Folkmar Bornemann

Numerical Linear Algebra

A Concise Introduction with MATLAB and Julia

Translated by Walter Simson

Springer

Folkmar Bornemann
Center for Mathematical Sciences
Technical University of Munich
Garching, Germany

Translated by Walter Simson, Munich, Germany

ISSN 1615-2085 ISSN 2197-4144 (electronic)
Springer Undergraduate Mathematics Series
ISBN 978-3-319-74221-2 ISBN 978-3-319-74222-9 (eBook)
https://doi.org/978-3-319-74222-9

Library of Congress Control Number: 2018930015

Preface

This book was developed from the lecture notes of an undergraduate level course for students of mathematics at the Technical University of Munich, consisting of two lectures per week. Its goal is to present and pass on a skill set of algorithmic and numerical thinking based on the fundamental problem set of numerical linear algebra. Limiting the scope to linear algebra creates a stronger thematic coherency in the course material than would be found in other introductory courses on numerical analysis. Beyond the didactic advantages, numerical linear algebra represents the basis for the field of numerical analysis, and should therefore be learned and mastered as early as possible.

This exposition will emphasize the viability of partitioning vectors and matrices block-wise as compared to a more classic, component-by-component approach. In this way, we achieve not only a more lucid notation and shorter algorithms, but also a significant improvement in the execution time of calculations thanks to the ubiquity of modern vector processors and hierarchical memory architectures.

The motto for this book will therefore be:

A higher level of abstraction is actually an asset.

During our discussion of *error analysis* we will be diving uncompromisingly deep into the relevant concepts in order to gain the greatest possible understanding of assessing numerical processes. More shallow approaches (e.g., the infamous "rules of thumb") only result in unreliable, expensive and sometimes downright dangerous outcomes.

The algorithms and accompanying numerical examples will be provided in the programming environment MATLAB, which is near to ubiquitous at universities around the world. Additionally, one can find the same examples programmed in the trailblazing numerical language Julia from MIT in Appendix B. My hope is that the following passages will not only pass on the intended knowledge, but also inspire further computer experiments.

The accompanying e-book offers the ability to click through links in the passages. Links to references elsewhere in the book will appear blue, while external links will appear red. The latter will lead to explanations of terms and nomenclature which are assumed to be previous knowledge, or to supplementary material such as web-based computations, historical information and sources of the references.

Munich, October 2017

Folkmar Bornemann
bornemann@tum.de

Student's Laboratory

In order to enhance the learning experience when reading this book, I recommend creating one's own laboratory, and outfitting it with the following "tools".

Tool 1: Programming Environment Due to the current prevalence of the numerical development environment MATLAB by MathWorks in both academic and industrial fields, we will be using it as our "go-to" scripting language in this book. Nevertheless, I would advise every reader to look into the programming language Julia from MIT. This ingenious, forward-looking language is growing popularity, making it a language to learn. All programming examples in this book have been rewritten in Julia in Appendix B.

Tool 2: Calculation Work Horse I will be devoting most of the pages of this book to the ideas and concepts of numerical linear algebra, and will avoid getting caught up with tedious calculations. Since many of these calculations are mechanical in nature, I encourage every reader to find a suitable "calculation work horse" to accomplish these tasks for them. Some convenient options include computer algebra systems such as Maple or Mathematica; the latter offers a free "one-liner" version online in the form of WolframAlpha. Several examples can be found as external links in §14.

Tool 3: Textbook X In order to gain further perspective on the subject matter at hand, I recommend always having a second opinion, or "Textbook X", within reach. Below I have listed a few excellent options:

- Peter Deuflhard, Andreas Hohmann: *Numerical Analysis in Modern Scientific Computing*, 2nd ed., Springer-Verlag, New York, 2003.
  A refreshing book; according to the preface, my youthful enthusiasm helped form the presentation of error analysis.
- Lloyd N. Trefethen, David Bau: *Numerical Linear Algebra*, Society of Industrial and Applied Mathematics, Philadelphia, 1997.
  A classic and an all time bestseller of the publisher, written in a lively voice.
- James W. Demmel: *Applied Numerical Linear Algebra*, Society of Industrial and Applied Mathematics, Philadelphia, 1997.
  Deeper and more detailed than Trefethen–Bau, a classic as well.

Tool 4: Reference Material For even greater immersion and as a starting point for further research, I strongly recommend the following works:

- Gene H. Golub, Charles F. Van Loan: *Matrix Computations*, 4th ed., The Johns Hopkins University Press, Baltimore, 2013.
  The "Bible" on the topic.
- Nicholas J. Higham: *Accuracy and Stability of Numerical Algorithms*, 2nd ed., Society of Industrial and Applied Mathematics, Philadelphia, 2002.
  The thorough modern standard reference for error analysis (without eigenvalue problems, though).
- Roger A. Horn, Charles R. Johnson: *Matrix Analysis*, 2nd ed., Cambridge University Press, Cambridge, 2012.
  A classic on the topic of matrix theory; very thorough and detailed, a must-have reference.

Contents

1 Computing with Matrices

The purpose of computing is insight, not numbers.

(Richard Hamming 1962)

Applied mathematics is not engineering.

(Paul Halmos 1981)

1 What is Numerical Analysis?

1.1 Numerical analysis describes the construction and analysis of *efficient* discrete algorithms to solve *continuous* problems using large amounts of data. Here, these terms are meant as follows:

- *efficient*: the austere use of "resources" such as computation time and computer memory;
- *continuous*: the scalar field is, as in mathematical analysis, $\mathbb{R}$ or $\mathbb{C}$.

1.2 There is, however, a fundamental discrepancy between the worlds of continuous and discrete mathematics: ultimately, the results of analytical limits require infinitely long computation time as well as infinite amounts of memory resources. In order to calculate the continuous problems efficiently, we have to suitably discretize continuous quantities, therefore making them finite. Our tools for this task are *machine numbers, iteration and approximation*. This means that we purposefully allow the numerical results to differ from the "exact" solutions of our imagination, while controlling their accuracy. The trick is to find skillful ways to work with a balanced and predictable set of imprecisions in order to achieve greater efficiency.

1.3 Numerical linear algebra is a fundamental discipline. The study of numerical linear algebra not only teaches the thought process required for numerical analysis, but also introduces the student to problems that are omnipresent in modern scientific computing. Were one unable to efficiently solve these fundamental problems, the quest for solving higher-ranking ones would be futile.

2 Matrix Calculus

2.1 The language of numerical linear algebra is matrix calculus in $\mathbb{R}$ and $\mathbb{C}$. We will see that this is not merely a conceptional notation, but also a great help when striving to find efficient algorithms. By thinking in vectors and matrices instead of arrays of numbers, we greatly improve the usability of the subject matter. To this end, we will review a few topics from linear algebra, and adapt perspective as well as notation to our purposes.

2.2 For ease of notation, we denote by $\mathbb{K}$ the field of either real numbers $\mathbb{R}$ or complex numbers $\mathbb{C}$. In both cases,

$$\mathbb{K}^{m\times n} = \text{ the vector space of } m \times n \text{ matrices.}$$

Furthermore, we identify $\mathbb{K}^m = \mathbb{K}^{m\times 1}$ as *column vectors*, and $\mathbb{K} = \mathbb{K}^{1\times 1}$ as scalars, thereby including both in our matrix definition. We will refer to matrices in $\mathbb{K}^{1\times m}$ as *co-vectors* or *row vectors*.

2.3 In order to ease the reading of matrix calculus expressions, we will agree upon the following concise *notation convention* for this book:

- $\alpha, \beta, \gamma, \ldots, \omega$: scalars
- $a, b, c, \ldots, z$: vectors (= column vectors)
- $a', b', c', \ldots, z'$: co-vectors (= row vectors)
- $A, B, C, \ldots, Z$: matrices.

Special cases include:

- k, j, l, m, n, p: indices and dimensions with values in $\mathbb{N}_0$.

Remark. The notation for row vectors is compatible with the notation from §2.5 for the adjunction of matrices and vectors. This will prove itself useful in the course of this book.

2.4 We can write a vector $x \in \mathbb{K}^m$ or a matrix $A \in \mathbb{K}^{m\times n}$ using their components in the following form:

$$x = \begin{pmatrix} \xi_1 \\ \vdots \\ \xi_m \end{pmatrix} = (\xi_j)_{j=1:m}, \qquad A = \begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1n} \\ \vdots & & \vdots \\ \alpha_{m1} & \cdots & \alpha_{mn} \end{pmatrix} = (\alpha_{jk})_{j=1:m,k=1:n}.$$

In this case, $j = 1 : m$ is the short form of $j = 1, 2, \ldots, m$. This *colon notation* is widely used in *numerical* linear algebra and is supported by numerical programming languages such as Fortran 90/95, MATLAB and Julia.

2.5 The *adjoint* matrix $A' \in \mathbb{K}^{n\times m}$ of the matrix $A \in \mathbb{K}^{m\times n}$ is defined as

$$A' = \begin{pmatrix} \alpha'_{11} & \cdots & \alpha'_{m1} \\ \vdots & & \vdots \\ \alpha'_{1n} & \cdots & \alpha'_{mn} \end{pmatrix} = (\alpha'_{jk})_{k=1:n,j=1:m}$$

with $\alpha' = \alpha$ for real scalars and $\alpha' = \overline{\alpha}$ for complex scalars.[1] In the real case, one refers to a *transposed* matrix, while in the complex case we use the term *Hermitian transposed* Matrix. This way, the row vector

$$x' = (\xi'_1, \ldots, \xi'_m)$$

is the adjoint co-vector of the vector $x = (\xi_j)_{j=1:m}$.

2.6 Instead of diassembling a matrix into its components, we will often *partition* matrices $A \in \mathbb{K}^{m\times n}$ into their column vectors $a^k \in \mathbb{K}^m$ $(k = 1 : n)$

$$A = \begin{pmatrix} | & & | \\ a^1 & \cdots & a^n \\ | & & | \end{pmatrix}$$

or row vectors $a'_j \in \mathbb{K}^{1\times n}$ $(j = 1 : m)$

$$A = \begin{pmatrix} — \, a'_1 \, — \\ \vdots \\ — \, a'_m \, — \end{pmatrix}.$$

The process of adjunction swaps rows and columns: the columns of A' are therefore the vectors a_j $(j = 1 : m)$, and the rows are the co-vectors $(a^k)'$ $(k = 1 : n)$.

Remark. Memory aid: the superscript indexes correspond to the "standing" vectors (column vectors), while the subscript indexes correspond to "lying" vectors (row vectors).

2.7 The standard basis of $\mathbb{K}^m$ consists of the *canonical unit vectors*

$$e^k = ([j = k])_{j=1:m} \qquad (k = 1 : m),$$

where we employ the practical syntax of the Iverson bracket:[2]

$$[\mathcal{A}] = \begin{cases} 1 & \text{if the Statement } \mathcal{A} \text{ is correct,} \\ 0 & \text{otherwise.} \end{cases}$$

[1] The $'$-notation for adjunction has been adopted from programming languages such as MATLAB and Julia, but can nevertheless also be found in the German functional analysis literature.

[2] The Iverson-bracket is much more versatile than the Kronecker delta and deserves much wider recognition. A multi-faceted and virtuous exposure can be found in the classic textbook R. Graham, D. Knuth, O. Patashnik, *Concrete Mathematics*, 2nd ed., Addison Wesley, Reading, 1994.

The column vectors a^k of a matrix A are *defined* as the image of the canonical unit vector e^k under the linear map induced by A:

$$a^k = Ae^k \qquad (k = 1 : n). \tag{2.1}$$

2.8 Linearity therefore ensures that the image of the vector $x = (\xi_k)_{k=1:n} = \sum_{k=1}^n \xi_k e^k \in \mathbb{K}^n$ is

$$\mathbb{K}^m \ni Ax = \sum_{k=1}^n \xi_k a^k;$$

instead of using the term image, we will mostly refer to the *matrix-vector product*. In the special case of a co-vector $y' = (\eta'_1, \dots, \eta'_n)$, we obtain

$$\mathbb{K} \ni y'x = \sum_{k=1}^n \eta'_k \xi_k; \tag{2.2}$$

this expression is called the *inner product* of the vectors y and x.[3] If we then continue to read the matrix vector product row-wise, we attain

$$Ax = \begin{pmatrix} a'_1 x \\ \vdots \\ a'_m x \end{pmatrix}.$$

2.9 The inner product of a vector $x \in \mathbb{K}^n$ with itself, in accordance with (2.2), fulfills

$$x'x = \sum_{k=1}^n \xi'_k \xi_k = \sum_{k=1}^n |\xi_k|^2 \geqslant 0.$$

We can directly see that $x'x = 0 \Leftrightarrow x = 0$. The *Euclidean norm* we are familiar with from calculus is defined as $\|x\|_2 = \sqrt{x'x}$.

2.10 The product $C = AB \in \mathbb{K}^{m\times p}$ of two matrices $A \in \mathbb{K}^{m\times n}$ and $B \in \mathbb{K}^{n\times p}$ is defined as the composition of the corresponding linear maps. It therefore follows

$$c^k = Ce^k = (AB)e^k = A(Be^k) = Ab^k \qquad (k = 1 : p),$$

thus with the results on the matrix-vector product

$$AB \overset{(a)}{=} \begin{pmatrix} | & & | \\ Ab^1 & \cdots & Ab^p \\ | & & | \end{pmatrix} \overset{(b)}{=} \begin{pmatrix} a'_1 b^1 & \cdots & a'_1 b^p \\ \vdots & & \vdots \\ a'_m b^1 & \cdots & a'_m b^p \end{pmatrix} \overset{(c)}{=} \begin{pmatrix} — \, a'_1 B \, — \\ \vdots \\ — \, a'_m B \, — \end{pmatrix}. \tag{2.3}$$

The last equality arises from the fact that we read the result row-wise. In particular, we see that the product Ax provides the same result, independent of whether we view x as a vector or an $n \times 1$-matrix (therefore being consistent with the identification agreed upon in §2.2).

[3] One often writes $y'x = y \cdot x$; therefore the secondary name *dot product*.

2.11 Since adjunction is a *linear* involution[4] on $\mathbb{K}$, the inner product (2.2) satisfies the relationship

$$(y'x)' = x'y.$$

The second formula in (2.3) leads directly to the adjunction rule

$$(AB)' = B'A'.$$

2.12 The case of $xy' \in K^{m\times n}$ is called *outer product* of the vectors $x \in \mathbb{K}^m$, $y \in \mathbb{K}^n$,

$$xy' = \begin{pmatrix} \xi_1\,\eta_1' & \cdots & \xi_1\,\eta_n' \\ \vdots & & \vdots \\ \xi_m\,\eta_1' & \cdots & \xi_m\,\eta_n' \end{pmatrix}.$$

The outer product of $x, y \neq 0$ is a rank-1 matrix, whose image is actually spanned by the vector x:

$$(xy')z = \underbrace{(y'z)}_{\in\mathbb{K}} \cdot x \qquad (z \in \mathbb{K}^n).$$

2.13 For $x \in \mathbb{K}^m$ we will define the associated *diagonal matrix* by[5]

$$\operatorname{diag}(x) = \begin{pmatrix} \xi_1 & & & \\ & \xi_2 & & \\ & & \ddots & \\ & & & \xi_m \end{pmatrix}.$$

Notice that $\operatorname{diag} : \mathbb{K}^m \to \mathbb{K}^{m\times m}$ is a linear map. With $\operatorname{diag}(e^k) = e^k \cdot e_k'$ (where we set $e_k = e^k$) we therefore obtain the basis representation

$$\operatorname{diag}(x) = \sum_{k=1}^{m} \xi_k\,(e^k \cdot e_k').$$

2.14 The unit matrix (*identity*) $I \in \mathbb{K}^{m\times m}$ is given by

$$I = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix} = \sum_{k=1}^{m} e^k \cdot e_k'. \tag{2.4}$$

[4] Involution on $\mathbb{K}$: $\xi'' = \xi$ for all $\xi \in \mathbb{K}$.

[5] For such a componentwise matrix notation, we will agree that, for the sake of clarity, zero entries of a matrix may simply be omitted from printing.

It fulfills $Ie_k = e_k$ $(k = 1 : m)$ and therefore (linearity) $Ix = x$ for all $x \in \mathbb{K}^m$; thus

$$x = \sum_{k=1}^{m} (e_k' x) \cdot e^k. \tag{2.5}$$

This means $\xi_k = e_k' x$ $(k = 1 : m)$. After adjusting the dimensions, it notably holds $AI = A$ and $IA = A$. Since the row vectors of I are the e_k', the third formula in (2.3) provides a formula dual to (2.1)

$$a_k' = e_k' A \qquad (k = 1 : m).$$

2.15 Due to $AB = AIB$ for $A \in \mathbb{K}^{m\times n}$ and $B \in \mathbb{K}^{n\times p}$, it follows from (2.4)

$$AB = \sum_{k=1}^{n} Ae^k \cdot e_k' B = \sum_{k=1}^{n} a^k \cdot b_k', \tag{2.6}$$

that is, a matrix product can be represented as sum of rank-1 matrices.

2.16 There are a total of four formulas in (2.3.a-c) and (2.6) for the product of two matrices. One should realize that *every one* of these formulas, when written out componentwise, delivers the same formula as one is familiar with from introductory linear algebra courses (the components of A, B and $C = AB$ thereby defined as α_{jk}, β_{kl}, γ_{jl}):

$$\gamma_{jl} = \sum_{k=1}^{n} \alpha_{jk}\beta_{kl} \qquad (j = 1 : m, l = 1 : p). \tag{2.7}$$

This componentwise formula is by far not as important as the others. Notice that we have, without much calculation and independent of this equation, *conceptually* derived the other formulas.

2.17 All of the expressions for $C = AB$ until now are actually *special cases* of a very general formula for the product of two *block matrices*:

Lemma. *If we partition $A \in \mathbb{K}^{m\times n}$ and $B \in \mathbb{K}^{n\times p}$ as block matrices*

$$A = \begin{pmatrix} A_{11} & \cdots & A_{1r} \\ \vdots & & \vdots \\ A_{q1} & \cdots & A_{qr} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & \cdots & B_{1s} \\ \vdots & & \vdots \\ B_{r1} & \cdots & B_{rs} \end{pmatrix}$$

with the submatrices $A_{jk} \in \mathbb{K}^{m_j\times n_k}$, $B_{kl} \in \mathbb{K}^{n_k\times p_l}$ where

$$m = \sum_{j=1}^{q} m_j, \qquad n = \sum_{k=1}^{r} n_k, \qquad p = \sum_{l=1}^{s} p_l,$$

then the product $C = AB \in \mathbb{K}^{m\times p}$ is partitioned into blocks as follows:

$$C = \begin{pmatrix} C_{11} & \cdots & C_{1s} \\ \vdots & & \vdots \\ C_{q1} & \cdots & C_{qs} \end{pmatrix} \quad \text{with} \quad C_{jl} = \sum_{k=1}^{r} A_{jk}B_{kl} \qquad (j = 1:q, l = 1:s). \tag{2.8}$$

Remark. A comparison of the product formulas (2.7) and (2.8) shows that with such block partitioning, one can *formally* perform calculations "as if" the blocks were scalars. In the case of block matrices, one must pay careful attention to the correct order of the factors: in contrast to the componentwise $\beta_{kl}\alpha_{jk}$, the blockwise expression $B_{kl}A_{jk}$ is actually almost always *false* and often *meaningless* since the dimensions do not necessarily match.

Proof. In order to prove (2.8) we will partition the vectors $x \in \mathbb{K}^n$ according to

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_r \end{pmatrix}, \qquad x_k \in \mathbb{K}^{n_k} \quad (k = 1:r),$$

and we consider the linear map $N_k \in \mathbb{K}^{n_k\times n}$ defined by

$$N_k x = x_k.$$

Left multiplication with N_k therefore achieves the selection of the block rows that belong to n_k; adjunction shows that right multiplication with N_k' delivers the corresponding block columns. Similarly, we define $M_j \in \mathbb{K}^{m_j\times m}$ and $P_l \in \mathbb{K}^{p_l\times p}$. It therefore holds that

$$C_{jl} = M_j C P_l', \qquad A_{jk} = M_j A N_k', \qquad B_{kl} = N_k B P_l'.$$

The asserted formula (2.8) for block multiplication is therefore equivalent to

$$M_j C P_l' = M_j A \left(\sum_{k=1}^{r} N_k' N_k \right) B P_l' \qquad (j = 1:q, l = 1:s)$$

and it thus follows from $C = AB$ if only

$$\sum_{k=1}^{r} N_k' N_k = I \in \mathbb{K}^{n\times n}. \tag{*}$$

For $x, y \in \mathbb{K}^n$ with block parts $x_k = N_k x$, $y_k = N_k y$ this equality is successfully "tested" by executing the sum for the inner product *blockwise*,

$$x' I y = x' y = \sum_{k=1}^{r} x_k' y_k = x' \left(\sum_{k=1}^{r} N_k' N_k \right) y.$$

If we now let x and y traverse the standard basis of $\mathbb{K}^n$, we finally obtain (*). □

Exercise. Show that the five formulas in (2.3), (2.6), (2.7) as well as $C = AB$ itself are special cases of the block product (2.8). Visualize the corresponding block partitioning.

3 MATLAB

3.1 MATLAB (MATrix LABoratory) is the brand name of a commercial software that is prominent in both in academic and industrial settings and widely used for numerical simulations, data acquisition, and data analysis.[6] By means of a simple scripting language, MATLAB offers an elegant interface to matrix-based numerical analysis, as well as state of the art performance by levering the BLAS Library (Basic Linear Algebra Subprograms) from the processor manufacturer along with the high-performance Fortran library LAPACK for linear systems of equations, normal forms and eigenvalue problems.

MATLAB has become a standard skill that is currently required of every mathematician and engineer. Learning this skill though concrete practical problems is strongly recommended, which is why we will be programming all of the algorithms in the book in MATLAB. A very short introduction (for readers with a little previous programming experience) can be found in Appendix A.

3.2 In MATLAB scalars and vectors are both matrices; the identifications

$$\mathbb{K}^m = \mathbb{K}^{m\times 1}, \qquad \mathbb{K} = \mathbb{K}^{1\times 1},$$

are therefore design principles of the language. The fundamental operations of matrix calculus are:

meaning	formula	MATLAB
component of x	ξ_k	`x(k)`
component of A	α_{jk}	`A(j,k)`
column vector of A	a^k	`A(:,k)`
row vector of A	a'_j	`A(j,:)`
submatrix of A	$(\alpha_{jk})_{j=m:p,k=n:l}$	`A(m:p,n:l)`
adjoint matrix of A	A'	`A'`
matrix product	AB	`A*B`
identity matrix	$I \in \mathbb{K}^{m\times m}$	`eye(m)`
null matrix	$0 \in \mathbb{K}^{m\times n}$	`zeros(m,n)`

[6] Current open-source-alternatives include Julia, which is extensively discussed in Appendix B, and the Python libraries `NumPy` and `SciPy`, see H. P. Langtangen: *A Primer on Scientific Programming with Python*, 5th ed., Springer-Verlag, Berlin, 2016.

3.3 In order to practice, let us write all five formulas (2.3.a-c), (2.6) and (2.7) for the product $C = AB$ of the matrices $A \in \mathbb{K}^{m\times n}$, $B \in \mathbb{K}^{n\times p}$ as a MATLAB program:

Program 1 (Matrix Product: column-wise).

```
C = zeros(m,p);
for l=1:p
  C(:,l) = A*B(:,l);
end
```

$$C = \begin{pmatrix} | & & | \\ Ab^1 & \cdots & Ab^p \\ | & & | \end{pmatrix}$$

Program 2 (Matrix Product: row-wise).

```
C = zeros(m,p);
for j=1:m
  C(j,:) = A(j,:)*B;
end
```

$$C = \begin{pmatrix} — \, a_1' \, B \, — \\ \vdots \\ — \, a_m' \, B \, — \end{pmatrix}$$

Program 3 (Matrix Product: inner product).

```
C = zeros(m,p);
for j=1:m
  for l=1:p
    C(j,l) = A(j,:)*B(:,l);
  end
end
```

$$C = \begin{pmatrix} a_1' \, b^1 & \cdots & a_1' \, b^p \\ \vdots & & \vdots \\ a_m' \, b^1 & \cdots & a_m' \, b^p \end{pmatrix}$$

Program 4 (Matrix Product: outer product).

```
C = zeros(m,p);
for k=1:n
  C = C + A(:,k)*B(k,:);
end
```

$$C = \sum_{k=1}^{n} a^k \cdot b_k'$$

Program 5 (Matrix Product: componentwise).

```
C = zeros(m,p);
for j=1:m
  for l=1:p
    for k=1:n
      C(j,l) = C(j,l) + A(j,k)*B(k,l);
    end
  end
end
```

$$C = \left(\sum_{k=1}^{n} \alpha_{jk} \beta_{kl} \right)_{j=1:m, l=1:p}$$

3.4 When we apply these programs to two random matrices $A, B \in \mathbb{R}^{1000\times 1000}$, we measure the following execution times (in seconds):[7]

program	#`for`-loops	MATLAB [s]	C & BLAS [s]
`A*B`	0	0.031	0.029
column-wise	1	0.47	0.45
row-wise	1	0.52	0.49
outer product	1	3.5	0.75
inner product	2	1.7	1.5
componentwise	3	20	1.6

To compare with a compiled language (i.e., translated into machine code), we have also executed the same programs implemented in C while still using the same optimized Fortran BLAS routines used by MATLAB.

We can observe discrepancies in execution times of a factor of up to 600 in MATLAB code and up to 60 in C & BLAS, which we will try to explain in §4 in order to better understand where they come from before moving on to other numerical problems.

Remark. Check for yourself that all six programs really execute the *exact* same additions and multiplications, only in a different order.

3.5 We can already observe one design pattern in use: the fewer `for`-loops, the faster the program. If it is possible to express an algorithm with matrix-matrix operations, it is faster than just working with matrix-vector operations; matrix-vector operations on the other hand are advantageous when compared to componentwise operations: *a higher level of abstraction is actually an asset*.

4 Execution Times

4.1 Cost factors for the execution time of a program in numerical analysis include:

- floating point operations (i.e., the *real* arithmetic operations: $+, -, \cdot, /, \sqrt{}$)
- memory access
- overhead (unaccounted operations and memory access)

4.2 In an ideal world, the floating point operations (flop) would be the only cost factor of the execution time of a program and it would suffice to count them.[8]

[7] The execution times were measured on a MacBook Pro 13" with 3.0 GHz Intel Core i7 processor.

[8] For $\mathbb{K} = \mathbb{C}$ we must multiply the flop count of $\mathbb{K} = \mathbb{R}$ with an average factor of four: complex multiplication costs actually six real flop and complex addition two real flop.

problem	dimension	# flop	# flop ($m = n = p$)
$x'y$	$x, y \in \mathbb{R}^m$	$2m$	$2m$
xy'	$x \in \mathbb{R}^m, y \in \mathbb{R}^n$	mn	m^2
Ax	$A \in \mathbb{R}^{m\times n}, x \in \mathbb{R}^n$	$2mn$	$2m^2$
AB	$A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times p}$	$2mnp$	$2m^3$

As in this table, we will only consider the *leading* order for growing dimensions:

Example. As displayed in (2.2), the inner product in $\mathbb{R}^m$ requires m multiplications and $m-1$ additions, that is $2m-1$ operations in total; the leading order is $2m$.

4.3 If a single floating point operation costs us one unit of time t_{flop}, the peak execution time (*peak performance*) is

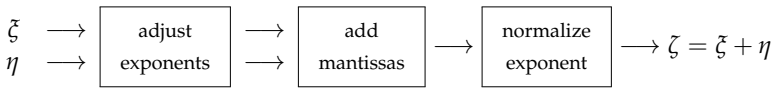
$$T_{\text{peak}} = \#\,\text{flop} \cdot t_{\text{flop}}$$

which defines an upper limit of algorithmic performance. The ambition of numerical mathematicians, computer scientists and computer manufacturers alike, at least for high-dimensional problems, is to come as close as possible to peak performance by minimizing the time needed for memory access and overhead.

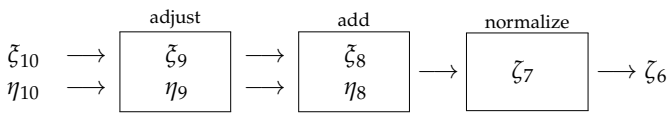
Example. The CPU used in §3.4 has a peak performance of approximately 72 Gflop per second. This means, $T_{\text{peak}} = 0.028\text{s}$ for $2 \cdot 10^9$ flop of the matrix multiplication where $m = 1000$; the actual execution time for program `A*B` was merely 4% slower.

4.4 Modern computer architectures use *pipelining in vector processors*. Let us examine the concepts with the help of the basic example of addition.

Example. In order to add two floating point numbers, we must *sequentially* execute the steps in the following scheme (cf. §11.6):



In this case, the processor needs 3 clock cycles for the addition. If we now add two m-dimensional floating-point vectors componentwise in this same way, we will need $3m$ clock cycles. Instead, let us now carry out the same operations using a kind of assembly line (pipeline). In this way, operations are executed *simultaneously*, and every work station completes its operation on the next component:



Thus, vectors of length m can be added in $m+2$ clock cycles and pipelining therefore accelerates the addition for large m by almost a factor of *three*.

4.5 In principle, memory access costs time. There are very fast memory chips, that are very costly, and slow memory chips that are more affordable. For this reason, modern computers work with a hierarchy of memories of different speeds that consist of a lot of inexpensive slow memory and a lesser amount of fast memory. Ordered from fast to slow, typical sizes for modern devices with a 3GHz CPU are:

memory type	typical size
CPU register	128 B
L1 cache	32 KiB
L2 cache	256 KiB
L3 cache	8 MiB
RAM	8 GiB
SSD	512 GiB
HD	3 TiB

The access speeds range from 20 GB/s to 100 MB/s, which correspond a difference of 2–3 in order of magnitude.

4.6 By cleverly coordinating the memory access pattern, one can ensure that a vector processor always has its next operand in the pipeline and only has to input and output that operand *once*. We will label the number of memory accesses to main memory (RAM) for input and output as # iop and the time needed for the access as t_{iop}. We can therefore write our *optimized* execution time

$$T = \#\,\text{flop} \cdot t_{\text{flop}} + \#\,\text{iops} \cdot t_{\text{iop}} = T_{\text{peak}} \left(1 + \frac{\tau}{q}\right),$$

where the machine dependent measure $\tau = t_{\text{iop}}/t_{\text{flop}}$ is $\tau \approx 30$ for modern computer architectures, and the algorithm dependent *efficiency ratio*

$$q = \frac{\#\,\text{flop}}{\#\,\text{iop}} = \text{flop per input-output operation.}$$

We therefore obtain in leading order (all dimensions $= m$):

	operation	# flop	# iop	q
$x'y$	inner product	$2m$	$2m$	1
$x\,y'$	outer product	m^2	m^2	1
Ax	matrix-vector product	$2m^2$	m^2	2
AB	matrix-matrix product	$2m^3$	$3m^2$	$2m/3$

Example. Now, we can *quantitatively* explain a number of differences from our table in §3.4: with $q = 2000/3$ and $\tau \approx 30$, the problem `A*B` should only be 4% slower than peak performance, which, according to §4.3, is sure enough the case. The row-wise and column-wise versions should be a factor of 15 slower, which has also been shown to be true. Furthermore, outer product calculations are slower by a factor of approximately 30 (which is the case for C & BLAS). MATLAB already experiences significant overhead for the outer product, and in general for calculations that only use vectors or components.

4.7 The BLAS Library was developed to *encapsulate* all hardware specific optimizations, and create a standard interface with which they could be accessed:[9]

- Level-1 BLAS (1973): vector operations, replacing *one* `for`-loop.
- Level-2 BLAS (1988): matrix-vector operations, replacing *two* `for`-loops.
- Level-3 BLAS (1990): matrix-matrix operations, replacing *three* `for`-loops.

Example. A selection of important BLAS routines ($q = \#\,\text{flop}/\#\,\text{iop}$):[10]

BLAS level	name	operation		q
1	`xAXPY`	$y \leftarrow \alpha x + y$	scaled addition	$2/3$
1	`xDOT`	$\alpha \leftarrow x'y$	inner product	1
2	`xGER`	$A \leftarrow \alpha xy' + A$	outer product	$3/2$
2	`xGEMV`	$y \leftarrow \alpha Ax + \beta y$	matrix-vector product	2
3	`xGEMM`	$C \leftarrow \alpha AB + \beta C$	matrix-matrix product	$m/2$

The notable efficiency gains of Level-3 BLAS are thanks to an efficiency ratio q that, as a matter of principle, scales linearly with the dimension m.

MATLAB includes optimized BLAS for all popular platforms and encapsulates it in a scripting language that is very close to mathematical syntax. When we look at code variations we should always formulate our algorithms using the highest BLAS level: *a higher level of abstraction is actually an asset*.

4.8 It is *not* coincidence that the row-wise multiplication from §3.4 is approximately 10% slower than the column-wise variant. The programming language

[9] Optimized implementations are delivered either by the CPU manufacturer or the ATLAS Project.

[10] The letter `x` in the names stands for either `S`, `D`, `C` or `Z`, meaning either single or double precision for both real and complex machine numbers (we will be discussing this topic in more detail in §12.4).

Fortran (and therefore BLAS, LAPACK and MATLAB) stores a matrix

$$A = \begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1n} \\ \vdots & & \vdots \\ \alpha_{m1} & \cdots & \alpha_{mn} \end{pmatrix}$$

column-wise (column-major order):

$\alpha_{11} \cdots \alpha_{m1}$	$\alpha_{12} \cdots \alpha_{m2}$	$\cdots$	$\alpha_{1n} \cdots \alpha_{mn}$

Therefore, unlike row-wise operations, operations on columns require no index calculations which would amount for further overhead; column-wise operations are therefore preferable. In contrast, C and Python store matrices *row-wise* (row-major order):

$\alpha_{11} \cdots \alpha_{1n}$	$\alpha_{21} \cdots \alpha_{2n}$	$\cdots$	$\alpha_{m1} \cdots \alpha_{mn}$

In this case, row-wise algorithms should be used.

4.9 The execution time of an algorithm is also influenced by the type of programming language used; the following factors play a role in slowing it down:[11]

- Assembler $\equiv$ Machine code *(hardware dependent)*: $1\times$
- Fortran, C, C++ $\xrightarrow{\text{Compiler}}$ machine code: $1\times - 2\times$
- MATLAB, Python, Julia $\xrightarrow[\text{Compiler}]{\text{JIT}}$ Bytecode for *virtual machine*: $1\times - 10\times$

Optimizing compilers replace vector operations with Level-1 BLAS routines.

5 Triangular Matrices

5.1 Triangular matrices are important building blocks of numerical linear algebra. We define *lower* and *upper* triangular matrices by the occupancy structure

$$L = \begin{pmatrix} * & & & \\ * & * & & \\ \vdots & \vdots & \ddots & \\ * & * & \cdots & * \end{pmatrix}, \qquad U = \begin{pmatrix} * & \cdots & * & * \\ & \ddots & \vdots & \vdots \\ & & * & * \\ & & & * \end{pmatrix}.$$

Components omitted from printing—as agreed upon in §2.13—represent zeros, '$*$' represent arbitrary elements in $\mathbb{K}$.

[11] JIT = just in time

5.2 With $V_k = \operatorname{span}\{e_1, \dots, e_k\} \subset \mathbb{K}^m$, one can characterize such matrices more abstractly: $U \in \mathbb{K}^{m\times m}$ is an upper triangular matrix if and only if[12]

$$UV_k = \operatorname{span}\{u^1, \dots, u^k\} \subset V_k \qquad (k = 1 : m),$$

and $L \in \mathbb{K}^{m\times m}$ is a lower triangular matrix if and only if

$$V_k' L = \operatorname{span}\{l_1', \dots, l_k'\} \subset V_k' \qquad (k = 1 : m).$$

Remark. Lower triangular matrices are the adjoints of upper triangular matrices.

5.3 Invertible lower (upper) triangular matrices are closed under multiplication and inversion. This means it holds:

Lemma. *Invertible lower (upper) triangular matrices in $\mathbb{K}^{m\times m}$ form a subgroup in $\mathrm{GL}(m;\mathbb{K})$ relative to matrix multiplication.*[13]

Proof. For an *invertible* upper triangular matrix, it holds that $UV_k \subset V_k$ and due to dimensional reasons even that $UV_k = V_k$. Therefore, $U^{-1}V_k = V_k$ $(k = 1 : m)$ and U^{-1} is upper triangular as well. Accordingly, if U_1, U_2 are upper triangular,

$$U_1U_2V_k \subset U_1V_k \subset V_k \qquad (k = 1 : m).$$

This means that U_1U_2 is also upper triangular. Through adjunction, we arrive at the assertion for lower triangular matrices. □

5.4 Building on Laplace's expansion it follows that

$$\det \begin{pmatrix} \lambda_1 & & & \\ * & \lambda_2 & & \\ \vdots & \vdots & \ddots & \\ * & * & \cdots & \lambda_m \end{pmatrix} = \lambda_1 \det \begin{pmatrix} \lambda_2 & & \\ \vdots & \ddots & \\ * & \cdots & \lambda_m \end{pmatrix} = \cdots = \lambda_1 \cdots \lambda_m;$$

that is, the determinant of a triangular matrix is the product of its diagonal entries:

A triangular matrix is invertible iff all its diagonal entries are non-zero.

Triangular matrices whose diagonals only exhibit the value 1 are called *unipotent*.

Exercise. Show that the unipotent lower (upper) triangular matrices form a subgroup of $\mathrm{GL}(m;\mathbb{K})$. *Hint:* Make use of the fact that unipotent upper triangular matrices U are characterized by

$$U = I + N, \qquad NV_k \subset V_{k-1} \quad (k = 1 : m).$$

Due to $N^m = 0$, such a matrix N is *nilpotent*. (Or, alternatively, prove the assertion through induction over the dimension by partitioning as in §5.5.)

[12] Here $u^1, \dots, u^m$ are the columns of U, $l_1', \dots, l_m'$ are the rows of L and $V_k' = \operatorname{span}\{e_1', \dots, e_k'\}$.

[13] The general linear group $\mathrm{GL}(m;\mathbb{K})$ consist of the *invertible* matrices in $\mathbb{K}^{m\times m}$ under matrix multiplication.

5.5 We now solve *linear systems of equations* with triangular matrices, such as

$$Lx = b$$

with a given vector b and an invertible lower triangular matrix L. We therefore set $L_m = L$, $b_m = b$ and $x_m = x$ and partition step by step according to

$$L_k = \left(\begin{array}{c|c} L_{k-1} & \\ \hline l'_{k-1} & \lambda_k \end{array}\right) \in \mathbb{K}^{k\times k}, \quad x_k = \left(\begin{array}{c} x_{k-1} \\ \hline \xi_k \end{array}\right) \in \mathbb{K}^k, \quad b_k = \left(\begin{array}{c} b_{k-1} \\ \hline \beta_k \end{array}\right) \in \mathbb{K}^k.$$

The two block-rows of the equation $L_k x_k = b_k$ can by expanded to

$$L_{k-1}x_{k-1} = b_{k-1}, \qquad l'_{k-1}x_{k-1} + \lambda_k \xi_k = \beta_k.$$

The first equality means only that we are consistent in our labeling; the second can be solved for ξ_k via the previous quantity x_{k-1} and thus provides the transition

$$x_{k-1} \mapsto x_k.$$

The process begins with an *empty*[14] vector x_0 (i.e., it does not actually appear in the partitioning of x_1) and leads to the solution of $x = x_m$ after m steps (dividing by λ_k is allowed since L is invertible):

$$\xi_k = (\beta_k - l'_{k-1}x_{k-1})/\lambda_k \qquad (k = 1:m).$$

This remarkably simple algorithm is called *forward substitution*.

Exercise. Formulate the *back substitution algorithm* to find the solution of $Ux = b$ with an invertible upper triangular matrix U.

5.6 The corresponding MATLAB program is:

Program 6 (Forward Substitution to Solve $Lx = b$).

```
x = zeros(m,1);
for k=1:m
  x(k) = (b(k) - L(k,1:k-1)*x(1:k-1))/L(k,k);
end
```

The algorithm is realized with one `for`-loop over inner products; in practice one should employ an optimized Level-2 BLAS routine.

[14] Empty parts in a partitioning lead to convenient base clauses for induction proofs and initializations for algorithms; inner products of empty parts (i.e., *zero-dimensional* vectors) equal zero.

5.7 The inner products require $2k$ flop in leading order. The computational cost of the entire loop is therefore[15]

$$\#\,\text{flop} = 2\sum_{k=1}^{m} k \doteq m^2.$$

The associated memory accesses for input and output are dominated by accessing the triangular matrix (in leading order $m^2/2$ elements) such that

$$q = \#\,\text{flop}/\#\,\text{iop} \doteq 2.$$

Remark. This means the computing time and the efficiency ratio q are *equal* (in leading order) to those of a matrix-vector multiplication with a triangular matrix.

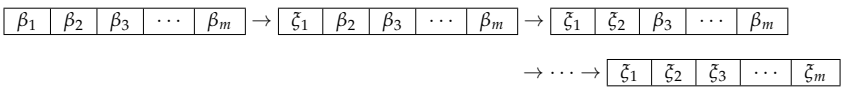
Forward and back substitution are standardized as Level-2 BLAS routines:

BLAS level	name	operation		# flop	q
2	xTRMV	$x \leftarrow Lx$ $x \leftarrow Rx$	matrix-vector multiplication	m^2	2
2	xTRSV	$x \leftarrow L^{-1}x$ $x \leftarrow R^{-1}x$	forward substitution back substitution	m^2	2

The MATLAB commands to solve $Lx = b$ and $Ux = b$ are (MATLAB analyzes the matrix and calls xTRSV for triangular matrices):

```
x = L\b, x = U\b
```

5.8 As soon as ξ_k is calculated, the components β_k are no longer needed for forward and back substitution; the memory from β_k can therefore be used for ξ_k:



One refers to an algorithm that overwrites part of the input data with the output data as *in situ* (in place) execution.

Like every other routine in the BLAS Library, xTRSV works in situ; the in situ version of the MATLAB program from §5.6 can be seen in the following code snippet:

Program 7 (Forward Substitution for $x \leftarrow L^{-1}x$).

```
for k=1:m
  x(k) = (x(k) - L(k,1:k-1)*x(1:k-1))/L(k,k);
end
```

[15] '$\doteq$' represents equality in leading order.

