

ETH Lecture 401-0663-00L Numerical Methods for CSE

# Numerical Methods for Computational Science and Engineering

Prof. R. Hiptmair, SAM, ETH Zurich

(with contributions from Prof. P. Arbenz and Dr. V. Gradinaru)

Autumn Term 2015  
(C) Seminar für Angewandte Mathematik, ETH Zürich

URL: <http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE/NumCSE15.pdf>



## Always under construction!

SVN revision 84575

The online version will always be work in progress and subject to change.

(Nevertheless, structure and main contents can be expected to be stable)



## Do not print!

Lecture recording:

<http://www.video.ethz.ch/Lectures/D-MATH/2015/Autumn/401-0663-00L>

Tablet notes:

[http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE/NCSE15\\_Notes/](http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE/NCSE15_Notes/)

# Contents

<b>0 Introduction</b>	<b>7</b>
0.0.1 Focus of this course . . . . .	7
0.0.2 Goals . . . . .	9
0.0.3 To avoid misunderstandings . . . . .	9
0.0.4 Reporting errors . . . . .	10
0.0.5 Literature . . . . .	10
0.1 Specific information . . . . .	11
0.1.1 Assistants and exercise classes . . . . .	11
0.1.2 Assignments . . . . .	12
0.1.3 Information on examinations . . . . .	12
0.2 Programming in C++11 . . . . .	13
0.2.1 Function Arguments and Overloading . . . . .	14
0.2.2 Templates . . . . .	15
0.2.3 Function Objects and Lambda Functions . . . . .	17
0.2.4 Multiple Return Values . . . . .	17
0.2.5 A Vector Class . . . . .	18
<b>1 Computing with Matrices and Vectors</b>	<b>32</b>
1.1 Fundamentals . . . . .	33
1.1.1 Notations . . . . .	33
1.1.2 Classes of matrices . . . . .	35
1.2 Software and Libraries . . . . .	37
1.2.1 MATLAB . . . . .	37
1.2.2 PYTHON . . . . .	40
1.2.3 EIGEN . . . . .	41
1.2.4 Matrix storage formats . . . . .	46
1.3 Basic linear algebra operations . . . . .	54
1.3.1 Elementary matrix-vector calculus . . . . .	54
1.3.2 BLAS – Basic Linear Algebra Subprograms . . . . .	61
1.4 Computational effort . . . . .	68
1.4.1 (Asymptotic) complexity . . . . .	69
1.4.2 Cost of basic operations . . . . .	70
1.4.3 Reducing complexity in numerical linear algebra: Some tricks . . . . .	71
1.5 Machine arithmetic . . . . .	75
1.5.1 Experiment: Loss of orthogonality . . . . .	75
1.5.2 Machine numbers . . . . .	79
1.5.3 Roundoff errors . . . . .	82
1.5.4 Cancellation . . . . .	86
1.5.5 Numerical stability . . . . .	99
1.6 Direct methods for linear systems . . . . .	104
1.6.1 Theory: Linear systems of equations . . . . .	107

1.6.1.1	Existence and uniqueness of solutions . . . . .	107
1.6.1.2	Sensitivity of linear systems . . . . .	107
1.6.2	Gaussian Elimination . . . . .	110
1.6.2.1	Basic algorithm . . . . .	110
1.6.2.2	LU-decomposition . . . . .	117
1.6.2.3	Pivoting . . . . .	124
1.6.3	Stability of Gaussian Elimination . . . . .	129
1.6.4	Elimination solvers for linear systems of equations . . . . .	136
1.6.5	Exploiting Structure when Solving Linear Systems . . . . .	141
1.7	Sparse Linear Systems . . . . .	146
1.7.1	Sparse matrix storage formats . . . . .	148
1.7.2	Sparse matrices in MATLAB . . . . .	150
1.7.3	Sparse matrices in EIGEN . . . . .	161
1.7.4	Direct Solution of Sparse Linear Systems of Equations . . . . .	162
1.7.5	LU-factorization of sparse matrices . . . . .	165
1.7.6	Banded matrices [15, Sect. 3.7] . . . . .	170
1.8	Stable Gaussian elimination without pivoting . . . . .	177
<b>2</b>	<b>Iterative Methods for Non-Linear Systems of Equations</b> . . . . .	<b>183</b>
2.1	Iterative methods . . . . .	185
2.1.1	Speed of convergence . . . . .	187
2.1.2	Termination criteria . . . . .	191
2.2	Fixed Point Iterations . . . . .	195
2.2.1	Consistent fixed point iterations . . . . .	195
2.2.2	Convergence of fixed point iterations . . . . .	197
2.3	Finding Zeros of Scalar Functions . . . . .	203
2.3.1	Bisection . . . . .	203
2.3.2	Model function methods . . . . .	205
2.3.2.1	Newton method in scalar case . . . . .	205
2.3.2.2	Special one-point methods . . . . .	208
2.3.2.3	Multi-point methods . . . . .	211
2.3.3	Asymptotic efficiency of iterative methods for zero finding . . . . .	216
2.4	Newton's Method . . . . .	218
2.4.1	The Newton iteration . . . . .	218
2.4.2	Convergence of Newton's method . . . . .	228
2.4.3	Termination of Newton iteration . . . . .	230
2.4.4	Damped Newton method . . . . .	232
2.4.5	Quasi-Newton Method . . . . .	236
2.5	Unconstrained Optimization . . . . .	242
2.5.1	Minima and minimizers: Some theory . . . . .	242
2.5.2	Newton's method . . . . .	242
2.5.3	Descent methods . . . . .	242
2.5.4	Quasi-Newton methods . . . . .	242
<b>3</b>	<b>Data Interpolation in 1D</b> . . . . .	<b>243</b>
3.1	Abstract interpolation . . . . .	244
3.2	Global Polynomial Interpolation . . . . .	249
3.2.1	Polynomials . . . . .	250
3.2.2	Polynomial Interpolation: Theory . . . . .	251
3.2.3	Polynomial Interpolation: Algorithms . . . . .	255
3.2.3.1	Multiple evaluations . . . . .	255
3.2.3.2	Single evaluation . . . . .	257

3.2.3.3	Extrapolation to zero . . . . .	260
3.2.3.4	Newton basis and divided differences . . . . .	262
3.2.4	Polynomial Interpolation: Sensitivity . . . . .	267
3.3	Shape preserving interpolation . . . . .	271
3.3.1	Shape properties of functions and data . . . . .	271
3.3.2	Piecewise linear interpolation . . . . .	273
3.4	Cubic Hermite Interpolation . . . . .	275
3.4.1	Definition and algorithms . . . . .	275
3.4.2	Local monotonicity preserving Hermite interpolation . . . . .	278
3.5	Splines . . . . .	281
3.5.1	Cubic spline interpolation . . . . .	282
3.5.2	Structural properties of cubic spline interpolants . . . . .	285
3.5.3	Shape Preserving Spline Interpolation . . . . .	289
<b>4</b>	<b>Approximation of Functions in 1D</b> . . . . .	<b>297</b>
4.1	Approximation by Global Polynomials . . . . .	299
4.1.1	Polynomial approximation: Theory . . . . .	300
4.1.2	Error estimates for polynomial interpolation . . . . .	304
4.1.3	Chebychev Interpolation . . . . .	313
4.1.3.1	Motivation and definition . . . . .	313
4.1.3.2	Chebychev interpolation error estimates . . . . .	317
4.1.3.3	Chebychev interpolation: computational aspects . . . . .	323
4.2	Mean Square Best Approximation . . . . .	327
4.2.1	Abstract theory . . . . .	327
4.2.1.1	Mean square norms . . . . .	327
4.2.1.2	Normal equations . . . . .	328
4.2.1.3	Orthonormal bases . . . . .	329
4.2.2	Polynomial mean square best approximation . . . . .	330
4.3	Uniform Best Approximation . . . . .	336
4.4	Trigonometric interpolation . . . . .	339
4.5	Approximation by piecewise polynomials . . . . .	348
4.5.1	Piecewise polynomial Lagrange interpolation . . . . .	349
4.5.2	Cubic Hermite interpolation: error estimates . . . . .	352
4.5.3	Cubic spline interpolation: error estimates [42, Ch. 47] . . . . .	356
4.6	Multi-dimensional Approximation on Tensor-Product Domains . . . . .	358
<b>5</b>	<b>Numerical Quadrature</b> . . . . .	<b>359</b>
5.1	Quadrature Formulas . . . . .	361
5.2	Polynomial Quadrature Formulas . . . . .	364
5.3	Gauss Quadrature . . . . .	367
5.4	Composite Quadrature . . . . .	380
5.5	Adaptive Quadrature . . . . .	387
<b>6</b>	<b>Least Squares</b> . . . . .	<b>395</b>
6.1	Normal Equations [15, Sect. 4.2], [42, Ch. 11] . . . . .	400
6.2	Orthogonal Transformation Methods [15, Sect. 4.4.2] . . . . .	403
6.3	Total Least Squares . . . . .	410
6.4	Constrained Least Squares . . . . .	411
6.4.1	Solution via normal equations . . . . .	412
6.4.2	Solution via SVD . . . . .	413
6.5	Non-linear Least Squares [15, Ch. 6] . . . . .	414
6.5.1	(Damped) Newton method . . . . .	414

6.5.2	Gauss-Newton method . . . . .	415
6.5.3	Trust region method (Levenberg-Marquardt method) . . . . .	418
<b>7</b>	<b>Eigenvalues</b>	<b>419</b>
7.1	Theory of eigenvalue problems . . . . .	423
7.2	“Direct” Eigensolvers . . . . .	425
7.3	Power Methods . . . . .	429
7.3.1	Direct power method . . . . .	429
7.3.2	Inverse Iteration [15, Sect. 7.6], [63, Sect. 5.3.2] . . . . .	439
7.3.3	Preconditioned inverse iteration (PINVIT) . . . . .	451
7.3.4	Subspace iterations . . . . .	454
7.3.4.1	Orthogonalization . . . . .	459
7.3.4.2	Ritz projection . . . . .	464
7.4	Krylov Subspace Methods . . . . .	469
7.5	Singular Value Decomposition . . . . .	480
<b>8</b>	<b>Krylov Methods for Linear Systems of Equations</b>	<b>502</b>
8.1	Descent Methods [63, Sect. 4.3.3] . . . . .	503
8.1.1	Quadratic minimization context . . . . .	503
8.1.2	Abstract steepest descent . . . . .	504
8.1.3	Gradient method for s.p.d. linear system of equations . . . . .	505
8.1.4	Convergence of the gradient method . . . . .	507
8.2	Conjugate gradient method (CG) [42, Ch. 9], [15, Sect. 13.4], [63, Sect. 4.3.4] . . . . .	510
8.2.1	Krylov spaces . . . . .	511
8.2.2	Implementation of CG . . . . .	512
8.2.3	Convergence of CG . . . . .	515
8.3	Preconditioning [15, Sect. 13.5], [42, Ch. 10], [63, Sect. 4.3.5] . . . . .	521
8.4	Survey of Krylov Subspace Methods . . . . .	527
8.4.1	Minimal residual methods . . . . .	527
8.4.2	Iterations with short recursions [63, Sect. 4.5] . . . . .	528
<b>9</b>	<b>Filtering Algorithms</b>	<b>531</b>
9.1	Discrete convolutions . . . . .	532
9.2	Discrete Fourier Transform (DFT) . . . . .	539
9.2.1	Discrete convolution via DFT . . . . .	544
9.2.2	Frequency filtering via DFT . . . . .	544
9.2.3	Real DFT . . . . .	551
9.2.4	Two-dimensional DFT . . . . .	552
9.2.5	Semi-discrete Fourier Transform [63, Sect. 10.11] . . . . .	555
9.3	Fast Fourier Transform (FFT) . . . . .	563
9.4	Trigonometric transformations . . . . .	568
9.4.1	Sine transform . . . . .	569
9.4.2	Cosine transform . . . . .	573
9.5	Toeplitz matrix techniques . . . . .	574
9.5.1	Toeplitz matrix arithmetic . . . . .	576
9.5.2	The Levinson algorithm . . . . .	577
<b>10</b>	<b>Clustering Techniques</b>	<b>579</b>
10.1	Kernel Matrices . . . . .	579
10.2	Local Separable Approximation . . . . .	580
10.3	Cluster Trees . . . . .	587
10.4	Algorithm . . . . .	592

<b>11 Numerical Integration – Single Step Methods</b>	<b>601</b>
11.1 Initial value problems (IVP) for ODEs . . . . .	601
11.1.1 Modeling with ordinary differential equations: Examples . . . . .	602
11.1.2 Theory of initial value problems . . . . .	606
11.1.3 Evolution operators . . . . .	610
11.2 Introduction: Polygonal Approximation Methods . . . . .	612
11.2.1 Explicit Euler method . . . . .	613
11.2.2 Implicit Euler method . . . . .	615
11.2.3 Implicit midpoint method . . . . .	616
11.3 General single step methods . . . . .	617
11.3.1 Definition . . . . .	617
11.3.2 Convergence of single step methods . . . . .	620
11.4 Explicit Runge-Kutta Methods . . . . .	626
11.5 Adaptive Stepsize Control . . . . .	633
<b>12 Single Step Methods for Stiff Initial Value Problems</b>	<b>647</b>
12.1 Model problem analysis . . . . .	648
12.2 Stiff Initial Value Problems . . . . .	661
12.3 Implicit Runge-Kutta Single Step Methods . . . . .	666
12.3.1 The implicit Euler method for stiff IVPs . . . . .	667
12.3.2 Collocation single step methods . . . . .	668
12.3.3 General implicit RK-SSMs . . . . .	671
12.3.4 Model problem analysis for implicit RK-SSMs . . . . .	673
12.4 Semi-implicit Runge-Kutta Methods . . . . .	679
12.5 Splitting methods . . . . .	682
<b>13 Structure Preserving Integration [38]</b>	<b>687</b>
<b>Index</b>	<b>692</b>
Symbols . . . . .	703
Examples . . . . .	705

# Chapter 0

## Introduction

### 0.0.1 Focus of this course

- ▷ on **algorithms** (principles, scope, and limitations),
- ▷ on (efficient, stable) **implementation** in MATLAB,
- ▷ on **numerical experiments** (design and interpretation).

#### (0.0.1) What is outside our scope

No emphasis will be put on

- theory and proofs (unless essential for understanding of algorithms).
  - ☞ 401-3651-00L Numerical Methods for Elliptic and Parabolic Partial Differential Equations
  - 401-3652-00L Numerical Methods for Hyperbolic Partial Differential Equations
  - (both courses offered in BSc Mathematics)
- hardware aware implementation (cache hierarchies, CPU pipelining, etc.)
  - ☞ 263-2300-00L How To Write Fast Numerical Code (Prof. M. Püschel, D-INFK)
- issues of high-performance computing (HPC, shard and distributed memory parallelisation, vectorization)
  - ☞ 401-0686-10L High Performance Computing for Science and Engineering (HPCSE, Profs. M. Troyer and P. Koumoutsakos)
  - 263-2800-00L Design of Parallel and High-Performance Computing (Prof. T. Höfler)

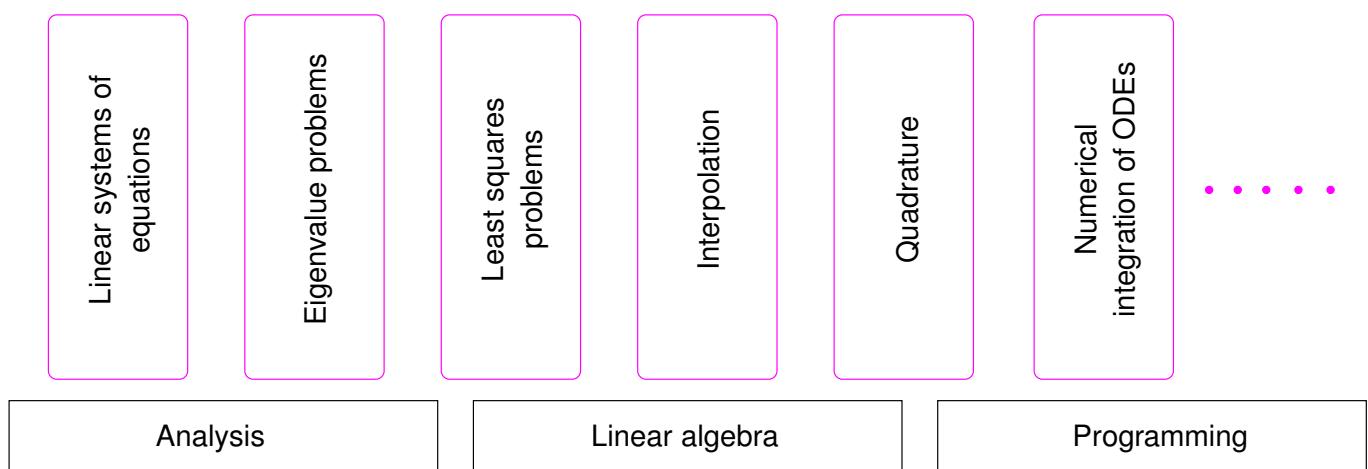
However, note that these other courses partly rely on knowledge of elementary numerical methods so that this course can be regarded as a prerequisite.

---

## Contents

#### (0.0.2) Prerequisites

This course will take for granted basic knowledge of linear algebra, calculus, and programming, that you should have acquired during your first year at ETH.



### (0.0.3) Numerical methods: A motley toolbox

This course discusses **elementary numerical methods and techniques**

They are vastly different in terms of ideas, design, analysis, and scope of application. They are the items in a **toolbox**, some only loosely related by the common purpose of being building blocks for codes for numerical simulation.

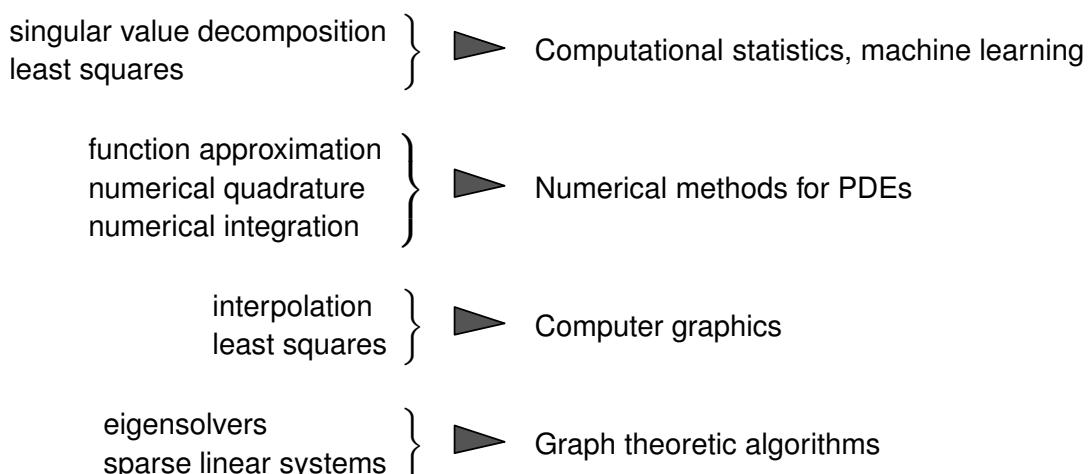
 Do not expect much coherence between the chapters of this course!

A: “Stop putting a hammer, screwdriver, and duct tape in one box! They have nothing to do with each other!”

B: “I might need any of these tools when fixing something about the house”

### (0.0.4) Relevance of this course

I am a student of computer science. After the exam, may I safely forget everything I have learned in this mandatory “numerical methods” course?



numerical integration }  Computer animation

and many more applications of fundamental numerical methods . . .

---

## 0.0.2 Goals

- ★ Knowledge of the fundamental algorithms in numerical mathematics
- ★ Knowledge of the essential terms in numerical mathematics and the techniques used for the analysis of numerical algorithms
- ★ Ability to choose the appropriate numerical method for concrete problems
- ★ Ability to interpret numerical results
- ★ Ability to implement numerical algorithms efficiently in C++ using numerical libraries

Indispensable:

Learning by doing (→ exercises)

## 0.0.3 To avoid misunderstandings

### (0.0.5) “Lecture notes”

These course materials are neither a textbook nor comprehensive lecture notes.  
They are meant to be supplemented by explanations given in class.

Some pieces of advice:

- ★ the lecture material is not designed to be self-contained, but to supplement explanations in class.
- ★ this document is not meant for mere reading, but for working with,
- ★ turn pages all the time and follow the numerous cross-references,
- ★ study the relevant section of the course material when doing homework problems,
- ★ study referenced literature to refresh prerequisite knowledge and for alternative presentation of the material (from a different angle, maybe).

### (0.0.6) Comprehension is a process . . .



The course is difficult and demanding (*i.e.* ETH level)

- ★ Do **not** expect to understand everything in class. The average student will
  - understand about one third of the material when attending the lectures,
  - understand another third when making a *serious effort* to solve the homework problems,

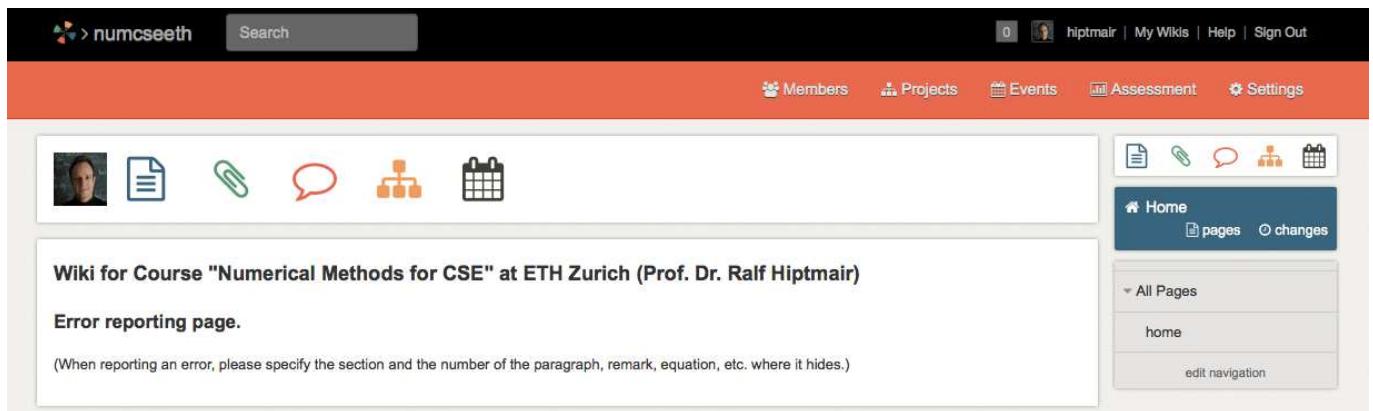
- hopefully understand the remaining third when studying for the examination after the end of the course.

Perseverance will be rewarded!

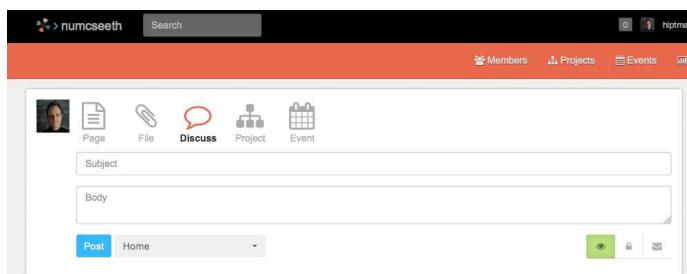
#### 0.0.4 Reporting errors

As the documents will always be in a state of flux, they will inevitably and invariably teem with small errors, mainly typos and omissions.

Please report errors in the lecture material through the **Course Wiki**!



The screenshot shows the homepage of the numcseeth course wiki. At the top, there's a navigation bar with links for Members, Projects, Events, Assessment, and Settings. Below the navigation bar is a toolbar with icons for User profile, Page, File, Discuss, Project, and Event. The main content area displays the title "Wiki for Course 'Numerical Methods for CSE' at ETH Zurich (Prof. Dr. Ralf Hiptmair)" and a section titled "Error reporting page." It includes a note: "(When reporting an error, please specify the section and the number of the paragraph, remark, equation, etc. where it hides.)". To the right, there's a sidebar with a "Home" section showing "pages" and "changes", and a "All Pages" section listing "home".



The screenshot shows a "Discuss" page on the numcseeth course wiki. The page has fields for "Subject" and "Body", and buttons for "Post" and "Home". To the right of the form, there is a message: "Please point out errors by leaving a **comment** in the Wiki ("Discuss" menu item)."

When reporting an error, please specify the section and the number of the paragraph, remark, equation, etc. where it hides. You need not give a page number.

#### 0.0.5 Literature

Parts of the following textbooks may be used as supplementary reading for this course. References to relevant sections will be provided in the course material.

Studying extra literature is not required for following this course!

\* [6] U. ASCHER AND C. GREIF, *A First Course in Numerical Methods*, SIAM, Philadelphia, 2011.

Comprehensive introduction to numerical methods with an algorithmic focus based on MATLAB.  
(Target audience: students of engineering subjects)

- \* [15] W. DAHMEN AND A. REUSKEN, *Numerik für Ingenieure und Naturwissenschaftler*, Springer, Heidelberg, 2006.

Good reference for large parts of this course; provides a lot of simple examples and lucid explanations, but also rigorous mathematical treatment.

(Target audience: undergraduate students in science and engineering)

Available for download as [PDF](#)

- \* [[42]] M. HANKE-BOURGEOIS, *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens*, Mathematische Leitfäden, B.G. Teubner, Stuttgart, 2002.

Gives detailed description and mathematical analysis of algorithms and relies on MATLAB. Profound treatment of theory way beyond the scope of this course. (Target audience: undergraduates in mathematics)

- \* [63] A. QUARTERONI, R. SACCO, AND F. SALERI, *Numerical mathematics*, vol. 37 of Texts in Applied Mathematics, Springer, New York, 2000.

Classical introductory numerical analysis text with many examples and detailed discussion of algorithms. (Target audience: undergraduates in mathematics and engineering)

- \* [19] P. DEUFLHARD AND A. HOHMANN, *Numerische Mathematik. Eine algorithmisch orientierte Einführung*, DeGruyter, Berlin, 1 ed., 1991.

Modern discussion of numerical methods with profound treatment of theoretical aspects (Target audience: undergraduate students in mathematics).

Essential prerequisite for this course is a solid knowledge in linear algebra and calculus. Familiarity with the topics covered in the first semester courses is taken for granted, see

- \* [59] K. NIPP AND D. STOFFER, *Lineare Algebra*, vdf Hochschulverlag, Zürich, 5 ed., 2002.
- \* [34] M. GUTKNECHT, *Lineare algebra*, lecture notes, SAM, ETH Zürich, 2009.  
[PDF](#) available online.
- \* [77] M. STRUWE, *Analysis für Informatiker*. Lecture notes, ETH Zürich, 2009.

## 0.1 Specific information

### 0.1.1 Assistants and exercise classes

Lecturer:	<a href="#">Prof. Ralf Hiptmair</a>	HG G 58.2,	✉ 044 632 3404,	<a href="mailto:hiptmair@sam.math.ethz.ch">hiptmair@sam.math.ethz.ch</a>
Assistants:	<a href="#">Giovanni Alberti</a> , <a href="#">Filippo Leonardi</a> , David Gontier, Francisco Romero, Daniel Hupp, Marija Kranjčević, Fabian Thüring, Alexander Bohn, Stefan Beyeler, Thomas Graf,	HG J 47, HG J 45,	✉ 044 632 4320, ✉ 044 633 9379,	<a href="mailto:giovanni.alberti@sam.math.ethz.ch">giovanni.alberti@sam.math.ethz.ch</a> <a href="mailto:filippo.leonardi@sam.math.ethz.ch">filippo.leonardi@sam.math.ethz.ch</a> <a href="mailto:david.gontier@sam.math.ethz.ch">david.gontier@sam.math.ethz.ch</a> <a href="mailto:francisco.romero@sam.math.ethz.ch">francisco.romero@sam.math.ethz.ch</a> <a href="mailto:huppd@inf.ethz.ch">huppd@inf.ethz.ch</a> <a href="mailto:marija.kranjcevic@inf.ethz.ch">marija.kranjcevic@inf.ethz.ch</a> <a href="mailto:thfabian@student.ethz.ch">thfabian@student.ethz.ch</a> <a href="mailto:abohn@student.ethz.ch">abohn@student.ethz.ch</a> <a href="mailto:stbeyele@student.ethz.ch">stbeyele@student.ethz.ch</a> <a href="mailto:thomas.graf@student.ethz.ch">thomas.graf@student.ethz.ch</a>

Though the assistants email addresses are provided above, their use should be restricted to cases of

emergency:

Avoid sending email messages to the lecturer or the assistants. They will not be answered!

Questions should be asked in class (in public or during the break in private), in the tutorials, or in the study center hours.

Classes: Thu, 08.15-10.00 (HG G 5), Fri, 13.15-15.00 (HG F 7)

Tutorials: Mon, 10.15-12.00 (CLA E 4, LFW E 11, LFW E13, ML H 41.1, ML J 34.1)

Mon, 13.15-15.00 (CLA E 4, HG E 33.3, HG E 33.5, HG G 26.5, LEE D 105)

Study center: Mon, 18.00-20.00 (HG E 41)

Before the first tutorial you will receive a link where you can register to a tutorial class. Keep in mind that one tutorial will be held in German and one will be reserved for CSE students.

## 0.1.2 Assignments

A steady and persistent effort spent on homework problems is essential for success in this course.

You should expect to spend 4-6 hours per week on trying to solve the homework problems. Since many involve small coding projects, the time it will take an individual student to arrive at a solution is hard to predict.

### (0.1.1) Assignment sheet and tutors' corrections

- ★ The assignment sheets will be uploaded on the course [webpage](#) on Thursday every week.
- ★ Some or all of the problems of an assignment sheet will be discussed in the tutorial classes on Monday 1<sup>1/2</sup> weeks after the problem sheet has been published.
- ★ A few problems on each sheet will be marked as **core problems**. Every participant of the course is strongly advised to try and solve *at least* the core problems.
- ★ If you want your tutor to examine your solution of the current problem sheet, please put it into the plexiglass trays in front of HG G 53/54 by the Thursday after the publication. You should submit your codes using the [online submission interface](#). This is voluntary, but feedback on your performance on homework problems can be important.
- ★ You are encouraged to hand-in incomplete and wrong solutions, you can receive valuable feedback even on incomplete attempts.
- ★ Please clearly mark the homework problems that you want your tutor to inspect.

## 0.1.3 Information on examinations

### (0.1.2) Examinations during the teaching period

From the ETH course directory:

A 30-minute **mid-term** and a 30-minute **end-term** exam will be held during the teaching period on dates specified below. They will be regarded as **central elements**, graded on a pass-fail basis, and *at least one of them has to be passed in order to be admitted to the final exam* in the exam session. In case of proven illness, either exam can be repeated in the end of the teaching period.

Both will be closed book examinations on paper.

Dates:

- **mid-term:** **Friday, 23rd of October 2015**
- **end-term:** **Friday, 18th of December 2015**
- **Make-up exam:** **Monday, January 11, 2016**

---

If you have **cogent reasons** why you cannot take the exams on 23.10 or 18.12, please contact Giovanni Alberti by email and send him a request to be allowed to take part in the make-up exam on 11.01. You will be notified, if your request has been approved.

#### (0.1.3) Final examination during exam session

- ✿ Three-hour written examination involving coding problems to be done at the computer on  
**Feb 2, 2016**
- ✿ Dry-run for computer based examination:  
TBA, registration via course website
- ✿ Subjects of examination:
  - All topics, which have been addressed in class or in a homework problem.
- ✿ Lecture documents will be available as PDF during the examination. The corresponding final version of the lecture documents will be made available on TBA
- ✿ The exam questions will be asked in English.

---

## 0.2 Programming in C++11

C++11 is the *current* ANSI/ISO standard for the programming language C++. on the one hand, it offers a wealth of features and possibilities. On the other hand, this can be confusing and even be prone to inconsistencies. A major cause of inconsistent design is the requirement with backward compatibility with the C programming language and the earlier standard C++98.

However, C++ has become the main language in computational science and engineering and high performance computing. Therefore this course relies on C++ to discuss the implementation of numerical methods.

In fact C++ is a blend of different programming paradigms:

- an **object oriented** core providing classes, inheritance, and runtime polymorphism,
- a powerful **template mechanism** for parametric types and partial specialization, enabling *template meta-programming* and compile-time polymorphism,
- a collection of abstract data containers and basic algorithms provided by the **Standard Template Library** (STL).



*Supplementary reading.* A popular book for learning C++ that has been upgraded to include the latest C++11 standard is [52].

The book [46] gives a comprehensive presentation of the new features of C++11 compared to earlier versions of C++.

There are plenty of online reference pages for C++, for instance <http://en.cppreference.com> and <http://www.cplusplus.com/>.

The following sections highlight a few particular aspects of C++11.

## 0.2.1 Function Arguments and Overloading

### (0.2.1) Function overloading

Argument types are an integral part of a function declaration in C++. Hence the following functions are different

```
int* f(int);           // use this in the case of a single numeric argument
double f(int *);      // use only, if pointer to a integer is given
void f(const MyClass &); // use when called for a MyClass object
```

and the compiler selects the function to be used depending on the type of the arguments following rather sophisticated rules, refer to [overload resolution rules](#). Complications arise, because implicit type conversions have to be taken into account. In case of ambiguity a compile-time error will be triggered. Functions cannot be distinguished by return type!

For member functions (methods) of classes an additional distinction can be introduced by the **const** specifier:

```
struct MyClass {
    double &f(double);          // use for a mutable object of type MyClass
    double f(double) const;     // use this version for a constant object
    ...
};
```

The second version of the method `f` is invoked for *constant objects* of type **MyClass**.

### (0.2.2) Operator overloading

In C++ unary and binary **operators** like `=`, `==`, `+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`, `%`, `&&`, `||`, etc. are regarded as functions with a fixed number of arguments (one or two). For built-in numeric and logic types they are defined already. They can be extended to any other type, for instance

```
 MyClass operator + (const MyClass &, const MyClass &);  
 MyClass operator + (const MyClass &, double);  
 MyClass operator + (const MyClass &); // unary + !
```

The same selection rules as for function overloading apply. Of course, operators can also be introduced as class member functions.

C++ gives complete freedom to overload operators. However, the semantics of the new operators should be close to the customary use of the operator.

### (0.2.3) Passing arguments by value and by reference

Consider a generic function declared as follows:

```
 void f(MyClass x); // Argument x passed by value.
```

When `f` is invoked, a *temporary copy* of the argument is created through the **copy constructor** or the **move constructor** of **MyClass**. The new temporary object is a *local variable* inside the function body.

When a function is declared as follows

```
 void f(MyClass &x); // Argument x passed by reference.
```

then the argument is passed to the scope of the function and can be changed inside the function. *No copies* are created. If one wants to avoid the creation of temporary objects, which may be costly, but also wants to indicate that the argument will not be modified inside `f`, then the declaration should read

```
 void f(const MyClass &x); // Argument x passed by constant reference.
```

New in C++11 is **move semantics**, enabled in the following definition

```
 void f(const MyClass &&x); // Optional shallow copy
```

In this case, if the scope of the object passed as argument is merely the function call or `std::move()` tags it as disposable, the **move constructor** of **MyClass** is invoked, which will usually do a *shallow copy* only. Refer to Code 0.2.19 for an example.

## 0.2.2 Templates

### (0.2.4) Function templates

The template mechanism supports parameterization of definitions of classes and functions by type. An example of a **function templates** is

```
template <typename ScalarType, typename VectorType>
VectorType saxpy(ScalarType alpha, const VectorType &x, const
                  VectorType &y)
{ return (alpha*x+y); }
```

Depending on the concrete type of the arguments the compiler will instantiate particular versions of this function, for instance `saxpy<float, double>`, when `alpha` is of type `float` and both `x` and `y` are of type `double`. In this case the return type will be `float`.

For the above example the compiler will be able to deduce the types `ScalarType` and `VectorType` from the arguments. The programmer can also specify the types directly through the `< >`-syntax: `saxpy<double,` if an instantiation for all arguments of type `double` is desired. In case, the arguments do not supply enough information about the type parameters, specifying (some of) them through `< >` is mandatory.

### (0.2.5) Class templates

A **class template** defines a class depending on one or more type parameters, for instance

```
template <typename T>
class MyClstTempl {
public:
    using parm_t = typename T::value_t; // T-dependent type
    MyClstTempl(void);           // Default constructor
    MyClstTempl(const T&);       // Constructor with an argument
    template <typename U>
    T memfn(const T&, const U&) const; // Templatized member function
private:
    T *ptr; // Data member, T-pointer
};
```

Types `MyClstTempl<T>` for a concrete choice of `T` are instantiated when a corresponding object is declared, for instance via

```
double x = 3.14;
MyClass myobj; // Default construction of an object
MyClstTempl<double> tinstd; // Instantiation for T = double
MyClstTempl<MyClass> mytinst(myobj); // Instantiation for T = MyClass
MyClass ret = mytinst.memfn(myobj,x); // Instantiation of member
                                         function for U = double, automatic type deduction
```

The types spawned by a template for different parameter types have nothing to do with each other.

### Requirements on parameter types

The parameter types for a template have to provide all type definitions, member functions, operators, and data to make possible the instantiation (“compilation”) of the class of function template.

### 0.2.3 Function Objects and Lambda Functions

A function object is an object of a type that provides an overloaded “function call” **operator** `( )`. Function objects can be implemented in two different ways:

- (I) through special classes like the following that realizes a function  $\mathbb{R} \mapsto \mathbb{R}$

```
class MyFun {
public:
    ...
    double operator (double x) const; // Evaluation operator
    ...
};
```

The evaluation operator can take more than one argument and need not be declared `const`.

- (II) through **lambda functions**, an “anonymous function” defined as

```
[<capture list>] (<arguments>) -> <return type> { body; }
```

where `<capture list>` is a list of variables from the local scope to be passed to the lambda function; an & indicates passing by reference,  
`<arguments>` is a comma separated list of function arguments complete with types,  
`<return type>` is an *optional* return type; often the compiler will be able to deduce the return type from the definition of the function.

Function classes should be used, when the function is needed in different places, whereas lambda functions for short functions intended for single use.

#### C++11 code 0.2.7: Demonstration of use of lambda function

```
1 int main() {
2     // initialize a vector from an initializer list
3     std::vector<double> v({1.2,2.3,3.4,4.5,5.6,6.7,7.8});
4     // A vector of the same length
5     std::vector<double> w(v.size());
6     // Do cumulative summation of v and store result in w
7     double sum = 0;
8     std::transform(v.begin(),v.end(),w.begin(),
9                 [&sum] (double x) { sum += x; return sum; });
10    cout << "sum = " << sum << ", w = [ ";
11    for(auto x: w) cout << x << ' ';
12    cout << ']' << endl;
13    return(0);
14 }
```

In this code the lambda function captures the local variable `sum` by reference.

### 0.2.4 Multiple Return Values

In MATLAB it is customary to return several variables from a function call:

```
function [x,y,z] = f(a,b)
```

In C++ this is possible by using the **tuple** utility. For instance, the following function computes the minimal and maximal element of a vector and also returns its cumulative sum. It returns all these values.

#### C++11 code 0.2.8: Function with multiple return values

```

1 template<typename T>
2 std::tuple<T,T,std::vector<T>> extcumsum(const std::vector<T> &v) {
3     // Local summation variable captured by reference by lambda function
4     T sum{};
5     // temporary vector for returning cumulative sum
6     std::vector<T> w{};
7     // cumulative summation
8     std::transform(v.cbegin(),v.cend(),back_inserter(w),
9                 [&sum] (T x) { sum += x; return(sum); });
10    return(std::tuple<T,T,std::vector<T>>
11           (*std::min_element(v.cbegin(),v.cend()), 
12            *std::max_element(v.cbegin(),v.cend()), 
13            std::move(w)));
14 }
```

This code snippet shows how to extract the individual components of the tuple returned by the previous function.

#### C++11 code 0.2.9: Calling a function with multiple return values

```

1 int main () {
2     // initialize a vector from an initializer list
3     std::vector<double> v({1.2,2.3,3.4,4.5,5.6,6.7,7.8});
4     // Variables for return values
5     double minv,maxv; // Extremal elements
6     std::vector<double> cs; // Cumulative sums
7     std::tie(minv,maxv,cs) = extcumsum(v);
8     cout << "min = " << minv << ", max = " << maxv << endl;
9     cout << "cs = [ "; for(double x: cs) cout << x << ' '; cout << "] "
10    << endl;
11    return(0);
12 }
```

Be careful: many temporary objects might be created! A demonstration of this hidden cost is given in Exp. 0.2.36.

## 0.2.5 A Vector Class

Since C++ is an object oriented programming language, datatypes defined by **classes** play a pivotal role in every C++ program. Here, we demonstrate the main ingredients of a class definition and other important facilities of C++ for the class **MyVector** meant for objects representing vectors from  $\mathbb{R}^n$ .

**C++11 class 0.2.10: Definition of a simple vector class **MyVector****

```

1  namespace myvec {
2      class MyVector {
3          public:
4              using value_t = double;
5              // Constructor creating constant vector, also default constructor
6              explicit MyVector(std::size_t n = 0, double val = 0.0);
7              // Constructor: initialization from an STL container
8              template <typename Container> MyVector(const Container &v);
9              // Constructor: initialization from an STL iterator range
10             template <typename Iterator> MyVector(Iterator first, Iterator last);
11             // Copy constructor, computational cost  $O(n)$ 
12             MyVector(const MyVector &mv);
13             // Move constructor, computational cost  $O(1)$ 
14             MyVector(MyVector &&mv);
15             // Copy assignment operator, computational cost  $O(n)$ 
16             MyVector &operator = (const MyVector &mv);
17             // Move assignment operator, computational cost  $O(1)$ 
18             MyVector &operator = (MyVector &&mv);
19             // Destructor
20             virtual ~MyVector(void);
21             // Type conversion to STL vector
22             operator std::vector<double> () const;
23
24             // Returns length of vector
25             std::size_t length(void) const { return n; }
26             // Access operators: rvalue & lvalue, with range check
27             double operator [] (std::size_t i) const;
28             double &operator [] (std::size_t i);
29             // Comparison operators
30             bool operator == (const MyVector &mv) const;
31             bool operator != (const MyVector &mv) const;
32             // Transformation of a vector by a function  $\mathbb{R} \rightarrow \mathbb{R}$ 
33             template <typename Functor>
34             MyVector &transform(Functor &&f);
35
36             // Overloaded arithmetic operators
37             // In place vector addition:  $x += y;$ 
38             MyVector &operator +=(const MyVector &mv);
39             // In place vector subtraction:  $x -= y;$ 
40             MyVector &operator -=(const MyVector &mv);
41             // In place scalar multiplication:  $x *= a;$ 
42             MyVector &operator *=(double alpha);
43             // In place scalar division:  $x /= a;$ 
44             MyVector &operator /=(double alpha);
45             // Vector addition
46             MyVector operator + (MyVector mv) const;
47             // Vector subtraction
48             MyVector operator - (const MyVector &mv) const;
49             // Scalar multiplication from right and left:  $x = a*y;$   $x = y*a$ 

```

```

50  MyVector operator * (double alpha) const;
51  friend MyVector operator * (double alpha, const MyVector &);
52  // Scalar division: x = y/a;
53  MyVector operator / (double alpha) const;

54
55  // Euclidean norm
56  double norm(void) const;
57  // Euclidean inner product
58  double operator *(const MyVector &) const;
59  // Output operator
60  friend std::ostream &
61  operator << (std::ostream &, const MyVector &mv);

62
63  static bool dbg; // Flag for verbose output
private:
64  std::size_t n; // Length of vector
65  double *data; // data array (standard C array)
66 }
67 }
68

```

Note the use of a public **static** data member `dbg` in Line 63 that can be used to control debugging output by setting `MyVector::dbg = true` or `MyVector::dbg = false`.

### Remark 0.2.11 (Contiguous arrays in C++)

The class **MyVector** uses a C-style array and dynamic memory management with **new** and **delete** to store the vector components. This is for demonstration purposes only and not recommended.

#### Arrays in C++

In C++ use the **STL container `std::vector<T>`** for storing data in contiguous memory locations.

### (0.2.13) Member and friend functions of **MyVector**

#### C++11 code 0.2.14: **Constructor for constant vector, also default constructor, see Line 28**

```

1  MyVector::MyVector(std::size_t _n, double _a):n(_n),data(nullptr) {
2      if (dbg) cout << "{ Constructor MyVector(" << _n
3          << ") called " << '}' << endl;
4      if (n > 0) data = new double [_n];
5      for (std::size_t l=0;l<n;++l) data[l] = _a;
6  }

```

This constructor can also serve as default constructor (a constructor that can be invoked without any argument), because defaults are supplied for all its arguments.

The following two constructors initialize a vector from sequential containers according to the conventions of the STL.

#### C++11 code 0.2.15: Templated **constructors** copying vector entries from an STL container

```

1 template <typename Container>
2 MyVector::MyVector(const Container &v):n(v.size()),data(nullptr) {
3     if (dbg) cout << "{MyVector(length " << n
4                                     << ") constructed from container" << '}' << endl;
5     if (n > 0) {
6         double *tmp = (data = new double [n]);
7         for(auto i: v) *tmp++ = i; // foreach loop
8     }
9 }
```

Note the use of the new C++11 facility of a “foreach loop” iterating through a container in Line 7.

#### C++11 code 0.2.16: **Constructor** initializing vector from STL iterator range

```

1 template <typename Iterator>
2 MyVector::MyVector(Iterator first,Iterator last):n(0),data(nullptr) {
3     n = std::distance(first,last);
4     if (dbg) cout << "{MyVector(length " << n
5                                     << ") constructed from range" << '}' << endl;
6     if (n > 0) {
7         data = new double [n];
8         std::copy(first,last,data);
9     }
10 }
```

The use of these constructors is demonstrated in the following code

#### C++11 code 0.2.17: Initialization of a **MyVector** object from an STL vector

```

1 int main() {
2     myvec::MyVector::dbg = true;
3     std::vector<int> ivec = { 1,2,3,5,7,11,13 }; // initializer list
4     myvec::MyVector v1(ivec.cbegin(),ivec.cend());
5     myvec::MyVector v2(ivec);
6     myvec::MyVector vr(ivec.crbegin(),ivec.crend());
7     cout << "v1 = " << v1 << endl;
8     cout << "v2 = " << v2 << endl;
9     cout << "vr = " << vr << endl;
10    return(0);
11 }
```

The following output is produced:

```
{MyVector(length 7) constructed from range}
{MyVector(length 7) constructed from container}
{MyVector(length 7) constructed from range}
```

```
v1 = [ 1,2,3,5,7,11,13 ]
v2 = [ 1,2,3,5,7,11,13 ]
vr = [ 13,11,7,5,3,2,1 ]
{Destructor for MyVector(length = 7)}
{Destructor for MyVector(length = 7)}
{Destructor for MyVector(length = 7)}
```

The copy constructor listed next relies on the *STL algorithm* `std::copy` to copy the elements of an existing object into a newly created object. This takes *n* operations.

#### C++11 code 0.2.18: Copy constructor

```
1 MyVector::MyVector(const MyVector &mv):n(mv.n),data(nullptr) {
2     if (dbg) cout << "{Copy construction of MyVector(length "
3             << n << ")" << '}' << endl;
4     if (n > 0) {
5         data = new double [n];
6         std::copy_n(mv.data,n,data);
7     }
8 }
```

An important new feature of C++11 is **move semantics** which helps avoid expensive copy operations. The following implementation just performs a shallow copy of pointers and, thus, for large *n* is much cheaper than a call to the copy constructor from Code 0.2.18. The source vector is left in an empty vector state.

#### C++11 code 0.2.19: Move constructor

```
1 MyVector::MyVector(MyVector &&mv):n(mv.n),data(mv.data) {
2     if (dbg) cout << "{Move construction of MyVector(length "
3             << n << ")" << '}' << endl;
4     mv.data = nullptr; mv.n = 0; // Reset victim of data theft
5 }
```

The following code demonstrates the use of `std::move()` to mark a vector object as disposable and allow the compiler the use of the move constructor. The code also uses left multiplication with a scalar, see Code 0.2.32.

#### C++11 code 0.2.20: Invocation of copy and move constructors

```
1 int main() {
2     myvec::MyVector::dbg = true;
3     myvec::MyVector v1(std::vector<double>(
4         {1.2,2.3,3.4,4.5,5.6,6.7,7.8,8.9}));
5     myvec::MyVector v2(2.0*v1); // Scalar multiplication
6     myvec::MyVector v3(std::move(v1));
7     cout << "v1 = " << v1 << endl;
8     cout << "v2 = " << v2 << endl;
9     cout << "v3 = " << v3 << endl;
10    return (0);
11 }
```

This code produces the following output. We observe that `v1` is empty after its data have been “stolen” by `v2`.

```
{MyVector(length 8) constructed from container}
{operator a*, MyVector of length 8}
{Copy construction of MyVector(length 8)}
{operator *=, MyVector of length 8}
{Move construction of MyVector(length 8)}
{Destructor for MyVector(length = 0)}
{Move construction of MyVector(length 8)}
v1 = [ ]
v2 = [ 2.4,4.6,6.8,9,11.2,13.4,15.6,17.8 ]
v3 = [ 1.2,2.3,3.4,4.5,5.6,6.7,7.8,8.9 ]
{Destructor for MyVector(length = 8)}
{Destructor for MyVector(length = 8)}
{Destructor for MyVector(length = 0)}
```

We observe that the object `v1` is reset after having been moved to `v3`.

The next operator effects copy assignment of an rvalue `MyVector` object to an lvalue `MyVector`. This involves  $O(n)$  operations.

#### C++11 code 0.2.21: Copy assignment operator

```
1 MyVector &MyVector::operator = (const MyVector &mv) {
2     if (dbg) cout << "{Copy assignment of MyVector(length "
3             << n << "←" << mv.n << ")" << '}' << endl;
4     if (this == &mv) return (*this);
5     if (n != mv.n) {
6         n = mv.n;
7         if (data != nullptr) delete [] data;
8         if (n > 0) data = new double [n]; else data = nullptr;
9     }
10    if (n > 0) std::copy_n(mv.data, n, data);
11    return (*this);
12 }
```

The move semantics is realized by an assignment operator relying on shallow copying.

#### C++11 code 0.2.22: Move assignment operator

```
1 MyVector &MyVector::operator = (MyVector &&mv) {
2     if (dbg) cout << "{Move assignment of MyVector(length "
3             << n << "←" << mv.n << ")" << '}' << endl;
4     if (data != nullptr) delete [] data;
5     n = mv.n; data = mv.data;
6     mv.n = 0; mv.data = nullptr;
7     return (*this);
8 }
```

The destructor releases memory allocated by `new` during construction or assignment.

**C++11 code 0.2.23: Destructor: releases allocated memory**

```

1 MyVector::~MyVector(void) {
2     if (dbg) cout << "{ Destructor for MyVector(length = "
3             << n << ")" << '}' << endl;
4     if (data != nullptr) delete [] data;
5 }
```

The **operator** keyword is also used to define **implicit type conversions**.

**C++11 code 0.2.24: Type conversion operator: copies contents of vector into STL vector**

```

1 MyVector::operator std::vector<double> () const {
2     if (dbg) cout << "{ Conversion to std::vector, length = " << n <<
3         '}' << endl;
4     return std::move(std::vector<double>(data, data+n));
5 }
```

The bracket operator `[]` can be used to fetch and set vector components. Note that index range checking is performed; an *exception* is thrown for invalid indices. The following code also gives an example of operator overloading as discussed in § 0.2.2.

**C++11 code 0.2.25: rvalue and lvalue access operators**

```

1 double MyVector::operator [] (std::size_t i) const {
2     if (i >= n) throw(std::logic_error("[] out of range"));
3     return data[i];
4 }
5
6 double &MyVector::operator [] (std::size_t i) {
7     if (i >= n) throw(std::logic_error("[] out of range"));
8     return data[i];
9 }
```

Componentwise direct comparison of vectors. Can be dangerous in numerical codes, cf. Rem. 1.5.35.

**C++11 code 0.2.26: Comparison operators**

```

1 bool MyVector::operator == (const MyVector &mv) const
{
3     if (dbg) cout << "{ Comparison ==: " << n << " <-> " << mv.n << '}' "
4         << endl;
5     if (n != mv.n) return(false);
6     else {
7         for(std::size_t l=0;l<n;++l)
8             if (data[l] != mv.data[l]) return(false);
9     }
10    return(true);
}
```

```

11  bool MyVector::operator != (const MyVector &mv) const {
12      return !(*this == mv);
13
14 }
```

The `transform` method applies a function to every vector component and overwrites it with the value returned by the function. The function is passed as an object of a type providing a `()`-operator that accepts a single argument convertible to `double` and returns a value convertible to `double`.

#### C++11 code 0.2.27: Transformation of a vector through a functor `double → double`

```

1 template <typename Functor>
2 MyVector &MyVector::transform(Functor &&f) {
3     for(std::size_t l=0;l<n;++l) data[l] = f(data[l]);
4     return (*this);
5 }
```

The following code demonstrates the use of the `transform` method in combination with

1. a **function object** of the following type

#### C++11 code 0.2.28:

```

1 struct SimpleFunction {
2     SimpleFunction(double _a = 1.0):cnt(0),a(_a) {}
3     double operator () (double x) { cnt++; return(x+a); }
4     int cnt;           // internal counter
5     const double a;   // increment value
6 };
```

2. a **lambda function** defined directly inside the call to `transform`.

#### C++11 code 0.2.29:

```

1 int main() {
2     myvec::MyVector::dbg = false;
3     double a = 2.0; // increment
4     int cnt = 0;    // external counter used by lambda function
5     myvec::MyVector mv(std::vector<double>(
6         {1.2,2.3,3.4,4.5,5.6,6.7,7.8,8.9}));
7     mv.transform([a,&cnt] (double x) { cnt++; return(x+a); });
8     cout << cnt << " operations , mv transformed = " << mv << endl;
9     SimpleFunction trf(a); mv.transform(trf);
10    cout << trf.cnt << " operations , mv transformed = " << mv << endl;
11    mv.transform(SimpleFunction(-4.0));
12    cout << "Final vector = " << mv << endl;
13
14 }
```

The output is

```
8 operations , mv transformed = [ 3.2,4.3,5.4,6.5,7.6,8.7,9.8,10.9 ]
8 operations , mv transformed = [ 5.2,6.3,7.4,8.5,9.6,10.7,11.8,12.9 ]
Final vector = [ 1.2,2.3,3.4,4.5,5.6,6.7,7.8,8.9 ]
```

Operator overloading provides the “natural” vector operations in  $\mathbb{R}^n$  both in place and with a new vector created for the result.

#### C++11 code 0.2.30: In place arithmetic operations (one argument)

```
1 MyVector &MyVector::operator +=(const MyVector &mv) {
2     if (dbg) cout << "{operator +=, MyVector of length "
3             << n << '}' << endl;
4     if (n != mv.n) throw(std::logic_error("+=: vector size mismatch"));
5     for(std::size_t l=0;l<n;++l) data[l] += mv.data[l];
6     return(*this);
7 }
8
9 MyVector &MyVector::operator -=(const MyVector &mv) {
10    if (dbg) cout << "{operator -=, MyVector of length "
11        << n << '}' << endl;
12    if (n != mv.n) throw(std::logic_error("-=: vector size mismatch"));
13    for(std::size_t l=0;l<n;++l) data[l] -= mv.data[l];
14    return(*this);
15 }
16
17 MyVector &MyVector::operator *=(double alpha) {
18     if (dbg) cout << "{operator *=, MyVector of length "
19         << n << '}' << endl;
20     for(std::size_t l=0;l<n;++l) data[l] *= alpha;
21     return(*this);
22 }
23
24 MyVector &MyVector::operator /=(double alpha) {
25     if (dbg) cout << "{operator /=, MyVector of length "
26         << n << '}' << endl;
27     for(std::size_t l=0;l<n;++l) data[l] /= alpha;
28     return(*this);
29 }
```

#### C++11 code 0.2.31: Binary arithmetic operators (two arguments)

```
1 MyVector MyVector::operator + (MyVector mv) const {
2     if (dbg) cout << "{operator +, MyVector of length "
3             << n << '}' << endl;
4     if (n != mv.n) throw(std::logic_error("+: vector size mismatch"));
5     mv += *this;
6     return(std::move(mv));
7 }
8
```

```

9  MyVector MyVector::operator - (const MyVector &mv) const {
10 if (dbg) cout << "{operator +, MyVector of length "
11           << n << '}' << endl;
12 if (n != mv.n) throw(std::logic_error("+: vector size mismatch"));
13 MyVector tmp(*this); tmp -= mv;
14 return (std::move(tmp));
15 }

16
17 MyVector MyVector::operator * (double alpha) const {
18 if (dbg) cout << "{operator *a, MyVector of length "
19           << n << '}' << endl;
20 MyVector tmp(*this); tmp *= alpha;
21 return (std::move(tmp));
22 }

23
24 MyVector MyVector::operator / (double alpha) const {
25 if (dbg) cout << "{operator /, MyVector of length " << n << '}' <<
26           endl;
27 MyVector tmp(*this); tmp /= alpha;
28 return (std::move(tmp));
}

```

### C++11 code 0.2.32: Non-member function for left multiplication with a scalar

```

1 MyVector operator * (double alpha, const MyVector &mv) {
2   if (MyVector::dbg) cout << "{operator a*, MyVector of length "
3           << mv.n << '}' << endl;
4   MyVector tmp(mv); tmp *= alpha;
5   return (std::move(tmp));
6 }

```

### C++11 code 0.2.33: Euclidean norm

```

1 double MyVector::norm(void) const {
2   if (dbg) cout << "{norm: MyVector of length " << n << '}' << endl;
3   double s = 0;
4   for (std::size_t l=0;l<n;++l) s += (data[l]*data[l]);
5   return (std::sqrt(s));
6 }

```

Adopting the notation in some linear algebra texts, the operator `*` has been chosen to designate the Euclidean inner product:

### C++11 code 0.2.34: Euclidean inner product

```

1 double MyVector::operator *(const MyVector &mv) const {
2   if (dbg) cout << "{dot *, MyVector of length " << n << '}' << endl;
3   if (n != mv.n) throw(std::logic_error("dot: vector size mismatch"));

```

```

4   double s = 0;
5   for(std::size_t l=0;l<n;++l) s += (data[l]*mv.data[l]);
6   return(s);
7 }
```

At least for debugging purposes every reasonably complex class should be equipped with output functionality.

#### C++11 code 0.2.35: Non-member function **output operator**

```

1 std::ostream &operator << (std::ostream &o,const MyVector &mv) {
2   o << "[ ";
3   for(std::size_t l=0;l<mv.n;++l)
4     o << mv.data[l] << (l==mv.n-1?',' : ',');
5   return(o << "]");
6 }
```

#### Experiment 0.2.36 (“Behind the scenes” of **MyVector** arithmetic)

The following code highlights the use of operator overloading to obtain readable and compact expressions for vector arithmetic.

#### C++11 code 0.2.37:

```

1 int main() {
2   myvec::MyVector::dbg = true;
3   myvec::MyVector
4     x(std::vector<double>({1.2,2.3,3.4,4.5,5.6,6.7,7.8,8.9}));
5   myvec::MyVector
6     y(std::vector<double>({2.1,3.2,4.3,5.4,6.5,7.6,8.7,9.8}));
```

$$\text{auto } z = x + (x * y) * x + 2.0 * y / (x - y).norm();$$

We run the code and trace calls. This is printed to the console:

```
{MyVector(length 8) constructed from container}
{MyVector(length 8) constructed from container}
{dot *, MyVector of length 8}
{operator a*, MyVector of length 8}
{Copy construction of MyVector(length 8)}
{operator *=, MyVector of length 8}
{Move construction of MyVector(length 8)}
{Destructor for MyVector(length = 0)}
{operator +, MyVector of length 8}
{operator +=, MyVector of length 8}
{Move construction of MyVector(length 8)}
{operator a*, MyVector of length 8}
```

```

{Copy construction of MyVector(length 8)}
{operator *=, MyVector of length 8}
{Move construction of MyVector(length 8)}
{Destructor for MyVector(length = 0)}
{operator +, MyVector of length 8}
{Copy construction of MyVector(length 8)}
{operator -=, MyVector of length 8}
{Move construction of MyVector(length 8)}
{Destructor for MyVector(length = 0)}
{norm: MyVector of length 8}
{operator /, MyVector of length 8}
{Copy construction of MyVector(length 8)}
{operator *=, MyVector of length 8}
{Move construction of MyVector(length 8)}
{Destructor for MyVector(length = 0)}
{operator +, MyVector of length 8}
{operator +=, MyVector of length 8}
{Move construction of MyVector(length 8)}
{Destructor for MyVector(length = 0)}
{Destructor for MyVector(length = 8)}
{Destructor for MyVector(length = 8)}
{Destructor for MyVector(length = 8)}
{Destructor for MyVector(length = 0)}
{Destructor for MyVector(length = 8)}

```

Several temporary objects are created and destroyed and quite a few *copy operations* take place. The situation would be worse unless move semantics was available; if we had not supplied a move constructor, six more copy operations would have been triggered. Even worse, the frequent copying of data runs a high risk of cache misses. This is certainly not an efficient way to do elementary vector operations though it looks elegant at first glance.

### Example 0.2.38 (Gram-Schmidt orthonormalization based on **MyVector** implementation)

Gram-Schmidt orthonormalization has been taught in linear algebra and its theory will be revisited in § 1.5.1. Here we use this simple algorithm from linear algebra to demonstrate the use of the vector class **MyVector** defined in Code 0.2.10.

The templated function `gramschmidt` takes a sequence of vectors stored in a `std::vector` object. The actual vector type is passed as a template parameter. It has to supply `length` and `norm` member functions as well as in place arithmetic operations `-=`, `/` and `=`. Note the use of the highlighted methods of the `std::vector` class.

#### C++11 code 0.2.39: templated function for Gram-Schmidt orthonormalization

```

1 template <typename Vec>
2 std ::vector<Vec> gramschmidt(const std ::vector<Vec> &A, double
   eps=1E-14) {
3   const int k = A.size(); // no. of vectors to orthogonalize

```

```

4   const int n = A[0].length(); // length of vectors
5   cout << "gramschmidt orthogonalization for " << k << ' ' << n <<
6     "vectors" << endl;
7   std::vector<Vec> Q({A[0]/A[0].norm()}); // output vectors
8   for(int j=1;(j<k) && (j<n);++j) {
9     Q.push_back(A[j]);
10    for(int l=0;l<j;++l) Q.back() -= (A[j]*Q[l])*Q[l];
11    if (Q.back().norm() < eps*A[j].norm()) { // premature termination ?
12      Q.pop_back(); break;
13    }
14    Q.back() /= Q.back().norm(); // normalization
15  }
16  return (std::move(Q)); // return at end of local scope
}

```

This driver program calls a function that initializes a sequence of vectors and then orthonormalizes them by means of the Gram-Schmidt algorithm. Eventually orthonormality of the computed vectors is tested. Please pay attention to

- the use of auto to avoid cumbersome type declarations,
- the for loops following the “foreach” syntax.
- automatic indirect template type deduction for the templated function `gramschmidt` from its argument. In Line 6 the function `gramschmidt<MyVector>` is instantiated.

#### C++11 code 0.2.40: Driver code for Gram-Schmidt orthonormalization

```

1 int main() {
2   myvec::MyVector::dbg = false;
3   const int n = 7; const int k = 7;
4   auto A(initvectors(n,k,[] (int i,int j)
5     { return std::min(i+1,j+1); }));
6   auto Q(gramschmidt(A)); // instantiate template for MyVector
7   cout << "Set of vectors to be orthonormalized:" << endl;
8   for (const auto &a: A) { cout << a << endl; }
9   cout << "Output of Gram–Schmidt orthonormalization: " << endl;
10  for (const auto &q: Q) { cout << q << endl; }
11  cout << "Testing orthogonality:" << endl;
12  for (const auto &qi: Q) {
13    for (const auto &qj: Q)
14      cout << std::setprecision(3) << std::setw(9) << qi*qj << ' ';
15    cout << endl; }
16  return(0);
17 }

```

This initialization function takes a functor argument as discussed in Section 0.2.3.

#### C++11 code 0.2.41: Initialization of a set of vectors through a functor with two arguments

```
1 template<typename Functor>
2     std::vector<myvec::MyVector>
3         initvectors( std::size_t n, std::size_t k, Functor &&f ) {
4             std::vector<MyVector> A{ };
5             for( int j=0;j<k;++j ) {
6                 A.push_back( MyVector(n) );
7                 for( int i=0;i<n;++i )
8                     (A.back())[i] = f(i,j);
9             }
10            return( std::move(A) );
11        }
```

# Chapter 1

## Computing with Matrices and Vectors

### (1.0.1) Prerequisite knowledge for Chapter 1

This chapter heavily relies on concepts and techniques from [linear algebra](#) as taught in the 1st semester introductory course. Knowledge of the following topics from linear algebra will be taken for granted and they should be refreshed in case of gaps:

- Operations involving matrices and vectors [[59](#), Ch. 2]
- Computations with block-structured matrices
- Linear systems of equations: existence and uniqueness of solutions [[59](#), Sects. 1.2, 3.3]
- Gaussian elimination [[59](#), Ch. 2]
- LU-decomposition and its connection with Gaussian elimination [[59](#), Sect. 2.4]

### (1.0.2) Levels of operations in simulation codes

The lowest level of real arithmetic available on computers are the [elementary operations](#) “+”, “-”, “\*”, “\”, “^”, usually implemented in hardware. The next level comprises computations on finite arrays of real numbers, the [elementary linear algebra operations](#) (BLAS). On top of them we build complex algorithms involving iterations and approximations.

Complex iterative/recursive/approximative algorithms
Linear algebra operations on arrays (BLAS)
Elementary operations in $\mathbb{R}$

Same as hardly ever anyone will contemplate implementing elementary operations on binary data formats, well tested and optimised code libraries should be used for all elementary linear algebra operations in simulation codes. This chapter will introduce you to such libraries and how to use them smartly.

## Contents

1.1 Fundamentals . . . . .	33
1.1.1 Notations . . . . .	33
1.1.2 Classes of matrices . . . . .	35

<b>1.2 Software and Libraries . . . . .</b>	<b>37</b>
1.2.1 MATLAB . . . . .	37
1.2.2 PYTHON . . . . .	40
1.2.3 EIGEN . . . . .	41
1.2.4 Matrix storage formats . . . . .	46
<b>1.3 Basic linear algebra operations . . . . .</b>	<b>54</b>
1.3.1 Elementary matrix-vector calculus . . . . .	54
1.3.2 BLAS – Basic Linear Algebra Subprograms . . . . .	61
<b>1.4 Computational effort . . . . .</b>	<b>68</b>
1.4.1 (Asymptotic) complexity . . . . .	69
1.4.2 Cost of basic operations . . . . .	70
1.4.3 Reducing complexity in numerical linear algebra: Some tricks . . . . .	71
<b>1.5 Machine arithmetic . . . . .</b>	<b>75</b>
1.5.1 Experiment: Loss of orthogonality . . . . .	75
1.5.2 Machine numbers . . . . .	79
1.5.3 Roundoff errors . . . . .	82
1.5.4 Cancellation . . . . .	86
1.5.5 Numerical stability . . . . .	99
<b>1.6 Direct methods for linear systems . . . . .</b>	<b>104</b>
1.6.1 Theory: Linear systems of equations . . . . .	107
1.6.1.1 Existence and uniqueness of solutions . . . . .	107
1.6.1.2 Sensitivity of linear systems . . . . .	107
1.6.2 Gaussian Elimination . . . . .	110
1.6.2.1 Basic algorithm . . . . .	110
1.6.2.2 LU-decomposition . . . . .	117
1.6.2.3 Pivoting . . . . .	124
1.6.3 Stability of Gaussian Elimination . . . . .	129
1.6.4 Elimination solvers for linear systems of equations . . . . .	136
1.6.5 Exploiting Structure when Solving Linear Systems . . . . .	141
<b>1.7 Sparse Linear Systems . . . . .</b>	<b>146</b>
1.7.1 Sparse matrix storage formats . . . . .	148
1.7.2 Sparse matrices in MATLAB . . . . .	150
1.7.3 Sparse matrices in EIGEN . . . . .	161
1.7.4 Direct Solution of Sparse Linear Systems of Equations . . . . .	162
1.7.5 LU-factorization of sparse matrices . . . . .	165
1.7.6 Banded matrices [15, Sect. 3.7] . . . . .	170
<b>1.8 Stable Gaussian elimination without pivoting . . . . .</b>	<b>177</b>

## 1.1 Fundamentals

### 1.1.1 Notations

The notations in this course try to adhere to established conventions. Since these may not be universal, idiosyncrasies cannot be avoided completely. Notations in textbooks may be different, beware!

☞ notation for generic field of numbers:  $\mathbb{K}$

In this course,  $\mathbb{K}$  will designate either  $\mathbb{R}$  (real numbers) or  $\mathbb{C}$  (complex numbers); complex arithmetic [77, Sect. 2.5] plays a crucial role in many applications, for instance in signal processing.

### (1.1.1) Notations for vectors

- \* **Vectors** = are  $n$ -tuples ( $n \in \mathbb{N}$ ) with components  $\in \mathbb{K}$ .

vector = one-dimensional array (of real/complex numbers)

vectors will usually be denoted by small **bold** symbols:  $\mathbf{a}, \mathbf{b}, \dots, \mathbf{x}, \mathbf{y}, \mathbf{z}$

- \* Default in this lecture: vectors = **column vectors**

$$\begin{array}{c} \left[ \begin{array}{c} x_1 \\ \vdots \\ x_n \end{array} \right] \in \mathbb{K}^n \\ \text{column vector} \end{array} \quad \left| \quad \begin{array}{c} [x_1 \dots x_n] \in \mathbb{K}^{1,n} \\ \text{row vector} \end{array} \right.$$

$\mathbb{K}^n \triangleq$  vector space of *column vectors* with  $n$  components in  $\mathbb{K}$ .

notation for column vectors: **bold** small roman letters, e.g.  $\mathbf{x}, \mathbf{y}, \mathbf{z}$

- \* **Transposing:**  $\left\{ \begin{array}{ccc} \text{column vector} & \mapsto & \text{row vector} \\ \text{row vector} & \mapsto & \text{column vector} \end{array} \right.$

$$\left[ \begin{array}{c} x_1 \\ \vdots \\ x_n \end{array} \right]^T = [x_1 \dots x_n], \quad [x_1 \dots x_n]^T = \left[ \begin{array}{c} x_1 \\ \vdots \\ x_n \end{array} \right]$$

Notation for row vectors:  $\mathbf{x}^T, \mathbf{y}^T, \mathbf{z}^T$

- \* Addressing vector components:

two notations:  $\mathbf{x} = [x_1 \dots x_n]^T \rightarrow x_i, i = 1, \dots, n$   
 $\mathbf{x} \in \mathbb{K}^n \rightarrow (\mathbf{x})_i, i = 1, \dots, n$

- \* Selecting sub-vectors:

notation:  $\mathbf{x} = [x_1 \dots x_n]^T \Rightarrow (\mathbf{x})_{k:l} = (x_k, \dots, x_l)^T, 1 \leq k \leq l \leq n$

\*  $j$ -th unit vector:  $\mathbf{e}_j = [0, \dots, 1, \dots, 0]^T, (\mathbf{e}_j)_i = \delta_{ij}, i, j = 1, \dots, n$ .

notation: Kronecker symbol  $\delta_{ij} := 1$ , if  $i = j$ ,  $\delta_{ij} := 0$ , if  $i \neq j$ .

### (1.1.2) Notations and notions for matrices

- \* **Matrices** = two-dimensional arrays of real/complex numbers

$$\mathbf{A} := \left[ \begin{array}{ccc} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{array} \right] \in \mathbb{K}^{n,m}, \quad n, m \in \mathbb{N}.$$

vector space of  $n \times m$ -matrices: ( $n \triangleq$  number of rows,  $m \triangleq$  number of columns)

notation: **bold** CAPITAL roman letters, e.g.,  $\mathbf{A}, \mathbf{S}, \mathbf{Y}$

$$\mathbb{K}^{n,1} \leftrightarrow \text{column vectors}, \quad \mathbb{K}^{1,n} \leftrightarrow \text{row vectors}$$

\* Writing a matrix as a tuple of its columns or rows

$$\mathbf{c}_i \in \mathbb{K}^n, \quad i = 1, \dots, m \quad \triangleright \quad \mathbf{A} = [\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_m] \in \mathbb{K}^{n,m},$$

$$\mathbf{r}_i \in \mathbb{K}^m, \quad i = 1, \dots, n \quad \triangleright \quad \mathbf{A} = \begin{bmatrix} \mathbf{r}_1^\top \\ \vdots \\ \mathbf{r}_n^\top \end{bmatrix} \in \mathbb{K}^{n,m}.$$

\* Addressing matrix entries & sub-matrices (notation):

$$\mathbf{A} := \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix} \quad \begin{aligned} &\rightarrow \text{entry } (\mathbf{A})_{i,j} = a_{ij}, \quad 1 \leq i \leq n, 1 \leq j \leq m, \\ &\rightarrow \text{$i$-th row, } 1 \leq i \leq n: \quad a_{i,:} = (\mathbf{A})_{i,:}, \\ &\rightarrow \text{$j$-th column, } 1 \leq j \leq m: \quad a_{:,j} = (\mathbf{A})_{:,j}, \\ &\rightarrow \text{matrix block} \quad (a_{ij})_{\substack{i=k,\dots,l \\ j=r,\dots,s}} = (\mathbf{A})_{k:l,r:s}, \quad 1 \leq k \leq l \leq n, \\ &\quad \quad \quad \text{(sub-matrix)} \quad 1 \leq r \leq s \leq m. \end{aligned}$$

\* Transposed matrix:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}^\top := \begin{bmatrix} a_{11} & \dots & a_{n1} \\ \vdots & & \vdots \\ a_{1m} & \dots & a_{mn} \end{bmatrix} \in \mathbb{K}^{m,n}.$$

\* Adjoint matrix (Hermitian transposed):

$$\mathbf{A}^H := \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}^H := \begin{bmatrix} \bar{a}_{11} & \dots & \bar{a}_{n1} \\ \vdots & & \vdots \\ \bar{a}_{1m} & \dots & \bar{a}_{mn} \end{bmatrix} \in \mathbb{K}^{m,n}.$$

notation:  $\bar{a}_{ij} = \Re(a_{ij}) - i\Im(a_{ij})$  complex conjugate of  $a_{ij}$ .

## 1.1.2 Classes of matrices

Most matrices occurring in mathematical modelling have a special structure. This section presents a few of these. More will come up throughout the remainder of this chapter.

### (1.1.3) Special matrices

Terminology and notations for a few very special matrices:

Identity matrix:  $\mathbf{I} = \begin{bmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{bmatrix} \in \mathbb{K}^{n,n},$

Zero matrix:  $\mathbf{O} = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix} \in \mathbb{K}^{n,m}$ ,

Diagonal matrix:  $\mathbf{D} = \begin{bmatrix} d_1 & & 0 \\ & \ddots & \\ 0 & & d_n \end{bmatrix} \in \mathbb{K}^{n,n}, \quad d_j \in \mathbb{K}, \quad j = 1, \dots, n.$

### (1.1.4) Diagonal and triangular matrices

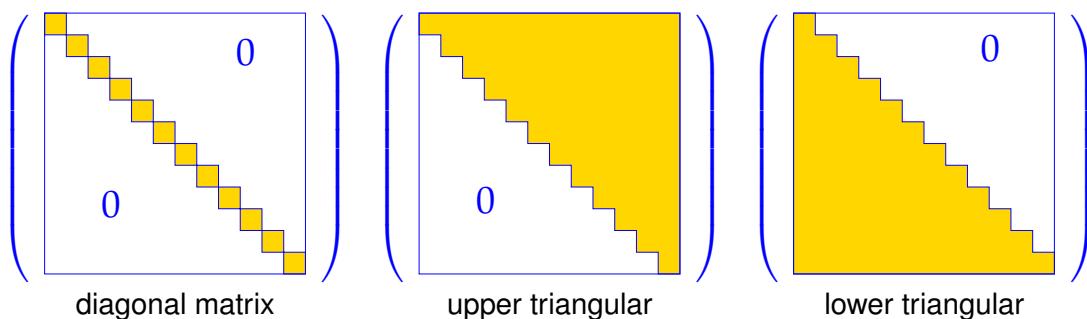
A little terminology to quickly refer to matrices whose non-zero entries occupy special locations:

#### Definition 1.1.5. Types of matrices

A matrix  $\mathbf{A} = (a_{ij}) \in \mathbb{R}^{m,n}$  is

- **diagonal matrix**, if  $a_{ij} = 0$  for  $i \neq j$ ,
- **upper triangular matrix** if  $a_{ij} = 0$  for  $i > j$ ,
- **lower triangular matrix** if  $a_{ij} = 0$  for  $i < j$ .

A triangular matrix is **normalized**, if  $a_{ii} = 1$ ,  $i = 1, \dots, \min\{m, n\}$ .



### (1.1.6) Symmetric matrices

#### Definition 1.1.7. Hermitian/symmetric matrices

A matrix  $\mathbf{M} \in \mathbb{K}^{n,n}$ ,  $n \in \mathbb{N}$ , is **Hermitian**, if  $\mathbf{M}^H = \mathbf{M}$ . If  $\mathbb{K} = \mathbb{R}$ , the matrix is called **symmetric**.

**Definition 1.1.8. Symmetric positive definite (s.p.d.) matrices** → [15, Def. 3.31], [63, Def. 1.22]

$\mathbf{M} \in \mathbb{K}^{n,n}$ ,  $n \in \mathbb{N}$ , is **symmetric (Hermitian) positive definite (s.p.d.)**, if

$$\mathbf{M} = \mathbf{M}^H \quad \text{and} \quad \forall \mathbf{x} \in \mathbb{K}^n: \quad \mathbf{x}^H \mathbf{M} \mathbf{x} > 0 \iff \mathbf{x} \neq 0.$$

If  $\mathbf{x}^H \mathbf{M} \mathbf{x} \geq 0$  for all  $\mathbf{x} \in \mathbb{K}^n$  ▷ **M** positive semi-definite.

**Lemma 1.1.9. Necessary conditions for s.p.d.** → [15, Satz 3.33], [63, Prop. 1.18]

For a symmetric/Hermitian positive definite matrix  $\mathbf{M} = \mathbf{M}^H \in \mathbb{K}^{n,n}$  holds true:

1.  $m_{ii} > 0, i = 1, \dots, n,$
2.  $m_{ii}m_{jj} - |m_{ij}|^2 > 0 \quad \forall 1 \leq i < j \leq n,$
3. all eigenvalues of  $\mathbf{M}$  are positive. ( $\leftarrow$  also sufficient for symmetric/Hermitian  $\mathbf{M}$ )

**Remark 1.1.10 (S.p.d. Hessians)**

Recall from analysis: in an isolated local minimum  $x^*$  of a  $C^2$ -function  $f : \mathbb{R}^n \mapsto \mathbb{R}$   $\rightarrow$  Hessian  $D^2 f(x^*)$  s.p.d. (see Def. 2.4.10 for the definition of the Hessian)

To compute the minimum of a  $C^2$ -function iteratively by means of Newton's method (→ Sect. 2.4) a linear system of equations with the s.p.d. Hessian as system matrix has to be solved in each step.

The solutions of many equations in science and engineering boils down to finding the minimum of some (energy, entropy, etc.) function, which accounts for the prominent role of s.p.d. linear systems in applications.

## 1.2 Software and Libraries

Whenever algorithms involve matrices and vectors (in the sense of linear algebra) it is advisable to rely on suitable code libraries or numerical programming environments.

### 1.2.1 MATLAB

**MATLAB** (“matrix laboratory”) is a commercial (sold by MathWorks Corp.)

- full fledged high level **programming language** designed for numerical algorithms,
- integrated development environment (**IDE**) offering editor, debugger, profiler, tracing facilities,
- rather comprehensive collection of **numerical libraries**.

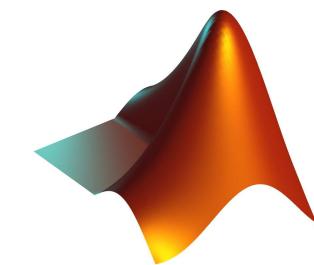


Fig. 1

In its basic form MATLAB is an interpreted scripting language without strict type-binding. This, together with its uniform IDE across many platforms, makes it a very popular tool for *rapid prototyping* and *testing* in CSE.

Plenty of resources are available for MATLAB's users, among them

- MATLAB documentation accessible through the Help menu or through this [link](#),
- MATLAB's help facility through the commands **help <function>** or **doc <function>**,
- A concise [MATLAB primer](#), one of many available online, see also [here](#)
- [MATLAB-Einführung](#) (in German) by P. Arbenz.

“In MATLAB everything is a matrix”

(Fundamental “data type” in MATLAB = **matrix** of complex numbers)

- In MATLAB vectors are represented as  $n \times 1$ -matrices (column vectors) or  $1 \times n$ -matrices (row vectors).

Note: The treatment of vectors as special matrices is consistent with the basic operations from matrix calculus.

### (1.2.1) Fetching the dimensions of a matrix

- ☞ `v = size(A)` yields a row vector `v` of length 2 with `v(1)` containing the number of rows and `v(2)` containing the number of columns of the matrix `A`.
- ☞ `numel(A)` returns the total number of entries of `A`; if `A` is a (row or column) vector, we get the length of `A`.

### (1.2.2) Access to matrix and vector components in MATLAB

Access (rvalue & lvalue) to components of a vector and entries of a matrix in MATLAB is possible through the `()`-operator:

- ☞ `r = v(i)`: retrieve `i`-th entry of vector `v`. `i` must be an integer and smaller or equal `numel(v)`.
- ☞ `r = A(i, j)`: get matrix entry  $(A)_{i,j}$  for two (valid) integer indices `i` and `j`.
- ☞ `r = A(end-1, end-2)`: get matrix entry  $(A)_{n-1,m-2}$  of an  $n \times m$ -matrix `A`.
- ! In case the matrix `A` is too small to contain an entry  $(A)_{i,j}$ , write access to `A(i, j)` will automatically trigger a dynamic adjustment of the matrix size to hold the accessed entry. The other new entries are filled with zeros.

```
Output: M=
% Caution: matrices are dynamically expanded when
% out of range entries are accessed
M = [1,2,3;4,5,6]; M(4,6) = 1.0; M,
      1   2   3   0   0   0
      4   5   6   0   0   0
      0   0   0   0   0   0
      0   0   0   0   0   1
```

- Danger of accidental exhaustion of memory!

### (1.2.3) Access to submatrices in MATLAB

For any two (row or column) vectors `I, J` of positive integers `A(I, J)` selects the submatrix

$$[(A)_{i,j}]_{\substack{i \in I \\ j \in J}} \in \mathbb{K}^{\#I, \#J}.$$

`A(I, J)` can be used as both r-value and l-value; in the former case the maximal components of `I` and `J` have to be smaller or equal the corresponding matrix dimensions, lest MATLAB issue the error message

Index exceeds matrix dimensions. In the latter case, the size of the matrix is grown, if needed, see § 1.2.2.

---

### (1.2.4) Initialization of matrices in MATLAB by concatenation

Inside square brackets [ ] the following two **matrix construction operators** can be used:

- , -operator  $\hat{=}$  adding another matrix to the right (horizontal concatenation)
  - ; -operator  $\hat{=}$  adding another matrix at the bottom (vertical concatenation)
- (The , -operator binds more strongly than the ; -operator!)

! Matrices joined by the , -operator must have the same number of rows.

Matrices concatenated vertically must have the same number of columns

▷ Filling a small matrix:  $A = [1, 2; 3, 4; 5, 6];$      $A = [[1; 3; 5], [2; 4; 6]]; \rightarrow 3 \times 2$  matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}.$$

▷ Initialization of vectors in MATLAB:

column vectors  $x = [1; 2; 3];$   
row vectors  $y = [1, 2, 3];$

▷ Building a matrix from blocks:

```
% MATLAB script demonstrating the construction of a matrix from blocks
A = [1, 2; 3, 4]; B = [5, 6; 7, 8];
C = [A, B; -B, A], % use concatenation
```

Output: C=

1	2	5	6
3	4	7	8
-5	-6	1	2
-7	-8	3	4

### (1.2.5) Special matrices in MATLAB

- ☞  $n \times n$  identity matrix:  $I = \text{eye}(n);$
  - ☞  $m \times n$  zero matrix:  $O = \text{zeros}(m, n);$
  - ☞  $m \times n$  random matrix with entries equidistributed in  $[0, 1]: R = \text{rand}(m, n);$
  - ☞  $n \times n$  diagonal matrix with components of  $n$ -vector  $d$  (both row or column vectors are possible) on its diagonal:  $D = \text{diag}(d);$
- 

### (1.2.6) Initialization of equispaced vectors (“loop index vectors”)

In MATLAB  $v = (a:s:b)$ , where  $a, b, s$  are real numbers, creates a *row vector* as initialised by the following code

```
if ((b >= a) && (s > 0))
    v = [a]; while (v(end)+s <= b), v = [v, v(end)+s]; end
elseif ((b <= a) && (s < 0))
    v = [a]; while (v(end)+s >= b), v = [v, v(end)+s]; end
```

```
else v = [];
end
```

Examples:

```
>> v = (3:-0.5:-0.3)
v = 3.0000 2.5000 2.0000 1.5000 1.0000 0.5000 0
>> v = (1:2.5:-13)
v = Empty matrix: 1-by-0
```

These vectors can be used to program loops in MATLAB

```
for i = (a:s:b)
% Do something with the loop variable i
end
```

In general we could also pass a matrix as “loop index vector”. In this case the loop variable will run through the *columns* of the matrix

```
% MATLAB loop over columns of a matrix
M = [1,2,3;4,5,6];
for i = M; i, end
```

Output:

i = 1	i = 2	i = 3
4	5	6

### (1.2.7) Special structural operations on matrices in MATLAB

- \*  $\mathbf{A}' \triangleq$  Hermitian transpose of a matrix  $\mathbf{A}$ , transposing without complex conjugation done by `transpose(A)`.
- \* `triu(A)` and `tril(A)` return the upper and lower **triangular parts** of a matrix  $\mathbf{A}$  as r-value (copy): If  $\mathbf{A} \in \mathbb{K}^{m,n}$

$$(\text{triu}(\mathbf{A}))_{i,j} = \begin{cases} (\mathbf{A})_{i,j} & , \text{if } i \leq j \\ 0 & \text{else.} \end{cases} \quad (\text{tril}(\mathbf{A}))_{i,j} = \begin{cases} (\mathbf{A})_{i,j} & , \text{if } i \geq j \\ 0 & \text{else.} \end{cases}$$

- \* `diag(A)` for a matrix  $\mathbf{A} \in \mathbb{K}^{m,n}$ ,  $\min\{m, n\} \geq 2$  returns the column vector  $[(\mathbf{A})_{i,i}]_{i=1,\min\{m,n\}} \in \mathbb{K}^{\min\{m,n\}}$ .

## 1.2.2 PYTHON

PYTHON is a widely used general-purpose and open source programming language. Together with the packages like NUMPY and MATPLOTLIB it delivers similar functionality like MATLAB for free. For interactive computing can IPYTHON can be used. All those packages belong to the SciPY ecosystem.

PYTHON features a good documentation and several scientific distributions are available (e.g. Anaconda, Enthought) which contain the most important packages. On most Linux-distributions the SciPy ecosystem is also available in the software repository, as well as many other packages including for example the Spyder IDE delivered with Anaconda.

A good introduction tutorial to numerical PYTHON are the [SciPy-lectures](#). The full documentation of NUMPY and SciPY can be found [here](#). For former MATLAB-users there's also a [guide](#). The scripts in this lecture notes follow the official [PYTHONstyle guide](#).

In PYTHON we have to import the numerical packages explicitly before use. This is normally done at the beginning of the file with lines like `import numpy as np` and `from matplotlib import pyplot as plt`. Those import statements are often skipped in this lecture notes to focus on the actual computations. But you can always assume the import statements as given here, e.g. `np.ravel(A)` is a call to a NUMPY function and `plt.loglog(x, y)` is a call to a MATPLOTLIB pyplot function.

### (1.2.8) Matrices and Vectors

The basic numeric data type in PYTHON are NUMPY's n-dimensional arrays. Vectors are normally implemented as 1D arrays and no distinction is made between row and column vectors. Matrices are represented as 2D arrays.

- ☞ `v = np.array([1, 2, 3])` creates a 1D array with the three elements 1, 2 and 3.
- ☞ `A = np.array([[1, 2], [3, 4]])` creates a 2D array.
- ☞ `A.shape` gives the n-dimensional size of an array.
- ☞ `A.size` gives the total number of entries in an array.

Note: There's also a matrix class in NUMPY with different semantics but its use is officially discouraged and it might even be removed in future release.

### (1.2.9) Manipulating arrays in PYTHON

There are many possibilities listed in the documentation how to [create](#), [index](#) and [manipulate](#) arrays.

An important difference to MATLAB is, that all arithmetic operations are normally performed element-wise, e.g. `A * B` is not the matrix-matrix product but element-wise multiplication (in MATLAB: `A.*A`). Also `A * v` does a [broadcasted](#) element-wise product. For the matrix product one has to use `np.dot(A, B)` or `A.dot(B)` explicitly.

## 1.2.3 EIGEN

Currently, the most widely used programming language for the development of new simulation software in scientific and industrial high-performance computing is C++. We are going to discuss [EIGEN](#) as an example for a C++ library for numerical linear algebra.

[EIGEN](#) is a [header-only](#) C++ template library designed to enable easy, natural and efficient numerical linear algebra: it provides data structures and a wide range of operations for matrices and vectors, see below. EIGEN also implements many more fundamental algorithms (see the [documentation page](#) or the discussion below).

EIGEN relies on [expression templates](#) to allow the efficient evaluation of complex expressions involving matrices and vectors. Refer to the [example](#) given in the EIGEN documentation for details.

- [Link](#) to an “EIGEN Cheat Sheet” (quick reference)

### (1.2.10) Compilation of codes using EIGEN

Compiling and linking on Mac OS X 10.10:

```
clang -D_HAS_CPP0X -std=c++11 -Wall -g \
-Wno-deprecated-register -DEIGEN3_ACTIVATED \
-I/opt/local/include -I/usr/local/include/eigen3 \
-o main.cpp.o -c main.cpp
/usr/bin/c++ -std=c++11 -Wall -g -Wno-deprecated-register \
-DEIGEN3_ACTIVATED -Wl,-search_paths_first \
-Wl,-headerpad_max_install_names main.cpp.o \
-o executable /opt/local/lib/libboost_program_options-mt.dylib
```

Of course, different compilers may be used on different platforms. In all cases, basic EIGEN functionality can be used without linking with a special library.

### (1.2.11) Matrix and vector data types in EIGEN

A generic matrix data type is given by the templated class

```
Matrix<typename Scalar, int RowsAtCompileTime, int
ColsAtCompileTime>
```

Here **Scalar** is the underlying scalar type of the matrix entries, which must support the usual operations '+', '-', '\*', '/', and '+=' , '\*'= , '~= ', etc. Usually the scalar type will be either **double**, **float**, or **complex<>**. The cardinal template arguments **RowsAtCompileTime** and **ColsAtCompileTime** can pass a **fixed** size of the matrix, if it is known at compile time. There is a specialization selected by the template argument **Eigen::Dynamic** supporting variable size "dynamic" matrices.

```
#include <Eigen/Dense >

template<typename Scalar>
void eigenTypeDemo(unsigned int dim)
{
    using dynMat_t =
        Eigen::Matrix<Scalar,Eigen::Dynamic,Eigen::Dynamic>;
    using dynColVec_t = Eigen::Matrix<Scalar,Eigen::Dynamic,1>;
    using dynRowVec_t = Eigen::Matrix<Scalar,1,Eigen::Dynamic>;
    using index_t = typename dynMat_t::Index;
    using entry_t = typename dynMat_t::Scalar;

    dynColVec_t colvec(dim);
    dynRowVec_t rowvec(dim);
    for (index_t i=0; i< colvec.size(); ++i)
        colvec(i) = (Scalar)i;
    for (index_t i=0; i< rowvec.size(); ++i)
        rowvec(i) = (Scalar)1/(i+1);
    dynMat_t vecprod = colvec*rowvec;
    const int nrows = vecprod.rows();
    const int ncols = vecprod.cols();
}
```

The following convenience data type are provided by EIGEN, see [documentation](#):

- **MatrixXd**  $\hat{=}$  generic variable size matrix with **double** precision entries
- **VectorXd, RowVectorXd**  $\hat{=}$  dynamic column and row vectors  
(= dynamic matrices with one dimension equal to 1)
- **MatrixNd** with  $N = 2, 3, 4$  for small fixed size square  $N \times N$ -matrices (type **double**)
- **VectorNd** with  $N = 2, 3, 4$  for small column vectors with fixed length  $N$ .

The **d** in the type name may be replaced with **i** (for **int**), **f** (for **float**), and **cd** (for **complex<double>**) to select another basic scalar type.

All matrix type feature the methods **cols()**, **rows()**, and **size()** telling the number of columns, rows, and total number of entries.

### (1.2.12) Initialization of **dense** matrices in EIGEN

The entry access operator (**int i, int j**) allows the most direct setting of matrix entries; there is hardly any runtime penalty.

Special static initialisation methods:

```
#include <Eigen/Dense >
// Just allocate space for matrix, no initialisation
Eigen::MatrixXd A(rows,cols);
// Zero matrix. Similar to matlab command zeros(rows,cols);
Eigen::MatrixXd B = MatrixXd::Zero(rows, cols);
// Ones matrix. Similar to matlab command ones(rows,cols);
Eigen::MatrixXd C = MatrixXd::Ones(rows, cols);
// Matrix with all entries same as value.
Eigen::MatrixXd D = MatrixXd::Constant(rows, cols, value);
// Random matrix, entries uniformly distributed in [0,1]
Eigen::MatrixXd E = MatrixXd::Random(rows, cols);
// (Generalized) identity matrix, 1 on main diagonal
Eigen::MatrixXd I = MatrixXd::Identity(rows,cols);
std::cout << "size of A = (" << A.rows() << ',' << A.cols() << ')'
<< std::endl;
```

A versatile way to initialize a matrix relies on a combination of the operators `<<` and `,`, which allows the construction of a matrix from blocks:

```
MatrixXd mat3(6,6);
mat3 <<
    MatrixXd::Constant(4,2,1.5), // top row, first block
    MatrixXd::Constant(4,3,3.5), // top row, second block
    MatrixXd::Constant(4,1,7.5), // top row, third block
    MatrixXd::Constant(2,4,2.5), // bottom row, left block
    MatrixXd::Constant(2,2,4.5); // bottom row, right block
```

The matrix is filled top to bottom left to right, block dimensions have to match (like in MATLAB).

### (1.2.13) Access to submatrices in EIGEN ( $\rightarrow$ [documentation](#))

The method `block(int i, int j, int p, int q)` returns a reference to the submatrix with upper left corner at position  $(i,j)$  and size  $p \times q$ .

The methods `row(int i)` and `col(int j)` provide a reference to the corresponding row and column of the matrix. Even more specialised access methods are `topLeftCorner(p, q)`, `bottomLeftCorner(p, q)`, `topRightCorner(p, q)`, `ottomRightCorner(p, q)`, `topRows(q)`, `bottomRows(q)`, `leftCols(p)` and `rightCols(q)` with obvious meanings.

#### C++11-code 1.2.14: Demonstration code for access to matrix blocks in EIGEN

```

1  template<typename MatType> void
2      blockAccess(Eigen::MatrixBase<MatType> &M)
3  {
4      using index_t = typename Eigen::MatrixBase<MatType>::Index;
5      using entry_t = typename Eigen::MatrixBase<MatType>::Scalar;
6      const index_t nrows(M.rows()); // No. of rows
7      const index_t ncols(M.cols()); // No. of columns
8
8      cout << "Matrix M = " endl << M << endl; // Print matrix
9      // Block size half the size of the matrix
10     index_t p = nrows/2, q = ncols/2;
11     // Output submatrix with left upper entry at position (i,i)
12     for(index_t i=0; i < min(p,q); i++)
13         cout << "Block (" << i << ',' << i << ',' << p << ',' << q
14         << ") = " << M.block(i,i,p,q) << endl;
15     // l-value access: modify sub-matrix by adding a constant
16     M.block(1,1,p,q) += MatrixXd::Constant(p,q,1.0);
17     cout << "M = " endl << M << endl;
18     // r-value access: extract sub-matrix
19     MatrixXd B(M.block(1,1,p,q));
20     cout << "Isolated modified block = " endl << B << endl;
21     // Special sub-matrices
22     cout << p << " top rows of m = " << M.topRows(p) << endl;
23     cout << p << " bottom rows of m = " << M.bottomRows(p) << endl;
24     cout << q << " left cols of m = " << M.leftCols(q) << endl;
25     cout << q << " right cols of m = " << M.rightCols(p) << endl;
26     // r-value access to upper triangular part
27     const MatrixXd T = M.template triangularView<Upper>(); //
28     cout << "Upper triangular part = " << endl << T << endl;
29     // l-value access to upper triangular part
30     M.template triangularView<Lower>() *= -1.5; //
31     cout << "Matrix M = " << endl << M << endl;
32 }
```

EIGEN offers `views` for access to triangular parts of a matrix, see Line 27 and Line 30, according to

`M.triangularView<XX>()`

where `XX` can stand for one of the following: `Upper`, `Lower`, `StrictlyUpper`, `StrictlyLower`,

`UnitUpper`, `UnitLower`, see [documentation](#).

For column and row vectors references to sub-vectors can be obtained by the methods `head(int length)`, `tail(int length)`, and `segment(int pos, int length)`.

Note: Unless the preprocessor switch `NDEBUG` is set, EIGEN performs range checks on all indices.

### (1.2.15) Componentwise operations in EIGEN

Since operators like MATLAB's `.*` are not available, EIGEN uses the [Array concept](#) to furnish entry-wise operations on matrices. An EIGEN-Array contains the same data as a matrix, supports the same methods for initialisation and access, but replaces the operators of matrix arithmetic with entry-wise actions. Matrices and arrays can be converted into each other by the `array()` and `matrix()` methods, see [documentation](#) for details.

#### C++11-code 1.2.16: Using Array in EIGEN

```

1 void matArray(int nrows, int ncols)
2 {
3     Eigen::MatrixXd m1(nrows, ncols), m2(nrows, ncols);
4     for(int i = 0; i < m1.rows(); i++)
5         for(int j=0; j < m1.cols(); j++) {
6             m1(i, j) = (double)(i+1)/(j+1);
7             m2(i, j) = (double)(j+1)/(i+1);
8         }
9     // Entry-wise product, not a matrix product
10    Eigen::MatrixXd m3 = (m1.array() * m2.array()).matrix();
11    // Explicit entry-wise operations on matrices are possible
12    Eigen::MatrixXd m4(m1.cwiseProduct(m2));
13    // Entry-wise logarithm
14    cout << "Log(m1) = " << endl << log(m1.array()) << endl;
15    // Entry-wise boolean expression, true cases counted
16    cout << (m1.array() > 3).count() << " entries of m1 > 3" << endl;
17 }
```

The application of a [functor](#) (→ Section 0.2.3) to all entries of a matrix can also been done via the `unaryExpr()` method of a matrix:

```
// Apply a lambda function to all entries of a matrix
auto fnct = [] (double x) { return (x+1.0/x); };
cout << "f(m1) = " << endl << m1.unaryExpr(fnct) << endl;
```

### Remark 1.2.17 (EIGEN in use)

- ☞ EIGEN is used as one of the base libraries for the [Robot Operating System](#) (ROS), an open source project with strong ETH participation.

- ☞ The geometry processing library [libigl](#) uses EIGEN as its basic linear algebra engine. At ETH it is being used and developed at ETH Zurich, at the [Interactive Geometry Lab](#) and [Advanced Technologies Lab](#).

### 1.2.4 Matrix storage formats

All numerical libraries store the entries of a (generic = *dense*) matrix  $\mathbf{A} \in \mathbb{K}^{m,n}$  in a [linear](#) array of length  $mn$  (or longer). Accessing entries entails suitable index computations.

Two natural options for “vectorisation” of a matrix: *row major*, *column major*

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Row major (C-arrays, bitmaps, Python):

A_arr	1	2	3	4	5	6	7	8	9
-------	---	---	---	---	---	---	---	---	---

Column major (Fortran, MATLAB, EIGEN):

A_arr	1	4	7	2	5	8	3	6	9
-------	---	---	---	---	---	---	---	---	---

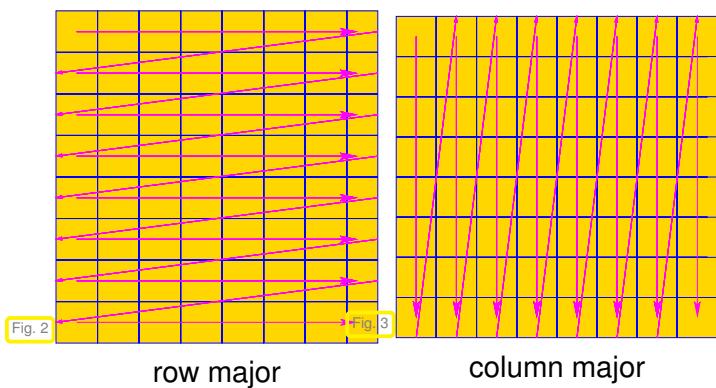
Access to entry  $(\mathbf{A})_{ij}$  of  $\mathbf{A} \in \mathbb{K}^{n,m}$ ,  
 $i = 1, \dots, n, j = 1, \dots, m$ :

row major:

$$(\mathbf{A})_{ij} \leftrightarrow A\_arr(m*(i-1)+(j-1))$$

column major:

$$(\mathbf{A})_{ij} \leftrightarrow A\_arr(n*(j-1)+(i-1))$$



#### Example 1.2.18 (Accessing matrix data as a vector)

Both in MATLAB and EIGEN the single index access operator relies on the linear data layout:

```
A = [1 2 3; 4 5 6; 7 8 9]; A(:)',
```

produces the terminal output

```
1      4      7      2      5      8      3      6      9
```

which clearly reveals the *column major* storage format.

In PYTHON the default data layout is row major, but it can be explicitly set. Further, array transposition does not change any data, but only the memory order and array shape.

#### PYTHON-code 1.2.19: Storage order in PYTHON

```
1 # array creation
2 A = np.array([[1, 2], [3, 4]]) # default (row major) storage
3 B = np.array([[1, 2], [3, 4]], order='F') # column major storage
4
5 # show internal storage
6 np.ravel(A, 'K') # array elements as stored in memory: [1, 2, 3, 4]
```

```

7 | np.ravel(B, 'K')    # array elements as stored in memory: [1, 3, 2, 4]
8 |
9 | # nothing happens to the data on transpose, just the storage order
10| changes
10| np.ravel(A.T, 'K')   # array elements as stored in memory: [1, 2, 3,
11| 4]
11| np.ravel(B.T, 'K')   # array elements as stored in memory: [1, 3, 2,
12| 4]
12|
13| # storage order can be accessed by checking the array's flags
14| A.flags['C_CONTIGUOUS'] # True
15| B.flags['F_CONTIGUOUS'] # True
16| A.T.flags['F_CONTIGUOUS'] # True
17| B.T.flags['C_CONTIGUOUS'] # True

```

In EIGEN the data layout can be controlled by a template argument; default is column major.

#### C++11-code 1.2.20: Single index access of matrix entries in EIGEN

```

1 void storageOrder(int nrows=6,int ncols=7)
2 {
3     cout << "Different matrix storage layouts in Eigen" << endl;
4     // Template parameter ColMajor selects column major data layout
5     Matrix<double,Dynamic,Dynamic,ColMajor> mcm(nrows,ncols);
6     // Template parameter RowMajor selects row major data layout
7     Matrix<double,Dynamic,Dynamic,RowMajor> mrm(nrows,ncols);
8     // Direct initialization; lazy option: use int as index type
9     for (int l=1,i= 0; i< nrows; i++)
10        for (int j= 0; j< ncols; j++,l++)
11            mcm(i,j) = mrm(i,j) = l;
12
13     cout << "Matrix mrm = " << endl << mrm << endl;
14     cout << "mcm linear = ";
15     for (int l=0;l < mcm.size(); l++) cout << mcm(l) << ',';
16     cout << endl;
17
18     cout << "mrm linear = ";
19     for (int l=0;l < mrm.size(); l++) cout << mrm(l) << ',';
20     cout << endl;
21 }

```

The function call `storageOrder(3, 3)`, cf. Code 1.2.20 yields the output

```

Different matrix storage layouts in Eigen
Matrix mrm =
1 2 3
4 5 6
7 8 9
mcm linear = 1,4,7,2,5,8,3,6,9,
mrm linear = 1,2,3,4,5,6,7,8,9,

```

**Remark 1.2.21 (MATLAB command `reshape`)**

matlab offers the built-in command `reshape` for changing the dimensions of a matrix  $\mathbf{A} \in \mathbb{K}^{m,n}$ :

```
B = reshape(k, l, A); % error, in case kl ≠ mn
```

This command will create an  $k \times l$ -matrix by just reinterpreting the linear array of entries of  $\mathbf{A}$  as data for a matrix with  $k$  rows and  $l$  columns. Regardless of the size and entries of the matrices the following test will always produce a `equal = true` result

```
if ((prod(size(A)) ~= (k*l)), error('Size mismatch'); end
B = reshape(A, k, l);
equal = (B(:) == A(:));
```

**Remark 1.2.22 (NumPy function `reshape`)**

NumPy offers the function `np.reshape` for changing the dimensions of a matrix  $\mathbf{A} \in \mathbb{K}^{m,n}$ :

```
# read elements of A in row major order (default)
B = np.reshape(A, (k, l)) # error, in case kl ≠ mn
B = np.reshape(A, (k, l), order='C') # same as above
# read elements of A in column major order
B = np.reshape(A, (k, l), order='F')
# read elements of A as stored in memory
B = np.reshape(A, (k, l), order='A')
```

This command will create an  $k \times l$ -array by reinterpreting the array of entries of  $\mathbf{A}$  as data for an array with  $k$  rows and  $l$  columns. The order in which the elements of  $\mathbf{A}$  are read can be set by the `order` argument to row major (default, '`C`'), column major ('`F`') or  $\mathbf{A}$ 's internal storage order, i.e. row major if  $\mathbf{A}$  is row major or column major if  $\mathbf{A}$  is column major ('`A`').

**Remark 1.2.23 (Reshaping matrices in EIGEN)**

If you need a reshaped view of a matrix' data in EIGEN you can obtain it via the raw data vector belonging to the matrix. Then use this information to create a matrix view by means of `Map`.

**C++11-code 1.2.24: Demonstration on how reshape a matrix in EIGEN**

```
1 template<typename MatType>
2 void reshapatetest(MatType &M)
3 {
4     using index_t = typename MatType::Index;
5     using entry_t = typename MatType::Scalar;
6     const index_t nsize(M.size());
7
8     // reshaping possible only for matrices with an even number of
9     // entries
10    if ((nsize %2) == 0) {
```

```

10   entry_t *Mdat = M.data();
11   Map<Eigen::Matrix<entry_t, Dynamic, Dynamic>> R(Mdat, 2, nsize/2);
12   cout << "Matrix M = " << endl << M << endl;
13   cout << "reshaped to " << R.rows() << 'x' << R.cols()
14     << " = " << endl << R << endl;
15   // Modifying R affects M, because they share the data space !
16   R *= -1.5;
17   cout << "Matrix M = " << endl << M << endl;
18 }
19 }
```

This function has to be called with an l-value of a matrix type object.

### Experiment 1.2.25 (Impact of matrix data access patterns on runtime)

Modern CPU feature several levels of memories (registers, L1 cache, L2 cache, ..., main memory) of different latency, bandwidth, and size. Frequently accessing memory locations with widely different addresses results in many cache misses and will considerably slow down the CPU.

<pre>A = <b>randn</b>(n,n); <b>for</b> j = 1:n-1,     A(:,j+1) = A(:,j+1) - A(:,j); <b>end</b></pre>	<pre>A = <b>randn</b>(n,n); <b>for</b> i = 1:n-1,     A(i+1,:) = A(i+1,:) - A(i,:); <b>end</b></pre>
column oriented access	row oriented access

### MATLAB-code 1.2.26: Timing for row and column oriented matrix access in MATLAB

```

1 % Timing for row/column operations on matrices
2 % We conduct K runs in order to reduce the risk of skewed measurements
3 % due to OS activity during MATLAB run.
4 K = 3; res = [];
5 for n=2.^4:13
6   A = randn(n,n);
7
8   t1 = realmax;
9   for k=1:K, tic;
10    for j = 1:n-1, A(:,j+1) = A(:,j+1) - A(:,j); end;
11    t1 = min(toc,t1);
12  end
13  t2 = realmax;
14  for k=1:K, tic;
15    for i = 1:n-1, A(i+1,:) = A(i+1,:) - A(i,:); end;
16    t2 = min(toc,t2);
17  end
18  res = [res; n, t1 , t2];
19 end
20
21 % Plot runtimes versus matrix sizes
```

```

22 figure; plot(res(:,1),res(:,2),'r+', res(:,1),res(:,3),'m*');
23 xlabel('{\bf n}', 'fontsize',14);
24 ylabel('{\bf runtime [s]}', 'fontsize',14);
25 legend('A(:,j+1) = A(:,j+1) - A(:,j)', 'A(i+1,:) = A(i+1,:) -
26     A(i,:)', ...
27         'location','northwest');
28 print -depsc2 '../PICTURES/accessrtlin.eps';
29
30 figure; loglog(res(:,1),res(:,2),'r+', res(:,1),res(:,3),'m*');
31 xlabel('{\bf n}', 'fontsize',14);
32 ylabel('{\bf runtime [s]}', 'fontsize',14);
33 legend('A(:,j+1) = A(:,j+1) - A(:,j)', 'A(i+1,:) = A(i+1,:) -
34     A(i,:)', ...
35         'location','northwest');
36 print -depsc2 '../PICTURES/accessrtlog.eps';

```

### PYTHON-code 1.2.27: Timing for row and column oriented matrix access in PYTHON

```

1 import numpy as np
2 import timeit
3 from matplotlib import pyplot as plt
4
5 def col_wise(A):
6     for j in range(A.shape[1] - 1):
7         A[:, j + 1] -= A[:, j]
8
9 def row_wise(A):
10    for i in range(A.shape[0] - 1):
11        A[i + 1, :] -= A[i, :]
12
13 # Timing for row/column-wise operations on matrix, we conduct k runs in
14 # order
15 # to reduce risk of skewed measurements due to OS activity during run.
16
17 k = 3
18 res = []
19 for n in 2**np.mgrid[4:14]:
20     A = np.random.normal(size=(n, n))
21
22     t1 = min(timeit.repeat(lambda: col_wise(A), repeat=k,
23                           number=1))
24     t2 = min(timeit.repeat(lambda: row_wise(A), repeat=k,
25                           number=1))
26
27     res.append((n, t1, t2))
28
29 # plot runtime versus matrix sizes
30 ns, t1s, t2s = np.transpose(res)
31
32 plt.figure()

```

```

30 plt.plot(ns, t1s, '+', label='A[:, j + 1] -= A[:, j]')
31 plt.plot(ns, t2s, 'o', label='A[i + 1, :] -= A[i, :]')
32 plt.xlabel(r'n')
33 plt.ylabel(r'runtime [s]')
34 plt.legend(loc='upper left')
35 plt.savefig('../PYTHON_PICTURES/accessrtlin.eps')
36
37 plt.figure()
38 plt.loglog(ns, t1s, '+', label='A[:, j + 1] -= A[:, j]')
39 plt.loglog(ns, t2s, 'o', label='A[i + 1, :] -= A[i, :]')
40 plt.xlabel(r'n')
41 plt.ylabel(r'runtime [s]')
42 plt.legend(loc='upper left')
43 plt.savefig('../PYTHON_PICTURES/accessrtlog.eps')
44
45 plt.show()

```

### C++11 code 1.2.28: Timing for row and column oriented matrix access for EIGEN

```

1 #define NDEBUG true
2 #include <Eigen/Dense>
3 #include <iostream>
4 #include <chrono>
5 using namespace std::chrono;
6
7 /** Timing of row and column oriented access for Eigen */
8 void rowcolaccesstiming(void)
9 {
10    const int K = 3; // Number of repetitions
11    const int N_min = 4; // Smalles matrix size 16
12    const int N_max = 13; // Scan until matrix size of 8192
13    unsigned long n = (1L << N_min);
14
15    for(int l=4; l<= N_max; l++, n*=2) {
16        Eigen::MatrixXd A = Eigen::MatrixXd::Random(n,n);
17        double t1 = 1000.0;
18        for(int k=0;k<K;k++) {
19            auto tic = high_resolution_clock::now();
20            for(int j=0; j < n-1; j++) A.row(j+1) -= A.row(j); // row access
21            auto toc = high_resolution_clock::now();
22            double t =
23                (double)duration_cast<microseconds>(toc-tic).count() /1E6;
24            t1 = std::min(t1,t);
25        }
26        double t2 = 1000.0;
27        for(int k=0;k<K;k++) {
28            auto tic = high_resolution_clock::now();
29            for(int j=0; j < n-1; j++) A.col(j+1) -= A.col(j); //column
access
            auto toc = high_resolution_clock::now();

```

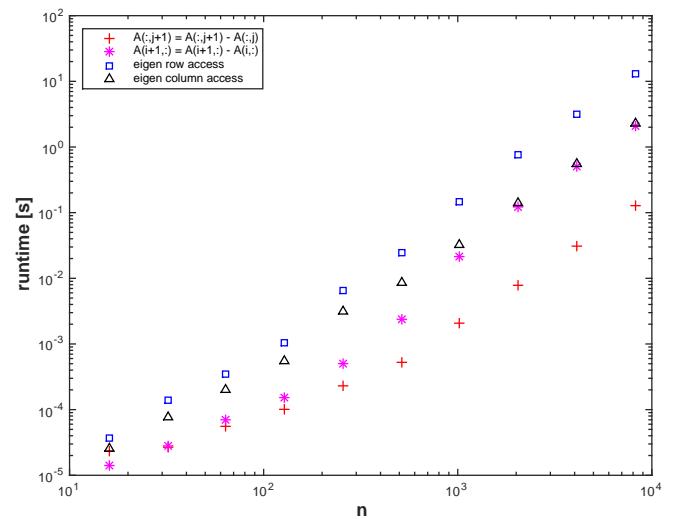
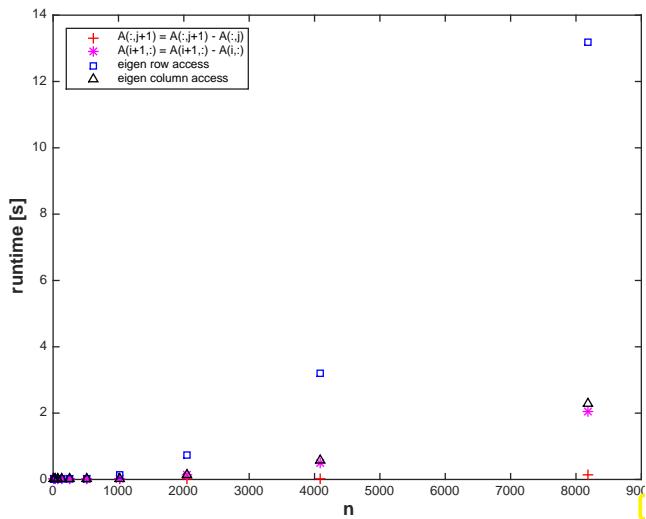
```

30     double t =
31         (double)duration_cast<microseconds>(toc-tic).count()/1E6;
32     t2 = std::min(t2,t);
33 }
34 std::cout << "n = " << n << ", t(row) = " << t1 << ", t(col) " <<
35     t2 << std::endl;
36 }
37 // Overhead of returning by value
38 // First version return by reference
39 void retMatRef(const Eigen::VectorXd &v, Eigen::MatrixXd &R)
40 {
41     using index_t = typename Eigen::VectorXd::Index;
42     const index_t n = v.size();
43     R = (v*v.transpose() + Eigen::MatrixXd::Identity(n,n));
44 }
45
46 Eigen::MatrixXd retMatVal(const Eigen::VectorXd &v)
47 {
48     using index_t = typename Eigen::VectorXd::Index;
49     const index_t n = v.size();
50     return (v*v.transpose() + Eigen::MatrixXd::Identity(n,n));
51 }
52
53 void retmattiming(void)
54 {
55     std::cout << "Timing return by reference vs. return by value" <<
56         std::endl;
57     const int K = 3; // Number of repetitions
58     const int N_min = 4; // Smalles matrix size 16
59     const int N_max = 13; // Scan until matrix size of 8192
60     Eigen::MatrixXd res(N_max-N_min+1,4);
61     unsigned long n = (1L << N_min);
62
63     for(int l=N_min; l<= N_max; l++, n*=2) {
64         Eigen::VectorXd v = Eigen::VectorXd::LinSpaced(n,0.0,1.0);
65         Eigen::MatrixXd M(n,n);
66         double t1 = 10000.0;
67         for(int k=0;k<K;k++) {
68             auto tic = high_resolution_clock::now();
69             retMatRef(v,M); M(n-1,n-1) += k;
70             auto toc = high_resolution_clock::now();
71             double t =
72                 (double)duration_cast<microseconds>(toc-tic).count()/1E6;
73             t1 = std::min(t1,t);
74             v *= 1.5;
75         }
76         double t2 = 10000.0;
77         for(int k=0;k<K;k++) {
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175

```

```

76     auto tic = high_resolution_clock::now();
77     M = retMatVal(v); M(n-1,n-1) += k;
78     auto toc = high_resolution_clock::now();
79     double t =
80         (double)duration_cast<microseconds>(toc-tic).count()/1E6;
81     t2 = std::min(t2,t);
82     v *= 1.5;
83 }
84 std::cout << "n = " << n << ", t(Ref) = " << t1 << ", t(Val) " <<
85     t2 << std::endl;
86 res(I-N_min,0) = n; res(I-N_min,1) = t1; res(I-N_min,2) = t2;
87     res(I-N_min,3) = t1/t2;
88 }
89 std::cout << "n t(Ref) t(val) tr/tv" << std::endl << res <<
90     std::endl;
91 }
92
93 int main(int argc,char **argv) {
94     if (argc != 2) {
95         std::cerr << "Usage: " << argv[0] << " <selection>" << std::endl;
96         return(-1L);
97     }
98     else {
99         const int sel = atoi(argv[1]);
100        switch (sel) {
101            case 1: { rowcolaccesstiming(); break; }
102            case 2: { retmattiming(); break; }
103            default: { std::cerr << "Invalid selection" << std::endl;
104                         exit(-1L); }
105        }
106    }
107    return 0;
108 }
```



(Intel Core i7, 2.66 GHz, MacOS X 1.10, MATLAB 8.5.0 (R2015a), Apple LLVM version 6.1.0)

We observe a glaring discrepancy of CPU time required for accessing entries of a matrix in rowwise or columnwise fashion. This reflects the impact of features of the underlying hardware architecture, like cache size and memory bandwidth:

Interpretation of timings: Since matrices in MATLAB are stored column major all the matrix elements in a column occupy contiguous memory locations, which will all reside in the cache together. Hence, column oriented access will mainly operate on data in the cache even for large matrices. Conversely, row oriented access addresses matrix entries that are stored in distant memory locations, which incurs frequent cache misses (**cache thrashing**).

The impact of hardware architecture on the performance of algorithms will **not** be taken into account in this course, because hardware features tend to be both intricate and ephemeral. However, for modern high performance computing it is essential to adapt implementations to the hardware on which the code is supposed to run.

We also observe that the EIGEN implementation is significantly slower than the MATLAB code!

---

## 1.3 Basic linear algebra operations

First we refresh the basic rules of vector and matrix calculus. Then we will learn about a very old programming interface for simple dense linear algebra operations.

### 1.3.1 Elementary matrix-vector calculus

What you should know from linear algebra:

\* vector space operations in matrix space  $\mathbb{K}^{m,n}$  (addition, multiplication with scalars)



dot product:  $\mathbf{x}, \mathbf{y} \in \mathbb{K}^n, n \in \mathbb{N}$ :  $\mathbf{x} \cdot \mathbf{y} := \mathbf{x}^H \mathbf{y} = \sum_{i=1}^n x_i y_i \in \mathbb{K}$   
(in MATLAB: `dot(x, y)`)



tensor product:  $\mathbf{x} \in \mathbb{K}^m, \mathbf{y} \in \mathbb{K}^n, n \in \mathbb{N}$ :  $\mathbf{x}\mathbf{y}^H = (x_i y_j)_{\substack{i=1, \dots, m \\ j=1, \dots, n}} \in \mathbb{K}^{m,n}$

\* All are special cases of the **matrix product**:

$$\mathbf{A} \in \mathbb{K}^{m,n}, \quad \mathbf{B} \in \mathbb{K}^{n,k}: \quad \mathbf{AB} = \left[ \sum_{j=1}^n a_{ij} b_{jl} \right]_{\substack{i=1, \dots, m \\ l=1, \dots, k}} \in \mathbb{K}^{m,k}. \quad (1.3.1)$$

Recall from linear algebra basic properties of the matrix product: for all  $\mathbb{K}$ -matrices  $\mathbf{A}, \mathbf{B}, \mathbf{C}$  (of suitable sizes),  $\alpha, \beta \in \mathbb{K}$

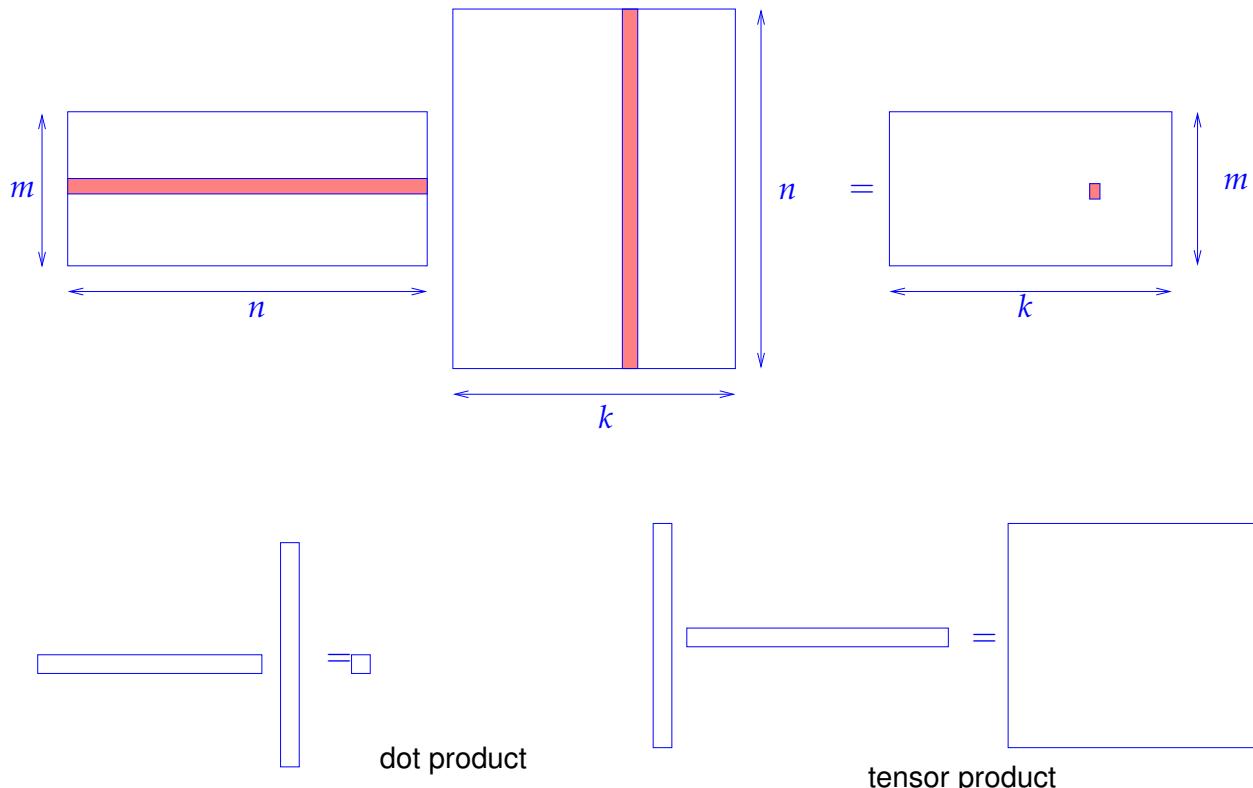
$$\text{associative: } (\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC}),$$

$$\text{bi-linear: } (\alpha\mathbf{A} + \beta\mathbf{B})\mathbf{C} = \alpha(\mathbf{AC}) + \beta(\mathbf{BC}), \quad \mathbf{C}(\alpha\mathbf{A} + \beta\mathbf{B}) = \alpha(\mathbf{CA}) + \beta(\mathbf{CB}),$$

non-commutative:  $\mathbf{AB} \neq \mathbf{BA}$  in general.

### (1.3.2) Visualisation of special matrix products

Dependency of an entry of a product matrix:



### Remark 1.3.3 (Row-wise & column-wise view of matrix product)

To understand what is going on when forming a matrix product, it is often useful to decompose it into matrix  $\times$  vector operations in one of the following two ways:

$\mathbf{A} \in \mathbb{K}^{m,n}$ ,  $\mathbf{B} \in \mathbb{K}^{n,k}$ :

$$\mathbf{AB} = \begin{bmatrix} \mathbf{A}(\mathbf{B})_{:,1} & \dots & \mathbf{A}(\mathbf{B})_{:,k} \end{bmatrix}, \quad \mathbf{AB} = \begin{bmatrix} (\mathbf{A})_{1,:}\mathbf{B} \\ \vdots \\ (\mathbf{A})_{m,:}\mathbf{B} \end{bmatrix}. \quad (1.3.4)$$

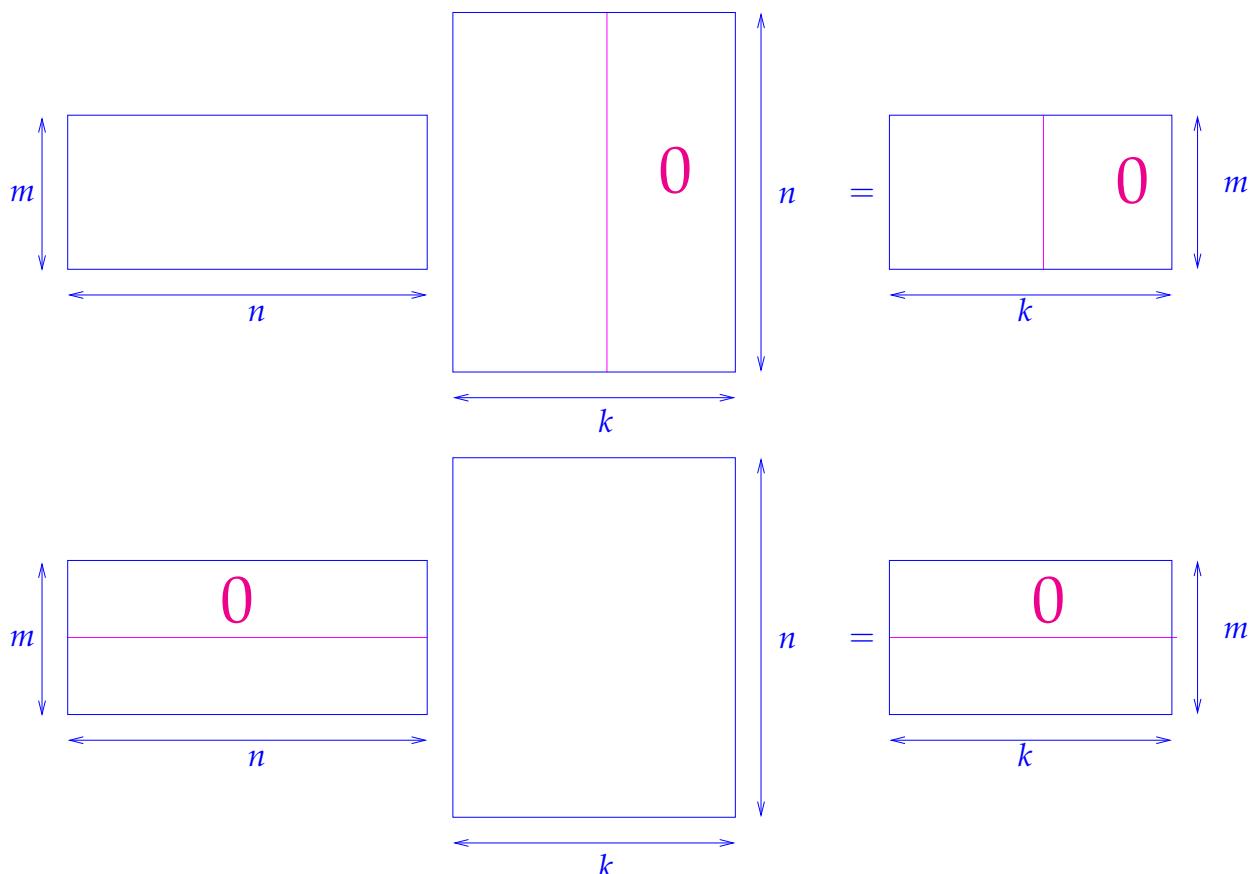
↓                              ↓  
matrix assembled from columns    matrix assembled from rows

For notations refer to Sect. 1.1.1.

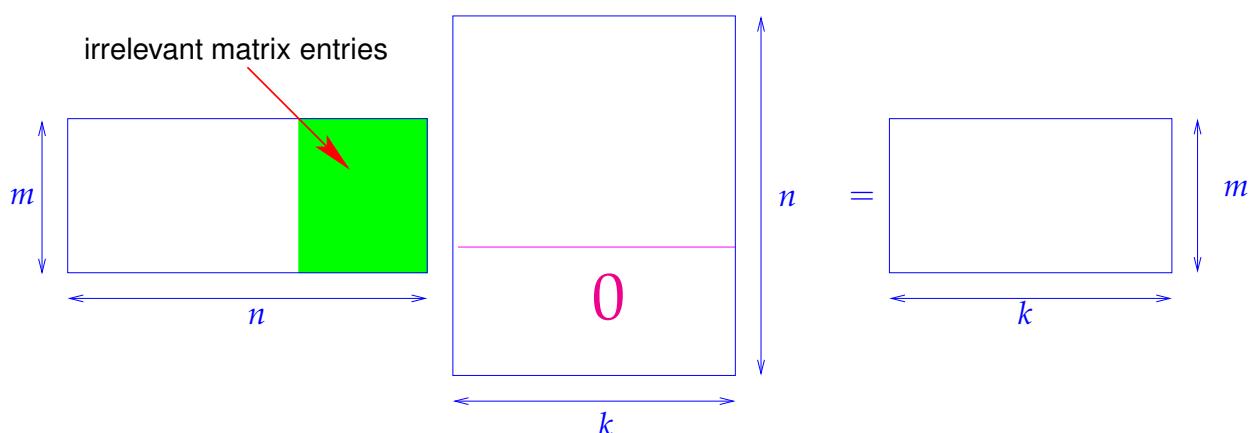
**Remark 1.3.5 (Understanding the structure of product matrices)**

A “mental image” of matrix multiplication is useful for telling special properties of product matrices.

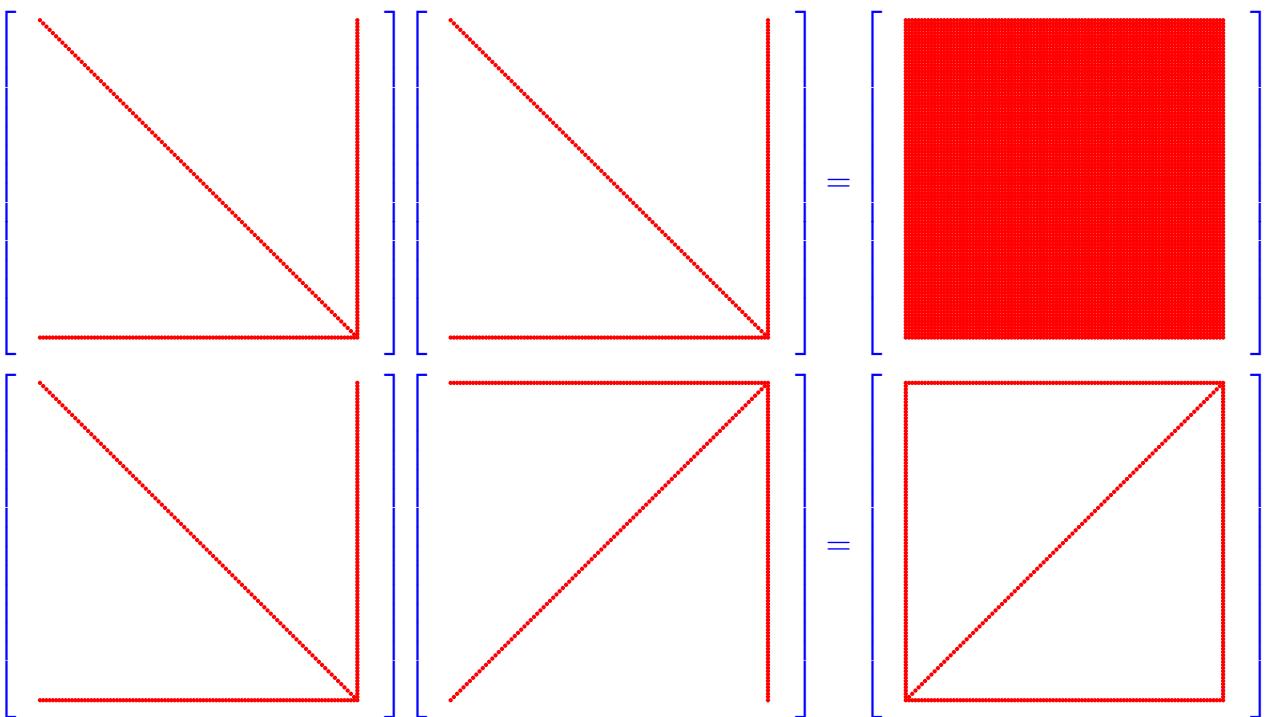
For instance, zero blocks of the product matrix can be predicted easily in the following situations using the idea explained in Rem. 1.3.3 (try to understand how):



“Seeing”, which parts of a matrix factor matter in a product:



“Seeing” the structure/pattern of a matrix product:



MATLAB-command for visualizing the structure of a matrix: `spy (M)`

#### MATLAB-code 1.3.6: Visualizing the structure of matrices in MATLAB

```

1 n = 100; A = [diag(1:n-1), (1:n-1)'; (1:n) ]; B = A(n:-1:1,:);
2 C = A*A; D = A*B;
3 figure; spy(A,'r'); axis off; print -depsc2
  '../PICTURES/Aspy.eps';
4 figure; spy(B,'r'); axis off; print -depsc2
  '../PICTURES/Bspy.eps';
5 figure; spy(C,'r'); axis off; print -depsc2
  '../PICTURES/Cspy.eps';
6 figure; spy(D,'r'); axis off; print -depsc2
  '../PICTURES/Dspy.eps';

```

PYTHON/MATPLOTLIB-command for visualizing the structure of a matrix: `plt.spy (M)`

#### PYTHON-code 1.3.7: Visualizing the structure of matrices in PYTHON

```

1 n = 100
2 A = np.diag(np.mgrid[:n])
3 A[:, -1] = A[-1, :] = np.mgrid[:n]
4 plt.spy(A)
5 plt.spy(A[::-1, :])
6 plt.spy(np.dot(A, A))
7 plt.spy(np.dot(A, B))

```

### Remark 1.3.8 (Multiplying triangular matrices)

The following result is useful when dealing with matrix decompositions that often involve triangular matrices.

#### Lemma 1.3.9. Group of regular diagonal/triangular matrices

$$\mathbf{A}, \mathbf{B} \quad \left\{ \begin{array}{l} \text{diagonal} \\ \text{upper triangular} \\ \text{lower triangular} \end{array} \right. \Rightarrow \quad \mathbf{AB} \quad \text{and} \quad \mathbf{A}^{-1} \quad \left\{ \begin{array}{l} \text{diagonal} \\ \text{upper triangular} \\ \text{lower triangular} \end{array} \right. .$$

(assumes that  $\mathbf{A}$  is regular)

“Proof by visualization” → Rem. 1.3.5

### Experiment 1.3.10 (Scaling a matrix)

**Scaling** = multiplication with diagonal matrices (with non-zero diagonal entries):

It is important to know the different effect of multiplying with a diagonal matrix from left or right:

- \* multiplication with diagonal matrix *from left* ➤ **row scaling**

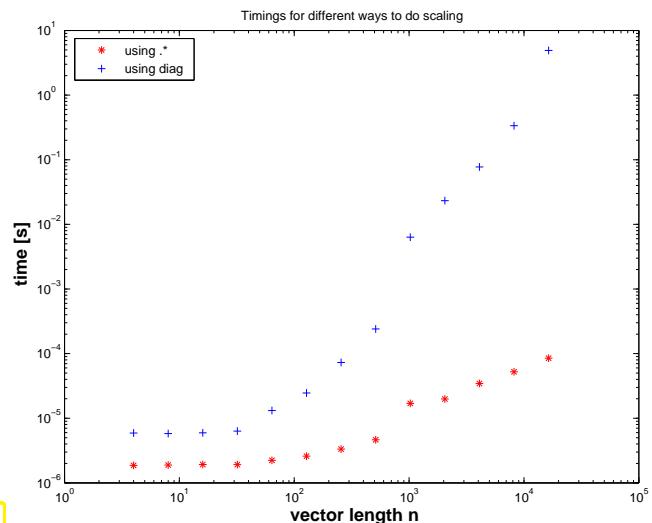
$$\begin{bmatrix} d_1 & 0 & 0 \\ 0 & d_2 & 0 \\ 0 & \ddots & d_n \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & & a_{2m} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} = \begin{bmatrix} d_1 a_{11} & d_1 a_{12} & \dots & d_1 a_{1m} \\ d_2 a_{21} & d_2 a_{22} & \dots & d_2 a_{2m} \\ \vdots & & & \vdots \\ d_n a_{n1} & d_n a_{n2} & \dots & d_n a_{nm} \end{bmatrix} = \begin{bmatrix} d_1 (\mathbf{A})_{1,:} \\ \vdots \\ d_n (\mathbf{A})_{n,:} \end{bmatrix} .$$

- \* multiplication with diagonal matrix *from right* ➤ **column scaling**

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & & a_{2m} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} \begin{bmatrix} d_1 & 0 & 0 \\ 0 & d_2 & 0 \\ 0 & \ddots & d_m \end{bmatrix} = \begin{bmatrix} d_1 a_{11} & d_2 a_{12} & \dots & d_m a_{1m} \\ d_1 a_{21} & d_2 a_{22} & \dots & d_m a_{2m} \\ \vdots & & & \vdots \\ d_1 a_{n1} & d_2 a_{n2} & \dots & d_m a_{nm} \end{bmatrix} = \begin{bmatrix} d_1 (\mathbf{A})_{:,1} & \dots & d_m (\mathbf{A})_{:,m} \end{bmatrix} .$$

Multiplication with a scaling matrix  $\mathbf{D} = \text{diag}(d_1, \dots, d_n) \in \mathbb{R}^{n,n}$  in MATLAB can be realised in two ways:

```
y = diag(d)*x;
y = d.*x;
```



### MATLAB-code 1.3.11: Timing multiplication with scaling matrix in MATLAB

```

1 % MATLAB script for timing a smart and foolish way to carry out
   multiplication
2 % with a scaling matrix in MATLAB, see Rem. 1.3.10.
3 nruns = 3; timings = [];
4 for n=2.^2:14
5     d = rand(n,1); x = rand(n,1);
6     tbad = realmax; tgood = realmax;
7     for j=1:nruns
8         tic; y = diag(d)*x; tbad = min(tbad,toc);
9         tic; y = d.*x; tgood = min(tgood,toc);
10    end
11    timings = [timings; n, tgood, tbad];
12 end
13
14 figure('name','scaletimings');
15 loglog(timings(:,1),timings(:,2),'r*',...
16         timings(:,1),timings(:,3),'b+');
17 xlabel('{\bf vector length n}', 'fontsize', 14);
18 ylabel('{\bf time [s]}', 'fontsize', 14);
19 title('Timings for different ways to do scaling');
20 legend('using .*', 'using diag', 'location', 'best');
21
22 print -depsc2 '../PICTURES/scaletiming.eps';

```

### PYTHON-code 1.3.12: Timing multiplication with scaling matrix in PYTHON

```

1 import numpy as np
2 from matplotlib import pyplot as plt
3 import timeit
4
5 # script for timing a smart and foolish way to carry out
6 # multiplication with a scaling matrix
7
8 nruns = 3

```

```

 9 | res = []
10 | for n in 2**np.mgrid[2:15]:
11 |     d = np.random.uniform(size=n)
12 |     x = np.random.uniform(size=n)
13 |
14 |     tbad = min(timeit.repeat(lambda: np.dot(np.diag(d), x),
15 |                               repeat=nruns, number=1))
16 |     tgood = min(timeit.repeat(lambda: d * x, repeat=nruns,
17 |                               number=1))
18 |
19 |     res.append((n, tbad, tgood))
20 |
21 ns, tbads, tgoods = np.transpose(res)
22 plt.figure()
23 plt.loglog(ns, tbads, '+', label='using np.diag')
24 plt.loglog(ns, tgoods, 'o', label='using *')
25 plt.legend(loc='best')
26 plt.title('Timing for different ways to do scaling')
27 plt.savefig('../PYTHON_PICTURES/scaletiming.eps')
28 plt.show()

```

Hardly surprising, the component-wise multiplication of the two vectors is way faster than the intermittent initialisation of a diagonal matrix (main populated by zeros) and the computation of a matrix  $\times$  vector product. Nevertheless, such blunders keep on haunting MATLAB codes.

### Remark 1.3.13 (Row and column transformations)

Simple operations on rows/columns of matrices, *cf.* what was done in Exp. 1.2.25, can often be expressed as multiplication with special matrices: For instance, given  $\mathbf{A} \in \mathbb{K}^{n,m}$  we obtain  $\mathbf{B}$  by adding row  $(\mathbf{A})_{j,:}$  to row  $(\mathbf{A})_{j+1,:}$ ,  $1 \leq j < n$ .

Realisation through matrix product

$$\mathbf{B} = \begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & 1 \\ & & 1 & \\ & & & \ddots \\ & & & & 1 \end{bmatrix} \mathbf{A} .$$

The matrix multiplying  $\mathbf{A}$  from the left is a specimen of a **transformation matrix**, a matrix that coincides with the identity matrix  $\mathbf{I}$  except for a single off-diagonal entry.

left-multiplication right-multiplication	with transformation matrices	$\rightarrow$ row transformations column transformations
---	------------------------------	--

row/column transformations will play a central role in Sect. 1.6.2

### Remark 1.3.14 (Matrix algebra)

A vector space  $(V, \mathbb{K}, +, \cdot)$ , where  $V$  is additionally equipped with a bi-linear and associative “multiplication” is called an algebra. Hence, the vector space of square matrices  $\mathbb{K}^{n,n}$  with matrix multiplication is an algebra with *unit element*  $\mathbf{I}$ .

### (1.3.15) Block matrix product

Given matrix dimensions  $M, N, K \in \mathbb{N}$  block sizes  $1 \leq n < N$  ( $n' := N - n$ ),  $1 \leq m < M$  ( $m' := M - m$ ),  $1 \leq k < K$  ( $k' := K - k$ ) we start from the following matrices:

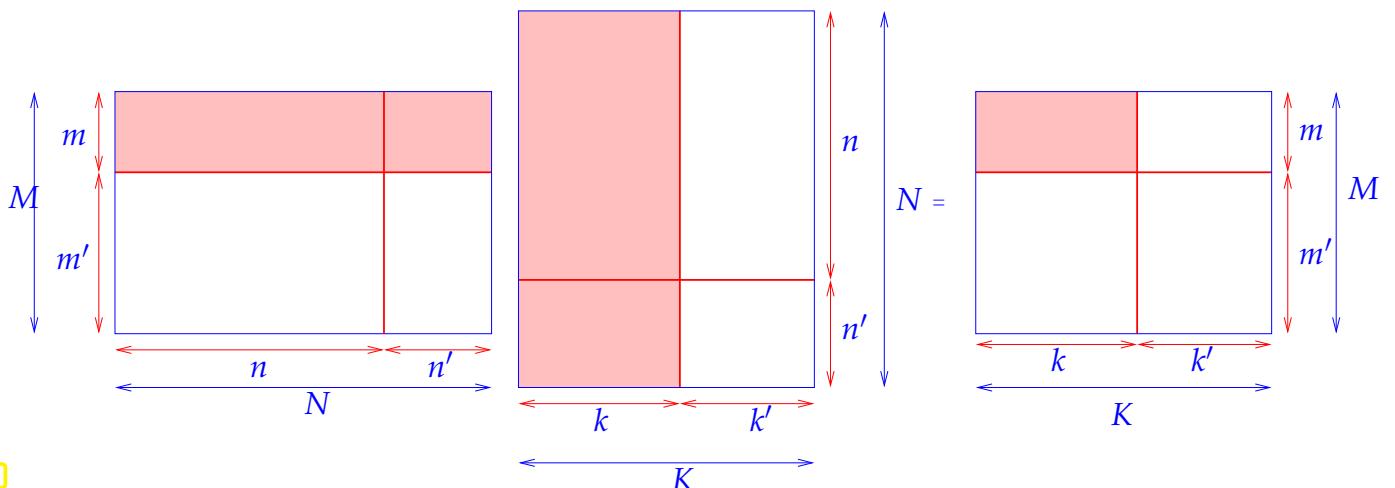
$$\begin{array}{lll} \mathbf{A}_{11} \in \mathbb{K}^{m,n} & \mathbf{A}_{12} \in \mathbb{K}^{m,n'} & \mathbf{B}_{11} \in \mathbb{K}^{n,k} \\ \mathbf{A}_{21} \in \mathbb{K}^{m',n} & \mathbf{A}_{22} \in \mathbb{K}^{m',n'} & , \quad \mathbf{B}_{21} \in \mathbb{K}^{n',k} \\ & & \mathbf{B}_{22} \in \mathbb{K}^{n',k'} \end{array}.$$

This matrices serve as sub-matrices or matrix blocks and are assembled into larger matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \in \mathbb{K}^{M,N} \quad , \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} \in \mathbb{K}^{N,K}.$$

It turns out that the matrix product  $\mathbf{AB}$  can be computed by the same formula as the product of simple  $2 \times 2$ -matrices:

$$\blacktriangleright \quad \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{bmatrix}. \quad (1.3.16)$$



Bottom line: one can compute with block-structured matrices in *almost* (\*) the same ways as with matrices with real/complex entries, see [63, Sect. 1.3.3].

 (\*) you must not use the commutativity of multiplication (because matrix multiplication is not commutative).

## 1.3.2 BLAS – Basic Linear Algebra Subprograms

**BLAS** (Basic Linear Algebra Subprograms) is a specification (API) that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot

products, linear combinations, and matrix multiplication. They are the de facto low-level routines for linear algebra libraries ([Wikipedia](#)).

The BLAS API is standardised by the [BLAS technical forum](#) and, due to its history dating back to the 70s, follows conventions of FORTRAN 77, see the [Quick Reference Guide](#) for examples. However, wrappers for other programming languages are available. CPU manufacturers and/or developers of operating systems usually supply highly optimised implementations:

- [OpenBLAS](#): open source implementation with some general optimisations, available under BSD license.
- [ATLAS](#) (Automatically Tuned Linear Algebra Software): open source BLAS implementation with auto-tuning capabilities. Comes with C and FORTRAN interfaces and is included in Linux distributions.
- [Intel MKL](#) (Math Kernel Library): commercial highly optimised BLAS implementation available for all Intel CPUs. Used by most proprietary simulation software and also MATLAB.

### Experiment 1.3.17 (Multiplying matrices in MATLAB)

#### MATLAB-code 1.3.18: Timing different implementations of matrix multiplication in MATLAB

```

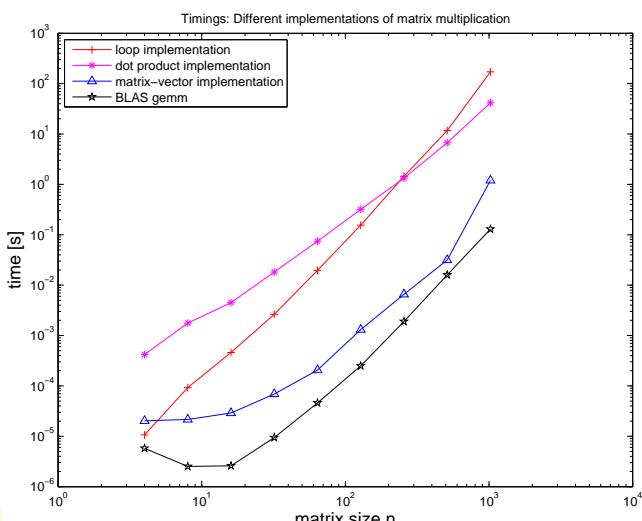
1 % MATLAB script for timing different implementations of matrix
2 % multiplications
3 nruns = 3; times = [];
4 for n=2.^2:10 % n = size of matrices
5   fprintf('matrix size n = %d\n',n);
6   A = rand(n,n); B = rand(n,n); C = zeros(n,n);
7   t1 = realmax;
8   % loop based implementation (no BLAS)
9   for l=1:nruns
10     tic;
11     for i=1:n, for j=1:n
12       for k=1:n, C(i,j) = C(i,j) + A(i,k)*B(k,j); end
13     end, end
14     t1 = min(t1,toe);
15   end
16   t2 = realmax;
17   % dot product based implementation (BLAS level 1)
18   for l=1:nruns
19     tic;
20     for i=1:n
21       for j=1:n, C(i,j) = dot(A(i,:),B(:,j)); end
22     end
23     t2 = min(t2,toe);
24   end
25   t3 = realmax;
26   % matrix-vector based implementation (BLAS level 2)
27   for l=1:nruns
28     tic;
29     for j=1:n, C(:,j) = A*B(:,j); end
30     t3 = min(t3,toe);
31   end

```

```

30    end
31    t4 = realmax;
32    % BLAS level 3 matrix multiplication
33    for l=1:nruns
34        tic; C = A*B; t4 = min(t4,toc);
35    end
36    times = [ times; n t1 t2 t3 t4];
37 end
38
39 figure('name','mmtiming');
40 loglog(times(:,1),times(:,2),'r+-',...
41         times(:,1),times(:,3),'m*-',...
42         times(:,1),times(:,4),'b^-',...
43         times(:,1),times(:,5),'kp-');
44 title('Timings: Different implementations of matrix
45       multiplication');
46 xlabel('matrix size n','fontsize',14);
47 ylabel('time [s]','fontsize',14);
48 legend('loop implementation','dot product implementation',...
49         'matrix-vector implementation','BLAS gemm (MATLAB *)',...
50         'location','northwest');
51 print -depsc2 '../PICTURES/mvtiming.eps';

```



The same applies to PYTHON code, a corresponding timing script is given here:

#### PYTHON-code 1.3.19: Timing different implementations of matrix multiplication in PYTHON

```

1  # script for timing different implementations of matrix multiplications
2  import numpy as np
3  from matplotlib import pyplot as plt
4  import timeit
5
6  def mm_loop_based(A, B, C):
7      m, n = A.shape
8      _, p = B.shape

```

#### Platform:

- \* Mac OS X 10.6
- \* Intel Core i7, 2.66 GHz
- \* L2 256 kB, L3 4 MB, Mem 4 GB
- \* MATLAB 7.10.0 (R 2010a)

In MATLAB we can achieve a tremendous gain in execution speed by relying on compact matrix/vector operations that invoke efficient BLAS routine.

Advise: avoid loops in MATLAB and replace them with vectorised operations.

```

9   for i in range(m):
10    for j in range(p):
11      for k in range(n):
12          C[i, j] += A[i, k] * B[k, j]
13  return C
14
15 def mm_blas1(A, B, C):
16     m, n = A.shape
17     _, p = B.shape
18     for i in range(m):
19         for j in range(p):
20             C[i, j] = np.dot(A[i, :], B[:, j])
21     return C
22
23 def mm_blas2(A, B, C):
24     m, n = A.shape
25     _, p = B.shape
26     for i in range(m):
27         C[i, :] = np.dot(A[i, :], B)
28     return C
29
30 def mm_blas3(A, B, C):
31     C = np.dot(A, B)
32     return C
33
34 def main():
35     nruns = 3
36     res = []
37     for n in 2**np.mgrid[2:11]:
38         print('matrix size n = {}'.format(n))
39         A = np.random.uniform(size=(n, n))
40         B = np.random.uniform(size=(n, n))
41         C = np.random.uniform(size=(n, n))
42
43         tloop = min(timeit.repeat(lambda: mm_loop_based(A, B, C),
44                                     repeat=nruns, number=1))
45         tblas1 = min(timeit.repeat(lambda: mm_blas1(A, B, C),
46                                     repeat=nruns, number=1))
47         tblas2 = min(timeit.repeat(lambda: mm_blas2(A, B, C),
48                                     repeat=nruns, number=1))
49         tblas3 = min(timeit.repeat(lambda: mm_blas3(A, B, C),
50                                     repeat=nruns, number=1))
51         res.append((n, tloop, tblas1, tblas2, tblas3))
52
53     ns, tloops, tblas1s, tblas2s, tblas3s = np.transpose(res)
54     plt.figure()
55     plt.loglog(ns, tloops, 'o', label='loop implementation')
56     plt.loglog(ns, tblas1s, '+', label='dot product
57                 implementation')
58     plt.loglog(ns, tblas2s, '*', label='matrix-vector
59

```

```

    implementation')
58 plt.loglog(ns, blas3s, '^', label='BLAS gemm (np.dot)')
59 plt.legend(loc='upper left')
60 plt.savefig('../PYTHON_PICTURES/mvtiming.eps')
61 plt.show()
62
63 if __name__ == '__main__':
64     main()

```

BLAS routines are grouped into “levels” according to the amount of data and computation involved (asymptotic complexity, see Section 1.4.1 and [28, Sect. 1.1.12]):

- **Level 1:** **vector** operations such as scalar products and vector norms.  
asymptotic complexity  $O(n)$ , (with  $n \hat{=} \text{vector length}$ ),  
e.g.: dot product:  $\rho = \mathbf{x}^\top \mathbf{y}$
- **Level 2:** **vector-matrix** operations such as matrix-vector multiplications.  
asymptotic complexity  $O(mn)$ , (with  $(m, n) \hat{=} \text{matrix size}$ ),  
e.g.: matrix  $\times$  vector multiplication:  $\mathbf{y} = \alpha \mathbf{A} \mathbf{x} + \beta \mathbf{y}$
- **Level 3:** **matrix-matrix** operations such as matrix additions or multiplications.  
asymptotic complexity often  $O(nmk)$ , (with  $(n, m, k) \hat{=} \text{matrix sizes}$ ),  
e.g.: matrix product:  $\mathbf{C} = \mathbf{AB}$

### Syntax of BLAS calls:

The functions have been implemented for different types, and are distinguished by the first letter of the function name. E.g. *sdot* is the dot product implementation for single precision and *ddot* for double precision.

#### \* **BLAS LEVEL 1:** vector operations, asymptotic complexity $O(n)$ , $n \hat{=} \text{vector length}$

- dot product  $\rho = \mathbf{x}^\top \mathbf{y}$

$$\textcolor{red}{\texttt{XDOT}}(N, X, \texttt{INCX}, Y, \texttt{INCY})$$

- $\textcolor{pink}{x} \in \{\text{S, D}\}$ , scalar type: S  $\hat{=} \text{type float}$ , D  $\hat{=} \text{type double}$
- $\textcolor{blue}{N} \hat{=} \text{length of vector (modulo stride INCX)}$
- $\textcolor{blue}{X} \hat{=} \text{vector } \mathbf{x}: \text{array of type } \textcolor{blue}{x}$
- $\texttt{INCX} \hat{=} \text{stride for traversing vector } X$
- $\textcolor{blue}{Y} \hat{=} \text{vector } \mathbf{y}: \text{array of type } \textcolor{blue}{x}$
- $\texttt{INCY} \hat{=} \text{stride for traversing vector } Y$

- vector operations  $\mathbf{y} = \alpha \mathbf{x} + \mathbf{y}$

$$\textcolor{red}{\texttt{XAXPY}}(N, \texttt{ALPHA}, X, \texttt{INCX}, Y, \texttt{INCY})$$

- $\textcolor{pink}{x} \in \{\text{S, D, C, Z}\}$ , S  $\hat{=} \text{type float}$ , D  $\hat{=} \text{type double}$ , C  $\hat{=} \text{type complex}$
- $\textcolor{blue}{N} \hat{=} \text{length of vector (modulo stride INCX)}$

- $\text{ALPHA} \hat{=} \text{scalar } \alpha$
- $\text{X} \hat{=} \text{vector } \mathbf{x}: \text{array of type } \mathbf{x}$
- $\text{INCX} \hat{=} \text{stride for traversing vector } \mathbf{X}$
- $\text{Y} \hat{=} \text{vector } \mathbf{y}: \text{array of type } \mathbf{x}$
- $\text{INCY} \hat{=} \text{stride for traversing vector } \mathbf{Y}$

\* **BLAS LEVEL 2:** matrix-vector operations, asymptotic complexity  $O(mn)$ ,  $(m, n) \hat{=} \text{matrix size}$

- matrix  $\times$  vector multiplication  $\mathbf{y} = \alpha \mathbf{A} \mathbf{x} + \beta \mathbf{y}$

$\text{XGEMV}(\text{TRANS}, M, N, \text{ALPHA}, \mathbf{A}, \text{LDA}, \mathbf{x},$

$\text{INCX}, \text{BETA}, \mathbf{y}, \text{INCY})$

- $\mathbf{x} \in \{\text{S, D, C, Z}\}$ , scalar type: S  $\hat{=} \text{type float}$ , D  $\hat{=} \text{type double}$ , C  $\hat{=} \text{type complex}$
- $M, N \hat{=} \text{size of matrix } \mathbf{A}$
- $\text{ALPHA} \hat{=} \text{scalar parameter } \alpha$
- $\mathbf{A} \hat{=} \text{matrix } \mathbf{A} \text{ stored in } \text{linear array of length } M \cdot N \text{ (column major arrangement)}$

$$(\mathbf{A})_{i,j} = \mathbf{A}[N * (j - 1) + i].$$

- LDA  $\hat{=} \text{"leading dimension"} \text{ of } \mathbf{A} \in \mathbb{K}^{n,m}$ , that is, the number  $n$  of rows.
- $\mathbf{X} \hat{=} \text{vector } \mathbf{x}: \text{array of type } \mathbf{x}$
- INCX  $\hat{=} \text{stride for traversing vector } \mathbf{X}$
- BETA  $\hat{=} \text{scalar parameter } \beta$
- $\mathbf{Y} \hat{=} \text{vector } \mathbf{y}: \text{array of type } \mathbf{x}$
- INCY  $\hat{=} \text{stride for traversing vector } \mathbf{Y}$

• **BLAS LEVEL 3:** matrix-matrix operations, asymptotic complexity  $O(mnk)$ ,  $(m, n, k) \hat{=} \text{matrix sizes}$

- matrix  $\times$  matrix multiplication  $\mathbf{C} = \alpha \mathbf{AB} + \beta \mathbf{C}$

$\text{XGEMM}(\text{TRANSA}, \text{TRANSB}, M, N, K,$   
 $\text{ALPHA}, \mathbf{A}, \text{LDA}, \mathbf{x}, \mathbf{B}, \text{LDB},$   
 $\text{BETA}, \mathbf{C}, \text{LDC})$

(☞ meaning of arguments as above)

### Remark 1.3.20 (BLAS calling conventions)

The BLAS calling syntax seems queer in light of modern object oriented programming paradigms, but it is a legacy of FORTRAN77, which was (and partly still is) the programming language, in which the BLAS

routines were coded.

It is a very common situation in scientific computing that one has to rely on old codes and libraries implemented in an old-fashioned style.

### Example 1.3.21 (Calling BLAS routines from C/C++)

When calling BLAS library functions from C, all arguments have to be passed by reference (as pointers), in order to comply with the argument passing mechanism of FORTRAN77, which is the model followed by BLAS.

#### C++-code 1.3.22: BLAS-based SAXPY operation in C++

```

1 #define daxpy_ daxpy
2 #include <iostream>
3
4 // Definition of the required BLAS function. This is usually done
5 // in a header file like blas.h that is included in the EIGEN3
6 // distribution
7 extern "C" {
8     int daxpy_(const int* n, const double* da, const double* dx,
9                 const int* incx, double* dy, const int* incy);
10 }
11
12 using namespace std;
13
14 int main() {
15     const int      n      = 5; // length of vector
16     const int      incx   = 1; // stride
17     const int      incy   = 1; // stride
18     double alpha = 2.5;      // scaling factor
19
20     // Allocated raw arrays of doubles
21     double* x  = new double [n];
22     double* y  = new double [n];
23
24     for (size_t i=0; i<n; i++){
25         x[i] = 3.1415 * i;
26         y[i] = 1.0 / (double)(i+1);
27     }
28
29     cout << "x=[ " ; for (size_t i=0; i<n; i++) cout << x[i] << ' ';
30     cout << " ] " << endl;
31     cout << "y=[ " ; for (size_t i=0; i<n; i++) cout << y[i] << ' ';
32     cout << " ] " << endl;
33
34     // Call the BLAS library function passing pointers to all arguments
35     // (Necessary when calling FORTRAN routines from C
36     daxpy_(&n, &alpha, x, &incx, y, &incy);

```

```

37
38 cout << "y = " << alpha << " * x + y = ]";
39 for (int i=0; i<n; i++) cout << y[i] << ' ' ; cout << "]" << endl;
40 return(0);
41 }
```

## 1.4 Computational effort

Large scale numerical computations require immense resources and execution time of numerical codes often becomes a central concern. Therefore, much emphasis has to be put on

1. designing algorithms that produce a desired result with (nearly) minimal computational effort (defined precisely below),
2. exploit possibilities for parallel and vectorised execution,
3. organising algorithms in order to make them fit memory hierarchies,
4. implementing codes that make optimal use of hardware resources and capabilities,

While Item 2–Item 4 are out of the scope of this course and will be treated in more advanced lectures, Item 1 will be a recurring theme.

The following definition encapsulates what is regarded as a measure for the “cost” of an algorithm in computational mathematics.

### Definition 1.4.1. Computational effort

The **computational effort** required by a numerical code amounts to the number of **elementary operations** (additions, subtractions, multiplications, divisions, square roots) executed in a run.

### (1.4.2) What computational effort does not tell us

Fifty years ago counting elementary operations provided good predictions of runtimes, but nowadays this is no longer true.



The computational effort involved in a run of a numerical code is only loosely related to overall execution time on modern computers.

This is conspicuous in Exp. 1.2.25, where algorithms incurring exactly the same computational effort took different times to execute.

The reason is that on today's computers a key bottleneck for fast execution is latency and bandwidth of memory, *cf.* the discussion at the end of Exp. 1.2.25 and [48]. Thus, concepts like **I/O-complexity** [1, 32] might be more appropriate for gauging the efficiency of a code, because they take into account the pattern of memory access.

### 1.4.1 (Asymptotic) complexity

The concept of computational effort from Def. 1.4.1 is still useful in a particular context:

#### Definition 1.4.3. (Asymptotic) complexity

The **asymptotic complexity** of an algorithm characterises the worst-case dependence of its computational effort on one or more **problem size parameter(s)** when these tend to  $\infty$ .

- *Problem size parameters* in numerical linear algebra usually are the lengths and dimensions of the vectors and matrices that an algorithm takes as inputs.
- *Worst case* indicates that the maximum effort over a set of admissible data is taken into account.

When dealing with asymptotic complexities a mathematical formalism comes handy:

#### Definition 1.4.4. Landau symbol [6, p. 7]

We write  $F(n) = O(G(n))$  for two functions  $F, G : \mathbb{N} \rightarrow \mathbb{R}$ , if there exists a constant  $C > 0$  and  $n_* \in \mathbb{N}$  such that

$$F(n) \leq C G(n) \quad \forall n \geq n_* .$$

More generally,  $F(n_1, \dots, n_k) = O(G(n_1, \dots, n_k))$  for two functions  $F, G : \mathbb{N}^k \rightarrow \mathbb{R}$  implies the existence of a constant  $C > 0$  and a threshold value  $n_* \in \mathbb{N}$  such that

$$F(n_1, \dots, n_k) \leq C G(n_1, \dots, n_k) \quad \forall n_1, \dots, n_k \in \mathbb{N}, \quad n_\ell \geq n_*, \quad \ell = 1, \dots, k .$$

#### Remark 1.4.5 (Meaningful “ $O$ -bounds” for complexity)

Of course, the definition of the Landau symbol leaves ample freedom for stating meaningless bounds; an algorithm that runs with linear complexity  $O(n)$  can be correctly labelled as possessing  $O(\exp(n))$  complexity.

Yet, whenever the Landau notation is used to describe asymptotic complexities, the bounds have to be **sharp** in the sense that no function with slower asymptotic growth will be possible inside the  $O$ . To make this precise we stipulate the following.

#### Sharpness of a complexity bound

Whenever the asymptotic complexity of an algorithm is stated as  $O(n^\alpha \log^\beta n \exp(\gamma n^\delta))$  with non-negative parameters  $\alpha, \beta, \gamma, \delta \geq 0$  in terms of the problem size parameter  $n$ , we take for granted that choosing a smaller value for any of the parameters will no longer yield a valid (or provable) asymptotic bound.

In particular

- \* complexity  $O(n)$  means that the complexity is not  $O(n^\alpha)$  for any  $\alpha < 1$ ,
- \* complexity  $O(\exp(n))$  excludes asymptotic complexity  $O(n^p)$  for any  $p \in \mathbb{R}$ .

Terminology: If the asymptotic complexity of an algorithm is  $O(n^p)$  with  $p = 1, 2, 3$  we say that it is of “linear”, “quadratic”, and “cubic” complexity, respectively.

### Remark 1.4.7 (Relevance of asymptotic complexity)

§ 1.4.2 warned us that computational effort and, thus, asymptotic complexity, of an algorithm for a concrete problem on a particular platform may not have much to do with the actual runtime (the blame goes to memory hierarchies, internal pipelining, vectorisation, etc.).

Then, why do we pay so much attention to asymptotic complexity in this course?

To a certain extent, the asymptotic complexity allows to predict the *dependence of the runtime* of a particular implementation of an algorithm *on the problem size* (for large problems).

For instance, an algorithm with asymptotic complexity  $O(n^2)$  is likely to take  $4\times$  as much time when the problem size is doubled.

### (1.4.8) Concluding polynomial complexity from runtime measurements

Available: “Measured runtimes”  $t_i = t_i(n_i)$  for different values  $n_1, n_2, \dots, n_N, n_i \in \mathbb{N}$ , of the problem size parameter

Conjectured: power law dependence  $t_i \approx Cn_i^\alpha$  (also “algebraic dependence”),  $\alpha \in \mathbb{R}$

How can we glean evidence that supports or refutes our conjecture from the data? Look at the data in **doubly logarithmic scale!**

$$t_i = Cn_i^\alpha \Rightarrow \log(t_i) \approx \log C + \alpha \log(n_i), \quad i = 1, \dots, N.$$

► If the conjecture holds true, then the points  $(n_i, t_i)$  will approximately lie on a *straight line* with **slope**  $\alpha$  in a doubly logarithmic plot (which can be created in MATLAB by the **loglog** plotting command).

➤ quick “visual test” of conjectured asymptotic complexity

More rigorous: Perform linear regression on  $(\log n_i, \log t_i), i = 1, \dots, N$  ( $\rightarrow$  Chapter 6)

## 1.4.2 Cost of basic operations

Performing elementary BLAS-type operations through simple (nested) loops, we arrive at the following obvious complexity bounds:

operation	description	#mul/div	#add/sub	asymp. complexity
dot product	$(\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x}^H \mathbf{y}$	$n$	$n - 1$	$O(n)$
tensor product	$(\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x}\mathbf{y}^H$	$nm$	0	$O(mn)$
matrix product <sup>(*)</sup>	$(\mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{B} \in \mathbb{R}^{n,k}) \mapsto \mathbf{AB}$	$mnk$	$mk(n - 1)$	$O(mnk)$

### Remark 1.4.9 (“Fast” matrix multiplication)

(\*): The  $\mathcal{O}(mnk)$  complexity bound applies to “straightforward” matrix multiplication according to (1.3.1).

For  $m = n = k$  there are (sophisticated) variants with better asymptotic complexity, e.g., the **divide-and-conquer Strassen algorithm** [76] with asymptotic complexity  $\mathcal{O}(n^{\log_2 7})$ :

Start from  $\mathbf{A}, \mathbf{B} \in \mathbb{K}^{n,n}$  with  $n = 2\ell$ ,  $\ell \in \mathbb{N}$ . The idea relies on the block matrix product (1.3.16) with  $\mathbf{A}_{ij}, \mathbf{B}_{ij} \in \mathbb{K}^{\ell,\ell}$ ,  $i, j \in \{1, 2\}$ . Let  $\mathbf{C} := \mathbf{AB}$  be partitioned accordingly:  $\mathbf{C} = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix}$ . Then tedious elementary computations reveal

$$\begin{aligned}\mathbf{C}_{11} &= \mathbf{Q}_0 + \mathbf{Q}_3 - \mathbf{Q}_4 + \mathbf{Q}_6, \\ \mathbf{C}_{21} &= \mathbf{Q}_1 + \mathbf{Q}_3, \\ \mathbf{C}_{12} &= \mathbf{Q}_2 + \mathbf{Q}_4, \\ \mathbf{C}_{22} &= \mathbf{Q}_0 + \mathbf{Q}_2 - \mathbf{Q}_1 + \mathbf{Q}_5,\end{aligned}$$

where the  $\mathbf{Q}_k \in \mathbb{K}^{\ell,\ell}$ ,  $k = 1, \dots, 7$  are obtained from

$$\begin{aligned}\mathbf{Q}_0 &= (\mathbf{A}_{11} + \mathbf{A}_{22}) * (\mathbf{B}_{11} + \mathbf{B}_{22}), \\ \mathbf{Q}_1 &= (\mathbf{A}_{21} + \mathbf{A}_{22}) * \mathbf{B}_{11}, \\ \mathbf{Q}_2 &= \mathbf{A}_{11} * (\mathbf{B}_{12} - \mathbf{B}_{22}), \\ \mathbf{Q}_3 &= \mathbf{A}_{22} * (-\mathbf{B}_{11} + \mathbf{B}_{21}), \\ \mathbf{Q}_4 &= (\mathbf{A}_{11} + \mathbf{A}_{12}) * \mathbf{B}_{22}, \\ \mathbf{Q}_5 &= (-\mathbf{A}_{11} + \mathbf{A}_{21}) * (\mathbf{B}_{11} + \mathbf{B}_{12}), \\ \mathbf{Q}_6 &= (\mathbf{A}_{12} - \mathbf{A}_{22}) * (\mathbf{B}_{21} + \mathbf{B}_{22}).\end{aligned}$$

Beside a considerable number of matrix additions (computational effort  $\mathcal{O}(n^2)$ ) it takes only 7 multiplications of matrices of size  $n/2$  to compute  $\mathbf{C}$ ! Strassen’s algorithm boils down to the *recursive application* of these formulas for  $n = 2^k$ ,  $k \in \mathbb{N}$ .

A refined algorithm of this type can achieve complexity  $\mathcal{O}(n^{2.36})$ , see [14].

### 1.4.3 Reducing complexity in numerical linear algebra: Some tricks

In computations involving matrices and vectors complexity of algorithms can often be reduced by performing the operations in a particular order:

#### Example 1.4.10 (Efficient associative matrix multiplication)

We consider the multiplication with a **rank-1-matrix**. Matrices with rank 1 can always be obtained as the tensor product of two vectors, that is, the matrix product of a column vector and a row vector. Given  $\mathbf{a} \in \mathbb{K}^m$ ,  $\mathbf{b} \in \mathbb{K}^n$ ,  $\mathbf{x} \in \mathbb{K}^n$  we may compute the vector  $\mathbf{y} = \mathbf{ab}^\top \mathbf{x}$  in two ways:

$$\mathbf{y} = (\mathbf{ab}^\top) \mathbf{x}. \quad (1.4.11) \qquad \qquad \qquad \mathbf{y} = \mathbf{a}(\mathbf{b}^\top \mathbf{x}). \quad (1.4.12)$$

$$\mathbf{T} = \mathbf{a} * \mathbf{b}' ; \quad \mathbf{y} = \mathbf{T} * \mathbf{x}; \qquad \qquad \qquad \mathbf{t} = \mathbf{b}' * \mathbf{x}; \quad \mathbf{y} = \mathbf{a} * \mathbf{t};$$

► complexity  $\mathcal{O}(mn)$       ► complexity  $\mathcal{O}(n + m)$  (“linear complexity”)

Visualization of evaluation according to (1.4.11):

$$\left[ \begin{array}{|c|} \hline \end{array} \right] \cdot \left[ \begin{array}{|c|} \hline \end{array} \right] = \left[ \begin{array}{|c|} \hline \end{array} \right] \left[ \begin{array}{|c|} \hline \end{array} \right]$$

Visualization of evaluation according to (1.4.12):

$$\left[ \begin{array}{|c|} \hline \end{array} \right] \left[ \begin{array}{|c|} \hline \end{array} \right] \cdot \left[ \begin{array}{|c|} \hline \end{array} \right] = \left[ \begin{array}{|c|} \hline \end{array} \right] \left[ \begin{array}{|c|} \hline \end{array} \right]$$

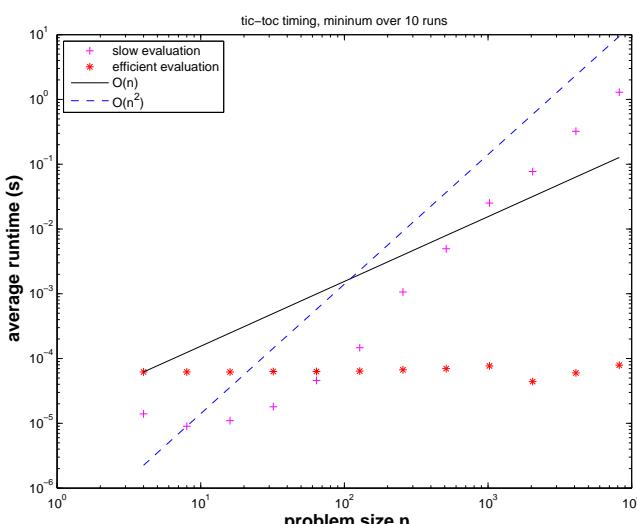


Fig. 9

▷ average runtimes for efficient/inefficient matrix×vector multiplication with rank-1 matrices (MATLAB `tic-toc` timing), see § 1.4.8 for the rationale behind choosing a *doubly logarithmic plot*.

Platform:

- MATLAB 7.4.0.336 (R2007a)
- Genuine Intel(R) CPU T2500 @ 2.00GHz
- Linux 2.6.16.27-0.9-smp

### MATLAB-code 1.4.13: MATLAB code for Ex. 1.4.10

```

1 function dottenstimering(N,nruns)
2 % This function compares the runtimes for the multiplication of a
3 % vector with a
4 % rank-1 matrix  $\mathbf{ab}^\top$ ,  $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$  using different associative
5 % evaluations measurements consider minimal time for several (nruns)
6 % runs
7
8
9 times = [] ; % matrix for storing recorded runtimes
10 for n=N
11 % Initialize dense vectors  $\mathbf{a}$ ,  $\mathbf{b}$ ,  $\mathbf{x}$  (column vectors!)
12 a = (1:n)'; b = (n:-1:1)'; x = rand(n,1);
13
14 % Measuring times using MATLAB tic-toc commands
15 tfool = realmax; for i=1:nruns, tic; y = (a*b')*x; tfool =
16     min(tfool,toc); end;
    tsmart = realmax; for i=1:nruns, tic; y = a.*dot(b',x); tsmart =
        min(tsmart,toc); end;
```

```

17 times = [times;n, tfool, tsmart];
18 end
19
20 % log-scale plot for investigation of asymptotic complexity
21 figure('name','dottenstming');
22 loglog(times(:,1),times(:,2),'m+',...
23 times(:,1),times(:,3),'r*',...
24 times(:,1),times(:,1)*times(1,3)/times(1,1),'k-',...
25 times(:,1),(times(:,1).^2)*times(2,2)/(times(2,1)^2),'b--');
26 xlabel('{\bf problem size n}', 'fontsize', 14);
27 ylabel('{\bf average runtime (s)}', 'fontsize', 14);
28 title('tic-toc timing, minimum over 10 runs');
29 legend('slow evaluation', 'efficient evaluation',...
30 'O(n)', 'O(n^2)', 'location', 'northwest');
31 print -depsc2 '../PICTURES/dottenstming.eps';

```

Complexity can sometimes be reduced by reusing intermediate results.

### Example 1.4.14 (Hidden summation)

The asymptotic complexity of the MATLAB function

```

function y = lrtrimult(A,B,x)
y = triu(A*B')*x;

```

when supplied with two **low-rank** matrices  $\mathbf{A}, \mathbf{B} \in \mathbb{K}^{n,p}$ ,  $p \ll n$  in terms of  $n \rightarrow \infty$  obviously is  $O(n^2)$ , because an intermediate  $n \times n$ -matrix  $\mathbf{AB}^T$  is built.

First, consider the case of a tensor product (= rank-1) matrix, that is,  $p = 1$ ,  $\mathbf{A} \leftrightarrow \mathbf{a} = [a_1, \dots, a_n]^\top \in \mathbb{K}^n$ ,  $\mathbf{B} \leftrightarrow \mathbf{b} = [b_1, \dots, b_n] \in \mathbb{K}^n$ . Then

$$\begin{aligned}
\mathbf{y} = \text{triu}(\mathbf{ab}^T)\mathbf{x} &= \begin{bmatrix} a_1b_1 & a_1b_2 & \dots & \dots & \dots & a_1b_n \\ 0 & a_2b_2 & a_2b_3 & \dots & \dots & a_2b_n \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & a_nb_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \\
&= \begin{bmatrix} a_1 \\ \ddots \\ \ddots \\ a_n \end{bmatrix} \underbrace{\begin{bmatrix} 1 & 1 & \dots & \dots & 1 \\ 0 & 1 & 1 & \dots & \dots & 1 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & 1 \end{bmatrix}}_{\mathbf{T}} \begin{bmatrix} b_1 \\ \ddots \\ \ddots \\ b_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}.
\end{aligned}$$

Thus, the core problem is the fast multiplication of a vector with an upper triangular matrix  $\mathbf{T}$  described in MATLAB syntax by `triu(ones(n,n))`. Note that multiplication of a vector  $\mathbf{x}$  with  $\mathbf{T}$  yields a vector

of partial sums of components of  $\mathbf{x}$  starting from last component. This can be achieved by invoking the special MATLAB command **cumsum**. We also observe that

$$\mathbf{AB}^T = \sum_{\ell=1}^p (\mathbf{A})_{:, \ell} ((\mathbf{B})_{:, \ell})^\top,$$

so that the computations for the special case  $p = 1$  discussed above can simply be reused  $p$  times!

#### MATLAB-code 1.4.15: Efficient multiplication with the upper diagonal part of a rank- $p$ -matrix

```

1 function y = lrtrimulteff(A,B,x)
2 [n,p] = size(A);
3 if (size(B) ~= [n,p]), error('size mismatch'); end
4 y = zeros(n,1);
5 for l=1:p, y = y + A(:,l).*cumsum(B(:,l).*x,'reverse'); end
```

This code enjoys the obvious complexity of  $O(pn)$  for  $p, n \rightarrow \infty$ ,  $p < n$ .

The next concept from linear algebra is important in the context of computing with multi-dimensional arrays.

#### Definition 1.4.16. Kronecker product

[Kronecker product] The **Kronecker product**  $\mathbf{A} \otimes \mathbf{B}$  of two matrices  $\mathbf{A} \in \mathbb{K}^{m,n}$  and  $\mathbf{B} \in \mathbb{K}^{l,k}$ ,  $m, n, l, k \in \mathbb{N}$ , is the  $(ml) \times (nk)$ -matrix

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} (\mathbf{A})_{11}\mathbf{B} & (\mathbf{A})_{1,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{1,n}\mathbf{B} \\ (\mathbf{A})_{2,1}\mathbf{B} & (\mathbf{A})_{2,2}\mathbf{B} & & & \vdots \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ (\mathbf{A})_{m,1}\mathbf{B} & (\mathbf{A})_{m,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{m,n}\mathbf{B} \end{bmatrix} \in \mathbb{K}^{nl, nk}.$$

In MATLAB the Kronecker product of two matrices can be formed by calling the function **kron**.

#### Example 1.4.17 (Multiplication of Kronecker product with vector)

The MATLAB code

```
y = kron(A, B)*x
```

when invoked with two matrices  $\mathbf{A} \in \mathbb{K}^{m,n}$  and  $\mathbf{B} \in \mathbb{K}^{l,k}$  and a vector  $\mathbf{x} \in \mathbb{K}^{nk}$ , will suffer an asymptotic complexity of  $O(m \cdot n \cdot l \cdot k)$ , determined by the size of the intermediate dense matrix  $\mathbf{A} \otimes \mathbf{B} \in \mathbb{K}^{ml, nk}$ .

Using the partitioning of the vector  $\mathbf{x}$  into  $n$  equally long sub-vectors

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}^1 \\ \mathbf{x}^2 \\ \vdots \\ \mathbf{x}^n \end{bmatrix}, \quad \mathbf{x}^j \in \mathbb{K}^k,$$

we find the representation

$$(\mathbf{A} \otimes \mathbf{B})\mathbf{x} = \begin{bmatrix} (\mathbf{A})_{1,1}\mathbf{Bx}^1 + (\mathbf{A})_{1,2}\mathbf{Bx}^2 + \cdots + (\mathbf{A})_{1,n}\mathbf{Bx}^n \\ (\mathbf{A})_{2,1}\mathbf{Bx}^1 + (\mathbf{A})_{2,2}\mathbf{Bx}^2 + \cdots + (\mathbf{A})_{2,n}\mathbf{Bx}^n \\ \vdots \\ \vdots \\ (\mathbf{A})_{m,1}\mathbf{Bx}^1 + (\mathbf{A})_{m,2}\mathbf{Bx}^2 + \cdots + (\mathbf{A})_{m,n}\mathbf{Bx}^n \end{bmatrix}.$$

The idea is to form the products  $\mathbf{Bx}^j$ ,  $j = 1, \dots, n$ , once, and then combine them linearly with coefficients given by the entries in the rows of  $\mathbf{A}$ :

#### MATLAB-code 1.4.18: Efficient multiplication of Kronecker product with vector in MATLAB

```

1 function y = kronmultv(A,B,x)
2 n = size(A,2); k = size(B,2);
3 if (numel(x) ~= n*k), error('size mismatch'); end
4 % Chop up vector into n pieces of length k
5 X = reshape(x,k,n);
6 % Multiply each piece with matrix B
7 T = B*X; %
8 % Form weighted sums of products
9 % weights supplied by rows of A
10 T = T*transpose(A); %
11 % Put matrix columns on top of each other
12 y = T(:);
```

#### PYTHON-code 1.4.19: Efficient multiplication of Kronecker product with vector in PYTHON

```

1 def kronmultv(A, B, x):
2     n, k = A.shape[1], B.shape[1]
3     assert x.size == n * k, 'size mismatch'
4     xx = np.reshape(x, (n, k))
5     Z = np.dot(xx, B.T)
6     yy = np.dot(A, Z)
7     return np.ravel(yy)
```

Note that different reshaping is used in the PYTHON code due to the default row major storage order.

The asymptotic complexity of this code is determined by the two matrix multiplications in Line 7 and Line 10. This yields the asymptotic complexity  $O(lkn + mnl)$  for  $l, k, m, n \rightarrow \infty$ .

## 1.5 Machine arithmetic

### 1.5.1 Experiment: Loss of orthogonality

### (1.5.1) Gram-Schmidt orthogonalisation

From linear algebra we recall the fundamental algorithm of **Gram-Schmidt orthogonalisation** of an ordered finite set  $\{\mathbf{a}^1, \dots, \mathbf{a}^k\}$ ,  $k \in \mathbb{N}$ , of vectors  $\mathbf{a}^\ell \in \mathbb{K}^n$ :

```

1:  $\mathbf{q}^1 := \frac{\mathbf{a}^1}{\|\mathbf{a}^1\|_2}$  % 1st output vector
2: for  $j = 2, \dots, k$  do
   { % Orthogonal projection
3:    $\mathbf{q}^j := \mathbf{a}^j$ 
4:   for  $\ell = 1, 2, \dots, j-1$  do
5:     {  $\mathbf{q}^j \leftarrow \mathbf{q}^j - \mathbf{a}^j \cdot \mathbf{q}^\ell \mathbf{q}^\ell$  }
6:   if ( $\mathbf{q}^j = \mathbf{0}$ ) then STOP
7:   else {  $\mathbf{q}^j \leftarrow \frac{\mathbf{q}^j}{\|\mathbf{q}^j\|_2}$  }
}

```

In linear algebra we have learnt that, if it does not **STOP** prematurely, this algorithm will compute **orthonormal vectors**  $\mathbf{q}^1, \dots, \mathbf{q}^k$  satisfying

$$\text{Span}\{\mathbf{q}^1, \dots, \mathbf{q}^k\} = \text{Span}\{\mathbf{a}^1, \dots, \mathbf{a}^k\}, \quad (1.5.2)$$

for all  $\ell \in \{1, \dots, k\}$ .

More precisely, if  $\mathbf{a}^1, \dots, \mathbf{a}^\ell$ ,  $\ell \leq k$ , are linearly independent, then the Gram-Schmidt algorithm will not terminate before the  $\ell + 1$ -th step.

☞ Notation:  $\|\cdot\|_2 \hat{=} \text{Euclidean norm of a vector } \in \mathbb{K}^n$

#### MATLAB-code 1.5.3: Gram-Schmidt orthogonalisation in MATLAB

```

1 function Q = gramschmidt(A)
2 % Gram-Schmidt orthogonalization of column vectors
3 % Arguments: Matrix A passes vectors in its columns
4 % Return values: orthonormal system in columns of matrix Q
5 [n,k] = size(A); % Get number k of vectors and dimension n of space
6 Q = A(:,1)/norm(A(:,1)); % First basis vector
7 for j=2:k
8   q = A(:,j) - Q*(Q'*A(:,j)); % Orthogonal projection; loop-free
   implementation
9   nq = norm(q); % Check premature termination
10  if (nq < (1E-9)*norm(A(:,j))), break; end % Safe check for == 0
11  Q = [Q, q/nq]; % Add new basis vector as another column of Q
12 end

```

We will soon learn the rationale behind the odd test in Line 10. An implementation of Gram-Schmidt orthonormalization in C++11 based on a simple vector class is given in Ex. 0.2.38, see Code 0.2.39.

#### PYTHON-code 1.5.4: Gram-Schmidt orthogonalisation in PYTHON

```

1 def gramschmidt(A):
2   _, k = A.shape
3   Q = A[:, [0]] / np.linalg.norm(A[:, 0])
4   for j in range(1, k):
5     q = A[:, j] - np.dot(Q, np.dot(Q.T, A[:, j]))
6     nq = np.linalg.norm(q)
7     if nq < 1e-9 * np.linalg.norm(A[:, j]):
8       break
9     Q = np.column_stack([Q, q / nq])
10    return Q

```

Note the different loop range due to the zero-based indexing in PYTHON.

### Experiment 1.5.5 (Unstable Gram-Schmidt orthonormalization)

#### MATLAB-code 1.5.6: Wrong result from Gram-Schmidt orthogonalisation

```

1 % MATLAB script demonstrating the effect of roundoff on the result of
2 % Gram-Schmidt orthogonalization
3 format short; % Print only a few digits in outputs
4 % Create special matrix the so-called Hilbert matrix: (A)i,j = (i+j-1)-1
5 A = hilb(10); % 10x10 Hilbert matrix
6 Q = gramschmidt(A); % Gram-Schmidt orthogonalization of columns of A
7 % Test orthonormality of column of Q, which should be an orthogonal
8 % matrix according to theory
9 I = Q'*Q, % Should be the unit matrix, but isn't !
10
11 % MATLAB's internal Gram-Schmidt orthogonalization by QR-decomposition
12 [Q1,R1] = qr(A,0);
13 D = A - Q1*R1, % Check whether we get the expected result
14 I1 = Q1'*Q1, % Test orthonormality

```

Code Code 1.5.6 produces the following output

```

I =
1.0000 0.0000 -0.0000 0.0000 -0.0000 0.0000 -0.0000 -0.0000 -0.0000 -0.0000
0.0000 1.0000 -0.0000 0.0000 -0.0000 0.0000 -0.0000 -0.0000 -0.0000 -0.0000
-0.0000 -0.0000 1.0000 0.0000 -0.0000 0.0000 -0.0000 -0.0000 -0.0000 -0.0000
0.0000 0.0000 0.0000 1.0000 -0.0000 0.0000 -0.0000 -0.0000 -0.0000 -0.0000
-0.0000 -0.0000 -0.0000 1.0000 0.0000 -0.0000 -0.0008 -0.0007 -0.0007 -0.0006
0.0000 0.0000 0.0000 0.0000 1.0000 -0.0540 -0.0430 -0.0430 -0.0360 -0.0289
-0.0000 -0.0000 -0.0000 -0.0008 -0.0540 1.0000 0.9999 0.9998 0.9996
-0.0000 -0.0000 -0.0000 -0.0007 -0.0430 0.9999 1.0000 1.0000 0.9999
-0.0000 -0.0000 -0.0000 -0.0007 -0.0360 0.9998 1.0000 1.0000 1.0000
-0.0000 -0.0000 -0.0000 -0.0006 -0.0289 0.9996 0.9999 1.0000 1.0000

```

```

I1 =
1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 -0.0000 0.0000 0.0000 0.0000
0.0000 1.0000 0 -0.0000 -0.0000 0.0000 0.0000 0 -0.0000 0
0.0000 0 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0
0.0000 -0.0000 0.0000 1.0000 -0.0000 0.0000 -0.0000 0.0000 -0.0000 -0.0000
0.0000 -0.0000 0.0000 -0.0000 1.0000 0.0000 0.0000 -0.0000 0.0000 0
0.0000 -0.0000 0.0000 0.0000 0.0000 1.0000 -0.0000 -0.0000 0.0000 0
0.0000 0.0000 0.0000 -0.0000 0.0000 -0.0000 1.0000 0.0000 0.0000 -0.0000
-0.0000 0 0.0000 0.0000 0.0000 -0.0000 0.0000 1.0000 0.0000 -0.0000
0.0000 -0.0000 0.0000 -0.0000 -0.0000 0.0000 0.0000 0.0000 1.0000 0.0000
0.0000 0 0 -0.0000 0 0 -0.0000 -0.0000 0.0000 1.0000

```

Obviously, the vectors produced by the function `gramschmidt` fail to be orthonormal, contrary to the predictions of rigorous results from linear algebra!

Computers cannot compute “properly” in  $\mathbb{R}$ : numerical computations may not respect the laws of analysis and linear algebra!

This introduces an important new aspect in the study of numerical algorithms.

**Remark 1.5.7 (QR-decomposition** → [59, Satz 5.2], [33, Sect. 7.3])

In Code 1.5.6 we saw the use of the MATLAB function `qr()` for the purpose of Gram-Schmidt orthogonalisation. In fact, the Gram-Schmidt algorithm provides the following result from linear algebra:

### Theorem 1.5.8. QR-decomposition

For any matrix  $\mathbf{A} \in \mathbb{K}^{n,k}$  with  $\text{rank}(\mathbf{A}) = k$  there exists

- (i) a unique Matrix  $\mathbf{Q}_0 \in \mathbb{R}^{n,k}$  that satisfies  $\mathbf{Q}_0^H \mathbf{Q}_0 = \mathbf{I}_k$ , and a unique *upper triangular* Matrix  $\mathbf{R}_0 \in \mathbb{K}^{k,k}$  with  $(\mathbf{R}_0)_{i,i} > 0$ ,  $i \in \{1, \dots, k\}$ , such that

$$\mathbf{A} = \mathbf{Q}_0 \cdot \mathbf{R}_0 \quad (\text{"economical" QR-decomposition}),$$

- (ii) a *unitary* Matrix  $\mathbf{Q} \in \mathbb{K}^{n,n}$  and a unique *upper triangular*  $\mathbf{R} \in \mathbb{K}^{n,k}$  with  $(\mathbf{R})_{i,i} > 0$ ,  $i \in \{1, \dots, n\}$ , such that

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \quad (\text{full QR-decomposition}).$$

If  $\mathbb{K} = \mathbb{R}$  all matrices will be real and  $\mathbf{Q}$  is then *orthogonal*.

Visualisation: “economical” QR-decomposition:  $\mathbf{Q}_0^H \mathbf{Q}_0 = \mathbf{I}_k$ ,

$$\left[ \begin{array}{c|c} \mathbf{A} & \\ \hline \end{array} \right] = \left[ \begin{array}{c|c} \mathbf{Q}_0 & \\ \hline \end{array} \right] \left[ \begin{array}{c|c} \mathbf{R}_0 & \\ \hline \end{array} \right], \quad \mathbf{A} = \mathbf{Q}_0 \mathbf{R}_0, \quad \mathbf{Q}_0 \in \mathbb{K}^{n,k}, \quad \mathbf{R}_0 \in \mathbb{K}^{k,k},$$

(1.5.9)

Visualisation: full QR-decomposition:  $\mathbf{Q}^H \mathbf{Q} = \mathbf{I}_n$ ,

$$\left[ \begin{array}{c|c} \mathbf{A} & \\ \hline \end{array} \right] = \left[ \begin{array}{c|c} \mathbf{Q} & \\ \hline \end{array} \right] \left[ \begin{array}{c|c} \mathbf{R} & \\ \hline \end{array} \right], \quad \mathbf{A} = \mathbf{Q} \mathbf{R}, \quad \mathbf{Q} \in \mathbb{K}^{n,n}, \quad \mathbf{R} \in \mathbb{K}^{n,k},$$

(1.5.10)

For square  $\mathbf{A}$ , that is,  $n = k$ , both QR-decompositions coincide.

- Up to signs the columns of the matrix  $\mathbf{Q}$  available from the QR-decomposition of  $\mathbf{A}$  are the same vectors as produced by the Gram-Schmidt orthogonalisation of the columns of  $\mathbf{A}$ .

! Code 1.5.6 demonstrates a case where a desired result can be obtained by two *algebraically equivalent* computations, that is, they yield the same result in a mathematical sense. Yet, when implemented on a computer, the results can be *vastly different*. One algorithm may produce junk (“unstable algorithm”), whereas the other lives up to the expectations (“stable algorithm”)

Supplement to Exp. 1.5.5: despite its ability to produce orthonormal vectors, we get as output for  $\mathbf{D} = \mathbf{A} - \mathbf{Q}\mathbf{1} \star \mathbf{R}\mathbf{1}$ :

```
D = 1.0e-15 *
 0.2220   0.4441   0.3331   0.3053   0.2220   0.1388   0.1665   0.1249   0.1110   0.1388
    0   0.0555   0.0555      0   0.0278      0   -0.0278   -0.0139      0   0.0139
 -0.0555   0.0555      0      0      0      0   -0.0139   0.0278      0
    0   0.0278   0.0278      0      0   -0.0278      0      0   0.0139   0.0278
    0   0.0278   0.0278   0.0139      0   0.0278      0      0   0.0139   0.0278
 -0.0278   0.0278      0   0.0139   0.0139   0.0139   -0.0139   0.0139   0.0139
    0   0.0139      0      0      0   -0.0139   0.0139      0      0
    0   0.0278   0.0139      0   -0.0139   -0.0139   -0.0139   0.0278   0.0069
  0.0139   0.0278   0.0278   0.0139   0.0139      0      0      0   0.0139
    0   0.0278      0   -0.0139   -0.0139   0.0139      0   0.0069   -0.0069
```

- The computed QR-decomposition apparently fails to meet the exact algebraic requirements stipulated by Thm. 1.5.8. However, note the tiny size of the “defect”.

### Remark 1.5.11 (QR-decomposition in MATLAB)

The two different QR-decomposition (1.5.9) and (1.5.10) of a matrix  $\mathbf{A} \in \mathbb{K}^{n,k}$ ,  $n, k \in \mathbb{N}$ , can be computed in MATLAB as follows:

```
[Q, R] = qr(A, 0); % Economical QR-decomposition (1.5.9)
[Q, R] = qr(A); % Full QR-decomposition (1.5.10)
```

The returned matrices  $\mathbf{Q}$  and  $\mathbf{R}$  correspond to the QR-factors  $\mathbf{Q}$  and  $\mathbf{R}$  as defined above.

### Remark 1.5.12 (QR-decomposition in PYTHON)

The two different QR-decomposition (1.5.9) and (1.5.10) of a matrix  $\mathbf{A} \in \mathbb{K}^{n,k}$ ,  $n, k \in \mathbb{N}$ , can be computed in PYTHON as follows:

```
1 Q, R = np.linalg.qr(A, mode='reduced') # Economical (1.5.9)
2 Q, R = np.linalg.qr(A, mode='complete') # Full (1.5.10)
```

The returned matrices  $\mathbf{Q}$  and  $\mathbf{R}$  correspond to the QR-factors  $\mathbf{Q}$  and  $\mathbf{R}$  as defined above.

## 1.5.2 Machine numbers

### (1.5.13) The finite and discrete set of machine numbers

The reason, why computers must fail to execute exact computations with real numbers is clear:

Computer = finite automaton



can handle only *finitely many* numbers, not  $\mathbb{R}$

machine numbers, set  $\mathbb{M}$

Essential property:  $\mathbb{M}$  is a **finite, discrete** subset of  $\mathbb{R}$  (its numbers separated by gaps)

The set of machine numbers  $\mathbb{M}$  cannot be closed under elementary arithmetic operations  $+, -, \cdot, /$ , that is, when adding, multiplying, etc., two machine numbers the result may not belong to  $\mathbb{M}$ .

The results of elementary operations with operands in  $\mathbb{M}$  have to be mapped back to  $\mathbb{M}$ , an operation called **rounding**.



roundoff errors (*ger.*: Rundungsfehler) are inevitable

The impact of roundoff means that mathematical identities may not carry over to the computational realm. As we have seen above in Exp. 1.5.5

Computers cannot compute “properly” !



**numerical computations**  $\neq$  **analysis  
linear algebra**

This introduces a *new* and *important* aspect in the study of numerical algorithms!

#### (1.5.14) Internal representation of machine numbers

Now we give a brief sketch of the internal structure of machine numbers  $\in \mathbb{M}$ . The main insight will be that

“Computers use floating point numbers (scientific notation)”

#### Example 1.5.15 (Decimal floating point numbers)

Some 3-digit normalized decimal floating point numbers:

valid:  $0.723 \cdot 10^2$ ,  $0.100 \cdot 10^{-20}$ ,  $-0.801 \cdot 10^5$   
 invalid:  $0.033 \cdot 10^2$ ,  $1.333 \cdot 10^{-4}$ ,  $-0.002 \cdot 10^3$

General form of an  $m$ -digit **normalized decimal floating point number**:

$x = \pm$  . ...  $\cdot 10^E$   
*never = 0!*  
*exponent*  $\in \mathbb{Z}$

Of course, computers are restricted to a *finite range* of exponents:

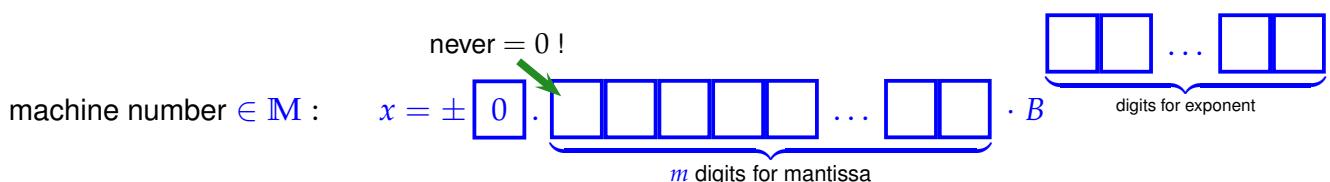
### Definition 1.5.16. Machine numbers/floating point numbers

Given

- ☞ basis  $B \in \mathbb{N} \setminus \{1\}$ ,
- ☞ exponent range  $\{e_{\min}, \dots, e_{\max}\}$ ,  $e_{\min}, e_{\max} \in \mathbb{Z}$ ,  $e_{\min} < e_{\max}$ ,
- ☞ number  $m \in \mathbb{N}$  of digits (for mantissa),

the corresponding set of machine numbers is

$$\mathbb{M} := \{d \cdot B^E : d = i \cdot B^{-m}, i = B^{m-1}, \dots, B^m - 1, E \in \{e_{\min}, \dots, e_{\max}\}\}$$



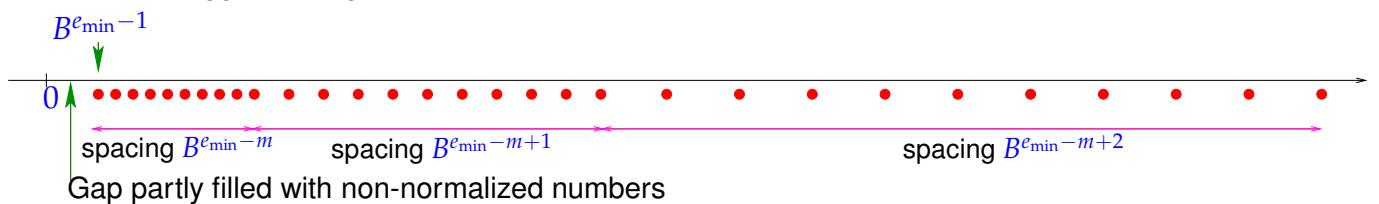
### Remark 1.5.17 (The boundaries of $\mathbb{M}$ )

Clearly, there is a largest element of  $\mathbb{M}$  and two that are closest to zero. These are mainly determined by the range for the exponent  $E$ , cf. Def. 1.5.16.

- Largest machine number (in modulus) :  $x_{\max} = \max |\mathbb{M}| = (1 - B^{-m}) \cdot B^{e_{\max}}$
- Smallest machine number (in modulus) :  $x_{\min} = \min |\mathbb{M}| = B^{-1} \cdot B^{e_{\min}}$
- In MATLAB these extremal machine numbers are accessible through the `realmax` and `realmin` variables.

### Remark 1.5.18 (Distribution of machine numbers)

From Def. 1.5.16 it is clear that there are equi-spaced sections of  $\mathbb{M}$  and that the gaps between machine numbers are bigger for larger numbers:



Non-normalized numbers violate the lower bound for the mantissa  $i$  in Def. 1.5.16.

### (1.5.19) IEEE standard 754 for machine numbers → [60], → link

No surprise: for modern computers  $B = 2$  (binary system), the other parameters of the universally implemented machine number system are

single precision :  $m = 24^*, E \in \{-125, \dots, 128\}$   $\geq 4$  bytes

double precision :  $m = 53^*, E \in \{-1021, \dots, 1024\}$   $\geq 8$  bytes

\*: including bit indicating sign

The standardisation of machine numbers is important, because it ensures that the same numerical algorithm, executed on different computers will nevertheless produce the same result.

### Remark 1.5.20 (Special cases in IEEE standard)

```

1 >> x = exp(1000), y = 3/x, z = x*sin(pi), w = x*log(1)
2   x = Inf
3   y = 0
4   z = Inf
5   w = NaN

```



$E = e_{\max}, M \neq 0 \hat{=} \text{NaN} = \text{Not a number} \rightarrow \text{exception}$

$E = e_{\max}, M = 0 \hat{=} \text{Inf} = \text{Infinity} \rightarrow \text{overflow}$

$E = 0 \hat{=} \text{Non-normalized numbers} \rightarrow \text{underflow}$

$E = 0, M = 0 \hat{=} \text{number } 0$

### (1.5.21) Characteristic parameters of IEEE floating point numbers (double precision)

☞ MATLAB *always* uses double precision according to the IEEE standard.

```

1 >> format hex; realmin, format long; realmin
2 ans = 0010000000000000
3 ans = 2.225073858507201e-308
4 >> format hex; realmax, format long; realmax
5 ans = 7fefffffffffffe
6 ans = 1.797693134862316e+308

```

## 1.5.3 Roundoff errors

### Experiment 1.5.22 (Input errors and roundoff errors)

The following computations would always result in 0, if done in exact arithmetic.

Listing 1.1: input errors and roundoff errors

```

>> format long;
>> a = 4/3; b = a-1; c = 3*b; e = 1-c
e = 2.220446049250313e-16
>> a = 1012/113; b = a-9; c = 113*b; e = 5+c
e = 6.750155989720952e-14

```

```
>> a = 83810206/6789; b = a-12345; c = 6789*b; e = c-1
e = -1.607986632734537e-09
```

Can you devise a similar calculation, whose result is even farther off zero? Apparently the rounding that inevitably accompanies arithmetic operations in  $\mathbb{M}$  can lead to results that are far away from the true result.

For the discussion of errors introduced by rounding we need important notions.

### Definition 1.5.23. Relative error

Let  $\tilde{x} \in \mathbb{K}$  be an approximation of  $x \in \mathbb{K}$ . Then its **absolute error** is given by

$$\epsilon_{\text{abs}} := |x - \tilde{x}|,$$

and its **relative error** is defined as

$$\epsilon_{\text{rel}} := \frac{|x - \tilde{x}|}{|x|}.$$

### Remark 1.5.24 (Relative error and number of correct digits)

The **number of correct** (significant, valid) **digits** of an approximation  $\tilde{x}$  of  $x \in \mathbb{K}$  is defined through the relative error:

If  $\epsilon_{\text{rel}} := \frac{|x - \tilde{x}|}{|x|} \leq 10^{-\ell}$ , then  $\tilde{x}$  has  $\ell$  correct digits,  $\ell \in \mathbb{N}_0$

### (1.5.25) Floating point operations

We may think of the elementary binary operations  $+, -, *, /$  in  $\mathbb{M}$  comprising two steps:

- ① Compute the exact result of the operation.
- ② Perform rounding of the result of ① to map it back to  $\mathbb{M}$ .

### Definition 1.5.26. Correct rounding

Correct rounding (“rounding up”) is given by the function

$$\text{rd} : \left\{ \begin{array}{ccc} \mathbb{R} & \rightarrow & \mathbb{M} \\ x & \mapsto & \max \arg\min_{\tilde{x} \in \mathbb{M}} |x - \tilde{x}| \end{array} \right.$$

(Recall that  $\arg\min_x F(x)$  is the set of arguments of a real valued function  $F$  that makes it attain its (global) minimum.)

Of course, ① above is not possible in a strict sense, but the effect of both steps can be realised and yields

a floating point realization of  $\star \in \{+, -, \cdot, /\}$ .

☞ Notation: write  $\tilde{\star}$  for the floating point realization of  $\star \in \{+, -, \cdot, /\}$ :

Then ❶ and ❷ may be summed up into

$$\text{For } \star \in \{+, -, \cdot, /\}: \quad x \tilde{\star} y := \text{rd}(x \star y) .$$

### Remark 1.5.27 (Breakdown of associativity)

As a consequence of rounding addition  $\tilde{+}$  and multiplication  $\tilde{\cdot}$  as implemented on computers fail to be associative. They will usually be commutative, though this is not guaranteed.

### (1.5.28) Estimating roundoff errors

Let us denote by  $\text{EPS}$  the largest relative error ( $\rightarrow$  Def. 1.5.23) incurred through rounding:

$$\text{EPS} := \max_{x \in I \setminus \{0\}} \frac{|\text{rd}(x) - x|}{|x|} , \quad (1.5.29)$$

where  $I = [\min |\mathbb{M}|, \max |\mathbb{M}|]$  is the range of positive machine numbers.

For machine numbers according to Def. 1.5.16  $\text{EPS}$  can be computed from the defining parameters  $B$  (base) and  $m$  (length of mantissa):

$$\text{EPS} = \frac{1}{2} B^{1-m} . \quad (1.5.30)$$

However, when studying roundoff errors, we do not want to delve into the intricacies of the internal representation of machine numbers. This can be avoided by just using a single bound for the relative error due to rounding, and, thus, also for the relative error potentially suffered in each elementary operation.

### Assumption 1.5.31. “Axiom” of roundoff analysis

There is a small positive number  $\text{EPS}$ , the machine precision, such that for the elementary arithmetic operations  $\star \in \{+, -, \cdot, /\}$  and “hard-wired” functions\*  $f \in \{\exp, \sin, \cos, \log, \dots\}$  holds

$$x \tilde{\star} y = (x \star y)(1 + \delta) \quad , \quad \tilde{f}(x) = f(x)(1 + \delta) \quad \forall x, y \in \mathbb{M} ,$$

with  $|\delta| < \text{EPS}$ .

\*: this is an ideal, which may not be accomplished even by modern CPUs.

► relative roundoff errors of elementary steps in a program bounded by machine precision !

### Example 1.5.32 (Machine precision for IEEE standard)

MATLAB tells the machine precision in the variable `eps`:

#### MATLAB-code 1.5.33: Finding out EPS in MATLAB

```
1 >> format hex; eps, format long; eps
2 ans = 3cb00000000000000
3 ans = 2.220446049250313e-16
```

Knowing the machine precision can be important for checking the validity of computations or coding termination conditions for iterative approximations.

### Experiment 1.5.34 (Adding EPS to 1)

`EPS` is the smallest positive number  $\in \mathbb{M}$  for which  $1 + \tilde{\text{EPS}} \neq 1$  (in  $\mathbb{M}$ ):

```
>> fprintf ('%30.25f\n', 1+0.5*eps)
1.0000000000000000000000000000000
>> fprintf ('%30.25f\n', 1-0.5*eps)
0.999999999999998889776975
>> fprintf ('%30.25f\n', (1+2/eps)-2/eps);
0.0000000000000000000000000000000
```

In fact  $1 + \tilde{\text{EPS}} = 1$  would comply with the “axiom” of roundoff error analysis, Ass. 1.5.31:

$$1 = (1 + \text{EPS})(1 + \delta) \Rightarrow |\delta| = \left| \frac{\text{EPS}}{1 + \text{EPS}} \right| < \text{EPS},$$

$$\frac{2}{\text{EPS}} = (1 + \frac{2}{\text{EPS}})(1 + \delta) \Rightarrow |\delta| = \left| \frac{\text{EPS}}{2 + \text{EPS}} \right| < \text{EPS}.$$

Do we have to worry about these tiny roundoff errors ?



**YES**

(→ Exp. 1.5.5):

- accumulation of roundoff errors
- amplification of roundoff errors

### Remark 1.5.35 (Testing equality with zero)



Since results of numerical computations are almost always polluted by roundoff errors:  
Tests like `if (x == 0)` are pointless and even dangerous, if `x` contains the result  
of a numerical computation.

Remedy: Test `if (abs(x) < eps*s) ...`,  
 $s \hat{=} \text{positive number, compared to which } |x| \text{ should be small.}$

### Remark 1.5.36 (Overflow and underflow)

overflow )  $\hat{=}$  |result of an elementary operation|  $>$  max{|M|}  
 $\hat{=}$  IEEE standard  $\Rightarrow$  Inf  
underflow  $\hat{=}$   $0 < |\text{result of an elementary operation}| < \min\{|M \setminus \{0\}|\}$   
 $\hat{=}$  IEEE standard  $\Rightarrow$  use non-normalized numbers (!)

The Axiom of roundoff analysis Ass. 1.5.31 does *not hold* once non-normalized numbers are encountered:

```
>> format long; res=pi*realmin/123456789101112
res = 5.681754927174335e-322
>> res=res*123456789101112/realmin
res = 3.15248510554597
```

► Try to avoid underflow and overflow

A simple example teaching how to avoid overflow during the computation of the norm of a 2D vector:

$$r = \sqrt{x^2 + y^2}$$

$$r = \begin{cases} |x| \sqrt{1 + (y/x)^2} & , \text{ if } |x| \geq |y| , \\ |y| \sqrt{1 + (x/y)^2} & , \text{ if } |y| > |x| . \end{cases}$$

straightforward evaluation: overflow, when  $|x| > \sqrt{\max |M|}$  or  $|y| > \sqrt{\max |M|}$ .

➤ no overflow!

## 1.5.4 Cancellation

In general, predicting the impact of roundoff errors on the result of a multi-stage computation is very difficult, if possible at all. However, there is a constellation that is particularly prone to dangerous amplification of roundoff and still can be detected easily.

### Example 1.5.37 (Computing the zeros of a quadratic polynomial)

The following simple MATLAB code computes the real roots of a quadratic polynomial  $p(\xi) = \xi^2 + \alpha\xi + \beta$  by the discriminant formula

$$p(\xi_1) = p(\xi_2) = 0 , \quad \xi_{1/2} = \frac{1}{2}(\alpha \pm \sqrt{D}) , \text{ if } D := \alpha^2 - 4\beta \geq 0 . \quad (1.5.38)$$

### MATLAB-code 1.5.39: Discriminant formula for the real roots of $p(\xi) = \xi^2 + \alpha\xi + \beta$

```

1 function z = zerosquadpol(alpha,beta)
2 % MATLAB function computing the zeros of a quadratic polynomial
3 %  $\xi \rightarrow \xi^2 + \alpha\xi + \beta$  by means of the familiar discriminant
4 % formula  $\xi_{1,2} = \frac{1}{2}(-\alpha \pm \sqrt{\alpha^2 - 4\beta})$ . However
5 % this implementation is vulnerable to round-off! The zeros are
6 % returned in a column vector
7 D = alpha^2-4*beta; % discriminant
8 if (D < 0), z = []; % No real zeros
9 else
10     % The famous discriminant formula
11     wD = sqrt(D);
```

```

12 z = 0.5*[-alpha-wD;-alpha+wD];
13 end

```

This formula is applied to the quadratic polynomial  $p(\xi) = (\xi - \gamma)(\xi - \frac{1}{\gamma})$  after its coefficients  $\alpha, \beta$  have been computed from  $\gamma$ , which will have introduced *small* relative roundoff errors (of size  $\text{EPS}$ ).

#### MATLAB-code 1.5.40: Testing the accuracy of computed roots of a quadratic polynomial

```

% MATLAB script for testing the computation of the zeros of a parabola
1 res = []; % array for storing results of computations.
2 gammavec = 2:10:1000; % Define test cases
3 for gamma=gammavec
    % Polynomial p(\xi) = (\xi - \gamma)(\xi - \frac{1}{\gamma})
    4 alpha = -(gamma + 1/gamma); % compute coefficients of polynomial
    5 beta = 1.0;
    6 z1 = zerosquadpol(alpha,beta); % Unstable way to compute zeros
    7 z2 = zerosquadpolstab(alpha,beta); % Stable implementation
    8 % Compute relative errors of the left zero
    9 ztrue = 1/gamma; z2true = gamma;
    10 res = [res; gamma, abs((z1(1)-ztrue)/ztrue), ...
    11       abs((z2(1)-ztrue)/ztrue),abs((z1(2)-z2true)/z2true)];
    12
    13 end
    % Graphical output of relative error of roots computed by unstable
    % implementation
    14 figure; plot(res(:,1),res(:,2),'r+',res(:,1),res(:,4),'b*');
    15 legend('small root','large root','location','best');
    16 xlabel('{\bf \gamma}', 'fontsize',14);
    17 ylabel('{\bf relative errors in \xi_1, \xi_2}', 'fontsize',14);
    18 title('Roots of a parabola computed in an unstable manner');
    19 print -depsc2 '../PICTURES/zqperrinstab.eps';
    % Graphical output of relative errors (comparison)
    20 figure; plot(res(:,1),res(:,2),'r+',res(:,1),res(:,3),'m*');
    21 legend('unstable','stable');
    22 xlabel('{\bf \gamma}', 'fontsize',14);
    23 ylabel('{\bf relative error in \xi_1}', 'fontsize',14);
    24 title('Roundoff in the computation of zeros of a parabola');
    25 print -depsc2 '../PICTURES/zqperr.eps';

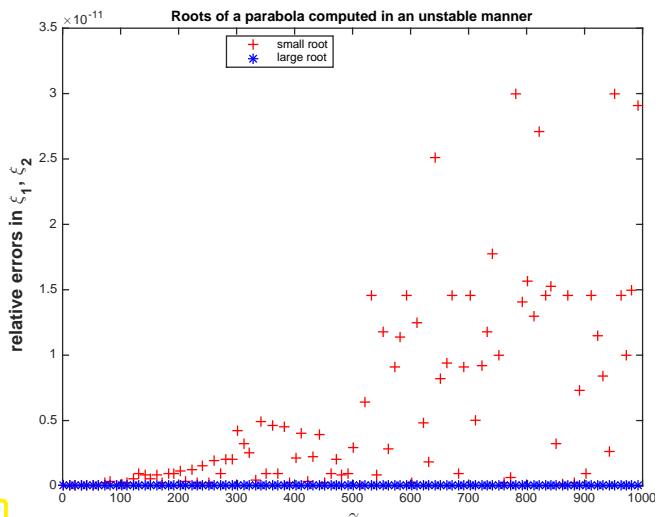
```

Observation:

Roundoff incurred during the computation of  $\alpha$  and  $\beta$  leads to “wrong” roots.

For large  $\gamma$  the computed small root may be fairly inaccurate as regards its *relative error*, which can be several orders of magnitude larger than machine precision `EPS`.

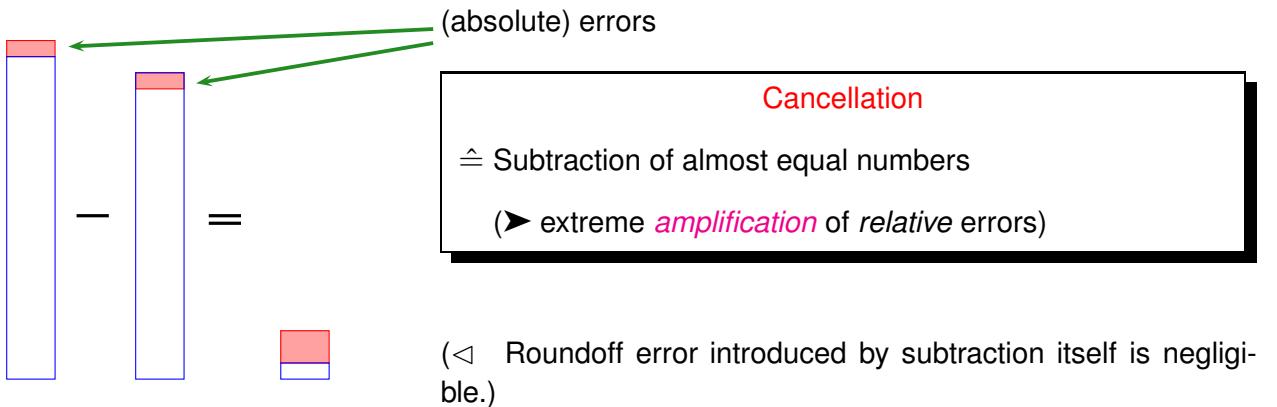
The large root always enjoys a small relative error about the size of `EPS`.



In order to understand why the small root is much more severely affected by roundoff, note that its computation involves the subtraction of two large numbers, if  $\gamma$  is large. This is the typical situation, in which **cancellation** occurs.

#### (1.5.41) Visualisation of cancellation effect

We look at the *exact* subtraction of two almost equal positive numbers both of which have small relative errors (red boxes) with respect to some desired exact value (indicated by blue boxes). The result of the subtraction will be small, but the errors may add up during the subtraction, ultimately constituting a large fraction of the result.



#### Example 1.5.42 (Cancellation in decimal system)

We consider two positive numbers  $x, y$  of about the same size afflicted with relative errors  $\approx 10^{-7}$ . This means that their seventh decimal digits are perturbed, here indicated by \*. When we subtract the two numbers the perturbed digits are shifted to the left, resulting in a possible relative error of  $\approx 10^{-3}$ :

$$\begin{array}{rcl}
 x & = & 0.123467* \\
 y & = & 0.123456* \\
 \hline
 x - y & = & 0.000011* = 0.11*000 \cdot 10^{-4}
 \end{array}$$

← 7th digit perturbed  
 ← 7th digit perturbed  
 ← 3rd digit perturbed  
 ↑ padded zeroes

Again, this example demonstrates that cancellation wreaks havoc through **error amplification**, not through the roundoff error due to the subtraction.

**Example 1.5.43 (Cancellation when evaluating difference quotients → [15, Sect. 8.2.6])**

From analysis we know that the derivative of a differentiable function  $f : I \subset \mathbb{R} \rightarrow \mathbb{R}$  at a point  $x \in I$  is the limit of a **difference quotient**

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}.$$

This suggests the following approximation of the derivative by a difference quotient with small but finite  $h > 0$

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad \text{for } |h| \ll 1.$$

Results from analysis tell us that the **approximation error** should tend to zero for  $h \rightarrow 0$ . More precise quantitative information is provided by the Taylor formula for a twice continuously differentiable function

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(\xi)h^2 \quad \text{for some } \xi = \xi(x, h) \in [\min\{x, x+h\}, \max\{x, x+h\}], \quad (1.5.44)$$

from which we infer

$$\frac{f(x+h) - f(x)}{h} - f'(x) = \frac{1}{2}hf''(\xi) \quad \text{for some } \xi = \xi(x, h) \in [\min\{x, x+h\}, \max\{x, x+h\}]. \quad (1.5.45)$$

We investigate the approximation of the derivative by difference quotients for  $f = \exp$ ,  $x = 0$ , and different values of  $h > 0$ :

**MATLAB-code 1.5.46: Difference quotient approximation of the derivative of exp**

```

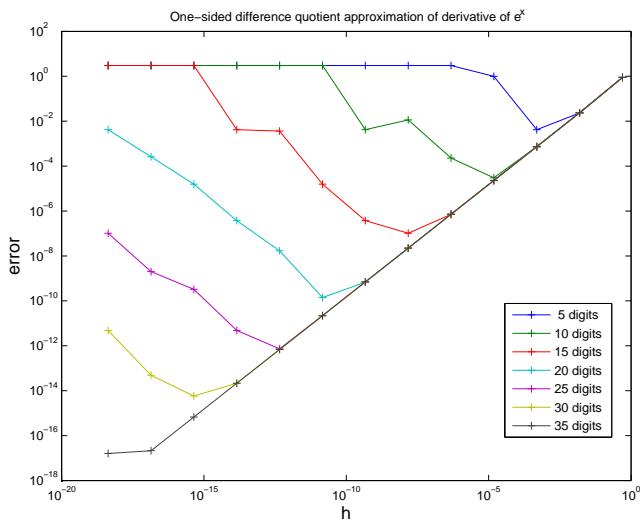
1 h = 0.1; x = 0.0;
2 for i = 1:16
3 df = (exp(x+h)-exp(x))/h;
4 fprintf('%d %16.14f\n', i, df);
5 h = h*0.1;
6 end

```

Measured relative errors ▷

We observe an initial decrease of the relative approximation error followed by a steep increase when  $h$  drops below  $10^{-8}$ .

$\log_{10}(h)$	relative error
-1	0.05170918075648
-2	0.00501670841679
-3	0.00050016670838
-4	0.00005000166714
-5	0.00000500000696
-6	0.00000049996218
-7	0.00000004943368
-8	-0.00000000607747
-9	0.00000008274037
-10	0.00000008274037
-11	0.00000008274037
-12	0.00008890058234
-13	-0.00079927783736
-14	-0.00079927783736
-15	0.11022302462516
-16	-1.0000000000000000



That the observed errors are really due to round-off errors is confirmed by the following numerical results reported besides, using a variable precision floating point extension of MATLAB, the [Advanpix extended precision library](#) for MATLAB.

#### MATLAB-code 1.5.47: Evaluation of difference quotients with variable precision

```

1 function experr = numericaldifferentiation()
2 % Numerical differentiation of exponential function with extended
3 % precision arithmetic
4 % Uses the Advanpix extended precision library for MATLAB,
5 % www.advanpix.com
6
7 mpstartup; % Initialization of multi-precision library
8 experr = [] ; l = 1;
9 for ndigits=[5 10 15 20 25 30 35]
10 mp.Digits(ndigits), % Set number of digits
11 x = mp(1.1); % Evaluation point in extended precision
12 h = mp(2).^-[-1:-5:-61]; % width of difference quotient in extended
13 % precision
14 experr = [experr; abs(((exp(x+h)-exp(x))./h) - exp(x))]; % %
15 % compute (absolute) error
16 leg{l} = sprintf('%2.0d digits',ndigits);
17 l = l+1;
18 end
19
20 % Graphical output of error
21 figure('name','numdiff');
22 loglog(h,experr,'+-');
23 title('One-sided difference quotient approximation of derivative
24 of e^x');
25 xlabel('h','fontsize',14);
26 ylabel('error','fontsize',14);
27 legend(leg,'Location','best');
28 print -depsc2 'expnumdiffmultiprecision.eps';

```

- Obvious culprit: **cancellation** when computing the numerator of the difference quotient for small  $|h|$  leads to a strong amplification of inevitable errors introduced by the evaluation of the transcendent exponential function.

We witness the competition of two opposite effects: Smaller  $h$  results in a better approximation of the

derivative by the difference quotient, but the impact of cancellation is the stronger the smaller  $|h|$ .

$$\text{Approximation error } f'(x) - \frac{f(x+h) - f(x)}{h} \rightarrow 0 \quad \left. \begin{array}{l} \text{Impact of roundoff} \rightarrow \infty \\ \end{array} \right\} \text{ as } h \rightarrow 0 .$$

In order to provide a rigorous underpinning for our conjecture, in this example we embark on our first **roundoff error analysis** merely based on the “Axiom of roundoff analysis” Ass. 1.5.31: As in the computational example above we study the approximation of  $f'(x) = e^x$  for  $f = \exp$ ,  $x \in \mathbb{R}$ .

$$\begin{aligned} df &= \frac{e^{x+h} (1 + \delta_1) - e^x (1 + \delta_2)}{h} && \text{correction factors take into account roundoff:} \\ &= e^x \left( \frac{e^h - 1}{h} + \frac{\delta_1 e^h - \delta_2}{h} \right) && (\rightarrow \text{"axiom of roundoff analysis", Ass. 1.5.31}) \\ &\Rightarrow |df| \leq e^x \left( \frac{e^h - 1}{h} + \text{eps} \frac{1+e^h}{h} \right) && |\delta_1|, |\delta_2| \leq \text{eps} . \end{aligned}$$

$1 + O(h)$        $O(h^{-1})$     for  $h \rightarrow 0$

(Note that the estimate for the term  $(e^h - 1)/h$  is a particular case of (1.5.45).)

► relative error:  $\left| \frac{e^x - df}{e^x} \right| \approx h + \frac{2\text{eps}}{h} \rightarrow \min \text{ for } h = \sqrt{2 \text{eps}} .$

In double precision:  $\sqrt{2 \text{eps}} = 2.107342425544702 \cdot 10^{-8}$

### Remark 1.5.48 (Cancellation during the computation of relative errors)

In the numerical experiment of Ex. 1.5.43 we computed the relative error of the result by subtraction, see Code 1.5.46. Of course, massive cancellation will occur! Do we have to worry?

In this case cancellation can be tolerated, because we are *interested only* in the **magnitude** of the relative error. Even if it was affected by a large relative error, this information is still not compromised.

For example, if the relative error has the exact value  $10^{-8}$ , but can be computed only with a huge relative error of 10%, then the perturbed value would still be in the range  $[0.9 \cdot 10^{-8}, 1.1 \cdot 10^{-8}]$ . Therefore it will still have the correct magnitude and still permit us to conclude the number of valid digits correctly.

### Remark 1.5.49 (Cancellation in Gram-Schmidt orthogonalisation of Exp. 1.5.5)

The matrix created by the MATLAB command  $A = \text{hilb}(10)$ , the so-called **Hilbert matrix**, has columns

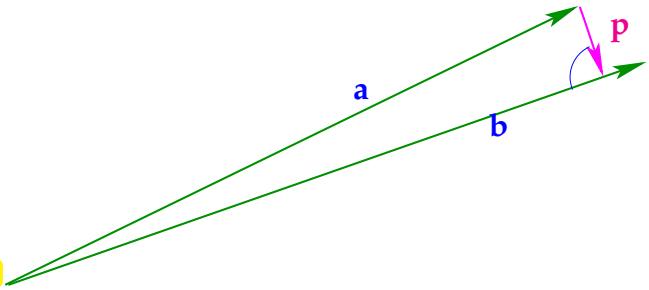
that are almost linearly dependent.

Cancellation when computing orthogonal projection  
of vector  $\mathbf{a}$  onto space spanned by vector  $\mathbf{b}$  ▷

$$\mathbf{p} = \mathbf{a} - \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{b} \cdot \mathbf{b}} \mathbf{b}.$$

If  $\mathbf{a}, \mathbf{b}$  point in almost the same direction,  $\|\mathbf{p}\| \ll \|\mathbf{a}\|, \|\mathbf{b}\|$ , so that a “tiny” vector  $\mathbf{p}$  is obtained by subtracting two “long” vectors, which implies cancellation.

Fig. 11



This can happen in Line 8 of Code 1.5.4.

### (1.5.50) Avoiding disastrous cancellation

The following examples demonstrate a few fundamental techniques for steering clear of cancellation by using alternative formulas that yield the same value (in exact arithmetic), but do not entail subtracting two numbers of almost equal size.

#### Example 1.5.51 (Stable discriminant formula → Ex. 1.5.37)

If  $\xi_1$  and  $\xi_2$  are the two roots of the quadratic polynomial  $p(\xi) = \xi^2 + \alpha\xi + \beta$ , then  $\xi_1 \cdot \xi_2 = \beta$  (Vieta's formula). Thus once we have computed a root, we can obtain the other by simple division.



Idea:

- ① Depending on the sign of  $\alpha$  compute “stable root” without cancellation.
- ② Compute other root from Vieta’s formula

**MATLAB-code 1.5.52: Stable computation of real root of a quadratic polynomial**

```

1 function z = zerosquadpolstab(alpha , beta)
2 % MATLAB function computing the real zeros of a quadratic polynomial
3 %  $\xi \rightarrow \xi^2 + \alpha\xi + \beta$  by means of the familiar discriminant
4 % formula  $\xi_{1,2} = \frac{1}{2}(-\alpha \pm \sqrt{\alpha^2 - 4\beta})$  and
5 % This is a stable implementation based on Vieta's theorem.
6 D = alpha^2-4*beta; % discriminant
7 if (D < 0), z = [];
8 else
9     wD = sqrt(D);
10    % Use discriminant formula only for zero far away from 0
11    % in order to avoid cancellation. For the other zero
12    % use Vieta's formula.
13    if (alpha >= 0)
14        t = 0.5*(-alpha-wD); %
15        z = [t; beta/t];
16    else
17        t = 0.5*(-alpha+wD); %
18        z = [beta/t;t];
19    end
20 end

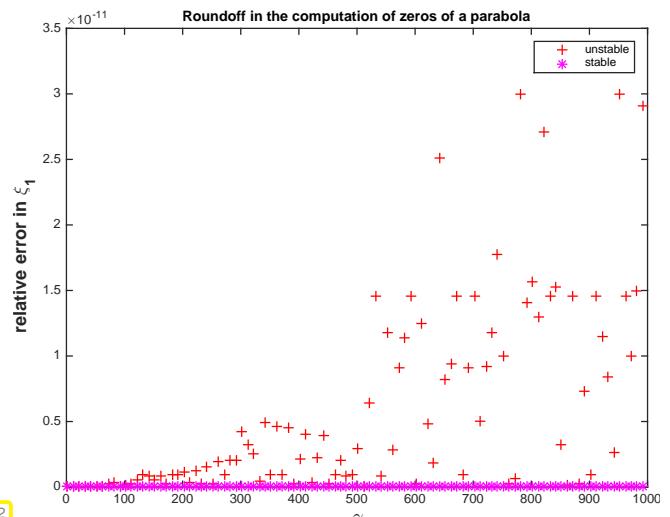
```

→ Invariably, we add numbers with the same sign in Line 14 and Line 17.

Numerical experiment based on the driver code Code 1.5.40.

Observation:

The new code can also compute the small root of the polynomial  $p(\xi) = (\xi - \gamma)(\xi - \frac{1}{\gamma})$  (expanded in monomials) with a relative error  $\approx \text{EPS}$ .

**Example 1.5.53 (Exploiting trigonometric identities to avoid cancellation)**

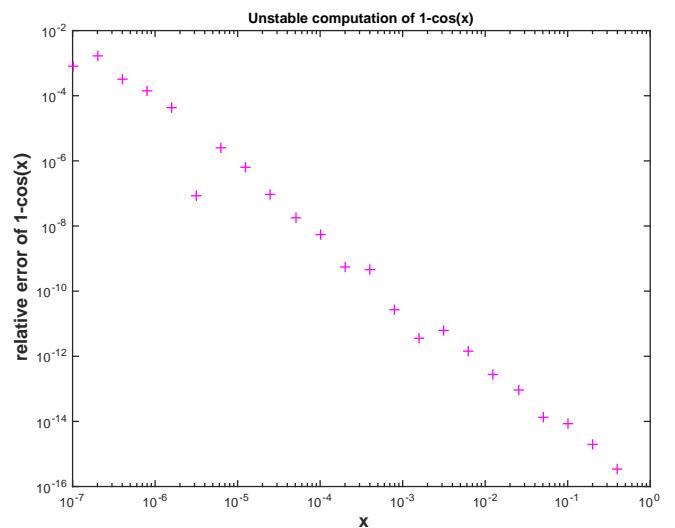
The task is to evaluate the integral

$$\int_0^x \sin t dt = \underbrace{1 - \cos x}_I = \underbrace{2 \sin^2(x/2)}_{II} \quad \text{for } 0 < x \ll 1, \quad (1.5.54)$$

and this can be done by the two different formulas  $I$  and  $II$ .

Relative error of expression  $I$  ( $1 - \cos(x)$ ) with respect to equivalent expression  $II$  ( $2 \cdot \sin(x/2)^2$ )

▷ Expression  $I$  is affected by cancellation for  $|x| \ll 1$ , since then  $\cos x \approx 1$ , whereas expression  $II$  can be evaluated with a relative error  $\approx \text{EPS}$  for all  $x$ .

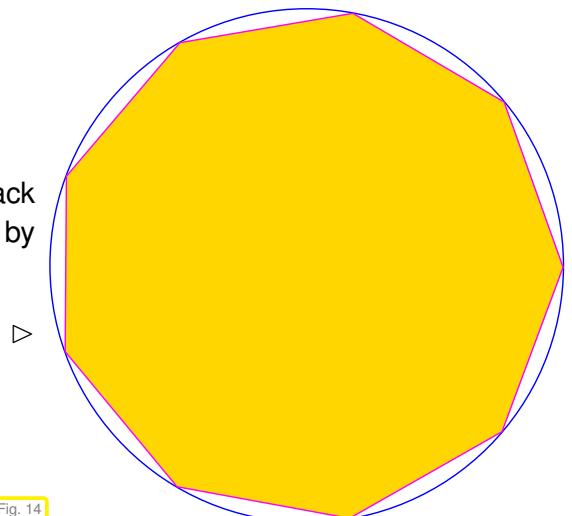


Analytic manipulations offer ample opportunity to rewrite expressions in equivalent form immune to cancellation.

### Example 1.5.55 (Switching to equivalent formulas to avoid cancellation)

Now we see an example of a computation allegedly dating back to Archimedes, who tried to approximate the area of a circle by the areas of inscribed regular polygons.

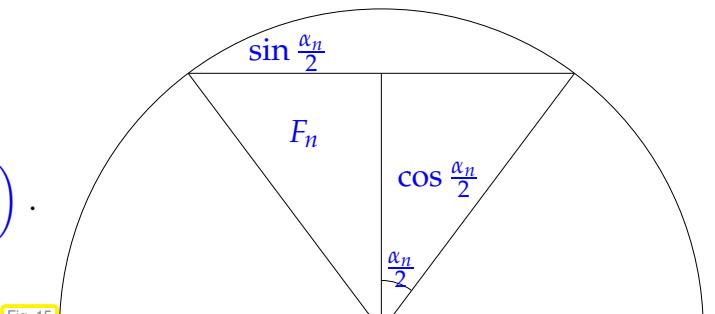
Approximation of circle by regular  $n$ -gon,  $n \in \mathbb{N}$



Area of  $n$ -gon:

$$A_n = n \cos \frac{\alpha_n}{2} \sin \frac{\alpha_n}{2} = \frac{n}{2} \sin \alpha_n = \frac{n}{2} \sin\left(\frac{2\pi}{n}\right).$$

Recursion formula for  $A_n$  derived from



$$\sin \frac{\alpha_n}{2} = \sqrt{\frac{1 - \cos \alpha_n}{2}} = \sqrt{\frac{1 - \sqrt{1 - \sin^2 \alpha_n}}{2}},$$

$$\text{Initial approximation: } A_6 = \frac{3}{2} \sqrt{3}.$$

**MATLAB-code 1.5.56: Tentative computation of circumference of regular polygon**

```

1 tol = 1e-8                      % set tolerance
2 s = sqrt(3)/2; An=3*s; n = 6;    % initialization (hexagon case)
3 Z = [n An An-pi s];             % matrix for storing results
4 while (s > tol)                 % terminate when s is 'small enough'
5     s = sqrt((1-sqrt(1-s*s))/2); % recursion for area
6     n=2*n; An = n/2*s;           % new estimate for circumference
7     Z = [Z; n An An-pi s];       % store results and (absolute) )error
8 end

```

The approximation deteriorates after applying the recursion formula many times:

<i>n</i>	<i>A<sub>n</sub></i>	<i>A<sub>n</sub> - π</i>	$\sin \alpha_n$
6	2.598076211353316	-0.543516442236477	0.866025403784439
12	3.000000000000000	-0.141592653589794	0.500000000000000
24	3.105828541230250	-0.035764112359543	0.258819045102521
48	3.132628613281237	-0.008964040308556	0.130526192220052
96	3.139350203046872	-0.002242450542921	0.065403129230143
192	3.141031950890530	-0.000560702699263	0.032719082821776
384	3.141452472285344	-0.000140181304449	0.016361731626486
768	3.141557607911622	-0.000035045678171	0.008181139603937
1536	3.141583892148936	-0.000008761440857	0.004090604026236
3072	3.141590463236762	-0.000002190353031	0.002045306291170
6144	3.141592106043048	-0.000000547546745	0.001022653680353
12288	3.141592516588155	-0.000000137001638	0.000511326906997
24576	3.141592618640789	-0.000000034949004	0.000255663461803
49152	3.141592645321216	-0.000000008268577	0.000127831731987
98304	3.141592645321216	-0.000000008268577	0.000063915865994
196608	3.141592645321216	-0.000000008268577	0.000031957932997
393216	3.141592645321216	-0.000000008268577	0.000015978966498
786432	3.141593669849427	0.000001016259634	0.000007989485855
1572864	3.141592303811738	-0.000000349778055	0.000003994741190
3145728	3.141608696224804	0.000016042635011	0.000001997381017
6291456	3.141586839655041	-0.0000005813934752	0.000000998683561
12582912	3.141674265021758	0.000081611431964	0.000000499355676
25165824	3.141674265021758	0.000081611431964	0.000000249677838
50331648	3.143072740170040	0.001480086580246	0.000000124894489
100663296	3.159806164941135	0.018213511351342	0.000000062779708
201326592	3.181980515339464	0.040387861749671	0.000000031610136
402653184	3.354101966249685	0.212509312659892	0.000000016660005
805306368	4.242640687119286	1.101048033529493	0.000000010536712
1610612736	6.000000000000000	2.858407346410207	0.000000007450581

Where does cancellation occur in Line 5 of Code 1.5.56? Since  $s \ll 1$ , computing  $1 - s$  will not trigger cancellation. However, the subtraction  $1 - \sqrt{1 - s^2}$  will, because  $\sqrt{1 - s^2} \approx 1$  for  $s \ll 1$ :

$$\sin \frac{\alpha_n}{2} = \sqrt{\frac{1 - \cos \alpha_n}{2}} = \sqrt{\frac{1 - \sqrt{1 - \sin^2 \alpha_n}}{2}}$$

For  $\alpha_n \ll 1$ :  $\sqrt{1 - \sin^2 \alpha_n} \approx 1$

Cancellation here!

We arrive at an *equivalent* formula not vulnerable to cancellation essentially using the identity  $(a + b)(a -$

b)  $= a^2 - b^2$  in order to eliminate the difference of square roots in the numerator.

$$\begin{aligned}\sin \frac{\alpha_n}{2} &= \sqrt{\frac{1 - \sqrt{1 - \sin^2 \alpha_n}}{2}} = \sqrt{\frac{1 - \sqrt{1 - \sin^2 \alpha_n}}{2} \cdot \frac{1 + \sqrt{1 - \sin^2 \alpha_n}}{1 + \sqrt{1 - \sin^2 \alpha_n}}} \\ &= \sqrt{\frac{1 - (1 - \sin^2 \alpha_n)}{2(1 + \sqrt{1 - \sin^2 \alpha_n})}} = \frac{\sin \alpha_n}{\sqrt{2(1 + \sqrt{1 - \sin^2 \alpha_n})}}.\end{aligned}$$

### MATLAB-code 1.5.57: Stable recursion for area of regular $n$ -gon

```

1 s = sqrt(3)/2; An=3*s; n = 6; % Initialization (hexagon)
2 Z = [n A An-pi s]; % Matrix for storing results
3 while (s > tol) % Terminate when tolerance is reached
4     s = s/sqrt(2*(1+sqrt((1+s)*(1-s)))); % Stable recursion without
        cancellation
5     n=2*n; An = n/2*s; % Compute area of polygon
6     Z = [Z; n An An-pi s]; % store results
7 end

```

Using the stable recursion, we observe better approximation for polygons with more corners:

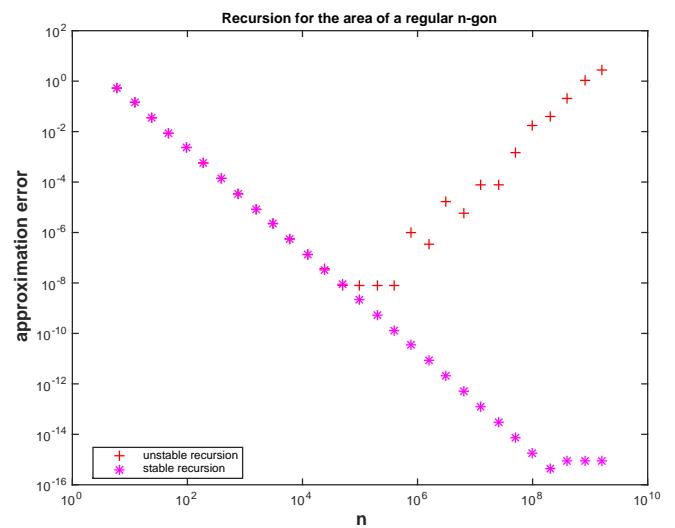
$n$	$A_n$	$A_n - \pi$	$\sin \alpha_n$
6	2.598076211353316	-0.543516442236477	0.866025403784439
12	3.000000000000000	-0.141592653589793	0.500000000000000
24	3.105828541230249	-0.035764112359544	0.258819045102521
48	3.132628613281238	-0.008964040308555	0.130526192220052
96	3.139350203046867	-0.002242450542926	0.065403129230143
192	3.141031950890509	-0.000560702699284	0.032719082821776
384	3.141452472285462	-0.000140181304332	0.016361731626487
768	3.141557607911857	-0.000035045677936	0.008181139603937
1536	3.141583892148318	-0.000008761441475	0.004090604026235
3072	3.141590463228050	-0.000002190361744	0.002045306291164
6144	3.141592105999271	-0.000000547590522	0.001022653680338
12288	3.141592516692156	-0.000000136897637	0.000511326907014
24576	3.141592619365383	-0.000000034224410	0.000255663461862
49152	3.141592645033690	-0.000000008556103	0.000127831731976
98304	3.141592651450766	-0.000000002139027	0.000063915866118
196608	3.141592653055036	-0.000000000534757	0.000031957933076
393216	3.141592653456104	-0.000000000133690	0.000015978966540
786432	3.141592653556371	-0.000000000033422	0.000007989483270
1572864	3.141592653581438	-0.000000000008355	0.000003994741635
3145728	3.141592653587705	-0.000000000002089	0.000001997370818
6291456	3.141592653589271	-0.000000000000522	0.000000998685409
12582912	3.141592653589663	-0.000000000000130	0.000000499342704
25165824	3.141592653589761	-0.000000000000032	0.000000249671352
50331648	3.141592653589786	-0.000000000000008	0.000000124835676
100663296	3.141592653589791	-0.000000000000002	0.000000062417838
201326592	3.141592653589794	0.000000000000000	0.000000031208919
402653184	3.141592653589794	0.000000000000001	0.000000015604460
805306368	3.141592653589794	0.000000000000001	0.000000007802230
1610612736	3.141592653589794	0.000000000000001	0.000000003901115

Plot of errors for approximations of  $\pi$  as computed by the two algebraically equivalent recursion formulas▷

Observation, cf. Ex. 1.5.43

Amplified roundoff errors due to cancellation supersedes approximation error for  $n \geq 10^5$ .

Roundoff errors merely of magnitude  $\text{EPS}$  in the case of stable recursion



### Example 1.5.58 (Summation of exponential series)

In principle, the function value  $\exp(x)$  can be approximated up to any accuracy by summing sufficiently many terms of the globally convergent exponential series.

$$\begin{aligned}\exp(x) &= \sum_{k=0}^{\infty} \frac{x^k}{k!} \\ &= 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots\end{aligned}$$

### MATLAB-code 1.5.59: Summation of exponential series

```

1 function y = expeval(x,tol)
2 % Initialization
3 y=1; term=1; k=1;
4 % Termination
5 while
6   (abs(term)>tol*min(y,1))
7     % Next summand
8     term = term*x/k;
9     % Summation
10    y = y + term; %
11    k = k+1;
12 end
```

Results for  $\text{tol} = 10^{-8}$ ,  $\widetilde{\exp}$  designates the approximate value for  $\exp(x)$  returned by the function from Code 1.5.59. Rightmost column lists relative errors, which tells us the number of valid digits in the approximate result.

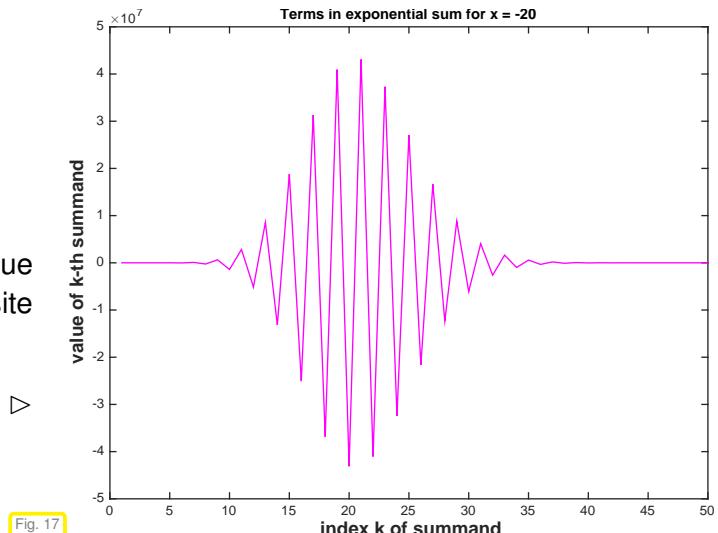
$x$	Approximation $\widetilde{\exp}(x)$	$\exp(x)$	$\frac{ \exp(x) - \widetilde{\exp}(x) }{\exp(x)}$
-20	5.6218844674e-09	2.0611536224e-09	1.727542676201181
-18	1.5385415977e-08	1.5229979745e-08	0.010205938187564
-16	1.1254180496e-07	1.1253517472e-07	0.000058917020257
-14	8.3152907681e-07	8.3152871910e-07	0.000000430176956
-12	6.1442133148e-06	6.1442123533e-06	0.000000156480737
-10	4.5399929556e-05	4.5399929762e-05	0.000000004544414
-8	3.3546262817e-04	3.3546262790e-04	0.000000000788902
-6	2.4787521758e-03	2.4787521767e-03	0.000000000333306
-4	1.8315638879e-02	1.8315638889e-02	0.000000000530694
-2	1.3533528320e-01	1.3533528324e-01	0.000000000273603
0	1.0000000000e+00	1.0000000000e+00	0.000000000000000
2	7.3890560954e+00	7.3890560989e+00	0.000000000479969
4	5.4598149928e+01	5.4598150033e+01	0.0000000001923058
6	4.0342879295e+02	4.0342879349e+02	0.0000000001344248
8	2.9809579808e+03	2.9809579870e+03	0.0000000002102584
10	2.2026465748e+04	2.2026465795e+04	0.0000000002143800
12	1.6275479114e+05	1.6275479142e+05	0.0000000001723845
14	1.2026042798e+06	1.2026042842e+06	0.0000000003634135
16	8.8861105010e+06	8.8861105205e+06	0.0000000002197990
18	6.5659968911e+07	6.5659969137e+07	0.0000000003450972
20	4.8516519307e+08	4.8516519541e+08	0.0000000004828738

Observation:

Large relative approximation errors for  $x \ll 0$ .

For  $x \ll 0$  we have  $|\exp(x)| \ll 1$ , but this value is computed by summing large numbers of opposite sign.

Terms summed up for  $x = -20$



Remedy: Cancellation can be avoided by using identity



$$\exp(x) = \frac{1}{\exp(-x)} \quad , \text{ if } x < 0 .$$

### Example 1.5.60 (Combat cancellation by approximation)

In a computer code we have to provide a routine for the evaluation of

$$\int_0^1 e^{at} dt = \frac{\exp(a) - 1}{a} \quad \text{for any } a > 0 , \quad (1.5.61)$$

cf. the discussion of cancellation in the context of numerical differentiation in Ex. 1.5.43. There we

observed massive cancellation.

Recall the **Taylor expansion** formula in one dimension for a function that is  $m + 1$  times continuously differentiable in a neighborhood of  $x$  [77, Satz 5.5.1]

$$f(x+h) = \sum_{k=0}^m \frac{1}{k!} f^{(k)}(x) h^k + R_m(x, h), \quad R_m(x, h) = \frac{1}{(m+1)!} f^{(m+1)}(\xi) h^{m+1},$$

for some  $\xi \in [\min\{x, x+h\}, \max\{x, x+h\}]$ , and for all sufficiently small  $|h|$ . Here  $R(x, h)$  is called the **remainder** term and  $f^{(k)}$  denotes the  $k$  derivative of  $f$ .

Cancellation in (1.5.61) can be avoided by replacing  $\exp(a)$ ,  $a > 0$ , with a suitable Taylor expansion around  $a = 0$  and then dividing by  $a$ :

$$\frac{\exp(a) - 1}{a} = \sum_{k=0}^m \frac{1}{(k+1)!} a^k + R_m(a), \quad R_m(a) = \frac{1}{(m+1)!} \exp(\xi) a^{m+1} \text{ for some } 0 \leq \xi \leq a.$$

Issue: How to choose the number  $m$  of terms to be retained in the Taylor expansion? We have to pick  $m$  large enough such that the relative approximation error remains below a prescribed threshold  $\text{tol}$ . To estimate the relative approximation error, we use the expression for the remainder together with the simple estimate  $(\exp(a) - 1)/a > 1$  for all  $a > 0$ :

$$\text{rel. err.} = \frac{(e^a - 1)/a - \sum_{k=0}^m \frac{1}{(k+1)!} a^k}{(e^a - 1)/a} \leq \frac{1}{(m+1)!} \exp(\xi) a^m \leq \frac{1}{(m+1)!} \exp(a) a^m.$$

For  $a = 10^{-3}$  we get

$m$	1	2	3	4	5
	1.0010e-03	5.0050e-07	1.6683e-10	4.1708e-14	8.3417e-18

Hence, keeping  $m = 3$  terms is enough for achieving about 10 valid digits.

Relative error of unstable formula  $(\exp(a) - 1.0)/a$  and relative error, when using a Taylor expansion approximation for small  $a$  ▷

```

if (abs(a) < 1E-3)
    v = 1.0 + (1.0/2 + 1.0/6*a)*a;
else
    v = (exp(a)-1.0)/a;
end

```

Error computed by comparison with matlab's built-in function `expml` that provides a stable implementation of  $\exp(x) - 1$ .

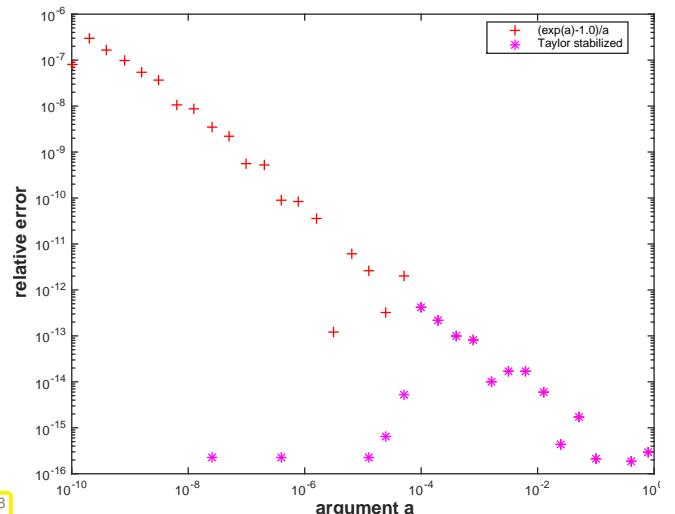


Fig. 18

## 1.5.5 Numerical stability

We have seen that a particular “problem” can be tackled by different “algorithms”, which produce different results due to roundoff errors. This section will clarify what distinguishes a “good” algorithm from a rather abstract point of view.

**(1.5.62) The “problem”**

A mathematical notion of “**problem**”:

- \* data space  $X$ , usually  $X \subset \mathbb{R}^n$
- \* result space  $Y$ , usually  $Y \subset \mathbb{R}^m$
- \* mapping (problem function)  $F : X \mapsto Y$

A problem is a well defined *function* that assigns to each datum a result.

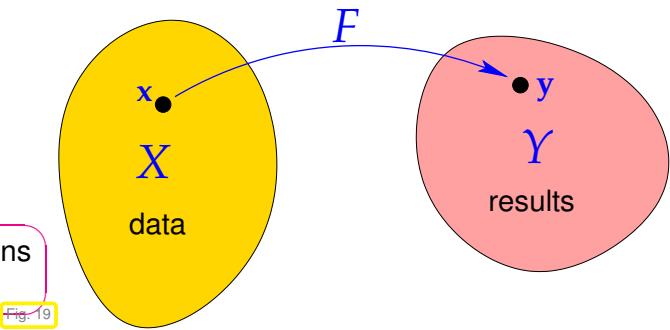


Fig. 1.9

Note: In this course, both the data space  $X$  and the result space  $Y$  will always be subsets of finite dimensional **vector spaces**.

**Example 1.5.63 (The “matrix  $\times$  vector-multiplication problem”)**

We consider the “problem” of computing the product  $\mathbf{Ax}$  for a given matrix  $\mathbf{A} \in \mathbb{K}^{m,n}$  and a given vector  $\mathbf{x} \in \mathbb{K}^n$ .

- > • Data space  $X = \mathbb{K}^{m,n} \times \mathbb{K}^n$  (input is a matrix and a vector)
- Result space  $Y = \mathbb{R}^m$  (space of column vectors)
- Problem function  $F : X \rightarrow Y$ ,  $F(\mathbf{a}, \mathbf{x}) := \mathbf{Ax}$

**(1.5.64) Norms on spaces of vectors and matrices**

Norms provide tools for measuring errors. Recall from linear algebra and calculus [59, Sect. 4.3], [34, Sect. 6.1]:

**Definition 1.5.65. Norm**

$X$  = vector space over field  $\mathbb{K}$ ,  $\mathbb{K} = \mathbb{C}, \mathbb{R}$ . A map  $\|\cdot\| : X \mapsto \mathbb{R}_0^+$  is a **norm** on  $X$ , if it satisfies

- (i)  $\forall \mathbf{x} \in X : \mathbf{x} \neq 0 \Leftrightarrow \|\mathbf{x}\| > 0$  (definite),
- (ii)  $\|\lambda \mathbf{x}\| = |\lambda| \|\mathbf{x}\| \quad \forall \mathbf{x} \in X, \lambda \in \mathbb{K}$  (homogeneous),
- (iii)  $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\| \quad \forall \mathbf{x}, \mathbf{y} \in X$  (triangle inequality).

Examples: (for vector space  $\mathbb{K}^n$ , vector  $\mathbf{x} = (x_1, x_2, \dots, x_n)^T \in \mathbb{K}^n$ )

name	:	definition	MATLAB function
Euclidean norm	:	$\ \mathbf{x}\ _2 := \sqrt{ x_1 ^2 + \dots +  x_n ^2}$	<code>norm(x)</code>
1-norm	:	$\ \mathbf{x}\ _1 :=  x_1  + \dots +  x_n $	<code>norm(x, 1)</code>
$\infty$ -norm, max norm	:	$\ \mathbf{x}\ _\infty := \max\{ x_1 , \dots,  x_n \}$	<code>norm(x, inf)</code>

**Remark 1.5.66 (Inequalities between vector norms)**

All norms on the vector space  $\mathbb{K}^n$ ,  $n \in \mathbb{N}$ , are **equivalent** in the sense that for arbitrary two norms  $\|\cdot\|_1$  and  $\|\cdot\|_2$  we can always find a constant  $C > 0$  such that

$$\|\mathbf{v}\|_1 \leq C \|\mathbf{v}\|_2 \quad \forall \mathbf{v} \in \mathbb{K}^n. \quad (1.5.67)$$

Of course, the constant  $C$  will usually depend on  $n$  and the norms under consideration.

For the vector norms introduced above, explicit expressions for the constants “ $C$ ” are available: for all  $\mathbf{x} \in \mathbb{K}^n$

$$\|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1 \leq \sqrt{n}\|\mathbf{x}\|_2, \quad (1.5.68)$$

$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \sqrt{n}\|\mathbf{x}\|_\infty, \quad (1.5.69)$$

$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_1 \leq n\|\mathbf{x}\|_\infty. \quad (1.5.70)$$

The matrix space  $\mathbb{K}^{m,n}$  is a vector space, of course, and can also be equipped with various norms. Of particular importance are norms *induced by vector norms* on  $\mathbb{K}^n$  and  $\mathbb{K}^m$ .

### Definition 1.5.71. Matrix norm

Given vector norms  $\|\cdot\|_1$  and  $\|\cdot\|_2$  on  $\mathbb{K}^n$  and  $\mathbb{K}^m$ , respectively, the associated **matrix norm** is defined by

$$\mathbf{M} \in \mathbb{R}^{m,n}: \quad \|\mathbf{M}\| := \sup_{\mathbf{x} \in \mathbb{R}^n \setminus \{\mathbf{0}\}} \frac{\|\mathbf{M}\mathbf{x}\|_2}{\|\mathbf{x}\|_1}.$$

By virtue of definition the matrix norms enjoy an important property, they are **sub-multiplicative**:

$$\forall \mathbf{A} \in \mathbb{K}^{n,m}, \mathbf{B} \in \mathbb{K}^{m,k}: \quad \|\mathbf{AB}\| \leq \|\mathbf{A}\| \|\mathbf{B}\|. \quad (1.5.72)$$

 notations for matrix norms for *quadratic matrices* associated with standard vector norms:

$$\|\mathbf{x}\|_2 \rightarrow \|\mathbf{M}\|_2, \quad \|\mathbf{x}\|_1 \rightarrow \|\mathbf{M}\|_1, \quad \|\mathbf{x}\|_\infty \rightarrow \|\mathbf{M}\|_\infty$$

### Example 1.5.73 (Matrix norm associated with $\infty$ -norm and 1-norm)

Rather simple formulas are available for the matrix norms induced by the vector norms  $\|\cdot\|_\infty$  and  $\|\cdot\|_1$

$$\begin{aligned} \text{e.g. for } \mathbf{M} \in \mathbb{K}^{2,2}: \quad & \|\mathbf{M}\|_\infty = \max\{|m_{11}x_1 + m_{12}x_2|, |m_{21}x_1 + m_{22}x_2|\} \\ & \leq \max\{|m_{11}| + |m_{12}|, |m_{21}| + |m_{22}|\} \|\mathbf{x}\|_\infty, \\ & \|\mathbf{M}\|_1 = |m_{11}x_1 + m_{12}x_2| + |m_{21}x_1 + m_{22}x_2| \\ & \leq \max\{|m_{11}| + |m_{21}|, |m_{12}| + |m_{22}|\}(|x_1| + |x_2|). \end{aligned}$$

For general  $\mathbf{M} \in \mathbb{K}^{m,n}$

$$\gg \text{matrix norm} \leftrightarrow \|\cdot\|_\infty = \text{row sum norm} \quad \|\mathbf{M}\|_\infty := \max_{i=1,\dots,m} \sum_{j=1}^n |m_{ij}|, \quad (1.5.74)$$

$$\gg \text{matrix norm} \leftrightarrow \|\cdot\|_1 = \text{column sum norm} \quad \|\mathbf{M}\|_1 := \max_{j=1,\dots,n} \sum_{i=1}^m |m_{ij}|. \quad (1.5.75)$$

Sometimes special formulas for the Euclidean matrix norm come handy [28, Sect. 2.3.3]:

**Lemma 1.5.76. Formula for Euclidean norm of a Hermitian matrix**

$$\mathbf{A} \in \mathbb{K}^{n,n}, \mathbf{A} = \mathbf{A}^H \Rightarrow \|\mathbf{A}\|_2 = \max_{\mathbf{x} \neq 0} \frac{|\mathbf{x}^H \mathbf{A} \mathbf{x}|}{\|\mathbf{x}\|_2^2}.$$

*Proof.* Recall from linear algebra: Hermitian matrices (a special class of normal matrices) enjoy unitary similarity to diagonal matrices:

$$\exists \mathbf{U} \in \mathbb{K}^{n,n}, \text{ diagonal } \mathbf{D} \in \mathbb{R}^{n,n}: \mathbf{U}^{-1} = \mathbf{U}^H \quad \text{and} \quad \mathbf{A} = \mathbf{U}^H \mathbf{D} \mathbf{U}.$$

Since multiplication with an unitary matrix preserves the 2-norm of a vector, we conclude

$$\|\mathbf{A}\|_2 = \left\| \mathbf{U}^H \mathbf{D} \mathbf{U} \right\|_2 = \|\mathbf{D}\|_2 = \max_{i=1,\dots,n} |d_i|, \quad \mathbf{D} = \text{diag}(d_1, \dots, d_n).$$

On the other hand, for the same reason:

$$\max_{\|\mathbf{x}\|_2=1} \mathbf{x}^H \mathbf{A} \mathbf{x} = \max_{\|\mathbf{x}\|_2=1} (\mathbf{U} \mathbf{x})^H \mathbf{D} (\mathbf{U} \mathbf{x}) = \max_{\|\mathbf{y}\|_2=1} \mathbf{y}^H \mathbf{D} \mathbf{y} = \max_{i=1,\dots,n} |d_i|.$$

Hence, both expressions in the statement of the lemma agree with the largest modulus of eigenvalues of  $\mathbf{A}$ .  $\square$

**Corollary 1.5.77. Euclidean matrix norm and eigenvalues**

For  $\mathbf{A} \in \mathbb{K}^{n,n}$  the Euclidean matrix norm  $\|\mathbf{A}\|_2$  is the square root of the largest (in modulus) eigenvalue of  $\mathbf{A}^H \mathbf{A}$ .

For a *normal* matrix  $\mathbf{A} \in \mathbb{K}^{n,n}$  (that is,  $\mathbf{A}$  satisfies  $\mathbf{A}^H \mathbf{A} = \mathbf{A} \mathbf{A}^H$ ) the Euclidean matrix norm agrees with the modulus of the largest eigenvalue.

**(1.5.78) (Numerical) algorithm**

When we talk about an “algorithm” we have in mind a *concrete code function* in MATLAB or C++; the only way to describe an algorithm is through a piece of code. We assume that this function defines another mapping  $\tilde{F} : X \rightarrow Y$  on the data space of the problem. Of course, we can only feed data to the MATLAB/C++-function, if they can be represented in the set  $\mathbb{M}$  of machine numbers. Hence, implicit in the definition of  $\tilde{F}$  is the assumption that input data are subject to *rounding* before passing them to the code function proper.

**(1.5.79) Stable algorithm**

- \* We study a problem ( $\rightarrow$  § 1.5.62)  $F : X \rightarrow Y$  on data space  $X$  into result space  $Y$ .

- \* We assume that both  $X$  and  $Y$  are equipped with norms  $\|\cdot\|_X$  and  $\|\cdot\|_Y$ , respectively ( $\rightarrow$  Def. 1.5.65).
- \* We consider a concrete algorithm  $\tilde{F} : X \rightarrow Y$  according to § 1.5.78.

We write  $w(\mathbf{x})$ ,  $\mathbf{x} \in X$ , for the **computational effort** ( $\rightarrow$  Def. 1.4.1) required by the algorithm for input  $\mathbf{x}$ .

### Definition 1.5.80. Stable algorithm

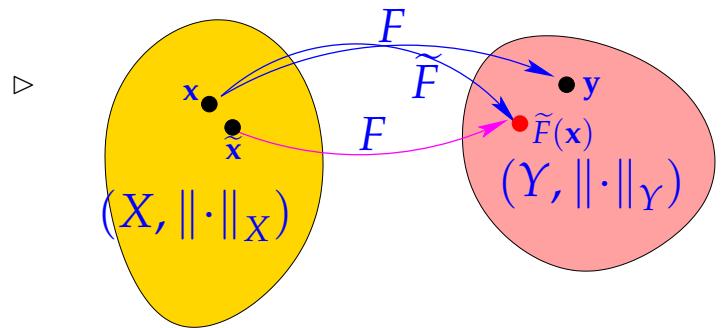
An algorithm  $\tilde{F}$  for solving a problem  $F : X \mapsto Y$  is **numerically stable**, if for all  $\mathbf{x} \in X$  its result  $\tilde{F}(\mathbf{x})$  (possibly affected by roundoff) is the exact result for “slightly perturbed” data:

$$\exists C \approx 1: \quad \forall \mathbf{x} \in X: \quad \exists \tilde{\mathbf{x}} \in X: \quad \|\mathbf{x} - \tilde{\mathbf{x}}\|_X \leq C w(\mathbf{x}) \text{eps} \|\mathbf{x}\|_X \quad \wedge \quad \tilde{F}(\mathbf{x}) = F(\tilde{\mathbf{x}}).$$

Here **EPS** should be read as machine precision according to the “Axiom” of roundoff analysis Ass. 1.5.31.

Illustration of Def. 1.5.80  
( $\mathbf{y} \doteq$  exact result for exact data  $\mathbf{x}$ )

Terminology:  
Def. 1.5.80 introduces stability in the sense of  
**backward error analysis**



Sloppily speaking, the impact of roundoff (\*) on a *stable algorithm* is of the same order of magnitude as the effect of the inevitable perturbations due to rounding the input data.

➤ For stable algorithms roundoff errors are “harmless”.

(\*) In some cases the definition of  $\tilde{F}$  will also involve some approximations as in Ex. 1.5.60. Then the above statement also includes approximation errors.

### Example 1.5.81 (Testing stability of matrix×vector multiplication)

Assume you are given a black box implementation of a function

$\mathbf{y} = \text{mvmult}(\mathbf{A}, \mathbf{x})$

that purports to provide a stable implementation of  $\mathbf{Ax}$  for  $\mathbf{A} \in \mathbb{K}^{m,n}$ ,  $\mathbf{x} \in \mathbb{K}^n$ , cf. Ex. 1.5.63. How can we verify this claim for particular data. Both,  $\mathbb{K}^{m,n}$  and  $\mathbb{K}^n$  are equipped with the Euclidean norm.

The task is, given  $\mathbf{y} \in \mathbb{K}^n$  as returned by the function, to find conditions on  $\mathbf{y}$  that ensure the existence of a  $\tilde{\mathbf{A}} \in \mathbb{K}^{m,n}$  such that

$$\tilde{\mathbf{A}}\mathbf{x} = \mathbf{y} \quad \text{and} \quad \left\| \tilde{\mathbf{A}} - \mathbf{A} \right\|_2 \leq C_{mn} \text{EPS} \|\mathbf{A}\|_2, \quad (1.5.82)$$

for a small constant  $\approx 1$ .

In fact we can choose (easy computation)

$$\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{z}\mathbf{x}^T, \quad \mathbf{z} := \frac{\mathbf{y} - \mathbf{Ax}}{\|\mathbf{x}\|_2^2} \in \mathbb{K}^m,$$

and we find

$$\|\tilde{\mathbf{A}} - \mathbf{A}\|_2 = \|\mathbf{z}\mathbf{x}^T\|_2 = \sup_{\mathbf{w} \in \mathbb{K}^n \setminus \{0\}} \frac{\mathbf{x} \cdot \mathbf{w} \|\mathbf{z}\|_2}{\|\mathbf{w}\|_2} \leq \|\mathbf{x}\|_2 \|\mathbf{z}\|_2 = \frac{\|\mathbf{y} - \mathbf{Ax}\|_2}{\|\mathbf{x}\|_2}.$$

Hence, in principle stability of an algorithm for computing  $\mathbf{Ax}$  is confirmed, if for every  $\mathbf{x} \in X$  the computed result  $\mathbf{y} = \mathbf{y}(\mathbf{x})$  satisfies

$$\|\mathbf{y} - \mathbf{Ax}\|_2 \leq C mn \text{EPS} \|\mathbf{x}\|_2 \|\mathbf{A}\|_2,$$

with a small constant  $C > 0$  independent of data and problem size.

### Remark 1.5.83 (Numerical stability and sensitive dependence on data)

A problem shows **sensitive dependence** on the data, if small perturbations of input data lead to large perturbations of the output. Such problems are also called **ill-conditioned**. For such problems stability of an algorithm is easily accomplished.

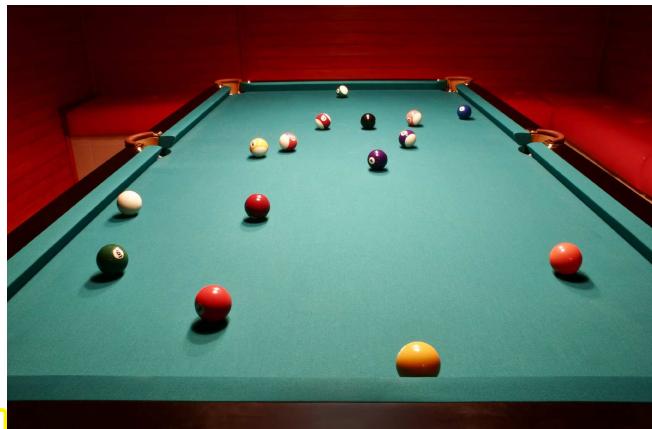


Fig. 20

Example: The problem is the prediction of the position of the billiard ball after ten bounces given the initial position, velocity, and spin.

It is well known, that tiny changes of the initial conditions can shift the final location of the ball to virtually any point on the table: the billiard problem is **chaotic**.

Hence, a stable algorithm for its solution may just output a **fixed** or **random** position without even using the initial conditions!

## 1.6 Direct methods for linear systems

### (1.6.1) The problem: solving a linear system

Given : **square** matrix  $\mathbf{A} \in \mathbb{K}^{n,n}$ , vector  $\mathbf{b} \in \mathbb{K}^n$ ,  $n \in \mathbb{N}$

Sought : solution vector  $\mathbf{x} \in \mathbb{K}^n$ :  $\boxed{\mathbf{Ax} = \mathbf{b}}$   $\leftarrow$  (square) **linear system of equations** (LSE)  
(Formal problem mapping  $(\mathbf{A}, \mathbf{b}) \mapsto \mathbf{A}^{-1}\mathbf{b}$ )

(Terminology:  $\mathbf{A} \hat{=} \text{system matrix}$ ,  $\mathbf{b} \hat{=} \text{right hand side vector}$  )

Linear systems with rectangular system matrices  $\mathbf{A} \in \mathbb{K}^{m,n}$ , called “overdetermined” for  $m > n$ , and “underdetermined” for  $m < n$  will be treated in Chapter 6.

### (1.6.2) LSE: key components of mathematical models in many fields

Linear systems of equations are ubiquitous in computational science: they are encountered

- with discrete linear models in network theory (see Ex. 1.6.3), control, statistics;
- in the case of *discretized* boundary value problems for ordinary and partial differential equations ( $\rightarrow$  course “Numerical methods for partial differential equations”, 4th semester);
- as a result of linearization (e.g. “Newton’s method”  $\rightarrow$  Sect. 2.4).

### Example 1.6.3 (Nodal analysis of (linear) electric circuit [63, Sect. 4.7.1])

Now we study a very important application of numerical simulation, where (large, sparse) linear systems of equations play a central role: **Numerical circuit analysis**. We begin with *linear circuits* in the *frequency domain*, which are directly modelled by complex linear systems of equations. Later we tackle circuits with non-linear elements, see Ex. 2.0.1, and, finally, will learn about numerical methods for computing the transient (time-dependent) behavior of circuits, see Ex. 11.1.13.

Modeling of simple linear circuits takes only elementary physical laws as covered in any introductory course of physics (or even in secondary school physics). There is no sophisticated physics or mathematics involved.

**Node** (ger.: Knoten)  $\hat{=}$  junction of wires

☞ number nodes  $1, \dots, n$

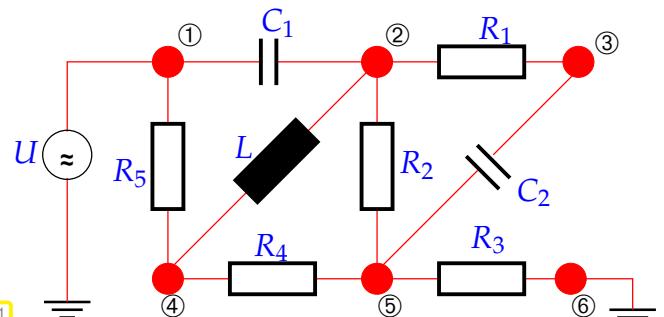
$I_{kj}$ : current from node  $k \rightarrow$  node  $j$ ,  $I_{kj} = -I_{jk}$

**Kirchhoff current law (KCL)** : sum of node currents =

0:

$$\forall k \in \{1, \dots, n\}: \sum_{j=1}^n I_{kj} = 0 .$$

(1.6.4)  
Fig. 21



Unknowns:

nodal potentials  $U_k$ ,  $k = 1, \dots, n$ .

(some may be known: grounded nodes: ⑥ in Fig. 21, voltage sources: ① in Fig. 21)

Constitutive relations for circuit elements: (in *frequency domain* with angular frequency  $\omega > 0$ ):

- Ohmic resistor:  $I = \frac{U}{R}$ ,  $[R] = 1 \text{VA}^{-1}$
- capacitor:  $I = i\omega C U$ , capacitance  $[C] = 1 \text{AsV}^{-1}$
- coil/inductor :  $I = \frac{U}{i\omega L}$ , inductance  $[L] = 1 \text{VsA}^{-1}$

$$\Rightarrow I_{kj} = \begin{cases} R^{-1}(U_k - U_j) , \\ i\omega C(U_k - U_j) , \\ -i\omega^{-1}L^{-1}(U_k - U_j) . \end{cases}$$

notation:  $\imath \hat{=} \text{imaginary unit } \imath := \sqrt{-1}$ ,  $\imath = \exp(\imath\pi/2)$

Here we face the special case of a **linear circuit**: all relationships between branch currents and voltages are of the form

$$I_{kj} = \alpha_{kj}(U_k - U_j) \quad \text{with} \quad \alpha_{kj} \in \mathbb{C}. \quad (1.6.5)$$

The concrete value of  $\alpha_{kj}$  is determined by the circuit element connecting node  $k$  and node  $j$ .

These constitutive relations are derived by assuming a harmonic time-dependence of all quantities, which is termed circuit analysis in the **frequency domain** (AC-mode).

$$\text{voltage: } u(t) = \operatorname{Re}\{U \exp(\imath\omega t)\}, \quad \text{current: } i(t) = \operatorname{Re}\{I \exp(\imath\omega t)\}. \quad (1.6.6)$$

Here  $U, I \in \mathbb{C}$  are called complex amplitudes. This implies for temporal derivatives (denoted by a dot):

$$\frac{du}{dt}(t) = \operatorname{Re}\{\imath\omega U \exp(\imath\omega t)\}, \quad \frac{di}{dt}(t) = \operatorname{Re}\{\imath\omega I \exp(\imath\omega t)\}. \quad (1.6.7)$$

For a capacitor the total charge is proportional to the applied voltage:

$$q(t) = Cu(t) \quad \frac{i(t)}{i(t)} = \dot{q}(t) \Rightarrow i(t) = C\dot{u}(t).$$

For a coil the voltage is proportional to the rate of change of current:  $u(t) = L\dot{i}(t)$ . Combined with (1.6.6) and (1.6.7) this leads to the above constitutive relations.

Constitutive relations + (1.6.4)  linear system of equations:

$$\begin{aligned} \textcircled{2}: \quad & \imath\omega C_1(U_2 - U_1) + R_1^{-1}(U_2 - U_3) - \imath\omega^{-1}L^{-1}(U_2 - U_4) + R_2^{-1}(U_2 - U_5) = 0, \\ \textcircled{3}: \quad & R_1^{-1}(U_3 - U_2) + \imath\omega C_2(U_3 - U_5) = 0, \\ \textcircled{4}: \quad & R_5^{-1}(U_4 - U_1) - \imath\omega^{-1}L^{-1}(U_4 - U_2) + R_4^{-1}(U_4 - U_5) = 0, \\ \textcircled{5}: \quad & R_2^{-1}(U_5 - U_2) + \imath\omega C_2(U_5 - U_3) + R_4^{-1}(U_5 - U_4) + R_3^{-1}(U_5 - U_6) = 0, \\ & U_1 = U, \quad U_6 = 0. \end{aligned}$$

No equations for nodes  $\textcircled{1}$  and  $\textcircled{6}$ , because these nodes are connected to the “outside world” so that the Kirchhoff current law (1.6.4) does not hold (from a local perspective). This is fitting, because the voltages in these nodes are known anyway.



$$\begin{pmatrix} \imath\omega C_1 + \frac{1}{R_1} - \frac{i}{\omega L} + \frac{1}{R_2} & -\frac{1}{R_1} & \frac{i}{\omega L} & -\frac{1}{R_2} & \\ -\frac{1}{R_1} & \frac{1}{R_1} + \imath\omega C_2 & 0 & -\imath\omega C_2 & \\ \frac{i}{\omega L} & 0 & \frac{1}{R_5} - \frac{i}{\omega L} + \frac{1}{R_4} & -\frac{1}{R_4} & \\ -\frac{1}{R_2} & -\imath\omega C_2 & -\frac{1}{R_4} & \frac{1}{R_2} + \imath\omega C_2 + \frac{1}{R_4} + R_3^{-1} & \end{pmatrix} \begin{pmatrix} U_2 \\ U_3 \\ U_4 \\ U_5 \end{pmatrix} = \begin{pmatrix} \imath\omega C_1 U \\ 0 \\ \frac{1}{R_5} U \\ 0 \end{pmatrix}$$

This is a linear system of equations with *complex* coefficients:  $A \in \mathbb{C}^{4,4}$ ,  $b \in \mathbb{C}^4$ . For the algorithms to be discussed below this does not matter, because they work alike for real and complex numbers.

## 1.6.1 Theory: Linear systems of equations

### 1.6.1.1 Existence and uniqueness of solutions

Known from linear algebra [59, Sect. 1.2], [34, Sect. 1.3]:

**Definition 1.6.8. Invertible matrix** → [59, Sect. 2.3]

$$\mathbf{A} \in \mathbb{K}^{n,n} \quad \begin{array}{l} \text{invertible /} \\ \text{regular} \end{array} \quad :\Leftrightarrow \quad \exists_1 \mathbf{B} \in \mathbb{K}^{n,n}: \quad \mathbf{AB} = \mathbf{BA} = \mathbf{I}.$$

$\mathbf{B}$  ≈ inverse of  $\mathbf{A}$ , (notation  $\mathbf{B} = \mathbf{A}^{-1}$ )

**Definition 1.6.9. Rank of a matrix** → [59, Sect. 2.4], [63, Sect. 1.5]

The rank of a matrix  $\mathbf{M} \in \mathbb{K}^{m,n}$ , denoted by  $\text{rank}(\mathbf{M})$ , is the maximal number of linearly independent rows/columns of  $\mathbf{M}$ .

**Theorem 1.6.10. Criteria for invertibility of matrix** → [59, Sect. 2.3 & Cor. 3.8]

A square matrix  $\mathbf{A} \in \mathbb{K}^{n,n}$  is invertible/regular if one of the following equivalent conditions is satisfied:

1.  $\exists \mathbf{B} \in \mathbb{K}^{n,n}: \quad \mathbf{BA} = \mathbf{AB} = \mathbf{I}$ ,
2.  $\mathbf{x} \mapsto \mathbf{Ax}$  defines an endomorphism of  $\mathbb{K}^n$ ,
3. the columns of  $\mathbf{A}$  are linearly independent (full column rank),
4. the rows of  $\mathbf{A}$  are linearly independent (full row rank),
5.  $\det \mathbf{A} \neq 0$  (non-vanishing determinant),
6.  $\text{rank}(\mathbf{A}) = n$  (full rank).

Formal way to denote solution of LSE:

$$\mathbf{A} \in \mathbb{K}^{n,n} \text{ regular} \quad \& \quad \mathbf{Ax} = \mathbf{b} \quad \Rightarrow \quad \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$


**Remark 1.6.11 (The inverse matrix and solution of a LSE)**

MATLAB: inverse of a matrix  $\mathbf{A}$  available through `inv(A)`

⚠ Always avoid computing the inverse of a matrix (which can almost always be avoided)!  
 In particular, never ever even contemplate using `x = inv(A) * b` to solve the linear system of equations  $\mathbf{Ax} = \mathbf{b}$ . The next sections present a sound way to do this.

### 1.6.1.2 Sensitivity of linear systems

The **Sensitivity** of a problem (for given data) gauges the impact of small perturbations of the data on the result.

Now we study the sensitivity of the problem of finding the solution of a linear system of equations  $\mathbf{Ax} = \mathbf{b}$ ,  $\mathbf{A} \in \mathbb{R}^{n,n}$  regular,  $\mathbf{b} \in \mathbb{R}^n$ , see § 1.6.1. We write  $\tilde{\mathbf{x}}$  for the solution of the perturbed linear system.

Question: To what extent do perturbations in the data  $\mathbf{A}, \mathbf{b}$  cause a

$$\text{(normwise) relative error: } \epsilon_r := \frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|}.$$

( $\|\cdot\|$   $\hat{=}$  suitable vector norm, e.g., maximum norm  $\|\cdot\|_\infty$ )

Perturbed linear system:

$$\mathbf{Ax} = \mathbf{b} \leftrightarrow (\mathbf{A} + \Delta\mathbf{A})\tilde{\mathbf{x}} = \mathbf{b} + \Delta\mathbf{b} \quad \blacktriangleright \quad (\mathbf{A} + \Delta\mathbf{A})(\tilde{\mathbf{x}} - \mathbf{x}) = \Delta\mathbf{b} - \Delta\mathbf{Ax}. \quad (1.6.12)$$



### Theorem 1.6.13. Conditioning of LSEs $\rightarrow [63, \text{Thm. 3.1}]$

If  $\mathbf{A}$  regular,  $\|\Delta\mathbf{A}\| < \|\mathbf{A}^{-1}\|^{-1}$  and (1.6.12), then

$$\frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \frac{\|\mathbf{A}^{-1}\| \|\mathbf{A}\|}{1 - \|\mathbf{A}^{-1}\| \|\mathbf{A}\| \|\Delta\mathbf{A}\| / \|\mathbf{A}\|} \left( \frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} + \frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|} \right).$$

↑ relative error      ↑ relative perturbations

The proof is based on the following fundamental result:

### Lemma 1.6.14 (Perturbation lemma). $\rightarrow [63, \text{Thm. 1.5}]$

$$\mathbf{B} \in \mathbb{R}^{n,n}, \|\mathbf{B}\| < 1 \Rightarrow \mathbf{I} + \mathbf{B} \text{ regular} \wedge \|(\mathbf{I} + \mathbf{B})^{-1}\| \leq \frac{1}{1 - \|\mathbf{B}\|}.$$

*Proof.*  $\triangle$ -inequality  $\Rightarrow \|(\mathbf{I} + \mathbf{B})\mathbf{x}\| \geq (1 - \|\mathbf{B}\|)\|\mathbf{x}\|, \forall \mathbf{x} \in \mathbb{R}^n \Rightarrow \mathbf{I} + \mathbf{B} \text{ regular.}$

$$\blacktriangleright \|(\mathbf{I} + \mathbf{B})^{-1}\| = \sup_{\mathbf{x} \in \mathbb{R}^n \setminus \{0\}} \frac{\|(\mathbf{I} + \mathbf{B})^{-1}\mathbf{x}\|}{\|\mathbf{x}\|} = \sup_{\mathbf{y} \in \mathbb{R}^n \setminus \{0\}} \frac{\|\mathbf{y}\|}{\|(\mathbf{I} + \mathbf{B})\mathbf{y}\|} \leq \frac{1}{1 - \|\mathbf{B}\|}$$

*Proof* (of Thm. 1.6.13) Lemma 1.6.14  $\Rightarrow \|(\mathbf{A} + \Delta\mathbf{A})^{-1}\| \leq \frac{\|\mathbf{A}^{-1}\|}{1 - \|\mathbf{A}^{-1}\|\|\Delta\mathbf{A}\|} \quad \& \quad (1.6.12)$

$$\Rightarrow \|\Delta\mathbf{x}\| \leq \frac{\|\mathbf{A}^{-1}\|}{1 - \|\mathbf{A}^{-1}\|\|\Delta\mathbf{A}\|} (\|\Delta\mathbf{b}\| + \|\Delta\mathbf{Ax}\|) \leq \frac{\|\mathbf{A}^{-1}\| \|\mathbf{A}\|}{1 - \|\mathbf{A}^{-1}\| \|\Delta\mathbf{A}\|} \left( \frac{\|\Delta\mathbf{b}\|}{\|\mathbf{A}\| \|\mathbf{x}\|} + \frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|} \right) \|\mathbf{x}\|.$$

### Definition 1.6.15. Condition (number) of a matrix

Condition (number) of a matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$ :

$$\text{cond}(\mathbf{A}) := \|\mathbf{A}^{-1}\| \|\mathbf{A}\|$$

Note:

$\text{cond}(\mathbf{A})$  depends on  $\|\cdot\|$ !

Rewriting estimate of Thm. 1.6.13 with  $\Delta\mathbf{b} = 0$ ,

$$\epsilon_r := \frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \frac{\text{cond}(\mathbf{A}) \delta_A}{1 - \text{cond}(\mathbf{A}) \delta_A}, \quad \delta_A := \frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|}. \quad (1.6.16)$$

- (1.6.16)  $\Rightarrow$
- \* If  $\text{cond}(\mathbf{A}) \gg 1$ , small perturbations in  $\mathbf{A}$  can lead to large relative errors in the solution of the LSE.
  - \* If  $\text{cond}(\mathbf{A}) \gg 1$ , a stable algorithm ( $\rightarrow$  Def. 1.5.80) can produce solutions with large relative error !

Recall Thm. 1.6.13: for regular  $\mathbf{A} \in \mathbb{K}^{n,n}$ , small  $\Delta\mathbf{A}$ , generic vector/matrix norm  $\|\cdot\|$

$$\begin{aligned} \mathbf{Ax} = \mathbf{b} \\ (\mathbf{A} + \Delta\mathbf{A})\tilde{\mathbf{x}} = \mathbf{b} + \Delta\mathbf{b} \end{aligned} \Rightarrow \frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \frac{\text{cond}(\mathbf{A})}{1 - \text{cond}(\mathbf{A})\|\Delta\mathbf{A}\|/\|\mathbf{A}\|} \left( \frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} + \frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|} \right). \quad (1.6.17)$$

- $\blacktriangleright \text{cond}(\mathbf{A}) \gg 1 \Rightarrow$  small relative changes of data  $\mathbf{A}, \mathbf{b}$  may effect huge relative changes in solution.
- $\blacktriangleright \text{cond}(\mathbf{A})$  indicates sensitivity of “LSE problem”  $(\mathbf{A}, \mathbf{b}) \mapsto \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$   
(as “amplification factor” of relative perturbations in the data  $\mathbf{A}, \mathbf{b}$ ).

Terminology:

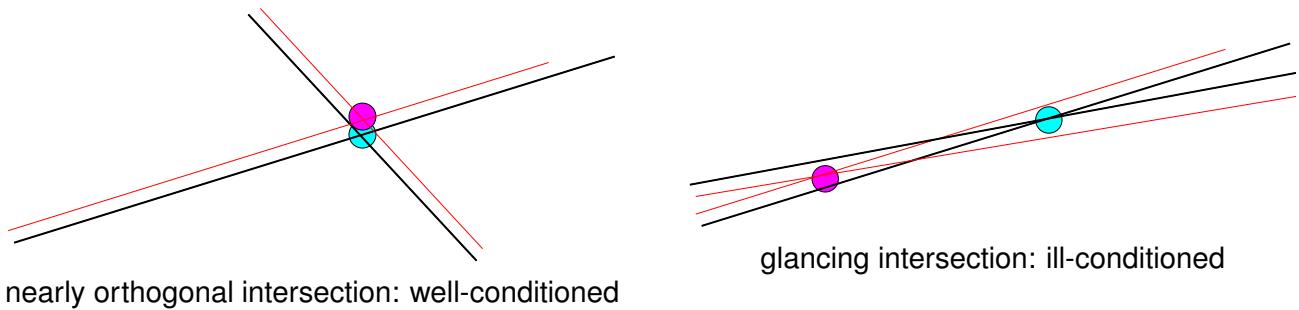
- Small changes of data  $\Rightarrow$  small perturbations of result : well-conditioned problem  
Small changes of data  $\Rightarrow$  large perturbations of result : ill-conditioned problem

Note: sensitivity gauge depends on the chosen norm !

### Example 1.6.18 (Intersection of lines in 2D)

Solving a  $2 \times 2$  linear system of equations amounts to finding the intersection of two lines in the coordinate plane. This relationship allows a geometric view of “sensitivity of a linear system”:

In distance metric (Euclidean vector norm):



Hessian normal form of line # $i$ ,  $i = 1, 2$ :

$$L_i = \{\mathbf{x} \in \mathbb{R}^2 : \mathbf{x}^T \mathbf{n}_i = d_i\}, \quad \mathbf{n}_i \in \mathbb{R}^2, d_i \in \mathbb{R}.$$

$\blacktriangleright$  LSE for finding intersection:

$$\underbrace{\begin{bmatrix} \mathbf{n}_1^T \\ \mathbf{n}_2^T \end{bmatrix}}_{=: \mathbf{A}} \mathbf{x} = \underbrace{\begin{bmatrix} d_1 \\ d_2 \end{bmatrix}}_{=: \mathbf{b}},$$

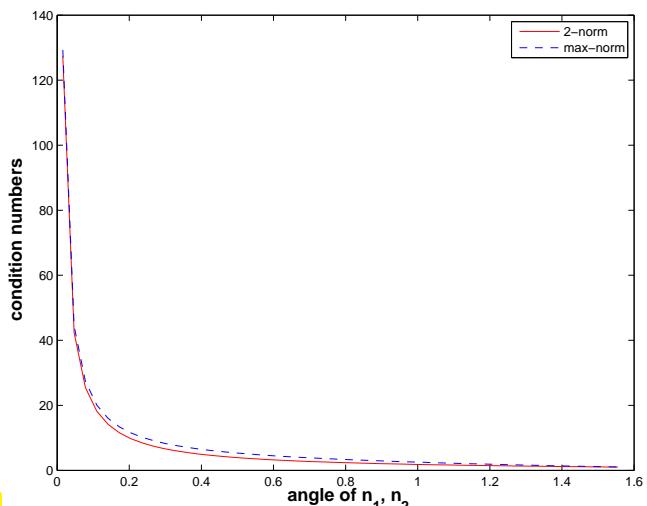
$\mathbf{n}_i \hat{=} \text{(unit) direction vectors}$ ,  $d_i \hat{=} \text{distance to origin}$ .

**MATLAB-code 1.6.19: condition numbers of  
 $2 \times 2$  matrices**

```

1 r = [];
2 for phi=pi/200:pi/100:pi/2
3     A = [1,cos(phi);
4           0,sin(phi)];
5     r = [r; phi,
6           cond(A),cond(A,'inf')];
7 end
8 plot(r(:,1),r(:,2),'r-',
9       r(:,1),r(:,3),'b--');
10 xlabel('{\bf angle of n_1, n_2}', 'fontsize', 14);
11 ylabel('{\bf condition numbers}', 'fontsize', 14);
12 legend('2-norm', 'max-norm');
13 print -depsc2
14      '../PICTURES/linesec.eps';

```



We clearly observe a blow-up of  $\text{cond}(\mathbf{A})$  (with respect to the Euclidean vector norms) as the angle enclosed by the two lines shrinks.

This corresponds to a large sensitivity of the location of the intersection point in the case of glancing incidence.

**Heuristics for predicting large  $\text{cond}(\mathbf{A})$** 

$\text{cond}(\mathbf{A}) \gg 1 \leftrightarrow$  columns/rows of  $\mathbf{A}$  “almost linearly dependent”

## 1.6.2 Gaussian Elimination

### 1.6.2.1 Basic algorithm

! Exceptional feature of linear systems of equations (LSE):  
☞ “exact” solution computable with finitely many elementary operations

Algorithm: **Gaussian elimination** ( $\rightarrow$  secondary school, linear algebra,)

Familiarity with the algorithm of Gaussian elimination for a square linear system of equations will be taken for granted.

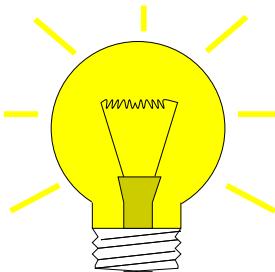


**Supplementary reading.** In case you cannot remember the main facts about Gaussian elimination, very detailed accounts and examples can be found in

- M. Gutknecht's lecture notes [34, Ch. 1],
- the textbook by Nipp & Stoffer [59, Ch. 1],
- the numerical analysis text by Quarteroni et al. [63, Sects. 3.2 & 3.3],
- the textbook by Ascher & Greif [6, Sect. 5.1],

and, to some extend, below, see Ex. 1.6.20.

**Wikipedia:** Although the method is named after mathematician **Carl Friedrich Gauss**, the earliest presentation of it can be found in the important Chinese mathematical text *Jiuzhang suanshu* or The Nine Chapters on the Mathematical Art, dated approximately 150 B.C., and commented on by **Liu Hui** in the 3rd century.



Idea: transformation to “simpler”, but equivalent LSE by means of successive (invertible) *row transformations*

Ex. 1.3.13: row transformations  $\leftrightarrow$  left-multiplication with transformation matrix

Obviously, left multiplication with a regular matrix does not affect the solution of an LSE: for any *regular*  $T \in \mathbb{K}^{n,n}$

$$Ax = b \Rightarrow A'x = b' , \text{ if } A' = TA, b' = Tb .$$

So we may try to convert the linear system of equations to a form that can be solved more easily by multiplying with regular matrices from left, which boils down to applying row transformations. A suitable target format is a diagonal linear system of equations, for which all equations are completely decoupled. This is the gist of Gaussian elimination.

### Example 1.6.20 (Gaussian elimination)

① (Forward) elimination:

$$\begin{pmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ -3 \end{pmatrix} \longleftrightarrow \begin{array}{rcl} x_1 + x_2 & = & 4 \\ 2x_1 + x_2 - x_3 & = & 1 \\ 3x_1 - x_2 - x_3 & = & -3 \end{array} .$$

$$\begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 1 \\ -3 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 3 & -1 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ -7 \\ -3 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & -4 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ -7 \\ -15 \end{bmatrix}$$

$$\rightarrow \underbrace{\begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 3 \end{bmatrix}}_{=U} \begin{bmatrix} 4 \\ -7 \\ 13 \end{bmatrix}$$

= pivot row, pivot element **bold**.

➤ transformation of LSE to **upper triangular form**

② Solve by **back substitution**: back substitution = Rücksubstitution

$$\begin{array}{rcl} x_1 + x_2 & = & 4 \\ -x_2 - x_3 & = & -7 \\ 3x_3 & = & 13 \end{array} \Rightarrow \begin{array}{rcl} x_3 & = & \frac{13}{3} \\ x_2 & = & 7 - \frac{13}{3} = \frac{8}{3} \\ x_1 & = & 4 - \frac{8}{3} = \frac{4}{3} \end{array} .$$

More detailed examples: [34, Sect. 1.1], [59, Sect. 1.1].

More general:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots &\quad \vdots \quad \vdots \\ \vdots &\quad \vdots \quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

- $i$ -th row -  $l_{i1} \cdot$  1st row (**pivot row**),  $l_{i1} := a_{i1}/a_{11}$ ,  $i = 2, \dots, n$

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{22}^{(1)}x_2 + \cdots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\ \vdots &\quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\ \vdots &\quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\ a_{n2}^{(1)}x_2 + \cdots + a_{nn}^{(1)}x_n &= b_n^{(1)} \end{aligned} \quad \text{with} \quad \begin{aligned} a_{ij}^{(1)} &= a_{ij} - a_{1j}l_{i1}, \quad i, j = 2, \dots, n, \\ b_i^{(1)} &= b_i - b_1l_{i1}, \quad i = 2, \dots, n. \end{aligned}$$

- $i$ -th row -  $l_{i1} \cdot$  2nd row (**pivot row**),  $l_{i2} := a_{i2}^{(1)}/a_{22}^{(1)}$ ,  $i = 3, \dots, n$ .

