

Numerical Methods for Physics

(401-1662-10 L)

Dr. Vasile Gradinaru and Prof. Ralf Hiptmair

(C) Seminar für Angewandte Mathematik, ETH Zürich

Contents

0.1	About this course	7
0.2	Appetizer	14

I Systems of Equations19

1	Iterative Methods for Non-Linear Systems of Equations	20
1.1	Iterative methods	21
1.1.1	Speed of convergence	25
1.1.2	Termination criteria	36
1.2	Fixed Point Iterations	42
1.2.1	Consistent fixed point iterations	43
1.2.2	Convergence of fixed point iterations	47
1.3	Zero Finding	57

1.3.1	Bisection	58	Num. Meth. Phys.
1.3.2	Model function methods	60	
1.3.2.1	Newton method in scalar case	61	
1.3.2.2	Special one-point methods	62	
1.3.2.3	Multi-point methods	70	
1.4	Newton’s Method	77	
1.4.1	The Newton iteration	78	
1.4.2	Convergence of Newton’s method	90	
1.4.3	Termination of Newton iteration	93	
1.4.4	Damped Newton method	96	
1.4.5	Quasi-Newton Method	102	
1.5	Essential Skills Learned in Chapter 1	110	
2	Intermezzo on (Numerical)Linear Algebra	112	Gradinaru D-MATH
2.1	QR-Factorization/QR-decomposition	120	
2.2	Singular Value Decomposition	139	
2.3	Essential Skills Learned in Chapter 2	154	
3	Least Squares	155	
3.1	Normal Equations	161	
3.2	Orthogonal Transformation Methods	166	
3.3	Non-linear Least Squares	172	
3.3.1	(Damped) Newton method	174	
3.3.2	Gauss-Newton method	176	
3.3.3	Trust region method (Levenberg-Marquardt method)	181	
3.4	Essential Skills Learned in Chapter 3	183	

4	Eigenvalues	184	Num. Meth. Phys.
4.1	Theory of eigenvalue problems	190	
4.2	“Direct” Eigensolvers	195	
4.3	Power Methods	208	
4.3.1	Direct power method	208	
4.3.2	Inverse Iteration	213	
4.3.3	Preconditioned inverse iteration (PINVIT)	220	
4.3.4	Subspace iterations	228	
4.4	Krylov Subspace Methods	236	
4.5	Essential Skills Learned in Chapter 4	260	
II	Interpolation and Approximation	261	Gradinaru D-MATH
5	Polynomial Interpolation	267	
5.1	Polynomials	267	
5.2	Newton basis and divided differences [10, Sect. 8.2.4]	269	
5.3	Error estimates for polynomial interpolation	275	
5.4	Chebyshev Interpolation	288	
5.4.1	Motivation and definition	288	
5.4.2	Chebyshev interpolation error estimates	295	
5.4.3	Chebyshev interpolation: computational aspects	301	
5.5	Essential Skills Learned in Chapter 5	305	
			0.0
			p. 4

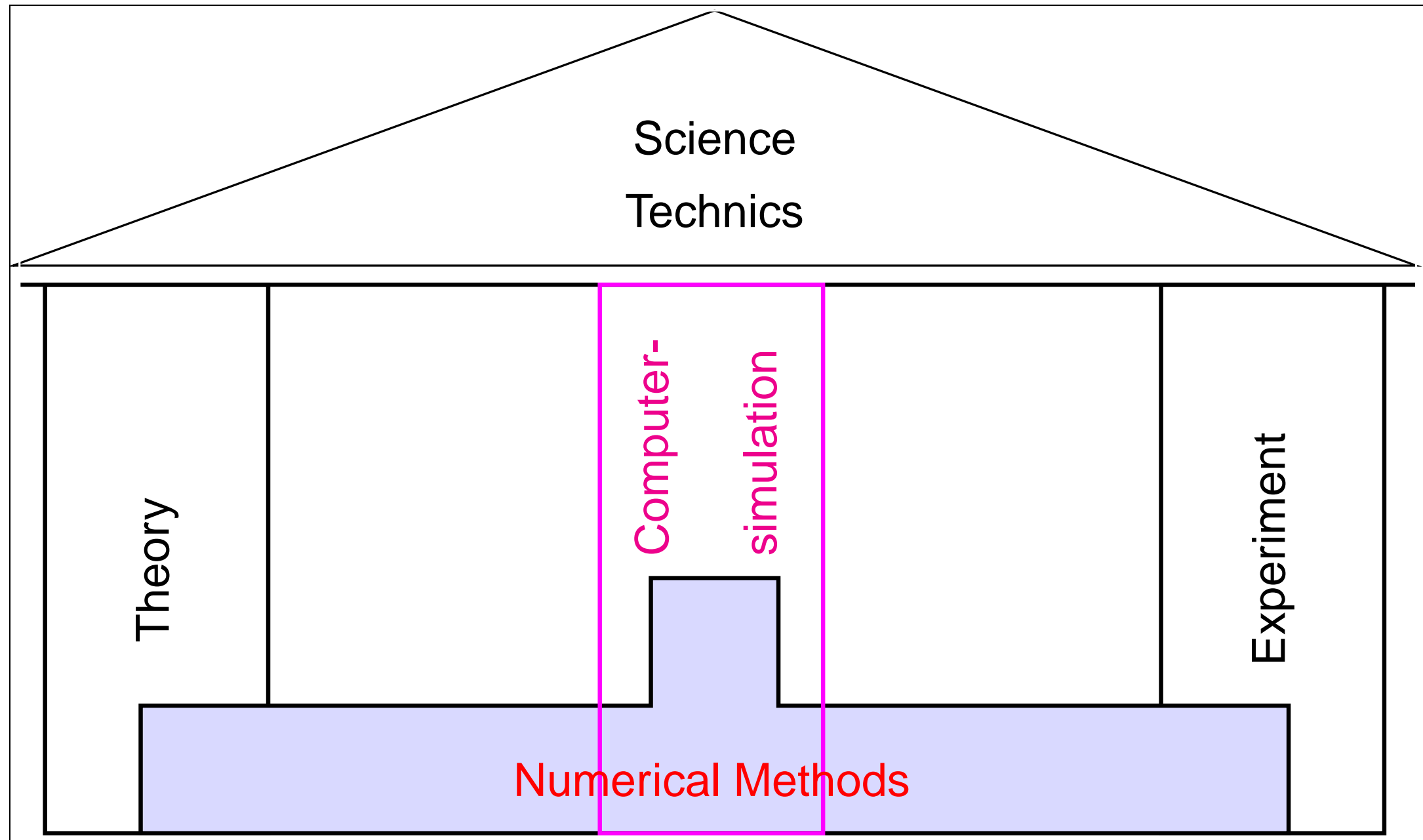
6	Trigonometric Interpolation	307
6.1	Discrete Fourier Transform (DFT)	312
6.2	Fast Fourier Transform (FFT)	316
6.3	Trigonometric Interpolation: Computational Aspects	331
6.4	Trigonometric Interpolation: Error Estimates	346
6.5	DFT and Chebychev Interpolation	353
6.6	Essential Skills Learned in Chapter 6	358
7	Numerical Quadrature	359
7.1	Quadrature Formulas	362
7.2	Composite Quadrature	378
7.3	Adaptive Quadrature	398
7.4	Multidimensional Quadrature	407
7.4.1	Quadrature on Classical Sparse Grids	409
7.5	Monte-Carlo Quadrature	415
7.6	Essential Skills Learned in Chapter 7	430

III Integration of Ordinary Differential Equations

432

8	Single Step Methods	433
8.1	Initial value problems (IVP) for ODEs	433
8.2	Euler methods	447
8.3	Convergence of single step methods	457
8.4	Structure Preservation	467
8.4.1	Implicit Midpoint Rule	470
8.4.2	Störmer-Verlet Method [?]	475
8.4.3	Examples	483
8.5	Splitting methods [27, Sect. 2.5]	494
8.6	Runge-Kutta methods	499
8.7	Stepsize control	511
8.8	Essential Skills Learned in Chapter 8	544

9	Stiff Integrators	546	Num. Meth. Phys.
9.1	Model problem analysis	551	
9.2	Stiff problems	559	
9.3	(Semi-)implicit Runge-Kutta methods	574	
9.4	Essential Skills Learned in Chapter 9	585	
			Gradinaru D-MATH
			0.1 p. 6



- ▷ on **algorithms** (principles, scope, and limitations)
- ▷ on **implementation** (efficiency, stability)
- ▷ on **numerical experiments** (design and interpretation)

no emphasis on

- theory and proofs (unless essential for understanding of algorithms)
- hardware-related issues (e.g. parallelization, vectorization, memory access)

Contents

Goals

- Knowledge of the fundamental algorithms in numerical mathematics
- Ability to choose the appropriate numerical method for concrete problems
- Ability to interpret numerical results
- Ability to implement numerical algorithms efficiently

Indispensable:

Learning by doing (→ exercises)

- ▷ active attendance of the lecture
- ▷ active participation to the exercises class
- ▷ at least **5 hours of additional work per week** (reading the lecture, answering the questions in the “Essential skills”-section, performing the numerical experiments from the slides and from the homework)

The final exam will take place at computer and will reside on **practical implementations**, based on the questions in the “Essential skills”-section of each chapter.

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

Some pieces of advice:

- these lecture slides are 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.

What to expect

• The course is difficult and demanding (*ie.* 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!

Books

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.

- M. HANKE-BOURGEOIS, *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens*, Mathematische Leitfäden, B.G. Teubner, Stuttgart, 2002.
- P. DEUFLHARD AND A. HOHMANN, *Numerische Mathematik. Eine algorithmisch orientierte Einführung*, DeGruyter, Berlin, 1 ed., 1991.

- J. Stör and R. Bulirsch, *Einführung in die Numerische Mathematik*, 1976. English version, *Intro. to Numerical Mathematics*, Springer Verlag, 1980. Latest edition is 2002.
- Quarteroni, Sacco and Saleri, *Numerische Mathematik 1 + 2*, Springer Verlag 2002

Essential prerequisite for this course is a solid knowledge in basic calculus and linear algebra. Familiarity with the topics covered in the textbook [39] is taken for granted.

Extra questions for course evaluation

Course number (LV-ID): 401-1662-10 L

Date of evaluation: 2.5.2011

D1: I try to do all programming exercises.

D2: The programming exercises help understand the numerical methods.

D3: The programming exercises offer too little benefit for the effort spent on them.

- D4:** Scope and limitations of methods are properly addressed in the course.
- D5:** Numerical examples in class provide useful insights and motivation.
- D6:** There should be more examples presented and discussed in class.
- D7:** Too much information is crammed onto the lecture slides
- D8:** The course requires too much prior knowledge in linear algebra
- D9:** The course requires too much prior knowledge in analysis
- 10:** My prior knowledge of Python was insufficient for the course
- 11:** More formal proofs would be desirable
- 12:** The explanations on the blackboard promote understanding
- 13:** The codes included in the lecture material convey useful information
- 14:** The model solutions for exercise problems offer too little guidance.
- 15:** The relevance of the numerical methods taught in the course is convincingly conveyed.

Scoring: 5: I agree fully
4: I agree to a large extent
3: I agree partly
2: I disagree
1: I disagree strongly

Please enter the shortcut code after the LV-ID in the three separate boxes.

Example 0.2.1 (Euler method for pendulum equation).

Hamiltonian form of equations of motion for pendulum

$$\text{angular velocity } p := \dot{\alpha} \Rightarrow \frac{d}{dt} \begin{pmatrix} \alpha \\ p \end{pmatrix} = \begin{pmatrix} p \\ -\frac{g}{l} \sin \alpha \end{pmatrix}, \quad g = 9.8, l = 1. \quad (0.2.1)$$

- numerical solution with explicit/implicit Euler method (8.2.1)/(8.2.4),
- constant time-step $h = T/N$, end time $T = 5$ fixed, $N \in \{50, 100, 200\}$,
- initial value: $\alpha(0) = \pi/4, p(0) = 0$.

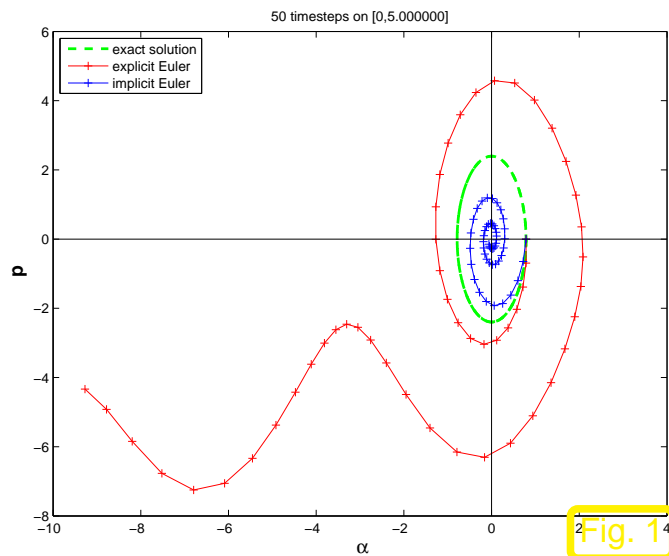


Fig. 1

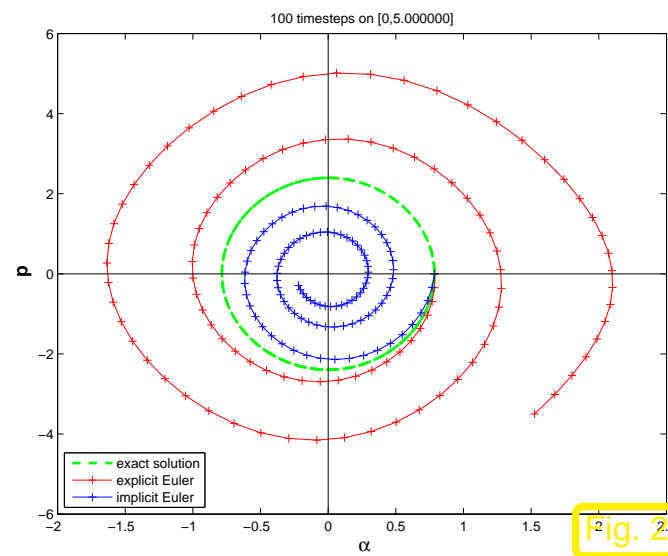


Fig. 2

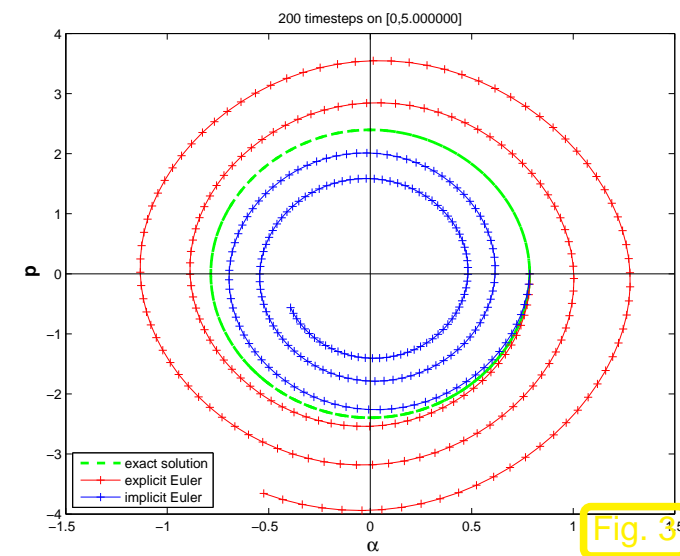


Fig. 3

Behavior of the computed energy: kinetic energy : $E_{\text{kin}}(t) = \frac{1}{2}p(t)^2$
 potential energy : $E_{\text{pot}}(t) = -\frac{g}{l} \cos \alpha(t)$

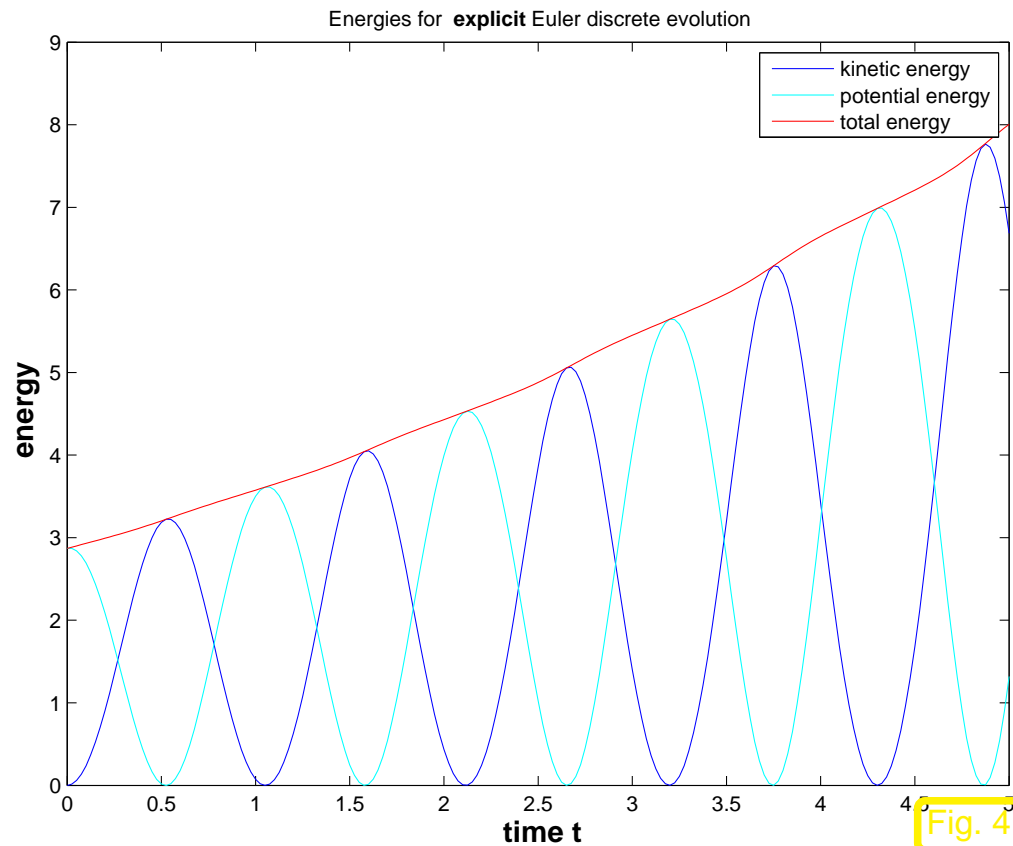


Fig. 4

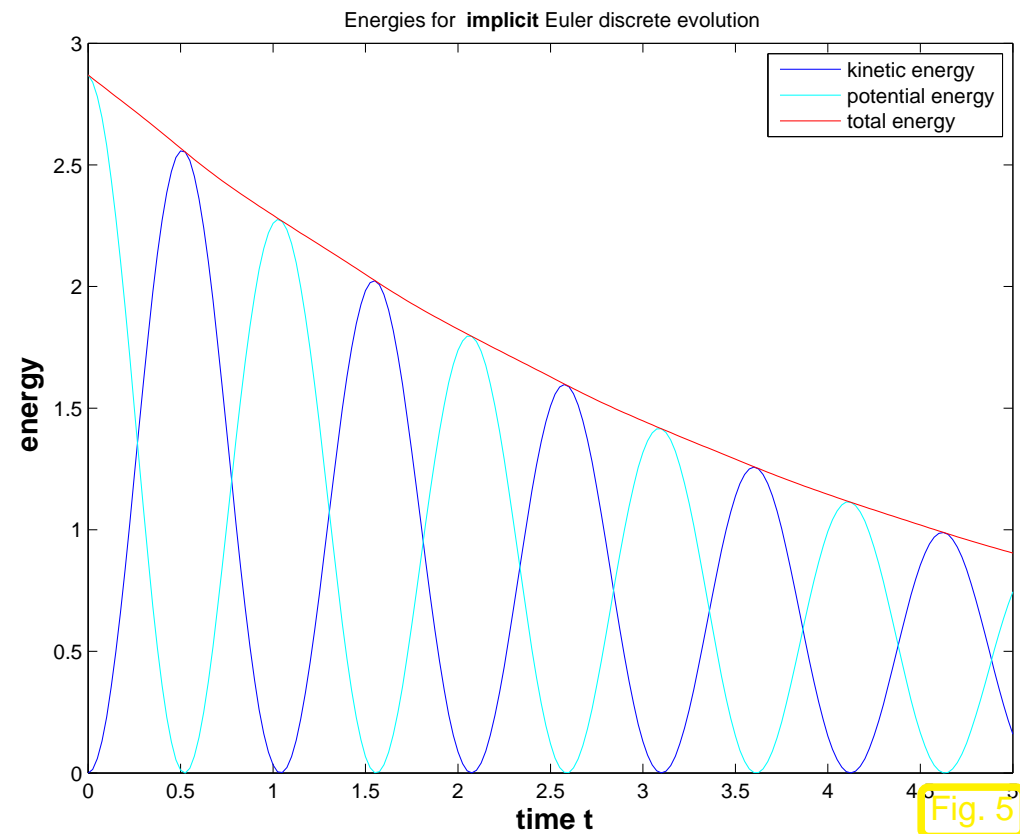


Fig. 5



explicit Euler: increase of total energy



implicit Euler: decrease of total energy ("numerical friction")



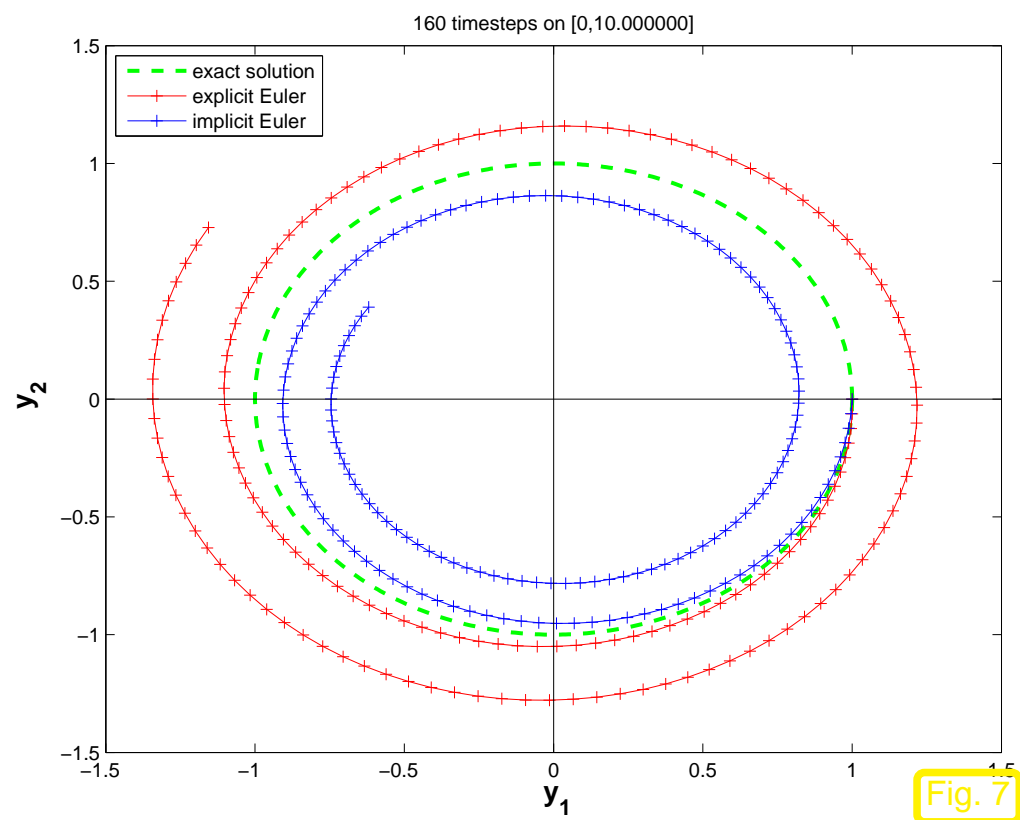
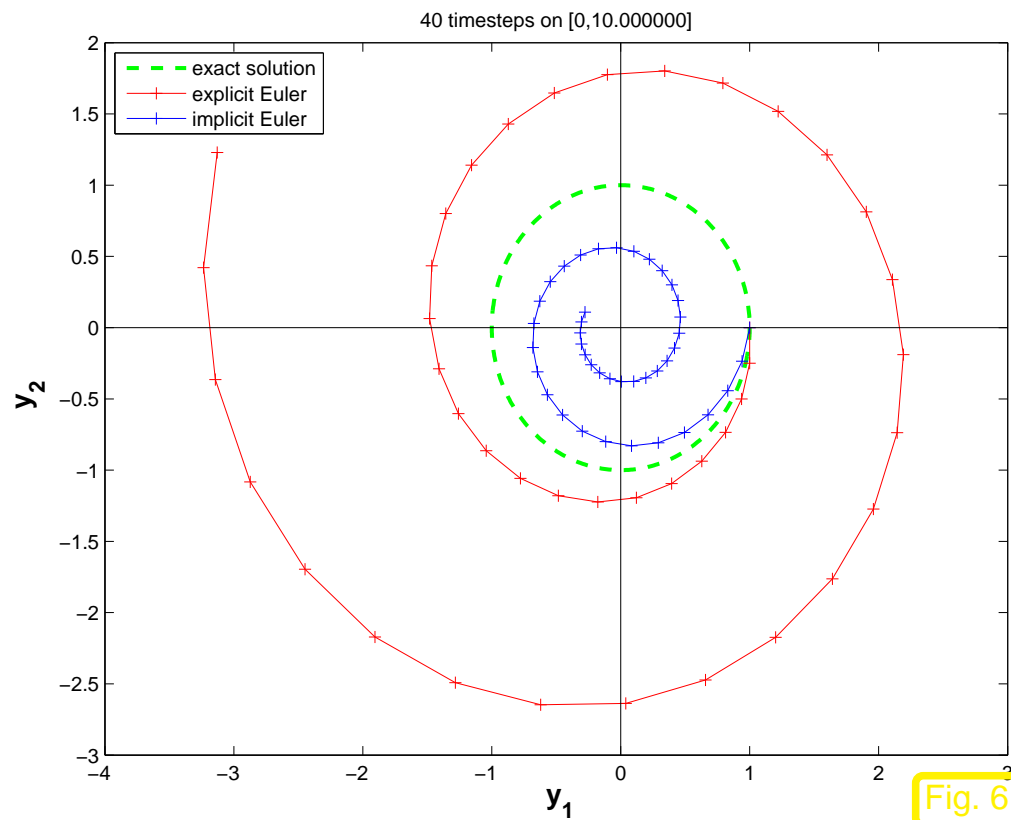
Example 0.2.2 (Euler method for long-time evolution).

Initial value problem for , $D = \mathbb{R}^2$:

$$\dot{\mathbf{y}} = \begin{pmatrix} y_2 \\ -y_1 \end{pmatrix}, \quad \mathbf{y}(0) = \mathbf{y}_0 \quad \blacktriangleright \quad \mathbf{y}(t) = \begin{pmatrix} \cos t & \sin t \\ -\sin t & \cos t \end{pmatrix} \mathbf{y}_0.$$

Note that $I(\mathbf{y}) = \|\mathbf{y}\|$ is constant.

(movement with constant velocity on the circle)



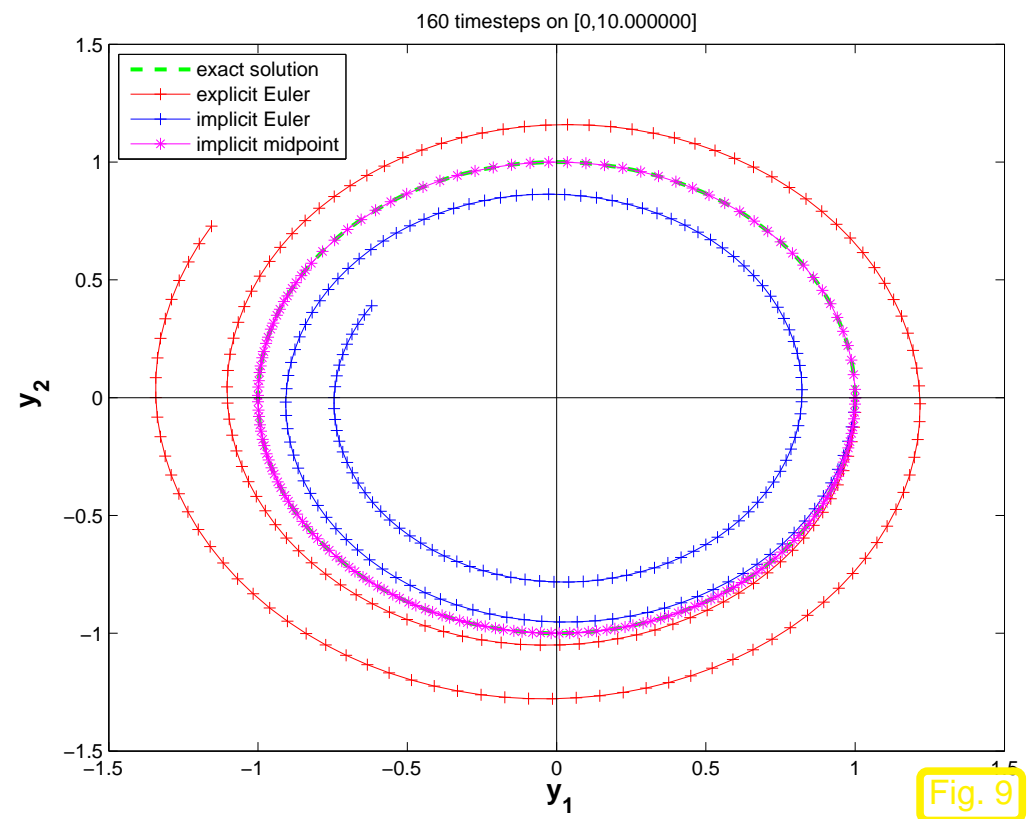
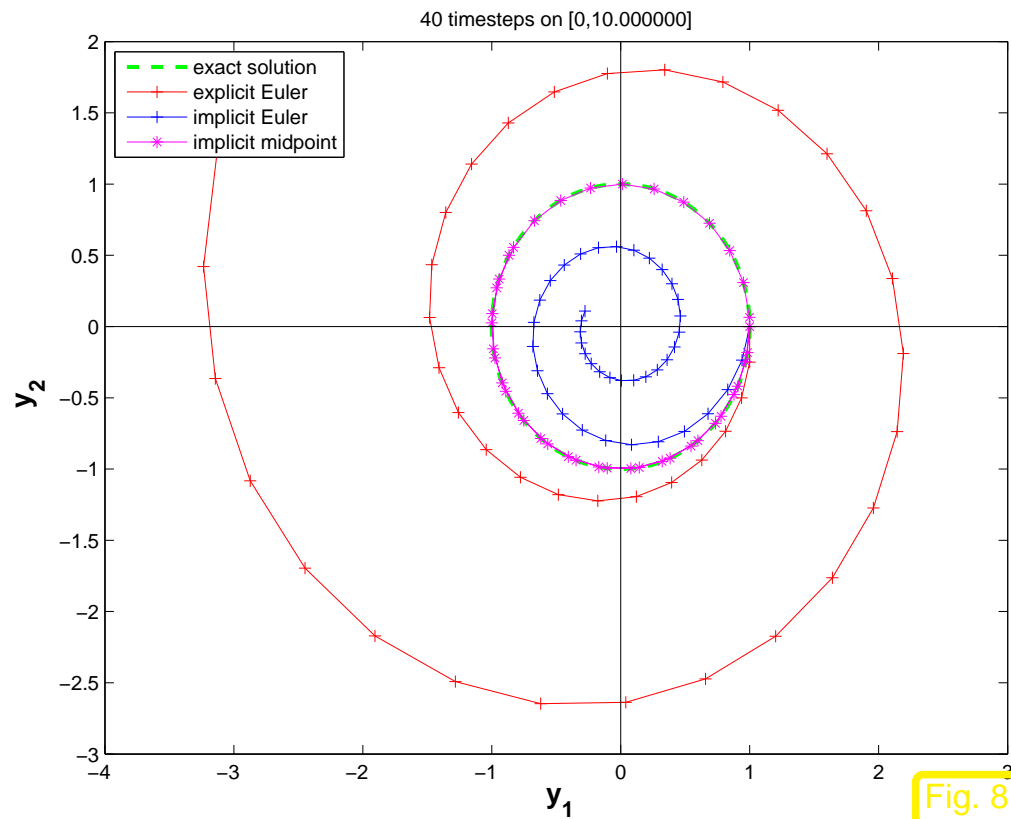
explicit Euler: numerical solution flies away

implicit Euler: numerical solution falls off into the center



Can we avoid the energy drift ?

Example 0.2.3 (Implicit midpoint rule for circular motion).

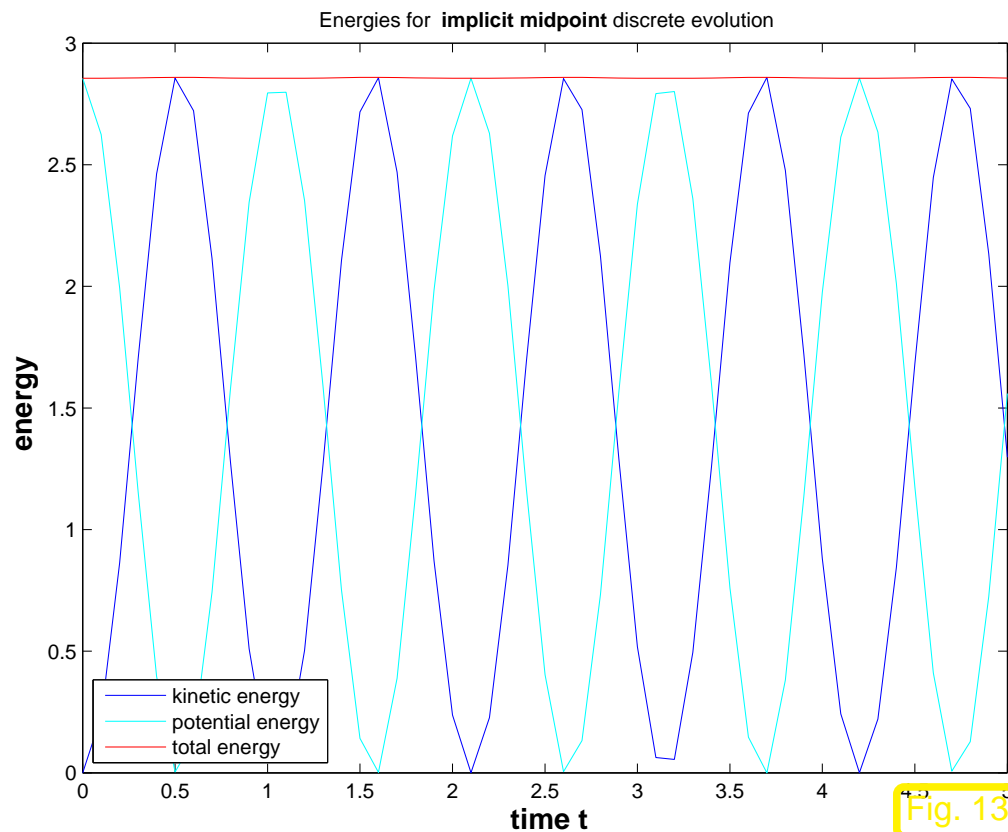
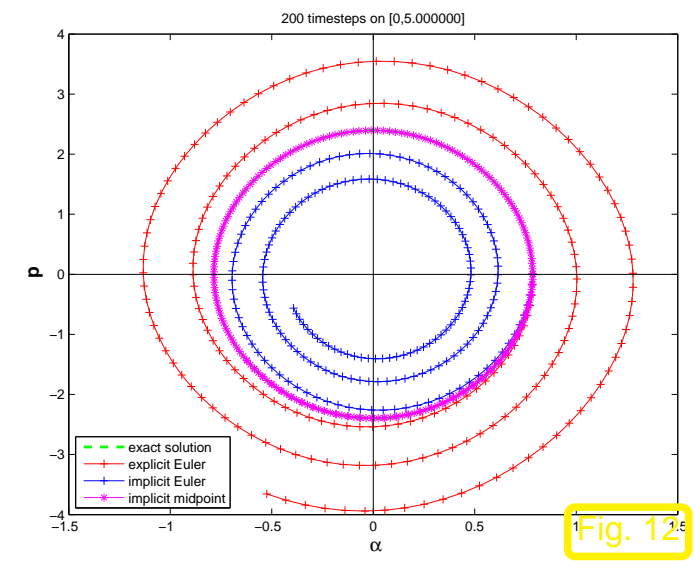
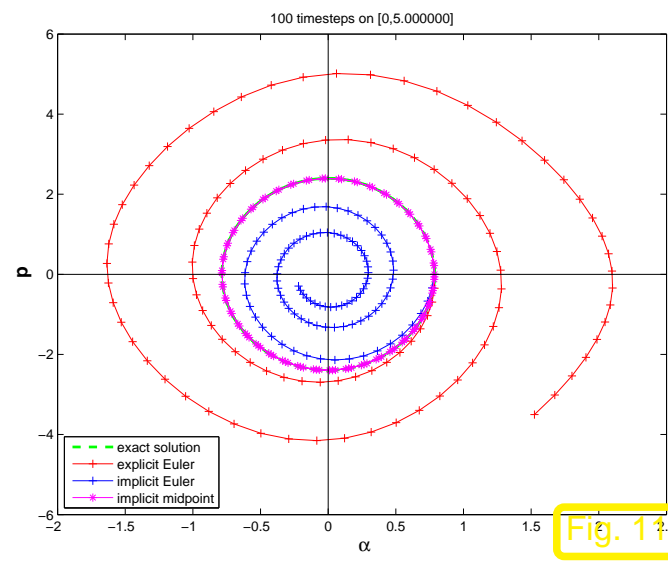
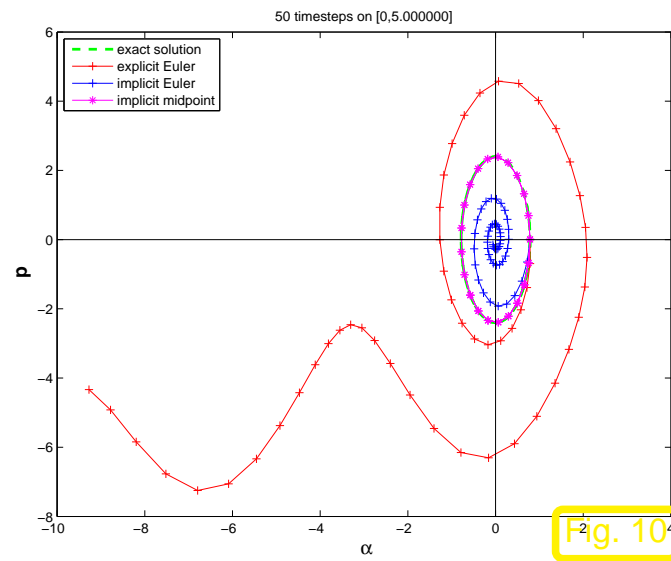


Implicit midpoint rule: perfect conservation of length !



Example 0.2.4 (Implicit midpoint rule for pendulum).

Initial values and problem as in Bsp. 8.4.1



◁ Behavior of the energy of the numerical solution computed with the midpoint rule (8.4.2), $N = 50$.
No energy drift although large time step)

Part I

Systems of Equations

1 Iterative Methods for 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.

Given: function $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$, $n \in \mathbb{N}$



Possible meaning: Availability of a **procedure** function $y=F(x)$ evaluating F

Sought: solution of **non-linear equation** $F(x) = 0$

Note: $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n \iff$ “same number of equations and unknowns”

In general no existence & uniqueness of solutions

Remark 1.1.1 (Necessity of iterative approximation).

Gaussian elimination provides an algorithm that, if carried out in exact arithmetic, 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. Certainly this is the case whenever irrational numbers are involved.



An **iterative method** for (approximately) solving the non-linear equation $F(\mathbf{x}) = 0$ is an algorithm generating a sequence $(\mathbf{x}^{(k)})_{k \in \mathbb{N}_0}$ of **approximate solutions**.

Initial guess

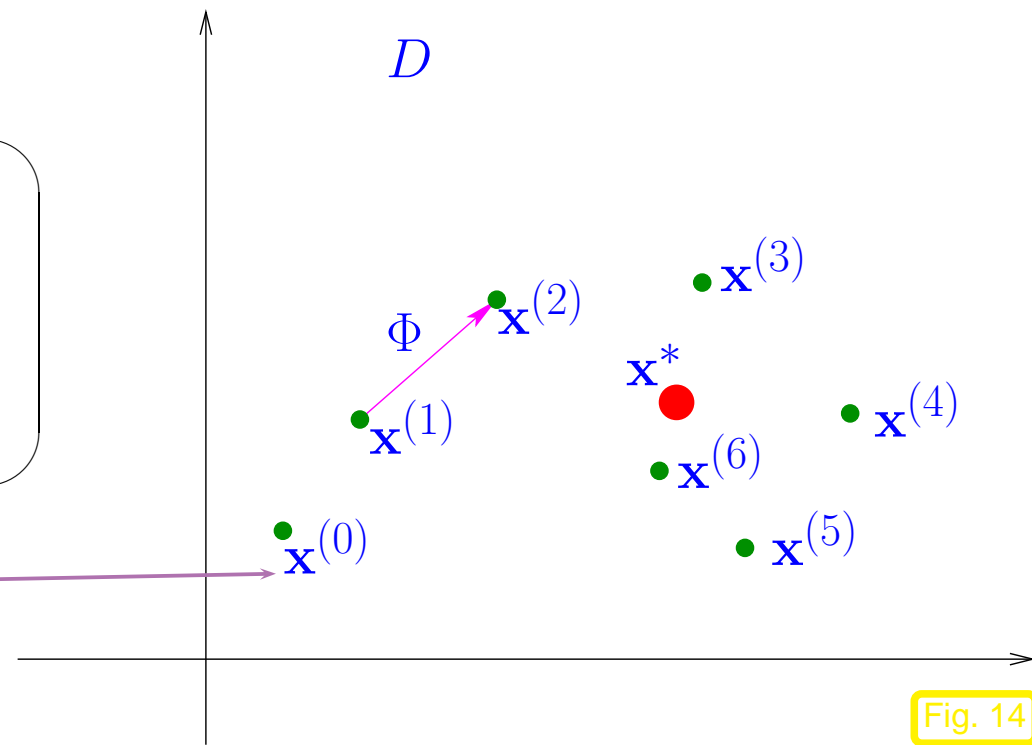


Fig. 14

Fundamental concepts: **convergence** \Rightarrow **speed of convergence**
consistency

- iterate $\mathbf{x}^{(k)}$ depends on F and (one or several) $\mathbf{x}^{(n)}$, $n < k$, e.g.,

$$\mathbf{x}^{(k)} = \underbrace{\Phi_F(\mathbf{x}^{(k-1)}, \dots, \mathbf{x}^{(k-m)})}_{\text{iteration function for } m\text{-point method}} \quad (1.1.1)$$

- $\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(m-1)}$ = **initial guess(es)** (ger.: Anfangsnäherung)

Definition 1.1.1 (Convergence of iterative methods).

An iterative method **converges** (for fixed initial guess(es)) $:\Leftrightarrow \mathbf{x}^{(k)} \rightarrow \mathbf{x}^*$ and $F(\mathbf{x}^*) = 0$.

Definition 1.1.2 (Consistency of iterative methods).

An iterative method is **consistent** with $F(\mathbf{x}) = 0$

$$:\Leftrightarrow \Phi_F(\mathbf{x}^*, \dots, \mathbf{x}^*) = \mathbf{x}^* \quad \Leftrightarrow \quad F(\mathbf{x}^*) = 0$$

Terminology: **error** of iterates $\mathbf{x}^{(k)}$ is defined as: $\mathbf{e}^{(k)} := \mathbf{x}^{(k)} - \mathbf{x}^*$

Definition 1.1.3 (Local and global convergence).

An iterative method **converges locally** to $\mathbf{x}^* \in \mathbb{R}^n$, if there is a neighborhood $U \subset D$ of \mathbf{x}^* , such that

$$\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(m-1)} \in U \Rightarrow \mathbf{x}^{(k)} \text{ well defined} \quad \wedge \quad \lim_{k \rightarrow \infty} \mathbf{x}^{(k)} = \mathbf{x}^*$$

for the sequences $(\mathbf{x}^{(k)})_{k \in \mathbb{N}_0}$ of iterates.

If $U = D$, the iterative method is **globally convergent**.

local convergence



(Only initial guesses “sufficiently close” to \mathbf{x}^* guarantee convergence.)

