing

$$A = \alpha I_n + UV^* = \begin{bmatrix} \alpha I_n & U \end{bmatrix} \begin{bmatrix} I_n \\ V^* \end{bmatrix} =: BC, \quad B \in \mathbb{C}^{n\times(n+k)}, C \in \mathbb{C}^{(n+k)\times n}$$

and employing the identity $f(BC)B = Bf(CB)$ [18, Cor. 1.34], we get

$$A^{1/2} = (BC)^{1/2} = B(CB)^{1/2}B^*(BB^*)^{-1}, \tag{5.20}$$

where $BB^* = \alpha^2 I + UU^*$ is positive definite and

$$CB = \begin{bmatrix} \alpha I_n & U \\ \alpha V^* & V^*U \end{bmatrix}$$

is a block arrowhead matrix with leading diagonal block. We can exploit this special structure of CB when computing its Schur decomposition, in the initial step of the QR algorithm in which the Hessenberg form is computed. Compared with a full $(n+k) \times (n+k)$ matrix where we need zero out $n^2/2 + kn + k^2/2 + O(n)$ elements to get a Hessenberg form, there are only $kn + k^2/2 + O(k)$ nonzero elements that we need to tackle in CB. In the case $k \ll n$, which is the necessarily the situation when this method is worth considering, Givens rotations should be preferred in order to fully exploit the structure in CB. Under this assumption the initial reduction to Hessenberg form only costs $O(kn^2)$ flops, and the total cost of the Schur method [6] on CB reduces to approximately $25(n+k)^3$ flops.

In the setting $k \ll n$, the total cost of this block-factorized method via (5.20) is $27\frac{2}{3}n^3$ flops in the leading order, where the matrix products involving B or B^* can be formed in $O(kn^2)$ flops by exploiting the partitioned block structure so are negligible.

Even though we can exploit the structure in CB, this block-factorized method is still too expensive to be considered in practice.

5.4.3 Factorizing UV^* from factorizations of V^*U

The spectral decomposition and the Schur decomposition of the $n \times n$ matrix UV^* can be computed from the corresponding decomposition of the $k \times k$ matrix V^*U. The methods should be of interest only when $k \ll n$ as to computational efficiency.

Spectral decomposition. In order to compute the spectral decomposition of the matrix A in (5.1) we will exploit the fact that the nonzero eigenvalues of UV^* are eigenvalues of V^*U [18, Thm. 1.32], [20, Thm. 1.3.22].

Let (λ, x) be an eigenpair of $V^*U \in \mathbb{C}^{k\times k}$, then $V^*Ux = \lambda x$ implies that $(UV^*)Ux = \lambda Ux$ or, in other words, that (λ, Ux) is an eigenpair of $UV^* \in \mathbb{C}^{n\times n}$. Therefore, we can compute the nonzero eigenvalues of UV^* and the corresponding eigenvectors by computing the eigensystem of the $k \times k$ matrix V^*U. However, in the general case for nonnormal V^*U its eigenvectors may not form a basis for $\mathbb{C}^k$ or its eigenvectors can be ill-conditioned, in which cases this method is not suitable.

If we assume that $U = V$, then by the spectral theorem there exist $X \in \mathbb{C}^{k\times k}$ unitary and $\Lambda \in \mathbb{C}^{k\times k}$ diagonal such that $U^*U = X\Lambda X^*$. The columns of $UX \in \mathbb{C}^{n\times k}$ are orthogonal, since $(UX)^*(UX) = X^*U^*UX = \Lambda$, and are eigenvectors of UU^*. We can normalize the columns of UX and then add $n-k$ orthonormal columns to the matrix until we obtain $Y \in \mathbb{C}^{n\times n}$, an orthonormal basis of $\mathbb{C}^n$ that satisfies

$$UU^* = Y\widetilde{\Lambda}Y^*, \quad \widetilde{\Lambda} = \begin{bmatrix} \Lambda & 0 \\ 0 & 0 \end{bmatrix} \in \mathbb{C}^{n\times n}. \tag{5.21}$$

Alternatively, we can obtain a spectral decomposition of UU^* by using a QR decomposition of U, exploiting the fact it can be formed by the product of k Householder reflectors (in the case of U being square with $n = k$, there are $k - 1$ Householder reflectors needed, but our main interest is in the case $k \ll n$):

$$P_k P_{k-1} \dots P_2 P_1 U = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}, \quad P_i = I_n - \frac{2}{v_i^* v_i} v_i v_i^* \tag{5.22}$$

for some $0 \neq v_i \in \mathbb{C}^n$, each of which can be efficiently formed as a matrix-vector product followed by an outer product. The total cost of the QR decomposition $U = QR$ is $O(kn^2)$ flops (with explicit formation of the unitary factor $Q = P_1 P_2 \dots P_{k-1} P_k$). Suppose

$$U = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$$

is a (full) QR decomposition of U, where $Q_1 \in \mathbb{C}^{n\times k}$, $Q_2 \in \mathbb{C}^{n\times(n-k)}$, and $R_1 \in \mathbb{C}^{k\times k}$. Then we have

$$UU^* = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 R_1^* & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} Q_1^* \\ Q_2^* \end{bmatrix} = Q_1 R_1 R_1^* Q_1^*.$$

Therefore, if a spectral decomposition $R_1 R_1^* = P \Lambda P^{-1} = P \Lambda P^*$ is formed, where $P \in \mathbb{C}^{k\times k}$ is unitary and $\Lambda \in \mathbb{C}^{k\times k}$ is diagonal, we obtain a decomposition of exactly the same form as (5.21), in which case the $Y \in \mathbb{C}^{n\times n}$, an orthonormal basis of $\mathbb{C}^n$, is computed by explicitly forming $Q_1 P \in \mathbb{C}^{n\times k}$ and then adding $n - k$ orthonormal columns to it.

Then we have $A = \alpha I_n + UU^* = \alpha YY^* + Y\widetilde{\Lambda}Y^* = Y(\alpha I_n + \widetilde{\Lambda})Y^*$ and thus

$$A^{1/2} = Y(\alpha I_n + \widetilde{\Lambda})^{1/2} Y^*. \tag{5.23}$$

The asymptotic cost of the algorithm is $4\frac{2}{3}n^3$ flops in total. The dominant computational complexity of this algorithm is in *completing* the orthogonal matrix Y, which costs $8/3n^3$ flops [11, sect. 5.2.9], and forming the root $A^{1/2}$, which requires $2n^3$ flops.

In fact, once we have formed the k Householder reflectors $P_1P_2\dots P_{k-1}P_k$ of (5.22) in the QR decomposition of U, a more efficient computation scheme is to apply these Householder reflectors directly to the matrix $\alpha I_n + UU^*$. We then have

$$P_kP_{k-1}\dots P_2P_1(\alpha I_n + UU^*)P_1P_2\dots P_{k-1}P_k = \begin{bmatrix} \alpha I_k + R_1R_1^* & 0 \\ 0 & \alpha I_{n-k} \end{bmatrix},$$

from which we have

$$(\alpha I_n + UU^*)^{1/2} = P_1\dots P_k \begin{bmatrix} (\alpha I_k + R_1R_1^*)^{1/2} & 0 \\ 0 & \alpha^{1/2} I_{n-k} \end{bmatrix} P_k\dots P_1.$$

The computation of the square root of $\alpha I_k + R_1R_1^*$ only requries $O(k^3)$ flops, and applying each Householder reflector requires $O(n^2)$ flops (as discussed above), so the asymptotic cost of the algorithm is only $O(kn^2)$ flops. Finally, we note that this algorithm could be made more efficient if the Householder reflectors are applied in blocks [11, sect. 5.2.3].

Schur decomposition. The analogous method for the Schur decomposition exploits a similar idea, but the triangular form creates additional difficulties.