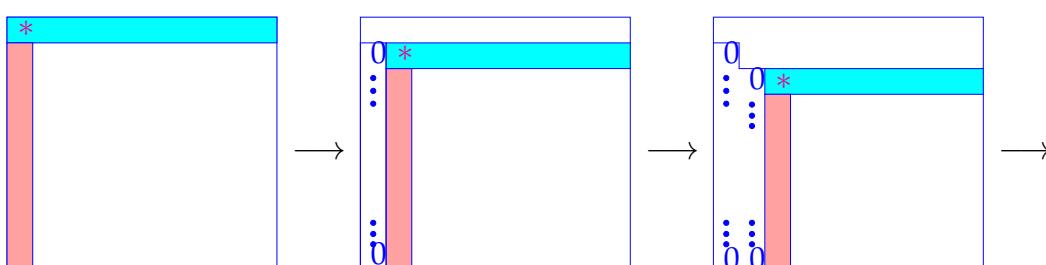
$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\ a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \cdots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\ a_{33}^{(2)}x_3 + \cdots + a_{3n}^{(2)}x_n &= b_3^{(2)} \\ \vdots &\quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\ a_{n3}^{(2)}x_3 + \cdots + a_{nn}^{(2)}x_n &= b_n^{(2)} \end{aligned}$$

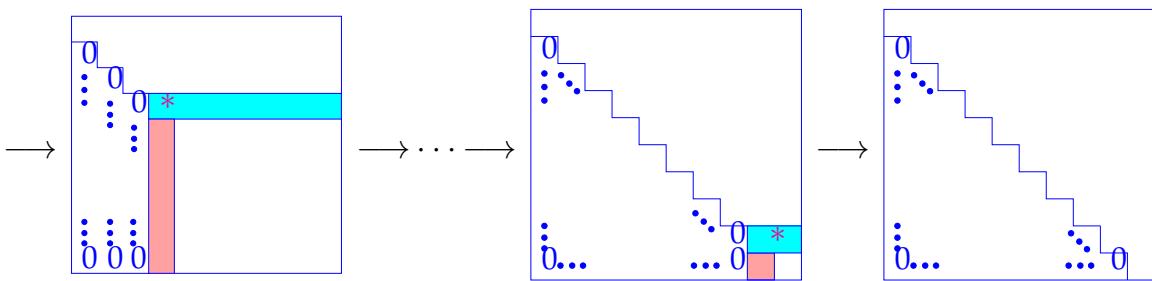
► After  $n - 1$  steps: linear systems of equations in **upper triangular form**

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\ a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \cdots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\ a_{33}^{(2)}x_3 + \cdots + a_{3n}^{(2)}x_n &= b_3^{(2)} \\ \vdots &\quad \vdots \quad \vdots \quad \vdots \quad \vdots \\ \vdots &\quad \vdots \quad \vdots \quad \vdots \quad \vdots \\ a_{nn}^{(n-1)}x_n &= b_n^{(n-1)} \end{aligned}$$

Terminology:  $a_{11}, a_{22}^{(1)}, a_{33}^{(2)}, \dots, a_{n-1,n-1}^{(n-2)}$  = **pivots/pivot elements**

Graphical depiction:





$* \triangleq \text{pivot (necessarily } \neq 0 \text{)} \rightarrow \text{here: assumption), }$  = pivot row

In  $k$ -th step (starting from  $\mathbf{A} \in \mathbb{K}^{n,n}$ ,  $1 \leq k < n$ , pivot row  $\mathbf{a}_k^\top$ ):

transformation:  $\mathbf{Ax} = \mathbf{b} \rightarrow \mathbf{A}'\mathbf{x} = \mathbf{b}'$ .

with

$$a'_{ij} := \begin{cases} a_{ij} - \frac{a_{ik}}{a_{kk}} a_{kj} & \text{for } k < i, j \leq n, \\ 0 & \text{for } k < i \leq n, j = k, \\ a_{ij} & \text{else,} \end{cases} \quad b'_i := \begin{cases} b_i - \frac{a_{ik}}{a_{kk}} b_k & \text{for } k < i \leq n, \\ b_i & \text{else.} \end{cases} \quad (1.6.21)$$

multipliers  $l_{ik}$

### (1.6.22) Gaussian elimination: algorithm

Here we give a direct MATLAB implementation of Gaussian elimination for LSE  $\mathbf{Ax} = \mathbf{b}$  (grossly inefficient!).

#### MATLAB-code 1.6.23: Solving LSE $\mathbf{Ax} = \mathbf{b}$ with Gaussian elimination

```

1 function x = gausselimsolve(A,b)
2 % Gauss elimination without pivoting, x = A\b
3 % A must be an n×n-matrix, b an n-vector
4 n = size(A,1); A = [A,b]; %
5 % Forward elimination (cf. step ① in Ex. 1.6.20)
6 for i=1:n-1, pivot = A(i,i);
7   for k=i+1:n, fac = A(k,i)/pivot;
8     A(k,i+1:n+1) = A(k,i+1:n+1) - fac*A(i,i+1:n+1); %
9   end
10 end
11 % Back substitution (cf. step ② in Ex. 1.6.20)
12 A(n,n+1) = A(n,n+1)/A(n,n);
13 for i=n-1:-1:1
14   for l=i+1:n
15     A(i,n+1) = A(i,n+1) - A(l,n+1)*A(i,l);
16   end
17   A(i,n+1) = A(i,n+1)/A(i,i);
18 end
19 x = A(:,n+1); %

```

Line 4: right hand side vector set as last column of matrix, facilitates simultaneous row transformations of matrix and r.h.s.

Variable `fac`  $\hat{=}$  multiplier

Line 19: extract solution from last column of transformed matrix.

### (1.6.24) Computational effort of Gaussian elimination

Forward elimination: three nested loops (note: compact vector operation in line 8 involves another loop from  $i+1$  to  $m$ )

Back substitution: two nested loops

computational cost ( $\leftrightarrow$  number of elementary operations) of Gaussian elimination [59, Sect. 1.3]:

$$\begin{aligned} \text{elimination : } & \sum_{i=1}^{n-1} (n-i)(2(n-i)+3) = n(n-1)(\frac{2}{3}n + \frac{7}{6}) \text{ Ops. ,} \\ \text{back substitution : } & \sum_{i=1}^n 2(n-i) + 1 = n^2 \text{ Ops. .} \end{aligned} \quad (1.6.25)$$

$$\text{asymptotic complexity } (\rightarrow \text{Sect. 1.4}) \text{ of Gaussian elimination} \quad = \quad \frac{2}{3}n^3 + O(n^2) = O(n^3)$$

(without pivoting) for generic LSE  $\mathbf{Ax} = \mathbf{b}$ ,  $\mathbf{A} \in \mathbb{R}^{n,n}$

### Experiment 1.6.26 (Runtime of Gaussian elimination)

#### MATLAB-code 1.6.27: Measuring runtimes of Code 1.6.23 vs. MATLAB \ -operator

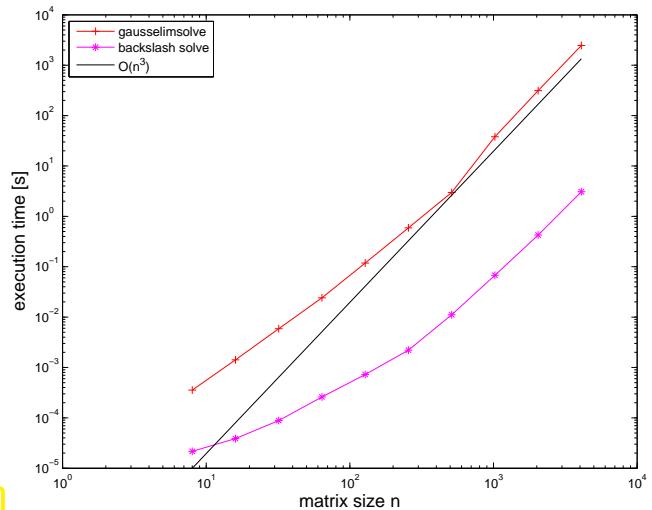
```

1 % MATLAB script for timing numerical solution of linear systems
2 nruns = 3; times = [];
3 for n = 2.^3:12)
4   fprintf('Matrix size n = %d\n', n);
5   % Initialized random matrix and right hand side
6   A = rand(n,n) + n*eye(n); b = rand(n,1);
7   t1 = realmax; t2 = realmax;
8   for j=1:nruns
9     tic; x1 = gausselimsolve(A,b); t1 = min(t1, toc);
10    tic; x2 = A\b; t2 = min(t2, toc);
11    norm(x1-x2),
12  end
13  times = [times; n t1 t2];
14 end
15
16 figure('name','gausstimer');
17 loglog(times(:,1),times(:,2),'r+',times(:,1),times(:,3),'m-*',...
18         times(:,1),times(:,1).^(3*(1E-5/(times(1,1)^3))), 'k-');
19 xlabel('matrix size n','fontsize',14);
20 ylabel('execution time [s]','fontsize',14);
21 legend('gausselimsolve','backslash
22           solve','O(n^3)', 'location','northwest');
```

23 `print -depsc2 '../PICTURES/gausstimering.eps';`

MATLAB

- ▷ based on LAPACK/MKL
- ▷ based on BLAS ( $\rightarrow$  Sect. 1.3.2)
- ▶ \ about two orders of magnitude faster than a direct implementation



Never implement Gaussian elimination yourself !

use numerical libraries (LAPACK/MKL) or MATLAB ! (MATLAB operator: \ )

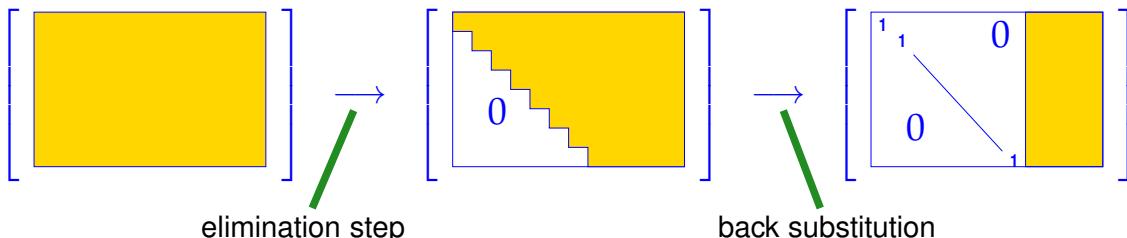
A concise list of libraries for numerical linear algebra and related problems can be found [here](#).

#### Remark 1.6.28 (Gaussian elimination for non-square matrices)

In Code 1.6.23: the right hand side vector  $\mathbf{b}$  was first appended to matrix  $\mathbf{A}$  as rightmost column, and then forward elimination and back substitution were carried out on the resulting matrix.

➢ Gaussian elimination for  $\mathbf{A} \in \mathbb{K}^{n,n+1}$ !

“fat matrix”:  $\mathbf{A} \in \mathbb{K}^{n,m}$ ,  $m > n$ :



Recall Code 1.6.23 ( $m = n + 1$ ): the solution vector  $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$  was recovered as the rightmost column of the augmented matrix  $(\mathbf{A}, \mathbf{b})$  after forward elimination and back substitution. In the above cartoon it would be contained in the yellow part of the matrix on the right.

**MATLAB-code 1.6.29: Gaussian elimination with multiple r.h.s.**

```

1 function X = gausselimsolvemult(A,B)
2 % Gauss elimination without pivoting,
3 % X = A\B
4 n = size(A,1); m = n + size(B,2); A
5 = [A,B];
6 for i=1:n-1, pivot = A(i,i);
7   for k=i+1:n, fac = A(k,i)/pivot;
8     A(k,i+1:m) = A(k,i+1:m) -
9       fac*A(i,i+1:m);
10  end
11 end
12 A(n,n+1:m) = A(n,n+1:m) /A(n,n);
13 for i=n-1:-1:1
14   for l=i+1:n
15     A(i,n+1:m) = A(i,n+1:m) -
16       A(l,n+1:m)*A(i,l);
17   end
18 A(i,n+1:m) = A(i,n+1:m)/A(i,i);
19 end
20 X = A(:,n+1:m);

```

Simultaneous solving of LSE with multiple right hand sides

Given regular  $A \in \mathbb{K}^{n,n}$ ,  $B \in \mathbb{K}^{n,k}$ , seek  $X \in \mathbb{K}^{n,k}$

$$AX = B \Leftrightarrow X = A^{-1}B$$

MATLAB:

$$X = A \setminus B; \text{ asymptotic complexity: } O(n^2(n+k))$$

Next two remarks: For understanding or analyzing special variants of Gaussian elimination, it is useful to be aware of

- the effects of elimination steps on the level of *matrix blocks*, cf. Rem. 1.3.15,
- and of the *recursive nature* of Gaussian elimination.

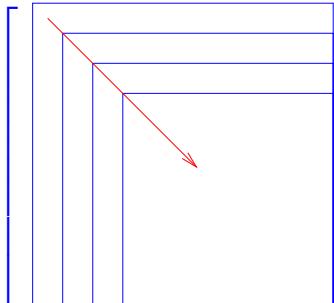
**Remark 1.6.30 (Gaussian elimination via rank-1 modifications)**

Block perspective (first step of Gaussian elimination with pivot  $\alpha \neq 0$ ), cf. (1.6.21):

$$\begin{array}{c}
 \left[ \begin{array}{cc} \alpha & c^\top \\ d & C \end{array} \right] \xrightarrow{\quad} \left[ \begin{array}{cc} \alpha & c^\top \\ 0 & C' := C - \frac{dc^\top}{\alpha} \end{array} \right] \\
 \text{rank-1 modification of } C
 \end{array} \quad (1.6.31)$$

Adding a tensor product of two vectors to a matrix is called a **rank-1 modification** of that matrix.

(1.6.31) suggests a **recursive** variant of Gaussian elimination:



```

1 function A = blockgs(A)
2 %in-situ recursive Gaussian elimination, no pivoting
3 %right hand side in rightmost column of A: A(:,end)
4 n=size(A,1);
5 if (n>1)
6 C=blockgs(A(2:end,2:end)-A(2:end,1)...
7           *A(1,2:end)/A(1,1));
8 A=[A(1,:); zeros(n-1,1),C];
9 end
```

r.h.s.  $\mathbf{b} \sim A(:, \text{end})$  ➤

In this code the Gaussian elimination is carried out **in situ**: the matrix  $A$  is replaced with the transformed matrices during elimination. If the matrix is not needed later this offers maximum efficiency.

### Remark 1.6.32 (Block Gaussian elimination)

Recall “principle” from Ex. 1.3.15: deal with block matrices (“matrices of matrices”) like regular matrices (except for commutativity of multiplication!).

Given: regular matrix  $\mathbf{A} \in \mathbb{K}^{n,n}$  with sub-matrices  $\mathbf{A}_{11} := (\mathbf{A})_{1:k,1:k}$ ,  $\mathbf{A}_{22} = (\mathbf{A})_{k+1:n,k+1:n}$ ,  $\mathbf{A}_{12} = (\mathbf{A})_{1:k,k+1:n}$ ,  $\mathbf{A}_{21} := (\mathbf{A})_{k+1:n,1:k}$ ,  $k < n$ , right hand side vector  $\mathbf{b} \in \mathbb{K}^n$ ,  $\mathbf{b}_1 = (\mathbf{b})_{1:k}$ ,  $\mathbf{b}_2 = (\mathbf{b})_{k+1:n}$

$$\left( \begin{array}{cc|c} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{b}_1 \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{b}_2 \end{array} \right) \xrightarrow{\textcircled{1}} \left( \begin{array}{cc|c} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{b}_1 \\ 0 & \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12} & \mathbf{b}_2 - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{b}_1 \end{array} \right) \xrightarrow{\textcircled{2}} \left( \begin{array}{cc|c} \mathbf{I} & 0 & \mathbf{A}_{11}^{-1}(\mathbf{b}_1 - \mathbf{A}_{12}\mathbf{S}^{-1}\mathbf{b}_2) \\ 0 & \mathbf{I} & \mathbf{S}^{-1}\mathbf{b}_2 \end{array} \right),$$

where  $\mathbf{S} := \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$  (Schur complement, see Rem. 1.6.46),  $\mathbf{b}_S := \mathbf{b}_2 - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{b}_1$

**①:** elimination step, **②:** backsubstitution step

*Assumption:* (sub-)matrices regular, if required.

### 1.6.2.2 LU-decomposition

A **matrix factorization** (ger. Matrixzerlegung) expresses a general matrix  $\mathbf{A}$  as product of two **special** (factor) matrices. Requirements for these special matrices define the matrix factorization.

Mathematical issue: existence & uniqueness

Numerical issue: algorithm for computing factor matrices

Matrix factorizations

- ☞ often capture the essence of algorithms in compact form (here: Gaussian elimination),
- ☞ are important building blocks for complex algorithms,
- ☞ are key theoretical tools for algorithm analysis.

In this section: forward elimination step of Gaussian elimination will be related to a special matrix factorization, the so-called LU-decomposition or LU-factorization.

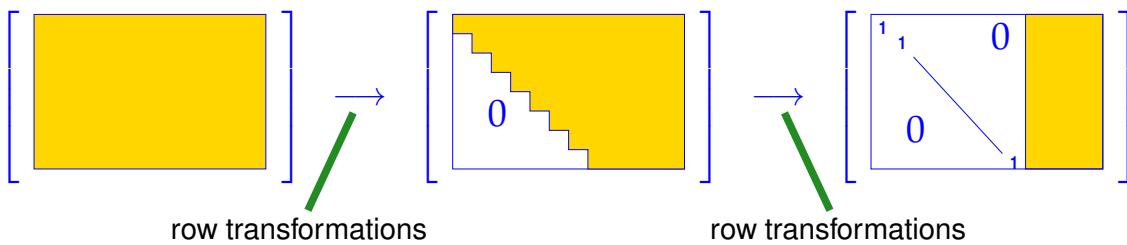


*Supplementary reading.* The LU-factorization should be well known from the introductory linear algebra course. In case you need to refresh your knowledge, please consult one of the following:

- textbook by Nipp & Stoffer [59, Sect. 2.4],
- book by M. Hanke-Bourgeois [42, II.4],
- linear algebra lecture notes by M. Gutknecht [34, Sect. 3.1],
- textbook by Quarteroni et al. [63, Sect. 3.3.1],
- Sect. 3.5 of the book by Dahmen & Reusken,
- Sect. 5.1 of the textbook by Ascher & Greif [6].

See also (1.6.33) below.

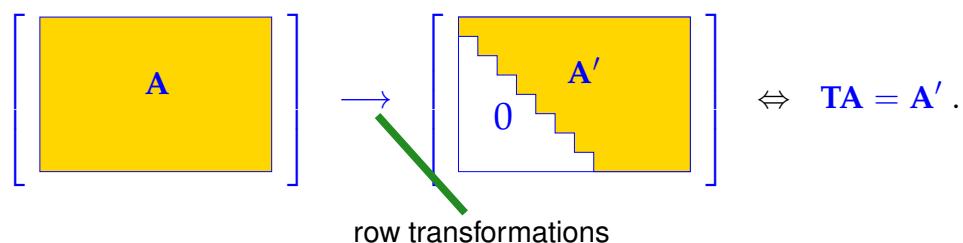
The gist of Gaussian elimination:



Here: **row transformation** = adding a multiple of a matrix row to another row, or  
multiplying a row with a non-zero scalar (number)  
(more special than row transformations discussed in Ex. 1.3.13)

Note: these row transformations preserve regularity of a matrix (why ?)  
▷ suitable for transforming linear systems of equations  
(solution will not be affected)

Ex. 1.3.13: row transformations can be realized by multiplication *from left* with suitable transformation matrices. When multiplying these transformation matrices we can emulate the effect to successive row transformations through left multiplication with a matrix  $T$ :



Now we want to determine the  $\mathbf{T}$  for the forward elimination step of Gaussian elimination.

**Example 1.6.33 (Gaussian elimination and LU-factorization → [59, Sect. 2.4], [42, II.4], [34, Sect. 3.1])**

LSE from Ex. 1.6.20: consider (forward) Gaussian elimination:

$$\begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ -3 \end{bmatrix} \longleftrightarrow \begin{array}{l} x_1 + x_2 = 4 \\ 2x_1 + x_2 - x_3 = 1 \\ 3x_1 - x_2 - x_3 = -3 \end{array}$$

$$\begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 1 \\ -3 \end{bmatrix} \xrightarrow{\quad} \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ -7 \\ -3 \end{bmatrix} \xrightarrow{\quad}$$

$$\begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ -7 \\ -15 \end{bmatrix} \xrightarrow{\quad} \underbrace{\begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & 4 & 1 \end{bmatrix}}_{=L} \underbrace{\begin{bmatrix} 4 \\ -7 \\ 13 \end{bmatrix}}_{=U}$$

= pivot row, pivot element **bold**, negative multipliers **red**

Details: link between Gaussian elimination and **matrix factorization** → Ex. 1.6.33  
(row transformation = multiplication with elimination matrix)

$$a_1 \neq 0 \xrightarrow{\quad} \begin{bmatrix} 1 & 0 & \cdots & 0 \\ -\frac{a_2}{a_1} & 1 & & 0 \\ -\frac{a_3}{a_1} & & \ddots & \\ \vdots & & & \\ -\frac{a_n}{a_1} & 0 & & 1 \end{bmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} a_1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}. \quad (1.6.34)$$

►  $n-1$  steps of Gaussian elimination: ► matrix factorization (→ Ex. 1.6.20)  
(non-zero pivot elements assumed)

$$\mathbf{A} = \mathbf{L}_1 \cdots \mathbf{L}_{n-1} \mathbf{U} \quad \text{with} \quad \begin{array}{l} \text{elimination matrices } \mathbf{L}_i, i = 1, \dots, n-1, \\ \text{upper triangular matrix } \mathbf{U} \in \mathbb{R}^{n,n}. \end{array}$$

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_2 & 1 & & 0 \\ l_3 & & \ddots & \\ \vdots & & & \\ l_n & 0 & & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ 0 & h_3 & 1 & \\ \vdots & \vdots & & \\ 0 & h_n & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_2 & 1 & & 0 \\ l_3 & h_3 & 1 & \\ \vdots & \vdots & & \\ l_n & h_n & 0 & 1 \end{bmatrix}$$

►  $\mathbf{L}_1 \cdots \mathbf{L}_{n-1}$  are **normalized lower triangular matrices**  
(entries = multipliers  $-\frac{a_{ik}}{a_{kk}}$  from (1.6.21) → Ex. 1.6.20)

The (forward) Gaussian elimination (without pivoting), for  $\mathbf{Ax} = \mathbf{b}$ ,  $\mathbf{A} \in \mathbb{R}^{n,n}$ , if possible, is *algebraically equivalent* to an LU-factorization/LU-decomposition  $\mathbf{A} = \mathbf{LU}$  of  $\mathbf{A}$  into a normalized lower triangular matrix  $\mathbf{L}$  and an upper triangular matrix  $\mathbf{U}$ , [15, Thm. 3.2.1], [59, Thm. 2.10], [34, Sect. 3.1].

Algebraically equivalent  $\hat{=}$  when carrying out the forward elimination in situ as in Code 1.6.23 and storing the multipliers in a lower triangular matrix as in Ex. 1.6.33, then the latter will contain the  $\mathbf{L}$ -factor and the original matrix will be replaced with the  $\mathbf{U}$ -factor.

### Lemma 1.6.35. Existence of LU-decomposition

The LU-decomposition of  $\mathbf{A} \in \mathbb{K}^{n,n}$  exists, if all submatrices  $(\mathbf{A})_{1:k, 1:k}$ ,  $1 \leq k \leq n$ , are regular.

*Proof.* by block matrix perspective ( $\rightarrow$  Rem. 1.3.15) and induction w.r.t.  $n$ :

$n = 1$ : assertion trivial

$n - 1 \rightarrow n$ : Induction hypothesis ensures existence of normalized lower triangular matrix  $\tilde{\mathbf{L}}$  and regular upper triangular matrix  $\tilde{\mathbf{U}}$  such that  $\tilde{\mathbf{A}} = \tilde{\mathbf{L}}\tilde{\mathbf{U}}$ , where  $\tilde{\mathbf{A}}$  is the upper left  $(n - 1) \times (n - 1)$  block of  $\mathbf{A}$ :

$$\left[ \begin{array}{c|c} \tilde{\mathbf{A}} & \mathbf{b} \\ \hline \mathbf{a}^\top & \alpha \end{array} \right] = \left[ \begin{array}{c|c} \tilde{\mathbf{L}} & 0 \\ \hline \mathbf{x}^\top & 1 \end{array} \right] \left[ \begin{array}{c|c} \tilde{\mathbf{U}} & \mathbf{y} \\ \hline 0 & \xi \end{array} \right] =: \mathbf{LU} .$$

Then solve

- ①  $\tilde{\mathbf{L}}\mathbf{y} = \mathbf{b}$   $\rightarrow$  provides  $\mathbf{y} \in \mathbb{K}^n$ ,
- ②  $\mathbf{x}^\top \tilde{\mathbf{U}} = \mathbf{a}^\top$   $\rightarrow$  provides  $\mathbf{x} \in \mathbb{K}^n$ ,
- ③  $\mathbf{x}^\top \mathbf{y} + \xi = \alpha$   $\rightarrow$  provides  $\xi \in \mathbb{K}$ .

Regularity of  $\mathbf{A}$  involves  $\xi \neq 0$  (why?) so that  $\mathbf{U}$  will be regular, too.

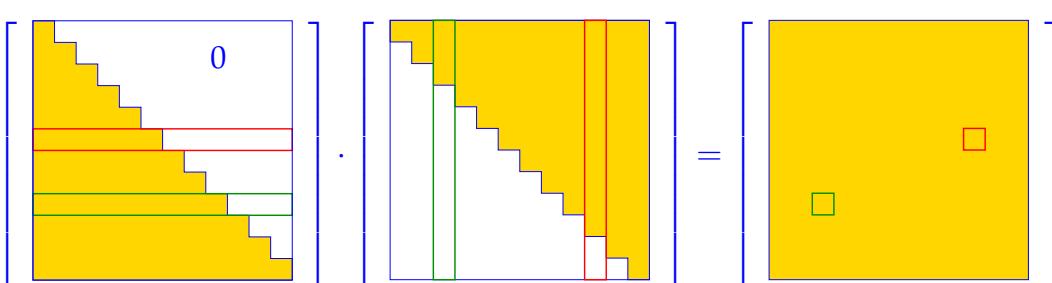
### (1.6.36) Uniqueness of LU-decomposition

Regular upper triangular matrices and normalized lower triangular matrices form *matrix groups* ( $\rightarrow$  Lemma 1.3.9). Their only common element is the identity matrix.

$$\mathbf{L}_1 \mathbf{U}_1 = \mathbf{L}_2 \mathbf{U}_2 \Rightarrow \mathbf{L}_2^{-1} \mathbf{L}_1 = \mathbf{U}_2 \mathbf{U}_1^{-1} = \mathbf{I} .$$

### (1.6.37) Basic algorithm for computing LU-decomposition

A direct way to determine the factor matrices of the LU-decomposition [34, Sect. 3.1], [63, Sect. 3.3.3]:

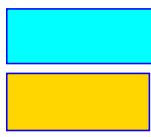
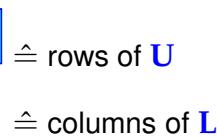


$$\mathbf{LU} = \mathbf{A} \Rightarrow a_{ik} = \sum_{j=1}^{\min\{i,k\}} l_{ij}u_{jk} = \begin{cases} \sum_{j=1}^{i-1} l_{ij}u_{jk} + 1 \cdot u_{ik} & , \text{ if } i \leq k , \\ \sum_{j=1}^{k-1} l_{ij}u_{jk} + l_{ik}u_{kk} & , \text{ if } i > k . \end{cases} \quad (1.6.38)$$

- row by row computation of  $\mathbf{U}$
- column by column computation of  $\mathbf{L}$

Entries of  $\mathbf{A}$  can be replaced with those of  $\mathbf{L}$ ,  $\mathbf{U}$ !  
(so-called *in situ/in place* computation)

(Crout's algorithm, [34, Alg. 3.1])

  $\hat{=}$  rows of  $\mathbf{U}$   
  $\hat{=}$  columns of  $\mathbf{L}$

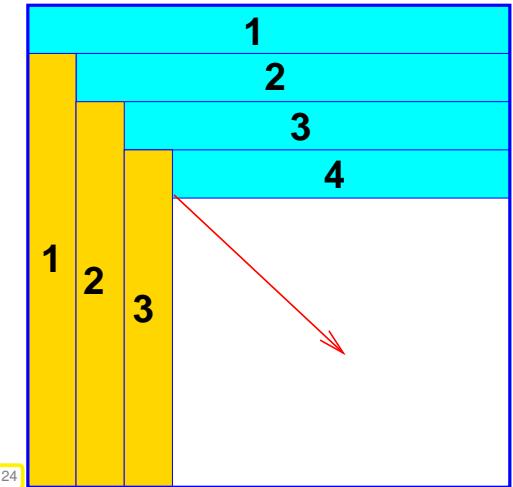


Fig. 24

LU-factorization = “inversion” of matrix multiplication:

LU-factorization

```

1 function [L,U] = lufak(A)
2 % Algorithm of Crout:
3 % LU-factorization of A ∈ Kn,n
4 n = size(A,1); if (size(A,2) ~= n), error('n ~= m'); end
5 L = eye(n); U = zeros(n,n);
6 for k=1:n
7 % Compute row of U
8   for j=k:n
9     U(k,j) = A(k,j) -
10      L(k,1:k-1)*U(1:k-1,j);
11   end
12 % Compute column of L
13   for i=k+1:n
14     L(i,k) = (A(i,k) -
15       L(i,1:k-1)*U(1:k-1,k)) /
16       U(k,k);
17   end
18 end

```

matrix multiplication  $\mathbf{L} \cdot \mathbf{U}$

```

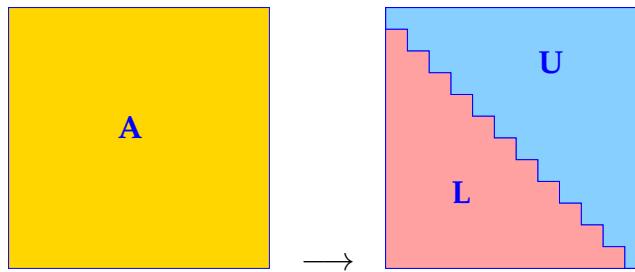
1 function A = lumult(L,U)
2 % Multiplication of normalized
3 % lower/upper triangular matrices
4 n = size(L,1); A = zeros(n,n);
5 if ((size(L,2) ~= n) ||
6   (size(U,1) ~= n) ||
7   (size(U,2) ~= n))
8   error('size mismatch'); end
9 for k=n:-1:1
10   for j=k:n
11     A(k,j) = U(k,j) +
12       L(k,1:k-1)*U(1:k-1,j);
13   end
14   for i=k+1:n
15     A(i,k) =
16       L(i,1:k-1)*U(1:k-1,k) +
17       L(i,k)*U(k,k);
18   end
19 end

```

Observe: Solving for entries  $L(i,k)$  of  $\mathbf{L}$  and  $U(k,j)$  of  $\mathbf{U}$  in the multiplication of an upper triangular and normalized lower triangular matrix ( $\rightarrow$  MATLAB function `lumult`) yields the algorithm for LU-factorization ( $\rightarrow$  MATLAB function `lufak`).

asymptotic complexity of LU-factorization of  $\mathbf{A} \in \mathbb{R}^{n,n} = \frac{1}{3}n^3 + O(n^2) = O(n^3)$

(1.6.39)

**Remark 1.6.40 (In-situ LU-decomposition)**

Replace entries of **A** with entries of **L** (strict lower triangle) and **U** (upper triangle).

**Remark 1.6.41 (Recursive LU-factorization)**

Recall: recursive view of Gaussian elimination → Rem. 1.6.30

In light of the close relationship between Gaussian elimination and LU-factorization there will also be a recursive version of LU-factorization.

Recursive **in situ** (in place) LU-decomposition of  $\mathbf{A} \in \mathbb{R}^{n,n}$  (without pivoting):

**L, U** stored in place of **A**:

```

1 function [L,U] =
  lurecdriver(A)
2 A = lurec(A);
3 % post-processing:
4 % extract L and U
5 U = triu(A);
6 L = tril(A,-1) + eye(size(A));

```

```

1 function A = lurec(A)
2 % insitu recursive LU-factorization
3 if size(A,1)>1
4   fac = A(2:end,1)/A(1,1);
5   C = lurec(A(2:end,2:end)...
6     -fac*A(1,2:end));
7   A=[A(1,:);fac,C];
8 end

```

Refer to (1.6.31) to understand **lurec**: the rank-1 modification of the lower  $(n-1) \times (n-1)$ -block of the matrix is done in lines 5-6 of the code.

**(1.6.42) Using LU-factorization to solve a linear system of equations**

Solving an  $n \times n$  linear system of equations by LU-factorization:

- |                              |  |
|------------------------------|--|
| $\mathbf{Ax} = \mathbf{b}$ : | ① <b>LU</b> -decomposition $\mathbf{A} = \mathbf{LU}$ , #elementary operations $\frac{1}{3}n(n-1)(n+1)$<br>② <b>forward substitution</b> , solve $\mathbf{Lz} = \mathbf{b}$ , #elementary operations $\frac{1}{2}n(n-1)$<br>③ <b>backward substitution</b> , solve $\mathbf{Ux} = \mathbf{z}$ , #elementary operations $\frac{1}{2}n(n+1)$ |
|------------------------------|--|

➤ asymptotic complexity: (in leading order) the same as for Gaussian elimination

However, the perspective of LU-factorization reveals that the solution of linear systems of equations can be split into two separate phases with different asymptotic complexity in terms of the number  $n$  of unknowns:

**setup phase**  
(factorization)  
Cost:  $O(n^3)$

+

**elimination phase**  
(forward/backward substitution)  
Cost:  $O(n^2)$

### Remark 1.6.43 (Rationale for using LU-decomposition in algorithms)

Gauss elimination and LU-factorization for the solution of a linear system of equations ( $\rightarrow$  § 1.6.42) are equivalent and only differ in the ordering of the steps.

Then, why is it important to know about LU-factorization?

Because in the case of LU-factorization the expensive forward elimination and the less expensive (forward/backward) substitutions are separated, which sometimes can be exploited to reduce computational cost, as highlighted in Rem. 1.6.87 below.

### Remark 1.6.44 ("Partial LU-decompositions" of principal minors)

Principal minor  $\hat{=}$  left upper block of a matrix

The following “visual rule” help identify the structure of the LU-factors of a matrix.

$$\boxed{\begin{bmatrix} & \\ \textcolor{pink}{\square} & \end{bmatrix}} = \boxed{\begin{bmatrix} \textcolor{cyan}{\triangle} & \\ \textcolor{cyan}{\triangle} & \end{bmatrix}} \quad \boxed{\begin{bmatrix} & \\ \textcolor{cyan}{\triangle} & \end{bmatrix}}$$

(1.6.45)

The left-upper blocks of both  $\mathbf{L}$  and  $\mathbf{U}$  in the LU-factorization of  $\mathbf{A}$  depend only on the corresponding left-upper block of  $\mathbf{A}$ !

### Remark 1.6.46 (Block LU-factorization)

In the spirit of Rem. 1.3.15: block perspective of LU-factorization.

Natural in light of the close connection between matrix multiplication and matrix factorization, cf. the relationship between matrix factorization and matrix multiplication found in § 1.6.37:

Block matrix multiplication (1.3.16)  $\cong$  block LU-decomposition:

With  $\mathbf{A}_{11} \in \mathbb{K}^{n,n}$  regular,  $\mathbf{A}_{12} \in \mathbb{K}^{n,m}$ ,  $\mathbf{A}_{21} \in \mathbb{K}^{m,n}$ ,  $\mathbf{A}_{22} \in \mathbb{K}^{m,m}$ :

$$\underbrace{\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}}_{\text{block LU-factorization}} = \underbrace{\begin{bmatrix} \mathbf{I} & 0 \\ \mathbf{A}_{21}\mathbf{A}_{11}^{-1} & \mathbf{I} \end{bmatrix}}_{\text{Schur complement}} \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ 0 & \mathbf{S} \end{bmatrix}, \quad \boxed{\mathbf{S} := \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}} \quad (1.6.47)$$

$\rightarrow$  block Gaussian elimination, see Rem. 1.6.32.

### 1.6.2.3 Pivoting

Known from linear algebra [59, Sect. 1.1]:

$$\begin{array}{ccc} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} & & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_2 \\ b_1 \end{bmatrix} \\ \text{breakdown of Gaussian elimination} & & \text{Gaussian elimination feasible} \\ \text{pivot element} = 0 & & \end{array}$$

Idea (in linear algebra): Avoid zero pivot elements by **swapping rows**

#### Example 1.6.48 (Pivoting and numerical stability $\rightarrow$ [15, Example 3.2.3])

```
% Example: numerical instability without
% pivoting
A = [5.0E-17 , 1; 1 , 1];
b = [1;2];
x1 = A\b,
x2 = gausselim(A,b), % see Code 1.6.29
[L,U] = lufak(A); % see Code ???
z = L\b; x3 = U\z,
```

Output of MATLAB run:

```
x1 = 1
      1
x2 = 0
      1
x3 = 0
      1
```

$$\mathbf{A} = \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \Rightarrow \mathbf{x} = \begin{bmatrix} \frac{1}{1-\epsilon} \\ \frac{1-2\epsilon}{1-\epsilon} \end{bmatrix} \approx \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ for } |\epsilon| \ll 1.$$

**What is wrong with MATLAB?** Needed: insight into **roundoff errors**, which we already have  $\rightarrow$  Section 1.5.3

Armed with knowledge about the behavior of machine numbers and roundoff errors we can now understand what is going on in Ex. 1.6.48

Straightforward LU-factorization: if  $\epsilon \leq \frac{1}{2}\text{EPS}$ , EPS  $\triangleq$  **machine precision**,

$$\blacktriangleright \mathbf{L} = \begin{bmatrix} 1 & 0 \\ \epsilon^{-1} & 1 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} \epsilon & 1 \\ 0 & 1 - \epsilon^{-1} \end{bmatrix} \stackrel{(*)}{=} \widetilde{\mathbf{U}} := \begin{bmatrix} \epsilon & 1 \\ 0 & -\epsilon^{-1} \end{bmatrix} \text{ in } \mathbb{M}! \quad (1.6.49)$$

(\*): because  $\tilde{1}/\text{EPS} = 2/\text{EPS}$ , see Exp. 1.5.34.

► Solution of  $\mathbf{L}\tilde{\mathbf{U}}\mathbf{x} = \mathbf{b}$ :  $\mathbf{x} = \begin{bmatrix} 2\epsilon \\ 1-2\epsilon \end{bmatrix}$  (meaningless result !)

LU-factorization after swapping rows:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ \epsilon & 1 \end{bmatrix} \Rightarrow \mathbf{L} = \begin{bmatrix} 1 & 0 \\ \epsilon & 1 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{bmatrix} = \tilde{\mathbf{U}} := \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \text{ in } \mathbb{M}. \quad (1.6.50)$$

► Solution of  $\mathbf{L}\tilde{\mathbf{U}}\mathbf{x} = \mathbf{b}$ :  $\mathbf{x} = \begin{bmatrix} 1+2\epsilon \\ 1-2\epsilon \end{bmatrix}$  (sufficiently accurate result !)

no row swapping,  $\rightarrow$  (1.6.49):  $\mathbf{L}\tilde{\mathbf{U}} = \mathbf{A} + \mathbf{E}$  with  $\mathbf{E} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$  ► unstable !

after row swapping,  $\rightarrow$  (1.6.50):  $\mathbf{L}\tilde{\mathbf{U}} = \tilde{\mathbf{A}} + \mathbf{E}$  with  $\mathbf{E} = \begin{bmatrix} 0 & 0 \\ 0 & \epsilon \end{bmatrix}$  ► stable !

Introduction to the notion of **stability**  $\rightarrow$  ??, Def. 1.5.80, see also [15, Sect. 2.3].

Suitable pivoting essential for controlling impact of roundoff errors  
on Gaussian elimination ( $\rightarrow$  Section 1.5.5, [59, Sect. 2.5])

### Example 1.6.51 (Gaussian elimination with pivoting for $3 \times 3$ -matrix)

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 2 & -3 & 2 \\ 1 & 24 & 0 \end{bmatrix} \xrightarrow{\textcircled{1}} \begin{bmatrix} 2 & -3 & 2 \\ 1 & 2 & 2 \\ 1 & 24 & 0 \end{bmatrix} \xrightarrow{\textcircled{2}} \begin{bmatrix} 2 & -3 & 2 \\ 0 & 3.5 & 1 \\ 0 & 25.5 & -1 \end{bmatrix} \xrightarrow{\textcircled{3}} \begin{bmatrix} 2 & -7 & 2 \\ 0 & 25.5 & -1 \\ 0 & 3.5 & 1 \end{bmatrix} \xrightarrow{\textcircled{4}} \begin{bmatrix} 2 & -7 & 2 \\ 0 & 25.5 & -1 \\ 0 & 0 & 1.373 \end{bmatrix}$$

- ①: swap rows 1 & 2.
- ②: elimination with top row as pivot row
- ③: swap rows 2 & 3
- ④: elimination with 2nd row as pivot row

### (1.6.52) Algorithm: Gaussian elimination with partial pivoting

#### MATLAB-code 1.6.53: Gaussian elimination with pivoting: extension of Code 1.6.23

```

1 function x = gepiv(A,b)
2 % Solving an LSE  $\mathbf{Ax} = \mathbf{b}$  by Gaussian elimination with partial pivoting
3 n = size(A,1); A = [A,b]; %
4 % Forward elimination by rank-1 modification, see Rem. 1.6.30
5 for k=1:n-1
6 [p,j] = max(abs(A(k:n,k))./max(abs(A(k:n,k:n))'))'; %
7 if (p < eps*norm(A(k:n,k:n),1)), %
8 disp('A nearly singular'); end

```

```

9   A([k, j+k-1], k:n+1) = A([j+k-1, k], k:n+1);
10  A(k+1:n, k+1:n+1) =
11    A(k+1:n, k+1:n+1) - (A(k+1:n, k) * A(k, k+1:n+1)) / A(k, k); %
12  end
13  % Back substitution (same as in Code 1.6.23)
14  A(n, n+1) = A(n, n+1) / A(n, n);
15  for i=n-1:-1:1
16    A(i, n+1) = (A(i, n+1) - A(i, i+1:n) * A(i+1:n, n+1)) / A(i, i);
17  end
18  x = A(:, n+1); %

```

choice of pivot row index  $j$  (Line 6 of code): relatively largest pivot [59, Sect. 2.5],

$$j \in \{k, \dots, n\} \text{ such that } \frac{|a_{ji}|}{\max\{|a_{jl}|, l = k, \dots, n\}} \rightarrow \max \quad (1.6.54)$$

for  $k = j, k \in \{i, \dots, n\}$ : partial pivoting

Explanations to Code 1.6.53:

Line 3: Augment matrix  $\mathbf{A}$  by right hand side vector  $\mathbf{b}$ , see comments on Code 1.6.23 for explanations.

Line 6: Select index  $j$  for pivot row according to the recipe of partial pivoting, see (1.6.54).

Note: Inefficient implementation above (too many comparisons)! Try to do better!

Line 7: If the pivot element is still very small relative to the norm of the matrix, then we have encountered an entire column that is close to zero. Gaussian elimination may not be possible in a stable fashion for this matrix; warn user and terminate.

Line 9: A way to swap rows of a matrix in MATLAB.

Line 10: Forward elimination by means of rank-1-update, see (1.6.31).

Line 17: As in Code 1.6.23: after back substitution last column of augmented matrix supplies solution  $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ .

### (1.6.55) Algorithm: LU-factorization with pivoting

Recall: close relationship between Gaussian elimination and LU-factorization

➤ LU-factorization with pivoting? Of course, just by rearranging the operations of Gaussian forward elimination with pivoting.

MATLAB-code for recursive in place LU-factorization of  $\mathbf{A} \in \mathbb{R}^{n,n}$  with partial pivoting:

#### MATLAB-code 1.6.56: recursive LU-factorization with partial pivoting

```
1 function A = gsrecpiv(A)
```

```

2 | n = size(A,1);
3 | if (n > 1)
4 | [p,j] = max(abs(A(:,1))./max(abs(A)'))'; %
5 | if (p < eps*norm(A(:,1:n),1)), disp('A nearly singular'); end %
6 | A([1,j],:) = A([j,1],:); %
7 | fac = A(2:end,1)/A(1,1); %
8 | C = gsrecpiv(A(2:end,2:end)-fac*A(1,2:end)); %
9 | A = [A(1,:); -fac, C];
10| end

```

Explanations to Code 1.6.56:

Line 4: Find the *relatively largest* pivot element  $p$  and the index  $j$  of the corresponding row of the matrix, see (1.6.54)

Line 5: If the pivot element is still very small relative to the norm of the matrix, then we have encountered an entire column that is close to zero. The matrix is (close to) singular and LU-factorization does not exist.

Line 6: Swap the first and the  $j$ -th row of the matrix.

Line 7: Initialize the vector of multiplier.

Line 8: Call the routine for the upper right  $(n-1) \times (n-1)$ -block of the matrix after subtracting suitable multiples of the first row from the other rows, cf. Rem. 1.6.30 and Rem. 1.6.41.

Line 9: Reassemble the parts of the LU-factors. The vector of multipliers yields a column of  $\mathbf{L}$ , see Ex. 1.6.33.

### (1.6.57) Algorithm: in-situ LU-factorization

C++-code (non-recursive implementation, of course) for in-situ LU-factorization ( $\rightarrow$  Sect. 1.6.2.2) of  $\mathbf{A} \in \mathbb{R}^{n,n}$  with **partial pivoting** ➤

Row permutations recorded in vector  $\mathbf{p}$ !

Usual choice of pivot, pivot row index  $j$  according to (1.6.54)

Note: row swapping  $\leftrightarrow$  pointer swapping

```

template<class Matrix>
void LU(Matrix &A, std::vector<int> &p) {
    int n = A.dim();
    for(int i=1;i<=n;i++) p[i] = i;
    for(int i=1;i<n;i++) {
        Choose index j ∈ {i,...,n} of pivot row
        std::swap(p[i],p[j]);
        for(int k=i+1;k<=n;k++) {
            A(p[k],i) /= A(p[i],i);
            for(int l=i+1;l<=n;l++) {
                A(p[k],l) -= A(p[k],i)*A(p[i],l);
            } } }
    }
```

### Remark 1.6.58 (Rationale for partial pivoting policy (1.6.54) → [59, Page 47])

Why *relatively* largest pivot element in (1.6.54)? scaling invariance desirable

Scale linear system of equations from Ex. 1.6.48:

$$\begin{bmatrix} 2/\epsilon & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 & 2/\epsilon \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2/\epsilon \\ 1 \end{bmatrix}$$

No row swapping, if absolutely largest pivot element is used:

$$\begin{bmatrix} 2 & 2/\epsilon \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 2/\epsilon \\ 0 & 1 - 2/\epsilon \end{bmatrix} \doteq \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 2/\epsilon \\ 0 & -2/\epsilon \end{bmatrix} \text{ in } \mathbb{M}.$$

### MATLABscript 1.6.59: Effect of pivoting

```

1 % Example: importance of
2 % scale-invariant pivoting
3 epsilon = 5.0E-17;
4 A = [epsilon, 1, 1, 1]; b = [1; 2];
5 D = [1/epsilon, 0, 0, 1];
6 A = D*A; b = D*b;
7 x1 = A\b, % MATLAB internal Gaussian
8 % elimination
9 x2 = gausselim(A,b), % see Code 1.6.29
10 [L,U] = lufak(A); % see Code ???
11 z = L\b; x3 = U\z,

```

Output of MATLAB run:

```

x1 = 1
      1
x2 = 0
      1
x3 = 0
      1

```

## Pivoting: Theoretical perspective

### Definition 1.6.60. Permutation matrix

An  $n$ -permutation,  $n \in \mathbb{N}$ , is a bijective mapping  $\pi : \{1, \dots, n\} \mapsto \{1, \dots, n\}$ . The corresponding permutation matrix  $\mathbf{P}_\pi \in \mathbb{K}^{n,n}$  is defined by

$$(\mathbf{P}_\pi)_{ij} = \begin{cases} 1 & \text{if } j = \pi(i), \\ 0 & \text{else.} \end{cases}$$

permutation  $(1, 2, 3, 4) \mapsto (1, 3, 2, 4) \hat{=} \mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ .

Note:

- \*  $\mathbf{P}^\top = \mathbf{P}^{-1}$  for any permutation matrix  $\mathbf{P}$  ( $\rightarrow$  permutation matrices orthogonal/unitary)
- \*  $\mathbf{P}_\pi \mathbf{A}$  effects  $\pi$ -permutation of rows of  $\mathbf{A} \in \mathbb{K}^{n,m}$
- \*  $\mathbf{A} \mathbf{P}_\pi$  effects  $\pi$ -permutation of columns of  $\mathbf{A} \in \mathbb{K}^{m,n}$

### Lemma 1.6.61. Existence of LU-factorization with pivoting $\rightarrow$ [15, Thm. 3.25], [42, Thm. 4.4]

For any regular  $\mathbf{A} \in \mathbb{K}^{n,n}$  there is a permutation matrix ( $\rightarrow$  Def. 1.6.60)  $\mathbf{P} \in \mathbb{K}^{n,n}$ , a normalized lower triangular matrix  $\mathbf{L} \in \mathbb{K}^{n,n}$ , and a regular upper triangular matrix  $\mathbf{U} \in \mathbb{K}^{n,n}$  ( $\rightarrow$  Def. 1.1.5), such that  $\mathbf{PA} = \mathbf{LU}$ .

*Proof.* (by induction)

Every regular matrix  $\mathbf{A} \in \mathbb{K}^{n,n}$  admits a row permutation encoded by the permutation matrix  $\mathbf{P} \in \mathbb{K}^{n,n}$ , such that  $\mathbf{A}' := (\mathbf{A})_{1:n-1, 1:n-1}$  is regular (why?).

By induction assumption there is a permutation matrix  $\mathbf{P}' \in \mathbb{K}^{n-1,n-1}$  such that  $\mathbf{P}'\mathbf{A}'$  possesses a LU-factorization  $\mathbf{A}' = \mathbf{L}'\mathbf{U}'$ . There are  $\mathbf{x}, \mathbf{y} \in \mathbb{K}^{n-1}$ ,  $\gamma \in \mathbb{K}$  such that

$$\begin{bmatrix} \mathbf{P}' & 0 \\ 0 & 1 \end{bmatrix} \mathbf{PA} = \begin{bmatrix} \mathbf{P}' & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{A}' & \mathbf{x} \\ \mathbf{y}^\top & \gamma \end{bmatrix} = \begin{bmatrix} \mathbf{L}'\mathbf{U}' & \mathbf{x} \\ \mathbf{y}^\top & \gamma \end{bmatrix} = \begin{bmatrix} \mathbf{L}' & 0 \\ \mathbf{c}^\top & 1 \end{bmatrix} \begin{bmatrix} \mathbf{U} & \mathbf{d} \\ 0 & \alpha \end{bmatrix},$$

if we choose

$$\mathbf{d} = (\mathbf{L}')^{-1}\mathbf{x}, \quad \mathbf{c} = (\mathbf{u}')^{-T}\mathbf{y}, \quad \alpha = \gamma - \mathbf{c}^\top \mathbf{d},$$

which is always possible.  $\square$

### Example 1.6.62 (Ex. 1.6.51 cnt'd)

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 2 & -3 & 2 \\ 1 & 24 & 0 \end{bmatrix} \xrightarrow{\textcircled{1}} \begin{bmatrix} 2 & -3 & 2 \\ 1 & 2 & 2 \\ 1 & 24 & 0 \end{bmatrix} \xrightarrow{\textcircled{2}} \begin{bmatrix} 2 & -3 & 2 \\ 0 & 3.5 & 1 \\ 0 & 25.5 & -1 \end{bmatrix} \xrightarrow{\textcircled{3}} \begin{bmatrix} 2 & -7 & 2 \\ 0 & 25.5 & -1 \\ 0 & 3.5 & 1 \end{bmatrix} \xrightarrow{\textcircled{4}} \begin{bmatrix} 2 & -7 & 2 \\ 0 & 25.5 & -1 \\ 0 & 0 & 1.373 \end{bmatrix}$$

↓

$$\mathbf{U} = \begin{bmatrix} 2 & -3 & 2 \\ 0 & 25.5 & -1 \\ 0 & 0 & 1.373 \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0.1373 & 1 \end{bmatrix}, \quad \mathbf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

Two permutations: in step **①** swap rows #1 and #2, in step **③** swap rows #2 and #3. Apply these swaps to the identity matrix and you will recover  $\mathbf{P}$ . See also [15, Ex. 3.30].

MATLAB function:

`[L, U, P] = lu(A) (P = permutation matrix)`

### Remark 1.6.63 (Row swapping commutes with forward elimination)

Any kind of pivoting only involves comparisons and row/column permutations, but no arithmetic operations on the matrix entries. This makes the following observation plausible:

The LU-factorization of  $\mathbf{A} \in \mathbb{K}^{n,n}$  with partial pivoting by § 1.6.55 is *numerically equivalent* to the LU-factorization of  $\mathbf{PA}$  without pivoting ( $\rightarrow$  Code in § 1.6.37), when  $\mathbf{P}$  is a permutation matrix gathering the row swaps entailed by partial pivoting.

*numerically equivalent*  $\hat{=}$  same result when executed with the same machine arithmetic

- The above statement means that whenever we study the impact of roundoff errors on LU-factorization it is safe to consider only the basic version without pivoting, because we can always assume that row swaps have been conducted beforehand.

## 1.6.3 Stability of Gaussian Elimination

### (1.6.64) Probing stability of a direct solver for LSE

Assume that you have downloaded a direct solver for a general (dense) linear system of equations  $\mathbf{Ax} = \mathbf{b}$ ,  $\mathbf{A} \in \mathbb{K}^{n,n}$  regular,  $\mathbf{b} \in \mathbb{K}^n$ . When given the data  $\mathbf{A}$  and  $\mathbf{b}$  it returns the perturbed solution  $\tilde{\mathbf{x}}$ . How can we tell that  $\tilde{\mathbf{x}}$  is the exact solution of a linear system with slightly perturbed data (in the sense of a tiny relative error of size  $\approx \text{EPS}$ ,  $\text{EPS}$  the machine precision, see § 1.5.28). That is, how can we tell that  $\tilde{\mathbf{x}}$  is an acceptable solution in the sense of backward error analysis, cf. Def. 1.5.80. A similar question was explored in Ex. 1.5.81 for matrix  $\times$  vector multiplication.

- ①  $\mathbf{x} - \tilde{\mathbf{x}}$  accounted for by perturbation of right hand side:

$$\begin{aligned} \mathbf{Ax} &= \mathbf{b} \\ \mathbf{A}\tilde{\mathbf{x}} &= \mathbf{b} + \Delta\mathbf{b} \end{aligned} \quad \Rightarrow \quad \Delta\mathbf{b} = \mathbf{A}\tilde{\mathbf{x}} - \mathbf{b} =: -\mathbf{r} \quad (\text{residual, Def. 1.6.72}) .$$

Hence,  $\tilde{\mathbf{x}}$  can be accepted as a solution, if  $\frac{\|\mathbf{r}\|}{\|\mathbf{b}\|} \leq Cn^3 \cdot \text{EPS}$ , for some small constant  $C \approx 1$ , see Def. 1.5.80. Here,  $\|\cdot\|$  can be any vector norm on  $\mathbb{K}^n$ .

- ②  $\mathbf{x} - \tilde{\mathbf{x}}$  accounted for by perturbation of system matrix:

$$\begin{aligned} \mathbf{Ax} &= \mathbf{b} , \quad (\mathbf{A} + \Delta\mathbf{A})\tilde{\mathbf{x}} = \mathbf{b} \\ [ \quad \text{try perturbation} \quad \Delta\mathbf{A} = \mathbf{u}\tilde{\mathbf{x}}^H, \quad \mathbf{u} \in \mathbb{K}^n \quad ] \\ \blacktriangleright \quad \mathbf{u} &= \frac{\mathbf{r}}{\|\mathbf{x}\|_2^2} \quad \Rightarrow \quad \Delta\mathbf{A} = \frac{\mathbf{r}\tilde{\mathbf{x}}^H}{\|\mathbf{x}\|_2^2} . \end{aligned}$$

As in Ex. 1.5.81 we find

$$\frac{\|\Delta\mathbf{A}\|_2}{\|\mathbf{A}\|_2} = \frac{\|\mathbf{r}\|}{\|\mathbf{A}\|_2\|\tilde{\mathbf{x}}\|_2} \leq \frac{\|\mathbf{r}\|_2}{\|\mathbf{A}\tilde{\mathbf{x}}\|_2} .$$

Thus,  $\tilde{\mathbf{x}}$  is ok in the sense of backward error analysis, if  $\frac{\|\mathbf{r}\|}{\|\mathbf{A}\tilde{\mathbf{x}}\|} \leq Cn^3 \cdot \text{EPS}$ .

In any case, we have to examine the relative norm of the residual  $\mathbf{r} := \mathbf{b} - \mathbf{A}\tilde{\mathbf{x}}$ .

The roundoff error analysis of Gaussian elimination based Ass. 1.5.31 is rather involved. Here we merely summarise the results:

Simplification: equivalence of Gaussian elimination and LU-factorization extends to machine arithmetic, cf. Sect. 1.6.2.2

#### Lemma 1.6.65. Equivalence of Gaussian elimination and LU-factorization

The following algorithms for solving the LSE  $\mathbf{Ax} = \mathbf{b}$  ( $\mathbf{A} \in \mathbb{K}^{n,n}$ ,  $\mathbf{b} \in \mathbb{K}^n$ ) are numerically equivalent:

- ① Gaussian elimination (forward elimination and back substitution) without pivoting, see Algorithm 1.6.22.
- ② LU-factorization of  $\mathbf{A}$  ( $\rightarrow$  Code in § 1.6.37) followed by forward and backward substitution, see Algorithm 1.6.42.

Rem. 1.6.63 ➤ sufficient to consider LU-factorization without pivoting

A profound roundoff analysis of Gaussian elimination/LU-factorization can be found in [28, Sect. 3.3 & 3.5] and [43, Sect. 9.3]. A less rigorous, but more lucid discussion is given in [81, Lecture 22].

Here we only quote a result due to Wilkinson, [43, Thm. 9.5]:

### Theorem 1.6.66. Stability of Gaussian elimination with partial pivoting

Let  $\mathbf{A} \in \mathbb{R}^{n,n}$  be regular and  $\mathbf{A}^{(k)} \in \mathbb{R}^{n,n}$ ,  $k = 1, \dots, n-1$ , denote the intermediate matrix arising in the  $k$ -th step of § 1.6.55 (Gaussian elimination with partial pivoting) when carried out with exact arithmetic.

For the approximate solution  $\tilde{\mathbf{x}} \in \mathbb{R}^n$  of the LSE  $\mathbf{Ax} = \mathbf{b}$ ,  $\mathbf{b} \in \mathbb{R}^n$ , computed as in § 1.6.55 (based on machine arithmetic with machine precision  $\text{EPS}$ , → Ass. 1.5.31) there is  $\Delta\mathbf{A} \in \mathbb{R}^{n,n}$  with

$$\|\Delta\mathbf{A}\|_\infty \leq n^3 \frac{3\text{EPS}}{1 - 3n\text{EPS}} \rho \|\mathbf{A}\|_\infty, \quad \rho := \frac{\max_{i,j,k} |(\mathbf{A}^{(k)})_{ij}|}{\max_{i,j} |(\mathbf{A})_{ij}|},$$

such that  $(\mathbf{A} + \Delta\mathbf{A})\tilde{\mathbf{x}} = \mathbf{b}$ .

$\rho$  “small” ➔ Gaussian elimination with partial pivoting is stable (→ Def. 1.5.80)

If  $\rho$  is “small”, the computed solution of a LSE can be regarded as the exact solution of a LSE with “slightly perturbed” system matrix (perturbations of size  $\mathcal{O}(n^3\text{EPS})$ ).



Bad news:

exponential growth  $\rho \sim 2^n$  is possible !

### Example 1.6.67 (Wilkinson's counterexample)

$$\begin{aligned} n=10: \\ a_{ij} = \begin{cases} 1 & , \text{ if } i = j \vee j = n , \\ -1 & , \text{ if } i > j , \\ 0 & \text{ else.} \end{cases} , \quad \mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & 1 & 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 \end{bmatrix} \end{aligned}$$

Partial pivoting does not trigger row permutations !

$$\blacktriangleright \quad \mathbf{A} = \mathbf{LU}, \quad l_{ij} = \begin{cases} 1 & , \text{ if } i = j , \\ -1 & , \text{ if } i > j , \\ 0 & \text{ else} \end{cases} \quad u_{ij} = \begin{cases} 1 & , \text{ if } i = j , \\ 2^{i-1} & , \text{ if } j = n , \\ 0 & \text{ else.} \end{cases}$$

► Exponential blow-up of entries of  $\mathbf{U}$  !

Blow-up of entries of  $\mathbf{U}$  !  
 $\Updownarrow (*)$   
 However,  $\text{cond}_2(\mathbf{A})$  is small!

▷ Instability of Gaussian elimination!

#### MATLAB-code 1.6.68: GE for “Wilkinson system”

```

1 res = [];
2 for n = 10:10:1000
3   A = [ tril (-ones (n,n-1)) + 2 * eye (n-1);
4         zeros (1,n-1) ], ones (n,1) ];
5   x = ((-1).^(1:n))';
6   relerr = norm (A \ (A*x) - x) / norm (x);
7   res = [res; n, relerr];
8 end
9 plot (res (:,1), res (:,2), 'm-*');

```

- (\*) If  $\text{cond}_2(\mathbf{A})$  was huge, then big errors in the solution of a linear system can be caused by small perturbations of either the system matrix or the right hand side vector, see (1.6.75) and the message of Thm. 1.6.13, (1.6.16). In this case, a stable algorithm can obviously produce a grossly “wrong” solution, as was already explained after (1.6.16).

Hence, lack of stability of Gaussian elimination will only become apparent for linear systems with well-conditioned system matrices.

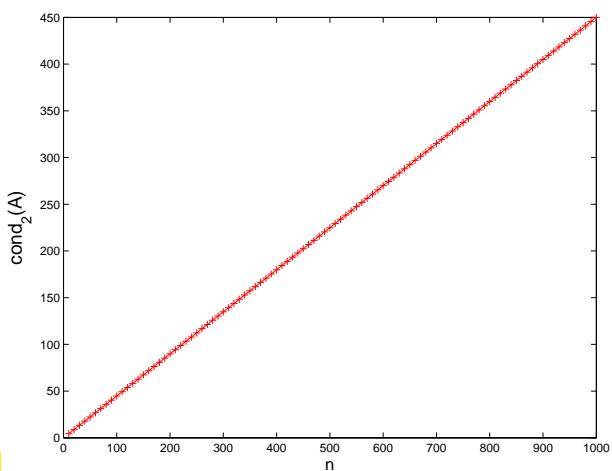


Fig. 25

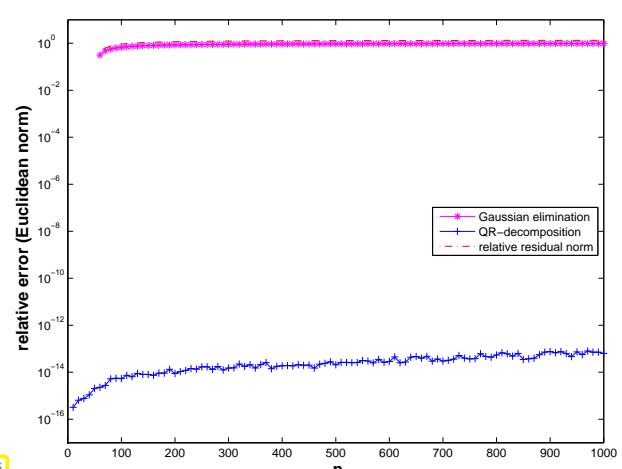
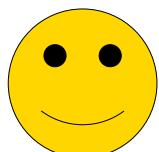


Fig. 26

These observations match Thm. 1.6.66, because in this case we encounter an exponential growth of  $\rho = \rho(n)$ , see Ex. 1.6.67.



Observation: In practice  $\rho$  (almost) always grows only mildly (like  $O(\sqrt{n})$ ) with  $n$

Discussion in [81, Lecture 22]: growth factors larger than the order  $O(\sqrt{n})$  are exponentially rare in certain relevant classes of *random matrices*.

#### Example 1.6.69 (Stability by small random perturbations)

#### MATLAB-code 1.6.70: Perturbations cure instability of Gaussian elimination

```

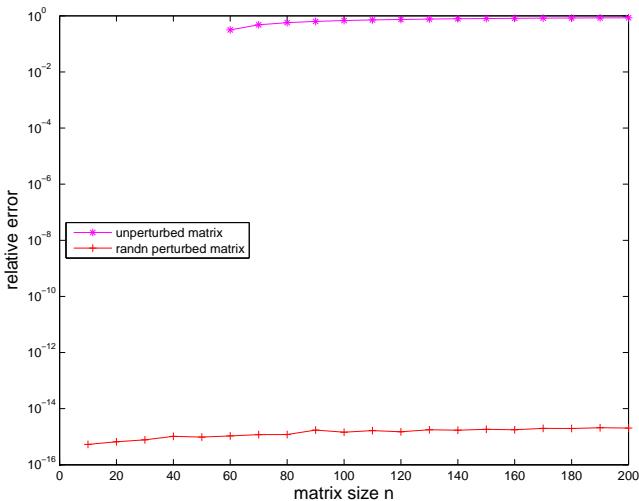
1 % Curing Wilkinson's counterexample by random perturbation

```

```

2 % Theory: Spielman and Teng
3 res = [];
4 for n=10:10:200
5 % Build Wilkinson matrix
6 A = [ tril (-ones(n,n-1))+2*[eye(n-1);
7 zeros(1,n-1)], ones(n,1) ];
8 % imposed solution
9 x = ((-1).^(1:n))';
10 relerr = norm(A\ (A*x)-x)/norm(x);
11 % Randomly perturbed Wilkinson matrix by matrix with iid
12 % N(0,eps) distributed entries
13 Ap = A + eps*randn(size(A));
14 relerrp = norm(Ap\ (A*x)-x)/norm(x);
15 res = [res; n relerr relerrp];
16 end
17 semilogy(res(:,1),res(:,2),'m-*',res(:,1),res(:,3),'r-+');
18 xlabel('matrix size n','fontsize',14);
19 ylabel('relative error','fontsize',14);
20 legend('unperturbed matrix','randn perturbed
matrix','location','west');
21
22 print -depsc2 '../PICTURES/wilkpert.eps';

```



Recall the statement made above about “improbability” of matrices for which Gaussian elimination with partial pivoting is unstable. This is now matched by the observation that a *tiny random* perturbation of the matrix (almost certainly) cures the problem. This is investigated by the brand-new field of **smoothed analysis** of numerical algorithms, see [69].

Gaussian elimination/LU-factorization with partial pivoting is stable (\*)  
(for all practical purposes) !

(\*): stability refers to maximum norm  $\|\cdot\|_\infty$ .

### Experiment 1.6.71 (Conditioning and relative error → Ex. 1.6.73 cnt'd)

In the discussion of numerical stability (→ Def. 1.5.80, Rem. 1.5.83) we have seen that a stable algorithm may produce results with large errors for ill-conditioned problems. The conditioning of the problem of

solving a linear system of equations is determined by the condition number ( $\rightarrow$  Def. 1.6.15) of the system matrix, see Thm. 1.6.13.

Hence, for an **ill-conditioned** linear system, whose system matrix is beset with a huge condition number, (stable) Gaussian elimination may return “solutions” with large errors. This will be demonstrated in this experiment.

Numerical experiment with *nearly singular matrix* from Ex. 1.6.73

$$\begin{aligned} \mathbf{A} &= \mathbf{u}\mathbf{v}^T + \epsilon\mathbf{I}, \\ \mathbf{u} &= \frac{1}{3}(1, 2, 3, \dots, 10)^T, \\ \mathbf{v} &= (-1, \frac{1}{2}, -\frac{1}{3}, \frac{1}{4}, \dots, \frac{1}{10})^T \end{aligned}$$

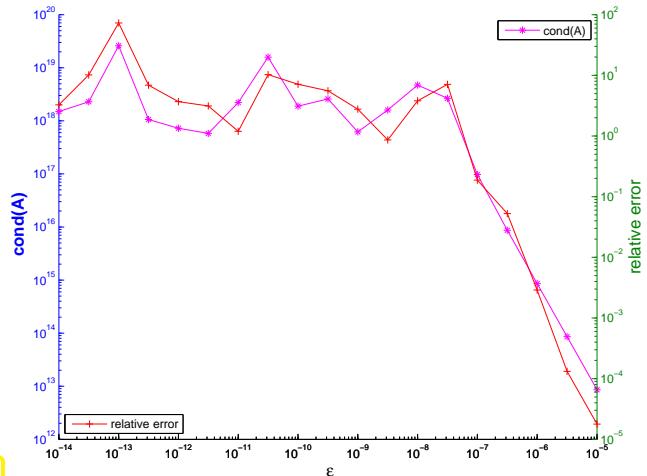


Fig. 28

The practical stability of Gaussian elimination is reflected by the size of a particular vector that can easily be computed after the elimination solver has finished:

In practice *Gaussian elimination/LU-factorization with partial pivoting* produces “relatively small residuals”

### Definition 1.6.72. Residual

Given an approximate solution  $\tilde{\mathbf{x}} \in \mathbb{K}^n$  of the LSE  $\mathbf{Ax} = \mathbf{b}$  ( $\mathbf{A} \in \mathbb{K}^{n,n}$ ,  $\mathbf{b} \in \mathbb{K}^n$ ), its **residual** is the vector

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\tilde{\mathbf{x}}.$$

Simple consideration:

$$(\mathbf{A} + \Delta\mathbf{A})\tilde{\mathbf{x}} = \mathbf{b} \Rightarrow \mathbf{r} = \mathbf{b} - \mathbf{A}\tilde{\mathbf{x}} = \Delta\mathbf{A}\tilde{\mathbf{x}} \Rightarrow \|\mathbf{r}\| \leq \|\Delta\mathbf{A}\| \|\tilde{\mathbf{x}}\|,$$

for any vector norm  $\|\cdot\|$ . This means that, if a direct solver for an LSE is stable in the sense of backward error analysis, that is, the perturbed solution could be obtained as the exact solution for a slightly relatively perturbed system matrix, then the *residual will be (relatively) small*.

### Experiment 1.6.73 (Small residuals by Gaussian elimination)

Numerical experiment with  
nearly singular matrix

$$\mathbf{A} = \mathbf{u}\mathbf{v}^T + \epsilon\mathbf{I},$$

singular rank-1 matrix

with

$$\mathbf{u} = \frac{1}{3}(1, 2, 3, \dots, 10)^T,$$

$$\mathbf{v} = (-1, \frac{1}{2}, -\frac{1}{3}, \frac{1}{4}, \dots, \frac{1}{10})^T$$

### MATLAB-code 1.6.74: small residuals for GE

```

1 n = 10; u = (1:n)'/3; v =
2   (1./u).*(-1).^((1:n)');
3 x = ones(10,1); nx = norm(x, 'inf');
4 result = [];
5 for epsilon = 10.^(-5:-0.5:-14)
6   A = u*v' + epsilon*eye(n);
7   b = A*x; nb = norm(b, 'inf');
8   xt = A\b; % Gaussian elimination
9   r = b - A*xt; % residual
10  result = [result; epsilon,
11    norm(x-xt, 'inf')/nx,
12    norm(r, 'inf')/nb,
13    cond(A, 'inf')];
14 end

```

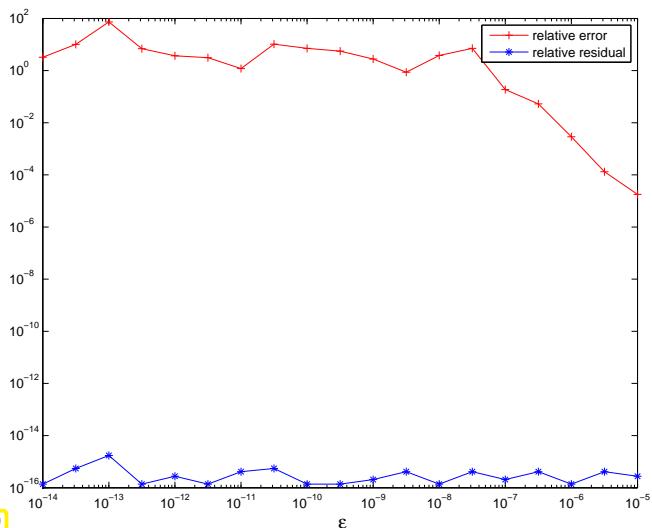


Fig. 29

Observations (w.r.t  $\|\cdot\|_\infty$ -norm)

- \* for  $\epsilon \ll 1$  large relative error in computed solution  $\tilde{\mathbf{x}}$
- \* small residuals for any  $\epsilon$

How can a *large* relative error be reconciled with a *small* relative residual ?

$$\begin{aligned} \mathbf{Ax} = \mathbf{b} &\leftrightarrow \mathbf{A}\tilde{\mathbf{x}} \approx \mathbf{b} \\ \left\{ \begin{array}{l} \mathbf{A}(\mathbf{x} - \tilde{\mathbf{x}}) = \mathbf{r} \\ \mathbf{Ax} = \mathbf{b} \end{array} \right. &\Rightarrow \|\mathbf{x} - \tilde{\mathbf{x}}\| \leq \|\mathbf{A}^{-1}\| \|\mathbf{r}\| \quad \Rightarrow \quad \frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \|\mathbf{A}\| \|\mathbf{A}^{-1}\| \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}. \end{aligned} \quad (1.6.75)$$

➤ If  $\text{cond}(\mathbf{A}) := \|\mathbf{A}\| \|\mathbf{A}^{-1}\| \gg 1$ , then a small relative residual may not imply a small relative error. Also recall the discussion in Exp. 1.6.71.

### Experiment 1.6.76 (Instability of multiplication with inverse)

An important justification for Rem. 1.6.11 is conveyed by this experiment. We again consider the nearly singular matrix from Ex. 1.6.73.

**MATLAB-code 1.6.77: Instability of multiplication with inverse**

```

1 n = 10; u = (1:n)'/3; v = (1./u).*(-1).^u;
2 x = ones(10,1); nx = norm(x,'inf');
3
4 result = [];
5 for epsilon = 10.^(-5:-0.5:-14)
6     A = u*v' + epsilon*rand(n,n);
7     b = A*x; nb = norm(b,'inf');
8     xt = A\b;          % Gaussian elimination
9     r = b - A*xt;      % residualB
10    B = inv(A); xi = B*b;
11    ri = b - A*xi;    % residual
12    R = eye(n) - A*B; % residual
13    result = [result; epsilon, norm(r,'inf')/nb,
14              norm(ri,'inf')/nb, norm(R,'inf')/norm(B,'inf') ];
15 end

```

Computation of the inverse  $\mathbf{B} := \text{inv}(\mathbf{A})$  is affected by roundoff errors, but does not benefit from favorable compensation of roundoff errors as does Gaussian elimination.

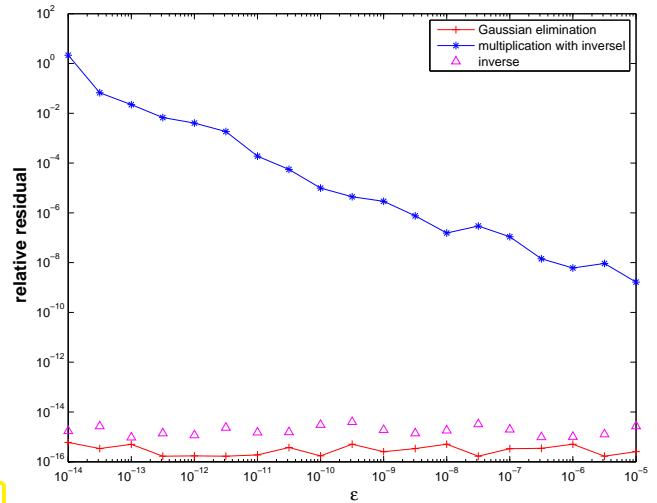


Fig. 30

**1.6.4 Elimination solvers for linear systems of equations**

All direct (\*) solver algorithms for square linear systems of equations  $\mathbf{Ax} = \mathbf{b}$  with given matrix  $\mathbf{A} \in \mathbb{K}^{n,n}$ , right hand side vector  $\mathbf{b} \in \mathbb{K}^n$  and unknown  $\mathbf{x} \in \mathbb{K}^n$  rely on variants of Gaussian elimination with pivoting, see Section 1.6.2.3. Sophisticated, optimised and verified implementations are available in numerical libraries like LAPACK/MKL.

(\*): a direct solver terminates after a predictable finite number of elementary operations for every admissible input.

Never contemplate implementing a general solver for linear systems of equations!

If possible, use algorithms from numerical libraries! ( $\rightarrow$  Exp. 1.6.26)

Therefore, familiarity with details of Gaussian elimination is not required, but one must know when and how to use the library functions and one must be able to assess the computational effort they involve.

### (1.6.78) Computational effort for direct elimination

We repeat the reasoning of § 1.6.24: Gaussian elimination for a general (dense) matrix invariably involves *three nested loops* of length  $n$ , see Code 1.6.23, Code 1.6.53.

#### Theorem 1.6.79. Cost of Gaussian elimination → § 1.6.24

Given a linear system of equations  $\mathbf{Ax} = \mathbf{b}$  with  $\mathbf{A} \in \mathbb{K}^{n,n}$  regular,  $\mathbf{b} \in \mathbb{K}^n$ ,  $n \in \mathbb{N}$ , the asymptotic computational effort ( $\rightarrow$  Def. 1.4.1) for its direct solution by means of Gaussian elimination in terms of the problem size parameter  $n$  is  $O(n^3)$  for  $n \rightarrow \infty$ .

The constant hidden in the Landau symbol can be expected to be rather small ( $\approx 1$ ) as is clear from (1.6.25).

The cost for solving are substantially lower, if certain properties of the matrix  $\mathbf{A}$  are known. This is clear, if  $\mathbf{A}$  is diagonal or orthogonal/unitary. It is also true for triangular matrices ( $\rightarrow$  Def. 1.1.5), because they can be solved by simple back substitution or forward elimination. We recall the observation made in see § 1.6.42.

#### Theorem 1.6.80. Cost for solving triangular systems → § 1.6.24

In the setting of Thm. 1.6.79, the asymptotic computational effort for solving a *triangular* linear system of equations is  $O(n^2)$  for  $n \rightarrow \infty$ .

### (1.6.81) Direct solution of linear systems of equations in MATLAB

Given: system matrix  $\mathbf{A} \in \mathbb{K}^{n,n}$  regular  $\leftrightarrow$   $\mathbf{A}$  ( $n \times n$  MATLAB matrix)  
right hand side vectors  $\mathbf{B} \in \mathbb{K}^{n,\ell}$   $\leftrightarrow$   $\mathbf{B}$  ( $n \times \ell$  MATLAB matrix)  
(corresponds to multiple right hand sides, cf. Code 1.6.29)

linear algebra	MATLAB
$\mathbf{X} = \mathbf{A}^{-1}\mathbf{B} = [\mathbf{A}^{-1}(\mathbf{B})_{:,1}, \dots, \mathbf{A}^{-1}(\mathbf{B})_{:,\ell}]$	$\mathbf{X} = \mathbf{A} \backslash \mathbf{B}$ ("backslash, mldivide")

#### Remark 1.6.82 (Communicating special properties of system matrices in MATLAB)

Sometimes, the coefficient matrix of a linear system of equations is known to have certain analytic properties that a direct solver can exploit to perform elimination more efficiently. These properties may even be impossible to detect by an algorithm, because matrix entries that should vanish exactly might have been perturbed due to roundoff.

Thus, MATLAB furnishes a function, which, unlike  $\backslash$ , allows the user to tell it about properties of the coefficient matrix:

```
Y = linsolve(A, B, opts);
```

`opts` is a record, which may have fields with boolean value `true` as follows:

opts.LT : coefficient matrix is lower triangular  
 opts.UT : coefficient matrix is upper triangular  
 opts.SYM : coefficient matrix is real symmetric or complex Hermitian  
 opts.POSDEF: coefficient matrix is positive definite

### Experiment 1.6.83 (Linsolve versus backslash)

In this numerical experiment we study the gain in efficiency achievable by make the direct solver aware of important matrix properties.

#### MATLAB-code 1.6.84: Direct solver applied to a (nearly) upper triangular matrix

```

1 % MATLAB test script: assessing the gain from using linsolve
2
3 K = 3; % number of runs (to offset OS activity)
4 T = []; % Matrix for recording results
5 for n=2.^4:12
6   % Create test matrix
7   A = triu(diag(1:n) + ones(n,1)*ones(1,n));
8   % Slight perturbation below the diagonal
9   A = A + flipud(diag(eps*rand(n,1))); %
10  b = rand(n,1);
11  t_b = realmax; t_l = realmax;
12  opts.UT = true;
13  for k=1:K
14    tic; xb = A\b; t = toc; t_b = min(t_b,t);
15    tic; xl = linsolve(A,b,opts); t = toc; t_l = min(t_l,t);
16  end
17  T = [T; n t_b t_l norm(xb-xl)];
18 end
19
20 figure; loglog(T(:,1),T(:,2),'r+',T(:,1),T(:,3),'m*');
21 xlabel('{\bf matrix size n}', 'fontsize', 14);
22 ylabel('{\bf runtime for direct solver [s]}', 'fontsize', 14);
23 legend('backslash', 'linsolve(UT)', 'location', 'best');
24 print -depsc2 '../PICTURES/linsolvetest.eps';

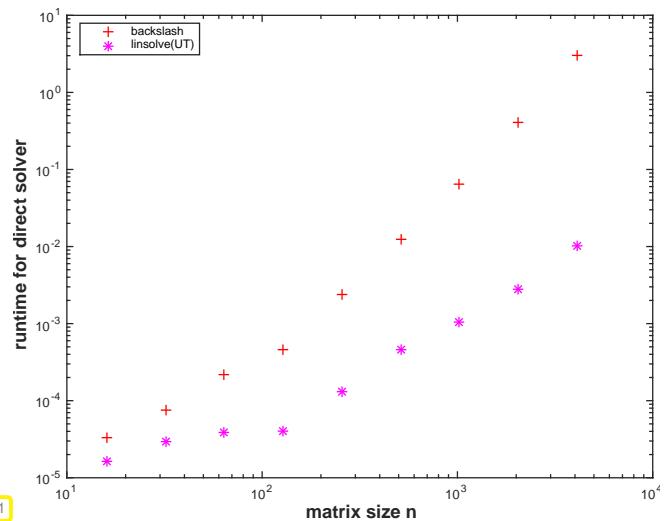
```

Observation:



Being told that only the upper triangular part of the matrix needs to be taken into account, Gaussian elimination reduces to cheap backward elimination, which is much faster than full elimination.

Without the perturbation from Line 9 the gap in runtime is hardly noticeable, because \ can detect that the matrix is upper triangular.



### (1.6.85) Direct solvers for LSE in EIGEN

Invocation of direct solvers in EIGEN is a two stage process:

- ❶ Request a decomposition (LU,QR,LDLT) of the matrix and store it in a temporary “decomposition object”.
- ❷ Perform backward & forward substitutions by calling the `solve()` method of the decomposition object.

#### C++11-code 1.6.86: Direct solution of dense linear system in EIGEN

```

1 #include <Eigen/Dense>
2 using namespace Eigen;
3 using namespace std;
4 ...
5 // Initialize a special invertible matrices
6 MatrixXd mat = MatrixXd::Identity(n,n) +
7     VectorXd::Constant(n,1.0)*RowVectorXd::Constant(n,1.0);
8 cout << "Matrix mat = " << endl << mat << endl;
9 // Multiple right hand side vectors stored in matrix, cf. MATLAB
10 MatrixXd B = MatrixXd::Random(n,2);
11 // Solve linear system using various decompositions
12 MatrixXd X = mat.lu().solve(B);
13 MatrixXd X2 = mat.fullPivLu().solve(B);
14 MatrixXd X3 = mat.householderQr().solve(B);
15 MatrixXd X4 = mat.llt().solve(B);
16 MatrixXd X5 = mat.ldlt().solve(B);
17 cout << " |X2-X| = " << (X2-X).norm() << endl;
18 cout << " |X3-X| = " << (X3-X).norm() << endl;
19 cout << " |X4-X| = " << (X4-X).norm() << endl;
20 cout << " |X5-X| = " << (X5-X).norm() << endl;

```

The [different decompositions](#) trade speed for stability and accuracy: fully pivoted and QR-based decompositions also work for nearly singular matrices, for which the standard LU-factorization may no longer be reliable.

### Remark 1.6.87 (Many sequential solutions of LSE)

Both eigen and matlab provide functions that return decompositions of matrices, here the LU-decomposition ( $\rightarrow$  Section 1.6.2.2):

```
EIGEN : MatrixXd A(n,n); auto ludec = mat.lu();
MATLAB: [L,U] = lu(A)
```

Based on the *precomputed decompositions*, a linear system of equations with coefficient matrix  $\mathbf{A} \in \mathbb{K}^{n,n}$  can be solved with asymptotic computational effort  $O(n^2)$ , cf. § 1.6.42.

The following example illustrates a special situation, in which matrix decompositions can curb computa-

tional cost:

#### MATLAB-code 1.6.88: Wasteful approach!

```

1 % Setting: N >> 1,
2 % large matrix A ∈ ℝn,n
3 for j=1:N
4     x = A\b;
5     b = some_function(x);
6 end

```

computational effort  $O(Nn^3)$

#### MATLAB-code 1.6.89: Smart approach!

```

1 % Setting: N >> 1,
2 % large matrix A ∈ ℝn,n
3 [L,U] = lu(A);
4 for j=1:N
5     x = U\ (L\b);
6     b = some_function(x);
7 end

```

computational effort  $O(n^3 + Nn^2)$

A concrete example is the so-called **inverse power iteration**, see Chapter 7, for which a skeleton code is given next. It computes the iterates

$$\mathbf{x}^* := \mathbf{A}^{-1}\mathbf{x}^{(k)}, \quad \mathbf{x}^{(k+1)} := \frac{\mathbf{x}^*}{\|\mathbf{x}^*\|_2}, \quad k = 0, 1, 2, \dots, \quad (1.6.90)$$

#### C++11-code 1.6.91: Efficient implementation of inverse power method in EIGEN

```

1 template<class VecType, class MatType>
2 VecType invpowit(const Eigen::MatrixBase<MatType> &A, double tol)
3 {
4     using index_t = typename MatType::Index;
5     using scalar_t = typename VecType::Scalar;
6     // Make sure that the function is called with a square matrix
7     const index_t n = A.cols();
8     const index_t m = A.rows();
9     eigen_assert(n == m);
10    // Request LU-decomposition
11    auto A_lu_dec = A.lu();
12    // Initial guess for inverse power iteration
13    VecType xo = VecType::Zero(n);
14    VecType xn = VecType::Random(n);
15    // Normalize vector
16    xn /= xn.norm();
17    // Terminate if relative (normwise) change below threshold
18    while ((xo-xn).norm() > xn.norm()*tol) {
19        xo = xn;
20        xn = A_lu_dec.solve(xo);
21        xn /= xn.norm();
22    }
23    return (xn);
24 }

```

The use of `Eigen::MatrixBase<MatType>` makes it possible to call `invpowit` with an expression argument

```
MatrixXd A(n,n), B(n,n);
```

```
.....
VectorXd ev = invpowit<VectorXd>(A+B,tol);
```

This is necessary, because  $A+B$  will spawn an auxiliary object of a “strange” type determined by the **expression template** mechanism.

---

## 1.6.5 Exploiting Structure when Solving Linear Systems

By “structure” of a linear system we mean prior knowledge that

- \* either certain entries of the system matrix vanish,
- \* or the system matrix is generated by a particular formula.

### (1.6.92) Triangular linear systems

Triangular linear systems are linear systems of equations whose system matrix is a *triangular matrix* ( $\rightarrow$  Def. 1.1.5).

Thm. 1.6.80 tells us that (dense) triangular linear systems can be solved by backward/forward elimination with  $O(n^2)$  asymptotic computational effort ( $n \doteq$  number of unknowns) compared to an asymptotic complexity of  $O(n^3)$  for solving a generic (dense) linear system of equations ( $\rightarrow$  Thm. 1.6.79).

This is the simplest case where exploiting special structure of the system matrix leads to faster algorithms for the solution of a special class of linear systems.

---

### (1.6.93) Block elimination

Remember that thanks to the possibility to compute the matrix product in a block-wise fashion ( $\rightarrow$  § 1.3.15), Gaussian elimination can be conducted on the level of matrix blocks. We recall Rem. 1.6.32 and Rem. 1.6.46.

For  $k, \ell \in \mathbb{N}$  consider the block partitioned square  $n \times n$ ,  $n := k + \ell$ , linear system

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix}, \quad \mathbf{A}_{11} \in \mathbb{K}^{k,k}, \mathbf{A}_{12} \in \mathbb{K}^{k,\ell}, \mathbf{A}_{21} \in \mathbb{K}^{\ell,k}, \mathbf{A}_{22} \in \mathbb{K}^{\ell,\ell}, \quad (1.6.94)$$

Using block matrix multiplication (applied to the matrix  $\times$  vector product in (1.6.94)) we find an equivalent way to write the block partitioned linear system of equations:

$$\begin{aligned} \mathbf{A}_{11}\mathbf{x}_1 + \mathbf{A}_{12}\mathbf{x}_2 &= \mathbf{b}_1, \\ \mathbf{A}_{21}\mathbf{x}_1 + \mathbf{A}_{22}\mathbf{x}_2 &= \mathbf{b}_2. \end{aligned} \quad (1.6.95)$$

We assume that  $\mathbf{A}_{11}$  is *regular* (invertible) so that we can solve for  $\mathbf{x}_1$  from the first equation.

➤ quad By elementary algebraic manipulations (“block Gaussian elimination”) we find

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{A}_{11}^{-1}(\mathbf{b}_1 - \mathbf{A}_{12}\mathbf{x}_2), \\ \mathbf{x}_2 &= \underbrace{(\mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12})}_{\text{Schur complement}} \mathbf{x}_2 = \mathbf{b}_2 - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{b}_1, \end{aligned}$$

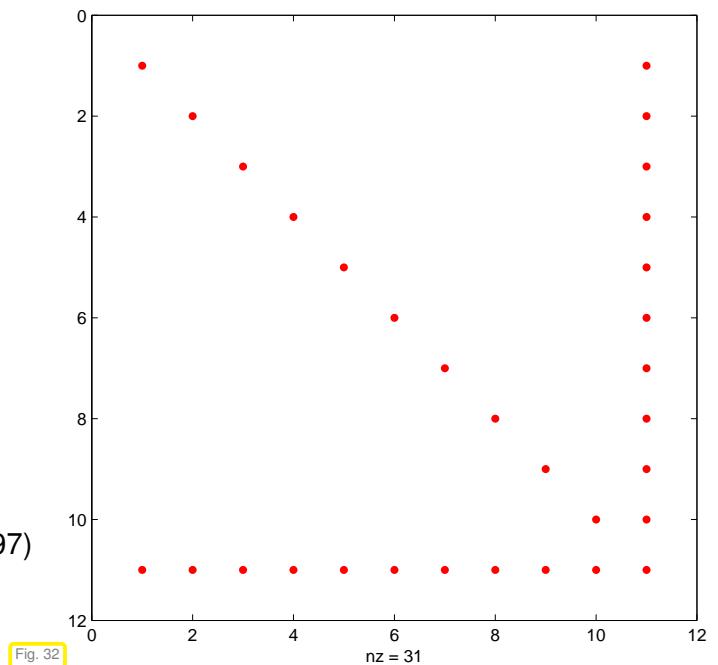
The resulting  $\ell \times \ell$  linear system of equations for the unknown vector  $\mathbf{x}_2$  is called the **Schur complement system** for (1.6.94).

Unless  $\mathbf{A}$  has a special structure that allows the efficient solution of linear systems with system matrix  $\mathbf{A}_{11}$ , the Schur complement system is mainly of theoretical interest.

### Example 1.6.96 (Linear systems with arrow matrices)

From  $n \in \mathbb{N}$ , a **diagonal** matrix  $\mathbf{D} \in \mathbb{K}^{n,n}$ ,  $\mathbf{c} \in \mathbb{K}^n$ ,  $\mathbf{b} \in \mathbb{K}^n$ , and  $\alpha \in \mathbb{K}$ , we can build an  $(n+1) \times (n+1)$  **arrow matrix**.

$$\mathbf{A} = \begin{bmatrix} \mathbf{D} & \mathbf{c} \\ \mathbf{b}^\top & \alpha \end{bmatrix} \quad (1.6.97)$$



We can apply the block partitioning (1.6.94) with  $k = n$  and  $\ell = 1$  to a linear system  $\mathbf{Ax} = \mathbf{y}$  with system matrix  $\mathbf{A}$  and obtain  $\mathbf{A}_{11} = \mathbf{D}$ , which can be inverted easily, provided that all diagonal entries of  $\mathbf{D}$  are different from zero. In this case

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{D} & \mathbf{c} \\ \mathbf{b}^\top & \alpha \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \xi \end{bmatrix} = \mathbf{y} := \begin{bmatrix} \mathbf{y}_1 \\ \eta \end{bmatrix}, \quad (1.6.98)$$

$$\begin{aligned} \Rightarrow \quad \xi &= \frac{\eta - \mathbf{b}^\top \mathbf{D}^{-1} \mathbf{y}_1}{\alpha - \mathbf{b}^\top \mathbf{D}^{-1} \mathbf{c}}, \\ \mathbf{x}_1 &= \mathbf{D}^{-1} (\mathbf{y}_1 - \xi \mathbf{c}). \end{aligned} \quad (1.6.99)$$

These formulas make sense, if  $\mathbf{D}$  is regular and  $\alpha - \mathbf{b}^\top \mathbf{D}^{-1} \mathbf{c} \neq 0$ , which is another condition for the invertibility of  $\mathbf{A}$ .

Using the formula (1.6.99) we can solve the linear system (1.6.98) with an asymptotic complexity  $O(n)$ ! This superior speed compared to Gaussian elimination applied to the (dense) linear system is evident in runtime measurements.

#### MATLAB-code 1.6.100: Dense Gaussian elimination applied to arrow system

```

1 function x = arrowsys_slow(d,c,b,alpha,y)
2 A = [diag(d),c;transpose(b),alpha];
3 x = A\y;

```

- Asymptotic complexity  $O(n^3)$ !  
 (Due to the serious blunder of accidentally creating a matrix full of zeros by using **diag**, cf. Exp. 1.3.10.)

#### MATLAB-code 1.6.101: Solving an arrow system according to (1.6.99)

```

1 function x = arrowsys_fast(d,c,b,alpha,y)
2 z = c./d; % z = D^-1c
3 w = y(1:end-1)./d; % w = D^-1y_1
4 xi = (y(end)-dot(b,w)) / (alpha - dot(b,z));
5 x = [w-xi*c./d;xi];

```

- Asymptotic complexity  $O(n)$

#### MATLAB-code 1.6.102: Runtime measurement of Code 1.6.100 vs. Code 1.6.101

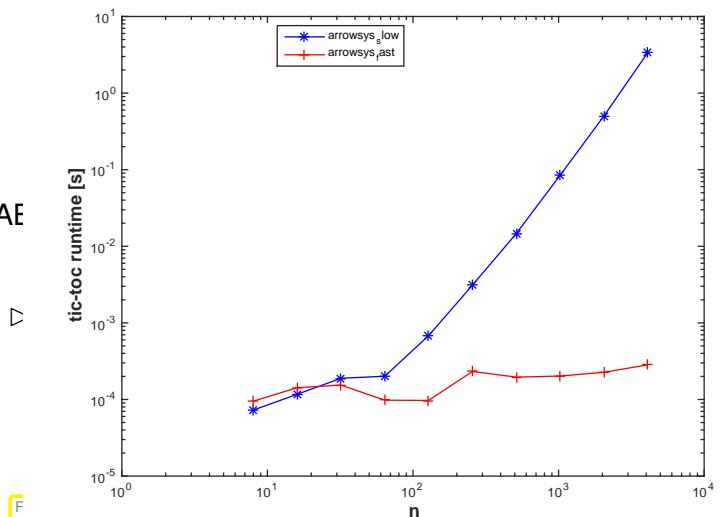
```

1 t = []; K = 3;
2 for l=3:12
3   n = 2^l;
4   alpha = 2; b = ones(n,1); c = (1:n)'; d = -ones(n,1);
5   y = (-1).^ (1:(n+1))';
6   t1 = realmax; t2 = realmax;
7   for k=1:K
8     tic; x1 = arrowsys_slow(d,c,b,alpha,y); t1 = min(t1,toc);
9     tic; x2 = arrowsys_fast(d,c,b,alpha,y); t2 = min(t2,toc);
10   end
11   t = [t; n t1 t2];
12 end
13 loglog(t(:,1),t(:,2),'b-*',t(:,1),t(:,3),'r-+');
14 xlabel('{\bf n}',fontsize,14);
15 ylabel('{\bf tic-toc runtime [s]}',fontsize,14);
16 legend('arrowsys_slow','arrowsys_fast','location','best');
17 print -depsc '../PICTURES/arrowsystiming.eps';

```

(Intel Core i7, 2.66 GHz, MacOS X 1.10, MATLAB 8.5.0 R2015a)

No comment!



### Remark 1.6.103 (Sacrificing numerical stability for efficiency)

The vector based implementation of the solver of Code 1.6.101 can be vulnerable to roundoff errors, because, upon closer inspection, the algorithm turns out to be equivalent to Gaussian elimination **without pivoting**, cf. Section 1.6.2.3, Ex. 1.6.48.



Caution:

stability at risk in Code 1.6.101

Yet, there are classes of matrices for which Gaussian elimination without pivoting is guaranteed to be stable. For such matrices algorithms like that of Code 1.6.101 are safe.

### (1.6.104) Solving LSE with low-rank perturbation of system matrix

Given a regular matrix  $\mathbf{A} \in \mathbb{K}^{n,n}$ , let us assume that at some point in a code we are in a position to solve any linear system  $\mathbf{Ax} = \mathbf{b}$  “fast”, because

- \* either  $\mathbf{A}$  has a favorable structure, eg. triangular, see § 1.6.92,
- \* or an LU-decomposition of  $\mathbf{A}$  is already available, see § 1.6.42.

Now, a  $\tilde{\mathbf{A}}$  is obtained by changing a *single entry* of  $\mathbf{A}$ :

$$\mathbf{A}, \tilde{\mathbf{A}} \in \mathbb{K}^{n,n}: \quad \tilde{a}_{ij} = \begin{cases} a_{ij} & , \text{ if } (i,j) \neq (i^*, j^*) , \\ z + a_{ij} & , \text{ if } (i,j) = (i^*, j^*) , \end{cases} \quad , \quad i^*, j^* \in \{1, \dots, n\} . \quad (1.6.105)$$

►  $\tilde{\mathbf{A}} = \mathbf{A} + z \cdot \mathbf{e}_{i^*} \mathbf{e}_{j^*}^T$  . (1.6.106)

(Recall:  $\mathbf{e}_i \doteq i$ -th unit vector.) The question is whether we can reuse some of the computations spent on solving  $\mathbf{Ax} = \mathbf{b}$  in order to solve  $\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \mathbf{b}$  with less effort than entailed by a direct Gaussian elimination

from scratch.

We may also consider a matrix modification affecting a single row: Changing a single row: given  $\mathbf{z} \in \mathbb{K}^n$

$$\mathbf{A}, \tilde{\mathbf{A}} \in \mathbb{K}^{n,n}: \tilde{a}_{ij} = \begin{cases} a_{ij} & , \text{ if } i \neq i^*, \\ (\mathbf{z})_j + a_{ij} & , \text{ if } i = i^*, \end{cases} , \quad i^*, j^* \in \{1, \dots, n\} .$$

►  $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{e}_{i^*} \mathbf{z}^T$  .

(1.6.107)

Both matrix modifications (1.6.105) and (1.6.107) represent **rank-1-modifications** of  $\mathbf{A}$ . A generic rank-1-modification reads

$$\mathbf{A} \in \mathbb{K}^{n,n} \mapsto \tilde{\mathbf{A}} := \mathbf{A} + \boxed{\mathbf{u}\mathbf{v}^H} , \quad \mathbf{u}, \mathbf{v} \in \mathbb{K}^n .$$

general rank-1-matrix



Idea: Block elimination of an extended linear system, see § 1.6.93

(1.6.108)

We consider the block partitioned linear system

$$\begin{bmatrix} \mathbf{A} & \mathbf{u} \\ \mathbf{v}^H & -1 \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{x}} \\ \xi \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} .$$
(1.6.109)

The **Schur complement** system after elimination of  $\xi$  reads

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^H)\tilde{\mathbf{x}} = \mathbf{b} \Leftrightarrow \tilde{\mathbf{A}}\tilde{\mathbf{x}} = \mathbf{b} . !$$
(1.6.110)

Hence, we have solved the modified LSE, once we have found the component  $\tilde{\mathbf{x}}$  of the solution of the extended linear system (1.6.109). We do block elimination again, now getting rid of  $\tilde{\mathbf{x}}$  first, which yields the other **Schur complement** system

$$(1 + \mathbf{v}^H \mathbf{A}^{-1} \mathbf{u})\xi = \mathbf{v}^H \mathbf{A}^{-1} \mathbf{b} .$$
(1.6.111)

►  $\mathbf{A}\tilde{\mathbf{x}} = \mathbf{b} - \frac{\mathbf{u}\mathbf{v}^H \mathbf{A}^{-1}}{1 + \mathbf{v}^H \mathbf{A}^{-1} \mathbf{u}} \mathbf{b} .$

(1.6.112)

The generalization of this formula to rank- $k$ -perturbations is given in the following lemma:

**Lemma 1.6.113. Sherman-Morrison-Woodbury formula**

For regular  $\mathbf{A} \in \mathbb{K}^{n,n}$ , and  $\mathbf{U}, \mathbf{V} \in \mathbb{K}^{n,k}$ ,  $n, k \in \mathbb{N}$ ,  $k \leq n$ , holds

$$(\mathbf{A} + \mathbf{U}\mathbf{V}^H)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I} + \mathbf{V}^H \mathbf{A}^{-1} \mathbf{U})^{-1} \mathbf{V}^H \mathbf{A}^{-1} ,$$

if  $\mathbf{I} + \mathbf{V}^H \mathbf{A}^{-1} \mathbf{U}$  is regular.

*Proof.* Straightforward algebra:

$$\begin{aligned} & \left( \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I} + \mathbf{V}^H \mathbf{A}^{-1} \mathbf{U})^{-1} \mathbf{V}^H \mathbf{A}^{-1} \right) (\mathbf{A} + \mathbf{U}\mathbf{V}^H) = \\ & \quad \mathbf{I} - \mathbf{A}^{-1}\mathbf{U} \underbrace{(\mathbf{I} + \mathbf{V}^H \mathbf{A}^{-1} \mathbf{U})^{-1}(\mathbf{I} + \mathbf{V}^H \mathbf{A}^{-1} \mathbf{U})}_{=\mathbf{I}} \mathbf{V}^H + \mathbf{A}^{-1}\mathbf{U}\mathbf{V}^H = \mathbf{I} . \end{aligned}$$

Uniqueness of the inverse settles the case. □

We use this result to solve  $\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \mathbf{b}$  with  $\tilde{\mathbf{A}}$  from (1.6.108) more efficiently than straightforward elimination could deliver, provided that the LU-factorisation  $\mathbf{a} = \mathbf{LU}$  is already known.

Apply Lemma 1.6.113 for  $k = 1$ :

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} - \frac{\mathbf{A}^{-1}\mathbf{u}(\mathbf{v}^H(\mathbf{A}^{-1}\mathbf{b}))}{1 + \mathbf{v}^H(\mathbf{A}^{-1}\mathbf{u})}.$$

LU-decomposition of  $\mathbf{A}$  is available.

Efficient implementation !

Asymptotic complexity  $O(n^2)$   
(back substitutions)

#### MATLAB-code 1.6.114: solving a rank-1 modified LSE

```

1 function x = smw(L,U,u,v,b)
2 z = U \ (L \ b); w = U \ (L \ u);
3 alpha = 1+dot(v,w);
4 if (abs(alpha) <
    eps*norm(U,1)),
    error ('Nearly singular
matrix'); end;
5 x = z - w*dot(v,z)/alpha;
```

#### Example 1.6.115 (Resistance to currents map)

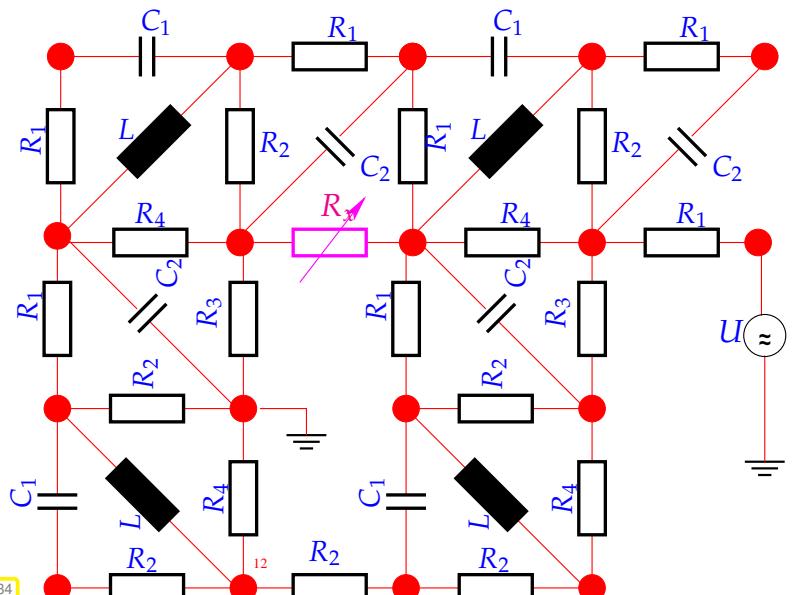
Many lineare systems with system matrices that differ in a single entry only have to be solved when we want to determine the dependence of the total impedance of a (linear) circuit from the parameters of a single component.

Large (linear) electric circuit (modelling → Ex. 1.6.3) ▷

Sought:

Dependence of (certain) branch currents on “continuously varying” resistance  $R_x$

(> currents for many different values of  $R_x$ )



Only a few entries of the nodal analysis matrix  $\mathbf{A}$  (→ Ex. 1.6.3) are affected by variation of  $R_x$ !  
(If  $R_x$  connects nodes  $i$  &  $j$  ⇒ only entries  $a_{ii}, a_{jj}, a_{ij}, a_{ji}$  of  $\mathbf{A}$  depend on  $R_x$ )

## 1.7 Sparse Linear Systems

A (rather fuzzy) classification of matrices according to their numbers of zeros:

Dense(ly populated) matrices



sparse(ly populated) matrices

### Notion 1.7.1. Sparse matrix

$\mathbf{A} \in \mathbb{K}^{m,n}$ ,  $m, n \in \mathbb{N}$ , is **sparse**, if

$$\text{nnz}(\mathbf{A}) := \#\{(i,j) \in \{1, \dots, m\} \times \{1, \dots, n\} : a_{ij} \neq 0\} \ll mn .$$

Sloppy parlance: matrix **sparse**  $\Leftrightarrow$  “almost all” entries  $= 0$  /“only a few percent of” entries  $\neq 0$

J.H. Wilkinson’s informal working definition for a developer of simulation codes:

### Notion 1.7.2. Sparse matrix

A matrix with enough zeros that it pays to take advantage of them should be *treated as sparse*.

A more rigorous “mathematical” definition:

### Definition 1.7.3. Sparse matrices

Given a strictly increasing sequences  $m : \mathbb{N} \mapsto \mathbb{N}$ ,  $n : \mathbb{N} \mapsto \mathbb{N}$ , a family  $(\mathbf{A}^{(l)})_{l \in \mathbb{N}}$  of matrices with  $\mathbf{A}^{(l)} \in \mathbb{K}^{m_l, n_l}$  is **sparse** (opposite: dense), if

$$\lim_{l \rightarrow \infty} \frac{\text{nnz}(\mathbf{A}^{(l)})}{n_l m_l} = 0 .$$

Simple example: families of diagonal matrices ( $\rightarrow$  Def. 1.1.5)

### Example 1.7.4 (Sparse LSE in circuit modelling)

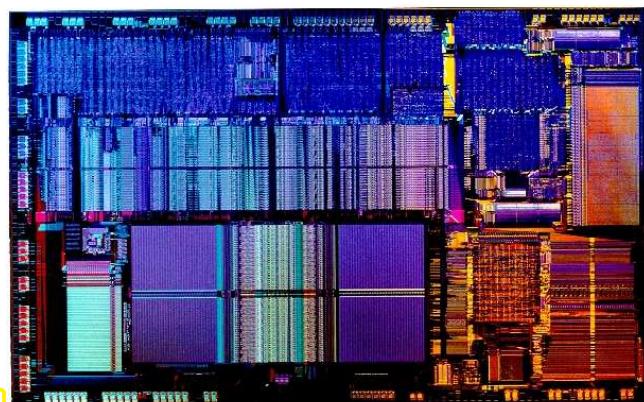
See Ex. 1.6.3 for the description of a linear electric circuit by means of a linear system of equations for nodal voltages. For large circuits the system matrices will invariably be huge and sparse.

Modern electric circuits (VLSI chips):

$10^5 - 10^7$  circuit elements

- Each element is connected to only a few nodes
- Each node is connected to only a few elements  
[In the case of a linear circuit]

nodal analysis  $\rightarrow$  **sparse** circuit matrix  
(Impossible to even store as dense matrices)



### Remark 1.7.5 (Sparse matrices from the discretization of linear partial differential equations)

Another important context in which sparse matrices usually arise:

- spatial discretization of linear boundary value problems for partial differential equations by means of finite element (FE), finite volume (FV), or finite difference (FD) methods ( $\rightarrow$  4th semester course “Numerical methods for PDEs”).

### 1.7.1 Sparse matrix storage formats

Sparse matrix storage formats for storing a “sparse matrix”  $\mathbf{A} \in \mathbb{K}^{m,n}$  are designed to achieve two objectives:

- ❶ Amount of memory required is only slightly more than  $\text{nnz}(\mathbf{A})$  scalars.
- ❷ Computational effort for matrix  $\times$  vector multiplication is proportional to  $\text{nnz}(\mathbf{A})$ .

In this section we see a few schemes used by numerical libraries.

#### (1.7.6) Triplet/coordinate list (COO) format

In the case of a sparse matrix  $\mathbf{A} \in \mathbb{K}^{m,n}$ , this format stores triplets  $(i, j, \alpha_{i,j})$ ,  $1 \leq i \leq m$ ,  $1 \leq j \leq n$ :

```
struct TripletMatrix {
    size_t m,n;           // Number of rows and columns
    vector<size_t> I;    // row indices
    vector<size_t> J;    // column indices
    vector<scalar_t> a; // values associated with index pairs
};
```

All vectors have the same size  $\geq \text{nnz}(\mathbf{a})$ .

We write “ $\geq$ ”, because *repetitions* of index pairs  $(i, j)$  are *allowed*. The matrix entry  $(\mathbf{A})_{i,j}$  is defined to be the *sum* of all values  $\alpha_{i,j}$  associated with the index pair  $(i, j)$ . The next code clear demonstrates this summation.

#### C++-code 1.7.7: Matrix $\times$ vector product $\mathbf{y} = \mathbf{A}\mathbf{x}$ in triplet format

```
1 void multTripIMatvec(const TripletMatrix &A,
2                         const vector<scalar_t> &x,
3                         vector<scalar_t> &y)
4     for (size_t l=0; l<A.a.size(); l++) {
5         y[A.I[l]] += A.a[l]*x[A.J[l]];
6     }
```

Note that this code assumes that the result vector  $y$  has the appropriate length; no index checks are performed.

Code 1.7.7: computational effort is proportional to the number of triplets. (This might be much larger than  $\text{nnz}(\mathbf{A})$  in case of many repetitions of triplets.)

#### (1.7.8) The zoo of sparse matrix formats

Special **sparse matrix storage formats** store *only* non-zero entries:

- Compressed Row Storage (CRS)
- Compressed Column Storage (CCS) → used by MATLAB

- Block Compressed Row Storage (BCRS)
- Compressed Diagonal Storage (CDS)
- Jagged Diagonal Storage (JDS)
- Skyline Storage (SKS)

All of these formats achieve the two objectives stated above. Some have been designed for sparse matrices with additional structure or for seamless cooperation with direct elimination algorithms (JDS,SKS).

### Example 1.7.9 (Compressed row-storage (CRS) format)

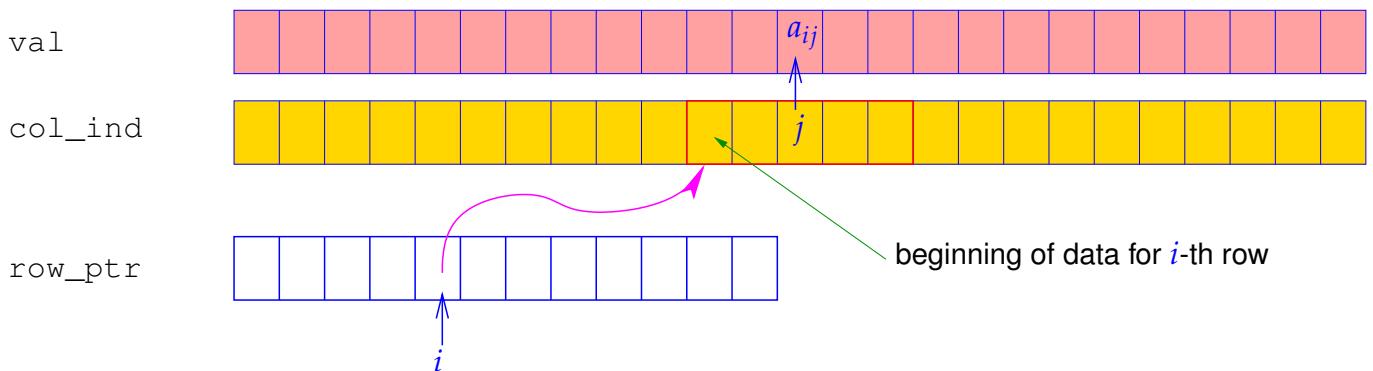
The CRS format for a sparse matrix  $\mathbf{A} = (a_{ij}) \in \mathbb{K}^{n,n}$  keeps the data in three arrays:

vector<scalar_t> val	$\text{size } \text{nnz}(\mathbf{A}) := \#\{(i,j) \in \{1, \dots, n\}^2, a_{ij} \neq 0\}$
vector<size_t> col_ind	$\text{size } \text{nnz}(\mathbf{A})$
vector<size_t> row_ptr	$\text{size } n+1 \text{ & } \text{row\_ptr}[n+1] = \text{nnz}(\mathbf{A}) + 1$ (sentinel value)

As above we write  $\text{nnz}(\mathbf{A}) \doteq (\text{number of nonzeros})$  of  $\mathbf{A}$

Access to matrix entry  $a_{ij} \neq 0, 1 \leq i, j \leq n$ :

$$\text{val}[k] = a_{ij} \Leftrightarrow \begin{cases} \text{col\_ind}[k] = j, \\ \text{row\_ptr}[i] \leq k < \text{row\_ptr}[i+1], \end{cases} \quad 1 \leq k \leq \text{nnz}(\mathbf{A}).$$



$$\mathbf{A} = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}$$

val-vector:	[10 -2 3 9 3 7 8 7 3 ... 9 13 4 2 -1]
col_ind-array:	[1 5 1 2 6 2 3 4 1 ... 5 6 2 5 6]
row_ptr-array:	[1 3 6 9 13 17 20]

Variant: diagonal CRS format (matrix diagonal stored in separate array)

The CCS format is equivalent to CRS format for the transposed matrix.

## 1.7.2 Sparse matrices in MATLAB



*Supplementary reading.* A detailed discussion of sparse matrix formats and how to work with them efficiently is given in [25]. An interesting related article on MATLAB-central can be found [here](#).

### (1.7.10) Creating sparse matrices in MATLAB

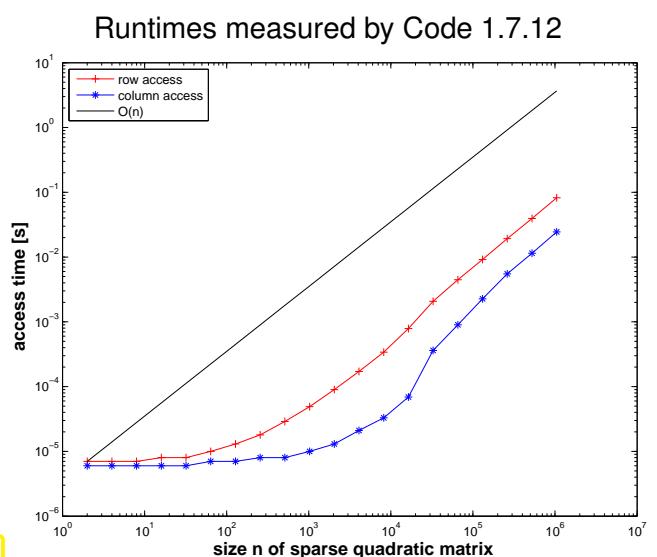
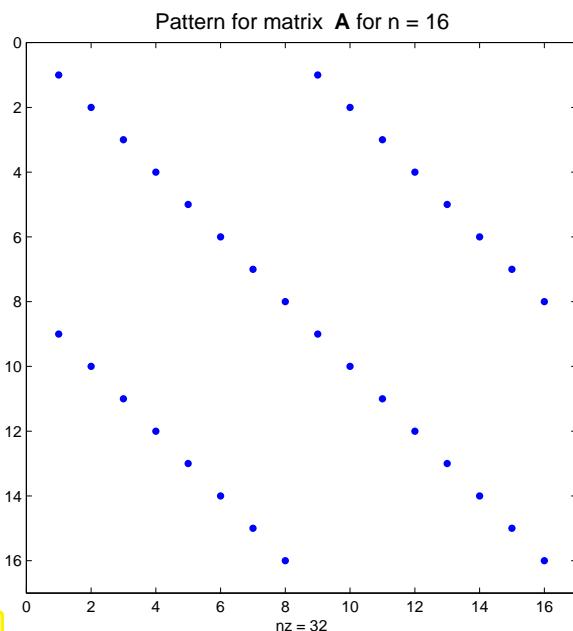
Matrices have to be created explicitly in sparse format by one of the following commands in order to let MATLAB know that the CCS format is to be used for internal representation.

<code>A = sparse(m, n);</code>	create empty $m \times n$ “sparse matrix”
<code>A = spalloc(m, n, nnz);</code>	create $m \times n$ sparse matrix & reserve memory
<code>A = sparse(I, J, a, m, n);</code>	initialize $m \times n$ sparse matrix from triplets → § 1.7.6
<code>A = spdiags(B, d, m, n);</code>	create sparse banded matrix → Section 1.7.6
<code>A = speye(n);</code>	sparse identity matrix

☞ Consult the MATLAB documentation for details.

### Experiment 1.7.11 (Accessing rows and columns of sparse matrices)

The CCS internal data format used by MATLAB has an impact on the speed of access operations, cf. Exp. 1.2.25 for a similar effect.



### MATLAB-code 1.7.12: timing access to rows/columns of a sparse matrix

```

1 figure; spy(spdiags(repmat([-1 2 5],16,1),[-8,0,8],16,16)); %
2 title ('Pattern for matrix {\bf A} for n = 16','fontsize',14);
3 print -depsc2 '../PICTURES/spdiagsmatspy.eps';

```

```

4 t = [];
5 for i=1:20
6 n = 2^i; m = n/2;
7 A = spdiags(repmat([-1 2 5],n,1), [-n/2, 0, n/2], n, n); %
8
9
10 t1 = inf; for j=1:5, tic; v = A(m,:)+j; t1 = min(t1,toc); end
11 t2 = inf; for j=1:5, tic; v = A(:,m)+j; t2 = min(t2,toc); end
12 t = [t; size(A,1), nnz(A), t1, t2 ];
13 end
14
15 figure;
16 loglog(t(:,1),t(:,3),'r+-', t(:,1),t(:,4),'b*-',...
17 t(:,1),t(1,3)*t(:,1)/t(1,1),'k-');
18 xlabel('{\bf size n of sparse quadratic matrix}', 'fontsize',14);
19 ylabel('{\bf access time [s]}', 'fontsize',14);
20 legend('row access', 'column access', 'O(n)', 'location', 'northwest');
21
22 print -depsc2 '../PICTURES/sparseaccess.eps';

```

MATLAB uses compressed column storage (CCS), which entails  $O(n)$  searches for index  $j$  in the index array when accessing all elements of a matrix row. Conversely, access to a column does not involve any search operations.

Note the use of the MATLAB command `repmat` in Line 1 and Line 8 of the above code. It can be used to build structured matrices. Consult the MATLAB documentation for details.

### Experiment 1.7.13 (Efficient Initialization of sparse matrices in MATLAB)

We study different ways to set a few non-zero entries of a sparse matrix. The first code just uses the `( )`-operator to set matrix entries.

#### MATLAB-code 1.7.14: Initialization of sparse matrices: entry-wise (I)

```

1 A1 = sparse(n,n);
2 for i=1:n
3     for j=1:n
4         if (abs(i-j) == 1), A1(i,j) = A1(i,j) + 1; end;
5         if (abs(i-j) == round(n/3)), A1(i,j) = A1(i,j) -1; end;
6     end; end

```

The second and third code rely on an [intermediate triplet format](#) ( $\rightarrow$  § 1.7.6) to build the sparse matrix and finally pass this to MATLAB's `sparse` function.

#### MATLAB-code 1.7.15: Initialization of sparse matrices: triplet based (II)

```

1 dat = [];
2 for i=1:n
3     for j=1:n

```

```

4      if (abs(i-j) == 1), dat = [dat; i,j,1.0]; end;
5      if (abs(i-j) == round(n/3)), dat = [dat; i,j,-1.0];
6  end; end; end;
7 A2 = sparse(dat(:,1),dat(:,2),dat(:,3),n,n);

```

**MATLAB-code 1.7.16: Initialization of sparse matrices: triplet based (III)**

```

1 dat = zeros(6*n,3); k = 0;
2 for i=1:n
3     for j=1:n
4         if (abs(i-j) == 1), k=k+1; dat(k,:) = [i,j,1.0];
5             end;
6         if (abs(i-j) == round(n/3))
7             k=k+1; dat(k,:) = [i,j,-1.0];
8             end;
9     end; end;
10    A3 = sparse(dat(1:k,1),dat(1:k,2),dat(1:k,3),n,n);

```

**MATLAB-code 1.7.17: Initialization of sparse matrices: driver script**

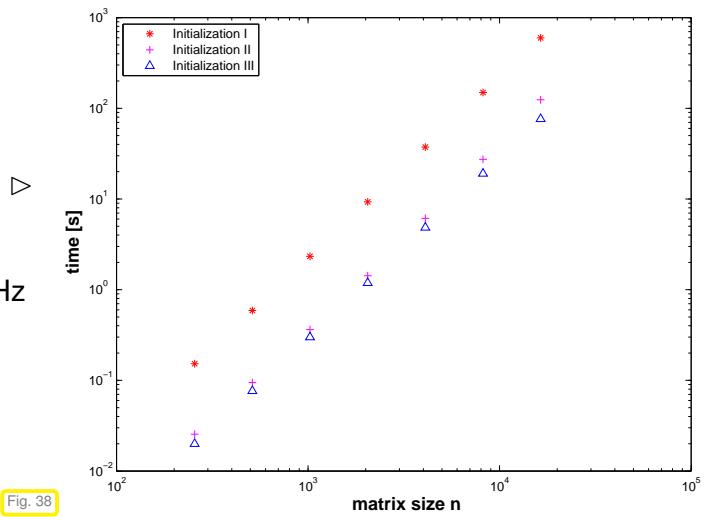
```

1 % Driver routine for initialization of sparse matrices
2 K = 3; r = [];
3 for n=2.^8:14
4     t1= 1000; for k=1:K, fprintf('sparse1, %d, %d\n',n,k); tic;
5         sparse1; t1 = min(t1,toc); end
6     t2= 1000; for k=1:K, fprintf('sparse2, %d, %d\n',n,k); tic;
7         sparse2; t2 = min(t2,toc); end
8     t3= 1000; for k=1:K, fprintf('sparse3, %d, %d\n',n,k); tic;
9         sparse3; t3 = min(t3,toc); end
10    r = [r; n, t1 , t2, t3];
11 end
12
13 loglog(r(:,1),r(:,2),'r*',r(:,1),r(:,3),'m+',r(:,1),r(:,4),'b^');
14 xlabel('{\bf matrix size n}', 'fontsize',14);
15 ylabel('{\bf time [s]}', 'fontsize',14);
16 legend('Initialization I','Initialization II','Initialization III',...
17 'location','northwest');
18 print -depsc2 '../PICTURES/sparseinit.eps';

```

Output of Code 1.7.17:

- OS: Linux 2.6.16.27-0.9-smp
- CPU: Genuine Intel(R) CPU T2500 2.00GHz
- MATLAB 7.4.0.336 (R2007a)



- It is grossly inefficient to initialize a matrix in CCS format ( $\rightarrow$  Ex. 1.7.9) by setting individual entries one after another, because this usually entails moving large chunks of memory to create space for new non-zero entries.

Instead calls like

```
sparse(dat(1:k,1), dat(1:k,2), dat(1:k,3), n, n);
```

where the triplet format is defined by ( $\rightarrow$  § 1.7.6)

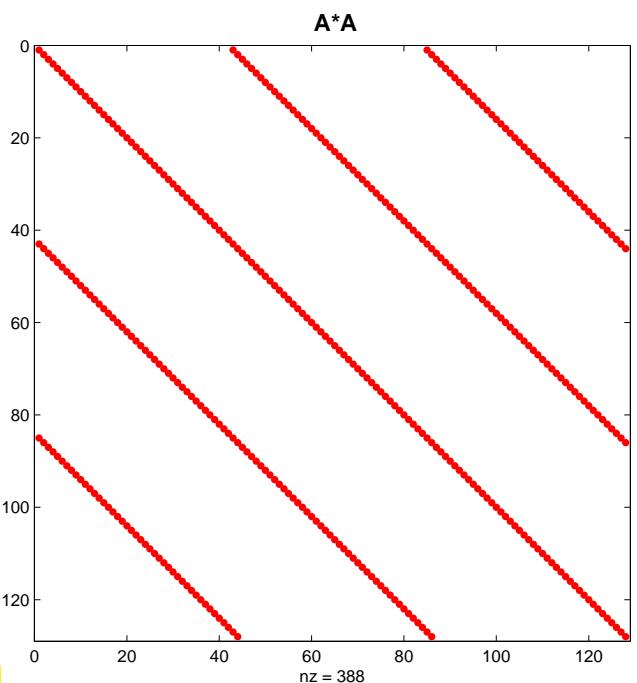
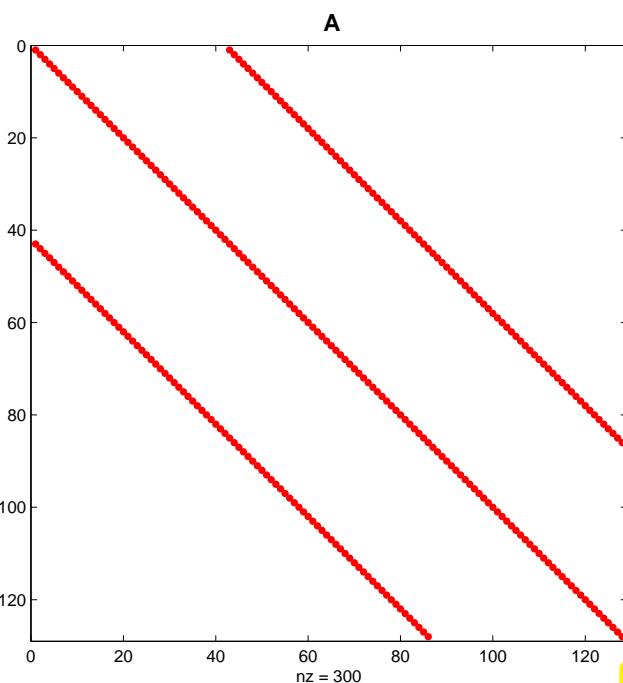
$$\text{dat}(1:k,1) = i \quad \text{and} \quad \text{dat}(1:k,2) = j \quad \Rightarrow \quad a_{ij} = \text{dat}(1:k,3),$$

allow MATLAB to allocate memory and initialize the arrays in one sweep.

### Experiment 1.7.18 (Multiplication of sparse matrices)

We study a sparse matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$  initialized by setting some of its (off-)diagonals with MATLAB's **spdiags** function:

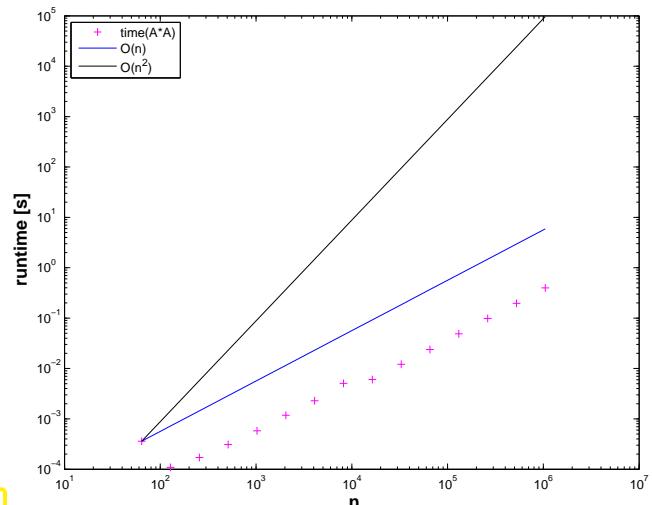
```
A = spdiags([(1:n)', ones(n,1), (n:-1:1)'],...  
[-floor(n/3), 0, floor(n/3)], n, n);
```



- $\mathbf{A}^2$  is still a sparse matrix ( $\rightarrow$  Notion 1.7.1)

Runtimes for matrix multiplication  $\mathbf{A} \star \mathbf{A}$  in MATLAB  
(`tic/toc` timing)

(same platform as in Ex. 1.7.13)



- We observe  $O(n)$  asymptotic complexity: “optimal” implementation !

### Remark 1.7.19 (Silly MATLAB)

A strange behavior of MATLAB from version R2010a:

#### MATLAB-code 1.7.20: Extracting an entry of a sparse matrix

```

1 % MATLAB script demonstrating the awkward effects of treating entries
2 % of
3 % sparse matrices as sparse matrices themselves.
4 A = spdiags(ones(100,3), -1:1, 100, 100);
5 b = A(1,1),
6 c = full(b),
7 whos('b','c');
8 sum=0; tic; for i=1:1e6, sum = sum + b; end, toc
9 sum=0; tic; for i=1:1e6, sum = sum + c; end, toc

```

Output (MATLAB-version 7.12.0.635 (R2011a), MacOS X 10.6, Intel Core i7):

```

>> sparseentry
b = (1,1) 1
c = 1

```

Name	Size	Bytes	Class	Attributes
b	1x1	32	double	sparse
c	1x1	8	double	

```

Elapsed time is 2.962332 seconds.
Elapsed time is 0.514712 seconds.

```

When extracting a single entry from a sparse matrix, this entry will be stored in sparse format though it is a mere number! This will considerably slow down all operations on that entry.

**Change in Indexing for Sparse Matrix Access.** Now subscripted reference into a sparse matrix always returns a sparse matrix. In previous versions of MATLAB, using a double scalar to index into a sparse matrix resulted in full scalar output.

### Example 1.7.21 (Smoothing of a triangulation)

This example demonstrates that sparse linear systems of equations naturally arise in the handling of triangulations.

#### Definition 1.7.22. Planar triangulation

A planar triangulation (mesh)  $\mathcal{M}$  consists of a set  $\mathcal{N}$  of  $N \in \mathbb{N}$  distinct points  $\in \mathbb{R}^2$  and a set  $\mathcal{T}$  of triangles with vertices in  $\mathcal{N}$ , such that the following two conditions are satisfied:

1. the interiors of the triangles are mutually disjoint (“no overlap”),
2. for every two *closed* distinct triangles  $\in \mathcal{T}$  their intersection satisfies exactly one of the following conditions:
  - (a) it is empty
  - (b) it is exactly one vertex from  $\mathcal{N}$ ,
  - (c) it is a **common edge** of both triangles

The points in  $\mathcal{N}$  are also called the **nodes** of the mesh, the triangles the **cells**, and all line segments connecting two nodes and occurring as a side of a triangle form the set of **edges**. We always assume a consecutive numbering of the nodes and cells of the triangulation (starting from 1, MATLAB’s convention).

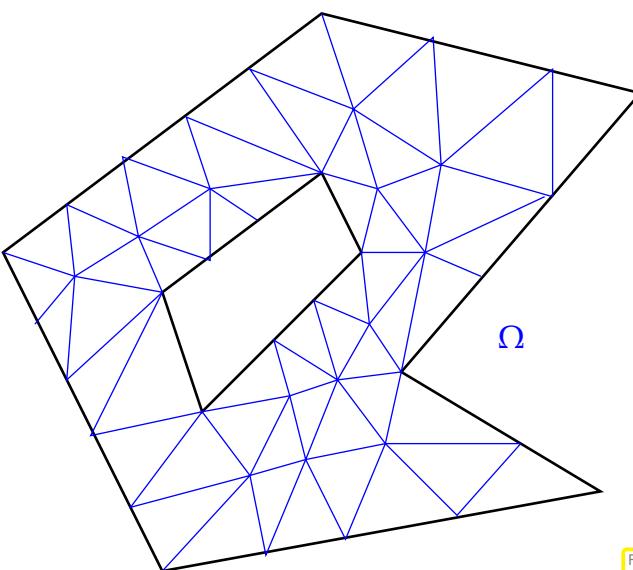


Fig. 42

Valid planar triangulation

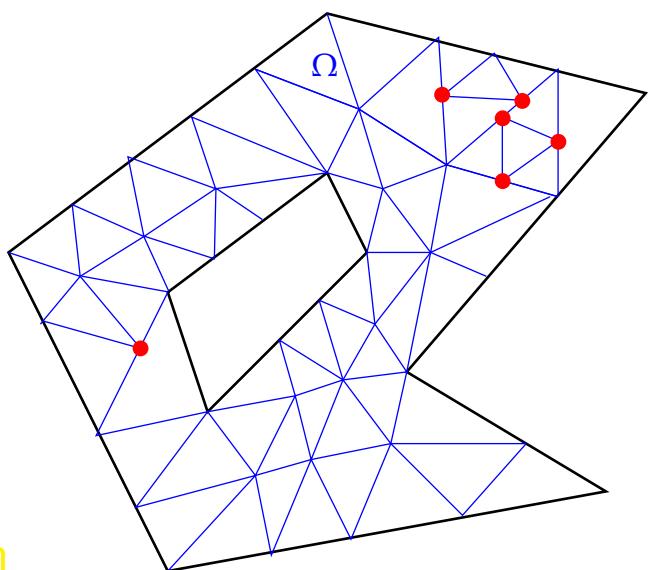
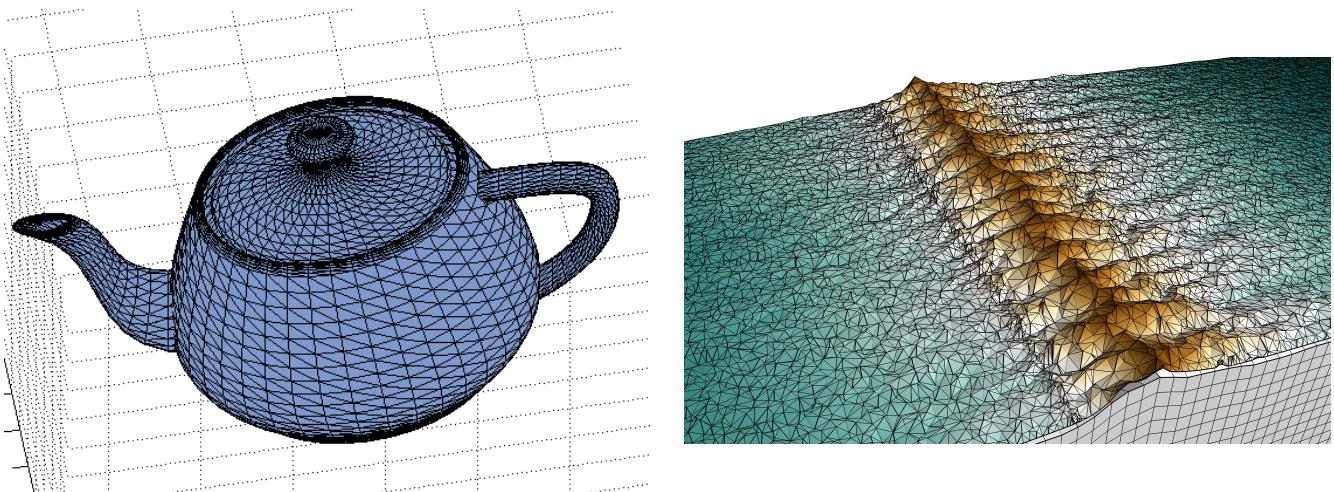


Fig. 43

Mesh with “illegal” **hanging nodes**

Triangulations are of fundamental importance for computer graphics, landscape models, geodesy, and numerical methods. They need not be planar, but the algorithmic issues remain the same.



MATLAB data structure for describing a triangulation with  $N$  nodes and  $M$  cells:

- column vector  $\mathbf{x} \in \mathbb{R}^N$ :  $x$ -coordinates of nodes
- column vector  $\mathbf{y} \in \mathbb{R}^N$ :  $y$ -coordinates of nodes
- $M \times 3$ -matrix  $\mathbf{T}$  whose rows contain the index numbers of the vertices of the cells.  
(This matrix is a so-called triangle-node **incidence matrix**.)

MATLAB provides the function `triplot` for drawing planar triangulations:

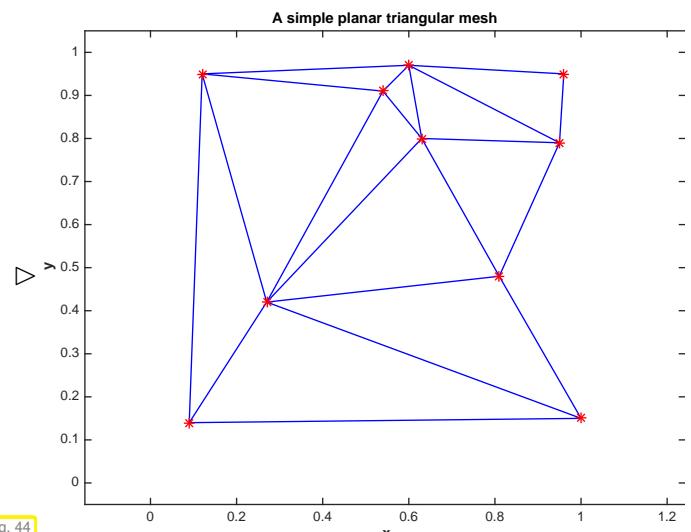
#### MATLAB-code 1.7.23: Initializing and drawing a simple planar triangulation

```

1 % MATLAB demonstration for visualizing a planes triangular mesh
2 % Initialize node coordinates
3 % First the x-coordinates
4 x = [1.0;0.60;0.12;0.81;0.63;0.09;0.27;0.54;0.95;0.96];
5 % Next the y-coordinates
6 y = [0.15;0.97;0.95;0.48;0.80;0.14;0.42;0.91;0.79;0.95];
7 % Then specify triangles through the indices of their vertices. These
8 % indices refer to the ordering of the coordinates as given in the
9 % vectors x and y.
10 T = [8 2 3;6 7 3;5 2 8;7 8 3;7 5 8;7 6 1;...
11     4 7 1;9 5 4;4 5 7;9 2 5;10 2 9];
12 % Call the plotting routine; draw mesh with blue edges
13 triplot(T,x,y,'b-'); title('A simple planar triangular mesh');
14 xlabel('{\bf x}'); ylabel('{\bf y}');
15 axis([-0.05 1.05 -0.05 1.05]); axis equal;
16 % Mark nodes with red stars
17 hold on; plot(x,y,'r*');
18
19 % Save plot a vector graphics
20 print -depsc2 'meshplot.eps';

```

Output of Code 1.7.23



The cells of a mesh may be rather distorted triangles (with very large and/or small angles), which is usually not desirable. We study an algorithm for **smoothing** a mesh without changing the planar domain covered by it.

#### Definition 1.7.24. Boundary edge

Every edge that is adjacent to only one cell is a **boundary edge** of the triangulation. Nodes that are endpoints of boundary edges are **boundary nodes**.

- ☞ Notation:  $\Gamma \subset \{1, \dots, N\} \hat{=} \text{set of indices of boundary nodes.}$
- ☞ Notation:  $\mathbf{p}^i = (p_1^i, p_2^i) \in \mathbb{R}^2 \hat{=} \text{coordinate vector of node } \#i, i = 1, \dots, N$   
 $(p_1^i \leftrightarrow x(i), p_2^i \leftrightarrow y(i) \text{ in MATLAB})$

We define

$$S(i) := \{j \in \{1, \dots, N\} : \text{nodes } i \text{ and } j \text{ are connected by an edge}\}, \quad (1.7.25)$$

as the set of node indices of the “neighbours” of the node with index number  $i$ .

#### Definition 1.7.26. Smoothed triangulation

A triangulation is called **smoothed**, if

$$\mathbf{p}^i = \frac{1}{\#S(i)} \sum_{j \in S(i)} \mathbf{p}^j \Leftrightarrow \#S(i)p_d^i = \sum_{j \in S(i)} p_d^j, d = 1, 2, \text{ for all } i \in \{1, \dots, N\} \setminus \Gamma, \quad (1.7.27)$$

that is, every interior node is located in the center of gravity of its neighbours.

The relations (1.7.27) correspond to the lines of a sparse linear system of equations! In order to state it, we insert the coordinates of all nodes into a column vector  $\mathbf{z} \in \mathbb{K}^{2N}$ , according to

$$z_i = \begin{cases} p_1^i & , \text{if } 1 \leq i \leq N, \\ p_2^i & , \text{if } N+1 \leq i \leq 2N. \end{cases} \quad (1.7.28)$$

For the sake of ease of presentation, in the sequel we *assume* (which is not the case in usual triangulation data) that interior nodes have index numbers smaller than that of boundary nodes.

From (1.7.25) we infer that the system matrix  $\mathbf{C} \in \mathbb{R}^{2n, 2N}$ ,  $n := N - |\Gamma|$ , of that linear system has the following structure:

$$\mathbf{C} = \begin{bmatrix} \mathbf{A} & \mathbf{O} \\ \mathbf{O} & \mathbf{A} \end{bmatrix}, \quad (\mathbf{A})_{i,j} = \begin{cases} \#S(i) & , \text{ if } i = j, \\ -1 & , \text{ if } j \in S(i), \\ 0 & \text{else.} \end{cases} \quad i \in \{1, \dots, n\}, \quad j \in \{1, \dots, N\}. \quad (1.7.29)$$

$$\blacktriangleright \quad (1.7.27) \Leftrightarrow \mathbf{Cz} = \mathbf{0}. \quad (1.7.30)$$

$\Rightarrow \text{nnz}(\mathbf{A}) \leq$  number of edges of  $\mathcal{M}$  + number of interior nodes of  $\mathcal{M}$ .

$\Rightarrow$  The matrix  $\mathbf{C}$  associated with  $\mathcal{M}$  according to (1.7.29) is clearly sparse.

$\Rightarrow$  The sum of the entries in every row of  $\mathbf{C}$  vanishes.

We partition the vector  $\mathbf{z}$  into coordinates of nodes in the interior and of nodes on the boundary

$$\mathbf{z}^T = \begin{bmatrix} \mathbf{z}_1^{\text{int}} \\ \mathbf{z}_1^{\text{bd}} \\ \mathbf{z}_2^{\text{int}} \\ \mathbf{z}_2^{\text{bd}} \end{bmatrix} := [z_1, \dots, z_n, z_{n+1}, \dots, z_N, z_{N+1}, \dots, z_{N+n}, z_{N+n+1}, \dots, z_{2N}]^\top.$$

This induces the following block partitioning of the linear system (1.7.30):

$$\begin{bmatrix} \mathbf{A}_{\text{int}} & \mathbf{A}_{\text{bd}} & \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \mathbf{O} & \mathbf{A}_{\text{int}} & \mathbf{A}_{\text{bd}} \end{bmatrix} \begin{bmatrix} \mathbf{z}_1^{\text{int}} \\ \mathbf{z}_1^{\text{bd}} \\ \mathbf{z}_2^{\text{int}} \\ \mathbf{z}_2^{\text{bd}} \end{bmatrix} = \mathbf{0}, \quad \mathbf{A}_{\text{int}} \in \mathbb{R}^{n,n}, \quad \mathbf{A}_{\text{bd}} \in \mathbb{R}^{n,N-n}.$$

$\Updownarrow$

$\mathbf{A}_{\text{int}}$	$\mathbf{A}_{\text{bd}}$	
		$\mathbf{A}_{\text{int}}$
	$\mathbf{A}_{\text{bd}}$	


=  $\mathbf{0}$ .

(1.7.31)

The linear system (1.7.31) holds the key to the algorithmic realization of mesh smoothing; when smoothing the mesh

- (i) the node coordinates belonging to interior nodes have to be adjusted to satisfy the equilibrium condition (1.7.27), they are **unknowns**,
- (ii) the coordinates of nodes located on the boundary are fixed, that is, their values are known.



unknown  $\mathbf{z}_1^{\text{int}}, \mathbf{z}_2^{\text{int}}$ , known  $\mathbf{z}_1^{\text{bd}}, \mathbf{z}_2^{\text{bd}}$   
(yellow in (1.7.31)) (pink in (1.7.31))

$$(1.7.30)/(1.7.31) \Leftrightarrow \mathbf{A}_{\text{int}} [\mathbf{z}_1^{\text{int}} \mathbf{z}_2^{\text{int}}] = -[\mathbf{A}_{\text{bd}} \mathbf{z}_1^{\text{bd}} \mathbf{A}_{\text{bd}} \mathbf{z}_2^{\text{bd}}]. \quad (1.7.32)$$

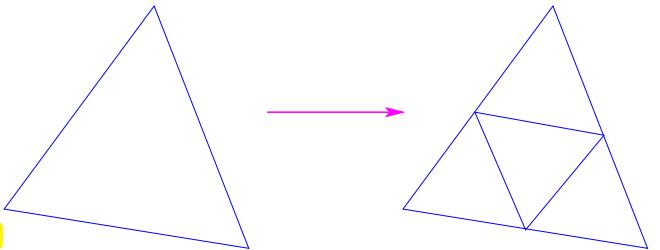
This is a square linear system with an  $n \times n$  system matrix, to be solved for two different right hand side vectors. The matrix  $\mathbf{A}_{\text{int}}$  is also known as the matrix of the **combinatorial graph Laplacian**.

We examine the sparsity pattern of the system matrices  $\mathbf{A}_{\text{int}}$  for a sequence of triangulations created by regular refinement.

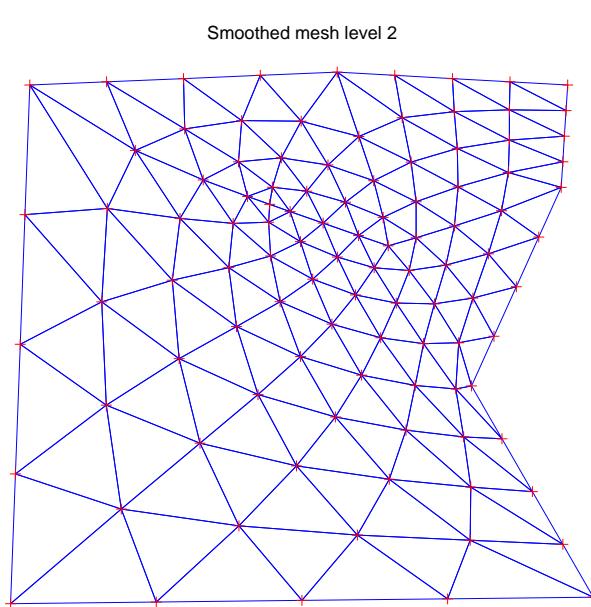
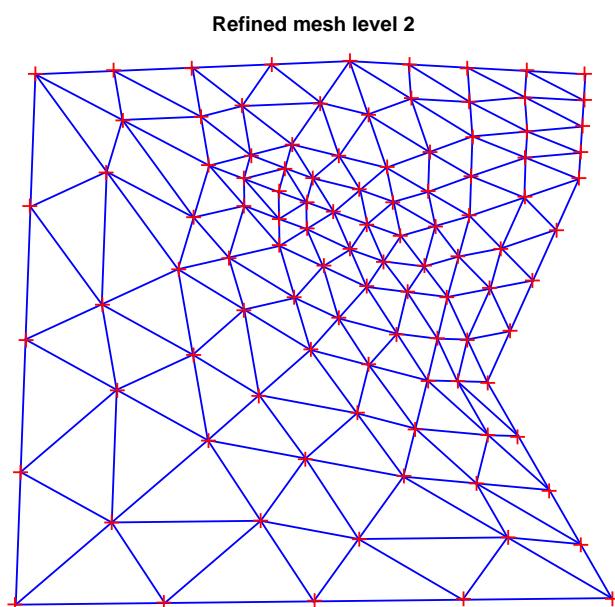
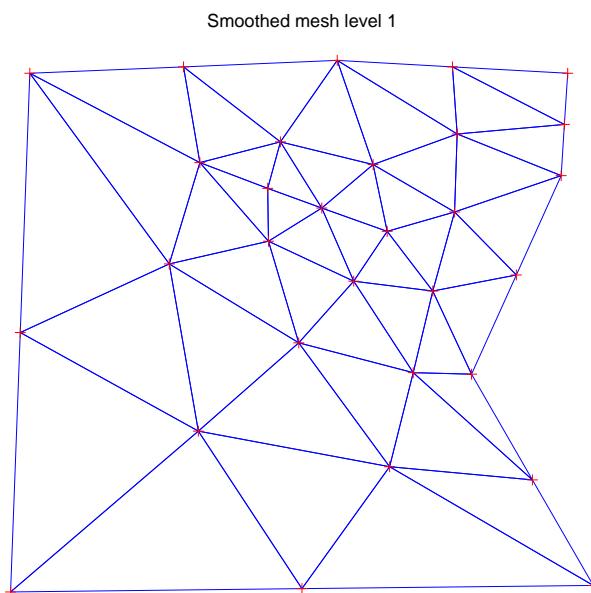
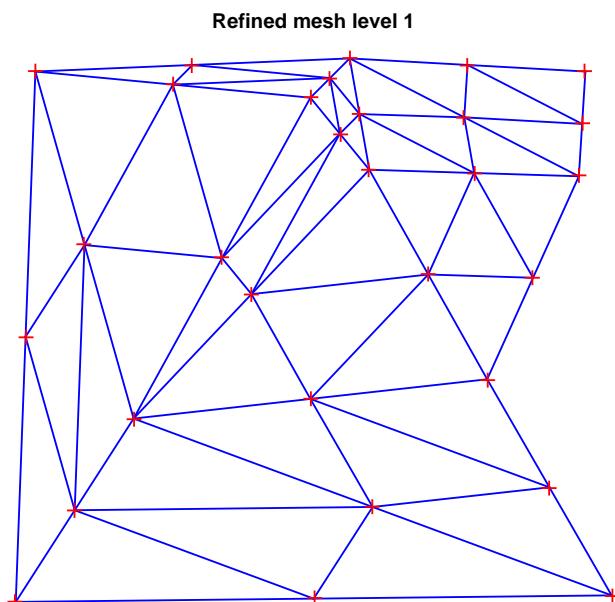
### Definition 1.7.33. Regular refinement of a planar triangulation

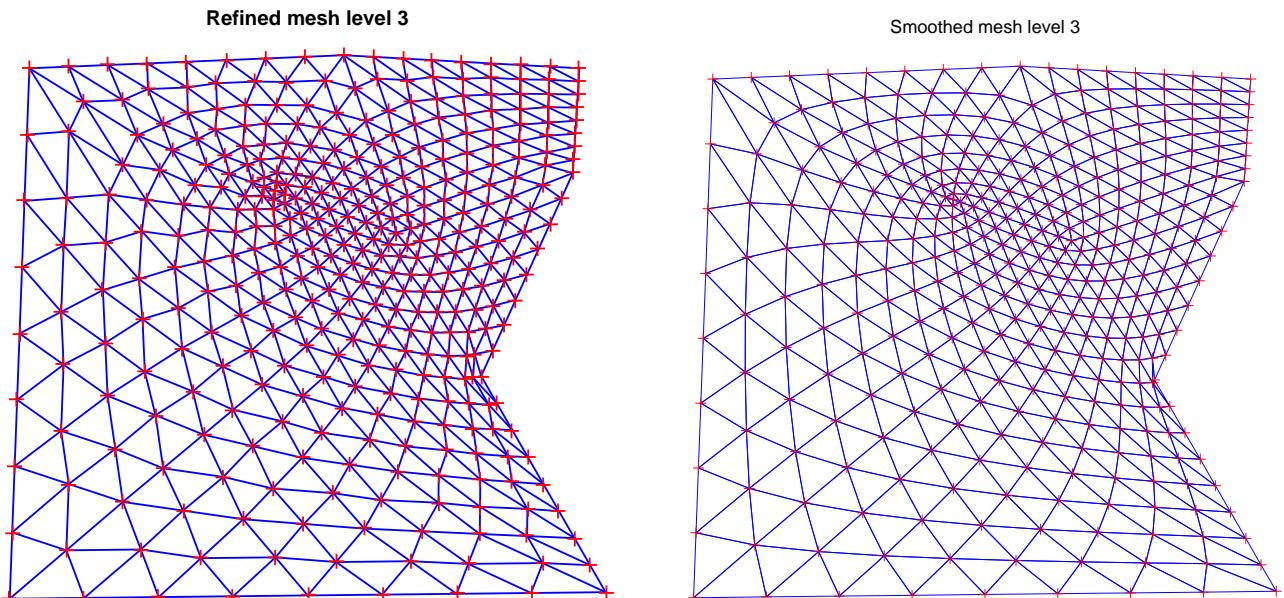
The planar triangulation with cells obtained by splitting all cells of a planar triangulation  $\mathcal{M}$  into four congruent triangles is called the **regular refinement** of  $\mathcal{M}$ .

Fig. 45

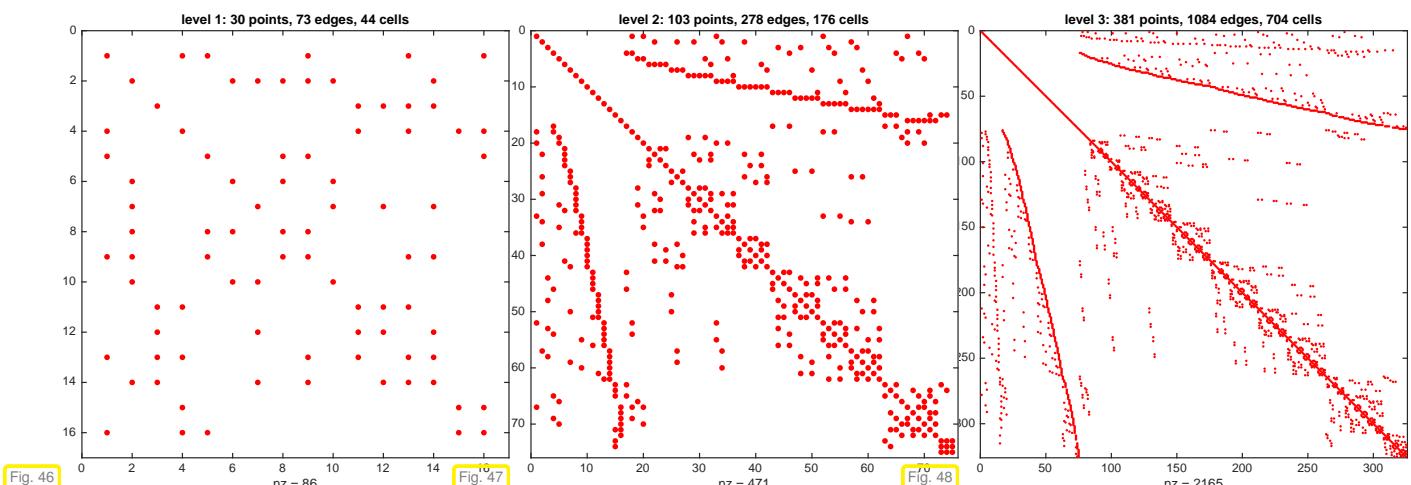


We start from the triangulation of Fig. 44 and in turns perform regular refinement and smoothing (left  $\leftrightarrow$  after refinement, right  $\leftrightarrow$  after smoothing)





spy plots of the system matrices  $\mathbf{A}_{\text{int}}$  for the first three triangulations of the sequence:



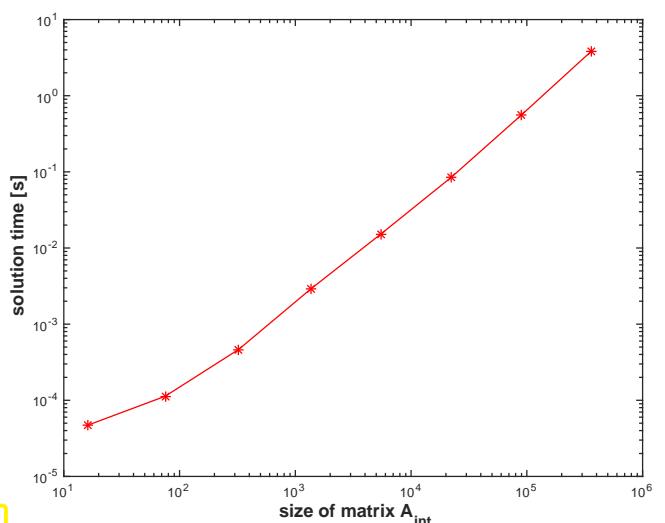
### Experiment 1.7.34 (Timing sparse elimination for the combinatorial graph Laplacian)

We consider a sequence of planar triangulations created by successive regular refinement ( $\rightarrow$  Def. 1.7.33) of the planar triangulation of Fig. 44, see Ex. 1.7.21. We use MATLAB's  $\backslash$ -solver for the linear system of equations (1.7.32) associated with each mesh.

tic-toe timing results  $\triangleright$

(Intel Core i7, 2.66 GHz, MacOS X 1.10, MATLAB 8.5.0 (R2015a))

We observe an *empirical* asymptotic complexity ( $\rightarrow$  Def. 1.4.3) of  $\mathcal{O}(n^{1.6})$ , way better than the asymptotic complexity of  $\mathcal{O}(n^3)$  expected for Gaussian elimination in the case of dense matrices.



### 1.7.3 Sparse matrices in EIGEN

Eigen can handle **sparse matrices** in the standard **Compressed Row Storage (CRS)** and **Compressed Column Storage (CCS)** format, see Ex. 1.7.9 and the [documentation](#):

```
#include <Eigen/Sparse>
Eigen::SparseMatrix<int, Eigen::ColMajor> Asp(rows,cols); // CRS
format
Eigen::SparseMatrix<double, Eigen::RowMajor> Bsp(rows,cols); // CCS
format
```

As already discussed in Exp. 1.7.13, sparse matrices must not be filled by setting entries through index-pair access. As in MATLAB, also for EIGEN the matrix should first be assembled in **triplet format**, from which a sparse matrix is built. EIGEN offers special facilities for handling triplets.

```
std::vector <Eigen::Triplet <double> > triplets;
// .. fill the std::vector triplets ..
Eigen::SparseMatrix<double, Eigen::RowMajor> spMat(rows, cols);
spMat.setFromTriplets(triplets.begin(), triplets.end());
spMat.makeCompressed();
```

The call to `makeCompressed()` removes zero entries that are still kept in the raw sparse matrix format for the sake of efficient operations. For a concrete C++ code dedicated the initialization of a sparse EIGEN matrix from triplets, see Code 1.7.41.

A triplet object can be initialized as demonstrated in the following example:

```
unsigned int row_idx = 2;
unsigned int col_idx = 4;
double value = 2.5;
Eigen::Triplet<double> triplet(row_idx,col_idx,value);
std::cout << '(' << triplet.row() << ',' << triplet.col()
<< ',' << triplet.value() << ')' << std::endl;
```

As shown, a `Triplet` object offers the access member functions `row()`, `col()`, and `value()` to fetch the row index, column index, and scalar value stored in a `Triplet`.

The statement that entry-wise initialization of sparse matrices is not efficient has to be qualified in Eigen. Entries can be set, provided that *enough space* for each row (in `RowMajor` format) is reserved in advance. This done by the `reserve()` method that takes an integer vector of maximal expected numbers of non-zero entries per row:

**C++11-code 1.7.35: Accessing entries of a sparse matrix: potentially inefficient!**

```

1 unsigned int rows, cols, max_no_nnz_per_row;
2 ...
3 SparseMatrix<double, RowMajor> mat(rows, cols);
4 mat.reserve(RowVectorXi::Constant(cols, max_no_nnz_per_row));
5 // do many (incremental) initializations
6 for ( ) {
7     mat.insert(i, j) = value_ij;
8     mat.coeffRef(i, j) += increment_ij;
9 }
10 mat.makeCompressed();

```

`insert(i, j)` sets an entry of the sparse matrix, which is rather efficient, provided that enough space has been reserved. `coeffRef(i, j)` gives l-value and r-value access to any matrix entry, creating a non-zero entry, if needed: costly!

The usual matrix operations are supported for sparse matrices; addition and subtraction may involve only sparse matrices stored in the *same format*. These operations may incur large hidden costs and have to be used with care!

**Example 1.7.36 (Initialization of sparse matrices in Eigen)**

We study the runtime behavior of the initialization of a sparse matrix in Eigen parallel to the tests from Exp. 1.7.13. We use the methods described above.

Runtimes (in ms) for the initialization of a banded matrix (with 5 non-zero diagonals, that is, a maximum of 5 non-zero entries per row) using different techniques in Eigen.

Green line: timing for entry-wise initialization with only 4 non-zero entries per row reserved in advance.

(OS: Ubuntu Linux 14.04, CPU: Intel i5@1.80 Ghz, Compiler: g++-4.8.2, -O2)

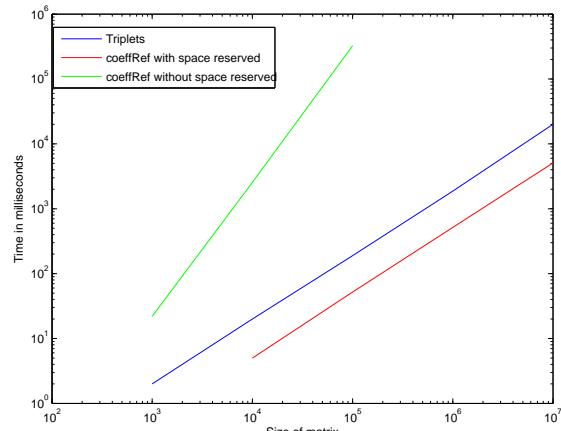


Fig. 50

Observation: insufficient advance allocation of memory massively slows down the set-up of a sparse matrix in the case of direct entry-wise initialization.

Reason: Massive internal copying of data required to create space for “unexpected” entries.

**1.7.4 Direct Solution of Sparse Linear Systems of Equations****Experiment 1.7.37 (Sparse elimination for arrow matrix)**

In Ex. 1.6.96 we saw that applying the  $\backslash$ -solver to a sparse arrow matrix results in an extreme waste of computational resources.

Yet, MATLAB can do much better! The main mistake was the creation of a dense matrix instead of storing the arrow matrix in sparse format. If the  $\backslash$ -solver is passed a matrix in sparse format, then it will rely on particular **sparse elimination techniques**. They still rely of Gaussian elimination with (partial) pivoting ( $\rightarrow$  Code 1.6.53), but take pains to operate on non-zero entries only. This can greatly boost the speed of the elimination.

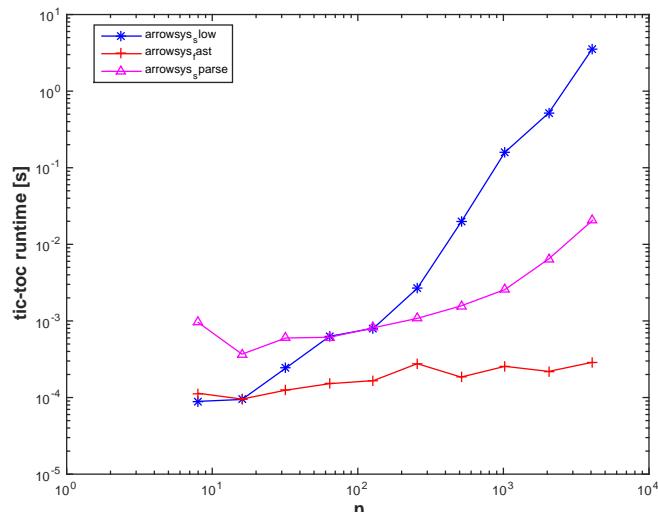
#### MATLAB-code 1.7.38: Invoking sparse elimination solver for arrow matrix

```
1 function x = arrowsys_sparse(d, c, b, alpha, y)
2 n = numel(d);
3 A = [spdiags(d, [0], n, n), c; transpose(b), alpha];
4 x = A\y;
```

Observation:

The sparse elimination solver is several orders of magnitude faster than  $\backslash$  operating on a dense matrix.

The sparse solver is still slower than Code 1.6.101. The reason is that it is a general algorithm that has to keep track of non-zero entries and has to be prepared to do pivoting.



When solving linear systems of equations directly **dedicated sparse elimination solvers** from *numerical libraries* have to be used!

System matrices are passed to these algorithms in sparse storage formats ( $\rightarrow$  1.7.1) to convey information about zero entries.



Never ever even think about implementing a general sparse elimination solver by yourself!

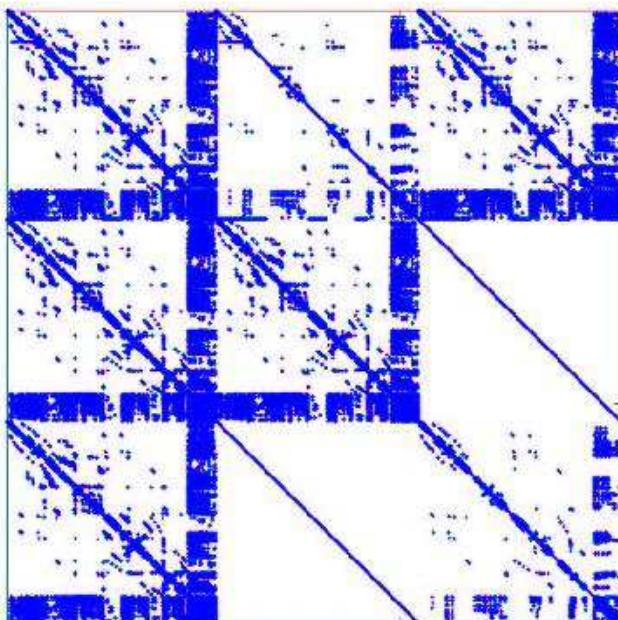
MATLAB automatically calls a sparse solver, when  $\backslash$  is applied with a matrix in sparse format.

#### (1.7.39) Implementations of sparse solvers

Widely used implementations of sparse solvers are:

- SuperLU (<http://www.cs.berkeley.edu/~demmel/SuperLU.html>),
- UMFPACK (<http://www.cise.ufl.edu/research/sparse/umfpack/>), used by MATLAB's  $\backslash$ ,

→ PARDISO [71] (<http://www.pardiso-project.org/>), incorporated into MKL



▷ fill-in ( $\rightarrow$  Def. 1.7.45) during sparse elimination with PARDISO

PARDISO has been developed by Prof. O. Schenk and his group (formerly University of Basel, now USI Lugano).



Fig. 52

### (1.7.40) Solving sparse linear systems in EIGEN

#### C++11-code 1.7.41: Initialisation of sample sparse matrix in eigen

```

1  template <class SpMat>
2  SpMat initSparseMatrix(size_t n)
3  {
4      using index_t = typename SpMat::Index;
5      using scalar_t = typename SpMat::Scalar;
6      vector<Eigen::Triplet<scalar_t>> triplets(5*n);
7
8      for (size_t l=0; l<n; ++l)
9          triplets.push_back(Eigen::Triplet<scalar_t>(l,l,5.0));
10     for (size_t l=1; l<n; ++l) {
11         triplets.push_back(Eigen::Triplet<scalar_t>(l-1,l,1.0));
12         triplets.push_back(Eigen::Triplet<scalar_t>(l,l-1,1.0));
13     }
14     const size_t m = n/2;
15     for (size_t l=0; l<m; ++l) {
16         triplets.push_back(Eigen::Triplet<scalar_t>(l,l+m,1.0));
17         triplets.push_back(Eigen::Triplet<scalar_t>(l+m,l,1.0));
18     }
19     SpMat M(n,n);
20     M.setFromTriplets(triplets.begin(), triplets.end());
21     M.makeCompressed();
22     return M;
23 }
```

**C++11-code 1.7.42: Solving a sparse linear system of equations in EIGEN**

```

1 int solveSparseTest(size_t n)
2 {
3     using SpMat = Eigen::SparseMatrix<double, Eigen::RowMajor>;
4
5     const SpMat M = initSparseMatrix<SpMat>(n);
6     cout << "M = " << M.rows() << 'x' << M.cols() << "-matrix with "
7         << M.nonZeros() << "non-zeros" << endl;
8
9     const Eigen::VectorXd b = Eigen::VectorXd::Random(n);
10    Eigen::VectorXd x(n);
11
12    Eigen::SparseLU<SpMat> solver; solver.compute(M);
13    if(solver.info() != Eigen::Success) {
14        cerr << "Decomposition failed!" << endl; return(-1);
15    }
16    x = solver.solve(b);
17    if(solver.info() != Eigen::Success) {
18        cerr << "Solver failed!" << endl; return(-1);
19    }
20    cout << "Residual norm = " << (M*x-b).norm() << endl;
21    return(0);
22}

```

The `compute` method of the solver object triggers the actual sparse LU-decomposition. The `solve` method then does forward and backward elimination, *cf.* § 1.6.42. It can be called multiple times, see Rem. 1.6.87.

## 1.7.5 LU-factorization of sparse matrices

In Sect. 1.7.1 we have seen, how sparse matrices can be stored requiring  $\mathcal{O}(\text{nnz}(\mathbf{A}))$  memory.

In Ex. 1.7.18 we found that (sometimes) matrix multiplication of sparse matrices can also be carried out with optimal complexity, that is, with computational effort proportional to the total number of non-zero entries of all matrices involved.

Does this carry over to the solution of linear systems of equations with sparse system matrices? Section 1.7.4 says “Yes”, when sophisticated library routines are used. In this section, we examine some aspects of Gaussian elimination  $\leftrightarrow$  LU-factorisation when applied in a sparse context.

### Example 1.7.43 (LU-factorization of sparse matrices)

We examine the following “sparse” matrix with a typical structure and inspect the **pattern** of the LU-factors

returned by MATLAB, see Code 1.7.44.

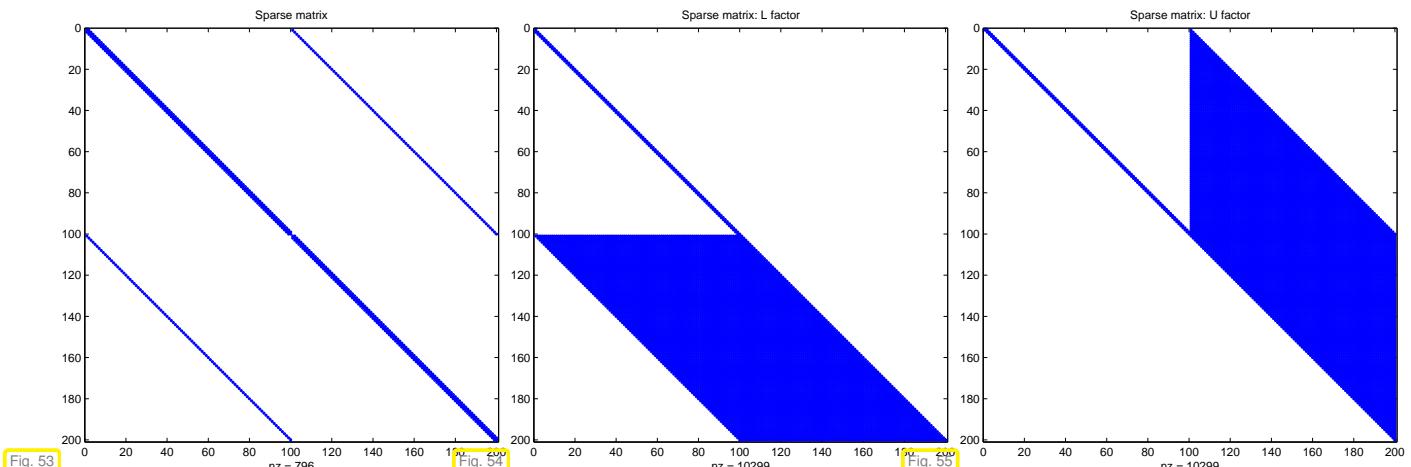
$$\mathbf{A} = \left( \begin{array}{cccc|ccc} 3 & -1 & & & -1 & & & \\ -1 & \ddots & \ddots & & \ddots & \ddots & & \\ & \ddots & \ddots & -1 & & & \ddots & \\ & & -1 & 3 & 3 & -1 & & -1 \\ \hline -1 & & & 3 & -1 & & & \\ & \ddots & & -1 & \ddots & \ddots & & \\ & & \ddots & & \ddots & -1 & \ddots & \\ & & & -1 & & -1 & 3 & -1 \end{array} \right) \in \mathbb{R}^{n,n}, n \in \mathbb{N}$$

#### MATLAB-code 1.7.44: LU-factorization of sparse matrix

```

1 % Demonstration of fill-in for LU-factorization of sparse matrices
2 n = 100;
3 A = [ gallery ('tridiag',n,-1,3,-1), speye(n); speye(n) ,
      gallery ('tridiag',n,-1,3,-1)];
4 [L,U,P] = lu (A); // LU-decomposition
5 figure; spy(A); title ('Sparse matrix');
6 print -depsc2 '../PICTURES/sparseA.eps';
7 figure; spy(L); title ('Sparse matrix: L factor');
8 print -depsc2 '../PICTURES/sparseL.eps';
9 figure; spy(U); title ('Sparse matrix: U factor');
10 print -depsc2 '../PICTURES/sparseU.eps';

```



Observation:

$\mathbf{A}$  sparse  $\not\Rightarrow$  LU-factors sparse

Of course, in case the LU-factors of a sparse matrix possess many more non-zero entries than the matrix itself, the effort for solving a linear system with direct elimination will increase significantly. This can be quantified by means of the following concept:

#### Definition 1.7.45. Fill-in

Let  $\mathbf{A} = \mathbf{LU}$  be an LU-factorization ( $\rightarrow$  Sect. 1.6.2.2) of  $\mathbf{A} \in \mathbb{K}^{n,n}$ . If  $l_{ij} \neq 0$  or  $u_{ij} \neq 0$  though  $a_{ij} = 0$ , then we encounter **fill-in** at position  $(i,j)$ .

### Example 1.7.46 (Sparse LU-factors)

Ex. 1.7.43 ➤ massive fill-in can occur for sparse matrices

This example demonstrates that fill-in can largely be avoided, if the matrix has favorable structure. In this case a LSE with this particular system matrix  $\mathbf{A}$  can be solved efficiently, that is, with a computational effort  $\mathcal{O}(\text{nnz}(\mathbf{A}))$  by Gaussian elimination.

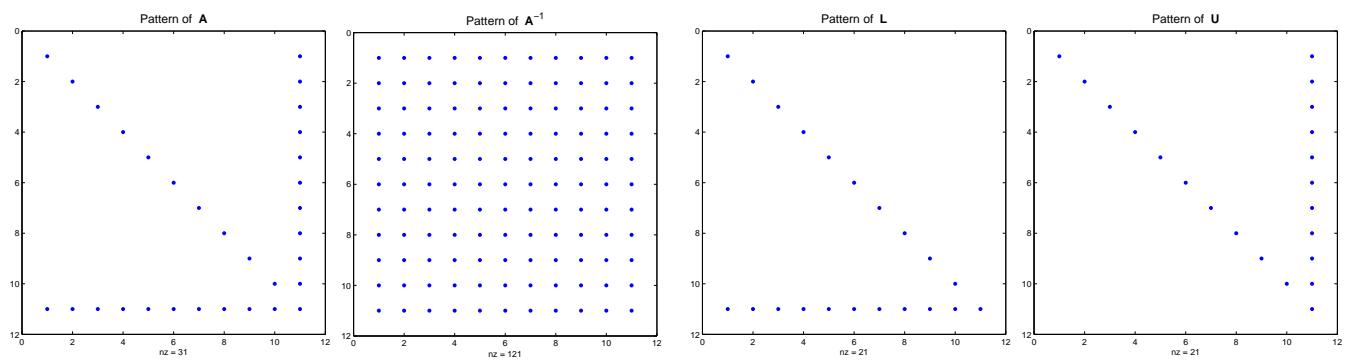
```

1 % Simple example for dense inverse despite sparse LU-factors
2 A = [ diag(1:10), ones(10,1); ones(1,10), 2];
3 [L, U] = lu(A); spy(L); spy(U); spy(inv(A));

```

$\mathbf{A}$  is called an “arrow matrix”, see the pattern of non-zero entries below and Ex. 1.6.96.

Recalling Rem. 1.6.44 it is easy to see that the LU-factors of  $\mathbf{A}$  will be sparse and that their sparsity patterns will be as depicted below. Observe that despite sparse LU-factors,  $\mathbf{A}^{-1}$  will be densely populated.



$\mathbf{L}, \mathbf{U}$  sparse  $\not\Rightarrow \mathbf{A}^{-1}$  sparse !

Besides stability and efficiency issues, see Exp. 1.6.73, this is another reason why using  $\mathbf{x} = \text{inv}(\mathbf{A}) * \mathbf{y}$  instead of  $\mathbf{y} = \mathbf{A} \backslash \mathbf{b}$  is usually a major blunder.

### Example 1.7.47 (LU-decomposition of flipped “arrow matrix”)

Recall the discussion in Ex. 1.6.96. Here we look at an arrow matrix in a slightly different form:

$$\mathbf{M} = \begin{bmatrix} \alpha & \mathbf{b}^\top \\ \mathbf{c} & \mathbf{D} \end{bmatrix}, \quad \begin{aligned} \alpha &\in \mathbb{R}, \\ \mathbf{b}, \mathbf{c} &\in \mathbb{R}^{n-1}, \\ \mathbf{D} &\in \mathbb{R}^{n-1, n-1} \text{ regular diagonal matrix, } \rightarrow \text{Def. 1.1.5} \end{aligned} \tag{1.7.48}$$

Run the algorithm from § 1.6.37 (LU decomposition without pivoting):

- \* LU-decomposition **dense** factor matrices with  $O(n^2)$  non-zero entries.
- \* asymptotic computational cost:  $O(n^3)$

#### MATLAB-code 1.7.49: LU-factorization of arrow matrix

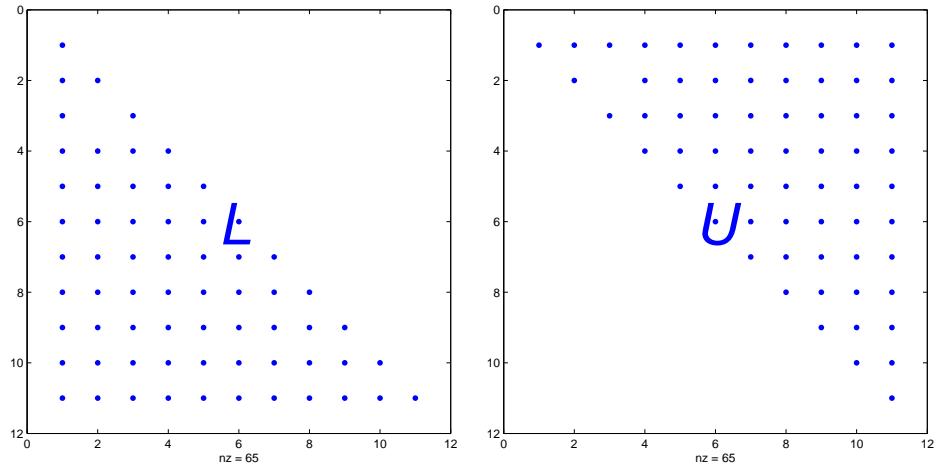
```

1 n = 10;
2 M = [ n+1, (n:-1:1);
        ones(n,1), eye(n,n) ];
3 [L,U,P] = lu(M);
4 spy(L); spy(U);

```

Output of code Code 1.7.49:

Obvious fill-in ( $\rightarrow$  Def. 1.7.45)



Now it comes as a surprise that the arrow matrix  $\mathbf{A}$  from Ex. 1.6.96, (1.6.97) has sparse LU-factors!

Arrow matrix (1.6.97)

$$\mathbf{A} = \begin{pmatrix} & & \\ & \mathbf{D} & \\ & & \mathbf{c} \\ \mathbf{b}^\top & & \alpha \end{pmatrix} \quad \triangleright$$

$$\mathbf{A} = \underbrace{\begin{pmatrix} & \mathbf{I} & \\ & & 0 \\ \mathbf{b}^\top \mathbf{D}^{-1} & & 1 \end{pmatrix}}_{=: \mathbf{L}} \cdot \underbrace{\begin{pmatrix} & \mathbf{D} & \\ & & \mathbf{c} \\ 0 & & \sigma \end{pmatrix}}_{=: \mathbf{U}}, \quad \sigma := \alpha - \mathbf{b}^\top \mathbf{D}^{-1} \mathbf{c}.$$

- In this case LU-factorisation is possible **without fill-in**, cost merely  $O(n)$ !



Idea: Transform  $\mathbf{A}$  into  $\mathbf{A}'$  by row and column **permutations** before performing LU-decomposition.

Details: Apply a **cyclic permutation** of rows/columns:

- 1st row/column  $\rightarrow n$ -th row/column
- $i$ -th row/column  $\rightarrow i-1$ -th row/column,  $i = 2, \dots, n$

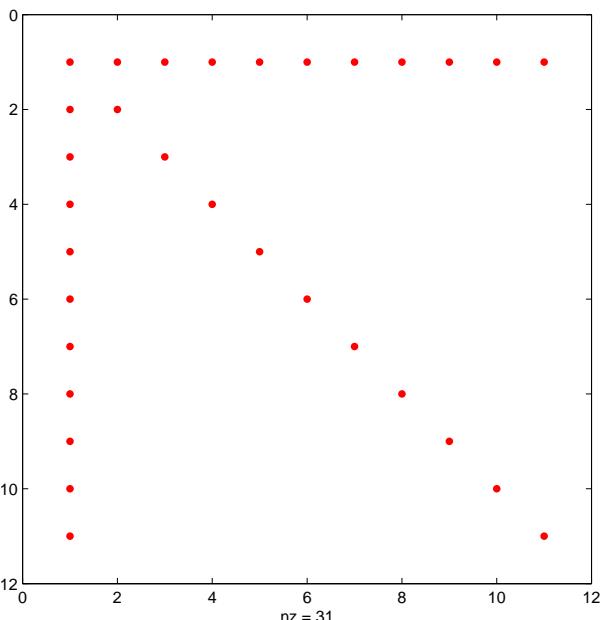


Fig. 56

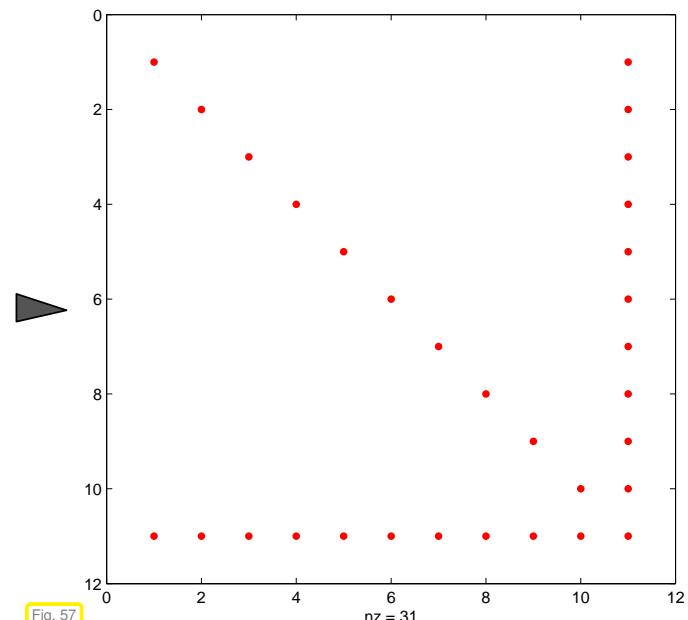


Fig. 57

- Then LU-factorization (*without pivoting*) of the resulting matrix requires  $O(n)$  operations.

#### MATLAB-code 1.7.50: Permuting arrow matrix, see Figs. 56, 57

```

1 n = 10; A = [ n+1, (n:-1:1) ; ones(n,1), eye(n,n) ];
2 % Permutation matrix (→ Def. 1.6.60) encoding cyclic permutation
3 P = [ zeros(n,1), eye(n); 1, zeros(1,n) ];
4
5 figure ('name','A');
6 spy(A,'r.'); print -depsc '../PICTURES/InvArrowSpy.eps';
7 figure ('name','PAPT');
8 spy(P*A*P','r.'); print -depsc '../PICTURES/ArrowSpy.eps';

```

#### Example 1.7.51 (Pivoting destroys sparsity)

In Ex. 1.7.47 we found that permuting a matrix can make it amenable to Gaussian elimination/LU-decomposition with much less fill-in (→ Def. 1.7.45). However, recall from Section 1.6.2.3 that pivoting, which may be essential for achieving numerical stability, amounts to permuting the rows (or even columns) of the matrix.

Thus, we may face the awkward situation that pivoting tries to reverse the very permutation we applied to minimize fill-in! The next example shows that this can happen for an arrow matrix.

#### MATLAB-code 1.7.52: fill-in due to pivoting

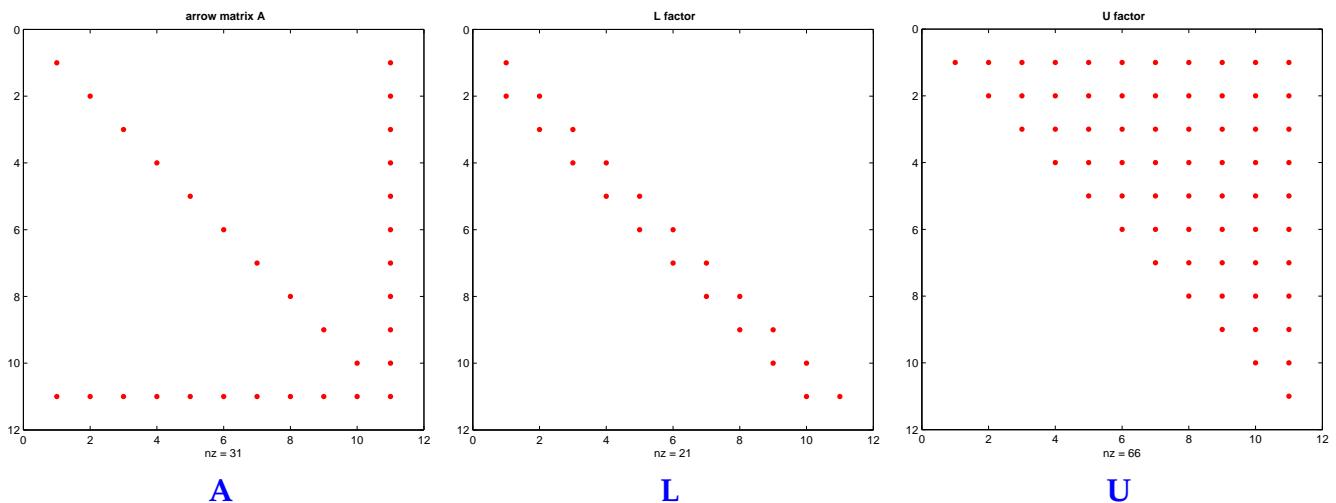
```

1 % Study of fill-in with LU-factorization due to pivoting
2 n = 10; D = diag(1./(1:n));
3 A = [ D , 2*ones(n,1); 2*ones(1,n), 2];
4 [L,U,P] = lu(A);
5 figure; spy(A,'r'); title ('{\bf arrow matrix A}');
6 print -depsc2 '../PICTURES/fillinpivotA.eps';
7 figure; spy(L,'r'); title ('{\bf L factor}');
8 print -depsc2 '../PICTURES/fillinpivotL.eps';
9 figure; spy(U,'r'); title ('{\bf U factor}');
10 print -depsc2 '../PICTURES/fillinpivotU.eps';

```

$$\mathbf{A} = \begin{bmatrix} 1 & & & & 2 \\ & \frac{1}{2} & & & 2 \\ & & \ddots & & \vdots \\ 2 & & \dots & \frac{1}{10} & 2 \end{bmatrix} \rightarrow \text{arrow matrix, Ex. 1.7.46}$$

The distributions of non-zero entries of the computed LU-factors (“spy-plots”) are as follows:



In this case the solution of a LSE with system matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$  of the above type by means of Gaussian elimination with partial pivoting would incur costs of  $O(n^3)$ .

#### 1.7.6 Banded matrices [15, Sect. 3.7]

Banded matrices are a special class of sparse matrices ( $\rightarrow$  Notion 1.7.1 with extra structure):

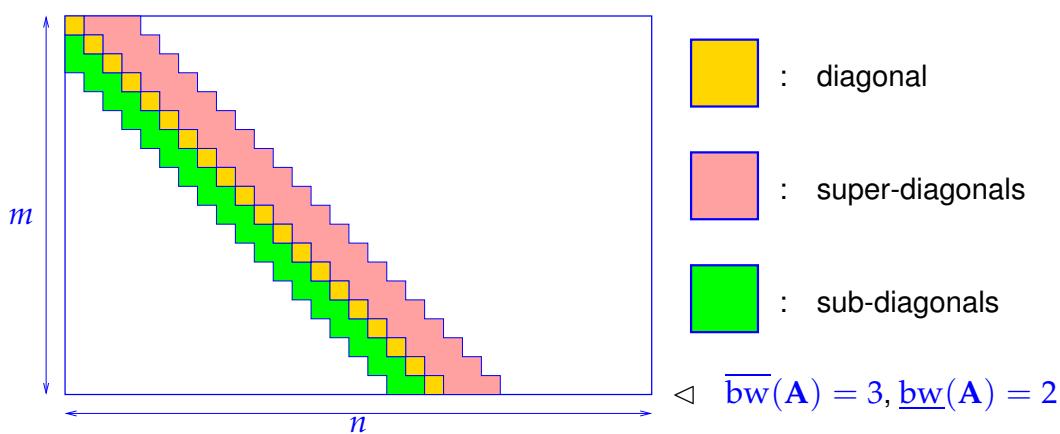
**Definition 1.7.53. Bandwidth**

For  $\mathbf{A} = (a_{ij})_{i,j} \in \mathbb{K}^{m,n}$  we call

$$\begin{aligned}\overline{\text{bw}}(\mathbf{A}) &:= \min\{k \in \mathbb{N}: j - i > k \Rightarrow a_{ij} = 0\} \text{ upper bandwidth ,} \\ \underline{\text{bw}}(\mathbf{A}) &:= \min\{k \in \mathbb{N}: i - j > k \Rightarrow a_{ij} = 0\} \text{ lower bandwidth .}\end{aligned}$$

$$\text{bw}(\mathbf{A}) := \overline{\text{bw}}(\mathbf{A}) + \underline{\text{bw}}(\mathbf{A}) + 1 = \text{bandwidth von } \mathbf{A}.$$

- $\text{bw}(\mathbf{A}) = 1 \Rightarrow \mathbf{A}$  diagonal matrix,  $\rightarrow$  Def. 1.1.5
- $\overline{\text{bw}}(\mathbf{A}) = \underline{\text{bw}}(\mathbf{A}) = 1 \Rightarrow \mathbf{A}$  tridiagonal matrix
- More general:  $\mathbf{A} \in \mathbb{R}^{n,n}$  with  $\text{bw}(\mathbf{A}) \ll n \hat{=} \text{banded matrix}$



► for banded matrix  $\mathbf{A} \in \mathbb{K}^{m,n}$ :  $\text{nnz}(\mathbf{A}) \leq \min\{m, n\} \text{bw}(\mathbf{A})$

MATLAB functions for creating banded matrices:

dense matrix	$\mathbf{X} = \text{diag}(\mathbf{v})$ ;
sparse matrix	$\mathbf{X} = \text{spdiags}(\mathbf{B}, \mathbf{d}, \mathbf{m}, \mathbf{n})$ ; (sparse storage !)
tridiagonal matrix	$\mathbf{X} = \text{gallery}('tridiag', \mathbf{c}, \mathbf{d}, \mathbf{e})$ ; (sparse storage !)

We now examine a generalization of the concept of a banded matrix that is particularly useful in the context of Gaussian elimination:

**Definition 1.7.54. Matrix envelope**

For  $\mathbf{A} \in \mathbb{K}^{n,n}$  define

$$\text{row bandwidth} \quad \text{bw}_i^R(\mathbf{A}) := \max\{0, i - j : a_{ij} \neq 0, 1 \leq j \leq n\}, i \in \{1, \dots, n\}$$

$$\text{column bandwidth} \quad \text{bw}_j^C(\mathbf{A}) := \max\{0, j - i : a_{ij} \neq 0, 1 \leq i \leq n\}, j \in \{1, \dots, n\}$$

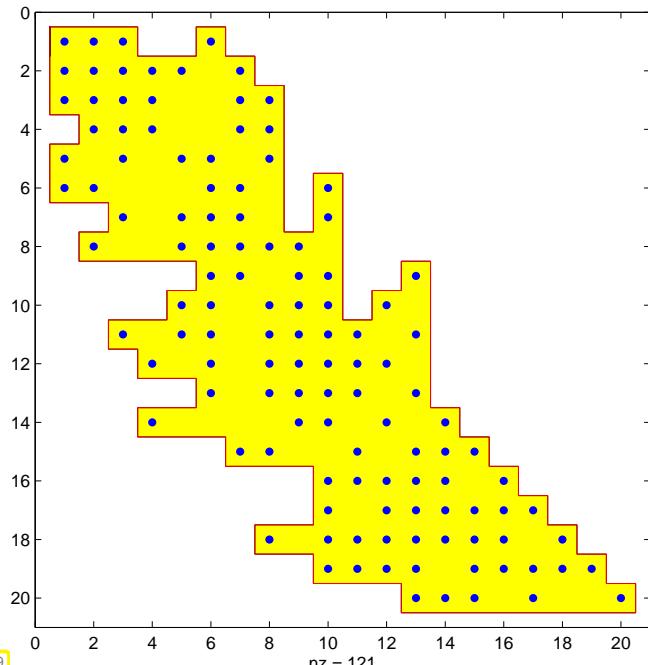
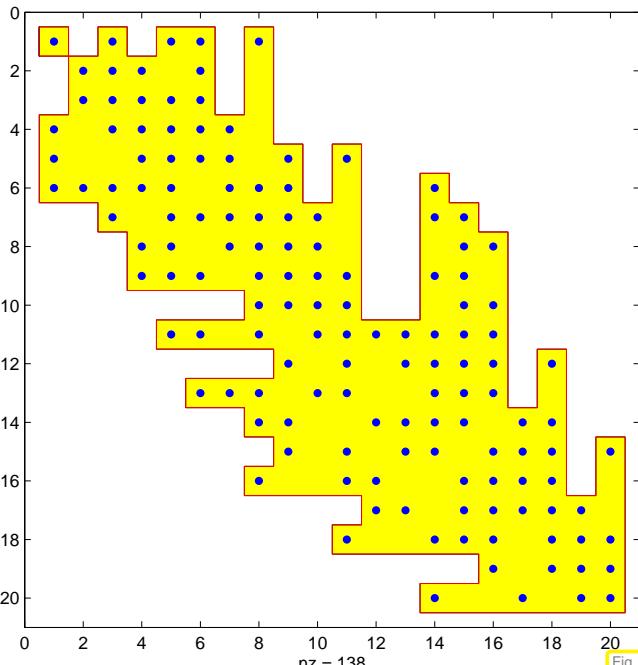
$$\text{envelope} \quad \text{env}(\mathbf{A}) := \left\{ (i, j) \in \{1, \dots, n\}^2 : \begin{array}{l} i - \text{bw}_i^R(\mathbf{A}) \leq j \leq i, \\ j - \text{bw}_j^C(\mathbf{A}) \leq i \leq j \end{array} \right\}$$

**Example 1.7.55 (Envelope of a matrix)**

$$\mathbf{A} = \begin{pmatrix} * & 0 & * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & * & 0 & 0 \\ * & 0 & * & 0 & 0 & 0 & * \\ 0 & 0 & 0 & * & * & 0 & * \\ 0 & * & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & * & * & 0 \\ 0 & 0 & * & * & 0 & 0 & * \end{pmatrix} \quad \begin{aligned} \text{bw}_1^R(A) &= 0 \\ \text{bw}_2^R(A) &= 0 \\ \text{bw}_3^R(A) &= 2 \\ \text{bw}_4^R(A) &= 0 \\ \text{bw}_5^R(A) &= 3 \\ \text{bw}_6^R(A) &= 1 \\ \text{bw}_7^R(A) &= 4 \end{aligned}$$

$\text{env}(A)$  = red entries  
 $*$   $\hat{=}$  non-zero matrix entry  $a_{ij} \neq 0$

Starting from a “spy-plot”, it is easy to find the envelope:



Note: the envelope of the arrow matrix from Ex. 1.7.46 is just the set of index pairs of its non-zero entries. Hence, the following theorem provides another reason for the sparsity of the LU-factors in that example.

### Theorem 1.7.56. Envelope and fill-in $\rightarrow$ [63, Sect. 3.9]

If  $\mathbf{A} \in \mathbb{K}^{n,n}$  is regular with LU-factorization  $\mathbf{A} = \mathbf{LU}$ , then fill-in ( $\rightarrow$  Def. 1.7.45) is confined to  $\text{env}(\mathbf{A})$ .

Gaussian elimination without pivoting

*Proof.* (by induction, version I) Examine first step of Gaussian elimination without pivoting,  $a_{11} \neq 0$

$$\mathbf{A} = \begin{pmatrix} a_{11} & \mathbf{b}^\top \\ \mathbf{c} & \tilde{\mathbf{A}} \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 \\ -\frac{\mathbf{c}}{a_{11}} & \mathbf{I} \end{pmatrix}}_{\mathbf{L}^{(1)}} \underbrace{\begin{pmatrix} a_{11} & \mathbf{b}^\top \\ 0 & \tilde{\mathbf{A}} - \frac{\mathbf{c}\mathbf{b}^\top}{a_{11}} \end{pmatrix}}_{\mathbf{U}^{(1)}}$$

$$\text{If } (i, j) \notin \text{env}(\mathbf{A}) \Rightarrow \begin{cases} c_{i-1} = 0 & , \text{ if } i > j, \\ b_{j-1} = 0 & , \text{ if } i < j. \end{cases}$$

$$\Rightarrow \text{env}(\mathbf{L}^{(1)}) \subset \text{env}(\mathbf{A}), \quad \text{env}(\mathbf{U}^{(1)}) \subset \text{env}(\mathbf{A}).$$

Moreover,  $\text{env}(\tilde{\mathbf{A}} - \frac{\mathbf{c}\mathbf{b}^\top}{a_{11}}) = \text{env}(\mathbf{A}(2:n, 2:n))$

□

*Proof.* (by induction, version II) Use block-LU-factorization, cf. Rem. 1.6.46 and proof of Lemma 1.6.35:

$$\left( \begin{array}{c|c} \tilde{\mathbf{A}} & \mathbf{b} \\ \hline \mathbf{c}^\top & \alpha \end{array} \right) = \left( \begin{array}{c|c} \tilde{\mathbf{L}} & 0 \\ \hline \mathbf{1}^\top & 1 \end{array} \right) \left( \begin{array}{c|c} \tilde{\mathbf{U}} & \mathbf{u} \\ \hline 0 & \xi \end{array} \right) \Rightarrow \begin{array}{l} \tilde{\mathbf{U}}^\top \mathbf{1} = \mathbf{c}, \\ \tilde{\mathbf{L}} \mathbf{u} = \mathbf{b}. \end{array} \quad (1.7.57)$$

From Def. 1.7.54:

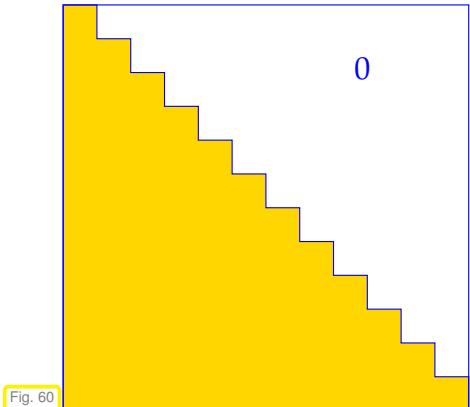


Fig. 60

If  $m_n^R(\mathbf{A}) = m$ , then  $c_1, \dots, c_{n-m} = 0$  (entries of  $\mathbf{c}$  from (1.7.57))

If  $m_n^C(\mathbf{A}) = m$ , then  $b_1, \dots, b_{n-m} = 0$  (entries of  $\mathbf{b}$  from (1.7.57))

▷ for lower triangular LSE:

If  $c_1, \dots, c_k = 0$  then  $l_1, \dots, l_k = 0$

If  $b_1, \dots, b_k = 0$ , then  $u_1, \dots, u_k = 0$



assertion of the theorem □

Thm. 1.7.56 immediately suggests a policy for saving computational effort when solving linear system whose system matrix  $\mathbf{A} \in \mathbb{K}^{n,n}$  is sparse due to *small envelope*:

$$\#\text{env}(\mathbf{A}) \ll n^2 :$$

► Policy Confine elimination to envelope!

Details will be given now:

► **Envelope-aware LU-factorization:**

#### MATLAB-code 1.7.58: computing row bandwidths, → Def. 1.7.54

```

1 function mr = rowbandwidth(A)
2 % computes row bandwidth numbers  $m_i^R(\mathbf{A})$  of  $\mathbf{A}$ 
3 n = size(A,1); mr = zeros(n,1);
4 for i=1:n, mr(i) = max(0,i-min(find(A(i,:)==0))); end

```

#### MATLAB-code 1.7.59: envelope aware forward substitution

```

1 function y = substenv(L,y,mc)
2 % envelope aware forward substitution for
3 %  $\mathbf{Lx} = \mathbf{y}$ 
4 % ( $\mathbf{L}$  = lower triangular matrix)
5 % argument mc: column bandwidth vector
6 n = size(L,1); y(1) = y(1)/L(1,1);
7 for i=2:n
8 if (mr(i) > 0)
9 zeta =
10 L(i,i-mr(i):i-1)*y(i-mr(i):i-1);
11 y(i) = (y(i) - zeta)/L(i,i);
12 else y(i) = y(i)/L(i,i); end
13 end

```

Asymptotic complexity of envelope aware forward substitution, cf. § 1.6.42, for  $\mathbf{Lx} = \mathbf{y}$ ,  $\mathbf{L} \in \mathbb{K}^{n,n}$  regular lower triangular matrix is

$$O(\#\text{env}(\mathbf{L})) !$$

By block LU-factorization ( $\rightarrow$  Rem. 1.6.46) we find

$$\left( \begin{array}{c|c} (\mathbf{A})_{1:n-1,1:n-1} & (\mathbf{A})_{1:n-1,n} \\ \hline (\mathbf{A})_{n,1:n-1} & (\mathbf{A})_{n,n} \end{array} \right) = \left( \begin{array}{c|c} \mathbf{L}_1 & 0 \\ \hline \mathbf{l}^\top & 1 \end{array} \right) \left( \begin{array}{c|c} \mathbf{U}_1 & \mathbf{u} \\ \hline 0 & \gamma \end{array} \right), \quad (1.7.60)$$

$$\Rightarrow (\mathbf{A})_{1:n-1,1:n-1} = \mathbf{L}_1 \mathbf{U}_1, \quad \mathbf{L}_1 \mathbf{u} = (\mathbf{A})_{1:n-1,n}, \quad \mathbf{U}_1^\top \mathbf{l} = (\mathbf{A})_{n,1:n-1}^\top, \quad \mathbf{l}^\top \mathbf{u} + \gamma = (\mathbf{A})_{n,n}. \quad (1.7.61)$$

### MATLAB-code 1.7.62: envelope aware recursive LU-factorization

```

1 function [L,U] = luenv(A)
2 % envelope aware recursive LU-factorization
3 % of structurally symmetric matrix
4 n = size(A,1);
5 if (size(A,2) ~= n),
6   error('A must be square'); end
7 if (n == 1), L = eye(1); U = A;
8 else
9   mr = rowbandwidth(A);
10  [L1,U1] = luenv(A(1:n-1,1:n-1));
11  u = substenv(L1,A(1:n-1,n),mr);
12  l = substenv(U1',A(n,1:n-1)',mr);
13  if (mr(n) > 0)
14    gamma = A(n,n) -
15      l(n-mr(n):n-1)' * u(n-mr(n):n-1);
16  else gamma = A(n,n); end
17  L = [L1, zeros(n-1,1); l', 1];
18  U = [U1,u; zeros(1,n-1) , gamma];
end
```

$\triangleleft$  recursive implementation of envelope aware recursive LU-factorization (no pivoting!)

Assumption:

$\mathbf{A} \in \mathbb{K}^{n,n}$  is structurally symmetric  
Asymptotic complexity  $(\mathbf{A} \in \mathbb{K}^{n,n})$

$$O(n \cdot \# \text{env}(\mathbf{A})).$$

### Definition 1.7.63. Structurally symmetric matrix

$\mathbf{A} \in \mathbb{K}^{n,n}$  is structurally symmetric, if

$$(\mathbf{A})_{i,j} \neq 0 \Leftrightarrow (\mathbf{A})_{j,i} \neq 0 \quad \forall i, j \in \{1, \dots, n\}.$$

Since by Thm. 1.7.56 fill-in is confined to the envelope, we need store only the matrix entries  $a_{ij}$ ,  $(i, j) \in \text{env}(\mathbf{A})$  when computing (in situ) LU-factorization of structurally symmetric  $\mathbf{A} \in \mathbb{K}^{n,n}$

- Storage required:  $n + 2 \sum_{i=1}^n m_i(\mathbf{A})$  floating point numbers
- terminology: envelope oriented matrix storage

### Example 1.7.64 (Envelope oriented matrix storage)

Linear envelope oriented matrix storage of symmetric  $\mathbf{A} = \mathbf{A}^\top \in \mathbb{R}^{n,n}$ :

Two arrays:

`scalar_t * val size  $P$ ,`  
`size_t * dptr size  $n$`

$$P := n + \sum_{i=1}^n m_i(A) . \quad (1.7.65)$$

$$\mathbf{A} = \begin{pmatrix} * & 0 & * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & * & 0 & 0 \\ * & 0 & * & 0 & 0 & 0 & * \\ 0 & 0 & 0 & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & 0 \\ 0 & 0 & * & * & 0 & 0 & * \end{pmatrix}$$

Indexing rule:

$$\begin{aligned} \text{dptr}[j] = k \\ \Updownarrow \\ \text{val}[k] = a_{jj} \end{aligned}$$

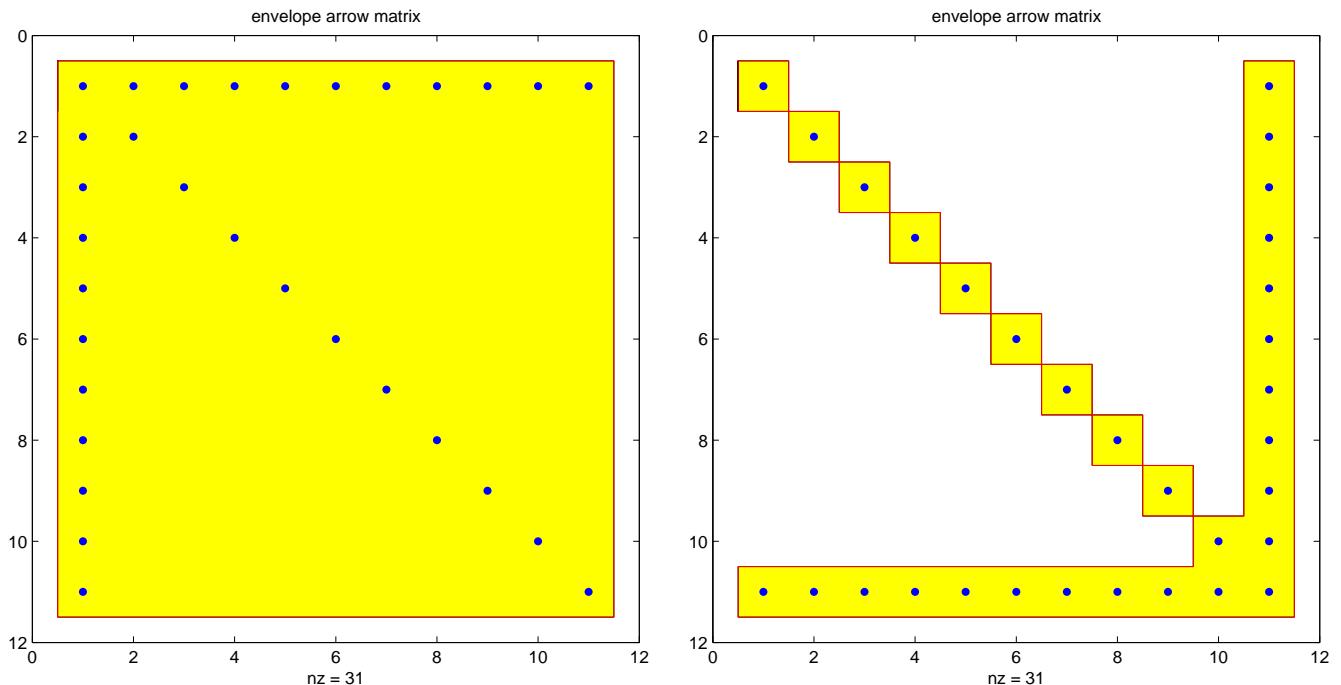
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
val		$a_{11}$	$a_{22}$	$a_{31}$	$a_{32}$	$a_{33}$	$a_{44}$	$a_{52}$	$a_{53}$	$a_{54}$	$a_{55}$	$a_{65}$	$a_{66}$	$a_{73}$	$a_{74}$	$a_{75}$	$a_{76}$	$a_{77}$
dptr	0	1	2	5	6	10	12	17										

Minimizing bandwidth/envelope:

Goal: Minimize  $m_i(\mathbf{A})$ ,  $\mathbf{A} = (a_{ij}) \in \mathbb{R}^{N,N}$ , by permuting rows/columns of  $\mathbf{A}$

### Example 1.7.66 (Reducing bandwidth by row/column permutations)

Recall: cyclic permutation of rows/columns of arrow matrix applied in Ex. 1.7.47. This can be viewed as a drastic shrinking of the envelope:



Another example: Reflection at cross diagonal  $\rightarrow$  reduction of  $\# \text{env}(\mathbf{A})$

$$\begin{array}{c} \left[ \begin{array}{cccccc} * & 0 & 0 & * & * & * \\ 0 & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 & 0 \\ * & 0 & 0 & * & * & * \\ * & 0 & 0 & 0 & * & * \\ * & 0 & 0 & * & * & * \end{array} \right] \xrightarrow{\hspace{1cm}} \left[ \begin{array}{cccccc} * & * & * & 0 & 0 & * \\ * & * & * & 0 & 0 & * \\ * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & 0 \\ * & * & * & 0 & 0 & * \end{array} \right] \\ i \leftarrow N+1-i \\ \# \text{env}(\mathbf{A}) = 30 \qquad \qquad \qquad \# \text{env}(\mathbf{A}) = 22 \end{array}$$

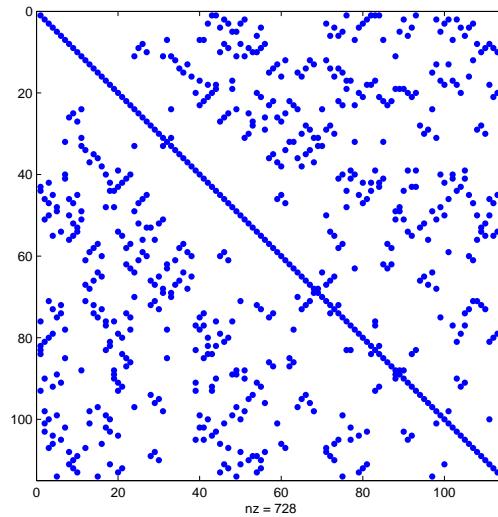
### Example 1.7.67 (Reducing fill-in by reordering)

Envelope reducing permutations are at the heart of all modern sparse solvers ( $\rightarrow$  § 1.7.39). They employ elaborate algorithms for the analysis of **matrix graphs**, that is, the connections between components of the vector of unknowns defined by non-zero entries of the matrix. For further discussion see [6, Sect. 5.7].

MATLAB supplies a few functions that return permutations aiming for minimal bandwidth/envelope of a given sparse matrix. We study an example with a  $114 \times 114$  symmetric matrix **M** originating in the numerical solution of partial differential equations, *cf.* Rem. 1.7.5.

Pattern of **M**

(Here: no row swaps from pivoting !)



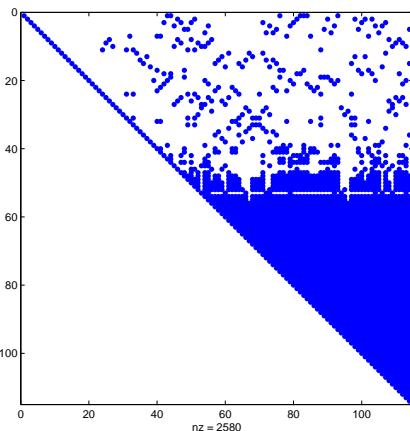
### MATLAB-code 1.7.68: preordering in MATLAB

```

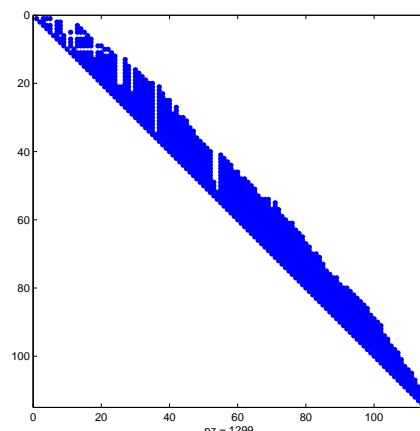
1 spy (M) ;
2 [L,U] = lu (M) ; spy (U) ;
3 r = symrcm (M) ;
4 [L,U] = lu (M(r,r)) ; spy (U) ;
5 m = symamd (M) ;
6 [L,U] = lu (M(m,m)) ; spy (U) ;

```

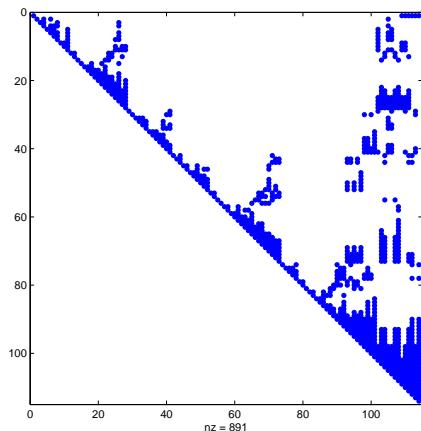
Examine patterns of LU-factors ( $\rightarrow$  Sect. 1.6.2.2) after reordering:



no reordering



reverse Cuthill-McKee



approximate minimum degree

Cuthill-McKee algorithm  $\rightarrow$  [63, Sect. 3.9.1]

## 1.8 Stable Gaussian elimination without pivoting

Recall insights, examples and experiments:

- Thm. 1.7.56  $\Rightarrow$  special structure of the matrix helps avoid fill-in in Gaussian elimination/LU-factorization *without* pivoting.
- Ex. 1.7.51  $\Rightarrow$  pivoting can trigger huge fill-in that would not occur without it.
- Ex. 1.7.67  $\Rightarrow$  fill-in reducing effect of reordering can be thwarted by later row swapping in the course of pivoting.
- **BUT** pivoting is essential for stability of Gaussian elimination/LU-factorization  $\rightarrow$  Ex. 1.6.48.

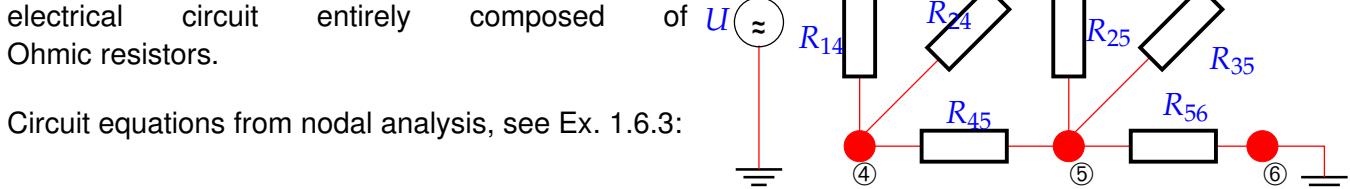
► Very desirable: a priori criteria, when Gaussian elimination/LU-factorization remains stable even without pivoting. This can help avoid the extra work for partial pivoting and makes it possible to exploit structure without worrying about stability.

This section will introduce classes of matrices that allow Gaussian elimination without pivoting. Fortunately, linear systems of equations featuring system matrices from these classes are very common in applications.

### Example 1.8.1 (Diagonally dominant matrices from nodal analysis $\rightarrow$ Ex. 1.6.3)

Consider:

electrical circuit entirely composed of  $U$



Circuit equations from nodal analysis, see Ex. 1.6.3:

$$\begin{aligned} \textcircled{2}: \quad & R_{12}^{-1}(U_2 - U_1) + R_{23}^{-1}(U_2 - U_3) + R_{24}^{-1}(U_2 - U_4) + R_{25}^{-1}(U_2 - U_5) = 0, \\ \textcircled{3}: \quad & R_{23}^{-1}(U_3 - U_2) + R_{35}^{-1}(U_3 - U_5) = 0, \\ \textcircled{4}: \quad & R_{14}^{-1}(U_4 - U_1) + R_{24}^{-1}(U_4 - U_2) + R_{45}^{-1}(U_4 - U_5) = 0, \\ \textcircled{5}: \quad & R_{25}^{-1}(U_5 - U_2) + R_{35}^{-1}(U_5 - U_3) + R_{45}^{-1}(U_5 - U_4) + R_{56}^{-1}(U_5 - U_6) = 0, \\ & U_1 = U, \quad U_6 = 0. \end{aligned}$$



$$\left( \begin{array}{ccccc} \frac{1}{R_{12}} + \frac{1}{R_{23}} + \frac{1}{R_{24}} + \frac{1}{R_{25}} & -\frac{1}{R_{23}} & -\frac{1}{R_{24}} & -\frac{1}{R_{25}} & \\ -\frac{1}{R_{23}} & \frac{1}{R_{23}} + \frac{1}{R_{35}} & 0 & -\frac{1}{R_{35}} & \\ 0 & -\frac{1}{R_{24}} & \frac{1}{R_{24}} + \frac{1}{R_{45}} & -\frac{1}{R_{45}} & \\ -\frac{1}{R_{25}} & -\frac{1}{R_{35}} & -\frac{1}{R_{45}} & \frac{1}{R_{22}} + \frac{1}{R_{35}} + \frac{1}{R_{45}} + \frac{1}{R_{56}} & \end{array} \right) \begin{pmatrix} U_2 \\ U_3 \\ U_4 \\ U_5 \end{pmatrix} = \begin{pmatrix} \frac{1}{R_{12}} \\ 0 \\ \frac{1}{R_{14}} \\ 0 \end{pmatrix} U$$

$\Rightarrow$  Matrix  $A \in \mathbb{R}^{n,n}$  arising from nodal analysis satisfies

- $A = A^\top, \quad a_{kk} > 0, \quad a_{kj} \leq 0 \text{ for } k \neq j,$  (1.8.2)

- $\sum_{j=1}^n a_{kj} \geq 0, \quad k = 1, \dots, n,$  (1.8.3)

- $\mathbf{A}$  is regular. (1.8.4)

All these properties are obvious except for the fact that  $\mathbf{A}$  is regular.

Proof of (1.8.4): By Thm. 1.6.10 it suffices to show that the nullspace of  $\mathbf{A}$  is trivial:  $\mathbf{Ax} = 0 \Rightarrow \mathbf{x} = 0$ .

Pick  $\mathbf{x} \in \mathbb{R}^n$ ,  $\mathbf{Ax} = 0$ , and  $i \in \{1, \dots, n\}$  so that

$$|x_i| = \max\{|x_j|, j = 1, \dots, n\}.$$

Intermediate goal: show that all entries of  $\mathbf{x}$  are the same

$$\mathbf{Ax} = 0 \Rightarrow x_i = \sum_{j \neq i} \frac{a_{ij}}{a_{ii}} x_j \Rightarrow |x_i| \leq \sum_{j \neq i} \frac{|a_{ij}|}{|a_{ii}|} |x_j|. \quad (1.8.5)$$

By (1.8.3) and the sign condition from (1.8.2) we conclude

$$\sum_{j \neq i} \frac{|a_{ij}|}{|a_{ii}|} \leq 1. \quad (1.8.6)$$

Hence, (1.8.6) combined with the above estimate (1.8.5) that tells us that the maximum is smaller equal than a mean implies  $|x_j| = |x_i|$  for all  $j = 1, \dots, n$ . Finally, the sign condition  $a_{kj} \leq 0$  for  $k \neq j$  enforces the same sign of all  $x_i$ . Thus, we conclude, w.l.o.g.,  $x_1 = x_2 = \dots = x_n$ . As

$$\exists i \in \{1, \dots, n\}: \quad \sum_{j=1}^n a_{ij} > 0 \quad (\text{strict inequality}),$$

$\mathbf{Ax} = 0$  is only possible for  $\mathbf{x} = 0$ .

### (1.8.7) Diagonally dominant matrices

**Definition 1.8.8. Diagonally dominant matrix** → [63, Def. 1.24]

$\mathbf{A} \in \mathbb{K}^{n,n}$  is **diagonally dominant**, if

$$\forall k \in \{1, \dots, n\}: \quad \sum_{j \neq k} |a_{kj}| \leq |a_{kk}|.$$

The matrix  $\mathbf{A}$  is called **strictly diagonally dominant**, if

$$\forall k \in \{1, \dots, n\}: \quad \sum_{j \neq k} |a_{kj}| < |a_{kk}|.$$

### Lemma 1.8.9. LU-factorization of diagonally dominant matrices

$$\mathbf{A} \text{ regular, diagonally dominant with positive diagonal} \Leftrightarrow \left\{ \begin{array}{l} \mathbf{A} \text{ has LU-factorization} \\ \Downarrow \\ \text{Gaussian elimination feasible without pivoting}^{(*)} \end{array} \right.$$

(\*): partial pivoting & diagonally dominant matrices  $\Rightarrow$  triggers no row permutations !

((1.6.54) will always be satisfied for  $j = k$ )

$\Rightarrow$  Pivoting can be dispensed with without compromising stability.

*Proof.(of Lemma 1.8.9)*

(1.6.31)  $\rightarrow$  induction w.r.t.  $n$ :

Clear: partial pivoting in the first step selects  $a_{11}$  as pivot element, cf. (1.6.54).

After 1st step of elimination:

$$a_{ij}^{(1)} = a_{ij} - \frac{a_{i1}}{a_{11}} a_{1j}, \quad i, j = 2, \dots, n \quad \Rightarrow \quad a_{ii}^{(1)} > 0.$$

$$\begin{aligned} \blacktriangleright |a_{ii}^{(1)}| - \sum_{\substack{j=2 \\ j \neq i}}^n |a_{ij}^{(1)}| &= \left| a_{ii} - \frac{a_{i1}}{a_{11}} a_{1i} \right| - \sum_{\substack{j=2 \\ j \neq i}}^n \left| a_{ij} - \frac{a_{i1}}{a_{11}} a_{1j} \right| \\ &\geq a_{ii} - \frac{|a_{i1}| |a_{1i}|}{a_{11}} - \sum_{\substack{j=2 \\ j \neq i}}^n |a_{ij}| - \frac{|a_{i1}|}{a_{11}} \sum_{\substack{j=2 \\ j \neq i}}^n |a_{1j}| \\ &\geq a_{ii} - \frac{|a_{i1}| |a_{1i}|}{a_{11}} - \sum_{\substack{j=2 \\ j \neq i}}^n |a_{ij}| - |a_{i1}| \frac{a_{11} - |a_{1i}|}{a_{11}} \geq a_{ii} - \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}| \geq 0. \end{aligned}$$

A regular, diagonally dominant  $\Rightarrow$  partial pivoting according to (1.6.54) selects  $i$ -th row in  $i$ -th step.

### (1.8.10) Gaussian elimination for symmetric positive definite (s.p.d.) matrices

The class of symmetric positive definite (s.p.d.) matrices has been defined in Def. 1.1.8. They permit stable Gaussian elimination without pivoting:

#### Theorem 1.8.11. Gaussian elimination for s.p.d. matrices

Every symmetric/Hermitian positive definite matrix ( $\rightarrow$  Def. 1.1.8) possesses an LU-decomposition ( $\rightarrow$  Sect. 1.6.2.2).

Equivalent to the assertion of the theorem: Gaussian elimination is feasible *without pivoting*

In fact, this theorem is a corollary of Lemma 1.6.35, because all principal minors of an s.p.d. matrix are s.p.d. themselves.

*Sketch of alternative self-contained proof.*

Proof by induction: consider first step of elimination

$$\mathbf{A} = \begin{pmatrix} a_{11} & \mathbf{b}^\top \\ \mathbf{b} & \tilde{\mathbf{A}} \end{pmatrix} \xrightarrow[\text{Gaussian elimination}]{\text{1. step}} \begin{pmatrix} a_{11} & \mathbf{b}^\top \\ 0 & \tilde{\mathbf{A}} - \frac{\mathbf{b}\mathbf{b}^\top}{a_{11}} \end{pmatrix}.$$

$\Rightarrow$  to show:  $\tilde{\mathbf{A}} - \frac{\mathbf{b}\mathbf{b}^\top}{a_{11}}$  s.p.d. ( $\rightarrow$  step of induction argument)



Evident: symmetry of  $\tilde{\mathbf{A}} - \frac{\mathbf{b}\mathbf{b}^\top}{a_{11}} \in \mathbb{R}^{n-1, n-1}$



As  $\mathbf{A}$  s.p.d. ( $\rightarrow$  Def. 1.1.8), for every  $\mathbf{y} \in \mathbb{R}^{n-1} \setminus \{0\}$

$$0 < \begin{pmatrix} -\frac{\mathbf{b}^\top \mathbf{y}}{a_{11}} \\ \mathbf{y} \end{pmatrix}^\top \begin{pmatrix} a_{11} & \mathbf{b}^\top \\ \mathbf{b} & \tilde{\mathbf{A}} \end{pmatrix} \begin{pmatrix} -\frac{\mathbf{b}^\top \mathbf{y}}{a_{11}} \\ \mathbf{y} \end{pmatrix} = \mathbf{y}^\top (\tilde{\mathbf{A}} - \frac{\mathbf{b}\mathbf{b}^\top}{a_{11}}) \mathbf{y}.$$

►  $\tilde{\mathbf{A}} - \frac{\mathbf{b}\mathbf{b}^\top}{a_{11}}$  positive definite. □

The proof can also be based on the identities

$$\begin{aligned} \left( \begin{array}{c|c} (\mathbf{A})_{1:n-1, 1:n-1} & (\mathbf{A})_{1:n-1, n} \\ \hline (\mathbf{A})_{n, 1:n-1} & (\mathbf{A})_{n, n} \end{array} \right) &= \left( \begin{array}{c|c} \mathbf{L}_1 & 0 \\ \hline \mathbf{1}^\top & 1 \end{array} \right) \left( \begin{array}{c|c} \mathbf{U}_1 & \mathbf{u} \\ \hline 0 & \gamma \end{array} \right), \\ \Rightarrow \quad (\mathbf{A})_{1:n-1, 1:n-1} &= \mathbf{L}_1 \mathbf{U}_1, \quad \mathbf{L}_1 \mathbf{u} = (\mathbf{A})_{1:n-1, n}, \quad \mathbf{U}_1^\top \mathbf{1} = (\mathbf{A})_{n, 1:n-1}^\top, \quad \mathbf{1}^\top \mathbf{u} + \gamma = (\mathbf{A})_{n, n}, \end{aligned} \quad (1.7.60)$$

noticing that the principal minor  $(\mathbf{A})_{1:n-1, 1:n-1}$  is also s.p.d. This allows a simple induction argument.

Note: no pivoting required ( $\rightarrow$  Sect. 1.6.2.3)  
(partial pivoting always picks current pivot row)

The next result gives a useful criterion for telling whether a given symmetric/Hermitian matrix is s.p.d.:

### Lemma 1.8.12. Diagonal dominance and definiteness

A diagonally dominant Hermitian/symmetric matrix with non-negative diagonal entries is positive semi-definite.

A strictly diagonally dominant Hermitian/symmetric matrix with positive diagonal entries is positive definite.

*Proof.* For  $\mathbf{A} = \mathbf{A}^H$  diagonally dominant, use inequality between arithmetic and geometric mean (AGM)  $ab \leq \frac{1}{2}(a^2 + b^2)$ :

$$\begin{aligned} \mathbf{x}^H \mathbf{A} \mathbf{x} &= \sum_{i=1}^n \left( a_{ii} |x_i|^2 + \sum_{i \neq j} a_{ij} \bar{x}_i x_j \right) \geq \sum_{i=1}^n \left( a_{ii} |x_i|^2 - \sum_{i \neq j} |a_{ij}| |x_i| |x_j| \right) \\ &\stackrel{\text{AGM}}{\geq} \sum_{i=1}^n a_{ii} |x_i|^2 - \frac{1}{2} \sum_{i \neq j} |a_{ij}| (|x_i|^2 + |x_j|^2) \\ &\geq \frac{1}{2} \left( \sum_{i=1}^n \{a_{ii} |x_i|^2 - \sum_{j \neq i} |a_{ij}| |x_i|^2\} \right) + \frac{1}{2} \left( \sum_{j=1}^n \{a_{jj} |x_j|^2 - \sum_{i \neq j} |a_{ij}| |x_j|^2\} \right) \\ &\geq \sum_{i=1}^n |x_i|^2 \left( a_{ii} - \sum_{j \neq i} |a_{ij}| \right) \geq 0. \end{aligned}$$

### (1.8.13) Cholesky decomposition

**Lemma 1.8.14. Cholesky decomposition for s.p.d. matrices** → [34, Sect. 3.4], [42, Sect. II.5], [63, Thm. 3.6]

For any s.p.d.  $\mathbf{A} \in \mathbb{K}^{n,n}$ ,  $n \in \mathbb{N}$ , there is a unique upper triangular matrix  $\mathbf{R} \in \mathbb{K}^{n,n}$  with  $r_{ii} > 0$ ,  $i = 1, \dots, n$ , such that  $\mathbf{A} = \mathbf{R}^H \mathbf{R}$  (Cholesky decomposition).

Thm. 1.8.11  $\Rightarrow \mathbf{A} = \mathbf{LU}$  (unique LU-decomposition of  $\mathbf{A}$ , Lemma 1.6.35)

$$\mathbf{A} = \mathbf{LD}\tilde{\mathbf{U}} \quad , \quad \begin{aligned} \mathbf{D} &\triangleq \text{diagonal of } \mathbf{U} , \\ \tilde{\mathbf{U}} &\triangleq \text{normalized upper triangular matrix} \rightarrow \text{Def. 1.1.5} \end{aligned}$$

Due to uniqueness of LU-decomposition

$$\mathbf{A} = \mathbf{A}^\top \Rightarrow \mathbf{U} = \mathbf{DL}^\top \Rightarrow \boxed{\mathbf{A} = \mathbf{LDL}^\top} ,$$

with unique  $\mathbf{L}, \mathbf{D}$  (diagonal matrix)

$$\mathbf{x}^\top \mathbf{Ax} > 0 \quad \forall \mathbf{x} \neq 0 \Rightarrow \mathbf{y}^\top \mathbf{Dy} > 0 \quad \forall \mathbf{y} \neq 0 .$$

►  $\mathbf{D}$  has positive diagonal  $\Rightarrow \mathbf{R} = \sqrt{\mathbf{DL}^\top}$ .  $\square$

Formulas analogous to (1.6.38)

$$\mathbf{R}^H \mathbf{R} = \mathbf{A} \Rightarrow a_{ik} = \sum_{j=1}^{\min\{i,k\}} \bar{r}_{ji} r_{jk} = \begin{cases} \sum_{j=1}^{i-1} \bar{r}_{ji} r_{jk} + \bar{r}_{ii} r_{ik} & , \text{if } i < k , \\ \sum_{j=1}^{i-1} |r_{ji}|^2 + r_{ii}^2 & , \text{if } i = k . \end{cases} \quad (1.8.15)$$

### MATLAB-code 1.8.16: Simple Cholesky factorization

```

1 function R = cholfac(A)
2 % simple Cholesky
3 % factorization
4 n = size(A,1);
5 for k = 1:n
6   for j=k+1:n
7     A(j,j:n) = A(j,j:n) -
8       A(k,j:n)*A(k,j)/A(k,k);
9   end
10  A(k,k:n) =
11    A(k,k:n)/sqrt(A(k,k));
12 end
13 R = triu(A);

```

Computational costs (# elementary arithmetic operations) of Cholesky decomposition:  $\frac{1}{6}n^3 + O(n^2)$

( $\geq$  “half the costs” of LU-factorization, cf. Code in § 1.6.37, but this does not mean “twice as fast” in a concrete implementation, because memory access patterns will have a crucial impact, see Rem. 1.4.7.)

Gains of efficiency hardly justify the use of Cholesky decomposition in modern numerical algorithms. Savings in memory compared to standard LU-factorization (only one factor  $\mathbf{R}$  has to be stored) offer a stronger reason to prefer the Cholesky decomposition.

MATLAB function:

$\mathbf{R} = \text{chol}(\mathbf{A})$

The computation of Cholesky-factorization by means of the algorithm of Code 1.8.16 is numerically stable ( $\rightarrow$  Def. 1.5.80)!

Reason: recall Thm. 1.6.66: Numerical instability of Gaussian elimination (with any kind of pivoting) manifests itself in massive growth of the entries of intermediate elimination matrices  $\mathbf{A}^{(k)}$ .

Use the relationship between LU-factorization and Cholesky decomposition, which tells us that we only have to monitor the growth of entries of intermediate upper triangular “Cholesky factorization matrices”  $\mathbf{A} = (\mathbf{R}^{(k)})^H \mathbf{R}^{(k)}$ .

Consider: Euclidean vector norm/matrix norm ( $\rightarrow$  Def. 1.5.71)  $\|\cdot\|_2$

$$\mathbf{A} = \mathbf{R}^H \mathbf{R} \Rightarrow \|\mathbf{A}\|_2 = \sup_{\|\mathbf{x}\|_2=1} \mathbf{x}^H \mathbf{R}^H \mathbf{R} \mathbf{x} = \sup_{\|\mathbf{x}\|_2=1} (\mathbf{R} \mathbf{x})^H (\mathbf{R} \mathbf{x}) = \|\mathbf{R}\|_2^2.$$

► For all intermediate Cholesky factorization matrices holds:  $\|(\mathbf{R}^{(k)})^H\|_2 = \|\mathbf{R}^{(k)}\|_2 = \|\mathbf{A}\|_2^{1/2}$

This rules out a blowup of entries of the  $\mathbf{R}^{(k)}$ .

Computation of the Cholesky decomposition largely agrees with the computation of LU-factorization (without pivoting). Using the latter together with forward and backward substitution ( $\rightarrow$  Sect. 1.6.2.2) to solve a linear system of equations is algebraically and numerically equivalent to using Gaussian elimination without pivoting.

From these equivalences we conclude:

Solving LSE with s.p.d. system matrix via

Cholesky decomposition + forward & backward substitution

is numerically stable ( $\rightarrow$  Def. 1.5.80)



Gaussian elimination *without pivoting* is a *numerically stable* way to solve LSEs with s.p.d. system matrix.

## Learning Outcomes

Principal take-home knowledge and skills from this chapter:

- Knowledge about the syntax of fundamental operations on matrices and vectors in MATLAB and EIGEN.
- Understanding of the concepts of computational effort/cost and asymptotic complexity in numerics.
- Awareness of the asymptotic complexity of basic linear algebra operations and dense Gaussian elimination
- Ability to determine the (asymptotic) computational effort for a concrete (numerical linear algebra) algorithm.
- Ability to manipulate simple expressions involving matrices and vectors in order to reduce the computational cost for their evaluation.
- Familiarity with “sparse matrices”: notion, data structures, initialization, benefits

# Chapter 2

## Iterative Methods for Non-Linear Systems of Equations

### Example 2.0.1 (Non-linear electric circuit)

Non-linear systems naturally arise in mathematical models of electrical circuits, once non-linear circuit elements are introduced. This generalizes Ex. 1.6.3, where the current-voltage relationship for all circuit elements was the simple proportionality (1.6.5) (of the complex amplitudes  $U$  and  $I$ ).

As an example we consider the

**Schmitt trigger circuit**

Its key non-linear circuit element is the NPN bipolar junction transistor:

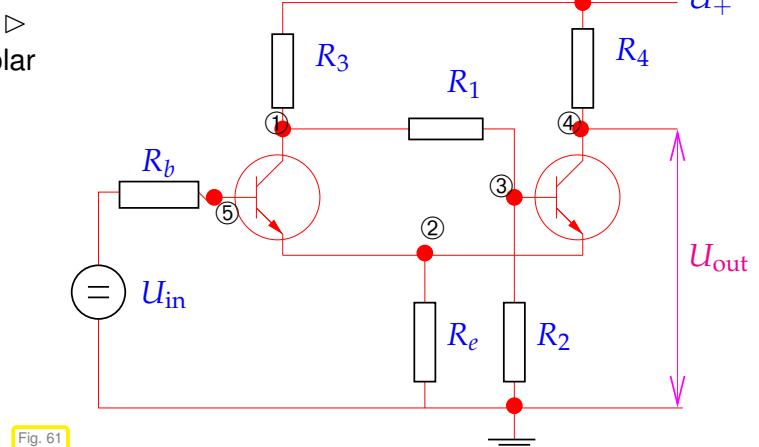
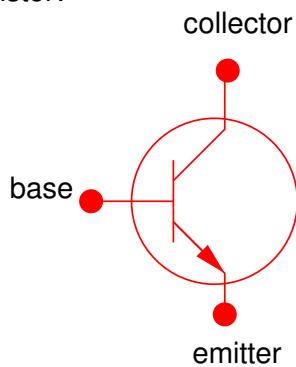


Fig. 61

A transistor has three ports: emitter, collector, and base. Transistor models give the port currents as functions of the applied voltages, for instance the **Ebers-Moll model** (large signal approximation):

$$\begin{aligned} I_C &= I_S \left( e^{\frac{U_{BE}}{U_T}} - e^{\frac{U_{BC}}{U_T}} \right) - \frac{I_S}{\beta_R} \left( e^{\frac{U_{BC}}{U_T}} - 1 \right) = I_C(U_{BE}, U_{BC}), \\ I_B &= \frac{I_S}{\beta_F} \left( e^{\frac{U_{BE}}{U_T}} - 1 \right) + \frac{I_S}{\beta_R} \left( e^{\frac{U_{BC}}{U_T}} - 1 \right) = I_B(U_{BE}, U_{BC}), \\ I_E &= I_S \left( e^{\frac{U_{BE}}{U_T}} - e^{\frac{U_{BC}}{U_T}} \right) + \frac{I_S}{\beta_F} \left( e^{\frac{U_{BE}}{U_T}} - 1 \right) = I_E(U_{BE}, U_{BC}). \end{aligned} \quad (2.0.2)$$

$I_C, I_B, I_E$ : current in collector/base/emitter,

$U_{BE}, U_{BC}$ : potential drop between base-emitter, base-collector.

The parameters have the following meanings:  $\beta_F$  is the forward common emitter current gain (20 to 500),  $\beta_R$  is the reverse common emitter current gain (0 to 20),  $I_S$  is the reverse saturation current (on the order of  $10^{-15}$  to  $10^{-12}$  amperes),  $U_T$  is the thermal voltage (approximately 26 mV at 300 K).

The circuit of Fig. 61 has 5 nodes ①–⑤ with unknown nodal potentials. Kirchhoff's law (1.6.4) plus the constitutive relations gives an equation for each of them.

► **Non-linear system of equations** from nodal analysis, static case ( $\rightarrow$  Ex. 1.6.3):

$$\begin{aligned} \textcircled{1} : \quad & R_3(U_1 - U_+) + R_1(U_1 - U_3) + I_B(U_5 - U_1, U_5 - U_2) = 0, \\ \textcircled{2} : \quad & R_e U_2 + I_E(U_5 - U_1, U_5 - U_2) + I_E(U_3 - U_4, U_3 - U_2) = 0, \\ \textcircled{3} : \quad & R_1(U_3 - U_1) + I_B(U_3 - U_4, U_3 - U_2) = 0, \\ \textcircled{4} : \quad & R_4(U_4 - U_+) + I_C(U_3 - U_4, U_3 - U_2) = 0, \\ \textcircled{5} : \quad & R_b(U_5 - U_{in}) + I_B(U_5 - U_1, U_5 - U_2) = 0. \end{aligned} \quad (2.0.3)$$

5 equations  $\leftrightarrow$  5 unknowns  $U_1, U_2, U_3, U_4, U_5$

Formally:

$$(2.0.3) \longleftrightarrow F(\mathbf{u}) = \mathbf{0} \quad \text{with a function } F : \mathbb{R}^5 \rightarrow \mathbb{R}^5$$

#### Remark 2.0.4 (General non-linear systems of equations)

A **non-linear system of equations** is a concept almost *too abstract to be useful*, because it covers an extremely wide variety of problems. Nevertheless in this chapter we will mainly look at “generic” methods for such systems. This means that every method discussed may take a good deal of fine-tuning before it will really perform satisfactorily for a given non-linear system of equations.

#### (2.0.5) Generic/general non-linear system of equations

Given:

$$\text{function } F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n, \quad n \in \mathbb{N}$$

$\Updownarrow$

Possible meanings: ☐  $F$  is known as an analytic expression

☒ Mere availability of a (black-box) **procedure** function  $y=F(x)$  evaluating  $F$

Here,  $D$  is the domain of definition of the function  $F$ , which cannot be evaluated for  $x \notin D$ .

Sought: solution(s)  $x \in D$  of **non-linear equation**  $F(x) = 0$

Note:  $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n \leftrightarrow$  “same number of equations and unknowns”

In contrast to the situation for linear systems of equations ( $\rightarrow$  Thm. 1.6.10), the class of non-linear systems is far too big to allow a general theory:

There are no general results existence & uniqueness of solutions of  $F(x) = 0$

## Contents

2.1 Iterative methods . . . . .	185
2.1.1 Speed of convergence . . . . .	187
2.1.2 Termination criteria . . . . .	191
2. Iterative Methods for Non-Linear Systems of Equations, 2. Iterative Methods for Non-Linear Systems of Equations	184

<b>2.2 Fixed Point Iterations . . . . .</b>	<b>195</b>
2.2.1 Consistent fixed point iterations . . . . .	195
2.2.2 Convergence of fixed point iterations . . . . .	197
<b>2.3 Finding Zeros of Scalar Functions . . . . .</b>	<b>203</b>
2.3.1 Bisection . . . . .	203
2.3.2 Model function methods . . . . .	205
2.3.2.1 Newton method in scalar case . . . . .	205
2.3.2.2 Special one-point methods . . . . .	208
2.3.2.3 Multi-point methods . . . . .	211
2.3.3 Asymptotic efficiency of iterative methods for zero finding . . . . .	216
<b>2.4 Newton's Method . . . . .</b>	<b>218</b>
2.4.1 The Newton iteration . . . . .	218
2.4.2 Convergence of Newton's method . . . . .	228
2.4.3 Termination of Newton iteration . . . . .	230
2.4.4 Damped Newton method . . . . .	232
2.4.5 Quasi-Newton Method . . . . .	236
<b>2.5 Unconstrained Optimization . . . . .</b>	<b>242</b>
2.5.1 Minima and minimizers: Some theory . . . . .	242
2.5.2 Newton's method . . . . .	242
2.5.3 Descent methods . . . . .	242
2.5.4 Quasi-Newton methods . . . . .	242

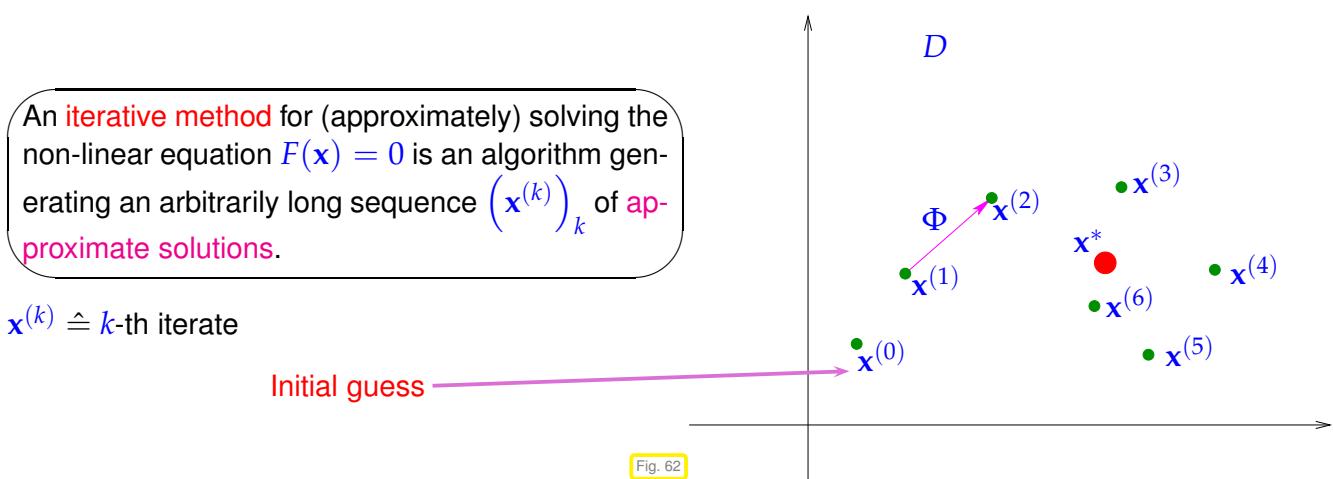
## 2.1 Iterative methods

### Remark 2.1.1 (Necessity of iterative approximation)

Gaussian elimination ( $\rightarrow$  Section 1.6.2) provides an algorithm that, if carried out in exact arithmetic (no roundoff errors), computes the solution of a linear system of equations with a *finite* number of elementary operations. However, linear systems of equations represent an exceptional case, because it is hardly ever possible to solve general systems of non-linear equations using only finitely many elementary operations.

### (2.1.2) Generic iterations

All methods for general non-lineare systems of equations are *iterative* in the sense that they will usually yield only *approximate solutions* whenever they terminate after finite time.



### (2.1.3) (Stationary) $m$ -point iterative method

All the iterative methods discussed below fall in the class of (stationary)  $m$ -point,  $m \in \mathbb{N}$ , iterative methods, for which the iterate  $\mathbf{x}^{(k)}$  depends on  $F$  and the  $m$  most recent iterates  $\mathbf{x}^{(k-1)}, \dots, \mathbf{x}^{(k-m)}$ , e.g.,

$$\mathbf{x}^{(k)} = \underbrace{\Phi_F(\mathbf{x}^{(k-1)}, \dots, \mathbf{x}^{(k-m)})}_{\text{iteration function for } m\text{-point method}} \quad (2.1.4)$$

Terminology:  $\Phi_F$  is called the **iteration function**.

Note: The **initial guess(es)**  $\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(m-1)} \in \mathbb{R}^n$  have to be provided.

### (2.1.5) Key issues with iterative methods

When applying an iterative method to solve a non-linear system of equations  $F(\mathbf{x}) = 0$ , the following issues arise:

- \* **Convergence**: Does the sequence  $(\mathbf{x}^{(k)})_k$  converge to a limit:  $\lim_{k \rightarrow \infty} \mathbf{x}^{(k)} = \mathbf{x}^*$ ?
- \* **Consistency**: Does the limit, if it exists, provide a solution of the non-linear system of equations:  $F(\mathbf{x}^*) = 0$ ?
- \* **Speed** of convergence: How “fast” does  $\|\mathbf{x}^{(k)} - \mathbf{x}^*\|$  ( $\|\cdot\|$  a suitable norm on  $\mathbb{R}^N$ ) decrease for increasing  $k$ ?

More formal definitions can be given:

#### Definition 2.1.6. Convergence of iterative methods

An iterative method **converges** (for fixed initial guess(es)) : $\Leftrightarrow$   $\mathbf{x}^{(k)} \xrightarrow{k \rightarrow \infty} \mathbf{x}^*$  and  $F(\mathbf{x}^*) = 0$ .

#### Definition 2.1.7. Consistency of iterative methods

A stationary  $m$ -point iterative method is **consistent** with the non-linear system of equations  $F(\mathbf{x}) = 0$

$$\Leftrightarrow \Phi_F(\mathbf{x}^*, \dots, \mathbf{x}^*) = \mathbf{x}^* \Leftrightarrow F(\mathbf{x}^*) = 0$$

For a consistent stationary iterative method we can study the **error** of the iterates  $\mathbf{x}^{(k)}$  defined as:  $\mathbf{e}^{(k)} := \mathbf{x}^{(k)} - \mathbf{x}^*$

Unfortunately, convergence may critically depend on the choice of initial guesses. The property defined next weakens this dependence:

**Definition 2.1.8. Local and global convergence** → [42, Def. 17.1]

As stationary  $m$ -point iterative method converges locally to  $\mathbf{x}^* \in \mathbb{R}^n$ , if there is a neighborhood  $U \subset D$  of  $\mathbf{x}^*$ , such that

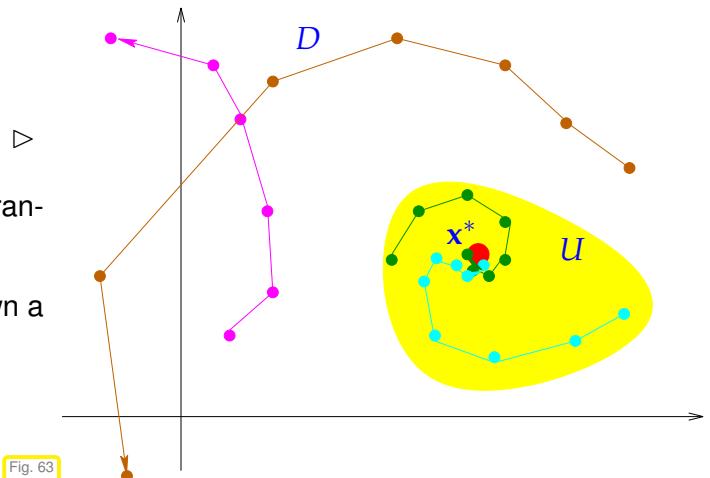
$$\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(m-1)} \in U \Rightarrow \mathbf{x}^{(k)} \text{ well defined} \wedge \lim_{k \rightarrow \infty} \mathbf{x}^{(k)} = \mathbf{x}^*$$

where  $(\mathbf{x}^{(k)})_{k \in \mathbb{N}_0}$  is the (infinite) sequence of iterates.  
If  $U = D$ , the iterative method is globally convergent.

Illustration of local convergence

(Only initial guesses “sufficiently close” to  $\mathbf{x}^*$  guarantee convergence.)

Unfortunately, the neighborhood  $U$  is rarely known a priori. It may also be very small.



Our goal: Given a non-linear system of equations, find iterative methods that converge (locally) to a solution of  $F(\mathbf{x}) = 0$ .

Two general questions: How to measure, describe, and predict the speed of convergence?  
When to terminate the iteration?

### 2.1.1 Speed of convergence

Here and in the sequel,  $\|\cdot\|$  designates a generic vector norm on  $\mathbb{R}^n$ , see Def. 1.5.65. Any occurring matrix norm is induced by this vector norm, see Def. 1.5.71.

It is important to be aware which statements depend on the choice of norm and which do not!

“Speed of convergence”  $\leftrightarrow$  decrease of norm (see Def. 1.5.65) of iteration error

**Definition 2.1.9. Linear convergence**

A sequence  $\mathbf{x}^{(k)}$ ,  $k = 0, 1, 2, \dots$ , in  $\mathbb{R}^n$  converges linearly to  $\mathbf{x}^* \in \mathbb{R}^n$ ,

$$\exists L < 1: \quad \|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq L \|\mathbf{x}^{(k)} - \mathbf{x}^*\| \quad \forall k \in \mathbb{N}_0 .$$

Terminology: least upper bound for  $L$  gives the rate of convergence

**Remark 2.1.10 (Impact of choice of norm)**

<i>Fact of convergence</i> of iteration is	<b>independent</b>	of choice of norm
<i>Fact of linear convergence</i>	<b>depends</b>	on choice of norm
<i>Rate of linear convergence</i>	<b>depends</b>	on choice of norm

The first statement is a consequence of the equivalence of all norms on the finite dimensional vector space  $\mathbb{K}^n$ :

### Definition 2.1.11. Equivalence of norms

Two norms  $\|\cdot\|_a$  and  $\|\cdot\|_b$  on a vector space  $V$  are equivalent if

$$\exists \underline{C}, \bar{C} > 0: \underline{C}\|v\|_a \leq \|v\|_b \leq \bar{C}\|v\|_a \quad \forall v \in V.$$

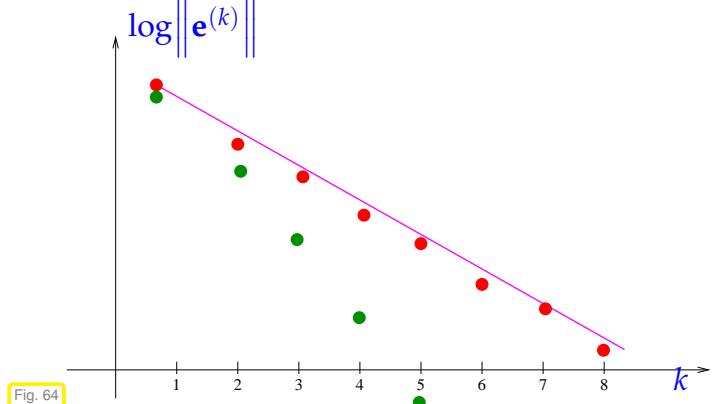
### Theorem 2.1.12. Equivalence of all norms on finite dimensional vector spaces

If  $\dim V < \infty$  all norms ( $\rightarrow$  Def. 1.5.65) on  $V$  are equivalent ( $\rightarrow$  Def. 2.1.11).

### Remark 2.1.13 (Detecting linear convergence)

Often we will study the behavior of a consistent iterative method for a model problem in a numerical experiments and measure the norms of the iteration errors  $\mathbf{e}^{(k)} := \mathbf{x}^{(k)} - \mathbf{x}^*$ . How can we tell that the method enjoys linear convergence?

$$\begin{aligned} &\text{norms of iteration errors} \\ &\uparrow \\ &\sim \text{straight line in lin-log plot} \\ \\ &\left\| \mathbf{e}^{(k)} \right\| \leq L^k \left\| \mathbf{e}^{(0)} \right\|, \\ &\log(\left\| \mathbf{e}^{(k)} \right\|) \leq k \log(L) + \log(\left\| \mathbf{e}^{(0)} \right\|). \end{aligned}$$



Let us abbreviate the error norm in step  $k$  by  $\epsilon_k := \|\mathbf{x}^{(k)} - \mathbf{x}^*\|$ . In the case of linear convergence (see Def. 2.1.9) assume (with  $0 < L < 1$ )

$$\epsilon_{k+1} \approx L\epsilon_k \Rightarrow \log \epsilon_{k+1} \approx \log L + \log \epsilon_k \Rightarrow \log \epsilon_k \approx k \log L + \log \epsilon_0. \quad (2.1.14)$$

We conclude that  $\log L < 0$  determines the slope of the graph in lin-log error chart.

Related: guessing time complexity  $O(n^\alpha)$  of an algorithm from measurements, see § 1.4.8.

Note the green dots ● in Fig. 64: Any “faster” convergence also qualifies as linear convergence in the strict sense of the definition. However, whenever this term is used, we tacitly imply, that no “faster convergence” prevails.

### Example 2.1.15 (Linearly convergent iteration)

Iteration ( $n = 1$ ):

$$x^{(k+1)} = x^{(k)} + \frac{\cos x^{(k)} + 1}{\sin x^{(k)}}.$$

In this MATLAB code  $x$  has to be initialized with the different values for  $x_0$ .

#### MATLABcode 2.1.16: simple fixed point iteration in 1D

```

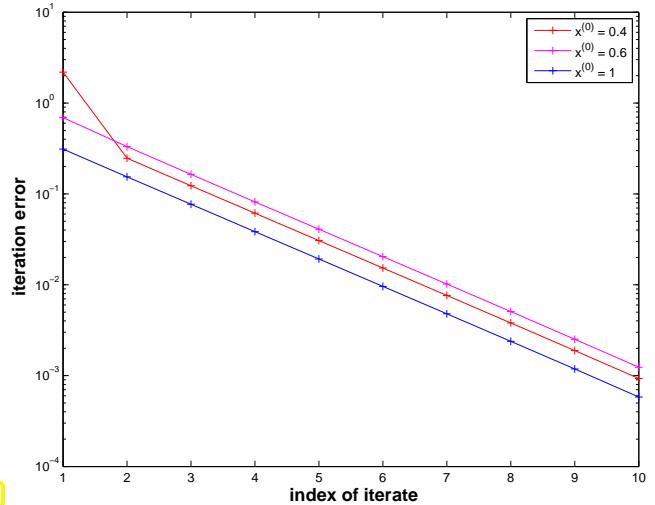
1 y = [ ];
2 for i = 1:15
3     x = x + (cos(x)+1) / sin(x);
4     y = [y, x];
5 end
6 err = y - x;
7 rate = err(2:15) ./ err(1:14);
```

Note: The final iterate  $x^{(15)}$  replaces the exact solution  $x^*$  in the computation of the rate of convergence.

$k$	$x^{(0)} = 0.4$		$x^{(0)} = 0.6$		$x^{(0)} = 1$	
	$x^{(k)}$	$\frac{ x^{(k)} - x^{(15)} }{ x^{(k-1)} - x^{(15)} }$	$x^{(k)}$	$\frac{ x^{(k)} - x^{(15)} }{ x^{(k-1)} - x^{(15)} }$	$x^{(k)}$	$\frac{ x^{(k)} - x^{(15)} }{ x^{(k-1)} - x^{(15)} }$
2	3.3887	0.1128	3.4727	0.4791	2.9873	0.4959
3	3.2645	0.4974	3.3056	0.4953	3.0646	0.4989
4	3.2030	0.4992	3.2234	0.4988	3.1031	0.4996
5	3.1723	0.4996	3.1825	0.4995	3.1224	0.4997
6	3.1569	0.4995	3.1620	0.4994	3.1320	0.4995
7	3.1493	0.4990	3.1518	0.4990	3.1368	0.4990
8	3.1454	0.4980	3.1467	0.4980	3.1392	0.4980



Rate of convergence  $\approx 0.5$



Linear convergence as in Def. 2.1.9



error graphs = straight lines in lin-log scale  
 $\rightarrow$  Rem. 2.1.13

There are notions of convergence that guarantee a much faster (asymptotic) decay of norm of the iteration error than linear convergence from Def. 2.1.9.

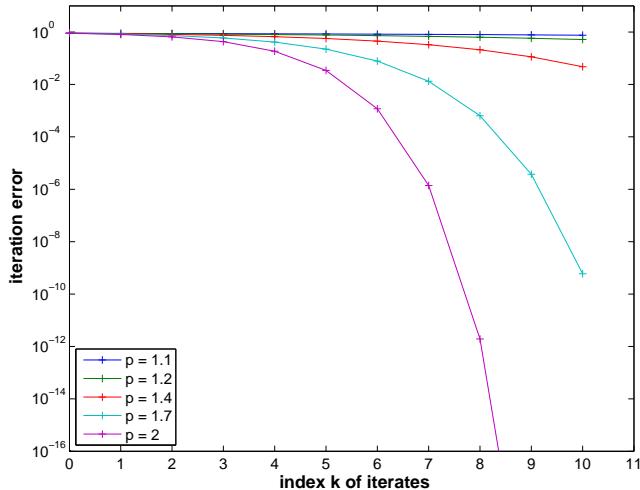
**Definition 2.1.17. Order of convergence** → [42, Sect. 17.2], [15, Def. 5.14], [63, Def. 6.1]

A convergent sequence  $\mathbf{x}^{(k)}$ ,  $k = 0, 1, 2, \dots$ , in  $\mathbb{R}^n$  with limit  $\mathbf{x}^* \in \mathbb{R}^n$  converges with **order  $p$** , if

$$\exists C > 0: \|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq C \|\mathbf{x}^{(k)} - \mathbf{x}^*\|^p \quad \forall k \in \mathbb{N}_0 , \quad (2.1.18)$$

and, in addition,  $C < 1$  in the case  $p = 1$  (linear convergence → Def. 2.1.9).

Of course, the order  $p$  of convergence of an iterative method refers to the largest possible  $p$  in the definition, that is, the error estimate will in general not hold, if  $p$  is replaced with  $p + \epsilon$  for any  $\epsilon > 0$ , cf. Rem. 1.4.5.



▷ Qualitative error graphs for convergence of order  $p$   
(lin-log scale)

In the case of convergence of order  $p$  ( $p > 1$ ) (see Def. 2.1.17):

$$\begin{aligned} \epsilon_{k+1} \approx C\epsilon_k^p &\Rightarrow \log \epsilon_{k+1} = \log C + p \log \epsilon_k \Rightarrow \log \epsilon_{k+1} = \log C \sum_{l=0}^k p^l + p^{k+1} \log \epsilon_0 \\ &\Rightarrow \log \epsilon_{k+1} = -\frac{\log C}{p-1} + \left( \frac{\log C}{p-1} + \log \epsilon_0 \right) p^{k+1}. \end{aligned}$$

In this case, the error graph is a concave power curve (for sufficiently small  $\epsilon_0$  !)

**Remark 2.1.19 (Detecting order of convergence)**

How to guess the order of convergence (→ Def. 2.1.17) from tabulated error norms measured in a numerical experiment?

Abbreviate  $\epsilon_k := \|\mathbf{x}^{(k)} - \mathbf{x}^*\|$  (norm of iteration error):

$$\text{assume } \epsilon_{k+1} \approx C\epsilon_k^p \Rightarrow \log \epsilon_{k+1} \approx \log C + p \log \epsilon_k \Rightarrow \frac{\log \epsilon_{k+1} - \log \epsilon_k}{\log \epsilon_k - \log \epsilon_{k-1}} \approx p.$$

➤ monitor the quotients  $(\log \epsilon_{k+1} - \log \epsilon_k) / (\log \epsilon_k - \log \epsilon_{k-1})$  over several steps of the iteration.

**Example 2.1.20 (quadratic convergence = convergence of order 2)**

From your analysis course [77, Bsp. 3.3.2(iii)] recall the famous iteration for computing  $\sqrt{a}$ ,  $a > 0$ :

$$x^{(k+1)} = \frac{1}{2}(x^{(k)} + \frac{a}{x^{(k)}}) \Rightarrow |x^{(k+1)} - \sqrt{a}| = \frac{1}{2x^{(k)}}|x^{(k)} - \sqrt{a}|^2. \quad (2.1.21)$$

By the arithmetic-geometric mean inequality (AGM)  $\sqrt{ab} \leq \frac{1}{2}(a+b)$  we conclude:  $x^{(k)} > \sqrt{a}$  for  $k \geq 1$ . Therefore estimate from (2.1.21) means that the sequence from (2.1.21) converges with order 2 to  $\sqrt{a}$ .

Note:  $x^{(k+1)} < x^{(k)}$  for all  $k \geq 2$   $\Rightarrow (x^{(k)})_{k \in \mathbb{N}_0}$  converges as a decreasing sequence that is bounded from below ( $\rightarrow$  analysis course)

Numerical experiment: iterates for  $a = 2$ :

$k$	$x^{(k)}$	$e^{(k)} := x^{(k)} - \sqrt{2}$	$\log \frac{ e^{(k)} }{ e^{(k-1)} } : \log \frac{ e^{(k-1)} }{ e^{(k-2)} }$
0	2.0000000000000000000	0.58578643762690485	
1	1.5000000000000000000	0.08578643762690485	
2	1.4166666666666652	0.00245310429357137	1.850
3	1.41421568627450966	0.00000212390141452	1.984
4	1.41421356237468987	0.000000000000159472	2.000
5	1.41421356237309492	0.00000000000000022	0.630

Note the **doubling** of the number of significant digits in each step !

[impact of roundoff !]

The doubling of the number of significant digits for the iterates holds true for any quadratically convergent iteration:

Recall from Rem. 1.5.24 that the relative error ( $\rightarrow$  Def. 1.5.23) tells the number of significant digits. Indeed, denoting the relative error in step  $k$  by  $\delta_k$ , we have in the case of quadratic convergence.

$$\begin{aligned} x^{(k)} &= x^*(1 + \delta_k) \Rightarrow x^{(k)} - x^* = \delta_k x^*. \\ \Rightarrow |x^* \delta_{k+1}| &= |x^{(k+1)} - x^*| \leq C|x^{(k)} - x^*|^2 = C|x^* \delta_k|^2 \\ \Rightarrow |\delta_{k+1}| &\leq C|x^*|^2 \delta_k^2. \end{aligned} \quad (2.1.22)$$

Note:  $\delta_k \approx 10^{-\ell}$  means that  $x^{(k)}$  has  $\ell$  significant digits.

Also note that if  $C \approx 1$ , then  $\delta_k = 10^{-\ell}$  and (2.1.20) implies  $\delta_{k+1} \approx 10^{-2\ell}$ .

## 2.1.2 Termination criteria



*Supplementary reading.* Also discussed in [6, Sect. 3.1, p. 42].

As remarked above, usually (even without roundoff errors) an iteration will never arrive at an/the exact solution  $x^*$  after finitely many steps. Thus, we can only hope to compute an *approximate* solution by accepting  $x^{(K)}$  as result for some  $K \in \mathbb{N}_0$ . Termination criteria (stopping rules) are used to determine a suitable value for  $K$ .

For the sake of efficiency  $\Rightarrow$  stop iteration when iteration error is just “small enough” (“small enough” depends on the concrete problem and user demands.)

### (2.1.23) Classification of termination criteria (stopping rules) for iterative solvers for non-linear systems of equations

A **termination criterion** (stopping rule) is an algorithm deciding in each step of an iterative method whether to **STOP** or to **CONTINUE**.

<b>A priori</b> termination	<b>A posteriori</b> termination
Decision to stop based on information about $F$ and $\mathbf{x}^{(0)}$ , made before starting iteration.	Beside $\mathbf{x}^{(0)}$ and $F$ , also current and past iterates are used to decide about termination.

A termination criterion for a convergent iteration is deemed **reliable**, if it lets the iteration **CONTINUE**, until the iteration error  $\mathbf{e}^{(k)} := \mathbf{x}^{(k)} - \mathbf{x}^*$ ,  $\mathbf{x}^*$  the limit value, satisfies certain conditions (usually imposed before the start of the iteration).

### (2.1.24) Ideal termination

Termination criteria are usually meant to ensure accuracy of the final iterate  $\mathbf{x}^{(K)}$  in the following sense:

$$\begin{aligned} \|\mathbf{x}^{(K)} - \mathbf{x}^*\| &\leq \tau_{\text{abs}}, \quad \tau_{\text{abs}} \triangleq \text{prescribed (absolute) tolerance.} \\ &\text{or} \\ \|\mathbf{x}^{(K)} - \mathbf{x}^*\| &\leq \tau_{\text{rel}} \|\mathbf{x}^*\|, \quad \tau_{\text{rel}} \triangleq \text{prescribed (relative) tolerance.} \end{aligned}$$

it seems that the second criterion, asking that the **relative** ( $\rightarrow$  Def. 1.5.23) iteration error be below a prescribed threshold, alone would suffice, but the absolute tolerance should be checked, if, by “accident”,  $\|\mathbf{x}^*\| = 0$  is possible. Otherwise, the iteration might fail to terminate at all.

Both criteria enter the “ideal (a posteriori) termination rule”:

$$\text{STOP at step } K = \operatorname{argmin}_{k \in \mathbb{N}_0} \|\mathbf{x}^{(k)} - \mathbf{x}^*\| \leq \begin{cases} \tau_{\text{abs}} \\ \text{or} \\ \tau_{\text{rel}} \|\mathbf{x}^*\| \end{cases}. \quad (2.1.25)$$

Obviously, (2.1.25) achieves the optimum in terms of efficiency and reliability. As obviously, this termination criterion is not practical, because  $\mathbf{x}^*$  is not known.

### (2.1.26) Practical termination criteria for iterations

The following termination criteria are commonly used in numerical codes:

- ① **A priori termination:** stop iteration after fixed number of steps (possibly depending on  $\mathbf{x}^{(0)}$ ).



Drawback: hard to ensure prescribed accuracy!

(A priori  $\hat{=}$  without actually taking into account the computed iterates, see § 2.1.23)

Invoking additional properties of either the non-linear system of equations  $F(\mathbf{x}) = \mathbf{0}$  or the iteration it is sometimes possible to tell that for sure  $\|\mathbf{x}^{(k)} - \mathbf{x}^*\| \leq \tau$  for all  $k \geq K$ , though this  $K$  may be (significantly) larger than the optimal termination index from (2.1.25), see Rem. 2.1.28.

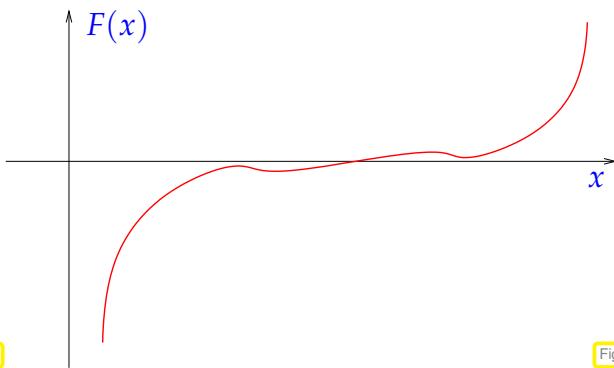
- ② **Residual based** termination: STOP convergent iteration  $\{\mathbf{x}^{(k)}\}_{k \in \mathbb{N}_0}$ , when

$$\|F(\mathbf{x}^{(k)})\| \leq \tau, \quad \tau \hat{=} \text{prescribed tolerance} > 0.$$

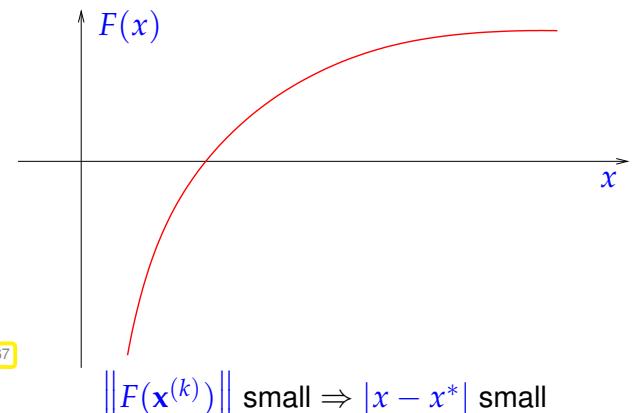


no guaranteed accuracy

Consider the case  $n = 1$ . If  $F : D \subset \mathbb{R} \rightarrow \mathbb{R}$  is “flat” in the neighborhood of a zero  $x^*$ , then a small value of  $|F(x)|$  does not mean that  $x$  is close to  $x^*$ .



$\|F(\mathbf{x}^{(k)})\| \text{ small } \not\Rightarrow |x - x^*| \text{ small}$



$\|F(\mathbf{x}^{(k)})\| \text{ small } \Rightarrow |x - x^*| \text{ small}$

- ③ **Correction based** termination: STOP convergent iteration  $\{\mathbf{x}^{(k)}\}_{k \in \mathbb{N}_0}$ , when

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| \leq \begin{cases} \frac{\tau_{\text{abs}}}{\tau_{\text{rel}} \|\mathbf{x}^{(k+1)}\|}, & \tau_{\text{abs}} \text{ prescribed} \\ \text{or} \\ \tau_{\text{rel}} & \tau_{\text{rel}} \text{ relative tolerances} > 0. \end{cases}$$

Also for this criterion, we have no guarantee that (2.1.25) will be satisfied only remotely.

A special variant of correction based termination exploits that  $\mathbb{M}$  is finite! ( $\rightarrow$  Section 1.5.3)

Wait until (convergent) iteration becomes stationary in the discrete set  $\mathbb{M}$  of machine numbers!

possibly grossly inefficient!  
(always computes “up to machine precision”)

**MATLAB-code 2.1.27: Square root iteration**  
 $\rightarrow$  Ex. 2.1.20

```

1 function x = sqrtit(a)
2 x_old = -1; x = a;
3 while (x_old ~= x)
4     x_old = x;
5     x = 0.5 * (x+a/x);
6 end

```

**Remark 2.1.28 (A posteriori termination criterion for linearly convergent iterations**  $\rightarrow$  [15, Lemma 5.17, 5.19])

Let us assume that we know that an iteration linearly convergent ( $\rightarrow$  Def. 2.1.9) with rate of convergence  $0 < L < 1$ :

The following simple manipulations give an a posteriori termination criterion (for linearly convergent iterations with rate of convergence  $0 < L < 1$ ):

$$\|\mathbf{x}^{(k)} - \mathbf{x}^*\| \stackrel{\triangle\text{-inequ.}}{\leq} \|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| + \|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq \|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| + L\|\mathbf{x}^{(k)} - \mathbf{x}^*\|.$$

► Iterates satisfy: 
$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq \frac{L}{1-L}\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|. \quad (2.1.29)$$

This suggests that we take the right hand side of (2.1.29) as a posteriori error bound and use it instead of the inaccessible  $\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\|$  for checking absolute and relative accuracy in (2.1.25). The resulting termination criterium will be **reliable** ( $\rightarrow$  § 2.1.23), since we will certainly have achieved the desired accuracy when we stop the iteration.

 Estimating the rate of convergence  $L$  might be difficult.

 Pessimistic estimate for  $L$  will not compromise reliability.

(Using  $\tilde{L} > L$  in (2.1.29) still yields a valid *upper bound* for  $\|\mathbf{x}^{(k)} - \mathbf{x}^*\|$ .)

### Example 2.1.30 (A posteriori error bound for linearly convergent iteration)

We revisit the iteration of Ex. 2.1.15:

$$x^{(k+1)} = x^{(k)} + \frac{\cos x^{(k)} + 1}{\sin x^{(k)}} \Rightarrow x^{(k)} \rightarrow \pi \quad \text{for } x^{(0)} \text{ close to } \pi.$$

Observed rate of convergence:  $L = 1/2$

Error and error bound for  $x^{(0)} = 0.4$ :

$k$	$ x^{(k)} - \pi $	$\frac{L}{1-L} x^{(k)} - x^{(k-1)} $	slack of bound
1	2.191562221997101	4.933154875586894	2.741592653589793
2	0.247139097781070	1.944423124216031	1.697284026434961
3	0.122936737876834	0.124202359904236	0.001265622027401
4	0.061390835206217	0.061545902670618	0.000155067464401
5	0.030685773472263	0.030705061733954	0.000019288261691
6	0.015341682696235	0.015344090776028	0.000002408079792
7	0.007670690889185	0.007670991807050	0.000000300917864
8	0.003835326638666	0.003835364250520	0.000000037611854
9	0.001917660968637	0.001917665670029	0.000000004701392
10	0.000958830190489	0.000958830778147	0.000000000587658
11	0.000479415058549	0.000479415131941	0.000000000073392
12	0.000239707524646	0.000239707533903	0.000000000009257
13	0.000119853761949	0.000119853762696	0.000000000000747
14	0.000059926881308	0.000059926880641	0.000000000000667
15	0.000029963440745	0.000029963440563	0.000000000000181

Hence: the a posteriori error bound is highly accurate in this case!

## 2.2 Fixed Point Iterations



*Supplementary reading.* The contents of this section are also treated in [15, Sect. 5.3], [63, Sect. 6.3], [6, Sect. 3.3]

As before we consider a non-linear system of equations  $F(\mathbf{x}) = 0$ ,  $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$ .

1-point stationary iterative methods, see (2.1.4), for  $F(\mathbf{x}) = 0$  are also called **fixed point iterations**.

► A fixed point iteration is defined by **iteration function**  $\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$ :

$$\begin{array}{l} \text{iteration function} \quad \Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n \\ \text{initial guess} \quad \mathbf{x}^{(0)} \in U \end{array} \Rightarrow \boxed{\text{iterates } (\mathbf{x}^{(k)})_{k \in \mathbb{N}_0}: \quad \mathbf{x}^{(k+1)} := \Phi(\mathbf{x}^{(k)})} .$$

$\underbrace{\quad\quad\quad}_{\rightarrow \text{1-point method, cf. (2.1.4)}}$

Here,  $U$  designates the domain of definition of the iteration function  $\Phi$ .

Note that the sequence of iterates need not be well defined:  $\mathbf{x}^{(k)} \notin U$  possible !

### 2.2.1 Consistent fixed point iterations

Next, we specialize Def. 2.1.7 for fixed point iterations:

#### Definition 2.2.1. Consistency of fixed point iterations, c.f. Def. 2.1.7

A fixed point iteration  $\mathbf{x}^{(k+1)} = \Phi(\mathbf{x}^{(k)})$  is **consistent** with  $F(\mathbf{x}) = 0$ , if, for  $\mathbf{x} \in U \cap D$ ,

$$F(\mathbf{x}) = 0 \Leftrightarrow \Phi(\mathbf{x}) = \mathbf{x} .$$

Note: iteration function  
 $\Phi$  continuous      and      fixed point iteration (locally)  
convergent to  $\mathbf{x}^* \in U$        $\Rightarrow$        $\mathbf{x}^*$  is a  
fixed point of  $\Phi$ .

This is an immediate consequence that for a continuous function limits and function evaluations commute [77, Sect. 4.1].

General construction of fixed point iterations that is consistent with  $F(\mathbf{x}) = 0$ :

- ① Rewrite equivalently  $F(\mathbf{x}) = 0 \Leftrightarrow \Phi(\mathbf{x}) = \mathbf{x}$  and then
- ② use the fixed point iteration

$$\mathbf{x}^{(k+1)} := \Phi(\mathbf{x}^{(k)}) . \quad (2.2.2)$$

Note: there are *many ways* to transform  $F(\mathbf{x}) = 0$  into a fixed point form !

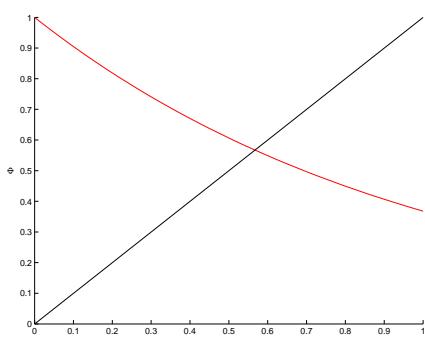
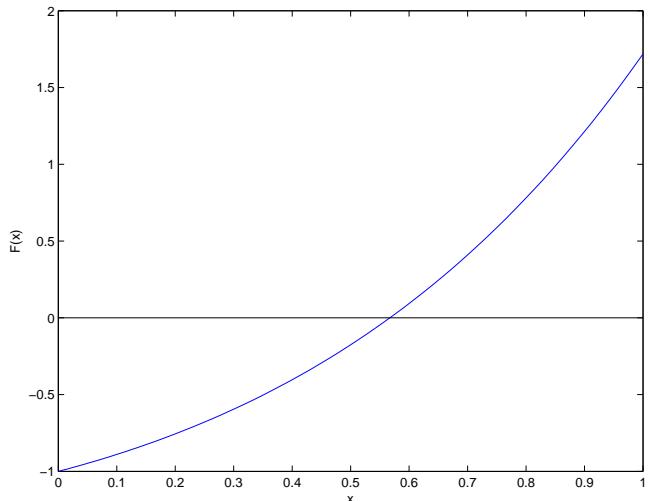
### Experiment 2.2.3 (Many choices for consistent fixed point iterations)

In this example we construct three different consistent fixed point iteration for a single scalar ( $n = 1$ ) non-linear equation  $F(x) = 0$ . In numerical experiments we will see that they behave very differently.

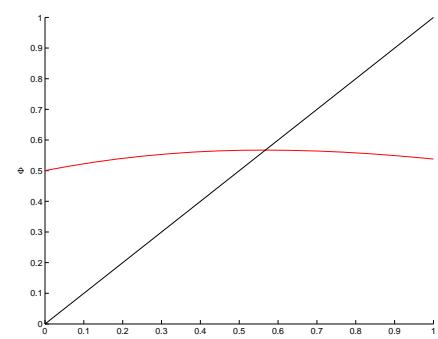
$$F(x) = xe^x - 1, \quad x \in [0, 1].$$

Different fixed point forms:

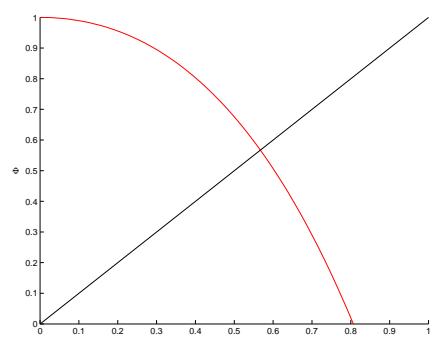
$$\begin{aligned}\Phi_1(x) &= e^{-x}, \\ \Phi_2(x) &= \frac{1+x}{1+e^x}, \\ \Phi_3(x) &= x + 1 - xe^x.\end{aligned}$$



function  $\Phi_1$



function  $\Phi_2$



function  $\Phi_3$

With the same initial guess  $x^{(0)} = 0.5$  for all three fixed point iterations we obtain the following iterates:

$k$	$x^{(k+1)} := \Phi_1(x^{(k)})$	$x^{(k+1)} := \Phi_2(x^{(k)})$	$x^{(k+1)} := \Phi_3(x^{(k)})$
0	0.5000000000000000	0.5000000000000000	0.5000000000000000
1	0.606530659712633	0.566311003197218	0.675639364649936
2	0.545239211892605	0.567143165034862	0.347812678511202
3	0.579703094878068	0.567143290409781	0.855321409174107
4	0.560064627938902	0.567143290409784	-0.156505955383169
5	0.571172148977215	0.567143290409784	0.977326422747719
6	0.564862946980323	0.567143290409784	-0.619764251895580
7	0.568438047570066	0.567143290409784	0.713713087416146
8	0.566409452746921	0.567143290409784	0.256626649129847
9	0.567559634262242	0.567143290409784	0.924920676910549
10	0.566907212935471	0.567143290409784	-0.407422405542253

We can also tabulate the modulus of the iteration error and mark correct digits with red:

$k$	$ x_1^{(k+1)} - x^* $	$ x_2^{(k+1)} - x^* $	$ x_3^{(k+1)} - x^* $
0	0.067143290409784	0.067143290409784	0.067143290409784
1	0.039387369302849	0.000832287212566	0.108496074240152
2	0.021904078517179	0.000000125374922	0.219330611898582
3	0.012559804468284	0.000000000000003	0.288178118764323
4	0.007078662470882	0.000000000000000	0.723649245792953
5	0.004028858567431	0.000000000000000	0.410183132337935
6	0.002280343429460	0.000000000000000	1.186907542305364
7	0.001294757160282	0.000000000000000	0.146569797006362
8	0.000733837662863	0.000000000000000	0.310516641279937
9	0.000416343852458	0.000000000000000	0.357777386500765
10	0.000236077474313	0.000000000000000	0.974565695952037

Observed: linear convergence of  $x_1^{(k)}$ , quadratic convergence of  $x_2^{(k)}$ ,  
no convergence (erratic behavior of  $x_3^{(k)}$ ) ( $x_i^{(0)} = 0.5$  in all cases).

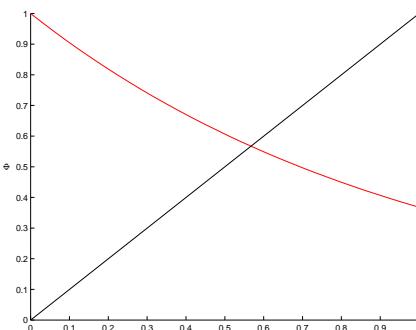
Question: Can we explain/forecast the behaviour of a fixed point iteration?

## 2.2.2 Convergence of fixed point iterations

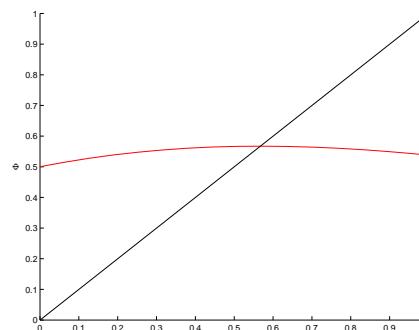
In this section we will try to find easily verifiable conditions that ensure convergence (of a certain order) of fixed point iterations. It will turn out that these conditions are surprisingly simple and general.

### Experiment 2.2.4 (Exp. 2.2.3 revisited)

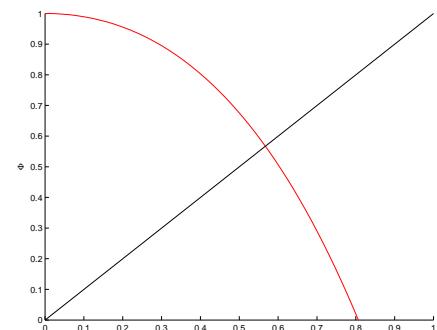
In Exp. 2.2.3 we observed vastly different behavior of different fixed point iterations for  $n = 1$ . Is it possible to predict this from the shape of the graph of the iteration functions?



$\Phi_1$ : linear convergence ?



$\Phi_2$ : quadratic convergence ?



function  $\Phi_3$ : no convergence

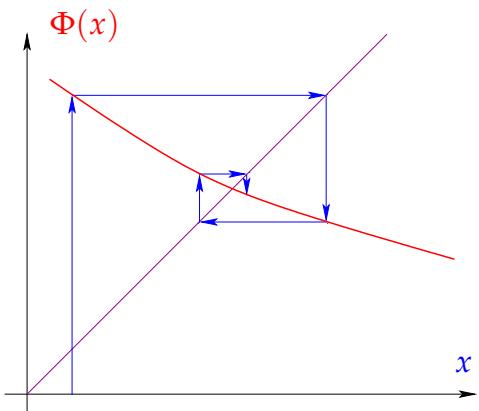
### Example 2.2.5 (Fixed point iteration in 1D)

1D setting ( $n = 1$ ):  $\Phi : \mathbb{R} \mapsto \mathbb{R}$  continuously differentiable,  $\Phi(x^*) = x^*$

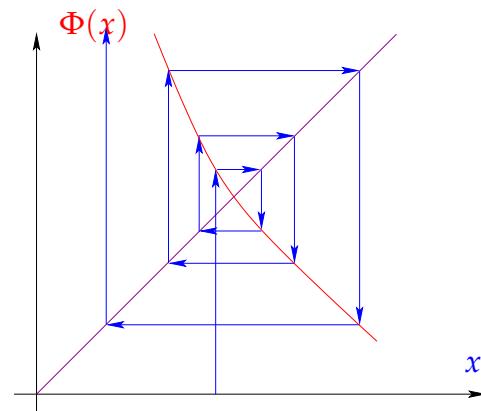
fixed point iteration:  $x^{(k+1)} = \Phi(x^{(k)})$

In 1D it is possible to visualize the different convergence behavior of fixed point iterations: In order to construct  $x^{(k+1)}$  from  $x^{(k)}$  one moves vertically to  $(x^{(k)}, x^{(k+1)} = \Phi(x^{(k)}))$ , then horizontally to the

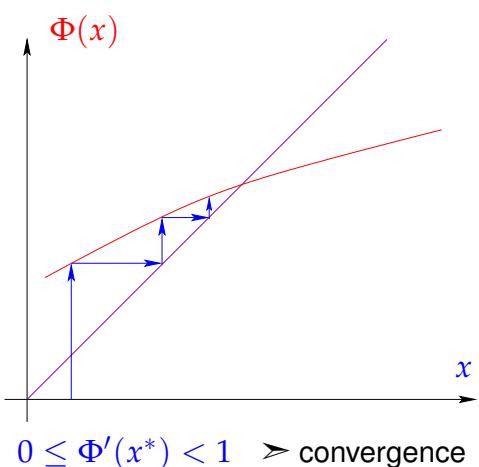
angular bisector of the first/third quadrant, that is, to the point  $(x^{(k+1)}, x^{(k+1)})$ . Returning vertically to the abscissa gives  $x^{(k+1)}$ .



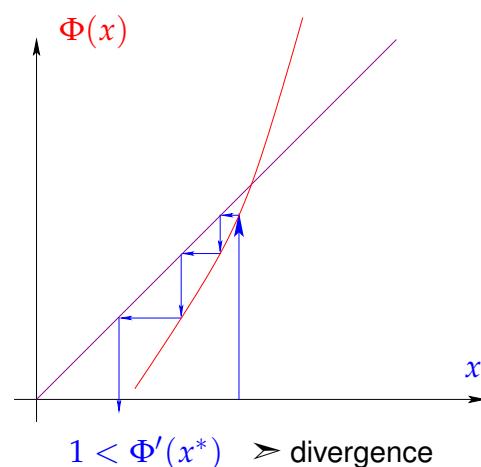
$-1 < \Phi'(x^*) \leq 0 \Rightarrow$  convergence



$\Phi'(x^*) < -1 \Rightarrow$  divergence



$0 \leq \Phi'(x^*) < 1 \Rightarrow$  convergence



$1 < \Phi'(x^*) \Rightarrow$  divergence

Numerical examples for iteration functions  $\Rightarrow$  Exp. 2.2.3, iteration functions  $\Phi_1$  and  $\Phi_3$

It seems that the slope of the iteration function  $\Phi$  in the fixed point, that is, in the point where it intersects the bisector of the first/third quadrant, is crucial.

Now we investigate rigorously, when a fixed point iteration will lead to a convergent iteration with a particular qualitative kind of convergence according to Def. 2.1.17.

#### Definition 2.2.6. Contractive mapping

$\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$  is contractive (w.r.t. norm  $\|\cdot\|$  on  $\mathbb{R}^n$ ), if

$$\exists L < 1: \|\Phi(\mathbf{x}) - \Phi(\mathbf{y})\| \leq L \|\mathbf{x} - \mathbf{y}\| \quad \forall \mathbf{x}, \mathbf{y} \in U. \quad (2.2.7)$$

A simple consideration: if  $\Phi(\mathbf{x}^*) = \mathbf{x}^*$  (fixed point), then a fixed point iteration induced by a contractive mapping  $\Phi$  satisfies

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| = \|\Phi(\mathbf{x}^{(k)}) - \Phi(\mathbf{x}^*)\| \stackrel{(2.2.7)}{\leq} L \|\mathbf{x}^{(k)} - \mathbf{x}^*\|,$$

that is, the iteration **converges** (at least) **linearly** ( $\rightarrow$  Def. 2.1.9).

Note that

$\Phi$  contractive  $\Rightarrow$   $\Phi$  has **at most one** fixed point.

**Remark 2.2.8 (Banach's fixed point theorem)** → [77, Satz 6.5.2], [15, Satz 5.8])

A key theorem in calculus (also functional analysis):

**Theorem 2.2.9. Banach's fixed point theorem**

If  $D \subset \mathbb{K}^n$  ( $\mathbb{K} = \mathbb{R}, \mathbb{C}$ ) closed and bounded and  $\Phi : D \mapsto D$  satisfies

$$\exists L < 1: \|\Phi(\mathbf{x}) - \Phi(\mathbf{y})\| \leq L\|\mathbf{x} - \mathbf{y}\| \quad \forall \mathbf{x}, \mathbf{y} \in D,$$

then there is a unique fixed point  $\mathbf{x}^* \in D$ ,  $\Phi(\mathbf{x}^*) = \mathbf{x}^*$ , which is the limit of the sequence of iterates  $\mathbf{x}^{(k+1)} := \Phi(\mathbf{x}^{(k)})$  for any  $\mathbf{x}^{(0)} \in D$ .

*Proof.* Proof based on 1-point iteration  $\mathbf{x}^{(k)} = \Phi(\mathbf{x}^{(k-1)})$ ,  $\mathbf{x}^{(0)} \in D$ :

$$\begin{aligned} \|\mathbf{x}^{(k+N)} - \mathbf{x}^{(k)}\| &\leq \sum_{j=k}^{k+N-1} \|\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)}\| \leq \sum_{j=k}^{k+N-1} L^j \|\mathbf{x}^{(1)} - \mathbf{x}^{(0)}\| \\ &\leq \frac{L^k}{1-L} \|\mathbf{x}^{(1)} - \mathbf{x}^{(0)}\| \xrightarrow{k \rightarrow \infty} 0. \end{aligned}$$

$(\mathbf{x}^{(k)})_{k \in \mathbb{N}_0}$  Cauchy sequence ➤ convergent  $\mathbf{x}^{(k)} \xrightarrow{k \rightarrow \infty} \mathbf{x}^*$ .  
Continuity of  $\Phi$  ➤  $\Phi(\mathbf{x}^*) = \mathbf{x}^*$ . Uniqueness of fixed point is evident.

□

A simple criterion for a differentiable  $\Phi$  to be contractive:

**Lemma 2.2.10. Sufficient condition for local linear convergence of fixed point iteration** → [42, Thm. 17.2], [15, Cor. 5.12]

If  $\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$ ,  $\Phi(\mathbf{x}^*) = \mathbf{x}^*$ ,  $\Phi$  differentiable in  $\mathbf{x}^*$ , and  $\|\mathbf{D}\Phi(\mathbf{x}^*)\| < 1$ , then the fixed point iteration

$$\mathbf{x}^{(k+1)} := \Phi(\mathbf{x}^{(k)}), \tag{2.2.2}$$

converges locally and at least linearly.

matrix norm, Def. 1.5.71 !



notation:  $\mathbf{D}\Phi(\mathbf{x}) \triangleq$  **Jacobian** (ger.: Jacobi-Matrix) of  $\Phi$  at  $\mathbf{x} \in D$  → [77, Sect. 7.6]

$$\mathbf{D}\Phi(\mathbf{x}) = \left[ \frac{\partial \Phi_i}{\partial x_j}(\mathbf{x}) \right]_{i,j=1}^n = \begin{bmatrix} \frac{\partial \Phi_1}{\partial x_1}(\mathbf{x}) & \frac{\partial \Phi_1}{\partial x_2}(\mathbf{x}) & \cdots & \cdots & \frac{\partial \Phi_1}{\partial x_n}(\mathbf{x}) \\ \frac{\partial \Phi_2}{\partial x_1}(\mathbf{x}) & & & & \frac{\partial \Phi_2}{\partial x_n}(\mathbf{x}) \\ \vdots & & & & \vdots \\ \frac{\partial \Phi_n}{\partial x_1}(\mathbf{x}) & \frac{\partial \Phi_n}{\partial x_2}(\mathbf{x}) & \cdots & \cdots & \frac{\partial \Phi_n}{\partial x_n}(\mathbf{x}) \end{bmatrix}. \tag{2.2.11}$$

“Visualization” of the statement of Lemma 2.2.10 in Ex. 2.2.5: The iteration converges *locally*, if  $\Phi$  is flat in a neighborhood of  $\mathbf{x}^*$ , it will diverge, if  $\Phi$  is steep there.

*Proof.* (of Lemma 2.2.10) By definition of derivative

$$\|\Phi(\mathbf{y}) - \Phi(\mathbf{x}^*) - D\Phi(\mathbf{x}^*)(\mathbf{y} - \mathbf{x}^*)\| \leq \psi(\|\mathbf{y} - \mathbf{x}^*\|)\|\mathbf{y} - \mathbf{x}^*\|,$$

with  $\psi : \mathbb{R}_0^+ \mapsto \mathbb{R}_0^+$  satisfying  $\lim_{t \rightarrow 0} \psi(t) = 0$ .