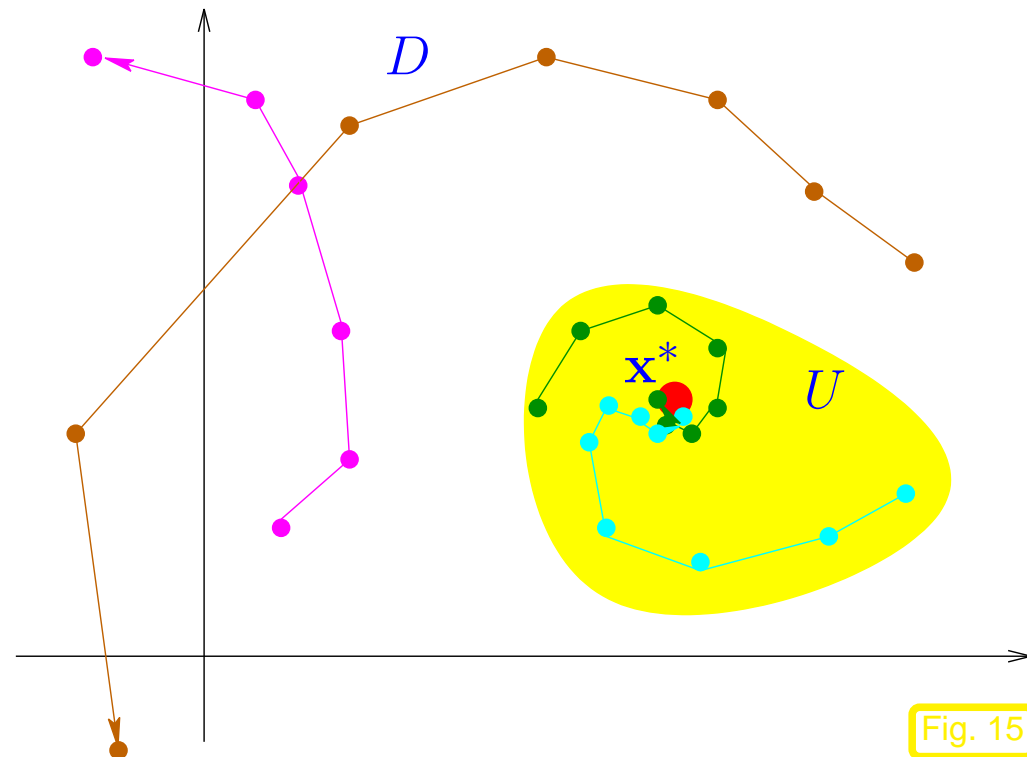


Fig. 15

Goal: Find iterative methods that converge (locally) to a solution of $F(\mathbf{x}) = 0$.

Two general questions: How to measure the speed of convergence?
When to terminate the iteration?

1.1.1 Speed of convergence

Here and in the sequel, $\|\cdot\|$ designates a generic vector norm, see Def. 1.1.9. Any occurring matrix norm is induced by this vector norm, see Def. 1.1.12.

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. 1.1.9) of iteration error

Definition 1.1.4 (Linear convergence).

A sequence $\mathbf{x}^{(k)}$, $k = 0, 1, 2, \dots$, in \mathbb{R}^n *converges linearly* to $\mathbf{x}^* \in \mathbb{R}^n$, if

$$\exists L < 1: \quad \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^* \right\| \leq L \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| \quad \forall k \in \mathbb{N}_0 .$$

Terminology: least upper bound for L gives the *rate of convergence*

Remark 1.1.2 (Impact of choice of norm).

<i>Fact of convergence</i> of iteration is	<i>independent</i>	of choice of norm
<i>Fact of linear convergence</i>	<i>depends</i>	on choice of norm
<i>Rate of linear convergence</i>	<i>depends</i>	on choice of norm

]

Norms provide tools for measuring errors. Recall from linear algebra and calculus:

Definition 1.1.9 (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	numpy.linalg 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>
∞ -norm, max norm	:	$\ \mathbf{x}\ _\infty := \max\{ x_1 , \dots, x_n \}$	<code>norm(x, inf)</code>

Recall: equivalence of all norms on finite dimensional vector space \mathbb{K}^n :

Definition 1.1.10 (Equivalence of norms).

Two norms $\|\cdot\|_1$ and $\|\cdot\|_2$ on a vector space V are equivalent if

$$\exists \underline{C}, \overline{C} > 0: \quad \underline{C} \|v\|_1 \leq \|v\|_2 \leq \overline{C} \|v\|_1 \quad \forall v \in V.$$

Theorem 1.1.11 (Equivalence of all norms on finite dimensional vector spaces).

If $\dim V < \infty$ all norms (\rightarrow Def. 1.1.9) on V are equivalent (\rightarrow Def. 1.1.10).



Simple explicit norm equivalences: for all $\mathbf{x} \in \mathbb{K}^n$

$$\|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1 \leq \sqrt{n} \|\mathbf{x}\|_2, \quad (1.1.7)$$

$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \sqrt{n} \|\mathbf{x}\|_\infty, \quad (1.1.8)$$

$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_1 \leq n \|\mathbf{x}\|_\infty. \quad (1.1.9)$$

Gradinaru
D-MATH

Definition 1.1.12 (Matrix norm).

Given a vector norm $\|\cdot\|$ on \mathbb{R}^n , the associated **matrix norm** is defined by

$$\mathbf{M} \in \mathbb{R}^{m,n}: \quad \|\mathbf{M}\| := \sup_{\mathbf{x} \in \mathbb{R}^n \setminus \{0\}} \frac{\|\mathbf{M}\mathbf{x}\|}{\|\mathbf{x}\|}.$$

sub-multiplicative:

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



notation:

$$\|\mathbf{x}\|_2 \rightarrow \|\mathbf{M}\|_2, \quad \|\mathbf{x}\|_1 \rightarrow \|\mathbf{M}\|_1, \quad \|\mathbf{x}\|_\infty \rightarrow \|\mathbf{M}\|_\infty$$

Example 1.1.4 (Matrix norm associated with ∞ -norm and 1-norm).

$$\begin{aligned} \text{e.g. for } \mathbf{M} \in \mathbb{K}^{2,2}: \quad \|\mathbf{M}\mathbf{x}\|_\infty &= \max\{|m_{11}x_1 + m_{12}x_2|, |m_{21}x_1 + m_{22}x_2|\} \\ &\leq \max\{|m_{11}| + |m_{12}|, |m_{21}| + |m_{22}|\} \|\mathbf{x}\|_\infty, \\ \|\mathbf{M}\mathbf{x}\|_1 &= |m_{11}x_1 + m_{12}x_2| + |m_{21}x_1 + m_{22}x_2| \\ &\leq \max\{|m_{11}| + |m_{21}|, |m_{12}| + |m_{22}|\} (|x_1| + |x_2|). \end{aligned}$$

For general $\mathbf{M} \in \mathbb{K}^{m,n}$

$$\text{➤ matrix norm } \leftrightarrow \|\cdot\|_\infty = \text{row sum norm} \quad \|\mathbf{M}\|_\infty := \max_{i=1,\dots,m} \sum_{j=1}^n |m_{ij}|, \quad (1.1.10)$$

$$\text{➤ matrix norm } \leftrightarrow \|\cdot\|_1 = \text{column sum norm} \quad \|\mathbf{M}\|_1 := \max_{j=1,\dots,n} \sum_{i=1}^m |m_{ij}|. \quad (1.1.11)$$

Remark 1.1.5 (Seeing linear convergence).

norms of iteration errors

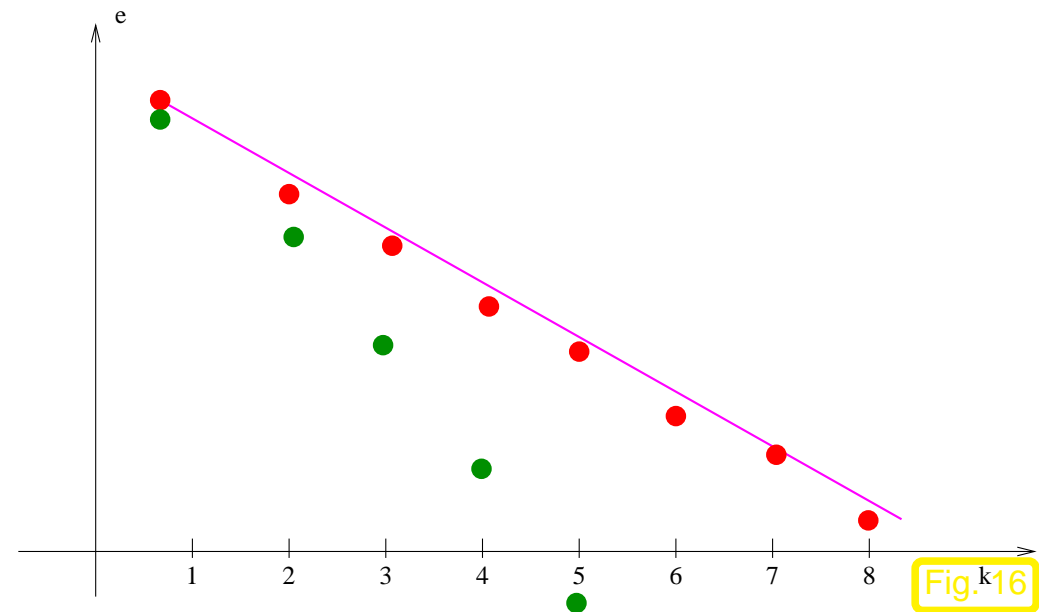


~ straight line in **lin-log** plot

$$\left\| \mathbf{e}^{(k)} \right\| \leq L^k \left\| \mathbf{e}^{(0)} \right\| ,$$

$$\log(\left\| \mathbf{e}^{(k)} \right\|) \leq k \log(L) + \log(\left\| \mathbf{e}^{(0)} \right\|) .$$

(●: Any “faster” convergence also qualifies as linear !)



Let us abbreviate the error norm in step k by $\epsilon_k := \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|$. In the case of linear convergence (see Def. 1.1.4) assume (with $0 < L < 1$)

$$\epsilon_{k+1} \approx L \epsilon_k \Rightarrow \log \epsilon_{k+1} \approx \log L + \log \epsilon_k \Rightarrow \log \epsilon_{k+1} \approx k \log L + \log \epsilon_0 . \quad (1.1.12)$$

We conclude that $\log L < 0$ describes slope of graph in lin-log error chart.



Example 1.1.6 (Linearly convergent iteration).

Iteration (dimension $n = 1$):

$$x^{(k+1)} = x^{(k)} + \frac{\cos x^{(k)} + 1}{\sin x^{(k)}}.$$

Code 1.1.7: simple fixed point iteration

```
1 def lincvg(x):
2     y = []
3     for k in xrange(15):
4         x = x +(cos(x)+1)/sin(x)
5         y += [x]
6     err = array(y) - x
7     rate = err[1:]/err[: -1]
8     return err, rate
```

Note: $x^{(15)}$ 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

Linear convergence as in Def. 1.1.4



error graphs = straight lines in lin-log scale

→ Rem. 1.1.5

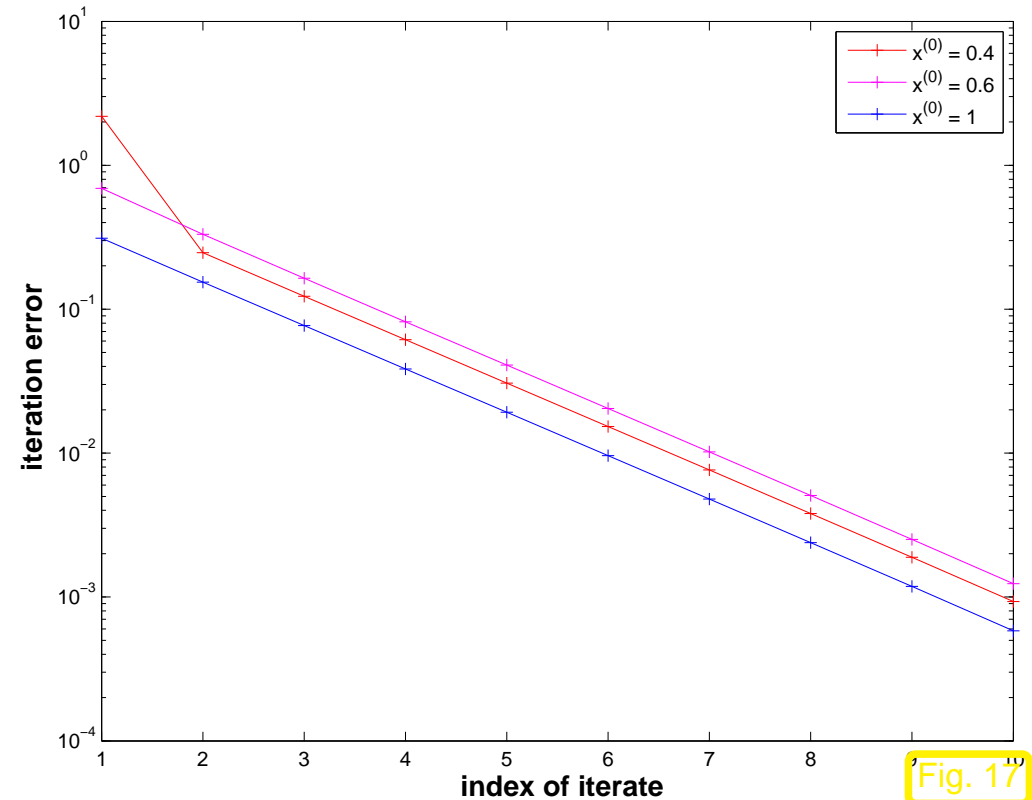


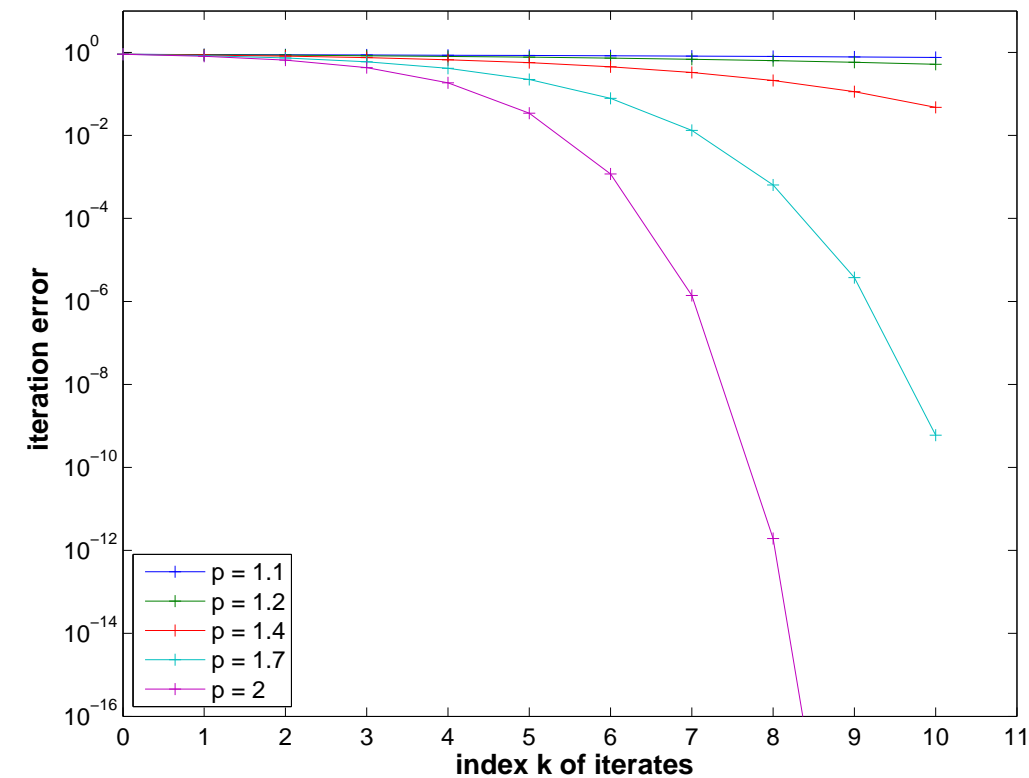
Fig. 17

Definition 1.1.13 (Order of convergence).

A **convergent** sequence $\mathbf{x}^{(k)}$, $k = 0, 1, 2, \dots$, in \mathbb{R}^n converges with **order** p to $\mathbf{x}^* \in \mathbb{R}^n$, if

$$\exists C > 0: \quad \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^* \right\| \leq C \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|^p \quad \forall k \in \mathbb{N}_0,$$

with $C < 1$ for $p = 1$ (linear convergence → Def. 1.1.4)



◁ Qualitative error graphs for convergence of order p (lin-log scale)

In the case of convergence of order p ($p > 1$) (see Def. 1.1.13):

$$\begin{aligned} \epsilon_{k+1} \approx C \epsilon_k^p &\Rightarrow \log \epsilon_{k+1} = \log C + p \log \epsilon_k \Rightarrow \log \epsilon_{k+1} = \log C \sum_{l=0}^k p^l + p^{k+1} \log \epsilon_0 \\ &\Rightarrow \log \epsilon_{k+1} = -\frac{\log C}{p-1} + \left(\frac{\log C}{p-1} + \log \epsilon_0 \right) p^{k+1}. \end{aligned}$$

In this case, the error graph is a concave power curve (for sufficiently small ϵ_0 !)

Example 1.1.8 (quadratic convergence). (= convergence of order 2)

Iteration for computing \sqrt{a} , $a > 0$:

$$x^{(k+1)} = \frac{1}{2}\left(x^{(k)} + \frac{a}{x^{(k)}}\right) \Rightarrow |x^{(k+1)} - \sqrt{a}| = \frac{1}{2x^{(k)}}|x^{(k)} - \sqrt{a}|^2. \quad (1.1.13)$$

By the arithmetic-geometric mean inequality (AGM) $\sqrt{ab} \leq \frac{1}{2}(a + b)$ we conclude: $x^{(k)} > \sqrt{a}$ for $k \geq 1$.

\Rightarrow sequence from (1.1.13) 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)

How to guess the order of convergence in a numerical experiment?

Abbreviate $\epsilon_k := \|\mathbf{x}^{(k)} - \mathbf{x}^*\|$ and then

$$\epsilon_{k+1} \approx C \epsilon_k^p \Rightarrow \log \epsilon_{k+1} \approx \log C + p \log \epsilon_k \Rightarrow \frac{\log \epsilon_{k+1} - \log \epsilon_k}{\log \epsilon_k - \log \epsilon_{k-1}} \approx p.$$

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	2.000000000000000000	0.58578643762690485	
1	1.500000000000000000	0.08578643762690485	
2	1.416666666666666652	0.00245310429357137	1.850
3	1.41421568627450966	0.00000212390141452	1.984
4	1.41421356237468987	0.00000000000159472	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 convergent second-order iteration:

Indeed, denoting the relative error in step k by δ_k , we have:

$$\begin{aligned}
 x^{(k)} &= x^*(1 + \delta_k) \Rightarrow x^{(k)} - x^* = \delta_k x^* . \\
 \Rightarrow |x^* \delta_{k+1}| &= |x^{(k+1)} - x^*| \leq C |x^{(k)} - x^*|^2 = C |x^* \delta_k|^2 \\
 &\Rightarrow |\delta_{k+1}| \leq C |x^*| \delta_k^2 .
 \end{aligned} \tag{1.1.14}$$

Note: $\delta_k \approx 10^{-\ell}$ means that $\mathbf{x}^{(k)}$ has ℓ significant digits.

Also note that if $C \approx 1$, then $\delta_k = 10^{-\ell}$ and (1.1.8) implies $\delta_{k+1} \approx 10^{-2\ell}$.

1.1.2 Termination criteria

Usually (even without roundoff errors) the iteration will never arrive at an/the exact solution \mathbf{x}^* after finitely many steps. Thus, we can only hope to compute an *approximate* solution by accepting $\mathbf{x}^{(K)}$ as result for some $K \in \mathbb{N}_0$. Termination criteria (*ger.*: Abbruchbedingungen) are used to determine a suitable value for K .

For the sake of efficiency: \triangleright stop iteration when iteration error is just “small enough”

“small enough” depends on concrete setting:

Usual goal:
$$\left\| \mathbf{x}^{(K)} - \mathbf{x}^* \right\| \leq \tau, \quad \tau \hat{=} \text{prescribed tolerance.}$$

Ideal:
$$K = \operatorname{argmin}\{k \in \mathbb{N}_0: \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| < \tau\} . \quad (1.1.15)$$

- ① **A priori termination:** stop iteration after fixed number of steps (possibly depending on $\mathbf{x}^{(0)}$).



Drawback: hardly ever possible !

Alternative:

A posteriori termination criteria

use already computed iterates to decide when to stop

- ② Reliable termination: stop iteration $\{\mathbf{x}^{(k)}\}_{k \in \mathbb{N}_0}$ with limit \mathbf{x}^* , when

$$\left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| \leq \tau, \quad \tau \hat{=} \text{prescribed tolerance} > 0. \quad (1.1.16)$$

Gradinaru
D-MATH



\mathbf{x}^* not known !

Invoking additional properties of either the non-linear system of equations $F(\mathbf{x}) = 0$ or the iteration it is sometimes possible to tell that for sure $\left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| \leq \tau$ for all $k \geq K$, though this K may be (significantly) larger than the optimal termination index from (1.1.15), see Rem. 1.1.10.

Code 1.1.9: stationary iteration

```

1 from numpy import sqrt, array
2 def sqrtit(a,x):
3     exact = sqrt(a)
4     e = [x]
5     x_old = -1.
6     while x_old != x:
7         x_old = x
8         x = 0.5*(x+a/x)
9         e += [x]
10    e = array(e)
11    e = abs(e-exact)
12    return e
13
14 e = sqrtit(2.,1.)
15 print e

```

Gradinaru
D-MATH

③ use that M is finite! (\rightarrow Sect. ??)

➤ possible to wait until (convergent) iteration becomes stationary



possibly grossly inefficient!
(always computes “up to
machine precision”)

④ **Residual based** termination: stop convergent iteration $\{\mathbf{x}^{(k)}\}_{k \in \mathbb{N}_0}$, when

$$\left\| F(\mathbf{x}^{(k)}) \right\| \leq \tau, \quad \tau \hat{=} \text{prescribed tolerance} > 0.$$



no guaranteed accuracy

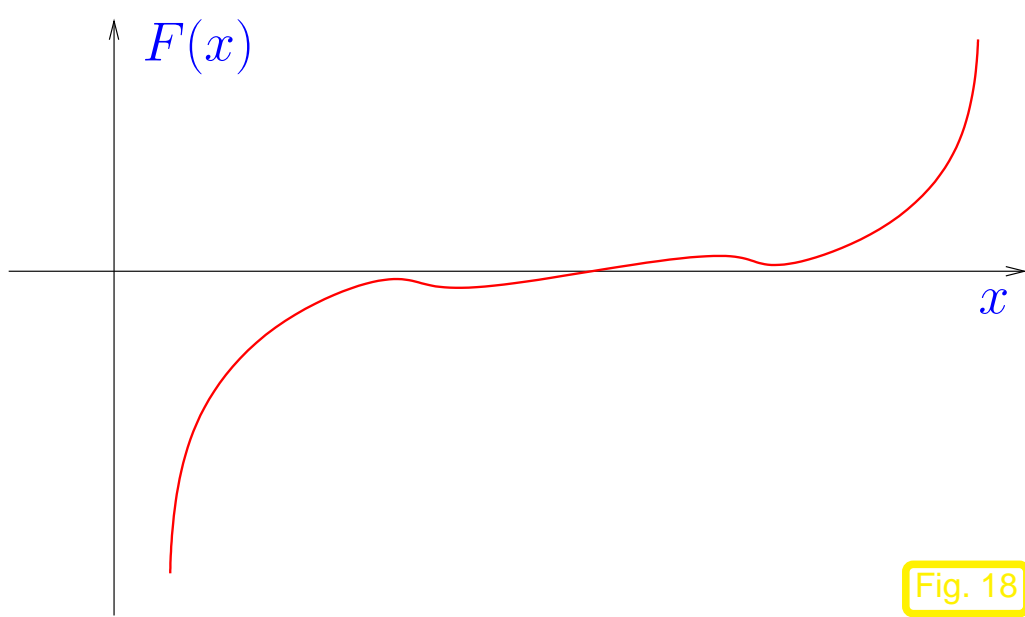


Fig. 18

$$\|F(\mathbf{x}^{(k)})\| \text{ small } \not\Rightarrow |x - x^*| \text{ small}$$

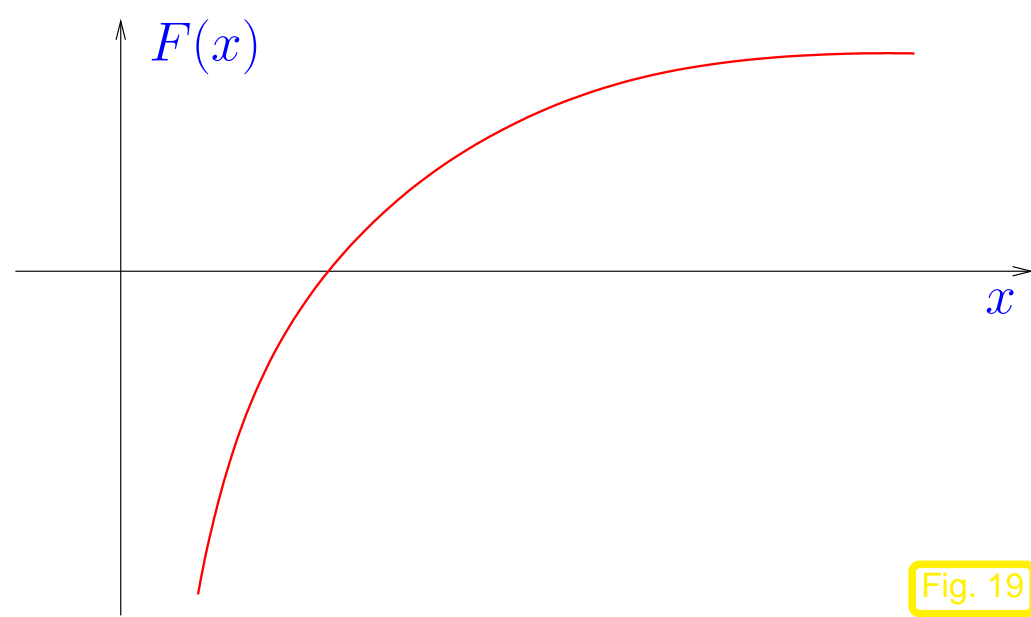


Fig. 19

$$\|F(\mathbf{x}^{(k)})\| \text{ small } \Rightarrow |x - x^*| \text{ small}$$

Sometimes extra knowledge about the type/speed of convergence allows to achieve **reliable termination** in the sense that (1.1.16) can be guaranteed though the number of iterations might be (slightly) too large.

Remark 1.1.10 (A posteriori termination criterion for linearly convergent iterations).

Known: iteration linearly convergent with rate of convergence $0 < L < 1$:

Derivation of a posteriori termination criterion for linearly convergent iterations with rate of convergence $0 < L < 1$:

$$\left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| \stackrel{\Delta\text{-inequ.}}{\leq} \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| + \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^* \right\| \leq \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| + L \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| .$$

Iterates satisfy:

$$\left\| \mathbf{x}^{(k+1)} - \mathbf{x}^* \right\| \leq \frac{L}{1-L} \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| . \quad (1.1.17)$$

This suggests that we take the right hand side of (1.1.17) as a posteriori error bound.



Example 1.1.11 (A posteriori error bound for linearly convergent iteration).

Iteration of Example 1.1.6:

$$x^{(k+1)} = x^{(k)} + \frac{\cos x^{(k)} + 1}{\sin x^{(k)}} \Rightarrow x^{(k)} \rightarrow \pi \quad \text{for } x^{(0)} \text{ close to } \pi .$$

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	0.0000000000000667
15	0.000029963440745	0.000029963440563	0.0000000000000181

Hence: the a posteriori error bound is highly accurate in this case!



Note: If L not known then using $\tilde{L} > L$ in error bound is playing safe.

1.2 Fixed Point Iterations

Non-linear system of equations $F(\mathbf{x}) = 0$, $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$,

A **fixed point iteration** is defined by **iteration function** $\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$:

iteration function $\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$
initial guess $\mathbf{x}^{(0)} \in U$ \Rightarrow iterates $(\mathbf{x}^{(k)})_{k \in \mathbb{N}_0} : \mathbf{x}^{(k+1)} := \Phi(\mathbf{x}^{(k)})$.
 \rightarrow 1-point method, cf. (1.1.1)

Sequence of iterates need not be well defined: $\mathbf{x}^{(k)} \notin U$ possible !

1.2.1 Consistent fixed point iterations

Definition 1.2.1 (Consistency of fixed point iterations, *c.f.* Def. 1.1.2).

A fixed point iteration $\mathbf{x}^{(k+1)} = \Phi(\mathbf{x}^{(k)})$ is **consistent** with $F(\mathbf{x}) = 0$, if

$$F(\mathbf{x}) = 0 \quad \text{and} \quad \mathbf{x} \in U \cap D \quad \Leftrightarrow \quad \Phi(\mathbf{x}) = \mathbf{x} .$$

Note: Φ continuous & fixed point iteration (locally) convergent to \mathbf{x}^* then \mathbf{x}^* is **fixed point** of iteration function Φ .

General construction of fixed point iterations that is consistent with $F(\mathbf{x}) = 0$:

rewrite $F(\mathbf{x}) = 0 \Leftrightarrow \Phi(\mathbf{x}) = \mathbf{x}$ and then

$$\text{use the fixed point iteration} \quad \mathbf{x}^{(k+1)} := \Phi(\mathbf{x}^{(k)}) . \quad (1.2.1)$$

Note: there are *many* ways to transform $F(\mathbf{x}) = 0$ into a fixed point form !

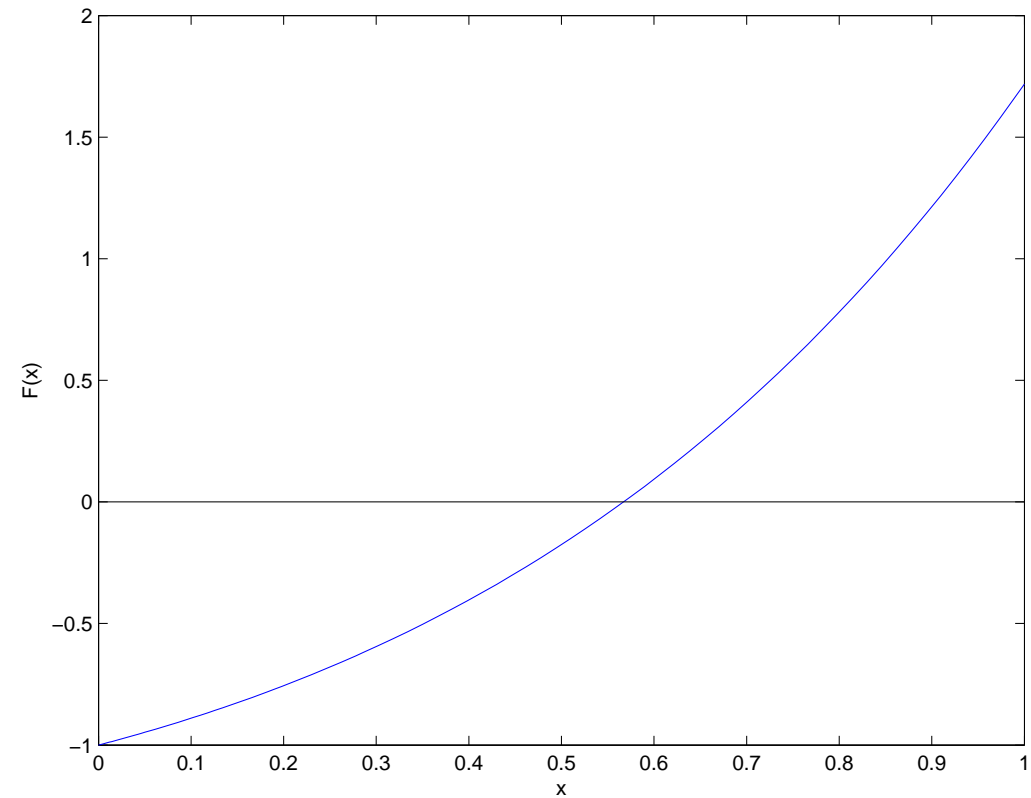
$$F(x) = xe^x - 1, \quad x \in [0, 1].$$

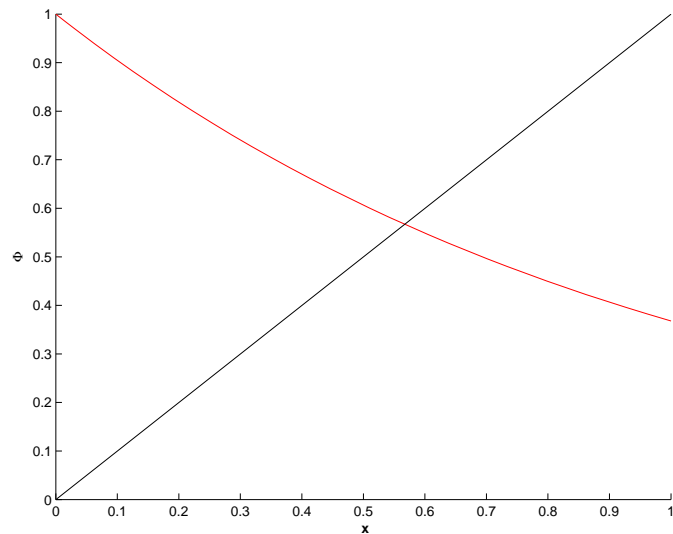
Different fixed point forms:

$$\Phi_1(x) = e^{-x},$$

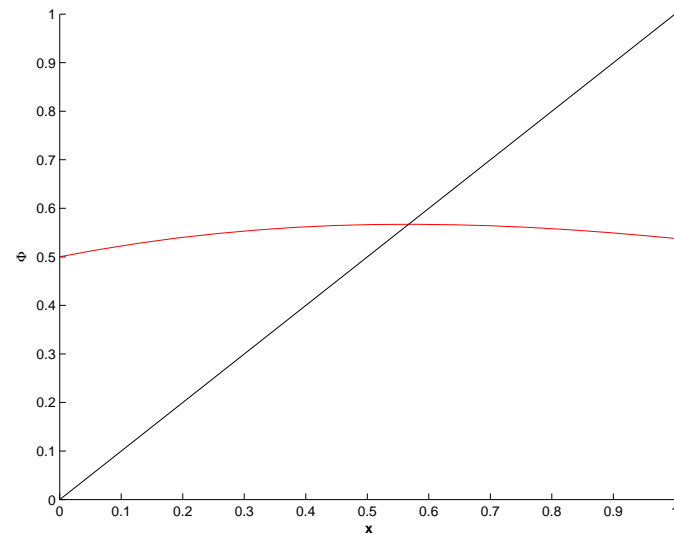
$$\Phi_2(x) = \frac{1+x}{1+e^x},$$

$$\Phi_3(x) = x + 1 - xe^x.$$

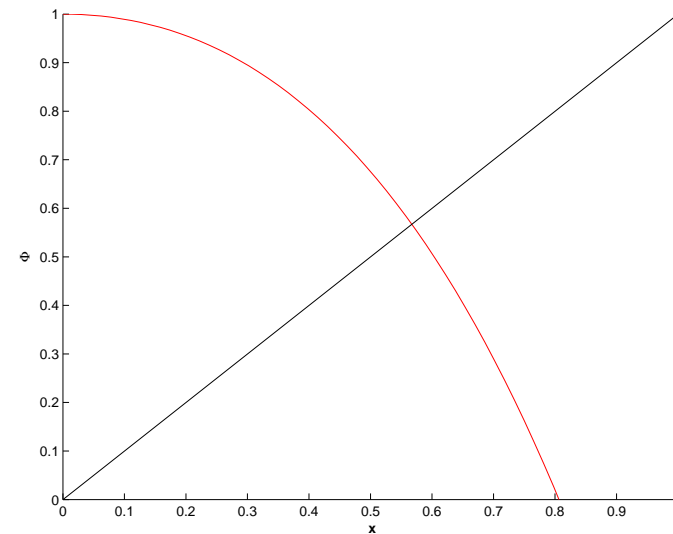




function Φ_1



function Φ_2



function Φ_3

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

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	0.0000000000000003	0.288178118764323
4	0.007078662470882	0.0000000000000000	0.723649245792953
5	0.004028858567431	0.0000000000000000	0.410183132337935
6	0.002280343429460	0.0000000000000000	1.186907542305364
7	0.001294757160282	0.0000000000000000	0.146569797006362
8	0.000733837662863	0.0000000000000000	0.310516641279937
9	0.000416343852458	0.0000000000000000	0.357777386500765
10	0.000236077474313	0.0000000000000000	0.974565695952037

Observed: linear convergence of $x_1^{(k)}$, quadratic convergence of $x_2^{(k)}$,
no convergence (erratic behavior) of $x_3^{(k)}$, $x_i^{(0)} = 0.5$.



Question: can we explain/forecast the behaviour of the iteration?

1.2.2 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.

Definition 1.2.2 (Contractive mapping).

$\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$ is **contractive** (w.r.t. norm $\|\cdot\|$ on \mathbb{R}^n), if

$$\exists L < 1: \quad \|\Phi(\mathbf{x}) - \Phi(\mathbf{y})\| \leq L \|\mathbf{x} - \mathbf{y}\| \quad \forall \mathbf{x}, \mathbf{y} \in U. \quad (1.2.2)$$

A simple consideration: if $\Phi(\mathbf{x}^*) = \mathbf{x}^*$ (fixed point), then a fixed point iteration induced by a contractive mapping Φ satisfies

$$\left\| \mathbf{x}^{(k+1)} - \mathbf{x}^* \right\| = \left\| \Phi(\mathbf{x}^{(k)}) - \Phi(\mathbf{x}^*) \right\| \stackrel{(1.2.2)}{\leq} L \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|,$$

that is, the iteration **converges** (at least) **linearly** (\rightarrow Def. 1.1.4).

Remark 1.2.2 (Banach's fixed point theorem). \rightarrow [52, Satz 6.5.2]

A key theorem in calculus (also functional analysis):

Theorem 1.2.3 (Banach's fixed point theorem).

If $D \subset \mathbb{K}^n$ ($\mathbb{K} = \mathbb{R}, \mathbb{C}$) closed and $\Phi : D \mapsto D$ satisfies

$$\exists L < 1: \quad \|\Phi(\mathbf{x}) - \Phi(\mathbf{y})\| \leq L \|\mathbf{x} - \mathbf{y}\| \quad \forall \mathbf{x}, \mathbf{y} \in D,$$

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 $\mathbf{x}^{(k)} = \Phi(\mathbf{x}^{(k-1)})$, $\mathbf{x}^{(0)} \in D$:

$$\begin{aligned} \left\| \mathbf{x}^{(k+N)} - \mathbf{x}^{(k)} \right\| &\leq \sum_{j=k}^{k+N-1} \left\| \mathbf{x}^{(j+1)} - \mathbf{x}^{(j)} \right\| \leq \sum_{j=k}^{k+N-1} L^j \left\| \mathbf{x}^{(1)} - \mathbf{x}^{(0)} \right\| \\ &\leq \frac{L^k}{1-L} \left\| \mathbf{x}^{(1)} - \mathbf{x}^{(0)} \right\| \xrightarrow{k \rightarrow \infty} 0. \end{aligned}$$

$(\mathbf{x}^{(k)})_{k \in \mathbb{N}_0}$ Cauchy sequence \Rightarrow convergent $\mathbf{x}^{(k)} \xrightarrow{k \rightarrow \infty} \mathbf{x}^*$.

Continuity of $\Phi \Rightarrow \Phi(\mathbf{x}^*) = \mathbf{x}^*$. Uniqueness of fixed point is evident.

□

△

Lemma 1.2.4 (Sufficient condition for local linear convergence of fixed point iteration).

If $\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$, $\Phi(\mathbf{x}^*) = \mathbf{x}^*$, Φ differentiable in \mathbf{x}^* , and $\|D\Phi(\mathbf{x}^*)\| < 1$, then the fixed point iteration (1.2.1) converges locally and at least linearly.

matrix norm, Def. 1.1.12 !



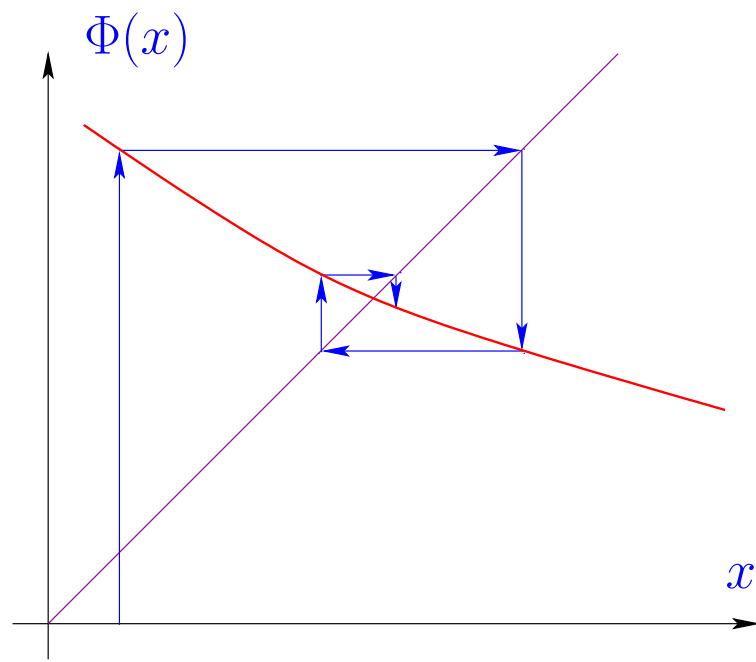
notation: $D\Phi(\mathbf{x}) \hat{=}$ **Jacobian** (ger.: Jacobi-Matrix) of Φ at $\mathbf{x} \in D$

Example 1.2.3 (Fixed point iteration in 1D).

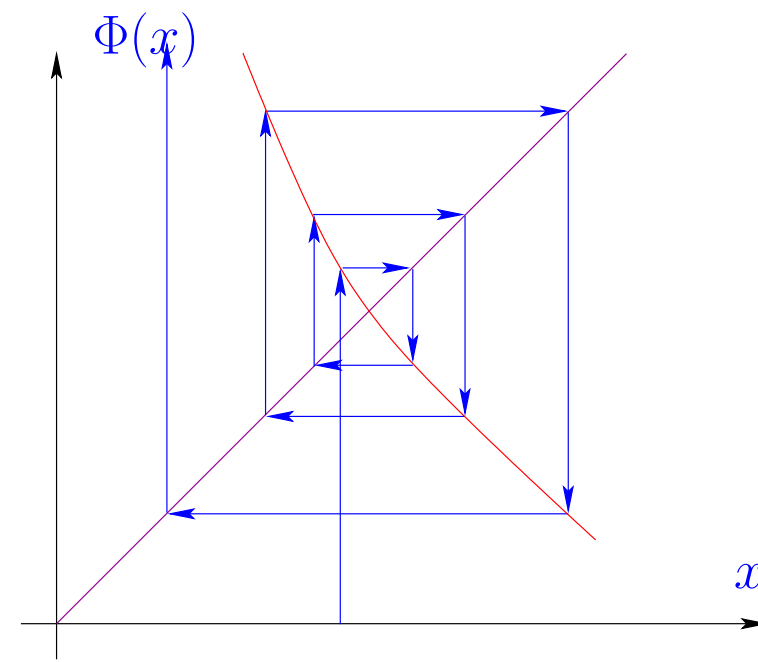
1D setting ($n = 1$): $\Phi : \mathbb{R} \mapsto \mathbb{R}$ continuously differentiable, $\Phi(x^*) = x^*$

fixed point iteration: $x^{(k+1)} = \Phi(x^{(k)})$

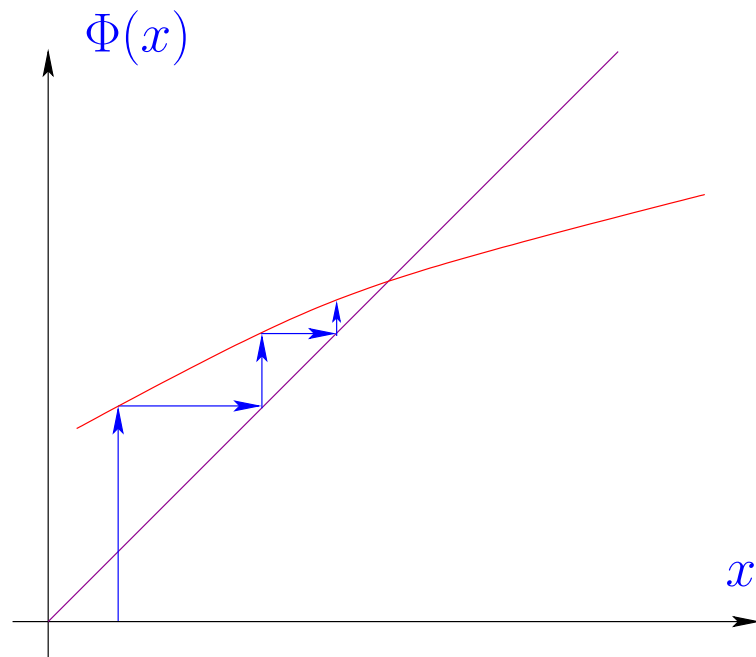
“Visualization” of the statement of Lemma 1.2.4: The iteration converges *locally*, if Φ is flat in a neighborhood of x^* , it will diverge, if Φ is steep there.