For any $U, V \in \mathbb{C}^{n\times k}$ there exist $W \in \mathbb{C}^{k\times k}$ unitary and $S \in \mathbb{C}^{k\times k}$ quasi-upper triangular such that $V^*U = WSW^*$. By multiplying both sides of this equation by U on the left and by W on the right, we obtain

$$(UV^*)(UW) = (UW)S. \tag{5.24}$$

The matrix UW is not unitary in general, but it has the thin QR factorization $UW =: QR$, where $Q \in \mathbb{C}^{n\times k}$ has orthonormal row and $R \in \mathbb{C}^{k\times k}$ is upper triangular. We can rewrite (5.24) as $Q^*(UV^*)Q = RSR^{-1}$ and computing the Schur decomposition of the right-hand side yields

$$Q^*(UV^*)Q = \widetilde{P}\widetilde{T}\widetilde{P}^*, \tag{5.25}$$

where $\widetilde{P}, \widetilde{T} \in \mathbb{C}^{k \times k}$ are unitary and quasi-upper triangular, respectively. The matrix $\widetilde{Q} := Q\widetilde{P}$ has orthonormal columns, and we can construct a unitary basis of $\mathbb{C}^n$ by finding a matrix $\widetilde{Y} \in \mathbb{C}^{n \times (n-k)}$ such that $P := \begin{bmatrix} \widetilde{Q} & \widetilde{Y} \end{bmatrix}$ is unitary. Thus we have

$$P^*(UV^*)P = \begin{bmatrix} \widetilde{Q}^* \\ \widetilde{Y}^* \end{bmatrix} UV^* \begin{bmatrix} \widetilde{Q} & \widetilde{Y} \end{bmatrix} = \begin{bmatrix} \widetilde{Q}^*UV^*\widetilde{Q} & \widetilde{Q}^*UV^*\widetilde{Y} \\ \widetilde{Y}^*UV^*\widetilde{Q} & \widetilde{Y}^*UV^*\widetilde{Y} \end{bmatrix}.$$

From (5.25) we can easily derive $UV^* = \widetilde{Q}\widetilde{T}\widetilde{Q}^*$, and since $\widetilde{Y}^*\widetilde{Q} = 0$ by construction, we have that $\widetilde{Y}^*UV^* = 0$. Therefore, we have that

$$UV^* = P^* \begin{bmatrix} \widetilde{T} & \widetilde{Q}^*UV^*\widetilde{Y} \\ 0 & 0 \end{bmatrix} P =: PTP^* \tag{5.26}$$

is a Schur decomposition of UV^*, and a Schur decomposition of A in (5.1) is $A = \alpha I_n + UV^* = \alpha PP^* + PTP^* = P(\alpha I_n + T)P^*$, from which $A^{1/2}$ can be computed using the Schur algorithm [6], [16].

The total cost of the algorithm is $6n^3$ flops in the leading order, which includes completing the orthogonal matrix P that costs $8/3n^3$ flops [11, sect. 5.2.9] and computing the square root $A^{1/2}$ and forming A, which costs $n^3/3$ flops and $3n^3$ flops, respectively. The methods discussed above are therefore asymptotically more expensive than using formula (5.9) with Schur method (or spectral method, in the symmetric case) under the setting $k \ll n$, so again we do not consider them further.

5.5 COST COMPARISON OF THE METHODS

Now we compare the computational cost of the methods discussed in the previous sections for computing the square root of the matrix A in (5.1). The methods that are not considered practical or only applicable to A with $U = V$ are excluded.

In Table 5.1 the methods are divided into two categories: Schur-based methods and Newton methods. We report the asymptotic cost of the methods, where we assume that the linear systems are solved using LU factorization. We give the cost in terms

Table 5.1: Asymptotic cost of methods for computing $(\alpha I + UV^*)^{1/2}$ for $U, V \in \mathbb{C}^{n\times k}$. The second and third column report the cost of the methods in terms of flops and matrix operations, respectively. Here, D_k, I_k, and M_k denote the solution of a $k \times k$ linear system with k right-hand sides, the inversion of a matrix of order k, and the multiplication of two matrices of order k, respectively. For the iterative methods, N is the total number of iterations performed.

	Method	Total flops	Operations per iteration
Schur-based:	Schur method	$28\frac{1}{3}n^3$	–
	Formula (5.9) with Schur method	$2kn^2 + 4k^2n + 29k^3$	–
Newton:	DB iteration	$4Nn^3$	$2\,\mathrm{I}_n$
	Product DB iteration	$4Nn^3$	$\mathrm{M}_n + \mathrm{I}_n$
	Structured DB	$2kn^2 + 4k^2n + 9\frac{1}{3}Nk^3$	$2\,\mathrm{M}_k + 2\,\mathrm{D}_k$
	Structured product DB	$2kn^2 + 4k^2n + 8\frac{2}{3}Nk^3$	$3\,\mathrm{M}_k + \mathrm{D}_k$
	Formula (5.9) with DB	$2kn^2 + 4k^2n + 4Nk^3$	$2\,\mathrm{I}_k$
	Formula (5.9) with product DB	$2kn^2 + 4k^2n + 4Nk^3$	$\mathrm{M}_k + \mathrm{I}_k$

of flops and in terms of matrix multiplications, matrix inversions, and multiple-right-hand-side system solves.

For $k \ll n$, the computational cost of computing $A^{1/2}$ is reduced from $O(n^3)$ for the standard (unstructured) methods to $O(n^2)$ for the methods that exploit the low-rank structure, assuming that for the Newton methods the number of iterations does not depend on k or n. Among the Schur-free methods that exploit the low-rank structure, formula (5.9) has the least cost, regardless of what form of the DB iteration is used to compute the $k \times k$ square root. It will be cheaper to evaluate (5.9) using the DB iteration (plain or in product form) as long as convergence is achieved in no more than 7 steps, whilst the Schur method will be more convenient for matrices that would require 8 or more iterations.

5.6 NUMERICAL EXPERIMENTS

In this section we evaluate the performance of four methods for computing the square root of matrices of the form (5.1), which are implemented in the following MATLAB codes.

- `schur_full`: an algorithm that first builds the matrix A in (5.1) by computing the outer product and then computes its square root using the MATLAB function `sqrtm`, which implements the Schur method [6], [8], [16]. If $U = V$, the matrix A is normal and its triangular Schur factor is diagonal. In this case, this algorithm reduces to the computation of $Q\Lambda^{1/2}Q^*$ where $A = Q\Lambda Q^*$ is a spectral decomposition of A.

- `schur_k`: an implementation of (5.9), where the square root of the $k \times k$ matrix is computed using `sqrtm` and the $k \times k$ linear system with k right-hand sides is solved using the MATLAB backslash operator.

- `db_prod_k`: an implementation of (5.9), where the square root of the $k \times k$ matrix is computed using the DB iteration in product form (5.15) with determinantal scaling and the $k \times k$ linear system with k right-hand sides is solved using the MATLAB backslash operator.

- `db_prod_struct`: the structured DB iteration in product form discussed in section 5.3, which iterates on $k \times k$ matrices. The algorithm uses the determinantal scaling in (5.16) and (5.17).

The experiments were run using the 64-bit GNU/Linux version of MATLAB 9.11.0 (R2021b Update 1) on a machine equipped with an AMD Ryzen 7 Pro 5850U running at 1.90GHz and 32 GiB of RAM. The code we used to produce the results in this section is available on GitHub.[3]

For the DB iterations we use the following stopping criterion: we look at B_i, one of the $k \times k$ matrices on which we iterate, and we return the current approximation when the 1-norm of the relative change between two successive iterations falls below a given tolerance τ, which for our experiments was set to $10u_{64}$ for binary64 arithmetic and $8u_{32}$ for binary32 precision, where $u_{64} = 2^{-53} \approx 1 \times 10^{-16}$ and $u_{32} = 2^{-24} \approx 6 \times 10^{-8}$ are the unit roundoffs of binary64 and binary32 arithmetic, respectively.