Choose  $\delta > 0$  such that

$$L := \psi(t) + \|D\Phi(\mathbf{x}^*)\| \leq \frac{1}{2}(1 + \|D\Phi(\mathbf{x}^*)\|) < 1 \quad \forall 0 \leq t < \delta.$$

By inverse triangle inequality we obtain for fixed point iteration

$$\begin{aligned} \|\Phi(\mathbf{x}) - \mathbf{x}^*\| - \|D\Phi(\mathbf{x}^*)(\mathbf{x} - \mathbf{x}^*)\| &\leq \psi(\|\mathbf{x} - \mathbf{x}^*\|)\|\mathbf{x} - \mathbf{x}^*\| \\ \Rightarrow \|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| &\leq (\psi(t) + \|D\Phi(\mathbf{x}^*)\|)\|\mathbf{x}^{(k)} - \mathbf{x}^*\| \leq L\|\mathbf{x}^{(k)} - \mathbf{x}^*\|, \end{aligned}$$

if  $\|\mathbf{x}^{(k)} - \mathbf{x}^*\| < \delta$ . □

### Lemma 2.2.12. Sufficient condition for linear convergence of fixed point iteration

Let  $U$  be convex and  $\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$  be continuously differentiable with

$$L := \sup_{\mathbf{x} \in U} \|D\Phi(\mathbf{x})\| < 1.$$

If  $\Phi(\mathbf{x}^*) = \mathbf{x}^*$  for some interior point  $\mathbf{x}^* \in U$ , then the fixed point iteration  $\mathbf{x}^{(k+1)} = \Phi(\mathbf{x}^{(k)})$  converges to  $\mathbf{x}^*$  at least linearly with rate  $L$ .

Recall:  $U \subset \mathbb{R}^n$  convex  $\Leftrightarrow (t\mathbf{x} + (1-t)\mathbf{y}) \in U$  for all  $\mathbf{x}, \mathbf{y} \in U, 0 \leq t \leq 1$

*Proof.* (of Lemma 2.2.12) By the mean value theorem

$$\begin{aligned} \Phi(\mathbf{x}) - \Phi(\mathbf{y}) &= \int_0^1 D\Phi(\mathbf{x} + \tau(\mathbf{y} - \mathbf{x}))(\mathbf{y} - \mathbf{x}) d\tau \quad \forall \mathbf{x}, \mathbf{y} \in \text{dom}(\Phi). \\ \Rightarrow \quad \|\Phi(\mathbf{x}) - \Phi(\mathbf{y})\| &\leq L\|\mathbf{y} - \mathbf{x}\|, \\ \Rightarrow \quad \|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| &\leq L\|\mathbf{x}^{(k)} - \mathbf{x}^*\|. \end{aligned}$$

We find that  $\Phi$  is contractive on  $U$  with unique fixed point  $\mathbf{x}^*$ , to which  $\mathbf{x}^{(k)}$  converges linearly for  $k \rightarrow \infty$ . □

### Remark 2.2.13 (Bound for asymptotic rate of linear convergence)

By asymptotic rate of a linearly converging iteration we mean the contraction factor for the norm of the iteration error that we can expect, when we are already very close to the limit  $\mathbf{x}^*$ .

If  $0 < \|D\Phi(\mathbf{x}^*)\| < 1$ ,  $\mathbf{x}^{(k)} \approx \mathbf{x}^*$  then the (worst) asymptotic rate of linear convergence is  $L = \|D\Phi(\mathbf{x}^*)\|$

### Example 2.2.14 (Multidimensional fixed point iteration)

In this example we encounter the first genuine *system* of non-linear equations and apply Lemma 2.2.12 to it.

$$\begin{array}{l} \text{System of equations} \quad \text{in} \quad \text{fixed point form:} \\ \left\{ \begin{array}{l} x_1 - c(\cos x_1 - \sin x_2) = 0 \\ (x_1 - x_2) - c \sin x_2 = 0 \end{array} \right. \Rightarrow \left\{ \begin{array}{l} c(\cos x_1 - \sin x_2) = x_1 \\ c(\cos x_1 - 2 \sin x_2) = x_2 \end{array} \right. \\ \text{Define: } \Phi \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = c \begin{bmatrix} \cos x_1 - \sin x_2 \\ \cos x_1 - 2 \sin x_2 \end{bmatrix} \Rightarrow D\Phi \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = -c \begin{bmatrix} \sin x_1 & \cos x_2 \\ \sin x_1 & 2 \cos x_2 \end{bmatrix}. \end{array}$$

Choose *appropriate* norm:  $\|\cdot\| = \infty$ -norm  $\|\cdot\|_\infty$  ( $\rightarrow$  Example 1.5.73);

$$\text{if } c < \frac{1}{3} \Rightarrow \|D\Phi(\mathbf{x})\|_\infty < 1 \quad \forall \mathbf{x} \in \mathbb{R}^2,$$

$\Rightarrow$  (at least) linear convergence of the fixed point iteration. The existence of a fixed point is also guaranteed, because  $\Phi$  maps into the closed set  $[-3, 3]^2$ . Thus, the Banach fixed point theorem, Thm. 2.2.9, can be applied.

What about higher order convergence ( $\rightarrow$  Def. 2.1.17, cf.  $\Phi_2$  in Ex. 2.2.3)? Also in this case we should study the derivatives of the iteration functions in the fixed point (limit point).

We give a refined convergence result only for  $n = 1$  (scalar case,  $\Phi : \text{dom}(\Phi) \subset \mathbb{R} \mapsto \mathbb{R}$ ):

**Theorem 2.2.15. Taylor's formula**  $\rightarrow$  [77, Sect. 5.5]

If  $\Phi : U \subset \mathbb{R} \mapsto \mathbb{R}$ ,  $U$  interval, is  $m + 1$  times continuously differentiable,  $x \in U$

$$\Phi(y) - \Phi(x) = \sum_{k=1}^m \frac{1}{k!} \Phi^{(k)}(x)(y-x)^k + O(|y-x|^{m+1}) \quad \forall y \in U. \quad (2.2.16)$$

Now apply Taylor expansion (2.2.16) to iteration function  $\Phi$ :

If  $\Phi(x^*) = x^*$  and  $\Phi : \text{dom}(\Phi) \subset \mathbb{R} \mapsto \mathbb{R}$  is “sufficiently smooth”, it tells us that

$$x^{(k+1)} - x^* = \Phi(x^{(k)}) - \Phi(x^*) = \sum_{l=1}^m \frac{1}{l!} \Phi^{(l)}(x^*)(x^{(k)} - x^*)^l + O(|x^{(k)} - x^*|^{m+1}). \quad (2.2.17)$$

Here we used the Landau symbol  $O(\cdot)$  to describe the local behavior of a remainder term in the vicinity of  $x^*$ .

**Lemma 2.2.18. Higher order local convergence of fixed point iterations**

If  $\Phi : U \subset \mathbb{R} \mapsto \mathbb{R}$  is  $m + 1$  times continuously differentiable,  $\Phi(x^*) = x^*$  for some  $x^*$  in the interior of  $U$ , and  $\Phi^{(l)}(x^*) = 0$  for  $l = 1, \dots, m$ ,  $m \geq 1$ , then the fixed point iteration (2.2.2) converges locally to  $x^*$  with  $\text{order} \geq m + 1$  ( $\rightarrow$  Def. 2.1.17).

*Proof.* For neighborhood  $\mathcal{U}$  of  $x^*$

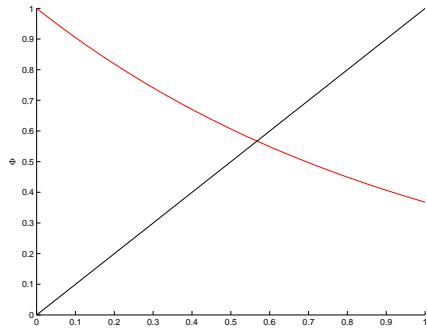
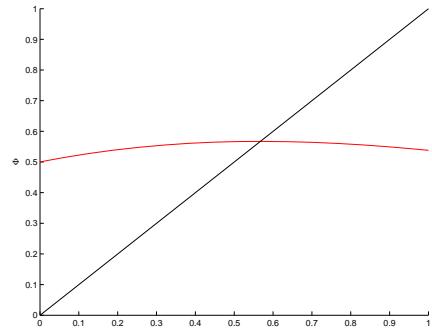
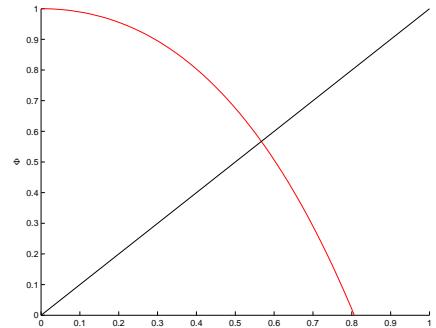
$$(2.2.17) \Rightarrow \exists C > 0: |\Phi(y) - \Phi(x^*)| \leq C |y - x^*|^{m+1} \quad \forall y \in \mathcal{U}. \\ \delta^m C < 1/2: |x^{(0)} - x^*| < \delta \Rightarrow |x^{(k)} - x^*| < 2^{-k} \delta \Rightarrow \text{local convergence}.$$

Then appeal to (2.2.17)

□

### Experiment 2.2.19 (Exp. 2.2.4 continued)

Now, Lemma 2.2.12 and Lemma 2.2.18 permit us a precise prediction of the (asymptotic) convergence we can expect from the different fixed point iterations studied in Exp. 2.2.3.

function  $\Phi_1$ function  $\Phi_2$ function  $\Phi_3$ 

$$\Phi'_2(x) = \frac{1 - xe^x}{(1 + e^x)^2} = 0 \quad \text{, if } xe^x - 1 = 0 \quad \text{hence quadratic convergence ! .}$$

Since  $x^*e^{x^*} - 1 = 0$ , simple computations yield

$$\begin{aligned} \Phi'_1(x) = -e^{-x} &\Rightarrow \Phi'_1(x^*) = -x^* \approx -0.56 \quad \text{hence local linear convergence .} \\ \Phi'_3(x) = 1 - xe^x - e^x &\Rightarrow \Phi'_3(x^*) = -\frac{1}{x^*} \approx -1.79 \quad \text{hence no convergence .} \end{aligned}$$

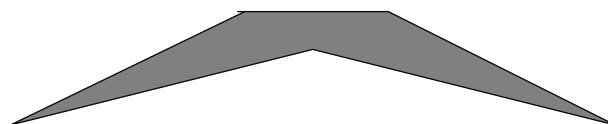
### Remark 2.2.20 (Termination criterion for contractive fixed point iteration)

We recall the considerations of Rem. 2.1.28 about a termination criterion for contractive fixed point iteration (= linearly convergent fixed point iteration → Def. 2.1.9), c.f. (2.2.7), with contraction factor (= rate of convergence)  $0 \leq L < 1$ :

$$\begin{aligned} \|\mathbf{x}^{(k+m)} - \mathbf{x}^{(k)}\| &\stackrel{\triangle\text{-ineq.}}{\leq} \sum_{j=k}^{k+m-1} \|\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)}\| \leq \sum_{j=k}^{k+m-1} L^{j-k} \|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| \\ &= \frac{1 - L^m}{1 - L} \|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| \leq \frac{1 - L^m}{1 - L} L^{k-l} \|\mathbf{x}^{(l+1)} - \mathbf{x}^{(l)}\|. \end{aligned}$$

Hence, for  $m \rightarrow \infty$ , with  $\mathbf{x}^* := \lim_{k \rightarrow \infty} \mathbf{x}^{(k)}$  we find the estimate

$$\|\mathbf{x}^* - \mathbf{x}^{(k)}\| \leq \frac{L^{k-l}}{1 - L} \|\mathbf{x}^{(l+1)} - \mathbf{x}^{(l)}\|. \quad (2.2.21)$$



<p>Set <math>l = 0</math> in (2.2.21)</p> <p>a priori termination criterion</p> $\ \mathbf{x}^* - \mathbf{x}^{(k)}\  \leq \frac{L^k}{1-L} \ \mathbf{x}^{(1)} - \mathbf{x}^{(0)}\  \quad (2.2.22)$	<p>Set <math>l = k - 1</math> in (2.2.21)</p> <p>a posteriori termination criterion</p> $\ \mathbf{x}^* - \mathbf{x}^{(k)}\  \leq \frac{L}{1-L} \ \mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\  \quad (2.2.23)$
---	---

With the same arguments as in Rem. 2.1.28 we see that **overestimating  $L$** , that is, using a value for  $L$  that is larger than the true value, still gives **reliable** termination criteria.

However, whereas overestimating  $L$  in (2.2.23) will not lead to a severe deterioration of the bound, unless  $L \approx 1$ , using a pessimistic value for  $L$  in (2.2.22) will result in a bound way bigger than the true bound, if  $k \gg 1$ . Then the a priori termination criterion (2.2.22) will recommend termination many iterations after the accuracy requirements have already been met. This will **thwart** the **efficiency** of the method.

## 2.3 Finding Zeros of Scalar Functions



*Supplementary reading.* [6, Ch. 3] is also devoted to this topic. The algorithm of “bisection” discussed in the next subsection, is treated in [15, Sect. 5.5.1] and [6, Sect. 3.2].

Now, we focus on scalar case  $n = 1$ :

$F : I \subset \mathbb{R} \mapsto \mathbb{R}$  **continuous**,  $I$  interval

Sought:

$$x^* \in I : F(x^*) = 0$$

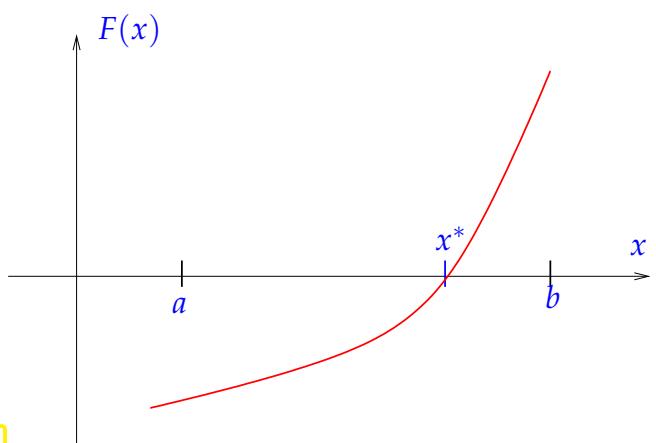
### 2.3.1 Bisection

Idea: use ordering of real numbers & intermediate value theorem [77, Sect. 4.6]

Input:  $a, b \in I$  such that  $F(a)F(b) < 0$  (different signs !)

$$\Rightarrow \exists x^* \in ]\min\{a, b\}, \max\{a, b\}[ : \\ F(x^*) = 0 ,$$

as we conclude from the intermediate value theorem.



### (2.3.1) Bisection method

The following MATLAB code implements the bisection method for finding the zeros of a function passed through the **function handle**  $F$  in the interval  $[a, b]$  with **absolute tolerance**  $\text{tol}$ .

#### MATLAB-code 2.3.2: Bisection method for solving $F(x) = 0$ on $[a, b]$

```

1 function x = bisect(F,a,b,tol)
2 % Searching zero of F in [a,b] by bisection
3 if (a>b), t=a; a=b; b=t; end;
4 fa = F(a); fb = F(b);
5 if (fa*fb>0), error('f(a), f(b) same sign'); end;
6 if (fa > 0), v=-1; else v = 1; end
7 x = 0.5*(b+a);
8 while ((b-a) > tol) && ((a<x) & (x<b))
9     if (v*F(x)>0), b=x; else a=x; end;
10    x = 0.5*(a+b)
11 end

```

Line 8: the highlighted expression offers a safeguard against an infinite loop in case  $\text{tol} <$  resolution of  $\mathbb{M}$  at zero  $x^*$  (cf. “ $\mathbb{M}$ -based termination criterion”).

This is an example for an algorithm that (in the case of  $\text{tol}=0$ ) uses the properties of machine arithmetic to define an a posteriori termination criterion, see Section 2.1.2. The iteration will terminate, when, e.g.,  $a \tilde{+} \frac{1}{2}(b - a) = a$  ( $\tilde{+}$  is the floating point realization of addition), which, by the Ass. 1.5.31 can only happen, when

$$\left| \frac{1}{2}(b - a) \right| \leq \text{EPS} \cdot |a| .$$

Since the exact zero is located between  $a$  and  $b$ , this condition implies a relative error  $\leq \text{EPS}$  of the computed zero.



Advantages:

- “foolproof”, **robust**: will always terminate with a zero of requested accuracy,
- requires only point evaluations of  $F$ ,
- works with *any* continuous function  $F$ , no derivatives needed.



Drawbacks:

Merely “linear-type” (\*) convergence:  $|x^{(k)} - x^*| \leq 2^{-k}|b - a|$

(\*): the convergence of a bisection algorithm is not linear in the sense of Def. 2.1.9, because the condition  $\|x^{(k+1)} - x^*\| \leq L \|x^{(k)} - x^*\|$  might be violated at any step of the iteration.

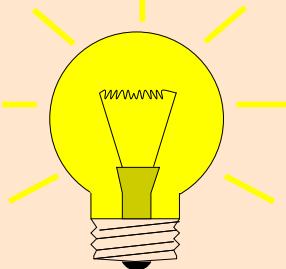
#### Remark 2.3.3 (Generalized bisection methods)

It is straightforward to combine the bisection idea with more elaborate “model function methods” as they will be discussed in the next section: Instead of stubbornly choosing the midpoint of the probing interval  $[a, b]$  ( $\rightarrow$  Code 2.3.2) as next iterate, one may use a refined guess for the location of a zero of  $F$  in  $[a, b]$ .

A method of this type is used by MATLAB’s `fzero` function for root finding in 1D [63, Sect. 6.2.3].

## 2.3.2 Model function methods

≈ class of iterative methods for finding zeroes of  $F$ : iterate in step  $k+1$  is computed according to the following idea:



Idea: Given recent iterates (approximate zeroes)  $x^{(k)}, x^{(k-1)}, \dots, x^{(k-m+1)}, m \in \mathbb{N}$

- ① replace  $F$  with model function  $\tilde{F}_k$   
(based on function values  $F(x^{(k)}), F(x^{(k-1)}), \dots, F(x^{(k-m+1)})$  and, possibly, derivative values  $F'(x^{(k)}), F'(x^{(k-1)}), \dots, F'(x^{(k-m+1)})$ )
- ②  $x^{(k+1)} :=$  zero of  $\tilde{F}_k$ :  $\tilde{F}_k(x^{(k+1)}) = 0$   
(has to be readily available  $\leftrightarrow$  analytic formula)

Distinguish (see § 2.1.2 and (2.1.4)):

one-point methods :  $x^{(k+1)} = \Phi_F(x^{(k)}), k \in \mathbb{N}$  (e.g., fixed point iteration → Section 2.2)

multi-point methods :  $x^{(k+1)} = \Phi_F(x^{(k)}, x^{(k-1)}, \dots, x^{(k-m)}), k \in \mathbb{N}, m = 2, 3, \dots$

### 2.3.2.1 Newton method in scalar case



*Supplementary reading.* Newton's method in 1D is discussed in [42, Sect. 18.1], [15, Sect. 5.5.2], [6, Sect. 3.4].

Again we consider the problem of finding zeros of the function  $F : I \subset \mathbb{R} \rightarrow \mathbb{R}$ .

Now we assume that  $F : I \subset \mathbb{R} \mapsto \mathbb{R}$  is continuously differentiable.

Now model function := tangent at  $F$  in  $x^{(k)}$ :

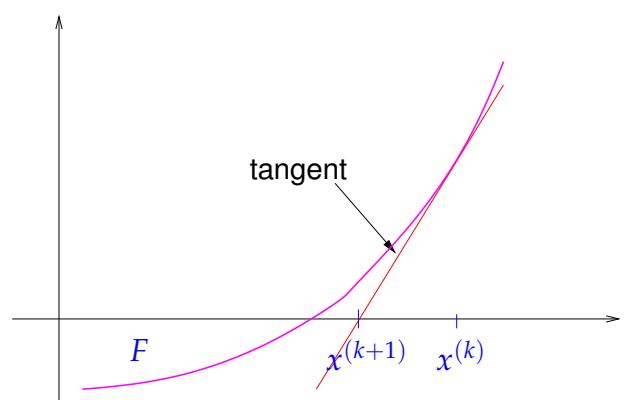
$$\tilde{F}_k(x) := F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)})$$

take  $x^{(k+1)} :=$  zero of tangent

We obtain the **Newton iteration**

$$x^{(k+1)} := x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})},$$

(2.3.4)



that requires  $F'(x^{(k)}) \neq 0$ .

**C++11-code 2.3.5: Newton method in the scalar case  $n = 1$** 

```

1  template <typename FuncType, typename DervType, typename Scalar>
2  Scalar newton1D(const FuncType &F, const DervType &DF,
3                  const Scalar &x0, double rtol, double atol)
4  {
5      Scalar s, z = x0;
6      do {
7          s = F(z)/DF(z); // compute Newton correction
8          z -= s;           // compute next iterate
9      }
10     // correction based termination (relative and absolute)
11     while ((std::abs(s) > rtol*std::abs(z)) && (std::abs(s) > atol));
12     return (z);
13 }
```

**Example 2.3.6 (Square root iteration as Newton's method)**

In Ex. 2.1.20 we learned about the *quadratically convergent* fixed point iteration (2.1.21) for the approximate computation of the square root of a positive number. It can be derived as a Newton iteration (2.3.4)!

For  $F(x) = x^2 - a$ ,  $a > 0$ , we find  $F'(x) = 2x$ , and, thus, the Newton iteration for finding zeros of  $F$  reads:

$$x^{(k+1)} = x^{(k)} - \frac{(x^{(k)})^2 - a}{2x^{(k)}} = \frac{1}{2}(x^{(k)} + \frac{a}{x^{(k)}}),$$

which is exactly (2.1.21). Thus, for this  $F$  Newton's method converges globally with order  $p = 2$ .

**Example 2.3.7 (Newton method in 1D ( $\rightarrow$  Exp. 2.2.3))**

Newton iterations for two different scalar non-linear equations  $F(x) = 0$  with the same solution sets:

$$\begin{aligned} F(x) = xe^x - 1 \Rightarrow F'(x) = e^x(1+x) \Rightarrow x^{(k+1)} &= x^{(k)} - \frac{x^{(k)}e^{x^{(k)}} - 1}{e^{x^{(k)}}(1+x^{(k)})} = \frac{(x^{(k)})^2 + e^{-x^{(k)}}}{1+x^{(k)}} \\ F(x) = x - e^{-x} \Rightarrow F'(x) = 1 + e^{-x} \Rightarrow x^{(k+1)} &= x^{(k)} - \frac{x^{(k)} - e^{-x^{(k)}}}{1 + e^{-x^{(k)}}} = \frac{1 + x^{(k)}}{1 + e^{x^{(k)}}} \end{aligned}$$

Exp. 2.2.3 confirms **quadratic convergence** in both cases! ( $\rightarrow$  Def. 2.1.17)

Note that for the computation of its zeros, the function  $F$  in this example can be recast in different forms!

In fact, based on Lemma 2.2.18 it is straightforward to show local quadratic convergence of Newton's method to a zero  $x^*$  of  $F$ , provided that  $F'(x^*) \neq 0$ :

Newton iteration (2.3.4)  $\doteq$  fixed point iteration ( $\rightarrow$  Section 2.2) with iteration function

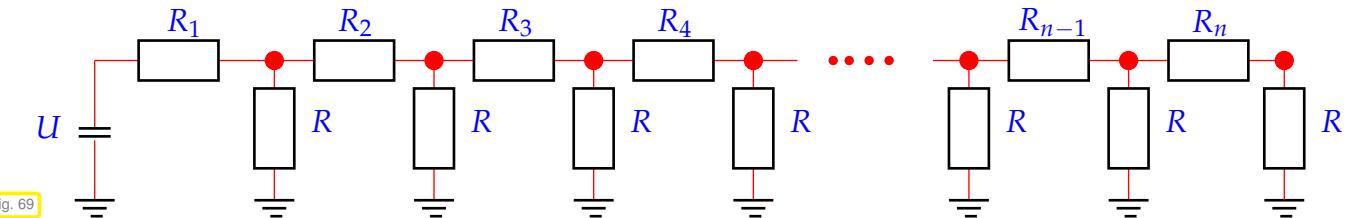
$$\Phi(x) = x - \frac{F(x)}{F'(x)} \Rightarrow \Phi'(x) = \frac{F(x)F''(x)}{(F'(x))^2} \Rightarrow \Phi'(x^*) = 0, \text{ if } F(x^*) = 0, F'(x^*) \neq 0.$$

Thus from Lemma 2.2.18 we conclude the following result:

### Convergence of Newton's method in 1D

Newton's method locally quadratically converges ( $\rightarrow$  Def. 2.1.17) to a zero  $x^*$  of  $F$ , if  $F'(x^*) \neq 0$

### Example 2.3.9 (Implicit differentiation of $F$ )



How do we have to choose the leak resistance  $R$  in the linear circuit displayed in Fig. 69 in order to achieve a prescribed potential at one of the nodes?

Using nodal analysis of the circuit introduced in Ex. 1.6.3, this problem can be formulated as: find  $x \in \mathbb{R}$  such that

$$F(x) = 0 \quad \text{with} \quad F : \begin{cases} \mathbb{R} & \rightarrow \mathbb{R} \\ x & \mapsto \mathbf{w}^\top (\mathbf{A} + x\mathbf{I})^{-1} \mathbf{b} - 1 \end{cases}, \quad (2.3.10)$$

where  $\mathbf{A} \in \mathbb{R}^{n,n}$  is a symmetric, tridiagonal, diagonally dominant matrix,  $\mathbf{w} \in \mathbb{R}^n$  is a unit vector singling out the node of interest, and  $\mathbf{b}$  takes into account the exciting voltage  $U$ .

In order to apply Newton's method to (2.3.10), we have to determine the derivative  $F'(x)$  and so by implicit differentiation [77, Sect. 7.8], first rewriting ( $\mathbf{u}(x) \doteq$  vector of nodal potentials as a function of  $x = R^{-1}$ )

$$F(x) = \mathbf{w}^\top \mathbf{u}(x) - 1, \quad (\mathbf{A} + x\mathbf{I})\mathbf{u}(x) = \mathbf{b}.$$

Then we differentiate the linear system of equations defining  $\mathbf{u}(x)$  on both sides with respect to  $x$  using the product rule (2.4.9)

$$\begin{aligned} (\mathbf{A} + x\mathbf{I})\mathbf{u}(x) = \mathbf{b} &\xrightarrow{\frac{d}{dx}} (\mathbf{A} + x\mathbf{I})\mathbf{u}'(x) + \mathbf{u}(x) = \mathbf{0}. \\ \mathbf{u}'(x) &= -(\mathbf{A} + x\mathbf{I})^{-1}\mathbf{u}(x). \end{aligned} \quad (2.3.11)$$

$$\mathbf{u}'(x) = -(\mathbf{A} + x\mathbf{I})^{-1}\mathbf{u}(x). \quad (2.3.12)$$

Thus, the Newton iteration for (2.3.10) reads:

$$x^{(k+1)} = x^{(k)} + \frac{\mathbf{w}^\top \mathbf{u}(x^{(k)}) - 1}{\mathbf{w}^\top (\mathbf{A} + x^{(k)}\mathbf{I})^{-1}\mathbf{u}(x^{(k)})}, \quad (\mathbf{A} + x^{(k)}\mathbf{I})\mathbf{u}(x^{(k)}) = \mathbf{b}. \quad (2.3.13)$$

In each step of the iteration we have to solve two linear systems of equations, which can be done with asymptotic effort  $\mathcal{O}(n)$  in this case, because  $\mathbf{A} + x^{(k)}\mathbf{I}$  is tridiagonal.

Note that in a practical application one must demand  $x > 0$ , in addition, because the solution must provide a meaningful conductance (= inverse resistance.)

Also note that bisection ( $\rightarrow$  2.3.1) is a viable alternative to using Newton's method in this case.

### 2.3.2.2 Special one-point methods

Idea underlying other one-point methods: non-linear local approximation

Useful, if *a priori knowledge* about the structure of  $F$  (e.g. about  $F$  being a rational function, see below) is available. This is often the case, because many problems of 1D zero finding are posed for functions given in analytic form with a few parameters.

Prerequisite: Smoothness of  $F$ :  $F \in C^m(I)$  for some  $m > 1$

#### Example 2.3.14 (Halley's iteration) → [42, Sect. 18.3]

This example demonstrates that non-polynomial model functions can offer excellent approximation of  $F$ . In this example the model function is chosen as a quotient of two linear function, that is, from the simplest class of true rational functions.

Of course, that this function provides a good model function is merely “a matter of luck”, unless you have some more information about  $F$ . Such information might be available from the application context.

Given  $x^{(k)} \in I$ , next iterate := zero of model function:  $h(x^{(k+1)}) = 0$ , where

$$h(x) := \frac{a}{x+b} + c \quad (\text{rational function}) \text{ such that } F^{(j)}(x^{(k)}) = h^{(j)}(x^{(k)}), \quad j = 0, 1, 2.$$



$$\frac{a}{x^{(k)} + b} + c = F(x^{(k)}), \quad -\frac{a}{(x^{(k)} + b)^2} = F'(x^{(k)}), \quad \frac{2a}{(x^{(k)} + b)^3} = F''(x^{(k)}).$$



$$x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})} \cdot \frac{1}{1 - \frac{1}{2} \frac{F(x^{(k)})F''(x^{(k)})}{F'(x^{(k)})^2}}.$$

Halley's iteration for  $F(x) = \frac{1}{(x+1)^2} + \frac{1}{(x+0.1)^2} - 1$ ,  $x > 0$ : and  $x^{(0)} = 0$

$k$	$x^{(k)}$	$F(x^{(k)})$	$x^{(k)} - x^{(k-1)}$	$x^{(k)} - x^*$
1	0.19865959351191	10.90706835180178	-0.19865959351191	-0.84754290138257
2	0.69096314049024	0.94813655914799	-0.49230354697833	-0.35523935440424
3	1.02335017694603	0.03670912956750	-0.33238703645579	-0.02285231794846
4	1.04604398836483	0.00024757037430	-0.02269381141880	-0.00015850652965
5	1.04620248685303	0.00000001255745	-0.00015849848821	-0.00000000804145

Compare with Newton method (2.3.4) for the same problem:

$k$	$x^{(k)}$	$F(x^{(k)})$	$x^{(k)} - x^{(k-1)}$	$x^{(k)} - x^*$
1	0.04995004995005	44.38117504792020	-0.04995004995005	-0.99625244494443
2	0.12455117953073	19.62288236082625	-0.07460112958068	-0.92165131536375
3	0.23476467495811	8.57909346342925	-0.11021349542738	-0.81143781993637
4	0.39254785728080	3.63763326452917	-0.15778318232269	-0.65365463761368
5	0.60067545233191	1.42717892023773	-0.20812759505112	-0.44552704256257
6	0.82714994286833	0.46286007749125	-0.22647449053641	-0.21905255202615
7	0.99028203077844	0.09369191826377	-0.16313208791011	-0.05592046411604
8	1.04242438221432	0.00592723560279	-0.05214235143588	-0.00377811268016
9	1.04618505691071	0.00002723158211	-0.00376067469639	-0.00001743798377
10	1.04620249452271	0.00000000058056	-0.00001743761199	-0.00000000037178

Note that Halley's iteration is superior in this case, since  $F$  is a rational function.

! Newton method converges more slowly, but also needs less effort per step (→ Section 2.3.3)

In the previous example Newton's method performed rather poorly. Often its convergence can be boosted by converting the non-linear equation to an equivalent one (that is, one with the same solutions) for another function  $g$ , which is “closer to a linear function”:

Assume  $F \approx \widehat{F}$ , where  $\widehat{F}$  is invertible with an inverse  $\widehat{F}^{-1}$  that can be evaluated with little effort.

$$\Rightarrow g(x) := \widehat{F}^{-1}(F(x)) \approx x .$$

Then apply Newton's method to  $g(x)$ , using the formula for the derivative of the inverse of a function

$$\frac{d}{dy}(\widehat{F}^{-1})(y) = \frac{1}{\widehat{F}'(\widehat{F}^{-1}(y))} \Rightarrow g'(x) = \frac{1}{\widehat{F}'(g(x))} \cdot F'(x) .$$

### Example 2.3.15 (Adapted Newton method)

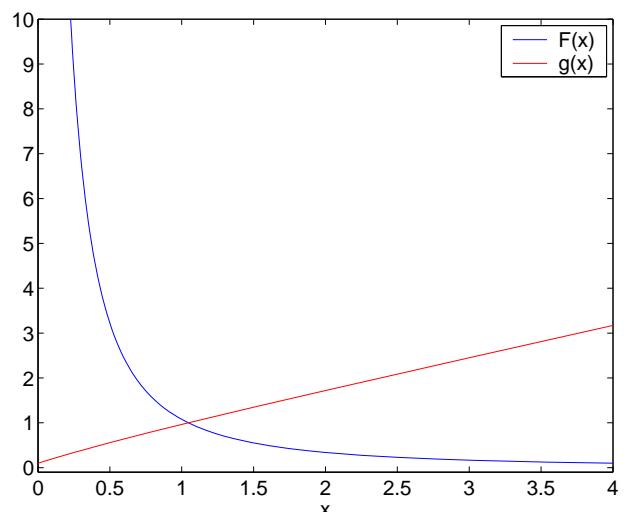
As in Ex. 2.3.14:

$$F(x) = \frac{1}{(x+1)^2} + \frac{1}{(x+0.1)^2} - 1 , \quad x > 0 :$$

Observation:

$$F(x) + 1 \approx 2x^{-2} \text{ for } x \gg 1$$

and so  $g(x) := \frac{1}{\sqrt{F(x)+1}}$  is “almost” linear for  $x \gg 1$ .



Idea: instead of  $F(x) \stackrel{!}{=} 0$  tackle  $g(x) \stackrel{!}{=} 1$  with Newton's method (2.3.4).

$$\begin{aligned} x^{(k+1)} &= x^{(k)} - \frac{g(x^{(k)}) - 1}{g'(x^{(k)})} = x^{(k)} + \left( \frac{1}{\sqrt{F(x^{(k)}) + 1}} - 1 \right) \frac{2(F(x^{(k)}) + 1)^{3/2}}{F'(x^{(k)})} \\ &= x^{(k)} + \frac{2(F(x^{(k)}) + 1)(1 - \sqrt{F(x^{(k)}) + 1})}{F'(x^{(k)})}. \end{aligned}$$

Convergence recorded for  $x^{(0)} = 0$ :

$k$	$x^{(k)}$	$F(x^{(k)})$	$x^{(k)} - x^{(k-1)}$	$x^{(k)} - x^*$
1	0.91312431341979	0.24747993091128	0.91312431341979	-0.13307818147469
2	1.04517022155323	0.00161402574513	0.13204590813344	-0.00103227334125
3	1.04620244004116	0.00000008565847	0.00103221848793	-0.00000005485332
4	1.04620249489448	0.0000000000000000	0.00000005485332	-0.0000000000000000

For zero finding there is wealth of iterative methods that offer higher order of convergence. One class is discussed next.

### (2.3.16) Modified Newton methods

Taking the cue from the iteration function of Newton's method (2.3.4), we extend it by introducing an extra function  $H$ :

► new fixed point iteration :  $\Phi(x) = x - \frac{F(x)}{F'(x)}H(x)$  with "proper"  $H : I \mapsto \mathbb{R}$ .

Still, every zero of  $F$  is a fixed point of this  $\Phi$ , that is, the fixed point iteration is still consistent ( $\rightarrow$  Def. 2.2.1).

Aim: find  $H$  such that the method is of  $p$ -th order. The main tool is Lemma 2.2.18, which tells us that we have to ensure  $\Phi^{(\ell)}(x^*) = 0, 1 \leq \ell \leq p-1$ , guarantees local convergence of order  $p$ .

Assume:  $F$  smooth "enough" and  $\exists x^* \in I : F(x^*) = 0, F'(x^*) \neq 0$ . Then we can compute the derivatives of  $\Phi$  appealing to the product rule and quotient rule for derivatives.

$$\begin{aligned} \Phi &= x - uH \quad , \quad \Phi' = 1 - u'H - uH' \quad , \quad \Phi'' = -u''H - 2u'H - uH'' \quad , \\ \text{with } u = \frac{F}{F'} &\Rightarrow u' = 1 - \frac{FF''}{(F')^2} \quad , \quad u'' = -\frac{F''}{F'} + 2\frac{F(F'')^2}{(F')^3} - \frac{FF'''}{(F')^2}. \end{aligned}$$

$$F(x^*) = 0 \quad \Rightarrow \quad u(x^*) = 0, u'(x^*) = 1, u''(x^*) = -\frac{F''(x^*)}{F'(x^*)}.$$

$$\quad \Rightarrow \quad \Phi'(x^*) = 1 - H(x^*) \quad , \quad \Phi''(x^*) = \frac{F''(x^*)}{F'(x^*)}H(x^*) - 2H'(x^*) . \quad (2.3.17)$$

Lemma 2.2.18 ► **Necessary** conditions for local convergence of order  $p$ :

$$p = 2 \text{ (quadratical convergence):} \quad H(x^*) = 1,$$

$$p = 3 \text{ (cubic convergence):} \quad H(x^*) = 1 \quad \wedge \quad H'(x^*) = \frac{1}{2} \frac{F''(x^*)}{F'(x^*)}.$$

Trial expression:  $H(x) = G(1 - u'(x))$  with “appropriate”  $G$

► fixed point iteration  $x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})} G\left(\frac{F(x^{(k)})F''(x^{(k)})}{(F'(x^{(k)}))^2}\right)$ . (2.3.18)

### Lemma 2.3.19. Cubic convergence of modified Newton methods

If  $F \in C^2(I)$ ,  $F(x^*) = 0$ ,  $F'(x^*) \neq 0$ ,  $G \in C^2(U)$  in a neighbourhood  $U$  of  $0$ ,  $G(0) = 1$ ,  $G'(0) = \frac{1}{2}$ , then the fixed point iteration (2.3.18) converge locally cubically to  $x^*$ .

*Proof.* We apply Lemma 2.2.18, which tells us that both derivatives from (2.3.17) have to vanish. Using the definition of  $H$  we find.

$$H(x^*) = G(0) , \quad H'(x^*) = -G'(0)u''(x^*) = G'(0)\frac{F''(x^*)}{F'(x^*)} .$$

Plugging these expressions into (2.3.17) finishes the proof. □

### Experiment 2.3.20 (Application of modified Newton methods)

- $G(t) = \frac{1}{1 - \frac{1}{2}t} \rightarrow$  Halley's iteration ( $\rightarrow$  Ex. 2.3.14)
- $G(t) = \frac{2}{1 + \sqrt{1 - 2t}} \rightarrow$  Euler's iteration
- $G(t) = 1 + \frac{1}{2}t \rightarrow$  quadratic inverse interpolation

Numerical experiment:

$$F(x) = xe^x - 1 ,$$

$$x^{(0)} = 5$$

$k$	$e^{(k)} := x^{(k)} - x^*$		
	Halley	Euler	Quad. Inv.
1	2.81548211105635	3.57571385244736	2.03843730027891
2	1.37597082614957	2.76924150041340	1.02137913293045
3	0.34002908011728	1.95675490333756	0.28835890388161
4	0.00951600547085	1.25252187565405	0.01497518178983
5	0.00000024995484	0.51609312477451	0.00000315361454
6		0.14709716035310	
7		0.00109463314926	
8		0.00000000107549	

### 2.3.2.3 Multi-point methods



*Supplementary reading.* The secant method is presented in [42, Sect. 18.2], [15, Sect. 5.5.3], [6, Sect. 3.4].

### Construction of multi-point iterations in 1D



Idea:

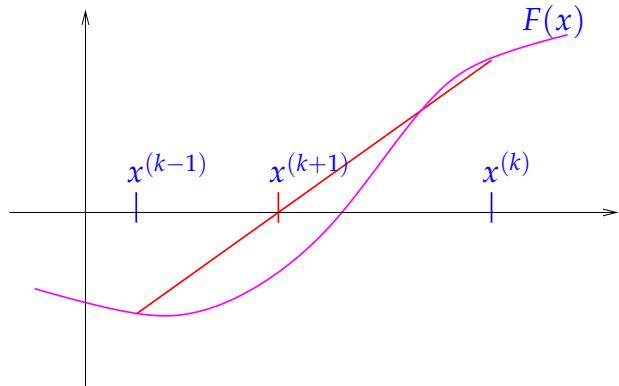
Replace  $F$  with **interpolating polynomial**  
producing interpolatory model function methods

#### (2.3.22) The secant method

Simplest representative of model function multi-point methods:

secant method

$x^{(k+1)}$  = zero of secant (red line ▷)



The secant line is the graph of the function

$$s(x) = F(x^{(k)}) + \frac{F(x^{(k)}) - F(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}(x - x^{(k)}), \quad (2.3.23)$$

$$\Rightarrow x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})(x^{(k)} - x^{(k-1)})}{F(x^{(k)}) - F(x^{(k-1)})}. \quad (2.3.24)$$

MATLAB implementation of secant method:

- Only *one* function evaluation per step
- **no derivatives required!**
- 2-point method: two initial guesses needed

User can specify absolute and relative tolerances.

#### MATLAB-code 2.3.25: secant method

```

1 function x = secant (x0,x1,F,rtol,atol)
2 fo = F (x0);
3 for i=1:MAXIT
4 fn = F (x1);
5 s = fn*(x1-x0)/(fn-fo); % correction
6 x0 = x1; x1 = x1-s;
7 if ((abs(s) <
8 max(atol,rtol*min(abs([x0;x1])))))
9 x = x1; return; end
10 fo = fn;
end

```

Remember:  $F(x)$  may only be available as output of a (complicated) procedure. In this case it is difficult to find a procedure that evaluates  $F'(x)$ . Thus the significance of methods that do not involve evaluations of derivatives.

#### Experiment 2.3.26 (Convergence of secant method)

Model problem: find zero of  $F(x) = xe^x - 1$ , using secant method of Code 2.3.25 with initial guesses  $x^{(0)} = 0, x^{(1)} = 5$ .

$k$	$x^{(k)}$	$F(x^{(k)})$	$e^{(k)} := x^{(k)} - x^*$	$\frac{\log  e^{(k+1)}  - \log  e^{(k)} }{\log  e^{(k)}  - \log  e^{(k-1)} }$
2	0.00673794699909	-0.99321649977589	-0.56040534341070	
3	0.01342122983571	-0.98639742654892	-0.55372206057408	24.43308649757745
4	0.98017620833821	1.61209684919288	0.41303291792843	2.70802321457994
5	0.38040476787948	-0.44351476841567	-0.18673852253030	1.48753625853887
6	0.50981028847430	-0.15117846201565	-0.05733300193548	1.51452723840131
7	0.57673091089295	0.02670169957932	0.00958762048317	1.70075240166256
8	0.56668541543431	-0.00126473620459	-0.00045787497547	1.59458505614449
9	0.56713970649585	-0.00000990312376	-0.00000358391394	1.62641838319117
10	0.56714329175406	0.00000000371452	0.00000000134427	
11	0.56714329040978	-0.0000000000000001	-0.0000000000000000	

Rem. 2.1.19: the rightmost column of the table provides an estimate for the order of convergence  $\rightarrow$  Def. 2.1.17. For further explanations see Rem. 2.1.19.

A startling observation: the method seems to have a *fractional* (!) order of convergence, see Def. 2.1.17.

### Remark 2.3.27 (Fractional order of convergence of secant method)

Indeed, a fractional order of convergence can be proved for the secant method, see [42, Sect. 18.2]. Here we give an *asymptotic* argument that holds, if the iterates are already very close to the zero  $x^*$  of  $F$ .

We can write the secant method in the form (2.1.4)

$$x^{(k+1)} = \Phi(x^{(k)}, x^{(k-1)}) \quad \text{with} \quad \Phi(x, y) = \Phi(x, y) := x - \frac{F(x)(x - y)}{F(x) - F(y)}. \quad (2.3.28)$$

Using  $\Phi$  we find a recursion for the iteration error  $e^{(k)} := x^{(k)} - x^*$ :

$$e^{(k+1)} = \Phi(x^* + e^{(k)}, x^* + e^{(k-1)}) - x^*. \quad (2.3.29)$$

Thanks to the asymptotic perspective we may assume that  $|e^{(k)}|, |e^{(k-1)}| \ll 1$  so that we can rely on two-dimensional Taylor expansion around  $(x^*, x^{(*)})$ , cf. [77, Satz 7.5.2]:

$$\begin{aligned} \Phi(x^* + h, x^* + k) &= \Phi(x^*, x^*) + \frac{\partial \Phi}{\partial x}(x^*, x^*)h + \frac{\partial \Phi}{\partial y}(x^*, x^*)k + \\ &\quad \frac{1}{2} \frac{\partial^2 \Phi}{\partial x^2}(x^*, x^*)h^2 \frac{\partial^2 \Phi}{\partial x \partial y}(x^*, x^*)hk \frac{1}{2} \frac{\partial^2 \Phi}{\partial y^2}(x^*, x^*)k^2 + R(x^*, h, k), \\ &\quad \text{with } |R| \leq C(h^3 + h^2k + hk^2 + k^3). \end{aligned} \quad (2.3.30)$$

Computations invoking the quotient rule and product rule and using  $F(x^*) = 0$  show

$$\Phi(x^*, x^*) = x^*, \quad \frac{\partial \Phi}{\partial x}(x^*, x^*) = \frac{\partial \Phi}{\partial y}(x^*, x^*) = \frac{1}{2} \frac{\partial^2 \Phi}{\partial x^2}(x^*, x^*) = \frac{1}{2} \frac{\partial^2 \Phi}{\partial y^2}(x^*, x^*) = 0.$$

We may also use MAPLE to find the Taylor expansion (assuming  $F$  sufficiently smooth):

```
> Phi := (x, y) -> x-F(x)*(x-y)/(F(x)-F(y));
> F(s) := 0;
> e2 = normal(mtaylor(Phi(s+e1, s+e0)-s, [e0, e1], 4));
```

➤ truncated error propagation formula (products of three or more error terms ignored)

$$e^{(k+1)} \doteq \frac{1}{2} \frac{F''(x^*)}{F'(x^*)} e^{(k)} e^{(k-1)} = C e^{(k)} e^{(k-1)} . \quad (2.3.31)$$

How can we deduce the order of converge from this recursion formula? We try  $e^{(k)} = K(e^{(k-1)})^p$  inspired by the estimate in Def. 2.1.17:

$$\begin{aligned} & \Rightarrow e^{(k+1)} = K^{p+1} (e^{(k-1)})^{p^2} \\ \Rightarrow & (e^{(k-1)})^{p^2-p-1} = K^{-p} C \Rightarrow p^2 - p - 1 = 0 \Rightarrow p = \frac{1}{2}(1 \pm \sqrt{5}) . \end{aligned}$$

As  $e^{(k)} \rightarrow 0$  for  $k \rightarrow \infty$  we get the order of convergence  $p = \frac{1}{2}(1 + \sqrt{5}) \approx 1.62$  (see Exp. 2.3.26 !)

### Example 2.3.32 (local convergence of the secant method)

Model problem: find zero of

$$F(x) = \arctan(x)$$

- $\hat{=}$  secant method converges for a pair  $(x^{(0)}, x^{(1)}) \in \mathbb{R}_+^2$  of initial guesses.

We observe that the secant method will converge only for initial guesses sufficiently close to  $0 =$  local convergence  $\rightarrow$  Def. 2.1.8

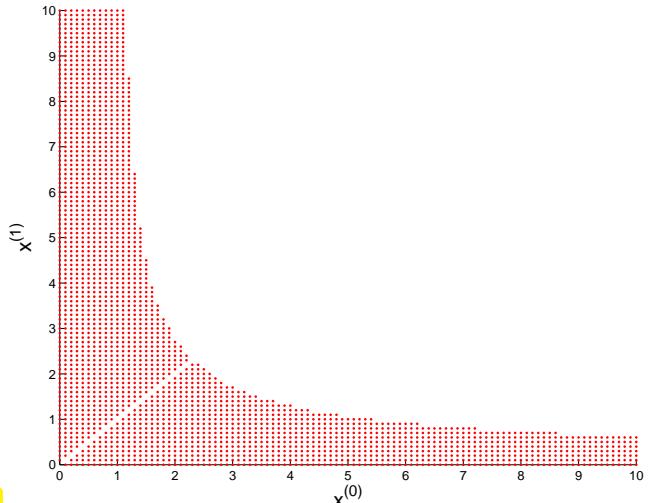


Fig. 70

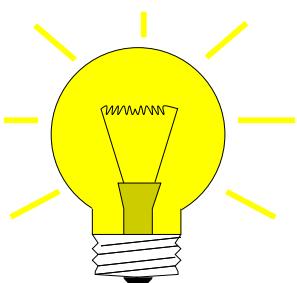
### (2.3.33) Inverse interpolation

Another class of multi-point methods: *inverse interpolation*

Assume:

$F : I \subset \mathbb{R} \mapsto \mathbb{R}$  one-to-one (monotone)

$$F(x^*) = 0 \Rightarrow F^{-1}(0) = x^* .$$



- Interpolate  $F^{-1}$  by polynomial  $p$  of degree  $m-1$  determined by

$$p(F(x^{(k-j)})) = x^{(k-j)} , \quad j = 0, \dots, m-1 .$$

- New approximate zero  $x^{(k+1)} := p(0)$

The graph of  $F^{-1}$  can be obtained by reflecting the graph of  $F$  at the angular bisector.

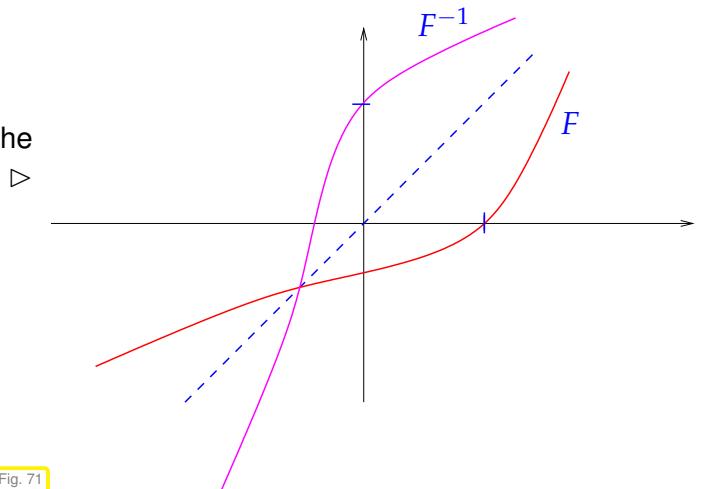


Fig. 71

Case  $m = 2$  (2-point method)  
 ➤ secant method

The interpolation polynomial is a line. In this case we do not get a new method, because the inverse function of a linear function (polynomial of degree 1) is again a polynomial of degree 1.

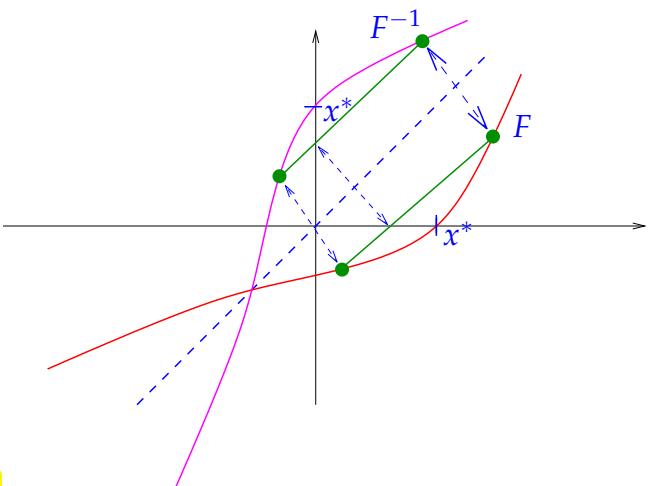


Fig. 72

Case  $m = 3$ : quadratic inverse interpolation, a 3-point method, see [56, Sect. 4.5]

We interpolate the points  $(F(x^{(k)}), x^{(k)}), (F(x^{(k-1)}), x^{(k-1)}), (F(x^{(k-2)}), x^{(k-2)})$  with a parabola (polynomial of degree 2). Note the importance of monotonicity of  $F$ , which ensures that  $F(x^{(k)}), F(x^{(k-1)}), F(x^{(k-2)})$  are mutually different.

MAPLE code:

```
p := x-> a*x^2+b*x+c;
solve({p(f0)=x0, p(f1)=x1, p(f2)=x2}, {a,b,c});
assign(%); p(0);
```

$$\Rightarrow x^{(k+1)} = \frac{F_0^2(F_1 x_2 - F_2 x_1) + F_1^2(F_2 x_0 - F_0 x_2) + F_2^2(F_0 x_1 - F_1 x_0)}{F_0^2(F_1 - F_2) + F_1^2(F_2 - F_0) + F_2^2(F_0 - F_1)}.$$

$$(F_0 := F(x^{(k-2)}), F_1 := F(x^{(k-1)}), F_2 := F(x^{(k)}), x_0 := x^{(k-2)}, x_1 := x^{(k-1)}, x_2 := x^{(k)})$$

### Experiment 2.3.34 (Convergence of quadratic inverse interpolation)

We test the method for the model problem/initial guesses  $F(x) = xe^x - 1$ ,  $x^{(0)} = 0$ ,  $x^{(1)} = 2.5$ ,  $x^{(2)} = 5$ .

$k$	$x^{(k)}$	$F(x^{(k)})$	$e^{(k)} := x^{(k)} - x^*$	$\frac{\log  e^{(k+1)}  - \log  e^{(k)} }{\log  e^{(k)}  - \log  e^{(k-1)} }$
3	0.08520390058175	-0.90721814294134	-0.48193938982803	
4	0.16009252622586	-0.81211229637354	-0.40705076418392	3.33791154378839
5	0.79879381816390	0.77560534067946	0.23165052775411	2.28740488912208
6	0.63094636752843	0.18579323999999	0.06380307711864	1.82494667289715
7	0.56107750991028	-0.01667806436181	-0.00606578049951	1.87323264214217
8	0.56706941033107	-0.00020413476766	-0.00007388007872	1.79832936980454
9	0.56714331707092	0.00000007367067	0.00000002666114	1.84841261527097
10	0.56714329040980	0.00000000000003	0.00000000000001	

Also in this case the numerical experiment hints at a fractional rate of convergence  $p \approx 1.8$ , as in the case of the secant method, see Rem. 2.3.27.

### 2.3.3 Asymptotic efficiency of iterative methods for zero finding

Efficiency is measured by forming the ratio of gain and the effort required to achieve it. For iterative methods for solving  $F(\mathbf{x}) = 0$ ,  $F : D \subset \mathbb{R}^n \rightarrow \mathbb{R}^n$ , this means the following:

<b>Efficiency</b> of an iterative method (for solving $F(\mathbf{x}) = 0$ )	$\leftrightarrow$	<b>computational effort</b> to reach prescribed number of significant digits in result.
--	-------------------	--

Ingredient ①:

$W \triangleq$  computational **effort** per step

$$(e.g., \quad W \approx \frac{\#\{\text{evaluations of } DF\}}{\text{step}} + n \cdot \frac{\#\{\text{evaluations of } F'\}}{\text{step}} + \dots)$$

Ingredient ②: number of steps  $k = k(\rho)$  to achieve *relative reduction of error* (= **gain**)

$$\|e^{(k)}\| \leq \rho \|e^{(0)}\|, \quad \rho > 0 \text{ prescribed.} \quad (2.3.35)$$

Let us consider an iterative method of order  $p \geq 1$  ( $\rightarrow$  Def. 2.1.17). Its error recursion can be converted into expressions (2.3.36) and (2.3.37) that relate the error norm  $\|e^{(k)}\|$  to  $\|e^{(0)}\|$  and lead to quantitative bounds for the number of steps to achieve (2.3.35):

$$\exists C > 0: \quad \|e^{(k)}\| \leq C \|e^{(k-1)}\|^p \quad \forall k \geq 1 \quad (C < 1 \text{ for } p = 1).$$

Assuming  $C \|e^{(0)}\|^{p-1} < 1$  (guarantees convergence!), we find the following minimum number of steps to achieve (2.3.35) for sure:

$$p = 1: \quad \|e^{(k)}\| \leq C^k \|e^{(0)}\| \quad \text{requires} \quad k \geq \frac{\log \rho}{\log C}, \quad (2.3.36)$$

$$p > 1: \quad \|e^{(k)}\| \leq C^{\frac{p^k - 1}{p-1}} \|e^{(0)}\|^{p^k} \quad \text{requires} \quad p^k \geq 1 + \frac{\log \rho}{\log C/p-1 + \log(\|e^{(0)}\|)} \\ \Rightarrow \quad k \geq \log(1 + \frac{\log \rho}{\log L_0}) / \log p, \quad (2.3.37)$$

$$L_0 := C^{1/p-1} \|e^{(0)}\| < 1.$$

Now we adopt an asymptotic perspective and ask for a large reduction of the error, that is  $\rho \ll 1$ .

If  $\rho \ll 1$ , then  $\log(1 + \frac{\log \rho}{\log L_0}) \approx \log |\log \rho| - \log |\log L_0| \approx \log |\log \rho|$ . This simplification will be made in the context of asymptotic considerations  $\rho \rightarrow 0$  below.

Notice:

$$|\log \rho| \leftrightarrow \text{Gain in no. of significant digits of } x^{(k)}$$

Measure for efficiency:

$$\text{Efficiency} := \frac{\text{no. of digits gained}}{\text{total work required}} = \frac{|\log \rho|}{k(\rho) \cdot W} \quad (2.3.38)$$

► **asymptotic efficiency for  $\rho \ll 1$**   $\rightarrow |\log \rho| \rightarrow \infty$ :

$$\text{Efficiency}_{|\rho \ll 1} = \begin{cases} -\frac{\log C}{W} & , \text{if } p = 1, \\ \frac{\log p}{W} \cdot \frac{|\log \rho|}{\log(|\log \rho|)} & , \text{if } p > 1. \end{cases} \quad (2.3.39)$$

We conclude that

- when requiring high accuracy, linearly convergent iterations should not be used, because their efficiency does not increase for  $\rho \rightarrow 0$ ,
- for method of order  $p > 1$ , the factor  $\frac{\log p}{W}$  offers a gauge for efficiency.

### Example 2.3.40 (Efficiency of iterative methods)

We choose  $\|e^{(0)}\| = 0.1, \rho = 10^{-8}$ .

The plot displays the number of iteration steps according to (2.3.37).

Higher-order method require substantially fewer steps compared to low-order methods.

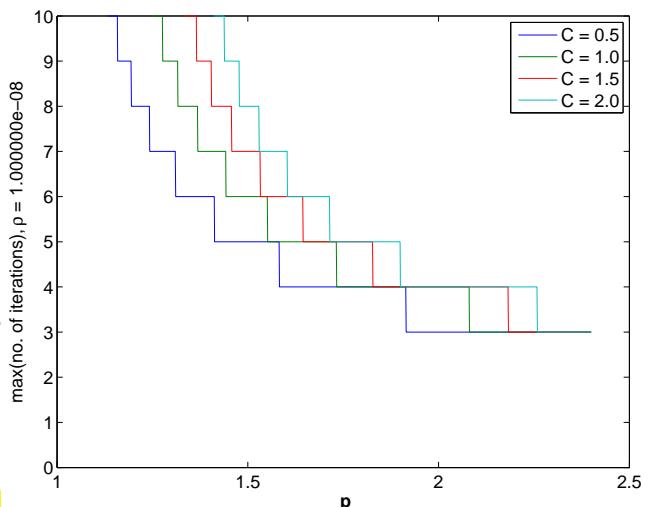


Fig. 73

We compare

Newton's method  $\leftrightarrow$  secant method

in terms of number of steps required for a prescribed guaranteed error reduction, assuming  $C = 1$  in both cases and for  $\|e^{(0)}\| = 0.1$ .

Newton's method requires only marginally fewer steps than the secant method.

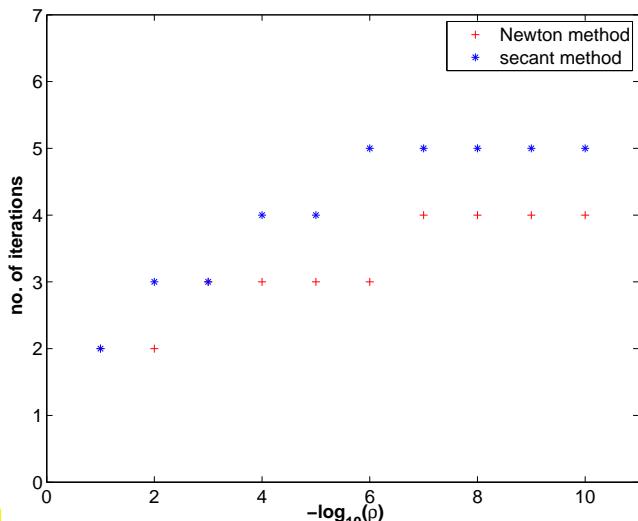


Fig. 74

We draw conclusions from the discussion above and (2.3.39):

$$\begin{aligned} W_{\text{Newton}} &= 2W_{\text{secant}}, \\ p_{\text{Newton}} &= 2, p_{\text{secant}} = 1.62 \end{aligned} \Rightarrow \frac{\log p_{\text{Newton}}}{W_{\text{Newton}}} : \frac{\log p_{\text{secant}}}{W_{\text{secant}}} = 0.71 .$$

We set the effort for a step of Newton's method to twice that for a step of the secant method from Code 2.3.25, because we need an addition evaluation of  $F'$  in Newton's method.

➤ secant method is more efficient than Newton's method!

## 2.4 Newton's Method



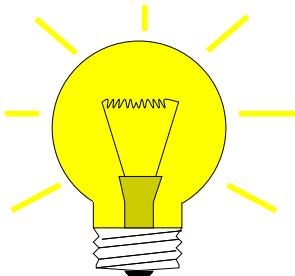
*Supplementary reading.* The multi-dimensional Newton method is also presented in [42, Sect. 19], [15, Sect. 5.6], [6, Sect. 9.1].

We consider a non-linear **system** of  $n$  equations with  $n$  unknowns:

$$\text{for } F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n \quad \text{find } \mathbf{x}^* \in D: \quad F(\mathbf{x}^*) = 0.$$

We assume:  $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$  is **continuously differentiable**

### 2.4.1 The Newton iteration



Idea ( $\rightarrow$  Section 2.3.2.1):

Given  $\mathbf{x}^{(k)} \in D \succ \mathbf{x}^{(k+1)}$  as zero of affine linear model function

$$F(\mathbf{x}) \approx \tilde{F}_k(\mathbf{x}) := F(\mathbf{x}^{(k)}) + D F(\mathbf{x}^{(k)})(\mathbf{x} - \mathbf{x}^{(k)}),$$

$$D F(\mathbf{x}) \in \mathbb{R}^{n,n} = \text{Jacobian}, \quad D F(\mathbf{x}) := \left[ \frac{\partial F_j}{\partial x_k}(\mathbf{x}) \right]_{j,k=1}^n.$$

► Newton iteration: (generalizes (2.3.4) to  $n > 1$ )

$$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} - D F(\mathbf{x}^{(k)})^{-1} F(\mathbf{x}^{(k)}) , \quad [\text{if } D F(\mathbf{x}^{(k)}) \text{ regular}] \quad (2.4.1)$$

Terminology:  $-D F(\mathbf{x}^{(k)})^{-1} F(\mathbf{x}^{(k)})$  = Newton correction

Illustration of idea of Newton's method for  $n = 2$ : ▷

Sought: intersection point  $\mathbf{x}^*$  of the curves  $F_1(\mathbf{x}) = 0$  and  $F_2(\mathbf{x}) = 0$ .

Idea:  $\mathbf{x}^{(k+1)}$  = the intersection of two straight lines (= zero sets of the components of the model function, cf. Ex. 1.6.18) that are approximations of the original curves

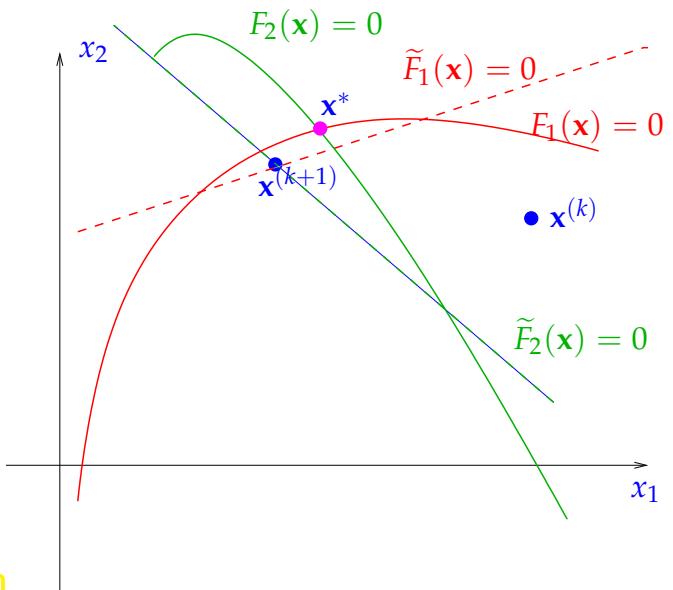


Fig. 75

#### MATLAB-code : Newton's method

MATLAB template for Newton method:

Solve linear system:

$A \setminus b = A^{-1}b \rightarrow \S\ 1.6.81$

$F$ ,  $DF$ : function handles

A posteriori termination criterion

```

1 function x = newton(x, F, DF, rtol, atol)
2 for i=1:MAXIT
3   s = DF(x) \ F(x);
4   x = x-s;
5   if ((norm(s)<rtol*norm(x)) || (norm(s)<atol))
6     return; end;
7 end

```

Here a **correction based** a posteriori termination criterion for the Newton iteration is used; it stops the iteration if the *relative* size of the Newton correction drops below the prescribed **relative tolerance**  $rtol$ . If  $\mathbf{x}^* \approx 0$  also the *absolute* size of the Newton correction has to be tested against an **absolute tolerance**  $atol$  in order to avoid non-termination despite convergence of the iteration,

#### C++11-code 2.4.2: Newton's method in C++

```

1 template <typename FuncType, typename JacType, typename VecType>
2 void newton(const FuncType &F, const JacType &DFinv,
3             VecType &x, double rtol, double atol)
4 {
5   using index_t = typename VecType::Index;
6   using scalar_t = typename VecType::Scalar;
7   const index_t n = x.size();
8   VecType s(n);
9   scalar_t sn;

```

```

10  do {
11      s = DFinv(x,F(x)); // compute Newton correction
12      x -= s;           // compute next iterate
13      sn = s.norm();
14  }
15  // correction based termination (relative and absolute)
16  while ((sn > rtol*x.norm()) && (sn > atol));
17 }
```

- Objects of type **FuncType** must feature

VecType **operator (const** VecType &x);

that evaluates  $F(\mathbf{x})$  ( $\nabla \mathbf{x} \leftrightarrow \mathbf{x}$ ).

- Objects of type **JacType** must provide a method

VecType **operator (const** VecType &x, **const** VectType &f);

that computes the Newton correction, that is it returns the solution of a linear system with system matrix  $D F(\mathbf{x})$  ( $\mathbf{x} \leftrightarrow \mathbf{x}$ ) and right hand side  $\mathbf{f} \leftrightarrow f$ .

- The argument  $\mathbf{x}$  will be overwritten with the computed solution of the non-linear system.

The next code demonstrates the invocation of `newton` for a  $2 \times 2$  non-linear system from a code relying on EIGEN. It also demonstrates the use of fixed size eigen matrices and vectors.

#### C++11-code 2.4.3: Calling `newton` with EIGEN data types

```

1 void newton2Ddriver(void)
2 {
3     % Function F defined through lambda function
4     auto F = [](const Eigen::Vector2d &x) {
5         Eigen::Vector2d z;
6         const double x1 = x(0), x2=x(1);
7         z << x1*x1-2*x1-x2+1, x1*x1+x2*x2-1;
8         return(z);
9     };
10    % \texttt{JacType} lambda function based on DF
11    auto DF = [](const Eigen::Vector2d &x, const Eigen::Vector2d &f) {
12        Eigen::Matrix2d J;
13        const double x1 = x(0), x2=x(1);
14        J << 2*x1-2, -1, 2*x1, 2*x2;
15        Eigen::Vector2d s = J.lu().solve(f);
16        return(s);
17    };
18    Eigen::Vector2d x; x << 2,3; % initial guess
19    newton(F,DF,x,1E-6,1E-8);
20    std::cout << "||F(x)|| = " << F(x).norm() << std::endl;
21 }
```



New aspect for  $n \gg 1$  (compared to  $n = 1$ -dimensional case, Section 2.3.2.1):  
Computation of the Newton correction may be expensive!  
(because it involves the solution of a LSE, cf. Thm. 1.6.79)

#### Remark 2.4.4 (Affine invariance of Newton method)

An important property of the Newton iteration (2.4.1): **affine invariance** → [20, Sect. 1.2.2]

$$\text{set } G_{\mathbf{A}}(\mathbf{x}) := \mathbf{A}F(\mathbf{x}) \quad \text{with regular } \mathbf{A} \in \mathbb{R}^{n,n} \quad \text{so that } F(\mathbf{x}^*) = 0 \Leftrightarrow G_{\mathbf{A}}(\mathbf{x}^*) = 0.$$



**Affine invariance:** Newton iterations for  $G_{\mathbf{A}}(\mathbf{x}) = 0$  are the same for all regular  $\mathbf{A}$ !

This is a simple computation:

$$D G(\mathbf{x}) = \mathbf{A} D F(\mathbf{x}) \Rightarrow D G(\mathbf{x})^{-1} G(\mathbf{x}) = D F(\mathbf{x})^{-1} \mathbf{A}^{-1} \mathbf{A} F(\mathbf{x}) = D F(\mathbf{x})^{-1} F(\mathbf{x}).$$

Use affine invariance as guideline for

- convergence theory for Newton's method: assumptions and results should be affine invariant, too.
- modifying and extending Newton's method: resulting schemes should preserve affine invariance.

In particular, termination criteria for Newton's method should also be affine invariant in the sense that, when applied for  $G_{\mathbf{A}}$  they STOP the iteration at exactly the same step for any choice of  $\mathbf{A}$ .

Frequently, the function  $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$  defining the non-linear system of equations is given in rather explicit form. In this case,  $D F$  has to be computed *symbolically* in order to obtain concrete formulas for the Newton iteration. We now learn how these symbolic computations can be carried out harnessing advanced techniques of multi-variate calculus.

#### (2.4.5) Differentiation rules → Repetition of basic analysis skills

Statement of the Newton iteration (2.4.1) for  $F : \mathbb{R}^n \mapsto \mathbb{R}^n$  given as analytic expression entails computing the Jacobian  $D F$ . The safe, but tedious way is to use the definition (2.2.11) directly and compute the partial derivatives.

To avoid cumbersome component-oriented considerations, it is sometimes useful to know the *rules of multidimensional differentiation*:

##### Definition 2.4.6. Derivative of functions between vector spaces

Let  $V, W$  be finite dimensional vector spaces and  $F : D \subset V \mapsto W$  a sufficiently smooth mapping. The **derivative** (differential)  $D F(\mathbf{x})$  of  $F$  in  $\mathbf{x} \in V$  is the *unique linear* mapping  $D F(\mathbf{x}) : V \mapsto W$  such that there is a  $\delta > 0$  and a function  $\epsilon : [0, \delta] \rightarrow \mathbb{R}^+$  satisfying  $\lim_{\xi \rightarrow 0} \epsilon(\xi) = 0$  such that

$$\|F(\mathbf{x} + \mathbf{h}) - F(\mathbf{x}) - D F(\mathbf{x})\mathbf{h}\| = \epsilon(\|\mathbf{h}\|) \quad \forall \mathbf{h} \in V, \|\mathbf{h}\| < \delta. \quad (2.4.7)$$

☞ Note that  $D F(\mathbf{x})\mathbf{h} \in W$  is the vector returned by the linear mapping  $D F(\mathbf{x})$  when applied to  $\mathbf{h} \in V$ .

- In Def. 2.4.6  $\|\cdot\|$  can be any norm on  $V$  ( $\rightarrow$  Def. 1.5.65).
- A common shorthand notation for (2.4.7) relies on the “little- $o$ ” Landau symbol:

$$\|F(\mathbf{x} + \mathbf{h}) - F(\mathbf{x}) - D F(\mathbf{x})\mathbf{h}\| = o(\mathbf{h}) \quad \text{for } \mathbf{h} \rightarrow 0,$$

which designates a remainder term tending to 0 as its arguments tends to 0.

- Choosing bases of  $V$  and  $W$ ,  $D F(\mathbf{x})$  can be described by the **Jacobian** (matrix) (2.2.11), because every linear mapping between finite dimensional vector spaces has a matrix representation after bases have been fixed. Thus, the derivative is usually written as a matrix-valued function on  $D$ .

Immediate from Def. 2.4.6 are the following differentiation rules:

- For  $F : V \mapsto W$  linear, we have  $D F(\mathbf{x}) = F$  for all  $\mathbf{x} \in V$

(For instance, if  $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ ,  $F(\mathbf{x}) = \mathbf{A}\mathbf{x}$ ,  $\mathbf{A} \in \mathbb{R}^{m,n}$ , then  $D F(\mathbf{x}) = \mathbf{A}$  for all  $\mathbf{x} \in \mathbb{R}^n$ .)

- Chain rule:** For  $F : V \mapsto W$ ,  $G : W \mapsto U$  sufficiently smooth

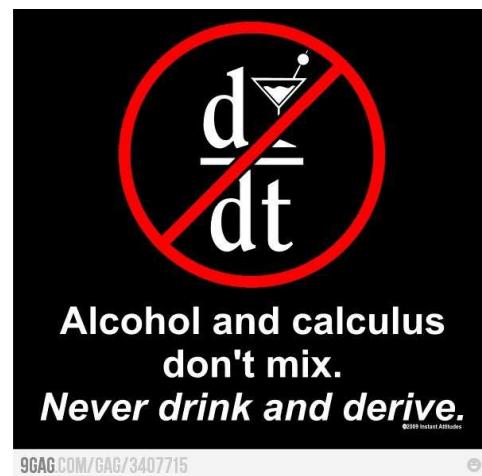
$$D(G \circ F)(\mathbf{x})\mathbf{h} = D G(F(\mathbf{x}))(D F(\mathbf{x}))\mathbf{h}, \quad \mathbf{h} \in V, \mathbf{x} \in D. \quad (2.4.8)$$

- Product rule:**  $F : D \subset V \mapsto W$ ,  $G : D \subset V \mapsto U$  sufficiently smooth,  $b : W \times U \mapsto Z$  bilinear, i.e., linear in each argument:

$$T(\mathbf{x}) = b(F(\mathbf{x}), G(\mathbf{x})) \Rightarrow D T(\mathbf{x})\mathbf{h} = b(D F(\mathbf{x})\mathbf{h}, G(\mathbf{x})) + b(F(\mathbf{x}), D G(\mathbf{x})\mathbf{h}), \quad (2.4.9)$$

$$\mathbf{h} \in V, \mathbf{x} \in D.$$

Advice: If you do not feel comfortable with these rules of multidimensional differential calculus, please resort to detailed componentwise/entrywise calculations according to (2.2.11) (“pedestrian differentiation”), though they may be tedious.



The first and second derivatives of real-valued functions occur frequently and have special names, see [77, Def. 7.3.2] and [77, Satz 7.5.3].

#### Definition 2.4.10. Gradient and Hessian

For sufficiently smooth  $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}$  the **gradient**  $\text{grad } F : D \mapsto \mathbb{R}^n$ , and the **Hessian** (matrix)  $H F(\mathbf{x}) : D \mapsto \mathbb{R}^{n,n}$  are defined as

$$(\text{grad } F(\mathbf{x})^T)\mathbf{h} := D F(\mathbf{x})\mathbf{h}, \quad \mathbf{h}_1^T H F(\mathbf{x})\mathbf{h}_2 := D(D F(\mathbf{x})(\mathbf{h}_1))(\mathbf{h}_2), \quad \mathbf{h}, \mathbf{h}_1, \mathbf{h}_2 \in V.$$

#### Example 2.4.11 (Derivative of a bilinear form)

A simple example:

$$\Psi : \mathbb{R}^n \mapsto \mathbb{R}, \quad \Psi(\mathbf{x}) := \mathbf{x}^T \mathbf{A} \mathbf{x}, \quad \text{with } \mathbf{A} \in \mathbb{R}^{n,n}$$

This is the general matrix representation of a **bilinear form** on  $\mathbb{R}^n$ .

“High level differentiation”: We apply the **product rule** (2.4.9) with  $F, G = Id$ , which means  $D F(\mathbf{x}) = D G(\mathbf{x}) = \mathbf{I}$ , and the **bilinear form**  $b(\mathbf{x}, \mathbf{y}) := \mathbf{x}^T \mathbf{A} \mathbf{y}$ :

$$\blacktriangleright \quad D \Psi(\mathbf{x}) \mathbf{h} = \mathbf{h}^T \mathbf{A} \mathbf{x} + \mathbf{x}^T \mathbf{A} \mathbf{h} = \underbrace{(\mathbf{x}^T \mathbf{A}^\top + \mathbf{x}^T \mathbf{A}) \mathbf{h}}_{=(\mathbf{grad} \Psi(\mathbf{x}))^\top},$$

Hence,  $\mathbf{grad} \Psi(\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top) \mathbf{x}$  according to the definition of a gradient ( $\rightarrow$  Def. 2.4.10)

“Low level differentiation”: Using the rules of matrix  $\times$  vector multiplication,  $\Psi$  can be written in terms of the vector components  $x_i, i = 1, \dots, n$ :

$$\Psi(\mathbf{x}) = \sum_{k=1}^n \sum_{j=1}^n (\mathbf{A})_{k,j} x_k x_j = (\mathbf{A})_{i,i} x_i^2 + \sum_{j=1}^n (\mathbf{A})_{i,j} x_i x_j + \sum_{k=1}^n (\mathbf{A})_{k,i} x_k x_i + \sum_{k=1}^n \sum_{j=1}^n (\mathbf{A})_{k,j} x_k x_j,$$

which leads to the partial derivatives

$$\begin{aligned} \frac{\partial \Psi(\mathbf{x})}{\partial x_i}(\mathbf{x}) &= 2(\mathbf{A})_{i,i} x_i + \sum_{j=1}^n (\mathbf{A})_{i,j} x_j + \sum_{k=1}^n (\mathbf{A})_{k,i} x_k = \sum_{j=1}^n (\mathbf{A})_{i,j} x_j + \sum_{k=1}^n (\mathbf{A})_{k,i} x_k \\ &= (\mathbf{A} \mathbf{x} + \mathbf{A}^\top \mathbf{x})_i. \end{aligned}$$

Of course, both results must agree!

### Example 2.4.12 (Derivative of Euclidean norm)

We seek the derivative of the Euclidean norm, that is, of the function  $F(\mathbf{x}) := \|\mathbf{x}\|_2$ ,  $\mathbf{x} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$  ( $F$  is defined but not differentiable in  $\mathbf{x} = \mathbf{0}$ , just look at the case  $n = 1$ !)

“High level differentiation”: We can write  $F$  as the composition of two functions  $F = G \circ H$  with

$$\begin{aligned} G : \mathbb{R}_+ &\rightarrow \mathbb{R}_+, \quad G(\xi) := \sqrt{\xi}, \\ H : \mathbb{R}^n &\rightarrow \mathbb{R}, \quad H(\mathbf{x}) := \mathbf{x}^T \mathbf{x}. \end{aligned}$$

Using the rule for the differentiation of bilinear forms from Ex. 2.4.11 for the case  $\mathbf{A} = \mathbf{I}$  and basic calculus, we find

$$\begin{aligned} D H(\mathbf{x}) \mathbf{h} &= 2 \mathbf{x}^T \mathbf{h}, \quad \mathbf{x}, \mathbf{h} \in \mathbb{R}^n, \\ D G(\xi) \zeta &= \frac{\zeta}{2\sqrt{\zeta}}, \quad \zeta > 0, \quad \zeta \in \mathbb{R}. \end{aligned}$$

Finally, the **chain rule** (2.4.8) gives

$$\begin{aligned} D F(\mathbf{x}) \mathbf{h} &= D G(H(\mathbf{x}))(D H(\mathbf{x}) \mathbf{h}) = \frac{2 \mathbf{x}^T \mathbf{h}}{2\sqrt{\mathbf{x}^T \mathbf{x}}} = \frac{\mathbf{x}^T}{\|\mathbf{x}\|_2} \cdot \mathbf{h}. \\ \text{Def. 2.4.10} \Rightarrow \mathbf{grad} F(\mathbf{x}) &= \frac{\mathbf{x}}{\|\mathbf{x}\|_2}. \end{aligned} \tag{2.4.13}$$

“Pedestrian differentiation”: For a single component of  $F(\mathbf{x})$  we have

$$\begin{aligned} (F(\mathbf{x}))_i &= \sqrt{x_1^2 + x_2^2 + \cdots + x_i^2 + \cdots + x_n^2}, \\ \Rightarrow \frac{\partial(F(\mathbf{x}))_i}{\partial x_i} &= \frac{1}{2\sqrt{x_1^2 + x_2^2 + \cdots + x_i^2 + \cdots + x_n^2}} \cdot 2x_i \quad [\text{chain rule}]. \\ \Rightarrow \mathbf{grad} F(\mathbf{x}) &= \frac{1}{\|\mathbf{x}\|_2} (x_1, x_2, \dots, x_n)^\top. \end{aligned}$$

#### (2.4.14) A “quasi-linear” system of equations

We call a quasilinear system of equations a non-linear equation of the form

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b} \quad \text{with } \mathbf{b} \in \mathbb{R}^n, \quad (2.4.15)$$

where  $\mathbf{A} : D \subset \mathbb{R}^n \rightarrow \mathbb{R}^{n,n}$  is a matrix-valued function. On other words, a quasi-linear system is a “linear system of equations with solution-dependent system matrix”.

For many quasi-linear systems, for which there exist solutions, the **fixed point iteration** ( $\rightarrow$  Section 2.2)

$$\mathbf{x}^{(k+1)} = \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{b} \Leftrightarrow \mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k+1)} = \mathbf{b}, \quad (2.4.16)$$

provides a convergent iteration, provided that a good initial guess is available.

We can also reformulate

$$(2.4.15) \Leftrightarrow F(\mathbf{x}) = 0 \quad \text{with} \quad F(\mathbf{x}) = \mathbf{A}(\mathbf{x})\mathbf{x} - \mathbf{b}.$$

If  $\mathbf{x} \mapsto \mathbf{A}(\mathbf{x})$  is differentiable, the product rule (2.4.9) yields

$$\mathbf{D}F(\mathbf{x})\mathbf{h} = (\mathbf{D}\mathbf{A}(\mathbf{x})\mathbf{h})\mathbf{x} + \mathbf{A}(\mathbf{x})\mathbf{h}, \quad \mathbf{h} \in \mathbb{R}^n. \quad (2.4.17)$$

Note that  $\mathbf{D}\mathbf{A}(\mathbf{x}^{(k)})$  is a mapping from  $\mathbb{R}^n$  into  $\mathbb{R}^{n,n}$ , which gets  $\mathbf{h}$  as an argument. Then the **Newton iteration** reads

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{s}, \quad \mathbf{D}F(\mathbf{x})\mathbf{s} = (\mathbf{D}\mathbf{A}(\mathbf{x}^{(k)})\mathbf{s})\mathbf{x}^{(k)} + \mathbf{A}(\mathbf{x}^{(k)})\mathbf{s} = \mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}. \quad (2.4.18)$$

#### Example 2.4.19 (A special quasi-linear system of equations)

We consider the quasi-linear system of equations

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b}, \quad \mathbf{A}(\mathbf{x}) = \begin{pmatrix} \gamma(\mathbf{x}) & 1 & & & \\ 1 & \gamma(\mathbf{x}) & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \\ & & & 1 & \gamma(\mathbf{x}) & 1 \\ & & & & 1 & \gamma(\mathbf{x}) \\ & & & & & 1 & \gamma(\mathbf{x}) \end{pmatrix} \in \mathbb{R}^{n \times n}, \quad (2.4.20)$$

where  $\gamma(\mathbf{x}) := 3 + \|\mathbf{x}\|_2$  (Euclidean vector norm), the right hand side vector  $\mathbf{b} \in \mathbb{R}^n$  is given and  $\mathbf{x} \in \mathbb{R}^n$  is unknown.

In order to compute the derivative of  $F(\mathbf{x}) := \mathbf{A}(\mathbf{x})\mathbf{x} - \mathbf{b}$  it is advisable to rewrite

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{T}\mathbf{x} + \mathbf{x}\|\mathbf{x}\|_2, \quad \mathbf{T} := \begin{pmatrix} 3 & 1 & & & \\ 1 & 3 & 1 & & \\ & \ddots & 3 & \ddots & \\ & & \ddots & \ddots & \ddots \\ & & & 1 & 3 & 1 \\ & & & & 1 & 3 \end{pmatrix}.$$

The derivative of the first term is straightforward, because it is linear in  $\mathbf{x}$ , see the discussion following Def. 2.4.6.

The “pedestrian” approach to the second term starts with writing it explicitly in components as

$$(\mathbf{x}\|\mathbf{x}\|)_i = x_i \sqrt{x_1^2 + \cdots + x_n^2}, \quad i = 1, \dots, n.$$

Then we can compute the Jacobian according to (2.2.11) by taking partial derivatives:

$$\begin{aligned} \frac{\partial}{\partial x_i} (\mathbf{x}\|\mathbf{x}\|)_i &= \sqrt{x_1^2 + \cdots + x_n^2} + x_i \frac{x_i}{\sqrt{x_1^2 + \cdots + x_n^2}}, \\ \frac{\partial}{\partial x_j} (\mathbf{x}\|\mathbf{x}\|)_i &= x_i \frac{x_j}{\sqrt{x_1^2 + \cdots + x_n^2}}, \quad j \neq i. \end{aligned}$$

For the “high level” treatment of the second term  $\mathbf{x} \mapsto \mathbf{x}\|\mathbf{x}\|_2$  we apply the product rule (2.4.9), together with (2.4.13):

$$DF(\mathbf{x})\mathbf{h} = \mathbf{T}\mathbf{h} + \|\mathbf{x}\|_2 \mathbf{h} + \mathbf{x} \frac{\mathbf{x}^\top \mathbf{h}}{\|\mathbf{x}\|_2} = \left( \mathbf{A}(\mathbf{x}) + \frac{\mathbf{x}\mathbf{x}^\top}{\|\mathbf{x}\|_2} \right) \mathbf{h}.$$

Thus, in concrete terms the Newton iteration (2.4.18) becomes

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left( \mathbf{A}(\mathbf{x}^{(k)}) + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k)}\|_2} \right)^{-1} (\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}).$$

Note that that matrix of the linear system to be solved in each step is a rank-1-modification (1.6.108) of the symmetric positive definite tridiagonal matrix  $\mathbf{A}(\mathbf{x}^{(k)})$ , cf. Lemma 1.8.12. Thus the Sherman-Morrison-Woodbury formula from Lemma 1.6.113 can be used to solve it efficiently.

### Example 2.4.21 (Derivative of matrix inversion)

We consider matrix inversion as a mapping and (formally) compute its derivative, that is, the derivative of function

$$\text{inv} : \begin{cases} \mathbb{R}_{*}^{n,n} & \rightarrow \mathbb{R}^{n,n} \\ \mathbf{X} & \mapsto \mathbf{X}^{-1} \end{cases},$$

where  $\mathbb{R}_*^{n,n}$  denotes the (open) set of invertible  $n \times n$ -matrices,  $n \in \mathbb{N}$ .

Technique: **implicit differentiation** [77, Sect. 7.8], applied to the equation

$$\text{inv}(\mathbf{X}) \cdot \mathbf{X} = \mathbf{I}, \quad \mathbf{X} \in \mathbb{R}_*^{n,n}.$$

Differentiation on both sides by means of the product rule (2.4.9) yields

$$\begin{aligned} \mathbf{D} \text{inv}(\mathbf{X}) \mathbf{H} \cdot \mathbf{X} + \text{inv}(\mathbf{X}) \cdot \mathbf{H} &= \mathbf{0}, \quad \mathbf{H} \in \mathbb{R}^{n,n}, \\ \Rightarrow \quad \mathbf{D} \text{inv}(\mathbf{X}) \mathbf{H} &= -\mathbf{X}^{-1} \mathbf{H} \mathbf{X}^{-1}, \quad \mathbf{H} \in \mathbb{R}^{n,n}. \end{aligned} \quad (2.4.22)$$

For  $n = 1$  we get  $\mathbf{D} \text{inv}(x)h = -\frac{h}{x^2}$ , which recovers the well-known derivative of the function  $x \rightarrow x^{-1}$ .

### Example 2.4.23 (Matrix inversion by means of Newton's method → [61, 3])

Given a regular matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$ , its inverse can be defined as the unique zero of a function:

$$\mathbf{X} = \mathbf{X}^{-1} \iff F(\mathbf{X}) = 0 \quad \text{for } F : \left\{ \begin{array}{ccc} \mathbb{R}_*^{n,n} & \xrightarrow{\quad} & \mathbb{R}^{n,n} \\ \mathbf{X} & \mapsto & \mathbf{A} - \mathbf{X}^{-1} \end{array} \right..$$

Using (2.4.22) we find for the derivative of  $F$  in  $\mathbf{X} \in \mathbb{R}_*^{n,n}$

$$\mathbf{D} F(\mathbf{X}) \mathbf{H} = \mathbf{X}^{-1} \mathbf{H} \mathbf{X}^{-1}, \quad \mathbf{H} \in \mathbb{R}^{n,n}. \quad (2.4.24)$$

► The abstract Newton iteration (2.4.1) for  $F$  reads

$$\mathbf{X}^{(k+1)} = \mathbf{X}^{(k)} - \mathbf{S}, \quad \mathbf{S} := \mathbf{D} F(\mathbf{X}^{(k)})^{-1} F(\mathbf{X}^{(k)}). \quad (2.4.25)$$

The **Newton correction**  $\mathbf{S}$  in the  $k$ -th step solves the *linear system of equations*

$$\begin{aligned} \mathbf{D} F(\mathbf{X}^{(k)}) \mathbf{S} &\stackrel{(2.4.24)}{=} (\mathbf{X}^{(k)})^{-1} \mathbf{S} (\mathbf{X}^{(k)})^{-1} = F(\mathbf{X}^{(k)}) = \mathbf{A} - (\mathbf{X}^{(k)})^{-1}. \\ \Rightarrow \quad \mathbf{S} &= \mathbf{X}^{(k)} (\mathbf{A} - (\mathbf{X}^{(k)})^{-1}) \mathbf{X}^{(k)} = \mathbf{X}^{(k)} \mathbf{A} \mathbf{X}^{(k)} - \mathbf{X}^{(k)}. \end{aligned} \quad (2.4.26)$$

in (2.4.25)

$$\mathbf{X}^{(k+1)} = \mathbf{X}^{(k)} - (\mathbf{X}^{(k)} \mathbf{A} \mathbf{X}^{(k)} - \mathbf{X}^{(k)}) = \mathbf{X}^{(k)} (2\mathbf{I} - \mathbf{A} \mathbf{X}^{(k)}). \quad (2.4.27)$$

To study the convergence of this iteration we derive a recursion for the iteration errors  $\mathbf{E}^{(k)} := \mathbf{X}^{(k)} - \mathbf{A}^{-1}$ :

$$\begin{aligned} \mathbf{E}^{(k+1)} &= \mathbf{X}^{(k+1)} - \mathbf{A}^{-1} \stackrel{(2.4.27)}{=} \mathbf{X}^{(k)} (2\mathbf{I} - \mathbf{A} \mathbf{X}^{(k)}) - \mathbf{A}^{-1} = (\mathbf{E}^{(k)} + \mathbf{A}^{-1})(2\mathbf{I} - \mathbf{A}(\mathbf{E}^{(k)} + \mathbf{A}^{-1})) - \mathbf{A}^{-1} \\ &= (\mathbf{E}^{(k)} + \mathbf{A}^{-1})(\mathbf{I} - \mathbf{A}\mathbf{E}^{(k)}) - \mathbf{A}^{-1} = -\mathbf{E}^{(k)} \mathbf{A} \mathbf{E}^{(k)}. \end{aligned}$$

For the norm of the iteration error (a matrix norm → Def. 1.5.71) we conclude from submultiplicativity (1.5.72) a recursive estimate

$$\|\mathbf{E}^{(k+1)}\| \leq \|\mathbf{E}^{(k)}\|^2 \|\mathbf{A}\|. \quad (2.4.28)$$

This holds for any matrix norm according to Def. 1.5.71, which is induced by a vector norm. For the relative iteration error we obtain

$$\underbrace{\frac{\|\mathbf{E}^{(k+1)}\|}{\|\mathbf{A}\|}}_{\text{relative error}} \leq \left( \underbrace{\frac{\|\mathbf{E}^{(k)}\|}{\|\mathbf{A}\|}}_{\text{relative error}} \right)^2 \underbrace{\|\mathbf{A}\| \|\mathbf{A}^{-1}\|}_{=\text{cond}(\mathbf{A})}, \quad (2.4.29)$$

where the **condition number** is defined in Def. 1.6.15.

From (2.4.28) we conclude that the iteration will converge ( $\lim_{k \rightarrow \infty} \|\mathbf{E}^{(k)}\| = 0$ ), if

$$\|\mathbf{E}^{(0)} \mathbf{A}\| = \|\mathbf{X}^{(0)} \mathbf{A} - \mathbf{I}\| < 1, \quad (2.4.30)$$

which gives a condition on the initial guess  $\mathbf{S}^{(0)}$ . Now let us consider the Euclidean matrix norm  $\|\cdot\|_2$ , which can be expressed in terms of eigenvalues, see Cor. 1.5.77. Motivated by this relationship, we use the initial guess  $\mathbf{X}^{(0)} = \alpha \mathbf{A}^\top$  with  $\alpha > 0$  still to be determined.

$$\|\mathbf{X}^{(0)} \mathbf{A} - \mathbf{I}\|_2 = \|\alpha \mathbf{A}^\top \mathbf{A} - \mathbf{I}\|_2 = \alpha \|\mathbf{A}\|_2^2 - 1 \stackrel{!}{<} 1 \Leftrightarrow \alpha < \frac{2}{\|\mathbf{A}\|_2^2},$$

which is a sufficient condition for the initial guess  $\mathbf{X}^{(0)} = \alpha \mathbf{A}^\top$ , in order to make (2.4.27) converge. In this case we infer **quadratic convergence** from both (2.4.28) and (2.4.29).

### Remark 2.4.31 (Simplified Newton method [15, Sect. 5.6.2])

#### MATLAB-code 2.4.32: Efficient implementation of simplified Newton method

```

1 function x = simpnewton(x,F,DF,rtol,atol)
2 % MATLAB template for simplified Newton method
3 [L,U] = lu(DF(x)); % one LU-decomposition
4 s = U\ (L\ F(x)); x = x-s;
5 % termination based on relative and absolute tolerance
6 ns = norm(s); nx = norm(x);
7 while ((ns > rtol*nx) && (ns > atol))
8     s = U\ (L\ F(x)); x = x-s;
9 end
```

Simplified Newton Method: ➤ use the same Jacobian  $D\mathbf{F}(\mathbf{x}^{(k)})$  for all/several steps

► Possible to reuse of LU-decomposition, cf. Rem. 1.6.87.

➤ (usually) merely linear convergence instead of quadratic convergence

### Remark 2.4.33 (Numerical Differentiation for computation of Jacobian)

If  $D\mathbf{F}(\mathbf{x})$  is not available (e.g. when  $F(\mathbf{x})$  is given only as a procedure) we may resort to approximation by difference quotients:

Numerical Differentiation: 
$$\frac{\partial F_i}{\partial x_j}(\mathbf{x}) \approx \frac{F_i(\mathbf{x} + h\vec{e}_j) - F_i(\mathbf{x})}{h}.$$

Caution: Roundoff errors wreak havoc for small  $h \rightarrow$  Ex. 1.5.43 ! Therefore use  $h \approx \sqrt{\text{EPS}}$ .

## 2.4.2 Convergence of Newton's method

Newton iteration (2.4.1)  $\hat{=}$  fixed point iteration ( $\rightarrow$  Section 2.2) with iteration function

$$\Phi(\mathbf{x}) = \mathbf{x} - D F(\mathbf{x})^{-1} F(\mathbf{x}) .$$

[“product rule” (2.4.9) :  $D \Phi(\mathbf{x}) = \mathbf{I} - D(\{\mathbf{x} \mapsto D F(\mathbf{x})^{-1}\})F(\mathbf{x}) - D F(\mathbf{x})^{-1} D F(\mathbf{x})$  ]

$F(\mathbf{x}^*) = 0 \Rightarrow D \Phi(\mathbf{x}^*) = 0 ,$

that is, the derivative (Jacobian) of the iteration function of the Newton fixed point iteration vanishes in the limit point. Thus from Lemma 2.2.18 we draw the same conclusion as in the scalar case  $n = 1$ , cf. Section 2.3.2.1.

*Local* quadratic convergence of Newton's method, if  $D F(\mathbf{x}^*)$  regular

### Experiment 2.4.34 (Convergence of Newton's method in 2D)

We study the convergence of Newton's method empirically for  $n = 2$  for

$$F(\mathbf{x}) = \begin{bmatrix} x_1^2 - x_2^4 \\ x_1 - x_2^3 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2 \quad \text{with solution} \quad F\left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}\right) = 0 . \quad (2.4.35)$$

Jacobian (analytic computation):  $D F(\mathbf{x}) = \begin{bmatrix} \partial_{x_1} F_1(x) & \partial_{x_2} F_1(x) \\ \partial_{x_1} F_2(x) & \partial_{x_2} F_2(x) \end{bmatrix} = \begin{bmatrix} 2x_1 & -4x_2^3 \\ 1 & -3x_2^2 \end{bmatrix}$

Realization of Newton iteration (2.4.1):

1. Solve LSE

$$\begin{bmatrix} 2x_1 & -4x_2^3 \\ 1 & -3x_2^2 \end{bmatrix} \Delta \mathbf{x}^{(k)} = - \begin{bmatrix} x_1^2 - x_2^4 \\ x_1 - x_2^3 \end{bmatrix} ,$$

where  $\mathbf{x}^{(k)} = [x_1, x_2]^T$ .

2. Set  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta \mathbf{x}^{(k)}$ .

#### MATLAB-code 2.4.36: Newton iteration for (2.4.35)

```

1 F=@(x) [x(1)^2-x(2)^4; x(1)-x(2)^3];
2 DF=@(x) [2*x(1),-4*x(2)^3;1,-3*x(2)^2];
3 x=[0.7;0.7]; x_ast=[1;1]; tol=1E-10;
4
5 res=[0,x',norm(x-x_ast)];
6 s = DF(x)\F(x); x = x-s;
7 res = [res; 1,x',norm(x-x_ast)]; k=2;
8 while (norm(s) > tol*norm(x))
9     s = DF(x)\F(x); x = x-s;
10    res = [res; k,x',norm(x-x_ast)];
11    k = k+1;

```

```

12 | end
13 |
14 | ld = diff(log(res(:, 4))); %
15 | rates = ld(2:end) ./ ld(1:end-1); %

```

Line 14, Line 15: estimation of order of convergence, see Rem. 2.1.19.

$k$	$\mathbf{x}^{(k)}$	$\epsilon_k := \ \mathbf{x}^* - \mathbf{x}^{(k)}\ _2$	$\frac{\log \epsilon_{k+1} - \log \epsilon_k}{\log \epsilon_k - \log \epsilon_{k-1}}$
0	$[0.7, 0.7]^T$	4.24e-01	
1	$[0.878500000000000, 1.064285714285714]^T$	1.37e-01	1.69
2	$[1.01815943274188, 1.00914882463936]^T$	2.03e-02	2.23
3	$[1.00023355916300, 1.00015913936075]^T$	2.83e-04	2.15
4	$[1.0000000583852, 1.00000002726552]^T$	2.79e-08	1.77
5	$[0.999999999999998, 1.000000000000000]^T$	2.11e-15	
6	$[1, 1]^T$		

☞ (Some) evidence of **quadratic convergence**, see Rem. 2.1.19.

There is a sophisticated theory about the convergence of Newton's method. For example one can find the following theorem in [19, Thm. 4.10], [20, Sect. 2.1]):

### Theorem 2.4.37. Local quadratic convergence of Newton's method

If:

- (A)  $D \subset \mathbb{R}^n$  open and convex,
- (B)  $F : D \mapsto \mathbb{R}^n$  continuously differentiable,
- (C)  $D F(\mathbf{x})$  regular  $\forall \mathbf{x} \in D$ ,
- (D)  $\exists L \geq 0: \|D F(\mathbf{x})^{-1}(D F(\mathbf{x} + \mathbf{v}) - D F(\mathbf{x}))\|_2 \leq L \|\mathbf{v}\|_2 \quad \forall \mathbf{v} \in \mathbb{R}^n, \mathbf{v} + \mathbf{x} \in D, \quad \forall \mathbf{x} \in D$ ,
- (E)  $\exists \mathbf{x}^*: F(\mathbf{x}^*) = 0$  (*existence of solution in D*)
- (F) initial guess  $\mathbf{x}^{(0)} \in D$  satisfies  $\rho := \|\mathbf{x}^* - \mathbf{x}^{(0)}\|_2 < \frac{2}{L} \quad \wedge \quad B_\rho(\mathbf{x}^*) \subset D$ .

then the Newton iteration (2.4.1) satisfies:

- (i)  $\mathbf{x}^{(k)} \in B_\rho(\mathbf{x}^*) := \{\mathbf{y} \in \mathbb{R}^n, \|\mathbf{y} - \mathbf{x}^*\| < \rho\}$  for all  $k \in \mathbb{N}$ ,
- (ii)  $\lim_{k \rightarrow \infty} \mathbf{x}^{(k)} = \mathbf{x}^*$ ,
- (iii)  $\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\|_2 \leq \frac{L}{2} \|\mathbf{x}^{(k)} - \mathbf{x}^*\|_2^2 \quad (\text{local quadratic convergence})$ .

☞ notation: ball  $B_\rho(\mathbf{z}) := \{\mathbf{x} \in \mathbb{R}^n: \|\mathbf{x} - \mathbf{z}\|_2 \leq \rho\}$

Terminology: (D)  $\hat{=}$  affine invariant **Lipschitz condition**

 Usually, it is hardly possible to verify the assumptions of the theorem for a concrete non-linear system of equations, because neither  $L$  nor  $\mathbf{x}^*$  are known.

► In general: a priori estimates as in Thm. 2.4.37 are of little practical relevance.

### 2.4.3 Termination of Newton iteration

An abstract discussion of ways to stop iterations for solving  $F(\mathbf{x}) = 0$  was presented in Section 2.1.2, with “ideal termination” ( $\rightarrow \S\ 2.1.24$ ) as ultimate, but unfeasible, goal.

Yet, in 2.4.2 we saw that Newton’s method enjoys (asymptotic) quadratic convergence, which means rapid decrease of the relative error of the iterates, once we are close to the solution, which is exactly the point, when we want to **STOP**. As a consequence, **asymptotically**, the Newton correction (difference of two consecutive iterates) yields rather precise information about the size of the error:

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \ll \|\mathbf{x}^{(k)} - \mathbf{x}^*\| \Rightarrow \|\mathbf{x}^{(k)} - \mathbf{x}^*\| \approx \|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|. \quad (2.4.38)$$

This suggests the following **correction based** termination criterion:

$$\begin{aligned} \text{STOP, as soon as } \|\Delta\mathbf{x}^{(k)}\| &\leq \tau_{\text{rel}} \|\mathbf{x}^{(k)}\| \quad \text{or} \quad \|\Delta\mathbf{x}^{(k)}\| \leq \tau_{\text{abs}}, \\ \text{with Newton correction } \Delta\mathbf{x}^{(k)} &:= \mathbf{D}F(\mathbf{x}^{(k)})^{-1}F(\mathbf{x}^{(k)}). \end{aligned} \quad (2.4.39)$$

Here,  $\|\cdot\|$  can be any suitable vector norm,  $\tau_{\text{rel}} \hat{=} \text{relative tolerance}$ ,  $\tau_{\text{abs}} \hat{=} \text{absolute tolerance}$ , see  $\S\ 2.1.24$ .

➤ quit iterating as soon as  $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| = \|\mathbf{D}F(\mathbf{x}^{(k)})^{-1}F(\mathbf{x}^{(k)})\| < \tau \|\mathbf{x}^{(k)}\|$ ,  
with  $\tau = \text{tolerance}$

→ uneconomical: one needless update, because  $\mathbf{x}^{(k)}$  would already be accurate enough.

#### Remark 2.4.40 (Newton’s iteration; computational effort and termination)

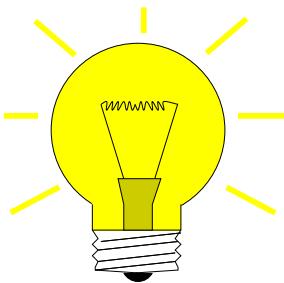
Some facts about the Newton method for solving **large** ( $n \gg 1$ ) non-linear systems of equations:

- ☛ Solving the linear system to compute the Newton correction may be expensive (asymptotic computational effort  $\mathcal{O}(n^3)$  for direct elimination  $\rightarrow \S\ 1.6.24$ ) and accounts for the bulk of numerical cost of a single step of the iteration.
- ☛ In applications only **very few steps** of the iteration will be needed to achieve the desired accuracy due to fast quadratic convergence.
- ▷ The termination criterion (2.4.39) computes the last Newton correction  $\Delta\mathbf{x}^{(k)}$  needlessly, because  $\mathbf{x}^{(k)}$  already accurate enough!

Therefore we would like to use an a-posteriori termination criterion that dispenses with computing (and “inverting”) another Jacobian  $\mathbf{D}F(\mathbf{x}^{(k)})$  just to tell us that  $\mathbf{x}^{(k)}$  is already accurate enough.

#### (2.4.41) Termination of Newton iteration based on simplified Newton correction

Due to fast asymptotic quadratic convergence, we can expect  $\mathbf{D}F(\mathbf{x}^{(k-1)}) \approx \mathbf{D}F(\mathbf{x}^{(k)})$  during the final steps of the iteration.



Idea: Replace  $D F(\mathbf{x}^{(k)})$  with  $D F(\mathbf{x}^{(k-1)})$  in any correction based termination criterion.

Rationale: LU-decomposition of  $D F(\mathbf{x}^{(k-1)})$  is already available ➤ less effort.

Terminology:  $\Delta \bar{\mathbf{x}}^{(k)} := D F(\mathbf{x}^{(k-1)})^{-1} F(\mathbf{x}^{(k)})$  ≈ simplified Newton correction

► Economical correction based termination criterion for Newton's method:

$$\text{STOP, as soon as } \|\Delta \bar{\mathbf{x}}^{(k)}\| \leq \tau_{\text{rel}} \|\mathbf{x}^{(k)}\| \quad \text{or} \quad \|\Delta \bar{\mathbf{x}}^{(k)}\| \leq \tau_{\text{abs}}, \quad (2.4.42)$$

with Newton correction  $\Delta \mathbf{x}^{(k)} := D F(\mathbf{x}^{(k)})^{-1} F(\mathbf{x}^{(k)})$ .

Note that (2.4.42) is affine invariant → Rem. 2.4.4.

Effort: Reuse of LU-factorization (→ Rem. 1.6.87) of  $D F(\mathbf{x}^{(k-1)})$  ➤  $\Delta \bar{\mathbf{x}}^{(k)}$  available with  $O(n^2)$  operations

#### C++11 code 2.4.43: Generic Newton iteration with termination criterion (2.4.42)

```

1 template <typename FuncType, typename JacType, typename VecType>
2 void newton_stc(const FuncType &F, const JacType &DF,
3                  VecType &x, double rtol, double atol)
4 {
5     using scalar_t = typename VecType::Scalar;
6     scalar_t sn;
7     do {
8         auto jacfac = DF(x).lu(); // LU-factorize Jacobian ]
9         x -= jacfac.solve(F(x)); // Compute next iterate
10        // Compute norm of simplified Newton correction
11        sn = jacfac.solve(F(x)).norm();
12    }
13    // Termination based on simplified Newton correction
14    while ((sn > rtol*x.norm()) && (sn > atol));
15 }
```

#### Remark 2.4.44 (Residual based termination of Newton's method)

If we used the residual based termination criterion

$$\|F(\mathbf{x}^{(k)})\| \leq \tau,$$

then the resulting algorithm would not be affine invariant, because for  $F(\mathbf{x}) = 0$  and  $\mathbf{A}F(\mathbf{x}) = 0$ ,  $\mathbf{A} \in \mathbb{R}^{n,n}$  regular, the Newton iteration might terminate with different iterates.

### Summary: Newton's method



converges *asymptotically* very fast: doubling of number of significant digits in each step



often a very small region of convergence, which requires an initial guess rather close to the solution.

### 2.4.4 Damped Newton method

Potentially big problem: Newton method converges quadratically, but only *locally*, which may render it useless, if convergence is guaranteed only for initial guesses very close to exact solution, see also Ex. 2.3.32.

In this section we study a method to enlarge the region of convergence, at the expense of quadratic convergence, of course.

#### Example 2.4.46 (Local convergence of Newton's method)

The dark side of local convergence (→ Def. 2.1.8): for many initial guesses  $x^{(0)}$  Newton's method will not converge!

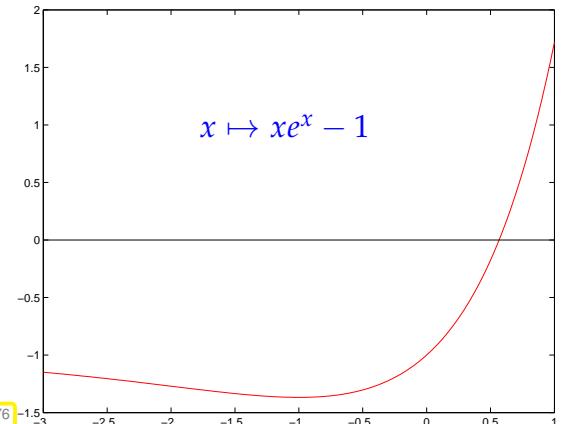
In 1D two main causes can be identified:

- ① “Wrong direction” of Newton correction:

$$F(x) = xe^x - 1 \Rightarrow F'(-1) = 0$$

$$\begin{aligned} x^{(0)} < -1 &\Rightarrow x^{(k)} \rightarrow -\infty, \\ x^{(0)} > -1 &\Rightarrow x^{(k)} \rightarrow x^*, \end{aligned}$$

because all Newton corrections for  $x^{(k)} < -1$  make the iterates decrease even further.

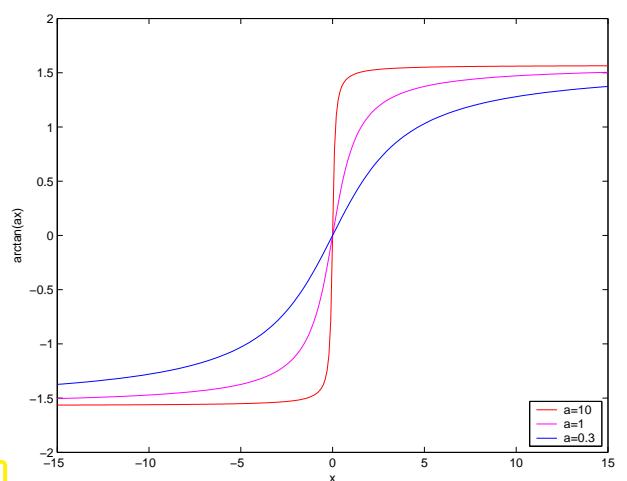


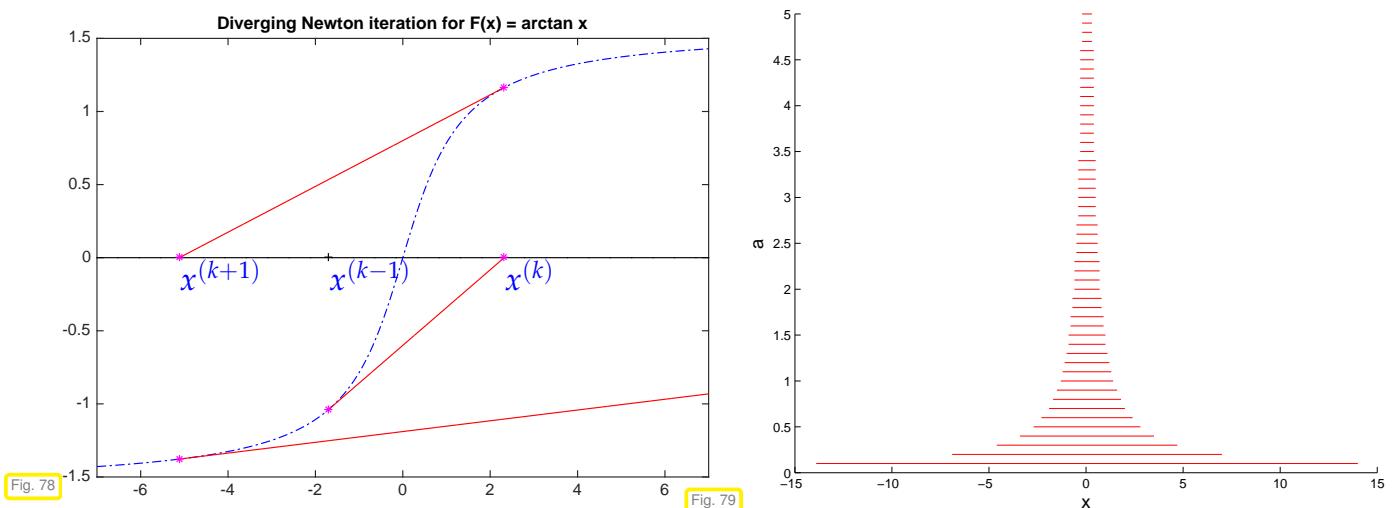
- ② Newton correction is too large:

$$F(x) = \arctan(ax), \quad a > 0, x \in \mathbb{R}$$

with zero  $x^* = 0$ .

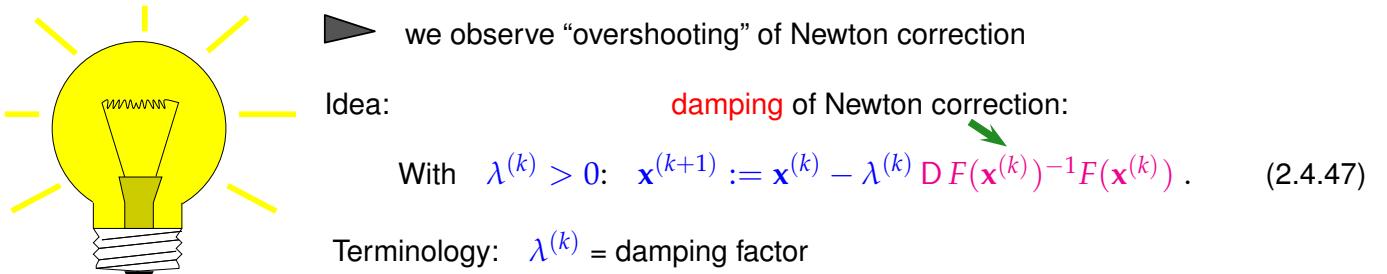
If  $x^{(k)}$  is located where the function is “flat”, the intersection of the tangents with the  $x$ -axis is “far out”, see Fig. 78.





In Fig. 79 the red zone =  $\{x^{(0)} \in \mathbb{R}, x^{(k)} \rightarrow 0\}$ , domain of initial guesses for which Newton's method converges.

If the Newton correction points in the wrong direction (Item ①), no general remedy is available. If the Newton correction is too large (Item ②), there is an effective cure:



### Affine invariant damping strategy

Choice of damping factor: affine invariant natural monotonicity test [20, Ch. 3]:

$$\text{choose “maximal” } 0 < \lambda^{(k)} \leq 1: \quad \left\| \Delta \bar{x}(\lambda^{(k)}) \right\| \leq \left(1 - \frac{\lambda^{(k)}}{2}\right) \left\| \Delta x^{(k)} \right\|_2 \quad (2.4.49)$$

where  $\Delta x^{(k)} := D F(x^{(k)})^{-1} F(x^{(k)})$  → current Newton correction ,

$\Delta \bar{x}(\lambda^{(k)}) := D F(x^{(k)})^{-1} F(x^{(k)} + \lambda^{(k)} \Delta x^{(k)})$  → tentative simplified Newton correction .

Heuristics behind control of damping:

- \* When the method converges  $\Leftrightarrow$  size of Newton correction decreases  $\Leftrightarrow$  (2.4.49) satisfied.
- \* In the case of strong damping ( $\lambda^{(k)} \ll 1$ ) the size of the Newton correction cannot be expected to shrink significantly, since iterates do not change much  $>$  factor  $(1 - \frac{1}{2}\lambda^{(k)})$  in (2.4.49).

Note: As before, reuse of LU-factorization in the computation of  $\Delta x^{(k)}$  and  $\Delta \bar{x}(\lambda^{(k)})$ .

## MATLAB-code Damped Newton method

```

1 function [x,cvg] =
2   dampnewton(x,F,DF,rtol,atol)
3   [L,U] = lu(DF(x)); s = U\ (L\f(x));
4   xn = x-s; lambda = 1; cvg = 0;
5   f = F(xn); st = U\ (L\f); stn = norm(st);
6   while ((stn>rtol*norm(xn)) && (stn > atol))
7     while (norm(st) > (1-lambda/2)*norm(s))
8       lambda = lambda/2;
9       if (lambda < LMIN), cvg = -1; return; end
10      xn = x-lambda*s; f = F(xn);
11      st = U\ (L\f);
12    end
13    x = xn; [L,U] = lu(DF(x)); s = U\ (L\f);
14    lambda = min(2*lambda,1);
15    xn = x-lambda*s; f = F(xn); st = U\ (L\f);
16  end
17  x = xn;

```

Reuse of LU-factorization, see Rem. 1.6.87

a-posteriori termination criterion (based on simplified Newton correction, cf. Section 2.4.3)

Natural monotonicity test (2.4.49)

Reduce damping factor  $\lambda$

Note: LU-factorization of Jacobi matrix  $DF(\mathbf{x}^{(k)})$  is done once per *successful* iteration step (Line 12 of the above code) and reused for the computation of the simplified Newton correction in Line 10, Line 14 of the above MATLAB code.

Policy: Reduce damping factor by a factor  $q \in ]0,1[$  (usually  $q = \frac{1}{2}$ ) until the affine invariant natural monotonicity test (2.4.49) passed, see Line 13 in the above MATLAB code.

**C++11 code 2.4.50: Generic damped Newton method based on natural monotonicity test**

```

1  template <typename FuncType,typename JacType,typename VecType>
2  void dampnewton(const FuncType &F,const JacType &DF,
3                  VecType &x,double rtol,double atol)
4  {
5      using index_t = typename VecType::Index;
6      using scalar_t = typename VecType::Scalar;
7      const index_t n = x.size();
8      const scalar_t lmin = 1E-3; // Minimal damping factor
9      scalar_t lambda = 1.0; // Initial and actual damping factor
10     VecType s(n),st(n); // Newton corrections
11     VecType xn(n); // Tentative new iterate
12     scalar_t sn,stn; // Norms of Newton corrections
13
14     do {
15         auto jacfac = DF(x).lu(); // LU-factorize Jacobian
16         s = jacfac.solve(F(x)); // Newton correction
17         sn = s.norm(); // Norm of Newton correction
18         lambda *= 2.0;
19         do {
20             lambda /= 2;
21             if (lambda < lmin) throw "No convergence: lambda -> 0";
22             xn = x-lambda*s; // Tentative next iterate
23             st = jacfac.solve(F(xn)); // Simplified Newton correction
24             stn = st.norm();
25         }
26         while (stn > (1-lambda/2)*sn); // Natural monotonicity test
27         x = xn; // Now: xn accepted as new iterate
28         lambda = std::min(2.0*lambda,1.0); // Try to mitigate damping
29     }
30     // Termination based on simplified Newton correction
31     while ((stn > rtol*x.norm()) && (stn > atol));
32 }
```

The arguments for Code 2.4.50 are the same as for Code 2.4.43. As termination criterion is uses (2.4.42). Note that all calls to `solve` boil down to forward/backward elimination for triangular matrices and incur cost of  $O(n^2)$  only.

**Experiment 2.4.51 (Damped Newton method)**

We test the damped Newton method for Item ② of Ex. 2.4.46, where excessive Newton corrections made Newton's method fail.

$$F(x) = \arctan(x),$$

- $x^{(0)} = 20$

- $q = \frac{1}{2}$

- $\text{LMIN} = 0.001$

We observe that damping is effective and asymptotic quadratic convergence is recovered.

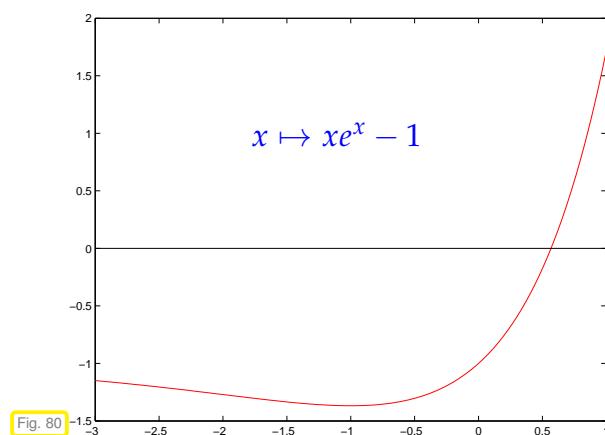
$k$	$\lambda^{(k)}$	$x^{(k)}$	$F(x^{(k)})$
1	0.03125	0.94199967624205	0.75554074974604
2	0.06250	0.85287592931991	0.70616132170387
3	0.12500	0.70039827977515	0.61099321623952
4	0.25000	0.47271811131169	0.44158487422833
5	0.50000	0.20258686348037	0.19988168667351
6	1.00000	-0.00549825489514	-0.00549819949059
7	1.00000	0.00000011081045	0.00000011081045
8	1.00000	-0.0000000000000001	-0.0000000000000001

### Experiment 2.4.52 (Failure of damped Newton method)

We examine the effect of damping in the case of Item ① of Ex. 2.4.46.

- \* As in Ex. 2.4.46:  
 $F(x) = xe^x - 1,$

- \* Initial guess for damped Newton method  $x^{(0)} = -1.5$



Observation:

Newton correction pointing in “wrong direction”

► no convergence despite damping

$k$	$\lambda^{(k)}$	$x^{(k)}$	$F(x^{(k)})$
1	0.25000	-4.4908445351690	-1.0503476286303
2	0.06250	-6.1682249558799	-1.0129221310944
3	0.01562	-7.6300006580712	-1.0037055902301
4	0.00390	-8.8476436930246	-1.0012715832278
5	0.00195	-10.5815494437311	-1.0002685596314
Bailed out because of lambda < LMIN !			

## 2.4.5 Quasi-Newton Method



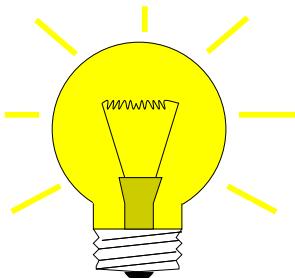
*Supplementary reading.* For related expositions refer to [63, Sect. 7.1.4], [85, 2.3.2].

How can we solve  $F(\mathbf{x}) = \mathbf{0}$  iteratively, in case  $D F(\mathbf{x})$  is not available and numerical differentiation (see Rem. 2.4.33) is too expensive?

In 1D ( $n = 1$ ) we can choose among many derivative-free methods that rely on  $F$ -evaluations alone, for instance the secant method (2.3.24) from Section 2.3.2.3:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \frac{F(\mathbf{x}^{(k)})(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)})}{F(\mathbf{x}^{(k)}) - F(\mathbf{x}^{(k-1)})}. \quad (2.3.24)$$

Recall that the secant method converges locally with order  $p \approx 1.6$  and beats Newton's method in terms of efficiency ( $\rightarrow$  Section 2.3.3).

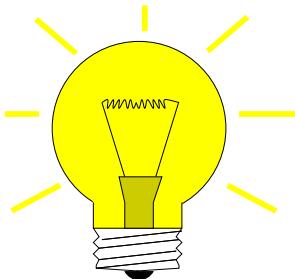


Comparing with (2.3.4) we realize that this iteration amounts to a “Newton-type iteration” with the approximation

$$F'(x^{(k)}) \approx \frac{F(x^{(k)}) - F(x^{(k-1)})}{x^{(k)} - x^{(k-1)}} \quad \begin{matrix} \text{“difference quotient”} \\ \text{already computed!} \end{matrix} \rightarrow \text{cheap} \quad (2.4.53)$$



Not clear how to generalize the secant method to  $n > 1$ ?



Idea: rewrite (2.4.53) as a **secant condition** for an approximation  $J_k \approx D F(x^{(k)})$ ,  $x^{(k)} \doteq$  iterate:

$$J_k(x^{(k)} - x^{(k-1)}) = F(x^{(k)}) - F(x^{(k-1)}). \quad (2.4.54)$$

$$\Rightarrow \text{Iteration: } x^{(k+1)} := x^{(k)} - J_k^{-1}F(x^{(k)}). \quad (2.4.55)$$



However, many matrices  $J_k$  fulfill (2.4.54)!

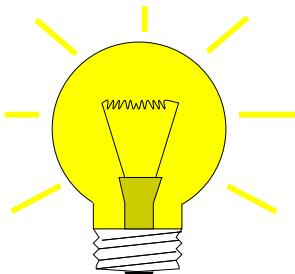


We need extra conditions to fix  $J_k \in \mathbb{R}^{n,n}$ .

Reasoning: If we assume that  $J_k$  is a good approximation of  $D F(x^{(k)})$ , then it would be foolish not to use the information contained in  $J_k$  for the construction of  $J_k$ .

$\Rightarrow$  Guideline: obtain  $J_k$  through a “small” **modification** of  $J_{k-1}$  compliant with (2.4.54)

What can “small modification” mean: Demand that  $J_k$  acts like  $J_{k-1}$  on a complement of the span of  $x^{(k)} - x^{(k-1)}$ !



$\Rightarrow$  Broyden's conditions:  $J_k z = J_{k-1} z \quad \forall z: z \perp (x^{(k)} - x^{(k-1)})$ . (2.4.56)

i.e.:

$$J_k := J_{k-1} + \frac{F(x^{(k)})(x^{(k)} - x^{(k-1)})^\top}{\|x^{(k)} - x^{(k-1)}\|_2^2} \quad (2.4.57)$$

- \* The conditions (2.4.54) and (2.4.56) uniquely define  $J_k$
- \* The update formula (2.4.57) means that  $J_k$  is spawned by a **rank-1-modification** of  $J_{k-1}$ .

Final form of Broyden's quasi-Newton method for solving  $F(\mathbf{x}) = 0$ :

$$\begin{aligned}\mathbf{x}^{(k+1)} &:= \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}, \quad \Delta\mathbf{x}^{(k)} := -\mathbf{J}_k^{-1}F(\mathbf{x}^{(k)}), \\ \mathbf{J}_{k+1} &:= \mathbf{J}_k + \frac{F(\mathbf{x}^{(k+1)})(\Delta\mathbf{x}^{(k)})^\top}{\|\Delta\mathbf{x}^{(k)}\|_2^2}.\end{aligned}\tag{2.4.58}$$

To start the iteration we have to initialize  $\mathbf{J}_0$ , e.g. with the exact Jacobi matrix  $\mathbf{D}F(\mathbf{x}^{(0)})$ .

**Remark 2.4.59 (Minimality property of Broyden's rank-1-modification)**

in another sense  $\mathbf{J}_l$  is closest to  $\mathbf{J}_{k-1}$  under the constraint of the secant condition (2.4.54):

Let  $\mathbf{x}^{(k)}$  and  $\mathbf{J}_k$  be the iterates and matrices, respectively, from Broyden's method (2.4.58), and let  $\mathbf{J} \in \mathbb{R}^{n,n}$  satisfy the same secant condition (2.4.54) as  $\mathbf{J}_{k+1}$ :

$$\mathbf{J}(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = F(\mathbf{x}^{(k+1)}) - F(\mathbf{x}^{(k)}).\tag{2.4.60}$$

Then from  $\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} = -\mathbf{J}_k^{-1}F(\mathbf{x}^{(k)})$  we obtain

$$(\mathbf{I} - \mathbf{J}_k^{-1}\mathbf{J})(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = -\mathbf{J}_k^{-1}F(\mathbf{x}^{(k)}) - \mathbf{J}_k^{-1}(F(\mathbf{x}^{(k+1)}) - F(\mathbf{x}^{(k)})) = -\mathbf{J}_k^{-1}F(\mathbf{x}^{(k+1)}).\tag{2.4.61}$$

From this we get the identity

$$\begin{aligned}\mathbf{I} - \mathbf{J}_k^{-1}\mathbf{J}_{k+1} &= \mathbf{I} - \mathbf{J}_k^{-1}\left(\mathbf{J}_k + \frac{F(\mathbf{x}^{(k+1)})(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_2^2}\right) \\ &= -\mathbf{J}_k^{-1}F(\mathbf{x}^{(k+1)})\frac{(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_2^2} = \\ &\stackrel{(2.4.61)}{=} (\mathbf{I} - \mathbf{J}_k^{-1}\mathbf{J})\frac{(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_2^2}.\end{aligned}$$

Using the submultiplicative property (1.5.72) of the Euclidean matrix norm, we conclude

$$\left\| \mathbf{I} - \mathbf{J}_k^{-1}\mathbf{J}_{k+1} \right\| \leq \left\| \mathbf{I} - \mathbf{J}_k^{-1}\mathbf{J} \right\|, \text{ because } \left\| \frac{(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_2^2} \right\|_2 \leq 1,$$

which we saw in Ex. 1.5.81. This estimate holds for *all* matrices  $\mathbf{J}$  satisfying (2.4.60).

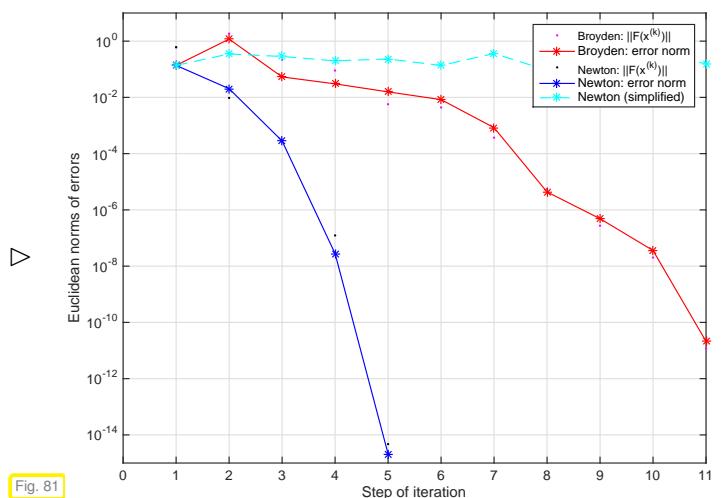
We may read this as follows: (2.4.57) gives the  $\|\cdot\|_2$ -minimal relative correction of  $\mathbf{J}_{k-1}$ , such that the secant condition (2.4.54) holds.

**Experiment 2.4.62 (Broydens quasi-Newton method: Convergence)**

We revisit the  $2 \times 2$  non-linear system of the Exp. 2.4.34 and take  $\mathbf{x}^{(0)} = [0.7, 0.7]^T$ . As starting value for the matrix iteration we use  $\mathbf{J}_0 = \mathbf{D}F(\mathbf{x}^{(0)})$ .

The numerical example shows that, in terms of convergence, the method is:

- slower than Newton method (2.4.1),
- faster than the simplified Newton method (see Rem. 2.4.31)



### Remark 2.4.63 (Convergence monitors)

In general, any iterative methods for non-linear systems of equations convergence can fail, that is it may stall or even diverge.

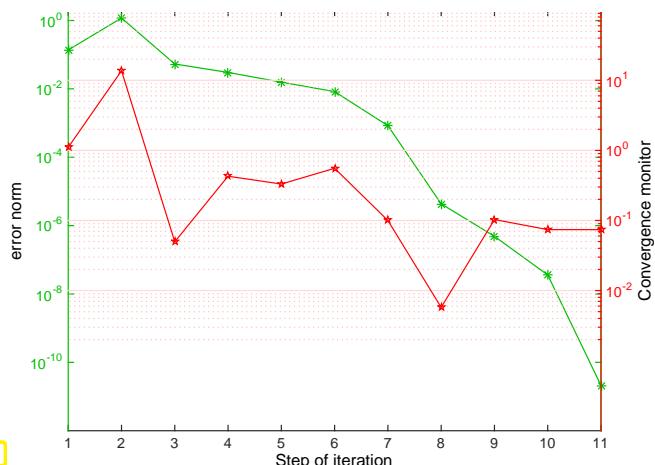
Demand on good numerical software: Algorithms should warn users of impending failure. For iterative methods this is the task of **convergence monitors**, that is, conditions, *cheaply verifiable* a posteriori during the iteration, that indicate stalled convergence or divergence.

For the damped Newton's method this role can be played by the natural monotonicity test, see Code 2.4.50; if it fails repeatedly, then the iteration should terminate with an error status.

For Broyden's quasi-Newton method, a similar strategy can rely on the relative size of the "simplified Broyden correction"  $\mathbf{J}_k F(\mathbf{x}^{(k+1)})$ :

$$\text{Convergence monitor for (2.4.58)} : \quad \mu := \frac{\|\mathbf{J}_{k-1}^{-1} F(\mathbf{x}^{(k)})\|}{\|\Delta \mathbf{x}^{(k-1)}\|} < 1 ? \quad (2.4.64)$$

### Experiment 2.4.65 (Monitoring convergence for Broyden's quasi-Newton method)



We rely on the setting of Exp. 2.4.62.

We track

1. the Euclidean norm of the iteration error,
2. and the value of the convergence monitor from (2.4.64).

▷ Decay of (norm of) iteration error and  $\mu$  are well correlated.

### Remark 2.4.66 (Damped Broyden method)

Option to improve robustness (increase region of local convergence):

damped Broyden method (cf. same idea for Newton's method, Section 2.4.4)

### (2.4.67) Implementation of Broyden's quasi-Newton method

As remarked, (2.4.58) represents a rank-1-update as already discussed in § 1.6.104.

Idea: use Sherman-Morrison-Woodbury update-formula from Lemma 1.6.113, which yields

$$\mathbf{J}_{k+1}^{-1} = \left( \mathbf{I} - \frac{\mathbf{J}_k^{-1} F(\mathbf{x}^{(k+1)}) (\Delta \mathbf{x}^{(k)})^T}{\|\Delta \mathbf{x}^{(k)}\|_2^2 + \Delta \mathbf{x}^{(k)} \cdot \mathbf{J}_k^{-1} F(\mathbf{x}^{(k+1)})} \right) \mathbf{J}_k^{-1} = \left( \mathbf{I} + \frac{\Delta \mathbf{x}^{(k+1)} (\Delta \mathbf{x}^{(k)})^T}{\|\Delta \mathbf{x}^{(k)}\|_2^2} \right) \mathbf{J}_k^{-1}. \quad (2.4.68)$$

This gives a well defined  $\mathbf{J}_{k+1}$ , if

$$\|\mathbf{J}_k^{-1} F(\mathbf{x}^{(k+1)})\|_2 < \|\Delta \mathbf{x}^{(k)}\|_2. \quad (2.4.69)$$

"simplified Quasi-Newton correction"

Note that the condition (2.4.69) is checked by the convergence monitor (2.4.64).

Iterated application of (2.4.68) pays off, if iteration terminates after only a few steps. For large  $n \gg 1$  it is not advisable to form the matrices  $\mathbf{J}_k^{-1}$  (which will usually be dense in contrast to  $\mathbf{J}_k$ ), but we employ fast successive multiplications with rank-1-matrices ( $\rightarrow$  Ex. 1.4.10) to apply  $\mathbf{J}_k^{-1}$  to a vector. This is implemented in the following code.

MATLAB-code : Broyden method (2.4.58)

```

1 function x = broyden(F,x,J,tol)
2 k = 1;
3 [L,U] = lu(J);
4 s = U\ (L\f(x));
5 sn = dot(s,s);
6 dx = [s];
7 x = x - s;
8 while (sqrt(sn) > tol)
9     w = U\ (L\f);
10    for l=2:k-1
11        w = w+dx(:,l)*(dx(:,l-1)'*w);
12        /dxn(l-1);
13    end
14    if (norm(w)>=sn)
15        warning('Dubious step %d!',k);
16    end
17    z = s'*w;
18    s = (1+z/(sn-z))*w;
19    sn=s'*s;
20    dx = [dx,s];
21    dxn = [dxn,sn];
22    x = x - s;
23    f = F(x);
24 end

```

- Computational cost :      $\bullet$   $O(N^2 \cdot n)$  operations with vectors, (Level I)  
 $N$  steps                $\bullet$  1 LU-decomposition of  $J$ ,  $N \times$  solutions of SLEs, see Section 1.6.2.2  
                             $\bullet$   $N$  evaluations of  $F$  !

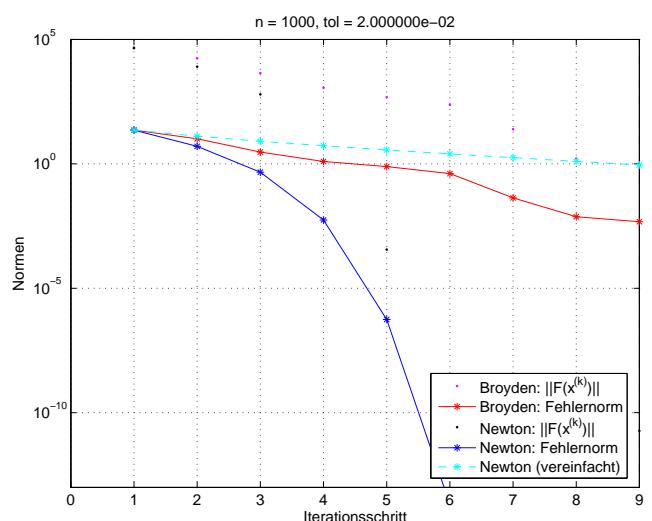
- Memory cost :      $\bullet$  LU-factors of  $J$  + auxiliary vectors  $\in \mathbb{R}^n$   
 $N$  steps                $\bullet$   $N$  vectors  $x^{(k)} \in \mathbb{R}^n$

### Experiment 2.4.70 (Broyden method for a large non-linear system)

$$\begin{aligned} F(x) &= \begin{cases} \mathbb{R}^n \mapsto \mathbb{R}^n \\ x \mapsto \text{diag}(x)Ax - b, \end{cases} \\ b &= [1, 2, \dots, n] \in \mathbb{R}^n, \\ A &= I + aa^T \in \mathbb{R}^{n,n}, \\ a &= \frac{1}{\sqrt{1 \cdot b - 1}}(b - 1). \end{aligned}$$

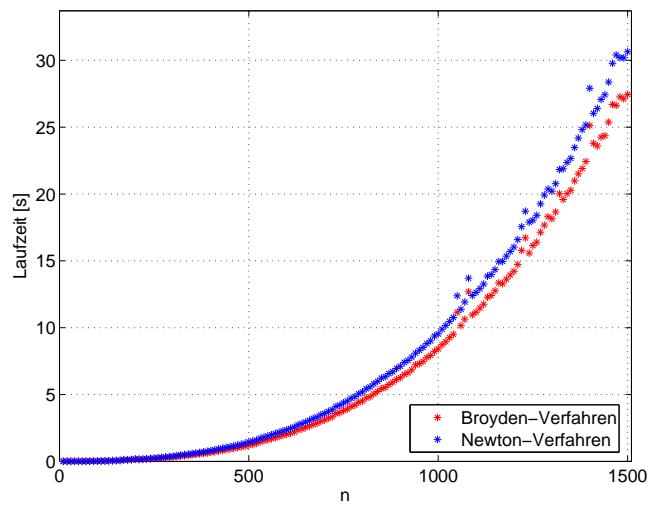
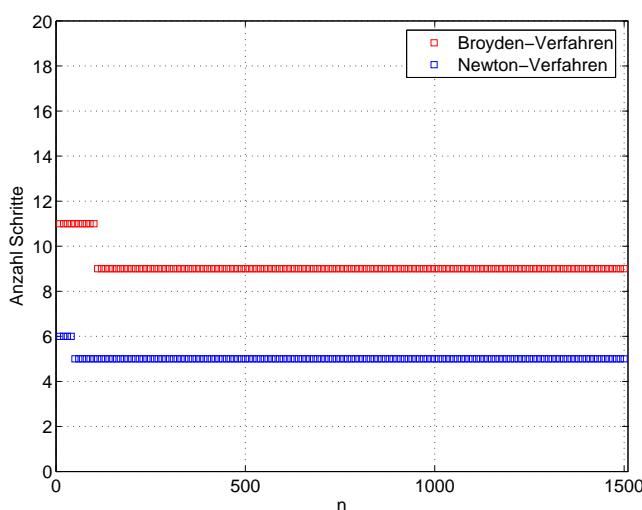
Initial guess:  $h = 2/n$ ;  $x_0 = (2:h:4-h)'$ ;

The results resemble those of Exp. 2.4.62



Efficiency comparison:

Broyden method  $\longleftrightarrow$  Newton method:  
(in case of dimension  $n$  use tolerance  $\text{tol} = 2n \cdot 10^{-5}$ ,  $h = 2/n$ ;  $x_0 = (2:h:4-h)'$  ; )



- ☞ In conclusion,  
the Broyden method is worthwhile for dimensions  $n \gg 1$  and low accuracy requirements.

## 2.5 Unconstrained Optimization

### 2.5.1 Minima and minimizers: Some theory

### 2.5.2 Newton's method

### 2.5.3 Descent methods

### 2.5.4 Quasi-Newton methods

## Learning Outcomes

- Knowledge about concepts related to the speed of convergence of an iteration for solving a non-linear system of equations.
- Ability to estimate type and orders of convergence from empiric data.
- Ability to predict asymptotic linear, quadratic and cubic convergence by inspection of the iteration function.
- Familiarity with (damped) Newton's method for general non-linear systems of equations and with the secant method in 1D.
- Ability to derive the Newton iteration for an (implicitly) given non-linear system of equations.
- Knowledge about quasi-Newton method as multi-dimensional generalizations of the secant method.

# Chapter 3

## Data Interpolation in 1D

### Contents

---

<b>3.1 Abstract interpolation . . . . .</b>	<b>244</b>
<b>3.2 Global Polynomial Interpolation . . . . .</b>	<b>249</b>
3.2.1 Polynomials . . . . .	250
3.2.2 Polynomial Interpolation: Theory . . . . .	251
3.2.3 Polynomial Interpolation: Algorithms . . . . .	255
3.2.3.1 Multiple evaluations . . . . .	255
3.2.3.2 Single evaluation . . . . .	257
3.2.3.3 Extrapolation to zero . . . . .	260
3.2.3.4 Newton basis and divided differences . . . . .	262
3.2.4 Polynomial Interpolation: Sensitivity . . . . .	267
<b>3.3 Shape preserving interpolation . . . . .</b>	<b>271</b>
3.3.1 Shape properties of functions and data . . . . .	271
3.3.2 Piecewise linear interpolation . . . . .	273
<b>3.4 Cubic Hermite Interpolation . . . . .</b>	<b>275</b>
3.4.1 Definition and algorithms . . . . .	275
3.4.2 Local monotonicity preserving Hermite interpolation . . . . .	278
<b>3.5 Splines . . . . .</b>	<b>281</b>
3.5.1 Cubic spline interpolation . . . . .	282
3.5.2 Structural properties of cubic spline interpolants . . . . .	285
3.5.3 Shape Preserving Spline Interpolation . . . . .	289

---

### 3.1 Abstract interpolation

The task of (multidimensional) **data interpolation** (point interpolation) can be described as follows:

Given: data points  $(\mathbf{x}_i, \mathbf{y}_i)$ ,  $i = 0, \dots, n$ ,  $n \in \mathbb{N}$ ,  $\mathbf{x}_i \in D \subset \mathbb{R}^m$ ,  $\mathbf{y}_i \in \mathbb{R}^d$

Objective: reconstruction of a (continuous) function  $\mathbf{f} : D \mapsto \mathbb{R}^d$  satisfying the  $n+1$  interpolation conditions

$$\mathbf{f}(\mathbf{x}_i) = \mathbf{y}_i, \quad i = 0, \dots, n.$$

The function we find is called the **interpolant**  $\mathbf{f}$  of the given data set  $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=0}^n$ .

Minimal requirement on data:  $\mathbf{x}_i$  pairwise distinct:  $\mathbf{x}_i \neq \mathbf{x}_j$ , if  $i \neq j$ ,  $i, j \in \{0, \dots, n\}$ .

Focus in this chapter:  $m = 1$ ,  $d = 1$

$\Leftrightarrow$  interpolation of *scalar* data depending on *one* parameter ( $t \in \mathbb{R}$ ),

$\Leftrightarrow$  Data points  $(t_i, y_i)$ ,  $i = 0, \dots, n$ ,  $t_i \in \mathbb{R}$  called **nodes**,  $y_i \in \mathbb{R}$  called **values**,

$\Leftrightarrow$  interpolant  $f : I \subset \mathbb{R} \rightarrow \mathbb{R}$  is a function of one real variable.

interpolation conditions:  $f(t_i) = y_i, \quad i = 0, \dots, n$ . (3.1.1)

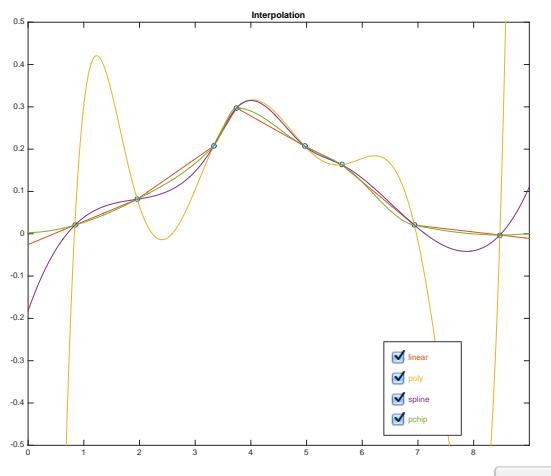
For ease of presentation we will usually assume that the nodes are ordered:  $t_0 < t_1 < \dots < t_n$  and  $[t_0, t_n] \subset I$ . However, algorithms often must not take for granted sorted nodes.

#### (3.1.2) Interpolation schemes

When we talk about “**interpolation schemes**” in 1D, we mean a mapping

$$I : \left\{ \begin{array}{ccc} \mathbb{R}^{n+1} \times \mathbb{R}^{n+1} & \rightarrow & \{f : I \rightarrow \mathbb{R}\} \\ ([t_i]_{i=0}^n, [y_i]_{i=0}^n) & \mapsto & \text{interpolant} \end{array} \right..$$

Once the function space to which the interpolant belongs is specified, then an interpolation scheme defines an “**interpolation problem**” in the sense of § 1.5.62. Sometimes, only the data values  $y_i$  are considered input data, whereas the dependence of the interpolant on the nodes  $t_i$  is suppressed, see Section 3.2.4.



- ▷ There are infinitely many ways to fix an interpolant for given data points.  
Interpolants can have vastly different properties.

Fig. 83

► We may (have to!) impose additional requirements on the interpolant:

- \* minimal **smoothness** of  $f$ , e.g.  $f \in C^1$ , etc.
- special **shape** of  $f$  (positivity, monotonicity, convexity → Section 3.3 below)

### Example 3.1.3 (Constitutive relations from measurements)

This example addresses an important application of data interpolation in 1D.

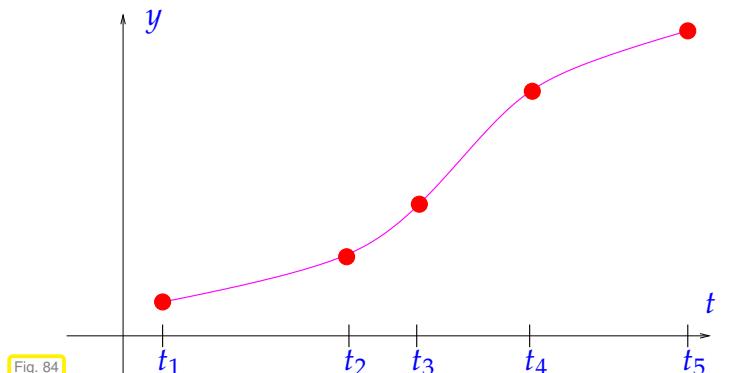
In this context:  $t, y \doteq$  two state variables of a physical system, where  $t$  determines  $y$ : a functional dependence  $y = y(t)$  is assumed.

Examples:  $t$  and  $y$  could be

$t$	$y$
voltage $U$	current $I$
pressure $p$	density $\rho$
magnetic field $H$	magnetic flux $B$
...	...

Known: several *accurate* (\*) measurements

$$(t_i, y_i), \quad i = 1, \dots, m$$



Imagine that  $t, y$  correspond to the voltage  $U$  and current  $I$  measured for a 2-port non-linear circuit element (like a diode). This element will be part of a circuit, which we want to simulate based on nodal analysis as in Ex. 2.0.1. In order to solve the resulting non-linear system of equations  $F(\mathbf{u}) = 0$  for the nodal potentials (collected in the vector  $\mathbf{u}$ ) by means of Newton's method (→ Section 2.4) we need the voltage-current relationship for the circuit element as a continuously differentiable function  $I = f(U)$ .

(\*) Meaning of attribute “accurate”: justification for interpolation. If measured values  $y_i$  were affected by considerable errors, one would not impose the interpolation conditions (3.1.1), but opt for **data fitting** (→ Ex. 6.0.1).

We can distinguish two aspects of the interpolation problem:

- \*① Find interpolant  $f : I \subset \mathbb{R} \rightarrow \mathbb{R}$  and store/represent it (internally).
- ② Evaluate  $f$  at a few or many evaluation points  $x \in I$

**Remark 3.1.4 (Mathematical functions in a numerical code)**

What does is meant to “represent” or “make available” a function  $f : I \subset \mathbb{R} \mapsto \mathbb{R}$  in a computer code?

- ! A general “mathematical” function  $f : I \subset \mathbb{R} \mapsto \mathbb{R}^d$ ,  $I$  an interval, contains an “infinite amount of information”.

Rather, in the context of numerical methods, “function” should be read as “subroutine”, a piece of code that can, for any  $x \in I$ , compute  $f(x)$  in finite time. Even this has to be qualified, because we can only pass machine numbers  $x \in I \cap \mathbb{M}$  ( $\rightarrow$  § 1.5.13) and, of course, in most cases,  $f(x)$  will be an approximation. In a C++ code a simple real valued function can be incarnated through an function object of a type as given in Code 3.1.5.

**C++-code 3.1.5: C++ function**

```

1 class Function {
2   private:
3     // various internal data describing f
4   public:
5     // Constructor: accepts information necessary for specifying the
      function
6     Function(/* .... */);
7     // Evaluation operator
8     double operator() (double t) const;
9 }
```

**(3.1.6) Internal representation of a mathematical function**

→ Idea: **parametrization**, a finite number of parameters  $c_0, \dots, c_m$ ,  $m \in \mathbb{N}$ , characterizes  $f$ .

Special case: Representation with *finite linear combination* of **basis functions**

$$b_j : I \subset \mathbb{R} \mapsto \mathbb{R}, \quad j = 0, \dots, m:$$

$$f = \sum_{j=0}^m c_j b_j, \quad c_j \in \mathbb{R}^d. \quad (3.1.7)$$

→  $f \in$  finite dimensional **function space**  $V_m := \text{Span}\{b_0, \dots, b_m\}$ ,  $\dim V_m = m + 1$ .

Of course, the basis functions  $b_j$  should be “simple” in the sense that  $b_j(x)$  can be computed efficiently for every  $x \in I$  and every  $j = 0, \dots, m$ .

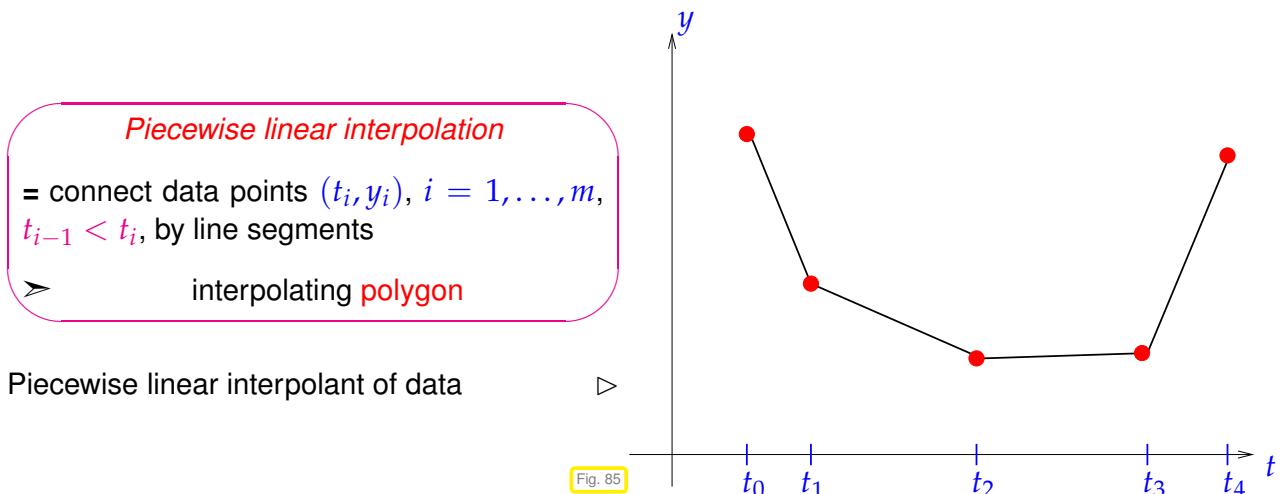
Note that the basis functions *may depend on the nodes  $t_i$* , but they *must not depend on the values  $y_i$* .

→ The internal representation of  $f$  (in the data member section of the class **Function** from Code 3.1.5) will then boil down to storing the **coefficients/parameters**  $c_j$ ,  $j = 0, \dots, m$ .

Note: The focus in this chapter will be on the special case that the data interpolants belong to a finite-dimensional space of functions spanned by “simple” basis functions.

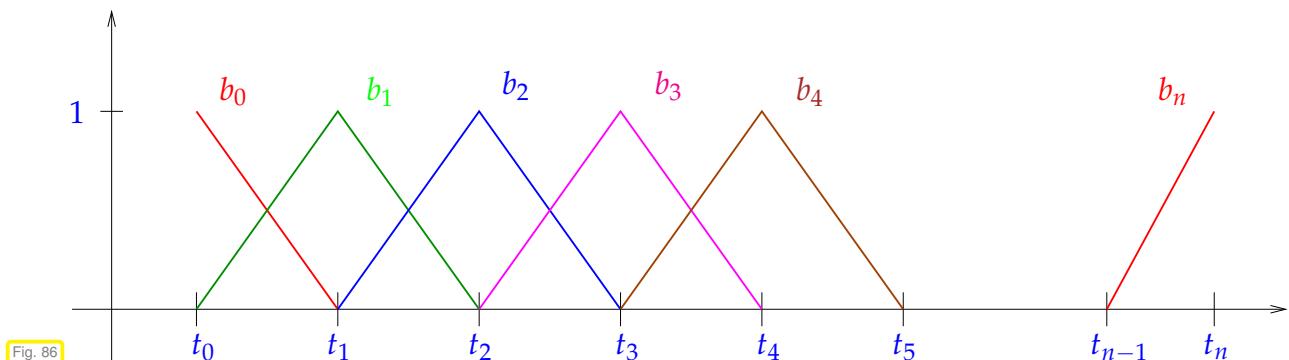
### Example 3.1.8 (Piecewise linear interpolation, see also Section 3.3.2)

Recall: A linear function in 1D is a function of the form  $x \mapsto a + bx$ ,  $a, b \in \mathbb{R}$  (polynomial of degree 1).



What could be a convenient set of basis functions  $\{b_j\}_{j=0}^n$  for representing the piecewise linear interpolant through  $n$  data points?

“Tent function” (“hat function”) basis:



Note: in Fig. 86 the basis functions have to be extended by zero outside the  $t$ -range where they are drawn.

Explicit formulas for these basis functions can be given and bear out that they are really “simple”:

$$\begin{aligned}
 b_0(t) &= \begin{cases} 1 - \frac{t-t_0}{t_1-t_0} & \text{for } t_0 \leq t < t_1, \\ 0 & \text{for } t \geq t_1. \end{cases} \\
 b_j(t) &= \begin{cases} 1 - \frac{t_j-t}{t_{j+1}-t_j} & \text{for } t_{j-1} \leq t < t_j, \\ 1 - \frac{t-t_j}{t_{j+1}-t_j} & \text{for } t_j \leq t < t_{j+1}, \quad , \quad j = 1, \dots, n-1, \\ 0 & \text{elsewhere in } [t_0, t_n]. \end{cases} \\
 b_n(t) &= \begin{cases} 1 - \frac{t_n-t}{t_n-t_{n-1}} & \text{for } t_{n-1} \leq t < t_n, \\ 0 & \text{for } t < t_{n-1}. \end{cases}
 \end{aligned} \tag{3.1.9}$$

Moreover, these basis functions are *uniquely* determined by the conditions

- $b_j$  is continuous on  $[t_0, t_n]$ ,
- $b_j$  is linear on each interval  $[t_{i-1}, t_i]$ ,  $i = 1, \dots, n$ ,
- $b_j(t_i) = \delta_{ij} := \begin{cases} 1 & \text{if } i = j \\ 0 & \text{else.} \end{cases}$   $\Rightarrow$  a so-called **cardinal basis** for the node set  $\{t_i\}_{i=0}^n$ .

This last condition implies a simple basis representation of a (the ?) piecewise linear interpolant of the data points  $(t_i, y_i)$ ,  $i = 0, \dots, n$ :

$$f(t) = \sum_{j=0}^n y_j b_j(t), \quad t_0 \leq t \leq t_n, \quad (3.1.10)$$

where the  $b_j$  are given by (3.1.9).

### (3.1.11) Interpolation as a linear mapping

We consider the setting for interpolation that the interpolant belongs to a finite-dimension space  $V_m$  of functions spanned by basis functions  $b_0, \dots, b_m$ , see Rem. 3.1.4. Then the interpolation conditions imply that the basis expansion coefficients satisfy a linear system of equations:

$$(3.1.1) \& (3.1.7) \Rightarrow f(t_i) = \sum_{j=0}^m c_j b_j(t_i) = y_i, \quad i = 0, \dots, n, \quad (3.1.12)$$

$\Updownarrow$

$$\mathbf{Ac} := \begin{bmatrix} b_0(t_0) & \dots & b_m(t_0) \\ \vdots & & \vdots \\ b_0(t_n) & \dots & b_m(t_n) \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix} =: \mathbf{y}. \quad (3.1.13)$$

This is an  $(m+1) \times (n+1)$  linear system of equations !

The interpolation problem in  $V_m$  and the linear system (3.1.13) are really equivalent in the sense that (unique) solvability of one implies (unique) solvability of the other.



**Necessary condition** for unique solvability :  $m = n$

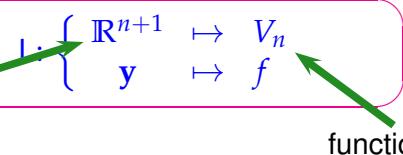
If  $m = n$  and  $\mathbf{A}$  from (3.1.13) regular ( $\rightarrow$  Def. 1.6.8), then for any values  $y_j$ ,  $j = 0, \dots, n$  we can find coefficients  $c_j$ ,  $j = 0, \dots, n$ , and, from them build the interpolant according to (3.1.7):

$$f = \sum_{j=0}^n (\mathbf{A}^{-1} \mathbf{y})_j b_j. \quad (3.1.14)$$



For fixed nodes  $t_i$  the interpolation problem (3.1.12) defines linear mapping

data space



function space

Beware, “linear” in the statement above has nothing to do with a linear function or piecewise linear interpolation discussed in Ex. 3.1.8!

### Definition 3.1.15. Linear interpolation operator

An interpolation operator  $I : \mathbb{R}^{n+1} \mapsto C^0([t_0, t_m])$  for the given nodes  $t_0 < t_1 < \dots < t_n$  is called linear, if

$$I(\alpha\mathbf{y} + \beta\mathbf{z}) = \alpha I(\mathbf{y}) + \beta I(\mathbf{z}) \quad \forall \mathbf{y}, \mathbf{z} \in \mathbb{R}^{n+1}, \alpha, \beta \in \mathbb{R}. \quad (3.1.16)$$

☞ Notation:  $C^0([t_0, t_m]) \hat{=} \text{vector space of continuous functions on } [t_0, t_m]$

### Remark 3.1.17 (A data type designed for of interpolation problem)

If a constitutive relationship for a circuit element is needed in a C++ simulation code ( $\rightarrow$  Ex. 3.1.3), the following data type could be used to represent it:

#### C++-code 3.1.18: Interpolant class

```

1  class Interpolant {
2      private:
3          // Various internal data describing f
4          // Can be the coefficients of a basis representation (3.1.7)
5      public:
6          // Constructor: computation of coefficients c_j of representation
7          // (3.1.7)
8          Interpolant(const vector<double> &t, const vector<double> &y);
9          // Evaluation operator for interpolant f
10         double operator() (double t) const;
11     };

```

Practical object oriented implementation of interpolation operator:

- ✿ Constructor: “setup phase”, e.g. building and solving linear system of equations (3.1.13)
- ✿ Evaluation operator, e.g., implemented as evaluation of linear combination (3.1.7)

Crucial issue:

computational effort for evaluation of interpolant at single point:  $O(1)$  or  $O(n)$  (or in between)?

## 3.2 Global Polynomial Interpolation

(Global) polynomial interpolation, that is, interpolation into spaces of functions spanned by polynomials up to a certain degree, is the simplest interpolation scheme and of great importance as building block for more complex algorithms.

### 3.2.1 Polynomials

Notation: Vector space of the polynomials of degree  $\leq k, k \in \mathbb{N}$ :

$$\mathcal{P}_k := \{t \mapsto \alpha_k t^k + \alpha_{k-1} t^{k-1} + \cdots + \alpha_1 t + \alpha_0, \alpha_j \in \mathbb{R}\}. \quad (3.2.1)$$

leading coefficient

Terminology: the functions  $t \mapsto t^k, k \in \mathbb{N}_0$ , are called **monomials**

$t \mapsto \alpha_k t^k + \alpha_{k-1} t^{k-1} + \cdots + \alpha_0$  = **monomial representation** of a polynomial.

Monomial representation is a linear combination of basis functions  $t \mapsto t^k$ , see Rem. 3.1.4.

Obvious:  $\mathcal{P}_k$  is a vector space, see [59, Sect. 4.2, Bsp. 4]. What is its dimension?

#### Theorem 3.2.2. Dimension of space of polynomials

$$\dim \mathcal{P}_k = k + 1 \quad \text{and} \quad \mathcal{P}_k \subset C^\infty(\mathbb{R}).$$

*Proof.* Dimension formula by linear independence of monomials.  $\square$

As a consequence of Thm. 3.2.2 the monomial representation of a polynomial is unique.

#### (3.2.3) The charms of polynomials

Why are polynomials important in computational mathematics?

- Easy to compute (only elementary operations required), integrate and differentiate
- Vector space & algebra
- Analysis: Taylor polynomials & power series

#### Remark 3.2.4 (Polynomials in MATLAB)

MATLAB represents polynomials through the vector of their monomial coefficients:

MATLAB:  $\alpha_k t^k + \alpha_{k-1} t^{k-1} + \cdots + \alpha_0 \rightarrow$  Vector  $(\alpha_k, \alpha_{k-1}, \dots, \alpha_0)$  (ordered!).

#### Remark 3.2.5 (Horner scheme → [15, Bem. 8.11])

Efficient evaluation of a polynomial in monomial representation through **Horner scheme** according to

$$p(t) = t(\cdots t(t(\alpha_n t + \alpha_{n-1}) + \alpha_{n-2}) + \cdots + \alpha_1) + \alpha_0. \quad (3.2.6)$$

**MATLAB-code 3.2.7: Horner scheme (polynomial in MATLAB format, see Rem. 3.2.4)**

```

1 function y = polyval(p,x)
2 y = p(1); for i=2:length(p), y = x.*y+p(i); end

```

Optimal asymptotic complexity:  $O(n)$

Realized in MATLAB “built-in”-function `polyval(p,x)`. (The argument  $x$  can be a matrix or a vector. In this case the function evaluates the polynomial described by  $p$  for each entry/component.)

### 3.2.2 Polynomial Interpolation: Theory



*Supplementary reading.* This topic is also presented in [15, Sect. 8.2.1], [63, Sect. 8.1], [6, Ch. 10].

Now we consider the interpolation problem introduced in Section 3.1 for the special case that the sought interpolant belongs to the polynomial space  $\mathcal{P}_k$  (with suitable degree  $k$ ).

#### Lagrange polynomial interpolation problem

Given the simple nodes  $t_0, \dots, t_n$ ,  $n \in \mathbb{N}$ ,  $-\infty < t_0 < t_1 < \dots < t_n < \infty$  and the values  $y_0, \dots, y_n \in \mathbb{R}$  compute  $p \in \mathcal{P}_n$  such that

$$p(t_j) = y_j \quad \text{for } j = 0, \dots, n. \quad (3.2.9)$$

Is this a well-defined problem? Obviously, it fits the framework developed in Rem. 3.1.4 and § 3.1.11, because  $\mathcal{P}_n$  is a finite-dimensional space of functions, for which we already know a basis, the monomials. Thus, in principle, we could examine the matrix  $\mathbf{A}$  from (3.1.13) to decide, whether the polynomial interpolant exists and is unique. However, there is a shorter way.

#### (3.2.10) Lagrange polynomials

For nodes  $t_0 < t_1 < \dots < t_n$  ( $\rightarrow$  Lagrange interpolation) consider the

$$\text{Lagrange polynomials} \quad L_i(t) := \prod_{\substack{j=0 \\ j \neq i}}^n \frac{t - t_j}{t_i - t_j}, \quad i = 0, \dots, n. \quad (3.2.11)$$

→ Evidently, the Lagrange polynomials satisfy  $L_i \in \mathcal{P}_n$  and

$$L_i(t_j) = \delta_{ij}$$

Recall the Kronecker symbol  $\delta_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{else.} \end{cases}$

From this relationship we infer that the Lagrange polynomials are linearly independent. Since there are  $n+1 = \dim \mathcal{P}_n$  different Lagrange polynomials, we conclude that they form a **basis** of  $\mathcal{P}_n$ , which is a **cardinal basis** for the node set  $\{t_i\}_{i=0}^n$ .

### Example 3.2.12 ( Lagrange polynomials for uniformly spaced nodes)

Consider the equidistant nodes in  $[-1, 1]$ :

$$\mathcal{T} := \{t_j = -1 + \frac{2}{n} j\}, \quad j = 0, \dots, n.$$

The plot shows the Lagrange polynomials for this set of nodes that do not vanish in the nodes  $t_0$ ,  $t_2$ , and  $t_5$ , respectively.

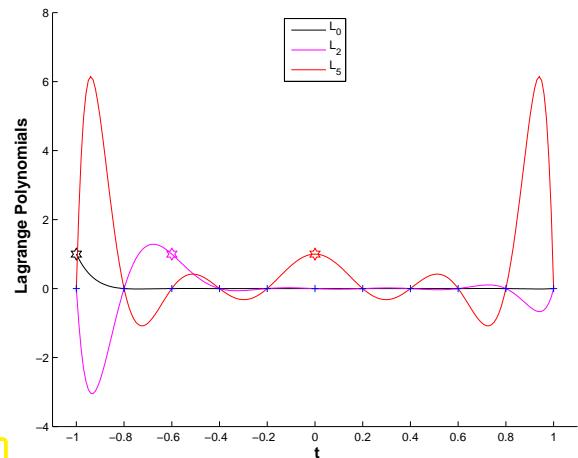


Fig. 87

The Lagrange polynomial interpolant  $p$  for data points  $(t_i, y_i)_{i=0}^n$  allows a straightforward representation with respect to the basis of Lagrange polynomials for the node set  $\{t_i\}_{i=0}^n$ :

$$p(t) = \sum_{i=0}^n y_i L_i(t) \quad \Leftrightarrow \quad p \in \mathcal{P}_n \quad \text{and} \quad p(t_i) = y_i. \quad (3.2.13)$$

**Theorem 3.2.14. Existence & uniqueness of Lagrange interpolation polynomial** → [63, Thm. 8.1], [15, Satz 8.3]

*The general Lagrange polynomial interpolation problem admits a unique solution  $p \in \mathcal{P}_n$ .*

*Proof.* Consider the *linear* evaluation operator

$$\text{eval}_{\mathcal{T}} : \begin{cases} \mathcal{P}_n &\mapsto \mathbb{R}^{n+1}, \\ p &\mapsto (p(t_i))_{i=0}^n, \end{cases}$$

which maps between finite-dimensional vector spaces of the same dimension, see Thm. 3.2.2.

Representation (3.2.13)     $\Rightarrow$     existence of interpolating polynomial  
 $\Rightarrow$   $\text{eval}_{\mathcal{T}}$  is **surjective** ("onto")

Known from linear algebra: for a linear mapping  $T : V \mapsto W$  between finite-dimensional vector spaces with  $\dim V = \dim W$  holds the equivalence

$$T \text{ surjective} \iff T \text{ bijective} \iff T \text{ injective.}$$

Applying this equivalence to  $\text{eval}_{\mathcal{T}}$  yields the assertion of the theorem

□

**Corollary 3.2.15. Lagrange interpolation as linear mapping → § 3.1.11**

The polynomial interpolation in the nodes  $\mathcal{T} := \{t_j\}_{j=0}^n$  defines a linear operator

$$\mathbf{I}_{\mathcal{T}} : \begin{cases} \mathbb{R}^{n+1} & \rightarrow \mathcal{P}_n, \\ (y_0, \dots, y_n)^T & \mapsto \text{interpolating polynomial } p . \end{cases} \quad (3.2.16)$$

**Remark 3.2.17 (Vandermonde matrix)**

Lagrangian polynomial interpolation leads to linear systems of equations also for the representation coefficients of the polynomial interpolant in **monomial basis**, see § 3.1.11:

$$\begin{aligned} p(t_j) = y_j &\iff \sum_{i=0}^n a_i t_j^i = y_j, \quad j = 0, \dots, n \\ &\iff \text{solution of } (n+1) \times (n+1) \text{ linear system } \mathbf{V}\mathbf{a} = \mathbf{y} \text{ with matrix} \end{aligned}$$

$$\mathbf{V} = \begin{bmatrix} 1 & t_0 & t_0^2 & \cdots & t_0^n \\ 1 & t_1 & t_1^2 & \cdots & t_1^n \\ 1 & t_2 & t_2^2 & \cdots & t_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_n & t_n^2 & \cdots & t_n^n \end{bmatrix} \in \mathbb{R}^{n+1, n+1}. \quad (3.2.18)$$

A matrix in the form of  $\mathbf{V}$  is called **Vandermonde matrix**.

In MATLAB, given a column vector  $\mathbf{t}$  of node positions, the corresponding Vandermonde matrix can be generated by

- `for j = 1 : length(t); V(j,:) = t(j).^(0 : length(t) - 1); end;`
- `for j = 1 : length(t); V(:,j) = t.^ (j - 1); end;`
- `fliplr(vander(t))`

**Remark 3.2.19 (Matrix representation of interpolation operator)**

In the case of Lagrange interpolation:

- if Lagrange polynomials are chosen as basis for  $\mathcal{P}_n$ , then  $\mathbf{I}_{\mathcal{T}}$  is represented by the identity matrix;
- if monomials are chosen as basis for  $\mathcal{P}_n$ , then  $\mathbf{I}_{\mathcal{T}}$  is represented by the inverse of the Vandermonde matrix  $\mathbf{V}$ , see Eq. (3.2.18).

**Remark 3.2.20 (Generalized polynomial interpolation → [15, Sect. 8.2.7], [63, Sect. 8.4])**

The following generalization of Lagrange interpolation is possible: We still seek a polynomial interpolant, but beside function values also prescribe **derivatives** up to a certain order for interpolating polynomial at given nodes.

Convention: indicate occurrence of derivatives as interpolation conditions by *multiple nodes*.

### Generalized polynomial interpolation problem

Given the (possibly multiple) nodes  $t_0, \dots, t_n$ ,  $n \in \mathbb{N}$ ,  $-\infty < t_0 \leq t_1 \leq \dots \leq t_n < \infty$  and the values  $y_0, \dots, y_n \in \mathbb{R}$  compute  $p \in \mathcal{P}_n$  such that

$$\frac{d^k}{dt^k} p(t_j) = y_j \quad \text{for } k = 0, \dots, l_j \quad \text{and } j = 0, \dots, n, \quad (3.2.21)$$

where  $l_j := \max\{i - i': t_j = t_i = t_{i'}, i, i' = 0, \dots, n\}$  is the *multiplicity* of the nodes  $t_j$ .

The most important case of generalized Lagrange interpolation is when all the multiplicities are equal to 2. It is called **Hermite interpolation** (or osculatory interpolation) and the generalized interpolation conditions read for nodes  $t_0 = t_1 < t_2 = t_3 < \dots < t_{n-1} = t_n$  (note the **double nodes!**) [63, Ex. 8.6]:

$$p(t_{2j}) = y_{2j}, \quad p'(t_{2j}) = y_{2j+1}, \quad j = 0, \dots, n/2.$$

### Theorem 3.2.22. Existence & uniqueness of generalized Lagrange interpolation polynomials

The generalized polynomial interpolation problem Eq. (3.2.21) admits a unique solution  $p \in \mathcal{P}_n$ .

### Definition 3.2.23. Generalized Lagrange polynomials

The **generalized Lagrange polynomials** for the nodes  $\mathcal{T} = \{t_j\}_{j=0}^n \subset \mathbb{R}$  (multiple nodes allowed) are defined as  $L_i := |_{\mathcal{T}}(\mathbf{e}_{i+1})$ ,  $i = 0, \dots, n$ , where  $\mathbf{e}_i = (0, \dots, 0, 1, 0, \dots, 0)^T \in \mathbb{R}^{n+1}$  are the unit vectors.

Note: The linear interpolation operator  $|_{\mathcal{T}}$  in this definition refers to generalized Lagrangian interpolation. Its existence is guaranteed by Thm. 3.2.22.

### Example 3.2.24 (Generalized Lagrange polynomials for Hermite Interpolation)

Consider the node set

$$\mathcal{T} = \{t_0 = 0, t_1 = 0, t_2 = 1, t_3 = 1\}.$$

The plot shows the four unique generalized Lagrange polynomials of degree  $n = 3$  for these nodes. They satisfy

$$\begin{aligned} p_0(0) &= 1, p_0'(0) = p_0'(1) = 0, \\ p_1(1) &= 1, p_1(0) = p_1'(0) = p_1'(1) = 0, \\ p_2'(0) &= 1, p_2(1) = p_2(0) = p_2'(1) = 0, \\ p_3'(1) &= 1, p_3(1) = p_3(0) = p_3'(0) = 0. \end{aligned}$$

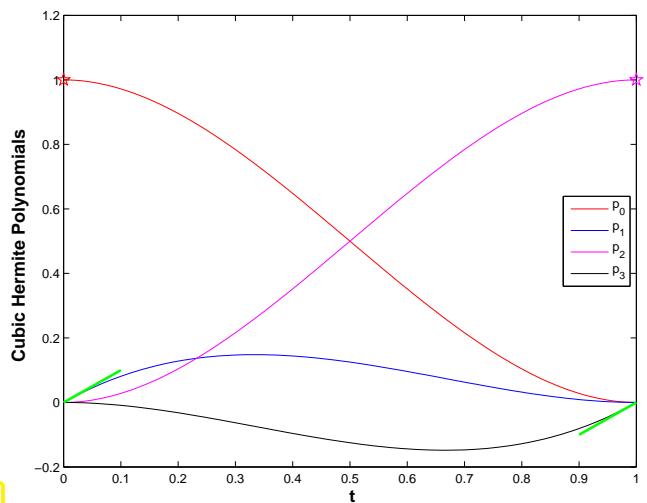


Fig. 88

More details are given in Section 3.4. For explicit formulas for the polynomials see (3.4.5).

### 3.2.3 Polynomial Interpolation: Algorithms

Now we consider the algorithmic realization of Lagrange interpolation as introduced in Section 3.2.2. The setting is as follows:

Given: nodes  $\mathcal{T} := \{-\infty < t_0 < t_1 < \dots < t_n < \infty\}$ ,  
values  $\mathbf{y} := \{y_0, y_1, \dots, y_n\}$ ,

We also write  $p := l_{\mathcal{T}}(\mathbf{y})$  for the unique Lagrange interpolation polynomial given by Thm. 3.2.14.

When used in a numerical code, different demands can be made for a routine that implements Lagrange interpolation. They determine, which algorithm is most suitable.

#### 3.2.3.1 Multiple evaluations

Task: For ① a *fixed* set  $\{t_0, \dots, t_n\}$  of nodes,  
and ② *many* different given data values  $y_i^k$ ,  
and ③ *many* arguments  $x_k, k = 1, \dots, N, N \gg 1$ ,  
*efficiently* compute all  $p(x_k)$  for  $p \in \mathcal{P}_n$  interpolating in  $(t_i, y_i^k), i = 0, \dots, n$ .

The definition of a possible interpolator data type could be as follows:

#### C++-code 3.2.25: Polynomial Interpolation

```

1  class PolyInterp {
2  private:
3      // various internal data describing p
4      Eigen::VectorXd t;
5  public:
6      // Constructors taking node vector (t_0, ..., t_n) as argument
7      PolyInterp(const Eigen::VectorXd &t);
8      template <typename SeqContainer>
9          PolyInterp(const SeqContainer &v);
10     // Evaluation operator for data (y_0, ..., y_n); computes
11     // p(x_k) for x_k's passed in x
12     Eigen::VectorXd eval(const Eigen::VectorXd &y,
13                          const Eigen::VectorXd &x) const;
14 }
```

The member function `eval(y, x)` expects  $n$  data values in  $y$  and (any number of) evaluation points in  $x$  ( $\leftrightarrow [x_1, \dots, x_N]$ ) and returns the vector  $[p(x_1), \dots, p(x_N)]^\top$ , where  $p$  is the Lagrange polynomial interpolant.

An implementation directly based on the evaluation of Lagrange polynomials (3.2.11) and (3.2.13) would incur an asymptotic computational effort of  $O(n^2N)$  for every single invocation of `eval` and large  $n, N$ .

### (3.2.26) Barycentric interpolation formula

By means of *pre-calculations* the asymptotic effort for `eval` can be reduced substantially:

Simple manipulations starting from (3.2.13) give an alternative representation of  $p$ :

$$p(t) = \sum_{i=0}^n L_i(t) y_i = \sum_{i=0}^n \prod_{\substack{j=0 \\ j \neq i}}^n \frac{t - t_j}{t_i - t_j} y_i = \sum_{i=0}^n \lambda_i \prod_{\substack{j=0 \\ j \neq i}}^n (t - t_j) y_i = \prod_{j=0}^n (t - t_j) \cdot \sum_{i=0}^n \frac{\lambda_i}{t - t_i} y_i.$$

where  $\lambda_i = \frac{1}{(t_i - t_0) \cdots (t_i - t_{i-1})(t_i - t_{i+1}) \cdots (t_i - t_n)}$ ,  $i = 0, \dots, n$ .

From above formula, with  $p(t) \equiv 1$ ,  $y_i = 1$ :

$$\begin{aligned} 1 &= \prod_{j=0}^n (t - t_j) \sum_{i=0}^n \frac{\lambda_i}{t - t_i} \quad \Rightarrow \quad \prod_{j=0}^n (t - t_j) = \frac{1}{\sum_{i=0}^n \frac{\lambda_i}{t - t_i}} \\ &\quad \blacktriangleright \text{Barycentric interpolation formula} \quad p(t) = \frac{\sum_{i=0}^n \frac{\lambda_i}{t - t_i} y_i}{\sum_{i=0}^n \frac{\lambda_i}{t - t_i}}. \end{aligned} \quad (3.2.27)$$

with  $\lambda_i = \frac{1}{(t_i - t_0) \cdots (t_i - t_{i-1})(t_i - t_{i+1}) \cdots (t_i - t_n)}$ ,  $i = 0, \dots, n$ , independent of  $t$  and  $y_i$   
 → precompute !

The use of (3.2.27) involves

- \* computation of weights  $\lambda_i$ ,  $i = 0, \dots, n$ : cost  $O(n^2)$  (only once!),
  - \* cost  $O(n)$  for every subsequent evaluation of  $p$ .
- ⇒ total asymptotic complexity  $O(Nn) + O(n^2)$

The following MATLAB code demonstrates an efficient implementation.

#### MATLAB-code 3.2.28: Evaluation of the interpolation polynomials with barycentric formula

```

1 function p = intpolyval(t, y, x)
2 % t: row vector of nodes t_0, ..., t_n
3 % y: row vector of data y_0, ..., y_n
4 % x: row vector of evaluation points x_1, ..., x_N
5 n = length(t); % number of interpolation nodes = degree of polynomial
6 N = length(x); % Number of evaluation points stored in x
7 % Precompute the weights lambda_i with effort O(n^2)
8 for k = 1:n
9     lambda(k) = 1 / prod(t(k) - t([1:k-1, k+1:N])); end;
10 for i = 1:N
11     % Compute quotient of weighted sums of  $\frac{\lambda_i}{t - t_i}$ , effort O(n)

```

```

12 z = (x(i)-t); j = find(z == 0);
13 if (~isempty(j)), p(i) = y(j); % avoid division by zero
14 else
15 mu = lambda./z; p(i) = sum(mu.*y)/sum(mu);
16 end
17 end

```

As an exception, the test for equality with zero in Line 12 is possible. If  $x(i)$  is almost equal to  $t$ , then the corresponding entry of the vector  $\mu$  will be huge and the value of the barycentric sum will almost agree with  $p(i)$ , the same value returned, if  $x(i)$  is exactly zero. Hence, it does not matter, if the test returns `false` because of some small perturbations of the values.

Runtime measurements (tic-toe-computational time), of MATLAB's `polyval` vs. barycentric formula are given in Exp. 3.2.34.

### 3.2.3.2 Single evaluation



*Supplementary reading.* This topic is also discussed in [15, Sect. 8.2.2].

Task: Given a set of interpolation points  $(t_j, y_j)$ ,  $j = 0, \dots, n$ , with pairwise different interpolation nodes  $t_j$ , perform *a single* point evaluation of the Lagrange polynomial interpolant  $p$  at  $x \in \mathbb{R}$ .

We discuss the efficient implementation of the following function for  $n \gg 1$ . It is meant for a single evaluation of a Lagrange interpolant.

```

double eval(const Eigen::VectorXd &t, const Eigen::VectorXd &y,
            double x);

```

### (3.2.29) Aitken-Neville scheme

The starting point is a recursion formula for partial Lagrange interpolants: For  $0 \leq k \leq \ell \leq n$  define

$p_{k,\ell} :=$  unique interpolating polynomial of degree  $\ell - k$  through  $(t_k, y_k), \dots, (t_\ell, y_\ell)$ ,

From the uniqueness of polynomial interpolants ( $\rightarrow$  Thm. 3.2.14) we find

$$\begin{aligned}
 p_{k,k}(x) &\equiv y_k \quad (\text{"constant polynomial"}) , \quad k = 0, \dots, n , \\
 p_{k,\ell}(x) &= \frac{(x - t_k)p_{k+1,\ell}(x) - (x - t_\ell)p_{k,\ell-1}(x)}{t_\ell - t_k} \\
 &= p_{k+1,\ell}(x) + \frac{x - t_\ell}{t_\ell - t_k} (p_{k+1,\ell}(x) - p_{k,\ell-1}(x)) , \quad 0 \leq k \leq \ell \leq n ,
 \end{aligned} \tag{3.2.30}$$

because the left and right hand sides represent polynomials of degree  $\ell - k$  through the points  $(t_j, y_j)$ ,  $j = k, \dots, \ell$ .

Thus the values of the partial Lagrange interpolants can be computed sequentially and their dependencies can be expressed by the following so-called **Aitken-Neville scheme**:

$n =$	0	1	2	3
$t_0$	$y_0 =: p_{0,0}(x)$	$\rightarrow p_{0,1}(x)$	$\rightarrow p_{0,2}(x)$	$\rightarrow p_{0,3}(x)$
$t_1$	$y_1 =: p_{1,1}(x)$	$\rightarrow p_{1,2}(x)$	$\rightarrow p_{1,3}(x)$	
$t_2$	$y_2 =: p_{2,2}(x)$	$\rightarrow p_{2,3}(x)$		
$t_3$	$y_3 =: p_{3,3}(x)$			

Here, the arrows indicate contributions to the convex linear combinations of (3.2.30). The computation can advance from left to right, which is done in following MATLAB code.

#### MATLAB-code 3.2.31: Aitken-Neville algorithm

```

1 function v = ANipoleval(t,y,x)
2 for i=1:length(y)
3     for k=i-1:-1:1
4         y(k) = y(k+1)+(y(k+1)-y(k))*(x-t(i))/(t(i)-t(k));
5     end
6 end
7 v = y(1);

```

The vector  $y$  contains the columns of the above triangular tableaux in turns from left to right.

Asymptotic complexity of `ANipoleval` in terms of number of data points:  $O(n^2)$  (two nested loops).

#### (3.2.32) Polynomial interpolation with data updates

The Aitken-Neville algorithm has another interesting feature, when we run through the Aitken-Neville scheme from the top left corner:

$n =$	0	1	2	3
$t_0$	$y_0 =: p_{0,0}(x)$	$\rightarrow p_{0,1}(x)$	$\rightarrow p_{0,2}(x)$	$\rightarrow p_{0,3}(x)$
$t_1$	$y_1 =: p_{1,1}(x)$	$\rightarrow p_{1,2}(x)$	$\rightarrow p_{1,3}(x)$	
$t_2$	$y_2 =: p_{2,2}(x)$	$\rightarrow p_{2,3}(x)$		
$t_3$	$y_3 =: p_{3,3}(x)$			

Thus, the values of partial polynomial interpolants at  $x$  can be computed before all data points are even processed. This results in an “update-friendly” algorithm that can efficiently supply the point values  $p_{0,k}(x)$ ,  $k = 0, \dots, n$ , while being supplied with the data points  $(t_i, y_i)$ . It can be used for the efficient implementation of the following interpolator class:

#### C++-code 3.2.33: Polynomial evaluations

```

1 class PolyEval {
2     private:
3         // evaluation point and various internal data describing the
4         // polynomials
4     public:
5         // Constructor taking the evaluation point as argument

```

```

6   PolyEval(double x);
7   // Add another data point and update internal information
8   void addPoint(t,y);
9   // Value of current interpolating polynomial at x
10  double eval(void) const;
11 }

```

### Experiment 3.2.34 (Timing polynomial evaluations)

Comparison of the computational time needed for polynomial interpolation of

$$\{t_i = i\}_{i=1,\dots,n}, \quad \{y_i = \sqrt{i}\}_{i=1,\dots,n}$$

$n = 3, \dots, 200$ , and evaluation in a single point  $x \in [0, n]$ .

Minimum `tic-toc`-computational time over 100 runs →

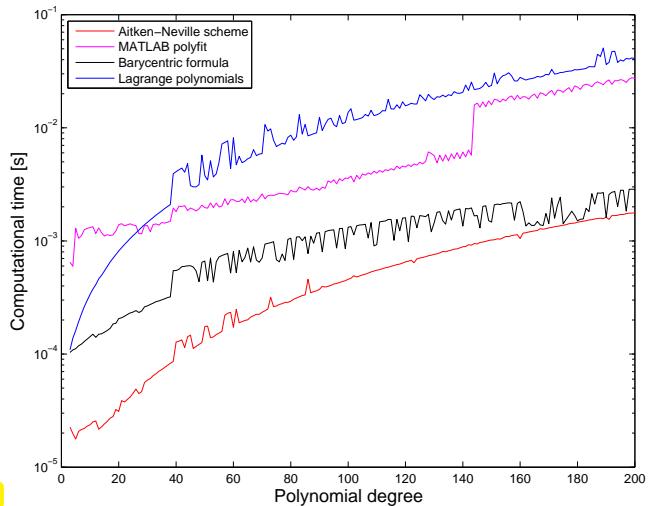


Fig. 89

### MATLAB-code 3.2.35: Timing polynomial evaluations

```

1 time=zeros(1,4);
2 f=@(x) sqrt(x); % function to interpolate:
3 for k=1:100
4   res = [];
5   for n=3:1:200 % n = increasing polynomial degree
6     t = (1:n); y = f(t); x=n*rand;
7     tic; v1 = ANipoleval(t,y,x); time(1) = toc;
8     tic; v2 = ipoleval(t,y,x); time(2) = toc;
9     tic; v3 = intpolyval(t,y,x); time(3) = toc;
10    tic; v4 = intpolyval_lag(t,y,x); time(4) = toc;
11    res = [res; n,time];
12  end
13  if (k == 1), finres = res;
14  else, finres = min(finres,res); end
15 end
16 figure
17 semilogy(finres(:,1),finres(:,2),'r-',finres(:,1),finres(:,3),'m-',
18 finres(:,1),finres(:,4),'k-', finres(:,1),finres(:,5),'b-');
19 xlabel('Polynomial degree','FontSize',14);
20 ylabel('Computational time [s]','FontSize',14);
21 legend('Aitken-Neville scheme','MATLAB polyfit', 'Barycentric
22 formula', 'Lagrange polynomials',2);

```

This uses functions given in Code 3.2.28, Code 3.2.31 and the MATLAB function `polyfit` (with a clearly

greater computational effort !)

#### MATLAB-code 3.2.36: MATLAB polynomial evaluation using built-in function polyfit

```

1 function v=ipoleval(t,y,x)
2 p = polyfit(t,y,length(y)-1);
3 v=polyval(p,x);
```

#### MATLAB-code 3.2.37: Lagrange polynomial interpolation and evaluation

```

1 function p = intpolyval_lag(t,y,x)
2 p=zeros(size(x));
3 for k=1:length(t); p=p + y(k)*lagrangepoly(x, k-1, t); end
4
5 function L=lagrangepoly(x, index, nodes)
6 L=1;
7 for j=[0:index-1, index+1:length(nodes)-1];
8     L = L .* (x-nodes(j+1)) ./ ( nodes(index+1)-nodes(j+1) );
9 end
```

### 3.2.3.3 Extrapolation to zero

**Extrapolation** is the same as interpolation but the evaluation point  $t$  is outside the interval  $[\inf_{j=0,\dots,n} t_j, \sup_{j=0,\dots,n} t_j]$ . In the sequel we assume  $t = 0, t_i > 0$ .

Of course, Lagrangian interpolation can also be used for extrapolation. In this section we give a very important application of this “Lagrangian extrapolation”.

Task: compute the limit  $\lim_{h \rightarrow 0} \psi(h)$  with prescribed accuracy, though the evaluation of the function  $\psi = \psi(h)$  (maybe given in procedural form only) for very small arguments  $|h| \ll 1$  is difficult, usually because of numerically instability ( $\rightarrow$  Section 1.5.5).

The extrapolation technique introduced below works well, if

- \*  $\psi$  is an even function of its argument:  $\psi(t) = \psi(-t)$ ,
- \*  $\psi = \psi(h)$  behaves “nicely” around  $h = 0$ .

Theory: existence of an **asymptotic expansion** in  $h^2$

$$f(h) = f(0) + A_1 h^2 + A_2 h^4 + \cdots + A_n h^{2n} + R(h) , \quad A_k \in \mathbb{R} ,$$

with remainder estimate  $|R(h)| = O(h^{2n+2})$  for  $h \rightarrow 0$ .

#### Idea: computing inaccessible limit by extrapolation to zero



- ① Pick  $h_0, \dots, h_n$  for which  $\psi$  can be evaluated “safely”.
- ② evaluation of  $\psi(h_i)$  for different  $h_i, i = 0, \dots, n, |t_i| > 0$ .
- ③  $\psi(0) \approx p(0)$  with interpolating polynomial  $p \in \mathcal{P}_n, p(h_i) = \psi(h_i)$ .

### (3.2.39) Numerical differentiation through extrapolation

In Ex. 1.5.43 we have already seen a situation, where we wanted to compute the limit of a function  $\psi(h)$  for  $h \rightarrow 0$ , but could not do it with sufficient accuracy. In this case  $\psi(h)$  was a one-sided difference quotient with span  $h$ , meant to approximate  $f'(x)$  for a differentiable function  $f$ . The cause of numerical difficulties was cancellation  $\rightarrow$  § 1.5.41.

Now we will see how to dodge cancellation in difference quotients and how to use extrapolation to zero to compute derivatives with high accuracy:

Given: smooth function  $f : I \subset \mathbb{R} \mapsto \mathbb{R}$  in procedural form: function  $y = f(x)$

Sought: (approximation of)  $f'(x)$ ,  $x \in I$ .

Natural idea: approximation of derivative by (symmetric) difference quotient

$$\frac{df}{dx}(x) \approx \frac{f(x+h) - f(x-h)}{2h}. \quad (3.2.40)$$

► straightforward implementation fails due to cancellation in the numerator, see also Ex. 1.5.43.

#### MATLAB-Script 3.2.41: Numeric differentiation through difference quotients

```

1 x=1.1; h=2.^[-1:-5:-36];
2 atanerr = abs(dirnumdiff(atan,x,h)-1/(1+x^2))*(1+x^2);
3 sqrtterr = abs(dirnumdiff(sqrt,x,h)-1/(2*sqrt(x)))*(2*sqrt(x));
4 experr = abs(dirnumdiff(exp,x,h)-exp(x))/exp(x);
5
6 function [df]=dirnumdiff(f,x,h)
7 df=(f(x+h)-f(x))./h;
8 end

```

$$f(x) = \arctan(x)$$

$$f(x) = \sqrt{x}$$

$$f(x) = \exp(x)$$

$h$	Relative error
$2^{-1}$	0.20786640808609
$2^{-6}$	0.00773341103991
$2^{-11}$	0.00024299312415
$2^{-16}$	0.00000759482296
$2^{-21}$	0.00000023712637
$2^{-26}$	0.00000001020730
$2^{-31}$	0.00000005960464
$2^{-36}$	0.00000679016113

$h$	Relative error
$2^{-1}$	0.09340033543136
$2^{-6}$	0.00352613693103
$2^{-11}$	0.00011094838842
$2^{-16}$	0.00000346787667
$2^{-21}$	0.00000010812198
$2^{-26}$	0.00000001923506
$2^{-31}$	0.00000001202188
$2^{-36}$	0.00000198842224

$h$	Relative error
$2^{-1}$	0.29744254140026
$2^{-6}$	0.00785334954789
$2^{-11}$	0.00024418036620
$2^{-16}$	0.00000762943394
$2^{-21}$	0.00000023835113
$2^{-26}$	0.0000000429331
$2^{-31}$	0.00000012467100
$2^{-36}$	0.00000495453865

Recall the considerations elaborated in Ex. 1.5.43. Owing to the impact of roundoff errors amplified by cancellation,  $h \rightarrow 0$  does not achieve arbitrarily high accuracy. Rather, we observe fewer correct digits for very small  $h$ !

Extrapolation offers a numerically stable ( $\rightarrow$  Def. 1.5.80) alternative, because for a  $2(n+1)$ -times continuously differentiable function  $f : I \subset \mathbb{R} \mapsto \mathbb{R}$ ,  $x \in I$  we find that the symmetric difference quotient behaves like a polynomial in  $h^2$  in the vicinity of  $h = 0$ . Consider Taylor sum of  $f$  in  $x$  with Lagrange remainder term:

$$\psi(h) := \frac{f(x+h) - f(x-h)}{2h} \sim f'(x) + \sum_{k=1}^n \frac{1}{(2k)!} \frac{d^{2k}f}{dx^{2k}}(x) h^{2k} + \frac{1}{(2n+2)!} f^{(2n+2)}(\xi(x)).$$

Since  $\lim_{h \rightarrow 0} \psi(h) = f'(x)$   $\rightarrow$  approximate  $f'(x)$  by interpolation of  $\psi$  in points  $h_i$ .

#### MATLAB-code 3.2.42: Numerical differentiation by extrapolation to zero

```

1 function d = diffex(f,x,h0,atol,rtol)
2 % f: handle of to a function defined in a neighborhood of x ∈ ℝ,
3 % x: point at which approximate derivative is desired,
4 % h0: initial distance from x,
5 % tol: relative target tolerance
6 h = h0;
7 % Aitken–Neville scheme, see Code 3.2.31 (x = 0!)
8 y(1) = (f(x+h0)-f(x-h0))/(2*h0);
9 for i=2:10
10    h(i) = h(i-1)/2;
11    y(i) = f(x+h(i))-f(x-h(i))/h(i-1);
12    for k=i-1:-1:1
13        y(k) = y(k+1)-(y(k+1)-y(k))*h(i)/(h(i)-h(k));
14    end
15    % termination of extrapolation, when desired tolerance is achieved
16    errest = abs(y(2)-y(1)); % error indicator
17    if ((errest < rtol*abs(y(1))) || (errest < atol)), break; end %
18 end
19 d = y(1);

```

diffex2(@atan, 1.1, 0.5)		diffex2(@sqrt, 1.1, 0.5)		diffex2(@exp, 1.1, 0.5)	
Degree	Relative error	Degree	Relative error	Degree	Relative error
0	0.04262829970946	0	0.02849215135713	0	0.04219061098749
1	0.02044767428982	1	0.01527790811946	1	0.02129207652215
2	0.00051308519253	2	0.00061205284652	2	0.00011487434095
3	0.00004087236665	3	0.00004936258481	3	0.00000825582406
4	0.00000048930018	4	0.00000067201034	4	0.000000000589624
5	0.00000000746031	5	0.00000001253250	5	0.00000000009546
6	0.00000000001224	6	0.000000000004816	6	0.00000000000002
		7	0.0000000000021		

Advantage:

guaranteed accuracy  $\rightarrow$  efficiency

#### 3.2.3.4 Newton basis and divided differences



*Supplementary reading.* We also refer to [15, Sect. 8.2.4], [63, Sect. 8.2].

In § 3.2.29 we have seen a method to evaluate partial polynomial interpolants for a single or a few evaluation points efficiently. Now we want to do this for *many* evaluation points that may not be known when we receive information about the first interpolation points.

**C++code 3.2.43: Polynomial evaluation**

```

1 class PolyEval {
2     private:
3         // evaluation point and various internal data describing the
4         // polynomials
5     public:
6         // Idle Constructor
7         PolyEval();
8         // Add another data point and update internal information
9         void addPoint(t, y);
10        // Evaluation of current interpolating polynomial at x
11        Eigen::VectorXd operator() (const Eigen::VectorXd &x) const;
12    };

```

The challenge: Both `addPoint()` and the evaluation operator may be called many times and the implementation has to remain efficient under these circumstances.

Why not use the techniques from § 3.2.26? Drawback of the Lagrange basis or barycentric formula: adding another data point affects *all* basis polynomials/all precomputed values!

**(3.2.44) Newton basis for  $\mathcal{P}_n$** 

Our tool now: “update friendly” representation: **Newton basis** for  $\mathcal{P}_n$

$$N_0(t) := 1, \quad N_1(t) := (t - t_0), \quad \dots, \quad N_n(t) := \prod_{i=0}^{n-1} (t - t_i). \quad (3.2.45)$$

Note:  $N_n \in \mathcal{P}_n$  with *leading coefficient 1*  $\succ$  linear independence  $\succ$  basis property.

The abstract considerations of § 3.1.11 still apply and we get a linear system of equations for the coefficients  $a_j$  of the polynomial interpolant in Newton basis:

$$a_j \in \mathbb{R}: \quad a_0 N_0(t_j) + a_1 N_1(t_j) + \dots + a_n N_n(t_j) = y_j, \quad j = 0, \dots, n.$$

$\Leftrightarrow$  triangular linear system

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & (t_1 - t_0) & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 1 & (t_n - t_0) & \cdots & \prod_{i=0}^{n-1} (t_n - t_i) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Solution of the system with forward substitution:

$$\begin{aligned} a_0 &= y_0, \\ a_1 &= \frac{y_1 - a_0}{t_1 - t_0} = \frac{y_1 - y_0}{t_1 - t_0}, \end{aligned}$$

$$a_2 = \frac{y_2 - a_0 - (t_2 - t_0)a_1}{(t_2 - t_0)(t_2 - t_1)} = \frac{y_2 - y_0 - (t_2 - t_0)\frac{y_1 - y_0}{t_1 - t_0}}{(t_2 - t_0)(t_2 - t_1)} = \frac{\frac{y_2 - y_0}{t_2 - t_0} - \frac{y_1 - y_0}{t_1 - t_0}}{t_2 - t_1},$$

⋮

Observation: same quantities computed again and again !

### (3.2.46) Divided differences

In order to reveal the pattern, we turn to a new interpretation of the coefficients  $a_j$  of the interpolating polynomials in Newton basis.

Newton basis polynomial  $N_j(t)$ : degree  $j$  and leading coefficient 1  
 $\Rightarrow a_j$  is the leading coefficient of the interpolating polynomial  $p_{0,j}$ .

(the notation  $p_{\ell,m}$  for partial polynomial interpolants through the data points  $(t_\ell, y_\ell), \dots, (t_m, y_m)$  was introduced in Section 3.2.3.2, see (3.2.30))

- Recursion (3.2.30) implies a recursion for the leading coefficients  $a_{\ell,m}$  of the interpolating polynomials  $p_{\ell,m}$ ,  $0 \leq \ell \leq m \leq n$ :

$$a_{\ell,m} = \frac{a_{\ell+1,m} - a_{\ell,m-1}}{t_m - t_\ell}. \quad (3.2.47)$$

Hence, instead of using elimination for a triangular linear system, we find a simpler and more efficient algorithm using the so-called **divided differences**:

$$\begin{aligned} y[t_i] &= y_i \\ y[t_i, \dots, t_{i+k}] &= \frac{y[t_{i+1}, \dots, t_{i+k}] - y[t_i, \dots, t_{i+k-1}]}{t_{i+k} - t_i} \quad (\text{recursion}) \end{aligned} \quad (3.2.48)$$

### (3.2.49) Efficient computation of divided differences

Recursive calculation by **divided differences scheme**, cf. Aitken-Neville scheme, Code 3.2.31:

$$\begin{array}{c|c} t_0 & y[t_0] \\ t_1 & y[t_1] > y[t_0, t_1] \\ t_2 & y[t_2] > y[t_1, t_2] > y[t_0, t_1, t_2] \\ t_3 & y[t_3] > y[t_2, t_3] > y[t_1, t_2, t_3] > y[t_0, t_1, t_2, t_3], \end{array} \quad (3.2.50)$$

The elements can be computed from left to right, every “ $>$ ” indicates the evaluation of the recursion formula (3.2.48).

However, we can again resort to the idea of § 3.2.32 and traverse (3.2.50) along the diagonals from top to bottom: If a new datum  $(t_{n+1}, y_{n+1})$  is added, it is enough to compute the  $n+2$  new terms

$$y[t_{n+1}], y[t_n, t_{n+1}], \dots, y[t_0, \dots, t_{n+1}].$$

The following MATLAB code computes divided differences for data points  $(t_i, y_i)$ ,  $i = 0, \dots, n$ , in this fashion. It is implemented by recursion to elucidate the successive use of data points. The divided differences  $y[t_0], y[t_0, t_1], \dots, y[t_0, \dots, t_n]$ , are accumulated in the vector  $y$ .

#### MATLAB-code 3.2.51: Divided differences, recursive implementation, in situ computation

```

1 function y = divdiff(t,y)
2 n = length(y)-1;
3 if (n > 0)
4 y(1:n) = divdiff(t(1:n),y(1:n));
5 for j=0:n-1
6     y(n+1) = (y(n+1)-y(j+1)) / (t(n+1)-t(j+1)); % Recursion (3.2.48)
7 end
8 end
```

By derivation: computed finite differences are the coefficients of interpolating polynomials in Newton basis:

$$p(t) = a_0 + a_1(t - t_0) + a_2(t - t_0)(t - t_1) + \dots + a_n \prod_{j=0}^{n-1} (t - t_j) \quad (3.2.52)$$

$$a_0 = y[t_0], a_1 = y[t_0, t_1], a_2 = y[t_0, t_1, t_2], \dots$$

Thus, Code 3.2.51 computes the coefficients  $a_j$ ,  $j = 0, \dots, n$ , of the polynomial interpolant with respect to the Newton basis. It uses only the first  $j + 1$  data points to find  $a_j$ .

#### (3.2.53) Efficient evaluation of polynomial in Newton form

“Backward evaluation” of  $p(t)$  in the spirit of Horner’s scheme ( $\rightarrow$  Rem. 3.2.5, [15, Alg. 8.20]):

$$p \leftarrow a_n, \quad p \leftarrow (t - t_{n-1})p + a_{n-1}, \quad p \leftarrow (t - t_{n-2})p + a_{n-2}, \quad \dots$$

#### MATLAB-code 3.2.54: Divided differences evaluation by modified Horner scheme

```

1 function p = evaldivdiff(t,y,x)
2 dd=divdiff(t,y); % Compute divided differences, see Code 3.2.51
3 n = length(y)-1;
4 p=dd(n+1);
5 for j=n:-1:1, p = (x-t(j)).*p+dd(j); end
```

Computational effort:

- \*  $O(n^2)$  for computation of divided differences (“setup phase”),
- \*  $O(n)$  for every single evaluation of  $p(t)$ .

(both operations can be interleaved, see Code 3.2.43)

### Example 3.2.55 (Class PolyEval)

Implementation of a C++ class supporting the efficient update and evaluation of an interpolating polynomial making use of

- presentation in Newton basis (3.2.45),
- computation of representation coefficients through divided difference scheme (3.2.50), see Code 3.2.51,
- evaluation by means of Horner scheme, see Code 3.2.54.

### C++-code 3.2.56: Definition of a class for “update friendly” polynomial interpolant

```

1  class PolyEval {
2  private:
3      std::vector<double> t; // Interpolation nodes
4      std::vector<double> y; // Coefficients in Newton representation
5  public:
6      PolyEval(); // Idle constructor
7      void addPoint(double t, double y); // Add another data point
8      // evaluate value of current interpolating polynomial at x,
9      double operator() (double x) const;
10 private:
11     // Update internal representation, called by addPoint()
12     void divdiff();
13 };

```

### C++-code 3.2.57: Implementation of class PolyEval

```

1 PolyEval::PolyEval() {}
2
3 void PolyEval::addPoint(double td, double yd)
4 { t.push_back(td); y.push_back(yd); divdiff(); }
5
6 void PolyEval::divdiff() {
7     int n = t.size();
8     for(int j=0; j<n-1; j++) y[n-1] = ((y[n-1]-y[j])/(t[n-1]-t[j]));
9 }
10
11 double PolyEval::operator() (double x) const {
12     double s = y.back();
13     for(int i = y.size()-2; i>= 0; --i) s = s*(x-t[i])+y[i];
14     return s;
15 }

```

### Remark 3.2.58 (Divided differences and derivatives)

If  $y_0, \dots, y_n$  are the values of a smooth function  $f$  in the points  $t_0, \dots, t_n$ , that is,  $y_j := f(t_j)$ , then

$$y[t_i, \dots, t_{i+k}] = \frac{f^{(k)}(\xi)}{k!}$$

for a certain  $\xi \in [t_i, t_{i+k}]$ , see [15, Thm. 8.21].

---

### 3.2.4 Polynomial Interpolation: Sensitivity



*Supplementary reading.* For related discussions see [63, Sect. 8.1.3].

---

This section addresses a *major shortcoming of polynomial interpolation* in case the interpolation knots  $t_i$  are imposed, which is usually the case when given data points have to be interpolated, cf. Ex. 3.1.3.

**Example 3.2.59 (Oscillating polynomial interpolant (Runge's counterexample) → [15, Sect. 8.3], [63, Ex. 8.1])**

This example offers a glimpse of the problems haunting polynomial interpolation.

We examine the polynomial Lagrange interpolant (→ Section 3.2.2, (3.2.9)) for uniformly spaced nodes and the following data:

$$\begin{aligned} \mathcal{T} &:= \left\{ -5 + \frac{10}{n} j \right\}_{j=0}^n, \\ y_j &= \frac{1}{1+t_j^2}, \quad j = 0, \dots, n. \end{aligned}$$

Plotted is the interpolant for  $n = 10$  and the interpolant for which the data value at  $t = 0$  has been perturbed by 0.1

(See also Ex. 4.1.34 below.)

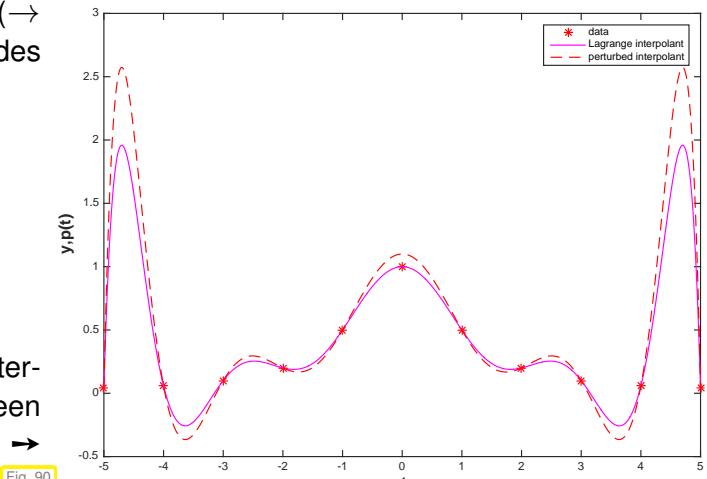


Fig. 90



- ☞ possible strong oscillations of interpolating polynomials of *high degree* on *uniformly spaced nodes*!
- ☞ Slight perturbations of data values can engender strong variations of a *high-degree* Lagrange interpolant “far away”.

#### (3.2.60) Problem map and sensitivity

In Section 1.6.1.2 we introduced the concept of *sensitivity* to describe how perturbations of the data affect the output for a problem map as defined in § 1.5.62. Concretely, in Section 1.6.1.2 we discussed the

sensitivity for linear systems of equations. Motivated by Ex. 3.2.59, we now examine the sensitivity of Lagrange interpolation with respect to perturbations in the data values.

The problem: data  $\leftrightarrow$  values  $y_i$ ,  $i = 0, \dots, n$   $\geq$  data space  $\mathbb{R}^{n+1}$ .  
problem map  $\leftrightarrow$  polynomial interpolation mapping  $I_{\mathcal{T}}$ , see Cor. 3.2.15, result space  $\mathcal{P}_n$ .

Thus, the (pointwise) sensitivity of polynomial interpolation will tell us to what extent perturbations in the  $y$ -data will affect the values of the interpolating function somewhere else. In the case of high sensitivity small perturbations in the data can cause big variations in some function values, which is clearly undesirable.

### (3.2.61) Norms on spaces of functions

Necessary for studying sensitivity of polynomial interpolation in quantitative terms are norms ( $\rightarrow$  Def. 1.5.65) on the vector space of continuous functions  $C(I)$ ,  $I \subset \mathbb{R}$ . The following norms are the most relevant:

$$\text{supremum norm } \|f\|_{L^\infty(I)} := \sup\{|f(t)| : t \in I\}, \quad (3.2.62)$$

$$L^2\text{-norm } \|f\|_{L^2(I)}^2 := \int_I |f(t)|^2 dt, \quad (3.2.63)$$

$$L^1\text{-norm } \|f\|_{L^1(I)} := \int_I |f(t)| dt. \quad (3.2.64)$$

Note the relationship with the vector norms introduced in § 1.5.64.

### (3.2.65) Sensitivity of linear problem maps

In § 3.1.11 we have learned that (polynomial) interpolation gives rise to a **linear** problem map, see Def. 3.1.15. For this class of problem maps the investigation of sensitivity has to study **operator norms**, a generalization of matrix norms ( $\rightarrow$  Def. 1.5.71).

Let  $L : X \rightarrow Y$  be a *linear* problem map between two normed spaces, the data space  $X$  (with norm  $\|\cdot\|_X$ ) and the result space  $Y$  (with norm  $\|\cdot\|_Y$ ). Thanks to linearity, perturbations of the result  $y := L(x)$  for the input  $x \in X$  can be expressed as follows:

$$L(x + \delta x) = L(x) + L(\delta x) = y + L(\delta x).$$

Hence, the sensitivity (in terms of propagation of absolute errors) can be measured by the **operator norm**

$$\|L\|_{X \rightarrow Y} := \sup_{\delta x \in X \setminus \{0\}} \frac{\|L(\delta x)\|_Y}{\delta x \|x\|_X}. \quad (3.2.66)$$

This can be read as the “matrix norm of  $L$ ”, cf. Def. 1.5.71.

It seems challenging to compute the operator norm (3.2.66) for  $L = I_{\mathcal{T}}$  ( $I_{\mathcal{T}}$  the Lagrange interpolation operator for node set  $\mathcal{T} \subset I$ ),  $X = \mathbb{R}^{n+1}$  (equipped with a vector norm), and  $Y = C(I)$  (endowed with a norm from § 3.2.61). The next lemma will provide surprisingly simple concrete formulas.

**Lemma 3.2.67. Absolute conditioning of polynomial interpolation**

Given a mesh  $\mathcal{T} \subset \mathbb{R}$  with generalized Lagrange polynomials  $L_i$ ,  $i = 0, \dots, n$ , and fixed  $I \subset \mathbb{R}$ , the norm of the interpolation operator satisfies

$$\|\mathbf{I}_{\mathcal{T}}\|_{\infty \rightarrow \infty} := \sup_{\mathbf{y} \in \mathbb{R}^{n+1} \setminus \{0\}} \frac{\|\mathbf{I}_{\mathcal{T}}(\mathbf{y})\|_{L^\infty(I)}}{\|\mathbf{y}\|_\infty} = \left\| \sum_{i=0}^n |L_i| \right\|_{L^\infty(I)}, \quad (3.2.68)$$

$$\|\mathbf{I}_{\mathcal{T}}\|_2 \leq \sup_{\mathbf{y} \in \mathbb{R}^{n+1} \setminus \{0\}} \frac{\|\mathbf{I}_{\mathcal{T}}(\mathbf{y})\|_{L^2(I)}}{\|\mathbf{y}\|_2} \leq \left( \sum_{i=0}^n \|L_i\|_{L^2(I)}^2 \right)^{\frac{1}{2}}. \quad (3.2.69)$$

*Proof.* (for the  $L^\infty$ -Norm) By  $\triangle$ -inequality

$$\|\mathbf{I}_{\mathcal{T}}(\mathbf{y})\|_{L^\infty(I)} = \left\| \sum_{j=0}^n y_j L_j \right\|_{L^\infty(I)} \leq \sup_{t \in I} \sum_{j=0}^n |y_j| |L_j(t)| \leq \|\mathbf{y}\|_\infty \left\| \sum_{i=0}^n |L_i| \right\|_{L^\infty(I)},$$

equality in (3.2.68) for  $\mathbf{y} := (\text{sgn}(L_j(t^*)))_{j=0}^n$ ,  $t^* := \text{argmax}_{t \in I} \sum_{i=0}^n |L_i(t)|$ .  $\square$

*Proof.* (for the  $L^2$ -Norm) By  $\triangle$ -inequality and Cauchy-Schwarz inequality

$$\|\mathbf{I}_{\mathcal{T}}(\mathbf{y})\|_{L^2(I)} \leq \sum_{j=0}^n |y_j| \|L_j\|_{L^2(I)} \leq \left( \sum_{j=0}^n |y_j|^2 \right)^{\frac{1}{2}} \left( \sum_{j=0}^n \|L_j\|_{L^2(I)}^2 \right)^{\frac{1}{2}}.$$

$\square$

Terminology: **Lebesgue constant** of  $\mathcal{T}$ :  $\lambda_{\mathcal{T}} := \left\| \sum_{i=0}^n |L_i| \right\|_{L^\infty(I)} = \|\mathbf{I}_{\mathcal{T}}\|_{\infty \rightarrow \infty}$

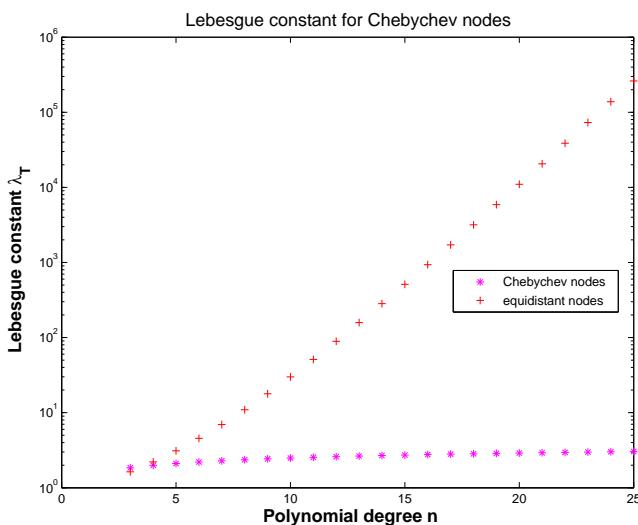
**Remark 3.2.70 (Lebesgue constant for equidistant nodes)**

We consider Lagrange interpolation for the special setting

$$I = [-1, 1], \quad \mathcal{T} = \{-1 + \frac{2k}{n}\}_{k=0}^n \quad (\text{uniformly spaced nodes}).$$

Asymptotic estimate (with (3.2.11) and Stirling formula): for  $n = 2m$

$$|L_m(1 - \frac{1}{n})| = \frac{\frac{1}{n} \cdot \frac{1}{n} \cdot \frac{3}{n} \cdots \frac{n-3}{n} \cdot \frac{n+1}{n} \cdots \frac{2n-1}{n}}{\left( \frac{2}{n} \cdot \frac{4}{n} \cdots \frac{n-2}{n} \cdot 1 \right)^2} = \frac{(2n)!}{(n-1)2^{2n}((n/2)!)^2 n!} \sim \frac{2^{n+3/2}}{\pi(n-1)n}$$



Sophisticated theory [13] gives a lower bound for the Lebesgue constant for uniformly spaced nodes:

$$\lambda_{\mathcal{T}} \geq C e^{n/2}$$

with  $C > 0$  independent of  $n$ .

We can also perform a numerical evaluation of the expression

$$\lambda_{\mathcal{T}} = \left\| \sum_{i=0}^n |L_i| \right\|_{L^\infty(I)},$$

for the Lebesgue constant of polynomial interpolation, see Lemma 3.2.67.

#### MATLAB-code 3.2.71: MATLAB code for approximate computation of Lebesgue constants

```

1 function l = lebesgue(t,N)
2 % Computation of Lebesgue constant of polynomial interpolation with
3 % knots  $t_i$  passed in the row vector t based on
4 % (3.2.68). N specifies the number of sampling points for the approximate
5 % computation of the maximum norm of the Lagrange polynomial on the
6 % interval
7 % [-1,1].
8 n = length(t);
9 den = []; % denominators of normalized Lagrange polynomials relative
10 % to the nodes t
11 for i=1:n
12     den = [den; prod(t(i)-t([1:i-1,i+1:end]))];
13 end
14
15 % Default argument
16 if (nargin < 2), N = 1E5; end
17 l = 0; % return value
18 for x=-1:2/N:1 % sampling points for approximate computation of
19     ||·||_{L^\infty([-1,1])}
20     s = 0;
21     for i = 1:n
22         % v provides value of the normalized Lagrange polynomials
23         v = prod(x-t([1:i-1,i+1:end]))/den(i);
24         s = s+abs(v); % sum over the modulus of the polynomials
25     end
26     l = max(l,s); % maximum of sampled values
27 end
```

Note: In Code 3.2.71 the norm  $\|L_i\|_{L^\infty(I)}$  can be computed only approximate by taking the maximum modulus of function values in many sampling points.

#### (3.2.72) Importance of knowing the sensitivity of polynomial interpolation

In Ex. 3.1.3 we learned that interpolation is an important technique for obtaining a mathematical (and algorithmic) description of a constitutive relationship from measured data.

If the interpolation operator is poorly conditioned, tiny measurement errors will lead to big (local) deviations of the interpolant from its “true” form.

Since measurement errors are inevitable, poorly conditioned interpolation procedures are useless for determining constitutive relationships from measurements.

### 3.3 Shape preserving interpolation

When reconstructing a quantitative dependence of quantities from measurements, first principles from physics often stipulate qualitative constraints, which translate into *shape properties* of the function  $f$ , e.g., when modelling the material law for a gas:

$t_i$  pressure values,  $y_i$  densities  $\geq f$  positive & monotonic.

Notation: given data:  $(t_i, y_i) \in \mathbb{R}^2, i = 0, \dots, n, n \in \mathbb{N}, t_0 < t_1 < \dots < t_n$ .

#### Example 3.3.1 (Magnetization curves)

For many materials physics stipulates properties of the functional dependence of magnetic flux  $B$  from magnetic field strength  $H$ :

- \*  $H \mapsto B(H)$  smooth (at least  $C^1$ ),
- \*  $H \mapsto B(H)$  monotonic (increasing),
- \*  $H \mapsto B(H)$  concave

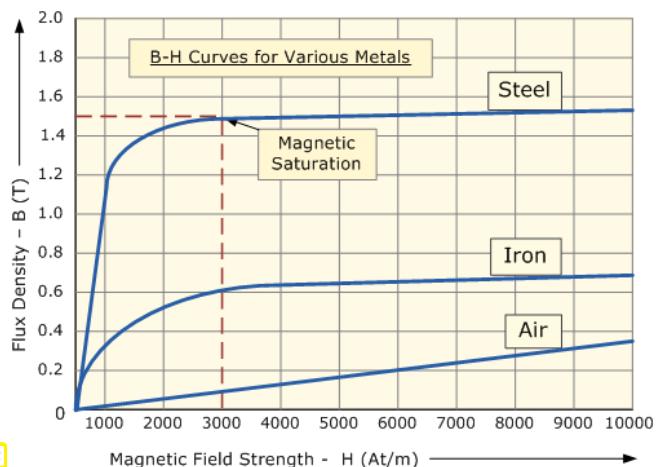


Fig. 92

#### 3.3.1 Shape properties of functions and data

##### (3.3.2) The “shape” of data

The section is about “shape preservation”. In the previous example we have already seen a few properties that constitute the “shape” of a function: sign, monotonicity and curvature. Now we have to identify analogous properties of data sets in the form of sequences of interpolation points  $(t_j, y_j), j = 0, \dots, n, t_j$  pairwise distinct.

##### Definition 3.3.3. monotonic data

The data  $(t_i, y_i)$  are called **monotonic** when  $y_i \geq y_{i-1}$  or  $y_i \leq y_{i-1}$ ,  $i = 1, \dots, n$ .

##### Definition 3.3.4. Convex/concave data

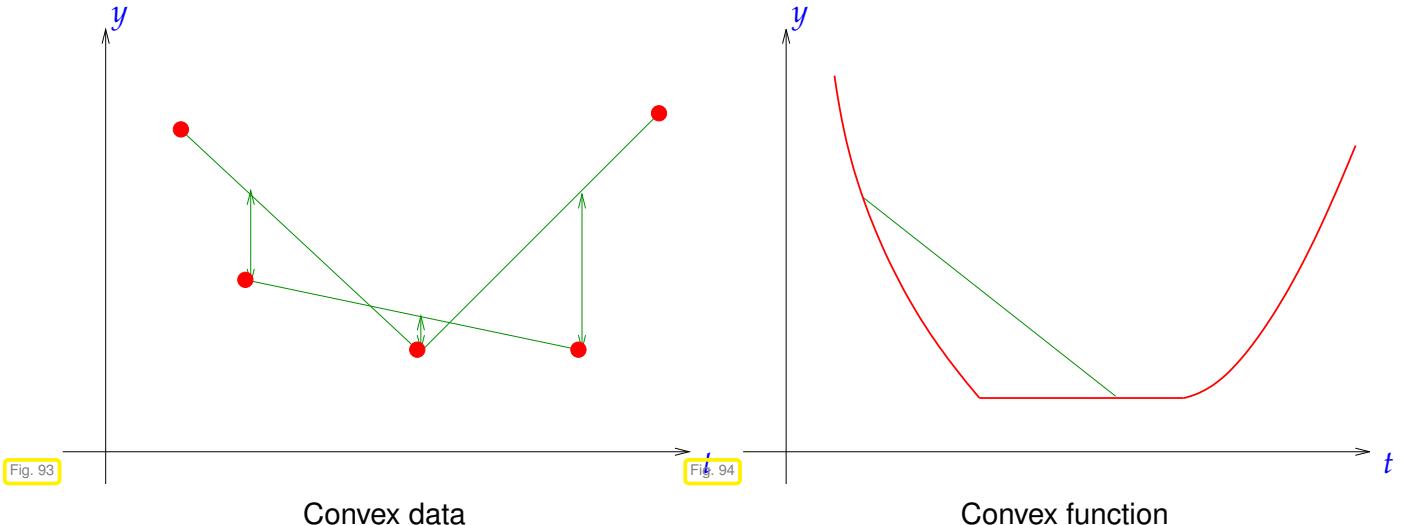
The data  $\{(t_i, y_i)\}_{i=0}^n$  are called **convex** (**concave**) if

$$\Delta_j \stackrel{(\geq)}{\leq} \Delta_{j+1}, \quad j = 1, \dots, n-1, \quad \Delta_j := \frac{y_j - y_{j-1}}{t_j - t_{j-1}}, \quad j = 1, \dots, n.$$

Mathematical characterization of convex data:

$$y_i \leq \frac{(t_{i+1} - t_i)y_{i-1} + (t_i - t_{i-1})y_{i+1}}{t_{i+1} - t_{i-1}} \quad \forall i = 1, \dots, n-1,$$

i.e., each data point lies below the line segment connecting the other data, cf. definition of convexity of a function [77, Def. 5.5.2].



### Definition 3.3.5. Convex/concave function → [77, Def. 5.5.2]

$$f : I \subset \mathbb{R} \mapsto \mathbb{R} \quad \begin{array}{l} \text{convex} \\ \text{concave} \end{array} \quad :\Leftrightarrow \quad \begin{array}{ll} f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) & \forall 0 \leq \lambda \leq 1, \\ f(\lambda x + (1-\lambda)y) \geq \lambda f(x) + (1-\lambda)f(y) & \forall x, y \in I. \end{array}$$

### (3.3.6) (Local) shape preservation

Now consider interpolation problem: data  $(t_i, y_i), i = 0, \dots, n$  → interpolant  $f$

Goal: shape preserving interpolation:

positive data	→	positive interpolant $f$ ,
monotonic data	→	monotonic interpolant $f$ ,
convex data	→	convex interpolant $f$ .

More ambitious goal: local shape preserving interpolation: for each subinterval  $I = (t_i, t_{i+j})$

positive data in $I$	→	locally positive interpolant $f _I$ ,
monotonic data in $I$	→	locally monotonic interpolant $f _I$ ,
convex data in $I$	→	locally convex interpolant $f _I$ .

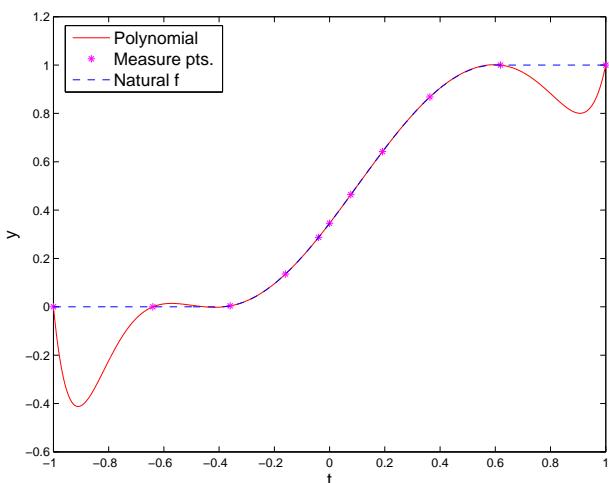
### Experiment 3.3.7 (Bad behavior of global polynomial interpolants)

We perform Lagrange interpolation for the following positive and monotonic data:

$t_i$	-1.0000	-0.6400	-0.3600	-0.1600	-0.0400	0.0000	0.0770	0.1918	0.3631	0.6187	1.0000
$y_i$	0.0000	0.0000	0.0039	0.1355	0.2871	0.3455	0.4639	0.6422	0.8678	1.0000	1.0000

created by taking points on the graph of

$$f(t) = \begin{cases} 0 & \text{if } t < -\frac{2}{5}, \\ \frac{1}{2}(1 + \cos(\pi(t - \frac{3}{5}))) & \text{if } -\frac{2}{5} < t < \frac{3}{5}, \\ 1 & \text{otherwise.} \end{cases}$$



← Interpolating polynomial, degree = 10

Oscillations at the endpoints of the interval (see Fig. 90)

- No locality
- No positivity
- No monotonicity
- No local conservation of the curvature

### 3.3.2 Piecewise linear interpolation

There is a very simple method of achieving perfect shape preservation by means of a linear ( $\rightarrow$  § 3.1.11) interpolation operator into the space of continuous functions:

Data:  $(t_i, y_i) \in \mathbb{R}^2, i = 0, \dots, n, n \in \mathbb{N}, t_0 < t_1 < \dots < t_n$ .

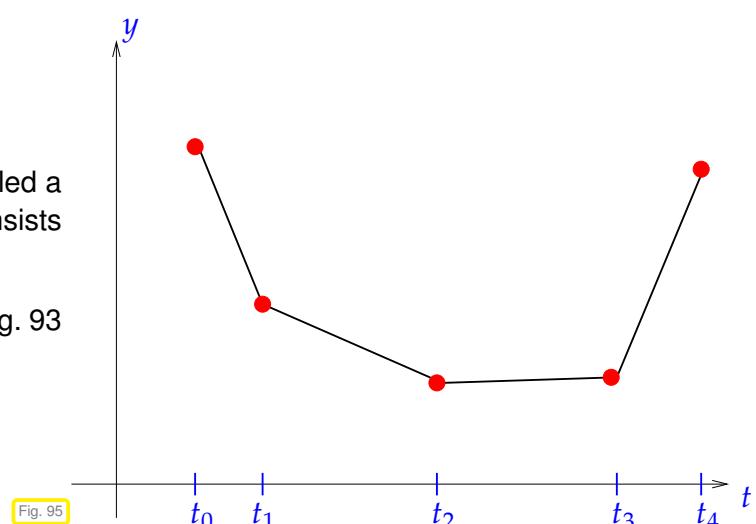
Then the piecewise linear interpolant  $s : [t_0, t_n] \rightarrow \mathbb{R}$  is defined as, cf. Ex. 3.1.8:

$$s(t) = \frac{(t_{i+1} - t)y_i + (t - t_i)y_{i+1}}{t_{i+1} - t_i} \quad \text{for } t \in [t_i, t_{i+1}]. \quad (3.3.8)$$

The piecewise linear interpolant is also called a polygonal curve. It is continuous and consists of  $n$  line segments.

Piecewise linear interpolant of data from Fig. 93

▷



Piecewise linear interpolation means simply “connect the data points in  $\mathbb{R}^2$  using straight lines”.

Obvious: linear interpolation is **linear** (as mapping  $\mathbf{y} \mapsto s$ , see Def. 3.1.15) and **local** in the following sense:

$$y_j = \delta_{ij}, \quad i, j = 0, \dots, n \Rightarrow \text{supp}(s) \subset [t_{i-1}, t_{i+1}]. \quad (3.3.9)$$

As obvious are the properties asserted in the following theorem. The local preservation of curvature is a straightforward consequence of Def. 3.3.4.

### Theorem 3.3.10. Local shape preservation by piecewise linear interpolation

Let  $s \in C([t_0, t_n])$  be the piecewise linear interpolant of  $(t_i, y_i) \in \mathbb{R}^2$ ,  $i = 0, \dots, n$ , for every subinterval  $I = [t_j, t_k] \subset [t_0, t_n]$ :

- |   |   |
|---|---|
| if $(t_i, y_i) _I$ are positive/negative<br>if $(t_i, y_i) _I$ are monotonic (increasing/decreasing)<br>if $(t_i, y_i) _I$ are convex/concave | $\Rightarrow s _I$ is positive/negative,<br>$\Rightarrow s _I$ is monotonic (increasing/decreasing),<br>$\Rightarrow s _I$ is convex/concave. |
|---|---|

Local shape preservation = perfect shape preservation!

Bad news: none of this properties carries over to local polynomial interpolation of higher polynomial degree  $d > 1$ .

### Example 3.3.11 (Piecewise quadratic interpolation)

We consider the following generalization of piecewise linear interpolation of data points  $(t_j, y_j) \in \mathbb{R} \times \mathbb{R}$ ,  $j = 0, \dots, n$ .

From Thm. 3.2.14 we know that a parabola (polynomial of degree 2) is uniquely determined by 3 data points. Thus, the idea is to form groups of three adjacent data points and interpolate each of these triplets by a 2nd-degree polynomial (parabola).

Assume:  $n = 2m$  even

► piecewise quadratic interpolant  $q : [\min\{t_i\}, \max\{t_i\}] \mapsto \mathbb{R}$  is defined by

$$q_j := q|_{[t_{2j-2}, t_{2j}]} \in \mathcal{P}_2, \quad q_j(t_i) = y_i, \quad i = 2j-2, 2j-1, 2j, \quad j = 1, \dots, m. \quad (3.3.12)$$

Nodes as in Exp. 3.3.7

Piecewise linear (blue) and quadratic (red) interpolants



No shape preservation for piecewise quadratic interpolant

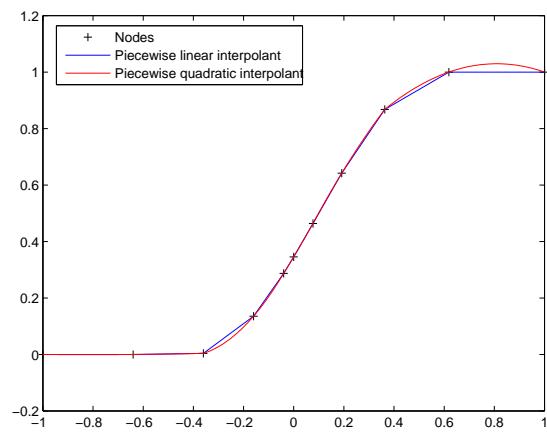


Fig. 96

The “only” drawback of piecewise linear interpolation:

interpolant is only  $C^0$  but not  $C^1$  (no continuous derivative).

However: Interpolant usually serves as input for other numerical methods like a Newton-method for solving non-linear systems of equations, see Section 2.4, which requires derivatives.

## 3.4 Cubic Hermite Interpolation

Aim: construct local shape-preserving ( $\rightarrow$  Section 3.3) (linear ?) interpolation operator that fixes shortcoming of piecewise linear interpolation by ensuring  $C^1$ -smoothness of the interpolant.

☞ notation:  $C^1([a, b]) \hat{=} \text{space of continuously differentiable functions } [a, b] \mapsto \mathbb{R}$ .

### 3.4.1 Definition and algorithms

Given: mesh points  $(t_i, y_i) \in \mathbb{R}^2, i = 0, \dots, n, t_0 < t_1 < \dots < t_n$ .

Goal: build function  $f \in C^1([t_0, t_n])$  satisfying the interpolation conditions  $f(t_i) = y_i, i = 0, \dots, n$ .

#### Definition 3.4.1. Cubic Hermite polynomial interpolant

Given data points  $(t_j, y_j) \in \mathbb{R} \times \mathbb{R}, j = 0, \dots, n$ , with pairwise distinct ordered nodes  $t_j$ , and slopes  $c_j \in \mathbb{R}$ , the piecewise cubic Hermite interpolant  $s : [t_0, t_n] \rightarrow \mathbb{R}$  is defined by the requirements

$$s|_{[t_{i-1}, t_i]} \in \mathcal{P}_3, \quad i = 1, \dots, n, \quad s(t_i) = y_i, \quad s'(t_i) = c_i, \quad i = 0, \dots, n.$$

#### Corollary 3.4.2. Smoothness of cubic Hermite polynomial interpolant

Piecewise cubic Hermite interpolants are continuously differentiable on their interval of definition.

*Proof.* The assertion of the corollary follows from the agreement of function values and first derivative values on nodes shared by two intervals, on each of which the piecewise cubic Hermite interpolant is a polynomial of degree 3.  $\square$

#### (3.4.3) Local representation of piecewise cubic Hermite interpolant

Locally, we can write a piecewise cubic Hermite interpolant as a linear combination of generalized cardinal basis functions with coefficients supplied by the data values  $y_j$  and the slopes  $c_j$ :

►  $s(t) = y_{i-1}H_1(t) + y_iH_2(t) + c_{i-1}H_3(t) + c_iH_4(t), \quad t \in [t_{i-1}, t_i], \quad (3.4.4)$

where the basic functions  $H_k, k = 1, 2, 3, 4$ , are as follows:

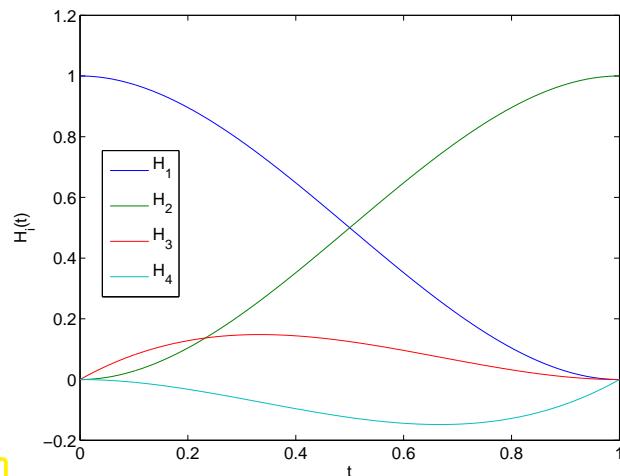


Fig. 97

$$\begin{aligned}
 H_1(t) &:= \phi\left(\frac{t_i-t}{h_i}\right), & H_2(t) &:= \phi\left(\frac{t-t_{i-1}}{h_i}\right), \\
 H_3(t) &:= -h_i\psi\left(\frac{t_i-t}{h_i}\right), & H_4(t) &:= h_i\psi\left(\frac{t-t_{i-1}}{h_i}\right), \\
 h_i &:= t_i - t_{i-1}, \\
 \phi(\tau) &:= 3\tau^2 - 2\tau^3, \\
 \psi(\tau) &:= \tau^3 - \tau^2.
 \end{aligned} \tag{3.4.5}$$

◁ Local basis polynomial on  $[0, 1]$

By tedious, but straightforward computations using the chain rule we find the following values for  $H_k$  and  $H'_k$  at the endpoints of the interval  $[t_{i-1}, t_i]$ .

	$H(t_{i-1})$	$H(t_i)$	$H'(t_{i-1})$	$H'(t_i)$
$H_1$	1	0	0	0
$H_2$	0	1	0	0
$H_3$	0	0	1	0
$H_4$	0	0	0	1

This amounts to a proof for (3.4.4) (why?).

The formula (3.4.4) is handy for the local evaluation of piecewise cubic Hermite interpolants.

Piecewise cubic polynomial  $s$  on  $t_1, t_2$  with  $s(t_1) = y_1, s(t_2) = y_2, s'(t_1) = c_1, s'(t_2) = c_2$ :

efficient local evaluation →

#### MATLAB-code 3.4.6: Local evaluation of cubic Hermite polynomial

```

1 function
2     s=hermloceval(t,t1,t2,y1,y2,c1,c2)
3 % y1, y2: data values, c1, c2: slopes
4 h = t2-t1; t = (t-t1)/h;
5 a1 = y2-y1; a2 = a1-h*c1;
6 a3 = h*c2-a1-a2;
7 s = y1+(a1+(a2+a3*t).* (t-1)).*t;

```

#### (3.4.7) Linear Hermite interpolation

However, the data for an interpolation problem (→ Section 3.1) are merely the interpolation points  $(t_j, y_j)$ ,  $j = 0, \dots, n$ , but not the slopes of the interpolant at the nodes. Thus, in order to define an interpolation operator into the space of piecewise cubic Hermite functions, we have supply a mapping  $\mathbb{R}^{n+1} \times \mathbb{R}^{n+1} \rightarrow \mathbb{R}^{n+1}$  computing the slopes  $c_j$  from the data points.

Since this mapping should be local it is natural to rely on (weighted) averages of the local slopes  $\Delta_j$  (→ Def. 3.3.4) of the data, for instance

$$c_i = \begin{cases} \Delta_1 & , \text{ for } i = 0, \\ \Delta_n & , \text{ for } i = n, \\ \frac{t_{i+1}-t_i}{t_{i+1}-t_{i-1}}\Delta_i + \frac{t_i-t_{i-1}}{t_{i+1}-t_{i-1}}\Delta_{i+1} & , \text{ if } 1 \leq i < n. \end{cases} \quad , \quad \Delta_j := \frac{y_j - y_{j-1}}{t_j - t_{j-1}}, j = 1, \dots, n. \tag{3.4.8}$$

- Leads to a **linear** ( $\rightarrow$  Def. 3.1.15), **local** Hermite interpolation operator

“Local” means, that, if the values  $y_j$  are non-zero for only a few adjacent data points with indices  $j = k, \dots, k+m$ ,  $m \in \mathbb{N}$  small, then the Hermite interpolant  $s$  is supported on  $[t_{k-\ell}, t_{k+m+\ell}]$  for small  $\ell \in \mathbb{N}$  independent of  $k$  and  $m$ .

### Example 3.4.9 (Piecewise cubic Hermite interpolation)

Data points:

- \* 11 equispaced nodes

$$t_j = -1 + 0.2j, \quad j = 0, \dots, 10.$$

in the interval  $I = [-1, 1]$ ,

- \*  $y_i = f(t_i)$  with

$$f(x) := \sin(5x) e^x.$$

Here we used weighted averages of slopes as in Eq. (3.4.8).

For details see Code 3.4.10.

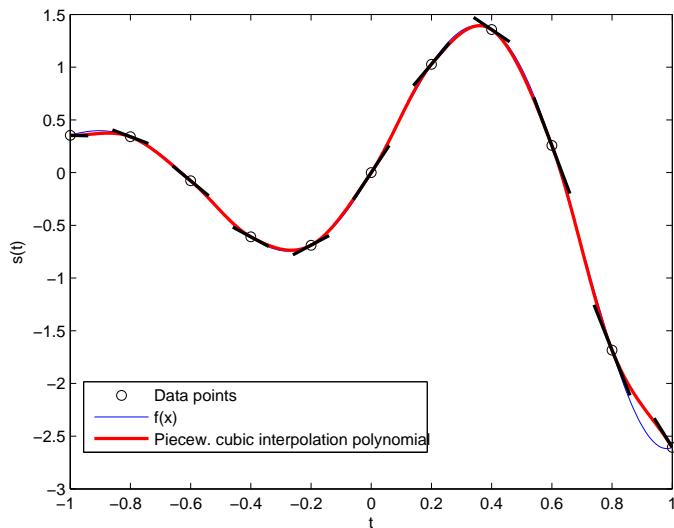


Fig. 98



No local/global preservation of monotonicity!

### MATLAB-code 3.4.10: Piecewise cubic Hermite interpolation

```

1 function hermintpl(f,t)
2 % compute and plot the cubic Hermite interpolant of the function f in
3 % the nodes t
4 % using weighted averages according to (3.4.8) as local slopes
5 n = length(t); h = diff(t); % computes lengths of intervals between
6 % nodes: h_i := t_{i+1} - t_i
7 y = f(t); % f must support collective evaluation for row vector
8 % argument
9 delta = diff(y)./h; % slopes of piecewise linear interpolant
10 c = [delta(1),...
11     ((h(2:end).*delta(1:end-1)+h(1:end-1).*delta(2:end))...
12     ./ (t(3:end) - t(1:end-2))),...
13     delta(end)]; % slopes from weighted average, see (3.4.8)
14
15 figure('Name','Hermite Interpolation');
16 plot(t,y,'ko'); hold on; % plot data points
17 fplot(f,[t(1), t(n)]);
18 for j=1:n-1 % compute and plot the Hermite interpolant with slopes c
19     vx = linspace(t(j),t(j+1), 100);
20     plot(vx,hermloceval(vx,t(j),t(j+1),y(j),y(j+1),c(j),c(j+1)), 'r-',
21          'LineWidth',2);

```

```

18 end
19 for j=2:n-1 % plot segments indicating the slopes c_i
20 plot([t(j)-0.3*h(j-1),t(j)+0.3*h(j)],...
21 [y(j)-0.3*h(j-1)*c(j),y(j)+0.3*h(j)*c(j)],'k-','LineWidth',2);
22 end
23 plot([t(1),t(1)+0.3*h(1)], [y(1),y(1)+0.3*h(1)*c(1)],'k-',
24 'LineWidth',2);
25 plot([t(end)-0.3*h(end),t(end)], [y(end)-0.3*h(end)*c(end),y(end)],'k-',
26 'LineWidth',2);
27 xlabel('t'); ylabel('s(t)');
28 legend('Data points','f(x)','Piecew. cubic interpolation polynomial');
29 hold off;

```

Invocation: `hermintpl( @(x) sin(5*x).*exp(x), [-1:0.2:1]);`

### 3.4.2 Local monotonicity preserving Hermite interpolation

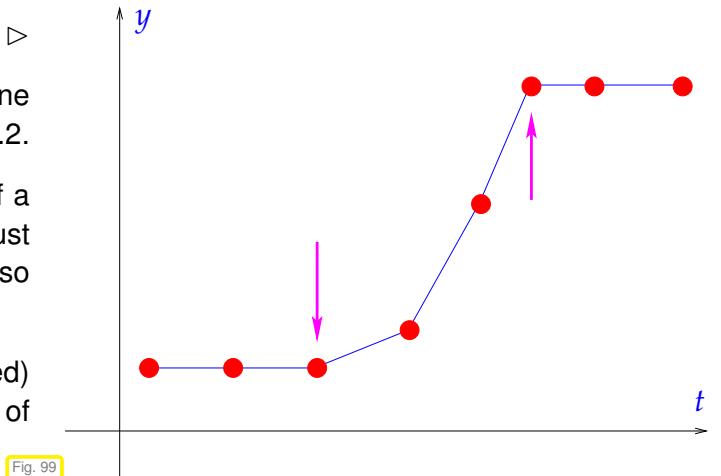
From Ex. 3.4.9 we learn that, if the slopes are chosen according to Eq. (3.4.8), then the resulting Hermite interpolation does not preserve monotonicity.

Consider the situation sketched on the right  $\Rightarrow$

The red circles ( $\bullet$ ) represent data points, the blue line ( $-$ ) the piecewise linear interpolant  $\rightarrow$  Section 3.3.2.

In the nodes marked with  $\mapsto$  the first derivative of a monotonicity preserving  $C^1$ -smooth interpolant must vanish! Otherwise an “overshoot” occurs, see also Fig. 96.

Of course, this will be violated, when a (weighted) arithmetic average is used for the computation of slopes for cubic Hermite interpolation.



$\triangleleft$  Consider the situation sketched on the left.

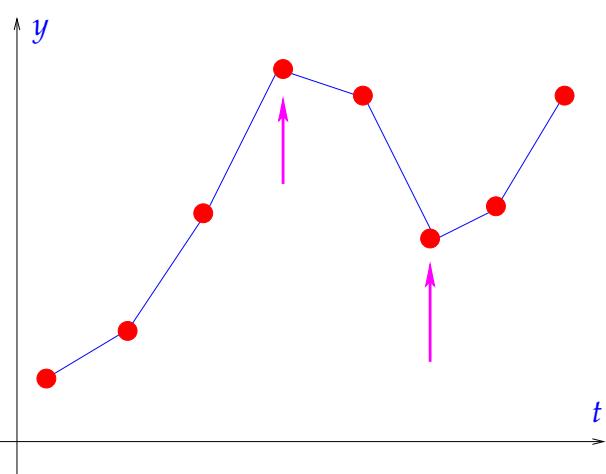
The red circles ( $\bullet$ ) represent data points, the blue line ( $-$ ) the piecewise linear interpolant  $\rightarrow$  Section 3.3.2.

A *local* monotonicity preserving  $C^1$ -smooth interpolant ( $\rightarrow$  § 3.3.6)  $s$  must be flat (= vanishing first derivative) in data points  $(t_j, y_j)$ , for which

$$\begin{aligned} y_{j-1} < y_j \quad &\text{and} \quad y_{j+1} < y_j, \\ y_{j-1} > y_j \quad &\text{and} \quad y_{j+1} > y_j, \end{aligned}$$

$\triangleleft$  in “local extrema” of the data set.

Otherwise, overshoots or undershoots would destroy local monotonicity on one side of the extremum.



### (3.4.11) Limiting of local slopes

From the discussion of Fig. 99 and Fig. 100 it is clear that local monotonicity preservation entails that the local slopes  $c_i$  of a cubic Hermite interpolant ( $\rightarrow$  Def. 3.4.1) have to fulfill

$$c_i = \begin{cases} 0 & , \text{ if } \operatorname{sgn}(\Delta_i) \neq \operatorname{sgn}(\Delta_{i+1}) , \\ \text{some "average" of } \Delta_i, \Delta_{i+1} & \text{otherwise} \end{cases}, \quad i = 1, \dots, n-1 . \quad (3.4.12)$$

notation: sign function  $\operatorname{sgn}(\xi) = \begin{cases} 1 & , \text{ if } \xi > 0 , \\ 0 & , \text{ if } \xi = 0 , \\ -1 & , \text{ if } \xi < 0 . \end{cases}$

A slope selection rule that enforces (3.4.12) is called a **limiter**.

Of course, testing for equality with zero does not make sense for data that may be affected by measurement or roundoff errors. Thus, the “average” in (3.4.12) must be close to zero already when either  $\Delta_i \approx 0$  or  $\Delta_{i+1} \approx 0$ . This is satisfied by the **weighted harmonic mean**

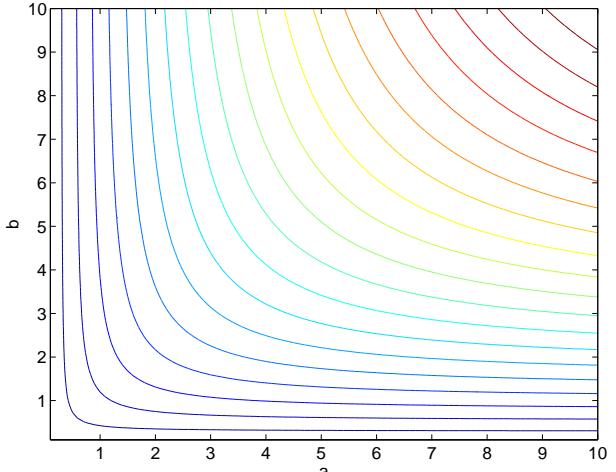
$$c_i = \frac{1}{\frac{w_a}{\Delta_i} + \frac{w_b}{\Delta_{i+1}}}, \quad (3.4.13)$$

with weights  $w_a > 0, w_b > 0, (w_a + w_b = 1)$ .

The harmonic mean = “smoothed  $\min(\cdot, \cdot)$ -function”.

Obviously  $\Delta_i \rightarrow 0$  or  $\Delta_{i+1} \rightarrow 0$  in (3.4.13), then  $c_i \rightarrow 0$ .

Contour plot of the harmonic mean of  $a$  and  $b$   $\rightarrow$  ( $w_a = w_b = 1/2$ ).



A good choice of the weights is:

$$w_a = \frac{2h_{i+1} + h_i}{3(h_{i+1} + h_i)}, \quad w_b = \frac{h_{i+1} + 2h_i}{3(h_{i+1} + h_i)},$$

This yields the following local slopes, unless (3.4.12) enforces  $c_i = 0$ :

$$\operatorname{sgn}(\Delta_1) = \operatorname{sgn}(\Delta_2) \quad c_i = \begin{cases} \Delta_1 & , \text{ if } i = 0 , \\ \frac{3(h_{i+1} + h_i)}{\frac{2h_{i+1} + h_i}{\Delta_i} + \frac{2h_i + h_{i+1}}{\Delta_{i+1}}} & , \text{ for } i \in \{1, \dots, n-1\} , \quad h_i := t_i - t_{i-1} . \\ \Delta_n & , \text{ if } i = n , \end{cases} \quad (3.4.14)$$

Piecewise cubic Hermite interpolation with local slopes chosen according to (3.4.12) and (3.4.14) is available through the MATLAB function  $v = \text{pchip}(t, y, x)$ , where  $t$  passes the interpolation nodes,  $y$  the corresponding data values, and  $x$  is a vector of evaluation points, see `doc phchip` for details.

**Example 3.4.15 (Monotonicity preserving piecewise cubic polynomial interpolation)**

Data from Exp. 3.3.7

Plot created with MATLAB-function call

```
v = pchip(t,y,x);
```

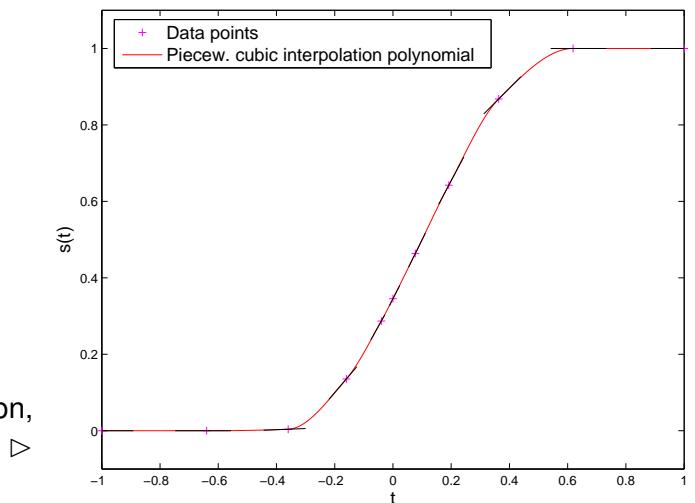
t: Data nodes  $t_j$

y: Data values  $y_j$

x: Evaluation points  $x_i$

v: Vector  $s(x_i)$

We observe perfect local monotonicity preservation,  
no under- or overshoots at extrema.


**Remark 3.4.16 (Non-linear cubic Hermite interpolation)**

Note that the mapping  $\mathbf{y} := [y_0, \dots, y_n] \rightarrow c_i$  defined by (3.4.12) and (3.4.14) is **not linear**.

- The “pchip” interpolator operator does **not** provide a **linear** mapping from data space  $\mathbb{R}^{n+1}$  into  $C^1([t_0, t_n])$  (in the sense of Def. 3.1.15).

In fact, the non-linearity of the piecewise cubic Hermite interpolation operator is necessary for (only global) monotonicity preservation:

**Theorem 3.4.17. Property of linear, monotonicity preserving interpolation into  $C^1$** 

If, for fixed node set  $\{t_j\}_{j=0}^n$ ,  $n \geq 2$ , an interpolation scheme  $I : \mathbb{R}^{n+1} \rightarrow C^1(I)$  is **linear** as a mapping from data values to continuous functions on the interval covered by the nodes ( $\rightarrow$  Def. 3.1.15), and **monotonicity preserving**, then  $I(\mathbf{y})'(t_j) = 0$  for all  $\mathbf{y} \in \mathbb{R}^{n+1}$  and  $j = 1, \dots, n-1$ .

Of course, an interpolant that is flat in all data points, as stipulated by Thm. 3.4.17 for a linear, monotonicity preserving,  $C^1$ -smooth interpolation scheme, does not make much sense.

At least, the piecewise cubic Hermite interpolation operator is **local** (in the sense discussed in § 3.4.7).

**Theorem 3.4.18. Monotonicity preservation of limited cubic Hermite interpolation**

The cubic Hermite interpolation polynomial with slopes as in Eq. (3.4.14) provides a local monotonicity-preserving  $C^1$ -interpolant.

*Proof.* See F. FRITSCH UND R. CARLSON, *Monotone piecewise cubic interpolation*, SIAM J. Numer. Anal., 17 (1980), S. 238–246.



The next code demonstrates the calculation of the slopes  $c_i$  in MATLAB's pchip (details in [23]):

#### MATLAB-code 3.4.19: Monotonicity preserving slopes in pchip

```

1 function c = pchipslopes(t,y)
2 % Calculation of local slopes  $c_i$  for shape preserving cubic Hermite
   % interpolation, see (3.4.12), (3.4.14)
3 % t, y are row vectors passing the data points
4 n = length(t); h = diff(t); delta = diff(y)./h; % linear slopes
5 c = zeros(size(h));
6 k = find(sign(delta(1:n-2))..*sign(delta(2:n-1))>0)+1;
7 % Compute reconstruction slope according to (3.4.14)
8 w1 = 2*h(k)+h(k-1); w2 = h(k)+2*h(k-1);
9 c(k) = (w1+w2)./(w1./delta(k-1) + w2./delta(k));
10 % Special slopes at endpoints, beyond (3.4.14)
11 c(1) = pchipend(h(1),h(2),delta(1),delta(2));
12 c(n) = pchipend(h(n-1),h(n-2),delta(n-1),delta(n-2));
13
14 function d = pchipend(h1,h2,dell1,del2)
15 % Noncentered, shape-preserving, three-point formula.
16 d = ((2*h1+h2)*dell1 - h1*del2)/(h1+h2);
17 if sign(d) ~= sign(dell1), d = 0;
18 elseif (sign(dell1) ~= sign(del2)) (abs(d)>abs(3*dell1)), d = 3*dell1;
   end
```

## 3.5 Splines



*Supplementary reading.* Splines are also presented in [15, Ch. 9].

Piecewise cubic Hermite Interpolation presented in Section 3.4 entailed determining reconstruction slopes  $c_i$ . Now we learn about a way how to do piecewise polynomial interpolation, which results in  $C^k$ -interpolants,  $k > 0$ , and dispenses with auxiliary slopes. The idea is to obtain the missing conditions implicitly from extra continuity conditions.

#### Definition 3.5.1. Spline space → [63, Def. 8.1]

Given an interval  $I := [a, b] \subset \mathbb{R}$  and a knot set/mesh  $\mathcal{M} := \{a = t_0 < t_1 < \dots < t_{n-1} < t_n = b\}$ , the vector space  $\mathcal{S}_{d,\mathcal{M}}$  of the **spline functions** of degree  $d$  (or order  $d+1$ ) is defined by

$$\mathcal{S}_{d,\mathcal{M}} := \{s \in C^{d-1}(I) : s_j := s|_{[t_{j-1}, t_j]} \in \mathcal{P}_d \quad \forall j = 1, \dots, n\} .$$

↑  $d-1$ -times continuously differentiable      ↙ locally polynomial of degree  $d$

Obviously, spline spaces are mapped onto each other by differentiation & integration:

$$s \in \mathcal{S}_{d,\mathcal{M}} \Rightarrow s' \in \mathcal{S}_{d-1,\mathcal{M}} \quad \wedge \quad \int_a^t s(\tau) d\tau \in \mathcal{S}_{d+1,\mathcal{M}} .$$

Spline spaces of the lowest degrees:

- $d = 0$  :  $\mathcal{M}$ -piecewise constant *discontinuous* functions
- $d = 1$  :  $\mathcal{M}$ -piecewise linear *continuous* functions
- $d = 2$  : *continuously differentiable*  $\mathcal{M}$ -piecewise quadratic functions

The dimension of spline space can be found by a **counting argument** (heuristic): We count the number of “degrees of freedom” (d.o.f.s) possessed by a  $\mathcal{M}$ -piecewise polynomial of degree  $d$ , and subtract the number of **linear constraints** implicit contained in Def. 3.5.1:

$$\dim \mathcal{S}_{d,\mathcal{M}} = n \cdot \dim \mathcal{P}_d - \#\{C^{d-1} \text{ continuity constraints}\} = n \cdot (d+1) - (n-1) \cdot d = n + d.$$

We already know the special case of interpolation in  $\mathcal{S}_{1,\mathcal{M}}$ , when the interpolation nodes are the knots of  $\mathcal{M}$ , because this boils down to simple piecewise linear interpolation, see Section 3.3.2.

### 3.5.1 Cubic spline interpolation



*Supplementary reading.* More details in [42, XIII, 46], [63, Sect. 8.6.1].

Cognitive psychology teaches us that the human eye perceives a  $C^2$ -functions as “smooth”, while it can still spot the abrupt change of curvature at the possible discontinuities of the second derivatives of a cubic Hermite interpolant ( $\rightarrow$  Def. 3.4.1).

For this reason the simplest spline functions featuring  $C^2$ -smoothness are of great importance in computer aided design (CAD). They are the **cubic splines**,  $\mathcal{M}$ -piecewise polynomials of degree 3 contained in  $\mathcal{S}_{3,\mathcal{M}}$  ( $\rightarrow$  Def. 3.5.1).

#### (3.5.2) Definition of cubic spline interpolant

In this section we study **cubic spline interpolation** (related to cubic Hermite interpolation, Section 3.4)

Task: Given a mesh  $\mathcal{M} := \{t_0 < t_1 < \dots < t_n\}$ ,  $n \in \mathbb{N}$ , “find” a cubic spline  $s \in \mathcal{S}_{3,\mathcal{M}}$  that complies with the interpolation conditions

$$s(t_j) = y_j \quad , \quad j = 0, \dots, n. \quad (3.5.3)$$

$\hat{=}$  interpolation at knots !

From **dimensional considerations** it is clear that the interpolation conditions will fail to fix the interpolating cubic spline uniquely:

$$\dim \mathcal{S}_{3,\mathcal{M}} - \#\{\text{interpolation conditions}\} = (n+3) - (n+1) = 2 \text{ free d.o.f.}$$

“two conditions are missing”  $\succ$  interpolation problem is not yet well defined!

### (3.5.4) Computing cubic spline interpolants

We opt for a linear interpolation scheme ( $\rightarrow$  Def. 3.1.15) into the spline space  $\mathcal{S}_{3,\mathcal{M}}$ . As explained in § 3.1.11, this will lead to an equivalent linear system of equations for expansion coefficients with respect to a suitable basis.

We reuse the local representation of a cubic spline through cubic Hermite cardinal basis polynomials from (3.4.5):

$$\blacktriangleright \quad s|_{[t_{j-1}, t_j]}(t) = \begin{aligned} & s(t_{j-1}) \cdot (1 - 3\tau^2 + 2\tau^3) + \\ & s(t_j) \cdot (3\tau^2 - 2\tau^3) + \\ & h_j s'(t_{j-1}) \cdot (\tau - 2\tau^2 + \tau^3) + \\ & h_j s'(t_j) \cdot (-\tau^2 + \tau^3), \end{aligned} \quad (3.5.5)$$

with  $h_j := t_j - t_{j-1}$ ,  $\tau := (t - t_{j-1})/h_j$ .

$\geq$  Task of cubic spline interpolation boils down to finding slopes  $s'(t_j)$  in the knots of the mesh  $\mathcal{M}$ .

Once these slopes are known, the efficient local evaluation of a cubic spline function can be done as for a cubic Hermite interpolant, see Section 3.4.1, Code 3.4.6.

Note: if  $s(t_j)$ ,  $s'(t_j)$ ,  $j = 0, \dots, n$ , are fixed, then the representation Eq. (3.5.5) already guarantees  $s \in C^1([t_0, t_n])$ , cf. the discussion for cubic Hermite interpolation, Section 3.4.

$\geq$  only continuity of  $s''$        $\bullet$  has to be enforced by choice of  $s'(t_j)$   
 $\Updownarrow$   
 $\bullet$  will yield extra conditions to fix the  $s'(t_j)$

However, do the

$\bullet$  interpolation conditions Eq. (3.5.3)  $s(t_j) = y_j$ ,  $j = 0, \dots, n$ , and the  
 $\bullet$  smoothness constraint  $s \in C^2([t_0, t_n])$

uniquely determine the unknown slopes  $c_j := s'(t_j)$  ?

From  $s \in C^2([t_0, t_n])$  we obtain  $n - 1$  continuity constraints for  $s''(t)$  at the internal nodes

$$s''|_{[t_{j-1}, t_j]}(t_j) = s''|_{[t_j, t_{j+1}]}(t_j), \quad j = 1, \dots, n - 1. \quad (3.5.6)$$

Based on Eq. (3.5.5), we express Eq. (3.5.6) in concrete terms, using

$$\begin{aligned} s''|_{[t_{j-1}, t_j]}(t) &= s(t_{j-1})h_j^{-2}( -1 + 2\tau ) + s(t_j)h_j^{-2}( 1 - 2\tau ) \\ &\quad + h_j^{-1}s'(t_{j-1})( -4 + 6\tau ) + h_j^{-1}s'(t_j)( -2 + 6\tau ), \quad \tau := (t - t_{j-1})/h_j, \end{aligned} \quad (3.5.7)$$

which can be obtained by the chain rule and from  $\frac{d\tau}{dt} = h_j^{-1}$ .

$$\begin{aligned} \xrightarrow{\text{Eq. (3.5.7)}} \quad s''|_{[t_{j-1}, t_j]}(t_{j-1}) &= -6 \cdot s(t_{j-1})h_j^{-2} + 6 \cdot s(t_j)h_j^{-2} - 4 \cdot h_j^{-1}s'(t_{j-1}) - 2 \cdot h_j^{-1}s'(t_j), \\ s''|_{[t_{j-1}, t_j]}(t_j) &= 6 \cdot s(t_{j-1})h_j^{-2} + -6 \cdot s(t_j)h_j^{-2} + 2 \cdot h_j^{-1}s'(t_{j-1}) + 4 \cdot h_j^{-1}s'(t_j). \end{aligned}$$

Eq. (3.5.6)  $\rightarrow$   $n - 1$  linear equations for  $n$  slopes  $c_j := s'(t_j)$ ; with  $s(t_i) = y_i$ ,

$$\frac{1}{h_j} c_{j-1} + \left( \frac{2}{h_j} + \frac{2}{h_{j+1}} \right) c_j + \frac{1}{h_{j+1}} c_{j+1} = 3 \left( \frac{y_j - y_{j-1}}{h_j^2} + \frac{y_{j+1} - y_j}{h_{j+1}^2} \right), \quad (3.5.8)$$

for  $j = 1, \dots, n - 1$ .

Eq. (3.5.8)  $\Leftrightarrow$  underdetermined  $(n - 1) \times (n + 1)$  linear system of equations.

$n - 1$ : no. of interpolation conditions

$n + 1$ : dimension of cubic spline space on knot set  $\{t_0 < t_1 < \dots < t_n\}$

$$\begin{bmatrix} b_0 & a_1 & b_1 & 0 & \cdots & \cdots & 0 \\ 0 & b_1 & a_2 & b_2 & & & \\ 0 & \ddots & \ddots & \ddots & & \vdots & \\ \vdots & & \ddots & \ddots & \ddots & & \\ & & & \ddots & a_{n-2} & b_{n-2} & 0 \\ 0 & \cdots & \cdots & 0 & b_{n-2} & a_{n-1} & b_{n-1} \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} 3 \left( \frac{y_1 - y_0}{h_1^2} + \frac{y_2 - y_1}{h_2^2} \right) \\ \vdots \\ 3 \left( \frac{y_{n-1} - y_{n-2}}{h_{n-1}^2} + \frac{y_n - y_{n-1}}{h_n^2} \right) \end{bmatrix}. \quad (3.5.9)$$

with

$$\begin{aligned} b_i &:= \frac{1}{h_{i+1}}, \quad i = 0, 1, \dots, n - 1, \\ a_i &:= \frac{2}{h_i} + \frac{2}{h_{i+1}}, \quad i = 0, 1, \dots, n - 1. \\ [b_i, a_i > 0, \quad a_i = 2(b_i + b_{i-1})] \end{aligned}$$

$\rightarrow$  two additional constraints are required, as already noted in § 3.5.2.

### (3.5.10) Types of cubic spline interpolants

To saturate the remaining two degrees of freedom the following three approaches are popular:

① **Complete cubic spline interpolation:**  $s'(t_0) = c_0, s'(t_n) = c_n$  prescribed.

Then the first and last column can be removed from the system matrix of (3.5.9). Their products with  $c_0$  and  $c_n$ , respectively, have to be subtracted from the right hand side of (3.5.9).

② **Natural cubic spline interpolation:**  $s''(t_0) = s''(t_n) = 0$

$$\frac{2}{h_1} c_0 + \frac{1}{h_1} c_1 = 3 \frac{y_1 - y_0}{h_1^2}, \quad \frac{1}{h_n} c_{n-1} + \frac{2}{h_n} c_n = 3 \frac{y_n - y_{n-1}}{h_n^2}.$$

Combining these two extra equations with (3.5.9), we arrive at a linear system of equations with tridiagonal s.p.d. ( $\rightarrow$  Def. 1.1.8, Lemma 1.8.12) system matrix and unknowns  $c_0, \dots, c_n$ . Due to Thm. 1.7.56 it can be solved with an asymptotic computational effort of  $\mathcal{O}(n)$ .

- ③ **Periodic cubic spline interpolation:**  $s'(t_0) = s'(t_n)$  ( $\Rightarrow c_0 = c_n$ ),  $s''(t_0) = s''(t_n)$

This removes one unknown and adds another equations so that we end up with an  $n \times n$ -linear system with s.p.d. ( $\rightarrow$  Def. 1.1.8) system matrix

$$\mathbf{A} := \begin{bmatrix} a_1 & b_1 & 0 & \cdots & 0 & b_0 \\ b_1 & a_2 & b_2 & & & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & & & \ddots & a_{n-1} & b_{n-1} \\ b_0 & 0 & \cdots & 0 & b_{n-1} & a_0 \end{bmatrix}, \quad b_i := \frac{1}{h_{i+1}}, \quad i = 0, 1, \dots, n-1, \\ a_i := \frac{2}{h_i} + \frac{2}{h_{i+1}}, \quad i = 0, 1, \dots, n-1.$$

This linear system can be solved with rank-1-modifications techniques (see § 1.6.104, Lemma 1.6.113) + tridiagonal elimination: asymptotic computational effort  $O(n)$ .

### Remark 3.5.11 (Splines in MATLAB)

MATLAB provides many tools for computing and dealing with splines.

MATLAB-function: `v = spline(t, y, x)`: natural / complete spline interpolation

There is even a “Curve Fitting Toolbox” that provides extended functionality.

### Remark 3.5.12 (Piecewise cubic interpolation schemes)

- \* Piecewise cubic local Lagrange interpolation
  - > Extra degrees of freedom fixed by putting four nodes in one interval
  - yields merely  $C^0$ -interpolant; perfectly local.
- \* Cubic Hermite interpolation
  - > Extra degrees of freedom fixed by reconstruction slopes
  - yields  $C^1$ -interpolant; still local.
- \* Cubic spline interpolation
  - > Extra degrees of freedom fixed by  $C^2$ -smoothness, complete/natural/periodic constraint.
  - yields  $C^2$ -interpolant; non-local.

## 3.5.2 Structural properties of cubic spline interpolants

(3.5.13) Extremal properties of natural cubic spline interpolants → [63, Sect. 8.6.1, Property 8.2]

For a function  $f : [a, b] \mapsto \mathbb{R}$ ,  $f \in C^2([a, b])$ , the term

$$E_{\text{bd}}(f) := \frac{1}{2} \int_a^b |f''(t)|^2 dt ,$$

models the elastic **bending energy** of a rod, whose shape is described by the graph of  $f$  (Soundness check: zero bending energy for straight rod). We will show that cubic spline interpolants have minimal bending energy among all  $C^2$ -smooth interpolating functions.

Given: mesh  $\mathcal{M} := \{a = t_0 < t_1 < \dots < t_n = b\}$  of  $[a, b]$  with knots  $t_j$

Set  $s \in \mathcal{S}_{3, \mathcal{M}}$  := natural cubic spline interpolant of data points  $(t_i, y_i) \in \mathbb{R}^2$ ,  $i = 0, \dots, n$ .

### Theorem 3.5.14. Optimality of natural cubic spline interpolant

The natural cubic spline interpolant  $s$  minimizes the elastic curvature energy among all interpolating functions in  $C^2([a, b])$ , that is

$$E_{\text{bd}}(s) \leq E_{\text{bd}}(f) \quad \forall f \in C^2([a, b]), f(t_i) = y_i, i = 0, \dots, n .$$

Idea of proof: **variational calculus**

We show that any small perturbation of  $s$  such that the perturbed spline still satisfies the interpolation conditions leads to an increase in elastic energy.

Pick *perturbation direction*  $k \in C^2([t_0, t_n])$  satisfying  $k(t_i) = 0$ ,  $i = 0, \dots, n$ :

$$\begin{aligned} E_{\text{bend}}(s + k) &= \frac{1}{2} \int_a^b |s'' + \lambda k''|^2 dt \\ &= E_{\text{bend}}(s) + \underbrace{\int_a^b s''(t)k''(t) dt}_{:= I} + \underbrace{\frac{1}{2} \int_a^b |k''|^2 dt}_{\geq 0} . \end{aligned} \tag{3.5.15}$$

Scrutiny of  $I$ : split in interval contributions, integrate by parts twice, and use  $s^{(4)} \equiv 0$ :

$$\begin{aligned} I &= \sum_{j=1}^n \int_{t_{j-1}}^{t_j} s''(t)k''(t) dt \\ &= - \sum_{j=1}^n \left( s'''(t_j^-) \underbrace{k(t_j)}_{=0} - s'''(t_{j-1}^+) \underbrace{k(t_{j-1})}_{=0} \right) + \underbrace{s''(t_n)}_{=0} k'(t_n) - \underbrace{s''(t_0)}_{=0} k'(t_0) = 0 . \end{aligned}$$

In light of (3.5.15): non perturbation compatible with interpolation conditions can make the bending energy of  $s$  decrease!

### Remark 3.5.16 (Origin of the term “Spline”)

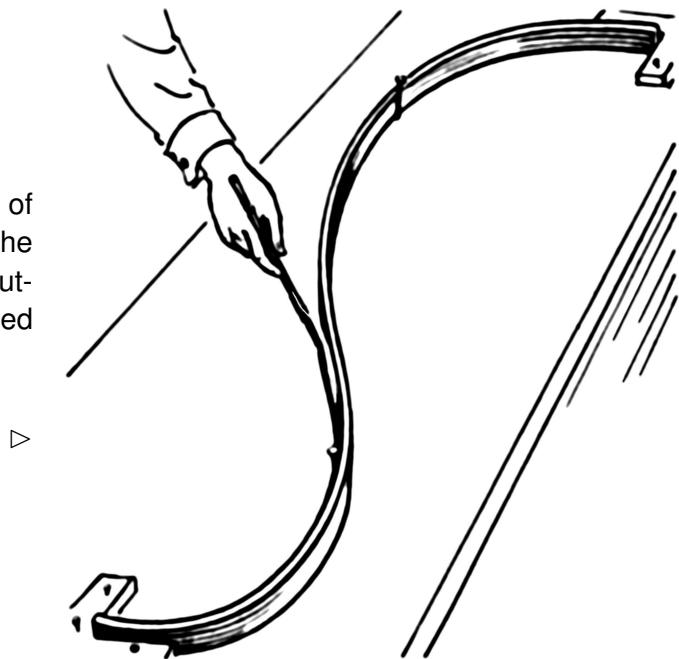
§ 3.5.13: (Natural) cubic spline interpolant provides  $C^2$ -curve of minimal elastic bending energy that travels through prescribed points.

↔

Nature: A thin elastic rod fixed at certain points attains a shape that minimizes its potential bending energy (virtual work principle of statics).

- Cubic spline interpolation approximates shape of elastic rods. Such rods were in fact used in the manufacturing of ship hulls as “analog computers” for “interpolating points” that were specified by the designer of the ship.

Cubic spline interpolation  
before MATLAB



### Remark 3.5.17 (Shape preservation of cubic spline interpolation)

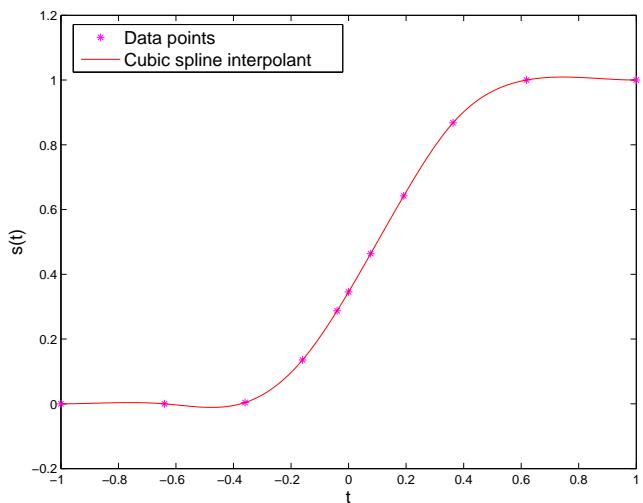
Data  $s(t_j) = y_j$  from Exp. 3.3.7 and

$$c_0 := \frac{y_1 - y_0}{t_1 - t_0},$$

$$c_n := \frac{y_n - y_{n-1}}{t_n - t_{n-1}}.$$

The cubic spline interpolant is **nor** monotonicity **nor** curvature preserving

This is not surprising in light of Thm. 3.4.17, because cubic spline interpolation is a linear interpolation scheme.



### (3.5.18) Weak locality of an interpolation scheme

In terms of locality of interpolation schemes, in the sense of § 3.4.7, we have seen:

- Piecewise linear interpolation (→ Section 3.3.2) is strictly local: Changing a single data value  $y_j$  affects the interpolant only on the interval  $[t_{j-1}, t_{j+1}]$ .
- Monotonicity preserving piecewise cubic Hermite interpolation (→ Section 3.4.2) is still local, because changing  $y_j$  will lead to a change in the interpolant only in  $[t_{j-2}, t_{j+2}]$  (the remote intervals are affected through the averaging of local slopes).
- Polynomial Lagrange interpolation is highly non-local, see Ex. 3.2.59.

We can weaken the notion of **locality** of an interpolation scheme on an ordered node set  $\{t_i\}_{i=0}^n$ :

- (weak) locality measures the impact of a perturbation of a data value  $y_j$  at points  $t \in [t_0, t_n]$  as a function of  $|t - t_j|$ .
- an interpolation scheme is **weakly local**, if the impact of the perturbation of  $y_i$  displays a rapid (e.g. exponential) decay as  $|t - t_i|$  increases.

For a **linear** interpolation scheme ( $\rightarrow$  § 3.1.11) locality can be deduced from the decay of the **cardinal interpolants/cardinal basis functions** ( $\rightarrow$  Lagrange polynomials of § 3.2.10), that is, the functions  $b_j := I(\mathbf{e}_j)$ , where  $\mathbf{e}_j$  is the  $j$ -th unit vector, and  $I$  the interpolation operator. Then weak locality can be quantified as

$$\exists \lambda > 0: |b_j(t)| \leq \exp(-\lambda|t_j|), \quad t \in [t_0, t_n]. \quad (3.5.19)$$

Remember:

- Lagrange polynomials satisfying (3.2.11) provide cardinal interpolants for polynomial interpolation  $\rightarrow$  § 3.2.10. As is clear from Fig. 87, they do not display any decay away from their “base node”. Rather, they grow strongly. Hence, there is no locality in global polynomial interpolation.
- Tent functions ( $\rightarrow$  Fig. 86) are the cardinal basis functions for piecewise linear interpolation, see Ex. 3.1.8. Hence, this scheme is perfectly local, see (3.3.9).

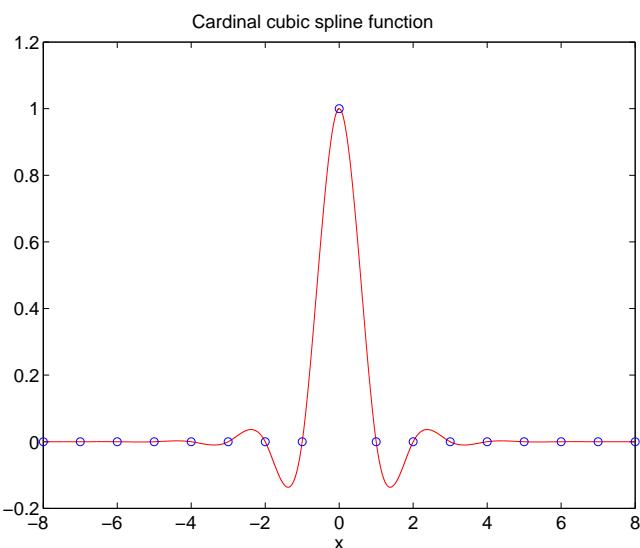
### Experiment 3.5.20 (Weak locality of the natural cubic spline interpolation)

Given a grid  $\mathcal{M} := \{t_0 < t_1 < \dots < t_n\}$  the  $i$ th natural **cardinal spline** is defined as

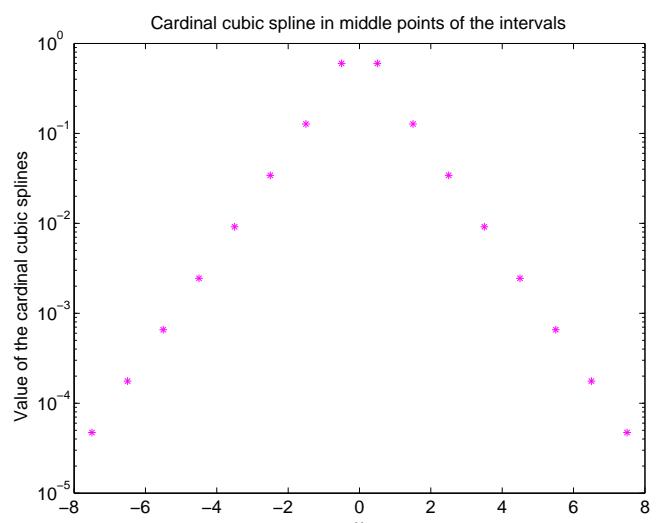
$$L_i \in \mathcal{S}_{3,\mathcal{M}}, \quad L_i(t_j) = \delta_{ij}, \quad L_i''(t_0) = L_i''(t_n) = 0. \quad (3.5.21)$$

► Natural spline interpolant:  $s(t) = \sum_{j=0}^n y_j L_j(t)$ .

Decay of  $L_i \leftrightarrow$  **weak locality** of natural cubic spline interpolation.



Exponential decay of the cardinal splines  $\rightarrow$  cubic spline interpolation is weakly local.



### 3.5.3 Shape Preserving Spline Interpolation

According to Rem. 3.5.17 is cubic spline interpolation neither monotonicity preserving nor curvature preserving. Necessarily so, because it is a *linear* interpolation scheme, see Thm. 3.4.17.

This section presents a non-linear *quadratic spline* ( $\rightarrow$  Def. 3.5.1,  $C^1$ -functions) based interpolation scheme that manages to preserve both monotonicity and curvature of data even in a local sense, cf. Section 3.3.

Given: data points  $(t_i, y_i) \in \mathbb{R}^2, i = 0, \dots, n$ , assume ordering  $t_0 < t_1 < \dots < t_n$ .

Sought:

- \* extended knot set  $\mathcal{M} \subset [t_0, t_n]$  ( $\rightarrow$  Def. 3.5.1),
- \* an interpolating quadratic spline function  $s \in \mathcal{S}_{2, \mathcal{M}}$ ,  $s(t_i) = y_i, i = 0, \dots, n$   
that preserves the “shape” of the data in the sense of § 3.3.2.

Notice that here  $\mathcal{M} \neq \{t_j\}_{j=0}^n$ :  $s$  interpolates the data in the points  $t_i$  but is piecewise polynomial with respect to  $\mathcal{M}$ ! The interpolation nodes will usually not belong to  $\mathcal{M}$ .

We proceed in four steps:

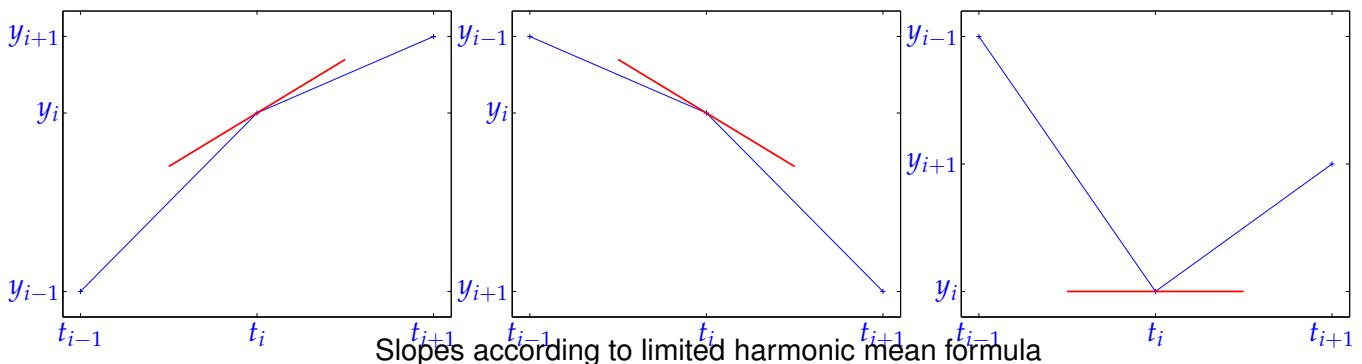
① Shape preserving choice of slopes  $c_i, i = 0, \dots, n$  [53, 62], analogous to Section 3.4.2

Recall Eq. (3.4.12) and Eq. (3.4.14): we fix the slopes  $c_i$  in the nodes using the harmonic mean of data slopes  $\Delta_j$ , the final interpolant will be tangent to these segments in the points  $(t_i, y_i)$ . If  $(t_i, y_i)$  is a local maximum or minimum of the data,  $c_i$  is set to zero ( $\rightarrow$  § 3.4.11)

$$\text{Limiter } c_i := \begin{cases} \frac{2}{\Delta_i^{-1} + \Delta_{i+1}^{-1}} & , \text{ if } \text{sign}(\Delta_i) = \text{sign}(\Delta_{i+1}), \\ 0 & \text{otherwise,} \end{cases} \quad i = 1, \dots, n-1 .$$

$$c_0 := 2\Delta_1 - c_1, \quad c_n := 2\Delta_n - c_{n-1},$$

where  $\Delta_j = \frac{y_j - y_{j-1}}{t_j - t_{j-1}}$ .



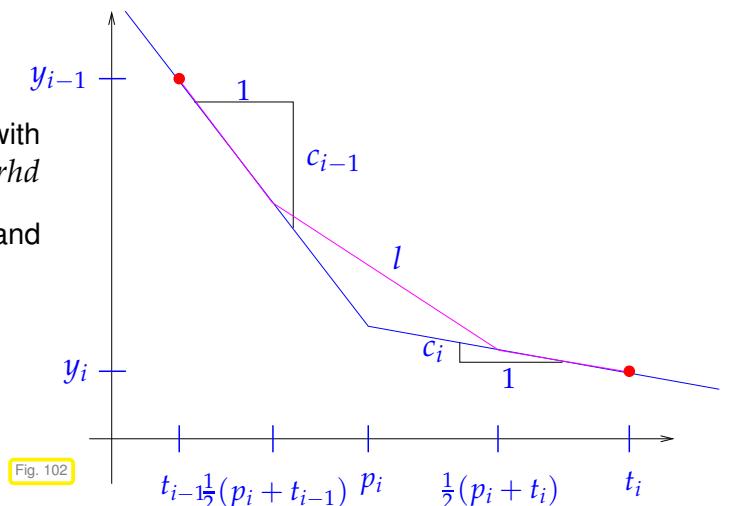
② Choice of “extra knots”  $p_i \in ]t_{i-1}, t_i[, i = 1, \dots, n$ :

Rule:

Let  $T_i$  be the unique straight line through  $(t_i, y_i)$  with slope  $c_i$ ; — in figure

If the intersection of  $T_{i-1}$  and  $T_i$  is non-empty and has a  $t$ -coordinate  $\in [t_{i-1}, t_i]$ ,

- ☞ then  $p_i := t$ -coordinate of  $T_{i-1} \cap T_i$ ,
- ☞ otherwise  $p_i = \frac{1}{2}(t_{i-1} + t_i)$ .



These points will be used to build the knot set for the final **quadratic spline**:

$$\mathcal{M} = \{t_0 < p_1 \leq t_1 < p_2 \leq \dots < p_n \leq t_n\}.$$

#### MATLAB-code 3.5.22: Quadratic spline: selection of $p_i$

```

1 p = (t(1)-1)*ones(1, length(t)-1);
2 for j=1:n-1
3 if (c(j) ~= c(j+1))
4     p(j)=(y(j+1)-y(j)+t(j)*c(j)-t(j+1)*c(j+1))/(c(j)-c(j+1));
5 end
6 if ((p(j)<t(j)) | (p(j)>t(j+1)))
7     p(j) = 0.5*(t(j)+t(j+1));
8 end;
9 end

```

③ Set  $\mathcal{M}' = \{t_0 < \frac{1}{2}(t_0 + p_1) < \frac{1}{2}(p_1 + t_1) < \frac{1}{2}(t_1 + p_2) < \dots < \frac{1}{2}(t_{n-1} + p_n) < \frac{1}{2}(p_n + t_n) < t_n\}$ ,

with  $l(t_i) = y_i$ ,  $l'(t_i) = c_i$ .

- In each interval  $(\frac{1}{2}(p_j + t_j), \frac{1}{2}(t_j + p_{j+1}))$  the spline corresponds to the segment of slope  $c_j$  passing through the data point  $(t_j, y_j)$ .
- In each interval  $(\frac{1}{2}(t_j + p_{j+1}), \frac{1}{2}(p_{j+1} + t_{j+1}))$  the spline corresponds to the segment connecting the previous ones, see Fig. 102.

$l$  “inherits” local monotonicity and curvature from the data.

#### Example 3.5.23 (Auxiliary construction for shape preserving quadratic spline interpolation)

Data points:  $t = (0 : 12); y = \cos(t);$

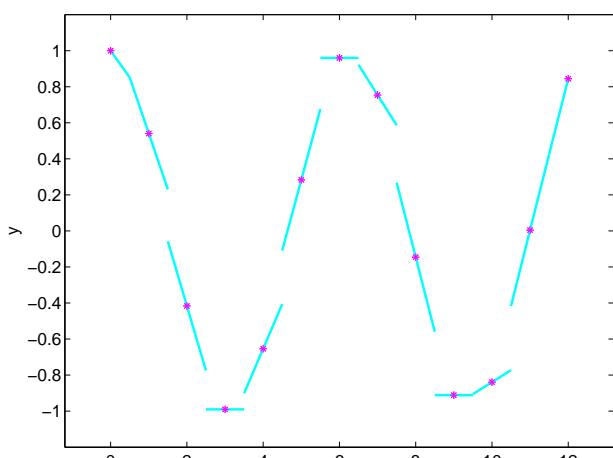


Fig. 103

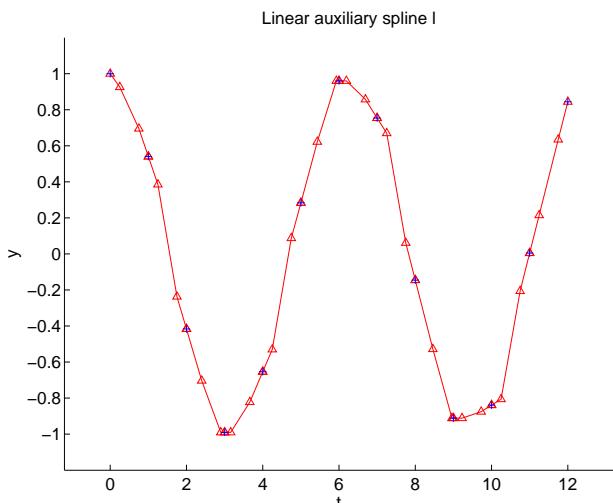
Local slopes  $c_i, i = 0, \dots, n$ 

Fig. 104

Linear auxiliary spline  $l$ 

④ Local quadratic approximation / interpolation of  $l$ :

Tedious, but elementary calculus, confirms the following fact:

**Lemma 3.5.24.**

If  $g$  is a linear spline through the three points

$$(a, y_a), \left(\frac{1}{2}(a+b), w\right), (b, y_b) \quad \text{with } a < b, \quad y_a, y_b, w \in \mathbb{R},$$

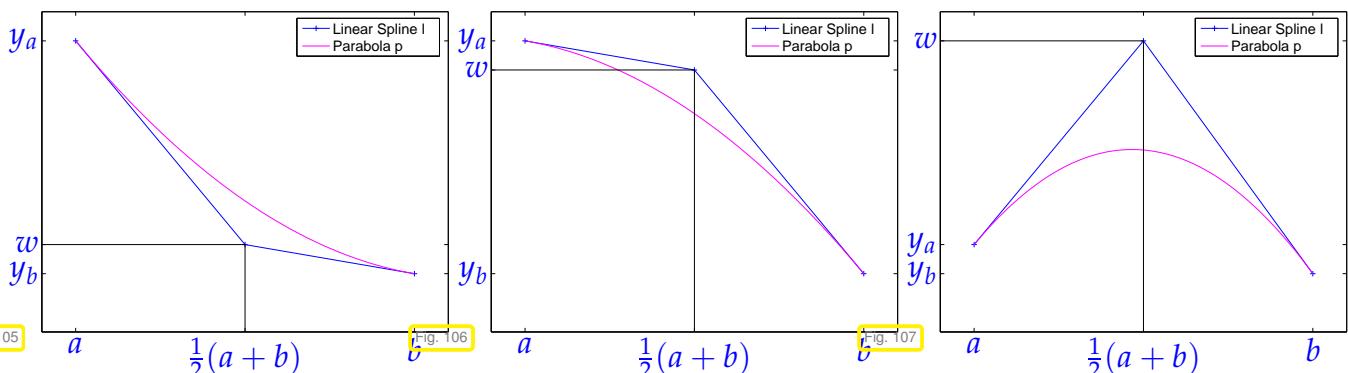
then the parabola

$$p(t) := (y_a(b-t)^2 + 2w(t-a)(b-t) + y_b(t-a)^2)/(b-a)^2, \quad a \leq t \leq b,$$

satisfies

1.  $p(a) = y_a, \quad p(b) = y_b, \quad p'(a) = g'(a), \quad p'(b) = g'(b),$
2.  $g$  monotonic increasing / decreasing  $\Rightarrow p$  monotonic increasing / decreasing,
3.  $g$  convex / concave  $\Rightarrow p$  convex / concave.

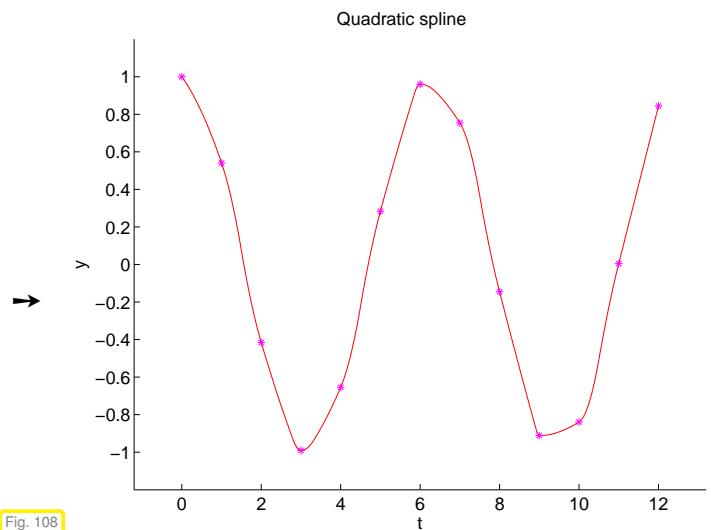
The proof boils down to discussing many cases as indicated in the following plots:



Lemma 3.5.24 implies that the final quadratic spline that passes through the points  $(t_j, y_j)$  with slopes  $c_j$  can be built locally as the parabola  $p$  using the linear spline  $l$  that plays the role of  $g$  in the lemma.

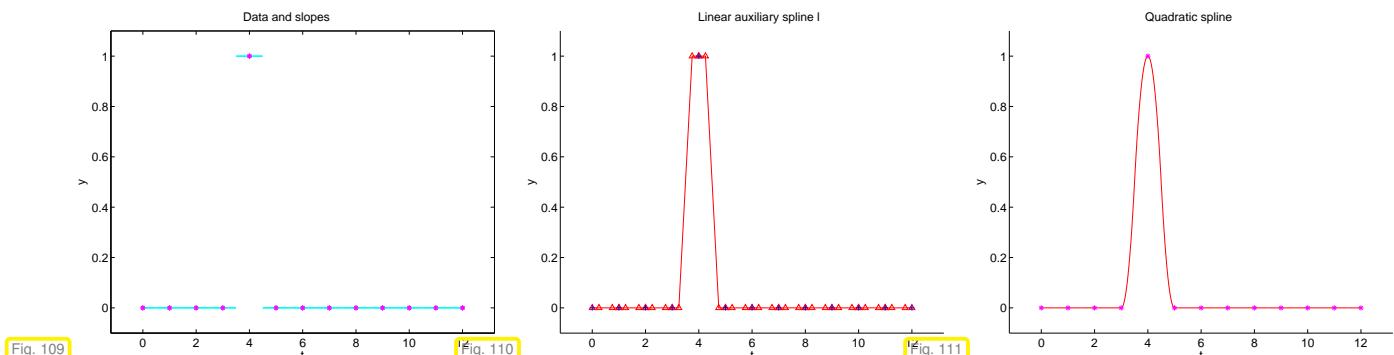
Continuation of Ex. 3.5.23:

Interpolating quadratic spline



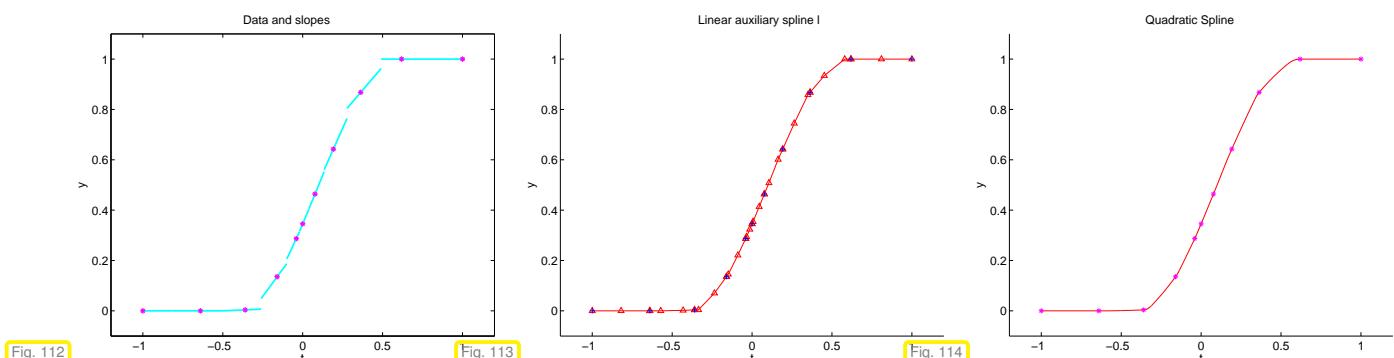
### Example 3.5.25 (Cardinal shape preserving quadratic spline)

We examine the shape preserving quadratic spline that interpolates data values  $y_j = 0, j \neq i, y_i, i \in \{0, \dots, n\}$ , on an equidistant node set.