6 Unitary Matrices

6.1 Along with triangular matrices, there is another important group of matrices that belongs to the building blocks of numerical linear algebra. We call $Q \in \mathbb{K}^{m\times m}$ *unitary* (for $\mathbb{K} = \mathbb{R}$ sometimes also *orthogonal*) if

$$Q^{-1} = Q',$$

or, equivalently,

$$Q'Q = QQ' = I.$$

The solution of a linear system $Qx = b$ is then simply $x = Q'b$; the computational cost of $2m^2$ flop is two times larger than that of a triangular system.[16]

6.2 As with triangular matrices, unitary matrices form a group:

Lemma. *The unitary matrices in $\mathbb{K}^{m\times m}$ form a subgroup of $\mathrm{GL}(m;\mathbb{K})$ under matrix multiplication; this subgroup is denoted by $U(m)$ for $\mathbb{K} = \mathbb{C}$ and by $O(m)$ for $\mathbb{K} = \mathbb{R}$.*

Proof. Due to $Q'' = Q$, if Q is unitary, the inverse $Q^{-1} = Q'$ is also unitary. For unitary matrices Q_1, Q_2 it follows from

$$(Q_1Q_2)'Q_1Q_2 = Q_2'Q_1'Q_1Q_2 = Q_2'IQ_2 = Q_2'Q_2 = I$$

that the product Q_1Q_2 is also unitary. □

6.3 The adjoints of the column vectors of Q are the row vectors of Q':

$$Q = \begin{pmatrix} | & & | \\ q_1 & \cdots & q_m \\ | & & | \end{pmatrix}, \qquad Q' = \begin{pmatrix} — \, q_1' \, — \\ \vdots \\ — \, q_m' \, — \end{pmatrix}.$$

Therefore, the matrix product $Q'Q = I$ as defined by (2.3.b) provides

$$q_j' q_l = [j = l] \qquad (j = 1:m, l = 1:m);$$

Vector systems with this property form an *orthonormal basis* of $\mathbb{K}^m$.

Remark. A *rectangular* matrix $Q \in \mathbb{K}^{m\times n}$ with $Q'Q = I \in \mathbb{K}^{n\times n}$ is called *column-orthonormal* since its columns form an *orthonormal system*. It this case we have $n \leqslant m$ (why?).

[16] Yet, the efficiency ratio remains $q \doteq 2$.

6.4 If we write $QQ' = I$ using outer products as in (2.6), we obtain the following generalization of (2.4) for an orthonormal basis:

$$I = \sum_{k=1}^{m} q_k q_k'.$$

When applied to the vector $x \in \mathbb{K}^m$, this equation provides its expansion into the orthonormal basis as given by (2.5):

$$x = \sum_{k=1}^{m} (q_k' x)\, q_k.$$

The components of x with respect to q_k are thus $q_k' x$, they form the vector $Q'x$.

Permutation Matrices

6.5 The operation of swapping the columns a^k of a matrix $A \in \mathbb{K}^{n\times m}$ according to a permutation[17] $\pi \in S_m$ corresponds to a linear *column operation*, that is, a multiplication with some matrix P_π from the *right* (such column operations affect every row in the same manner and are therefore structured like (2.3.c)):

$$AP_\pi = \begin{pmatrix} | & & | \\ a^{\pi(1)} & \cdots & a^{\pi(m)} \\ | & & | \end{pmatrix}, \text{ i.e., } \quad P_\pi = IP_\pi = \begin{pmatrix} | & & | \\ e_{\pi(1)} & \cdots & e_{\pi(m)} \\ | & & | \end{pmatrix}.$$

Since the columns of P_π form an orthonormal basis, P_π is unitary.

Exercise. Show that $\pi \mapsto P_\pi$ provides a *group monomorphism* $S_m \to U(m)$ ($S_m \to O(m)$); the permutation matrices thus form a subgroup of $\mathrm{GL}(m;\mathbb{K})$ that is *isomorphic* to S_m.

6.6 Accordingly, the adjoint $P_\pi' = P_\pi^{-1} = P_{\pi^{-1}}$ swaps the rows a_k' of the matrix $A \in \mathbb{K}^{m\times n}$ by means of multiplication from the *left*:

$$P_\pi' A = \begin{pmatrix} - & a'_{\pi(1)} & - \\ & \vdots & \\ - & a'_{\pi(m)} & - \end{pmatrix}, \text{ i.e., } \quad P_\pi' = P_\pi' I = \begin{pmatrix} - & e'_{\pi(1)} & - \\ & \vdots & \\ - & e'_{\pi(m)} & - \end{pmatrix}.$$

A transposition τ satisfies $\tau^{-1} = \tau$ and therefore $P_\tau' = P_\tau$.

6.7 A permutation $\pi \in S_m$ is encoded as $\mathtt{p} = [\pi(1), \ldots, \pi(m)]$ in MATLAB. In this way, the row and column permutations $P_\pi' A$ and AP_π can be expressed as:

```
A(p,:), A(:,p)
```

One therefore obtains the permutation matrix P_π as follows:

```
I = eye(m); P = I(:,p);
```

[17] The group of all permutations of the set of indexes $\{1, 2, \ldots, m\}$ is the *symmetric group* S_m.

II Matrix Factorization

> Although matrix factorization is not a new subject, I have found no evidence that is has been utilized so directly in the problem of solving matrix equations.
>
> *(Paul Dwyer 1944)*

> We may therefore interpret the elimination method as the combination of two tricks: First, it decomposes A into a product of two triangular matrices. Second, it forms the solution by a simple, explicit, inductive process.
>
> *(John von Neumann, Herman Goldstine 1947)*

7 Triangular Decomposition

7.1 The two "tricks" of the above 1947 von Neumann and Goldstine quote for solving a linear system of equations $Ax = b$ can more formally be written as:

(1) *Factorize* A into invertible lower and upper triangular matrices if possible,

$$A = LU.$$

We refer to this process as the *triangular decomposition* of A.

(2) *Calculate* x by means of one forward and one back substitution

$$Lz = b, \qquad Ux = z.$$

It follows that $b = L(Ux) = Ax$.

We *normalize* such a triangular decomposition by specifying L as *unipotent*.

Remark. The elimination method one might be familiar with can be expressed as a combination of both of the "tricks" listed above. More often than not, this method is attributed to Carl Friedrich Gauss, who spoke of *eliminatio vulgaris* (lat.: common elimination) in 1809, but did not himself invent the method.[18] We will use the term *(Gaussian) elimination* as a synonym of triangular decomposition.

[18] J. F. Grcar: *Mathematicians of Gaussian Elimination*, Notices Amer. Math. Soc. 58, 782–792, 2011.

7.2 As it were, not every invertible matrix has a normalized triangular decomposition (we will be addressing this issue in §7.9), but nevertheless, if one exists, it is *uniquely* defined.

Lemma. *If for a given $A \in \mathrm{GL}(m;\mathbb{K})$ there exists a normalized triangular decomposition $A = LU$ (where L is unipotent lower triangular and U invertible upper triangular), then both factors are uniquely defined.*

Proof. Given two such factorizations $A = L_1U_1 = L_2U_2$, then due to the group properties of (unipotent) triangular matrices, the matrix

$$L_2^{-1}L_1 = U_2U_1^{-1}$$

is simultaneously unipotent lower triangular as well as upper triangular, and must therefore be the identity matrix. Thus $L_1 = L_2$ and $U_1 = U_2$. □

7.3 To calculate the *normalized* triangular decomposition $A = LU$ of $A \in \mathrm{GL}(m;\mathbb{K})$ we let $A_1 = A$, $L_1 = L$, $U_1 = U$ and partition recursively according to

$$A_k = \left(\begin{array}{c|c} \alpha_k & u_k' \\ \hline b_k & B_k \end{array}\right), \quad L_k = \left(\begin{array}{c|c} 1 & \\ \hline l_k & L_{k+1} \end{array}\right), \quad U_k = \left(\begin{array}{c|c} \alpha_k & u_k' \\ \hline & U_{k+1} \end{array}\right), \tag{7.1a}$$

whereby always

$$A_k = L_kU_k.$$

In the kth step, the row (α_k, u_k') of U and the column l_k of L are calculated; in doing so, the dimension is reduced by one:

$$A_k \xrightarrow[(7.1a)]{\text{partition}} \alpha_k, u_k', \underbrace{b_k, B_k}_{\text{auxiliary quantities}} \xrightarrow[(7.1b)\ \&\ (7.1c)]{\text{calculate}} l_k, A_{k+1} \qquad (k = 1:m).$$

The resulting A_{k+1} provides the input for the following step.

When we perform the multiplication for the second block row of $A_k = L_kU_k$ (the first row is *identical* by design), we obtain

$$b_k = l_k\alpha_k, \qquad B_k = l_ku_k' + \underbrace{L_{k+1}U_{k+1}}_{=A_{k+1}}.$$

For $\alpha_k \neq 0$, we can simply solve for l_k and A_{k+1} and are already done:[19]

$$l_k = b_k/\alpha_k, \tag{7.1b}$$

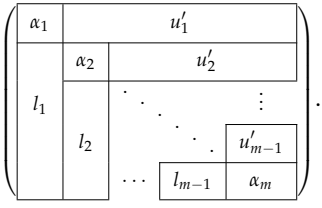
$$A_{k+1} = B_k - l_ku_k'. \tag{7.1c}$$

Only in (7.1b) and (7.1c) we do actual calculations; (7.1a) is merely "book keeping". Since $\alpha_1, \ldots, \alpha_m$ make up the diagonal of U, we have additionally proven:

[19] The matrix A_{k+1} is called the *Schur complement* of α_k in A_k.

Lemma. *A matrix $A \in \mathrm{GL}(m;\mathbb{K})$ has a normalized triangular decomposition if and only if all so-called pivot elements $\alpha_1, \ldots, \alpha_m$ are non-zero.*

7.4 The normalized triangular decomposition can be executed *in situ* by overwriting b_k with l_k and B_k with A_{k+1}. In the end, the location in memory that initially contained A has been overwritten by



From this *compact memory scheme* we subsequently obtain both factors in the form

$$L = \begin{pmatrix} 1 & & & \\ l_1 & 1 & & \\ & l_2 & \ddots & \\ & & \ddots & 1 \\ & & l_{m-1} & \end{pmatrix}, \qquad U = \begin{pmatrix} \alpha_1 & u'_1 & & \\ & \alpha_2 & \ddots & \\ & & \ddots & u'_{m-1} \\ & & & \alpha_m \end{pmatrix}.$$

7.5 In MATLAB, this kind of *in situ* execution of (7.1a)–(7.1c) can be completed with the help of the following commands:

Program 8 (Trianglular Decomposition).

object	access in MATLAB
α_k	`A(k,k)`
u'_k	`A(k,k+1:m)`
b_k, l_k	`A(k+1:m,k)`
B_k, A_{k+1}	`A(k+1:m,k+1:m)`

```
for k=1:m
  A(k+1:m,k) = A(k+1:m,k)/A(k,k);                          % (7.1b)
  A(k+1:m,k+1:m) = A(k+1:m,k+1:m) - A(k+1:m,k)*A(k,k+1:m); % (7.1c)
end
```

Since the last loop iteration ($k = m$) only finds empty (zero dimensional) objects, we can just as well end the loop at $k = m - 1$.

The reconstruction of L and U is achieved with the help of the following commands:

```
L = tril(A,-1) + eye(m);
U = triu(A);
```

Due to the `-1` in `tril(A,-1)`, only elements *below* the diagonal are read.

7.6 The number[20] of floating point operations needed for triangular decomposition is dictated in leading order by the *rank 1 operation* in (7.1c):

$$\#\text{ flop for the calculation of } A_{k+1} = 2(m-k)^2.$$

Therefore the total cost is

$$\#\,\text{flop for } LU\text{-Factorization} \doteq 2\sum_{k=1}^{m}(m-k)^2 = 2\sum_{k=1}^{m-1}k^2 \doteq \frac{2}{3}m^3.$$

This cost of triangular decomposition is furthermore the dominating factor in solving a linear system of equations (the subsequent substitutions for the second "trick" from §7.1 only require $2m^2$ flop).

Since A is only read once, and subsequently overwritten once by the compact memory access scheme for L and U, only $2m^2$ memory accesses have to be carried out for input and output, leading to an efficiency ratio of

$$q = \#\,\text{flop}/\#\,\text{iop} \doteq \frac{m}{3}.$$

Due to this linear dimensional growth, a memory access optimized Level-3 BLAS implementation can achieve near peak performance for large m.

Exercise. Show that the cost for LU factorization is the same as for the associated multiplication: even the product LU of a given lower triangular matrix L with an upper triangular matrix U requires a leading order of $2m^3/3$ flop with an efficiency ratio of $q \doteq m/3$.

7.7 The approach to solving linear systems of equations $Ax = b$ with the help of triangular decomposition as described in §7.1 offers many *advantages*; here are two examples:

Example. Given n and the right-hand-sides $b_1, \ldots, b_n \in \mathbb{K}^m$, we want to calculate the solutions $x_1, \ldots, x_n \in \mathbb{K}^m$. From these vectors we form the matrices

$$B = \begin{pmatrix} | & & | \\ b_1 & \cdots & b_n \\ | & & | \end{pmatrix} \in \mathbb{K}^{m\times n}, \qquad X = \begin{pmatrix} | & & | \\ x_1 & \cdots & x_n \\ | & & | \end{pmatrix} \in \mathbb{K}^{m\times n}$$

and calculate

(1) the triangular decomposition $A = LU$ (cost $\doteq 2m^3/3$ flop);

(2) with the *matrix version* (Level-3-BLAS: `xTRSM`) of the forward and back substitution solutions Z and X of (cost $\doteq 2n \cdot m^2$ flop)

$$LZ = B, \qquad UX = Z.$$

[20] We assume $\mathbb{K} = \mathbb{R}$ by default; for $\mathbb{K} = \mathbb{C}$ we have to multiply, on average, with a factor of four.

For $n \ll m$ the cost of the triangular decomposition outweighs the others. An example application will be discussed in §20.6.

Example. For the simultaneous solution of $Ax = b$ *and* $A'y = c$ we calculate

(1) triangular decomposition $A = LU$ (and thereby, *automatically*, $A' = U'L'$);

(2) by forward and back substitution the solutions z, x, w and y of

$$Lz = b, \qquad Ux = z, \qquad U'w = c, \qquad L'y = w.$$

7.8 As stated in §7.3, as soon as $\alpha_k = 0$ for just one pivot element, a (normalized) triangular decomposition of A does no longer exist. For example, the *invertible* matrix

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

already exhibits $\alpha_1 = 0$. This *theoretical* limitation is accompanied by a *practical* problem. For example, by replacing the zero in A with a small number, we get

$$A = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 1 \end{pmatrix} = LU, \qquad L = \begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{pmatrix}.$$

On a computer only capable of calculations with 16 significant digits, rounding leads to the actual numbers (symbol for such a representation by rounding: '$\doteq$')

$$\begin{aligned} 1 - 10^{20} &= -0.99999\,99999\,99999\,99999 \cdot 10^{20} \\ &\doteq -1.00000\,00000\,00000 \qquad\quad \cdot 10^{20} = -10^{20}; \end{aligned}$$

The subtraction of 1 from 10^{20} is therefore under the *resolution threshold*. Consequently, instead of U, the computer returns the *rounded* triangular matrix $\widehat{U}$,

$$\widehat{U} = \begin{pmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{pmatrix},$$

which corresponds to the triangular decomposition of

$$\widehat{A} = L\widehat{U} = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 0 \end{pmatrix}$$

and not that of A. If we were to solve the system of equations defined by $Ax = e_1$, instead of the computer correctly returning the rounded value

$$x = \frac{1}{1 - 10^{-20}} \begin{pmatrix} -1 \\ 1 \end{pmatrix} \doteq \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

it would return the solution

$$\hat{x} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

of $\widehat{A}\hat{x} = e_1$, with an *unacceptable* error of 100% in the first component.

Conclusion. In general, take the "Zeroth Law of Numerical Analysis" to heart:

> *If theoretical analysis struggles for $\alpha = 0$, numerical analysis does so for $\alpha \approx 0$.*

Triangular Decomposition with Partial Pivoting

7.9 Once one realizes that the row numbering of a matrix is completely *arbitrary*, both problems mentioned in §7.8 can be solved with the following strategy:

> *Partial Pivoting:* In the first column of A_k, the first element with the *largest* absolute value in that column is selected as the pivot element α_k; the associated row is then swapped with the first row of A_k.

Thus, by calling τ_k the transposition of the selected row numbers, we replace (7.1a) by the block partitioning (given $A_1 = A$, $L_1 = L$, $U_1 = U$)

$$P'_{\tau_k} A_k = \left(\begin{array}{c|c} \alpha_k & u'_k \\ \hline b_k & B_k \end{array}\right), \quad L_k = \left(\begin{array}{c|c} 1 & \\ \hline l_k & L_{k+1} \end{array}\right), \quad U_k = \left(\begin{array}{c|c} \alpha_k & u'_k \\ \hline & U_{k+1} \end{array}\right) \tag{7.2a}$$

where[21]

$$|b_k| \leqslant |\alpha_k|.$$

We however *cannot* simply set $P'_{\tau_k} A_k = L_k U_k$, since we must accumulate the row swaps of the subsequent steps in order to multiply consistently. To this end, we inductively introduce the permutation matrices

$$P'_k = \left(\begin{array}{c|c} 1 & \\ \hline & P'_{k+1} \end{array}\right) P'_{\tau_k} \tag{7.2b}$$

and complete the triangular decomposition of the resulting matrix A_k that is created *after* the successive row swaps have been completed:

$$P'_k A_k = L_k U_k. \tag{7.2c}$$

By multiplying for the second row of blocks in this equation, we attain

$$P'_{k+1} b_k = \alpha_k l_k, \qquad P'_{k+1} B_k = l_k u'_k + \underbrace{L_{k+1} U_{k+1}}_{= P'_{k+1} A_{k+1}},$$

which (after left multiplication with P_{k+1}) can be simplified to

$$b_k = \alpha_k v_k, \qquad B_k = v_k u'_k + A_{k+1} \qquad (\text{where } v_k = P_{k+1} l_k).$$

[21] Such inequalities (and the absolute values) have to be read *componentwise*.

We have thus performed *exactly the same* computational steps as described in §7.3, namely

$$v_k = b_k / \alpha_k, \tag{7.2d}$$

$$A_{k+1} = B_k - v_k u_k'. \tag{7.2e}$$

The only difference between this approach and that of (7.1b) and (7.1c), is that we attain l_k only after performing all remaining row-swaps on v_k:

$$l_k = P_{k+1}' v_k. \tag{7.2f}$$

Remark. The *in situ* variant of the algorithm is realized by executing the row swaps of *all* columns of the matrix in memory (i.e., for A_k as well as $v_1, \dots, v_k$).

7.10 The *in situ* execution of (7.2a)–(7.2e) can be written in MATLAB as:

Program 9 (Triangular Decomposition with Partial Pivoting).
We calculate $P'A = LU$ for a given A. As in §6.7, we can realize the row permutation $P'A$ as `A(p,:)` with the vector representation `p` of the corresponding permutation.

```
p = 1:m;  % initialization of the permutation

for k=1:m
  [~,j] = max(abs(A(k:m,k))); j = k-1+j;          % pivot search
  p([k j]) = p([j k]); A([k j],:) = A([j k],:); % row swap
  A(k+1:m,k) = A(k+1:m,k)/A(k,k);                                  % (7.2d)
  A(k+1:m,k+1:m) = A(k+1:m,k+1:m) - A(k+1:m,k)*A(k,k+1:m);  % (7.2e)
end
```

Here too, the reconstruction of L and U can be completed with the following commands

```
L = tril(A,-1) + eye(m);
U = triu(A);
```

In order to reach peak performance, one should replace this program with the MATLAB interface to the `xGETRF` routine from LAPACK:

```
[L,U,p] = lu(A,'vector');
```

Without rounding errors, `A(p,:)=L*U` would be valid.

7.11 As it turns out, the algorithm in §7.9 works actually for all *invertible* matrices:

Theorem. *Given $A \in \mathrm{GL}(m;\mathbb{K})$, the triangular decomposition with partial pivoting results in a permutation matrix P, a unipotent lower triangular matrix L and an invertible upper triangular matrix U where*

$$P'A = LU, \qquad |L| \leqslant 1.$$

Notably, all pivot elements are non-zero.

Proof. Through induction we can show that all A_k are invertible and $\alpha_k \neq 0$. By assumption $A_1 = A$ is invertible (base case). As our induction hypothesis, we then stipulate A_k is also invertible. If the first column of A_k were to be the null vector, A_k could not be invertible. Therefore, the first element with the largest absolute value is some $\alpha_k \neq 0$ and the division for the calculation v_k in (7.2d) is valid. From (7.2a)–(7.2e) it directly follows

$$\left(\begin{array}{c|c} 1 & \\ \hline -v_k & I \end{array}\right) P'_{\tau_k} A_k = \left(\begin{array}{c|c} 1 & \\ \hline -v_k & I \end{array}\right) \cdot \left(\begin{array}{c|c} \alpha_k & u'_k \\ \hline b_k & B_k \end{array}\right) = \left(\begin{array}{c|c} \alpha_k & u'_k \\ \hline 0 & A_{k+1} \end{array}\right).$$

Since the left most entry, being the product of invertible matrices, is itself an invertible matrix, the triangular block matrix on the right must also be invertible. Thus, A_{k+1} is invertible and the induction step is complete.

If we set $k = 1$ in (7.2c) and $P = P_1$, we can see that $P'A = LU$. From $|b_k| \leqslant |\alpha_k|$ and (7.2d) follows $|v_k| \leqslant 1$ and with it $|l_k| \leqslant 1$, such that above all $|L| \leqslant 1$.[22] □

7.12 The methods to find a solution of the linear system of equations discussed in §§7.1 and 7.7 do not change *at all* structurally when we transition from $A = LU$ to $P'A = LU$. In this way, $Ax = b$ becomes $P'Ax = P'b$. We must only replace b with $P'b$ in the resulting formula (i.e., permute the rows of b).

In MATLAB, the solution $X \in \mathbb{K}^{m\times n}$ of $AX = B \in \mathbb{K}^{m\times n}$ is therefore given by:

```
[L, U, p] = lu(A,'vector');
Z = L\B(p,:);
X = U\Z;
```

or *equivalently*, if we do not require the decomposition $P'A = LU$ for re-use:

```
X = A\B;
```

8 Cholesky Decomposition

8.1 For many important applications, from geodesy to quantum mechanics to statistics, the following matrices are of utmost importance:

Definition. A matrix $A \in \mathbb{K}^{m\times m}$ with $A' = A$ is called *self-adjoint* (for $\mathbb{K} = \mathbb{R}$ often also *symmetric* and for $\mathbb{K} = \mathbb{C}$ *hermitian*). The matrix A is furthermore called *positive definite*, when

$$x'Ax > 0 \qquad (0 \neq x \in \mathbb{K}^m).$$

We will refer to self-adjoint positive definite matrices by the acronym *s.p.d.*.

Remark. For self-adjoint A, the term $x'Ax = x'A'x = (x'Ax)'$ is always *real*, even for $\mathbb{K} = \mathbb{C}$. Positive definite matrices have a trivial kernel and are therefore invertible.

[22] Recall that inequalities and absolute values have to be read componentwise.

8.2 If we partition a matrix $A \in \mathbb{K}^{m\times m}$ in the form

$$A = \left(\begin{array}{c|c} A_k & B \\ \hline C & D \end{array}\right), \qquad A_k \in \mathbb{K}^{k\times k},$$

then A_k is called a *principal submatrix* of A. Generally, at least in theory, there is no need to pivot in getting the triangular decomposition of A when all the principal submatrices A_k inherit some structure of the matrix A implying invertibility.

Exercise. Show that $A \in \mathrm{GL}(m;\mathbb{K})$ has a triangular decomposition if and only if all principal submatrices satisfy $A_k \in \mathrm{GL}(k;\mathbb{K})$ $(k = 1:m)$. *Hint:* Partition similarly to §8.3.

For example, along with A, the A_k are s.p.d., too. The self-adjointness of the A_k is clear and for $0 \neq x \in \mathbb{K}^k$ it follows from the positive definiteness of A that

$$x'A_k\,x = \left(\begin{array}{c} x \\ \hline 0 \end{array}\right)' \left(\begin{array}{c|c} A_k & B \\ \hline C & D \end{array}\right) \left(\begin{array}{c} x \\ \hline 0 \end{array}\right) > 0.$$

8.3 For s.p.d. matrices there exists a special form of triangular decomposition which generalizes the notion of the *positive* square root of positive numbers:

Theorem (Cholesky Decomposition). *Every s.p.d. matrix A can be uniquely represented in the form*

$$A = LL',$$

whereby L is a lower triangular matrix with a positive diagonal. The factor L can be constructed row-wise by means of the algorithm (8.1a)–(8.1c).

Proof. We construct the Cholesky decomposition $A_k = L_kL_k'$ of the principal submatrices A_k of $A = A_m$ step by step by partitioning

$$A_k = \left(\begin{array}{c|c} A_{k-1} & a_k \\ \hline a_k' & \alpha_k \end{array}\right), \quad L_k = \left(\begin{array}{c|c} L_{k-1} & \\ \hline l_k' & \lambda_k \end{array}\right), \quad L_k' = \left(\begin{array}{c|c} L_{k-1}' & l_k \\ \hline & \lambda_k \end{array}\right). \tag{8.1a}$$

In the kth step, the row (l_k', λ_k) of L is calculated. In doing this, we prove by induction that L_k is uniquely defined as a lower triangular matrix with a *positive* diagonal. When multiplied out, $A_k = L_kL_k'$ shows initially

$$A_{k-1} = L_{k-1}L_{k-1}', \quad L_{k-1}l_k = a_k, \qquad l_k'L_{k-1}' = a_k', \qquad l_k'l_k + \lambda_k^2 = \alpha_k.$$

The first equation is nothing more than the factorization from step $k-1$; the third equation is the adjoint of the second; the second and fourth one can easily be solved as follows:

$$l_k = L_{k-1}^{-1}a_k \qquad \text{(forward substitution)}, \tag{8.1b}$$

$$\lambda_k = \sqrt{\alpha_k - l_k'l_k}. \tag{8.1c}$$

Here, in accordance with our induction hypothesis L_{k-1} is uniquely defined as a lower triangular matrix with positive diagonal, and therefore notably invertible so that l_k in (8.1b) is also uniquely given (there is nothing to be done for $k = 1$). We must still show that

$$\alpha_k - l_k' l_k > 0,$$

so that the (unique) *positive* square root for λ_k can be extracted from (8.1c). For this, we use the positive definiteness of the principal submatrix A_k from §8.2. Given the solution x_k from $L_{k-1}' x_k = -l_k$ it actually holds that

$$0 < \begin{pmatrix} x_k \\ \hline 1 \end{pmatrix}' A_k \begin{pmatrix} x_k \\ \hline 1 \end{pmatrix} = \left(\begin{array}{c|c} x_k' & 1 \end{array} \right) \left(\begin{array}{c|c} L_{k-1} L_{k-1}' & L_{k-1} l_k \\ \hline l_k' L_{k-1}' & \alpha_k \end{array} \right) \begin{pmatrix} x_k \\ \hline 1 \end{pmatrix}$$

$$= \underbrace{x_k' L_{k-1} L_{k-1}' x_k}_{= l_k' l_k} + \underbrace{x_k' L_{k-1} l_k}_{= -l_k' l_k} + \underbrace{l_k' L_{k-1}' x_k}_{= -l_k' l_k} + \alpha_k = \alpha_k - l_k' l_k.$$

Due to $\lambda_k > 0$, L_k can now be formed as uniquely given lower triangular matrix with positive diagonal; the induction step is complete. □

Remark. The algorithm (8.1a)–(8.1c) was developed by André-Louis Cholesky from 1905 to 1910 for the *Service géographique de l'armée* and in 1924 was posthumously published by Major Ernest Benoît; Cholesky's hand-written manuscript from Dec. 2, 1910 was first discovered in his estate in 2004. For a long time, only a small number of french geodesists had knowledge of this method. This changed when John Todd held a lecture on numerical mathematics in 1946 at the King's College of London, thereby introducing the world to the Cholesky decomposition.[23]

8.4 We code the Cholesky decomposition (8.1a)–(8.1c) in MATLAB as follows:

Program 10 (Cholesky Decomposition of A).

```
L = zeros(m);
for k=1:m
  lk = L(1:k-1,1:k-1)\A(1:k-1,k);      % (8.1b)
  L(k,1:k-1) = lk';                    % (8.1a)
  L(k,k) = sqrt(A(k,k) - lk'*lk);      % (8.1c)
end
```

Notice that the elements of A above the diagonal are not read by the computer (the algorithm "knows" the symmetry). In principle, this memory could be used for other purposes.

Exercise. Modify the program so that it returns an error and a vector $x \in \mathbb{K}^m$ with $x'Ax \leqslant 0$ if the (self-adjoint) matrix A *is not* positive definite.

[23] More on this in C. Brezinski, D. Tournès: *André-Louis Cholesky*, Birkhäuser, Basel, 2014.

In order to reach the peak performance, one should call the MATLAB interface to the xPOTRF routine from LAPACK:

```
L = chol(A,'lower');
```

When $U = L'$, then $A = U'U$ is an alternative form of the Cholesky decomposition. Since the factor U can be constructed *column-wise* in accordance with (8.1a), its calculation is slightly faster in LAPACK and MATLAB than that of L (cf. §4.8):

```
U = chol(A);
```

8.5 The number of floating point operations needed for a Cholesky decomposition is dominated by the operation count of the forward substitutions in (8.1b). Therefore, the total computational cost is

$$\#\,\text{flop for Cholesky decomposition} \doteq \sum_{k=1}^{m} k^2 \doteq \frac{1}{3}m^3,$$

and is thereby (asymptotically) only *half as large* as the cost of a normalized triangular decomposition, without exploiting symmetry, as found in §7.6.

Since only the lower half of A must be read and only the triangular factor L must be stored, only m^2 memory accesses are necessary for input and output (in leading order). Hence, as with triangular decomposition the efficiency ratio is

$$q = \#\,\text{flop}/\#\,\text{iop} \doteq \frac{m}{3},$$

so that with the help of Level-3 BLAS, an memory-access-optimized implementation can reach near peak performance for large m.

9 QR Decomposition

9.1 We will refer to $A \in \mathbb{K}^{m\times n}$ as a matrix with *full column rank* when its columns are linearly independent. Such matrices can be characterized in a number of ways:

Lemma. *A full column rank of $A \in \mathbb{K}^{m\times n}$ is equivalent to each of the following properties:*

$$\text{(1) } \operatorname{rank} A = \dim \operatorname{im} A = n \leqslant m, \qquad \text{(2) } \ker A = \{0\}, \qquad \text{(3) } A'A \text{ is s.p.d.}$$

The matrix $A'A$ is called the Gramian matrix of the columns of A.

Proof. The equivalence to (1) and (2) follows directly from §2.8 and should in fact be well known from past introductions to linear algebra. According to §2.11, $A'A$ is self-adjoint and it holds according to §2.9 that

$$x'(A'A)x = (Ax)'(Ax) \geqslant 0 \qquad (x \in \mathbb{K}^n).$$

Due to $(Ax)'(Ax) = 0 \Leftrightarrow Ax = 0$, both (2) and (3) are equivalent. □

9.2 Our goal is to factorize a matrix $A \in \mathbb{K}^{m\times n}$ with full column rank as

$$A = QR$$

with $Q \in \mathbb{K}^{m\times n}$ being column orthonormal[24] and $R \in \mathrm{GL}(n;\mathbb{K})$ upper triangular. Such a QR *decomposition* is said to be *normalized*, if the diagonal of R is positive.

Remark. Since the columns of Q span the image of $A = QR$, they form, by definition, an *orthonormal basis* of the image.

9.3 The QR decomposition of A is closely related to the Cholesky decomposition of the Gramian matrix $A'A$:

Theorem. *Every matrix $A \in \mathbb{K}^{m\times n}$ with full column rank has a unique normalized QR decomposition. The factors can be determined as follows:*

$$\begin{aligned} A'A &= R'R \quad \text{(Cholesky decomposition of } A'A\text{)},\\ R'Q' &= A' \quad \text{(forward substitution for } Q'\text{)}. \end{aligned}$$

Proof. We assume that A has a normalized QR decomposition. Then,

$$A'A = R'\underbrace{Q'Q}_{=I}R = R'R$$

is according to Theorem 8.3 a *unique* Cholesky decomposition of the Gramian matrix $A'A$, which according to Theorem 9.1 is s.p.d.. Given the upper triangular factor R with positive diagonal, the expression

$$Q = AR^{-1}$$

is therefore also uniquely defined. By showing the column orthonormality of the thus *defined* factor Q, we can conversely ensure the existence of the QR decomposition itself:

$$Q'Q = (L^{-1}A')(AR^{-1}) = L^{-1}LRR^{-1} = I$$

with $L = R'$ from the Cholesky decomposition $A'A = R'R$. □

9.4 The construction of the QR decomposition via the Cholesky decomposition of the Gramian $A'A$ is only seldom used in practical numerical work for the following two reasons (see also §16.5):

- It is *more expensive* for $n \approx m$ than algorithms which work directly on A.
- The orthonormality of the factor Q is *not explicitly* built into the process, but is rather only the implicit result of the theory. Such an indirect approach is extremely *susceptible* to the influence of rounding errors.

Exercise. Show that the number of floating point operations in the algorithm from Theorem 9.3 is (in leading order) $2mn^2 + n^3/3$. Discuss the cases of $n \ll m$ and $n \approx m$.

[24] Recall from §6.3 that such matrices are defined by $Q'Q = I \in \mathbb{K}^{n\times n}$.

Modified Gram–Schmidt

9.5 For the *direct* calculation of the normalized QR decomposition of A from §9.3, we set $A_1 = A$, $Q_1 = Q$, $R_1 = R$ and partition stepwise according to[25]

$$A_k = \left(b_k \;\middle|\; B_k \right) = Q_k R_k, \quad Q_k = \left(q_k \;\middle|\; Q_{k+1} \right), \quad R_k = \left(\begin{array}{c|c} \rho_k & r_k' \\ \hline & R_{k+1} \end{array} \right). \tag{9.1a}$$

In the kth step, the column q_k of Q and the row (ρ_k, r_k') of R_k are calculated:

$$A_k \xrightarrow[(9.1a)]{\text{partition}} \underbrace{b_k, B_k}_{\text{auxiliary quantities}} \xrightarrow[(9.1b)\text{–}(9.1e)]{\text{calculate}} \rho_k, q_k, r_k', A_{k+1} \qquad (k = 1 : n).$$

The output A_{k+1} thus provides the input for the next step. If we expand the *single* block row of $A_k = Q_k R_k$, we obtain

$$b_k = q_k \rho_k, \qquad B_k = q_k r_k' + \underbrace{Q_{k+1} R_{k+1}}_{=A_{k+1}},$$

which we solve for ρ_k, q_k, r_k', A_{k+1}. From $\|q_k\|_2 = (q_k' q_k)^{1/2} = 1$ we get with $\rho_k > 0$

$$\rho_k = \|b_k\|_2, \tag{9.1b}$$

$$q_k = b_k / \rho_k, \tag{9.1c}$$

and from $q_k' Q_{k+1} = 0$ it follows that

$$q_k' B_k = \underbrace{q_k' q_k}_{=1} r_k' + \underbrace{q_k' Q_{k+1}}_{=0} R_{k+1} = r_k',$$

so that finally

$$r_k' = q_k' B_k, \tag{9.1d}$$

$$A_{k+1} = B_k - q_k r_k'. \tag{9.1e}$$

The algorithm (9.1a)–(9.1e) is called *modified Gram–Schmidt* (MGS).

Remark. In 1907, Erhard Schmidt described a process for orthonormalization which he attributed, in a footnote, to the 1879 dissertation of the insurance mathematician Jørgen Pedersen Gram (who had however used determinants); the process was first referred to as *Gram–Schmidt* by the statistician Y. K. Wong in 1935. The slightly *modified* Gram–Schmidt process is absolutely preferable for practical numerical work and can essentially be found in the famous fundamental work of Pierre-Simon Laplace on probability theory (1816).[26]

[25] We will thereby *employ* Q_k and R_k as submatrices of the factors Q and R, whose unique existence we have already ensured in Theorem 9.3. In particular, Q_k is column orthogonal and R_k is an upper triangular matrix with positive diagonal so that $q_k' q_k = 1$, $q_k' Q_{k+1} = 0$ and $\rho_k > 0$.

[26] S. J. Leon, Å. Björck, W. Gander: *Gram–Schmidt orthogonalization: 100 years and more*, Numer. Linear Algebra Appl. 20, 492–532, 2013.

Exercise. Observe $A_{k+1} = (I - q_k' q_k) B_k$ and show: (1) if a_k is the kth column of A then

$$\rho_k q_k = (I - q_{k-1} q_{k-1}') \cdots (I - q_1 q_1') a_k \qquad (k = 1 : n).$$

(2) The *orthogonal projection* $P = I - qq'$ with $q'q = 1$ satisfies $Pq = 0$ and $Pu = u$ if $q'u = 0$.

9.6 The MGS-Algorithm (9.1a)–(9.1e) can be executed *in situ* by overwriting b_k with q_k and B_k with A_{k+1} so that finally the matrix Q is found in memory where the input A was previously stored. In MATLAB, the process can be coded as:

Program 11 (QR Decomposition with the MGS Algorithm).

object	access in MATLAB
ρ_k	`R(k,k)`
r_k'	`R(k,k+1:n)`
b_k, q_k	`A(:,k)`
B_k, A_{k+1}	`A(:,k+1:n)`

```
R = zeros(n);
for k=1:n
  R(k,k) = norm(A(:,k),2);                          % (9.1b)
  A(:,k) = A(:,k)/R(k,k);                           % (9.1c)
  R(k,k+1:n) = A(:,k)'*A(:,k+1:n);                  % (9.1d)
  A(:,k+1:n) = A(:,k+1:n) - A(:,k)*R(k,k+1:n);      % (9.1e)
end
```

After execution of the program, the MATLAB variable `A` contains the factor Q.

9.7 The computational cost for QR decomposition is dictated in leading order by the Level-2 BLAS operations in (9.1d) and (9.1e):

$$\#\,\text{flop for the computation of } r_k' = 2m(n-k),$$

$$\#\,\text{flop for the computation of } A_{k+1} = 2m(n-k).$$

The total computational cost is therefore

$$\#\,\text{flop for } QR \text{ decomposition with MGS} \doteq 4m \sum_{k=1}^{n} (n-k) \doteq 2mn^2.$$

Since A is read and subsequently overwritten with Q and also R has to be stored, input-output requires $2mn + n^2/2$ memory accesses. Hence, the efficiency ratio is

$$q = \#\,\text{flop}/\#\,\text{iop} \doteq \frac{2mn^2}{2mn + n^2/2} \approx \begin{cases} n & n \ll m, \\ \frac{4}{5}m & n \approx m. \end{cases}$$

Thanks to Level-3 BLAS, peak performance can be approximately reached for large n; a large m alone, however, would not be enough.

9.8 For the calculation of an (often *not* normalized) QR decomposition, MATLAB offers the interface

```
[Q,R] = qr(A,0);
```

to the LAPACK routines `xGEQRF`, `xORGQR` und `xUNGQR`. These do *not* use the MGS algorithms but rather the Householder method; details can be found in Appendix D. The factor Q can thus be much more accurate (more on this later) and the factor R is of similar quality. Nevertheless, this gained accuracy does come with a cost of

$$\#\,\text{flop for } QR \text{ decomposition with Householder} \doteq 4mn^2 - \frac{4}{3}n^3,$$

which is a factor of $4/3$ to 2 higher than the MGS algorithm. These costs can however be reduced by a factor of 2 when Q is not explicitly computed and instead an implicit representation is used to calculate matrix-vector products such as Qx or $Q'y$. The computational cost of the latter operations is then proportional to mn. Unfortunately, MATLAB does not provide an interface to these representations. Still, the factor Q is not needed for many applications, and the factor R can be directly calculated with the commands

```
R = triu(qr(A)); R = R(1:n,1:n);
```

for half the cost, i.e., $2mn^2 - 2n^3/3$ flop.

Givens Rotations

9.9 The normalized QR decomposition of a vector $0 \neq x \in \mathbb{K}^2$ is given by

$$\underbrace{\begin{pmatrix} \xi_1 \\ \xi_2 \end{pmatrix}}_{=x} = \underbrace{\begin{pmatrix} \xi_1/\rho \\ \xi_2/\rho \end{pmatrix}}_{=q_1} \rho, \qquad \rho = \|x\|_2.$$

By extending q_1 to a suitable orthonormal basis q_1, q_2, we directly obtain the *unitary* matrix (check it!)

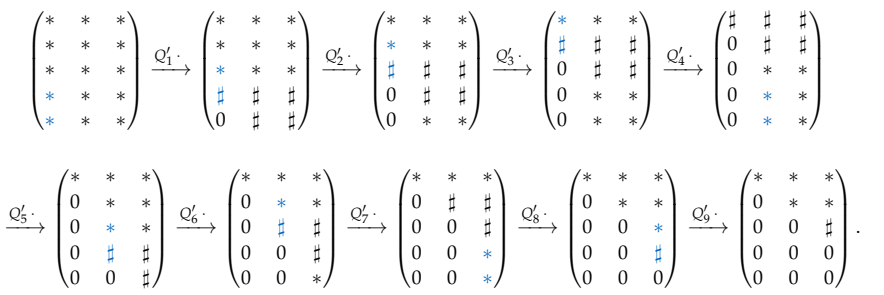
$$\Omega = \left(q_1 \;\middle|\; q_2 \right) = \rho^{-1} \begin{pmatrix} \xi_1 & -\xi_2' \\ \xi_2 & \xi_1' \end{pmatrix}, \qquad \det \Omega = 1,$$

for which it then holds

$$x = \Omega \begin{pmatrix} \|x\|_2 \\ 0 \end{pmatrix}, \qquad \Omega' x = \begin{pmatrix} \|x\|_2 \\ 0 \end{pmatrix}. \tag{9.2}$$

By multiplying with Ω' we thereby "eliminate" the second component of x; such an Ω is called *Givens rotation* induced by the vector x. Using $\Omega = I$, we see that $x = 0$ is actually *no* exception and the elimination relationship (9.2) can be made to hold for all $x \in \mathbb{K}^2$.

9.10 Using Givens rotations, a QR decomposition of a general matrix $A \in \mathbb{K}^{m\times n}$ can be calculated *without any condition on the column rank* by column-wise "eliminating", entry by entry, every component below the diagonal from the bottom up. Schematically this process takes on the following form:



Here, the '$*$' represent arbitrary elements. In every step, the two elements that make up a Givens rotation Ω_j are colored blue. This way, assuming a suitable dimension for I, we have

$$Q'_j = \begin{pmatrix} I & & \\ & \Omega'_j & \\ & & I \end{pmatrix}, \qquad \Omega'_j \begin{pmatrix} * \\ * \end{pmatrix} = \begin{pmatrix} \sharp \\ 0 \end{pmatrix},$$

as constructed in (9.2). In order to make it clear that Q_j only has an affect on the two related rows, we label the elements that undergo a change during multiplication with a '$\sharp$'. Finally, after s such steps, the product[27] $Q' = Q'_s \cdots Q'_1$ lets us arrive at:

Theorem. *For $A \in \mathbb{K}^{m\times n}$ with $m \geqslant n$, there exists a unitary matrix $Q \in \mathbb{K}^{m\times m}$ and an upper triangular matrix $R \in \mathbb{K}^{n\times n}$, such that*[28]

$$A = \underbrace{\left(\begin{array}{c|c} Q_1 & Q_2 \end{array}\right)}_{=Q} \begin{pmatrix} R \\ \hline 0 \end{pmatrix}, \qquad A = Q_1 R; \tag{9.3}$$

The first relationship in (9.3) is called the full QR decomposition of A, while the second is called the reduced QR decomposition of A.

Exercise. State a corresponding theorem for $m < n$.

[27] In practical numerical work, one *only* stores the Givens rotations Ω_j and *does not* explicitly calculate the matrix Q. The matrix vector products Qx and $Q'y$ can then be *evaluated* in $6s$ flop by applying Ω_j and Ω'_j to the appropriate components of x and y.