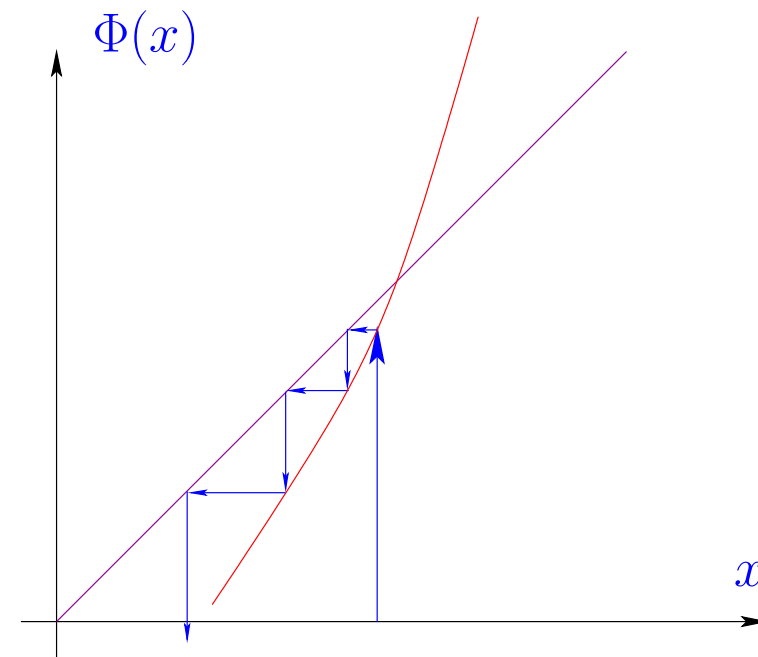
$-1 < \Phi'(x^*) \leq 0 \Rightarrow$ convergence



$\Phi'(x^*) < -1 \Rightarrow$ divergence



$0 \leq \Phi'(x^*) < 1 \Rightarrow$ convergence



$1 < \Phi'(x^*) \Rightarrow$ divergence



Proof. (of Lemma 1.2.4) 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

$$\begin{aligned} \|\Phi(\mathbf{x}) - \mathbf{x}^*\| - \|D\Phi(\mathbf{x}^*)(\mathbf{x} - \mathbf{x}^*)\| &\leq \psi(\|\mathbf{x} - \mathbf{x}^*\|) \|\mathbf{x} - \mathbf{x}^*\| \\ \blacktriangleright \quad \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^* \right\| &\leq (\psi(t) + \|D\Phi(\mathbf{x}^*)\|) \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| \leq L \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| , \end{aligned}$$

if $\left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| < \delta$.

□

Contractivity also guarantees the *uniqueness* of a fixed point, see the next Lemma.

Recalling the Banach fixed point theorem Thm. 1.2.3 we see that under some additional (usually mild) assumptions, it also ensures the *existence* of a fixed point.

Lemma 1.2.5 (Sufficient condition for local linear convergence of fixed point iteration (II)).

Let U be convex and $\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$ continuously differentiable with $L := \sup_{\mathbf{x} \in U} \|D\Phi(\mathbf{x})\| < 1$.

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}^* locally at least linearly.

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 1.2.5) By the mean value theorem

$$\Phi(\mathbf{x}) - \Phi(\mathbf{y}) = \int_0^1 D\Phi(\mathbf{x} + \tau(\mathbf{y} - \mathbf{x}))(\mathbf{y} - \mathbf{x}) d\tau \quad \forall \mathbf{x}, \mathbf{y} \in \text{dom}(\Phi).$$

$$\blacktriangleright \quad \|\Phi(\mathbf{x}) - \Phi(\mathbf{y})\| \leq L \|\mathbf{y} - \mathbf{x}\|.$$

There is $\delta > 0$: $B := \{\mathbf{x} : \|\mathbf{x} - \mathbf{x}^*\| \leq \delta\} \subset \text{dom}(\Phi)$. Φ is contractive on B with unique fixed point \mathbf{x}^* .

Remark 1.2.4.

If $0 < \|D\Phi(\mathbf{x}^*)\| < 1$, $\mathbf{x}^{(k)} \approx \mathbf{x}^*$ then the **asymptotic** rate of linear convergence is $L = \|D\Phi(\mathbf{x}^*)\|$



Example 1.2.5 (Multidimensional fixed point iteration).

$$\begin{array}{ccc} \text{System of equations} & \text{in} & \text{fixed point form:} \\ \left\{ \begin{array}{l} x_1 - c(\cos x_1 - \sin x_2) = 0 \\ (x_1 - x_2) - c \sin x_2 = 0 \end{array} \right. & \Rightarrow & \left\{ \begin{array}{l} c(\cos x_1 - \sin x_2) = x_1 \\ c(\cos x_1 - 2 \sin x_2) = x_2 \end{array} \right. . \end{array}$$

$$\text{Define: } \Phi \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = c \begin{pmatrix} \cos x_1 - \sin x_2 \\ \cos x_1 - 2 \sin x_2 \end{pmatrix} \Rightarrow D\Phi \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = -c \begin{pmatrix} \sin x_1 & \cos x_2 \\ \sin x_1 & 2 \cos x_2 \end{pmatrix} .$$

Choose *appropriate* norm: $\|\cdot\| = \infty\text{-norm } \|\cdot\|_\infty$ (\rightarrow Example 1.1.4) ;

$$\text{if } c < \frac{1}{3} \Rightarrow \|D\Phi(\mathbf{x})\|_\infty < 1 \quad \forall \mathbf{x} \in \mathbb{R}^2 ,$$

➤ (at least) linear convergence of the fixed point iteration.

The existence of a fixed point is also guaranteed, because Φ maps into the closed set $[-3, 3]^2$. Thus, the Banach fixed point theorem, Thm. 1.2.3, can be applied.



What about higher order convergence (\rightarrow Def. 1.1.13) ?

Refined convergence analysis for $n = 1$ (scalar case, $\Phi : \text{dom}(\Phi) \subset \mathbb{R} \mapsto \mathbb{R}$):

Theorem 1.2.6 (Taylor's formula). $\rightarrow [52, \text{Sect. 5.5}]$

If $\Phi : U \subset \mathbb{R} \mapsto \mathbb{R}$, U interval, is $m + 1$ times continuously differentiable, $x \in U$

$$\Phi(y) - \Phi(x) = \sum_{k=1}^m \frac{1}{k!} \Phi^{(k)}(x)(y - x)^k + O(|y - x|^{m+1}) \quad \forall y \in U. \quad (1.2.3)$$

Apply Taylor expansion (1.2.3) to iteration function Φ :

If $\Phi(x^*) = x^*$ and $\Phi : \text{dom}(\Phi) \subset \mathbb{R} \mapsto \mathbb{R}$ is “sufficiently smooth”

$$x^{(k+1)} - x^* = \Phi(x^{(k)}) - \Phi(x^*) = \sum_{l=1}^m \frac{1}{l!} \Phi^{(l)}(x^*)(x^{(k)} - x^*)^l + O(|x^{(k)} - x^*|^{m+1}). \quad (1.2.4)$$

Lemma 1.2.7 (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 (1.2.1) converges locally to x^* with **order** $\geq m + 1$ (\rightarrow Def. 1.1.13).

Proof. For neighborhood \mathcal{U} of x^*

$$(1.2.4) \Rightarrow \exists C > 0: |\Phi(y) - \Phi(x^*)| \leq C |y - x^*|^{m+1} \quad \forall y \in \mathcal{U} .$$

$$\delta^m C < 1/2 : \quad |x^{(0)} - x^*| < \delta \Rightarrow |x^{(k)} - x^*| < 2^{-k} \delta \quad \supset \quad \text{local convergence} .$$

Then appeal to (1.2.4)

□

Example 1.2.1 continued:

$$\Phi_2'(x) = \frac{1 - xe^x}{(1 + e^x)^2} = 0 \quad , \text{ if } \quad xe^x - 1 = 0 \quad \text{hence quadratic convergence !} .$$

Example 1.2.1 continued: Since $x^*e^{x^*} - 1 = 0$

$$\Phi_1'(x) = -e^{-x} \Rightarrow \Phi_1'(x^*) = -x^* \approx -0.56 \quad \text{hence local linear convergence} .$$

$$\Phi_3'(x) = 1 - xe^x - e^x \Rightarrow \Phi_3'(x^*) = -\frac{1}{x^*} \approx -1.79 \quad \text{hence no convergence} .$$

Remark 1.2.6 (Termination criterion for contractive fixed point iteration).

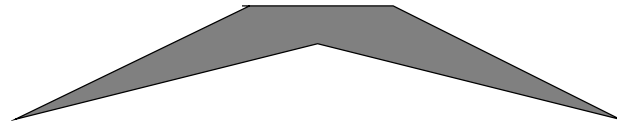
Recap of Rem. 1.1.10:

Termination criterion for contractive fixed point iteration, c.f. (1.2.3), with contraction factor $0 \leq L < 1$:

$$\begin{aligned} \left\| \mathbf{x}^{(k+m)} - \mathbf{x}^{(k)} \right\| &\stackrel{\Delta\text{-ineq.}}{\leq} \sum_{j=k}^{k+m-1} \left\| \mathbf{x}^{(j+1)} - \mathbf{x}^{(j)} \right\| \leq \sum_{j=k}^{k+m-1} L^{j-k} \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| \\ &= \frac{1 - L^m}{1 - L} \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| \leq \frac{1 - L^m}{1 - L} L^{k-l} \left\| \mathbf{x}^{(l+1)} - \mathbf{x}^{(l)} \right\| . \end{aligned}$$

hence for $m \rightarrow \infty$, with $\mathbf{x}^* := \lim_{k \rightarrow \infty} \mathbf{x}^{(k)}$:

$$\left\| \mathbf{x}^* - \mathbf{x}^{(k)} \right\| \leq \frac{L^{k-l}}{1 - L} \left\| \mathbf{x}^{(l+1)} - \mathbf{x}^{(l)} \right\| . \quad (1.2.5)$$



Set $l = 0$ in (1.2.5)

a priori termination criterion

$$\left\| \mathbf{x}^* - \mathbf{x}^{(k)} \right\| \leq \frac{L^k}{1 - L} \left\| \mathbf{x}^{(1)} - \mathbf{x}^{(0)} \right\| \quad (1.2.6)$$

Set $l = k - 1$ in (1.2.5)

a posteriori termination criterion

$$\left\| \mathbf{x}^* - \mathbf{x}^{(k)} \right\| \leq \frac{L}{1 - L} \left\| \mathbf{x}^{(k)} - \mathbf{x}^{(k-1)} \right\| \quad (1.2.7)$$



1.3 Zero Finding

Now, focus on scalar case $n = 1$: $F : I \subset \mathbb{R} \mapsto \mathbb{R}$ **continuous**, I interval

Sought:

$$x^* \in I: \quad F(x^*) = 0$$

1.3.1 Bisection

Idea: use ordering of real numbers & intermediate value theorem

Input: $a, b \in I$ such that $F(a)F(b) < 0$
(different signs !)

$\Rightarrow \exists x^* \in]\min\{a, b\}, \max\{a, b\}[$:
 $F(x^*) = 0$.

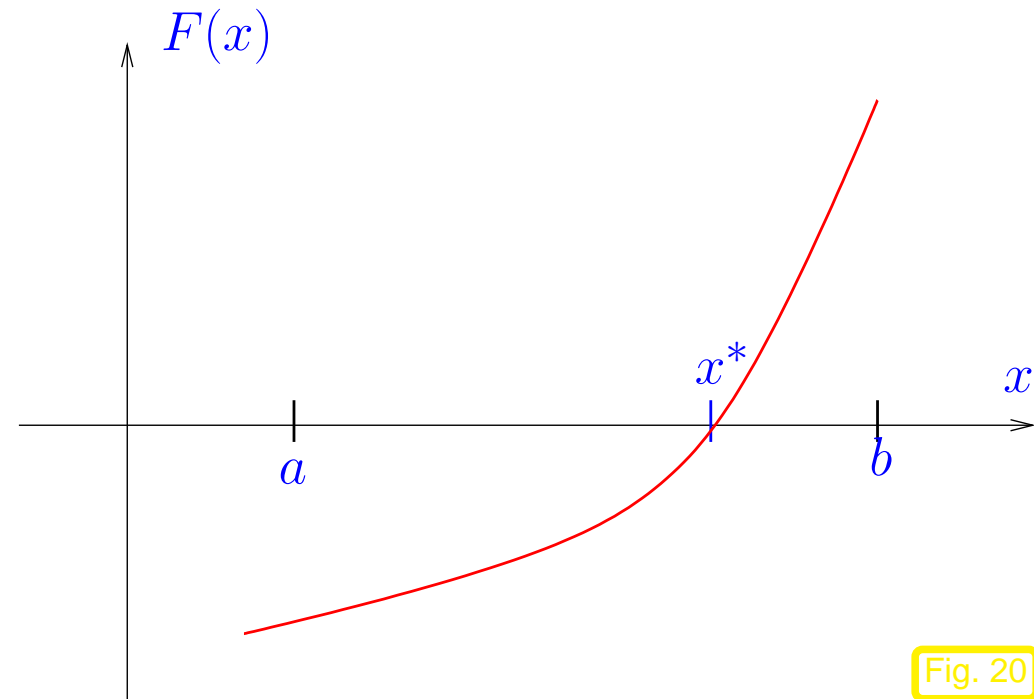


Fig. 20

Algorithm 1.3.1 (Bisection method).

Code 1.3.2: Bisection method

```
1 def mybisect(f, a, b, tol=1e-12):  
2     if a > b:  
3         t = b; a = b; b = t  
4     fa = f(a); fb = f(b)  
5     if fa*fb > 0: raise ValueError
```

```
6 v = 1
7 if fa > 0: v = -1
8 x = 0.5*(a+b)
9 k = 1
10 while (b-a > tol) and (a < x) and (x < b):
11     if v*f(x) > 0: b = x
12     else: a = x
13     x = 0.5*(a+b)
14     k += 1
15 return x, k
16
17 if __name__ == '__main__':
18     f = lambda x: x**3 - 2*x**2 + 4.*x/3. - 8./27.
19     x, k = mybisection(f, 0, 1)
20     print 'x_bisection =', x, 'after k =', k, 'iterations'
21     from scipy.optimize import fsolve, bisection
22     x = fsolve(f, 0, full_output=True)
23     print x
24     print bisection(f, 0, 1)
```



Advantages:

- “foolproof”
- requires only F evaluations

Merely linear convergence: $|x^{(k)} - x^*| \leq 2^{-k}|b - a|$



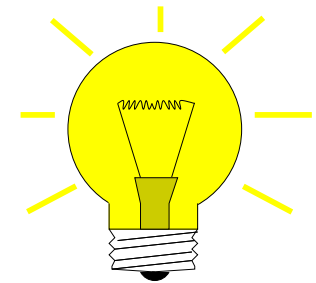
Drawbacks:

► $\log_2 \left(\frac{|b - a|}{\text{tol}} \right)$ steps necessary

1.3.2 Model function methods

$\hat{=}$ class of iterative methods for finding zeroes of F :

Idea: Given: approximate zeroes $x^{(k)}, x^{(k-1)}, \dots, x^{(k-m)}$



- ❶ replace F with **model function** \tilde{F}
(using function values/derivative values in $x^{(k)}, x^{(k-1)}, \dots, x^{(k-m)}$)
- ❷ $x^{(k+1)} :=$ zero of \tilde{F}
(has to be readily available \leftrightarrow analytic formula)

Distinguish (see (1.1.1)):

one-point methods : $x^{(k+1)} = \Phi_F(x^{(k)}), k \in \mathbb{N}$ (e.g., fixed point iteration \rightarrow Sect. 1.2)

multi-point methods : $x^{(k+1)} = \Phi_F(x^{(k)}, x^{(k-1)}, \dots, x^{(k-m)}), k \in \mathbb{N}, m = 2, 3, \dots$

1.3.2.1 Newton method in scalar case

Assume: $F : I \mapsto \mathbb{R}$ continuously differentiable

model function:= tangent at F in $x^{(k)}$:

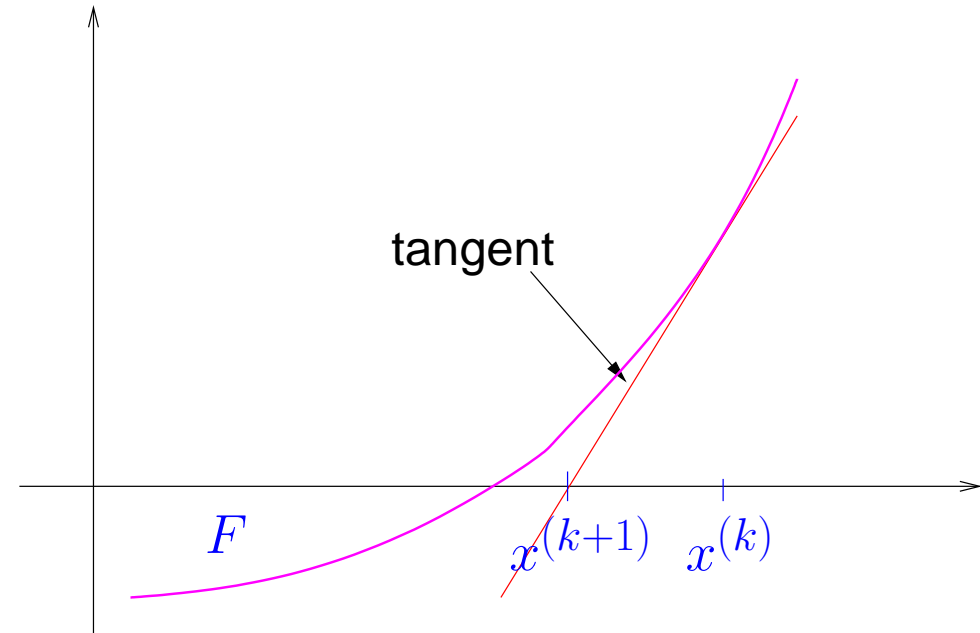
$$\tilde{F}(x) := F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)})$$

take $x^{(k+1)} :=$ zero of tangent

We obtain **Newton iteration**

$$x^{(k+1)} := x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})}, \quad (1.3.1)$$

that requires $F'(x^{(k)}) \neq 0$.



Example 1.3.3 (Newton method in 1D). (\rightarrow Ex. 1.2.1)

Newton iterations for two different scalar non-linear equation with the same solution sets:

$$F(x) = xe^x - 1 \Rightarrow F'(x) = e^x(1+x) \Rightarrow x^{(k+1)} = x^{(k)} - \frac{x^{(k)}e^{x^{(k)}} - 1}{e^{x^{(k)}}(1+x^{(k)})} = \frac{(x^{(k)})^2 + e^{-x^{(k)}}}{1+x^{(k)}}$$

$$F(x) = x - e^{-x} \Rightarrow F'(x) = 1 + e^{-x} \Rightarrow x^{(k+1)} = x^{(k)} - \frac{x^{(k)} - e^{-x^{(k)}}}{1 + e^{-x^{(k)}}} = \frac{1 + x^{(k)}}{1 + e^{x^{(k)}}}$$

1.2.1

1.1.13

Newton iteration (1.3.1) $\hat{=}$ fixed point iteration (\rightarrow Sect.1.2) with iteration function

$$\Phi(x) = x - \frac{F(x)}{F'(x)} \Rightarrow \Phi'(x) = \frac{F(x)F''(x)}{(F'(x))^2} \Rightarrow \Phi'(x^*) = 0, \quad \text{if } F(x^*) = 0, F'(x^*) \neq 0.$$

From Lemma 1.2.7:

Newton method locally quadratically convergent (\rightarrow Def. 1.1.13) to zero x^* , if $F'(x^*) \neq 0$

1.3.2.2 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 1.3.4 (Halley's iteration).

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 (1.3.1) 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 (\rightarrow Sect. ??) \diamond

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 $F \approx \hat{F}$, where \hat{F} is invertible with an inverse \hat{F}^{-1} that can be evaluated with little effort.

$$\blacktriangleright \quad g(x) := \hat{F}^{-1}(F(x)) \approx x .$$

Then apply Newton's method to $g(x)$, using the formula for the derivative of the inverse of a function

$$\frac{d}{dy}(\hat{F}^{-1})(y) = \frac{1}{\hat{F}'(\hat{F}^{-1}(y))} \Rightarrow g'(x) = \frac{1}{\hat{F}'(g(x))} \cdot F'(x) .$$

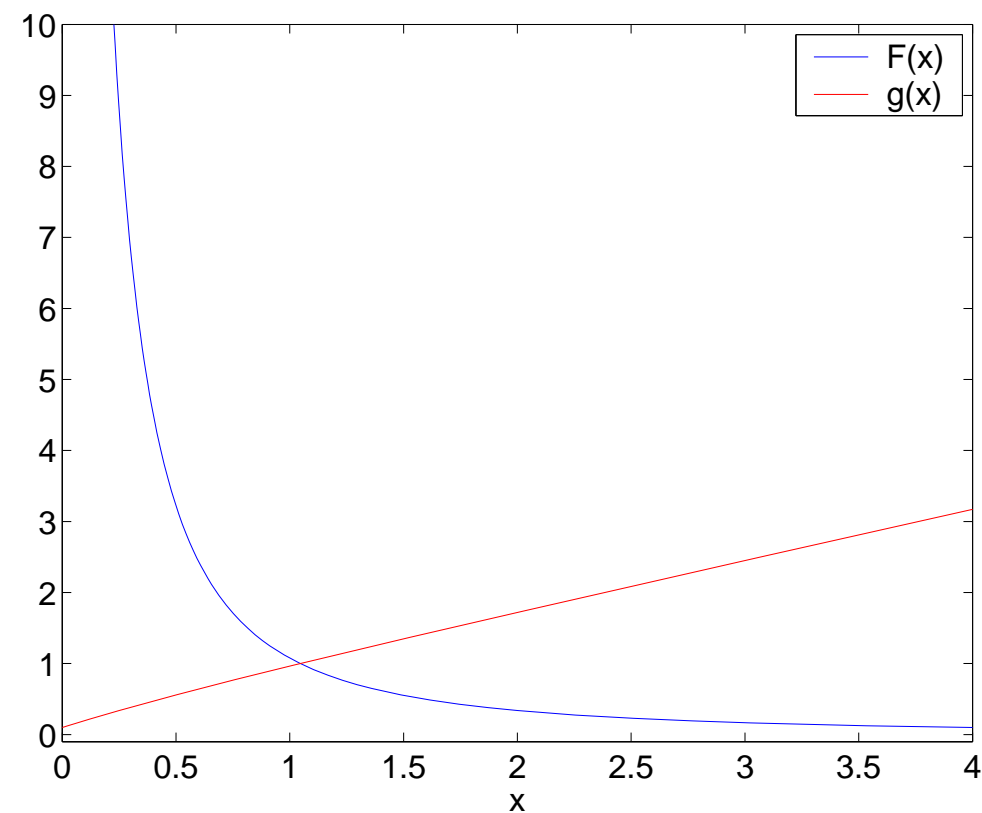
Example 1.3.5 (Adapted Newton method).

$$F(x) = \frac{1}{(x+1)^2} + \frac{1}{(x+0.1)^2} - 1, \quad x > 0 :$$

Observation:

$$F(x) + 1 \approx 2x^{-2} \text{ for } x \gg 1$$

and so $g(x) := \frac{1}{\sqrt{F(x) + 1}}$ “almost” linear for $x \gg 1$



Idea: instead of $F(x) \stackrel{!}{=} 0$ tackle $g(x) \stackrel{!}{=} 1$ with Newton's method (1.3.1).

$$\begin{aligned} x^{(k+1)} &= x^{(k)} - \frac{g(x^{(k)}) - 1}{g'(x^{(k)})} = x^{(k)} + \left(\frac{1}{\sqrt{F(x^{(k)}) + 1}} - 1 \right) \frac{2(F(x^{(k)}) + 1)^{3/2}}{F'(x^{(k)})} \\ &= x^{(k)} + \frac{2(F(x^{(k)}) + 1)(1 - \sqrt{F(x^{(k)}) + 1})}{F'(x^{(k)})}. \end{aligned}$$

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	0.00000000000000	0.00000005485332	-0.00000000000000



For zero finding there is wealth of iterative methods that offer higher order of convergence.

One idea: **consistent modification** of the Newton-Iteration:

► fixed point iteration : $\Phi(x) = x - \frac{F(x)}{F'(x)}H(x)$ with "proper" $H : I \mapsto \mathbb{R}$.

Aim: find H such that the method is of p -th order; tool: Lemma 1.2.7.

Assume: F smooth "enough" and $\exists x^* \in I: F(x^*) = 0, F'(x^*) \neq 0$.

$$\Phi = x - uH \quad , \quad \Phi' = 1 - u'H - uH' \quad , \quad \Phi'' = -u''H - 2u'H - uH'' \quad ,$$

$$\text{with } u = \frac{F}{F'} \Rightarrow u' = 1 - \frac{FF''}{(F')^2} \quad , \quad u'' = -\frac{F''}{F'} + 2\frac{F(F'')^2}{(F')^3} - \frac{FF'''}{(F')^2} \quad .$$

$$F(x^*) = 0 \quad \blacktriangleright \quad u(x^*) = 0, u'(x^*) = 1, u''(x^*) = -\frac{F''(x^*)}{F'(x^*)}.$$

$$\blacktriangleright \quad \Phi'(x^*) = 1 - H(x^*) \quad , \quad \Phi''(x^*) = \frac{F''(x^*)}{F'(x^*)} H(x^*) - 2H'(x^*) . \quad (1.3.2)$$

Lemma 1.2.7 \blacktriangleright **Necessary** conditions for local convergencd of order p :

$$p = 2 \text{ (quadratical convergence): } H(x^*) = 1 ,$$

$$p = 3 \text{ (cubic convergence): } H(x^*) = 1 \quad \wedge \quad H'(x^*) = \frac{1}{2} \frac{F''(x^*)}{F'(x^*)} .$$

In particular: $H(x) = G(1 - u'(x))$ with "proper" G

$$\blacktriangleright \quad \text{fixed point iteration} \quad x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})} G \left(\frac{F(x^{(k)}) F''(x^{(k)})}{(F'(x^{(k)}))^2} \right) . \quad (1.3.3)$$

Lemma 1.3.1. *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 (1.3.3) converge locally cubically to x^* .*

Proof: Lemma 1.2.7, (1.3.2) and

$$H(x^*) = G(0) \quad , \quad H'(x^*) = -G'(0)u''(x^*) = G'(0)\frac{F''(x^*)}{F'(x^*)} .$$

Example 1.3.6 (Example of modified Newton method).

- $G(t) = \frac{1}{1 - \frac{1}{2}t}$ \Rightarrow Halley's iteration (\rightarrow Ex. 1.3.4)
- $G(t) = \frac{2}{1 + \sqrt{1 - 2t}}$ \Rightarrow Euler's iteration
- $G(t) = 1 + \frac{1}{2}t$ \Rightarrow quadratic inverse interpolation

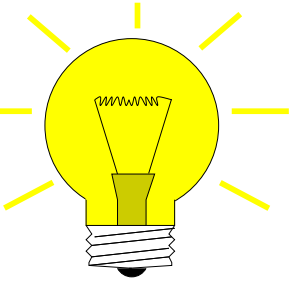
Numerical experiment:

$$F(x) = xe^x - 1 \, ,$$

$$x^{(0)} = 5$$

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	





Idea:

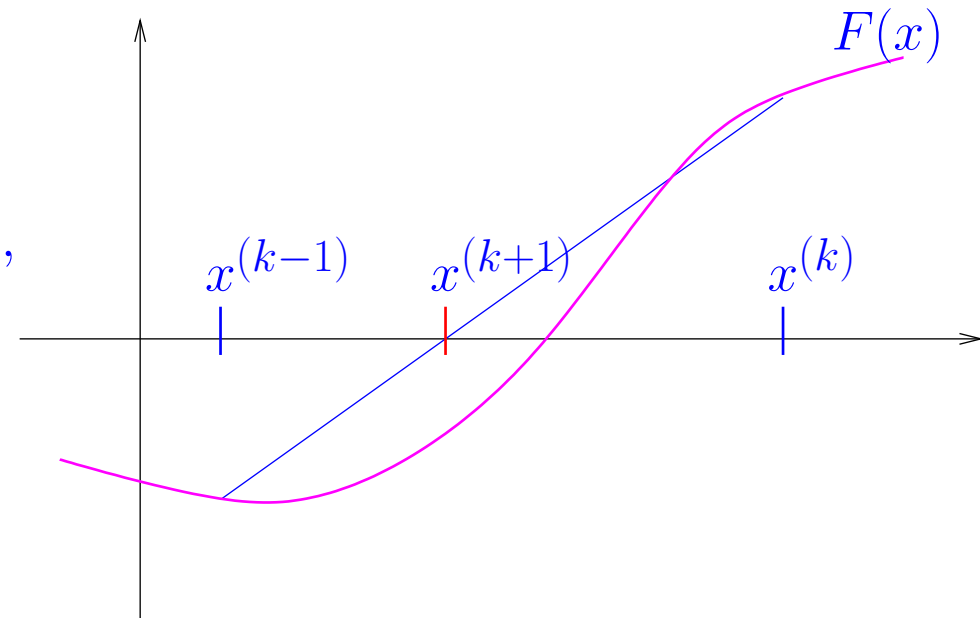
Replace F with **interpolating polynomial**
producing interpolatory model function methods

Simplest representative: **secant method**

$x^{(k+1)}$ = zero of secant

$$s(x) = F(x^{(k)}) + \frac{F(x^{(k)}) - F(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}(x - x^{(k)}) , \quad (1.3.4)$$

►
$$x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})(x^{(k)} - x^{(k-1)})}{F(x^{(k)}) - F(x^{(k-1)})} . \quad (1.3.5)$$



Code 1.3.7: secant method

```
1 def secant(f , x0 , x1 , maxit=50 , tol=1e-12):
2     fo = f(x0)
3     for k in xrange(maxit):
4         fn = f(x1)
5         print 'x1=', x1 , ' f(x1)=', fn
6         s = fn*(x1-x0)/(fn-fo)
7         x0 = x1; x1 -= s
8         if abs(s)<tol:
9             x = x1
10            return x, k
11        fo = fn
12    x = NaN
13    return x, maxit
```

secant method
(python implementation)

- Only one function evaluation per step
- **no derivatives required !**

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.

Example 1.3.8 (secant method). $F(x) = xe^x - 1$, $x^{(0)} = 0$, $x^{(1)} = 5$.

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	-0.000000000000001	-0.000000000000000	

A startling observation: the method seems to have a *fractional* (!) order of convergence, see Def. 1.1.13.



Remark 1.3.9 (Fractional order of convergence of secant method).

Indeed, a fractional order of convergence can be proved for the secant method, see[29, Sect. 18.2]. Here is a crude outline of the reasoning:

Asymptotic convergence of secant method: error $e^{(k)} := x^{(k)} - x^*$

$$e^{(k+1)} = \Phi(x^* + e^{(k)}, x^* + e^{(k-1)}) - x^* \quad , \text{ with } \Phi(x, y) := x - \frac{F(x)(x - y)}{F(x) - F(y)} . \quad (1.3.6)$$

Use MAPLE to find 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));
```

➤ linearized **error propagation formula**:

$$e^{(k+1)} \doteq \frac{1}{2} \frac{F''(x^*)}{F'(x^*)} e^{(k)} e^{(k-1)} = C e^{(k)} e^{(k-1)} . \quad (1.3.7)$$

Try $e^{(k)} = K(e^{(k-1)})^p$ to determine the order of convergence (\rightarrow Def. 1.1.13):

$$\begin{aligned} \Rightarrow e^{(k+1)} &= K^{p+1} (e^{(k-1)})^{p^2} \\ \Rightarrow (e^{(k-1)})^{p^2-p-1} &= K^{-p} C \Rightarrow p^2 - p - 1 = 0 \Rightarrow p = \frac{1}{2}(1 \pm \sqrt{5}) . \end{aligned}$$

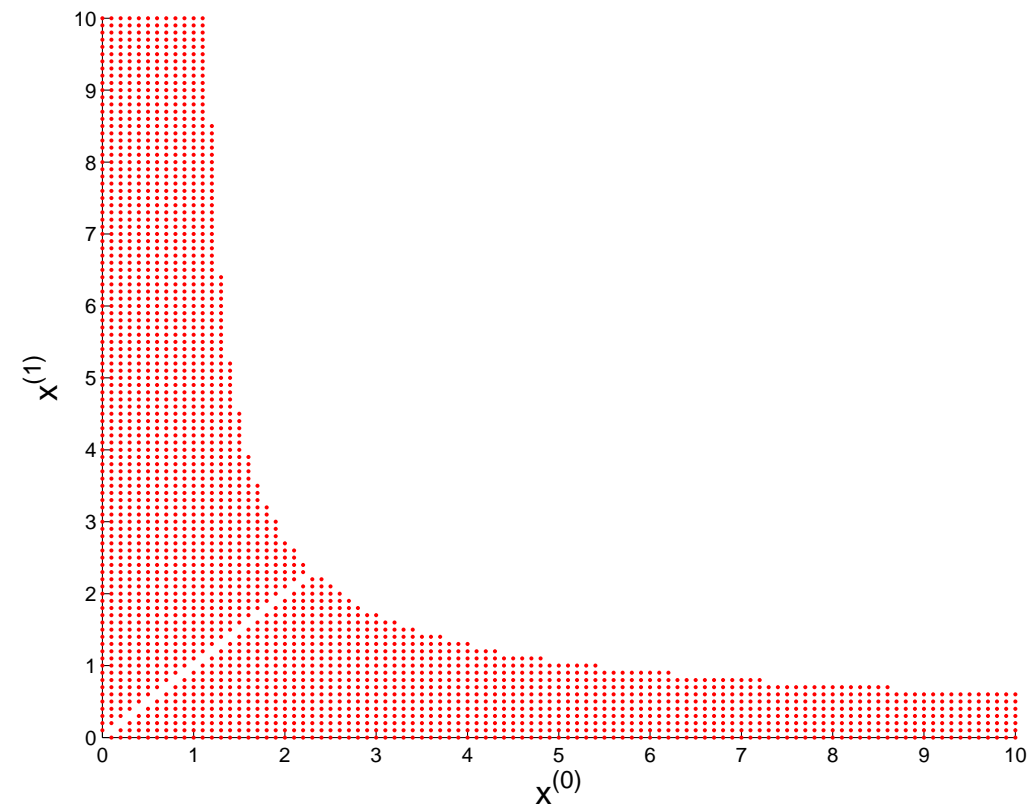
As $e^{(k)} \rightarrow 0$ for $k \rightarrow \infty$ we get the rate of convergence $p = \frac{1}{2}(1 + \sqrt{5}) \approx 1.62$ (see Ex. 1.3.8 !)

Example 1.3.10 (local convergence of secant method).

$$F(x) = \arctan(x)$$

• $\hat{=}$ secant method converges for a pair $(x^{(0)}, x^{(1)})$ of initial guesses.

= local convergence \rightarrow Def. 1.1.3



Another class of multi-point methods: *inverse interpolation*

Assume:

$F : I \subset \mathbb{R} \mapsto \mathbb{R}$ one-to-one

$$F(x^*) = 0 \Rightarrow F^{-1}(0) = x^* .$$

• Interpolate F^{-1} by polynomial p of degree d determined by

$$p(F(x^{(k-m)})) = x^{(k-m)} , \quad m = 0, \dots, d .$$

• New approximate zero $x^{(k+1)} := p(0)$

$F(x^*) = 0 \Leftrightarrow F^{-1}(0) = x^*$

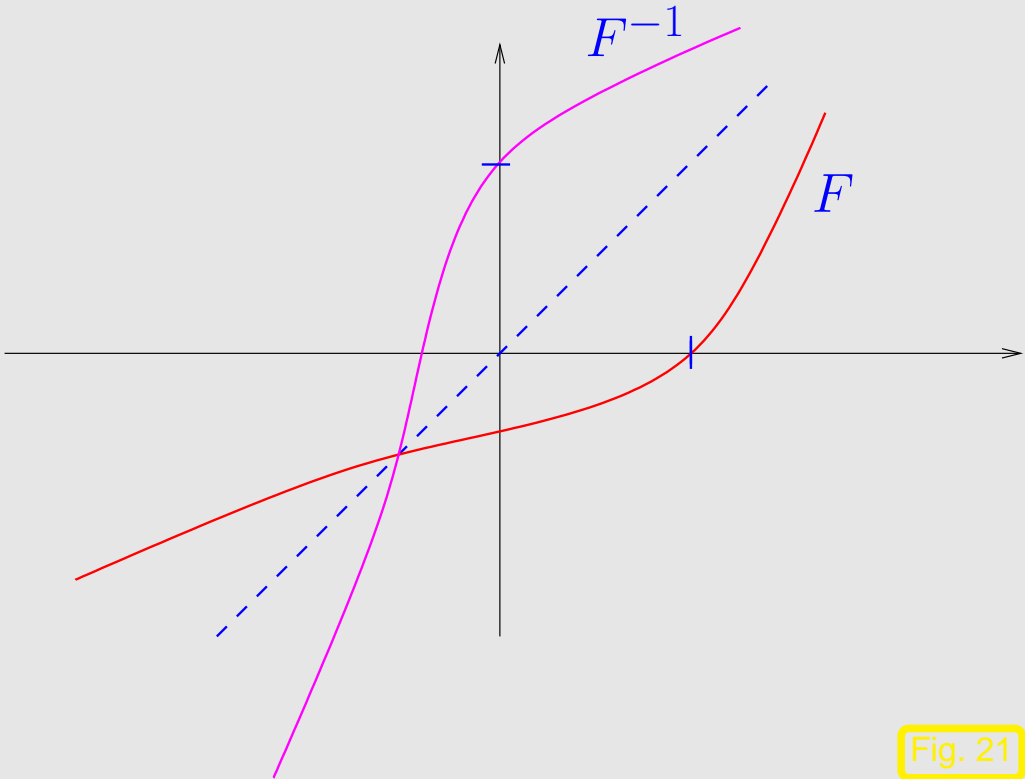


Fig. 21

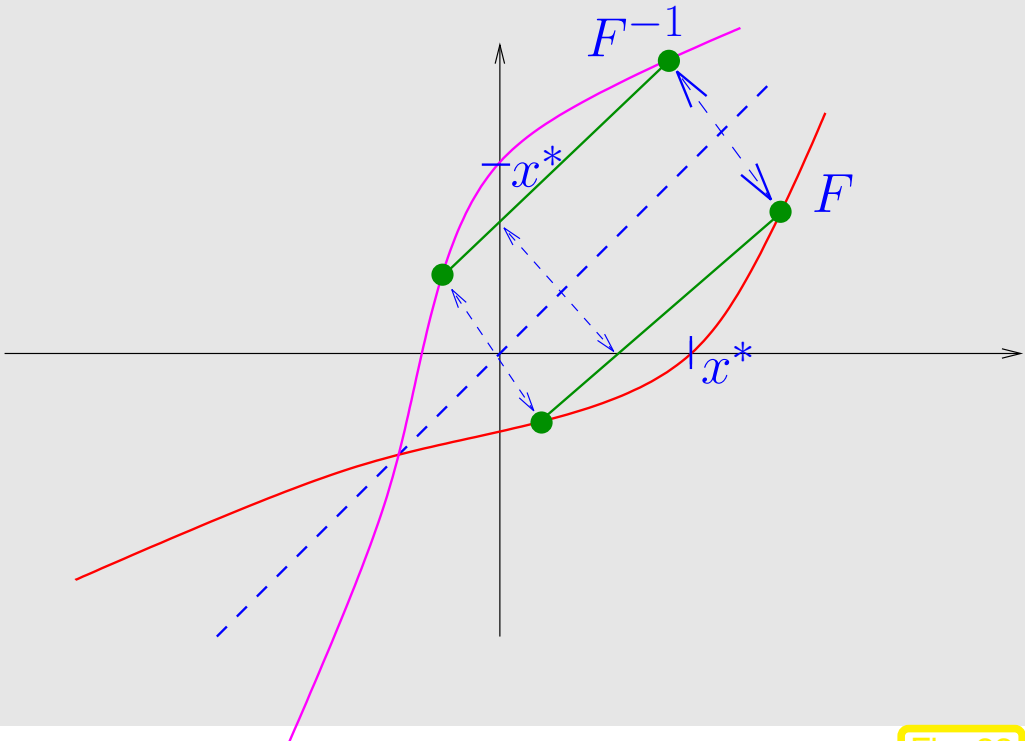



Fig. 22

Case $m = 1 \Rightarrow$ secant method

Case $m = 2$: quadratic inverse interpolation, see [36, Sect. 4.5]

```
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);
```



$$x^{(k+1)} = \frac{F_0^2(F_1x_2 - F_2x_1) + F_1^2(F_2x_0 - F_0x_2) + F_2^2(F_0x_1 - F_1x_0)}{F_0^2(F_1 - F_2) + F_1^2(F_2 - F_0) + F_2^2(F_0 - F_1)} .$$

($F_0 := F(x^{(k-2)}), F_1 := F(x^{(k-1)}), F_2 := F(x^{(k)}), x_0 := x^{(k-2)}, x_1 := x^{(k-1)}, x_2 := x^{(k)}$)

Example 1.3.11 (quadratic inverse interpolation). $F(x) = xe^x - 1$, $x^{(0)} = 0$, $x^{(1)} = 2.5$, $x^{(2)} = 5$.

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	0.000000000000003	0.000000000000001	

Also in this case the numerical experiment hints at a fractional rate of convergence, as in the case of the secant method, see Rem. 1.3.9.



1.4 Newton's Method

Non-linear system of equations: for $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$ find $\mathbf{x}^* \in D$: $F(\mathbf{x}^*) = 0$

Assume: $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$ continuously differentiable

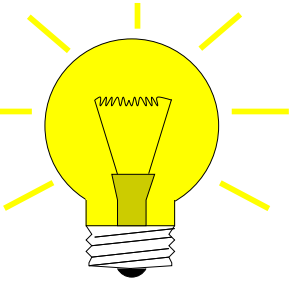
Idea (\rightarrow Sect. 1.3.2.1):

local linearization:

Given $\mathbf{x}^{(k)} \in D \supset \mathbf{x}^{(k+1)}$ as zero of affine linear model function

$$F(\mathbf{x}) \approx \tilde{F}(\mathbf{x}) := F(\mathbf{x}^{(k)}) + DF(\mathbf{x}^{(k)})(\mathbf{x} - \mathbf{x}^{(k)}) ,$$

$$DF(\mathbf{x}) \in \mathbb{R}^{n,n} = \text{Jacobian (ger.: Jacobi-Matrix)}, \quad DF(\mathbf{x}) = \left(\frac{\partial F_j}{\partial x_k}(\mathbf{x}) \right)_{j,k=1}^n .$$



Newton iteration: (\leftrightarrow (1.3.1) for $n = 1$)

$$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} - DF(\mathbf{x}^{(k)})^{-1} F(\mathbf{x}^{(k)}) , \quad [\text{if } DF(\mathbf{x}^{(k)}) \text{ regular}] \quad (1.4.1)$$

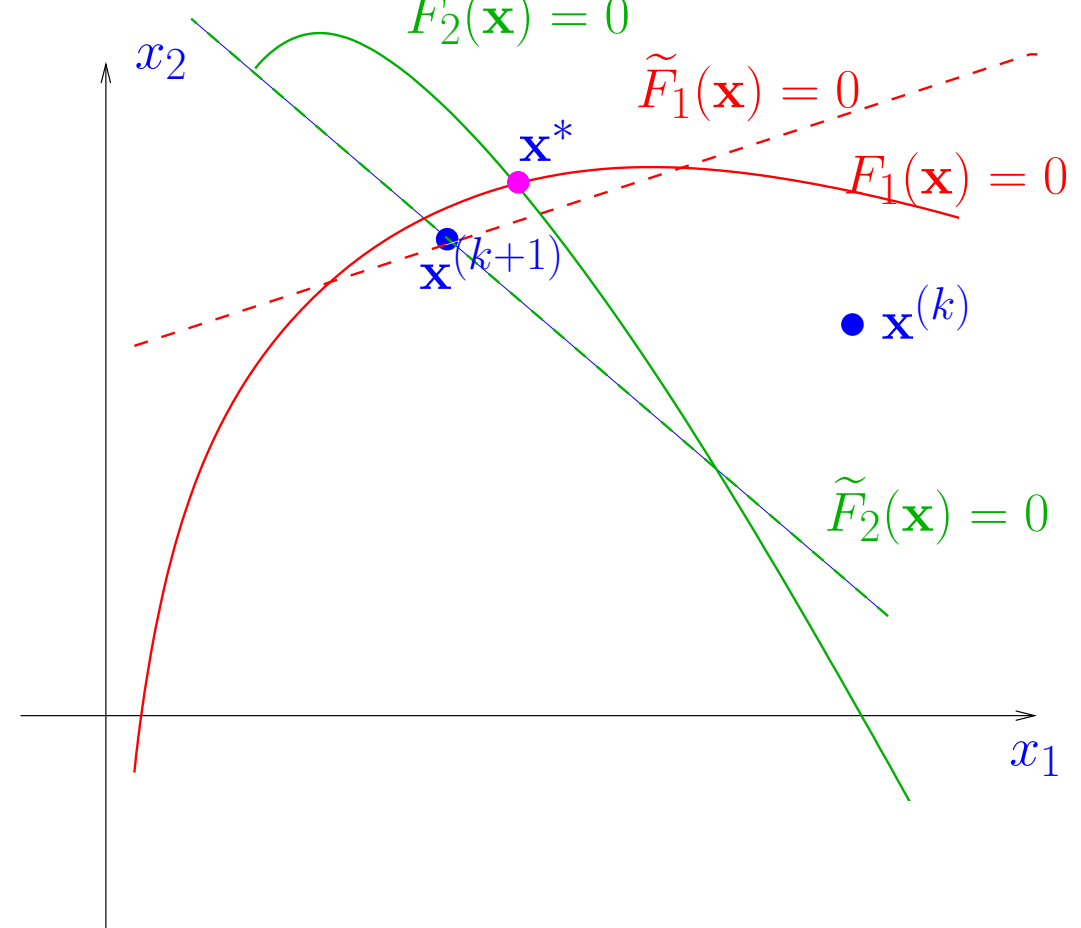
Terminology: $-DF(\mathbf{x}^{(k)})^{-1} F(\mathbf{x}^{(k)})$ = Newton correction

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) that are approximations of the original curves



MATLAB-CODE: Newtonverfahren