3 https://github.com/Xiaobo-Liu/sqrtm-lrpsi

For each computed square root $\widehat{X}$ of A, we compute the 2-norm relative residual in (5.5) and we gauge the quality of $\widehat{X}$ by comparing the relative residual with the quantity $\alpha_2(X)u$, where

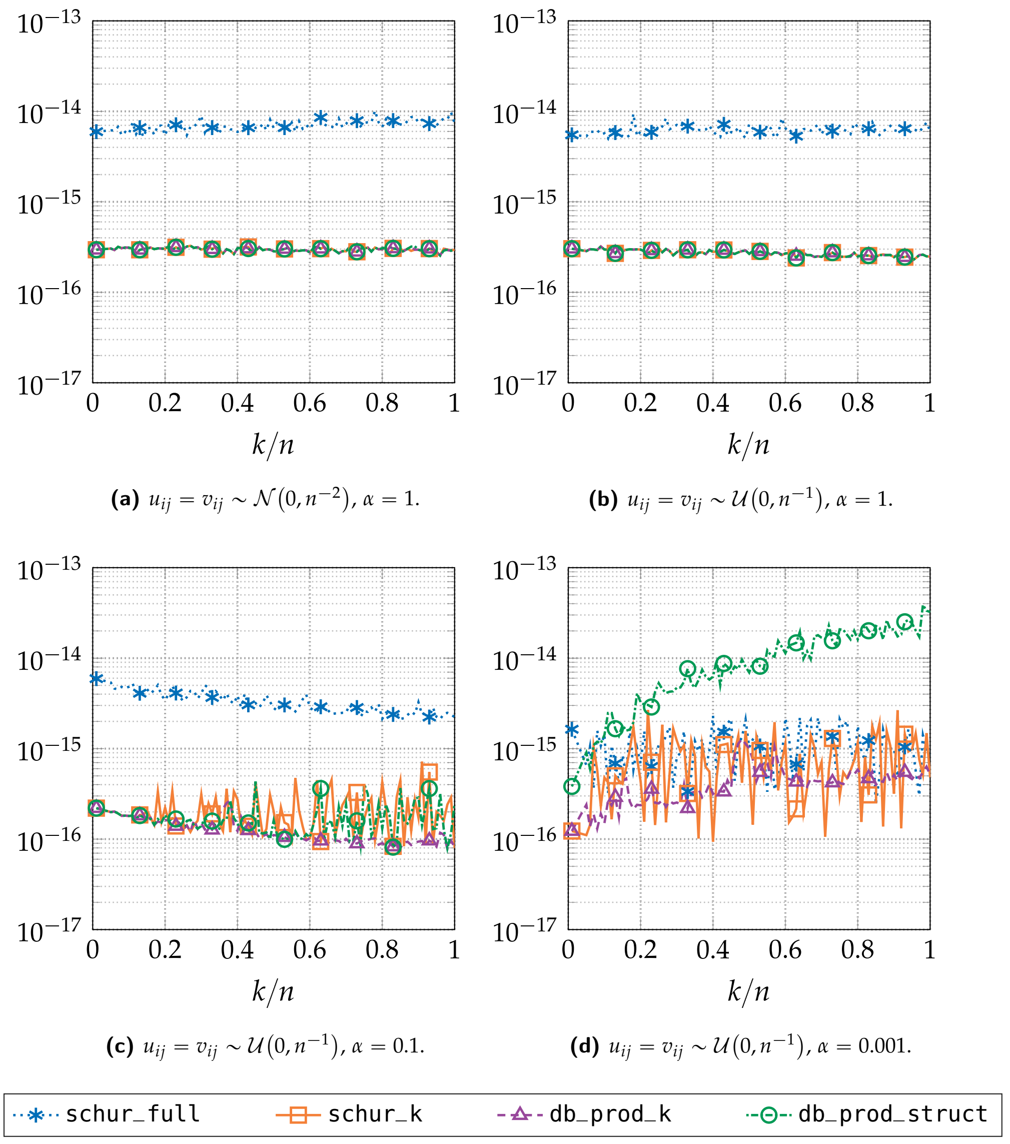
$$\alpha_2(X) = \frac{\|X\|_2^2}{\|A\|_2}$$

can be regarded as a condition number for the relative residual [6, sect. 4], [16, sect. 5], and u is the unit roundoff of the working precision. We note that $\alpha_2(X) \equiv 1$ if A is normal [6, sect. 4], thus we do not report the value of $\alpha_2(X)$ in such cases.

5.6.1 Quality

First we compare our four implementations in terms of the quality of the computed solution. In these experiments we consider the matrix A in (5.1) for $U = V$ and $n = 100$. For the dimension k we vary the ratio k/n from 0 to 1 with increment of 0.05 and replace the zero ratio by 0.01. Figure 5.2 reports the relative residual (5.5). The matrix U has entries drawn from $\mathcal{N}(0, n^{-2})$ in Figure 5.2a, and from the uniform distribution over the open interval $(0, n^{-1})$, which we denote by $\mathcal{U}(0, n^{-1})$, in Figure 5.2b. In these two figures $\alpha = 1$ is used. In Figure 5.2c and Figure 5.2d the matrix U has entries drawn from $\mathcal{U}(0, n^{-1})$ with $\alpha = 0.1$ and $\alpha = 0.001$, respectively.

In Figure 5.2a and Figure 5.2b the relative residual of `db_prod_struct` is indistinguishable from that of `schur_k` and `db_prod_k`, which exploit the formula (5.9). The relative residual of `schur_full`, on the other hand, is about one and a half orders of magnitude larger in both cases, and in fact `schur_full` is the only algorithm that produces a relative residual not of the same magnitude as $\alpha_2(X)u$; the reason for this mild instability is not clear. In Figure 5.2c the performance of the algorithms does not change much from that in Figure 5.2b when α decreases to 0.1. In Figure 5.2d we see that if $\alpha = 0.001$ then `db_prod_struct` shows signs of instability as k/n approaches 1, while the other algorithms remain largely stable and `schur_full` has relatively better performance for small α. With the chosen values of α, the matrix A is very well condi-



(a) $u_{ij} = v_{ij} \sim \mathcal{N}(0, n^{-2})$, $\alpha = 1$.

(b) $u_{ij} = v_{ij} \sim \mathcal{U}(0, n^{-1})$, $\alpha = 1$.