[28] In MATLAB (though it is based on the Householder method): `[Q,R] = qr(A);`

9.11 The Givens process from §9.10 is *especially* effective when only a few elements need to be eliminated in order to reach a triangular form.

Example. Only $s = 4$ Givens rotations are required for the calculation of the QR decomposition

$$H = \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{pmatrix} \xrightarrow{Q'\cdot} \begin{pmatrix} \sharp & \sharp & \sharp & \sharp & \sharp \\ 0 & \sharp & \sharp & \sharp & \sharp \\ 0 & 0 & \sharp & \sharp & \sharp \\ 0 & 0 & 0 & \sharp & \sharp \\ 0 & 0 & 0 & 0 & \sharp \end{pmatrix} = R.$$

In general, matrices $H \in \mathbb{K}^{m\times m}$ with such an occupancy structure are called *upper Hessenberg matrices*. As in §5.2 we characterize these matrices by

$$HV_k \subset V_{k+1} \qquad (k = 1 : m-1).$$

Their QR decomposition can be calculated with only $s = m - 1$ Givens rotations. (This will play a large role late during our discussion of eigenvalue problems).

Exercise. How many floating point operations are needed for a QR decomposition of an upper Hessenberg matrix provided Q is *not* expanded but rather only the individual Givens rotations $\Omega_1, \ldots, \Omega_s$ are stored? *Answer*: # flop $\doteq 3m^2$ for $\mathbb{K} = \mathbb{R}$.

Exercise. We have introduced three *standard form* matrix factorizations: (a) triangular decomposition with pivoting: $A = PLU$; (b) Cholesky: $A = LL'$; (c) orthogonalization: $A = QR$.

- Do there exist factorizations of the following form (pay attention to the underlying assumptions and dimensions)?

  (a) PUL (b) UU' (c) QL, RQ, LQ

- Which of the variants can be reduced to its respective standard form? If applicable, write a *short* MATLAB program using the basic commands `lu`, `chol` or `qr`.

Hint: A diagonal splits a flat square into two congruent triangles. Which *geometric* transformations map one triangle onto the other? What does that mean for a matrix if considered a flat array of numbers?

III Error Analysis

Most often, instability is caused not by the accumulation of millions of rounding errors, but by the insidious growth of just a few.

(Nick Higham 1996)

Competent error-analysts are extremely rare.

(Velvel Kahan 1998)

Contrary to a futile ideal of absolute, uncompromisingly precise calculation, the real world knows an abundance of unavoidable *inaccuracies* or *perturbations*, in short *errors*, which are generated, e.g., from the following sources:

- *modeling errors* in expressing a scientific problem as a mathematical one;
- *measurement errors* in input data or parameters;
- *rounding errors* in the calculation process on a computer;
- *approximation errors* in resolving limits by iteration and approximation.

Despite the negative connotation of their names, such errors, inaccuracies and perturbations represent something fundamentally valuable:

Errors are what make efficient numerical computing possible in the first place.

There is a trade off between computational cost and accuracy: accuracy comes at a cost and we should therefore not require more accuracy than is needed or even technically possible. On the other hand, we have to learn to avoid *unnecessary* errors, and learn to classify and select algorithms. For this reason, it is important to become familiar with systematic error analysis early on.

10 Error Measures

10.1 For a tuple of matrices (vectors, scalars), we can measure the perturbation

$$T = (A_1, \ldots, A_t) \xrightarrow{\text{perturbation}} \tilde{T} = T + E = (A_1 + E_1, \ldots, A_t + E_t)$$

with the help of a basic set of norms[29] by defining an *error measure* $[\![E]\!]$ as follows:

- *absolute error* $\quad [\![E]\!]_{\text{abs}} = \max_{j=1:t} \|E_j\|$
- *relative error*[30] $\quad [\![E]\!]_{\text{rel}} = \max_{j=1:t} \|E_j\| / \|A_j\|$

How we split a data set into such a tuple is of course not uniquely defined, but rather depends on the identification of *independently* perturbed components. If a tuple consists of scalars (e.g., the *components* of a matrix or a vector), we refer to it as a *componentwise* error measure.

10.2 For *fixed* A the error measure $E \mapsto [\![E]\!]$ is subject to the *exact* same rules as a norm; the only difference being that a relative error can also assume the value ∞. A relative error can also be characterized as follows:

$$[\![E]\!]_{\text{rel}} \leqslant \epsilon \quad \Leftrightarrow \quad \|E_j\| \leqslant \epsilon \cdot \|A_j\| \quad (j = 1:t).$$

It notably must follow from $A_j = 0$ that $E_j = 0$, that is unless $[\![E]\!]_{\text{rel}} = \infty$. Hence, *Componentwise* relative error is for example taken into account when the occupancy structure (or sparsity structure) of a matrix is itself *not* subject to perturbations.

10.3 The choice of a relative or absolute error measure can be made on a problem to problem basis, e.g.:

- quantities with physical dimensions (time, distance, etc.): *relative* error;
- rounding error in floating point arithmetic (cf. §12): *relative* error;
- numbers with a fixed scale (probabilities, counts, etc.): *absolute* error.

Yet sometimes, this decision is just pragmatically based on which of the concepts allows for a *simpler* mathematical result such as an estimate.

[29] For a review of norms see Appendix C. We will limit ourselves to the norms from §C.9.
[30] In this context we agree upon $0/0 = 0$ and $\epsilon/0 = \infty$ for $\epsilon > 0$.

11 Conditioning of a Problem

11.1 We will formalize a computational problem

$$\boxed{\text{data}} \longrightarrow \boxed{\text{math. model } f} \longrightarrow \boxed{\text{result}}$$

as the evaluation of a mapping[31] $f : x \mapsto y = f(x)$. Here, there are no "actual", precise values of the data that could claim any mathematical truth to themselves, but data (unlike, if any, parameters) are inherently subject to perturbations:

$$\boxed{\textit{perturbation} \text{ of data}} \longrightarrow \boxed{\text{math. model } f} \longrightarrow \boxed{\textit{pertubation} \text{ of result}}$$

If such a perturbation of x on the input yields some $\tilde{x}$, by mathematical necessity, rather than $y = f(x)$ we "only" obtain the result $\tilde{y} = f(\tilde{x})$.

11.2 The *condition number* describes the ratio of the perturbation of the result to the perturbation of the input. For a given problem f, at a data point x, using the measure $[\![\cdot]\!]$, the *worst case* of this ratio is—in the limit of very small perturbations—defined as

$$\kappa(f;x) = \limsup_{\tilde{x}\to x} \frac{[\![f(\tilde{x}) - f(x)]\!]}{[\![\tilde{x} - x]\!]}.$$

At a data point x, a problem f is called

- *well-conditioned* if $\kappa(f;x) \not\gg 1$;
- *ill-conditioned* if $\kappa(f;x) \gg 1$;
- *ill-posed*, if $\kappa(f;x) = \infty$.

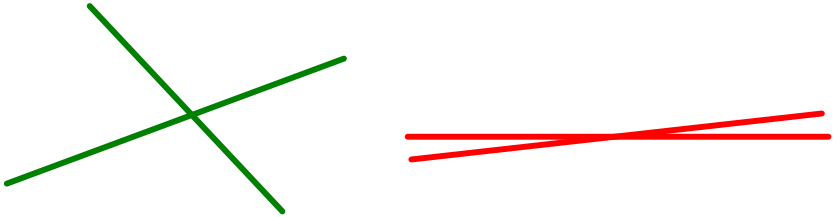
Where exactly do we draw the line for "$\kappa \gg 1$" (read: condition number far greater than 1)? The phenomenon of ill-conditioning quantitatively depends on the requirements and the accuracy structure of the application at hand. For the sake of definiteness, take a value of say $\kappa \geqslant 10^5$ for the following passages.

Remark. To put it as lucent as possible: whether a problem is ill-conditioned (ill-posed) or not depends only on the underlying mathematical model. The question does not, never ever, depend on the prospect that the problem might be, at some point, be tried to be solved using algorithms on a computer. The question of ill-conditioning (ill-posedness) is therefore *completely independent of the realm of algorithms and computers*. In the case of a highly sensitive problem, one has to carefully consider why a result should be calculated in the first place and to which perturbations it is actually subjected.

[31] In this very general description, x represents a tuple of matrices as in §10.1.

11.3 To exemplify this concept, we use a "classic" of geometric linear algebra.

Example. The conditioning of a problem can be nicely visualized when we attempt to find the intersection of two lines (that is, hereby, the data):



The lines shown are only accurate within their *width*. The intersection on the left is essentially as accurate as the data; on the right, for two lines close to each other, the accuracy of the intersection is drastically reduced. This is called a *glancing intersection*. The problem on the left is therefore *well*-conditioned and the one on the right is *ill*-conditioned if put to the extreme.

11.4 The definition of the condition number of a problem f at a data point x can alternatively be put as follows: take the *smallest* number $\kappa(f;x) \geqslant 0$ such that

$$[\![f(x+w) - f(x)]\!] \leqslant \kappa(f;x) \cdot [\![w]\!] + o([\![w]\!]) \qquad ([\![w]\!] \to 0).$$

Here, the *Landau-Symbol* $o(\epsilon)$ represents a term which converge *superlinearly* towards zero as $\epsilon \to 0$: $o(\epsilon)/\epsilon \to 0$. When we omit such additive, superlinearly decaying terms and only list the term of *leading order* relative to the perturbation, we can denote this briefly using the symbol '$\dot{\leqslant}$' (and accordingly '$\dot{\geqslant}$' and '$\doteq$'):

$$[\![f(x+w) - f(x)]\!] \dot{\leqslant} \kappa(f;x) \cdot [\![w]\!].$$

11.5 For perturbations $w \to 0$, a *differentiable* map $f : D \subset \mathbb{R}^m \to \mathbb{R}$ has, by definition, the linearization

$$f(x+w) \doteq f(x) + f'(x) \cdot w.$$

Here, the derivative is the *row vector* composed of the partial derivatives

$$f'(x) = (\partial_1 f(x), \ldots, \partial_m f(x)) \in \mathbb{K}^{1\times m}.$$

Through linearization, *closed* formulas for the condition number can be attained. The case of *componentwise* relative error can be completed as follows:

$$\kappa(f;x) = \limsup_{w\to 0} \frac{[\![f(x+w) - f(x)]\!]}{[\![w]\!]} = \lim_{\epsilon\to 0} \sup_{[\![w]\!]\leqslant\epsilon} \frac{|f(x+w) - f(x)|}{|f(x)| \cdot [\![w]\!]}$$

$$= \lim_{\epsilon\to 0} \sup_{[\![w]\!]\leqslant\epsilon} \frac{|f'(x)\cdot w|}{|f(x)| \cdot [\![w]\!]} \overset{\text{(a)}}{=} \sup_{[\![w]\!]\neq 0} \frac{|f'(x)\cdot w|}{|f(x)| \cdot [\![w]\!]} \overset{\text{(b)}}{=} \frac{|f'(x)| \cdot |x|}{|f(x)|}.$$

Here, (a) holds, because numerator and denominator are absolutely homogeneous in w. By also proving (b) below, we obtain the following *condition number formula*:

Theorem. *For a differentiable $f : D \subset \mathbb{R}^m \to \mathbb{R}$ and $f(x) \neq 0$ it holds*

$$\kappa(f;x) = \frac{|f'(x)| \cdot |x|}{|f(x)|} \tag{11.1}$$

with respect to componentwise relative error in x and relative error in $f(x)$. This formula provides the smallest number $\kappa(f;x) \geqslant 0$ such that

$$\frac{|f(x+w) - f(x)|}{|f(x)|} \dot{\leqslant} \kappa(f;x)\epsilon \quad \text{where} \quad |w| \leqslant \epsilon|x| \text{ as } \epsilon \to 0.$$

We call the value $\kappa(f;x)$ the componentwise relative condition number of f at x.

Proof. Set $y' = f'(x)$. The perturbation w of x satisfies, with respect to componentwise relative error, $[\![w]\!] = \|w_x\|_\infty$, where $w_x \in \mathbb{R}^m$ represents the componentwise division of w by x; the componentwise product of y and x will be denoted by ${}^x y \in \mathbb{R}^m$. Hence, in accordance with (C.3),

$$\sup_{[\![w]\!] \neq 0} \frac{|y' \cdot w|}{[\![w]\!]} = \sup_{w_x \neq 0} \frac{|{}^x y' \cdot w_x|}{\|w_x\|_\infty} = \sup_{v \neq 0} \frac{|{}^x y' \cdot v|}{\|v\|_\infty} = \|{}^x y'\|_\infty = \|{}^x y\|_1 = |y'| \cdot |x|.$$

whereby (b) is proven.[32] □

11.6 With formula (11.1) at hand, we can directly ascertain the *componentwise relative* condition number κ of the elementary arithmetic operations f:

f	$\xi_1 + \xi_2$	$\xi_1 - \xi_2$	$\xi_1 \xi_2$	ξ_1 / ξ_2	ξ_1^{-1}	$\sqrt{\xi_1}$
κ	1	$\frac{\lvert\xi_1\rvert + \lvert\xi_2\rvert}{\lvert\xi_1 - \xi_2\rvert}$	2	2	1	1/2

for $\xi_1, \xi_2 > 0$.

We will walk through the calculations for addition and subtraction, the remaining values are left as an exercise. When $x = (\xi_j)_{j=1:2}$ and $f(x) = \xi_1 \pm \xi_2$ it holds

$$\kappa(f;x) = \frac{|f'(x)| \cdot |x|}{|f(x)|} = \frac{|(1, \pm 1)| \cdot \begin{pmatrix} |\xi_1| \\ |\xi_2| \end{pmatrix}}{|\xi_1 \pm \xi_2|} = \frac{(1,1) \cdot \begin{pmatrix} |\xi_1| \\ |\xi_2| \end{pmatrix}}{|\xi_1 \pm \xi_2|} = \frac{|\xi_1| + |\xi_2|}{|\xi_1 \pm \xi_2|}.$$

In the case of addition, this can be reduced for $\xi_1, \xi_2 > 0$ to $\kappa = 1$. With the notable exception of genuine subtraction, all elementary operations are well conditioned with respect to componentwise relative error. Genuine subtraction, however, is ill-conditioned in the case of *cancellation*, that is, in the case

$$|\xi_1 \pm \xi_2| \ll |\xi_1| + |\xi_2|, \tag{11.2}$$

and it is even ill-posed for $\xi_1 \pm \xi_2 = 0$.

[32] Notice that the zero components of x in this example are not a problem (why?)

Example. Let us subtract the numbers

$$\xi_1 = 1.2345689? \cdot 10^0$$
$$\xi_2 = 1.2345678? \cdot 10^0,$$

where the '?' represents uncertainty in the 9th decimal place. This yields the result

$$\xi_1 - \xi_2 = 0.0000011? \cdot 10^0 = 1.1? \cdot 10^{-6},$$

in which the uncertainty has been moved to the 3rd decimal place of the *normalized* representation (where the leading position is nonzero): *We therefore have lost the significance of 6 decimal places*. In this case, the condition number of subtraction is

$$\kappa \approx 2.2 \cdot 10^6.$$

Quite generally it can be stated that:

> *Compared to the data, a result loses about* $\log_{10} \kappa$ *significant digits in accuracy (with* κ *the componentwise relative condition number of the problem).*

Exercise. Show the componentwise relative condition number of the inner product $x'y$ to be

$$\kappa = 2\frac{|x'| \cdot |y|}{|x' \cdot y|} \qquad (x, y \in \mathbb{R}^m). \tag{11.3}$$

Characterize the ill-conditioned/ill-posed case and compare with §11.7.

Conditioning of Matrix Products and of Linear Systems of Equations

11.7 Let us examine the matrix product AB for $A \in \mathbb{K}^{m\times n}$, $B \in \mathbb{K}^{n\times p}$ subjected to perturbations with a small relative error in the form of

$$\tilde{A} = A + E, \quad \|E\| \leqslant \epsilon\|A\|, \qquad \tilde{B} = B + F, \quad \|F\| \leqslant \epsilon\|B\|.$$

It then holds, given the triangular inequality and submultiplicativity, that

$$\|\tilde{A}\tilde{B} - AB\| \leqslant \underbrace{\|E\| \cdot \|B\|}_{\leqslant \epsilon\|A\|\cdot\|B\|} + \underbrace{\|A\| \cdot \|F\|}_{\leqslant \epsilon\|A\|\cdot\|B\|} + \underbrace{\|E\| \cdot \|F\|}_{\leqslant \epsilon^2\|A\|\cdot\|B\|} \dot{\leqslant} 2\epsilon\|A\| \cdot \|B\|,$$

so that the relative error of the perturbed result satisfies the following estimate:

$$\frac{\|\tilde{A}\tilde{B} - AB\|}{\|AB\|} \dot{\leqslant} 2\frac{\|A\| \cdot \|B\|}{\|A \cdot B\|}\,\epsilon.$$

Here, if only *one* of the two factors A, B, is perturbed, we can omit the factor 2.

Remark. The relative condition number κ of the matrix product AB therefore fulfills the estimate

$$\kappa \leqslant 2\frac{\|A\| \cdot \|B\|}{\|A \cdot B\|}.$$

In specific cases, such *upper bounds* provide either *proof* of a good condition number or otherwise *hint* towards a poor condition number.

Exercise. For *componentwise* perturbations of the form

$$\tilde{A} = A + E, \quad |E| \leqslant \epsilon |A|, \qquad \tilde{B} = B + F, \quad |F| \leqslant \epsilon |B|,$$

show the following estimates (the factor 2 is omitted when only one factor is perturbed)

$$|\tilde{A}\tilde{B} - AB| \dot{\leqslant} 2\epsilon |A| \cdot |B|, \qquad \frac{\|\tilde{A}\tilde{B} - AB\|_\infty}{\|AB\|_\infty} \dot{\leqslant} 2 \frac{\||A| \cdot |B|\|_\infty}{\|A \cdot B\|_\infty} \epsilon. \tag{11.4}$$

11.8 For a linear system of equations $Ax = b$ we distinguish two cases:

Perturbation of $b \in \mathbb{K}^m$ with a relative error of the form

$$\tilde{b} = b + r, \qquad \|r\| \leqslant \epsilon \|b\|,$$

leads to $A\tilde{x} = b + r$, so that

$$\tilde{x} - x = A^{-1} r.$$

With the *induced* matrix norm it holds

$$\begin{aligned} \|\tilde{x} - x\| &\leqslant \|A^{-1}\| \cdot \|r\| \\ &\leqslant \|A^{-1}\| \cdot \|A\| \cdot \|x\| \, \epsilon. \end{aligned}$$

The relative error of the result therefore fulfills the estimate

$$\frac{\|\tilde{x} - x\|}{\|x\|} \leqslant \kappa(A) \, \epsilon.$$

Perturbation of $A \in \mathrm{GL}(m; \mathbb{K})$ with a relative error of the form

$$\tilde{A} = A + E, \qquad \|E\| \leqslant \epsilon \|A\|,$$

leads to $(A + E)\tilde{x} = b$, so that

$$x - \tilde{x} = A^{-1} E \tilde{x}.$$

With an *inducing* vector norm it holds

$$\begin{aligned} \|x - \tilde{x}\| &\leqslant \|A^{-1}\| \cdot \|E\| \cdot \|\tilde{x}\| \\ &\leqslant \|A^{-1}\| \cdot \|A\| \cdot \|\tilde{x}\| \, \epsilon. \end{aligned}$$

The relative error of the result (here in reference to $\tilde{x}$) therefore fulfills

$$\frac{\|x - \tilde{x}\|}{\|\tilde{x}\|} \leqslant \kappa(A) \, \epsilon. \tag{11.5}$$

Here, we have defined

$$\kappa(A) = \|A^{-1}\| \cdot \|A\|$$

as the *condition number of A*; for a non-invertible $A \in \mathbb{K}^{m \times m}$ we set $\kappa(A) = \infty$.

Remark. In both cases, we see in the limit $\epsilon \to 0$ of small perturbations that the relative condition number κ of the linear system of equations is bounded by $\kappa \leqslant \kappa(A)$. Actually, equality holds true if just the matrix A is perturbed (cf. §11.9 as well as (16.4)).

Exercise. Show that given a *componentwise* perturbation of $Ax = b$ where

$$\tilde{b} = b + r, \quad |r| \leqslant \epsilon |b|, \qquad \text{or} \qquad \tilde{A} = A + E, \quad |E| \leqslant \epsilon |A|,$$

the relative error of the perturbed result $\tilde{x}$ is bounded by

$$\frac{\|\tilde{x} - x\|_\infty}{\|x\|_\infty} \dot{\leqslant} \operatorname{cond}(A, x) \, \epsilon, \qquad \operatorname{cond}(A, x) = \frac{\| \, |A^{-1}| \, |A| \, |x| \, \|_\infty}{\|x\|_\infty}.$$

This *Skeel–Bauer condition number* $\operatorname{cond}(A, x)$ of a linear system of equations further satisfies

$$\operatorname{cond}(A, x) \leqslant \operatorname{cond}(A) = \| \, |A^{-1}| \, |A| \, \|_\infty \leqslant \kappa_\infty(A) = \|A^{-1}\|_\infty \|A\|_\infty.$$

Construct a matrix A where $\kappa_\infty(A) \gg 1$ that nevertheless satisfies $\operatorname{cond}(A) \approx 1$.

11.9 Geometrically speaking, the condition number of a matrix A describes the distance to the *nearest* non-invertible (short: *singular*) matrix:

Theorem (Kahan 1966). *For the condition number $\kappa(A)$ belonging to an induced matrix norm it holds that*

$$\kappa(A)^{-1} = \min \left\{ \frac{\|E\|}{\|A\|} : A + E \text{ is singular} \right\}.$$

This is consistent with the extension of the definition to $\kappa(A) = \infty$ for a singular A.

Proof. If $A + E$ is singular, there is an $x \neq 0$ where $(A + E)x = 0$, which implies

$$0 < \|x\| = \|A^{-1}Ex\| \leqslant \|A^{-1}\| \, \|E\| \, \|x\| = \kappa(A) \frac{\|E\|}{\|A\|} \|x\|$$

and therefore via division $\kappa(A)^{-1} \leqslant \|E\| / \|A\|$.

In a second step, we must construct an E such that equality holds (for the sake of simplicity, we will restrict ourselves to the $\| \cdot \|_2$-norm). To this end, we choose a vector y normalized to $\|y\|_2 = 1$, for which the maximum is taking in

$$\|A^{-1}\|_2 = \max_{\|u\|_2 = 1} \|A^{-1}u\|_2 = \|A^{-1}y\|_2.$$

We set $x = A^{-1}y \neq 0$ and select the perturbation of A as the outer product

$$E = -\frac{y \, x'}{x'x}.$$

The spectral norm can be calculated as (we will leave step (a) as an exercise)

$$\|E\|_2 \overset{\text{(a)}}{=} \frac{\|y\|_2 \|x\|_2}{\|x\|_2^2} = \frac{1}{\|x\|_2} = \frac{1}{\|A^{-1}\|_2}, \qquad \frac{\|E\|_2}{\|A\|_2} = \kappa_2(A)^{-1}.$$

But now $A + E$ is singular, because $(A + E)x = y - y = 0$ and $x \neq 0$. □

Exercise. Carry out the second half of the proof for the matrix norms $\| \cdot \|_1$ and $\| \cdot \|_\infty$.

11.10 If a matrix A is subjected to an *uncertainty* of relative error ϵ while having a large condition number $\kappa_2(A) \geqslant \epsilon^{-1}$, it then could actually "represent" some singular matrix $\tilde{A}$ as provided by Kahan's theorem (think of an uncertainty as a kind of blur that makes things indistinguishable). Hence, we define:

Matrices A with $\kappa_2(A) \geqslant \epsilon^{-1}$ are called ϵ-singular.[33]

[33] Anticipating the discussion in §12.6 we put on record that ϵ-singular matrices are called *numerically singular* if $\epsilon \leqslant \epsilon_{\text{mach}}$, where ϵ_{mach} is the machine precision to be defined in Lemma 12.2.

12 Machine Numbers

12.1 A t-digit floating point number $\xi \neq 0$ of base $\beta \in \mathbb{N}_{\geqslant 2}$ is written as

$$\xi = \pm \underbrace{d_1.d_2 \cdots d_t}_{\text{Mantissa}} \times \beta^e \quad \text{with digits} \quad d_k \in \{0, 1, \ldots, \beta - 1\};$$

where the exponent $e \in \mathbb{Z}$ is *normalized* such that the *leading digit* satisfies $d_1 \neq 0$.[34] Alternatively, we can also represent such ξ *uniquely* in the form

$$\xi = \pm m \cdot \beta^{e+1-t}, \qquad m \in \{\beta^{t-1}, \beta^{t-1} + 1, \ldots, \beta^t - 1\} \subset \mathbb{N}, \quad e \in \mathbb{Z}.$$

We will call the set of all such floating point numbers, together with zero, *machine numbers* $\mathbb{F} = \mathbb{F}_{\beta,t}$.

12.2 The operation of *rounding* $\mathrm{fl} : \mathbb{R} \to \mathbb{F}$ maps $\xi \in \mathbb{R}$ to the *nearest* machine number $\mathrm{fl}(\xi) = \hat{\xi} \in \mathbb{F}$.[35] Rounding is obviously

- *monotone* $\quad \xi \leqslant \eta \Rightarrow \mathrm{fl}(\xi) \leqslant \mathrm{fl}(\eta)$;
- *idempotent* $\quad \mathrm{fl}(\xi) = \xi$ for $\xi \in \mathbb{F}$.

Lemma. *The relative error of rounding* $\mathrm{fl}(\xi) = \hat{\xi}$ *is bounded by the so-called machine precision*[36] $\epsilon_{\text{mach}} = \frac{1}{2}\beta^{1-t}$,

$$\frac{|\hat{\xi} - \xi|}{|\xi|} \leqslant \epsilon_{\text{mach}}.$$

Equivalently, $\hat{\xi} = \xi(1 + \epsilon)$ *where* $|\epsilon| \leqslant \epsilon_{\text{mach}}$.

Proof. Without loss of generality let $\xi > 0$ which implies that $0 < \xi_0 \leqslant \xi < \xi_1$ for two consecutive machine numbers $\xi_0, \xi_1 \in \mathbb{F}$. Then

$$\hat{\xi} \in \{\xi_0, \xi_1\} \quad \text{with} \quad |\hat{\xi} - \xi| \leqslant (\xi_1 - \xi_0)/2$$

and from the representation $\xi_k = (m + k)\beta^{e+1-t}$ with $\beta^{t-1} \leqslant m < \beta^t$ it follows

$$\frac{|\hat{\xi} - \xi|}{|\xi|} \leqslant \frac{1}{2}\frac{\xi_1 - \xi_0}{\xi_0} = \frac{1}{2m} \leqslant \frac{1}{2}\beta^{1-t} = \epsilon_{\text{mach}}.$$

The second statement is only a rewording of the first with $\epsilon = (\hat{\xi} - \xi)/\xi$. □

[34] When $\beta = 2$, the leading digit is always $d_1 = 1$. As *hidden bit* it does not need to be stored.

[35] If there is a tie between two possibilities, we choose the one with an *even* d_t. Being statistically unbiased, this rounding mode is generally preferred and is called *round-to-even*.

[36] In the literature this quantity is also called *machine epsilon* or *unit roundoff*, often denoted by u.

12.3 On a computer, the exponent e is restricted to a *finite* set

$$\{e_{\min}, e_{\min}+1, \dots, e_{\max}-1, e_{\max}\} \subset \mathbb{Z}.$$

Numbers $\hat{\xi} \in \mathbb{F}$ with an absolute value that is too large or small can therefore no longer be represented and can lead to exponent *overflow* or *underflow* respectively. As a default, computers "catch" these limiting cases with assigning the values $\pm\infty$ and zero without any notice. By applying some care, over- and underflow do not pose all too great a threat in numerical linear algebra (*except* for determinants).

12.4 Essentially every computer since 1985 implements the IEEE 754 standard, which offers two binary formats of *hardware arithmetic*:[37]

single precision

32 bits = 4 bytes with the storage scheme

$s:1$	$e:8$	$f:23$

$$\xi = \begin{cases} (-1)^s\, 2^{e-127} \times 1.f & e = 1:254 \\ (-1)^s\, 2^{-126} \times 0.f & e = 0 \\ (-1)^s \infty & e = 255, f = 0 \\ \texttt{NaN} & e = 255, f \neq 0 \end{cases}$$

base β:	2
mantissa length t:	24
overflow threshold:	$2^{127}(2-2^{-23}) \approx 3.4 \times 10^{38}$
underflow threshold:	$2^{-126} \approx 1.2 \times 10^{-38}$
precision ϵ_{mach}:	$2^{-24} \approx 5.96 \times 10^{-8}$

corresponds to approximately 8 decimal places

double precision

64 bits = 8 bytes with the storage scheme

$s:1$	$e:11$	$f:52$

$$\xi = \begin{cases} (-1)^s\, 2^{e-1023} \times 1.f & e = 1:2046 \\ (-1)^s\, 2^{-1022} \times 0.f & e = 0 \\ (-1)^s \infty & e = 2047, f = 0 \\ \texttt{NaN} & e = 2047, f \neq 0 \end{cases}$$

base β:	2
mantissa length t:	53
overflow threshold:	$2^{1023}(2-2^{-52}) \approx 1.8 \times 10^{308}$
underflow threshold:	$2^{-1022} \approx 2.2 \times 10^{-308}$
precision ϵ_{mach}:	$2^{-53} \approx 1.11 \times 10^{-16}$

corresponds to approximately 16 decimal places

IEEE 754 also recognizes the symbols

- $\pm\infty$, e.g., as the value of $\pm 1/0$;
- `NaN` *not a number*, e.g., as the result of $0/0$ or $\infty - \infty$.

Both alleviate and standardize handling of arithmetic exceptions.

12.5 IEEE 754 furthermore defines that arithmetic operations and the square root are computed for machine numbers by *correctly rounding* the exact result: if we denote the realization of an operation $\star$ by $\hat{\star}$ we get for $\xi, \eta \in \mathbb{F}$ that

$$\xi \,\hat{\star}\, \eta = \mathrm{fl}(\xi \star \eta) \qquad (\star \in \{+, -, \cdot, /, \sqrt{}\}).$$

Hence, Lemma 12.2 implies the following *standard model for machine numbers*:

[37] By default, MATLAB uses double precision, but also allows the use of single precision.

For $\xi, \eta \in \mathbb{F}$ and $\star \in \{+, -, \cdot, /, \sqrt{}\}$ there is an $|\epsilon| \leqslant \epsilon_{\text{mach}}$, such that

$$\xi \mathbin{\hat{\star}} \eta = (\xi \star \eta)(1 + \epsilon) \tag{12.1}$$

for the machine-realization $\hat{\star}$ of the operation $\star$.

Remark. Both the associative law and the distributive law loose there validity in machine arithmetic: parentheses should therefore be placed very deliberately (since otherwise "dramatic" consequences could follow as we will see in §14.4).

Exercise. In order to realize complex arithmetic with real operations, how must ϵ_{mach} be adjusted for the standard model to remain valid in $\mathbb{K} = \mathbb{C}$?

12.6 The first rounding errors can occur as early as during the *input* of data into the computer. This is, for example, because the simplest decimal fractions are already be represented by *non-terminating* binary fractions, such as

$$0.1_{10} = 0.000\overline{1100}_2,$$

which therefore must be rounded to a machine number. In the case of data x (this can be a tuple of matrices) and a problem $x \mapsto f(x)$ as in §11.1, this causes an input of machine numbers $\hat{x} = \text{fl}(x)$ with a *componentwise* relative error

$$|\hat{x} - x| \leqslant \epsilon_{\text{mach}} |x|.$$

If we first apply a monotone norm and then the equivalences from §C.9, this means that with respect to *every* relative error measure there is an input perturbation of the form[38]

$$[\![\hat{x} - x]\!]_{\text{rel}} = O(\epsilon_{\text{mach}}).$$

This kind of input perturbation leads to *unavoidable* perturbations of the result; even when the computer calculations after data input were *exact*. With an appropriate condition number, §11.4 provides the estimate

$$[\![f(\hat{x}) - f(x)]\!] = O(\kappa(f; x)\, \epsilon_{\text{mach}}).$$

[38] The Landau symbol $O(\epsilon_{\text{mach}})$ stands for a bound of the form

$$|O(\epsilon_{\text{mach}})| \leqslant c\epsilon_{\text{mach}} \qquad (\epsilon_{\text{mach}} \leqslant \epsilon_0)$$

with some constant $c > 0$. Here, for *statements* concerning problem classes and algorithms, let us agree that c does not depend on any specific data or the machine precision ϵ_{mach}, but may well polynomially depend on the dimensions of the problem at hand.

When *assessing* concrete instances (of the problem and ϵ_{mach} at hand), we will accept such constants c (as well as ratios $c = a/b$ for comparisons of the form $a \not\gg b$) which, for example, allow for a maximum loss of accuracy of one third of the mantissa length:

$$c \leqslant \epsilon_{\text{mach}}^{-1/3}.$$

When working with such assessments, always remember: "your mileage may vary"—the principle, however, should have become clear.

13 Stability of an Algorithm

13.1 An *algorithm* for the evaluation of f is ultimately a decomposition

$$f = \underbrace{f_s \circ \cdots \circ f_1}_{\text{algorithm}}$$

into sequentially executed computational steps such as elementary arithmetic operations, Level-n BLAS or other *standardized* operations. However, when compared with the map f, on the computer the use of machine numbers leads to a perturbed map[39]

$$\hat{f} = \hat{f}_s \circ \cdots \circ \hat{f}_1,$$

whereby $\hat{f}_j$ represents the execution of the standardized operations f_j under consideration of all *rounding errors* involved.

13.2 An algorithm $\hat{f}$ for the evaluation of the problem f is called *stable*, if

$$[\![\hat{f}(x) - f(\tilde{x})]\!] = O(\epsilon_{\text{mach}})$$

for suitably perturbed inputs $\tilde{x}$ (depending on the actual input x) within the scope of machine precision

$$[\![\tilde{x} - x]\!] = O(\epsilon_{\text{mach}});$$

otherwise it is *unstable*. According to Trefethen and Bau, we can formulate this as:

> *A stable algorithm gives nearly the right answer to nearly the right question.*

Remark. A stable algorithm for the problem f therefore behaves completely *comparably* to the sequence $\text{fl} \circ f \circ \text{fl}$, which corresponds to the minimum amount of rounding needed.

13.3 Many algorithms in numerical linear algebra fulfill a concept, that is both stronger and simpler than stability. An algorithm $\hat{f}$ for the evaluation of the problem f is called *backward stable* if actually

$$\hat{f}(x) = f(\tilde{x}) \quad \text{for a perturbed input } \tilde{x} \text{ with } \quad [\![\tilde{x} - x]\!] = O(\epsilon_{\text{mach}}).$$

We call such $\tilde{x} - x$ the *backward error* of the algorithm $\hat{f}$; comparing the backward error with the machine precision is called *backward analysis*. According to Trefethen and Bau we can formulate this all as:

> *A backward stable algorithm gives exactly the right answer to nearly the right question.*

[39] Due to the dependence on the considered algorithm, there is no "absolute", unique $\hat{f}$. We therefore always identify $\hat{f}$ as the realization of a concrete algorithm using a given machine arithmetic.

Example. The standard model (12.1) of machine arithmetic implies that the arithmetic operations are realized as backward stable. For $\xi, \eta \in \mathbb{F}$ it holds actually

$$\begin{aligned} \xi \mathbin{\hat{\pm}} \eta &= \tilde{\xi} \pm \tilde{\eta} \\ \xi \mathbin{\hat{\cdot}} \eta &= \tilde{\xi} \cdot \eta \\ \xi \mathbin{\hat{/}} \eta &= \tilde{\xi} / \eta \end{aligned}$$

where $\tilde{\xi} = \xi(1+\epsilon)$, and $\tilde{\eta} = \eta(1+\epsilon)$, for suitably chosen $|\epsilon| \leqslant \epsilon_{\text{mach}}$.

Remark. When correctly rounded, $\hat{\pm}$ is actually *exact* in the case of cancellation (11.2).

Exercise. Show that the square root is realized as backward stable in the standard model.

13.4 Now we ask, how *accurate* is the result of a *stable* algorithm? Since we can estimate the accuracy of $f(\tilde{x})$ according to §11.4, backward stability yields directly the *error estimate*[40]

$$[\![\hat{f}(x) - f(x)]\!] = O(\kappa(f;x)\,\epsilon_{\text{mach}});$$

we call $\hat{f}(x) - f(x)$ the *forward error* of the algorithm $\hat{f}$.

It would be futile to require a *greater* accuracy from an algorithm since, according to §12.6 the input into the computer of x alone creates a perturbed $\hat{x}$ whose *exact* result were subject to the same error estimate:

$$[\![f(\hat{x}) - f(x)]\!] = O(\kappa(f;x)\,\epsilon_{\text{mach}}).$$

Comparing the forward error of an algorithm with the unavoidable error determined by the conditioning of the problem is called *forward analysis*.

Stability Analysis of Matrix Products[41]

13.5 The inner product $\pi_m = y'x$ of two machine vectors $x, y \in \mathbb{F}^m$ can be realized recursively with the following simple algorithm:

$$\hat{\pi}_0 = 0, \qquad \hat{\pi}_k = \hat{\pi}_{k-1} \mathbin{\hat{+}} \eta_k \mathbin{\hat{\cdot}} \xi_k \quad (k = 1:m).$$

According to the standard model, we write this in the form

$$\hat{\pi}_k = \Big(\hat{\pi}_{k-1} + (\eta_k \cdot \xi_k)(1+\epsilon_k)\Big)(1+\delta_k) \qquad (k = 1:m)$$

with the relative errors $|\epsilon_k|, |\delta_k| \leqslant \epsilon_{\text{mach}}$ (where $\delta_1 = 0$). If we collect all of these errors as an *backward error* of the components of y, the expanded result is

$$\hat{\pi}_m = \tilde{y}' \cdot x, \qquad |\tilde{y} - y| \dot{\leqslant} m\,\epsilon_{\text{mach}}|y|,$$

[40] The same result holds for *stable* algorithms, if $\kappa(f;x)^{-1} = O(1)$ (which is usually the case).

[41] Notation as in §§2.8 and 2.10

where $\tilde{\eta}_k = \eta_k(1+\epsilon_k)(1+\delta_k)\cdots(1+\delta_m) = \eta_k(1+\theta_k)$, so that $|\theta_k| \dot{\leqslant} m\,\epsilon_{\text{mach}}$.[42] Since this all is valid *independent* of the order of the terms, we keep for the record:

Conclusion. *Level-1-BLAS computes backward stable inner products.*

13.6 The matrix vector product $y = Ax$ can be realized *row-wise* as the inner product of $x \in \mathbb{F}^n$ with the row vectors of the matrix $A \in \mathbb{F}^{m\times n}$. If we collect the errors as the backward error of the *row vectors*, the following backward error estimate holds for the machine result $\widehat{y} \in \mathbb{F}^m$

$$\widehat{y} = (A+E)x, \qquad |E| \dot{\leqslant} n\,\epsilon_{\text{mach}}|A|.$$

Conclusion. *Level-2-BLAS computes backward stable matrix vector products.*

13.7 For $C = AB$ with $A \in \mathbb{F}^{m\times n}$, $B \in \mathbb{F}^{n\times p}$ we obtain *column-wise*

$$\widehat{c}^j = (A+E_j)b^j, \quad |E_j| \dot{\leqslant} n\,\epsilon_{\text{mach}}|A| \qquad (j = 1:p).$$

The column vectors $\widehat{c}^j$ are therefore computed *backward stably*. In general, this does not apply to the computed product matrix $\widehat{C}$ itself.

Example. For an outer product $C = xy'$, the value $\widehat{C}$ is generally *no longer* a rank 1 matrix.

Fortunately enough, we obtain this way an estimate of the forward error that lies within the order of the *unavoidable* error, cf. (11.4):

$$|\widehat{C} - C| \dot{\leqslant} n\,\epsilon_{\text{mach}}|A|\cdot|B|.$$

If $A \geqslant 0$, $B \geqslant 0$ componentwise, it holds $|A|\cdot|B| = |C|$ and the result $\widehat{C}$ behaves as the input of C subject to a machine precision of $n\epsilon_{\text{mach}}$.

Simple Criteria for the Analysis of Numerical Stability

13.8 Comprehensive error analysis as presented in the prior pages can quickly become tedious. In order to spot the weak points of an algorithm that would seriously endanger stability, we consider the *error transport* within an algorithm sectionwise by writing

$$f = \underbrace{f_s \circ \cdots \circ f_{k+1}}_{=h} \circ \underbrace{f_k \circ \cdots \circ f_1}_{=g} = h \circ g;$$

we call h an *end section* and g a *start section* of the algorithm. The total error of $\hat{f}$ is—either considered as a forward or as a backward error—in leading order the *superposition* of the contributions created by just rounding *between* such sections:

$$f_* = h \circ \text{fl} \circ g.$$

[42] Due to $\delta_1 = 0$ there is a maximum of m factors of the form $(1+\alpha)$ with $|\alpha| \leqslant \epsilon_{\text{mach}}$ to be found in the expression η_k .

If these contributions do not "miraculously" cancel each other out (which one should not count on), any large individual contribution will already be responsibly for *instability*. In this sectionwise analysis, the intermediate rounding $\hat{z} = \mathrm{fl}(g(x))$ in f_* contributes

- to the forward error by

$$[\![f_*(x) - f(x)]\!] = O(\kappa(h; g(x))\, \epsilon_{\mathrm{mach}}),$$

- to the backward error (for *invertible* g) by

$$[\![x_* - x]\!] = O(\kappa(g^{-1}; g(x))\, \epsilon_{\mathrm{mach}})$$

  where the data $x_* = g^{-1}(\hat{z})$ is *reconstructed* from the intermediate result $\hat{z}$.

Forward analysis (F) and backward analysis (B) directly provide the following:

Instability Criteria. *Important indicators for the instability of an algorithm with a section decomposition $f = h \circ g$ are*[43]

F: *an ill-conditioned end section h with $\kappa(h; g(x)) \gg \kappa(f; x)$;*

B: *an inversely ill-conditioned start section g with $\kappa(g^{-1}; g(x)) \gg 1$.*

Remark. Actual *evidence* of instability must always be based on a concrete numerical example.

Exercise. Show that for *scalar* start and end sections g and h Criteria F and B are *equivalent* with respect to componentwise relative errors; it then actually holds

$$\kappa(g^{-1}; g(x)) = \frac{\kappa(h; g(x))}{\kappa(f; x)}. \tag{13.1}$$

13.9 By definition of the condition number, an algorithm $\hat{f}$ given by the decomposition

$$f = f_s \circ \cdots \circ f_2 \circ f_1$$

leads directly to a multiplicative upper bound of the condition number of f, i.e.,

$$\kappa(f; x) \leqslant \kappa(f_s; f_{s-1}(\cdots)) \cdots \kappa(f_2; f_1(x)) \cdot \kappa(f_1; x),$$

which is generally a *severe over-estimation*: it represents the worst case error amplification of every single step but these critical cases do not necessarily match one another. The estimate is therefore neither suitable for estimating the condition of the problem f, nor for judging the stability of the algorithm $\hat{f}$. One *exception* is notably where effectively *everything* is benign:

[43] When $a, b > 0$, $a \gg b$ is equivalent to $a/b \gg 1$. By doing so, we consistently choose one and the same "pain threshold" as our interpretation of "very large".