```
def newton(x, F, DF, tol, maxit):
    for i in xrange(maxit):
```

template for Newton method:

Solve linear system:

$$\text{solve}(A, b) = A^{-1}b$$

F, DF : functions

A posteriori termination criterion

```
        s = solve(DF(x), F(x))
```

```
        x -= s
```

```
    if norm(s) < tol*norm(x): return
```

Example 1.4.1 (Newton method in 2D).

$$F(\mathbf{x}) = \begin{pmatrix} x_1^2 - x_2^4 \\ x_1 - x_2^3 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \mathbb{R}^2 \quad \text{with solution} \quad F \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 0.$$

Jacobian (analytic computation):

$$DF(\mathbf{x}) = \begin{pmatrix} \partial_{x_1} F_1(x) & \partial_{x_2} F_1(x) \\ \partial_{x_1} F_2(x) & \partial_{x_2} F_2(x) \end{pmatrix} = \begin{pmatrix} 2x_1 & -4x_2^3 \\ 1 & -3x_2^2 \end{pmatrix}$$

Realization of Newton iteration (1.4.1):

1. Solve LSE

$$\begin{pmatrix} 2x_1 & -4x_2^3 \\ 1 & -3x_2^2 \end{pmatrix} \Delta \mathbf{x}^{(k)} = \begin{pmatrix} x_1^2 - x_2^4 \\ x_1 - x_2^3 \end{pmatrix},$$

where $\mathbf{x}^{(k)} = (x_1, x_2)^T$.

2. Set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}$

Code 1.4.2: Newton iteration in 2D

```

from scipy import array, diff, log, zeros, vstack
from scipy.linalg import norm, solve

F = lambda x: array([x[0]**2 - x[1]**4, x[0]-x[1]**3])
DF = lambda x: array([[2*x[0], - 4*x[1]**3], [1, -3*x[1]**2]])

x = array([0.7, 0.7])
x0 = array([1., 1.])
tol = 1e-10

res = zeros(4); res[1:3] = x; res[3] = norm(x-x0)
print DF(x)
print F(x)
s = solve(DF(x),F(x))
x -= s
res1 = zeros(4); res[0] = 1.; res1[1:3] = x; res1[3] = norm(x-x0)
res = vstack((res,res1))
k = 2
while norm(s) > tol*norm(x):
    s = solve(DF(x),F(x))

```

```

x -= s
res1 = zeros(4); res1[0] = k; res1[1:3] = x; res1[3] = norm(x-x0)
res = vstack((res,res1))
k += 1

```

```

logdiff = diff(log(res[:,3]))
rates = logdiff[1:]/logdiff[:-1]

```

```

print res
print rates

```

k	$\mathbf{x}^{(k)}$	$\epsilon_k := \ \mathbf{x}^* - \mathbf{x}^{(k)}\ _2$
0	$(0.7, 0.7)^T$	4.24e-01
1	$(0.878500000000000, 1.064285714285714)^T$	1.37e-01
2	$(1.01815943274188, 1.00914882463936)^T$	2.03e-02
3	$(1.00023355916300, 1.00015913936075)^T$	2.83e-04
4	$(1.000000000583852, 1.000000002726552)^T$	2.79e-08
5	$(0.9999999999999998, 1.0000000000000000)^T$	2.11e-15
6	$(1, 1)^T$	

New aspect for $n \gg 1$ (compared to $n = 1$ -dimensional case, section. 1.3.2.1):

Computation of the Newton correction is eventually costly!



Remark 1.4.3 (Affine invariance of Newton method).

An important property of the Newton iteration (1.4.1): **affine invariance** \rightarrow [12, Sect .1.2.2]

set $G(\mathbf{x}) := \mathbf{A}F(\mathbf{x})$ with regular $\mathbf{A} \in \mathbb{R}^{n,n}$ so that $F(\mathbf{x}^*) = 0 \Leftrightarrow G(\mathbf{x}^*) = 0$.



affine invariance: Newton iteration for $G(\mathbf{x}) = 0$ is the same for all $\mathbf{A} \in GL(n)$!

This is a simple computation:

$$DG(\mathbf{x}) = \mathbf{A}DF(\mathbf{x}) \Rightarrow DG(\mathbf{x})^{-1}G(\mathbf{x}) = DF(\mathbf{x})^{-1}\mathbf{A}^{-1}\mathbf{A}F(\mathbf{x}) = DF(\mathbf{x})^{-1}F(\mathbf{x}) .$$

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.

Remark 1.4.4 (Differentiation rules). \rightarrow Repetition: basic analysis

Statement of the Newton iteration (1.4.1) for $F : \mathbb{R}^n \mapsto \mathbb{R}^n$ given as analytic expression entails computing the Jacobian DF . To avoid cumbersome component-oriented considerations, it is useful to know the *rules of multidimensional differentiation*:

Let V, W be finite dimensional vector spaces, $F : D \subset V \mapsto W$ sufficiently smooth. The **differential** $DF(\mathbf{x})$ of F in $\mathbf{x} \in V$ is the *unique*

$$\begin{aligned} & \text{linear mapping } DF(\mathbf{x}) : V \mapsto W, \\ \text{such that } & \|F(\mathbf{x} + \mathbf{h}) - F(\mathbf{x}) - DF(\mathbf{x})\mathbf{h}\| = o(\|\mathbf{h}\|) \quad \forall \mathbf{h}, \|\mathbf{h}\| < \delta. \end{aligned}$$

Gradinaru
D-MATH

- For $F : V \mapsto W$ linear, i.e. $F(\mathbf{x}) = \mathbf{A}\mathbf{x}$, \mathbf{A} matrix $\rightarrow DF(\mathbf{x}) = \mathbf{A}$.
- **Chain rule**: $F : V \mapsto W, G : W \mapsto U$ sufficiently smooth

$$D(G \circ F)(\mathbf{x})\mathbf{h} = DG(F(\mathbf{x}))(DF(\mathbf{x}))\mathbf{h}, \quad \mathbf{h} \in V, \mathbf{x} \in D. \quad (1.4.2)$$

- **Product rule:** $F : D \subset V \mapsto W$, $G : D \subset V \mapsto U$ sufficiently smooth, $b : W \times U \mapsto Z$ **bilinear**

$$T(\mathbf{x}) = b(F(\mathbf{x}), G(\mathbf{x})) \Rightarrow DT(\mathbf{x})\mathbf{h} = b(DF(\mathbf{x})\mathbf{h}, G(\mathbf{x})) + b(F(\mathbf{x}), DG(\mathbf{x})\mathbf{h}), \quad (1.4.3)$$

$$\mathbf{h} \in V, \mathbf{x} \in D.$$

For $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}$ the **gradient** $\text{grad } F : D \mapsto \mathbb{R}^n$, and the **Hessian matrix** $HF(\mathbf{x}) : D \mapsto \mathbb{R}^{n,n}$ are defined as

$$\text{grad } F(\mathbf{x})^T \mathbf{h} := DF(\mathbf{x})\mathbf{h}, \quad \mathbf{h}_1^T HF(\mathbf{x})\mathbf{h}_2 := D(DF(\mathbf{x})(\mathbf{h}_1))(\mathbf{h}_2), \quad \mathbf{h}, \mathbf{h}_1, \mathbf{h}_2 \in V.$$



Remark 1.4.5 (Simplified Newton method).

Simplified Newton Method: use the same $DF(\mathbf{x}^{(k)})$ for more steps

➤ (usually) merely linear convergence instead of quadratic convergence



Remark 1.4.6 (Numerical Differentiation for computation of Jacobian).

If $DF(\mathbf{x})$ is not available (e.g. when $F(\mathbf{x})$ is given only as a procedure):

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: impact of roundoff errors for small h !



Example 1.4.7 (Roundoff errors and difference quotients).

Approximate derivative by difference quotient:
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

Calculus: better approximation for smaller $h > 0$, isn't it ?

$\log_{10}(h)$	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	-1.00000000000000

MATLAB-CODE: Numerical differentiation of $\exp(x)$

```
h = 0.1; x = 0.
for i in xrange(16):
    df = (exp(x+h)-exp(x))/h
    print -i, df-1
    h *= 0.1
```

Recorded relative error, $f(x) = e^x$, $x = 0$ \triangleright

Note: An analysis based on expressions for remainder terms of Taylor expansions shows that the **approximation error** cannot be blamed for the loss of accuracy as $h \rightarrow 0$ (as expected).

Explanation relying on roundoff error analysis, see Sect. ??:

MATLAB-CODE: Numerical differentiation of exp(x)

```
h = 0.1; x = 0.0
for i in xrange(16):
    df = (exp(x+h)-exp(x))/h
    print -i, df-1
    h *= 0.1
```

Obvious **cancellation** → error amplification

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

$\log_{10}(h)$	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	-1.00000000000000

Analysis for $f(x) = \exp(x)$:

$$\text{df} = \frac{e^{x+h} (1 + \delta_1) - e^x (1 + \delta_2)}{h}$$

correction factors take into account roundoff:
(→ "axiom of roundoff analysis", Ass. ??)

$$= e^x \left(\frac{e^h - 1}{h} + \frac{\delta_1 e^h - \delta_2}{h} \right)$$

$$|\delta_1|, |\delta_2| \leq \text{eps}.$$

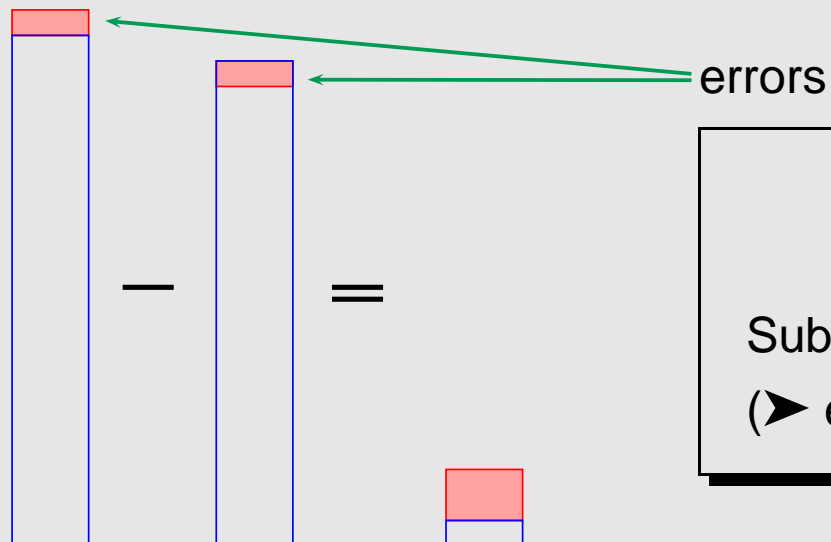
$$1 + O(h) \quad O(h^{-1}) \quad \text{für } h \rightarrow 0$$

► relative error: $\left| \frac{e^x - \text{df}}{e^x} \right| \approx h + \frac{2\text{eps}}{h} \rightarrow \min \quad \text{for } h = \sqrt{2\text{eps}} .$

In double precision: $\sqrt{2\text{eps}} = 2.107342425544702 \cdot 10^{-8}$



What is this mysterious cancellation (*ger.:* Auslöschung) ?



Cancellation

$\hat{=}$

Subtraction of almost equal numbers
(► extreme amplification of relative errors)

Example 1.4.8 (cancellation in decimal floating point arithmetic).

x, y afflicted with relative errors $\approx 10^{-7}$:

$$\begin{array}{rcl}
 x & = & 0.123467* \quad \leftarrow \text{7th digit perturbed} \\
 y & = & 0.123456* \quad \leftarrow \text{7th digit perturbed} \\
 \hline
 x - y & = & 0.000011* = 0.11*000 \cdot 10^{-4} \quad \leftarrow \text{3rd digit perturbed} \\
 & & \quad \quad \quad \uparrow \\
 & & \quad \quad \text{padded zeroes}
 \end{array}$$



1.4.2 Convergence of Newton's method

Newton iteration (1.4.1) $\hat{=}$ fixed point iteration (\rightarrow Sect. 1.2) with

$$\Phi(\mathbf{x}) = \mathbf{x} - DF(\mathbf{x})^{-1}F(\mathbf{x}) .$$

$$\begin{array}{c}
 \text{[“product rule” : } DF(\mathbf{x}) = \mathbf{I} - D(DF(\mathbf{x})^{-1})F(\mathbf{x}) - DF(\mathbf{x})^{-1}DF(\mathbf{x}) \text{]} \\
 \hline
 \end{array}$$

$$F(\mathbf{x}^*) = 0 \Rightarrow D\Phi(\mathbf{x}^*) = 0 .$$

1.2.7

Local quadratic convergence of Newton's method, if $DF(\mathbf{x}^*)$ regular

Example 1.4.9 (Convergence of Newton's method).

Ex. 1.4.1 cnt'd: record of iteration errors, see Code 1.4.1:

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	$(1.000000000583852, 1.000000002726552)^T$	2.79e-08	1.77
5	$(0.9999999999999998, 1.0000000000000000)^T$	2.11e-15	
6	$(1, 1)^T$		

There is a sophisticated theory about the convergence of Newton's method. For example one can find the following theorem in [14, Thm. 4.10], [12, Sect. 2.1]):

Theorem 1.4.1 (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) $DF(\mathbf{x})$ regular $\forall \mathbf{x} \in D$,
- (D) $\exists L \geq 0$: $\left\| DF(\mathbf{x})^{-1}(DF(\mathbf{x} + \mathbf{v}) - DF(\mathbf{x})) \right\|_2 \leq L \|\mathbf{v}\|_2 \quad \forall \mathbf{v} \in \mathbb{R}^n, \mathbf{v} + \mathbf{x} \in D, \quad \forall \mathbf{x} \in D$,
- (E) $\exists \mathbf{x}^*$: $F(\mathbf{x}^*) = 0$ (*existence of solution in D*)
- (F) initial guess $\mathbf{x}^{(0)} \in D$ satisfies $\rho := \left\| \mathbf{x}^* - \mathbf{x}^{(0)} \right\|_2 < \frac{2}{L} \wedge B_\rho(\mathbf{x}^*) \subset D$.

then the Newton iteration (1.4.1) 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) $\left\| \mathbf{x}^{(k+1)} - \mathbf{x}^* \right\|_2 \leq \frac{L}{2} \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|_2^2$ (*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**

Problem: Usually neither ω nor x^* are known !

► In general: a priori estimates as in Thm. 1.4.1 are of little practical relevance.

1.4.3 Termination of Newton iteration

A first viable idea:

Asymptotically due to quadratic convergence:

$$\left\| \mathbf{x}^{(k+1)} - \mathbf{x}^* \right\| \ll \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| \quad \Rightarrow \quad \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| \approx \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| . \quad (1.4.4)$$

➤ quit iterating as soon as $\left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| = \left\| DF(\mathbf{x}^{(k)})^{-1} F(\mathbf{x}^{(k)}) \right\| < \tau \left\| \mathbf{x}^{(k)} \right\|$,
with τ = tolerance

→ uneconomical: one needless update, because $\mathbf{x}^{(k)}$ already accurate enough !

Remark 1.4.10. New aspect for $n \gg 1$: computation of Newton correction may be expensive !



Therefore we would like to use an a-posteriori termination criterion that dispenses with computing (and “inverting”) another Jacobian $DF(\mathbf{x}^{(k)})$ just to tell us that $\mathbf{x}^{(k)}$ is already accurate enough.



Practical a-posteriori termination criterion for Newton's method:

$$DF(\mathbf{x}^{(k-1)}) \approx DF(\mathbf{x}^{(k)}): \text{ quit as soon as } \left\| DF(\mathbf{x}^{(k-1)})^{-1} F(\mathbf{x}^{(k)}) \right\| < \tau \left\| \mathbf{x}^{(k)} \right\|$$

affine invariant termination criterion

Justification: we expect $DF(\mathbf{x}^{(k-1)}) \approx DF(\mathbf{x}^{(k)})$, when Newton iteration has converged. Then appeal to (1.4.4).

If we used the residual based termination criterion

$$\left\| F(\mathbf{x}^{(k)}) \right\| \leq \tau ,$$

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.

Terminology: $\Delta \bar{\mathbf{x}}^{(k)} := DF(\mathbf{x}^{(k-1)})^{-1} F(\mathbf{x}^{(k)}) \hat{=}$ **simplified Newton correction**

Reuse of LU-factorization of $DF(\mathbf{x}^{(k-1)}) \rightarrow \Delta \bar{\mathbf{x}}^{(k)}$ available
with $O(n^2)$ operations

Summary: The Newton 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.

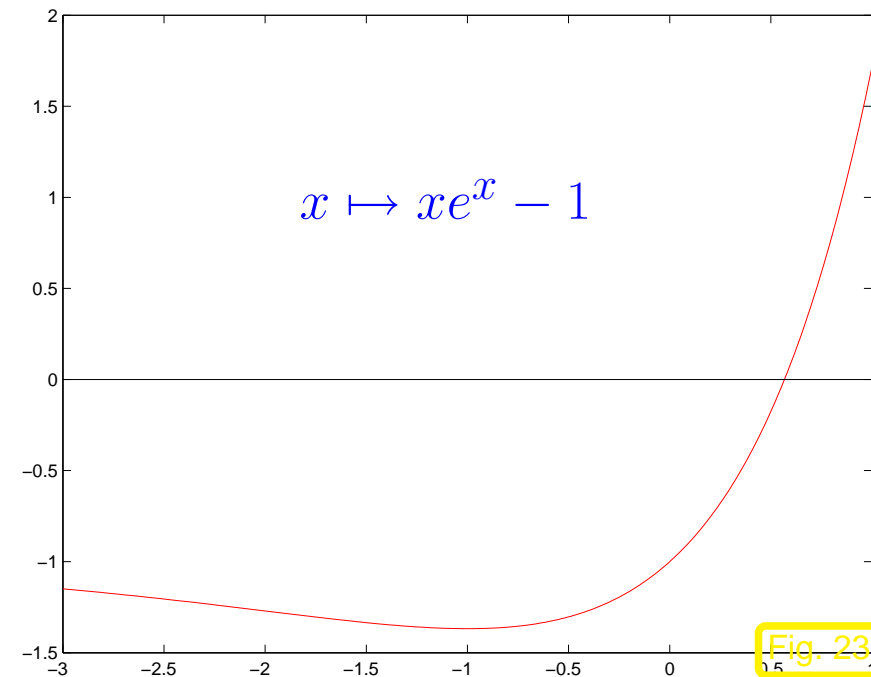
1.4.4 Damped Newton method

Example 1.4.11 (Local convergence of Newton's method).

$$F(x) = xe^x - 1 \Rightarrow F'(-1) = 0$$

$$x^{(0)} < -1 \Rightarrow x^{(k)} \rightarrow -\infty$$

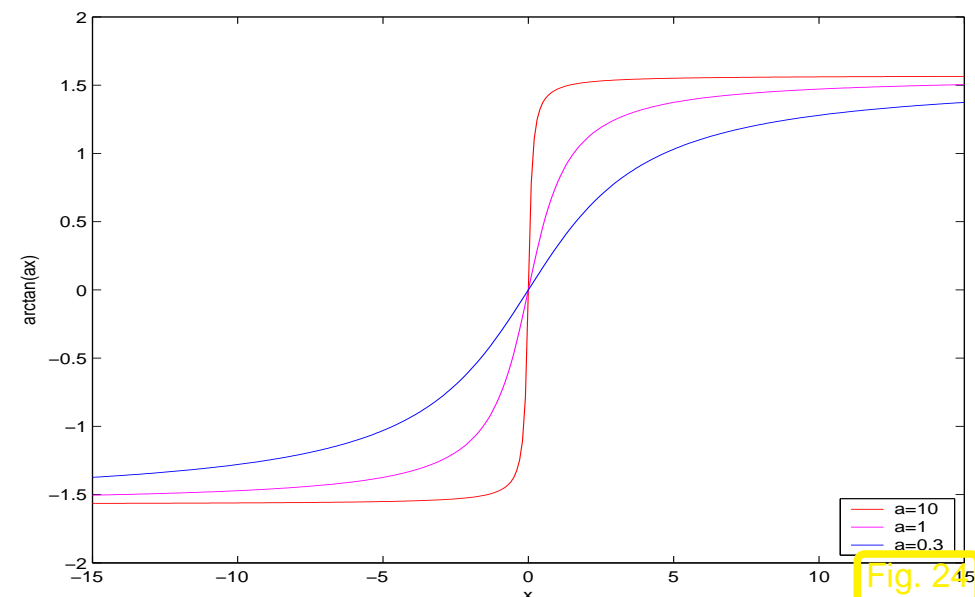
$$x^{(0)} > -1 \Rightarrow x^{(k)} \rightarrow x^*$$

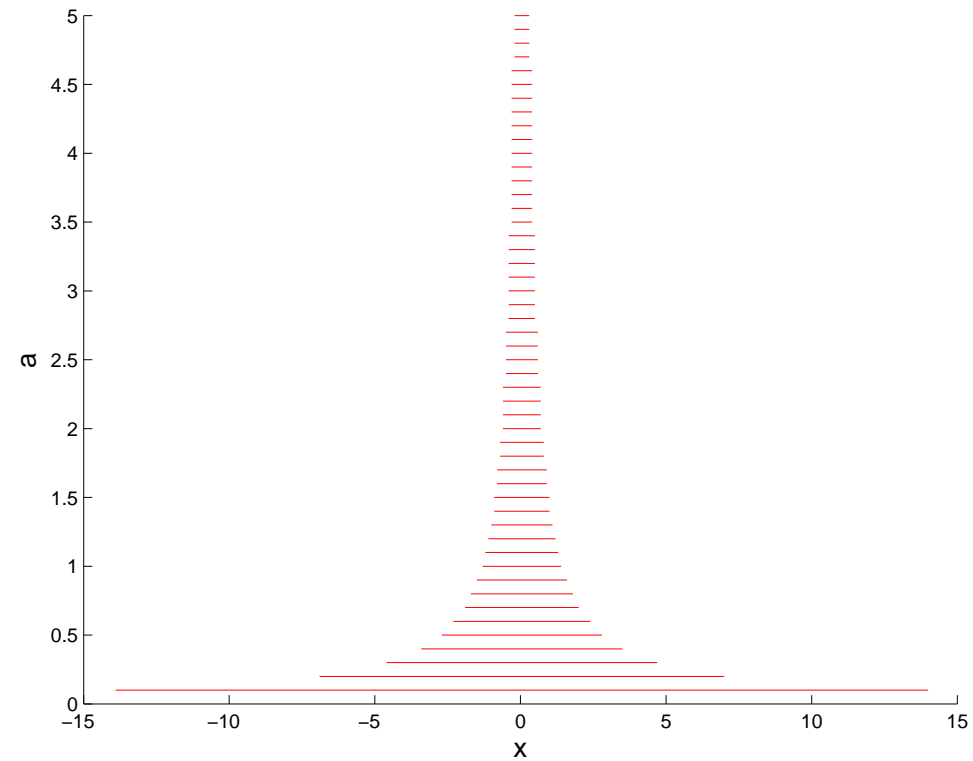
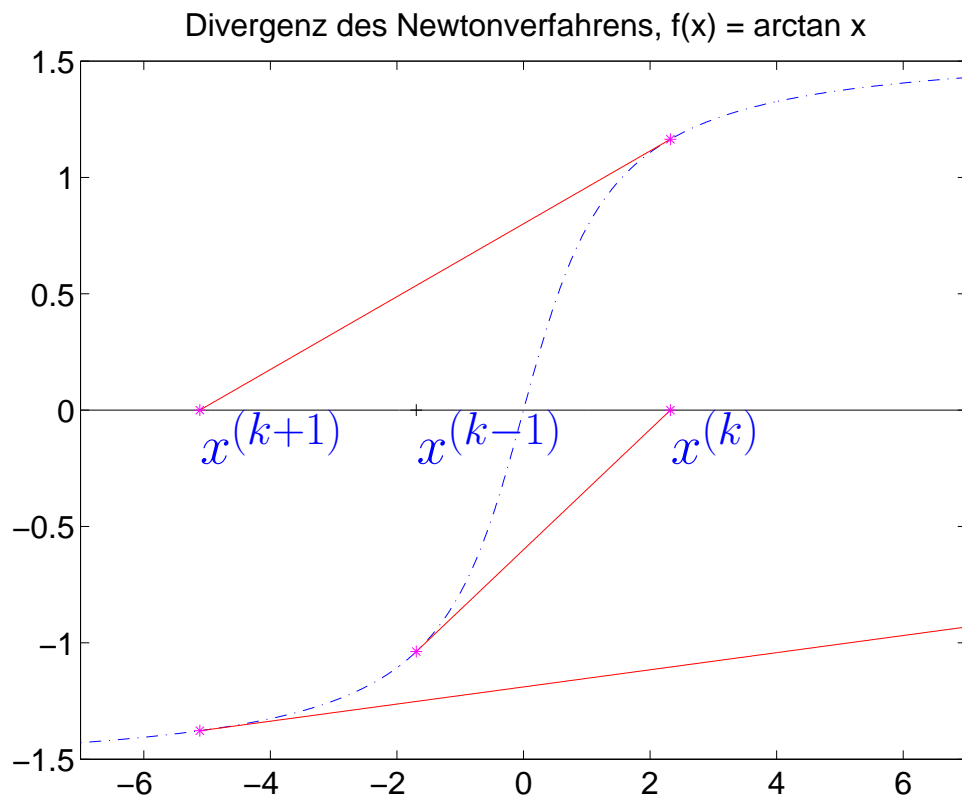


Example 1.4.12 (Region of convergence of Newton method).

$$F(x) = \arctan(ax), \quad a > 0, x \in \mathbb{R}$$

with zero $x^* = 0$.





red zone = $\{x^{(0)} \in \mathbb{R}, x^{(k)} \rightarrow 0\}$

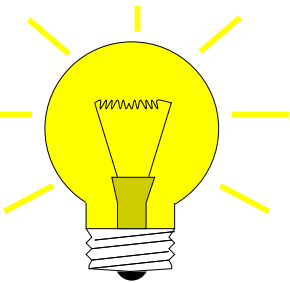
► we observe “overshooting” of Newton correction

Idea:

damping of Newton correction:

With $\lambda^{(k)} > 0$: $\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} - \lambda^{(k)} DF(\mathbf{x}^{(k)})^{-1} F(\mathbf{x}^{(k)})$. (1.4.5)

Terminology: $\lambda^{(k)}$ = damping factor



Choice of damping factor: affine invariant **natural monotonicity test**

$$\text{“maximal” } \lambda^{(k)} > 0: \quad \left\| \Delta \bar{\mathbf{x}}(\lambda^{(k)}) \right\| \leq \left(1 - \frac{\lambda^{(k)}}{2}\right) \left\| \Delta \mathbf{x}^{(k)} \right\|_2 \quad (1.4.6)$$

where $\Delta \mathbf{x}^{(k)} := DF(\mathbf{x}^{(k)})^{-1} F(\mathbf{x}^{(k)}) \rightarrow$ current Newton correction ,
 $\Delta \bar{\mathbf{x}}(\lambda^{(k)}) := DF(\mathbf{x}^{(k)})^{-1} F(\mathbf{x}^{(k)}) - \lambda^{(k)} \Delta \mathbf{x}^{(k)} \rightarrow$ tentative simplified Newton correction .

Heuristics: convergence \Leftrightarrow size of Newton correction decreases

Code 1.4.13: Damped Newton method

```
from scipy.linalg import lu_solve, lu_factor, norm
from scipy import array, arctan, exp

def dampnewton(x, F, DF, q=0.5, tol=1e-10):
    cvg = []
    lup = lu_factor(DF(x))
    s = lu_solve(lup, F(x))
    xn = x - s
    lam = 1
    st = lu_solve(lup, F(xn))
    while norm(st) > tol * norm(xn): #a posteriori termination criteria
        while norm(st) > (1 - lam * 0.5) * norm(s): #natural monotonicity test
            lam *= 0.5 # reduce damping factor
            if lam < 1e-10:
                cvg = -1
                print 'DAMPED_NEWTON: Failure of convergence'
```

```

        return x, cvg
    xn = x-lam*s
    st = lu_solve(lup,F(xn)) #simplified Newton cf. Sect. 1.4.3
    cvg += [[lam, norm(xn), norm(F(xn))]]
    x = xn
    lup = lu_factor(DF(x))
    s = lu_solve(lup,F(x))
    lam = min(lam/q, 1.)
    xn = x-lam*s
    st = lu_solve(lup,F(xn)) #simplified Newton cf. Sect. 1.4.3
x = xn
return x, array(cvg)

```

```

if __name__=='__main__':
    print '_____2D_F_____',
    F = lambda x: array([x[0]**2 - x[1]**4, x[0]-x[1]**3])
    DF = lambda x: array([[2*x[0], - 4*x[1]**3], [1, -3*x[1]**2]])
    x = array([0.7, 0.7])
    x0 = array([1., 1.])
    x, cvg = dampnewton(x,F,DF)
    print x
    print cvg

    print '_____arctan_____',

```

```
F = lambda x: array([arctan(x[0])])
DF = lambda x: array([[1./(1.+x[0]**2)])]
x = array([20.])
x, cvg = dampnewton(x,F,DF)
print x
print cvg

print '_____x_e^x_1_____'
F = lambda x: array([x[0]*exp(x[0]) - 1.])
DF = lambda x: array([[exp(x[0])*(x[0]+1.)]])
x = array([-1.5])
x, cvg = dampnewton(x,F,DF)
print x
print cvg
```

Policy: Reduce damping factor by a factor $q \in]0, 1[$ (usually $q = \frac{1}{2}$) until the affine invariant natural monotonicity test (1.4.6) passed.

Example 1.4.14 (Damped Newton method). (\rightarrow Ex. 1.4.12)

$$F(x) = \arctan(x) ,$$

- $x^{(0)} = 20$
- $q = \frac{1}{2}$
- LMIN = 0.001

Observation: asymptotic
quadratic convergence

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	-0.000000000000001	-0.000000000000001



Example 1.4.15 (Failure of damped Newton method).

- As in Ex. 1.4.11:
 $F(x) = xe^x - 1,$
- Initial guess for damped
Newton method $x^{(0)} = -1.5$

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 $\lambda < \text{LMIN}$!			

Observation: Newton correction pointing in “wrong direction” so no convergence.



What to do when $DF(\mathbf{x})$ is not available and numerical differentiation (see remark 1.4.6) is too expensive?

Idea: in one dimension ($n = 1$) apply the secant method (1.3.4) of section 1.3.2.3

$$F'(x^{(k)}) \approx \frac{F(x^{(k)}) - F(x^{(k-1)})}{x^{(k)} - x^{(k-1)}} \quad \text{"difference quotient"} \quad (1.4.7)$$

already computed ! \rightarrow cheap



Generalisation for $n > 1$?

Idea: rewrite (1.4.7) as a **secant condition** for the approximation $\mathbf{J}_k \approx DF(\mathbf{x}^{(k)})$, $\mathbf{x}^{(k)} \hat{=}$ iterate:

$$\mathbf{J}_k(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}) = F(\mathbf{x}^{(k)}) - F(\mathbf{x}^{(k-1)}) . \quad (1.4.8)$$

BUT:

many matrices \mathbf{J}_k fulfill (1.4.8)

Hence:

we need more conditions for $\mathbf{J}_k \in \mathbb{R}^{n,n}$

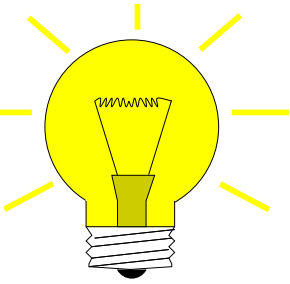
Idea: get

\mathbf{J}_k by a **modification** of \mathbf{J}_{k-1}

Broyden conditions: $\mathbf{J}_k \mathbf{z} = \mathbf{J}_{k-1} \mathbf{z} \quad \forall \mathbf{z}: \mathbf{z} \perp (\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)})$. (1.4.9)

i.e.:

$$\mathbf{J}_k := \mathbf{J}_{k-1} + \frac{F(\mathbf{x}^{(k)})(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)})^T}{\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_2^2} \quad (1.4.10)$$



Broydens Quasi-Newton Method for solving $F(\mathbf{x}) = 0$:

$$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + \Delta \mathbf{x}^{(k)}, \quad \Delta \mathbf{x}^{(k)} := -\mathbf{J}_k^{-1} F(\mathbf{x}^{(k)}), \quad \mathbf{J}_{k+1} := \mathbf{J}_k + \frac{F(\mathbf{x}^{(k+1)})(\Delta \mathbf{x}^{(k)})^T}{\|\Delta \mathbf{x}^{(k)}\|_2^2} \quad (1.4.11)$$

Initialize \mathbf{J}_0 e.g. with the exact Jacobi matrix $DF(\mathbf{x}^{(0)})$.

Remark 1.4.16 (Minimal property of Broydens rank 1 modification).

Let $\mathbf{J} \in \mathbb{R}^{n,n}$ fulfill (1.4.8)
and $\mathbf{J}_k, \mathbf{x}^{(k)}$ from (1.4.11)

then $(\mathbf{I} - \mathbf{J}_k^{-1} \mathbf{J})(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = -\mathbf{J}_k^{-1} F(\mathbf{x}^{(k+1)})$

and hence

$$\begin{aligned} \left\| \mathbf{I} - \mathbf{J}_k^{-1} \mathbf{J}_{k+1} \right\|_2 &= \left\| \frac{-\mathbf{J}_k^{-1} F(\mathbf{x}^{(k+1)}) \Delta \mathbf{x}^{(k)}}{\left\| \Delta \mathbf{x}^{(k)} \right\|_2^2} \right\|_2 = \left\| (\mathbf{I} - \mathbf{J}_k^{-1} \mathbf{J}) \frac{\Delta \mathbf{x}^{(k)} (\Delta \mathbf{x}^{(k)})^T}{\left\| \Delta \mathbf{x}^{(k)} \right\|_2^2} \right\|_2 \\ &\leq \left\| \mathbf{I} - \mathbf{J}_k^{-1} \mathbf{J} \right\|_2 . \end{aligned}$$

In conclusion,

(1.4.10) gives the $\|\cdot\|_2$ -minimal relative correction of \mathbf{J}_{k-1} , such that the secant condition (1.4.8) holds.

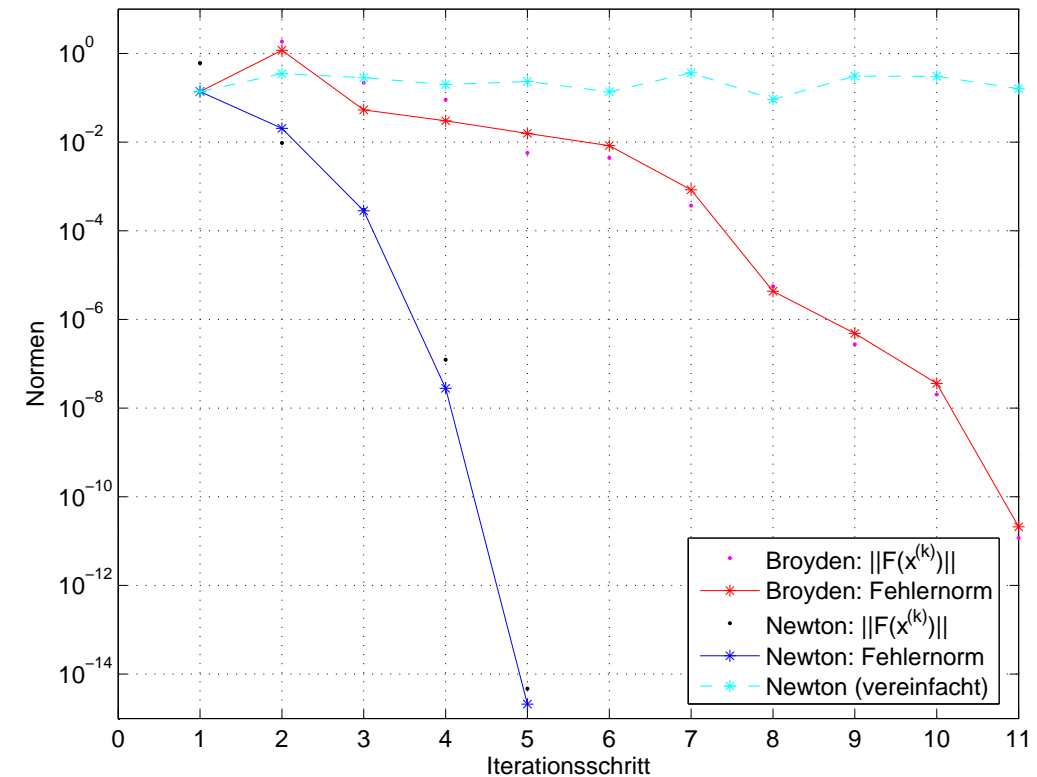
△

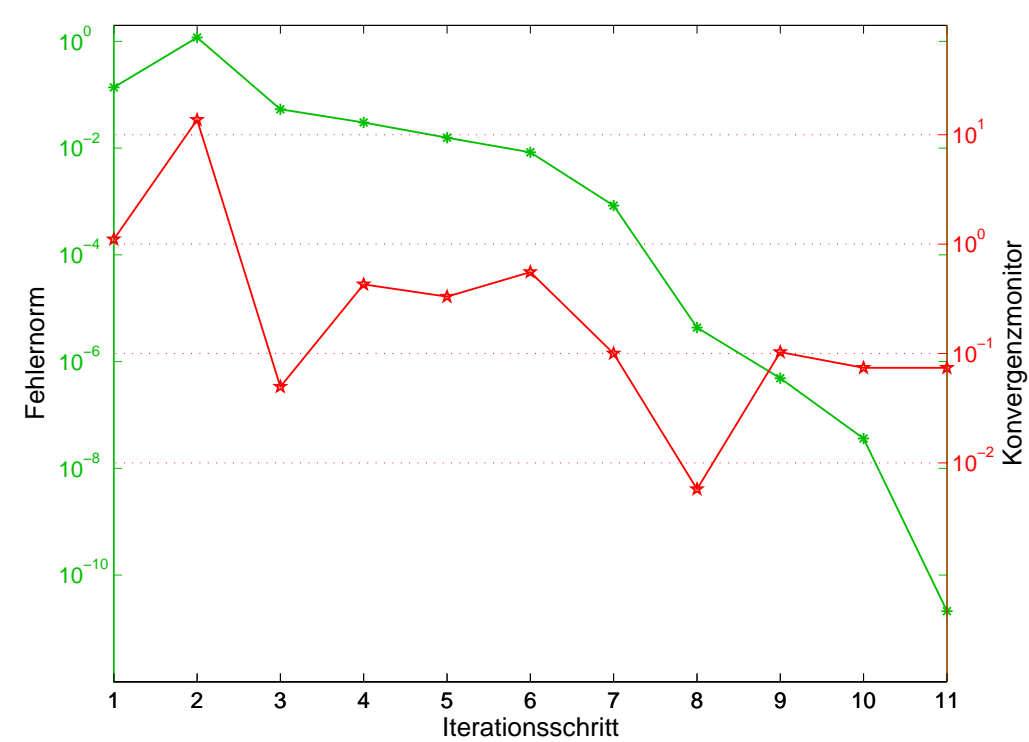
Example 1.4.17 (Broydens Quasi-Newton Method: Convergence).

- In the non-linear system of the example 1.4.1, $n = 2$ take $\mathbf{x}^{(0)} = (0.7, 0.7)^T$ and $\mathbf{J}_0 = DF(\mathbf{x}^{(0)})$

The numerical example shows that the method is:

slower than Newton method (1.4.1), but
better than simplified Newton method
(see remark. 1.4.5)





convergence monitor

=

quantity that displays difficulties in the convergence of an iteration

Here:

$$\mu := \frac{\left\| \mathbf{J}_{k-1}^{-1} F(\mathbf{x}^{(k)}) \right\|}{\left\| \Delta \mathbf{x}^{(k-1)} \right\|}$$

Heuristics: no convergence whenever $\mu > 1$



Gradinaru
D-MATH



Remark 1.4.18. Option: damped Broyden method (as for the Newton method, section 1.4.4)

Implementation of (1.4.11): with Sherman-Morrison-Woodbury Update-Formula

$$\mathbf{J}_{k+1}^{-1} = \left(\mathbf{I} - \frac{\mathbf{J}_k^{-1} F(\mathbf{x}^{(k+1)}) (\Delta \mathbf{x}^{(k)})^T}{\left\| \Delta \mathbf{x}^{(k)} \right\|_2^2 + \Delta \mathbf{x}^{(k)} \cdot \mathbf{J}_k^{-1} F(\mathbf{x}^{(k+1)})} \right) \mathbf{J}_k^{-1} = \left(\mathbf{I} + \frac{\Delta \mathbf{x}^{(k+1)} (\Delta \mathbf{x}^{(k)})^T}{\left\| \Delta \mathbf{x}^{(k)} \right\|_2^2} \right) \mathbf{J}_k^{-1} \quad (1.4.12)$$

that makes sense in the case that

$$\left\| \mathbf{J}_k^{-1} F(\mathbf{x}^{(k+1)}) \right\|_2 < \left\| \Delta \mathbf{x}^{(k)} \right\|_2$$

"simplified Quasi-Newton correction"

Code 1.4.19: Broyden method

```
from scipy.linalg import lu_solve, lu_factor, norm, solve
from scipy import dot, zeros
```

```
def fastbroyd(x0, F, J, tol=1e-12, maxit=20):
    x = x0.copy()
    lup = lu_factor(J)
    k = 0; s = lu_solve(lup, F(x))
    x -= s; f = F(x); sn = dot(s, s)
    dx = zeros((maxit, len(x)))
    dxn = zeros(maxit)
    dx[k] = s; dxn[k] = sn
    k += 1; tol *= tol
    while sn > tol and k < maxit:
        w = lu_solve(lup, f)
```