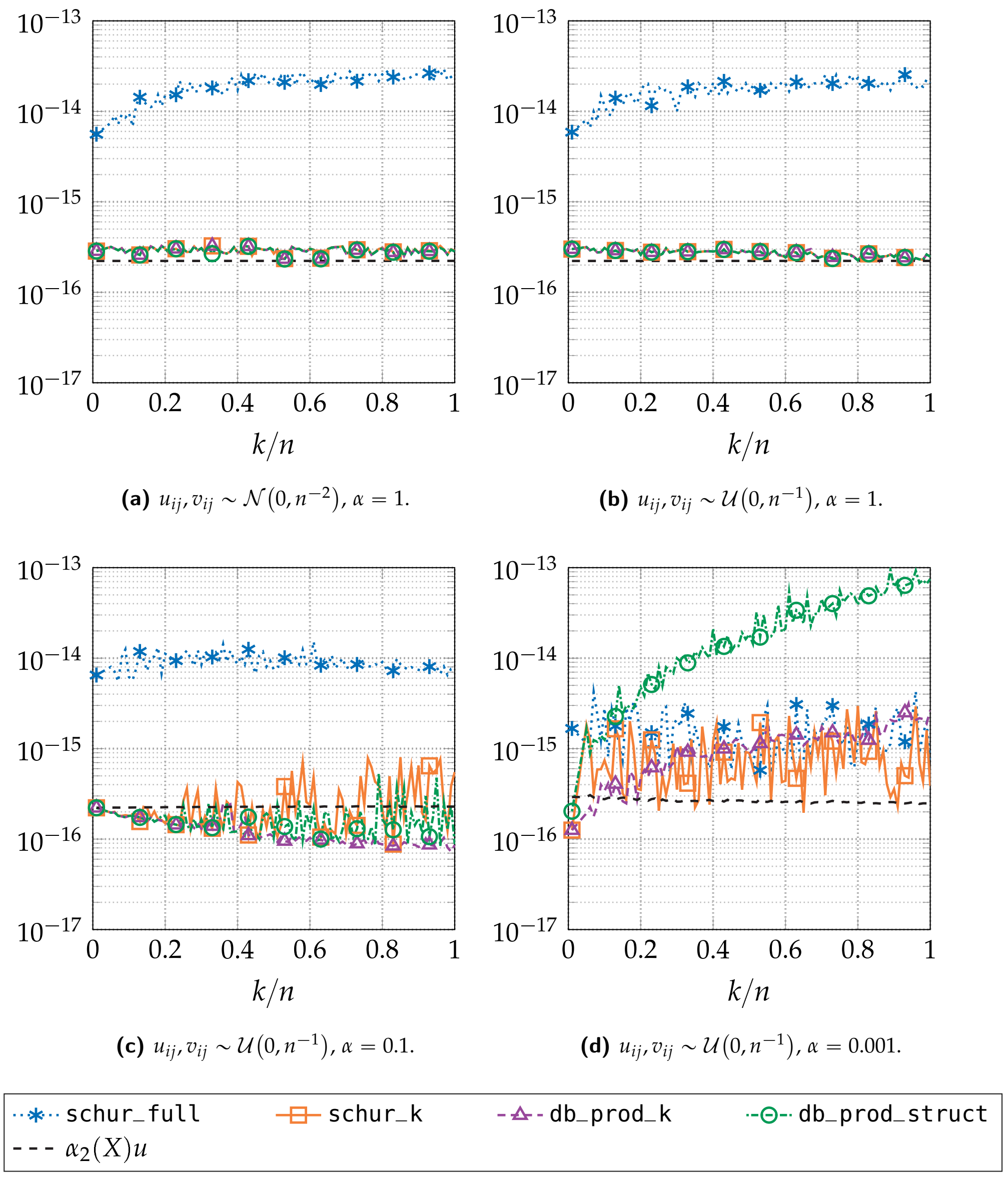
(c) $u_{ij} = v_{ij} \sim \mathcal{U}(0, n^{-1})$, $\alpha = 0.1$.

(d) $u_{ij} = v_{ij} \sim \mathcal{U}(0, n^{-1})$, $\alpha = 0.001$.

Figure 5.2: Relative residuals of algorithms for computing the square root. The matrix A has the form (5.1) for $n = 100$, various choices of α, and $V = U$. The elements of U are drawn from different distributions.

tioned, as $\kappa_2(A) < 10$. We repeated the experiment with the setting in Figure 5.2d but further decreasing α to 10^{-6}. In this case $\kappa_2(A) = O(n)$, and the algorithms behaved as in Figure 5.2d.

Next we test the algorithms on nonsymmetric matrices. We use the same experimental settings as in previous tests, but we now set $U \neq V$ so that A is nonsymmetric. The results are shown in Figure 5.3. There is no substantial difference between the behavior of the algorithms on symmetric and nonsymmetric matrices, although we



(a) $u_{ij}, v_{ij} \sim \mathcal{N}(0, n^{-2})$, $\alpha = 1$.

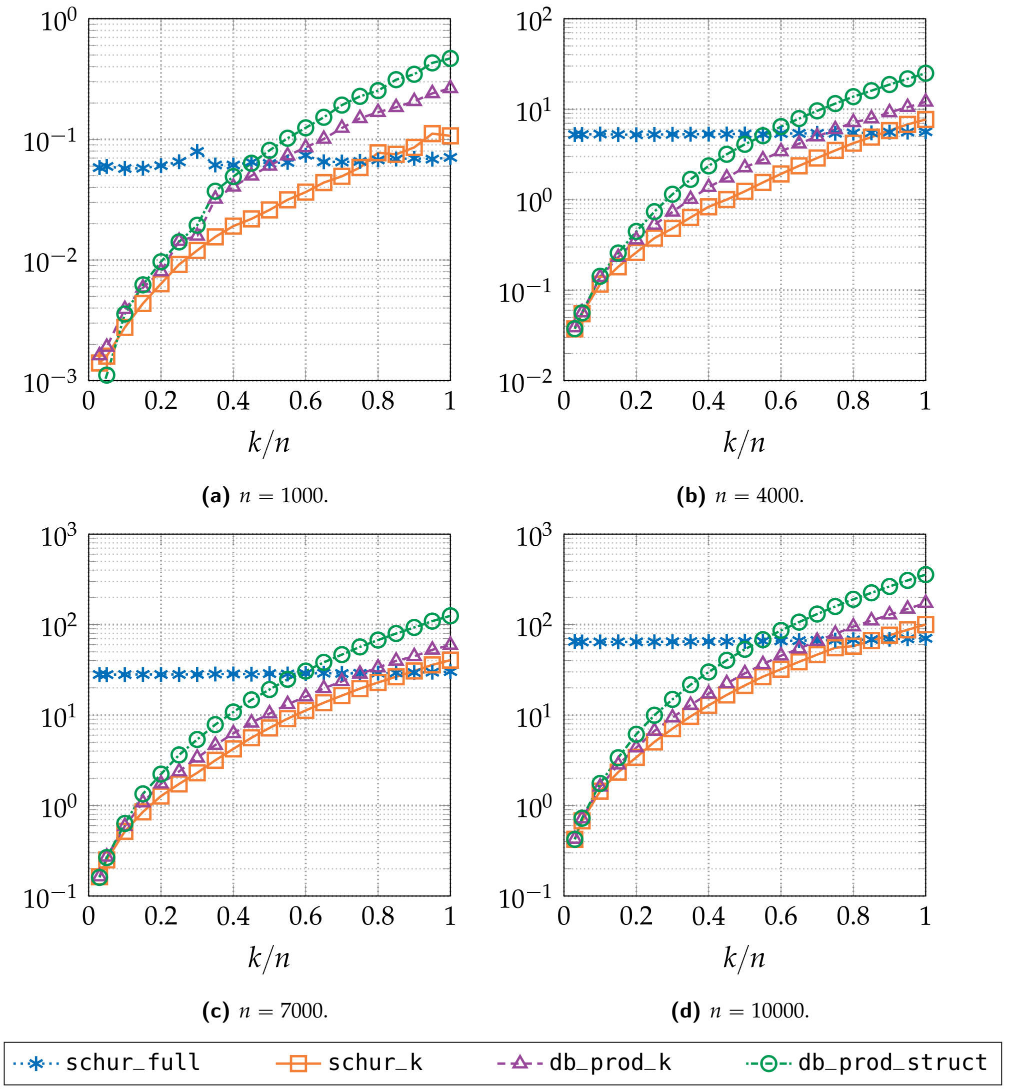
(b) $u_{ij}, v_{ij} \sim \mathcal{U}(0, n^{-1})$, $\alpha = 1$.

(c) $u_{ij}, v_{ij} \sim \mathcal{U}(0, n^{-1})$, $\alpha = 0.1$.

(d) $u_{ij}, v_{ij} \sim \mathcal{U}(0, n^{-1})$, $\alpha = 0.001$.

Figure 5.3: Relative residuals of algorithms for computing the square root. The matrix A has the form (5.1) for $n = 100$, various choices of α, and $V \neq U$. The elements of U and V are drawn from different distributions.

note that the quality of the solutions computed by the Schur-based algorithms slightly deteriorates in Figure 5.3 compared with the results in Figure 5.2.