Stability Criterion for "Short" Algorithms. *If it simultaneously holds that*

(1) *all* $\hat{f}_j$ *are stable,*

(2) *all* f_j *are well-conditioned,*

(3) *the number of steps* s *is small,*

then $\hat{f} = \hat{f}_s \circ \cdots \circ \hat{f}_1$ *is a stable algorithm and* f *a well-conditioned problem.*

Example. Conditions (1) and (2) are satisfied with respect to componentwise relative errors if the f_j consist exclusively of the *benign* operations of the standard model, that is, if they do *not* contain a genuine subtraction.[44] If

$$s = s_{\text{add}} + s_{\text{sub}} + s_{\text{mult}} + s_{\text{div}} + s_{\text{quad}}$$

represents the decomposition into the number of *genuine* additions, *genuine* subtractions, multiplications, divisions and square roots, then, according to §11.6, the following condition number estimate for the problem f is valid:[45]

$$s_{\text{sub}} = 0 \quad \Rightarrow \quad \kappa(f;x) \leqslant 2^{s_{\text{mult}}+s_{\text{div}}}. \tag{13.2}$$

Genuine subtractions could be problematic in the case of *cancellation*; one should either *avoid* them altogether or carefully *justify* them with an expert analysis.

Exercise. Compare (13.2) with (11.3) for $x'y$ if $x, y \in \mathbb{R}^m$ have positive components.

14 Three Exemplary Error Analyses

In this section we will restrict ourselves to componentwise relative errors.

Error Analysis 1: Quadratic Equation

14.1 In junior high school, one learns the quadratic equation $x^2 - 2px - q = 0$ and its textbook solution formula (which is an algorithm if taken literally)

$$x_0 = p - \sqrt{p^2+q}, \qquad x_1 = p + \sqrt{p^2+q}.$$

In order to avoid a tedious case-by-case study, we will limit ourselves to $p, q > 0$. The error analysis can be done by a mere "clever inspection", i.e., basically without any calculations that would go beyond simple counting:

- According to the stability criterion for short algorithms, the formula for x_1 is *stable*; additionally, (13.2) proves *well*-conditioning: $\kappa(x_1; p, q) \leqslant 2$.

[44] We identify $\xi \pm \eta$ for $|\xi \pm \eta| = |\xi| + |\eta|$ as genuine addition, otherwise as genuine subtraction.

[45] For square roots we have to set $\kappa = 1$ instead of $\kappa = 1/2$, since generally data and intermediate results remain *stored* during execution and their perturbations remain untouched.

- The formula for x_0 exhibits the section decomposition

$$f : (p,q) \overset{g}{\longmapsto} \left(p, \sqrt{p^2+q}\right) = (p,r) \overset{h}{\longmapsto} p - r = x_0.$$

For $q \ll p^2$, the subtraction in the end section h is afflicted with *cancellation* and thus *ill*-conditioned (cf. §11.6): therefore, by Criterion F, the formula is presumably at risk of being *unstable* (more precise information can only be acquired via conditioning analysis of $f : (p,q) \mapsto x_0$).

Example. A numerical example in MATLAB (using double precision) is illustrating this very clearly:

```
>> p = 400000; q=1.234567890123456;
>> r = sqrt(p^2+q); x0 = p - r
x0 =
    -1.543201506137848e-06
```

Here, actually $11 = 6+5$ decimal places are lost by cancellation, so that the computed solution `x0` only contains *at most* approximately $5 = 16-11$ correct decimals places.[46] We still however have to clarify whether to blame the algorithm for this loss of accuracy or, after all, the ill-conditioning of $f : (p,q) \mapsto x_0$.

- The inverse start section $g^{-1} : (p,r) \mapsto (p,q)$ is also subject to *cancellation* for $q \ll p^2$, in particular the reconstruction of the coefficient

$$q = r^2 - p^2.$$

Here, accurate information about q is *lost* through g.[47] By Criterion B, the formula for x_0 is therefore at risk of being *unstable*.

Example. In the numerical example, we expect the reconstruction of q to exhibit a loss (that is, a backward error) of $11 = 2 \cdot 6 - 1$ decimal places, i.e., once again we get only approximately $5 = 16 - 11$ correct decimal places (which upon comparison with q can instantly be observed):

```
>> r^2-p^2
ans =
     1.234558105468750e+00
```

Since the backward and forward errors are both of the same magnitude (a loss of 11 decimal places), the map $f : (p,q) \mapsto x_0$ should be well-conditioned.

[46] The further 11 decimal places of "numerical garbage" arise through the conversion of `x0` from a binary number in where zeros have been appended to the end of the mantissa (here: concretely 38 bits): the MATLAB command `num2hex(x0)` reveals beb9e40000000000.

[47] Since this *loss of information* in q does *not* affect the stability of the formula for x_1, it must be the case that for $q \ll p^2$, the value of x_1 is quite *immune* to a perturbation of q: in fact, one calculates that $\kappa(x_1;q) = (1 - p/\sqrt{p^2+q})/2 \ll 1$.

14.2 A *stable* algorithm for x_0 must therefore (a) eventually get along without subtraction, and (b) use the information in the coefficients q in a much more *direct fashion*. The factorization $x^2 - 2px - q = (x - x_0)(x - x_1)$ ultimately provides us with a further formula for x_0, which can do both (called *Vieta's formula*):

$$x_0 = -q/x_1, \qquad x_1 = p + \sqrt{p^2 + q}.$$

As with the formula for x_1, it is stable according to the stability criterion for short algorithms; (13.2) now proves $(p, q) \mapsto x_0$ to be *well*-conditioned: $\kappa(x_0; p, q) \leqslant 4$.

Example. In the numerical example, this looks as follows:

```
>> x1 = p + r;
>> x0 = -q/x1
x0 =
   -1.543209862651343e-06
```

At this point all 16 shown decimal places should be (and are) correct (due to the stability of the formula *and* the well-conditioning of the problem); a comparison with the results of the "textbook formula" for x_0 in §14.1 now verifies that, as predicted, only 5 decimal places of the latter were correct:

In the case of cancellation, $q \ll p^2$, the "textbook formula" for x_0 is *unstable*.

Remark. A detailed conditioning analysis of both solution maps $(p, q) \mapsto x_0$ and $(p, q) \mapsto x_1$ is actually no longer necessary. The condition formula (11.1) yields

$$\kappa(x_0; p, q) = \frac{1}{2} + \frac{3p}{2\sqrt{p^2 + q}} \leqslant 2, \qquad \kappa(x_1; p, q) = \frac{1}{2} + \frac{p}{2\sqrt{p^2 + q}} \leqslant 1;$$

our expert analysis "by inspection" overestimated the upper bounds by just a factor of 2.

Exercise. When $q < 0$ and $p \in \mathbb{R}$, discuss the cancellation case $0 < p^2 + q \ll p^2 + |q|$.

Error Analysis 2: Evaluation of $\log(1 + x)$

14.3 In a safe distance to the singularity at $x = -1$, the function $f(x) = \log(1 + x)$ is *well-conditioned*:

$$\kappa(f; x) = \frac{|f'(x)|\,|x|}{|f(x)|} = \frac{x}{(1 + x)\log(1 + x)} \leqslant 2 \qquad (x \geqslant -0.7).$$

Let us take the expression $\log(1 + x)$ literally as an *algorithm* by decomposing it to

$$f : x \xmapsto{g} 1 + x = w \xmapsto{h} \log w.$$

This exposes the risk of *instability* for $x \approx 0$ in accordance with both instability criteria, Criterion F and B:[48]

[48] Recall that they are equivalent for scalar functions, cf. (13.1).

F: In the vicinity of its *root* $w = 1$, $h(w) = \log(w)$ is *ill-conditioned*.[49]

B: When $w \approx 1$, the expression $g^{-1}(w) = w - 1$ experiences cancellation, and is therefore *ill*-conditioned (information about $x \approx 0$ gets lost in $w = 1 + x$.)

Example. A numerical example exhibits the *instability*:

```
>> x = 1.234567890123456e-10;
>> w = 1+x; f = log(w)
f =
       1.23456800330697e-10
>> w-1
ans =
       1.23456800338317e-10
```

For $w - 1$, a total of $10 = 1 + 9$ digits are canceled out, leaving only $6 = 16 - 10$ decimals places correct (backward error). Due to $\kappa(f; x) \approx 1$, only approximately 6 decimals places in f are expected to be correct (forward error).

14.4 A *stable* algorithm for f must therefore (a) process the intermediate result w in a much better conditioned fashion and (b) use the information in x in a more *direct* way. To this end, Velvel Kahan came up with the following "twisted" idea:

$$w = 1 + x, \qquad f(x) = \begin{cases} \dfrac{\log w}{w - 1} \cdot x & w \neq 1, \\ x & w = 1. \end{cases}$$

The trick is that the section $\phi : w \mapsto \log(w)/(w - 1)$ is actually consistently *well*-conditioned (and the 1 is *not* subject to rounding errors on the computer):

$$\kappa(\phi; w) = \frac{|\phi'(w)|\,|w|}{|\phi(w)|} = \frac{1 - w + w \log w}{(w - 1) \log w} \leqslant 1 \qquad (w > 0).$$

The cancellation in the denominator $w - 1$ of ϕ is therefore canceled away by the perfectly *correlated* imprecision of $\log w$ in the numerator:

> *In a given algorithm, imprecise intermediate results are allowed every time their errors are subsequently compensated for.*

According to the stability criterion for short algorithms, Kahan's idea is *stable*.[50]

Example. Using the numerical values from above, Kahan's idea yields:

```
>> f = log(w)/(w-1)*x
f =
       1.23456789004725e-10
```

[49] As is the case with functions in the vicinity of their roots in general if we use *relative* errors.
[50] As long as the library routine for $\log w$ is stable.

In this case, (due to the stability of Kahan's algorithm *and* the well-conditioning of the problem) almost all 16 shown decimal places should be correct (they all are). A comparison with the result of the "naive" expression in §14.3 verifies that, as predicted, only 6 decimal places of the latter were correct.[51]

Remark. Kahan's algorithm serves as a "dramatic" example of the invalidity of the associative law in machine arithmetic: $(1 \mathbin{\widehat{+}} x) \mathbin{\widehat{-}} 1 \neq x$ for $0 \approx x \in \mathbb{F}$. It would therefore be a profound *blunder* to "simplify" a program such as

```
log(1+x)*x/((1+x)-1)
```

to `log(1+x)`. Unfortunately, there are programming languages (e.g., Java) that offer such bizarre "performance features"; Kahan fought for years against this very thing.[52]

Error Analysis 3: Sample Variance

14.5 Descriptive statistics offer two rival formulas (that is, algorithms if taken literally) for the sample variance S^2 of a real data set $x = (x_1, \ldots, x_m)' \in \mathbb{R}^m$:

$$S^2 \overset{\text{(a)}}{=} \frac{1}{m-1}\sum_{j=1}^{m}(x_j - \bar{x})^2 \overset{\text{(b)}}{=} \frac{1}{m-1}\left(\sum_{j=1}^{m} x_j^2 - \frac{1}{m}\left(\sum_{j=1}^{m} x_j\right)^2\right), \qquad \bar{x} = \frac{1}{m}\sum_{j=1}^{m} x_j.$$

Formula (a) requires *two* passes over the data: one to initially calculate the sample mean $\bar{x}$, and then a further pass to accumulate the sum $\sum_j (x_j - \bar{x})^2$. If the sample variance is to be calculated *while* new data are being generated, many statistics textbooks will recommend to alternatively use formula (b): in this case, just a *single pass* is necessary, whereby both sums $\sum_j x_j$ and $\sum_j x_j^2$ are accumulated.

Unfortunately, formula (b) is numerically *unstable*, while formula (a) is *stable*:[53]

- The end section of formula (b) is executing (aside from the well-conditioned and stable, and thus harmless division by $m-1$) a subtraction. In the case of cancellation, i.e., the case of a relatively small variance

  $$S^2 \ll \bar{x}^2,$$

  this subtraction is actually *ill*-conditioned. Hence, according to Criterion F, formula (b) is presumably at risk of being *unstable*.

  Example. For an actual *demonstration* of instability, numerical examples can easily be found in which formula (b) returns a ridiculous *negative* result in machine arithmetic:

```
>> x = [10000000.0; 10000000.1; 10000000.2]; m = length(x);
>> S2 = (sum(x.^2)-sum(x)^2/m)/(m-1)
S2 =
     -3.125000000000000e-02
```

[51] MATLAB offers Kahan's algorithm for $\log(1+x)$ using the command `log1p(x)`.

[52] W. Kahan, J. D. Darcy: *How Java's Floating-Point Hurts Everyone Everywhere*, UC Berkeley, 1998–2004.

[53] For a stable *single-pass* algorithm see T. F. Chan, G. H. Golub, R. J. LeVeque: *Algorithms for computing the sample-variance: analysis and recommendations*. Amer. Statist. 37, 242–247, 1983.

The cancellation of $17 = 2 \cdot 8 + 1 > 16$ decimal places explains how the result could become such a "total loss".

- In contrast, formula (a) contains the (genuine) subtractions in its *start section*,

$$g : (x_1, \ldots, x_m) \mapsto (x_1 - \bar{x}, \ldots, x_m - \bar{x}) = (\delta_1, \ldots, \delta_m).$$

Here, the reconstruction of data by

$$g^{-1} : (\delta_1, \ldots, \delta_m) \mapsto (\delta_1 + \bar{x}, \ldots, \delta_m + \bar{x})$$

is the decisive factor for Criterion B. Since it is *well*-conditioned for small fluctuations $|\delta_j| = |x_j - \bar{x}| \ll |\bar{x}|$, the start section does *not* pose a threat to the stability. As a sum over m non-negative terms, the end section

$$h : (\delta_1, \ldots, \delta_m) \mapsto \sum_{j=1}^{m} \delta_j^2$$

is *well*-conditioned and is easily coded on the computer in a backward stable fashion. Formula (a) is numerically *stable*, indeed.

Example. In the numerical example, we get the following:

```
>> xbar = mean(x);
>> S2 = sum((x-xbar).^2)/(m-1)
S2 =
      9.999999925494194e-03
```

A quick mental calculation provides $S^2 = 0.01$ so long as we regard the data set x as *exact* decimal fractions. The loss of 8 decimal places in our stably calculated solution must therefore be due to an *ill-conditioning* of about $\kappa(S^2; x) \approx 10^8$.

Without knowledge of the "exact" result (which, of course, we would never know when doing *serious* computations), an evaluation of the *accuracy* of a *stably* calculated result requires at least a rough estimate of the conditioning of the problem.

14.6 The componentwise relative condition number κ of the sample variance S^2 relative to the data set $x \in \mathbb{R}^m$ can be directly calculated with (11.1) as[54]

$$\kappa(S^2; x) = \frac{2}{(m-1)S^2} \sum_{j=1}^{m} |x_j - \bar{x}| \, |x_j| \leqslant \frac{2\|x\|_2}{S\sqrt{m-1}}.$$

Example. Hence, the numerical example *confirms* finally that $\kappa \approx 2 \cdot 10^8$:

```
>> kappa = 2*abs((x-xbar))'*abs(x)/S2/(m-1)
kappa =
    2.0000e+08
```

In this case, the loss of approximately 8 decimal places was therefore *unavoidable*.

Remark. Notice that only the *order of magnitude* of a condition number is relevant information.

[54] The upper bound is found with the help of Cauchy–Schwarz inequality (T. F. Chan, J. G. Lewis 1978).

15 Error Analysis of Linear Systems of Equations

In this section, $A \in \mathrm{GL}(m;\mathbb{K})$ and $b \in \mathbb{K}^m$ and error measures are generally normwise relative.

A posteriori Assessment of Approximate Solutions

15.1 We would like to asses the error of a given approximate solution $\tilde{x} \in \mathbb{K}^m$ of a linear system of equations $Ax = b$. There are two options to look at, forward and backward error, which differ significantly in their accessibility:

- Without knowledge of an exact solution x, the *forward error* $\|\tilde{x} - x\| / \|x\|$ can only be estimated and an assessment must be made in comparison with the unavoidable error in terms of the condition number $\kappa(A)$.
- The *backward error* is defined as the smallest perturbation of the matrix A which makes $\tilde{x}$ *exact*:

$$\omega(\tilde{x}) = \min\left\{ \frac{\|E\|}{\|A\|} : (A+E)\tilde{x} = b \right\};$$

  as seen next, it can be computed without resorting to an exact solution and can be assessed by direct comparison with the accuracy of the *data*.

15.2 There is actually an explicit formula for the backward error.

Theorem (Rigal–Gaches 1967). *The backward error $\omega(\tilde{x})$ of $\tilde{x} \neq 0$ is*

$$\omega(\tilde{x}) = \frac{\|r\|}{\|A\|\,\|\tilde{x}\|}, \qquad r = b - A\tilde{x}. \tag{15.1}$$

Here, r is called the residual of the approximate solution $\tilde{x}$ of $Ax = b$.

Proof. [55] From $(A+E)\tilde{x} = b$, hence $E\tilde{x} = r$, it follows that

$$\|r\| \leqslant \|E\|\|\tilde{x}\|, \qquad \frac{\|r\|}{\|A\|\,\|\tilde{x}\|} \leqslant \frac{\|E\|}{\|A\|}.$$

We must now construct E such that *equality* holds in this estimate. For the sake of simplicity, we limit ourselves to the $\|\cdot\|_2$-norm:

$$E = \frac{r\tilde{x}'}{\tilde{x}'\tilde{x}} \quad \text{with} \quad \|E\|_2 = \frac{\|r\|_2\|\tilde{x}\|_2}{\|\tilde{x}\|_2^2}$$

satisfies $E\tilde{x} = r$ and $\|r\|_2 = \|E\|_2\|\tilde{x}\|_2$. □

Remark. A numerical example for (15.1) and (15.3) can be found in §§15.10 and 15.13.

[55] Notice the similarity to the proof of Theorem 11.9.

Exercise. For *componentwise* analysis, the backward error is similarly defined as

$$\omega_\bullet(\tilde{x}) = \min\{\epsilon : (A+E)\tilde{x} = b, |E| \leqslant \epsilon|A|\}.$$

Prove the *Oettli–Prager Theorem* (1964) (recall the convention in Footnote 30):

$$\omega_\bullet(\tilde{x}) = \max_{j=1:m} \frac{|r_j|}{(|A|\cdot|\tilde{x}|)_j}, \qquad r = b - A\tilde{x}. \tag{15.2}$$

Compare $\omega_\bullet(\tilde{x})$ with the normwise backward error $\omega(\tilde{x})$ (with respect to the ∞-norm).

15.3 From $x - \tilde{x} = A^{-1}r$ and the expression (15.1) of the backward error $\omega(\tilde{x})$ it directly follows a simple *forward error estimate*, cf. (11.5):

$$\frac{\|x-\tilde{x}\|}{\|\tilde{x}\|} \leqslant \frac{\|A^{-1}\|\,\|r\|}{\|\tilde{x}\|} = \kappa(A)\cdot\omega(\tilde{x}). \tag{15.3}$$

Remark. Since the *calculation* of $\kappa(A) = \|A^{-1}\|\cdot\|A\|$ would often be much too costly and the only thing that matters in an estimate like (15.3) is actually the *order of magnitude* of the given bound, in practice, one often uses much more "low-cost" *estimates* of $\kappa(A)$; e.g.,[56]

```
condest(A)   % estimate of κ_1(A)
condest(A') % estimate of κ_∞(A), cf. (C.3)
```

Exercise. Show the alternative forward error estimate

$$\frac{\|x-\tilde{x}\|_\infty}{\|\tilde{x}\|_\infty} \leqslant \mathrm{cond}(A,\tilde{x})\cdot\omega_\bullet(\tilde{x}).$$

using the Oettli–Prager Theorem and the Skeel–Bauer condition number. Given reasons why this estimate is sharper than (15.3) if $\tilde{x}$ has been computed in a componentwise backward stable fashion.

A priori Stability Analysis of a Solution by Matrix Factorization

15.4 In Chapter II we solved the linear system of equations $Ax = b$ with a suitable factorization $A = MN$ (where the right hand side b is supposed to be *fixed*):

$$f : A \overset{g}{\longmapsto} (M,N) \overset{h}{\longmapsto} x.$$

In doing so, the start section g is exactly the factorization step; specifically, one of

- triangular decomposition $P'A = LU$ with pivoting: $M = PL$, $N = U$;
- Cholesky decomposition $A = LL'$ (A s.p.d.): $M = L$, $N = L'$;
- QR decomposition $A = QR$: $M = Q$, $N = R$.

[56] N. J. Higham: *Algorithm 674: FORTRAN codes for estimating the L1-norm of a real or complex matrix, with applications to condition number estimation*, ACM Trans. Math. Software 14, 381–396, 1988.

The end section h contains the computation of x from the factorization, a process which can be realized algorithmically in a *backward stable* fashion by either forward substitution, back substitution or by multiplication with Q': in the end, such triangular or unitary factors $F \in \mathbb{F}^{m\times m}$ yield computed solutions $\hat{u} \in \mathbb{F}^m$ of the system $Fu = v \in \mathbb{F}^m$ which satisfy

$$(F + \Delta F)\hat{u} = v, \qquad \|\Delta F\| = O(\epsilon_{\text{mach}})\|F\|.$$

The proof can be carried out similarly to §§13.5 and 13.6.

15.5 It remains to better understand the start section g. Because of

$$g^{-1} : (M, N) \mapsto M \cdot N$$

its contribution to the *backward error* in A is of order (see §§13.8 and 11.7)

$$O\left(\frac{\|M\| \cdot \|N\|}{\|A\|}\epsilon_{\text{mach}}\right).$$

In fact, all three factorizations (with A s.p.d. if Cholesky decomposition is employed) yield a solution $\hat{x} \in \mathbb{F}^m$ of $Ax = b$ that satisfies[57]

$$(A + E)\hat{x} = b, \qquad \|E\| = O(\|M\|\,\|N\|\,\epsilon_{\text{mach}}) \tag{15.4}$$

for machine data A and b. The factorization based algorithm for the solution of a linear system of equations $Ax = b$ is therefore

- *vulnerable* to instability in the *malignant* case $\|M\| \cdot \|N\| \gg \|A\|$;
- provably *backward stable* in the *benign* case $\|M\| \cdot \|N\| \approx \|A\|$.

Exercise. State similar criteria for componentwise backward stability.

15.6 By unitary invariance (C.2), a QR decomposition $A = Q \cdot R$ satisfies

$$\|Q\|_2\|R\|_2 = \|R\|_2 = \|QR\|_2 = \|A\|_2,$$

so that the benign case is at hand. With the modified Gram–Schmidt, Givens or Householder methods, the linear system of equations $Ax = b$ is therefore provably solved in a normwise *backward stable* fashion.

Exercise. Construct a 2×2 matrix A and a right-hand-side b, such that the numerical solution $\tilde{x}$ of the linear system of equations $Ax = b$ by QR decomposition is *not* backward stable with respect to componentwise relative errors. How about the stability of $x_0 = A\backslash b$?

Hint: Calculate the componentwise backward error $\omega_\bullet(\tilde{x})$ with the help of (15.2).

[57] Cf. §§9–10 and 18 in N. J. Higham: *Accuracy and Stability of Numerical Algorithms*, 2nd ed., Society of Industrial and Applied Mathematics, Philadelphia, 2002.

15.7 If A is s.p.d., there is $A' = A$ and $\lambda_j(A) > 0$. Hence, its spectral norm satisfies

$$\|A\|_2^2 \overset{\S C.8}{=} \max_{j=1:m} \lambda_j(AA') = \max_{j=1:m} \lambda_j(A^2) = \max_{j=1:m} \lambda_j(A)^2 = \left(\max_{j=1:m} \lambda_j(A) \right)^2$$

and that of the factors of the Cholesky decomposition $A = LL'$ satisfies

$$\|L\|_2^2 = \|L'\|_2^2 = \max_{j=1:m} \lambda_j(LL') = \max_{j=1:m} \lambda_j(A).$$

Thereby, we have shown that the benign case is at hand,

$$\|L\|_2 \|L'\|_2 = \|A\|_2,$$

and the linear system is thus provably solved in a *backward stable* fashion, too.

15.8 It is more challenging to handle the topic of triangular decomposition. Without loss of generality we assume that the matrix $A \in \mathrm{GL}(m; \mathbb{K})$ has the normalized triangular decomposition $A = LU$ (in general, of course, we have to use partial pivoting: simply replace A with $P'A$ in the following analysis).

The most useful estimates can be obtained by studying the condition number of $g^{-1} : (L, U) \mapsto L \cdot U$ for *componentwise* relative perturbations of L and U. The corresponding condition number estimate (11.4) suggests immediately the following tightening of (15.4):

Theorem (Wilkinson 1961). *The triangular decomposition $A = LU$ of an invertible matrix A yields a numerical solution $\hat{x}$ of $Ax = b$ with the backward error*

$$(A + E)\hat{x} = b, \qquad \|E\|_\infty = O\left(\| |L| \cdot |U| \|_\infty \right) \epsilon_{\mathrm{mach}}.$$

In the benign case $\| |L| \cdot |U| \|_\infty \approx \|A\|_\infty$, triangular decomposition is provably a backward stable solver, whereas in the malignant case $\| |L| \cdot |U| \|_\infty \gg \|A\|_\infty$ stability is at risk.

Example. With this criterion, we can develop a deeper understanding for the necessity of pivoting by going back to the example of §7.8, which illustrated numerical instability:

$$A = \begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} = LU, \quad L = \begin{pmatrix} 1 & 0 \\ \epsilon^{-1} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} \epsilon & 1 \\ 0 & 1 - \epsilon^{-1} \end{pmatrix}, \qquad 0 < \epsilon \ll 1.$$

This is a *malignant* case of triangular decomposition since

$$\|A\|_\infty = 2 \ll \| |L| \cdot |U| \|_\infty = 2\epsilon^{-1}.$$

However, swapping both of the rows by partial pivoting,

$$P'A = \begin{pmatrix} 1 & 1 \\ \epsilon & 1 \end{pmatrix} = LU, \quad L = \begin{pmatrix} 1 & 0 \\ \epsilon & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 1 & 1 \\ 0 & 1 - \epsilon \end{pmatrix},$$

maneuvers the system into a benign case: due to $L, U \geqslant 0$, there is $\|P'A\|_\infty = \| |L| \cdot |U| \|_\infty$.

15.9 Backward stability of using the triangular decomposition $P'A = LR$ with partial pivoting[58] therefore depends on the *growth factor*[59]

$$\gamma(A) = \frac{\||L| \cdot |R|\|_\infty}{\|A\|_\infty} \leqslant \frac{\|L\|_\infty \|R\|_\infty}{\|A\|_\infty} \leqslant \|L\|_\infty \|L^{-1}\|_\infty = \kappa_\infty(L). \tag{15.5}$$

This factor can be bounded independently of A (for *fixed* dimension m):

Lemma. *For a unipotent lower triangular matrix $L \in \mathbb{K}^{m\times m}$ with $|L| \leqslant 1$ it holds*

$$\|L\|_\infty \leqslant m, \qquad \|L^{-1}\|_\infty \leqslant 2^{m-1},$$

and therefore $\gamma(A) \leqslant m \cdot 2^{m-1}$ for triangular decomposition with partial pivoting.

Proof. Recall from §5.4 that a unipotent triangular $L = (\lambda_{jk})_{jk}$ is invertible. By partitioning L^{-1} into its rows z_j' and by using that L is unipotent, expansion of the relation $I = LL^{-1}$ yields

$$e_j' = \sum_{k=1}^{j} \lambda_{jk} z_k' = z_j' + \sum_{k=1}^{j-1} \lambda_{jk} z_k', \quad \text{that is,} \quad z_j' = e_j' - \sum_{k=1}^{j-1} \lambda_{jk} z_k' \qquad (j = 1:m).$$

With $|\lambda_{jk}| \leqslant 1$ and $\|e_j'\|_1 = 1$, the triangle inequality implies

$$\|z_j'\|_1 \leqslant 1 + \sum_{k=1}^{j-1} \|z_k'\|_1 \qquad (j = 1:m).$$

Majorizing by $2^{j-1} = 1 + \sum_{k=1}^{j-1} 2^{k-1}$ inductively yields $\|z_j'\|_1 \leqslant 2^{j-1}$. Consequently, in accordance with the definition of the *max-row-sum norm* (§C.8), we get

$$\|L^{-1}\|_\infty = \max_{j=1:m} \|z_j'\|_1 \leqslant 2^{m-1}.$$

It directly follows from $|\lambda_{jk}| \leqslant 1$ that $\|L\|_\infty \leqslant m$; thus the proof is complete. □

Triangular decomposition with partial pivoting is therefore *provably* backward stable in the following cases:

- the dimension is small (let us say $1 \leqslant m \leqslant 10$); or
- the growth factor (15.5) fulfills $\gamma(A) \not\gg 1$.

[58] See Theorem 7.11.

[59] The max-row-sum norm is *invariant* under column permutations: $\|P'A\|_\infty = \|A\|_\infty$.

15.10 In "the wild", very large growth factors $\gamma(A)$ have only *extremely seldom* been documented so far;[60] but they can nevertheless be "artificially" constructed. All inequalities in the proof of Lemma 15.9 become equalities for the particular triangular matrix

$$L = \begin{pmatrix} 1 & & & & \\ -1 & 1 & & & \\ -1 & -1 & \ddots & & \\ \vdots & \vdots & \ddots & \ddots & \\ -1 & -1 & \cdots & -1 & 1 \end{pmatrix} \in \mathbb{K}^{m\times m},$$

so that the upper bounds are taken:

$$\|L\|_\infty = m, \qquad \|L^{-1}\|_\infty = 2^{m-1}.$$

This is the L factor of the *Wilkinson matrix* (due to $|L| \leqslant 1$ there is *no* pivoting)

$$A = \begin{pmatrix} 1 & & & & 1 \\ -1 & 1 & & & 1 \\ -1 & -1 & \ddots & & 1 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ -1 & -1 & \cdots & -1 & 1 \end{pmatrix} = LU, \qquad U = \begin{pmatrix} 1 & & & & 1 \\ & 1 & & & 2 \\ & & \ddots & & \vdots \\ & & & 1 & 2^{m-2} \\ & & & & 2^{m-1} \end{pmatrix},$$

with a growth factor that grows *exponentially* in the dimension m:

$$\gamma(A) = \frac{\||L|\cdot|U|\|_\infty}{\|A\|_\infty} = \frac{m + 2^m - 2}{m} \simeq 2^m/m.$$

It is, however, $\kappa_\infty(A) = m$. Already for moderate dimensions m, there should be signs of severe numerical *instability*.

Example. A numerical example for $m = 25$ exhibits such an instability:

```
>> m = 25;
>> A = 2*eye(m)-tril(ones(m)); A(:,m)=1; % Wilkinson-Matrix
>> rng(847); b = randn(m,1); % reproducible random right-hand-side
>> [L,U,p] = lu(A,'vector'); % triangular decomposition with partial pivoting
>> x = U\(L\b(p));           % substitutions
>> r = b - A*x;              % residual
>> omega = norm(r,inf)/(norm(A,inf)*norm(x,inf)) % backward error (15.1)
omega =
   7.7456e-11
```

The backward error $\omega(\hat{x})$ is therefore by a factor of about $\gamma(A)/2$ larger than ϵ_{mach}. As a comparison we add a backward stable solution with QR decomposition:

[60] Please inform me if a relevant practical example has been "brought down".

```
>> [Q,R_qr] = qr(A);    % QR decomposition
>> x_qr = R_qr\(Q'*b); % substitutions
>> r_qr = b - A*x_qr;   % residual
>> omega_qr = norm(r_qr,inf)/(norm(A,inf)*norm(x_qr,inf)) % back. error
omega_qr =
   4.9507e-17
```

Since A with $\kappa_\infty(A) = 25$ is well-conditioned, the discrepancy between both results also very clearly substantiates the instability of using triangular decomposition in this case:

```
>> norm(x-x_qr,inf)/norm(x,inf) % relative discrepancy to QR result
ans =
   1.0972e-09
```

This limited accuracy fits well with the forward error estimate (15.3): $\kappa_\infty(A)\omega(\hat{x}) \approx 2 \cdot 10^{-9}$. Here, the theoretical predictions only overestimate both, forward and backward error, by a factor of 2.

Iterative Refinement

15.11 Arguing strictly *formally*, we could try to correct a numerical solution $\hat{x}$ of $Ax = b$ by applying the following linear system of equations for the error w:

$$r = b - A\hat{x}, \qquad Aw = r, \qquad x = \hat{x} + w.$$

Since machine arithmetic itself only provides a somewhat perturbed result $\hat{w}$, one is invited to *iterate* over this correction step: with $x_0 = \hat{x}$, for $k = 0, 1, 2, \ldots$

$$r_k = b - Ax_k, \quad Aw_k = r_k, \quad x_{k+1} = x_k + \hat{w}_k.$$

In this process, the matrix factorization of A only needs to be calculated *once*. Instead of asking about the convergence of this *iterative refinement*, one should ask whether the *stopping criterion* of backward stability will have been reached within a number of, say, n steps: $\omega(\hat{x}_n) = O(\epsilon_{\text{mach}})$.

15.12 This stopping criterion should be matched for a *well-conditioned* matrix A (for which forward and backward errors of the numerical solution are of the same magnitude), if the leading n-th part of the solution mantissa is always computed correctly using a fixed matrix factorization of A. Since the significant digits of w_k start where the significant digits of x_k end, a correct n-th part of the solution mantissa is added in every step of the iterative refinement: the first correction corrects the second n-th part of the mantissa, the second one corrects the third n-th part, etc.. This way, after $n - 1$ correction steps, approximately all the digits of the solution x have been computed correctly.

15.13 Assuming that $\kappa_\infty(A) = O(1)$ and $\gamma(A) = O(\epsilon_{\text{mach}}^{-1/2})$, these preliminary considerations suggest that triangular decomposition is good for at least half of the mantissa and therefore just a *single* step of iterative refinement should be adequate for computing an accurate result.[61] As a matter of fact, one can prove the following remarkable *theorem*:[62]

> **Theorem** (Skeel 1980). *For $\gamma(A)^2\kappa_\infty(A)\epsilon_{\text{mach}} = O(1)$, triangular decomposition with partial pivoting yields a backward stable solution after one single step of iterative refinement.*

Example. We will illustrate this theorem with the numerical example from §15.10: here, the prerequisite is fulfilled by means of $\gamma(A)^2\kappa_\infty(A)\epsilon_{\text{mach}} \approx 1$ for $m = 25$.

```
>> w = U\(L\r(p)); % correction from the stored triangular decomposition
>> x = x + w; r = b - A*x; % first step of it. refinement and new residual
>> omega = norm(r,inf)/(norm(A,inf)*norm(x,inf)) % backward stability!
omega =
   6.1883e-18
>> norm(x-x_qr,inf)/norm(x,inf) % relative deviation to QR result
ans =
   9.2825e-16
```

As a comparison: the *unavoidable* error in this case is about $\kappa_\infty(A) \cdot \epsilon_{\text{mach}} \approx 2 \cdot 10^{-15}$.

Exercise. Test the limitations of iterative refinement for larger dimensions of m.

[61] This way, one still saves a factor of 2 in computational cost when compared to QR decomposition, cf. §§7.7 and 9.8.

[62] Cf. §E.1; for a proof with componentwise relative errors see R. D. Skeel: *Iterative refinement implies numerical stability for Gaussian elimination*, Math. Comp. 35, 817–832, 1980.

IV Least Squares

The choice of the square was purely arbitrary.

(Carl Friedrich Gauss 1839)

The method of least squares is the automobile of modern statistical analysis: despite its limitations, occasional accidents, and incidental pollution, it is known and valued by nearly all.

(Stephen Stigler 1981)

16 Normal Equation

16.1 In experimental and observational sciences, one is often confronted with the task of estimating parameters $p = (\theta_1, \ldots, \theta_n)'$ of a mathematical model from a set of *noisy* measurements.[63] If the parameters enter the model *linearly*, such a set-up can be written in the form

$$b = Ap + e$$

where

- $b \in \mathbb{R}^m$ is the measured *observation vector*;
- $A \in \mathbb{R}^{m \times n}$ is the *design matrix*;
- $p \in \mathbb{R}^n$ is the parameter vector or *feature vector* to be estimated;
- $e = (\epsilon_1, \ldots, \epsilon_m)'$ is the vector of inaccessible, random *perturbations* (*noise*).

The number of measurements m is often larger than the number of parameters n.

[63] This is referred to as *data fitting* or *regression analysis* in parametric statistics.

16.2 An estimator x of the parameter vector p replaces the actual perturbation e with the *residual* r where

$$b = Ax + r, \qquad r = (\rho_1, \dots, \rho_m).$$

The *least squares estimator* then solves the minimization problem

$$\|r\|_2^2 = \sum_{j=1}^{m} \rho_j^2 = \min!;$$

the *Gauss–Markov theorem* asserts its optimality[64] if the noise obeys certain statistical properties: centered, uncorrelated, equal variance. For unequal variances or correlated noise one uses the weighted or generalized least squares method.

Remark. The *least squares method* was first published in 1805 by Adrien-Marie Legendre in his work on trajectories of comets; it had however already been used in 1801 by Carl Friedrich Gauss for calculating the trajectory of the minor planet Ceres. Both mathematicians, having previously got in each other's way while studying number theory and elliptic functions, bitterly fought for decades over their priority in this discovery.[65]

Exercise. Show that the least squares estimator of the parameter θ from the measurements

$$\beta_j = \theta + \epsilon_j \quad (j = 1 : m)$$

is equal to the sample mean $\frac{1}{m}\sum_{j=1}^{m} \beta_j$. Solve for it in two ways: directly and with (16.1).

16.3 We therefore consider the *least squares problem*

$$x = \operatorname*{arg\,min}_{y \in \mathbb{R}^n} \|b - Ay\|_2, \qquad A \in \mathbb{R}^{m \times n},\ b \in \mathbb{R}^m.$$

Equivalently, we have to minimize the function

$$F(y) = \tfrac{1}{2}\|b - Ay\|_2^2 = \tfrac{1}{2}(b'b - 2y'A'b + y'A'Ay)$$

whose gradient is $\nabla F(y) = -A'b + A'Ay$. The necessary optimality condition $\nabla F(x) = 0$ is therefore equivalent to the *normal equation* (as introduced by Gauss)

$$A'Ax = A'b. \tag{16.1}$$

For A with full column rank (which we want to assume from now on), and thus for $A'A$ s.p.d. (Lemma 9.1), the normal equation possesses a *unique* solution $x \in \mathbb{R}^n$. This actually yields the unique minimum of F:

$$F(y) = \tfrac{1}{2}(b'b - 2y'A'Ax + y'A'Ay) = F(x) + \tfrac{1}{2}(y - x)'A'A(y - x) \geqslant F(x)$$

with equality for precisely $y = x$ ($A'A$ is s.p.d.). Hence, we have proven for $\mathbb{K} = \mathbb{R}$:

[64] That is to say that the it is the best linear unbiased estimator (BLUE).

[65] R. L. Plackett: *Studies in the History of Probability and Statistics. XXIX: The discovery of the method of least squares*, Biometrika 59, 239–251, 1972.

Theorem. *For $A \in \mathbb{K}^{m\times n}$ with full column rank, the least squares problem*

$$x = \operatorname*{arg\,min}_{y\in\mathbb{K}^n} \|b - Ay\|_2, \qquad A \in \mathbb{K}^{m\times n},\ b \in \mathbb{K}^m,$$

is equivalent to the uniquely solvable normal equation $A'Ax = A'b$, where $A'A$ is s.p.d..

Exercise. Show the theorem to hold for $\mathbb{K} = \mathbb{C}$, too. *Hint:* Separate real and imaginary parts.

Remark. The solution map $A^\dagger : b \mapsto x = (A'A)^{-1}A'b$ is *linear* in b; the induced matrix is called the *pseudoinverse* of A and coincides with the inverse for $m = n$.

16.4 The normal equation leads to the solution algorithm

$$A \xmapsto[\text{normal equation}]{\text{assembly of the}} A'A \xmapsto[\text{normal equation}]{\text{solution of the}} x \tag{16.2}$$

of the least squares problem (the vector b being fixed); the normal equation itself is of course solved with the Cholesky decomposition from §8.3.[66] Using the symmetry of $A'A$, the computational cost is therefore (in leading order)

$$\#\text{flop(matrix product } A'A) \ +\ \#\text{flop(Cholesky decomposition of } A'A)$$
$$\doteq mn^2 + \frac{1}{3}n^3.$$

16.5 To evaluate the stability of (16.2) using Criterion F from §13.8,[67] we compare the condition number $\kappa_2(A'A)$ of the normal equation with that of the least squares problem. Since it can be shown for $m = n$, as was the case in §15.7, that

$$\kappa_2(A) = \sqrt{\kappa_2(A'A)}, \tag{16.3}$$

we *define* $\kappa_2(A)$ as (16.3) for the case of $m > n$. The relative condition number κ_{LS} of the least squares problem with respect to perturbations in A is bounded as[68]

$$\max\left(\kappa_2(A), \omega\cdot\kappa_2(A)^2\right) \leqslant \kappa_{\text{LS}} \leqslant \kappa_2(A) + \omega\cdot\kappa_2(A)^2 \tag{16.4}$$

with the relative measure of the residual (cf. (15.1))

$$\omega = \frac{\|r\|_2}{\|A\|_2\,\|x\|_2}, \qquad r = b - Ax.$$

[66] Cholesky had originally developed his method for the system of normal equations in the field of mathematical geodesy.

[67] The start section is in this case *not* invertible, so Criterion B is not applicable.

[68] In contrast, it holds for perturbations in b

$$\omega\kappa_2(A) \leqslant \kappa_{\text{LS}} = \kappa_2(A)\|b\|_2/(\|A\|_2\|x\|_2) \leqslant (1+\omega)\kappa_2(A).$$

See P.-Å. Wedin: *Perturbation theory for pseudo-inverses*, BIT 13, 217–232, 1973; A. van der Sluis: *Stability of the solutions of linear least squares problems*, Numer. Math. 23, 241–254, 1975.

Therefore, (16.2) is likely *unstable* if the residual r is relatively small ($\omega \ll 1$) and the matrix A is ill-conditioned ($\kappa_2(A) \gg 1$): it then holds

$$\kappa_{\mathrm{LS}} \leqslant (\kappa_2(A)^{-1} + \omega)\kappa_2(A)^2 \ll \kappa_2(A)^2 = \kappa_2(A'A).$$

In contrast, for all other cases, i.e., for relatively large residuals or well-conditioned matrices, the use of the normal equation is *stable*.