```

for r in xrange(1,k):
    w += dx[r]*( dot(dx[r-1],w) )/dxn[r-1]
z = dot(s,w)
s = (1+z/(sn-z))*w
sn = dot(s,s)
dx[k] = s; dxn[k] = sn
x -= s; f = F(x); k+=1

return x, k

```

Computational cost :

- $O(N^2 \cdot n)$ operations with vectors, (Level I)

N steps

- 1 LU-decomposition of \mathcal{J} , $N \times$ solutions of SLEs, see section ??
- N evaluations of F !

Memory cost :

- LU-factors of \mathcal{J} + auxiliary vectors $\in \mathbb{R}^n$

N steps

- N vectors $\mathbf{x}^{(k)} \in \mathbb{R}^n$

Example 1.4.20 (Broyden method for a large non-linear system).

$$F(\mathbf{x}) = \begin{cases} \mathbb{R}^n \mapsto \mathbb{R}^n \\ \mathbf{x} \mapsto \text{diag}(\mathbf{x})\mathbf{A}\mathbf{x} - \mathbf{b} \end{cases},$$

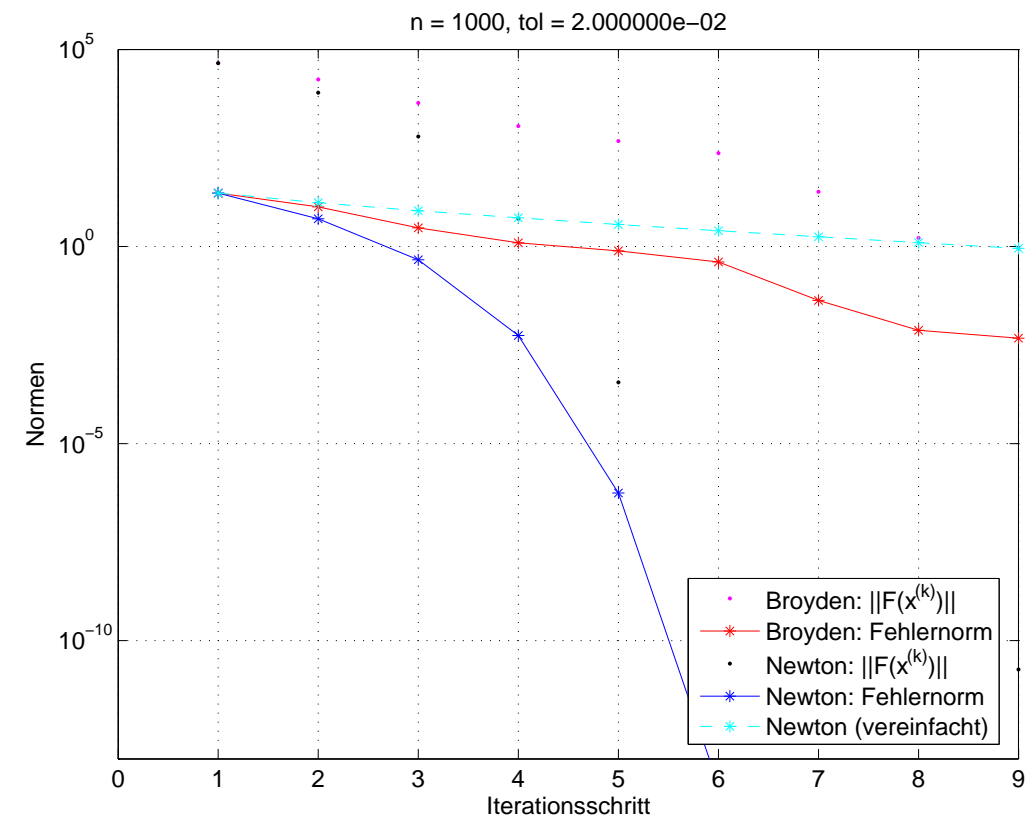
$$\mathbf{b} = (1, 2, \dots, n) \in \mathbb{R}^n,$$

$$\mathbf{A} = \mathbf{I} + \mathbf{a}\mathbf{a}^T \in \mathbb{R}^{n,n},$$

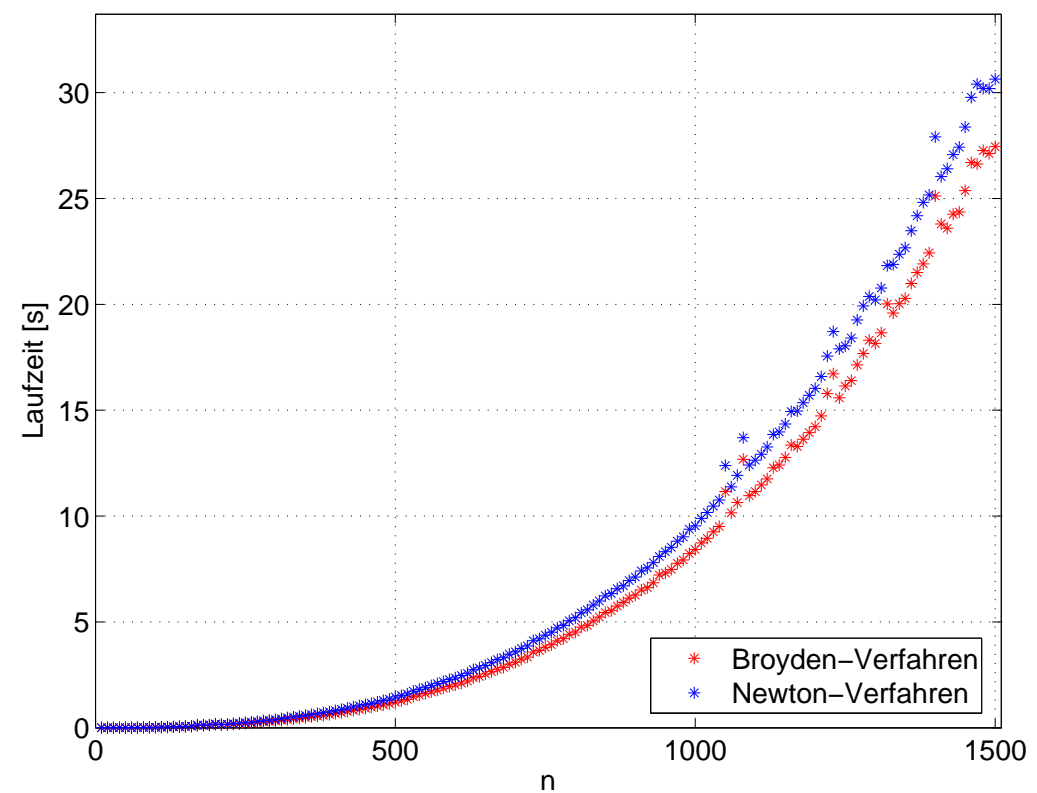
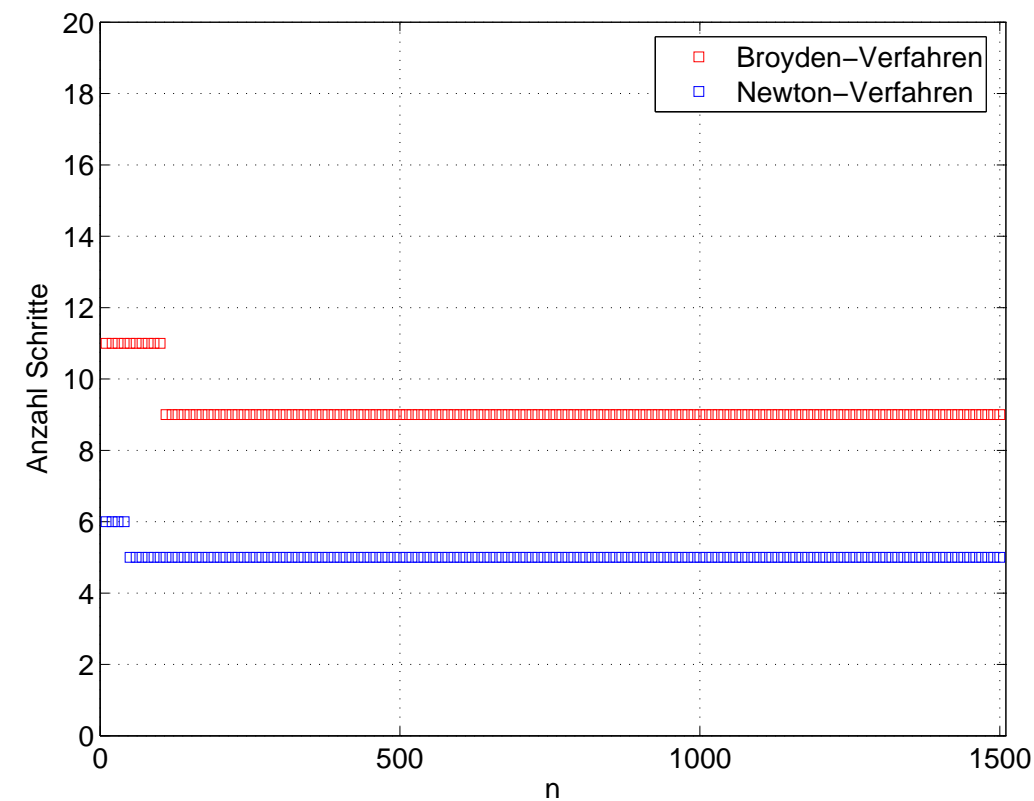
$$\mathbf{a} = \frac{1}{\sqrt{\mathbf{1} \cdot \mathbf{b} - 1}}(\mathbf{b} - \mathbf{1}).$$

The interpretation of the results resemble the example 1.4.17 \triangleright

$h = 2/n; \mathbf{x}_0 = (2:h:4-h)';$



Efficiency comparison: Broyden method \longleftrightarrow Newton method:
 (in case of dimension n use tolerance $\text{tol} = 2n \cdot 10^{-5}$, $h = 2/n; \mathbf{x}_0 = (2:h:4-h)';$)



In conclusion,
the Broyden method is worthwhile for dimensions $n \gg 1$ and low accuracy requirements.



1.5 Essential Skills Learned in Chapter 1

You should know:

- what is a linear convergent iteration, its rate and dependence of the choice of the norm
- what is the the order of convergence and how to recognize it from plots or from error data
- possible termination criteria and their risks
- how to use fixed-point iterations; convergence criteria
- bisection-method: pros and contras
- Newton-iteration: pros and contras
- the idea behind multi-point methods and an example
- how to use the Newton-method in several dimensions and how to reduce its computational effort (simplified Newton, quasi-Newton, Broyden method)

2

Intermezzo on (Numerical) Linear Algebra

Let A be an $n \times n$ -matrix.

Note that the result of the multiplication of the matrix A with a vector x is a linear combination of the columns of A :

$$Ax = \sum_{i=1}^n x_i A_{:,i}$$

Nonsingular	Singular
A is invertible	A is not invertible
The columns are independent	The columns are dependent
The rows are independent	The rows are dependent
$\det A \neq 0$	$\det A = 0$
$Ax = 0$ has one solution $x = 0$	$Ax = 0$ has infinitely many solutions
$Ax = b$ has one solution $x = A^{-1}b$	$Ax = b$ has no solution or infinitely many
A has full rank	A has rank $r < n$
A has n nonzero pivots	A has $r < n$ pivots
$\text{span}\{A_{:,1}, \dots, A_{:,n}\}$ has dimension n	$\text{span}\{A_{:,1}, \dots, A_{:,n}\}$ has dimension $r < n$
$\text{span}\{A_{1,:}, \dots, A_{n,:}\}$ has dimension n	$\text{span}\{A_{1,:}, \dots, A_{n,:}\}$ has dimension $r < n$
All eigenvalues of A are nonzero	0 is eigenvalue of A
$0 \notin \sigma(A) = \text{Spectrum of } A$	$0 \in \sigma(A)$
$A^H A$ is symmetric positive definite	$A^H A$ is only semidefinite
A has n (positive) singular values	A has $r < n$ (positive) singular values

Essential Decompositions

(1) Gaussian-elimination is nothing than LU-decomposition: $A = LU$ with lower triangular matrix L having ones on the main diagonal

Example 2.0.1 (Gaussian elimination and LU-factorization).

consider (forward) Gaussian elimination:


$$\begin{pmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ -3 \end{pmatrix} \iff \begin{aligned} x_1 + x_2 &= 4 \\ 2x_1 + x_2 - x_3 &= 1 \\ 3x_1 - x_2 - x_3 &= -3 \end{aligned}.$$

$$\begin{aligned} &\begin{pmatrix} 1 & & \\ & 1 & \\ & & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{pmatrix} \begin{pmatrix} 4 \\ 1 \\ -3 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & & \\ \color{red}{2} & 1 & \\ & 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \color{red}{0} & -1 & -1 \\ 3 & -1 & -1 \end{pmatrix} \begin{pmatrix} 4 \\ -7 \\ -3 \end{pmatrix} \\ &\Rightarrow \begin{pmatrix} 1 & & \\ 2 & 1 & \\ \color{red}{3} & 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 0 & -1 & -1 \\ \color{red}{0} & -4 & -1 \end{pmatrix} \begin{pmatrix} 4 \\ -7 \\ -15 \end{pmatrix} \Rightarrow \underbrace{\begin{pmatrix} 1 & & \\ 2 & 1 & \\ 3 & \color{red}{4} & 1 \end{pmatrix}}_{=L} \underbrace{\begin{pmatrix} 1 & 1 & 0 \\ 0 & \mathbf{-1} & \mathbf{-1} \\ 0 & \color{red}{0} & 3 \end{pmatrix}}_{=U} \begin{pmatrix} 4 \\ -7 \\ 13 \end{pmatrix} \end{aligned}$$

 = pivot row, pivot element **bold**, negative multipliers **red**


Perspective: link Gaussian elimination to **matrix factorization**
(row transformation = multiplication with elimination matrix)

$$a_1 \neq 0 \blacktriangleright \begin{pmatrix} 1 & 0 & \dots & \dots & 0 \\ -\frac{a_2}{a_1} & 1 & & & 0 \\ -\frac{a_3}{a_1} & & \ddots & & \\ \vdots & & & \ddots & \\ -\frac{a_n}{a_1} & 0 & & & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} a_1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$


 $n - 1$ steps of Gaussian elimination: \implies matrix factorization
 (non-zero pivot elements assumed)

$\mathbf{A} = \mathbf{L}_1 \cdots \mathbf{L}_{n-1} \mathbf{U}$ with elimination matrices $\mathbf{L}_i, i = 1, \dots, n - 1$,
 upper triangular matrix $\mathbf{U} \in \mathbb{R}^{n,n}$.

$$\begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ l_2 & 1 & & & 0 \\ l_3 & & \ddots & & \\ \vdots & & & \ddots & \\ l_n & 0 & & & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & & & 0 \\ 0 & h_3 & 1 & & \\ \vdots & \vdots & & \ddots & \\ 0 & h_n & 0 & & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ l_2 & 1 & & & 0 \\ l_3 & h_3 & 1 & & \\ \vdots & \vdots & & \ddots & \\ l_n & h_n & 0 & & 1 \end{pmatrix}$$

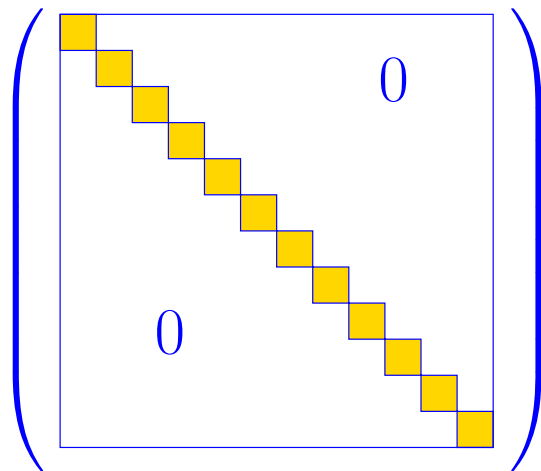

 $\mathbf{L}_1 \cdots \mathbf{L}_{n-1}$ are normalized lower triangular matrices
 (entries = multipliers $-\frac{a_{ik}}{a_{kk}}$)

Definition 2.0.1 (Types of matrices).

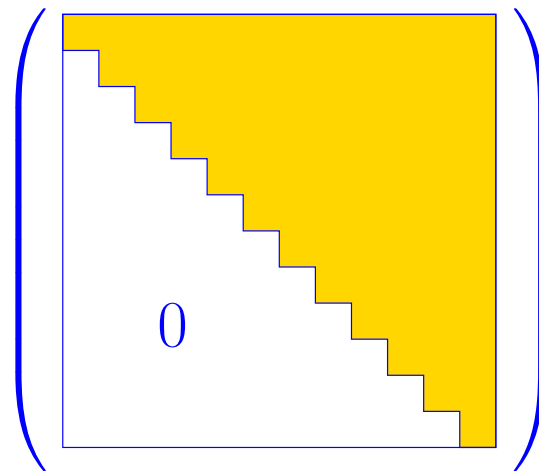
A matrix $\mathbf{A} = (a_{ij}) \in \mathbb{R}^{m,n}$ is

- **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 $i < j$.

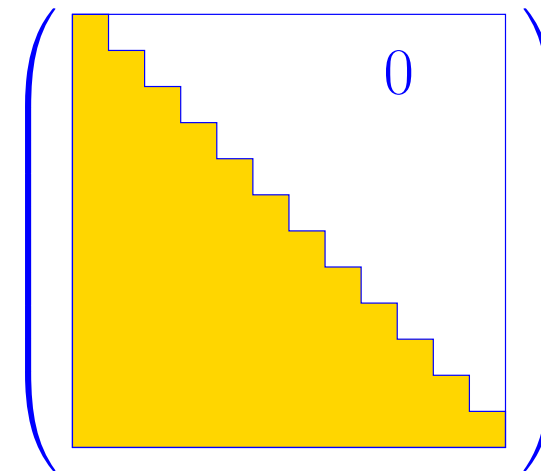
A triangular matrix is **normalized**, if $a_{ii} = 1, i = 1, \dots, \min\{m, n\}$.



diagonal matrix



upper triangular



lower triangular

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} .

Solving a linear system of equations by LU-factorization:

Algorithm 2.0.2 (Using LU-factorization to solve a linear system of equations).

① *LU*-decomposition $\mathbf{A} = \mathbf{L}\mathbf{U}$, #elementary operations $\frac{1}{3}n(n-1)(n+1)$

$\mathbf{Ax} = \mathbf{b}$: ② **forward substitution**, solve $\mathbf{Lz} = \mathbf{b}$, #elementary operations $\frac{1}{2}n(n-1)$

③ **backward substitution**, solve $\mathbf{Ux} = \mathbf{z}$, #elementary operations $\frac{1}{2}n(n+1)$

Stability needs pivoting:

Example 2.0.3.

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 2 \\ 2 & -7 & 2 \\ 1 & 24 & 0 \end{pmatrix} \xrightarrow{\textcircled{1}} \begin{pmatrix} 2 & -7 & 2 \\ 1 & 2 & 2 \\ 1 & 24 & 0 \end{pmatrix} \xrightarrow{\textcircled{2}} \begin{pmatrix} 2 & -7 & 2 \\ 0 & 5.5 & 1 \\ 0 & 27.5 & -1 \end{pmatrix} \xrightarrow{\textcircled{3}} \begin{pmatrix} 2 & -7 & 2 \\ 0 & 27.5 & -1 \\ 0 & 5.5 & 1 \end{pmatrix} \xrightarrow{\textcircled{4}} \begin{pmatrix} 2 & -7 & 2 \\ 0 & 27.5 & -1 \\ 0 & 0 & 1.2 \end{pmatrix}$$

$$\mathbf{U} = \begin{pmatrix} 2 & -7 & 2 \\ 0 & 27.5 & -1 \\ 0 & 0 & 1.2 \end{pmatrix}, \quad \mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0.2 & 1 \end{pmatrix}, \quad \mathbf{P} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}.$$



Lemma 2.0.2 (Existence of LU-factorization with pivoting).

For any regular $\mathbf{A} \in \mathbb{K}^{n,n}$ there is a permutation matrix $\mathbf{P} \in \mathbb{K}^{n,n}$, a normalized lower triangular matrix $\mathbf{L} \in \mathbb{K}^{n,n}$, and a regular upper triangular matrix $\mathbf{U} \in \mathbb{K}^{n,n}$, such that $\mathbf{PA} = \mathbf{LU}$.

python-function:

`LU, piv = scipy.linalg.lu_factor(A)`

`LU` = Matrix containing `U` in its upper triangle, and `L` in its lower triangle;

`piv` = pivot indices representing the permutation matrix `P`: row `i` of matrix was interchanged with row `piv[i]`

Round-off errors can be dangerous.

Definition 2.0.3 (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 $\|\cdot\|$!

- 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 $\text{cond}(\mathbf{A}) \gg 1$, an algorithm can produce solutions with large relative error !

(1') If \mathbf{A} is symmetric, then $\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^H = \mathbf{R}^H\mathbf{R}$ with diagonal matrix \mathbf{D} containing the pivots (Choleski-decomposition) and $\mathbf{R} = \sqrt{\mathbf{D}}\mathbf{L}^H$. No pivoting is necessary.

python-function:

`scipy.linalg.cho_factor(A)`

(2) Orthogonalisation: $A = QR$ with the matrix Q having orthonormal columns: $Q^H Q = Q Q^H = I$
(see below)

(3) Singular value decomposition (SVD):

$$A = U \Sigma V^H$$

where each of the matrices U and V have orthonormal columns,
 $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_r, 0, \dots, 0)$, $r = \text{rank}(A)$, $\sigma_1 \geq \sigma_2 \geq \sigma_r > 0$. Actually, $\sigma_1^2, \dots, \sigma_r^2$ are the eigenvalues of $A^H A$ (see below).

(4) Schur-decomposition:

$$\forall A \in \mathbb{K}^{n,n}: \quad \exists U \in \mathbb{C}^{n,n} \text{ unitary: } U^H A U = T \quad \text{with } T \in \mathbb{C}^{n,n} \text{ upper triangular .}$$

Remark 2.0.4. All presented python-functions are in fact wrappers to LAPACK Fortran- or C-routines.



Remark 2.0.5. Not discussed in this lecture, but of essential importance in applications are the **sparse matrices** (i.e. having the number of non-zero elements much smaller than n). Special storing schemes and algorithms can sometimes keep the factors L and U sparse, but in general this is difficult or impossible. For such cases, **iterative methods** for LSE (as e.g. preconditioned conjugate gradient) are the methods of choice.



2.1 QR-Factorization/QR-decomposition

Recall from linear algebra:

Definition 2.1.1 (Unitary and orthogonal matrices).

- $Q \in \mathbb{K}^{n,n}$, $n \in \mathbb{N}$, is **unitary**, if $Q^{-1} = Q^H$.
- $Q \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, is **orthogonal**, if $Q^{-1} = Q^T$.

Theorem 2.1.2 (Criteria for Unitarity).

$$Q \in \mathbb{C}^{n,n} \text{ unitary} \Leftrightarrow \|Q\mathbf{x}\|_2 = \|\mathbf{x}\|_2 \quad \forall \mathbf{x} \in \mathbb{K}^n.$$

► $Q \text{ unitary} \Rightarrow \text{cond}(Q) = 1$ (??) ► unitary transformations enhance (numerical) stability

If $Q \in \mathbb{K}^{n,n}$ unitary, then

- 🟢 all rows/columns (regarded as vectors $\in \mathbb{K}^n$) have Euclidean norm $= 1$,
- 🟢 all rows/columns are pairwise orthogonal (w.r.t. Euclidean inner product),
- 🟢 $|\det \mathbf{Q}| = 1$, and all eigenvalues $\in \{z \in \mathbb{C}: |z| = 1\}$.
- 🟢 $\|\mathbf{Q}\mathbf{A}\|_2 = \|\mathbf{A}\|_2$ for any matrix $\mathbf{A} \in \mathbb{K}^{n,m}$

Drawbacks of LU -factorization:

- 😞 often pivoting required (\rightarrow destroys structure, Ex. ??, leads to fill-in)
- 😞 Possible (theoretical) instability of partial pivoting \rightarrow Ex. ??

Stability problems of Gaussian elimination without pivoting are due to the fact that row transformations can convert well-conditioned matrices to ill-conditioned matrices, cf. Ex. ??

Which bijective row transformations preserve the Euclidean condition number of a matrix ?

➤ transformations that preserve the Euclidean norm of a vector !

► Investigate algorithms that use orthogonal/unitary row transformations to convert a matrix to upper triangular form.

Goal: find unitary row transformation rendering certain matrix elements zero.

$$Q \left(\begin{array}{c} \text{yellow square} \end{array} \right) = \left(\begin{array}{c} \text{yellow square with first column zeroed out} \end{array} \right) \quad \text{with} \quad Q^H = Q^{-1}.$$

This “annihilation of column entries” is the key operation in Gaussian forward elimination, where it is achieved by means of non-unitary row transformations, see Sect. ?? . Now we want to find a counterpart of Gaussian elimination based on unitary row transformations on behalf of numerical stability.

In 2D: two possible orthogonal transformations make 2nd component of $\mathbf{a} \in \mathbb{R}^2$ vanish:

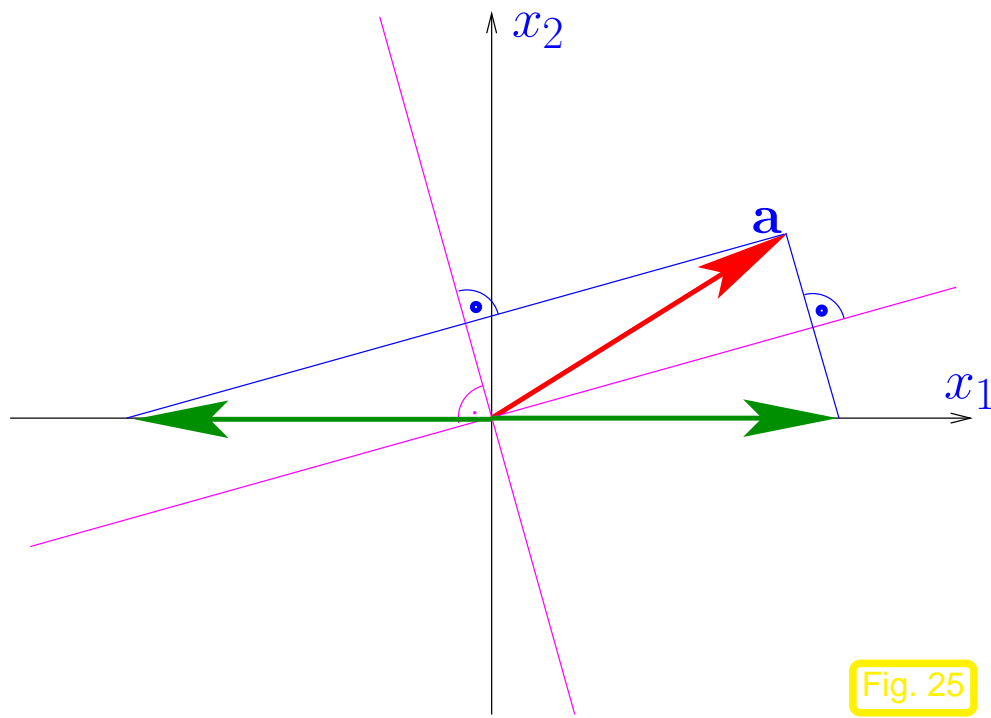


Fig. 25

reflection at angle bisector,

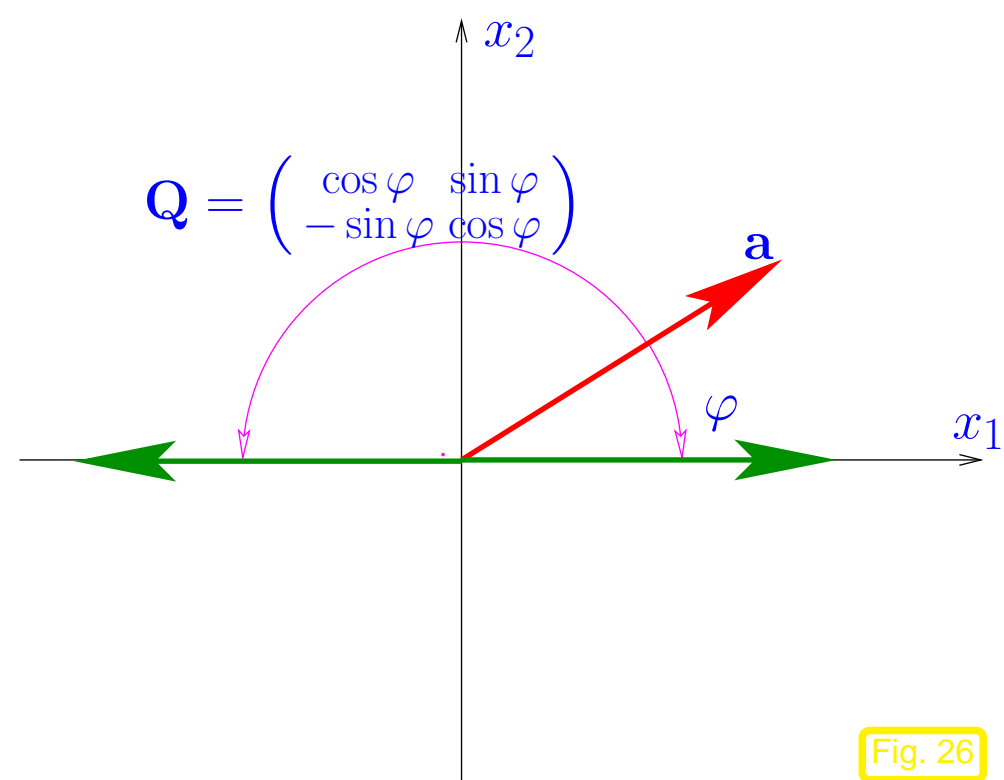


Fig. 26

rotation turning **a** onto x_1 -axis.

Note: two possible reflections/rotations

In n D: given $\mathbf{a} \in \mathbb{R}^n$ find orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{n,n}$ such that $\mathbf{Q}\mathbf{a} = \|\mathbf{a}\|_2 \mathbf{e}_1$, $\mathbf{e}_1 \hat{=}$ 1st unit vector.

Choice 1: Householder reflections

$$\mathbf{Q} = \mathbf{H}(\mathbf{v}) := \mathbf{I} - 2 \frac{\mathbf{v}\mathbf{v}^H}{\mathbf{v}^H \mathbf{v}} \quad \text{with} \quad \mathbf{v} = \frac{1}{2}(\mathbf{a} \pm \|\mathbf{a}\|_2 \mathbf{e}_1). \quad (2.1.1)$$

“Geometric derivation” of Householder reflection, see Figure 25

Given $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ with $\|\mathbf{a}\| = \|\mathbf{b}\|$, the difference vector $\mathbf{v} = \mathbf{b} - \mathbf{a}$ is orthogonal to the bisector.

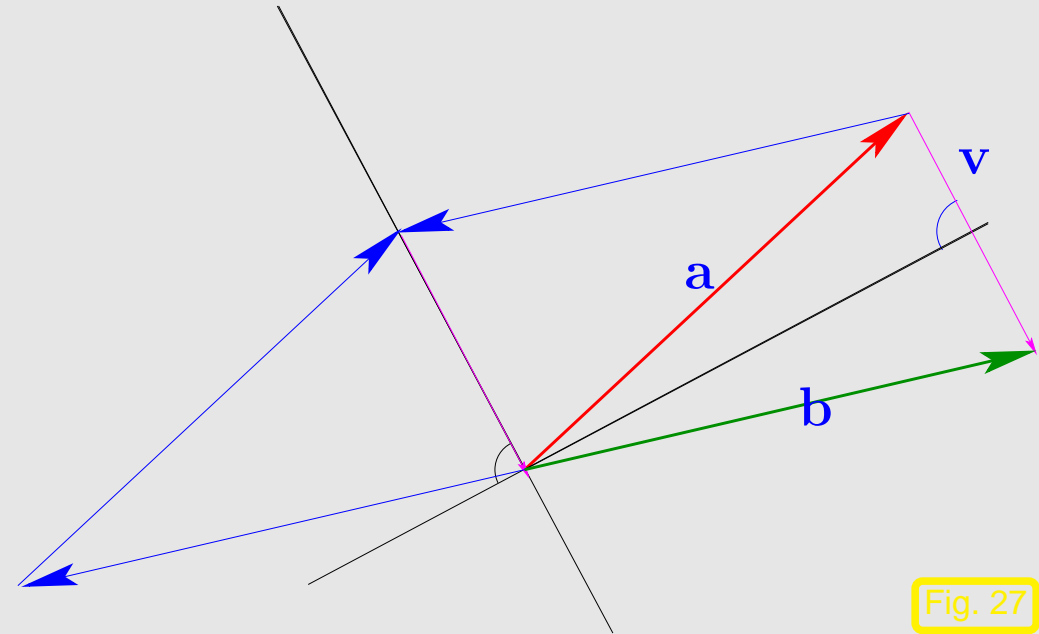


Fig. 27

$$\mathbf{b} = \mathbf{a} - (\mathbf{a} - \mathbf{b}) = \mathbf{a} - \mathbf{v} \frac{\mathbf{v}^T \mathbf{v}}{\mathbf{v}^T \mathbf{v}} = \mathbf{a} - 2\mathbf{v} \frac{\mathbf{v}^T \mathbf{a}}{\mathbf{v}^T \mathbf{v}} = \mathbf{a} - 2 \frac{\mathbf{v} \mathbf{v}^T}{\mathbf{v}^T \mathbf{v}} \mathbf{a} = \mathbf{H}(\mathbf{v}) \mathbf{a} ,$$

because, due to orthogonality $(\mathbf{a} - \mathbf{b}) \perp (\mathbf{a} + \mathbf{b})$

$$(\mathbf{a} - \mathbf{b})^T (\mathbf{a} - \mathbf{b}) = (\mathbf{a} - \mathbf{b})^T (\mathbf{a} - \mathbf{b} + \mathbf{a} + \mathbf{b}) = 2(\mathbf{a} - \mathbf{b})^T \mathbf{a} .$$

Remark 2.1.1 (Details of Householder reflections).

- Practice: for the sake of numerical stability (in order to avoid so-called *cancellation*) choose

$$\mathbf{v} = \begin{cases} \frac{1}{2}(\mathbf{a} + \|\mathbf{a}\|_2 \mathbf{e}_1) & , \text{ if } a_1 > 0 , \\ \frac{1}{2}(\mathbf{a} - \|\mathbf{a}\|_2 \mathbf{e}_1) & , \text{ if } a_1 \leq 0 . \end{cases}$$

However, this is not really needed [30, Sect. 19.1] !

- If $\mathbb{K} = \mathbb{C}$ and $a_1 = |a_1| \exp(i\varphi)$, $\varphi \in [0, 2\pi[$, then choose

$$\mathbf{v} = \frac{1}{2}(\mathbf{a} \pm \|\mathbf{a}\|_2 \mathbf{e}_1 \exp(-i\varphi)) \quad \text{in (2.1.1).}$$

- efficient storage of Householder matrices \rightarrow [3]

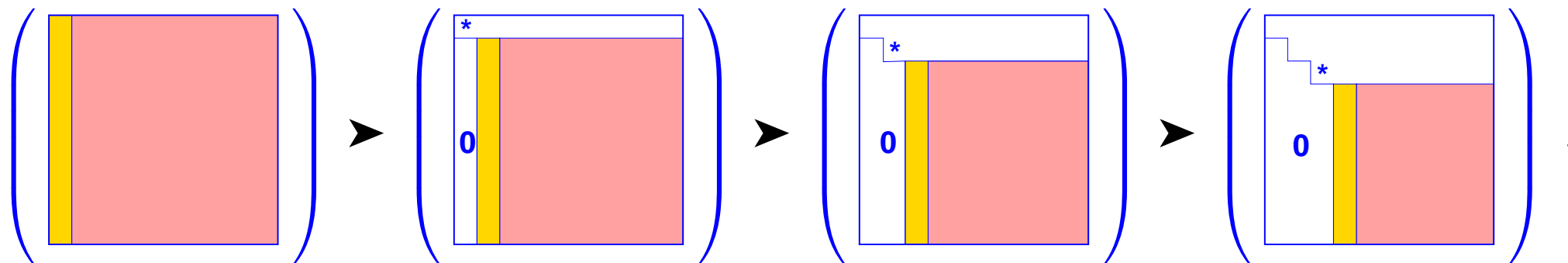


Choice 2: successive **Givens rotations** (\rightarrow 2D case)

$$\mathbf{G}_{1k}(a_1, a_k) \mathbf{A} := \begin{pmatrix} \bar{\gamma} & \cdots & \bar{\sigma} & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ -\sigma & \cdots & \gamma & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ \vdots \\ a_k \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} a_1^{(1)} \\ \vdots \\ 0 \\ \vdots \\ a_n \end{pmatrix}, \text{ if } \begin{aligned} \gamma &= \frac{a_1}{\sqrt{|a_1|^2 + |a_k|^2}}, \\ \sigma &= \frac{a_k}{\sqrt{|a_1|^2 + |a_k|^2}}. \end{aligned} \quad (2.1.2)$$



Transformation to *upper triangular form* by successive unitary transformations:



= “target column **a**” (determines unitary transformation),

= modified in course of transformations.



$$\mathbf{Q}_{n-1} \mathbf{Q}_{n-2} \cdots \mathbf{Q}_1 \mathbf{A} = \mathbf{R},$$

QR-factorization

(QR-decomposition)

of $\mathbf{A} \in \mathbb{C}^{n,n}$:

$$\mathbf{A} = \mathbf{Q}\mathbf{R},$$

$\mathbf{Q} := \mathbf{Q}_1^H \cdots \mathbf{Q}_{n-1}^H$ unitary matrix ,
 \mathbf{R} upper triangular matrix .

Num.
Meth.
Phys.

Generalization to $\mathbf{A} \in \mathbb{K}^{m,n}$:

$$m > n: \quad \left(\begin{array}{c} \mathbf{A} \end{array} \right) = \left(\begin{array}{c} \mathbf{Q} \end{array} \right) \left(\begin{array}{c} \mathbf{R} \end{array} \right), \quad \mathbf{A} = \mathbf{Q}\mathbf{R}, \quad \begin{array}{l} \mathbf{Q} \in \mathbb{K}^{m,n}, \\ \mathbf{R} \in \mathbb{K}^{n,n}, \end{array}$$

(2.1.3)

where $\mathbf{Q}^H \mathbf{Q} = \mathbf{I}$ (orthonormal columns), \mathbf{R} upper triangular matrix.

Gradinaru
D-MATH

Lemma 2.1.3 (Uniqueness of QR-factorization).

The “economical” QR-factorization (2.1.3) of $\mathbf{A} \in \mathbb{K}^{m,n}$, $m \geq n$, with $\text{rank}(\mathbf{A}) = n$ is unique, if we demand $r_{ii} > 0$.

Proof. we observe that \mathbf{R} is regular, if \mathbf{A} has full rank n . Since the regular upper triangular matrices form a group under multiplication:

$$\mathbf{Q}_1 \mathbf{R}_1 = \mathbf{Q}_2 \mathbf{R}_2 \Rightarrow \mathbf{Q}_1 = \mathbf{Q}_2 \mathbf{R} \text{ with upper triangular } \mathbf{R} := \mathbf{R}_2 \mathbf{R}_1^{-1} .$$

$$\blacktriangleright \quad \mathbf{I} = \mathbf{Q}_1^H \mathbf{Q}_1 = \mathbf{R}^H \underbrace{\mathbf{Q}_2^H \mathbf{Q}_2}_{=\mathbf{I}} \mathbf{R} = \mathbf{R}^H \mathbf{R} .$$

The assertion follows by uniqueness of Cholesky decomposition, Lemma ??.

□

$$m < n: \quad \left(\begin{array}{|c|} \hline \mathbf{A} \\ \hline \end{array} \right) = \left(\begin{array}{|c|} \hline \mathbf{Q} \\ \hline \end{array} \right) \left(\begin{array}{|c|} \hline \mathbf{R} \\ \hline \end{array} \right) ,$$

$$\mathbf{A} = \mathbf{QR} \quad , \quad \mathbf{Q} \in \mathbb{K}^{m,m} \quad , \quad \mathbf{R} \in \mathbb{K}^{m,n} ,$$

where \mathbf{Q} unitary, \mathbf{R} upper triangular matrix.

Remark 2.1.2 (Choice of unitary/orthogonal transformation).

When to use which unitary/orthogonal transformation for QR-factorization ?

- ▶ Householder reflections advantageous for fully populated target columns (dense matrices).
- ▶ Givens rotations more efficient (\leftarrow more selective), if target column sparsely populated.



functions:

$$Q, R = \text{qr}(A) \quad Q \in \mathbb{K}^{m,m}, R \in \mathbb{K}^{m,n} \text{ for } A \in \mathbb{K}^{m,n}$$

$$Q, R = \text{qr}(A, \text{econ}=\text{True}) \quad Q \in \mathbb{K}^{m,n}, R \in \mathbb{K}^{n,n} \text{ for } A \in \mathbb{K}^{m,n}, m > n$$

(economical QR-factorization)

Computational effort for Householder QR-factorization of $A \in \mathbb{K}^{m,n}$, $m > n$:

$$Q, R = \text{qr}(A) \quad \rightarrow \text{Costs: } O(m^2n)$$

$$Q, R = \text{qr}(A, \text{econ}=\text{True}) \quad \rightarrow \text{Costs: } O(mn^2)$$

Example 2.1.3 (Complexity of Householder QR-factorization).