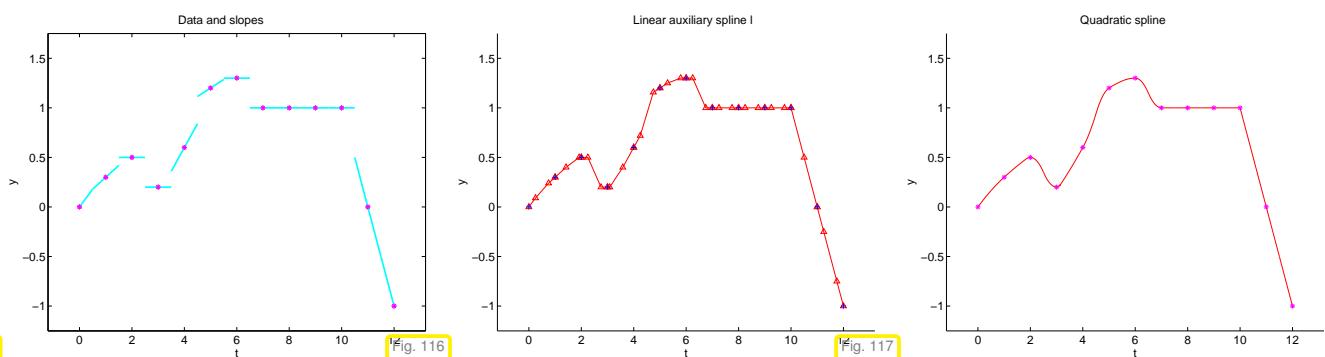


### Example 3.5.26 (Shape preserving quadratic spline interpolation)

Data from Exp. 3.3.7:



Data from [53]:	$t_i$	0	1	2	3	4	5	6	7	8	9	10	11	12
	$y_i$	0	0.3	0.5	0.2	0.6	1.2	1	1	1	1	0	-1	



In all cases we observe excellent local shape preservation.

### MATLAB-code 3.5.27: Step by step shape preserving spline interpolation

```

1 % 22.06.2009 shapepresintp.m
2 % Shape preserving interpolation through nodes (t,y)
3 % Build a quadratic spline interpolation without overshooting
4 % In 4 steps, see the comments in each step
5 %
6 % example:
7 % shapepresintp([1, 1.5, 3:10], [ 1 2.5 4 3.8 3 2 1 4 2 1])
8 % shapepresintp([0:0.05:1], cosf([0:0.05:1]))
9 % shapepresintp([0:0.05:1], [zeros(1,5),1,zeros(1,15) ] )
10 % shapepresintp([0:12], [0 0.3 0.5 0.2 0.6 1.2 1.3 1 1 1 0 -1])

11
12 function shapepresintp(t,y)

13
14 n=length(t)-1;
15 % plot the data and prepare the figure
16 figure;
17 hplot(1)=plot(t,y,'k*-');
18 hold on;
19 newaxis=axis;
20 newaxis([3,4])=[newaxis(3)-0.5, newaxis(4)+0.5];
21 axis(newaxis); % enlarge the vertical size of the plot
22 title('Data points - Press enter to continue')
23 plot(t,ones(1,n+1)* (newaxis(3)+0.25),'k.');
24 set(gca, 'XTick', t)
25 leg={'Data','Slopes','Middle points','Linear spline','Sh. pres. spline'};
26 legend(hplot(1),leg{1});
27 pause;

28
29 % ===== Step 1: choice of slopes =====
30 % shape-faithful slopes (c) in the nodes using harmonic mean of data slopes
31 % the final interpolant will be tangents to these segments

32
33 disp('STEP 1')
34 title('Shape-faithful slopes - Press enter to continue')
35 h=diff(t);

```

```

36 delta = diff(y) ./ h;                                % slopes of data
37 c=zeros(size(t));
38 for j=1:n-1
39   if (delta(j)*delta(j+1) >0)
40     c(j+1) = 2/(1/delta(j) + 1/delta(j+1));
41   end
42 end
43 c(1)=2*delta(1)-c(2);    c(n+1)=2*delta(n)-c(n);
44
45 % plot segments indicating the slopes c(i):
46 % use (vector) plot handle 'hplot' to reduce the linewidth in step 2
47 hplots=zeros(1,n+1);
48 for j=2:n
49   hplotsl(j)=plot([t(j)-0.3*h(j-1),t(j)+0.3*h(j)],
50                   [y(j)-0.3*h(j-1)*c(j),y(j)+0.3*h(j)*c(j)],'-','linewidth',2);
51 end
52 hplotsl(1)=plot([t(1),t(1)+0.3*h(1)], [y(1),y(1)+0.3*h(1)*c(1)],
53                 '-','linewidth',2);
54 hplotsl(n+1)= plot([t(end)-0.3*h(end),t(end)],
55                      [y(end)-0.3*h(end)*c(end),y(end)], '-','linewidth',2);
56 legend([hplot(1), hplotsl(1)], leg{1:2});
57 pause;
58
59 % ===== Step 2: choice of middle points =====
60 % fix points p(j) in [t(j), t(j+1)], depending on the slopes c(j),
61 % c(j+1)
62
63 disp('STEP 2')
64 title('Middle points - Press enter to continue')
65 set(hplotsl,'linewidth',1)
66
67 p = (t(1)-1)*ones(1,length(t)-1);
68 for j=1:n
69   if (c(j) ~= c(j+1))
70     p(j)=(y(j+1)-y(j)+ t(j)*c(j)-t(j+1)*c(j+1)) / (c(j)-c(j+1));
71   end
72   % check and repair if p(j) is outside its interval:
73   if ((p(j)<t(j)) || (p(j)>t(j+1)));    p(j) = 0.5*(t(j)+t(j+1));
74   end;
75 end
76
77 hplot(2)=plot(p,ones(1,n)*(newaxis(3)+0.25),'go');
78 legend([hplot(1), hplotsl(1), hplot(2)], leg{1:3});
79 pause;
80
81 % ===== Step 3: auxiliary linear spline =====
82 % build the linear spline with nodes in:
83 % -t(j)
84 % -the middle points between t(j) and p(j)
85 % -the middle points between p(j) and t(j+1)

```

```

81 % -t(j+1)
82 % and with slopes c(j) in t(j), for every j
83
84 disp('STEP 3')
85 title('Auxiliary linear spline - Press enter to continue')
86
87 for j=1:n
88 hplot(3)=plot([t(j) 0.5*(p(j)+t(j)) 0.5*(p(j)+t(j+1)) t(j+1)],
89 [y(j) y(j)+0.5*(p(j)-t(j))*c(j)
90 y(j+1)+0.5*(p(j)-t(j+1))*c(j+1) y(j+1)], 'm^-');
91 plot([t(j) 0.5*(p(j)+t(j)) 0.5*(p(j)+t(j+1)) t(j+1)],
92 ones(1,4)*(newaxis(3)+0.25), 'm^');
93 end
94 legend([hplot(1), hplotsl(1), hplot(2), hplot(3)], leg{1:4});
95 pause;
96
97 % ===== Step 4: quadratic spline =====
98 % final quadratic shape preserving spline
99 % quadratic polynomial in the intervals [t(j), p(j)] and [p(j), t(j)]
100 % tangent in t(j) and p(j) to the linear spline of step 3
101
102 disp('STEP 4')
103 title('Quadratic spline')
104
105 % for every interval 2 quadratic interpolations
106 % a, b, ya, yb = extremes and values in the subinterval
107 % w = value in middle point that gives the right slope
108 for j=1:n
109 a=t(j);
110 b=p(j);
111 ya=y(j);
112 w=y(j)+0.5*(p(j)-t(j))*c(j);
113 yb=((t(j+1)-p(j))*(y(j)+0.5*(p(j)-t(j))*c(j))+...
114 (p(j)-t(j))*(y(j+1)+0.5*(p(j)-t(j+1))*c(j+1)))/(t(j+1)-t(j));
115 x=linspace(a,b,100);
116 pb=(ya*(b-x).^2+2*w*(x-a).* (b-x)+yb*(x-a).^2)/((b-a)^2);
117 hplot(4)=plot(x,pb,'r-', 'linewidth',2);
118
119 a=b;
120 b=t(j+1);
121 ya=yb;
122 yb=y(j+1);
123 w=y(j+1)+0.5*(p(j)-t(j+1))*c(j+1);
124 x=(a:(b-a)/100:b);
125 pb=(ya*(b-x).^2+2*w*(x-a).* (b-x)+yb*(x-a).^2)/((b-a)^2);
126 plot(x,pb,'r-', 'linewidth',2);
127
128 plot(p(j),ya,'go');
129 end

```

```
128 % replot initial nodes over the other plots:  
129 plot(t,y,'k*');  
130 % plot(p,yb,'go')  
131 legend([hplot(1), hplotsl(1), hplot(2:4)], leg);  
132 title('Shape preserving interpolation')
```

## Learning outcomes

After you have studied this chapter you should

- understand the use of basis functions for representing functions on a computer,
- know the concept of a interpolation operator and what its linearity means,
- know the connection between linear interpolation operators and linear systems of equations,
- be familiar with efficient algorithms for polynomial interpolation in different settings,
- know the meaning and significance of “sensitivity” in the context of interpolation,
- Be familiar with the notions of “shape preservation” for an interpolation scheme and its different aspects (monotonicity, curvature),
- know the details of cubic Hermite interpolation and how to ensure that it is monotonicity preserving.
- know what splines are and how cubic spline interpolation with different endpoint constraints works.

# Chapter 4

## Approximation of Functions in 1D

### (4.0.1) General approximation problem

#### Approximation of functions: Generic view

Given: function  $\mathbf{f} : D \subset \mathbb{R}^n \mapsto \mathbb{R}^d$  (often in procedural form `y=feval(x)`, Rem. 3.1.4)

Goal: Find a “simple”<sup>(\*)</sup> function  $\tilde{\mathbf{f}} : D \mapsto \mathbb{R}^d$  such that the **approximation error**  $\mathbf{f} - \tilde{\mathbf{f}}$  is “small”<sup>(♣)</sup>

(\*): What is “simple” ?

The function  $\tilde{\mathbf{f}}$  can be encoded by small amount of information and is easy to evaluate.

For instance, this is the case for polynomial or piecewise polynomial  $\tilde{\mathbf{f}}$ .

(♣): What does “small approximation error” mean ?

$\|\mathbf{f} - \tilde{\mathbf{f}}\|$  is small for some norm  $\|\cdot\|$  on the space  $C^0(\overline{D})$  of (piecewise) continuous functions.

The most commonly used norms are

- \* the **supremum norm**  $\|\mathbf{g}\|_{\infty} := \|\mathbf{g}\|_{L^{\infty}(D)} := \max_{x \in D} |\mathbf{g}(x)|$ , see (3.2.62).

If the approximation error is small with respect to the supremum norm,  $\tilde{\mathbf{f}}$  is also called a good **uniform** approximant of  $\mathbf{f}$ .

- \* the  **$L^2$ -norm**  $\|\mathbf{g}\|_2^2 := \|\mathbf{g}\|_{L^2(D)}^2 = \int_D |\mathbf{g}(x)|^2 dx$ , see (3.2.63).

Below we consider only the case  $n = d = 1$ : approximation of scalar valued functions defined on an interval. The techniques can be applied componentwise in order to cope with the case of vector valued function ( $d > 1$ ).

#### Remark 4.0.3 (Model reduction by interpolation)

The non-linear circuit sketched beside has to ports.

▷

For the sake of circuit simulation it should be replaced by a non-linear lumped circuit element characterized by a single voltage-current constitutive relationship  $I = I(U)$ . For any value of the voltage  $U$  the current  $I$  can be computed by solving a non-linear system of equations, see Ex. 2.0.1.

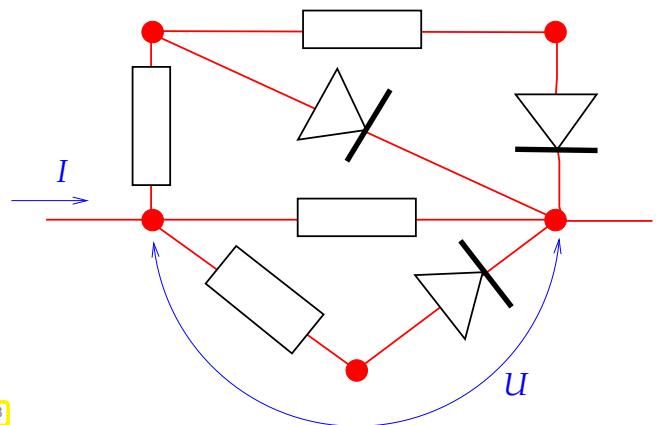


Fig. 118

A faster alternative is the advance approximation of the function  $U \mapsto I(U)$  based on a few computed values  $I(U_i)$ ,  $i = 0, \dots, n$ , followed by the fast evaluation of the approximant  $U \mapsto \tilde{I}(U)$  during actual circuit simulations. This is an example of **model reduction** by approximation of functions.

#### (4.0.4) Approximation schemes

We define an abstract concept for the sake of clarity: When in this chapter we talk about an “**approximation scheme**” (in 1D) we refer to a mapping  $A : X \mapsto V$ , where  $X$  and  $V$  are spaces of functions  $I \mapsto \mathbb{K}$ ,  $I \subset \mathbb{R}$  an interval.

Examples are

- $X = C^k(I)$ , the spaces of functions  $I \mapsto \mathbb{K}$  that are  $k$  times continuously differentiable,  $k \in \mathbb{N}$ .
- $V = \mathcal{P}_m(I)$ , the space of polynomials of degree  $\leq k$ , see Section 3.2.1
- $V = \mathcal{S}_{d,M}$ , the space of splines of degree  $d$  on the knot set  $M \subset I$ , see Def. 3.5.1.

#### (4.0.5) Approximation by interpolation

In Chapter 3 we discussed ways to construct functions whose graph runs through given data points, see 3.1. We can hope that the interpolant will approximate the function, if the data points are also located on the graph of that function. Thus every interpolation scheme, see § 3.1.2, spawns a corresponding approximation scheme.

**Interpolation scheme + sampling → approximation scheme**

$$f : I \subset \mathbb{R} \rightarrow \mathbb{K} \xrightarrow{\text{sampling}} (t_i, y_i := f(t_i))_{i=0}^m \xrightarrow{\text{interpolation}} \tilde{f} := I_T y \quad (\tilde{f}(t_i) = y_i).$$

↑  
free choice of nodes  $t_i \in I$

In this chapter we will mainly study approximation by interpolation relying on the interpolation schemes (→ § 3.1.2) introduced in Section 3.2, Section 3.4, and Section 3.5.

There is *additional freedom* compared to data interpolation: we can choose the interpolation nodes in a smart way in order to obtain an accurate interpolant  $\tilde{f}$ .

**Remark 4.0.7 (Interpolation and approximation: enabling technologies)**

Approximation and interpolation ( $\rightarrow$  Chapter 3) are key components of many numerical methods, like for integration, differentiation and computation of the solutions of differential equations, as well as for computer graphics and generation of smooth curves and surfaces.

► This chapter is a “foundations” part of the course

## Contents

<b>4.1 Approximation by Global Polynomials . . . . .</b>	<b>299</b>
4.1.1 Polynomial approximation: Theory . . . . .	300
4.1.2 Error estimates for polynomial interpolation . . . . .	304
4.1.3 Chebychev Interpolation . . . . .	313
4.1.3.1 Motivation and definition . . . . .	313
4.1.3.2 Chebychev interpolation error estimates . . . . .	317
4.1.3.3 Chebychev interpolation: computational aspects . . . . .	323
<b>4.2 Mean Square Best Approximation . . . . .</b>	<b>327</b>
4.2.1 Abstract theory . . . . .	327
4.2.1.1 Mean square norms . . . . .	327
4.2.1.2 Normal equations . . . . .	328
4.2.1.3 Orthonormal bases . . . . .	329
4.2.2 Polynomial mean square best approximation . . . . .	330
<b>4.3 Uniform Best Approximation . . . . .</b>	<b>336</b>
<b>4.4 Trigonometric interpolation . . . . .</b>	<b>339</b>
<b>4.5 Approximation by piecewise polynomials . . . . .</b>	<b>348</b>
4.5.1 Piecewise polynomial Lagrange interpolation . . . . .	349
4.5.2 Cubic Hermite interpolation: error estimates . . . . .	352
4.5.3 Cubic spline interpolation: error estimates [42, Ch. 47] . . . . .	356
<b>4.6 Multi-dimensional Approximation on Tensor-Product Domains . . . . .</b>	<b>358</b>

## 4.1 Approximation by Global Polynomials

The space  $\mathcal{P}_k$  of polynomials of degree  $\leq k$  has been introduced in Section 3.2.1. For reasons listed in § 3.2.3 polynomials are the most important theoretical and practical tool for the approximation of functions. The next example presents an important case of approximation by polynomials.

**Example 4.1.1 (Taylor approximation  $\rightarrow$  [77, Sect. 5.5])**

The local approximation of sufficiently smooth functions by polynomials is a key idea in calculus, which manifests itself in the importance of approximation by **Taylor polynomials**: For  $f \in C^k(I)$ ,  $k \in \mathbb{N}$ ,  $I \subset \mathbb{R}$  an interval, we approximate

$$f(t) \approx \underbrace{\sum_{j=0}^k \frac{f^{(j)}(t_0)}{j!} (t - t_0)^j}_{=:T_k(t)}, \quad \text{for some } t_0 \in I.$$

The Taylor polynomial  $T_k$  of degree  $k$  approximates  $f$  in a neighbourhood  $J \subset I$  of  $t_0$  ( $J$  can be small!). This can be quantified by the **remainder formula** [77, Bem. 5.5.1]

$$f(t) - T_k(t) = f^{(k+1)}(\xi) \frac{(t - t_0)^{k+1}}{(k+1)!}, \quad \xi = \xi(t, t_0) \in ]\min(t, t_0), \max(t, t_0)[, \quad (4.1.2)$$

which shows that for  $f \in C^{k+1}(I)$  the Taylor polynomial  $T_k$  is pointwise close to  $f \in C^{k+1}(I)$ , if the interval  $I$  is small.

Approximation by Taylor polynomials is easy and direct but inefficient: a polynomial of lower degree often gives the same accuracy.

### (4.1.3) Nested approximation spaces of polynomials

Obviously, for every interval  $I \subset \mathbb{R}$ , the spaces of polynomials are **nested** in the following sense:

$$\mathcal{P}_0 \subset \mathcal{P}_1 \subset \cdots \subset \mathcal{P}_m \subset \mathcal{P}_{m+1} \subset \cdots \subset C^\infty(I), \quad (4.1.4)$$

with finite, but increasing dimensions  $\dim \mathcal{P}_m = m + 1$  according to Thm. 3.2.2.

With this family of nested spaces of polynomials at our disposal, it is natural to study *associated families of approximation schemes*, one for each degree, mapping into  $\mathcal{P}_m$ ,  $m \in \mathbb{N}_0$ .

## 4.1.1 Polynomial approximation: Theory

### (4.1.5) Scope of polynomial approximation

Sloppily speaking, according to (4.1.2) the Taylor polynomials from Ex. 4.1.1 provide **uniform** ( $\rightarrow$  § 4.0.1) approximation of a *smooth* function  $f$  in (small) intervals, provided that its derivatives do not blow up “too fast” (We do not want to make this precise here).

The question is, whether polynomials still offer uniform approximation on arbitrary bounded closed intervals and for functions that are merely continuous, but not any smoother. The answer is YES and this profound result is known as the **Weierstrass Approximation Theorem**. Here we give an extended version with a concrete formula due to Bernstein, see [16, Section 6.2].

#### Theorem 4.1.6. Uniform approximation by polynomials

For  $f \in C^0([0, 1])$ , define the ***n*-th Bernstein approximant** as

$$p_n(t) = \sum_{j=0}^n f(j/n) \binom{n}{j} t^j (1-t)^{n-j}, \quad p_n \in \mathcal{P}_n. \quad (4.1.7)$$

It satisfies  $\|f - p_n\|_\infty \rightarrow 0$  for  $n \rightarrow \infty$ . If  $f \in C^m([0, 1])$ , then even  $\|f^{(k)} - p_n^{(k)}\|_\infty \rightarrow 0$  for  $n \rightarrow \infty$  and all  $0 \leq k \leq m$ .

Notation:  $g^{(k)} \doteq k\text{-th derivative of a function } g : I \subset \mathbb{R} \rightarrow \mathbb{K}$

### Experiment 4.1.8 (Bernstein approximants)

We compute and plot  $p_n, n = 1, \dots, 25$ , for two functions

$$f_1(t) := \begin{cases} 0 & , \text{ if } |2t - 1| > \frac{1}{2}, \\ \frac{1}{2}(1 + \cos(2\pi(2t - 1))) & \text{else,} \end{cases}$$

$$f_2(t) := \frac{1}{1 + e^{-12(x-1/2)}}.$$

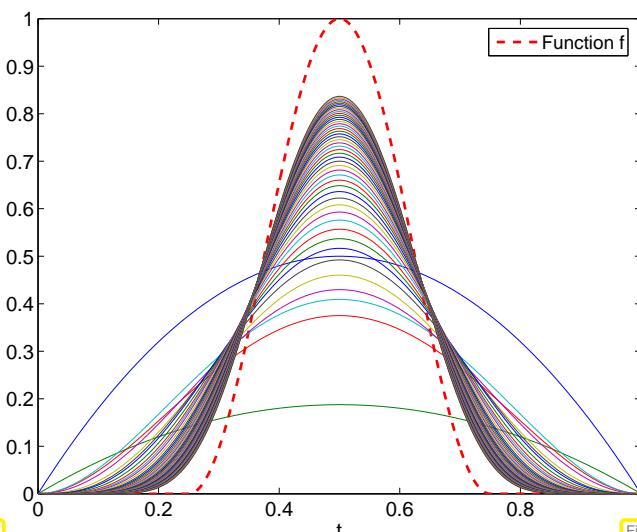


Fig. 119

$$f = f_1$$

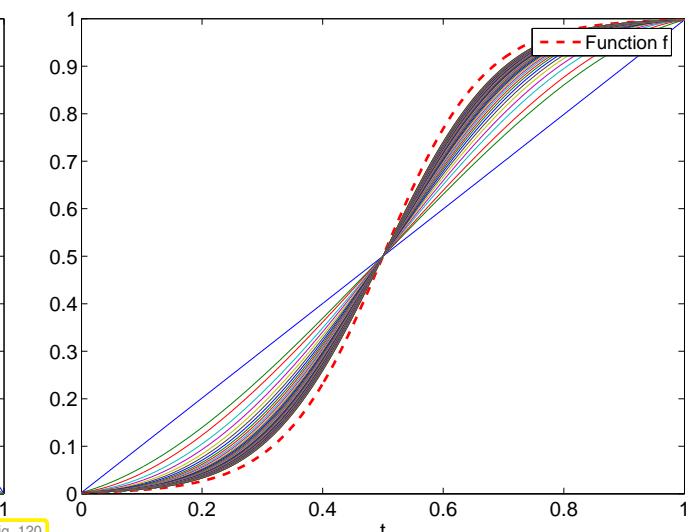


Fig. 120

$$f = f_2$$

We see that the Bernstein approximants “slowly” edge closer and closer to  $f$ . Apparently it takes a very large degree to get really close to  $f$ .

### (4.1.9) Best approximation

Now we introduce a concept needed to gauge how close an approximation scheme gets to the best possible performance.

#### Definition 4.1.10. (Size of) best approximaton error

Let  $\|\cdot\|$  be a (semi-)norm on a space  $X$  of functions  $I \mapsto \mathbb{K}$ ,  $I \subset \mathbb{R}$  an interval. The (size of the) **best approximation error** of  $f \in X$  in the space  $\mathcal{P}_k$  of polynomials of degree  $\leq k$  with respect to  $\|\cdot\|$  is

$$\text{dist}_{\|\cdot\|}(f, \mathcal{P}_k) := \inf_{p \in \mathcal{P}_k} \|f - p\|.$$

For the  $L^2$ -norm  $\|\cdot\|_2$  and the supremum norm  $\|\cdot\|_\infty$  the best approximation error is well defined for  $C = C^0(I)$ .

The polynomial realizing best approximation w.r.t.  $\|\cdot\|$  may neither be unique nor computable with reason-

able effort. Often one is content with rather sharp upper bounds like those asserted in the next theorem, due to Jackson [16, Thm. 13.3.7].

#### Theorem 4.1.11. $L^\infty$ polynomial best approximation estimate

If  $f \in C^r([-1, 1])$  ( $r$  times continuously differentiable),  $r \in \mathbb{N}$ , then, for  $n \geq r$ ,

$$\inf_{p \in \mathcal{P}_n} \|f - p\|_{L^\infty([-1, 1])} \leq (1 + \pi^2/2)^r \frac{(n-r)!}{n!} \|f^{(r)}\|_{L^\infty([-1, 1])}.$$

As above,  $f^{(r)}$  stands for the  $r$ -th derivative of  $f$ . Using Stirling's formula

$$\sqrt{2\pi} n^{n+1/2} e^{-n} \leq n! \leq e n^{n+1/2} e^{-n} \quad \forall n \in \mathbb{N}, \quad (4.1.12)$$

we can get a looser bound of the form

$$\inf_{p \in \mathcal{P}_n} \|f - p\|_{L^\infty([-1, 1])} \leq C(r) n^{-r} \|f^{(r)}\|_{L^\infty([-1, 1])}, \quad (4.1.13)$$

with  $C(r)$  dependent on  $r$ , but *independent of  $f$*  and, in particular, the polynomial degree  $n$ . Using the Landau symbol from Def. 1.4.4 we can rewrite the statement of (4.1.13) in asymptotic form

$$(4.1.13) \Rightarrow \inf_{p \in \mathcal{P}_n} \|f - p\|_{L^\infty([-1, 1])} = O(n^{-r}) \quad \text{for } n \rightarrow \infty.$$

#### Remark 4.1.14 (Transformation of polynomial approximation schemes)

What if a polynomial approximation scheme is defined only on a special interval, say  $[-1, 1]$ . Then by the following trick it can be transferred to any interval  $[a, b] \subset \mathbb{R}$ .

Assume that an interval  $[a, b] \subset \mathbb{R}$ ,  $a < b$ , and a polynomial approximation scheme  $\widehat{\mathcal{A}} : C^0([-1, 1]) \rightarrow \mathcal{P}_n$  are given. Based on the *affine linear* mapping

$$\Phi : [-1, 1] \rightarrow [a, b], \quad \Phi(\hat{t}) := a + \frac{1}{2}(\hat{t} + 1)(b - a), \quad -1 \leq \hat{t} \leq 1, \quad (4.1.15)$$

we can introduce the *affine pullback* of functions:

$$\Phi^* : C^0([a, b]) \rightarrow C^0([-1, 1]), \quad \Phi^*(f)(\hat{t}) := f(\Phi(\hat{t})), \quad -1 \leq \hat{t} \leq 1. \quad (4.1.16)$$

We add the important observations that affine pullbacks are linear and bijective, they are isomorphisms of the involved vector spaces of functions (what is the inverse?).

#### Lemma 4.1.17. Affine pullbacks and polynomials

If  $\Phi^* : C^0([a, b]) \rightarrow C^0([-1, 1])$  is an affine pullback according to (4.1.15) and (4.1.16), then  $\Phi^* : \mathcal{P}_n \rightarrow \mathcal{P}_n$  is a *bijective* linear mapping for any  $n \in \mathbb{N}_0$ .

*Proof.* This is a consequence of the fact that translations and dilations take polynomials to polynomials of the same degree:

$$\Phi^*\{t \rightarrow t^n\} = \{\hat{t} \rightarrow (a + \frac{1}{2}(\hat{t} + 1)(b - a))^n\} \in \mathcal{P}_n.$$

□

The lemma tells us that the spaces of polynomials of some maximal degree are *invariant under affine pullback*. Thus, we can *define* a polynomial approximation scheme  $\mathbf{A}$  on  $C^0([a, b])$  by

$$\mathbf{A} : C^0([a, b]) \rightarrow \mathcal{P}_n , \quad \boxed{\mathbf{A} := (\Phi^*)^{-1} \circ \widehat{\mathbf{A}} \circ \Phi^*} . \quad (4.1.18)$$

### Remark 4.1.19 (Transforming approximation error estimates)

Thm. 4.1.11 targets only the special interval  $[-1, 1]$ . What does it imply for polynomial best approximation on a general interval  $[a, b]$ ? To answer this question we apply techniques from Rem. 4.1.14, in particular the pullback (4.1.16).

We first have to study the change of norms of functions under the action of affine pullbacks:

#### Lemma 4.1.20. Transformation of norms under affine pullbacks

For every  $f \in C^0([a, b])$  we have

$$\|f\|_{L^\infty([a,b])} = \|\Phi^* f\|_{L^\infty([-1,1])} , \quad \|f\|_{L^2([a,b])} = \sqrt{|b-a|} \|\Phi^* f\|_{L^2([-1,1])} . \quad (4.1.21)$$

*Proof.* The first estimate should be evident, and the second is a consequence of the transformation formula for integrals [77, Satz 6.1.5] and the definition of the  $L^2$ -norm from (3.2.63). □

Thus, for norms of the approximation errors of polynomial approximation schemes defined by affine transformation (4.1.18) we get

$$\begin{aligned} \|f - \mathbf{A}f\|_{L^\infty([a,b])} &= \|\Phi^* f - \widehat{\mathbf{A}}(\Phi^* f)\|_{L^\infty([-1,1])} , \\ \|f - \mathbf{A}f\|_{L^2([a,b])} &= \sqrt{|b-a|} \|\Phi^* f - \widehat{\mathbf{A}}(\Phi^* f)\|_{L^2([-1,1])} , \end{aligned} \quad \forall f \in C^0([a, b]) . \quad (4.1.22)$$

Equipped with approximation error estimates for  $\mathbf{A}$ , we can infer corresponding estimates for  $\widehat{\mathbf{A}}$ .

The bounds for approximation errors often involve norms of derivatives as in Thm. 4.1.11. Hence, it is important to understand the interplay of pullback and differentiation: By the 1D chain rule

$$\frac{d}{dt}(\Phi^* f)(\hat{t}) = \frac{df}{dt}(\Phi(\hat{t})) \frac{d\Phi}{d\hat{t}} = \frac{df}{dt}(\Phi(\hat{t})) \cdot \frac{1}{2}(b-a) ,$$

which implies

$$(\Phi^* f)^{(r)} = \left( \frac{b-a}{2} \right)^r f^{(r)} . \quad (4.1.23)$$

Lemma 4.1.20

$$\|(\Phi^* f)^{(r)}\|_{L^\infty([-1,1])} = \left( \frac{b-a}{2} \right)^r \|f^{(r)}\|_{L^\infty([a,b])} , \quad f \in C^r([a, b]), r \in \mathbb{N}_0 . \quad (4.1.24)$$

### 4.1.2 Error estimates for polynomial interpolation

In Section 3.2.2, Cor. 3.2.15, we introduced the Lagrangian polynomial interpolation operator  $\mathbf{I}_{\mathcal{T}} : \mathbb{K}^{n+1} \rightarrow \mathcal{P}_n$  belonging to a node set  $\mathcal{T} = \{t_j\}_{j=0}^n$ . In the spirit of § 4.0.5 it induces an approximation scheme on  $C^0(I)$ ,  $I \subset \mathbb{R}$  an interval, if  $\mathcal{T} \subset I$ .

#### Definition 4.1.25. Lagrangian (interpolation polynomial) approximation scheme

Given an interval  $I \subset \mathbb{R}$ ,  $n \in \mathbb{N}$ , a node set  $\mathcal{T} = \{t_0, \dots, t_n\} \subset I$ , the **Lagrangian (interpolation polynomial) approximation scheme**  $\mathbf{L}_{\mathcal{T}} : C^0(I) \rightarrow \mathcal{P}_n$  is defined by

$$\mathbf{L}_{\mathcal{T}}(f) := \mathbf{I}_{\mathcal{T}}(\mathbf{y}) \in \mathcal{P}_n \quad \text{with} \quad \mathbf{y} := (f(t_0), \dots, f(t_n))^T \in \mathbb{K}^{n+1}.$$

Our goal in this section will be

to estimate the norm of the **interpolation error**  $\|f - \mathbf{I}_{\mathcal{T}}f\|$  (for relevant norm on  $C(I)$ ).

#### (4.1.26) Families of Lagrangian interpolation polynomial approximation schemes

Already Thm. 4.1.11 considered the size of the best approximation error in  $\mathcal{P}_n$  as a function of the polynomial degree  $n$ . In the same vein, we may study a *family* of Lagrange interpolation schemes  $\{\mathbf{L}_{\mathcal{T}_n}\}_{n \in \mathbb{N}_0}$  on  $I \subset \mathbb{R}$  induced by a *family of node sets*  $\{\mathcal{T}_n\}_{n \in \mathbb{N}_0}$ ,  $\mathcal{T}_n \subset I$ , according to Def. 4.1.25.

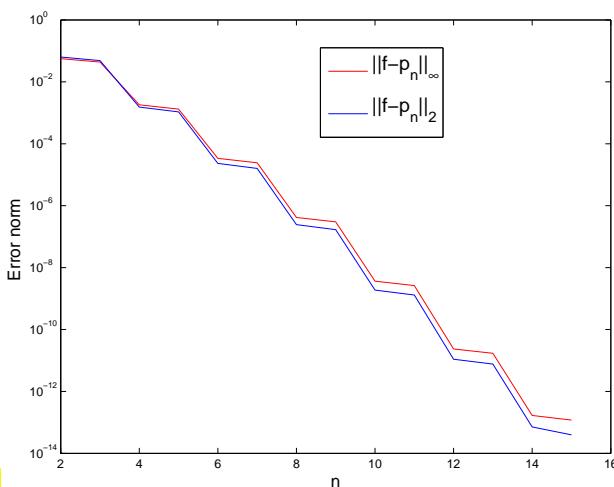
An example for such a family of node sets on  $I := [a, b]$  are the **equidistant** or **equispaced** nodes

$$\mathcal{T}_n := \{t_j^{(n)} := a + (b - a) \frac{j}{n} : j = 0, \dots, n\} \subset I. \quad (4.1.27)$$

For families of Lagrange interpolation schemes  $\{\mathbf{L}_{\mathcal{T}_n}\}_{n \in \mathbb{N}_0}$  we can shift the focus onto estimating the **asymptotic** behavior of the norm of the interpolation error for  $n \rightarrow \infty$ .

#### Experiment 4.1.28 (Asymptotic behavior of Lagrange interpolation error)

We perform polynomial interpolation of  $f(t) = \sin t$  on equispaced nodes in  $I = [0, \pi]$ :  $\mathcal{T}_n = \{j\pi/n\}_{j=0}^n$ . Write  $p$  for the polynomial interpolant:  $p := \mathbf{L}_{\mathcal{T}_n}f \in \mathcal{P}_n$ .



In the numerical experiment the norms of the interpolation errors can be computed only approximately as follows.

- $L^\infty$ -norm: approximated by sampling on a grid of meshsize  $\pi/1000$ .
- $L^2$ -norm: numerical quadrature ( $\rightarrow$  Chapter 5) with trapezoidal rule (5.4.4) on a grid of mesh-size  $\pi/1000$ .

$\triangleleft$  approximate norms  $\|f - \mathbf{L}_{\mathcal{T}_n}f\|_*$ ,  $* = 2, \infty$ .

Fig. 4.1.28

### (4.1.29) Classification of asymptotic behavior of norms of the interpolation error

In the previous experiment we observed a clearly visible regular behavior of  $\|f - L_{T_n}f\|$  as we increased the polynomial degree  $n$ . The prediction of the decay law for  $\|f - L_{T_n}f\|$  for  $n \rightarrow \infty$  is one goal in the study of interpolation errors.

Often this goal can be achieved, even if a rigorous quantitative bound for a norm of the interpolation error remains elusive. In other words, in many cases

no bound for  $\|f - L_{T_n}f\|$  can be given, but its decay for increasing  $n$  can be described precisely.

Now we introduce some important terminology for the qualitative description of the behavior of  $\|f - L_{T_n}f\|$  as a function of the polynomial degree  $n$ . We assume that

$$\exists C \neq C(n) > 0: \|f - L_{T_n}f\| \leq C T(n) \quad \text{for } n \rightarrow \infty. \quad (4.1.30)$$

#### Definition 4.1.31. Types of asymptotic convergence of approximation schemes

Writing  $T(n)$  for the bound of the norm of the interpolation error according to (4.1.30) we distinguish the following *types of asymptotic behavior*:

$$\begin{aligned} \exists p > 0: \quad T(n) \leq n^{-p} & : \text{algebraic convergence, with rate } p > 0, \quad \forall n \in \mathbb{N}. \\ \exists 0 < q < 1: \quad T(n) \leq q^n & : \text{exponential convergence,} \end{aligned}$$

The bounds are assumed to be sharp in the sense, that no bounds with larger rate  $p$  (for algebraic convergence) or smaller  $q$  (for exponential convergence) can be found.

Convergence behavior of norms of the interpolation error is often expressed by means of the Landau-O-notation, *cf.* Def. 1.4.4:

Algebraic convergence:	$\ f - L_T f\  = O(n^{-p})$
Exponential convergence:	$\ f - L_T f\  = O(q^n)$

#### Remark 4.1.32 (Different meanings of “convergence”)

Beware: same concept  $\leftrightarrow$  different meanings:

- **convergence** of a sequence (e.g. of iterates  $x^{(k)}$  → Section 2.1 )
- **convergence** of an approximation (dependent on an approximation parameter, e.g.  $n$ )

#### Remark 4.1.33 (Determining the type of convergence in numerical experiments → § 1.4.8)

Given pairs  $(n_i, \epsilon_i)$ ,  $i = 1, 2, 3, \dots$ ,  $n_i \hat{=}$  polynomial degrees,  $\epsilon_i \hat{=}$  (measured) norms of interpolation errors, how can be tease out the likely type of convergence according to Def. 4.1.31? A similar task was

already encountered in § 1.4.8, where we had to extract information about asymptotic complexity from runtime measurements.

① Conjectured: algebraic convergence:  $\epsilon_i \approx Cn^{-p}$

$$\log(\epsilon_i) \approx \log(C) - p \log n_i \quad (\text{affine linear in log-log scale}).$$

Apply linear regression (MATLAB `polyfit`) to points  $(\log n_i, \log \epsilon_i)$   $\Rightarrow$  estimate for rate  $p$ .

① Conjectured: exponential convergence:  $\epsilon_i \approx C \exp(-\beta n_i)$

$$\log \epsilon_i \approx \log(C) - \beta n_i \quad (\text{affine linear in lin-log scale}).$$

Apply linear regression (Ex. 6.0.9, MATLAB `polyfit`, Ex. 6.0.9) to points  $(n_i, \log \epsilon_i)$   $\Rightarrow$  estimate for  $q := \exp(-\beta)$ .

☞ Fig. 121: we suspect exponential convergence in Exp. 4.1.28.

### Example 4.1.34 (Runge's example → Ex. 3.2.59)

We examine the polynomial interpolant of  $f(t) = \frac{1}{1+t^2}$  for equispaced nodes:

$$\mathcal{T}_n := \left\{ t_j := -5 + \frac{10}{n} j \right\}_{j=0}^n, \quad j = 0, \dots, n \quad \Rightarrow \quad y_j = \frac{1}{1+t_j^2}.$$

We rely on an approximate computation of the supremum norm of the interpolation error by means of sampling as in Exp. 4.1.28.

#### MATLAB-code 4.1.35: Computing the interpolation error for Runge's example

```

1 % interpolation error plot for Runge's example
2 % ("Quick and dirty" MATLAB implementation, see 3.2.3)
3 f = @(x) (1./(1+x.^2));
4 x = -5:0.01:5; % sampling point for approximate maximum norm
5 fv = f(x);
6 err = [];
7 for d=1:20
8     t = -5+(0:d)*10/d;
9     p = polyfit(t,f(t),d);
10    y = polyval(p,x);
11    err = [err; d, max(abs(y-fv))];
12 end
13
14 figure; semilogy(err(:,1),err(:,2),'r--+');
15 xlabel('{\bf degree d}', 'fontsize', 14);
16 ylabel('{\bf interpolation error (maximum norm)}', 'fontsize', 14);
17 print -depsc2 '../PICTURES/rungeerrmax.eps';

```

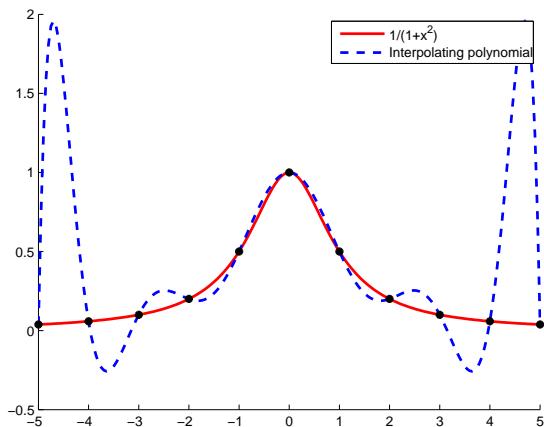


Fig. 122

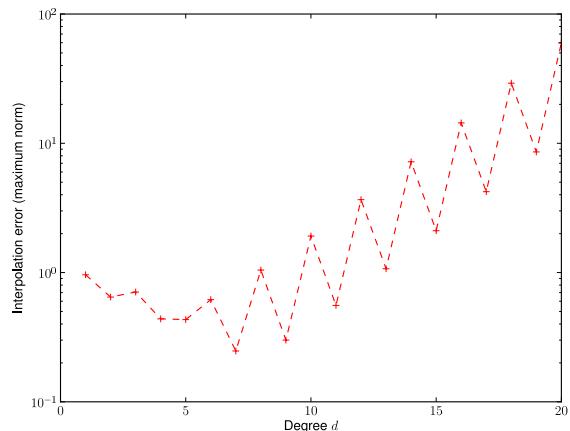
Interpolating polynomial,  $n = 10$ 

Fig. 123

Approximate  $\|f - L_{T_n}f\|_\infty$  on  $[-5, 5]$ Note: approximation of  $\|f - L_{T_n}f\|_\infty$  by *sampling* in 1000 equidistant points.Observation: Strong oscillations of  $L_T f$  near the endpoints of the interval, which seem to cause

$$\|f - L_T f\|_{L^\infty([-5, 5])} \xrightarrow{n \rightarrow \infty} \infty.$$

Though polynomials possess great power to approximate functions, see Thm. 4.1.11 and Thm. 4.1.6, here polynomial interpolants fail completely. Approximation theorists even discovered the following “negative result”:

### Theorem 4.1.36. Divergent polynomial interpolants

Given a sequence of meshes of increasing size  $\{\mathcal{T}_n\}_{n=1}^\infty$ ,  $\mathcal{T}_j = \{t_0^{(n)}, \dots, t_n^{(n)}\} \subset [a, b]$ ,  $a \leq t_0^{(n)} < t_2^{(j)} < \dots < t_n^{(n)} \leq b$ , there exists a continuous function  $f$  such that the sequence of interpolating polynomials  $(L_{\mathcal{T}_n} f)_{n=1}^\infty$  does not converge to  $f$  uniformly as  $n \rightarrow \infty$ .

Now we aim to establish bounds for the supremum norm of the interpolation error of Lagrangian interpolation similar to the result of Thm. 4.1.11.

### Theorem 4.1.37. Representation of interpolation error [15, Thm. 8.22], [42, Thm. 37.4]

We consider  $f \in C^{n+1}(I)$  and the Lagrangian interpolation approximation scheme ( $\rightarrow$  Def. 4.1.25) for a node set  $\mathcal{T} := \{t_0, \dots, t_n\} \subset I$ . Then, for every  $t \in I$  there exists a  $\tau_t \in [\min\{t, t_0, \dots, t_n\}, \max\{t, t_0, \dots, t_n\}]$  such that

$$f(t) - L_{\mathcal{T}}(f)(t) = \frac{f^{(n+1)}(\tau_t)}{(n+1)!} \cdot \prod_{j=0}^n (t - t_j). \quad (4.1.38)$$

*Proof.* Write  $w_{\mathcal{T}}(t) := \prod_{j=0}^n (t - t_j) \in \mathcal{P}_{n+1}$  and fix  $t \in I \setminus \mathcal{T}$ .

$$t \neq t_j \Rightarrow w_{\mathcal{T}}(t) \neq 0 \Rightarrow \exists c = c(t) \in \mathbb{R}: f(t) - L_{\mathcal{T}}(f)(t) = c(t)w_{\mathcal{T}}(t) \quad (4.1.39)$$

Consider the auxiliary function  $\varphi(x) := f(x) - L_{\mathcal{T}}(f)(x) - cw_{\mathcal{T}}(x)$  that has  $n+2$  distinct zeros  $t_0, \dots, t_n, t$ . By iterated application of the mean value theorem [77, Thm. 5.2.1] to higher and higher derivatives, we conclude that

$\varphi^{(m)}$  has  $n + 2 - m$  distinct zeros in  $I$ .

$$\stackrel{m:=n+1}{\Rightarrow} \exists \tau_t \in I: \varphi^{(n+1)}(\tau_t) = f^{(n+1)}(\tau_t) - c(n+1)! = 0.$$

This fixes the value of  $c = \frac{f^{(n+1)}(\tau_t)}{(n+1)!}$  and by (4.1.39) this amounts to the assertion of the theorem.  $\square$

#### Remark 4.1.40 (Explicit representation of error of polynomial interpolation)

The previous theorem can be refined:

##### Lemma 4.1.41. Error of the polynomial interpolation

For  $f \in C^{n+1}(I)$ :  $\forall t \in I$ :

$$\begin{aligned} f(t) - L_T(f)(t) &= \int_0^1 \int_0^{\tau_1} \cdots \int_0^{\tau_{n-1}} \int_0^{\tau_n} f^{(n+1)}(t_0 + \tau_1(t_1 - t_0) + \cdots \\ &\quad + \tau_n(t_n - t_{n-1}) + \tau(t - t_n)) d\tau d\tau_n \cdots d\tau_1 \cdot \prod_{j=0}^n (t - t_j). \end{aligned}$$

*Proof.* By induction on  $n$ , use (3.2.30) and the fundamental theorem of calculus [65, Sect. 3.1]:  $\square$

#### Remark 4.1.42 (Error representation for generalized Lagrangian interpolation)

A result analogous top Lemma 4.1.41 holds also for general polynomial interpolation with multiple nodes as defined in (3.2.21).

Lemma 4.1.41 provides an *exact* formula (4.1.38) for the interpolation error. From it we can derive *estimates* for the supremum norm of the interpolation error on the interval  $I$  as follows:

- ① first bound the right hand side via  $\|f^{(n+1)}\|_{L^\infty(I)}$ ,
- ② then increase the right hand side further by switching to the maximum (in modulus) w.r.t.  $t$  (the resulting bound does no longer depend on  $t!$ ),
- ③ and, finally, take the maximum w.r.t.  $t$  on the left of  $\leq$ .

This yields the following *interpolation error estimate* for degree- $n$  Lagrange interpolation on the node set  $\{t_0, \dots, t_n\}$ :

$$\text{Thm. 4.1.37} \Rightarrow \|f - L_T f\|_{L^\infty(I)} \leq \frac{\|f^{(n+1)}\|_{L^\infty(I)}}{(n+1)!} \max_{t \in I} |(t - t_0) \cdots (t - t_n)|. \quad (4.1.43)$$

**Remark 4.1.44 (Significance of smoothness of interpolant)**

The estimate (4.1.43) hinges on bounds for (higher) derivatives of the interpolant  $f$ , which, essentially, should belong to  $C^{n+1}(I)$ . The same can be said about the estimate of Thm. 4.1.11.

This reflects a general truth about estimates of norms of the interpolation error:

Quantitative interpolation error estimates rely on smoothness!

**Example 4.1.45 (Error of polynomial interpolation   Ex. 4.1.28 cnt'd)**

Now we are in a position to give a theoretical explanation for exponential convergence observed for polynomial interpolation of  $f(t) = \sin(t)$  on equidistant nodes: by Lemma 4.1.41 and (4.1.43)

$$\begin{aligned} \|f^{(k)}\|_{L^\infty(I)} &\leq 1, \quad \Rightarrow \quad \|f - p\|_{L^\infty(I)} \leq \frac{1}{(1+n)!} \max_{t \in I} |(t-0)(t-\frac{\pi}{n})(t-\frac{2\pi}{n}) \cdots (t-\pi)| \\ \forall k \in \mathbb{N}_0 \quad &\leq \frac{1}{n+1} \left(\frac{\pi}{n}\right)^{n+1}. \end{aligned}$$

- Uniform asymptotic (even more than) exponential convergence of the interpolation polynomials (independently of the set of nodes  $\mathcal{T}$ ). In fact,  $\|f - p\|_{L^\infty(I)}$  decays even faster than exponential!

**Example 4.1.46 (Runge's example   Ex. 4.1.34 cnt'd)**

How can the blow-up of the interpolation error observed in Ex. 4.1.34 be reconciled with Lemma 4.1.41 ?

Here  $f(t) = \frac{1}{1+t^2}$  allows only to conclude  $|f^{(n)}(t)| = 2^n n! \cdot O(|t|^{-2-n})$  for  $n \rightarrow \infty$ .

- Possible blow-up of error bound from Thm. 4.1.37  $\rightarrow \infty$  for  $n \rightarrow \infty$ .

**Remark 4.1.47 ( $L^2$ -error estimates for polynomial interpolation)**

Thm. 4.1.37 gives error estimates for the  $L^\infty$ -norm. What about other norms?

From Lemma 4.1.41 using Cauchy-Schwarz inequality:

$$\begin{aligned} \|f - L_{\mathcal{T}}(f)\|_{L^2(I)}^2 &= \int_I \left| \int_0^1 \int_0^{\tau_1} \cdots \int_0^{\tau_{n-1}} \int_0^{\tau_n} f^{(n+1)}(\dots) d\tau d\tau_n \cdots d\tau_1 \cdot \underbrace{\prod_{j=0}^n (t-t_j)}_{|t-t_j| \leq |I|} \right|^2 dt \\ &\leq \int_I |I|^{2n+2} \underbrace{\text{vol}_{(n+1)}(S_{n+1})}_{=1/(n+1)!} \int_{S_{n+1}} |f^{(n+1)}(\dots)|^2 d\tau dt \\ &= \int_I \frac{|I|^{2n+2}}{(n+1)!} \int_I \underbrace{\text{vol}_n(C_{t,\tau})}_{\leq 2^{(n-1)/2}/n!} |f^{(n+1)}(\tau)|^2 d\tau dt, \end{aligned}$$

where

$$\begin{aligned} S_{n+1} &:= \{\mathbf{x} \in \mathbb{R}^{n+1} : 0 \leq x_n \leq x_{n-1} \leq \dots \leq x_1 \leq 1\} \quad (\text{unit simplex}), \\ C_{t,\tau} &:= \{\mathbf{x} \in S_{n+1} : t_0 + x_1(t_1 - t_0) + \dots + x_n(t_n - t_{n-1}) + x_{n+1}(t - t_n) = \tau\}. \end{aligned}$$

This gives the bound for the  $L^2$ -norm of the error:

$$\Rightarrow \|f - L_T(f)\|_{L^2(I)} \leq \frac{2^{(n-1)/4}|I|^{n+1}}{\sqrt{(n+1)!n!}} \left( \int_I |f^{(n+1)}(\tau)|^2 d\tau \right)^{1/2}. \quad (4.1.48)$$

Notice:

$f \mapsto \|f^{(n)}\|_{L^2(I)}$  defines a **seminorm** on  $C^{n+1}(I)$   
(**Sobolev-seminorm**, measure of the smoothness of a function).

Estimates like (4.1.48) play a key role in the analysis of numerical methods for solving partial differential equations (→ course “Numerical methods for partial differential equations”).

#### Remark 4.1.49 (Interpolation error estimates and the Lebesgue constant)

The sensitivity of a polynomial interpolation scheme  $I_T : \mathbb{K}^{n+1} \rightarrow C^0(I)$ ,  $T \subset I$  a node set, as introduced in Section 3.2.4 and expressed by the **Lebesgue constant**  $\lambda_T$  establishes an important connection between the norms of the interpolation error and of the best approximation error.

We first observe that the polynomial approximation scheme  $L_T$  induced by  $I_T$  preserves polynomials of degree  $\leq n$ :

$$L_T p = I_T([p(t)]_{t \in T}) = p \quad \forall p \in \mathcal{P}_n. \quad (4.1.50)$$

Thus, by the triangle inequality, for a generic norm on  $C^0(I)$  and  $\|L_T\|$  designating the associated operator norm of the linear mapping  $L_T$ , cf. (3.2.66),

$$\begin{aligned} \|f - L_T f\| &\stackrel{(4.1.50)}{=} \|(f - p) - L_T(f - p)\| \leq (1 + \|L_T\|) \|f - p\| \quad \forall p \in \mathcal{P}_n, \\ \Rightarrow \|f - L_T f\| &\leq (1 + \|L_T\|) \underbrace{\inf_{p \in \mathcal{P}_n} \|f - p\|}_{\text{best approximation error}}. \end{aligned} \quad (4.1.51)$$

Note that for  $\|\cdot\| = \|\cdot\|_{L^\infty(I)}$ , since  $\|[f(t)]_{t \in T}\|_\infty \leq \|f\|_{L^\infty(I)}$ , we can estimate the operator norm, cf. (3.2.66),

$$\|L_T\|_{L^\infty(I) \rightarrow L^\infty(I)} \leq \|I_T\|_{\mathbb{R}^{n+1} \rightarrow L^\infty(I)} = \lambda_T, \quad (4.1.52)$$

$$\Rightarrow \|f - L_T f\|_{L^\infty(I)} \leq (1 + \lambda_T) \inf_{p \in \mathcal{P}_n} \|f - p\|_{L^\infty(I)} \quad \forall f \in C^0(I). \quad (4.1.53)$$

Hence, if a bound for  $\lambda_T$  is available, the best approximation error estimate of Thm. 4.1.11 immediately yields interpolation error estimates.

#### (4.1.54) Interpolation error estimates for analytic interpolants

Exponential convergence can often be observed for families of Lagrangian approximation schemes, when they are applied to an analytic interpolant.

##### Definition 4.1.55. Analyticity of a complex valued function

let  $D \subset \mathbb{C}$  be an open set in the complex plane. A function  $f : D \rightarrow \mathbb{C}$  is called analytic/holomorphic in  $D$ , if  $f \in C^\infty(D)$  and it possesses a convergent Taylor series at every point  $z \in D$ .

The mathematical area of complex analysis ( $\rightarrow$  course in the BSc program CSE) studies analytic functions. Analyticity gives access to powerful tools provided by complex analysis. One of these tools is the residue theorem.

##### Theorem 4.1.56. Residue theorem [66, Ch. 13]

Let  $D \subset \mathbb{C}$  be an open set,  $G \subset D$  a closed set contained in  $D$ ,  $\gamma := \partial G$  its (piecewise smooth and oriented) boundary, and  $\Pi$  a finite set contained in the interior of  $G$ .

Then for each function  $f$  that is analytic in  $D \setminus \Pi$  holds

$$\frac{1}{2\pi i} \int_{\gamma} f(z) dz = \sum_{p \in \Pi} \text{res}_p f,$$

where  $\text{res}_p f$  is the residual of  $f$  in  $p \in \mathbb{C}$ .

- Note that the integral  $\int_{\gamma}$  in Thm. 4.1.56 is a path integral in the complex plane (“contour integral”): If the path of integration  $\gamma$  is described by a parameterization  $\tau \in J \mapsto \gamma(\tau) \in \mathbb{C}$ ,  $J \subset \mathbb{R}$ , then

$$\int_{\gamma} f(z) dz := \int_J f(\gamma(\tau)) \cdot \dot{\gamma}(\tau) d\tau, \quad (4.1.57)$$

where  $\dot{\gamma}$  designates the derivative of  $\gamma$  with respect to the parameter, and  $\cdot$  indicates multiplication in  $\mathbb{C}$ . For contour integrals we have the estimate

$$\left| \int_{\gamma} f(z) dz \right| \leq |\gamma| \max_{z \in \gamma} |f(z)|. \quad (4.1.58)$$

- $\Pi$  often stands for the set of poles of  $f$ , that is, points where “ $f$  attains the value  $\infty$ ”.

The residue theorem is very useful, because there are simple formulas for  $\text{res}_p f$ :

##### Lemma 4.1.59. Residual formula for quotients

let  $g$  and  $h$  be complex valued functions that are both analytic in a neighborhood of  $p \in \mathbb{C}$ , and satisfy  $h(p) = 0$ ,  $h'(p) \neq 0$ . Then

$$\text{res}_p \frac{g}{h} = \frac{g(p)}{h'(p)}.$$

Now we consider a polynomial Lagrangian approximation scheme on the interval  $I := [a, b] \subset \mathbb{R}$ , based

on the node set  $\mathcal{T} := \{t_0, \dots, t_n\} \subset I$ .

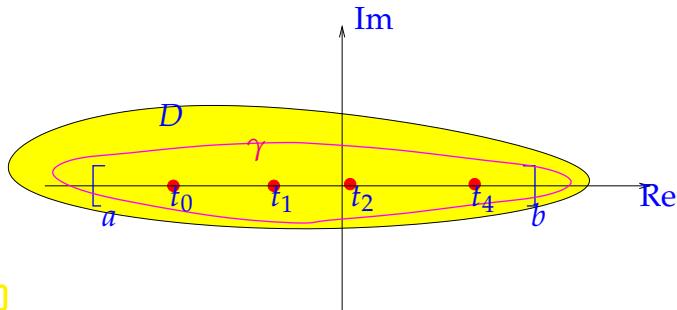


Fig. 124

**Assumption 4.1.60. Analyticity of interpoland**

We assume that the interpoland  $f : I \rightarrow \mathbb{C}$  can be extended to a function  $f : D \subset \mathbb{C} \rightarrow \mathbb{C}$ , which is **analytic** ( $\rightarrow$  Def. 4.1.55) on the open set  $D \subset \mathbb{C}$  with  $[a, b] \subset D$ .

Key is the following representation of the Lagrange polynomials (3.2.11) for node set  $\mathcal{T} = \{t_0, \dots, t_n\}$ :

$$L_j(t) = \prod_{k=0, k \neq j}^n \frac{t - t_k}{t_j - t_k} = \frac{w(t)}{\prod_{k=0, k \neq j}^n (t_j - t_k)} = \frac{w(t)}{(t - t_j)w'(t_j)}, \quad (4.1.61)$$

where  $w(t) = (t - t_0) \cdots (t - t_n) \in \mathcal{P}_{n+1}$ .

Consider the following parameter dependent function  $g_t$ , whose set of poles in  $D$  is  $\Pi = \{t, t_0, \dots, t_n\}$

$$g_t(z) := \frac{f(z)}{(z - t)w(z)}, \quad z \in \mathbb{C} \setminus \Pi, \quad t \in [a, b] \setminus \{t_0, \dots, t_n\}.$$

- $g_t$  is analytic on  $D \setminus \{t, t_0, \dots, t_n\}$  ( $t$  must be regarded as parameter!)
- Apply residue theorem Thm. 4.1.56 to  $g_t$  and a closed path of integration  $\gamma \subset D$  winding once around  $[a, b]$ , such that its interior is simply connected, see the **magenta** curve in Fig. 124:

$$\frac{1}{2\pi i} \int_{\gamma} g_t(z) dz = \text{res}_t g_t + \sum_{j=0}^n \text{res}_{t_j} g_t \stackrel{\text{Lemma 4.1.59}}{=} \frac{f(t)}{w(t)} + \sum_{j=0}^n \frac{f(t_j)}{(t_j - t)w'(t_j)}$$

Possible, because all zeros of  $w$  are single zeros!

$$\Rightarrow f(t) = \underbrace{- \sum_{j=1}^n f(t_j) \underbrace{\frac{w(t)}{(t_j - t)w'(t_j)}}_{\text{Lagrange polynomial!}}}_{\text{polynomial interpolant!}} + \underbrace{\frac{w(t)}{2\pi i} \int_{\gamma} g_t(z) dz}_{\text{interpolation error!}}. \quad (4.1.62)$$

This is another representation formula for the interpolation error, an alternative to that of Thm. 4.1.37 and Lemma 4.1.41. We conclude that for all  $t \in [a, b]$

$$|f(t) - L_{\mathcal{T}} f(t)| \leq \left| \frac{w(t)}{2\pi i} \int_{\gamma} \frac{f(z)}{(z - t)w(z)} dz \right| \leq \frac{|\gamma|}{2\pi} \frac{\max_{a \leq \tau \leq b} |w(\tau)|}{\min_{z \in \gamma} |w(z)|} \cdot \frac{\max_{z \in \gamma} |f(z)|}{\text{dist}([a, b], \gamma)}. \quad (4.1.63)$$

In concrete setting, in order to exploit the estimate (4.1.63) to study the  $n$ -dependence of the supremum norm of the interpolation error, we need to know

- an upper bound for  $|w(t)|$  for  $a \leq t \leq b$ ,

- an a lower bound for  $|w(z)|$ ,  $z \in \gamma$ , for a suitable path of integration  $\gamma \subset D$ ,
- a lower bound for the distance of the path  $\gamma$  and the interval  $[a, b]$  in the complex plane.

**Remark 4.1.64 (Determining the domain of analyticity)**

The subset of  $\mathbb{C}$ , where a function  $f$  given by a formula, is analytic can often be determined without computing derivatives using the following consequence of the chain rule:

**Theorem 4.1.65. Composition of analytic functions**

If  $f : D \subset \mathbb{C} \rightarrow \mathbb{C}$  and  $g : U \subset \mathbb{C} \rightarrow \mathbb{C}$  are analytic in the open sets  $D$  and  $U$ , respectively, then their composition  $f \circ g$  is analytic in

$$\{z \in U : g(z) \in D\}.$$

This can be combined with the following facts:

- Polynomials,  $\exp(z)$ ,  $\sin(z)$ ,  $\cos(z)$ ,  $\sinh(z)$ ,  $\cosh(z)$  are analytic on  $\mathbb{C}$  (entire functions).
- Rational functions (quotients of polynomials) are analytic everywhere except in the zeros of their denominator.
- The square root  $z \rightarrow \sqrt{z}$  is analytic in  $\mathbb{C} \setminus [-\infty, 0]$ .

For example, according to these rules the function  $f(t) = (1 + t^2)^{-1}$  can be extended to an analytic function on  $\mathbb{C} \setminus \{-i, i\}$ . If  $\mathbf{A} \in \mathbb{C}^{n,n}$ ,  $\mathbf{b} \in \mathbb{R}^n$ ,  $n \in \mathbb{N}$ , then  $z \mapsto (\mathbf{A} + z\mathbf{I})^{-1}\mathbf{b} \in \mathbb{C}^n$  is (componentwise) analytic in  $\mathbb{C} \setminus \sigma(\mathbf{A})$ , where  $\sigma(\mathbf{A})$  denotes the set of eigenvalues of  $\mathbf{A}$  (the spectrum).

### 4.1.3 Chebychev Interpolation

As pointed out in § 4.0.5, when we build approximation schemes from interpolation schemes, we have the extra freedom to choose the sampling points (= interpolation nodes). Now, based on the insight into the structure of the interpolation error gained from Thm. 4.1.37, we seek to choose “optimal” sampling points. They will give rise to the so-called Chebychev polynomial approximation schemes, also known as Chebychev interpolation.

#### 4.1.3.1 Motivation and definition

Setting:

- Without loss of generality ( $\rightarrow$  Rem. 4.1.14):  $I = [-1, 1]$ ,
- interpoland  $f : I \rightarrow \mathbb{R}$  at least continuous,  $f \in C^0(I)$ ,
- set of interpolation nodes  $\mathcal{T} := \{-1 \leq t_0 < t_1 < \dots < t_{n-1} < t_n \leq 1\}$ ,  $n \in \mathbb{N}$ .

Recall Thm. 4.1.37:

$$\|f - L_T f\|_{L^\infty(I)} \leq \frac{1}{(n+1)!} \|f^{(n+1)}\|_{L^\infty(I)} \|w\|_{L^\infty(I)},$$

$$\text{with } w(t) := (t - t_0) \cdots (t - t_n).$$

### Optimal choice of interpolation nodes independent of interpoland



Idea: choose nodes  $t_0, \dots, t_n$  such that  $\|w\|_{L^\infty(I)}$  is minimal!

This is equivalent to finding a polynomial  $q \in \mathcal{P}_{n+1}$

- \* with leading coefficient = 1,
- \* such that it minimizes the norm  $\|q\|_{L^\infty(I)}$ .

Then choose nodes  $t_0, \dots, t_n$  as zeros of  $q$  (caution:  $t_j$  must belong to  $I$ ).

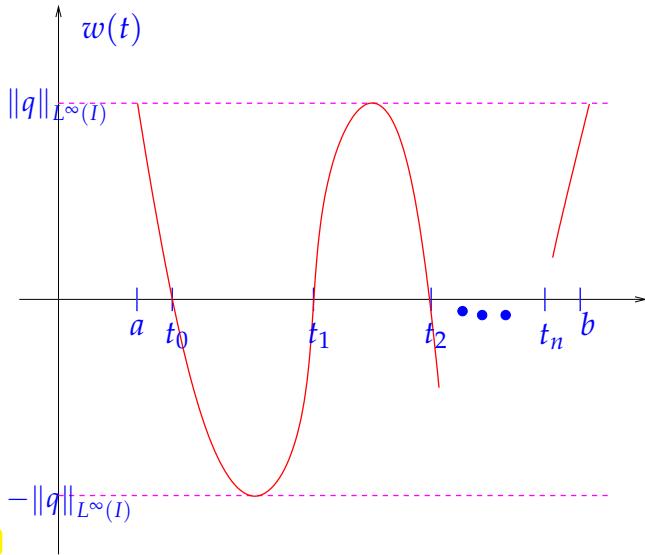


Fig. 125

Heuristic demands:

- $t^*$  extremal point of  $q \rightarrow |q(t^*)| = \|q\|_{L^\infty(I)}$ ,
- $q$  has  $n+1$  zeros in  $I$  (\*),
- $|q(-1)| = |q(1)| = \|q\|_{L^\infty(I)}$ .

(\*) is motivated by an indirect argument:

If  $q(t) = (t - t_0) \cdots (t - t_{n+1})$  with  $t_0 < -1$ , then

$$|p(t)| := |(t + 1) \cdots (t - t_{n+1})| < |q(t)| \quad \forall t \in I \quad (\text{why?}),$$

which contradicts the minimality property of  $q$ . Same argument for  $t_0 > 1$ . The reasonings leading to the above heuristic demands will be elaborated in the proof of Thm. 4.1.73.

Are there polynomials satisfying these requirements? If so, do they allow a simple characterization?

#### Definition 4.1.67. Chebychev polynomials → [42, Ch. 32]

The  $n^{\text{th}}$  Chebychev polynomial is  $T_n(t) := \cos(n \arccos t)$ ,  $-1 \leq t \leq 1, n \in \mathbb{N}$ .

The next result confirms that the  $T_n$  are polynomials, indeed.

#### Theorem 4.1.68. 3-term recursion for Chebychev polynomials → [42, (32.2)]

The function  $T_n$  defined in Def. 4.1.67 satisfy the 3-term recursion

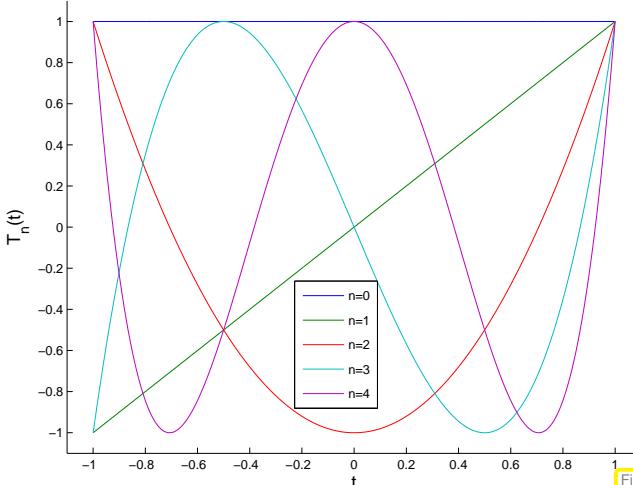
$$T_{n+1}(t) = 2t T_n(t) - T_{n-1}(t), \quad T_0 \equiv 1, \quad T_1(t) = t, \quad n \in \mathbb{N}. \quad (4.1.69)$$

*Proof.* Just use the trigonometric identity  $\cos(n+1)x = 2\cos nx \cos x - \cos(n-1)x$  with  $\cos x = t$ . □

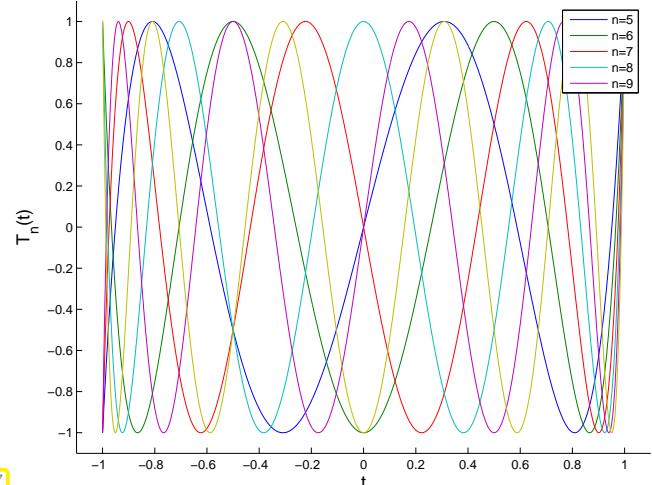
The theorem implies:

- $T_n \in \mathcal{P}_n$ ,
- their leading coefficients are equal to  $2^{n-1}$ ,
- the  $T_n$  are linearly independent,
- $\{T_j\}_{j=0}^n$  is a basis of  $\mathcal{P}_n = \text{Span}\{T_0, \dots, T_n\}$ ,  $n \in \mathbb{N}_0$ .

See Code 4.1.70 for algorithmic use of the 3-term recursion (4.1.69).



Chebychev polynomials  $T_0, \dots, T_4$



Chebychev polynomials  $T_5, \dots, T_9$

#### MATLAB-code 4.1.70: Efficient evaluation of Chebychev polynomials up to a certain degree

```

1 function V = chebpoltmult(d,x)
2 % Computes the values of the Chebychev polynomials  $T_0, \dots, T_d$ ,  $d \geq 2$ 
3 % at points passed in x using the 3-term recursion (4.1.69).
4 % The values  $T_k(x_j)$ , are returned in row  $k+1$  of V.
5 V = ones(1, length(x)); %  $T_0 \equiv 1$ 
6 V = [V; x]; %  $T_1(x) = x$ 
7 for k=1:d-1
8 V = [V; 2*x.*V(end, :) - V(end-1, :)]; % 3-term recursion
9 end

```

#### MATLAB-code 4.1.71: Plotting Chebychev polynomials, see Fig. 126, 127

```

1 function chebpoltplot(nmax)
2 % plots Chebychev polynomials up to degree nmax on [-1,1]
3 N = 1000; x = -1:2/N:1;
4 V = chebpoltmult(nmax,x); % compute values of Chebychev polynomials
5 for j=0:nmax, leg{j} = sprintf('n=%i',j); end
6 plot(x,V', '-');
7 legend(leg);
8 xlabel('{\bf t}', 'Fontsize', 14);
9 ylabel('{\bf T}_n(t)', 'Fontsize', 14);
10 axis([-1.1 1.1 -1.1 1.1]);

```

From Def. 4.1.67 we conclude that  $T_n$  attains the values  $\pm 1$  in its extrema with alternating signs, thus matching our heuristic demands:

$$|T_n(\bar{t}_k)| = 1 \Leftrightarrow \exists k = 0, \dots, n: \bar{t}_k = \cos \frac{k\pi}{n}, \quad \|T_n\|_{L^\infty([-1,1])} = 1. \quad (4.1.72)$$

What is still open is the validity of the heuristics guiding the choice of the optimal nodes. The next fundamental theorem will demonstrate that, after scaling, the  $T_n$  really supply polynomials on  $[-1, 1]$  with fixed leading coefficient and minimal supremum norm.

**Theorem 4.1.73. Minimax property of the Chebychev polynomials [19, Section 7.1.4.], [42, Thm. 32.2]**

The polynomials  $T_n$  from Def. 4.1.67 minimize the supremum norm in the following sense:

$$\|T_n\|_{L^\infty([-1,1])} = \inf\{\|p\|_{L^\infty([-1,1])} : p \in \mathcal{P}_n, p(t) = 2^{n-1}t^n + \dots\}, \quad \forall n \in \mathbb{N}.$$

*Proof.* (indirect) Assume

$$\exists q \in \mathcal{P}_n, \text{ leading coefficient } = 2^{n-1} : \|q\|_{L^\infty([-1,1])} < \|T_n\|_{L^\infty([-1,1])}. \quad (4.1.74)$$

- $(T_n - q)(x) > 0$  in local maxima of  $T_n$   
 $(T_n - q)(x) < 0$  in local minima of  $T_n$

From knowledge of local extrema of  $T_n$ , see (4.1.72):

$$T_n - q \text{ changes sign at least } n+1 \text{ times}$$

$$\Rightarrow T_n - q \text{ has at least } n \text{ zeros}$$

$$T_n - q \equiv 0, \text{ because } T_n - q \in \mathcal{P}_{n-1} \text{ (same leading coefficient!)}$$

This cannot be reconciled with the properties (4.1.74) of  $q$  and, thus, leads to a contradiction.  $\square$

The zeros of  $T_n$  are

$$t_k = \cos\left(\frac{2k+1}{2n}\pi\right), \quad k = 0, \dots, n-1. \quad (4.1.75)$$

To see this, notice

$$\begin{aligned} T_n(t) = 0 &\stackrel{\text{zeros of } \cos}{\Leftrightarrow} n \arccos t \in (2\mathbb{Z} + 1)\frac{\pi}{2} \\ &\stackrel{\arccos \in [0, \pi]}{\Leftrightarrow} t \in \left\{ \cos\left(\frac{2k+1}{n}\frac{\pi}{2}\right), k = 0, \dots, n-1 \right\}. \end{aligned}$$

Thus, we have identified the  $t_k$  from (4.1.75) as optimal interpolation nodes for a Lagrangian approximation scheme. The  $t_k$  are known as **Chebychev nodes**. Their distribution in  $[-1, 1]$  is plotted in Fig. 128, a geometric construction is indicated in Fig. 129.

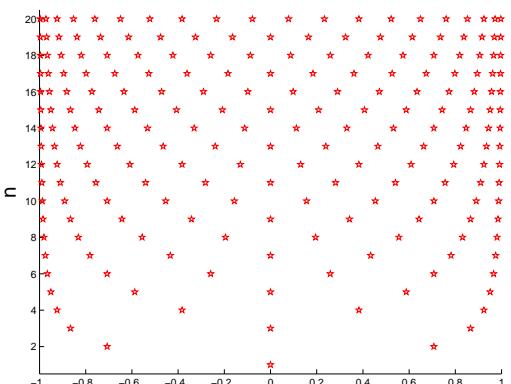


Fig. 128

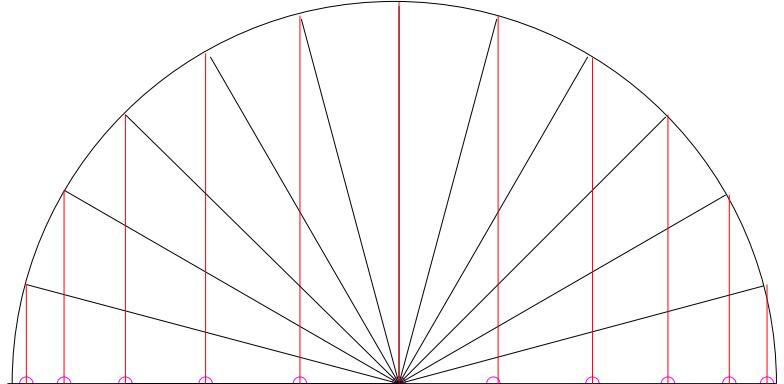


Fig. 129

When we use Chebychev nodes for polynomial interpolation we call the resulting Lagrangian approximation scheme **Chebychev interpolation**. On the interval  $[-1, 1]$  it is characterized by:

- “optimal” interpolation nodes  $\mathcal{T} = \left\{ \cos\left(\frac{2k+1}{2(n+1)}\pi\right), k = 0, \dots, n \right\}$ ,
- $w(t) = (t - t_0) \cdots (t - t_n) = 2^{-n} T_{n+1}(t)$ ,  $\|w\|_{L^\infty(I)} = 2^{-n}$ , with leading coefficient 1.

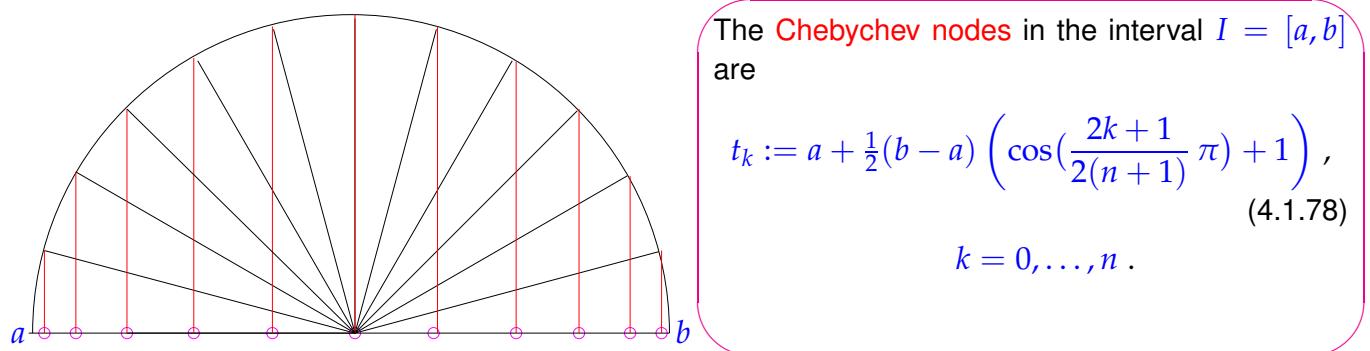
Then, by Thm. 4.1.37, we immediately get an interpolation error estimate for Chebychev interpolation of  $f \in C^{n+1}([-1, 1])$ :

$$\|f - I_T(f)\|_{L^\infty([-1, 1])} \leq \frac{2^{-n}}{(n+1)!} \|f^{(n+1)}\|_{L^\infty([-1, 1])}. \quad (4.1.76)$$

### Remark 4.1.77 (Chebychev polynomials on arbitrary interval)

Following the recipe of Rem. 4.1.14 Chebychev interpolation on an arbitrary interval  $[a, b]$  can immediately be defined. The same polynomial Lagrangian approximation scheme is obtained by transforming the Chebychev nodes (4.1.75) from  $[-1, 1]$  to  $[a, b]$  using the unique **affine transformation** (4.1.15):

$$\hat{t} \in [-1, 1] \mapsto t := a + \frac{1}{2}(\hat{t} + 1)(b - a) \in [a, b].$$



$$p \in \mathcal{P}_n \wedge p(t_j) = f(t_j) \Leftrightarrow \hat{p} \in \mathcal{P}_n \wedge \hat{p}(\hat{t}_j) = \hat{f}(\hat{t}_j).$$

With transformation formula for the integrals &  $\frac{d^n \hat{f}}{d\hat{t}^n}(\hat{t}) = (\frac{1}{2}|I|)^n \frac{d^n f}{dt^n}(t)$ :

$$\begin{aligned} \|f - I_T(f)\|_{L^\infty(I)} &= \|\hat{f} - I_{\hat{\mathcal{T}}}(\hat{f})\|_{L^\infty([-1, 1])} \leq \frac{2^{-n}}{(n+1)!} \left\| \frac{d^{n+1} \hat{f}}{d\hat{t}^{n+1}} \right\|_{L^\infty([-1, 1])} \\ &\leq \frac{2^{-2n-1}}{(n+1)!} |I|^{n+1} \|f^{(n+1)}\|_{L^\infty(I)}. \end{aligned} \quad (4.1.79)$$

### 4.1.3.2 Chebychev interpolation error estimates

#### Example 4.1.80 (Polynomial interpolation: Chebychev nodes versus equidistant nodes)

We consider Runge's function  $f(t) = \frac{1}{1+t^2}$ , see Ex. 4.1.34, and compare polynomial interpolation based on uniformly spaced nodes and Chebychev nodes in terms of behavior of interpolants.

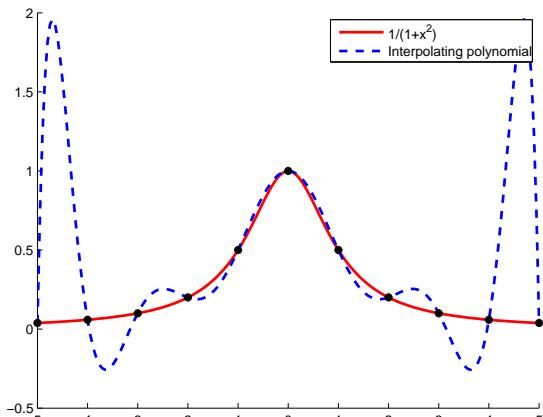


Fig. 130

Equidistant nodes

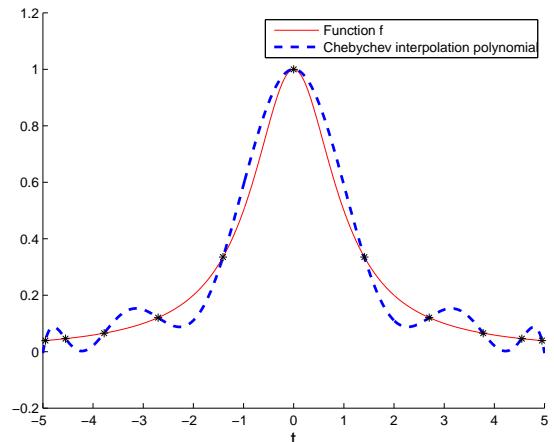


Fig. 131

Chebychev nodes

We observe that the Chebychev nodes cluster at the endpoints of the interval, which successfully suppresses the huge oscillations haunting equidistant interpolation there.

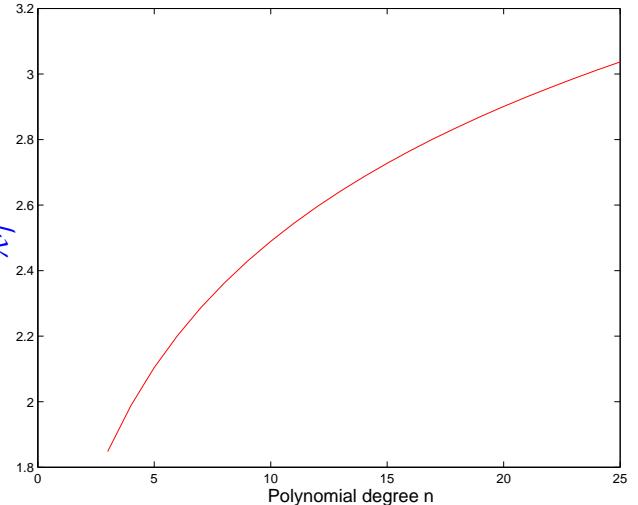
**Remark 4.1.81 (Lebesgue Constant for Chebychev nodes)** → [Section 3.2.4](#)

We saw in Rem. 3.2.70 that the Lebesgue constant  $\lambda_{\mathcal{T}}$  that measures the sensitivity of a polynomial interpolation scheme, blows up exponentially with increasing number of *equispaced* interpolation nodes. In stark contrast  $\lambda_{\mathcal{T}}$  grows only *logarithmically* in the number of Chebychev nodes.

More precisely, sophisticated theory [12, 84, 83] supplies the bound

$$\lambda_{\mathcal{T}} \leq \frac{2}{\pi} \log(1 + n) + 1 . \quad (4.1.82)$$

Measured Lebesgue constant for Chebychev nodes based on approximate evaluation of (3.2.68) by sampling. ▷



Combining (4.1.82) with the general estimate from Rem. 4.1.49

$$\|f - L_{\mathcal{T}}f\|_{L^\infty(I)} \leq (1 + \lambda_{\mathcal{T}}) \inf_{p \in \mathcal{P}_n} \|f - p\|_{L^\infty(I)} \quad \forall f \in C^0(I) , \quad (4.1.53)$$

and the bound for the best approximation error for polynomials from Thm. 4.1.11

$$\inf_{p \in \mathcal{P}_n} \|f - p\|_{L^\infty([-1,1])} \leq (1 + \pi^2/2)^r \frac{(n-r)!}{n!} \|f^{(r)}\|_{L^\infty([-1,1])} ,$$

we end up with a bound for the supremum norm of the interpolation error in the case of Chebychev interpolation on  $[-1, 1]$

$$\|f - L_{\mathcal{T}}f\|_{L^\infty([-1,1])} \leq (2/\pi \log(1 + n) + 2)(1 + \pi^2/2)^r \frac{(n-r)!}{n!} \|f^{(r)}\|_{L^\infty([-1,1])} . \quad (4.1.83)$$

### Experiment 4.1.84 (Chebychev interpolation errors)

Now we empirically investigate the behavior of norms of the interpolation error for Chebychev interpolation and functions with different (smoothness) properties as we increase the number of interpolation nodes.

In the experiments, for  $I = [a, b]$  we set  $x_l := a + \frac{b-a}{N}l$ ,  $l = 0, \dots, N$ ,  $N = 1000$ , and we approximate the norms of the interpolation error as follows ( $p \doteq$  interpolating polynomial):

$$\|f - p\|_\infty \approx \max_{0 \leq l \leq N} |f(x_l) - p(x_l)| \quad (4.1.85)$$

$$\|f - p\|_2^2 \approx \frac{b-a}{2N} \sum_{0 \leq l < N} (|f(x_l) - p(x_l)|^2 + |f(x_{l+1}) - p(x_{l+1})|^2) \quad (4.1.86)$$

- ①  $f(t) = (1+t^2)^{-1}$ ,  $I = [-5, 5]$  (see Ex. 4.1.34): analytic in a neighborhood of  $I$ .  
Interpolation with  $n = 10$  Chebychev nodes (plot on the left).

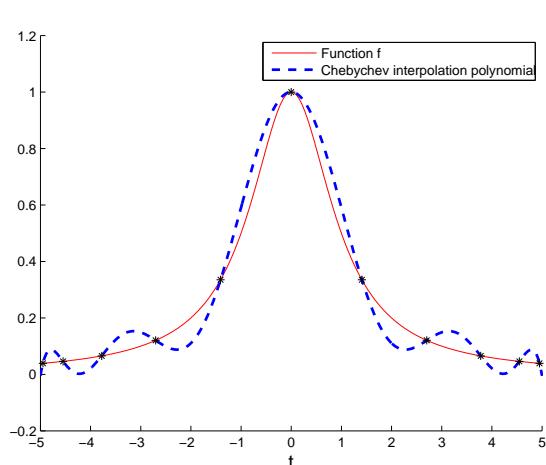
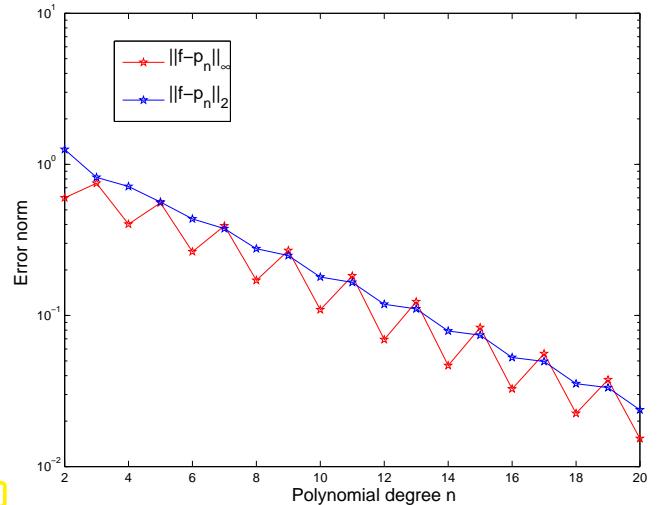


Fig. 132



Notice: exponential convergence ( $\rightarrow$  Def. 4.1.31) of the Chebychev interpolation:

$$p_n \rightarrow f, \quad \|f - I_n f\|_{L^\infty([-5,5])} \approx 0.8^n$$

- ②  $f(t) = \max\{1 - |t|, 0\}$ ,  $I = [-2, 2]$ ,  $n = 10$  nodes (plot on the left).

Now  $f \in C^0(I)$  but  $f \notin C^1(I)$ .

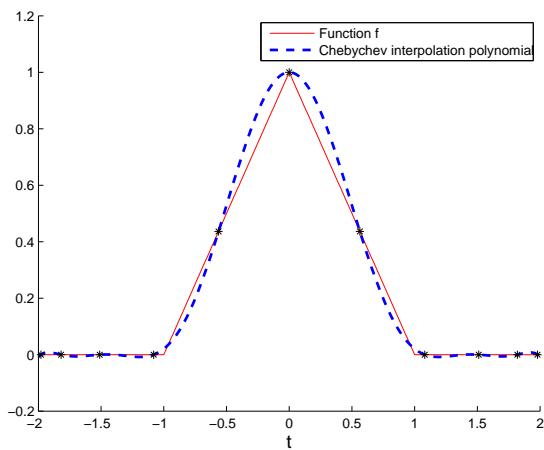


Fig. 133

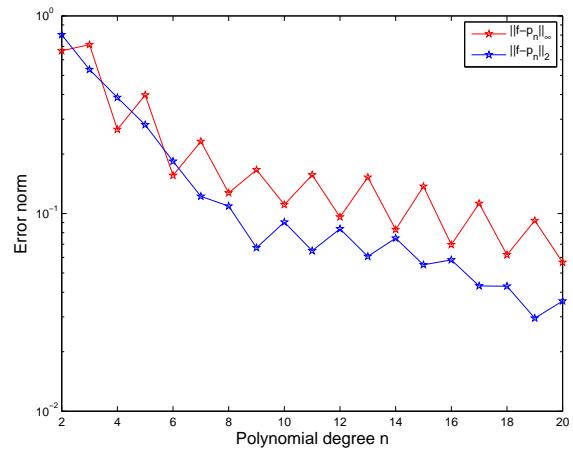


Fig. 134

From the doubly logarithmic plot we conclude →

- no exponential convergence
- algebraic convergence (?)

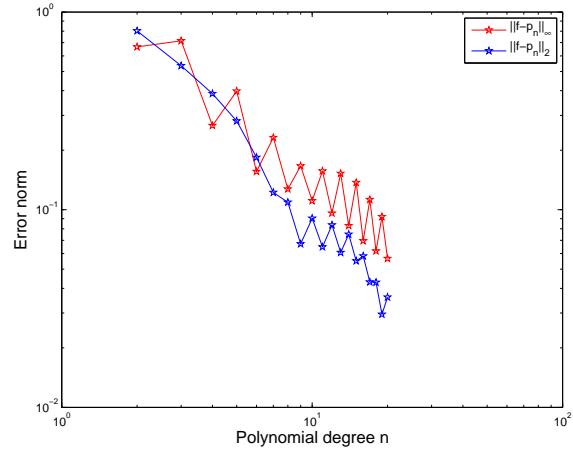


Fig. 135

$$\textcircled{3} \quad f(t) = \begin{cases} \frac{1}{2}(1 + \cos \pi t) & |t| < 1 \\ 0 & 1 \leq |t| \leq 2 \end{cases} \quad I = [-2, 2], \quad n = 10 \quad (\text{plot on the left}).$$

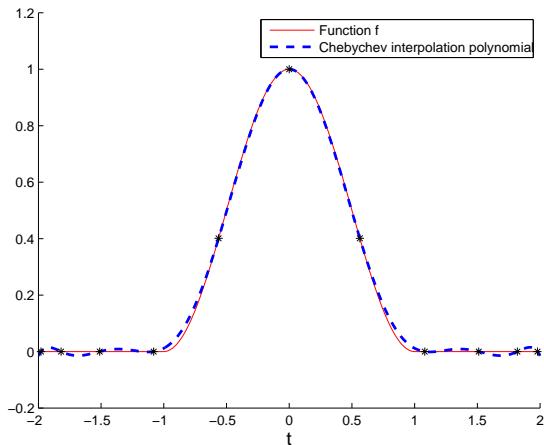


Fig. 136

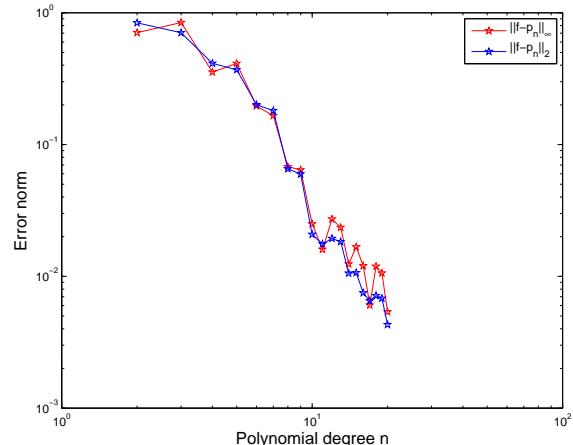


Fig. 137

Notice: only (vaguely) algebraic convergence.

Summary of observations, cf. Rem. 4.1.33:

- \* Essential role of smoothness of  $f$ : slow convergence of approximation error of the Cheychev interpolant if  $f$  enjoys little smoothness, cf. also (4.1.43),
- \* for smooth  $f \in C^\infty$  approximation error of the Cheychev interpolant seems to decay to zero **exponentially** in the polynomial degree  $n$ .

**Remark 4.1.87 (Chebychev interpolation of analytic functions)**

Assuming that the interpoland  $f$  possesses an analytic extension to a complex neighborhood  $D$  of  $[-1, 1]$ , we now apply the theory of § 4.1.54 to bound the supremum norm of the Chebychev interpolation error of  $f$  on  $[-1, 1]$ .

To convert

$$|f(t) - \mathcal{L}_T f(t)| \leq \left| \frac{w(t)}{2\pi i} \int_{\gamma} \frac{f(z)}{(z-t)w(z)} dz \right| \leq \frac{|\gamma|}{2\pi} \frac{\max_{a \leq \tau \leq b} |w(\tau)|}{\min_{z \in \gamma} |w(z)|} \cdot \frac{\max_{z \in \gamma} |f(z)|}{\text{dist}([a, b], \gamma)}, \quad (4.1.63)$$

as obtained in § 4.1.54, into a more concrete estimate, we have to study the behavior of

$$w_n(t) = (t - t_0)(t - t_1) \cdots (t - t_n), \quad t_k = \cos\left(\frac{2k+1}{2n+2}\pi\right), \quad k = 0, \dots, n,$$

where the  $t_k$  are the Chebychev nodes according to (4.1.78). They are the zeros of the Chebychev polynomial ( $\rightarrow$  Def. 4.1.67) of degree  $n+1$ . Since  $w$  has leading coefficient 1, we conclude  $w = 2^{-n}T_{n+1}$ , and

$$\max_{-1 \leq t \leq 1} |w(t)| \leq 2^{-n}. \quad (4.1.88)$$

Next, we fix a suitable path  $\gamma \subset D$  for integration: For a constant  $\rho > 1$  we set

$$\begin{aligned} \gamma &:= \{z = \cos(\theta - i \log \rho), 0 \leq \theta \leq 2\pi\} \\ &= \left\{ z = \frac{1}{2}(\exp(i(\theta - i \log \rho)) + \exp(-i(\theta - i \log \rho))), 0 \leq \theta \leq 2\pi \right\} \\ &= \left\{ z = \frac{1}{2}(\rho e^{i\theta} + \rho^{-1} e^{-iu\theta}), 0 \leq \theta \leq 2\pi \right\} \\ &= \left\{ z = \frac{1}{2}(\rho + \rho^{-1}) \cos \theta + i \frac{1}{2}(\rho - \rho^{-1}) \sin \theta, 0 \leq \theta \leq 2\pi \right\} \end{aligned}$$

Thus, we see that  $\gamma$  is an **ellipsis** with foci  $\pm 1$ , large axis  $\frac{1}{2}(\rho + \rho^{-1}) > 1$  and small axis  $\frac{1}{2}(\rho - \rho^{-1}) > 0$ .

Elliptical integration contours for different values of  $\rho$

▷

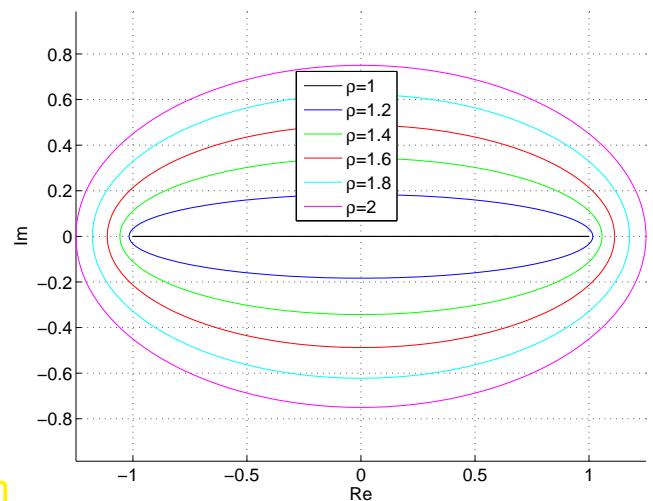


Fig. 138

Appealing to geometric evidence, we find  $\text{dist}(\gamma, [-1, 1]) = \frac{1}{2}(\rho + \rho^{-1}) - 1$ , which gives another term in (4.1.63).

The rationale for choosing this particular integration contour is that the **cos** in its definition nicely cancels

the  $\arccos$  in the formula for the Chebychev polynomials. This lets us compute ( $s := n + 1$ )

$$\begin{aligned}
 |T_s(\cos(\theta - i \log \rho))|^2 &= |\cos(s(\theta - i \log \rho))|^2 \\
 &= \cos(s(\theta - i \log \rho)) \cdot \overline{\cos(s(\theta - i \log \rho))} \\
 &= \frac{1}{4}(\rho^s e^{is\theta} + \rho^{-s} e^{-is\theta})(\rho^s e^{-is\theta} + \rho^{-s} e^{is\theta}) \\
 &= \frac{1}{4}(\rho^{2s} + \rho^{-2s} + e^{2is\theta} + e^{-2is\theta}) \\
 &= \frac{1}{4}(\rho^s - \underbrace{\rho^{-s}}_{<1})^2 + \underbrace{\frac{1}{4}(e^{is\theta} + e^{-is\theta})^2}_{=\cos^2(s\theta) \geq 0} \geq \frac{1}{4}(\rho^s - 1)^2,
 \end{aligned}$$

for all  $0 \leq \theta \leq 2\pi$ , which provides a lower bound for  $|w_n|$  on  $\gamma$ . Plugging all these estimates into (4.1.63) we arrive at

$$\|f - L_T f\|_{L^\infty([-1,1])} \leq \frac{2|\gamma|}{\pi} \frac{1}{(\rho^{n+1} - 1)(\rho + \rho^{-1} - 2)} \cdot \max_{z \in \gamma} |f(z)|. \quad (4.1.89)$$

Note that instead on the nodal polynomial  $w$  we have inserted  $T_{n+1}$  into (4.1.63), which is a simple multiple. The factor will cancel.

► The supremum norm of the interpolation converges exponentially ( $\rho > 1!$ ):

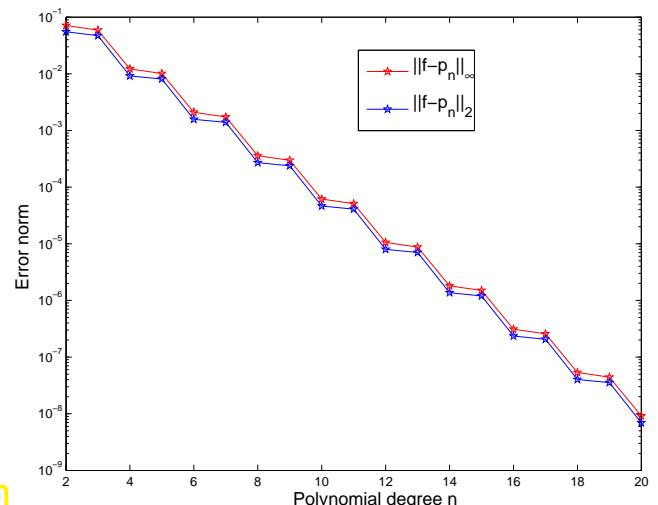
$$\|f - L_T f\|_{L^\infty([-1,1])} = O((2\rho)^{-n}) \quad \text{for } n \rightarrow \infty.$$

### Experiment 4.1.90 (Chebychev interpolation of analytic function → Exp. 4.1.84 cnt'd)

Modification: the same function  $f(t) = (1 + t^2)^{-1}$  on a smaller interval  $I = [-1, 1]$ .

(Faster) exponential convergence than on the interval  $I = [-5, 5]$ :

$$\|f - I_n f\|_{L^2([-1,1])} \approx 0.42^n.$$



Explanation, cf. Rem. 4.1.87: for  $I = [-1, 1]$  the poles  $\pm i$  of  $f$  are farther away relative to the size of the interval than for  $I = [-5, 5]$ .

#### 4.1.3.3 Chebychev interpolation: computational aspects

Task: Given: given degree  $n \in \mathbb{N}$ , continuous function  $f : [-1, 1] \mapsto \mathbb{R}$

Sought: efficient representation/evaluation of Chebychev interpolant  $p \in \mathcal{P}_n$  (= polynomial Lagrange interpolant of degree  $\leq n$  in Chebychev nodes (4.1.78) on  $[-1, 1]$ ).

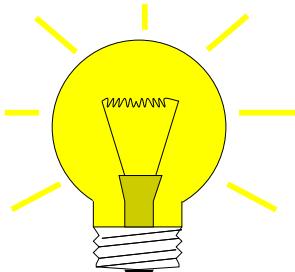
More concretely, this boils down to a implementation of the following class:

##### MATLAB-code 4.1.91: Definition of class for Chebychev interpolation

```

1 class ChebInterp {
2   private:
3     // various internal data describing Chebychev interpolating
4     // polynomial p
5   public:
6     // Constructor taking function f and degree n as arguments
7     template <typename Function>
8       PolyInterp(const Function &f, unsigned int n);
9     // Evaluation operator: y_j = p(x_j), j = 1, ..., m (m "large")
10    double eval(const vector<double> &x, vector <double> &y) const;
11  };

```



Trick: internally represent  $p$  as a linear combination of Chebychev polynomials, a Chebychev expansion:

$$p = \sum_{j=0}^n \alpha_j T_j, \quad \alpha_j \in \mathbb{R}. \quad (4.1.92)$$

The representation (4.1.92) is always possible, because  $\{T_0, \dots, T_n\}$  is a basis of  $\mathcal{P}_n$ .

##### Remark 4.1.93 (Fast evaluation of Chebychev expansion → [42, Alg. 32.1])

Use the 3-term recurrence (4.1.69)

$$T_j(x) = 2xT_{j-1}(x) - T_{j-2}(x), \quad j = 2, 3, \dots,$$

to rewrite (4.1.92) as

$$p(x) = \sum_{j=0}^{n-1} \tilde{\alpha}_j T_j \quad \text{with} \quad \tilde{\alpha}_j = \begin{cases} \alpha_j + 2x\alpha_{j+1} & , \text{if } j = n-1, \\ \alpha_j - \alpha_{j+2} & , \text{if } j = n-2, \\ \alpha_j & \text{else.} \end{cases} \quad (4.1.94)$$

► recursive algorithm, see Code 4.1.95.

##### MATLAB-code 4.1.95: Recursive evaluation of Chebychev expansion (4.1.92)

```

1 function y = recclenshaw(a, x)

```

```

2 | % Recursive evaluation of a polynomial  $p = \sum_{j=1}^{n+1} a_j T_{j-1}$  at point  $x$ , see
   | (4.1.94)
3 | % The coefficients  $a_j$  have to be passed in a row vector.
4 n = length(a)-1;
5 if (n<2), y = a(1)+x*a(2);
6 else
7 y = recclelshaw([a(1:n-2), a(n-1)-a(n+1), a(n)+2*x*a(n+1)], x);
8 end

```

Non-recursive version: **Clenshaw algorithm**

#### MATLAB-code 4.1.96: Clenshaw algorithm for evalation of Chebychev expansion (4.1.92)

```

1 function y = clenshaw(a,x)
2 % Clenshaw algorithm for evaluating  $p = \sum_{j=1}^{n+1} a_j T_{j-1}$  at points passed in
   % vector x
3 n = length(a)-1; % degree of polynomial
4 d = repmat(reshape(a,n+1,1),1,length(x));
5 for j=n:-1:2
6 d(j,:) = d(j,:) + 2*x.*d(j+1,:); % see (4.1.94)
7 d(j-1,:) = d(j-1,:) - d(j+1,:);
8 end
9 y = d(1,:) + x.*d(2,:);

```

► Computational effort :  $O(nm)$  for evaluation at  $m$  points

Issue: How to compute the Chebychev expansion coefficients  $\alpha_j$  in (4.1.92) efficiently from the interpolation conditions

$$p(t_k) = f(t_k), \quad k = 0, \dots, n, \quad \text{for } t_k := \cos\left(\frac{2k+1}{2(n+1)}\pi\right). \quad (4.1.97)$$

↑  
Chebychev nodes

#### (4.1.98) Computation of Chebychev expansions

Chebychev interpolation is a linear interpolation scheme, see § 3.1.11. Thus, the expansion  $\alpha_j$  in (4.1.92) can be computed by solving a linear system of equations of the form (3.1.13). However, for Chebychev interpolation this linear system can be cast into a very special form, which paves the way for its fast direct

solution:

Trick: transformation of  $p$  into a 1-periodic function:

$$\begin{aligned}
 q(s) &:= p(\cos 2\pi s) = \sum_{j=0}^n \alpha_j T_j(\cos 2\pi s) \exp(-2\pi i n s) \stackrel{\text{Def. 4.1.67}}{=} \sum_{j=0}^n \alpha_j \cos(2\pi j s) \\
 &= \sum_{j=0}^n \frac{1}{2} \alpha_j (\exp(2\pi i j s) + \exp(-2\pi i j s)) \quad [\text{by } \cos z = \frac{1}{2}(e^z + e^{-z})] \\
 &= \sum_{j=-n}^{n+1} \beta_j \exp(-2\pi i j s), \quad \text{with } \beta_j := \begin{cases} 0 & , \text{for } j = n+1, \\ \frac{1}{2} \alpha_j & , \text{for } j = 1, \dots, n, \\ \alpha_0 & , \text{for } j = 0, \\ \frac{1}{2} \alpha_{-j} & , \text{for } j = -n, \dots, -1. \end{cases}
 \end{aligned} \tag{4.1.99}$$

Transformed interpolation conditions (4.1.97) for  $q$ :

$$t = \cos(2\pi s) \stackrel{(4.1.97)}{\implies} q\left(\frac{2k+1}{4(n+1)}\right) = y_k := f(t_k), \quad k = 0, \dots, n. \tag{4.1.100}$$

Also observe the symmetry

$$\begin{aligned}
 q(s) &= q(1-s) \\
 \Downarrow \leftarrow (4.1.100) \\
 q\left(1 - \frac{2k+1}{4(n+1)}\right) &= y_k, \quad k = 0, \dots, n.
 \end{aligned}$$

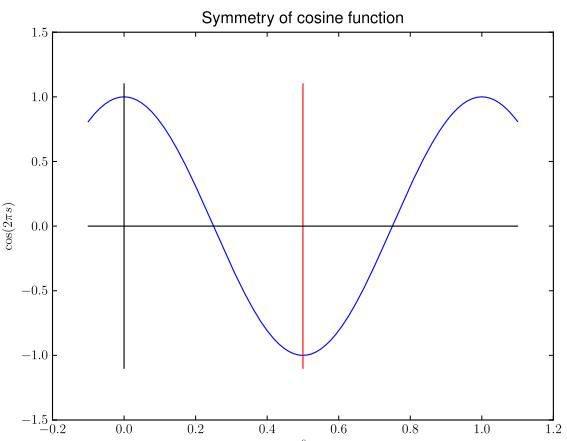


Fig. 140

Extend interpolation conditions (4.1.100) by symmetry, see Fig. 141

$$q\left(\frac{k}{2(n+1)} + \frac{1}{4(n+1)}\right) = z_k := \begin{cases} y_k & , \text{for } k = 0, \dots, n, \\ y_{2n+1-k} & , \text{for } k = n+1, \dots, 2n+1. \end{cases} \tag{4.1.101}$$

In a sense, we can mirror the interpolation conditions at  $x = \frac{1}{2}$ :

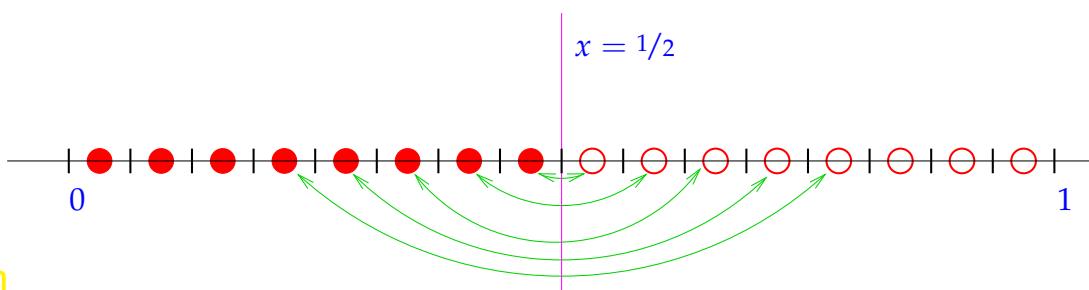


Fig. 141

(4.1.101)  $\iff$  linear system of equations (at last):

$$\begin{aligned}
 q\left(\frac{k}{2(n+1)} + \frac{1}{4(n+1)}\right) &= \sum_{j=-n}^{n+1} \left( \beta_j \exp\left(-\frac{2\pi ij}{4(n+1)}\right) \right) \exp\left(-\frac{2\pi i}{2n+1}kj\right) = z_k . \\
 &\Updownarrow \\
 \sum_{j=0}^{2n+1} \left( \beta_j \exp\left(-\frac{2\pi i(j-n)}{4(n+1)}\right) \right) &= \underbrace{\exp\left(-\frac{2\pi i}{2n+1}kj\right)}_{\omega_{2(n+1)}^{kj}} ! = \exp\left(-\pi i \frac{nk}{n+1}\right) z_k , \quad k = 0, \dots, 2n+1 . \\
 &\Updownarrow \\
 \mathbf{F}_{2(n+1)} \mathbf{c} = \mathbf{z} \quad \text{with} \quad \mathbf{c} = \left( \beta_j \exp\left(-\frac{2\pi ij}{4(n+1)}\right) \right)_{j=0}^{2n+1} . \tag{4.1.102}
 \end{aligned}$$

(2n+2) × (2n+2) Fourier matrix, see (9.2.8)

► solve (4.1.102) with inverse discrete Fourier transform, see 9.2:

asymptotic complexity  $\mathcal{O}(n \log n)$  ( $\rightarrow$  Section 9.3)

Note: by symmetry of  $\mathbf{z}$   $\Rightarrow \beta_{2n+1} = 0$ , cf. (4.1.99)!

#### MATLAB-code 4.1.103: Efficient computation of Chebychev expansion coefficient of Chebychev interpolant

```

1 function a = chebexp(y)
2 % Efficiently compute coefficients  $\alpha_j$  in the Chebychev expansion
3 %  $p = \sum_{j=0}^n \alpha_j T_j$  of  $p \in \mathcal{P}_n$  based on values  $y_k$ ,
4 %  $k = 0, \dots, n$ , in Chebychev nodes  $t_k$ ,  $k = 0, \dots, n$ . These values are
5 % passed in the row vector  $y$ .
6 n = length(y)-1; % degree of polynomial
7 % create vector  $\mathbf{z}$  by wrapping and componentwise scaling
8 z = exp(-pi*i*n/(n+1)*(0:2*n+1)).*[y,y(end:-1:-1)]; % r.h.s.
9 % vector
10 c = ifft(z); % Solve linear system (4.1.102) with effort  $\mathcal{O}(n \log n)$ 
11 b = real(exp(0.5*pi*i/(n+1)*(-n:n+1)).*c); % recover  $\beta_j$ , see (4.1.102)
12 a = [b(n+1), 2*b(n+2:2*n+1)]; % recover  $\alpha_j$ , see (4.1.99)

```

#### Remark 4.1.104 (Chebychev representation of built-in functions)

Computers use approximation by sums of Chebychev polynomials in the computation of functions like `log`, `exp`, `sin`, `cos`, .... The evaluation by means of Clenshaw algorithm according to Code 4.1.96 is more efficient and stable than the approximation by Taylor polynomials.

## 4.2 Mean Square Best Approximation

There is a particular family of norms for which the best approximant of a function  $f$  in a finite dimensional function space  $V_N$ , that is, the element of  $V_N$  that is closest to  $f$  with respect to that particular norm can actually be computed. It turns out that this computation boils down to solving a kind of least squares problem, similar to the least squares problems in  $\mathbb{K}^n$  discussed in Chapter 6.

### 4.2.1 Abstract theory

Concerning mean square best approximation it is useful to learn an abstract framework first into which the concrete examples can be fit later.

#### 4.2.1.1 Mean square norms

Mean square norms generalize the Euclidean norm on  $\mathbb{K}^n$ , see [59, Sect. 4.4]. In a sense, they endow a vector space with a geometry and give a meaning to concepts like “orthogonality”.

##### Definition 4.2.1. (Semi-)inner product [59, Sect. 4.4]

Let  $V$  be a vector space over the field  $\mathbb{K}$ . A mapping  $b : V \times V \rightarrow \mathbb{K}$  is called an **inner product** on  $V$ , if it satisfies

- (i)  $b$  is **linear** in the first argument:  $b(\alpha v + \beta w, u) = \alpha b(v, u) + \beta b(w, u)$  for all  $\alpha, \beta \in \mathbb{K}$ ,  $v, w \in V$ ,
- (ii)  $b$  is (anti-)**symmetric**:  $b(v, w) = \overline{b(w, v)}$  ( $\overline{\phantom{x}} \hat{=} \text{complex conjugation}$ ),
- (iii)  $b$  is **positive definite**:  $v \neq 0 \Leftrightarrow b(v, v) > 0$ .

$b$  is a **semi-inner product**, if it still complies with (i) and (ii), but is only positive semi-definite:  $b(v, v) \geq 0$  for all  $v \in V$ .

notation: usually we write  $(\cdot, \cdot)_V$  for an inner product on the vector space  $V$ .

##### Definition 4.2.2. Orthogonality

Let  $V$  be a vector space equipped with a (semi-)inner product  $(\cdot, \cdot)_V$ . Any two elements  $v$  and  $w$  of  $V$  are called **orthogonal**, if  $(v, w)_V = 0$ . We write  $v \perp w$ .

notation: If  $W \subset V$  is a (closed) subspace:  $v \perp W \Leftrightarrow (v, w)_V = 0 \forall w \in W$ .

##### Theorem 4.2.3. Mean square (semi-)norm/Inner product (semi-)norm

If  $(\cdot, \cdot)_V$  is a (semi-)inner product ( $\rightarrow$  Def. 4.2.1) on the vector space  $V$ , then

$$\|v\|_V := \sqrt{(\overline{v}, v)_V}$$

defines a (semi-)norm ( $\rightarrow$  Def. 1.5.65) on  $V$ , the **mean square (semi-)norm**/ inner product (semi-)norm induced by  $(\cdot, \cdot)_V$ .

#### (4.2.4) Examples for mean square norms

- \* The Euclidean norm on  $\mathbb{K}^n$  induced by the dot product (Euclidean inner product).
- \* The  $L^2$ -norm (3.2.63) on  $C^0([a, b])$  induced by the  $L^2([a, b])$  inner product

$$(f, g)_{L^2([a, b])} := \int_a^b f(\tau) \bar{g}(\tau) d\tau, \quad f, g \in C^0([a, b]). \quad (4.2.5)$$

##### 4.2.1.2 Normal equations

$X \triangleq$  a vector space over  $\mathbb{K} = \mathbb{R}$ , equipped with a mean square semi-norm  $\|\cdot\|_X$  induced by a semi-inner product  $(\cdot, \cdot)_X$ , see Thm. 4.2.3.  
It can be an infinite dimensional function space, e.g.,  $X = C^0([a, b])$ .

$V \triangleq$  a finite-dimensional subspace of  $X$ , with basis  $\mathfrak{B}_V := \{b_1, \dots, b_N\} \subset V$ ,  $N := \dim V$ .

##### Assumption 4.2.6.

The semi-inner product  $(\cdot, \cdot)_X$  is a genuine inner product ( $\rightarrow$  Def. 4.2.1) on  $V$ , that is, it is positive definite:  $(v, v)_X > 0 \forall v \in V \setminus \{0\}$ .

Now we give a formula for the element  $q$  of  $V$ , which is nearest to a given element  $f$  of  $X$  with respect to the norm  $\|\cdot\|_X$ .

##### Theorem 4.2.7. Mean square norm best approximation through normal equations

Given any  $f \in X$  there is a unique  $q \in V$  such that

$$\|f - q\|_X = \inf_{p \in V} \|f - p\|_X.$$

Its coefficients  $\gamma_j$ ,  $j = 1, \dots, N$ , with respect to the basis  $\mathfrak{B}_V := \{b_1, \dots, b_N\}$  of  $V$  ( $q = \sum_{j=1}^N \gamma_j b_j$ ) are the unique solution of the normal equations

$$\mathbf{M}[\gamma_j]_{j=1}^N = [(f, b_j)]_{j=1}^N, \quad \mathbf{M} := \begin{bmatrix} (b_1, b_1)_X & \dots & (b_1, b_N)_X \\ \vdots & & \vdots \\ (b_N, b_1)_X & \dots & (b_N, b_N)_X \end{bmatrix} \in \mathbb{K}^{N,N}. \quad (4.2.8)$$

*Proof.* We first show that  $\mathbf{M}$  is s.p.d. ( $\rightarrow$  Def. 1.1.8). Symmetry is clear from the definition and the symmetry of  $(\cdot, \cdot)_X$ . That  $\mathbf{M}$  is even positive definite follows from

$$\mathbf{x}^H \mathbf{M} \mathbf{x} = \sum_{k=1}^N \sum_{j=1}^N \xi_k \bar{\xi}_j (b_k, b_j)_X = \left\| \sum_{j=1}^N \xi_j b_j \right\|_X^2 > 0, \quad (4.2.9)$$

if  $\mathbf{x} := [\xi_j]_{j=1}^N \neq \mathbf{0} \Leftrightarrow \sum_{j=1}^N \xi_j b_j \neq 0$ , since  $\|\cdot\|_X$  is a norm on  $V$  by Ass. 4.2.6.

Now, writing  $\mathbf{c} := [\gamma_j]_{j=1}^N \in \mathbb{K}^N$ ,  $\mathbf{b} := [(f, b_j)_X]_{j=1}^N \in \mathbb{K}^N$ , and using the basis representation

$$q = \sum_{j=1}^N \gamma_j b_j ,$$

we find

$$\Phi(\mathbf{c}) := \|f - q\|_X^2 = \|f\|_X^2 - 2\mathbf{b}^\top \mathbf{c} + \mathbf{c}^\top \mathbf{M} \mathbf{c} .$$

Applying the differentiation rules from Ex. 2.4.11 to  $\Phi : \mathbb{K}^N \rightarrow \mathbb{R}$ ,  $\mathbf{c} \mapsto \Phi(\mathbf{c})$ , we obtain

$$\text{grad } \Phi(\mathbf{c}) = 2(\mathbf{M}\mathbf{c} - \mathbf{b}) , \quad (4.2.10)$$

$$\mathbf{H} \Phi(\mathbf{c}) = 2\mathbf{M} . \quad (\text{independent of } \mathbf{c}!) \quad (4.2.11)$$

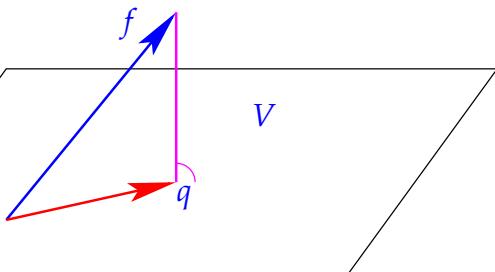
Since  $\mathbf{M}$  is s.p.d., the unique solution of  $\text{grad } \Phi(\mathbf{c}) = \mathbf{M}\mathbf{c} - \mathbf{b} = 0$  yields the unique global minimizer of  $\Phi$ ; the Hessian  $2\mathbf{M}$  is s.p.d. everywhere!  $\square$

The unique  $q$  from Thm. 4.2.7 is called the **best approximant** of  $f$  in  $V$ .

### Corollary 4.2.12. Best approximant by orthogonal projection

If  $q$  is the best approximant of  $f$  in  $V$ , then  $f - q$  is orthogonal to every  $p \in V$ :

$$(f - q, p)_X = 0 \quad \forall p \in V \Leftrightarrow f - q \perp V .$$



The message of Cor. 4.2.12:

- the best approximation error  $f - q$  for  $f \in X$  in  $V$  is orthogonal to the subspace  $V$ .

See Section 6.1 for related discussion in  $\mathbb{K}^n$ .

### 4.2.1.3 Orthonormal bases

In the setting of Section 4.2.1.2 we may ask: Which choice of basis  $\mathcal{B} = \{b_1, \dots, b_N\}$  of  $V \subset X$  renders the normal equations (4.2.8) particularly simple? Answer: A basis  $\mathcal{B}$ , for which  $(b_k, b_j)_X = \delta_{kj}$  ( $\delta_{kj}$  the Kronecker symbol), because this will imply  $\mathbf{M} = \mathbf{I}$  for the coefficient matrix of the normal equations.

### Definition 4.2.13. Orthonormal basis

A subset  $\{b_1, \dots, b_N\}$  of an  $N$ -dimensional vector space  $V$  with inner product ( $\rightarrow$  Def. 4.2.1)  $(\cdot, \cdot)_V$  is an **orthonormal basis (ONB)**, if  $(b_k, b_j)_V = \delta_{kj}$ .

A basis of  $\{b_1, \dots, b_N\}$  of  $V$  is called **orthogonal**, if  $(b_k, b_j)_V = 0$  for  $k \neq j$ .

**Corollary 4.2.14. ONB representation of best approximant**

If  $\{b_1, \dots, b_N\}$  is an orthonormal basis ( $\rightarrow$  Def. 4.2.13) of  $V \subset X$ , then the best approximant  $q := \operatorname{argmin}_{p \in V} \|f - p\|_X$  of  $f \in X$  has the representation

$$q = \sum_{j=1}^N (f, b_j)_X b_j. \quad (4.2.15)$$

**(4.2.16) Gram-Schmidt orthonormalization**

From Section 1.5.1 we already know how to compute orthonormal bases: The algorithm from § 1.5.1 can be run in the framework of any vector space  $V$  endowed with an inner product  $(\cdot, \cdot)_V$  and induced mean square norm  $\|\cdot\|_V$ .

```

1:  $b_1 := \frac{p_1}{\|p_1\|_V}$  % 1st output vector
2: for  $j = 2, \dots, k$  do
   { % Orthogonal projection
3:    $b_j := p_j$ 
4:   for  $\ell = 1, 2, \dots, j-1$  do      (4.2.17)
5:     {  $b_j \leftarrow b_j - (p_j, b_\ell)_V b_\ell$  }
6:   if ( $b_j = \mathbf{0}$ ) then STOP
7:   else {  $b_j \leftarrow \frac{b_j}{\|b_j\|_V}$  }
   }
```

**Theorem 4.2.18. Gram-Schmidt orthonormalization**

When supplied with  $k \in \mathbb{N}$  linearly independent vectors  $p_1, \dots, p_k \in V$  in a vector space with inner product  $(\cdot, \cdot)_V$ , Algorithm (4.2.17) computes vectors  $b_1, \dots, b_k$  with

$$(b_\ell, b_j)_V = \delta_{\ell j}, \quad \ell, j \in \{1, \dots, k\}, \\ \text{Span}\{b_1, \dots, b_\ell\} = \text{Span}\{p_1, \dots, p_\ell\}$$

for all  $\ell \in \{1, \dots, k\}$ .

This suggests the following alternative approach to the computation of the mean square best approximant  $q$  in  $V$  of  $f \in X$ :

- ❶ Orthonormalize a basis  $\{b_1, \dots, b_N\}$  of  $V$ ,  $N := \dim V$ , using Gram-Schmidt algorithm (4.2.17).
- ❷ Compute  $q$  according to (4.2.15).

Number of inner products to be evaluated:  $O(N^2)$  for  $N \rightarrow \infty$ .

## 4.2.2 Polynomial mean square best approximation

Now we apply the results of Section 4.2.1 in the following setting:

$X$ : function space  $C^0([a, b])$ ,  $-\infty < a < b < \infty$ , of  $\mathbb{R}$ -values continuous functions,  
 $V$ : space  $\mathcal{P}_m$  of polynomials of degree  $\leq m$ .

**Remark 4.2.19 (Inner products on spaces  $\mathcal{P}_m$  of polynomials)**

To match the abstract framework of Section 4.2.1 we need to find (semi-)inner products on  $C^0([a, b])$  that supply positive definite inner products on  $\mathcal{P}_m$ . The following options are commonly considered:

- \* On any interval  $[a, b]$  we can use the  $L^2([a, b])$ -inner product  $(\cdot, \cdot)_{L^2([a, b])}$ , defined in (4.2.5).
- \* Given a positive **weight function**  $w : [a, b] \rightarrow \mathbb{R}$ ,  $w(t) > 0$  for all  $t \in [a, b]$ , we can consider the weighted  $L^2$ -inner product on the interval  $[a, b]$

$$(f, g)_{w, [a, b]} := \int_a^b w(\tau) f(\tau) \overline{g(\tau)} d\tau . \quad (4.2.20)$$

- \* For  $n \geq m$  and  $n + 1$  distinct points collected in the set  $\mathcal{T} := \{t_0, t_1, \dots, t_n\} \subset [a, b]$  we can use the **discrete  $L^2$ -inner product**

$$(f, g)_{\mathcal{T}} := \sum_{j=0}^n f(t_j) \overline{g(t_j)} . \quad (4.2.21)$$

Since a polynomial of degree  $\leq m$  must be zero everywhere, if it vanishes in at least  $m + 1$  distinct points,  $(f, g)_{\mathcal{T}}$  is positive definite on  $\mathcal{P}_m$ .

For all these inner products on  $\mathcal{P}_m$  holds

$$(\{t \mapsto tf(t)\}, g)_X = (f, \{t \mapsto tg(t)\})_X , \quad f, g \in C^0([a, b]) , \quad (4.2.22)$$

that is, multiplication with the independent variable can be shifted to the other function inside the inner product.

☞ notation: Note that we have to plug a function into the slots of the inner products; this is indicated by the notation  $\{t \mapsto \dots\}$ .

#### Assumption 4.2.23. Self-adjointness of multiplication operator

We assume the inner product  $(\cdot, \cdot)_X$  to satisfy (4.2.22).

The ideas of Section 4.2.1.3 that center around the use of orthonormal bases can also be applied to polynomials.

#### Definition 4.2.24. Orthonormal polynomials → Def. 4.2.13

Let  $(\cdot, \cdot)_X$  be an inner product on  $\mathcal{P}_m$ . A sequence  $r_0, r_1, \dots, r_m$  provides **orthonormal polynomials** (ONPs) with respect to  $(\cdot, \cdot)_X$ , if

$$r_\ell \in \mathcal{P}_\ell , \quad (r_k, r_\ell)_X = \delta_{k\ell} , \quad \ell, k \in \{0, \dots, m\} . \quad (4.2.25)$$

The polynomials are just **orthogonal**, if  $(r_k, r_\ell)_X = 0$  for  $k \neq \ell$ .

By virtue of Thm. 4.2.18 orthonormal polynomials can be generated by applying Gram-Schmidt orthonormalization from § 4.2.16 to the ordered basis of monomials  $\{t \mapsto t^j\}_{j=0}^m$ .

**Lemma 4.2.26. Uniqueness of orthonormal polynomials**

The sequence of orthonormal polynomials from Def. 4.2.24 is unique up to signs, supplies an  $(\cdot, \cdot)_X$ -orthonormal basis ( $\rightarrow$  Def. 4.2.13) of  $\mathcal{P}_m$ , and satisfies

$$\text{Span}\{r_0, \dots, r_k\} = \mathcal{P}_k, \quad k \in \{0, \dots, m\}. \quad (4.2.27)$$

*Proof.* Comparing Def. 4.2.13 and (4.2.25) the ONB-property of  $\{r_0, \dots, r_m\}$  is immediate. Then (4.2.25) follows from dimensional considerations.

$r_0$  must be a constant, which, up to sign, is fixed by the normalization condition  $\|r_0\|_X = 1$ .

$\mathcal{P}_{k-1} \subset \mathcal{P}_k$  has co-dimension 1 so that there is a unit “vector” in  $\mathcal{P}_k$ , which is orthogonal to  $\mathcal{P}_{k-1}$  and unique up to sign.  $\square$

**(4.2.28) Orthonormal polynomials by orthogonal projection**

Let  $r_0, \dots, r_m \in \mathcal{T}_p$  be a sequence of orthonormal polynomials according to Def. 4.2.24. From (4.2.25) we conclude that  $r_k \in \mathcal{P}_k$  has leading coefficient  $\neq 0$ .

Hence  $s_k(t) := t \cdot r_k(t)$  is a polynomial of degree  $k+1$  with leading coefficient  $\neq 0$ , that is  $s_k \in \mathcal{P}_{k+1} \setminus \mathcal{P}_k$ . Therefore,  $r_{k+1}$  can be obtained by *orthogonally projecting*  $s_k$  onto  $\mathcal{P}_k$  plus normalization, cf. Lines 4-5 of Algorithm (4.2.17):

$$r_{k+1} = \pm \frac{\tilde{r}_{k+1}}{\|\tilde{r}_{k+1}\|_X}, \quad \tilde{r}_{k+1} = s_k - \sum_{j=0}^k (s_k, r_j)_X r_j. \quad (4.2.29)$$

Straightforward computations confirm that  $r_{k+1} \perp \mathcal{P}_k$ .

The sum in (4.2.29) collapses to two terms! In fact, since  $(r_k, q)_X = 0$  for all  $q \in \mathcal{P}_{k-1}$ , by Ass. 4.2.23

$$(s_k, r_j)_X = (\{t \mapsto tr_k(t)\}, r_j)_X \stackrel{(4.2.22)}{=} (r_k, \{t \mapsto tr_j\})_X = 0, \quad \text{if } j < k-1,$$

because in this case  $\{t \mapsto tr_j\} \in \mathcal{P}_{k-1}$ . As a consequence (4.2.29) reduces to the **3-term recursion**

$$\begin{aligned} r_{k+1} &= \pm \frac{\tilde{r}_{k+1}}{\|\tilde{r}_{k+1}\|_X}, \\ \tilde{r}_{k+1} &= s_k - (\{t \mapsto tr_k\}, r_k)_X r_k - (\{t \mapsto tr_k\}, r_{k-1})_X r_{k-1} \end{aligned}, \quad k = 1, \dots, m-1. \quad (4.2.30)$$

The recursion starts with  $r_{-1} := 0$ ,  $r_0 = \{t \mapsto 1/\|1\|_X\}$ .

---

The 3-term recursion (4.2.30) can be recast in various ways. Forgoing normalization the next theorem presents one of them.

**Theorem 4.2.31. 3-term recursion for orthogonal polynomials**

Given an inner product  $(\cdot, \cdot)_X$  on  $\mathcal{P}_m$ ,  $m \in \mathbb{N}$ , define  $p_{-1} := 0$ ,  $p_0 = 1$ , and

$$p_{k+1}(t) := (t - \alpha_{k+1})p_k(t) - \beta_k p_{k-1}(t), \quad k = 0, 1, \dots, m-1, \\ \text{with } \alpha_{k+1} := \frac{(\{t \mapsto tp_k(t)\}, p_k)_X}{\|p_k\|_X^2}, \quad \beta_k := \frac{\|p_k\|_X^2}{\|p_{k-1}\|_X^2}. \quad (4.2.32)$$

Then  $p_k \in \mathcal{P}_k$  with leading coefficient = 1, and  $\{p_0, p_1, \dots, p_m\}$  is an **orthogonal basis** of  $\mathcal{P}_m$ .

*Proof.* (by rather straightforward induction) We first confirm, thanks to the definition of  $\alpha_1$ ,

$$(p_0, p_1)_X = (p_0, \{t \mapsto (t - \alpha_1)p_0(t)\})_X = (p_0, \{t \mapsto tp_0(t)\})_X - \alpha_1(p_0, p_0)_X = 0.$$

For the induction step we assume that the assertion is true for  $p_0, \dots, p_k$  and observe that for  $p_{k+1}$  according to (4.2.32) we have

$$\begin{aligned} (p_k, p_{k+1})_X &= (p_k, \{t \mapsto (t - \alpha_{k+1})p_k(t)\} - \beta_k p_{k-1})_X \\ &= (p_k, \{t \mapsto tp_k(t)\})_X - \underbrace{\alpha_{k+1}(p_k, p_k)_X}_{=\{t \mapsto tp_k(t)\}, p_k)_X} - \underbrace{\beta_k(p_k, p_{k-1})_X}_{=0} = 0, \\ (p_{k-1}, p_{k+1})_X &= (p_{k-1}, \{t \mapsto (t - \alpha_{k+1})p_k(t)\} - \beta_k p_{k-1})_X \\ &= (p_{k-1}, \{t \mapsto tp_k(t)\})_X - \alpha_{k+1}(p_{k-1}, p_k)_X - \beta_k(p_k, p_{k-1})_X = 0, \\ (p_\ell, p_{k+1})_X &= (p_\ell, \{t \mapsto (t - \alpha_{k+1})p_k(t)\} - \beta_k p_{k-1})_X \\ &= \underbrace{(p_\ell, \{t \mapsto tp_k(t)\})_X}_{=0} - \underbrace{\alpha_{k+1}(p_\ell, p_k)_X}_{0} - \underbrace{\beta_k(p_\ell, p_{k-1})_X}_{0} = 0, \quad \ell = 0, \dots, k-2. \end{aligned}$$

This amounts to the assertion of orthogonality for  $k+1$ . Above, several inner products vanish because of the induction hypothesis!  $\square$

**(4.2.33) Discrete orthogonal polynomials**

Since they involve integrals, weighted  $L^2$ -inner products (4.2.20) are not accessible computationally, unless one resigns to approximation, see Chapter 5 for corresponding theory and techniques.

Therefore, given a point set  $\mathcal{T} := \{t_0, t_1, \dots, t_n\}$ , we focus on the associated **discrete  $L^2$ -inner product**

$$(f, g)_X := (f, g)_{\mathcal{T}} := \sum_{j=0}^n f(t_j) \overline{g(t_j)}, \quad f, g \in C^0([a, b]), \quad (4.2.21)$$

which is positive definite on  $\mathcal{P}_n$  and satisfies Ass. 4.2.23.

The polynomials  $p_k$  generated by the 3-term recursion (4.2.32) from Thm. 4.2.31 are then called **discrete orthogonal polynomials**. The following MATLAB code computes the recursion coefficients  $\alpha_k$  and  $\beta_k$ ,  $k = 1, \dots, n-1$ .

**MATLAB-code 4.2.34: Computation of weights in 3-term recursion for discrete orhtogonal polynomials**

```

1 function [alpha, beta] = coefforth(t,n)
2 % Vector t passes the points in the definition of the discrete  $L^2$ -inner
3 % product, n the maximal index desired
4 m = numel(t); % Maximal degree of orthogonal polynomial
5 alpha(1) = sum(t)/m;
6 % Initialization of recursion; we store only the values of
7 % the polynomials at the points in  $\mathcal{T}$ .
8 p1 = ones(size(t));
9 p2 = t-alpha(1);
10 % Main loop
11 for k=1:min(n-1,m-2)
12     p0 = p1; p1 = p2;
13     % 3-term recursion (4.2.32),
14     alpha(k+1) = dot(p1, (t.*p1))/norm(p1)^2;
15     beta(k) = (norm(p1)/norm(p0))^2;
16     p2 = (t-alpha(k+1)).*p1-beta(k)*p0;
17 end

```

### (4.2.35) Polynomial fitting

Given a point set  $\mathcal{T} := \{t_0, t_1, \dots, t_n\} \subset [a, b]$ , and a function  $f : [a, b] \rightarrow \mathbb{K}$ , we may seek to approximate  $f$  by its polynomial best approximant with respect to the discrete  $L^2$ -norm  $\|\cdot\|_{\mathcal{T}}$  induced by the discrete  $L^2$ -inner product (4.2.21).

#### Definition 4.2.36. Fitted polynomial

Given a point set  $\mathcal{T} := \{t_0, t_1, \dots, t_n\} \subset [a, b]$ , and a function  $f : [a, b] \rightarrow \mathbb{K}$  we call

$$q_k := \underset{p \in \mathcal{P}_k}{\operatorname{argmin}} \|f - p\|_{\mathcal{T}}, \quad k \in \{0, \dots, n\},$$

the **fitting polynomial** to  $f$  on  $\mathcal{T}$  of degree  $k$ .

The stable and efficient computation of fitting polynomials can rely on combining Thm. 4.2.31 with Cor. 4.2.14:

- ① (Pre-)compute the weights  $\alpha_\ell$  and  $\beta_\ell$  for the 3-term recursion (4.2.32).
- ② (Pre-)compute the values of the orthogonal polynomials  $p_k$  at desired evaluation points  $x_i \in \mathbb{R}$ ,  $i = 1, \dots, N$ .
- ③ Compute the inner product  $(f, p_\ell)_X$ ,  $\ell = 0, \dots, k$ , and use (4.2.15) to linearly combine the vectors  $[p_\ell(x_i)]_{i=1}^N$ ,  $\ell = 0, \dots, k$ .

This yields  $[q(x_i)]_{i=1}^N$ ,  $q \in \mathcal{P}_k$  the fitting polynomial.

#### Example 4.2.37 (Approximation by discrete polynomial fitting)

We use equidistant points  $\mathcal{T} := \{t_k = -1 + k \frac{2}{m}, k = 0, \dots, m\} \subset [-1, 1]$ ,  $m \in \mathbb{N}$  to compute fitting polynomials ( $\rightarrow$  Def. 4.2.36) for two different functions.

We monitor the  $L^2$ -norm and  $L^\infty$ -norm of the approximation error, both norms approximated by sampling in  $\xi_j = -1 + \frac{j}{500}$ ,  $j = 0, \dots, 1000$ .

①  $f(t) = (1 + (5t)^2)^{-1}$ ,  $I = [-1, 1]$   $\rightarrow$  Ex. 4.1.34, analytic in complex neighborhood of  $[-1, 1]$ :

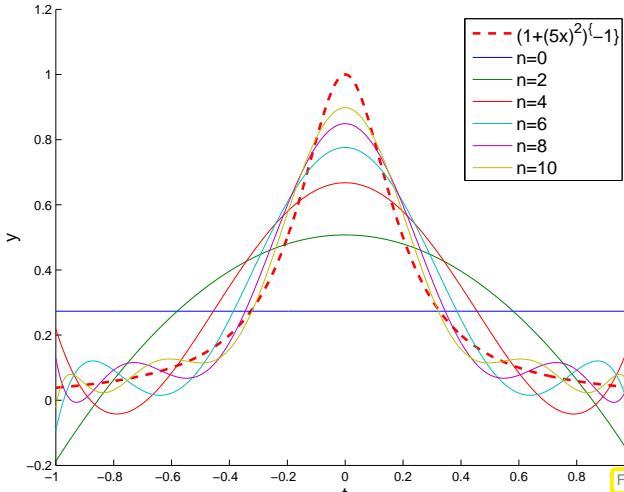
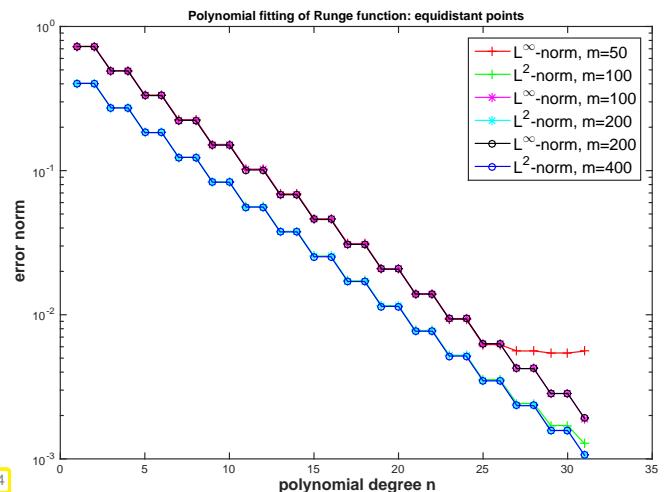


Fig. 143



We observe exponential convergence ( $\rightarrow$  Def. 4.1.31) in the polynomial degree  $n$ .

②  $f(t) = \max\{0, 1 - 2 * |x + \frac{1}{4}| \}$ ,  $f$  only in  $C^0([-1, 1])$ :

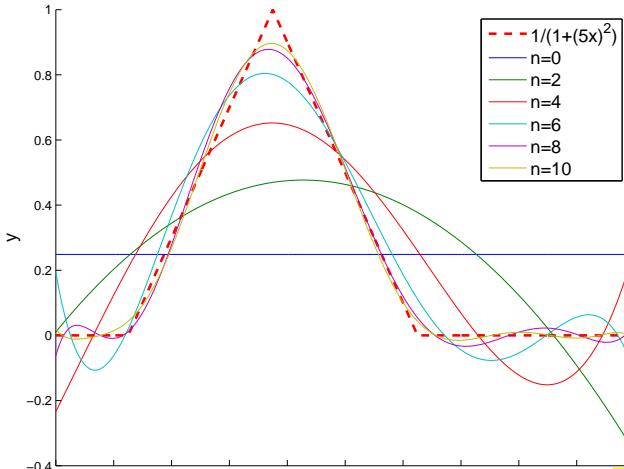
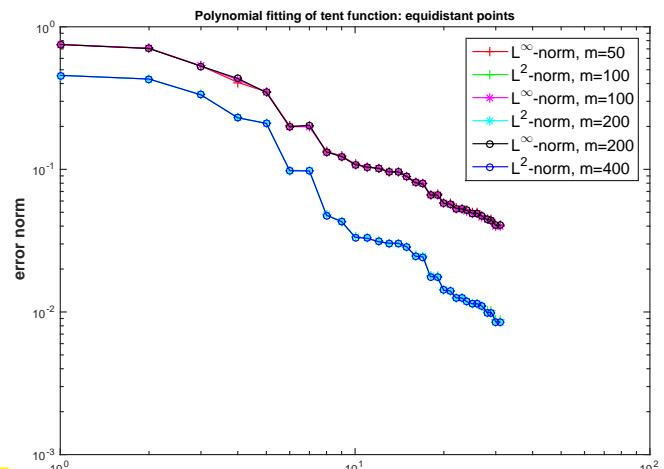


Fig. 145



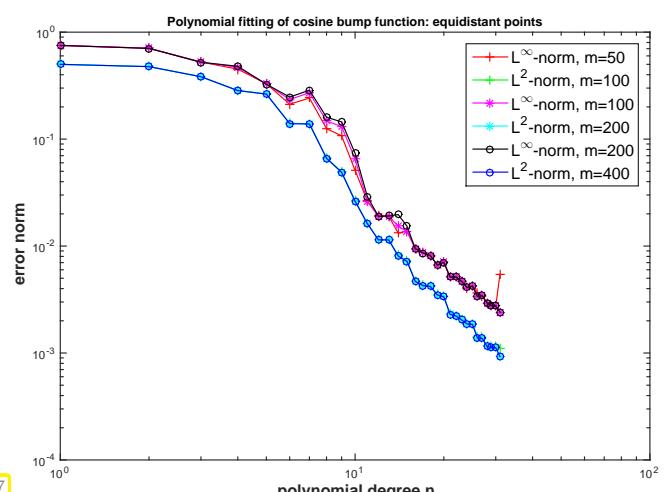
We observe only algebraic convergence ( $\rightarrow$  Def. 4.1.31 in the polynomial degree  $n$  (for  $n \ll m$ !).

③ “bump function”

$$f(t) = \max\{\cos(4\pi|t + \frac{1}{4}|), 0\} .$$

$\Rightarrow$  Merely  $f \in C^1([-1, 1])$

Doubly logarithmic plot suggests “asymptotic” algebraic convergence.



## 4.3 Uniform Best Approximation

### (4.3.1) The alternation theorem

Given an interval  $[a, b]$  we seek a best approximant of a function  $f \in C^0([a, b])$  in the space  $\mathcal{P}_n$  of polynomials of degree  $\leq n$  with respect to the supremum norm  $\|\cdot\|_{L^\infty([a,b])}$ :

$$q \in \operatorname{argmin}_{p \in \mathcal{P}_n} \|f - p\|_{L^\infty(I)}.$$

The results of Section 4.2.1 cannot be applied because the supremum norm is not induced by an inner product on  $\mathcal{P}_n$ .

Theory provides us with surprisingly precise necessary and sufficient conditions to be satisfied by the polynomial  $L^\infty([a, b])$ -best approximant  $q$ .

### Theorem 4.3.2. Chebychev alternation theorem

Given  $f \in C^0[a, b]$ ,  $a < b$ , and a polynomial degree  $n \in \mathbb{N}$ , a polynomial  $q \in \mathcal{P}_n$  satisfies

$$q = \operatorname{argmin}_{p \in \mathcal{P}_n} \|f - p\|_{L^\infty(I)}$$

if and only if there exist  $n + 2$  points  $a \leq \xi_0 < \xi_1 < \dots < \xi_{n+1} \leq b$  such that

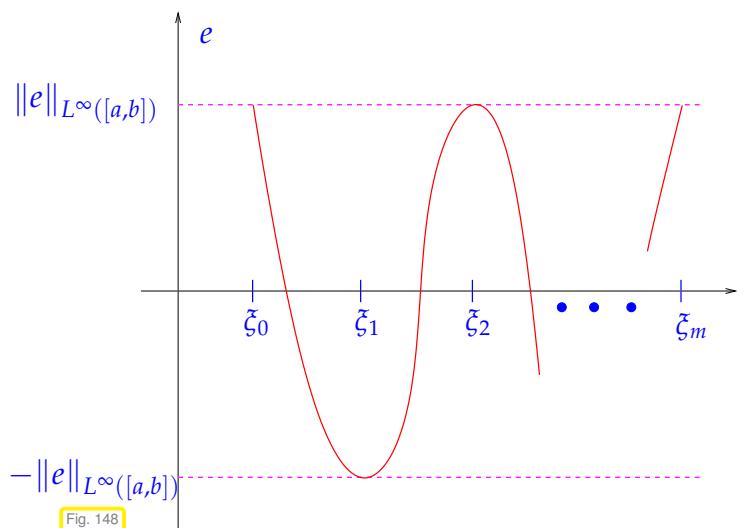
$$\begin{aligned} |e(\xi_j)| &= \|e\|_{L^\infty([a,b])}, \quad j = 0, \dots, n+1, \\ e(\xi_j) &= -e(\xi_{j+1}), \quad j = 0, \dots, n, \end{aligned}$$

where  $e := f - q$  denotes the approximation error.

Visualization of the behavior of the  $L^\infty([a, b])$ -best approximation error  $e := f - q$  according to the Chebychev alternation theorem Thm. 4.3.2.  $\triangleright$

The extrema of the approximation error are sometimes called **alternants**.

Compare with the shape of the Chebychev polynomials  $\rightarrow$  Def. 4.1.67.



### (4.3.3) Remez algorithm

The widely used iterative algorithm (**Remez algorithm**) for finding an  $L^\infty$ -best approximant is motivated by the alternation theorem. The idea is to determine successively better approximations of the set of alternants:  $\mathcal{A}^{(0)} \rightarrow \mathcal{A}^{(1)} \rightarrow \dots$ ,  $\#\mathcal{A}^{(l)} = n+2$ .

Key is the observation that, due to the alternation theorem, the polynomial  $L^\infty([a,b])$ -best approximant  $q$  will satisfy (one of the) **interpolation conditions**

$$q(\xi_k) \pm (-1)^k \delta = f(\xi_k), \quad k = 0, \dots, n+1, \quad \delta := \|f - q\|_{L^\infty([a,b])}. \quad (4.3.4)$$

- ① Initial guess  $\mathcal{A}^{(0)} := \{\xi_0^{(0)} < \xi_1^{(0)} < \dots < \xi_n^{(0)} < \xi_{n+1}^{(0)}\} \subset [a, b]$  “arbitrary”, for instance extremal points of the Chebychev polynomial  $T_{n+1}$ ,  $\rightarrow$  Def. 4.1.67, so so-called **Chebychev alternants**

$$\xi_j^{(0)} = \frac{1}{2}(a+b) + \frac{1}{2}(b-a) \cos\left(\frac{j}{n+1}\pi\right), \quad j = 0, \dots, n+1. \quad (4.3.5)$$

- ② Given approximate alternants  $\mathcal{A}^{(l)} := \{\xi_0^{(l)} < \xi_1^{(l)} < \dots < \xi_n^{(l)} < \xi_{n+1}^{(l)}\} \subset [a, b]$  determine  $q \in \mathcal{P}_n$  and a deviation  $\delta \in \mathbb{R}$  satisfying the extended interpolation condition

$$q(\xi_k^{(l)}) + (-1)^k \delta = f(\xi_k^{(l)}), \quad k = 0, \dots, n+1. \quad (4.3.6)$$

After choosing a basis for  $\mathcal{P}_n$ , this is  $(n+2) \times (n+2)$  linear system of equations, cf. § 3.1.11.

- ③ Choose  $\mathcal{A}^{(l+1)}$  as the set of extremal points of  $f - q$ , truncated in case more than  $n+2$  of these exist.

These extreme can be located approximately by sampling on a fine grid covering  $[a, b]$ . If the derivative of  $f \in C^1([a, b])$  is available, too, then search for zeros of  $(f - p)'$  using the secant method from § 2.3.22.

- ④ If  $\|f - q\|_{L^\infty([a,b])} \leq \text{TOL} \cdot \|d\|_{L^\infty([a,b])}$  STOP, else GOTO ②.  
( $\text{TOL}$  is a prescribed relative tolerance.)

#### MATLAB-code 4.3.7: Remez algorithm for uniform polynomial approximation on an interval

```

1 function c = remes(f,f1,a,b,d,tol)
2 % f is a handle to the function, f1 to its derivative
3 % d = polynomial degree (positive integer)
4 % a,b = interval boundaries
5 % returns coefficients of polynomial in monomial basis
6 % (MATLAB convention, see Rem. 3.2.4).
7
8 n = 8*d; % n = number of sampling points
9 xtab=[a:(b-a)/(n-1):b]'; % Points of sampling grid
10 ftab = feval(f,xtab); % Function values at sampling points
11 fsupn = max(abs(ftab)); % Approximate supremum norm of f
12 f1tab = feval(f1,xtab); % Derivative values at sampling points
13 % The vector xe stores the current guess for the alternants; initial
14 % guess is Chebychev alternants (4.3.5).

```

```

15 h=pi/(d+1); xe=(a+b)/2 + (a-b)/2*cos(h*[0:d+1]');
16 fxe=feval(f,xe);
17
18 maxit = 10;
19 % Main iteration loop of Remez algorithm
20 for k=1:maxit
21 % Interpolation at d+2 points xe with deviations ±δ
22 % Algorithm uses monomial basis, which is not optimal
23 V=vander(xe); A=[V(:,2:d+2), (-1).^(0:d+1)']; % LSE
24 c=A\fxe; % Solve for coefficients of polynomial q
25 c1=[d:-1:1]'.*c(1:d); % Monomial coefficients of derivative q'
26
27 % Find initial guesses for the inner extremes by sampling; track sign
28 % changes of the derivative of the approximation error
29 deltab = (polyval(c1,xtab) - f1tab);
30 s=[deltab(1:n-1)].*[deltab(2:n)];
31 ind=find(s<0); xx0=xtab(ind); % approximate zeros of e'
32 nx = length(ind); % number of approximate zeros
33 % Too few extrema; bail out
34 if (nx < d), error('Too few extrema'); end
35
36 % Secant method to determine zeros of derivative
37 % of approximation error
38 F0 = polyval(c1,xx0) - feval(f1,xx0);
39 % Initial guess from shifted sampling points
40 xx1=xx0+(b-a)/(2*n);
41 F1 = polyval(c1,xx1) - feval(f1,xx1);
42 % Main loop of secant method
43 while min(abs(F1)) > 1e-12,
44 xx2=xx1-F1./(F1-F0).* (xx1-xx0);
45 xx0=xx1; xx1=xx2; F0=F1;
46 F1=polyval(c1,xx1) - feval(f1,xx1);
47 end;
48
49 % Determine new approximation for alternants; store in xe
50 % If too many zeros of the derivative (f - p)'
51 % have been found, select those, where the deviation is maximal.
52 if (nx == d)
53 xe=[a;xx0;b];
54 elseif (nx == d+1)
55 xmin = min(xx0); xmax = max(xx0);
56 if ((xmin - a) > (b-xmax)), xe = [a;xx0];
57 else xe = [xx0;b]; end
58 elseif (nx == d+2)
59 xe = xx0;
60 else
61 fx = feval(f,xx0);
62 del = abs(polyval(c(1:d+1),xx0) - fx);
63 [dummy,ind] = sort(del);
64 xe = xx0(ind(end-d-1:end));
65 end
66
67 % Deviation in sampling points and approximate alternants

```

```

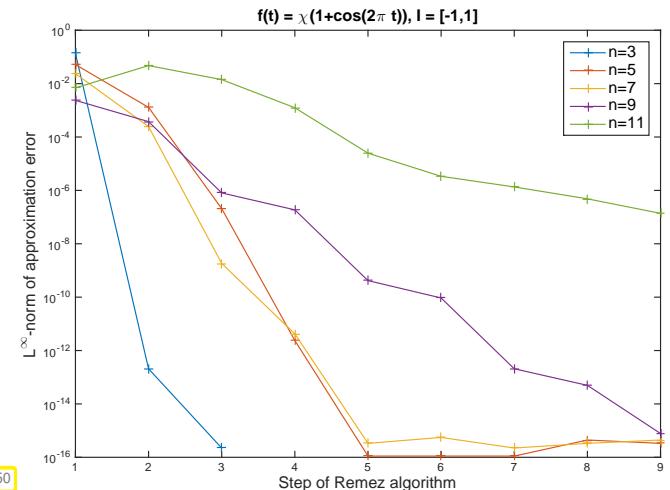
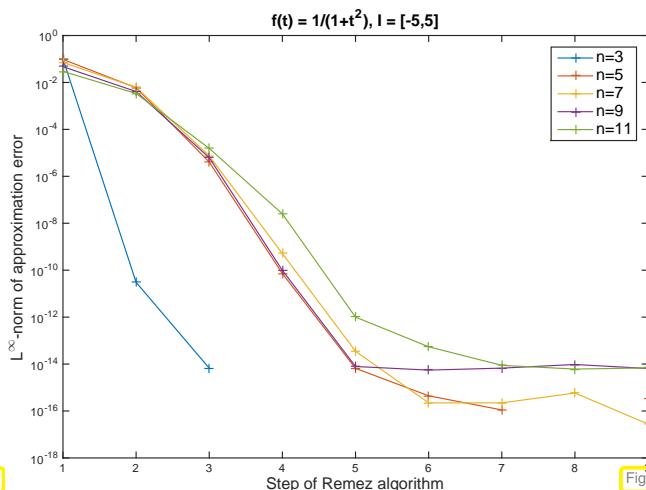
68 fxe=feval(f,xe);
69 del = [ polyval(c(1:d+1),a) - ftab(1); ...
70      polyval(c(1:d+1),xe) - fxe; ...
71      polyval(c(1:d+1),b) - ftab(end) ];
72 % Approximation of supremum norm of approximation error
73 dev = max(abs(del));
74 % Termination of Remez iteration
75 if ( dev < tol*fsupn), break; end
76 end;
77 c = c(1:d+1);

```

### Experiment 4.3.8 (Convergence of Remez algorithm)

We examine the convergence of the Remez algorithm from Code 4.3.7 for two different functions:

- $f(t) = (1+t^2)^{-1}$ ,  $I = [-5, 5] \rightarrow$  Bsp. 4.1.34
- $f(t) = \begin{cases} \frac{1}{2}(1 + \cos(2\pi t)) & , \text{ falls } |t| < \frac{1}{2}, \\ 0 & \text{sonst.} \end{cases}$



Convergence in both cases; faster convergence observed for smooth function, for which machine precision is reached after a few steps.

## 4.4 Trigonometric interpolation

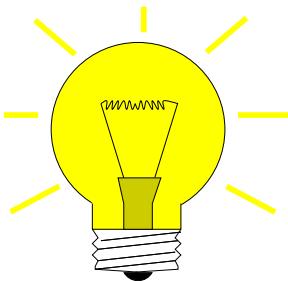


*Supplementary reading.* The topic is presented in [15, Sect. 8.5].

Task: approximation of a continuous **1-periodic** function

$$f \in C^0(\mathbb{R}) \quad , \quad f(t+1) = f(t) \quad \forall t \in \mathbb{R} .$$

Dubious: approximation by global polynomials, because they are not 1-periodic  
 ➤ the global polynomial Lagrange interpolant will lack an essential structural property.



Idea: approximation by interpolation based on vector spaces of 1-periodic functions

Natural candidates: **trigonometric functions** (sines and cosines)

### Trigonometric interpolation [15, pp. 304]

Given nodes  $t_0 < t_1 < \dots < t_{2n}$ ,  $t_k \in [0, 1[$ , and values  $y_k \in \mathbb{R}$ ,  $k = 0, \dots, 2n$  find

$$q \in \mathcal{P}_{2n}^T := \text{Span}\{t \mapsto \cos(2\pi jt), t \mapsto \sin(2\pi jt)\}_{j=0}^n, \quad (4.4.2)$$

$$\text{with } q(t_k) = y_k \text{ for all } k = 0, \dots, 2n. \quad (4.4.3)$$

Terminology:  $\mathcal{P}_{2n}^T \doteq$  space of **trigonometric polynomials** of degree  $2n$ .

Trigonometric interpolation = linear interpolation mapping ( $\rightarrow$  § 3.1.11) into function space  $\mathcal{P}_{2n}^T$

#### (4.4.4) Trigonometric polynomials

- \*  $\dim \mathcal{P}_{2n}^T = 2n + 1$ , because functions are linearly independent, sine dropped for  $j = 0$
- \* The functions defined in (4.4.2) provide basis functions for interpolation, cf. (3.1.7).

Recall expressions for trigonometric functions via **complex** exponentials:

$$e^{it} = \cos t + i \sin t \Rightarrow \begin{cases} \cos t = \frac{1}{2}(e^{it} + e^{-it}) \\ \sin t = \frac{1}{2i}(e^{it} - e^{-it}) \end{cases}. \quad (4.4.5)$$

In the spirit of § 3.1.11 start from a basis representation of the interpolant  $q \in \mathcal{P}_{2n}^T$  with unknown coefficients  $\alpha_j, \beta_j$ :

$$q(t) = \alpha_0 + \sum_{j=1}^n \alpha_j \cos(2\pi jt) + \beta_j \sin(2\pi jt), \quad \alpha_j, \beta_j \in \mathbb{R}. \quad (4.4.6)$$

Then (3.1.13) will immediately give us the corresponding linear system of equations. Its structure can be revealed more clearly, if we make use of (4.4.5) and switch to the representation of trigonometric polynomials by means of complex exponentials

$$\begin{aligned} q(t) &= \alpha_0 + \sum_{j=1}^n \alpha_j \cos(2\pi jt) + \beta_j \sin(2\pi jt) \\ &= \alpha_0 + \frac{1}{2} \left\{ \sum_{j=1}^n (\alpha_j - i\beta_j) e^{2\pi jt} + (\alpha_j + i\beta_j) e^{-2\pi jt} \right\} \\ &= \alpha_0 + \frac{1}{2} \sum_{j=-n}^{-1} (\alpha_{-j} + i\beta_{-j}) e^{2\pi jt} + \frac{1}{2} \sum_{j=1}^n (\alpha_j - i\beta_j) e^{2\pi jt} \end{aligned}$$

$$= e^{-2\pi int} \sum_{j=0}^{2n} \gamma_j e^{2\pi i jt}, \quad \text{with} \quad \gamma_j = \begin{cases} \frac{1}{2}(\alpha_{n-j} + i\beta_{n-j}) & \text{for } j = 0, \dots, n-1, \\ \alpha_0 & \text{for } j = n, \\ \frac{1}{2}(\alpha_{j-n} - i\beta_{j-n}) & \text{for } j = n+1, \dots, 2n. \end{cases} \quad (4.4.7)$$

Note:  $\gamma_j \in \mathbb{C}$   $\Rightarrow$  work in  $\mathbb{C}$  in the context of trigonometric interpolation! Admit  $y_k \in \mathbb{C}$ .

From the above manipulations we observe

$$q \in \mathcal{P}_{2n+1}^T \Rightarrow q(t) = e^{-2\pi int} \cdot p(e^{2\pi it}) \quad \text{with} \quad p(z) = \sum_{j=0}^{2n} \gamma_j z^j \in \mathcal{P}_{2n}, \quad \text{a polynomial!}$$

and  $\gamma_j$  from (4.4.7).

$t \rightarrow \exp(2\pi int)q(t)$  is a polynomial  $p \in \mathcal{P}_{2n}$  restricted to the unit circle  $S^1$  in  $\mathbb{C}$ .

☞ notation:  $S^1 := \{z \in \mathbb{C}: |z| = 1\}$  is the unit circle in the complex plane.

The relationship (4.4.8) justifies calling  $\mathcal{P}_{2n}^T$  a space of trigonometric *polynomials*!

#### (4.4.9) Trigonometric interpolation and Lagrange interpolation

Next we relate trigonometric interpolation to approximation by polynomial Lagrange interpolation according to Def. 4.1.25. In fact, we slightly extend the definition, because now we admit *complex interpolation nodes*.

The key tool is a smooth bijective mapping between  $I := [0, 1[$  and  $S^1$  defined as

$$\Phi_{S^1} : I \rightarrow S^1, \quad t \mapsto z := \exp(2\pi it). \quad (4.4.10)$$

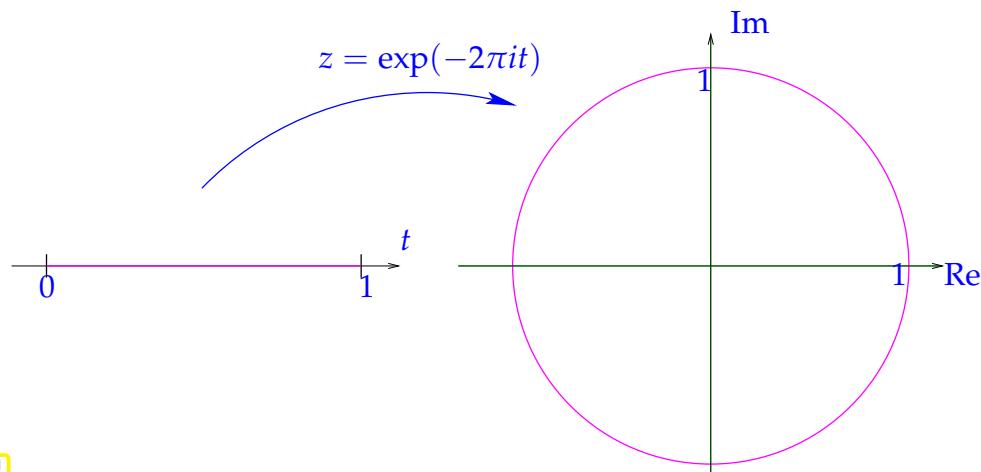


Fig. 151

Trigonometric interpolation through data points $(t_k, y_k)$	$\iff$	Polynomial interpolation through data points $(e^{2\pi it_k}, y_k)$
Trigonometric interpolation of $f$ on $I$	$\iff$	Polynomial interpolation of pullback $(\Phi_{S^1}^{-1})^* f$ on $S^1$

Here we deal with a non-affine pullback, but the definition is the same as the one given in (4.1.16) for an affine pullback:

$$(\Phi_{S^1}^{-1})^* : C^0([0, 1[) \rightarrow C^0(S^1), \quad ((\Phi_{S^1}^{-1})^* f)(z) := f(\Phi_{S^1}^{-1}(z)), \quad z \in S^1. \quad (4.4.11)$$

Trigonometric interpolation = polynomial interpolation on  $S^1$

- All theoretical results and algorithms from polynomial interpolation carry over to trigonometric interpolation
- \* Existence and uniqueness of trigonometric interpolation polynomial, see Thm. 3.2.14,
- \* Concept of Lagrange polynomials, see (3.2.11),
- \* the algorithms and representations discussed in Section 3.2.3.

#### MATLAB-code 4.4.12: Evaluation of trigonometric interpolation polynomial in many points

```

1 function q = trigpolyval(t,y,x)
2 % Evaluation of trigonometric interpolant at numerous points
3 % t: row vector of nodes  $t_0, \dots, t_n \in [0,1]$ 
4 % y: row vector of data  $y_0, \dots, y_n$ 
5 % x: row vector of evaluation points  $x_1, \dots, x_N$ 
6 N = length(y); if (mod(N,2) ~= 1), error('#pts odd required'); end
7 n = (N-1)/2;
8 tc = exp(2*pi*i*t); % Interpolation nodes on unit circle
9 z = exp(2*pi*i*n*t).*y; % Rescaled values, according to
  q(t) = e^{-2\pi i nt} \cdot p(e^{2\pi i t})
10 % Evaluation of interpolating polynomial on unit circle, see
  % Code 3.2.28
11 p = intpolyval(tc,z,exp(2*pi*i*x));
12 q = exp(-2*pi*i*n*x).*p; % Undo the scaling, see (4.4.8)

```

#### (4.4.13) Algorithms for equidistant trigonometric interpolation

Remember: additional freedom to choose interpolation nodes in the setting of function approximation

Reasonable choice for generic 1-periodic  $f$ : uniformly distributed nodes  $t_k = \frac{k}{2n+1}$ ,  $k = 0, \dots, 2n$

Justification: the circle is utterly “isotropic”!

- $(2n+1) \times (2n+1)$  linear system of equations:

$$\sum_{j=0}^{2n} \gamma_j \exp\left(2\pi i \frac{jk}{2n+1}\right) = z_k := \exp\left(2\pi i \frac{nk}{2n+1}\right) y_k, \quad k = 0, \dots, 2n.$$

$\Updownarrow$

$$\bar{\mathbf{F}}_{2n+1} \mathbf{c} = \mathbf{z}, \quad \mathbf{c} = (\gamma_0, \dots, \gamma_{2n})^T \quad \xrightarrow{\text{Lemma 9.2.9}} \quad \mathbf{c} = \frac{1}{2n+1} \mathbf{F}_{2n} \mathbf{z}. \quad (4.4.14)$$

$(2n+1) \times (2n+1)$  (conjugate) Fourier matrix, see (9.2.8)

- Fast solution by means of FFT:  $O(n \log n)$  asymptotic complexity, see Section 9.3

**MATLAB-code 4.4.15: Efficient computation of coefficient of trigonometric interpolation polynomial (equidistant nodes)**

```

1 function [a,b] = trigipequid(y)
2 % Efficient computation of coefficients in expansion (4.4.6) for a
3 % trigonometric
4 % interpolation polynomial in equidistant points  $(\frac{j}{2n+1}, y_j)$ ,  $j = 0, \dots, 2n$ 
5 % y has to be a row vector of odd length, return values are column
6 % vectors
7 N = length(y); if (mod(N, 2) ~= 1), error('#pts odd!'); end;
8 n = (N-1)/2;
9 C = fft(exp(2*pi*i*(n/N)*(0:2*n)).*y)/N; % see (4.4.14)
10 % From (4.4.7):  $\alpha_j = \frac{1}{2}(\gamma_{n-j} + \gamma_{n+j})$  and  $\beta_j = \frac{1}{2i}(\gamma_{n-j} - \gamma_{n+j})$ ,  $j = 1, \dots, n$ ,  $\alpha_0 = \gamma_n$ 
11 a = transpose([c(n+1), c(n:-1:1)+c(n+2:N)]);
12 b = transpose(-i*[c(n:-1:1)-c(n+2:N)]);

```

**MATLAB-code 4.4.16: Computation of coefficients of trigonometric interpolation polynomial, general nodes**

```

1 function [a,b] = trigpolycoeff(t,y)
2 % Computes expansion coefficients of trigonometric polynomials (4.4.6)
3 % t: row vector of nodes  $t_0, \dots, t_n \in [0, 1[$ 
4 % y: row vector of data  $y_0, \dots, y_n$ 
5 % return values are column vectors of expansion coefficients  $\alpha_j$ ,  $\beta_j$ 
6 N = length(y); if (mod(N, 2) ~= 1), error('#pts odd!'); end
7 n = (N-1)/2;
8 M = [cos(2*pi*t'* (0:n)), sin(2*pi*t'* (1:n))];
9 c = M\y';
10 a = c(1:n+1); b = c(n+2:end);

```

**Example 4.4.17 (Runtime comparison for computation of coefficient of trigonometric interpolation polynomials)**

tic-toe-timings

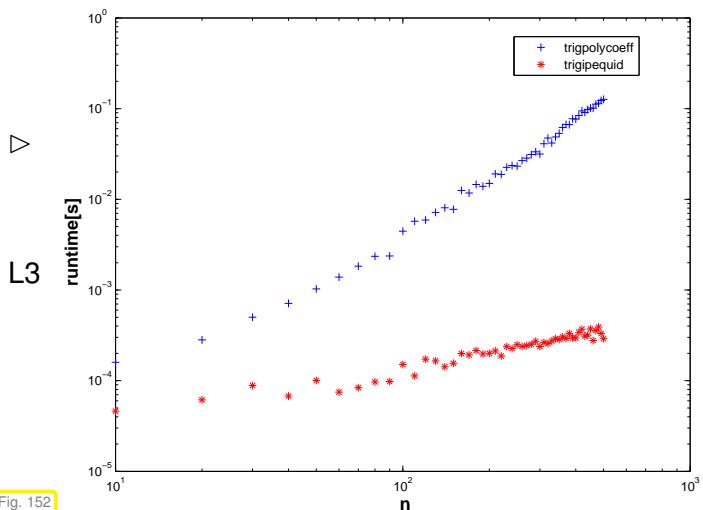


Fig. 152

**MATLAB-code 4.4.18: Runtime comparison**

```

1 function trigipequidtiming

```

```

2 | % Runtime comparison between efficient (→ Code 4.4.15) and direct
   | computation
3 | % (→ Code 4.4.16 of coefficients of trigonoetric interpolation
   | polynomial in
4 | % equidistant points.
5 | Nruns = 3; times = [];
6 | for n = 10:10:500
7 |   disp(n)
8 |   N = 2*n+1; t = 0:1/N:(1-1/N); y = exp(cos(2*pi*t));
9 |   t1 = realmax; t2 = realmax;
10 |   for k=1:Nruns
11 |     tic; [a,b] = trigpolycoeff(t,y); t1 = min(t1, toc);
12 |     tic; [a,b] = trigipequid(y); t2 = min(t2, toc);
13 |   end
14 |   times = [times; n , t1 , t2];
15 | end
16 |
17 | figure; loglog(times(:,1),times(:,2),'b+',...
18 |                      times(:,1),times(:,3),'r*');
19 | xlabel('{\bf n}','fontsize',14);
20 | ylabel('{\bf runtime[s]}','fontsize',14);
21 | legend('trigpolycoeff','trigipequid','location','best');
22 |
23 | print -depsc2 '../PICTURES/trigipequidtiming.eps';

```

Same observation as in Ex. 9.3.1: massive gain in efficiency through relying on FFT.

### Remark 4.4.19 (Efficient evaluation of trigonometric interpolation polynomials)

Task: evaluation of trigonometric polynomial (4.4.6) at *equidistant* points  $\frac{k}{N}$ ,  $N > 2n$ .  $k = 0, \dots, N-1$ .

$$(4.4.7) \quad \Rightarrow \quad q(k/N) = e^{-2\pi i k/N} \sum_{j=0}^{2n} \gamma_j \exp(2\pi i \frac{kj}{N}), \quad k = 0, \dots, N-1. \\ \Rightarrow \quad q(k/N) = e^{-2\pi i k n/N} v_j \quad \text{with} \quad \mathbf{v} = \overline{\mathbf{F}}_N \tilde{\mathbf{c}}, \quad (4.4.20)$$

Fourier matrix, see (9.2.8).

where  $\tilde{\mathbf{c}} \in \mathbb{C}^N$  is obtained by *zero padding* of  $\mathbf{c} := (\gamma_0, \dots, \gamma_{2n})^T$ :

$$(\tilde{\mathbf{c}})_k = \begin{cases} \gamma_j & , \text{for } k = 0, \dots, 2n, \\ 0 & , \text{for } k = 2n+1, \dots, N-1. \end{cases}$$

### MATLAB-code 4.4.21: Fast evaluation of trigonometric polynomial at *equidistant* points

```

1 | function q = trigipequidcomp(a,b,N)
2 | % Efficient evaluation of trigonometric polynomial at equidistant
   | points
3 | % column vectors a and b pass coefficients  $\alpha_j$ ,  $\beta_j$  in

```

```

4 % representation (4.4.6)
5 n = length(a)-1; if (N < (2*n-1)), error('N too small'); end;
6 gamma = transpose(0.5*[a(end):-1:2]+i*b(end:-1:1); ...
7 2*a(1);a(2:end)-i*b(1:end));
8 ch = [gamma, zeros(1,N-(2*n+1))]; % zero padding
9 v = conj(fft(conj(ch))); % Multiplication with conjugate Fourier
matrix
10 q = exp(-2*pi*i*n*(0:N-1)/N).*v; % undo rescaling

```

### MATLAB-code 4.4.22: Equidistant points: fast on the fly evaluation of trigonometric interpolation polynomial

```

1 function q = trigpolyvalequid(y,N)
2 % Evaluation of trigonometric interpolation polynomial through  $(\frac{j}{2n+1}, y_j)$ ,
3 % in equidistant points  $\frac{k}{N}$ ,  $k = 0, N - 1$ 
4 % y has to be a row vector of odd length, values returned as rwo
vector, too.
5 N = length(y); if (mod(N, 2) ~= 1), error('#pts odd!'); end;
6 n = (N-1)/2;
7 % Compute coefficients  $\gamma_j$  in (4.4.7), see (4.4.14)
8 c = fft(exp(2*pi*i*(n/N)*(0:2*n)).*y)/N;
9 ch = [gamma, zeros(1,N-(2*n+1))]; % zero padding
10 v = conj(fft(conj(ch))); % Evaluate multiplication with conjugate
Fourier matrix
11 q = exp(-2*pi*i*n*(0:N-1)/N).*v; % undo rescaling

```

### Experiment 4.4.23 (Interpolation error: trigonometric interpolation)

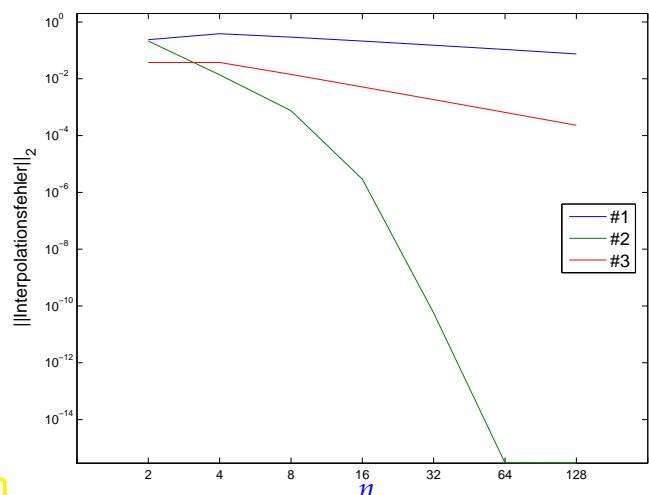
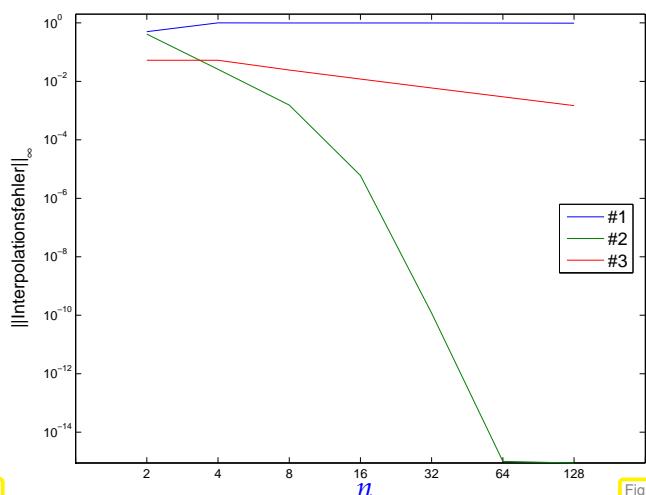
Now we study the asymptotic behavior of the error of equidistant trigonometric interpolation as  $n \rightarrow \infty$  in a numerical experiment for functions with different smoothness properties.

#1 Step function:  $f(t) = 0$  for  $|t - \frac{1}{2}| > \frac{1}{4}$ ,  $f(t) = 1$  for  $|t - \frac{1}{2}| \leq \frac{1}{4}$

#2  $C^\infty$  periodic function:  $f(t) = \frac{1}{\sqrt{1 + \frac{1}{2}\sin(2\pi t)}}$ .

#3 “wedge function”:  $f(t) = |t - \frac{1}{2}|$

Approximate computation of norms of interpolation errors on equidistant grid with 4096 points.



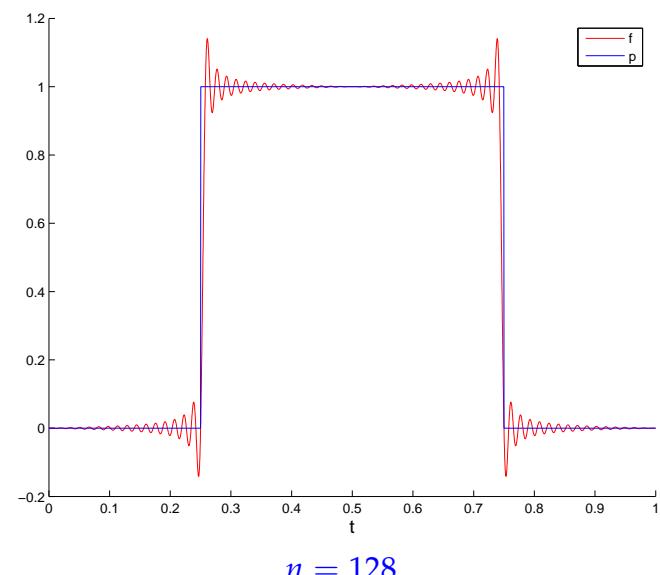
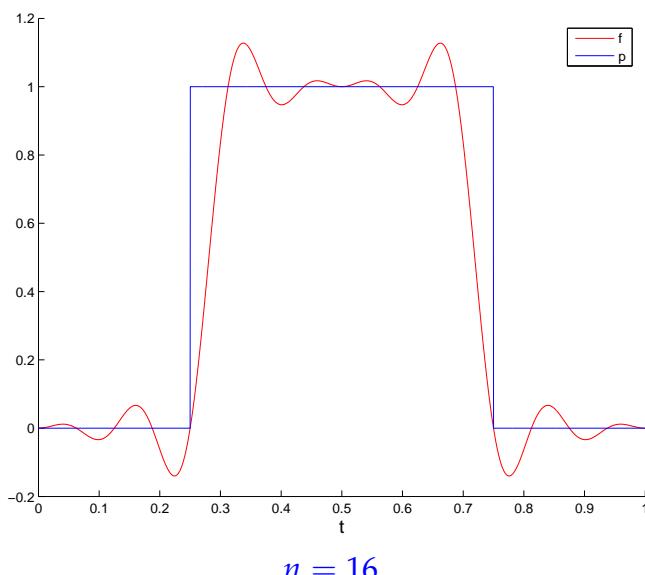
Observation: Function #1: no convergence in  $L^\infty$ -norm, algebraic convergence in  $L^2$ -norm

Function #3: algebraic convergence in both norms

Function #2: exponential convergence in both norms

#### Experiment 4.4.24 (Gibbs phenomenon)

Trigonometric interpolants of step function fail to converge in  $L^\infty$ -norm in Exp. 4.4.23. A closer inspection of the interpolants shows why:



Observation: overshooting in neighborhood of discontinuity: Gibbs phenomenon

#### Experiment 4.4.25 (Trigonometric interpolation of analytic functions)

We study the convergence of equidistant trigonometric interpolation for the interpoland

$$f(t) = \frac{1}{\sqrt{1 - \alpha \sin(2\pi t)}} \quad \text{on } I = [0, 1]. \quad (4.4.26)$$

Approximative computations of error norms by “oversampling” in 4096 points. ▷

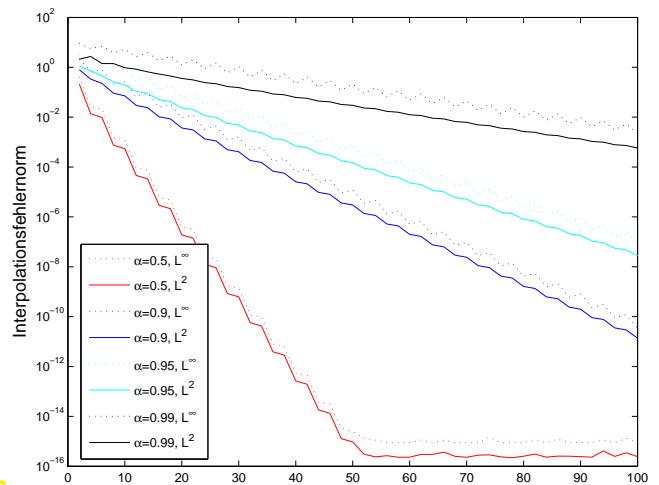


Fig. 155

➤ Observation: exponential convergence in  $n$ , faster for smaller  $\alpha$

#### (4.4.27) Convergence of trigonometric interpolation for analytic interpolants

Explanation of the observation made in Exp. 4.4.25, cf. Rem. 4.1.87:

Similar to Chebychev interpolants, also trigonometric interpolants converge exponentially fast, if the interpoland  $f$  is 1-periodic *analytic* ( $\rightarrow$  Def. 4.1.55) in a strip around the real axis in  $\mathbb{C}$ . The speed of convergence depends on the width of the strip.

Note that  $f$  from (4.4.26) is holomorphic, where

$$1 + \alpha \sin(2\pi z) \notin \mathbb{R}_0^- \Leftrightarrow \sin(2\pi z) = \sin(2\pi x) \cosh(2\pi y) + i \cos(2\pi x) \sinh(2\pi y) \notin ]-\infty, -1 - \frac{1}{\alpha}] ,$$

because the square root is holomorphic in  $\mathbb{C} \setminus \mathbb{R}_0^-$ .

Domain of analyticity of  $f$ :

$$\mathbb{C} \setminus \bigcup_{k \in \mathbb{Z}} \left( \frac{k}{2} + \frac{1}{4} + i(\mathbb{R} \setminus ]-\zeta, \zeta[) \right), \quad \zeta \in \mathbb{R}^+, \quad \cosh(2\pi\zeta) = 1 + \frac{1}{\alpha} .$$

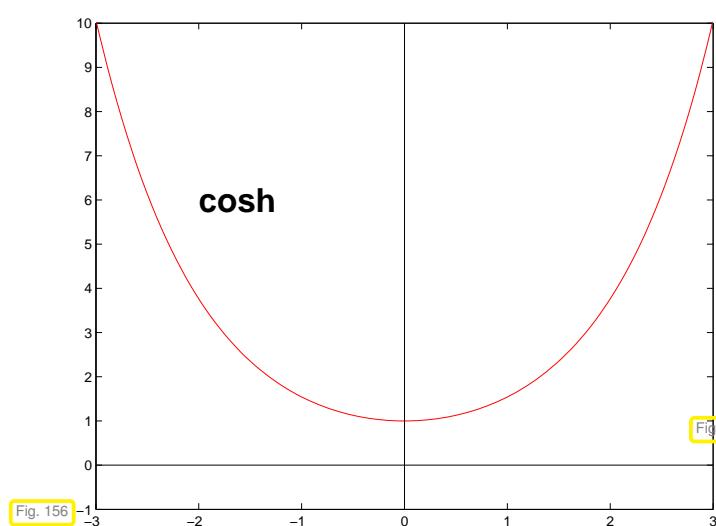
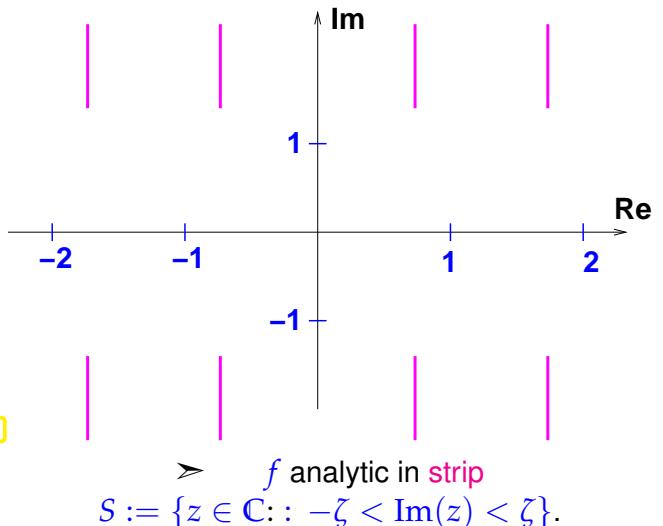


Fig. 156



➤  $f$  analytic in strip  
 $S := \{z \in \mathbb{C} : -\zeta < \text{Im}(z) < \zeta\}$ .

➤ As  $\alpha$  decreases the strip of analyticity becomes wider, since  $x \rightarrow \cosh(x)$  is increasing for  $x > 0$ .

## 4.5 Approximation by piecewise polynomials

Recall some alternatives to interpolation by global polynomials discussed in Chapter 3:

- piecewise linear/quadratic interpolation → Section 3.3.2, Ex. 3.3.11,
  - cubic Hermite interpolation → Section 3.4,
  - (cubic) spline interpolation → Section 3.5.1.
- ☞ All these interpolation schemes rely on **piecewise polynomials** (of different global smoothness)

Focus in this section: function approximation by *piecewise polynomial* interpolants

### (4.5.1) Grid/mesh

The attribute “piecewise” refers to *partitioning of the interval* on which we aim to approximate. In the case of data interpolation the natural choice was to use intervals defined by interpolation nodes. Yet we already saw exceptions in the case of shape-preserving interpolation by means of quadratic splines, see Section 3.5.3.

In the case of function approximation based on an interpolation scheme the additional freedom to choose the interpolation nodes suggests that those be decoupled from the partitioning.



Idea: use **piecewise polynomials** with respect to a **grid/mesh**

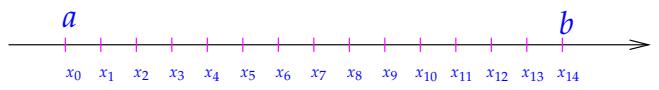
$$\mathcal{M} := \{a = x_0 < x_1 < \dots < x_{m-1} < x_m = b\} \quad (4.5.2)$$

to approximate function  $f : [a, b] \mapsto \mathbb{R}$ ,  $a < b$ .

Borrowing from terminology for splines, cf. Def. 3.5.1, the underlying mesh for piecewise polynomial approximation is sometimes called the “knot set”.

Terminology:

- $x_j \hat{=} \text{nodes}$  of the mesh  $\mathcal{M}$ ,
- $[x_{j-1}, x_j] \hat{=} \text{intervals/cells}$  of the mesh,
- $h_{\mathcal{M}} := \max_j |x_j - x_{j-1}| \hat{=} \text{mesh width}$ ,
- If  $x_j = a + jh \hat{=} \text{equidistant}$  (uniform) mesh with meshwidth  $h > 0$



### Remark 4.5.3 (Local approximation by piecewise polynomials)

We will see that most approximation schemes relying on piecewise polynomials are local in the sense that finding the approximant on a cell of the mesh relies only on a fixed number of function evaluations in a neighborhood of the cell.

- $\mathcal{O}(1)$  computational effort to find interpolant on  $[x_{j-1}, x_j]$  (independent of  $m$ )
- $\mathcal{O}(m)$  computational effort to determine piecewise polynomial approximant for  $m \rightarrow \infty$  (“fine meshes”)

Contrast this with the computational cost of computing global polynomial interpolants, which will usually be  $\mathcal{O}(n^2)$  for polynomial degree  $n \rightarrow \infty$ .

### 4.5.1 Piecewise polynomial Lagrange interpolation

Given: interval  $[a, b] \subset \mathbb{R}$  endowed with mesh  $\mathcal{M} := \{a = x_0 < x_1 < \dots < x_{m-1} < x_m = b\}$ .

Recall theory of polynomial interpolation → Section 3.2.2:  $n+1$  data points needed to fix interpolating polynomial, see Thm. 3.2.14.

► Approach to *local* Lagrange interpolation (→ (3.2.9)) of  $f \in C(I)$  on mesh  $\mathcal{M}$

#### General local Lagrange interpolation on a mesh

- ① Choose *local degree*  $n_j \in \mathbb{N}_0$  for each cell of the mesh,  $j = 1, \dots, m$ .
- ② Choose set of *local* interpolation points

$$\mathcal{T}^j := \{t_0^j, \dots, t_{n_j}^j\} \subset I_j := [x_{j-1}, x_j], \quad j = 1, \dots, m,$$

for each mesh cell/grid interval  $I_j$ .

- ③ Define *piecewise polynomial* interpolant  $s : [x_0, x_m] \rightarrow \mathbb{K}$ :

$$s_j := s|_{I_j} \in \mathcal{P}_{n_j} \quad \text{and} \quad s_j(t_i^j) = f(t_i^j) \quad i = 0, \dots, n_j, \quad j = 1, \dots, m. \quad (4.5.5)$$

Owing to Thm. 3.2.14,  $s_j$  is well defined.

#### Corollary 4.5.6. Piecewise polynomials Lagrange interpolation operator

The mapping  $f \mapsto s$  defines a *linear* operator  $\mathbf{l}_{\mathcal{M}} : C^0([a, b]) \mapsto C_{\mathcal{M}, \text{pw}}^0([a, b])$  in the sense of Def. 3.1.15.

Obviously,  $\mathbf{l}_{\mathcal{M}}$  depends on  $\mathcal{M}$ , the local degrees  $n_j$ , and the sets  $\mathcal{T}_j$  of local interpolation points (the latter two are suppressed in notation).

#### Corollary 4.5.7. Continuous local Lagrange interpolants

If the local degrees  $n_j$  are at least 1 and the local interpolation nodes  $t_k^j, j = 1, \dots, m, k = 0, \dots, n_j$ , for local Lagrange interpolation satisfy

$$t_{n_j}^j = t_0^{j+1} \quad \forall j = 1, \dots, m-1 \quad \Rightarrow \quad s \in C^0([a, b]), \quad (4.5.8)$$

then the piecewise polynomial Lagrange interpolant according to (4.5.5) is *continuous* on  $[a, b]$ :  $s \in C^0([a, b])$ .

Focus:

*asymptotic* behavior of (some norm of) interpolation error

$$\|f - \mathbf{l}_{\mathcal{M}}f\| \leq CT(N) \quad \text{for} \quad N \rightarrow \infty, \quad (4.5.9)$$

where  $N := \sum_{j=1}^m (n_j + 1)$ .

The decay of the bound  $T(N)$  will characterize the type of convergence:

- ☛ algebraic convergence or exponential convergence, see Section 4.1.2, Def. 4.1.31.

But why do we choose this strange number  $N$  as parameter when investigating the approximation error?

Because, by Thm. 3.2.2, it agrees with the dimension of the space of discontinuous, piecewise polynomials functions

$$\{q : [a, b] \rightarrow \mathbb{R} : q|_{I_j} \in \mathcal{P}_{n_j} \forall j = 1, \dots, m\} \quad !$$

This dimension tells us the number of real parameters we need to describe the interpolant  $s$ , that is, the “information cost” of  $s$ .  $N$  is also proportional to the number of interpolation conditions, which agrees with the number of  $f$ -evaluations needed to compute  $s$  (why only proportional in general?).

Special case: uniform polynomial degree  $n_j = n$  for all  $j = 1, \dots, m$ .

► Then we may aim for estimates  $\|f - I_M f\| \leq CT(h_M)$  for  $h_M \rightarrow 0$

in terms of meshwidth  $h_M$ .

Terminology: investigations of this kind are called the study of  **$h$ -convergence**.

### Example 4.5.10 ( $h$ -convergence of piecewise polynomial interpolation)

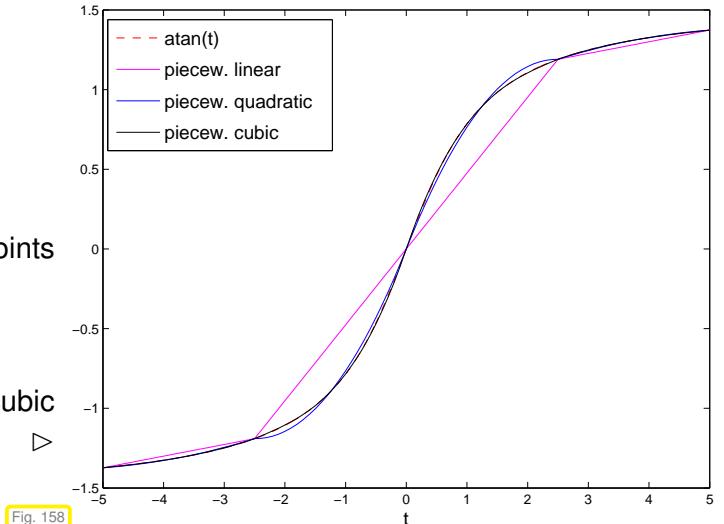
Compare Exp. 3.3.7:

$$f(t) = \arctan t, I = [-5, 5]$$

$$\text{Grid } \mathcal{M} := \{-5, -\frac{5}{2}, 0, \frac{5}{2}, 5\}$$

Local interpolation nodes equidistant in  $I_j$ , endpoints included, (4.5.8) satisfied.

Plots of the piecewise linear, quadratic and cubic polynomial interpolants

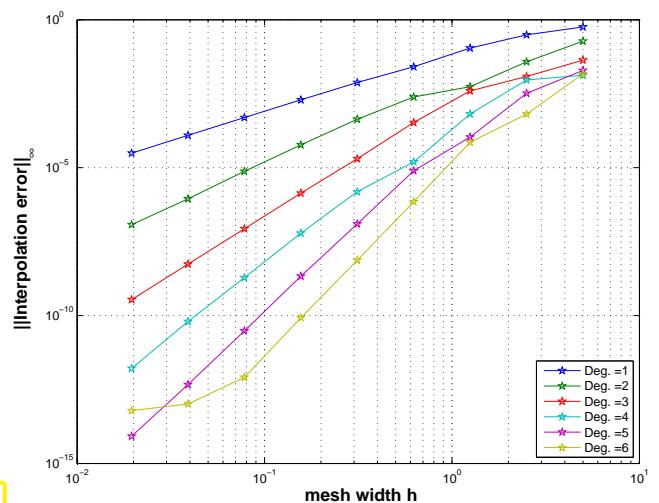
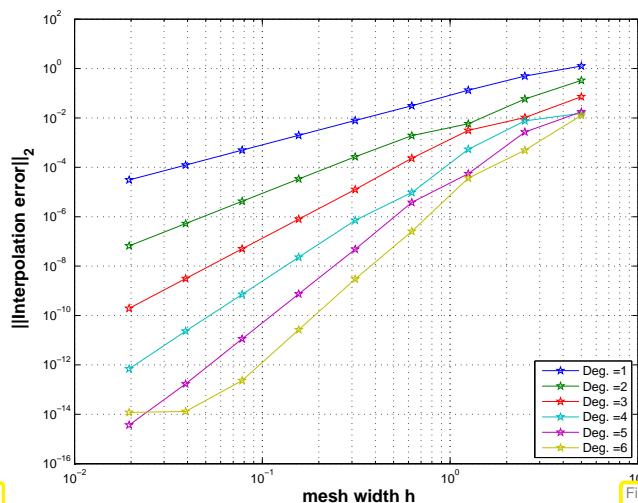


- \* Sequence of (equidistant) meshes:  $\mathcal{M}_i := \{-5 + j 2^{-i} 10\}_{j=0}^{2^i}, i = 1, \dots, 6$ .
- \* Equidistant local interpolation nodes (endpoints of grid intervals included).

Monitored: interpolation error in (approximate)  $L^\infty$ - and  $L^2$ -norms, see (4.1.86), (4.1.85)

$$\|g\|_{L^\infty([-5, 5])} \approx \max_{j=0, \dots, 1000} |g(-5 + j/100)|,$$

$$\|g\|_{L^2([-5, 5])} \approx \sqrt{\frac{1}{1000}} \cdot \left( \frac{1}{2} g(-5)^2 + \sum_{j=1}^{999} |g(-5 + j/100)|^2 + \frac{1}{2} g(5)^2 \right)^{1/2}.$$



Observation: Algebraic convergence ( $\rightarrow$  Def. 4.1.31) for meshwidth  $h_M \rightarrow 0$

(nearly linear error norm graphs in doubly logarithmic scale, see Rem. 4.1.33)

Observation: rate of algebraic convergence increases with polynomial degree  $n$

Rates  $\alpha$  of algebraic convergence  $O(h_M^\alpha)$  of norms of interpolation error:

$n$	1	2	3	4	5	6
w.r.t. $L^2$ -norm	1.9957	2.9747	4.0256	4.8070	6.0013	5.2012
w.r.t. $L^\infty$ -norm	1.9529	2.8989	3.9712	4.7057	5.9801	4.9228

- Higher polynomial degree provides faster algebraic decrease of interpolation error norms. Empiric evidence for rates  $\alpha = p + 1$

Here: rates estimated by linear regression ( $\rightarrow$  Ex. 6.0.9) based on MATLAB's `polyfit` and the interpolation errors for meshwidth  $h \leq 10 \cdot 2^{-5}$ . This was done in order to avoid erratic "preasymptotic", that is, for large meshwidth  $h$ , behavior of the error.

The bad rates for  $n = 6$  are probably due to the impact of roundoff, because the norms of the interpolation error had dropped below machine precision, see Fig. 159, 160.

#### (4.5.11) Approximation error estimates for piecewise polynomial Lagrange interpolation

The observations made in Ex. 4.5.10 are easily explained by applying the polynomial interpolation error estimates of Section 4.1.2 *locally* on the mesh intervals  $[x_{j-1}, x_j]$ ,  $j = 1, \dots, m$ : for constant polynomial degree  $n = n_j$ ,  $j = 1, \dots, m$ , we get

$$(4.1.43) \Rightarrow \|f - s\|_{L^\infty([x_0, x_m])} \leq \frac{h_M^{n+1}}{(n+1)!} \|f^{(n+1)}\|_{L^\infty([x_0, x_m])}, \quad (4.5.12)$$

with mesh width  $h_M := \max\{|x_j - x_{j-1}| : j = 1, \dots, m\}$ .

Another special case: fixed mesh  $M$ , uniform polynomial degree  $n$

Study estimates  $\|f - I_M f\| \leq CT(n)$  for  $n \rightarrow \infty$ .

Terminology:

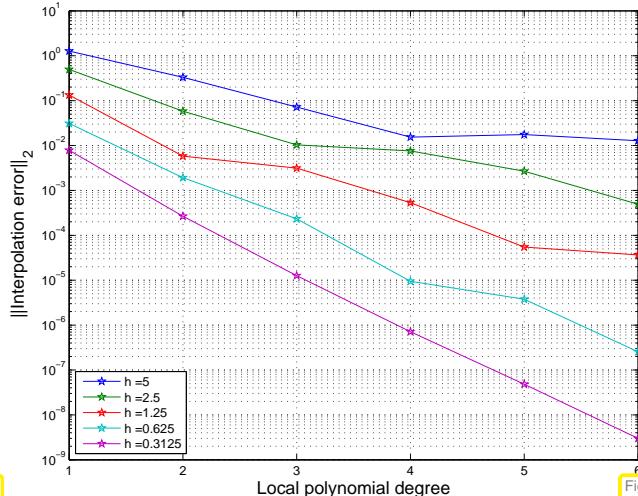
investigation of  $p$ -convergence**Example 4.5.13 ( $p$ -convergence of piecewise polynomial interpolation)**We study  $p$ -convergence in the setting of Ex. 4.5.10.

Fig. 161

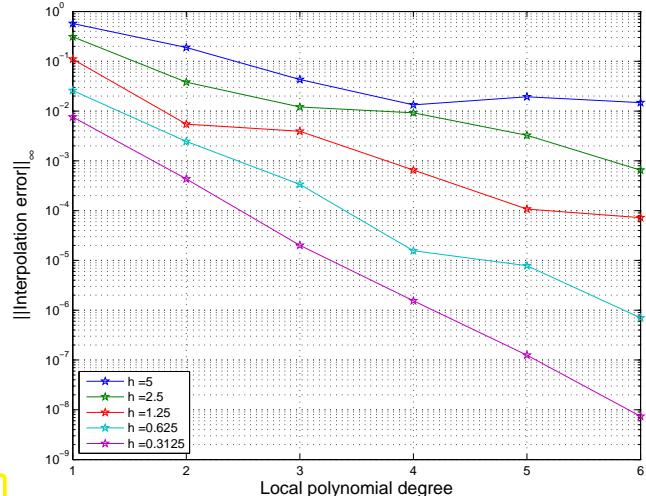


Fig. 162

Observation: (apparent) exponential convergence in polynomial degree

Note: in the case of  $p$ -convergence the situation is the same as for standard polynomial interpolation, see 4.1.2.

In this example we deal with an analytic function, see Rem. 4.1.87. Though equidistant local interpolation nodes are used cf. Ex. 4.1.34, the mesh intervals seems to be small enough that even in this case exponential convergence prevails.

**4.5.2 Cubic Hermite interpolation: error estimates**

See Section 3.4 for definition and algorithms for cubic Hermite interpolation of data points, with a focus on shape preservation, however. If the derivative  $f'$  of the interpoland  $f$  is available (in procedural form), then it can be used to fix local cubic polynomials by prescribing point values and derivative values in the endpoints of grid intervals.

**Definition 4.5.14. Piecewise cubic Hermite interpolant (with exact slopes) → Def. 3.4.1**

[Piecewise cubic Hermite interpolant (with exact slopes)] Given  $f \in C^1([a, b])$  and a mesh  $\mathcal{M} := \{a = x_0 < x_1 < \dots < x_{m-1} < x_m = b\}$  the **piecewise cubic Hermite interpolant** (with exact slopes)  $s : [a, b] \rightarrow \mathbb{R}$  is defined as

$$s|_{[x_{j-1}, x_j]} \in \mathcal{P}_3, \quad j = 1, \dots, m, \quad s(x_j) = f(x_j), \quad s'(x_j) = f'(x_j), \quad j = 0, \dots, m.$$

Clearly, the piecewise cubic Hermite interpolant is continuously differentiable:  $s \in C^1([a, b])$ , cf. Cor. 3.4.2.**Experiment 4.5.15 (Convergence of Hermite interpolation with exact slopes)**

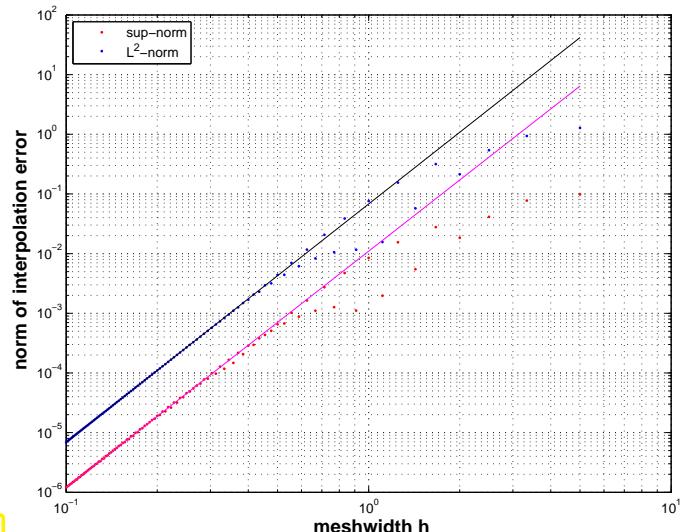
In this experiment we study the  $h$ -convergence of Cubic Hermite interpolation for a smooth function.

Piecewise cubic Hermite interpolation of

$$f(x) = \arctan(x).$$

- \* domain:  $I = (-5, 5)$
- \* mesh  $\mathcal{T} = \{-5 + h j\}_{j=0}^n \subset I$ ,  $h = \frac{10}{n}$ ,
- \* exact slopes  $c_i = f'(t_i)$ ,  $i = 0, \dots, n$

► algebraic convergence  $O(h^4)$



Approximate computation of error norms analogous to Ex. 4.5.10.

#### MATLAB-code 4.5.16: Hermite approximation and orders of convergence with exact slopes

```

1 function hermiteapprox1(f,df,a,b,N)
2 % Investigation of interpolation error norms for cubic Hermite
3 % interpolation of f (handle f)
4 % on [a,b] with slopes given by f' (function handle df).
5 % N gives the maximum number of mesh intervals
6 err = [];
7 for j=2:N
8     xx=a; % xx is the fine mesh on which the error norms are computed
9     val=f(a); % function values of mesh xx
10
11    t = a:(b-a)/j:b; % mesh nodes
12    y = f(t); c=df(t); % function values and slopes provide coefficients
13
14    for k=1:j-1
15        vx = linspace(t(k),t(k+1), 100);
16        % See Code 3.4.6 for local evaluation of Hermite interpolant
17        locval=hermloceval(vx,t(k),t(k+1),y(k),y(k+1),c(k),c(k+1));
18        xx=[xx, vx(2:100)];
19        val=[val,locval(2:100)];
20    end
21    d = abs(feval(f,xx)-val); % pointwise error on mesh xx
22    h = (b-a)/j;
23    % compute L^2 norm of the error using trapezoidal rule
24    l2 = sqrt(h*(sum(d(2:end-1).^2)+(d(1)^2+d(end)^2)/2) );
25    % columns of err = meshwidth, sup-norm error, L^2 error:
26    err = [err; h, max(d),l2];
27
28 figure('Name','Hermite approximation');
29 loglog(err(:,1),err(:,2),'r.',err(:,1),err(:,3),'b.');
30 grid on;
31 xlabel('{\bf meshwidth h}', 'fontsize',14);
32 ylabel('{\bf norm of interpolation error}', 'fontsize',14);

```

```

33 legend('sup-norm','L^2-norm','location','northwest');
34
35 % compute estimates for algebraic orders of convergence
36 % using linear regression on half of the data points
37 pI = polyfit(log(err(ceil(N/2):N-2,1)),log(err(ceil(N/2):N-2,2)),1);
38 exp_rate_Linf=pI(1)
39 pL2 = polyfit(log(err(ceil(N/2):N-2,1)),log(err(ceil(N/2):N-2,3)),1);
40 exp_rate_L2=pL2(1)
41 hold on;
42 plot([err(1,1),err(N-1,1)],
43 [err(1,1),err(N-1,1)].^pI(1)*exp(pI(2)), 'm')
44 plot([err(1,1),err(N-1,1)],
45 [err(1,1),err(N-1,1)].^pL2(1)*exp(pL2(2)), 'k')
46
47 print -depsc2 '../PICTURES/hermiperrslopes.eps';

```

The observation made in Exp. 4.5.15 matches the theoretical prediction of the rate of algebraic convergence for cubic Hermite interpolation with exact slopes for a smooth function.

#### Theorem 4.5.17. Convergence of approximation by cubic Hermite interpolation

Let  $s$  be the cubic Hermite interpolant of  $f \in C^4([a, b])$  on a mesh  $\mathcal{M} := \{a = x_0 < x_1 < \dots < x_{m-1} < x_m = b\}$  according to Def. 4.5.14. Then

$$\|f - s\|_{L^\infty([a,b])} \leq \frac{1}{4!} h_{\mathcal{M}}^4 \|f^{(4)}\|_{L^\infty([a,b])},$$

with the meshwidth  $h_{\mathcal{M}} := \max_j |x_j - x_{j-1}|$ .

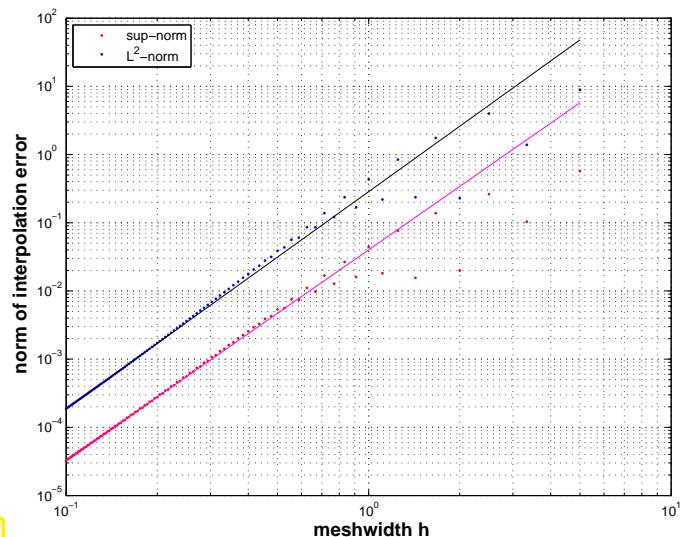
In Section 3.4.2 we saw variants of cubic Hermite interpolation, for which the slopes  $c_j = s'(x_j)$  were computed from the values  $y_j$  in preprocessing step. Now we study the use of such a scheme for approximation.

#### Experiment 4.5.18 (Convergence of Hermite interpolation with averaged slopes)

Piecewise cubic Hermite interpolation of

$$f(x) = \arctan(x).$$

- \* domain:  $I = (-5, 5)$
  - \* equidistant mesh  $\mathcal{T}$  in  $I$ , see Exp. 4.5.15,
  - \* averaged local slopes, see (3.4.8)
- algebraic convergence  $O(h^3)$  in meshwidth  
(see Code 4.5.19)



We observe lower rate of algebraic convergence compared to the use of exact slopes due to averaging (3.4.8). From the plot we deduce  $O(h^3)$  asymptotic decay of the  $L^2$ - and  $L^\infty$ -norms of the approximation error for meshwidth  $h \rightarrow 0$ .

#### MATLAB-code 4.5.19: Hermite approximation and orders of convergence

```

1 % 16.11.2009 hermiteapprox.m
2 % Plot convergence of approximation error of cubic Hermite
3 % interpolation
4 % with respect to the meshwidth
5 % print the algebraic order of convergence in sup and L^2 norms
6 % Slopes: weighted average of local slopes
7 %
8 % inputs: f function to be interpolated
9 % a, b left and right extremes of the interval
10 % N maximum number of subdomains
11
12 function hermiteapprox(f,a,b,N)
13 % Investigation of interpolation error norms for cubic Hermite
14 % interpolation of f (handle f)
15 % on [a,b] with linearly averaged slopes according to (3.4.8).
16 % N gives the maximum number of mesh intervals
17 err = [];
18 for j=2:N
19     xx=a;          % xx is the fine mesh on which the error norms are computed
20     val=f(a);      % function values on xx
21
22     t = a:(b-a)/j:b;        % mesh nodes
23     y = f(t); c=slopes1(t,y); % coefficients for Hermit polynomial
24     representation
25
26     for k=1:j-1
27         vx = linspace(t(k),t(k+1), 100);
28         locval=hermloceval(vx,t(k),t(k+1),y(k),y(k+1),c(k),c(k+1));
29         xx=[xx, vx(2:100)];
30         val=[val,locval(2:100)];
31     end
32     d = abs(feval(f,xx)- val);
33     h = (b-a)/j;

```

```

31 % compute  $L^2$  norm of the error using trapezoidal rule
32 l2 = sqrt(h*(sum(d(2:end-1).^2)+(d(1)^2+d(end)^2)/2));
33 % columns of err = meshwidth, sup-norm error,  $L^2$  error:
34 err = [err; h,max(d),l2];
35 end
36
37 figure('Name','Hermite approximation');
38 loglog(err(:,1),err(:,2),'r.',err(:,1),err(:,3),'b.');
39 grid on;
40 xlabel('{\bf meshwidth h}', 'fontsize', 14);
41 ylabel('{\bf norm of interpolation error}', 'fontsize', 14);
42 legend('sup-norm', 'L^2-norm', 'location', 'northwest');
43
44 % compute estimates for algebraic orders of convergence
45 % using linear regression on half of the data points
46 pI = polyfit(log(err(ceil(N/2):N-2,1)), log(err(ceil(N/2):N-2,2)), 1);
    exp_rate_Linf=pI(1)
47 pL2 = polyfit(log(err(ceil(N/2):N-2,1)), log(err(ceil(N/2):N-2,3)), 1);
    exp_rate_L2=pL2(1)
48 hold on;
49 plot([err(1,1),err(N-1,1)],
      [err(1,1),err(N-1,1)].^pI(1)*exp(pI(2)), 'm')
50 plot([err(1,1),err(N-1,1)],
      [err(1,1),err(N-1,1)].^pL2(1)*exp(pL2(2)), 'k')
51
52 print -depsc2 '../PICTURES/hermiperravgsl.eps';
53
54 %-----
55 function c=slopes1(t,y)
56 h = diff(t);                                % increments in t
57 delta = diff(y)./h;                          % slopes of piecewise
58 % linear interpolant
59 c = [delta(1), ...
60       ((h(2:end).*delta(1:end-1)+h(1:end-1).*delta(2:end))...
61        ./ (t(3:end) - t(1:end-2))), ...
62       delta(end)];
63

```

### 4.5.3 Cubic spline interpolation: error estimates [42, Ch. 47]

Recall concept and algorithms for cubic spline interpolation from Section 3.5.1. As an interpolation scheme it can also serve as the foundation for an approximation scheme according to § 4.0.5: the mesh will double as known set, see Def. 3.5.1. Cubic spline interpolation is not local as we saw in § 3.5.18. Nevertheless, cubic spline interpolants can be computed with an effort of  $\mathcal{O}(m)$  as elaborated in § 3.5.4.

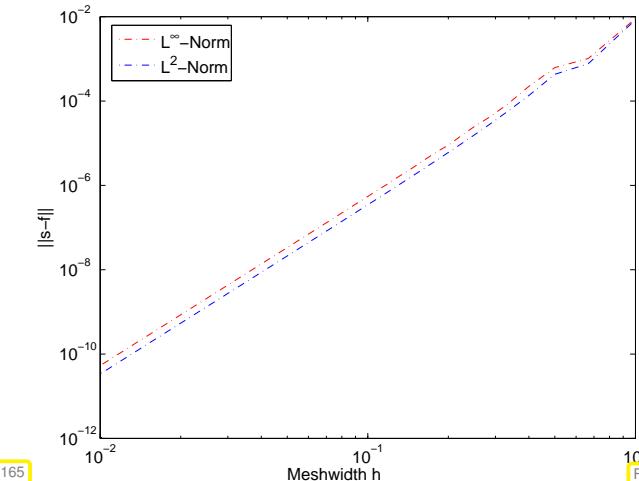
#### Experiment 4.5.20 (Approximation by complete cubic spline interpolants)

We take  $I = [-1, 1]$  and rely on an equidistant mesh (knot set)  $\mathcal{M} := \{-1 + \frac{2}{n}j\}_{j=0}^n$ ,  $n \in \mathbb{N}$   $\rightarrow$

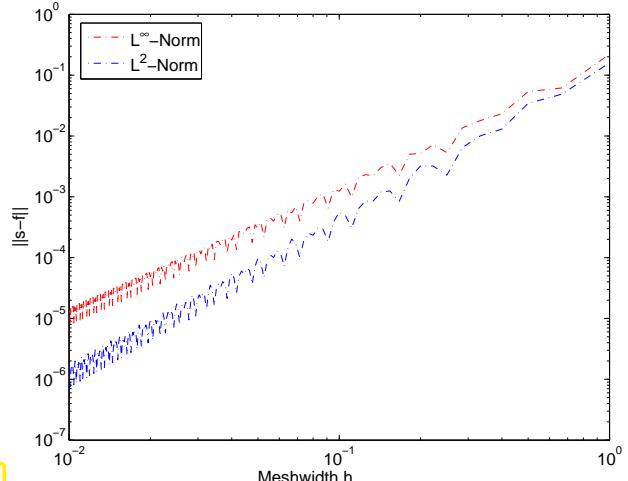
meshwidth  $h = 2/n$ .

We study  $h$ -convergence of **complete** ( $\rightarrow$  § 3.5.10) cubic spline interpolation, where the slopes at the endpoints of the interval are made to agree with the derivatives of the interpoland at these points. As interpolands we consider

$$f_1(t) = \frac{1}{1+e^{-2t}} \in C^\infty(I) \quad , \quad f_2(t) = \begin{cases} 0 & , \text{ if } t < -\frac{2}{5}, \\ \frac{1}{2}(1 + \cos(\pi(t - \frac{3}{5}))) & , \text{ if } -\frac{2}{5} < t < \frac{3}{5}, \\ 1 & \text{otherwise.} \end{cases} \in C^1(I).$$



$$\|f_1 - s\|_{L^\infty([-1,1])} = O(h^4)$$



$$\|f_2 - s\|_{L^\infty([-1,1])} = O(h^2)$$

We observe algebraic order of convergence in  $h$  with empiric rate approximately given by  $\min\{1 + \text{regularity of } f, 4\}$ .

We remark that there is the following theoretical result [41], [15, Rem. 9.2]:

$$f \in C^4([t_0, t_n]) \quad \|f - s\|_{L^\infty([t_0, t_n])} \leq \frac{5}{384} h^4 \|f^{(4)}\|_{L^\infty([t_0, t_n])}.$$

#### MATLAB-code 4.5.21: Spline approximation error

```

1 function splineapprox(f,df,a,b,N)
2 % Study of interpolation error norms for complete cubic spline
3 % interpolation of f
4 x = a:0.00025:b; fv = feval(f,x); % fine mesh for norm evaluation
5 dfa = feval(df,a); dfb = feval(df,b); % Slopes at endpoints
6 err = [];
7 for j=2:N
8 t = a:(b-a)/j:b; % spline knots
9 y = [dfa, feval(f,t),dfb];
10 % compute complete spline imposing exact first derivative at the
11 % endpoints
12 % Please refer to MATLAB documentation of spline
13 v = spline(t,y,x);

```

```

13 d = abs(fv-v);
14 h = x(2:end)-x(1:end-1);
15 % compute  $L^2$  norm of the error using trapezoidal rule on mesh x
16 l2 = sqrt(0.5*dot(h, (d(1:end-1).^2+d(2:end).^2)));
17 % columns of err = meshwidth,  $L^\infty$  error,  $L^2$  error:
18 err = [err; (b-a)/j, max(d), l2];
19 end
20
21 figure('Name','Spline interpolation');
22 plot(t,y(2:end-1),'m*',x,fv,'b-',x,v,'r-');
23 xlabel('{\bf t}', 'fontsize',14); ylabel('{\bf s(t)}', 'fontsize',14);
24 legend('Data points','f','Cubic spline interpolant','location','best');
25
26 figure('Name','Spline approximation error');
27 loglog(err(:,1),err(:,2),'r.-',err(:,1),err(:,3),'b.-');
28 grid on; xlabel('Meshwidth h'); ylabel('||s-f||');
29 legend('sup-norm',' $L^2$ -norm', 'Location','NorthWest');
30 %compute algebraic orders of convergence using polynomial fit
31 p = polyfit(log(err(:,1)), log(err(:,2)),1); exp_rate_Linf=p(1)
32 p = polyfit(log(err(:,1)), log(err(:,3)),1); exp_rate_L2=p(1)

```

#### MATLAB-code 4.5.22: Spline approximation: driver script

```

1 splineapprox(@atan, @(x) 1./(1+x.^2), -5,5,100);
2 splineapprox(@(x) 1./(1+exp(-2*x)), ...
3 @(x) 2*exp(-2*x)./(1+exp(-2*x)).^2, -1,1,100);

```

## 4.6 Multi-dimensional Approximation on Tensor-Product Domains

### Summary and Learning Outcomes

# Chapter 5

## Numerical Quadrature



*Supplementary reading.* Numerical quadrature is covered in [42, VII] and [15, Ch. 10].

Numerical quadrature deals with the *approximate* numerical evaluation of integrals  $\int_{\Omega} f(\mathbf{x}) d\mathbf{x}$  for a given (closed) integration domain  $\Omega \subset \mathbb{R}^d$ . Thus, the underlying problem in the sense of § 1.5.62 is the mapping

$$I : \begin{cases} C^0(\Omega) & \rightarrow \mathbb{R} \\ f & \mapsto \int_{\Omega} f(\mathbf{x}) d\mathbf{x} \end{cases} \quad (5.0.1)$$

If  $f$  is complex-valued or vector-valued, then so is the integral. The methods presented in this chapter can immediately be generalized to this case by componentwise application.

### (5.0.2) Integrands in procedural form

The integrand  $f$ , a continuous function  $f : \Omega \subset \mathbb{R}^d \mapsto \mathbb{R}$  should not be thought of as given by an analytic expression, but as given in **procedural form**,

- ☞ in MATLAB    `function y = f(x)`
- ☞ in C++ as a data type providing an evaluation operator **double operator** (Point &) or a corresponding member function, see Code 3.1.5 for an example.
- General methods for numerical quadrature should rely only on finitely many *point evaluations* of the integrand.

In this chapter the focus is on the special case  $d = 1$ :  $\Omega = [a, b]$  (an interval)  
(Multidimensional numerical quadrature is substantially more difficult and will be treated in the course “Numerical methods for partial differential equations”.)

### Remark 5.0.3 (Importance of numerical quadrature)

- ☞ Numerical quadrature methods are key building blocks for so-called variational methods for the numerical treatment of partial differential equations. A prominent example is the **finite element method**.

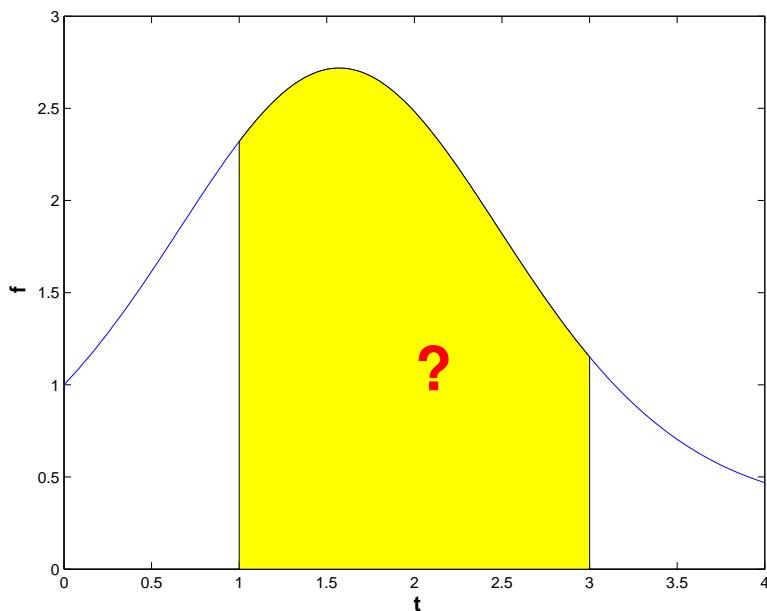
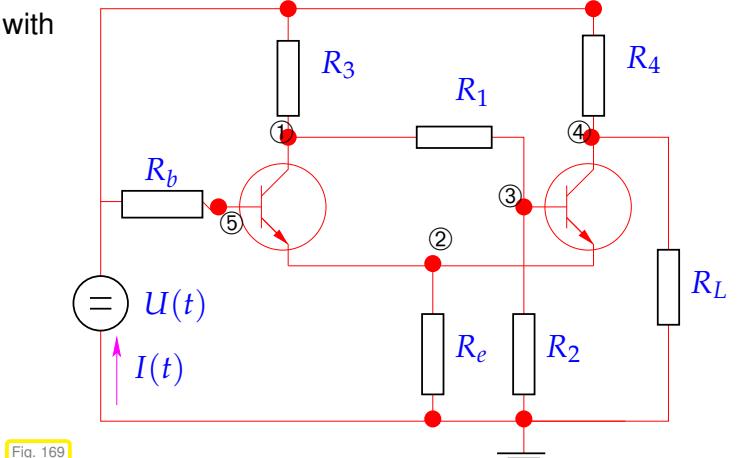
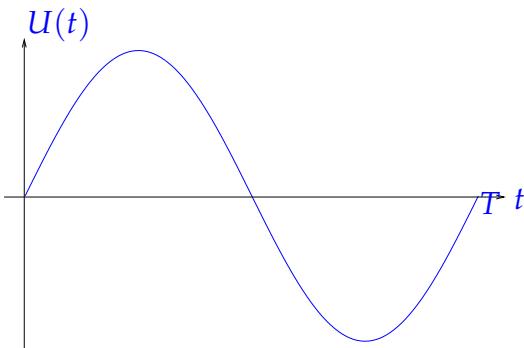


Fig. 167

### Example 5.0.4 (Heating production in electrical circuits)

In Ex. 1.6.3 we learned about the nodal analysis of electrical circuits. Its application to a non-linear circuit was discussed in Ex. 2.0.1. In this example we consider a non-linear circuit in quasi-stationary operation (capacitors and inductances are ignored). Then the computation of branch currents and nodal voltages entails solving a non-linear system of equations.

Now assume time-harmonic periodic excitation with period  $T > 0$ .



The goal is to compute the energy dissipated by the circuit, which is equal to the energy injected by the voltage source. This energy can be obtained by integrating the power  $P(t) = U(t)I(t)$  over period  $[0, T]$ :

$$W_{\text{therm}} = \int_0^T U(t)I(t) dt, \quad \text{where } I = I(U).$$

function  $I = \text{current}(U)$  involves solving non-linear system of equations, see Ex. 2.0.1!

This is a typical example where “point evaluation” by solving the non-linear circuit equations is the only way to gather information about the integrand.

## Contents

5.1 Quadrature Formulas . . . . .	361
5.2 Polynomial Quadrature Formulas . . . . .	364

5.3 Gauss Quadrature . . . . .	367
5.4 Composite Quadrature . . . . .	380
5.5 Adaptive Quadrature . . . . .	387

## 5.1 Quadrature Formulas

Quadrature formulas realize the approximation of an integral through finitely many point evaluations of the integrand.

### Definition 5.1.1. Quadrature formula/quadrature rule

An  $n$ -point quadrature formula/quadrature rule on  $[a, b]$  provides an approximation of the value of an integral through a weighted sum of point values of the integrand:

$$\int_a^b f(t) dt \approx Q_n(f) := \sum_{j=1}^n w_j^n f(c_j^n). \quad (5.1.2)$$

Terminology:  $w_j^n$  : quadrature weights  $\in \mathbb{R}$   
 $c_j^n$  : quadrature nodes  $\in [a, b]$

Obviously (5.1.2) is compatible with integrands  $f$  given in procedural form as function  $y = f(x)$ , compare § 5.0.2.

### MATLAB-code 5.1.3: MATLAB template implementing generic quadrature formula

```

1 function I = quadformula(f, c, w)
2 % Generic numerical quadrature routine implementing (5.1.2):
3 % f is a handle to a function of type f = @(x) ....
4 % c, w pass quadrature nodes c_j ∈ [a, b], and weights w_j > 0
5 n = length(c);
6 if (length(w) ~= n), error('#weights != #nodes'); end
7 I = 0; for j=1:n, I = I + w(j)*f(c(j)); end

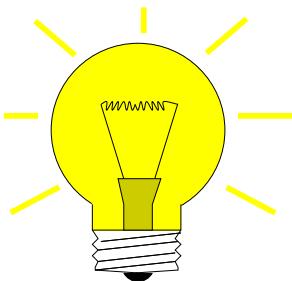
```

A single invocation costs  $n$  point evaluations of the integrand plus  $n$  additions and multiplications.

### Remark 5.1.4 (Transformation of quadrature rules)

In the setting of function approximation by polynomials we learned in Rem. 4.1.14 that an approximation schemes for any interval could be obtained from an approximation scheme on a single reference interval  $[-1, 1]$  in Rem. 4.1.14) by means of affine pullback, see (4.1.18). A similar affine transformation technique makes it possible to derive quadrature formula for an arbitrary interval from a single quadrature formula on a reference interval.

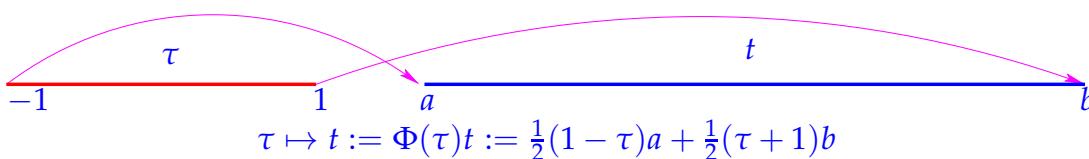
Given: quadrature formula  $(\hat{c}_j, \hat{w}_j)_{j=1}^n$  on reference interval  $[-1, 1]$



Idea: transformation formula for integrals

$$\int_a^b f(t) dt = \frac{1}{2}(b-a) \int_{-1}^1 \hat{f}(\tau) d\tau , \quad (5.1.5)$$

$$\hat{f}(\tau) := f\left(\frac{1}{2}(1-\tau)a + \frac{1}{2}(\tau+1)b\right).$$



Note that  $\hat{f}$  is the affine pullback  $\Phi^*f$  of  $f$  to  $[-1, 1]$  as defined in Eq. (4.1.16).

► quadrature formula for general interval  $[a, b]$ ,  $a, b \in \mathbb{R}$ :

$$\int_a^b f(t) dt \approx \frac{1}{2}(b-a) \sum_{j=1}^n \hat{w}_j \hat{f}(\hat{c}_j) = \sum_{j=1}^n w_j f(c_j) \quad \text{with} \quad c_j = \frac{1}{2}(1-\hat{c}_j)a + \frac{1}{2}(1+\hat{c}_j)b ,$$

$$w_j = \frac{1}{2}(b-a)\hat{w}_j .$$

In words, the nodes are just mapped through the affine transformation  $c_j = \Phi(\hat{c}_j)$ , the weights are scaled by the ratio of lengths of  $[a, b]$  and  $[-1, 1]$ .

► A 1D quadrature formula on arbitrary intervals can be specified by providing its weights  $\hat{w}_j$ /nodes  $\hat{c}_j$  for the integration domain  $[-1, 1]$  (reference interval). Then the above transformation is assumed.

Another common choice for the reference interval:  $[0, 1]$ , pay attention!

### (5.1.6) Convergence of numerical quadrature

In general the quadrature formula (5.1.2) will only provide an approximate value for the integral.

► For a generic integrand we will encounter a non-vanishing

$$\text{quadrature error} \quad E_n(f) := \left| \int_a^b f(t) dt - Q_n(f) \right|$$

As in the case of function approximation by interpolation Section 4.1.2, our focus will on the **asymptotic** behavior of the quadrature error as a function of the number  $n$  of point evaluations of the integrand.

Therefore consider **families** of quadrature rules  $\{Q_n\}_n$  ( $\rightarrow$  Def. 5.1.1) described by

- \* quadrature weights  $\{w_j^n, j = 1, \dots, n\}_{n \in \mathbb{N}}$  and
- \* quadrature nodes  $\{c_j^n, j = 1, \dots, n\}_{n \in \mathbb{N}}$ .

We study the *asymptotic* behavior of the **quadrature error**  $E(n)$  for  $n \rightarrow \infty$

As in the case of interpolation errors in § 4.1.29 we make the usual qualitative distinction, see Def. 4.1.31:

- ▷ algebraic convergence  $E(n) = O(n^{-p})$ , rate  $p > 0$
- ▷ exponential convergence  $E(n) = O(q^n)$ ,  $0 \leq q < 1$

Note that the number  $n$  of nodes agrees with the number of  $f$ -evaluations required for evaluation of the quadrature formula. This is usually used as a *measure for the cost* of computing  $Q_n(f)$ .

Therefore, in the sequel, we consider the quadrature error as a function of  $n$ .

### Remark 5.1.7 (Tables of quadrature rules)

In many codes families of quadrature rules are used to control the quadrature error. Usually, suitable sequences of weights  $w_j^n$  and nodes  $c_j^n$  are **precomputed** and stored in tables up to sufficiently large values of  $n$ . A possible interface could be the following:

```
struct QuadTab {
    template <typename VecType>
    static void getrule(int n, VecType &c, VecType &w, double
        a=-1.0, double b=1.0);
}
```

Calling the method `getrule()` fills the vectors  $c$  and  $w$  with the nodes and the weights for a desired  $n$ -point quadrature on  $[a, b]$  with  $[-1, 1]$  being the default reference interval. For **VecType** we may assume the basic functionality of **Eigen::VectorXd**.

### (5.1.8) Quadrature by approximation schemes

Every approximation scheme  $A : C^0([a, b]) \rightarrow V$ ,  $V$  a space of “simple functions” on  $[a, b]$ , see § 4.0.4, gives rise to a method for numerical quadrature according to

$$\int_a^b f(t) dt \approx Q_A(f) := \int_a^b (Af)(t) dt . \quad (5.1.9)$$

As explained in § 4.0.5 every interpolation scheme  $I_{\mathcal{T}}$  based on the node set  $\mathcal{T} = \{t_0, t_1, \dots, t_{n+1}\} \subset [a, b]$  ( $\rightarrow$  § 3.1.2) induces an approximation scheme, and, hence, also an quadrature scheme on  $[a, b]$ :

$$\int_a^b f(t) dt \approx \int_a^b I_{\mathcal{T}}[f(t_0), \dots, f(t_n)]^{\top}(t) dt . \quad (5.1.10)$$

#### Lemma 5.1.11. Quadrature formulas from linear interpolation schemes

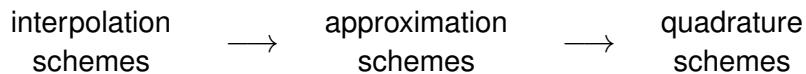
Every linear interpolation operator  $I_{\mathcal{T}}$  according to Def. 3.1.15 spawns a quadrature formula ( $\rightarrow$  Def. 5.1.1) by (5.1.10).

*Proof.* Writing  $e_j$  for the  $j$ -th unit vector of  $\mathbb{R}^n$ , we have by linearity

$$\int_a^b I_{\mathcal{T}}[f(t_0), \dots, f(t_n)]^{\top} dt = \sum_{j=0}^n f(t_j) \underbrace{\int_a^b (I_{\mathcal{T}}(e_j))(t) dt}_{\text{weight } w_j} . \quad (5.1.12)$$

Hence, we have arrived at an  $n + 1$ -point quadrature formula with nodes  $t_j$ , whose weights are the integrals of the **cardinal interpolants** for the interpolation scheme  $\mathcal{T}$ .  $\square$

Summing up, we have found



### (5.1.13) Quadrature error from approximation error

Bounds for the maximum norm of the approximation error of an approximation scheme directly translate into estimates of the quadrature error of the induced quadrature scheme (5.1.9):

$$\left| \int_a^b f(t) dt - QA(f) \right| \leq \int_a^b |f(t) - A(f)(t)| dt \leq |b-a| \|f - A(f)\|_{L^\infty([a,b])}. \quad (5.1.14)$$

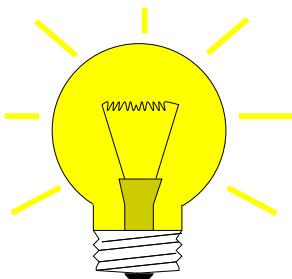
Hence, the various estimates derived in Section 4.1.2 and Section 4.1.3.2 give us quadrature error estimates “for free”. More details will be given in the next section.

## 5.2 Polynomial Quadrature Formulas

Now we specialize the general recipe of § 5.1.8 for approximation schemes based on global polynomials, the Lagrange approximation scheme as introduced in Section 4.1, Def. 4.1.25.



*Supplementary reading.* This topic is discussed in [15, Sect. 10.2].



Idea: replace integrand  $f$  with  $p_{n-1} := l_{\mathcal{T}} \in \mathcal{P}_{n-1}$  = polynomial Lagrange interpolant of  $f$  ( $\rightarrow$  Cor. 3.2.15) for given node set  $\mathcal{T} := \{t_0, \dots, t_{n-1}\} \subset [a, b]$

$$\Rightarrow \int_a^b f(t) dt \approx Q_n(f) := \int_a^b p_{n-1}(t) dt. \quad (5.2.1)$$

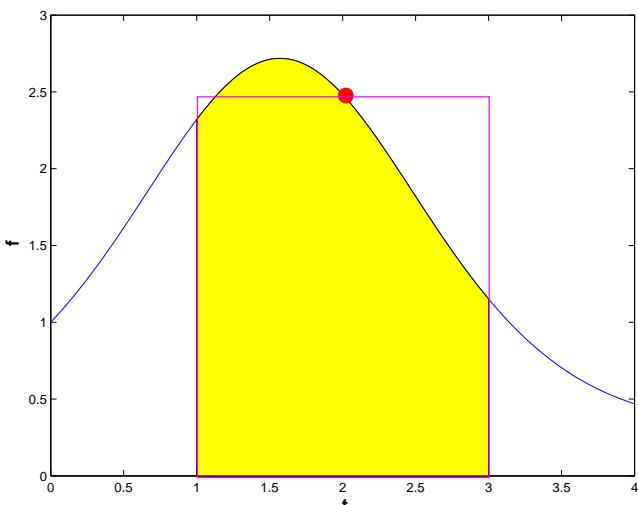
The cardinal interpolants for Lagrange interpolation are the Lagrange polynomials (3.2.11)

$$L_i(t) := \prod_{\substack{j=0 \\ j \neq i}}^{n-1} \frac{t - t_j}{t_i - t_j}, \quad i = 0, \dots, n-1 \quad \xrightarrow{(3.2.13)} \quad p_{n-1}(t) = \sum_{i=0}^{n-1} f(t_i) L_i(t).$$

Then (5.1.12) amounts to the  $n$ -point quadrature formula

$$\int_a^b p_{n-1}(t) dt = \sum_{i=0}^{n-1} f(t_i) \int_a^b L_i(t) dt \quad \xrightarrow{\begin{array}{l} \text{nodes } c_i = t_{i-1}, \\ \text{weights } w_i := \int_a^b L_{i-1}(t) dt. \end{array}} \quad (5.2.2)$$

### Example 5.2.3 (Midpoint rule)



The **midpoint rule** is (5.2.2) for  $n = 1$  and  $t_0 = \frac{1}{2}(a + b)$ . It leads to the 1-point quadrature formula

$$\int_a^b f(t) dt \approx Q_{\text{mp}}(f) = (b - a)f\left(\frac{1}{2}(a + b)\right).$$

“midpoint”

▷ the area under the graph of  $f$  is approximated by the area of a rectangle.

### Example 5.2.4 (Newton-Cotes formulas → [42, Ch. 38])

The Newton-Cotes formulas arise from Lagrange interpolation in equidistant nodes (4.1.27) in the integration interval  $[a, b]$ :

► **Equidistant quadrature nodes**

$$t_j := a + hj, \quad h := \frac{b - a}{n}, \quad j = 0, \dots, n:$$

The weights for the interval  $[0, 1]$  can be found, e.g., by symbolic computation using MAPLE: the following MAPLE function expects the polynomial degree as input argument.

```
> newtoncotes := n -> factor(int(interp([seq(i/n, i=0..n)], [seq(f(i/n), i=0..n)], z), z=0..1)):
```

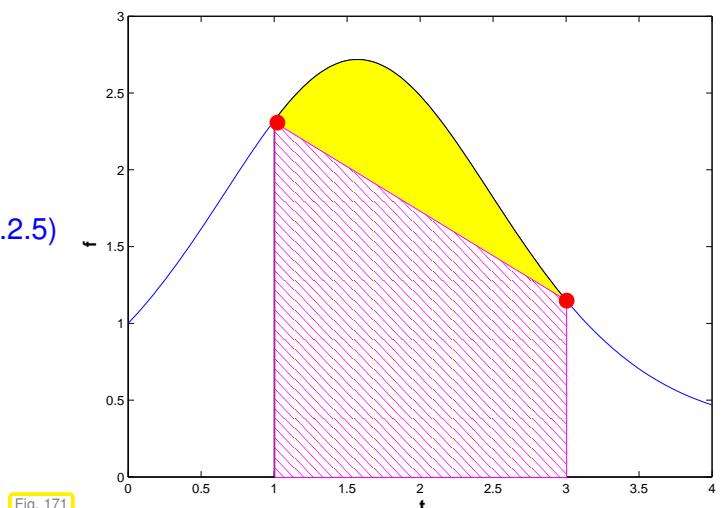
Weights on general intervals  $[a, b]$  can then be deduced by the affine transformation rule as explained in Rem. 5.1.4.

- $n = 1$ : **Trapezoidal rule** (integrate linear interpolant of integrand in endpoints)

```
> trapez := newtoncotes(1);
```

$$\hat{Q}_{\text{trp}}(f) := \frac{1}{2}(f(0) + f(1)) \quad (5.2.5)$$

$$\left( \int_a^b f(t) dt \approx \frac{b - a}{2}(f(a) + f(b)) \right)$$



- $n = 2$ : **Simpson rule**

```
> simpson := newtoncotes(2);
```

$$\frac{h}{6} \left( f(0) + 4f\left(\frac{1}{2}\right) + f(1) \right) \quad \left( \int_a^b f(t) dt \approx \frac{b-a}{6} \left( f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right) \right) \quad (5.2.6)$$

- $n = 4$ : Milne rule

```
> milne := newtoncotes(4);
```

$$\begin{aligned} & \frac{1}{90} h \left( 7f(0) + 32f\left(\frac{1}{4}\right) + 12f\left(\frac{1}{2}\right) + 32f\left(\frac{3}{4}\right) + 7f(1) \right) \\ & \left( \frac{b-a}{90} (7f(a) + 32f(a+(b-a)/4) + 12f(a+(b-a)/2) + 32f(a+3(b-a)/4) + 7f(b)) \right) \end{aligned}$$

- $n = 6$ : Weddle rule

```
> weddle := newtoncotes(6);
```

$$\begin{aligned} & \frac{1}{840} h (41f(0) + 216f\left(\frac{1}{6}\right) + 27f\left(\frac{1}{3}\right) + 272f\left(\frac{1}{2}\right) \\ & + 27f\left(\frac{2}{3}\right) + 216f\left(\frac{5}{6}\right) + 41f(1)) \end{aligned}$$

- $n \geq 8$ : quadrature formulas with *negative weights*

```
> newtoncotes(8);
```

$$\begin{aligned} & \frac{1}{28350} h (989f(0) + 5888f\left(\frac{1}{8}\right) - 928f\left(\frac{1}{4}\right) + 10496f\left(\frac{3}{8}\right) \\ & - 4540f\left(\frac{1}{2}\right) + 10496f\left(\frac{5}{8}\right) - 928f\left(\frac{3}{4}\right) + 5888f\left(\frac{7}{8}\right) + 989f(1)) \end{aligned}$$

From Ex. 4.1.34 we know that the approximation error incurred by Lagrange interpolation in equidistant nodes can blow up even for analytic functions. This blow-up can also infect the quadrature error of Newton-Cotes formulas for large  $n$ , which renders them essentially useless. In addition they will be marred by large (in modulus) and negative weights, which compromises numerical stability ( $\rightarrow$  Def. 1.5.80)

### Remark 5.2.7 (Clenshaw-Curtis quadrature rules [79])

The considerations of Section 4.1.3 confirmed the superiority of the “optimal” Chebychev nodes (4.1.75) for globally polynomial Lagrange interpolation. This suggests that we use these nodes also for numerical quadrature with weights given by (5.2.2). This yields the so-called **Clenshaw-Curtis rules** with the following rather desirable property:

#### Theorem 5.2.8. Positivity of Clenshaw-Curtis weights

*The weights  $w_j^n$ ,  $j = 1, \dots, n$ , for every  $n$ -point Clenshaw-Curtis rule are positive.*

The weights of any  $n$ -point Clenshaw-Curtis rule can be computed with a computational effort of  $O(n \log n)$  using FFT.

### (5.2.9) Error estimates for polynomial quadrature

As a concrete application of § 5.1.13, (5.1.14) we use the  $L^\infty$ -bound (4.1.43) for Lagrange interpolation

$$\|f - L_T f\|_{L^\infty(I)} \leq \frac{\|f^{(n+1)}\|_{L^\infty(I)}}{(n+1)!} \max_{t \in I} |(t - t_0) \cdots (t - t_n)|. \quad (4.1.43)$$

to conclude for any  $n$ -point quadrature rule based on polynomial interpolation:

$$f \in C^n([a, b]) \Rightarrow \left| \int_a^b f(t) dt - Q_n(f) \right| \leq \frac{1}{n!} (b-a)^{n+1} \|f^{(n)}\|_{L^\infty([a, b])}. \quad (5.2.10)$$

Much sharper estimates for Clenshaw-Curtis rules ( $\rightarrow$  Rem. 5.2.7) can be inferred from the interpolation error estimate (4.1.79) for Chebychev interpolation. For functions with *limited smoothness algebraic convergence* of the quadrature error for Clenshaw-Curtis quadrature follows from (4.1.83). For integrands that possess an *analytic* extension to the complex plane in a neighborhood of  $[a, b]$ , we can conclude *exponential* convergence from (4.1.89).

## 5.3 Gauss Quadrature



### Supplementary reading.

Gauss quadrature is discussed in detail in [42, Ch. 40-41], [15,

Sect.10.3]

How to gauge the “quality” of an  $n$ -point quadrature formula  $Q_n$  without testing it for specific integrands? The next definition gives an answer.

### Definition 5.3.1. Order of a quadrature rule

The **order** of quadrature rule  $Q_n : C^0([a, b]) \rightarrow \mathbb{R}$  is defined as

$$\text{order}(Q_n) := \max\{m \in \mathbb{N}_0 : Q_n(p) = \int_a^b p(t) dt \quad \forall p \in \mathcal{P}_m\} + 1, \quad (5.3.2)$$

that is, as the **maximal** degree  $+1$  of polynomials for which the quadrature rule is guaranteed to be exact.

### (5.3.3) Order of polynomial quadrature rules

First we note a simple consequence of the invariance of the polynomial space  $\mathcal{P}_n$  under affine pullback, see Lemma 4.1.17.

**Corollary 5.3.4. Invariance of order under affine transformation**

An affine transformation of a quadrature rule according to Rem. 5.1.4 does not change its order.

Further, by construction all polynomial  $n$ -point quadrature rules possess order at least  $n$ .

**Theorem 5.3.5. Sufficient order conditions for quadrature rules**

An  $n$ -point quadrature rule on  $[a, b]$  ( $\rightarrow$  Def. 5.1.1)

$$Q_n(f) := \sum_{j=1}^n w_j f(t_j), \quad f \in C^0([a, b]),$$

with nodes  $t_j \in [a, b]$  and weights  $w_j \in \mathbb{R}$ ,  $j = 1, \dots, n$ , has **order  $\geq n$** , if and only if

$$w_j = \int_a^b L_{j-1}(t) dt, \quad j = 1, \dots, n,$$

where  $L_k$ ,  $k = 0, \dots, n-1$ , is the  $k$ -th **Lagrange polynomial** (3.2.11) associated with the ordered node set  $\{t_1, t_2, \dots, t_n\}$ .

*Proof.* The conclusion of the theorem is a direct consequence of the facts that

$$\mathcal{P}_{n-1} = \text{Span}\{L_0, \dots, L_{n-1}\} \quad \text{and} \quad L_k(t_j) = \delta_{k+1,j}, \quad k+1, j \in \{1, \dots, n\} :$$

just plug a Lagrange polynomial into the quadrature formula.  $\square$

By construction (5.2.2) polynomial  $n$ -point quadrature formulas (5.2.1) exact for  $f \in \mathcal{P}_{n-1} \Rightarrow$   $n$ -point polynomial quadrature formula has **at least** order  $n$ .

**Remark 5.3.6 (Choice of (local) quadrature weights)**

Thm. 5.3.5 provides a concrete formula for quadrature weights, which guaranteed order  $n$  for an  $n$ -point quadrature formula. Yet evaluating integrals of Lagrange polynomials may be cumbersome. Here we give a general recipe for finding the weights  $w_j$  according to Thm. 5.3.5 without dealing with Lagrange polynomials.

Given: arbitrary nodes  $c_1, \dots, c_n$  for  $n$ -point (local) quadrature formula on  $[a, b]$

From Def. 5.3.1 we immediately conclude the following procedure: If  $p_0, \dots, p_{n-1}$  is a basis of  $\mathcal{P}_n$ , then, thanks to the linearity of the integral and quadrature formulas,

$$Q_n(p_j) = \int_a^b p_j(t) dt \quad \forall j = 0, \dots, n-1 \quad \Leftrightarrow \quad Q_n \text{ has order } \geq n. \quad (5.3.7)$$

➤  $n \times n$  linear system of equations, see (5.3.14) for an example:

$$\begin{bmatrix} p_0(c_1) & \dots & p_0(c_n) \\ \vdots & & \vdots \\ p_{n-1}(c_1) & \dots & p_{n-1}(c_n) \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} \int_a^b p_0(t) dt \\ \vdots \\ \int_a^b p_{n-1}(t) dt \end{bmatrix}. \quad (5.3.8)$$

For instance, for the computation of quadrature weights, one may choose the monomial basis  $p_j(t) = t^j$ .

---

### Example 5.3.9 (Orders of simple polynomial quadrature formulas)

From the order rule for polynomial quadrature rule we immediately conclude the orders of simple representatives.

$n$	Order
1 midpoint rule	2
2 trapezoidal rule (5.2.5)	2
3 Simpson rule (5.2.6)	4
4 $\frac{3}{8}$ -rule	4
5 Milne rule	6

The orders for even  $n$  surpass the predictions of Thm. 5.3.5 by 1, which can be verified by straightforward computations; following Def. 5.3.1 check the exactness of the quadrature rule on  $[0, 1]$  (this is sufficient  $\rightarrow$  Cor. 5.3.4) for monomials  $\{t \mapsto t^k\}, k = 0, \dots, q - 1$ , which form a basis of  $\mathcal{P}_q$ , where  $q$  is the order that is to be confirmed: essentially one has to show

$$Q(\{t \mapsto t^k\}) = \sum_{j=1}^n w_j c_j^k = \frac{1}{k+1}, \quad k = 0, \dots, q-1, \quad (5.3.10)$$

where  $Q \triangleq$  quadrature rule on  $[0, 1]$  given by (5.1.2).

For the Simpson rule (5.2.6) we can also confirm order 4 with symbolic calculations in MAPLE:

```
> rule := 1/3*h*(f(2*h)+4*f(h)+f(0))
> err := taylor(rule - int(f(x), x=0..2*h), h=0, 6);
```

$$err := \left( \frac{1}{90} D^{(4)}(f)(0)h^5 + O(h^6), h, 6 \right)$$

➤ Composite Simpson rule possesses order 4, indeed !

---

### (5.3.11) Maximal order of $n$ -point quadrature rules

Natural question: Can an  $n$ -point quadrature formula achieve an order  $> n$  ?

A negative result limits the maximal order that can be achieved:

**Theorem 5.3.12. Maximal order of  $n$ -point quadrature rule**

*The maximal order of an  $n$ -point quadrature rule is  $2n$ .*

*Proof.* Consider a generic  $n$ -point quadrature rule according to Def. 5.1.1

$$Q_n(f) := \sum_{j=1}^n w_j^n f(c_j^n) , \quad (5.1.2)$$

We build a polynomial of degree  $2n$  that cannot be integrated exactly by  $Q_n$ . We choose polynomial

$$q(t) := (t - c_1)^2 \cdots (t - c_n)^2 \in \mathcal{P}_{2n} .$$

On the one hand,  $q$  is strictly positive almost everywhere, which means

$$\int_a^b q(t) dt > 0 .$$

On the other hand, we find a different value

$$Q_n(q) = \sum_{j=1}^n w_j^n \underbrace{q(c_j^n)}_{=0} = 0 .$$

□

Can we at least find  $n$ -point rules with order  $2n$ ?

Heuristics: A quadrature formula has order  $m \in \mathbb{N}$  already, if it is exact for  $m$  polynomials  $\in \mathcal{P}_{m-1}$  that form a basis of  $\mathcal{P}_{m-1}$  (recall Thm. 3.2.2).

↑  
An  $n$ -point quadrature formula has  $2n$  “degrees of freedom” ( $n$  node positions,  $n$  weights).

↓  
It might be possible to achieve order  $2n = \dim \mathcal{P}_{2n-1}$   
("No. of equations = No. of unknowns")

**Example 5.3.13 (2-point quadrature rule of order 4)**

Necessary & sufficient conditions for order 4, cf. (5.3.8):

$$Q_n(p) = \int_a^b p(t) dt \quad \forall p \in \mathcal{P}_3 \quad \Leftrightarrow \quad Q_n(\{t \mapsto t^q\}) = \frac{1}{q+1} (b^{q+1} - a^{q+1}) , \quad q = 0, 1, 2, 3 .$$



4 equations for weights  $w_j$  and nodes  $c_j, j = 1, 2$  ( $a = -1, b = 1$ ), cf. Rem. 5.3.6

$$\begin{aligned} \int_{-1}^1 1 dt &= 2 = 1w_1 + 1w_2 , & \int_{-1}^1 t dt &= 0 = c_1 w_1 + c_2 w_2 \\ \int_{-1}^1 t^2 dt &= \frac{2}{3} = c_1^2 w_1 + c_2^2 w_2 , & \int_{-1}^1 t^3 dt &= 0 = c_1^3 w_1 + c_2^3 w_2 . \end{aligned} \quad (5.3.14)$$

Solve using MAPLE:

```
> eqns := {seq(int(x^k, x=-1..1) = w[1]*xi[1]^k+w[2]*xi[2]^k, k=0..3)};
> sols := solve(eqns, indets(eqns, name));
> convert(sols, radical);
```

➤ weights & nodes:  $\{w_2 = 1, w_1 = 1, c_1 = 1/3\sqrt{3}, c_2 = -1/3\sqrt{3}\}$

➤ quadrature formula (order 4):  $\int_{-1}^1 f(x) dx \approx f\left(\frac{1}{\sqrt{3}}\right) + f\left(-\frac{1}{\sqrt{3}}\right)$  (5.3.15)

### (5.3.16) Construction of $n$ -point quadrature rules with maximal order $2n$

First we search for *necessary conditions* that have to be met by the nodes, if an  $n$ -point quadrature rule has order  $2n$ .

Optimist's **assumption**:  $\exists$  family of  $n$ -point quadrature formulas on  $[-1, 1]$

$$Q_n(f) := \sum_{j=1}^n w_j^n f(c_j^n) \approx \int_{-1}^1 f(t) dt, \quad w_j \in \mathbb{R}, n \in \mathbb{N},$$

of order  $2n \Leftrightarrow$  exact for polynomials  $\in \mathcal{P}_{2n-1}$ . (5.3.17)

Define  $\bar{P}_n(t) := (t - c_1^n) \cdots (t - c_n^n), \quad t \in \mathbb{R} \Rightarrow \bar{P}_n \in \mathcal{P}_n$ .

Note:  $\bar{P}_n$  has leading coefficient = 1.

By assumption on the order of  $Q_n$ : for any  $q \in \mathcal{P}_{n-1}$

$$\begin{aligned} \int_{-1}^1 \underbrace{q(t)\bar{P}_n(t)}_{\in \mathcal{P}_{2n-1}} dt &\stackrel{(5.3.17)}{=} \sum_{j=1}^n w_j^n q(c_j^n) \underbrace{\bar{P}_n(c_j^n)}_{=0} = 0. \\ \Rightarrow L^2([-1, 1])\text{-orthogonality } \int_{-1}^1 q(t)\bar{P}_n(t) dt &= 0 \quad \forall q \in \mathcal{P}_{n-1}. \end{aligned} \quad (5.3.18)$$

  $L^2([-1, 1])$ -inner product of  $q$  and  $\bar{P}_n$ , see (4.2.5).

Recall:  $(f, g) \mapsto \int_a^b f(t)g(t) dt$  is an inner product on  $C^0([a, b])$ , the  $L^2$ -inner product, see Rem. 4.2.19, [59, Sect. 4.4, Ex. 2], [34, Ex. 6.5]

- Treat space of polynomials  $\mathcal{P}_n$  as a vector space equipped with an inner product.
- As we have seen in Section 4.2.2, abstract techniques for vector spaces with inner product can be applied to polynomials, for instance **Gram-Schmidt orthogonalization**, cf. § 4.2.16, [59, Thm. 4.8], [34, Alg. 6.1].

Now carry out the abstract Gram-Schmidt orthogonalization according to Algorithm (4.2.17) and recall Thm. 4.2.18: in a vector space with inner product  $\cdot$  orthogonal vectors  $q_0, q_1, \dots$  spanning the same subspaces as the linearly independent vectors  $v_0, v_1, \dots$  are constructed recursively via

$$q_{n+1} := v_{n+1} - \sum_{k=0}^n \frac{v_{n+1} \cdot q_k}{q_k \cdot q_k} q_k , \quad q_0 := v_0 . \quad (5.3.19)$$

➤ Construction of  $\bar{P}_n$  by **Gram-Schmidt orthogonalization** of monomial basis  $\{1, t, t^2, \dots, t^{n-1}\}$  (the  $v_k$ s!) of  $\mathcal{P}_{n-1}$  w.r.t.  $L^2([-1, 1])$ -inner product:

$$\bar{P}_0(t) := 1 , \quad \bar{P}_{n+1}(t) = t^n - \sum_{k=0}^n \frac{\int_{-1}^1 t^n \bar{P}_k(t) dt}{\int_{-1}^1 \bar{P}_k^2(t) dt} \cdot \bar{P}_k(t) \quad (5.3.20)$$

Note:  $\bar{P}_n$  has leading coefficient  $= 1 \Rightarrow \bar{P}_n$  uniquely defined (up to sign) by (5.3.20).

The considerations so far only reveal *necessary conditions* on the nodes of an  $n$ -point quadrature rule of order  $2n$ :

The nodes of an  $n$ -point quadrature formula, if it exists, must coincide with the zeros of the polynomial  $\bar{P}_n$  defined recursively by (5.3.20).

They do by no means confirm the existence of such rules, but offer a clear hint on how to construct them:

### Theorem 5.3.21. Existence of $n$ -point quadrature formulas of order $2n$

Let  $\{\bar{P}_n\}_{n \in \mathbb{N}_0}$  be a family of non-zero polynomials that satisfies

- $\bar{P}_n \in \mathcal{P}_n$ ,
- $\int_{-1}^1 q(t) \bar{P}_n(t) dt = 0$  for all  $q \in \mathcal{P}_{n-1}$  ( $L^2([-1, 1])$ -orthogonality),
- The set  $\{c_j^n\}_{j=1}^m$ ,  $m \leq n$ , of real zeros of  $\bar{P}_n$  is contained in  $[-1, 1]$ .

Then the quadrature rule ( $\rightarrow$  Def. 5.1.1)

$$Q_n(f) := \sum_{j=1}^m w_j^n f(c_j^n)$$

with weights chosen according to Thm. 5.3.5 provides a quadrature formula of order  $2n$  on  $[-1, 1]$ .

*Proof.* Conclude from the orthogonality of the  $\bar{P}_n$  that  $\{\bar{P}_k\}_{k=0}^n$  is a basis of  $\mathcal{P}_n$  and

$$\int_{-1}^1 h(t) \bar{P}_n(t) dt = 0 \quad \forall h \in \mathcal{P}_{n-1} . \quad (5.3.22)$$

Recall division of polynomials with remainder (Euclid's algorithm  $\rightarrow$  Course "Diskrete Mathematik"): for any  $p \in \mathcal{P}_{2n-1}$

$$p(t) = h(t) \bar{P}_n(t) + r(t) , \quad \text{for some } h \in \mathcal{P}_{n-1}, r \in \mathcal{P}_{n-1} . \quad (5.3.23)$$

Apply this representation to the integral:

$$\int_{-1}^1 p(t) dt = \underbrace{\int_{-1}^1 h(t) \bar{P}_n(t) dt}_{=0 \text{ by (5.3.22)}} + \int_{-1}^1 r(t) dt \stackrel{(*)}{=} \sum_{j=1}^m w_j^n r(c_j^n) , \quad (5.3.24)$$

(\*) by choice of weights according to Rem. 5.3.6  $\bar{Q}_n$  is exact for polynomials of degree  $\leq n - 1$ !

By choice of nodes as zeros of  $\bar{P}_n$  using (5.3.22):

$$\sum_{j=1}^m w_j^n p(c_j^n) \stackrel{(5.3.23)}{=} \sum_{j=1}^m w_j^n h(c_j^n) \underbrace{\bar{P}_n(c_j^n)}_{=0} + \sum_{j=1}^m w_j^n r(c_j^n) \stackrel{(5.3.24)}{=} \int_{-1}^1 p(t) dt.$$

□

### (5.3.25) Legendre polynomials and Gauss-Legendre quadrature

The family of polynomials  $\{\bar{P}_n\}_{n \in \mathbb{N}_0}$  are so-called **orthogonal polynomials** w.r.t. the  $L^2([-1, 1])$ -inner product, see Def. 4.2.24. We have made use of orthogonal polynomials already in Section 4.2.2.  $L^2([-1, 1])$ -orthogonal polynomials play a key role in analysis.

#### Definition 5.3.26. Legendre polynomials

The  $n$ -th **Legendre polynomial**  $P_n$  is defined by

- $P_n \in \mathcal{P}_n$ ,
- $\int_{-1}^1 P_n(t) q(t) dt = 0 \forall q \in \mathcal{P}_{n-1}$ ,
- $P_n(1) = 1$ .

Legendre polynomials  $P_0, \dots, P_5$

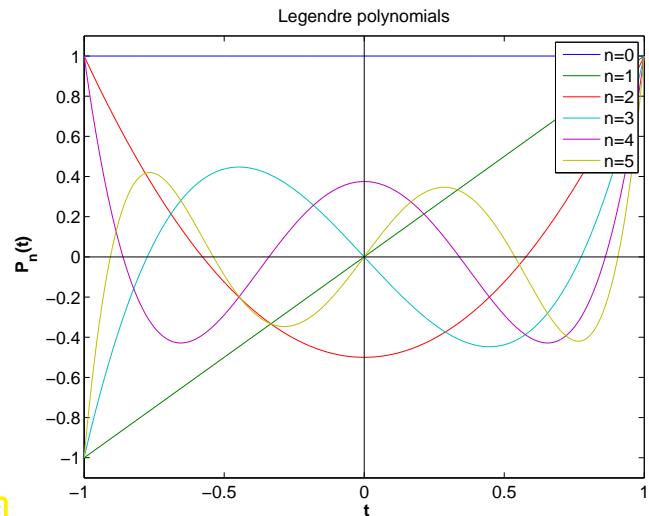


Fig. 172

Notice: the polynomials  $\bar{P}_n$  defined by (5.3.20) and the Legendre polynomials  $P_n$  of Def. 5.3.26 (merely) differ by a constant factor!



Gauss points  $\xi_j^n =$  zeros of Legendre polynomial  $P_n$

Note: the above considerations, recall (5.3.18), show that the nodes of an  $n$ -point quadrature formula of order  $2n$  on  $[-1, 1]$  must agree with the zeros of  $L^2([-1, 1])$ -orthogonal polynomials.



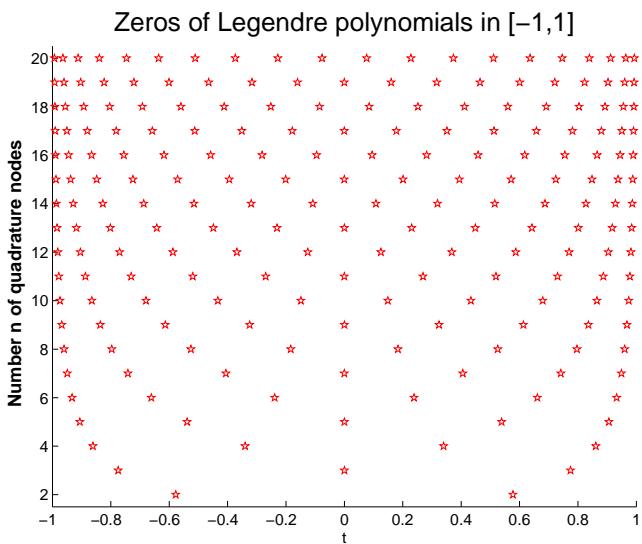
$n$ -point quadrature formulas of order  $2n$  are unique

This is not surprising in light of “ $2n$  equations for  $2n$  degrees of freedom”.



We are not done yet: the zeros of  $\bar{P}_n$  from (5.3.20) may lie outside  $[-1, 1]$ . In principle  $\bar{P}_n$  could also have less than  $n$  real zeros.

The next lemma shows that all this cannot happen.



▷ Obviously:

**Lemma 5.3.27. Zeros of Legendre polynomials**

$P_n$  has  $n$  distinct zeros in  $[-1, 1]$ .

Zeros of Legendre polynomials = Gauss points

*Proof.* (indirect) Assume that  $P_n$  has only  $m < n$  zeros  $\zeta_1, \dots, \zeta_m$  in  $[-1, 1]$  at which it changes sign. Define

$$\begin{aligned} q(t) := \prod_{j=1}^m (t - \zeta_j) &\Rightarrow qP_n \geq 0 \quad \text{or} \quad qP_n \leq 0. \\ &\Rightarrow \int_{-1}^1 q(t)P_n(t) dt \neq 0. \end{aligned}$$

As  $q \in \mathcal{P}_{n-1}$ , this contradicts (5.3.22). □

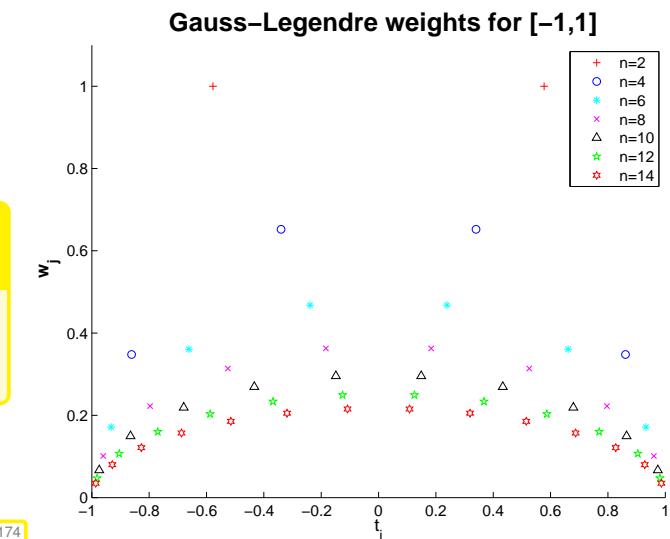
### Definition 5.3.28. Gauss-Legendre quadrature formulas

The  $n$ -point Quadrature formulas whose nodes, the **Gauss points**, are given by the zeros of the  $n$ -th Legendre polynomial (→ Def. 5.3.26), and whose weights are chosen according to Thm. 5.3.5, are called **Gauss-Legendre quadrature formulas**.

Obviously

### Lemma 5.3.29. Positivity of Gauss-Legendre quadrature weights

The weights of the Gauss-Legendre quadrature formulas are positive.



*Proof.* Writing  $\xi_j^n$ ,  $j = 1, \dots, n$ , for the nodes (Gauss points) of the  $n$ -point Gauss-Legendre quadrature formula,  $n \in \mathbb{N}$ , we define

$$q_k(t) = \prod_{\substack{j=1 \\ j \neq k}}^n (t - \xi_j^n)^2 \Rightarrow q_k \in \mathcal{P}_{2n-2}.$$

This polynomial is integrated exactly by the quadrature rule: since  $q_k(\xi_j^n) = 0$  for  $j \neq k$

$$0 < \int_{-1}^1 q(t) dt = w_k^n \underbrace{q(\xi_k^n)}_{>0} ,$$

where  $w_j^n$  are the quadrature weights. □

### Remark 5.3.30 (3-Term recursion for Legendre polynomials)

From Thm. 4.2.31 we learn the orthogonal polynomials satisfy the 3-term recursion (4.2.32). To keep this chapter self-contained we derive it independently for Legendre polynomials.

Note: the polynomials  $\bar{P}_n$  from (5.3.20) are uniquely characterized by the two properties (try a proof!)

- (i)  $\bar{P}_n \in \mathcal{P}_n$  with leading coefficient 1:  $\bar{P}(t) = t^n + \dots$ ,
- (ii)  $\int_{-1}^1 \bar{P}_k(t) \bar{P}_j(t) dt = 0$ , if  $j \neq k$  ( $L^2([-1, 1])$ -orthogonality).

➤ we get the same polynomials  $\bar{P}_n$  by another Gram-Schmidt orthogonalization procedure, cf. (5.3.19) and § 4.2.28:

$$\bar{P}_{n+1}(t) = t\bar{P}_n(t) - \sum_{k=0}^n \frac{\int_{-1}^1 \tau \bar{P}_n(\tau) \bar{P}_k(\tau) d\tau}{\int_{-1}^1 \bar{P}_k^2(\tau) d\tau} \cdot \bar{P}_k(t)$$

By the orthogonality property (5.3.22) the sum collapses, since

$$\int_{-1}^1 \tau \bar{P}_n(\tau) \bar{P}_k(\tau) d\tau = \int_{-1}^1 \bar{P}_n(\tau) \underbrace{(\tau \bar{P}_k(\tau))}_{\in \mathcal{P}_{k+1}} d\tau = 0 ,$$

if  $k+1 < n$ :

$$\bar{P}_{n+1}(t) = t\bar{P}_n(t) - \frac{\int_{-1}^1 \tau \bar{P}_n(\tau) \bar{P}_n(\tau) d\tau}{\int_{-1}^1 \bar{P}_n^2(\tau) d\tau} \cdot \bar{P}_n(t) - \frac{\int_{-1}^1 \tau \bar{P}_n(\tau) \bar{P}_{n-1}(\tau) d\tau}{\int_{-1}^1 \bar{P}_{n-1}^2(\tau) d\tau} \cdot \bar{P}_{n-1}(t) . \quad (5.3.31)$$

After rescaling (tedious!): 3-term recursion for Legendre polynomials

$$P_{n+1}(t) := \frac{2n+1}{n+1} t P_n(t) - \frac{n}{n+1} P_{n-1}(t) , \quad P_0 := 1 , \quad P_1(t) := t . \quad (5.3.32)$$

Reminder (→ Section 4.1.3.1): we have a similar 3-term recursion (4.1.69) for Chebychev polynomials. Coincidence? Of course not, nothing in mathematics holds “by accident”. By Thm. 4.2.31 3-term recursions are a distinguishing feature of so-called families of **orthogonal polynomials**, to which the Chebychev polynomials belong as well, spawned by Gram-Schmidt orthogonalization with respect to a weighted  $L^2$ -inner product, however, see [42, VI].

- Efficient and *stable* evaluation of Legendre polynomials by means of 3-term recursion (5.3.32), cf. the analogous algorithm for Chebychev polynomials given in Code 4.1.70.

**MATLAB-code 5.3.33: computing Legendre polynomials**

```

1 function V= legendre(n,x)
2 V = ones(size(x)); V = [V; x];
3 for j=1:n-1
4   V = [V; ((2*j+1)/(j+1)).*x.*V(end,:)-j/(j+1)*V(end-1,:)];
5 end

```

Comments on Code 5.3.33:

- return value: matrix  $\mathbf{V}$  with  $(\mathbf{V})_{ij} = P_i(x_j)$
- line 2: takes into account initialization of Legendre 3-term recursion (5.3.32)

**Remark 5.3.34 (Computing Gauss nodes and weights)**

There are several efficient ways to find the Gauss points. Here we discuss an intriguing connection with an eigenvalue problem.

Compute nodes/weights of Gaussian quadrature by solving an eigenvalue problem!

(**Golub-Welsch algorithm** [24, Sect. 3.5.4], [79, Sect. 1])

In codes Gauss nodes and weights are usually retrieved from tables, cf. Rem. 5.1.7.

**MATLAB-code 5.3.35: Golub-Welsch algorithm**

```

1 function [x,w]=gaussquad(n)
2 % Computation of weights and nodes of n-point
3 % Gaussian quadrature rule on [-1,1].
4 if (n==1), x = 0; w = 2;
5 else
6   b = zeros(n-1,1); @\label{gw:1}@
7   for i=1:(n-1), b(i)=i/sqrt(4*i*i-1);
8     end@\label{gw:2}@
9   J=diag(b,-1)+diag(b,1);
10  [ev,ew]=eig(J);@\label{gw:3}@
11  x=diag(ew);
12  w=(2*(ev(1,:).*ev(1,:)))';@\label{gw:4}@
13 end

```

Justification: rewrite 3-term recurrence (5.3.32) for scaled Legendre polynomials  $\tilde{P}_n = \frac{1}{\sqrt{n+1/2}} P_n$

$$t\tilde{P}_n(t) = \underbrace{\frac{n}{\sqrt{4n^2-1}}}_{=: \beta_n} \tilde{P}_{n-1}(t) + \underbrace{\frac{n+1}{\sqrt{4(n+1)^2-1}}}_{=: \beta_{n+1}} \tilde{P}_{n+1}(t). \quad (5.3.36)$$

For fixed  $t \in \mathbb{R}$  (5.3.36) can be expressed as

$$t \begin{bmatrix} \tilde{P}_0(t) \\ \tilde{P}_1(t) \\ \vdots \\ \tilde{P}_{n-1}(t) \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & \beta_1 & & & \\ \beta_1 & 0 & \beta_2 & & \\ & \beta_2 & \ddots & \ddots & \\ & & \ddots & \ddots & \ddots \\ & & & 0 & \beta_{n-1} \\ & & & & \beta_{n-1} & 0 \end{bmatrix}}_{=: \mathbf{J}_n \in \mathbb{R}^{n,n}} \begin{bmatrix} \tilde{P}_0(t) \\ \tilde{P}_1(t) \\ \vdots \\ \tilde{P}_{n-1}(t) \end{bmatrix} + \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \beta_n \tilde{P}_n(t) \end{bmatrix}$$



$$\widetilde{P}_n(\xi) = 0 \Leftrightarrow \xi \mathbf{p}(\xi) = \mathbf{J}_n \mathbf{p}(\xi) \quad (\text{eigenvalue problem!}) .$$

► The zeros of  $P_n$  can be obtained as the  $n$  real eigenvalues of the symmetric tridiagonal matrix  $\mathbf{J}_n \in \mathbb{R}^{n,n}$ !

This matrix  $\mathbf{J}_n$  is initialized in ?? of Code 5.3.35. The computation of the weights in ?? of Code 5.3.35 is explained in [24, Sect. 3.5.4].

### (5.3.37) Quadrature error and best approximation error

The positivity of the weights  $w_j^n$  for all  $n$ -point Gauss-Legendre and Clenshaw-Curtis quadrature rules has important consequences.

#### Theorem 5.3.38. Quadrature error estimate for quadrature rules with positive weights

For every  $n$ -point quadrature rule  $Q_n$  as in (5.1.2) of order  $q \in \mathbb{N}$  with weights  $w_j \geq 0, j = 1, \dots, n$  the quadrature error satisfies

$$E_n(f) := \left| \int_a^b f(t) dt - Q_n(f) \right| \leq 2|b-a| \underbrace{\inf_{p \in \mathcal{P}_{k-1}} \|f - p\|_{L^\infty([a,b])}}_{\text{best approximation error}} \quad \forall f \in C^0([a,b]) . \quad (5.3.39)$$

*Proof.* The proof runs parallel to the derivation of (4.1.53). Writing  $E_n(f)$  for the quadrature error, the left hand side of (5.3.39), we find by the definition Def. 5.3.1 of the order of a quadrature rule

$$\begin{aligned} E_n(f) &= E_n(f - p) \leq \left| \int_a^b (f - p)(t) dt \right| + \left| \sum_{j=1}^n w_j (f - p)(c_j) \right| \\ &\leq |b-a| \|f - p\|_{L^\infty([a,b])} + \left( \sum_{j=1}^n |w_j| \right) \|f - p\|_{L^\infty([a,b])} . \end{aligned} \quad (5.3.40)$$

Since the quadrature rule is exact for constants and  $w_j \geq 0$

$$\sum_{j=1}^n |w_j| = \sum_{j=1}^n w_j = |b-a| ,$$

which finishes the proof. □

Drawing on best approximation estimates from Section 4.1.1 and Rem. 4.1.87, we immediately get results about the asymptotic decay of the quadrature error for  $n$ -point Gauss-Legendre and Clenshaw-Curtis quadrature as  $n \rightarrow \infty$ :

$$\begin{aligned} f \in C^r([a,b]) &\Rightarrow E_n(f) \rightarrow 0 \text{ algebraically with rate } r, \\ f \text{ "analytically extensible"} &\Rightarrow E_n(f) \rightarrow 0 \text{ exponentially,} \end{aligned}$$

as  $n \rightarrow \infty$ , see Def. 4.1.31 for type of convergence.

Appealing to Thm. 4.1.11 and Rem. 4.1.19, and (4.1.43), the dependence of the constants on the length of the integration interval can be quantified for integrands with limited smoothness.

### Lemma 5.3.41. Quadrature error estimates for $C^r$ -integrands

Under the assumptions of Thm. 5.3.38 and with the notations introduced there, we find for  $f \in C^r([a, b])$ ,  $r \in \mathbb{N}_0$ , that the quadrature error  $E_n(f)$  satisfies

$$\text{in the case } q \geq r: \quad E_n(f) \leq C q^{-r} |b-a|^{r+1} \|f^{(r)}\|_{L^\infty([a,b])}, \quad (5.3.42)$$

$$\text{in the case } q < r: \quad E_n(f) \leq \frac{|b-a|^{q+1}}{q!} \|f^{(q)}\|_{L^\infty([a,b])}, \quad (5.3.43)$$

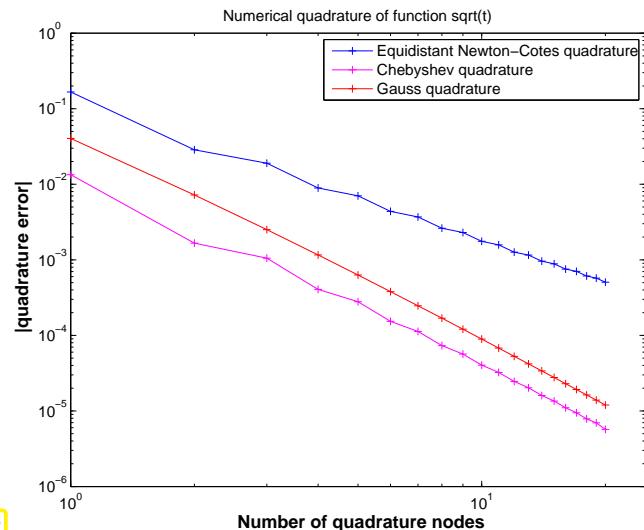
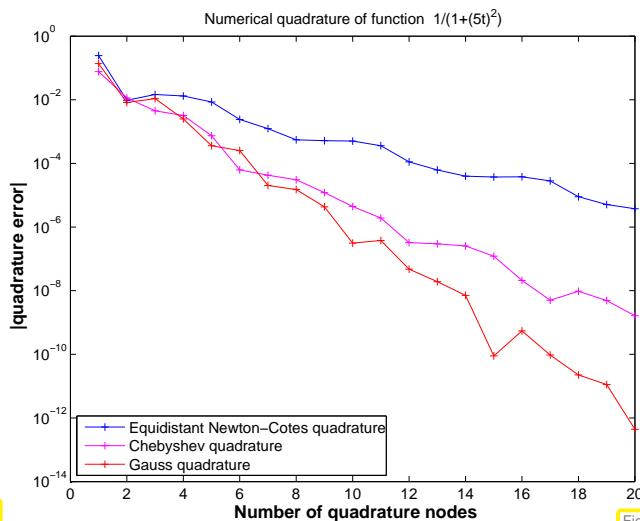
with a constant  $C > 0$  independent of  $n$ ,  $f$ , and  $[a, b]$ .

Please note the different estimates depending on whether the smoothness of  $f$  (as described by  $r$ ) or the order of the quadrature rule is the “limiting factor”.

### Example 5.3.44 (Convergence of global quadrature rules)

We examine three families of global polynomial ( $\rightarrow$  Thm. 5.3.5) quadrature rules: Newton-Cotes formulas, Gauss-Legendre rules, and Clenshaw-Curtis rules. We record the convergence of the quadrature errors for the interval  $[0, 1]$  and two different functions

1.  $f_1(t) = \frac{1}{1+(5t)^2}$ , an analytic function, see Rem. 4.1.64,
2.  $f_2(t) = \sqrt{t}$ , merely continuous, derivatives singular in  $t = 0$ .



quadrature error,  $f_1(t) := \frac{1}{1+(5t)^2}$  on  $[0, 1]$

quadrature error,  $f_2(t) := \sqrt{t}$  on  $[0, 1]$

Asymptotic behavior of quadrature error  $\epsilon_n := \left| \int_0^1 f(t) dt - Q_n(f) \right|$  for " $n \rightarrow \infty$ ":

- exponential convergence  $\epsilon_n \approx O(q^n)$ ,  $0 < q < 1$ , for  $C^\infty$ -integrand  $f_1 \rightsquigarrow$ : Newton-Cotes quadrature :  $q \approx 0.61$ , Clenshaw-Curtis quadrature :  $q \approx 0.40$ , Gauss-Legendre quadrature :  $q \approx 0.27$
- algebraic convergence  $\epsilon_n \approx O(n^{-\alpha})$ ,  $\alpha > 0$ , for integrand  $f_2$  with singularity at  $t = 0 \rightsquigarrow$  Newton-Cotes quadrature :  $\alpha \approx 1.8$ , Clenshaw-Curtis quadrature :  $\alpha \approx 2.5$ , Gauss-Legendre quadrature :  $\alpha \approx 2.7$

**Remark 5.3.45 (Removing a singularity by transformation)**

From Ex. 5.3.44 teaches us that a lack of smoothness of the integrand can thwart exponential convergence and severely limits the rate of algebraic convergence of a global quadrature rule for  $n \rightarrow \infty$ .

Idea: recover integral with smooth integrand by “analytic preprocessing”

Here is an example:

For a general but smooth  $f \in C^\infty([0, b])$  compute  $\int_0^b \sqrt{t}f(t) dt$  via a quadrature rule, e.g.,  $n$ -point Gauss-Legendre quadrature on  $[0, b]$ . Due to the presence of a square-root singularity at  $t = 0$  the direct application of  $n$ -point Gauss-Legendre quadrature will result in a rather slow algebraic convergence of the quadrature error as  $n \rightarrow \infty$ , see Ex. 5.3.44.

Trick: Transformation of integrand by substitution rule:

$$\text{substitution } s = \sqrt{t}: \int_0^b \sqrt{t}f(t) dt = \int_0^{\sqrt{b}} 2s^2 f(s^2) ds. \quad (5.3.46)$$

Then: Apply Gauss-Legendre quadrature rule to smooth integrand

**Remark 5.3.47 (The message of asymptotic estimates)**

There is one blot on most  $n$ -asymptotic estimates obtained from Thm. 5.3.38: the bounds usually involve quantities like norms of higher derivatives of the interpolant that are elusive in general, in particular for integrands given only in procedural form, see § 5.0.2. Such unknown quantities are often hidden in “generic constants  $C$ ”. Can we extract useful information from estimates marred by the presence of such constants?

For fixed integrand  $f$  let us assume *sharp algebraic convergence* (in  $n$ ) with rate  $r \in \mathbb{N}$  of the quadrature error  $E_n(f)$  for a family of  $n$ -point quadrature rules:

$$E_n(f) = O(n^{-r}) \stackrel{\text{sharp}}{\implies} E_n(f) \approx Cn^{-r}, \quad (5.3.48)$$

with a “generic constant  $C > 0$ ” *independent* of  $n$ .

Goal: Reduction of the quadrature error by a factor of  $\rho > 1$

Which (minimal) increase in the number  $n$  of quadrature points accomplishes this?

$$\frac{Cn_{\text{old}}^{-r}}{Cn_{\text{new}}^{-r}} = \rho \Leftrightarrow n_{\text{new}} : n_{\text{old}} = \sqrt[r]{\rho}. \quad (5.3.49)$$

In the case of *algebraic convergence* with rate  $r \in \mathbb{R}$  a reduction of the quadrature error by a factor of  $\rho$  is bought by an increase of the number of quadrature points by a factor of  $\rho^{1/r}$ .

(5.3.42)  $\Rightarrow$  gains in accuracy are “cheaper” for smoother integrands!

Now assume *sharp exponential convergence* (in  $n$ ) of the quadrature error  $E_n(f)$  for a family of  $n$ -point quadrature rules,  $0 \leq q < 1$ :

$$E_n(f) = O(q^n) \stackrel{\text{sharp}}{\implies} E_n(f) \approx Cq^n, \quad (5.3.50)$$

with a “generic constant  $C > 0$ ” independent of  $n$ .

Error reduction by a factor  $\rho > 1$  results from

$$\frac{Cq^{n_{\text{old}}}}{Cq^{n_{\text{new}}}} = \rho \Leftrightarrow n_{\text{new}} - n_{\text{old}} = -\frac{\log \rho}{\log q}.$$

In the case of *exponential convergence* (5.3.50) a *fixed* increase of the number of quadrature points by  $-\log \rho : \log q$  results in a reduction of the quadrature error by a factor of  $\rho > 1$ .

## 5.4 Composite Quadrature

In 4, Section 4.5.1 we studied approximation by piecewise polynomial interpolants. A similar idea underlies the so-called composite quadrature rules on an interval  $[a, b]$ . Analogously to piecewise polynomial techniques they start from a grid/mesh

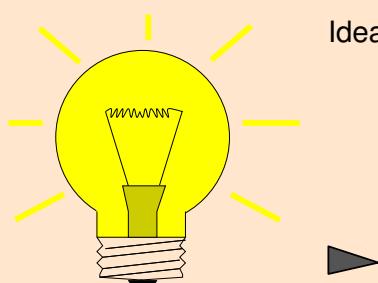
$$\mathcal{M} := \{a = x_0 < x_1 < \dots < x_{m-1} < x_m = b\} \quad (4.5.2)$$

and appeal to the trivial identity

$$\int_a^b f(t) dt = \sum_{j=1}^m \int_{x_{j-1}}^{x_j} f(t) dt. \quad (5.4.1)$$

On each mesh interval  $[x_{j-1}, x_j]$  we then use a *local quadrature rule*, which may be one of the polynomial quadrature formulas from 5.2.

### General construction of composite quadrature rules



Idea: Partition integration domain  $[a, b]$  by a *mesh/grid* ( $\rightarrow$  Section 4.5)

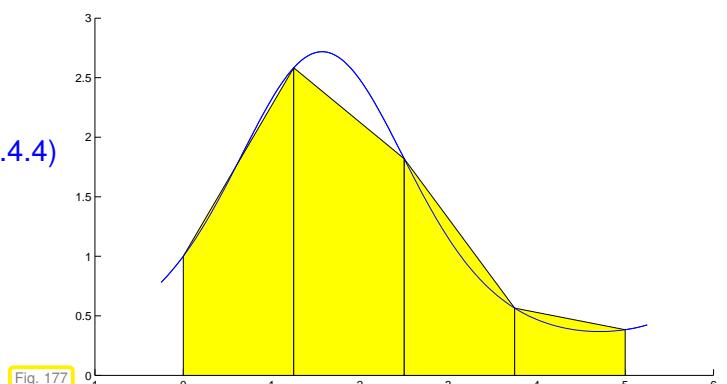
- $\mathcal{M} := \{a = x_0 < x_1 < \dots < x_m = b\}$
- Apply quadrature formulas from Section 5.2, Section 5.3 *locally* on mesh intervals  $I_j := [x_{j-1}, x_j]$ ,  $j = 1, \dots, m$ , and sum up.

composite quadrature rule

### Example 5.4.3 (Simple composite polynomial quadrature rules)

Composite trapezoidal rule, cf. (5.2.5)

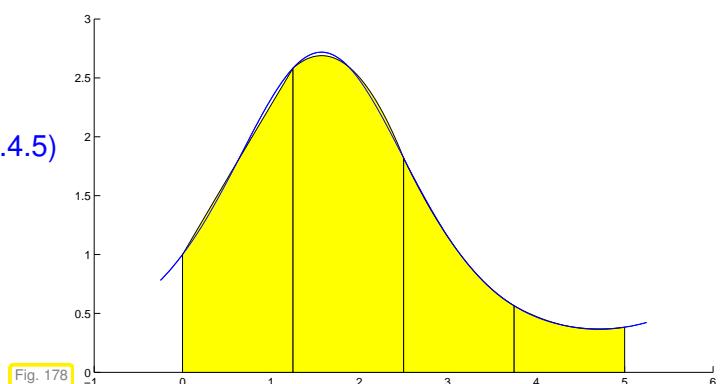
$$\int_a^b f(t) dt = \frac{1}{2}(x_1 - x_0)f(a) + \sum_{j=1}^{m-1} \frac{1}{2}(x_{j+1} - x_{j-1})f(x_j) + \frac{1}{2}(x_m - x_{m-1})f(b). \quad (5.4.4)$$



➤ arising from piecewise linear interpolation of  $f$ .

Composite Simpson rule, cf. (5.2.6)

$$\int_a^b f(t) dt = \frac{1}{6}(x_1 - x_0)f(a) + \sum_{j=1}^{m-1} \frac{1}{6}(x_{j+1} - x_{j-1})f(x_j) + \sum_{j=1}^m \frac{2}{3}(x_j - x_{j-1})f(\frac{1}{2}(x_j + x_{j-1})) + \frac{1}{6}(x_m - x_{m-1})f(b). \quad (5.4.5)$$



➤ related to piecewise quadratic Lagrangian interpolation.

Formulas (5.4.4), (5.4.5) directly suggest efficient implementation with minimal number of  $f$ -evaluations.

#### MATLAB-code 5.4.6: Equidistant composite trapezoidal rule (5.4.4)

```

1 function res = trapezoidal(fnct,a,b,N)
2 % Numerical quadrature based on trapezoidal rule
3 % fnct handle to y = f(x)
4 % a,b bounds of integration interval
5 % N+1 = number of equidistant integration points (can be a vector)
6 res = [];
7 for n = N
8     h = (b-a)/n; x = (a:h:b); w = [0.5 ones(1,n-1) 0.5];
9     res = [res; h, h*dot(w,feval(fnct,x))];
10 end
```

#### MATLAB-code 5.4.7: Equidistant composite Simpson rule (5.4.5)

```

1 function res = simpson(fnct,a,b,N)
2 % Numerical quadrature based on Simpson rule
```

```

3 % fnct handle to y = f(x)
4 % a,b bounds of integration interval
5 % N+1 = number of equidistant integration points (can be a vector)
6
7 res = [];
8 for n = N
9     h = (b-a)/n;
10    x = (a:h/2:b);
11    fv = feval(fnct,x);
12    val = sum(h*(fv(1:2:end-2)+4*fv(2:2:end-1)+fv(3:2:end)))/6;
13    res = [res; h, val];
14 end

```

Note: `fnct` is supposed to accept vector arguments and return the function value for each vector component!

### Remark 5.4.8 (Composite quadrature and piecewise polynomial interpolation)

Composite quadrature scheme based on local polynomial quadrature can usually be understood as “quadrature by approximation schemes” as explained in § 5.1.8. The underlying approximation schemes belong to the class of general local Lagrangian interpolation schemes introduced in Section 4.5.1.

In other words, many composite quadrature schemes arise from replacing the integrand by a piecewise interpolating polynomial, see Fig. 177 and Fig. 178 and compare with Fig. 158.

To see the main rationale behind the use of composite quadrature rules recall Lemma 5.3.41: for a polynomial quadrature rule (5.2.1) of order  $q$  with positive weights and  $f \in C^r([a, b])$  the quadrature error shrinks with the  $\min\{r, q\} + 1$ -st power of the length  $|b - a|$  of the integration domain! Hence, applying polynomial quadrature rules to small mesh intervals should lead to a small overall quadrature error.

### (5.4.9) Quadrature error estimate for composite polynomial quadrature rules

Assume a composite quadrature rule  $Q$  on  $[x_0, x_m]$  based on local quadrature rules  $Q_{n_j}^j$  with positive weights and of fixed orders  $q_j \in \mathbb{N}$  on each mesh interval  $[x_{j-1}, x_j]$ . From Lemma 5.3.41 recall the estimate for  $f \in C^r([x_{j-1}, x_j])$

$$\left| \int_{x_{j-1}}^{x_j} f(t) dt - Q_{n_j}^j(f) \right| \leq C |x_j - x_{j-1}|^{\min\{r, q_j\} + 1} \|f^{(\min\{r, q_j\})}\|_{L^\infty([x_{j-1}, x_j])}. \quad (5.2.10)$$

with  $C > 0$  independent of  $f$  and  $j$ .

For  $f \in C^r([a, b])$ , summing up these bounds we get for the global quadrature error

$$\left| \int_{x_0}^{x_m} f(t) dt - Q(f) \right| \leq C \sum_{j=1}^m h_j^{\min\{r, q_j\} + 1} \|f^{(\min\{r, q_j\})}\|_{L^\infty([x_{j-1}, x_j])},$$

with local meshwidths  $h_j = x_j - x_{j-1}$ . If  $q_j = q$ ,  $q \in \mathbb{N}$ , for all  $j = 1, \dots, m$ , then

$$\left| \int_{x_0}^{x_m} f(t) dt - Q(f) \right| \leq C h_M^{\min\{q,r\}} |b-a| \|f^{(\min\{q,r\})}\|_{L^\infty([a,b])}, \quad (5.4.10)$$

with (global) meshwidth  $h_M := \max_j h_j$ .

(5.4.10)  $\longleftrightarrow$  Algebraic convergence in no. of  $f$ -evaluations for  $n \rightarrow \infty$

### (5.4.11) Constructing families of composite quadrature rules

As with polynomial quadrature rules, we study the asymptotic behavior of the quadrature error for families of composite quadrature rules as a function on the total number  $n$  of function evaluations.

As in the case of  $M$ -piecewise polynomial approximation of function ( $\rightarrow$  Section 4.5.1) families of composite quadrature rules can be generated in two different ways:

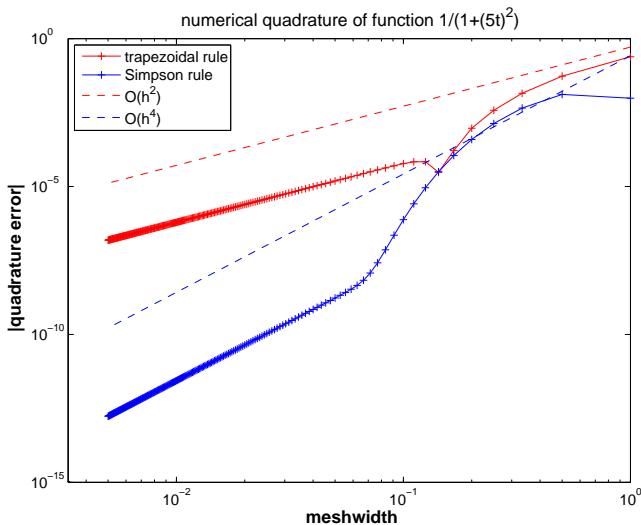
- (I) use a sequence of successively refined meshes  $(M_k = \{x_j^k\}_j)_{k \in \mathbb{N}}$  with  $\#M = m(k) + 1$ ,  $m(k) \rightarrow \infty$  for  $k \rightarrow \infty$ , combined with the same (transformed,  $\rightarrow$  Rem. 5.1.4) local quadrature rule on all mesh intervals  $[x_{j-1}^k, x_j^k]$ . Examples are the composite trapezoidal rule and composite Simpson rule from Ex. 5.4.3 on sequences of equidistant meshes.  
➤ h-convergence
- (II) On a fixed mesh  $M = \{x_j\}_{j=0}^m$ , on each cell use the same (transformed) local quadrature rule taken from a sequence of polynomial quadrature rules of increasing order.  
➤ p-convergence

### Example 5.4.12 (Quadrature errors for composite quadrature rules)

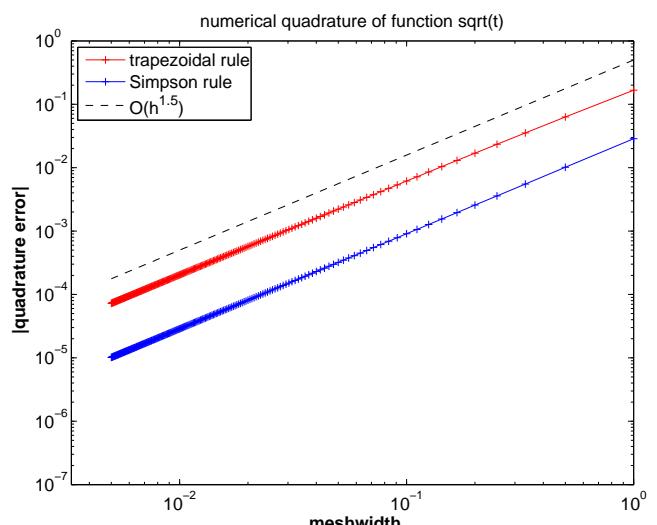
Composite quadrature rules based on

- trapezoidal rule (5.2.5) ➤ local order 2 (exact for linear functions, see Ex. 5.3.9),
- Simpson rule (5.2.6) ➤ local order 4 (exact for cubic polynomials, see Ex. 5.3.9)

on equidistant mesh  $\mathcal{M} := \{jh\}_{j=0}^n$ ,  $h = 1/n$ ,  $n \in \mathbb{N}$ .



$$\text{quadrature error, } f_1(t) := \frac{1}{1+(5t)^2} \text{ on } [0, 1]$$



$$\text{quadrature error, } f_2(t) := \sqrt{t} \text{ on } [0, 1]$$

Asymptotic behavior of quadrature error  $E(n) := \left| \int_0^1 f(t) dt - Q_n(f) \right|$  for meshwidth " $h \rightarrow 0$ "

- ☛ algebraic convergence  $E(n) = O(h^\alpha)$  of order  $\alpha > 0$ ,  $n = h^{-1}$
  - Sufficiently smooth integrand  $f_1$ : trapezoidal rule  $\rightarrow \alpha = 2$ , Simpson rule  $\rightarrow \alpha = 4$  !?
  - singular integrand  $f_2$ :  $\alpha = 3/2$  for trapezoidal rule & Simpson rule !
- (lack of) smoothness of integrand limits convergence!