Example. The numerical instability of the algorithm (16.2) can famously be illustrated with the following vanishing-residual example (P. Läuchli 1961):

$$A = \begin{pmatrix} 1 & 1 \\ \epsilon & 0 \\ 0 & \epsilon \end{pmatrix},\ b = \begin{pmatrix} 2 \\ \epsilon \\ \epsilon \end{pmatrix},\ x = \begin{pmatrix} 1 \\ 1 \end{pmatrix};\quad A'A = \begin{pmatrix} 1+\epsilon^2 & 1 \\ 1 & 1+\epsilon^2 \end{pmatrix},\ \kappa_2(A'A) = \frac{2}{\epsilon^2} + 1.$$

Here, $A'A$ is actually *exactly* numerically singular for $\epsilon^2 < 2\epsilon_{\mathrm{mach}}$ since then $1 \mathbin{\widehat{+}} \epsilon^2 = 1$ (the information about ϵ is completely lost while assembling $A'A$; after the start section, the parameter ϵ can no longer be reconstructed, cf. Criterion B from §13.8). For a slightly larger ϵ we obtain the following numerical example:

```
>> e = 1e-7;                          % ϵ = 10^-7
>> A = [1 1;e 0;0 e]; b = [2;e;e];    % Läuchli example
>> x = (A'*A)\(A'*b)                  % solution of the normal equation
x =
      1.011235955056180e+00
      9.887640449438204e-01
```

The loss of 14 decimal places is consistent with $\kappa_2(A'A) \approx 2 \cdot 10^{14}$; however, due to the condition number $\kappa_{LS} = \kappa_2(A) \approx 10^7$, at most a loss of approximately 7 decimal places would be acceptable.

17 Orthogonalization

For the following, we will assume that $A \in \mathbb{K}^{m\times n}$ has full column rank and, therefore, $m \geqslant n$.

17.1 A stable algorithm for the least squares problem should access A directly[69] and not via the detour of $A'A$. With the help of a (normalized) QR decomposition $A = QR$, we can simplify the normal equations according to Theorem 9.3 to

$$\underbrace{A'A}_{=R'R}\, x = \underbrace{A'}_{=R'Q'}\, b$$

and therefore equivalently reduce via multiplication with $(R')^{-1}$ to

$$Rx = Q'b. \tag{17.1}$$

[69] In 1961, Eduard Stiefel spoke of the "principle of direct attack" in the field of numerical analysis.

The *backward stability* of this algorithm—with the modified Gram–Schmidt, Givens or Householder method for the calculation of the reduced QR decomposition—can be understood as in §15.6 by the section decomposition

$$A \xmapsto[QR \text{ factorization}]{\text{reduced}} (Q, R) \xmapsto[\text{backward substitution}]{\text{multiplication with } Q'} x.$$

Here, due to the column orthonormality of Q, the inverse start section, that is $(Q, R) \mapsto Q \cdot R = A$, is actually well-conditioned: $\|Q\|_2 \|R\|_2 = \|A\|_2$.

Exercise. Show for the full QR decomposition (9.3): $r = b - Ax = (I - Q_1 Q_1')b = Q_2 Q_2' b$. Explain why only the third formula is stable and should therefore be used for the residual.

17.2 By means of a simple trick, we can actually by and large forgo explicit knowledge of the factor Q. To do so, we calculate the normalized QR decomposition of the matrix A augmented by the column b, i.e.,

$$\left(A \;\middle|\; b \right) = \left(Q \;\middle|\; q \right) \cdot \left(\begin{array}{c|c} R & z \\ \hline & \rho \end{array} \right).$$

By expanding we obtain not only the QR decomposition $A = QR$, but also

$$b = Qz + \rho q.$$

Due to $Q'Q = I$ and $Q'q = 0$, multiplication with Q' provides the relationship

$$Q'b = z.$$

Thus, the constitutive equation (17.1) can be simplified to

$$Rx = z.$$

Further, $q'q = \|q\|_2^2 = 1$ implies that the *positive* ρ is the norm of the residual:

$$\rho = \|\rho q\|_2 = \|b - Qz\|_2 = \|b - QRx\|_2 = \|b - Ax\|_2.$$

To summarize, this algorithm can be written in MATLAB as follows:

Program 12 (Q-Free Solution of the Least Squares Problem).

Notice: in MATLAB the QR decomposition is *not* normalized to a positive diagonal of R.

```
R = triu(qr([A b]));          % R-factor of the matrix A extended by b
x = R(1:n,1:n)\R(1:n,n+1);    % solution of Rx = z
rho = abs(R(n+1,n+1));        % norm of the residual
```

As for linear systems of equations, the least squares problem is briefly solved by the line

```
x = A\b;
```

Example. Let us re-examine the numerical example from §16.5:

```
>> x = A\b
x =
      9.999999999999996e-01
      1.000000000000000e+00
```

Despite the relatively large condition number $\kappa_{LS} = \kappa_2(A) \approx 10^7$, almost all digits are correct; this time the machine arithmetic does not "trigger" the worst case scenario.

Exercise. Let the columns of a matrix $B \in \mathbb{R}^{m\times n}$ constitute a basis of the n-dimensional subspace $U \subset \mathbb{R}^m$. For $x \in \mathbb{R}^m$, denote by $d(x, U)$ the distance from x to U.

- Render the calculation of $d(x, U)$ as a least squares problem.
- Write a two line MATLAB code that calculates $d(x, U)$ given the input (B, x).

17.3 The computational cost of such a Q-free solution of a least squares problem using the Householder method is, according to §§9.8 and D.5,

$$2mn^2 - 2n^3/3 \text{ flop.}$$

For $m \gg n$, the computational cost is thus two times larger than the cost to calculate the solution via the normal equation with Cholesky decomposition, which in accordance with §16.4 is

$$mn^2 + n^3/3 \text{ flop.}$$

For $m \approx n$, both are approximately $4m^3/3$ flop and therefore of comparable size. Therefore, both algorithms have their merit: the normal equation with Cholesky should be used for $m \gg n$ with a relatively large residual (in the sense of $\omega \not\ll 1$) or a comparatively well-conditioned matrix A; orthogonalization should be used in all other cases (cf. §16.5).

Exercise. Let a matrix $A \in \mathbb{R}^{m\times n}$ be given with full *row rank* $m < n$. Examine the *underdetermined* linear system of equations $Ax = b$.

- Show that there exists a *unique* minimal solution x_* for which it holds

  $$x_* = \arg\min\{\|x\|_2 : Ax = b\}.$$

  This solution is given by $x_* = A'w$, where w solves $AA'w = b$.
- Explain why this two-step algorithm is potentially unstable.
  Hint: Use, without proof, the fact that[70] $\kappa(x_*; A) \leqslant 2\kappa_2(A)$ with $\kappa_2(A) = \sqrt{\kappa_2(AA')}$.
- Develop an efficient and stable algorithm to calculate x_*.
- Supplement this algorithm with the calculation of $\ker(A)$.

[70] See Theorem 5.6.1 in G. H. Golub, C. F. Van Loan: *Matrix Computations*, 4th ed., The Johns Hopkins University Press, Baltimore, 2013.

V Eigenvalue Problems

When I was a graduate student working at Oak Ridge, my office was dubbed the Eigencave.

(Pete Stewart 1998)

While in theoretical mathematics a problem can often be simplified by means of transformations, the indiscriminate application of such methods in numerical mathematics often leads to failure. This is due to the fact that the transformed problem contains less numerical information than the original problem.

(Eduard Stiefel 1961)

18 Basic Concepts

18.1 An *eigenpair* $(\lambda, x) \in \mathbb{C} \times \mathbb{C}^m$ of a matrix $A \in \mathbb{C}^{m\times m}$, composed of an *eigenvalue* λ and an *eigenvector* x, is defined by

$$Ax = \lambda x, \qquad x \neq 0; \tag{18.1}$$

the set of eigenvalues is the *spectrum* $\sigma(A)$ of A. Since the eigenvalue equation is *homogeneous* in x, we often consider eigenvectors *normalized* by $\|x\|_2 = 1$.

18.2 Obviously, λ is an eigenvalue, if and only if $\lambda I - A$ is singular and therefore λ is a root of the *characteristic polynomial*

$$\chi(\zeta) = \det(\zeta I - A);$$

the multiplicity of this root is the (algebraic) *multiplicity* of the eigenvalue.

Remark. The fundamental theorem of algebra ensures the existence of eigenvalues λ, corresponding eigenvectors are obtained, at least theoretically, as a kernel vector of $\lambda I - A$.

18.3 The connection to the characteristic polynomial χ suggests an algorithm for calculating the eigenvalues by first finding the coefficients and then the roots of χ:

$$A \overset{g}{\longmapsto} \chi \overset{h}{\longmapsto} (\lambda_1, \ldots, \lambda_m).$$

Unfortunately, this is *numerically unstable* due to the ill-conditioned end section h.

Example. The application to a diagonal matrix with the eigenvalues $1, \dots, 22$ yields:

```
>> A = diag(1:22);       % diagonal matrix with eigenvalues 1:22
>> chi = charpoly(A);    % coefficents of the charachteristic polynomial
>> lambda = roots(chi); % roots
>> lambda(7:8)
ans =
            15.4373281959839 +       1.07363608241233i
            15.4373281959839 -       1.07363608241233i
```

A comparison with the *nearest* eigenvalue $\lambda = 15$ yields an absolute error of ≈ 1. We will see in §19.4 that only an absolute error of $O(\epsilon_{\text{mach}})$ would be acceptable because of the well-conditioned nature of the eigenvalue problem itself.

Exercise. Find the condition number of a simple root of the polynomial χ as a function of its coefficients; use componentwise relative error for the coefficients and absolute error for the roots. What is the condition number for $\lambda = 15$ in the example above?

Answer: $\kappa(\lambda) = \chi^\sharp(|\lambda|)/|\chi'(\lambda)|$, where the polynomial $\chi^\sharp$ is obtained from χ by applying absolute values to the coefficients. In the example, $\kappa(15) \approx 6 \cdot 10^{16}$, so that $\kappa(15) \cdot \epsilon_{\text{mach}} \approx 6$:

```
>> lambda = 15;
>> kappa = polyval(abs(chi),lambda)/abs(polyval(polyder(chi),lambda))
kappa =
   5.7345e+16
```

Remark. In numerical analysis, one goes the other way round: the roots of a polynomial

$$\zeta^m + \alpha_{m-1}\zeta^{m-1} + \cdots + \alpha_1\zeta + \alpha_0$$

are namely the eigenvalues of the corresponding *companion matrix*

$$\begin{pmatrix} 0 & & & & -\alpha_0 \\ 1 & \ddots & & & -\alpha_1 \\ & \ddots & \ddots & & \vdots \\ & & \ddots & 0 & -\alpha_{m-2} \\ & & & 1 & -\alpha_{m-1} \end{pmatrix}.$$

18.4 If the eigenpair (λ, x) is already known, it can be "split off" from A. To do so, we extend the *normalized* eigenvector x to an orthonormal basis, that is, let

$$Q = \left(x \;\middle|\; U \right)$$

be unitary.[71] It then actually holds

$$Q'AQ = \begin{pmatrix} x' \\ \hline U' \end{pmatrix} \left(\lambda x \;\middle|\; AU \right) = \left(\begin{array}{c|c} \lambda & x'AU \\ \hline & A_\lambda \end{array} \right), \qquad A_\lambda = U'AU,$$

[71] Such a Q can be constructed by a complete QR decomposition as in Theorem 9.10: `[Q,~] = qr(x)`.

so that because of (an implicit polynomial division)

$$\det(\zeta I - A) = (\zeta - \lambda)\det(\zeta I - A_\lambda)$$

the eigenvalues of the *deflated* matrix A_λ supply the remaining eigenvalues of A. This reduction of dimension is called *deflation* of A.

Exercise. Given an eigenpair (μ, y) of A_λ, construct a corresponding eigenpair (μ, z) of A.

18.5 By applying the deflation technique once more to the deflated matrix A_λ itself and by repeating this process recursively, we obtain, row-wise from top to bottom, a *Schur decomposition* or *unitary triangulation* (for a column-wise construction see §21.8)

$$Q'AQ = T = \begin{pmatrix} \lambda_1 & * & \cdots & * \\ & \ddots & \ddots & \vdots \\ & & \ddots & * \\ & & & \lambda_m \end{pmatrix}, \qquad \sigma(A) = \{\lambda_1, \ldots, \lambda_m\}, \tag{18.2}$$

with a unitary matrix Q and an upper triangular matrix T, where the eigenvalues of A appear on the diagonal of T according to their multiplicity.

Remark. Numerically, a Schur decomposition is always, *without exception*, to be preferred to the Jordan normal form (why?). For most applications, all of the information of interest in the Jordan normal form can essentially already be found in a Schur decomposition.

18.6 If A is *normal*, i.e., $A'A = AA'$, it follows from $QQ' = I$ that also T is normal:

$$T'T = Q'A'QQ'AQ = Q'A'AQ = Q'AA'Q = Q'AQQ'A'Q = TT'.$$

Normal triangular matrices are diagonal: the proof goes inductively, by writing

$$T = \left(\begin{array}{c|c} \tau & t \\ \hline & * \end{array}\right)$$

and expanding $T'T = TT'$, one gets as the upper left entry

$$|\tau|^2 = |\tau|^2 + \|t\|_2^2, \quad \text{thus} \quad t = 0.$$

Thus, a normal (and hence, a fortiori, self-adjoint) matrix A can be *unitarily diagonalized*: there is a unitary matrix Q, such that

$$Q'AQ = D = \operatorname{diag}(\lambda_1, \ldots, \lambda_m), \qquad \sigma(A) = \{\lambda_1, \ldots, \lambda_m\}. \tag{18.3}$$

Inversely, unitary diagonalizable matrices are, of course, always normal.

Remark. Since $AQ = QD$, every column vector of Q is an eigenvector of A; hence, every normal matrix possesses an orthonormal basis of eigenvectors.

Exercise. Show that for a *self adjoint* matrix A, the eigenvalues are real. For a *real self-adjoint* matrix A, the matrix Q can also be constructed as *real*.

19 Perturbation Theory

Backward Error

19.1 Let the matrix A be given along with a *perturbed* eigenpair (λ, x), $x \neq 0$. As with linear systems of equations, we define *backward stability* as the smallest perturbation of A for which this pair becomes an *exact* eigenpair:

$$\omega = \min\left\{ \frac{\|E\|_2}{\|A\|_2} : (A+E)x = \lambda x \right\}.$$

This quantity can be directly calculated with the help of Theorem 15.2 of Rigal and Gaches by setting $b = \lambda x$:

$$\omega = \frac{\|r\|_2}{\|A\|_2\|x\|_2}, \qquad r = Ax - \lambda x.$$

Thus, a calculation of a numerical eigenpair with $\omega = O(\epsilon_{\text{mach}})$ is *backward stable*.

19.2 Let an approximate eigenvector $x \neq 0$ be given. Which $\lambda \in \mathbb{C}$ is the best approximate eigenvalue fitting to it in terms of the backward error ω? Since the denominator of the expression ω does not depend on the approximate eigenvalue, it suffices to minimize the residual itself:

$$\lambda = \operatorname*{arg\,min}_{\mu \in \mathbb{C}} \|Ax - x \cdot \mu\|_2.$$

The *normal equation* (cf. Theorem 16.3) of this least squares problem provides (notice here that x plays the role of the design matrix and Ax that of the observation)

$$x'x \cdot \lambda = x'Ax, \quad \text{such that} \quad \lambda = \frac{x'Ax}{x'x};$$

this expression of the optimal λ is called the *Rayleigh quotient* of A in x.

19.3 Now, let an approximate eigenvalue λ be given. Its *absolute* backward error

$$\eta = \min\{\|E\|_2 : \lambda \in \sigma(A+E)\} = \min\{\|E\|_2 : \lambda I - (A+E) \text{ is singular}\}$$

is exactly $\eta = \|\lambda I - A\|_2 / \kappa_2(\lambda I - A)$ by Kahan's Theorem 11.9, that is, we have

$$\eta = \operatorname{sep}(\lambda, A) = \begin{cases} \|(\lambda I - A)^{-1}\|_2^{-1}, & \lambda \notin \sigma(A), \\ 0, & \text{otherwise}; \end{cases} \tag{19.1}$$

where $\operatorname{sep}(\lambda, A)$ is called the *separation* between λ and A. According to §11.2, the *absolute* condition number $\kappa_{\text{abs}}(\lambda)$ of an eigenvalue is therefore given by

$$\kappa_{\text{abs}}(\lambda) = \limsup_{\tilde{\lambda} \to \lambda} \frac{|\tilde{\lambda} - \lambda|}{\operatorname{sep}(\tilde{\lambda}, A)}, \qquad \lambda \in \sigma(A).$$

Exercise. Show that $\operatorname{sep}(\tilde{\lambda}, A) \leqslant \operatorname{dist}(\tilde{\lambda}, \sigma(A))$ and therefore invariably $\kappa_{\text{abs}}(\lambda) \geqslant 1$.

Conditioning of the Eigenvalue Problem with Normal Matrices

19.4 In the case of normal matrices, the separation is the distance to the spectrum:

Theorem. *If the matrix A is normal, then it holds* $\operatorname{sep}(\lambda, A) = \operatorname{dist}(\lambda, \sigma(A))$.

Proof. Unitary diagonalization (18.3) implies that

$$(\lambda I - A)^{-1} = Q(\lambda I - D)^{-1} Q'$$

with a diagonal matrix D composed of the eigenvalues of A and therefore

$$\|(\lambda I - A)^{-1}\|_2 = \| \underbrace{(\lambda I - D)^{-1}}_{\text{diagonal matrix}} \|_2 = \max_{\mu \in \sigma(A)} \frac{1}{|\lambda - \mu|} = \frac{1}{\operatorname{dist}(\lambda, \sigma(A))}.$$

Here, we have used the unitary invariance of the spectral norm and (C.4). □

Exercise. Construct an A with $0 \in \sigma(A)$ such that $\operatorname{dist}(\lambda, \sigma(A)) / \operatorname{sep}(\lambda, A) \to \infty$ for $\lambda \to 0$.

Together with (19.1) this theorem directly shows that the eigenvalues of normal matrices are always well-conditioned in the sense of *absolute* error:

Corollary (Bauer–Fike 1960). *Let A be normal. It then holds that for $\lambda \in \sigma(A + E)$*

$$\operatorname{dist}(\lambda, \sigma(A)) \leqslant \|E\|_2.$$

The absolute condition number of an eigenvalue $\lambda \in \sigma(A)$ satisfies $\kappa_{abs}(\lambda) = 1$.

Exercise. Show for diagonalizable $A = XDX^{-1}$ that $\operatorname{dist}(\lambda, \sigma(A)) \leqslant \kappa_2(X)\|E\|_2$.

19.5 For *simple* eigenvalues, the condition number of the corresponding eigenvector can be estimated at least. Since only the direction of eigenvectors matters, we measure their perturbation in terms of angles:

Theorem (Davis–Kahan 1970). *Let (λ, x) be an eigenpair of a normal matrix A, where the eigenvalue λ is simple; furthermore let $(\tilde{\lambda}, \tilde{x})$ be an eigenpair of $A + E$. It then holds*

$$|\sin \measuredangle(x, \tilde{x})| \leqslant \frac{\|E\|_2}{\operatorname{dist}(\tilde{\lambda}, \sigma(A) \setminus \{\lambda\})}.$$

For this error measure, the condition number is bounded by $\kappa \leqslant \operatorname{dist}(\lambda, \sigma(A) \setminus \{\lambda\})^{-1}$. This distance is called the spectral gap of A in λ.

Proof. Without loss of generality x and $\tilde{x}$ are normalized. The construction from §18.4 leads to

$$1 = \|\tilde{x}\|_2^2 = \|Q'\tilde{x}\|_2^2 = \underbrace{|x'\tilde{x}|^2}_{=\cos^2 \measuredangle(x,\tilde{x})} + \|U'\tilde{x}\|_2^2, \text{ i.e., } |\sin \measuredangle(x, \tilde{x})| = \|U'\tilde{x}\|_2. \quad (19.2)$$

From $U'A = A_\lambda U'$ (why?) and $E\tilde{x} = \tilde{\lambda}\tilde{x} - A\tilde{x}$ follows

$$U'E\tilde{x} = \tilde{\lambda}U'\tilde{x} - U'A\tilde{x} = (\tilde{\lambda}I - A_\lambda)U'\tilde{x}, \quad \text{thus} \quad U'\tilde{x} = (\tilde{\lambda}I - A_\lambda)^{-1}U'E\tilde{x},$$

so that by $\|U'\|_2 = 1$ and $\|\tilde{x}\|_2 = 1$ we obtain the bound (for general A up to here)

$$\sin\measuredangle(x,\tilde{x}) \leqslant \|(\tilde{\lambda}I - A_\lambda)^{-1}\|_2\|E\|_2 = \|E\|_2/\operatorname{sep}(\tilde{\lambda}, A_\lambda)\ .$$

Along with A, the matrix A_λ is also normal (cf. §18.6) and since λ is simple it holds $\sigma(A_\lambda) = \sigma(A) \setminus \{\lambda\}$, so that Theorem 19.4 finally yields the assertion. □

Remark. Determining eigenvectors belonging to *degenerate* eigenvalues is generally an ill-posed problem. For instance, *every* direction in the plane constitutes an eigenvector of the identity matrix $I \in \mathbb{C}^{2\times 2}$ with double eigenvalue 1, but only the directions of the coordinate axes correspond to the eigenvectors of the perturbed matrix $I + E = \operatorname{diag}(1+\epsilon, 1)$ for an arbitrarily small $\epsilon \neq 0$.

Exercise. For $x \neq 0$ let $P = xx'/(x'x)$ be the orthogonal projection onto $\operatorname{span}\{x\}$. Show:

$$|\sin\measuredangle(x,\tilde{x})| = \|P(x) - P(\tilde{x})\|_2.$$

19.6 For *non-normal* matrices A, the study of *ϵ-pseudospectra* (with small $\epsilon > 0$)

$$\sigma_\epsilon(A) = \{\lambda \in \mathbb{C} : \operatorname{sep}(\lambda, A) \leqslant \epsilon\} \overset{\S 19.3}{=} \{\lambda \in \mathbb{C} : \lambda \in \sigma(A+E) \text{ for a } \|E\|_2 \leqslant \epsilon\}$$

is often much more meaningful than just the examination of $\sigma(A)$. This is a very attractive area of research with many applications from random matrices to card shuffling to the control of flutter instability for commercial airplanes.[72]

20 Power Iteration

20.1 Since according to §§18.2–18.3 the eigenvalue problem is formally equivalent to calculating the roots of polynomials, the Abel–Ruffini theorem tells us that for dimensions $m \geqslant 5$ there exists *no* "solution formula" that would just use the operations $+, -, \times, /, \sqrt[n]{\ }$. Instead, one constructs iterative sequences

$$(\mu_0, v_0), (\mu_1, v_1), (\mu_2, v_2), \ldots$$

of approximate eigenpairs and studies their convergence.

[72] See the monograph by L. N. Trefethen, M. Embree: *Spectra and Pseudospectra. The Behavior of Nonnormal Matrices and Operators*, Princeton University Press, Princeton and Oxford, 2005.

20.2 In practice, such iterations are stopped, at the latest, when backward stability is reached at the nth iterate (cf. §19.1):[73]

$$\omega_n = O(\epsilon_{\text{mach}}), \qquad \omega_n = \frac{\|r_n\|_2}{\|A\|_2\|v_n\|_2}, \qquad r_n = Av_n - \mu_n v_n.$$

Since the spectral norm of a matrix would have to be approximated as an eigenvalue problem itself (§C.8), which is costly and would be a circular reasoning here, we replace it with the Frobenius norm and use the following bound instead (§C.9)

$$\tilde{\omega}_n = \frac{\|r_n\|_2}{\|A\|_F\|v_n\|_2}, \qquad \tilde{\omega}_n \leqslant \omega_n \leqslant \sqrt{m}\cdot\tilde{\omega}_n. \tag{20.1}$$

Concretely, the approximation is stopped at a *user defined* $\tilde{\omega}_n \leqslant \text{tol}$ or one iterates until $\tilde{\omega}_n$ saturates at the level $O(\epsilon_{\text{mach}})$ of numerical noise (cf. Fig. 1).

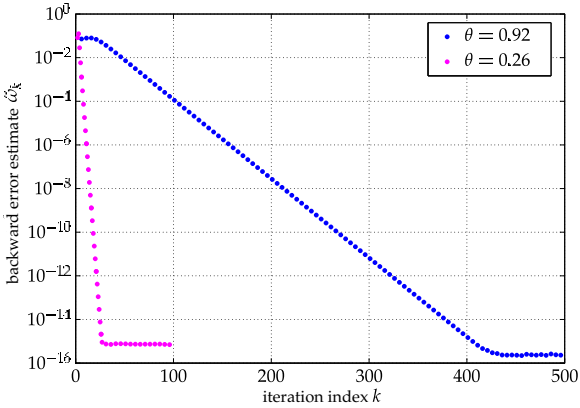


Figure 1: Convergence of the power iteration for two different normal 1000×1000 matrices.

20.3 By considering (18.1) a fixed point equation for the direction of the eigenvector x, the fixed point iteration $k = 1, 2, \ldots$ starting at a vector v_0 is:

$$\begin{aligned} w_k &= Av_{k-1} && \text{(application of the matrix)} \\ v_k &= w_k / \|w_k\|_2 && \text{(normalization)} \\ \mu_k &= v_k' \underbrace{Av_k}_{=w_{k+1}} && \text{(Rayleigh quotient as in §19.2)} \end{aligned}$$

In the literature, this is called *power iteration*, because v_k is the vector obtained from normalizing $A^k v_0$ (R. von Mises 1929).

[73] This non-asymptotic use of the O-notation is "par abus de langage" (N. Bourbaki); in this case it means $\omega_k \leqslant c\epsilon_{\text{mach}}$ with a suitably *chosen* constant c, which we allow to be polynomially dependent on the dimension m (say, to be concrete, $c = 10m$).

Program 13 (Power Iteration). The program is organized in such a way that intermediate results are never re-calculated but stored instead.

```
normA = norm(A,'fro');      % store ‖A‖_F
omega = inf;                % initialize the backward error
w = A*v;                    % initialize w = Av
while omega > tol           % is the backward error still too large?
  v = w/norm(w);            % new v
  w = A*v;                  % new w
  mu = v'*w;                % Rayleigh quotient
  r = w - mu*v;             % residual
  omega = norm(r)/normA;    % backward error estimate ω̃
end
```

20.4 Power iteration generally converges to an eigenvector corresponding to the eigenvalue of *largest size* as long as it is unique, then called the *dominant* eigenvalue. For the sake of simplicity, we limit ourselves to normal matrices:

Theorem. *Let it be possible to order the eigenvalues of a normal matrix A as*

$$|\lambda_1| > |\lambda_2| \geqslant \cdots \geqslant |\lambda_m|, \quad \textit{so that especially} \quad \theta = \left|\frac{\lambda_2}{\lambda_1}\right| < 1.$$

Let x_1 be a normalized eigenvector corresponding to the dominant eigenvalue λ_1. If the starting vector v_0 satisfies the condition[74] *$x_1' v_0 \neq 0$, power iteration calculates approximate eigenpairs (μ_k, v_k) converging as*[75]

$$\sin \measuredangle(v_k, x_1) = O(\theta^k), \qquad |\mu_k - \lambda_1| = O(\theta^{2k}) \qquad (k \to \infty).$$

Proof. We will use the unitary diagonalization of A from §18.6 in the form

$$Q'AQ = \left(\begin{array}{c|c} \lambda_1 & \\ \hline & D \end{array}\right), \quad Q = \left(\begin{array}{c|c} x_1 & U \end{array}\right), \quad D = U'AU = \operatorname{diag}(\lambda_2, \ldots, \lambda_m).$$

Using the transformed vector $y = U'v_0$ it follows from the diagonal structure that

$$v_0 = \eta_1 x_1 + Uy, \qquad \eta_1 = x_1' v_0 \neq 0, \qquad A^k v_0 = \eta_1 \lambda_1^k x_1 + UD^k y.$$

The diagonal matrix D satisfies $\|D^k\|_2 = |\lambda_2|^k = |\lambda_1|^k \theta^k$ (cf. §C.11). Therefore, from $\eta_1 \lambda_1^k \neq 0$ we obtain

$$A^k v_0 = \eta_1 \lambda_1^k (x_1 + O(\theta^k)) \sim \eta_1 \lambda_1^k x_1 \qquad (k \to \infty).$$

Thus $A^k v_0$ becomes *asymptotically parallel* to the eigenvector x_1 at a rate of $O(\theta^k)$.

[74] This condition is best realized with a *random* choice of v_0.

[75] We do *not* claim that the v_k converge itself: in general, $v_k = \sigma_k x_1 + O(\theta^k)$ with phases (phase = "complex sign") $|\sigma_k| = 1$ which do not necessarily converge—"the signs may jump."

The precise convergence estimates can be accounted for as follows: since v_k and $A^k v_0$ only differ by a normalization, (19.2) provides

$$|\sin \measuredangle(v_k, x_1)| = \|U' v_k\|_2 = \frac{\|U' A^k v_0\|_2}{\|A^k v_0\|_2} = \frac{\|D^k y\|_2}{\|\eta_1 \lambda_1^k x_1 + U D^k y\|_2}$$
$$\leqslant \frac{\|D^k y\|_2}{|\eta_1|\,|\lambda_1|^k - \|D^k y\|_2} \leqslant \frac{\theta^k \|y\|_2/|\eta_1|}{1 - \theta^k \|y\|_2/|\eta_1|} = O(\theta^k),$$

where we have used $\eta_1 \lambda_1^k \neq 0$ and, eventually, $\theta^k \|y\|_2/|\eta_1| < 1$. Also due to (19.2), the approximate eigenvalue μ_k satisfies

$$\mu_k = v_k' A v_k = \lambda_1 \underbrace{|v_k' x_1|^2}_{=1-\|U' v_k\|_2^2} + v_k' U D U' v_k = \lambda_1 - \lambda_1 \|U' v_k\|_2^2 + v_k' U D U' v_k,$$

so that finally from $\|D\|_2 = |\lambda_2|$ the estimate

$$|\mu_k - \lambda_1| \leqslant (|\lambda_1| + |\lambda_2|)\|U' v_k\|_2^2 = (|\lambda_1| + |\lambda_2|) \sin^2 \measuredangle(v_k, x_1) = O(\theta^{2k})$$

appears and everything is proven. □

Exercise. Show that the convergence rate of the backward error of (μ_k, v_k) is $\omega_k = O(\theta^k)$. Explain the convergence plot in Fig. 1 both qualitatively and quantitatively.

Hint: Backward stability is reached at about $15/|\log_{10}\theta| \approx 50/|\log_2 \theta|$ iteration steps.

Remark. According to Theorem 19.5, an *ill-conditioned* eigenvector x_1 implies $\lambda_1 \approx \lambda_2$, so that then, due to $\theta \approx 1$, power iteration converges only very slowly.

Inverse Iteration

20.5 We notice the basic, but extremely useful equivalence

$$(\lambda, x) \text{ eigenpair of } A \quad \Leftrightarrow \quad ((\lambda - \mu)^{-1}, x) \text{ eigenpair of } (A - \mu I)^{-1}.$$

With the help of a *shift* μ, that is closer to a simple eigenvalue λ than to the rest of the spectrum of A, the transformed eigenvalue $(\lambda - \mu)^{-1}$ can be made dominant for the transformed matrix $S = (A - \mu I)^{-1}$. By Theorem 20.4, power iteration applied to the matrix S now converges towards the eigenvector x. In this way, simple eigenpairs can be *selectively* computed.

20.6 This yields the *inverse iteration with shift* μ (H. Wielandt 1944), for $k = 1, 2, \ldots$:

$$\begin{aligned} (A - \mu I) w_k &= v_{k-1} && \text{(linear system of equations)} \\ v_k &= w_k / \|w_k\|_2 && \text{(normalization)} \\ \mu_k &= v_k' A v_k && \text{(Rayleigh quotient from §19.2)} \end{aligned}$$

Since the matrix of this linear system of equations is not changed during the iteration, just a *single* matrix factorization needs to be calculated (which is an enormous advantage for large dimensions, even for a limited number of iterations).

20.7 To avoid the explicit matrix multiplication Av_k during the evaluation of the Rayleigh quotients $\mu_k = v_k' A v_k$ and residuals $r_k = Av_k - \mu_k v_k$, the following auxiliary quantities are introduced:

$$z_k = \frac{v_{k-1}}{\|w_k\|_2} = Av_k - \mu v_k = r_k + (\mu_k - \mu)v_k, \quad \rho_k = v_k' z_k = v_k' A v_k - \mu = \mu_k - \mu.$$

This way we obtain $\mu_k = \mu + \rho_k$ and $r_k = z_k - \rho_k v_k$, thereby cutting the cost of each iteration step in half ($2m^2$ instead of $4m^2$ flop).

Program 14 (Inverse Iteration with Shift).

```
normA = norm(A,'fro');      % store ‖A‖_F
omega = inf;                % initialize the backward error
[L,R,p] = lu(A - muShift*eye(m),'vector'); % store the triangular decomp.
while omega > tol           % is the backward error still too large?
  w = R\(L\v(p));           % solve system of linear equations
  normW = norm(w);          % ‖w‖_2
  z = v/normW;              % auxiliary quantity z
  v = w/normW;              % new v
  rho = v'*z;               % auxiliary quantity ρ
  mu = muShift + rho;       % Rayleigh quotient
  r = z - rho*v;            % residual
  omega = norm(r)/normA;    % backward error estimate ω̃
end
```

20.8 Theorem 20.4 now readily implies a convergence result for inverse iteration:

Theorem. *Let it be possible to order the eigenvalues of a normal matrix A with respect to a given shift $\mu \in \mathbb{C}$ as*

$$|\lambda_1 - \mu| < |\lambda_2 - \mu| \leqslant \cdots \leqslant |\lambda_m - \mu|, \quad \textit{so that especially} \quad \theta = \left|\frac{\lambda_1 - \mu}{\lambda_2 - \mu}\right| < 1.$$

Let the vector x_1 be a normalized eigenvector corresponding to the eigenvalue λ_1. If the starting vector v_0 satisfies the condition $x_1' v_0 \neq 0$, inverse iteration with shift μ calculates approximate eigenpairs (μ_k, v_k) converging as

$$\sin \measuredangle(v_k, x_1) = O(\theta^k), \qquad |\mu_k - \lambda_1| = O(\theta^{2k}) \qquad (k \to \infty).$$

A Misguided Objection: Ill-Conditioning of the Linear System to be Solved

We assume that $A \in \mathbb{C}^{m\times m}$ is normal.

20.9 If the shift μ is very close to the spectrum of A, then the linear system of equations for the inverse iteration, that is,

$$(A - \mu I)w = v, \qquad \|v\|_2 = 1,$$

is ill-conditioned: indeed in keeping with §§19.3–19.4 we get

$$\kappa_2(A-\mu I) = \frac{\|A-\mu I\|_2}{\operatorname{dist}(\mu,\sigma(A))} = \frac{\max_{\lambda\in\sigma(A)}|\lambda-\mu|}{\min_{\lambda\in\sigma(A)}|\lambda-\mu|} \gg 1.$$

On the computer, we compute a $\hat{w}$ that only approximates w very *inaccurately*. Does this point towards a possible numerical *instability* of the inverse iteration?

20.10 The answer is a resounding "no!", because we are not at all interested in the vector w itself, but rather only in its *direction* (i.e., in the normalized vector $w/\|w\|_2$).[76] A backward stable solution $\hat{w}$ of the linear system actually satisfies

$$((A+E)-\mu I)\hat{w} = v, \qquad \|E\|_2 = O(\|A\|_2\cdot\epsilon_{\text{mach}}). \tag{20.2}$$

Were the perturbation E to the be same for all iteration steps (cf. the model for iterative refinement in §E.1), we would directly get convergence to an eigenvector of $A+E$ and thus achieve *backward stability*. With just a little more work, this analysis can be carried over to the case where E depends on the iteration index.

20.11 At the extreme, inverse iteration can be used to compute an eigenvector corresponding to a backward stable eigenvalue μ taken as shift. Then §19.4 implies

$$\operatorname{dist}(\mu,\sigma(A)) = O(\|A\|_2\cdot\epsilon_{\text{mach}}),$$

so that the iteration matrix $A-\mu I$ is even *numerically singular*. Combined with μ, an approximate eigenvector $\hat{w}$ calculated in course of inverse iteration forms an approximate eigenpair $(\mu,\hat{w})$. Using §19.1, its backward error can be bounded by

$$\omega = \frac{\|A\hat{w}-\mu\hat{w}\|_2}{\|A\|_2\|\hat{w}\|_2} = \frac{\|v-E\hat{w}\|_2}{\|A\|_2\|\hat{w}\|_2} \leqslant \frac{1+\|E\|_2\|\hat{w}\|_2}{\|A\|_2\|\hat{w}\|_2} = \frac{1}{\|A\|_2\|\hat{w}\|_2} + O(\epsilon_{\text{mach}}).$$

If $1/(\|A\|_2\|\hat{w}\|_2) = O(\epsilon_{\text{mach}})$, the eigenpair is thus already *backward stable* without the exact length of $\|w\|_2$ having played any particular role. In principle, if $\|v\|_2 = 1$, this magnitude is within reach as suggested by the following lower bound:

$$\frac{1}{\|A\|_2\|w\|_2} \geqslant \frac{1}{\|A\|_2\|(A-\mu I)^{-1}\|_2\|v\|_2} = \frac{\operatorname{dist}(\mu,\sigma(A))}{\|A\|_2} = O(\epsilon_{\text{mach}}).$$

Actually, for a randomly chosen normalized vector v, there is a high probability of equal order of magnitude for both sides of this bound.[77] We are thus led to claim:

[76] "The failure to grasp this fact has resulted in a lot of misguided papers." (Pete Stewart 1998)

[77] More specifically, random vectors v generated by "`v = randn(m,1); v = v/norm(v);`" satisfy (for a proof see §E.15), with probability $\geqslant 1-\delta$, the bound

$$\frac{1}{\|A\|_2\|w\|_2} \leqslant O(\delta^{-1}\epsilon_{\text{mach}}) \qquad (0<\delta\leqslant 1).$$

Using a backward stable approximate eigenvalue as shift, just a single step of inverse iteration applied to a random starting vector computes, with high probability, a backward stable approximate eigenpair.

Exercise. How can one avoid arithmetic exceptions entering during inverse iteration if the iteration matrix $A - \mu I$ is *exactly* singular in machine arithmetic?

Example. Here, despite the warning

```
Matrix is close to singular or badly scaled. Results may be inaccurate.
```

issued by MATLAB at line 6 of the following code, a backward stable eigenvector to a given eigenvalue has been found after just one single step of inverse iteration:

```
1 >> m = 1000; rng(847);                                % initialization
2 >> lambda = (1:m) + (-5:m-6)*i;                       % given eigenvalues
3 >> [Q,~] = qr(randn(m)); A = Q*diag(lambda)*Q';       % suitable normal matrix
4 >> mu = 1 - 5i;                                       % shift = known eigenvalue
5 >> v = randn(m,1); v = v/norm(v);                     % random start vector
6 >> w = (A - mu*eye(m))\v;                             % 1 step inverse iteration
7 >> omega = sqrt(m)*norm(A*w-mu*w)/norm(A,'fro')/norm(w) % Eq. (20.1)
8 omega =
9    1.3136e-15
```

21 QR Algorithm

21.1 For select eigenvalues λ of a matrix A corresponding Schur decompositions can be obtained in the form

$$Q'AQ = T = \left(\begin{array}{c|c} * & * \\ \hline & \lambda \end{array}\right).$$

The last row of this decomposition states $e_m'Q'AQ = \lambda e_m'$, so that for $x = Qe_m \neq 0$

$$x'A = \lambda x', \qquad x \neq 0. \tag{21.1}$$

Such an x is called a *left eigenvector* of A corresponding to the eigenvalue λ; (λ, x) is called a *left eigenpair*.

Remark. Adjunction takes (21.1) to the equivalent form $A'x = \bar{\lambda}x$. It therefore holds that $\sigma(A') = \overline{\sigma(A)}$ and the left eigenvectors of A correspond with the eigenvectors of A'. Hence, all of the concepts from this chapter carry over from eigenpairs to left eigenpairs.

21.2 Starting with $A_0 = A$, we now aim at *iteratively* computing just the last line of a Schur decomposition by constructing a sequence

$$A_{k+1} = Q_k'A_kQ_k \to \left(\begin{array}{c|c} * & * \\ \hline & \lambda \end{array}\right) \qquad (k \to \infty) \tag{21.2}$$

where all the base changes Q_k are meant to be *unitary*. Notably then, both A_k and A are unitarily *similar* which implies that spectrum and norm remain invariant:

$$\sigma(A_k) = \sigma(A), \qquad \|A_k\|_2 = \|A\|_2.$$

The last line of (21.2) amounts to saying that the last unit basis vector e_m asymptotically becomes a left eigenvector of A_k corresponding to the eigenvalue λ:

$$e_m' A_k \to \lambda e_m' \qquad (k \to \infty).$$

21.3 The partitioning

$$A_k = \left(\begin{array}{c|c} * & * \\ \hline r_k' & \lambda_k \end{array}\right)$$

shows the Rayleigh quotient for the approximate left eigenvector e_m of A_k to be given by

$$\lambda_k = e_m' A_k e_m$$

and the corresponding residual to be of the form

$$e_m' A_k - \lambda_k e_m' = \left(\, r_k' \;\middle|\; 0 \,\right).$$

The backward error of the perturbed left eigenpair (λ_k, e_m) is thus $\omega_k = \|r_k\|_2 / \|A\|_2$ and it vanishes in the limit if and only if $\|r_k\|_2 \to 0$ as required in (21.2).

21.4 Let us construct A_{k+1} from A_k by improving e_k as an approximate left eigenvector with *one* step of inverse iteration (with a yet unspecified shift μ_k):

(1) solve $w_k'(A_k - \mu_k I) = e_m'$;

(2) normalize $v_k = w_k / \|w_k\|_2$;

(3) extend to an orthonormal basis, i.e., construct a unitary $Q_k = \left(\, * \;\middle|\; v_k \,\right)$;

(4) change basis according to $A_{k+1} = Q_k' A_k Q_k$, such that v_k becomes $e_m = Q_k' v_k$.

21.5 The crucial point is now that we attain steps (1)–(3) *in one fell swoop* by using a normalized QR decomposition for the solution of (1):[78]

$$A_k - \mu_k I = Q_k R_k, \qquad R_k = \left(\begin{array}{c|c} * & * \\ \hline & \rho_k \end{array}\right),$$

[78] We follow G. W. Stewart: *Afternotes goes to Graduate School*, SIAM, Philadelphia, 1997, pp. 137–139.

where Q_k is unitary and R_k is upper triangular with a positive diagonal. From the last row of R_k, namely $e'_m R_k = \rho_k e'_m$, we obtain the last row of Q'_k in the form

$$\underbrace{e'_m Q'_k}_{\text{normalized vector}} = e'_m R_k (A_k - \mu_k I)^{-1} = \underbrace{\rho_k}_{>0} \cdot \underbrace{e'_m (A_k - \mu_k I)^{-1}}_{=w'_k},$$

that is,

$$e'_m Q'_k = v'_k,$$

which means that the last column of Q_k is already the vector v_k. In addition, step (4) can be briefly written as:

$$A_{k+1} = Q'_k A_k Q_k = Q'_k (A_k - \mu_k I) Q_k + \mu_k I = Q'_k Q_k R_k Q_k + \mu_k I = R_k Q_k + \mu_k I.$$

21.6 We have thus reached one of the most remarkable, important and elegant algorithms of the 20th century, namely the *QR* algorithm, independently[79] developed from John Francis (1959) and Vera Kublanovskaya (1960): for $A_0 = A$, a sequence of suitable shifts μ_k and $k = 0, 1, 2, \ldots$ the algorithm proceeds by

$$A_k - \mu_k I = Q_k R_k \qquad (QR \text{ factorization}),$$

$$A_{k+1} = R_k Q_k + \mu_k I \qquad (\text{matrix product } RQ).$$

For *constant* shift $\mu_k = \mu$ this iteration is, by construction, nothing else than inverse iteration with continued unitary basis changes that are keeping the left eigenvector approximation fixed at the last unit vector e_m. In this case, the convergence theory from Theorem 20.8 is directly applicable.