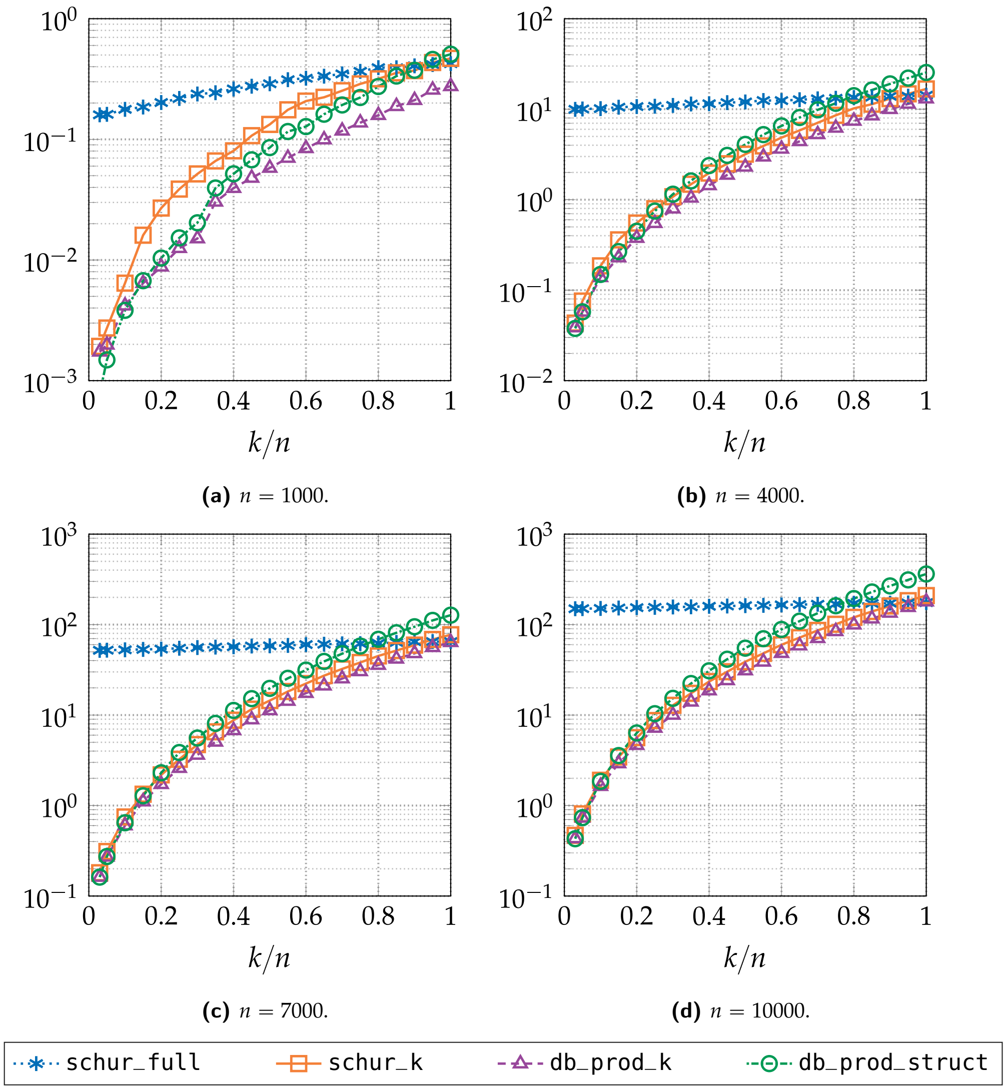
(a) $n = 1000$. **(b)** $n = 4000$.

(c) $n = 7000$. **(d)** $n = 10000$.

Figure 5.4: Execution times (in seconds) of algorithms for computing the square root. The matrix A has the form (5.1) for $\alpha = 0.1$ and $V = U$. The elements of U are drawn from $\mathcal{N}(0, n^{-2})$.

5.6.2 Timings

In Figure 5.4 we gauge how the execution time of our MATLAB implementations changes as the ratio k/n varies. In this experiment we consider the matrix A in (5.1) for $\alpha = 0.1$, $U = V$, and $n = 1000$, 4000, 7000, and 10000. The results reported here are for $u_{ij} \sim \mathcal{N}(0, n^{-2})$, but we have repeated the experiment with $u_{ij} \sim \mathcal{U}(0, n^{-1})$ and found that the behavior of the methods does not change significantly. As predicted by the analysis of the computational cost in section 5.5, `schur_full` is the only algorithm whose timings depend only on the order n of the input matrix but not on the rank k



(a) $n = 1000$. **(b)** $n = 4000$.

(c) $n = 7000$. **(d)** $n = 10000$.

Figure 5.5: Execution times (in seconds) of algorithms for computing the square root. The matrix A has the form (5.1) for $\alpha = 0.1$ and $V \neq U$. The elements of U and V are drawn from $\mathcal{N}(0, n^{-2})$.

of the perturbation. The methods that exploit the structure of A, on the other hand, become slower as the ratio k/n grows. The fastest of the four implementations is `schur_k`, and its execution time never exceeds that of `schur_full` significantly. In this experiment `db_prod_struct` typically requires 7 iterations to converge and is the slowest, and `db_prod_k` typically requires 6 iterations to converge and is just slightly slower than `schur_k`.

We repeated the experiments with $\alpha = 1$ and obtained very similar results, although on this simpler problem `db_prod_k` and `db_prod_struct` were usually faster and required at most 2 and 3 iterations, respectively.

Table 5.2: Characteristics of the test matrices provided as part of the Lingvo framework and our approximations to them. For the test matrices B_i we report the size, n, the smallest and the largest eigenvalues $\lambda_{\min}$ and $\lambda_{\max}$ computed by the MATLAB `eig` function using binary32 arithmetic, and the numerical rank r as returned by the MATLAB function `rank`. For the approximations $\widetilde{B}_i$ we report the order t_i of Σ_t in the truncated spectral decomposition with tolerance ε_i, as discussed in section 5.6.3.

	n	$\lambda_{\min}$	$\lambda_{\max}$	r		ε_1	t_1	ε_2	t_2
B_1	1024	-1.3×10^{-2}	7.8×10^{4}	20	$\widetilde{B}_1$	1.0×10^{-1}	82	2.0×10^{-3}	128
B_2	512	-2.4×10^{-4}	5.6×10^{3}	173	$\widetilde{B}_2$	1.0×10^{-1}	221	6.9×10^{-4}	418
B_3	512	-1.9×10^{-4}	1.8×10^{3}	243	$\widetilde{B}_3$	1.0×10^{-1}	177	6.9×10^{-4}	511

The picture is different for nonsymmetric matrices, as shown by Figure 5.5, which reports the results of the same experiments for the matrix A of the form (5.1) with $n = 1000$, 4000, 7000, and 10000, $\alpha = 0.1$, and $U \neq V$. The behavior of the algorithms does not change significantly in this setting, although `db_prod_k` becomes the fastest method for all matrix sizes, `schur_k` becomes the slowest by a considerable margin for $n = 1000$, whereas `db_prod_struct` is typically the slowest for larger matrices.

We remark that in both cases, the new algorithms we discuss can be up to two orders of magnitude faster than the traditional approach based on the Schur decomposition, when the ratio k/n is below 1/10, which can be considered the typical range for low-rank updates.

5.6.3 Positive definite matrices from applications

Now we compare the structured iterations and the direct algorithms on three test matrices from machine learning applications [1]. These are provided as part of the Lingvo framework for TensorFlow [29] and are available on GitHub.[4] The matrices, which are provided in binary32 format, are formed by accumulating matrix products of the form (5.10) for $\alpha = 0$, and thus they are real and symmetric but are numerically indefinite because of rounding errors in the computation, having small negative real eigenvalues (see Table 5.2). We do not have access to the terms G_s in (5.10) used to generate the test matrices, and for the sake of our experiment we recover them from the test matrices as we now explain.