### Remark 5.4.13 (Composite quadrature rules vs. global quadrature rules)

For a fixed integrand  $f \in C^r([a, b])$  of limited smoothness on an interval  $[a, b]$  we compare

- a family of composite quadrature rules based on single local  $\ell$ -point rule (with positive weights) of order  $q$  on a sequence of equidistant meshes  $(\mathcal{M}_k = \{x_j^k\}_{j=0}^{\ell})_{k \in \mathbb{N}}$ ,
- the family of Gauss-Legendre quadrature rules from Def. 5.3.28.

We study the asymptotic dependence of the quadrature error on the number  $n$  of function evaluations.

For the composite quadrature rules we have  $n \approx \ell \#\mathcal{M}_k \approx \ell h_M^{-1}$ . Combined with (5.4.10), we find for quadrature error  $E_n^{\text{comp}}(f)$  of the composite quadrature rules

$$E_n^{\text{comp}}(f) \leq C_1 n^{-\min\{q, r\}}, \quad (5.4.14)$$

with  $C_1 > 0$  independent of  $\mathcal{M} = \mathcal{M}_k$ .

The quadrature errors  $E_n^{\text{GL}}(f)$  of the  $n$ -point Gauss-Legendre quadrature rules are given in Lemma 5.3.41, (5.3.42):

$$E_n^{\text{GL}}(f) \leq C_2 n^{-r}, \quad (5.4.15)$$

with  $C_2 > 0$  independent of  $n$ .

Gauss-Legendre quadrature converges *at least as fast* fixed order composite quadrature on equidistant meshes.

Moreover, Gauss-Legendre quadrature “automatically detects” the smoothness of the integrand, and enjoys fast exponential convergence for analytic integrands.

Use Gauss-Legendre quadrature instead of fixed order composite quadrature on equidistant meshes.

### Experiment 5.4.16 (Empiric convergence of equidistant trapezoidal rule)

Sometimes there are surprises: Now we will witness a convergence behavior of a composite quadrature rule that is much better than predicted by the order of the local quadrature formula.

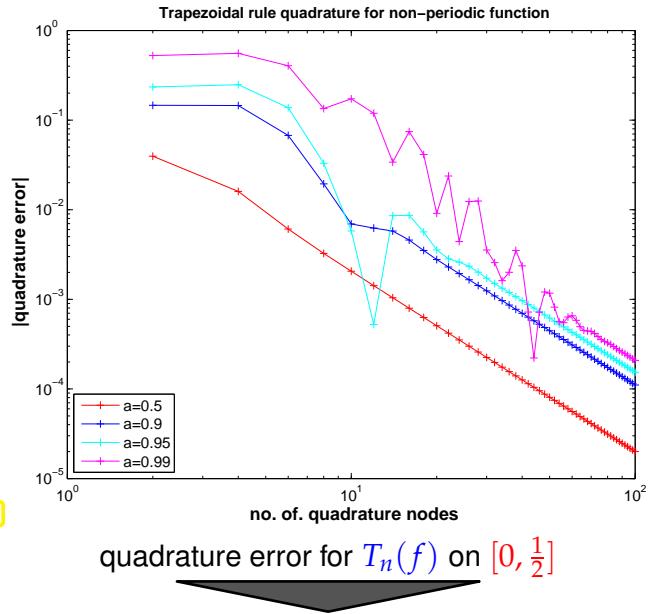
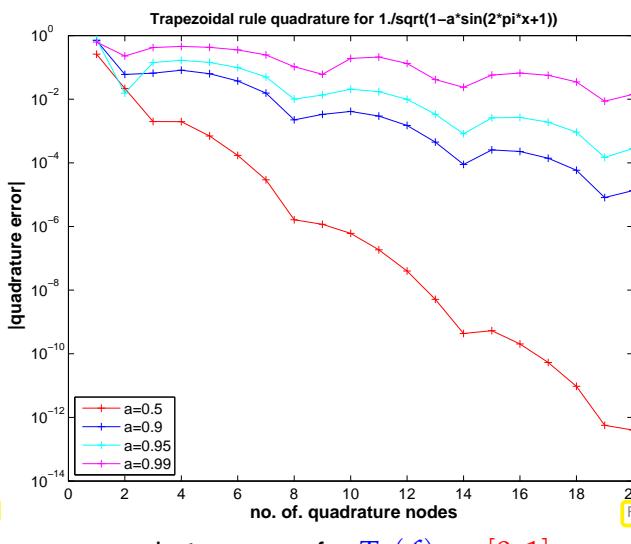
We consider Equidistant trapezoidal rule (order 2), see (5.4.4), Code 5.4.6

$$\int_a^b f(t) dt \approx T_m(f) := h \left( \frac{1}{2}f(a) + \sum_{k=1}^{m-1} f(kh) + \frac{1}{2}f(b) \right), \quad h := \frac{b-a}{m}. \quad (5.4.17)$$

and the 1-periodic smooth (analytic) integrand

$$f(t) = \frac{1}{\sqrt{1 - a \sin(2\pi t - 1)}}, \quad 0 < a < 1.$$

(“exact value of integral”: use  $T_{500}$ )



exponential convergence !!

merely algebraic convergence

### (5.4.18) Equidistant trapezoidal rule (for periodic integrands)

In this § we use  $I := [0, 1]$  as a reference interval, cf. Exp. 5.4.16. We rely on similar techniques as in Section 4.4, § 4.4.9. Again, a key tool will be the bijective mapping, see Fig. 151,

$$\Phi_{S^1} : I \rightarrow S^1 := \{z \in \mathbb{C} : |z| = 1\} \quad , \quad t \mapsto z := \exp(2\pi i t), \quad (4.4.10)$$

which induces the general pullback, *c.f.* (4.1.16),

$$(\Phi_{S^1}^{-1})^* : C^0([0,1]) \rightarrow C^0(S^1) \quad , \quad ((\Phi_{S^1}^{-1})^* f)(z) := f(\Phi_{S^1}^{-1}(z)) \quad , \quad z \in S^1 .$$

If  $f \in C^r(\mathbb{R})$  and **1-periodic**, then  $(\Phi_{S^1}^{-1})^* f \in C^r(S^1)$ . Further,  $\Phi_{S^1}$  maps equidistant nodes on  $I := [0,1]$  to equispaced nodes on  $S^1$ , which are the **roots of unity**:

$$\Phi_{S^1}\left(\frac{j}{n}\right) = \exp(2\pi i \frac{j}{n}) \quad [ \exp(2\pi i \frac{j}{n})^n = 1; ] . \quad (5.4.19)$$

Now consider an  $n$ -point polynomial quadrature rule on  $S^1$  based on the set of equidistant nodes  $\mathcal{Z} := \{z_j := \exp(2\pi i \frac{j-1}{n}), j = 1, \dots, n\}$  and defined as

$$Q_n^{S^1}(g) := \int_{S^1} (\mathcal{L}_{\mathcal{Z}} g)(\tau) dS(\tau) = \sum_{j=1}^n w_j^{S^1} g(z_j) , \quad (5.4.20)$$

where  $\mathcal{L}_{\mathcal{Z}}$  is the Lagrange interpolation operator ( $\rightarrow$  Def. 4.1.25). This means that the weights obey Thm. 5.3.5, where the definition (3.2.11) of Lagrange polynomials remains the same for complex nodes. By sheer *symmetry*, all the weights have to be the same, which, since the rule will be at least of order 1, means

$$w_j^{S^1} = \frac{2\pi}{n} , \quad j = 1, \dots, n .$$

Moreover, the quadrature rule  $Q_n^{S^1}$  will be of order  $n$ , see Def. 5.3.1, that is, it will *integrate polynomial of degree  $\leq n - 1$ , exactly*.

By transformation ( $\rightarrow$  Rem. 5.1.4 and pullback (5.4.18)),  $Q_n^{S^1}$  induces a quadrature rule on  $I := [0,1]$  by

$$Q_n^I(f) := \frac{1}{2\pi} Q_n^{S^1}((\Phi_{S^1}^{-1})^* f) = \frac{1}{2\pi} \sum_{j=1}^n w_j^{S^1} f(\Phi_{S^1}^{-1}(z_j)) = \sum_{j=1}^n \frac{1}{n} f\left(\frac{j-1}{n}\right) . \quad (5.4.21)$$

This is exactly the **equidistant trapezoidal rule** (5.4.17), if  $f$  is 1-periodic,  $f(0) = f(1)$ :  $Q_n^I = T_n$ . Hence we arrive at the following estimate for the quadrature error

$$E_n(f) := \left| \int_0^1 f(t) dt - T_n(f) \right| \leq 2\pi \max_{z \in S^1} \left| ((\Phi_{S^1}^{-1})^* f)(z) - (\mathcal{L}_{\mathcal{Z}}(\Phi_{S^1}^{-1})^* f)(z) \right| .$$

Equivalently, one can show that  $T_n$  integrates trigonometric polynomials up to degree  $2n - 1$  exactly:

$$f(t) = e^{2\pi i k t} \quad \Rightarrow \quad \begin{cases} \int_0^1 f(t) dt = \begin{cases} 0 & , \text{if } k \neq 0 , \\ 1 & , \text{if } k = 0 . \end{cases} \\ T_n(f) = \frac{1}{n} \sum_{l=0}^{n-1} e^{\frac{2\pi i}{n} lk} \stackrel{(9.2.4)}{=} \begin{cases} 0 & , \text{if } k \notin n\mathbb{Z} , \\ 1 & , \text{if } k \in n\mathbb{Z} . \end{cases} \end{cases}$$

The equidistant trapezoidal rule  $T_n$  is exact for trigonometric polynomials of degree  $< 2n$  !

**Remark 5.4.22 (Approximate computation of Fourier coefficients)**

Recall from Section 9.2.5: recovery of signal  $(y_k)_{k \in \mathbb{Z}}$  from its Fourier transform  $c(t)$

$$y_j = \int_0^1 c(t) \exp(2\pi i jt) dt . \quad (9.2.44)$$

Task: approximate computation of  $y_j$

Recall:  $c(t)$  obtained from  $(y_k)_{k \in \mathbb{Z}}$  through Fourier series

$$c(t) = \sum_{k \in \mathbb{Z}} y_k \exp(-2\pi i kt) . \quad (9.2.34)$$

➤  $c(t)$  smooth & 1-periodic for finite/rapidly decaying  $(y_k)_{k \in \mathbb{Z}}$ .

Exp. 5.4.16 ➔ use **equidistant trapezoidal rule** (5.4.17)  
for approximate evaluation of integral in (9.2.44).

☞ Boils down to inverse DFT (9.2.15); hardly surprising in light of the derivation of (9.2.44) in Section 9.2.5.

**MATLAB-code 5.4.23: DFT-based approximate computation of Fourier coefficients**

```

1 function y = fourcoeffcomp(c,m,ovsmpl)
2 % Compute the Fourier coefficients y_{-m},...,y_m of the function
3 % c : [0,1] → C using an oversampling factor ovsmpl.
4 % c must be a handle to a function @t, which accepts row
5 % vector arguments
6 if (nargin < 3), ovsmpl = 2; else ovsmpl = ceil(ovsmpl); end
7 N = (2*m+1)*ovsmpl; h = 1/N; % Number of quadrature points
8 % (Inverse) discrete Fourier transform
9 y = ifft(c(0:h:1-h));
10 % Undo oversampling and wrapping of Fourier coefficient array
11 y = [y(N-m+1:N),y(1:m+1)];

```

## 5.5 Adaptive Quadrature

Hitherto, we have just “blindly” applied quadrature rules for the approximate evaluation of  $\int_a^b f(t) dt$ , oblivious of any properties of the integrand  $f$ . This led us to the conclusion of Rem. 5.4.13 that Gauss-Legendre quadrature (→ Def. 5.3.28) should be preferred to composite quadrature rules (→ Section 5.4) in general. Now the composite quadrature rule will partly be rehabilitated, because they offer the flexibility to *adjust the quadrature rule to the integrand*, a policy known as adaptive quadrature.

## Adaptive numerical quadrature

The policy of **adaptive quadrature** approximates  $\int_a^b f(t) dt$  by a quadrature formula (5.1.2), whose nodes  $c_j^n$  are chosen depending on the integrand  $f$ .

We distinguish

- (I) **a priori** adaptive quadrature: the nodes are *fixed* before the evaluation of the quadrature formula, taking into account external information about  $f$ , and
- (II) **a posteriori** adaptive quadrature: the node positions are chosen or improved based on information gleaned *during the computation* inside a loop. It terminates when sufficient accuracy has been reached.

In this section we will chiefly discuss a posteriori adaptive quadrature for composite quadrature rules ( $\rightarrow$  Section 5.4) based on a single local quadrature rule (and its transformation).



*Supplementary reading.* [19, Sect. 9.7]

### Example 5.5.2 (Rationale for adaptive quadrature)

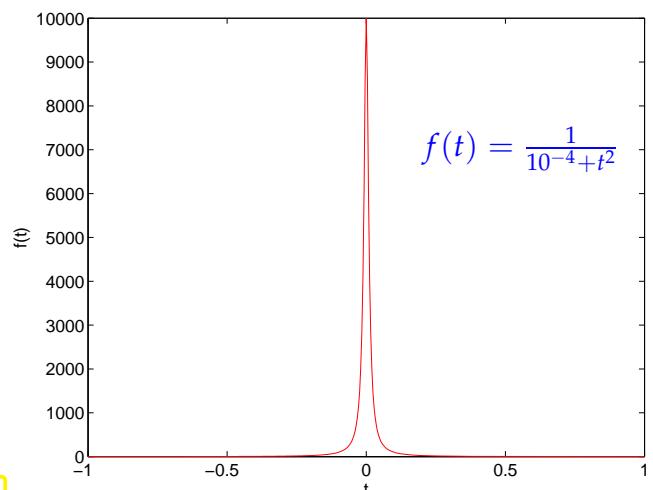
This example presents an extreme case. We consider the composite trapezoidal rule (5.4.4) on a mesh  $\mathcal{M} := \{a = x_0 < x_1 < \dots < x_m = b\}$  and for the integrand  $f(t) = \frac{1}{10^{-4}+t^2}$  on  $[-1, 1]$ .

$f$  is a spike-like function

▷

Intuition: quadrature nodes should cluster around 0, whereas hardly any are needed close to the endpoints of the integration interval, where the function has very small (in modulus) values.

➤ Use locally refined mesh !



A quantitative justification can appeal to (5.2.10) and the resulting bound for the **local** quadrature error (for  $f \in C^2([a, b])$ ):

$$\int_{x_{k-1}}^{x_k} f(t) dt - \frac{1}{2}(f(x_{k-1}) + f(x_k)) \leq h_k^3 \|f''\|_{L^\infty([x_{k-1}, x_k])}, \quad h_k := x_k - x_{k-1}. \quad (5.5.3)$$

➤ Suggests the use of small mesh intervals, where  $|f''|$  is large !

### (5.5.4) Goal: equidistribution of errors

The ultimate but elusive goal is to find a mesh with a minimal number of cells that just delivers a quadrature error below a prescribed threshold. A more practical goal is to adjust the local meshwidths  $h_k := x_k - x_{k-1}$  in order to *achieve a minimal sum of local error bounds*. This leads to the **constrained minimization problem**:

$$\sum_{k=1}^m h_k^3 \|f''\|_{L^\infty([x_{k-1}, x_k])} \rightarrow \min \quad \text{s.t.} \quad \sum_{k=1}^m h_k = b - a . \quad (5.5.5)$$

#### Lemma 5.5.6.

Let  $f : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$  be a **convex** function with  $f(0) = 0$  and  $x > 0$ . Then the constrained minimization problem: seek  $\zeta_1, \dots, \zeta_m \in \mathbb{R}_0^+$  such that

$$\sum_{k=1}^m f(\zeta_k) \rightarrow \min \quad \text{and} \quad \sum_{k=1}^m \zeta_k = x , \quad (5.5.7)$$

has the solution  $\zeta_1 = \zeta_2 = \dots = \zeta_m = \frac{x}{m}$ .

This means that we should strive for equal bounds  $h_k^3 \|f''\|_{L^\infty([x_{k-1}, x_k])}$  for all mesh cells.

#### Error equidistribution principle

The mesh for *a posteriori adaptive composite numerical quadrature* should be chosen to achieve equal contributions of all mesh intervals to the quadrature error

As indicated above, guided by the equidistribution principle, the improvement of the mesh will be done gradually in an iteration. The change of the mesh in each step is called **mesh adaptation** and there are two fundamentally different ways to do it:

- (I) by **moving** nodes, keeping their total number, but making them cluster where mesh intervals should be small, or
- (II) by **adding** nodes, where mesh intervals should be small (**mesh refinement**).

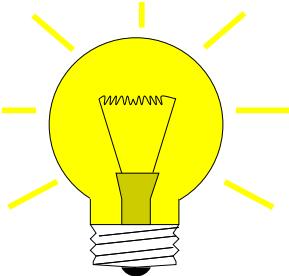
Algorithms for *a posteriori adaptive quadrature* based on mesh refinement usually have the following structure:

#### Adaptation loop for numerical quadrature

- (1) **ESTIMATE**: based on available information compute an approximation for the quadrature error on every mesh interval.
- (2) **CHECK TERMINATION**: if total error sufficient small → **STOP**
- (3) **MARK**: single out mesh intervals with the largest or above average error contributions.
- (4) **REFINE**: add node(s) inside the marked mesh intervals. GOTO (1)

### (5.5.10) Adaptive multilevel quadrature

We now see a concrete algorithm based on the two composite quadrature rules introduced in Ex. 5.4.3.



Idea: local error estimation by comparing local results of two quadrature formulas  $Q_1, Q_2$  of *different* order  $\rightarrow$  local error estimates

heuristics:  $\text{error}(Q_2) \ll \text{error}(Q_1) \Rightarrow \text{error}(Q_1) \approx Q_2(f) - Q_1(f)$ .

Here:  $Q_1$  = trapezoidal rule (order 2)  $\leftrightarrow$   $Q_2$  = Simpson rule (order 4)

Given: initial mesh  $\mathcal{M} := \{a = x_0 < x_1 < \dots < x_m = b\}$

#### ① (Error estimation)

For  $I_k = [x_{k-1}, x_k], k = 1, \dots, m$  (midpoints  $p_k := \frac{1}{2}(x_{k-1} + x_k)$ )

$$\text{EST}_k := \left| \underbrace{\frac{h_k}{6}(f(x_{k-1}) + 4f(p_k) + f(x_k))}_{\text{Simpson rule}} - \underbrace{\frac{h_k}{4}(f(x_{k-1}) + 2f(p_k) + f(x_k))}_{\text{trapezoidal rule on split mesh interval}} \right|. \quad (5.5.11)$$

#### ② (Check termination)

Simpson rule on  $\mathcal{M} \Rightarrow$  intermediate approximation  $I \approx \int_a^b f(t) dt$

$$\text{If } \sum_{k=1}^m \text{EST}_k \leq \text{RTOL} \cdot I \quad (\text{RTOL} := \text{prescribed relative tolerance}) \Rightarrow \text{STOP} \quad (5.5.12)$$

#### ③ (Marking)

$$\text{Marked intervals: } \mathcal{S} := \{k \in \{1, \dots, m\}: \text{EST}_k \geq \eta \cdot \frac{1}{m} \sum_{j=1}^m \text{EST}_j\}, \quad \eta \approx 0.9. \quad (5.5.13)$$

#### ④ (Local mesh refinement)

$$\text{new mesh: } \mathcal{M}^* := \mathcal{M} \cup \{p_k := \frac{1}{2}(x_{k-1} + x_k): k \in \mathcal{S}\}. \quad (5.5.14)$$

Then continue with step ① and mesh  $\mathcal{M} \leftarrow \mathcal{M}^*$ .

The following code give a non-optimal recursive MATLAB implementation:

**MATLAB-code 5.5.15: *h*-adaptive numerical quadrature**

```

1 function I = adaptquad(f,M,rtol,abstol)
2 % adaptive numerical quadrature: f is a function handle to integrand
3 h = diff(M); % distances of quadrature nodes
4 mp = 0.5*(M(1:end-1)+M(2:end)); % positions of midpoints
5 fx = f(M); fm = f(mp); %
6 trp_loc = h.* (fx(1:end-1)+2*fm+fx(2:end))/4; % trapezoidal rule
   (5.4.4)
7 simp_loc = h.* (fx(1:end-1)+4*fm+fx(2:end))/6; % Simpson rule (5.4.5)
8 I = sum(simp_loc); % Simpson approximation of integral
   value
9 est_loc = abs(simp_loc -trp_loc); % local error estimate (5.5.11)
10 err_tot = sum(est_loc); % estimate for quadrature error
11 % Termination based on (5.5.12)
12 if ((err_tot > rtol*abs(I)) || (err_tot > abstol)) %
13 refcells = find(est_loc > 0.9*sum(est_loc)/length(h));
14 I = adaptquad(f, sort([M,mp(refcells)]),rtol,abstol); %
15 end

```

Comments on Code 5.5.15:

- Arguments:  $f \hat{=} \text{handle}$  to function  $f$ ,  $M \hat{=} \text{initial mesh}$ ,  $\text{rtol} \hat{=} \text{relative tolerance for termination}$ ,  $\text{abstol} \hat{=} \text{absolute tolerance for termination}$ , necessary in case the exact integral value = 0, which renders a relative tolerance meaningless.
- Line 3: compute lengths of mesh-intervals  $[x_{j-1}, x_j]$ ,
- Line 4: store positions of midpoints  $p_j$ ,
- Line 5: evaluate function (vector arguments!),
- Line 6: local composite trapezoidal rule (5.4.4),
- Line 7: local simpson rule (5.2.6),
- Line 8: value obtained from composite simpson rule is used as intermediate approximation for integral value,
- Line 9: difference of values obtained from local composite trapezoidal rule ( $\sim Q_1$ ) and local simpson rule ( $\sim Q_2$ ) is used as an estimate for the local quadrature error.
- Line 10: estimate for global error by summing up **moduli** of local error contributions,
- Line 12: terminate, once the estimated total error is below the relative or absolute error threshold,
- Line 14 otherwise, add midpoints of mesh intervals with large error contributions according to (5.5.14) to the mesh and continue.

**Remark 5.5.16 (Estimation of “wrong quadrature error”?)**

In Code 5.5.15 we use the higher order quadrature rule, the Simpson rule of order 4, to compute an approximate value for the integral. This is reasonable, because it would be foolish not to use this information

after we have collected it for the sake of error estimation.

Yet, according to our heuristics, what `est_loc` and `est_tot` give us are estimates for the error of the second-order trapezoidal rule, which we do not use for the actual computations.

However, experience teaches that

`est_loc` gives useful (for the sake of mesh refinement) information about the *distribution* of the error of the Simpson rule, though it fails to capture its *size*.

Therefore, the termination criterion of Line 12 may not be appropriate!

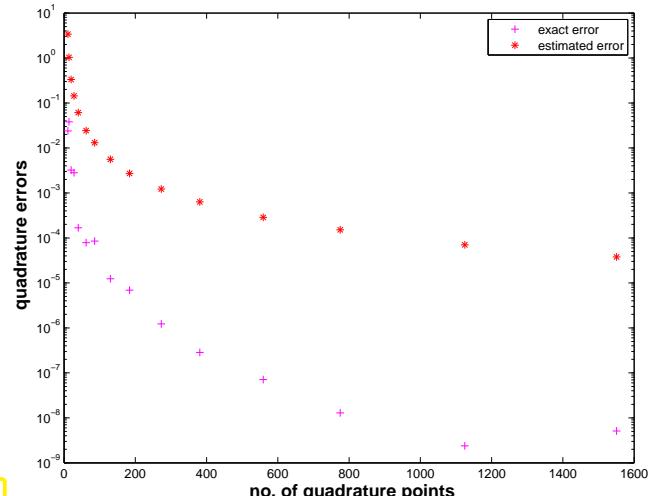
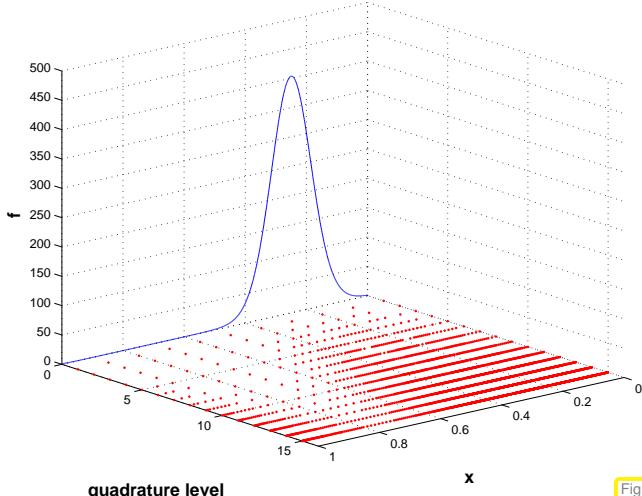
### Experiment 5.5.17 ( $h$ -adaptive numerical quadrature)

In this numerical test we investigate whether the adaptive technique from § 5.5.10 produces an appropriate distribution of integration nodes. We do this for different functions.

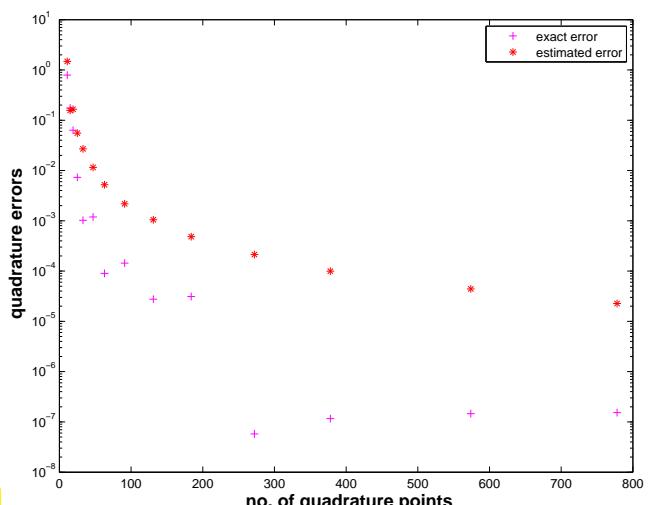
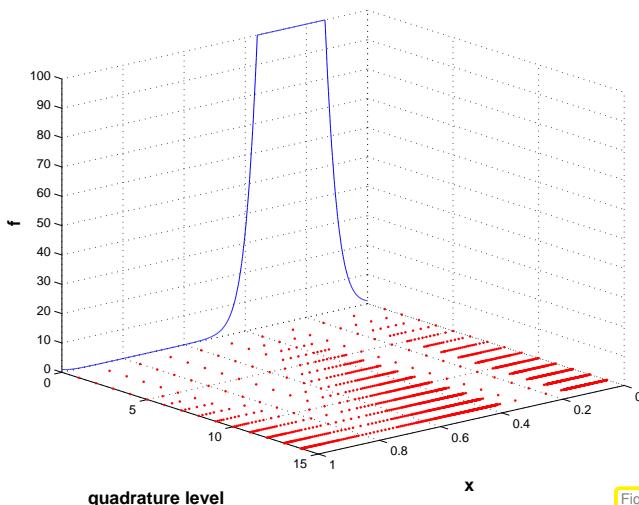
- \* approximate  $\int_0^1 \exp(6 \sin(2\pi t)) dt$ , initial mesh  $\mathcal{M}_0 = \{j/10\}_{j=0}^{10}$

Algorithm: adaptive quadrature, Code 5.5.15 with tolerances  $\text{rtol} = 10^{-6}$ ,  $\text{abstol} = 10^{-10}$

We monitor the distribution of quadrature points during the adaptive quadrature and the true and estimated quadrature errors. The “exact” value for the integral is computed by composite Simpson rule on an equidistant mesh with  $10^7$  intervals.



- \* approximate  $\int_0^1 \min\{\exp(6 \sin(2\pi t)), 100\} dt$ , initial mesh as above



Observation:

- Adaptive quadrature locally decreases meshwidth where integrand features variations or kinks.
- Trend for estimated error mirrors behavior of true error.
- Overestimation may be due to taking the modulus in (5.5.11)

However, the important piece of information we want to extract from  $\text{EST}_k$  is about the *distribution* of the quadrature error.

#### Remark 5.5.18 (Adaptive quadrature in MATLAB)

```
q = quad(fun,a,b,tol): adaptive multigrid quadrature  
                           (local low order quadrature formulas)  
q = quadl(fun,a,b,tol): adaptive Gauss-Lobatto quadrature
```

## Learning Outcomes

- ✿ You should know what is a quadrature formula and terminology connected with it,
- ✿ You should be able to transform quadrature formulas to arbitrary intervals.
- ✿ You should understand how interpolation and approximation schemes spawn quadrature formulas and how quadrature errors are connected to interpolation/approximation errors.
- ✿ You should be able to compute the weights of polynomial quadrature formulas.
- ✿ You should know the concept of order of a quadrature rule and why it is invariant under (affine) transformation
- ✿ You should remember the maximal and minimal order of polynomial quadrature rules.
- ✿ You should know the order of the  $n$ -point Gauss-Legendre quadrature rule.
- ✿ You should understand why Gauss-Legendre quadrature converges exponentially for integrands that can be extended analytically and algebraically for integrands with limited smoothness.
- ✿ You should be able to apply regularizing transformations to integrals with non-smooth integrands.

- ✿ You should know about asymptotic convergence of the  $h$ -version of composite quadrature.
- ✿ You should know the principles of adaptive composite quadrature.

# Chapter 6

## Least Squares

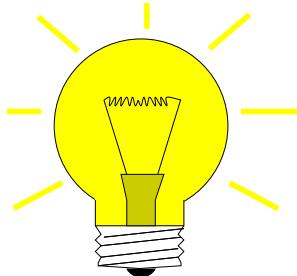
### Example 6.0.1 (Least squares data fitting)

In Section 3.1 we discussed the reconstruction of a *parameterized function*  $f(x_1, \dots, x_n; \cdot) : D \subset \mathbb{R} \mapsto \mathbb{R}$  from data points  $(t_i, y_i)$ ,  $i = 1, \dots, n$ , by imposing interpolation conditions (3.1.1). Necessary was that the number  $n$  of parameters agreed with number of data points. The interpolation approach is justified in the case of *highly accurate data*.

Frequently encountered:

*inaccurate data* (due to *measurement errors*)

- interpolation approach dubious (impact of “outliers”!)



Mitigate impact of data uncertainty by  
choosing fewer parameters than data points  
(measurement errors can “average out”)

Non-linear least squares fitting problem: [15, Sect. 6.1]

Given:     \* data points  $(t_i, y_i)$ ,  $i = 1, \dots, m$   
             \* (symbolic formula) for parameterized function  
$$f(x_1, \dots, x_n; \cdot) : D \subset \mathbb{R} \mapsto \mathbb{R}, \quad n < m$$

Sought: parameter values  $x_1^*, \dots, x_n^* \in \mathbb{R}$  such that

$$(x_1^*, \dots, x_n^*) = \underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} \sum_{i=1}^m |f(x_1, \dots, x_n; t_i) - y_i|^2. \quad (6.0.2)$$

### Example 6.0.3 (Linear data fitting → [15, Sect. 4.1])

[Linear data fitting → [15, Ex. 4.2 & Ex. 4.3]]

Special case, cf. in Section 3.1, (3.1.7): Representation of  $f$  by finite linear combination of basis functions  
 $b_j : D \subset \mathbb{R} \mapsto \mathbb{R}, j = 1, \dots, n$ :

$$f(t) = \sum_{j=1}^n x_j b_j(t) , \quad x_j \in \mathbb{R}^d . \quad (6.0.4)$$

→  $f \in$  finite dimensional function space  $V_n := \text{Span}\{b_1, \dots, b_n\}$ .

### Linear least squares fitting problem:

Given:

- \* data points  $(t_i, y_i), i = 1, \dots, m$
- \* basis functions  $b_j : I \mapsto \mathbb{K}, j = 1, \dots, n, n < m$

Sought: coefficients  $x_j^* \in \mathbb{R}, j = 1, \dots, n$ , such that

$$(x_1^*, \dots, x_n^*) = \underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} \sum_{i=1}^m \left| \sum_{j=1}^n x_j b_j(t_i) - y_i \right|^2 . \quad (6.0.5)$$

### Example 6.0.6 (Polynomial fitting)

Special variant of linear data fitting (→ Ex. 6.0.3): choose  $f$  as polynomial of degree  $n - 1$ ,

$$V_n = \mathcal{P}_{n-1} , \quad \text{e.g. } b_j(t) = t^{j-1} \quad (\text{monomial basis, Section 3.2.1}) .$$

→ MATLAB-function for solving (6.0.2) in this case:

`p = polyfit(t, y, d);`

$d \hat{=} \text{polynomial degree}$ ,  $t \hat{=} \text{vector } (t_i)_{i=1}^n$ ,  $y \hat{=} \text{vector } (y_i)_{i=1}^n$

Note:  $p \hat{=} \text{vector of monomial coefficients in MATLAB convention, see Rem. 3.2.4.}$

### Example 6.0.7 (Polybomial interpolation vs. polynomial fitting)

#### MATLAB-code 6.0.8: Fitting and interpolating polynomial

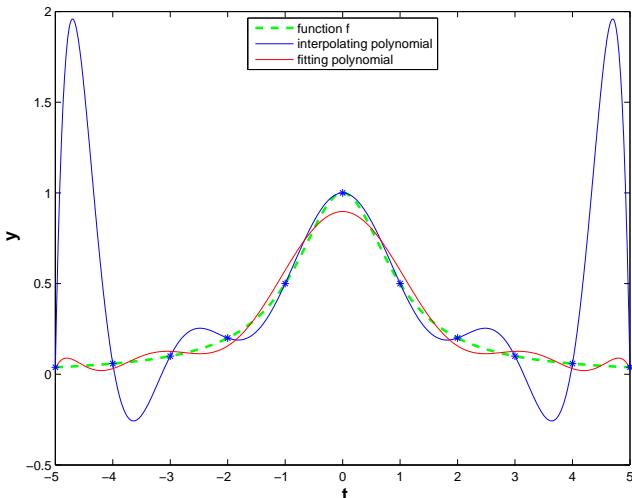
```
% Comparison of polynomial interpolation and polynomial fitting
% ("Quick and dirty" MATLAB implementation, see 3.2.3)
f = @(x) 1./(1+x.^2); % Function providing data points
d = 10; % Polynomial degree
tip = -5+(0:d)*10/d; % d+1 nodes for interpolation
tft = -5+(0:3*d)*10/(3*d); % 3d+1 Nodes for polynomial fitting
pip = polyfit(tip, f(tip), d); % Interpolating polynomial (degree d)
pft = polyfit(tft, f(tft), d); % Fitting polynomial (degree d)
x = -5+(0:1000)/100;
h = plot(x, f(x), 'g--', ...
```

```

x, polyval(pip,x), 'b-', ...
x, polyval(pft,x), 'r-', ...
tip, f(tip), 'b*');

set(h(1),'linewidth',2);
xlabel('{\bf t}', 'fontsize',14);
ylabel('{\bf y}', 'fontsize',14);
legend('function f','interpolating polynomial','fitting
polynomial','location','north');
print -depsc2 '../PICTURES/interpfit.eps';

```



Data from function  $f(t) = \frac{1}{1+t^2}$ ,

- \* polynomial degree  $d = 10$ ,
- \* interpolation through data points  $(t_j, f(t_j))$ ,  $j = 0, \dots, d$ ,  $t_j = -5 + j$ , see Ex. 3.2.59,
- \* fitting to data points  $(t_j, f(t_j))$ ,  $j = 0, \dots, 3d$ ,  $t_j = -5 + j/3$ .

Fitting helps curb oscillations that haunt polynomial interpolation!

### Example 6.0.9 (linear regression) → [15, Ex. 4.1])

[linear regression]

☞ example for multidimensional linear least squares data fitting:

Given: measured data points  $(\mathbf{x}_i, y_i)$ ,  $\mathbf{x}_i \in \mathbb{R}^n$ ,  $y_i \in \mathbb{R}$ ,  $i = 1, \dots, m$ ,  $m \geq n+1$   
 $(y_i, \mathbf{x}_i)$  affected by measurement errors).

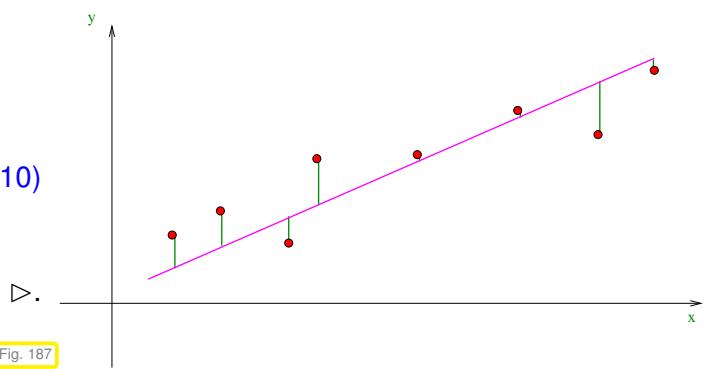
Known: without measurement errors data would satisfy  
affine linear relationship  $y = \mathbf{a}^\top \mathbf{x} + c$ ,  $\mathbf{a} \in \mathbb{R}^n$ ,  $c \in \mathbb{R}$ .

Goal: estimate parameters  $\mathbf{a}$ ,  $c$ .

least squares estimate

$$(\mathbf{a}, c) = \underset{\mathbf{p} \in \mathbb{R}^n, q \in \mathbb{R}}{\operatorname{argmin}} \sum_{i=1}^m |y_i - \mathbf{p}^\top \mathbf{x}_i - q|^2 \quad (6.0.10)$$

linear regression for  $n = 2, m = 8$



Remark, see [15, Sect. 4.5]: In statistics we learn that the least squares estimate provides a maximum likelihood estimate if the measurement errors are uniformly and independently normally distributed.

**Remark 6.0.11 (Overdetermined linear systems)**

In Ex. 6.0.9 we could try to find  $\mathbf{a}, \mathbf{c}$  by solving the linear system of equations

$$\begin{bmatrix} \mathbf{x}_1^\top & 1 \\ \vdots & \vdots \\ \mathbf{x}_m^\top & 1 \end{bmatrix} \begin{bmatrix} \mathbf{a} \\ \mathbf{c} \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}, \quad (6.0.12)$$

but in case  $m > n + 1$  we encounter more equations than unknowns.

In Ex. 6.0.3 the same idea leads to the linear system

$$\begin{bmatrix} b_1(t_1) & \dots & b_n(t_1) \\ \vdots & & \vdots \\ b_1(t_m) & \dots & b_n(t_m) \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}, \quad (6.0.13)$$

with the same problem in case  $m > n$ .

Now we elaborate the common mathematical structure underlying the linear least squares fitting problem from Ex. 6.0.3 and Ex. 6.0.9.

(Full rank linear) least squares problem: [15, Sect. 4.2]

given:  $\mathbf{A} \in \mathbb{R}^{m,n}$ ,  $m, n \in \mathbb{N}$ ,  $m \geq n$ ,  $\text{rank}(\mathbf{A}) = n$ ,  $\mathbf{b} \in \mathbb{R}^m$ ,

find:  $\mathbf{x} \in \mathbb{R}^n$  such that  $\|\mathbf{Ax} - \mathbf{b}\|_2 = \inf_{\mathbf{y} \in \mathbb{R}^n} \|\mathbf{Ay} - \mathbf{b}\|_2$   
 $\Updownarrow$   
 $\mathbf{x} = \underset{\mathbf{y} \in \mathbb{R}^n}{\operatorname{argmin}} \|\mathbf{Ay} - \mathbf{b}\|_2$

Sloppy notation for the minimization problem (6.0.14):  $\|\mathbf{Ax} - \mathbf{b}\|_2 \rightarrow \min$

Recast as linear least squares problem, cf. Rem. 6.0.11:

$$\text{Ex. 6.0.9: } \mathbf{A} = \begin{bmatrix} \mathbf{x}_1^\top & 1 \\ \vdots & \vdots \\ \mathbf{x}_m^\top & 1 \end{bmatrix} \in \mathbb{R}^{m,n+1}, \quad \mathbf{b} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \in \mathbb{R}^n, \quad \mathbf{x} = \begin{bmatrix} \mathbf{a} \\ \mathbf{c} \end{bmatrix} \in \mathbb{R}^{n+1}.$$

$$\text{Ex. 6.0.3: } \mathbf{A} = \begin{bmatrix} b_1(t_1) & \dots & b_n(t_1) \\ \vdots & & \vdots \\ b_1(t_m) & \dots & b_n(t_m) \end{bmatrix} \in \mathbb{R}^{m,n}, \quad \mathbf{b} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \in \mathbb{R}^m, \quad \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n.$$

In both cases the residual norm  $\|\mathbf{b} - \mathbf{Ax}\|_2$  allows to gauge the quality of the model.

What if  $\text{rank } \mathbf{A} < n$  in (6.0.14) ?

$$\mathbf{A} \in \mathbb{R}^{m,n}, \quad m \geq n, \quad \text{rank } \mathbf{A} < n \quad \stackrel{[59, \text{Sect. 6.1}]}{\Rightarrow} \quad \exists \mathbf{z} \neq 0: \quad \mathbf{Az} = 0. \quad (6.0.15)$$

- Solution of (6.0.14) cannot be unique!
  - We need another condition to single out a unique minimizer: **minimum norm condition**

(General linear) least squares problem:

given:  $\mathbf{A} \in \mathbb{R}^{m,n}$ ,  $m, n \in \mathbb{N}$ ,  $\mathbf{b} \in \mathbb{R}^m$ ,

find:  $x \in \mathbb{R}^n$  such that

(i)  $\|\mathbf{Ax} - \mathbf{b}\|_2 = \inf\{\|\mathbf{Ay} - \mathbf{b}\|_2 : \mathbf{y} \in \mathbb{R}^n\},$  (6.0.16)  
(ii)  $\|\mathbf{x}\|_2$  is minimal under the condition (i).

**Lemma 6.0.17. Existence & uniqueness of solutions of the least squares problem**

The least squares problem for  $\mathbf{A} \in \mathbb{K}^{m,n}$ ,  $\mathbf{A} \neq 0$ , has a unique solution for every  $\mathbf{b} \in \mathbb{K}^m$ .

*Proof.* The proof is given by formula (6.2.6) and its derivation, see Section 6.2.

MATLAB “black-box” solver for general linear least squares problems (6.0.16)

$\mathbf{x} = \mathbf{A} \backslash \mathbf{b}$  ("backslash") solves (6.0.16) for  $\mathbf{A} \in \mathbb{K}^{m,n}$ ,  $m \neq n$ .

Reassuring: stable ( $\rightarrow$  Def. 1.5.80) implementation (for dense matrices).

### **Remark 6.0.18 (Rank defect in linear least squares problems)**

Consider linear least squares fitting problem (6.0.5) with  $\mathbf{A} \in \mathbb{R}^{m,n}$  from (6.0.13).

$\text{rank}(\mathbf{A}) < n \Leftrightarrow$  columns of  $\mathbf{A}$  are linearly dependent

- ⇒ at least one of the basis functions  $b_j$  is redundant, because it can not be distinguished from a linear combination of some others on  $\{t_i\}$
- > too many basis functions   ⇒   too few data points

Usually  $\text{rank}(\mathbf{A}) < n$  in linear fitting hints at inadequate modelling

**Remark 6.0.19 (Pseudoinverse → [42, Ch. 12])**

By Lemma 6.0.17 the solution operator of the least squares problem (6.0.16) defines a linear mapping  $\mathbf{b} \mapsto \mathbf{x}$ , which has a matrix representation.

### Definition 6.0.20. Pseudoinverse

The **pseudoinverse**  $\mathbf{A}^+ \in \mathbb{K}^{n,m}$  of  $\mathbf{A} \in \mathbb{K}^{m,n}$  is the matrix representation of the (linear) solution operator  $\mathbb{R}^m \mapsto \mathbb{R}^n$ ,  $\mathbf{b} \mapsto \mathbf{x}$  of the least squares problem (6.0.16)  $\|\mathbf{Ax} - \mathbf{b}\| \rightarrow \min$ ,  $\|\mathbf{x}\| \rightarrow \min$ .

MATLAB

`P = pinv(A)` computes the pseudoinverse.

**Remark 6.0.21 (Conditioning of the least squares problem** → [15, Sect. 4.3])

**Definition 6.0.22. Generalized condition (number) of a matrix,** → Def. 1.6.15

Let  $\sigma_1 \geq \sigma_2 \geq \sigma_r > \sigma_{r+1} = \dots = \sigma_p = 0$ ,  $p := \min\{m, n\}$ , be the singular values (→ Def. 7.5.8) of  $\mathbf{A} \in \mathbb{K}^{m,n}$ . Then

$$\text{cond}_2(\mathbf{A}) := \frac{\sigma_1}{\sigma_r}$$

is the **generalized condition (number)** (w.r.t. the 2-norm) of  $\mathbf{A}$ .

**Theorem 6.0.23.**

For  $m \geq n$ ,  $\mathbf{A} \in \mathbb{K}^{m,n}$ ,  $\text{rank}(\mathbf{A}) = n$ , let  $\mathbf{x} \in \mathbb{K}^n$  be the solution of the least squares problem  $\|\mathbf{Ax} - \mathbf{b}\| \rightarrow \min$  and  $\hat{\mathbf{x}}$  the solution of the perturbed least squares problem  $\|(\mathbf{A} + \Delta\mathbf{A})\hat{\mathbf{x}} - \mathbf{b}\| \rightarrow \min$ . Then

$$\frac{\|\mathbf{x} - \hat{\mathbf{x}}\|_2}{\|\mathbf{x}\|_2} \leq \left( 2 \text{cond}_2(\mathbf{A}) + \text{cond}_2^2(\mathbf{A}) \frac{\|\mathbf{r}\|_2}{\|\mathbf{A}\|_2 \|\mathbf{x}\|_2} \right) \frac{\|\Delta\mathbf{A}\|_2}{\|\mathbf{A}\|_2}$$

holds, where  $\mathbf{r} = \mathbf{Ax} - \mathbf{b}$  is the **residual**.

This means: if  $\|\mathbf{r}\|_2 \ll 1$  ► condition of the least squares problem  $\approx \text{cond}_2(\mathbf{A})$   
if  $\|\mathbf{r}\|_2$  “large” ► condition of the least squares problem  $\approx \text{cond}_2^2(\mathbf{A})$

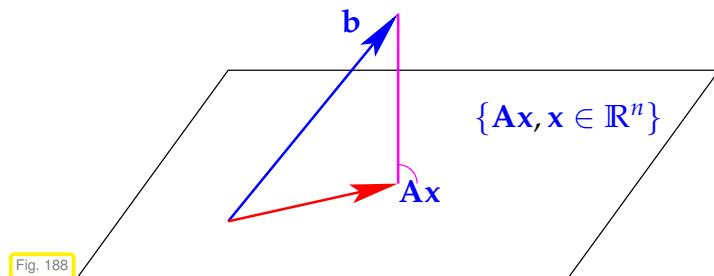
## Contents

6.1	Normal Equations [15, Sect. 4.2], [42, Ch. 11]	400
6.2	Orthogonal Transformation Methods [15, Sect. 4.4.2]	403
6.3	Total Least Squares	410
6.4	Constrained Least Squares	411
6.4.1	Solution via normal equations	412
6.4.2	Solution via SVD	413
6.5	Non-linear Least Squares [15, Ch. 6]	414
6.5.1	(Damped) Newton method	414
6.5.2	Gauss-Newton method	415
6.5.3	Trust region method (Levenberg-Marquardt method)	418

## 6.1 Normal Equations [15, Sect. 4.2], [42, Ch. 11]

Setting (6.0.14):  $\mathbf{A} \in \mathbb{R}^{m,n}$ ,  $m \geq n$ , with full rank  $\text{rank}(\mathbf{A}) = n$ :

$$\|\mathbf{Ax} - \mathbf{b}\|_2 \rightarrow \min .$$



Geometric interpretation of linear least squares problem (6.0.16):

$\mathbf{x} \hat{=} \text{orthogonal projection of } \mathbf{b} \text{ on the subspace } \text{Im}(\mathbf{A}) := \text{Span}\{(\mathbf{A})_{:,1}, \dots, (\mathbf{A})_{:,n}\}.$

Geometric interpretation: the least squares problem (6.0.16) amounts to searching the point  $\mathbf{p} \in \text{Im}(\mathbf{A})$  nearest (w.r.t. Euclidean distance) to  $\mathbf{b} \in \mathbb{R}^m$ .

Geometric intuition, see Fig. 188:  $\mathbf{p}$  is the orthogonal projection of  $\mathbf{b}$  onto  $\text{Im}(\mathbf{A})$ , that is  $\mathbf{b} - \mathbf{p} \perp \text{Im}(\mathbf{A})$ . Note the equivalence

$$\mathbf{b} - \mathbf{p} \perp \text{Im}(\mathbf{A}) \Leftrightarrow \mathbf{b} - \mathbf{p} \perp (\mathbf{A})_{:,j}, \quad j = 1, \dots, n \Leftrightarrow \mathbf{A}^H(\mathbf{b} - \mathbf{p}) = 0,$$

Representation  $\mathbf{p} = \mathbf{Ax}$  leads to normal equations (6.1.2). General discussion in [15, Sect. 4.6].

Solve (6.0.16) for  $\mathbf{b} \in \mathbb{R}^m$

$$\mathbf{x} \in \mathbb{R}^n: \quad \|\mathbf{Ax} - \mathbf{b}\|_2 \rightarrow \min \Leftrightarrow f(\mathbf{x}) := \|\mathbf{Ax} - \mathbf{b}\|_2^2 \rightarrow \min. \quad (6.1.1)$$

A quadratic functional, cf. Section 8.1.1, (8.1.4)

$$f(\mathbf{x}) = \|\mathbf{Ax} - \mathbf{b}\|_2^2 = \mathbf{x}^H(\mathbf{A}^H\mathbf{A})\mathbf{x} - 2\mathbf{b}^H\mathbf{Ax} + \mathbf{b}^H\mathbf{b}.$$

Minimization problem for  $f$  ➤ study gradient, cf. (8.1.9)

$$\mathbf{grad} f(\mathbf{x}) = 2(\mathbf{A}^H\mathbf{A})\mathbf{x} - 2\mathbf{A}^H\mathbf{b}.$$



$$\mathbf{grad} f(\mathbf{x}) \stackrel{!}{=} 0: \quad \boxed{\mathbf{A}^H\mathbf{A}\mathbf{x} = \mathbf{A}^H\mathbf{b}} \quad = \text{normal equation of (6.1.1)} \quad (6.1.2)$$

Notice:  $\text{rank}(\mathbf{A}) = n \Rightarrow \mathbf{A}^H\mathbf{A} \in \mathbb{R}^{n,n}$  s.p.d. (→ Def. 1.1.8, [15, Rem. 4.6])

### Remark 6.1.3 (Conditioning of normal equations [15, pp. 128])



Caution: danger of instability, with SVD  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^H$

$$\text{cond}_2(\mathbf{A}^H\mathbf{A}) = \text{cond}_2(\mathbf{V}\Sigma^H\mathbf{U}^H\mathbf{U}\Sigma\mathbf{V}^H) = \text{cond}_2(\Sigma^H\Sigma) = \frac{\sigma_1^2}{\sigma_n^2} = \text{cond}_2(\mathbf{A})^2.$$

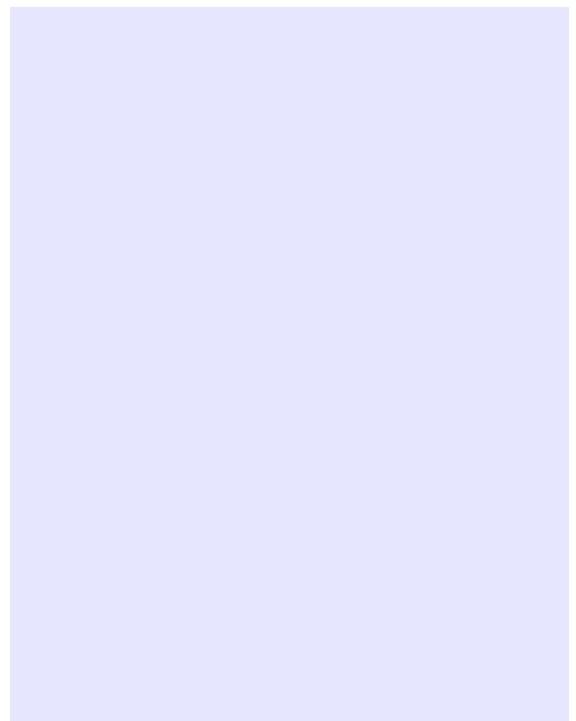
- For fairly ill-conditioned  $\mathbf{A}$  using the normal equations (6.1.2) to solve the linear least squares problem (6.1.1) numerically may run the risk of huge amplification of roundoff errors incurred during the computation of the right hand side  $\mathbf{A}^H\mathbf{b}$ : potential instability (→ Def. 1.5.80) of normal equation approach.

### Example 6.1.4 (Instability of normal equations → [15, Ex. 4.12])

Caution: loss of information in the computation of  $\mathbf{A}^H \mathbf{A}$ , e.g.

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ \delta & 0 \\ 0 & \delta \end{bmatrix}$$

$$\Rightarrow \mathbf{A}^H \mathbf{A} = \begin{bmatrix} 1 + \delta^2 & 1 \\ 1 & 1 + \delta^2 \end{bmatrix}$$



```
>> rank(A'*A)
```

If  $\delta < \sqrt{\text{eps}}$   $\Rightarrow 1 + \delta^2 = 1$  in  $\mathbb{M}$ , i.e.  $\mathbf{A}^H \mathbf{A}$  “numeric singular”, though  $\text{rank}(\mathbf{A}) = 2$ , see ??, in particular Exp. 1.5.34.

---

```
ans = 1
```

#### Remark 6.1.6 (Loss of sparsity when forming normal equations)

Another reason not to compute  $\mathbf{A}^H \mathbf{A}$ , when both  $m, n$  large:

$\mathbf{A}$  sparse  $\not\Rightarrow \mathbf{A}^T \mathbf{A}$  sparse



- \* Potential memory overflow, when computing  $\mathbf{A}^T \mathbf{A}$
- \* Squanders possibility to use efficient sparse direct elimination techniques, see Section 1.7.5

#### Remark 6.1.7 (Extended normal equations)

A way to avoid the computation of  $\mathbf{A}^H \mathbf{A}$ :

Extend normal equations (6.1.2): introduce residual  $\mathbf{r} := \mathbf{Ax} - \mathbf{b}$  as new unknown:

$$\mathbf{A}^H \mathbf{A} \mathbf{x} = \mathbf{A}^H \mathbf{b} \Leftrightarrow \mathbf{B} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} := \begin{bmatrix} -\mathbf{I} & \mathbf{A} \\ \mathbf{A}^H & 0 \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}. \quad (6.1.8)$$

More general substitution  $\mathbf{r} := \alpha^{-1}(\mathbf{A}\mathbf{x} - \mathbf{b})$ ,  $\alpha > 0$  to improve the conditioning:

$$\mathbf{A}^H \mathbf{A} \mathbf{x} = \mathbf{A}^H \mathbf{b} \Leftrightarrow \mathbf{B}_\alpha \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} := \begin{bmatrix} -\alpha \mathbf{I} & \mathbf{A} \\ \mathbf{A}^H & 0 \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}. \quad (6.1.9)$$

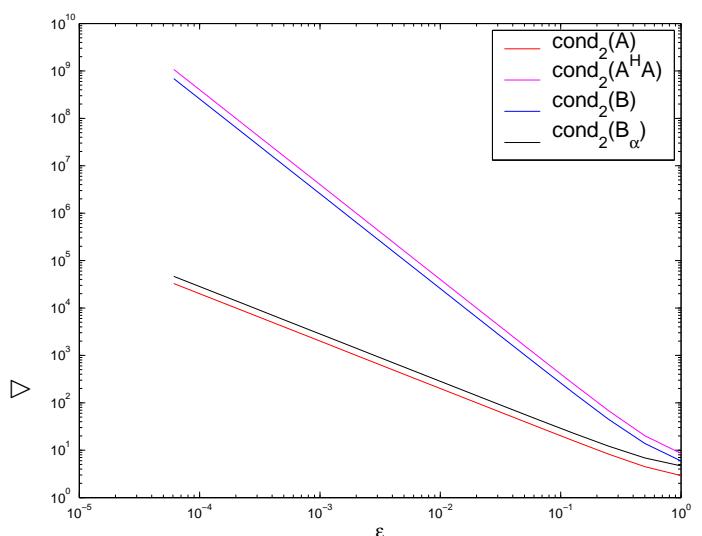
For  $m, n \gg 1$ ,  $\mathbf{A}$  sparse, both (6.1.8) and (6.1.9) lead to large sparse linear systems of equations, amenable to sparse direct elimination techniques, see Section 1.7.5.

### Example 6.1.10 (Conditioning of the extended normal equations)

Consider (6.1.8), (6.1.9) for

$$\mathbf{A} = \begin{bmatrix} 1+\epsilon & 1 \\ 1-\epsilon & 1 \\ \epsilon & \epsilon \end{bmatrix}.$$

Plot of different condition numbers  
in dependence on  $\epsilon$   
( $\alpha = \epsilon \|\mathbf{A}\|_2 / \sqrt{2}$ )



## 6.2 Orthogonal Transformation Methods [15, Sect. 4.4.2]

We consider the linear least squares problem (6.0.14)

$$\text{given } \mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{b} \in \mathbb{R}^m \text{ find } \mathbf{x} = \underset{\mathbf{y} \in \mathbb{R}^n}{\operatorname{argmin}} \|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2. \quad (6.0.14)$$

Assumption:  $m \geq n$  and  $\mathbf{A}$  has full (maximum) rank:  $\operatorname{rank}(\mathbf{A}) = n$ .

Recall: rationale behind Gaussian elimination ( $\rightarrow$  Section 1.6.2, Ex. 1.6.20)

- Row transformations of LSE  $\mathbf{A}\mathbf{x} = \mathbf{b}$  to *equivalent* (in terms of solution) LSE  $\mathbf{U}\mathbf{x} = \tilde{\mathbf{b}}$ , which is easier to solve because it has triangular form.

How to adapt this policy to linear least squares problem (6.0.14) ?

- Two questions:
- ① Which linear least squares problems are “easy to solve” ?
  - ② How do we get them by *equivalent transformations* of (6.0.14) ?

Answer to question ① (same as for LSE):

Linear least squares problems (6.0.14) with upper *triangular*  $\mathbf{A}$  are easy to solve!

$$\begin{array}{c}
 \left[ \begin{array}{cc} R & x \\ 0 & b \end{array} \right] - \left[ \begin{array}{c} x_1 \\ \vdots \\ x_n \end{array} \right] = \left[ \begin{array}{c} b_1 \\ \vdots \\ b_m \end{array} \right] \\
 \rightarrow \min \quad \xrightarrow{(*)} \quad x = \left[ \begin{array}{c} R \\ b \end{array} \right]^{-1} \left[ \begin{array}{c} b_1 \\ \vdots \\ b_m \end{array} \right]
 \end{array}$$

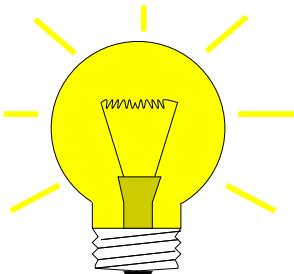
How can we draw the conclusion  $(*)$ ? Obviously, the components  $n+1, \dots, m$  of the vector inside the norm are fixed and do not depend on  $\mathbf{x}$ . All we can do is to make the first components  $1, \dots, n$  vanish, by choosing a suitable  $\mathbf{x}$ , see [15, Thm. 4.13].

Note: since  $\mathbf{A}$  has full rank  $n$  the triangular part  $\mathbf{R} \in \mathbb{R}^{n,n}$  of  $\mathbf{A}$  is regular!

### Answer to question ②:

Recall ??: orthogonal (unitary) transformations ( $\rightarrow$  Def. 4.2.2) leave 2-norm invariant.

Idea: Transformation of  $\mathbf{Ax} - \mathbf{b}$  to simpler form by *orthogonal* row transformations:



$$\underset{\mathbf{y} \in \mathbb{R}^n}{\operatorname{argmin}} \|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2 = \underset{\mathbf{y} \in \mathbb{R}^n}{\operatorname{argmin}} \left\| \tilde{\mathbf{A}}\mathbf{y} - \tilde{\mathbf{b}} \right\|_2,$$

As in the case of LSE ( $\rightarrow$  ??): “simpler form” = triangular form, see ① above.

Is orthogonal transformation to upper triangular form always possible ?

Recall: Gram-Schmidt orthonormalization ( $\rightarrow$  Section 7.3.4.1)

A close scrutiny of (7.3.91), (7.3.92) and Code 7.3.93 (with roles of *m* and *n* swapped) reveals

$$\begin{aligned} \mathbf{q}_1 &= t_{11}\mathbf{a}_1 \\ \mathbf{q}_2 &= t_{12}\mathbf{a}_1 + t_{22}\mathbf{a}_2 \\ \mathbf{q}_3 &= t_{13}\mathbf{a}_1 + t_{23}\mathbf{a}_2 + t_{33}\mathbf{a}_3 \\ &\vdots \\ \mathbf{q}_n &= t_{1n}\mathbf{a}_1 + t_{2n}\mathbf{a}_2 + \cdots + t_{nn}\mathbf{a}_n \end{aligned}$$

$$\exists \mathbf{T} \in \mathbb{R}^{n,n} \text{ upper triangular: } \mathbf{O} = \mathbf{AT}$$

where  $\mathbf{Q} = (\mathbf{q}_1, \dots, \mathbf{q}_n) \in \mathbb{R}^{m,n}$  (with orthonormal columns),  $\mathbf{A} = (\mathbf{a}_1, \dots, \mathbf{a}_n) \in \mathbb{R}^{m,n}$ . By Lemma 1.3.9 we have found an *upper triangular*  $\mathbf{R} := \mathbf{T}^{-1} \in \mathbb{R}^{n,n}$  such that

$$\mathbf{A} = \mathbf{QR} \Leftrightarrow \begin{bmatrix} \mathbf{A} \end{bmatrix} = \begin{bmatrix} \mathbf{Q} \\ \mathbf{R} \end{bmatrix}.$$

Now “augmentation by zero”: add  $m - n$  zero rows at the bottom of  $\mathbf{R}$

complement columns of  $\mathbf{Q}$  to an orthonormal basis of  $\mathbb{R}^m$   
 $\Rightarrow$  orthogonal matrix  $\tilde{\mathbf{Q}} \in \mathbb{R}^{m,m}$

►  $\mathbf{A} = \tilde{\mathbf{Q}} \begin{bmatrix} \mathbf{R} \\ 0 \end{bmatrix} \Leftrightarrow \begin{bmatrix} \mathbf{A} \end{bmatrix} = \begin{bmatrix} \mathbf{Q} \\ \mathbf{R} \\ 0 \end{bmatrix}.$

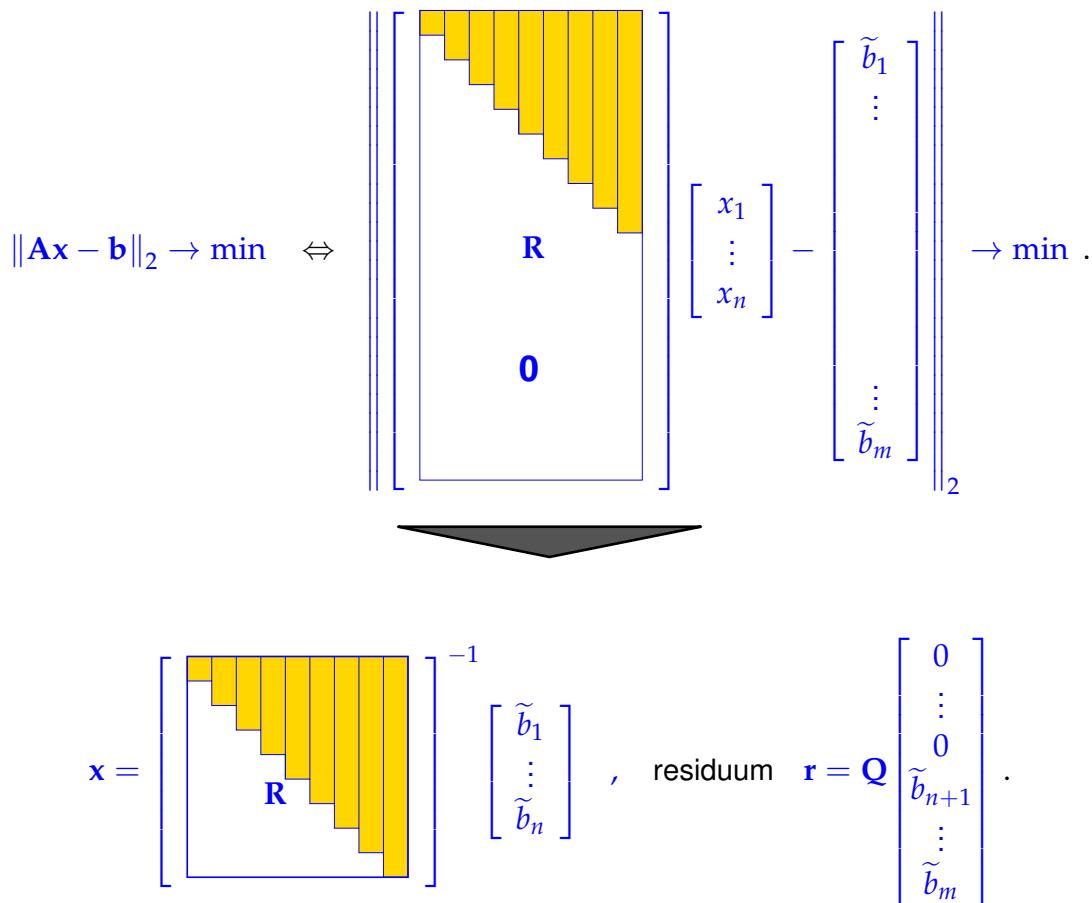
$\Leftrightarrow \tilde{\mathbf{Q}}^\top \mathbf{A} = \begin{bmatrix} \mathbf{R} \\ 0 \end{bmatrix}.$

Finally, let the columns of  $\mathbf{A}$  play the roles of the  $\mathbf{a}_i$  in Gram-Schmidt orthonormalization.

Concrete realization of the “equivalent orthonormal transformation to upper triangular form”-idea by means of **QR-decomposition** ([15, Sect. 4.4.2], recall ??).

**QR-decomposition:**  $\mathbf{A} = \mathbf{QR}$ ,  $\mathbf{Q} \in \mathbb{K}^{m,m}$  unitary,  $\mathbf{R} \in \mathbb{K}^{m,n}$  (regular) upper triangular matrix.

$$\|\mathbf{Ax} - \mathbf{b}\|_2 = \left\| \mathbf{Q}(\mathbf{Rx} - \mathbf{Q}^H \mathbf{b}) \right\|_2 = \left\| \mathbf{Rx} - \tilde{\mathbf{b}} \right\|_2, \quad \tilde{\mathbf{b}} := \mathbf{Q}^H \mathbf{b}.$$



Note: by ?? residual norm readily available  $\|r\|_2 = \sqrt{\tilde{b}_{n+1}^2 + \dots + \tilde{b}_m^2}$ .

#### MATLAB-code 6.2.1: QR-based solver for full rank linear least squares problem (6.0.14)

```

function [x,res] = qrlsqsolve(A,b)
% Solution of linear least squares problem (6.0.14) by means of
% QR-decomposition
% Note: A ∈ ℝm,n with m > n, rank(A) = n is assumed
[m,n] = size(A);
R = triu(qr([A,b],0)), % economical QR-decomposition of extended
% matrix
x = R(1:n,1:n)\R(1:n,n+1); % x̂ = (R)-11:n,1:n(QTb)1:n
res = R(n+1,n+1); % = ||A x̂ - b||2 (why ?)

```

Discussion of (some) details of implementation in Code 6.2.1:

- Gram-Schmidt orthogonalization is done in a numerically stable way by the MATLAB command **qr**; the call **triu**(**qr**(A)),  $A \in \mathbb{K}^{m,n}$  with full rank,  $m \geq n$ , returns the (generalized) upper triangular matrix  $Q^H A \in \mathbb{K}^{m,n}$ , where  $Q$  is an orthogonal matrix, whose first  $n$  columns agree with the vectors produced by Gram-Schmidt orthonormalization of the columns of  $A$ .
- Asymptotic computational cost of **qr**(A):  $O(mn^2)$
- Algorithmically, the orthogonal/unitary transformation are effected by successive orthogonal row transformations (by means of Householder reflections ?? for general matrices, and Givens rotations ?? for banded matrices, see ?? for details).
- The orthogonal/unitary transformation with  $Q^H$  is applied to the augmented matrix  $(A, b) \in \mathbb{R}^{m,n+1}$  (Line 5 of Code 6.2.1), which is transformed into  $(R, \tilde{b})$ . Thus, the matrix  $Q$  need not be stored to

compute the transformed vector  $\tilde{\mathbf{b}} = \mathbf{Q}^H \mathbf{b}$ . Same idea is used for Gaussian elimination, see Code 1.6.23, Alg. 1.6.22, and also Alg. ??.

- Line 7: note that  $(\mathbf{Q}^H(\mathbf{A}, \mathbf{b}))_{n+2:m,:} = 0$  so that only the first  $n+1$  components of  $\tilde{\mathbf{b}}$  do not vanish.
- A QR-based algorithm is implemented in the least-squares-solver of the MATLAB-operator “\” (for dense matrices).

Alternative: Solving linear least squares problem (6.0.16) by SVD ( $\rightarrow$  Def. 7.5.8)

Most general setting:  $\mathbf{A} \in \mathbb{K}^{m,n}$ ,  $\text{rank}(\mathbf{A}) = r \leq \min\{m, n\}$ :

Here we drop the assumption of full rank of  $\mathbf{A}$ . This means that condition (ii) in the definition (6.0.16) of a linear least squares problem may be required for singling out a unique solution.

$$\text{SVD: } \mathbf{A} = [\mathbf{U}_1 \quad \mathbf{U}_2] \begin{bmatrix} \Sigma_r & 0 \\ 0 & 0 \end{bmatrix}$$

with  $\mathbf{U}_1 \in \mathbb{K}^{m,r}$ ,  $\mathbf{U}_2 \in \mathbb{K}^{m,m-r}$  with orthonormal columns,  
 $\Sigma_r = \text{diag}(\sigma_1, \dots, \sigma_r) \in \mathbb{R}^{r,r}$  (singular values, Def. 7.5.8),  
 $\mathbf{V}_1 \in \mathbb{K}^{n,r}$ ,  $\mathbf{V}_2 \in \mathbb{K}^{n,n-r}$  with orthonormal columns.

Then we use the invariance of the 2-norm of a vector with respect to multiplication with  $\mathbf{U} = [\mathbf{U}_1, \mathbf{U}_2]$ , see ??, together with the fact that  $\mathbf{U}$  is unitary, see Def. 4.2.2:

$$[\mathbf{U}_1, \mathbf{U}_2] \cdot \begin{bmatrix} (\mathbf{U}_1^H) \\ (\mathbf{U}_2^H) \end{bmatrix} = \mathbf{I} .$$

$$\|\mathbf{Ax} - \mathbf{b}\|_2 = \left\| [\mathbf{U}_1 \quad \mathbf{U}_2] \begin{bmatrix} \Sigma_r & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{V}_1^H \\ \mathbf{V}_2^H \end{bmatrix} \mathbf{x} - \mathbf{b} \right\|_2 = \left\| \begin{bmatrix} \Sigma_r \mathbf{V}_1^H \mathbf{x} \\ 0 \end{bmatrix} - \begin{bmatrix} \mathbf{U}_1^H \mathbf{b} \\ \mathbf{U}_2^H \mathbf{b} \end{bmatrix} \right\|_2 \quad (6.2.3)$$

Logical strategy: choose  $\mathbf{x}$  such that the first  $r$  components of  $\begin{bmatrix} \Sigma_r \mathbf{V}_1^H \mathbf{x} \\ 0 \end{bmatrix} - \begin{bmatrix} \mathbf{U}_1^H \mathbf{b} \\ \mathbf{U}_2^H \mathbf{b} \end{bmatrix}$  vanish:

$$\geq \text{(possibly underdetermined) } r \times n \text{ linear system} \quad \Sigma_r \mathbf{V}_1^H \mathbf{x} = \mathbf{U}_1^H \mathbf{b} . \quad (6.2.4)$$

To fix a unique solution in the case  $r < n$  we appeal to the **minimal norm condition** in (6.0.16): solution  $\mathbf{x}$  of (6.2.4) is unique up to contributions from

$$\text{Kern } \mathbf{V}_1^H = \text{Im}(\mathbf{V}_1)^\perp = \text{Im}(\mathbf{V}_2). \quad (6.2.5)$$

Since  $\mathbf{V}$  is unitary, the minimal norm solution is obtained by setting contributions from  $\text{Im}(\mathbf{V}_2)$  to zero, which amounts to choosing  $\mathbf{x} \in \text{Im}(\mathbf{V}_1)$ . This converts (6.2.4) into

$$\Sigma_r \underbrace{\mathbf{V}_1^H \mathbf{V}_1}_{=I} \mathbf{z} = \mathbf{U}_1^H \mathbf{b} \Rightarrow \mathbf{z} = \Sigma_r^{-1} \mathbf{U}_1^H \mathbf{b}.$$



solution

$$\mathbf{x} = \mathbf{V}_1 \Sigma_r^{-1} \mathbf{U}_1^H \mathbf{b}, \quad \|\mathbf{r}\|_2 = \left\| \mathbf{U}_2^H \mathbf{b} \right\|_2. \quad (6.2.6)$$

#### MATLAB-code 6.2.7: Solving LSQ problem via SVD

```
function y = lsqsvd(A,b)
[U,S,V] = svd(A,0);
sv = diag(S);
&\pnode{NRT}&r =
max(find(sv/sv(1) > eps));
y = V(:,1:r)*(diag(1./sv(1:r))*...
(U(:,1:r)'*b));
```

Practical implementation:

“numerical rank” test:

$$r = \max\{i : \sigma_i/\sigma_1 > \text{tol}\}$$

#### Remark 6.2.8.

Pseudoinverse and SVD → Rem. 6.0.19, [42, Ch. 12], [15, Sect. 4.7][Pseudoinverse and SVD]

The solution formula (6.2.6) directly yields a representation of the pseudoinverse  $\mathbf{A}^+$  (→ Def. 6.0.20) of any matrix  $\mathbf{A}$ :

#### Theorem 6.2.9. Pseudoinverse and SVD

If  $\mathbf{A} \in \mathbb{K}^{m,n}$  has the SVD decomposition (6.2.2), then  $\mathbf{A}^+ = \mathbf{V}_1 \Sigma_r^{-1} \mathbf{U}_1^H$  holds.

#### Remark 6.2.10 (Normal equations vs. orthogonal transformations method)

Superior numerical stability (→ Def. 1.5.80) of orthogonal transformations methods:

- Use orthogonal transformations methods for least squares problems (6.0.16), whenever  $\mathbf{A} \in \mathbb{R}^{m,n}$  dense and  $n$  small.

SVD/QR-factorization cannot exploit sparsity:

- Use normal equations in the expanded form (6.1.8)/(6.1.9), when  $\mathbf{A} \in \mathbb{R}^{m,n}$  sparse ( $\rightarrow$  Notation 1.7.1) and  $m, n$  big.

### Example 6.2.11 (Fit of hyperplanes)

This example studies the power and versatility of orthogonal transformations in the context of (generalized) least squares minimization problems.

The **Hesse normal form** of a hyperplane  $\mathcal{H}$  (= affine subspace of dimension  $d - 1$ ) in  $\mathbb{R}^d$  is:

$$\mathcal{H} = \{\mathbf{x} \in \mathbb{R}^d : c + \mathbf{n}^\top \mathbf{x} = 0\}, \quad \|\mathbf{n}\|_2 = 1. \quad (6.2.12)$$

► Euclidean distance of  $\mathbf{y} \in \mathbb{R}^d$  from the plane:  $\text{dist}(\mathcal{H}, \mathbf{y}) = |c + \mathbf{n}^\top \mathbf{y}|$ .  $(6.2.13)$

Goal: given the points  $\mathbf{y}_1, \dots, \mathbf{y}_m$ ,  $m > d$ , find  $\mathcal{H} \leftrightarrow \{c \in \mathbb{R}, \mathbf{n} \in \mathbb{R}^d, \|\mathbf{n}\|_2 = 1\}$ , such that

$$\sum_{j=1}^m \text{dist}(\mathcal{H}, \mathbf{y}_j)^2 = \sum_{j=1}^m |c + \mathbf{n}^\top \mathbf{y}_j|^2 \rightarrow \min. \quad (6.2.14)$$

Note: (6.2.14)  $\neq$  linear least squares problem due to constraint  $\|\mathbf{n}\|_2 = 1$ .

$$(6.2.14) \Leftrightarrow \left\| \begin{bmatrix} 1 & y_{1,1} & \cdots & y_{1,d} \\ 1 & y_{2,1} & \cdots & y_{2,d} \\ \vdots & \vdots & & \vdots \\ 1 & y_{m,1} & \cdots & y_{m,d} \end{bmatrix} \begin{bmatrix} c \\ \mathbf{n} \end{bmatrix} \right\|_2 \rightarrow \min \quad \text{under constraint } \|\mathbf{n}\|_2 = 1.$$

$\underbrace{\quad}_{=: \mathbf{A}}$

Step ①: QR-decomposition ( $\rightarrow$  Section ??)

$$\mathbf{A} := \begin{bmatrix} 1 & y_{1,1} & \cdots & y_{1,d} \\ 1 & y_{2,1} & \cdots & y_{2,d} \\ \vdots & \vdots & & \vdots \\ 1 & y_{m,1} & \cdots & y_{m,d} \end{bmatrix} = \mathbf{Q}\mathbf{R}, \quad \mathbf{R} := \begin{bmatrix} r_{11} & r_{12} & \cdots & \cdots & r_{1,d+1} \\ 0 & r_{22} & \cdots & \cdots & r_{2,d+1} \\ \vdots & \ddots & & & \vdots \\ 0 & \cdots & \cdots & & r_{d+1,d+1} \\ 0 & \cdots & \cdots & 0 & \\ \vdots & & & & \vdots \\ 0 & \cdots & \cdots & 0 & \end{bmatrix} \in \mathbb{R}^{m,d+1}.$$

$$\|\mathbf{Ax}\|_2 \rightarrow \min \Leftrightarrow \|\mathbf{Rx}\|_2 = \left\| \begin{bmatrix} r_{11} & r_{12} & \cdots & \cdots & r_{1,d+1} \\ 0 & r_{22} & \cdots & \cdots & r_{2,d+1} \\ \vdots & \ddots & & & \vdots \\ 0 & \cdots & \cdots & & r_{d+1,d+1} \\ 0 & \cdots & \cdots & 0 & \\ \vdots & & & & \vdots \\ 0 & \cdots & \cdots & 0 & \end{bmatrix} \begin{bmatrix} c \\ \mathbf{n} \end{bmatrix} \right\|_2 \rightarrow \min. \quad (6.2.15)$$

Step ② Note that necessarily (why?)

$$c \cdot r_{11} + n_1 \cdot r_{12} + \cdots + r_{1,d+1} \cdot n_d = 0.$$

This insight converts (6.2.15) to

$$\left\| \begin{bmatrix} r_{22} & r_{23} & \cdots & \cdots & r_{2,d+1} \\ 0 & r_{33} & \cdots & \cdots & r_{3,d+1} \\ \vdots & \ddots & & & \vdots \\ 0 & & & & r_{d+1,d+1} \end{bmatrix} \begin{bmatrix} n_1 \\ \vdots \\ n_d \end{bmatrix} \right\|_2 \rightarrow \min, \quad \|n\|_2 = 1. \quad (6.2.16)$$

(6.2.16) = problem of type (7.5.34), minimization on the Euclidean sphere.

➤ Solve (6.2.16) using SVD !

Note: Since  $r_{11} = \|(\mathbf{A})_{:,1}\|_2 = \sqrt{m} \neq 0 \Rightarrow c = -r_{11}^{-1} \sum_{j=1}^d r_{1,j+1} n_j$ .

#### MATLAB-code 6.2.17: (Generalized) distance fitting of a hyperplane: solution of (6.2.18)

```
function [c,n] = clsq(A,dim);
% Solves constrained linear least squares problem (6.2.18) with dim passing
% d
[m,p] = size(A);
if p < dim+1, error ('not enough unknowns'); end;
if m < dim, error ('not enough equations'); end;
m = min(m, p);
R = triu(qr(A)); % First step: orthogonal transformation, see
% Code 6.2.1
[U,S,V] = svd(R(p-dim+1:m,p-dim+1:p)); % Solve (6.2.16)
n = V(:,dim);
c = -R(1:p-dim,1:p-dim)\R(1:p-dim,p-dim+1:p)*n;
```

Eq. (6.2.18) solves the general problem: For  $\mathbf{A} \in \mathbb{K}^{m,n}$  find  $\mathbf{n} \in \mathbb{R}^d$ ,  $\mathbf{c} \in \mathbb{R}^{n-d}$  such that

$$\left\| \mathbf{A} \begin{bmatrix} \mathbf{c} \\ \mathbf{n} \end{bmatrix} \right\|_2 \rightarrow \min \quad \text{with constraint } \|\mathbf{n}\|_2 = 1. \quad (6.2.18)$$

## 6.3 Total Least Squares

Given: overdetermined linear system of equations  $\mathbf{Ax} = \mathbf{b}$ ,  $\mathbf{A} \in \mathbb{R}^{m,n}$ ,  $\mathbf{b} \in \mathbb{R}^m$ ,  $m \geq n$ .

Known: LSE solvable  $\Leftrightarrow \mathbf{b} \in \text{Im}(\mathbf{A})$ , if  $\mathbf{A}, \mathbf{b}$  were not perturbed,  
but  $\mathbf{A}, \mathbf{b}$  are perturbed (measurement errors).

Sought: Solvable overdetermined system of equations  $\widehat{\mathbf{Ax}} = \widehat{\mathbf{b}}$ ,  $\widehat{\mathbf{A}} \in \mathbb{R}^{m,n}$ ,  $\widehat{\mathbf{b}} \in \mathbb{R}^m$ ,  
“nearest” to  $\mathbf{Ax} = \mathbf{b}$ .

- ☞ least squares problem “turned upside down”: now we are allowed to tamper with system matrix and right hand side vector!

**Total least squares problem:**

Given:  $\mathbf{A} \in \mathbb{R}^{m,n}$ ,  $m \geq n$ ,  $\text{rank}(\mathbf{A}) = n$ ,  $\mathbf{b} \in \mathbb{R}^m$ ,

find:  $\widehat{\mathbf{A}} \in \mathbb{R}^{m,n}$ ,  $\widehat{\mathbf{b}} \in \mathbb{R}^m$  with

$$\left\| \underbrace{\begin{bmatrix} \mathbf{A} & \mathbf{b} \end{bmatrix}}_{=: \mathbf{C}} - \underbrace{\begin{bmatrix} \widehat{\mathbf{A}} & \widehat{\mathbf{b}} \end{bmatrix}}_{=: \widehat{\mathbf{C}}} \right\|_F \rightarrow \min , \quad \widehat{\mathbf{b}} \in \text{Im } \widehat{\mathbf{A}} .$$

$$\widehat{\mathbf{b}} \in \text{Im } \widehat{\mathbf{A}} \Rightarrow \text{rank}(\widehat{\mathbf{C}}) = n \quad \blacktriangleright \quad (6.3.1) \Rightarrow \min_{\text{rank}(\widehat{\mathbf{C}})=n} \left\| \mathbf{C} - \widehat{\mathbf{C}} \right\|_F . \quad (6.3.1)$$



$\widehat{\mathbf{C}}$  is the rank- $n$  best approximation of  $\mathbf{C}$ !

Thm. 7.5.39 ► use the SVD decomposition of  $\mathbf{C}$  to construct  $\widehat{\mathbf{C}}$ :

$$\mathbf{C} = \mathbf{U} \Sigma \mathbf{V}^H = \sum_{j=1}^{n+1} \sigma_j(\mathbf{U})_{:,j} (\mathbf{V})_{:,j}^H \xrightarrow{\text{Thm. 7.5.39}} \widehat{\mathbf{C}} = \sum_{j=1}^n \sigma_j(\mathbf{U})_{:,j} (\mathbf{V})_{:,j}^H . \quad (6.3.2)$$

$$\xrightarrow{\mathbf{V} \text{ orthogonal}} \widehat{\mathbf{C}}(\mathbf{V})_{:,n+1} = 0 . \quad (6.3.3)$$

Recall interpretation:  $\widehat{\mathbf{A}} = (\widehat{\mathbf{C}})_{1:n,1:n}$ ,  $\widehat{\mathbf{b}} = \widehat{\mathbf{C}}_{1:n,n+1}$

► (6.3.3) provides solution

$$\mathbf{x} := \widehat{\mathbf{A}}^{-1} \widehat{\mathbf{b}} = (\mathbf{V})_{:,n+1} / (\mathbf{V})_{n+1,n+1} . \quad (6.3.4)$$

#### MATLAB-code 6.3.5: Total least squares via SVD

```
function x = lsqtotal(A,b);
% computes only solution x of fitted consistent LSE
[m,n]=size(A);
[U, Sigma, V] = svd([A,b]); % see (6.3.2)
s = V(n+1,n+1);
if s == 0, error('No solution'); end
x = -V(1:n,n+1)/s; % see (6.3.4)
```

## 6.4 Constrained Least Squares

Section 3.1: interpolation ➤ linear system of equations (3.1.13),  
Ex. 6.0.3: linear data fitting ➤ linear least squares problem (6.0.5).

What if *some* data points are considered accurate?

► mix interpolation and linear least squares fitting!

► linear least squares problem with linear constraint

*Linear least squares problem with linear constraint:*

Given:  $\mathbf{A} \in \mathbb{R}^{m,n}$ ,  $m \geq n$ ,  $\text{rank}(\mathbf{A}) = n$ ,  $\mathbf{b} \in \mathbb{R}^m$ ,  
 $\mathbf{C} \in \mathbb{R}^{p,n}$ ,  $p < n$ ,  $\text{rank}(\mathbf{C}) = p$ ,  $\mathbf{d} \in \mathbb{R}^p$

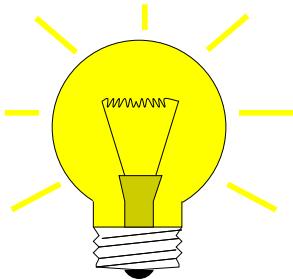
Find:  $\mathbf{x} \in \mathbb{R}^n$  with

$$\|\mathbf{Ax} - \mathbf{b}\|_2 \rightarrow \min \quad , \quad \boxed{\mathbf{Cx} = \mathbf{d}} . \quad (6.4.1)$$

Linear constraint

### 6.4.1 Solution via normal equations

Recall important technique from multidimensional calculus: [Lagrange multipliers](#), see [77, Sect. 7.9].



Idea: coupling the constraint using the [Lagrange multiplier](#)  $\mathbf{m} \in \mathbb{R}^p$

$$\mathbf{x} = \underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} \underset{\mathbf{m} \in \mathbb{R}^p}{\max} L(\mathbf{x}, \mathbf{m}) , \quad (6.4.2)$$

$$L(\mathbf{x}, \mathbf{m}) := \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|^2 + \mathbf{m}^H (\mathbf{Cx} - \mathbf{d}) . \quad (6.4.3)$$

The simple heuristics behind Lagrange multipliers:

$$\max_{\mathbf{m} \in \mathbb{R}^p} L(\mathbf{x}, \mathbf{m}) = \infty, \quad \text{in case } \mathbf{Cx} \neq \mathbf{d} !$$

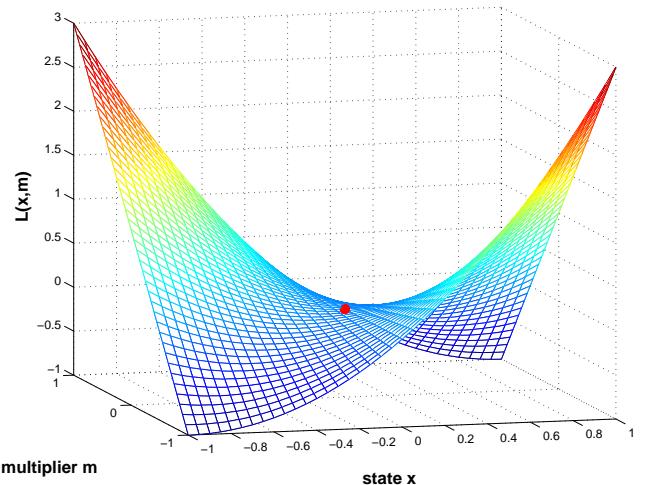
► A minimum in (6.4.2) can only attained, if the constraint is satisfied!

(6.4.2) is called a [saddle point problem](#).

Solution of min-max problem:

[saddle point](#)  
 $F(x, m) = x^2 - 2xm$

Note that the function is “flat” in the saddle point •



Necessary (and sufficient) condition for the minimizer ( $\rightarrow$  Section 6.1)

$$\frac{\partial L}{\partial \mathbf{x}}(\mathbf{x}, \mathbf{m}) = \mathbf{A}^H(\mathbf{Ax} - \mathbf{b}) + \mathbf{C}^H \mathbf{m} \stackrel{!}{=} 0 \quad , \quad \frac{\partial L}{\partial \mathbf{m}}(\mathbf{x}, \mathbf{m}) = \mathbf{C} \mathbf{x} - \mathbf{d} \stackrel{!}{=} 0 . \quad (6.4.4)$$



$$\begin{bmatrix} \mathbf{A}^H \mathbf{A} & \mathbf{C}^H \\ \mathbf{C} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{m} \end{bmatrix} = \begin{bmatrix} \mathbf{A}^H \mathbf{b} \\ \mathbf{d} \end{bmatrix} \quad \text{Augmented normal equations (matrix saddle point problem)} \quad (6.4.5)$$

Algorithm for (6.4.5) (based on block-LU-decomposition):

$$\begin{bmatrix} \mathbf{A}^H \mathbf{A} & \mathbf{C}^H \\ \mathbf{C} & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{R}^H & 0 \\ \mathbf{G} & -\mathbf{S}^H \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{G}^H \\ 0 & \mathbf{S} \end{bmatrix}, \quad \mathbf{R}, \mathbf{S} \in \mathbb{R}^{n,n} \text{ upper triangular matrix, } \mathbf{G} \in \mathbb{R}^{p,n}.$$

$\mathbf{R}$  from  $\mathbf{R}^H \mathbf{R} = \mathbf{A}^H \mathbf{A} \rightarrow$  Cholesky decomposition  $\rightarrow$  Section 1.8,  
 $\mathbf{G}$  from  $\mathbf{R}^H \mathbf{G}^H = \mathbf{C}^H \rightarrow$   $n$  forward substitution  $\rightarrow$  Section 1.6.2.2,  
 $\mathbf{S}$  from  $\mathbf{S}^H \mathbf{S} = \mathbf{G} \mathbf{G}^H \rightarrow$  Cholesky decomposition  $\rightarrow$  Section 1.8.

Caution Section 6.1: the computation of  $\mathbf{A}^H \mathbf{A}$  can be expensive and problematic!  
(remedy through introduction of a new unknown  $\mathbf{r} = \mathbf{A}\mathbf{x} - \mathbf{b}$ , cf. (6.1.8))

$$\begin{bmatrix} -\mathbf{I} & \mathbf{A} & 0 \\ \mathbf{A}^H & 0 & \mathbf{C}^H \\ 0 & \mathbf{C} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \\ \mathbf{m} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \\ \mathbf{d} \end{bmatrix}. \quad (6.4.6)$$

## 6.4.2 Solution via SVD

Idea: identify the subspace in which the solution can vary without violating the constraint. Since  $\mathbf{C}$  has *full rank*, this subspace agrees with the nullspace/kernel of  $\mathbf{C}$ .

Lemma 7.5.14 ➤ SVD can be used to compute  $\text{Kern } \mathbf{C}$

① Compute orthonormal basis of  $\text{Kern } \mathbf{C}$  using SVD ( $\rightarrow$  Lemma 7.5.14, (6.2.2)):

$$\mathbf{C} = \mathbf{U}[\Sigma \ 0] \begin{bmatrix} \mathbf{V}_1^H \\ \mathbf{V}_2^H \end{bmatrix}, \quad \mathbf{U} \in \mathbb{R}^{p,p}, \Sigma \in \mathbb{R}^{p,p}, \mathbf{V}_1 \in \mathbb{R}^{n,p}, \mathbf{V}_2 \in \mathbb{R}^{n,n-p}$$

►  $\text{Kern } \mathbf{C} = \text{Im } \mathbf{V}_2$ .

and the **particular solution** of the constraint equation

$$\mathbf{x}_0 := \mathbf{V}_1 \Sigma^{-1} \mathbf{U}^H \mathbf{d}.$$

Representation of the solution  $\mathbf{x}$  of (6.4.1):  $\mathbf{x} = \mathbf{x}_0 + \mathbf{V}_2 \mathbf{y}, \mathbf{y} \in \mathbb{R}^{n-p}$ .

② Insert this representation in (6.4.1) ➤ standard linear least squares

$$\|\mathbf{A}(\mathbf{x}_0 + \mathbf{V}_2 \mathbf{y}) - \mathbf{b}\|_2 \rightarrow \min \Leftrightarrow \|\mathbf{A}\mathbf{V}_2 \mathbf{y} - (\mathbf{b} - \mathbf{A}\mathbf{x}_0)\| \rightarrow \min.$$

## 6.5 Non-linear Least Squares [15, Ch. 6]

### Example 6.5.1 (Non-linear data fitting (parametric statistics) → Ex. 6.0.1 revisited)

Given: data points  $(t_i, y_i), i = 1, \dots, m$  with measurement errors.

Known:  $y = f(\mathbf{x}, t)$  through a function  $f : \mathbb{R}^n \times \mathbb{R} \mapsto \mathbb{R}$  depending non-linearly and smoothly on parameters  $\mathbf{x} \in \mathbb{R}^n$ .

Example:  $f(t) = x_1 + x_2 \exp(-x_3 t), n = 3$ .

Determine parameters by non-linear least squares data fitting:

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} \sum_{i=1}^m |f(\mathbf{x}, t_i) - y_i|^2 = \underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} \frac{1}{2} \|F(\mathbf{x})\|_2^2, \quad (6.5.2)$$

with  $F(\mathbf{x}) = \begin{bmatrix} f(\mathbf{x}, t_1) - y_1 \\ \vdots \\ f(\mathbf{x}, t_m) - y_m \end{bmatrix}.$

### Non-linear least squares problem

Given:  $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^m, m, n \in \mathbb{N}, m > n$ .

Find:  $\mathbf{x}^* \in D: \mathbf{x}^* = \underset{\mathbf{x} \in D}{\operatorname{argmin}} \Phi(\mathbf{x}), \Phi(\mathbf{x}) := \frac{1}{2} \|F(\mathbf{x})\|_2^2$ . (6.5.3)

Terminology:  $D \hat{=} \text{parameter space}, x_1, \dots, x_n \hat{=} \text{parameter}$ .

As in the case of linear least squares problems (→ Rem. 6.0.11): a non-linear least squares problem is related to an overdetermined non-linear system of equations  $F(\mathbf{x}) = 0$ .

As for non-linear systems of equations (→ Chapter 2): existence and uniqueness of  $\mathbf{x}^*$  in (6.5.3) has to be established in each concrete case!

We require “independence for each parameter”: → Rem. 6.0.18

$\exists$  neighbourhood  $\mathcal{U}(\mathbf{x}^*)$  such that  $DF(\mathbf{x})$  has full rank  $n \quad \forall \mathbf{x} \in \mathcal{U}(\mathbf{x}^*)$ . (6.5.4)

(It means: the columns of the Jacobi matrix  $DF(\mathbf{x})$  are linearly independent.)

If (6.5.4) is not satisfied, then the parameters are redundant in the sense that fewer parameters would be enough to model the same dependence (locally at  $\mathbf{x}^*$ ), cf. Rem. 6.0.18.

### 6.5.1 (Damped) Newton method

$$\Phi(\mathbf{x}^*) = \min \Rightarrow \mathbf{grad} \Phi(\mathbf{x}) = 0, \mathbf{grad} \Phi(\mathbf{x}) := (\frac{\partial \Phi}{\partial x_1}(\mathbf{x}), \dots, \frac{\partial \Phi}{\partial x_n}(\mathbf{x}))^T \in \mathbb{R}^n.$$

Simple idea: use Newton's method ( $\rightarrow$  Section 2.4) to determine a zero of  $\mathbf{grad} \Phi : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$ .

Newton iteration (2.4.1) for non-linear system of equations  $\mathbf{grad} \Phi(\mathbf{x}) = 0$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - H\Phi(\mathbf{x}^{(k)})^{-1} \mathbf{grad} \Phi(\mathbf{x}^{(k)}) , \quad (H\Phi(\mathbf{x}) = \text{Hessian matrix}) . \quad (6.5.5)$$

Expressed in terms of  $F : \mathbb{R}^n \mapsto \mathbb{R}^m$  from (6.5.3):

$$\begin{aligned} \text{chain rule (2.4.8)} &\Rightarrow \mathbf{grad} \Phi(\mathbf{x}) = DF(\mathbf{x})^T F(\mathbf{x}) , \\ \text{product rule (2.4.9)} &\Rightarrow H\Phi(\mathbf{x}) := D(\mathbf{grad} \Phi)(\mathbf{x}) = DF(\mathbf{x})^T DF(\mathbf{x}) + \sum_{j=1}^m F_j(\mathbf{x}) D^2 F_j(\mathbf{x}) , \\ &\Downarrow \\ (H\Phi(\mathbf{x}))_{i,k} &= \sum_{j=1}^n \frac{\partial^2 F_j}{\partial x_i \partial x_k}(\mathbf{x}) F_j(\mathbf{x}) + \frac{\partial F_j}{\partial x_k}(\mathbf{x}) \frac{\partial F_j}{\partial x_i}(\mathbf{x}) . \end{aligned}$$

Recommendation, cf. § 2.4.5: when in doubt, differentiate components of matrices and vectors!

The above derivative formulas allow to rewrite (6.5.5) in concrete terms:

► For Newton iterate  $\mathbf{x}^{(k)}$ : Newton correction  $\mathbf{s} \in \mathbb{R}^n$  from LSE

$$\underbrace{\left( DF(\mathbf{x}^{(k)})^T DF(\mathbf{x}^{(k)}) + \sum_{j=1}^m F_j(\mathbf{x}^{(k)}) D^2 F_j(\mathbf{x}^{(k)}) \right)}_{=H\Phi(\mathbf{x}^{(k)})} \mathbf{s} = - \underbrace{DF(\mathbf{x}^{(k)})^T F(\mathbf{x}^{(k)})}_{=\mathbf{grad} \Phi(\mathbf{x}^{(k)})} . \quad (6.5.6)$$

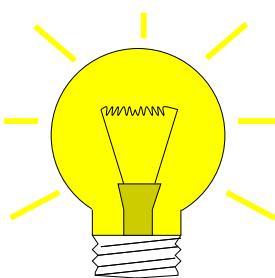
### Remark 6.5.7 (Newton method and minimization of quadratic functional)

Newton's method (6.5.5) for (6.5.3) can be read as successive minimization of a local quadratic approximation of  $\Phi$ :

$$\begin{aligned} \Phi(\mathbf{x}) &\approx Q(\mathbf{s}) := \Phi(\mathbf{x}^{(k)}) + \mathbf{grad} \Phi(\mathbf{x}^{(k)})^T \mathbf{s} + \frac{1}{2} \mathbf{s}^T H\Phi(\mathbf{x}^{(k)}) \mathbf{s} , \\ \mathbf{grad} Q(\mathbf{s}) = 0 &\Leftrightarrow H\Phi(\mathbf{x}^{(k)}) \mathbf{s} + \mathbf{grad} \Phi(\mathbf{x}^{(k)}) = 0 \Leftrightarrow (6.5.6) . \end{aligned} \quad (6.5.8)$$

► So we deal with yet another model function method ( $\rightarrow$  Section 2.3.2) with quadratic model function for  $Q$ .

## 6.5.2 Gauss-Newton method



Idea:

local linearization of  $F$ :  $F(\mathbf{x}) \approx F(\mathbf{y}) + DF(\mathbf{y})(\mathbf{x} - \mathbf{y})$

► sequence of linear least squares problems

$$\underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} \|F(\mathbf{x})\|_2 \quad \text{is approximated by} \quad \underbrace{\underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} \|F(\mathbf{x}_0) + DF(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)\|_2}_{(\spadesuit)} ,$$

where  $\mathbf{x}_0$  is an approximation of the solution  $\mathbf{x}^*$  of (6.5.3).

$$(\spadesuit) \Leftrightarrow \underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} \|\mathbf{Ax} - \mathbf{b}\| \quad \text{with} \quad \mathbf{A} := DF(\mathbf{x}_0) \in \mathbb{R}^{m,n}, \quad \mathbf{b} := -F(\mathbf{x}_0) + DF(\mathbf{x}_0)\mathbf{x}_0 \in \mathbb{R}^m.$$

This is a linear least squares problem of the form (6.0.16).

Note: (6.5.4)  $\Rightarrow \mathbf{A}$  has full rank, if  $\mathbf{x}_0$  sufficiently close to  $\mathbf{x}^*$ .

Note: This approach is different from local quadratic approximation of  $\Phi$  underlying Newton's method for (6.5.3), see Section 6.5.1, Rem. 6.5.7.

► **Gauss-Newton iteration** (under assumption (6.5.4))

$$\begin{aligned} & \text{Initial guess } \mathbf{x}^{(0)} \in D \\ & \mathbf{x}^{(k+1)} := \underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} \|F(\mathbf{x}^{(k)}) + DF(\mathbf{x}^{(k)})(\mathbf{x} - \mathbf{x}^{(k)})\|_2 . \end{aligned} \quad (6.5.9)$$

↑  
linear least squares problem

MATLAB-` used to solve linear least squares problem in each step:

for  $\mathbf{A} \in \mathbb{R}^{m,n}$

$$\begin{aligned} \mathbf{x} &= \mathbf{A} \backslash \mathbf{b} \\ &\Downarrow \\ \mathbf{x} &\text{ minimizer of } \|\mathbf{Ax} - \mathbf{b}\|_2 \\ &\text{with minimal 2-norm} \end{aligned}$$

#### MATLAB-code 6.5.10: template for Gauss-Newton method

```
function x = gn(x,F,J,tol)
s = J(x)\F(x); %
x = x-s;
while (norm(s) > tol*norm(x)) %
    s = J(x)\F(x); %
    x = x-s;
end
```

Comments on Code 6.5.10:

- Argument  $x$  passes initial guess  $\mathbf{x}^{(0)} \in \mathbb{R}^n$ , argument  $F$  must be a *handle* to a function  $F : \mathbb{R}^n \mapsto \mathbb{R}^m$ , argument  $J$  provides the Jacobian of  $F$ , namely  $DF : \mathbb{R}^n \mapsto \mathbb{R}^{m,n}$ , argument  $tol$  specifies the tolerance for termination
- Line 4: iteration terminates if relative norm of correction is below threshold specified in  $tol$ .

Note: Code 6.5.10 also implements Newton's method ( $\rightarrow$  Section 2.4.1) in the case  $m = n$ !

Summary:

Advantage of the Gauss-Newton method : second derivative of  $F$  not needed.

Drawback of the Gauss-Newton method : no local quadratic convergence.

#### Example 6.5.11 (Non-linear data fitting (II)) → Ex. 6.5.1)

Non-linear data fitting problem (6.5.2) for  $f(t) = x_1 + x_2 \exp(-x_3 t)$ .

$$F(\mathbf{x}) = \begin{bmatrix} x_1 + x_2 \exp(-x_3 t_1) - y_1 \\ \vdots \\ x_1 + x_2 \exp(-x_3 t_m) - y_m \end{bmatrix} : \mathbb{R}^3 \mapsto \mathbb{R}^m, DF(\mathbf{x}) = \begin{bmatrix} 1 & e^{-x_3 t_1} & -x_2 t_1 e^{-x_3 t_1} \\ \vdots & \vdots & \vdots \\ 1 & e^{-x_3 t_m} & -x_2 t_m e^{-x_3 t_m} \end{bmatrix}$$

Numerical experiment:

convergence of the Newton method, damped Newton method ( $\rightarrow$  Section 2.4.4) and Gauss-Newton method for different initial values

#### MATLAB-code 6.5.12:

```
1 rand ('seed', 0);
2 t = (1:0.3:7)';
3 y = x(1) + x(2)*exp(-x(3)*t);
4 y =
y+0.1*(rand(length(y), 1)-0.5);
```

- \* initial value  $(1.8, 1.8, 0.1)^T$  (red curves, blue curves)
- \* initial value  $(1.5, 1.5, 0.1)^T$  (cyan curves, green curves)

First experiment ( $\rightarrow$  Section 6.5.1): iterative solution of non-linear least squares data fitting problem by means of the Newton method (6.5.6) and the damped Newton method from Section 2.4.4

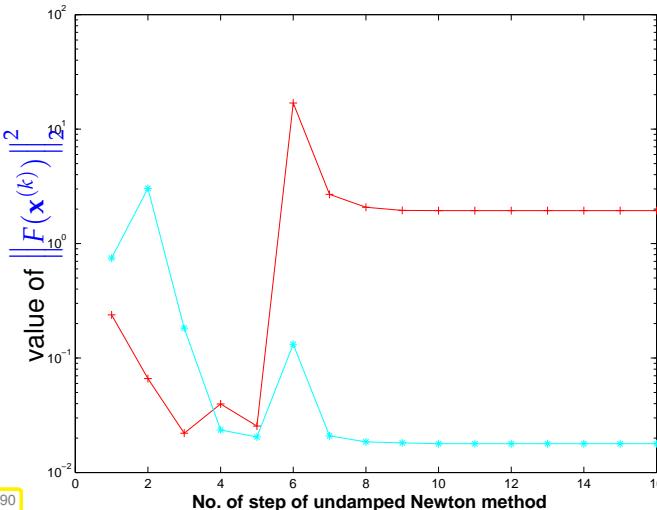


Fig. 190

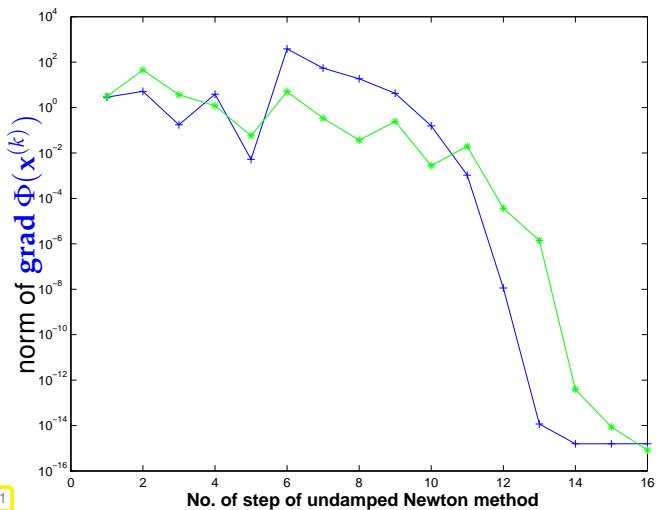


Fig. 191

Convergence behaviour of plain Newton method:

initial value  $(1.8, 1.8, 0.1)^T$  (red curve)  $\rightarrow$  Newton method caught in local minimum,  
initial value  $(1.5, 1.5, 0.1)^T$  (cyan curve)  $\rightarrow$  fast (locally quadratic) convergence.

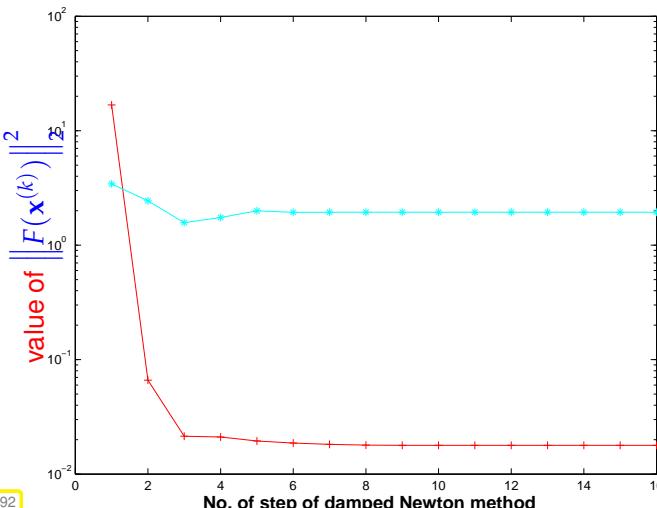


Fig. 192

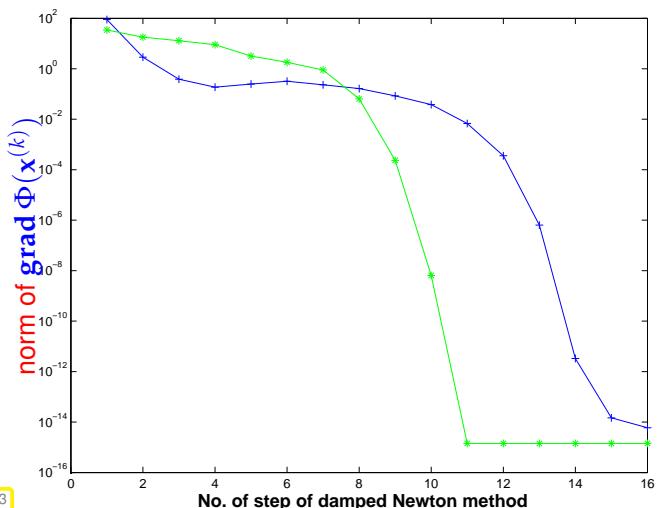


Fig. 193

Convergence behavior of damped Newton method:

initial value  $(1.8, 1.8, 0.1)^T$  (red curve)  $\Rightarrow$  fast (locally quadratic) convergence,  
 initial value  $(1.5, 1.5, 0.1)^T$  (cyan curve)  $\Rightarrow$  Newton method caught in local minimum.

Second experiment: iterative solution of non-linear least squares data fitting problem by means of the Gauss-Newton method (6.5.9), see Code 6.5.10.

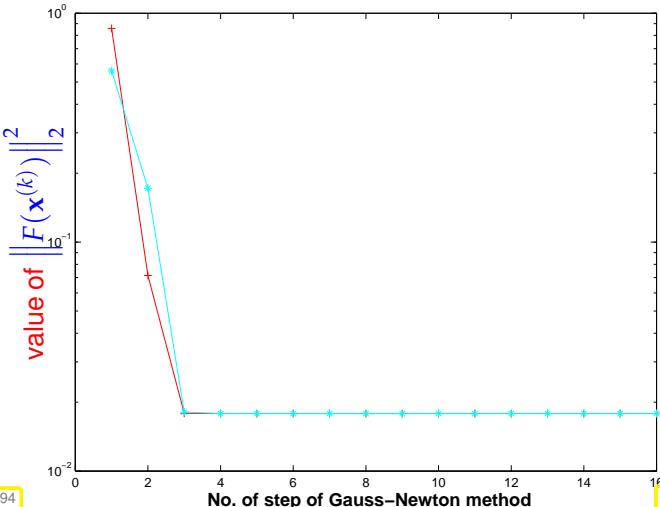


Fig. 194

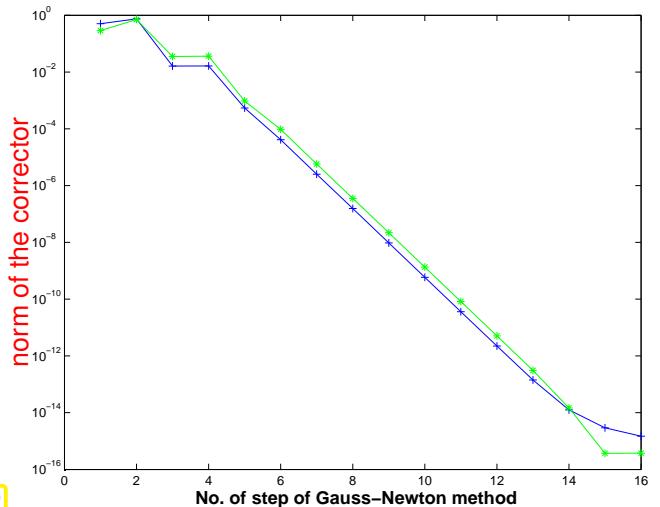


Fig. 195

We observe: linear convergence for all initial values, cf. Def. 2.1.9, Rem. 2.1.13.

### 6.5.3 Trust region method (Levenberg-Marquardt method)

As in the case of Newton's method for non-linear systems of equations, see Section 2.4.4: often overshooting of Gauss-Newton corrections occurs.

Remedy as in the case of Newton's method: **damping**.

Idea: damping of the Gauss-Newton correction in (6.5.9) using a **penalty term**

$$\text{instead of } \|F(\mathbf{x}^{(k)}) + DF(\mathbf{x}^{(k)})\mathbf{s}\|^2 \quad \text{minimize} \quad \|F(\mathbf{x}^{(k)}) + DF(\mathbf{x}^{(k)})\mathbf{s}\|^2 + \lambda \|\mathbf{s}\|_2^2.$$

$\lambda > 0 \doteq$  penalty parameter (how to choose it ?  $\rightarrow$  heuristic)

$$\lambda = \gamma \|F(\mathbf{x}^{(k)})\|_2, \quad \gamma := \begin{cases} 10 & , \text{ if } \|F(\mathbf{x}^{(k)})\|_2 \geq 10, \\ 1 & , \text{ if } 1 < \|F(\mathbf{x}^{(k)})\|_2 < 10, \\ 0.01 & , \text{ if } \|F(\mathbf{x}^{(k)})\|_2 \leq 1. \end{cases}$$

$\Rightarrow$  Modified (regularized) equation for the corrector  $\mathbf{s}$ :

$$(DF(\mathbf{x}^{(k)})^T DF(\mathbf{x}^{(k)}) + \lambda \mathbf{I})\mathbf{s} = -DF(\mathbf{x}^{(k)})F(\mathbf{x}^{(k)}). \quad (6.5.13)$$

# Chapter 7

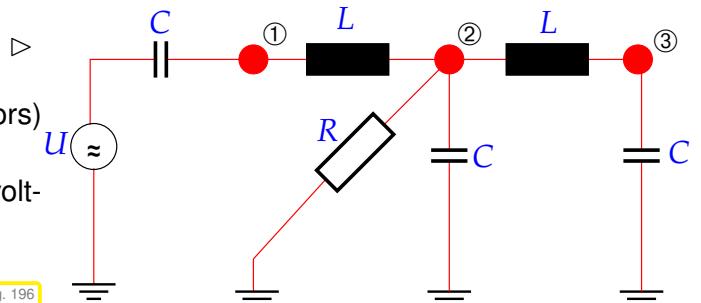
## Eigenvalues



*Supplementary reading.* [8] offers comprehensive presentation of numerical methods for the solution of eigenvalue problems from an algorithmic point of view.

### Example 7.0.1 (Resonances of linear electric circuits)

Simple electric circuit, cf. Ex. 1.6.3



Ex. 1.6.3: nodal analysis of linear ( $\leftrightarrow$  composed of resistors, inductors, capacitors) electric circuit **in frequency domain (at angular frequency  $\omega > 0$ )**, see (1.6.6)

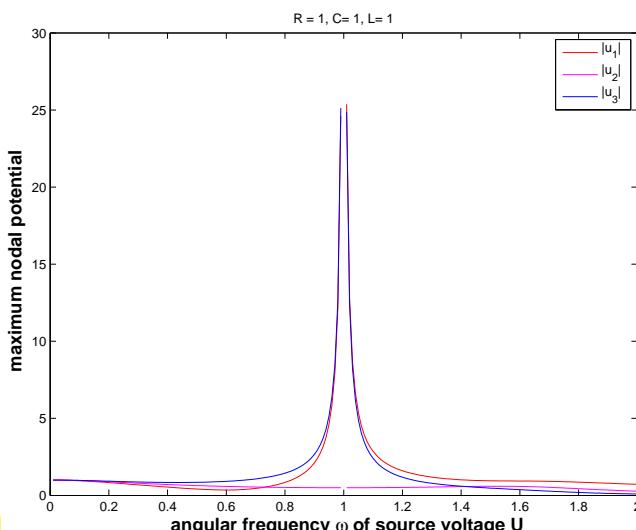
➢ linear system of equations for nodal potentials with complex system matrix **A**

For circuit of Code 7.0.3: three unknown nodal potentials

➢ system matrix from nodal analysis at angular frequency  $\omega > 0$ :

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} i\omega C + \frac{1}{i\omega L} & -\frac{1}{i\omega L} & 0 \\ -\frac{1}{i\omega L} & i\omega C + \frac{1}{R} + \frac{2}{i\omega L} & -\frac{1}{i\omega L} \\ 0 & -\frac{1}{i\omega L} & i\omega C + \frac{1}{i\omega L} \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{1}{R} & 0 \\ 0 & 0 & 0 \end{bmatrix} + i\omega \begin{bmatrix} C & 0 & 0 \\ 0 & C & 0 \\ 0 & 0 & C \end{bmatrix} + 1/i\omega \begin{bmatrix} \frac{1}{L} & -\frac{1}{L} & 0 \\ -\frac{1}{L} & \frac{2}{L} & -\frac{1}{L} \\ 0 & -\frac{1}{L} & \frac{1}{L} \end{bmatrix}. \end{aligned}$$

$$\mathbf{A}(\omega) := \mathbf{W} + i\omega\mathbf{C} - i\omega^{-1}\mathbf{S} \quad , \quad \mathbf{W}, \mathbf{C}, \mathbf{S} \in \mathbb{R}^{n,n} \text{ symmetric} . \quad (7.0.2)$$



$\triangleleft$  plot of  $|u_i(U)|$ ,  $i = 1, 2, 3$  for  $R = L = C = 1$  (scaled model)

Blow-up of some nodal potentials for certain  $\omega$  !

### MATLAB-code 7.0.3: Computation of nodal potential for circuit of Code 7.0.3

```

1 function rescirc(R,L,C)
2 % Ex. ???: Numerical nodal analysis of the resonant circuit
3 % R, L, C  $\hat{=}$  network component parameters
4
5 Z = 1/R; K = 1/L;
6
7 % Matrices W, C, S for nodal analysis of circuit
8 Wmat = [0 0 0; 0 Z 0; 0 0 0];
9 Cmat = [C 0 0; 0 C 0; 0 0 C];
10 Smat = [K -K 0; -K 2*K -K; 0 -K K];
11 % System matrix from nodal analysis
12 Amat = @(w) (Wmat+i*w*Cmat+Smat/(i*w));
13
14 % Scanning source currents
15 res = [];
16 for w=0.01:0.01:2
17 res = [res; w, abs(Amat(w)\[C;0;0])'];
18 end
19
20 figure('name','resonant circuit');
21 plot(res(:,1),res(:,2),'r-',res(:,1),res(:,3),'m-',res(:,1),res(:,4),'b-')
22 xlabel('{\bf angular frequency} \omega of source voltage U','fontsize',14);
23 ylabel('{\bf maximum nodal potential}','fontsize',14);
24 title(sprintf('R = %d, C= %d, L= %d',R,L,C));
25 legend('|u_1|','|u_2|','|u_3|');
26
27 print -depsc2 '../PICTURES/rescircpot.eps'
28
29 % Solving generalized eigenvalue problem (7.0.5)
30 Zmat = zeros(3,3); Imat = eye(3,3);
31 % Assemble 6x6-matrices M and B
32 Mmat = [Wmat,Smat; Imat, Zmat];
33 Bmat = [-i*Cmat, Zmat; Zmat , i*Imat];

```

```

34 % Solve generalized eigenvalue problem, cf. (7.0.6)
35 omega = eig(Mmat,Bmat);
36
37 figure('name','resonances');
38 plot(real(omega),imag(omega),'r*'); hold on;
39 ax = axis;
40 plot([ax(1) ax(2)], [0 0], 'k-');
41 plot([ 0 0], [ax(3) ax(4)], 'k-');
42 grid on;
43 xlabel('{\bf Re}(\omega)', 'fontsize',14);
44 ylabel('{\bf Im}(\omega)', 'fontsize',14);
45 title(sprintf('R = %d, C= %d, L= %d', R,L,C));
46 legend('omega');
47
48 print -depsc2 '../PICTURES/rescirmomega.eps'

```

$$\text{resonant frequencies} = \omega \in \{\omega \in \mathbb{R}: \mathbf{A}(\omega) \text{ singular}\}$$

If the circuit is operated at a real resonant frequency, the circuit equations will not possess a solution. Of course, the real circuit will always behave in a well-defined way, but the linear model will break down due to extremely large currents and voltages. In an experiment this breakdown manifests itself as a rather explosive meltdown of circuits components. Hence, it is vital to determine resonant frequencies of circuits in order to avoid their destruction.

→ relevance of numerical methods for solving:

$$\text{Find } \omega \in \mathbb{C} \setminus \{0\}: \mathbf{W} + i\omega \mathbf{C} + \frac{1}{i\omega} \mathbf{S} \text{ singular.}$$

This is a **quadratic eigenvalue problem**: find  $\mathbf{x} \neq 0, \omega \in \mathbb{C} \setminus \{0\}$ ,

$$\mathbf{A}(\omega)\mathbf{x} = (\mathbf{W} + i\omega \mathbf{C} + \frac{1}{i\omega} \mathbf{S})\mathbf{x} = 0. \quad (7.0.4)$$

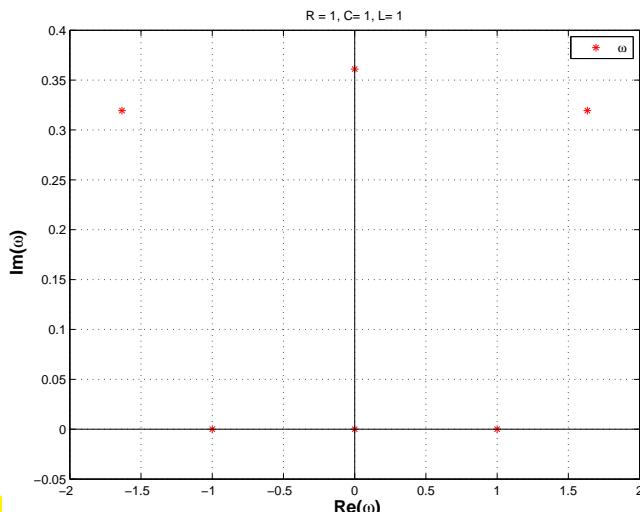
Substitution:  $\mathbf{y} = \frac{1}{i\omega} \mathbf{x} \leftrightarrow \mathbf{x} = i\omega \mathbf{y}$  [78, Sect. 3.4]:

$$(7.0.4) \Leftrightarrow \underbrace{\begin{bmatrix} \mathbf{W} & \mathbf{S} \\ \mathbf{I} & 0 \end{bmatrix}}_{:= \mathbf{M}} \underbrace{\begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix}}_{:= \mathbf{z}} = \omega \underbrace{\begin{bmatrix} -i\mathbf{C} & 0 \\ 0 & -i\mathbf{I} \end{bmatrix}}_{:= \mathbf{B}} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} \quad (7.0.5)$$

➤ **generalized linear eigenvalue problem** of the form: find  $\omega \in \mathbb{C}, \mathbf{z} \in \mathbb{C}^{2n} \setminus \{0\}$  such that

$$\mathbf{M}\mathbf{z} = \omega \mathbf{B}\mathbf{z}. \quad (7.0.6)$$

In this example one is mainly interested in the **eigenvalues**  $\omega$ , whereas the **eigenvectors**  $\mathbf{z}$  usually need not be computed.



▷ resonant frequencies for circuit from Code 7.0.3  
(including decaying modes with  $\text{Im}(\omega) > 0$ )

**Example 7.0.7 (Analytic solution of homogeneous linear ordinary differential equations → [77, Remark 5.6.1], [34, Sect. 10.1], [59, Sect. 8.1], [15, Ex. 7.3])**

Autonomous homogeneous linear ordinary differential equation (ODE):

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} , \quad \mathbf{A} \in \mathbb{C}^{n,n} . \quad (7.0.8)$$

$$\mathbf{A} = \mathbf{S} \underbrace{\begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}}_{=: \mathbf{D}} \mathbf{S}^{-1}, \quad \mathbf{S} \in \mathbb{C}^{n,n} \text{ regular} \implies (\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} \xleftrightarrow{z=\mathbf{S}^{-1}\mathbf{y}} \dot{\mathbf{z}} = \mathbf{D}\mathbf{z}) .$$

➤ solution of initial value problem:

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} , \quad \mathbf{y}(0) = \mathbf{y}_0 \in \mathbb{C}^n \Rightarrow \mathbf{y}(t) = \mathbf{S}\mathbf{z}(t) , \quad \dot{\mathbf{z}} = \mathbf{D}\mathbf{z} , \quad \mathbf{z}(0) = \mathbf{S}^{-1}\mathbf{y}_0 .$$

The initial value problem for the *decoupled* homogeneous linear ODE  $\dot{\mathbf{z}} = \mathbf{D}\mathbf{z}$  has a simple analytic solution

$$\mathbf{z}_i(t) = \exp(\lambda_i t)(\mathbf{z}_0)_i = \exp(\lambda_i t) \left( (\mathbf{S}^{-1})_{i,:}^T \mathbf{y}_0 \right) .$$

In light of Rem. 1.3.3:

$$\mathbf{A} = \mathbf{S} \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} \mathbf{S}^{-1} \Leftrightarrow \mathbf{A}((\mathbf{S})_{:,i}) = \lambda_i ((\mathbf{S})_{:,i}) \quad i = 1, \dots, n . \quad (7.0.9)$$

In order to find the transformation matrix  $\mathbf{S}$  all non-zero solution vectors (= **eigenvectors**)  $\mathbf{x} \in \mathbb{C}^n$  of the **linear eigenvalue problem**

$$\mathbf{Ax} = \lambda \mathbf{x}$$

have to be found.

## Contents

7.1	Theory of eigenvalue problems	423
7.2	"Direct" Eigensolvers	425
7.3	Power Methods	429
7.3.1	Direct power method	429
7.3.2	Inverse Iteration [15, Sect. 7.6], [63, Sect. 5.3.2]	439
7.3.3	Preconditioned inverse iteration (PINVIT)	451
7.3.4	Subspace iterations	454
7.3.4.1	Orthogonalization	459
7.3.4.2	Ritz projection	464
7.4	Krylov Subspace Methods	469
7.5	Singular Value Decomposition	480

## 7.1 Theory of eigenvalue problems



*Supplementary reading.* [59, Ch. 7], [34, Ch. 9], [63, Sect. 1.7]

**Definition 7.1.1. Eigenvalues and eigenvectors** → [59, Sects. 7.1,7.2], [34, Sect. 9.1]

- $\lambda \in \mathbb{C}$  eigenvalue (ger.: Eigenwert) of  $\mathbf{A} \in \mathbb{K}^{n,n}$  : $\Leftrightarrow$   $\underbrace{\det(\lambda\mathbf{I} - \mathbf{A})}_\text{characteristic polynomial } \chi(\lambda) = 0$
- spectrum of  $\mathbf{A} \in \mathbb{K}^{n,n}$ :  $\sigma(\mathbf{A}) := \{\lambda \in \mathbb{C}: \lambda \text{ eigenvalue of } \mathbf{A}\}$
- eigenspace (ger.: Eigenraum) associated with eigenvalue  $\lambda \in \sigma(\mathbf{A})$ :  $\mathbf{Eig}\mathbf{A}\lambda := \text{Kern } \lambda\mathbf{I} - \mathbf{A}$
- $\mathbf{x} \in \mathbf{Eig}\mathbf{A}\lambda \setminus \{0\} \Rightarrow \mathbf{x}$  is eigenvector
- Geometric multiplicity (ger.: Vielfachheit) of an eigenvalue  $\lambda \in \sigma(\mathbf{A})$ :  $m(\lambda) := \dim \mathbf{Eig}\mathbf{A}\lambda$

Two simple facts:

$$\lambda \in \sigma(\mathbf{A}) \Rightarrow \dim \mathbf{Eig}\mathbf{A}\lambda > 0, \quad (7.1.2)$$

$$\det(\mathbf{A}) = \det(\mathbf{A}^T) \quad \forall \mathbf{A} \in \mathbb{K}^{n,n} \Rightarrow \sigma(\mathbf{A}) = \sigma(\mathbf{A}^T). \quad (7.1.3)$$

☞ notation:  $\rho(\mathbf{A}) := \max\{|\lambda|: \lambda \in \sigma(\mathbf{A})\} \hat{=} \text{spectral radius}$  of  $\mathbf{A} \in \mathbb{K}^{n,n}$

**Theorem 7.1.4. Bound for spectral radius**

For any matrix norm  $\|\cdot\|$  induced by a vector norm (→ Def. 1.5.71)

$$\rho(\mathbf{A}) \leq \|\mathbf{A}\|.$$

*Proof.* Let  $\mathbf{z} \in \mathbb{C}^n \setminus \{0\}$  be an eigenvector to the largest (in modulus) eigenvalue  $\lambda$  of  $\mathbf{A} \in \mathbb{C}^{n,n}$ . Then

$$\|\mathbf{A}\| := \sup_{\mathbf{x} \in \mathbb{C}^{n,n} \setminus \{0\}} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} \geq \frac{\|\mathbf{Az}\|}{\|\mathbf{z}\|} = |\lambda| = \rho(\mathbf{A}).$$

□

**Lemma 7.1.5. Gershgorin circle theorem** → [15, Thm. 7.13], [42, Thm. 32.1], [63, Sect. 5.1]

For any  $\mathbf{A} \in \mathbb{K}^{n,n}$  holds true

$$\sigma(\mathbf{A}) \subset \bigcup_{j=1}^n \left\{ z \in \mathbb{C} : |z - a_{jj}| \leq \sum_{i \neq j} |a_{ji}| \right\}.$$

**Lemma 7.1.6. Similarity and spectrum** → [34, Thm. 9.7], [15, Lemma 7.6], [59, Thm. 7.2]

The spectrum of a matrix is invariant with respect to *similarity transformations*:

$$\forall \mathbf{A} \in \mathbb{K}^{n,n}: \quad \sigma(\mathbf{S}^{-1}\mathbf{A}\mathbf{S}) = \sigma(\mathbf{A}) \quad \forall \text{ regular } \mathbf{S} \in \mathbb{K}^{n,n}.$$

**Lemma 7.1.7.**

*Existence of a one-dimensional invariant subspace*

$$\forall \mathbf{C} \in \mathbb{C}^{n,n}: \quad \exists \mathbf{u} \in \mathbb{C}^n: \quad \mathbf{C}(\text{Span}\{\mathbf{u}\}) \subset \text{Span}\{\mathbf{u}\}.$$

**Theorem 7.1.8. Schur normal form** → [37, Thm .2.8.1]

$$\forall \mathbf{A} \in \mathbb{K}^{n,n}: \quad \exists \mathbf{U} \in \mathbb{C}^{n,n} \text{ unitary: } \mathbf{U}^H \mathbf{A} \mathbf{U} = \mathbf{T} \quad \text{with } \mathbf{T} \in \mathbb{C}^{n,n} \text{ upper triangular.}$$

**Corollary 7.1.9. Principal axis transformation**

$$\mathbf{A} \in \mathbb{K}^{n,n}, \mathbf{A} \mathbf{A}^H = \mathbf{A}^H \mathbf{A}: \quad \exists \mathbf{U} \in \mathbb{C}^{n,n} \text{ unitary: } \mathbf{U}^H \mathbf{A} \mathbf{U} = \text{diag}(\lambda_1, \dots, \lambda_n), \quad \lambda_i \in \mathbb{C}.$$

A matrix  $\mathbf{A} \in \mathbb{K}^{n,n}$  with  $\mathbf{A} \mathbf{A}^H = \mathbf{A}^H \mathbf{A}$  is called *normal*.

- Examples of normal matrices are
- Hermitian matrices:  $\mathbf{A}^H = \mathbf{A}$   $\Rightarrow \sigma(\mathbf{A}) \subset \mathbb{R}$
  - unitary matrices:  $\mathbf{A}^H = \mathbf{A}^{-1}$   $\Rightarrow |\sigma(\mathbf{A})| = 1$
  - skew-Hermitian matrices:  $\mathbf{A} = -\mathbf{A}^H$   $\Rightarrow \sigma(\mathbf{A}) \subset i\mathbb{R}$

- Normal matrices can be diagonalized by *unitary* similarity transformations
- Symmetric real matrices can be diagonalized by *orthogonal* similarity transformations

- In Cor. 7.1.9:
- $\lambda_1, \dots, \lambda_n$  = eigenvalues of  $\mathbf{A}$
  - Columns of  $\mathbf{U}$  = orthonormal basis of eigenvectors of  $\mathbf{A}$

### Eigenvalue problems:

- (EVPs)
- ① Given  $\mathbf{A} \in \mathbb{K}^{n,n}$  find **all** eigenvalues (= spectrum of  $\mathbf{A}$ ).
  - ② Given  $\mathbf{A} \in \mathbb{K}^{n,n}$  find  $\sigma(\mathbf{A})$  plus **all** eigenvectors.
  - ③ Given  $\mathbf{A} \in \mathbb{K}^{n,n}$  find **a few** eigenvalues and associated eigenvectors

(Linear) generalized eigenvalue problem:

Given  $\mathbf{A} \in \mathbb{C}^{n,n}$ , regular  $\mathbf{B} \in \mathbb{C}^{n,n}$ , seek  $\mathbf{x} \neq 0, \lambda \in \mathbb{C}$

$$\mathbf{Ax} = \lambda \mathbf{Bx} \Leftrightarrow \mathbf{B}^{-1} \mathbf{Ax} = \lambda \mathbf{x}. \quad (7.1.10)$$

$\mathbf{x}$   $\hat{=}$  generalized eigenvector,  $\lambda$   $\hat{=}$  generalized eigenvalue

Obviously every generalized eigenvalue problem is equivalent to a standard eigenvalue problem

$$\mathbf{Ax} = \lambda \mathbf{Bx} \Leftrightarrow \mathbf{B}^{-1} \mathbf{Ax} = \lambda \mathbf{x}.$$

However, usually it is not advisable to use this equivalence for numerical purposes!

### Remark 7.1.11 (Generalized eigenvalue problems and Cholesky factorization)

If  $\mathbf{B} = \mathbf{B}^H$  s.p.d. ( $\rightarrow$  Def. 1.1.8) with Cholesky factorization  $\mathbf{B} = \mathbf{R}^H \mathbf{R}$

$$\mathbf{Ax} = \lambda \mathbf{Bx} \Leftrightarrow \tilde{\mathbf{A}}\mathbf{y} = \lambda \mathbf{y} \quad \text{where } \tilde{\mathbf{A}} := \mathbf{R}^{-H} \mathbf{A} \mathbf{R}^{-1}, \mathbf{y} := \mathbf{Rx}.$$

→ This transformation can be used for efficient computations.

## 7.2 “Direct” Eigensolvers

Purpose: solution of eigenvalue problems ①, ② for **dense** matrices “up to machine precision”

MATLAB-function:

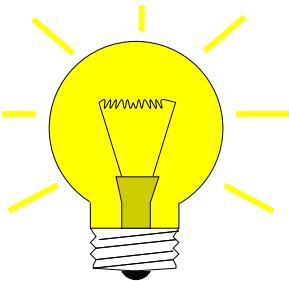
`eig`

$\mathbf{d} = \text{eig}(\mathbf{A})$  : computes spectrum  $\sigma(\mathbf{A}) = \{d_1, \dots, d_n\}$  of  $\mathbf{A} \in \mathbb{C}^{n,n}$   
 $[\mathbf{V}, \mathbf{D}] = \text{eig}(\mathbf{A})$  : computes  $\mathbf{V} \in \mathbb{C}^{n,n}$ , diagonal  $\mathbf{D} \in \mathbb{C}^{n,n}$  such that  $\mathbf{AV} = \mathbf{VD}$

**Remark 7.2.1 (QR-Algorithm)** → [28, Sect. 7.5], [59, Sect. 10.3], [42, Ch. 26], [63, Sect. 5.5-5.7])

Note:

All “direct” eigensolvers are iterative methods

Idea: Iteration based on successive **unitary** similarity transformations

$$\mathbf{A} = \mathbf{A}^{(0)} \xrightarrow{\text{green arrow}} \mathbf{A}^{(1)} \xrightarrow{\text{green arrow}} \dots \xrightarrow{\text{green arrow}} \left. \begin{array}{l} \text{diagonal matrix} \\ \text{upper triangular matrix} \\ (\rightarrow \text{Thm. 7.1.8}) \end{array} \right\}$$

(superior stability of unitary transformations, see ??)

**QR-algorithm (with shift)**

- \* in general: quadratic convergence
  - \* cubic convergence for normal matrices
- (→ [28, Sect. 7.5, 8.2])

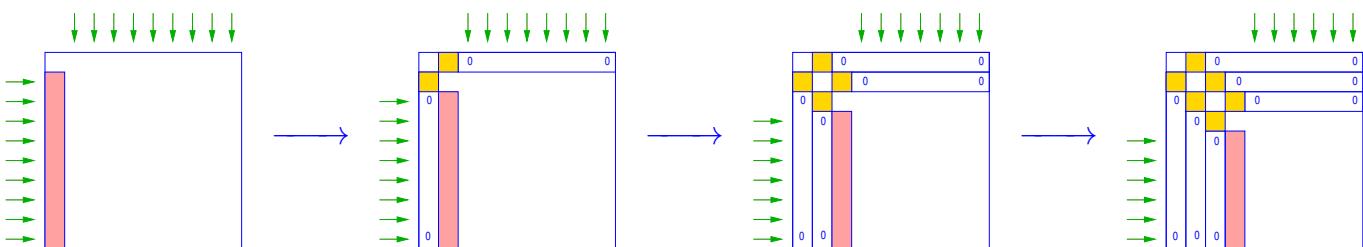
**MATLAB-code 7.2.2: QR-algorithm with shift**

```

1 function d = eigqr(A,tol)
2 n = size(A,1);
3 while (norm(tril(A,-1)) >
4 tol*norm(A))
5 % shift by eigenvalue of lower right 2x2
6 % block closest to (A)n,n
7 sc = eig(A(n-1:n,n-1:n));
8 [dummy,si] = min(abs(sc-A(n,n)));
9 shift = sc(si);
10 [Q,R] = qr(A - shift * eye(n));
11 A = Q'*A*Q;
12 end
13 d = diag(A);

```

Computational cost:  $O(n^3)$  operations per step of the QR-algorithm
 Library implementations of the QR-algorithm provide *numerically stable* eigensolvers (→ Def. 1.5.80)

**Remark 7.2.3 (Unitary similarity transformation to tridiagonal form)**
Successive Householder similarity transformations of  $\mathbf{A} = \mathbf{A}^H$ :(→  $\hat{\wedge}$  affected rows/columns,  $\hat{\square}$  targeted vector)

► transformation to tridiagonal form! (for general matrices a similar strategy can achieve a similarity transformation to upper Hessenberg form)

- this transformation is used as a preprocessing step for QR-algorithm ➤ `eig`.

Similar functionality for generalized EVP  $\mathbf{A}\mathbf{x} = \lambda \mathbf{B}\mathbf{x}$ ,  $\mathbf{A}, \mathbf{B} \in \mathbb{C}^{n,n}$

`d = eig(A, B)` : computes all generalized eigenvalues  
`[V, D] = eig(A, B)` : computes  $\mathbf{V} \in \mathbb{C}^{n,n}$ , diagonal  $\mathbf{D} \in \mathbb{C}^{n,n}$  such that  $\mathbf{AV} = \mathbf{BVD}$

Note: (Generalized) eigenvectors can be recovered as columns of  $\mathbf{V}$ :

$$\mathbf{AV} = \mathbf{VD} \Leftrightarrow \mathbf{A}(\mathbf{V})_{:,i} = (\mathbf{D})_{i,i} \mathbf{V}_{:,i},$$

if  $\mathbf{D} = \text{diag}(d_1, \dots, d_n)$ .

#### Remark 7.2.4 (Computational effort for eigenvalue computations)

Computational effort (#elementary operations) for `eig()`:

eigenvalues & eigenvectors of $\mathbf{A} \in \mathbb{K}^{n,n}$	$\sim 25n^3 + O(n^2)$	$O(n^3)$ !
only eigenvalues of $\mathbf{A} \in \mathbb{K}^{n,n}$	$\sim 10n^3 + O(n^2)$	
eigenvalues and eigenvectors $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$	$\sim 9n^3 + O(n^2)$	
only eigenvalues of $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$	$\sim \frac{4}{3}n^3 + O(n^2)$	
only eigenvalues of tridiagonal $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$	$\sim 30n^2 + O(n)$	

Note:

`eig` not available for sparse matrix arguments

Exception:

`d=eig(A)` for sparse Hermitian matrices

#### Example 7.2.5 (Runtimes of `eig`)

##### MATLAB-code 7.2.6:

```
1 A = rand(500,500); B = A'*A; C = gallery('tridiag',500,1,3,1);
```

- \*
- \* **A** generic dense matrix
  - \* **B** symmetric (s.p.d. → Def. 1.1.8) matrix
  - \* **C** s.p.d. *tridiagonal* matrix

##### MATLAB-code 7.2.7: measuring runtimes of `eig`

```
1 function eigtiming
2
3 A = rand(500,500); B = A'*A;
4 C = gallery('tridiag',500,1,3,1);
5 times = [];
6 for n=5:5:500
7     An = A(1:n,1:n); Bn = B(1:n,1:n); Cn = C(1:n,1:n);
8     t1 = 1000; for k=1:3, tic; d = eig(An); t1 = min(t1,toc); end
```

```

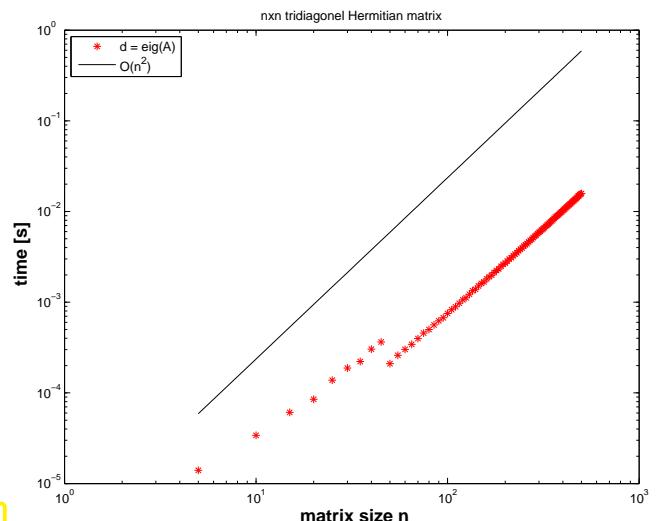
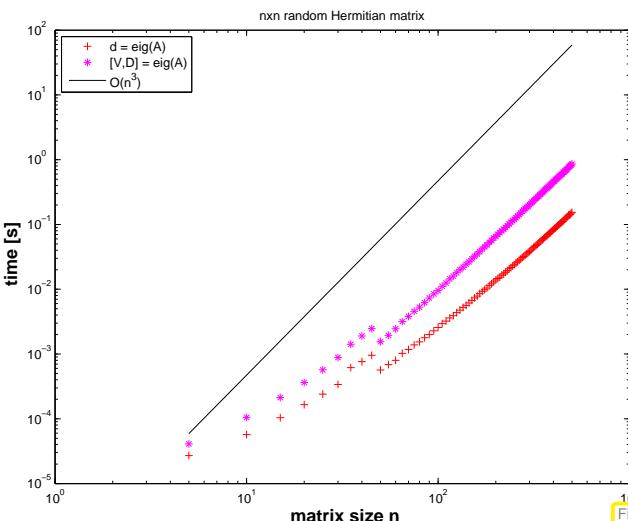
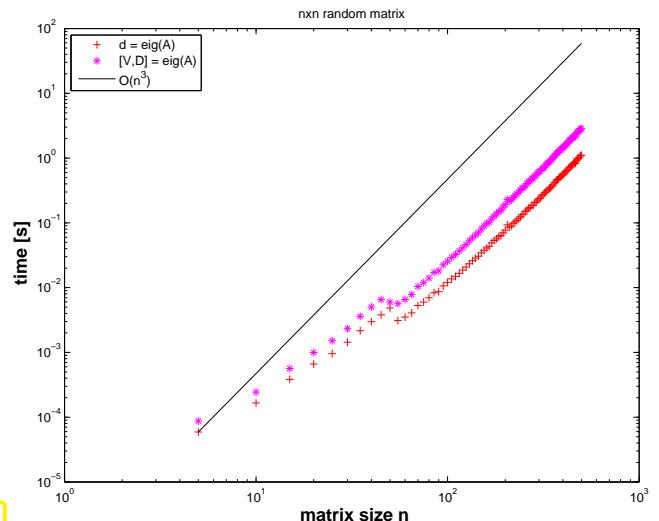
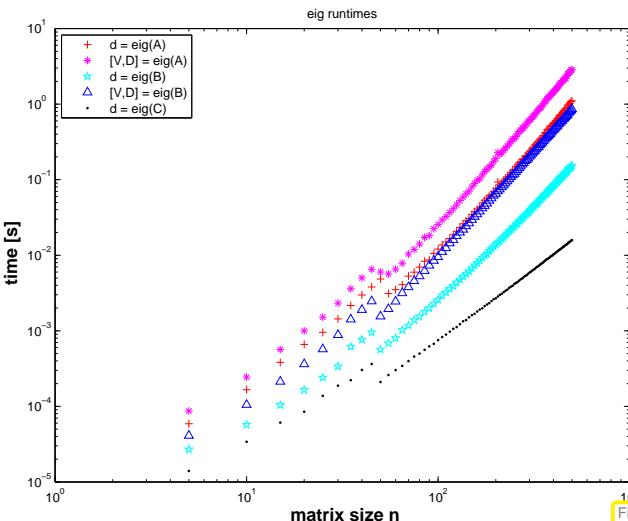
 9 | t2 = 1000; for k=1:3, tic; [V,D] = eig(An); t2 = min(t2,toc);
10 | end
11 | t3 = 1000; for k=1:3, tic; d = eig(Bn); t3 = min(t3,toc); end
12 | t4 = 1000; for k=1:3, tic; [V,D] = eig(Bn); t4 = min(t4,toc);
13 | end
14 | t5 = 1000; for k=1:3, tic; d = eig(Cn); t5 = min(t5,toc); end
15 | times = [times; n t1 t2 t3 t4 t5];
16 | end
17 |
18 | figure;
19 | loglog(times(:,1),times(:,2),'r+', times(:,1),times(:,3),'m*',...
20 |         times(:,1),times(:,4),'cp', times(:,1),times(:,5),'b^',...
21 |         times(:,1),times(:,6),'k.');
22 | xlabel('{\bf matrix size n}', 'fontsize', 14);
23 | ylabel('{\bf time [s]}', 'fontsize', 14);
24 | title('eig runtimes');
25 | legend('d = eig(A)', '[V,D] = eig(A)', 'd = eig(B)', '[V,D] =...
26 |         eig(B)', 'd = eig(C)', ...
27 |         'location', 'northwest');
28 |
29 | print -depsc2 '../PICTURES/eigtimingall.eps'
30 |
31 | figure;
32 | loglog(times(:,1),times(:,2),'r+', times(:,1),times(:,3),'m*',...
33 |         times(:,1),(times(:,1).^3)/(times(1,1)^3)*times(1,2),'k-');
34 | xlabel('{\bf matrix size n}', 'fontsize', 14);
35 | ylabel('{\bf time [s]}', 'fontsize', 14);
36 | title('nxn random matrix');
37 | legend('d = eig(A)', '[V,D] =...
38 |         eig(A)', 'O(n^3)', 'location', 'northwest');
39 |
40 | print -depsc2 '../PICTURES/eigtimingA.eps'
41 |
42 | figure;
43 | loglog(times(:,1),times(:,4),'r+', times(:,1),times(:,5),'m*',...
44 |         times(:,1),(times(:,1).^3)/(times(1,1)^3)*times(1,2),'k-');
45 | xlabel('{\bf matrix size n}', 'fontsize', 14);
46 | ylabel('{\bf time [s]}', 'fontsize', 14);
47 | title('nxn random Hermitian matrix');
48 | legend('d = eig(A)', '[V,D] =...
49 |         eig(A)', 'O(n^3)', 'location', 'northwest');
50 |
51 | print -depsc2 '../PICTURES/eigtimingB.eps'
52 |
53 | figure;
54 | loglog(times(:,1),times(:,6),'r*',...
55 |         times(:,1),(times(:,1).^2)/(times(1,1)^2)*times(1,2),'k-');
56 | xlabel('{\bf matrix size n}', 'fontsize', 14);
57 | ylabel('{\bf time [s]}', 'fontsize', 14);
58 | title('nxn tridiagonel Hermitian matrix');

```

```

54 legend('d = eig(A)', 'O(n^2)', 'location', 'northwest');
55
56 print -depsc2 '../PICTURES/eigtimingsC.eps'

```



- For the sake of efficiency: think which information you really need when computing eigenvalues/eigen-vectors of dense matrices

Potentially more efficient methods for *sparse matrices* will be introduced below in Section 7.3, 7.4.

## 7.3 Power Methods

### 7.3.1 Direct power method



*Supplementary reading.* [15, Sect. 7.5], [63, Sect. 5.3.1], [63, Sect. 5.3]

**Example 7.3.1 ((Simplified) Page rank algorithm** → [51, 27])

Model: **Random surfer** visits a web page, stays there for fixed time  $\Delta t$ , and then

- ❶ either follows each of  $\ell$  links on a page with probability  $1/\ell$ .
- ❷ or resumes surfing at a randomly (with equal probability) selected page

Option ❷ is chosen with probability  $d$ ,  $0 \leq d \leq 1$ , option ❶ with probability  $1 - d$ .

► Stationary **Markov chain**, state space  $\hat{=}$  set of all web pages

Question: Fraction of time spent by random surfer on  $i$ -th page (= **page rank**  $x_i \in [0, 1]$ )

This number  $\in [0, 1]$  can be used to gauge the “importance” of a web page, which, in turns, offers a way to sort the hits resulting from a keyword query: the GOOGLE idea.

Method: Stochastic simulation



### MATLAB-code 7.3.2: stochastic page rank simulation

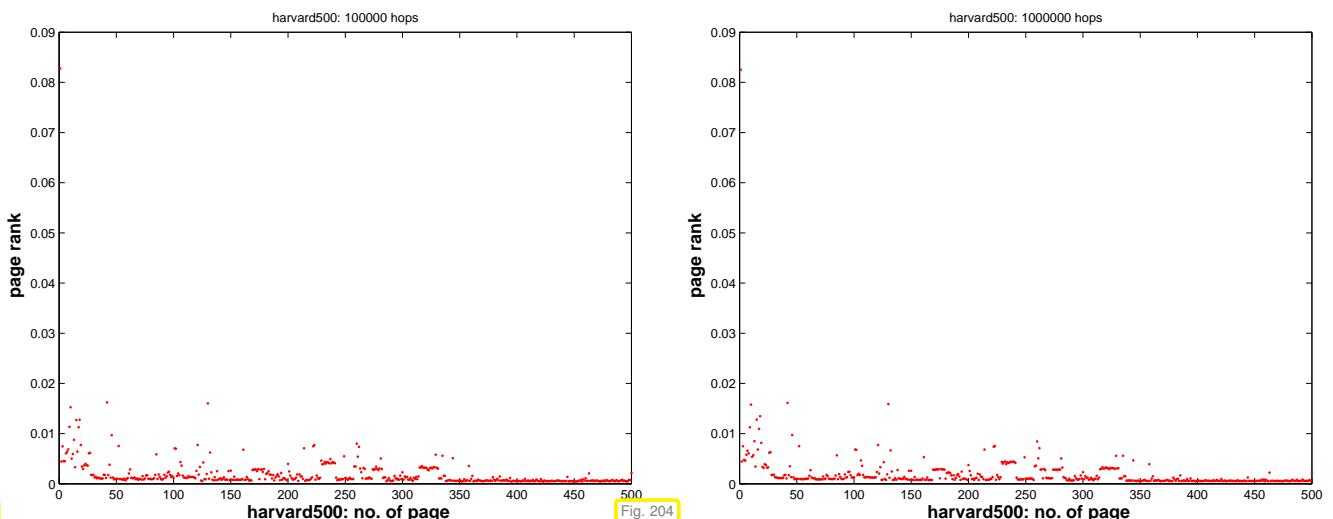
```

1 function prstochsim(Nhops)
2 % Load web graph data stored in  $N \times N$ -matrix G
3 load harvard500.mat;
4 N = size(G,1); d = 0.15;
5 count = zeros(1,N); cp = 1;
6 figure('position',[0 0 1200 1000]); pause;
7 for n=1:Nhops
8 % Find links from current page cp
9 idx = find(G(:,cp)); l = size(idx,1); rn = rand(); %
10 % If no links, jump to any other pages with equal probability
11 if (isempty(idx)), cp = floor(rn*N)+1;
12 % With probability d jump to any other page
13 elseif (rn < d), cp = floor(rn/d*N)+1;
14 % Follow outgoing links with equal probability
15 else cp = idx(floor((rn-d)/(1-d)*l)+1,1);
16 end
17 count(cp) = count(cp) + 1;
18 plot(1:N,count/n,'r.');
19 xlabel('{\bf harvard500: no. of page}', 'fontsize', 14);
20 ylabel('{\bf page rank}', 'fontsize', 14);
21 title(sprintf('{\bf page rank, harvard500: %d
22 hops}',n), 'fontsize', 14);
23 drawnow;
end
```

Explanations Code 7.3.2:

- \* Line 9: **rand** generates uniformly distributed pseudo-random numbers  $\in [0, 1]$
- \* Web graph encoded in  $\mathbf{G} \in \{0, 1\}^{N \times N}$ :

$$(\mathbf{G})_{ij} = 1 \Rightarrow \text{link } j \rightarrow i,$$



Observation: relative visit times stabilize as the number of hops in the stochastic simulation  $\rightarrow \infty$ .

The limit distribution is called **stationary distribution/invariant measure** of the Markov chain. This is what we seek.

- \* Numbering of pages  $1, \dots, N$ ,  $\ell_i \hat{=} \text{number of links from page } i$
- \*  $N \times N$ -matrix of transition probabilities page  $j \rightarrow \text{page } i$ :  $\mathbf{A} = (a_{ij})_{i,j=1}^N \in \mathbb{R}^{N,N}$

$a_{ij} \in [0, 1] \hat{=} \text{probability to jump from page } j \text{ to page } i$ .

$$\Rightarrow \sum_{i=1}^N a_{ij} = 1. \quad (7.3.3)$$

A matrix  $\mathbf{A} \in [0, 1]^{N,N}$  with the property (7.3.3) is called a (column) **stochastic matrix**.

“Meaning” of  $\mathbf{A}$ : given  $\mathbf{x} \in [0, 1]^N$ ,  $\|\mathbf{x}\|_1 = 1$ , where  $x_i$  is the probability of the surfer to visit page  $i$ ,  $i = 1, \dots, N$ , at an instance  $t$  in time,  $\mathbf{y} = \mathbf{Ax}$  satisfies

$$y_j \geq 0, \quad \sum_{j=1}^N y_j = \sum_{j=1}^N \sum_{i=1}^N a_{ji} x_i = \sum_{i=1}^N x_i \underbrace{\sum_{j=1}^N a_{ij}}_{=1} = \sum_{i=1}^N x_i = 1.$$

►  $y_j \hat{=} \text{probability for visiting page } j \text{ at time } t + \Delta t$ .

Transition probability matrix for page rank computation

$$(\mathbf{A})_{ij} = \begin{cases} \frac{1}{N} & , \text{ if } (\mathbf{G})_{ij} = 0 \forall i = 1, \dots, N, \\ d/N + (1-d) \frac{(\mathbf{G})_{ij}}{\ell_j} & \text{else.} \end{cases} \quad (7.3.4)$$

random jump to any other page      follow link

**MATLAB-code 7.3.5: transition probability matrix for page rank**

```

1 function A = prbuildA(G, d)
2 N = size(G, 1);

3 l = full(sum(G)); idx = find(l>0);
4 s = zeros(N, 1); s(idx) = 1./l(idx);
5 ds = ones(N, 1)/N; ds(idx) = d/N;

6 A = ones(N, 1)*ones(1, N)*diag(ds) +
    (1-d)*G*diag(s);

```

Note: special treatment of zero columns of  $\mathbf{G}$ , cf. (7.3.4)!



Stochastic simulation based on a single surfer is *slow*. Alternatives?

Thought experiment: Instead of a single random surfer we may consider  $m \in \mathbb{N}$ ,  $m \gg 1$ , of them who visit pages independently. The fraction of time  $m \cdot T$  they all together spend on page  $i$  will obviously be the same for  $T \rightarrow \infty$  as that for a single random surfer.

Instead of counting the surfers we watch the proportions of them visiting particular web pages at an instance of time. Thus, after the  $k$ -th hop we can assign a number  $x_i^{(k)} \in [0, 1]$  to web page  $i$ , which gives the proportion of surfers currently on that page:  $x_i^{(k)} := \frac{n_i^{(k)}}{m}$ , where  $n_i^{(k)} \in \mathbb{N}_0$  designates the number of surfers on page  $i$  after the  $k$ -th hop.

Now consider  $m \rightarrow \infty$ . The law of **large numbers** suggests that the (“infinitely many”) surfers visiting page  $j$  will move on to other pages proportional to the transition probabilities  $a_{ij}$ : in terms of proportions, for  $m \rightarrow \infty$  the stochastic evolution becomes a deterministic discrete dynamical system and we find

$$x_i^{(k+1)} = \sum_{j=1}^N a_{ij} x_j^{(k)}, \quad (7.3.6)$$

that is, the proportion of surfers ending up on page  $i$  equals the sum of the proportions on the “source pages” weighted with the transition probabilities.

Notice that (7.3.6) amounts to matrix  $\times$  vector. Thus, writing  $\mathbf{x}^{(0)} \in [0, 1]^N$ ,  $\|\mathbf{x}^{(0)}\| = 1$  for the initial distribution of the surfers on the net we find

$$\mathbf{x}^{(k)} = \mathbf{A}^k \mathbf{x}^{(0)}$$

will be their mass distribution after  $k$  hops. If the limit exists, the  $i$ -th component of  $\mathbf{x}^* := \lim_{k \rightarrow \infty} \mathbf{x}^{(k)}$  tells us which fraction of the (infinitely many) surfers will be visiting page  $i$  most of the time. Thus,  $\mathbf{x}^*$  yields the stationary distribution of the Markov chain.

**MATLAB-code 7.3.7: tracking fractions of many surfers**

```

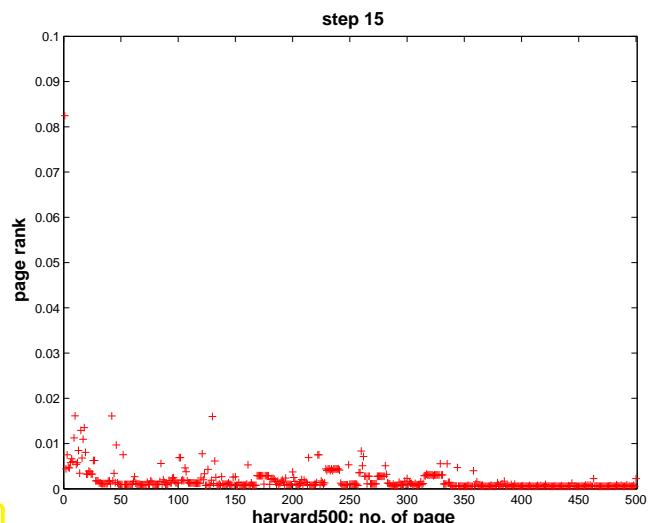
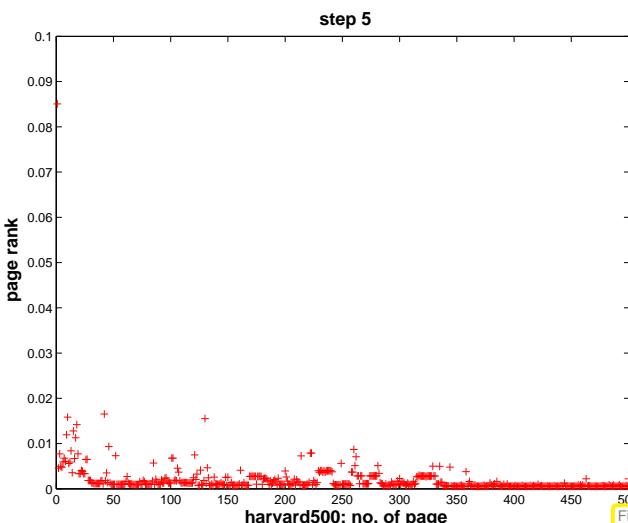
1 function prpowitsim(d, Nsteps)
2 % MATLAB way of specifying Default arguments
3 if (nargin < 2), Nsteps = 5; end
4 if (nargin < 1), d = 0.15; end

```

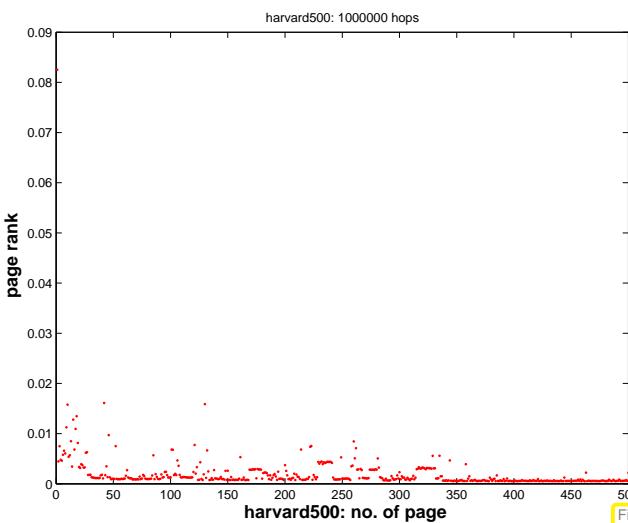
```

5 % load connectivity matrix and build transition matrix
6 load harvard500.mat; A = prbuildA(G,d);
7 N = size(A,1); x = ones(N,1)/N;
8
9 figure('position',[0 0 1200 1000]);
10 plot(1:N,x,'r+'); axis([0 N+1 0 0.1]);
11 % Plain power iteration for stochastic matrix A
12 for l=1:Nsteps
13 pause; x = A*x; plot(1:N,x,'r+'); axis([0 N+1 0 0.1]);
14 title(sprintf('{\bf step %d}',l),'fontsize',14);
15 xlabel('{\bf harvard500: no. of page}','fontsize',14);
16 ylabel('{\bf page rank}','fontsize',14); drawnow;
17 end

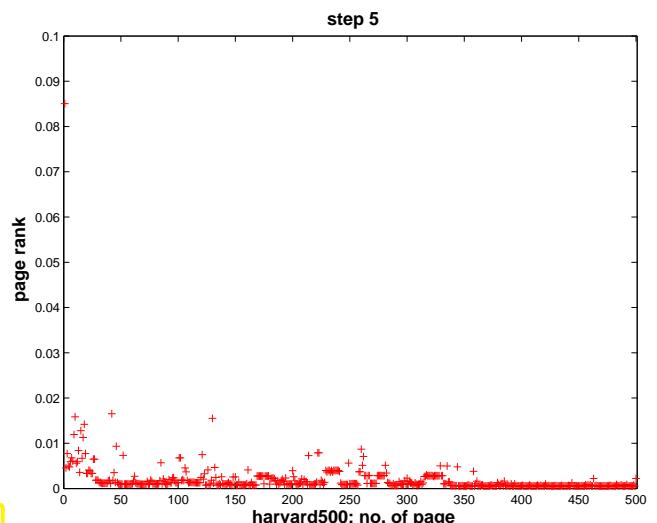
```



Comparison:



Single surfer stochastic simulation



Power method, Code 7.3.7

Observation: Convergence of the  $\mathbf{x}^{(k)} \rightarrow \mathbf{x}^*$ , and the limit must be a fixed point of the iteration function:

$$\Rightarrow \mathbf{Ax}^* = \mathbf{x}^* \Rightarrow \mathbf{x}^* \in \text{EigA1}.$$

Does  $\mathbf{A}$  possess an eigenvalue  $= 1$ ? Does the associated eigenvector really provide a probability distribution (after scaling), that is, are all of its entries non-negative? Is this probability distribution unique? To answer these questions we have to study the matrix  $\mathbf{A}$ :

For every stochastic matrix  $\mathbf{A}$ , by definition (7.3.3)

$$\begin{aligned} \mathbf{A}^T \mathbf{1} &= \mathbf{1} \stackrel{(7.1.3)}{\Rightarrow} \mathbf{1} \in \sigma(\mathbf{A}), \\ (1.5.75) \Rightarrow \|\mathbf{A}\|_1 &= 1 \stackrel{\text{Thm. 7.1.4}}{\Rightarrow} \rho(\mathbf{A}) = 1, \end{aligned}$$

where  $\rho(\mathbf{A})$  is the spectral radius of the matrix  $\mathbf{A}$ , see Section 7.1.

For  $\mathbf{r} \in \text{EigA1}$ , that is,  $\mathbf{Ar} = \mathbf{r}$ , denote by  $|\mathbf{r}|$  the vector  $(|r_i|)_{i=1}^N$ . Since all entries of  $\mathbf{A}$  are non-negative, we conclude by the triangle inequality that  $\|\mathbf{Ar}\|_1 \leq \|\mathbf{A}|\mathbf{r}|\|_1$

$$\begin{aligned} \Rightarrow 1 &= \|\mathbf{A}\|_1 = \sup_{\mathbf{x} \in \mathbb{R}^N} \frac{\|\mathbf{Ax}\|_1}{\|\mathbf{x}\|_1} \geq \frac{\|\mathbf{A}|\mathbf{r}|\|_1}{\|\mathbf{r}\|_1} \geq \frac{\|\mathbf{Ar}\|_1}{\|\mathbf{r}\|_1} = 1. \\ \Rightarrow \|\mathbf{A}|\mathbf{r}|\|_1 &= \|\mathbf{Ar}\|_1 \stackrel{\text{if } a_{ij} > 0}{\Rightarrow} |\mathbf{r}| = \pm \mathbf{r}. \end{aligned}$$

Hence, different components of  $\mathbf{r}$  cannot have opposite sign, which means, that  $\mathbf{r}$  can be chosen to have non-negative entries, if the entries of  $\mathbf{A}$  are strictly positive, which is the case for  $\mathbf{A}$  from (7.3.4). After normalization  $\|\mathbf{r}\|_1 = 1$  the eigenvector can be regarded as a probability distribution on  $\{1, \dots, N\}$ .

If  $\mathbf{Ar} = \mathbf{r}$  and  $\mathbf{As} = \mathbf{s}$  with  $(\mathbf{r})_i \geq 0$ ,  $(\mathbf{s})_i \geq 0$ ,  $\|\mathbf{r}\|_1 = \|\mathbf{s}\|_1 = 1$ , then  $\mathbf{A}(\mathbf{r} - \mathbf{s}) = \mathbf{r} - \mathbf{s}$ . Hence, by the above considerations, also all the entries of  $\mathbf{r} - \mathbf{s}$  are either non-negative or non-positive. By the assumptions on  $\mathbf{r}$  and  $\mathbf{s}$  this is only possible, if  $\mathbf{r} - \mathbf{s} = \mathbf{0}$ . We conclude that

$$\mathbf{A} \in ]0, 1]^{N,N} \text{ stochastic} \Rightarrow \dim \text{EigA1} = 1. \quad (7.3.8)$$

Sorting the pages according to the size of the corresponding entries in  $\mathbf{r}$  yields the famous “page rank”.

#### MATLAB-code 7.3.9: computing page rank vector $\mathbf{r}$ via `eig`

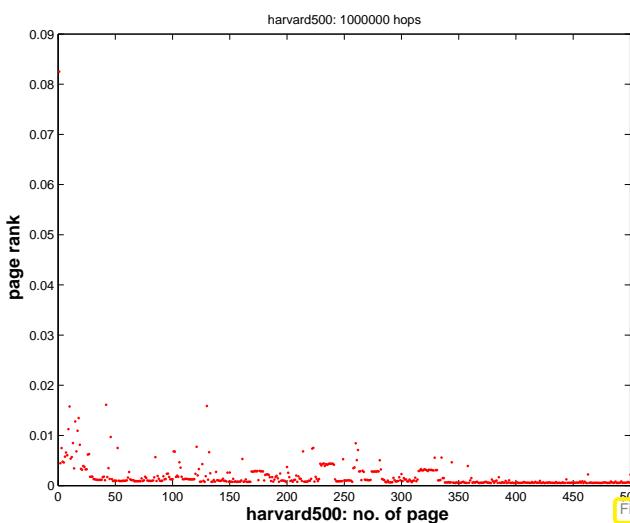
```
function prevp
load harvard500.mat; d = 0.15;
[V,D] = eig(prbuildA(G,d));

figure; r = V(:,1); N = length(r);
plot(1:N,r/sum(r),'m.');
axis([0 N+1 0 0.1]);
xlabel('{\bf harvard500: no. of
page}', 'fontsize', 14);
ylabel('{\bf entry of r-vector}', 'fontsize', 14);
title('harvard 500: Perron-Frobenius vector');
print -depsc2 '../PICTURES/prevp.eps';
```

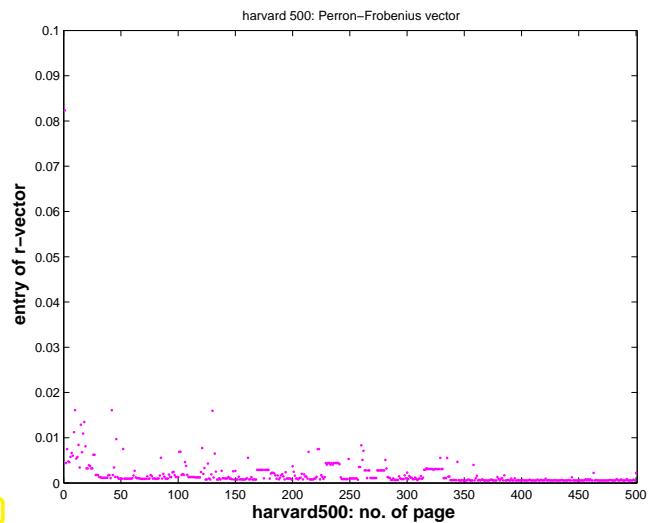
Plot of entries of unique vector  $\mathbf{r} \in \mathbb{R}^N$  with

$$\begin{aligned} 0 \leq (\mathbf{r})_i &\leq 1, \\ \|\mathbf{r}\|_1 &= 1, \\ \boxed{\mathbf{Ar} = \mathbf{r}}. \end{aligned}$$

Inefficient implementation!



stochastic simulation



eigenvector computation

The possibility to compute the stationary probability distribution of a Markov chain through an eigenvector of the transition probability matrix is due to a property of stationary Markov chains called **ergodicity**.

$\mathbf{A} \triangleq$  page rank transition probability matrix, see Code 7.3.5,  $d = 0.15$ , harvard500 example.

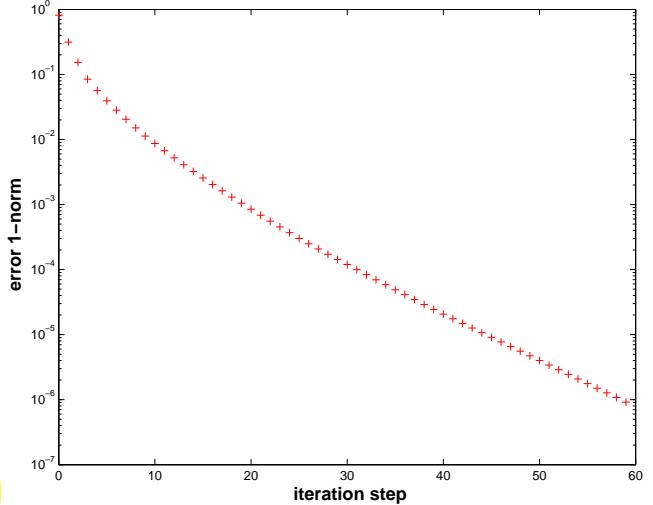
Errors:



$$\left\| \mathbf{A}^k \mathbf{x}_0 - \mathbf{r} \right\|_1,$$

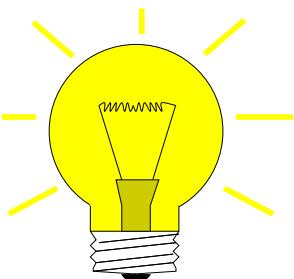
with  $\mathbf{x}_0 = \mathbf{1}/N$ ,  $N = 500$ .

We observe **linear convergence!** ( $\rightarrow$  Def. 2.1.9, iteration error vs. iteration count  $\approx$  straight line lin-log plot)



The computation of page rank amounts to finding the eigenvector of the matrix  $\mathbf{A}$  of transition probabilities that belongs to its *largest* eigenvalue 1. This is addressed by an important class of practical eigenvalue problems:

Task:	given $\mathbf{A} \in \mathbb{K}^{n,n}$ , find <b>largest</b> (in modulus) eigenvalue of $\mathbf{A}$ and (an) associated eigenvector.
-------	---



Idea: (suggested by page rank computation, Code 7.3.7)

Iteration:  $\mathbf{z}^{(k+1)} = \mathbf{A}\mathbf{z}^{(k)}$ ,  $\mathbf{z}^{(0)}$  arbitrary

<b>Example 7.3.10 (Power iteration</b> → Ex. 7.3.1)
---

Try the above iteration for general  $10 \times 10$ -matrix, largest eigenvalue 10, algebraic multiplicity 1.

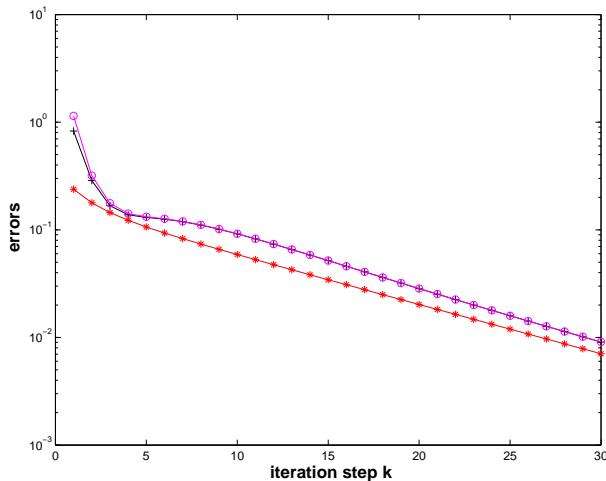


Fig. 212

**MATLAB-code 7.3.11:**

```
d = (1:10)'; n = length(d);
S = triu(diag(n:-1:1, 0) + ...
ones(n, n));
A = S * diag(d, 0) * inv(S);
```

$$\triangleq \text{error norm } \left\| \frac{\mathbf{z}^{(k)}}{\|\mathbf{z}^{(k)}\|} - (\mathbf{S})_{:,10} \right\|$$

(Note:  $(\mathbf{S})_{:,10} \hat{=} \text{eigenvector for eigenvalue 10}$ )

$\mathbf{z}^{(0)}$  = random vector

Observation: linear convergence of (normalized) eigenvectors!

► Suggests direct power method (ger.: Potenzmethode): iterative method ( $\rightarrow$  Section 2.1)

$$\begin{aligned} \text{initial guess: } & \mathbf{z}^{(0)} \text{ "arbitrary"}, \\ \text{next iterate: } & \mathbf{w} := \mathbf{A}\mathbf{z}^{(k-1)}, \quad \mathbf{z}^{(k)} := \frac{\mathbf{w}}{\|\mathbf{w}\|_2}, \quad k = 1, 2, \dots \end{aligned} \quad (7.3.12)$$

Note: the “normalization” of the iterates in (7.3.12) does not change anything (in exact arithmetic) and helps avoid overflow in floating point arithmetic.

Computational effort:  $1 \times \text{matrix} \times \text{vector per step} \Rightarrow$  inexpensive for sparse matrices

A persuasive theoretical justification for the direct power method:

Assume  $\mathbf{A} \in \mathbb{K}^{n,n}$  to be diagonalizable:

$$\Leftrightarrow \exists \text{ basis } \{\mathbf{v}_1, \dots, \mathbf{v}_n\} \text{ of eigenvectors of } \mathbf{A}: \mathbf{A}\mathbf{v}_j = \lambda_j \mathbf{v}_j, \lambda_j \in \mathbb{C}.$$

Assume

$$|\lambda_1| \leq |\lambda_2| \leq \dots \leq |\lambda_{n-1}| < |\lambda_n|, \quad \|\mathbf{v}_j\|_2 = 1. \quad (7.3.13)$$

Key observations for power iteration (7.3.12)

$$\mathbf{z}^{(k)} = \mathbf{A}^k \mathbf{z}^{(0)} \quad (\rightarrow \text{name "power method"}) \quad (7.3.14)$$

$$\mathbf{z}^{(0)} = \sum_{j=1}^n \zeta_j \mathbf{v}_j \Rightarrow \mathbf{z}^{(k)} = \sum_{j=1}^n \zeta_j \lambda_j^k \mathbf{v}_j. \quad (7.3.15)$$

Due to (7.3.13) for large  $k \gg 1$  ( $\Rightarrow |\lambda_n^k| \gg |\lambda_j^k|$  for  $j \neq n$ ) the contribution of  $\mathbf{v}_n$  (size  $\zeta_n \lambda_n^k$ ) in the eigenvector expansion (7.3.15) will be much larger than the contribution (size  $\zeta_n \lambda_j^k$ ) of any other eigenvector (, if  $\zeta_n \neq 0$ ): the eigenvector for  $\lambda_n$  will swamp all other for  $k \rightarrow \infty$ .

Further (7.3.15) nutures expectation:  $\mathbf{v}_n$  will become dominant in  $\mathbf{z}^{(k)}$  the faster, the better  $|\lambda_n|$  is separated from  $|\lambda_{n-1}|$ , see Thm. 7.3.21 for rigorous statement.

$\mathbf{z}^{(k)}$  → eigenvector, but how do we get the associated eigenvalue  $\lambda_n$  ?

When (7.3.12) has converged, two common ways to recover  $\lambda_{\max}$  → [15, Alg. 7.20]

$$\textcircled{1} \quad \mathbf{A}\mathbf{z}^{(k)} \approx \lambda_{\max}\mathbf{z}^{(k)} \Rightarrow |\lambda_n| \approx \frac{\|\mathbf{A}\mathbf{z}^{(k)}\|}{\|\mathbf{z}^{(k)}\|} \quad (\text{modulus only!})$$

$$\textcircled{2} \quad \lambda_{\max} \approx \underset{\theta \in \mathbb{R}}{\operatorname{argmin}} \left\| \mathbf{A}\mathbf{z}^{(k)} - \theta \mathbf{z}^{(k)} \right\|_2^2 \Rightarrow \lambda_{\max} \approx \frac{(\mathbf{z}^{(k)})^H \mathbf{A} \mathbf{z}^{(k)}}{\|\mathbf{z}^{(k)}\|_2^2}.$$

This latter formula is extremely useful, which has earned it a special name:

### Definition 7.3.16.

For  $\mathbf{A} \in \mathbb{K}^{n,n}$ ,  $\mathbf{u} \in \mathbb{K}^n$  the **Rayleigh quotient** is defined by

$$\rho_{\mathbf{A}}(\mathbf{u}) := \frac{\mathbf{u}^H \mathbf{A} \mathbf{u}}{\mathbf{u}^H \mathbf{u}}.$$

An immediate consequence of the definitions:

$$\lambda \in \sigma(\mathbf{A}) , \mathbf{z} \in \operatorname{Eig}_{\lambda} \mathbf{A} \Rightarrow \rho_{\mathbf{A}}(\mathbf{z}) = \lambda. \quad (7.3.17)$$

### Example 7.3.18 (Direct power method → Ex. 7.3.18 cnt'd)

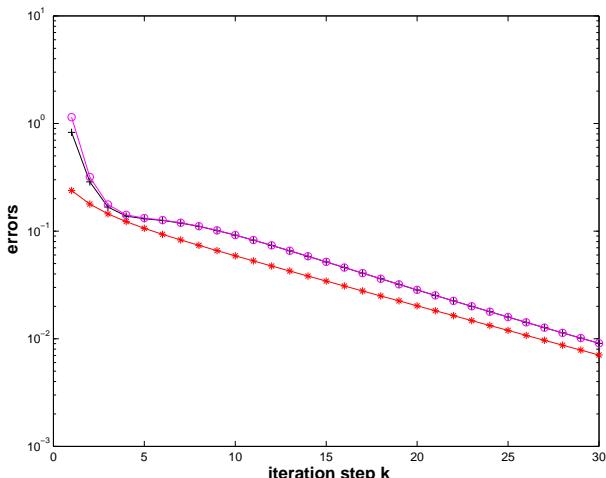


Fig. 213

#### MATLAB-code 7.3.19:

```
n = length(d); S = triu(diag(n:-1:1,0)+...  
ones(n,n)); A = S*diag(d,0)*inv(S);
```

$\textcircled{d} = (1:10)'$ ;

$\textcircled{o}$  : error  $|\lambda_n - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})|$

$\textcircled{*}$  : error norm  $\|\mathbf{z}^{(k)} - \mathbf{s}_{.,n}\|$

$\textcircled{+}$  :  $\left| \lambda_n - \frac{\|\mathbf{A}\mathbf{z}^{(k-1)}\|_2}{\|\mathbf{z}^{(k-1)}\|_2} \right|$

$\mathbf{z}^{(0)}$  = random vector

Test matrices:

- ①  $d = (1:10)'$ ;  $\Rightarrow |\lambda_{n-1}| : |\lambda_n| = 0.9$
- ②  $d = [\text{ones}(9, 1); 2]'$ ;  $\Rightarrow |\lambda_{n-1}| : |\lambda_n| = 0.5$
- ③  $d = 1-2.^{-(-1:0.5:5)'}'$ ;  $\Rightarrow |\lambda_{n-1}| : |\lambda_n| = 0.9866$

#### MATLAB-code 7.3.20: Investigating convergence of direct power method

```
1 % Demonstration of direct power method for Ex. 7.3.18  
2 maxit = 30; d = [1:10]'; n = length(d);  
3 % Initialize the matrix A
```

```

4 S = triu(diag(n:-1:1,0)+ones(n,n));
5 A = S*diag(d,0)*inv(S);
6 % This calculates the exact eigenvalues (for error calculation)
7 [V,D] = eig(A);

8
9 k = find(d == max(abs(d)));
10 if (length(k) > 1), error('No single largest EV'); end;
11 ev = X(:,k(1)); ev = ev/norm(ev); ev
12 ew = d(k(1)); ew
13 if (ew < 0), sgn = -1; else sgn = 1; end
14
15 z = rand(n,1); z = z/norm(z);
16 s = 1;
17 res = [];
18
19 % Actual direct power iteration
20 for i=1:maxit
21 w = A*z; l = norm(w); rq = real(dot(w,z)); z = w/l;
22 res = [res;i,l,norm(s*z-ev),abs(l-abs(ew)),abs(sgn*rq-ew)];
23 s = s * sgn;
24 end
25
26 % Plot the result
27 semilogy(res(:,1),res(:,3),'r-*',res(:,1),res(:,4),'k-+',res(:,1),res(:,5),'m-o')
28 xlabel({'\bf iteration step k'},'FontSize',14);
29 ylabel({'\bf errors}','FontSize',14);
30 print -deps2c '../PICTURES/pml.eps';

```

$$\rho_{\text{EV}}^{(k)} := \frac{\|\mathbf{z}^{(k)} - \mathbf{s}_{\cdot,n}\|}{\|\mathbf{z}^{(k-1)} - \mathbf{s}_{\cdot,n}\|},$$

$$\rho_{\text{EW}}^{(k)} := \frac{|\rho_{\mathbf{A}}(\mathbf{z}^{(k)}) - \lambda_n|}{|\rho_{\mathbf{A}}(\mathbf{z}^{(k-1)}) - \lambda_n|}.$$

$k$	①		②		③	
	$\rho_{\text{EV}}^{(k)}$	$\rho_{\text{EW}}^{(k)}$	$\rho_{\text{EV}}^{(k)}$	$\rho_{\text{EW}}^{(k)}$	$\rho_{\text{EV}}^{(k)}$	$\rho_{\text{EW}}^{(k)}$
22	0.9102	0.9007	0.5000	0.5000	0.9900	0.9781
23	0.9092	0.9004	0.5000	0.5000	0.9900	0.9791
24	0.9083	0.9001	0.5000	0.5000	0.9901	0.9800
25	0.9075	0.9000	0.5000	0.5000	0.9901	0.9809
26	0.9068	0.8998	0.5000	0.5000	0.9901	0.9817
27	0.9061	0.8997	0.5000	0.5000	0.9901	0.9825
28	0.9055	0.8997	0.5000	0.5000	0.9901	0.9832
29	0.9049	0.8996	0.5000	0.5000	0.9901	0.9839
30	0.9045	0.8996	0.5000	0.5000	0.9901	0.9844

Observation:

linear convergence ( $\rightarrow ??$ )

**Theorem 7.3.21. Convergence of direct power method** → [15, Thm. 25.1]

Let  $\lambda_n > 0$  be the largest (in modulus) eigenvalue of  $\mathbf{A} \in \mathbb{K}^{n,n}$  and have (algebraic) multiplicity 1. Let  $\mathbf{v}, \mathbf{y}$  be the left and right eigenvectors of  $\mathbf{A}$  for  $\lambda_n$  normalized according to  $\|\mathbf{y}\|_2 = \|\mathbf{v}\|_2 = 1$ . Then there is convergence

$$\left\| \mathbf{A} \mathbf{z}^{(k)} \right\|_2 \rightarrow \lambda_n \quad , \quad \mathbf{z}^{(k)} \rightarrow \pm \mathbf{v} \quad \text{linearly with rate} \quad \frac{|\lambda_{n-1}|}{|\lambda_n|} ,$$

where  $\mathbf{z}^{(k)}$  are the iterates of the direct power iteration and  $\mathbf{y}^H \mathbf{z}^{(0)} \neq 0$  is assumed.

**Remark 7.3.22 (Initial guess for power iteration)**

roundoff errors ➤  $\mathbf{y}^H \mathbf{z}^{(0)} \neq 0$  always satisfied in practical computations

Usual (not the best!) choice for  $\mathbf{x}^{(0)}$  = random vector

**Remark 7.3.23 (Termination criterion for direct power iteration)**

(→ Section 2.1.2)

Adaptation of a posteriori termination criterion (2.2.23)

“relative change”  $\leq \text{tol}$ :

$$\begin{cases} \min \left\| \mathbf{z}^{(k)} \pm \mathbf{z}^{(k-1)} \right\| \leq (1/L - 1)\text{tol} , \\ \left| \frac{\|\mathbf{A} \mathbf{z}^{(k)}\|}{\|\mathbf{z}^{(k)}\|} - \frac{\|\mathbf{A} \mathbf{z}^{(k-1)}\|}{\|\mathbf{z}^{(k-1)}\|} \right| \leq (1/L - 1)\text{tol} \end{cases} \quad \text{see (2.1.29).}$$

Estimated rate of convergence

**7.3.2 Inverse Iteration [15, Sect. 7.6], [63, Sect. 5.3.2]****Example 7.3.24 (Image segmentation)**

Given: gray-scale image: intensity matrix  $\mathbf{P} \in \{0, \dots, 255\}^{m,n}$ ,  $m, n \in \mathbb{N}$   
 $((\mathbf{P})_{ij} \leftrightarrow \text{pixel}, 0 \hat{=} \text{black}, 255 \hat{=} \text{white})$

Loading and displaying images  
in MATLAB ➤

**MATLAB-code 7.3.25: loading and displaying an image**

```

1 M = imread('eth.pbm');
2 [m, n] = size(M);
3 fprintf('%dx%d grey scale pixel
        image\n', m, n);
4 figure; image(M); title('ETH view');
5 col = [0:1/215:1]*[1, 1, 1]; colormap(col);

```

(Fuzzy) task: Local segmentation

Find connected patches of image of the same shade/color

More general segmentation problem (non-local): identify parts of the image, not necessarily connected, with the same texture.

Next: Statement of (rigorously defined) problem, cf. ??:

Preparation: Numbering of pixels  $1 \dots, mn$ , e.g, lexicographic numbering:

- \* pixel set  $\mathcal{V} := \{1 \dots, nm\}$
- \* indexing: index(pixel <sub>$i,j$</sub> ) =  $(i-1)n + j$

notation:  $p_k := (\mathbf{P})_{ij}$ , if  $k = \text{index}(\text{pixel}_{i,j}) = (i-1)n + j$ ,  $k = 1, \dots, N := mn$

Local similarity matrix:

$$\mathbf{W} \in \mathbb{R}^{N,N}, \quad N := mn, \quad (7.3.26)$$

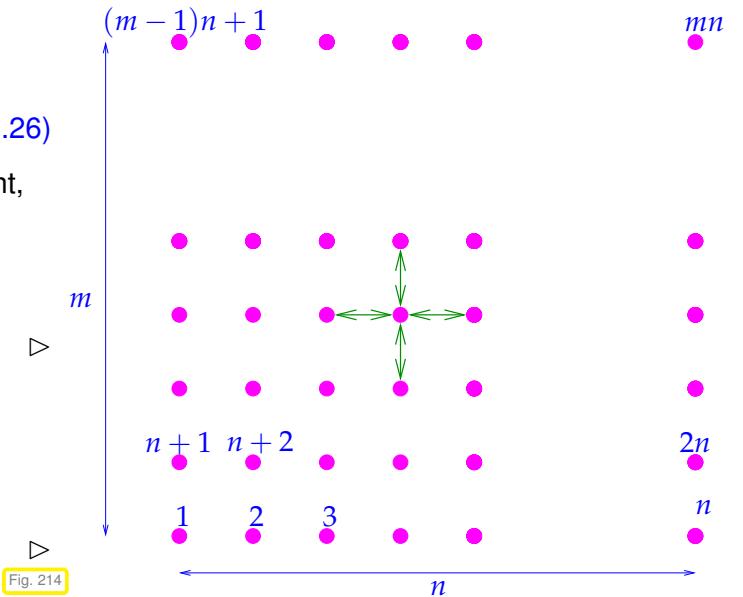
$$(\mathbf{W})_{ij} = \begin{cases} 0 & , \text{if pixels } i, j \text{ not adjacent,} \\ 0 & , \text{if } i = j, \\ \sigma(p_i, p_j) & , \text{if pixels } i, j \text{ adjacent.} \end{cases}$$

$\Leftrightarrow \hat{\triangleq}$  adjacent pixels

Similarity function, e.g., with  $\alpha > 0$

$$\sigma(x, y) := \exp(-\alpha(x - y)^2), \quad x, y \in \mathbb{R}.$$

Lexicographic numbering



The entries of the matrix  $\mathbf{W}$  measure the “similarity” of neighboring pixels: if  $(\mathbf{W})_{ij}$  is large, they encode (almost) the same intensity, if  $(\mathbf{W})_{ij}$  is close to zero, then they belong to parts of the picture with very different brightness. In the latter case, the boundary of the segment may separate the two pixels.

### Definition 7.3.27. Normalized cut ( $\rightarrow$ [73, Sect. 2])

For  $\mathcal{X} \subset \mathcal{V}$  define the normalized cut as

$$\text{Ncut}(\mathcal{X}) := \frac{\text{cut}(\mathcal{X})}{\text{weight}(\mathcal{X})} + \frac{\text{cut}(\mathcal{X})}{\text{weight}(\mathcal{V} \setminus \mathcal{X})},$$

with  $\text{cut}(\mathcal{X}) := \sum_{i \in \mathcal{X}, j \notin \mathcal{X}} w_{ij}$ ,  $\text{weight}(\mathcal{X}) := \sum_{i \in \mathcal{X}, j \in \mathcal{X}} w_{ij}$ .

In light of local similarity relationship:

- $\text{cut}(\mathcal{X})$  big  $\Rightarrow$  substantial similarity of pixels across interface between  $\mathcal{X}$  and  $\mathcal{V} \setminus \mathcal{X}$ .
- $\text{weight}(\mathcal{X})$  big  $\Rightarrow$  a lot of similarity of adjacent pixels inside  $\mathcal{X}$ .

► Segmentation problem (rigorous statement):

$$\text{find } \mathcal{X}^* \subset \mathcal{V}: \quad \mathcal{X}^* = \operatorname{argmin}_{\mathcal{X} \subset \mathcal{V}} \text{Ncut}(\mathcal{X}) \quad (7.3.28)$$



NP-hard combinatorial optimization problem !

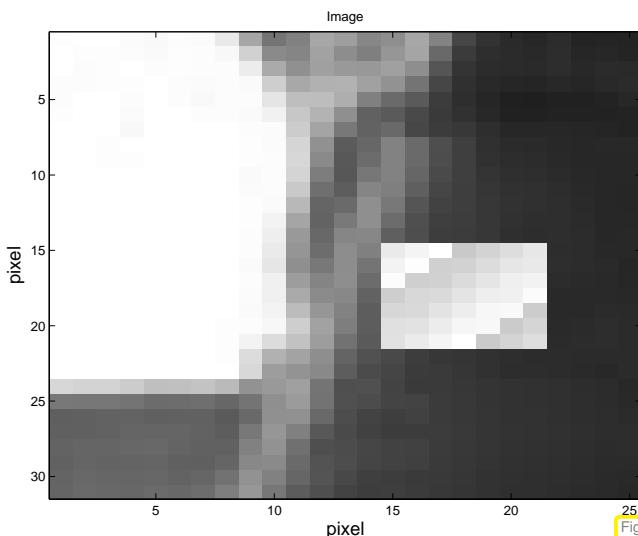


Fig. 215

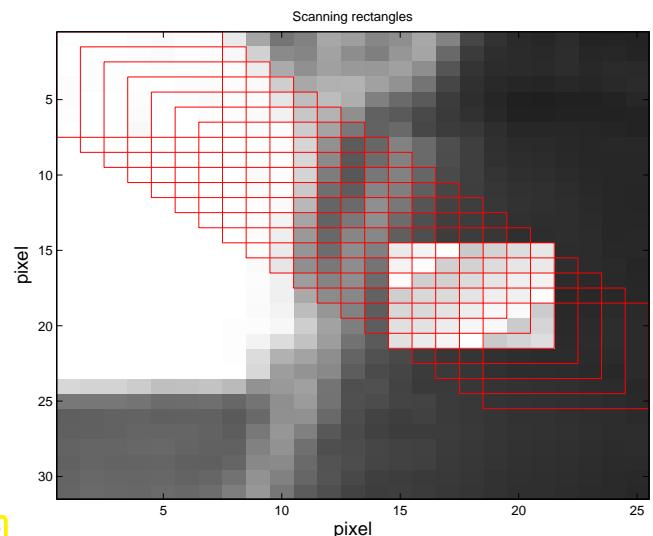


Fig. 216

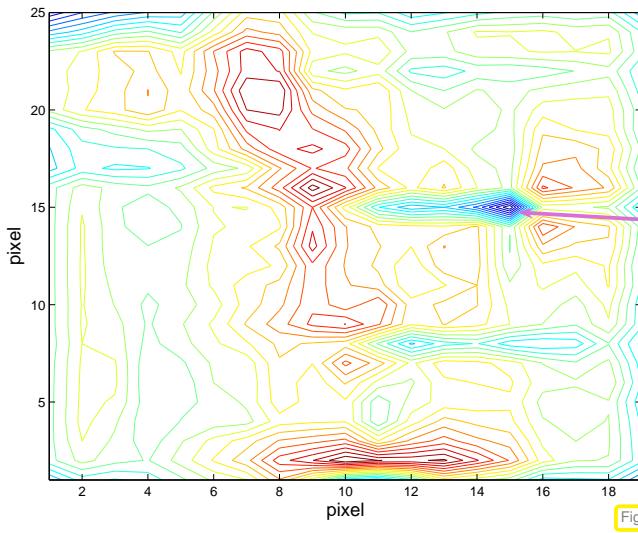


Fig. 217

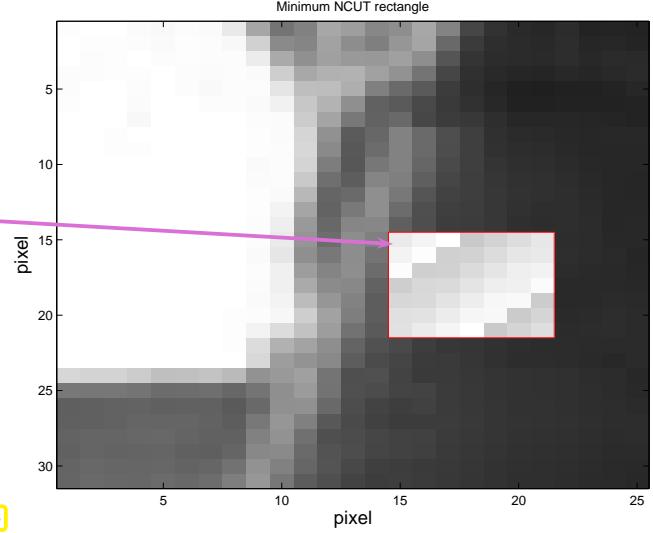


Fig. 218

△  $\text{Ncut}(\mathcal{X})$  for pixel subsets  $\mathcal{X}$  defined by sliding rectangles, see Fig. 216.

*Equivalent reformulation:*

indicator function:  $z : \{1, \dots, N\} \mapsto \{-1, 1\}, \quad z_i := z(i) = \begin{cases} 1 & , \text{if } i \in \mathcal{X}, \\ -1 & , \text{if } i \notin \mathcal{X}. \end{cases} \quad (7.3.29)$

►  $\text{Ncut}(\mathcal{X}) = \frac{\sum_{z_i>0, z_j<0} -w_{ij}z_i z_j}{\sum_{z_i>0} d_i} + \frac{\sum_{z_i>0, z_j<0} -w_{ij}z_i z_j}{\sum_{z_i<0} d_i}, \quad (7.3.30)$

$$d_i = \sum_{j \in \mathcal{V}} w_{ij} = \text{weight}(\{i\}) . \quad (7.3.31)$$

Sparse matrices:

$$\mathbf{D} := \text{diag}(d_1, \dots, d_N) \in \mathbb{R}^{N,N}, \quad \mathbf{A} := \mathbf{D} - \mathbf{W} = \mathbf{A}^\top . \quad (7.3.32)$$

Summary: (obvious) properties of these matrices

- \*  $\mathbf{A}$  has positive diagonal and non-positive off-diagonal entries.
- \*  $\mathbf{A}$  is **diagonally dominant** ( $\rightarrow$  Def. 1.8.8)  $\Rightarrow$   $\mathbf{A}$  is positive semidefinite by Lemma 1.8.12.
- \*  $\mathbf{A}$  has row sums = 0:

$$\mathbf{1}^\top \mathbf{A} = \mathbf{A} \mathbf{1} = 0 . \quad (7.3.33)$$

#### MATLAB-code 7.3.34: assembly of $\mathbf{A}$ , $\mathbf{D}$

```

1 function [A,D] = imgsegmat(P)
2 P = double(P); [n,m] = size(P);
3 spdata = zeros(4*n*m,1); spidx = zeros(4*n*m,2);
4 k = 1;
5 for ni=1:n
6   for mi=1:m
7     mni = (mi-1)*n+ni;
8     if (ni-1>0), spidx(k,:) = [mni,mni-1];
9       spdata(k) = Sfun(P(ni,mi),P(ni-1,mi));
10      k = k + 1;
11    end
12    if (ni+1<=n), spidx(k,:) = [mni,mni+1];
13      spdata(k) = Sfun(P(ni,mi),P(ni+1,mi));
14      k = k + 1;
15    end
16    if (mi-1>0), spidx(k,:) = [mni,mni-n];
17      spdata(k) = Sfun(P(ni,mi),P(ni,mi-1));
18      k = k + 1;
19    end
20    if (mi+1<=m), spidx(k,:) = [mni,mni+n];
21      spdata(k) = Sfun(P(ni,mi),P(ni,mi+1));
22      k = k + 1;
23    end
24  end
25 end
26 % Efficient initialization, see Sect. 1.7.2, Ex. 1.7.13
27 W = sparse(spidx(1:k-1,1),spidx(1:k-1,2),spdata(1:k-1),n*m,n*m);
28 D = spdiags(full(sum(W'))',[0],n*m,n*m);
29 A = D-W;

```

**Lemma 7.3.35. Ncut and Rayleigh quotient** ( $\rightarrow$  [73, Sect. 2])

With  $\mathbf{z} \in \{-1, 1\}^N$  according to (7.3.29) there holds

$$\text{Ncut}(\mathcal{X}) = \frac{\mathbf{y}^\top \mathbf{A} \mathbf{y}}{\mathbf{y}^\top \mathbf{D} \mathbf{y}} , \quad \mathbf{y} := (\mathbf{1} + \mathbf{z}) - \beta(\mathbf{1} - \mathbf{z}) , \quad \beta := \frac{\sum_{z_i > 0} d_i}{\sum_{z_i < 0} d_i} .$$

generalized Rayleigh quotient  $\rho_{\mathbf{A}, \mathbf{D}}(\mathbf{y})$

*Proof.* Note that by (7.3.29)  $(\mathbf{1} - \mathbf{z})_i = 0 \Leftrightarrow i \in \mathcal{X}$ ,  $(\mathbf{1} + \mathbf{z})_i = 0 \Leftrightarrow i \notin \mathcal{X}$ . Hence, since  $(\mathbf{1} + \mathbf{z})^\top \mathbf{D} (\mathbf{1} - \mathbf{z}) = 0$ ,

$$4 \text{Ncut}(\mathcal{X}) = (\mathbf{1} + \mathbf{z})^\top \mathbf{A} (\mathbf{1} + \mathbf{z}) \left( \frac{1}{\kappa \mathbf{1}^\top \mathbf{D} \mathbf{1}} + \frac{1}{(1 - \kappa) \mathbf{1}^\top \mathbf{D} \mathbf{1}} \right) = \frac{\mathbf{y}^\top \mathbf{A} \mathbf{y}}{\beta \mathbf{1}^\top \mathbf{D} \mathbf{1}} ,$$

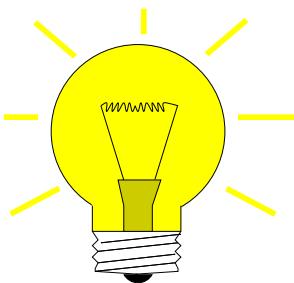
where  $\kappa := \sum_{z_i > 0} d_i / \sum_i d_i = \frac{\beta}{1 + \beta}$ . Also observe

$$\begin{aligned} \mathbf{y}^\top \mathbf{D} \mathbf{y} &= (\mathbf{1} + \mathbf{z})^\top \mathbf{D} (\mathbf{1} + \mathbf{z}) + \beta^2 (\mathbf{1} - \mathbf{z})^\top \mathbf{D} (\mathbf{1} - \mathbf{z}) = \\ &4 \left( \sum_{z_i > 0} d_i + \beta^2 \sum_{z_i < 0} d_i \right) = 4\beta \sum_i d_i = 4\beta \mathbf{1}^\top \mathbf{D} \mathbf{1} . \end{aligned}$$

This finishes the proof.  $\square$

- \* (7.3.33)  $\Rightarrow \mathbf{1} \in \text{Eig}\mathbf{A}0$
  - \* Lemma 1.8.12:  $\mathbf{A}$  diagonally dominant  $\implies \mathbf{A}$  is positive semidefinite ( $\rightarrow$  Def. 1.1.8)
- $\text{Ncut}(\mathcal{X}) \geq 0$  and 0 is the smallest eigenvalue of  $\mathbf{A}$ .

However, we are by no means interested in a minimizer  $\mathbf{y} \in \text{Span}\{\mathbf{1}\}$  (with constant entries) that does not provide a meaningful segmentation.



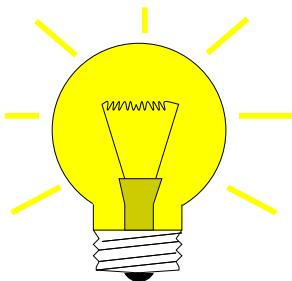
Idea: weed out undesirable constant minimizers by imposing **orthogonality constraint** (orthogonality w.r.t. inner product induced by  $\mathbf{D}$ , cf. Section 8.1)

$$\mathbf{y} \perp \mathbf{D} \mathbf{1} \Leftrightarrow \mathbf{1}^\top \mathbf{D} \mathbf{y} = 0 . \quad (7.3.36)$$

► segmentation problem (7.3.28)  $\Leftrightarrow \underset{\mathbf{y} \in \{2, -2\beta\}^N, \mathbf{1}^\top \mathbf{D} \mathbf{y} = 0}{\text{argmin}} \rho_{\mathbf{A}, \mathbf{D}}(\mathbf{y}) . \quad (7.3.37)$

still NP-hard

► Minimizing  $\text{Ncut}(\mathcal{X})$  amounts to minimizing a (generalized) Rayleigh quotient ( $\rightarrow$  Def. 7.3.16) over a *discrete* set of vectors, which is still an NP-hard problem.



Idea:

Relaxation

Discrete optimization problem → continuous optimization problem

$$(7.3.37) \rightarrow \underset{\mathbf{y} \in \mathbb{R}^N, \mathbf{y} \neq 0, \mathbf{1}^\top \mathbf{D}\mathbf{y} = 0}{\operatorname{argmin}} \rho_{\mathbf{A}, \mathbf{D}}(\mathbf{y}). \quad (7.3.38)$$

Task: (7.3.38) ⇔

Find minimizer of (generalized) Rayleigh quotient under linear constraint

Here: linear constraint on  $\mathbf{y}$ :  $\mathbf{1}^\top \mathbf{D}\mathbf{y} = 0$ 

The next theorem establishes a link between argument vectors that render the Rayleigh quotient extremal and eigenspace for extremal eigenvalues.

### Theorem 7.3.39. Rayleigh quotient theorem

Let  $\lambda_1 < \lambda_2 < \dots < \lambda_m$ ,  $m \leq n$ , be the sorted sequence of all (real!) eigenvalues of  $\mathbf{A} = \mathbf{A}^H \in \mathbb{C}^{n,n}$ . Then

$$\operatorname{Eig}\mathbf{A}\lambda_1 = \underset{\mathbf{y} \in \mathbb{C}^{n,n} \setminus \{0\}}{\operatorname{argmin}} \rho_{\mathbf{A}}(\mathbf{y}) \quad \text{and} \quad \operatorname{Eig}\mathbf{A}\lambda_m = \underset{\mathbf{y} \in \mathbb{C}^{n,n} \setminus \{0\}}{\operatorname{argmax}} \rho_{\mathbf{A}}(\mathbf{y}).$$

### Remark 7.3.40 (Min-max theorem)

Thm. 7.3.39 is an immediate consequence of the following more general and fundamentally important result.

### Theorem 7.3.41. Courant-Fischer min-max theorem → [28, Thm. 8.1.2]

Let  $\lambda_1 < \lambda_2 < \dots < \lambda_m$ ,  $m \leq n$ , be the sorted sequence of the (real!) eigenvalues of  $\mathbf{A} = \mathbf{A}^H \in \mathbb{C}^{n,n}$ . Write

$$U_0 = \{0\}, \quad U_\ell := \sum_{j=1}^{\ell} \operatorname{Eig}\mathbf{A}\lambda_j, \quad \ell = 1, \dots, m \quad \text{and} \quad U_\ell^\perp := \{\mathbf{x} \in \mathbb{C}^n : \mathbf{u}^H \mathbf{x} = 0 \ \forall \mathbf{u} \in U_\ell\}.$$

Then

$$\min_{\mathbf{y} \in U_{\ell-1}^\perp \setminus \{0\}} \rho_{\mathbf{A}}(\mathbf{y}) = \lambda_\ell, \quad 1 \leq \ell \leq m, \quad \underset{\mathbf{y} \in U_{\ell-1}^\perp \setminus \{0\}}{\operatorname{argmin}} \rho_{\mathbf{A}}(\mathbf{y}) \subset \operatorname{Eig}\mathbf{A}\lambda_\ell.$$

*Proof.* For diagonal  $\mathbf{A} \in \mathbb{R}^{n,n}$  the assertion of the theorem is obvious. Thus, Cor. 7.1.9 settles everything. □

A simple conclusion from Thm. 7.3.41: If  $\mathbf{A} = \mathbf{A}^\top \in \mathbb{R}^{n,n}$  with eigenvalues  $\lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$ , then

$$\lambda_1 = \min_{\mathbf{z} \in \mathbb{R}^n} \rho_{\mathbf{A}}(\mathbf{z}) , \quad \lambda_2 = \min_{\mathbf{z} \in \mathbb{R}^n, \mathbf{z} \perp \mathbf{v}_1} \rho_{\mathbf{A}}(\mathbf{z}) , \quad (7.3.42)$$

where  $\mathbf{v}_1 \in \text{Eig}_{\mathbf{A}} \lambda_1 \setminus \{0\}$ .

---

Well, in Lemma 7.3.35 we encounter a generalized Rayleigh quotient  $\rho_{\mathbf{A}, \mathbf{D}}(\mathbf{y})$ ! How can Thm. 7.3.39 be applied to it?

► Transformation idea:  $\rho_{\mathbf{A}, \mathbf{D}}(\mathbf{D}^{-1/2}\mathbf{z}) = \rho_{\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}}(\mathbf{z})$ ,  $\mathbf{y} \in \mathbb{R}^n$ . (7.3.43)

► Apply Thm. 7.3.41 to transformed matrix  $\tilde{\mathbf{A}} := \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ . Elementary manipulations show

$$(7.3.38) \Leftrightarrow \begin{aligned} & \underset{\mathbf{1}^\top \mathbf{D} \mathbf{y} = 0}{\operatorname{argmin}} \rho_{\mathbf{A}, \mathbf{D}}(\mathbf{y}) \stackrel{\mathbf{z} = \mathbf{D}^{1/2}\mathbf{y}}{=} \underset{\mathbf{1}^\top \mathbf{D}^{1/2}\mathbf{z} = 0}{\operatorname{argmin}} \rho_{\mathbf{A}, \mathbf{D}}(\mathbf{D}^{-1/2}\mathbf{z}) \\ & = \underset{\mathbf{1}^\top \mathbf{D}^{1/2}\mathbf{z} = 0}{\operatorname{argmin}} \rho_{\tilde{\mathbf{A}}}(\mathbf{z}) \quad \text{with} \quad \tilde{\mathbf{A}} := \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}. \end{aligned} \quad (7.3.44)$$

Related: transformation of a generalized eigenvalue problem into a standard eigenvalue problem according to

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{B}\mathbf{x} \quad \stackrel{\mathbf{z} = \mathbf{B}^{1/2}\mathbf{x}}{\implies} \quad \mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2}\mathbf{z} = \lambda \mathbf{z}. \quad (7.3.45)$$

$\mathbf{B}^{1/2} \triangleq$  square root of s.p.d. matrix  $\mathbf{B} \rightarrow$  Rem. 8.3.2.

For segmentation problem:  $\mathbf{B} = \mathbf{D}$  diagonal with positive diagonal entries, see (7.3.32)

►  $\mathbf{D}^{-1/2} = \text{diag}(d_1^{-1/2}, \dots, d_N^{-1/2})$  and  $\tilde{\mathbf{A}} := \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$  can easily be computed.

► In the sequel consider minimization problem/related eigenvalue problem

$$\mathbf{z}^* = \underset{\mathbf{1}^\top \mathbf{D}^{1/2}\mathbf{z} = 0}{\operatorname{argmin}} \rho_{\tilde{\mathbf{A}}}(\mathbf{z})$$

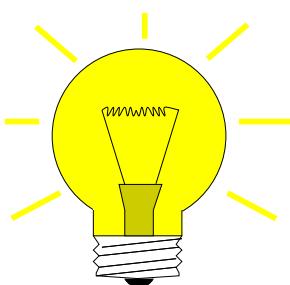
$$\longleftrightarrow \quad \tilde{\mathbf{A}}\mathbf{z} = \lambda \mathbf{z}. \quad (7.3.46)$$

Recover solution  $\mathbf{y}^*$  of (7.3.38) as  $\mathbf{y}^* = \mathbf{D}^{-1/2}\mathbf{z}^*$ .



Still, Thm. 7.3.39 cannot be applied to (7.3.46):

How to deal with constraint  $\mathbf{1}^\top \mathbf{D}^{1/2}\mathbf{z} = 0$  ?



Idea:

Penalization

Add term  $P(\mathbf{z})$  to  $\rho_{\tilde{\mathbf{A}}}(\mathbf{z})$  that becomes “sufficiently large” in case the constraint is violated.

►  $\mathbf{z}^*$  can only be a minimizer of  $\rho_{\tilde{\mathbf{A}}}(\mathbf{z}) + P(\mathbf{z})$ , if  $P(\mathbf{z}) = 0$ .

How to choose the **penalty function**  $P(\mathbf{z})$  for the segmentation problem ?

$$\left\{ \mathbf{1}^\top \mathbf{D}^{1/2} \mathbf{z} \neq 0 \Rightarrow P(\mathbf{z}) > 0 \right\} \text{ satisfied for } P(\mathbf{z}) = \mu \frac{|\mathbf{1}^\top \mathbf{D}^{1/2} \mathbf{z}|^2}{\|\mathbf{z}\|_2^2},$$

with **penalty parameter**  $\mu > 0$ .

► **penalized minimization problem**

$$\begin{aligned} \mathbf{z}^* &= \underset{\mathbf{z} \in \mathbb{R}^N \setminus \{0\}}{\operatorname{argmin}} \rho_{\tilde{\mathbf{A}}}(\mathbf{z}) + P(\mathbf{z}) = \underset{\mathbf{z} \in \mathbb{R}^N \setminus \{0\}}{\operatorname{argmin}} \rho_{\tilde{\mathbf{A}}}(\mathbf{z}) + \frac{\mathbf{z}^\top (\mathbf{D}^{1/2} \mathbf{1} \mathbf{1}^\top \mathbf{D}^{1/2}) \mathbf{z}}{\mathbf{z}^\top \mathbf{z}} \\ &= \underset{\mathbf{z} \in \mathbb{R}^N \setminus \{0\}}{\operatorname{argmin}} \rho_{\hat{\mathbf{A}}}(\mathbf{z}) \quad \text{with} \quad \hat{\mathbf{A}} := \tilde{\mathbf{A}} + \mathbf{D}^{1/2} \mathbf{1} \mathbf{1}^\top \mathbf{D}^{1/2}. \end{aligned} \quad (7.3.47)$$

dense rank-1 matrix  
↓

How to choose the penalty parameter  $\mu$  ?

In general: finding a “suitable” value for  $\mu$  may be difficult or even impossible!

Here we are lucky:

$$(7.3.33) \Rightarrow \mathbf{A} \mathbf{1} = 0 \Rightarrow \tilde{\mathbf{A}}(\mathbf{D}^{1/2} \mathbf{1}) = 0 \Leftrightarrow \mathbf{D}^{1/2} \mathbf{1} \in \operatorname{Eig} \tilde{\mathbf{A}} 0.$$

► Constraint in (7.3.46) means

Minimize over the **orthogonal complement** of an *eigenvector*. (7.3.48)

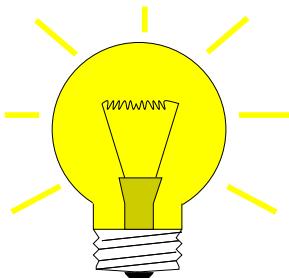
Cor. 7.1.9 ► The orthogonal complement of an eigenvector of a *symmetric* matrix is spanned by the other eigenvectors (orthonormalization of eigenvectors belonging to the same eigenvalue is assumed).

(7.3.48) ► The minimizer of (7.3.46) will be one of the other eigenvectors of  $\tilde{\mathbf{A}}$  that belongs to the smallest eigenvalue.

Note: This eigenvector  $\mathbf{z}^*$  will be orthogonal to  $\mathbf{D}^{1/2} \mathbf{1}$ , it satisfies the constraint, and, thus,  $P(\mathbf{z}^*) = 0$ !

Note: eigenspaces of  $\tilde{\mathbf{A}}$  and  $\hat{\mathbf{A}}$  agree.

Note: Lemma 1.8.12  $\implies \tilde{\mathbf{A}}$  is *positive semidefinite* ( $\rightarrow$  Def. 1.1.8) with smallest eigenvalue  $0.n$



Idea: Choose penalization parameter  $\mu$  in (7.3.47) such that  $\mathbf{D}^{1/2} \mathbf{1}$  is guaranteed not to be an eigenvector belonging to the smallest eigenvalue of  $\hat{\mathbf{A}}$ .

Safe choice: choose  $\mu$  such that  $\mathbf{D}^{1/2} \mathbf{1}$  will belong to the largest eigenvalue of  $\hat{\mathbf{A}}$ : Thm. 7.1.4 suggests

$$\mu = \left\| \tilde{\mathbf{A}} \right\|_\infty \stackrel{(1.5.74)}{=} 2. \quad (7.3.49)$$

$$\blacktriangleright \quad \boxed{\mathbf{z}^* = \underset{\mathbf{1}^\top \mathbf{D}^{1/2} \mathbf{z} = 0}{\operatorname{argmin}} \rho_{\tilde{\mathbf{A}}}(\mathbf{z}) = \underset{\mathbf{z} \neq 0}{\operatorname{argmin}} \rho_{\hat{\mathbf{A}}}(\mathbf{z})}. \quad (7.3.50)$$

By Thm. 7.3.39:

$$\begin{aligned} \mathbf{z}^* &= \text{eigenvector belonging to minimal eigenvalue of } \hat{\mathbf{A}}, \\ &\Updownarrow \\ \mathbf{z}^* &= \text{eigenvector } \perp \mathbf{D}^{1/2}\mathbf{1} \text{ belonging to minimal eigenvalue of } \tilde{\mathbf{A}}, \\ &\Updownarrow \\ \mathbf{D}^{-1/2}\mathbf{z}^* &= \text{minimizer for (7.3.38)}. \end{aligned}$$

### (7.3.51) Algorithm outline: Binary grayscale image segmentation

- ① Given similarity function  $\sigma$  compute (sparse!) matrices  $\mathbf{W}, \mathbf{D}, \mathbf{A} \in \mathbb{R}^{N,N}$ , see (7.3.26), (7.3.32).
- ② Compute  $\mathbf{y}^*$ ,  $\|\mathbf{y}^*\|_2 = 1$ , as eigenvector belonging to the *smallest eigenvalue* of  $\hat{\mathbf{A}} := \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2} + 2(\mathbf{D}^{1/2}\mathbf{1})(\mathbf{D}^{1/2}\mathbf{1})^\top$ .
- ③ Set  $\mathbf{x}^* = \mathbf{D}^{-1/2}\mathbf{y}^*$  and define the image segment as pixel set

$$\mathcal{X} := \left\{ i \in \{1, \dots, N\} : x_i^* > \frac{1}{N} \sum_{i=1}^N x_i^* \right\}. \quad (7.3.52)$$

mean value of entries of  $\mathbf{x}^*$

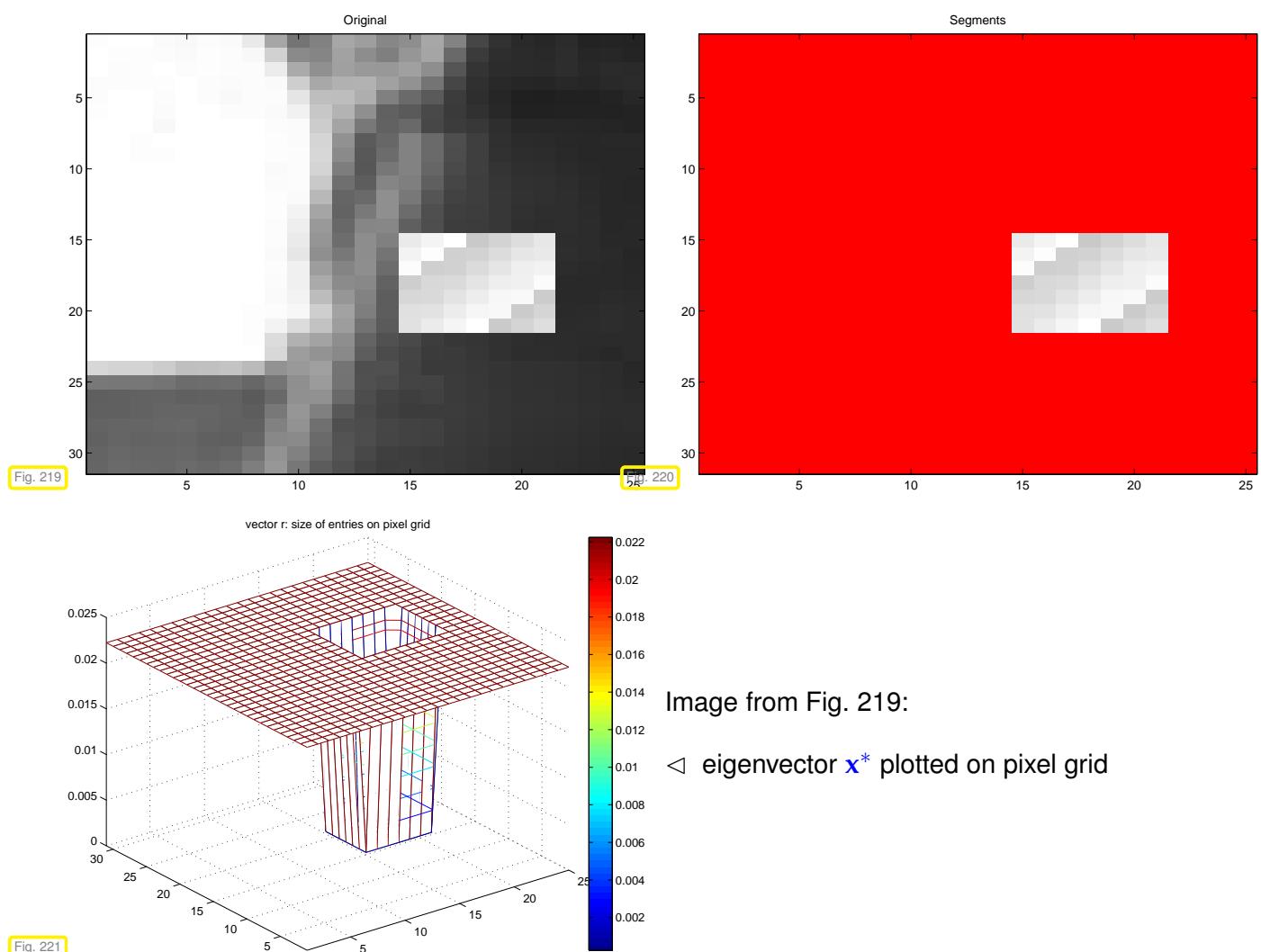
### MATLAB-code 7.3.53: 1st stage of segmentation of grayscale image

```

1 % Read image and build matrices, see Code 7.3.34 and (7.3.32)
2 P = imread('image.pbm'); [m,n] = size(P); [A,D] = imgsegmat(P);
3 % Build scaling matrices
4 N = size(A,1); dv = sqrt(spdiags(A, 0));
5 Dm = spdiags(1./dv, [0], N, N); % D^{-1/2}
6 Dp = spdiags(dv, [0], N, N); % D^{-1/2}
7 % Build (densely populated !) matrix \hat{A}
8 C = Dp*ones(N,1); Ah = Dm*A*Dm + 2*c*c';
9 % Compute and sort eigenvalues; grossly inefficient !
10 [W,E] = eig(full(Ah)); [ev,idx] = sort(diag(E));
11 % Obtain eigenvector x^* belonging to 2nd smallest generalized
12 % eigenvalue of A and D
13 x = W(:,1); x = Dm*x;
14 % Extract segmented image
15 xs = reshape(x,m,n); Xidx = find(xs > (sum(sum(xs)) / (n*m)));

```

1st-stage of segmentation of  $31 \times 25$  grayscale pixel image (root.pbm, red pixels  $\hat{=} \mathcal{X}$ ,  $\sigma(x,y) = \exp(-(x-y/10)^2)$ )



To identify more segments, the same algorithm is *recursively applied* to segment parts of the image already determined.

Practical segmentation algorithms rely on many more steps of which the above algorithm is only one, preceded by substantial preprocessing. Moreover, they dispense with the strictly local perspective adopted above and take into account more distant connections between image parts, often in a randomized fashion [73].

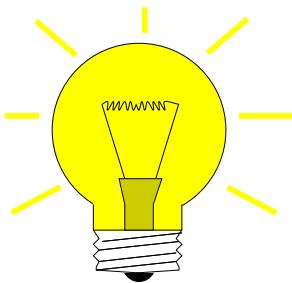
The image segmentation problem falls into the wider class of **graph partitioning problems**. Methods based on (a few of) the eigenvector of the connectivity matrix belonging to the smallest eigenvalues are known as **spectral partitioning methods**. The eigenvector belonging to the smallest non-zero eigenvalue that we computed above is usually called the **Fiedler vector** of the graph, see [74, 4].

---

The solution of the image segmentation problem by means of `eig` in Code 7.3.53 amounts a tremendous waste of computational resources: we compute *all* eigenvalues/eigenvectors of dense matrices, though only a single eigenvector associated with the smallest eigenvalue is of interest.

This motivates the quest to find *efficient* numerical methods for the following task.

Task:	given $\mathbf{A} \in \mathbb{K}^{n,n}$ , find <b>smallest</b> (in modulus) eigenvalue of regular $\mathbf{A} \in \mathbb{K}^{n,n}$ and (an) associated eigenvector.
-------	---



If  $\mathbf{A} \in \mathbb{K}^{n,n}$  regular:

$$\text{Smallest (in modulus) EV of } \mathbf{A} = \left( \text{Largest (in modulus) EV of } \mathbf{A}^{-1} \right)^{-1}$$



Direct power method ( $\rightarrow$  Section 7.3.1) for  $\mathbf{A}^{-1} = \text{inverse iteration}$

#### MATLAB-code 7.3.54: inverse iteration for computing $\lambda_{\min}(\mathbf{A})$ and associated eigenvector

```

1 function [lmin,y] = invit(A,tol)
2 [L,U] = lu(A); % single intial LU-factorization, see Rem. 1.6.87
3 n = size(A,1); x = rand(n,1); x = x/norm(x); % random initial guess
4 y = U\ (L\x); lmin = 1/norm(y); y = y*lmin; lold = 0;
5 while (abs(lmin-lold) > tol*lmin) % termination, if small relative
   change
6   lold = lmin; x = y;
7   y = U\ (L\x); % core iteration: y = A-1x,
8   lmin = 1/norm(y); % new approximation of λmin(A)
9   y = y*lmin; % normalization y := y / ||y||2
10 end

```

Note: reuse of LU-factorization, see Rem. 1.6.87

#### Remark 7.3.55 (Shifted inverse iteration)

More general task:

For  $\alpha \in \mathbb{C}$  find  $\lambda \in \sigma(\mathbf{A})$  such that  $|\alpha - \lambda| = \min\{|\alpha - \mu|, \mu \in \sigma(\mathbf{A})\}$

Shifted inverse iteration: [15, Alg .7.24]

$$\mathbf{z}^{(0)} \text{ arbitrary}, \quad \mathbf{w} = (\mathbf{A} - \alpha \mathbf{I})^{-1} \mathbf{z}^{(k-1)}, \quad \mathbf{z}^{(k)} := \frac{\mathbf{w}}{\|\mathbf{w}\|_2}, \quad k = 1, 2, \dots, \quad (7.3.56)$$

where:  $(\mathbf{A} - \alpha \mathbf{I})^{-1} \mathbf{z}^{(k-1)} \doteq \text{solve } (\mathbf{A} - \alpha \mathbf{I}) \mathbf{w} = \mathbf{z}^{(k-1)}$  based on Gaussian elimination ( $\leftrightarrow$  a single LU-factorization of  $\mathbf{A} - \alpha \mathbf{I}$  as in Code 7.3.54).

#### Remark 7.3.57 ((Nearly) singular LSE in shifted inverse iteration)

What if “by accident”  $\alpha \in \sigma(\mathbf{A})$  ( $\Leftrightarrow \mathbf{A} - \alpha \mathbf{I}$  singular) ?

Stability of Gaussian elimination/LU-factorization ( $\rightarrow ??$ ) will ensure that “ $\mathbf{w}$  from (7.3.56) points in the right direction”

In other words, roundoff errors may badly affect the length of the solution  $\mathbf{w}$ , but not its direction.

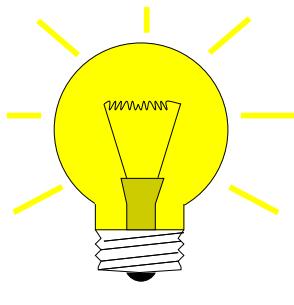
Practice [29]: If, in the course of Gaussian elimination/LU-factorization a zero pivot element is really encountered, then we just *replace it with*  $\text{eps}$ , in order to avoid  $\inf$  values!

Thm. 7.3.21 ➤ Convergence of shifted inverse iteration for  $\mathbf{A}^H = \mathbf{A}$ :

Asymptotic linear convergence, Rayleigh quotient  $\rightarrow \lambda_j$  with rate

$$\frac{|\lambda_j - \alpha|}{\min\{|\lambda_i - \alpha|, i \neq j\}} \quad \text{with} \quad \lambda_j \in \sigma(\mathbf{A}), \quad |\alpha - \lambda_j| \leq |\alpha - \lambda| \quad \forall \lambda \in \sigma(\mathbf{A}).$$

► Extremely fast for  $\alpha \approx \lambda_j$ !



Idea:

A posteriori adaptation of shift

Use  $\alpha := \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})$  in  $k$ -th step of inverse iteration.



### (7.3.58) Rayleigh quotient iteration → [42, Alg. 25.2]

#### MATLAB-code 7.3.59: Rayleigh quotient iteration (for normal $\mathbf{A} \in \mathbb{R}^{n,n}$ )

```

1 function [z,lmin] = rqui(A,tol,maxit)
2 alpha = 0; n = size(A,1);
3 z = rand(size(A,1),1); z = z/norm(z); % z^(0)
4 for i=1:maxit
5     z = (A-alpha*speye(n))\z; % z^(k+1) = (A - rho_A(z^(k))I)^{-1}x^(k)
6     z = z/norm(z); lmin=dot(A*z,z); % Computation of rho_A(z^(k+1))
7     if (abs(alpha-lmin) < tol*lmin) % Desired relative accuracy
        reached ?
        break;
    end;
8     alpha = lmin;
9 end
10

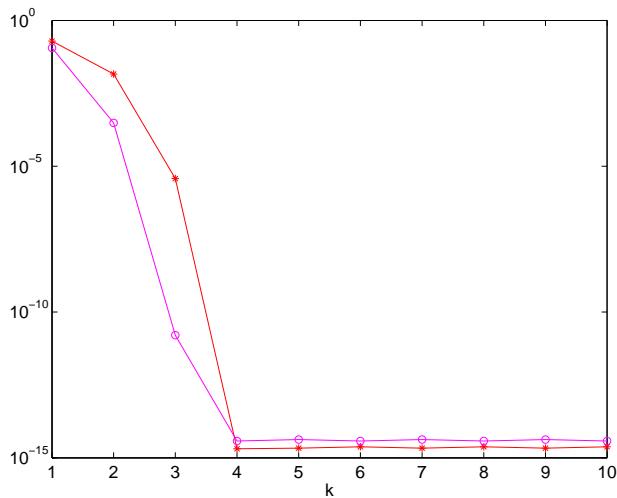
```

Line 5: note use of `speye` to preserve sparse matrix data format!

- \* Drawback compared with Code 7.3.54: reuse of LU-factorization no longer possible.
- \* Even if LSE nearly singular, stability of Gaussian elimination guarantees correct direction of  $\mathbf{z}$ , see discussion in Rem. 7.3.57.

#### Example 7.3.60 (Rayleigh quotient iteration)

Monitored: iterates of Rayleigh quotient iteration (7.3.59) for s.p.d.  $\mathbf{A} \in \mathbb{R}^{n,n}$



$k$	$ \lambda_{\min} - \rho_A(z^{(k)}) $	$\ z^{(k)} - x_j\ $
1	0.09381702342056	0.20748822490698
2	0.00029035607981	0.01530829569530
3	0.000000000001783	0.00000411928759
4	0.000000000000000	0.000000000000000
5	0.000000000000000	0.000000000000000

**MATLAB-code 7.3.61:**

```
d = (1:10)'; n = length(d); Z =
diag(sqrt(1:n), 0)+ones(n, n);
[Q, R] = qr(Z);
o : |λ_min - ρ_A(z^(k))|
* : ||z^(k) - x_j||, λ_min = λ_j, x_j ∈ EigAλ_j,
A = Q * diag((d, 0) * Q);
: ||x_j||_2 = 1
```

**Theorem 7.3.62.** → [42, Thm. 25.4]

If  $A = A^H$ , then  $\rho_A(z^{(k)})$  converges locally of order 3 (→ Def. 2.1.17) to the smallest eigenvalue (in modulus), when  $z^{(k)}$  are generated by the Rayleigh quotient iteration (7.3.59).

### 7.3.3 Preconditioned inverse iteration (PINVIT)

Task: given  $A \in \mathbb{K}^{n,n}$ , find **smallest** (in modulus) eigenvalue of regular  $A \in \mathbb{K}^{n,n}$  and (an) associated eigenvector.

► Options: inverse iteration (→ Code 7.3.54) and Rayleigh quotient iteration (7.3.59).



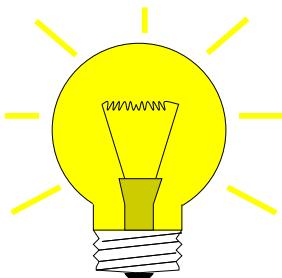
What if direct solution of  $Ax = b$  not feasible ?

This can happen, in case

- for large sparse  $A$  the amount of fill-in exhausts memory, despite sparse elimination techniques (→ Section 1.7.5),
- $A$  is available only through a routine `evalA(x)` providing  $A \times \text{vector}$ .

We expect that an approximate solution of the linear systems of equations encountered during inverse iteration should be sufficient, because we are dealing with approximate eigenvectors anyway.

Thus, iterative solvers for solving  $Aw = z^{(k-1)}$  may be considered, see Chapter 8. However, the required accuracy is not clear a priori. Here we examine an approach that completely dispenses with an iterative solver and uses a *preconditioner* (→ Notion 8.3.3) instead.



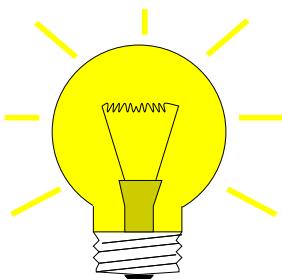
Idea: (for inverse iteration without shift,  $\mathbf{A} = \mathbf{A}^H$  s.p.d.)  
Instead of solving  $\mathbf{A}\mathbf{w} = \mathbf{z}^{(k-1)}$  compute  $\mathbf{w} = \mathbf{B}^{-1}\mathbf{z}^{(k-1)}$  with  
“inexpensive” s.p.d. **approximate inverse**  $\mathbf{B}^{-1} \approx \mathbf{A}^{-1}$

>  $\mathbf{B} \triangleq$  Preconditioner for  $\mathbf{A}$ , see Notion 8.3.3



Possible to replace  $\mathbf{A}^{-1}$  with  $\mathbf{B}^{-1}$  in inverse iteration ?

**NO**, because we are not interested in smallest eigenvalue of  $\mathbf{B}$  !



Replacement  $\mathbf{A}^{-1} \rightarrow \mathbf{B}^{-1}$  possible only when applied to **residual quantity**  
**residual quantity** = quantity that  $\rightarrow 0$  in the case of convergence to exact solution

Natural residual quantity for eigenvalue problem  $\mathbf{Ax} = \lambda\mathbf{x}$ :

$$\mathbf{r} := \mathbf{Az} - \rho_{\mathbf{A}}(\mathbf{z})\mathbf{z}, \quad \rho_{\mathbf{A}}(\mathbf{z}) = \text{Rayleigh quotient} \rightarrow \text{Def. 7.3.16}.$$

Note: only *direction* of  $\mathbf{A}^{-1}\mathbf{z}$  matters in inverse iteration (7.3.56)

$$(\mathbf{A}^{-1}\mathbf{z}) \parallel (\mathbf{z} - \mathbf{A}^{-1}(\mathbf{Az} - \rho_{\mathbf{A}}(\mathbf{z})\mathbf{z})) \Rightarrow \text{defines same next iterate!}$$



[Preconditioned inverse iteration (PINVIT) for s.p.d.  $\mathbf{A}$ ]

$$\begin{aligned} \mathbf{z}^{(0)} &\text{ arbitrary,} & \mathbf{w} &= \mathbf{z}^{(k-1)} - \mathbf{B}^{-1}(\mathbf{Az}^{(k-1)} - \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{z}^{(k-1)}), \\ & & \mathbf{z}^{(k)} &= \frac{\mathbf{w}}{\|\mathbf{w}\|_2}, & k &= 1, 2, \dots. \end{aligned} \quad (7.3.63)$$

**MATLAB-code 7.3.64: preconditioned inverse iteration (7.3.63)**

```

1 function [lmin,z,res] =
2     pinvit(evalA,n,invB,tol,maxit)
3 % invB ≈ handle to function implementing
4 % preconditioner  $B^{-1}$ 
5 z = (1:n)'; z = z/norm(z); % initial guess
6 res = []; rho = 0;
7 for i=1:maxit
8     v = evalA(z); rhon = dot(v,z); % Rayleigh
9         quotient
10    r = v - rhon*z; % residual
11    z = z - invB(r); % iteration according to
12        (7.3.63)
13    z = z/norm(z); % normalization
14    res = [res; rhon]; % tracking iteration
15    if abs(rho-rhon) < tol*abs(rhon)), break;
16    else rho = rhon; end
17 end
18 lmin = dot(evalA(z),z); res = [res; lmin],

```

Computational effort:

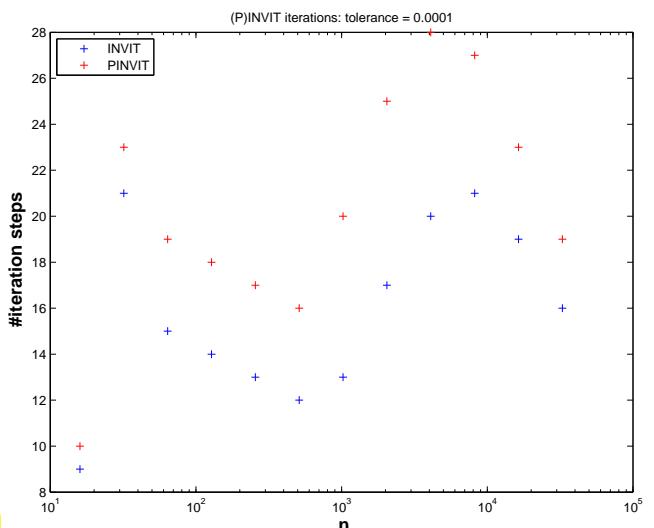
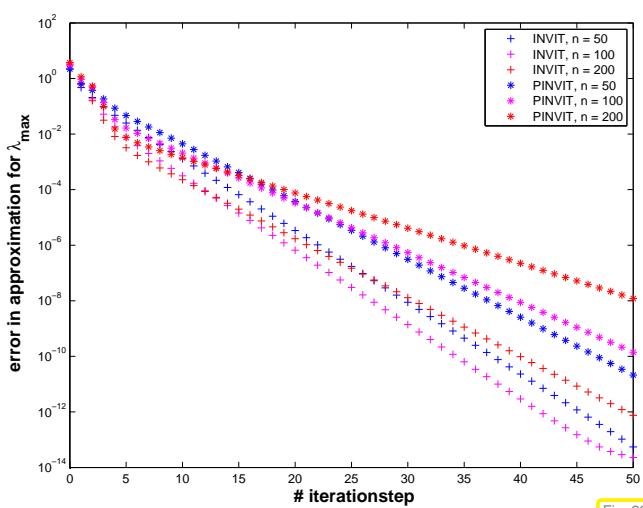
1 matrix  $\times$  vector  
 1 evaluation of pre-  
 conditioner  
 A few  
 AXPY-operations

**Example 7.3.65 (Convergence of PINVIT)**S.p.d. matrix  $A \in \mathbb{R}^{n,n}$ , tridiagonal preconditioner, see Ex. 8.3.11**MATLAB-code 7.3.66:**

```

1 A = spdiags(repmat([1/n,-1,2*(1+1/n),-1,1/n],n,1),
2 [-n/2,-1,0,1,n/2],n,n);
3 evalA = @(x) A*x;
4 % inverse iteration
5 invB = @(x) A\b;
6 % tridiagonal preconditioning
7 B = spdiags(spdiags(A,[-1,0,1]),[-1,0,1],n,n); invB = @(x) B\b;

```

Monitored: error decay during iteration of Code 7.3.64:  $|\rho_A(z^{(k)}) - \lambda_{\min}(A)|$ 

Observation: linear convergence of eigenvectors also for PINVIT.

Theory [58, 57]:

- \* linear convergence of (7.3.63)
- \* fast convergence, if spectral condition number  $\kappa(\mathbf{B}^{-1}\mathbf{A})$  small

The theory of PINVIT [58, 57] is based on the identity

$$\mathbf{w} = \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{A}^{-1}\mathbf{z}^{(k-1)} + (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})(\mathbf{z}^{(k-1)} - \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{A}^{-1}\mathbf{z}^{(k-1)}). \quad (7.3.67)$$

For small residual  $\mathbf{A}\mathbf{z}^{(k-1)} - \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{z}^{(k-1)}$  PINVIT almost agrees with the regular inverse iteration.

### 7.3.4 Subspace iterations

#### Remark 7.3.68 (Excited resonances)

Consider the non-autonomous ODE (excited harmonic oscillator)

$$\ddot{y} + \lambda^2 y = \cos(\omega t), \quad (7.3.69)$$

with general solution

$$y(t) = \begin{cases} \frac{1}{\lambda^2 - \omega^2} \cos(\omega t) + A \cos(\lambda t) + B \sin(\lambda t) & , \text{ if } \lambda \neq \omega, \\ \frac{t}{2\omega} \sin(\omega t) + A \cos(\lambda t) + B \sin(\lambda t) & , \text{ if } \lambda = \omega. \end{cases} \quad (7.3.70)$$

► growing solutions possible in **resonance case**  $\lambda = \omega$ !

Now consider harmonically excited vibration modelled by ODE

$$\ddot{\mathbf{y}} + \mathbf{A}\mathbf{y} = \mathbf{b} \cos(\omega t), \quad (7.3.71)$$

with *symmetric, positive (semi)definite* matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$ ,  $\mathbf{b} \in \mathbb{R}^n$ . By Cor. 7.1.9 there is an *orthogonal* matrix  $\mathbf{Q} \in \mathbb{R}^{n,n}$  such that

$$\mathbf{Q}^\top \mathbf{A} \mathbf{Q} = \mathbf{D} := \text{diag}(\lambda_1, \dots, \lambda_n).$$

where the  $0 \leq \lambda_1 < \lambda_2 < \dots < \lambda_n$  are the eigenvalues of  $\mathbf{A}$ .

► Transform ODE as in Ex. 7.0.7: with  $\mathbf{z} = \mathbf{Q}^\top \mathbf{y}$

$$(7.3.71) \quad \mathbf{Q}^\top \ddot{\mathbf{z}} + \mathbf{D}\mathbf{z} = \mathbf{Q}^\top \mathbf{b} \cos(\omega t).$$

We have obtained decoupled linear 2nd-order scalar ODEs of the type (7.3.69).

► (7.3.71) can have growing (with time) solutions, if  $\omega = \sqrt{\lambda_i}$  for some  $i = 1, \dots, n$

If  $\omega = \sqrt{\lambda_j}$  for one  $j \in \{1, \dots, n\}$ , then the solution for the initial value problem for (7.3.71) with  $\mathbf{y}(0) = \dot{\mathbf{y}}(0) = 0$  ( $\leftrightarrow \mathbf{z}(0) = \dot{\mathbf{z}}(0) = 0$ ) is

$$\mathbf{z}(t) \sim \frac{t}{2\omega} \sin(\omega t) \mathbf{e}_j + \text{bounded oscillations}$$

$$\Updownarrow$$

$$\mathbf{y}(t) \sim \frac{t}{2\omega} \sin(\omega t) (\mathbf{Q})_{:,j} + \text{bounded oscillations}.$$

$\downarrow$   
 $j$ -th eigenvector of  $\mathbf{A}$

► Eigenvectors of  $\mathbf{A} \leftrightarrow$  excitable states

### Example 7.3.72 (Vibrations of a truss structure)

cf. [42, Sect. 3], MATLAB's `truss demo`

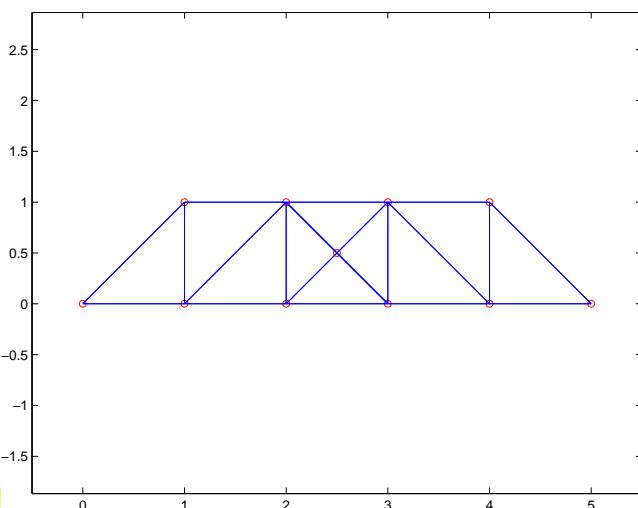


Fig. 224

▷ a “bridge” truss]

A **truss** is a structure composed of (massless) rods and point masses; we consider in-plane (2D) trusses.

Encoding: positions of masses + (sparse) connectivity matrix

### MATLAB-code 7.3.73: Data for “bridge truss”

```

1 % Data for truss structure "bridge"
2 pos = [ 0 0; 1 0; 2 0; 3 0; 4 0; 5 0; 1 1; 2 1; 3 1; 4 1; 2.5 0.5];
3 con = [1 2; 2 3; 3 4; 4 5; 5 6; 1 7; 2 7; 3 8; 2 8; 4 8; 5 9; 5 10; 6 10; 7
8; 8 9; 9 10; 8 11 ...;
; 9 11; 3 11; 4 11; 4 9];
4 n = size(pos,1);
5 top = sparse(con(:,1),con(:,2),ones(size(con,1),1),n,n);
6 top = sign(top+top');
```

Assumptions:

- ★ Truss in static equilibrium (perfect balance of forces at each point mass).
- ★ Rods are *perfectly elastic* (i.e., frictionless).

► Hook's law holds for force in the direction of a rod:

$$F = \alpha \frac{\Delta l}{l}, \quad (7.3.74)$$

where      ★  $l$  is the equilibrium length of the rod,

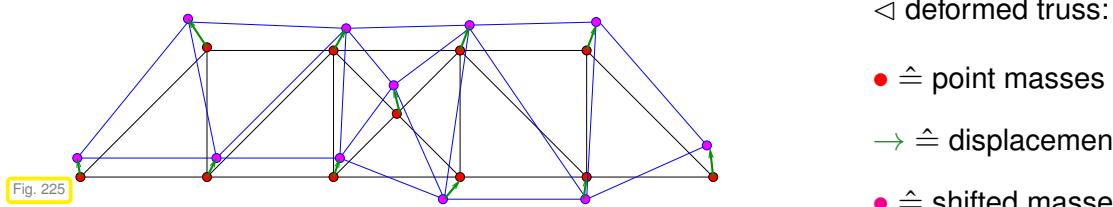
★  $\Delta l$  is the elongation of the rod effected by the force  $F$  in the direction of the rod

\*  $\alpha$  is a material coefficient (Young's modulus).

$n$  point masses are numbered  $1, \dots, n$ :  $\mathbf{p}^i \in \mathbb{R}^2 \hat{=} \text{position of } i\text{-th mass}$

We consider a swaying truss: description by time-dependent **displacements**  $\mathbf{u}^i(t) \in \mathbb{R}^2$  of point masses:

$$\text{position of } i\text{-th mass at time } t = \mathbf{p}^i + \mathbf{u}^i(t);.$$



Equilibrium length and (time-dependent) elongation of rod connecting point masses  $i$  and  $j$ ,  $i \neq j$ :

$$l_{ij} := \|\Delta \mathbf{p}^{ji}\|_2, \quad \Delta \mathbf{p}^{ji} := \mathbf{p}^j - \mathbf{p}^i, \quad (7.3.75)$$

$$\Delta l_{ij}(t) := \|\Delta \mathbf{p}^{ji} + \Delta \mathbf{u}^{ji}(t)\|_2 - l_{ij}, \quad \Delta \mathbf{u}^{ji}(t) := \mathbf{u}^j(t) - \mathbf{u}^i(t). \quad (7.3.76)$$

► Extra (reaction) force on masses  $i$  and  $j$ :

$$F_{ij}(t) = -\alpha_{ij} \frac{\Delta l_{ij}}{l_{ij}} \cdot \frac{\Delta \mathbf{p}^{ji} + \Delta \mathbf{u}^{ji}(t)}{\|\Delta \mathbf{p}^{ji} + \Delta \mathbf{u}^{ji}(t)\|_2}. \quad (7.3.77)$$

Assumption:	Small displacements
-------------	---------------------

► Possibility of **linearization** by neglecting terms of order  $\|\mathbf{u}^i\|_2^2$

$$(7.3.75) \quad \blacktriangleright \quad F_{ij}(t) = \alpha_{ij} \left( \frac{1}{\|\Delta \mathbf{p}^{ji} + \Delta \mathbf{u}^{ji}(t)\|_2} - \frac{1}{\|\Delta \mathbf{p}^{ji}\|_2} \right) \cdot (\Delta \mathbf{p}^{ji} + \Delta \mathbf{u}^{ji}(t)). \quad (7.3.78)$$

### Lemma 7.3.79. Taylor expansion of inverse distance function

For  $\mathbf{x} \in \mathbb{R}^d \setminus \{0\}$ ,  $\mathbf{y} \in \mathbb{R}^d$ ,  $\|\mathbf{y}\|_2 < \|\mathbf{x}\|_2$  holds for  $\mathbf{y} \rightarrow 0$

$$\frac{1}{\|\mathbf{x} + \mathbf{y}\|_2} = \frac{1}{\|\mathbf{x}\|_2} - \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|_2^3} + O(\|\mathbf{y}\|_2^2).$$

*Proof.* Simple Taylor expansion up to linear term for  $f(\mathbf{x}) = (x_1^2 + \dots + x_d^2)^{-1/2}$  and  $f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + \mathbf{grad} f(\mathbf{x}) \cdot \mathbf{y} + O(\|\mathbf{y}\|_2^2)$ . □

Linearization of force: apply Lemma 7.3.79 to (7.3.78) and drop terms  $O(\|\Delta \mathbf{u}^{ji}\|_2^2)$ :

$$\begin{aligned} \blacktriangleright \quad F_{ij}(t) &\approx -\alpha_{ij} \frac{\Delta \mathbf{p}^{ji} \cdot \Delta \mathbf{u}^{ji}(t)}{l_{ij}^3} \cdot (\Delta \mathbf{p}^{ji} + \Delta \mathbf{u}^{ji}(t)) \\ &\approx -\alpha_{ij} \frac{\Delta \mathbf{p}^{ji} \cdot \Delta \mathbf{u}^{ji}(t)}{l_{ij}^3} \cdot \Delta \mathbf{p}^{ji}. \end{aligned} \quad (7.3.80)$$

Newton's second law of motion: ( $F_i \hat{=} \text{total force acting on } i\text{-th mass}$ )

$$m_i \frac{d^2}{dt^2} \mathbf{u}^i(t) = F_i = \sum_{\substack{j=1 \\ j \neq i}}^n -F_{ij}(t), \quad (7.3.81)$$

$m_i \hat{=} \text{mass of point mass } i.$

$$\Rightarrow m_i \frac{d^2}{dt^2} \mathbf{u}^i(t) = \sum_{\substack{j=1 \\ j \neq i}}^n \alpha_{ij} \frac{1}{l_{ij}^3} (\Delta \mathbf{p}^{ji} (\Delta \mathbf{p}^{ji})^\top) (\mathbf{u}^j(t) - \mathbf{u}^i(t)). \quad (7.3.82)$$

Compact notation: collect all displacements into one vector  $\mathbf{u}(t) = (\mathbf{u}^i(t))_{i=1}^n \in \mathbb{R}^{2n}$

$$(7.3.82) \Rightarrow \mathbf{M} \frac{d^2 \mathbf{u}}{dt^2}(t) + \mathbf{A} \mathbf{u}(t) = \mathbf{f}(t). \quad (7.3.83)$$

with **mass matrix**  $\mathbf{M} = \text{diag}(m_1, m_1, \dots, m_n, m_n)$

and **stiffness matrix**  $\mathbf{A} \in \mathbb{R}^{2n,2n}$  with  $2 \times 2$ -blocks

$$\begin{aligned} (\mathbf{A})_{2i-1:2i, 2i-1:2i} &= \sum_{\substack{j=1 \\ j \neq i}}^n \alpha_{ij} \frac{1}{l_{ij}^3} (\Delta \mathbf{p}^{ji} (\Delta \mathbf{p}^{ji})^\top), \quad i = 1, \dots, n, \\ (\mathbf{A})_{2i-1:2i, 2j-1:2j} &= -\alpha_{ij} \frac{1}{l_{ij}^3} (\Delta \mathbf{p}^{ji} (\Delta \mathbf{p}^{ji})^\top), \quad i \neq j. \end{aligned} \quad (7.3.84)$$

and external forces  $\mathbf{f}(t) = (\mathbf{f}^i(t))_{i=1}^n$ .

Note: stiffness matrix  $\mathbf{A}$  is *symmetric, positive semidefinite* ( $\rightarrow$  Def. 1.1.8).

Rem. 7.3.68: if periodic external forces  $\mathbf{f}(t) = \cos(\omega t)\mathbf{f}$ ,  $\mathbf{f} \in \mathbb{R}^{2n}$ , (wind, earthquake) act on the truss they can excite vibrations of (linearly in time) growing amplitude, if  $\omega$  coincides with  $\sqrt{\lambda_j}$  for an eigenvalue  $\lambda_j$  of  $\mathbf{A}$ .

Excited vibrations can lead to the collapse of a truss structure, cf. the notorious Tacoma-Narrows bridge disaster.

$\Rightarrow$  It is essential to know whether eigenvalues of a truss structure fall into a range that can be excited by external forces.

These will typically (\*) be the **low modes**  $\leftrightarrow$  a few of the smallest eigenvalues.

(\*) Reason: fast oscillations will quickly be damped due to friction, which was neglected in our model.)

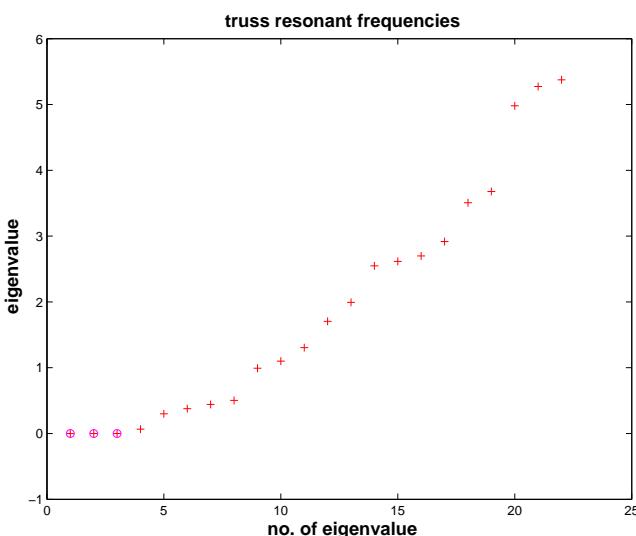
#### MATLAB-code 7.3.85: Computing resonant frequencies and modes of elastic truss

```
1 function [lambda,V] = trussvib(pos,top)
2 % Computes vibration modes of a truss structure, see Ex. 7.3.72. Mass
3 % point
4 % positions passed in the n×2-matrix pos and the connectivity encoded |
```

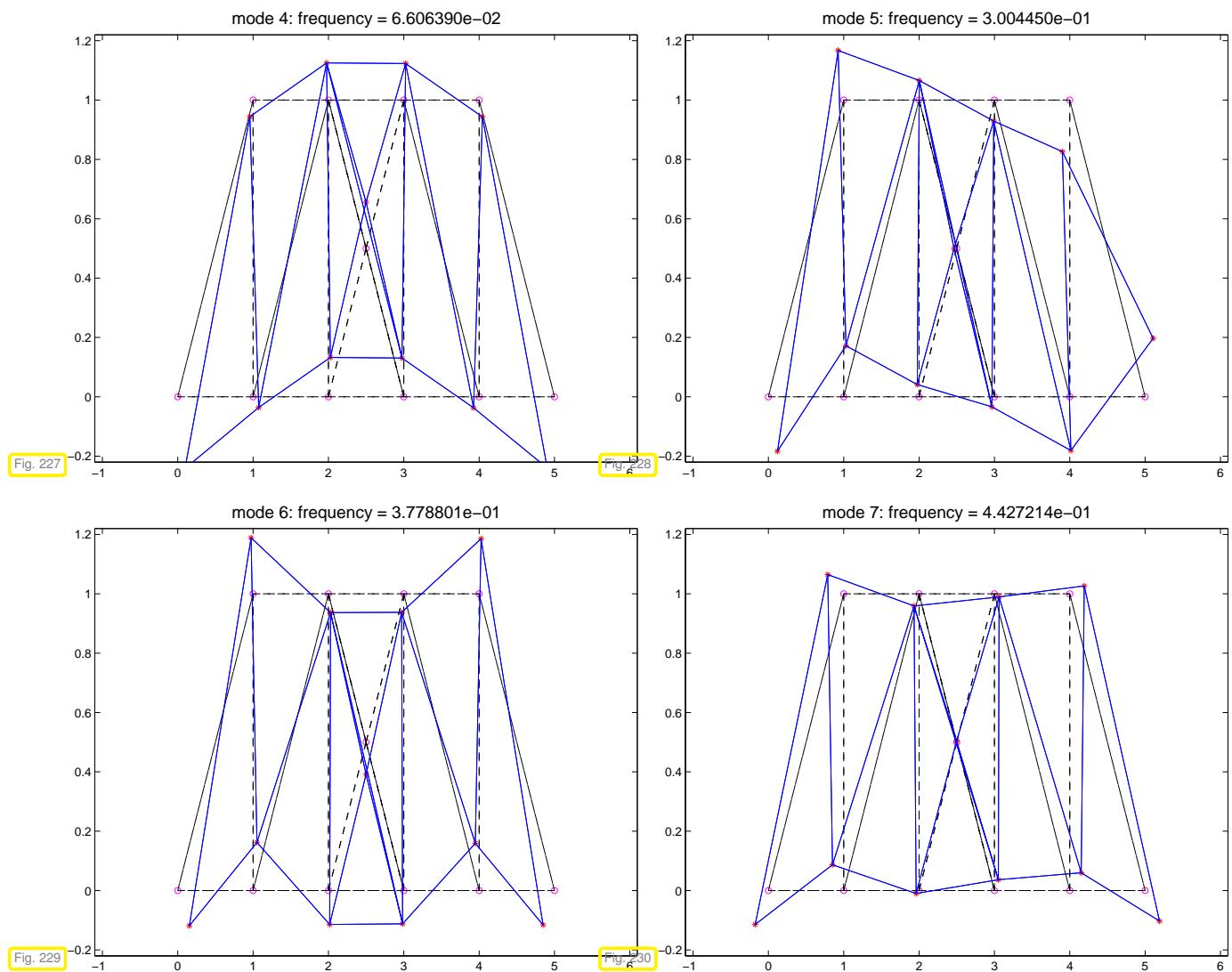
```

4 |   in
4 | % the sparse symmetric matrix top. In addition top(i,j) also stores
4 | % the
5 | % Young's moduli  $\alpha_{ij}$ .
6 | % The  $2n$  resonant frequencies are returned in the vector lambda, the
7 | % eigenmodes in the column of V, where entries at odd positions contain
7 | % the
8 | %  $x_1$ -coordinates, entries at even positions the  $x_2$ -coordinates
9 | n = size(pos,1); % no. of point masses
10 | % Assembly of stiffness matrix according to (7.3.84)
11 | A = zeros(2*n,2*n);
12 | [Iidx,Jidx] = find(top); idx = [Iidx,Jidx]; % Find connected masses
13 | for ij = idx'
14 |   i = ij(1); j = ij(2);
15 |   dp = [pos(j,1);pos(j,2)] - [pos(i,1);pos(i,2)]; %  $\Delta p^{ji}$ 
16 |   lij = norm(dp); %  $l_{ij}$ 
17 |   A(2*i-1:2*i,2*j-1:2*j) = -(dp*dp')/(lij^3);
18 | end
19 | % Set Young's moduli  $\alpha_{ij}$  (stored in top matrix)
20 | A = A.*full(kron(top,[1 1;1 1]));
21 | % Set  $2 \times 2$  diagonal blocks
22 | for i=1:n
23 |   A(2*i-1:2*i,2*i-1) = -sum(A(2*i-1:2*i,1:2:end))';
24 |   A(2*i-1:2*i,2*i) = -sum(A(2*i-1:2*i,2:2:end))';
25 | end
26 | % Compute eigenvalues and eigenmodes
27 | [V,D] = eig(A); lambda = diag(D);

```



▷ resonant frequencies of bridge truss from Fig. 224.  
The stiffness matrix will always possess three zero eigenvalues corresponding to **rigid body modes** (= displacements without change of length of the rods)



To compute *a few* of a truss's lowest resonant frequencies and excitable mode, we need efficient numerical methods for the following tasks. Obviously, Code 7.3.85 cannot be used for large trusses, because `eig` invariable operates on dense matrices and will be prohibitively slow and gobble up huge amounts of memory, also recall the discussion of Code 7.3.53.

Task:	Compute $m$ , $m \ll n$ , of the smallest/largest (in modulus) eigenvalues of $\mathbf{A} = \mathbf{A}^H \in \mathbb{C}^{n,n}$ and associated eigenvectors.
-------	---

Of course, we aim to tackle this task by iterative methods generalizing power iteration ( $\rightarrow$  Section 7.3.1) and inverse iteration ( $\rightarrow$  Section 7.3.2).

### 7.3.4.1 Orthogonalization

Preliminary considerations (in  $\mathbb{R}$ ,  $m = 2$ ):

According to Cor. 7.1.9: For  $\mathbf{A} = \mathbf{A}^T \in \mathbb{R}^{n,n}$  there is a factorization  $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{U}^T$  with  $\mathbf{D} = \text{diag}(\lambda_1, \dots, \lambda_n)$ ,  $\lambda_j \in \mathbb{R}$ ,  $\lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$ , and  $\mathbf{U}$  orthogonal. Thus,  $\mathbf{u}_j := (\mathbf{U})_{:,j}$ ,  $j = 1, \dots, n$ , are (mutually orthogonal) eigenvectors of  $\mathbf{A}$ .

Assume  $0 \leq \lambda_1 \leq \dots \leq \lambda_{n-2} < \lambda_{n-1} < \lambda_n$  (largest eigenvalues are simple).

If we just carry out the direct power iteration (7.3.12) for two vectors both sequences will converge to the largest (in modulus) eigenvector. However, we recall that all eigenvectors are mutually orthogonal. This

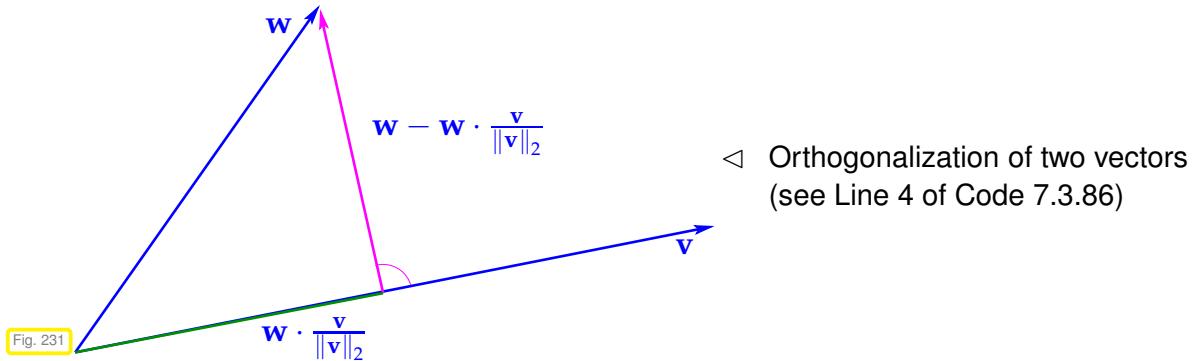
suggests that we orthogonalize the iterates of the second power iteration (that is to yield the eigenvector for the second largest eigenvalue) with respect to those of the first. This idea spawns the following iteration, cf. Gram-Schmidt orthogonalization in (8.2.11):

**MATLAB-code 7.3.86: one step of subspace power iteration,  $m = 2$** 

```

1 function [v,w] = sspowitstep(A,v,w)
2 v = A*v; w = A*w; % "power iteration", cf. (7.3.12)
3 % orthogonalization, cf. Gram-Schmidt orthogonalization (8.2.11)
4 v = v/norm(v); w = w - dot(v,w)*v; w = w/norm(w); % now w ⊥ v

```



Analysis through eigenvector expansions ( $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ ,  $\|\mathbf{v}\|_2 = \|\mathbf{w}\|_2 = 1$ )

$$\begin{aligned}
& \mathbf{v} = \sum_{i=1}^n \alpha_j \mathbf{u}_j , \quad \mathbf{w} = \sum_{i=1}^n \beta_j \mathbf{u}_j , \\
\Rightarrow & \mathbf{Av} = \sum_{i=1}^n \lambda_j \alpha_j \mathbf{u}_j , \quad \mathbf{Aw} = \sum_{i=1}^n \lambda_j \beta_j \mathbf{u}_j , \\
& \mathbf{v}_0 := \frac{\mathbf{v}}{\|\mathbf{v}\|_2} = \left( \sum_{i=1}^n \lambda_j^2 \alpha_j^2 \right)^{-1/2} \sum_{i=1}^n \lambda_j \alpha_j \mathbf{u}_j , \\
& \mathbf{Aw} - (\mathbf{v}_0^\top \mathbf{Aw}) \mathbf{v}_0 = \sum_{i=1}^n \left( \beta_j - \left( \sum_{i=1}^n \lambda_j^2 \alpha_j \beta_j / \sum_{i=1}^n \lambda_j^2 \alpha_j^2 \right) \alpha_j \right) \lambda_j \mathbf{u}_j .
\end{aligned}$$

We notice that  $\mathbf{v}$  is just mapped to the next iterate in the regular direct power iteration (7.3.12). After many steps, it will be very close to  $\mathbf{u}_n$ , and, therefore, we may now assume  $\mathbf{v} = \mathbf{u}_n \Leftrightarrow \alpha_j = \delta_{j,n}$  (Kronecker symbol).

$$\begin{aligned}
\mathbf{z} &:= \mathbf{Aw} - (\mathbf{v}_0^\top \mathbf{Aw}) \mathbf{v}_0 = 0 \cdot \mathbf{u}_n + \sum_{i=1}^{n-1} \lambda_i \beta_i \mathbf{u}_i , \\
\mathbf{w}^{(\text{new})} &:= \frac{\mathbf{z}}{\|\mathbf{z}\|_2} = \left( \sum_{i=1}^{n-1} \lambda_i^2 \beta_i^2 \right)^{-1/2} \sum_{i=1}^{n-1} \lambda_i \beta_i \mathbf{u}_i .
\end{aligned}$$

The sequence  $\mathbf{w}^{(k)}$  produced by repeated application of the mapping given by Code 7.3.86 asymptotically (that is, when  $\mathbf{v}^{(k)}$  has already converged to  $\mathbf{u}_n$ ) agrees with the sequence produced by the direct power method for  $\tilde{\mathbf{A}} := \mathbf{U} \text{diag}(\lambda_1, \dots, \lambda_{n-1}, 0)$ . Its convergence will be governed by the relative gap  $\lambda_{n-2}/\lambda_{n-1}$ , see Thm. 7.3.21.

However: if  $\mathbf{v}^{(k)}$  itself converges slowly, this reasoning does not apply.

### Example 7.3.87 (Subspace power iteration with orthogonal projection)

☞ construction of matrix  $\mathbf{A} = \mathbf{A}^T$  as in Ex. 7.3.60

#### MATLAB-code 7.3.88: power iteration with orthogonal projection for two vectors

```

1 function sppowitdriver(d,maxit)
2 % monitor power iteration with orthogonal projection for finding
3 % the two largest (in modulus) eigenvalues and associated eigenvectors
4 % of a symmetric matrix with prescribed eigenvalues passed in d
5 if (nargin < 10), maxit = 20; end
6 if (nargin < 1), d = (1:10)'; end
7 % Generate matrix
8 n = length(d);
9 Z = diag(sqrt(1:n),0) + ones(n,n);
10 [Q,R] = qr(Z); % generate orthogonal matrix
11 A = Q*diag(d,0)*Q'; % "synthetic"  $\mathbf{A} = \mathbf{A}^T$  with spectrum  $\sigma(\mathbf{A}) = \{d_1, \dots, d_n\}$ 
12 % Compute "exact" eigenvectors and eigenvalues
13 [V,D] = eig(A); [d,idx] = sort(diag(D)),
14 v_ex = V(:,idx(n)); w_ex = V(:,idx(n-1));
15 lv_ex = d(n); lw_ex = d(n-1);

16
17 v = ones(n,1); w = (-1).^v; % (Arbitrary) initial guess for
18 % eigenvectors
19 v = v/norm(v); w = w/norm(w);
20 result = [];
21 for k=1:maxit
22     v_new = A*v; w_new = A*w; % "power iteration", cf. (7.3.12)
23     % Rayleigh quotients provide approximate eigenvalues
24     lv = dot(v_new,v); lw = dot(w_new,w);
25     % orthogonalization, cf. Gram-Schmidt orthogonalization (8.2.11):
26     %  $w \perp v$ 
27     v = v_new/norm(v_new); w = w_new - dot(v,w_new)*v; w =
28         w/norm(w);
29     % Record errors in eigenvalue and eigenvector approximations. Note
30     % that the
31     % direction of the eigenvectors is not specified.
32     result = [result; k, abs(lv-lv_ex), abs(lw-lw_ex), ...
33             min(norm(v-v_ex),norm(v+v_ex)),
34             min(norm(w-w_ex),norm(w+w_ex))];
35
36 end

37
38 figure('name','sspowit');
39 semilogy(result(:,1),result(:,2),'m-+',...
40           result(:,1),result(:,3),'r-*',...
41           result(:,1),result(:,4),'k^-',...
42           result(:,1),result(:,5),'b-p');
43 title('d = [0.5*(1:8),9.5,10]');
44 xlabel('{\bf power iteration step}', 'fontsize', 14);
45 ylabel('{\bf error}', 'fontsize', 14);
46 legend('error in \lambda_n', 'error in \lambda_{n-1}', 'error in
47           v', 'error in w', 'location', 'northeast');
```

```

41 print -depsc2 '../PICTURES/sspowitcvg1.eps';

42
43 rates = result(2:end,2:end)./result(1:end-1,2:end);
44 figure('name','rates');
45 plot(result(2:end,1),rates(:,1),'m+-',...
46      result(2:end,1),rates(:,2),'r-*',...
47      result(2:end,1),rates(:,3),'k-^',...
48      result(2:end,1),rates(:,4),'b-p');
49 axis([0 maxit 0.5 1]);
50 title('d = [0.5*(1:8),9.5,10]');
51 xlabel('{\bf power iteration step}', 'fontsize', 14);
52 ylabel('{\bf error quotient}', 'fontsize', 14);
53 legend('error in \lambda_n', 'error in \lambda_{n-1}', 'error in v',...
54      'error in w', 'location', 'southeast');
55 print -depsc2 '../PICTURES/sspowitcvgrates1.eps';

```

$\sigma(\mathbf{A}) = \{1, 2, \dots, 10\}$ :

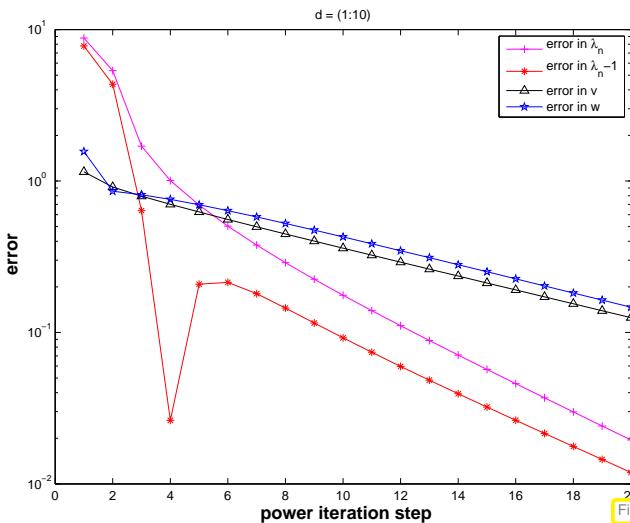


Fig. 232

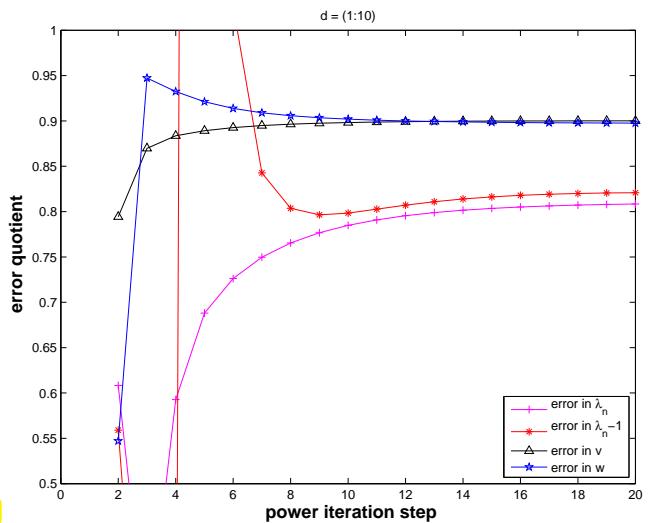


Fig. 233

$\sigma(\mathbf{A}) = \{0.5, 1, \dots, 4, 9.5, 10\}$ :

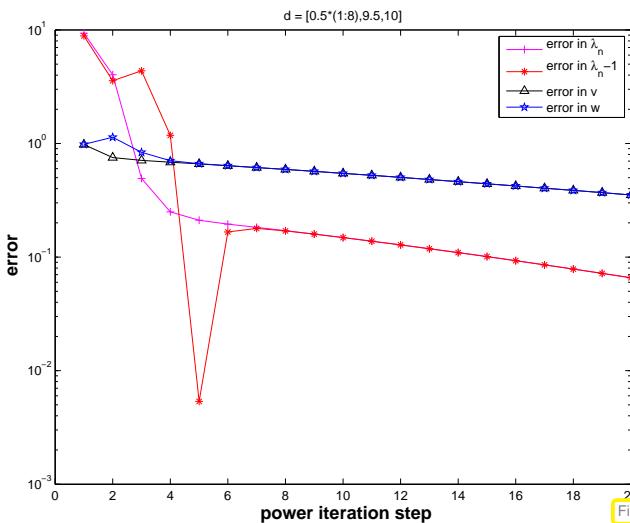


Fig. 234

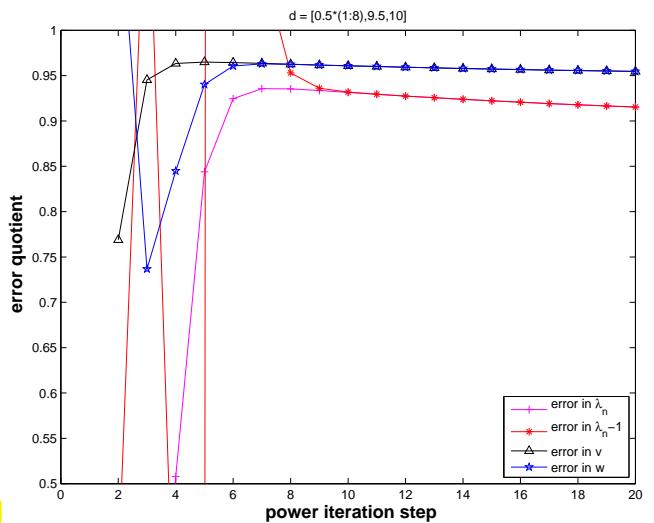


Fig. 235

Issue: generalization of orthogonalization idea to subspaces of dimension  $> 2$

Nothing new:

### Gram-Schmidt orthonormalization

(→ [59, Thm. 4.8], [34, Alg. 6.1], [63, Sect. 3.4.3])

Given: linearly independent vectors  $\mathbf{v}_1, \dots, \mathbf{v}_m \in \mathbb{R}^n$ ,  $m \in \mathbb{N}$

Sought: vectors  $\mathbf{q}_1, \dots, \mathbf{q}_m \in \mathbb{R}^n$  such that

$$\textcircled{2} \quad \mathbf{q}_l^\top \mathbf{q}_k = \delta_{lk} \quad (\text{orthonormality}), \quad (7.3.89)$$

$$\textcircled{2} \quad \text{Span}\{\mathbf{q}_1, \dots, \mathbf{q}_k\} = \text{Span}\{\mathbf{v}_1, \dots, \mathbf{v}_k\} \quad \text{for all } k = 1, \dots, m. \quad (7.3.90)$$

Constructive proof & algorithm for Gram-Schmidt orthonormalization:

$$\begin{aligned} \mathbf{z}_1 &= \mathbf{v}_1, \\ \mathbf{z}_2 &= \mathbf{v}_2 - \frac{\mathbf{v}_2^\top \mathbf{z}_1}{\mathbf{z}_1^\top \mathbf{z}_1} \mathbf{z}_1, \\ \mathbf{z}_3 &= \mathbf{v}_3 - \frac{\mathbf{v}_3^\top \mathbf{z}_1}{\mathbf{z}_1^\top \mathbf{z}_1} \mathbf{z}_1 - \frac{\mathbf{v}_3^\top \mathbf{z}_2}{\mathbf{z}_2^\top \mathbf{z}_2} \mathbf{z}_2, \\ &\vdots \end{aligned} \quad (7.3.91)$$

$$+ \text{normalization} \quad \mathbf{q}_k = \frac{\mathbf{z}_k}{\|\mathbf{z}_k\|_2}, \quad k = 1, \dots, m. \quad (7.3.92)$$

Easy computation: the vectors  $\mathbf{q}_1, \dots, \mathbf{q}_m$  produced by (7.3.91) satisfy (7.3.89) and (7.3.90).

#### MATLAB-code 7.3.93: Gram-Schmidt orthonormalization (do not use, unstable algorithm!)

```

1 function Q = gso(V)
2 % Gram-Schmidt orthonormalization of the columns of V ∈ ℝn,m, see
3 % (7.3.91). The vectors q1, ..., qm are returned as the columns of
4 % the orthogonal matrix Q.
5 m = size(V, 2);
6 Q = V(:, 1)/norm(V(:, 1)); % normalization
7 for l=2:m
8     q = V(:, l);
9     % orthogonalization
10    for k=1:l-1
11        q = q - dot(Q(:, k), V(:, l)) * Q(:, k);
12    end
13    Q = [Q, q/norm(q)]; % normalization
14 end

```



Warning! Code 7.3.93 provides an unstable implementation of Gram-Schmidt orthonormalization: for large  $n, m$  impact of round-off will destroy the orthogonality of the columns of  $\mathbf{Q}$ .

A stable implementation of Gram-Schmidt orthogonalization of the columns of a matrix  $\mathbf{V} \in \mathbb{K}^{n,m}$ ,  $m \leq n$ , is provided by the following MATLAB command:

$[Q, ~] = qr(V, 0)$  (Asymptotic computational cost:  $O(m^2n)$ )  
dummy return value (for our purposes) dummy argument

Detailed description of the algorithm behind  $qr$  and meaning of the return value  $R \rightarrow ??$ .

### Example 7.3.94 (qr based orthogonalization, $m = 2$ )

The following two MATLAB code snippets perform the same function, cf. Code 7.3.86:

#### MATLAB-code 7.3.95:

```

1 v = v/norm(v);
2 w = w - dot(v,w)*v;
3 w = w/norm(w);

```

#### MATLAB-code 7.3.96:

```

1 [Q,R] = qr([v,w],0);
2 v = Q(:,1);
3 w = Q(:,2);

```

Explanation ➤ discussion of Gram-Schmidt orthonormalization.

#### MATLAB-code 7.3.97: General subspace power iteration step with qr based orthonormalization

```

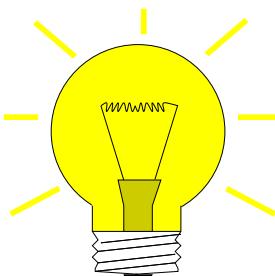
1 function V = sspowitstep(A,V)
2 % power iteration with orthonormalization for A = A^T.
3 % columns of matrix V span subspace for power iteration.
4 V = A*V; % actual power iteration on individual columns
5 [V,R] = qr(V,0); % Gram-Schmidt orthonormalization (7.3.91)

```

#### 7.3.4.2 Ritz projection

Observations on Code 7.3.86:

- \* the first column of  $\mathbf{V}$ ,  $(\mathbf{V})_{:,1}$ , is a sequence of vectors created by the standard direct power method (7.3.12).
- \* reasoning: the other columns of  $\mathbf{V}$ , after each multiplication with  $\mathbf{A}$  can be expected to contain a significant component in the direction of the eigenvector associated with the eigenvalue of largest modulus.



Idea: use information in  $(\mathbf{V})_{:,2}, \dots, (\mathbf{V})_{:,end}$  to accelerate convergence of  $(\mathbf{V})_{:,1}$ .

Since the columns of  $\mathbf{V}$  span a *subspace* of  $\mathbb{R}^n$ , this idea can be recast as the following task:

Task: given  $\mathbf{v}_1, \dots, \mathbf{v}_k \in \mathbb{K}^n$ ,  $k \ll n$ , extract (good approximations of) eigenvectors of  $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$  contained in  $\text{Span}\{\mathbf{v}_1, \dots, \mathbf{v}_m\}$ .

We take for granted that  $\{\mathbf{v}_1, \dots, \mathbf{v}_m\}$  is linearly independent.

Assumption:

$$\text{Eig}\mathbf{A}\lambda \cap V := \text{Span}\{\mathbf{v}_1, \dots, \mathbf{v}_m\} \neq \{0\}$$

$$\begin{aligned}
 &\Leftrightarrow V \text{ contains an eigenvector of } \mathbf{A} \\
 &\Leftrightarrow \exists \mathbf{w} \in V \setminus \{\mathbf{0}\}: \mathbf{Aw} = \lambda \mathbf{w} \\
 &\Leftrightarrow \exists \mathbf{u} \in \mathbb{K}^m \setminus \{\mathbf{0}\}: \mathbf{Av}_i = \lambda \mathbf{v}_i \\
 &\Rightarrow \exists \mathbf{u} \in \mathbb{K}^m \setminus \{\mathbf{0}\}: \mathbf{V}^H \mathbf{AVu} = \lambda \mathbf{V}^H \mathbf{Vu} ,
 \end{aligned} \tag{7.3.98}$$

where  $\mathbf{V} := (\mathbf{v}_1, \dots, \mathbf{v}_m) \in \mathbb{K}^{n,m}$  and we used

$$V = \{\mathbf{Vu} : \mathbf{u} \in \mathbb{K}^m\} \quad (\text{linear combinations of the } \mathbf{v}_i).$$

(7.3.98)  $\Rightarrow \mathbf{u} \in \mathbb{K}^k \setminus \{\mathbf{0}\}$  solves  $m \times m$  generalized eigenvalue problem

$$(\mathbf{V}^H \mathbf{AV}) \mathbf{u} = \lambda (\mathbf{V}^H \mathbf{V}) \mathbf{u} . \tag{7.3.99}$$

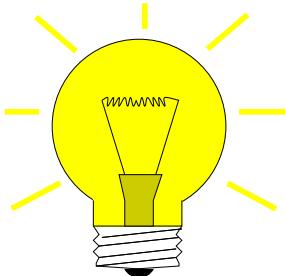
Note:

$\{\mathbf{v}_1, \dots, \mathbf{v}_m\}$  linearly independent

$\Updownarrow$   
 $\mathbf{V}$  has full rank  $m$  ( $\rightarrow$  Def. 1.6.9)

$\Updownarrow$   
 $\mathbf{V}^H \mathbf{V}$  is symmetric positive definite ( $\rightarrow$  Def. 1.1.8)

If our initial assumption holds true and  $\mathbf{u}$  solves (7.3.99) and is a simple eigenvalue, a corresponding  $\mathbf{x} \in \text{Eig} \mathbf{A} \lambda$  can be recovered as  $\mathbf{x} = \mathbf{Vu}$ .



Idea: Given a subspace  $V = \text{Im}(\mathbf{V}) \subset \mathbb{K}^n$ ,  $\mathbf{V} \in \mathbb{K}^{n,m}$ ,  $\dim(V) = m$ , obtain **approximations** of (a few) eigenvalues and eigenvectors  $\mathbf{x}_1, \dots, \mathbf{x}_m$  of  $\mathbf{A}$  by

① solving the generalized eigenvalue problem (7.3.99)

$\rightarrow$  eigenvectors  $\mathbf{u}_1, \dots, \mathbf{u}_k \in \mathbb{K}^m$  (linearly independent),

② and transforming them back according to  $\mathbf{x}_k = \mathbf{Vu}_k$ ,  $k = 1, \dots, m$ .

Terminology: (7.3.99) is called the **Ritz projection** of EVP  $\mathbf{Ax} = \lambda \mathbf{x}$  onto  $V$

Terminology:  $\sigma(\mathbf{V}^H \mathbf{AV}) \doteq$  **Ritz values**,  
eigenvectors of  $\mathbf{V}^H \mathbf{AV} \doteq$  **Ritz vectors**

Example: Ritz projection of  $\mathbf{Ax} = \lambda \mathbf{x}$  onto  $\text{Span}\{\mathbf{v}, \mathbf{w}\}$ :

$$(\mathbf{v}, \mathbf{w})^H \mathbf{A} (\mathbf{v}, \mathbf{w}) \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \lambda (\mathbf{v}, \mathbf{w})^H (\mathbf{v}, \mathbf{w}) \begin{pmatrix} \alpha \\ \beta \end{pmatrix} .$$

Note: If  $\mathbf{V}$  is *unitary* ( $\rightarrow$  Def. 4.2.2), then the generalized eigenvalue problem (7.3.99) will become a standard linear eigenvalue problem.

### Remark 7.3.100 (Justification of Ritz projection by min-max theorem)

We revisit  $m = 2$ , see Code 7.3.86. Recall that by the min-max theorem Thm. 7.3.41

$$\mathbf{u}_n = \underset{\mathbf{x} \in \mathbb{R}^n}{\text{argmax}} \rho_{\mathbf{A}}(\mathbf{x}) , \quad \mathbf{u}_{n-1} = \underset{\mathbf{x} \in \mathbb{R}^n, \mathbf{x} \perp \mathbf{u}_n}{\text{argmax}} \rho_{\mathbf{A}}(\mathbf{x}) . \tag{7.3.101}$$

Idea: maximize Rayleigh quotient over  $\text{Span}\{\mathbf{v}, \mathbf{w}\}$ , where  $\mathbf{v}, \mathbf{w}$  are output by Code 7.3.86. This leads to the optimization problem

$$(\alpha^*, \beta^*) := \underset{\alpha, \beta \in \mathbb{R}, \alpha^2 + \beta^2 = 1}{\operatorname{argmax}} \rho_{\mathbf{A}}(\alpha \mathbf{v} + \beta \mathbf{w}) = \underset{\alpha, \beta \in \mathbb{R}, \alpha^2 + \beta^2 = 1}{\operatorname{argmax}} \rho_{(\mathbf{v}, \mathbf{w})^\top \mathbf{A}(\mathbf{v}, \mathbf{w})} \left( \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \right). \quad (7.3.102)$$

Then a better approximation for the eigenvector to the largest eigenvalue is

$$\mathbf{v}^* := \alpha^* \mathbf{v} + \beta^* \mathbf{w}.$$

Note that  $\|\mathbf{v}^*\|_2 = 1$ , if both  $\mathbf{v}$  and  $\mathbf{w}$  are normalized, which is guaranteed in Code 7.3.86.

Then, orthogonalizing  $\mathbf{w}$  w.r.t  $\mathbf{v}^*$  will produce a new iterate  $\mathbf{w}^*$ .

Again the min-max theorem Thm. 7.3.41 tells us that we can find  $(\alpha^*, \beta^*)^\top$  as eigenvector to the largest eigenvalue of

$$(\mathbf{v}, \mathbf{w})^\top \mathbf{A}(\mathbf{v}, \mathbf{w}) \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \lambda \begin{pmatrix} \alpha \\ \beta \end{pmatrix}. \quad (7.3.103)$$

Since eigenvectors of symmetric matrices are mutually orthogonal, we find  $\mathbf{w}^* = \alpha_2 \mathbf{v} + \beta_2 \mathbf{w}$ , where  $(\alpha_2, \beta_2)^\top$  is the eigenvector of (7.3.103) belonging to the smallest eigenvalue. This assumes orthonormal vectors  $\mathbf{v}, \mathbf{w}$ .

#### MATLAB-code 7.3.104: one step of subspace power iteration with Ritz projection, matrix version

```

1 function V = sspowitsteppr(A,V)
2 V = A*V; % power iteration applied to columns of V
3 [Q,R] = qr(V,0); % orthonormalization, see Section 7.3.4.1
4 [U,D] = eig(Q'*A*Q); % Solve Ritz projected m x m eigenvalue problem
5 V = Q*U; % recover approximate eigenvectors
6 ev = diag(D); % approximate eigenvalues

```

Note that the orthogonalization step in Code 7.3.104 is actually redundant, if exact arithmetic could be employed, because the Ritz projection could also be realized by solving the generalized eigenvalue problem.

However, prior orthogonalization is essential for numerical stability ( $\rightarrow$  Def. 1.5.80), cf. the discussion in ??.

#### Example 7.3.105 (Power iteration with Ritz projection)

Listing 7.1: Main loop: power iteration with Ritz projection for two eigenvectors

```

1 % See Code 7.3.88 for generation of matrix A and output
2 for k=1:maxit
3     v_new = A*v; w_new = A*w; % "power iteration", cf. (7.3.12)
4     [Q,R] = qr([v_new,w_new],0); % orthogonalization, see Sect. 7.3.4.1
5     [U,D] = eig(Q'*A*Q); % Solve Ritz projected eigenvalue problem
6     [ev,idx] = sort(abs(diag(D))), % Sort eigenvalues
7     w = Q*U(:,idx(1)); v = Q*U(:,idx(2)); % Recover approximate
        eigenvectors
8

```

```

9 % Record errors in eigenvalue and eigenvector approximations. Note that
10 % the
11 % direction of the eigenvectors is not specified.
12 result = [result; k, abs(ev(2)-lv_ex), abs(ev(1)-lw_ex), ...
13 min(norm(v-v_ex),norm(v+v_ex)), ...
14 min(norm(w-w_ex),norm(w+w_ex))];
end

```

Matrix as in Ex. 7.3.87,  $\sigma(\mathbf{A}) = \{0.5, 1, \dots, 4, 9.5, 10\}$ :

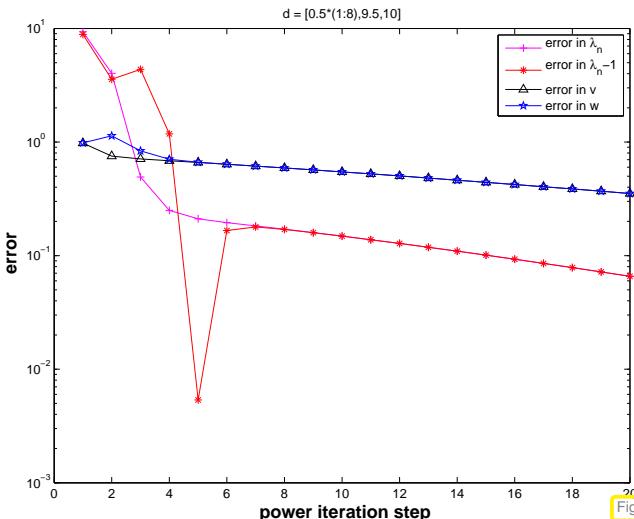
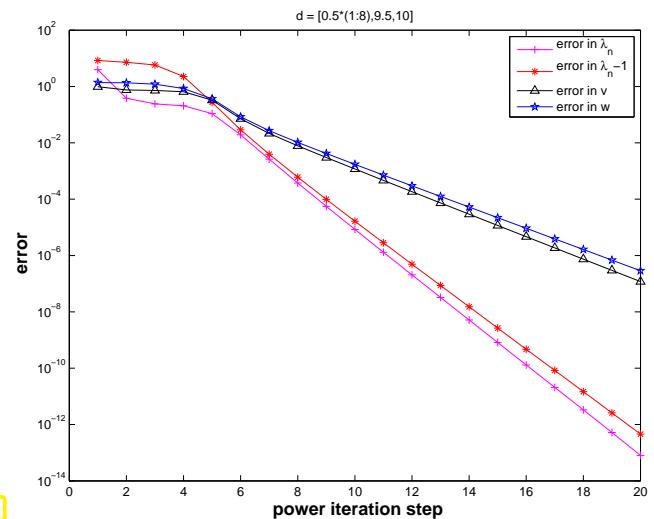


Fig. 236



simple orthonormalization, Ex. 7.3.87

with Ritz projection, Code 7.3.104

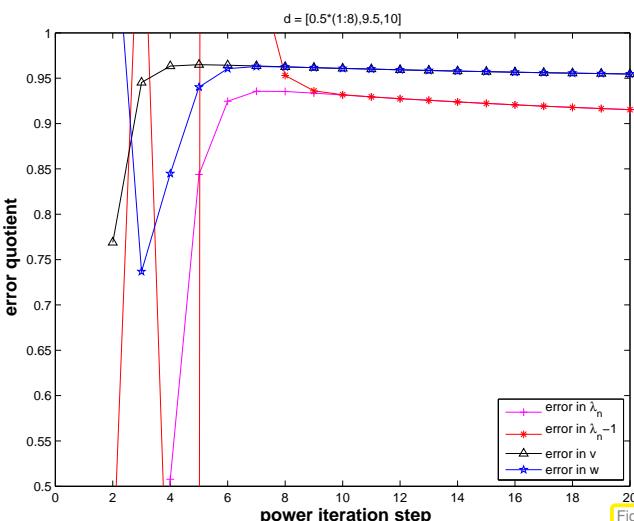
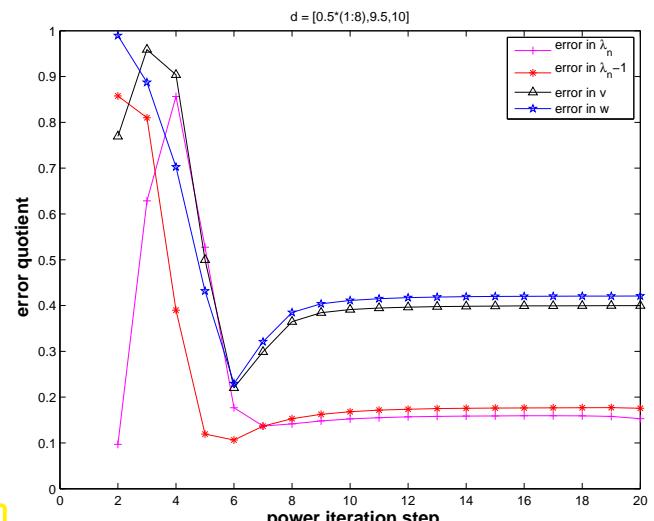


Fig. 238



simple orthonormalization, Ex. 7.3.87

with Ritz projection, Code 7.3.104

Observation: tremendous acceleration of power iteration through Ritz projection, convergence still linear but with much better rates.