21.7 Although the sequence of the A_k is already converging itself, at least under proper assumptions, towards a Schur decomposition (in fact even *without* shifts, see §E.2), it is nevertheless more efficient—and conceptually also clearer for the selection of suitable shifts μ_k—to *split off* sufficiently accurate eigenvalues *row-wise* from bottom to top. The *QR* algorithm is therefore stopped when, partitioning

$$A_n = \left(\begin{array}{c|c} B_n & w_n \\ \hline r'_n & \lambda_n \end{array}\right)$$

as in §21.3, there is backward stability of the left eigenpair (λ_n, e_m):

$$\|r_n\|_2 = O(\|A\|_2 \cdot \epsilon_{\text{mach}}).$$

At this point, we perform a *numerical deflation* in such a way that

$$A_n \mapsto \tilde{A}_n = \left(\begin{array}{c|c} B_n & w_n \\ \hline 0 & \lambda_n \end{array}\right) = A_n - E_n, \qquad E_n = e_m \left(r'_n \;\middle|\; 0 \right),$$

[79] G. Golub, F. Uhlig: *The QR algorithm: 50 years later*, IMA J. Numer. Anal. 29, 467–485, 2009.

i.e., we simply make e_m an *exact* left eigenvector of a slightly *perturbed* matrix $\tilde{A}_n$. By construction, this perturbation fulfills

$$\|E_n\|_2 = \|e_m\|_2 \cdot \|r_n\|_2 = O(\|A\|_2 \cdot \epsilon_{\text{mach}}).$$

From

$$A_n = Q'_{n-1} \cdots Q'_0 A \underbrace{Q_0 \cdots Q_{n-1}}_{=U_n} = U'_n A U_n,$$

the unitarity of U_n and the unitary *invariance*[80] of the spectral norm it follows that

$$\tilde{A}_n = U'_n \tilde{A} U_n, \qquad \tilde{A} = A + \underbrace{E}_{U_n E_n U'_n}, \qquad \|E\|_2 = O(\|A\|_2 \cdot \epsilon_{\text{mach}}).$$

In this way, $(\lambda_n, U_n e_m)$ is an *exact* left eigenpair of the perturbed matrix $\tilde{A}$ and therefore a *backward stable* left eigenpair of A.

Program 15 (QR Algorithm).

```
function [lambda,U,B,w] = QR_algorithm(A)

[m,~] = size(A); I = eye(m); U = I;    % initialization
normA = norm(A,'fro');                 % store ‖A‖_F
while norm(A(m,1:m-1)) > eps*normA     % backward error still too large?
  mu = ... ;                           % select shift μ_k (§§21.10/21.12)
  [Q,R] = qr(A-mu*I);                  % QR decomposition
  A = R*Q+mu*I;                        % RQ multiplication
  U = U*Q;                             % transformation matrix U_k = Q_1 ··· Q_k
end
lambda = A(m,m);                       % eigenvalue
B = A(1:m-1,1:m-1);                    % deflated matrix
w = A(1:m-1,m);                        % last column in Schur decomposition
```

21.8 Similarly, we notice that every backward stable left eigenpair of the deflated matrix B_n belongs to a backward stable left eigenpair of A. If we now apply the QR algorithm to the matrix B_n, which is one dimension smaller, followed by numerical deflation, and repeat this process over and over again, we obtain, column by column, a backward stable Schur decomposition of A (cf. §18.5). By suppressing indices and skipping backward stable perturbations for simplicity, the structure of one step of this process can be written as follows: if

$$Q'AQ = \left(\begin{array}{c|c} B & W \\ \hline & T \end{array}\right)$$

[80] Once more we see the fundamental importance of *unitary* transformations in numerical analysis: perturbations do not get amplified.

is already calculated with Q unitary and T upper triangular, the application of the QR algorithm on B with subsequent deflation provides

$$U'BU = \left(\begin{array}{c|c} \tilde{B} & w \\ \hline & \lambda \end{array}\right).$$

Combined, these two sub-steps result in the same structure as we started with,

$$\left(\begin{array}{c|c} U' & \\ \hline & I \end{array}\right) Q'AQ \left(\begin{array}{c|c} U & \\ \hline & I \end{array}\right) = \left(\begin{array}{c|c} U'BU & U'W \\ \hline & T \end{array}\right)$$

$$= \left(\begin{array}{cc|c} \tilde{B} & w & \\ & \lambda & U'W \\ \hline & & T \end{array}\right) = \left(\begin{array}{cc|c} \tilde{B} & * & \\ & \lambda & * \\ \hline & & T \end{array}\right) = \left(\begin{array}{c|c} \tilde{B} & * \\ \hline & \tilde{T} \end{array}\right),$$

but with the dimension of the new triangular matrix $\tilde{T}$ in the lower right block being increased by one and the eigenvalue λ being added to its diagonal.

21.9 In MATLAB, this process of computing a Schur decomposition $Q'AQ = T$ can be executed *in situ* as follows:

Program 16 (Schur Decomposition).

object	MATLAB for column k
W	`T(1:k,k+1:m)`
w	`T(1:k-1,k)`
λ	`T(k,k)`

```
T = zeros(m); Q = eye(m);                % initialization
for k=m:-1:1                             % column-wise from back to front
  [lambda,U,A,w] = QR_algorithm(A); % deflation with the QR algorithm
  Q(:,1:k) = Q(:,1:k)*U;                 % transform first k columns of Q
  T(1:k,k+1:m) = U'*T(1:k,k+1:m);        % transform first k remaining rows of T
  T(1:k,k) = [w;lambda];                 % new kth column of T
end
```

Here, `QR_algorithm(A)` is the function from Program 15.

Shift Strategies and Convergence Results

21.10 A look at §21.3 suggests a candidate of the shift μ_k in step $A_k \mapsto A_{k+1}$ of the QR algorithm: the Rayleigh quotient

$$\mu_k = e_m' A_k e_m = \lambda_k.$$

We refer to this as *Rayleigh shift*. It specifies line 6 in Program 15 to complete as:

```
6 mu = A(m,m);
```

Using the notation from §21.3, the following convergence of the residual can be shown (for a proof see §E.7–E.13): if the initial residual $\|r_0\|_2$ is sufficiently small, it holds

$$\|r_{k+1}\|_2 = \begin{cases} O(\|r_k\|_2^2) & \text{in general,} \\ O(\|r_k\|_2^3) & \text{for } \textit{normal} \text{ matrices.} \end{cases}$$

Such a convergence is called *locally quadratic* and *locally cubic* respectively.[81]

Exercise. Consider an iteration with quadratic (cubic) convergence of the backward error being applied to a well-conditioned problem. Show that the number of correct digits will, eventually, at least double (triple) in every step.

21.11 Since for real matrices A the QR algorithm with Rayleigh shift generates a sequence A_k of *real* matrices, *complex* eigenvalues cannot be directly approximated. Other than in (21.2), the sequence A_k would then converge to a limit of the form

$$\left(\begin{array}{c|cc} * & * & \\ \hline & \alpha & \beta \\ & \gamma & \lambda \end{array}\right).$$

According to Theorem 20.8, such a convergence does not only occur when the lower right 2×2-matrix has conjugate-complex eigenvalues but also when both of its eigenvalues are *symmetrical* to the Rayleigh shift $\mu = \lambda$.

Example. Because of $RQ = A$, the permutation matrix

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \underbrace{Q}_{=A} \underbrace{R}_{=I}$$

with the eigenvalues $\lambda = \pm 1$ is a *fixed point* of the QR algorithm with shift $\mu = 0$.

[81] "local": for starting values in the *vicinity* of a limit; "global": for all admissible starting values

21.12 In 1968, J. Wilkinson proposed the following shift strategy to address this problem: given

$$A_k = \left(\begin{array}{c|cc} * & \multicolumn{2}{c}{*} \\ \hline & \alpha_k & \beta_k \\ * & \gamma_k & \lambda_k \end{array}\right). \tag{21.3}$$

one selects as shift μ_k the eigenvalue of the 2×2 matrix[82]

$$\begin{pmatrix} \alpha_k & \beta_k \\ \gamma_k & \lambda_k \end{pmatrix}$$

that is *closest* to the Rayleigh shift λ_k; we refer to this as the *Wilkinson shift*. In Program 15 line 6 is therefore completed as:

```
6 mu = eig(A(m-1:m,m-1:m)); [~,ind] = min(abs(mu-A(m,m))); mu = mu(ind);
```

As with the Rayleigh shift, the convergence is locally quadratic in general or, for normal matrices, locally cubic (for a proof see §E.14). In practice, one needs approximately 2–4 iterations per eigenvalue on average.[83]

Exercise. (a) Find how many iterative steps of the QR algorithm Program 16 needs on average per eigenvalue for a random 100×100 matrix. (b) Show that for a real self-adjoint matrix A, the Wilkinson shift is always real.

Remark. Unfortunately, even the Wilkinson shift is not completely *foolproof*—except for the extremely important case of real symmetric *tridiagonal* matrices (cf. Phase 2 in §21.14), where global convergence can actually be proven.[84] For instance, the permutation matrix

$$A = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

is a fixed point of the QR algorithm with Wilkinson shift (which is $\mu = 0$). Therefore, several more fine-tuned shift strategies have been developed such as *multi-shifts*, *exception-shifts*, etc.. The search for a shift strategy that can be proven to converge for *every* matrix is still an open mathematical problem.

Cost Reduction

21.13 Even though only a limited number $O(1)$ of iterations per eigenvalue is needed, the QR algorithm would still be too costly for the calculation of a Schur

[82] Eigenvalues of 2×2 matrices are computed as the solution of a quadratic equation; we refer to §§14.1–14.2 on a discussion of the numerical stability thereof.

[83] As a bonus, eigenvalues calculated later in the process benefit from the steps of the QR algorithm that have addressed preceding ones.

[84] J. Wilkinson: *Global convergence of tridiagonal QR algorithm with origin shifts*, Linear Algebra and Appl. 1, 409–420, 1968; a more structural proof can be found in W. Hoffmann, B. N. Parlett: *A new proof of global convergence for the tridiagonal QL algorithm*, SIAM J. Numer. Anal. 15, 929–937, 1978.

decomposition if one applied it to A without any preprocessing:

$$\#\text{flop for } \underbrace{\text{deflation}}_{\text{Program 15}} = \underbrace{\#\text{ iterations}}_{=O(1)} \cdot \underbrace{\#\text{ flop for one } QR \text{ step}}_{=O(m^3)} = O(m^3)$$

$$\underbrace{\text{total cost}}_{\#\text{ flop Program 16}} = O(m^3 + (m-1)^3 + \cdots + 1) = O(m^4).$$

The goal of the rest of this section is to reduce the cost from $O(m^4)$ to just $O(m^3)$ (which is then comparable to the cost of a QR decomposition).

21.14 To achieve this goal, the calculation of a Schur decomposition is broken down into *two phases*:

(1) unitary *reduction* of A with a *direct* $O(m^3)$ algorithm to some "simple" form

$$H = Q'AQ \in \mathcal{H};$$

(2) use of the QR algorithm on $H \in \mathcal{H}$.

To this end, the space $\mathcal{H}$ of "simple" matrices should be invariant under a QR step, that is

$$\mathcal{H} \ni H - \mu I = QR \;\Rightarrow\; H_* = RQ + \mu I \in \mathcal{H}.$$

Here, computing the QR decomposition and the product RQ should both only require $O(m^2)$ flop. Recalling §9.11, the *upper Hessenberg matrices* invite themselves to form such a space $\mathcal{H}$: for them the computational cost of a QR step is $\doteq 6m^2$.

21.15 The invariance of $\mathcal{H}$ under a QR step of phase (2) follows directly from:

Lemma. *Let $\mathcal{H}$ and $\mathcal{R}$ denote the matrix spaces of upper Hessenberg and upper triangular matrices respectively. It then holds*

$$\mathcal{H} \cdot \mathcal{R} \subset \mathcal{H}, \qquad \mathcal{R} \cdot \mathcal{H} \subset \mathcal{H}.$$

If $H = QR$ is the QR decomposition of $H \in \mathcal{H}$ as in §9.11, then it holds $Q \in \mathcal{H}$.

Proof. According to §§5.2 and 9.11, the matrix spaces $\mathcal{R}$ and $\mathcal{H}$ can be characterized with the help of the canonical subspaces $V_k = \operatorname{span}\{e_1, \ldots, e_k\} \subset \mathbb{K}^m$ by

$$R \in \mathcal{R} \;\Leftrightarrow\; RV_k \subset V_k \quad (k = 1:m), \qquad H \in \mathcal{H} \;\Leftrightarrow\; HV_k \subset V_{k+1} \quad (k = 1:m-1).$$

Step 1. For $R \in \mathcal{R}$ and $H \in \mathcal{H}$ we have $HR, RH \in \mathcal{H}$, since

$$HR(V_k) \subset HV_k \subset V_{k+1}, \qquad RH(V_k) \subset RV_{k+1} \subset V_{k+1} \qquad (k = 1:m-1).$$

Step 2. If $H = QR$ is the QR decomposition of an *invertible* $H \in \mathcal{H}$, it follows from the group structure of $\mathcal{R}$ with respect to multiplication (Lemma 5.3) that $R^{-1} \in \mathcal{R}$ and therefore $Q = HR^{-1} \in \mathcal{H}$. Since, as shown in §9.11, QR decomposition can be made continuous in H, it follows that $Q \in \mathcal{H}$ also for a general $H \in \mathcal{H}$ by approximating $H_n \to H$ with a sequence of invertible $H_n \in \mathcal{H}$. □

21.16 As for phase (1), the column-wise unitary reduction of A to an upper Hessenberg matrix H is best described by looking at the first column: from

$$A = \left(\begin{array}{c|c} \alpha & y' \\ \hline x & B \end{array}\right)$$

using a *full* QR decomposition $x = \xi U e_1$ of $x \in \mathbb{C}^{(m-1)\times 1}$, we get

$$\underbrace{\left(\begin{array}{c|c} 1 & \\ \hline & U' \end{array}\right)}_{=Q_1'} A \underbrace{\left(\begin{array}{c|c} 1 & \\ \hline & U \end{array}\right)}_{=Q_1} = \left(\begin{array}{c|c} \alpha & y'U \\ \hline \xi & \\ 0 & \\ \vdots & U'BU \\ 0 & \end{array}\right).$$

If we execute the same action on $U'BU$ and repeat the process, we thus finally obtain the desired Hessenberg matrix H, column-wise from left to right.

Exercise. Show that the computational cost of this algorithm is $O(m^3)$ if one uses an implicit representation of Q (by just storing all factors, e.g., Givens rotations, and not expanding their product).

Exercise. Show that if A is self-adjoint, the preprocessed matrix H is also self-adjoint and therefore *tridiagonal*. What is the cost of the two phases in this case?

Answer: Phase (1) costs $O(m^3)$, whereas Phase (2) costs only $O(m^2)$ flop.

21.17 MATLAB offers access to LAPACK programs of the algorithms from this section (for LAPACK 'GE' stands for a general Hermitian matrix, 'SY' for a real symmetrical matrix and 'HE' for a complex hermitian matrix; cf. §8.1):

Problem	MATLAB	LAPACK
Hessenberg $H = Q'AQ$	`[Q,T] = hess(A)`	xGEHR/xSYTRD/xHETRD
Schur $T = Q'AQ$	`[Q,T] = schur(A,'complex')`	xGEES/xSYEV/xHEEV
all eigenvalues $\lambda_j(A)$ $(j = 1:m)$	`lambda = eig(A)`	xGEEV/xSYEV/xHEEV

The calculation of *just* the complete set of eigenvalues is computationally much less expensive than a fully fledged Schur decomposition since then there is no need to deal with the unitary transformations.

Example. Here are a few execution times for random real 2000×2000 matrices:

Command	General matrix [s]	Symmetric matrix [s]
`[L,R,p] = lu(A,'vector');`	0.11	0.11
`R = triu(qr(A));`	0.23	0.23
`[Q,R] = qr(A);`	0.44	0.44
`H = hess(A);`	2.0	0.83
`[Q,H] = hess(A);`	2.2	1.0
`lambda = eig(A);`	3.8	0.89
`[Q,T] = schur(A,'complex');`	13	1.4

21.18 In the case of *normal* matrices, the columns of the unitary factor Q of a Schur decomposition $Q'AQ = T$ form an orthonormal basis of eigenvectors (see §18.6). For general matrices, the MATLAB command `[V,D] = eig(A)` aims at calculating a numerical solution $V \in \mathbb{C}^{m\times m}$ of

$$AV = VD, \qquad D = \operatorname{diag}(\lambda_1, \ldots, \lambda_m).$$

The columns of V would therefore have to be eigenvectors of A; if A does *not*, however, have a basis of eigenvalues, V must be singular—such a matrix A is then called *non-diagonalizable*. On the computer this manifests itself by returning a numerically singular matrix V (cf. §11.10) with

$$\kappa_2(V) \gtrsim \epsilon_{\text{mach}}^{-1}.$$

Remark. Individual *select* eigenvalues can be best calculated as described in §20.11.

Exercise. Discuss the result of `[V,D] = eig([1 1; 0 1])`.

Exercise. Let $T \in \mathbb{R}^{m\times m}$ be a symmetric tridiagonal matrix of the form

$$T = \begin{pmatrix} a_1 & b_1 & & \\ b_1 & a_2 & \ddots & \\ & \ddots & \ddots & b_{m-1} \\ & & b_{m-1} & a_m \end{pmatrix} \qquad (b_j \neq 0).$$

This exercise develops an efficient algorithm for the calculation of the eigenvalues of T that are within a given half open interval $[\alpha, \beta)$.

- Show that triangular decomposition *without* pivoting leads (when feasible) to a decomposition of the form $T - \mu I = LDL'$, where $D = \operatorname{diag}(d_1, \ldots, d_m)$ is given by the recursion

$$d_1 = a_1 - \mu, \qquad d_j = a_j - \mu - \frac{b_{j-1}^2}{d_{j-1}} \quad (j = 2:m). \tag{21.4}$$

The indices of inertia of $T - \mu I$ are then given by

$$\nu_\pm(\mu) = \#\{\lambda \in \sigma(T) : \lambda \gtrless \mu\} = \#\{d_j : d_j \gtrless 0\},$$
$$\nu_0(\mu) = \#\{\lambda \in \sigma(T) : \lambda = \mu\} = \#\{d_j : d_j = 0\}$$

with $\nu_0(\mu) \in \{0, 1\}$. Characterize the feasibility of the decomposition in exact arithmetic.

- Show that the calculation of the indices of inertia based on (21.4) is *backward stable*. Using the standard model (12.1), prove that the *actually* calculated quantities $\hat{d}_j$ have the same signs as the $\tilde{d}_j$ obtained from a perturbed recursion of the form

$$\tilde{d}_1 = a_1 - \mu, \qquad \tilde{d}_j = a_j - \mu - \frac{\tilde{b}_{j-1}^2}{\tilde{d}_{j-1}} \quad (j = 2 : m).$$

  Find a suitable estimate for the relative error of $\tilde{b}_j$.
- Show that in machine arithmetic (21.4) makes sense even when $d_{j_*} = 0$ for a $j_* < m$. Furthermore, the calculated index $\nu_-(\mu)$ would remain *invariant* (but not $\nu_+(\mu)$) if one prevented $d_{j_*} = 0$ with a sufficiently small perturbation of the form $a_{j_*} \pm \epsilon$.
- Show that the count $\nu(\alpha, \beta) = \#\big(\sigma(T) \cap [\alpha, \beta)\big)$ is given by $\nu(\alpha, \beta) = \nu_-(\beta) - \nu_-(\alpha)$.
- Implement the following *bisection algorithm*: by means of continued bisection of intervals, the starting interval $[\alpha, \beta)$ is broken down into sub-intervals $[\alpha_*, \beta_*)$ which are only however further considered in this process if

$$\nu_* = \#\big(\sigma(T) \cap [\alpha_*, \beta_*)\big) > 0.$$

  Bisection of a sub-interval is stopped if it is sufficiently accurate: $\beta_* - \alpha_* \leqslant \text{tol}$. The output of the algorithm is a list of intervals $[\alpha_*, \beta_*)$ with their count of eigenvalues.
- Estimate the computational cost of the algorithm as a function of m, $\nu(\alpha, \beta)$ und tol.
- Generalize the algorithm to arbitrary real symmetric matrices A by adding a Phase (1) similar to §21.14. Why does the assumption $b_j \neq 0$ on T not pose a restriction then?

Exercise. Let the matrices $A, B \in \mathbb{R}^{m\times m}$ be symmetric, and B be additionally positive definite. Consider the *generalized* eigenvalue problem of the form

$$Ax = \lambda Bx, \qquad x \neq 0. \tag{21.5}$$

The set of all generalized eigenvalues λ is labeled $\sigma(A, B)$.

- Give a formula for the (normwise relative) backward error ω of a perturbed eigenpair $(\tilde{\lambda}, \tilde{x})$, $\tilde{x} \neq 0$. Here only the matrix A should get perturbed.
- With respect to perturbations of A measured with the 2-norm, the *absolute* condition number of a *simple* generalized eigenvalue λ (let x be a corresponding eigenvector) is given by[85]

$$\kappa_{\text{abs}}(\lambda; A) = \frac{x'x}{x'Bx}.$$

[85] V. Frayssé, V. Toumazou: *A note on the normwise perturbation theory for the regular generalized eigenproblem*, Numer. Linear Algebra Appl. 5, 1–10, 1998.

From this, deduce the bounds $|\lambda|\,\|A\|_2^{-1} \leqslant \kappa_{\text{abs}}(\lambda; A) \leqslant \|B^{-1}\|_2$ (the lower one only in the case of $A \neq 0$) and make the following statement more precise:

> *A generalized eigenvalue λ is ill-conditioned if B is ill-conditioned and $|\lambda|$ is relatively large.*

Hint: Without loss of generality we can assume that $\|B\|_2 = 1$. Why?

- Use ω and $\kappa_{\text{abs}}(\lambda; A)$ for an algorithmically simple bound of the *absolute* forward error of λ.
- Show that a decomposition of the form $B = GG'$ transforms the generalized eigenvalue problem (21.5) to the following equivalent symmetric eigenvalue problem:

$$A_G z = \lambda z, \qquad A_G = G^{-1} A (G^{-1})', \qquad z = G'x \neq 0. \tag{21.6}$$

  Get formulas for such G from both the Cholesky and a Schur decomposition of B.
- Apply the inverse iteration with a shift μ from §§20.6–20.7 to A_G and transform it back so that only the matrices A and B appear, but not the factor G. State a suitable convergence result.
- Implement the thus generalized inverse iteration efficiently by modifying Program 14. Along with the generalized eigenpair (λ, x) the output should include the calculated *absolute* condition number $\kappa_{\text{abs}}(\lambda; A)$ as well as a bound of the absolute forward error of the generalized eigenvalue λ.
- With the help of the transformation (21.6), write a function `eigChol(A,B)` and a function `eigSchur(A,B)`, which both calculate $\sigma(A, B)$ based on the MATLAB command `eig(`A_G`)`. How many flop are needed for each of the two variants?
- With the three programs developed in this problem as well as the MATLAB command `eig(A,B)`, calculate all the eigenvalues for the following example:

```
1  A = [1.0 2.0 3.0; 2.0 4.0 5.0; 3.0 5.0 6.0];
2  B = [1.0e-6 0.001 0.002; 0.001 1.000001 2.001;
     0.002 2.001 5.000001];
```

  Assess the accuracy of the resulting eigenvalues (stability, number of correct digits) and rate the methods. Take the condition number of the problem into account.

Appendix

A MATLAB: A Very Short Introduction

MATLAB (MATrix LABoratory) is proprietary commercial software that is used in both industry and academia for numerical simulations, data acquisition and data analysis. It offers an elegant interface to matrix based numerical calculations, as well as state of the art optimized BLAS libraries (Basic Linear Algebra Subprograms) from processor manufacturers and the high-performance Fortran library LAPACK for systems of linear equations, matrix decompositions and eigenvalue problems. This all is accessible by means of a *simple scripting language*.

General Commands

A.1 Help:

```
help command, doc command
```

shows the help text of a *command* in the console or browser.

A.2 ',' and ';' separate commands, whereby ';' suppresses displaying the output.

A.3 Information:

```
whos
```

provides information about the variables in memory.

A.4 Measuring execution time:

```
tic, statements, toc
```

executes the *statements* and measures the computation time required.

A.5 Comments:

```
% comments can be written behind a percent sign
```

Matrices

A.6 MATLAB identifies scalars with 1×1 matrices, and column vectors (row vectors) of dimension m with $m \times 1$- ($1 \times m$-) matrices.

A.7 Assignment:

`A = expression;`

assigns the variable `A` the value of *expression*.

A.8 `+,-,*` are matrix addition, subtraction, multiplication. `A/B` computes the solution X to the linear system $A = XB$ and `A\B` the solution X to $AX = B$.

A.9 `.*` is the componentwise multiplication of two matrices.

A.10 `A'` is the adjoint matrix of `A`.

A.11 Matrix input:

```
A = [ 0.32975 -0.77335  0.12728;
     -0.17728 -0.73666  0.45504];
```

or

```
A = [ 0.32975 -0.77335  0.12728; -0.17728 -0.73666  0.45504];
```

assigns the following matrix to the variable `A`:

$$\begin{pmatrix} 0.32975 & -0.77335 & 0.12728 \\ -0.17728 & -0.73666 & 0.45504 \end{pmatrix}.$$

A.12 Identity matrix:

`eye(m)`

generates the identity matrix of dimension `m`.

A.13 Zero matrix:

`zeros(m,n), zeros(m)`

generates a matrix of zeros with the dimension `m` $\times$ `n` and `m` $\times$ `m` respectively.

A.14 Ones matrix:

`ones(m,n), ones(m)`

generates a matrix of ones with the dimension `m` $\times$ `n` and `m` $\times$ `m` respectively.

A.15 Random matrix with with uniformly distributed entries:

`rand(m,n), rand(m)`

generates a matrix of i.i.d. random entries, uniformly distributed on $[0,1]$, with the dimension `m` $\times$ `n` and `m` $\times$ `m` respectively.

A.16 Random matrices with normally distributed entries:

`randn(m,n), randn(m)`

generates a matrix of i.i.d. random entries, having standard normal distribution, with the dimension `m` $\times$ `n` and `m` $\times$ `m` respectively.

A.17 Index vectors:

`j:k, j:s:k`

are the row vectors `[j,j+1,...,k-1,k]` and `[j,j+s,...,j+m*s]` with

$$m = \lfloor (k-j)/s \rfloor.$$

A.18 Components:

`A(j,k)`

is the element of the matrix `A` in the `j`th row and `k`th column.

A.19 Submatrices:

`A(j1:j2,k1:k2)`

is the submatrix of the matrix `A`, in which the row indices are `j1` to `j2` and the column indices are `k1` to `k2`.

`A(j1:j2,:), A(:,k1:k2)`

are the submatrices built from rows `j1` to `j2` and from the columns `k1` to `k2`.

A.20 Matrix as vector:

`v = A(:)`

stacks the columns of $A \in \mathbb{K}^{m\times n}$ consecutively on top of one another to make a column vector $v \in \mathbb{K}^{mn}$.

A.21 Closing index:

`x(end)`

is the last component of the vector `x` and

`A(end,:), A(:,end)`

is the last row and column respectively of the matrix `A`.

A.22 Triangular matrices:

`tril(A)`, `triu(A)`

extracts the lower and upper triangular part of the matrix `A` and fills up with zeros.

A.23 Dimension of a matrix:

`size(A,1)` bzw. `size(A,2)`

outputs the number of rows and columns of the matrix `A` respectively;

`size(A)`

returns the row vector `[`*number of rows*`,`*number of columns*`]`.

A.24 Dimension of a vector:

`length(x)`

returns the dimension of the column or row vector `x`.

Functions

A.25 The definition of a MATLAB function `myfunc` begins with the line

```
function [o_1,o_2,...,o_n] = myfunc(i_1,i_2,...,i_m)
```

where `i_1,i_2,...,i_m` represent the input variables and `o_1,o_2,...,o_n` represent the output variables. The function has to be saved as the file `myfunc.m` in the project directory. The function is called in the command line (or in further function definitions) by:

```
[o_1,o_2,...,o_n] = myfunc(i_1,i_2,...,i_m)
```

Output variables can be ignored by omitting the last variables or by replacing them with the place holder '~':

```
[~,o_2] = myfunc(i_1,i_2,...,i_m)
```

If there is only one output variable, one can omit the square brackets:

```
o_1 = myfunc(i_1,i_2,...,i_m)
```

A.26 Function handle:

```
f = @myfunc;
[o_1,o_2,...,o_n] = f(i_1,i_2,...,i_m)
```

A function handle can be created with the '@' symbol and assigned to a variable that then acts like the function itself.

A.27 Anonymous function:

```
f = @(i_1,i_2,...,i_m) expression;
```

generates a handle for a function that assigns the value of the expression to `i_1,i_2,...,i_m`. The call is executed by

```
o_1 = f(i_1,i_2,...,i_m)
```

Control Flow

A.28 Conditional branches:

```
if expression
    instructions
elseif expression
    instructions
else
    instructions
end
```

The *instructions* are executed if the real part of all entries of the (matrix valued) *expression* is non-zero; 'elseif' and else' are optional and multiple 'elseif' can be used.

A.29 for loops:

```
for Variable = vector
    instructions
end
```

executes *instructions* multiple times where the *variable* is successively assigned the values of the components of *vector*.

A.30 while loops:

```
while expression
    instructions
end
```

executes *instructions* repeatedly as long as the real part of all entries of the (matrix valued) expressions is non-zero.

A.31 'break' stops the execution of a for or a while loop.

A.32 'continue' jumps to the next instance of a for or a while loop.

A.33 'return' leaves the current function by returning control to the invoking function.

Logic Functions

A.34 Comparisons:

```
A == B, A ~= B, A <= B, A >= B, A < B, A > B
```

componentwise comparisons of matrices; 1 stands for 'true' and 0 for 'false'.

A.35 Test for "all" or "at least one" true element:

```
all(A < B), any(A < B)
```

tests whether all comparisons, or at least one, is true; equivalently valid for the other operators ==, ~=, <=, >=, >.

A.36 Logic Boolean operations:

```
a && b, a || b
```

are the logical 'and' and 'or' for scalar values a and b. Every non-zero value is understood as true, and the value zero as 'false'; the result is 0 or 1.

A.37 Negation:

~expression, not(*expression*)

both return the logical negation of *expression*.

A.38 Test for an empty matrix:

```
isempty(A)
```

is 1 if the matrix A is empty, otherwise 0.

Componentwise Operations

A.39 Componentwise ("point-wise") multiplication, division and power:

```
A.*B, A./B, A.^2, A.^B
```

A.40 Componentwise functions:

```
abs(A), sqrt(A), sin(A), cos(A)
```

B Julia: A Modern Alternative to MATLAB

B.1 Since 2012, the highly performant, expressive and dynamical programming language Julia (open source with the MIT/GPL license) has been developed by a group lead by the mathematician Alan Edelman at MIT. Julia, by means of its modern design, is well suited for both the requirements of scientific computing, as well as more generalized programming tasks and includes elements such as

- JIT-compilation using LLVM,[86]
- parametric polymorphism and type inference at compile-time,
- object oriented programming based on multiple dispatch,
- homoiconicity and „hygienic“ macro programming,
- the ability to directly call C- and Fortran libraries.

This way Julia combines the expressiveness of programming languages like MATLAB and Python with the performance of programming languages like C, C++ and Fortran. For a more complete account and introduction, I strongly recommend reading the following paper:

> J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah: *Julia: A Fresh Approach to Numerical Computing*, SIAM Review 59, 65–98, 2017.

An international and extremely active developer community has augmented the functionality of Julia by more than 1500 packages, pushing Julia into the ranks of one of the fastest growing programming languages ever. By means of its modern concept, high performance, and free accessibility, Julia has already become a highly attractive alternative to MATLAB and may very well overtake in the university setting in the near future.

In order to aid the commendable transition, I have listed all MATLAB programs of this book below in Julia 0.6 and will go into the differences and advantages for each of these examples.[87]

B.2 Julia's language elements for numerical linear algebra are basically very closely related to MATLAB. I refer you to the 650-pages reference manual and the help function for details:

```
?command
```

[86] "Low Level Virtual Machine", a modular compiler infrastructure architecture that is also the basis of the highly optimized C compiler `Clang`.

[87] In order to easily test Julia, the Julia environment juliabox.com allows one to use Julia directly in a browser without local installation.

One small *syntactical* difference is the use of square brackets instead of round brackets for matrix indices:

```
A[j,k], A[j1:j2,k1:k2], A[j1:j2,:], A[:,k1:k2], A[:]
```

B.3 In contrast to MATLAB, vectors and column vectors (as well as 1×1 matrices and scalars) are *not* identified in Julia, but rather treated as fundamentally different types (type differences lead to distinctively compiled programs and a great increase in Julia's execution speed, since types no longer must be identified at run-time). When working with column vectors, this phenomenon is barely noticeable, since matrix-vector products are compatible with matrix-column-vector products. When working with *row vectors* however, one must take care, since *both* of the expressions

```
A[:,k], A[j,:]
```

create a vector from the corresponding components of the columns and rows of the matrix A. Explicit generation of a column or row vector requires the following (MATLAB compatible) syntax:

```
A[:,k:k], A[j:j,:]
```

Hence, the table from §3.2 changes in Julia as follows:

meaning	formula	Julia
components of x	ξ_k	`x[k]`
components of A	α_{jk}	`A[j,k]`
column vectors of A	a^k	`A[:,k]` and `A[:,k:k]`
row vectors of A	a'_j	`A[j:j,:]`
submatrix of A	$(\alpha_{jk})_{j=m:p,k=n:l}$	`A[m:p,n:l]`
adjunct matrix of A	A'	`A'`
matrix product	AB	`A*B`
identity matrix	$I \in \mathbb{R}^{m\times m}$, $\mathbb{C}^{m\times m}$	`eye(m)`, `complex(eye(m))`
zero matrix	$0 \in \mathbb{R}^{m\times n}$, $\mathbb{C}^{m\times n}$	`zeros(m,n)`, `complex(...)`

B.4 One significant *semantic* difference between MATLAB and Julia is the passing of matrices for assignments or function calls. Julia actually passes references to an object ("call by reference") so that in the following example, the variables `A` and `B` both refer to the *same* matrix in memory:

```
1 >> A = [1 2; 3 4];
2 >> B = A;
3 >> B[1,1] = -1;
4 >> A[1,1]
5 -1
```

In comparison, MATLAB passes copies of the values ("call by value") and, for this example, would return the original value of `1` for `A(1,1)`. In Julia, this behavior can be achieved by creating an explicit copy:

```
B = copy(A)
```

B.5 As with MATLAB, Julia also creates a copy for "slices" such as `A[:,k:k]` (so here a copy of a $m \times 1$ submatrix), but also offers direct *views* of A in memory with the commands `view(A,:,k:k)` for a $m \times 1$ dimensional matrix and `view(A,:,k)` for an m dimensional vector. Alternately, one can use the `@view` macro:

```
x = A[:,k] # the vector x is a copy of the kth column of A
y = @view A[:,k] # the vector y is identical with the kth column of A in memory
```

These views do not only save space in memory, but often also execution time by avoiding unnecessary copy operations.

B.6 Matrix multiplication

Remark. For $\mathbb{K} = \mathbb{C}$, the first line of code must be `C = complex(zeros(m,p))`.

Program 1 (Matrix Product: column-wise).

```
1 C = zeros(m,p)
2 for l=1:p
3   C[:,l] = A*B[:,l]
4 end
```

Program 2 (Matrix Product: row-wise).

```
1 C = zeros(m,p)
2 for j=1:m
3   C[j:j,:] = A[j:j,:]*B
4 end
```

Since Julia calls the BLAS routine `xGEMM` for matrix-matrix multiplication to execute the matrix-column-vector product in line 3, this program be accelerated through the explicit use of the `xGEMV` BLAS routine (notice that the order of factors requires the transposition flag `'T'` to be set):[88]

Program 2 (Matrix Product: row-wise, accelerated version).

```
1 C = zeros(m,p)
2 for j=1:m
3   C[j,:] = BLAS.gemv('T',B,A[j,:])
4 end
```

[88] Julia offers a very user friendly interface to BLAS and LAPACK where a routine named 'xyz' can be called by `BLAS.xyz` and `LAPACK.xyz`.

Program 3 (Matrix Product: inner products).

```
C = zeros(m,p)
for j=1:m
  for l=1:p
    C[j:j,l] = A[j:j,:]*B[:,l]
  end
end
```

This version can be considerably accelerated when Julia explicitly uses a BLAS-1 routine and does not create a copy of the row and column vectors:

Program 3 (Matrix Product: inner product, accelerated version).

```
C = zeros(m,p)
for j=1:m
  for l=1:p
    C[j,l] = dot(conj(view(A,j,:)),view(B,:,l))
  end
end
```

Remark. Notice that the `dot` command takes complex conjugates of the first factor.

Program 4 (Matrix Product: outer product).

```
C = zeros(m,p)
for k=1:n
  C += A[:,k]*B[k:k,:]
end
```

Since Julia calls a matrix-matrix multiplication for computing the outer product, the program can be accelerated with the suitable Level-2 BLAS routine `xGER`:

Program 4 (Matrix Product: outer products, accelerated version for real matrices).

```
C = zeros(m,p)
for k=1:n
  BLAS.ger!(1.0,A[:,k],B[k,:],C) # in situ call to xGER
end
```

Remark. Julia uses an '`!`' at the end of a function name when the routine runs *in situ* and thereby at least one argument is *overwritten*. In this way, `BLAS.ger!(alpha,x,y,A)` executes the rank-1 update $A + \alpha xy' \to A$ right in place where `A` is stored in memory.

Exercise. Modify this program so that it works correctly for complex matrices, too.

Program 5 (Matrix Product: componentwise).

```
C = zeros(m,p)
for j=1:m
  for l=1:p
    for k=1:n
      C[j,l] += A[j,k]*B[k,l]
    end
  end
end
```

Thanks to the extremely effective JIT-compiler (that creates native machine code directly before the first execution of the program), componentwise multiplication is much faster in Julia than in MATLAB, and is almost as fast as in C. The full performance of C can be achieved by using the compiler directive `@inbounds`, which tells Julia that the indices will stay in bounds so that it can safely pass on admissibility checks:

Program 5 (Matrix product: componentwise, accelerated version).

```
C = zeros(m,p)
for j=1:m
  for l=1:p
    for k=1:n
      @inbounds C[j,l] += A[j,k]*B[k,l]
    end
  end
end
```

The execution times show that Julia is especially faster than MATLAB when loops are involved, in which case the full performance of a compiled language can be exploited. Below, Table 3.4 has been amended:

program	BLAS level	MATLAB [s]	C & BLAS [s]	Julia [s]
`A*B`	3	0.031	0.029	0.030
column-wise	2	0.47	0.45	0.48
row-wise	2	0.52	0.49	1.2 | 0.52
outer product	2	3.5	0.75	6.53 | 0.75
inner product	1	1.7	1.5	8.5 | 1.8
componentwise	0	20	1.6	3.1 | 1.6

The minimal discrepancy between C and Julia can be accounted for by the use of different BLAS libraries: MKL from Intel is used in the C implementation (as well as by MATLAB) and OpenBLAS by Julia.

B.7 Execution times can either be measured with the `@time` macro,

```
@time instructions
```

or similar to MATLAB with

```
tic(); instructions; toc()
```

More accurate measurements can be taken with the package `BenchmarkTools`, which automatically calculates an average of multiple runs if the execution is fast.

The number of local processor threads used for a possibly parallel execution of BLAS routines can be easily controlled with the following command:

```
BLAS.set_num_threads(number of coras)
```

B.8 Forward substitution to solve $Lx = b$:

Program 6.

```
x = zeros(b) # zero vector of the same size and type (real/complex) as b
for k=1:m
  x[k:k] = (b[k:k] - L[k:k,1:k-1]*x[1:k-1])/L[k,k]
end
```

Remark. Notice the consistent "double" use of the index k in the form `k:k`. This is because in Julia the product of row and column vectors

```
L[k:k,1:k-1]*x[1:k-1]
```

returns a result of the type vector (in $\mathbb{K}^1$), whereas the components `b[k]`, `x[k]` are scalars. Julia does not provide a "type promotion" functionality that would allow the assignment of vectors in $\mathbb{K}^1$ or matrices in $\mathbb{K}^{1\times 1}$ to scalars $\mathbb{K}$. We must therefore write `b[k:k]` and `x[k:k]` in order to ensure the correct type is matched. Alternatively, the `dot` command can be used, but for $\mathbb{K} = \mathbb{C}$ the complex conjugate in the first factor must be taken into account:

```
x = zeros(b)
for k=1:m
  x[k] = (b[k] - dot(conj(L[k,1:k-1]),x[1:k-1]))/L[k,k]
end
```

For in situ execution, the first variant can be written as:

Program 7 (Forward Substitution for $x \leftarrow L^{-1}x$).

```
for k=1:m
  x[k:k] -= L[k:k,1:k-1]*x[1:k-1]
  x[k] /= L[k,k]
end
```

In Julia, the command to solve a triangular system of equations such as $Lx = b$ or $Ux = b$ can also be written succinctly (and exactly as in MATLAB, Julia thereby analyzes the structure of the matrix):

```
x = L\b, x = U\b
```

B.9 If we encode a permutation $\pi \in S_m$ in Julia by `p` $= [\pi(1), \ldots, \pi(m)]$, row and column permutations $P'_\pi A$ and AP_π can be expressed as:

```
A[p,:], A[:,p]
```

The permutation matrix P_π itself can be obtained as follows:

```
E = eye(m), P = E[:,p]
```

In Julia, the symbol `I` is reserved for a *universal* identity matrix (see the reconstruction of the L-factor in §B.10 and the program §B.23) and should therefore never be overwritten.