4 https://github.com/tensorflow/lingvo/tree/master/lingvo/core/testdata

By the spectral theorem, the symmetric test matrix $B_i \in \mathbb{R}^{n\times n}$ can be decomposed as $Q\Sigma Q^T$, where $Q \in \mathbb{R}^{n\times n}$ is orthogonal and $\Sigma \in \mathbb{R}^{n\times n}$ is diagonal and has diagonal elements sorted in decreasing order. Let us now define the matrix $\widetilde{B}_i = Q_t\Sigma_t Q_t^T$, where $Q_t \in \mathbb{R}^{n\times t}$ collects the first t columns of Q and $\Sigma_t \in \mathbb{R}^{t\times t}$ is the leading principal submatrix minor of Σ of order t. In other words, $\widetilde{B}_i$ approximates B_i by truncating its spectral decomposition to rank t. By taking $G = Q_t{\Sigma_t}^{1/2}$, we can rewrite this approximation as $GG^T = \widetilde{B}_i \approx B_i$, which is implicitly of the form $\sum_{s=1}^{t} G_s G_s^T$ in (5.10). We choose t according to some tolerance $\varepsilon > 0$ such that all eigenvalues of $B_i \in \mathbb{R}^{n\times n}$ not less than ε are retained in $\Sigma_t \in \mathbb{R}^{t\times t}$. In the experiments we consider two different choices of the tolerance: $\varepsilon_1 = 0.1$ and $\varepsilon_2 = n^{3/2}u_{32}$.

In Table 5.2 we list some important characteristics of the original test matrices, which we denote by B_1, B_2, and B_3, and our approximations to them, which we denote by $\widetilde{B}_1$, $\widetilde{B}_2$, and $\widetilde{B}_3$, respectively.

We examine the performance of the algorithms for computing the principal square root of $A_i = \alpha I_n + \widetilde{B}_i$ in binary32 arithmetic, where α is a positive real constant chosen so that the smallest eigenvalue of A_i is positive, which implies that A_i is positive definite. Given that practical values of the regularizing scalar α are not mentioned in [1], in the experiments we test three choices: $\alpha = 10^{-6}$, 10^{-3}, and 1.

The results are given in Table 5.3, and Figure 5.6 presents the same data pictorially. The matrices do not appear in the same order in the two panels of Figure 5.6: they are grouped by value of α in Figure 5.6a and by size and rank in Figure 5.6b.

All the methods except `db_prod_struct` converge for all test matrices with relative residual of the order $\alpha_2(A^{1/2})u_{32}$ in most cases, which indicates good numerical stability. In general, `db_prod_k` gives the solution that has the smallest residual; the other iterative method, `db_prod_struct` computes an unsatisfactory solution for $\alpha = 10^{-6}$ and $\alpha = 10^{-3}$, but for $\alpha = 1$ its performance is on par with that of `db_prod_k`.

In terms of timings, `schur_full` is by far the slowest choice for B_1, but becomes comparable with `db_prod_k` and `db_prod_struct` for B_2 and B_3 when the rank of $\widetilde{B}_i$ is moderate compared with the size. `schur_k` is the fastest method, while its

Table 5.3: Relative residual and execution time (in seconds) of algorithms for computing the square root. The matrices are those in Table 5.2.

	t	Method	$\alpha = 10^{-6}$		$\alpha = 10^{-3}$		$\alpha = 1$	
			res	time	res	time	res	time
B_1	82	`schur_full`	1×10^{-6}	5×10^{-2}	1×10^{-6}	5×10^{-2}	2×10^{-6}	4×10^{-2}
		`schur_k`	9×10^{-7}	2×10^{-3}	9×10^{-7}	1×10^{-3}	1×10^{-6}	1×10^{-3}
		`db_prod_k`	4×10^{-7}	6×10^{-3}	3×10^{-7}	6×10^{-3}	5×10^{-7}	5×10^{-3}
		`db_prod_struct`	1×10^{-1}	5×10^{-3}	1×10^{-4}	3×10^{-3}	2×10^{-7}	3×10^{-3}
	128	`schur_full`	2×10^{-6}	4×10^{-2}	2×10^{-6}	4×10^{-2}	2×10^{-6}	4×10^{-2}
		`schur_k`	2×10^{-6}	2×10^{-3}	9×10^{-7}	2×10^{-3}	2×10^{-6}	2×10^{-3}
		`db_prod_k`	5×10^{-7}	1×10^{-2}	4×10^{-7}	1×10^{-2}	5×10^{-7}	1×10^{-2}
		`db_prod_struct`	1×10^{-1}	8×10^{-3}	1×10^{-4}	7×10^{-3}	4×10^{-7}	1×10^{-2}
B_2	221	`schur_full`	2×10^{-6}	9×10^{-3}	1×10^{-6}	8×10^{-3}	1×10^{-6}	8×10^{-3}
		`schur_k`	1×10^{-6}	3×10^{-3}	2×10^{-6}	3×10^{-3}	9×10^{-7}	3×10^{-3}
		`db_prod_k`	4×10^{-7}	2×10^{-2}	8×10^{-8}	2×10^{-2}	4×10^{-7}	2×10^{-2}
		`db_prod_struct`	1×10^{-1}	2×10^{-2}	1×10^{-4}	1×10^{-2}	5×10^{-7}	1×10^{-2}
	418	`schur_full`	2×10^{-6}	9×10^{-3}	2×10^{-6}	8×10^{-3}	2×10^{-6}	8×10^{-3}
		`schur_k`	7×10^{-6}	8×10^{-3}	6×10^{-6}	8×10^{-3}	6×10^{-6}	8×10^{-3}
		`db_prod_k`	4×10^{-7}	4×10^{-2}	7×10^{-8}	4×10^{-2}	4×10^{-7}	4×10^{-2}
		`db_prod_struct`	1×10^{-1}	6×10^{-2}	1×10^{-4}	4×10^{-2}	5×10^{-7}	4×10^{-2}
B_3	177	`schur_full`	2×10^{-6}	9×10^{-3}	1×10^{-6}	8×10^{-3}	3×10^{-6}	8×10^{-3}
		`schur_k`	1×10^{-6}	2×10^{-3}	1×10^{-6}	2×10^{-3}	7×10^{-7}	2×10^{-3}
		`db_prod_k`	3×10^{-7}	1×10^{-2}	1×10^{-7}	1×10^{-2}	2×10^{-7}	1×10^{-2}
		`db_prod_struct`	1×10^{-1}	1×10^{-2}	1×10^{-4}	9×10^{-3}	3×10^{-7}	9×10^{-3}
	511	`schur_full`	2×10^{-6}	8×10^{-3}	2×10^{-6}	9×10^{-3}	3×10^{-6}	8×10^{-3}
		`schur_k`	3×10^{-6}	1×10^{-2}	3×10^{-6}	1×10^{-2}	1×10^{-6}	1×10^{-2}
		`db_prod_k`	3×10^{-7}	8×10^{-2}	1×10^{-7}	8×10^{-2}	2×10^{-7}	8×10^{-2}
		`db_prod_struct`	1×10^{-1}	1×10^{-1}	1×10^{-4}	6×10^{-2}	3×10^{-7}	6×10^{-2}

advantage over `db_prod_k` and `db_prod_struct` becomes negligible when $\widetilde{B}_i$ has low rank. The execution time of the two iterative methods `db_prod_k` and `db_prod_struct` is similar on most of the test matrices. Again, we observe that exploiting the structure of A delivers a significant performance improvement when $k \ll n$, in line with what suggested by the cost comparison in Table 5.1.

We conclude by mentioning that we derived, implemented, and tested the structured version of other variants of the Newton iteration, including the incremental form of Iannazzo [21] and the Newton–Schulz iteration. We found that their performance is similar to that of the structured DB iteration in product form.