In Code 7.3.104: diagonal entries of  $\mathbf{D}$  provide approximations of eigenvalues. Their (relative) changes can be used as a termination criterion.

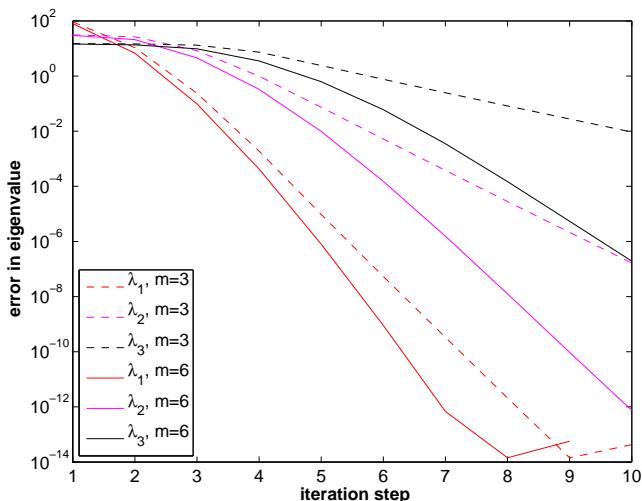
### (7.3.106) Subspace variant of direct power method with Ritz projection

**MATLAB-code 7.3.107: Subspace power iteration with Ritz projection**

```

1 function [ev,V] = sspowitrp(A,k,m,tol,maxit)
2 % Power iteration with Ritz projection for matrix  $\mathbf{A} = \mathbf{A}^T \in \mathbb{R}^{n,n}$ :
3 % Subspace of dimension  $m \leq n$  is used to compute the  $k \leq m$  largest
4 % eigenvalues of  $\mathbf{A}$  and associated eigenvectors.
5 n = size(A,1); V = eye(n,m); d = zeros(m,1); % (Arbitrary) initial
   eigenvectors
6 % The approximate eigenvectors are stored in the columns of  $\mathbf{V} \in \mathbb{R}^{n,m}$ 
7 for i=1:maxit
8   [Q,R] = qr(A*V,0); % Power iteration and orthonormalization
9   [U,D] = eig(Q'*A*Q); % Small  $m \times m$  eigenvalue problem for Ritz
   projection
10  [ev,idx] = sort(diag(D)); % eigenvalue approximations in diagonal
    of D
11  V = Q*U; % 2nd part of Ritz projection
12  if (abs(ev-d) < tol*max(abs(ev))), break; end
13  d = ev;
14 end
15 ev = ev(m-k+1:end);
16 V = V(:,idx(m-k+1:end));

```

**Example 7.3.108 (Convergence of subspace variant of direct power method)**

S.p.d. test matrix:  $a_{ij} := \min\left\{\frac{i}{j}, \frac{j}{i}\right\}$   
 $n=200$ ;  $\mathbf{A} = \text{gallery('lehmer', } n)$ ;  
 “Initial eigenvector guesses”:  
 $\mathbf{V} = \text{eye}(n, m)$ ;

- Observation:  
linear convergence of eigenvalues
- choice  $m > k$  boosts convergence  
of eigenvalues

**Remark 7.3.109 (Subspace power methods)**

Analogous to § 7.3.106: construction of subspace variants of inverse iteration (→ Code 7.3.54), PINVIT (7.3.63), and Rayleigh quotient iteration (7.3.59).

## 7.4 Krylov Subspace Methods



*Supplementary reading.* [42, Sect. 30]

All power methods ( $\rightarrow$  Section 7.3) for the eigenvalue problem (EVP)  $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$  only rely on the last iterate to determine the next one (1-point methods, cf. (2.1.4))

$\triangleright$  NO MEMORY, cf. discussion in the beginning of Section 8.2.

“Memory for power iterations”: pursue same idea that led from the gradient method, § 8.1.11, to the conjugate gradient method, § 8.2.17: use information from previous iterates to achieve efficient minimization over larger and larger subspaces.

Min-max theorem,  
Thm. 7.3.41 :  $\mathbf{A} = \mathbf{A}^H \Rightarrow$  EVPs  $\Leftrightarrow$  Finding extrema/stationary points  
of Rayleigh quotient ( $\rightarrow$  Def. 7.3.16)

Setting: EVP  $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$  for real s.p.d. ( $\rightarrow$  Def. 1.1.8) matrix  $\mathbf{A} = \mathbf{A}^T \in \mathbb{R}^{n,n}$

notations used below:  $0 < \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$ : eigenvalues of  $\mathbf{A}$ , counted with multiplicity, see Def. 7.1.1,

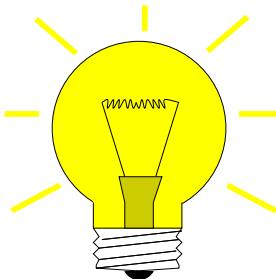
$\mathbf{u}_1, \dots, \mathbf{u}_n \hat{=}$  corresponding orthonormal eigenvectors, cf. Cor. 7.1.9.

$\blacktriangleright \mathbf{AU} = \mathbf{DU}, \quad \mathbf{U} = (\mathbf{u}_1, \dots, \mathbf{u}_n) \in \mathbb{R}^{n,n}, \quad \mathbf{D} = \text{diag}(\lambda_1, \dots, \lambda_n).$

We recall

- \* the direct power method (7.3.12) from Section 7.3.1
- \* and the inverse iteration from Section 7.3.2

and how they produce sequences  $(\mathbf{z}^{(k)})_{k \in \mathbb{N}_0}$  of vectors that are supposed to converge to a vector  $\in \text{Eig}\mathbf{A}\lambda_1$  or  $\in \text{Eig}\mathbf{A}\lambda_n$ , respectively.



Idea: Better  $\mathbf{z}^{(k)}$  from Ritz projection onto  $V := \text{Span}\{\mathbf{z}^{(0)}, \dots, \mathbf{z}^{(k)}\}$   
(= space spanned by previous iterates)

Recall ( $\rightarrow$  Code 7.3.104) **Ritz projection** of an EVP  $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$  onto a subspace  $V := \text{Span}\{\mathbf{v}_1, \dots, \mathbf{v}_m\}$ ,  $m < n \Rightarrow$  smaller  $m \times m$  generalized EVP

$$\underbrace{\mathbf{V}^T \mathbf{A} \mathbf{V}}_{:= \mathbf{H}} \mathbf{V} \mathbf{x} = \lambda \mathbf{V}^T \mathbf{V} \mathbf{x}, \quad \mathbf{V} := (\mathbf{v}_1, \dots, \mathbf{v}_m) \in \mathbb{R}^{n,m}. \quad (7.4.1)$$

From Rayleigh quotient Thm. 7.3.39 and considerations in Section 7.3.4.2:

$$\mathbf{u}_n \in V \Rightarrow \text{largest eigenvalue of (7.4.1)} = \lambda_{\max}(\mathbf{A}),$$

$$\mathbf{u}_1 \in V \Rightarrow \text{smallest eigenvalue of (7.4.1)} = \lambda_{\min}(\mathbf{A}) .$$

Intuition: If  $\mathbf{u}_n(\mathbf{u}_1)$  “well captured” by  $V$  (that is, the angle between the vector and the space  $V$  is small), then we can expect that the largest (smallest) eigenvalue of (7.4.1) is a good approximation for  $\lambda_{\max}(\mathbf{A})(\lambda_{\min}(\mathbf{A}))$ , and that, assuming normalization

$$\mathbf{Vw} \approx \mathbf{u}_1 \quad (\text{or } \mathbf{Vw} \approx \mathbf{u}_n) ,$$

where  $\mathbf{w}$  is the corresponding eigenvector of (7.4.1).

► For direct power method (7.3.12):

$$\mathbf{z}^{(k)} || \mathbf{A}^k \mathbf{z}^{(0)}$$

$$V = \text{Span}\{\mathbf{z}^{(0)}, \mathbf{A}\mathbf{z}^{(0)}, \dots, \mathbf{A}^{(k)}\mathbf{z}^{(0)}\} = \mathcal{K}_{k+1}(\mathbf{A}, \mathbf{z}^{(0)}) \quad \text{a Krylov space, } \rightarrow \text{Def. 8.2.6 .} \quad (7.4.2)$$

### MATLAB-code 7.4.3: Ritz projections onto Krylov space (7.4.2)

```

1 function [V,D] = kryleig(A,m)
2 % Ritz projection onto Krylov subspace. An
   orthonormal basis of  $\mathcal{K}_m(\mathbf{A}, \mathbf{1})$  is assembled
   into the columns of  $\mathbf{V}$ .
3 n = size(A,1); V = (1:n)'; V =
   V/norm(V);
4 for l=1:m-1
5   V = [V,A*V(:,end)]; [Q,R] = qr(V,0);
6   [W,D] = eig(Q'*A*Q); V = Q*W;
7 end
```

► direct power method with Ritz projection onto Krylov space from (7.4.2), cf. § 7.3.106.

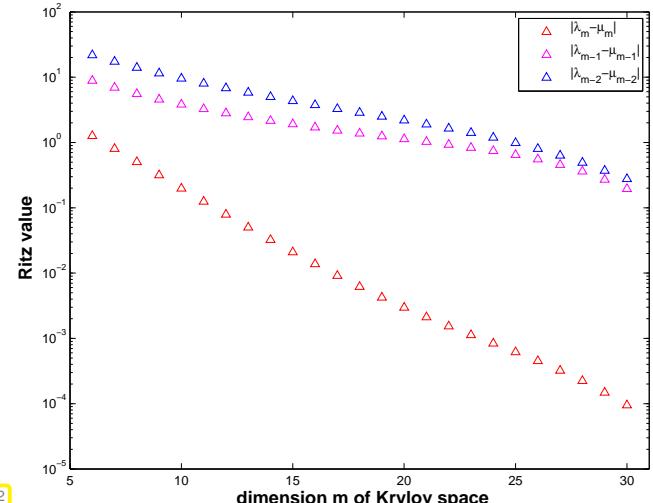
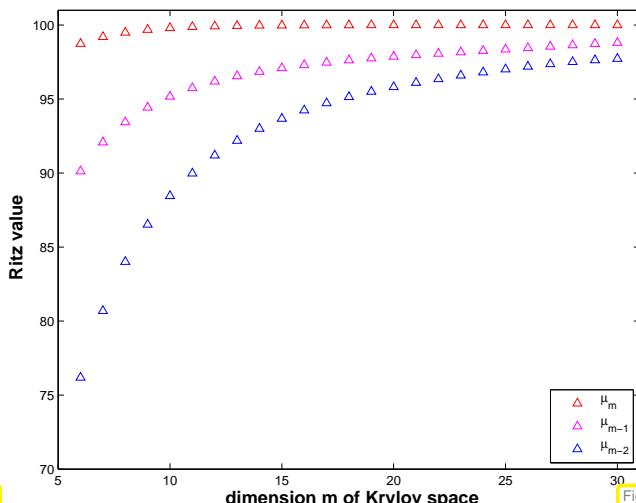
Note: implementation for demonstration purposes only (inefficient for sparse matrix  $\mathbf{A}$ !)

### Example 7.4.4 (Ritz projections onto Krylov space)

#### MATLAB-code 7.4.5:

```

1 n=100;
2 M=gallery('tridiag',-0.5*ones(n-1,1),2*ones(n,1),-1.5*ones(n-1,1));
3 [Q,R]=qr(M); A=Q'*diag(1:n)*Q; % synthetic matrix,
    $\sigma(\mathbf{A}) = \{1, 2, 3, \dots, 100\}$ 
```



Observation: “vaguely linear” convergence of largest Ritz values (notation  $\mu_i$ ) to largest eigenvalues.  
Fastest convergence of largest Ritz value → largest eigenvalue of  $\mathbf{A}$

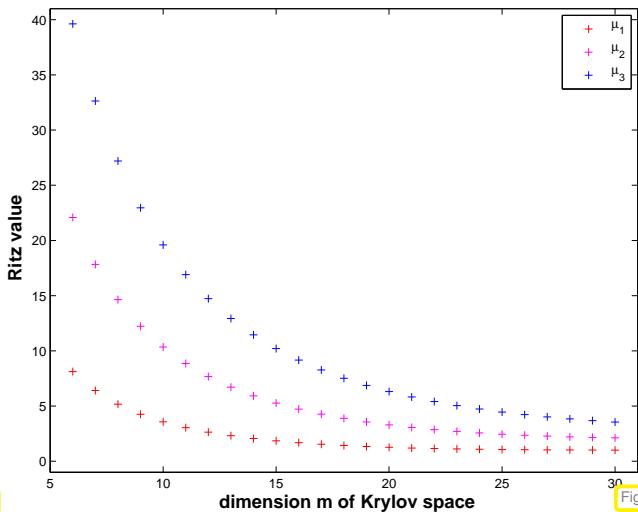


Fig. 243

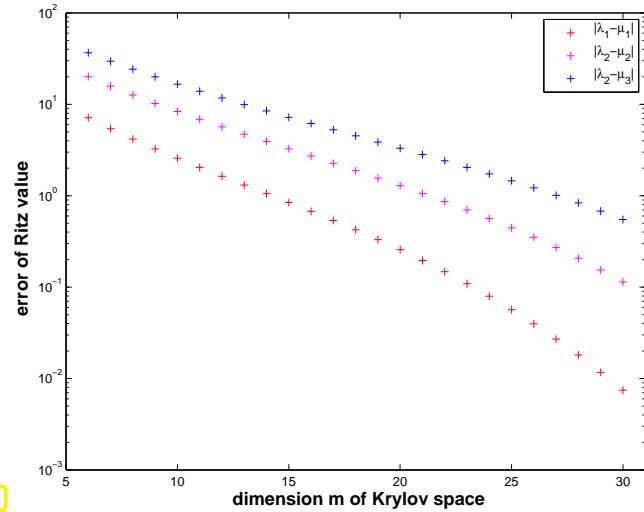


Fig. 244

Observation: *Also the smallest Ritz values converge “vaguely linearly” to the smallest eigenvalues of  $\mathbf{A}$ .*  
Fastest convergence of smallest Ritz value → smallest eigenvalue of  $\mathbf{A}$ .



Why do smallest Ritz values converge to smallest eigenvalues of  $\mathbf{A}$ ?

Consider direct power method (7.3.12) for  $\tilde{\mathbf{A}} := \nu\mathbf{I} - \mathbf{A}$ ,  $\nu > \lambda_{\max}(\mathbf{A})$ :

$$\mathbf{z}^{(0)} \text{ arbitrary}, \quad \tilde{\mathbf{z}}^{(k+1)} = \frac{(\nu\mathbf{I} - \mathbf{A})\tilde{\mathbf{z}}^{(k)}}{\|(\nu\mathbf{I} - \mathbf{A})\tilde{\mathbf{z}}^{(k)}\|_2} \quad (7.4.6)$$

As  $\sigma(\nu\mathbf{I} - \mathbf{A}) = \nu - \sigma(\mathbf{A})$  and eigenspaces agree, we infer from Thm. 7.3.21

$$\lambda_1 < \lambda_2 \Rightarrow \mathbf{z}^{(k)} \xrightarrow{k \rightarrow \infty} \mathbf{u}_1 \quad \& \quad \rho_{\mathbf{A}}(\mathbf{z}^{(k)}) \xrightarrow{k \rightarrow \infty} \lambda_1 \quad \text{linearly}. \quad (7.4.7)$$

By the binomial theorem (also applies to matrices, if they commute)

$$(\nu\mathbf{I} - \mathbf{A})^k = \sum_{j=0}^k \binom{k}{j} \nu^{k-j} \mathbf{A}^j \Rightarrow (\nu\mathbf{I} - \mathbf{A})^k \tilde{\mathbf{z}}^{(0)} \in \mathcal{K}_k(\mathbf{A}, \mathbf{z}^{(0)}),$$

$$\mathcal{K}_k(\nu\mathbf{I} - \mathbf{A}, \mathbf{x}) = \mathcal{K}_k(\mathbf{A}, \mathbf{x}). \quad (7.4.8)$$

➤  $\mathbf{u}_1$  can also be expected to be “well captured” by  $\mathcal{K}_k(\mathbf{A}, \mathbf{x})$  and the smallest Ritz value should provide a good approximation for  $\lambda_{\min}(\mathbf{A})$ .

Recall from Section 8.2.2 Lemma 8.2.12 :

Residuals  $\mathbf{r}_0, \dots, \mathbf{r}_{m-1}$  generated in CG iteration, § 8.2.17 applied to  $\mathbf{Ax} = \mathbf{z}$  with  $\mathbf{x}^{(0)} = 0$ , provide orthogonal basis for  $\mathcal{K}_m(\mathbf{A}, \mathbf{z})$  (, if  $\mathbf{r}_k \neq 0$ ).

► Inexpensive Ritz projection of  $\mathbf{Ax} = \lambda \mathbf{x}$  onto  $\mathcal{K}_m(\mathbf{A}, \mathbf{z})$ :  
 $\mathbf{V}_m^T \mathbf{AV}_m \mathbf{x} = \lambda \mathbf{x}$ ,  $\mathbf{V}_m := \left( \frac{\mathbf{r}_0}{\|\mathbf{r}_0\|}, \dots, \frac{\mathbf{r}_{m-1}}{\|\mathbf{r}_{m-1}\|} \right) \in \mathbb{R}^{n,m}$ . orthogonal matrix (7.4.9)

recall: residuals generated by *short recursions*, see § 8.2.17

### Lemma 7.4.10. Tridiagonal Ritz projection from CG residuals

$\mathbf{V}_m^T \mathbf{AV}_m$  is a tridiagonal matrix.

*Proof.* Lemma 8.2.12:  $\{\mathbf{r}_0, \dots, \mathbf{r}_{\ell-1}\}$  is an orthogonal basis of  $\mathcal{K}_\ell(\mathbf{A}, \mathbf{r}_0)$ , if all the residuals are non-zero. As  $\mathbf{A}\mathcal{K}_{\ell-1}(\mathbf{A}, \mathbf{r}_0) \subset \mathcal{K}_\ell(\mathbf{A}, \mathbf{r}_0)$ , we conclude the orthogonality  $\mathbf{r}_m^T \mathbf{A} \mathbf{r}_j$  for all  $j = 0, \dots, m-2$ . Since

$$(\mathbf{V}_m^T \mathbf{AV}_m)_{ij} = \mathbf{r}_{i-1}^T \mathbf{A} \mathbf{r}_{j-1}, \quad 1 \leq i, j \leq m,$$

the assertion of the theorem follows. □

$$\mathbf{V}_l^H \mathbf{AV}_l = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \\ & & & \ddots & \beta_{k-1} \\ & & & & \alpha_k \end{bmatrix} =: \mathbf{T}_l \in \mathbb{K}^{k,k} \quad [\text{tridiagonal matrix}] \quad (7.4.11)$$

Algorithm for computing  $\mathbf{V}_l$  and  $\mathbf{T}_l$ :  
**Lanczos process**

Computational effort/step:

- 1 ×  $\mathbf{A} \times \text{vector}$
- 2 dot products
- 2 AXPY-operations
- 1 division

Closely related to CG iteration,  
 § 8.2.17, Code 8.2.18.

### MATLAB-code 7.4.12: Lanczos process, cf. Code 8.2.18

```

1 function [V,alph,bet] = lanczos(A,k,z0)
2 % Note: this implementation of the Lanczos
3 % process also records the orthonormal CG
4 % residuals in the columns of the matrix V,
5 % which is not needed when only eigenvalue
6 % approximations are desired.
7 V = z0/norm(z0);
8 % Vectors storing entries of tridiagonal matrix
9 alph=zeros(k,1); bet = zeros(k,1);
10 for j=1:k
11 q = A*V(:,j); alph(j) = dot(q,V(:,j));
12 w = q - alph(j)*V(:,j);
13 if (j > 1), w = w - bet(j-1)*V(:,j-1);
14 end;
15 bet(j) = norm(w); V = [V,w/bet(j)];
16 end
17 bet = bet(1:end-1);

```

Total computational effort for  $l$  steps of Lanczos process, if  $\mathbf{A}$  has at most  $k$  non-zero entries per row:  
 $O(nkl)$

Note: Code 7.4.12 assumes that no residual vanishes. This could happen, if  $\mathbf{z}_0$  exactly belonged to the span of a few eigenvectors. However, in practical computations inevitable round-off errors will always ensure that the iterates do not stay in an invariant subspace of  $\mathbf{A}$ , cf. Rem. 7.3.22.

Convergence (what we expect from the above considerations) → [19, Sect. 8.5])

$$\text{In } l\text{-th step: } \lambda_n \approx \mu_l^{(l)}, \lambda_{n-1} \approx \mu_{l-1}^{(l)}, \dots, \lambda_1 \approx \mu_1^{(l)}, \\ \sigma(\mathbf{T}_l) = \{\mu_1^{(l)}, \dots, \mu_l^{(l)}\}, \quad \mu_1^{(l)} \leq \mu_2^{(l)} \leq \dots \leq \mu_l^{(l)}.$$

### Example 7.4.13 (Lanczos process for eigenvalue computation)

$\mathbf{A}$  from Ex. 7.4.4

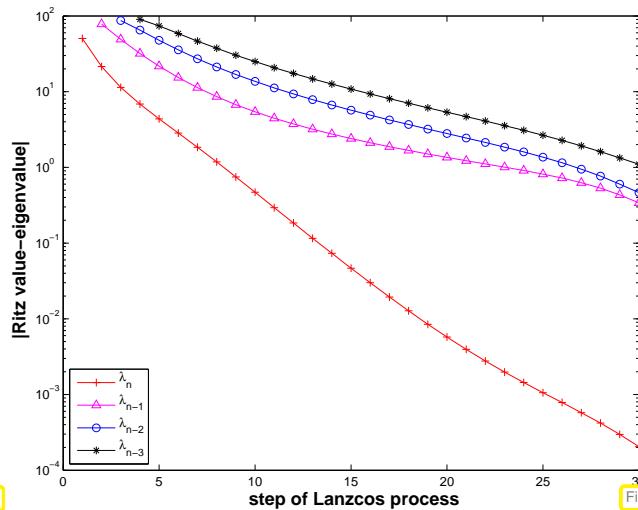
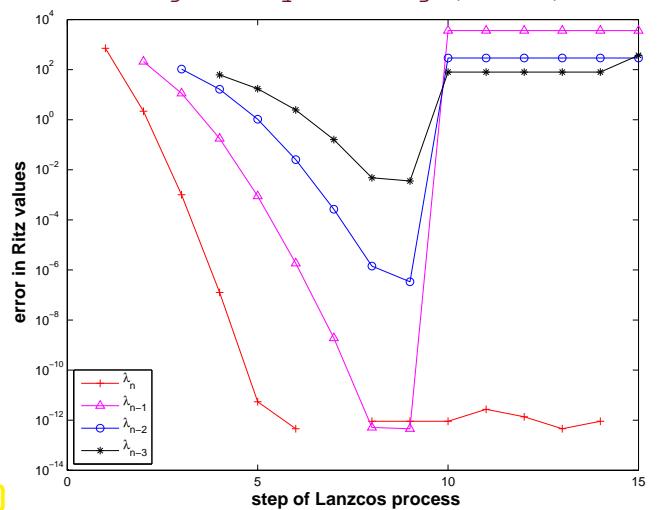


Fig. 245

$\mathbf{A} = \text{gallery('minij', 100);}$



Observation: same as in Ex. 7.4.4, linear convergence of Ritz values to eigenvalues.

However for  $\mathbf{A} \in \mathbb{R}^{10,10}$ ,  $a_{ij} = \min\{i, j\}$  good initial convergence, but sudden “jump” of Ritz values off eigenvalues!

Conjecture: Impact of roundoff errors, cf. Ex. 8.2.21

### Example 7.4.14 (Impact of roundoff on Lanczos process)

$$\mathbf{A} \in \mathbb{R}^{10,10} \quad , \quad a_{ij} = \min\{i,j\} . \quad \text{A} = \text{gallery('minij', 10);}$$

Computed by [V, alpha, beta] = lanczos(A, n, ones(n, 1)); see Code 7.4.12:

$$\mathbf{T} = \begin{pmatrix} 38.500000 & 14.813845 & & & & & & & & \\ 14.813845 & 9.642857 & 2.062955 & & & & & & & \\ & 2.062955 & 2.720779 & 0.776284 & & & & & & \\ & & 0.776284 & 1.336364 & 0.385013 & & & & & \\ & & & 0.385013 & 0.826316 & 0.215431 & & & & \\ & & & & 0.215431 & 0.582380 & 0.126781 & & & \\ & & & & & 0.126781 & 0.446860 & 0.074650 & & \\ & & & & & & 0.074650 & 0.363803 & 0.043121 & & \\ & & & & & & & 0.043121 & 3.820888 & 11.991094 & \\ & & & & & & & & 11.991094 & 41.254286 \end{pmatrix}$$

$$\sigma(\mathbf{A}) = \{0.255680, 0.273787, 0.307979, 0.366209, 0.465233, 0.643104, 1.000000, 1.873023, 5.048917, 44.766069\}$$

$$\sigma(\mathbf{T}) = \{0.263867, 0.303001, 0.365376, 0.465199, 0.643104, 1.000000, 1.873023, 5.048917, 44.765976, 44.766069\}$$

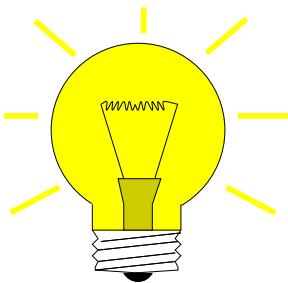
► Uncanny cluster of computed eigenvalues of  $\mathbf{T}$  (“ghost eigenvalues”, [28, Sect. 9.2.5])

$$\mathbf{V}^H \mathbf{V} = \begin{pmatrix} 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000251 & 0.258801 & 0.883711 \\ 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000106 & 0.109470 & 0.373799 \\ 0.000000 & -0.000000 & 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000005 & 0.005373 & 0.018347 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000096 & 0.000032 & \\ 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & 0.000000 & 0.000000 & 0.000001 & 0.000003 & \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 & \\ 0.000251 & 0.000106 & 0.000005 & 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 \\ 0.258801 & 0.109470 & 0.005373 & 0.000096 & 0.000001 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & 0.000000 \\ 0.883711 & 0.373799 & 0.018347 & 0.000328 & 0.000003 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{pmatrix}$$

► Loss of orthogonality of residual vectors due to roundoff

(compare: impact of roundoff on CG iteration, Ex. 8.2.21)

$l$	$\sigma(\mathbf{T}_l)$								
1	38.500000								
2	3.392123 44.750734								
3	1.117692 4.979881 44.766064								
4	0.597664 1.788008 5.048259 44.766069								
5	0.415715 0.925441 1.870175 5.048916 44.766069								
6	0.336507 0.588906 0.995299 1.872997 5.048917 44.766069								
7	0.297303 0.431779 0.638542 0.999922 1.873023 5.048917 44.766069								
8	0.276160 0.349724 0.462449 0.643016 1.000000 1.873023 5.048917 44.766069								
9	0.276035 0.349451 0.462320 0.643006 1.000000 1.873023 3.821426 5.048917 44.766069								
10	0.263867 0.303001 0.365376 0.465199 0.643104 1.000000 1.873023 5.048917 44.765976 44.766069								



Idea:

- \* do not rely on orthogonality relations of Lemma 8.2.12
- \* use explicit **Gram-Schmidt orthogonalization** [59, Thm. 4.8], [34, Alg .6.1]

Details: inductive approach: given  $\{\mathbf{v}_1, \dots, \mathbf{v}_l\}$  ONB of  $\mathcal{K}_l(\mathbf{A}, \mathbf{z})$ 

$$\blacktriangleright \quad \tilde{\mathbf{v}}_{l+1} := \mathbf{A}\mathbf{v}_l - \sum_{j=1}^l (\mathbf{v}_j^H \mathbf{A}\mathbf{v}_l) \mathbf{v}_j, \quad \mathbf{v}_{l+1} := \frac{\tilde{\mathbf{v}}_{l+1}}{\|\tilde{\mathbf{v}}_{l+1}\|_2} \Rightarrow \mathbf{v}_{l+1} \perp \mathcal{K}_l(\mathbf{A}, \mathbf{z}). \quad (7.4.15)$$

(Gram-Schmidt, cf. (8.2.11))

orthogonal

- $\blacktriangleright$  **Arnoldi process:** In step  $l$ :
- |            |                            |
|------------|----------------------------|
| $1 \times$ | $\mathbf{A} \times$ vector |
| $l+1$      | dot products               |
| $l$        | AXPY-operations            |
| $n$        | divisions                  |

$\geq$  Computational cost for  $l$  steps, if at most  $k$  non-zero entries in each row of  $\mathbf{A}$ :  $O(nkl^2)$

#### MATLAB-code 7.4.16: Arnoldi process

```

1 function [V,H] = arnoldi(A,k,v0)
2 % Columns of V store orthonormal basis of Krylov spaces  $\mathcal{K}_l(\mathbf{A}, \mathbf{v}_0)$ .
3 % H returns Hessenberg matrix, see Lemma 7.4.17.
4 V = [v0/norm(v0)];
5 H = zeros(k+1,k);
6 for l=1:k
7   vt = A*V(:,l); % "power iteration", next basis vector
8   for j=1:l
9     % Gram-Schmidt orthogonalization, cf. Sect. 7.3.4.1
10    H(j,l) = dot(V(:,j),vt);
11    vt = vt - H(j,l)*V(:,j);
12  end
13  H(l+1,l) = norm(vt);
14  if (H(l+1,l) == 0), break; end % "theoretical" termination
15  V = [V, vt/H(l+1,l)];
16 end

```

If it does not stop prematurely, the Arnoldi process of Code 7.4.16 will yield an *orthonormal basis* (ONB) of  $\mathcal{K}_{k+1}(\mathbf{A}, \mathbf{v}_0)$  for a **general**  $\mathbf{A} \in \mathbb{C}^{n,n}$ .

Algebraic view of the Arnoldi process of Code 7.4.16, meaning of output H:

$$\mathbf{V}_l = [\mathbf{v}_1, \dots, \mathbf{v}_l] : \quad \mathbf{A}\mathbf{V}_l = \mathbf{V}_{l+1}\tilde{\mathbf{H}}_l \quad , \quad \tilde{\mathbf{H}}_l \in \mathbb{K}^{l+1,l} \text{ mit } \tilde{h}_{ij} = \begin{cases} \mathbf{v}_i^H \mathbf{A} \mathbf{v}_j & , \text{if } i \leq j , \\ \|\tilde{\mathbf{v}}_i\|_2 & , \text{if } i = j + 1 , \\ 0 & \text{else.} \end{cases}$$

►  $\tilde{\mathbf{H}}_l$  = non-square upper Hessenberg matrices

$$\left( \begin{array}{c|c} \mathbf{A} & \\ \hline & \end{array} \right) \left( \begin{array}{c|c|c|c} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_l \end{array} \right) = \left( \begin{array}{c|c|c|c} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_l \\ & & & \text{red} \\ & & & 1+\epsilon \end{array} \right) \left( \begin{array}{c|c} \mathbf{H}_l & \\ \hline & \end{array} \right)$$

Translate Code 7.4.16 to matrix calculus:

#### Lemma 7.4.17. Theory of Arnoldi process

For the matrices  $\mathbf{V}_l \in \mathbb{K}^{n,l}$ ,  $\tilde{\mathbf{H}}_l \in \mathbb{K}^{l+1,l}$  arising in the  $l$ -th step,  $l \leq n$ , of the Arnoldi process holds

- (i)  $\mathbf{V}_l^H \mathbf{V}_l = \mathbf{I}$  (unitary matrix),
- (ii)  $\mathbf{A} \mathbf{V}_l = \mathbf{V}_{l+1} \tilde{\mathbf{H}}_l$ ,  $\tilde{\mathbf{H}}_l$  is non-square upper Hessenberg matrix,
- (iii)  $\mathbf{V}_l^H \mathbf{A} \mathbf{V}_l = \mathbf{H}_l \in \mathbb{K}^{l,l}$ ,  $h_{ij} = \tilde{h}_{ij}$  for  $1 \leq i, j \leq l$ ,
- (iv) If  $\mathbf{A} = \mathbf{A}^H$  then  $\mathbf{H}_l$  is tridiagonal ( $\Rightarrow$  Lanczos process)

*Proof.* Direct from Gram-Schmidt orthogonalization and inspection of Code 7.4.16.  $\square$

#### Remark 7.4.18 (Arnoldi process and Ritz projection)

Interpretation of Lemma 7.4.17 (iii) & (i):

$\mathbf{H}_l \mathbf{x} = \lambda \mathbf{x}$  is a (generalized) Ritz projection of EVP  $\mathbf{A} \mathbf{x} = \lambda \mathbf{x}$ , cf. Section 7.3.4.2.

► Eigenvalue approximation for general EVP  $\mathbf{A} \mathbf{x} = \lambda \mathbf{x}$  by Arnoldi process:

**MATLAB-code 7.4.19: Arnoldi eigenvalue approximation**

```

1 function [dn,V,Ht] =
2     arnoldieig(A,v0,k,tol)
3 n = size(A,1); V = [v0/norm(v0)];
4 H = zeros(1,0); dn = zeros(k,1);
5 for l=1:n
6     d = dn;
7     Ht = [Ht, zeros(l,1); zeros(1,l)];
8     vt = A*V(:,l);
9     for j=1:l
10         Ht(j,l) = dot(V(:,j),vt);
11         vt = vt - Ht(j,l)*V(:,j);
12     end
13     ev = sort(eig(Ht(1:l,1:l)));
14     dn(1:min(l,k)) =
15         ev(end:-1:end-min(l,k)+1);
16     if (norm(d-dn) <
17         tol*norm(dn)) &\pnodenode{SPOWITX} &,
18         break; end;
19     Ht(l+1,l) = norm(vt);
20     V = [V, vt/Ht(l+1,l)];
21 end

```

**MATLAB-code 7.4.20: MATLAB-CODE Arnoldi eigenvalue approximation**

```

function [dn,V,Ht] =
arnoldieig(A,v0,k,tol)
n = size(A,1); V = [v0/norm(v0)];
H = zeros(1,0); dn = zeros(k,1);
for l=1:n
    d = dn;
    Ht = [Ht, zeros(l,1); zeros(1,l)];
    vt = A*V(:,l);
    for j=1:l
        Ht(j,l) = dot(V(:,j),vt);
        vt = vt - Ht(j,l)*V(:,j);

    end
    ev = sort(eig(Ht(1:l,1:l)));
    dn(1:min(l,k)) =
        ev(end:-1:end-min(l,k)+1);
    if (norm(d-dn) < tol*norm(dn)), break;
    end;
    Ht(l+1,l) = norm(vt);
    V = [V, vt/Ht(l+1,l)];
end

```

Heuristic termination criterion

Arnoldi process for computing the  $k$  largest (in modulus) eigenvalues of  $\mathbf{A} \in \mathbb{C}^{n,n}$

1  $\mathbf{A} \times$  vector per step  
 ( $\Rightarrow$  attractive for sparse matrices)

However: required storage increases with number of steps,  
*cf.* situation with GMRES, Section 8.4.1.

Heuristic termination criterion

### Example 7.4.21 (Stability of Arnoldi process)

$$\mathbf{A} \in \mathbb{R}^{100,100}, \quad a_{ij} = \min\{i,j\}.$$

`A = gallery('minij', 100);`

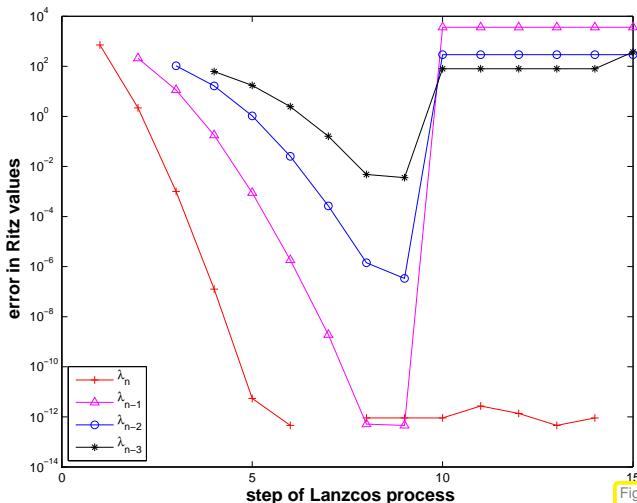


Fig. 247

Lanczos process: Ritz values

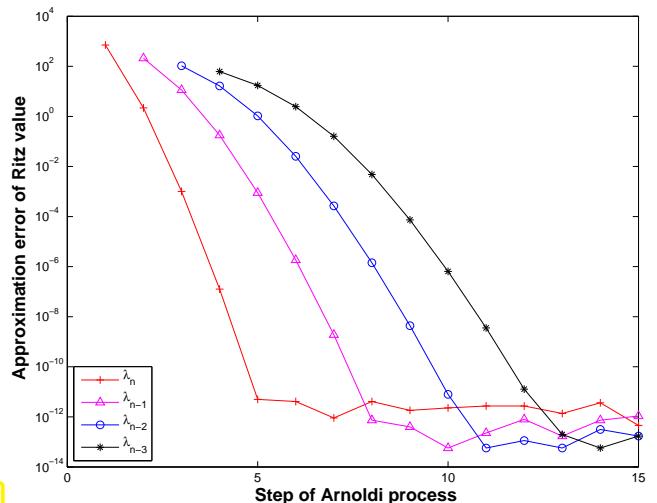


Fig. 248

Arnoldi process: Ritz values

Ritz values during Arnoldi process for `A = gallery('minij', 10);`  $\leftrightarrow$  Ex. 7.4.13

$l$	$\sigma(\mathbf{H}_l)$							
1								
2								38.500000
3								3.392123 44.750734
4								1.117692 4.979881 44.766064
5					0.415715	0.925441	1.870175	5.048916 44.766069
6				0.336507	0.588906	0.995299	1.872997	5.048917 44.766069
7			0.297303	0.431779	0.638542	0.999922	1.873023	5.048917 44.766069
8		0.276159	0.349722	0.462449	0.643016	1.000000	1.873023	5.048917 44.766069
9	0.263872	0.303009	0.365379	0.465199	0.643104	1.000000	1.873023	5.048917 44.766069
10	0.255680	0.273787	0.307979	0.366209	0.465233	0.643104	1.000000	1.873023 5.048917 44.766069

Observation: (almost perfect approximation of spectrum of  $\mathbf{A}$ )

For the above examples both the Arnoldi process and the Lanczos process are *algebraically equivalent*, because they are applied to a symmetric matrix  $\mathbf{A} = \mathbf{A}^T$ . However, they behave strikingly differently, which indicates that they are *not numerically equivalent*.

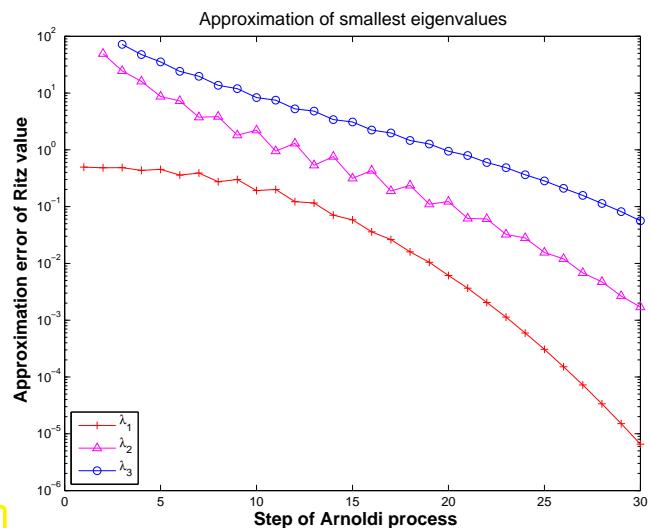
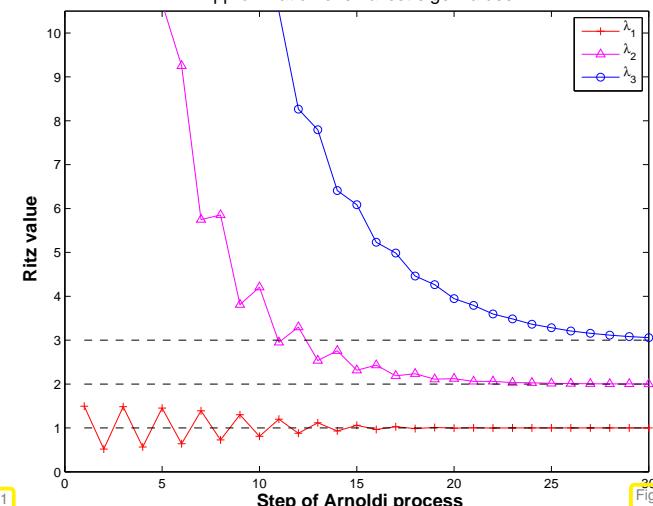
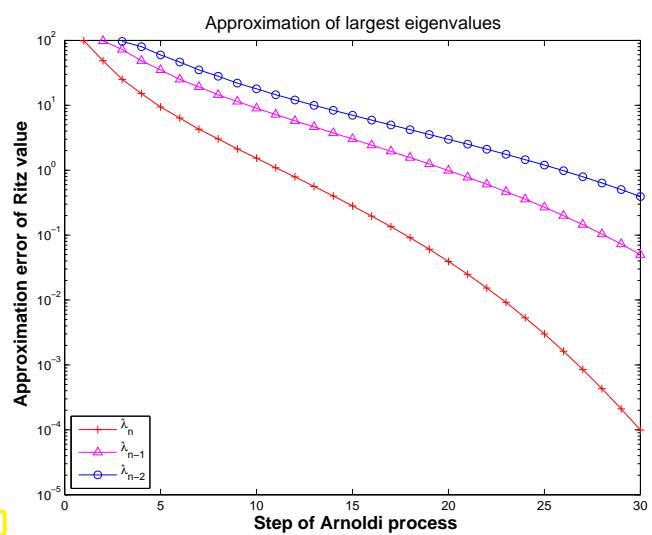
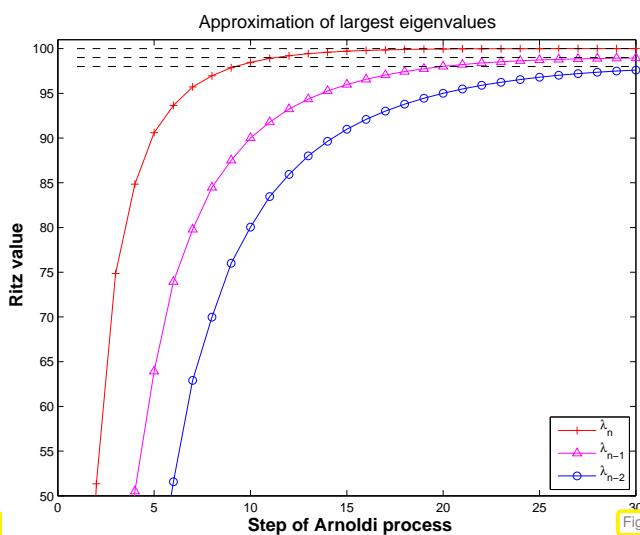
The Arnoldi process is much less affected by roundoff than the Lanczos process, because it does not take for granted orthogonality of the “residual vector sequence”. Hence, the Arnoldi process enjoys superior numerical stability ( $\rightarrow ??$ , Def. 1.5.80) compared to the Lanczos process.

### Example 7.4.22 (Eigenvalue computation with Arnoldi process)

Eigenvalue approximation from Arnoldi process for *non-symmetric*  $\mathbf{A}$ , initial vector `ones(100, 1)`;

#### MATLAB-code 7.4.23:

```
1 n=100;
2 M=full(gallery('tridiag', -0.5*ones(n-1,1), 2*ones(n,1), -1.5*ones(n-1,1)));
3 A=M*diag(1:n)*inv(M);
```



Observation: “vaguely linear” convergence of largest and smallest eigenvalues, cf. Ex. 7.4.4.

Krylov subspace iteration methods (= Arnoldi process, Lanczos process) attractive for computing a few of the largest/smallest eigenvalues and associated eigenvectors of *large sparse matrices*.

#### Remark 7.4.24 (Krylov subspace methods for generalized EVP)

Adaptation of Krylov subspace iterative eigensolvers to generalized EVP:  $\mathbf{Ax} = \lambda \mathbf{Bx}$ ,  $\mathbf{B}$  s.p.d.: replace Euclidean inner product with “ $\mathbf{B}$ -inner product”  $(\mathbf{x}, \mathbf{y}) \mapsto \mathbf{x}^H \mathbf{B} \mathbf{y}$ .

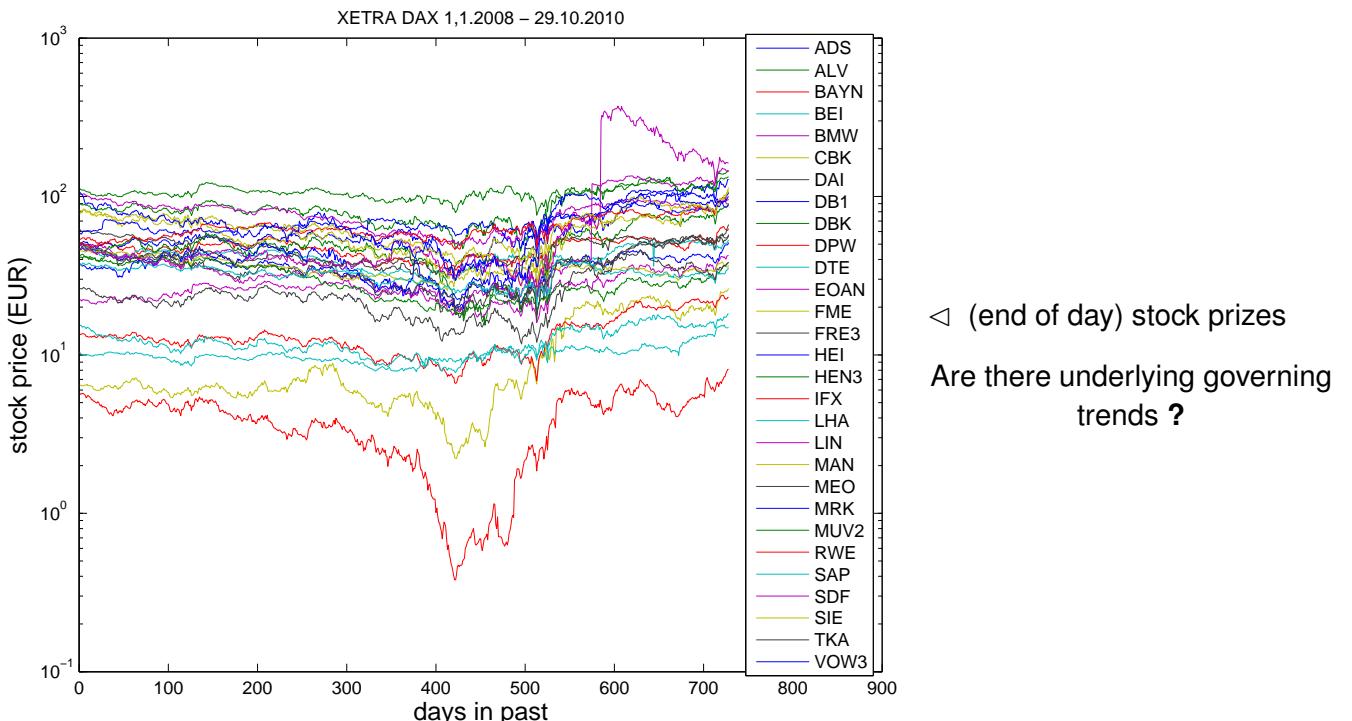
MATLAB-functions:

```
d = eigs(A,k,sigma) : k largest/smallest eigenvalues of A
d = eigs(A,B,k,sigma) : k largest/smallest eigenvalues for generalized EVP Ax = λBx, B
                           s.p.d.
d = eigs(Afun,n,k)   : Afun = handle to function providing matrix×vector for A/A⁻¹/A -
                           αI/(A - αB)⁻¹. (Use flags to tell eigs about special properties of
                           matrix behind Afun.)
```

`eigs` just calls routines of the open source [ARPACK](#) numerical library.

## 7.5 Singular Value Decomposition

#### Example 7.5.1 (Trend analysis)



### Example 7.5.2 (Classification from measured data)

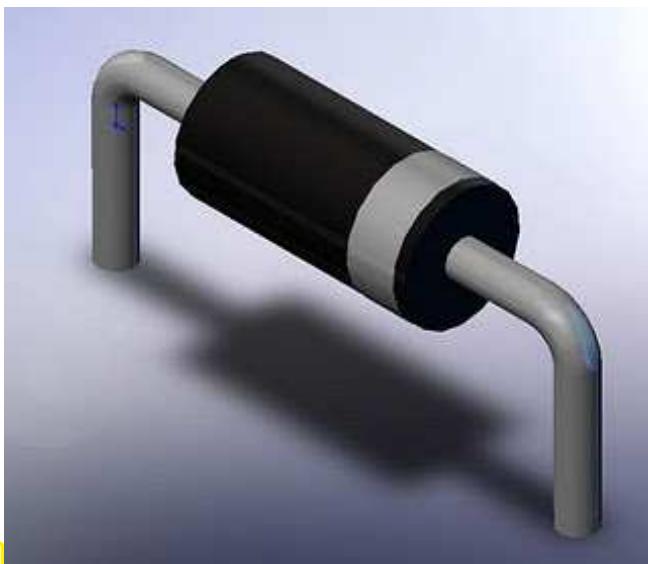
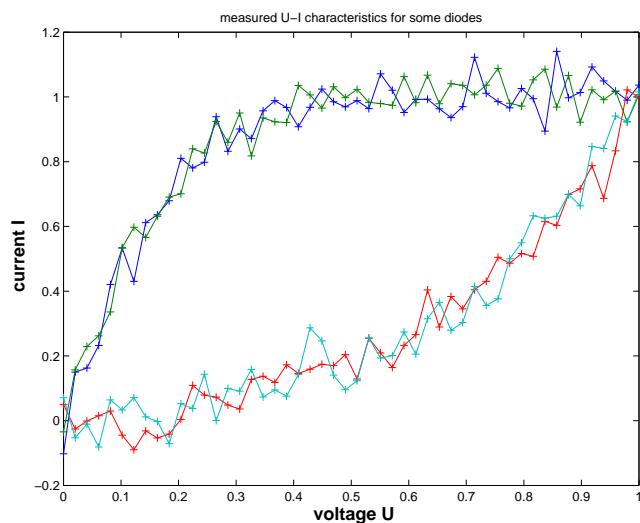


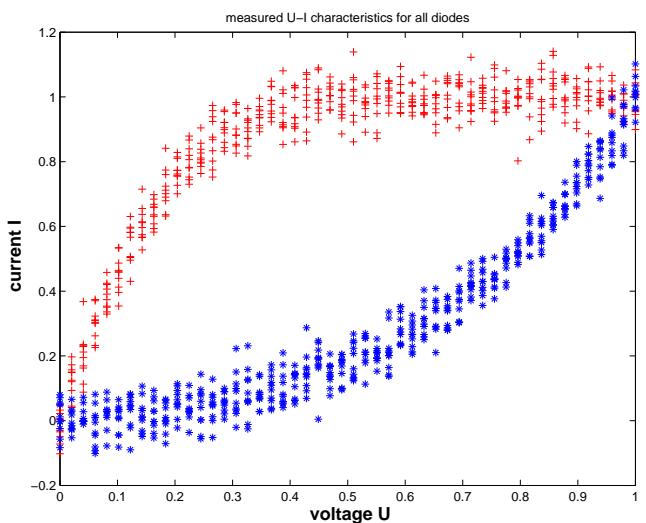
Fig. 254



Given: measured  $U$ - $I$  characteristics of diodes

Find out

- how many types,
- the  $U$ - $I$  characteristic of each type.



### Remark 7.5.3 (Principal component analysis (PCA))

Given:  $n$  data points  $\mathbf{a}_j \in \mathbb{R}^m$ ,  $j = 1, \dots, n$ , in  $m$ -dimensional (feature) space  
(e.g.,  $\mathbf{a}_j$  may represent a finite *time series* or a *measured relationship* of physical quantities)

In Ex. 7.5.1:  $n \hat{=} \text{number of stocks}$   
 $m \hat{=} \text{number of days, for which stock prices are recorded}$

- \* Extreme case: all stocks follow exactly *one* trend

$$\leftrightarrow \quad \mathbf{a}_j \in \text{Span}\{\mathbf{u}\} \quad \forall j = 1, \dots, n ,$$

for a **trend vector**  $\mathbf{u} \in \mathbb{R}^m$ ,  $\|\mathbf{u}\|_2 = 1$ .

- \* Unlikely case: all stocks prices are governed by  $p < n$  trends:

$$\leftrightarrow \quad \mathbf{a}_j \in \text{Span}\{\mathbf{u}_1, \dots, \mathbf{u}_p\} \quad \forall j = 1, \dots, m , \quad (7.5.4)$$

with **orthonormal trend vectors**  $\mathbf{u}_i \in \mathbb{R}^m$ ,  $i = 1, \dots, p$ .

Why orthonormal ? Trends should be as “independent as possible” (minimally correlated)

Perspective of linear algebra:

$$(7.5.4) \Leftrightarrow \text{rank}(\mathbf{A}) = p \text{ for } \mathbf{A} := (\mathbf{a}_1, \dots, \mathbf{a}_n) \in \mathbb{R}^{m,n}, \quad \text{Im}(\mathbf{A}) = \text{Span}\{\mathbf{u}_1, \dots, \mathbf{u}_p\} \quad (7.5.5)$$

- \* Realistic: stock prizes *approximately* follow a few trends

$$\mathbf{a}_j \in \text{Span}\{\mathbf{u}_1, \dots, \mathbf{u}_p\} + \text{“small perturbations”} \quad \forall j = 1, \dots, m,$$

with orthonormal trend vectors  $\mathbf{u}_i, i = 1, \dots, p$ .

Task (PCA): determine (minimal)  $p$  and orthonormal trend vectors  $\mathbf{u}_i, i = 1, \dots, p$

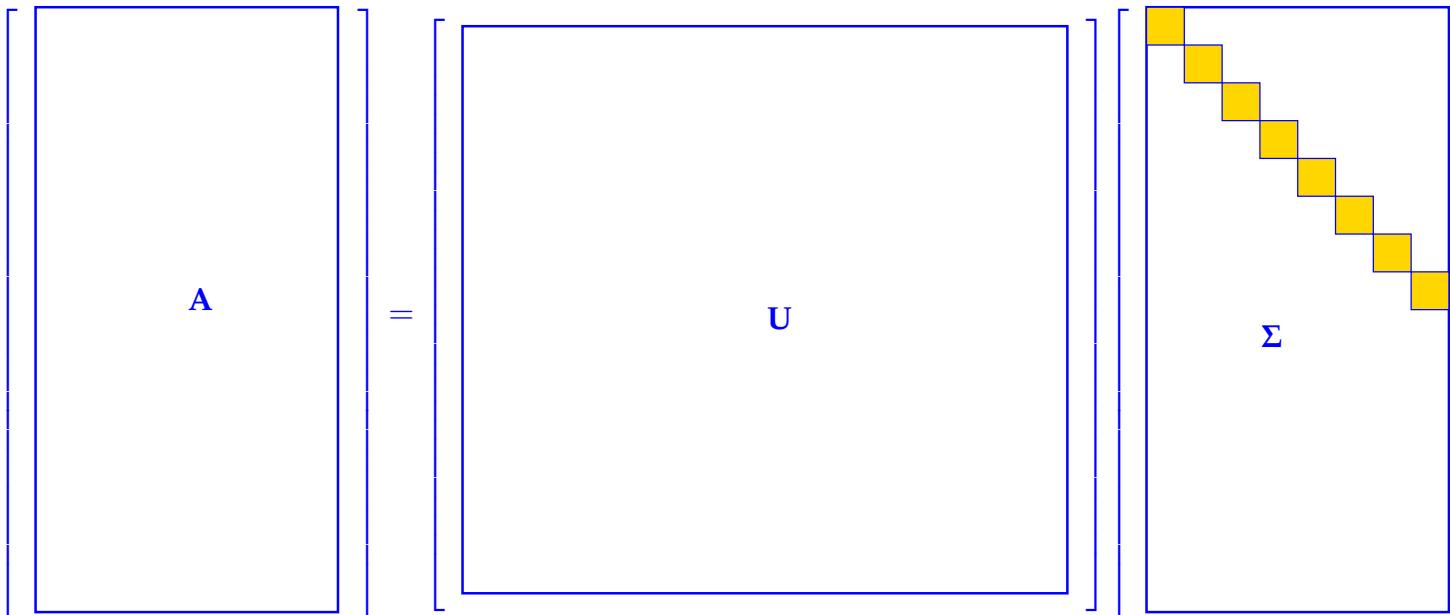
Issue: how to distinguish between trends and perturbations ?

**Theorem 7.5.6. singular value decomposition** → [59, Thm. 9.6], [34, Thm. 11.1]

For any  $\mathbf{A} \in \mathbb{K}^{m,n}$  there are unitary matrices  $\mathbf{U} \in \mathbb{K}^{m,m}$ ,  $\mathbf{V} \in \mathbb{K}^{n,n}$  and a (generalized) diagonal matrix  $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_p) \in \mathbb{R}^{m,n}$ ,  $p := \min\{m, n\}$ ,  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_p \geq 0$  such that

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^H.$$

(\*):  $\Sigma$  (generalized) diagonal matrix : $\Leftrightarrow$   $(\Sigma)_{i,j} = 0$ , if  $i \neq j$ ,  $1 \leq i \leq m$ ,  $1 \leq j \leq n$ .



$$\boxed{\mathbf{A}} = \boxed{\mathbf{U}} \boxed{\Sigma}$$

*Proof.* (of Thm. 7.5.6, by induction) [77, Thm. 4.2.3]: Continuous functions attain extremal values on compact sets (here the unit ball  $\{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x}\|_2 \leq 1\}$ )

$$\blacktriangleright \exists \mathbf{x} \in \mathbb{K}^n, \mathbf{y} \in \mathbb{K}^m, \|\mathbf{x}\| = \|\mathbf{y}\|_2 = 1 : \mathbf{Ax} = \sigma \mathbf{y}, \sigma = \|\mathbf{A}\|_2,$$

where we used the definition of the matrix 2-norm, see Def. 1.5.71. By Gram-Schmidt orthogonalization:  $\exists \tilde{\mathbf{V}} \in \mathbb{K}^{n,n-1}, \tilde{\mathbf{U}} \in \mathbb{K}^{m,m-1}$  such that

$$\mathbf{V} = (\mathbf{x} \ \tilde{\mathbf{V}}) \in \mathbb{K}^{n,n}, \quad \mathbf{U} = (\mathbf{y} \ \tilde{\mathbf{U}}) \in \mathbb{K}^{m,m} \text{ are unitary.}$$

$$\blacktriangleright \mathbf{U}^H \mathbf{A} \mathbf{V} = (\mathbf{y} \ \tilde{\mathbf{U}})^H \mathbf{A} (\mathbf{x} \ \tilde{\mathbf{V}}) = \begin{bmatrix} \mathbf{y}^H \mathbf{A} \mathbf{x} & \mathbf{y}^H \mathbf{A} \tilde{\mathbf{V}} \\ \tilde{\mathbf{U}}^H \mathbf{A} \mathbf{x} & \tilde{\mathbf{U}}^H \mathbf{A} \tilde{\mathbf{V}} \end{bmatrix} = \begin{bmatrix} \sigma & \mathbf{w}^H \\ 0 & \mathbf{B} \end{bmatrix} =: \mathbf{A}_1.$$

Since

$$\left\| \mathbf{A}_1 \begin{bmatrix} \sigma \\ \mathbf{w} \end{bmatrix} \right\|_2^2 = \left\| \begin{bmatrix} \sigma^2 + \mathbf{w}^H \mathbf{w} \\ \mathbf{B} \mathbf{w} \end{bmatrix} \right\|_2^2 = (\sigma^2 + \mathbf{w}^H \mathbf{w})^2 + \|\mathbf{B} \mathbf{w}\|_2^2 \geq (\sigma^2 + \mathbf{w}^H \mathbf{w})^2,$$

we conclude

$$\|\mathbf{A}_1\|_2^2 = \sup_{\mathbf{x} \neq 0} \frac{\|\mathbf{A}_1 \mathbf{x}\|_2^2}{\|\mathbf{x}\|_2^2} \geq \frac{\|\mathbf{A}_1 \begin{pmatrix} \sigma \\ \mathbf{w} \end{pmatrix}\|_2^2}{\|\begin{pmatrix} \sigma \\ \mathbf{w} \end{pmatrix}\|_2^2} \geq \frac{(\sigma^2 + \mathbf{w}^H \mathbf{w})^2}{\sigma^2 + \mathbf{w}^H \mathbf{w}} = \sigma^2 + \mathbf{w}^H \mathbf{w}. \quad (7.5.7)$$

$$\blacktriangleright \mathbf{A}_1 = \begin{bmatrix} \sigma & 0 \\ 0 & \mathbf{B} \end{bmatrix}.$$

Then apply induction argument to  $\mathbf{B}$ . □

### Definition 7.5.8. Singular value decomposition (SVD)

The decomposition  $\mathbf{A} = \mathbf{U} \Sigma \mathbf{V}^H$  of Thm. 7.5.6 is called **singular value decomposition (SVD)** of  $\mathbf{A}$ . The diagonal entries  $\sigma_i$  of  $\Sigma$  are the **singular values** of  $\mathbf{A}$ .

**Lemma 7.5.9.**

The squares  $\sigma_i^2$  of the non-zero singular values of  $\mathbf{A}$  are the non-zero eigenvalues of  $\mathbf{A}^H \mathbf{A}$ ,  $\mathbf{A} \mathbf{A}^H$  with associated eigenvectors  $(\mathbf{V})_{:,1}, \dots, (\mathbf{V})_{:,p}$ ,  $(\mathbf{U})_{:,1}, \dots, (\mathbf{U})_{:,p}$ , respectively.

*Proof.*  $\mathbf{A} \mathbf{A}^H$  and  $\mathbf{A}^H \mathbf{A}$  are similar ( $\rightarrow$  Lemma 7.1.6) to diagonal matrices with non-zero diagonal entries  $\sigma_i^2$  ( $\sigma_i \neq 0$ ), e.g.,

$$\mathbf{A} \mathbf{A}^H = \mathbf{U} \Sigma \mathbf{V}^H \mathbf{V} \Sigma^H \mathbf{U}^H = \mathbf{U} \underbrace{\Sigma \Sigma^H}_{\text{diagonal matrix}} \mathbf{U}^H.$$

□

□

**Remark 7.5.10 (SVD and additive rank-1 decomposition** → [34, Cor. 11.2], [59, Thm. 9.8])

Recall from linear algebra:

rank-1 matrices are tensor products of vectors

$$\mathbf{A} \in \mathbb{K}^{m,n} \quad \text{and} \quad \text{rank}(\mathbf{A}) = 1 \iff \exists \mathbf{u} \in \mathbb{K}^m, \mathbf{v} \in \mathbb{K}^n: \mathbf{A} = \mathbf{u} \mathbf{v}^H, \quad (7.5.11)$$

because  $\text{rank}(\mathbf{A}) = 1$  means that  $\mathbf{Ax} = \mu(\mathbf{x})\mathbf{u}$  for some  $\mathbf{u} \in \mathbb{K}^m$  and linear form  $\mathbf{x} \mapsto \mu(\mathbf{x})$ . By the Riesz representation theorem the latter can be written as  $\mu(\mathbf{x}) = \mathbf{v}^H \mathbf{x}$ .

► Singular value decomposition provides additive decomposition into rank-1 matrices:

$$\mathbf{A} = \mathbf{U} \Sigma \mathbf{V}^H = \sum_{j=1}^p \sigma_j (\mathbf{U})_{:,j} (\mathbf{V})_{:,j}^H. \quad (7.5.12)$$

**Remark 7.5.13 (Uniqueness of SVD)**

SVD of Def. 7.5.8 is not (necessarily) unique, but the singular values are.

*Proof.* Proof by contradiction: assume that  $\mathbf{A}$  has two singular value decompositions

$$\mathbf{A} = \mathbf{U}_1 \Sigma_1 \mathbf{V}_1^H = \mathbf{U}_2 \Sigma_2 \mathbf{V}_2^H \Rightarrow \mathbf{U}_1 \underbrace{\Sigma_1 \Sigma_1^H}_{=\text{diag}(s_1^1, \dots, s_m^1)} \mathbf{U}_1^H = \mathbf{A} \mathbf{A}^H = \mathbf{U}_2 \underbrace{\Sigma_2 \Sigma_2^H}_{=\text{diag}(s_1^2, \dots, s_m^2)} \mathbf{U}_2^H.$$

Two similar diagonal matrices with non-increasing diagonal entries are equal !

□

MATLAB-functions (for algorithms see [28, Sect. 8.3]):

$s = \text{svd}(\mathbf{A})$	: computes singular values of matrix $\mathbf{A}$
$[\mathbf{U}, \mathbf{S}, \mathbf{V}] = \text{svd}(\mathbf{A})$	: computes singular value decomposition according to Thm. 7.5.6
$[\mathbf{U}, \mathbf{S}, \mathbf{V}] = \text{svd}(\mathbf{A}, 0)$	: “economical” singular value decomposition for $m > n$ : : $\mathbf{U} \in \mathbb{K}^{m,n}$ , $\Sigma \in \mathbb{R}^{n,n}$ , $\mathbf{V} \in \mathbb{K}^{n,n}$
$s = \text{svds}(\mathbf{A}, k)$	: $k$ largest singular values (important for sparse $\mathbf{A} \rightarrow$ Notion 1.7.1)
$[\mathbf{U}, \mathbf{S}, \mathbf{V}] = \text{svds}(\mathbf{A}, k)$	: partial singular value decomposition: $\mathbf{U} \in \mathbb{K}^{m,k}$ , $\mathbf{V} \in \mathbb{K}^{n,k}$ , $\Sigma \in \mathbb{R}^{k,k}$ diagonal with $k$ largest singular values of $\mathbf{A}$ .

Explanation: “economical” singular value decomposition:

$$\begin{bmatrix} \mathbf{A} \end{bmatrix} = \begin{bmatrix} \mathbf{U} \end{bmatrix} \begin{bmatrix} \Sigma \end{bmatrix} \begin{bmatrix} \mathbf{V}^H \end{bmatrix}$$

(MATLAB) algorithm for computing SVD is (numerically) stable → Def. 1.5.80

Complexity:

$$\begin{aligned} & 2mn^2 + 2n^3 + O(n^2) + O(mn) \quad \text{for } \mathbf{s} = \text{svd}(\mathbf{A}), \\ & 4m^2n + 22n^3 + O(mn) + O(n^2) \quad \text{for } [\mathbf{U}, \mathbf{S}, \mathbf{V}] = \text{svd}(\mathbf{A}), \\ & O(mn^2) + O(n^3) \quad \text{for } [\mathbf{U}, \mathbf{S}, \mathbf{V}] = \text{svd}(\mathbf{A}, 0), m \gg n. \end{aligned}$$

- Application of SVD: computation of rank (→ Def. 1.6.9), kernel and range of a matrix

#### Lemma 7.5.14. SVD and rank of a matrix → [59, Cor. 9.7]

If the singular values of  $\mathbf{A}$  satisfy  $\sigma_1 \geq \dots \geq \sigma_r > \sigma_{r+1} = \dots = \sigma_p = 0$ , then

- $\text{rank}(\mathbf{A}) = r$ ,
- $\text{Kern } \mathbf{A} = \text{Span}\{(\mathbf{V})_{:,r+1}, \dots, (\mathbf{V})_{:,n}\}$ ,
- $\text{Im } \mathbf{A} = \text{Span}\{(\mathbf{U})_{:,1}, \dots, (\mathbf{U})_{:,r}\}$ .

Illustration:

$$\begin{bmatrix} \mathbf{A} \end{bmatrix} = \begin{bmatrix} \mathbf{U} \end{bmatrix} \begin{bmatrix} \Sigma_r & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{V}^H \end{bmatrix}$$

(7.5.15)

**MATLAB-code 7.5.16: Computing an ONB of the kernel of a matrix**

```

1 function V = kerncomp(A,tol)
2 % computes an orthonormal basis of KernA using Lemma 7.5.14
3 % kernel selection with relative tolerance tol
4 if (nargin < 2), tol = eps; end
5 [U,S,V] = svd(A); % Singular value decomposition
6 S = diag(S); % Extract vector of singular values
7 % find singular values of relative (w.r.t.  $\sigma_1$ ) size  $\leq tol$ 
8 r = min(find(s/s(1) <= tol)); % "Numerical rank" +1
9 V = V(:,r:end); % rightmost columns of V

```

Remark: MATLAB function  $r=\text{rank}(A)$  relies on  $\text{svd}(A)$

- Application of SVD: PCA by SVD ( $\rightarrow$  Rem. 7.5.3)

By Rem. 7.5.10, see (7.5.12), if  $\mathbf{u}_j \hat{=} \text{columns of } \mathbf{U}$ ,  $\mathbf{v}_j \hat{=} \text{columns of } \mathbf{V}$ ,

$$\begin{bmatrix} \mathbf{A} \end{bmatrix} = \sigma_1 \begin{bmatrix} \mathbf{u}_1 \end{bmatrix} \begin{bmatrix} \mathbf{v}_1^\top \end{bmatrix} + \sigma_2 \begin{bmatrix} \mathbf{u}_2 \end{bmatrix} \begin{bmatrix} \mathbf{v}_2^\top \end{bmatrix} + \dots$$

- ① no perturbations:

SVD:  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^H$  satisfies  $\sigma_1 \geq \sigma_2 \geq \dots \sigma_p > \sigma_{p+1} = \dots = \sigma_{\min\{m,n\}} = 0$ ,  
 $V = \underbrace{\text{Span}\{(\mathbf{U})_{:,1}, \dots, (\mathbf{U})_{:,p}\}}_{\text{orthonormal trend vectors}}$ .

- ② with perturbations:

SVD:  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^H$  satisfies  $\sigma_1 \geq \sigma_2 \geq \dots \sigma_p \gg \sigma_{p+1} \approx \dots \approx \sigma_{\min\{m,n\}} \approx 0$ ,  
 $V = \underbrace{\text{Span}\{(\mathbf{U})_{:,1}, \dots, (\mathbf{U})_{:,p}\}}_{\text{orthonormal trend vectors}}$ .

If there is a pronounced gap in distribution of the singular values, which separates  $p$  large from  $\min\{m, n\} - p$  relatively small singular values, this hints that  $\text{Im } \mathbf{A}$  has essentially dimension  $p$ . It depends on the application what one accepts as a “pronounced gap”.

Frequently used criterion:

$$p = \min \left\{ q : \sum_{j=1}^q \sigma_j^2 \geq (1 - \tau) \sum_{j=1}^{\min\{m,n\}} \sigma_j^2 \right\} \quad \text{for } \tau \ll 1.$$

Information carried by  $\mathbf{V}$  in PCA context:

$$\begin{bmatrix} \mathbf{A} \end{bmatrix} = \sigma_1 \begin{bmatrix} \mathbf{u}_1 \end{bmatrix} \begin{bmatrix} \mathbf{v}_1^\top \end{bmatrix} + \sigma_2 \begin{bmatrix} \mathbf{u}_2 \end{bmatrix} \begin{bmatrix} \mathbf{v}_2^\top \end{bmatrix} + \dots$$

$j$ -th data set ( $\leftrightarrow$  time series # $j$ ) in  $j$ -th column of  $\mathbf{A}$

$$(7.5.12) \Rightarrow (\mathbf{A})_{:,j} = \sigma_1 \mathbf{u}_1(\mathbf{v}_1)_j + \sigma_2 \mathbf{u}_2(\mathbf{v}_2)_j + \dots$$

► The  $j$ -th row of  $\mathbf{V}$  (up to the  $p$ -th component) gives the weights with which the  $p$  identified trends contribute to data set  $j$ .

### Example 7.5.17 (Data points confined to a subspace)

#### MATLAB-code 7.5.18: PCA in three dimensions via SVD

```

1 % Use of SVD for PCA with perturbations
2
3 V = [1 , -1; 0 , 0.5; -1 , 0]; A = V * rand(2,20)+0.1*rand(3,20);
4 [U,S,V] = svd(A,0);
5
6 figure; sv = diag(S(1:3,1:3))
7
8 [X,Y] = meshgrid(-2:0.2:0,-1:0.2:1); n = size(X,1); m =
9     size(X,2);
10 figure; plot3(A(1,:),A(2,:),A(3,:),'r*'); grid on; hold on;
11 M =
12     U(:,1:2)*[reshape(X,1,prod(size(X)));reshape(Y,1,prod(size(Y)))] ;
13 mesh(reshape(M(1,:),n,m),reshape(M(2,:),n,m),reshape(M(3,:),n,m));
14 colormap(cool); view(35,10);
15
16 print -depsc2 '../PICTURES/svdPCA.eps';

```

singular values:

3.1378  
1.8092  
0.1792

third singular value  $\gg$  the data points essentially lie in a 2D subspace.

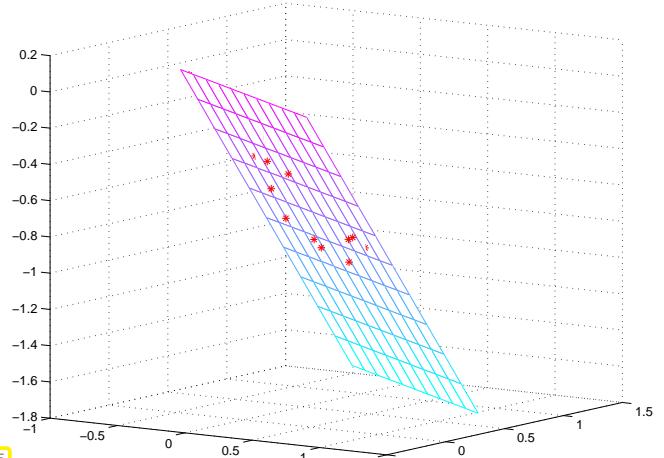


Fig. 255

**Example 7.5.19 (Principal component analysis for data classification) → Ex. 7.5.2 cnt'd)**

Given: measured  $U$ - $I$  characteristics of  $n = 20$  unknown diodes,  $I(U)$  available for  $m = 50$  voltages.

Sought: Number of different types of diodes in batch and reconstructed  $U$ - $I$  characteristic for each type.

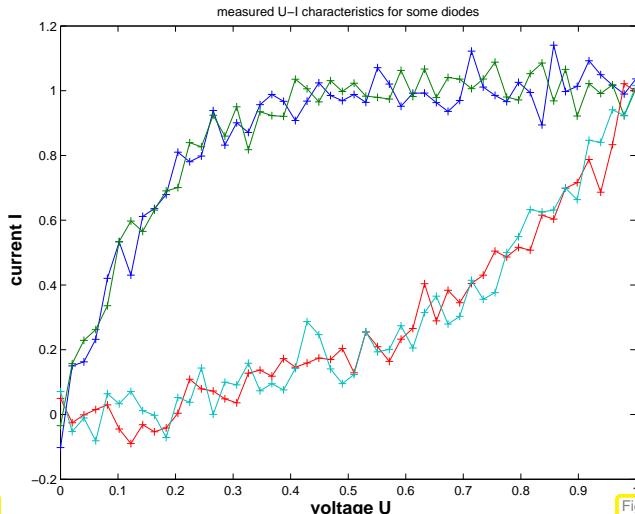


Fig. 256

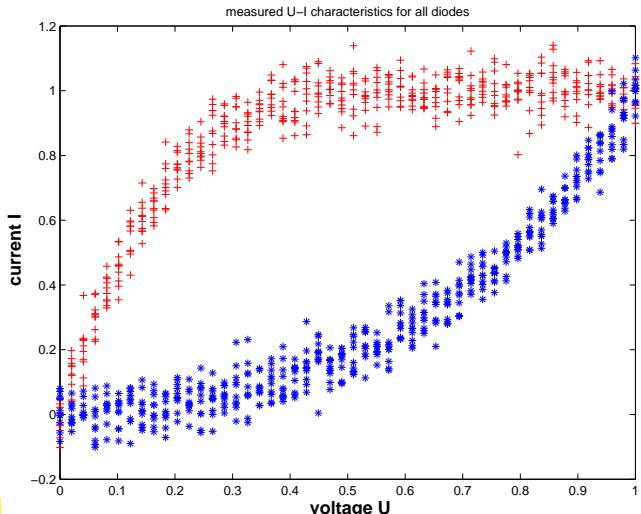


Fig. 257

**MATLAB-code 7.5.20: Generation of synthetic perturbed  $U$ - $I$  characteristics**

```

1 % Generate synthetic measurement curves with random multiplicative and
2 % additive
3 % perturbations supposed to reflect measurement errors and
4 % manufacturing tolerances
5
6 % Voltage-current characteristics of both kinds of diodes
7 i1 = @(u) (2./(1+exp(-10*u))-1);
8 i2 = @(u) ((exp(3*u)-1)/(exp(3)-1));
9 % Simulated measurements
10 m = 50; % number of measurements for different input voltages
11 n = 10; % no. of diodes of each kind
12 na = 0.05; % level of additive noise (normally distributed)
13 nm = 0.02; % level of multiplicative noise (normally distributed)
14
15 uvals = (0:1/(m-1):1);
16 D1 = (1+nm*randn(n,m)).*(i1(repmat(uvals,n,1)))+na*randn(n,m);
17 D2 = (1+nm*randn(n,m)).*(i2(repmat(uvals,n,1)))+na*randn(n,m);
18 A = ([D1;D2])'; A = A(1:size(A,1),randperm(1:size(A,2)));

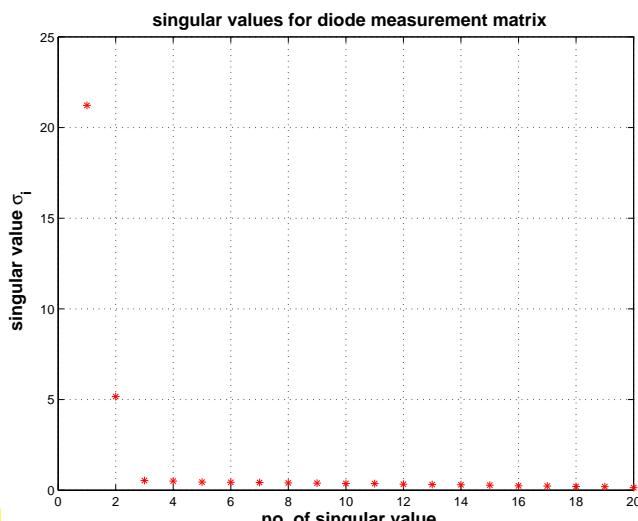
```

Data matrix  $\mathbf{A} \in \mathbb{R}^{m,n}$ ,  $m \gg n$ :

Columns  $\mathbf{A}$  → series of measurements for different diodes (times/locations etc.),  
 Rows of  $\mathbf{A}$  → measured values corresponding to one diode (time/location etc.).

Goal of PCA:

detect linear correlations between columns of  $\mathbf{A}$



← distribution of singular values of matrix  
two dominant singular values !

measurements display linear correlation with **two principal components**

**two types of diodes in batch**

### MATLAB-code 7.5.21: PCA for measured $U$ - $I$ characteristics

```

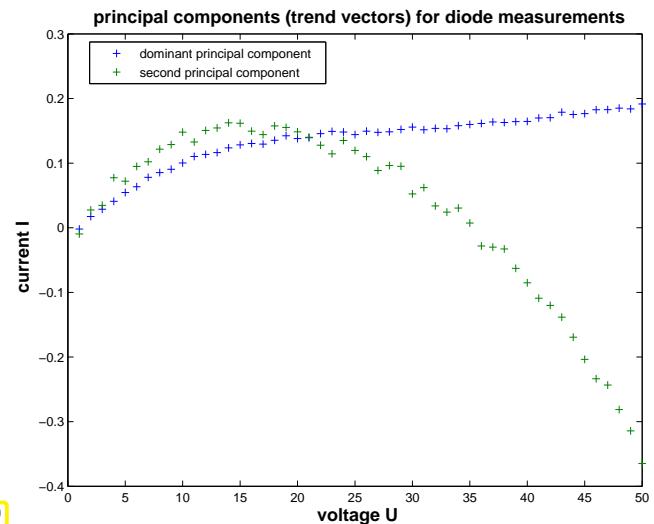
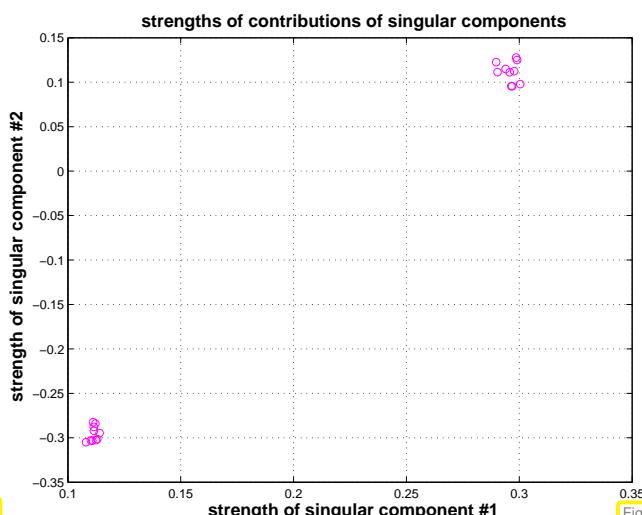
1 clear all; diodedata;
2
3 figure('name','U-I characteristics for a few diodes');
4 plot(uvals,(D(:,[1 7 13 17]))','-+');
5 xlabel('{\bf voltage U}', 'fontsize', 14);
6 ylabel('{\bf current I}', 'fontsize', 14);
7 title('measured U-I characteristics for some diodes');
8 print -depsc2 '../PICTURES/diodepcafewmeas.eps';
9
10 figure('name','measured U-I characteristics');
11 plot(uvals,D1,'r+'); hold on;
12 plot(uvals,D2,'b*');
13 xlabel('{\bf voltage U}', 'fontsize', 14);
14 ylabel('{\bf current I}', 'fontsize', 14);
15 title('measured U-I characteristics for all diodes');
16 print -depsc2 '../PICTURES/diodepcameas.eps';
17
18 % Perform SVD based PCA
19 [U,S,V] = svd(D);
20
21 figure('name','singular values');
22 sv = diag(S(1:2*n,1:2*n));
23 plot(1:2*n,sv,'r*'); grid on;
24 xlabel('{\bf index i of singular value}', 'fontsize', 14);
25 ylabel('{\bf singular value \sigma_i}', 'fontsize', 14);
26 title('{\bf singular values for diode measurement
matrix}', 'fontsize', 14);
27 print -depsc2 '../PICTURES/diodepcasv.eps';
28
29 figure('name','trend vectors');
30 plot(1:m,U(:,1:2),'+');
31 xlabel('{\bf voltage U}', 'fontsize', 14);
32 ylabel('{\bf current I}', 'fontsize', 14);
33 title('{\bf principal components (trend vectors) for diode
}
```

```

    measurements}', 'fontsize', 14);
34 legend('dominant principal component', 'second principal
    component', 'location', 'best');
35 print -depsc2 '../PICTURES/diodepcav.eps';

36
37 figure('name', 'strength');
38 plot (V(:,1), V(:,2), 'mo'); grid on;
39 xlabel('{\bf strength of singular component #1}', 'fontsize', 14);
40 ylabel('{\bf strength of singular component #2}', 'fontsize', 14);
41 title('{\bf strengths of contributions of singular
    components}', 'fontsize', 14);
42 print -depsc2 '../PICTURES/diodepcav.eps';

```



Observations:

- First two rows of  $\mathbf{V}$ -matrix specify strength of contribution of the two leading principal components to each measurement
  - Points  $(\mathbf{V})_{:,1:2}$ , which correspond to different diodes are neatly clustered in  $\mathbb{R}^2$ . To determine the type of diode  $i$ , we have to identify the cluster to which the point  $((\mathbf{V})_{i,1}, \mathbf{V}_{i,2})$  belongs ( $\rightarrow$  **cluster analysis**, course “machine learning”, next Rem. 7.5.22).
- The principal components themselves do not carry much useful information in this example.

### Remark 7.5.22.

Algorithm for cluster analysis

Given: •  $N$  data points  $\mathbf{x}_i \in \mathbb{R}^k$ ,  $i = 1, \dots, N$ ,  
• No.  $n$  of desired clusters.

Sought: Partitioning of index set  $\{1, \dots, N\} = I_1 \cup \dots \cup I_n$ , achieving minimal **mean least squares error**

$$\text{mlse} := \sum_{l=1}^n \sum_{i \in I_l} \|\mathbf{x}_i - \mathbf{m}_l\|_2^2, \quad \mathbf{m}_l = \frac{1}{\#I_l} \sum_{i \in I_l} \mathbf{x}_i. \quad (7.5.23)$$

The subsets  $\{\mathbf{x}_i : i \in I_l\}$  are called the **clusters**. The points  $\mathbf{m}_l$  are their **centers of gravity**.

The Algorithm involves two components:

- ❶ Improvement of clusters using the **Lloyd-Max algorithm**, see Code 7.5.28. It involves two steps in turns:

- (a) Given centers of gravity  $\mathbf{m}_l$  redistribute points according to

$$I_l := \{i \in \{1, \dots, N\} : \|\mathbf{x}_i - \mathbf{m}_l\|_2 \leq \|\mathbf{x}_i - \mathbf{m}_k\|_2 \forall k \neq l\}. \quad (7.5.24)$$

- (b) Recompute centers of gravity

$$\mathbf{m}_l = \frac{1}{\#I_l} \sum_{i \in I_l} \mathbf{x}_i. \quad (7.5.25)$$

- ❷ Splitting of cluster by separation along its **principal axis**, see Code 7.5.27:

$$\mathbf{a}_l := \underset{\|\mathbf{v}\|_2=1}{\operatorname{argmax}} \left\{ \sum_{i \in I_l} |(\mathbf{x}_i - \mathbf{m}_l)^T \mathbf{v}|^2 \right\} \quad (7.5.26)$$

#### MATLAB-code 7.5.27: Principal axis point set separation

```

1 function [i1,i2] = princaxissep(X)
2 % Separation of a set of points whose coordinates are stored in the
3 % columns of
4 % X according to their location w.r.t. the principal axis
5
6 N = size(X,2); % no. of. points
7 g = sum(X')'/N; % Compute center of gravity, cf. (7.5.25)
8 Y = X - repmat(g,1,N); % Normalize point coordinates.
9
10 % Compute principal axes, cf. (7.5.26) and (7.5.35). Note
11 % that the SVD of a symmetric matrix is available through an orthonormal
12 % basis of
13 % eigenvectors.
14 [V,D] = eig(Y*Y');
15 a = V(:,end); % Major principal axis
16 c = a'*Y; % Coordinates of points w.r.t. to major principal axis
17 % Split point set according to locations of projections on principal
18 % axis
19 i1 = find(c < 0); i2 = find(c >= 0);

```

#### MATLAB-code 7.5.28: Lloyd-Max algorithm for cluster identification

```

1 function [C,idx,cds] = lloydmax(X,C,tol)
2 % Lloyd-Max iterative vector quantization algorithm for discrete point
3 % sets; the
4 % columns of X contain the points  $\mathbf{x}_i$ , the columns of
5 % C initial approximations for the centers of the clusters. The final

```

```

5 % are returned in C, the index vector idx specifies the association
6 % of points with centers.
7 k = size(X,1); % dimension of space
8 N = size(X,2); % no. of points
9 n = size(C,2); % no. of clusters
10 if (k ~= size(C,1)), error('dimension mismatch'); end
11 if (nargin < 3), tol = 0.0001; end
12
13 sd_old = realmax;
14 [sd,idx] = distcomp(X,C),
15 % Terminate, if sum of squared minimal distances has not changed much
16 while ((sd_old-sd)/sd > tol)
17 % Compute new centers of gravity according to (7.5.25)
18 for j=1:n
19 idj = find(idx == j);
20 nj = length(idj);
21 if (nj > 0), C(:,j) = sum(X(:,idj)').'/nj; end
22 end
23 sd_old = sd;
24 [sd,idx,cds] = distcomp(X,C),
25 end
26
27 end
28
29 function [sumd,idx,cds] = distcomp(X,C)
30 % Compute squared distances
31 d = [];
32 for j=1:size(C,2)
33 Dv = X - repmat(C(:,j),1,size(X,2));
34 d = [d; sum(Dv.*Dv)];
35 end
36 % Compute minimum distance point association and sum of minimal squared
37 % distances
37 [mx,idx] = min(d);
38 sumd = sum(mx);
39 % Computer sum of squared distances within each cluster
40 for j=1:size(C,2)
41 cds(j) = sum(mx(find(idx == j)));
42 end
43 end

```

### MATLAB-code 7.5.29: Clustering of point set

```

1 function [C,idx] = pointcluster(X,n)
2 % n-quantization of point set in k-dimensional space based on
3 % minimizing the mean square
4 % error of Euclidean distances. The columns of the matrix X contain
5 % the point
6 % coordinates, n specifies the desired number of clusters.

```

```

6 N = size(X,2); % no. of points
7 k = size(X,1); % dimension of space
8
9 % Start with two clusters obtained by principal axis separation
10 nc = 1; % Current number of clusters
11 Ibig = 1:N; % Initial single cluster encompassing all points
12 nbig = 1; % Index of largest cluster
13 C = sum(X')'/N; % center of gravity
14 idx = ones(1,N);
15
16 while (nc < n)
17     % Split largest cluster into two using the principal axis separation
18     % algorithm
19     [i1,i2] = princaxissep(X(:,Ibig));
20     i1 = Ibig(i1); i2 = Ibig(i2);
21     n1 = length(i1); n2 = length(i2);
22     c1 = sum(X(:,i1)')/n1; c2 = sum(X(:,i2)')/n2;
23     C(:,nbig) = c1; C = [C,c2];
24     nc = nc+1;
25     % Improve clusters by Lloyd-Max iteration
26     [C,idx,cds] = lloydmax(X,C);
27     % Identify cluster with biggest contribution to mean square error
28     [cdm,nbig] = max(cds);
29     Ibig = find(idx == nbig);
30 end

```

### Example 7.5.30 (Principal component analysis for data analysis) → Ex. 7.5.1 cnt'd)

$\mathbf{A} \in \mathbb{R}^{m,n}$ ,  $m \gg n$ :

Columns  $\mathbf{A}$  → series of measurements at different times/locations etc.  
 Rows of  $\mathbf{A}$  → measured values corresponding to one time/location etc.

Goal: detect linear correlations

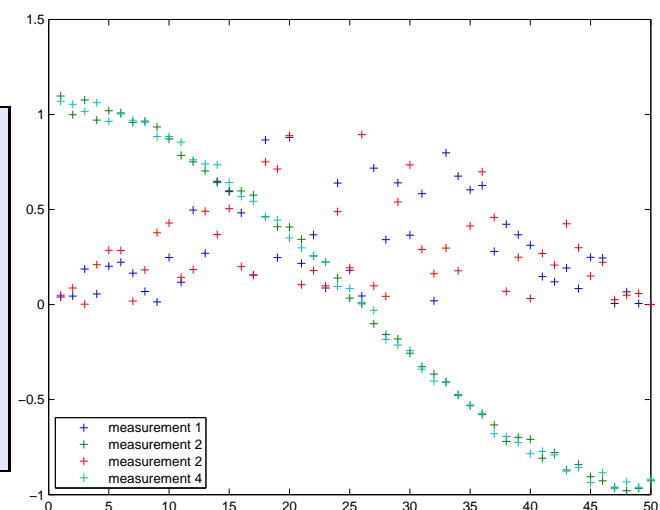
Concrete: two quantities measured over one year at 10 different sites

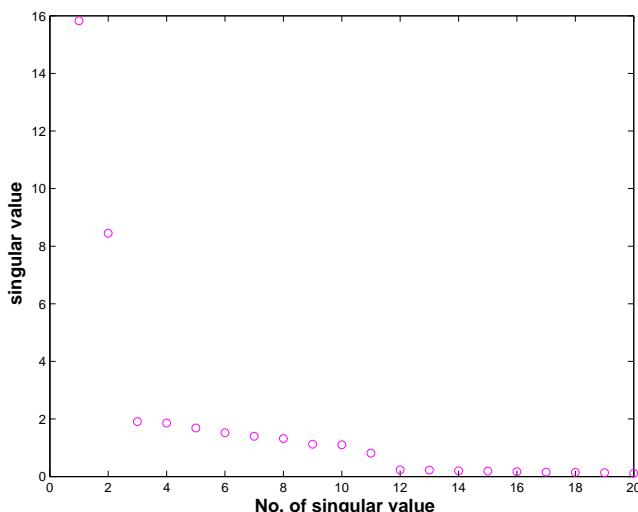
(Of course, measurements affected by errors/fluctuations)

```

1 n = 10;
2 m = 50;
3 x = sin(pi*(1:m)' /m);
4 y = cos(pi*(1:m)' /m);
5 A = [];
6 for i = 1:n
7     A = [A, x.*rand(m,1), ...
8           y+0.1*rand(m,1)];
9 end

```





← distribution of singular values of matrix  
two dominant singular values !

measurements display linear correlation with **two principal components**

principal components =  $\mathbf{u}_{\cdot,1}, \mathbf{u}_{\cdot,2}$   
their relative weights =  $\mathbf{v}_{\cdot,1}, \mathbf{v}_{\cdot,2}$

(leftmost columns of **U**-matrix of SVD)  
(leftmost columns of **V**-matrix of SVD)

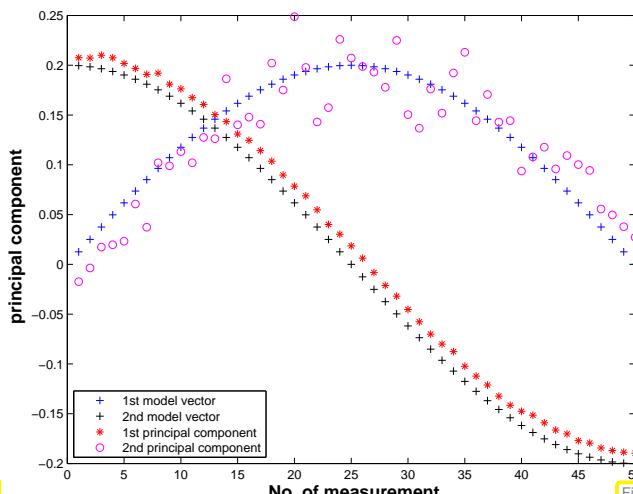


Fig. 261

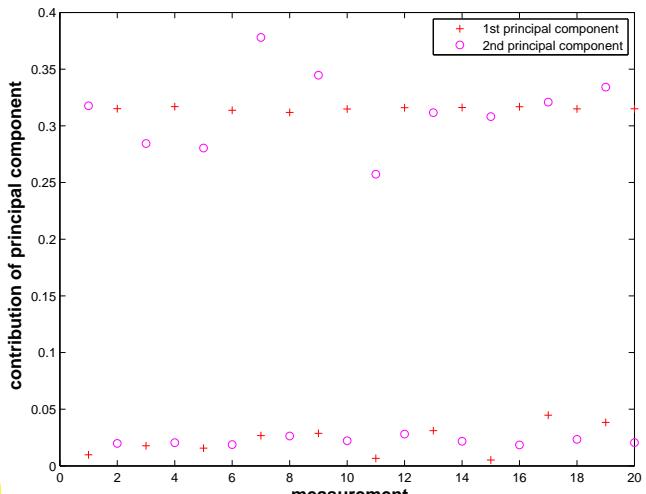


Fig. 262

### Example 7.5.31 (PCA of stock prices → Ex. 7.5.1)

columns of **A** → time series of end of day stock prices of individual stocks  
rows of **A** → closing prices of DAX stocks on a particular day

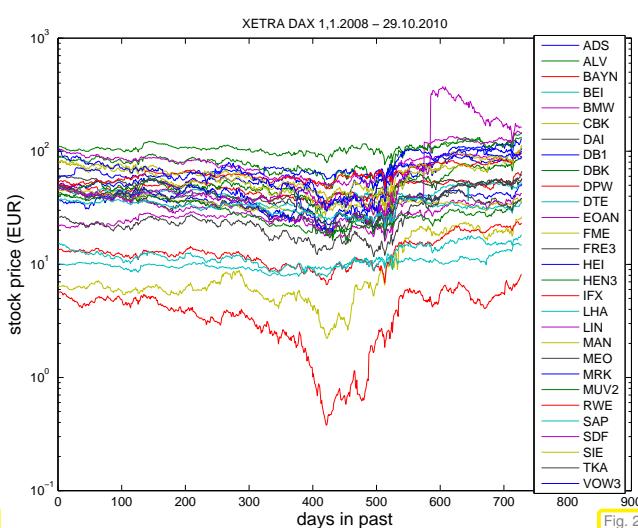


Fig. 263

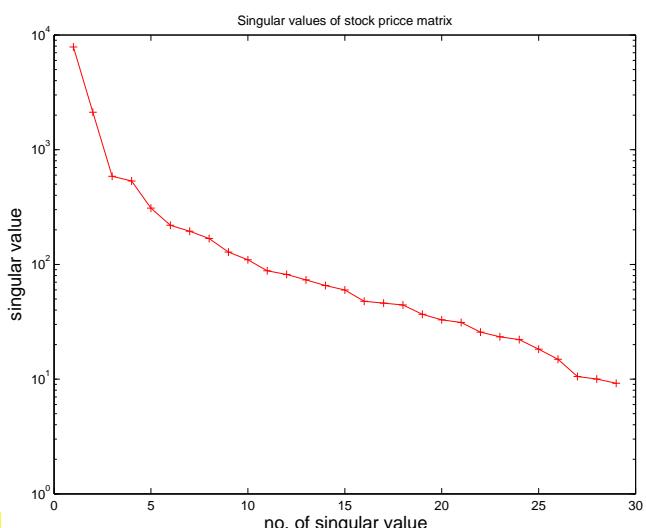
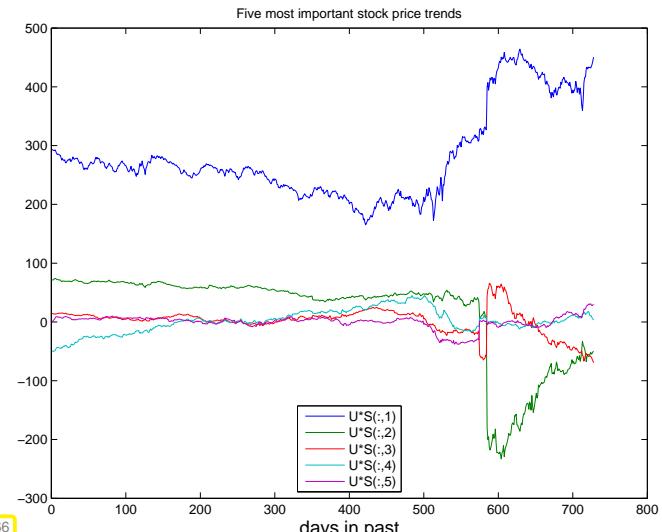
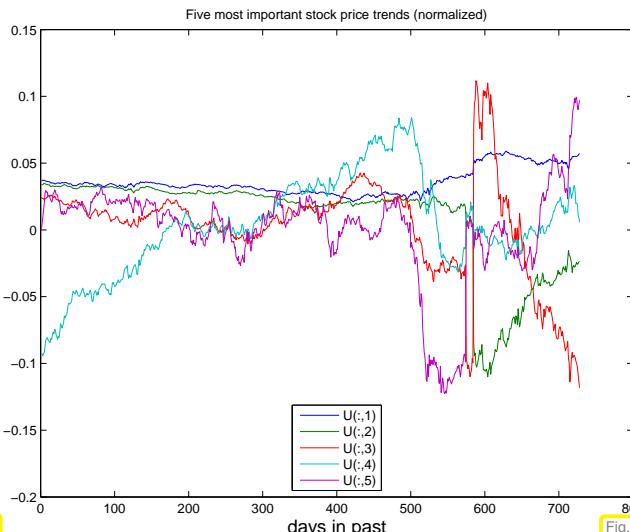


Fig. 264

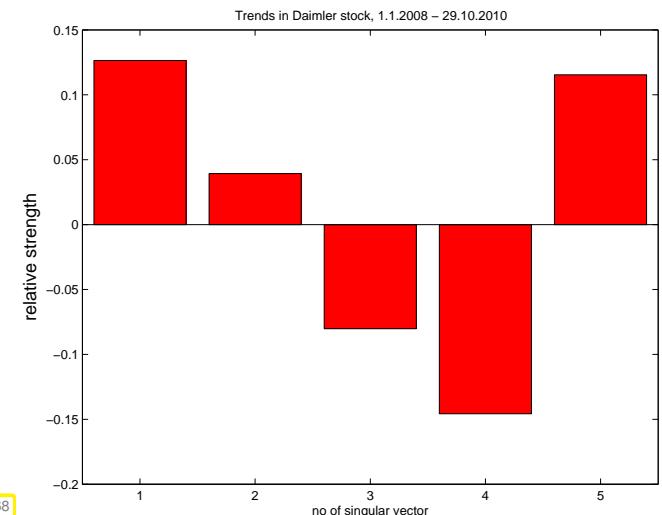
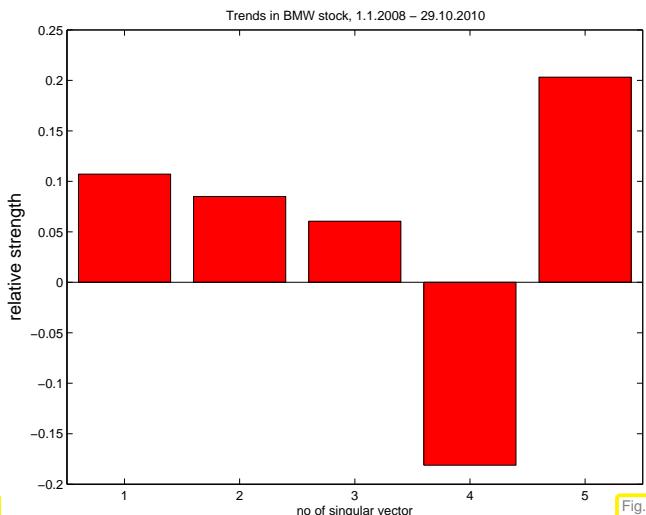
We observe a pronounced decay of the singular values ( $\approx$  exponential decay, logarithmic scale in Fig. 264)

- a few trends (corresponding to a few of the largest singular values) govern the time series.



Columns of  $\mathbf{U}$  (→ Fig. 265) in SVD  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$  provide trend vectors, cf. Rem. 7.5.3 & Ex. 7.5.30.

When weighted with the corresponding singular value, the importance of a trend contribution emerges, see Fig. 266



Stocks of companies from the same sector of the economy should display similar contributions of major trend vectors, because their prices can be expected to be more closely correlated than stock prices in general.

Data obtained from Yahoo Finance:

#### MATLAB-code 7.5.32:

```
#!/bin/csh
foreach i (ADS ALV BAYN BEI BMW CBK DAI DBK DB1 LHA DPW DTE EOAN FRE3 \
           FME HEI HEN3 IFX SDF LIN MAN MRK MEO MUV2 RWE SAP SIE TKA VOW3)
wget -O "i".csv
"http://ichart.finance.yahoo.com/table.csv?s=i.DE&a=00&b=1&
c=2008&d=09&e=30&f=2010&g=d&ignore=.csv"
sed -i -e 's/-/,/g' "i".csv
end
```

**MATLAB-code 7.5.33: PCA of stock prices in MATLAB**

```

1 % Read end of day XETRA-DAX stock quotes into matrices
2 k = 1;
3 M{k} = dlmread('ADS.csv',',',1,0); leg{k} = 'ADS'; k = k + 1;
4 M{k} = dlmread('ALV.csv',',',1,0); leg{k} = 'ALV'; k = k + 1;
5 M{k} = dlmread('BAYN.csv',',',1,0); leg{k} = 'BAYN'; k = k + 1;
6 M{k} = dlmread('BEI.csv',',',1,0); leg{k} = 'BEI'; k = k + 1;
7 M{k} = dlmread('BMW.csv',',',1,0); leg{k} = 'BMW'; k = k + 1;
8 M{k} = dlmread('CBK.csv',',',1,0); leg{k} = 'CBK'; k = k + 1;
9 M{k} = dlmread('DAI.csv',',',1,0); leg{k} = 'DAI'; k = k + 1;
10 M{k} = dlmread('DB1.csv',',',1,0); leg{k} = 'DB1'; k = k + 1;
11 M{k} = dlmread('DBK.csv',',',1,0); leg{k} = 'DBK'; k = k + 1;
12 M{k} = dlmread('DPW.csv',',',1,0); leg{k} = 'DPW'; k = k + 1;
13 M{k} = dlmread('DTE.csv',',',1,0); leg{k} = 'DTE'; k = k + 1;
14 M{k} = dlmread('EOAN.csv',',',1,0); leg{k} = 'EOAN'; k = k + 1;
15 M{k} = dlmread('FME.csv',',',1,0); leg{k} = 'FME'; k = k + 1;
16 M{k} = dlmread('FRE3.csv',',',1,0); leg{k} = 'FRE3'; k = k + 1;
17 M{k} = dlmread('HEI.csv',',',1,0); leg{k} = 'HEI'; k = k + 1;
18 M{k} = dlmread('HEN3.csv',',',1,0); leg{k} = 'HEN3'; k = k + 1;
19 M{k} = dlmread('IFX.csv',',',1,0); leg{k} = 'IFX'; k = k + 1;
20 M{k} = dlmread('LHA.csv',',',1,0); leg{k} = 'LHA'; k = k + 1;
21 M{k} = dlmread('LIN.csv',',',1,0); leg{k} = 'LIN'; k = k + 1;
22 M{k} = dlmread('MAN.csv',',',1,0); leg{k} = 'MAN'; k = k + 1;
23 M{k} = dlmread('MEO.csv',',',1,0); leg{k} = 'MEO'; k = k + 1;
24 M{k} = dlmread('MRK.csv',',',1,0); leg{k} = 'MRK'; k = k + 1;
25 M{k} = dlmread('MUV2.csv',',',1,0); leg{k} = 'MUV2'; k = k + 1;
26 M{k} = dlmread('RWE.csv',',',1,0); leg{k} = 'RWE'; k = k + 1;
27 M{k} = dlmread('SAP.csv',',',1,0); leg{k} = 'SAP'; k = k + 1;
28 M{k} = dlmread('SDF.csv',',',1,0); leg{k} = 'SDF'; k = k + 1;
29 M{k} = dlmread('SIE.csv',',',1,0); leg{k} = 'SIE'; k = k + 1;
30 M{k} = dlmread('TKA.csv',',',1,0); leg{k} = 'TKA'; k = k + 1;
31 M{k} = dlmread('VOW3.csv',',',1,0); leg{k} = 'VOW3';
32 % Data format of matrices M:
33 % M(:,1): year, M(:,2): month, M(:,3): day, M(:,6): end of day
34 % stock quote
35 % Clean data: remove/interpolate days without trading
36 dv = [];
37 for j=1:k
38     M{j}(:,1) = round(366*M{j}(:,1)+31*M{j}(:,2)+M{j}(:,3));
39     dv = [dv; M{j}(:,1)];
40 end
41 dv = unique(dv); mxd = max(dv)+1;
42 dv = mxd - dv; mdays = max(dv);
43 A = zeros(mdays,k);
44 for j=1:k, A(mxd - M{j}(:,1),j) = M{j}(:,6); end
45 idx = find(sum(A') ~= 0); A = A(idx,:);
46 for j=size(A,1):-1:2
47     zidx = find(A(j-1,:) == 0);
48     A(j-1,zidx) = A(j,zidx);
49 end
50 for j=2:size(A,1)
51     zidx = find(A(j,:)== 0);

```

```

52     A(j,zidx) = A(j-1,zidx);
53 end
54
55 figure('name','DAX');
56 semilogy(A); set(gca,'xlim',[0 900]);
57 legend(leg,'location','east');
58 xlabel('days in past','fontsize',14);
59 ylabel('stock price (EUR)','fontsize',14);
60 title('XETRA DAX 1.1.2008 - 29.10.2010');
61 print -depsc2 '../PICTURES/stockprizes.eps';
62
63 [U,S,V] = svd(A,0);
64 figure('name','singular values');
65 semilogy(diag(S),'r-+');
66 xlabel('no. of singular value','fontsize',14);
67 ylabel('singular value','fontsize',14);
68 title('Singular values of stock pricce matrix');
69 print -depsc2 '../PICTURES/stocksingval.eps';
70
71 figure('name','trend vectors');
72 plot(U(:,1:5));
73 xlabel('days in past','fontsize',14);
74 title('Five most important stock price trends (normalized)');
75 legend('U(:,1)', 'U(:,2)', 'U(:,3)', 'U(:,4)', 'U(:,5)', 'location', 'south');
76 print -depsc2 '../PICTURES/stocktrendsns.eps';
77
78 figure('name','trend vectors');
79 plot(U(:,1:5)*S(1:5,1:5));
80 xlabel('days in past','fontsize',14);
81 title('Five most important stock price trends');
82 legend('U*S(:,1)', 'U*S(:,2)', 'U*S(:,3)', 'U*S(:,4)', 'U*S(:,5)', 'location', 'south');
83 print -depsc2 '../PICTURES/stocktrends.eps';
84
85 figure('name','trend components BMW');
86 trendcomp(V,5,5);
87 title('Trends in BMW stock, 1.1.2008 - 29.10.2010');
88 print -depsc2 '../PICTURES/bmw.eps';
89
90 figure('name','trend components DAI');
91 trendcomp(V,7,5);
92 title('Trends in Daimler stock, 1.1.2008 - 29.10.2010');
93 print -depsc2 '../PICTURES/dai.eps';
94
95 figure('name','trend components DBK');
96 trendcomp(V,9,5);
97 title('Trends in Deutsche Bank stock, 1.1.2008 - 29.10.2010');
98 print -depsc2 '../PICTURES/dbk.eps';
99
100 figure('name','trend components CBK');
101 trendcomp(V,6,5);
102 title('Trends in Commerzbank stock, 1.1.2008 - 29.10.2010');
103 print -depsc2 '../PICTURES/cbk.eps';

```

- Application of SVD: extrema of quadratic forms on the unit sphere

A minimization problem on the Euclidean unit sphere  $\{\mathbf{x} \in \mathbb{K}^n : \|\mathbf{x}\|_2 = 1\}$ :

$$\text{given } \mathbf{A} \in \mathbb{K}^{m,n}, m > n, \text{ find } \mathbf{x} \in \mathbb{K}^n, \|\mathbf{x}\|_2 = 1, \|\mathbf{Ax}\|_2 \rightarrow \min. \quad (7.5.34)$$

Use that multiplication with unitary matrices preserves the 2-norm ( $\rightarrow ??$ ) and the singular value decomposition  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^H$  ( $\rightarrow$  Def. 7.5.8):

$$\begin{aligned} \min_{\|\mathbf{x}\|_2=1} \|\mathbf{Ax}\|_2^2 &= \min_{\|\mathbf{x}\|_2=1} \|\mathbf{U}\Sigma\mathbf{V}^H\mathbf{x}\|_2^2 = \min_{\|\mathbf{V}^H\mathbf{x}\|_2=1} \|\mathbf{U}\Sigma(\mathbf{V}^H\mathbf{x})\|_2^2 \\ &= \min_{\|\mathbf{y}\|_2=1} \|\Sigma\mathbf{y}\|_2^2 = \min_{\|\mathbf{y}\|_2=1} (\sigma_1^2 y_1^2 + \dots + \sigma_n^2 y_n^2) \geq \sigma_n^2. \end{aligned}$$

The minimum  $\sigma_n^2$  is attained for  $\mathbf{y} = \mathbf{e}_n \Rightarrow \text{minimizer } \mathbf{x} = \mathbf{V}\mathbf{e}_n = (\mathbf{V})_{:,n}$ .

By similar arguments:

$$\sigma_1 = \max_{\|\mathbf{x}\|_2=1} \|\mathbf{Ax}\|_2, \quad (\mathbf{V})_{:,1} = \operatorname{argmax}_{\|\mathbf{x}\|_2=1} \|\mathbf{Ax}\|_2. \quad (7.5.35)$$

Recall: 2-norm of the matrix  $\mathbf{A}$  ( $\rightarrow$  Def. 1.5.71) is defined as the maximum in (7.5.35). Thus we have proved the following theorem:

### Lemma 7.5.36. SVD and Euclidean matrix norm

- $\forall \mathbf{A} \in \mathbb{K}^{m,n}: \|\mathbf{A}\|_2 = \sigma_1(\mathbf{A}),$
- $\forall \mathbf{A} \in \mathbb{K}^{n,n} \text{ regular: } \operatorname{cond}_2(\mathbf{A}) = \sigma_1/\sigma_n.$

Remark: MATLAB functions `norm(A)` and `cond(A)` rely on `svd(A)`

- Application of SVD: *best low rank approximation*

### Definition 7.5.37. Frobenius norm

The **Frobenius norm** of  $\mathbf{A} \in \mathbb{K}^{m,n}$  is defined as

$$\|\mathbf{A}\|_F^2 := \sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2.$$

Obvious (why ?):  $\|\mathbf{A}\|_F$  invariant under unitary transformations of  $\mathbf{A}$

Frobenius norm and SVD:

$$\|\mathbf{A}\|_F^2 = \sum_{j=1}^p \sigma_j^2 \quad (7.5.38)$$

notation:  $\mathcal{R}_k(m, n) := \{\mathbf{A} \in \mathbb{K}^{m,n} : \text{rank}(\mathbf{A}) \leq k\}, m, n, k \in \mathbb{N}$

**Theorem 7.5.39. best low rank approximation** → [34, Thm. 11.6]

Let  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^H$  be the SVD of  $\mathbf{A} \in \mathbb{K}^{m,n}$  (→ Thm. 7.5.6). For  $1 \leq k \leq \text{rank}(\mathbf{A})$  set  $\mathbf{U}_k := [\mathbf{u}_{\cdot,1}, \dots, \mathbf{u}_{\cdot,k}] \in \mathbb{K}^{m,k}$ ,  $\mathbf{V}_k := [\mathbf{v}_{\cdot,1}, \dots, \mathbf{v}_{\cdot,k}] \in \mathbb{K}^{n,k}$ ,  $\Sigma_k := \text{diag}(\sigma_1, \dots, \sigma_k) \in \mathbb{K}^{k,k}$ . Then, for  $\|\cdot\| = \|\cdot\|_F$  and  $\|\cdot\| = \|\cdot\|_2$ , holds true

$$\|\mathbf{A} - \mathbf{U}_k \Sigma_k \mathbf{V}_k^H\| \leq \|\mathbf{A} - \mathbf{F}\| \quad \forall \mathbf{F} \in \mathcal{R}_k(m, n).$$

Thm. 7.5.39: the rank- $k$ -matrix that is *closest* to  $\mathbf{A}$  (rank- $k$  best approximation) in either the Euclidean matrix norm or the Frobeniusnorm (→ Def. 7.5.37) can be obtained by truncating the rank-1sum expansion (7.5.12) from the SVD of  $\mathbf{A}$  after  $k$  terms.

*Proof.* Write  $\mathbf{A}_k = \mathbf{U}_k \Sigma_k \mathbf{V}_k^H$ . Obviously, with  $r = \text{rank } \mathbf{A}$ ,

$$\text{rank } \mathbf{A}_k = k \quad \text{and} \quad \|\mathbf{A} - \mathbf{A}_k\| = \|\Sigma - \Sigma_k\| = \begin{cases} \sigma_{k+1} & , \text{for } \|\cdot\| = \|\cdot\|_2, \\ \sqrt{\sigma_{k+1}^2 + \dots + \sigma_r^2} & , \text{for } \|\cdot\| = \|\cdot\|_F. \end{cases}$$

① Pick  $\mathbf{B} \in \mathbb{K}^{n,n}$ ,  $\text{rank } \mathbf{B} = k$ .

$$\Rightarrow \dim \text{Kern } \mathbf{B} = n - k \Rightarrow \text{Kern } \mathbf{B} \cap \text{Span}\{\mathbf{v}_1, \dots, \mathbf{v}_{k+1}\} \neq \{0\},$$

where  $\mathbf{v}_i, i = 1, \dots, n$  are the columns of  $\mathbf{V}$ . For  $\mathbf{x} \in \text{Kern } \mathbf{B} \cap \text{Span}\{\mathbf{v}_1, \dots, \mathbf{v}_{k+1}\}$ ,  $\|\mathbf{x}\|_2 = 1$

$$\begin{aligned} \mathbf{x} &= \sum_{j=1}^{k+1} (\mathbf{v}_j^H \mathbf{x}) \mathbf{v}_j, \\ \|\mathbf{A} - \mathbf{B}\|_2^2 &\geq \|(\mathbf{A} - \mathbf{B})\mathbf{x}\|_2^2 = \|\mathbf{A}\mathbf{x}\|_2^2 = \left\| \sum_{j=1}^{k+1} \sigma_j (\mathbf{v}_j^H \mathbf{x}) \mathbf{u}_j \right\|_2^2 = \sum_{j=1}^{k+1} \sigma_j^2 (\mathbf{v}_j^H \mathbf{x})^2 \geq \sigma_{k+1}^2, \end{aligned}$$

because  $\sum_{j=1}^{k+1} (\mathbf{v}_j^H \mathbf{x})^2 = \|\mathbf{x}\|_2^2 = 1$ .

② Find ONB  $\{\mathbf{z}_1, \dots, \mathbf{z}_{n-k}\}$  of  $\text{Kern } \mathbf{B}$  and assemble it into a matrix  $\mathbf{Z} = [\mathbf{z}_1 \dots \mathbf{z}_{n-k}] \in \mathbb{K}^{n,n-k}$

$$\|\mathbf{A} - \mathbf{B}\|_F^2 \geq \|(\mathbf{A} - \mathbf{B})\mathbf{Z}\|_F^2 = \|\mathbf{A}\mathbf{Z}\|_F^2 = \sum_{i=1}^{n-k} \|\mathbf{A}\mathbf{z}_i\|_2^2 = \sum_{i=1}^{n-k} \sum_{j=1}^r \sigma_j^2 (\mathbf{v}_j^H \mathbf{z}_i)^2$$

□

Since matrix norms  $\|\cdot\|_2$  and  $\|\cdot\|_F$  are invariant under multiplication with orthogonal (unitary) matrices,

$$\|\mathbf{A} - \mathbf{U}_k \Sigma_k \mathbf{V}_k^H\|_2 = \sigma_{k+1}, \quad (7.5.40)$$

$$\left\| \mathbf{A} - \mathbf{U}_k \Sigma_k \mathbf{V}_k^H \right\|_F^2 = \sum_{j=k+1}^{\min\{m,n\}} \sigma_j^2. \quad (7.5.41)$$

This provides precise information about the best approximation error for rank- $k$ -matrices.

Note: information content of a rank- $k$  matrix  $\mathbf{M} \in \mathbb{K}^{m,n}$  is equivalent to  $k(m+n)$  numbers!

Approximation by low-rank matrices  $\leftrightarrow$  matrix compression

### Example 7.5.42 (Image compression )

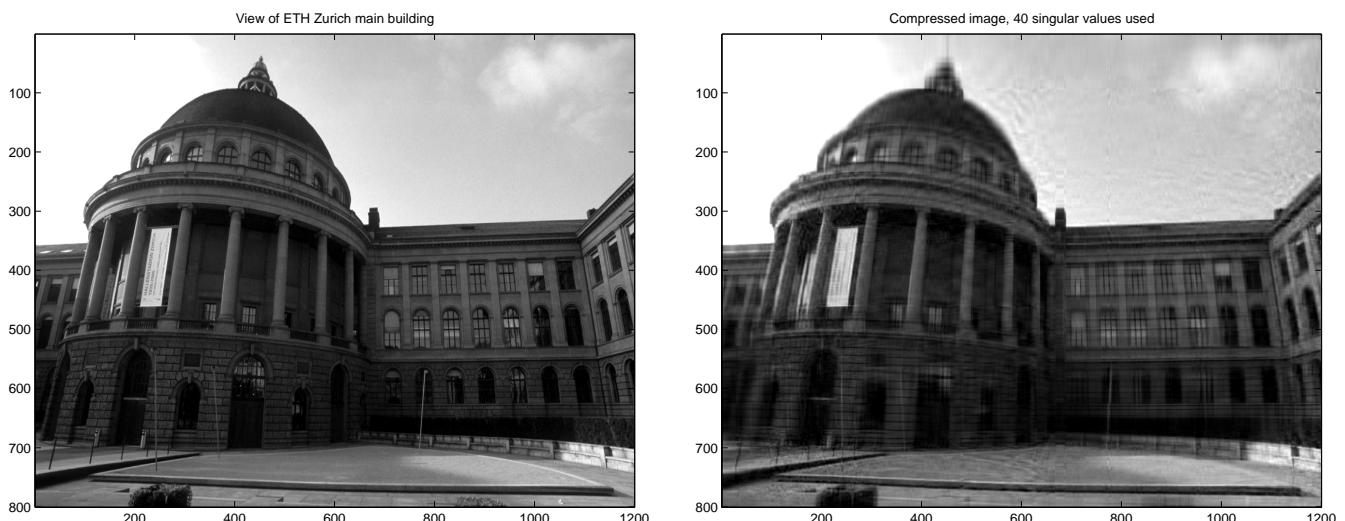
Image composed of  $m \times n$  pixels (greyscale, BMP format)  $\leftrightarrow$  matrix  $\mathbf{A} \in \mathbb{R}^{m,n}$ ,  $a_{ij} \in \{0, \dots, 255\}$ , see Ex. 7.3.24

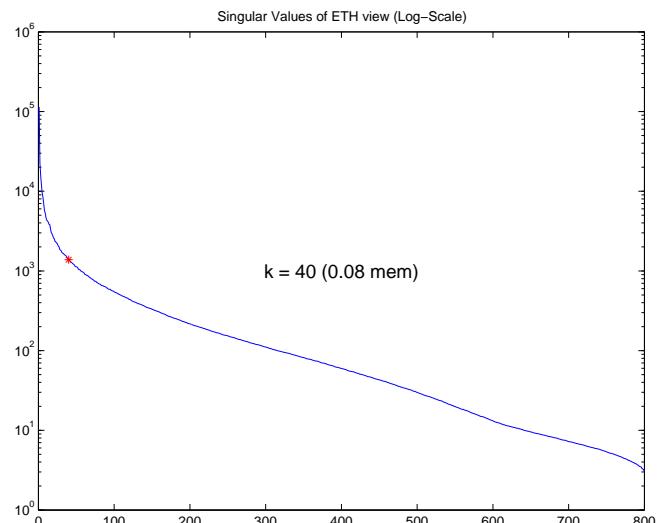
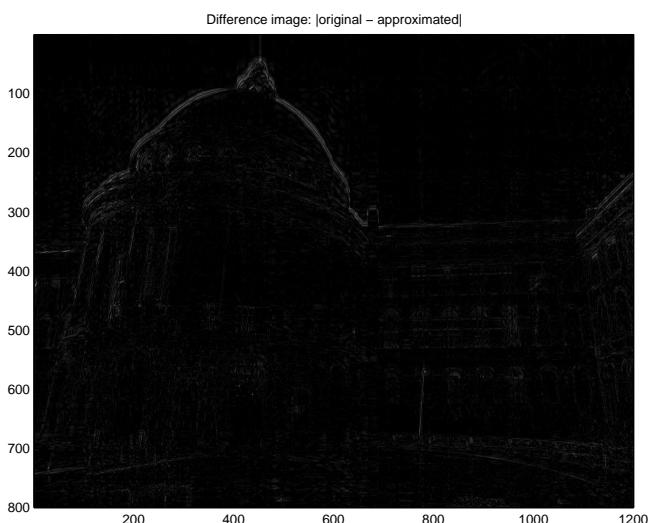
► Thm. 7.5.39  $\geq$  best low rank approximation of image:  $\tilde{\mathbf{A}} = \mathbf{U}_k \Sigma_k \mathbf{V}^\top$

#### MATLAB-code 7.5.43: SVD based image compression

```

1 P = double(imread('eth.pbm'));
2 [m,n] = size(P); [U,S,V] = svd(P); s = diag(S);
3 k = 40; S(k+1:end,k+1:end) = 0; PC = U*S*V';
4
5 figure('position',[0 0 1600 1200]); col = [0:1/215:1]*[1,1,1];
6 subplot(2,2,1); image(P); title('original image'); colormap(col);
7 subplot(2,2,2); image(PC); title('compressed (40 S.V.)');
    colormap(col);
8 subplot(2,2,3); image(abs(P-PC)); title('difference');
    colormap(col);
9 subplot(2,2,4); cla; semilogy(s); hold on; plot(k,s(k),'ro');
```





#### MATLAB-code 7.5.44: Image compression

```

1 [img_data] = imread(<file>.jpg');
2 img_data = double(img_data);
3 image(img_data);
4 [U,S,V] = svd(img_data(:,:,1));
5 s = diag(S);
6 k=20;
7 img_data_comp = U(:,1:k)*diag(s(1:k))*V(:,1:k)';
8 image(img_data_comp);
9 col = [0:1/215:1]*[1,1,1];
10 colormap(col);

```

However: there are better and faster ways to compress images than SVD (JPEG, Wavelets, etc.)

# Chapter 8

## Krylov Methods for Linear Systems of Equations



*Supplementary reading.* There is a wealth of literature on iterative methods for the solution of linear systems of equations: The two books [37] and [68] offer a comprehensive treatment of the topic (the latter is available online for ETH students and staff).

Concise presentations can be found in [63, Ch. 4] and [15, Ch. 13].

---

Learning outcomes:

- Understanding when and why iterative solution of linear systems of equations may be preferred to direct solvers based on Gaussian elimination.
- 

= A class of **iterative methods** ( $\rightarrow$  Section 2.1) for approximate solution of large linear systems of equations  $\mathbf{Ax} = \mathbf{b}$ ,  $\mathbf{A} \in \mathbb{K}^{n,n}$ .

BUT, we have reliable *direct* methods (Gauss elimination  $\rightarrow$  Section 1.6.2, LU-factorization  $\rightarrow$  § 1.6.42, QR-factorization  $\rightarrow$  ??) that provide an (apart from roundoff errors) exact solution with a *finite* number of elementary operations!

- Alas, direct elimination may **not** be **feasible**, or may be grossly inefficient, because
- it may be too expensive (e.g. for  $\mathbf{A}$  too large, sparse),  $\rightarrow$  (1.6.39),
  - inevitable fill-in may exhaust main memory,
  - the system matrix may be available only as procedure  $y = \text{evalA}(x) \leftrightarrow \mathbf{y} = \mathbf{Ax}$

### Contents

8.1	Descent Methods [63, Sect. 4.3.3]	503
8.1.1	Quadratic minimization context	503
8.1.2	Abstract steepest descent	504
8.1.3	Gradient method for s.p.d. linear system of equations	505
8.1.4	Convergence of the gradient method	507
8.2	Conjugate gradient method (CG) [42, Ch. 9], [15, Sect. 13.4], [63, Sect. 4.3.4]	510
8.2.1	Krylov spaces	511
8.2.2	Implementation of CG	512
8.2.3	Convergence of CG	515
8.3	Preconditioning [15, Sect. 13.5], [42, Ch. 10], [63, Sect. 4.3.5]	521

8.4 Survey of Krylov Subspace Methods . . . . .	527
8.4.1 Minimal residual methods . . . . .	527
8.4.2 Iterations with short recursions [63, Sect. 4.5] . . . . .	528

## 8.1 Descent Methods [63, Sect. 4.3.3]

Focus:

Linear system of equations  $\mathbf{Ax} = \mathbf{b}$ ,  $\mathbf{A} \in \mathbb{R}^{n,n}$ ,  $\mathbf{b} \in \mathbb{R}^n$ ,  $n \in \mathbb{N}$  given,  
with **symmetric positive definite** (s.p.d.,  $\rightarrow$  Def. 1.1.8) system matrix  $\mathbf{A}$



$\mathbf{A}$ -inner product  $(\mathbf{x}, \mathbf{y}) \mapsto \mathbf{x}^\top \mathbf{A} \mathbf{y} \Rightarrow$  “ $\mathbf{A}$ -geometry”

**Definition 8.1.1. Energy norm**  $\rightarrow$  [42, Def. 9.1]

A s.p.d. matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$  induces an **energy norm**

$$\|\mathbf{x}\|_A := (\mathbf{x}^\top \mathbf{A} \mathbf{x})^{1/2}, \quad \mathbf{x} \in \mathbb{R}^n.$$

**Remark 8.1.2 (Krylov methods for complex s.p.d. system matrices)**

In this chapter, for the sake of simplicity, we restrict ourselves to  $\mathbb{K} = \mathbb{R}$ .

However, the (conjugate) gradient methods introduced below also work for LSE  $\mathbf{Ax} = \mathbf{b}$  with  $\mathbf{A} \in \mathbb{C}^{n,n}$ ,  $\mathbf{A} = \mathbf{A}^H$  s.p.d. when  $^\top$  is replaced with  $^H$  (Hermitian transposed). Then, all theoretical statements remain valid unaltered for  $\mathbb{K} = \mathbb{C}$ .

### 8.1.1 Quadratic minimization context

**Lemma 8.1.3. S.p.d. LSE and quadratic minimization problem** [15, (13.37)]

A LSE with  $\mathbf{A} \in \mathbb{R}^{n,n}$  s.p.d. and  $\mathbf{b} \in \mathbb{R}^n$  is equivalent to a minimization problem:

$$\mathbf{Ax} = \mathbf{b} \Leftrightarrow \mathbf{x} = \arg \min_{\mathbf{y} \in \mathbb{R}^n} J(\mathbf{y}), \quad J(\mathbf{y}) := \frac{1}{2} \mathbf{y}^\top \mathbf{A} \mathbf{y} - \mathbf{b}^\top \mathbf{y}. \quad (8.1.4)$$

A **quadratic functional**

*Proof.* If  $\mathbf{x}^* := \mathbf{A}^{-1} \mathbf{b}$  a straightforward computation using  $\mathbf{A} = \mathbf{A}^\top$  shows

$$\begin{aligned} J(\mathbf{x}) - J(\mathbf{x}^*) &= \frac{1}{2} \mathbf{x}^\top \mathbf{A} \mathbf{x} - \mathbf{b}^\top \mathbf{x} - \frac{1}{2} (\mathbf{x}^*)^\top \mathbf{A} \mathbf{x}^* + \mathbf{b}^\top \mathbf{x}^* \\ &\stackrel{\mathbf{b}=\mathbf{A}\mathbf{x}^*}{=} \frac{1}{2} \mathbf{x}^\top \mathbf{A} \mathbf{x} - (\mathbf{x}^*)^\top \mathbf{A} \mathbf{x} + \frac{1}{2} (\mathbf{x}^*)^\top \mathbf{A} \mathbf{x}^* \\ &= \frac{1}{2} \|\mathbf{x} - \mathbf{x}^*\|_A^2. \end{aligned} \quad (8.1.5)$$

Then the assertion follows from the properties of the energy norm.  $\square$

### Example 8.1.6 (Quadratic functional in 2D)

Plot of  $J$  from (8.1.4) for  $\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ ,  $\mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ .

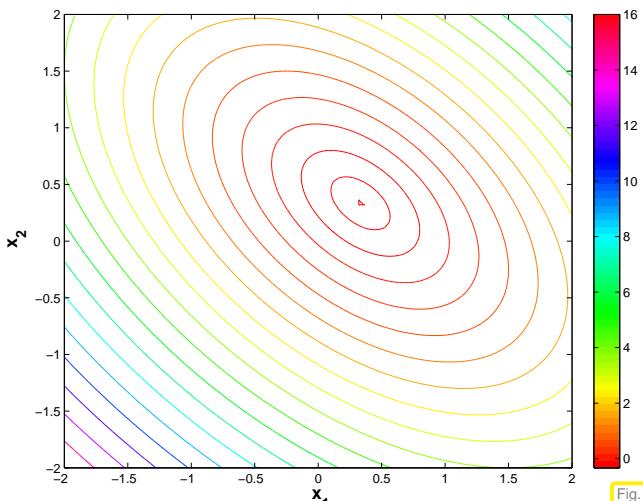


Fig. 269

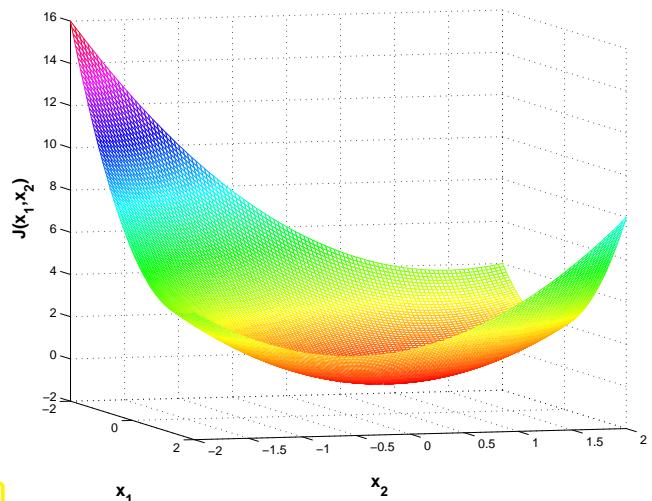
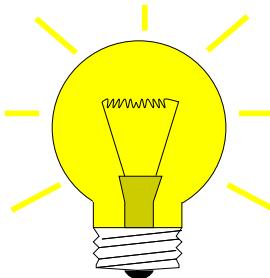


Fig. 270

Level lines of quadratic functionals with s.p.d.  $\mathbf{A}$  are (hyper)ellipses



Algorithmic idea: (Lemma 8.1.3  $\Rightarrow$ ) Solve  $\mathbf{Ax} = \mathbf{b}$  iteratively by successive solution of *simpler* minimization problems

## 8.1.2 Abstract steepest descent

Task:

Given continuously differentiable  $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}$ ,  
find minimizer  $\mathbf{x}^* \in D$ :  $\mathbf{x}^* = \underset{\mathbf{x} \in D}{\operatorname{argmin}} F(\mathbf{x})$

Note that a minimizer need not exist, if  $F$  is not bounded from below (e.g.,  $F(x) = x^3$ ,  $x \in \mathbb{R}$ , or  $F(x) = \log x$ ,  $x > 0$ ), or if  $D$  is open (e.g.,  $F(x) = \sqrt{x}$ ,  $x > 0$ ).

The existence of a minimizer is guaranteed if  $F$  is bounded from below and  $D$  is closed ( $\rightarrow$  Analysis).

The most natural iteration:

### (8.1.7) Steepest descent (ger.: steilster Abstieg)

```

Initial guess  $\mathbf{x}^{(0)} \in D$ ,  $k = 0$ 
repeat
   $\mathbf{d}_k := -\mathbf{grad} F(\mathbf{x}^{(k)})$ 
   $t^* := \underset{t \in \mathbb{R}}{\operatorname{argmin}} F(\mathbf{x}^{(k)} + t\mathbf{d}_k)$  (line search)
   $\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + t^*\mathbf{d}_k$ 
   $k := k + 1$ 
until  $\left( \begin{array}{l} \|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\| \leq \tau_{\text{rel}} \|\mathbf{x}^{(k)}\| \\ \|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\| \leq \tau_{\text{abs}} \end{array} \right)$  or

```

- \*  $\mathbf{d}_k \hat{=} \text{direction of steepest descent}$
- \* linear search  $\hat{=} 1\text{D minimization}$ : use Newton's method ( $\rightarrow$  Section 2.3.2.1) on derivative
- \* correction based a posteriori termination criterion, see Section 2.1.2 for a discussion.  
( $\tau \hat{=} \text{prescribed tolerance}$ )

The **gradient** ( $\rightarrow$  [77, Kapitel 7])

$$\mathbf{grad} F(\mathbf{x}) = \begin{bmatrix} \frac{\partial F}{\partial x_1} \\ \vdots \\ \frac{\partial F}{\partial x_n} \end{bmatrix} \in \mathbb{R}^n \quad (8.1.8)$$

provides the direction of **local** steepest ascent/descent of  $F$

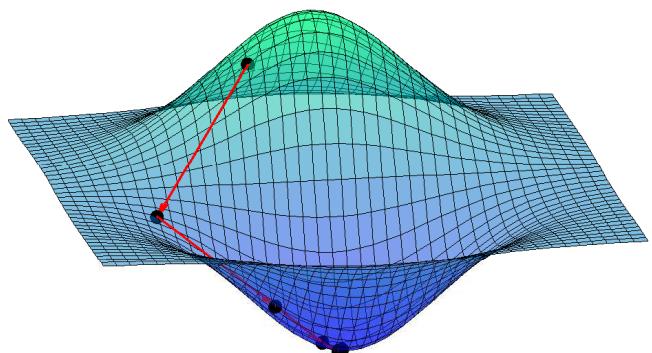


Fig. 271

Of course this very algorithm can encounter plenty of difficulties:

- iteration may get stuck in a *local minimum*,
- iteration may diverge or lead out of  $D$ ,
- line search may not be feasible.

### 8.1.3 Gradient method for s.p.d. linear system of equations

However, for the quadratic minimization problem (8.1.4) § 8.1.7 will converge:

("Geometric intuition", see Fig. 269: quadratic functional  $J$  with s.p.d.  $\mathbf{A}$  has unique global minimum,  $\mathbf{grad} J \neq 0$  away from minimum, pointing towards it.)

Adaptation: steepest descent algorithm § 8.1.7 for quadratic minimization problem (8.1.4), see [63, Sect. 7.2.4]:

$$F(\mathbf{x}) := J(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{b}^\top \mathbf{x} \Rightarrow \mathbf{grad} J(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}. \quad (8.1.9)$$

This follows from  $\mathbf{A} = \mathbf{A}^\top$ , the componentwise expression

$$J(\mathbf{x}) = \frac{1}{2} \sum_{i,j=1}^n a_{ij}x_i x_j - \sum_{i=1}^n b_i x_i$$

and the definition (8.1.8) of the gradient.

➤ For the descent direction in § 8.1.7 applied to the minimization of  $J$  from (8.1.4) holds

$$\mathbf{d}_k = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)} =: \mathbf{r}_k \quad \text{the residual} \ (\rightarrow \text{Def. 1.6.72}) \text{ for } \mathbf{x}^{(k-1)}.$$

§ 8.1.7 for  $F = J$  from (8.1.4): function to be minimized in line search step:

$$\varphi(t) := J(\mathbf{x}^{(k)} + t\mathbf{d}_k) = J(\mathbf{x}^{(k)}) + t\mathbf{d}_k^\top(\mathbf{A}\mathbf{x}^{(k)} - \mathbf{b}) + \frac{1}{2}t^2\mathbf{d}_k^\top\mathbf{A}\mathbf{d}_k \rightarrow \text{a parabola!}$$

$$\frac{d\varphi}{dt}(t^*) = 0 \Leftrightarrow t^* = \frac{\mathbf{d}_k^\top\mathbf{d}_k}{\mathbf{d}_k^\top\mathbf{A}\mathbf{d}_k} \quad (\text{unique minimizer}). \quad (8.1.10)$$

Note:  $\mathbf{d}_k = 0 \Leftrightarrow \mathbf{A}\mathbf{x}^{(k)} = \mathbf{b}$  (solution found!)

Note:  $\mathbf{A}$  s.p.d. ( $\rightarrow$  Def. 1.1.8)  $\Rightarrow \mathbf{d}_k^\top\mathbf{A}\mathbf{d}_k > 0$ , if  $\mathbf{d}_k \neq 0$

►  $\varphi(t)$  is a parabola that is bounded from below (upward opening)

Based on (8.1.9) and (8.1.10) we obtain the following steepest descent method for the minimization problem (8.1.4):

Steepest descent iteration = gradient method for LSE  $\mathbf{Ax} = \mathbf{b}$ ,  $\mathbf{A} \in \mathbb{R}^{n,n}$  s.p.d.,  $\mathbf{b} \in \mathbb{R}^n$ :

### (8.1.11) Gradient method for s.p.d. LSE

```

Initial guess  $\mathbf{x}^{(0)} \in \mathbb{R}^n, k = 0$ 
 $\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$ 
repeat
   $t^* := \frac{\mathbf{r}_k^\top \mathbf{r}_k}{\mathbf{r}_k^\top \mathbf{A}\mathbf{r}_k}$ 
   $\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + t^* \mathbf{r}_k$ 
   $\mathbf{r}_{k+1} := \mathbf{r}_k - t^* \mathbf{A}\mathbf{r}_k$ 
   $k := k + 1$ 
until  $\left( \begin{array}{l} \|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\| \leq \tau_{\text{rel}} \|\mathbf{x}^{(k)}\| \\ \|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\| \leq \tau_{\text{abs}} \end{array} \right)$  or

```

### MATLAB-code 8.1.12: gradient method for $\mathbf{Ax} = \mathbf{b}$ , $\mathbf{A}$ s.p.d.

```

1 function x =
  gradit(A,b,x,rtol,atol,maxit)
2 r = b-A*x; % residual →
  Def. 1.6.72
3 for k=1:maxit
4   p = A*x;
5   ts = (r'*r) / (r'*p); % cf. (8.1.10)
6   x = x + ts*r;
7   cn = (abs(ts)*norm(r)); % norm of
  correction
8   if ((cn < tol*norm(x)) || 
9     (cn < atol))
10    return; end
11   r = r - ts*p; %
12 end

```

Recursion for residuals, see Line 11 of Code 8.1.12:

$$\mathbf{r}_{k+1} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k+1)} = \mathbf{b} - \mathbf{A}(\mathbf{x}^{(k)} + t^* \mathbf{r}_k) = \mathbf{r}_k - t^* \mathbf{A}\mathbf{r}_k. \quad (8.1.13)$$

One step of gradient method involves

- \* A single matrix×vector product with  $\mathbf{A}$ ,
- \* 2 AXPY-operations ( $\rightarrow$  Section 1.3.2) on vectors of length  $n$ ,
- \* 2 dot products in  $\mathbb{R}^n$ .

Computational cost (per step) =  $\text{cost}(\text{matrix} \times \text{vector}) + O(n)$

- If  $\mathbf{A} \in \mathbb{R}^{n,n}$  is a sparse matrix ( $\rightarrow ??$ ) with “ $\mathcal{O}(n)$  nonzero entries”, and the data structures allow to perform the matrix  $\times$  vector product with a computational effort  $\mathcal{O}(n)$ , then a single step of the gradient method costs  $\mathcal{O}(n)$  elementary operations.
- Gradient method of § 8.1.11 only needs  $\mathbf{A} \times$  vector in procedural form  $\mathbf{y} = \text{evalA}(\mathbf{x})$ .

### 8.1.4 Convergence of the gradient method

#### Example 8.1.14 (Gradient method in 2D)

S.p.d. matrices  $\in \mathbb{R}^{2,2}$ :

$$\mathbf{A}_1 = \begin{bmatrix} 1.9412 & -0.2353 \\ -0.2353 & 1.0588 \end{bmatrix}, \quad \mathbf{A}_2 = \begin{bmatrix} 7.5353 & -1.8588 \\ -1.8588 & 0.5647 \end{bmatrix}$$

Eigenvalues:  $\sigma(\mathbf{A}_1) = \{1, 2\}$ ,  $\sigma(\mathbf{A}_2) = \{0.1, 8\}$

☞ notation: **spectrum** of a matrix  $\in \mathbb{K}^{n,n}$   $\sigma(\mathbf{M}) := \{\lambda \in \mathbb{C} : \lambda \text{ is eigenvalue of } \mathbf{M}\}$

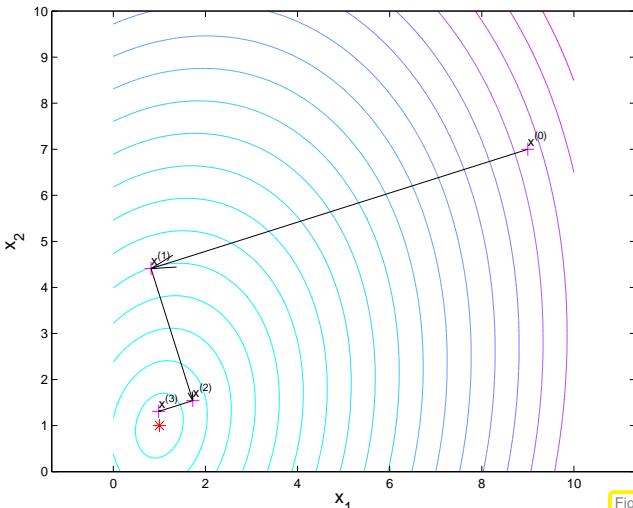


Fig. 272

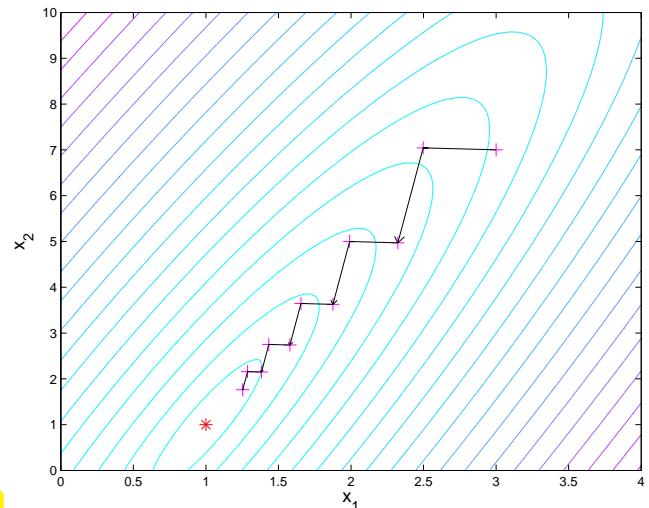


Fig. 273

iterates of § 8.1.11 for  $\mathbf{A}_1$

iterates of § 8.1.11 for  $\mathbf{A}_2$

Recall theorem on principal axis transformation: every real *symmetric* matrix can be diagonalized by *orthogonal* similarity transformations, see Cor. 7.1.9, [59, Thm. 7.8], [34, Satz 9.15],

$$\mathbf{A} = \mathbf{A}^\top \in \mathbb{R}^{n,n} \Rightarrow \exists \mathbf{Q} \in \mathbb{R}^{n,n} \text{ orthogonal: } \mathbf{A} = \mathbf{Q} \mathbf{D} \mathbf{Q}^\top, \quad \mathbf{D} = \text{diag}(d_1, \dots, d_n) \in \mathbb{R}^{n,n} \text{ diagonal}. \quad (8.1.15)$$

$$J(\mathbf{Q}\hat{\mathbf{y}}) = \frac{1}{2}\hat{\mathbf{y}}^\top \mathbf{D}\hat{\mathbf{y}} - \underbrace{(\mathbf{Q}^\top \mathbf{b})^\top}_{=: \hat{\mathbf{b}}^\top} \hat{\mathbf{y}} = \frac{1}{2} \sum_{i=1}^n d_i \hat{y}_i^2 - \hat{b}_i \hat{y}_i.$$

Hence, a rigid transformation (rotation, reflection) maps the level surfaces of  $J$  from (8.1.4) to ellipses with principal axes  $d_i$ . As  $\mathbf{A}$  s.p.d.  $d_i > 0$  is guaranteed.

Observations:

- Larger spread of spectrum leads to more elongated ellipses as level lines  $\Rightarrow$  slower convergence of gradient method, see Fig. 273.

- Orthogonality of successive residuals  $\mathbf{r}_k, \mathbf{r}_{k+1}$ .

Clear from definition of § 8.1.11:

$$\mathbf{r}_k^\top \mathbf{r}_{k+1} = \mathbf{r}_k^\top \mathbf{r}_k - \mathbf{r}_k^\top \frac{\mathbf{r}_k^\top \mathbf{r}_k}{\mathbf{r}_k^\top \mathbf{A} \mathbf{r}_k} \mathbf{A} \mathbf{r}_k = 0. \quad (8.1.16)$$

### Example 8.1.17 (Convergence of gradient method)

Convergence of gradient method for diagonal matrices,  $\mathbf{x}^* = [1, \dots, 1]^\top$ ,  $\mathbf{x}^{(0)} = 0$ :

```

1 d = 1:0.01:2; A1 = diag(d);
2 d = 1:0.1:11; A2 = diag(d);
3 d = 1:1:101; A3 = diag(d);

```

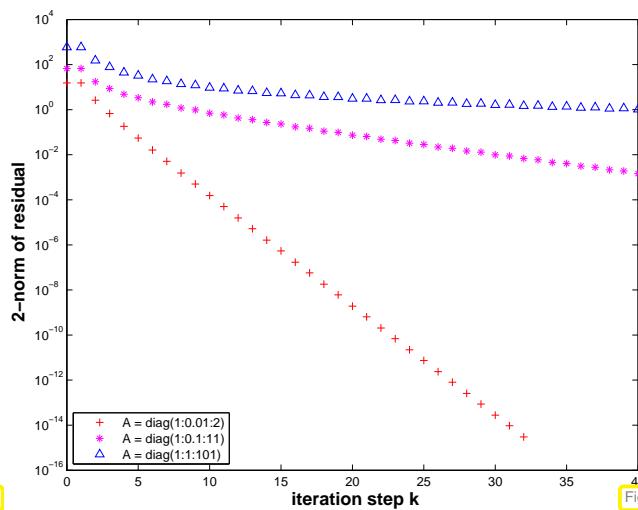


Fig. 274

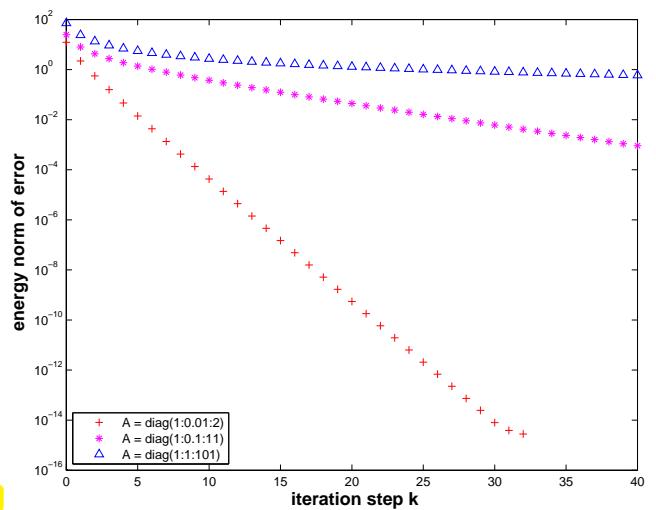


Fig. 275

Note: To study convergence it is *sufficient to consider diagonal matrices*, because

1. (8.1.15): for every  $\mathbf{A} \in \mathbb{R}^{n,n}$  with  $\mathbf{A}^\top = \mathbf{A}$  there is an orthogonal matrix  $\mathbf{Q} \in \mathbb{R}^{n,n}$  such that  $\mathbf{A} = \mathbf{Q}^\top \mathbf{D} \mathbf{Q}$  with a diagonal matrix  $\mathbf{D}$  (principal axis transformation),  $\rightarrow$  Cor. 7.1.9, [59, Thm. 7.8], [34, Satz 9.15],
2. when applying the gradient method § 8.1.11 to both  $\mathbf{A}\mathbf{x} = \mathbf{b}$  and  $\mathbf{D}\tilde{\mathbf{x}} = \tilde{\mathbf{b}} := \mathbf{Q}\mathbf{b}$ , then the iterates  $\mathbf{x}^{(k)}$  and  $\tilde{\mathbf{x}}^{(k)}$  are related by  $\mathbf{Q}\mathbf{x}^{(k)} = \tilde{\mathbf{x}}^{(k)}$ .

With  $\tilde{\mathbf{r}}_k := \mathbf{Q}\mathbf{r}_k$ ,  $\tilde{\mathbf{x}}^{(k)} := \mathbf{Q}\mathbf{x}^{(k)}$ , using  $\mathbf{Q}^\top \mathbf{Q} = \mathbf{I}$ :

Initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$ , $k = 0$
$\mathbf{r}_0 := \mathbf{b} - \mathbf{Q}^\top \mathbf{D} \mathbf{Q} \mathbf{x}^{(0)}$
<b>repeat</b>
$t^* := \frac{\mathbf{r}_k^\top \mathbf{Q}^\top \mathbf{Q} \mathbf{r}_k}{\mathbf{r}_k^\top \mathbf{Q}^\top \mathbf{D} \mathbf{Q} \mathbf{r}_k}$
$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + t^* \mathbf{r}_k$
$\mathbf{r}_{k+1} := \mathbf{r}_k - t^* \mathbf{Q}^\top \mathbf{D} \mathbf{Q} \mathbf{r}_k$
$k := k + 1$
<b>until</b> $\ \mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\  \leq \tau \ \mathbf{x}^{(k)}\ $

Initial guess $\tilde{\mathbf{x}}^{(0)} \in \mathbb{R}^n$ , $k = 0$
$\tilde{\mathbf{r}}_0 := \tilde{\mathbf{b}} - \mathbf{D}\tilde{\mathbf{x}}^{(0)}$
<b>repeat</b>
$t^* := \frac{\tilde{\mathbf{r}}_k^\top \tilde{\mathbf{r}}_k}{\tilde{\mathbf{r}}_k^\top \mathbf{D} \tilde{\mathbf{r}}_k}$
$\tilde{\mathbf{x}}^{(k+1)} := \tilde{\mathbf{x}}^{(k)} + t^* \tilde{\mathbf{r}}_k$
$\tilde{\mathbf{r}}_{k+1} := \tilde{\mathbf{r}}_k - t^* \mathbf{D} \tilde{\mathbf{r}}_k$
$k := k + 1$
<b>until</b> $\ \tilde{\mathbf{x}}^{(k)} - \tilde{\mathbf{x}}^{(k-1)}\  \leq \tau \ \tilde{\mathbf{x}}^{(k)}\ $

Observation:

- \* linear convergence ( $\rightarrow$  Def. 2.1.9), see also Rem. 2.1.13
- \* rate of convergence increases ( $\leftrightarrow$  speed of convergence decreases) with spread of spectrum of  $\mathbf{A}$

Impact of distribution of diagonal entries ( $\leftrightarrow$  eigenvalues) of (diagonal matrix)  $\mathbf{A}$

$(\mathbf{b} = \mathbf{x}^* = \mathbf{0}, \mathbf{x}_0 = \cos((1:n)'),)$

Test matrix #1:  $\mathbf{A} = \text{diag}(\mathbf{d}); \mathbf{d} = (1:100);$

Test matrix #2:  $\mathbf{A} = \text{diag}(\mathbf{d}); \mathbf{d} = [1 + (0:97)/97, 50, 100];$

Test matrix #3:  $\mathbf{A} = \text{diag}(\mathbf{d}); \mathbf{d} = [1 + (0:49)*0.05, 100 - (0:49)*0.05];$

Test matrix #4: eigenvalues exponentially dense at 1

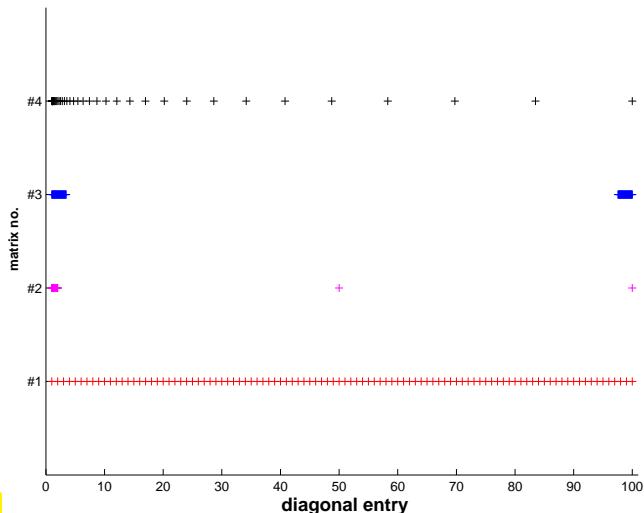
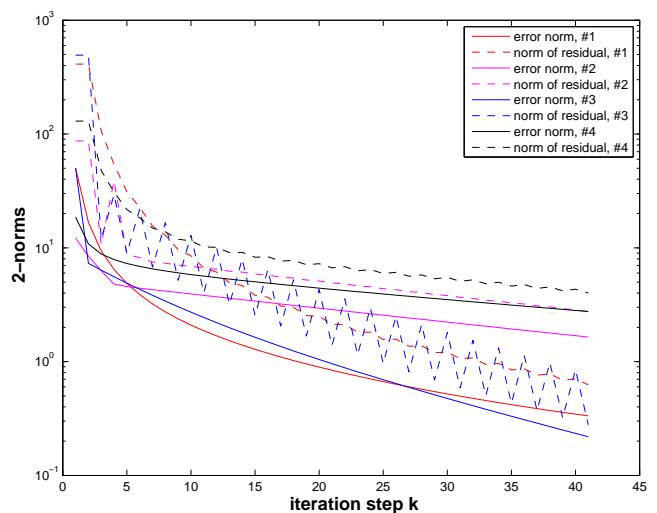


Fig. 276



Observation: Matrices #1, #2 & #4  $\gg$  little impact of distribution of eigenvalues on *asymptotic* convergence (exception: matrix #2)

Theory [35, Sect. 9.2.2], [63, Sect. 7.2.4]:

### Theorem 8.1.18. Convergence of gradient method/steepest descent

The iterates of the gradient method of § 8.1.11 satisfy

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\|_A \leq L \|\mathbf{x}^{(k)} - \mathbf{x}^*\|_A, \quad L := \frac{\text{cond}_2(\mathbf{A}) - 1}{\text{cond}_2(\mathbf{A}) + 1},$$

that is, the iteration converges at least linearly ( $\rightarrow$  Def. 2.1.9) w.r.t. energy norm ( $\rightarrow$  Def. 8.1.1).

notation:  $\text{cond}_2(\mathbf{A}) \triangleq$  condition number ( $\rightarrow$  Def. 1.6.15) of  $\mathbf{A}$  induced by 2-norm

### Remark 8.1.19 (2-norm from eigenvalues $\rightarrow$ [34, Sect. 10.6], [59, Sect. 7.4])

$$\begin{aligned} \mathbf{A} = \mathbf{A}^\top \Rightarrow \quad & \|\mathbf{A}\|_2 = \max(|\sigma(\mathbf{A})|), \\ & \|\mathbf{A}^{-1}\|_2 = \min(|\sigma(\mathbf{A})|)^{-1}, \text{ if } \mathbf{A} \text{ regular.} \end{aligned} \tag{8.1.20}$$

$$\mathbf{A} = \mathbf{A}^\top \Rightarrow \quad \text{cond}_2(\mathbf{A}) = \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})}, \quad \text{where} \quad \begin{aligned} \lambda_{\max}(\mathbf{A}) &:= \max(|\sigma(\mathbf{A})|), \\ \lambda_{\min}(\mathbf{A}) &:= \min(|\sigma(\mathbf{A})|). \end{aligned} \tag{8.1.21}$$

other notation  $\kappa(\mathbf{A}) := \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})} \hat{=} \text{spectral condition number of } \mathbf{A}$

(for general  $\mathbf{A}$ :  $\lambda_{\max}(\mathbf{A})/\lambda_{\min}(\mathbf{A})$  largest/smallest eigenvalue *in modulus*)

These results are an immediate consequence of the fact that

$$\forall \mathbf{A} \in \mathbb{R}^{n,n}, \quad \mathbf{A}^T = \mathbf{A} \quad \exists \mathbf{U} \in \mathbb{R}^{n,n}, \quad \mathbf{U}^{-1} = \mathbf{U}^T: \quad \mathbf{U}^T \mathbf{A} \mathbf{U} \quad \text{is diagonal,}$$

see (8.1.15), Cor. 7.1.9, [59, Thm. 7.8], [34, Satz 9.15].

Please note that for general regular  $\mathbf{M} \in \mathbb{R}^{n,n}$  we *cannot* expect  $\text{cond}_2(\mathbf{M}) = \kappa(\mathbf{M})$ .

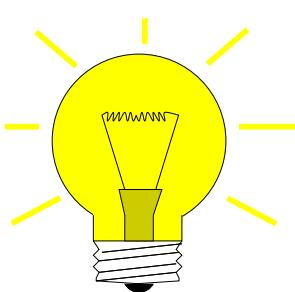
## 8.2 Conjugate gradient method (CG) [42, Ch. 9], [15, Sect. 13.4], [63, Sect. 4.3.4]

Again we consider a linear system of equations  $\mathbf{Ax} = \mathbf{b}$  with s.p.d. ( $\rightarrow$  Def. 1.1.8) system matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$  and given  $\mathbf{b} \in \mathbb{R}^n$ .

Liability of gradient method of Section 8.1.3:

NO MEMORY

1D line search in § 8.1.11 is oblivious of former line searches, which rules out reuse of information gained in previous steps of the iteration. This is a typical drawback of 1-point iterative methods.



Idea:

Replace linear search with **subspace correction**

Given:

- \* initial guess  $\mathbf{x}^{(0)}$
- \* nested subspaces  $U_1 \subset U_2 \subset U_3 \subset \dots \subset U_n = \mathbb{R}^n, \dim U_k = k$

$$\mathbf{x}^{(k)} := \underset{\mathbf{x} \in U_k + \mathbf{x}^{(0)}}{\operatorname{argmin}} J(\mathbf{x}), \quad (8.2.1)$$

quadratic functional from (8.1.4)

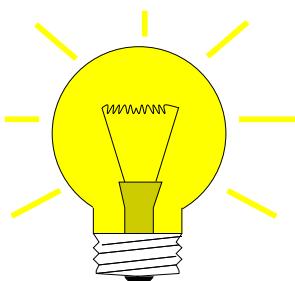
Note: Once the subspaces  $U_k$  and  $\mathbf{x}^{(0)}$  are fixed, the iteration (8.2.1) is well defined, because  $J|_{U_k + \mathbf{x}^{(0)}}$  always possess a unique minimizer.

Obvious (from Lemma 8.1.3):

$$\mathbf{x}^{(n)} = \mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b}$$

Thanks to (8.1.5), definition (8.2.1) ensures:

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\|_A \leq \|\mathbf{x}^{(k)} - \mathbf{x}^*\|_A$$



How to find suitable subspaces  $U_k$ ?

Idea:

$U_{k+1} \leftarrow U_k + \text{"local steepest descent direction"}$

given by  $-\mathbf{grad} J(\mathbf{x}^{(k)}) = \mathbf{b} - \mathbf{Ax}^{(k)} = \mathbf{r}_k$  (residual  $\rightarrow$  Def. 1.6.72)

$$U_{k+1} = \text{Span}\{U_k, \mathbf{r}_k\}, \quad \mathbf{x}^{(k)} \text{ from (8.2.1).} \quad (8.2.2)$$

Obvious:  $\mathbf{r}_k = 0 \Rightarrow \mathbf{x}^{(k)} = \mathbf{x}^* := \mathbf{A}^{-1}\mathbf{b}$  done ✓

**Lemma 8.2.3.**  $\mathbf{r}_k \perp U_k$

With  $\mathbf{x}^{(k)}$  according to (8.2.1),  $U_k$  from (8.2.2) the residual  $\mathbf{r}_k := \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$  satisfies

$$\mathbf{r}_k^\top \mathbf{u} = 0 \quad \forall \mathbf{u} \in U_k \quad ("r_k \perp U_k").$$

Geometric consideration: since  $\mathbf{x}^{(k)}$  is the minimizer of  $J$  over the affine space  $U_k + \mathbf{x}^{(0)}$ , the projection of the steepest descent direction  $\mathbf{grad} J(\mathbf{x}^{(k)})$  onto  $U_k$  has to vanish:

$$\mathbf{x}^{(k)} := \underset{\mathbf{x} \in U_k + \mathbf{x}^{(0)}}{\operatorname{argmin}} J(x) \Rightarrow \mathbf{grad} J(\mathbf{x}^{(k)}) \perp U_k. \quad (8.2.4)$$

*Proof.* Consider

$$\psi(t) = J(\mathbf{x}^{(k)} + t\mathbf{u}), \quad \mathbf{u} \in U_k, \quad t \in \mathbb{R}.$$

By (8.2.1),  $t \mapsto \psi(t)$  has a global minimum in  $t = 0$ , which implies

$$\frac{d\psi}{dt}(0) = \mathbf{grad} J(\mathbf{x}^{(k)})^\top \mathbf{u} = (\mathbf{A}\mathbf{x}^{(k)} - \mathbf{b})^\top \mathbf{u} = 0.$$

Since  $\mathbf{u} \in U_k$  was arbitrary, the lemma is proved. □

**Corollary 8.2.5.**

If  $\mathbf{r}_l \neq 0$  for  $l = 0, \dots, k$ ,  $k \leq n$ , then  $\{\mathbf{r}_0, \dots, \mathbf{r}_k\}$  is an *orthogonal basis* of  $U_k$ .

Lemma 8.2.3 also implies that, if  $U_0 = \{0\}$ , then  $\dim U_k = k$  as long as  $\mathbf{x}^{(k)} \neq \mathbf{x}^*$ , that is, before we have converged to the exact solution.

(8.2.1) and (8.2.2) define the **conjugate gradient method** (CG) for the iterative solution of  $\mathbf{Ax} = \mathbf{b}$   
(hailed as a “top ten algorithm” of the 20th century, SIAM News, 33(4))

## 8.2.1 Krylov spaces

**Definition 8.2.6. Krylov space**

For  $\mathbf{A} \in \mathbb{R}^{n,n}$ ,  $\mathbf{z} \in \mathbb{R}^n$ ,  $\mathbf{z} \neq 0$ , the  $l$ -th **Krylov space** is defined as

$$\mathcal{K}_l(\mathbf{A}, \mathbf{z}) := \text{Span}\{\mathbf{z}, \mathbf{Az}, \dots, \mathbf{A}^{l-1}\mathbf{z}\}.$$

Equivalent definition:

$$\mathcal{K}_l(\mathbf{A}, \mathbf{z}) = \{ p(\mathbf{A})\mathbf{z} : p \text{ polynomial of degree } \leq l \}$$

### Lemma 8.2.7.

The subspaces  $U_k \subset \mathbb{R}^n$ ,  $k \geq 1$ , defined by (8.2.1) and (8.2.2) satisfy

$$U_k = \text{Span}\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \dots, \mathbf{A}^{k-1}\mathbf{r}_0\} = \mathcal{K}_k(\mathbf{A}, \mathbf{r}_0),$$

where  $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$  is the initial residual.

*Proof.* (by induction) Obviously  $\mathbf{A}\mathcal{K}_k(\mathbf{A}, \mathbf{r}_0) \subset \mathcal{K}_{k+1}(\mathbf{A}, \mathbf{r}_0)$ . In addition

$$\mathbf{r}_k = \mathbf{b} - \mathbf{A}(\mathbf{x}^{(0)} + \mathbf{z}) \quad \text{for some } \mathbf{z} \in U_k \Rightarrow \mathbf{r}_k = \underbrace{\mathbf{r}_0}_{\in \mathcal{K}_{k+1}(\mathbf{A}, \mathbf{r}_0)} - \underbrace{\mathbf{A}\mathbf{z}}_{\in \mathcal{K}_{k+1}(\mathbf{A}, \mathbf{r}_0)}.$$

Since  $U_{k+1} = \text{Span}\{U_k, \mathbf{r}_k\}$ , we obtain  $U_{k+1} \subset \mathcal{K}_{k+1}(\mathbf{A}, \mathbf{r}_0)$ . Dimensional considerations based on Lemma 8.2.3 finish the proof.  $\square$

## 8.2.2 Implementation of CG

Assume: basis  $\{\mathbf{p}_1, \dots, \mathbf{p}_l\}$ ,  $l = 1, \dots, n$ , of  $\mathcal{K}_l(\mathbf{A}, \mathbf{r})$  available

$$\mathbf{x}^{(l)} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A}, \mathbf{r}_0) \Rightarrow \text{set } \mathbf{x}^{(l)} = \mathbf{x}^{(0)} + \gamma_1 \mathbf{p}_1 + \dots + \gamma_l \mathbf{p}_l.$$

For  $\psi(\gamma_1, \dots, \gamma_l) := J(\mathbf{x}^{(0)} + \gamma_1 \mathbf{p}_1 + \dots + \gamma_l \mathbf{p}_l)$  holds

$$(8.2.1) \Leftrightarrow \frac{\partial \psi}{\partial \gamma_j} = 0, \quad j = 1, \dots, l.$$

This leads to a linear system of equations by which the coefficients  $\gamma_j$  can be computed:

$$\begin{bmatrix} \mathbf{p}_1^\top \mathbf{A} \mathbf{p}_1 & \cdots & \mathbf{p}_1^\top \mathbf{A} \mathbf{p}_l \\ \vdots & \ddots & \vdots \\ \mathbf{p}_l^\top \mathbf{A} \mathbf{p}_1 & \cdots & \mathbf{p}_l^\top \mathbf{A} \mathbf{p}_l \end{bmatrix} \begin{bmatrix} \gamma_1 \\ \vdots \\ \gamma_l \end{bmatrix} = \begin{bmatrix} \mathbf{p}_1^\top \mathbf{r} \\ \vdots \\ \mathbf{p}_l^\top \mathbf{r} \end{bmatrix}, \quad \mathbf{r} := \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}. \quad (8.2.8)$$

Great simplification, if  $\{\mathbf{p}_1, \dots, \mathbf{p}_l\}$  **A-orthogonal basis** of  $\mathcal{K}_l(\mathbf{A}, \mathbf{r})$ :  $\mathbf{p}_j^\top \mathbf{A} \mathbf{p}_i = 0$  for  $i \neq j$ .

Recall: s.p.d. **A** induces an inner product  $\geq$  concept of orthogonality [59, Sect. 4.4], [34, Sect. 6.2]. “**A**-geometry” like standard Euclidean space.

Assume: **A**-orthogonal basis  $\{\mathbf{p}_1, \dots, \mathbf{p}_n\}$  of  $\mathbb{R}^n$  available, such that

$$\text{Span}\{\mathbf{p}_1, \dots, \mathbf{p}_l\} = \mathcal{K}_l(\mathbf{A}, \mathbf{r}).$$

► (Efficient) successive computation of  $\mathbf{x}^{(l)}$  becomes possible, see [15, Lemma 13.24]  
(LSE (8.2.8) becomes diagonal !)

Input: : initial guess  $\mathbf{x}^{(0)} \in \mathbb{R}^n$

Given: :  $\mathbf{A}$ -orthogonal bases  $\{\mathbf{p}_1, \dots, \mathbf{p}_l\}$  of  $\mathcal{K}_l(\mathbf{A}, \mathbf{r}_0)$ ,  $l = 1, \dots, n$

Output: : approximate solution  $\mathbf{x}^{(l)} \in \mathbb{R}^n$  of  $\mathbf{Ax} = \mathbf{b}$

$$\begin{aligned} \mathbf{r}_0 &:= \mathbf{b} - \mathbf{Ax}^{(0)}; \\ \text{for } j = 1 \text{ to } l \text{ do } \{ &\quad \mathbf{x}^{(j)} := \mathbf{x}^{(j-1)} + \frac{\mathbf{p}_j^\top \mathbf{r}_0}{\mathbf{p}_j^\top \mathbf{Ap}_j} \mathbf{p}_j \quad \} \end{aligned} \quad (8.2.9)$$

**Task:** Efficient computation of  $\mathbf{A}$ -orthogonal vectors  $\{\mathbf{p}_1, \dots, \mathbf{p}_l\}$  spanning  $\mathcal{K}_l(\mathbf{A}, \mathbf{r}_0)$  during the CG iteration.

$\mathbf{A}$ -orthogonalities/orthogonalities  $\rightarrow$  short recursions

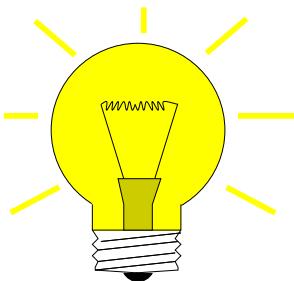
Lemma 8.2.3 implies orthogonality  $\mathbf{p}_j \perp \mathbf{r}_m := \mathbf{b} - \mathbf{Ax}^{(m)}$ ,  $1 \leq j \leq m \leq l$ . Also by  $\mathbf{A}$ -orthogonality of the  $\mathbf{p}_k$

$$\mathbf{p}_j^\top (\mathbf{b} - \mathbf{Ax}^{(m)}) = \mathbf{p}_j^\top \left( \underbrace{\mathbf{b} - \mathbf{Ax}^{(0)}}_{= \mathbf{r}_0} - \sum_{k=1}^m \frac{\mathbf{p}_k^\top \mathbf{r}_0}{\mathbf{p}_k^\top \mathbf{Ap}_k} \mathbf{Ap}_k \right) = 0. \quad (8.2.10)$$

From linear algebra we already know a way to construct orthogonal basis vectors:

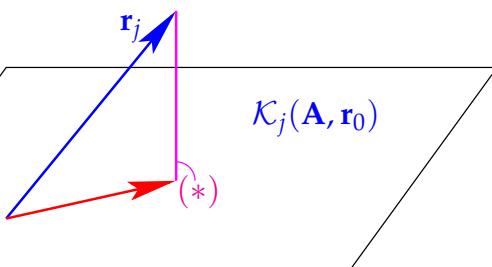
(8.2.10)  $\Rightarrow$  Idea:

Gram-Schmidt orthogonalization [59, Thm. 4.8], [34, Alg. 6.1], of residuals  $\mathbf{r}_j := \mathbf{b} - \mathbf{Ax}^{(j)}$  w.r.t.  $\mathbf{A}$ -inner product:



$$\mathbf{p}_1 := \mathbf{r}_0, \mathbf{p}_{j+1} := \underbrace{(\mathbf{b} - \mathbf{Ax}^{(j)})}_{=: \mathbf{r}_j} - \underbrace{\sum_{k=1}^j \frac{\mathbf{p}_k^\top \mathbf{Ar}_j}{\mathbf{p}_k^\top \mathbf{Ap}_k} \mathbf{p}_k}_{(*)}, \quad j = 1, \dots, l-1. \quad (8.2.11)$$

Fig. 277



Geometric interpretation of (8.2.11):

$(*) \hat{=} \text{orthogonal projection of } \mathbf{r}_j \text{ on the subspace } \text{Span}\{\mathbf{p}_1, \dots, \mathbf{p}_j\}$

### Lemma 8.2.12. Bases for Krylov spaces in CG

If they do not vanish, the vectors  $\mathbf{p}_j$ ,  $1 \leq j \leq l$ , and  $\mathbf{r}_j := \mathbf{b} - \mathbf{Ax}^{(j)}$ ,  $0 \leq j \leq l$ , from (8.2.9), (8.2.11) satisfy

- (i)  $\{\mathbf{p}_1, \dots, \mathbf{p}_l\}$  is  $\mathbf{A}$ -orthogonal basis von  $\mathcal{K}_l(\mathbf{A}, \mathbf{r}_0)$ ,
- (ii)  $\{\mathbf{r}_0, \dots, \mathbf{r}_{l-1}\}$  is orthogonal basis of  $\mathcal{K}_l(\mathbf{A}, \mathbf{r}_0)$ , cf. Cor. 8.2.5

*Proof.*  $\mathbf{A}$ -orthogonality of  $\mathbf{p}_j$  by construction, study (8.2.11).

$$\begin{aligned} (8.2.9) \& (8.2.11) \Rightarrow \mathbf{p}_{j+1} = \mathbf{r}_0 - \sum_{k=1}^j \frac{\mathbf{p}_k^\top \mathbf{r}_0}{\mathbf{p}_k^\top \mathbf{Ap}_k} \mathbf{Ap}_k - \sum_{k=1}^j \frac{\mathbf{p}_k^\top \mathbf{Ar}_j}{\mathbf{p}_k^\top \mathbf{Ap}_k} \mathbf{p}_k. \\ \Rightarrow \mathbf{p}_{j+1} &\in \text{Span}\{\mathbf{r}_0, \mathbf{p}_1, \dots, \mathbf{p}_j, \mathbf{Ap}_1, \dots, \mathbf{Ap}_j\}. \end{aligned}$$

A simple induction argument confirms (i)

$$(8.2.11) \Rightarrow \mathbf{r}_j \in \text{Span}\{\mathbf{p}_1, \dots, \mathbf{p}_{j+1}\} \quad \& \quad \mathbf{p}_j \in \text{Span}\{\mathbf{r}_0, \dots, \mathbf{r}_{j-1}\}. \quad (8.2.13)$$

►  $\text{Span}\{\mathbf{p}_1, \dots, \mathbf{p}_j\} = \text{Span}\{\mathbf{r}_0, \dots, \mathbf{r}_{j-1}\} = \mathcal{K}_l(\mathbf{A}, \mathbf{r}_0)$  . (8.2.14)

$$(8.2.10) \Rightarrow \mathbf{r}_j \perp \text{Span}\{\mathbf{p}_1, \dots, \mathbf{p}_j\} = \text{Span}\{\mathbf{r}_0, \dots, \mathbf{r}_{j-1}\}. \quad (8.2.15)$$

□

Orthogonalities from Lemma 8.2.12 ► short recursions for  $\mathbf{p}_k, \mathbf{r}_k, \mathbf{x}^{(k)}$  !

$$(8.2.10) \Rightarrow (8.2.11) \text{ collapses to } \mathbf{p}_{j+1} := \mathbf{r}_j - \frac{\mathbf{p}_j^\top \mathbf{A} \mathbf{r}_j}{\mathbf{p}_j^\top \mathbf{A} \mathbf{p}_j} \mathbf{p}_j, \quad j = 1, \dots, l.$$

recursion for residuals:

$$\begin{aligned} (8.2.9) \quad &\Rightarrow \quad \mathbf{r}_j = \mathbf{r}_{j-1} - \frac{\mathbf{p}_j^\top \mathbf{r}_0}{\mathbf{p}_j^\top \mathbf{A} \mathbf{p}_j} \mathbf{A} \mathbf{p}_j. \\ \text{Lemma 8.2.12, (i)} \quad &\Rightarrow \quad \mathbf{r}_{j-1}^H \mathbf{p}_j = \left( \mathbf{r}_0 + \sum_{k=1}^{m-1} \frac{\mathbf{r}_0^\top \mathbf{p}_k}{\mathbf{p}_k^\top \mathbf{A} \mathbf{p}_k} \mathbf{A} \mathbf{p}_k \right)^T \mathbf{p}_j = \mathbf{r}_0^\top \mathbf{p}_j. \end{aligned} \quad (8.2.16)$$

The orthogonality (8.2.16) together with (8.2.15) permits us to replace  $\mathbf{r}_0$  with  $\mathbf{r}_{j-1}$  in the actual implementation.



### (8.2.17) CG method for solving $\mathbf{Ax} = \mathbf{b}$ , $\mathbf{A}$ s.p.d. → [15, Alg. 13.27]

Input : initial guess  $\mathbf{x}^{(0)} \in \mathbb{R}^n$   
Output : approximate solution  $\mathbf{x}^{(l)} \in \mathbb{R}^n$

```

 $\mathbf{p}_1 = \mathbf{r}_0 := \mathbf{b} - \mathbf{Ax}^{(0)}$ ;
for  $j = 1$  to  $l$  do {
     $\mathbf{x}^{(j)} := \mathbf{x}^{(j-1)} + \frac{\mathbf{p}_j^\top \mathbf{r}_{j-1}}{\mathbf{p}_j^\top \mathbf{A} \mathbf{p}_j} \mathbf{p}_j$ ;
     $\mathbf{r}_j = \mathbf{r}_{j-1} - \frac{\mathbf{p}_j^\top \mathbf{r}_{j-1}}{\mathbf{p}_j^\top \mathbf{A} \mathbf{p}_j} \mathbf{A} \mathbf{p}_j$ ;
     $\mathbf{p}_{j+1} = \mathbf{r}_j - \frac{(\mathbf{A} \mathbf{p}_j)^\top \mathbf{r}_j}{\mathbf{p}_j^\top \mathbf{A} \mathbf{p}_j} \mathbf{p}_j$ ;
}
```

Input: initial guess  $\mathbf{x} \hat{=} \mathbf{x}^{(0)} \in \mathbb{R}^n$   
tolerance  $\tau > 0$   
Output: approximate solution  $\mathbf{x} \hat{=} \mathbf{x}^{(l)}$

```

 $\mathbf{p} := \mathbf{r}_0 := \mathbf{r} := \mathbf{b} - \mathbf{Ax}$ ;
for  $j = 1$  to  $l_{\max}$  do {
     $\beta := \mathbf{r}^\top \mathbf{r}$ ;
     $\mathbf{h} := \mathbf{A} \mathbf{p}$ ;
     $\alpha := \frac{\beta}{\mathbf{p}^\top \mathbf{h}}$ ;
     $\mathbf{x} := \mathbf{x} + \alpha \mathbf{p}$ ;
     $\mathbf{r} := \mathbf{r} - \alpha \mathbf{h}$ ;
    if  $\|\mathbf{r}\| \leq \tau \|\mathbf{r}_0\|$  then stop;
     $\beta := \frac{\mathbf{r}^\top \mathbf{r}}{\beta}$ ;
     $\mathbf{p} := \mathbf{r} + \beta \mathbf{p}$ ;
}
```

In CG algorithm:  $\mathbf{r}_j = \mathbf{b} - \mathbf{Ax}^{(k)}$  agrees with the residual associated with the current iterate (in exact arithmetic, cf. Ex. 8.2.21), but computation through short recursion is more efficient.

➤ We find that the CG method possesses all the algorithmic advantages of the gradient method, cf. the discussion in Section 8.1.3.

► 1 matrix  $\times$  vector product, 3 dot products, 3 **AXPY-operations** per step:  
If  $\mathbf{A}$  sparse,  $\text{nnz}(\mathbf{A}) \sim n$  ➤ computational effort  $\mathcal{O}(n)$  per step

### MATLAB-code 8.2.18: basic CG iteration for solving $\mathbf{Ax} = \mathbf{b}$ , § 8.2.17

```

1 function x = cg(evalA,b,x,tol,maxit)
2 % x supplies initial guess, maxit maximal number of CG steps
3 % evalA must pass a handle to a MATLAB function realizing A*x
4 r = b - evalA(x); rho = 1; n0 = norm(r);
5 for i = 1 : maxit
6     rho1 = rho; rho = r' * r;
7     if (i == 1), p = r;
8     else beta = rho/rho1; p = r + beta * p; end
9     q = evalA(p); alpha = rho/(p' * q);
10    x = x + alpha * p;      % update of approximate solution
11    if (norm(b-A*x) <= tol*n0) return; end % termination, see
12        Rem. 8.2.19
13    r = r - alpha * q;      % update of residual
14
15 end
```

MATLAB-function:

`x=pcg(A,b,tol,maxit,[],[],x0)` : Solve  $\mathbf{Ax} = \mathbf{b}$  with at most maxit CG steps: stop, when  $\|\mathbf{r}_l\| : \|\mathbf{r}_0\| < \text{tol}$ .

`x=pcg(Afun,b,tol,maxit,[],[],x0)`: Afun = handle to function for computing  $\mathbf{A} \times$ vector.

`[x,flag,relr,it,resv] = pcg(...)` : diagnostic information about iteration

### Remark 8.2.19 (A posteriori termination criterion for plain CG)

For any vector norm and associated matrix norm ( $\rightarrow$  Def. 1.5.71) hold (with residual  $\mathbf{r}_l := \mathbf{b} - \mathbf{Ax}^{(l)}$ )

$$\frac{1}{\text{cond}(\mathbf{A})} \frac{\|\mathbf{r}_l\|}{\|\mathbf{r}_0\|} \leq \frac{\|\mathbf{x}^{(l)} - \mathbf{x}^*\|}{\|\mathbf{x}^{(0)} - \mathbf{x}^*\|} \leq \text{cond}(\mathbf{A}) \frac{\|\mathbf{r}_l\|}{\|\mathbf{r}_0\|}. \quad (8.2.20)$$

↑  
relative decrease of iteration error

(8.2.20) can easily be deduced from the error equation  $\mathbf{A}(\mathbf{x}^{(k)} - \mathbf{x}^*) = \mathbf{r}_k$ , see Def. 1.6.72 and (1.6.75).

### 8.2.3 Convergence of CG

Note: CG is a *direct solver*, because (in exact arithmetic)  $\mathbf{x}^{(k)} = \mathbf{x}^*$  for some  $k \leq n$

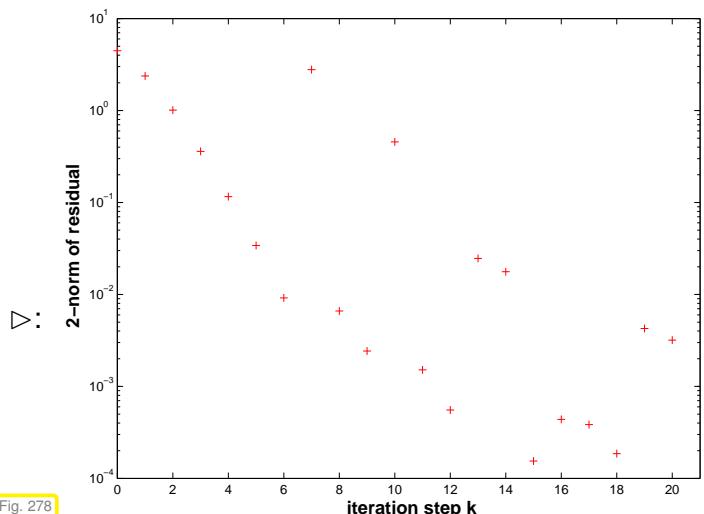
**Example 8.2.21 (Impact of roundoff errors on CG → [63, Rem. 4.3])**

Numerical experiment:  $A = \text{hilb}(20)$ ,  
 $\mathbf{x}^{(0)} = 0$ ,  $\mathbf{b} = [1, \dots, 1]^T$

Hilbert-Matrix: extremely ill-conditioned

residual norms during CG iteration

$$\mathbf{R} = [\mathbf{r}_0, \dots, \mathbf{r}^{(10)}]$$



$$\mathbf{R}^\top \mathbf{R} =$$

$$\begin{bmatrix} 1.000000 & -0.000000 & 0.000000 & -0.000000 & 0.000000 & -0.000000 & 0.016019 & -0.795816 & -0.430569 & 0.348133 \\ -0.000000 & 1.000000 & -0.000000 & 0.000000 & -0.000000 & 0.000000 & -0.012075 & 0.600068 & -0.520610 & 0.420903 \\ 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 & -0.000000 & 0.001582 & -0.078664 & 0.384453 & -0.310577 \\ -0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 & -0.000024 & 0.001218 & -0.024115 & 0.019394 \\ 0.000000 & -0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 & -0.000002 & 0.000151 & -0.000118 \\ -0.000000 & 0.000000 & -0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 & -0.000000 & 0.000000 \\ 0.016019 & -0.012075 & 0.001582 & -0.000024 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & -0.000000 & 0.000000 \\ -0.795816 & 0.600068 & -0.078664 & 0.001218 & -0.000002 & 0.000000 & -0.000000 & 1.000000 & 0.000000 & -0.000000 \\ -0.430569 & -0.520610 & 0.384453 & -0.024115 & 0.000151 & -0.000000 & -0.000000 & 0.000000 & 1.000000 & 0.000000 \\ 0.348133 & 0.420903 & -0.310577 & 0.019394 & -0.000118 & 0.000000 & 0.000000 & -0.000000 & 0.000000 & 1.000000 \end{bmatrix}$$

➤ Roundoff

- \* destroys orthogonality of residuals
- \* prevents computation of exact solution after  $n$  steps.

➤ Numerical instability (→ Def. 1.5.80) ➤ pointless to (try to) use CG as direct solver!

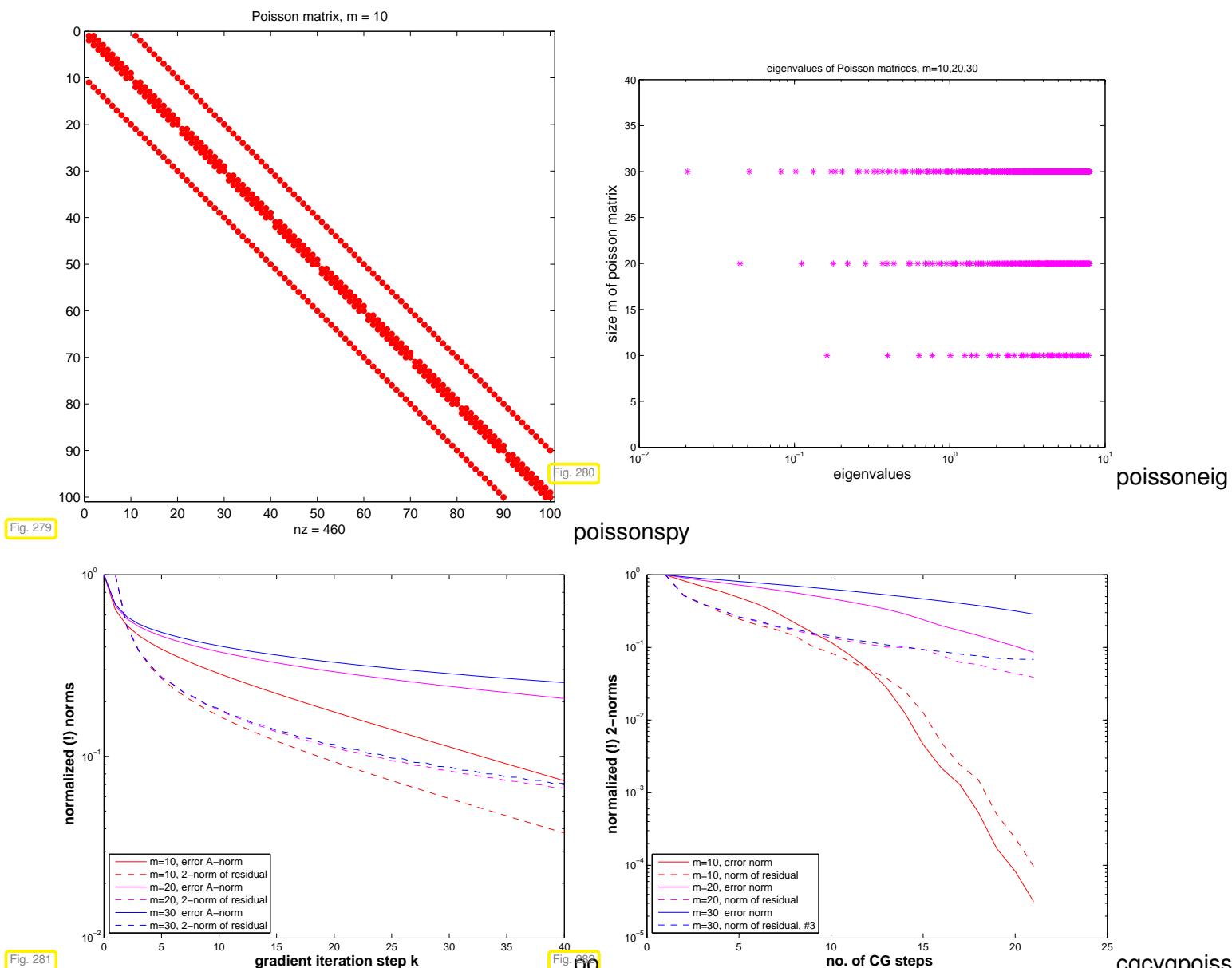
Practice: CG used for large  $n$  as *iterative solver*:  $\mathbf{x}^{(k)}$  for some  $k \ll n$  is expected to provide good approximation for  $\mathbf{x}^*$

**Example 8.2.22 (Convergence of CG as iterative solver)**

CG (Code 8.2.18) & gradient method (Code 8.1.12) for LSE with sparse s.p.d. “Poisson matrix”

```
A = gallery('poisson', m); x0 = (1:n)'; b = zeros(n, 1);
```

➤  $\mathbf{A} \in \mathbb{R}^{m^2, m^2}$



Observations:

- CG much faster than gradient method (as expected, because it has “memory”)
- Both, CG and gradient method converge more slowly for larger sizes of Poisson matrices.

### Convergence theory: [37, Sect. 9.4.3]

A simple consequence of (8.1.5) and (8.2.1):

#### Corollary 8.2.23. “Optimality” of CG iterates

Writing  $\mathbf{x}^* \in \mathbb{R}^n$  for the exact solution of  $\mathbf{Ax} = \mathbf{b}$  the CG iterates satisfy

$$\|\mathbf{x}^* - \mathbf{x}^{(l)}\|_A = \min\{\|\mathbf{y} - \mathbf{x}^*\|_A : \mathbf{y} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A}, \mathbf{r}_0)\} , \quad \mathbf{r}_0 := \mathbf{b} - \mathbf{Ax}^{(0)} .$$

This paves the way for a quantitative convergence estimate:

$$\mathbf{y} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A}, \mathbf{r}) \Leftrightarrow \mathbf{y} = \mathbf{x}^{(0)} + \mathbf{A} p(\mathbf{A})(\mathbf{x} - \mathbf{x}^{(0)}) , \quad p = \text{polynomial of degree } \leq l-1 .$$

►  $\mathbf{x} - \mathbf{y} = q(\mathbf{A})(\mathbf{x} - \mathbf{x}^{(0)}), \quad q = \text{polynomial of degree } \leq l, \quad q(0) = 1.$

$$\|\mathbf{x} - \mathbf{x}^{(l)}\|_A \leq \boxed{\min\left\{ \max_{\lambda \in \sigma(\mathbf{A})} |q(\lambda)| : q \text{ polynomial of degree } \leq l, \quad q(0) = 1 \right\}} \cdot \|\mathbf{x} - \mathbf{x}^{(0)}\|_A. \quad (8.2.24)$$

Bound this minimum for  $\lambda \in [\lambda_{\min}(\mathbf{A}), \lambda_{\max}(\mathbf{A})]$  by using suitable “polynomial candidates”

Tool: **Chebychev polynomials** ( $\rightarrow$  Section 4.1.3.1)  $\Rightarrow$  lead to the following estimate [35, Satz 9.4.2], [15, Satz 13.29]

### Theorem 8.2.25. Convergence of CG method

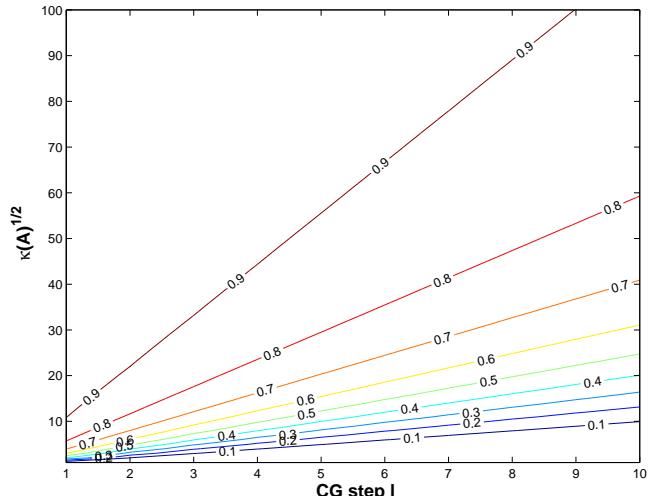
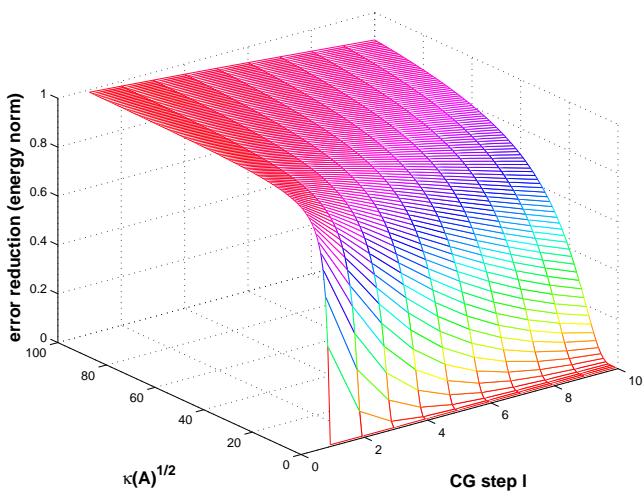
The iterates of the CG method for solving  $\mathbf{Ax} = \mathbf{b}$  (see Code 8.2.18) with  $\mathbf{A} = \mathbf{A}^\top$  s.p.d. satisfy

$$\begin{aligned} \|\mathbf{x} - \mathbf{x}^{(l)}\|_A &\leq \frac{2\left(1 - \frac{1}{\sqrt{\kappa(\mathbf{A})}}\right)^l}{\left(1 + \frac{1}{\sqrt{\kappa(\mathbf{A})}}\right)^{2l} + \left(1 - \frac{1}{\sqrt{\kappa(\mathbf{A})}}\right)^{2l}} \|\mathbf{x} - \mathbf{x}^{(0)}\|_A \\ &\leq 2\left(\frac{\sqrt{\kappa(\mathbf{A})} - 1}{\sqrt{\kappa(\mathbf{A})} + 1}\right)^l \|\mathbf{x} - \mathbf{x}^{(0)}\|_A. \end{aligned}$$

(recall:  $\kappa(\mathbf{A}) = \text{spectral condition number of } \mathbf{A}, \kappa(\mathbf{A}) = \text{cond}_2(\mathbf{A})$ )

The estimate of this theorem confirms *asymptotic linear convergence* of the CG method ( $\rightarrow$  Def. 2.1.9)  
with a rate of  $\frac{\sqrt{\kappa(\mathbf{A})} - 1}{\sqrt{\kappa(\mathbf{A})} + 1}$

Plots of bounds for error reduction (in energy norm) during CG iteration from Thm. 8.2.25:



**MATLAB-code 8.2.26: plotting theoretical bounds for CG convergence rate**

```

1 function plottheorate
2 % Python: ..../PYTHON/ConjugateGradient.py plottheorate()
3
4 [X,Y] = meshgrid(1:10,1:100); R = zeros(100,10);
5 for I=1:100
6 t = 1/I;
7 for j=1:10
8 R(I,j) = 2*(1-t)^j / ((1+t)^(2*j)+(1-t)^(2*j));
9 end
10 end
11
12 figure; view([-45,28]); mesh(X,Y,R); colormap hsv;
13 xlabel('{\bf CG step 1}','Fontsize',14);
14 ylabel('{\bf \kappa(A)^{1/2}}','Fontsize',14);
15 zlabel('{\bf error reduction (energy norm)}','Fontsize',14);
16
17 print -depsc2 '../PICTURES/theorate1.eps';
18
19 figure; [C,h] = contour(X,Y,R); clabel(C,h);
20 xlabel('{\bf CG step 1}','Fontsize',14);
21 ylabel('{\bf \kappa(A)^{1/2}}','Fontsize',14);
22
23 print -depsc2 '../PICTURES/theorate2.eps';

```

**Example 8.2.27 (Convergence rates for CG method)****MATLAB-code 8.2.28: CG for Poisson matrix**

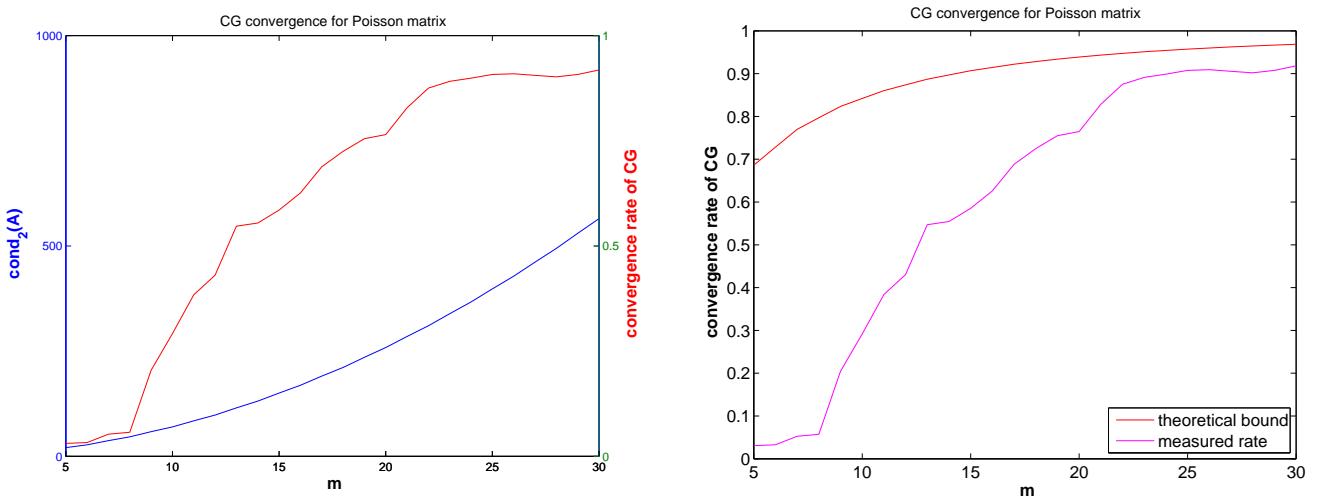
```

1 A = gallery('poisson',m); n =
2     size(A,1);
3 x0 = (1:n)'; b = ones(n,1); maxit =
4     30; tol =0;
5 [x,flag,relres,iter,resvec] =
6     pcg(A,b,tol,maxit,[],[],x0);

```

Measurement  
rate of (linear) convergence:

$$\text{rate} \approx \sqrt[10]{\frac{\|r_{30}\|_2}{\|r_{20}\|_2}}. \quad (8.2.29)$$



Justification for estimating the rate of linear convergence ( $\rightarrow$  Def. 2.1.9) of  $\|\mathbf{r}_k\|_2 \rightarrow 0$ :

$$\|\mathbf{r}_{k+1}\|_2 \approx L \|\mathbf{r}_k\|_2 \quad \Rightarrow \quad \|\mathbf{r}_{k+m}\|_2 \approx L^m \|\mathbf{r}_k\|_2.$$

### Example 8.2.30 (CG convergence and spectrum $\rightarrow$ Ex. 8.1.17)

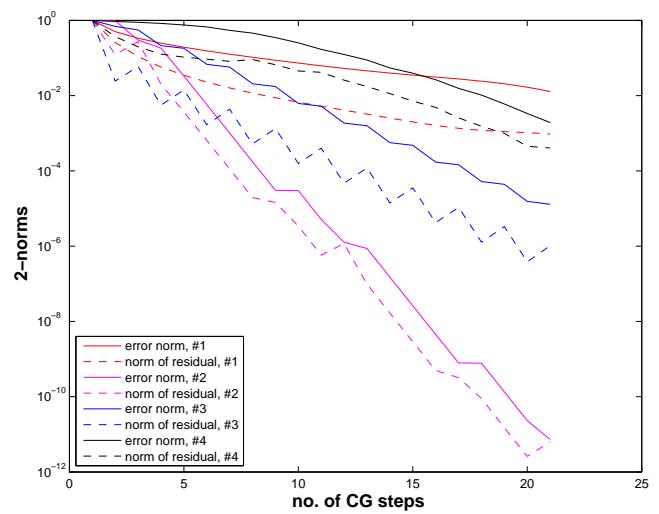
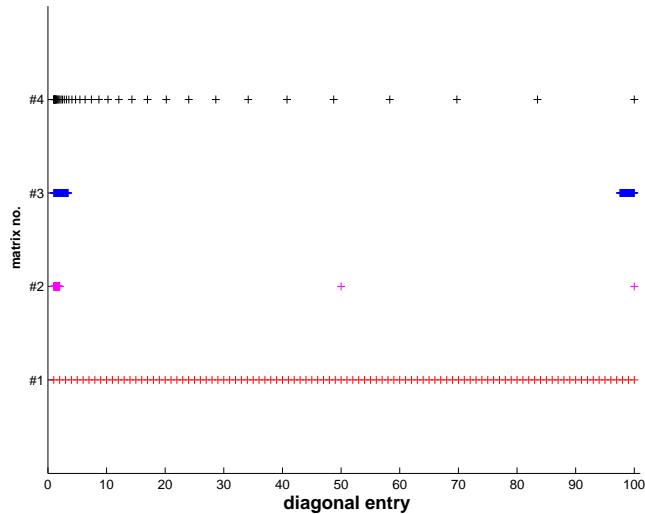
Test matrix #1:  $A = \text{diag}(d); d = (1:100);$

Test matrix #2:  $A = \text{diag}(d); d = [1 + (0:97)/97, 50, 100];$

Test matrix #3:  $A = \text{diag}(d); d = [1 + (0:49)*0.05, 100 - (0:49)*0.05];$

Test matrix #4: eigenvalues exponentially dense at 1

```
x0 = cos((1:n)'); b = zeros(n,1);
```



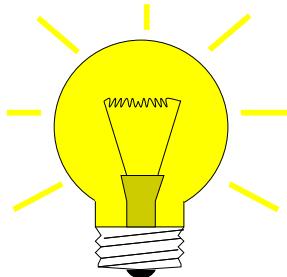
Observations: Distribution of eigenvalues has crucial impact on convergence of CG  
 (This is clear from the convergence theory, because detailed information about the spectrum allows a much better choice of “candidate polynomial” in (8.2.24) than merely using Chebychev polynomials)

➤ Clustering of eigenvalues leads to faster convergence of CG  
 (in stark contrast to the behavior of the gradient method, see Ex. 8.1.17)

CG convergence boosted by clustering of eigenvalues

## 8.3 Preconditioning [15, Sect. 13.5], [42, Ch. 10], [63, Sect. 4.3.5]

Thm. 8.2.25 ➤ (Potentially) slow convergence of CG in case  $\kappa(\mathbf{A}) \gg 1$ .



Idea:

Preconditioning

Apply CG method to transformed linear system

$$\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}, \quad \tilde{\mathbf{A}} := \mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2}, \quad \tilde{\mathbf{x}} := \mathbf{B}^{1/2}\mathbf{x}, \quad \tilde{\mathbf{b}} := \mathbf{B}^{-1/2}\mathbf{b}, \quad (8.3.1)$$

with “small”  $\kappa(\tilde{\mathbf{A}})$ ,  $\mathbf{B} = \mathbf{B}^\top \in \mathbb{R}^{N,N}$  s.p.d.  $\hat{=}$  preconditioner.

### Remark 8.3.2 (Square root of a s.p.d. matrix)

What is meant by the “square root”  $\mathbf{B}^{1/2}$  of a s.p.d. matrix  $\mathbf{B}$ ?

Recall (8.1.15) : for every  $\mathbf{B} \in \mathbb{R}^{n,n}$  with  $\mathbf{B}^\top = \mathbf{B}$  there is an orthogonal matrix  $\mathbf{Q} \in \mathbb{R}^{n,n}$  such that  $\mathbf{B} = \mathbf{Q}^\top \mathbf{D} \mathbf{Q}$  with a diagonal matrix  $\mathbf{D}$  ( $\rightarrow$  Cor. 7.1.9, [59, Thm. 7.8], [50, Satz 9.15]). If  $\mathbf{B}$  is s.p.d. the (diagonal) entries of  $\mathbf{D}$  are strictly positive and we can define

$$\mathbf{D} = \text{diag}(\lambda_1, \dots, \lambda_n), \quad \lambda_i > 0 \quad \Rightarrow \quad \mathbf{D}^{1/2} := \text{diag}(\sqrt{\lambda_1}, \dots, \sqrt{\lambda_n}).$$

This is generalized to

$$\mathbf{B}^{1/2} := \mathbf{Q}^\top \mathbf{D}^{1/2} \mathbf{Q},$$

and one easily verifies, using  $\mathbf{Q}^\top = \mathbf{Q}^{-1}$ , that  $(\mathbf{B}^{1/2})^2 = \mathbf{B}$  and that  $\mathbf{B}^{1/2}$  is s.p.d. In fact, these two requirements already determine  $\mathbf{B}^{1/2}$  uniquely:

$\mathbf{B}^{1/2}$  is the unique s.p.d. matrix such that  $(\mathbf{B}^{1/2})^2 = \mathbf{B}$ .

### Notion 8.3.3. Preconditioner

A s.p.d. matrix  $\mathbf{B} \in \mathbb{R}^{n,n}$  is called a **preconditioner** (ger.: Vorkonditionierer) for the s.p.d. matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$ , if

1.  $\kappa(\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2})$  is “small” and
2. the evaluation of  $\mathbf{B}^{-1}\mathbf{x}$  is about as expensive (in terms of elementary operations) as the matrix  $\times$  vector multiplication  $\mathbf{Ax}$ ,  $\mathbf{x} \in \mathbb{R}^n$ .

Recall: spectral condition number  $\kappa(\mathbf{A}) := \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})}$ , see (8.1.21)

There are several equivalent ways to express that  $\kappa(\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2})$  is “small”:

- $\kappa(\mathbf{B}^{-1}\mathbf{A})$  is “small”,  
because spectra agree  $\sigma(\mathbf{B}^{-1}\mathbf{A}) = \sigma(\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2})$  due to similarity ( $\rightarrow$  Lemma 7.1.6)
- $\exists 0 < \gamma < \Gamma$ ,  $\Gamma/\gamma$  “small”:  $\gamma(\mathbf{x}^\top \mathbf{B} \mathbf{x}) \leq \mathbf{x}^\top \mathbf{A} \mathbf{x} \leq \Gamma(\mathbf{x}^\top \mathbf{B} \mathbf{x}) \quad \forall \mathbf{x} \in \mathbb{R}^n$ ,  
where equivalence is seen by transforming  $\mathbf{y} := \mathbf{B}^{-1/2}\mathbf{x}$  and appealing to the min-max Thm. 7.3.41.

“Reader’s digest” version of Notion 8.3.3:

S.p.d.  $\mathbf{B}$  preconditioner : $\Leftrightarrow \mathbf{B}^{-1}$  = cheap approximate inverse of  $\mathbf{A}$

Problem:  $\mathbf{B}^{1/2}$ , which occurs prominently in (8.3.1) is usually not available with acceptable computational costs.

However, if one formally applies § 8.2.17 to the transformed system

$$\tilde{\mathbf{A}}\tilde{\mathbf{x}} := (\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2})(\mathbf{B}^{1/2}\mathbf{x}) = \tilde{\mathbf{b}} := \mathbf{B}^{-1/2}\mathbf{b},$$

from (8.3.1), it becomes apparent that, after suitable transformation of the iteration variables  $\mathbf{p}_j$  and  $\mathbf{r}_j$ ,  $\mathbf{B}^{1/2}$  and  $\mathbf{B}^{-1/2}$  invariably occur in products  $\mathbf{B}^{-1/2}\mathbf{B}^{-1/2} = \mathbf{B}^{-1}$  and  $\mathbf{B}^{1/2}\mathbf{B}^{-1/2} = \mathbf{I}$ . Thus, thanks to this **intrinsic transformation** square roots of  $\mathbf{B}$  are not required for the implementation!

### CG for $\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$

Input : initial guess  $\tilde{\mathbf{x}}^{(0)} \in \mathbb{R}^n$   
 Output : approximate solution  $\tilde{\mathbf{x}}^{(l)} \in \mathbb{R}^n$

```

 $\tilde{\mathbf{p}}_1 := \tilde{\mathbf{r}}_0 := \tilde{\mathbf{b}} - \mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2}\tilde{\mathbf{x}}^{(0)};$ 
for  $j = 1$  to  $l$  do {
     $\alpha := \frac{\tilde{\mathbf{p}}_j^T \tilde{\mathbf{r}}_{j-1}}{\tilde{\mathbf{p}}_j^T \mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2}\tilde{\mathbf{p}}_j}$ 
     $\tilde{\mathbf{x}}^{(j)} := \tilde{\mathbf{x}}^{(j-1)} + \alpha \tilde{\mathbf{p}}_j;$ 
     $\tilde{\mathbf{r}}_j = \tilde{\mathbf{r}}_{j-1} - \alpha \mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{1/2}\tilde{\mathbf{p}}_j;$ 
     $\tilde{\mathbf{p}}_{j+1} = \tilde{\mathbf{r}}_j - \frac{(\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2}\tilde{\mathbf{p}}_j)^T \tilde{\mathbf{r}}_j}{\tilde{\mathbf{p}}_j^T \mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2}\tilde{\mathbf{p}}_j} \tilde{\mathbf{p}}_j;$ 
}
    
```

### Equivalent CG with transformed variables

Input : initial guess  $\mathbf{x}^{(0)} \in \mathbb{R}^n$   
 Output : approximate solution  $\mathbf{x}^{(l)} \in \mathbb{R}^n$

```

 $\mathbf{B}^{1/2}\tilde{\mathbf{r}}_0 := \mathbf{B}^{1/2}\mathbf{b} - \mathbf{A}\mathbf{B}^{-1/2}\tilde{\mathbf{x}}^{(0)};$ 
 $\mathbf{B}^{-1/2}\tilde{\mathbf{p}}_1 := \mathbf{B}^{-1}(\mathbf{B}^{1/2}\tilde{\mathbf{r}}_0);$ 
for  $j = 1$  to  $l$  do {
     $\alpha := \frac{(\mathbf{B}^{-1/2}\tilde{\mathbf{p}}_j)^T \mathbf{B}^{1/2}\tilde{\mathbf{r}}_{j-1}}{(\mathbf{B}^{-1/2}\tilde{\mathbf{p}}_j)^T \mathbf{A}\mathbf{B}^{-1/2}\tilde{\mathbf{p}}_j}$ 
     $\mathbf{B}^{-1/2}\tilde{\mathbf{x}}^{(j)} := \mathbf{B}^{-1/2}\tilde{\mathbf{x}}^{(j-1)} + \alpha \mathbf{B}^{-1/2}\tilde{\mathbf{p}}_j;$ 
     $\mathbf{B}^{1/2}\tilde{\mathbf{r}}_j = \mathbf{B}^{1/2}\tilde{\mathbf{r}}_{j-1} - \alpha \mathbf{A}\mathbf{B}^{-1/2}\tilde{\mathbf{p}}_j;$ 
     $\mathbf{B}^{-1/2}\tilde{\mathbf{p}}_{j+1} = \mathbf{B}^{-1}(\mathbf{B}^{-1/2}\tilde{\mathbf{r}}_j) - \frac{(\mathbf{B}^{-1/2}\tilde{\mathbf{p}}_j)^T \mathbf{A}\mathbf{B}^{-1}(\mathbf{B}^{1/2}\tilde{\mathbf{r}}_j)}{(\mathbf{B}^{-1/2}\tilde{\mathbf{p}}_j)^T \mathbf{A}\mathbf{B}^{-1/2}\tilde{\mathbf{p}}_j} \mathbf{B}^{-1/2}\tilde{\mathbf{p}}_j;$ 
}
    
```

with the transformations:

$$\tilde{\mathbf{x}}^{(k)} = \mathbf{B}^{1/2}\mathbf{x}^{(k)}, \quad \tilde{\mathbf{r}}_k = \mathbf{B}^{-1/2}\mathbf{r}_k, \quad \tilde{\mathbf{p}}_k = \mathbf{B}^{-1/2}\mathbf{r}_k. \quad (8.3.4)$$

### (8.3.5) Preconditioned CG method (PCG) [15, Alg. 13.32], [42, Alg. 10.1]

Input: initial guess  $\mathbf{x} \in \mathbb{R}^n \doteq \mathbf{x}^{(0)} \in \mathbb{R}^n$ , tolerance  $\tau > 0$

Output: approximate solution  $\mathbf{x} \doteq \mathbf{x}^{(l)}$

```

 $\mathbf{p} := \mathbf{r} := \mathbf{b} - \mathbf{A}\mathbf{x}; \quad \mathbf{p} := \mathbf{B}^{-1}\mathbf{r}; \quad \mathbf{q} := \mathbf{p}; \quad \tau_0 := \mathbf{p}^\top \mathbf{r};$ 
for  $l = 1$  to  $l_{\max}$  do {
     $\beta := \mathbf{r}^\top \mathbf{q}; \quad \mathbf{h} := \mathbf{A}\mathbf{p}; \quad \alpha := \frac{\beta}{\mathbf{p}^\top \mathbf{h}};$ 
     $\mathbf{x} := \mathbf{x} + \alpha \mathbf{p};$ 
     $\mathbf{r} := \mathbf{r} - \alpha \mathbf{h};$ 
     $\mathbf{q} := \mathbf{B}^{-1}\mathbf{r}; \quad \beta := \frac{\mathbf{r}^\top \mathbf{q}}{\beta};$ 
    if  $|\mathbf{q}^\top \mathbf{r}| \leq \tau \cdot \tau_0$  then stop;
     $\mathbf{p} := \mathbf{q} + \beta \mathbf{p};$ 
}

```

(8.3.6)

### MATLAB-code 8.3.7: simple implementation of PCG algorithm § 8.3.5

```

1 function [x,rn,xk] = pcgbase(evalA,b,tol,maxit,invB,x)
2 % evalA must pass a handle to a function implementing A*x
3 % invB is to be a handle to a function providing the action of the
4 % preconditioner on a vector. The other arguments like for MATLAB's
5 % pcg.
6 r = b - evalA(x); rho = 1; rn = [];
7 if (nargout > 2), xk = x; end
8 for i = 1 : maxit
9     y = invB(r);
10    rho_old = rho; rho = r' * y; rn = [rn,rho];
11    if (i == 1), p = y; rho0 = rho;
12    elseif (rho < rho0*tol), return;
13    else beta = rho/rho_old; p = y+beta*p; end
14    q = evalA(p); alpha = rho / (p' * q);
15    x = x + alpha * p;
16    r = r - alpha * q;
17    if (nargout > 2), xk = [xk,x]; end
end

```

Computational effort per step: 1 evaluation  $\mathbf{A} \times \text{vector}$ ,  
                                   1 evaluation  $\mathbf{B}^{-1} \times \text{vector}$ ,  
                                   3 dot products, 3 AXPY-operations

### Remark 8.3.8 (Convergence theory for PCG)

Assertions of Thm. 8.2.25 remain valid with  $\kappa(\mathbf{A})$  replaced with  $\kappa(\mathbf{B}^{-1}\mathbf{A})$  and energy norm based on  $\tilde{\mathbf{A}}$  instead of  $\mathbf{A}$ .

### Example 8.3.9 (Simple preconditioners)

$\mathbf{B}$  = easily invertible “part” of  $\mathbf{A}$

- \*  $\mathbf{B} = \text{diag}(\mathbf{A})$ : Jacobi preconditioner (diagonal scaling)

$$\ast (\mathbf{B})_{ij} = \begin{cases} (\mathbf{A})_{ij} & , \text{if } |i - j| \leq k \\ 0 & \text{else,} \end{cases} \quad \text{for some } k \ll n.$$

#### Symmetric Gauss-Seidel preconditioner

Idea: Solve  $\mathbf{Ax} = \mathbf{b}$  approximately in two stages:

① Approximation  $\mathbf{A}^{-1} \approx \text{tril}(\mathbf{A})$  (lower triangular part):  $\tilde{\mathbf{x}} = \text{tril}(\mathbf{A})^{-1}\mathbf{b}$

② Approximation  $\mathbf{A}^{-1} \approx \text{triu}(\mathbf{A})$  (upper triangular part) and use this to approximately “solve” the error equation  $\mathbf{A}(\mathbf{x} - \tilde{\mathbf{x}}) = \mathbf{r}$ , with residual  $\mathbf{r} := \mathbf{b} - \mathbf{A}\tilde{\mathbf{x}}$ :

$$\mathbf{x} = \tilde{\mathbf{x}} + \text{triu}(\mathbf{A})^{-1}(\mathbf{b} - \mathbf{A}\tilde{\mathbf{x}}).$$

With  $\mathbf{L}_A := \text{tril}(\mathbf{A})$ ,  $\mathbf{U}_A := \text{triu}(\mathbf{A})$  one finds

$$\mathbf{x} = (\mathbf{L}_A^{-1} + \mathbf{U}_A^{-1} - \mathbf{U}_A^{-1}\mathbf{A}\mathbf{L}_A^{-1})\mathbf{b} \quad \blacktriangleright \quad \mathbf{B}^{-1} = \mathbf{L}_A^{-1} + \mathbf{U}_A^{-1} - \mathbf{U}_A^{-1}\mathbf{A}\mathbf{L}_A^{-1}. \quad (8.3.10)$$

For all these approaches the evaluation of  $\mathbf{B}^{-1}\mathbf{r}$  can be done with effort of  $\mathcal{O}(n)$  in the case of a sparse matrix  $\mathbf{A}$  (e.g. with  $\mathcal{O}(1)$  non-zero entries per row). However, there is absolutely no guarantee that  $\kappa(\mathbf{B}^{-1}\mathbf{A})$  will be reasonably small. It will crucially depend on  $\mathbf{A}$ , if this can be expected.

More complicated preconditioning strategies:

- \* Incomplete Cholesky factorization, MATLAB-`ichol`, [15, Sect. 13.5]]
- \* Sparse approximate inverse preconditioner (SPAI)

### Example 8.3.11 (Tridiagonal preconditioning)

Efficacy of preconditioning of sparse LSE with tridiagonal part:

#### MATLAB-code 8.3.12: LSE for Ex. 8.3.11

```

1 A =
2   spdiags(repmat([1/n,-1,2+2/n,-1,1/n],n,1),[-n/2,-1,0,1,n/2],n,n);
3 b = ones(n,1); x0 = ones(n,1); tol = 1.0E-4; maxit = 1000;
4 evalA = @(x) A*x;
5 % no preconditioning, see Code 8.3.7
6 invB = @(x) x; [x,rn] = pcgbase(evalA,b,tol,maxit,invB,x0);
7
8 % tridiagonal preconditioning, see Code 8.3.7

```

```

9 B = spdiags (spdiags (A, [-1, 0, 1]), [-1, 0, 1], n, n);
10 invB = @(x) B\x; [x, rnp] = pcgbase(evalA, b, tol, maxit, invB, x0);
    %

```

The Code 8.3.12 highlights the use of a preconditioner in the context of the PCG method; it only takes a function that realizes the application of  $\mathbf{B}^{-1}$  to a vector. In Line 10 of the code this function is passed as function handle `invB`.

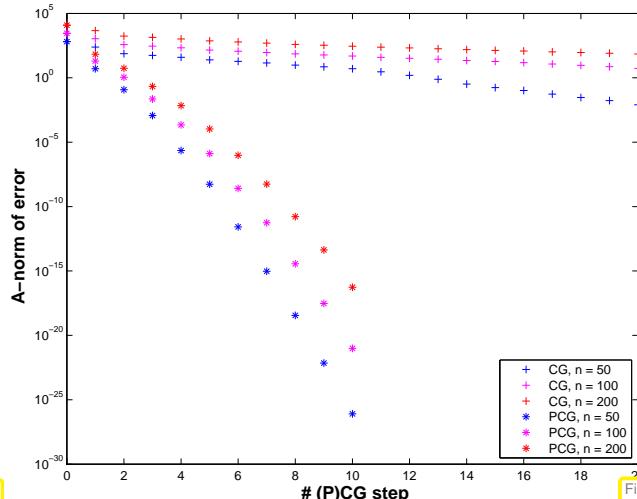


Fig. 283

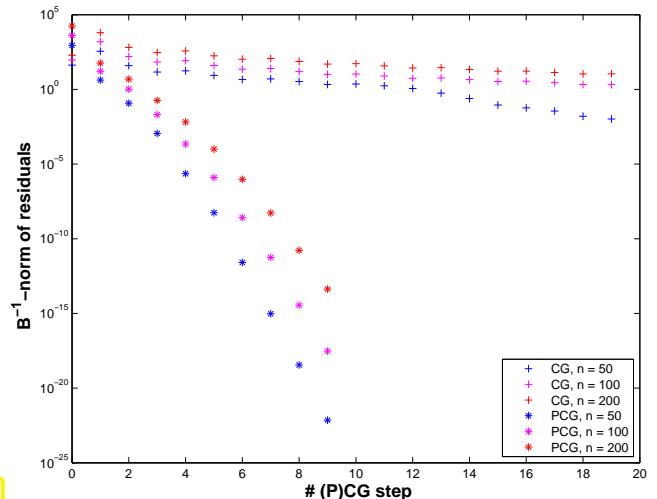
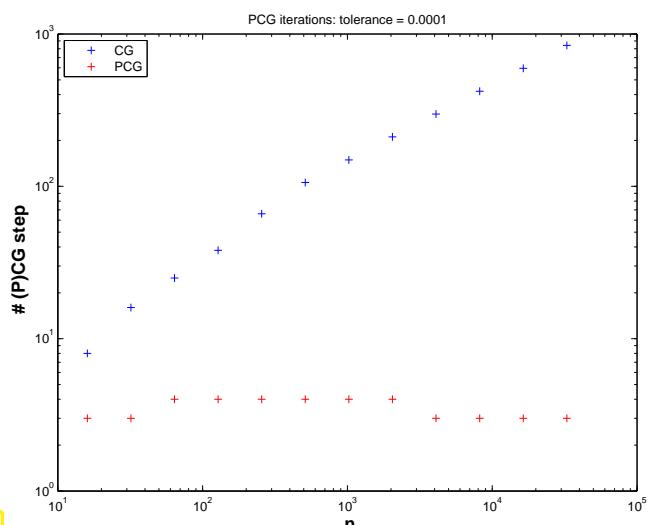


Fig. 284

$n$	# CG steps	# PCG steps
16	8	3
32	16	3
64	25	4
128	38	4
256	66	4
512	106	4
1024	149	4
2048	211	4
4096	298	3
8192	421	3
16384	595	3
32768	841	3

Fig. 285



Clearly in this example the tridiagonal part of the matrix is dominant for large  $n$ . In addition, its condition number grows  $\sim n^2$  as is revealed by a closer inspection of the spectrum.

Preconditioning with the tridiagonal part manages to suppress this growth of the condition number of  $\mathbf{B}^{-1}\mathbf{A}$  and ensures fast convergence of the preconditioned CG method

### Remark 8.3.13 (Termination of PCG)

Recall Rem. 8.2.19, (8.2.20):

$$\frac{1}{\text{cond}(\mathbf{A})} \frac{\|\mathbf{r}_l\|}{\|\mathbf{r}_0\|} \leq \frac{\|\mathbf{x}^{(l)} - \mathbf{x}^*\|}{\|\mathbf{x}^{(0)} - \mathbf{x}^*\|} \leq \text{cond}(\mathbf{A}) \frac{\|\mathbf{r}_l\|}{\|\mathbf{r}_0\|}. \quad (8.2.20)$$

$\mathbf{B}$  good preconditioner  $\Rightarrow \text{cond}_2(\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2})$  small ( $\rightarrow$  Notion 8.3.3)

Idea: consider (8.2.20) for

- \* Euclidean norm  $\|\cdot\| = \|\cdot\|_2 \leftrightarrow \text{cond}_2$
- \* transformed quantities  $\tilde{\mathbf{x}}, \tilde{\mathbf{r}}$ , see (8.3.1), (8.3.4)

Monitor 2-norm of transformed residual:

$$\tilde{\mathbf{r}} = \tilde{\mathbf{b}} - \tilde{\mathbf{A}}\tilde{\mathbf{x}} = \mathbf{B}^{-1/2}\mathbf{r} \Rightarrow \|\tilde{\mathbf{r}}\|_2^2 = \mathbf{r}^\top \mathbf{B}^{-1}\mathbf{r}.$$

(8.2.20)

$\blacktriangleright$  estimate for 2-norm of transformed iteration errors:  $\|\tilde{\mathbf{e}}^{(l)}\|_2^2 = (\mathbf{e}^{(l)})^\top \mathbf{B}\mathbf{e}^{(l)}$

Analogous to (8.2.20), estimates for energy norm ( $\rightarrow$  Def. 8.1.1) of error  $\mathbf{e}^{(l)} := \mathbf{x} - \mathbf{x}^{(l)}$ ,  $\mathbf{x}^* := \mathbf{A}^{-1}\mathbf{b}$ :

Use error equation  $\mathbf{A}\mathbf{e}^{(l)} = \mathbf{r}_l$ :

$$\begin{aligned} \mathbf{r}_l^\top \mathbf{B}^{-1}\mathbf{r}_l &= (\mathbf{B}^{-1}\mathbf{A}\mathbf{e}^{(l)})^\top \mathbf{A}\mathbf{e}^{(l)} \leq \lambda_{\max}(\mathbf{B}^{-1}\mathbf{A}) \|\mathbf{e}^{(l)}\|_A^2, \\ \|\mathbf{e}^{(l)}\|_A^2 &= (\mathbf{A}\mathbf{e}^{(l)})^\top \mathbf{e}^{(l)} = \mathbf{r}_l^\top \mathbf{A}^{-1}\mathbf{r}_l = \mathbf{B}^{-1}\mathbf{r}_l^\top \mathbf{B}\mathbf{A}^{-1}\mathbf{r}_l \leq \lambda_{\max}(\mathbf{B}\mathbf{A}^{-1}) (\mathbf{B}^{-1}\mathbf{r}_l)^\top \mathbf{r}_l. \end{aligned}$$



available during PCG iteration (8.3.6)

$$\frac{1}{\kappa(\mathbf{B}^{-1}\mathbf{A})} \frac{\|\mathbf{e}^{(l)}\|_A^2}{\|\mathbf{e}^{(0)}\|_A^2} \leq \frac{(\mathbf{B}^{-1}\mathbf{r}_l)^\top \mathbf{r}_l}{(\mathbf{B}^{-1}\mathbf{r}_0)^\top \mathbf{r}_0} \leq \kappa(\mathbf{B}^{-1}\mathbf{A}) \frac{\|\mathbf{e}^{(l)}\|_A^2}{\|\mathbf{e}^{(0)}\|_A^2} \quad (8.3.14)$$

$\kappa(\mathbf{B}^{-1}\mathbf{A})$  “small”  $\Rightarrow$   $\mathbf{B}^{-1}$ -energy norm of residual  $\approx \mathbf{A}$ -norm of error !  
 $(\mathbf{r}_l \cdot \mathbf{B}^{-1}\mathbf{r}_l = \mathbf{q}^\top \mathbf{r}$  in Algorithm (8.3.6))

MATLAB-function: `[x, flag, relr, it, rv] = pcg(A, b, tol, maxit, B, [], x0);`  
 $(\mathbf{A}, \mathbf{B}$  may be handles to functions providing  $\mathbf{A}\mathbf{x}$  and  $\mathbf{B}^{-1}\mathbf{x}$ , resp.)

**Remark 8.3.15 (Termination criterion in MATLAB-`pcg`  $\rightarrow$  [63, Sect. 4.6])**

Implementation (skeleton) of MATLAB built-in `pcg`:

Listing 8.1: MATLAB-code : PCG algorithm

```

1 function x = pcg(Afun,b,tol,maxit,Binvfun,x0)
2 x = x0; r = b - feval(Afun,x); rho = 1;
3 for i = 1 : maxit
4   y = feval(Binvfun,r);
5   rho1 = rho; rho = r' * y;
6   if (i == 1)
7     p = y;
8   else
9     beta = rho / rho1;
10    p = y + beta * p;
11   end
12   q = feval(Afun,p);
13   alpha = rho / (p' * q);
14   x = x + alpha * p;
15   if (norm(b - evalf(Afun,x)) <= tol*b*norm(b)) , return; end
16   r = r - alpha * q;
17 end

```

Dubious termination criterion !

## 8.4 Survey of Krylov Subspace Methods

### 8.4.1 Minimal residual methods

Idea: Replace Euclidean inner product in CG with  $\mathbf{A}$ -inner product

$$\left\| \mathbf{x}^{(l)} - \mathbf{x} \right\|_{\mathbf{A}} \text{ replaced with } \left\| \mathbf{A}(\mathbf{x}^{(l)} - \mathbf{x}) \right\|_2 = \|\mathbf{r}_l\|_2$$

► MINRES method [35, Sect. 9.5.2] (for any symmetric matrix !)

#### Theorem 8.4.1.

For  $\mathbf{A} = \mathbf{A}^H \in \mathbb{R}^{n,n}$  the residuals  $\mathbf{r}_l$  generated in the MINRES iteration satisfy

$$\begin{aligned} \|\mathbf{r}_l\|_2 &= \min \{ \|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2 : \mathbf{y} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A}, \mathbf{r}_0) \} \\ &\Rightarrow \|\mathbf{r}_l\|_2 \leq \frac{2 \left( 1 - \frac{1}{\kappa(\mathbf{A})} \right)^l}{\left( 1 + \frac{1}{\kappa(\mathbf{A})} \right)^{2l} + \left( 1 - \frac{1}{\kappa(\mathbf{A})} \right)^{2l}} \|\mathbf{r}_0\|_2. \end{aligned}$$

Note: similar formula for (linear) rate of convergence as for CG, see Thm. 8.2.25, but with  $\sqrt{\kappa(\mathbf{A})}$  replaced with  $\kappa(\mathbf{A})$  !

Iterative solver for  $\mathbf{Ax} = \mathbf{b}$  with symmetric system matrix  $\mathbf{A}$ :

- MATLAB-functions:
- `[x, flg, res, it, resv] = minres (A, b, tol, maxit, B, [], x0);`
  - `[...] = minres (Afun, b, tol, maxit, Binvfun, [], x0);`

Computational costs : 1  $\mathbf{A} \times$ vector, 1  $\mathbf{B}^{-1} \times$ vector per step, a few dot products & SAXPYs

Memory requirement: a few vectors  $\in \mathbb{R}^n$

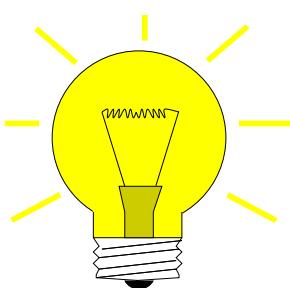
Extension to general regular  $\mathbf{A} \in \mathbb{R}^{n,n}$ :

Idea: Solver overdetermined linear system of equations

$$\mathbf{x}^{(l)} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A}, \mathbf{r}_0): \quad \mathbf{Ax}^{(l)} = \mathbf{b}$$

in *least squares sense*, → Chapter 6.

$$\mathbf{x}^{(l)} = \operatorname{argmin}\{\|\mathbf{Ay} - \mathbf{b}\|_2: \mathbf{y} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A}, \mathbf{r}_0)\}.$$



- **GMRES** method for general matrices  $\mathbf{A} \in \mathbb{R}^{n,n}$  → [42, Ch. 16], [63, Sect. 4.4.2]

- MATLAB-function:
- `[x, flag, relr, it, rv] = gmres (A, b, rs, tol, maxit, B, [], x0);`
  - `[...] = gmres (Afun, b, rs, tol, maxit, Binvfun, [], x0);`

Computational costs : 1  $\mathbf{A} \times$ vector, 1  $\mathbf{B}^{-1} \times$ vector per step,  
:  $O(l)$  dot products & SAXPYs in  $l$ -th step

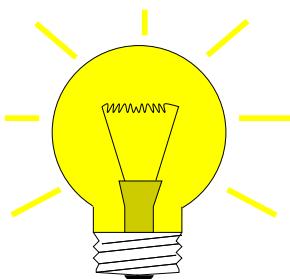
Memory requirements:  $O(l)$  vectors  $\in \mathbb{K}^n$  in  $l$ -th step

### Remark 8.4.2 (Restarted GMRES)

After many steps of GMRES we face considerable computational costs and memory requirements for every further step. Thus, the iteration may be *restarted* with the current iterate  $\mathbf{x}^{(l)}$  as initial guess →  $rs$ -parameter triggers restart after every  $rs$  steps (Danger: failure to converge).

## 8.4.2 Iterations with short recursions [63, Sect. 4.5]

Iterative methods for *general* regular system matrix  $\mathbf{A}$ :



Idea: Given  $\mathbf{x}^{(0)} \in \mathbb{R}^n$  determine (better) approximation  $\mathbf{x}^{(l)}$  through **Petrov-Galerkin condition**

$$\mathbf{x}^{(l)} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A}, \mathbf{r}_0): \quad \mathbf{p}^H(\mathbf{b} - \mathbf{Ax}^{(l)}) = 0 \quad \forall \mathbf{p} \in W_l,$$

with suitable **test space**  $W_l$ ,  $\dim W_l = l$ , e.g.  $W_l := \mathcal{K}_l(\mathbf{A}^H, \mathbf{r}_0)$  (→ bi-conjugate gradients, BiCG)

- Zoo of methods with short recursions (i.e. constant effort per step)

- MATLAB-function:
- `[x, flag, r, it, rv] = bicgstab (A, b, tol, maxit, B, [], x0);`
  - `[...] = bicgstab (Afun, b, tol, maxit, Binvfun, [], x0);`

Computational costs : 2  $\mathbf{A} \times$ vector, 2  $\mathbf{B}^{-1} \times$ vector, 4 dot products, 6 SAXPYs per step

Memory requirements: 8 vectors  $\in \mathbb{R}^n$

MATLAB-function:

- `[x, flag, r, it, rv] = qmr(A, b, tol, maxit, B, [], x0)`
- `[...] = qmr(Afun, b, tol, maxit, Binvfun, [], x0);`

Computational costs : 2  $\mathbf{A} \times \text{vector}$ , 2  $\mathbf{B}^{-1} \times \text{vector}$ , 2 dot products, 12 SAXPYs per step

Memory requirements: 10 vectors  $\in \mathbb{R}^n$



- \* little (useful) convergence theory available
- \* stagnation & “breakdowns” commonly occur

### Example 8.4.3 (Failure of Krylov iterative solvers)

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & \cdots & & \cdots & 0 \\ 0 & 0 & 1 & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & & \\ & & & & \ddots & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & 0 \\ 0 & & & & 0 & 1 & \\ 1 & 0 & \cdots & & \cdots & 0 & \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ 1 \end{bmatrix} \quad \Rightarrow \quad \mathbf{x} = \mathbf{e}_1.$$

$$\mathbf{x}^{(0)} = 0 \quad \Rightarrow \quad \mathbf{r}_0 = \mathbf{e}_n \quad \Rightarrow \quad \mathcal{K}_l(\mathbf{A}, \mathbf{r}_0) = \text{Span}\{\mathbf{e}_n, \mathbf{e}_{n-1}, \dots, \mathbf{e}_{n-l+1}\}$$

$$\Rightarrow \min\{\|\mathbf{y} - \mathbf{x}\|_2 : \mathbf{y} \in \mathcal{K}_l(\mathbf{A}, \mathbf{r}_0)\} = \begin{cases} 1 & , \text{if } l \leq n, \\ 0 & , \text{for } l = n. \end{cases}$$

TRY & PRAY

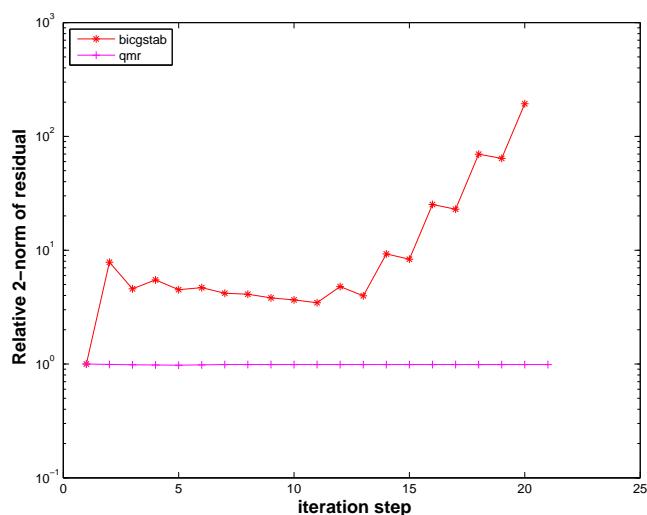
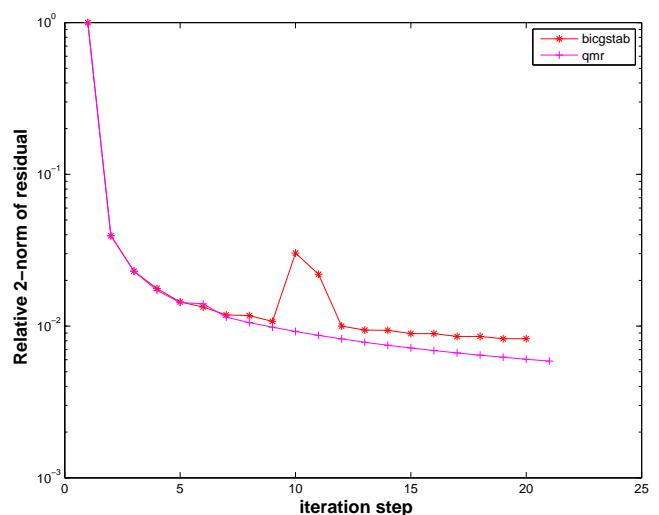
### Example 8.4.4 (Convergence of Krylov subspace methods for non-symmetric system matrix)

```

1 A = gallery('tridiag', -0.5*ones(n-1,1), 2*ones(n,1), -1.5*ones(n-1,1));
2 B = gallery('tridiag', 0.5*ones(n-1,1), 2*ones(n,1), 1.5*ones(n-1,1));

```

Plotted:  $\|\mathbf{r}_l\|_2 : \|\mathbf{r}_0\|_2$ :

tridiagonal matrix **A**tridiagonal matrix **B**

## Summary:

Advantages of Krylov methods vs. direct elimination (**IF** they converge at all/sufficiently fast).

- They require system matrix **A** in procedural form  $y = \text{evalA}(x) \leftrightarrow y = Ax$  only.
- They can perfectly exploit sparsity of system matrix.
- They can cash in on low accuracy requirements (**IF** viable termination criterion available).
- They can benefit from a good initial guess.

# Chapter 9

## Filtering Algorithms

This chapter continues the theme of *numerical linear algebra*, earlier covered in Chapter 1, 1.6, 8. We will come across very special linear transformations ( $\leftrightarrow$  matrices) and related algorithms. Surprisingly, these form the basis of a host of very important numerical methods for **signal processing**.

Perspective of **signal processing**:

$$\text{vector } \mathbf{x} \in \mathbb{R}^n \leftrightarrow \text{finite discrete (= sampled) signal.}$$

**Sampling:**

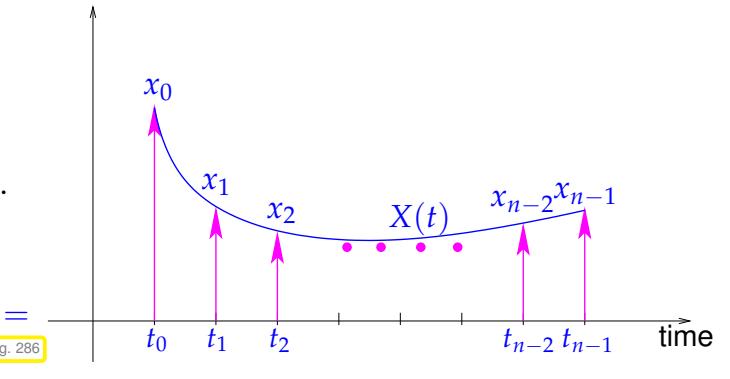
$X = X(t)$   $\hat{=}$  time-continuous signal,  $0 \leq t \leq T$ ,

“sampling”:  $x_j = X(j\Delta t)$ ,  $j = 0, \dots, n - 1$ ,  
 $n \in \mathbb{N}$ ,  $n\Delta t \leq T$ .

$\Delta t > 0$   $\hat{=}$  time between samples.

Sampled values arranged in a vector  $\mathbf{x} = (x_0, \dots, x_{n-1})^\top \in \mathbb{R}^n$ .

Note: vector indices  $0, \dots, n - 1$ ! (“C-style indexing”).



Generalization: (bi-infinite) time-discrete signals of infinite duration

$$\uparrow\downarrow \text{sequence } (x_k)_{k \in \mathbb{Z}}$$

## Contents

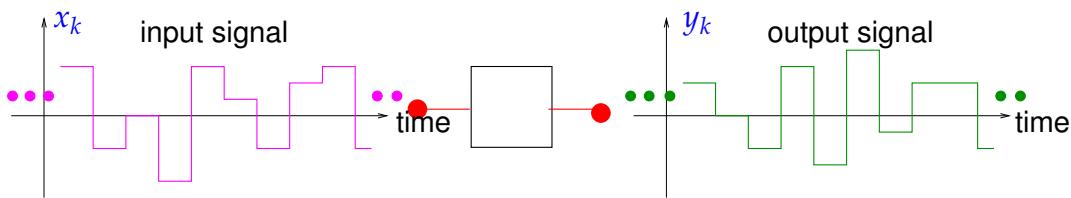
9.1	Discrete convolutions	532
9.2	Discrete Fourier Transform (DFT)	539
9.2.1	Discrete convolution via DFT	544
9.2.2	Frequency filtering via DFT	544
9.2.3	Real DFT	551
9.2.4	Two-dimensional DFT	552
9.2.5	Semi-discrete Fourier Transform [63, Sect. 10.11]	555
9.3	Fast Fourier Transform (FFT)	563
9.4	Trigonometric transformations	568
9.4.1	Sine transform	569
9.4.2	Cosine transform	573
9.5	Toepplitz matrix techniques	574
9.5.1	Toepplitz matrix arithmetic	576

## 9.1 Discrete convolutions

### Example 9.1.1 (Discrete finite linear time-invariant causal channel (filter))

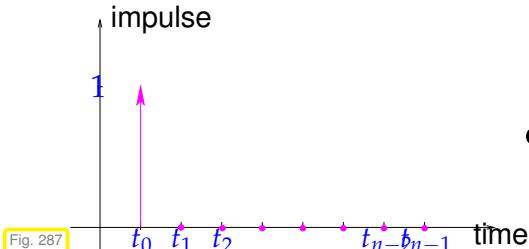
We study a **finite linear time-invariant causal channel (filter)**:  
(widely used model for digital communication channels, e.g. in wireless communication theory)

Mathematically speaking, a (discrete) channel/filter is a mapping from the space of input sequences  $\{x_i\}_{i \in \mathbb{Z}}$  to output sequences  $\{y_i\}_{i \in \mathbb{Z}}$ .

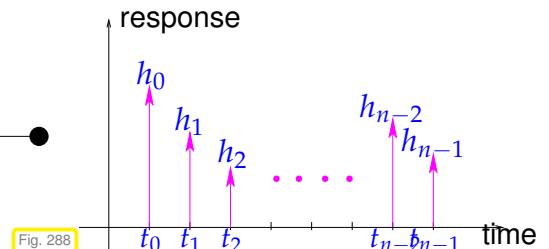


**Impulse response** = output when filter is fed with a single impulse of strength one, corresponding to input  $e_1$  (first unit vector).

Impulse response of channel (filter):



$$\mathbf{h} = (h_0, \dots, h_{n-1})^\top$$



In order to link digital filters to linear algebra, we have to assume certain properties that are indicated by the attributes “finite”, “linear”, “time-invariant” and “causal”:

**finite**: impulse response of finite duration  $\Rightarrow$  it can be described by a vector  $\mathbf{h}$  of finite length  $n$ .

**time-invariant**: when input is shifted in time, output is shifted by the same amount of time.

**linear**: input  $\mapsto$  output-map is linear

$$\text{output}(\mu \cdot \text{signal 1} + \lambda \cdot \text{signal 2}) = \mu \cdot \text{output}(\text{signal 1}) + \lambda \cdot \text{output}(\text{signal 2}) . \quad (9.1.2)$$

**causal** (or physical, or nonanticipative): output depends only on past and present inputs, not on the future.  
(Note: the impulse response depicted in Fig. 288 is finite and causal)

► The output  $(y_1, y_2, \dots)^\top$  for finite length input  $\mathbf{x} = (x_0, \dots, x_{n-1})^\top \in \mathbb{R}^n$  is  
a superposition of  $x_j$ -weighted  $j\Delta t$ -shifted impulse responses

Understand, why this is a consequence of linearity (9.1.2).

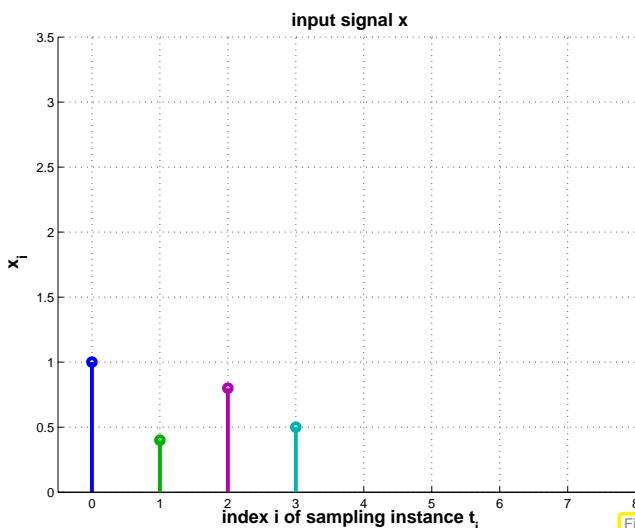


Fig. 299

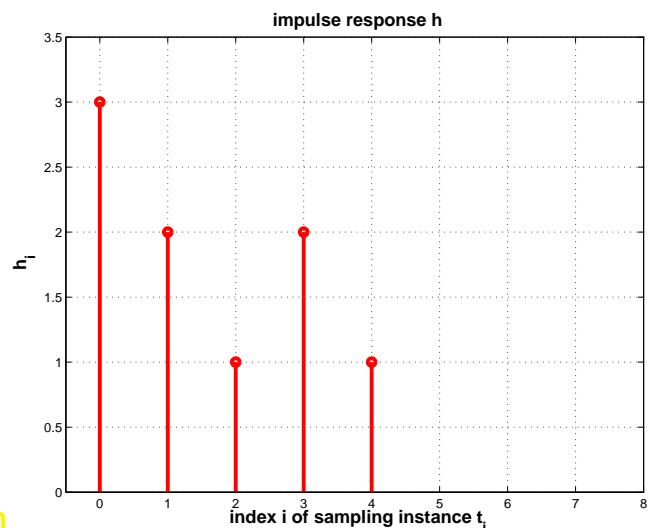
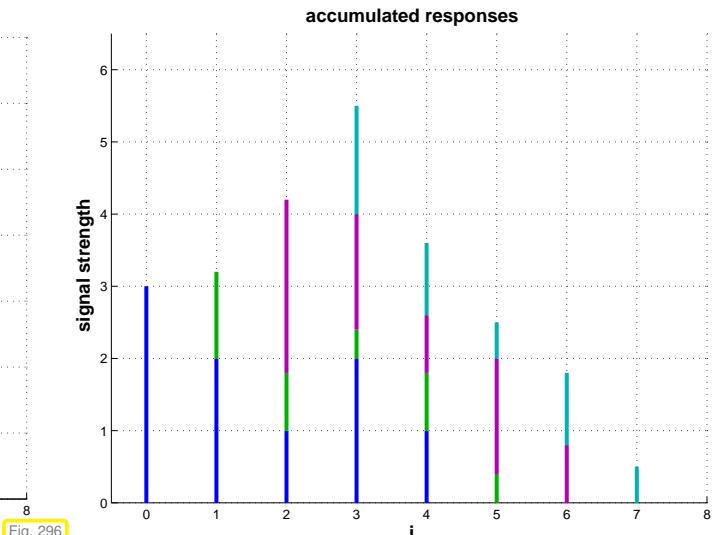
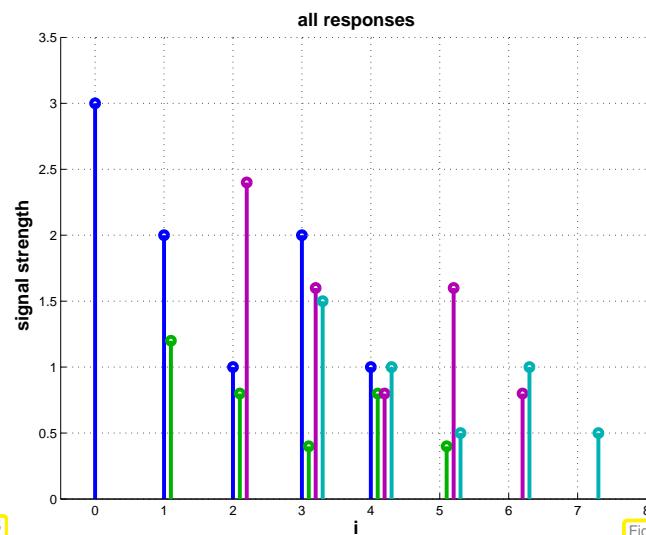
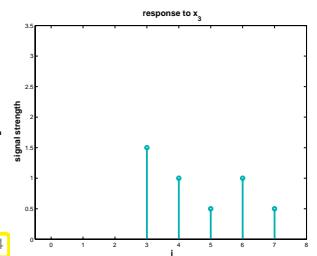
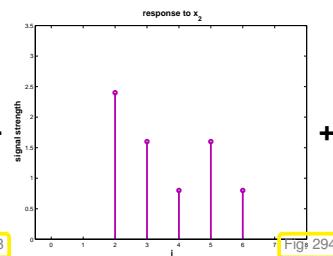
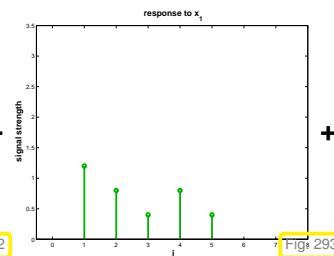
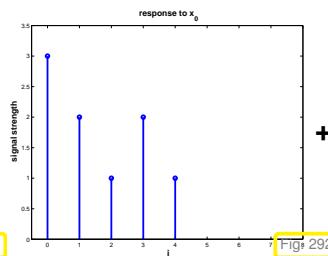


Fig. 299

Output = linear superposition of impulse responses:



“Duration” of output signal = “duration” of input signal + “duration” of impulse response

► General formula:

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \\ \vdots \\ y_{2n-3} \\ y_{2n-2} \end{bmatrix} = x_0 \begin{bmatrix} h_0 \\ \vdots \\ h_{n-1} \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + x_1 \begin{bmatrix} 0 \\ h_0 \\ \vdots \\ h_{n-1} \\ 0 \\ \vdots \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ h_0 \\ \vdots \\ h_{n-1} \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \cdots + x_{n-1} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ h_0 \\ \vdots \\ h_{n-1} \end{bmatrix}.$$

channel is causal!

$$y_k = \sum_{j=0}^{n-1} h_{k-j} x_j , \quad k = 0, \dots, 2n-2 \quad (h_j := 0 \text{ for } j < 0 \text{ and } j \geq n) . \quad (9.1.3)$$

$\mathbf{x} = (x_0, \dots, x_{n-1})^\top \in \mathbb{R}^n \hat{=} \text{input signal} \mapsto \mathbf{y} = (y_0, \dots, y_{2n-2})^\top \in \mathbb{R}^{2n-1} \hat{=} \text{output signal.}$

Matrix notation of (9.1.3):

$$\begin{bmatrix} y_0 \\ \vdots \\ y_{2n-2} \end{bmatrix} = \begin{bmatrix} h_0 & 0 & \cdots & 0 \\ h_1 & & & \\ \vdots & & & \\ h_{n-1} & 0 & \cdots & h_1 & h_0 \\ 0 & & & & \\ \vdots & & & & \\ 0 & & \cdots & 0 & h_{n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix} . \quad (9.1.4)$$

### Example 9.1.5 (Multiplication of polynomials)

“Surprisingly” the bilinear operation (9.1.3) that takes two input vectors and produces an output vector with double the number of entries ( $-1$ ) also governs the multiplication of polynomials:

$$p(z) = \sum_{k=0}^{n-1} a_k z^k , \quad q(z) = \sum_{k=0}^{n-1} b_k z^k \quad \Rightarrow \quad (pq)(z) = \sum_{k=0}^{2n-2} \underbrace{\left( \sum_{j=0}^k a_j b_{k-j} \right)}_{=:c_k} z^k \quad (9.1.6)$$

Here the roles of  $h_k$ ,  $x_k$  are played by the  $a_k$  and  $b_k$ .

➤ coefficients of product polynomial by **discrete convolution** of coefficients of polynomial factors!

Both in (9.1.3) and (9.1.6) we recognize the same pattern of a particular *bi-linear* combination of

- discrete signals in Ex. 9.1.1,
- polynomial coefficient sequences in Ex. 9.1.5.

### Definition 9.1.7. Discrete convolution

Given  $\mathbf{x} = (x_0, \dots, x_{n-1})^\top \in \mathbb{K}^n$ ,  $\mathbf{h} = (h_0, \dots, h_{n-1})^\top \in \mathbb{K}^n$  their **discrete convolution** (ger.: diskrete Faltung) is the vector  $\mathbf{y} \in \mathbb{K}^{2n-1}$  with components

$$y_k = \sum_{j=0}^{n-1} h_{k-j} x_j, \quad k = 0, \dots, 2n-2 \quad (h_j := 0 \text{ for } j < 0). \quad (9.1.8)$$

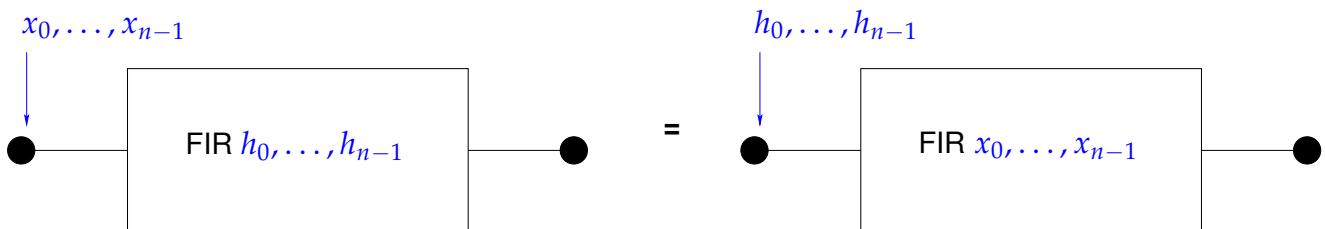
Notation for discrete convolution (9.1.8):

$$\mathbf{y} = \mathbf{h} * \mathbf{x}.$$

Defining  $x_j := 0$  for  $j < 0$ , we find that *discrete convolution is commutative*:

$$y_k = \sum_{j=0}^{n-1} h_{k-j} x_j = \sum_{l=0}^{n-1} h_l x_{k-l}, \quad k = 0, \dots, 2n-2, \quad (\text{that is, } \mathbf{h} * \mathbf{x} = \mathbf{x} * \mathbf{h}).$$

obtained by index transformation  $l \leftarrow k - j$ .



### Remark 9.1.9 (Convolution of sequences)

The notion of a discrete convolution of Def. 9.1.7 naturally extends to sequences  $\mathbb{N}_0 \mapsto \mathbb{K}$ : the (discrete) convolution of two sequences  $(x_j)_{j \in \mathbb{N}_0}$ ,  $(y_j)_{j \in \mathbb{N}_0}$  is the sequence  $(z_j)_{j \in \mathbb{N}_0}$  defined by

$$z_k := \sum_{j=0}^k x_{k-j} y_j = \sum_{j=0}^k x_j y_{k-j}, \quad k \in \mathbb{N}_0.$$

In this context recall: product formula for power series, **Cauchy product**

### Example 9.1.10 (Linear filtering of periodic signals)

**n-periodic** signal ( $n \in \mathbb{N}$ ) = sequence  $(x_j)_{j \in \mathbb{Z}}$  with  $x_{j+n} = x_j \quad \forall j \in \mathbb{Z}$

➤ **n-periodic** signal  $(x_j)_{j \in \mathbb{Z}}$  fixed by  $x_0, \dots, x_{n-1} \leftrightarrow$  vector  $\mathbf{x} = (x_0, \dots, x_{n-1})^\top \in \mathbb{R}^n$ .

Whenever the input signal of a time-invariant filter is **n-periodic**, so will be the output signal. Thus, in the **n-periodic** setting, a causal *linear* time-invariant filter will give rise to a *linear* mapping  $\mathbb{R}^n \mapsto \mathbb{R}^n$  according to

$$y_k = \sum_{j=0}^{n-1} p_{k-j} x_j \quad \text{for some } p_0, \dots, p_{n-1} \in \mathbb{R}, \quad p_k := p_{k-n} \text{ for all } k \in \mathbb{Z}. \quad (9.1.11)$$

Note:  $p_0, \dots, p_{n-1}$  does **not** agree with the impulse response of the filter.

Matrix notation:

$$\begin{bmatrix} y_0 \\ \vdots \\ y_{n-1} \end{bmatrix} = \underbrace{\begin{bmatrix} p_0 & p_{n-1} & p_{n-2} & \cdots & & \cdots & p_1 \\ p_1 & p_0 & p_{n-1} & & & & \vdots \\ p_2 & p_1 & p_0 & \ddots & & & \\ \vdots & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & & \\ \vdots & & & & \ddots & \ddots & \\ p_{n-1} & \cdots & & & & p_1 & p_0 \end{bmatrix}}_{=: \mathbf{P}} \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix}. \quad (9.1.12)$$

►  $(\mathbf{P})_{ij} = p_{i-j}$ ,  $1 \leq i, j \leq n$ , with  $p_j := p_{j+n}$  for  $1-n \leq j < 0$ .

### Definition 9.1.13. Discrete periodic convolution

The **discrete periodic convolution** of two  $n$ -periodic sequences  $(x_k)_{k \in \mathbb{Z}}$ ,  $(y_k)_{k \in \mathbb{Z}}$  yields the  $n$ -periodic sequence

$$(z_k) := (x_k) *_n (y_k), \quad z_k := \sum_{j=0}^{n-1} x_{k-j} y_j = \sum_{j=0}^{n-1} y_{k-j} x_j, \quad k \in \mathbb{Z}.$$

☞ notation for discrete periodic convolution:  $(x_k) *_n (y_k)$

Since  $n$ -periodic sequences can be identified with vectors in  $\mathbb{K}^n$  (see above), we can also introduce the discrete periodic convolution of vectors:

Def. 9.1.13 ➤ discrete periodic convolution of vectors:  $\mathbf{z} = \mathbf{x} *_n \mathbf{y} \in \mathbb{K}^n$ ,  $\mathbf{x}, \mathbf{y} \in \mathbb{K}^n$ .

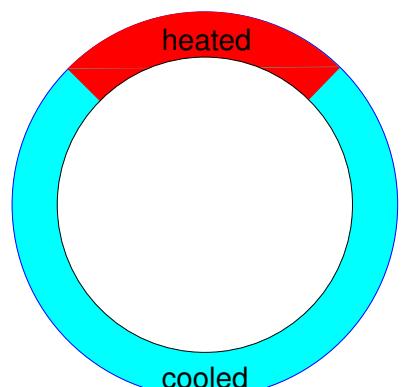
### Example 9.1.14 (Radiative heat transfer)

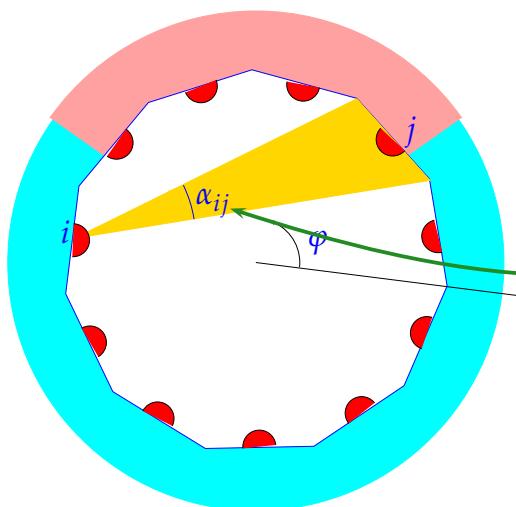
Beyond signal processing discrete periodic convolutions occur in mathematical models:

An engineering problem:

- \* cylindrical pipe,
- \* heated on part  $\Gamma_H$  of its perimeter ( $\rightarrow$  prescribed heat flux),
- \* cooled on remaining perimeter  $\Gamma_K$  ( $\rightarrow$  constant heat flux).

Task: compute local heat fluxes.





**Modeling (discretization):**

- approximation by regular  $n$ -polygon, edges  $\Gamma_j$ ,
- isotropic radiation of each edge  $\Gamma_j$  (power  $I_j$ ),

$$\text{radiative heat flow } \Gamma_j \rightarrow \Gamma_i: P_{ji} := \frac{\alpha_{ij}}{\pi} I_j,$$

$$\text{opening angle: } \alpha_{ij} = \pi \gamma_{|i-j|}, 1 \leq i, j \leq n,$$

$$\text{power balance: } \underbrace{\sum_{i=1, i \neq j}^n P_{ji}}_{=I_j} - \sum_{i=1, i \neq j}^n P_{ij} = Q_j. \quad (9.1.15)$$

$Q_j \triangleq$  heat flux through  $\Gamma_j$ , satisfies

$$Q_j := \int_{\frac{2\pi}{n}(j-1)}^{\frac{2\pi}{n}j} q(\varphi) d\varphi, \quad q(\varphi) := \begin{cases} \text{local heating} & , \text{ if } \varphi \in \Gamma_H, \\ -\frac{1}{|\Gamma_K|} \int_{\Gamma_H} q(\varphi) d\varphi & (\text{const.}), \text{ if } \varphi \in \Gamma_K. \end{cases}$$

$$(9.1.15) \Rightarrow \text{LSE: } I_j - \sum_{i=1, i \neq j}^n \frac{\alpha_{ij}}{\pi} I_i = Q_j, \quad j = 1, \dots, n.$$

$$n = 8: \quad \begin{bmatrix} 1 & -\gamma_1 & -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 \\ -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 \\ -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 \\ -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 & -\gamma_2 & -\gamma_4 \\ -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 & -\gamma_3 \\ -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 \\ -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 \\ -\gamma_1 & -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \\ I_5 \\ I_6 \\ I_7 \\ I_8 \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \\ Q_5 \\ Q_6 \\ Q_7 \\ Q_8 \end{bmatrix}. \quad (9.1.16)$$

This is a linear system of equations with symmetric, singular, and (by Lemma 7.1.5,  $\sum \gamma_i \leq 1$ ) positive semidefinite ( $\rightarrow$  Def. 1.1.8) system matrix.

Note: matrices from (9.1.12) and (9.1.16) have the same structure !

Observe: LSE from (9.1.16) can be written by means of the discrete periodic convolution ( $\rightarrow$  Def. 9.1.13) of vectors  $\mathbf{y} = (1, -\gamma_1, -\gamma_2, -\gamma_3, -\gamma_4, -\gamma_3, -\gamma_2, -\gamma_1)$ ,  $\mathbf{x} = (I_1, \dots, I_8)$

$$(9.1.16) \leftrightarrow \mathbf{y} *_8 \mathbf{x} = (Q_1, \dots, Q_8)^\top.$$

### Definition 9.1.17. Circulant matrix $\rightarrow$ [42, Sect. 54]

A matrix  $\mathbf{C} = (c_{ij})_{i,j=1}^n \in \mathbb{K}^{n,n}$  is **circulant** (ger.: zirkulant)  
 $\Leftrightarrow \exists (u_k)_{k \in \mathbb{Z}}$   $n$ -periodic sequence:  $c_{ij} = u_{j-i}$ ,  $1 \leq i, j \leq n$ .

- ☞ Circulant matrix has constant (main, sub- and super-) diagonals (for which indices  $j - i = \text{const.}$ ).
- ☞ columns/rows arise by *cyclic permutation* from first column/row.

Similar to the case of banded matrices ( $\rightarrow$  Section 1.7.6):

“information content” of circulant matrix  $\mathbf{C} \in \mathbb{K}^{n,n} = n$  numbers  $\in \mathbb{K}$ .  
 (obviously, one vector  $\mathbf{u} \in \mathbb{K}^n$  enough to define circulant matrix  $\mathbf{C} \in \mathbb{K}^{n,n}$ )

Structure of circulant matrix

$$\triangleright \quad \left[ \begin{array}{ccccccccc} u_0 & u_1 & u_2 & \cdots & & \cdots & & u_{n-1} \\ u_{n-1} & u_0 & & & & & & u_{n-2} \\ u_{n-2} & & u_0 & & & & & \vdots \\ \vdots & & & \ddots & & & & \vdots \\ \vdots & & & & \ddots & & & \vdots \\ u_2 & & u_2 & \cdots & & & & u_{n-1} \\ u_1 & u_2 & \cdots & & & & & u_0 \end{array} \right]$$

Write  $\mathbf{Z}((u_k)) \in \mathbb{K}^{n,n}$  for the circulant matrix generated by the  $n$ -periodic sequence  $(u_k)_{k \in \mathbb{Z}}$ . Denote by  $\mathbf{y} := (y_0, \dots, y_{n-1})^\top$ ,  $\mathbf{x} = (x_0, \dots, x_{n-1})^\top$  the vectors associated to  $n$ -periodic sequences.

Then the commutativity of the discrete periodic convolution ( $\rightarrow$  Def. 9.1.13) ensures

$$\mathbf{Z}((x_k))\mathbf{y} = \mathbf{Z}((y_k))\mathbf{x}. \quad (9.1.18)$$

### Remark 9.1.19 (Reduction to periodic convolution)

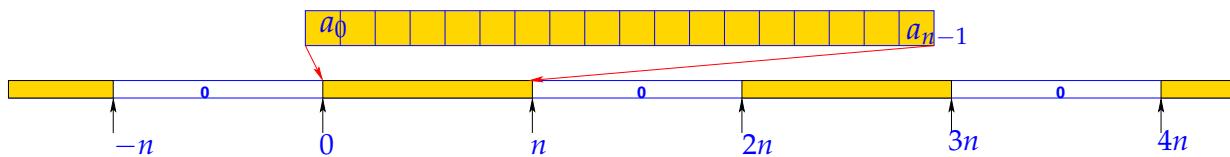
Recall discrete convolution ( $\rightarrow$  Def. 9.1.7) of  $\mathbf{a} = (a_0, \dots, a_{n-1})^\top \in \mathbb{K}^n$ ,  $\mathbf{b} = (b_0, \dots, b_{n-1})^\top \in \mathbb{K}^n$ :

$$(\mathbf{a} * \mathbf{b})_k = \sum_{j=0}^{n-1} a_j b_{k-j}, \quad k = 0, \dots, 2n-2.$$

Expand  $a_0, \dots, a_{n-1}$  and  $b_0, \dots, b_{n-1}$  to  $2n-1$ -periodic sequences by **zero padding**:

$$x_k := \begin{cases} a_k & , \text{ if } 0 \leq k < n, \\ 0 & , \text{ if } n \leq k < 2n-1 \end{cases}, \quad y_k := \begin{cases} b_k & , \text{ if } 0 \leq k < n, \\ 0 & , \text{ if } n \leq k < 2n-1 \end{cases}, \quad (9.1.20)$$

and periodic extension:  $x_k = x_{2n-1+k}$ ,  $y_k = y_{2n-1+k}$  for all  $k \in \mathbb{Z}$ .



►  $(\mathbf{a} * \mathbf{b})_k = (\mathbf{x} *_{2n-1} \mathbf{y})_k, \quad k = 0, \dots, 2n-2. \quad (9.1.21)$

Matrix view of reduction to periodic convolution, cf. (9.1.4)

$$\begin{bmatrix} y_0 \\ \vdots \\ y_{2n-2} \end{bmatrix} = \underbrace{\begin{bmatrix} h_0 & 0 & & & & & & \\ h_1 & & & & & & & \\ & \ddots & & & & & & \\ & & h_{n-1} & 0 & & & & \\ & & & 0 & & & & \\ & & & & h_1 & h_0 & 0 & \\ & & & & & h_1 & h_0 & \\ & & & & & & \ddots & \\ & & & & & & & 0 & h_{n-1} \\ & & & & & & & & 0 \\ & & & & & & & & & \ddots \\ & & & & & & & & & & x_0 \\ & & & & & & & & & & \vdots \\ & & & & & & & & & & x_{n-1} \\ & & & & & & & & & & 0 \end{bmatrix}}_{\text{a } (2n-1) \times (2n-1) \text{ circulant matrix!}} \cdot \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \quad (9.1.22)$$

## 9.2 Discrete Fourier Transform (DFT)

### Example 9.2.1 (Eigenvectors of circulant matrices)

Now we are about to discover a very deep truth . . .

Listing 9.1: Eigenvectors of random circulant matrices

```

1 function circeig
2 % Computing the eigenvectors of a random circulant matrix
3 n = 8;
4 % The MATLAB way to generate a circulant matrix
5 C = gallery ('circul',rand(n,1)); [V1,D1] = eig(C);
6
7 for j=1:n
8 figure; bar(1:n,[real(V1(:,j)),imag(V1(:,j))],1,'grouped');
9 title(sprintf('Circulant matrix 1, eigenvector %d',j));
10 xlabel('{\bf vector component index}', 'fontsize',14);
11 ylabel('{\bf vector component value}', 'fontsize',14);
12 legend('real part','imaginary part','location','southwest');
13 print('-depsc2', sprintf('..PICTURES/circeiglev%d.eps',j));
14 end
15
16 C = gallery ('circul',rand(n,1)); [V2,D2] = eig(C);
17
18 for j=1:n
19 figure; bar(1:n,[real(V2(:,j)),imag(V2(:,j))],1,'grouped');

```

```

20 title(sprintf('Circulant matrix 2, eigenvector %d',j));
21 xlabel('{\bf vector component index}','fontsize',14);
22 ylabel('{\bf vector component value}','fontsize',14);
23 legend('real part','imaginary part','location','southwest');
24 print('-depsc2',sprintf('../PICTURES/circeig2ev%d.eps',j));
25 end

26
27 figure; plot(1:n,real(diag(D1)),'r+',1:n,imag(diag(D1)),'b+',...
28 1:n,real(diag(D2)),'m*',1:n,imag(diag(D2)),'k*');
29 ax = axis; axis([0 n+1 ax(3) ax(4)]);
30 xlabel('{\bf index of eigenvalue}','fontsize',14);
31 ylabel('{\bf eigenvalue}','fontsize',14);
32 legend('C_1: real(ev)', 'C_1: imag(ev)', 'C_2: real(ev)', 'C_2: ...
33 imag(ev)', 'location','northeast');

34 print -depsc2 '../PICTURES/circeigev.eps';

```

Random  $8 \times 8$  circulant matrices  $\mathbf{C}_1, \mathbf{C}_2$  ( $\rightarrow$  Def. 9.1.17)

eigenvalues (real part) ▷

Generated by MATLAB-command:

`C = gallery('circul',rand(n,1));`

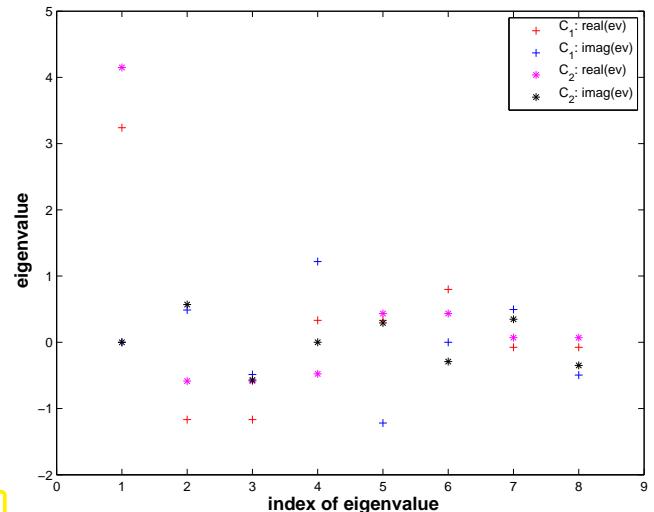
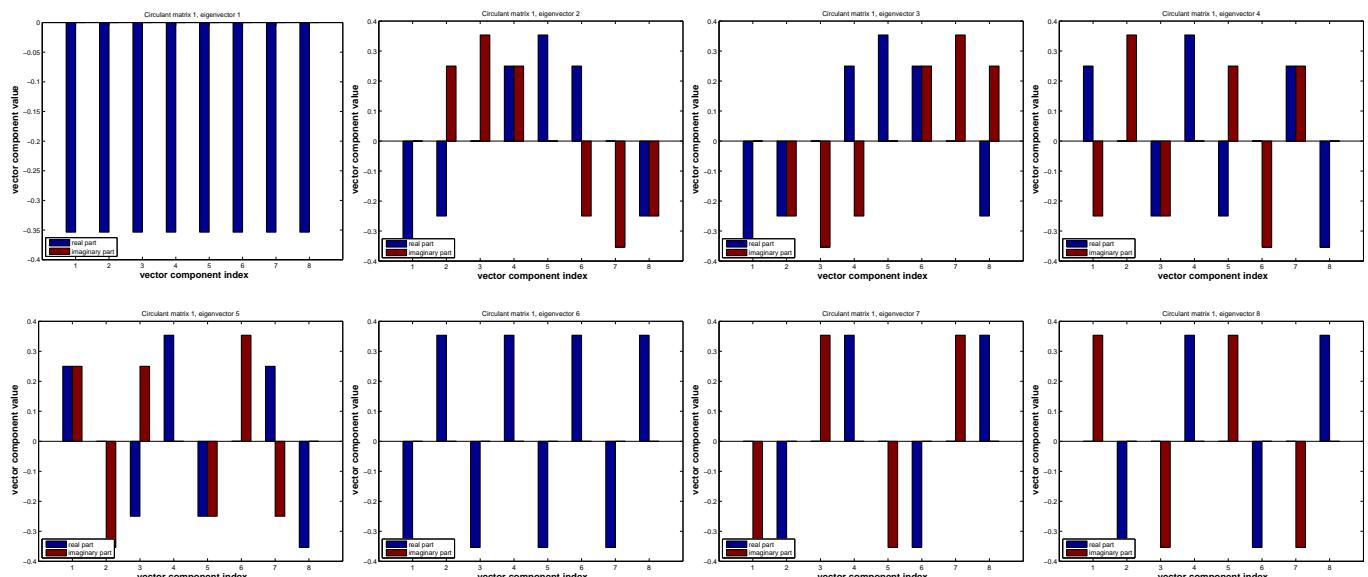


Fig. 297

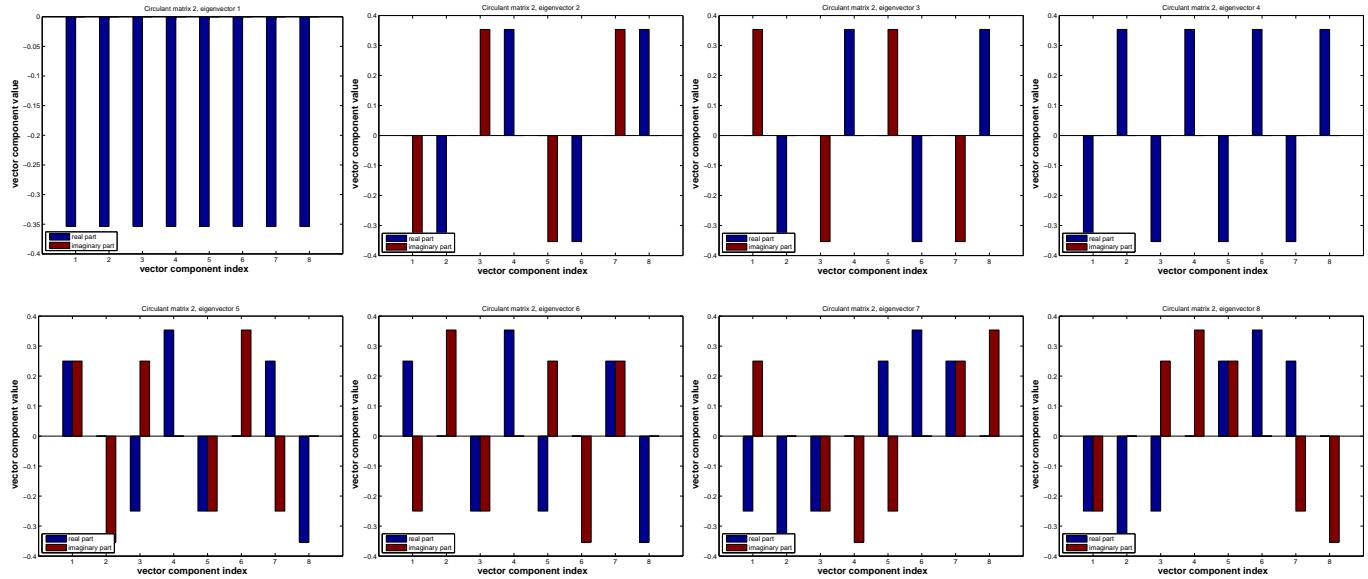
Little relationship between (complex!) eigenvalues can be observed, as can be expected from random matrices with entries  $\in [0,1]$ .

Now: the **surprise** . . .

Eigenvectors of matrix  $\mathbf{C}_1$ :

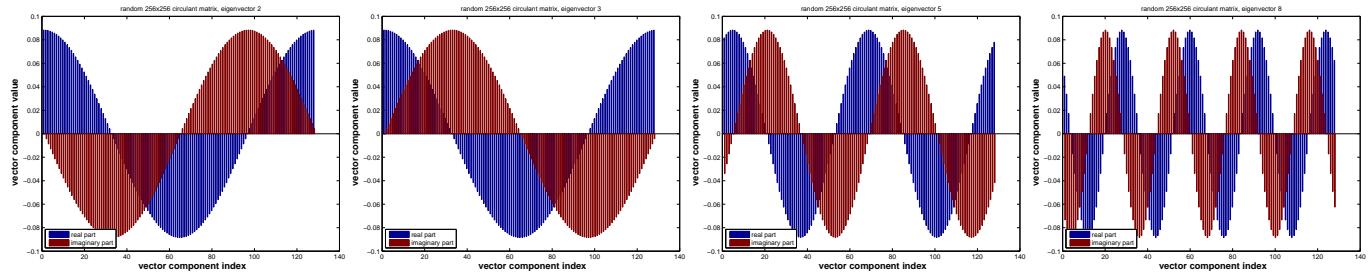


## Eigenvectors of matrix $\mathbf{C}_2$



Observation: different random circulant matrices have the **same eigenvectors!**

Eigenvectors of  $\mathbf{C} = \text{gallery}(\text{'circul'}, (1:128))$ :



The eigenvectors remind us of sampled *trigonometric functions*  $\cos(k/n)$ ,  $\sin(k/n)$ ,  $k = 0, \dots, n - 1$ !

### Remark 9.2.2 (Why using $\mathbb{K} = \mathbb{C}$ ?)

Ex. 9.2.1: *complex* eigenvalues/eigenvectors for general circulant matrices.

Recall from analysis: unified treatment of trigonometric functions via *complex exponential function*

$$\exp(it) = \cos(t) + i \sin(t), \quad t \in \mathbb{R}.$$

**C!** The field of complex numbers  $\mathbb{C}$  is the *natural framework* for the analysis of linear, time-invariant filters, and the development of algorithms for circulant matrices.

notation: *n*th root of unity  $\omega_n := \exp(-2\pi i/n) = \cos(2\pi/n) - i \sin(2\pi/n)$ ,  $n \in \mathbb{N}$

satisfies  $\overline{\omega_n} = \omega_n^{-1}$  ,  $\boxed{\omega_n^n = 1}$  ,  $\omega_n^{n/2} = -1$  ,  $\omega_n^k = \omega_n^{k+n} \quad \forall k \in \mathbb{Z}$  , (9.2.3)

$$\sum_{k=0}^{n-1} \omega_n^{kj} = \begin{cases} n & , \text{if } j = 0 \pmod{n} , \\ 0 & , \text{if } j \neq 0 \pmod{n} . \end{cases} \quad (9.2.4)$$

(9.2.4) is a simple consequence of the geometric sum formula

$$\begin{aligned} \sum_{k=0}^{n-1} q^k &= \frac{1-q^n}{1-q} \quad \forall q \in \mathbb{C} \setminus \{1\}, \quad n \in \mathbb{N}. \\ \Rightarrow \quad \sum_{k=0}^{n-1} \omega_n^{kj} &= \frac{1-\omega_n^{nj}}{1-\omega_n^j} = \frac{1-\exp(-2\pi ij)}{1-\exp(-2\pi ij/n)} = 0, \end{aligned} \quad (9.2.5)$$

because  $\exp(-2\pi ij) = \omega_n^{nj} = (\omega_n^n)^j = 1$  for all  $j \in \mathbb{Z}$ .

Now we want to confirm the conjecture gleaned from Ex. 9.2.1 that vectors with powers of roots of unity are eigenvectors for any circulant matrix. We do this by simple and straightforward computations:

Consider:  $\mathbf{C} \in \mathbb{C}^{n,n}$  circulant matrix ( $\rightarrow$  Def. 9.1.17),  $c_{ij} = u_{i-j}$ , for  $n$ -periodic sequence  $(u_k)_{k \in \mathbb{Z}}$ ,  $u_k \in \mathbb{C}$

$\mathbf{v}_k \in \mathbb{C}^n$  with  $\mathbf{v}_k := (\omega_n^{jk})_{j=0}^{n-1} \in \mathbb{C}^n$ ,  $k \in \{0, \dots, n-1\}$ .

$$\begin{aligned} &(u_{j-l}\omega_n^{lk})_{l \in \mathbb{Z}} \text{ is } n\text{-periodic!} \\ &(\mathbf{C}\mathbf{v}_k)_j = \sum_{l=0}^{n-1} u_{j-l}\omega_n^{lk} = \sum_{l=j-n+1}^j u_{j-l}\omega_n^{lk} \\ &= \sum_{l=0}^{n-1} u_l\omega_n^{(j-l)k} = \omega_n^{jk} \boxed{\sum_{l=0}^{n-1} u_l\omega_n^{-lk}} = \lambda_k \cdot \omega_n^{jk} = \lambda_k \cdot (\mathbf{v}_k)_j. \\ &\text{change of summation index} \quad \quad \quad \text{independent of } j! \end{aligned} \quad (9.2.6)$$

►  $\mathbf{v}$  is eigenvector of  $\mathbf{C}$  to eigenvalue  $\lambda_k = \sum_{l=0}^{n-1} u_l\omega_n^{-lk}$ .

Orthogonal trigonometric basis of  $\mathbb{C}^n$  = eigenvector basis for circulant matrices

$$\left\{ \begin{bmatrix} \omega_n^0 \\ \vdots \\ \omega_n^0 \end{bmatrix}, \begin{bmatrix} \omega_n^0 \\ \omega_n^1 \\ \vdots \\ \omega_n^{n-1} \end{bmatrix}, \dots, \begin{bmatrix} \omega_n^0 \\ \omega_n^{n-2} \\ \omega_n^{2(n-2)} \\ \vdots \\ \omega_n^{(n-1)(n-2)} \end{bmatrix}, \begin{bmatrix} \omega_n^0 \\ \omega_n^{n-1} \\ \omega_n^{2(n-1)} \\ \vdots \\ \omega_n^{(n-1)^2} \end{bmatrix} \right\}.$$

(9.2.4)  $\Rightarrow$  orthogonality of basis vectors:

$$\mathbf{v}_k := (\omega_n^{jk})_{j=0}^{n-1} \in \mathbb{C}^n: \quad \mathbf{v}_k^H \mathbf{v}_m = \sum_{j=0}^{n-1} \omega_n^{-jk} \omega_n^{jm} = \sum_{j=0}^{n-1} \omega_n^{(m-k)j} \stackrel{(9.2.4)}{=} 0, \text{ if } k \neq m. \quad (9.2.7)$$

Matrix of change of basis trigonometrical basis  $\rightarrow$  standard basis: Fourier-matrix

$$\mathbf{F}_n = \begin{bmatrix} \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \cdots & \omega_n^{2n-2} \\ \vdots & \vdots & & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} = \left( \omega_n^{lj} \right)_{l,j=0}^{n-1} \in \mathbb{C}^{n,n}. \quad (9.2.8)$$

**Lemma 9.2.9. Properties of Fourier matrix**

The scaled Fourier-matrix  $\frac{1}{\sqrt{n}} \mathbf{F}_n$  is unitary ( $\rightarrow$  Def. 4.2.2) :  $\mathbf{F}_n^{-1} = \frac{1}{n} \mathbf{F}_n^H = \frac{1}{n} \bar{\mathbf{F}}_n$ .

*Proof.* The lemma is immediate from (9.2.7) and (9.2.4), because

$$(\mathbf{F}_n \mathbf{F}_n^H)_{l,j} = \sum_{k=0}^{n-1} \omega_n^{(l-1)k} \bar{\omega}_n^{(j-1)k} = \sum_{k=0}^{n-1} \omega_n^{(l-1)k} \omega_n^{-(j-1)k} = \sum_{k=0}^{n-1} \omega_n^{k(l-j)}, \quad 1 \leq l, j \leq n.$$

□

**Remark 9.2.10 (Spectrum of Fourier matrix)**

$$\frac{1}{n^2} \mathbf{F}_n^4 = I \Rightarrow \sigma\left(\frac{1}{\sqrt{n}} \mathbf{F}_n\right) \subset \{1, -1, i, -i\},$$

because, if  $\lambda \in \mathbb{C}$  is an eigenvalue of  $\mathbf{F}_n$ , then there is an eigenvector  $\mathbf{x} \in \mathbb{C}^n \setminus \{0\}$  such that  $\mathbf{F}_n \mathbf{x} = \lambda \mathbf{x}$ , see Def. 7.1.1.

**Lemma 9.2.11. Diagonalization of circulant matrices ( $\rightarrow$  Def. 9.1.17)**

For any circulant matrix  $\mathbf{C} \in \mathbb{K}^{n,n}$ ,  $c_{ij} = u_{i-j}$ ,  $(u_k)_{k \in \mathbb{Z}}$   $n$ -periodic sequence, holds true

$$\mathbf{C} \bar{\mathbf{F}}_n = \bar{\mathbf{F}}_n \operatorname{diag}(d_1, \dots, d_n), \quad \mathbf{d} = \mathbf{F}_n(u_0, \dots, u_{n-1})^\top.$$

*Proof.* Straightforward computation, see (9.2.6).

□

Conclusion (from  $\bar{\mathbf{F}}_n = n \mathbf{F}_n^{-1}$ ):

$$\mathbf{C} = \mathbf{F}_n^{-1} \operatorname{diag}(d_1, \dots, d_n) \mathbf{F}_n. \quad (9.2.12)$$

Lemma 9.2.11, (9.2.12)  $\Rightarrow$  multiplication with Fourier-matrix will be crucial operation in algorithms for circulant matrices and discrete convolutions.

Therefore this operation has been given a special name:

**Definition 9.2.13. Discrete Fourier transform (DFT)**

The linear map  $\mathcal{F}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$ ,  $\mathcal{F}_n(\mathbf{y}) := \mathbf{F}_n \mathbf{y}$ ,  $\mathbf{y} \in \mathbb{C}^n$ , is called **discrete Fourier transform (DFT)**, i.e. for  $\mathbf{c} := \mathcal{F}_n(\mathbf{y})$

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj}, \quad k = 0, \dots, n-1. \quad (9.2.14)$$

Recall the convention also adopted for the discussion of the DFT: vector indexes range from 0 to  $n-1$ !

Terminology:  $\mathbf{c} = \mathbf{F}_n \mathbf{y}$  is also called the (discrete) Fourier transform of  $\mathbf{y}$

From  $\mathbf{F}_n^{-1} = \frac{1}{n} \bar{\mathbf{F}}_n$  ( $\rightarrow$  Lemma 9.2.9) we find the inverse discrete Fourier transform

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj} \Leftrightarrow y_j = \frac{1}{n} \sum_{k=0}^{n-1} c_k \omega_n^{-kj} \quad (9.2.15)$$

MATLAB-functions for discrete Fourier transform (and its inverse):

DFT: `c=fft(y)`  $\leftrightarrow$  inverse DFT: `y=ifft(c);`

### 9.2.1 Discrete convolution via DFT

Recall

$$\begin{aligned} \text{discrete periodic convolution } z_k &= \sum_{j=0}^{n-1} u_{k-j} x_j \quad (\rightarrow \text{Def. 9.1.13}), k = 0, \dots, n-1 \\ &\uparrow \\ \text{multiplication with circulant matrix } (\rightarrow \text{Def. 9.1.17}) \quad \mathbf{z} = \mathbf{Cx}, \quad \mathbf{C} := (u_{i-j})_{i,j=1}^n. \end{aligned}$$

Idea:

$$(9.2.12) \quad \mathbf{z} = \mathbf{F}_n^{-1} \operatorname{diag}(\mathbf{F}_n \mathbf{u}) \mathbf{F}_n \mathbf{x}$$

Listing 9.2: discrete periodic convolution: straightforward implementation

```

1 function z=pconv(u,x)
2 n = length(x); z = zeros(n,1);
3 for i=1:n, z(i)=dot(conj(u),
4   x([i:-1:1,n:-1:i+1]));
5 end

```

Listing 9.3: discrete periodic convolution: DFT implementation

```

1 function z=pconvfft(u,x)
2 % Implementation of (9.2.12), cf.
3 % Lemma 9.2.11
4 z = ifft(fft(u).*fft(x));

```

Rem. 9.1.19: discrete convolution of  $n$ -vectors ( $\rightarrow$  Def. 9.1.7) by *periodic* discrete convolution of  $2n - 1$ -vectors (obtained by zero padding, see Rem. 9.1.19):

Implementation of  
discrete convolution  
( $\rightarrow$   
Def. 9.1.7)  
based  
on  
periodic discrete convolution  
  
Built-in MATLAB-function:  
`y = conv(h,x);`

( $\rightarrow$   
on  
 $\triangleright$ )

Listing 9.4: discrete convolution: DFT implementation

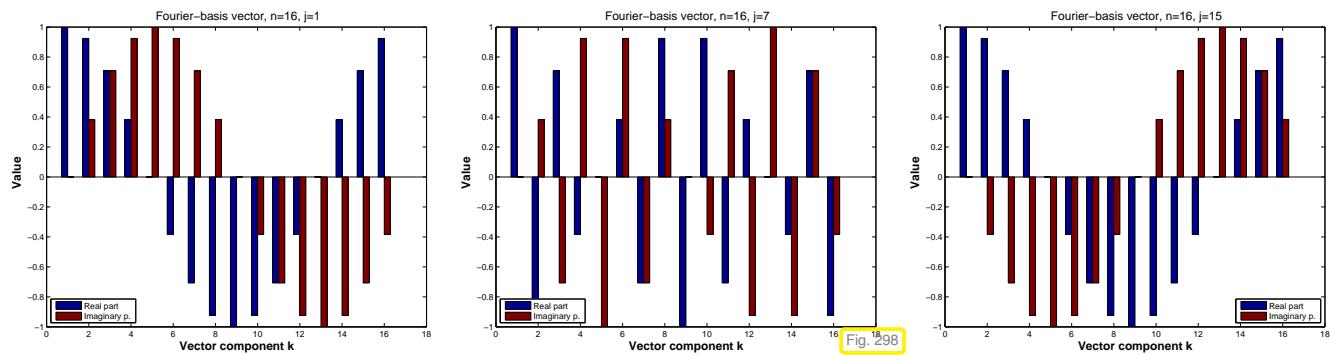
```

1 function y = myconv(h,x)
2 n = length(h);
3 % Zero padding, cf. (9.1.20)
4 h = [h;zeros(n-1,1)];
5 x = [x;zeros(n-1,1)];
6 % Periodic discrete convolution of length
7 % 2n - 1
8 y = pconvfft(h,x);

```

### 9.2.2 Frequency filtering via DFT

The trigonometric basis vectors, when interpreted as time-periodic signals, represent harmonic oscillations. This is illustrated when plotting some vectors of the trigonometric basis ( $n = 16$ ):



“slow oscillation/low frequency”      “fast oscillation/high frequency”      “slow oscillation/low frequency”

- Dominant coefficients of a signal after transformation to trigonometric basis indicate dominant frequency components.

Terminology: coefficients of a signal w.r.t. trigonometric basis = signal in **frequency domain** (ger.: Frequenzbereich), original signal = **time domain** (ger.: Zeitbereich).

Recall: DFT (9.2.14) and inverse DFT (9.2.15)

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj} \quad \Leftrightarrow \quad y_j = \frac{1}{n} \sum_{k=0}^{n-1} c_k \omega_n^{-kj} \quad (9.2.15)$$

Consider  $y_k \in \mathbb{R}$   $\Rightarrow c_k = \bar{c}_{n-k}$ , because  $\omega_n^{kj} = \bar{\omega}_n^{(n-k)j}$ , and  $n = 2m + 1$

$$\begin{aligned} ny_j &= c_0 + \sum_{k=1}^m c_k \omega_n^{-kj} + \sum_{k=m+1}^{2m} c_k \omega_n^{-kj} = c_0 + \sum_{k=1}^m c_k \omega_n^{-kj} + c_{n-k} \omega_n^{(k-n)j} \\ &= c_0 + 2 \sum_{k=1}^m \operatorname{Re}(c_k) \cos(2\pi kj/n) + \operatorname{Im}(c_k) \sin(2\pi kj/n), \end{aligned}$$

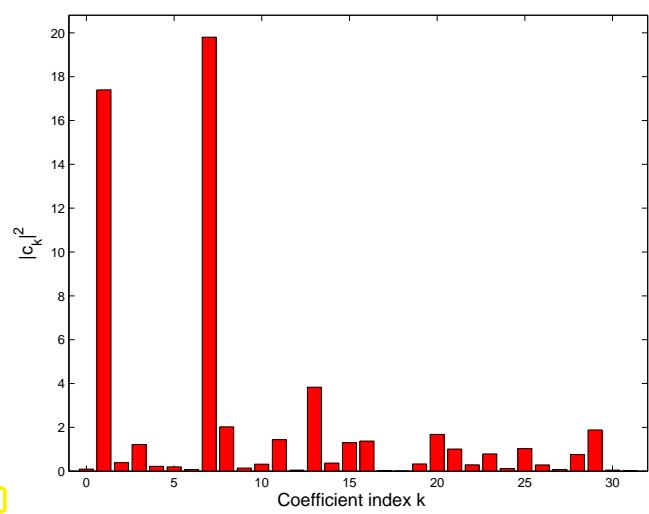
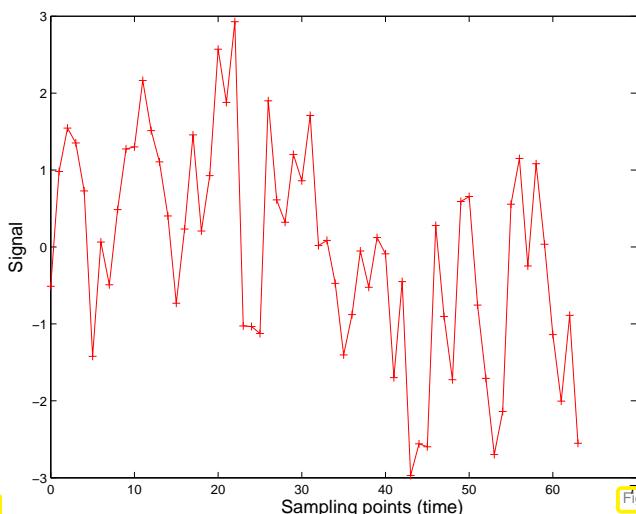
since  $\omega_n^\ell = \cos(2\pi \ell/n) + i \sin(2\pi \ell/n)$ .