B.10 *In situ* triangular decomposition without pivoting can be written as:

Program 8 (Triangular Decomposition).

```
for k=1:m
  A[k+1:m,k] /= A[k,k]                          # (7.1b)
  A[k+1:m,k+1:m] -= A[k+1:m,k]*A[k:k,k+1:m]     # (7.1c)
end
```

The factors L and U are then reconstructed with the commands:

```
L = tril(A,-1) + I
U = triu(A)
```

Here, *copies* of the data from matrix A are created and stored as full matrices (with the other triangle full of zeros). It is more effective to use the direct *view* of the memory of A in the form of a data structure for (unipotent) triangular matrices that can then nevertheless be used like standard matrices for further calculations:

```
L = LinAlg.UnitLowerTriangular(A)
U = LinAlg.UpperTriangular(A)
```

When pivoting is added, the code can be written as follows:

Program 9 (Triangular Decomposition with Partial Pivoting).

```
p = collect(1:m)     # initialization of the permutation vector
for k=1:m
  j = indmax(abs(A[k:m,k])); j = k-1+j          # pivot search
  p[[k,j]] = p[[j,k]]; A[[k,j],:] = A[[j,k],:]  # row swap
  A[k+1:m,k] /= A[k,k]                          # (7.2d)
  A[k+1:m,k+1:m] -= A[k+1:m,k]*A[k:k,k+1:m]     # (7.2e)
end
```

The triangular factors of the matrix `A[p,:]` are exactly reconstructed as above. Peak performance can be reached by means of the Julia interface to `xGETRF`:

```
L, U, p = lu(A)
```

B.11 In Julia, the solution $X \in \mathbb{K}^{m\times n}$ of the system $AX = B \in \mathbb{K}^{m\times n}$ is therefore:

```
L, U, p = lu(A)
Z = L\B[p,:]
X = U\Z
```

or completely *equivalently*, if the decomposition $P'A = LU$ is no longer needed:

```
X = A\B
```

Alternatively, Julia offers the compact data structure `Base.LinAlg.LU` to store the factorization:

```
F = lufact(A) # lufact(A,Val{false}) prevents pivoting
L = F[:L]; U = F[:U]; p = F[:p]
```

This way a system of the form $AX = B$ can be solved without re-factoring of the matrix A brief and succinctly as follows:

```
X = F\B
```

B.12 In Julia, the program for the Cholesky decomposition can be written as:

Program 10 (Cholesky Decomposition of A).

```
L = zeros(A)
for k=1:m
  lk = L[1:k-1,1:k-1]\A[1:k-1,k]       # (8.1b)
  L[k,1:k-1] = lk'                     # (8.1a)
  L[k:k,k] = sqrt(A[k:k,k] - lk'*lk)   # (8.1c)
end
```

Exercise. Write an *in situ* version that converts the lower triangular matrix of A into the lower triangular matrix of L. *Hint:* Use `LinAlg.LowerTriangular`, cf. §B.10.

Peak performance can be reached with the interface to xPOTRF:

```
L = chol(A,Val{:L})
```

The factor $U = L'$ is calculated slightly faster if a column-wise storage scheme is used for matrices on the machine at hand:

```
U = chol(A)
```

If a linear system of the form $AX = B$ is to be solved, the following approach should be taken (this way the factorization can also be easily re-used):[89]

```
F = cholfact(A)
X = F\B               # solution of the system of equations
L, U = F[:L], F[:U] # L factor and, alternatively, U = L' factor
```

B.13 The MGS algorithm of QR decomposition can be written *in situ* as follows:

Program 11 (QR Decomposition with MGS Algorithm).

```
R = zeros(n,n) # for 𝕂 = ℂ: R = complex(zeros(n,n))
for k=1:n
  R[k,k] = norm(A[:,k])
  A[:,k] /= R[k,k]
  R[k:k,k+1:n] = A[:,k]'*A[:,k+1:n]
  A[:,k+1:n] -= A[:,k]*R[k:k,k+1:n]
end
```

The calculation of an (often *not* normalized) QR decomposition with the Householder method can be compactly completed with the following LAPACK interface:

```
Q, R = qr(A, thin=true)
```

[89] `X = A\B` actually *does not* use the Cholesky decomposition of A, see the remark in §B.19.

Unlike MATLAB, Julia allows access to (and convenient use of) the implicit representation[90] of the matrix Q as mentioned in §9.8:

```
2 F = qrfact(A) # faster by a factor of 2
3 Q = F[:Q]     # of the type Base.LinAlg.QRCompactWYQ
4 R = F[:R]     # read the factor R
```

The object oriented nature of Julia allows for matrix-vector multiplication `Q*x` etc. even when `Q` is of the type `Base.LinAlg.QRCompactWYQ`, the details are completely hidden from the user. The unitary factor Q can be retroactively represented as an *ordinary* matrix if needed:

```
5 Q = full(F[:Q], thin=true) # needs more cpu time than qrfact(A) itself
```

The sum of the execution times from lines 2 and 5 is exactly equal to that of the command from line 1.

B.14 Error Analysis 1 from §§14.1–14.2: *quadratic equation*

```
1 >> p = 400000.; q = 1.234567890123456;
2 >> r = sqrt(p^2+q); x0 = p - r
3 -1.5432015061378 48e-6
4 >> r^2 - p^2
5 1.23455810546875
6 >> x1 = p + r;
7 >> x0 = -q/x1
8 -1.5432098626513432e-6
```

B.15 Error Analysis 2 from §§14.3–14.4: *evaluation of* $\log(1+x)$

```
1 >> x = 1.234567890123456e-10;
2 >> w = 1+x; f = log(w)
3 1.2345680033069662e-10
4 >> w-1
5 1.234568003383174e-10
6 >> f = log(w)/(w-1)*x
7 1.234567890047248e-10
8 >> log1p(x)
9 1.234567890047248e-10
```

B.16 Error Analysis 3 from §§14.5–14.6: *sample variance*

```
1 >> x = [10000000.0; 10000000.1; 10000000.2]; m = length(x);
2 >> S2 = (sum(x.^2) - sum(x)^2/m)/(m-1)
3 -0.03125
4 >> xbar = mean(x);
5 >> S2 = sum((x-xbar).^2)/(m-1)
```

[90] More specifically, both Julia and LAPACK use the *compact WY* representation of Q, see R. Schreiber, C. F. Van Loan: *A storage-efficient WY representation for products of Householder transformations*. SIAM J. Sci. Stat. Comput. 10, 53–57, 1989.

```
0.009999999925494194
>> var(x)
0.009999999925494194
>> kappa = 2*dot(abs(x-xbar),abs(x))/S2/(m-1)
2.000000027450581e8
```

B.17 Julia offers an interface to LAPACK's condition number estimate `xGECON`:

```
cond(A,1)    # estimate of κ₁(A)
cond(A,Inf)  # estimate of κ∞(A)
```

Here, a previous triangular decomposition of the matrix A can be elegantly and profitably re-used:

```
F = lufact(A) # triangular decomposition of A
cond(F,1)     # estimate of κ₁(A)
cond(F,Inf)   # estimate of κ∞(A)
```

B.18 The Wilkinson example of an instability of triangular decomposition with partial pivoting as discussed in §15.10 can be written in Julia as:

```
>> m = 25;
>> A = 2I-tril(ones(m,m)); A[:,m] = 1; # Wilkinson matrix
>> b = randn(MersenneTwister(123),m);  # reproducible right hand side
>> F = lufact(A);     # triangular decomposition with partial pivoting
>> x = F\b;           # substitutions
>> r = b - A*x;       # residual
>> omega = norm(r,Inf)/(norm(A,Inf)*norm(x,Inf)) # backward error (15.1)
2.6960071380884023e-11
```

Let us compare once more with the result using the QR decomposition:

```
>> F_qr = qrfact(A);       # QR decomposition
>> x_qr = F_qr\b;          # substitutions
>> r_qr = b - A*x_qr;      # residual
>> omega_qr = norm(r_qr,Inf)/(norm(A,Inf)*norm(x_qr,Inf)) # back. error
7.7342267691589e1e-17
>> cond(F,Inf)             # estimate of the condition number κ∞(A)
25.0
>> norm(x-x_qr,Inf)/norm(x,Inf)
4.848325731432268e-10
```

This accuracy, however, is consistent with the forward error estimate (15.3):

$$\kappa_\infty(A)\omega(\hat{x}) \approx 25 \times 2.7 \cdot 10^{-11} \approx 6.8 \cdot 10^{-10}.$$

As described in §15.13, just one step of iterative refinement repairs the instability of the triangular decomposition with partial pivoting. Notice that the previously calculated triangular decomposition `F` can elegantly be re-used:

```
>> w = F\r;
>> x = x + w; r = b - A*x;
>> omega = norm(r,Inf)/(norm(A,Inf)*norm(x,Inf))
7.031115244689928e-18
>> norm(x-x_qr,Inf)/norm(x,Inf)
1.582000930055234e-15
```

As a comparison: the *unavoidable* error of the solution is $\kappa_\infty(A) \cdot \epsilon_{\text{mach}} \approx 2 \cdot 10^{-15}$.

B.19 The Läuchli example from §16.5 can be written in Julia as:

```
>> e = 1e-7; # ϵ = 10^-7
>> A = [1 1; e 0; 0 e]; b = [2; e; e]; # Läuchli example
>> F = cholfact(A'A); # Cholesky decomposition of the Gram matrix
>> x = F\(A'b);       # solution of normal equation
>> print.(" ",x);
 1.01123595505618 0.9887640449438204
```

Remark. In contrast, the direct solution of the normal equation using the \ command, without explicitly using Cholesky decomposition, accidentally[91] yields the "correct" solution:

```
>> x = (A'A)\(A'b);   # solution of the normal equation
>> print.(" ",x);
 1.0000000000000002 1.0
```

Hence, a comparison with §16.5 shows that the \ command for self-adjoint matrices behaves differently in Julia than in MATLAB:

- In the case of self-adjoint matrices, Julia *always* calls the LAPACK routine `xSYTRF`, which uses a symmetrical variant of triangular decomposition with pivoting, namely in the form of the *Bunch–Kaufman–Parlett decomposition*

  $$P'AP = LDL'.$$

  Here, the permutation matrix P represents diagonal pivoting, L is lower triangular, and D is a block-diagonal matrix comprised of 1×1 or 2×2 blocks. As it were, this method has the same complexity of $m^3/3$ flop as the Cholesky decomposition, but for s.p.d. matrices it is less amenable to optimized BLAS and therefore slower due to the overhead of pivoting.
- In contrast, in the case of self-adjoint matrices with positive diagonals, MATLAB *attempts* to see whether a Cholesky decomposition can be successfully calculated (thereby checking whether the matrix is s.p.d.); otherwise, as in Julia, `xSYTRF` is used.

Since the positive definiteness of a given self-adjoint matrix is most often known based on the structure of the underlying problem, and Cholesky decompositions are only used in a targeted manner, the approach taken by Julia is more efficient for the rest of the cases (this way avoiding wasted trial Cholesky decompositions).

[91] Notice that the information loss in $A \mapsto A'A$ is *irreparably* unstable. Nevertheless, we cannot exclude that a particular decomposition method accidentally gives a correct solution in a *specific* example where one would reliably rather require a different, stable algorithmic approach. In the Läuchli example, the choice of the smaller value `e = 1e-8` would yield a *numerically singular* matrix $A'A$ which causes problems no matter what decomposition of $A'A$ is used.

B.20 The solution of least squares problems discussed in §17.2 reads in Julia as:

Program 12 (Q-Free Solution of the Least Squares Problem).

```
R = qrfact([A b])[:R]       # R factor of the matrix A augmented by b
x = R[1:n,1:n]\R[1:n,n+1]   # solution of Rx = z
rho = abs(R[n+1,n+1])       # norm of the residual
```

or simply `x = A\b`. The numerical example from §16.5 is then written as follows:

```
>> x = A\b;
>> print.(" ",x);
 0.9999999999999996 1.0000000000000004
```

B.21 The example from §18.3 cannot be directly copied into Julia, since the developers of Julia rightly do not see the point in providing unstable and inefficient routines to calculate eigenvalues as roots of the characteristic polynomial. We can, however, convey the essence of this example with the help of the `Polynomials` package:

```
>> using Polynomials
>> chi = poly(1.0:22.0) # polynomial with the roots 1,2,...,22
>> lambda = roots(chi)  # roots of the monomial representation of χ
>> lambda[7:8]
2-element Array{Complex{Float64},1}:
 15.5374+1.15386im
 15.5374-1.15386im
```

The condition number (absolute with respect to the result and relative with respect to the data) can be computed as:

```
>> lambda = 15.0
>> chi_abs = Poly(abs.(coeffs(chi)))
>> kappa = polyval(chi_abs,lambda)/abs.(polyval(polyder(chi),lambda))
5.711809187852336e16
```

As in §18.3, we get $\kappa(15) \cdot \epsilon_{\text{mach}} \approx 6$.

B.22 The power iteration from §20.3 is in Julia:

Program 13 (Power Iteration).

```
normA = vecnorm(A)          # store ‖A‖_F
omega = Inf                 # initialize the backward error
w = A*v                     # initialize w = Av
while omega > tol           # backward error still too large?
   v = w/norm(w)            # new v
   w = A*v                  # new w
   mu = dot(v,w)            # Rayleigh quotient
   r = w - mu*v             # residual
   omega = norm(r)/normA    # backward error estimate ω̃
end
```

B.23 In Julia, the inverse iteration with shift from §20.7 can elegantly be written as follows (notice the clever re-use of the triangular decomposition `F` in line 5):

Program 14 (Inverse Iteration).

```
1  normA = vecnorm(A)            # store ‖A‖_F
2  omega = Inf                   # initialize the backward error
3  F = lufact(A - muShift*I)     # store the triangular decomposition of A − μI
4  while omega > tol             # backward error still too large?
5    w = F\v                     # solve the linear system of equations
6    normW = norm(w)             # ‖w‖_2
7    z = v/normW                 # auxiliary quantity z
8    v = w/normW                 # new v
9    rho = dot(v,z)              # auxiliary quantity ρ
10   mu = muShift + rho          # Rayleigh quotient
11   r = z - rho*v               # residual
12   omega = norm(r)/normA       # backward error estimate ω̃
13 end
```

B.24 In §20.11 we explained that usually just one step of inverse iteration suffices to compute an eigenvector corresponding to a given backward stable eigenvalue. In Julia, the illustrative example can be written as follows:

```
1  >> m = 1000; rng = MersenneTwister(123)            # initializaton
2  >> lambda = collect(1:m) + collect(-5:m-6)*1.0im   # given eigenvalues
3  >> Q,= qr(randn(rng,m,m)); A = Q*diagm(lambda)*Q'  # suitable normal matrix
4  >> mu = 1.0 - 5.0im                                # shift=known eigenval.
5  >> v = randn(rng,m); v = v/norm(v)                 # random start vector
6  >> w = (A - mu*I)\v                                # 1 step inv. iteration
7  >> omega = sqrt(m)*norm(A*w-mu*w)/vecnorm(A)/norm(w) # Eq. (20.1)
8  1.6091332673393636e-15
```

B.25 The program from §21.7 for a QR-algorithm based deflation is in Julia:

Program 15 (QR Algorithm).

```
1  function QR_algorithm(A)
2    m, = size(A); U = I                    # initialize
3    normA = vecnorm(A)                     # store ‖A‖_F
4    while norm(A[m,1:m-1])>eps()*normA     # backward error still too large?
5      mu = ...                             # shift μ_k (see §§21.10/21.12)
6      Q, R = qr(A - mu*I)                  # QR decomposition
7      A = R*Q + mu*I                       # RQ multiplication
8      U = U*Q                              # transformation U_k = Q_1 ··· Q_k
9      end
10   lambda = A[m,m]                        # eigenvalue
11   B = A[1:m-1,1:m-1]                     # deflated matrix
12   w = A[1:m-1,m]                         # last column in Schur decomposition
13   return lambda, U, B, w
14 end
```

In program line 5 the Rayleigh shift from §21.10 is simply `mu = A[m,m];` whereas the Wilkinson shift from §21.12 is coded as:

```
5 mu, = eig(A[m-1:m,m-1:m]); ind = indmin(abs(mu-A[m,m])); mu = mu[ind]
```

A Schur decomposition is finally calculated as in §21.9, though in Julia the variables `T` and `Q` should be initialized as *complex* matrices:

Program 16 (Schur Decomposition).

```
T = complex(zeros(m,m))              # initializations ...
Q = complex(eye(m))                  # ... as complex matrices
for k=m:-1:1                         # column-wise from back to front
  lambda, U, A, w = QR_algorithm(A) # QR-algorithm based deflation
  Q[:,1:k] = Q[:,1:k]*U              # transform first k columns of q
  t[1:k,k+1:m] = u'*t[1:k,k+1:m]     # transform first k remaining rows of T
  T[1:k,k] = [w; lambda]             # new kth column in T
end
```

B.26 We summarize the Julia commands for the algorithms from Chapter V.

- Hessenberg decomposition $H = Q'AQ$:

```
F = hessfact(A)
Q = F[:Q]
H = F[:H]
```

Here, `Q` has an *implicit representation* of type `Base.LinAlg.HessenbergQ` (cf. the first exercise in §21.16), the fully fledged explicit matrix can be obtained with

```
Q = full(Q)
```

- Schur decomposition $T = Q'AQ$:

```
T, Q, lambda = schur(complex(A))          # 1st version
F = schurfact(complex(A))                 # 2nd version
T = F[:T]; Q = F[:Z]; lambda = F[:values] # extraction of factors
```

The diagonal of T, i.e., the eigenvalues of A, can be found in the vector `lambda`.

- If one is only interested in calculating the eigenvalues of A, which is less costly than a Schur decomposition since then there is no need to deal with the unitary transformations, one simply calls the following command:

```
lambda = eigvals(A)
```

C Norms: Recap and Supplement

C.1 $\|\cdot\| : V \to [0,\infty)$ is a *norm* on the vector space V, if for $v, w \in V$, $\lambda \in \mathbb{K}$:

(1) $\|v\| = 0 \Leftrightarrow v = 0$ (definiteness)

(2) $\|\lambda v\| = |\lambda| \cdot \|v\|$ (absolute homogeneity)

(3) $\|v + w\| \leqslant \|v\| + \|w\|$ (subadditivity or triangle inequality)

We call norms on $\mathbb{K}^m$ *vector norms* and those on $\mathbb{K}^{m\times n}$ *matrix norms*.

C.2 Numerical analysis uses the following vector norms for $x = (\xi_j)_{j=1:m} \in \mathbb{K}^m$:

- *"taxi cab" norm* $\|x\|_1 = \sum_{j=1}^{m} |\xi_j|$;
- *Euclidean norm* (cf. §2.9) $\|x\|_2 = \left(\sum_{j=1}^{m} |\xi_j|^2\right)^{1/2} = \sqrt{x'x}$;
- *maximum norm* $\|x\|_\infty = \max_{j=1:m} |\xi_j|$.

Exercise. Draw the "unit sphere" $\{x : \|x\| \leqslant 1\}$ for these three norms in $\mathbb{R}^2$.

C.3 *Hölder's inequality* and the *Cauchy–Schwarz inequality* are valid, i.e.,

$$|x'y| \leqslant \|x\|_1 \cdot \|y\|_\infty, \qquad |x'y| \leqslant \|x\|_2 \cdot \|y\|_2 \qquad (x, y \in \mathbb{K}^m).$$

Besides, the $\|\cdot\|_2$-norm is *invariant* under a column orthonormal $Q \in \mathbb{K}^{m\times n}$:

$$\|Qx\|_2^2 = (Qx)'(Qx) = x' \underbrace{Q'Q}_{=I} x = x'x = \|x\|_2^2 \qquad (x \in \mathbb{K}^n).$$

C.4 Given a matrix $A \in \mathbb{K}^{m\times n}$ written in components, columns and rows as

$$A = \begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1n} \\ \vdots & & \vdots \\ \alpha_{m1} & \cdots & \alpha_{mn} \end{pmatrix} = \begin{pmatrix} | & & | \\ a^1 & \cdots & a^n \\ | & & | \end{pmatrix} = \begin{pmatrix} — & a_1' & — \\ & \vdots & \\ — & a_m' & — \end{pmatrix}, \tag{C.1}$$

we define the *Frobenius norm* (also known as the *Schur* of *Hilbert–Schmidt norm*)

$$\|A\|_F = \left(\sum_{j=1}^{m}\sum_{k=1}^{n} |\alpha_{jk}|^2\right)^{1/2} = \left(\sum_{j=1}^{m} \|a_j\|_2^2\right)^{1/2} = \left(\sum_{k=1}^{n} \|a^k\|_2^2\right)^{1/2}.$$

Thanks to $\|a_j\|_2^2 = a_j'a_j$ and $\|a^k\|_2^2 = (a^k)'a^k$, the last two expressions can also be written in terms of the trace of the Gramian matrices AA' and $A'A$:

$$\|A\|_F^2 = \operatorname{tr}(AA') = \operatorname{tr}(A'A).$$

Since the trace of a matrix is defined as both the sum of the diagonal elements as well as the sum of the eigenvalues, it holds[92]

$$\|A\|_F^2 = \sum_{j=1}^{m} \lambda_j(AA') = \sum_{k=1}^{n} \lambda_k(A'A).$$

C.5 The Frobenius norm has the following properties:

- $\|I\|_F = \sqrt{\operatorname{tr}(I)} = \sqrt{m}$ for the identity matrix $I \in \mathbb{K}^{m\times m}$;
- *submultiplicativity* (follows from (2.3b) and the Cauchy–Schwarz inequality)

$$\|A \cdot B\|_F \leqslant \|A\|_F \cdot \|B\|_F \qquad (A \in \mathbb{K}^{m\times n}, B \in \mathbb{K}^{n\times p});$$

- *invariance* under column orthonormal Q as well as adjunction

$$\|QA\|_F = \Big(\operatorname{tr}(A' \underbrace{Q'Q}_{=I} A)\Big)^{1/2} = \|A\|_F, \qquad \|A'\|_F = \|A\|_F.$$

C.6 A vector norm induces a matrix norm for $A \in \mathbb{K}^{m\times n}$ in accordance with

$$\|A\| = \max_{\|x\|\leqslant 1} \|Ax\| \overset{(a)}{=} \max_{\|x\|=1} \|Ax\| \overset{(b)}{=} \max_{x\neq 0} \frac{\|Ax\|}{\|x\|}.$$

Remark. Here, (a) and (b) follow from the homogeneity of the vector norm; the maximum is attained since $\{x \in \mathbb{K}^n : \|x\| \leqslant 1\}$ is compact and $x \mapsto \|Ax\|$ is continuous.

For the identity matrix $I \in \mathbb{K}^{m\times m}$, $\|I\| = 1$ is always valid; therefore, the Frobenius norm is *not* an induced matrix norm for $m, n > 1$.

C.7 From the definition, it directly follows for induced matrix norms that

$$\|Ax\| \leqslant \|A\| \cdot \|x\| \qquad (x \in \mathbb{K}^n)$$

and therefore the *submultiplicativity*

$$\|AB\| \leqslant \|A\| \cdot \|B\| \qquad (A \in \mathbb{K}^{m\times n}, B \in \mathbb{K}^{n\times p}).$$

Remark. A vector norm on $\mathbb{K}^n$ always matches its induced matrix norm on $\mathbb{K}^{n\times 1}$; we can therefore continue to *identify* vectors as single column matrices and vice versa. The first inequality is then just the case $p = 1$ of general submultiplicativity.

[92] $\lambda_1(M), \ldots, \lambda_p(M)$ denote the eigenvalues of the matrix $M \in \mathbb{K}^{p\times p}$ in order of algebraic multiplicity.

C.8 Numerical analysis uses the following induced matrix norms for $A \in \mathbb{K}^{m\times n}$:

- *max-column-sum norm*

$$\|A\|_1 = \max_{\|x\|_1\leqslant 1} \|Ax\|_1 \overset{(a)}{=} \max_{k=1:n} \|a^k\|_1;$$

- *max-row-sum norm*

$$\|A\|_\infty = \max_{\|x\|_\infty\leqslant 1} \|Ax\|_\infty \overset{(b)}{=} \max_{j=1:m} \|a_j\|_1;$$

- *spectral norm*

$$\|A\|_2 = \max_{\|x\|_2\leqslant 1} \|Ax\|_2 \overset{(c)}{=} \left(\max_{j=1:m} \lambda_j(AA')\right)^{1/2} \overset{(d)}{=} \left(\max_{k=1:n} \lambda_k(A'A)\right)^{1/2}.$$

  The invariance of the Euclidean norm under column orthonormal Q directly induces[93] the invariance

$$\|QA\|_2 = \|A\|_2, \quad \text{therefore notably } \|Q\|_2 = \|QI\|_2 = \|I\|_2 = 1; \tag{C.2}$$

  both render column orthonormal matrices so important numerically.

Concerning adjunction, it directly follows from (a)–(d) that

$$\|A'\|_1 = \|A\|_\infty, \qquad \|A'\|_\infty = \|A\|_1, \qquad \|A'\|_2 = \|A\|_2. \tag{C.3}$$

Remark. The *calculation* of the matrix norms $\|\cdot\|_1$, $\|\cdot\|_\infty$ and $\|\cdot\|_F$ is fairly simple, but that of the spectral norm $\|\cdot\|_2$ is computationally far more costly by comparison: an eigenvalue problem would have to be solved. Since the same set of invariances is satisfied, the Frobenius norm often serves as a "poor man's" replacement for the spectral norm.

Exercise. Show that $\|xy'\|_2 = \|x\|_2\|y\|_2$ (as used in §§11.9, 15.2 and 21.7.)

C.9 Real analysis has taught us that all norms are *equivalent* for a *finite dimensional* vector space V: for $\|\cdot\|_p$ and $\|\cdot\|_q$, there exists a constant $c_{p,q} > 0$ such that

$$\|v\|_p \leqslant c_{p,q}\|v\|_q \qquad (v \in V).$$

Such a $c_{p,q}$ does however depend on V; therefore on the dimensions for vector and matrix norms. For the matrix norms from §§C.4 and C.8 (and accordingly with $n = 1$ for the vector norms from §C.2) we display the best possible values of the constant $c_{p,q}$ in the following table, where $r = \min(m, n)$:

$p \backslash q$	1	2	∞	F
1	1	$\sqrt{m}$	m	$\sqrt{m}$
2	$\sqrt{n}$	1	$\sqrt{m}$	1
∞	n	$\sqrt{n}$	1	$\sqrt{n}$
F	$\sqrt{n}$	$\sqrt{r}$	$\sqrt{m}$	1

[93] Alternatively, one once again uses $(QA)'(QA) = A'Q'QA = A'A$.

Exercise. Show that the table is valid. *Hint:* Consider $n = 1$ first and make use of A'.

C.10 The matrix $|A| \in \mathbb{K}^{m\times n}$ represents the *componentwise* application of the absolute value function on $A \in \mathbb{K}^{m\times n}$; we read inequalities written as

$$|A| \leqslant |B| \qquad (A, B \in \mathbb{K}^{m\times n})$$

componentwise (as introduced in §§7.9/7.11). Matrix norms that satisfy

$$\| \, |A| \, \| = \|A\| \qquad \text{and} \qquad |A| \leqslant |B| \;\Rightarrow\; \|A\| \leqslant \|B\|$$

are called *absolute* and *monotone* respectively.

Example. The norms $\|\cdot\|_1$, $\|\cdot\|_\infty$, $\|\cdot\|_F$ are both absolute and monotone. The same can be said of the spectral norm $\|\cdot\|_2$ in the case of vectors only ($m = 1$ or $n = 1$).

Remark. For finite dimensional norms, it generally holds: absolute $\Leftrightarrow$ monotone.

C.11 It holds for diagonal matrices that

$$\| \operatorname{diag}(x) \|_p \overset{\text{(a)}}{=} \|x\|_\infty \quad (p \in \{1, 2, \infty\}), \qquad \| \operatorname{diag}(x) \|_F = \|x\|_2. \tag{C.4}$$

Remark. For *induced* matrix norms, (a) is equivalent to the monotonicity (absoluteness) of the inducing vector norm.

Exercise. Prove the equivalences stated in the remarks in §§C.10 and C.11.

Exercise. Show that for normal matrices A (cf. 18.6) there holds $\|A\|_2 = \rho(A)$. Here,

$$\rho(A) = \max |\sigma(A)|$$

denotes the *spectral radius* of A.

Exercise. Let $P \in \mathbb{K}^{m\times m}$ be a projection[94] of rank r. Show that

- $\|P\|_2 \geqslant 1$;
- $\|P\|_2 = 1$ if and only if P is an orthogonal projection;
- if $0 < r < m$, it holds $\|P\|_2 = \|I - P\|_2$.

Hint: Construct a unitary U, such that

$$U'PU = \begin{pmatrix} I & S \\ 0 & 0 \end{pmatrix}, \qquad S \in \mathbb{K}^{r\times(m-r)},$$

and deduce that $\|P\|_2 = \sqrt{1 + \|S\|_2^2}$.

[94] A general projection P is defined by $P^2 = P$, and is additionally orthogonal when $P' = P$.

D The Householder Method for QR Decomposition

D.1 We want to calculate the full QR decomposition of a matrix $A \in \mathbb{R}^{m\times n}$ with full column rank in a *column-wise* manner. With a unitary Q_1, the first step is

$$Q_1' \underbrace{\begin{pmatrix} | & & | \\ a^1 & \cdots & a^n \\ | & & | \end{pmatrix}}_{=A} = \left(\begin{array}{c|c} \rho_1 & * \\ \hline & A_1 \end{array}\right)$$

and, thereafter, the matrix A_1 is used for the subsequent calculations.

Exercise. Thoroughly describe the algorithmic procedure used this way for the calculation of $Q'A = R$. Show how Q can be obtained as the product of the unitary matrices from the individual steps.

D.2 We are thus left with the problem of how to directly calculate a full QR decomposition of a column vector $0 \neq a \in \mathbb{R}^m$:

$$Q'a = \begin{pmatrix} \rho \\ \hline 0 \end{pmatrix} = \rho e_1.$$

D.3 To this end, we construct Q' as a *reflection* at a hyperplane, which we describe by a vector $v \neq 0$ perpendicular to the hyperplane. Such a reflection is then applied to the vector $x \in \mathbb{R}^m$ by flipping the sign of its components in the direction v:

$$\frac{v'x}{v'v} \mapsto -\frac{v'x}{v'v}.$$

Hence, it is nothing more than that the vector $2\frac{v'x}{v'v}v$ being subtracted from x,

$$Q'x = x - 2\frac{v'x}{v'v}v, \qquad Q' = I - \frac{2}{v'v}vv'.$$

Reflections of this form are called *Householder reflections* in numerical analysis.

Exercise. Show: $Q^2 = I$, $Q' = Q$, $\det Q = -1$, Q is unitary.

D.4 We must now find a vector v such that $Q'a = \rho e_1$, i.e.,

$$a - 2\frac{v'a}{v'v}v = \rho e_1.$$

Since a reflection (as with every unitary matrix) is isometric, we get in particular

$$|\rho| = \|a\|_2, \qquad v \in \operatorname{span}\{a - \rho e_1\}.$$

Since the length of v cancels itself out in Q', we can finally set

$$\rho = \pm\|a\|_2, \qquad v = a - \rho e_1.$$

We then choose the sign such that v is calculated, in any case, in a *cancellation free* manner (writing $a = (\alpha_j)_{j=1:m}$ and $v = (\omega_j)_{j=1:m}$):

$$\rho = -\operatorname{sign}(\alpha_1)\|a\|_2, \quad \omega_1 = \alpha_1 - \rho = \operatorname{sign}(\alpha_1)\left(|\alpha_1| + |\rho|\right), \quad \omega_j = \alpha_j \ (j > 1).$$

Hence, it holds

$$v'v = (a - \rho e_1)'(a - \rho e_1) = \|a\|_2^2 - 2\rho\alpha_1 + \rho^2 = 2\rho^2 - 2\rho\alpha_1 = 2|\rho|(|\rho| + |\alpha_1|)$$

and therefore in short (we know that $Q' = Q$)

$$Q = I - ww', \qquad w = v/\sqrt{|\rho|(|\rho| + |\alpha_1|)}, \qquad \|w\|_2 = \sqrt{2}.$$

Exercise. Generalize this construction to the complex case $a \in \mathbb{C}^m$.

D.5 The calculation of the decomposition $Q'a = \rho e_1$ therefore has computational cost $\#\,\text{flop} \doteq 3m$, as long as only w and ρ are calculated and Q is *not* explicitly stored as a matrix. The application of Q to a vector x, i.e.,

$$Qx = x - (w'x)w,$$

accordingly has computational cost $\#\,\text{flop} \doteq 4m$.

Exercise. Show that the calculation of a full QR decomposition of a matrix $A \in \mathbb{R}^{m\times n}$ ($m \geqslant n$) using Householder reflections has computational cost (again as long as Q *is not* explicitly constructed, but rather only the vectors w of the Householder factors are stored)

$$\#\,\text{flop} \doteq 4\sum_{k=0}^{n-1}(m-k)(n-k) \doteq 2mn^2 - 2n^3/3.$$

The evaluation of Qx or $Q'y$ each have cost $\#\,\text{flop} \doteq 4\sum_{k=0}^{n-1}(m-k) \doteq 4mn - 2n^2$.

E For the Curious, the Connoisseur, and the Capable

Model Backwards Analysis of Iterative Refinement

E.1 Skeel's Theorem 15.13 can be nicely explained using a *simplified* model:[95]

Only the factorization step $P'A = LU$ is calculated with error, so that[96]

$$P'(A+E) = \widehat{L}\cdot\widehat{U}, \qquad \|E\| = O(\gamma(A)\epsilon_{\text{mach}})\|A\|.$$

Hence, subsequent numerical solutions $\hat{u}$ of $Au = v$ fulfill $(A+E)\hat{u} = v$.

In this model, one step of iterative refinement applied to $Ax = b$ yields a numerical solution x_1 satisfying

$$x_0 = \hat{x}, \quad r_0 = b - Ax_0 = Ex_0, \qquad (A+E)\hat{w} = r_0, \qquad x_1 = x_0 + \hat{w};$$

the residual thereof is $r_1 = b - Ax_1 = r_0 - A\hat{w} = E\hat{w}$. We calculate

$$A\hat{w} = E(x_0 - \hat{w}) = E(x_1 - 2\hat{w}), \qquad E\hat{w} = EA^{-1}Ex_1 - 2EA^{-1}E\hat{w};$$

using norms results in the residual bound

$$\begin{aligned}\|r_1\| = \|E\hat{w}\| &\leqslant \|E\|^2\|A^{-1}\|\,\|x_1\| + 2\|E\|\,\|A^{-1}\|\,\|E\hat{w}\| \\ &= O(\gamma(A)^2\kappa(A)\|A\|\,\|x_1\|\epsilon_{\text{mach}}^2) + O(\gamma(A)\kappa(A)\epsilon_{\text{mach}})\|r_1\|.\end{aligned}$$

Now, if $\gamma(A)^2\kappa(A)\epsilon_{\text{mach}} = O(1)$ with $\gamma(A) \gg 1$, we get $\gamma(A)\kappa(A)\epsilon_{\text{mach}} \ll 1$. Thus, x_1 has an error in the model that already ensures *backward stability*:

$$\omega(x_1) = \frac{\|r_1\|}{\|A\|\,\|x_1\|} = O(\gamma(A)^2\kappa(A)\epsilon_{\text{mach}}^2) = O(\epsilon_{\text{mach}}).$$

Remark. For well conditioned matrices with $\kappa(A) = O(1)$, a *single* step of iterative refinement can thus compensate a growth factor of as large as

$$\gamma(A) = O(\epsilon_{\text{mach}}^{-1/2}),$$

which corresponds to an initial loss of accuracy of up to half the mantissa length (cf. the discussion in §15.12 and the numerical example in §§15.10/15.13).

Exercise. Construct an example to show that for $\gamma(A) \gg \epsilon_{\text{mach}}^{-1}$ even multiple steps of iterative refinement do not necessarily lead to an improvement in backward error.

---

[95] For a componentwise study of this model, see F. Bornemann: *A model for understanding numerical stability*, IMA J. Numer. Anal. 27, 219–231, 2007.

[96] Cf. §§15.8 and 15.9.

Global Convergence of the QR Algorithm without Shifts

E.2 Even though the convergence of the QR algorithm *without* shifts and deflation is much too slow to be of any practical importance, J. Wilkinson's classic and elegant convergence proof[97] is still very interesting for mathematical theory:

1. It explains how eigenvalues arrange themselves on the diagonal of the triangular factor of the Schur decomposition obtained in the limit.
2. It relates the QR algorithm to the power iteration: the unitary factor U_k of the QR decomposition of A^k becomes asymptotically the similarity transformation of A to triangular form.
3. It shows that in course of the QR algorithm with (convergent) shift, not only does the last row of A_k converge but the others are already "prepared".

E.3 The proof is based on a triangular decomposition of largely theoretical interest, which goes by applying a particular column pivoting strategy[98] to $A \in \mathrm{GL}(m;\mathbb{K})$: we will use the *first* zero element as pivot and will not merely transpose it into position, but rather *cyclically* permute all intermediate rows down one row. In the notation[99] from §7.9 it can easily be shown by induction (left as an exercise) that along with L_k its transformation

$$P_k L_k P_k'$$

is also a unipotent lower triangular. We therefore obtain a decomposition of the form $P_\pi' A = \tilde{L}R$, where $L = P_\pi \tilde{L} P_\pi'$ is also a unipotent lower triangular; it holds

$$A = LP_\pi R.$$

Such a decomposition is called a *(modified) Bruhat decomposition*.

Exercise. Show that the permutation π of a Bruhat decomposition $A = LP_\pi R$ is uniquely determined by the matrix $A \in \mathrm{GL}(m;\mathbb{K})$, but not the triangular factors L and R. These become unique by requiring that $\tilde{L} = P_\pi' L P_\pi$ is also lower triangular.

E.4 We assume that the eigenvalues of $A \in \mathrm{GL}(m;\mathbb{C})$ differ in absolute value (and are thus real if A is real), so that they can be ordered as follows:

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_m| > 0. \tag{E.1a}$$

[97] J. Wilkinson: *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1965, pp. 516–520; the presentation here follows roughly the argument from E. Tyrtyshnikov: *Matrix Bruhat decompositions with a remark on the QR (GR) algorithm*, Linear Algebra Appl. 250, 61–68, 1997.

[98] Wilkinson loc. cit. p. 519.

[99] In which here, of course, τ_k represents the cycle $(1\,2\,3\,\cdots\,r_k)$ instead of the transposition $(1\,r_k)$, when the pivot a_k is found at row number r_k of the matrix A_k.

In particular, there exists a corresponding basis of eigenvectors (which are real for real A); if we use this basis as columns of the matrix $X \in \mathbb{C}^{m\times m}$, it holds

$$X^{-1}AX = D, \qquad D = \operatorname{diag}(\lambda_1, \ldots, \lambda_m). \tag{E.1b}$$

Wilkinson's elegant idea consists of applying the Bruhat decomposition[100] to X^{-1},

$$X^{-1} = LP_\pi R, \tag{E.1c}$$

where L is unipotent lower triangular and R upper triangular. With triangular factors at hand, the specific order of the eigenvalues can be conveniently exploited in the following form: we generally have, for *any* lower triangular L_*, that

$$D^k L_* D^{-k} \to \operatorname{diag}(L_*) \qquad (k \to \infty). \tag{*}$$

This is because with $\theta = \max_{j>k} |\lambda_j/\lambda_k| < 1$ the elements l^*_{pq} of L_* satisfy

$$\left(\frac{\lambda_p}{\lambda_q}\right)^k l^*_{pq} = \begin{cases} O(\theta^k) \to 0 & \text{if } p > q, \\ l^*_{pp} & \text{if } p = q, \\ 0 & \text{otherwise.} \end{cases}$$

E.5 Starting with $A_0 = A$, the QR algorithm constructs the sequence

$$A_k = Q_k R_k, \qquad A_{k+1} = R_k Q_k \qquad (k = 0, 1, 2, \ldots).$$

It holds $A_k = U_k' A U_k$ and A^k has a QR decomposition of the form

$$A^k = U_k S_k, \qquad U_k = Q_0 \cdots Q_{k-1}, \qquad S_k = R_{k-1} \cdots R_0.$$

For $k = 0$, both of the claims are clear, and the induction steps from k to $k+1$ are

$$A_{k+1} = Q_k' A_k Q_k = Q_k' U_k' A U_k Q_k = U_{k+1}' A U_{k+1}$$

as well as

$$A^{k+1} = AA^k = AU_k S_k = U_k (U_k' A U_k) S_k = U_k A_k S_k = U_k (Q_k R_k) S_k = U_{k+1} S_{k+1}.$$

[100] In the "generic" case, there are no zero divisions to be avoided, which means that generally the eigenvalues will get sorted: $\pi = \mathrm{id}$. In the exceptional cases, however, the realm of rounding errors will lead in course of the iteration, by backward analysis, to increasingly perturbed A and, a fortiori, to increasingly perturbed X. The corresponding π therefore deviates less and less (in discrete jumps, now and then) from id and a numerical "self sorting" takes place in the end.

E.6 Using $P = P_\pi$ for short, we obtain from $A = XDX^{-1}$ and $X^{-1} = LPR$ that

$$U_k = A^k S_k^{-1} = XD^k(LPR)S_k^{-1}$$

and therefore from $A_k = U_k' A U_k$ (notice that $U_k' = U_k^{-1}$)

$$A_k = S_k R^{-1} P' L^{-1} \underbrace{D^{-k}(X^{-1}AX)D^k}_{=D} LPRS_k^{-1} = S_k R^{-1} P'(L^{-1}DL)PRS_k^{-1}.$$

In this way, we get $A_k = W_k^{-1} B_k W_k$ with $B_k = P'D^k(L^{-1}DL)D^{-k}P$ and the upper triangular matrices

$$W_k = (P'D^kP)RS_k^{-1} = P'(D^kL^{-1}D^{-k})D^kX^{-1}S_k^{-1} = P'(D^kL^{-1}D^{-k})X^{-1}U_k.$$

The last equality shows that the sequences W_k and W_k^{-1} are bounded: by (*), there holds $D^kL^{-1}D^{-k} \to \operatorname{diag}(L^{-1}) = I$; the U_k are unitary. Besides, it follows from (*)

$$B_k \to P' \operatorname{diag}(L^{-1}DL)P = P'DP = D_\pi, \qquad D_\pi = \operatorname{diag}(\lambda_{\pi(1)}, \dots, \lambda_{\pi(m)}).$$

In summary, we obtain the asymptotic result

$$A_k = \underbrace{W_k^{-1} D_\pi W_k}_{\text{upper triangular}} + \underbrace{W_k^{-1}(B_k - D_\pi)W_k}_{\to 0}$$

and therefore *Wilkinson's convergence theorem*: the assumptions (E.1) imply

$$\operatorname{diag}(A_k) \to D_\pi, \qquad \text{strict lower triangle of } A_k \to 0.$$

Remark. By compactness, a selection of subsequences yields $U_{k_\nu} \to Q$ within the space of unitary matrices and $W_{k_\nu} \to W$ within the space of non-singular upper triangular matrices. The limit of $A_{k_\nu} = U_{k_\nu}' A U_{k_\nu} = W_{k_\nu}^{-1} D_\pi W_{k_\nu} + o(1)$ then gives the *Schur decomposition*

$$Q'AQ = T, \qquad T = W^{-1}D_\pi W = \begin{pmatrix} \lambda_{\pi(1)} & * & \cdots & * \\ & \ddots & \ddots & \vdots \\ & & \ddots & * \\ & & & \lambda_{\pi(m)} \end{pmatrix}.$$

Exercise.

- By means of an example, show that the sequence A_k itself does generally *not* converge.
- Show that a selection of subsequences can be avoided if the sequence A_k is replaced by a sequence of the form $\Sigma_k' A_k \Sigma_k$ with suitable unitary diagonal matrices Σ_k.
- By constructing *specific* matrices A, show that every $\pi \in S_m$ can be attained (in theory only; see Footnote 100 for what to expect in actual numerical experiments).