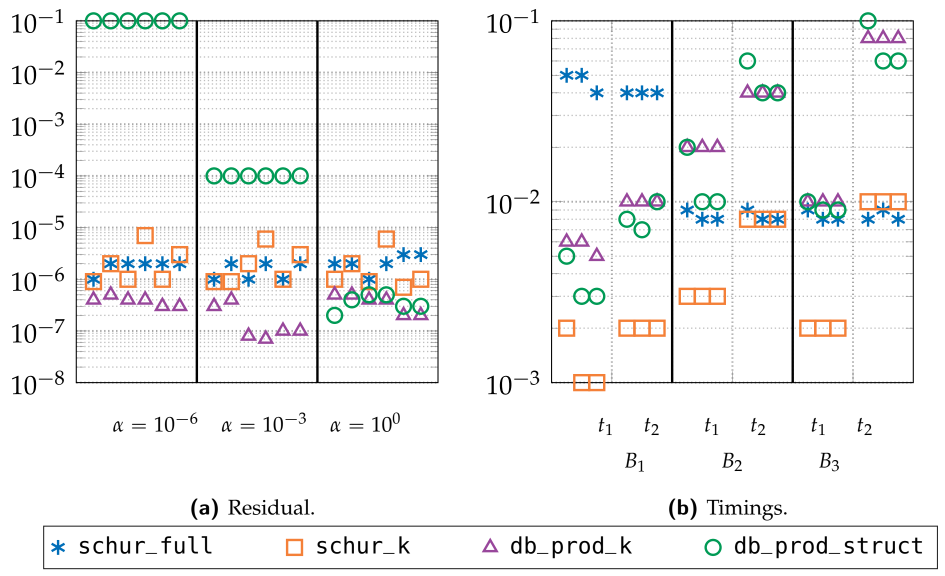


Figure 5.6: Relative residual (left) and execution time in seconds (right) of algorithms for computing the square root. The matrices are those in Table 5.2; they are grouped by value of α in the plot on the left, and by matrix in the plot on the right, where the two values of t for a matrix B_i are separated by a dotted line.

5.7 CONCLUDING REMARKS

We have investigated numerical methods for computing roots of a matrix $A = \alpha I_n + UV^*$, where U and V have rank $k \leqslant n$. We derived a new formula for $A^{1/p}$ that has the advantage over the existing formula from Theorem 5.1 of not requiring that V^*U be nonsingular. Focusing on the square root, we have also derived a new structured DB iteration that exploits the low-rank structure of UV^*.

Our numerical experiments confirm that when $k \ll n$, exploiting the structure yields algorithms that are much more efficient than simply applying the Schur method to A. If the Schur decomposition can be computed then using the Schur method to evaluate (5.9) is our preferred method overall. Otherwise, we recommend the use of the DB iteration, either in its structured form or as an unstructured algorithm to compute the $k \times k$ square root appearing in (5.9).

REFERENCES

[1] R. Anil, V. Gupta, T. Koren, K. Regan, and Y. Singer. *Scalable Second Order Optimization for Deep Learning*. preprint, arXiv:2002.09018v2 [cs.LG]. 2020. Revised March 2021 (cited on pp. 127, 147, 148).

[2] B. Beckermann, A. Cortinovis, D. Kressner, and M. Schweitzer. "Low-rank updates of matrix functions II: Rational Krylov methods." *SIAM J. Numer. Anal.* 59.3 (Jan. 2021), pp. 1325–1347 (cited on p. 126).

[3] B. Beckermann, D. Kressner, and M. Schweitzer. "Low-rank updates of matrix functions." *SIAM J. Matrix Anal. Appl.* 39.1 (Jan. 2018), pp. 539–565 (cited on p. 126).

[4] D. S. Bernstein and C. F. V. Loan. "Rational matrix functions and rank-1 updates." *SIAM J. Matrix Anal. Appl.* 22.1 (2000), pp. 145–154 (cited on p. 126).

[5] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. "Julia: A fresh approach to numerical computing." *SIAM Rev.* 59.1 (2017), pp. 65–98 (cited on p. 121).

[6] Å. Björck and S. Hammarling. "A Schur method for the square root of a matrix." *Linear Algebra Appl.* 52/53 (1983), pp. 127–140 (cited on pp. 121, 133, 135, 139, 141, 142).

[7] S. H. Cheng, N. J. Higham, C. S. Kenney, and A. J. Laub. "Approximating the logarithm of a matrix to specified accuracy." *SIAM J. Matrix Anal. Appl.* 22.4 (2001), pp. 1112–1125 (cited on p. 131).

[8] E. Deadman, N. J. Higham, and R. Ralha. "Blocked Schur Algorithms for Computing the Matrix Square Root." In: *Applied Parallel and Scientific Computing: 11th International Conference, PARA 2012, Helsinki, Finland*. Ed. by P. Manninen and P. Öster. Vol. 7782. Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 2013, pp. 171–182 (cited on pp. 121, 141).

[9] E. D. Denman and A. N. Beavers Jr. "The matrix sign function and computations in systems." *Appl. Math. Comput.* 2.1 (1976), pp. 63–94 (cited on p. 129).

[10] M. Fasi and N. J. Higham. "Multiprecision algorithms for computing the matrix logarithm." *SIAM J. Matrix Anal. Appl.* 39.1 (2018), pp. 472–491 (cited on p. 121).

[11] G. H. Golub and C. F. Van Loan. *Matrix Computations*. 4th ed., Baltimore, MD, USA: Johns Hopkins University Press, 2013, pp. xxi+756 (cited on pp. 121, 134, 135, 137–139).

[12] F. Greco and B. Iannazzo. "A binary powering Schur algorithm for computing primary matrix roots." *Numer. Algorithms* 55.1 (2010), pp. 59–78 (cited on p. 121).

[13] C.-H. Guo and N. J. Higham. "A Schur–Newton method for the matrix pth root and its inverse." *SIAM J. Matrix Anal. Appl.* 28.3 (2006), pp. 788–804 (cited on p. 127).

[14] V. Gupta, T. Koren, and Y. Singer. "Shampoo: Preconditioned Stochastic Tensor Optimization." In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by J. Dy and A. Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden, 2018, pp. 1842–1850 (cited on p. 127).

[15] L. A. Harris. "Computation of functions of certain operator matrices." *Linear Algebra Appl.* 194 (1993), pp. 31–34 (cited on p. 122).

[16] N. J. Higham. "Computing real square roots of a real matrix." *Linear Algebra Appl.* 88/89 (1987), pp. 405–430 (cited on pp. 121, 139, 141, 142).

[17] N. J. Higham. "Computing the polar decomposition—with applications." *SIAM J. Sci. Statist. Comput.* 7.4 (Oct. 1986), pp. 1160–1174 (cited on p. 133).

[18] N. J. Higham. *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008, pp. xx+425 (cited on pp. 120–122, 129, 131, 135, 136).

[19] N. J. Higham. "Newton's method for the matrix square root." *Math. Comp.* 46.174 (Apr. 1986), pp. 537–549 (cited on p. 129).

[20] R. A. Horn and C. R. Johnson. *Matrix Analysis*. 2nd. Cambridge, UK: Cambridge University Press, 2013, pp. xviii+643 (cited on pp. 122, 136).

[21] B. Iannazzo. "A note on computing the matrix square root." *Calcolo* 40.4 (2003), pp. 273–283 (cited on p. 149).

[22] B. Iannazzo and C. Manasse. "A Schur logarithmic algorithm for fractional powers of matrices." *SIAM J. Matrix Anal. Appl.* 34.2 (2013), pp. 794–813 (cited on p. 121).

[23] C. Ionescu, O. Vantzos, and C. Sminchisescu. "Matrix backpropagation for deep networks with structured layers." In: *Proceedings of the IEEE International Conference on Computer Vision*. Dec. 2015, pp. 2965–2973 (cited on p. 129).

[24] T.-Y. Lin. "Higher-Order Representations for Visual Recognition." PhD thesis. Massachusetts, USA: College of Information and Computer Sciences, University of Massachusetts Amherst, Feb. 2020, p. 105 (cited on pp. 127, 128).