- $|c_k|, |c_{n-k}|$  measures the strength with which an oscillation with frequency  $k$  is represented in the signal,  $0 \leq k \leq \lfloor \frac{n}{2} \rfloor$ .

### Example 9.2.16 (Frequency identification with DFT)

Extraction of characteristic frequencies from a distorted discrete periodical signal:

```
1 t = 0:63; x = sin(2*pi*t/64)+sin(7*2*pi*t/64);
2 y = x + randn(size(t)); %distortion
```



Generating Fig. 300:

```

1 c = fft(y); p = (c.*conj(c))/64;
2
3 figure('Name','power spectrum');
4 bar(0:31,p(1:32),'r');
5 set(gca,'fontsize',14);
6 axis([-1 32 0 max(p)+1]);
7 xlabel('{\bf index k of Fourier coefficient}', 'Fontsize',14);
8 ylabel('{\bf |c_k|^2}', 'Fontsize',14);

```

Frequencies present in unperturbed signal become evident in frequency domain.

### Example 9.2.17 (Detecting periodicity in data)

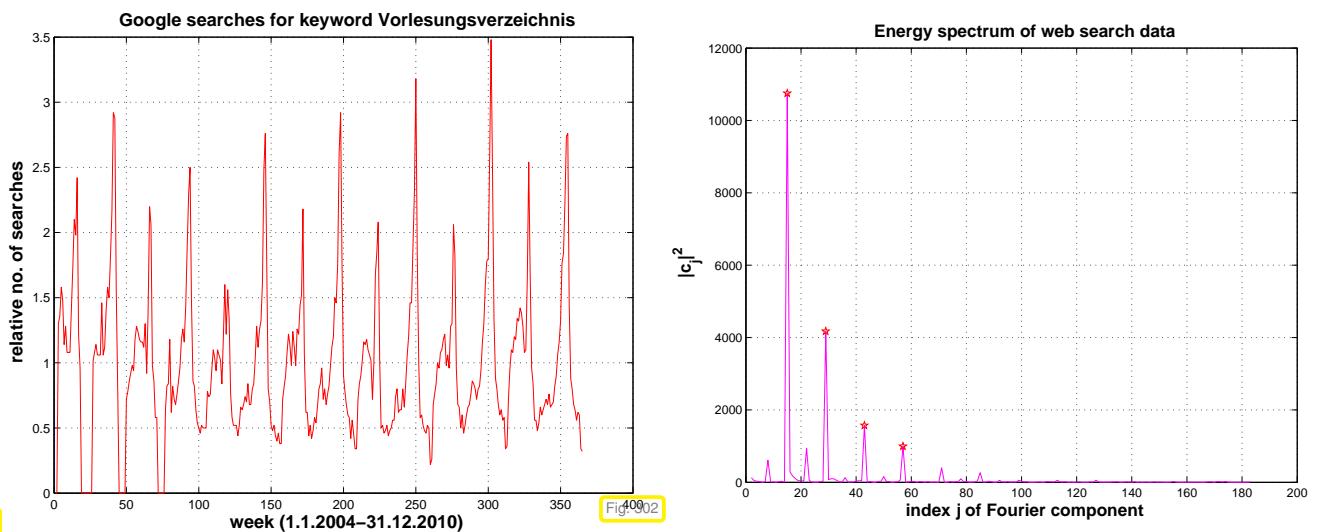
DFT: a computer's eye for periodic patterns in data

Listing 9.5: Extraction of periodic patterns by DFT

```

1 % Tracking of periodicity in data
2 % Data obtained from Google Trends, keyword "Vorlesungsverzeichnis"
3 % Exported as .csv-file, non data lines removed, preprocessed by command
4 %   cut -f 2 -d,
5
6 % read ASCII data from file
7 y = dlmread('trend.dat'); n = length(y);
8
9 figure('name','data');
10 plot(y,'r-'); grid on;
11 title('{\bf Google searches for keyword
12   Vorlesungsverzeichnis}', 'fontsize',14);
13 xlabel('{\bf week (1.1.2004-31.12.2010)sa}', 'fontsize',14);
14 ylabel('{\bf relative no. of searches}', 'fontsize',14);
15 print -depsc2 '../PICTURES/searchdata.eps';
16
17 % Periodicity analysis by means of DFT
18 c = fft(y);
19 p = abs(c(2:floor((n+1)/2))).^2; % Power spectrum
20 figure('name','Fourier spectrum');
21 plot(2:floor((n+1)/2),p,'m-'); grid on; hold on;
22 [mx,idx] = sort(p,'descend');
23 plot(1+idx(1:4),p(idx(1:4)),'rp');
24 xlabel('{\bf index j of Fourier component}', 'fontsize',14);
25 ylabel('{\bf |c_j|^2}', 'fontsize',14);
26 title('{\bf Energy spectrum of web search data}', 'fontsize',14);
27 print -depsc2 '../PICTURES/fourierdata.eps';

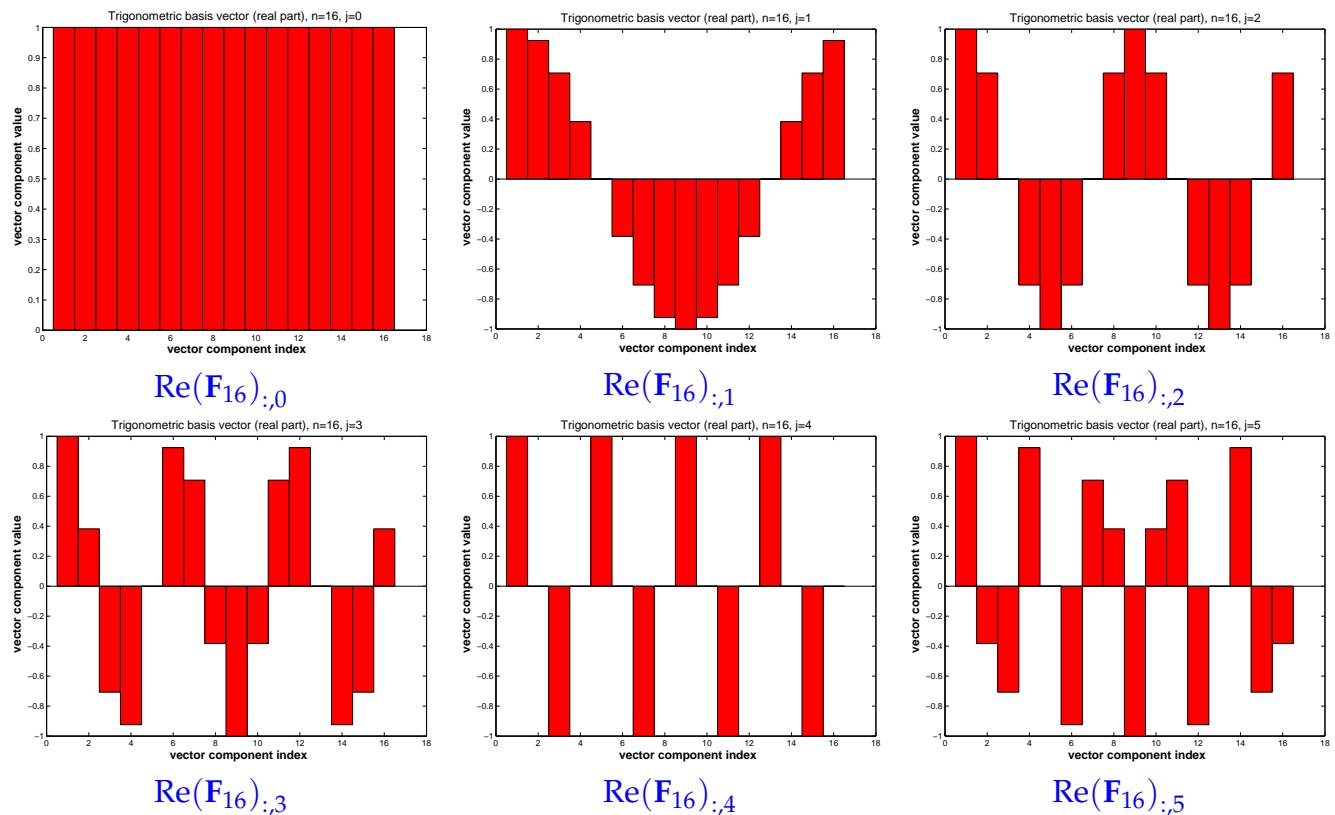
```

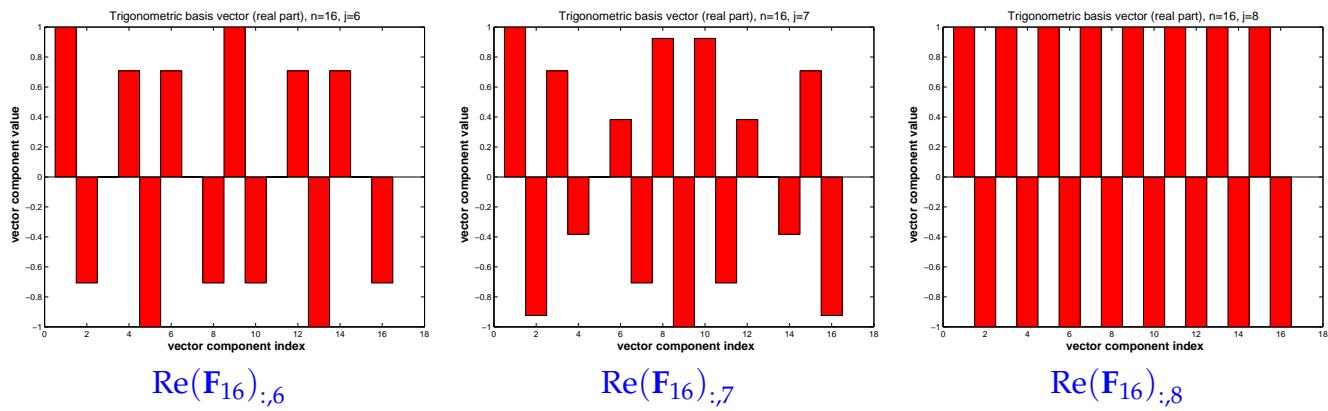


Pronounced peaks in the power spectrum point to periodic structure of the data. Location of peaks tells lengths of dominant periods.

### Remark 9.2.18 (“Low” and “high” frequencies)

Plots of real parts of trigonometric basis vectors  $(\mathbf{F}_n)_{:,j}$  (= columns of Fourier matrix),  $n = 16$ .





By elementary trigonometric identities:

$$\text{Re}(\mathbf{F}_n)[:,j] = \left( \text{Re} \omega_n^{(j-1)k} \right)_{k=0}^{n-1} = (\text{Re} \exp(2\pi i(j-1)k/n))_{k=0}^{n-1} = (\cos(2\pi(j-1)x))_{x=0, \frac{1}{n}, \dots, 1-\frac{1}{n}}.$$

Slow oscillations/low frequencies  $\leftrightarrow j \approx 1$  and  $j \approx n$ .

Fast oscillations/high frequencies  $\leftrightarrow j \approx n/2$ .

- Frequency filtering of real discrete periodic signals by suppressing certain “Fourier coefficients”.

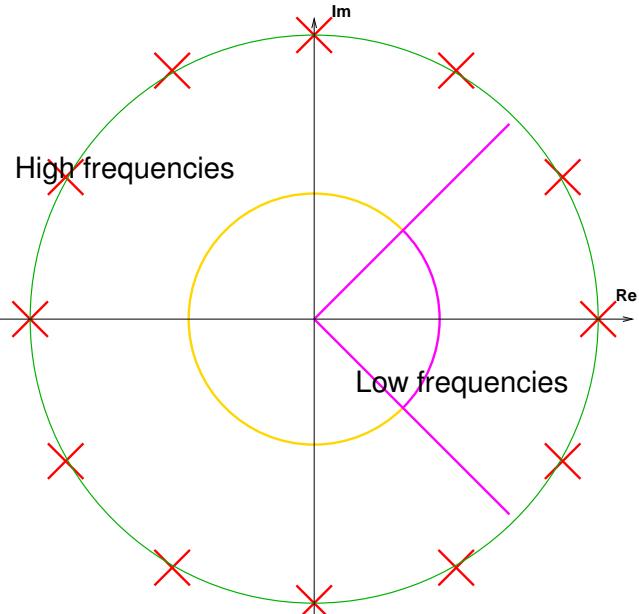
Listing 9.6: DFT-based frequency filtering

```

1 function [low,high] =
2     freqfilter(y,k)
3 m = length(y)/2; c = fft(y);
4 clow = c; clow(m+1-k:m+1+k) = 0;
5 chigh = c-clow;
6 low = real(ifft(clow));
7 high = real(ifft(chigh));

```

(can be optimised exploiting  $y_j \in \mathbb{R}$  and  $c_{n/2-k} = \bar{c}_{n/2+k}$ )



Map  $y \mapsto \text{low}$  (in Listing 9.6)  $\hat{=} \text{low pass filter}$  (ger.: Tiefpass).

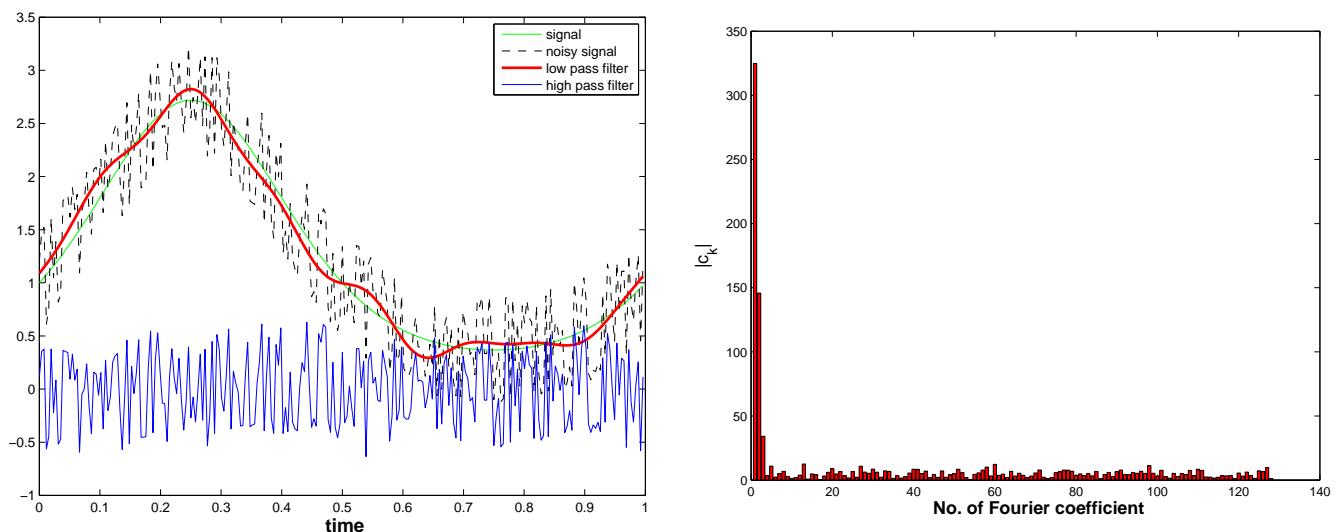
Map  $y \mapsto \text{high}$  (in Listing 9.6)  $\hat{=} \text{high pass filter}$  (ger.: Hochpass).

### Example 9.2.19 (Frequency filtering by DFT)

Noisy signal:

```
n = 256; y = exp(sin(2*pi*((0:n-1)')/n)) + 0.5*sin(exp(1:n)');
```

Frequency filtering by Listing 9.6 with  $k = 120$ .



Low pass filtering can be used for *denoising*, that is, the removal of high frequency perturbations of a signal.

### Example 9.2.20 (Sound filtering by DFT)

Listing 9.7: DFT based sound compression

```

1 % Low pass sound filtering by DFT
2
3 % Read sound data
4 [y,Fs,nbits] = wavread('hello.wav');
5 sound(y,Fs);
6
7 n = length(y);
8 fprintf('Read wav File: %d samples, rate = %d/s, nbits = %d\n',
9     n,Fs,nbits);
10 k = 1; s{k} = y; leg{k} = 'Sampled signal';
11
12 c = fft(y);
13
14 figure('name','sound signal');
15 plot((22000:44000)/Fs,s{1}(22000:44000),'r-');
16 title('samples sound signal','fontsize',14);
17 xlabel('{\bf time[s]}','fontsize',14);
18 ylabel('{\bf sound pressure}','fontsize',14);
19 grid on;
20
21 % print -depsc2 '../PICTURES/soundsignal.eps';
22
23 figure('name','sound frequencies');
24 plot(1:n,abs(c).^2,'m-');
25 title('power spectrum of sound signal','fontsize',14);
26 xlabel('{\bf index k of Fourier coefficient}','fontsize',14);
27 ylabel('{\bf |c_k|^2}','fontsize',14);
28 grid on;
```

```

28
29 % print -depsc2 '../PICTURES/soundpower.eps';
30
31 figure('name','sound frequencies');
32 plot(1:3000,abs(c(1:3000)).^2,'b-');
33 title('low frequency power spectrum','fontsize',14);
34 xlabel('{\bf index k of Fourier coefficient}','fontsize',14);
35 ylabel('{\bf |c_k|^2}','fontsize',14);
36 grid on;
37
38 % print -depsc2 '../PICTURES/soundlowpower.eps';
39
40 for m=[1000,3000,5000]
41
42 % Low pass filtering
43 cf = zeros(1,n);
44 cf(1:m) = c(1:m); cf(n-m+1:end) = c(n-m+1:end);
45
46 % Reconstruct filtered signal
47 yf = ifft(cf);
48 % No idea why this is necessary
49 wavwrite(yf,Fs,nbits,sprintf('hello%d.wav',m));
50 cy = wavread(sprintf('hello%d.wav',m));
51 sound(cy,Fs,nbits);
52
53 k = k+1;
54 s{k} = real(yf);
55 leg{k} = sprintf('cutt-off = %d',m);
56 end
57
58 % Plot original signal and filtered signals
59 figure('name','sound filtering');
60 plot((30000:32000)/Fs,s{1}(30000:32000),'r-',...
61 (30000:32000)/Fs,s{2}(30000:32000),'b--',...
62 (30000:32000)/Fs,s{3}(30000:32000),'m--',...
63 (30000:32000)/Fs,s{2}(30000:32000),'k--');
64 xlabel('{\bf time[s]}','fontsize',14);
65 ylabel('{\bf sound pressure}','fontsize',14);
66 legend(leg,'location','southeast');
67
68 % print -depsc2 '../PICTURES/soundfiltered.eps';

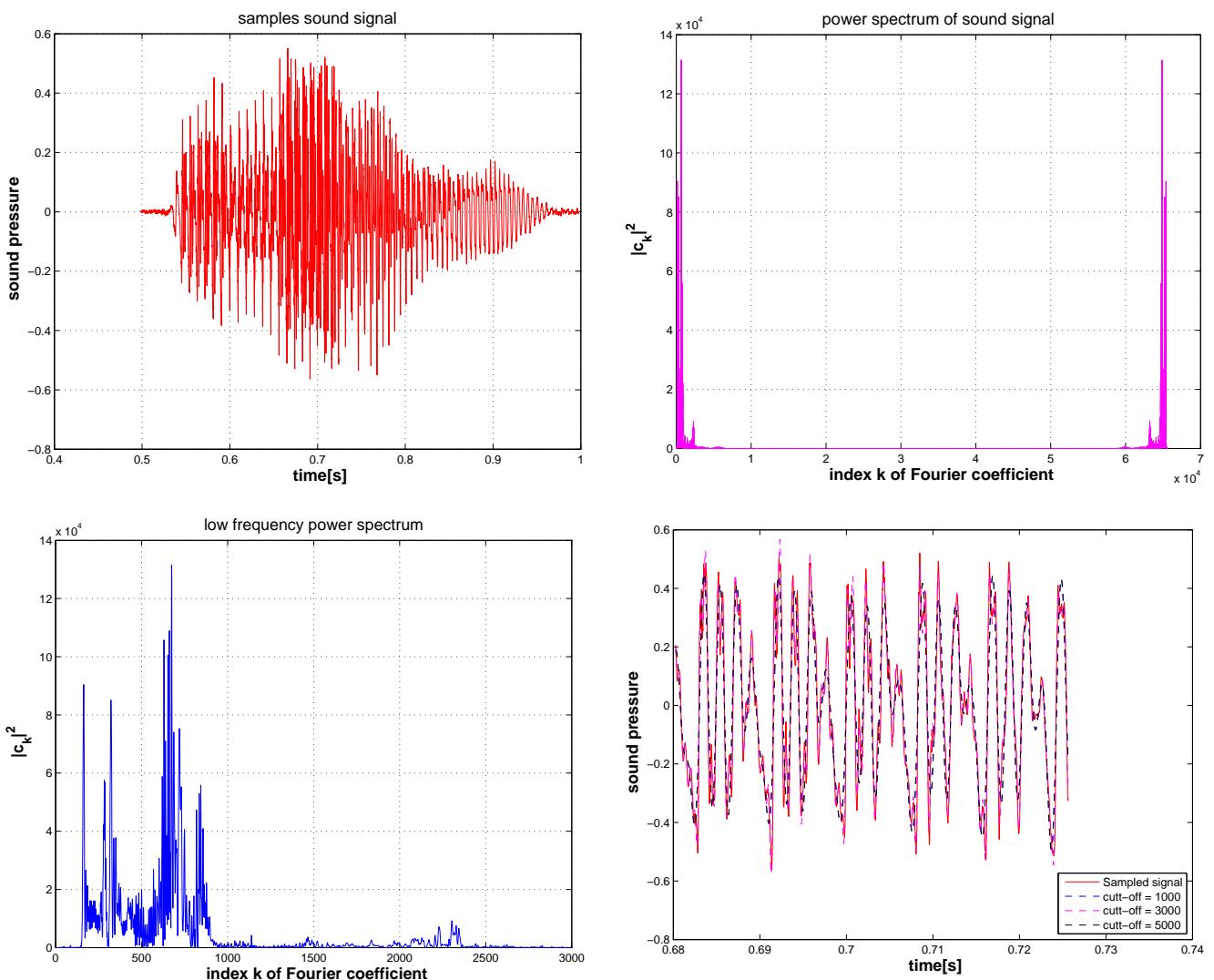
```

### MATLAB-code 9.2.21: DFT based low pass frequency filtering of sound

```

1 [y,sf,nb] = wavread('hello.wav');
2 c = fft(y); c(m+1:end-m) = 0;
3 wavwrite(ifft(c),sf,nb,'filtered.wav');

```



The **power spectrum** of a signal  $\mathbf{y} \in \mathbb{C}^n$  is the vector  $(|c_j|^2)_{j=0}^{n-1}$ , where  $\mathbf{c} = \mathbf{F}_n \mathbf{y}$  is the discrete Fourier transform of  $\mathbf{y}$ .

### 9.2.3 Real DFT

Signal obtained from sampling a time-dependent voltage: a **real** vector.

Aim: efficient DFT (Def. 9.2.13)  $(c_0, \dots, c_{n-1})$  for *real* coefficients  $(y_0, \dots, y_{n-1})^\top \in \mathbb{R}^n$ ,  $n = 2m$ ,  $m \in \mathbb{N}$ .

If  $y_j \in \mathbb{R}$  in DFT formula (9.2.14), we obtain redundant output

$$\begin{aligned} \omega_n^{(n-k)j} &= \overline{\omega_n^{kj}}, \quad k = 0, \dots, n-1, \\ \Rightarrow \bar{c}_k &= \sum_{j=0}^{n-1} y_j \overline{\omega_n^{kj}} = \sum_{j=0}^{n-1} y_j \omega_n^{(n-k)j} = c_{n-k}, \quad k = 1, \dots, n-1. \end{aligned}$$

**Idea:** map  $\mathbf{y} \in \mathbb{R}^n$  to  $\mathbb{C}^m$  and use DFT of length  $m$ .

$$h_k = \sum_{j=0}^{m-1} (y_{2j} + iy_{2j+1}) \omega_m^{jk} = \boxed{\sum_{j=0}^{m-1} y_{2j} \omega_m^{jk}} + i \cdot \boxed{\sum_{j=0}^{m-1} y_{2j+1} \omega_m^{jk}}, \quad (9.2.22)$$

$$\bar{h}_{m-k} = \sum_{j=0}^{m-1} \overline{y_{2j} + iy_{2j+1}} \overline{\omega_m^{j(m-k)}} = \boxed{\sum_{j=0}^{m-1} y_{2j} \omega_m^{jk}} - i \cdot \boxed{\sum_{j=0}^{m-1} y_{2j+1} \omega_m^{jk}}. \quad (9.2.23)$$

$$\Rightarrow \boxed{\sum_{j=0}^{m-1} y_{2j} \omega_m^{jk}} = \frac{1}{2}(h_k + \bar{h}_{m-k}) \quad , \quad \boxed{\sum_{j=0}^{m-1} y_{2j+1} \omega_m^{jk}} = -\frac{1}{2}i(h_k - \bar{h}_{m-k}).$$

Use simple identities for roots of unity:

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{jk} = \boxed{\sum_{j=0}^{m-1} y_{2j} \omega_m^{jk}} + \omega_n^k \cdot \boxed{\sum_{j=0}^{m-1} y_{2j+1} \omega_m^{jk}}. \quad (9.2.24)$$

$$\Rightarrow \begin{cases} c_k = \frac{1}{2}(h_k + \bar{h}_{m-k}) - \frac{1}{2}i\omega_n^k(h_k - \bar{h}_{m-k}), & k = 0, \dots, m-1, \\ c_m = \operatorname{Re}\{h_0\} - \operatorname{Im}\{h_0\}, \\ c_k = \bar{c}_{n-k}, & k = m+1, \dots, n-1. \end{cases} \quad (9.2.25)$$

Listing 9.8: DFT of real vectors

```

1 function c = fftreal(y)
2 n = length(y); m = n/2;
3 if (mod(n, 2) ~= 0), error('n must be even'); end
4 y = y(1:2:n)+i*y(2:2:n); h = fft(y); h =
5 [h; h(1)];
6 c = 0.5*(h+conj(h(m+1:-1:1))) - ...
7 (0.5*i*exp(-2*pi*i/n).^(0:m)').*...
8 (h-conj(h(m+1:-1:1)));
9 c = [c; conj(c(m:-1:2))];

```

MATLAB-Implementation  
(by a DFT of length  $n/2$ ):

(Note: not really optimal  
MATLAB implementation)

## 9.2.4 Two-dimensional DFT

A natural analogy:

one-dimensional data (“signal”)  $\longleftrightarrow$  vector  $\mathbf{y} \in \mathbb{C}^n$ ,

two-dimensional data (“image”)  $\longleftrightarrow$  matrix.  $\mathbf{Y} \in \mathbb{C}^{m,n}$

Two-dimensional trigonometric basis of  $\mathbb{C}^{m,n}$ :

tensor product matrices  $\{(\mathbf{F}_m)_{:,j}(\mathbf{F}_n)_{:\ell}^\top, 1 \leq j \leq m, 1 \leq \ell \leq n\}$ . (9.2.26)

Basis transform: for  $y_{j_1, j_2} \in \mathbb{C}, 0 \leq j_1 < m, 0 \leq j_2 < n$  compute (nested DFTs !)

$$c_{k_1, k_2} = \sum_{j_1=0}^{m-1} \sum_{j_2=0}^{n-1} y_{j_1, j_2} \omega_m^{j_1 k_1} \omega_n^{j_2 k_2}, \quad 0 \leq k_1 < m, 0 \leq k_2 < n.$$

MATLAB function:  $\text{fft2}(\mathbf{Y})$ .

Two-dimensional DFT by *nested one-dimensional DFTs* (9.2.14):

$$\text{fft2}(Y) = \text{fft}(\text{fft}(Y) \cdot') \cdot'$$

Here:  $\cdot'$  simply transposes the matrix (no complex conjugation)

### Example 9.2.27 (Deblurring by DFT)

Gray-scale pixel image  $P \in \mathbb{R}^{m,n}$ , actually  $P \in \{0, \dots, 255\}^{m,n}$ , see Ex. 7.3.24.

$(p_{l,k})_{l,k \in \mathbb{Z}}$   $\hat{=}$  periodically extended image:

$$p_{l,j} = (P)_{l+1,j+1} \quad \text{for } 1 \leq l \leq m, 1 \leq j \leq n, \quad p_{l,j} = p_{l+m,j+n} \quad \forall l, k \in \mathbb{Z}.$$

**Blurring** = pixel values get replaced by weighted averages of near-by pixel values  
(effect of distortion in optical transmission systems)

$$c_{l,j} = \sum_{k=-L}^L \sum_{q=-L}^L s_{k,q} p_{l+k,j+q}, \quad 0 \leq l < m, \quad 0 \leq j < n, \quad L \in \{1, \dots, \min\{m, n\}\}. \quad (9.2.28)$$

blurred image      point spread function

Does this ring a bell? Hidden in (9.2.28) are two (periodic) discrete convolutions, see Def. 9.1.13.

Hardly surprising that DFT comes handy for reversing the effect of the blurring!

Usually:  $L$  small,  $s_{k,m} \geq 0$ ,  $\sum_{k=-L}^L \sum_{q=-L}^L s_{k,q} = 1$  (an averaging)

Used in test calculations:  $L = 5$

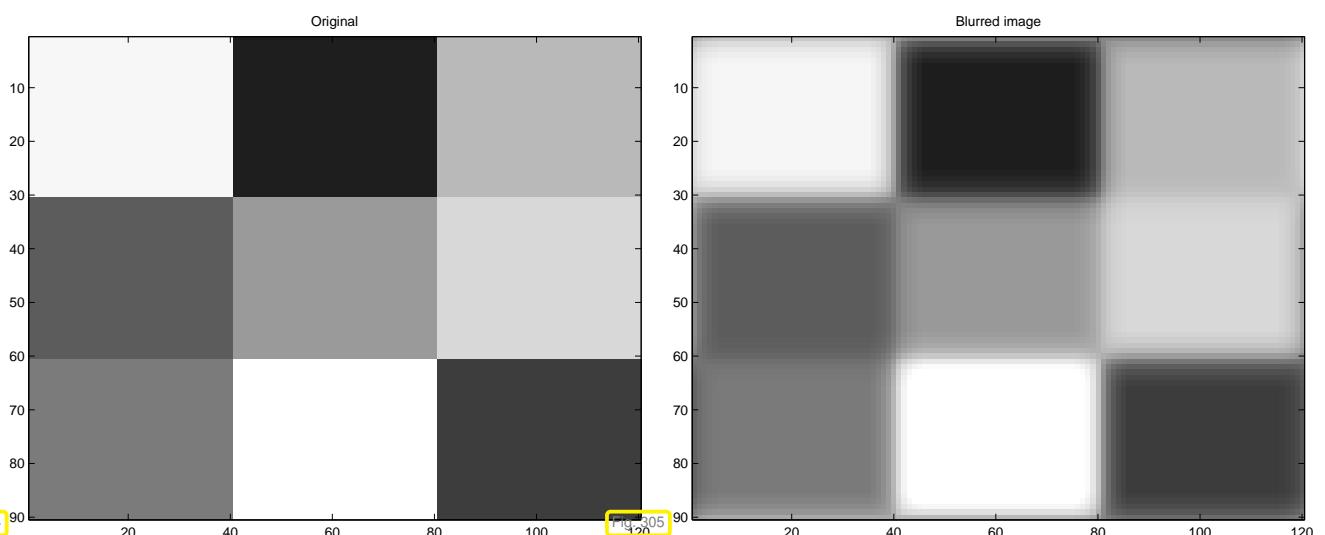
$$s_{k,q} = \frac{1}{1 + k^2 + q^2}.$$

Listing 9.9: point spread function

```

1 function S = psf(L)
2 [X, Y] = meshgrid(-L:1:L, -L:1:L);
3 S = 1 ./ (1 + X.*X + Y.*Y);
4 S = S / sum(sum(S));

```



Listing 9.10: MATLAB deblurring experiment

```

1 % Generate artificial "image"
2 P = kron(magic(3), ones(30, 40)) * 31;
3 col = [0:1/254:1]' * [1, 1, 1];
4 figure; image(P); colormap(col); title('Original');

```

```

5 print -depsc2 '../PICTURES/dborigimage.eps';
6 % Generate point spread function
7 L = 5; S = psf(L);
8 % Blur image
9 C = blur(P,S);
10 figure; image(floor(C)); colormap(col); title('Blurred image');
11 print -depsc2 '../PICTURES/dbblurredimage.eps';
12 % Deblur image
13 D = deblur(C,S);
14 figure; image(floor(real(D))); colormap(col);
15 fprintf('Difference of images (Frobenius norm): %f\n', norm(P-D));

```

Listing 9.11: blurring operator

```

1 function C = blur(P,S)
2 [m,n] = size(P); [M,N] = size(S);
3 if (M ~= N), error('S not quadratic'); end
4 L = (M-1)/2; C = zeros(m,n);
5 for l=1:m, for j=1:n
6 s = 0;
7 for k=1:(2*L+1), for q=1:(2*L+1)
8 kl = l+k-L-1;
9 if (kl < 1), kl = kl + m; end
10 if (kl > m), kl = kl - m; end
11 jm = j+q-L-1;
12 if (jm < 1), jm = jm + n; end
13 if (jm > n), jm = jm - n; end
14 s = s+P(kl,jm)*S(k,q);
15 end, end
16 C(l,j) = s;
17 end, end

```

Note:

(9.2.28) defines a linear operator  
 $\mathcal{B} : \mathbb{R}^{m,n} \mapsto \mathbb{R}^{m,n}$   
("blurring operator")

Note: more efficient implementation via MATLAB function `conv2(P,S)`

Recall: derivation of (9.2.6) and Lemma 9.2.11. Try this in 2D!

$$\left( \mathcal{B}(\left( \omega_m^{\nu k} \omega_n^{\mu q} \right)_{k,q \in \mathbb{Z}}) \right)_{l,j} = \sum_{k=-L}^L \sum_{q=-L}^L s_{k,q} \omega_m^{\nu(l+k)} \omega_n^{\mu(j+q)} = \omega_m^{\nu l} \omega_n^{\mu j} \underbrace{\sum_{k=-L}^L \sum_{q=-L}^L s_{k,q} \omega_m^{\nu k} \omega_n^{\mu q}}_{\text{2-dimensional DFT of point spread function!}}.$$

►  $\mathbf{V}_{\nu,\mu} := (\omega_m^{\nu k} \omega_n^{\mu q})_{k,q \in \mathbb{Z}}$ ,  $0 \leq \mu < m$ ,  $0 \leq \nu < n$  are the eigenvectors of  $\mathcal{B}$ :

$$\mathcal{B}\mathbf{V}_{\nu,\mu} = \lambda_{\nu,\mu} \mathbf{V}_{\nu,\mu} \quad , \quad \text{eigenvalue} \quad \lambda_{\nu,\mu} = \underbrace{\sum_{k=-L}^L \sum_{q=-L}^L s_{k,q} \omega_m^{\nu k} \omega_n^{\mu q}}_{\text{2-dimensional DFT of point spread function!}} \quad (9.2.29)$$

Listing 9.12: DFT based deblurring

```

1 function D = deblur(C,S,tol)
2 [m,n] = size(C); [M,N] = size(S);
3 if (M ~= N), error('S not quadratic'); end
4 L = (M-1)/2; Spad = zeros(m,n);
5 % Zero padding
6 Spad(1:L+1,1:L+1) = S(L+1:end,L+1:end);
7 Spad(m-L+1:m,n-L+1:n) = S(1:L,1:L);
8 Spad(1:L+1,n-L+1:n) = S(L+1:end,1:L);
9 Spad(m-L+1:m,1:L+1) = S(1:L,L+1:end);
10 % Inverse of blurring operator
11 SF = fft2(Spad);
12 % Test for invertibility
13 if (nargin < 3), tol = 1E-3; end
14 if (min(min(abs(SF))) < tol*max(max(abs(SF)))) )
15 error('Deblurring impossible');
16 end
17 % DFT based deblurring
18 D = fft2(ifft2(C)./SF);

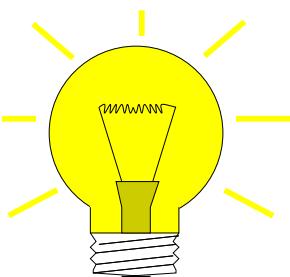
```

Inversion of blurring  
operator  
 $\Downarrow$   
componentwise scaling in  
“Fourier domain”

## 9.2.5 Semi-discrete Fourier Transform [63, Sect. 10.11]

Starting from Ex. 9.1.10 we mainly looked at time-discrete  $n$ -periodic signals, which can be mapped to vectors  $\in \mathbb{R}^n$ . This led to discrete periodic convolution ( $\rightarrow$  Def. 9.1.13) and the discrete Fourier transform (DFT) ( $\rightarrow$  Def. 9.2.13) as (bi-)linear mappings in  $\mathbb{C}^n$ .

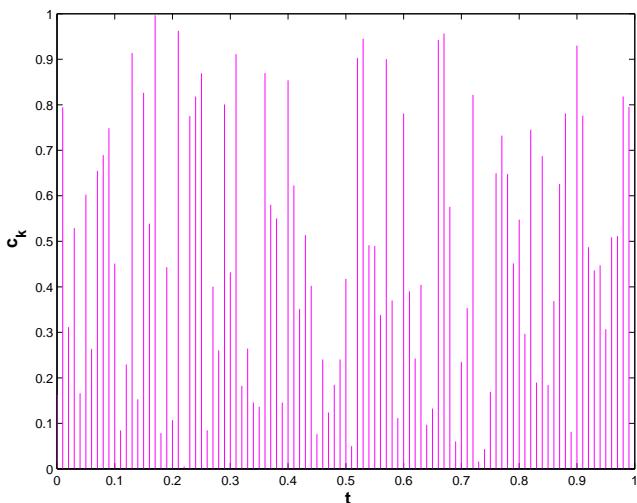
In this section we are concerned with non-periodic signals of infinite duration.



Idea: Study the limit  $n \rightarrow \infty$  for the  $n$ -periodic setting and DFT.

$(y_k)_{k \in \mathbb{Z}}$ :  $n$ -periodic sequence (signal),  $n = 2m + 1, m \in \mathbb{N}$ :

$$\text{DFT: } c_k = \sum_{j=-m}^m y_j \exp(-2\pi i \frac{kj}{n}), \quad k = 0, \dots, n-1. \quad (9.2.30)$$



▷ “Squeezing” a vector  $\in \mathbb{R}^n$  into  $[0, 1]$ .

$$c_k \leftrightarrow c(t_k),$$

$$t_k = \frac{k}{n}, \quad k = 0, \dots, n-1.$$

Now we associate a point  $t_k \in [0, 1]$  with each index  $k$

$$k \in \{0, \dots, n-1\} \iff t_k := \frac{k}{n}. \quad (9.2.31)$$

Thus, formally, we can rewrite (9.2.30) as

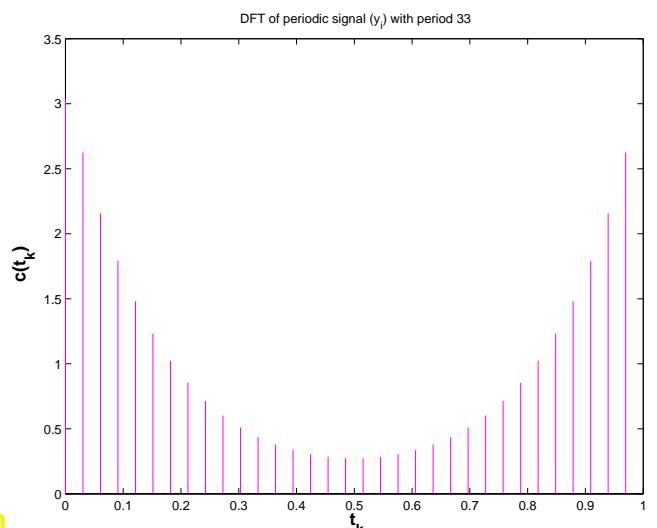
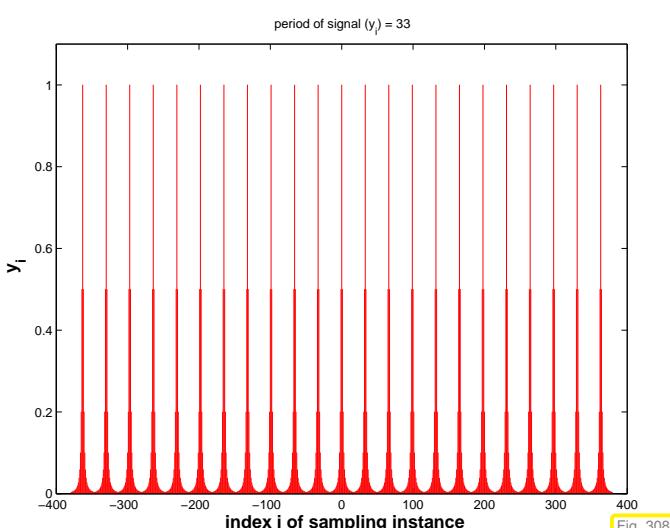
$$\text{DFT: } c(t_k) := c_k = \sum_{j=-m}^m y_j \exp(-2\pi i j t_k), \quad k = 0, \dots, n-1. \quad (9.2.32)$$

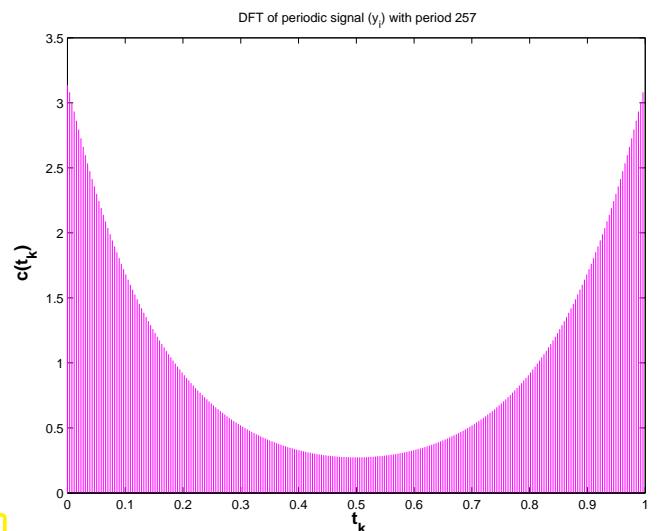
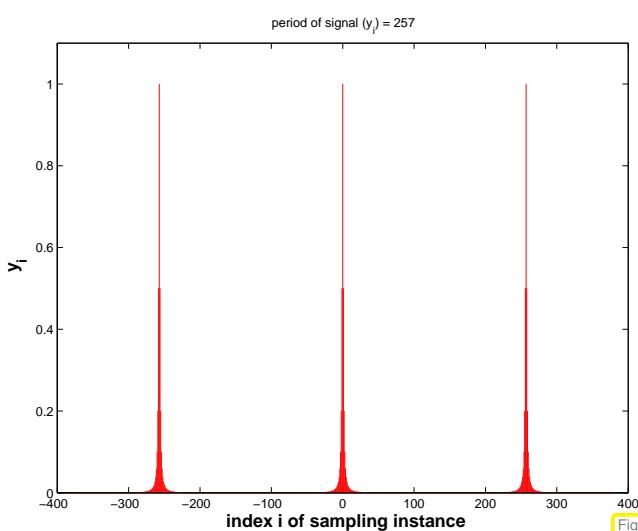
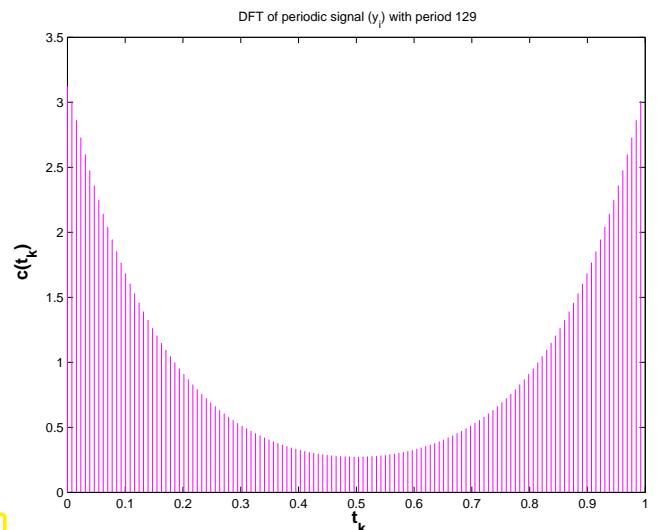
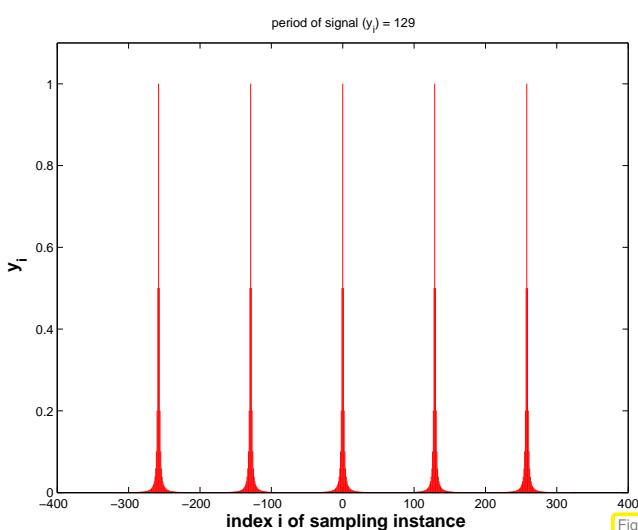
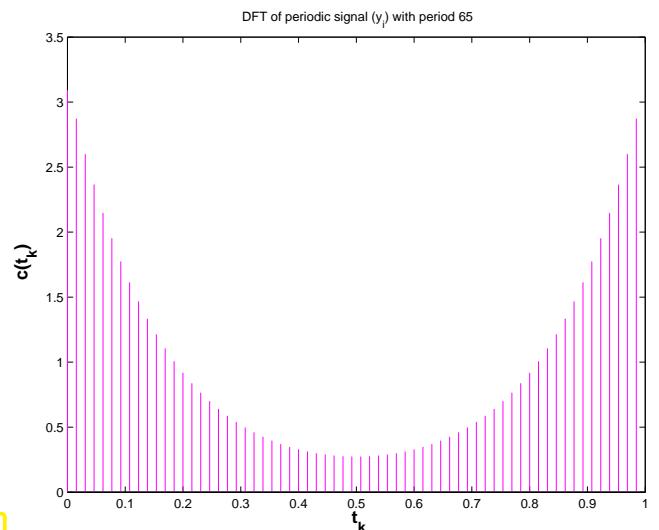
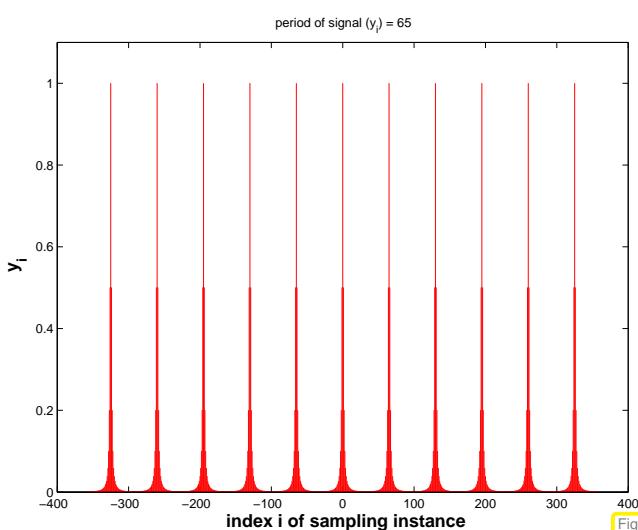
The notation indicates that we read  $c_k$  as the value of a *function*  $c : [0, 1] \mapsto \mathbb{C}$  for argument  $t_k$ .

### Example 9.2.33 (“Squeezed” DFT of a periodically truncated signal)

Bi-infinite discrete signal, “concentrated around 0”:  $y_j = \frac{1}{1+j^2}$ ,  $j \in \mathbb{Z}$ .

We examine the DFT of the  $2m+1$ -periodic signal obtained by periodic extension of  $(y_k)_{k=-m}^m$ .





Observation:  $c(t_k)$  “converge” to a function  $c : [0, 1] \rightarrow \mathbb{R}$

#### Listing 9.13: Plotting a periodically truncated signal and its DFT

```

1 % Visualize limit m → ∞ for a 2m+!-periodic signal and
2 % its discrete Fourier transform "squeezed" into [0,1].
3
4 % range of plot for visualization of discrete signal

```

```

5 Npow = 8; N = 3*(2^Npow+1);
6 % function defining discrete signal
7 yfn = @(k) 1./(1+k.*k);
8 % loop over different periods 2^l + 1
9 for mpow = [4 5 6 7]
10 m = 2^mpow;
11 ybas = yfn([(-m:-1), 0, (1:m)]);
12 Ncp = floor(N/(2*m+1));
13 y = repmat(ybas, 1, Ncp);
14
15 figure; hy = stem((1:length(y))-(length(y)+1)/2, y, 'r');
16 ax = axis; axis([ax(1) ax(2) 0 1.1]); hold on;
17 set(hy,'markersize',0);
18 xlabel('{\bf index i of sampling instance}', 'fontsize', 14);
19 ylabel('{\bf y_{i}}', 'fontsize', 14);
20 title(sprintf('period of signal (y_{i}) = %d', 2*m+1));
21 print('-depsc2', sprintf('../PICTURES/persig%d.eps', mpow));
22
23 % DFT of wrapped signal (one period)
24 c = fft([ybas(m+1:end), ybas(1:m)]);
25 hc = 1/(2*m+1);
26 figure; hc = stem(0:hc:1-hc, c, 'm');
27 set(hc,'markersize',0);
28 xlabel('{\bf t_{k}}', 'fontsize', 14);
29 ylabel('{\bf c(t_{k})}', 'fontsize', 14);
30 title(sprintf('DFT of periodic signal (y_{i}) with period
    %d', 2*m+1));
31 print('-depsc2', sprintf('../PICTURES/persigdft%d.eps', mpow));
32 end

```

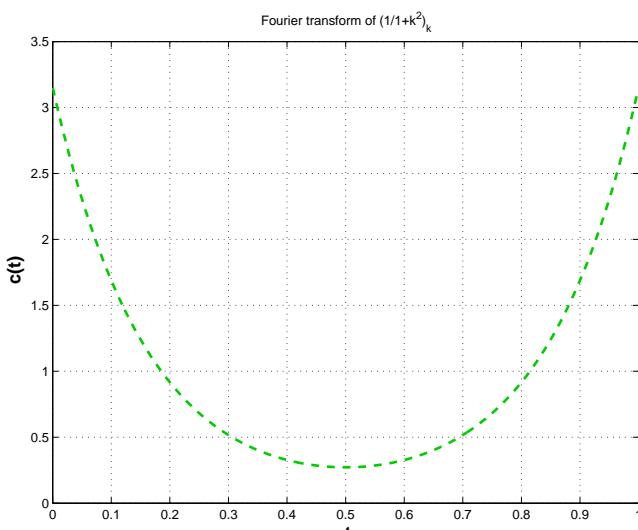
Now: pass to the limit  $m \rightarrow \infty$  (and keep the function perspective)

Note: passing to the limit amounts to dropping the assumption of periodicity!

$$c(t) = \sum_{k \in \mathbb{Z}} y_k \exp(-2\pi i k t) . \quad (9.2.34)$$

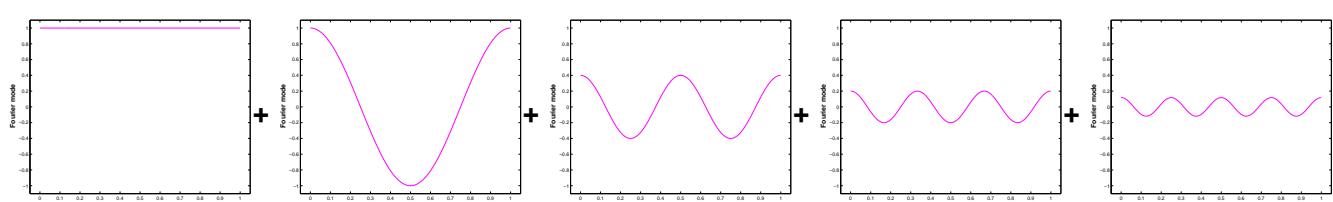
Terminology: the series (= infinite sum) on the right hand side of (9.2.34) is called a **Fourier series**

the function  $c : [0, 1] \mapsto \mathbb{C}$  defined by (9.2.34) is called the **Fourier transform** of the sequence  $(y_k)_{k \in \mathbb{Z}}$  (, if the series converges).



▷ Fourier transform of  $y_k := \frac{1}{1+k^2}$

Fourier transform  
= sum of Fourier modes



### Remark 9.2.35 (Decay conditions for bi-infinite signals)

The considerations above were based on

- \* truncation of  $(y_k)_{k \in \mathbb{Z}}$  to  $(y_k)_{k=-m}^m$  and
- \* periodic continuation to an  $2m+1$ -periodic signal.

Obviously, only if the signal is *concentrated around  $k = 0$*  this procedure will not lose essential information contained in the signal.

$$\text{Minimal requirement: } \lim_{k \rightarrow \infty} |y_k| = 0 , \quad (9.2.36)$$

$$\text{Stronger requirement: } \sum_{k \in \mathbb{Z}} |y_k| < \infty . \quad (9.2.37)$$

$$(9.2.37) \Rightarrow \begin{aligned} &\text{Fourier series (9.2.34) (link) converges uniformly [77, Def. 4.8.1]} \\ &\Rightarrow c : [0, 1] \mapsto \mathbb{C} \text{ is continuous [77, Thm. 4.8.1].} \end{aligned}$$

### Remark 9.2.38 (Numerical summation of Fourier series)

Assuming sufficiently fast decay of the signal  $(y_k)_{k \in \mathbb{Z}}$  for  $k \rightarrow \infty$  ( $\rightarrow$  Rem. 9.2.35), we can *approximate* the Fourier series (9.2.34) by a **Fourier sum**

$$c(t) \approx c_M(t) := \sum_{k=-M}^M y_k \exp(-2\pi i k t) , \quad M \gg 1 . \quad (9.2.39)$$

Task: Evaluation of  $c(t)$  at  $N$  equidistant points  $t_j := \frac{j}{N}$ ,  $j = 0, \dots, N$  (e.g., for plotting it).

$$c(t_j) = \lim_{M \rightarrow \infty} \sum_{k=-M}^M y_k \exp(-2\pi i k t_j) \approx \sum_{k=-M}^M y_k \exp(-2\pi i \frac{kj}{N}) , \quad (9.2.40)$$

for  $j = 0, \dots, N - 1$ .

Note: If  $N = M$  ➤ (9.2.40) is a **discrete Fourier transform** (→ Def. 9.2.13).

Listing 9.14: FFT-based evaluation of Fourier sum at equidistant points

```

1 function c = foursum(signal,M,N)
2 % Approximate evaluation of Fourier series, signal is a handle to a
3 % function of
4 % type @(k) providing the  $y_k$ , M specifies truncation
5 % of series according to (9.2.39), N is the number of equidistant
6 % evaluation points for c in [0,1[.
7
8 y = signal(-M:M); % Sample signal from -M to M
9 m = 2*M+1; % Length of signal
10 % Ensure that there are more sampling points than terms in series
11 if (m > N), l = ceil(m/N); N = l*N; else l = 1; end
12
13 % Zero padding and wrapping of signal, see Code 9.6
14 y_ext = zeros(1,N); y_ext(1:M+1) = y(M+1:end); y_ext(N-M+1:N) =
15 y(1:M);
16
17 % Perform DFT and decimate output vector
18 c = fft(y_ext); c = c(1:l:end);

```

Listing 9.15: Inefficient direct evaluation of Fourier sum at equidistant points

```

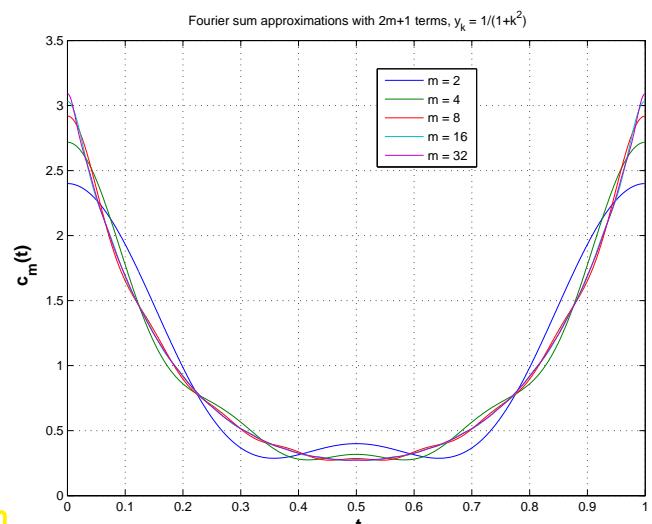
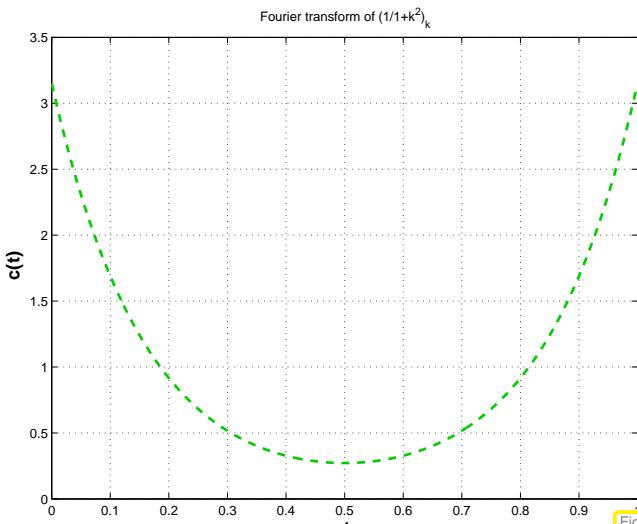
1 function c = foursumnaive(signal,M,N)
2 % Approximate direct evaluation of Fourier sum according to the defining
3 % formula (9.2.39), signal is a handle to a function of
4 % type @(k) providing the  $y_k$ , M specifies truncation
5 % of series according to (9.2.39), N is the number of equidistant
6 % evaluation points for c in [0,1[.
7
8 t = 0:1/N:1-1/N; % Evaluation points for Fourier sum c
9 c = signal(0)*ones(1,N);
10 omega = exp(-2*pi*i*t);
11 omp = omega; omm = 1./omega;
12
13 % Inefficient direct summation of Fourier series
14 for k=1:M
15   c = c+signal(k)*omp;
16   c = c+signal(-k)*omm;
17   omp = omp.*omega;
18   omm = omm./omega;
19 end

```

### Example 9.2.41 (Convergence of Fourier sums)

Infinite signal, satisfying decay condition (9.2.37):  $y_k = \frac{1}{1+k^2}$ , see Ex. 9.2.33.

Monitored: approximation of Fourier transform  $c(t)$  by Fourier sums  $c_m(t)$ , see (9.2.39).



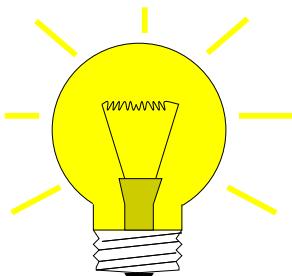
Observation: Convergence of Fourier sums in “eyeball norm”; quantitative statements about convergence can be deduced from ??.

Same limit considerations as above for the inverse DFT (9.2.15)

$$y_j = \frac{1}{n} \sum_{k=0}^{n-1} c_k \exp\left(2\pi i \frac{jk}{n}\right), \quad j = -m, \dots, m. \quad (9.2.42)$$

Adopt function perspective as before:  $c_k \leftrightarrow c(t_k)$ , cf. (9.2.31).

$$y_j = \frac{1}{n} \sum_{k=0}^{n-1} c(t_k) \exp(2\pi i j t_k), \quad j = -m, \dots, m. \quad (9.2.43)$$



Now: pass to the limit  $m \rightarrow \infty$  in (9.2.43)

Idea: right hand side of (9.2.43) = **Riemann sum**, cf. [77, Sect. 6.2]



in the limit  $m \rightarrow \infty$  the sum becomes an **integral**!

$$y_j = \int_0^1 c(t) \exp(2\pi i j t) dt. \quad (9.2.44)$$

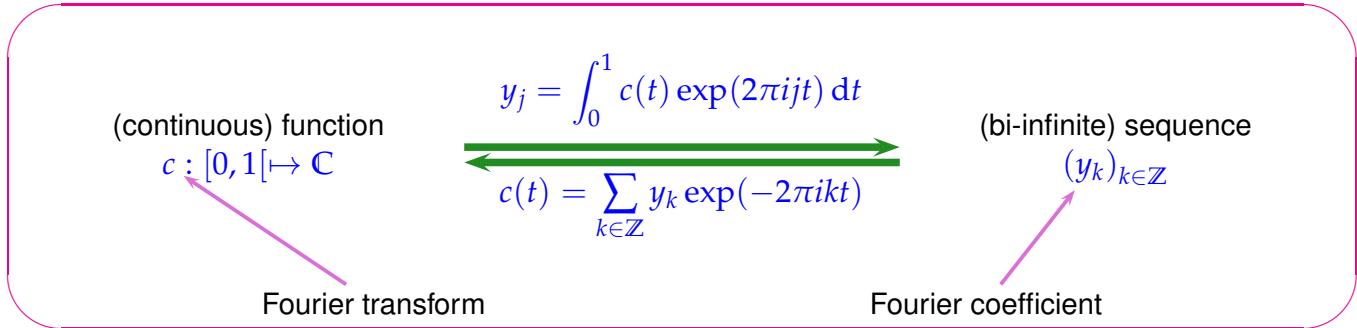
The formula (9.2.44) allows to recover the signal  $(y_k)_{k \in \mathbb{Z}}$  from its Fourier transform  $c(t)$ .

Terminology:  $y_j$  from (9.2.44) is called the **Fourier coefficient** of the function  $c$ .

Note: Both, the mapping  $(y_k)_{k \in \mathbb{Z}} \mapsto c$  from (9.2.34), and the mapping  $c \mapsto (y_k)_{k \in \mathbb{Z}}$  from (9.2.44) are **linear**!

(Recall the concept of a linear mapping as addressed in § 3.1.11)

Summary:



### Remark 9.2.45 (Filtering in Fourier domain)

Consider (bi-)infinite signal  $(x_k)_{k \in \mathbb{Z}}$  sent through a finite linear time-invariant causal channel with impulse response  $(h_0, \dots, h_{n-1})^\top$  ( $\rightarrow$  Ex. 9.1.1).

➤ output signal, see (9.1.3)

$$y_k = \sum_{j=0}^{n-1} h_j x_{k-j}, \quad k \in \mathbb{Z}. \quad (9.2.46)$$

Fourier transforms of signals:

$$(y_k)_{k \in \mathbb{Z}} \leftrightarrow c(t), \quad (x_j)_{j \in \mathbb{Z}} \leftrightarrow b(t)$$

(Assume that  $(x_k)_{k \in \mathbb{Z}}$  satisfies (9.2.37))

$$\begin{aligned} c(t) &= \sum_{k \in \mathbb{Z}} y_k \exp(-2\pi i kt) = \sum_{k \in \mathbb{Z}} \sum_{j=0}^{n-1} h_j x_{k-j} \exp(-2\pi i kt) \\ [\text{shift summation index } k] &= \sum_{j=0}^{n-1} \sum_{k \in \mathbb{Z}} h_j x_k \exp(-2\pi jt) \exp(-2\pi i kt) \\ &= \underbrace{\left( \sum_{j=0}^{n-1} h_j \exp(-2\pi jt) \right)}_{\text{trigonometric polynomial of degree } n-1} b(t). \end{aligned} \quad (9.2.47)$$

➤ Discrete convolution of a signal with finite impulse response  $\leftrightarrow$  multiplication of Fourier transform with trigonometric polynomial.

Conservation of power through Fourier transform:

Lemma 9.2.9 ➤ for Fourier matrix  $\mathbf{F}_n$ , see (9.2.8),  $\frac{1}{\sqrt{n}} \mathbf{F}_n$  is **unitary** ( $\rightarrow$  Def. 4.2.2)

??

$$\left\| \frac{1}{\sqrt{n}} \mathbf{F}_n \mathbf{y} \right\|_2 = \|\mathbf{y}\|_2. \quad (9.2.48)$$

Since DFT boils down to multiplication with  $\mathbf{F}_n$  ( $\rightarrow$  Def. 9.2.13), we conclude from (9.2.48)

$$\text{c}_k \text{ from (9.2.30)} \Rightarrow \frac{1}{n} \sum_{k=0}^{n-1} |c_k|^2 = \sum_{j=-m}^m |y_j|^2. \quad (9.2.49)$$

Now: function perspective  $c_k \leftrightarrow c(t_k)$   
 + passing to the limit  $m \rightarrow \infty$   
 + Riemann summation (see above)

$$(9.2.49) \xrightarrow{m \rightarrow \infty} \int_0^1 |c(t)|^2 dt = \sum_{j \in \mathbb{Z}} |y_j|^2. \quad (9.2.50)$$

### Theorem 9.2.51. Isometry property of Fourier transform

If  $\sum_{k \in \mathbb{Z}} |y_j|^2 < \infty$ , then

$$c(t) = \sum_{k \in \mathbb{Z}} y_k \exp(-2\pi i k t) \Rightarrow \int_0^1 |c(t)|^2 dt = \sum_{k \in \mathbb{Z}} |y_j|^2.$$

Recalling the concept of the  $L^2$ -norm of a function, see (3.2.63), the theorem can be stated as follows:

Thm. 9.2.51  $\leftrightarrow$  The  $L^2$ -norm of a Fourier transform agrees with the Euclidean norm of the corresponding signal.

Note: Euclidean norm of a sequence  $\|(y_k)_{k \in \mathbb{Z}}\|_2^2 := \sum_{k \in \mathbb{Z}} |y_j|^2$ .

## 9.3 Fast Fourier Transform (FFT)



*Supplementary reading.* [15, Sect. 8.7.3], [42, Sect. 53], [63, Sect. 10.9.2]

At first glance (at (9.2.14)): DFT in  $\mathbb{C}^n$  seems to require asymptotic computational effort of  $O(n^2)$  (matrix  $\times$  vector multiplication with dense matrix).

### Example 9.3.1 (Efficiency of fft)

`tic-toc`-timing in MATLAB: compare `fft`, loop based implementation, and direct matrix multiplication (MATLAB V6.5, Linux, Mobile Intel Pentium 4 - M CPU 2.40GHz, minimum over 5 runs)

Listing 9.16: timing of different implementations of DFT

```

1 res = [];
2 for n=1:1:3000, y = rand(n,1); c = zeros(n,1);
3 t1 = realmax; for k=1:5, tic;
```

```

4     omega = exp(-2*pi*i/n); c(1) = sum(y); s = omega;
5     for j=2:n, c(j) = y(n);
6         for k=n-1:-1:1, c(j) = c(j)*s+y(k); end
7         s = s*omega;
8     end
9     t1 = min(t1,toe);
10    end
11
12    [I,J] = meshgrid(0:n-1,0:n-1); F = exp(-2*pi*i.*I.*J/n);
13    t2 = realmax; for k=1:5, tic; c = F*y; t2 = min(t2,toe); end
14    t3 = realmax; for k=1:5, tic; d = fft(y); t3 = min(t3,toe); end
15    res = [res; n t1 t2 t3];
16
17 figure('name','FFT timing');
18 semilogy(res(:,1),res(:,2),'b-',res(:,1),res(:,3),'k-',
19           res(:,1),res(:,4),'r-');
20 ylabel('{\bf run time [s]}','Fontsize',14);
21 xlabel('{\bf vector length n}','Fontsize',14);
22 legend('loop based computation','direct matrix
multiplication','MATLAB fft() function',1);
print -deps2c '../PICTURES/ffftime.eps'

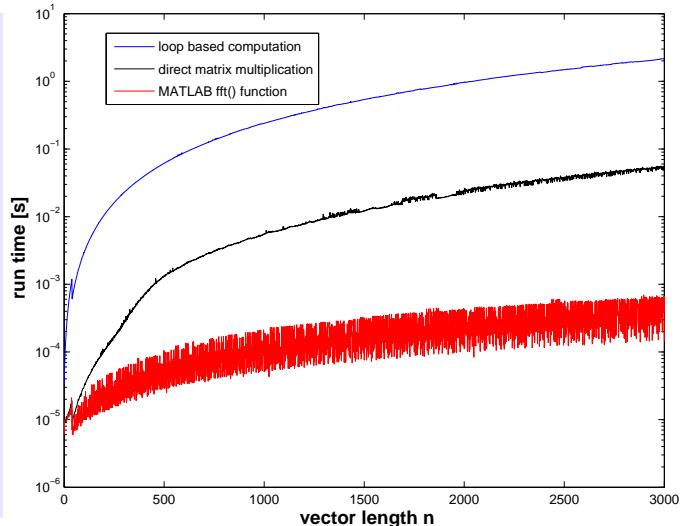
```

**MATLAB-code 9.3.2:** MATLAB-CODE naive DFT-implementation

```

1 c = zeros(n,1);
2 omega = exp(-2*pi*i/n);
3 c(1) = sum(y); s = omega;
4 for j=2:n
5     c(j) = y(n);
6     for k=n-1:-1:1
7         c(j) = c(j)*s+y(k);
8     end
9     s = s*omega;
10 end

```



Incredible! The MATLAB `fft()`-function clearly beats the  $O(n^2)$  asymptotic complexity of the other implementations. Note the logarithmic scale!

The secret of MATLAB's `fft()`:

the [Fast Fourier Transform algorithm \[21\]](#)

(discovered by C.F. Gauss in 1805, rediscovered by Cooley & Tuckey in 1965,  
one of the “[top ten algorithms of the century](#)”).

An elementary manipulation of (9.2.14) for  $n = 2m$ ,  $m \in \mathbb{N}$ :

$$\begin{aligned}
 c_k &= \sum_{j=0}^{n-1} y_j e^{-\frac{2\pi i}{n} jk} \\
 &= \sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{n} 2jk} + \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{n} (2j+1)k} \\
 &= \underbrace{\sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{m} jk}}_{=: \tilde{c}_k^{\text{even}}} + e^{-\frac{2\pi i}{n} k} \cdot \underbrace{\sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{m} jk}}_{=: \tilde{c}_k^{\text{odd}}}.
 \end{aligned} \tag{9.3.3}$$

Note  $m$ -periodicity:  $\tilde{c}_k^{\text{even}} = \tilde{c}_{k+m}^{\text{even}}$ ,  $\tilde{c}_k^{\text{odd}} = \tilde{c}_{k+m}^{\text{odd}}$ .

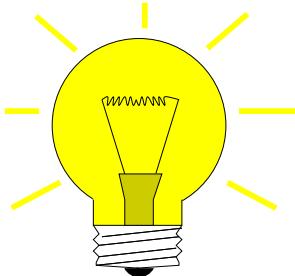
Note:  $\tilde{c}_k^{\text{even}}, \tilde{c}_k^{\text{odd}}$  from DFTs of length  $m$ !

with  $\mathbf{y}_{\text{even}} := (y_0, y_2, \dots, y_{n-2})^\top \in \mathbb{C}^m$ :  $(\tilde{c}_k^{\text{even}})_{k=0}^{m-1} = \mathbf{F}_m \mathbf{y}_{\text{even}}$ ,

with  $\mathbf{y}_{\text{odd}} := (y_1, y_3, \dots, y_{n-1})^\top \in \mathbb{C}^m$ :  $(\tilde{c}_k^{\text{odd}})_{k=0}^{m-1} = \mathbf{F}_m \mathbf{y}_{\text{odd}}$ .

(9.3.3):

DFT of length  $2m = 2 \times$  DFT of length  $m$  +  $2m$  additions & multiplications



Idea:

divide & conquer recursion

(for DFT of length  $n = 2^L$ )

Listing 9.17: Recursive FFT

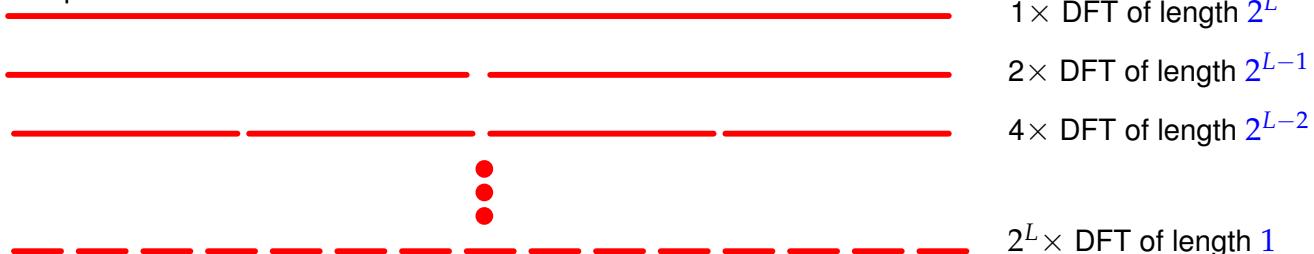
```

1 function c = fftrec(y)
2 n = length(y);
3 if (n == 1), c = y; return;
4 else
5   c1 = fftrec(y(1:2:n));
6   c2 = fftrec(y(2:2:n));
7   c = [c1;c1] +
8     (exp(-2*pi*i/n).^(0:n-1)').*
9     [c2;c2];
10 end

```

FFT-algorithm

Computational cost of `fftrec`:



Listing 9.17: each level of the recursion requires  $O(2^L)$  elementary operations.

Asymptotic complexity of FFT algorithm,  $n = 2^L$ :  $O(L2^L) = O(n \log_2 n)$

(MATLAB `fft`-function: cost  $\approx 5n \log_2 n$ ).

>

**Remark 9.3.4 (FFT algorithm by matrix factorization)**

For  $n = 2m$ ,  $m \in \mathbb{N}$ ,

$$\text{permutation } P_m^{\text{OE}}(1, \dots, n) = (1, 3, \dots, n-1, 2, 4, \dots, n).$$



As  $\omega_n^{2j} = \omega_m^j$ :

$$\text{permutation of rows } P_m^{\text{OE}} \mathbf{F}_n = \left[ \begin{array}{c|c|c|c} & \mathbf{F}_m & & \mathbf{F}_m \\ \hline \mathbf{F}_m & \begin{bmatrix} \omega_n^0 & \omega_n^1 & \ddots & \omega_n^{n/2-1} \end{bmatrix} & \mathbf{F}_m & \begin{bmatrix} \omega_n^{n/2} & \omega_n^{n/2+1} & \ddots & \omega_n^{n-1} \end{bmatrix} \\ \hline & \mathbf{I} & & \mathbf{I} \\ \hline \mathbf{F}_m & \begin{bmatrix} \omega_n^0 & \omega_n^1 & \ddots & \omega_n^{n/2-1} \end{bmatrix} & -\omega_n^0 & -\omega_n^1 \\ \hline & & \ddots & \ddots & -\omega_n^{n/2-1} \end{array} \right] =$$

Example: factorization of Fourier matrix for  $n = 10$

$$P_5^{\text{OE}} \mathbf{F}_{10} = \left[ \begin{array}{cc|cc} \omega^0 & \omega^0 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 \\ \omega^0 & \omega^4 & \omega^8 & \omega^2 & \omega^6 & \omega^0 & \omega^4 & \omega^8 & \omega^2 & \omega^6 \\ \omega^0 & \omega^6 & \omega^2 & \omega^8 & \omega^4 & \omega^0 & \omega^6 & \omega^2 & \omega^8 & \omega^4 \\ \omega^0 & \omega^8 & \omega^6 & \omega^4 & \omega^2 & \omega^0 & \omega^8 & \omega^6 & \omega^4 & \omega^2 \\ \hline \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 & \omega^8 & \omega^9 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 & \omega^2 & \omega^5 & \omega^8 & \omega^1 & \omega^4 & \omega^7 \\ \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 \\ \omega^0 & \omega^7 & \omega^4 & \omega^1 & \omega^8 & \omega^5 & \omega^2 & \omega^9 & \omega^6 & \omega^3 \\ \omega^0 & \omega^9 & \omega^8 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{array} \right], \quad \omega := \omega_{10}.$$

What if  $n \neq 2^L$ ? Quoted from MATLAB manual:

To compute an  $n$ -point DFT when  $n$  is composite (that is, when  $n = pq$ ), the FFTW library decomposes the problem using the Cooley-Tukey algorithm, which first computes  $p$  transforms of size  $q$ , and then computes  $q$  transforms of size  $p$ . The decomposition is applied recursively to both the  $p$ - and  $q$ -point DFTs until the problem can be solved using one of several machine-generated fixed-size "codelets." The codelets in turn use several algorithms in combination, including a variation of Cooley-Tukey, a prime factor algorithm, and a split-radix algorithm. The particular factorization of  $n$  is chosen heuristically.

The execution time for fft depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors → Ex. 9.3.1.

### Remark 9.3.5 (FFT based on general factorization)

Fast Fourier transform algorithm for DFT of length  $n = pq$ ,  $p, q \in \mathbb{N}$  (Cooley-Tuckey-Algorithm)

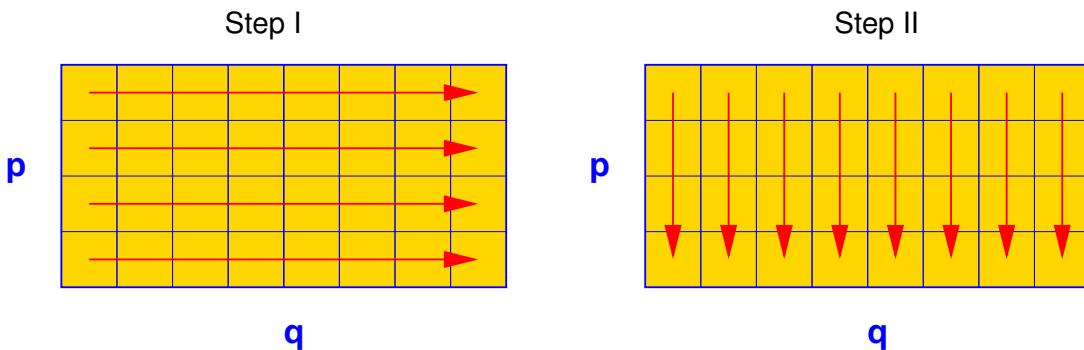
$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{jk} \stackrel{j := lp+m}{=} \sum_{m=0}^{p-1} \sum_{l=0}^{q-1} y_{lp+m} e^{-\frac{2\pi i}{pq}(lp+m)k} = \sum_{m=0}^{p-1} \omega_n^{mk} \sum_{l=0}^{q-1} y_{lp+m} \omega_q^{l(k \bmod q)}. \quad (9.3.6)$$

Step I: perform  $p$  DFTs of length  $q$        $z_{m,k} := \sum_{l=0}^{q-1} y_{lp+m} \omega_q^{lk}, \quad 0 \leq m < p, 0 \leq k < q$ .

Step II: for  $k =: rq + s, \quad 0 \leq r < p, 0 \leq s < q$

$$c_{rq+s} = \sum_{m=0}^{p-1} e^{-\frac{2\pi i}{pq}(rq+s)m} z_{m,s} = \sum_{m=0}^{p-1} (\omega_n^{ms} z_{m,s}) \omega_p^{mr}$$

and hence  $q$  DFTs of length  $p$  give all  $c_k$ .



### Remark 9.3.7 (FFT for prime $n$ )

When  $n \neq 2^L$ , even the Cooley-Tukey algorithm of Rem. 9.3.5 will eventually lead to a DFT for a vector with prime length.

Quoted from the MATLAB manual:

When  $n$  is a prime number, the FFTW library first decomposes an  $n$ -point problem into three  $(n - 1)$ -point problems using Rader's algorithm [64]. It then uses the Cooley-Tukey decomposition described above to compute the  $(n - 1)$ -point DFTs.

Details of Rader's algorithm: a theorem from number theory:

$$\forall p \in \mathbb{N} \text{ prime } \exists g \in \{1, \dots, p-1\}: \{g^k \pmod{p} : k = 1, \dots, p-1\} = \{1, \dots, p-1\},$$

► permutation  $P_{p,g} : \{1, \dots, p-1\} \mapsto \{1, \dots, p-1\}$ ,  $P_{p,g}(k) = g^k \pmod{p}$ ,  
 reversing permutation  $P_k : \{1, \dots, p-1\} \mapsto \{1, \dots, p-1\}$ ,  $P_k(i) = k - i + 1$ .

For Fourier matrix  $\mathbf{F} = (f_{ij})_{i,j=1}^p$ :  $P_{p-1} P_{p,g} (f_{ij})_{i,j=2}^p P_{p,g}^\top$  is circulant.

Example for  $p = 13$ :

$g = 2$ , permutation:  $(2 \ 4 \ 8 \ 3 \ 6 \ 12 \ 11 \ 9 \ 5 \ 10 \ 7 \ 1)$ .

$\omega^0$	$\omega^0$	$\omega^0$	$\omega^0$	$\omega^0$	$\omega^0$	$\omega^0$	$\omega^0$	$\omega^0$	$\omega^0$	$\omega^0$	$\omega^0$	$\omega^0$
$\omega^0$	$\omega^2$	$\omega^4$	$\omega^8$	$\omega^3$	$\omega^6$	$\omega^{12}$	$\omega^{11}$	$\omega^9$	$\omega^5$	$\omega^{10}$	$\omega^7$	$\omega^1$
$\omega^0$	$\omega^1$	$\omega^2$	$\omega^4$	$\omega^8$	$\omega^3$	$\omega^6$	$\omega^{12}$	$\omega^{11}$	$\omega^9$	$\omega^5$	$\omega^{10}$	$\omega^7$
$\omega^0$	$\omega^7$	$\omega^1$	$\omega^2$	$\omega^4$	$\omega^8$	$\omega^3$	$\omega^6$	$\omega^{12}$	$\omega^{11}$	$\omega^9$	$\omega^5$	$\omega^{10}$
$\omega^0$	$\omega^{10}$	$\omega^7$	$\omega^1$	$\omega^2$	$\omega^4$	$\omega^8$	$\omega^3$	$\omega^6$	$\omega^{12}$	$\omega^{11}$	$\omega^9$	$\omega^5$
$\omega^0$	$\omega^5$	$\omega^{10}$	$\omega^7$	$\omega^1$	$\omega^2$	$\omega^4$	$\omega^8$	$\omega^3$	$\omega^6$	$\omega^{12}$	$\omega^{11}$	$\omega^9$
$\mathbf{F}_{13}$	→	$\omega^0$	$\omega^9$	$\omega^5$	$\omega^{10}$	$\omega^7$	$\omega^1$	$\omega^2$	$\omega^4$	$\omega^8$	$\omega^3$	$\omega^6$
		$\omega^0$	$\omega^{11}$	$\omega^9$	$\omega^5$	$\omega^{10}$	$\omega^7$	$\omega^1$	$\omega^2$	$\omega^4$	$\omega^8$	$\omega^3$
		$\omega^0$	$\omega^{12}$	$\omega^{11}$	$\omega^9$	$\omega^5$	$\omega^{10}$	$\omega^7$	$\omega^1$	$\omega^2$	$\omega^4$	$\omega^8$
		$\omega^0$	$\omega^6$	$\omega^{12}$	$\omega^{11}$	$\omega^9$	$\omega^5$	$\omega^{10}$	$\omega^7$	$\omega^1$	$\omega^2$	$\omega^4$
		$\omega^0$	$\omega^3$	$\omega^6$	$\omega^{12}$	$\omega^{11}$	$\omega^9$	$\omega^5$	$\omega^{10}$	$\omega^7$	$\omega^1$	$\omega^4$
		$\omega^0$	$\omega^8$	$\omega^3$	$\omega^6$	$\omega^{12}$	$\omega^{11}$	$\omega^9$	$\omega^5$	$\omega^{10}$	$\omega^7$	$\omega^1$
		$\omega^0$	$\omega^4$	$\omega^8$	$\omega^3$	$\omega^6$	$\omega^{12}$	$\omega^{11}$	$\omega^9$	$\omega^5$	$\omega^7$	$\omega^2$

Then apply fast (FFT based!) algorithms for multiplication with circulant matrices to right lower  $(n-1) \times (n-1)$  block of permuted Fourier matrix .

Asymptotic complexity of  $\text{c}=\text{fft}(\mathbf{y})$  for  $\mathbf{y} \in \mathbb{C}^n = O(n \log n)$ .

← Section 9.2.1

Asymptotic complexity of discrete periodic convolution/multiplication with circulant matrix, see Listing 9.3:

$$\text{Cost}(\mathbf{z} = \text{pconvfft}(\mathbf{u}, \mathbf{x}), \mathbf{u}, \mathbf{x} \in \mathbb{C}^n) = O(n \log n).$$

Asymptotic complexity of discrete convolution, see Listing 9.4:

$$\text{Cost}(\mathbf{z} = \text{myconv}(\mathbf{h}, \mathbf{x}), \mathbf{h}, \mathbf{x} \in \mathbb{C}^n) = O(n \log n).$$

## 9.4 Trigonometric transformations



Supplementary reading. [42, Sect. 55]

Keeping in mind  $\exp(2\pi i x) = \cos(2\pi x) + i \sin(2\pi x)$  we may also consider the real/imaginary parts of the Fourier basis vectors  $(\mathbf{F}_n)_{:,j}$  as bases of  $\mathbb{R}^n$  and define the corresponding basis transformation. They can all be realized by means of `fft` with an asymptotic computational effort of  $O(n \log n)$ .

Details are given in the sequel.

### 9.4.1 Sine transform

Another trigonometric basis transform in  $\mathbb{R}^{n-1}$ ,  $n \in \mathbb{N}$ :

$$\text{Standard basis of } \mathbb{R}^{n-1} \quad \left\{ \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \dots, \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix} \right\} \quad \longleftarrow \quad \text{“Sine basis”} \quad \left\{ \begin{bmatrix} \sin(\frac{\pi}{n}) \\ \sin(\frac{2\pi}{n}) \\ \vdots \\ \sin(\frac{(n-1)\pi}{n}) \end{bmatrix}, \begin{bmatrix} \sin(\frac{2\pi}{n}) \\ \sin(\frac{4\pi}{n}) \\ \vdots \\ \sin(\frac{2(n-1)\pi}{n}) \end{bmatrix}, \dots, \begin{bmatrix} \sin(\frac{(n-1)\pi}{n}) \\ \sin(\frac{2(n-1)\pi}{n}) \\ \vdots \\ \sin(\frac{(n-1)^2\pi}{n}) \end{bmatrix} \right\}$$

Basis transform matrix (sine basis  $\rightarrow$  standard basis):  $\mathbf{S}_n := (\sin(jk\pi/n))_{j,k=1}^{n-1} \in \mathbb{R}^{n-1, n-1}$ .

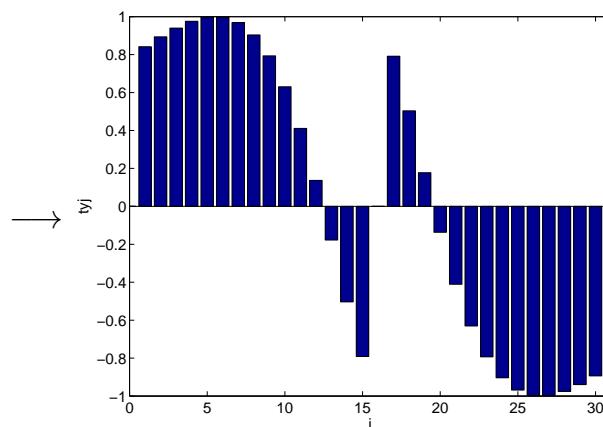
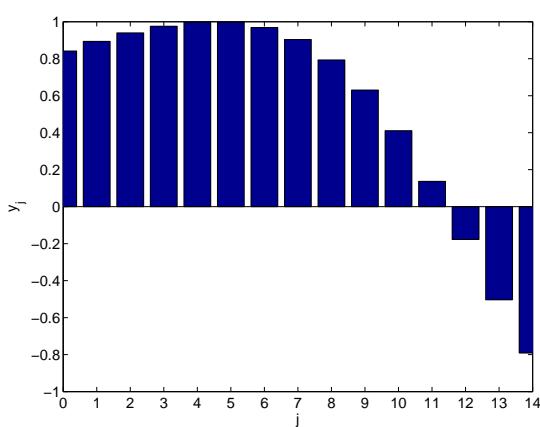
#### Lemma 9.4.1. Properties of the sine matrix

$\sqrt{2/n} \mathbf{S}_n$  is real, symmetric and orthogonal ( $\rightarrow$  Def. 4.2.2)

Sine transform:  $s_k = \sum_{j=1}^{n-1} y_j \sin(\pi j k / n)$  ,  $k = 1, \dots, n-1$ . (9.4.2)

DFT-based algorithm for the sine transform ( $\hat{=} \mathbf{S}_n \times$  vector):

“wrap around”:  $\tilde{\mathbf{y}} \in \mathbb{R}^{2n}$ :  $\tilde{y}_j = \begin{cases} y_j & , \text{ if } j = 1, \dots, n-1 , \\ 0 & , \text{ if } j = 0, n , \\ -y_{2n-j} & , \text{ if } j = n+1, \dots, 2n-1 . \end{cases}$  ( $\tilde{\mathbf{y}}$  “odd”)



$$\begin{aligned}
 (\mathbf{F}_{2n}\tilde{\mathbf{y}})_k &\stackrel{(9.2.14)}{=} \sum_{j=1}^{2n-1} \tilde{y}_j e^{-\frac{2\pi}{2n}kj} \\
 &= \sum_{j=1}^{n-1} y_j e^{-\frac{\pi}{n}kj} - \sum_{j=n+1}^{2n-1} y_{2n-j} e^{-\frac{\pi}{n}kj} \\
 &= \sum_{j=1}^{n-1} y_j (e^{-\frac{\pi}{n}kj} - e^{\frac{\pi}{n}kj}) \\
 &= -2i (\mathbf{S}_n \mathbf{y})_k , \quad k = 1, \dots, n-1 .
 \end{aligned}$$

**MATLAB-code 9.4.3:** [Wrap-around implementation] MATLAB-

CODE sine transform

```

1 function c = sinetrans(y)
2 n = length(y)+1;
3 yt = [0, y, 0, -y(end:-1:1)];
4 ct = fft(yt);
5 c = -ct(2:n) / (2*i);

```

**Remark 9.4.4 (Sine transform via DFT of half length)**

Step ①: transform of the coefficients

$$\tilde{y}_j = \sin(j\pi/n)(y_j + y_{n-j}) + \frac{1}{2}(y_j - y_{n-j}) , \quad j = 1, \dots, n-1 , \quad \tilde{y}_0 = 0 .$$

Step ②: real DFT ( $\rightarrow$  Section 9.2.3) of  $(\tilde{y}_0, \dots, \tilde{y}_{n-1}) \in \mathbb{R}^n$ :  $c_k := \sum_{j=0}^{n-1} \tilde{y}_j e^{-\frac{2\pi i}{n}jk}$

$$\begin{aligned}
 \text{Hence } \text{Re}\{c_k\} &= \sum_{j=0}^{n-1} \tilde{y}_j \cos(-\frac{2\pi i}{n}jk) = \sum_{j=1}^{n-1} (y_j + y_{n-j}) \sin(\frac{\pi j}{n}) \cos(\frac{2\pi i}{n}jk) \\
 &= \sum_{j=0}^{n-1} 2y_j \sin(\frac{\pi j}{n}) \cos(\frac{2\pi i}{n}jk) = \sum_{j=0}^{n-1} y_j \left( \sin(\frac{2k+1}{n}\pi j) - \sin(\frac{2k-1}{n}\pi j) \right) \\
 &= s_{2k+1} - s_{2k-1} . \\
 \text{Im}\{c_k\} &= \sum_{j=0}^{n-1} \tilde{y}_j \sin(-\frac{2\pi i}{n}jk) = - \sum_{j=1}^{n-1} \frac{1}{2}(y_j - y_{n-j}) \sin(\frac{2\pi i}{n}jk) = - \sum_{j=1}^{n-1} y_j \sin(\frac{2\pi i}{n}jk) \\
 &= -s_{2k} .
 \end{aligned}$$

Step ③: extraction of  $s_k$ 

$$\begin{aligned}
 s_{2k+1} , \quad k = 0, \dots, \frac{n}{2}-1 &\quad \blacktriangleright \text{ from recursion } s_{2k+1} - s_{2k-1} = \text{Re}\{c_k\} , \quad s_1 = \sum_{j=1}^{n-1} y_j \sin(\pi j/n) , \\
 s_{2k} , \quad k = 1, \dots, \frac{n}{2}-2 &\quad \blacktriangleright \quad s_{2k} = -\text{Im}\{c_k\} .
 \end{aligned}$$

MATLAB-Implementation (via a **fft** of length  $n/2$ ):**MATLAB-code 9.4.5:** MATLAB-CODE Sine transform

```

1 function s = sinetrans(y)
2 n = length(y)+1;
3 sinevals = imag(exp(i*pi/n).^(1:n-1));
4 yt = [0 (sinevals.* (y+y(end:-1:1)) + 0.5*(y-y(end:-1:1)))];
5 c = fftreal(yt);

```

```

6   s(1) = dot(sinevals,y);
7   for k=2:N-1
8     if (mod(k,2) == 0), s(k) = -imag(c(k/2+1));
9     else, s(k) = s(k-2) + real(c((k-1)/2+1)); end
10  end

```

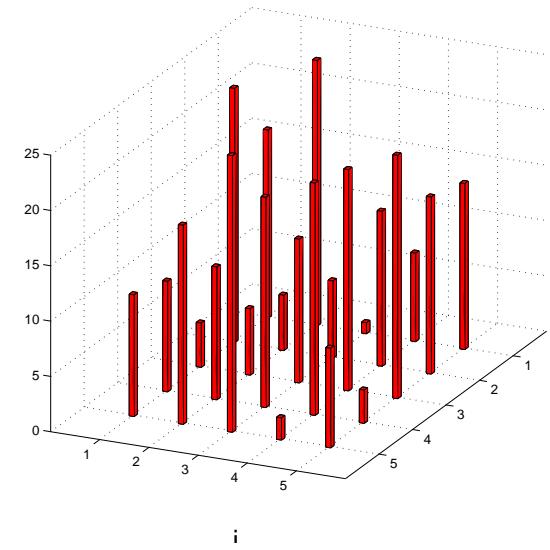
Application: diagonalization of local translation invariant linear operators.

5-points-stencil-operator on  $\mathbb{R}^{n,n}$ ,  $n \in \mathbb{N}$ , in grid representation:

$$T : \mathbb{R}^{n,n} \mapsto \mathbb{R}^{n,n}, \quad \mathbf{X} \mapsto T(\mathbf{X}) \\ (T(\mathbf{X}))_{ij} := c_x x_{ij} + c_y x_{i,j+1} + c_y x_{i,j-1} + c_x x_{i+1,j} + c_x x_{i-1,j}$$

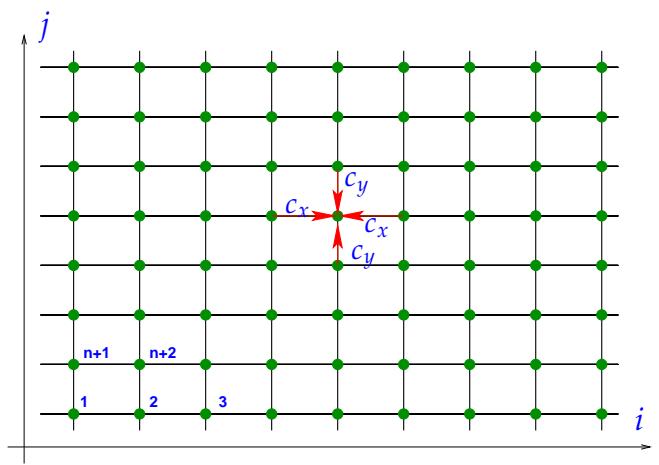
with  $c, c_y, c_x \in \mathbb{R}$ , convention:  $x_{ij} := 0$  for  $(i, j) \notin \{1, \dots, n\}^2$ .

$$\mathbf{X} \in \mathbb{R}^{n,n} \\ \uparrow \\ \text{grid function } \in \{1, \dots, n\}^2 \mapsto \mathbb{R}$$



Identification  $\mathbb{R}^{n,n} \cong \mathbb{R}^{n^2}$ ,  $x_{ij} \sim \tilde{x}_{(j-1)n+i}$  gives matrix representation  $\mathbf{T} \in \mathbb{R}^{n^2, n^2}$  of  $T$ :

$$\mathbf{T} = \begin{bmatrix} \mathbf{C} & c_y \mathbf{I} & 0 & \cdots & \cdots & 0 \\ c_y \mathbf{I} & \mathbf{C} & c_y \mathbf{I} & & & \vdots \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & c_y \mathbf{I} & \mathbf{C} & c_y \mathbf{I} & \\ 0 & \cdots & \cdots & 0 & c_y \mathbf{I} & \mathbf{C} \end{bmatrix} \in \mathbb{R}^{n^2, n^2}, \\ \mathbf{C} = \begin{bmatrix} c & c_x & 0 & \cdots & \cdots & 0 \\ c_x & c & c_x & & & \vdots \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & c_x & c & c_x & \\ 0 & \cdots & \cdots & 0 & c_x & c \end{bmatrix} \in \mathbb{R}^{n,n}.$$

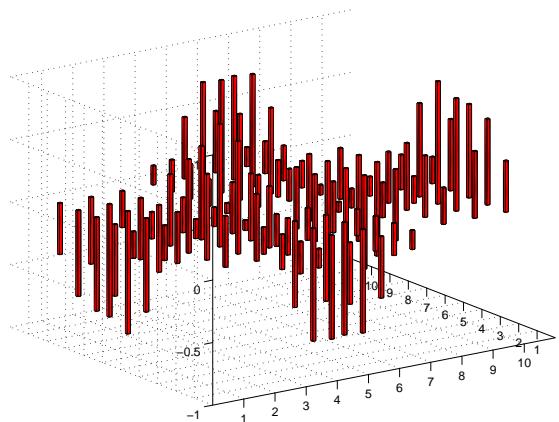


Sine basis of  $\mathbb{R}^{n,n}$ :

$$\mathbf{B}^{kl} = (\sin(\frac{\pi}{n+1}ki) \sin(\frac{\pi}{n+1}lj))_{i,j=1}^n. \quad (9.4.6)$$

$n = 10$ : grid function  $\mathbf{B}^{2,3}$

$\Rightarrow$



$$\begin{aligned} (T(\mathbf{B}^{kl}))_{ij} &= c \sin(\frac{\pi}{n}ki) \sin(\frac{\pi}{n}lj) + c_y \sin(\frac{\pi}{n}ki) (\sin(\frac{\pi}{n+1}l(j-1)) + \sin(\frac{\pi}{n+1}l(j+1))) + \\ &\quad c_x \sin(\frac{\pi}{n}lj) (\sin(\frac{\pi}{n+1}k(i-1)) + \sin(\frac{\pi}{n+1}k(i+1))) \\ &= \sin(\frac{\pi}{n}ki) \sin(\frac{\pi}{n}lj) (c + 2c_y \cos(\frac{\pi}{n+1}l) + 2c_x \cos(\frac{\pi}{n+1}k)) \end{aligned}$$

Hence  $\mathbf{B}^{kl}$  is eigenvector of  $T \leftrightarrow \mathbf{T}$  corresponding to eigenvalue  $c + 2c_y \cos(\frac{\pi}{n+1}l) + 2c_x \cos(\frac{\pi}{n+1}k)$ .

Algorithm for basis transform:

$$\mathbf{X} = \sum_{k=1}^n \sum_{l=1}^n y_{kl} \mathbf{B}^{kl} \Rightarrow x_{ij} = \sum_{k=1}^n \sin(\frac{\pi}{n+1}ki) \sum_{l=1}^n y_{kl} \sin(\frac{\pi}{n+1}lj).$$

Hence nested sine transforms ( $\rightarrow$  Section 9.2.4) for rows/columns of  $\mathbf{Y} = (y_{kl})_{k,l=1}^n$ .

Here: implementation of sine transform (9.4.2) with “wrapping”-technique.

**MATLAB-code 9.4.7:** MATLAB-CODE two dimensional sine tr.

```

1 function C = sinft2d(Y)
2 [m,n] = size(Y);
3 C = fft([zeros(1,n); Y; ...
4 zeros(1,n); ...
5 -Y(end:-1:1,:)]);
6 C = i*C(2:m+1,:)/2;
7 C = fft([zeros(1,m); C; ...
8 zeros(1,m); ...
9 -C(end:-1:1,:)]);
10 C= i*C(2:n+1,:)/2;
```

**MATLAB-code 9.4.8:** [ MATLAB-CODE FFT-based solution of local translation invariant linear operators

```

1 function X = fftsolve(B,c,cx,cy)
2 [m,n] = size(B);
3 [I,J] = meshgrid(1:m,1:n);
4 X = 4*sinft2d(sinft2d(B) ...
5 ./(c+2*cx*cos(pi/(n+1)*I)+...
6 2*cy*cos(pi/(m+1)*J)) ...
7 /((m+1)*(n+1));
```

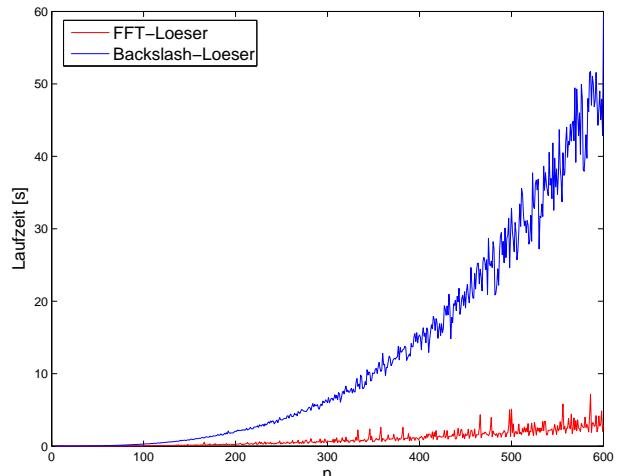
Diagonalization of  $\mathbf{T}$   
via 2D sine transform

efficient algorithm  
for solving linear system of equations  $\mathbf{T}(\mathbf{X}) = \mathbf{B}$   
computational cost  $\mathcal{O}(n^2 \log n)$ .

### Example 9.4.9 (Efficiency of FFT-based LSE-solver)

`tic-toe-timing (MATLAB V7, Linux, Intel Pentium 4 Mobile CPU 1.80GHz)`

```
A = gallery('poisson', n);
B = magic(n);
b = reshape(B, n*n, 1);
tic;
C = fftsolve(B, 4, -1, -1);
t1 = toc;
tic; x = A\b; t2 = toc;
```



## 9.4.2 Cosine transform

Another trigonometric basis transform in  $\mathbb{R}^n$ ,  $n \in \mathbb{N}$ :

$$\text{standard basis of } \mathbb{R}^n \quad \left\{ \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \dots, \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \right\} \leftarrow \text{"cosine basis"} \quad \left\{ \begin{bmatrix} 2^{-1/2} \\ \cos(\frac{\pi}{2n}) \\ \cos(\frac{2\pi}{2n}) \\ \vdots \\ \cos(\frac{(n-1)\pi}{2n}) \end{bmatrix}, \begin{bmatrix} 2^{-1/2} \\ \cos(\frac{3\pi}{2n}) \\ \cos(\frac{6\pi}{2n}) \\ \vdots \\ \cos(\frac{3(n-1)\pi}{2n}) \end{bmatrix}, \dots, \begin{bmatrix} 2^{-1/2} \\ \cos(\frac{(2n-1)\pi}{2n}) \\ \cos(\frac{2(2n-1)\pi}{2n}) \\ \vdots \\ \cos(\frac{(n-1)(2n-1)\pi}{2n}) \end{bmatrix} \right\}.$$

Basis transform matrix (cosine basis  $\rightarrow$  standard basis):

$$\mathbf{C}_n = (c_{ij}) \in \mathbb{R}^{n,n} \quad \text{with} \quad c_{ij} = \begin{cases} 2^{-1/2} & , \text{if } i = 1, \\ \cos((i-1)\frac{2j-1}{2n}\pi) & , \text{if } i > 1. \end{cases}$$

### Lemma 9.4.10. Properties of cosine matrix

$\sqrt{2/n} \mathbf{C}_n$  is real and orthogonal ( $\rightarrow$  Def. 4.2.2).

Note:  $\mathbf{C}_n$  is not symmetric

cosine transform:

$$c_k = \sum_{j=0}^{n-1} y_j \cos(k \frac{2j+1}{2n} \pi) \quad , \quad k = 1, \dots, n-1 , \quad (9.4.11)$$

$$c_0 = \frac{1}{\sqrt{2}} \sum_{j=0}^{n-1} y_j .$$

MATLAB-implementation of  $\mathbf{C}\mathbf{y}$  ("wrapping"-technique):

**MATLAB-code 9.4.12:** MATLAB-CODE cosine transform

```

1 function c = costrans(y)
2 n = length(y);
3 z = fft ([y,y(end):-1:1]);
4 C = real ([z(1)/(2*sqrt(2)), ...
5 0.5*(exp (-i*pi/(2*n)).^(1:n-1)).*z(2:n)]);

```

MATLAB-implementation of  $\mathbf{C}_n^{-1}\mathbf{y}$  (“Wrapping”-technique):

```

MATLAB-CODE : Inverse cosine transform
function y=icostrans(c)
n = length(c);
y = [sqrt(2)*c(1), (exp(i*pi/(2*n)).^(1:n-1)).*c(2:end)];
y = ifft([y,0,conj(y(end:-1:2))]);
y = 2*y(1:n);

```

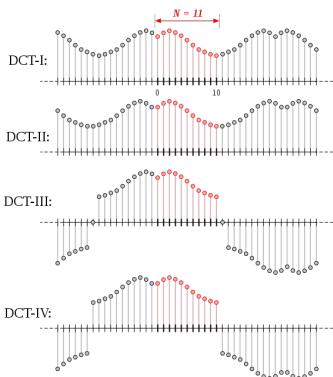
**Remark 9.4.13 (Cosine transforms for compression)**

The cosine transforms discussed above are named DCT-II and DCT-III.

Various cosine transforms arise by imposing various boundary conditions:

- DCT-II: even around  $-1/2$  and  $N - 1/2$
- DCT-III: even around  $0$  and odd around  $N$

DCT-II is used in JPEG-compression while a slightly modified DCT-IV makes the main component of MP3, AAC and WMA formats.



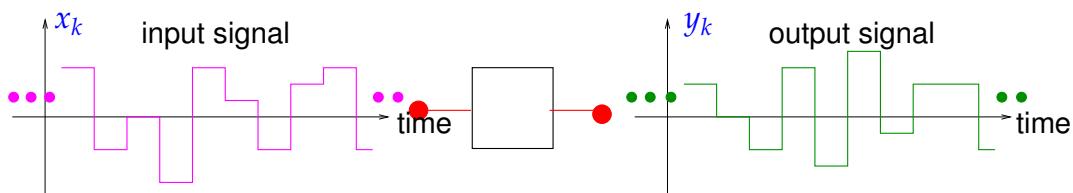
## 9.5 Toeplitz matrix techniques

**Example 9.5.1 (Parameter identification for linear time-invariant filters)**

- $(x_k)_{k \in \mathbb{Z}}$   $m$ -periodic discrete signal = *known* input
- $(y_k)_{k \in \mathbb{Z}}$   $m$ -periodic *measured*<sup>(\*)</sup> output signal of a *linear time-invariant filter*, see Ex. 9.1.1.  
<sup>(\*)</sup> → measurement errors !
- Known: impulse response of filter has maximal duration  $n\Delta t$ ,  $n \in \mathbb{N}$ ,  $n \leq m$

cf. (9.1.3)

$$\exists \mathbf{h} = (h_0, \dots, h_{n-1})^\top \in \mathbb{R}^n, \quad n \leq m : \quad y_k = \sum_{j=0}^{n-1} h_j x_{k-j}. \quad (9.5.2)$$



If the  $y_k$  were exact, we could retrieve  $h_0, \dots, h_{n-1}$  by examining only  $y_0, \dots, y_{n-1}$  and inverting the discrete periodic convolution ( $\rightarrow$  Def. 9.1.13) using (9.2.12).

However, in case the  $y_k$  are affected by measurement errors we have to use all available  $y_k$  for a **least squares estimate** of the impulse response.

Parameter identification problem: seek  $\mathbf{h} = (h_0, \dots, h_{n-1})^\top \in \mathbb{R}^n$  with

$$\|\mathbf{A}\mathbf{h} - \mathbf{y}\|_2 = \left\| \begin{bmatrix} x_0 & x_{-1} & \cdots & \cdots & x_{1-n} \\ x_1 & x_0 & x_{-1} & & \vdots \\ \vdots & x_1 & x_0 & \ddots & \\ & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & x_{-1} \\ x_{n-1} & & & x_1 & x_0 \\ x_n & x_{n-1} & & x_1 & x_1 \\ \vdots & & & \vdots & \\ x_{m-1} & \cdots & \cdots & x_{m-n} & \end{bmatrix} \begin{bmatrix} h_0 \\ \vdots \\ h_{n-1} \end{bmatrix} - \begin{bmatrix} y_0 \\ \vdots \\ y_{m-1} \end{bmatrix} \right\|_2 \rightarrow \min .$$

➤ **Linear least squares problem**,  $\rightarrow$  Chapter 6 with Toeplitz matrix  $\mathbf{A}$ :  $(\mathbf{A})_{ij} = x_{i-j}$ .

System matrix of normal equations ( $\rightarrow$  Section 6.1)

$$\mathbf{M} := \mathbf{A}^H \mathbf{A} , \quad (\mathbf{M})_{ij} = \sum_{k=1}^m x_{k-i} x_{k-j} = z_{i-j} \quad \text{due to periodicity of } (x_k)_{k \in \mathbb{Z}} .$$

➤  $\mathbf{M} \in \mathbb{R}^{n,n}$  is a *matrix with constant diagonals* & s.p.d.  
("constant diagonals"  $\Leftrightarrow (\mathbf{M})_{i,j}$  depends only on  $i - j$ )

### Example 9.5.3 (Linear regression for stationary Markov chains)

Sequence of scalar random variables:  $(Y_k)_{k \in \mathbb{Z}}$  = Markov chain

Assume: **stationary** (time-independent) correlation

$$\text{Expectation} \quad \mathcal{E}(Y_{i-j} Y_{i-k}) = u_{k-j} \quad \forall i, j, k \in \mathbb{Z} , \quad u_i = u_{-i} .$$

Model: finite linear relationship

$$\exists \mathbf{x} = (x_1, \dots, x_n)^\top \in \mathbb{R}^n: \quad Y_k = \sum_{j=1}^n x_j Y_{k-j} \quad \forall k \in \mathbb{Z} .$$

with *unknown* parameters  $x_j, j = 1, \dots, n$ : for fixed  $i \in \mathbb{Z}$

$$\text{Estimator} \quad \mathbf{x} = \underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} E \left| Y_i - \sum_{j=1}^n x_j Y_{i-j} \right|^2 \quad (9.5.4)$$

$$\Rightarrow E|Y_i|^2 - 2 \sum_{j=1}^n x_j u_k + \sum_{k,j=1}^n x_k x_j u_{k-j} \rightarrow \min .$$

$$\blacktriangleright \quad \mathbf{x}^\top \mathbf{A} \mathbf{x} - 2\mathbf{b}^\top \mathbf{x} \rightarrow \min \quad \text{with} \quad \mathbf{b} = (u_k)_{k=1}^n, \quad \mathbf{A} = (u_{i-j})_{i,j=1}^n.$$

Lemma 8.1.3  $\Rightarrow$   $\mathbf{x}$  solves  $\mathbf{Ax} = \mathbf{b}$  (Yule-Walker-equation, see below)

$\mathbf{A} \triangleq \text{Covariance matrix}$ : s.p.d. matrix with constant diagonals.

Matrices with constant diagonals occur frequently in mathematical models. They generalize circulant matrices  $\rightarrow$  Def. 9.1.17.

Note: “Information content” of a matrix  $\mathbf{M} \in \mathbb{K}^{m,n}$  with constant diagonals, that is,  $(\mathbf{M})_{i,j} = m_{i-j}$ , is  $m+n-1$  numbers  $\in \mathbb{K}$ .

### Definition 9.5.5. Toeplitz matrix

$\mathbf{T} = (t_{ij})_{i,j=1}^n \in \mathbb{K}^{m,n}$  is a **Toeplitz matrix**, if there is a vector  $\mathbf{u} = (u_{-m+1}, \dots, u_{n-1}) \in \mathbb{K}^{m+n-1}$  such that  $t_{ij} = u_{j-i}$ ,  $1 \leq i \leq m$ ,  $1 \leq j \leq n$ .

$$\mathbf{T} = \begin{bmatrix} u_0 & u_1 & \cdots & & \cdots & u_{n-1} \\ u_{-1} & u_0 & u_1 & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & u_1 \\ u_{1-m} & \cdots & & \cdots & u_{-1} & u_0 \end{bmatrix}$$

### 9.5.1 Toeplitz matrix arithmetic

$\mathbf{T} = (u_{j-i}) \in \mathbb{K}^{m,n}$  = Toeplitz matrix with generating vector  $\mathbf{u} = (u_{-m+1}, \dots, u_{n-1}) \in \mathbb{C}^{m+n-1}$

Task: efficient evaluation of matrix  $\times$  vector product  $\mathbf{Tx}$ ,  $\mathbf{x} \in \mathbb{K}^n$

Note: the following extended matrix is **circulant** ( $\rightarrow$  Def. 9.1.17)

$$\mathbf{C} = \begin{bmatrix} \mathbf{T} & \mathbf{S} \\ \mathbf{S} & \mathbf{T} \end{bmatrix} = \begin{bmatrix} u_0 & u_1 & \cdots & & \cdots & u_{n-1} & 0 & u_{1-n} & \cdots & & \cdots & u_{-1} \\ u_{-1} & u_0 & u_1 & & & \vdots & u_{n-1} & 0 & \ddots & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & \vdots & \vdots & \ddots & \ddots & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots & & & \ddots & & & \vdots \\ \vdots & & & \ddots & \ddots & u_1 & \vdots & & \ddots & \ddots & & u_{1-n} \\ u_{1-n} & \cdots & \cdots & u_{-1} & u_0 & \vdots & u_1 & & \cdots & \ddots & \ddots & u_{1-n} \\ 0 & u_{1-n} & \cdots & & \cdots & u_{-1} & u_0 & u_1 & \cdots & & \cdots & u_{n-1} \\ u_{n-1} & 0 & \ddots & & & \vdots & u_{-1} & u_0 & u_1 & & & \vdots \\ \vdots & \ddots & \ddots & & & \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & u_{1-n} & \vdots & \vdots & \ddots & \ddots & \ddots & & u_1 \\ u_1 & & u_{n-1} & 0 & u_{1-n} & u_{1-n} & \cdots & \cdots & u_{-1} & u_0 \end{bmatrix}$$

$\triangle$  This example demonstrates the case  $m = n$ .

Remember Rem. 9.1.19: extension to a circulant matrix to convert general discrete convolution ( $\rightarrow$  Def. 9.1.7) to periodic discrete convolution ( $\rightarrow$  Def. 9.1.13), compare (9.1.22) and the above matrix augmentation.

In general:

$$\begin{aligned} \mathbf{T} &= \text{toeplitz}(u(0:-1:1-m), u(0:n-1)); \\ \mathbf{S} &= \text{toeplitz}([0, u(n-1:-1:n-m+1)], [0, u(1-m:1:-1)]); \end{aligned}$$

$$\xrightarrow{\text{zero padding}} \mathbf{C} \begin{bmatrix} \mathbf{x} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{Tx} \\ \mathbf{Sx} \end{bmatrix}$$

 Computational effort  $O(n \log n)$  for computing  $\mathbf{Tx}$  (FFT based, Section 9.3)

### 9.5.2 The Levinson algorithm

Given: **S.p.d.** Toeplitz matrix  $\mathbf{T} = (u_{j-i})_{i,j=1}^n$ , generating vector  $\mathbf{u} = (u_{-n+1}, \dots, u_{n-1}) \in \mathbb{C}^{2n-1}$   
(Symmetry  $\leftrightarrow u_{-k} = u_k$ , w.l.o.g  $u_0 = 1$ )

Task: efficient solution algorithm for LSE  $\mathbf{Tx} = \mathbf{b}$ ,  $\mathbf{b} \in \mathbb{C}^n$   
(Yule-Walker problem)

Recursive (inductive) solution strategy:

Define:

- \*  $\mathbf{T}_k := (u_{j-i})_{i,j=1}^k \in \mathbb{K}^{k,k}$  (left upper block of  $\mathbf{T}$ )  $\Rightarrow$   $\mathbf{T}_k$  is s.p.d. Toeplitz matrix,
- \*  $\mathbf{x}^k \in \mathbb{K}^k$ :  $\mathbf{T}_k \mathbf{x}^k = (b_1, \dots, b_k)^\top \Leftrightarrow \mathbf{x}^k = \mathbf{T}_k^{-1} \mathbf{b}^k$ ,
- \*  $\mathbf{u}^k := (u_1, \dots, u_k)^\top$

Block-partitioned LSE, cf. Rem. 1.6.32, Rem. 1.6.46

$$\mathbf{T}_{k+1} \mathbf{x}^{k+1} = \left[ \begin{array}{c|c} \mathbf{T}_k & \begin{matrix} u_k \\ \vdots \\ u_1 \end{matrix} \\ \hline u_k & \cdots & u_1 & 1 \end{array} \right] \left[ \begin{array}{c} \tilde{\mathbf{x}}^{k+1} \\ \hline x_{k+1}^{k+1} \end{array} \right] = \left[ \begin{array}{c} b_1 \\ \vdots \\ b_k \\ \hline b_{k+1} \end{array} \right] = \left[ \begin{array}{c} \tilde{\mathbf{b}}^{k+1} \\ \hline b_{k+1} \end{array} \right] \quad (9.5.6)$$

Reversing permutation:  $P_k : \{1, \dots, k\} \mapsto \{1, \dots, k\}$ ,  $P_k(i) := k - i + 1$

$$\begin{aligned} \tilde{\mathbf{x}}_{k+1} &= \mathbf{T}_k^{-1} (\tilde{\mathbf{b}}^{k+1} - x_{k+1}^{k+1} P_k \mathbf{u}^k) = \mathbf{x}^k - x_{k+1}^{k+1} \mathbf{T}_k^{-1} P_k \mathbf{u}^k, \\ x_{k+1}^{k+1} &= b_{k+1} - P_k \mathbf{u}^k \cdot \tilde{\mathbf{x}}^{k+1} = b_{k+1} - P_k \cdot \mathbf{x}^k + x_{k+1}^{k+1} P_k \cdot \mathbf{T}_k^{-1} P_k \mathbf{u}^k. \end{aligned} \quad (9.5.7)$$

Efficient algorithm by using auxiliary vectors:  $\mathbf{y}^k := \mathbf{T}_k^{-1} P_k \mathbf{u}^k$

$$\mathbf{x}^{k+1} = \begin{bmatrix} \tilde{\mathbf{x}}^{k+1} \\ x_{k+1}^{k+1} \end{bmatrix} \quad \text{with} \quad \begin{aligned} x_{k+1}^{k+1} &= (b_{k+1} - P_k \mathbf{u}^k) / \sigma_k \\ \tilde{\mathbf{x}}^{k+1} &= \mathbf{x}^k - x_{k+1}^{k+1} \mathbf{y}^k \end{aligned}, \quad \sigma_k := 1 - P_k \mathbf{u}^k \cdot \mathbf{y}^k. \quad (9.5.8)$$

Levinson algorithm  
(recursive,  $u_{n+1}$  not used!)

Linear recursion:

Computational cost  $\sim (n - k)$  on level  $k$ ,  $k = 0, \dots, n - 1$

➤ Asymptotic complexity  $O(n^2)$

Listing 9.18: Levinson algorithm

```

1  function [x,y] = levinson(u,b)
2  k = length(u)-1;
3  if (k == 0), x=b(1); y = u(1);
   return; end
4  [xk,yk] =
   levinson(u(1:k),b(1:k));
5  sigma = 1-dot(u(1:k),yk);
6  t =
   (b(k+1)-dot(u(k:-1:1),xk))/sigma;
7  x = [ xk-t*yk(k:-1:1);t];
8  s =
   (u(k+1)-dot(u(k:-1:1),yk))/sigma;
9  y = [yk-s*yk(k:-1:1); s];

```

### Remark 9.5.9 (Fast Toeplitz solvers)

FFT-based algorithms for solving  $\mathbf{T}\mathbf{x} = \mathbf{b}$  with asymptotic complexity  $O(n \log^3 n)$  [75] !



*Supplementary reading.* [15, Sect. 8.5]: Very detailed and elementary presentation, but the discrete Fourier transform through trigonometric interpolation, which is not covered in this chapter. Hardly addresses discrete convolution.

[42, Ch. IX] presents the topic from a mathematical point of few stressing approximation and trigonometric interpolation. Good reference for algorithms for circulant and Toeplitz matrices.

[70, Ch. 10] also discusses the discrete Fourier transform with emphasis on interpolation and (least squares) approximation. The presentation of signal processing differs from that of the course.

There is a vast number of books and survey papers dedicated to discrete Fourier transforms, see, for instance, [21, 11]. Issues and technical details way beyond the scope of the course are treated there.

# Chapter 10

## Clustering Techniques

### Contents

10.1 Kernel Matrices . . . . .	579
10.2 Local Separable Approximation . . . . .	580
10.3 Cluster Trees . . . . .	587
10.4 Algorithm . . . . .	592

### 10.1 Kernel Matrices

TASK:

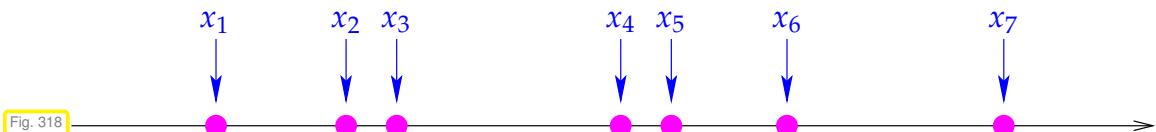
Given:

Kernel Function  $G : I \times J \mapsto \mathbb{C}$ ,  $I, J \subset \mathbb{R}$  Interval:  $G(x, y)$  smooth for  $x \neq y$   
Collocation Points  $x_1 < x_2 < \dots < x_n, x_j \in I, y_1 < y_2 < \dots < y_m, y_j \in J$

Collocation Matrix:  $\mathbf{M} \in \mathbb{C}^{n,m} \Leftrightarrow (\mathbf{M})_{ij} := G(x_i, y_j), 1 \leq i \leq n, 1 \leq j \leq m.$  (10.1.1)

We have to find: Efficient algorithms for approximate evaluation of  $\mathbf{M} \times$  Vector  
(Note: Computational Effort  $\mathcal{O}(mn)$  !)

#### Example 10.1.2 (Interaction calculations for many body systems)



$n$  parallel wires with current flowing through them.

Wire  $j$  has current  $c_j \in \mathbb{R}$ , and is at position  $x_j \in \mathbb{R}$

Our Aim : To compute magnetic force on each wire

- Force on wire  $j$  due to all wires:  $f_j = \sum_{\substack{k=1 \\ k \neq j}}^n \frac{1}{|x_j - x_k|} c_k c_j, j = 1, \dots, n.$
- Force on every wire is given by vector  $\mathbf{f} = \text{diag}(c_1, \dots, c_n) \mathbf{M} \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix},$

where  $\mathbf{M} = (m_{ij})_{i,j=1}^n$ ,  $m_{ij} = \begin{cases} \frac{1}{|x_j - x_i|} & \text{for } i \neq j, \\ 0 & \text{for } i = j. \end{cases}$

Collocation matrix  $\mathbf{M}$  will be formed using **kernel function**  $G(x, y) = \frac{1}{|x-y|}$

### Example 10.1.3 (Gravitational forces in galaxy)

Number of stars in galaxy  $n$  ( $\approx 10^9$ ) with position  $x_i \in \mathbb{R}^3$  and mass  $m_i$ ,  $i = 1, \dots, n$ .

Gravitational force on each star is required for simulation of dynamics of galaxy

Our Aim : To compute gravitational force on each star

- Gravitational force on star  $j$  :

$$f_j = \frac{G}{4\pi} \sum_{i \neq j} \frac{1}{\|\mathbf{x}_i - \mathbf{y}_j\|} m_i m_j, \\ j \in \{1, \dots, n\}.$$

Fig. 319



- Gravitational force on every star is given by

$$\mathbf{f} = \text{diag}(m_1, \dots, m_n) \mathbf{M} \begin{bmatrix} m_1 \\ \vdots \\ m_n \end{bmatrix}, \quad m_{ij} := \begin{cases} \frac{G}{4\pi} \frac{1}{\|\mathbf{x}_i - \mathbf{y}_j\|} & \text{for } i \neq j, \\ 0 & \text{for } i = j, \end{cases} \quad 1 \leq i, j \leq n.$$

The above example is a 3D generalization of our original task.

## 10.2 Local Separable Approximation

If kernel function is **separable** i.e. :  $G(x, y) = g(x)h(y)$ ,  $g : I \mapsto \mathbb{C}, h : J \mapsto \mathbb{C}$

$$\mathbf{M} = [g(x_j)]_{j=1}^n \cdot [(h(y_j))]_{j=1}^m]^T \quad \text{rank}(\mathbf{M}) = 1.$$

Computational Effort ( $\mathbf{M} \times$  Vector) =  $m + n$

$$\mathbf{M} \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \left\{ \begin{bmatrix} \text{Scalar Product} \end{bmatrix} \right\} \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}$$

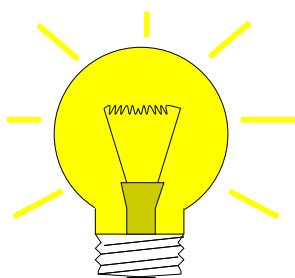
Generalization:  $G(x, y) = \sum_{j=1}^q g_j(x)h_j(y), \quad g_j : I \mapsto \mathbb{C}, h_j : J \mapsto \mathbb{C}, q \in \mathbb{N}$

$$\mathbf{M} = \mathbf{U}\mathbf{V}^T, \quad \mathbf{U} \in \mathbb{R}^{n,q}, \quad u_{ij} = g_j(x_i), \quad j \in \{1, \dots, q\}, i \in \{1, \dots, n\}, \\ \mathbf{V} \in \mathbb{R}^{q,m}, \quad v_{ij} = h_i(y_j), \quad i \in \{1, \dots, q\}, j \in \{1, \dots, m\}.$$

If  $\text{rank}(\mathbf{M}) = q$  then Computational Effort ( $\mathbf{M} \times$  Vector) =  $q(m + n)$

$$\mathbf{M} \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \left\{ \begin{bmatrix} \mathbf{U} \\ \mathbf{V}^T \end{bmatrix} \right\} \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}$$

$q$  Scalar Product



Idea: Global approximation of  $G$  through sum of separated kernel function:

$$G(x, y) \approx \tilde{G}(x, y) := \sum_{l=0}^d \sum_{k=0}^d \kappa_{l,k} g_l(x)h_k(y), \quad \kappa_{l,k} \in \mathbb{C}, \quad (x, y) \in I \times J. \quad (10.2.1)$$

As  $G$  is approximated by  $\tilde{G}$  therefore  $\mathbf{M}$  is approximated by  $\widetilde{\mathbf{M}}$ , where  $\tilde{m}_{ij} = \tilde{G}(x_i, y_j)$

**Remark 10.2.2 (Quality measure for kernel approximation)**

$$\|\mathbf{M} - \widetilde{\mathbf{M}}\|_2^2 \leq \|\mathbf{M} - \widetilde{\mathbf{M}}\|_F^2 = \sum_{i,j} (m_{ij} - \tilde{m}_{ij})^2 = \boxed{\sum_{i,j} (G(x_i, y_j) - \widetilde{G}(x_i, y_j))^2} .$$

Normalizing quality measure:  $\frac{1}{mn} \sum_{i,j} (G(x_i, y_j) - \widetilde{G}(x_i, y_j))^2 . \quad (10.2.3)$

Now, our aim is to find separable kernel approximation

**Definition 10.2.4. Tensor product interpolation polynomial**

$L_j^x \in \mathcal{P}_n$  ( $L_k^y \in \mathcal{P}_m$ ),  $j = 0, \dots, n$  ( $k = 0, \dots, m$ ), Lagrange polynomial on nodes of mesh  $\mathcal{X} := \{x_j\}_{j=0}^n \subset I$  ( $\mathcal{Y} := \{y_k\}_{k=0}^m \subset J$ ),  $I, J \subset \mathbb{R}$  interval. Continuous  $f : I \times J \mapsto \mathbb{C}$  defined by

$$(I_{\mathcal{X} \times \mathcal{Y}} f)(x, y) := \sum_{j=0}^n \sum_{k=0}^m f(x_j, y_k) L_j^x(x) L_k^y(y) , \quad x, y \in \mathbb{R} ,$$

This is **tensor product interpolation polynomial**.

Section 4.1.3: Approximating  $G$  using **tensor product Chebyshev interpolation polynomial**

$$G(x, y) \approx \widetilde{G}(x, y) := \sum_{j=0}^d \sum_{k=0}^d G(t_j^x, t_k^y) L_j^x(x) L_k^y(y) , \quad (10.2.5)$$

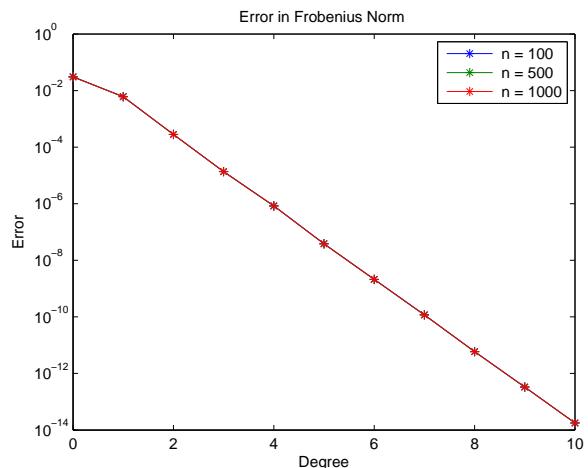
$t_0^x, \dots, t_d^x / t_0^y, \dots, t_d^y$  Chebyshev-nodes in  $I/J$ , and  $L_j^x, L_k^y$  subordinate Lagrange-polynomial.

**Example 10.2.6 (Global separable approximation by smooth kernel function)**

- Smooth (even analytic) kernel function  $G(x, y) = \frac{1}{1 + |x - y|^2}$ , collocation points  $x_i = y_i = \frac{i-1}{n}, i = 1, \dots, n$ .
- $\tilde{G}$  approximated by tensor product Chebyshev interpolation polynomial (10.2.10) of degree  $d$  [Fig. 320]

$$\frac{1}{n^2} \sum_{i,j} (G(x_i, y_j) - \tilde{G}(x_i, y_j))^2$$

as function of  $d$  for  $n \in \{100, 200, 400\}$



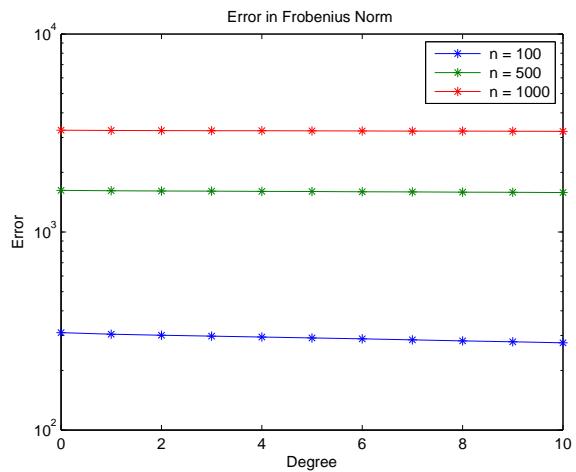
In file changeable.c

- put the admissibility condition as return true in admissible\_ClusterPair
- put kernel function as  $1.0 / (1.0 + \text{pow}(\text{fabs}(dX - dY), 2))$  in kernel\_function
- put NONE as return values in get\_iOperation
  - Run main.cpp, input as follows  
Choice - 2, NumberofPts - 100 1000 0,  
Number of Increments - 2, Increment 1 - 400, Increment 2 - 500  
Start of 1 - 0, End of 1 - 1, Start of 2 - 0, End of 2 - 1  
Degree - 0 10 1, Admissibility Coefficient - 0 0 1  
Output File from C++: Error.txt
  - OutputFile from MATLAB : Error\_in\_Frobenius\_Norm\_Pts.eps
  - Run error\_plot\_NumberofPts.m from Matlab

Note:

Exponential Convergence  $\|\mathbf{M} - \tilde{\mathbf{M}}\|_F \rightarrow 0$  in dependence of  $d$

### Example 10.2.7 (Global separable approximation by non-smooth kernel function)



Analysis as in Ex. 10.2.6, now for the kernel function

$$G(x, y) = \begin{cases} \frac{1}{|x-y|} & \text{if } x \neq y, \\ 0 & \text{otherwise.} \end{cases} \quad (\text{Fig. 321})$$

from Ex. 10.1.2.

In file changeable.c  
 a) put the admissibility condition as return `true` in `admissible_ClusterPair`  
 b) put kernel function as `0.0` for `fabs(dX-dY) < EPS` as `1.0/fabs(dX - dY)` otherwise in `kernel_function`  
 c) put `NONE` as return values in `get_iOperation`  
 1) Run `main.cpp`, input as follows  
 Choice - 2, NumberofPts - 100 1000 0,  
 Number of Increments - 2, Increment 1 - 400, Increment 2 - 500  
 Start of 1 - 0, End of 1 - 1, Start of 2 - 0, End of 2 - 1  
 Degree - 0 10 1, Admissibility Coefficient - 0 0 1  
 Output File from C++: `Error.txt`  
 OutputFile from MATLAB : `Error_in_Frobenius_Norm_Pts.eps`  
 2) Run `error_plot_NumberofPts.m` from Matlab

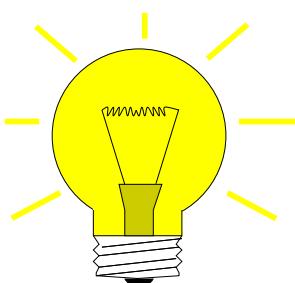
Note: (Virtually) No Convergence  $\|\mathbf{M} - \widetilde{\mathbf{M}}\|_F \rightarrow 0$  for  $d \rightarrow \infty$

Reason: Missing *global* smoothness

Poor approximation of  $G(x, y) := |x - y|^{-1}$  in region of  $\{(x, y) \in I \times J : x = y\}$ .

However  $G(x, y)$  from (10.2.8) is smooth (even analytic, Def. 4.1.55) at “large distances” from  $\{(x, y) \in I \times J : x = y\}$ .

Idea: *Local* approximation of  $G$  through sum of separated kernel function



$$G(x, y) \approx \sum_{l=0}^d \sum_{k=0}^d \kappa_{l,k} g_l(x) h_k(y), \quad \kappa_{l,j} \in \mathbb{C}, \quad (10.2.9)$$

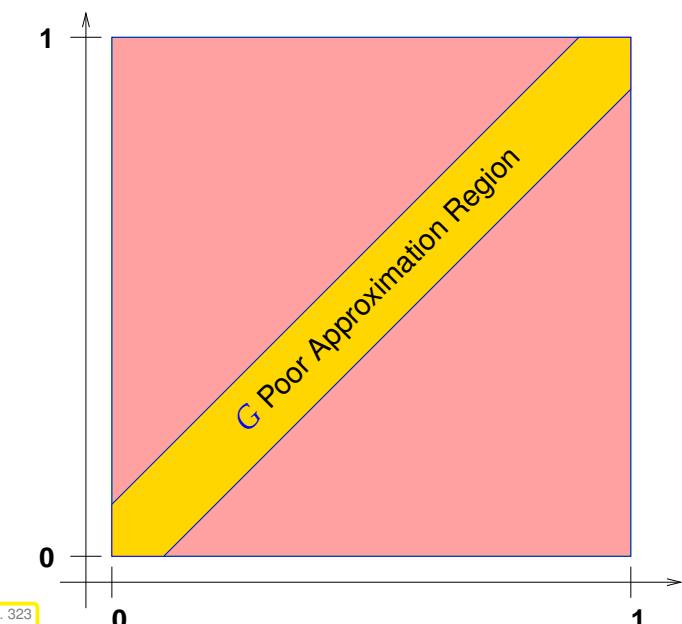
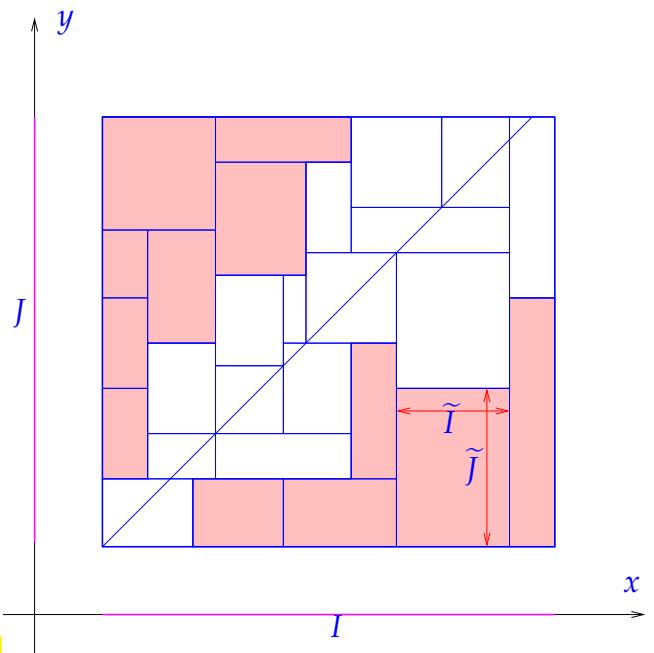
$$(x, y) \in \tilde{I} \times \tilde{Y}, \quad \tilde{I} \subset I, \quad \tilde{J} \subset J,$$

with  $\tilde{I} \times \tilde{J} \cap \{(x, y) : x = y\} = \emptyset$ .

Actually:

Local approximation of  $G$  on rectangle  $\tilde{I} \times \tilde{J}$  which is a *partition* of  $I \times J$

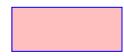
 Possible partition for separable kernel approximation through local tensor product Chebyshev polynomial interpolation (10.2.9)  
**(Admissible rectangles)**



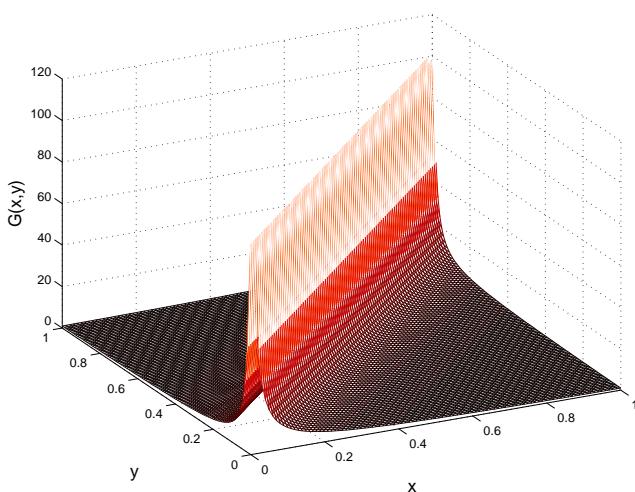
Terminology:



**Near Field:**  $G$  Poor approximation



**Far Field:**  $G$  Good approximation



Kernel function (10.2.8) for  $[0, 1]^2$

$$G(x, y) = \frac{1}{|x - y|}$$

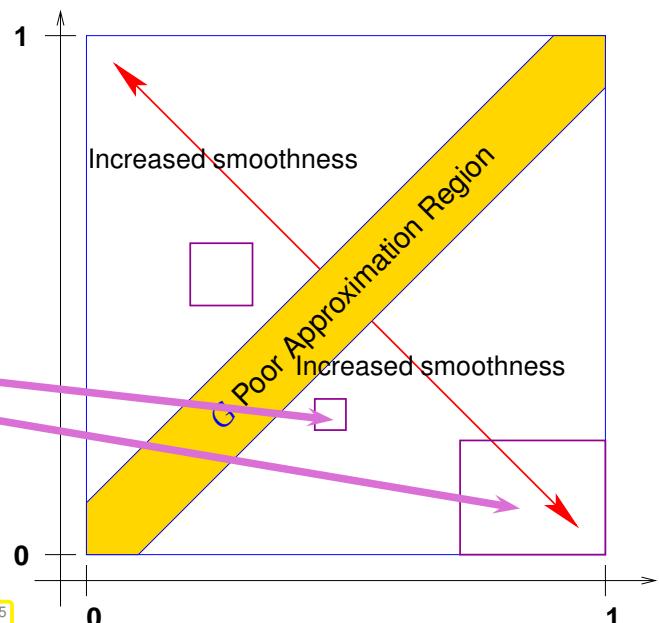
- Singular for  $x = y$
- Analytic far from  $\{(x, y) : x = y\}$ , analyticity ( $\rightarrow ??$ ) increases with  $|x - y|$

Similar kernel function:  $G(x, y) = \log|x - y|$ ,  $G(x, y) = \frac{\partial}{\partial x} \frac{1}{|x - y|}$ , etc.

"Smoothness of  $G(x, y) = \frac{1}{|x-y|}$  increases with growing distance from diagonal  $\{x = y\}$ :

- No approximation, when  $|x - y|$  is "small"
- Tensor product interpolation polynomial for
  - Small rectangles near diagonal
  - Large rectangles far from diagonal

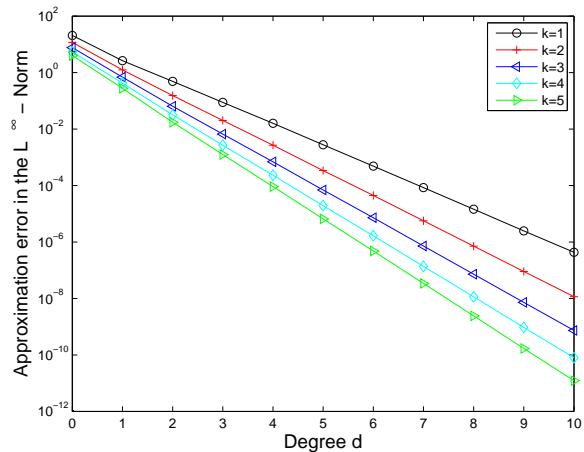
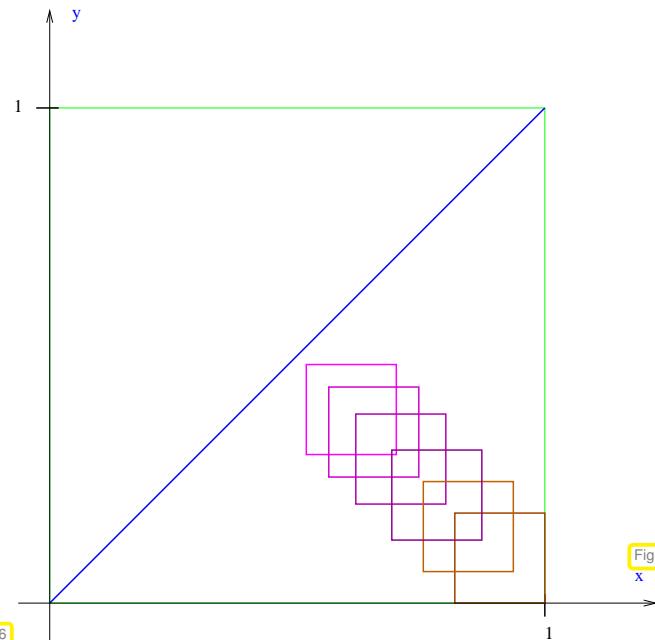
Now, our aim is to find appropriate sizes of above mentioned rectangles



#### Example 10.2.10 (Tensor product Chebyshev interpolation on rectangles)

$I = J = [0, 1]^2$ ,  $G(x, y) = |x - y|^{-1}$  from (10.2.8), (Approximate) Max norm of the error of local tensor product Chebychev interpolation polynomial of  $G$  on rectangles of constant size, but with growing distance from  $\{(x, y) : x = y\}$

$$\tilde{I}_k = [0.55 + k \cdot 0.05, 0.75 + k \cdot 0.05], \quad \tilde{J}_k = [0.25 - k \cdot 0.05, 0.45 - k \cdot 0.05], \quad k \in \{0, \dots, 5\}.$$



NOTE: Decreasing interpolation errors with increasing distance from  $\{(x, y) : x = y\}$

#### Example 10.2.11 (Tensor product Chebyshev interpolation for variable rectangle sizes)

Taking the variable rectangle sizes as

$$[\frac{1}{2}(\sqrt{2}-1)\xi + \frac{1}{2}, \frac{1}{2}(\sqrt{2}+1)\xi + \frac{1}{2}] \times [-\frac{1}{2}(\sqrt{2}-1)\xi + \frac{1}{2}, -\frac{1}{2}(\sqrt{2}+1)\xi + \frac{1}{2}], \quad 0.05 \leq \xi \leq \frac{1}{1+\sqrt{2}}.$$

that is, Size of rectangles  $\sim$  Distance from diagonal

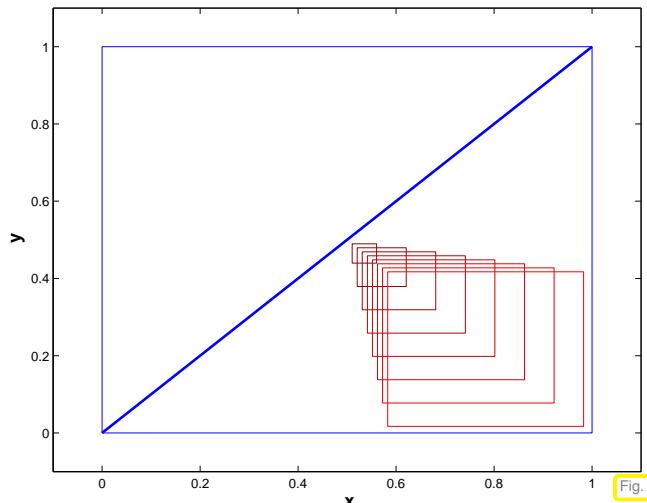


Fig. 328

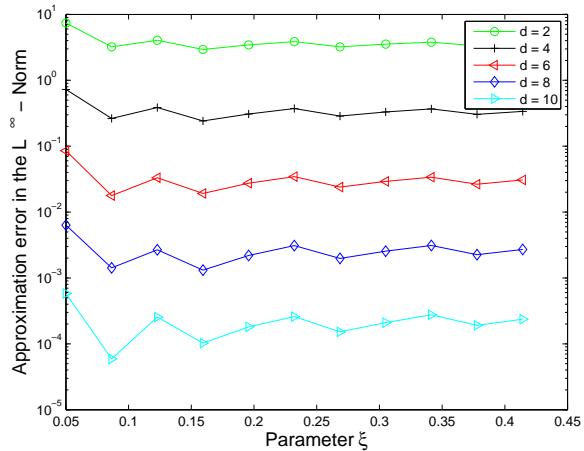


Fig. 329

**NOTE:** Controlled decrease of interpolation errors with increase in size of rectangles

### Acceptability Criteria

$[a, b] \times [c, d]$  is called  $\eta$ -admissible,  $\eta > 0$ , if

$$\eta \text{ dist}([a, b], [c, d]) \geq \max\{b - a, d - c\}. \quad (10.2.12)$$

$$(\text{dist}([a, b], [c, d])) := \min\{|x - y| : x \in [a, b], y \in [c, d]\}$$

distance from diagonal

Size of the rectangle

## 10.3 Cluster Trees

**TASK:** Given  $\eta > 0$ , find (efficient algorithms) partitioning of  $I \times J$  "far from diagonal" which are  $\eta$ -admissible rectangles.

### Remark 10.3.1.

Perspective:

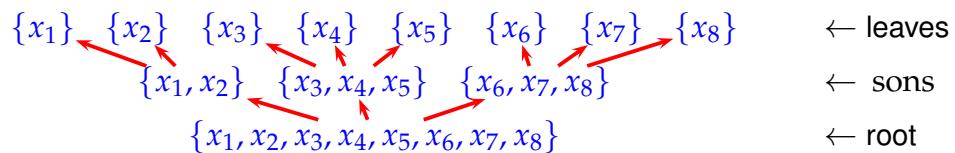
$$\begin{array}{ccc} \text{Partitioning} & \implies & \text{Partitioning} \\ \text{of } I \times J & & \text{of } \{x_i\}_{i=1}^n \times \{y_j\}_{j=1}^m \end{array} \Leftrightarrow \begin{array}{c} \text{Partitioning of} \\ \text{index mesh } \{1, \dots, n\} \times \{1, \dots, m\} \end{array}$$

### Definition 10.3.2. Cluster Tree

A Tree ( $\rightarrow$  Computer Science, Graph theory)  $T$  is called **Cluster Tree** on  $\mathbb{P} := \{x_1, \dots, x_n\} \subset \mathbb{R}$   
 $\Leftrightarrow$

- The nodes of the Tree  $T$  (= Cluster) are subset of  $\mathbb{P}$ .
- $\text{root}(T) = \mathbb{P}$ .
- For every node  $\sigma$  of  $T$ :  $\{\sigma' : \sigma' \in \text{sons}(\sigma)\}$  is Partitioning of  $\sigma$ .

Bounding Box of Cluster  $\sigma \in T$ :  $\Gamma(\sigma) := [\min\{x\}_{x \in \sigma}, \max\{x\}_{x \in \sigma}]$

Terminology ( $\rightarrow$  Graph theory):Cluster Tree  $T$ :  $\sigma \in T$ :  $\text{sons}(\sigma) = \emptyset \Leftrightarrow \sigma \text{ leaf}$ Cluster Tree  $T$ :

Recursive Construction

MATLAB-Data Structure:

 $N \times 6$ -Matrix,  $N = \#T$ ,Lines  $\hat{=}$  Cluster $T = [T; i, j, xl, xr, s1, s2]$  $i, j : \sigma = \{x_i, \dots, x_j\}$  $xl : xl = \min_{i \leq k \leq j} x_k$  $xr : xr = \max_{i \leq k \leq j} x_k$ 

s1, s2: The line indices of son

NOTE:  $\#T \leq 2n$ 

## MATLAB-code 10.3.3: Construction of Cluster Trees

```

1 function [T, idx] =
2   ct\_rec(x, ofs, m, T)
3 N = length(x);
4 if (N <= m)
5   T =
6     [T;ofs,ofs+N-1,x(1),x(end),0,0];
7 else
8   n = round(N/2);
9   [T,s1] = ct\_rec(x(1:n),ofs,m,T);
10  [T,s2] =
11    ct\_rec(x(n+1:N),ofs+n,m,T);
12  T =
13    [T;ofs,ofs+N-1,x(1),x(end),s1,s2];
14 end
15 idx = size(T,1);

```

**C++-code 10.3.4: Construction of Cluster Trees**

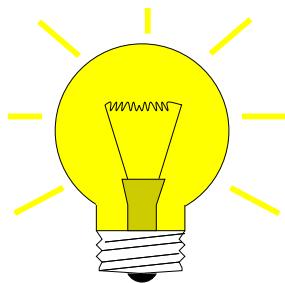
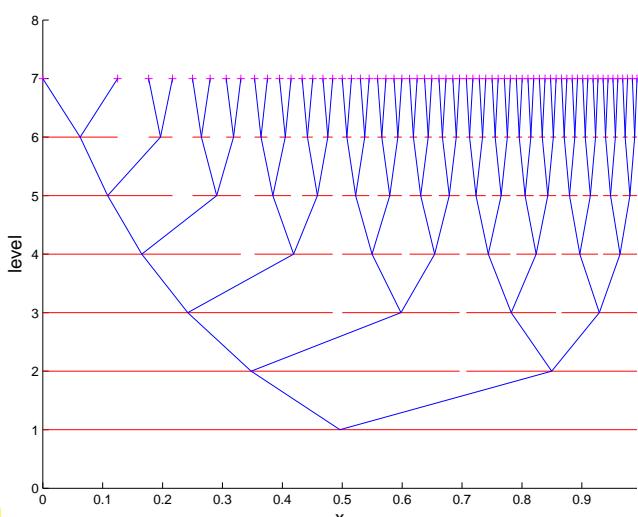
```
void clsClusteringApproximation::build_ClusterTree_Linear(int iOffset,
    int iNumberofPtsInClusterLinear, int* piDimension) {
    int iTempLeftChild, iTempRightChild, iStart, iEnd;
    int iStartLeft, iPtsInLeft, iStartRight, iPtsInRight;

    iStart = iOffset;
    iEnd = iOffset + iNumberofPtsInClusterLinear - 1;
    if (iNumberofPtsInClusterLinear == 1) {
        add_ClusterLinear(*piDimension, iStart, iEnd, -1, -1);
    }
    else{
        iStartLeft = iOffset;
        iPtsInLeft = iNumberofPtsInClusterLinear -
            (iNumberofPtsInClusterLinear/2);
        build_ClusterTree_Linear(iStartLeft, iPtsInLeft, piDimension);
        iTempLeftChild = vec2pclsClusterLinear[*piDimension].size() - 1;

        iStartRight = iOffset + iPtsInLeft;
        iPtsInRight = iNumberofPtsInClusterLinear/2;
        build_ClusterTree_Linear(iStartRight, iPtsInRight, piDimension);
        iTempRightChild = vec2pclsClusterLinear[*piDimension].size() - 1;

        add_ClusterLinear(*piDimension, iStart, iEnd, iTempLeftChild,
            iTempRightChild);
    }
}
```

### Example 10.3.5 (Cluster Tree)



Idea: Choose approximation rectangles = Tensor product of Bounding Box of Cluster  $\in T$

Cluster  $\sigma, \mu$  admissible : $\Leftrightarrow$   
 $\Gamma(\sigma), \Gamma(\mu)$  admissible

Recursive construction for admissible Partitioning:

Pnear = Near field:

The area near the diagonal ( $\rightarrow$   
No approximation)

Pfar = Far field:

Partitioning with  $\eta$ -admissible  
rectangles

Listing 10.1: Recursive construction for admissible partitioning

```

1 function [Pnear,Pfar] = ...
2     divide(Tx,Ty,σ,μ,Pnear,Pfar,η)
3     cls = Tx(σ,:); clm = Ty(μ,:);
4     f (σ = @\textrm{leaf}@ | μ =
5         @\textrm{leaf}@)
6     Pnear = [Pnear;
7             cls(2),cls(3),clm(2),clm(3)];
8     elseif @\rnode{CLA}{((σ,μ)
9         \textrm{admissible})}@
10    Pfar = [Pfar; cls(2),cls(3),
11             clm(2),clm(3)];
12    else
13        for s1 = cls(6:7), for s2 = clm(6:7)
14        [Pnear,Pfar] =
15            divide(Tx,Ty,s1,s2,Pnear,Pfar,η);
16        end; end
17    end

```

### MATLAB-code 10.3.6: Parent Code

```

1 function [Pnear,Pfar] = partition(Tx,Ty,η)

```

```

2 Pnear = [] ; Pfar = [] ;
3 σ = find(Tx(:,1) == min(Tx(:,1))) ;
4 μ = find(Ty(:,1) == min(Ty(:,1))) ;
5 [Pnear,Pfar] = divide(Tx,Ty,σ,μ,Pnear,Pfar,η) ;

```

Note: clusters of a far field cluster pair have a common level!

#### C++-code 10.3.7: Construction for Admissible Partitioning

```

1 void clsClusteringApproximation :: build_ClusterPair (int iIndex1, int
2   iIndex2){

3   clsClusterLinear *pcclsClusterLinear1, *pcclsClusterLinear2;

4

5   pcclsClusterLinear1 = vec2pcclsClusterLinear[0][iIndex1];
6   pcclsClusterLinear2 = vec2pcclsClusterLinear[1][iIndex2];
7   if (leaf(0,iIndex1) || leaf(1,iIndex2)){
8       add_ClusterPairNear(pcclsClusterLinear1, pcclsClusterLinear2);
9       (pcclsClusterLinear1->get_ptr_AppearsIn())
10      ->push_back(-vecpclsClusterPairNear.size());
11      (pcclsClusterLinear2->get_ptr_AppearsIn())
12      ->push_back(-vecpclsClusterPairNear.size());
13  }
14  else if (admissible_ClusterPair(iIndex1, iIndex2)){
15      add_ClusterPairFar(pcclsClusterLinear1, pcclsClusterLinear2);
16      (pcclsClusterLinear1->get_ptr_AppearsIn())
17      ->push_back(vecpclsClusterPairFar.size());
18      (pcclsClusterLinear2->get_ptr_AppearsIn())
19      ->push_back(vecpclsClusterPairFar.size());
20  }
21  else{
22      tree_traverse(pcclsClusterLinear1, pcclsClusterLinear2);
23  }
24}

```

#### C++-code 10.3.8: Preprocessing Code

```

1 void clsClusteringApproximation :: preprocess(){
2     int iLoopVari, iLoopVarj;
3     for (iLoopVari = 0; iLoopVari < iDimension; iLoopVari++){
4         build_ClusterTree_Linear(0, veciNumberofPts[iLoopVari],
5             &iLoopVari);
6     }
7     build_ClusterPair(vec2pcclsClusterLinear[0].size() - 1,
8         vec2pcclsClusterLinear[1].size() - 1);
9 }

```

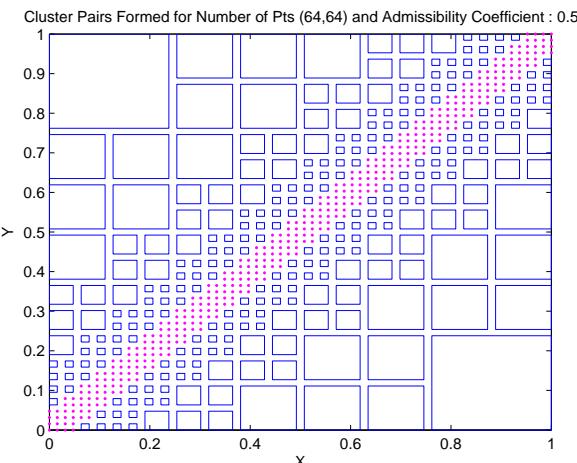


Fig. 331

```
x = (0:1/64:(1-1/64));
In file changeable.c
```

- a) put the admissibility condition in admissible\_ClusterPair
- b) put kernel function in kernel\_function
- c) put **NONE** as return values in get\_iOperation
  - 1) Run main.cpp, input as follows  
Choice - 2, NumberofPts - 64 64 1,  
Start of 1 - 0, End of 1 - 1, Start of 2 - 0, End of 2 - 1  
Degree - 0 0 1, Admissibility Coefficient - 0.5 0.5 1  
Output File from C++: ClusterLinear1\_AA.txt,  
ClusterLinear2\_AA.txt, ClusterFar\_AA.txt,  
ClusterLinearNear\_AA.txt,
  - 2) Run cluster\_pair\_indiv.m from Matlab, arguments:  
0 0 (argument here is for the Point Number  
and Admissibility Coefficient. As 64 Number of Pts is the  
first in the iteration over Number of Pts,  
therefore argument is given as 0  
same holds for Admissibility Coefficient)

Points on mesh(n): \* = Near field points , □ = Partition rectangles (Far field)

**NOTE :** Axis sections of the partition rectangles  $\in \{\Gamma(\sigma) : \sigma \in T\}$ ,  $T \triangleq$  Cluster Tree ( $\rightarrow$  Def. 10.3.2)

Local kernel approximation of  
Far field-Rectangles through Tensor  
product Chebyshev interpolation  
polynomial



Low rank approximation of  $\mathbf{M}$  on *blocks*  
as (10.1.1)  
see Fig. 331, Fig. 332

Notation:

$$\begin{aligned} \sigma &:= \{i_1, \dots, i_d\} \subset \{1, \dots, n\} \\ \mu &:= \{j_1, \dots, j_d\} \subset \{1, \dots, m\} \end{aligned} \Rightarrow \mathbf{M}_{|\sigma \times \mu}| := (m_{ij})_{\substack{i=i_1, \dots, i_d \\ j=j_1, \dots, j_d}}.$$

Consider kernel approximation of  $[x_{i_1}, x_{i_2}] \times [y_{j_1}, y_{j_2}]$  through Tensor product Chebyshev interpolation polynomial (see (10.2.10)):

$$G(x, y) \approx \tilde{G}(x, y) := \sum_{j=0}^d \sum_{k=0}^d G(t_j^x, t_k^y) L_j^x(x) L_k^y(y),$$

- \*  $t_0^x, \dots, t_d^x / t_0^y, \dots, t_d^y \hat{=} \text{Chebyshev nodes for } [x_{i_1}, x_{i_2}] / [y_{j_1}, y_{j_2}],$
  - \*  $L_j^x, L_k^y \hat{=} \text{associated Lagrange polynomial.}$

$$\widetilde{G}(x, y) = \sum_{j=0}^d \sum_{k=0}^d G(t_j^x, t_k^y) L_j^x(x) L_k^y(y), \quad x \in [x_{i_1}, x_{i_2}], y \in [y_{j_1}, y_{j_2}].$$

$$\widetilde{\mathbf{M}}_{|\sigma \times \mu} = \underbrace{\underbrace{\left( L_j^x(x_i) \right)_{\substack{i=1,\dots,i_2 \\ j=0,\dots,d}}}_{\in \mathbb{R}^{|\sigma|,d+1}} \underbrace{\left( G(t_j^x, t_k^y) \right)_{j,k=0,\dots,d}}_{\in \mathbb{R}^{d+1,d+1}} \underbrace{\left( L_k^y(y_j) \right)_{\substack{k=0,\dots,d \\ j=j_1,\dots,j_2}}}_{\in \mathbb{R}^{d+1,|\mu|}}}_{\text{rank} \leq d+1}.$$

$$\widetilde{\mathbf{M}}_{|\sigma \times \mu} = \left[ \begin{array}{c|c} & \\ \hline & \\ \end{array} \right] \quad \left[ \begin{array}{c|c} & \\ \hline & \\ \end{array} \right] \quad \left[ \begin{array}{c|c} & \\ \hline & \\ \end{array} \right]$$

## Application of partitioning

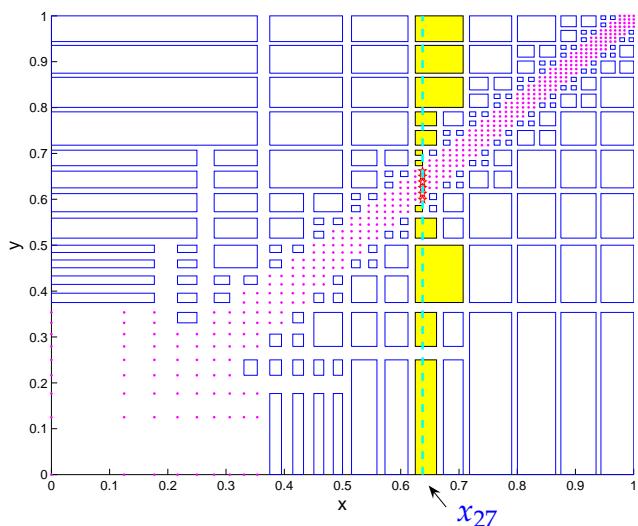
$$\{x_1, \dots, x_n\} \times \{y_1, \dots, y_m\} = \left( \bigcup_{(\sigma, \mu) \in P^{\text{near}}} \sigma \times \mu \right) \cup \left( \bigcup_{(\sigma, \mu) \in P^{\text{far}}} \sigma \times \mu \right).$$

Algorithm: Analysis of  $\mathbf{f} = \mathbf{M}\mathbf{c}$ ,  $\mathbf{M}$  as (10.1.1)

$$f_i = \sum_{j=1}^m G(x_i, y_j) c_j = \boxed{\sum_{j \in P_{\text{near}}(i)} G(x_i, y_j) c_j} + \boxed{\sum_{\substack{\sigma \in T_X \\ x_i \in \sigma}} \sum_{\substack{\mu \in T_Y \\ (\sigma, \mu) \in P_{\text{far}}}} \sum_{\substack{1 \leq j \leq m \\ y_j \in \mu}} G(x_i, y_j) c_j}.$$

↑  
Near field contribution      Far field contribution

Near field coupling indices  $P^{\text{near}}(i) := \{j \in \{1, \dots, m\} : (\{x_i\} \times \{y_j\}) \in P^{\text{near}}\}$



## Illustration:

```
x = sqrt(0:1/64:1);, i=27, η=1:
```

→ Far field clusters with contribution to  $f_i$

★→ Near field clusters with contribution to  $f_i$

Fig. 333

$$\begin{aligned} &\approx \sum_{j \in P^{\text{near}}(i)} G(x_i, y_j) c_j + \sum_{\substack{\sigma \in T_x \\ x_i \in \sigma}} \sum_{\substack{\mu \in T_y \\ (\sigma, \mu) \in P^{\text{far}}}} \sum_{\substack{1 \leq j \leq m \\ y_j \in \mu}} \sum_{l=0}^d \sum_{k=0}^d G(t_l^\sigma, t_k^\mu) L_l^\sigma(x_i) L_k^\mu(y_j) c_j \\ &\approx \sum_{j \in P^{\text{near}}(i)} G(x_i, y_j) c_j + \sum_{\substack{\sigma \in T_x \\ x_i \in \sigma}} \sum_{\substack{\mu \in T_y \\ (\sigma, \mu) \in P^{\text{far}}}} (\mathbf{V}_\sigma \mathbf{X}_{\sigma, \mu} \mathbf{V}_\mu^T \mathbf{c}_{|\mu})_i, \end{aligned}$$

with

$$t_l^\sigma, t_k^\mu := \text{Chebyshev nodes (4.1.78) in } \Gamma(\sigma), \Gamma(\mu), \quad l = 0, \dots, d,$$

$$\mathbf{X}_{\sigma, \mu} := (G(t_l^\sigma, t_k^\mu))_{l, k=0}^d \in \mathbb{R}^{d+1, d+1}, \quad (10.4.1)$$

$$\mathbf{V}_\sigma := (L_l^\sigma(x_i))_{\substack{i: x_i \in \sigma \\ l=0, \dots, d}} \in \mathbb{R}^{\#\sigma, d+1}, \quad (10.4.2)$$

$$\mathbf{V}_\mu := (L_k^\mu(y_j))_{\substack{j: y_j \in \mu \\ k=0, \dots, d}} \in \mathbb{R}^{\#\mu, d+1}, \quad (10.4.3)$$

### Analysis of Complexity (under the assumption $n \approx m$ ):

①: Trick: Calculate on above assumption

Compute  $\mathbf{w}_\mu := \mathbf{V}_\mu^T \mathbf{c}_{|\mu} \in \mathbb{R}^{d+1}, \mu \in T_y \rightarrow O(n)$  Operations on every level of  $T_y$

Computational Effort  $O((d+1)n \log_2 n)$ , Memory Required  $O((d+1)n \log_2 n)$

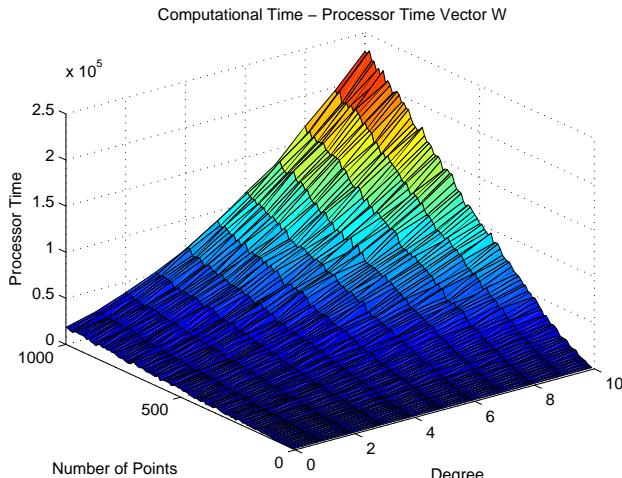


Fig. 334

In file changeable.c -

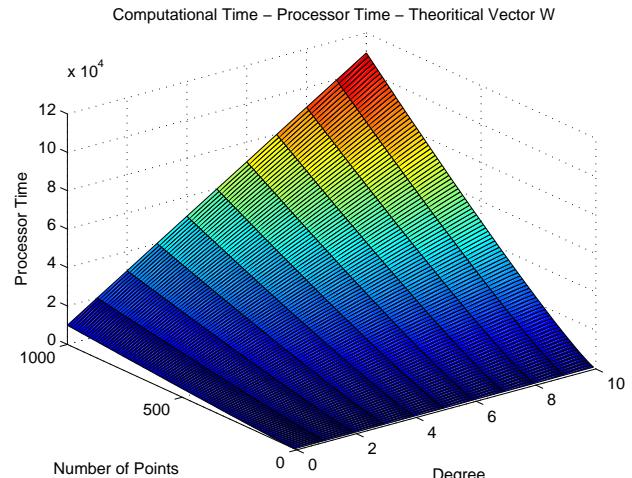


Fig. 335

a) put the admissibility condition in admissible\_ClusterPair

b) put kernel function in kernel\_function

c) put NONE as return values in get\_iOperation

d) put 100 as return values in get\_iNumber of Times

Time.txt

Time\_Vector\_W.eps, Time\_Theoretical\_Vector\_W.eps,

Time\_Theoretical\_Chebyshev.eps, Time\_Far.eps,

Time\_Theoretical\_Near.eps, Time\_All.eps,

Choice - 2, NumberofPts - 10 1000 10, Start of 1 - 0, End of 1 - 1,

Degree - 0 10 1, Admissibility Coefficient - 0.5 0.5 1.0

Output File from C++:

OutputFile from MATLAB :

Time\_Chebyshev.eps,

Time\_Theoretical\_Far.eps, Time\_Near.eps,

Time\_Theoretical\_All.eps

1) Run main.cpp, input as follows :  
Start of 2 - 0, End of 2 - 1

2) Run time\_mesh.m and time\_mesh\_theoretical from Matlab

②: Compute  $X_{\sigma,\mu}$  as in (10.4.1),  $\sigma \in T_x$ ,  $\mu \in T_y$ ,  $(\sigma, \mu) \in P^{\text{far}}$ .

Uniform distribution of points  $\max\{\#\{\mu \in T_y : (\sigma, \mu) \in P^{\text{far}}\}, \sigma \in T_x\} = O(1)$

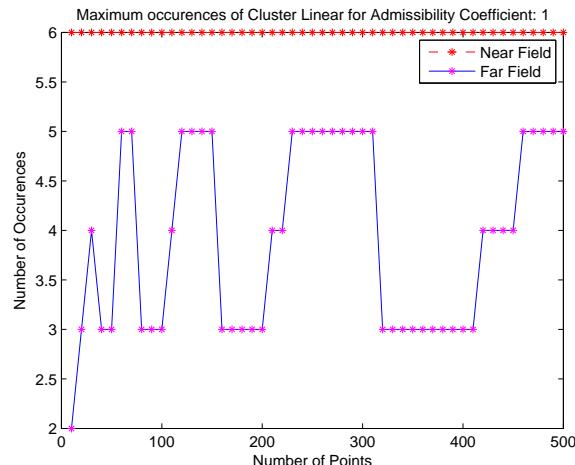


Fig. 336

rti

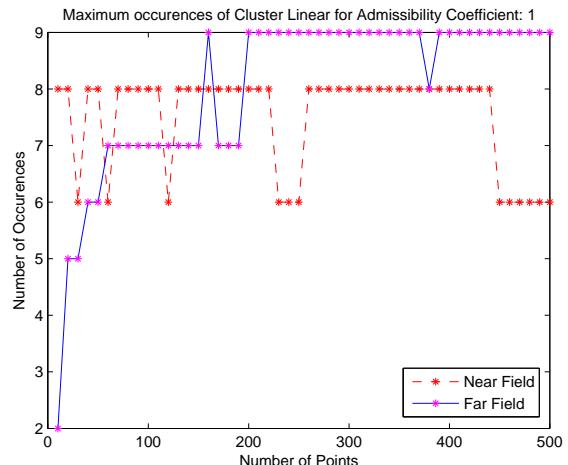


Fig. 337

$x = (0:1/n:(1-1/n)), \eta = 1$   
In file changeable.c

- a) put the admissibility condition in admissible\_ClusterPair
- b) put kernel function in kernel\_function
- c) put **NONE** as return values in get\_iOperation
  - 1) Run main.cpp, input as follows  
Choice - 2, NumberofPts - 10 500 10,  
Start of 1 - 0, End of 1 - 1, Start of 2 - 0, End of 2 - 1  
Degree - 0 0 1, Admissibility Coefficient - 1 1 1  
Output File from C++: Count.txt  
OutputFile from MATLAB : MaxCluster\_1.eps
  - 2) Run count\_indiv.m from Matlab, arguments:  
0 (argument here is for specifying the index of Admissibility Coefficient.)

Uniform distribution of  $x_i, y_j$ : for every  $\sigma \in T_x$

$$\#\{\mu \in T_y : (\sigma, \mu) \in P^{\text{far}}\} = O(1),$$

each cluster contributes only a small number of partitions of rectangle

$x = \sqrt{0:1/n:(1-1/n)}, \eta = 1$   
In file changeable.c

- a) put the admissibility condition in admissible\_ClusterPair
- b) put kernel function in kernel\_function
- c) put **SQRT** as return values in get\_iOperation
  - 1) Run main.cpp, input as follows  
Choice - 2, NumberofPts - 10 500 10,  
Start of 1 - 0, End of 1 - 1, Start of 2 - 0, End of 2 - 1  
Degree - 0 0 1, Admissibility Coefficient - 1 1 1  
Output File from C++: Count.txt  
OutputFile from MATLAB : MaxCluster\_1.eps
  - 2) Run count\_indiv.m from Matlab, arguments:  
0 (argument here is for specifying the index of Admissibility Coefficient.)

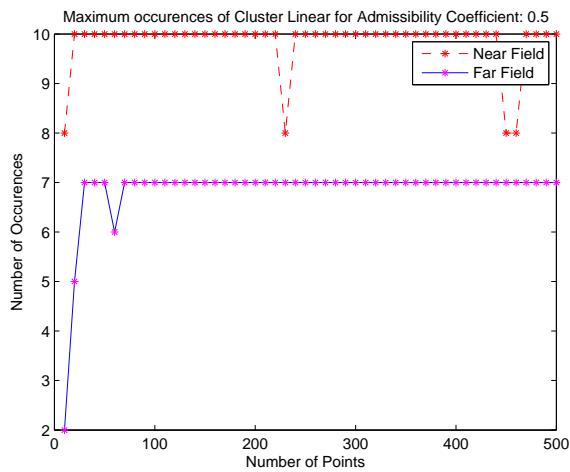
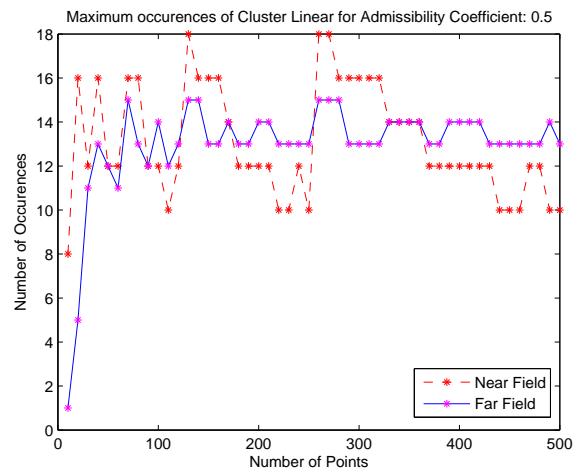


Fig. 338

$x = (0:1/n:(1-1/n));, \eta = 0.5$   
In file changeable.c

- a) put the admissibility condition in admissible\_ClusterPair
- b) put kernel function in kernel\_function
- c) put **NONE** as return values in get\_iOperation
  - 1) Run main.cpp, input as follows  
Choice - 2, NumberofPts - 10 500 10,  
Start of 1 - 0, End of 1 - 1, Start of 2 - 0, End of 2 - 1  
Degree - 0 0 1, Admissibility Coefficient - 0.5 0.5 1  
Output File from C++: Count.txt
  - OutputFile from MATLAB : MaxCluster\_0.5.eps
  - 2) Run count\_indiv.m from Matlab, arguments:  
0 (argument here is for specifying the index of Admissibility Coefficient.)



$x = \text{sqrt}(0:1/n:(1-1/n));, \eta = 0.5$   
In file changeable.c

- a) put the admissibility condition in admissible\_ClusterPair
- b) put kernel function in kernel\_function
- c) put **SQRT** as return values in get\_iOperation
  - 1) Run main.cpp, input as follows  
Choice - 2, NumberofPts - 10 500 10,  
Start of 1 - 0, End of 1 - 1, Start of 2 - 0, End of 2 - 1  
Degree - 0 0 1, Admissibility Coefficient - 0.5 0.5 1  
Output File from C++: Count.txt
  - OutputFile from MATLAB : MaxCluster\_0.5.eps
  - 2) Run count\_indiv.m from Matlab, arguments:  
0 (argument here is for specifying the index of Admissibility Coefficient.)

Computational Effort for step ②:  $O(n \log_2 n(d+1)^2)$ , Memory Required  $O(n \log_2 n(d+1)^2)$

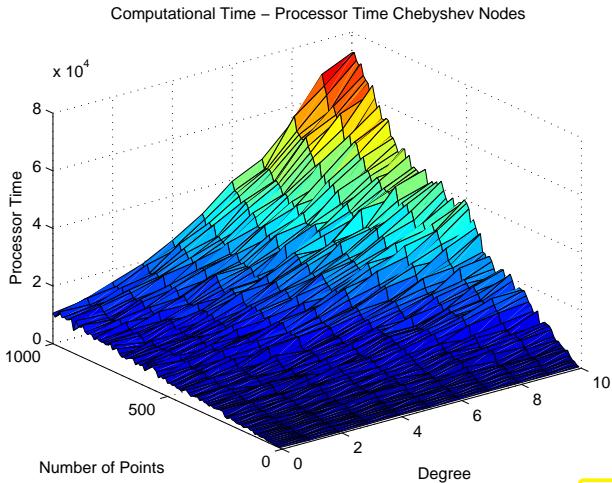


Fig. 340

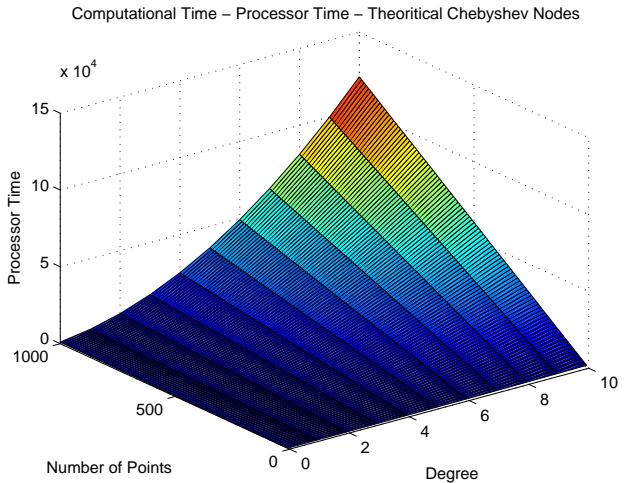


Fig. 341

③: Far field analysis  $\hat{=}$  Sum

$$\sum_{\sigma \in T_x} \sum_{\substack{\mu \in T_y \\ x_i \in \sigma \\ (\sigma, \mu) \in P_{\text{far}}}} \mathbf{V}_\sigma \mathbf{X}_{\sigma, \mu} \mathbf{w}_\mu$$

**for each**  $\sigma \in T_x$  { Compute  $\mathbf{V}_\sigma$ ;  $\mathbf{s} := 0$   
**for each**  $\mu \in T_y$ ,  $(\sigma, \mu) \in P_{\text{far}}$  {  $\mathbf{s} \leftarrow \mathbf{s} + \mathbf{X}_{\sigma, \mu} \mathbf{w}_\mu$  }  
 $\mathbf{f}_{|\sigma} \leftarrow \mathbf{f}_{|\sigma} + \mathbf{V}_\sigma \mathbf{s}$  }

Computational Effort  $O(n \log_2 n(d+1)^2)$ , Memory Required  $O(n(d+1))$

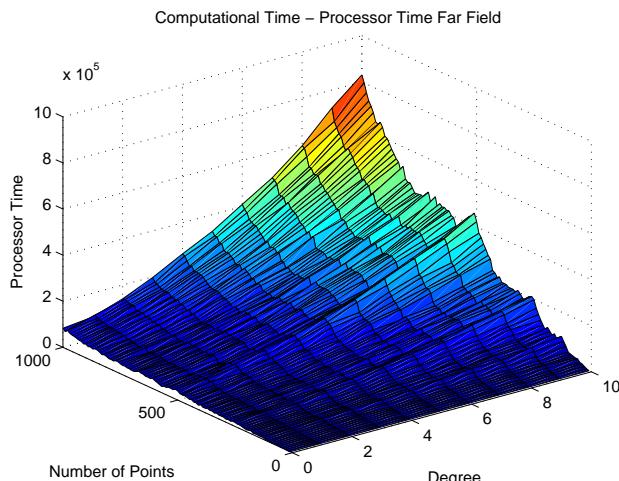


Fig. 342

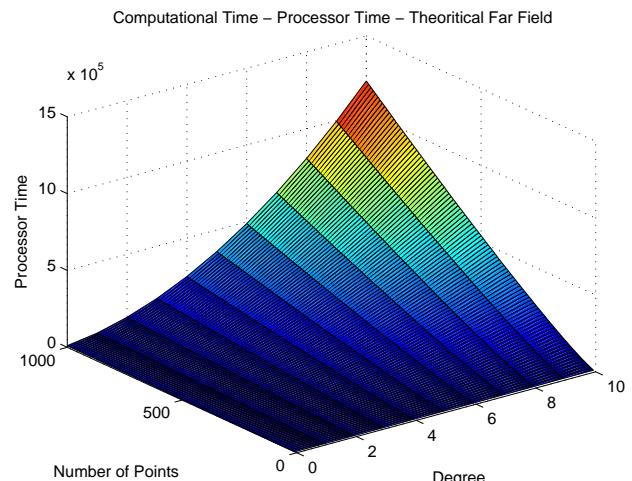


Fig. 343

#### ④: Near field Computation

```
for each  $(\sigma, \mu) \in P^{\text{near}}$  {   for each  $i: x_i \in \sigma$  {    $\mathbf{f}_i \leftarrow \mathbf{f}_i + \sum_{j: y_j \in \mu} G(x_i, y_j) c_j$  } }
```

Computational Effort  $O(n)$

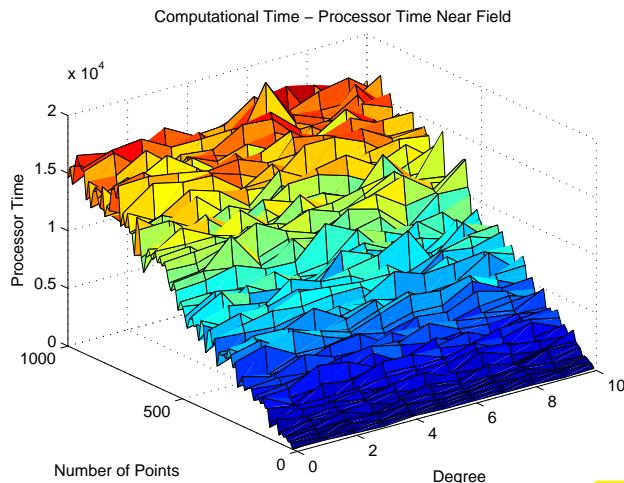


Fig. 344

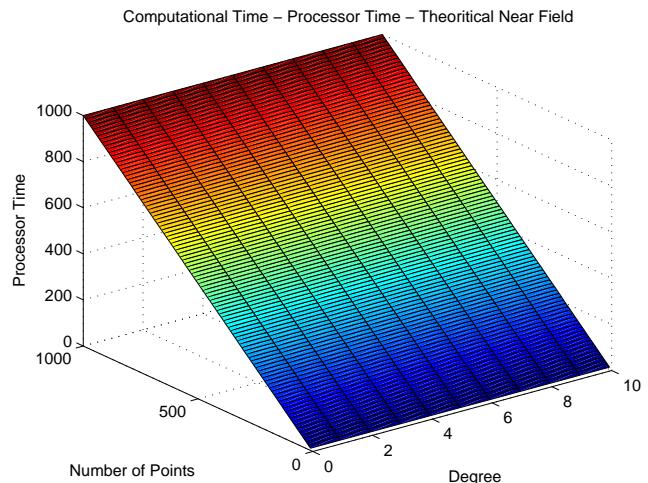


Fig. 345

Total Computational Effort/Total Memory Required  $O((d+1)^2 n \log_2 n)$

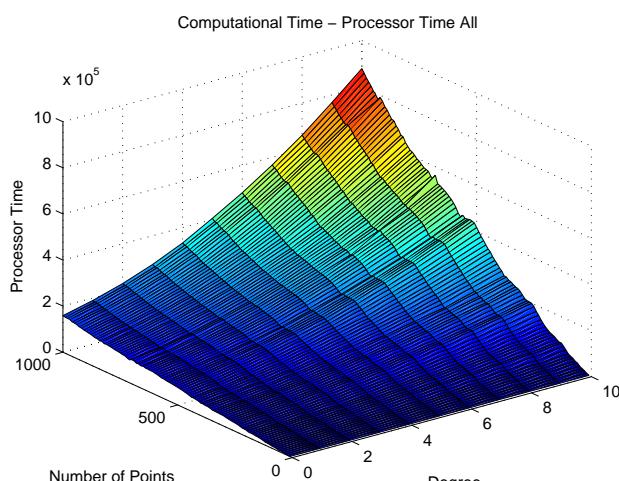


Fig. 346

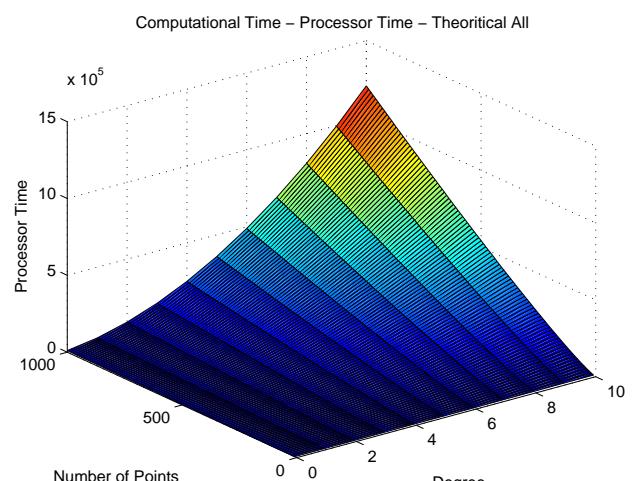


Fig. 347

**Remark 10.4.5 (Convergence of cluster approximation)**

if  $G(x, y)$  analytic ( $\rightarrow$  Def. 4.1.55) in  $\{(x, y) : x \neq y\}$  then

Cluster approximation inherits exponential convergence from Chebyshev interpolation , ??

### Example 10.4.6 (Convergence of clustering approximation with collocation matrix)

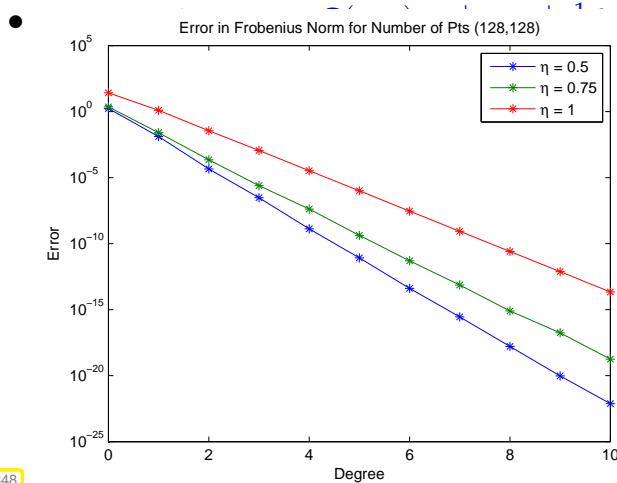


Fig. 348

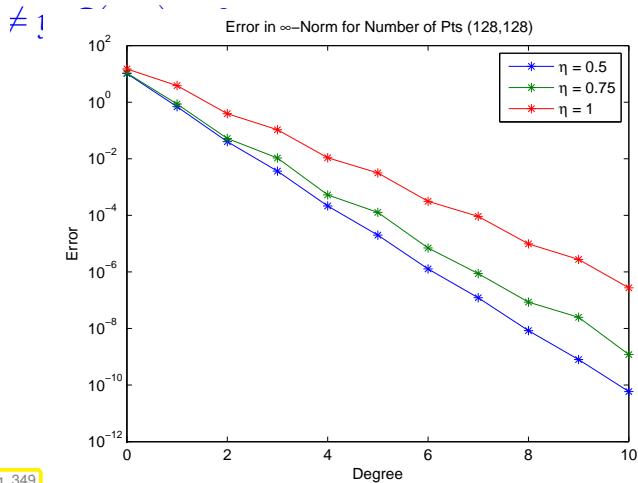


Fig. 349

In file changeable.c

a) put the admissibility condition in admissible\_ClusterPair

b) put kernel function in kernel\_function

c) put NONE as return values in get\_iOperation

1) Run main.cpp, input as follows

Choice - 2, NumberofPts - 128 128 1,

Start of 1 - 0, End of 1 - 1, Start of 2 - 0, End of 2 - 1

Degree - 0 10 1, Admissibility Coefficient - 0.5 1.0 0.25

Output File from C++: Error.txt

OutputFile from MATLAB : Error\_in\_Frobenius\_Norm\_128\_128.eps

2) Run error\_plot.m from Matlab, arguments:

0 (argument here is for specifying the index of Number of Pts.)

- $x = \text{sqrt}(0:1/128:1)$ ;  $G(x, y) = |x - y|^{-1}$  for  $x \neq y$ ,  $G(x, x) = 0$ .

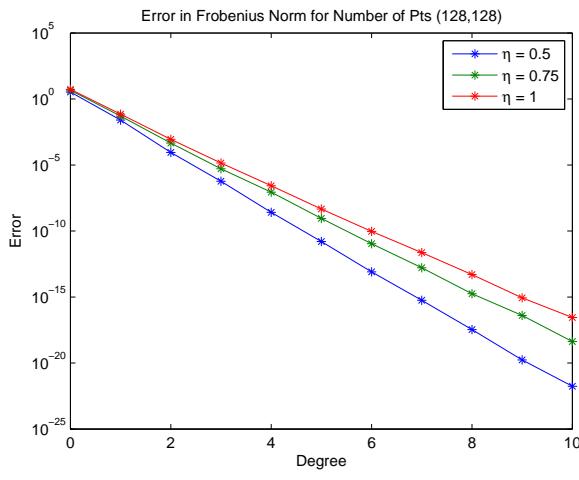


Fig. 350

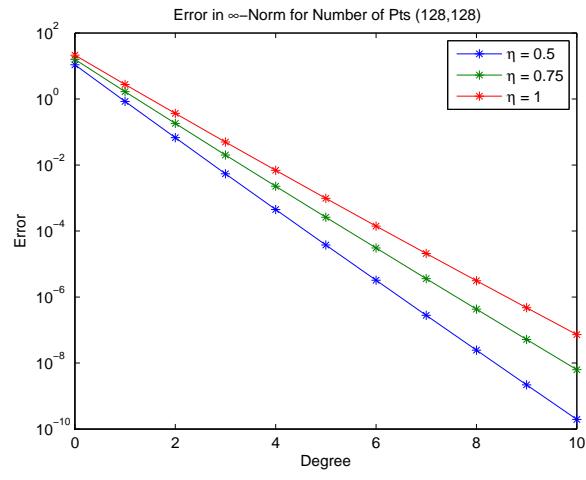


Fig. 351

In file changeable.c

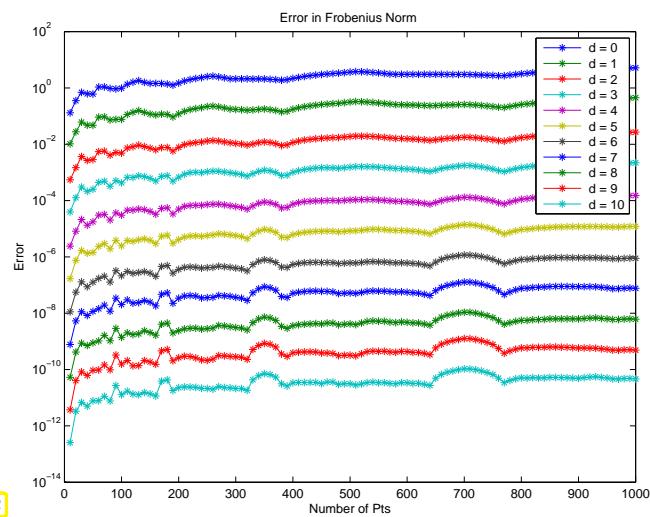
- put the admissibility condition in admissible\_ClusterPair
- put kernel function in kernel\_function
- put SQRT as return values in get\_iOperation

- Run main.cpp, input as follows  
Choice - 2, NumberofPts - 128 128 1,  
Start of 1 - 0, End of 1 - 1, Start of 2 - 0, End of 2 - 1  
Degree - 0 10 1, Admissibility Coefficient - 0.5 1.0 0.25  
Output File from C++: Error.txt

OutputFile from MATLAB : Error\_in\_Frobenius\_Norm\_128\_128.eps

- Run error\_plot.m from Matlab, arguments:  
0 (argument here is for specifying the index of Number of Pts.)

- Scaled Frobenius Error v/s Number of Points



In file changeable.c

- put the admissibility condition in admissible\_ClusterPair
- put kernel function in kernel\_function
- put NONE as return values in get\_iOperation

- Run main.cpp, input as follows  
Choice - 2, NumberofPts - 10 1000 10,  
Start of 1 - 0, End of 1 - 1, Start of 2 - 0, End of 2 - 1  
Degree - 0 10 1, Admissibility Coefficient - 0.5 0.5 1.0  
Output File from C++: Error.txt

OutputFile from MATLAB : Error\_in\_Frobenius\_Norm\_Degree.eps

- Run error\_plot\_Degree.m from Matlab

### Example 10.4.7 (Analysis of trigonometric polynomials)

[22]

Given:  $\{t_0, \dots, t_{n-1}\} \subset [0, 1[, \alpha_{-m+1}, \dots, \alpha_m \in \mathbb{C}$ ,  $n = 2m$ ,  $m \in \mathbb{N}$ , compute

$$c_k := p(t_k), \quad j = 0, \dots, n-1 \quad \text{for} \quad p(t) := \underbrace{\sum_{j=-m+1}^m \alpha_j e^{-2\pi i jt}}_{\text{Trigonometric Polynomial}}.$$

Discrete Fourier transformation (DFT)  $\rightarrow$  Section 9.2:

$$f_l := \sum_{j=-m+1}^m \alpha_j e^{-\frac{2\pi i}{n} jl} \quad \text{Lemma 9.2.9} \Leftrightarrow \alpha_j = \frac{1}{n} \sum_{l=-m+1}^m f_l e^{\frac{2\pi i}{n} lj}$$

FFT:  $f_l$ ,  $l = -m+1, \dots, m$ , calculated with computational effort  $O(n \log_2 n)$

$$\begin{aligned} c_k &= \sum_{j=-m+1}^m \alpha_j e^{2\pi i jt_k} = \sum_{j=-m+1}^m \frac{1}{n} \left( \sum_{l=-m+1}^m f_l e^{\frac{2\pi i}{n} lj} \right) e^{-2\pi i jt_k} \\ &= \frac{1}{n} \sum_{l=-m+1}^m f_l \sum_{j=-m+1}^m e^{-2\pi i j(t_k - l/n)} \end{aligned}$$

$$\begin{aligned}
&= \frac{1}{n} \sum_{l=-m+1}^m f_l e^{-2\pi i(t_k - l/n)(-m+1)} \frac{1 - e^{-2\pi i n(t_k - l/n)}}{1 - e^{-2\pi i(t_k - l/n)}} \\
&= \frac{1}{n} \sum_{l=-m+1}^m f_l e^{-\pi i t_k} \sin(\pi n t_k) \frac{1}{\sin(\pi(t_k - l/n))} (-1)^l e^{-\pi i l/n}.
\end{aligned}$$

$$\mathbf{c} = \text{diag}\left(\frac{\sin(\pi n t_k)}{e^{\pi i t_k}}\right)_{k=-m+1}^m \mathbf{M} \text{diag}\left((-1)^l e^{-\pi i l/n}\right)_{l=-m+1, \dots, m} \mathbf{f}.$$

in accordance with collocation matrix (10.1.1)

$$\mathbf{M} := \left[ \frac{1}{\sin(\pi(t_k - l/n))} \right]_{\substack{k=-m+1, \dots, m \\ l=-m+1, \dots, m}} \in \mathbb{R}^{2n, 2n}.$$

Approximative analysis of Clustering algorithms ! → USFFT

#### Remark 10.4.8.

Clustering approximation example for **fast approximative** implementation of algorithms of numerical linear algebra (→ ??) → Trend in numerical linear algebra ?

Problem	Exact/direct Method	Approximate/iterative Method
Linear equation systems	Gaussian Elimination	CG-like iterative solvers → ??
Eigenvalue problem	Transformation methods	Krylov-Subspace method → ??
Collocations matrix × Vector	BLAS (SAXPY)	Clustering techniques

With Clusteringapproximation related, or generalizations thereof:

- $\mathcal{H}$ -Matrix-Techniques [9]
- $\mathcal{H}^2$ -Matrix-Techniques [36]
- Multipol-Methods [67, 31] → further “Millennium algorithm” <http://orion.math.iastate.edu/bu>

# Chapter 11

## Numerical Integration – Single Step Methods

### Contents

---

<b>11.1</b>	<b>Initial value problems (IVP) for ODEs</b>	<b>601</b>
11.1.1	Modeling with ordinary differential equations: Examples	602
11.1.2	Theory of initial value problems	606
11.1.3	Evolution operators	610
<b>11.2</b>	<b>Introduction: Polygonal Approximation Methods</b>	<b>612</b>
11.2.1	Explicit Euler method	613
11.2.2	Implicit Euler method	615
11.2.3	Implicit midpoint method	616
<b>11.3</b>	<b>General single step methods</b>	<b>617</b>
11.3.1	Definition	617
11.3.2	Convergence of single step methods	620
<b>11.4</b>	<b>Explicit Runge-Kutta Methods</b>	<b>626</b>
<b>11.5</b>	<b>Adaptive Stepsize Control</b>	<b>633</b>

---

### 11.1 Initial value problems (IVP) for ODEs

Acronym:

ODE = ordinary differential equation

#### (11.1.1) Terminology and notations related to ordinary differential equations

In our parlance, a (first-order) ordinary differential equation (ODE) is an equation of the form

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) , \quad (11.1.2)$$

with

- ☞ a (continuous) right hand side function (r.h.s)  $f : I \times D \rightarrow \mathbb{R}^d$  of time  $t \in \mathbb{R}$  and state  $\mathbf{y} \in \mathbb{R}^d$
- ☞ defined on a (finite) time interval  $I \subset \mathbb{R}$ , and state space  $D \subset \mathbb{R}^d$ .

In the context of mathematical modeling the state vector  $\mathbf{y} \in \mathbb{R}^d$  is supposed to provide a complete (in

the sense of the model) description of a system. Examples will be provided below.

For  $d > 1$   $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$  can be viewed as a **system of ordinary differential equations**:

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \iff \begin{bmatrix} \dot{y}_1 \\ \vdots \\ \dot{y}_d \end{bmatrix} = \begin{bmatrix} f_1(t, y_1, \dots, y_d) \\ \vdots \\ f_d(t, y_1, \dots, y_d) \end{bmatrix}.$$

☞ Notation (Newton): dot  $\cdot$   $\hat{=}$  (total) derivative with respect to time  $t$

#### Definition 11.1.3. Solution of an ordinary differential equation

A **solution** of the ODE  $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$  with continuous right hand side function  $\mathbf{f}$  is a continuously differentiable **function** "of time  $t"$   $\mathbf{y} : J \subset I \rightarrow D$ , defined on an **open** interval  $J$ , for which  $\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t))$  holds for all  $t \in J$ .

A solution describes a continuous **trajectory** in state space, a one-parameter family of states, parameterized by time.

It goes without saying that smoothness of the right hand side function  $\mathbf{f}$  is inherited by solutions of the ODE:

#### Lemma 11.1.4. Smoothness of solutions of ODEs

Let  $\mathbf{y} : I \subset \mathbb{R} \rightarrow D$  be a solution of the ODE  $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$  on the time interval  $I$ .

If  $\mathbf{f} : I \times D \rightarrow \mathbb{R}^d$  is  $r$ -times continuously differentiable with respect to both arguments,  $r \in \mathbb{N}_0$ , then the trajectory  $t \mapsto \mathbf{y}(t)$  is  $r+1$ -times continuously differentiable in the interior of  $I$ .



**Supplementary reading.** Some grasp of the meaning and theory of ordinary differential equations (ODEs) is indispensable for understanding the construction and properties of numerical methods. Relevant information can be found in [77, Sect. 5.6, 5.7, 6.5].

Books dedicated to numerical methods for ordinary differential equations:

- [18] excellent textbook, but geared to the needs of students of mathematics.
- [39] and [40] : the standard reference.
- [38]: wonderful book conveying deep insight, with emphasis on mathematical concepts.

### 11.1.1 Modeling with ordinary differential equations: Examples

#### Example 11.1.5 (Growth with limited resources [5, Sect. 1.1], [42, Ch. 60])

This is an example from **population dynamics** with a one-dimensional state space  $D = \mathbb{R}_0^+$ ,  $d = 1$ :

$y : [0, T] \mapsto \mathbb{R}$ : bacterial population density as a function of time

ODE-based model: autonomous **logistic differential equations** [77, Ex. 5.7.2]

$$\dot{y} = f(y) := (\alpha - \beta y) y \quad (11.1.6)$$

- \*  $y \doteq$  population density,  $[y] = \frac{1}{\text{m}^2}$
- \* growth rate  $\alpha - \beta y$  with growth coefficients  $\alpha, \beta > 0$ ,  $[\alpha] = \frac{1}{\text{s}}$ ,  $[\beta] = \frac{\text{m}^2}{\text{s}}$ : decreases due to more fierce competition as population density increases.

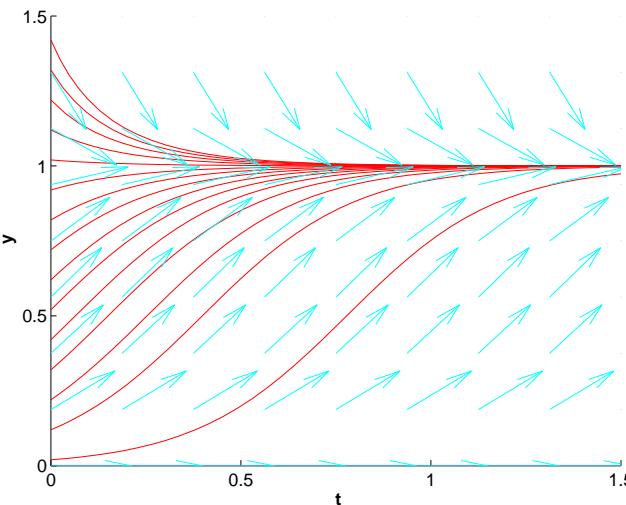


Fig. 353

Solution for different  $y(0)$  ( $\alpha, \beta = 5$ )

Note that by fixing the initial value  $y(0)$  we can single out a *unique* representative from the family of solutions. This will turn out to be a general principle, see Section 11.1.2.

### Definition 11.1.8. Autonomous ODE

An ODE of the form  $\dot{y} = f(y)$ , that is, with a right hand side function that does not depend on time, but only on state, is called **autonomous**.

For an autonomous ODE the right hand side function defines a vector field (“velocity field”)  $y \mapsto f(y)$  on state space.

### Example 11.1.9 (Predator-prey model [5, Sect. 1.1],[38, Sect. 1.1.1],[42, Ch. 60], [15, Ex. 11.3])

Predators and prey coexist in an ecosystem. Without predators the population of prey would be governed by a simple exponential growth law. However, the growth rate of prey will decrease with increasing numbers of predators and, eventually, become negative. Similar considerations apply to the predator population and lead to an ODE model.

ODE-based model: autonomous **Lotka-Volterra ODE**:

$$\begin{aligned} \dot{u} &= (\alpha - \beta v)u \\ \dot{v} &= (\delta u - \gamma)v \end{aligned} \Leftrightarrow \dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad \text{with} \quad \mathbf{y} = \begin{bmatrix} u \\ v \end{bmatrix}, \quad \mathbf{f}(\mathbf{y}) = \begin{bmatrix} (\alpha - \beta v)u \\ (\delta u - \gamma)v \end{bmatrix}, \quad (11.1.10)$$

with positive model parameters  $\alpha, \beta, \gamma, \delta > 0$ .

population densities:

$u(t) \rightarrow$  density of prey at time  $t$ ,

$v(t) \rightarrow$  density of predators at time  $t$

Right hand side vector field  $\mathbf{f}$  for Lotka-Volterra ODE

▷

Solution curves are trajectories of particles carried along by velocity field  $\mathbf{f}$ .

(Parameter values for Fig. 354:  $\alpha = 2, \beta = 1, \delta = 1, \gamma = 1$ .)

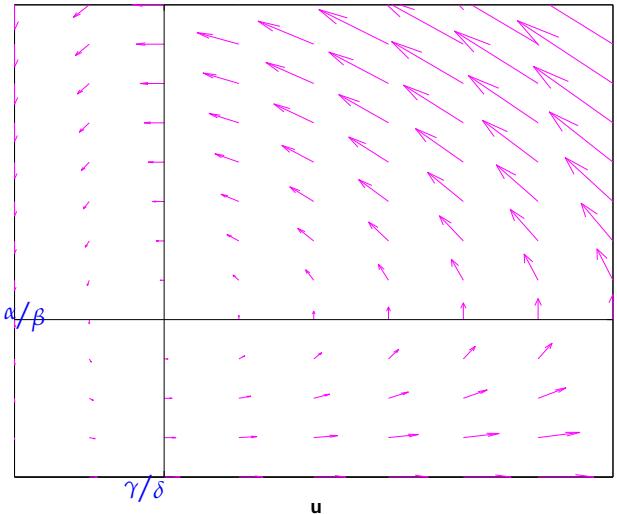


Fig. 354

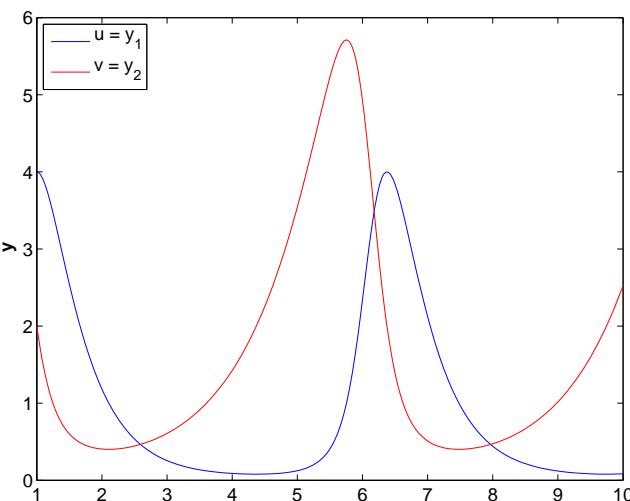


Fig. 355

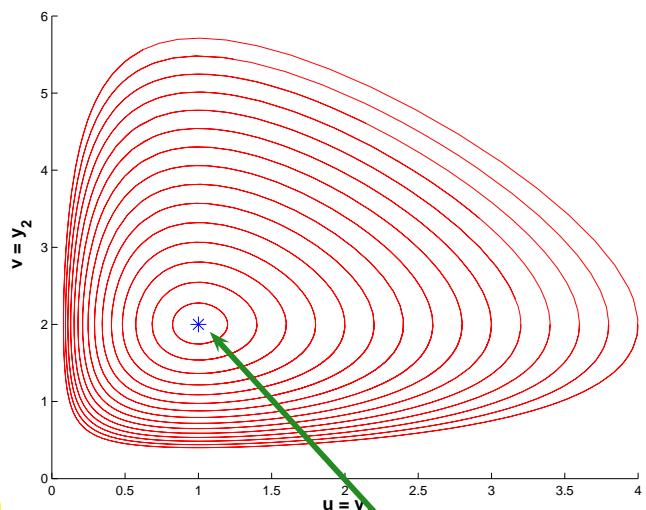


Fig. 356

Solution  $\begin{bmatrix} u(t) \\ v(t) \end{bmatrix}$  for  $\mathbf{y}_0 := \begin{bmatrix} u(0) \\ v(0) \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$

Parameter values for Fig. 356, 355:  $\alpha = 1, \beta = 1, \delta = 1, \gamma = 2$

Solution curves for (11.1.10)

stationary point

### Example 11.1.11 (Heartbeat model → [17, p. 655])

This example centers around a *phenomenological model* from physiology.

State of heart described by quantities:  $l = l(t) \hat{=} \text{length of muscle fiber}$   
 $p = p(t) \hat{=} \text{electro-chemical potential}$

Phenomenological model:  $\begin{aligned} \dot{l} &= -(l^3 - \alpha l + p), \\ \dot{p} &= \beta l, \end{aligned} \quad (11.1.12)$

with parameters:  $\alpha \hat{=} \text{pre-tension of muscle fiber}$

$\beta \hat{=} \text{(phenomenological) feedback parameter}$

This is the so-called Zeeman model: it is a phenomenological model entirely based on macroscopic observations without relying on knowledge about the underlying molecular mechanisms.

Vector fields and solutions for different choices of parameters:

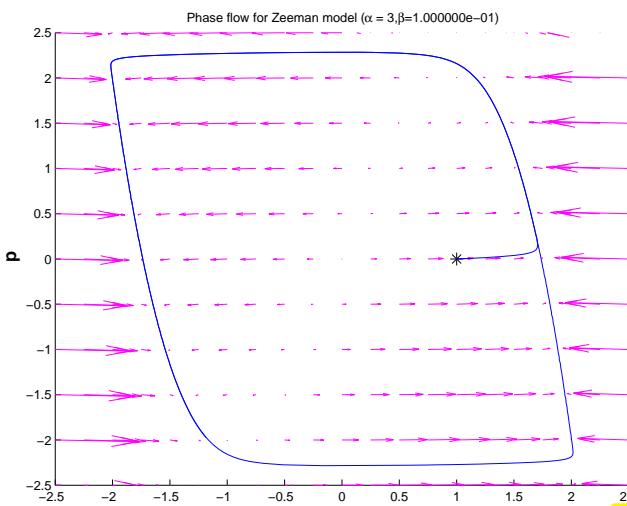


Fig. 357

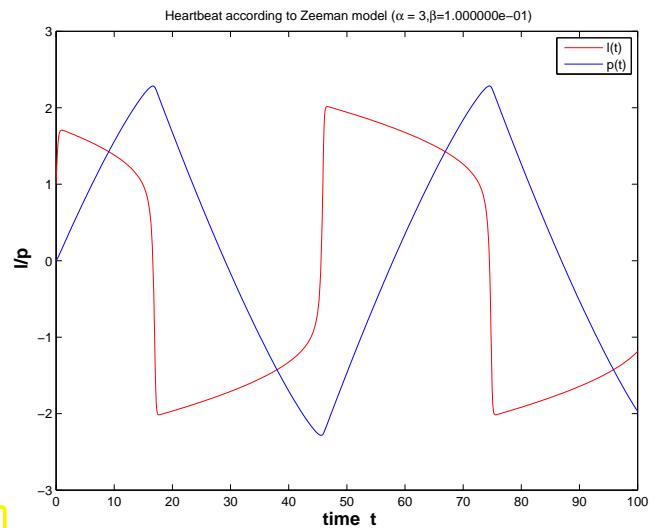


Fig. 358

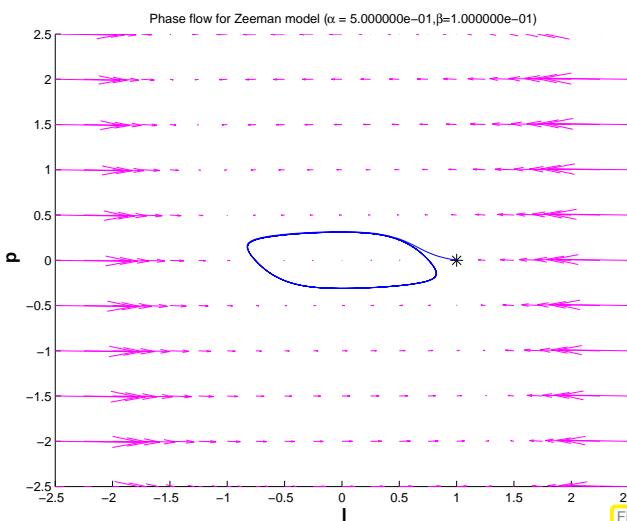


Fig. 359

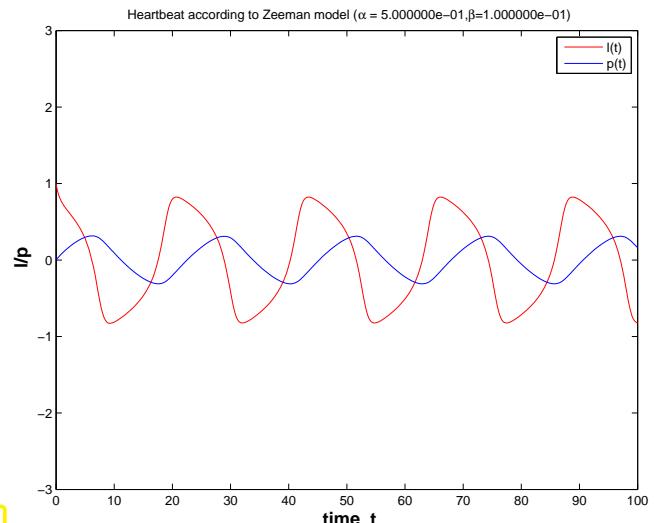


Fig. 360

Observation:  $\alpha \ll 1 \rightarrow$  ventricular fibrillation, a life-threatening condition.

### Example 11.1.13 (Transient circuit simulation [42, Ch. 64])

In Chapter 1 and Chapter 2 we discussed circuit analysis as a source of linear and non-linear systems of equations, see Ex. 1.6.3 and Ex. 2.0.1. In the former example we admitted time-dependent currents and potentials, but dependence on time was confined to be “sinusoidal”. This enabled us to switch to frequency domain, see (1.6.6), which gave us a complex linear system of equations for the complex nodal potentials. Yet, this trick is only possible for *linear* circuits. In the general case, circuits have to be modelled by ODEs connecting time-dependent potentials and currents. This will be briefly explained now.

The approach is transient nodal analysis, cf.  
Ex. 1.6.3, based on the Kirchhoff current law, cf.

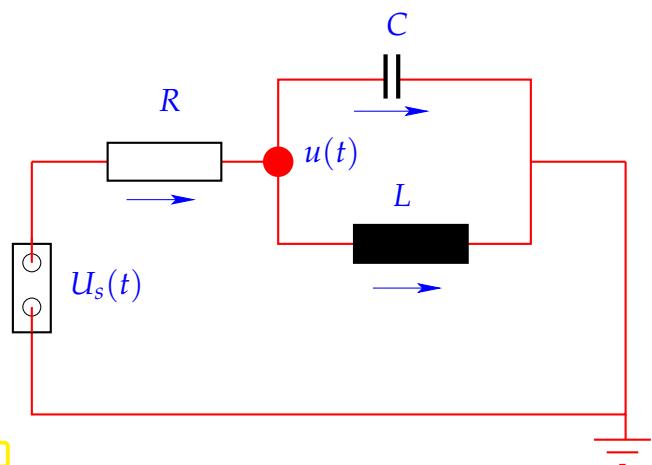
$$i_R(t) - i_L(t) - i_C(t) = 0. \quad (11.1.14)$$

Transient constitutive relations for basic linear circuit elements:

resistor:  $i_R(t) = R^{-1}u_R(t), \quad (11.1.15)$

capacitor:  $i_C(t) = C \frac{du_C}{dt}(t), \quad (11.1.16)$

coil:  $u_L(t) = L \frac{di_L}{dt}(t). \quad (11.1.17)$   
Fig. 361



Given: source voltage  $U_s(t)$

To apply nodal analysis to the circuit of Fig. 361 we differentiate (11.1.14) w.r.t.  $t$

$$\frac{di_R}{dt}(t) - \frac{di_L}{dt}(t) - \frac{di_C}{dt}(t) = 0,$$

and plug in the above constitutive relations for circuit elements:

$$\Rightarrow R^{-1} \frac{du_R}{dt}(t) - L^{-1}u_L(t) - C \frac{d^2u_C}{dt^2}(t) = 0.$$

We continue following the policy of nodal analysis and express all voltages by potential differences between nodes of the circuit.

$$u_R(t) = U_s(t) - u(t), \quad u_C(t) = u(t) - 0, \quad u_L(t) = u(t) - 0.$$

For this simple circuit there is only one node with unknown potential, see Fig. 361. Its time-dependent potential will be denoted by  $u(t)$  and this is the unknown of the model, a function of time obeying the ordinary differential equation

$$R^{-1}(\dot{U}_s(t) - \dot{u}(t)) - L^{-1}u(t) - C \frac{d^2u}{dt^2}(t) = 0.$$

► This is an autonomous **2nd-order** ordinary differential equation:

$$C\ddot{u} + R^{-1}\dot{u} + L^{-1}u = R^{-1}\dot{U}_s. \quad (11.1.18)$$

The attribute “2nd-order” refers to the occurrence of a second derivative with respect to time.

## 11.1.2 Theory of initial value problems



*Supplementary reading.* [63, Sect. 11.1], [15, Sect. 11.3]

### (11.1.19) Initial value problems

We start with an abstract mathematical description

A generic **Initial value problem** (IVP) for a **first-order ordinary differential equation** (ODE) ( $\rightarrow$  [77, Sect. 5.6], [15, Sect. 11.1]) can be stated as: find a function  $\mathbf{y} : I \rightarrow D$  that satisfies, cf. Def. 11.1.3,

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0 . \quad (11.1.20)$$

- $\mathbf{f} : I \times D \mapsto \mathbb{R}^d \hat{=} \text{right hand side}$  (r.h.s.) ( $d \in \mathbb{N}$ ),
- $I \subset \mathbb{R} \hat{=} \text{(time)interval} \leftrightarrow \text{"time variable" } t$ ,
- $D \subset \mathbb{R}^d \hat{=} \text{state space/phase space} \leftrightarrow \text{"state variable" } \mathbf{y}$ ,
- $\Omega := I \times D \hat{=} \text{extended state space}$  (of tuples  $(t, \mathbf{y})$ ),
- $t_0 \in I \hat{=} \text{initial time}, \quad \mathbf{y}_0 \in D \hat{=} \text{initial state} \Rightarrow \text{initial conditions.}$

### (11.1.21) IVPs for autonomous ODEs

Recall Def. 11.1.8: For an autonomous ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ , that is the right hand side  $\mathbf{f}$  does not depend on time  $t$ .

Hence, for autonomous ODEs we have  $I = \mathbb{R}$  and the right hand side function  $\mathbf{y} \mapsto \mathbf{f}(\mathbf{y})$  can be regarded as a stationary vector field (velocity field), see Fig. 354 or Fig. 357.

An important observation: If  $t \mapsto \mathbf{y}(t)$  is a solution of an autonomous ODE, then, for any  $\tau \in \mathbb{R}$ , also the shifted function  $t \mapsto \mathbf{y}(t - \tau)$  is a solution.

$\Rightarrow$  For initial value problems for autonomous ODEs the initial time is irrelevant and therefore we can always make the “canonical choice  $t_0 = 0$ ”.

Autonomous ODEs naturally arise when modeling **time-invariant** systems or phenomena. All examples for Section 11.1.1 belong to this class.

### (11.1.22) Autonomization: Conversion into autonomous ODE

In fact, autonomous ODEs already represent the general case, because every ODE can be converted into an autonomous one:

Idea: include time as an extra  $d + 1$ -st component of an extended state vector.

This solution component has to grow linearly  $\Leftrightarrow$  temporal derivative  $= 1$

$$\mathbf{z}(t) := \begin{bmatrix} \mathbf{y}(t) \\ t \end{bmatrix} = \begin{bmatrix} \mathbf{z}' \\ z_{d+1} \end{bmatrix} : \quad \dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \Leftrightarrow \dot{\mathbf{z}} = \mathbf{g}(\mathbf{z}), \quad \mathbf{g}(\mathbf{z}) := \begin{bmatrix} \mathbf{f}(z_{d+1}, \mathbf{z}') \\ 1 \end{bmatrix} .$$

$\Rightarrow$  We restrict ourselves to autonomous ODEs in the remainder of this chapter.

### Remark 11.1.23 (From higher order ODEs to first order systems [15, Sect. 11.2])

An ordinary differential equation of **order**  $n \in \mathbb{N}$  has the form

$$\mathbf{y}^{(n)} = \mathbf{f}(t, \mathbf{y}, \dot{\mathbf{y}}, \dots, \mathbf{y}^{(n-1)}) . \quad (11.1.24)$$

Notation: superscript  $^{(n)} \doteq n\text{-th temporal derivative } t: \frac{d^n}{dt^n}$

No special treatment of higher order ODEs is necessary, because (11.1.24) can be turned into a 1st-order ODE (a system of size  $nd$ ) by adding all derivatives up to order  $n - 1$  as additional components to the state vector. This extended state vector  $\mathbf{z}(t) \in \mathbb{R}^{nd}$  is defined as

$$\mathbf{z}(t) := \begin{bmatrix} \mathbf{y}(t) \\ \mathbf{y}^{(1)}(t) \\ \vdots \\ \mathbf{y}^{(n-1)}(t) \end{bmatrix} = \begin{bmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \\ \vdots \\ \mathbf{z}_n \end{bmatrix} \in \mathbb{R}^{dn}: \quad (11.1.24) \leftrightarrow \dot{\mathbf{z}} = \mathbf{g}(\mathbf{z}), \quad \mathbf{g}(\mathbf{z}) := \begin{bmatrix} \mathbf{z}_2 \\ \mathbf{z}_3 \\ \vdots \\ \mathbf{z}_n \\ \mathbf{f}(t, \mathbf{z}_1, \dots, \mathbf{z}_n) \end{bmatrix} . \quad (11.1.25)$$

Note that the extended system requires initial values  $\mathbf{y}(t_0), \dot{\mathbf{y}}(t_0), \dots, \mathbf{y}^{(n-1)}(t_0)$ : for ODEs of order  $n \in \mathbb{N}$  well-posed initial value problems need to specify initial values for the first  $n - 1$  derivatives.

Now we review results about existence and uniqueness of solutions of initial value problems for first-order ODEs. These are surprisingly general and do not impose severe constraints on right hand side functions.

#### Definition 11.1.26. Lipschitz continuous function ( $\rightarrow [77, \text{Def. 4.1.4}]$ )

Let  $\Theta := I \times D$ ,  $I \subset \mathbb{R}$  an interval,  $D \subset \mathbb{R}^d$  an open domain. A function  $\mathbf{f} : \Theta \mapsto \mathbb{R}^d$  is **Lipschitz continuous** (in the second argument) on  $\Theta$ , if

$$\exists L > 0: \quad \|\mathbf{f}(t, \mathbf{w}) - \mathbf{f}(t, \mathbf{z})\| \leq L \|\mathbf{w} - \mathbf{z}\| \quad \forall (t, \mathbf{w}), (t, \mathbf{z}) \in \Theta . \quad (11.1.27)$$

#### Definition 11.1.28. Local Lipschitz continuity ( $\rightarrow [77, \text{Def. 4.1.5}]$ )

Let  $\Omega := I \times D$ ,  $I \subset \mathbb{R}$  an interval,  $D \subset \mathbb{R}^d$  an open domain. A function  $\mathbf{f} : \Omega \mapsto \mathbb{R}^d$  is **locally Lipschitz continuous**, if for every  $(t, \mathbf{y}) \in \Omega$  there is a closed box  $B$  with  $(t, \mathbf{y}) \in B$  such that  $\mathbf{f}$  is Lipschitz continuous on  $B$ :

$$\begin{aligned} \forall (t, \mathbf{y}) \in \Omega: \quad & \exists \delta > 0, L > 0: \\ & \|\mathbf{f}(\tau, \mathbf{z}) - \mathbf{f}(\tau, \mathbf{w})\| \leq L \|\mathbf{z} - \mathbf{w}\| \\ & \forall \mathbf{z}, \mathbf{w} \in D: \|\mathbf{z} - \mathbf{y}\| \leq \delta, \|\mathbf{w} - \mathbf{y}\| \leq \delta, \forall \tau \in I: |t - \tau| \leq \delta . \end{aligned} \quad (11.1.29)$$

The property of local Lipschitz continuity means that the function  $(t, \mathbf{y}) \mapsto \mathbf{f}(t, \mathbf{y})$  has “locally finite slope” in  $\mathbf{y}$ .

#### Example 11.1.30 (A function that is not locally Lipschitz continuous)

The meaning of local Lipschitz continuity is best explained by giving an example of a function that fails to possess this property.

Consider the square root function  $t \mapsto \sqrt{t}$  on the *closed* interval  $[0, 1]$ . Its slope in  $t = 0$  is infinite and so it is not locally Lipschitz continuous on  $[0, 1]$ .

However, if we consider the square root on the *open* interval  $]0, 1[$ , then it is locally Lipschitz continuous there.

Notation:  $D_y \mathbf{f} \triangleq$  derivative of  $\mathbf{f}$  w.r.t. state variable, a Jacobian  $\in \mathbb{R}^{d,d}$  as defined in (2.2.11).

The next lemma gives a simple criterion for local Lipschitz continuity, which can be proved by the mean value theorem, *cf.* the proof of Lemma 2.2.12.

### Lemma 11.1.31. Criterion for local Lipschitz continuity

If  $\mathbf{f}$  and  $D_y \mathbf{f}$  are continuous on the extended state space  $\Omega$ , then  $\mathbf{f}$  is locally Lipschitz continuous ( $\rightarrow$  Def. 11.1.28).

### Theorem 11.1.32. Theorem of Peano & Picard-Lindelöf [5, Satz II(7.6)], [77, Satz 6.5.1], [15, Thm. 11.10], [42, Thm. 73.1]

If the right hand side function  $\mathbf{f} : \hat{\Omega} \mapsto \mathbb{R}^d$  is locally Lipschitz continuous ( $\rightarrow$  Def. 11.1.28) then for all initial conditions  $(t_0, \mathbf{y}_0) \in \hat{\Omega}$  the IVP (11.1.20) has a solution  $\mathbf{y} \in C^1(J(t_0, \mathbf{y}_0), \mathbb{R}^d)$  with maximal (temporal) domain of definition  $J(t_0, \mathbf{y}_0) \subset \mathbb{R}$ .

In light of § 11.1.22 and Thm. 11.1.32 henceforth we mainly consider

$$\text{autonomous IVPs: } \dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) , \quad \mathbf{y}(0) = \mathbf{y}_0 , \quad (11.1.33)$$

with locally Lipschitz continuous ( $\rightarrow$  Def. 11.1.28) right hand side  $\mathbf{f}$ .

### (11.1.34) Domain of definition of solutions of IVPs

Solutions of an IVP have an *intrinsic* maximal domain of definition

! domain of definition/domain of existence  $J(t_0, \mathbf{y}_0)$  usually depends on  $(t_0, \mathbf{y}_0)$  !

Terminology: if  $J(t_0, \mathbf{y}_0) = I$   $\rightarrow$  solution  $\mathbf{y} : I \mapsto \mathbb{R}^d$  is **global**.

Notation: for autonomous ODE we always have  $t_0 = 0$ , and therefore we write  $J(\mathbf{y}_0) := J(0, \mathbf{y}_0)$ .

### Example 11.1.35 (Finite-time blow-up)

Let us explain the still mysterious “maximal domain of definition” in statement of Thm. 11.1.32. It is related to the fact that every solution of an initial value problem (11.1.33) has its own largest possible time interval  $J(\mathbf{y}_0) \subset \mathbb{R}$  on which it is defined naturally.

As an example we consider the autonomous scalar ( $d = 1$ ) initial value problem, modeling “explosive growth” with a growth rate increasing linearly with the density:

$$\dot{y} = y^2 , \quad y(0) = y_0 \in \mathbb{R} . \quad (11.1.36)$$

We choose  $I = D = \mathbb{R}$ . Clearly,  $y \mapsto y^2$  is locally Lipschitz-continuous, but only locally! Why not globally?

We find the solutions

$$y(t) = \begin{cases} \frac{1}{y_0^{-1}-t} & , \text{ if } y_0 \neq 0 , \\ 0 & , \text{ if } y_0 = 0 , \end{cases} \quad (11.1.37)$$

with domains of definition

$$J(y_0) = \begin{cases} ]-\infty, y_0^{-1}[ & , \text{ if } y_0 > 0 , \\ \mathbb{R} & , \text{ if } y_0 = 0 , \\ ]y_0^{-1}, \infty[ & , \text{ if } y_0 < 0 . \end{cases}$$

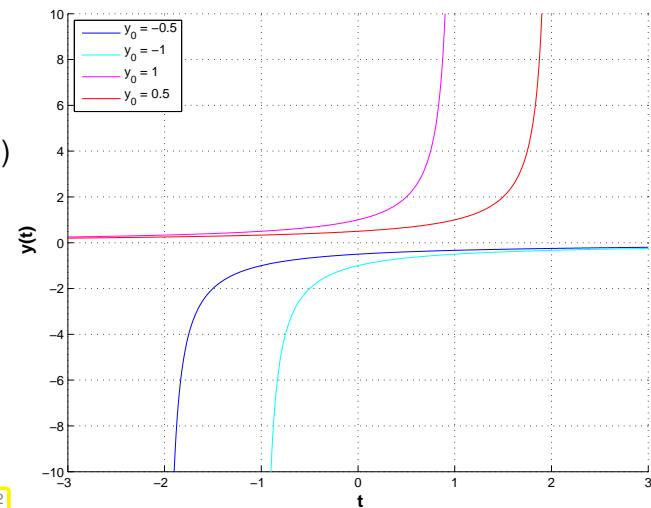


Fig. 362

In this example, for  $y_0 > 0$  the solution experiences a **blow-up** in finite time and ceases to exist afterwards.

### 11.1.3 Evolution operators

For the sake of simplicity we restrict the discussion to autonomous IVPs (11.1.33) with locally Lipschitz continuous right hand side and make the following assumption. A more general treatment is given in [18].

#### Assumption 11.1.38. Global solutions

All solutions of (11.1.33) are global:  $J(\mathbf{y}_0) = \mathbb{R}$  for all  $\mathbf{y}_0 \in D$ .

Now we return to the study of a generic ODE (11.1.2) instead of an IVP (11.1.20). We do this by temporarily changing the perspective: we fix a “time of interest”  $t \in \mathbb{R} \setminus \{0\}$  and follow all trajectories for the duration  $t$ . This induces a mapping of points in state space:

$$\geq \text{mapping} \quad \Phi^t : \begin{cases} D & \mapsto D \\ \mathbf{y}_0 & \mapsto \mathbf{y}(t) \end{cases} , \quad t \mapsto \mathbf{y}(t) \text{ solution of IVP (11.1.33)} ,$$

This is a well-defined mapping of the state space into itself, by Thm. 11.1.32 and Ass. 11.1.38.

Now, we may also let  $t$  vary, which spawns a *family* of mappings  $\{\Phi^t\}$  of the state space into itself. However, it can also be viewed as a mapping with two arguments, a duration  $t$  and an initial state value  $\mathbf{y}_0$ !

### Definition 11.1.39. Evolution operator/mapping

Under Ass. 11.1.38 the mapping

$$\Phi : \begin{cases} \mathbb{R} \times D & \mapsto D \\ (t, \mathbf{y}_0) & \mapsto \Phi^t \mathbf{y}_0 := \mathbf{y}(t) \end{cases},$$

where  $t \mapsto \mathbf{y}(t) \in C^1(\mathbb{R}, \mathbb{R}^d)$  is the unique (global) solution of the IVP  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ ,  $\mathbf{y}(0) = \mathbf{y}_0$ , is the evolution operator/mapping for the autonomous ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ .

Note that  $t \mapsto \Phi^t \mathbf{y}_0$  describes the solution of  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  for  $\mathbf{y}(0) = \mathbf{y}_0$  (a trajectory). Therefore, by virtue of definition, we have

$$\frac{\partial \Phi}{\partial t}(t, \mathbf{y}) = \mathbf{f}(\Phi^t \mathbf{y}).$$

### Example 11.1.40 (Evolution operator for Lotka-Volterra ODE (11.1.10))

For  $d = 2$  the action of an evolution operator can be visualized by tracking the movement of point sets in state space. Here this is done for the Lotka-Volterra ODE (11.1.10):

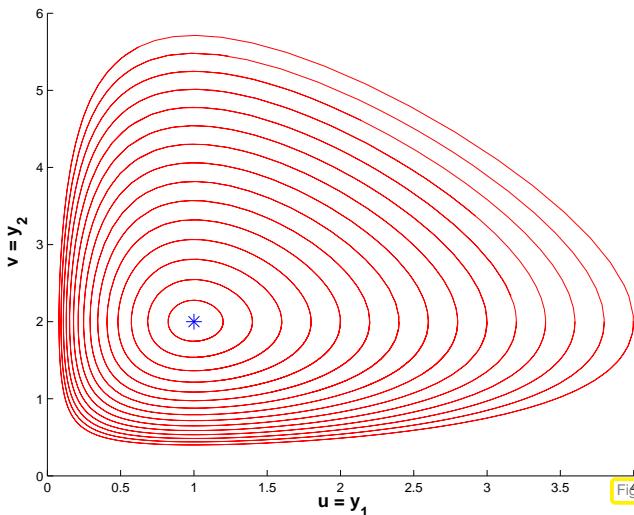
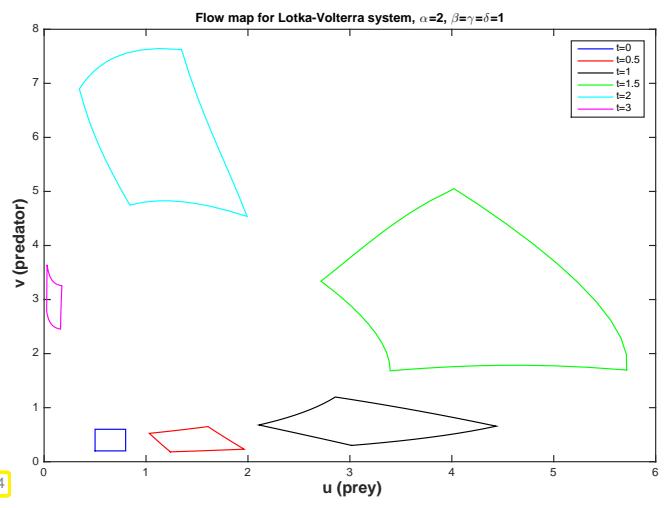


Fig. 363

trajectories  $t \mapsto \Phi^t \mathbf{y}_0$



state mapping  $\mathbf{y} \mapsto \Phi^t \mathbf{y}$

### Remark 11.1.41 (Group property of autonomous evolutions)

Under Ass. 11.1.38 the evolution operator gives rise to a group of mappings  $D \mapsto D$ :

$$\Phi^s \circ \Phi^t = \Phi^{s+t}, \quad \Phi^{-t} \circ \Phi^t = Id \quad \forall t \in \mathbb{R}. \quad (11.1.42)$$

This is a consequence of the uniqueness theorem Thm. 11.1.32. It is also intuitive: following an evolution up to time  $t$  and then for some more time  $s$  leads us to the same final state as observing it for the whole time  $s + t$ .

## 11.2 Introduction: Polygonal Approximation Methods

We target an initial value problem (11.1.20) for a first-order ordinary differential equation

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0 . \quad (11.1.20)$$

As usual, the right hand side function  $\mathbf{f}$  may be given only in **procedural form**, in MATLAB as

```
function v = f(t, y),
```

or in a C++ code as an object providing an evaluation operator, see Rem. 3.1.4. An evaluation of  $\mathbf{f}$  may involve costly computations.

### (11.2.1) Objectives of numerical integration

Two basic tasks can be identified in the field of **numerical integration** = approximate solution of initial value problems for ODEs (Please distinguish from “numerical quadrature”, see Chapter 5.):

- (I) Given initial time  $t_0$ , final time  $T$ , and initial state  $\mathbf{y}_0$  compute an approximation of  $\mathbf{y}(T)$ , where  $t \mapsto \mathbf{y}(t)$  is the solution of (11.1.20). A corresponding function in C++ could look like

```
State solveivp(double t0, double T, State y0);
```

Here **State** is a type providing a fixed size or variable size vector  $\in \mathbb{R}^d$ :

```
using State = Eigen::Matrix<double, statedim, 1>;
```

Here **statedim** is the dimension  $d$  of the state space that has to be known at compile time.

- (II) Output an *approximate* solution  $t \rightarrow \mathbf{y}_h(t)$  of (11.1.20) on  $[t_0, T]$  up to **final time**  $T \neq t_0$  for “all times”  $t \in [t_0, T]$  (actually for many times  $t_0 = \tau_0 < \tau_1 < \tau_2 < \dots < \tau_{m-1} < \tau_m = T$  consecutively): “plot solution”!

```
std::vector<State>
solveivp(State y0, const std::vector<double> &tauvec);
```

This section presents three methods that provide a **piecewise linear**, that is, “polygonal” approximation of solution trajectories  $t \mapsto \mathbf{y}(t)$ , cf. Ex. 3.1.8 for  $d = 1$ .

### (11.2.2) Temporal mesh

As in Section 4.5.1 the polygonal approximation in this section will be based on a **(temporal) mesh** ( $\rightarrow$  § 4.5.1)

$$\mathcal{M} := \{t_0 < t_1 < t_2 < \dots < t_{N-1} < t_N := T\} \subset [t_0, T] , \quad (11.2.3)$$

covering the time interval of interest between initial time  $t_0$  and final time  $T > t_0$ . We assume that the interval of interest is contained in the domain of definition of the solution of the IVP:  $[t_0, T] \subset J(t_0, \mathbf{y}_0)$ .

### 11.2.1 Explicit Euler method

#### Example 11.2.4 (Tangent field and solution curves)

For  $d = 1$  polygonal methods can be constructed by geometric considerations in the  $t - y$  plane, a model for the extended state space. We explain this for the **Riccati differential equation**, a scalar ODE:

$$\dot{y} = y^2 + t^2 \quad \Rightarrow \quad d = 1, \quad I, D = \mathbb{R}^+ . \quad (11.2.5)$$

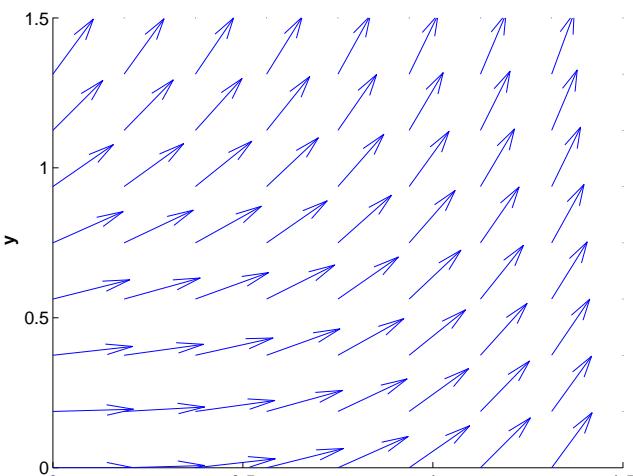


Fig. 365

tangent field

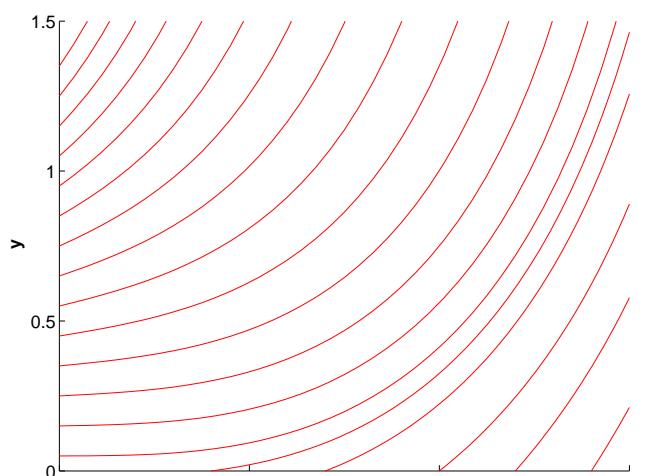
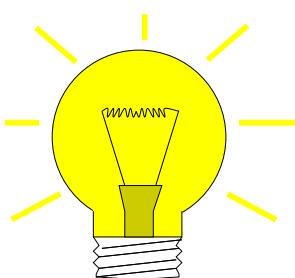


Fig. 366

solution curves

The solution curves run tangentially to the tangent field in each point of the extended state space.



Idea:

“follow the tangents over short periods of time”

- ① **timestepping**: successive approximation of evolution on *mesh intervals*  $[t_{k-1}, t_k]$ ,  $k = 1, \dots, N$ ,  $t_N := T$ ,
- ② approximation of solution on  $[t_{k-1}, t_k]$  by **tangent line to solution trajectory through  $(t_{k-1}, y_{k-1})$** .

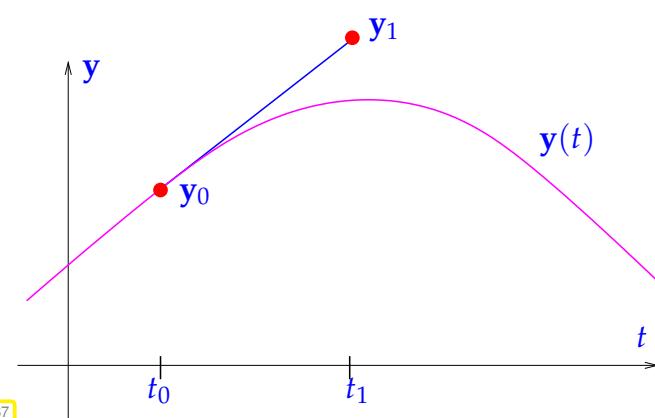


Fig. 367


**explicit Euler method** (Euler 1768)

- ▷ First step of explicit Euler method ( $d = 1$ ):  
Slope of tangent  $= f(t_0, y_0)$   
 $y_1$  serves as initial value for next step!  
See also [42, Ch. 74], [15, Alg. 11.4]

#### Example 11.2.6 (Visualization of explicit Euler method)

Temporal mesh

$$\mathcal{M} := \{t_j := j/5 : j = 0, \dots, 5\}.$$

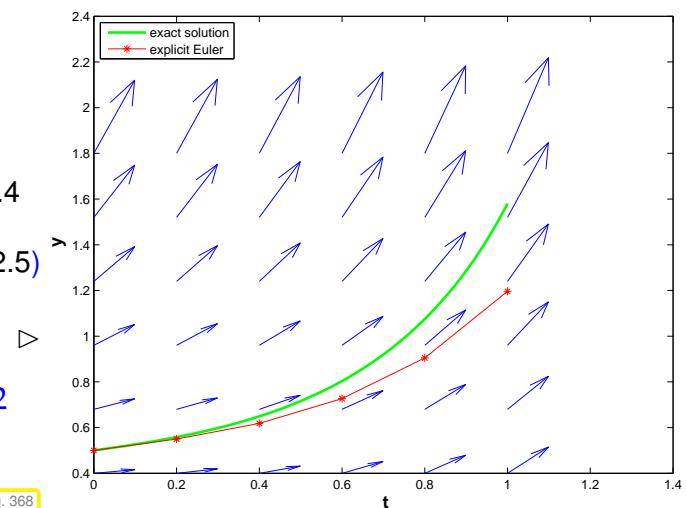
IVP for Riccati differential equation, see Ex. 11.2.4

$$\dot{y} = y^2 + t^2. \quad (11.2.5)$$

Here:  $y_0 = \frac{1}{2}$ ,  $t_0 = 0$ ,  $T = 1$ ,

— ≈ “Euler polygon” for uniform timestep  $h = 0.2$

→ ≈ tangent field of Riccati ODE



Formula: When applied to a general IVP of the form (11.1.20) the explicit Euler method generates a sequence  $(\mathbf{y}_k)_{k=0}^N$  by the recursion

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_k, \mathbf{y}_k), \quad k = 0, \dots, N-1, \quad (11.2.7)$$

with local (size of) timestep (stepsize)  $h_k := t_{k+1} - t_k$ .

### Remark 11.2.8 (Explicit Euler method as difference scheme)

One can obtain (11.2.7) by approximating the derivative  $\frac{d}{dt}$  by a forward difference quotient on the (temporal) mesh  $\mathcal{M} := \{t_0, t_1, \dots, t_N\}$ :

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \iff \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} = \mathbf{f}(t_k, \mathbf{y}_h(t_k)), \quad k = 0, \dots, N-1. \quad (11.2.9)$$

Difference schemes follow a simple policy for the *discretization* of differential equations: replace all derivatives by difference quotients connecting solution values on a set of discrete points (the mesh).

### Remark 11.2.10 (Output of explicit Euler method)

To begin with, the explicit Euler recursion (11.2.7) produces a sequence  $\mathbf{y}_0, \dots, \mathbf{y}_N$  of states. How does it deliver on the task (I) and (II) stated in § 11.2.1? By “geometric insight” we expect

$$\mathbf{y}_k \approx \mathbf{y}(t_k).$$

(As usual, we use the notation  $t \mapsto \mathbf{y}(t)$  for the exact solution of an IVP.)

Task (I): Easy, because  $\mathbf{y}_N$  already provides an approximation of  $\mathbf{y}(T)$ .

Task (II): The trajectory  $t \mapsto \mathbf{y}(t)$  is approximated by the piecewise linear function (“Euler polygon”)

$$\mathbf{y}_h : [t_0, t_N] \rightarrow \mathbb{R}^d, \quad \mathbf{y}_h(t) := \mathbf{y}_k \frac{t_{k+1} - t}{t_{k+1} - t_k} + \mathbf{y}_{k+1} \frac{t - t_k}{t_{k+1} - t_k} \quad \text{for } t \in [t_k, t_{k+1}], \quad (11.2.11)$$

see Fig. 368. This function can easily be sampled on any grid of  $[t_0, t_N]$ . In fact, it is the  $\mathcal{M}$ -piecewise linear interpolant of the data points  $(t_k, \mathbf{y}_k)$ ,  $k = 0, \dots, N$ , see Section 3.3.2).

The same considerations apply to the methods discussed in the next two sections and will not be repeated there.

### 11.2.2 Implicit Euler method

Why forward difference quotient and not backward difference quotient? Let's try!

On (temporal) mesh  $\mathcal{M} := \{t_0, t_1, \dots, t_N\}$  we obtain

$$\dot{\mathbf{y}} = f(t, \mathbf{y}) \iff \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} = f(t_{k+1}, \mathbf{y}_h(t_{k+1})) , \quad k = 0, \dots, N-1 . \quad (11.2.12)$$

backward difference quotient

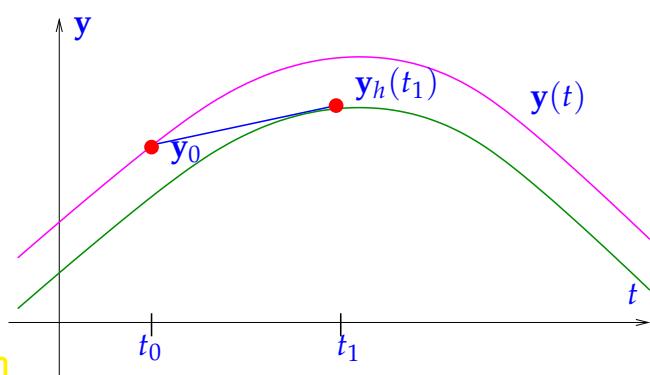
This leads to another simple timestepping scheme analogous to (11.2.7):

$$\boxed{\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1}) , \quad k = 0, \dots, N-1} , \quad (11.2.13)$$

with local timestep (stepsize)  $h_k := t_{k+1} - t_k$ .

(11.2.13) = implicit Euler method

Note: (11.2.13) requires solving a (possibly non-linear) system of equations to obtain  $\mathbf{y}_{k+1}$ !  
 (► Terminology “implicit”)



Geometry of implicit Euler method:

Approximate solution through  $(t_0, y_0)$  on  $[t_0, t_1]$  by

- straight line through  $(t_0, y_0)$
- with slope  $f(t_1, y_1)$

▷ —  $\hat{=}$  trajectory through  $(t_0, y_0)$ ,  
 —  $\hat{=}$  trajectory through  $(t_1, y_1)$ ,  
 —  $\hat{=}$  tangent at — in  $(t_1, y_1)$ .

#### Remark 11.2.14 (Feasibility of implicit Euler timestepping)

Issue: Is (11.2.13) well defined, that is, can we solve it for  $\mathbf{y}_{k+1}$  and is this solution unique?

Intuition: for small timesteps  $h > 0$  the right hand side of (11.2.13) is a “small perturbation of the identity”.

Formal: Consider an autonomous ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ , assume a continuously differentiable right hand side function  $\mathbf{f}, \mathbf{f} \in C^1(D, \mathbb{R}^d)$ , and regard (11.2.13) as an  $h$ -dependent non-linear system of equations:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1}) \Leftrightarrow G(h, \mathbf{y}_{k+1}) = 0 \quad \text{with} \quad G(h, \mathbf{z}) := \mathbf{z} - h \mathbf{f}(t_{k+1}, \mathbf{z}) - \mathbf{y}_k .$$

To investigate the solvability of this non-linear equation we start with an observation about a partial derivative of  $G$ :

$$\frac{dG}{d\mathbf{z}}(h, \mathbf{z}) = \mathbf{I} - h D_{\mathbf{y}} \mathbf{f}(t_{k+1}, \mathbf{z}) \Rightarrow \frac{dG}{d\mathbf{z}}(0, \mathbf{z}) = \mathbf{I} .$$

In addition,  $G(0, \mathbf{y}_k) = \mathbf{0}$ . Next, recall the **implicit function theorem** [77, Thm. 7.8.1]:

### Theorem 11.2.15. Implicit function theorem

Let  $G = G(\mathbf{x}, \mathbf{y})$  a continuously differentiable function of  $\mathbf{x} \in \mathbb{R}^k$  and  $\mathbf{y} \in \mathbb{R}^\ell$ , defined on the open set  $\Omega \subset \mathbb{R}^k \times \mathbb{R}^\ell$  with values in  $\mathbb{R}^\ell$ :  $G : \Omega \subset \mathbb{R}^k \times \mathbb{R}^\ell \rightarrow \mathbb{R}^\ell$ .

Assume that  $G$  has a zero in  $\mathbf{z}_0 := \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{y}_0 \end{bmatrix} \in \Omega$ ,  $\mathbf{x}_0 \in \mathbb{R}^k$ ,  $\mathbf{y}_0 \in \mathbb{R}^\ell$ :  $G(\mathbf{z}_0) = \mathbf{0}$ .

If the Jacobian  $\frac{\partial G}{\partial \mathbf{y}}(\mathbf{p}_0) \in \mathbb{R}^{\ell, \ell}$  is **invertible**, then there is an open neighborhood  $U$  of  $\mathbf{x}_0 \in \mathbb{R}^k$  and a continuously differentiable function  $\mathbf{g} : U \rightarrow \mathbb{R}^\ell$  such that

$$\mathbf{g}(\mathbf{x}_0) = \mathbf{y}_0 \quad \text{and} \quad G(\mathbf{x}, \mathbf{g}(\mathbf{x})) = \mathbf{0} \quad \forall \mathbf{x} \in U .$$

For **sufficiently small**  $|h|$  it permits us to conclude that the equation  $G(h, \mathbf{z}) = \mathbf{0}$  defines a continuous function  $\mathbf{g} = \mathbf{g}(h)$  with  $\mathbf{g}(0) = \mathbf{y}_k$ .

➤ for **sufficiently small**  $h > 0$  the equation (11.2.13) has a unique solution  $\mathbf{y}_{k+1}$ .

### 11.2.3 Implicit midpoint method

Beside using forward or backward difference quotients, the derivative  $\dot{\mathbf{y}}$  can also be approximated by the **symmetric difference quotient**, see also (3.2.40),

$$\dot{\mathbf{y}}(t) \approx \frac{\mathbf{y}(t+h) - \mathbf{y}(t-h)}{2h} . \quad (11.2.16)$$

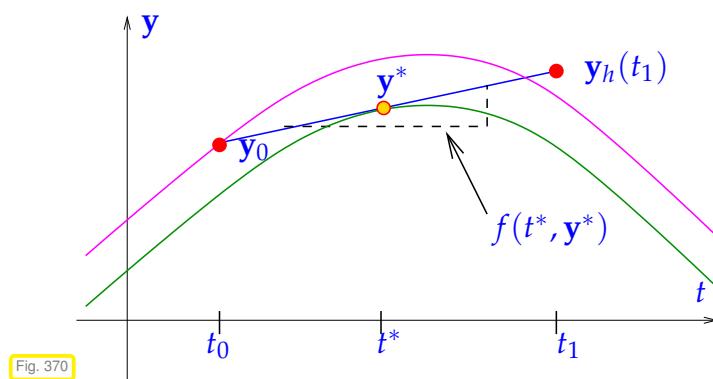
The idea is to apply this formula in  $t = \frac{1}{2}(t_k + t_{k+1})$  with  $h = h_k/2$ , which transforms the ODE into

$$\dot{\mathbf{y}} = f(t, \mathbf{y}) \iff \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} = f\left(\frac{1}{2}(t_k + t_{k+1}), \mathbf{y}_h\left(\frac{1}{2}(t_{k+1} + t_{k+1})\right)\right), \quad k = 0, \dots, N-1 . \quad (11.2.17)$$

The trouble is that the value  $\mathbf{y}_h\left(\frac{1}{2}(t_{k+1} + t_{k+1})\right)$  does not seem to be available, unless we recall that the approximate trajectory  $t \mapsto \mathbf{y}_h(t)$  is supposed to be piecewise linear, which implies  $\mathbf{y}_h\left(\frac{1}{2}(t_{k+1} + t_{k+1})\right) = \frac{1}{2}(\mathbf{y}_h(t_k) + \mathbf{y}_h(t_{k+1}))$ . This gives the recursion formula for the **implicit midpoint method** in analogy to (11.2.7) and (11.2.13):

$$\boxed{\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}\left(\frac{1}{2}(t_k + t_{k+1}), \frac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k+1})\right)}, \quad k = 0, \dots, N-1 , \quad (11.2.18)$$

with local **timestep** (stepsize)  $h_k := t_{k+1} - t_k$ .



Implicit midpoint method: a geometric view:

Approximate trajectory through  $(t_0, \mathbf{y}_0)$  on  $[t_0, t_1]$  by

- straight line through  $(t_0, \mathbf{y}_0)$
  - with slope  $f(t^*, \mathbf{y}^*)$ , where  

$$t^* := \frac{1}{2}(t_0 + t_1), \mathbf{y}^* = \frac{1}{2}(\mathbf{y}_0 + \mathbf{y}_1)$$
- $\triangleq$
- $\hat{\textcolor{magenta}{—}}$   $\hat{=}$  trajectory through  $(t_0, \mathbf{y}_0)$ ,
  - $\hat{\textcolor{green}{—}}$   $\hat{=}$  trajectory through  $(t^*, \mathbf{y}^*)$ ,
  - $\hat{\textcolor{blue}{—}}$   $\hat{=}$  tangent at  $\textcolor{green}{—}$  in  $(t^*, \mathbf{y}^*)$ .

As in the case of (11.2.13), also (11.2.18) entails solving a (non-linear) system of equations in order to obtain  $\mathbf{y}_{k+1}$ . Rem. 11.2.14 also holds true in this case: for sufficiently small  $h$  (11.2.18) will have a unique solution  $\mathbf{y}_{k+1}$ , which renders the recursion well defined.

## 11.3 General single step methods

Now we fit the numerical schemes introduced in the previous section into a more general class of methods for the solution of (autonomous) initial value problems (11.1.33) for ODEs. Throughout we assume that all times considered belong to the domain of definition of the unique solution  $t \rightarrow \mathbf{y}(t)$  of (11.1.33), that is, for  $T > 0$  we take for granted  $[0, T] \subset J(\mathbf{y}_0)$  (temporal domain of definition of the solution of an IVP is explained in § 11.1.34).

### 11.3.1 Definition

#### (11.3.1) Discrete evolution operators

Recall the Euler methods for autonomous ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ :

$$\begin{aligned} \text{explicit Euler: } \mathbf{y}_{k+1} &= \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_k), \\ \text{implicit Euler: } \mathbf{y}_{k+1} &= \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_{k+1}). \end{aligned}$$

Both formulas, for sufficiently small  $h$  ( $\rightarrow$  Rem. 11.2.14), provide a mapping

$$(\mathbf{y}_k, h_k) \mapsto \Psi(h, \mathbf{y}_k) := \mathbf{y}_{k+1}. \quad (11.3.2)$$

If  $\mathbf{y}_0$  is the initial value, then  $\mathbf{y}_1 := \Psi(h, \mathbf{y}_0)$  can be regarded as an approximation of  $\mathbf{y}(h)$ , the value returned by the evolution operator ( $\rightarrow$  Def. 11.1.39) for  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  applied to  $\mathbf{y}_0$  over the period  $h$ .  $\mathbf{y}(t_k)$ :

$$\mathbf{y}_1 = \Psi(h, \mathbf{y}_0) \iff \mathbf{y}(h) = \Phi^h \mathbf{y}_0 \Rightarrow \boxed{\Psi(h, \mathbf{y}) \approx \Phi^h \mathbf{y}}, \quad (11.3.3)$$

In a sense the polygonal approximation methods as based on approximations for the evolution operator associated with the ODE.

This is what every single step method does: it tries to approximate the evolution operator  $\Phi$  for an ODE by a mapping of the type (11.3.2).

→ mapping  $\Psi$  from (11.3.2) is called **discrete evolution**.

☞ Notation: for discrete evolutions we often write  $\Psi^h \mathbf{y} := \Psi(h, \mathbf{y})$

#### Remark 11.3.4 (Discretization)

The adjective “**discrete**” used above designates (components of) methods that attempt to approximate the solution of an IVP by a sequence of finitely many states. “**Discretization**” is the process of converting an ODE into a discrete model. This parlance is adopted for all procedures that reduce a “continuous model” involving ordinary or partial differential equations to a form with a finite number of unknowns.

Above we identified the discrete evolutions underlying the polygonal approximation methods. Vice versa, a mapping  $\Psi$  as given in (11.3.2) defines a single step method.

#### Definition 11.3.5. Single step method (for autonomous ODE) → [63, Def. 11.2]

Given a discrete evolution  $\Psi : \Omega \subset \mathbb{R} \times D \mapsto \mathbb{R}^d$ , an initial state  $\mathbf{y}_0$ , and a temporal mesh  $\mathcal{M} := \{0 =: t_0 < t_1 < \dots < t_N := T\}$  the recursion

$$\mathbf{y}_{k+1} := \Psi(t_{k+1} - t_k, \mathbf{y}_k), \quad k = 0, \dots, N-1, \quad (11.3.6)$$

defines a **single step method** (SSM) for the autonomous IVP  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ ,  $\mathbf{y}(0) = \mathbf{y}_0$  on the interval  $[0, T]$ .

☞ In a sense, a single step method defined through its associated discrete evolution does not approximate a concrete initial value problem, but tries to approximate an ODE in the form of its evolution operator.

In MATLAB syntax a discrete evolutions can be incarnated by a function of the following form:

```
 $\Psi^h \mathbf{y} \longleftrightarrow \text{function } y1 = \text{discevl}(h, y0).$ 
 $\quad (\text{function } y1 = \text{discevl}(@(y) \text{rhs}(y), h, y0))$ 
```

The concept of single step method according to Def. 11.3.5 can be generalized to non-autonomous ODEs, which leads to recursions of the form:

$$\mathbf{y}_{k+1} := \Psi(t_k, t_{k+1}, \mathbf{y}_k), \quad k = 0, \dots, N-1,$$

for a discrete evolution operator  $\Psi$  defined on  $I \times I \times D$ .

#### (11.3.7) Consistent single step methods

All meaningful single step methods turn out to be modifications of the explicit Euler method (11.2.7).

### Consistent discrete evolution

The discrete evolution  $\Psi$  defining a single step method according to Def. 11.3.5 and (11.3.6) for the autonomous ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  invariably is of the form

$$\Psi^h \mathbf{y} = \mathbf{y} + h\psi(h, \mathbf{y}) \quad \text{with} \quad \begin{aligned} \psi : I \times D &\rightarrow \mathbb{R}^d \text{ continuous,} \\ \psi(0, \mathbf{y}) &= \mathbf{f}(\mathbf{y}). \end{aligned} \quad (11.3.9)$$

### Definition 11.3.10. Consistent single step methods

A single step method according to Def. 11.3.5 based on a discrete evolution of the form (11.3.9) is called **consistent** with the ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ .

### Example 11.3.11 (Consistency of implicit midpoint method)

The discrete evolution  $\Psi$  and, hence, the function  $\psi = \psi(h, \mathbf{y})$  for the implicit midpoint method are defined only implicitly, of course. Thus, consistency cannot immediately be seen from a formula for  $\psi$ .

We examine consistency of the implicit midpoint method defined by

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}\left(\frac{1}{2}(t_k + t_{k+1}), \frac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k+1})\right), \quad k = 0, \dots, N-1. \quad (11.2.18)$$

Assume that

- the right hand side function  $\mathbf{f}$  is smooth, at least  $\mathbf{f} \in C^1(D)$ ,
- and  $|h|$  is sufficiently small to guarantee the existence of a solution  $\mathbf{y}_{k+1}$  of (11.2.18), see Rem. 11.2.14.

The idea is to verify (11.3.9) by formal Taylor expansion of  $\mathbf{y}_{k+1}$  in  $h$ . To that end we plug (11.2.18) into itself and rely on Taylor expansion of  $\mathbf{f}$ :

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}\left(\frac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k+1})\right) \stackrel{(11.2.18)}{=} \mathbf{y}_k + h\underbrace{\mathbf{f}(\mathbf{y}_k + \frac{1}{2}h\mathbf{f}(\frac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k+1})))}_{=\psi(h, \mathbf{y}_k)}. \quad .$$

Since, by the implicit function theorem,  $\mathbf{y}_{k+1}$  continuously depends on  $h$  and  $\mathbf{y}_k$ ,  $\psi(h, \mathbf{y}_k)$  has the desired properties, in particular  $\psi(0, \mathbf{y}) = \mathbf{f}(\mathbf{y})$  is clear.

### Remark 11.3.12 (Notation for single step methods)

Many authors specify a single step method by writing down the first step for a general stepsize  $h$

$$\mathbf{y}_1 = \text{expression in } \mathbf{y}_0, h \text{ and } \mathbf{f}.$$

Actually, this fixes the underlying discrete evolution. Also this course will sometimes adopt this practice.

### (11.3.13) Output of single step methods

Here we resume and continue the discussion of Rem. 11.2.10 for general single step methods according to Def. 11.3.5. Assuming unique solvability of the systems of equations faced in each step of an implicit method, every single step method based on a mesh  $\mathcal{M} = \{0 = t_0 < t_1 < \dots < t_N := T\}$  produces a finite sequence  $(\mathbf{y}_0, \mathbf{y}_1, \dots, \mathbf{y}_N)$  of states, where the first agrees with the initial state  $\mathbf{y}_0$ .

We expect that the states provide a pointwise approximation of the solution trajectory  $t \rightarrow \mathbf{y}(t)$ :

$$\mathbf{y}_k \approx \mathbf{y}(t_k), \quad k = 1, \dots, N.$$

Thus task (I) from § 11.2.1, computing an approximation for  $\mathbf{y}(T)$ , is again easy: output  $\mathbf{y}_N$  as an approximation of  $\mathbf{y}(T)$ .

Task (II) from § 11.2.1, computing the solution trajectory, requires interpolation of the data points  $(t_k, \mathbf{y}_k)$  using some of the techniques presented in Chapter 3. The natural option is  $\mathcal{M}$ -piecewise polynomial interpolation, generalizing the polygonal approximation (11.2.11) used in Section 11.2.

Note that from the ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  the derivatives  $\dot{\mathbf{y}}_h(t_k) = \mathbf{f}(\mathbf{y}_k)$  are available without any further approximation. This facilitates cubic Hermite interpolation (→ Def. 3.4.1), which yields

$$\mathbf{y}_h \in C^1([0, T]): \quad \mathbf{y}_h|_{[x_{k-1}, x_k]} \in \mathcal{P}_3, \quad \mathbf{y}_h(t_k) = \mathbf{y}_k, \quad \frac{d\mathbf{y}_h}{dt}(t_k) = \mathbf{f}(\mathbf{y}_k).$$

Summing up, an approximate trajectory  $t \mapsto \mathbf{y}_h(t)$  is built in two stages:

- (i) Compute sequence  $(\mathbf{y}_k)_k$  by running the single step method.
- (ii) Post-process the obtained sequence, usually by applying interpolation, to get  $\mathbf{y}_h$ .

## 11.3.2 Convergence of single step methods



*Supplementary reading.* See [15, Sect. 11.5] and [63, Sect. 11.3] for related presentations.

### (11.3.14) Discretization error of single step methods

Errors in numerical integration are called **discretization errors**, cf. Rem. 11.3.4.

Depending on the objective of numerical integration as stated in § 11.2.1 different notions of discretization error appropriate

- (I) If only the solution at final time is sought, the discretization error is

$$\epsilon_N := \|\mathbf{y}(T) - \mathbf{y}_N\|,$$

where  $\|\cdot\|$  is some vector norm on  $\mathbb{R}^d$ .

(II) If we want to approximate the solution trajectory for (11.1.33) the discretization error is the function

$$t \mapsto \mathbf{e}(t) , \quad \mathbf{e}(t) := \mathbf{y}(t) - \mathbf{y}_h(t) ,$$

where  $\rightarrow \mathbf{y}_h(t)$  is the approximate trajectory obtained by post-processing, see § 11.3.13. In this case accuracy of the method is gauged by looking at norms of the function  $\mathbf{e}$ , see § 3.2.61 for examples.

(III) Between (I) and (II) is the pointwise discretization error, which is the sequence (**grid function**)

$$\mathbf{e} : \mathcal{M} \rightarrow D , \quad \mathbf{e}_k := \mathbf{y}(t_k) - \mathbf{y}_k , \quad k = 0, \dots, N . \quad (11.3.15)$$

In this case we may consider the maximum error in the mesh points

$$\|(\mathbf{e})\|_\infty := \max_{k \in \{1, \dots, N\}} \|\mathbf{e}_k\| ,$$

where  $\|\cdot\|$  is a suitable vector norm on  $\mathbb{R}^d$ , usually the Euclidean vector norm.

### (11.3.16) Asymptotic convergence of single step methods

Once the discrete evolution  $\Psi$  associated with the ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  is specified, the single step method according to Def. 11.3.5 is fixed. The only way to control the accuracy of the solution  $\mathbf{y}_N$  or  $t \mapsto \mathbf{y}_h(t)$  is through the selection of the mesh  $\mathcal{M} = \{0 = t_0 < t_1 < \dots < t_N = T\}$ .

Hence we study convergence of single step methods for **families of meshes**  $\{\mathcal{M}_\ell\}$  and track the decay of (a norm) of the discretization error ( $\rightarrow$  § 11.3.14) as a function of the number  $N := |\mathcal{M}|$  of mesh points. In other words, we examine ***h*-convergence**. We already did this in the case of piecewise polynomial interpolation in Section 4.5.1 and composite numerical quadrature in Section 5.4.

When investigating asymptotic convergence of single step methods we often resort to families of **equidistant** meshes of  $[0, T]$ :

$$\mathcal{M}_N := \{t_k := \frac{k}{N}T : k = 0, \dots, N\} . \quad (11.3.17)$$

We also call this the use of **uniform** timesteps of size  $h := \frac{T}{N}$ .

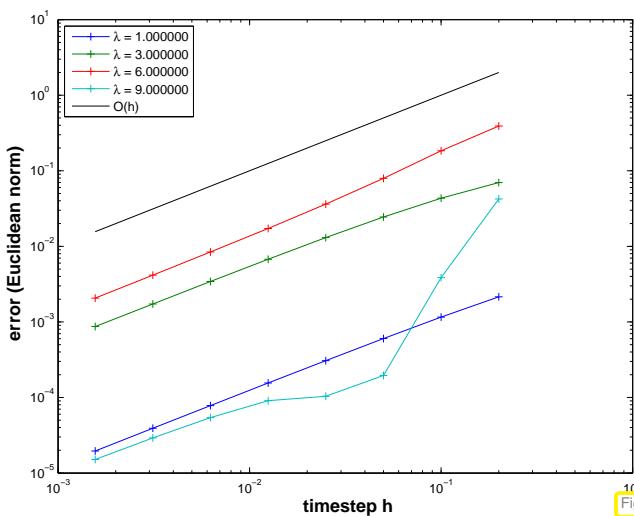
### Example 11.3.18 (Speed of convergence of Euler methods)

The setting for this experiment is as follows:

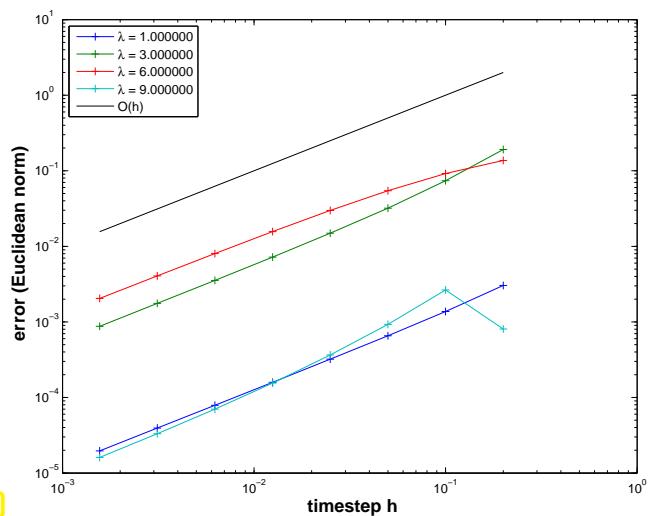
- \* We consider the following IVP for the logistic ODE, see Ex. 11.1.5

$$\dot{y} = \lambda y(1 - y) , \quad y(0) = 0.01 .$$

- \* We apply explicit and implicit Euler methods (11.2.7)/(11.2.13) with uniform timestep  $h = 1/N$ ,  $N \in \{5, 10, 20, 40, 80, 160, 320, 640\}$ .
- \* Monitored: Error at final time  $E(h) := |y(1) - y_N|$

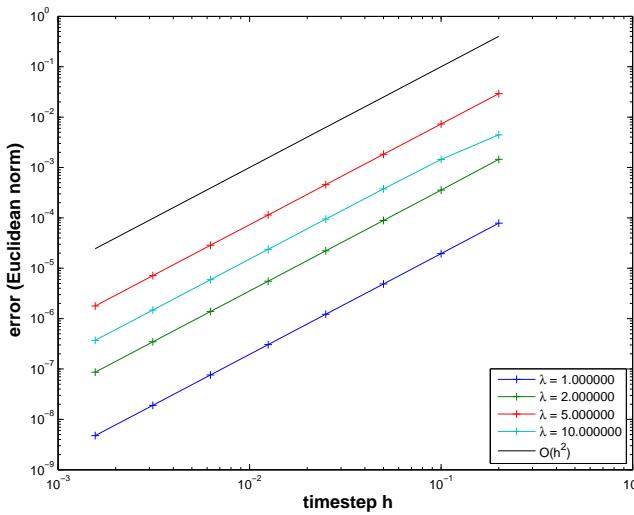


explicit Euler method



implicit Euler method

$O(N^{-1}) = O(h)$  algebraic convergence in both cases for  $h \rightarrow 0$



Parlance: based on the observed rate of algebraic convergence, the two Euler methods are said to “converge with first order”, whereas the implicit midpoint method is called “second-order convergent”.

The observations made for polygonal timestepping methods reflect a general pattern:

### Algebraic convergence of single step methods

Consider numerical integration of an initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0 , \quad (11.1.20)$$

with sufficiently smooth right hand side function  $\mathbf{f} : I \times D \rightarrow \mathbb{R}^d$ .

Then customary single step methods ( $\rightarrow$  Def. 11.3.5) will enjoy *algebraic convergence in the mesh-width*, more precisely, see [15, Thm. 11.25],

there is a  $p \in \mathbb{N}$  such that the sequence  $(\mathbf{y}_k)_k$  generated by the single step method

for  $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$  on a mesh  $\mathcal{M} := \{t_0 < t_1 < \dots < t_N = T\}$  satisfies

$$\max_k \|\mathbf{y}_k - \mathbf{y}(t_k)\| \leq Ch^p \quad \text{for } h := \max_{k=1,\dots,N} |t_k - t_{k-1}| \rightarrow 0 , \quad (11.3.20)$$

with  $C > 0$  independent of  $M$

### **Definition 11.3.21. Order of a single step method**

The minimal integer  $p \in \mathbb{N}$  for which (11.3.20) holds for a single step method when applied to an ODE with (sufficiently) smooth right hand side, is called the **order** of the method.

As in the case of quadrature rules ( $\rightarrow$  Def. 5.3.1) their order is the principal intrinsic indicator for the “quality” of a single step method.

(11.3.22) Convergence analysis for the explicit Euler method [42, Ch. 74]

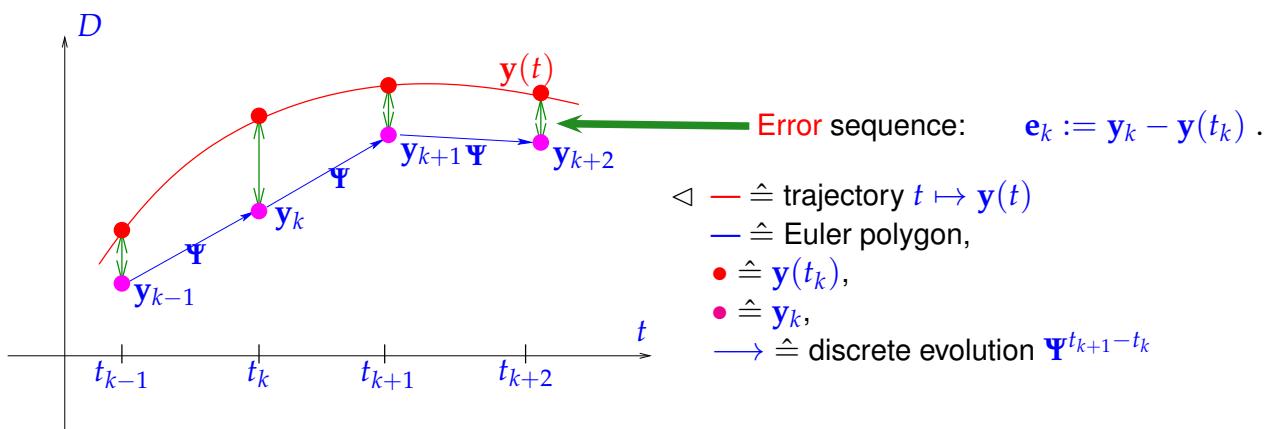
We consider the explicit Euler method (11.2.7) on a mesh  $\mathcal{M} := \{0 = t_0 < t_1 < \dots < t_N = T\}$  for a generic autonomous IVP (11.1.20) with sufficiently smooth and (*globally*) Lipschitz continuous  $\mathbf{f}$ , that is,

$$\exists L > 0: \quad \|\mathbf{f}(\mathbf{y}) - \mathbf{f}(\mathbf{z})\| \leq L \|\mathbf{y} - \mathbf{z}\| \quad \forall \mathbf{y}, \mathbf{z} \in D, \quad (11.3.23)$$

and exact solution  $t \mapsto \mathbf{y}(t)$ . Throughout we assume that solutions of  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  are defined on  $[0, T]$  for all initial states  $\mathbf{y}_0 \in D$ .

Recall: recursion for explicit Euler method

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_k), \quad k = 1, \dots, N-1. \quad (11.2.7)$$



### ① Abstract splitting of error:

Here and in what follows we rely on the abstract concepts of the evolution operator  $\Phi$  associated with the ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  ( $\rightarrow$  Def. 11.1.39) and discrete evolution operator  $\Psi$  defining the explicit Euler single step method, see Def. 11.3.5:

$$(11.2.7) \Rightarrow \Psi^h \mathbf{y} = \mathbf{y} + h\mathbf{f}(\mathbf{y}) . \quad (11.3.24)$$

We argue that in this context the abstraction pays off, because it helps elucidate a general technique for the convergence analysis of single step methods.

Fundamental error splitting

$$\begin{aligned} \mathbf{e}_{k+1} &= \Psi^{h_k} \mathbf{y}_k - \Phi^{h_k} \mathbf{y}(t_k) \\ &= \underbrace{\Psi^{h_k} \mathbf{y}_k - \Psi^{h_k} \mathbf{y}(t_k)}_{\text{propagated error}} + \underbrace{\Psi^{h_k} \mathbf{y}(t_k) - \Phi^{h_k} \mathbf{y}(t_k)}_{\text{one-step error}}. \end{aligned} \quad (11.3.25)$$

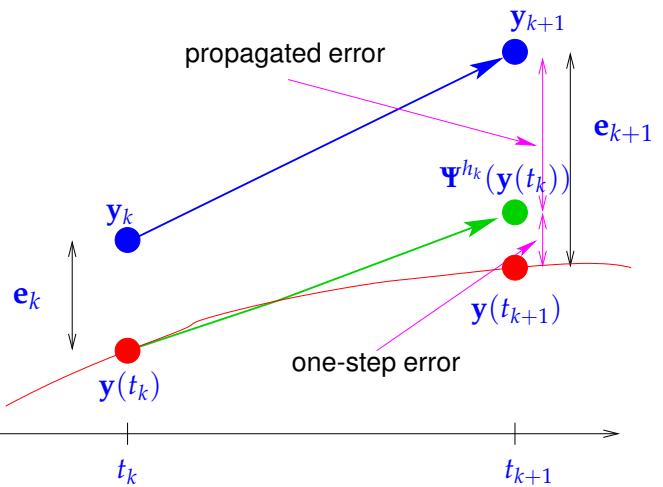


Fig. 374

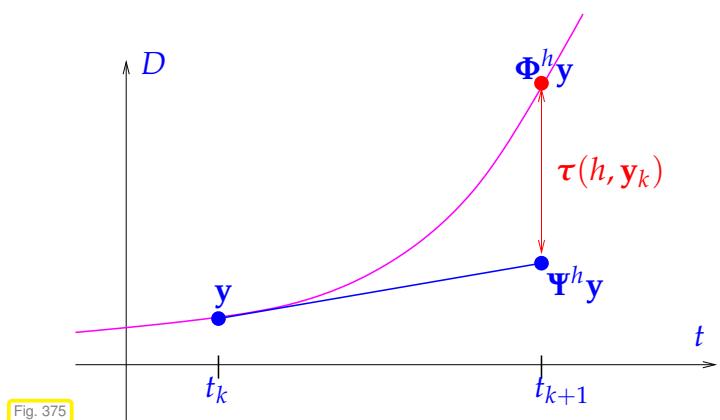


Fig. 375

A generic one-step error expressed through continuous and discrete evolutions:

$$\tau(h, \mathbf{y}) := \Psi^h \mathbf{y} - \Phi^h \mathbf{y}. \quad (11.3.26)$$

- ▷ geometric visualisation of one-step error for explicit Euler method (11.2.7), cf. Fig. 367,  $h := t_{k+1} - t_k$
- : solution trajectory through  $(t_k, \mathbf{y})$

## ② Estimate for one-step error $\tau(h_k, \mathbf{y}(t_k))$ :

Geometric considerations: distance of a smooth curve and its tangent shrinks as the square of the distance to the intersection point (curve locally looks like a parabola in the  $\xi - \eta$  coordinate system, see Fig. 377).

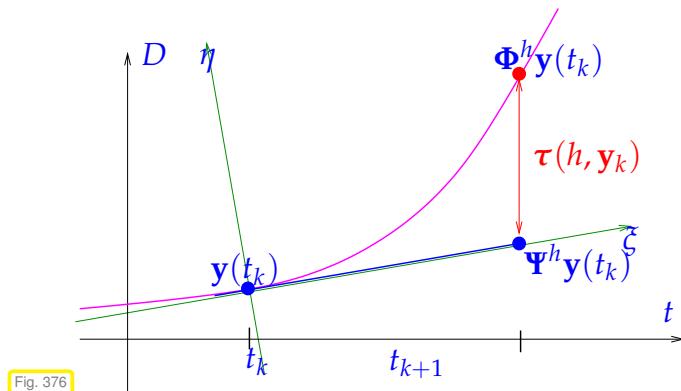


Fig. 376

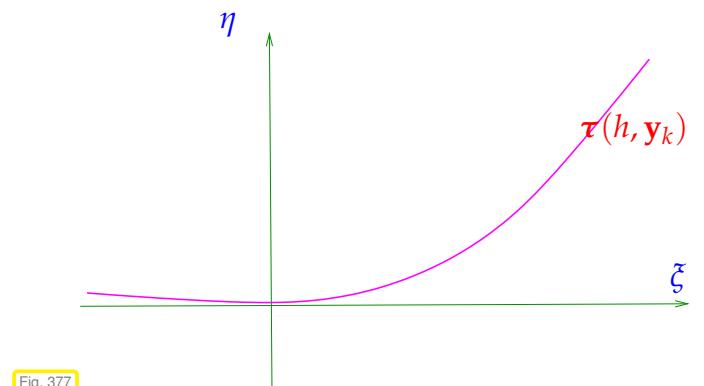


Fig. 377

The geometric considerations can be made rigorous by analysis: recall Taylor's formula for the function  $\mathbf{y} \in C^{K+1}$  [77, Satz 5.5.1]:

$$\begin{aligned} \mathbf{y}(t+h) - \mathbf{y}(t) &= \sum_{j=0}^K \mathbf{y}^{(j)}(t) \frac{h^j}{j!} + \underbrace{\int_t^{t+h} \mathbf{y}^{(K+1)}(\tau) \frac{(t+h-\tau)^K}{K!} d\tau}_{= \frac{\mathbf{y}^{(K+1)}(\xi)}{K!} h^{K+1}}, \end{aligned} \quad (11.3.27)$$

for some  $\xi \in [t, t+h]$ . We conclude that, if  $\mathbf{y} \in C^2([0, T])$ , which is ensured for smooth  $\mathbf{f}$ , see Lemma 11.1.4, then

$$\mathbf{y}(t_{k+1}) - \mathbf{y}(t_k) = \dot{\mathbf{y}}(t_k) h_k + \frac{1}{2} \ddot{\mathbf{y}}(\xi_k) h_k^2 = \mathbf{f}(\mathbf{y}(t_k)) h_k + \frac{1}{2} \ddot{\mathbf{y}}(\xi_k) h_k^2,$$

for some  $t_k \leq \xi_k \leq t_{k+1}$ . This leads to an expression for the one-step error from (11.3.26)

$$\begin{aligned}\tau(h_k, \mathbf{y}(t_k)) &= \Psi^{h_k} \mathbf{y}(t_k) - \mathbf{y}(t_{k+1}) \\ &\stackrel{(11.3.24)}{=} \mathbf{y}(t_k) + h_k \mathbf{f}(\mathbf{y}(t_k)) - \mathbf{y}(t_k) - \mathbf{f}(\mathbf{y}(t_k)) h_k + \frac{1}{2} \ddot{\mathbf{y}}(\xi_k) h_k^2 \\ &= \frac{1}{2} \ddot{\mathbf{y}}(\xi_k) h_k^2.\end{aligned}\quad (11.3.28)$$

Sloppily speaking, we observe  $\boxed{\tau(h_k, \mathbf{y}(t_k)) = O(h_k^2)}$  uniformly for  $h_k \rightarrow 0$ .

### ③ Estimate for the propagated error from (11.3.25)

$$\begin{aligned}\|\Psi^{h_k} \mathbf{y}_k - \Psi^{h_k} \mathbf{y}(t_k)\| &= \|\mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_k) - \mathbf{y}(t_k) - h_k \mathbf{f}(\mathbf{y}(t_k))\| \\ &\stackrel{(11.3.23)}{\leq} (1 + L h_k) \|\mathbf{y}_k - \mathbf{y}(t_k)\|.\end{aligned}\quad (11.3.29)$$


③ Obtain *recursion* for error norms  $\epsilon_k := \|\mathbf{e}_k\|$  by  $\triangle$ -inequality:

$$\epsilon_{k+1} \leq (1 + h_k L) \epsilon_k + \rho_k, \quad \rho_k := \frac{1}{2} h_k^2 \max_{t_k \leq \tau \leq t_{k+1}} \|\ddot{\mathbf{y}}(\tau)\|. \quad (11.3.30)$$

Taking into account  $\epsilon_0 = 0$ , this leads to

$$\epsilon_k \leq \sum_{l=1}^k \prod_{j=1}^{l-1} (1 + L h_j) \rho_l, \quad k = 1, \dots, N. \quad (11.3.31)$$

Use the elementary estimate  $(1 + L h_j) \leq \exp(L h_j)$  (by convexity of exponential function):

$$(11.3.31) \Rightarrow \epsilon_k \leq \sum_{l=1}^k \prod_{j=1}^{l-1} \exp(L h_j) \cdot \rho_l = \sum_{l=1}^k \exp(L \sum_{j=1}^{l-1} h_j) \rho_l.$$

Note:  $\sum_{j=1}^{l-1} h_j \leq T$  for final time  $T$  and conclude

$$\begin{aligned}\epsilon_k &\leq \exp(LT) \sum_{l=1}^k \rho_l \leq \exp(LT) \max_k \frac{\rho_k}{h_k} \sum_{l=1}^k h_l \leq T \exp(LT) \max_{l=1, \dots, k} h_l \cdot \max_{t_0 \leq \tau \leq t_k} \|\ddot{\mathbf{y}}(\tau)\|. \\ &\quad \blacktriangleright \|\mathbf{y}_k - \mathbf{y}(t_k)\| \leq T \exp(LT) \max_{l=1, \dots, k} h_l \cdot \max_{t_0 \leq \tau \leq t_k} \|\ddot{\mathbf{y}}(\tau)\|.\end{aligned}\quad (11.3.32)$$

We can summarize the insight gleaned through this theoretical analysis as follows:

Total error arises from accumulation of propagated one-step errors!

First conclusions from (11.3.32):

- \* error bound  $= O(h)$ ,  $h := \max_l h_l$  ( $\blacktriangleright$  1st-order algebraic convergence)
- \* Error bound grows exponentially with the length  $T$  of the integration interval.

### (11.3.33) One-step error and order of a single step method

In the analysis of the global discretization error of the explicit Euler method in § 11.3.22 a one-step error of size  $\mathcal{O}(h_k^2)$  led to a total error of  $\mathcal{O}(h)$  through the effect of error accumulation over  $N \approx h^{-1}$  steps. This relationship remains valid for almost all single step methods:

Consider an IVP (11.1.20) with solution  $t \mapsto \mathbf{y}(t)$  and a single step method defined by the discrete evolution  $\Psi$  ( $\rightarrow$  Def. 11.3.5). If the *one-step error along the solution trajectory* satisfies ( $\Phi$  is the evolution map associated with the ODE, see Def. 11.1.39)

$$\|\Psi^h \mathbf{y}(t) - \Phi^h \mathbf{y}(t)\| \leq Ch^{p+1} \quad \forall h \text{ sufficiently small}, t \in [0, T],$$

for some  $p \in \mathbb{N}$  and  $C > 0$ , then, usually,

$$\max_k \|\mathbf{y}_k - \mathbf{y}(t_k)\| \leq \bar{C} h_M^p,$$

with  $\bar{C} > 0$  independent of the temporal mesh  $\mathcal{M}$ .

A rigorous statement as a theorem would involve some particular assumptions on  $\Psi$ , which we do not want to give here. These assumptions are satisfied, for instance, for all the methods presented in the sequel.

## 11.4 Explicit Runge-Kutta Methods



*Supplementary reading.* [15, Sect. 11.6], [42, Ch. 76], [63, Sect. 11.8]

So far we only know first and second order methods from 11.2: the explicit and implicit Euler method (11.2.7) and (11.2.13), respectively, are of first order, the implicit midpoint rule of second order. We observed this in Ex. 11.3.18 and it can be proved rigorously for all three methods adapting the arguments of § 11.3.22.

Thus, barring the impact of roundoff, the low-order polygonal approximation methods are guaranteed to achieve any prescribed accuracy provided that the mesh is fine enough. Why should we need any other timestepping schemes?

#### Remark 11.4.1 (Rationale for high-order single step methods   cf. [15, Sect. 11.5.3])

We argue that the use of higher-order timestepping methods is highly *advisable for the sake of efficiency*. The reasoning is very similar to that of Rem. 5.3.47, when we considered numerical quadrature. The reader is advised to study that remark again.

As we saw in § 11.3.16 error bounds for single step methods for the solution of IVPs will inevitably feature unknown constants “ $C > 0$ ”. Thus they do **not** give useful information about the discretization error for

a concrete IVP and mesh. Hence, it is too ambitious to ask how many timesteps are needed so that  $\|\mathbf{y}(T) - \mathbf{y}_N\|$  stays below a prescribed bound, cf. the discussion in the context of numerical quadrature.

However, an easier question can be answered by *asymptotic estimates* like (11.3.20):

What extra computational effort buys a prescribed *reduction of the error*?

(also recall the considerations in Section 2.3.3!)

The usual concept of “computational effort” for single step methods ( $\rightarrow$  Def. 11.3.5) is as follows

- Computational effort**  $\sim$  total number of  $\mathbf{f}$ -evaluations for approximately solving the IVP,  
 $\sim$  number of timesteps, if evaluation of discrete evolution  $\Psi^h$  ( $\rightarrow$  Def. 11.3.5) requires fixed number of  $\mathbf{f}$ -evaluations,  
 $\sim h^{-1}$ , in the case of uniform timestep size  $h > 0$  (equidistant mesh (11.3.17)).

Now, let us consider a single step method of order  $p \in \mathbb{N}$ , employed with a uniform timestep  $h_{\text{old}}$ . We focus on the maximal discretization error in the mesh points, see § 11.3.14. As in (5.3.48) we assume that the asymptotic error bounds are *sharp*:

$$\text{err}(h) \approx Ch^p \quad \text{for small meshwidth } h > 0 ,$$

with a “generic constant”  $C > 0$  independent of the mesh.

$$\begin{aligned} \text{Goal: } \frac{\text{err}(h_{\text{new}})}{\text{err}(h_{\text{old}})} &\stackrel{!}{=} \frac{1}{\rho} \quad \text{for reduction factor } \rho > 1 . \\ (11.3.20) \Rightarrow \frac{h_{\text{new}}^p}{h_{\text{old}}^p} &\stackrel{!}{=} \frac{1}{\rho} \Leftrightarrow h_{\text{new}} = \rho^{-1/p} h_{\text{old}} . \end{aligned}$$

For single step method of order  $p \in \mathbb{N}$

increase effort by factor  $\rho^{1/p}$  reduce error by factor  $\rho > 1$

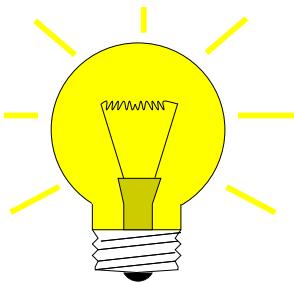
☞ the larger the order  $p$ , the less effort for a prescribed reduction of the error!

We remark that another (minor) rationale for using higher-order methods [15, Sect. 11.5.3]: curb impact of roundoff errors ( $\rightarrow$  Section 1.5.3) accumulating during timestepping.

### (11.4.2) Bootstrap construction of explicit single step methods

Now we will build a class of methods that are explicit and achieve orders  $p > 2$ . The starting point is a simple *integral equation* satisfied by any solution  $t \mapsto \mathbf{y}(t)$  of an initial value problems for the ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ :

$$\text{IVP: } \begin{aligned} \dot{\mathbf{y}}(t) &= \mathbf{f}(t, \mathbf{y}(t)) , & \Rightarrow & \mathbf{y}(t_1) = \mathbf{y}_0 + \int_{t_0}^{t_1} \mathbf{f}(\tau, \mathbf{y}(\tau)) d\tau \end{aligned}$$



Idea: approximate the integral by means of  $s$ -point quadrature formula ( $\rightarrow$  Section 5.1, defined on the reference interval  $[0, 1]$ ) with nodes  $c_1, \dots, c_s$ , weights  $b_1, \dots, b_s$ .

$$\mathbf{y}(t_1) \approx \mathbf{y}_1 = \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{f}(t_0 + c_i h, \boxed{\mathbf{y}(t_0 + c_i h)}) , \quad h := t_1 - t_0 . \quad (11.4.3)$$

Obtain these values by **bootstrapping**

“Bootstrapping” = use the same idea in a simpler version to get  $\mathbf{y}(t_0 + c_i h)$ , noting that these values can be replaced by other approximations obtained by methods already constructed (this approach will be elucidated in the next example).

What error can we afford in the approximation of  $\mathbf{y}(t_0 + c_i h)$  (under the assumption that  $\mathbf{f}$  is Lipschitz continuous)? We take the cue from the considerations in § 11.3.22.

Goal: aim for one-step error bound  $\mathbf{y}(t_1) - \mathbf{y}_1 = O(h^{p+1})$

Note that there is a factor  $h$  in front of the quadrature sum in (11.4.3). Thus, our goal can already be achieved, if only

$\mathbf{y}(t_0 + c_i h)$  is approximated up to an error  $O(h^p)$ ,

again, because in (11.4.3) a factor of size  $h$  multiplies  $\mathbf{f}(t_0 + c_i, \mathbf{y}(t_0 + c_i h))$ .

This is accomplished by a less accurate discrete evolution than the one we are about to build. Thus, we can construct discrete evolutions of higher and higher order, in turns, starting with the explicit Euler method. All these methods will be **explicit**, that is,  $\mathbf{y}_1$  can be computed directly from point values of  $\mathbf{f}$ .

#### Example 11.4.4 (Simple Runge-Kutta methods by quadrature & bootstrapping)

Now we apply the bootstrapping idea outlined above. We write  $\mathbf{k}_\ell \in \mathbb{R}^d$  for the approximations of  $\mathbf{y}(t_0 + c_i h)$ .

- Quadrature formula = trapezoidal rule (5.2.5):

$$Q(f) = \frac{1}{2}(f(0) + f(1)) \leftrightarrow s = 2: \quad c_1 = 0, c_2 = 1, \quad b_1 = b_2 = \frac{1}{2}, \quad (11.4.5)$$

and  $\mathbf{y}(t_1)$  approximated by explicit Euler step (11.2.7)

$$\mathbf{k}_1 = \mathbf{f}(t_0, \mathbf{y}_0), \quad \mathbf{k}_2 = \mathbf{f}(t_0 + h, \mathbf{y}_0 + h\mathbf{k}_1), \quad \mathbf{y}_1 = \mathbf{y}_0 + \frac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2). \quad (11.4.6)$$

(11.4.6) = **explicit trapezoidal method** (for numerical integration of ODEs).

- Quadrature formula  $\rightarrow$  simplest Gauss quadrature formula = midpoint rule ( $\rightarrow$  Ex. 5.2.3) &  $\mathbf{y}(\frac{1}{2}(t_1 + t_0))$  approximated by explicit Euler step (11.2.7)

$$\mathbf{k}_1 = \mathbf{f}(t_0, \mathbf{y}_0), \quad \mathbf{k}_2 = \mathbf{f}(t_0 + \frac{h}{2}, \mathbf{y}_0 + \frac{h}{2}\mathbf{k}_1), \quad \mathbf{y}_1 = \mathbf{y}_0 + h\mathbf{k}_2. \quad (11.4.7)$$

(11.4.7) = **explicit midpoint method** (for numerical integration of ODEs) [15, Alg. 11.18].

### Example 11.4.8 (Convergence of simple Runge-Kutta methods)

We perform an empiric study of the order of the explicit single step methods constructed in Ex. 11.4.4.

- ✿ IVP:  $\dot{y} = 10y(1 - y)$  (logistic ODE (11.1.6)),  $y(0) = 0.01$ ,  $T = 1$ ,
- ✿ Explicit single step methods, uniform timestep  $h$ .