Local Convergence of the QR Algorithm with Shifts

E.7 Our goal here is to understand, and to partly prove (under somewhat stronger conditions), the claims about convergence from §§21.10–21.12. To this end, we partition the matrices of the QR algorithm with shift as follows:

$$A_k - \mu_k I = \left(\begin{array}{c|c} B_k - \mu_k I & w_k \\ \hline r_k' & \lambda_k - \mu_k \end{array}\right) = \underbrace{\left(\begin{array}{c|c} P_k & u_k \\ \hline v_k' & \eta_k \end{array}\right)}_{=Q_k} \underbrace{\left(\begin{array}{c|c} S_k & s_k \\ \hline 0 & \rho_k \end{array}\right)}_{=R_k}$$

$$A_{k+1} - \mu_k I = \left(\begin{array}{c|c} B_{k+1} - \mu_k I & w_{k+1} \\ \hline r_{k+1}' & \lambda_{k+1} - \mu_k \end{array}\right) = \underbrace{\left(\begin{array}{c|c} S_k & s_k \\ \hline 0 & \rho_k \end{array}\right)}_{=R_k} \underbrace{\left(\begin{array}{c|c} P_k & u_k \\ \hline v_k' & \eta_k \end{array}\right)}_{=Q_k};$$

where, without loss of generality, the QR decomposition is chosen such that $\rho_k \geqslant 0$.

E.8 We aim at estimating $\|r_{k+1}\|_2$ in terms of $\|r_k\|_2$. Via multiplication, we obtain

$$r_k' = v_k' S_k, \qquad r_{k+1}' = \rho_k v_k',$$

and thus, with writing $\sigma_k = \|S_k^{-1}\|_2$ for short, the initial estimate

$$\|v_k\|_2 \leqslant \sigma_k \|r_k\|_2, \qquad \|r_{k+1}\|_2 \leqslant \rho_k \sigma_k \|r_k\|_2.$$

We must therefore estimate σ_k and ρ_k.

E.9 Since Q_k is unitary, it holds by the normalization of the last column and row

$$1 = \|u_k\|_2^2 + |\eta_k|^2 = \|v_k\|_2^2 + |\eta_k|^2,$$

therefore in particular

$$\|u_k\|_2 = \|v_k\|_2, \qquad |\eta_k| \leqslant 1.$$

It furthermore also follows from the unitarity that

$$\left(\begin{array}{c|c} P_k' & v_k \\ \hline u_k' & \eta_k' \end{array}\right) \left(\begin{array}{c|c} B_k - \mu_k I & w_k \\ \hline r_k' & \lambda_k - \mu_k \end{array}\right) = \left(\begin{array}{c|c} S_k & s_k \\ \hline 0 & \rho_k \end{array}\right)$$

and therefore

$$\rho_k = u_k' w_k + \eta_k'(\lambda_k - \mu_k), \qquad \rho_k \leqslant \|u_k\|_2 \|w_k\|_2 + |\lambda_k - \mu_k|.$$

For the *Rayleigh shift* $\mu_k = \lambda_k$, the estimates obtained so far imply

$$\rho_k \leqslant \sigma_k \|w_k\|_2 \|r_k\|_2, \qquad \|r_{k+1}\|_2 \leqslant \sigma_k^2 \|w_k\|_2 \|r_k\|_2^2.$$

E.10 As a sub-column of A_k the vector w_k remains bounded by

$$\|w_k\|_2 \leqslant \|A_k\|_2 = \|A\|_2.$$

If A and therefore also A_k are normal, it follows via the expansion of $A_k A_k' = A_k' A_k$ that the $(2,2)$ element of the partitioning satisfies

$$\|r_k\|_2^2 + |\lambda_k|^2 = \|w_k\|_2^2 + |\lambda_k|^2, \qquad \text{thus} \quad \|w_k\|_2 = \|r_k\|_2.$$

To summarize, we arrive at the following statement about the Rayleigh shift:

$$\|r_{k+1}\|_2 \leqslant \begin{cases} \sigma_k^2 \|r_k\|_2^3 & \text{if } A \text{ is normal,} \\ \sigma_k^2 \|A\|_2 \|r_k\|_2^2 & \text{otherwise.} \end{cases} \tag{E.2}$$

We are thus finished when we give reasons that σ_k actually remains bounded.

E.11 Due to $B_k - \mu_k I = P_k S_k$ and $\|P_k\|_2 \leqslant \|Q_k\|_2 = 1$ there is

$$\sigma_k = \|S_k^{-1}\|_2 \leqslant \|(B_k - \mu_k I)^{-1}\|_2 = \operatorname{sep}(\mu_k, B_k)^{-1}.$$

If the shift μ_k converges as $\|r_k\|_2 \to 0$ towards a *simple* eigenvalue λ of A, it must be the case that B_k converges towards a deflated matrix A_λ, where λ is no longer contained in the spectrum, so that

$$\sigma_k \leqslant \operatorname{sep}(\mu_k, B_k)^{-1} \to \operatorname{sep}(\lambda, A_\lambda)^{-1} < \infty.$$

Hence, if we assumed convergence as such, we would obtain from (E.2) the asserted quadratic or cubic *speed* of convergence.[101]

E.12 Without already assuming convergence, the boundedness of σ_k can be shown through perturbation theory. For the sake of simplicity, we will restrict ourselves to the case of a *self-adjoint* matrix A with *pairwise different* eigenvalues. Let the minimum mutual distance of two eigenvalues be $3\delta > 0$. Notice that along with A also the matrices A_k and B_k are self-adjoint and that furthermore $\sigma(A) = \sigma(A_k)$. Hence, upon writing

$$A_k = \left(\begin{array}{c|c} B_k & r_k \\ \hline r_k' & \lambda_k \end{array}\right) = \underbrace{\left(\begin{array}{c|c} B_k & \\ \hline & \mu_k \end{array}\right)}_{=F_k} + \underbrace{\left(\begin{array}{c|c} & r_k \\ \hline r_k' & \lambda_k - \mu_k \end{array}\right)}_{=E_k}$$

with the perturbation bound (using the Rayleigh-Shift)

$$\|E_k\|_2^2 \leqslant \|E_k\|_F^2 = 2\|r_k\|_2^2 + |\lambda_k - \mu_k|^2 = 2\|r_k\|_2^2,$$

[101] Up to here our argument has essentially followed the discussion in G. W. Stewart: *Afternotes goes to Graduate School*, SIAM, Philadelphia, 1997, pp. 139–141.

Corollary 19.3 implies under the *assumption* $\|r_k\|_2 \leqslant \delta/\sqrt{2}$ that for every $\lambda \in \sigma(A)$ there is exactly one $\mu \in \sigma(F_k)$ with $|\lambda - \mu| \leqslant \delta$. Therefore, the eigenvalues of F_k are pairwise different, too, and the distance between any two of them is at least δ (why?), so that in particular

$$\operatorname{dist}(\mu_k, \underbrace{\sigma(F_k) \setminus \{\mu_k\}}_{=\sigma(B_k)}) \geqslant \delta.$$

By Theorem 19.3, we finally obtain the bound

$$\sigma_k \leqslant \operatorname{sep}(\mu_k, B_k)^{-1} = \operatorname{dist}(\mu_k, \sigma(B_k))^{-1} \leqslant \delta^{-1}.$$

E.13 We can therefore state the local convergence result as follows:

Theorem. *Let A be self-adjoint with pairwise different eigenvalues that have a minimum mutual distance of $3\delta > 0$. If $\|r_0\|_2 \leqslant \delta/\sqrt{2}$, then as $k \to \infty$ the QR algorithm with Rayleigh shift yields cubic convergence of the form*

$$\|r_{k+1}\|_2 \leqslant \delta^{-2}\|r_k\|_2^3 \to 0.$$

Proof. It remains to be shown by induction that $\|r_k\|_2 \leqslant \delta/\sqrt{2}$. For $k = 0$, this is exactly the assumption on r_0; the induction step from k to $k+1$ is

$$\|r_{k+1}\|_2 \leqslant \sigma_k^2\|r_k\|_2^3 \leqslant \delta^{-2}\|r_k\|_2^3 \leqslant \tfrac{1}{2}\|r_k\|_2 \leqslant \|r_k\|_2 \leqslant \delta/\sqrt{2}.$$

Furthermore, the intermediate bound $\|r_{k+1}\|_2 \leqslant \|r_k\|_2/2$ inductively proves

$$\|r_k\|_2 \leqslant 2^{-k}\|r_0\|_2 \to 0 \qquad (k \to \infty),$$

which finally ensures convergence itself. □

E.14 The *Wilkinson shift* μ_k satisfies by §21.12 (with the notation used there)

$$(\lambda_k - \mu_k)(\alpha_k - \mu_k) = \beta_k\gamma_k, \qquad |\lambda_k - \mu_k| \leqslant |\alpha_k - \mu_k|.$$

Here, the last inequality holds since the solutions of the quadratic equation are also symmetric relative to the midpoint $(\lambda_k + \alpha_k)/2$ of α_k and λ_k. Hence, we get

$$|\lambda_k - \mu_k|^2 \leqslant |\beta_k\gamma_k| \leqslant \|w_k\|_2\|r_k\|_2$$

and, therefore, the bound

$$\|r_{k+1}\|_2 \leqslant \sigma_k^2\|w_k\|_2\|r_k\|_2^2 + \sigma_k\|w_k\|_2^{1/2}\|r_k\|_2^{3/2}.$$

Now, in the self-adjoint case, there holds $\|E_k\|_2 \leqslant \sqrt{3}\|r_k\|_2$ and thus $\sigma_k \leqslant \delta^{-1}$ if $\|r_k\|_2 \leqslant \delta/\sqrt{3}$. Altogether, completely similar to the proof for the Rayleigh shift, we obtain the following theorem:

Theorem. *Let A be self-adjoint with pairwise different eigenvalues that have a minimum mutual distance of $3\delta > 0$. If $\|r_0\|_2 \leqslant \delta/\sqrt{3}$, then as $k \to \infty$ the QR algorithm with Wilkinson shift yields quadratic convergence of the form*

$$\|r_{k+1}\|_2 \leqslant 2\delta^{-1}\|r_k\|_2^2 \to 0.$$

Remark. In the realm of the Wilkinson shift, the speed of convergence can be improved to that of the Rayleigh shift if there is[102]

$$\tau_k = \lambda_k - \alpha_k \to \tau \neq 0.$$

Actually, from

$$(\mu_k - \lambda_k)^2 + \tau_k(\mu_k - \lambda_k) = \beta_k\gamma_k, \qquad |\beta_k\gamma_k| \leqslant \|w_k\|_2\|r_k\|_2,$$

one then gets the substantially improved bound

$$|\mu_k - \lambda_k| = O(\tau^{-1}\|w_k\|_2\|r_k\|_2),$$

so that given a bounded σ_k there holds, just as in §E.9 for the Rayleigh-Shift, that

$$\rho_k = O(\|w_k\|_2\|r_k\|_2), \qquad \|r_{k+1}\|_2 = O(\|w_k\|_2\|r_k\|_2^2).$$

A Stochastic Upper Bound of the Spectral Norm

Disclaimer: this section is only suitable for more advanced readers.

E.15 Given $A \in \mathbb{C}^{m\times m}$ and $0 \neq x \in \mathbb{C}^m$ it holds by definition that

$$\frac{\|Ax\|_2}{\|x\|_2} \leqslant \|A\|_2.$$

In §20.11 we used the fact that for *random vectors* x a converse bound holds with high probability, too:

Theorem. *Let $A \in \mathbb{C}^{m\times m}$ be fixed and $x \in \mathbb{R}^m$ be a random vector with independent standard normal distributed components. It then holds with probability $\geqslant 1 - \delta$ that*

$$\|A\|_2 \leqslant \delta^{-1}\sqrt{m}\,\frac{\|Ax\|_2}{\|x\|_2} \qquad (0 < \delta \leqslant 1).$$

Proof. From the singular value decomposition of A (with U, V unitary), that is,

$$A = U\,\mathrm{diag}(\sigma_1, \ldots, \sigma_m)V', \qquad \|A\|_2 = \sigma_1 \geqslant \cdots \geqslant \sigma_m \geqslant 0,$$

[102] This holds, e.g., in the case that $\lambda_k \to \lambda \in \sigma(A)$ while the eigenvalues of $A - \lambda I$ differ in *absolute value*; cf. Theorem E.2.

as well as the unitary invariance of the spectral norm and of the multivariate normal distribution it follows with independent standard normal distributed random variables $\xi_1, \dots, \xi_m$

$$\mathbb{P}\left(\frac{\|Ax\|_2}{\|x\|_2} \leqslant \tau \|A\|_2\right) = \mathbb{P}\left(\frac{\sigma_1^2 \xi_1^2 + \dots + \sigma_m^2 \xi_m^2}{\xi_1^2 + \dots + \xi_m^2} \leqslant \tau^2 \sigma_1^2\right)$$

$$\leqslant \mathbb{P}\left(\frac{\xi_1^2}{\xi_1^2 + \dots + \xi_m^2} \leqslant \tau^2\right).$$

Now, for $m \geqslant 2$, the random variable $R^2 = \xi_1^2/(\xi_1^2 + \xi_2^2 + \dots + \xi_m^2)$ is of the form

$$X/(X+Y)$$

with independent chi-squared distributed random variables $X \sim \chi_1^2$ and $Y \sim \chi_{m-1}^2$; it is therefore beta distributed with parameters $(1/2, (m-1)/2)$ and, hence, has the distribution function $F(t) = \mathbb{P}(R^2 \leqslant t)$ with density[103]

$$F'(t) = \frac{1}{B\big(1/2, (m-1)/2\big)} t^{-1/2} (1-t)^{(m-3)/2} \qquad (0 \leqslant t \leqslant 1).$$

Thus, the probability distribution

$$G(\delta) = \mathbb{P}(R \leqslant \delta/\sqrt{m}) = F(\delta^2/m) \qquad (0 \leqslant \delta \leqslant \sqrt{m})$$

has, for $m \geqslant 2$, the density

$$G'(\delta) = \frac{2\delta}{m} F'(\delta^2/m) = \frac{2m^{-1/2}}{B\big(1/2, (m-1)/2\big)} \left(1 - \frac{\delta^2}{m}\right)^{(m-3)/2}.$$

From Stirling's formula follows the monotone limit

$$\frac{2m^{-1/2}}{B\big(1/2, (m-1)/2\big)} = 2m^{-1/2} \frac{\Gamma(m/2)}{\Gamma(1/2)\Gamma((m-1)/2)} \nearrow \sqrt{\frac{2}{\pi}}$$

and hence, for $m \geqslant 3$, the upper bounds[104]

$$G'(\delta) \leqslant \sqrt{\frac{2}{\pi}}, \qquad G(\delta) \leqslant \sqrt{\frac{2}{\pi}}\,\delta \leqslant \delta \qquad (0 \leqslant \delta \leqslant \sqrt{m}).$$

[103] All of this can be found in §9.2 of H.-O. Georgii: *Stochastics: Introduction to Probability and Statistics*, 3rd ed., Walter de Gruyter, Berlin, 2008.

[104] The first two bounds are asymptotically *sharp* for $\delta \geqslant 0$ small, since as $m \to \infty$

$$\left(1 - \frac{\delta^2}{m}\right)^{(m-3)/2} \to e^{-\delta^2/2}, \qquad G'(\delta) \to \sqrt{\frac{2}{\pi}} e^{-\delta^2/2} \qquad (0 \leqslant \delta < \infty).$$

For $m = 2$ it holds after integration

$$G(\delta) = \frac{2}{\pi} \arcsin(\delta/\sqrt{2}) \leqslant \delta \qquad (0 \leqslant \delta \leqslant \sqrt{2})$$

and for $m = 1$ the trivial approximation $G(\delta) = 0 \leqslant \delta$, where $0 \leqslant \delta < 1$. Therefore,

$$\mathbb{P}\left(\frac{\|Ax\|_2}{\|x\|_2} \leqslant \frac{\delta\|A\|_2}{\sqrt{m}}\right) \leqslant \mathbb{P}\left(R \leqslant \frac{\delta}{\sqrt{m}}\right) = G(\delta) \leqslant \delta \qquad (0 \leqslant \delta \leqslant 1),$$

which finally proves the assertion. □

Remark. As the proof shows, for $m \geqslant 3$ the bound of the theorem can be improved by yet a factor of

$$\sqrt{\frac{2}{\pi}} \approx 0.79788,$$

but this is then asymptotically *sharp* in the case $\sigma_1 = \cdots = \sigma_m = 0$ as $m \to \infty$ and $\delta \to 0$.

F More Exercises

> The student is advised to do practical exercises. Nothing but the repeated application of the methods will give him the whole grasp of the subject. For it is not sufficient to understand the underlying ideas, it is also necessary to acquire a certain facility in applying them. You might as well try to learn piano playing only by attending concerts as to learn the [numerical] methods only through lectures.
>
> *(Carl Runge 1909)*

There are 62 exercises already scattered throughout the book; here, 36 more will be added. Better yet, there are numerous other ones to be found in the "classics" listed on page viii which I would highly recommend to all the readers for supplementary training.

Computer Matrices

Exercise. Alternatively to §5.5, forward substitution can also be achieved with the following recursive partitioning of the lower triangular matrix $L_1 = L$:

$$L_k = \left(\begin{array}{c|c} \lambda_k & \\ \hline l_k & L_{k+1} \end{array}\right).$$

Based on this, write a MATLAB program for the *in situ* execution of forward substitution.

Exercise. Let $A \in \mathbb{K}^{m\times m}$ be given with coefficients α_{jk}. The goal is to calculate

$$\xi_j = \sum_{k=1}^{j} \alpha_{jk} \qquad (j = 1:m).$$

- Program the componentwise calculation using two `for`-loops.
- Program the calculation *without* `for`-loops but rather with matrix-vector operations.
- Compare the execution time for a real random matrix with $m = 10\,000$.
- What is the name of the corresponding custom-made BLAS routine?

Hint. Read `help tril; help triu; help ones;` and www.netlib.org/lapack/explore-html/.

Exercise. Let $u, v \in \mathbb{K}^m$ and $A = I + uv' \in \mathbb{K}^{m\times m}$.

- Under which condition (on u and v) is A invertible?
- Find a short formula for A^{-1} and $\det A$.

Hint. First, consider the specific case $v = e_1$ using a clever partitioning. Transform the general case to this specific case.

Matrix Factorization

Exercise. Given $A \in \mathrm{GL}(m;\mathbb{K})$ and $D \in \mathbb{K}^{n\times n}$, consider the block-partitioned matrix

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

- Find the *block-LU decomposition* of the form (I is the identity)

$$M = \begin{pmatrix} I & \\ C_* & I \end{pmatrix} \begin{pmatrix} A_* & \\ & D_* \end{pmatrix} \begin{pmatrix} I & B_* \\ & I \end{pmatrix}.$$

- Give a simple factorization of M^{-1} and characterize its existence.
- Write M^{-1} in the same way as M in form of a 2×2-block matrix.
- Compare the results with the textbook case $m = n = 1$.

Exercise. Consider the linear system of equations $Ax = b$ for the matrix

$$A = \left(\begin{array}{c|c} 1 & u' \\ \hline v & I \end{array}\right) \in \mathbb{K}^{m\times m}.$$

- Permute the rows and columns of A in such a way that the storage cost for the factors of an LU decomposition grows just *linearly* in m. Give an explicit form of L and U.
- Solve the system of equations with the help of the thus obtained LU decomposition. What is the condition for solvability?
- Write a MATLAB program *without* `for`-loops which finds the solution x given u, v and b. How many flop are required (in leading order)?

Exercise. Identify what the program

```
[m,n] = size(B);
for i=n:-1:1
  for j=m:-1:1
    B(j,i) = B(j,i)/A(j,j);
    for k=1:j-1
      B(k,i) = B(k,i) - B(j,i)*A(k,j);
    end
  end
end
```

does given the input $A \in \mathbb{K}^{m\times m}$, $B \in \mathbb{K}^{m\times n}$. Replace it with a short command *without* the use of `for`-loops.

Hint. First, replace two of the `for`-loops with matrix-vector operations and then study the inverse operation that reconstructs the input matrix (stored in B) given the output matrix B.

Exercise. Let $A \in \mathbb{K}^{m\times m}$ be invertible, $b \in \mathbb{K}^m$ and $n \in \mathbb{N}$. Describe an efficient algorithm for the solution of the linear system of equations

$$A^n x = b.$$

How many flop are required (in leading order) for $n \ll m$?

Exercise. Let a tridiagonal matrix

$$A = \begin{pmatrix} \delta_1 & \rho_2 & & & \\ \lambda_1 & \delta_2 & \rho_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \rho_m \\ & & & \lambda_{m-1} & \delta_m \end{pmatrix}$$

be given, which is *column-wise* strictly diagonally dominant:

$$|\delta_1| > |\lambda_1|\,, \qquad |\delta_j| > |\lambda_j| + |\rho_j| \quad (j = 2:m-1), \qquad |\delta_m| > |\rho_m|\,.$$

- Show that A has an LU decomposition with a unipotent lower bidiagonal factor L and a upper bidiagonal triangular factor U, such that U agrees with A above the diagonal. Why is there no need for pivoting?
- Formulate this decomposition as a MATLAB program, which receives the vectors

  $$d = (\delta_j)_{j=1:m} \in \mathbb{K}^m, \quad l = (\lambda_j)_{j=1:m-1} \in \mathbb{K}^{m-1}, \quad r = (\rho_j)_{j=2:m} \in \mathbb{K}^{m-1}$$

  as input. Pay attention to an efficient use of memory. How many flop are required (in leading order) by your program?

The method developed in this exercise is referred to in the literature as *Thomas algorithm*.

Exercise. The sweep operator $\mathcal{T}_k$ acts on a matrix $A \in \mathbb{K}^{m\times m}$ as follows:

$$Ax = y \quad \Rightarrow \quad \mathcal{T}_k A\tilde{x} = \tilde{y}$$

where $x = (\xi_1, \dots, \xi_{k-1}, \xi_k, \xi_{k+1}, \dots, \xi_m)'$, $y = (\eta_1, \dots, \eta_{k-1}, \eta_k, \eta_{k+1}, \dots, \eta_m)'$,
$\tilde{x} = (\xi_1, \dots, \xi_{k-1}, \eta_k, \xi_{k+1}, \dots, \xi_m)'$, $\tilde{y} = (\eta_1, \dots, \eta_{k-1}, -\xi_k, \eta_{k+1}, \dots, \eta_m)'$.

Find a clever formula for $\mathcal{T}_k A$. Under which condition on A is $\mathcal{T}_k$ well-defined? Show (while explicitly stating a simple condition that guarantees the operations to be well-defined):

- $\mathcal{T}_j$ and $\mathcal{T}_k$ commutate for $1 \leqslant j, k \leqslant m$.
- Sweep operators are of order 4, so that in particular ${\mathcal{T}_k}^{-1} = \mathcal{T}_k^3$.
- For a self-adjoint A, the matrix $\mathcal{T}_k A$ is also self-adjoint.
- If we partition A such that A_{11} is the $k \times k$ principal submatrix, it holds

  $$\mathcal{T}_1 \cdots \mathcal{T}_k \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} -A_{11}^{-1} & A_{11}^{-1}A_{12} \\ \hline A_{21}A_{11}^{-1} & A_{22} - A_{21}A_{11}^{-1}A_{12} \end{array}\right)$$

  The block $A_{22} - A_{21}A_{11}^{-1}A_{12}$ is called the *Schur complement* of A_{11} in A.
- It holds $\mathcal{T}_1 \cdots \mathcal{T}_m A = -A^{-1}$.

Though sweep operators are a popular "workhorse" in computational statistics,[105] they are potentially numerically *unstable* when used without pivoting (example?).

[105] See K. Lange: *Numerical Analysis for Statisticians*, 2nd ed., Springer-Verlag, New York, 2010.

Exercise. Let $x_1, \ldots, x_n \in \mathbb{R}^m$ be a basis of the space U. Write a MATLAB two-liner that calculates an orthonormal basis of the orthogonal complement of U.

Exercise. Given $A \in \mathbb{K}^{m\times n}$ where $n < m$, let a compactly stored *full* QR decomposition with `F = qrfact(A)` be *pre-calculated* in Julia. Only *afterwards* let a vector $a \in \mathbb{K}^m$ be given.

- Write, as a Julia function `QRUpdateR(F,a)`, an *efficient* algorithm to calculate *only* the R-factor of a reduced QR decomposition of the matrix

$$A_* = (A \,|\, a) \in \mathbb{K}^{m\times(n+1)}.$$

  Give reasons why the function performs as desired.

  Hint. The scaling or the addition/subtraction of vectors should both be *categorically* avoided.

- Estimate how many flop `QRUpdateR(F,a)` saves when compared to a re-calculation from scratch. Perform some benchmarking measurements.

  Hint. Recall that the matrix-vector products `F[:Q]*x` and `F[:Q]'*y` only require $O(mn)$ flop.

Error Analysis

Exercise. Consider for $0 < \epsilon \ll 1$ the linear system of equations $Ax = b$ where

$$A = \begin{pmatrix} 1 & 1 \\ 0 & \epsilon \end{pmatrix}, \qquad b = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

- Show that $\kappa_\infty(A) = 2 + 2\epsilon^{-1} \gg 1$. Find a perturbation $A + E$ with $\|E\|_\infty \leqslant \epsilon$, such that the relative error in x with respect to the $\|\cdot\|_\infty$-norm exceeds 100%.
- Show that the linear system of equations is *well* conditioned for relative error measures when *only* the input ϵ is perturbed.

Exercise. Calculate (with respect to componentwise relative errors) the condition number $\kappa(\det, A)$ of the map $A \in \mathbb{R}^{2\times 2} \mapsto \det A$. Characterize the *ill*-conditioned case.

Exercise. Let $\epsilon(\xi)$ be the *relative* distance from $0 < \xi \in \mathbb{F}_{\beta,t}$ to the *next nearest* machine number. Plot $\epsilon(\xi)$ in double logarithmic scale. Locate the machine precision ϵ_{mach}.

Exercise. MATLAB calculates:

```
>> format long e
>> exp(pi*sqrt(67)/6) - sqrt(5280)
ans =
      6.121204876308184e-08
```

How many decimal places are correct in this results (by "clever inspection" only, i.e., by mere counting and *without* the use of a computer or calculator)? What would one have to do in order to calculate all 16 decimal places correctly?

Exercise. Here we will practice to turn simple expressions into numerically stable ones:

- For which values of ξ are the following expressions unstable:

$$\sqrt{\xi+1}-\sqrt{\xi}, \quad \xi \in [0,\infty[; \qquad (1-\cos\xi)/\xi^2, \quad \xi \in (-\pi,\pi)\,?$$

- Find stable expressions. Compare for $\xi = 2^j$, $j = 48:53$, and $\xi = 10^{-j}$, $j = 4:9$.

Exercise. Here we will further practice to turn simple expressions numerically stable:

- Calculate the expression $(1+1/n)^n$ for $n = 10^k$, $k = 7:17$. What had you expected (from calculus)? Explain the observed „numerical limit “.
- Find a stable expression for $(1+1/n)^n$.

Exercise. For $x > 0$, the expression $\log\left(\sqrt{1+x^2}-1\right)$ defines a function $f(x)$.

- For which values of x is f ill-conditioned with respect to the relative error?
- For which values of x is the given expression a numerically unstable algorithm?
- Find a stable expression.

Hint. You may use the fact that $1 < xf'(x) < 2$ for $x > 0$.

Exercise. Let one be interested to calculate a root of the cubic polynomial

$$x^3 + 3qx - 2r = 0 \qquad (q, r > 0).$$

According to Cardano (1545) a real root is given by

$$x_0 = \sqrt[3]{r+\sqrt{q^3+r^2}} - \sqrt[3]{-r+\sqrt{q^3+r^2}}.$$

This formula is numerically unstable and, based on *two* cubic roots, also quite expensive.

- Explain why Cardano's formula is, for $r \ll q$ and for $r \gg q$, potentially unstable.
- Find a numerically stable formula for x_0 that allows the calculation with just *one* cubic and *one* square root.
- Show that the root x_0 is well-conditioned; it actually holds that $\kappa(x_0; q, r) \leqslant 2$.
- For the cases of $r \ll q$ and $r \gg q$, give numerical examples in which Cardano's formula looses at least half of the mantissa length in accuracy.

Exercise. MATLAB calculates:

```
>> rng(333);          % for reproducibility
>> A = randn(4000);
>> det(A)
ans =
   -Inf
```

Compute the determinant of the matrix A *without* under- or overflow. How many decimal places of the solution are then correct?

Exercise. Consider the complex square root $\sqrt{z}$ for $z = x + iy$ ($x, y \in \mathbb{R}$, $y \neq 0$).

- Write an efficient, numerically stable program that *only* uses arithmetic operations and *real* square roots. Be sure to avoid over- and underflow; the program should work, e.g., for the following input:

$$x = 1.23456789 \cdot 10^{200}, \qquad y = 9.87654321 \cdot 10^{-200}.$$

- Give simple estimates of the condition numbers of the following maps:

$$(x, y) \mapsto \mathrm{Re}(\sqrt{z}), \qquad (x, y) \mapsto \mathrm{Im}(\sqrt{z}).$$

Exercise. Let the system of linear equations

$$Hx = b, \qquad b = (1, \ldots, 1)' \in \mathbb{R}^{10},$$

be given with the *Hilbert matrix* $H = (h_{jk}) \in \mathbb{R}^{10\times 10}$, $h_{jk} = 1/(j + k - 1)$ for $j, k = 1 : 10$.

Hint. H can be generated with the MATLAB command `hilb`.

- Calculate a numerical solution $\hat{x}$ using LU decomposition with partial pivoting.
- Calculate the normalized relative backward error ω. Is $\hat{x}$ backward stable?
- Estimate the condition number $\kappa_\infty(H)$ with the help of the MATLAB command `condest` and compare the bound $\kappa_\infty(H) \cdot \omega$ of the relative forward error of $\hat{x}$ with its actual value

$$\frac{\|x - \hat{x}\|_\infty}{\|\hat{x}\|_\infty}.$$

  The *exact* solution $x \in \mathbb{Z}^{10}$ can be determined with the MATLAB command `invhilb`.

Exercise. The LAPACK manual warns of the following "professional blunder":

> *Do not attempt to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$.*

This exercise develops an explanation.

- Search online to find how many flop this "blunder" would require, when A^{-1} is computed with the LAPACK program `xGETRI` (which is behind the MATLAB command `inv`)? Compare the result with the standard method `x=A\b`.
- With the help of Criterion B from §13.8, characterize for which A the *backward stability* of the "blunder" is at risk.
- Construct a reproducible example (in MATLAB) where the "blunder" produces a *very large* backward error but the standard method results in one within the magnitude of machine precision. Estimate and assess the forward error, too.

Hint. Use normwise relative errors with respect to the $\|\cdot\|_\infty$ norm. An "exact" x would be taboo.

Exercise. A matrix $A \in \mathbb{K}^{m\times m}$ is called *row-wise diagonally dominant*, if

$$|a_{jj}| \geqslant \sum_{\substack{k=1\\k\neq j}}^{m} |a_{jk}| \qquad (j = 1:m).$$

It is called *column-wise* diagonally dominant, if A' is row-wise diagonally dominant.

- Show that if $A \in \mathrm{GL}(m;\mathbb{K})$ is column-wise diagonally dominant, triangular decomposition *with* partial pivoting does not actually perform *any* row swaps.
- Assess the suggestion to *turn off* partial pivoting for row-wise diagonally dominant matrices $A \in \mathrm{GL}(m;\mathbb{K})$.

  Hint. Compare the growth factors of A and A'.

Least Squares

Exercise. Consider the following three models (with random noise ϵ_j and parameters θ_k) for measurements $b = (\beta_1,\dots,\beta_m)' \in \mathbb{R}^m$ and $t = (\tau_1,\dots,\tau_m)' \in \mathbb{R}^m$:

$$\text{(i)} \quad \beta_j = \theta_1\tau_j + \theta_2 + \epsilon_j,$$

$$\text{(ii)} \quad \beta_j = \theta_1\tau_j^2 + \theta_2\tau_j + \theta_3 + \epsilon_j,$$

$$\text{(iii)} \quad \beta_j = \theta_1\sin(\tau_j) + \theta_2\cos(\tau_j) + \epsilon_j.$$

Find a least squares estimator x of $p = (\theta_1,\theta_2)'$ and $p = (\theta_1,\theta_2,\theta_3)'$ respectively.

- Write the estimator in terms of a least squares problem $\|Ax - b\|_2 = \text{min!}$.
- Write a MATLAB program which calculates, in a numerically stable fashion, the estimator x for each of the three models when passed the inputs b and t.

Exercise. The coefficients $(\alpha,\beta)' \in \mathbb{R}^2$ of a regression line $y = \alpha x + \beta$ fitting the measurements $(x_1,y_1),\dots,(x_m,y_m)$ are often given by the textbook formula

$$\alpha = \frac{\overline{xy} - \bar{x}\bar{y}}{\overline{xx} - \bar{x}\bar{x}}, \qquad \beta = \bar{y} - \alpha\bar{x},$$

where $\bar{x} = \frac{1}{m}\sum_{j=1}^{m} x_j$, $\overline{xy} = \frac{1}{m}\sum_{j=1}^{m} x_jy_j$, etc., denote the respective mean values.

- Show that the formula can be directly obtained when the normal equation of the least squares problem

  $$\|y - \alpha x - \beta\|_2 = \text{min!}$$

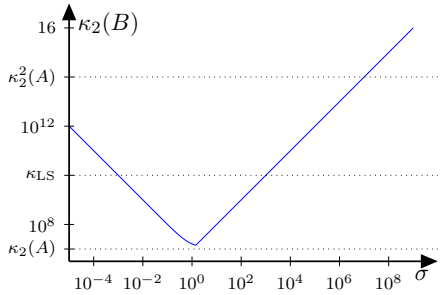
  is solved with *Cramer's rule*.
- Give reasons why the numerical stability of this formula is at risk.
- Construct a numerical example for which the formula is actually *unstable*.
- How are the coefficients α and β computed in a numerically stable manner instead?

Exercise. Let $A \in \mathbb{R}^{m\times n}$ with $m \gg n$ have full column rank.

- Show that $\|b - Ax\|_2 = \min!$ is *equivalent* to

$$\underbrace{\begin{pmatrix} I & A \\ A' & 0 \end{pmatrix}}_{=B} \begin{pmatrix} r \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}.$$

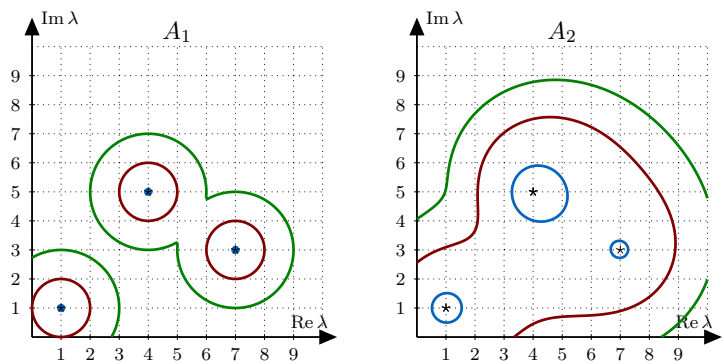
- Come up with a way to calculate the solution of this linear system (the sparsity of I and 0 can be ignored for the sake of simplicity). What is the computational cost?
- For a σ-dependent least squares problem, let $\kappa_2(A) = 10^7$ and $\kappa_{\mathrm{LS}} = 10^{10}$ be *independent* of σ, but let the condition number $\kappa_2(B)$ exhibit the following *dependence*:



For which values of σ would it make sense, for reasons of stability, to solve the least squares problem $\|b - Ax\|_2 = \min!$ via the system of equations with the matrix B?

Eigenvalue Problems

Exercise. The following figure shows the contour lines of the functions $F_k\colon \lambda \mapsto \mathrm{sep}(\lambda, A_k)$ for the values 0.1 (blue) 1.0 (red) and 2.0 (green); stars mark eigenvalues:



- Based on the images, specify which of the matrices *cannot* be normal.
- Estimate the *absolute* condition number of the eigenvalue $\lambda_* = 4 + 5i$ for A_1 and A_2.

Exercise. Given $A \in \mathbb{C}^{m\times m}$, show the *Gershgorin circle theorem*:

$$\sigma(A) \subset \bigcup_{j=1,\dots,n} K_j, \qquad \text{where} \qquad K_j = \Big\{ z \in \mathbb{C} : \ |z - \alpha_{jj}| \leqslant \sum_{k\neq j} |\alpha_{jk}| \Big\}.$$

Hint. Examine the row of $Ax = \lambda x$, for which the absolute value of the corresponding component of x is largest.

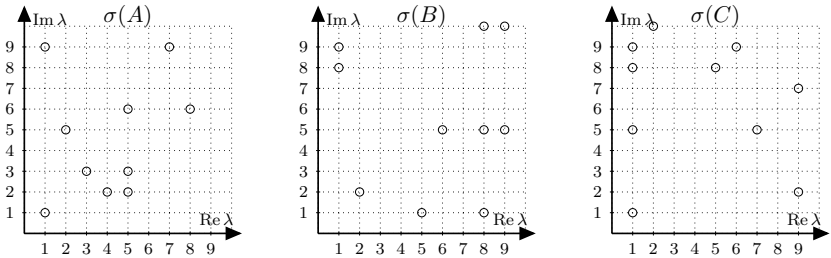
Exercise. Let $A \in \mathbb{C}^{m\times m}$ and (λ, x) be an eigenpair of A. The Rayleigh quotient

$$\rho(x, A) = \frac{x'Ax}{x'x}$$

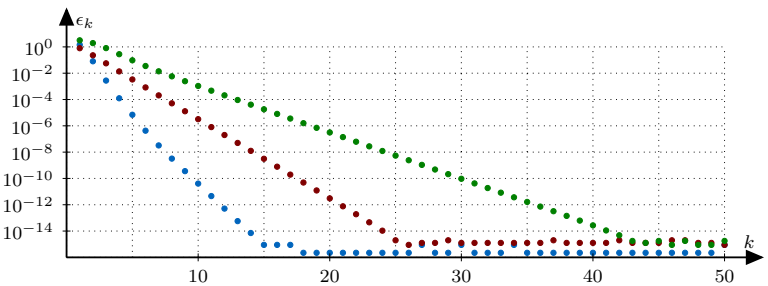
shows the following perturbation behavior ($\tilde{x} = x + h$, $h \to 0$):

$$\rho(\tilde{x}, A) = \rho(x, A) + O(\|h\|) \quad \text{and, for } \textit{normal} \text{ matrices,} \quad \rho(\tilde{x}, A) = \rho(x, A) + O(\|h\|^2).$$

Exercise. Let the normal matrices A, B and C have the following spectra:



For each matrix, inverse iteration with shift $\mu = 5 + 5i$ has been executed. The following figure shows the error $\epsilon_k = |\mu_k - \lambda|$ between the eigenvalue approximation μ_k in the jth step and the eigenvalue λ, towards which the iteration converges:



- Towards which eigenvalue λ does the iteration converge for A, B, and C?
- Match the convergence plots (green, red, blue) with the underlying matrix.
- Based on the spectra, calculate the convergence rates in the form $\epsilon_k = O(\rho^k)$.

Exercise. Given the eigenvalues of a normal matrix A and a shift μ, let there be

$$|\mu - \lambda_1| < |\mu - \lambda_2| \leqslant \cdots \leqslant |\mu - \lambda_m|, \qquad |\mu - \lambda_1| / |\mu - \lambda_2| = 0.125.$$

Estimate how many iteration steps k the inverse iteration will need to calculate an approximate eigenpair (μ_k, v_k) with a backward error of $O(\epsilon_{\text{mach}})$ in IEEE *single* precision. Which eigenpair is approximated?

Exercise. If one were to use the current eigenvalue approximation μ_{k-1} as a *dynamic* shift instead of a *fixed* shift $\mu_0 = \mu$ in course of the inverse iteration, extremely fast convergence would result for *real symmetric* matrices: in every step, the number of correct digits in the eigenvalue is *tripled* so that machine precision is already achieved after only 3–4 steps (cubic convergence, cf. §21.10). This method is called the *Rayleigh iteration*.

- Record the progress of μ_k in course of this iteration for the matrix

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 4 \end{pmatrix}$$

  when the initial shift value is given by $\mu_0 = 5$.
- How many steps would inverse iteration need for an example of dimension $m \gg 1$, in order to be actually more *expensive* than just 4 steps of the Rayleigh iteration?

Exercise. Let $\mu \in \mathbb{C}$. Write a MATLAB function that when passed a previously calculated Schur decomposition $Q'AQ = T$ subsequently calculates the following data:

- an eigenvalue $\lambda \in \sigma(A)$ closest to μ;
- a corresponding normalized eigenvector x of A;
- the backward error of the approximate eigenpair (λ, x).

Exercise. Let $A, B, C \in \mathbb{C}^{m \times m}$ be given. The Sylvester equation for $X \in \mathbb{C}^{m \times m}$, that is,

$$AX - XB = C \tag{*}$$

can be rendered by means of Schur decompositions $U'AU = R$ and $V'BV = S$ as

$$RY - YS = E \tag{**}$$

where $E = U'CV$ and $Y = U'XV$.

- Find an algorithm that computes the solution Y of (**).
  Hint. Write (**) as m equations for the columns of Y.
- Show that (*) has a unique solution if and only if $\sigma(A) \cap \sigma(B) = \emptyset$.
- Implement an algorithm in MATLAB that solves (*). How many flop are needed?
  Hint. Use the MATLAB command `schur`. It consumes $O(m^3)$ flop.

Exercise. Given the real matrix $A_0 \in \mathbb{R}^{m\times m}$, let the lower right 2×2 block in (21.3) have the complex conjugate eigenvalues $\lambda, \lambda' \notin \sigma(A_0)$.

- Show that after two steps of the QR iteration with the *Francis double shift*, that is,

$$A_0 - \lambda I = Q_0 R_0, \quad A_1 = R_0 Q_0 + \lambda I, \quad A_1 - \lambda' I = Q_1 R_1, \quad A_2 = R_1 Q_1 + \lambda' I,$$

  the matrix A_2 is also real if normalized QR decompositions were used: $A_2 \in \mathbb{R}^{m\times m}$.
- How can the passage $A_0 \mapsto A_2$ be realized with *real* operations only?

Notation

$\mathbb{K}$	field $\mathbb{R}$ or $\mathbb{C}$
$\alpha, \beta, \gamma, \ldots, \omega$	scalars
$a, b, c, \ldots, z$	vectors (column vectors)
$a', b', c', \ldots, z'$	co-vectors (row vectors)
$A, B, C, \ldots, Z$	matrices
i, j, l, m, n, p	indices/dimensions
$1:m$	colon notation for $1, \ldots, m$
$[\mathcal{A}]$	Iverson bracket: 1, if expression $\mathcal{A}$ is true, else 0
A'	adjoint of the matrix A
a^k, a'_j	kth column, jth row of the matrix A
e^k, e_k	kth identity vector
$\operatorname{diag}(x)$	the diagonal matrix made from the vector x
$\mathrm{GL}(m;\mathbb{K})$	general linear group of dimension m
P_π	permutation matrix
$\|E\|$, $[\![E]\!]$	norm and error measure of a matrix E
$\kappa(f;x)$	condition number of f in x
$\kappa(A)$	condition number of the matrix A
$\operatorname{cond}(A,x)$	Skeel–Bauer condition number of the system of equations $Ax = b$
$\mathbb{F}$	the set of floating point numbers
$\mathrm{fl}(\xi)$	representation of ξ as a floating point number
ϵ_{mach}	machine precision
$\doteq$	equality after rounding
$\doteq$, $\dot{\leqslant}$	equality and bound in leading order
$\gamma(A)$	growth factor of the matrix A
$\omega(\tilde{x})$	backward error of an approximate solution $\tilde{x}$
$\operatorname{sep}(\lambda, A)$	separation between λ and A

Index