[25] *Multiprecision Computing Toolbox*. Advanpix, Tokyo, Japan. `http://www.advanpix.com` (cited on p. 124).

[26] Y. Nakatsukasa. *Fast and Stable Randomized Low-rank Matrix Approximation*. preprint, arXiv:2009.11392 [math.NA]. 2020 (cited on p. 128).

[27] G. Schulz. "Iterative Berechung der reziproken Matrix." *Z. Angew. Math. Mech.* 13.1 (1933), pp. 57–59 (cited on p. 121).

[28] W. Shao, H. Yu, Z. Zhang, H. Xu, Z. Li, and P. Luo. *BWCP: Probabilistic Learning-to-Prune Channels for ConvNets via Batch Whitening*. preprint, arXiv:2105.06423 [cs.LG]. 2021 (cited on p. 129).

[29] J. Shen, P. Nguyen, Y. Wu, et al. *Lingvo: A Modular and Scalable Framework for Sequence-to-Sequence Modeling*. preprint, arXiv:1902.08295 [cs.LG]. 2019 (cited on p. 147).

[30] S. Shumeli, P. Drineas, and H. Avron. *Low-Rank Updates of Matrix Square Roots*. arXiv:2201.13156 [math.NA]. 2022 (cited on p. 127).

[31] M. I. Smith. "A Schur algorithm for computing matrix pth roots." *SIAM J. Matrix Anal. Appl.* 24.4 (2003), pp. 971–989 (cited on p. 121).

[32] Y. Song, N. Sebe, and W. Wang. *Fast Differentiable Matrix Square Root*. ArXiv:1209.5145. 2022 (cited on p. 121).

[33] *Symbolic Math Toolbox*. The MathWorks, Inc., Natick, MA, USA. `http://www.mathworks.co.uk/products/symbolic/` (cited on p. 121).

[34] D. S. Watkins. "Francis's algorithm." *Amer. Math. Monthly* 118.5 (2011), p. 387 (cited on p. 121).

[35] D. S. Watkins. "Understanding the *QR* algorithm." *SIAM Rev.* 24.4 (Oct. 1982), pp. 427–440 (cited on p. 121).

[36] D. S. Watkins. "The *QR* algorithm revisited." *SIAM Rev.* 50.1 (Jan. 2008), pp. 133–145 (cited on p. 121).

[37] C. Ye, X. Zhou, T. McKinney, Y. Liu, Q. Zhou, and F. Zhdanov. *Exploiting Invariance in Training Deep Neural Networks*. preprint, arXiv:2103.16634v2 [cs.CV]. 2021. Revised December 2021 (cited on p. 129).

6 CONCLUSIONS

The computation of functions of matrices has drawn much research interest over the decades, and various algorithms for computing different matrix functions have been proposed in the literature. Many existing state-of-the-art algorithms for computing matrix functions are tightly coupled to a specific precision of floating-point arithmetic and are not suited for an arbitrary precision setting, because their algorithmic design requires potentially expensive precision-dependent computations. In this thesis we developed numerical methods for computing matrix functions in arbitrary precision, which take the unit roundoff of the working precision as an input argument to reduce the dependence on characteristics of the computational environment, and so works in an arbitrary precision.

We exploited Davies's idea of randomized approximate diagonalization [2] within the Schur–Parlett framework [3] to build a multiprecision derivative–free algorithm for evaluating analytic matrix functions in arbitrary precision. The key idea is to add random diagonal perturbations to nontrivial blocks on the diagonal of the Schur form and then diagonalize the perturbed blocks in a precision higher than the working precision. An estimate of the condition number of the eigenvector matrix of the perturbed blocks is needed in order to determine the higher precision at which to perform the diagonalization. We therefore designed a scheme to estimate the condition number of the eigenvector matrix of a triangular matrix based only on its elements. This algorithm greatly expands the class of readily computable matrix functions, given that we have access to higher precision arithmetic which depends on the eigenvalue distribution of the matrix.

Experiments suggest that the condition numbers of the matrix square root, the sign function, and the logarithm at a triangular matrix in the direction of a full matrix can be significantly larger than that in the direction of a triangular matrix. This huge difference in the unstructured and structured conditioning of these functions at triangular matrices could be investigated in future research.

We developed a scaling and recovering algorithm for computing the matrix cosine in arbitrary precision. The fact that the computational environment is only known at the run time brings uncertainty and flexibility in selecting the parameters in the algorithm. We overcame this difficulty by combining new inexpensively computable forward error bounds from Taylor approximation that exploit the hyperbolic cosine with dynamic strategies for selecting the algorithmic parameters, to design a viable algorithm in arbitrary precision, where, in principle, the number of scalings and the degree of the approximant chosen by the algorithm can be arbitrarily large. We also derived a framework for computing the cosine and its Fréchet derivative simultaneously, where the main point is Fréchet differentiating the double angle formula $\cos(2A) = 2\cos^2 A - I$, and we built an efficient evaluation scheme based on the Paterson–Stockmeyer method for computing them simultaneously in arbitrary precision. This is the first algorithm that can compute in arbitrary precision a matrix function and its Fréchet derivative simultaneously. We finally showed how this scheme can be extended to evaluate the matrix sine, cosine, and their Fréchet derivatives all together.

The analysis and techniques in the algorithm can be readily adapted for computing other matrix trigonometric and hyperbolic functions in arbitrary precision arithmetic, such as those treated in [1] and the wave-kernel functions investigated in [6] and their Fréchet derivatives. Another possible future direction is to extend our algorithm to compute the action of these functions on a matrix in arbitrary precision as it is actually the matrix–vector products that are required in the solutions of wave equations.

We studied numerical methods for computing roots of a matrix $A = \alpha I + UV^*$, as a correction of $\alpha^{1/2}I$, where U and V have rank $k \leqslant n$, and derived a new formula for $A^{1/p}$ that has the advantage over the existing formula from [5, Thm. 1.35] of not requiring that V^*U be nonsingular; focusing on the square root, we derived a new class of Newton iterations that exploits the structure of A by simultaneously iterating on a scalar and a $k \times k$ matrix. We also proposed several Schur-based methods that can utilize the structure of A. These methods can be employed in arbitrary precision by simply executing all elementary scalar operations in arbitrary precision, and, additionally, for the iterative algorithms, by properly adjusting the internal tolerance that is used as stopping criterion.

In particular, within the Schur-based methods we established a scheme to obtain the Schur decomposition of the $n \times n$ matrix UV^* from the Schur decomposition of the $k \times k$ matrix V^*U, and this scheme may be exploited to derive an efficient Schur-based algorithm for computing matrix functions of a low-rank matrix A, in the combination with randomized algorithms such as [4] that construct a low-rank factorization of A. The study of this algorithm remains the subject of future work.

REFERENCES

[1] A. H. Al-Mohy. "A truncated Taylor series algorithm for computing the action of trigonometric and hyperbolic matrix functions." *SIAM J. Sci. Comput.* 40.3 (2018), A1696–A1713 (cited on p. 156).

[2] E. B. Davies. "Approximate diagonalization." *SIAM J. Matrix Anal. Appl.* 29.4 (2007), pp. 1051–1064 (cited on p. 155).

[3] P. I. Davies and N. J. Higham. "A Schur–Parlett algorithm for computing matrix functions." *SIAM J. Matrix Anal. Appl.* 25.2 (2003), pp. 464–485 (cited on p. 155).

[4] N. Halko, P.-G. Martinsson, and J. A. Tropp. "Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions." *SIAM Rev.* 53.2 (2011), pp. 217–288 (cited on p. 157).

[5] N. J. Higham. *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008, pp. xx+425 (cited on p. 156).

[6] P. Nadukandi and N. J. Higham. "Computing the wave-kernel matrix functions." *SIAM J. Sci. Comput.* 40.6 (2018), A4060–A4082 (cited on p. 156).