```
1 from numpy import r_, mat, vstack, eye, ones, zeros
2 from scipy.linalg import qr
3 import timeit
4
5 def qr_full():
6     global A
7     Q, R = qr(A)
8
9 def qr_econ():
10    global A
11    Q, R = qr(A, econ=True)
12
13 def qr_ovecon():
14    global A
15    Q, R = qr(A, econ=True, overwrite_a=True)
16
17 def qr_r():
18    global A
19    R = qr(A, mode='r')
20
21 def qr_recon():
22    global A
23    R = qr(A, mode='r', econ=True)
```

```
24
25 nrEXP = 4
26 sizes = 2**r_[2:7]
27 qrtimes = zeros((5, sizes.shape[0]))
28 k = 0
29 for n in sizes:
30     print 'n=', n
31     m = n**2#4*n
32     A = mat(1.*r_[1:m+1]).T*mat(1.*r_[1:n+1])
33     A += vstack((eye(n), ones((m-n,n)) ))
34
35     t = timeit.Timer('qr_full()', 'from __main__ import qr_full')
36     avqr = t.timeit(number=nrEXP)/nrEXP
37     print avqr
38     qrtimes[0,k] = avqr
39
40     t = timeit.Timer('qr_econ()', 'from __main__ import qr_econ')
41     avqr = t.timeit(number=nrEXP)/nrEXP
42     print avqr
43     qrtimes[1,k] = avqr
44
45     t = timeit.Timer('qr_ovecon()', 'from __main__ import qr_ovecon')
46     avqr = t.timeit(number=nrEXP)/nrEXP
47     print avqr
```

```
48     qrtimes[2,k] = avqr
49
50     t = timeit.Timer('qr_r()', 'from __main__ import qr_r')
51     avqr = t.timeit(number=nrEXP)/nrEXP
52     print avqr
53     qrtimes[3,k] = avqr
54
55     t = timeit.Timer('qr_recon()', 'from __main__ import qr_recon')
56     avqr = t.timeit(number=nrEXP)/nrEXP
57     print avqr
58     qrtimes[4,k] = avqr
59
60     k += 1
61
62 #print qrtimes[3]
63 import matplotlib.pyplot as plt
64 plt.loglog(sizes, qrtimes[0], 's', label='qr')
65 plt.loglog(sizes, qrtimes[1], '*', label="qr(econ=True)")
66 plt.loglog(sizes, qrtimes[2], '.', label="qr(econ=True, _
        overwrite_a=True)")
67 plt.loglog(sizes, qrtimes[3], 'o', label="qr(mode='r')")
68 plt.loglog(sizes, qrtimes[4], '+', label="qr(mode='r', _econ=True)")
69 v4 = qrtimes[1,1]* (sizes/sizes[1])**4
70 v6 = qrtimes[0,1]* (sizes/sizes[1])**6
```



```
71 plt.loglog(sizes, v4, label='O(n^4)')
72 plt.loglog(sizes, v6, '—', label='O(n^6)')
73 plt.legend(loc=2)
74 plt.xlabel('n')
75 plt.ylabel('time_[s]')
76 plt.savefig('qrtiming.eps')
77 plt.show()
```

timing of different variants of QR-factorization

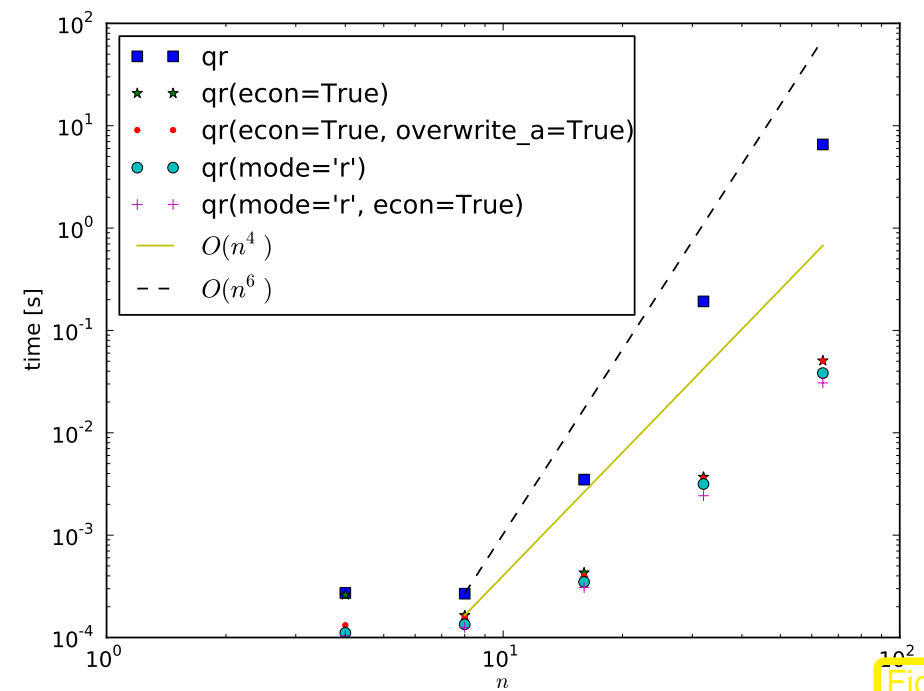


Fig. 28

Remark 2.1.5 (QR-orthogonalization).

$$\begin{pmatrix} \boxed{\mathbf{A}} \end{pmatrix} = \begin{pmatrix} \boxed{\mathbf{Q}} \end{pmatrix} \begin{pmatrix} \boxed{\mathbf{R}} \end{pmatrix}, \quad \mathbf{A}, \mathbf{Q} \in \mathbb{K}^{m,n}, \mathbf{R} \in \mathbb{K}^{n,n}.$$

If $m > n$, $\text{rank}(\mathbf{R}) = \text{rank}(\mathbf{A}) = n$ (full rank)

➤ $\{\mathbf{q}_{\cdot,1}, \dots, \mathbf{q}_{\cdot,n}\}$ is **orthonormal basis** of $\text{Im}(\mathbf{A})$ with
 $\text{Span}\{\mathbf{q}_{\cdot,1}, \dots, \mathbf{q}_{\cdot,k}\} = \text{Span}\{\mathbf{a}_{\cdot,1}, \dots, \mathbf{a}_{\cdot,k}\}, 1 \leq k \leq n.$



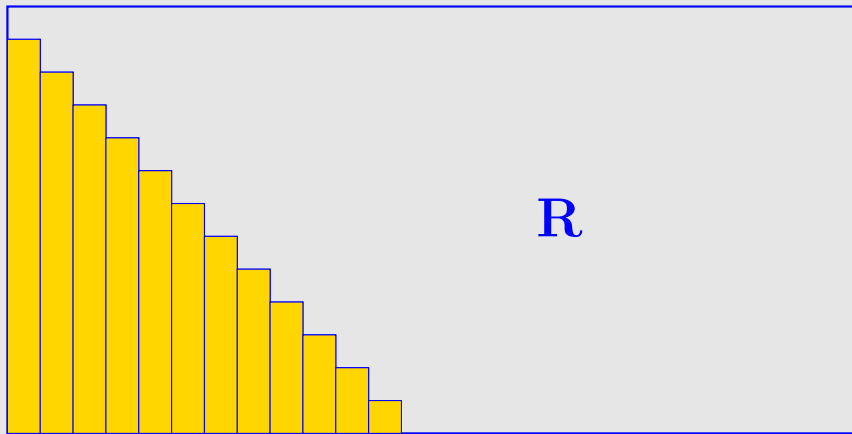
Remark 2.1.6 (Keeping track of unitary transformations).

How to store $\mathbf{G}_{i_1 j_1}(a_1, b_1) \cdots \mathbf{G}_{i_k j_k}(a_k, b_k), \quad ?$
 $\mathbf{H}(\mathbf{v}_1) \cdots \mathbf{H}(\mathbf{v}_k)$


☞ For Householder reflections

$\mathbf{H}(\mathbf{v}_1) \cdots \mathbf{H}(\mathbf{v}_k)$: store $\mathbf{v}_1, \dots, \mathbf{v}_k$

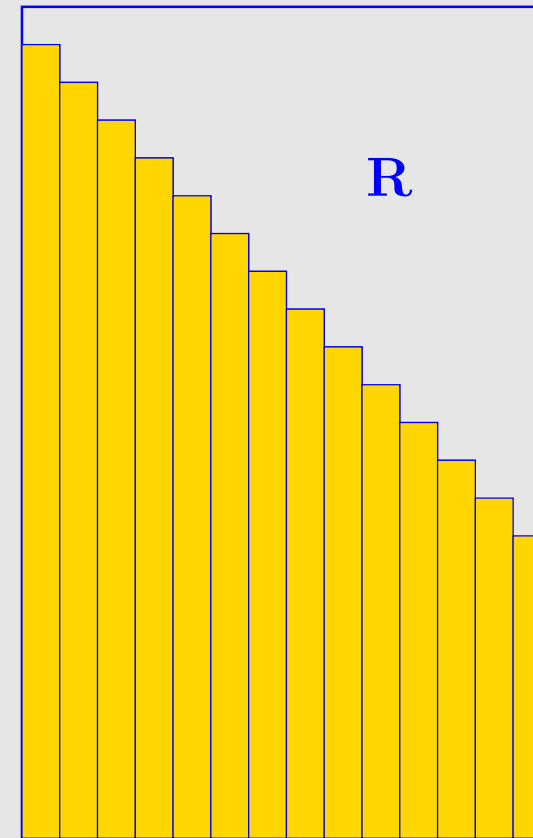
For in place QR-factorization of $\mathbf{A} \in \mathbb{K}^{m,n}$: store "Householder vectors" \mathbf{v}_j (decreasing size
 !) in lower triangle of \mathbf{A}



↑ Case $m < n$

 = Householder vectors

Case $m > n \rightarrow$



☞ Convention for Givens rotations ($\mathbb{K} = \mathbb{R}$)

$$\mathbf{G} = \begin{pmatrix} \bar{\gamma} & \bar{\sigma} \\ -\sigma & \gamma \end{pmatrix} \Rightarrow \text{store } \rho := \begin{cases} 1 & , \text{ if } \gamma = 0 , \\ \frac{1}{2} \text{sign}(\gamma)\sigma & , \text{ if } |\sigma| < |\gamma| , \\ 2 \text{sign}(\sigma)/\gamma & , \text{ if } |\sigma| \geq |\gamma| . \end{cases}$$

$$\blacktriangleright \begin{cases} \rho = 1 & \Rightarrow \gamma = 0 , \quad \sigma = 1 \\ |\rho| < 1 & \Rightarrow \sigma = 2\rho , \quad \gamma = \sqrt{1 - \sigma^2} \\ |\rho| > 1 & \Rightarrow \gamma = 2/\rho , \quad \sigma = \sqrt{1 - \gamma^2} . \end{cases}$$

Store $\mathbf{G}_{ij}(a, b)$ as triple (i, j, ρ)

Storing orthogonal transformation matrices is usually inefficient !

Algorithm 2.1.7 (Solving linear system of equations by means of QR-decomposition).

① QR-decomposition $\mathbf{A} = \mathbf{QR}$, computational costs $\frac{2}{3}n^3 + O(n^2)$ (about twice as expensive as LU -decomposition without pivoting)

$\mathbf{Ax} = \mathbf{b}$: ② orthogonal transformation $\mathbf{z} = \mathbf{Q}^H \mathbf{b}$, computational costs $4n^2 + O(n)$ (in the case of *compact storage* of reflections/rotations)

③ **Backward substitution**, solve $\mathbf{Rx} = \mathbf{z}$, computational costs $\frac{1}{2}n(n + 1)$

✌ Computing the generalized QR-decomposition $\mathbf{A} = \mathbf{QR}$ by means of Householder reflections or Givens rotations is (numerically stable) for any $\mathbf{A} \in \mathbb{C}^{m,n}$.

✌ For *any* regular system matrix an LSE can be solved by means of

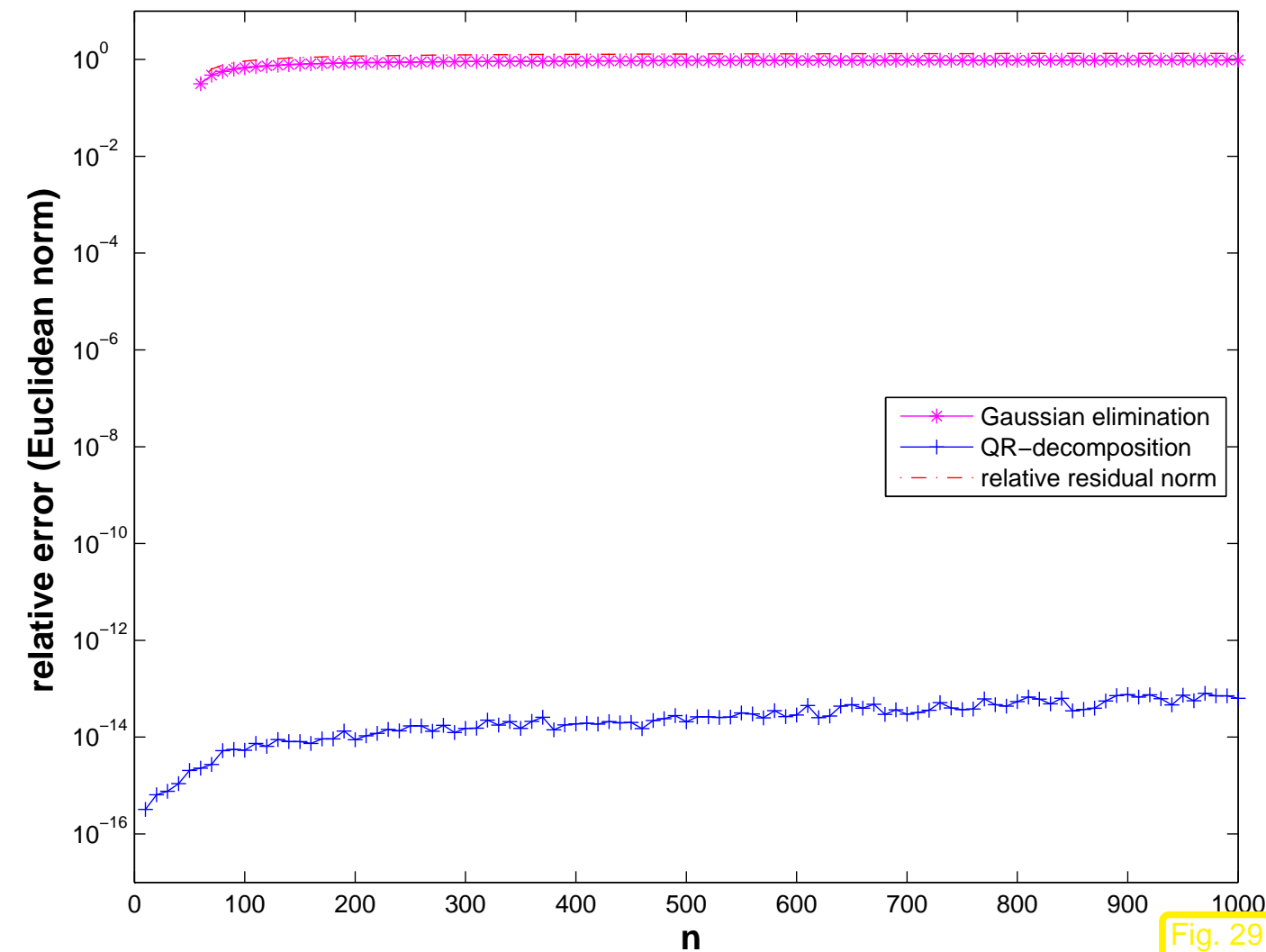
QR-decomposition + orthogonal transformation + backward substitution

in a stable manner.

Example 2.1.8 (Stable solution of LSE by means of QR-decomposition). → Ex. ??

Code 2.1.9: R-fac. \leftrightarrow Gaussian elimination

```
1 from numpy import tril , vstack , hstack , eye , zeros , ones , dot , r_  
2 from numpy.linalg import norm , solve , qr  
3 sizes = r_[10:1001:10]  
4 errlu = zeros(sizes.shape)  
5 errqr = zeros(sizes.shape)  
6 k = 0  
7 for n in sizes:  
8     A = -tril(ones((n,n-1))) + 2*vstack((eye(n-1),zeros(n-1)))  
9     A = hstack((A, ones((n,1))))  
10    x = (-1.)*r_[1:n+1]  
11    b = dot(A,x)  
12    Q, R = qr(A)  
13    errlu[k] = norm(solve(A,b)-x)/norm(x)  
14    errqr[k] = norm(solve(R,dot(Q.T.conj(),b))-x)/norm(x)  
15    k += 1  
16  
17 import matplotlib.pyplot as plt  
18 plt.semilogy(sizes, errlu, '*', label='LU')  
19 plt.semilogy(sizes, errqr, 'ro', label='QR')  
20 plt.legend(loc='center_right')  
21 plt.savefig('wilksolerr.eps')  
22 plt.show()
```



◁ superior stability of QR-decomposition !



Fill-in for QR-decomposition ?

bandwidth

$$\mathbf{A} \in \mathbb{C}^{n,n} \text{ with QR-decomposition } \mathbf{A} = \mathbf{Q}\mathbf{R} \Rightarrow m(\mathbf{R}) \leq m(\mathbf{A}) \quad (\rightarrow \text{Def. ??})$$



Example 2.1.10 (QR-based solution of tridiagonal LSE).

Elimination of Sub-diagonals by $n - 1$ successive Givens rotations:

$$\begin{pmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{pmatrix} \xrightarrow{\mathbf{G}_{12}} \begin{pmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{pmatrix} \xrightarrow{\mathbf{G}_{23}} \dots \xrightarrow{\mathbf{G}_{n-1,n}} \begin{pmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{pmatrix}$$

2.2 Singular Value Decomposition

Remark 2.2.1 (Principal component analysis (PCA)).

Given: n data points $\mathbf{a}_j \in \mathbb{R}^m$, $j = 1, \dots, n$, in m -dimensional (feature) space

Conjectured: “linear dependence”: $\mathbf{a}_j \in V$, $V \subset \mathbb{R}^m$ p -dimensional subspace,
 $p < \min\{m, n\}$ *unknown*
(\triangleright possibility of **dimensional reduction**)

Task (PCA): determine (minimal) p and (orthonormal basis of) V

Perspective of linear algebra:

Conjecture $\Leftrightarrow \text{rank}(\mathbf{A}) = p$ for $\mathbf{A} := (\mathbf{a}_1, \dots, \mathbf{a}_n) \in \mathbb{R}^{m,n}$, $\text{Im}(\mathbf{A}) = V$

Extension: Data affected by measurement errors
(but conjecture upheld for unperturbed data)

Application: \blacktriangleright Chemometrics (multivariate calibration methods for the analysis of chemical mixtures)



Theorem 2.2.1. 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 $\mathbf{\Sigma} = \text{diag}(\sigma_1, \dots, \sigma_p) \in \mathbb{R}^{m,n}$, $p := \min\{m, n\}$, $\sigma_1 \geq \sigma_2 \geq \sigma_p \geq 0$ such that

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H.$$

$(*)$: $\mathbf{\Sigma}$ (generalized) diagonal matrix $:\Leftrightarrow (\mathbf{\Sigma})_{i,j} = 0$, if $i \neq j$, $1 \leq i \leq m$, $1 \leq j \leq n$.

$$\left(\begin{array}{c} \mathbf{A} \end{array} \right) = \left(\begin{array}{c} \mathbf{U} \end{array} \right) \left(\begin{array}{c} \mathbf{\Sigma} \end{array} \right) \left(\begin{array}{c} \mathbf{V}^H \end{array} \right)$$

$$\left(\begin{array}{|c|} \hline \mathbf{A} \\ \hline \end{array} \right) = \left(\begin{array}{|c|} \hline \mathbf{U} \\ \hline \end{array} \right) \left(\begin{array}{|c|} \hline \begin{array}{c} \text{diagonal matrix } \Sigma \\ \text{with } \sigma_1, \dots, \sigma_r \text{ on the diagonal} \end{array} \\ \hline \end{array} \right) \left(\begin{array}{|c|} \hline \mathbf{V}^H \\ \hline \end{array} \right)$$

Proof. (of Thm. 2.2.1, by induction)

[52, Thm. 4.2.3]: Continuous functions attain extremal values on compact sets (here the unit ball $\{\mathbf{x} \in \mathbb{R}^n: \|\mathbf{x}\|_2 \leq 1\}$)

Gradinaru
D-MATH

$$\blacktriangleright \exists \mathbf{x} \in \mathbb{K}^n, \mathbf{y} \in \mathbb{K}^m, \quad \|\mathbf{x}\| = \|\mathbf{y}\|_2 = 1 : \quad \mathbf{A}\mathbf{x} = \sigma \mathbf{y}, \quad \sigma = \|\mathbf{A}\|_2,$$

where we used the definition of the matrix 2-norm, see Def. 1.1.12. 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

$$\mathbf{V} = (\mathbf{x} \ \tilde{\mathbf{V}}) \in \mathbb{K}^{n,n}, \quad \mathbf{U} = (\mathbf{y} \ \tilde{\mathbf{U}}) \in \mathbb{K}^{m,m} \quad \text{are unitary.}$$

$$\blacktriangleright \mathbf{U}^H \mathbf{A} \mathbf{V} = (\mathbf{y} \ \tilde{\mathbf{U}})^H \mathbf{A} (\mathbf{x} \ \tilde{\mathbf{V}}) = \left(\begin{array}{c|c} \mathbf{y}^H \mathbf{A} \mathbf{x} & \mathbf{y}^H \mathbf{A} \tilde{\mathbf{V}} \\ \hline \tilde{\mathbf{U}}^H \mathbf{A} \mathbf{x} & \tilde{\mathbf{U}}^H \mathbf{A} \tilde{\mathbf{V}} \end{array} \right) = \left(\begin{array}{c|c} \sigma & \mathbf{w}^H \\ \hline \mathbf{0} & \mathbf{B} \end{array} \right) =: \mathbf{A}_1.$$

$$\left\| \mathbf{A}_1 \begin{pmatrix} \sigma \\ \mathbf{w} \end{pmatrix} \right\|_2^2 = \left\| \begin{pmatrix} \sigma^2 + \mathbf{w}^H \mathbf{w} \\ \mathbf{B} \mathbf{w} \end{pmatrix} \right\|_2^2 = (\sigma^2 + \mathbf{w}^H \mathbf{w})^2 + \|\mathbf{B} \mathbf{w}\|_2^2 \geq (\sigma^2 + \mathbf{w}^H \mathbf{w})^2 ,$$

$$\|\mathbf{A}_1\|_2^2 = \sup_{0 \neq \mathbf{x} \in \mathbb{K}^n} \frac{\|\mathbf{A}_1 \mathbf{x}\|_2^2}{\|\mathbf{x}\|_2^2} \geq \frac{\|\mathbf{A}_1 \begin{pmatrix} \sigma \\ \mathbf{w} \end{pmatrix}\|_2^2}{\|\begin{pmatrix} \sigma \\ \mathbf{w} \end{pmatrix}\|_2^2} \geq \frac{(\sigma^2 + \mathbf{w}^H \mathbf{w})^2}{\sigma^2 + \mathbf{w}^H \mathbf{w}} = \sigma^2 + \mathbf{w}^H \mathbf{w} . \quad (2.2.1)$$

$$\sigma^2 = \|\mathbf{A}\|_2^2 = \left\| \mathbf{U}^H \mathbf{A} \mathbf{V} \right\|_2^2 = \|\mathbf{A}_1\|_2^2 \stackrel{(2.2.1)}{\implies} \|\mathbf{A}_1\|_2^2 = \|\mathbf{A}_1\|_2^2 + \|\mathbf{w}\|_2^2 \implies \mathbf{w} = 0 .$$

$$\blacktriangleright \quad \mathbf{A}_1 = \left(\begin{array}{c|c} \sigma & 0 \\ \hline 0 & \mathbf{B} \end{array} \right) .$$

Then apply induction argument to \mathbf{B}

□.

Definition 2.2.2 (Singular value decomposition (SVD)).

The decomposition $\mathbf{A} = \mathbf{U} \mathbf{\Sigma} \mathbf{V}^H$ of Thm. 2.2.1 is called *singular value decomposition (SVD)* of \mathbf{A} . The diagonal entries σ_i of $\mathbf{\Sigma}$ are the *singular values* of \mathbf{A} .

Lemma 2.2.3. The squares σ_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 4.1.4) to diagonal matrices with non-zero diagonal entries σ_i^2 ($\sigma_i \neq 0$), e.g.,

$$\mathbf{A} \mathbf{A}^H = \mathbf{U} \Sigma \mathbf{H}^H \mathbf{V} \Sigma^H \mathbf{U}^H = \mathbf{U} \underbrace{\Sigma \Sigma^H}_{\text{diagonal matrix}} \mathbf{U}^H. \quad \square$$

Remark 2.2.2 (SVD and additive rank-1 decomposition).

Recall from linear algebra: rank-1 matrices are tensor products of vectors

$$\mathbf{A} \in \mathbb{K}^{m,n} \quad \text{and} \quad \text{rank}(\mathbf{A}) = 1 \quad \Leftrightarrow \quad \exists \mathbf{u} \in \mathbb{K}^m, \mathbf{v} \in \mathbb{K}^n: \quad \mathbf{A} = \mathbf{u} \mathbf{v}^H, \quad (2.2.2)$$

because $\text{rank}(\mathbf{A}) = 1$ means that $\mathbf{A} \mathbf{x} = \mu(\mathbf{x}) \mathbf{u}$ 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:

$$\mathbf{A} = \mathbf{U} \Sigma \mathbf{V}^H = \sum_{j=1}^p \sigma_j (\mathbf{U})_{:,j} (\mathbf{V})_{:,j}^H. \quad (2.2.3)$$

Remark 2.2.3 (Uniqueness of SVD).

SVD of Def. 2.2.2 is not (necessarily) unique, but the singular values are.

Assume that \mathbf{A} has two singular value decompositions

$$\mathbf{A} = \mathbf{U}_1 \mathbf{\Sigma}_1 \mathbf{V}_1^H = \mathbf{U}_2 \mathbf{\Sigma}_2 \mathbf{V}_2^H \Rightarrow \mathbf{U}_1 \underbrace{\mathbf{\Sigma}_1 \mathbf{\Sigma}_1^H}_{=\text{diag}(s_1^1, \dots, s_m^1)} \mathbf{U}_1^H = \mathbf{A} \mathbf{A}^H = \mathbf{U}_2 \underbrace{\mathbf{\Sigma}_2 \mathbf{\Sigma}_2^H}_{=\text{diag}(s_1^2, \dots, s_m^2)} \mathbf{U}_2^H.$$

Two similar diagonal matrices are equal !

□

Gradinaru
D-MATH

△

Python-function: `scipy.linalg.svd`

`scipy.sparse.linalg.svds` in scipy 0.10

SVD on a large sparse matrix: package `divisi`

python-functions (for algorithms see [20, Sect. 8.3]):

$s = \text{svd}(A)$: computes singular values of matrix A
 $[U, S, V] = \text{svd}(A)$: computes singular value decomposition according to Thm. 2.2.1
 $[U, S, V] = \text{svd}(A, 0)$: “economical” singular value decomposition for $m > n$: : $U \in \mathbb{K}^{m,n}$, $\Sigma \in \mathbb{R}^{n,n}$, $V \in \mathbb{K}^{n,n}$
 $s = \text{svds}(A, k)$: k largest singular values (important for sparse $A \rightarrow$ Def. ??)
 $[U, S, V] = \text{svds}(A, k)$: partial singular value decomposition: $U \in \mathbb{K}^{m,k}$, $V \in \mathbb{K}^{n,k}$, $\Sigma \in \mathbb{R}^{k,k}$ diagonal with k largest singular values of A .

Explanation: “economical” singular value decomposition:

$$\begin{pmatrix} A \end{pmatrix} = \begin{pmatrix} U \end{pmatrix} \begin{pmatrix} \Sigma \end{pmatrix} \begin{pmatrix} V^H \end{pmatrix}$$

(python) algorithm for computing SVD is (numerically) stable

Complexity:

$$\begin{aligned} 2mn^2 + 2n^3 + O(n^2) + O(mn) & \text{ for } \mathbf{S} = \text{svd}(\mathbf{A}), \\ 4m^2n + 22n^3 + O(mn) + O(n^2) & \text{ for } [\mathbf{U}, \mathbf{S}, \mathbf{V}] = \text{svd}(\mathbf{A}), \\ O(mn^2) + O(n^3) & \text{ for } [\mathbf{U}, \mathbf{S}, \mathbf{V}] = \text{svd}(\mathbf{A}, 0), \quad m \gg n. \end{aligned}$$

- Application of SVD: computation of rank , kernel and range of a matrix

Lemma 2.2.4 (SVD and rank of a matrix).

If the singular values of \mathbf{A} satisfy $\sigma_1 \geq \dots \geq \sigma_r > \sigma_{r+1} = \dots = \sigma_p = 0$, then

- $\text{rank}(\mathbf{A}) = r$,
- $\text{Ker}(\mathbf{A}) = \text{Span} \{ (\mathbf{V})_{:,r+1}, \dots, (\mathbf{V})_{:,n} \}$,
- $\text{Im}(\mathbf{A}) = \text{Span} \{ (\mathbf{U})_{:,1}, \dots, (\mathbf{U})_{:,r} \}$.

Illustration:

$$\begin{pmatrix} \mathbf{A} \end{pmatrix} = \begin{pmatrix} \mathbf{U} \end{pmatrix} \begin{pmatrix} \begin{array}{c|c} \Sigma_r & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} \end{array} \end{pmatrix} \begin{pmatrix} \mathbf{V}^H \end{pmatrix} \quad (2.2.4)$$

columns = ONB of $\text{Im}(\mathbf{A})$

rows = ONB of $\text{Ker}(\mathbf{A})$

Remark: python function `r=rank(A)` relies on `svd(A)`

Lemma 2.2.4 \blacktriangleright PCA by SVD

❶ no perturbations:

SVD: $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^H$ satisfies $\sigma_1 \geq \sigma_2 \geq \dots \sigma_p > \sigma_{p+1} = \dots = \sigma_{\min\{m,n\}} = 0$,
 $V = \text{Span} \underbrace{\{(\mathbf{U})_{:,1}, \dots, (\mathbf{U})_{:,p}\}}_{\text{ONB of } V}$.

② with perturbations:

$$\text{SVD: } \mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H \text{ satisfies } \sigma_1 \geq \sigma_2 \geq \dots \sigma_p \gg \sigma_{p+1} \approx \dots \approx \sigma_{\min\{m,n\}} \approx 0, \\ V = \text{Span} \underbrace{\{(\mathbf{U})_{:,1}, \dots, (\mathbf{U})_{:,p}\}}_{\text{ONB of } V}.$$

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”.

Example 2.2.4 (Principal component analysis for data analysis).

$$\mathbf{A} \in \mathbb{R}^{m,n}, m \gg n:$$

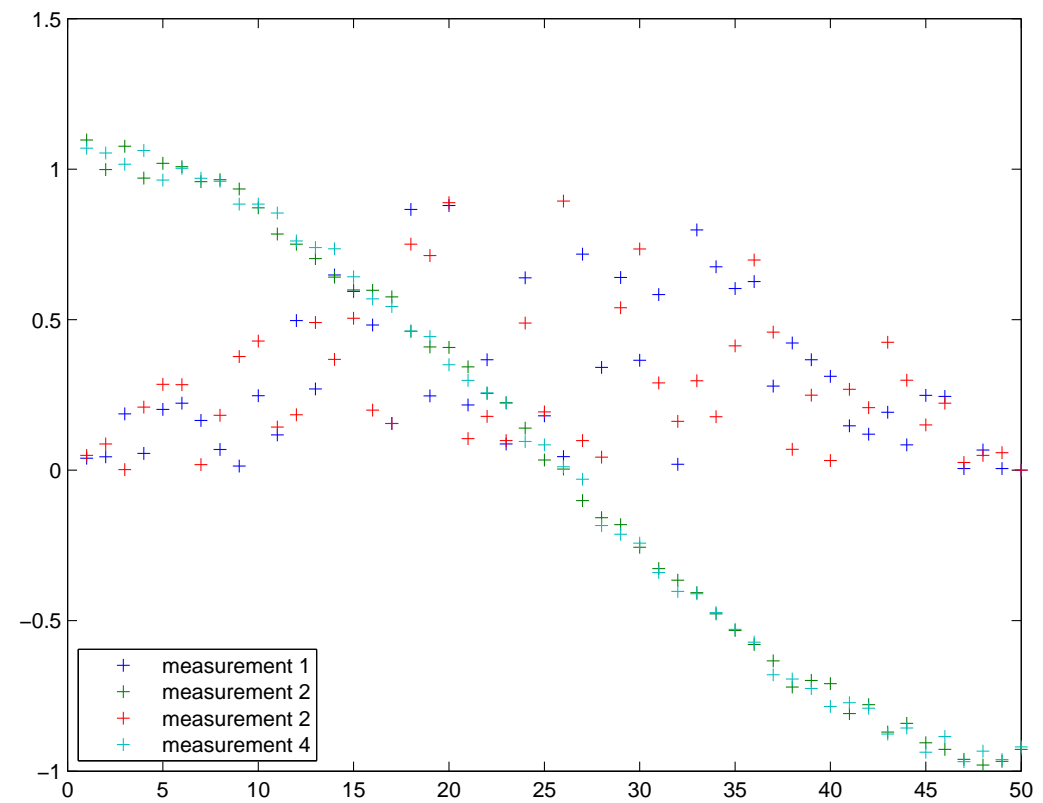
Columns \mathbf{A} \rightarrow series of measurements at different times/locations etc.
 Rows of \mathbf{A} \rightarrow measured values corresponding to one time/location etc.

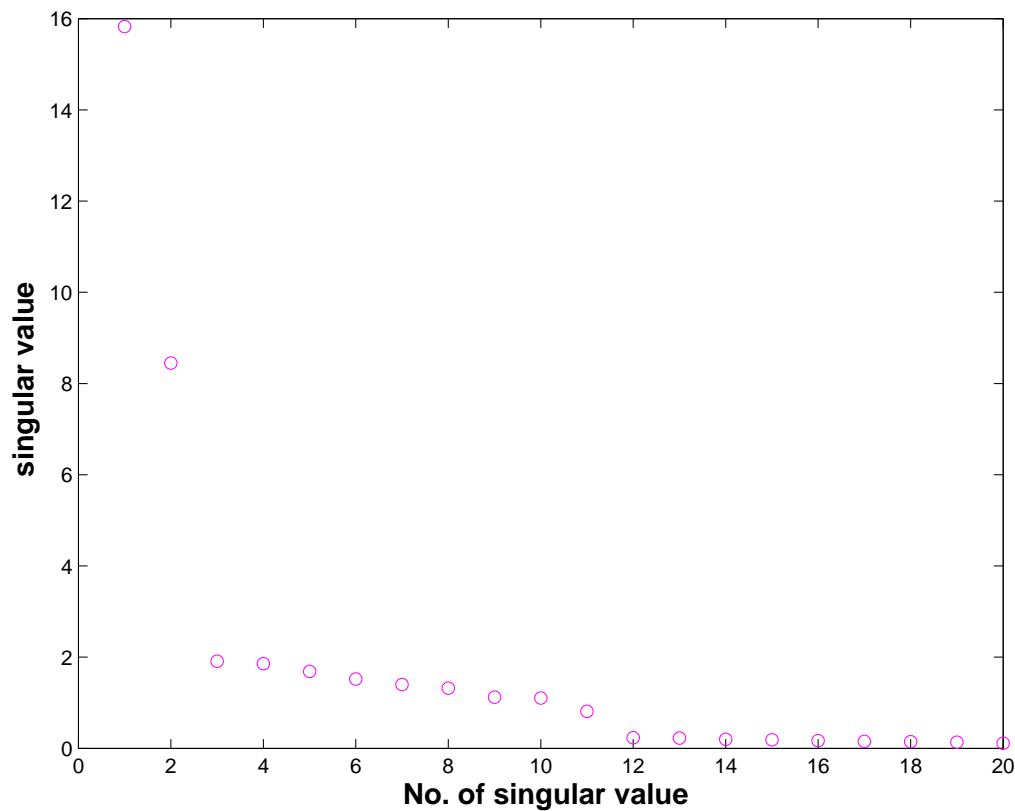
Goal: detect linear correlations

Concrete: two quantities measured over one year at 10 different sites

(Of course, measurements affected by errors/fluctuations)

```
n = 10; m = 50
r = linspace(1,m,m)
x = sin(pi*r/m)
y = cos(pi*r/m)
A = zeros((2*n,m))
for k in xrange(n):
    A[2*k] = x*rand(m)
    A[2*k+1] = y+0.1*rand(m)
```





← distribution of singular values of matrix

two dominant singular values !



measurements display linear correlation with **two**
principal components

principal components = $\mathbf{u}_{.,1}, \mathbf{u}_{.,2}$ (leftmost columns of \mathbf{U} -matrix of SVD)
 their relative weights = $\mathbf{v}_{.,1}, \mathbf{v}_{.,2}$ (leftmost columns of \mathbf{V} -matrix of SVD)

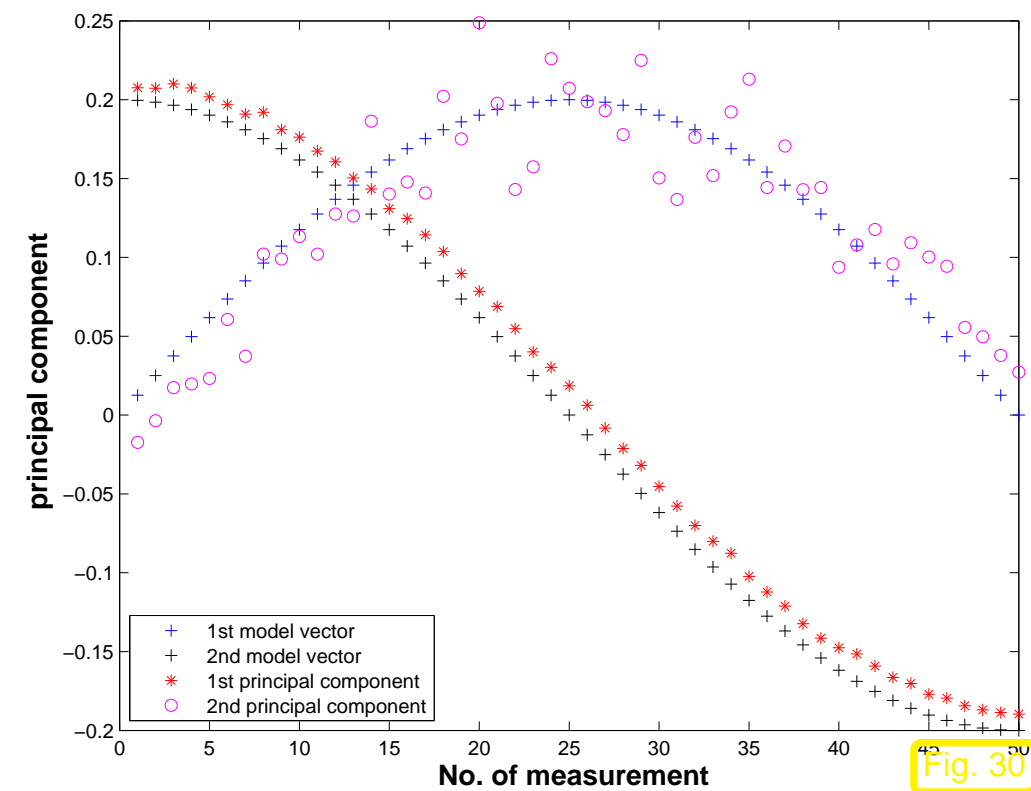


Fig. 30

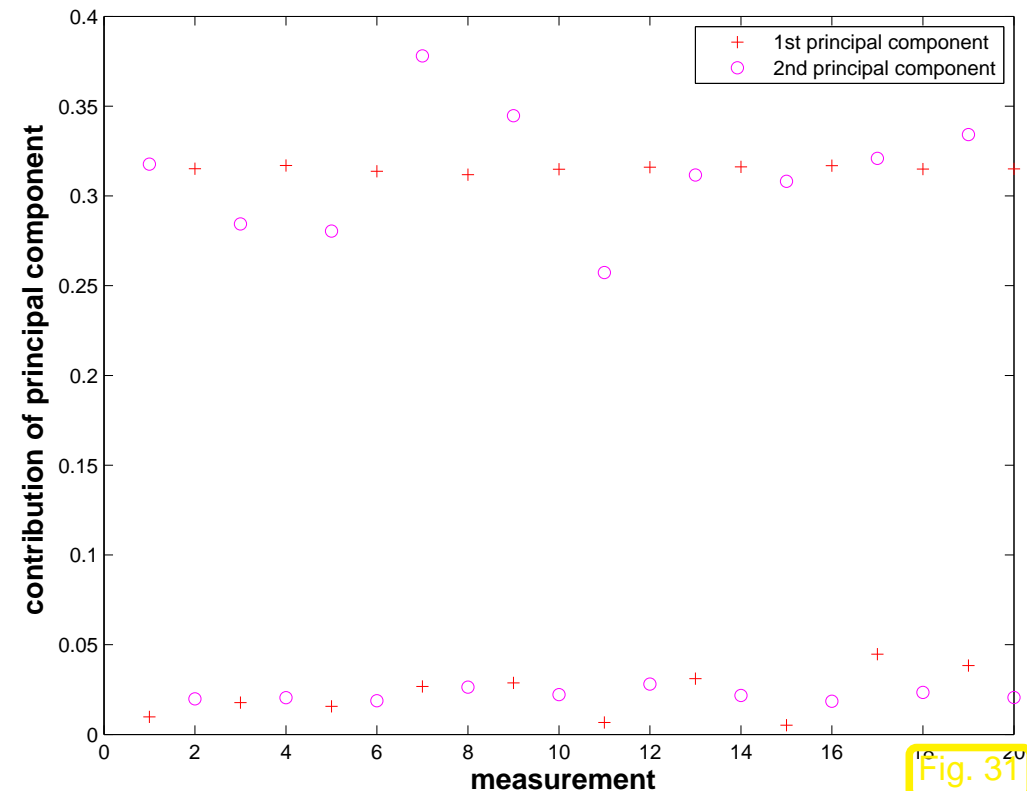


Fig. 31

• Application of SVD: extrema of quadratic forms on the unit sphere

A minimization problem on the Euclidean unit sphere $\{\mathbf{x} \in \mathbb{K}^n: \|\mathbf{x}\|_2 = 1\}$:

$$\text{given } \mathbf{A} \in \mathbb{K}^{m,n}, m > n, \text{ find } \mathbf{x} \in \mathbb{K}^n, \|\mathbf{x}\|_2 = 1, \quad \|\mathbf{A}\mathbf{x}\|_2 \rightarrow \min. \quad (2.2.5)$$

Use that multiplication with unitary matrices preserves the 2-norm and the singular value decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$ (\rightarrow Def. 2.2.2):

$$\min_{\|\mathbf{x}\|_2=1} \|\mathbf{A}\mathbf{x}\|_2^2 = \min_{\|\mathbf{x}\|_2=1} \|\mathbf{U}\mathbf{\Sigma}\mathbf{V}^H\mathbf{x}\|_2^2 = \min_{\|\mathbf{V}^H\mathbf{x}\|_2=1} \|\mathbf{U}\mathbf{\Sigma}(\mathbf{V}^H\mathbf{x})\|_2^2$$

$$= \min_{\|\mathbf{y}\|_2=1} \|\Sigma \mathbf{y}\|_2^2 = \min_{\|\mathbf{y}\|_2=1} (\sigma_1^2 y_1^2 + \cdots + \sigma_n^2 y_n^2) \geq \sigma_n^2 .$$

The minimum σ_n^2 is attained for $\mathbf{y} = \mathbf{e}_n \Rightarrow$ minimizer $\mathbf{x} = \mathbf{V}\mathbf{e}_n = (\mathbf{V})_{:,n}$.

By similar arguments:

$$\sigma_1 = \max_{\|\mathbf{x}\|_2=1} \|\mathbf{A}\mathbf{x}\|_2 \quad , \quad (\mathbf{V})_{:,1} = \operatorname{argmax}_{\|\mathbf{x}\|_2=1} \|\mathbf{A}\mathbf{x}\|_2 . \quad (2.2.6)$$

Recall: 2-norm of the matrix \mathbf{A} is defined as the maximum in (2.2.6). Thus we have proved the following theorem:

Lemma 2.2.5 (SVD and Euclidean matrix norm).

- $\forall \mathbf{A} \in \mathbb{K}^{m,n}$: $\|\mathbf{A}\|_2 = \sigma_1(\mathbf{A})$,
- $\forall \mathbf{A} \in \mathbb{K}^{n,n}$ *regular*: $\operatorname{cond}_2(\mathbf{A}) = \sigma_1/\sigma_n$.

Remark: functions `norm(A)` and `cond(A)` rely on `svd(A)`

Remark: Enhanced PCA in `matplotlib.mlab.PCA` and in the package MDA (Modular Toolkit for Data Processing)

2.3 Essential Skills Learned in Chapter 2

You should know:

- what is the QR-decomposition and possibilities to get it
- what is the singular value decomposition and how to use it
- applications of the svd: principal component analysis, extrema of quadratic forms on the unit sphere

3

Least Squares

Example 3.0.1 (linear regression).

Given: *measured data* y_i, \mathbf{x}_i , $y_i \in \mathbb{R}$, $\mathbf{x}_i \in \mathbb{R}^n$, $i = 1, \dots, m$, $m \geq n + 1$
(y_i, \mathbf{x}_i have measurement errors).

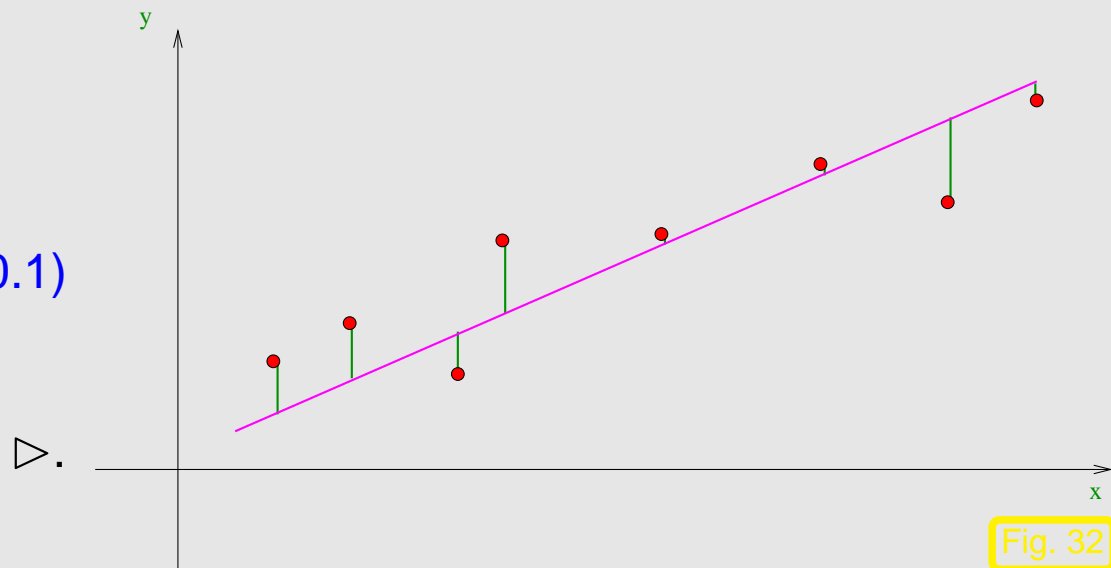
Known: without measurement errors data would satisfy
affine linear relationship $y = \mathbf{a}^T \mathbf{x} + c$, $\mathbf{a} \in \mathbb{R}^n$, $c \in \mathbb{R}$.

Goal: *estimate* parameters \mathbf{a} , c .

least squares estimate

$$(\mathbf{a}, c) = \underset{\mathbf{p} \in \mathbb{R}^n, q \in \mathbb{R}}{\operatorname{argmin}} \sum_{i=1}^m |y_i - \mathbf{p} \mathbf{x}_i - q|^2 \quad (3.0.1)$$

linear regression for $n = 2, m = 8$



Remark: In statistics we learn that the least squares estimate provides a maximum likelihood estimate, if the measurement errors are uniformly and independently normally distributed.

Example 3.0.2 (Linear data fitting). (→ Ex. 3.3.1 for a related problem)

Given: “nodes” $(t_i, y_i) \in \mathbb{K}^2$, $i = 1, \dots, m$, $t_i \in I \subset \mathbb{K}$,
basis functions $b_j : I \mapsto \mathbb{K}$, $j = 1, \dots, n$.
Find: coefficients $x_j \in \mathbb{K}$, $j = 1, \dots, n$, such that

$$\sum_{i=1}^m |f(t_i) - y_i|^2 \rightarrow \min \quad , \quad f(t) := \sum_{j=1}^n x_j b_j(t) . \quad (3.0.2)$$

Gradinaru
D-MATH

Special case: polynomial fit: $b_j(t) = t^{j-1}$.

MATLAB-function: `p = polyfit(t,y,n);` n = polynomial degree.



Remark 3.0.3 (Overdetermined linear systems).

In Ex. 3.0.1 we could try to find \mathbf{a} , c by solving the linear system of equations

$$\begin{pmatrix} \mathbf{x}_1^T & 1 \\ \vdots & \vdots \\ \mathbf{x}_m^T & 1 \end{pmatrix} \begin{pmatrix} \mathbf{a} \\ c \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix},$$

but in case $m > n + 1$ we encounter more equations than unknowns.

In Ex. 3.0.2 the same idea leads to the linear system

$$\begin{pmatrix} b_1(t_1) & \dots & b_n(t_1) \\ \vdots & & \vdots \\ b_1(t_m) & \dots & b_n(t_m) \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix},$$

with the same problem in case $m > n$.



(Linear) **least squares problem**:

given: $\mathbf{A} \in \mathbb{K}^{m,n}$, $m, n \in \mathbb{N}$, $\mathbf{b} \in \mathbb{K}^m$,

find: $\mathbf{x} \in \mathbb{K}^n$ such that

$$(i) \quad \|\mathbf{Ax} - \mathbf{b}\|_2 = \inf\{\|\mathbf{Ay} - \mathbf{b}\|_2 : \mathbf{y} \in \mathbb{K}^n\}, \quad (3.0.3)$$

(ii) $\|\mathbf{x}\|_2$ is minimal under the condition (i).

Recast as linear least squares problem, cf. Rem. 3.0.3:

$$\text{Ex. 3.0.1: } \mathbf{A} = \begin{pmatrix} \mathbf{x}_1^T & 1 \\ \vdots & \vdots \\ \mathbf{x}_m^T & 1 \end{pmatrix} \in \mathbb{R}^{m,n+1}, \quad \mathbf{b} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} \in \mathbb{R}^m, \quad \mathbf{x} = \begin{pmatrix} \mathbf{a} \\ c \end{pmatrix} \in \mathbb{R}^{n+1}.$$

$$\text{Ex. 3.0.2: } \mathbf{A} = \begin{pmatrix} b_1(t_1) & \dots & b_n(t_1) \\ \vdots & & \vdots \\ b_1(t_m) & \dots & b_n(t_m) \end{pmatrix} \in \mathbb{R}^{m,n}, \quad \mathbf{b} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} \in \mathbb{R}^m, \quad \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n.$$

In both cases the residual norm $\|\mathbf{b} - \mathbf{Ax}\|_2$ allows to gauge the quality of the model.

Lemma 3.0.1 (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 (3.2.4) and its derivation, see Sect. 3.2. □

`scipy.linalg.lstsq(A,b)` Reassuring: stable (\rightarrow Def.??) implementation (for dense matrices).

Remark 3.0.4 (Pseudoinverse).

By Lemma 3.0.1 the solution operator of the least squares problem (3.0.3) defines a linear mapping $\mathbf{b} \mapsto \mathbf{x}$, which has a matrix representation.

Definition 3.0.2 (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 (3.0.3) $\|\mathbf{Ax} - \mathbf{b}\| \rightarrow \min, \|\mathbf{x}\| \rightarrow \min$.

`scipy.linalg.pinv(A)` computes the pseudoinverse.



Remark 3.0.5 (Conditioning of the least squares problem).

Definition 3.0.3 (Generalized condition (number) of a matrix, \rightarrow Def. 2.0.3).

Let $\sigma_1 \geq \sigma_2 \geq \sigma_r > \sigma_{r+1} = \dots = \sigma_p = 0$, $p := \min\{m, n\}$, be the singular values (\rightarrow Def. 2.2.2) of $\mathbf{A} \in \mathbb{K}^{m,n}$. Then

$$\kappa(\mathbf{A}) = \frac{\sigma_1}{\sigma_r}$$

Theorem 3.0.4. 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

$$\frac{\|\mathbf{x} - \hat{\mathbf{x}}\|_2}{\|\mathbf{x}\|_2} \leq \left(2 \text{cond}_2(\mathbf{A}) + \text{cond}_2^2(\mathbf{A}) \frac{\|\mathbf{r}\|_2}{\|\mathbf{A}\|_2 \|\mathbf{x}\|_2} \right) \frac{\|\Delta\mathbf{A}\|_2}{\|\mathbf{A}\|_2}$$

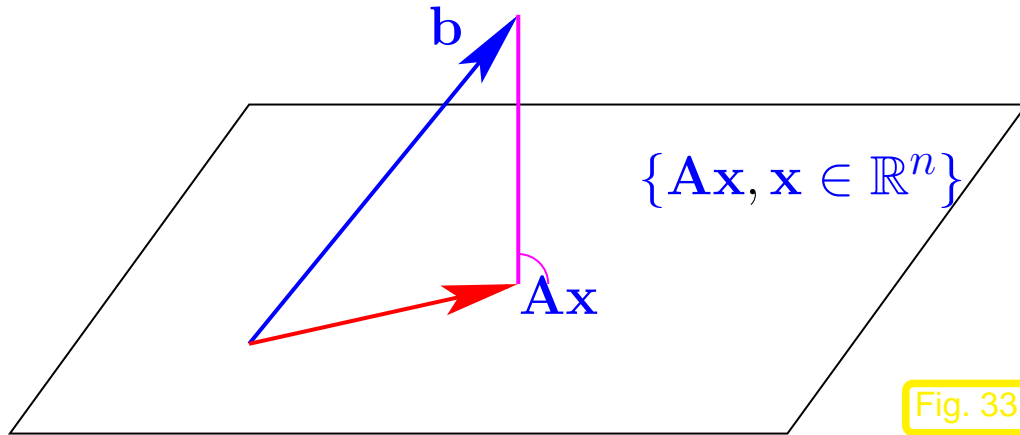
holds, where $\mathbf{r} = \mathbf{Ax} - \mathbf{b}$ is the residual.

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 $\approx \text{cond}_2^2(\mathbf{A})$



3.1 Normal Equations

Setting: $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, with **full rank** $\text{rank}(\mathbf{A}) = n$.



Geometric interpretation of linear least squares problem (3.0.3):

$\mathbf{x} \hat{=}$ orthogonal projection of \mathbf{b} on the subspace $\text{Im}(\mathbf{A}) := \text{Span} \{(\mathbf{A})_{:,1}, \dots, (\mathbf{A})_{:,n}\}$.

Gradinaru
D-MATH

Geometric interpretation: the least squares problem (3.0.3) 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. 33: \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 $\mathbf{p} = \mathbf{A}\mathbf{x}$ leads to normal equations (3.1.2).

Solve (3.0.3) for $\mathbf{b} \in \mathbb{R}^m$

$$\mathbf{x} \in \mathbb{R}^n: \quad \|\mathbf{Ax} - \mathbf{b}\|_2 \rightarrow \min \quad \Leftrightarrow \quad f(\mathbf{x}) := \|\mathbf{Ax} - \mathbf{b}\|_2^2 \rightarrow \min . \quad (3.1.1)$$

A quadratic functional, cf. (??)

$$f(\mathbf{x}) = \|\mathbf{Ax} - \mathbf{b}\|_2^2 = \mathbf{x}^H (\mathbf{A}^H \mathbf{A}) \mathbf{x} - 2\mathbf{b}^H \mathbf{Ax} + \mathbf{b}^H \mathbf{b} .$$

Minimization problem for $f \succ$ study gradient, cf. (??)

$$\text{grad } f(\mathbf{x}) = 2(\mathbf{A}^H \mathbf{A})\mathbf{x} - 2\mathbf{A}^H \mathbf{b} .$$

$$\text{grad } f(\mathbf{x}) \stackrel{!}{=} 0: \quad \boxed{\mathbf{A}^H \mathbf{Ax} = \mathbf{A}^H \mathbf{b}} \quad = \text{normal equation of (3.1.1)} \quad (3.1.2)$$

Gradinaru
D-MATH

Notice: $\text{rank}(\mathbf{A}) = n \Rightarrow \mathbf{A}^H \mathbf{A} \in \mathbb{R}^{n,n}$ s.p.d. (\rightarrow Def. ??)

Remark 3.1.1 (Conditioning of normal equations).

Caution: danger of instability, with SVD $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$

$$\text{cond}_2(\mathbf{A}^H \mathbf{A}) = \text{cond}_2(\mathbf{V}\mathbf{\Sigma}^H \mathbf{U}^H \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H) = \text{cond}_2(\mathbf{\Sigma}^H \mathbf{\Sigma}) = \frac{\sigma_1^2}{\sigma_n^2} = \text{cond}_2(\mathbf{A})^2 .$$

- For fairly ill-conditioned \mathbf{A} using the normal equations (3.1.2) to solve the linear least squares problem (3.1.1) 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** (\rightarrow Def. ??) of normal equation approach.

Example 3.1.2 (Instability of normal equations).

Caution: loss of information in the computation of $\mathbf{A}^H \mathbf{A}$, e.g.

$$\mathbf{A} = \begin{pmatrix} 1 & 1 \\ \delta & 0 \\ 0 & \delta \end{pmatrix} \Rightarrow \mathbf{A}^H \mathbf{A} = \begin{pmatrix} 1 + \delta^2 & 1 \\ 1 & 1 + \delta^2 \end{pmatrix}$$