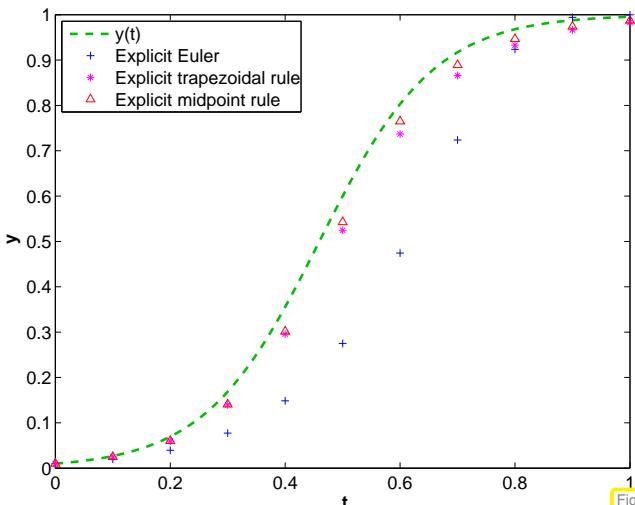


Fig. 378

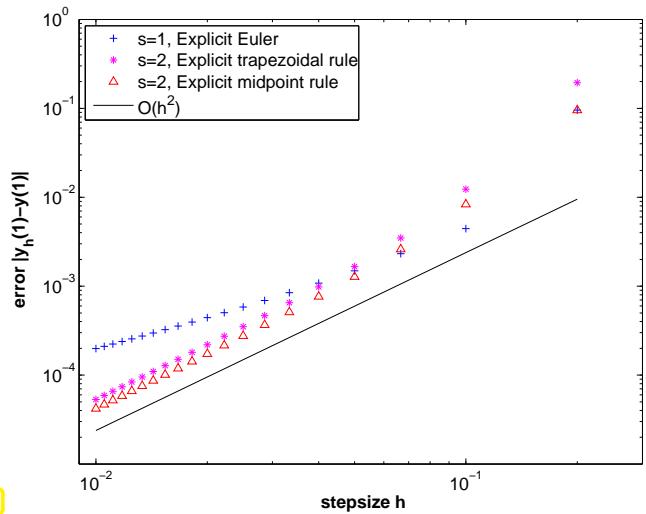


Fig. 379

$y_h(j/10)$ ,  $j = 1, \dots, 10$  for explicit RK-methods

Errors at final time  $y_h(1) - y(1)$

Observation: obvious *algebraic convergence* in meshwidth  $h$  with integer rates/orders:

explicit trapezoidal rule (11.4.6) → order 2  
explicit midpoint rule (11.4.7) → order 2

This is what one expects from the considerations in Ex. 11.4.4.

The formulas that we have obtained follow a general pattern:

#### Definition 11.4.9. Explicit Runge-Kutta method

For  $b_i, a_{ij} \in \mathbb{R}$ ,  $c_i := \sum_{j=1}^{i-1} a_{ij}$ ,  $i, j = 1, \dots, s$ ,  $s \in \mathbb{N}$ , an  $s$ -stage explicit Runge-Kutta single step method (RK-SSM) for the ODE  $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ ,  $\mathbf{f} : \Omega \rightarrow \mathbb{R}^d$ , is defined by ( $\mathbf{y}_0 \in D$ )

$$\mathbf{k}_i := \mathbf{f}\left(t_0 + c_i h, \mathbf{y}_0 + h \sum_{j=1}^{i-1} a_{ij} \mathbf{k}_j\right), \quad i = 1, \dots, s, \quad \mathbf{y}_1 := \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{k}_i.$$

The vectors  $\mathbf{k}_i \in \mathbb{R}^d$ ,  $i = 1, \dots, s$ , are called **increments**,  $h > 0$  is the size of the timestep.

Recall Rem. 11.3.12 to understand how the discrete evolution for an explicit Runge-Kutta method is specified in this definition by giving the formulas for the first step. This is a convention widely adopted in the literature about numerical methods for ODEs. Of course, the increments  $\mathbf{k}_i$  have to be computed anew in each timestep.

The implementation of an  $s$ -stage explicit Runge-Kutta single step method according to Def. 11.4.9 is straightforward: The increments  $\mathbf{k}_i \in \mathbb{R}^d$  are computed successively, starting from  $\mathbf{k}_1 = \mathbf{f}(t_0 + c_1 h, \mathbf{y}_0)$ .

- Only  $s$   $\mathbf{f}$ -evaluations and AXPY operations ( $\rightarrow$  Section 1.3.2) are required.

### Butcher scheme notation for explicit RK-SSM

Shorthand notation for (explicit) Runge-Kutta methods [15, (11.75)]  
 Butcher scheme  
 (Note:  $\mathfrak{A}$  is strictly lower triangular  $s \times s$ -matrix)

$$\begin{array}{c|ccccc} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T & := & c_1 & 0 & \cdots & 0 \\ & & & c_2 & a_{21} & \ddots & \vdots \\ & & & \vdots & \vdots & \ddots & \vdots \\ & & & c_s & a_{s1} & \cdots & a_{s,s-1} & 0 \\ \hline & & b_1 & \cdots & b_{s-1} & b_s & & \end{array} \quad (11.4.11)$$

Note that in Def. 11.4.9 the coefficients  $b_i$  can be regarded as weights of a quadrature formula on  $[0, 1]$ : apply explicit Runge-Kutta single step method to “ODE”  $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ . The quadrature rule with these weights and nodes  $c_j$  will have order  $\geq 1$ , if the weights add up to 1!

### Corollary 11.4.12. Consistent Runge-Kutta single step methods

A Runge-Kutta single step method according to Def. 11.4.9 is *consistent* ( $\rightarrow$  Def. 11.3.10) with the ODE  $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ , if and only if

$$\sum_{i=1}^s b_i = 1 .$$

### Example 11.4.13 (Butcher schemes for some explicit RK-SSM [15, Sect. 11.6.1])

The following explicit Runge-Kutta single step methods are often mentioned in literature.

- Explicit Euler method (11.2.7):

$$\begin{array}{c|cc} 0 & 0 \\ \hline 1 & \end{array} \quad > \quad \text{order} = 1$$

- explicit trapezoidal rule (11.4.6):

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{array} \quad > \quad \text{order} = 2$$

- explicit midpoint rule (11.4.7):

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline 0 & 0 & 1 \end{array} \quad > \quad \text{order} = 2$$

- Classical 4th-order RK-SSM:

0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
$\frac{1}{2}$	0	$\frac{1}{2}$	0	0
1	0	0	1	0
	$\frac{1}{6}$	$\frac{2}{6}$	$\frac{2}{6}$	$\frac{1}{6}$

order = 4

- Kutta's 3/8-rule:

0	0	0	0	0
$\frac{1}{3}$	$\frac{1}{3}$	0	0	0
$\frac{2}{3}$	$-\frac{1}{3}$	1	0	0
1	1	-1	1	0
	$\frac{1}{8}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{1}{8}$

order = 4

#### Remark 11.4.14 (Construction of higher order Runge-Kutta single step methods)

Runge-Kutta single step methods of order  $p > 2$  are not found by bootstrapping as in Ex. 11.4.4, because the resulting methods would have quite a lot of stages compared to their order.

Rather one derives **order conditions** yielding large non-linear systems of equations for the coefficients  $a_{ij}$  and  $b_i$  in Def. 11.4.9, see [18, Sect. 4.2.3] and [38, Ch. III]. This approach is similar to the construction of a Gauss quadrature rule in Ex. 5.3.13. Unfortunately, the systems of equations are very difficult to solve and no universal recipe is available. Nevertheless, through massive use of symbolic computation, Runge-Kutta methods of order up to 19 have been constructed in this way.

#### Remark 11.4.15 (“Butcher barriers” for explicit RK-SSM)

The following table gives lower bounds for the number of stages needed to achieve order  $p$  for an explicit Runge-Kutta method.

order $p$	1	2	3	4	5	6	7	8	$\geq 9$
minimal no. $s$ of stages	1	2	3	4	6	7	9	11	$\geq p + 3$

No general formula has been discovered. What is known is that for explicit Runge-Kutta single step methods according to Def. 11.4.9

$$\text{order } p \leq \text{number } s \text{ of stages of RK-SSM}$$

#### Remark 11.4.16 (Explicit ODE integrator in MATLAB)

MATLAB provides a built-in numerical integrator based on explicit RK-SSM, see [72] and [7, Sect. 7.2]. Its calling syntax is

 $[t, y] = \text{ode45}(\text{odefun}, \text{tspan}, y_0);$

`odefun` : Handle to a function of type  $\mathbf{f}(t, \mathbf{y}) \leftrightarrow$  r.h.s.  $\mathbf{f}(t, \mathbf{y})$   
`tspan` : vector  $[t_0, T]^T$ , initial and final time for numerical integration  
`y0` : (vector) passing initial state  $\mathbf{y}_0 \in \mathbb{R}^d$

Return values:

$\mathbf{t}$  : temporal mesh  $\{t_0 < t_1 < t_2 < \dots < t_{N-1} = t_N = T\}$   
 $\mathbf{y}$  : sequence  $(\mathbf{y}_k)_{k=0}^N$  (column vectors)

### MATLAB-code 11.4.17: Code excerpts from MATLAB's integrator `ode45`

```

1 function varargout = ode45(ode,tspan,y0,options,varargin)
2 % Processing of input parameters omitted
3 %
4 % Initialize method parameters, c.f. Butcher scheme (11.4.11)
5 pow = 1/5;
6 A = [1/5, 3/10, 4/5, 8/9, 1, 1];
7 B = [
8     1/5          3/40    44/45   19372/6561   9017/3168   35/384
9     0            9/40    -56/15  -25360/2187  -355/33      0
10    0            0       32/9    64448/6561   46732/5247
11    500/1113
12    0            0       0       -212/729    49/176      125/192
13    0            0       0       0           -5103/18656
14    -2187/6784
15    0            0       0       0           0           11/84
16    0            0       0       0           0           0
17 ];
18 % : (choice of stepsize and main loop omitted)
19 % ADVANCING ONE STEP.
20 hA = h * A;
21 hB = h * B;
22 f(:,2) = feval(odeFcn,t+hA(1),y+f*hB(:,1),odeArgs{:});
23 f(:,3) = feval(odeFcn,t+hA(2),y+f*hB(:,2),odeArgs{:});
24 f(:,4) = feval(odeFcn,t+hA(3),y+f*hB(:,3),odeArgs{:});
25 f(:,5) = feval(odeFcn,t+hA(4),y+f*hB(:,4),odeArgs{:});
26 f(:,6) = feval(odeFcn,t+hA(5),y+f*hB(:,5),odeArgs{:});
27
28 tnew = t + hA(6);
29 if done, tnew = tfinal; end % Hit end point exactly.
30 h = tnew - t;           % Purify h.
31 ynew = y + f*hB(:,6);
32 % : (stepsize control, see Sect. 11.5 dropped)

```

### Example 11.4.18 (Numerical integration of logistic ODE in MATLAB)

This example demonstrates the use of `ode45` for a scalar ODE ( $d = 1$ )

MATLAB-CODE: usage of `ode45`

```
fn = @(t,y) 5*y*(1-y);
[t,y] = ode45(fn,[0 1.5],y0);
plot(t,y,'r-');
```

MATLAB-integrator: `ode45()`:

Handle passing r.h.s. function  $\mathbf{f} = \mathbf{f}(t, y)$ ,  
 initial and final time as row vector,  
 initial state  $\mathbf{y}_0$ , as column vector,

## 11.5 Adaptive Stepsize Control

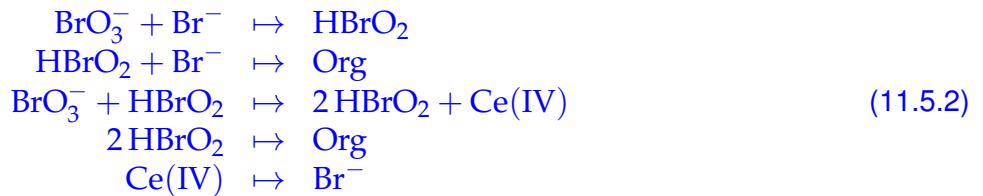


*Supplementary reading.* [15, Sect. 11.7], [63, Sect. 11.8.2]

### Example 11.5.1 (Oregonator reaction)

Chemical reaction kinetics is a field where ODE based models are very common. This example presents a famous reaction with extremely abrupt dynamics. Refer to [42, Ch. 62] for more information about the ODE-based modelling of kinetics of chemical reactions.

This is a special case of an “oscillating” Zhabotinski-Belousov reaction [30]:



$$\begin{aligned}
 y_1 := c(\text{BrO}_3^-): \quad \dot{y}_1 &= -k_1 y_1 y_2 - k_3 y_1 y_3, \\
 y_2 := c(\text{Br}^-): \quad \dot{y}_2 &= -k_1 y_1 y_2 - k_2 y_2 y_3 + k_5 y_5, \\
 y_3 := c(\text{HBrO}_2): \quad \dot{y}_3 &= k_1 y_1 y_2 - k_2 y_2 y_3 + k_3 y_1 y_3 - 2k_4 y_3^2, \\
 y_4 := c(\text{Org}): \quad \dot{y}_4 &= k_2 y_2 y_3 + k_4 y_3^2, \\
 y_5 := c(\text{Ce(IV)}): \quad \dot{y}_5 &= k_3 y_1 y_3 - k_5 y_5,
 \end{aligned} \tag{11.5.3}$$

with (non-dimensionalized) reaction constants:

$$k_1 = 1.34, \quad k_2 = 1.6 \cdot 10^9, \quad k_3 = 8.0 \cdot 10^3, \quad k_4 = 4.0 \cdot 10^7, \quad k_5 = 1.0.$$



periodic chemical reaction ➔ [Video 1](#), [Video 2](#)

MATLAB simulation with initial state  $y_1(0) = 0.06$ ,  $y_2(0) = 0.33 \cdot 10^{-6}$ ,  $y_3(0) = 0.501 \cdot 10^{-10}$ ,  $y_4(0) = 0.03$ ,  $y_5(0) = 0.24 \cdot 10^{-7}$ :

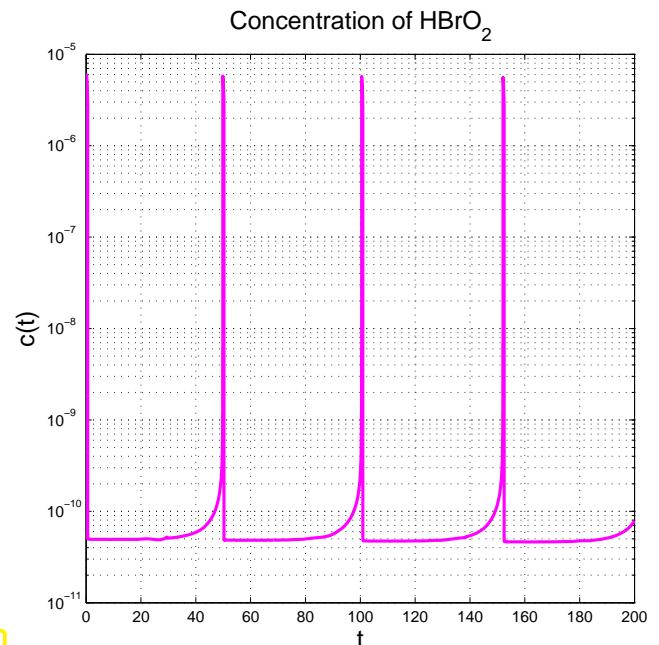
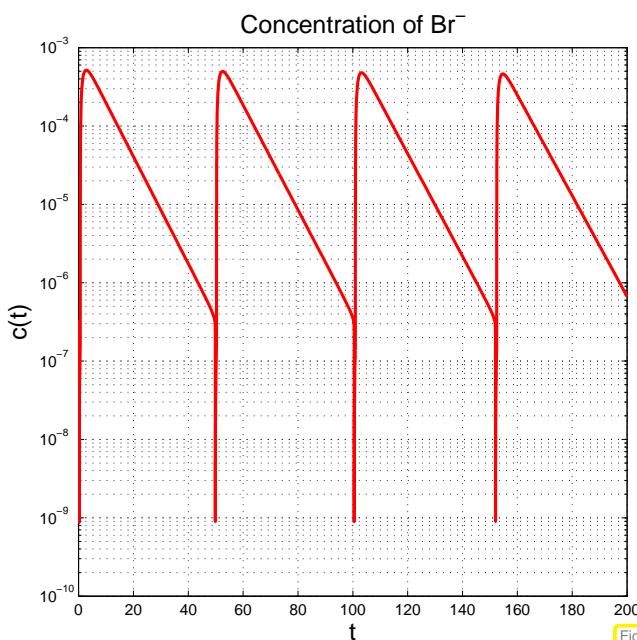


Fig. 380

Fig. 381

We observe a strongly non-uniform behavior of the solution in time.

This is very common with evolutions arising from practical models (circuit models, chemical reaction models, mechanical systems)

### Example 11.5.4 (Blow-up)

We return to the “explosion ODE” of Ex. 11.1.35 and consider the scalar autonomous IVP:

$$\dot{y} = y^2, \quad y(0) = y_0 > 0.$$

►  $y(t) = \frac{y_0}{1 - y_0 t}, \quad t < 1/y_0.$

As we have seen a solution exists only for finite time and then suffers a **Blow-up**, that is,  $\lim_{t \rightarrow 1/y_0} y(t) = \infty$   
:  $J(y_0) = ]-\infty, 1/y_0]$ !

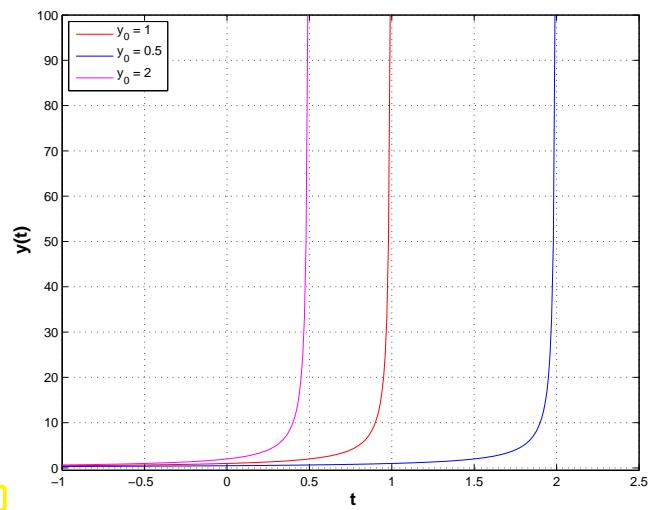
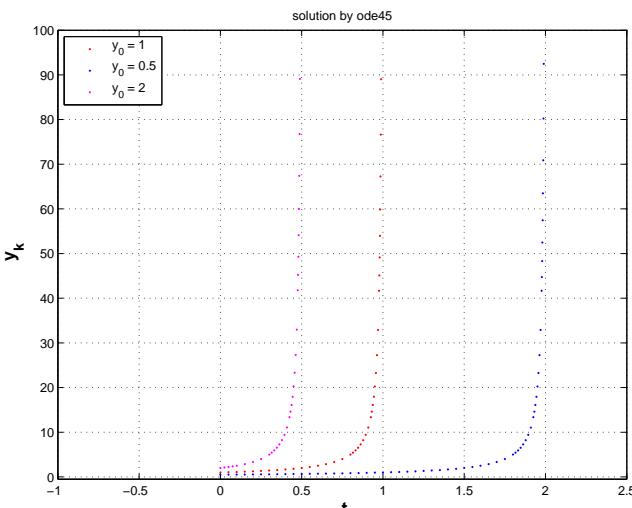


Fig. 382

How to choose temporal mesh  $\{t_0 < t_1 < \dots < t_{N-1} < t_N\}$  for single step method in case  $J(y_0)$  is not known, even worse, if it is not clear a priori that a blow up will happen?

Just imagine: what will result from equidistant explicit Euler integration (11.2.7) applied to the above IVP?



### MATLAB warning messages:

Warning: Failure at  $t=9.999694e-01$ . Unable to meet integration tolerances without reducing the step size below the smallest value allowed ( $1.776357e-15$ ) at time  $t$ .  
> In ode45 at 371  
  In simpleblowup at 22

Warning: Failure at  $t=1.999970e+00$ . Unable to meet integration tolerances without reducing the step size below the smallest value allowed ( $3.552714e-15$ ) at time  $t$ .  
> In ode45 at 371  
  In simpleblowup at 23

Warning: Failure at  $t=4.999660e-01$ . Unable to meet integration tolerances without reducing the step size below the smallest value allowed ( $8.881784e-16$ ) at time  $t$ .  
> In ode45 at 371  
  In simpleblowup at 24

We observe: `ode45` manages to reduce stepsize more and more as it approaches the singularity of the solution! How can it accomplish this feat!

Key challenge (discussed for autonomous ODEs below):

How to choose a *good temporal mesh*  $\{0 = t_0 < t_1 < \dots < t_{N-1} < t_N\}$   
for a given single step method applied to a concrete IVP?

What does “good” mean ?

Be efficient!

Be accurate!

### Stepsize adaptation for single step methods

*Objective:*  $N$  as small as possible &  $\max_{k=1,\dots,N} \|\mathbf{y}(t_k) - \mathbf{y}_k\| < \text{TOL}$ ,  $\text{TOL} = \text{tolerance}$   
 or  $\|\mathbf{y}(T) - \mathbf{y}_N\| < \text{TOL}$

*Policy:* Try to curb/balance one-step error by

- \* adjusting current stepsize  $h_k$ ,
- \* predicting suitable next timestep  $h_{k+1}$

*Tool:* Local-in-time one-step error estimator (*a posteriori*, based on  $\mathbf{y}_k, h_{k-1}$ )

local-in-time  
stepsize control

Why local-in-time timestep control (based on estimating only the one-step error)?

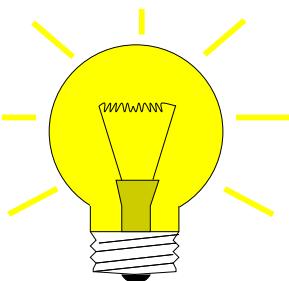
Consideration: If a small time-local error in a single timestep leads to large error  $\|\mathbf{y}_k - \mathbf{y}(t_k)\|$  at later times, then local-in-time timestep control is powerless about it and will not even notice!!

Nevertheless, local-in-time timestep control is used almost exclusively,

- ☞ because we do not want to discard past timesteps, which could amount to tremendous waste of computational resources,
- ☞ because it is inexpensive and it works for many practical problems,
- ☞ because there is no reliable method that can deliver guaranteed accuracy for general IVP.

### (11.5.7) Local-in-time error estimation

We “recycle” heuristics already employed for adaptive quadrature, see Section 5.5, § 5.5.10. There we tried to get an idea of the local quadrature error by comparing two approximations of different order. Now we pursue a similar idea over a single timestep.



Idea:

*Estimation of one-step error*

Compare results for two discrete evolutions  $\Psi^h, \tilde{\Psi}^h$  of different order over current timestep  $h$ :

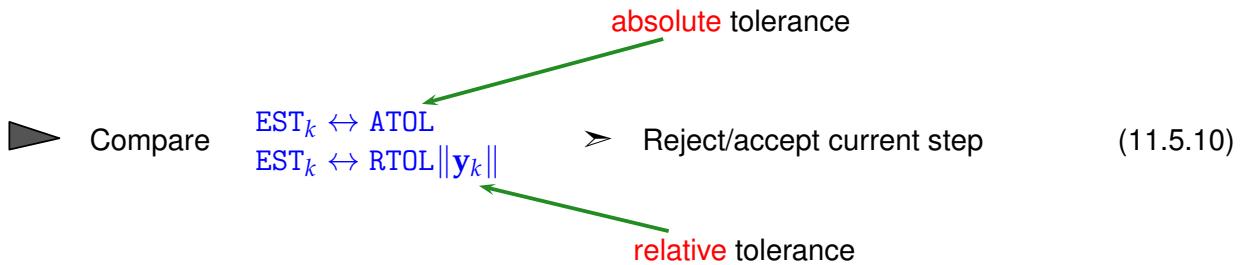
If  $\text{Order}(\tilde{\Psi}) > \text{Order}(\Psi)$ , then we expect

$$\underbrace{\Phi^h \mathbf{y}(t_k) - \Psi^h \mathbf{y}(t_k)}_{\text{one-step error}} \approx \text{EST}_k := \tilde{\Psi}^h \mathbf{y}(t_k) - \Psi^h \mathbf{y}(t_k). \quad (11.5.8)$$

Heuristics for concrete  $h$

### (11.5.9) Temporal mesh refinement

We take for granted a local error estimate  $\text{EST}_k$ .



For a similar use of absolute and relative tolerances see Section 2.1.2: termination criteria for iterations, in particular (2.1.25).

☞ Simple algorithm:

$\text{EST}_k < \max\{\text{ATOL}, \|\mathbf{y}_k\| \text{RTOL}\}$ : Carry out next timestep (stepsize  $h$ )  
Use larger stepsize (e.g.,  $\alpha h$  with some  $\alpha > 1$ ) for following step (\*)

$\text{EST}_k > \max\{\text{ATOL}, \|\mathbf{y}_k\| \text{RTOL}\}$ : Repeat current step with smaller stepsize  $< h$ , e.g.,  $\frac{1}{2}h$

Rationale for (\*): if the current stepsize guarantees sufficiently small one-step error, then it might be possible to obtain a still acceptable one-step error with a larger timestep, which would enhance efficiency (fewer timesteps for total numerical integration). This should be tried, since timestep control will usually provide a safeguard against undue loss of accuracy.

#### MATLAB-code 11.5.11: Simple local stepsize control for single step methods

```

1 function [t,y] =
2     odeintadapt(Psilow,Psihigh,T,y0,h0,reltol,abstol,hmin)
3 t = 0; y = y0; h = h0; %%
4 while ((t(end) < T) && (h > hmin)) %
5     yh = Psihigh(h,y0); % high order discrete evolution  $\tilde{\Psi}^h$ 
6     yH = Psilow(h,y0); % low order discrete evolution  $\Psi^h$ 
7     est = norm(yH-yh); %<-> EST_k
8
9     if (est < max(reltol*norm(y0),abstol)) %
10        y0 = yh; y = [y,y0]; t = [t,t(end) + min(T-t(end),h)]; %
11        h = 1.1*h; % step accepted, try with increased stepsize
12    else, h = h/2; end % step rejected, try with half the stepsize
13 end

```

Comments on Code 11.5.11:

- Input arguments:
  - $\text{Psilow}$ ,  $\text{Psihigh}$ : function handles to discrete evolution operators for autonomous ODE of different order, type  $@(\mathbf{y}, h)$ , expecting a state (column) vector as first argument, and a stepsize as second,
  - $T$ : final time  $T > 0$ ,
  - $\mathbf{y}_0$ : initial state  $\mathbf{y}_0$ ,

- $h_0$ : stepsize  $h_0$  for the first timestep
- $\text{reltol}, \text{abstol}$ : relative and absolute tolerances, see (11.5.10),
- $h_{\min}$ : minimal stepsize, timestepping terminates when stepsize control  $h_k < h_{\min}$ , which is relevant for detecting blow-ups or collapse of the solution.
- line 3: check whether final time is reached or timestepping has ground to a halt ( $h_k < h_{\min}$ ).
- line 4, 5: advance state by low and high order integrator.
- line 6: compute norm of estimated error, see (11.5.8).
- line 8: make comparison (11.5.10) to decide whether to accept or reject local step.
- line 9, 10: step accepted, update state and current time and suggest 1.1 times the current stepsize for next step.
- line 11 step rejected, try again with half the stepsize.
- Return values:
  - $t$ : temporal mesh  $t_0 < t_1 < t_2 < \dots < t_N < T$ , where  $t_N < T$  indicated premature termination (collapse, blow-up),
  - $y$ : sequence  $(y_k)_{k=0}^N$ .

### Remark 11.5.12 (Estimation of “wrong” error?)

We face the same conundrum as in the case of adaptive numerical quadrature, see Rem. 5.5.16:

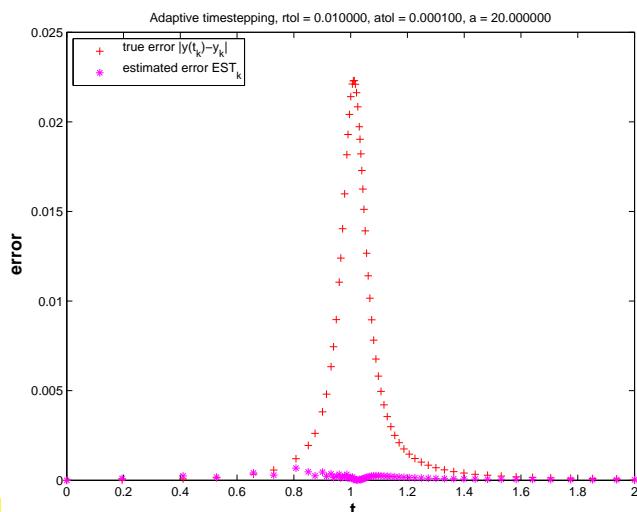
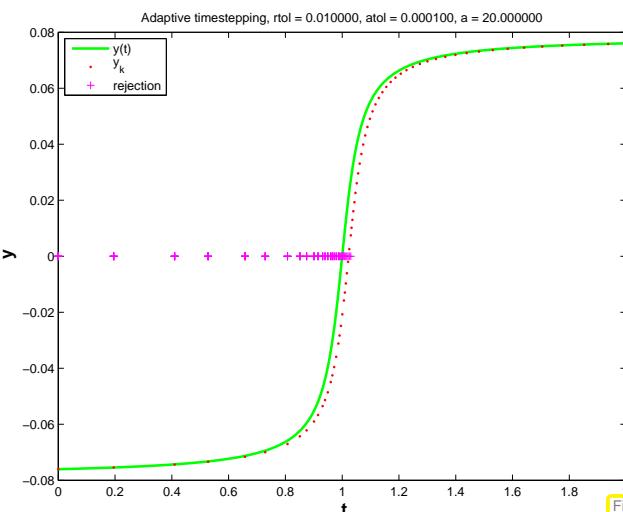
! By the heuristic considerations, see (11.5.8) it seems that  $\text{EST}_k$  measures the one-step error for the low-order method  $\Psi$  and that we should use  $y_{k+1} = \Psi^{h_k} y_k$ , if the timestep is accepted.

However, it would be foolish not to use the better value  $y_{k+1} = \tilde{\Psi}^{h_k} y_k$ , since it is available for free. This is what is done in every implementation of adaptive methods, also in Code 11.5.11, and this choice can be justified by control theoretic arguments [18, Sect. 5.2].

### Example 11.5.13 (Simple adaptive stepsize control)

We test adaptive timestepping routine from Code 11.5.11 for a scalar IVP and compare the estimated local error and true local error.

- \* IVP for ODE  $\dot{y} = \cos(\alpha y)^2$ ,  $\alpha > 0$ , solution  $y(t) = \arctan(\alpha(t - c))/\alpha$  for  $y(0) \in ]-\pi/2, \pi/2[$
- \* Simple adaptive timestepping based on explicit Euler (11.2.7) and explicit trapezoidal rule (11.4.6)



Statistics: 66 timesteps, 131 rejected timesteps

Observations:

- ☞ Adaptive timestepping well resolves local features of solution  $y(t)$  at  $t = 1$
- ☞ Estimated error (an estimate for the one-step error) and true error are **not** related! To understand this recall Rem. 11.5.12.

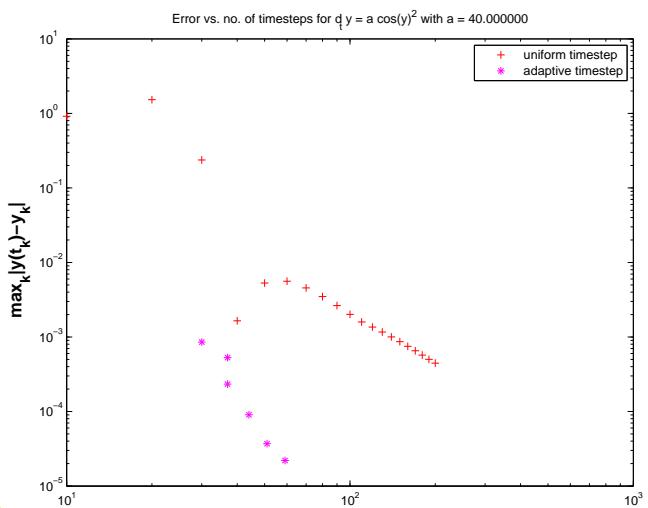
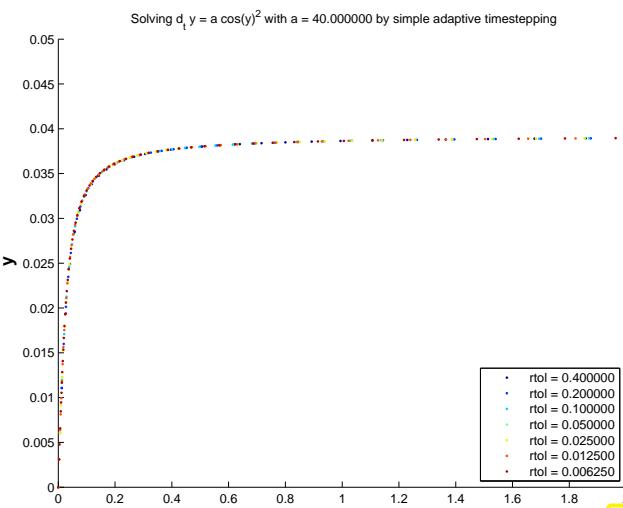
### Example 11.5.14 (Gain through adaptivity → Ex. 11.5.13)

In this experiment we want to explore whether adaptive timestepping is worth while, as regards reduction of computational effort without sacrificing accuracy.

We retain the simple adaptive timestepping from previous experiment Ex. 11.5.13 and also study the same IVP.

New: initial state  $y(0) = 0$ !

Now we examine the dependence of the maximal discretization error in mesh points on the computational effort. The latter is proportional to the number of timesteps.



Solutions  $(y_k)_k$  for different values of  $rtol$

Error vs. computational effort

Observations:

- ☞ Adaptive timestepping achieves much better accuracy for a fixed computational effort.

### Example 11.5.15 (“Failure” of adaptive timestepping → Ex. 11.5.14)

Same ODE and simple adaptive timestepping as in previous experiment Ex. 11.5.14.

$$\dot{y} = \cos^2(\alpha y) \Rightarrow y(t) = \arctan(\alpha(t - c)) / \alpha, y(0) \in ] - \frac{\pi}{2\alpha}, -\frac{\pi}{2\alpha} [ ,$$

for  $\alpha = 40$ .

Now: initial state  $y(0) = -0.0386 \approx \frac{\pi}{2\alpha}$  as in Ex. 11.5.13

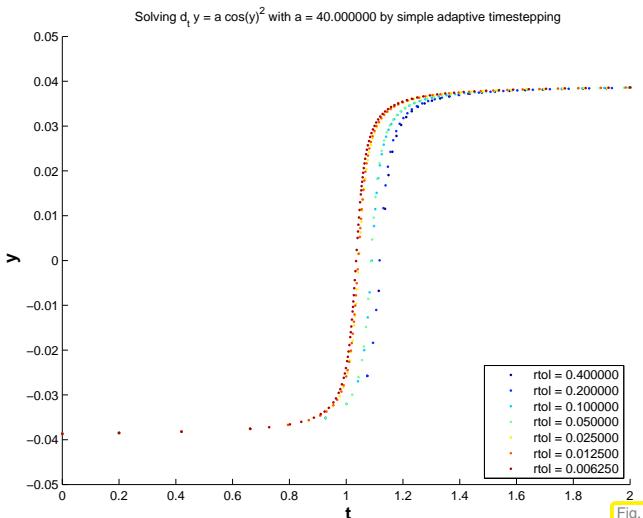
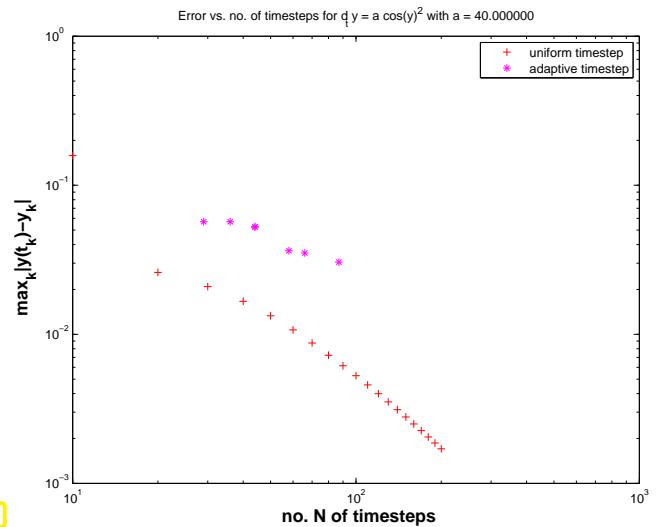


Fig. 388



Error vs. computational effort

Solutions  $(y_k)_k$  for different values of  $rtol$

Observations:

- ☞ Adaptive timestepping leads to larger errors at the same computational cost as uniform timestepping !

Explanation: the position of the steep step of the solution has a sensitive dependence on an initial value, if  $y(0) \approx \frac{\pi}{2\alpha}$ :

$$y(t) = \frac{1}{\alpha} \arctan(\alpha(t + \tan(y_0/\alpha))) , \text{ step at } \approx -\tan(y_0/\alpha) .$$

Hence, small local errors in the initial timesteps will lead to large errors at around time  $t \approx 1$ . The stepsize control is mistaken in condoning these small one-step errors in the first few steps and, therefore, incurs huge errors later.

However, the perspective of **backward error analysis** (→ § 1.5.79) rehabilitates adaptive stepsize control in this case: it gives us a numerical solution that is very close to the exact solution of the ODE with slightly perturbed initial state  $y_0$ .

### Remark 11.5.16 (Refined local stepsize control → [15, Sect. 11.7])

The above algorithm (Code 11.5.11) is simple, but the rule for increasing/shrinking of timestep “squanders” the information contained in  $\text{EST}_k : \text{TOL}$ :

More ambitious goal !	When $\text{EST}_k > \text{TOL}$ : stepsize <b>adjustment</b> better $h_k = ?$
	When $\text{EST}_k < \text{TOL}$ : stepsize <b>prediction</b> good $h_{k+1} = ?$

Assumption: At our disposal are two discrete evolutions:

- \*  $\Psi$  with  $\text{order}(\Psi) = p$  ( $\rightarrow$  “low order” single step method)
- \*  $\tilde{\Psi}$  with  $\text{order}(\tilde{\Psi}) > p$  ( $\rightarrow$  “higher order” single step method)

These are the same building blocks as for the simple adaptive strategy employed in Code 11.5.11 (passed as arguments  $\text{Psilow}, \text{Psihigh}$  there).

Asymptotic expressions for one-step error for  $h \rightarrow 0$ :

$$\begin{aligned}\Psi^{h_k} \mathbf{y}(t_k) - \Phi^{h_k} \mathbf{y}(t_k) &= ch^{p+1} + O(h_k^{p+2}), \\ \tilde{\Psi}^{h_k} \mathbf{y}(t_k) - \Phi^{h_k} \mathbf{y}(t_k) &= O(h_k^{p+2}),\end{aligned}\tag{11.5.17}$$

with some (unknown)  $c > 0$ .

Why  $h^{p+1}$ ? Remember estimate (11.3.28) from the error analysis of the explicit Euler method: we also found  $O(h^2)$  there for the one-step error of a single step method of order 1.

Heuristics: the timestep  $h_k$  is small  $\rightarrow$  “higher order terms”  $O(h_k^{p+2})$  can be ignored.

$$\begin{aligned}\Psi^{h_k} \mathbf{y}(t_k) - \Phi^{h_k} \mathbf{y}(t_k) &\doteq ch_k^{p+1} + O(h_k^{p+2}), \\ \tilde{\Psi}^{h_k} \mathbf{y}(t_k) - \Phi^{h_k} \mathbf{y}(t_k) &\doteq O(h_k^{p+2}).\end{aligned}\Rightarrow \boxed{\text{EST}_k \doteq ch_k^{p+1}}.\tag{11.5.18}$$

☞ notation:  $\doteq$  equality up to higher order terms in  $h_k$

$$\text{EST}_k \doteq ch_k^{p+1} \Rightarrow c \doteq \frac{\text{EST}_k}{h_k^{p+1}}.\tag{11.5.19}$$

Available in algorithm, see (11.5.8)

For the sake of *accuracy* (stipulates “ $\text{EST}_k < \text{TOL}$ ”) & *efficiency* (favors “ $>$ ”) we aim for

$$\text{EST}_k \stackrel{!}{=} \text{TOL} := \max\{\text{ATOL}, \|\mathbf{y}_k\| \text{RTOL}\}.\tag{11.5.20}$$

What timestep  $h_*$  can actually achieve (11.5.20), if we “believe” in (11.5.18) (and, therefore, in (11.5.19))?

$$(11.5.19) \& (11.5.20) \Rightarrow \text{TOL} = \frac{\text{EST}_k}{h_k^{p+1}} h_*^{p+1}.$$



“Optimal timestep”: (stepsize prediction)

$$h_* = h^{p+1} \sqrt{\frac{\text{TOL}}{\text{EST}_k}} . \quad (11.5.21)$$

adjusted stepsize (**R**)

suggested stepsize (**A**)

(Reject): In case  $\text{EST}_k > \text{TOL}$  ➤ repeat step with stepsize  $h_*$ .

(Accept): If  $\text{EST}_k \leq \text{TOL}$  ➤ use  $h_*$  as stepsize for next step.

#### MATLAB-code 11.5.22: Refined local stepsize control for single step methods

```

1 function [t,y] =
2   odeintssctrl(Psilow,p,Psihigh,T,y0,h0,reltol,abstol,hmin)
3   t = 0; y = y0; h = h0; % 
4   while ((t(end) < T) && (h > hmin)) %
5     yh = Psihigh(h,y0); % high order discrete evolution  $\tilde{\Psi}^h$ 
6     yH = Psilow(h,y0); % low order discrete evolution  $\Psi^h$ 
7     est = norm(yH-yh); %  $\leftrightarrow \text{EST}_k$ 
8
9     tol = max(reltol*norm(y(:,end)),abstol); %
10    h = h*max(0.5,min(2,(tol/est)(1/(p+1)))); % Optimal stepsize
11    % according to (11.5.21)
12    if (est < tol) %
13      y0 = yh; y = [y,y0]; t = [t,t(end) + min(T-t(end),h)]; % step
14      % accepted
15    end
16  end

```

Comments on Code 11.5.22 (see comments on Code 11.5.11 for more explanations):

- Input arguments as for Code 11.5.11, except for  $p \hat{=} \text{order of lower order discrete evolution}$ .
- line 9: compute presumably better local stepsize according to (11.5.21),
- line 10: decide whether to repeat the step or advance,
- line 11: extend output arrays if current step has not been rejected.

#### Remark 11.5.23 (Stepsize control in MATLAB)

The name of MATLAB's standard integrator `ode45` already indicates the orders of the pair of single step methods used for adaptive stepsize control:



Specifying tolerances for MATLAB's integrators is done as follows:

```

options = odeset('abstol',atol,'reltol',rtol,'stats','on');
[t,y] = ode45(@(t,x) f(t,x),tspan,y0,options);
(f = function handle, tspan  $\hat{=}$  [t0, T], y0  $\hat{=}$  y0, t  $\hat{=}$  tk, y  $\hat{=}$  yk)

```

The possibility to pass tolerances to numerical integrators based on adaptive timestepping may tempt one into believing that they allow to control the accuracy of the solutions. However, as is clear from Rem. 11.5.16, these tolerances are solely applied to local error estimates and, inherently, have nothing to do with global discretization errors, see Ex. 11.5.13.

### No global error control through local-in-time adaptive timestepping

The absolute/relative tolerances imposed for local-in-time adaptive timestepping do *not* allow to predict accuracy of solution!

### Remark 11.5.25 (Embedded Runge-Kutta methods)

For higher order RK-SSM with a considerable number of stages computing different sets of increments ( $\rightarrow$  Def. 11.4.9) for two methods of different order just for the sake of local-in-time stepsize control would mean incommensurate effort.

Embedding idea: Use two RK-SSMs based on the **same increments**, that is, built with the same coefficients  $a_{ij}$ , but different weights  $b_i$ , see Def. 11.4.9 for the formulas, and *different orders*  $p$  and  $p + 1$ .

Butcher scheme for **embedded explicit Runge-Kutta methods**

(Lower order scheme has weights  $\widehat{b}_i$ .)

$$\triangleright \quad \begin{array}{c|c} c & \alpha \\ \hline b^T & \\ \hline \widehat{b}^T & \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline b_1 & \cdots & b_s \\ \hline \widehat{b}_1 & \cdots & \widehat{b}_s \end{array} .$$

### Example 11.5.26 (Commonly used embedded explicit Runge-Kutta methods)

The following two embedded RK-SSM, presented in the form of their extended Butcher schemes, provided single step methods of orders 4 & 5.

$0$					
$\frac{1}{3}$	$\frac{1}{3}$				
$\frac{1}{3}$	$\frac{1}{6}$	$\frac{1}{6}$			
$\frac{1}{2}$	$\frac{1}{8}$	$0$		$\frac{3}{8}$	
$1$	$\frac{1}{2}$	$0$	$-\frac{3}{2}$	$2$	
$y_1$	$\frac{1}{6}$	$0$	$0$	$\frac{2}{3}$	$\frac{1}{6}$
$\hat{y}_1$	$\frac{1}{10}$	$0$	$\frac{3}{10}$	$\frac{2}{5}$	$\frac{1}{5}$

Merson's embedded RK-SSM

$0$	$\frac{1}{2}$
$\frac{1}{2}$	$\frac{1}{2}$
$\frac{1}{2}$	$0$
$1$	$0$
$\frac{3}{4}$	$\frac{5}{32}$
$y_1$	$\frac{1}{6}$
$\widehat{y}_1$	$-\frac{1}{2}$

Fehlberg's embedded RK-SSM

### Example 11.5.27 (Adaptive timestepping for mechanical problem)

We test the effect of adaptive stepsize control in MATLAB for the equations of motion describing the planar movement of a point mass in a conservative force field  $\mathbf{x} \in \mathbb{R}^2 \mapsto \mathbf{F}(\mathbf{x}) \in \mathbb{R}^2$ : Let  $t \mapsto \mathbf{y}(t) \in \mathbb{R}^2$  be the trajectory of point mass (in the plane).

From Newton's law:  $\ddot{\mathbf{y}} = F(\mathbf{y}) := -\frac{2\mathbf{y}}{\|\mathbf{y}\|_2^2}$ . (11.5.28)

As in Rem. 11.1.23 we can convert the second-order ODE (11.5.28) into an equivalent 1st-order ODE by introducing the **velocity**  $v := \dot{y}$  as an extra solution component:

$$(11.5.28) \quad \Rightarrow \quad \begin{bmatrix} \dot{\mathbf{y}} \\ \dot{\mathbf{v}} \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ -\frac{2\mathbf{y}}{\|\mathbf{y}\|_2^2} \end{bmatrix}. \quad (11.5.29)$$

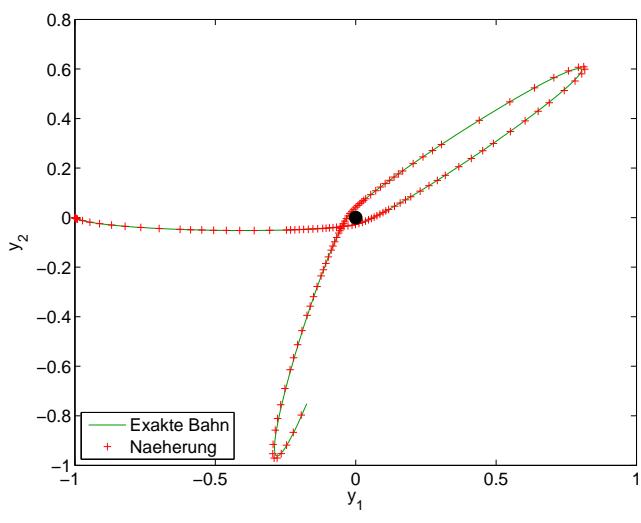
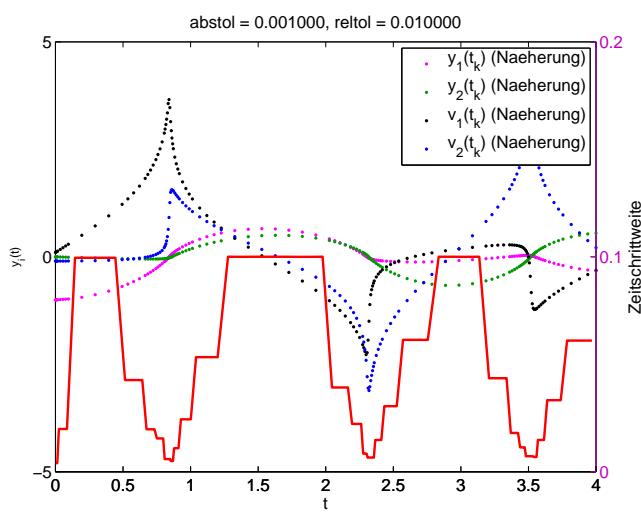
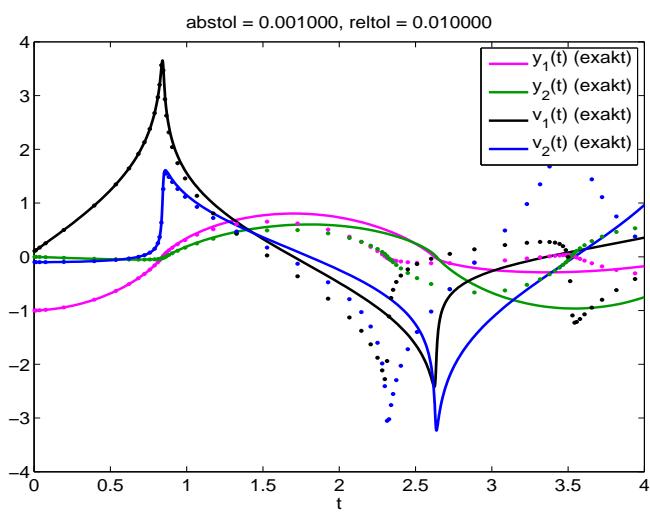
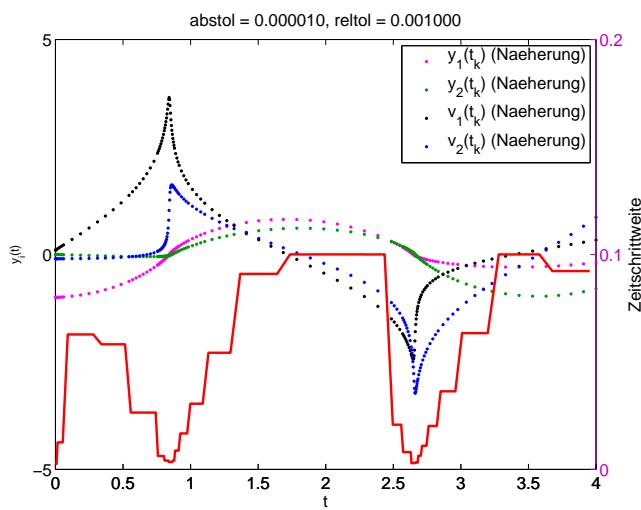
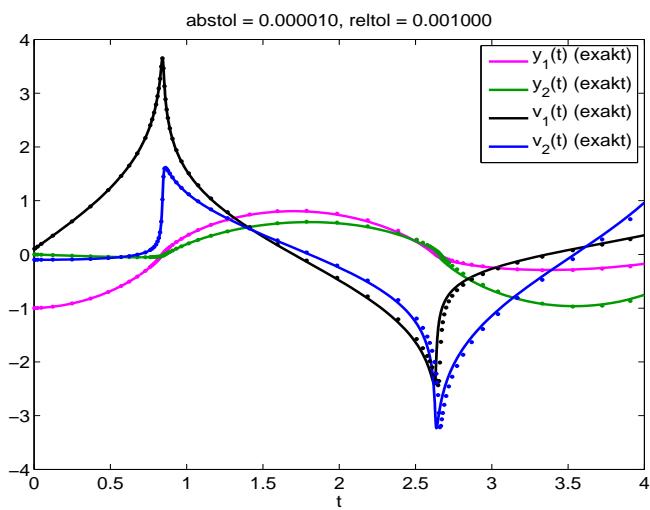
The following initial values used in the experiment:

$$\mathbf{y}(0) := \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \quad \mathbf{v}(0) := \begin{bmatrix} 0.1 \\ -0.1 \end{bmatrix}$$

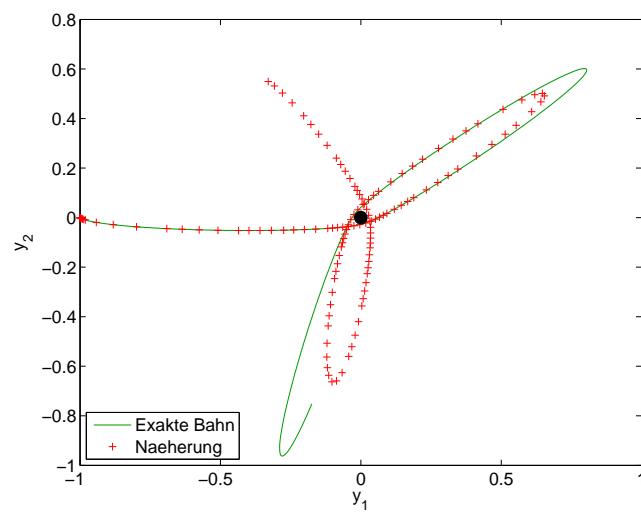
Adaptive numerical integration in MATLAB

```
ode45(@(t,x) f(t,x), [0 4], [-1;0;0.1;-0.1], options);
```

```
1 options = odeset('reltol',0.001,'abstol',1e-5);  
2 options = odeset('reltol',0.01,'abstol',1e-3);
```



reltol=0.001, abstol=1e-5



reltol=0.01, abstol=1e-3

### Observations:

- ☞ Fast changes in solution components captured by adaptive approach through very small timesteps.
- ☞ Completely wrong solution, if tolerance reduced slightly.

In this example we face a rather **sensitive dependence** of the trajectories on initial states or intermediate states. Small perturbations at one instance in time can be have a massive impact on the solution at later

times. Local stepsize control is powerless about preventing this.

---

## Summary and Learning Outcomes

# Chapter 12

## Single Step Methods for Stiff Initial Value Problems



*Supplementary reading.* [15, Sect. 11.9]

Explicit Runge-Kutta methods with stepsize control ( $\rightarrow$  Section 11.5) seem to be able to provide approximate solutions for any IVP with good accuracy provided that tolerances are set appropriately.

Everything settled about numerical integration?

### Example 12.0.1 (ode45 for stiff problem)

In this example we will witness the near failure of a high-order adaptive explicit Runge-Kutta method for a simple scalar autonomous ODE.

$$\text{IVP considered: } \dot{y} = \lambda y^2(1 - y), \quad \lambda := 500, \quad y(0) = \frac{1}{100}.$$

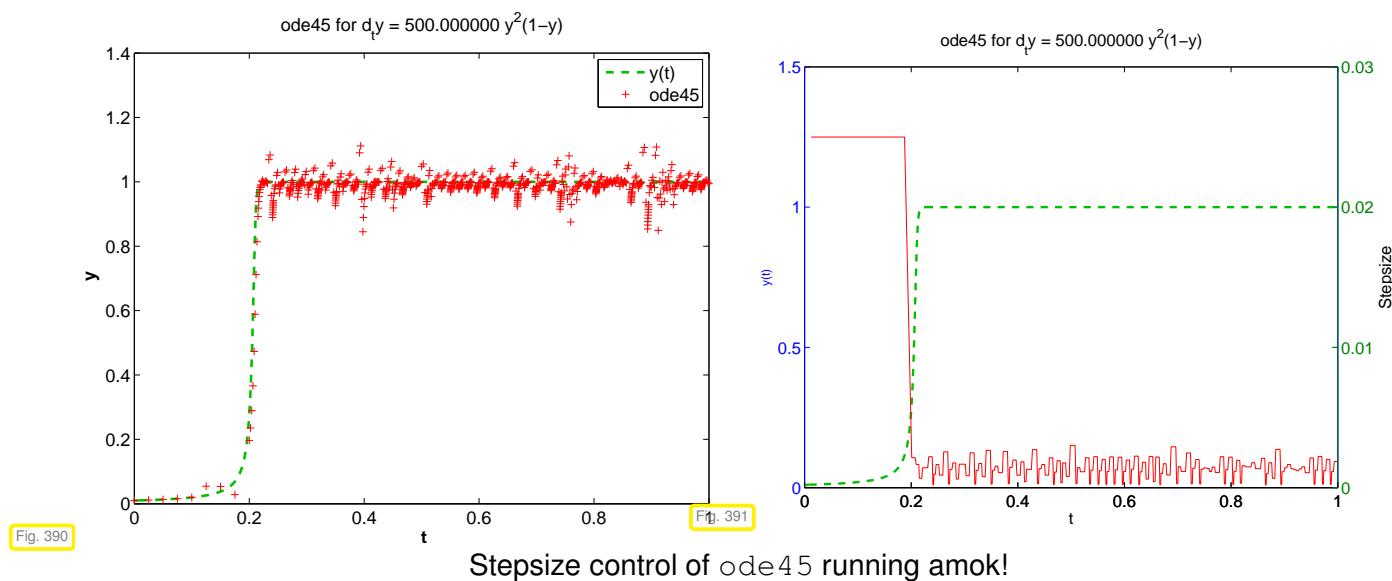
This is a logistic ODE as introduced in Ex. 11.1.5.

#### MATLAB-script 12.0.2: Use of MATLABintegrator ode45 for a stiff problem

```
1 fun = @(t,x) 500*x.^2*(1-x);  
2 options = odeset('reltol',0.1,'abstol',0.001,'stats','on');  
3 [t,y] = ode45(fun,[0 1],y0,options);
```

The option `stats = 'on'` makes MATLAB print statistics about the run of the integrators.

186 successful steps  
55 failed attempts  
1447 function evaluations



- ? The solution is virtually constant from  $t > 0.2$  and, nevertheless, the integrator uses tiny timesteps until the end of the integration interval.

## Contents

<b>12.1 Model problem analysis . . . . .</b>	<b>648</b>
<b>12.2 Stiff Initial Value Problems . . . . .</b>	<b>661</b>
<b>12.3 Implicit Runge-Kutta Single Step Methods . . . . .</b>	<b>666</b>
12.3.1 The implicit Euler method for stiff IVPs . . . . .	667
12.3.2 Collocation single step methods . . . . .	668
12.3.3 General implicit RK-SSMs . . . . .	671
12.3.4 Model problem analysis for implicit RK-SSMs . . . . .	673
<b>12.4 Semi-implicit Runge-Kutta Methods . . . . .</b>	<b>679</b>
<b>12.5 Splitting methods . . . . .</b>	<b>682</b>

## 12.1 Model problem analysis



*Supplementary reading.* See also [42, Ch. 77], [63, Sect. 11.3.3].

In this section we will discover a simple explanation for the startling behavior of **ode45** in Ex. 12.0.1.

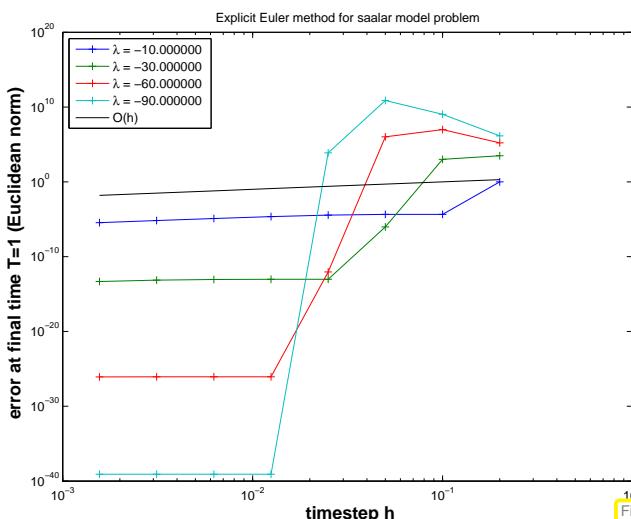
### Example 12.1.1 (Blow-up of explicit Euler method)

The simplest explicit RK-SSM is the explicit Euler method, see Section 11.2.1. We know that it should converge like  $\mathcal{O}(h)$  for meshwidth  $h \rightarrow 0$ . In this example we will see that this may be true only for sufficiently small  $h$ , which may be extremely small.

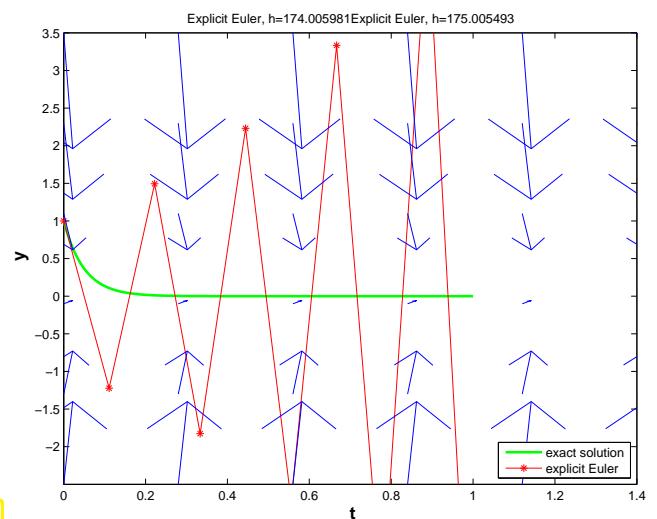
- \* We consider the IVP for the scalar linear decay ODE:

$$\dot{y} = f(y) := \lambda y \quad , \quad y(0) = 1 \quad .$$

- \* We apply the explicit Euler method (11.2.7) with uniform timestep  $h = 1/N$ ,  $N \in \{5, 10, 20, 40, 80, 160, 320, 640\}$ .



$\lambda$  large: blow-up of  $y_k$  for large timestep  $h$



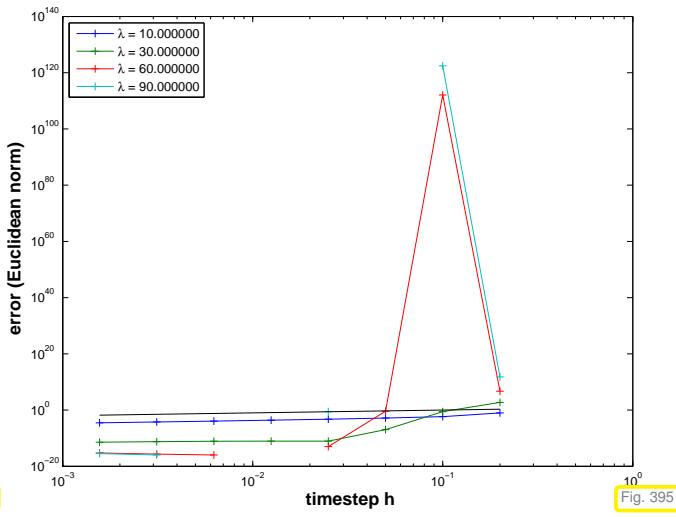
Explanation: From Fig. 393 we draw the geometric conclusion that, if  $h$  is “large in comparison with  $\lambda^{-1}$ ”, then the approximations  $y_k$  way miss the stationary point  $y = 0$  due to overshooting.

This leads to a sequence  $(y_k)_k$  with exponentially increasing oscillations.

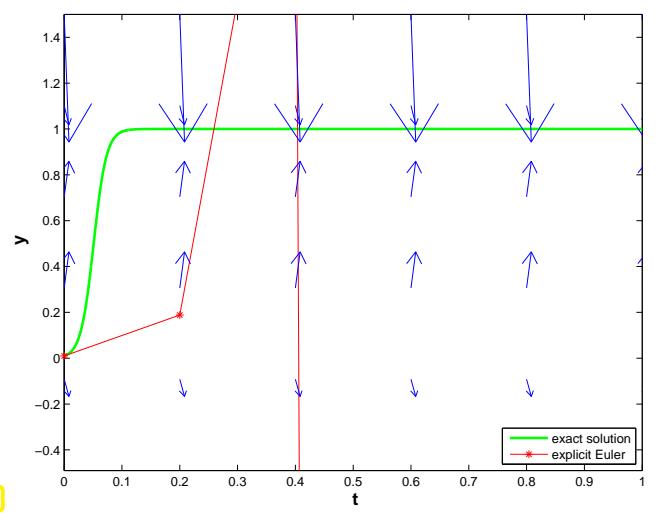
- \* Now we look at an IVP for the logistic ODE, see Ex. 11.1.5:

$$\dot{y} = f(y) := \lambda y(1 - y) , \quad y(0) = 0.01 .$$

- \* As before, we apply the explicit Euler method (11.2.7) with uniform timestep  $h = 1/N$ ,  $N \in \{5, 10, 20, 40, 80, 160, 320, 640\}$ .



$\lambda$  large: blow-up of  $y_k$  for large timestep  $h$



For large timesteps  $h$  we also observe oscillatory blow-up of the sequence  $(y_k)_k$ .

### Deeper analysis:

For  $y \approx 1$ :  $f(y) \approx \lambda(1 - y) \Rightarrow$  If  $y(t_0) \approx 1$ , then the solution of the IVP will behave like the solution of  $\dot{y} = \lambda(1 - y)$ , which is a linear ODE. Similary,  $z(t) := 1 - y(t)$  will behave like the solution of the “decay equation”  $\dot{z} = -\lambda z$ . Thus, around the stationary point  $y = 1$  the explicit Euler method behaves like it did for  $\dot{y} = \lambda y$  in the vicinity of the stationary point  $y = 0$ ; it grossly overshoots.

### (12.1.2) Linear model problem analysis: explicit Euler method

The phenomenon observed in the two previous examples is accessible to a remarkably simple rigorous analysis: Motivated by the considerations in Ex. 12.1.1 we study the explicit Euler method (11.2.7) for the

$$\text{linear model problem: } \dot{y} = \lambda y, \quad y(0) = y_0, \quad \text{with } \lambda \ll 0, \quad (12.1.3)$$

which has *exponentially decaying* exact solution

$$y(t) = y_0 \exp(\lambda t) \rightarrow 0 \quad \text{for } t \rightarrow \infty.$$

Recall the recursion for the explicit Euler with uniform timestep  $h > 0$  method for (12.1.3):

$$(11.2.7) \text{ for } f(y) = \lambda y: \quad y_{k+1} = y_k(1 + \lambda h). \quad (12.1.4)$$

We easily get a closed form expression for the approximations  $y_k$ :

►  $y_k = y_0(1 + \lambda h)^k \Rightarrow |y_k| \rightarrow \begin{cases} 0 & \text{, if } \lambda h > -2 \quad (\text{qualitatively correct}), \\ \infty & \text{, if } \lambda h < -2 \quad (\text{qualitatively wrong}) . \end{cases}$

#### Observed: timestep constraint

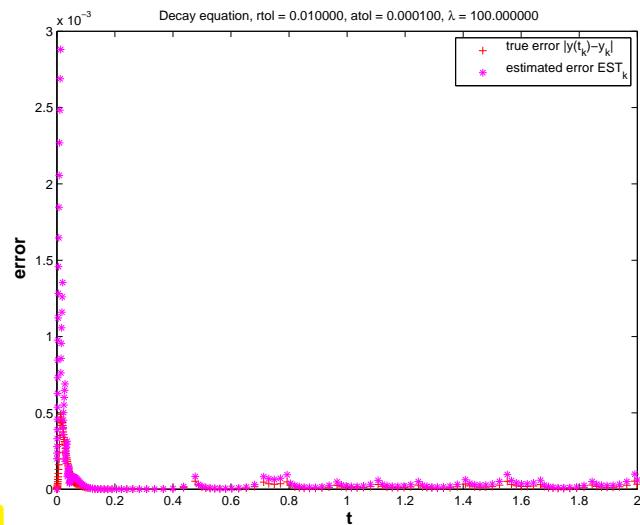
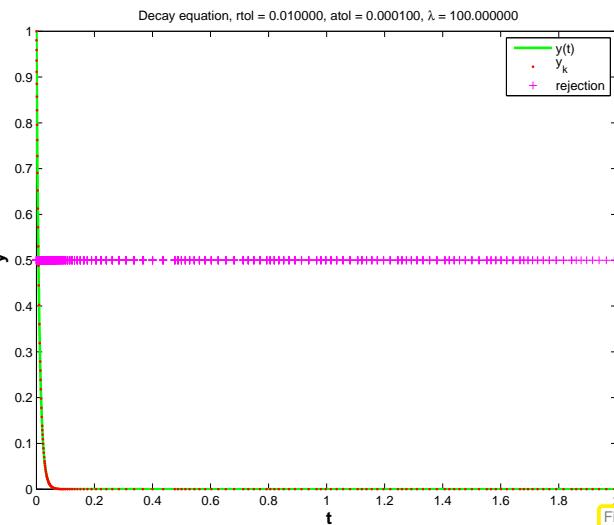
Only if  $|\lambda| h < 2$  we obtain a decaying solution by the explicit Euler method!

Could it be that the timestep control is desperately trying to enforce the qualitatively correct behavior of the numerical solution in Ex. 12.1.1? Let us examine how the simple stepsize control of Code 11.5.11 fares for model problem (12.1.3):

#### Example 12.1.6 (Simple adaptive timestepping for fast decay)

In this example we let a transparent adaptive timestep struggle with “overshooting”:

- \* “Linear model problem IVP”:  $\dot{y} = \lambda y, y(0) = 1, \lambda = -100$
- \* Simple adaptive timestepping method as in Ex. 11.5.13, see Code 11.5.11. Timestep control based on the pair of 1st-order explicit Euler method and 2nd-order explicit trapezoidal method.



Observation: in fact, stepsize control enforces small timesteps even if  $\mathbf{y}(t) \approx 0$  and persistently triggers rejections of timesteps. This is necessary to prevent overshooting in the Euler method, which contributes to the estimate of the one-step error.

We see the purpose of *stepsize control thwarted*, because after only a very short time the solution is almost zero and then, in fact, large timesteps should be chosen.

Are these observations a particular “flaw” of the explicit Euler method? Let us study the behavior of another simple explicit Runge-Kutta method applied to the linear model problem.

**Example 12.1.7 (Explicit trapezoidal method for decay equation → [15, Ex. 11.29])**

Recall recursion for the explicit trapezoidal method derived in Ex. 11.4.4:

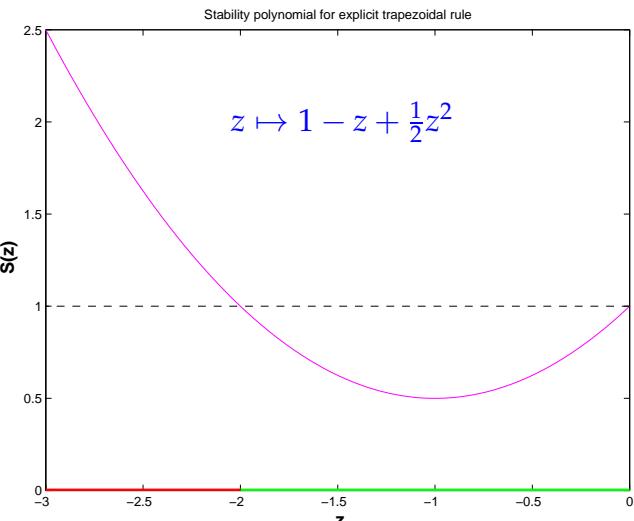
$$\mathbf{k}_1 = \mathbf{f}(t_0, \mathbf{y}_0), \quad \mathbf{k}_2 = \mathbf{f}(t_0 + h, \mathbf{y}_0 + h\mathbf{k}_1), \quad \mathbf{y}_1 = \mathbf{y}_0 + \frac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2). \quad (11.4.6)$$

Apply it to the model problem (12.1.3), that is, the scalar autonomous ODE with right hand side function  $\mathbf{f}(y) = f(y) = \lambda y, \lambda < 0$ :

$$\blacktriangleright \quad k_1 = \lambda y_0, \quad k_2 = \lambda(y_0 + hk_1) \Rightarrow y_1 = \underbrace{(1 + \lambda h + \frac{1}{2}(\lambda h)^2)}_{=:S(h\lambda)} y_0. \quad (12.1.8)$$

$\blacktriangleright$  the sequence of approximations generated by the explicit trapezoidal rule can be expressed in closed form as

$$y_k = S(h\lambda)^k y_0, \quad k = 0, \dots, N. \quad (12.1.9)$$



Clearly, blow-up can be avoided only if  $|S(h\lambda)| \leq 1$ :

$$|S(h\lambda)| < 1 \Leftrightarrow -2 < h\lambda < 0.$$

Qualitatively correct decay behavior of  $(y_k)_k$  only under **timestep constraint**

$$h \leq |2/\lambda|. \quad (12.1.10)$$

$\triangleleft$  the stability function for the explicit trapezoidal method

**(12.1.11) Model problem analysis for general explicit Runge-Kutta single step methods**

Apply the explicit Runge-Kutta method (→ Def. 11.4.9): encoded by the Butcher scheme  $\begin{array}{c|cc} \mathbf{c} & \mathbf{a} \\ \hline & \mathbf{b}^T \end{array}$  to the autonomous scalar linear ODE (12.1.3) ( $\dot{\mathbf{y}} = \lambda \mathbf{y}$ ). We write down the equations for the increments and  $y_1$

from Def. 11.4.9 for  $f(y) := \lambda y$  and then convert the resulting system of equations into matrix form:

$$\begin{array}{l} k_i = \lambda(y_0 + h \sum_{j=1}^{i-1} a_{ij} k_j), \\ y_1 = y_0 + h \sum_{i=1}^s b_i k_i \end{array} \Rightarrow \begin{bmatrix} \mathbf{I} - z\mathfrak{A} & \mathbf{0} \\ -z\mathbf{b}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{k} \\ y_1 \end{bmatrix} = y_0 \begin{bmatrix} \mathbf{1} \\ 1 \end{bmatrix}, \quad (12.1.12)$$

where  $\mathbf{k} \in \mathbb{R}^s \hat{=} \text{denotes the vector } [k_1, \dots, k_s]^\top / \lambda$  of increments, and  $z := \lambda h$ . Next we apply block Gaussian elimination ( $\rightarrow$  Rem. 1.6.30) to solve for  $y_1$  and obtain

$$y_1 = S(z)y_0 \quad \text{with} \quad S(z) := 1 + z\mathbf{b}^T(\mathbf{I} - z\mathfrak{A})^{-1}\mathbf{1}. \quad (12.1.13)$$

Alternatively we can express  $y_1$  through determinants appealing to Cramer's rule,

$$y_1 = y_0 \frac{\det \begin{bmatrix} \mathbf{I} - z\mathfrak{A} & \mathbf{1} \\ -z\mathbf{b}^T & 1 \end{bmatrix}}{\det \begin{bmatrix} \mathbf{I} - z\mathfrak{A} & \mathbf{0} \\ -z\mathbf{b}^T & 1 \end{bmatrix}} \Rightarrow S(z) = \det(\mathbf{I} - z\mathfrak{A} + z\mathbf{1}\mathbf{b}^T), \quad (12.1.14)$$

and note that  $\mathfrak{A}$  is a strictly lower triangular matrix, which means that  $\det(\mathbf{I} - z\mathfrak{A}) = 1$ . Thus we have proved the following theorem.

**Theorem 12.1.15. Stability function of explicit Runge-Kutta methods**  $\rightarrow$  [42, Thm. 77.2], [63, Sect. 11.8.4]

The discrete evolution  $\Psi_\lambda^h$  of an explicit  $s$ -stage Runge-Kutta single step method ( $\rightarrow$  Def. 11.4.9) with Butcher scheme  $\begin{array}{c|cc} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$  (see (11.4.11)) for the ODE  $\dot{y} = \lambda y$  amounts to a multiplication with the number

$$\Psi_\lambda^h = S(\lambda h) \Leftrightarrow y_1 = S(\lambda h)y_0,$$

where  $S$  is the **stability function**

$$S(z) := 1 + z\mathbf{b}^T(\mathbf{I} - z\mathfrak{A})^{-1}\mathbf{1} = \det(\mathbf{I} - z\mathfrak{A} + z\mathbf{1}\mathbf{b}^T), \quad \mathbf{1} := [1, \dots, 1]^\top \in \mathbb{R}^s. \quad (12.1.16)$$

### Example 12.1.17 (Stability functions of explicit Runge-Kutta single step methods)

From Thm. 12.1.15 and their Butcher schemes we can instantly compute the stability functions of explicit RK-SSM. We do this for a few methods whose Butcher schemes were listed in Ex. 11.4.13

- Explicit Euler method (11.2.7):

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \Rightarrow S(z) = 1 + z.$$

- Explicit trapezoidal method (11.4.6):

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \hline 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \Rightarrow S(z) = 1 + z + \frac{1}{2}z^2.$$

- Classical RK4 method:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array} \Rightarrow S(z) = 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4.$$

These examples confirm an immediate consequence of the determinant formula for the stability function  $S(z)$ .

#### Corollary 12.1.18. Polynomial stability function of explicit RK-SSM

For a consistent ( $\rightarrow$  Def. 11.3.10)  $s$ -stage explicit Runge-Kutta single step method according to Def. 11.4.9 the stability function  $S$  defined by (12.1.16) is a non-constant polynomial of degree  $\leq s$ :  $S \in \mathcal{P}_s$ .

#### Remark 12.1.19 (Stability function and exponential function)

Compare the two evolution operators:

- $\Phi \triangleq$  evolution operator ( $\rightarrow$  Def. 11.1.39) for  $\dot{y} = \lambda y$ ,
- $\Psi \triangleq$  discrete evolution operator ( $\rightarrow$  § 11.3.1) for an  $s$ -stage Runge-Kutta single step method.

$$\Phi^h y = e^{\lambda h} y \longleftrightarrow \Psi^h y = S(\lambda h) y .$$

In light of  $\Psi \approx \Phi$ , see (11.3.3), we expect that

$$S(z) \approx \exp(z) \quad \text{for small } |z| . \quad (12.1.20)$$

A more precise statement is made by the following lemma:

#### Lemma 12.1.21. Stability function as approximation of exp for small arguments

Let  $S$  denote the stability function of an  $s$ -stage explicit Runge-Kutta single step method of order  $q \in \mathbb{N}$ . Then

$$|S(z) - \exp(z)| = O(|z|^{q+1}) \quad \text{for } |z| \rightarrow 0 . \quad (12.1.22)$$

This means that the lowest  $q+1$  coefficients of  $S(z)$  must be equal to the first coefficients of the exponential series:

$$S(z) = \sum_{j=0}^q \frac{1}{j!} z^j + z^{q+1} p(z) \quad \text{with some } p \in \mathcal{P}_{s-q-1} .$$

#### Corollary 12.1.23. Stages limit order of explicit RK-SSM

An explicit  $s$ -stage RK-SSM has maximal order  $q \leq s$ .

**(12.1.24) Stability induced timestep constraint**

In § 12.1.11 we established that for the sequence  $(y_k)_{k=0}^{\infty}$  produced by an explicit Runge-Kutta single step method applied to the linear scalar model ODE  $\dot{y} = \lambda y$ ,  $\lambda \in \mathbb{R}$ , with uniform timestep  $h > 0$  holds

$$y_{k+1} = S(\lambda h)y_k \Rightarrow y_k = S(\lambda h^k)y_0.$$

►

$(y_k)_{k=0}^{\infty}$ non-increasing $(y_k)_{k=0}^{\infty}$ exponentially increasing	$\Leftrightarrow  S(\lambda h)  \leq 1$ $\Leftrightarrow  S(\lambda h)  > 1$	(12.1.25)
--	---	-----------

where  $S = S(z)$  is the stability function of the RK-SSM as defined in (12.1.16).

Invariably polynomials tend to  $\pm\infty$  for large (in modulus) arguments:

$$\forall S \in \mathcal{P}_s, S \neq \text{const} : \lim_{|z| \rightarrow \infty} S(z) = \infty \quad \text{uniformly}. \quad (12.1.26)$$

So, for any  $\lambda \neq 0$  there will be a threshold  $h_{\max} > 0$  so that  $|y_k| \rightarrow \infty$  as  $|h| > h_{\max}$ .

Reversing the argument we arrive at a **timestep constraint**, as already observed for the explicit Euler methods in § 12.1.2.

Only if one ensures that  $|\lambda h|$  is sufficiently small, one can avoid exponentially increasing approximations  $y_k$  (qualitatively wrong for  $\lambda < 0$ ) when applying an explicit RK-SSM to the model problem (12.1.3) with uniform timestep  $h > 0$ ,

For  $\lambda \ll 0$  this stability induced timestep constraint may force  $h$  to be much *smaller than required by demands on accuracy*: in this case timestepping becomes **inefficient**.

**Remark 12.1.27 (Stepsize control detects instability)**

Ex. 12.0.1, Ex. 12.1.6 send the message that local-in-time stepsize control as discussed in Section 11.5 selects timesteps that avoid blow-up, with a hefty price tag however in terms of computational cost and poor accuracy.

Objection: simple linear scalar IVP (12.1.3) may be an oddity rather than a model problem: the weakness of explicit Runge-Kutta methods discussed above may be just a peculiar response to an unusual situation. Let us extend our investigations to **systems of linear ODEs**,  $d > 1$ .

**(12.1.28) Systems of linear ordinary differential equations**

A generic linear ordinary differential equation on state space  $\mathbb{R}^d$  has the form

$$\dot{\mathbf{y}} = \mathbf{M}\mathbf{y} \quad \text{with a matrix } \mathbf{M} \in \mathbb{R}^{d,d}. \quad (12.1.29)$$

As explained in [59, Sect. 8.1], (12.1.29) can be solved by **diagonalization**: If we can find a *regular* matrix  $\mathbf{V} \in \mathbb{C}^{d,d}$  such that

$$\mathbf{M}\mathbf{V} = \mathbf{V}\mathbf{D} \quad \text{with diagonal matrix } \mathbf{D} = \begin{bmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_d \end{bmatrix} \in \mathbb{C}^{d,d}, \quad (12.1.30)$$

then the 1-parameter family of global solutions of (12.1.29) is given by

$$\mathbf{y}(t) = \mathbf{V} \begin{bmatrix} \exp(\lambda_1 t) & & 0 \\ & \ddots & \\ 0 & & \exp(\lambda_d t) \end{bmatrix} \mathbf{V}^{-1} \mathbf{y}_0, \quad \mathbf{y}_0 \in \mathbb{R}^d. \quad (12.1.31)$$

The columns of  $\mathbf{V}$  are a basis of **eigenvectors** of  $\mathbf{M}$ , the  $\lambda_j \in \mathbb{C}$ ,  $j = 1, \dots, d$  are the associated **eigenvalues** of  $\mathbf{M}$ , see Def. 7.1.1.

The idea behind diagonalization is the transformation of (12.1.29) into *d decoupled* scalar linear ODEs:

$$\dot{\mathbf{y}} = \mathbf{M}\mathbf{y} \quad \xrightarrow{\mathbf{z}(t) := \mathbf{V}^{-1}\mathbf{y}(t)} \quad \dot{\mathbf{z}} = \mathbf{D}\mathbf{z} \quad \leftrightarrow \quad \begin{array}{l} \dot{z}_1 = \lambda_1 z_1 \\ \vdots \\ \dot{z}_d = \lambda_d z_d \end{array}, \quad \text{since } \mathbf{M} = \mathbf{V}\mathbf{D}\mathbf{V}^{-1}.$$

The formula (12.1.31) can be generalized to

$$\mathbf{y}(t) = \exp(\mathbf{M}t)\mathbf{y}_0 \quad \text{with matrix exponential} \quad \exp(\mathbf{B}) := \sum_{k=0}^{\infty} \frac{1}{k!} \mathbf{B}^k, \quad \mathbf{B} \in \mathbb{C}^{d,d}. \quad (12.1.32)$$

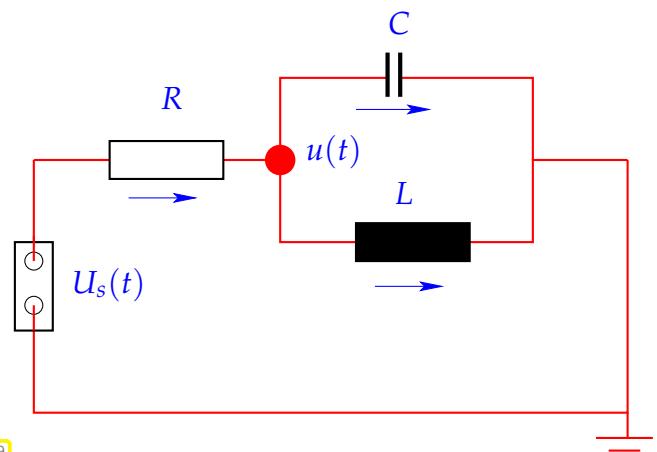
### Example 12.1.33 (Transient simulation of RLC-circuit)

Consider circuit from Ex. 11.1.13

Transient nodal analysis leads to the second-order linear ODE

$$\ddot{u} + \alpha\dot{u} + \beta u = g(t),$$

with coefficients  $\alpha := (RC)^{-1}$ ,  $\beta = (LC)^{-1}$ ,  $g(t) = \alpha U_s(t)$ .



We transform it to a linear 1st-order ODE as in Rem. 11.1.23 by introducing  $v := \dot{u}$  as additional solution component:

$$\underbrace{\begin{bmatrix} \dot{u} \\ \dot{v} \end{bmatrix}}_{=: \dot{\mathbf{y}}} = \underbrace{\begin{bmatrix} 0 & 1 \\ -\beta & -\alpha \end{bmatrix}}_{=: \mathbf{f}(t, \mathbf{y})} \begin{bmatrix} u \\ v \end{bmatrix} - \begin{bmatrix} 0 \\ g(t) \end{bmatrix}.$$

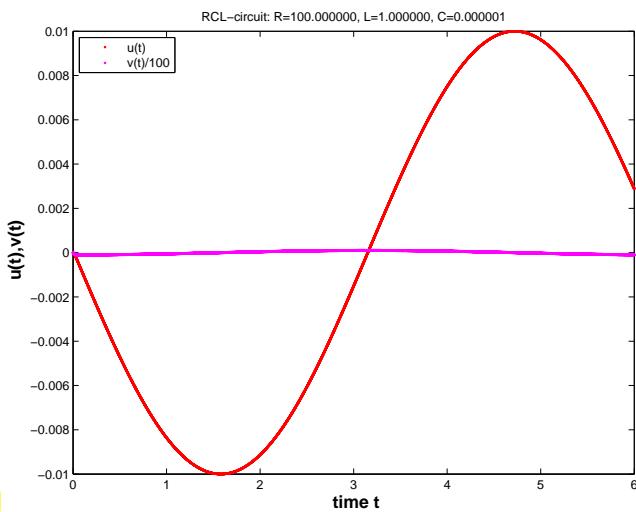
We integrate IVPs for this ODE by means of MATLAB's adaptive integrator `ode45`.

**MATLAB-code 12.1.34: simulation of linear RLC circuit using `ode45`**

```

1 function stiffcircuit(R,L,C,Us,tspan,filename)
2 % Transient simulation of simple linear circuit of
% Ex. refex:stiffcircuit
3 % R,L,C: parameters for circuits elements (compatible units required)
4 % Us: exciting time-dependent voltage  $U_s = U_s(t)$ , function handle
5 % zero initial values
6
7 % Coefficient for 2nd-order ODE  $\ddot{u} + \alpha\dot{u} + \beta = g(t)$ 
8 alpha = 1/(R*C); beta = 1/(C*L);
9 % Conversion to 1st-order ODE  $\mathbf{y} = \mathbf{My} + \begin{pmatrix} 0 \\ g(t) \end{pmatrix}$ . Set up right hand side
10 M = [0 , 1; -beta , -alpha]; rhs = @(t,y) (M*y - [ 0 ;
    alpha*Us(t)]);
11 % Set tolerances for MATLAB integrator, see Rem. 11.5.23
12 options = odeset('reltol',0.1,'abstol',0.001,'stats','on');
13 y0 = [0;0]; [t,y] = ode45(rhs,tspan,y0,options);

```



$R = 100\Omega$ ,  $L = 1H$ ,  $C = 1\mu F$ ,  $U_s(t) = 1V \sin(t)$ ,  $u(0) = v(0) = 0$  ("switch on")

`ode45` statistics:

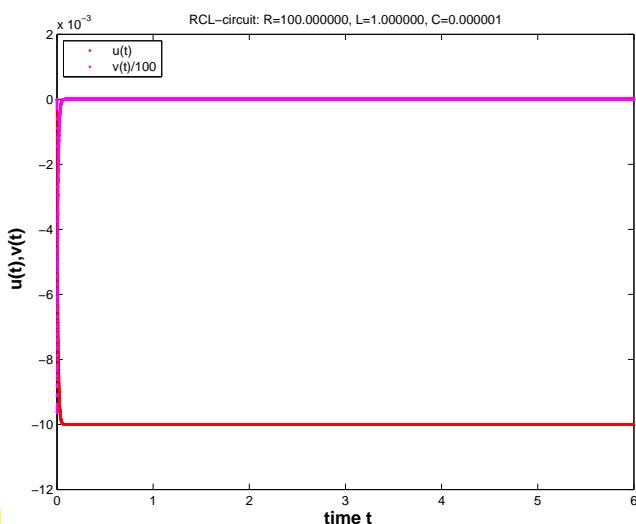
17897 successful steps

1090 failed attempts

113923 function evaluations

Inefficient: way more timesteps than required for resolving smooth solution, cf. remark in the end of § 12.1.24.

Maybe the time-dependent right hand side due to the time-harmonic excitation severely affects `ode45`? Let us try a constant exciting voltage:



$R = 100\Omega$ ,  $L = 1H$ ,  $C = 1\mu F$ ,  $U_s(t) = 1V$ ,  $u(0) = v(0) = 0$  ("switch on")

`ode45` statistics:

17901 successful steps

1210 failed attempts

114667 function evaluations

Tiny timesteps despite virtually constant solution!

We make the same observation as in Ex. 12.0.1, Ex. 12.1.6: the local-in-time stepsize control of `ode45` ( $\rightarrow$  Section 11.5) enforces extremely small timesteps though the solution almost constant except at  $t = 0$ .

To understand the structure of the solutions for this transient circuit example, let us apply the diagonalization technique from § 12.1.28 to the linear ODE

$$\dot{\mathbf{y}} = \underbrace{\begin{bmatrix} 0 & 1 \\ -\beta & -\alpha \end{bmatrix}}_{=: \mathbf{M}} \mathbf{y} , \quad \mathbf{y}(0) = \mathbf{y}_0 \in \mathbb{R}^2 . \quad (12.1.35)$$

Above we face the situation  $\beta \gg \frac{1}{4}\alpha^2 \gg 1$ .

We can obtain the general solution of  $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ ,  $\mathbf{M} \in \mathbb{R}^{2,2}$ , by diagonalization of  $\mathbf{M}$  (if possible):

$$\mathbf{MV} = \mathbf{M}(\mathbf{v}_1, \mathbf{v}_2) = (\mathbf{v}_1, \mathbf{v}_2) \begin{bmatrix} \lambda_1 & \\ & \lambda_2 \end{bmatrix} . \quad (12.1.36)$$

where  $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^2 \setminus \{0\}$  are the eigenvectors of  $\mathbf{M}$ ,  $\lambda_1, \lambda_2$  are the eigenvalues of  $\mathbf{M}$ , see Def. 7.1.1. For the latter we find

$$\lambda_{1/2} = \frac{1}{2}(\alpha \pm D), \quad D := \begin{cases} \sqrt{\alpha^2 - 4\beta} & , \text{ if } \alpha^2 \geq 4\beta , \\ i\sqrt{4\beta - \alpha^2} & , \text{ if } \alpha^2 < 4\beta . \end{cases}$$

Note that the eigenvalue have non-vanishing imaginary part in the setting of the experiment.

Then we transform  $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$  into *decoupled* scalar linear ODEs:

$$\dot{\mathbf{y}} = \mathbf{M}\mathbf{y} \Leftrightarrow \mathbf{V}^{-1}\dot{\mathbf{y}} = \mathbf{V}^{-1}\mathbf{M}\mathbf{V}(\mathbf{V}^{-1}\mathbf{y}) \stackrel{\mathbf{z}(t) := \mathbf{V}^{-1}\mathbf{y}(t)}{\Leftrightarrow} \dot{\mathbf{z}} = \begin{bmatrix} \lambda_1 & \\ & \lambda_2 \end{bmatrix} \mathbf{z} . \quad (12.1.37)$$

This yields the general solution of the ODE  $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ , see also [77, Sect. 5.6]:

$$\mathbf{y}(t) = A\mathbf{v}_1 \exp(\lambda_1 t) + B\mathbf{v}_2 \exp(\lambda_2 t) , \quad A, B \in \mathbb{R} . \quad (12.1.38)$$

Note:  $t \mapsto \exp(\lambda_i t)$  is general solution of the ODE  $\dot{z}_i = \lambda_i z_i$ .

### (12.1.39) “Diagonalization” of explicit Euler method

Recall discrete evolution of explicit Euler method (11.2.7) for ODE  $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ ,  $\mathbf{M} \in \mathbb{R}^{d,d}$ :

$$\Psi^h \mathbf{y} = \mathbf{y} + h\mathbf{M}\mathbf{y} \Leftrightarrow \mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{M}\mathbf{y}_k .$$

As in § 12.1.28 we assume that  $\mathbf{M}$  can be diagonalized, that is (12.1.30) holds:  $\mathbf{V}^{-1}\mathbf{M}\mathbf{V} = \mathbf{D}$  with a diagonal matrix  $\mathbf{D} \in \mathbb{C}^{d,d}$  containing the eigenvalues of  $\mathbf{M}$  on its diagonal. Next, apply the *decoupling by diagonalization* idea to the recursion of the explicit Euler method.

$$\mathbf{V}^{-1}\mathbf{y}_{k+1} = \mathbf{V}^{-1}\mathbf{y}_k + h\mathbf{V}^{-1}\mathbf{M}\mathbf{V}(\mathbf{V}^{-1}\mathbf{y}_k) \stackrel{\mathbf{z}_k := \mathbf{V}^{-1}\mathbf{y}_k}{\Leftrightarrow} \underbrace{(\mathbf{z}_{k+1})_i}_{\triangleq \text{explicit Euler step for } \dot{z}_i = \lambda_i z_i} = (\mathbf{z}_k)_i + h\lambda_i(\mathbf{z}_k)_i . \quad (12.1.40)$$

Crucial insight:

The explicit Euler method generates uniformly bounded solution sequences  $(\mathbf{y}_k)_{k=0}^\infty$  for  $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$  with diagonalizable matrix  $\mathbf{M} \in \mathbb{R}^{d,d}$  with eigenvalues  $\lambda_1, \dots, \lambda_d$ , if and only if it generates uniformly bounded sequences for all the scalar ODEs  $\dot{z} = \lambda_i z$ ,  $i = 1, \dots, d$ .

So far we conducted the model problem analysis under the premises  $\lambda < 0$ .

However, in Ex. 12.1.33 we face  $\lambda_{1/2} = -\frac{1}{2}(\alpha \pm i\sqrt{4\beta - \alpha^2})$  (complex eigenvalues!). Let us now examine how the explicit Euler method and even general explicit RK-methods respond to them.

### Example 12.1.41 (Explicit Euler method for damped oscillations)

Consider linear model IVP (12.1.3) for  $\lambda \in \mathbb{C}$ :

$$\operatorname{Re} \lambda < 0 \Rightarrow \text{exponentially decaying solution } y(t) = y_0 \exp(\lambda t),$$

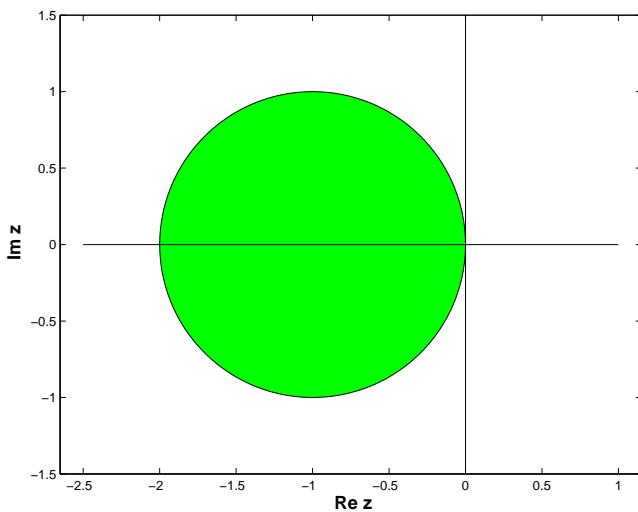
because  $|\exp(\lambda t)| = \exp(\operatorname{Re} \lambda \cdot t)$ .

The **model problem analysis** from Ex. 12.1.1, Ex. 12.1.7 can be extended verbatim to the case of  $\lambda \in \mathbb{C}$ . It yields the following insight for the explicit Euler method and  $\lambda \in \mathbb{C}$ :

The sequence generated by the explicit Euler method (11.2.7) for the model problem (12.1.3) satisfies

$$y_{k+1} = y_k(1 + h\lambda) \quad \Rightarrow \quad \lim_{k \rightarrow \infty} y_k = 0 \Leftrightarrow |1 + h\lambda| < 1. \quad (12.1.4)$$

↑  
timestep constraint to get decaying (discrete) solution !



$$\Lhd \{z \in \mathbb{C}: |1 + z| < 1\}$$

The green region of the complex plane marks values for  $\lambda h$ , for which the explicit Euler method will produce exponentially decaying solutions.

Now we can conjecture what happens in Ex. 12.1.33: the eigenvalues  $\lambda_{1/2} = -\frac{1}{2}\alpha \pm i\sqrt{\beta - \frac{1}{4}\alpha^2}$  of **M** have a very large (in modulus) negative real part. Since `ode45` can be expected to behave as if it integrates  $\dot{z} = \lambda_2 z$ , it faces a severe timestep constraint, if exponential blow-up is to be avoided, see Ex. 12.1.1. Thus stepsize control must resort to tiny timesteps.

### (12.1.42) Extended model problem analysis for explicit Runge-Kutta single step methods

We apply an explicit  $s$ -stage RK-SSM ( $\rightarrow$  `crefdef:rk`) described by the Butcher scheme  $\begin{array}{c|cc} c & \mathfrak{A} \\ \hline & b^T \end{array}$  to the autonomous linear ODE  $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ ,  $\mathbf{M} \in \mathbb{C}^{d,d}$ , and obtain (for the first step with timestep size  $h > 0$ )

$$\mathbf{k}_\ell = \mathbf{M}(\mathbf{y}_0 + h \sum_{j=1}^{s-1} a_{\ell j} \mathbf{k}_j), \quad \ell = 1, \dots, s, \quad \mathbf{y}_1 = \mathbf{y}_0 + h \sum_{\ell=1}^s b_\ell \mathbf{k}_\ell. \quad (12.1.43)$$

Now assume that  $\mathbf{M}$  can be diagonalized, that is (12.1.30) holds:  $\mathbf{V}^{-1}\mathbf{M}\mathbf{V} = \mathbf{D}$  with a diagonal matrix  $\mathbf{D} \in \mathbb{C}^{d,d}$  containing the eigenvalues  $\lambda_i \in \mathbb{C}$  of  $\mathbf{M}$  on its diagonal. Then apply the substitutions

$$\hat{\mathbf{k}}_\ell := \mathbf{V}^{-1}\mathbf{k}_\ell, \quad \ell = 1, \dots, s, \quad \hat{\mathbf{y}}_k := \mathbf{V}^{-1}\mathbf{y}_k, \quad k = 0, 1,$$

to (??), which yield

$$\hat{\mathbf{k}}_\ell = \mathbf{D}(\hat{\mathbf{y}}_0 + h \sum_{j=1}^{s-1} a_{\ell j} \hat{\mathbf{k}}_j), \quad \ell = 1, \dots, s, \quad \hat{\mathbf{y}}_1 = \hat{\mathbf{y}}_0 + h \sum_{\ell=1}^s b_\ell \hat{\mathbf{k}}_\ell. \quad (12.1.44)$$

$\Updownarrow$

$$(\hat{\mathbf{k}}_\ell)_i = \lambda_i((\mathbf{y}_0)_i + h \sum_{j=1}^{s-1} a_{\ell j} (\hat{\mathbf{k}}_j)_i), \quad (\hat{\mathbf{y}}_1)_i = (\hat{\mathbf{y}}_0)_i + h \sum_{\ell=1}^s b_\ell (\hat{\mathbf{k}}_\ell)_i, \quad i = 1, \dots, d. \quad (12.1.45)$$

We infer that, if  $(\mathbf{y}_k)_k$  is the sequence produced by an explicit RK-SSM applied to  $\dot{\mathbf{y}} = \mathbf{My}$ , then

$$\mathbf{y}_k = \mathbf{V} \begin{bmatrix} y_k^{[1]} & & 0 \\ & \ddots & \\ 0 & & y_k^{[d]} \end{bmatrix} \mathbf{V}^{-1},$$

where  $(y_k^{[i]})_k$  is the sequence generated by the same RK-SSM with the same sequence of timesteps for the IVP  $\dot{y} = \lambda_i y, y(0) = (\mathbf{V}^{-1}\mathbf{y}_0)_i$ .

The RK-SSM generates uniformly bounded solution sequences  $(\mathbf{y}_k)_{k=0}^\infty$  for  $\dot{\mathbf{y}} = \mathbf{My}$  with diagonalizable matrix  $\mathbf{M} \in \mathbb{R}^{d,d}$  with eigenvalues  $\lambda_1, \dots, \lambda_d$ , if and only if it generates uniformly bounded sequences for all the scalar ODEs  $\dot{z} = \lambda_i z, i = 1, \dots, d$ .

Hence, understanding the behavior of RK-SSM for autonomous scalar linear ODEs  $\dot{y} = \lambda y$  with  $\lambda \in \mathbb{C}$  is enough to predict their behavior for general autonomous linear systems of ODEs.

From the considerations of § 12.1.24 we deduce the following fundamental result.

### Theorem 12.1.46. (Absolute) stability of explicit RK-SSM for linear systems of ODEs

The sequence  $(\mathbf{y}_k)_k$  of approximations generated by an explicit RK-SSM ( $\rightarrow$  Def. 11.4.9) with stability function  $S$  (defined in (12.1.16)) applied to the linear autonomous ODE  $\dot{\mathbf{y}} = \mathbf{My}, \mathbf{M} \in \mathbb{C}^{d,d}$ , with uniform timestep  $h > 0$  decays exponentially for every initial state  $\mathbf{y}_0 \in \mathbb{C}^d$ , if and only if  $|S(\lambda_i h)| < 1$  for all eigenvalues  $\lambda_i$  of  $\mathbf{M}$ .

Please note that

$$\operatorname{Re} \lambda_i < 0 \quad \forall i \in \{1, \dots, d\} \implies \|\mathbf{y}(t)\| \rightarrow 0 \quad \text{for } t \rightarrow \infty,$$

for any solution of  $\dot{\mathbf{y}} = \mathbf{My}$ . This is obvious from the representation formula (12.1.31).

### (12.1.47) Region of (absolute) stability of explicit RK-SSM

We consider an explicit Runge-Kutta single step method with stability function  $S$  for the model linear scalar IVP  $\dot{y} = \lambda y$ ,  $y(0) = y_0$ ,  $\lambda \in \mathbb{C}$ . From Thm. 12.1.15 we learn that for uniform stepsize  $h > 0$  we have  $y_k = S(\lambda h)^k y_0$  and conclude that

$$y_k \rightarrow 0 \quad \text{for } k \rightarrow \infty \Leftrightarrow |S(\lambda h)| < 1. \quad (12.1.48)$$

Hence, the modulus  $|S(\lambda h)|$  tells us for which combinations of  $\lambda$  and stepsize  $h$  we achieve exponential decay  $y_k \rightarrow \infty$  for  $k \rightarrow \infty$ , which is the desirable behavior of the approximations for  $\operatorname{Re} \lambda < 0$ .

#### Definition 12.1.49. Region of (absolute) stability

Let the discrete evolution  $\Psi$  for a single step method applied to the scalar linear ODE  $\dot{y} = \lambda y$ ,  $\lambda \in \mathbb{C}$ , be of the form

$$\Psi^h y = S(z)y, \quad y \in \mathbb{C}, h > 0 \quad \text{with} \quad z := h\lambda \quad (12.1.50)$$

and a function  $S : \mathbb{C} \rightarrow \mathbb{C}$ . Then the **region of (absolute) stability** of the single step method is given by

$$\mathcal{S}_\Psi := \{z \in \mathbb{C} : |S(z)| < 1\} \subset \mathbb{C}.$$

Of course, by Thm. 12.1.15, in the case of explicit RK-SSM the function  $S$  will coincide with their **stability function** from (12.1.16).

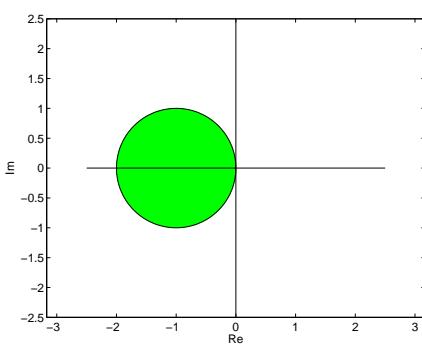
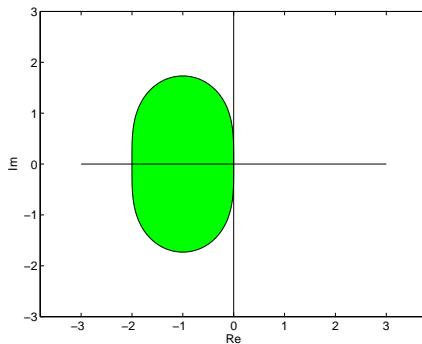
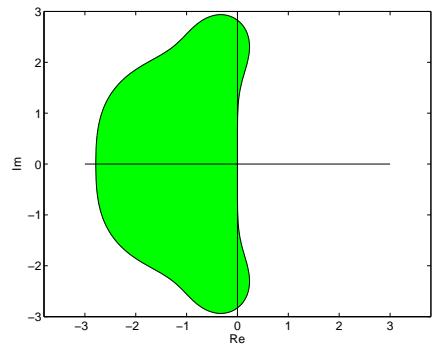
We can easily combine the statement of Thm. 12.1.46 with the concept of a region of stability and conclude that an explicit RK-SSM will generate exponentially decaying solutions for the linear ODE  $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ ,  $\mathbf{M} \in \mathbb{C}^{d,d}$ , for every initial state  $\mathbf{y}_0 \in \mathbb{C}^d$ , if and only if  $\lambda_i h \in \mathcal{S}_\Psi$  for all eigenvalues  $\lambda_i$  of  $\mathbf{M}$ .

Adopting the arguments of § 12.1.24 we conclude from Cor. 12.1.18 that

- \* the regions of (absolute) stability of explicit RK-SSM are **bounded**,
- \* a **timestep constraint** depending on the eigenvalues of  $\mathbf{M}$  is necessary to have a guaranteed exponential decay RK-solutions for  $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ .

#### Example 12.1.51 (Regions of stability of some explicit RK-SSM)

The green domains  $\subset \mathbb{C}$  depict the bounded regions of stability for some RK-SSM from Ex. 11.4.13.

 $\mathcal{S}_\Psi$ : explicit Euler (11.2.7) $\mathcal{S}_\Psi$ : explicit trapezoidal method $\mathcal{S}_\Psi$ : classical RK4 method

In general we have for a consistent RK-SSM ( $\rightarrow$  Def. 11.3.10) that their stability functions satisfy  $S(z) = 1 + z + O(z^2)$  for  $z \rightarrow 0$ . Therefore,  $\mathcal{S}_\Psi \neq \emptyset$  and the imaginary axis will be tangent to  $\mathcal{S}_\Psi$  in  $z = 0$ .

## 12.2 Stiff Initial Value Problems

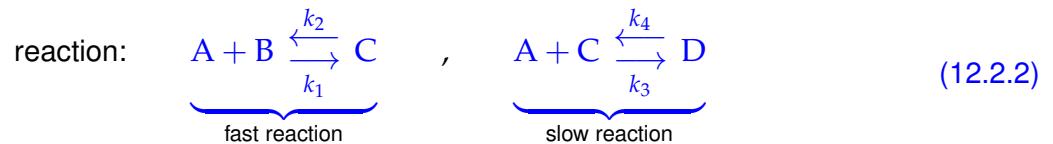


*Supplementary reading.* [63, Sect. 11.10]

This section will reveal that the behavior observed in Ex. 12.0.1 and Ex. 12.1.1 is typical for a large class of problems and that the model problem (12.1.3) really represents a “generic case”. This justifies the attention paid to linear model problem analysis in Section 12.1.

### Example 12.2.1 (Kinetics of chemical reactions $\rightarrow$ [42, Ch. 62])

In Ex. 11.5.1 we already saw an ODE model for the dynamics of a chemical reaction. Now we study an abstract reaction.



Vastly different reaction constants:

$$k_1, k_2 \gg k_3, k_4$$

► If  $c_A(0) > c_B(0)$  ➤ 2nd reaction determines overall long-term reaction dynamics

Mathematical model: non-linear ODE involving concentrations  $\mathbf{y}(t) = (c_A(t), c_B(t), c_C(t), c_D(t))^T$

$$\dot{\mathbf{y}} := \frac{d}{dt} \begin{bmatrix} c_A \\ c_B \\ c_C \\ c_D \end{bmatrix} = \mathbf{f}(\mathbf{y}) := \begin{bmatrix} -k_1 c_A c_B + k_2 c_C - k_3 c_A c_C + k_4 c_D \\ -k_1 c_A c_B + k_2 c_C \\ k_1 c_A c_B - k_2 c_C - k_3 c_A c_C + k_4 c_D \\ k_3 c_A c_C - k_4 c_D \end{bmatrix}. \quad (12.2.3)$$

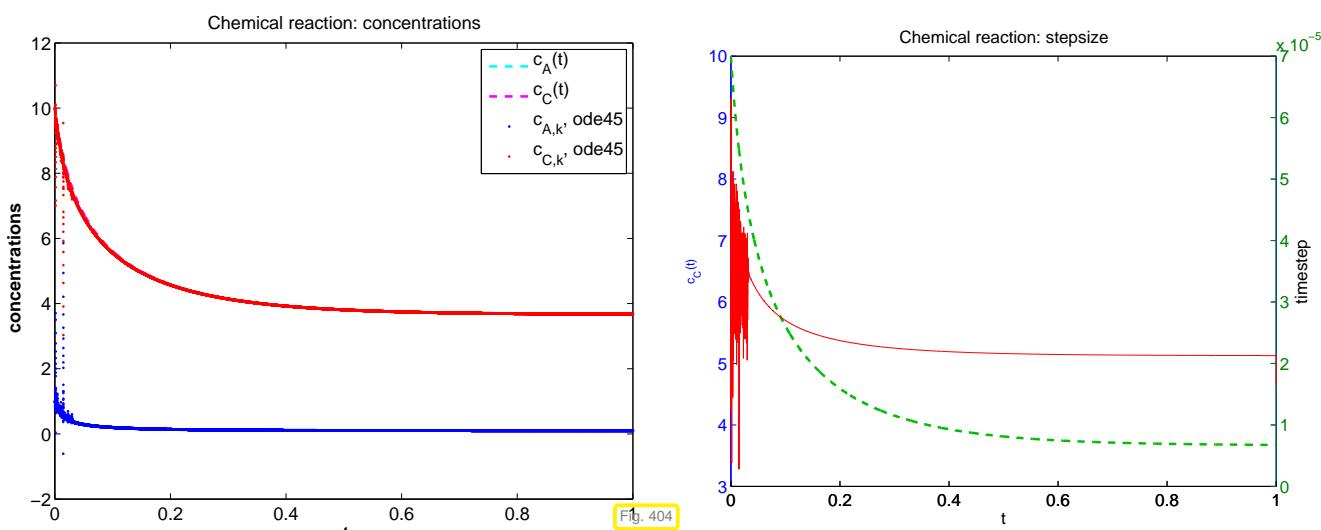
MATLAB computation:  $t_0 = 0$ ,  $T = 1$ ,  $k_1 = 10^4$ ,  $k_2 = 10^3$ ,  $k_3 = 10$ ,  $k_4 = 1$

**MATLAB-script 12.2.4: Simulation of “stiff” chemical reaction**

```

1 function chemstiff
2 % Simulation of kinetics of coupled chemical reactions with vastly
3 % different reaction
4 % rates, see (12.2.3) for the ODE model.
5 % reaction rates  $k_1, k_2, k_3, k_4$ ,  $k_1, k_2 \gg k_3, k_4$ .
6 k1 = 1E4; k2 = 1E3; k3 = 10; k4 = 1;
7 % definition of right hand side function for ODE solver
8 fun = @(t,y) ([-k1*y(1)*y(2) + k2*y(3) - k3*y(1)*y(3) + k4*y(4);
9             -k1*y(1)*y(2) + k2*y(3);
10            k1*y(1)*y(2) - k2*y(3) - k3*y(1)*y(3) + k4*y(4);
11            k3*y(1)*y(3) - k4*y(4)]);
12 tspan = [0 1]; % Integration time interval
13 L = tspan(2)-tspan(1); % Duration of simulation
14 y0 = [1;1;10;0]; % Initial value y0
15 % compute "exact" solution, using ode113 with tight error tolerances
16 options = odeset('reltol',10*eps,'abstol',eps,'stats','on');
17 % get the 'exact' solution using ode113
18 [tex,yex] = ode113(fun,[0 1],y0,options);

```



Observations: After a **fast initial transient** phase, the solution shows only slow dynamics. Nevertheless, the explicit adaptive integrator `ode113` insists on using a tiny timestep. It behaves very much like `ode45` in Ex. 12.0.1.

**Example 12.2.5 (Strongly attractive limit cycle)**

We consider the non-linear Autonomous ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  with

$$\mathbf{f}(\mathbf{y}) := \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{y} + \lambda(1 - \|\mathbf{y}\|^2) \mathbf{y}, \quad (12.2.6)$$

on the state space  $D = \mathbb{R}^2 \setminus \{0\}$

For  $\lambda = 0$ , the initial value problem  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ ,  $\mathbf{y}(0) = (\cos \varphi, \sin \varphi)$ ,  $\varphi \in \mathbb{R}$  has the solution

$$\mathbf{y}(t) = \begin{bmatrix} \cos(t - \varphi) \\ \sin(t - \varphi) \end{bmatrix}, \quad t \in \mathbb{R}. \quad (12.2.7)$$

For this solution we have  $\|\mathbf{y}(t)\|_2 = 1$  for all times.

- (12.2.7) provides a solution even for  $\lambda \neq 0$ , if  $\|\mathbf{y}(0)\|_2 = 1$ , because in this case the term  $\lambda(1 - \|\mathbf{y}\|^2) \mathbf{y}$  will never become non-zero on the solution trajectory.

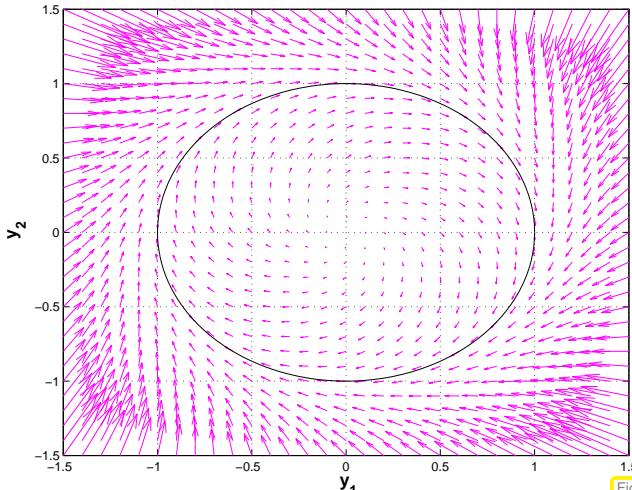


Fig. 405

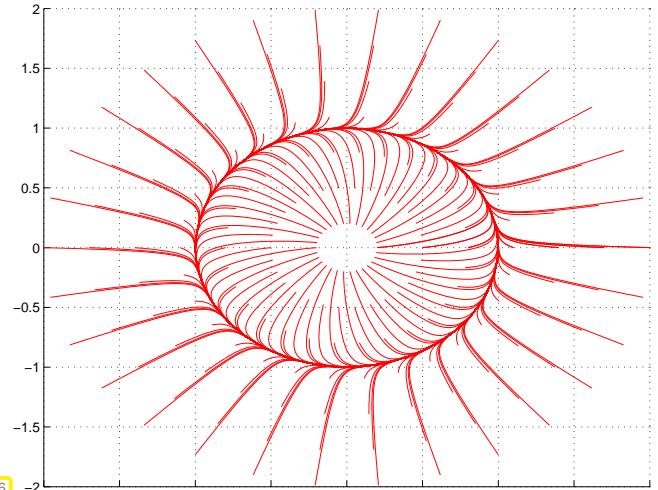


Fig. 406

vectorfield  $\mathbf{f} (\lambda = 1)$ solution trajectories ( $\lambda = 10$ )

### MATLAB-script 12.2.8: Application of ode45 for limit cycle problem

```

1 % MATLAB script for solving limit cycle ODE (12.2.6)
2 % define right hand side vectorfield
3 fun = @(t,y) ([ -y(2); y(1) ] +
4   lambda*(1-y(1)\symbol{94}2-y(2)\symbol{94}2)*y);
5 % standard invocation of MATLAB integrator, see Ex. 11.4.18
6 tspan = [0,2*pi]; y0 = [1,0];
7 opts = odeset('stats','on','reltol',1E-4,'abstol',1E-4);
8 [t45,y45] = ode45(fun,tspan,y0,opts);

```

We study the response of `ode45` to different choice of  $\lambda$  with initial state  $\mathbf{y}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ . According to the above considerations this initial state should completely “hide the impact of  $\lambda$  from our view”.

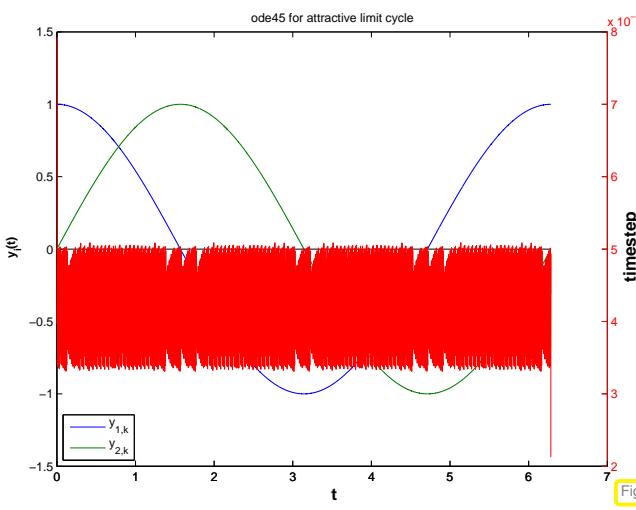
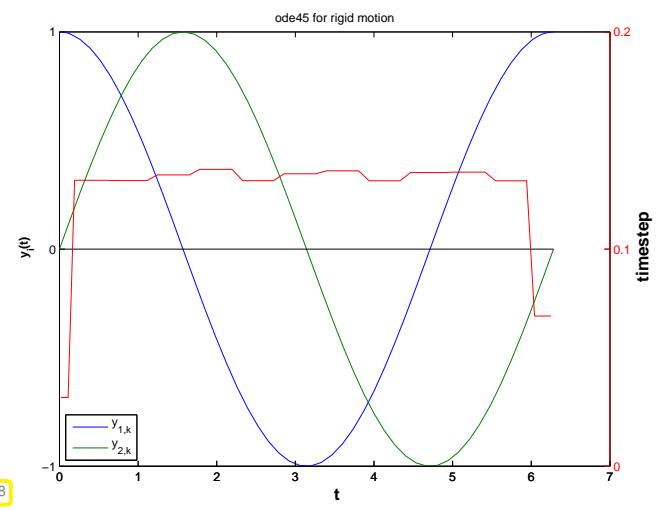


Fig. 407

many (3794) steps ( $\lambda = 1000$ )accurate solution with few steps ( $\lambda = 0$ )

Confusing observation: we have  $\|\mathbf{y}_0\| = 1$ , which implies  $\|\mathbf{y}(t)\| = 1 \quad \forall t!$

Thus, the term of the right hand side, which is multiplied by  $\lambda$  will always vanish on the exact solution trajectory, which stays on the unit circle.

Nevertheless, `ode45` is forced to use tiny timesteps by the *mere presence* of this term!

We want to find criteria that allow to predict the massive problems haunting explicit single step methods in the case of the *non-linear* IVP of Ex. 12.0.1, Ex. 12.2.1, and Ex. 12.2.5. Recall that for *linear* IVPs of the form  $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ ,  $\mathbf{y}(0) = \mathbf{y}_0$ , the model problem analysis of Section 12.1 tells us that, knowledge of the region of stability of the timestepping scheme, the eigenvalues of the matrix  $\mathbf{M} \in \mathbb{C}^{d,d}$  provide full information about timestep constraint we are going to face. Refer to Thm. 12.1.46 and § 12.1.47.

Issue: extension of stability analysis to non-linear ODEs ?

We start with a “phenomenological notion”, just a keyword to refer to the kind of difficulties presented by the IVPs of Ex. 12.0.1, Ex. 12.2.1, Ex. 12.1.6, and Ex. 12.2.5.

#### Notion 12.2.9. Stiff IVP

An initial value problem is called **stiff**, if stability imposes much tighter timestep constraints on *explicit single step methods* than the accuracy requirements.

#### (12.2.10) Linearization of ODEs

We consider a general autonomous ODE:

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \mathbf{f} : D \subset \mathbb{R}^d \rightarrow \mathbb{R}^d$$

As usual, we assume  $\mathbf{f}$  to be  $C^2$ -smooth and that it enjoys local Lipschitz continuity ( $\rightarrow$  Def. 11.1.28) on  $D$  so that unique solvability of IVPs is guaranteed by Thm. 11.1.32.

We fix a state  $\mathbf{y}^* \in D$ ,  $D$  the state space, write  $t \mapsto \mathbf{y}(t)$  for the solution with  $\mathbf{y}(0) = \mathbf{y}^*$ . We set  $\mathbf{z}(t) = \mathbf{y}(t) - \mathbf{y}^*$ , which satisfies

$$\mathbf{z}(0) = 0, \quad \dot{\mathbf{z}} = \mathbf{f}(\mathbf{y}^* + \mathbf{z}) = \mathbf{f}(\mathbf{y}^*) + D\mathbf{f}(\mathbf{y}^*)\mathbf{z} + R(\mathbf{y}^*, \mathbf{z}), \quad \text{with } \|R(\mathbf{y}^*, \mathbf{z})\| = O(\|\mathbf{z}\|^2).$$

This is obtained by Taylor expansion of  $\mathbf{f}$  at  $\mathbf{y}^*$ , see [77, Satz 7.5.2]. Hence, in a neighborhood of a state  $\mathbf{y}^*$  on a solution trajectory  $t \mapsto \mathbf{y}(t)$ , the deviation  $\mathbf{z}(t) = \mathbf{y}(t) - \mathbf{y}^*$  satisfies

$$\dot{\mathbf{z}} \approx \mathbf{f}(\mathbf{y}^*) + D\mathbf{f}(\mathbf{y}^*)\mathbf{z}. \quad (12.2.11)$$

► The short-time evolution of  $\mathbf{y}$  with  $\mathbf{y}(0) = \mathbf{y}^*$  is approximately governed by the **affine-linear ODE**

$$\dot{\mathbf{y}} = \mathbf{M}(\mathbf{y} - \mathbf{y}^*) + \mathbf{b}, \quad \mathbf{M} := D\mathbf{f}(\mathbf{y}^*) \in \mathbb{R}^{d,d}, \quad \mathbf{b} := \mathbf{f}(\mathbf{y}^*) \in \mathbb{R}^d. \quad (12.2.12)$$

#### (12.2.13) Linearization of explicit Runge-Kutta single step methods

We consider one step a general  $s$ -stage RK-SSM according to Def. 11.4.9 for the autonomous ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ , with smooth right hand side function  $\mathbf{f} : D \subset \mathbb{R}^d \rightarrow \mathbb{R}^d$ :

$$\mathbf{k}_i = \mathbf{f}(\mathbf{y}_0 + h \sum_{j=1}^{i-1} a_{ij} \mathbf{k}_j), \quad i = 1, \dots, s, \quad \mathbf{y}_1 = \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{k}_i.$$

We perform linearization at  $\mathbf{y}_0$  and ignore all terms at least quadratic in the timestep size  $h$ :

$$\mathbf{k}_i \approx \mathbf{f}(\mathbf{y}_0) + D\mathbf{f}(\mathbf{y}_0)h \sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j, \quad i = 1, \dots, s, \quad \mathbf{y}_1 = \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{k}_i.$$

We find that for small timesteps

the discrete evolution of the RK-SSM for  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  in the state  $\mathbf{y}^*$  is close to the discrete evolution of the same RK-SSM applied to the linearization (12.2.12) of the ODE in  $\mathbf{y}^*$ .

By straightforward manipulations of the defining equations of an explicit RK-SSM we find that, if

- $(\mathbf{y}_k)_k$  is the sequence of states generated by the RK-SSM applied to the affine-linear ODE  $\dot{\mathbf{y}} = \mathbf{M}(\mathbf{y} - \mathbf{y}_0) + \mathbf{b}$ ,  $\mathbf{M} \in \mathbb{C}^{d,d}$  regular,
- $(\mathbf{w}_k)_k$  is the sequence of states generated by the same RK-SSM applied to the linear ODE  $\dot{\mathbf{w}} = \mathbf{M}\mathbf{w}$  and  $\mathbf{w}_0 := \mathbf{M}^{-1}\mathbf{b}$ , then

$$\mathbf{w}_k = \mathbf{y}_k - \mathbf{y}_0 + \mathbf{M}^{-1}\mathbf{b}.$$

- The analysis of the behavior of an RK-SSM for an affine-linear ODE can be reduced to understanding its behavior for a linear ODE with the same matrix.

Combined with the insights from § 12.1.42 this means that

for small timestep the behavior of an explicit RK-SSM applied to  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  close to the state  $\mathbf{y}^*$  is determined by the eigenvalues of the Jacobian  $D\mathbf{f}(\mathbf{y}^*)$ .

In particular, if  $D\mathbf{f}(\mathbf{y}^*)$  has at least one eigenvalue whose modulus is large, then an exponential drift-off of the approximate states  $\mathbf{y}_k$  away from  $\mathbf{y}^*$  can only be avoided for sufficiently small timestep, again a **timestep constraint**.

### How to distinguish stiff initial value problems

An initial value problem for an autonomous ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  will probably be stiff, if, for substantial periods of time,

$$\min\{\operatorname{Re} \lambda : \lambda \in \sigma(D\mathbf{f}(\mathbf{y}(t)))\} \ll 0, \tag{12.2.15}$$

$$\max\{0, \operatorname{Re} \lambda : \lambda \in \sigma(D\mathbf{f}(\mathbf{y}(t)))\} \approx 0, \tag{12.2.16}$$

where  $t \mapsto \mathbf{y}(t)$  is the solution trajectory and  $\sigma(\mathbf{M})$  is the spectrum of the matrix  $\mathbf{M}$ , see Def. 7.1.1.

The condition (12.2.16) has to be read as “the real parts of all eigenvalues are below a bound with small modulus”. If this is not the case, then the exact solution will experience blow-up. It will change drastically over very short periods of time and small timesteps will be required anyway in order to resolve this.

### Example 12.2.17 (Predicting stiffness of non-linear IVPs)

❶ We consider the IVP from Ex. 12.0.1:

$$\text{IVP considered: } \dot{y} = f(y) := \lambda y^2(1 - y), \quad \lambda := 500, \quad y(0) = \frac{1}{100}.$$

We find

$$f'(y) = \lambda(2y - 3y^2) \Rightarrow f'(1) = -\lambda.$$

Hence, in case  $\lambda \gg 1$  as in Fig. 391, we face a stiff problem close to the stationary state  $y = 1$ . The observations made in Fig. 391 exactly match this prediction.

❷ The solution of the IVP from Ex. 12.2.5

$$\dot{\mathbf{y}} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{y} + \lambda(1 - \|\mathbf{y}\|^2) \mathbf{y}, \quad \|\mathbf{y}_0\|_2 = 1. \quad (12.2.6)$$

satisfies  $\|\mathbf{y}(t)\|_2 = 1$  for all times. Using the product rule (2.4.9) of multi-dimensional differential calculus, we find

$$\begin{aligned} D\mathbf{f}(\mathbf{y}) &= \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} + \lambda \left( -2\mathbf{y}\mathbf{y}^\top + (1 - \|\mathbf{y}\|_2^2) \mathbf{I} \right). \\ \blacktriangleright \quad \sigma(D\mathbf{f}(\mathbf{y})) &= \left\{ -\lambda - \sqrt{\lambda^2 - 1}, -\lambda + \sqrt{\lambda^2 - 1} \right\}, \quad \text{if } \|\mathbf{y}\|_2 = 1. \end{aligned}$$

Thus, for  $\lambda \gg 1$ ,  $D\mathbf{f}(\mathbf{y}(t))$  will always have an eigenvalue with large negative real part, whereas the other eigenvalue is close to zero: the IVP is stiff.

### Remark 12.2.18 (Characteristics of stiff IVPs)

Often one can already tell from the expected behavior of the solution of an IVP, which is often clear from the modeling context, that one has to brace for stiffness.

Typical features of stiff IVPs:

- ✿ Presence of **fast transients** in the solution, see Ex. 12.1.1, Ex. 12.1.33,
- ✿ Occurrence of **strongly attractive** fixed points/limit cycles, see Ex. 12.2.5

## 12.3 Implicit Runge-Kutta Single Step Methods

**Explicit** Runge-Kutta single step method cannot escape tight timestep constraints for stiff IVPs that may render them inefficient, see § 12.1.47. In this section we are going to augment the class of Runge-Kutta methods by timestepping schemes that can cope well with stiff IVPs.



*Supplementary reading.* [15, Sect. 11.6.2], [63, Sect. 11.8.3]

### 12.3.1 The implicit Euler method for stiff IVPs

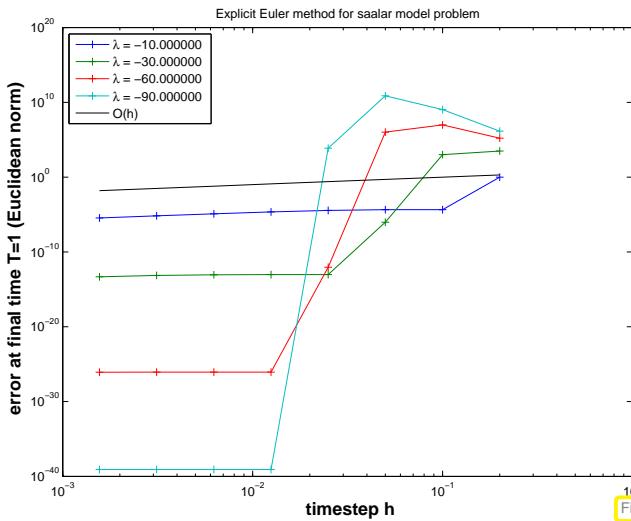
#### Example 12.3.1 (Euler methods for stiff decay IVP)

We revisit the setting of Ex. 12.1.1 and again consider Euler methods for the decay IVP

$$\dot{y} = \lambda y , \quad y(0) = 1 , \quad \lambda < 0 .$$

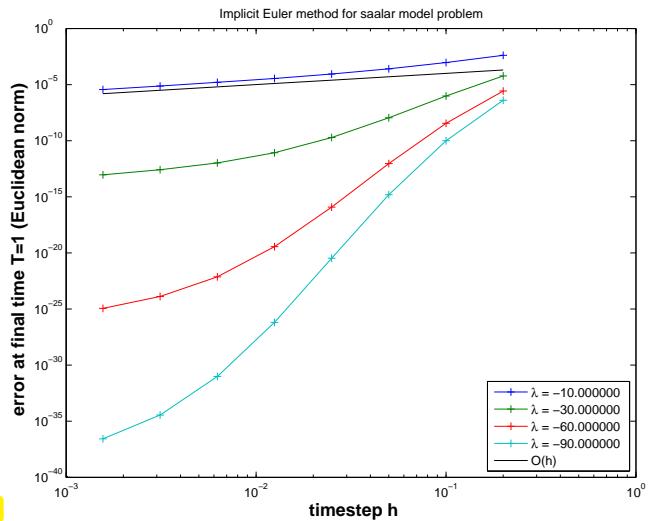
We apply both the explicit Euler method (11.2.7) and the implicit Euler method (11.2.13) with uniform timesteps  $h = 1/N$ ,  $N \in \{5, 10, 20, 40, 80, 160, 320, 640\}$  and monitor the error at final time  $T = 1$  for different values of  $\lambda$ .

##### Explicit Euler method (11.2.7)



$\lambda$  large: blow-up of  $y_k$  for large timestep  $h$

##### Implicit Euler method (11.2.13)



$\lambda$  large: stable for all timesteps  $h > 0$  !

We observe onset of convergence of the implicit Euler method already for large timesteps  $h$ .

#### (12.3.2) Linear model problem analysis: implicit Euler method

We follow the considerations of § 12.1.2 and consider the *implicit* Euler method (11.2.13) for the

$$\text{linear model problem: } \dot{y} = \lambda y , \quad y(0) = y_0 , \quad \text{with } \operatorname{Re} \lambda \ll 0 , \quad (12.1.3)$$

with *exponentially decaying* (maybe oscillatory for  $\operatorname{Im} \lambda \neq 0$ ) exact solution

$$y(t) = y_0 \exp(\lambda t) \rightarrow 0 \quad \text{for } t \rightarrow \infty .$$

The recursion of the implicit Euler method for (12.1.3) is defined by

$$(11.2.13) \text{ for } f(y) = \lambda y \Rightarrow y_{k+1} = y_k + \lambda h y_{k+1} \quad k \in \mathbb{N}_0 . \quad (12.3.3)$$

► generated sequence  $y_k := \left( \frac{1}{1 - \lambda h} \right)^k y_0 .$  (12.3.4)

$\Rightarrow \quad \operatorname{Re} \lambda < 0 \Rightarrow \lim_{k \rightarrow \infty} y_k = 0 \quad \forall h > 0 !$  (12.3.5)

No timestep constraint: qualitatively correct behavior of  $(y_k)_k$  for  $\operatorname{Re} \lambda < 0$  and any  $h > 0$ !

As in § 12.1.39 this analysis can be extended to linear systems of ODEs  $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ ,  $\mathbf{M} \in \mathbb{C}^{d,d}$ , by means of **diagonalization**.

As in § 12.1.28 and § 12.1.39 we assume that  $\mathbf{M}$  can be diagonalized, that is (12.1.30) holds:  $\mathbf{V}^{-1}\mathbf{M}\mathbf{V} = \mathbf{D}$  with a diagonal matrix  $\mathbf{D} \in \mathbb{C}^{d,d}$  containing the eigenvalues of  $\mathbf{M}$  on its diagonal. Next, apply the **decoupling by diagonalization** idea to the recursion of the implicit Euler method.

$$\mathbf{V}^{-1}\mathbf{y}_{k+1} = \mathbf{V}^{-1}\mathbf{y}_k + h \underbrace{\mathbf{V}^{-1}\mathbf{M}\mathbf{V}}_{=\mathbf{D}} (\mathbf{V}^{-1}\mathbf{y}_{k+1}) \stackrel{\mathbf{z}_k := \mathbf{V}^{-1}\mathbf{y}_k}{\Leftrightarrow} \underbrace{(\mathbf{z}_{k+1})_i = \frac{1}{1 - \lambda_i h} (\mathbf{z}_k)_i}_{\hat{\triangleq} \text{ implicit Euler step for } \dot{z}_i = \lambda_i z_i} . \quad (12.3.6)$$

Crucial insight:

For any timestep, the implicit Euler method generates exponentially decaying solution sequences  $(\mathbf{y}_k)_{k=0}^{\infty}$  for  $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$  with diagonalizable matrix  $\mathbf{M} \in \mathbb{R}^{d,d}$  with eigenvalues  $\lambda_1, \dots, \lambda_d$ , if  $\operatorname{Re} \lambda_i < 0$  for all  $i = 1, \dots, d$ .

Thus we expect that the implicit Euler method will not face stability induced timestep constraints for stiff problems ( $\rightarrow$  Notion 12.2.9).

### 12.3.2 Collocation single step methods

Unfortunately the implicit Euler method is of first order only, see Ex. 11.3.18. This section presents an algorithm for designing higher order single step methods generalizing the implicit Euler method.

Setting: We consider the general ordinary differential equation  $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ ,  $\mathbf{f} : I \times D \rightarrow \mathbb{R}^d$  locally Lipschitz continuous, which guarantees the local existence of unique solutions of initial value problems, see Thm. 11.1.32.

We define the single step method through specifying the first step  $\mathbf{y}_0 = \mathbf{y}(t_0) \rightarrow \mathbf{y}_1 \approx \mathbf{y}(t_1)$ , where  $\mathbf{y}_0 \in D$  is the initial step at initial time  $t_0 \in I$ . We assume that the exact solution trajectory  $t \mapsto \mathbf{y}(t)$  exists on  $[t_0, t_1]$ . Use as a timestepping scheme on a temporal mesh ( $\rightarrow$  § 11.2.2) in the sense of Def. 11.3.5 is straightforward.

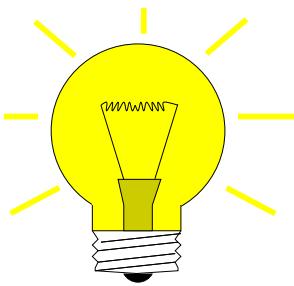
#### (12.3.7) Collocation principle

##### Abstract collocation idea

**Collocation** is a paradigm for the **discretization** ( $\rightarrow$  Rem. 11.3.4) of *differential equations*:

- (I) Write the discrete solution  $u_h$ , a function, as linear combination of  $N \in \mathbb{N}$  sufficiently smooth (basis) functions  $\geq N$  unknown coefficients.
- (II) Demand that  $u_h$  satisfies the differential equation at  $N$  points/times  $\geq N$  equations.

We apply this policy to the differential equation  $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$  on  $[t_0, t_1]$ :



Idea: ① Approximate  $t \mapsto \mathbf{y}(t)$ ,  $t \in [t_0, t_1]$ , by a function  $t \mapsto \mathbf{y}_h(t) \in V$ ,  $V$  an  $d \cdot (s+1)$ -dimensional trial space  $V$  of functions  $[t_0, t_1] \mapsto \mathbb{R}^d \rightarrow$  Item (I).

② Fix  $\mathbf{y}_h \in V$  by imposing collocation conditions

$$\mathbf{y}_h(t_0) = \mathbf{y}_0 , \quad \dot{\mathbf{y}}_h(\tau_j) = \mathbf{f}(\tau_j, \mathbf{y}_h(\tau_j)) , \quad j = 1, \dots, s , \quad (12.3.9)$$

for collocation points  $t_0 \leq \tau_1 < \dots < \tau_s \leq t_1 \rightarrow$  Item (II).

③ Choose  $\mathbf{y}_1 := \mathbf{y}_h(t_1)$ .

Our choice (the “standard option”):

(Componentwise) polynomial trial space  $V = (\mathcal{P}_s)^d$

Recalling  $\dim \mathcal{P}_s = s+1$  from Thm. 3.2.2 we see that our choice makes the number  $N := d(s+1)$  of collocation conditions match the dimension of the trial space  $V$ .

Now we want to derive a concrete representation for the polynomial  $\mathbf{y}_h$ . We draw on concepts introduced in Section 3.2.2. We define the collocation points as

$$\tau_j := t_0 + c_j h , \quad j = 1, \dots, s , \quad \text{for } 0 \leq c_1 < c_2 < \dots < c_s \leq 1 , \quad h := t_1 - t_0 .$$

Let  $\{L_j\}_{j=1}^s \subset \mathcal{P}_{s-1}$  denote the set of Lagrange polynomials of degree  $s-1$  associated with the node set  $\{c_j\}_{j=1}^s$ , see (3.2.11). They satisfy  $L_j(c_i) = \delta_{ij}$ ,  $i, j = 1, \dots, s$  and form a basis of  $\mathcal{P}_{s-1}$ .

In each of its  $d$  components, the derivative  $\dot{\mathbf{y}}_h$  is a polynomial of degree  $s-1$ :  $\dot{\mathbf{y}} \in (\mathcal{P}_{s-1})^d$ . Hence, it has the following representation, compare (3.2.13).

$$\dot{\mathbf{y}}_h(t_0 + \tau h) = \sum_{j=1}^s \dot{\mathbf{y}}_h(t_0 + c_j h) L_j(\tau) . \quad (12.3.10)$$

As  $\tau_j = t_0 + c_j h$ , the comcollocation conditions make it possible to replace  $\dot{\mathbf{y}}_h(c_j h)$  with an expression in the right hand side function  $\mathbf{f}$ :

$$(12.3.9) \quad \dot{\mathbf{y}}_h(t_0 + \tau h) = \sum_{j=1}^s \mathbf{k}_j L_j(\tau) \quad \text{with “coefficients” } \mathbf{k}_j := f(t_0 + c_j h, \mathbf{y}_h(t_0 + c_j h)) .$$

Next we integrate and use  $\mathbf{y}_h(t_0) = \mathbf{y}_0$

$$\Rightarrow \mathbf{y}_h(t_0 + \tau h) = \mathbf{y}_0 + h \sum_{j=1}^s \mathbf{k}_j \int_0^{\tau} L_j(\zeta) d\zeta .$$

This yields the following formulas for the computation of  $\mathbf{y}_1$ , which characterize the  $s$ -stage collocation single step method induced by the (normalized) collocation points  $c_j \in [0, 1]$ ,  $j = 1, \dots, s$ .

$$\begin{aligned} \mathbf{k}_i &= f(t_0 + c_i h, \mathbf{y}_0 + h \sum_{j=1}^s a_{ij} \mathbf{k}_j) , & \text{where } a_{ij} &:= \int_0^{c_i} L_j(\tau) d\tau , \\ \mathbf{y}_1 := \mathbf{y}_h(t_1) &= \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{k}_i . & b_i &:= \int_0^1 L_i(\tau) d\tau . \end{aligned} \quad (12.3.11)$$

Note that, since arbitrary  $\mathbf{y}_0 \in D$ ,  $t_0, t_1 \in I$  were admitted, this defines a discrete evolution  $\Psi : I \times I \times D \rightarrow \mathbb{R}^d$  by  $\Psi^{t_0, t_1} \mathbf{y}_0 := \mathbf{y}_h(t_1)$ .

### Remark 12.3.12 (Implicit nature of collocation single step methods)

Note that (12.3.11) represents a generically non-linear system of  $s \cdot d$  equations for the  $s \cdot d$  components of the vectors  $\mathbf{k}_i$ ,  $i = 1, \dots, s$ . Usually, it will not be possible to obtain  $\mathbf{k}_i$  by a fixed number of evaluations of  $\mathbf{f}$ . For this reason the single step methods defined by (12.3.11) are called **implicit**.

With similar arguments as in Rem. 11.2.14 one can prove that for sufficiently small  $|t_1 - t_0|$  a unique solution for  $\mathbf{k}_1, \dots, \mathbf{k}_s$  can be found.

### (12.3.13) Collocation single step methods and quadrature

Clearly, in the case  $d = 1$ ,  $f(t, \mathbf{y}) = f(t)$ ,  $\mathbf{y}_0 = 0$  the computation of  $\mathbf{y}_1$  boils down to the evaluation of a quadrature formula on  $[t_0, t_1]$ , because from (12.3.11) we get

$$\mathbf{y}_1 = h \sum_{i=1}^s b_i f(t_0 + c_i h), \quad b_i := \int_0^1 L_i(\tau) d\tau, \quad (12.3.14)$$

which is a polynomial quadrature formula (5.2.2) on  $[0, 1]$  with nodes  $c_j$  transformed to  $[t_0, t_1]$  according to (5.1.5).

### Experiment 12.3.15 (Empiric Convergence of collocation single step methods)

We consider the scalar logistic ODE (11.1.6) with parameter  $\lambda = 10$  ( $\rightarrow$  only mildly stiff), initial state  $\mathbf{y}_0 = 0.01$ ,  $T = 1$ .

Numerical integration by timestepping with uniform timestep  $h$  based on collocation single step method (12.3.11).

① Equidistant collocation points,  $c_j = \frac{j}{s+1}$ ,  $j = 1, \dots, s$ .

We observe **algebraic convergence** with the empiric rates

- $s = 1 : p = 1.96$
- $s = 2 : p = 2.03$
- $s = 3 : p = 4.00$
- $s = 4 : p = 4.04$

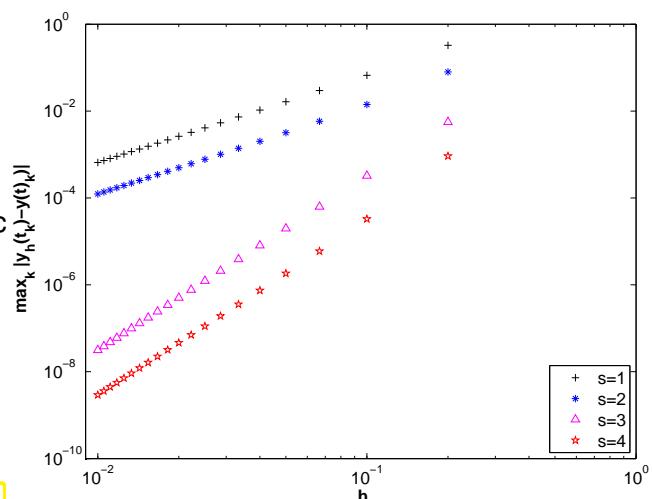


Fig. 411

In this case we conclude the following (empiric) order ( $\rightarrow$  Def. 11.3.21) of the collocation single step

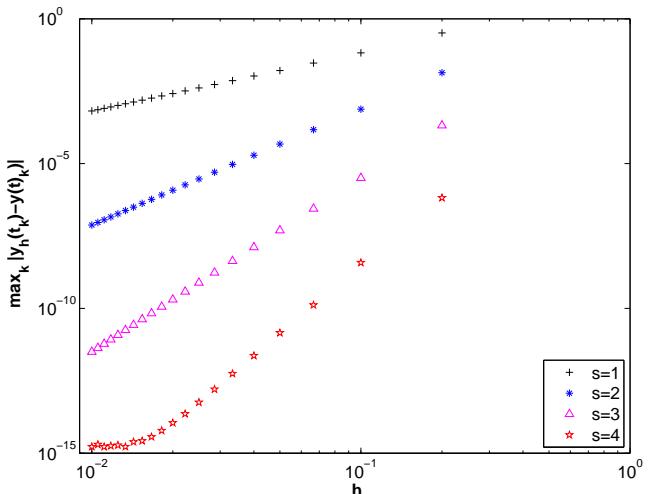
method:

$$\text{(empiric) order} = \begin{cases} s & \text{for even } s, \\ s+1 & \text{for odd } s. \end{cases}$$

- ① Gauss points in  $[0, 1]$  as normalized collocation points  $c_j, j = 1, \dots, s$ .

We observe **algebraic convergence** with the empiric rates

$$\begin{aligned} s=1 &: p=1.96 \\ s=2 &: p=4.01 \\ s=3 &: p=6.00 \\ s=4 &: p=8.02 \end{aligned}$$



Obviously, for the (empiric) order ( $\rightarrow$  Def. 11.3.21) of the **Gauss collocation single step method** holds

$$\text{(empiric) order} = 2s.$$

Note that the 1-stage Gauss collocation single step method is the implicit midpoint method from Section 11.2.3.

### (12.3.16) Order of collocation single step method

What we have observed in Exp. 12.3.15 reflects a fundamental result on collocation single step methods as defined in (12.3.11).

#### Theorem 12.3.17. Order of collocation single step method [18, Satz .6.40]

Provided that  $f \in C^p(I \times D)$ , the order ( $\rightarrow$  Def. 11.3.21) of an  $s$ -stage collocation single step method according to (12.3.11) agrees with the order ( $\rightarrow$  Def. 5.3.1) of the quadrature formula on  $[0, 1]$  with nodes  $c_j$  and weights  $b_j, j = 1, \dots, s$ .

- By Thm. 5.3.21 the  $s$ -stage **Gauss collocation single step method** whose nodes  $c_j$  are chosen as the  $s$  Gauss points on  $[0, 1]$  is of **order  $2s$** .

### 12.3.3 General implicit RK-SSMs

The notations in (12.3.11) have deliberately been chosen to allude to Def. 11.4.9. In that definition it takes only letting the sum in the formula for the increments run up to  $s$  to capture (12.3.11).

**Definition 12.3.18. General Runge-Kutta single step method (cf. Def. 11.4.9)**

For  $b_i, a_{ij} \in \mathbb{R}$ ,  $c_i := \sum_{j=1}^s a_{ij}$ ,  $i, j = 1, \dots, s$ ,  $s \in \mathbb{N}$ , an  $s$ -stage Runge-Kutta single step method (RK-SSM) for the IVP (11.1.20) is defined by

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + h \sum_{j=1}^s a_{ij} \mathbf{k}_j), \quad i = 1, \dots, s, \quad \mathbf{y}_1 := \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{k}_i.$$

As before, the  $\mathbf{k}_i \in \mathbb{R}^d$  are called **increments**.

Note: computation of increments  $\mathbf{k}_i$  may now require the solution of (*non-linear*) systems of equations of size  $s \cdot d$  ( $\rightarrow$  “implicit” method, cf. Rem. 12.3.12)

**General Butcher scheme notation for RK-SSM**

Shorthand notation for Runge-Kutta methods

**Butcher scheme**

Note: now  $\mathfrak{A}$  can be a general  $s \times s$ -matrix.

$$\begin{array}{c|cc} \mathbf{c} & \mathfrak{A} \\ \hline \mathbf{b}^T & \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline b_1 & \cdots & & b_s \end{array}. \quad (12.3.20)$$

Summary: terminology for Runge-Kutta single step methods:

- |   |   |
|---|---|
| $\mathfrak{A}$ strict lower triangular matrix<br>$\mathfrak{A}$ lower triangular matrix | <b>►</b> explicit Runge-Kutta method, Def. 11.4.9<br><b>►</b> diagonally-implicit Runge-Kutta method (DIRK) |
|---|---|

Many of the techniques and much of the theory discussed for explicit RK-SSMs carry over to general (implicit) Runge-Kutta single step methods:

- Sufficient condition for consistence from Cor. 11.4.12
- Algebraic convergence for meshwidth  $h \rightarrow 0$  and the related concept of order ( $\rightarrow$  Def. 11.3.21)
- Embedded methods and algorithms for adaptive stepsize control from Section 11.5

**Remark 12.3.21 (Stage form equations for increments)**

In Def. 12.3.18 instead of the increments we can consider the **stages**

$$\mathbf{g}_i := h \sum_{j=1}^s a_{ij} \mathbf{k}_j, \quad i = 1, \dots, s, \quad \Leftrightarrow \quad \mathbf{k}_i = \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + \mathbf{g}_i). \quad (12.3.22)$$

This leads to the equivalent defining equations in “stage form” for an implicit RK-SSM

$$\mathbf{g}_i = h \sum_{j=1}^s a_{ij} \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + \mathbf{g}_j), \quad \mathbf{y}_1 = \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + \mathbf{g}_i). \quad (12.3.23)$$

In terms of implementation there is no difference.

### Remark 12.3.24 (Solving the increment equations for implicit RK-SSMs)

We reformulate the increment equations in stage form (12.3.23) as a non-linear system of equations in standard form  $F(\mathbf{x}) = \mathbf{0}$ . Unknowns are the total  $s \cdot d$  components of the stage vectors  $\mathbf{g}_i, i = 1, \dots, s$  as defined in (12.3.22).

$$\mathbf{g} = [\mathbf{g}_1, \dots, \mathbf{g}_s]^\top, \quad \mathbf{g}_i := h \sum_{j=1}^s a_{ij} \mathbf{f}(t_0 + c_j h, \mathbf{y}_0 + \mathbf{g}_j) \quad \Rightarrow \quad F(\mathbf{g}) = \mathbf{g} - h(\mathfrak{A} \otimes \mathbf{I}) \begin{pmatrix} \mathbf{f}(t_0 + c_1 h, \mathbf{y}_0 + \mathbf{g}_1) \\ \vdots \\ \mathbf{f}(t_0 + c_s h, \mathbf{y}_0 + \mathbf{g}_s) \end{pmatrix} \stackrel{!}{=} \mathbf{0},$$

where  $\mathbf{I}$  is the  $d \times d$  identity matrix and  $\otimes$  designates the Kronecker product introduced in Def. 1.4.16.

We compute an approximate solution of  $F(\mathbf{g}) = \mathbf{0}$  iteratively by means of the simplified Newton method presented in Rem. 2.4.31. This is a Newton method with “frozen Jacobian”. As  $\mathbf{g} \rightarrow \mathbf{0}$  for  $h \rightarrow 0$ , we choose zero as initial guess:

$$\mathbf{g}^{(k+1)} = \mathbf{g}^{(k)} - D F(\mathbf{0})^{-1} F(\mathbf{g}^{(k)}) \quad k = 0, 1, 2, \dots, \quad \mathbf{g}^{(0)} = \mathbf{0}. \quad (12.3.25)$$

with the Jacobian

$$D F(\mathbf{0}) = \begin{bmatrix} \mathbf{I} - ha_{11} \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_0, \mathbf{y}_0) & \cdots & -ha_{1s} \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_0, \mathbf{y}_0) \\ \vdots & \ddots & \vdots \\ -ha_{s1} \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_0, \mathbf{y}_0) & \cdots & \mathbf{I} - ha_{ss} \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_0, \mathbf{y}_0) \end{bmatrix} \in \mathbb{R}^{sd, sd}. \quad (12.3.26)$$

Obviously,  $D F(\mathbf{0}) \rightarrow \mathbf{I}$  for  $h \rightarrow 0$ . Thus,  $D F(\mathbf{0})$  will be regular for sufficiently small  $h$ .

In each step of the simplified Newton method we have to solve a linear system of equations with coefficient matrix  $D F(\mathbf{0})$ . If  $s \cdot d$  is large, an efficient implementation has to reuse the LU-decomposition of  $D F(\mathbf{0})$ , see Code 2.4.32 and Rem. 1.6.87.

### 12.3.4 Model problem analysis for implicit RK-SSMs

**Model problem analysis** for general Runge-Kutta single step methods ( $\rightarrow$  Def. 12.3.18) runs parallel to that for explicit RK-methods as elaborated in Section 12.1, § 12.1.11. Familiarity with the techniques and results of this section is assumed. The reader is asked to recall the concept of **stability function** from Thm. 12.1.15, the **diagonalization technique** from § 12.1.42, and the definition of **region of (absolute) stability** from Def. 12.1.49.

**Theorem 12.3.27. Stability function of Runge-Kutta methods, cf. Thm. 12.1.15**

[Stability function of general Runge-Kutta methods]

The discrete evolution  $\Psi_\lambda^h$  of an  $s$ -stage Runge-Kutta single step method ( $\rightarrow$  Def. 12.3.18) with Butcher scheme  $\begin{array}{c|cc} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^T \end{array}$  (see (12.3.20)) for the ODE  $\dot{\mathbf{y}} = \lambda \mathbf{y}$  is given by a multiplication with

$$S(z) := \underbrace{1 + z\mathbf{b}^T(\mathbf{I} - z\mathbf{A})^{-1}\mathbf{1}}_{\text{stability function}} = \frac{\det(\mathbf{I} - z\mathbf{A} + z\mathbf{1}\mathbf{b}^T)}{\det(\mathbf{I} - z\mathbf{A})}, \quad z := \lambda h, \quad \mathbf{1} = [1, \dots, 1]^T \in \mathbb{R}^s.$$

**Example 12.3.28 (Regions of stability for simple implicit RK-SSM)**

We determine the Butcher schemes (12.3.20) for simple implicit RK-SSM and apply the formula from Thm. 12.3.27 to compute their stability functions.

- Implicit Euler method:  $\begin{array}{c|c} & 1 \\ \hline & 1 \end{array}$   $\Rightarrow S(z) = \frac{1}{1-z}.$

- Implicit midpoint method:  $\begin{array}{c|cc} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array}$   $\Rightarrow S(z) = \frac{1 + \frac{1}{2}z}{1 - \frac{1}{2}z}.$

Their regions of stability  $S_\Psi$  as defined in Def. 12.1.49 can easily found from the respective stability functions:

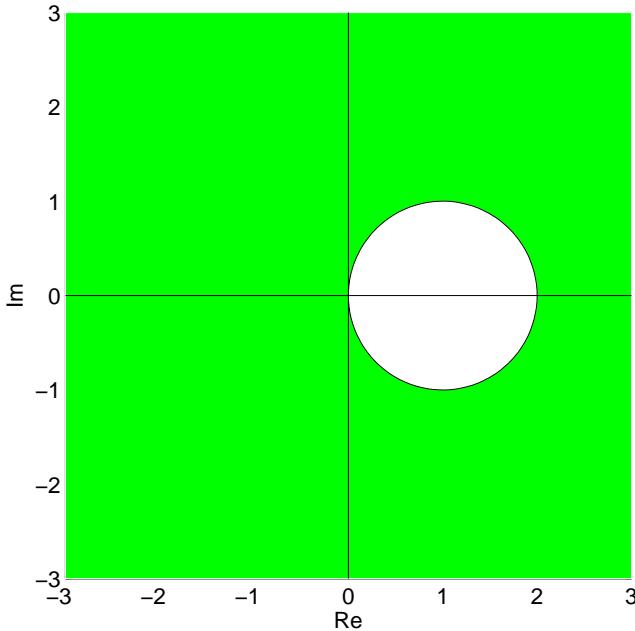
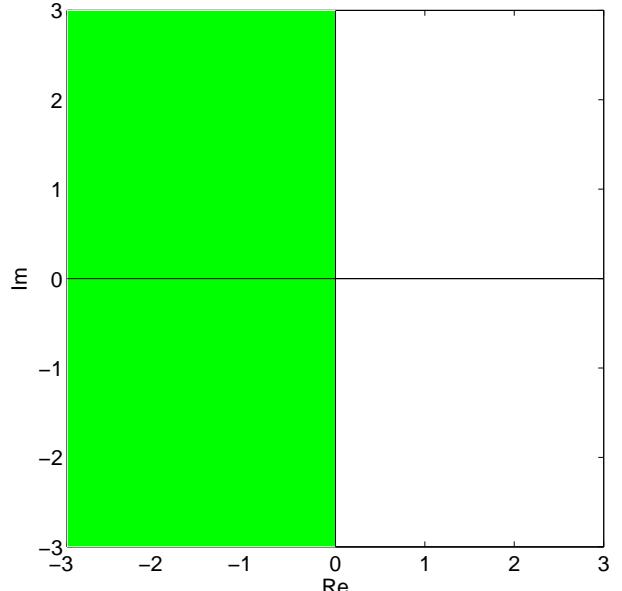


Fig. 413

$S_\Psi$ : implicit Euler method (11.2.13)



$S_\Psi$ : implicit midpoint method (11.2.18)

We see that in both cases  $|S(z)| < 1$ , if  $\operatorname{Re} z < 0$ .

From the determinant formula for the stability function  $S(z)$  we can conclude a generalization of Cor. 12.1.18.

**Corollary 12.3.29. Rational stability function of explicit RK-SSM**

For a consistent ( $\rightarrow$  Def. 11.3.10)  $s$ -stage general Runge-Kutta single step method according to Def. 12.3.18 the stability function  $S$  is a non-constant **rational function** of the form  $S(z) = \frac{P(z)}{Q(z)}$  with polynomials  $P \in \mathcal{P}_s$ ,  $Q \in \mathcal{P}_s$ .

samskip

Of course, a rational function  $z \mapsto S(z)$  can satisfy  $\lim_{|z| \rightarrow \infty} |S(z)| < 1$  as we have seen in Ex. 12.3.28. As a consequence, the region of stability for implicit RK-SSM need not be bounded.

**(12.3.30) A-stability**

A general RK-SSM with stability function  $S$  applied to the scalar linear IVP  $\dot{y} = \lambda y$ ,  $y(0) = y_0 \in \mathbb{C}$ ,  $\lambda \in \mathbb{C}$ , with uniform timestep  $h > 0$  will yield the sequence  $(y_k)_{k=0}^{\infty}$  defined by

$$y_k = S(z)^k y_0 \quad , \quad z = \lambda h . \quad (12.3.31)$$

Hence, the next property of a RK-SSM guarantees that the sequence of approximations decays exponentially whenever the exact solution of the model problem IVP (12.1.3) does so.

**Definition 12.3.32. A-stability of a Runge-Kutta single step method**

A Runge-Kutta single step method with stability function  $S$  is **A-stable**, if

$$\mathbb{C}^- := \{z \in \mathbb{C}: \operatorname{Re} z < 0\} \subset \mathcal{S}_{\Psi} . \quad (\mathcal{S}_{\Psi} \doteq \text{region of stability Def. 12.1.49})$$

From Ex. 12.3.28 we conclude that both the implicit Euler method and the implicit midpoint method are A-stable.

A-stable Runge-Kutta single step methods will not be affected by stability induced timestep constraints when applied to **stiff** IVP ( $\rightarrow$  Notion 12.2.9).

**(12.3.33) “Ideal” region of stability**

In order to reproduce the qualitative behavior of the exact solution, a single step method when applied to the scalar linear IVP  $\dot{y} = \lambda y$ ,  $y(0) = y_0 \in \mathbb{C}$ ,  $\lambda \in \mathbb{C}$ , with uniform timestep  $h > 0$ ,

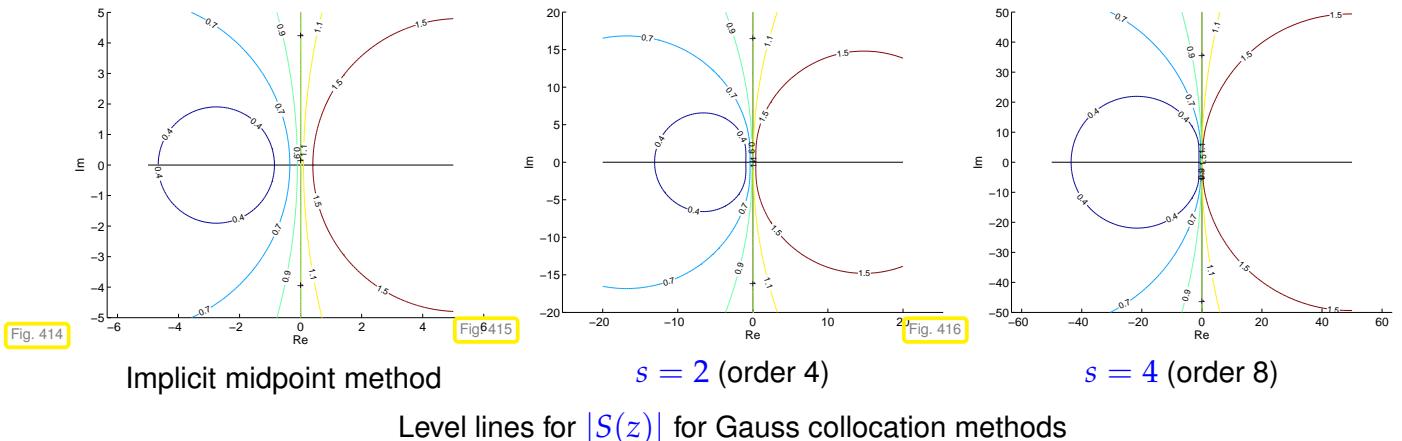
- should yield an *exponentially decaying* sequence  $(y_k)_{k=0}^{\infty}$ , whenever  $\operatorname{Re} \lambda < 0$ ,
- should produce an *exponentially increasing* sequence  $(y_k)_{k=0}^{\infty}$ , whenever  $\operatorname{Re} \lambda > 0$ .

Thus, in light of (12.3.31), we agree that the stability if

$$\text{“ideal” region of stability is } \mathcal{S}_{\Psi} = \mathbb{C}^- . \quad (12.3.34)$$

Are there RK-SSMs that can boast of an ideal region of stability?

Regions of stability of Gauss collocation single step methods, see Exp. 12.3.15:



**Theorem 12.3.35. Region of stability of Gauss collocation single step methods [18, Satz 6.44]**

*s*-stage Gauss collocation single step methods defined by (12.3.11) with the nodes  $c_s$  given by the *s* Gauss points on  $[0, 1]$ , feature the “ideal” stability domain:

$$\mathcal{S}_\Psi = \mathbb{C}^- . \quad (12.3.34)$$

In particular, all Gauss collocation single step methods are A-stable.

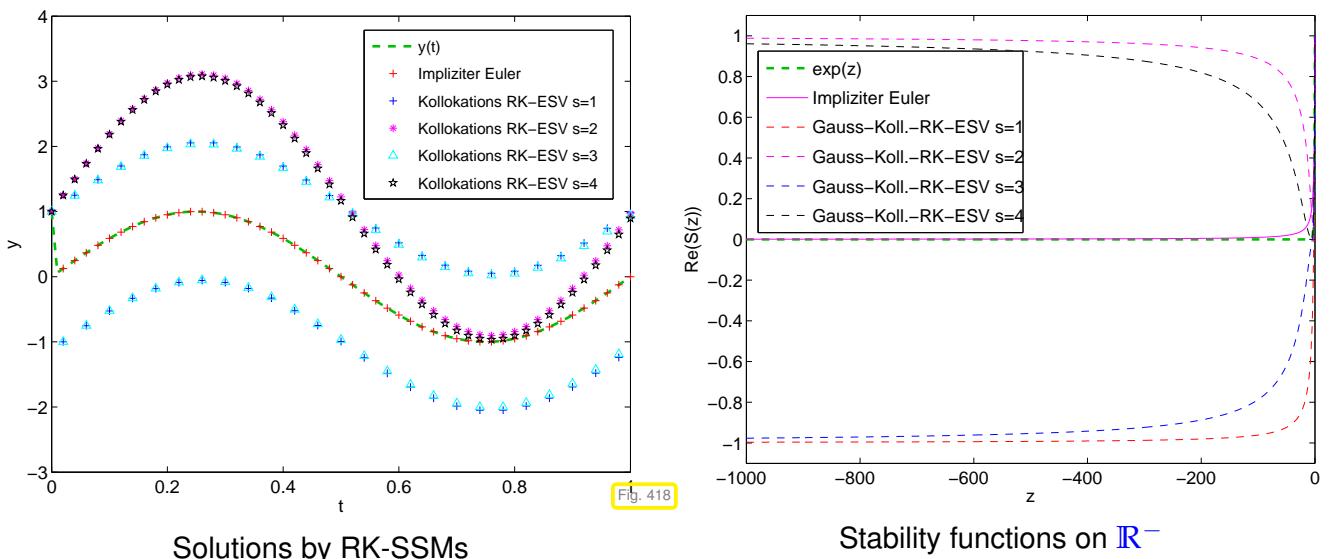
### Experiment 12.3.36 (Implicit RK-SSMs for stiff IVP)

We consider the stiff IVP

$$\dot{y} = -\lambda y + \beta \sin(2\pi t), \quad \lambda = 10^6, \beta = 10^6, \quad y(0) = 1,$$

whose solution essentially is the smooth function  $t \mapsto \sin(2\pi t)$ . Applying the criteria (12.2.15) and (12.2.16) we immediately see that this IVP is extremely stiff.

We solve it with different implicit RK-SSM on  $[0, 1]$  with large uniform timestep  $h = \frac{1}{20}$ .



We observe that Gauss collocation RK-SSMs incur a huge discretization error, whereas the simple implicit Euler method provides a perfect approximation!

Explanation: The stability functions for Gauss collocation RK-SSMs satisfy

$$\lim_{|z| \rightarrow \infty} |S(z)| = 1.$$

Hence, when they are applied to  $\dot{y} = \lambda y$  with extremely large (in modulus)  $\lambda < 0$ , they will produce sequences that decay only very slowly or even oscillate, which misses the very rapid decay of the exact solution. The stability function for the implicit Euler method is  $S(z) = (1 - z)^{-1}$  and satisfies  $\lim_{|z| \rightarrow \infty} S(z) = 0$ , which will mean a fast exponential decay of the  $y_k$ .

### (12.3.37) L-stability

In light of what we learned in the previous experiment we can now state what we expect from the stability function of a Runge-Kutta method that is suitable for stiff IVP ( $\rightarrow$  Notion 12.2.9):

#### Definition 12.3.38. L-stable Runge-Kutta method $\rightarrow$ [42, Ch. 77]

A Runge-Kutta method ( $\rightarrow$  Def. 12.3.18) is **L-stable/asymptotically stable**, if its stability function ( $\rightarrow$  Thm. 12.3.27) satisfies

$$(i) \quad \operatorname{Re} z < 0 \Rightarrow |S(z)| < 1, \quad (12.3.39)$$

$$(ii) \quad \lim_{\operatorname{Re} z \rightarrow -\infty} S(z) = 0. \quad (12.3.40)$$

Remember:

$$\text{L-stable} \Leftrightarrow \text{A-stable} \& "S(-\infty) = 0"$$

#### Remark 12.3.41 (Necessary condition for L-stability of Runge-Kutta methods)

Consider a Runge-Kutta single step method ( $\rightarrow$  Def. 12.3.18) described by the Butcher scheme  $\begin{array}{c|ccccc} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$ .

Assume that  $\mathfrak{A} \in \mathbb{R}^{s,s}$  is regular, which can be fulfilled only for an implicit RK-SSM.

For a rational function  $S(z) = \frac{P(z)}{Q(z)}$  the limit for  $|z| \rightarrow \infty$  exists and can easily be expressed by the leading coefficients of the polynomials  $P$  and  $Q$ :

$$\text{Thm. 12.3.27} \Rightarrow S(-\infty) = 1 - \mathbf{b}^T \mathfrak{A}^{-1} \mathbf{1}. \quad (12.3.42)$$

► If  $\mathbf{b}^T = (\mathfrak{A})_{:,j}^T$  (row of  $\mathfrak{A}$ )  $\Rightarrow S(-\infty) = 0$ . (12.3.43)

Butcher scheme (12.3.20) for L-stable RK-methods, see Def. 12.3.38

$$\triangleright \begin{array}{c|ccccc} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_{s-1} & a_{s-1,1} & \cdots & a_{s-1,s} \\ \hline 1 & b_1 & \cdots & b_s \\ \hline & b_1 & \cdots & b_s \end{array}.$$

A closer look at the coefficient formulas of (12.3.11) reveals that the algebraic condition (12.3.43) will automatically satisfied for a collocation single step method with  $c_s = 1$ !

### Example 12.3.44 (L-stable implicit Runge-Kutta methods)

There is a family of  $s$ -point quadrature formulas on  $[0, 1]$  with a node located in 1 and (maximal) order  $2s - 1$ : **Gauss-Radau formulas**. They induce the **L-stable** Gauss-Radau collocation single step methods of order  $2s - 1$  according to Thm. 12.3.17.

$$\begin{array}{c|c} 1 & 1 \\ \hline 1 & \end{array}$$

$$\begin{array}{c|cc} & \frac{5}{12} & -\frac{1}{12} \\ \hline \frac{1}{3} & \frac{3}{4} & \frac{1}{4} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}$$

Implicit Euler method

Radau RK-SSM, order 3

$$\begin{array}{c|cccc} & \frac{4-\sqrt{6}}{10} & \frac{88-7\sqrt{6}}{360} & \frac{296-169\sqrt{6}}{1800} & \frac{-2+3\sqrt{6}}{225} \\ \hline \frac{4+\sqrt{6}}{10} & \frac{296+169\sqrt{6}}{1800} & \frac{88+7\sqrt{6}}{360} & \frac{-2-3\sqrt{6}}{225} & \\ 1 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} & \\ \hline & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} & \end{array}$$

Radau RK-SSM, order 5

The stability functions of  $s$ -stage Gauss-Radau collocation SSMs are rational functions of the form

$$S(z) = \frac{P(z)}{Q(z)}, \quad P \in \mathcal{P}_{s-1}, Q \in \mathcal{P}_s.$$

Beware that also " $S(\infty) = 0$ ", which means that Gauss-Radau methods when applied to problems with fast exponential blow-up may produce a spurious decaying solution.

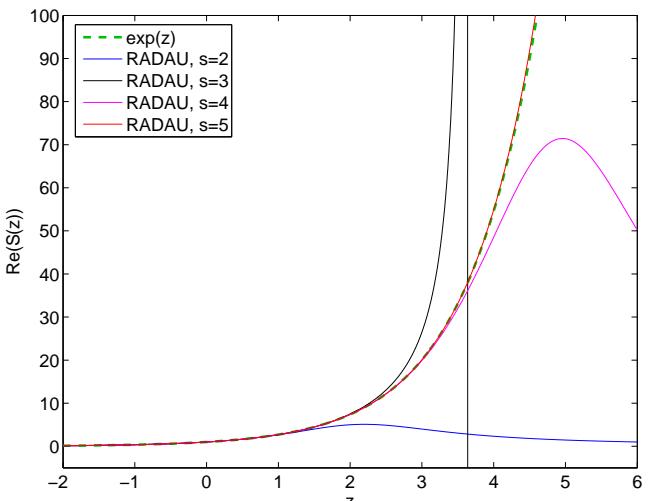


Fig. 419

Level lines of stability functions of  $s$ -stage Gauss-Radau collocation SSMs:

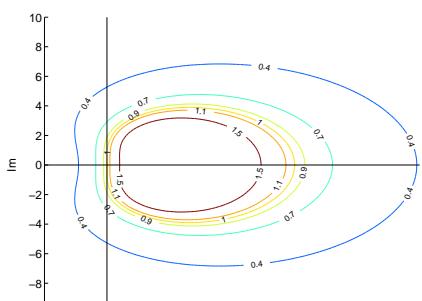


Fig. 420

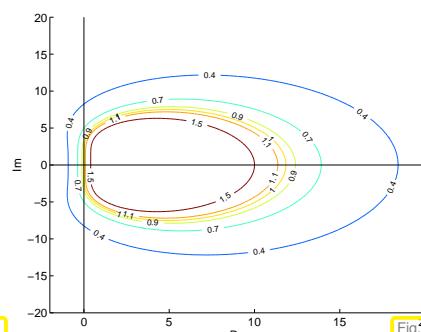
 $s = 2$ 

Fig. 421

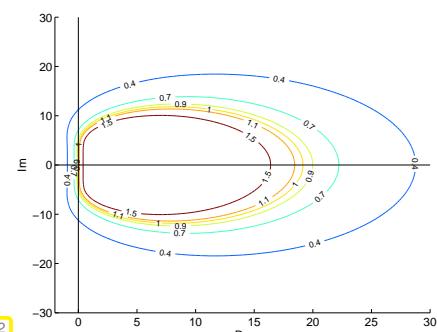
 $s = 3$ 

Fig. 422

 $s = 4$ 

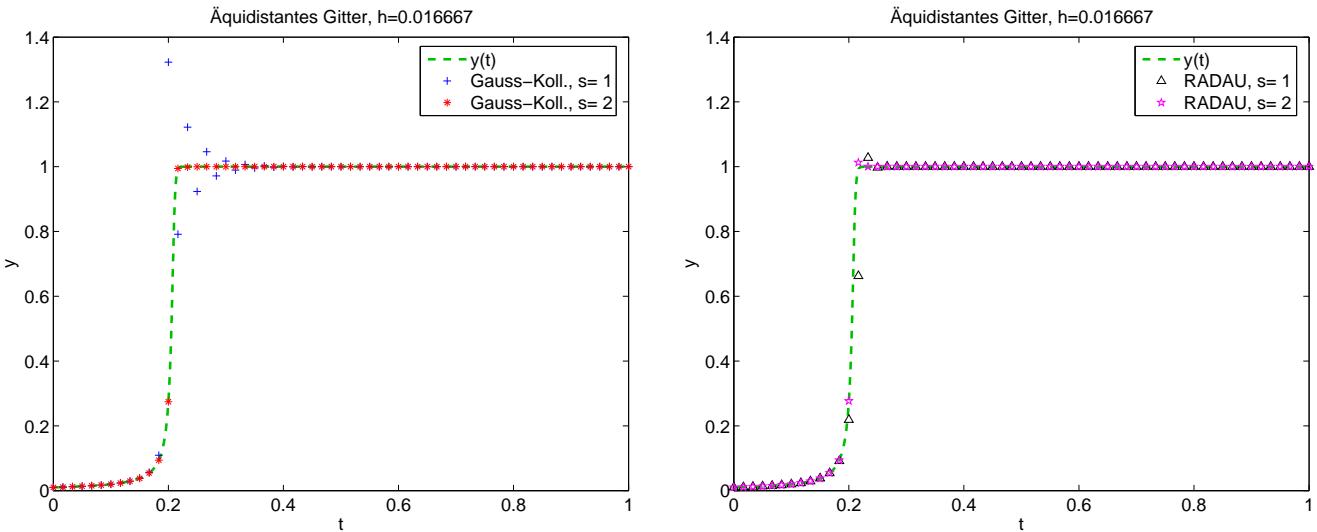
Further information about Radau-Runge-Kutta single step methods can be found in [42, Ch. 79].

### Experiment 12.3.45 (Gauss-Radau collocation SSM for stiff IVP)

We revisit the stiff IVP from Ex. 12.0.1

$$\dot{y}(t) = \lambda y^2(1 - y), \quad \lambda = 500, \quad y(0) = \frac{1}{100}.$$

We compare the sequences generated by 1-stage and 2-stage Gauss collocation and Gauss-Radau collocation SSMs, respectively (uniform timestep).



The 2nd-order Gauss collocation SSM (implicit midpoint method) suffers from spurious oscillations when homing in on the stable stationary state  $y = 1$ . The explanation from Exp. 12.3.36 also applies to this example.

The fourth-order Gauss method is already so accurate that potential overshoots when approaching  $y = 1$  are damped fast enough.

## 12.4 Semi-implicit Runge-Kutta Methods



*Supplementary reading.* [42, Ch. 80]



The equations fixing the increments  $\mathbf{k}_i \in \mathbb{R}^d$ ,  $i = 1, \dots, s$ , for an  $s$ -stage implicit RK-method constitute a (Non-)linear system of equations with  $s \cdot d$  unknowns.

Expensive iterations needed to find  $\mathbf{k}_i$  ?

Remember that we compute approximate solutions anyway, and the increments are weighted with the stepsize  $h \ll 1$ , see Def. 12.3.18. So there is no point in determining them with high accuracy!



Idea: Use only a fixed *small* number of Newton steps to solve for the  $\mathbf{k}_i$ ,  $i = 1, \dots, s$ .

Extreme case: use only a single Newton step! Let's try.

### Example 12.4.1 (Linearization of increment equations)

- We consider an Initial value problem for logistic ODE, see Ex. 11.1.5

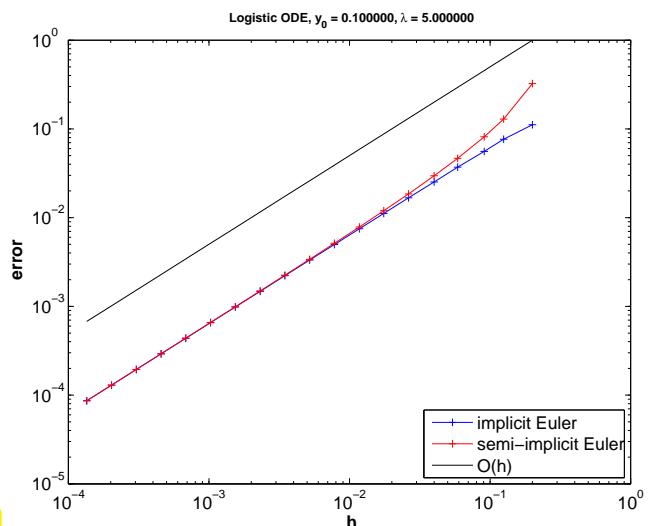
$$\dot{y} = \lambda y(1 - y) , \quad y(0) = 0.1 , \quad \lambda = 5 .$$

- We use the implicit Euler method (11.2.13) with uniform timestep  $h = 1/n$ ,  $n \in \{5, 8, 11, 17, 25, 38, 57, 85, 128, 192, 288, 432, 649, 973, 1460, 2189, 3284, 4926, 7389\}$ .

& approximate computation of  $y_{k+1}$  by  
1 Newton step with initial guess  $y_k$

= semi-implicit Euler method

- Measured error  $\text{err} = \max_{j=1,\dots,n} |y_j - y(t_j)|$



From (11.2.13) with timestep  $h > 0$

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}(\mathbf{y}_{k+1}) \Leftrightarrow F(\mathbf{y}_{k+1}) := \mathbf{y}_{k+1} - h\mathbf{f}(\mathbf{y}_{k+1}) - \mathbf{y}_k = 0 .$$

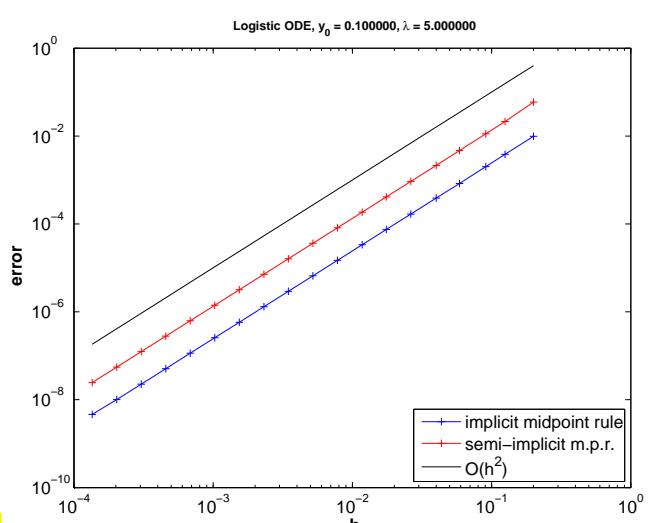
One Newton step (2.4.1) applied to  $F(\mathbf{y}) = 0$  with initial guess  $\mathbf{y}_k$  yields

$$\mathbf{y}_{k+1} = \mathbf{y}_k - D\mathbf{f}(\mathbf{y}_k)^{-1}F(\mathbf{y}_k) = \mathbf{y}_k + (I - hD\mathbf{f}(\mathbf{y}_k))^{-1}h\mathbf{f}(\mathbf{y}_k) .$$

Note: for linear ODE with  $\mathbf{f}(\mathbf{y}) = \mathbf{A}\mathbf{y}$ ,  $\mathbf{A} \in \mathbb{R}^{d,d}$ , we recover the original implicit Euler method!

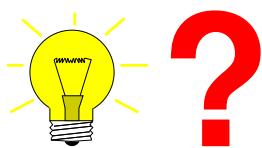
Observation: Approximate evaluation of defining equation for  $\mathbf{y}_{k+1}$  preserves 1st order convergence.

- Now, implicit midpoint method (11.2.18), uniform timestep  $h = 1/n$  as above
- & approximate computation of  $y_{k+1}$  by 1 Newton step, initial guess  $y_k$
- Fehlermass  $\text{err} = \max_{j=1,\dots,n} |y_j - y(t_j)|$



We still observe second-order convergence!

Try: Use linearized increment equations for implicit RK-SSM



$$\mathbf{k}_i := \mathbf{f}(\mathbf{y}_0 + h \sum_{j=1}^s a_{ij} \mathbf{k}_j), \quad i = 1, \dots, s$$

$$\mathbf{k}_i = \mathbf{f}(\mathbf{y}_0) + h D\mathbf{f}(\mathbf{y}_0) \left( \sum_{j=1}^s a_{ij} \mathbf{k}_j \right), \quad i = 1, \dots, s. \quad (12.4.2)$$

The good news is that all results about stability derived from model problem analysis ( $\rightarrow$  Section 12.1) remain valid despite linearization of the increment equations:

Linearization does nothing for linear ODEs  $\geq$  stability function ( $\rightarrow$  Thm. 12.3.27) not affected!

The bad news is that the preservation of the order observed in Ex. 12.4.1 will no longer hold in the general case.

### Example 12.4.3 (Convergence of naive semi-implicit Radau method)

- We consider an IVP for the logistic ODE from Ex. 11.1.5:

$$\dot{y} = \lambda y(1 - y), \quad y(0) = 0.1, \quad \lambda = 5.$$

- 2-stage Radau RK-SSM, Butcher scheme

$$\begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ \hline 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}, \quad (12.4.4)$$

order = 3, see Ex. 12.3.44.

- Increments from linearized equations (12.4.2)
- We monitor the error through  $\text{err} = \max_{j=1,\dots,n} |y_j - y(t_j)|$

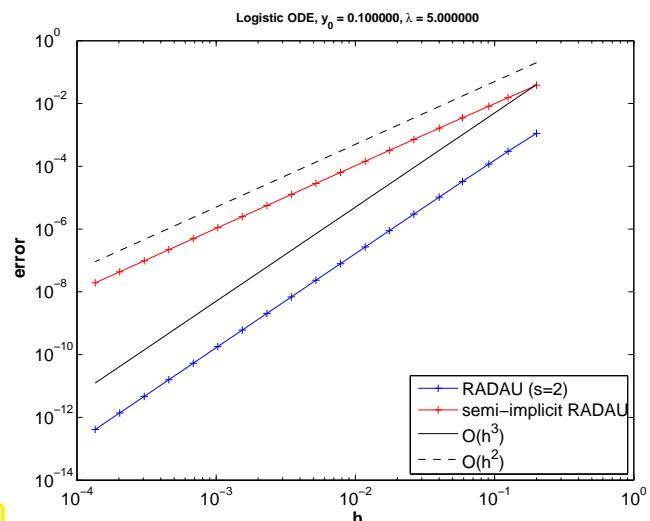


Fig. 425

Loss of order due to linearization !

### (12.4.5) Rosenbrock-Wanner methods

We have just seen that the simple linearization according to (12.4.2) will degrade the order of implicit RK-SSMs and leads to a substantial loss of accuracy. This is not an option.

Yet, the idea behind (12.4.2) has been refined. One does not start from a known RK-SSM, but introduces

general coefficients for structurally linear increment equations.



Class of  $s$ -stage semi-implicit (linearly implicit) Runge-Kutta methods (Rosenbrock-Wanner (ROW) methods):

$$\begin{aligned} (\mathbf{I} - ha_{ii}\mathbf{J})\mathbf{k}_i &= \mathbf{f}(\mathbf{y}_0 + h \sum_{j=1}^{i-1} (a_{ij} + d_{ij})\mathbf{k}_j) - h\mathbf{J} \sum_{j=1}^{i-1} d_{ij}\mathbf{k}_j, \quad \mathbf{J} = D\mathbf{f}(\mathbf{y}_0), \\ \mathbf{y}_1 &:= \mathbf{y}_0 + \sum_{j=1}^s b_j \mathbf{k}_j. \end{aligned} \quad (12.4.6)$$

Then the coefficients  $a_{ij}$ ,  $d_{ij}$ , and  $b_i$  are determined from order conditions by solving large non-linear systems of equations.

In each step  $s$  linear systems with coefficient matrices  $\mathbf{I} - ha_{ii}\mathbf{J}$  have to be solved. For methods used in practice one often demands that  $a_{ii} = \gamma$  for all  $i = 1, \dots, s$ . As a consequence, we have to solve  $s$  linear systems with the same coefficient matrix  $\mathbf{I} - h\gamma\mathbf{J} \in \mathbb{R}^{d,d}$ , which permits us to reuse LU-factorizations, see Rem. 1.6.87.

#### Remark 12.4.7 (Adaptive integrator for stiff problems in MATLAB)

A ROW method is the basis for the standard integrator that MATLAB offers for stiff problems:

```
Handle of type @ (t, y) J(t, y) to Jacobian Df : I × D ↦ ℝd,d
opts = odeset ('abstol', atol, 'reltol', rtol, 'Jacobian', J)
[t, y] = ode23s (odefun, tspan, y0, opts);
```

Stepsize control according to policy of Section 11.5:



## 12.5 Splitting methods

### (12.5.1) Splitting idea: composition of partial evolutions

Many relevant ordinary differential equations feature a right hand side function that is the sum to two (or more) terms. Consider an autonomous IVP with a right hand side function that can be split in an additive fashion:

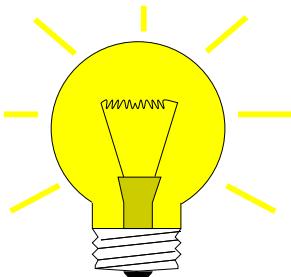
$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) + \mathbf{g}(\mathbf{y}) , \quad \mathbf{y}(0) = \mathbf{y}_0 , \quad (12.5.2)$$

with  $\mathbf{f} : D \subset \mathbb{R}^d \mapsto \mathbb{R}^d$ ,  $\mathbf{g} : D \subset \mathbb{R}^d \mapsto \mathbb{R}^d$  “sufficiently smooth”, locally Lipschitz continuous ( $\rightarrow$  Def. 11.1.28).

Let us introduce the evolution operators ( $\rightarrow$  Def. 11.1.39) for both summands:

$$\begin{array}{ll} \text{(Continuous) evolution maps:} & \Phi_f^t \leftrightarrow \text{ODE } \dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \\ & \Phi_g^t \leftrightarrow \text{ODE } \dot{\mathbf{y}} = \mathbf{g}(\mathbf{y}). \end{array}$$

Temporarily we assume that both  $\Phi_f^t$ ,  $\Phi_g^t$  are available in the form of analytic formulas or highly accurate approximations.

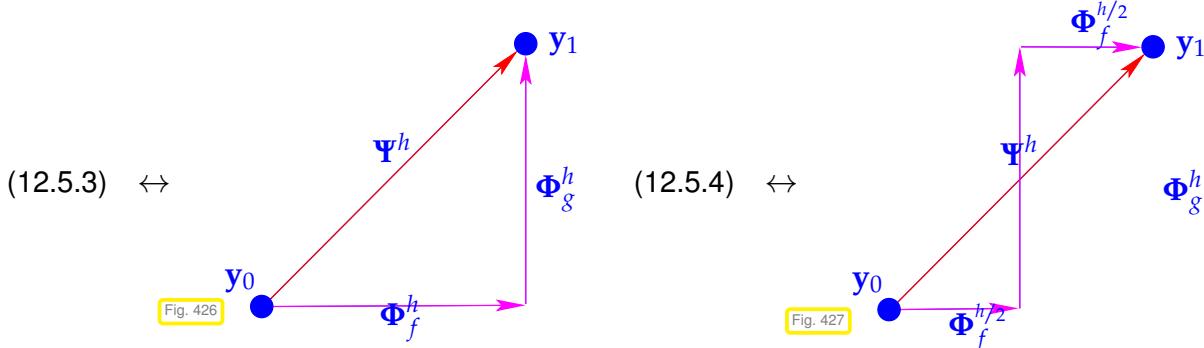


Idea: Build single step methods ( $\rightarrow$  Def. 11.3.5) based on the following discrete evolutions

$$\text{Lie-Trotter splitting: } \Psi^h = \Phi_g^h \circ \Phi_f^h, \quad (12.5.3)$$

$$\text{Strang splitting: } \Psi^h = \Phi_f^{h/2} \circ \Phi_g^h \circ \Phi_f^{h/2}. \quad (12.5.4)$$

These splittings are easily remembered in graphical form:



Note that over many timesteps the Strang splitting approach is not more expensive than Lie-Trotter splitting, because the actual implementation of (12.5.4) should be done as follows:

$$\begin{aligned} \mathbf{y}_{1/2} &:= \Phi_f^{h/2}, & \mathbf{y}_1 &:= \Phi_g^h \mathbf{y}_{1/2}, \\ \mathbf{y}_{3/2} &:= \Phi_f^h \mathbf{y}_1, & \mathbf{y}_2 &:= \Phi_g^h \mathbf{y}_{3/2}, \\ \mathbf{y}_{5/2} &:= \Phi_f^h \mathbf{y}_2, & \mathbf{y}_3 &:= \Phi_g^h \mathbf{y}_{5/2}, \\ &\vdots & &\vdots \end{aligned}$$

because  $\Phi_f^{h/2} \circ \Phi_f^{h/2} = \Phi_f^h$ . This means that a Strang splitting SSM differs from a Lie-Trotter splitting SSM in the first and the last step only.

### Example 12.5.5 (Convergence of simple splitting methods)

We consider the following IVP whose right hand side function is the sum of two functions for which the ODEs can be solved analytically:

$$\dot{y} = \underbrace{\lambda y(1-y)}_{=: f(y)} + \underbrace{\sqrt{1-y^2}}_{=: g(y)}, \quad y(0) = 0.$$

- $\Phi_f^t y = \frac{1}{1 + (y^{-1} - 1)e^{-\lambda t}}, t > 0, y \in [0, 1]$  (logistic ODE (11.1.6))
- $\Phi_g^t y = \begin{cases} \sin(t + \arcsin(y)) & , \text{if } t + \arcsin(y) < \frac{\pi}{2}, \\ 1 & , \text{else,} \end{cases} t > 0, y \in [0, 1].$

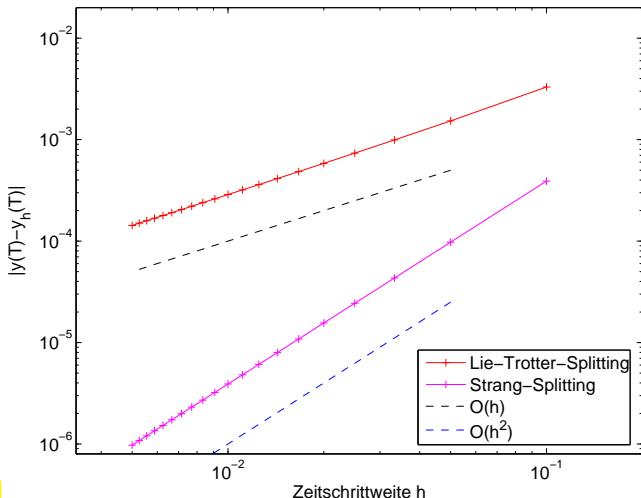


Fig. 428

We observe algebraic convergence of the two splitting methods, order 1 for (12.5.3), oder 2 for (12.5.4).

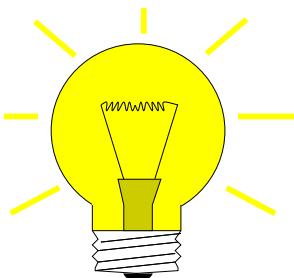
The observation made in Ex. 12.5.5 reflects a general truth:

#### Theorem 12.5.6. Order of simple splitting methods

*Die single step methods defined by (12.5.3) or (12.5.4) are of order (→ Def. 11.3.21) 1 and 2, respectively.*

#### (12.5.7) Inexact splitting methods

Of course, the assumption that  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  and  $\dot{\mathbf{y}} = \mathbf{g}(\mathbf{y})$  can be solved exactly will hardly ever be met. However, it should be clear that a “sufficiently accurate” approximation of the evolution maps  $\Phi_g^h$  and  $\Phi_f^h$  is all we need

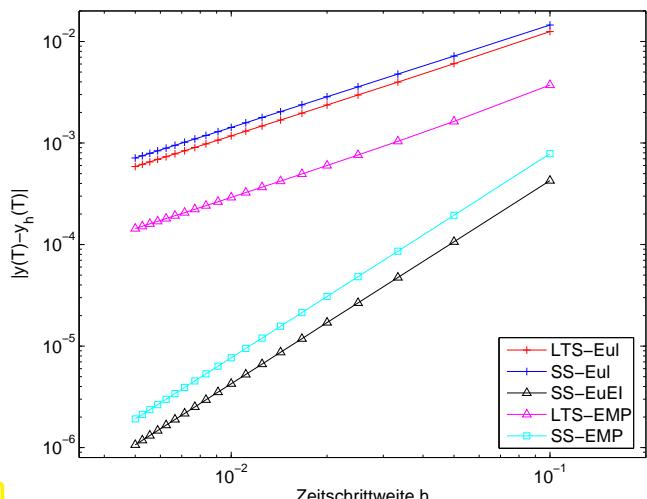


Idea: In (12.5.3)/(12.5.4) replace

exact evolutions  $\Phi_g^h, \Phi_f^h \longrightarrow$  discrete evolutions  $\Psi_g^h, \Psi_f^h$

#### Example 12.5.8 (Convergence of inexact simple splitting methods)

Again we consider the IVP of Ex. 12.5.5 and inexact splitting methods based on different single step methods for the two ODE corresponding to the summands.



- The order of splitting methods may be (but need not be) limited by the order of the SSMs used for  $\Phi_f^h$ ,  $\Phi_g^h$ .

### (12.5.9) Application of splitting methods

In the following situation the use splitting methods seems advisable:

#### “Splittable” ODEs

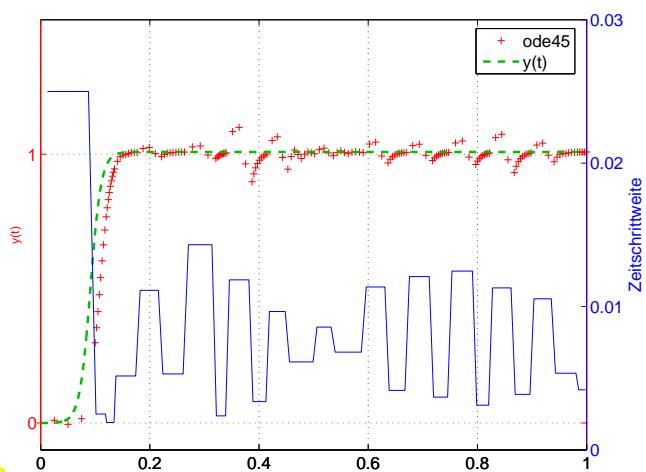
$\dot{\mathbf{y}} = f(\mathbf{y}) + g(\mathbf{y})$  “difficult” :  $\dot{\mathbf{y}} = f(\mathbf{y}) \rightarrow$  stiff, but with an analytic solution  
 (e.g., stiff  $\rightarrow$  Section 12.2)       $\dot{\mathbf{y}} = g(\mathbf{y})$  “easy”, amenable to explicit integration.

### Experiment 12.5.11 (Splitting off stiff components)

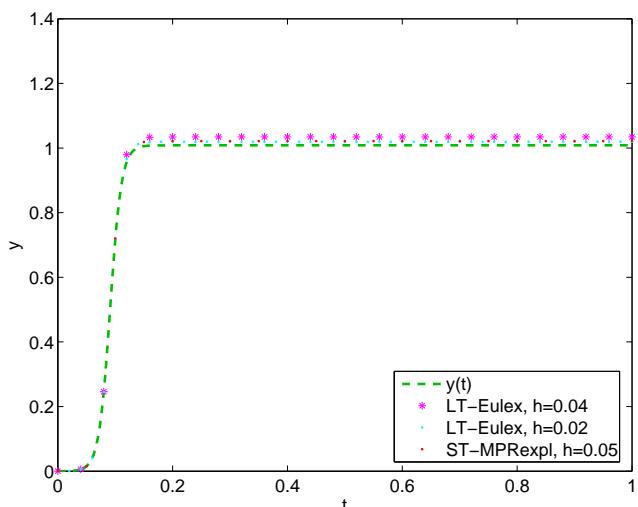
Recall Ex. 12.0.1 and the IVP studied there:

$$\text{AWP} \quad \dot{y} = \lambda y(1-y) + \alpha \sin(y), \quad \lambda = 100, \quad \alpha = 1, \quad y(0) = 10^{-4}.$$

small perturbation



Solution from `ode45`, see Ex. 12.0.1



inexacte splitting method: solution  $(y_k)$

	ode45:	152
Total number of timesteps	LT-Eulex, $h = 0.04$ :	25
	LT-Eulex, $h = 0.02$ :	50
	ST-MPExpl, $h = 0.05$ :	20

Details of the methods:

- LT-Eulex:  $\dot{y} = \lambda y(1 - y) \rightarrow$  exact evolution,  $\dot{y} = \alpha \sin y \rightarrow$  expl. Euler (11.2.7) & Lie-Trotter splitting (12.5.3)
- ST-MPExpl:  $\dot{y} = \lambda y(1 - y) \rightarrow$  exacte evolution,  $\dot{y} = \alpha \sin y \rightarrow$  expl. midpoint rule (11.4.7) & Strang splitting (12.5.4)

We observe that this splitting scheme can cope well with the stiffness of the problem, because the stiff term on the right hand side is integrated exactly.

### Example 12.5.12 (Splitting linear and local terms)

In the numerical treatment of partial differential equation one commonly encounters ODEs of the form

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := -\mathbf{A}\mathbf{y} + \begin{bmatrix} g(y_1) \\ \vdots \\ g(y_d) \end{bmatrix}, \quad \mathbf{A} = \mathbf{A}^\top \in \mathbb{R}^{d,d} \quad \text{positive definite} (\rightarrow \text{Def. 1.1.8}), \quad (12.5.13)$$

with state space  $D = \mathbb{R}^d$ , where  $\lambda_{\min}(\mathbf{A}) \approx 1$ ,  $\lambda_{\max}(\mathbf{A}) \approx d^2$ , and the derivative of  $g : \mathbb{R} \rightarrow \mathbb{R}$  is bounded. Then IVPs for (12.5.13) will be stiff, since the Jacobian

$$D\mathbf{f}(\mathbf{y}) = -\mathbf{A} + \begin{bmatrix} g'(y_1) & & \\ & \ddots & \\ & & g'(y_d) \end{bmatrix} \in \mathbb{R}^{d,d}$$

will have eigenvalues “close to zero” and others that are large (in modulus) and negative. Hence,  $D\mathbf{f}(\mathbf{y})$  will satisfy the criteria (12.2.15) and (12.2.16) for any state  $\mathbf{y} \in \mathbb{R}^d$ .

The natural splitting is

$$\mathbf{f}(\mathbf{y}) = \mathbf{g}(\mathbf{y}) + \mathbf{q}(\mathbf{y}) \quad \text{with} \quad \mathbf{g}(\mathbf{y}) := -\mathbf{A}\mathbf{y}, \quad \mathbf{q}(\mathbf{y}) := \begin{bmatrix} g(y_1) \\ \vdots \\ g(y_d) \end{bmatrix}.$$

- For the **linear ODE**  $\dot{\mathbf{y}} = \mathbf{g}(\mathbf{y})$  we have to use and L-stable ( $\rightarrow$  Def. 12.3.38) single step method, for instance a second-order *implicit* Runge-Kutta method. Its increments can be obtained by solving a *linear system of equations*, whose coefficient matrix will be the same for every step, if uniform timesteps are used.
- The ODE  $\dot{\mathbf{y}} = \mathbf{q}(\mathbf{y})$  boils down to **decoupled** scalar ODEs  $\dot{y}_j = g(y_j)$ ,  $j = 1, \dots, d$ . For them we can use an inexpensive *explicit* RK-SSM like the explicit trapezoidal method (11.4.6). According to our assumptions on  $g$  these ODEs are not haunted by stiffness.

## Summary and Learning Outcomes

# **Chapter 13**

## **Structure Preserving Integration [38]**

# Bibliography

- [1] A. Aggarwal and J.S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
- [2] H. Akima. A new method of interpolation and smooth curve fitting based on local procedures. *J. ACM*, 17(4):589–602, 1970.
- [3] A. Alexanderian. A basic note on iterative matrix inversion. Onlie document, 2012. <http://users.ices.utexas.edu/~alen/articles/mat-inv-rep.pdf>.
- [4] Charles J Alpert, Andrew B Kahng, and So-Zen Yao. Spectral partitioning with multiple eigenvectors. *Discrete Applied Mathematics*, 90(1-3):3 – 26, 1999.
- [5] H. Amann. *Gewöhnliche Differentialgleichungen*. Walter de Gruyter, Berlin, 1st edition, 1983.
- [6] Uri M. Ascher and Chen Greif. *A first course in numerical methods*, volume 7 of *Computational Science & Engineering*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2011.
- [7] R. Ashino, M. Nagase, and R. Vaillancourt. Behind and beyond the MATLAB ODE suite. *Comput. Math. Appl.*, 40(4-5):491–512, 2000.
- [8] Z.-J. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the Solution of Algebraic Eigenvalue Problems*. SIAM, Philadelphia, PA, 2000.
- [9] S. Börm, L. Grasedyck, and W. Hackbusch. Introduction to hierarchical matrices with applications. *Engineering Analysis with Boundary Elements*, 27:405–422, 2003.
- [10] A. Brandt and A.A. Lubrecht. Multilevel matrix multiplication and fast solution of integral equations. *J. Comp. Phys.*, 90(2):348–370, 1990.
- [11] E.O. Brigham. *The Fast Fourier Transform and Its Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [12] Q. Chen and I. Babuska. Approximate optimal points for polynomial interpolation of real functions in an interval and in a triangle. *Comp. Meth. Appl. Mech. Engr.*, 128:405–417, 1995.
- [13] D. Coppersmith and T.J. Rivlin. The growth of polynomials bounded at equally spaced points. *SIAM J. Math. Anal.*, 23(4):970–983, 1992.
- [14] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progression. *J. Symbolic Computing*, 9(3):251–280, 1990.
- [15] W. Dahmen and A. Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. Springer, Heidelberg, 2008.
- [16] P.J. Davis. *Interpolation and Approximation*. Dover, New York, 1975.
- [17] M.A.B. Deakin. Applied catastrophe theory in the social and biological sciences. *Bulletin of Mathematical Biology*, 42(5):647–679, 1980.

- [18] P. Deuflhard and F. Bornemann. *Scientific Computing with Ordinary Differential Equations*, volume 42 of *Texts in Applied Mathematics*. Springer, New York, 2 edition, 2002.
- [19] P. Deuflhard and A. Hohmann. *Numerical Analysis in Modern Scientific Computing*, volume 43 of *Texts in Applied Mathematics*. Springer, 2003.
- [20] Peter Deuflhard. *Newton methods for nonlinear problems*, volume 35 of *Springer Series in Computational Mathematics*. Springer, Heidelberg, 2011. Affine invariance and adaptive algorithms, First softcover printing of the 2006 corrected printing.
- [21] P. Duhamel and M. Vetterli. Fast fourier transforms: a tutorial review and a state of the art. *Signal Processing*, 19:259–299, 1990.
- [22] A. Dutt and V. Rokhlin. Fast Fourier transforms for non-equispaced data II. *Appl. Comput. Harmon. Anal.*, 2:85–100, 1995.
- [23] F.N. Fritsch and R.E. Carlson. Monotone piecewise cubic interpolation. *SIAM J. Numer. Anal.*, 17(2):238–246, 1980.
- [24] M. Gander, W. Gander, G. Golub, and D. Gruntz. *Scientific Computing: An introduction using MATLAB*. Springer, 2005. In Vorbereitung.
- [25] J.R. Gilbert, C.Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.
- [26] Andreas Glaser, Xiangtao Liu, and Vladimir Rokhlin. A fast algorithm for the calculation of the roots of special functions. *SIAM J. Sci. Comput.*, 29(4):1420–1438, 2007.
- [27] David F. Gleich. Pagerank beyond the web. *SIAM Review*, 57(3):321–363, 2015.
- [28] G.H. Golub and C.F. Van Loan. *Matrix computations*. John Hopkins University Press, Baltimore, London, 2nd edition, 1989.
- [29] Craig Gotsman and Sivan Toledo. On the computation of null spaces of sparse rectangular matrices. *SIAM J. Matrix Anal. Appl.*, 30(2):445–463, 2008.
- [30] C.R. Gray. An analysis of the Belousov-Zhabotinski reaction. *Rose-Hulman Undergraduate Math Journal*, 3(1), 2002. <http://www.rose-hulman.edu/mathjournal/archives/2002/vol3-n1/paper1/v3n1-1pd.pdf>.
- [31] L. Greengard and V. Rokhlin. A new version of the fast multipole method for the Laplace equation in three dimensions. *Acta Numerica*, pages 229–269, 1997.
- [32] Gero Greiner and Riko Jacob. The I/O Complexity of Sparse Matrix Dense Matrix Multiplication. In LopezOrtiz, A, editor, *LATIN 2010: THEORETICAL INFORMATICS*, volume 6034 of *Lecture Notes in Computer Science*, pages 143–156. Microsoft Res; Yahoo Res; Univ Waterloo, 2010. 9th Latin American Symposium on Theoretical Informatics (LATIN 2010), Benito Juarez Univ Oaxaca, Oaxaca City, MEXICO, APR 19-23, 2010.
- [33] M. Gutknecht. Linear algebra. Lecture Notes for Course “Lineare Algebra” for Computer Science, 2007.
- [34] M.H. Gutknecht. Lineare Algebra. Lecture notes, SAM, ETH Zürich, 2009. <http://www.sam.math.ethz.ch/~mhg/unt/LA/HS07/>.
- [35] W. Hackbusch. *Iterative Lösung großer linearer Gleichungssysteme*. B.G. Teubner–Verlag, Stuttgart, 1991.
- [36] W. Hackbusch and S. Börm. Data-sparse approximation by adaptive  $\mathcal{H}^2$ -matrices. *Computing*, 69(1):1–35, 2002.

- [37] Wolfgang Hackbusch. *Iterative solution of large sparse systems of equations*, volume 95 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 1994. Translated and revised from the 1991 German original.
- [38] E. Hairer, C. Lubich, and G. Wanner. *Geometric numerical integration*, volume 31 of *Springer Series in Computational Mathematics*. Springer, Heidelberg, 2 edition, 2006.
- [39] E. Hairer, S.P. Norsett, and G. Wanner. *Solving Ordinary Differential Equations I. Nonstiff Problems*. Springer-Verlag, Berlin, Heidelberg, New York, 2 edition, 1993.
- [40] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems*, volume 14 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 2011.
- [41] C.A. Hall and W.W. Meyer. Optimal error bounds for cubic spline interpolation. *J. Approx. Theory*, 16:105–122, 1976.
- [42] M. Hanke-Bourgeois. *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens*. Mathematische Leitfäden. B.G. Teubner, Stuttgart, 2002.
- [43] N.J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, 2 edition, 2002.
- [44] I.C.F. Ipsen and C.D. Meyer. The idea behind Krylov methods. Technical Report 97-3, Math. Dep., North Carolina State University, Raleigh, NC, January 1997.
- [45] S.G. Johnson. Notes on the convergence of trapezoidal-rule quadrature. MIT online course notes, <http://math.mit.edu/~stevenj/trapezoidal.pdf>, 2008.
- [46] N.M. Josuttis. *The C++ Standard Library*. Addison-Wesley, Boston, MA, 2012. <http://filepi.com/i/eFjsP3Z>.
- [47] D. Kalman. A singularly valuable decomposition: The SVD of a matrix. *The College Mathematics Journal*, 27:2–23, 1996.
- [48] M. Kowarschik and C. Weiss. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 213–232. Springer, Heidelberg, 2003.
- [49] Boris I. Kvasov. Monotone and convex interpolation by weighted quadratic splines. *Adv. Comput. Math.*, 40(1):91–116, 2014.
- [50] A.R. Laliena and F.-J. Sayas. Theoretical aspects of the application of convolution quadrature to scattering of acoustic waves. *Numer. Math.*, 112(4):637–678, 2009.
- [51] A.N. Lengville and C.D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, Princeton, NJ, 2006.
- [52] S. Lippman, J. Lajoie, and B. Moo. *C++ Primer*. Addison-Wesley, Boston, 5th edition, 2012.
- [53] D. McAllister and J. Roulier. An algorithm for computing a shape-preserving osculatory quadratic spline. *ACM Trans. Math. Software*, 7(3):331–347, 1981.
- [54] Matthias Messner, Martin Schanz, and Eric Darve. Fast directional multilevel summation for oscillatory kernels based on Chebyshev interpolation. *J. Comput. Phys.*, 231(4):1175–1196, 2012.
- [55] A. Mohsen. Correcting the function derivative estimation using lagrangian interpolation. *ZAMP*, 2015. Submitted.
- [56] C. Moler. *Numerical Computing with MATLAB*. SIAM, Philadelphia, PA, 2004.

- [57] K. Neymeyr. A geometric theory for preconditioned inverse iteration applied to a subspace. Technical Report 130, SFB 382, Universität Tübingen, Tübingen, Germany, November 1999. Submitted to *Math. Comp.*
- [58] K. Neymeyr. A geometric theory for preconditioned inverse iteration: III. Sharp convergence estimates. Technical Report 130, SFB 382, Universität Tübingen, Tübingen, Germany, November 1999.
- [59] K. Nipp and D. Stoffer. *Lineare Algebra*. vdf Hochschulverlag, Zürich, 5 edition, 2002.
- [60] M.L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM, Philadelphia, PA, 2001.
- [61] Victor Pan and Robert Schreiber. An improved newton iteration for the generalized inverse of a matrix, with applications. *SIAM Journal on Scientific and Statistical Computing*, 12(5):1109–1130, 1991.
- [62] A.L. Dontchev and H.-D. Qi, L.-Q. Qi, and H.-X. Yin. Convergence of Newton's method for convex best interpolation. *Numer. Math.*, 87(3):435–456, 2001.
- [63] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*, volume 37 of *Texts in Applied Mathematics*. Springer, New York, 2000.
- [64] C.M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56:1107–1108, 1968.
- [65] R. Rannacher. Einführung in die numerische mathematik. Vorlesungsskriptum Universität Heidelberg, 2000. <http://gaia.iwr.uni-heidelberg.de/>.
- [66] R. Remmert. *Funktionentheorie I*. Number 5 in Grundwissen Mathematik. Springer, Berlin, 1984.
- [67] V. Rokhlin. Rapid solution of integral equations of classical potential theory. *J. Comp. Phys.*, 60(2):187–207, 1985.
- [68] Yousef Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003.
- [69] A. Sankar, D.A. Spielman, and S.-H. Teng. Smoothed analysis of the condition numbers and growth factors of matrices. *SIAM J. Matrix Anal. Appl.*, 28(2):446–476, 2006.
- [70] T. Sauer. *Numerical analysis*. Addison Wesley, Boston, 2006.
- [71] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with pardiso. *J. Future Generation Computer Systems*, 20(3):475–487, 2004.
- [72] Lawrence F. Shampine and Mark W. Reichelt. The MATLAB ODE suite. *SIAM J. Sci. Comput.*, 18(1):1–22, 1997. Dedicated to C. William Gear on the occasion of his 60th birthday.
- [73] J.-B. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.
- [74] D.A. Spielman and Shang-Hua Teng. Spectral partitioning works: planar graphs and finite element meshes. In *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*, pages 96 –105, oct 1996.
- [75] M. Stewart. A superfast toeplitz solver with improved numerical stability. *SIAM J. Matrix Analysis Appl.*, 25(3):669–693, 2003.
- [76] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.
- [77] M. Struwe. Analysis für Informatiker. Lecture notes, ETH Zürich, 2009. <https://moodle-app1.net.ethz.ch/lms/mod/resource/index.php?id=145>.

- [78] F. Tisseur and K. Meerbergen. The quadratic eigenvalue problem. *SIAM Review*, 43(2):235–286, 2001.
- [79] Lloyd N. Trefethen. Is Gauss quadrature better than Clenshaw-Curtis? *SIAM Rev.*, 50(1):67–87, 2008.
- [80] Lloyd N. Trefethen and J. A. C. Weideman. The exponentially convergent trapezoidal rule, XX.
- [81] L.N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.
- [82] N. Trefethen. Six myths of polynomial interpolation and quadrature. Slides, 2014.
- [83] P. Vertesi. On the optimal lebesgue constants for polynomial interpolation. *Acta Math. Hungaria*, 47(1-2):165–178, 1986.
- [84] P. Vertesi. Optimal lebesgue constant for lagrange interpolation. *SIAM J. Numer. Anal.*, 27(5):1322–1331, 1990.
- [85] J. Werner. *Numerische Mathematik I. Lineare und nichtlineare Gleichungssysteme, Interpolation, numerische Integration*. vieweg studium. Aufbaukurs Mathematik. Vieweg, Braunschweig, 1992.

# Index

- LU*-decomposition
  - existence, 128
- $L^2$ -inner product, 371
- $h$ -convergence, 350
- $p$ -convergence, 352
- (Asymptotic) complexity, 69
- (Size of) best approximaton error, 301
- Fill-in, 166
- Preconditioner, 521
- BLAS
  - `axpy`, 65
- Kronecker product, 74
- MATLAB: `cumsum`, 73, 74
- MATLAB: `reshape`, 48
- MATLAB: `triu`, 73
- PYTHON: `reshape`, 48
- MATLAB-code: , 25, 28, 402, 417, 427, 436, 437, 451, 453, 464, 470, 479, 495, 635
- MATLAB-code:  $h$ -adaptive numerical quadrature, 391
- MATLAB-code: (Generalized) distance fitting of a hyperplane: solution of (6.2.18), 410
- MATLAB-code: 1st stage of segmentation of grayscale image, 447
- MATLAB-code: Accessing entries of a sparse matrix: potentially inefficient, 162
- MATLAB-code: Aitken-Neville algorithm, 258
- MATLAB-code: Application of `ode45` for limit cycle problem, 663
- MATLAB-code: Arnoldi eigenvalue approximation, 477
- MATLAB-code: Arnoldi process, 475
- MATLAB-code: Binary arithmetic operators (two arguments), 26
- MATLAB-code: Bisection method for solving  $F(x) = 0$  on  $[a, b]$ , 204
- MATLAB-code: C++ function, 246
- MATLAB-code: CG for Poisson matrix, 519
- MATLAB-code: Calling `newton` with EIGEN data types, 220
- MATLAB-code: Calling a function with multiple return values, 18
- MATLAB-code: Clenshaw algorithm for evalution of Chebychev expansion (4.1.92), 324
- MATLAB-code: Clustering of point set, 492
- MATLAB-code: Code excerpts from MATLAB's integrator `ode45`, 632
- MATLAB-code: Comparison operators, 24
- MATLAB-code: Computation of coefficients of trigonometric interpolation polynomial, general nodes, 343
- MATLAB-code: Computation of nodal potential for circuit of Code 7.0.3, 420
- MATLAB-code: Computation of weights in 3-term recursion for discrete orhtogonal polynomials, 333
- MATLAB-code: Computing an ONB of the kernel of a matrix, 486
- MATLAB-code: Computing resonant frequencies and modes of elastic truss, 457
- MATLAB-code: Computing the interpolation error for Runge's example, 306
- MATLAB-code: Construction for Admissible Partitioning, 591
- MATLAB-code: Construction of Cluster Trees, 588, 589
- MATLAB-code: DFT based low pass frequency filtering of sound, 550
- MATLAB-code: DFT-based approximate computation of Fourier coefficients, 387
- MATLAB-code: Data for "bridge truss", 455
- MATLAB-code: Definition of a class for "update friendly" polynomial interpolant, 266
- MATLAB-code: Definition of a simple vector class **MyVector**, 19
- MATLAB-code: Definition of class for Chebychev interpolation, 323
- MATLAB-code: Demonstration code for access to matrix blocks in EIGEN, 44
- MATLAB-code: Demonstration of use of lambda function, 17
- MATLAB-code: Demonstration on how reshape a matrix in EIGEN, 48
- MATLAB-code: Dense Gaussian elimination applied to arrow system, 142
- MATLAB-code: Difference quotient approximation

- of the derivative of  $\exp$ , 89
- MATLAB-code: Direct solution of dense linear system in EIGEN, 139
- MATLAB-code: Direct solver applied to a (nearly) upper triangular matrix, 138
- MATLAB-code: Discriminant formula for the real roots of  $p(\xi) = \xi^2 + \alpha\xi + \beta$ , 86
- MATLAB-code: Divided differences evaluation by modified Horner scheme, 265
- MATLAB-code: Divided differences, recursive implementation, in situ computation, 265
- MATLAB-code: Driver code for Gram-Schmidt orthonormalization, 30
- MATLAB-code: Effect of pivoting, 128
- MATLAB-code: Efficient computation of Chebychev expansion coefficient of Chebychev interpolant, 326
- MATLAB-code: Efficient computation of coefficient of trigonometric interpolation polynomial (*equidistant nodes*), 343
- MATLAB-code: Efficient evaluation of Chebychev polynomials up to a certain degree, 315
- MATLAB-code: Efficient implementation of inverse power method in EIGEN, 140
- MATLAB-code: Efficient implementation of simplified Newton method, 227
- MATLAB-code: Efficient multiplication of Kronecker product with vector in MATLAB, 75
- MATLAB-code: Efficient multiplication of Kronecker product with vector in PYTHON, 75
- MATLAB-code: Efficient multiplication with the upper diagonal part of a rank- $p$ -matrix, 74
- MATLAB-code: Equidistant composite Simpson rule (5.4.5), 381
- MATLAB-code: Equidistant composite trapezoidal rule (5.4.4), 381
- MATLAB-code: Euclidean inner product, 27
- MATLAB-code: Euclidean norm, 27
- MATLAB-code: Evaluation of difference quotients with variable precision, 90
- MATLAB-code: Evaluation of the interpolation polynomials with barycentric formula, 256
- MATLAB-code: Evaluation of trigonometric interpolation polynomial in many points, 342
- MATLAB-code: Extracting an entry of a sparse matrix, 154
- MATLAB-code: Fast evaluation of trigonometric polynomial at *equidistant* points, 344
- MATLAB-code: Finding out `EPS` in MATLAB, 85
- MATLAB-code: Fitting and interpolating polynomial, 396
- MATLAB-code: Function with multiple return values, 18
- MATLAB-code: GE for “Wilkinson system”, 132
- MATLAB-code: Gaussian elimination with multiple r.h.s., 116
- MATLAB-code: Gaussian elimination with pivoting: extension of Code 1.6.23, 125
- MATLAB-code: General subspace power iteration step with `qr` based orthonormalization, 464
- MATLAB-code: Generation of synthetic perturbed *U-I* characteristics, 488
- MATLAB-code: Generic Newton iteration with termination criterion (2.4.42), 231
- MATLAB-code: Generic damped Newton method based on natural monotonicity test, 235
- MATLAB-code: Golub-Welsch algorithm, 376
- MATLAB-code: Gram-Schmidt orthogonalisation in MATLAB, 76
- MATLAB-code: Gram-Schmidt orthogonalisation in PYTHON, 76
- MATLAB-code: Hermite approximation and orders of convergence, 355
- MATLAB-code: Hermite approximation and orders of convergence with exact slopes, 353
- MATLAB-code: Horner scheme (polynomial in MATLAB format, see Rem. 3.2.4), 251
- MATLAB-code: Image compression, 501
- MATLAB-code: Implementation of class **PolyEval**, 266
- MATLAB-code: In place arithmetic operations (one argumnt), 26
- MATLAB-code: Initialisation of sample sparse matrix in eigen, 164
- MATLAB-code: Initialization of a **MyVector** object from an STL vector, 21
- MATLAB-code: Initialization of a set of vectors through a functor with two arguments, 30
- MATLAB-code: Initialization of sparse matrices: driver script, 152
- MATLAB-code: Initialization of sparse matrices: entry-wise (I), 151
- MATLAB-code: Initialization of sparse matrices: triplet based (II), 151
- MATLAB-code: Initialization of sparse matrices: triplet based (III), 152
- MATLAB-code: Initializing and drawing a simple planar triangulation, 156
- MATLAB-code: Instability of multiplication with inverse, 136
- MATLAB-code: Interpolant class, 249

- MATLAB-code: Investigating convergence of direct power method, 437
- MATLAB-code: Invocation of copy and move constructors, 22
- MATLAB-code: Invoking sparse elimination solver for arrow matrix, 163
- MATLAB-code: LSE for Ex. 8.3.11, 524
- MATLAB-code: LU-factorization of arrow matrix, 168
- MATLAB-code: LU-factorization of sparse matrix, 166
- MATLAB-code: Lagrange polynomial interpolation and evaluation, 260
- MATLAB-code: Lanczos process, cf. Code 8.2.18, 472
- MATLAB-code: Lloyd-Max algorithm for cluster identification, 491
- MATLAB-code: Local evaluation of cubic Hermite polynomial, 276
- MATLAB-code: MATLAB code for *approximate* computation of Lebesgue constants, 270
- MATLAB-code: MATLAB template implementing generic quadrature formula, 361
- MATLAB-code: Matrix  $\times$  vector product  $\mathbf{y} = \mathbf{Ax}$  in triplet format, 148
- MATLAB-code: Measuring runtimes of Code 1.6.23 vs. MATLAB \operator, 114
- MATLAB-code: Monotonicity preserving slopes in pchip, 281
- MATLAB-code: Newton iteration for (2.4.35), 228
- MATLAB-code: Newton method in the scalar case  $n = 1$ , 206
- MATLAB-code: Newton's method in C++, 219
- MATLAB-code: Non-member function **output operator**, 28
- MATLAB-code: Non-member function for left multiplication with a scalar, 27
- MATLAB-code: Numeric differentiation through difference quotients, 261
- MATLAB-code: Numerical differentiation by extrapolation to zero, 262
- MATLAB-code: PCA for measured ***U-I*** characteristics, 489
- MATLAB-code: PCA in three dimensions via SVD, 487
- MATLAB-code: PCA of stock prices in MATLAB, 496
- MATLAB-code: Parent Code, 590
- MATLAB-code: Permuting arrow matrix, see Figs. 56, 57, 169
- MATLAB-code: Perturbations cure instability of Gaussian elimination, 132
- MATLAB-code: Piecewise cubic Hermite interpolation, 277
- MATLAB-code: Plotting Chebychev polynomials, see Fig. 126, 127, 315
- MATLAB-code: Polynomial Interpolation, 255
- MATLAB-code: Polynomial evaluation, 263
- MATLAB-code: Polynomial evaluations, 258
- MATLAB-code: Preprocessing Code, 591
- MATLAB-code: Principal axis point set separation, 491
- MATLAB-code: QR-algorithm with shift, 426
- MATLAB-code: QR-based solver for full rank linear least squares problem (6.0.14), 406
- MATLAB-code: Quadratic spline: selection of  $p_i$ , 290
- MATLAB-code: Rayleigh quotient iteration (for normal  $\mathbf{A} \in \mathbb{R}^{n,n}$ ), 450
- MATLAB-code: Recursive evaluation of Chebychev expansion (4.1.92), 323
- MATLAB-code: Refined local stepsize control for single step methods, 642
- MATLAB-code: Remez algorithm for uniform polynomial approximation on an interval, 337
- MATLAB-code: Ritz projections onto Krylov space (7.4.2), 470
- MATLAB-code: Runtime comparison, 343
- MATLAB-code: Runtime measurement of Code 1.6.100 vs. Code 1.6.101, 143
- MATLAB-code: SVD based image compression, 500
- MATLAB-code: Simple Cholesky factorization, 181
- MATLAB-code: Simple local stepsize control for single step methods, 637
- MATLAB-code: Simulation of "stiff" chemical reaction, 662
- MATLAB-code: Single index access of matrix entries in EIGEN, 47
- MATLAB-code: Smart approach, 140
- MATLAB-code: Solving LSE  $\mathbf{Ax} = \mathbf{b}$  with Gaussian elimination, 113
- MATLAB-code: Solving LSQ problem via SVD, 408
- MATLAB-code: Solving a sparse linear system of equations in EIGEN, 165
- MATLAB-code: Solving an arrow system according to (1.6.99), 143
- MATLAB-code: Spline approximation error, 357
- MATLAB-code: Spline approximation: driver script, 358
- MATLAB-code: Square root iteration  $\rightarrow$  Ex. 2.1.20, 193

MATLAB-code: Stable computation of real root of a quadratic polynomial, 93  
MATLAB-code: Stable recursion for area of regular  $n$ -gon, 96  
MATLAB-code: Step by step shape preserving spline interpolation, 293  
MATLAB-code: Storage order in PYTHON, 46  
MATLAB-code: Subspace power iteration with Ritz projection, 468  
MATLAB-code: Summation of exponential series, 97  
MATLAB-code: Templated **constructors** copying vector entries from an STL container, 21  
MATLAB-code: Tentative computation of circumference of regular polygon, 95  
MATLAB-code: Testing the accuracy of computed roots of a quadratic polynomial, 87  
MATLAB-code: Timing different implementations of matrix multiplication in MATLAB, 62  
MATLAB-code: Timing different implementations of matrix multiplication in PYTHON, 63  
MATLAB-code: Timing for row and column oriented matrix access for EIGEN, 51  
MATLAB-code: Timing for row and column oriented matrix access in MATLAB, 49  
MATLAB-code: Timing for row and column oriented matrix access in PYTHON, 50  
MATLAB-code: Timing multiplication with scaling matrix in MATLAB, 59  
MATLAB-code: Timing multiplication with scaling matrix in PYTHON, 59  
MATLAB-code: Timing polynomial evaluations, 259  
MATLAB-code: Total least squares via SVD, 411  
MATLAB-code: Transformation of a vector through a functor `double` → `double`, 25  
MATLAB-code: Use of MATLABintegrator `ode45` for a stiff problem, 647  
MATLAB-code: Using **Array** in EIGEN, 45  
MATLAB-code: Visualizing the structure of matrices in MATLAB, 57  
MATLAB-code: Visualizing the structure of matrices in PYTHON, 57  
MATLAB-code: Wasteful approach, 140  
MATLAB-code: Wrong result from Gram-Schmidt orthogonalisation, 77  
MATLAB-code: [Wrap-around implementation] MATLAB-CODE sine transform, 570  
MATLAB-code: [ MATLAB-CODE FFT-based solution of local]translation invariant linear operators, 572  
MATLAB-code: **Gram-Schmidt orthonormalization** (do not use, unstable algorithm ), 463  
MATLAB-code: BLAS-based SAXPY operation in C++, 67  
MATLAB-code: **Constructor** for constant vector, also default constructor, see Line 28, 20  
MATLAB-code: **Constructor** initializing vector from STL iterator range, 21  
MATLAB-code: **Copy assignment** operator, 23  
MATLAB-code: **Copy constructor**, 22  
MATLAB-code: **Destructor**: releases allocated memory, 24  
MATLAB-code: **Move assignment** operator, 23  
MATLAB-code: **Move constructor**, 22

MATLAB-code: **Type conversion** operator: copies contents of vector into STL vector, 24  
MATLAB-code: **Equispaced points**: fast on the fly evaluation of trigonometric interpolation polynomial, 345  
MATLAB-code: MATLAB code for Ex. 1.4.10, 72  
MATLAB-code: MATLAB polynomial evaluation using built-in function `polyfit`, 260  
MATLAB-code: MATLAB-CODE Arnoldi eigenvalue approximation, 477  
MATLAB-code: assembly of **A**, **D**, 442  
MATLAB-code: basic CG iteration for solving  $\mathbf{Ax} = \mathbf{b}$ , § 8.2.17, 515  
MATLAB-code: computing Legendre polynomials, 376  
MATLAB-code: computing page rank vector **r** via `eig`, 434  
MATLAB-code: computing row bandwidths, → Def. 1.7.54, 173  
MATLAB-code: condition numbers of  $2 \times 2$  matrices, 110  
MATLAB-code: envelope aware forward substitution, 173  
MATLAB-code: envelope aware recursive LU-factorization, 174  
MATLAB-code: fill-in due to pivoting, 170  
MATLAB-code: gradient method for  $\mathbf{Ax} = \mathbf{b}$ , **A** s.p.d., 506  
MATLAB-code: inverse iteration for computing  $\lambda_{\min}(\mathbf{A})$  and associated eigenvector, 449  
MATLAB-code: loading and displaying an image, 439  
MATLAB-code: listing theoretical bounds for CG convergence rate, 519  
MATLAB-code: measuring runtimes of `eig`, 427  
MATLAB-code: one step of subspace power iteration with Ritz projection, matrix version, 466  
MATLAB-code: one step of subspace power iteration, **m** = 2, 460  
MATLAB-code: power iteration with orthogonal projection for two vectors, 461  
MATLAB-code: preconditioned inverse iteration (7.3.63), 453  
MATLAB-code: preconditioning in MATLAB, 176  
MATLAB-code: recursive LU-factorization with partial pivoting, 126  
MATLAB-code: rvalue and lvalue access operators, 24  
MATLAB-code: secant method, 212  
MATLAB-code: simple fixed point iteration in 1D, 189  
MATLAB-code: simple implementation of PCG algorithm § 8.3.5, 523  
MATLAB-code: simulation of linear RLC circuit using `ode45`, 656  
MATLAB-code: small residuals for GE, 135  
MATLAB-code: solving a rank-1 modified LSE, 146  
MATLAB-code: stochastic page rank simulation, 430  
MATLAB-code: template for Gauss-Newton method, 416  
MATLAB-code: templated function for Gram-Schmidt orthonormalization, 29  
MATLAB-code: timing access to rows/columns of a sparse matrix, 150  
MATLAB-code: tracking fractions of many surfers, 432  
MATLAB-code: transition probability matrix for page rank, 432  
MATLAB-code: MATLAB-CODE Sine transform, 570  
MATLAB-code: MATLAB-CODE cosine transform, 574  
MATLAB-code: MATLAB-CODE naive DFT-implementation, 564  
MATLAB-code: MATLAB-CODE two dimensional sine tr., 572  
`ode45`, 631  
`odeset`, 644  
3-term recursion  
  for Chebychev polynomials, 314  
  for Legendre polynomials, 375  
3-term recursion  
  orthogonal polynomials, 332  
5-points-star-operator, 571  
  
a posteriori  
  adaptive quadrature, 388  
a posteriori error bound, 194  
a posteriori termination, 192  
a priori  
  adaptive quadrature, 388  
  a priori termination, 192  
A-inner product, 503  
A-orthogonal, 512  
A-stability of a Runge-Kutta single step method, 675  
A-stable single step method, 675  
absolute error, 83  
absolute tolerance, 192, 219, 637  
Acceptability Criteria, 587  
adaptive multigrid quadrature, 390  
adaptive quadrature, 387  
  a posteriori, 388  
  a priori, 388  
Adding EPS to 1, 85  
AGM, 191  
Aitken-Neville scheme, 257  
algebra, 61  
algebraic convergence, 305  
algebraic dependence, 70  
alternation theorem, 336  
Analyticity of a complex valued function, 311  
Approximation  
  Low rank, 592  
approximation  
  uniform, 297  
arrow matrix, 167  
Ass: "Axiom" of roundoff analysis, 84  
Ass: Analyticity of interpolant, 312  
Ass: Global solutions, 610  
Ass: Self-adjointness of multiplication operator, 331  
asymptotic complexity, 69  
  sharp bounds, 69  
asymptotic rate of linear convergence, 200  
autonomization, 607  
Autonomous ODE, 603  
AXPY operation, 515  
axy operation, 65  
  
back substitution, 111  
backward error analysis, 103  
backward substitution, 122  
Bandbreite  
  Zeilen-, 171  
banded matrix, 170  
bandwidth, 171  
  lower, 171  
  minimizing, 175  
  upper, 171

barycentric interpolation formula, 256  
 basis  
     cosine, 573  
     orthonormal, 425  
     sine, 569  
     trigonometric, 542  
 Belousov-Zhabotinsky reaction, 633  
 bending energy, 286  
 Bernstein approximant, 300  
 Besetzungsmuster, 176  
 best approximation  
     uniform, 336  
 best approximation error, 301, 329  
 best low rank approximation, 498  
 bicg, 528  
 BiCGStab, 528  
 bisection, 203  
 BLAS, 61  
 block LU-decomposition, 123  
 block matrix multiplication, 61  
 blow-up, 634  
 blurring operator, 554  
 Boundary edge, 157  
 Bounding Box, 587  
 Broyden  
     quasi-Newton method, 238  
 Broyden-Verfahren  
     convergence monitor, 239  
 Butcher scheme, 630, 672

cache miss, 54  
 cache thrashing, 54  
 cancellation, 86, 88  
 capacitance, 105  
 capacitor, 105  
 cardinal  
     spline, 288  
 cardinal basis, 248, 251  
 cardinal basis function, 288  
 cardinal interpolant, 288, 363  
 Cauchy product  
     of power series, 535  
 causal filter, 532  
 CCS format, 149  
 cell  
     of a mesh, 348  
 CG  
     convergence, 520  
     preconditioned, 522  
     termination criterion, 515  
 CG = conjugate gradient method, 510  
 CG algorithm, 514  
 chain rule, 222  
 Characteristic parameters of IEEE floating point numbers, 82  
 characteristic polynomial, 423  
 Chebychev interpolation, 323  
 Chebychev nodes, 316, 317  
 Chebychev polynomials, 314, 518  
     3-term recursion, 314  
 Chebychev-interpolation, 313  
 chemical reaction kinetics, 661  
 Cholesky decomposition  
     costs, 181  
 circuit simulation  
     transient, 605  
 circulant matrix, 537  
 Classical Runge-Kutta method  
     Butcher scheme, 631  
 Clenshaw algorithm, 324  
 Cluster  
     Tree, 587  
 cluster analysis, 490  
 Clustering Approximation  
     Task, 579  
 coil, 105  
 collocation, 668  
 collocation conditions, 668  
 Collocation Matrix, 579, 580  
 collocation points, 668  
 collocation single step methods, 668  
 column major matrix format, 46  
 column sum norm, 101  
 column transformation, 60  
 combinatorial graph Laplacian, 159  
 complexity  
     asymptotic, 69  
     linear, 71  
     of SVD, 485  
 composite quadrature formulas, 380  
 Compressed Column Storage (CCS), 161  
 compressed row storage, 149  
 Compressed Row Storage (CRS), 161  
 computational cost  
     Gaussian elimination, 114  
 computational costs  
     LU-decomposition, 122  
 Computational effort, 68  
 computational effort, 68, 216  
     eigenvalue computation, 427  
 concave  
     data, 271  
     function, 272  
 Condition (number) of a matrix, 108  
 condition number  
     of a matrix, 108

spectral, 510  
 conjugate gradient method, 510  
 consistency  
     of iterative methods, 186  
     fixed point iteration, 195  
 Consistency of fixed point iterations, 195  
 Consistency of iterative methods, 186  
 Consistent single step methods, 619  
 constant  
     Lebesgue, 318  
 constitutive relations, 105, 245  
 constrained least squares, 411  
 Contractive mapping, 198  
 Convergence, 186  
 convergence  
     algebraic, 305  
     asymptotic, 213  
     exponential, 305, 309, 319  
     global, 187  
     iterative method, 186  
     linear, 187  
     linear in Gauss-Newton method, 418  
     local, 187  
     numerical quadrature, 362  
     quadratic, 191  
     rate, 187  
 convergence monitor, 239  
     of Broyden method, 239  
 convex  
     data, 271  
     function, 272  
 Convex/concave data, 271  
 convex/concave function, 272  
 convolution  
     discrete, 532, 535  
     discrete periodic, 536  
     of sequences, 535  
 Corollary: "Optimality" of CG iterates, 517  
 Corollary: Best approximant by orthogonal projection, 329  
 Corollary: Consistent Runge-Kutta single step methods, 630  
 Corollary: Continuous local Lagrange interpolants, 349  
 Corollary: Euclidean matrix norm and eigenvalues, 102  
 Corollary: Invariance of order under affine transformation, 368  
 Corollary: Lagrange interpolation as linear mapping, 253  
 Corollary: ONB representation of best approximant, 330  
 Corollary: Piecewise polynomials Lagrange interpolation operator, 349  
 Corollary: Polynomial stability function of explicit RK-SSM, 653  
 Corollary: Principal axis transformation, 424  
 Corollary: Rational stability function of explicit RK-SSM, 675  
 Corollary: Smoothness of cubic Hermite polynomial interpolant, 275  
 Corollary: Stages limit order of explicit RK-SSM, 653  
 Correct rounding, 83  
 cosine  
     basis, 573  
     transform, 573  
 cosine matrix, 573  
 cosine transform, 573  
 costs  
     Cholesky decomposition, 181  
 Crout's algorithm, 121  
 CRS, 149  
 CRS format  
     diagonal, 149  
 cubic complexity, 69  
 cubic Hermite interpolation, 254, 277  
 Cubic Hermite polynomial interpolant, 275  
 cubic spline interpolation  
     error estimates, 356  
 cyclic permutation, 169

damped Newton method, 232  
 damping factor, 233  
 data fitting, 395  
     linear, 395  
     polynomial, 396  
 data interpolation, 244  
 deblurring, 553  
 deblurring = Entrauschen, 553  
 definite, 100  
 dense matrix, 147  
 derivative  
     in vector spaces, 221  
 Derivative of functions between vector spaces, 221  
 descent methods, 503  
 destructor, 23  
 DFT, 539, 543  
     two-dimensional, 552  
 Diagonal dominance, 178  
 diagonal matrix, 36  
 diagonalization  
     for solving linear ODEs, 655  
     of a matrix, 425  
 diagonalization of local translation invariant linear operators, 571  
 Diagonally dominant matrix, 178  
 diagonally implicit Runge-Kutta method, 672  
 difference quotient, 89  
     backward, 615  
     forward, 614  
     symmetric, 616  
 difference scheme, 614  
 differential, 221  
 direct power method, 436  
 DIRK-SSM, 672  
 discrete  $L^2$ -inner product, 331, 333  
 Discrete convolution, 535

discrete convolution, 532, 535  
 discrete evolution, 617  
 discrete Fourier transform, 539, 543  
 Discrete periodic convolution, 536  
 discrete periodic convolution, 536  
 discretization  
     of a differential equation, 618  
 discretization error, 620  
 discriminant formula, 86, 92  
 divided differences, 264  
 domain of definition, 184  
 dot product, 54  
 double nodes, 254  
 double precision, 82  
  
 economical singular value decomposition, 485  
 efficiency, 216  
 Eigen, 41  
     arrays, 45  
     data types, 42  
     initialisation, 43  
     sparse matrices, 161  
 eigen  
     accessing matrix entries, 44  
 eigenspace, 423  
 eigenvalue, 423  
     generalized, 425  
 eigenvalue problem  
     generalized, 425  
 eigenvalues and eigenvectors, 423  
 eigenvector, 423  
     generalized, 425  
 electric circuit, 105, 183  
     resonant frequencies, 419  
 elementary arithmetic operations, 80, 84  
 elimination matrix, 119  
 embedded Runge-Kutta methods, 643  
 Energy norm, 503  
 envelope  
     matrix, 171  
 Equation  
     non-linear, 184  
 equidistant mesh, 348  
 equidistribution principle  
     for quadrature error, 389  
 equivalence  
     of norms, 100  
 Equivalence of norms, 188  
 ergodicity, 435  
 error  
     absolute, 83  
     relative, 83  
 error estimator  
     a posteriori, 194  
 Euler method  
     explicit, 613  
     implicit, 615  
     implicit, stability function, 674  
     semi implicit, 680  
 Euler polygon, 614  
 Euler's iteration, 211  
 evolution operator, 611  
 Evolution operator/mapping, 611  
 expansion  
     asymptotic, 260  
 explicit Euler method, 613  
     Butcher scheme, 630  
 explicit midpoint rule  
     Butcher scheme, 630  
     for ODEs, 628  
 Explicit Runge-Kutta method, 629  
 explicit Runge-Kutta method, 629  
 explicit trapezoidal rule  
     Butcher scheme, 630  
 exponential convergence, 319  
 extended normal equations, 402  
 extended state space  
     of an ODE, 607  
 extrapolation, 260  
  
 Far field, 585  
 fast Fourier transform, 564  
 FFT, 564  
 fill-in, 166  
 filter  
     high pass, 548  
     low pass, 548  
 Finding out EPS in MATLAB, 85  
 finite filter, 532  
 Fitted polynomial, 334  
 fixed point, 195  
 fixed point form, 195  
 fixed point iteration, 195  
 fixed point iteration  
     consistency, 195  
     Newton's method, 228  
 floating point number, 81  
 floating point numbers, 79, 80  
 forward elimination, 111  
 forward substitution, 122  
 Fourier  
     matrix, 542  
 Fourier coefficient, 561  
 Fourier series, 558  
 Fourier transform, 558  
  
 discrete, 539, 543  
 fractional order of convergence, 213  
 frequency domain, 106  
 frequency filtering, 544  
 Frobenius norm, 498  
 function  
     concave, 272  
     convex, 272  
 function object, 25  
 function representation, 246  
 Funktion  
     shandles, 219  
  
 Gauss collocation single step method, 671  
 Gauss Quadrature, 367  
 Gauss-Legendre quadrature formulas, 374  
 Gauss-Newton method, 415  
 Gauss-Radau quadrature formulas, 678  
 Gauss-Seidel preconditioner, 524  
 Gaussian elimination, 110  
     block version, 117  
     by rank-1 modifications, 116  
     for non-square matrices, 115  
     instability, 132  
 GE for "Wilkinson system", 132  
 Generalized Lagrange polynomials, 254  
 global solution  
     of an IVP, 609  
 GMRES, 528  
 Golub-Welsch algorithm, 376  
 gradient, 222, 505  
 Gradient and Hessian, 222  
 Gram-Schmidt  
     Orthonormalisierung, 475  
 Gram-Schmidt orthogonalisation, 76  
 Gram-Schmidt orthogonalization, 330, 475, 513  
 Gram-Schmidt orthonormalization, 404, 463  
 graph partitioning, 448  
 grid, 348  
 grid cell, 348  
 grid function, 571  
 grid interval, 348  
  
 Halley's iteration, 211  
 harmonic mean, 279  
 hat function, 247  
 heartbeat model, 604  
 Hermite interpolation  
     cubic, 254  
 Hermitian matrix, 36  
 Hermitian/symmetric matrices, 36  
 Hessian, 37  
 Hessian matrix, 222  
 high pass filter, 548  
 Hilbert matrix, 91  
 homogeneous, 100  
 Hooke's law, 455  
 Horner scheme, 250  
  
 I/O-complexity, 68  
 identity matrix, 35  
 IEEE standard 754, 81  
 ill conditioned, 109  
 ill-conditioned problem, 104  
 image segmentation, 439  
 implicit differentiation, 226  
 implicit Euler method, 615  
 implicit function theorem, 616  
 implicit midpoint method, 616  
 impulse response, 532  
     of a filter, 532  
 in place, 121, 122  
 in situ, 117, 122  
 in-situ, 127  
 increment equations  
     linearized, 681  
 increments  
     Runge-Kutta, 629, 672  
 inductance, 105  
 inductor, 105  
 inexact splitting methods, 684  
 inf, 82  
 infinity, 82  
 initial guess, 186, 195  
 initial value problem  
     stiff, 664  
 initial value problem (IVP), 607  
 initial value problem (IVP) = Anfangswertproblem, 607  
 Inner product, 327  
 inner product  
     A-, 503  
 intermediate value theorem, 203  
 interpolant  
     piecewise linear, 273  
 interpolation  
     barycentric formula, 256  
     Chebychev, 313  
     complete cubic spline, 284  
     cubic Hermite, 277  
     Hermite, 254  
     Lagrange, 251  
     natural cubic spline, 284  
     periodic cubic spline, 285  
     spline cubic, 282

- spline cubic, locality, 288
- spline shape preserving, 289
- trigonometric, 339
- interpolation operator, 249
- interpolation problem, 244
- interpolation scheme, 244
- inverse interpolation, 214
- inverse iteration, 449
  - preconditioned, 451
- inverse matrix, 107
- Invertible matrix, 107
- invertible matrix, 107
- iteration, 185
  - Halley's, 211
  - Euler's, 211
    - quadratrical inverse interpolation, 211
- iteration function, 186, 195
- iterative method, 185
  - convergence, 186
- IVP, 607
- Jacobi preconditioner, 524
- Jacobian, 199, 218, 222
- Kernel Function, 579, 580
  - separable, 580
- kinetics
  - of chemical reaction, 661
- Kirchhoff (current) law, 105
- knots
  - spline, 281
- Konvergenz
  - Algebraische, Quadratur, 378
- Kronecker product, 74
- Kronecker symbol, 34
- Krylov space, 511
  - for Ritz projection, 470
- L-stable, 677
- L-stable Runge-Kutta method, 677
- Lagrange interpolation approximation scheme, 304
- Lagrange multiplier, 412
- Lagrangian (interpolation polynomial) approximation scheme, 304
- Lagrangian multiplier, 412
- lambda function, 17, 25
- Landau symbol, 69, 305
- Landau-O, 69
- Lapack, 115
- leading coefficient
  - of polynomial, 250
- Least squares
  - with linear constraint, 411
- least squares
  - total, 410
- least squares problem, 398, 399
  - conditioning, 400
- Lebesgue
  - constant, 318
- Lebesgue constant, 269
- Legendre polynomials, 373
- Lemma:  $r_k \perp U_k$ , 511
- Lemma: Absolute conditioning of polynomial interpolation, 269
- Lemma: Affine pullbacks and polynomials, 302
- Lemma: Bases for Krylov spaces in CG, 513
- Lemma: Cholesky decomposition, 181
- Lemma: Criterion for local Lipschitz continuity, 609
- Lemma: Cubic convergence of modified Newton methods, 211
- Lemma: Diagonal dominance and definiteness, 180
- Lemma: Diagonalization of circulant matrices, 543
- Lemma: Equivalence of Gaussian elimination and LU-factorization, 130
- Lemma: Error of the polynomial interpolation, 308
- Lemma: Existence & uniqueness of solutions of the least squares problem, 399
- Lemma: Existence of LU-decomposition, 120
- Lemma: Existence of LU-factorization with pivoting, 128
- Lemma: Formula for Euclidean norm of a Hermitian matrix, 102
- Lemma: Gershgorin circle theorem, 424
- Lemma: Group of regular diagonal/triangular matrices, 58
- Lemma: Higher order local convergence of fixed point iterations, 201
- Lemma: LU-factorization of diagonally dominant matrices, 178
- Lemma: Ncut and Rayleigh quotient ( $\rightarrow [73, \text{Sect. 2}]$ ), 443
- Lemma: Necessary conditions for s.p.d., 37
- Lemma: Positivity of Gauss-Legendre quadrature weights, 374
- Lemma: Properties of cosine matrix, 573
- Lemma: Properties of Fourier matrix, 543
- Lemma: Properties of the sine matrix, 569
- Lemma: Quadrature error estimates for  $C^1$ -integrands, 378
- Lemma: Quadrature formulas from linear interpolation schemes, 363
- Lemma: Residual formula for quotients, 311
- Lemma: S.p.d. LSE and quadratic minimization problem, 503
- Lemma: Sherman-Morrison-Woodbury formula, 145
- Lemma: Similarity and spectrum  $\rightarrow [34, \text{Thm. 9.7}], [15, \text{Lemma 7.6}], [59, \text{Thm. 7.2}]$ , 424
- Lemma: Smoothness of solutions of ODEs, 602
- Lemma: Stability function as approximation of  $\exp$  for small arguments, 653
- Lemma: Sufficient condition for linear convergence of fixed point iteration, 200
- Lemma: Sufficient condition for local linear convergence of fixed point iteration, 199
- Lemma: SVD and Euclidean matrix norm, 498
- Lemma: SVD and rank of a matrix  $\rightarrow [59, \text{Cor. 9.7}]$ , 485
- Lemma: Taylor expansion of inverse distance function, 456
- Lemma: Theory of Arnoldi process, 476
- Lemma: Transformation of norms under affine pullbacks, 303
- Lemma: Tridiagonal Ritz projection from CG residuals, 472
- Lemma: Uniqueness of orthonormal polynomials, 332
- Lemma: Zeros of Legendre polynomials, 374
- Levinson algorithm, 577
- Lie-Trotter splitting, 683
- limit cycle, 662
- limiter, 279
- line search, 504
- linear complexity, 69, 71
- Linear convergence, 187
- linear correlation, 488, 493
- linear data fitting, 395
- linear electric circuit, 105
- linear filter, 532
- Linear interpolation operator, 249
- linear operator, 249
  - diagonalization, 571
- linear ordinary differential equation, 422
- linear regression, 70
- linear system of equations, 104
  - multiple right hand sides, 116
- Lipschitz continuos function, 608
- Lloyd-Max algorithm, 490
- Local and global convergence, 187
- local Lagrange interpolation, 349
- local linearization, 218
- locality
  - of interpolation, 287
- logistic differential equation, 603
- Lotka-Volterra ODE, 603
- low pass filter, 548
- Low rank approximation, 592
- lower triangular matrix, 36
- LU-decomposition
  - blocked, 123
  - computational costs, 122
  - envelope aware, 173
  - existence, 120
  - in place, 122
- LU-factorization
  - envelope aware, 173
  - of sparse matrices, 165
  - with pivoting, 126
- machine number, 81
  - exponent, 81
- machine numbers, 80, 81
  - distribution, 81
  - extremal, 81
- Machine numbers/floating point numbers, 81
- machine precision, 84
- mantissa, 81
- Markov chain, 430, 575
  - stationary distribution, 431
- mass matrix, 457
- MATLAB \operator, 115
- Matrix
  - adjoint, 35
  - Collocation-, 579
  - Hermitian, 425
  - Hermitian transposed, 35
  - normal, 424
  - skew-Hermitian, 425
  - transposed, 35
  - unitary, 425
- matrix
  - banded, 170, 171
  - condition number, 108
  - dense, 147
  - diagonal, 36
  - envelope, 171
  - Fourier, 542
  - generalized condition number, 400
  - Hermitian, 36
  - Hessian, 222
  - lower triangular, 36
  - normalized, 36
  - positive definite, 36
  - positive semi-definite, 36
  - rank, 107
  - sine, 569
  - sparse, 147
  - storage formats, 46
  - structurally symmetric, 174
  - symmetric, 36
  - tridiagonal, 171
  - upper triangular, 36
- matrix algebra, 61
- matrix block, 35
- Matrix envelope, 171
- matrix exponential, 655
- matrix factorization, 117, 119
- Matrix norm, 101
- matrix norm, 101
  - column sums, 101
  - row sums, 101
- matrix storage
  - envelope oriented, 174
- member function, 14
- mesh, 348
  - equidistant, 348
  - in time, 618
  - temporal, 612
- mesh adaptation, 389
- mesh refinement, 389
- mesh width, 348
- Method
  - Quasi-Newton, 236
- method, 14

midpoint method  
     implicit, stability function, 674  
 midpoint rule, 364, 628  
 Millennium Algorithms, 600  
 Milne rule, 366  
 min-max theorem, 444  
 minimal residual methods, 527  
 model function, 205  
 Modellfunktionsverfahren, 205  
 modified Newton method, 210  
 monomial representation  
     of a polynomial, 250  
 monomials, 250  
 monotonic data, 271  
 multi-point methods, 205, 211  
 multiplicity  
     geometric, 423  
     of an interpolation node, 254

NaN, 82  
 Ncut, 440  
 Near field, 585  
 nested  
     subspaces, 510  
 nested spaces, 300  
 Newton  
     basis, 263  
     damping, 233  
     damping factor, 233  
     monotonicity test, 233  
     simplified method, 227  
 Newton correction, 219  
     simplified, 231  
 Newton iteration, 218  
     numerical Differentiation, 227  
     termination criterion, 230  
 Newton method  
     1D, 205  
     damped, 232  
     local quadratic convergence, 228  
     modified, 210  
 Newton's law of motion, 457  
 nodal analysis, 105, 183  
     transient, 605  
 nodal potentials, 105  
 node  
     double, 254  
     for interpolation, 251  
     in electric circuit, 105  
     multiple, 254  
     multiplicity, 254  
     of a mesh, 348  
     quadrature, 361  
 nodes, 251  
     Chebychev, 317  
     Chebychev nodes, 318  
     for interpolation, 251  
 non-linear data fitting, 414  
 non-normalized numbers, 82  
 Norm, 100  
 norm, 100  
      $L^1$ , 268  
      $L^2$ , 268  
      $\infty$ , 100  
     1-, 100  
     energy-, 503  
     Euclidean, 100  
     Frobenius norm, 498  
     of matrix, 101  
     Sobolev semi-, 310  
     supremum, 268  
 normal equations, 328, 401  
     extended, 402  
     with constraint, 413  
 normalization, 436  
 Normalized cut, 440  
 normalized lower triangular matrix, 119  
 normalized triangular matrix, 36  
 not a number, 82  
 Nullstellenbestimmung  
     Modellfunktionsverfahren, 205  
 Numerical differentiation  
     roundoff, 91  
 numerical Differentiation  
     Newton iteration, 227  
 numerical differentiation, 89  
 numerical quadrature, 359  
 numerical rank, 408

ODE, 607  
     scalar, 613  
 Ohmic resistor, 105  
 one-point methods, 205  
 one-step error, 623  
 order  
     of quadrature formula, 367  
 Order of a quadrature rule, 367  
 Order of a single step method, 623  
 order of convergence, 190  
     fractional, 213  
 ordinary differential equation  
     linear, 422  
 ordinary differential equation (ODE), 607  
 orogenator, 633  
 orthogonal polynomials, 373

orthogonal projection, 329  
 Orthogonality, 327  
 Orthonormal basis, 329  
     orthonormal basis, 329, 425  
     Orthonormal polynomials, 331  
 overflow, 82, 85  
 overloading  
     of functions, 14  
     of operators, 14

page rank, 429  
     stochastic simulation, 430  
 PARDISO, 163  
 partial pivoting, 126, 127  
 pattern  
     of a matrix, 57  
 PCA, 481  
 PCG, 522  
 Peano  
     Theorem of, 609  
 penalization, 445  
 penalty parameter, 446  
 periodic sequence, 535  
 permutation, 128  
 Permutation matrix, 128  
 permutation matrix, 128  
 perturbation lemma, 108  
 Petrov-Galerkin condition, 528  
 phase space  
     of an ODE, 607  
 Picard-Lindelöf  
     Theorem of, 609  
 Piecewise cubic Hermite interpolant (with exact slopes) → Def. 3.4.1, 352  
 PINVIT, 451  
 Pivot  
     choice of, 126  
     pivot, 111–113  
     pivot row, 111, 113  
     pivoting, 124  
 Planar triangulation, 155  
 point spread function, 553  
 polynomial  
     characteristic, 423  
     generalized Lagrange, 254  
     Lagrange, 251  
 Polynomial  
     Interpolation tensor product, 582  
 polynomial fitting, 396  
 polynomial interpolation  
     existence and uniqueness, 252  
     generalized, 253  
 polynomial space, 250  
 positive definite  
     criteria, 37  
     matrix, 36  
 potentials  
     nodal, 105  
 power spectrum  
     of a signal, 549  
 preconditioned CG method, 522  
 preconditioned inverse iteration, 451  
 preconditioner, 521  
 preconditioning, 521  
 predator-prey model, 603  
 principal axis, 490  
 principal axis transformation, 507  
 principal component, 489, 494  
 principal component analysis, 481  
 principal minor, 123  
 problem  
     ill conditioned, 109  
     ill-conditioned, 104  
     sensitivity, 107  
     well conditioned, 109  
 procedural form, 359  
 product rule, 222  
 propagated error, 623  
 pseudoinverse, 399, 408  
 pullback, 302, 361  
 Punkt  
     stationär, 604  
 pwer method  
     direct, 436  
 Python, 40

QR algorithm, 426  
 QR-algorithm with shift, 426  
 QR-decomposition, 77  
 quadratic complexity, 69  
 quadratic convergence, 202  
 quadratic eigenvalue problem, 419  
 quadratic functional, 503  
 quadratic inverse interpolation, 215  
 quadratrical inverse interpolation, 211  
 quadrature  
     adaptive, 387  
     polynomial formulas, 364  
 quadrature formula  
     order, 367  
 Quadrature formula/quadrature rule, 361  
 quadrature node, 361  
 quadrature numerical, 359  
 quadrature weight, 361  
 quasi-linear system, 224  
 Quasi-Newton method, 236, 238

Radau RK-method  
 order 3, 678  
 order 5, 678  
 radiative heat transfer, 536  
 rank  
 column rank, 107  
 computation, 485  
 numerical, 408  
 of a matrix, 107  
 row rank, 107  
 Rank of a matrix, 107  
 rank-1 modification, 116  
 rank-1-matrix, 71  
 rank-1-modification, 145, 237  
 rate  
 of algebraic convergence, 305  
 of convergence, 187  
 Rayleigh quotient, 437, 444  
 Rayleigh quotient iteration, 450  
 Region of (absolute) stability, 660  
 regular matrix, 107  
 Regular refinement of a planar triangulation, 159  
 Relative error, 83  
 relative error, 83  
 relative tolerance, 192, 219, 637  
 rem:Spec, 543  
 Residual, 134  
 residual quantity, 452  
 Riccati differential equation, 613, 614  
 Riemann sum, 561  
 right hand side  
 of an ODE, 607  
 right hand side vector, 105  
 rigid body mode, 458  
 Ritz projection, 465, 469  
 Ritz value, 465  
 Ritz vector, 465  
 root of unity, 541  
 roots of unity, 386  
 rounding, 83  
 rounding up, 83  
 roundoff  
 for numerical differentiation, 91  
 row major matrix format, 46  
 ROW methods, 682  
 row sum norm, 101  
 row transformation, 60, 111, 118  
 Runge's example, 267  
 Runge-Kutta  
 increments, 629, 672  
 Runge-Kutta method, 629, 672  
 L-stable, 677  
 Runge-Kutta methods  
 embedded, 643  
 semi-implicit, 679  
 stability function, 652, 674

saddle point problem, 412  
 matrix form, 413  
 scalar ODE, 613  
 scaling  
 of a matrix, 58  
 scheme  
 Horner, 250  
 Schur  
 Komplement, 124  
 Schur complement, 124, 141, 145  
 scientific notation, 80  
 secant condition, 237  
 secant method, 212, 215, 237  
 segmentation  
 of an image, 439  
 semi-implicit Euler method, 680  
 seminorm, 310  
 sensitive dependence, 104  
 sensitivity  
 of a problem, 107  
 shape  
 preservation, 274  
 preserving spline interpolation, 289  
 Sherman-Morrison-Woodbury formula, 145  
 shifted inverse iteration, 449  
 similarity  
 of matrices, 424  
 similarity function  
 for image segmentation, 440  
 similarity transformations, 424  
 similiary transformation  
 unitary, 426  
 Simpson rule, 365  
 sine  
 basis, 569  
 matrix, 569  
 transform, 569  
 Sine transform, 569  
 single precision, 82  
 Single step method, 618  
 single step method, 618  
 A-stability, 675  
 singular value decomposition, 480, 483  
 Singular value decomposition (SVD), 483  
 slopes  
 for cubic Hermite interpolation, 275  
 Smoothed triangulation, 157  
 Solution of an ordinary differential equation, 602

Sparse matrices, 147  
 Sparse matrix, 147  
 sparse matrix, 147  
 COO format, 148  
 initialization, 151  
 LU-factorization, 165  
 multiplication, 153  
 triplet format, 148  
 sparse matrix storage formats, 148  
 spectral condition number, 510  
 spectral partitioning, 448  
 spectral radius, 423  
 spectrum, 423  
 of a matrix, 507  
 spline, 281  
 cardinal, 288  
 complete cubic, 284  
 cubic, 282  
 cubic, locality, 288  
 knots, 281  
 natural cubic, 284  
 periodic cubic, 285  
 physical, 286  
 shape preserving interpolation, 289  
 Splines, 281  
 splitting  
 Lie-Trotter, 683  
 Strang, 683  
 splitting methods, 682  
 inexact, 684  
 spy, 57  
 stability function  
 of explicit Runge-Kutta methods, 652  
 of Runge-Kutta methods, 674  
 stable  
 algorithm, 103  
 numerically, 103  
 Stable algorithm, 103  
 stages, 672  
 state space  
 of an ODE, 607  
 stationary distribution, 431  
 steepest descent, 504  
 Stiff IVP, 664  
 stiffness matrix, 457  
 stochastic matrix, 431  
 stochastic simulation of page rank, 430  
 stopping rule, 191  
 Strang splitting, 683  
 Strassen's algorithm, 70  
 Structurally symmetric matrix, 174  
 structurally symmetric matrix, 174  
 sub-matrix, 35  
 sub-multiplicative, 101  
 subspace correction, 510  
 subspace iteration  
 for direct power method, 467  
 subspaces  
 nested, 510  
 SuperLU, 163  
 SVD, 480, 483  
 symmetric matrix, 36  
 Symmetric positive definite (s.p.d.) matrices, 36  
 symmetry  
 structural, 174  
 system matrix, 105  
 system of equations  
 linear, 104

tangent field, 613  
 Taylor expansion, 201  
 Taylor polynomial, 299  
 Taylor's formula, 299  
 template, 15  
 Tensor product  
 Chebyshev interpolation polynomial, 582, 592  
 interpolation polynomial, 582  
 tensor product, 54  
 Tensor product interpolation polynomial, 582  
 tent function, 247  
 Toeplitz matrices, 574  
 termination criterion, 191  
 ideal, 192  
 Newton iteration, 230  
 residual based, 193  
 Theorem: → [42, Thm. 25.4], 451  
 Theorem:  $L^\infty$  polynomial best approximation estimate, 302  
 Theorem: (Absolute) stability of explicit RK-SSM for linear systems of ODEs, 659  
 Theorem: 3-term recursion for Chebychev polynomials, 314  
 Theorem: 3-term recursion for orthogonal polynomials, 333  
 Theorem: Uniform approximation by polynomials, 300  
 Theorem: Courant-Fischer min-max theorem → [28, Thm. 8.1.2], 444  
 Theorem: Banach's fixed point theorem, 199  
 Theorem: best low rank approximation → [34, Thm. 11.6], 499  
 Theorem: Bound for spectral radius, 423  
 Theorem: Chebychev alternation theorem, 336  
 Theorem: Composition of analytic functions, 313  
 Theorem: Conditioning of LSEs, 108  
 Theorem: Convergence of gradient method/steepest descent, 509  
 Theorem: Convergence of approximation by cubic Hermite interpolation, 354  
 Theorem: Convergence of CG method, 518  
 Theorem: Convergence of direct power method → [15, Thm. 25.1], 439  
 Theorem: Cost for solving triangular systems, 137  
 Theorem: Cost of Gaussian elimination, 137  
 Theorem: Criteria for invertibility of matrix, 107

Theorem: Dimension of space of polynomials, 250  
Theorem: Divergent polynomial interpolants, 307  
Theorem: Envelope and fill-in, 172  
Theorem: Equivalence of all norms on finite dimensional vector spaces, 188  
Theorem: Existence & uniqueness of generalized Lagrange interpolation polynomials, 254  
Theorem: Existence & uniqueness of Lagrange interpolation polynomial, 252  
Theorem: Existence of  $n$ -point quadrature formulas of order  $2n$ , 372  
Theorem: Gaussian elimination for s.p.d. matrices, 179  
Theorem: Gram-Schmidt orthonormalization, 330  
Theorem: Implicit function theorem, 616  
Theorem: Isometry property of Fourier transform, 563  
Theorem: Local quadratic convergence of Newton's method, 229  
Theorem: Local shape preservation by piecewise linear interpolation, 274  
Theorem: Maximal order of  $n$ -point quadrature rule, 370  
Theorem: Mean square (semi-)norm/Inner product (semi-)norm, 327  
Theorem: Mean square norm best approximation through normal equations, 328  
Theorem: Minimax property of the Chebychev polynomials, 316  
Theorem: Monotonicity preservation of limited cubic Hermite interpolation, 280  
Theorem: Optimality of natural cubic spline interpolant, 286  
Theorem: Order of collocation single step method, 671  
Theorem: Order of simple splitting methods, 684  
Theorem: Positivity of Clenshaw-Curtis weights, 366  
Theorem: Property of linear, monotonicity preserving interpolation into  $C^1$ , 280  
Theorem: Pseudoinverse and SVD, 408  
Theorem: QR-decomposition, 78  
Theorem: Quadrature error estimate for quadrature rules with positive weights, 377  
Theorem: Rayleigh quotient, 444  
Theorem: Region of stability of Gauss collocation single step methods, 676  
Theorem: Representation of interpolation error, 307  
Theorem: Residue theorem, 311  
Theorem: Schur's lemma, 424  
Theorem: singular value decomposition → [59, Thm. 9.6], [34, Thm. 11.1], 482  
Theorem: Stability function of Runge-Kutta methods, cf. Thm. 12.1.15, 674  
Theorem: Stability function of explicit Runge-Kutta methods, 652  
Theorem: Stability of Gaussian elimination with partial pivoting, 131  
Theorem: Sufficient order conditions for quadrature rules, 368  
Theorem: Taylor's formula, 201  
Theorem: Theorem of Peano & Picard-Lindelöf [5, Satz II(7.6)], [77, Satz 6.5.1], [15, Thm. 11.10], [42, Thm. 73.1], 609  
time-invariant filter, 532  
timestep (size), 614  
timestep constraint, 654  
timestepping, 613  
Toeplitz matrix, 576  
Toeplitz solvers  
  fast algorithms, 578  
tolerance, 193  
  absolute, 637  
  absolute, 192, 219  
  for adaptive timestepping for ODEs, 636  
  for termination, 192  
  relative, 637  
  relative, 192, 219  
total least squares, 410  
trajectory, 604  
transform  
  cosine, 573  
  fast Fourier, 564  
  sine, 569  
transformation matrix, 60  
trapezoidal rule, 365, 385, 628  
  for ODEs, 628  
trend, 481  
trial space  
  for collocation, 668  
triangle inequality, 100  
triangular linear systems, 141  
triangulation, 155  
tridiagonal matrix, 171  
trigonometric basis, 542  
trigonometric function, 340  
trigonometric interpolation, 339  
trigonometric polynomial, 562  
trigonometric polynomials, 340  
trigonometric transformations, 569  
tripled format, 148  
truss structure  
  vibrations, 455  
trust region method, 418  
Types of asymptotic convergence of approximation schemes, 305  
Types of matrices, 36

UMFPACK, 163  
unconstrained optimization, 242  
underflow, 82, 85  
uniform approximation, 297  
uniform best approximation, 336  
Uniform convergence  
  of Fourier series, 559  
unit vector, 34  
unitary similiary transformation, 426  
upper Hessenberg matrix, 476  
upper triangular matrix, 36, 111, 119

Vandermonde matrix, 253  
variational calculus, 286  
vector field, 607  
Vieta's formula, 92

Weddle rule, 366  
weight  
  quadrature, 361  
weight function, 331  
weighted  $L^2$ -inner product, 331

# List of Symbols

- $(\mathbf{A})_{i,j} \hat{=} \text{reference to entry } a_{ij} \text{ of matrix } \mathbf{A}, 35$   
 $(\mathbf{A})_{k:l,r:s} \hat{=} \text{reference to submatrix of } \mathbf{A} \text{ spanning rows } k, \dots, l \text{ and columns } r, \dots, s, 35$   
 $(\mathbf{x})_i \hat{=} i\text{-th component of vector } \mathbf{x}, 34$   
 $(x_k) *_n (y_k) \hat{=} \text{discrete periodic convolution}, 536$   
 $C(I) \hat{=} \text{space of continuous functions } I \rightarrow \mathbb{R}, 268$   
 $C^1([a, b]) \hat{=} \text{space of continuously differentiable functions } [a, b] \mapsto \mathbb{R}, 275$   
 $J(t_0, \mathbf{y}_0) \hat{=} \text{maximal domain of definition of a solution of an IVP}, 609$   
 $O \hat{=} \text{zero matrix}, 36$   
 $O(\cdot) \hat{=} \text{Landau symbol}, 69$   
 $\mathcal{E} \hat{=} \text{expected value of a random variable}, 575$   
 $\mathcal{P}_n^T \hat{=} \text{space of trigonometric polynomials of degree } n, 340$   
 $\mathcal{R}_k(m, n), 499$   
 $D\Phi \hat{=} \text{Jacobian of } \Phi : D \mapsto \mathbb{R}^n \text{ at } \mathbf{x} \in D, 199$   
 $D_y \mathbf{f} \hat{=} \text{Derivative of } \mathbf{f} \text{ w.r.t. } y \text{ (Jacobian)}, 609$   
 $\text{EPS} \hat{=} \text{machine precision}, 84$   
 $\text{Eig} \mathbf{A} \lambda \hat{=} \text{eigenspace of } \mathbf{A} \text{ for eigenvalue } \lambda, 423$   
 $\text{Im } \mathbf{A} \hat{=} \text{range/column space of matrix } \mathbf{A}, 485$   
 $\text{Kern } \mathbf{A} \hat{=} \text{nullspace of matrix } \mathbf{A}, 485$   
 $\mathcal{K}_l(\mathbf{A}, \mathbf{z}) \hat{=} \text{Krylov subspace}, 511$   
 $\|\mathbf{Ax} - \mathbf{b}\|_2 \rightarrow \min \hat{=} \text{minimize } \|\mathbf{Ax} - \mathbf{b}\|_2, 398$   
 $\|\mathbf{A}\|_F^2, 498$   
 $\|\mathbf{x}\|_A \hat{=} \text{energy norm induced by s.p.d. matrix } \mathbf{A}, 503$   
 $\|\cdot\| \hat{=} \text{Euclidean norm of a vector } \in \mathbb{K}^n, 76$   
 $\|\cdot\| \hat{=} \text{norm on vector space}, 100$   
 $\|f\|_{L^\infty(I)}, 268$   
 $\|f\|_{L^1(I)}, 268$   
 $\|f\|_{L^2(I)}^2, 268$   
 $\mathcal{P}_k, 250$   
 $\Psi^h \mathbf{y} \hat{=} \text{discretei evolution for autonomous ODE}, 618$   
 $(\cdot, \cdot)_V \hat{=} \text{inner product on vector space } V, 327$   
 $\mathcal{S}_{d,\mathcal{M}}, 281$   
 $\mathbf{A}^+, 399$   
 $\mathbf{A}^\top \hat{=} \text{transposed matrix}, 35$   
 $\mathbf{I} \hat{=} \text{identity matrix}, 36$   
 $\mathbf{h} * \mathbf{x} \hat{=} \text{discrete convolution of two vectors}, 535$   
 $\mathbf{x} *_n \mathbf{y} \hat{=} \text{discrete periodic convolution of vectors}, 536$   
 $\bar{z} \hat{=} \text{complex conjugation}, 35$   
 $\mathbb{C}^- := \{z \in \mathbb{C}: \text{Re } z < 0\}, 675$   
 $\mathbb{K} \hat{=} \text{generic field of numbers, either } \mathbb{R} \text{ or } \mathbb{C}, 33$   
 $\mathbb{M} \hat{=} \text{set of machine numbers}, 80$   
 $\delta_{ij} \hat{=} \text{Kronecker symbol}, 34, 251$   
 $\mathbf{i} \hat{=} \text{imaginary unit, "i" := } \sqrt{-1}, 106$   
 $\kappa(\mathbf{A}) \hat{=} \text{spectral condition number}, 510$   
 $\lambda_{\mathcal{T}} \hat{=} \text{Lebesgue constant for Lagrange interpolation on node set } \mathcal{T}, 269$   
 $\lambda_{\max} \hat{=} \text{largest eigenvalue (in modulus)}, 510$   
 $\lambda_{\min} \hat{=} \text{smallest eigenvalue (in modulus)}, 510$   
 $\mathbf{1} = [1, \dots, 1]^\top, 652$   
 $\text{Ncut}(\mathcal{X}) \hat{=} \text{normalized cut of subset of weighted graph}, 440$   
 $\text{argmin} \hat{=} \text{(global) minimizer of a functional}, 504$   
 $\text{cond}(\mathbf{A}), 108$   
 $\text{cut}(\mathcal{X}) \hat{=} \text{cut of subset of weighted graph}, 440$   
 $\text{env}(\mathbf{A}), 171$   
 $\text{nnz}, 147$   
 $\text{rank}(\mathbf{A}) \hat{=} \text{rank of matrix } \mathbf{A}, 107$   
 $\text{sgn} \hat{=} \text{sign function}, 279$   
 $\text{weight}(\mathcal{X}) \hat{=} \text{connectivity of subset of weighted graph}, 440$   
 $\overline{\phantom{x}} \hat{=} \text{complex conjugation}, 327$   
 $\overline{m}(\mathbf{A}), 171$   
 $\rho(\mathbf{A}) \hat{=} \text{spectral radius of } \mathbf{A} \in \mathbb{K}^{n,n}, 423$   
 $\rho_{\mathbf{A}}(\mathbf{u}) \hat{=} \text{Rayleigh quotient}, 437$   
 $\mathbf{f} \hat{=} \text{right hand side of an ODE}, 607$   
 $\sharp \hat{=} \text{cardinality of a finite set}, 38$   
 $\sigma(\mathbf{A}) \hat{=} \text{spectrum of matrix } \mathbf{A}, 423$   
 $\sigma(\mathbf{M}) \text{ hat= spectrum of matrix } \mathbf{M}, 507$   
 $\widetilde{x}, 84$   
 $S^1 \hat{=} \text{unit circle in the complex plane}, 341$   
 $\underline{m}(\mathbf{A}), 171$   
 $f^{(k)} \hat{=} k\text{-th derivative of } f : I \subset \mathbb{R} \rightarrow \mathbb{K}, 301$   
 $f^{(k)} \hat{=} k \text{ derivative of } f, 99$   
 $m(\mathbf{A}), 171$   
 $y[t_i, \dots, t_{i+k}] \hat{=} \text{divided difference}, 264$   
 $\|\mathbf{x}\|_1, 100$   
 $\|\mathbf{x}\|_2, 100$

$\| \mathbf{x} \|_\infty$ , 100

$\cdot \hat{\triangle}$  Derivative w.r.t. time  $t$ , 602

TOL tolerance, 636

# Examples and Remarks

- LU*-decomposition of sparse matrices, 165  
 $L^2$ -error estimates for polynomial interpolation, 309  
 $h$ -adaptive numerical quadrature, 392  
 $p$ -convergence of piecewise polynomial interpolation, 352  
(Nearly) singular LSE in shifted inverse iteration, 449  
Linear data fitting → [15, Sect. 4.1], 395  
**Compressed row-storage (CRS)** format, 149  
BLAS calling conventions, 66  
EIGEN in use, 45  
General non-linear systems of equations, 184  
**Discrete finite linear time-invariant causal channel (filter)** 532  
**linear regression** → [15, Ex. 4.1], 397  
ode45 for stiff problem, 647  
'Partial LU-decompositions' of principal minors, 123  
"Behind the scenes" of **MyVector** arithmetic, 28  
"Butcher barriers" for explicit RK-SSM, 631  
"Failure" of adaptive timestepping, 640  
"Fast" matrix multiplication, 70  
"Low" and "high" frequencies, 547  
"Squeezed" DFT of a periodically truncated signal, 556  
 $\mathbf{B} = \mathbf{B}^H$  s.p.d. mit Cholesky-Zerlegung, 425  
L-stable implicit Runge-Kutta methods, 678  
fft  
    Efficiency, 563  
2-norm from eigenvalues, 509  
3-Term recursion for Legendre polynomials, 375  
Analytic solution of homogeneous linear ordinary differential equations, 422  
Classification from measured data, 481  
Convergence of PINVIT, 453  
Convergence of subspace variant of direct power method, 468  
Data points confined to a subspace, 487  
Direct power method, 437  
Eigenvalue computation with Arnoldi process, 479  
Image compression, 500  
Impact of roundoff on Lanczos process, 473  
Lagrange polynomials for uniformly spaced nodes, 252  
Lanczos process for eigenvalue computation, 473  
Page rank algorithm, 429  
PCA for data classification, 488  
PCA of stock prices, 494  
Power iteration with Ritz projection, 466  
qr based orthogonalization, 464  
Rayleigh quotient iteration, 450  
Resonances of linear electrical circuits, 419  
Ritz projections onto Krylov space, 470  
Runtimes of eig, 427  
Stability of Arnoldi process, 478  
Subspace power iteration with orthogonal projection, 461  
Trend analysis, 480  
Vibrations of a truss structure, 455  
A data type designed for of interpolation problem, 249  
A function that is not locally Lipschitz continuous, 608  
A posteriori error bound for linearly convergent iteration, 194  
A posteriori termination criterion for linearly convergent iterations, 193  
A posteriori termination criterion for plain CG, 515  
A special quasi-linear system of equations, 224  
Accessing matrix data as a vector, 46  
Accessing rows and columns of sparse matrices, 150  
Adapted Newton method, 209  
Adaptive integrator for stiff problems in MATLAB, 682  
Adaptive quadrature in MATLAB, 393  
Adaptive timestepping for mechanical problem, 644  
Adding EPS to 1, 85  
Adding EPS to 1, 85  
Affine invariance of Newton method, 221  
Analysis of trigonometric polynomials, 599  
Application of modified Newton methods, 211  
Approximate computaton of Fourier coefficients, 387

- Approximation by discrete polynomial fitting, 334  
 Arnoldi process Ritz projection, 476  
 Asymptotic behavior of Lagrange interpolation error, 304  
 Auxiliary construction for shape preserving quadratic spline interpolation, 290
- Bad behavior of global polynomial interpolants, 272  
 Banach's fixed point theorem, 199  
 Bernstein approximants, 301  
 Block LU-factorization, 123  
 Block Gaussian elimination, 117  
 Blow-up, 634  
 Blow-up of explicit Euler method, 648  
 Blow-up solutions of vibration equations, 454  
 Bound for *asymptotic* rate of linear convergence, 200  
 Breakdown of associativity, 84  
 Broyden method for a large non-linear system, 241  
 Broyden's quasi-Newton method: convergence, 238  
 Butcher scheme for some explicit RK-SSM, 630
- Calling BLAS routines from C/C++, 67  
 Cancellation during the computation of relative errors, 91  
 Cancellation in decimal system, 88  
 Cancellation in Gram-Schmidt orthogonalisation, 91  
 Cancellation when evaluating difference quotients, 89  
 Cardinal shape preserving quadratic spline, 292  
 CG convergence and spectrum, 520  
 Characteristics of stiff IVPs, 666  
 Chebychev interpolation errors, 319  
 Chebychev interpolation of analytic function, 322  
 Chebychev interpolation of analytic functions, 321  
 Chebychev nodes, 318  
 Chebychev polynomials on arbitrary interval, 317  
 Chebychev representation of built-in functions, 326  
 Chebychev vs equidistant nodes, 317  
 Choice of quadrature weights, 368  
 Class PolyEval, 266  
 Clenshaw-Curtis quadrature rules, 366  
 Cluster Tree, 590  
 Combat cancellation by approximation, 98  
 Commonly used embedded explicit Runge-Kutta methods, 643  
 Communicating special properties of system matrices in MATLAB, 137
- Composite quadrature and piecewise polynomial interpolation, 382  
 Composite quadrature rules vs. global quadrature rules, 384  
 Computational effort for eigenvalue computations, 427  
 Computing Gauss nodes and weights, 376  
 Computing the zeros of a quadratic polynomial, 86  
 condition extended system, 403  
 Conditioning and relative error, 133  
 Conditioning of normal equations, 401  
 Conditioning of the extended normal equations, 403  
 Conditioning of the least squares problem, 400  
 Consistency of implicit midpoint method, 619  
 Constitutive relations from measurements, 245  
 Construction of higher order Runge-Kutta single step methods, 631  
 Contiguous arrays in C++, 20  
 Convergence monitors, 239  
 Convergence of CG as iterative solver, 516  
 Convergence of cluster approximation, 597  
 Convergence of clustering approximation with collocation matrix, 598  
 Convergence of Fourier sums, 560  
 Convergence of global quadrature rules, 378  
 Convergence of gradient method, 508  
 Convergence of Hermite interpolation, 354  
 Convergence of Hermite interpolation with exact slopes, 352  
 Convergence of inexact simple splitting methods, 684  
 Convergence of Krylov subspace methods for non-symmetric system matrix, 529  
 Convergence of naive semi-implicit Radau method, 681  
 Convergence of Newton's method in 2D, 228  
 Convergence of quadratic inverse interpolation, 215  
 Convergence of Remez algorithm, 339  
 Convergence of secant method, 212  
 Convergence of simple Runge-Kutta methods, 629  
 Convergence of simple splitting methods, 683  
 Convergence rates for CG method, 519  
 Convergence theory for PCG, 523  
 Convolution of sequences, 535  
 Cosine transforms for compression, 574  
 Damped Broyden method, 240  
 Damped Newton method, 235

- Deblurring by DFT, 553  
 Decay conditions for bi-infinite signals, 559  
 Decimal floating point numbers, 80  
 Derivative of Euclidean norm, 223  
 Derivative of a bilinear form, 222  
 Derivative of matrix inversion, 225  
 Detecting linear convergence, 188  
 Detecting order of convergence, 190  
 Detecting periodicity in data, 546  
 Determining the domain of analyticity, 313  
 Determining the type of convergence in numerical experiments, 305  
 Diagonally dominant matrices from nodal analysis, 177  
 Different choices for consistent fixed point iterations (II), 197  
 Different choices for consistent iteration functions (III), 202  
 Different meanings of “convergence”, 305  
 Discretization, 618  
 Distribution of machine numbers, 81  
 Divided differences and derivatives, 266  
 Efficiency of `fft`, 563  
 Efficiency of FFT-based LSE-solver, 572  
 Efficiency of iterative methods, 217  
 Efficiency of `fft`, 563  
 Efficient associative matrix multiplication, 71  
 Efficient evaluation of trigonometric interpolation polynomials, 344  
 Efficient Initialization of sparse matrices in MATLAB, 151  
 Eigenvectors of circulant matrices, 539  
 Embedded Runge-Kutta methods, 643  
 Empiric Convergence of collocation single step methods, 670  
 Empiric convergence of equidistant trapezoidal rule, 385  
 Envelope of a matrix, 171  
 Envelope oriented matrix storage, 174  
 Error of polynomial interpolation, 309  
 Error representation for generalized Lagrangian interpolation, 308  
 Estimation of “wrong quadrature error”? , 391  
 Estimation of “wrong” error? , 638  
 Euler methods for stiff decay IVP, 667  
 Evolution operator for Lotka-Volterra ODE, 611  
 Ex. 1.6.51 cnt'd, 129  
 Explicit Euler method as difference scheme, 614  
 Explicit Euler method for damped oscillations, 658  
 Explicit ODE integrator in MATLAB, 631  
 Explicit representation of error of polynomial interpolation, 308  
 Explicit trapezoidal rule for decay equation, 651  
 Exploiting trigonometric identities to avoid cancellation, 93  
 Extended normal equations, 402  
 Failure of damped Newton method, 236  
 Failure of Krylov iterative solvers, 529  
 Fast evaluation of Chebychev expansion, 323  
 Fast Toeplitz solvers, 578  
 Feasibility of implicit Euler timestepping, 615  
 FFT algorithm by matrix factorization, 565  
 FFT based on general factorization, 567  
 FFT for prime vector length, 567  
 Filtering in Fourier domain, 562  
 Finite linear time-invariant causal channel, 532  
 Finite-time blow-up, 609  
 Fit of hyperplanes, 409  
 Fixed points in 1D, 197  
 Fractional order of convergence of secant method, 213  
 Frequency filtering by DFT, 548  
 Frequency identification with DFT, 545  
 From higher order ODEs to first order systems, 607  
 Gain through adaptivity, 639  
 Gauss-Radau collocation SSM for stiff IVP, 678  
 Gaussian elimination, 111  
 Gaussian elimination and LU-factorization, 119  
 Gaussian elimination for non-square matrices, 115  
 Gaussian elimination via rank-1 modifications, 116  
 Gaussian elimination with pivoting for  $3 \times 3$ -matrix, 125  
 Generalized bisection methods, 204  
 Generalized eigenvalue problems and Cholesky factorization, 425  
 Generalized Lagrange polynomials for Hermite Interpolation, 254  
 Generalized polynomial interpolation, 253  
 Gibbs phenomenon, 346  
 Global separable approximation by non-smooth kernel function, 583  
 Global separable approximation by smooth kernel function, 582  
 Gradient method in 2D, 507  
 Gram-Schmidt orthonormalization based on **MyVec-tor** implementation, 29  
 Gravitational forces in galaxy, 580  
 Group property of autonomous evolutions, 611  
 Growth with limited resources, 602

- Halley's iteration, 208  
 Heartbeat model, 604  
 Heating production in electrical circuits, 360  
 Hidden summation, 73  
 Horner scheme, 250  
 Image segmentation, 439  
 Impact of choice of norm, 187  
 Impact of matrix data access patterns on runtime, 49  
 Impact of roundoff errors on CG, 516  
 Implicit differentiation of  $F$ , 207  
 Implicit nature of collocation single step methods, 670  
 Implicit RK-SSMs for stiff IVP, 676  
 Importance of numerical quadrature, 359  
 In-situ LU-decomposition, 122  
 Inequalities between vector norms, 100  
 Initial guess for power iteration, 439  
 Initialization of sparse matrices in Eigen, 162  
 Inner products on spaces  $\mathcal{P}_m$  of polynomials, 330  
 Input errors and roundoff errors, 82  
 Instability of multiplication with inverse, 135  
 Instability of normal equations, 401  
 Interaction calculations for many body systems, 579  
 interpolation  
   piecewise cubic monotonicity preserving, 280  
   shape preserving quadratic spline, 292  
 Interpolation and approximation: enabling technologies, 299  
 Interpolation error estimates and the Lebesgue constant, 310  
 Interpolation error: trigonometric interpolation, 345  
 Intersection of lines in 2D, 109  
 Justification of Ritz projection by min-max theorem, 465  
 Kinetics of chemical reactions, 661  
 Krylov methods for complex s.p.d. system matrices, 503  
 Krylov subspace methods for generalized EVP, 480  
 Least squares data fitting, 395  
 Lebesgue constant for equidistant nodes, 269  
 Linear filtering of periodic signals, 535  
 Linear regression for stationary Markov chains, 575  
 Linear systems with arrow matrices, 142  
 Lineare zeitinvariante Systeme, 574  
 Linearization of increment equations, 679  
 Linearly convergent iteration, 189  
 Linsolve versus backslash, 138  
 Local approximation by piecewise polynomials, 348  
 Local convergence of Newton's method, 232  
 local convergence of the secant method, 214  
 Loss of sparsity when forming normal equations, 402  
 LU-decomposition of flipped "arrow matrix", 167  
 Machine precision for IEEE standard, 85  
 Magnetization curves, 271  
 Many choices for consistent fixed point iterations, 196  
 Many sequential solutions of LSE, 139  
 Mathematical functions in a numerical code, 246  
 MATLAB command reshape, 48  
 Matrix algebra, 61  
 Matrix inversion by means of Newton's method, 226  
 Matrix norm associated with  $\infty$ -norm and 1-norm, 101  
 Matrix representation of interpolation operator, 253  
 Meaningful " $O$ -bounds" for complexity, 69  
 Midpoint rule, 364  
 Min-max theorem, 444  
 Minimality property of Broyden's rank-1-modification, 238  
 Model reduction by interpolation, 297  
 Monitoring convergence for Broyden's quasi-Newton method, 239  
 Multidimensional fixed point iteration, 200  
 Multiplication of Kronecker product with vector, 74  
 Multiplication of polynomials, 534  
 Multiplication of sparse matrices, 153  
 Multiplying matrices in MATLAB, 62  
 Multiplying triangular matrices, 58  
 Necessary condition for L-stability, 677  
 Necessity of iterative approximation, 185  
 Newton method and minimization of quadratic functional, 415  
 Newton method in 1D, 206  
 Newton's iteration; computational effort and termination, 230  
 Newton-Cotes formulas, 365  
 Nodal analysis of linear electric circuit, 105  
 Non-linear cubic Hermite interpolation, 280  
 Non-linear data fitting, 414  
 Non-linear data fitting (II), 416  
 Non-linear electric circuit, 183  
 Normal equations vs. orthogonal transformations method, 408

- Notation for single step methods, 619  
 Numerical Differentiation for computation of Jacobian, 227  
 Numerical integration of logistic ODE in MATLAB, 632  
 Numerical stability and sensitive dependence on data, 104  
 Numerical summation of Fourier series, 559  
 NumPy command reshape, 48  
 Occurrence of clusters in partition rectangles, 595  
 Orders of simple polynomial quadrature formulas, 369  
 Oregonator reaction, 633  
 Origin of the term “Spline”, 286  
 Oscillating polynomial interpolant, 267  
 Output of explicit Euler method, 614  
 Overdetermined linear systems, 398  
 Overflow and underflow, 85  
 Parameter identification for linear time-invariant filters, 574  
 Piecewise cubic Hermite interpolation, 277  
 Piecewise cubic interpolation schemes, 285  
 Piecewise linear interpolation, 247  
 Piecewise polynomial interpolation, 350  
 Piecewise quadratic interpolation, 274  
 Pivoting and numerical stability, 124  
 Pivoting destroys sparsity, 169  
 Polybomial interpolation vs. polynomial fitting, 396  
 Polynomial fitting, 396  
 Polynomials in MATLAB, 250  
 Power iteration, 435  
 Predator-prey model, 603  
 Predicting stiffness of non-linear IVPs, 665  
 Principal component analysis, 481  
 Principal component analysis for data analysis, 493  
 Pseudoinverse, 399  
 QR-Algorithm, 426  
 QR-decomposition, 77  
 QR-decomposition in MATLAB, 79  
 QR-decomposition in PYTHON, 79  
 Quadratic convergence, 191  
 Quadratic functional in 2D, 504  
 Quadratur  
     Gauss-Legendre Ordnung 4, 370  
 Quadrature errors for composite quadrature rules, 383  
 Quality measure for kernel approximation, 582  
 Radiative heat transfer, 536  
 Rank defect in linear least squares problems, 399  
 Rationale for adaptive quadrature, 388  
 Rationale for high-order single step methods, 626  
 Rationale for partial pivoting policy, 127  
 Rationale for using LU-decomposition in algorithms, 123  
 Recursive LU-factorization, 122  
 Reducing fill-in by reordering, 176  
 Reducing bandwidth by row/column permutations, 175  
 Reduction to periodic convolution, 538  
 Refined local stepsize control, 640  
 Regions of stability for simple implicit RK-SSM, 674  
 Regions of stability of some explicit RK-SSM, 660  
 Relative error and number of correct digits, 83  
 Relevance of asymptotic complexity, 70  
 Removing a singularity by transformation, 379  
 Reshaping matrices in EIGEN, 48  
 Residual based termination of Newton's method, 231  
 Resistance to currents map, 146  
 Restarted GMRES, 528  
 Row and column transformations, 60  
 Row swapping commutes with forward elimination, 129  
 Row-wise & column-wise view of matrix product, 55  
 Runge's example, 306, 309  
 Runtime comparison for computation of coefficient of trigonometric interpolation polynomials, 343  
 Runtime of Gaussian elimination, 114  
 S.p.d. Hessians, 37  
 Sacrificing numerical stability for efficiency, 144  
 Scaling a matrix, 58  
 Shape preservation of cubic spline interpolation, 287  
 Shifted inverse iteration, 449  
 Significance of smoothness of interpoland, 309  
 Silly MATLAB, 154  
 Simple adaptive stepsize control, 638  
 Simple adaptive timestepping for fast decay, 650  
 Simple composite polynomial quadrature rules, 381  
 Simple preconditioners, 524  
 Simple Runge-Kutta methods by quadrature & bootstrapping, 628  
 Simplified Newton method, 227  
 Sine transform via DFT of half length, 570  
 Small residuals by Gaussian elimination, 134

- Smoothing of a triangulation, 155  
 Solving the increment equations for implicit RK-  
     SSMs, 673  
 Sound filtering by DFT, 549  
 Sparse  $LU$ -factors, 167  
 Sparse elimination for arrow matrix, 162  
 Sparse LSE in circuit modelling, 147  
 Sparse matrices from the discretization of linear  
     partial differential equations, 147  
 Special cases in IEEE standard, 82  
 Spectrum of Fourier matrix, 543  
 Speed of convergence of Euler methods, 621  
 spline  
     interpolants, approx. complete cubic, 356  
     shape preserving quadratic interpolation, 292  
 Splines in MATLAB, 285  
 Splitting linear and local terms, 686  
 Splitting off stiff components, 685  
 Square root iteration as Newton's method, 206  
 Square root of a s.p.d. matrix, 521  
 Stability by small random perturbations, 132  
 Stability function and exponential function, 653  
 Stability functions of explicit Runge-Kutta single  
     step methods, 652  
 Stable discriminant formula, 92  
 Stage form equations for increments, 672  
 Stepsize control detects instability, 654  
 Stepsize control in MATLAB, 642  
 Strongly attractive limit cycle, 662  
 Subspace power methods, 468  
 Summation of exponential series, 97  
 SVD and additive rank-1 decomposition, 484  
 Switching to equivalent formulas to avoid cancel-  
     lation, 94  
  
 Tables of quadrature rules, 363  
 Tangent field and solution curves, 613  
 Taylor approximation, 299  
 Tensor product Chebyshev interpolation for vari-  
     able rectangle sizes, 586  
 Tensor product Chebyshev interpolation on rect-  
     angles, 586  
 Termination criterion for contractive fixed point iter-  
     ation, 202  
 Termination criterion for direct power iteration, 439  
 Termination criterion in `pcg`, 526  
 Termination of PCG, 525  
 Testing equality with zero, 85  
 Testing stability of matrix  $\times$  vector multiplication,  
     103  
 The “matrix  $\times$  vector-multiplication problem”, 100  
 The boundaries of  $\mathbb{M}$ , 81  
  
 The inverse matrix and solution of a LSE, 107  
 The message of asymptotic estimates, 379  
 Timing polynomial evaluations, 259  
 Timing sparse elimination for the combinatorial  
     graph Laplacian, 160  
 Transformation of polynomial approximation schemes,  
     302  
 Transformation of quadrature rules, 361  
 Transforming approximation error estimates, 303  
 Transient circuit simulation, 605  
 Transient simulation of RLC-circuit, 655  
 Tridiagonal preconditioning, 524  
 Trigonometric interpolation of analytic functions,  
     346  
  
 Understanding the structure of product matrices,  
     56  
 Uniqueness of SVD, 484  
 Unitary similarity transformation to tridiagonal form,  
     426  
 Unstable Gram-Schmidt orthonormalization, 77  
  
 Vandermonde matrix, 253  
 Visualization of explicit Euler method, 613  
  
 Weak locality of the natural cubic spline interpo-  
     lation, 288  
 Why using  $\mathbb{K} = \mathbb{C}$ ?, 541  
 Wilkinson's counterexample, 131