```

1 from scipy import linalg
  as spla
2 import numpy as np
3 delta = 10**-6
4 A = np.array([[1, 1],
  [delta, 0], [0,
  delta]])
5 B = np.dot(A.T, A)
6 sA = spla.svdvals(A)
7 print 'rank(A)=',
  np.sum(sA > 1e-10)
8 sB = spla.svdvals(B)
9 print
  'rank(B)=', np.sum(sB >
  1e-10)
```

If $\delta < \sqrt{\text{eps}}$ \Rightarrow $1 + \delta^2 = 1$ in \mathbb{M} , i.e. $\mathbf{A}^H \mathbf{A}$ “numeric singular”, though $\text{rank}(\mathbf{A}) = 2$.



Another reason not to compute $\mathbf{A}^H \mathbf{A}$, when both m, n large:

$$\mathbf{A} \text{ sparse} \not\Rightarrow \mathbf{A}^T \mathbf{A} \text{ sparse}$$



- Potential memory overflow, when computing $\mathbf{A}^T \mathbf{A}$
- Squanders possibility to use efficient sparse direct elimination techniques, see Sect. ??

A way to avoid the computation of $\mathbf{A}^H \mathbf{A}$:

Expand normal equations (3.1.2): introduce residual $\mathbf{r} := \mathbf{A}\mathbf{x} - \mathbf{b}$ as new unknown:

$$\mathbf{A}^H \mathbf{A} \mathbf{x} = \mathbf{A}^H \mathbf{b} \Leftrightarrow \mathbf{B} \begin{pmatrix} \mathbf{r} \\ \mathbf{x} \end{pmatrix} := \begin{pmatrix} -\mathbf{I} & \mathbf{A} \\ \mathbf{A}^H & 0 \end{pmatrix} \begin{pmatrix} \mathbf{r} \\ \mathbf{x} \end{pmatrix} = \begin{pmatrix} \mathbf{b} \\ 0 \end{pmatrix}. \quad (3.1.3)$$

More general substitution $\mathbf{r} := \alpha^{-1}(\mathbf{Ax} - \mathbf{b})$, $\alpha > 0$ to improve the condition:

$$\mathbf{A}^H \mathbf{Ax} = \mathbf{A}^H \mathbf{b} \Leftrightarrow \mathbf{B}_\alpha \begin{pmatrix} \mathbf{r} \\ \mathbf{x} \end{pmatrix} := \begin{pmatrix} -\alpha \mathbf{I} & \mathbf{A} \\ \mathbf{A}^H & 0 \end{pmatrix} \begin{pmatrix} \mathbf{r} \\ \mathbf{x} \end{pmatrix} = \begin{pmatrix} \mathbf{b} \\ 0 \end{pmatrix}. \quad (3.1.4)$$

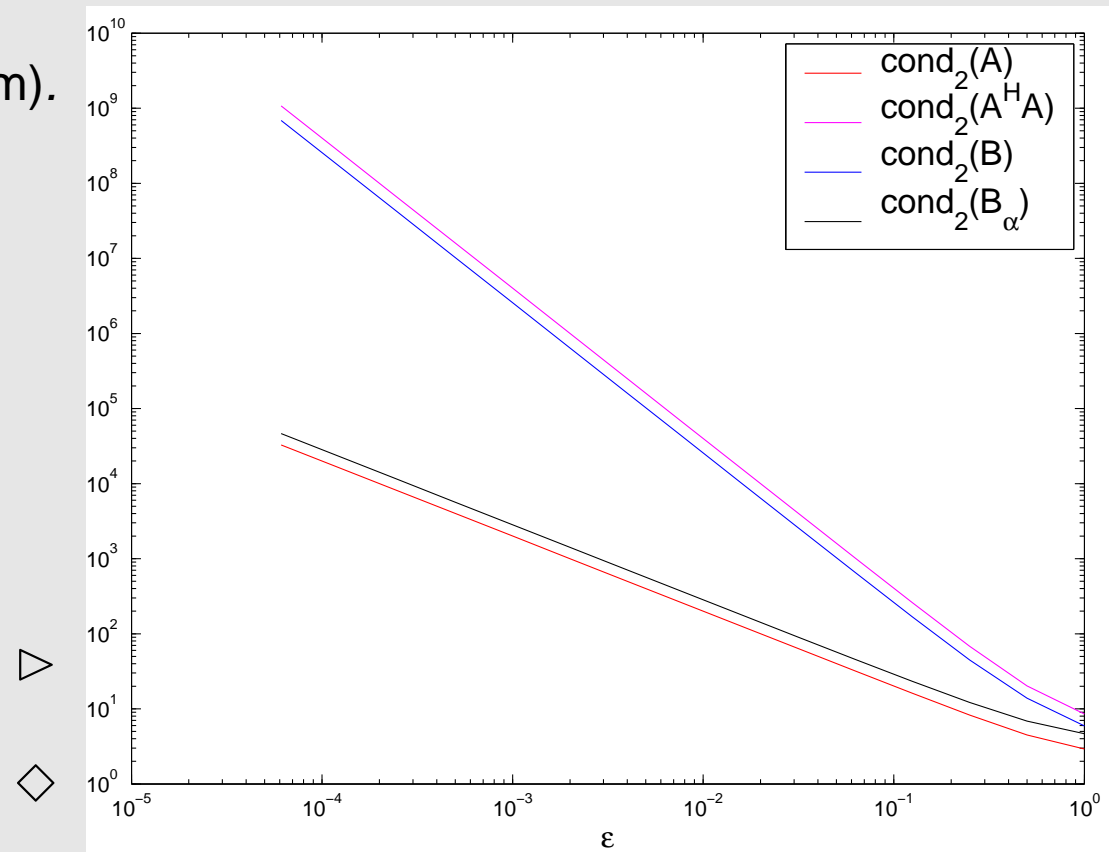
For $m, n \gg 1$, \mathbf{A} sparse, both (3.1.3) and (3.1.4) lead to large sparse linear systems of equations, amenable to sparse direct elimination techniques, see Sect. ??

Example 3.1.3 (Condition of the extended system).

Consider (3.1.3), (3.1.4) for

$$\mathbf{A} = \begin{pmatrix} 1 + \epsilon & 1 \\ 1 - \epsilon & 1 \\ \epsilon & \epsilon \end{pmatrix}.$$

Plot of different condition numbers
in dependence on ϵ
($\alpha = \epsilon \|\mathbf{A}\|_2 / \sqrt{2}$)



3.2 Orthogonal Transformation Methods

Consider the linear least squares problem (3.0.3)

$$\text{given } \mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{b} \in \mathbb{R}^m \quad \text{find } \mathbf{x} = \operatorname{argmin}_{\mathbf{y} \in \mathbb{R}^n} \|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2 .$$

Assumption: $m \geq n$ and \mathbf{A} has full (maximum) rank: $\operatorname{rank}(\mathbf{A}) = n$.

Recall Thm. 2.1.2: orthogonal (unitary) transformations (\rightarrow Def. 2.1.1) leave 2-norm invariant.

Idea: Transformation of $\mathbf{A}\mathbf{x} - \mathbf{b}$ to simpler form by *orthogonal* row transformations:

$$\operatorname{argmin}_{\mathbf{y} \in \mathbb{R}^n} \|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2 = \operatorname{argmin}_{\mathbf{y} \in \mathbb{R}^n} \|\tilde{\mathbf{A}}\mathbf{y} - \tilde{\mathbf{b}}\|_2 ,$$

where $\tilde{\mathbf{A}} = \mathbf{Q}\mathbf{A}$, $\tilde{\mathbf{b}} = \mathbf{Q}\mathbf{b}$ with orthogonal $\mathbf{Q} \in \mathbb{R}^{m,m}$.

As in the case of LSE (\rightarrow Sect. 2.1): “simpler form” = triangular form.

Concrete realization of this idea by means of **QR-decomposition** (\rightarrow Section 2.1).

QR-decomposition: $\mathbf{A} = \mathbf{Q}\mathbf{R}$, $\mathbf{Q} \in \mathbb{K}^{m,m}$ unitary, $\mathbf{R} \in \mathbb{K}^{m,n}$ (regular) upper triangular matrix.

$$\|\mathbf{Ax} - \mathbf{b}\|_2 = \left\| \mathbf{Q}(\mathbf{Rx} - \mathbf{Q}^H \mathbf{b}) \right\|_2 = \left\| \mathbf{Rx} - \tilde{\mathbf{b}} \right\|_2, \quad \tilde{\mathbf{b}} := \mathbf{Q}^H \mathbf{b}.$$

$$\|\mathbf{Ax} - \mathbf{b}\|_2 \rightarrow \min \Leftrightarrow \left\| \begin{pmatrix} \mathbf{R} \\ \mathbf{0} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} - \begin{pmatrix} \tilde{b}_1 \\ \vdots \\ \tilde{b}_m \end{pmatrix} \right\|_2 \rightarrow \min.$$

$$\mathbf{x} = \begin{pmatrix} \mathbf{R} \\ \mathbf{0} \end{pmatrix}^{-1} \begin{pmatrix} \tilde{b}_1 \\ \vdots \\ \tilde{b}_n \end{pmatrix}, \quad \text{residuum } \mathbf{r} = \mathbf{Q} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \tilde{b}_{n+1} \\ \vdots \\ \tilde{b}_m \end{pmatrix}.$$

Note: residual norm readily available $\|\mathbf{r}\|_2 = \sqrt{\tilde{b}_{n+1}^2 + \cdots + \tilde{b}_m^2}$.

Implementation: successive orthogonal row transformations (by means of Householder reflections (2.1.1) for general matrices, and Givens rotations (2.1.2) for banded matrices, see Sect. 2.1 for details) of augmented matrix $(\mathbf{A}, \mathbf{b}) \in \mathbb{R}^{m,n+1}$, which is transformed into $(\mathbf{R}, \tilde{\mathbf{b}})$

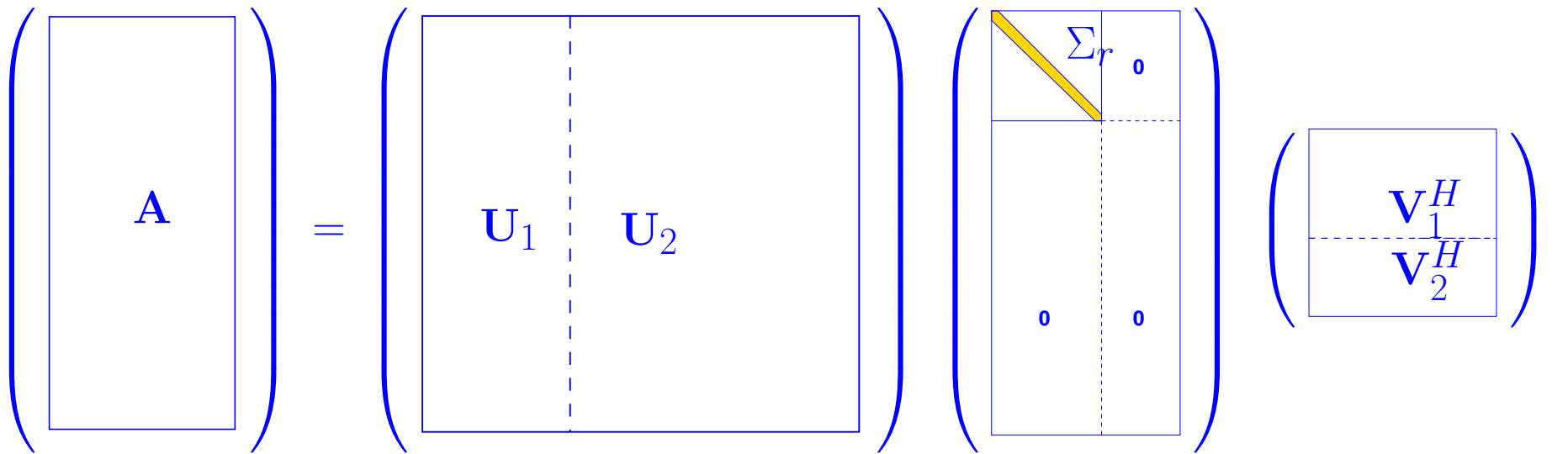
\mathbf{Q} need not be stored !

Alternative:

Solving linear least squares problem by SVD

Most general setting: $\mathbf{A} \in \mathbb{K}^{m,n}$, $\text{rank}(\mathbf{A}) = r \leq \min\{m, n\}$:

SVD: $\mathbf{A} = [\mathbf{U}_1 \quad \mathbf{U}_2] \begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{V}_1^H \\ \mathbf{V}_2^H \end{pmatrix}$



(3.2.1)

 Gradinaru
D-MATH

$\mathbf{U}_1 \in \mathbb{K}^{m,r}$, $\mathbf{U}_2 \in \mathbb{K}^{m,m-r}$, $\Sigma_r = \text{diag}(\sigma_1, \dots, \sigma_r) \in \mathbb{R}^{r,r}$, $\mathbf{V}_1 \in \mathbb{K}^{n,r}$, $\mathbf{V}_2 \in \mathbb{K}^{n,n-r}$,
the columns of $\mathbf{U}_1, \mathbf{U}_2, \mathbf{V}_1, \mathbf{V}_2$ are orthonormal.

$$\|\mathbf{Ax} - \mathbf{b}\|_2 = \left\| [\mathbf{U}_1 \quad \mathbf{U}_2] \begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{V}_1^H \\ \mathbf{V}_2^H \end{pmatrix} \mathbf{x} - \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix} \right\|_2 = \left\| \begin{pmatrix} \Sigma_r \mathbf{V}_1^H \mathbf{x} \\ 0 \end{pmatrix} - \begin{pmatrix} \mathbf{U}_1^H \mathbf{b}_1 \\ \mathbf{U}_2^H \mathbf{b}_2 \end{pmatrix} \right\|_2 \quad (3.2.2)$$

Logical strategy: choose \mathbf{x} such that the first r components of $\begin{pmatrix} \Sigma_r \mathbf{V}_1^H \mathbf{x} \\ 0 \end{pmatrix} - \begin{pmatrix} \mathbf{U}_1^H \mathbf{b}_1 \\ \mathbf{U}_2^H \mathbf{b}_2 \end{pmatrix}$ vanish:

$$\triangleright \text{ underdetermined linear system } \Sigma_r \mathbf{V}_1^H \mathbf{x} = \mathbf{U}_1^H \mathbf{b}_1 . \quad (3.2.3)$$

To fix a unique solution we appeal to the **minimal norm condition** in (3.0.3): solution \mathbf{x} of (3.2.3) is unique up to contributions from $\text{Ker}(\mathbf{V}_1) = \text{Im}(\mathbf{V}_2)$. Since \mathbf{V} is orthogonal, the minimal norm solution is obtained by setting contributions from $\text{Im}(\mathbf{V}_2)$ to zero, which amounts to choosing $\mathbf{x} \in \text{Im}(\mathbf{V}_1)$.

$$\blacktriangleright \text{ solution } \boxed{\mathbf{x} = (\mathbf{V}_1 \Sigma_r^{-1} \mathbf{U}_1^H) \mathbf{b}_1} , \quad \|\mathbf{r}\|_2 = \left\| \mathbf{U}_2^H \mathbf{b}_2 \right\|_2 . \quad (3.2.4)$$

Code 3.2.1: Solving LSQ problem via SVD

Practical implementation:

“numerical rank” test:

$$r = \max\{i: \sigma_i / \sigma_1 > \text{tol}\}$$

```

1 def lsqsvd(A,b,eps=1e-6):
2     U,s,Vh = svd(A)
3     r = 1+where(s/s[0]>eps)[0].max() #
        numerical rank
4     y = dot(Vh[:, :r].T,
        dot(U[:, :r].T,b)/s[:r] )
5     return y

```

Remark 3.2.2 (Pseudoinverse and SVD). \rightarrow Rem. 3.0.4

The solution formula (3.2.4) directly yields a representation of the pseudoinverse \mathbf{A}^+ (\rightarrow Def. 3.0.2) of any matrix \mathbf{A} :

Theorem 3.2.1 (Pseudoinverse and SVD).

If $\mathbf{A} \in \mathbb{K}^{m,n}$ has the SVD decomposition (3.2.1), then $\mathbf{A}^+ = \mathbf{V}_1 \Sigma_r^{-1} \mathbf{U}_1^H$ holds.

`scipy.linalg.pinv2(A)`

`numpy.linalg.pinv(A)`



Remark 3.2.3 (Normal equations vs. orthogonal transformations method).

Superior numerical stability (\rightarrow Def. ??) of orthogonal transformations methods:

► Use orthogonal transformations methods for least squares problems (3.0.3), 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 (3.1.3)/(3.1.4), when $\mathbf{A} \in \mathbb{R}^{m,n}$ sparse (\rightarrow Def. ??) and m, n big. △

3.3 Non-linear Least Squares

Example 3.3.1 (Non-linear data fitting (parametric statistics)).

Given: data points (t_i, y_i) , $i = 1, \dots, m$ with measurements errors.

Known: $y = f(t, \mathbf{x})$ through a function $f : \mathbb{R} \times \mathbb{R}^n \mapsto \mathbb{R}$ depending non-linearly and smoothly on parameters $\mathbf{x} \in \mathbb{R}^n$.

Example: $f(t) = x_1 + x_2 \exp(-x_3 t), \quad n = 3.$

Determine parameters by non-linear **least squares data fitting**:

$$\mathbf{x}^* = \operatorname{argmin}_{\mathbf{x} \in \mathbb{R}^n} \sum_{i=1}^m |f(t_i, \mathbf{x}) - y_i|^2 = \operatorname{argmin}_{\mathbf{x} \in \mathbb{R}^n} \frac{1}{2} \|F(\mathbf{x})\|_2^2, \quad (3.3.1)$$

with
$$F(\mathbf{x}) = \begin{pmatrix} f(t_1, \mathbf{x}) - y_1 \\ \vdots \\ f(t_m, \mathbf{x}) - y_m \end{pmatrix}.$$

Non-linear least squares problem

Given: $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^m$, $m, n \in \mathbb{N}$, $m > n$.

Find: $\mathbf{x}^* \in D$: $\mathbf{x}^* = \operatorname{argmin}_{\mathbf{x} \in D} \Phi(\mathbf{x})$, $\Phi(\mathbf{x}) := \frac{1}{2} \|F(\mathbf{x})\|_2^2$. (3.3.2)

Terminology: $D \hat{=}$ parameter space, $x_1, \dots, x_n \hat{=}$ parameter.

As in the case of linear least squares problems (\rightarrow Rem. 3.0.3): 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 (\rightarrow Chapter 1): existence and uniqueness of \mathbf{x}^* in (3.3.2) has to be established in each concrete case!

We require “independence for each parameter”:

$$\exists \text{ neighbourhood } \mathcal{U}(\mathbf{x}^*) \text{ such that } DF(\mathbf{x}) \text{ has full rank } n \quad \forall \mathbf{x} \in \mathcal{U}(\mathbf{x}^*) . \quad (3.3.3)$$

(It means: the columns of the Jacobi matrix $DF(\mathbf{x})$ are linearly independent.)

If (3.3.3) 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}^*).

3.3.1 (Damped) Newton method

$$\Phi(\mathbf{x}^*) = \min \Rightarrow \mathbf{grad} \Phi(\mathbf{x}) = 0, \quad \mathbf{grad} \Phi(\mathbf{x}) := \left(\frac{\partial \Phi}{\partial x_1}(\mathbf{x}), \dots, \frac{\partial \Phi}{\partial x_n}(\mathbf{x}) \right)^T \in \mathbb{R}^n.$$

Simple idea: use Newton’s method (\rightarrow Sect. 1.4) to determine a zero of $\mathbf{grad} \Phi : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$.

Newton iteration (1.4.1) for non-linear system of equations $\mathbf{grad} \Phi(\mathbf{x}) = 0$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - H\Phi(\mathbf{x}^{(k)})^{-1} \mathbf{grad} \Phi(\mathbf{x}^{(k)}) , \quad (H\Phi(\mathbf{x}) = \text{Hessian matrix}) . \quad (3.3.4)$$

Expressed in terms of $F : \mathbb{R}^n \mapsto \mathbb{R}^m$ from (3.3.2):

chain rule (1.4.2) \blacktriangleright $\mathbf{grad} \Phi(\mathbf{x}) = DF(\mathbf{x})^T F(\mathbf{x}) ,$

product rule (1.4.3) \blacktriangleright $H\Phi(\mathbf{x}) := D(\mathbf{grad} \Phi)(\mathbf{x}) = DF(\mathbf{x})^T DF(\mathbf{x}) + \sum_{j=1}^m F_j(\mathbf{x}) D^2 F_j(\mathbf{x}) ,$

\Updownarrow

$$(H\Phi(\mathbf{x}))_{i,k} = \sum_{j=1}^m \frac{\partial^2 F_j}{\partial x_i \partial x_k}(\mathbf{x}) F_j(\mathbf{x}) + \frac{\partial F_j}{\partial x_k}(\mathbf{x}) \frac{\partial F_j}{\partial x_i}(\mathbf{x}) .$$

\blacktriangleright For Newton iterate $\mathbf{x}^{(k)}$: Newton correction $\mathbf{s} \in \mathbb{R}^n$ from LSE

$$\left(DF(\mathbf{x}^{(k)})^T DF(\mathbf{x}^{(k)}) + \sum_{j=1}^m F_j(\mathbf{x}^{(k)}) D^2 F_j(\mathbf{x}^{(k)}) \right) \mathbf{s} = -DF(\mathbf{x}^{(k)})^T F(\mathbf{x}^{(k)}) . \quad (3.3.5)$$

Remark 3.3.2 (Newton method and minimization of quadratic functional).

Newton's method (3.3.4) for (3.3.2) can be read as *successive minimization* of a local **quadratic approximation** of Φ :

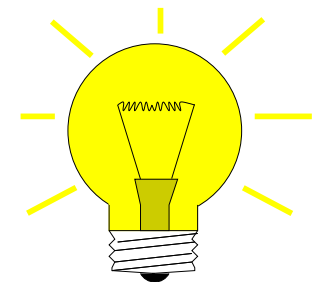
$$\Phi(\mathbf{x}) \approx Q(\mathbf{s}) := \Phi(\mathbf{x}^{(k)}) + \mathbf{grad} \Phi(\mathbf{x}^{(k)})^T \mathbf{s} + \frac{1}{2} \mathbf{s}^T H \Phi(\mathbf{x}^{(k)}) \mathbf{s}, \quad (3.3.6)$$

$$\mathbf{grad} Q(\mathbf{s}) = 0 \Leftrightarrow H \Phi(\mathbf{x}^{(k)}) \mathbf{s} + \mathbf{grad} \Phi(\mathbf{x}^{(k)}) = 0 \Leftrightarrow (3.3.5).$$

➤ Another model function method (→ Sect. 1.3.2) with quadratic model function for Q .



3.3.2 Gauss-Newton method



Idea: **local linearization** of F : $F(x) \approx F(y) + DF(\mathbf{y})(\mathbf{x} - \mathbf{y})$

➤ sequence of *linear* least squares problems

$$\operatorname{argmin}_{\mathbf{x} \in \mathbb{R}^n} \|F(\mathbf{x})\|_2 \text{ approximated by } \underbrace{\operatorname{argmin}_{\mathbf{x} \in \mathbb{R}^n} \|F(\mathbf{x}_0) + DF(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)\|_2}_{(\spadesuit)},$$

where \mathbf{x}_0 is an approximation of the solution \mathbf{x}^* of (3.3.2).

$$(\spadesuit) \Leftrightarrow \operatorname{argmin}_{\mathbf{x} \in \mathbb{R}^n} \|\mathbf{A}\mathbf{x} - \mathbf{b}\| \quad \text{with} \quad \mathbf{A} := DF(\mathbf{x}_0) \in \mathbb{R}^{m,n}, \quad \mathbf{b} := F(\mathbf{x}_0) - DF(\mathbf{x}_0)\mathbf{x}_0 \in \mathbb{R}^m.$$

This is a linear least squares problem of the form (3.0.3).

Note: (3.3.3) \Rightarrow \mathbf{A} has full rank, if \mathbf{x}_0 sufficiently close to \mathbf{x}^* .

Note: Approach different from local quadratic approximation of Φ underlying Newton's method for (3.3.2), see Sect. 3.3.1, Rem. 3.3.2.

Initial guess $\mathbf{x}^{(0)} \in D$

$$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} - \mathbf{s}, \quad \mathbf{s} := \operatorname{argmin}_{\mathbf{s} \in \mathbb{R}^n} \left\| F(\mathbf{x}^{(k)}) - DF(\mathbf{x}^{(k)})\mathbf{s} \right\|_2. \quad (3.3.7)$$

linear least squares problem

we solve a linear least squares problem in each step:

for $\mathbf{A} \in \mathbb{R}^{m,n}$

$\mathbf{x} = \text{numpy.linalg.lstsq}(\mathbf{A}, \mathbf{b})[0]$ \updownarrow

\mathbf{x} minimizer of $\|\mathbf{Ax} - \mathbf{b}\|_2$

with minimal 2-norm

Code 3.3.4: template for Gauss-Newton method

```
1 def gn(x, F, J, tol):
2     s = lstsq(J(x), F(x))[0] #
3     x = x - s
4     while norm(s) > tol * norm(x): #
5         s = lstsq(J(x), F(x))[0] #
6         x = x - s
7     return x
```

Gradinaru
D-MATH

Comments on Code 3.3.2:

➡ Argument \mathbf{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`.

Summary:

Advantage of the Gauss-Newton method : second derivative of F not needed.

Drawback of the Gauss-Newton method : no local quadratic convergence.

Example 3.3.5 (Non-linear data fitting (II)). → Ex. 3.3.1

Non-linear data fitting problem (3.3.1) for $f(t) = x_1 + x_2 \exp(-x_3 t)$.

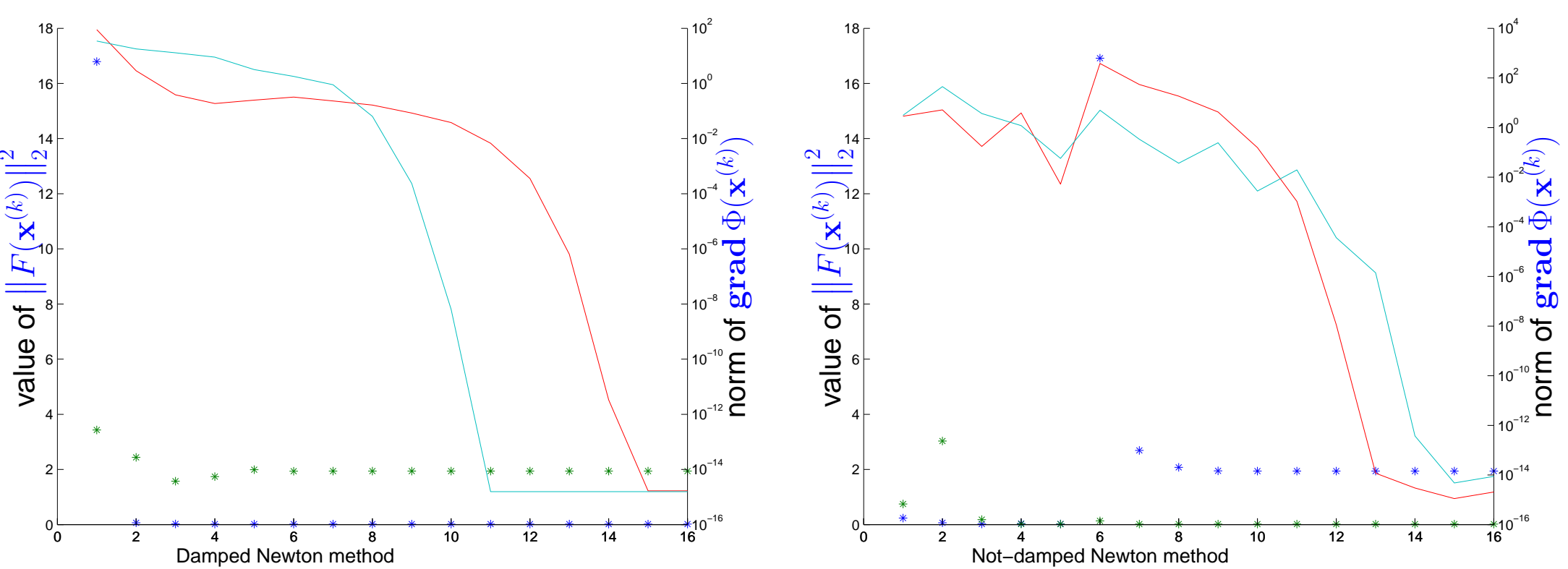
$$F(\mathbf{x}) = \begin{pmatrix} x_1 + x_2 \exp(-x_3 t_1) - y_1 \\ \vdots \\ x_1 + x_2 \exp(-x_3 t_m) - y_m \end{pmatrix} : \mathbb{R}^3 \mapsto \mathbb{R}^m, \quad DF(\mathbf{x}) = \begin{pmatrix} 1 & e^{-x_3 t_1} & -x_2 t_1 e^{-x_3 t_1} \\ \vdots & \vdots & \vdots \\ 1 & e^{-x_3 t_m} & -x_2 t_m e^{-x_3 t_m} \end{pmatrix}$$

Gradinaru
D-MATH

Numerical experiment:

convergence of the Newton method,
damped Newton method (→ Section
1.4.4) and Gauss-Newton method for
different initial values

```
t = r_[1:7:0.3]
y = x[0] + x[1]*exp(-x[2]*t)
y = y+0.1*(rand(len(y))-0.5)
```



Convergence behaviour of the Newton method:

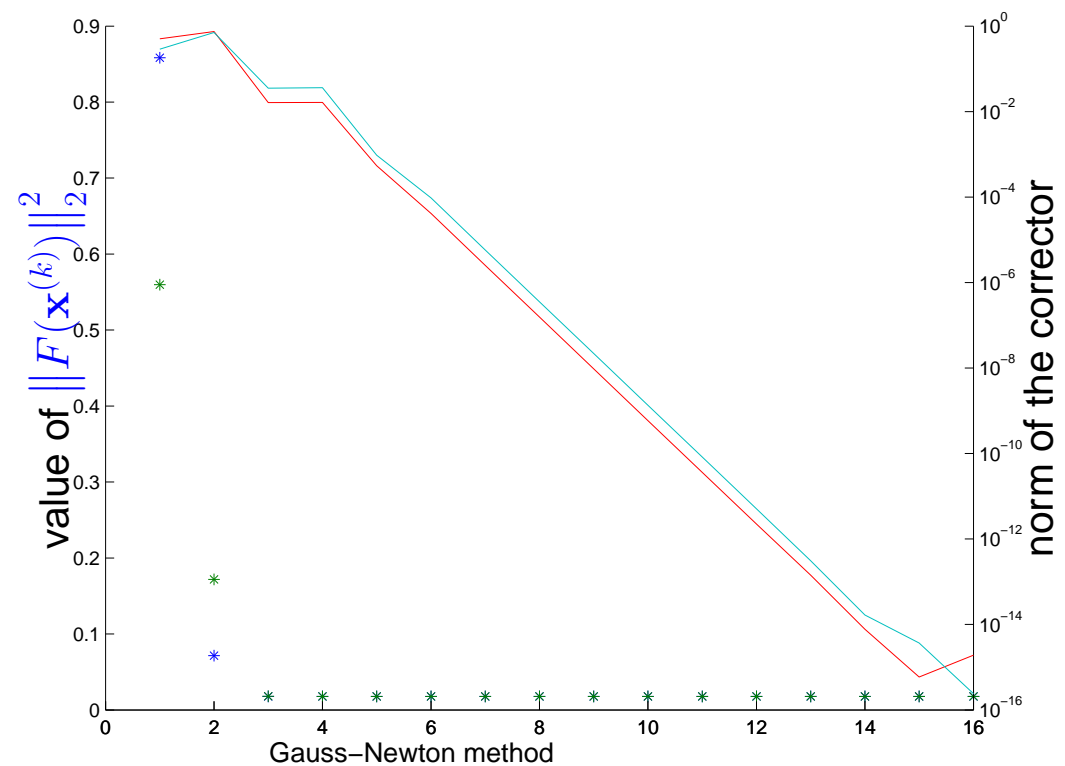
initial value $(1.8, 1.8, 0.1)^T$ (red curve) ➤ Newton method caught in **local minimum**,
 initial value $(1.5, 1.5, 0.1)^T$ (cyan curve) ➤ fast (locally quadratic) convergence.

Gauss-Newton method:

initial value $(1.8, 1.8, 0.1)^T$ (red curve),
initial value $(1.5, 1.5, 0.1)^T$ (cyan curve),

convergence in both cases.

Notice: **linear convergence.**



3.3.3 Trust region method (Levenberg-Marquardt method)

As in the case of Newton's method for non-linear systems of equations, see Sect. 1.4.4: 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 (3.3.7) using a **penalty term**

instead of $\left\| F(\mathbf{x}^{(k)}) + DF(\mathbf{x}^{(k)})\mathbf{s} \right\|^2$ minimize $\left\| F(\mathbf{x}^{(k)}) + DF(\mathbf{x}^{(k)})\mathbf{s} \right\|^2 + \lambda \|\mathbf{s}\|_2^2$.

$\lambda > 0 \hat{=}$ penalty parameter (how to choose it ? \rightarrow heuristic)

$$\lambda = \gamma \left\| F(\mathbf{x}^{(k)}) \right\|_2, \quad \gamma := \begin{cases} 10 & , \text{ if } \left\| F(\mathbf{x}^{(k)}) \right\|_2 \geq 10, \\ 1 & , \text{ if } 1 < \left\| F(\mathbf{x}^{(k)}) \right\|_2 < 10, \\ 0.01 & , \text{ if } \left\| F(\mathbf{x}^{(k)}) \right\|_2 \leq 1. \end{cases}$$

► Modified (regularized) equation for the corrector \mathbf{s} :

$$\left(DF(\mathbf{x}^{(k)})^T DF(\mathbf{x}^{(k)}) + \lambda \mathbf{I} \right) \mathbf{s} = -DF(\mathbf{x}^{(k)}) F(\mathbf{x}^{(k)}) . \quad (3.3.8)$$

`scipy.optimize.leastsq`

3.4 Essential Skills Learned in Chapter 3

You should know:

- several possibilities to solve linear least squares problems
- how to solve non-linear least squares problems

4

Eigenvalues

Example 4.0.1 (Normal mode analysis).

Lecture Physik I, Section 11.2.2: equations of motion for an atom of reduced mass m in the field of another atom, using an harmonic (i.e. quadratic) approximation for the potential:

$$\ddot{x} + \omega_0^2 x = 0, \text{ with } \omega_0 = \sqrt{\frac{k}{m}}, \text{ and } k = D^2U(x^*) .$$

Gradinaru
D-MATH

In the computation of the **IR spectra of a molecule** one is interested into the vibrational frequencies of a molecule which is described by n positional degrees of freedom $\mathbf{x} \in \mathbb{R}^n$ and corresponding velocities $\dot{\mathbf{x}} = \frac{d\mathbf{x}}{dt} \in \mathbb{R}^n$.

Suppose all masses are equal with an effective mass m \blacktriangleright kinetic energy $K(\mathbf{x}, \dot{\mathbf{x}}) = \frac{m}{2} \|\dot{\mathbf{x}}\|_2^2$

Model for total potential energy $U(\mathbf{x})$

1. find a local minimum \mathbf{x}^* of the potential energy:

$$DU(\mathbf{x}^*) = 0, \quad \mathbf{H} = D^2U(\mathbf{x}^*) \text{ sym. pos. semi-def.},$$

hence with Taylor we have near the local minimum: find a local minimum \mathbf{x}^* of the potential energy:

$$U(\mathbf{x}^* + \mathbf{a}) = U(\mathbf{x}^*) + \frac{1}{2}\mathbf{a}^T \mathbf{H} \mathbf{a}.$$

2. Newton'mechanics near the local minimum:

$$m\ddot{\mathbf{x}} = m\ddot{\mathbf{a}} = -D_{\mathbf{a}}U(\mathbf{x}^* + \mathbf{a}).$$

As we are around the minimum:

$$m\ddot{\mathbf{a}} = -D_{\mathbf{a}}(U(\mathbf{x}^*) + \frac{1}{2}\mathbf{a}^T \mathbf{H} \mathbf{a}) = -\mathbf{H} \mathbf{a}.$$

We obtained the equation of motion for small displacements:

$$\ddot{\mathbf{a}} + \frac{1}{m}\mathbf{H}\mathbf{a} = 0, \text{ with} \\ \mathbf{H} = D^2U(\mathbf{x}^*)$$

3. As \mathbf{H} is real and symmetric, its eigenvectors \mathbf{w}^j , with $j = 1, \dots, n$ are orthogonal and hence they form a convenient basis for representing any vector:

$$\mathbf{a} = c_1(t)\mathbf{w}^1 + \dots + c_n(t)\mathbf{w}^n .$$

Inserting into the system of second-order ODEs, we get:

$$m(\ddot{c}_1\mathbf{w}^1 + \dots + \ddot{c}_n\mathbf{w}^n) = -(c_1\mathbf{H}\mathbf{w}^1 + \dots + c_n\mathbf{H}\mathbf{w}^n) = -(c_1\lambda_1\mathbf{w}^1 + \dots + c_n\lambda_n\mathbf{w}^n)$$

where we denoted the associated eigenvalues by λ_j : $\mathbf{H}\mathbf{w}^j = \lambda_j\mathbf{w}^j$.

4. Taking the scalar product with the eigenvector \mathbf{w}^k we obtain an uncoupled set of equations for each $c_k(t)$, $k = 1, \dots, n$:

$$m\ddot{c}_k = -\lambda_k c_k .$$

5. Looking for solutions of the form $c_k = \alpha_k \sin(\omega_k t)$ and substituting it into the differential equation one get the **angular vibrational frequency of the normal mode** $\omega_k = \sqrt{\lambda_k/m}$

6. In case of different masses we end with the system

$$\mathbf{M}\ddot{\mathbf{a}} = -\mathbf{H}\mathbf{a} .$$

with the mass matrix \mathbf{M} which is symmetric, positiv-definite, but not necessarily diagonal. We thus must perform normal mode analysis on the matrix $\mathbf{M}^{-1}\mathbf{H}$:

$$\mathbf{M}^{-1}\mathbf{H}\mathbf{w}^j = \lambda_j \mathbf{w}^j \iff \mathbf{H}\mathbf{w}^j = \lambda_j \mathbf{M}\mathbf{w}^j .$$



Example 4.0.2. Physik I, Section 11.4: forced oscillation

$$\ddot{x} + 2\rho\dot{x} + \omega_0^2 x = \frac{F_0}{m} \cos \Omega t ,$$

has resonance for $\Omega = \omega_0 \sqrt{1 - 2\rho^2/\omega_0^2}$.

The system

$$\mathbf{M}\ddot{\mathbf{x}} + 2\mathbf{B}\dot{\mathbf{x}} + \mathbf{C}\mathbf{x} = \mathbf{f} ,$$

with mass matrix \mathbf{M} , damping matrix \mathbf{B} , stiffness matrix \mathbf{C} and force vector \mathbf{f} has analogously its eigenfrequencies given by the solution of the generalized eigenvalue problem The system

$$\mathbf{C}\mathbf{x} = \omega^2 \mathbf{M}\mathbf{x} .$$



Example 4.0.3 (Analytic solution of homogeneous linear ordinary differential equations). → [52, Remark 5.6.1]

Autonomous homogeneous linear ordinary differential equation (ODE):

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} \quad , \quad \mathbf{A} \in \mathbb{C}^{n,n} . \quad (4.0.1)$$

$$\mathbf{A} = \underbrace{\mathbf{S} \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix} \mathbf{S}^{-1}}_{=: \mathbf{D}} , \quad \mathbf{S} \in \mathbb{C}^{n,n} \text{ regular} \implies \left(\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} \quad \xleftrightarrow{\mathbf{z}=\mathbf{S}^{-1}\mathbf{y}} \quad \dot{\mathbf{z}} = \mathbf{D}\mathbf{z} \right) .$$

➤ solution of initial value problem:

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} , \quad \mathbf{y}(0) = \mathbf{y}_0 \in \mathbb{C}^n \implies \mathbf{y}(t) = \mathbf{S}\mathbf{z}(t) , \quad \dot{\mathbf{z}} = \mathbf{D}\mathbf{z} , \quad \mathbf{z}(0) = \mathbf{S}^{-1}\mathbf{y}_0 .$$

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. ??:

$$\mathbf{A} = \mathbf{S} \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix} \mathbf{S}^{-1} \Leftrightarrow \mathbf{A} ((\mathbf{S})_{:,i}) = \lambda_i ((\mathbf{S})_{:,i}) \quad i = 1, \dots, n . \quad (4.0.2)$$

In order to find the transformation matrix **S** all non-zero solution vectors (= **eigenvectors**) $\mathbf{x} \in \mathbb{C}^n$ of the **linear eigenvalue problem**

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

have to be found.



Example 4.0.4 (Vibrating String). Vibration of a string, fixed at both ends and under uniform tension:

$$\frac{\partial^2 u(x, t)}{\partial t^2} = \frac{T}{m(x)} \frac{\partial^2 u(x, t)}{\partial x^2},$$

with T and $m(x)$ being the tension and the mass per unit length. Separation of variables yields the problem

$$\frac{T}{m(x)} \frac{d^2 y(x)}{dx^2} + \omega^2 y(x) = 0,$$

with ω to be determined from the boundary conditions. In case of the string (one dimensional problem), we can obtain ω analytically, but in several dimensions not anymore.

A possibility is to use finite differences for the last differential equation ($x_i = x_1 + ih$, $y_i \approx y(x_i)$):

$$\frac{T}{m_i} \frac{y_{i-1} - 2y_i + y_{i+1}}{2h} + \omega^2 y_i = 0, i = 1, 2, \dots, N-1 \text{ bcom}$$

which builds down to the eigenvalue problem

$$\mathbf{A}\mathbf{y} = \omega^2 \mathbf{y} ,$$

with tridiagonal matrix \mathbf{A} and $\mathbf{y} = (y_1, \dots, y_{N-1})^T$.



4.1 Theory of eigenvalue problems

Definition 4.1.1 (Eigenvalues and eigenvectors).

- $\lambda \in \mathbb{C}$ **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 $\lambda \in \sigma(\mathbf{A})$:

$$\text{Eig}_{\mathbf{A}}(\lambda) := \text{Ker}(\lambda \mathbf{I} - \mathbf{A})$$
- $\mathbf{x} \in \text{Eig}_{\mathbf{A}}(\lambda) \setminus \{0\} \Rightarrow \mathbf{x}$ is **eigenvector**
- **Geometric multiplicity** (ger.: *Vielfachheit*) of an eigenvalue $\lambda \in \sigma(\mathbf{A})$:

Two simple facts:

$$\lambda \in \sigma(\mathbf{A}) \Rightarrow \dim \text{Eig}_{\mathbf{A}}(\lambda) > 0, \quad (4.1.1)$$

$$\det(\mathbf{A}) = \det(\mathbf{A}^T) \quad \forall \mathbf{A} \in \mathbb{K}^{n,n} \Rightarrow \sigma(\mathbf{A}) = \sigma(\mathbf{A}^T). \quad (4.1.2)$$

notation: $\rho(\mathbf{A}) := \max\{|\lambda| : \lambda \in \sigma(\mathbf{A})\} \hat{=}$ **spectral radius** of $\mathbf{A} \in \mathbb{K}^{n,n}$

Theorem 4.1.2 (Bound for spectral radius).

For any matrix norm $\|\cdot\|$ induced by a vector norm (\rightarrow Def. 1.1.12)

$$\rho(\mathbf{A}) \leq \|\mathbf{A}\|.$$

Lemma 4.1.3 (Gershgorin circle theorem). *For any $\mathbf{A} \in \mathbb{K}^{n,n}$ holds true*

$$\sigma(\mathbf{A}) \subset \bigcup_{j=1}^n \{z \in \mathbb{C} : |z - a_{jj}| \leq \sum_{i \neq j} |a_{ji}|\}.$$

Lemma 4.1.4 (Similarity and spectrum).

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 4.1.5. 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 4.1.6 (Schur normal form).

$$\forall \mathbf{A} \in \mathbb{K}^{n,n}: \quad \exists \mathbf{U} \in \mathbb{C}^{n,n} \text{ unitary: } \mathbf{U}^H \mathbf{A} \mathbf{U} = \mathbf{T} \quad \text{with } \mathbf{T} \in \mathbb{C}^{n,n} \text{ upper triangular} .$$

Corollary 4.1.7 (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: $\mathbf{A}^H = \mathbf{A} \quad \blacktriangleright \quad \sigma(\mathbf{A}) \subset \mathbb{R}$
 - unitary matrices: $\mathbf{A}^H = \mathbf{A}^{-1} \quad \blacktriangleright \quad |\sigma(\mathbf{A})| = 1$
 - skew-Hermitian matrices: $\mathbf{A} = -\mathbf{A}^H \quad \blacktriangleright \quad \sigma(\mathbf{A}) \subset i\mathbb{R}$

➤ Normal matrices can be diagonalized by *unitary* similarity transformations

Symmetric real matrices can be diagonalized by *orthogonal* similarity transformations

- In Thm. 4.1.7:
- $\lambda_1, \dots, \lambda_n$ = eigenvalues of \mathbf{A}
 - Columns of \mathbf{U} = orthonormal basis of eigenvectors of \mathbf{A}

Eigenvalue

- problems:** (EVPs)
- ❶ Given $\mathbf{A} \in \mathbb{K}^{n,n}$ find **all eigenvalues** (= **spectrum** of \mathbf{A}).
 - ❷ Given $\mathbf{A} \in \mathbb{K}^{n,n}$ find $\sigma(\mathbf{A})$ plus **all eigenvectors**.
 - ❸ 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 $\mathbf{x} \neq 0$, $\lambda \in \mathbb{C}$

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x} \Leftrightarrow \mathbf{B}^{-1}\mathbf{A}\mathbf{x} = \lambda\mathbf{x} . \quad (4.1.3)$$

$\mathbf{x} \hat{=}$ generalized eigenvector, $\lambda \hat{=}$ generalized eigenvalue

Obviously every generalized eigenvalue problem is equivalent to a standard eigenvalue problem

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x} \Leftrightarrow \mathbf{B}^{-1}\mathbf{A} = \lambda\mathbf{x} .$$

However, usually it is not advisable to use this equivalence for numerical purposes!

Remark 4.1.1 (Generalized eigenvalue problems and Cholesky factorization).

If $\mathbf{B} = \mathbf{B}^H$ s.p.d. (\rightarrow Def. ??) with Cholesky factorization $\mathbf{B} = \mathbf{R}^H\mathbf{R}$

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x} \Leftrightarrow \tilde{\mathbf{A}}\mathbf{y} = \lambda\mathbf{y} \quad \text{where } \tilde{\mathbf{A}} := \mathbf{R}^{-H}\mathbf{A}\mathbf{R}^{-1}, \mathbf{y} := \mathbf{R}\mathbf{x} .$$

→ This transformation can be used for efficient computations.

4.2 “Direct” Eigensolvers

Purpose: solution of eigenvalue problems ❶, ❷ for **dense** matrices “up to machine precision”

python-functions: `numpy.linalg.eig`, `scipy.linalg.eig`

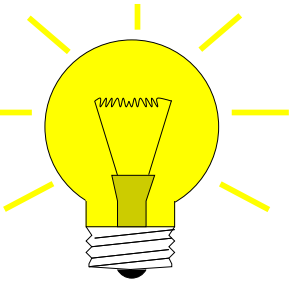
Gradinaru
D-MATH

<code>w = eigvals(A)</code>	:	computes spectrum $\sigma(\mathbf{A}) = \{w_1, \dots, w_n\}$ of $\mathbf{A} \in \mathbb{C}^{n,n}$
<code>w, V = eig(A)</code>	:	computes spectrum w and corresponding normed eigenvectors
<code>eigvalsh(A)</code> and <code>eigh(A)</code>	:	specialized algorithms for Hermitial matrices
	\Rightarrow	wrappers to <code>lapack</code> -functions

Remark 4.2.1 (QR-Algorithm). → [20, Sect. 7.5]

Note: All “direct” eigensolvers are iterative methods

Idea: Iteration based on successive **unitary** similarity transformations



$$\mathbf{A} = \mathbf{A}^{(0)} \rightarrow$$

$$\mathbf{A}^{(1)} \rightarrow$$

$$\dots \rightarrow$$

diagonal matrix

, if $\mathbf{A} = \mathbf{A}^H$,

upper triangular matrix, else.

(\rightarrow Thm. 4.1.6)

(superior stability of unitary transformations, see Rem. ??)


```

1  '''
2  QR-algorithm_with_shift
3  '''
4  from numpy import mat, argmin, eye, tril
5  from numpy.linalg import norm, eig, qr, eigvals
6
7  def eigqr(A, tol):
8      n = A.shape[0]
9      while (norm(tril(A, -1), ord=2) > tol * norm(A,
10              ord=2)):
11          # shift by ew of lower right 2×2 block closest to  $(\mathbf{A})_{n,n}$ 
12          sc, dummy = eig(A[n-2:n, n-2:n])
13          k = argmin(abs(sc - A[n-1, n-1]))
14          shift = sc[k]
15          Q, R = qr(A - shift * eye(n))
16          A = mat(Q).H * mat(A) * mat(Q);
17          d = A.diagonal()
18          return d
19
20 if __name__ == "__main__":
21     A = mat('1,2,3;_4,5,6;_7,8,9')
22     print 'numpy.linalg.eigvals:', eigvals(A)
23     print 'with_our_eigqr:', eigqr(A, 10**-6)

```

► **QR-algorithm** (with
shift)

• in general: quadratic
convergence

• cubic convergence
for normal matrices
(→ [20, Sect. 7.5, 8.2])

Computational cost: $O(n^3)$ operations per step of the QR-algorithm

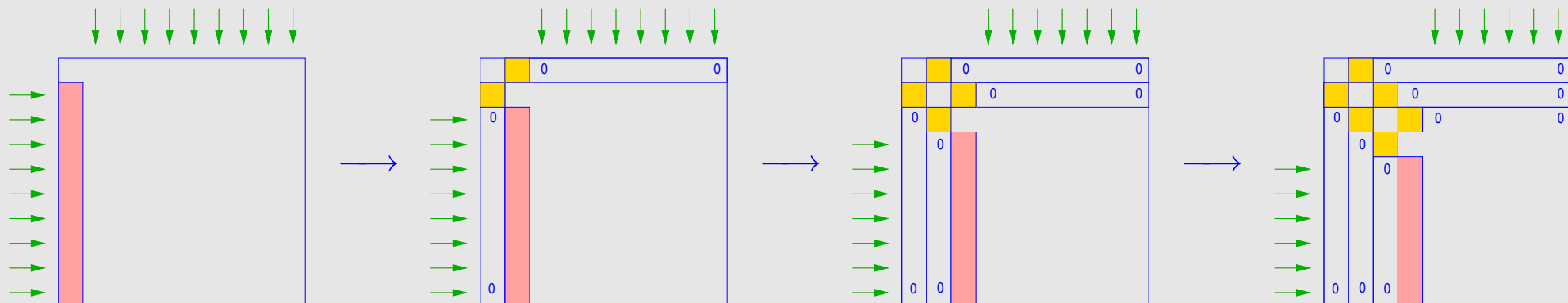
Library implementations of the QR-algorithm provide *numerically stable* eigensolvers



Remark 4.2.3 (Unitary similarity transformation to tridiagonal form).

Successive Householder similarity transformations of $\mathbf{A} = \mathbf{A}^H$:

($\rightarrow \hat{=}$ affected rows/columns, $\hat{=}$ 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 $\mathbf{Ax} = \lambda \mathbf{Bx}$, $\mathbf{A}, \mathbf{B} \in \mathbb{C}^{n,n}$

`scipy.linalg.eig(a, b=None, left=False, right=True)`: also for generalized eigenvalue problems

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 4.2.4 (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 $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$	$\sim 30n^2 + O(n)$		

Sparse eigensolvers: ARPACK

scipy 0.7.1: `from scipy.sparse.linalg.eigen.arpack import arpack`
`arpack.eigen`, and `arpack.eigen_symmetric`
 scipy 0.10: `scipy.sparse.linalg.eigs`

Note: `eig` not available in Matlab for sparse matrix arguments

Exception: `d=eig(A)` for sparse *Hermitian* matrices



Example 4.2.5 (Runtimes of `eig`).

Code 4.2.6: measuring runtimes of `eig`

```
import numpy as np
from scipy.sparse import lil_diags, lil_matrix
import scipy.sparse.linalg as sparsela
```

```
import time
from scipy.sparse.linalg.eigen.arpack import arpack
```

```
N = 500
A = np.random.rand(N,N); A = np.mat(A)
B = A.H*A
z = np.ones(N)

t0 = time.time()
C = lil_diags([3*z,z,z],[0,1,-1],(N,N))
t1 = time.time()
print 'C_constructed_in ', t1-t0, 'seconds'
D = C.tocsr()
```

```
nexp = 4
ns = np.arange(5,N+1,5)
times = []
for n in ns:
    print 'n=', n
    An = A[:n,:n]; Bn = B[:n,:n]; Dn = D[:n,:n]

    ts = np.zeros(nexp)
    for k in xrange(nexp):
```

```
ti = time.time()
w = np.linalg.eigvals(An)
tf = time.time()
ts[k] = tf-ti
t1 = ts.sum()/nexp
print 't1_=', t1
```

```
ts = np.zeros(nexp)
for k in xrange(nexp):
    ti = time.time()
    w, V = np.linalg.eig(An)
    tf = time.time()
    ts[k] = tf-ti
t2 = ts.sum()/nexp
print 't2_=', t2
```

```
ts = np.zeros(nexp)
for k in xrange(nexp):
    ti = time.time()
    w = np.linalg.eigvalsh(Bn)
    tf = time.time()
    ts[k] = tf-ti
t3 = ts.sum()/nexp
```

```
print 't3_=', t3
```

```
ts = np.zeros(nexp)
for k in xrange(nexp):
    ti = time.time()
    w, V = np.linalg.eigh(Bn)
    tf = time.time()
    ts[k] = tf-ti
t4 = ts.sum()/nexp
print 't4_=', t4
```

```
ts = np.zeros(nexp)
for k in xrange(nexp):
    ti = time.time()
    w, V = arpack.eigen_symmetric(Dn, k=n-1)
    tf = time.time()
    ts[k] = tf-ti
t6 = ts.sum()/nexp
print 't6_=', t6
```

```
times += [np.array([t1, t2, t3, t4, t6])]
```

```
#
times = np.array(times)
```

```
from matplotlib import pyplot as plt

plt.loglog(ns, times[:,0], 'r+')
plt.loglog(ns, times[:,1], 'm*')
plt.loglog(ns, times[:,2], 'cp')
plt.loglog(ns, times[:,3], 'b^')
plt.loglog(ns, times[:,4])
plt.xlabel('matrix_size_n')
plt.ylabel('time_[s]')
plt.title('eig_runtimes')
plt.legend(('w_=_eig(A)', 'w,_V_=_eig(A)', 'w_=_eig(B)', 'w,_V_=_eig(B)',
          'w_=_arpack.eigen_symmetric(C)'), loc='upper_left')
plt.savefig('eigtimingall.eps')
plt.show()

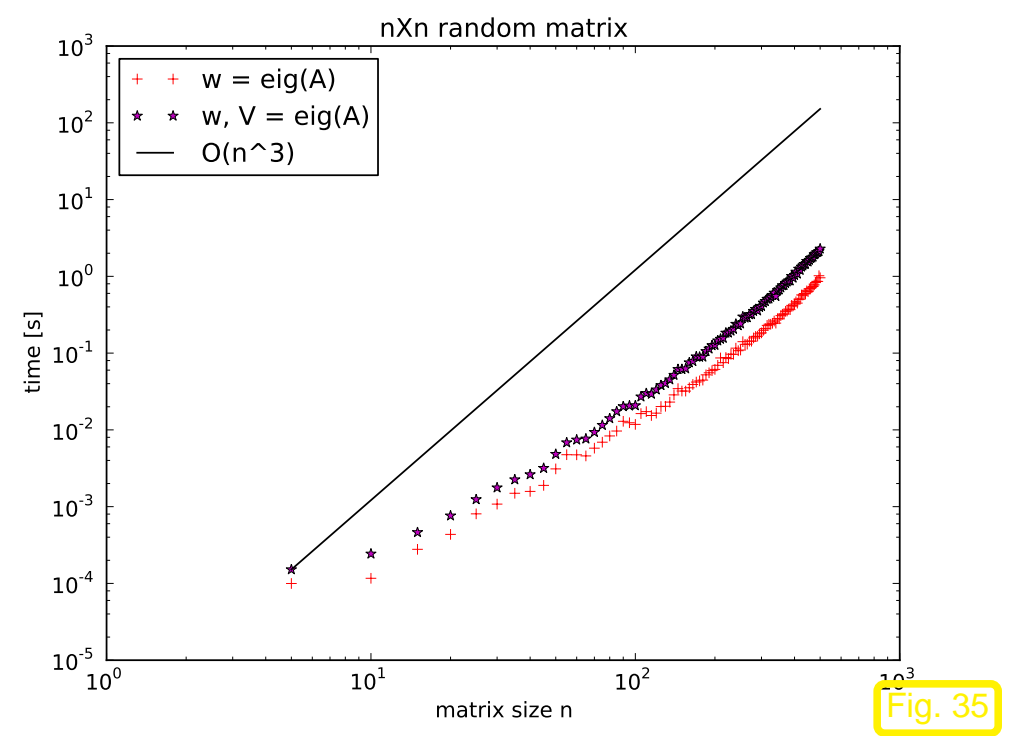
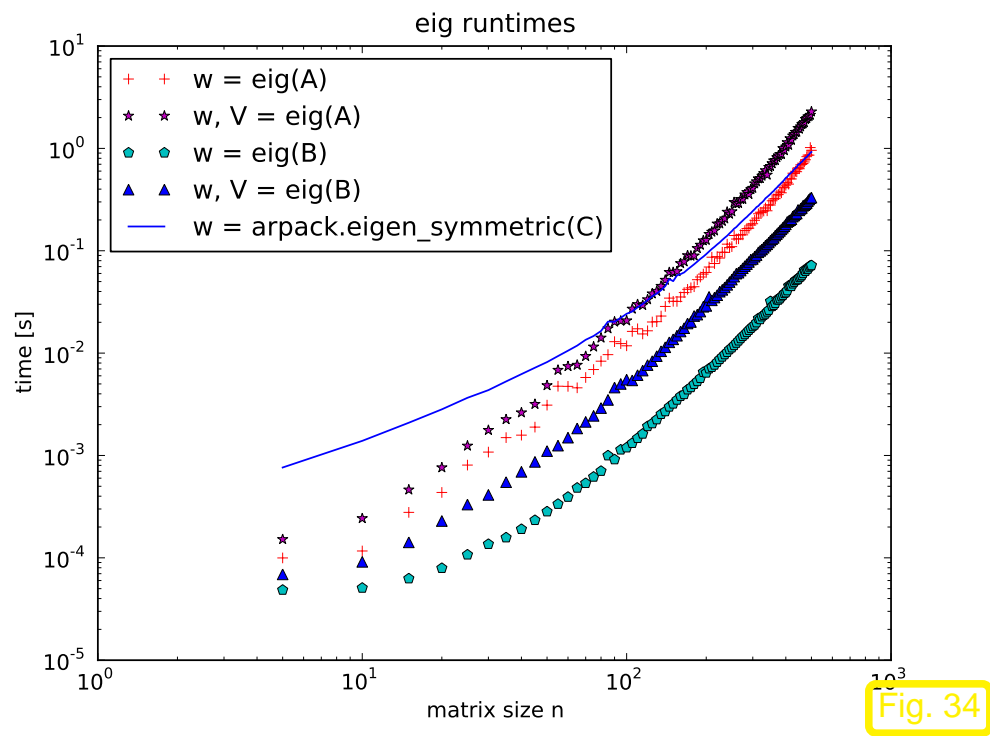
plt.clf()
plt.loglog(ns, times[:,0], 'r+')
plt.loglog(ns, times[:,1], 'm*')
plt.loglog(ns, ns**3./(ns[0]**3.)*times[0,1], 'k-')
plt.xlabel('matrix_size_n')
plt.ylabel('time_[s]')
plt.title('nXn_random_matrix')
plt.legend(('w_=_eig(A)', 'w,_V_=_eig(A)', 'O(n^3)'), loc='upper_left')
plt.savefig('eigtimingA.eps')
```



```
plt.show()

plt.clf()
plt.loglog(ns, times[:,2], 'cp')
plt.loglog(ns, times[:,3], 'b^')
plt.loglog(ns, ns**3./(ns[0]**3.)*times[0,2], 'k-')
plt.xlabel('matrix_size_n')
plt.ylabel('time_[s]')
plt.title('nXn_random_Hermitian_matrix')
plt.legend(('w_=_eig(B)', 'w,_V_=_eig(B)', 'O(n^3)'), loc='upper_left')
plt.savefig('eigtimingB.eps')
plt.show()
```

```
plt.clf()
plt.loglog(ns, times[:,4], 'cp')
plt.loglog(ns, ns**2./(ns[0]**2.)*times[0,4], 'k-')
plt.xlabel('matrix_size_n')
plt.ylabel('time_[s]')
plt.title('nXn_random_Hermitian_matrix')
plt.legend(('w_=_arpack.eigen_symmetric(C)', 'O(n^2)'), loc='upper_
left')
plt.savefig('eigtimingC.eps')
plt.show()
```



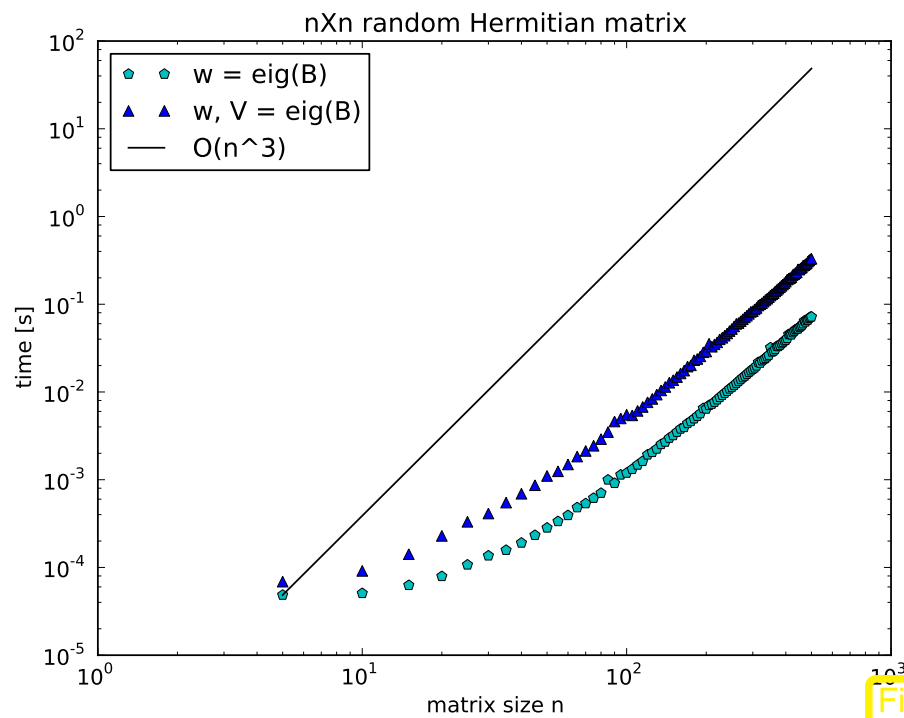


Fig. 36

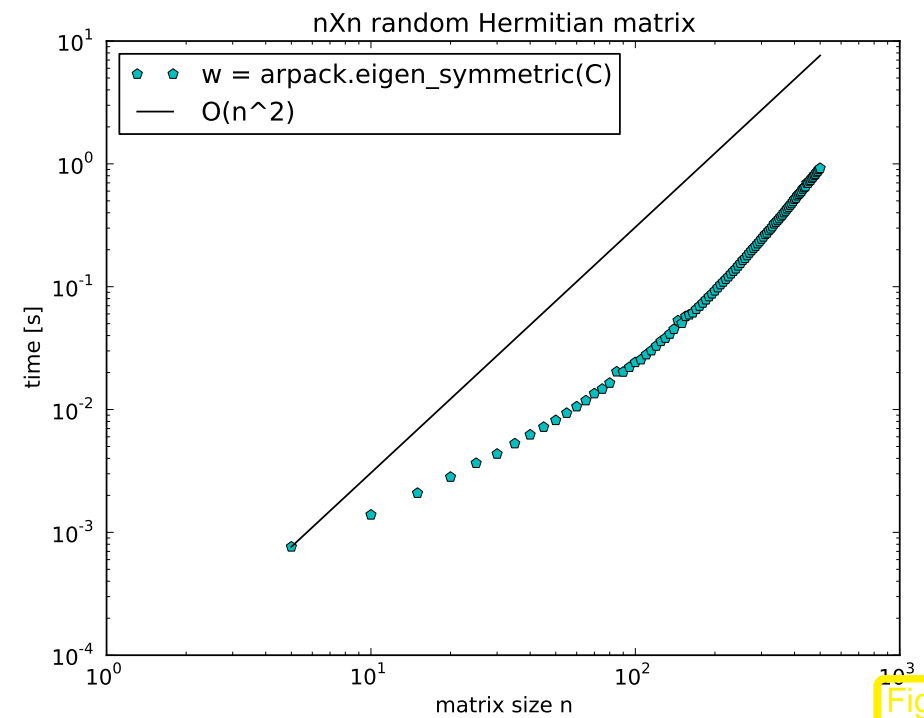


Fig. 37

➡ For the sake of efficiency: think which information you really need when computing eigenvalues/eigenvectors of dense matrices

Gradinaru
D-MATH

Potentially more efficient methods for *sparse matrices* will be introduced below in Sects. 4.3, 4.4.

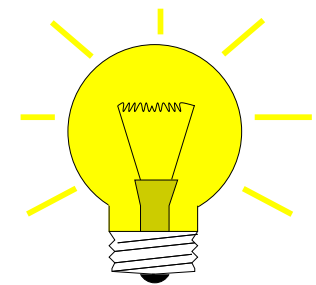


4.3.1 Direct power method

Task: given $\mathbf{A} \in \mathbb{K}^{n,n}$, find **largest** (in modulus) eigenvalue of \mathbf{A} and (an) associated eigenvector.

Idea for $\mathbf{A} \in \mathbb{K}^{n,n}$ diagonalizable: $\mathbf{S}^{-1}\mathbf{A}\mathbf{S} = \text{diag}(\lambda_1, \dots, \lambda_n)$

$$\mathbf{z} = \sum_{j=1}^n \zeta_j (\mathbf{S})_{:,j} \Rightarrow \mathbf{A}^k \mathbf{z} = \sum_{j=1}^n \zeta_j \lambda_j^k (\mathbf{S})_{:,j}.$$



If $|\lambda_1| \leq |\lambda_2| \leq \dots \leq |\lambda_{n-1}| < |\lambda_n|$, $\|(\mathbf{S})_{:,j}\|_2 = 1$, $j = 1, \dots, n$, $\zeta_n \neq 0$

$$\blacktriangleright \frac{\mathbf{A}^k \mathbf{z}}{\|\mathbf{A}^k \mathbf{z}\|} \rightarrow \pm (\mathbf{S})_{:,n} = \text{eigenvector for } \lambda_n \text{ for } k \rightarrow \infty. \quad (4.3.1)$$

► Suggests **direct power method** (ger.: Potenzmethode): iterative method (\rightarrow Sect. 1.1)

initial guess: $\mathbf{z}^{(0)}$ “arbitrary” ,

$$\text{next iterate: } \mathbf{w} := \mathbf{A}\mathbf{z}^{(k-1)}, \quad \mathbf{z}^{(k)} := \frac{\mathbf{w}}{\|\mathbf{w}\|_2}, \quad k = 1, 2, \dots \quad (4.3.2)$$

Computational effort: $1 \times \text{matrix} \times \text{vector}$ per step \triangleright inexpensive for sparse matrices

$\mathbf{z}^{(k)} \rightarrow$ eigenvector, but how do we get the associated eigenvalue λ_n ?

❶ upon convergence from (4.3.1) $\triangleright \mathbf{A}\mathbf{z}^{(k)} \approx \lambda_n \mathbf{z}^{(k)} \triangleright |\lambda_n| \approx \frac{\|\mathbf{A}\mathbf{z}^{(k)}\|}{\|\mathbf{z}^{(k)}\|}$

❷ for $\mathbf{A} = \mathbf{A}^T \in \mathbb{R}^{n,n}$: $\lambda_n \approx \operatorname{argmin}_{\theta \in \mathbb{R}} \|\mathbf{A}\mathbf{z}^{(k)} - \theta \mathbf{z}^{(k)}\|_2^2 \triangleright \lambda_n \approx \frac{(\mathbf{z}^{(k)})^T \mathbf{A}\mathbf{z}^{(k)}}{\|\mathbf{z}^{(k)}\|^2}.$

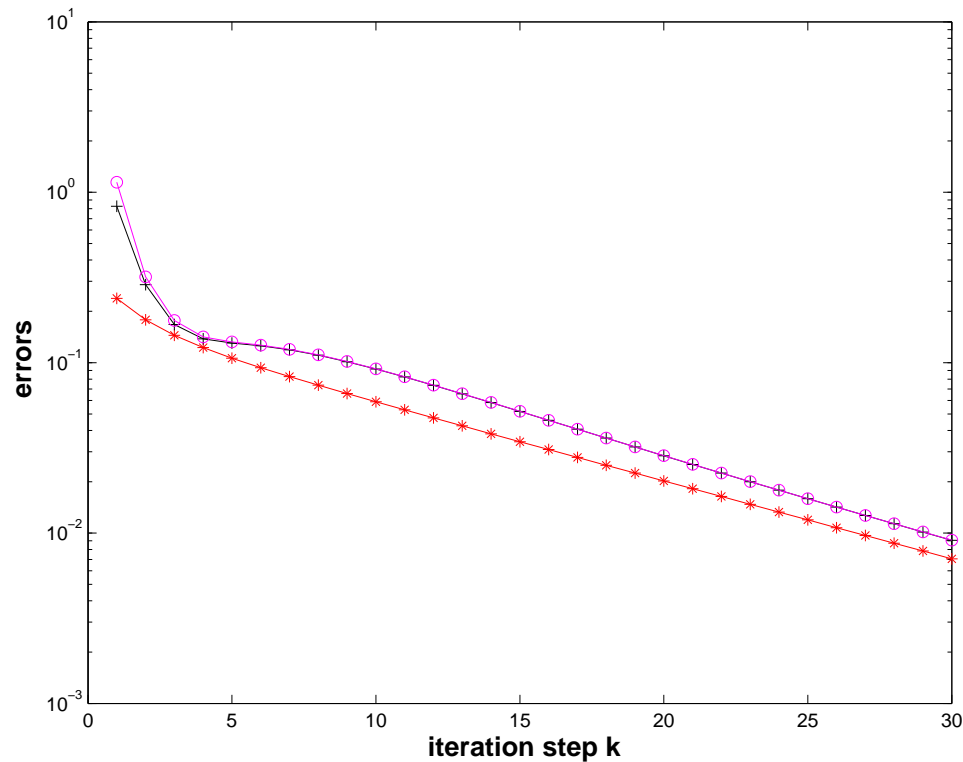
Definition 4.3.1. For $\mathbf{A} \in \mathbb{K}^{n,n}$, $\mathbf{u} \in \mathbb{K}^n$ the **Rayleigh quotient** is defined by

$$\rho_{\mathbf{A}}(\mathbf{u}) := \frac{\mathbf{u}^H \mathbf{A} \mathbf{u}}{\mathbf{u}^H \mathbf{u}}.$$

An immediate consequence of the definitions:

$$\lambda \in \sigma(\mathbf{A}) \quad , \quad \mathbf{z} \in \text{Eig}_\lambda(\mathbf{A}) \quad \Rightarrow \quad \rho_{\mathbf{A}}(\mathbf{z}) = \lambda . \quad (4.3.3)$$

Example 4.3.1 (Direct power method).



```
n = len(d) # size of the matrix
S = triu(diag(r_[n:0:-1])+ones((n,
A = dot(S, dot(diag(d), inv(S)) )
```

○ : error $|\lambda_n - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})|$

* : error norm $\|\mathbf{z}^{(k)} - \mathbf{s}_{\cdot,n}\|$

△

+ : $\left| \lambda_n - \frac{\|\mathbf{A}\mathbf{z}^{(k-1)}\|_2}{\|\mathbf{z}^{(k-1)}\|_2} \right|$

$\mathbf{z}^{(0)}$ = random vector

- ① $d = 1.+r_{-}[0:n] \succ |\lambda_{n-1}| : |\lambda_n| = 0.9$
- ② $d = \text{ones}(n); d[-1] = 2. \succ |\lambda_{n-1}| : |\lambda_n| = 0.5$
- ③ $d = 1.-0.5**r_{-}[1:5.1:0.5] \succ |\lambda_{n-1}| : |\lambda_n| = 0.9866$

$$\rho_{EV}^{(k)} := \frac{\left\| \mathbf{z}^{(k)} - \mathbf{s}_{\cdot,n} \right\|}{\left\| \mathbf{z}^{(k-1)} - \mathbf{s}_{\cdot,n} \right\|},$$
$$\rho_{EW}^{(k)} := \frac{|\rho_{\mathbf{A}}(\mathbf{z}^{(k)}) - \lambda_n|}{|\rho_{\mathbf{A}}(\mathbf{z}^{(k-1)}) - \lambda_n|}.$$

	①		②		③	
k	$\rho_{EV}^{(k)}$	$\rho_{EW}^{(k)}$	$\rho_{EV}^{(k)}$	$\rho_{EW}^{(k)}$	$\rho_{EV}^{(k)}$	$\rho_{EW}^{(k)}$
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 (→ Def. 1.1.4)



Theorem 4.3.2 (Convergence of direct power method).

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 λ_n normalized according to $\|\mathbf{y}\|_2 = \|\mathbf{v}\|_2 = 1$. Then there is convergence

$$\left\| \mathbf{A} \mathbf{z}^{(k)} \right\|_2 \rightarrow \lambda_n, \quad \mathbf{z}^{(k)} \rightarrow \pm \mathbf{v} \quad \text{linearly with rate } \frac{|\lambda_{n-1}|}{|\lambda_n|},$$

where $\mathbf{z}^{(k)}$ are the iterates of the direct power iteration and $\mathbf{y}^H \mathbf{z}^{(0)} \neq 0$ is assumed.

Remark 4.3.2 (Initial guess for power iteration).

roundoff errors $\blacktriangleright \mathbf{y}^H \mathbf{z}^{(0)} \neq 0$ always satisfied in practical computations

Usual (not the best!) choice for $\mathbf{x}^{(0)} =$ random vector



Remark 4.3.3 (Termination criterion for direct power iteration). (\rightarrow Sect. 1.1.2)

Adaptation of a posteriori termination criterion (1.2.7)

“relative change” $\leq \text{tol}$:

$$\left\{ \begin{array}{l} \left\| \mathbf{z}^{(k)} - \mathbf{z}^{(k-1)} \right\| \leq (1/L - 1) \text{tol} , \\ \left| \frac{\left\| \mathbf{A} \mathbf{z}^{(k)} \right\|}{\left\| \mathbf{z}^{(k)} \right\|} - \frac{\left\| \mathbf{A} \mathbf{z}^{(k-1)} \right\|}{\left\| \mathbf{z}^{(k-1)} \right\|} \right| \leq (1/L - 1) \text{tol} \quad \text{see (1.1.17)} . \end{array} \right.$$

Estimated rate of convergence

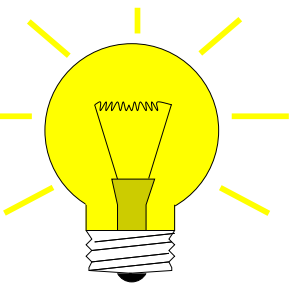


4.3.2 Inverse Iteration

Task: given $\mathbf{A} \in \mathbb{K}^{n,n}$, find **smallest** (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:

$$\text{Smallest (in modulus) EV of } \mathbf{A} = \left(\text{Largest (in modulus) EV of } \mathbf{A}^{-1} \right)^{-1}$$



Code 4.3.4: inverse iteration for computing $\lambda_{\min}(A)$ and associated eigenvector

```

1 import numpy as np
2 import scipy.linalg as splalg
3
4 def invit(A, tol):
5     LUP = splalg.lu_factor(A, overwrite_a=True)
6     n = A.shape[0]
7     x = np.random.rand(n)
8     x /= np.linalg.norm(x)
9     splalg.lu_solve(LUP, x, overwrite_b=True)
10    lold = 0
11    lmin = 1./np.linalg.norm(x)
12    x *= lmin
13    while(abs(lmin-lold) > tol*lmin):
14        lold = lmin
15        splalg.lu_solve(LUP, x, overwrite_b=True)
16        lmin = 1./np.linalg.norm(x)
17        x *= lmin
18    return lmin

```

Note: **reuse** of LU-factorization

Remark 4.3.5 (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:

$$\mathbf{z}^{(0)} \text{ arbitrary, } \mathbf{w} = (\mathbf{A} - \alpha \mathbf{I})^{-1} \mathbf{z}^{(k-1)}, \quad \mathbf{z}^{(k)} := \frac{\mathbf{w}}{\|\mathbf{w}\|_2}, \quad k = 1, 2, \dots, \quad (4.3.4)$$

where: $(\mathbf{A} - \alpha \mathbf{I})^{-1} \mathbf{z}^{(k-1)} \hat{=}$ 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 4.3.3).

What if “by accident” $\alpha \in \sigma(\mathbf{A})$ ($\Leftrightarrow \mathbf{A} - \alpha \mathbf{I}$ singular) ?

Stability of Gaussian elimination/LU-factorization will ensure that “**w** from (4.3.4) 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: If, in the course of Gaussian elimination/LU-factorization a zero pivot element is really encountered, then we just *replace it with* `eps`, in order to avoid `inf` values!

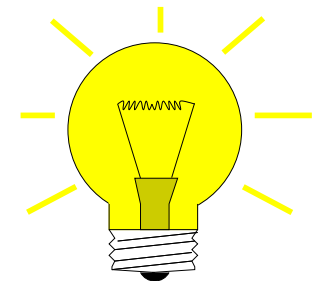
Thm. 4.3.2 ➤ Convergence of shifted inverse iteration for $\mathbf{A}^H = \mathbf{A}$:

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 $\alpha \approx \lambda_j$!

△ Gradinaru
D-MATH



Idea:

A posteriori adaptation of shift

Use $\alpha := \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})$ in k -th step of inverse iteration.



Algorithm 4.3.6 (Rayleigh quotient iteration).

(for normal $\mathbf{A} \in \mathbb{K}^{n,n}$)

Code 4.3.7: Rayleigh quotient iteration for computing $\lambda_{\min}(\mathbf{A})$ and associated eigenvector

```

1 import numpy as np
2 def rqi(A, maxit):
3     # For calculating the errors, the eigenvalues are calculated here with eig
4     w, V = np.linalg.eig(A)
5     t = np.where(w == abs(w).min())
6     k = t[0];
7     if len(k) > 1:
8         print 'Error:_no_single_smallest_EV'
9         raise ValueError
10    ev = V[:,k[0]]; ev /= np.linalg.norm(ev)
11    ew = w[k[0]]
12    #
13    n = A.shape[0]
14    alpha = 0.
15    z = np.random.rand(n); z /= np.linalg.norm(z)
16    for k in xrange(maxit):
17        z = np.linalg.solve(A-alpha*np.eye(n), z)

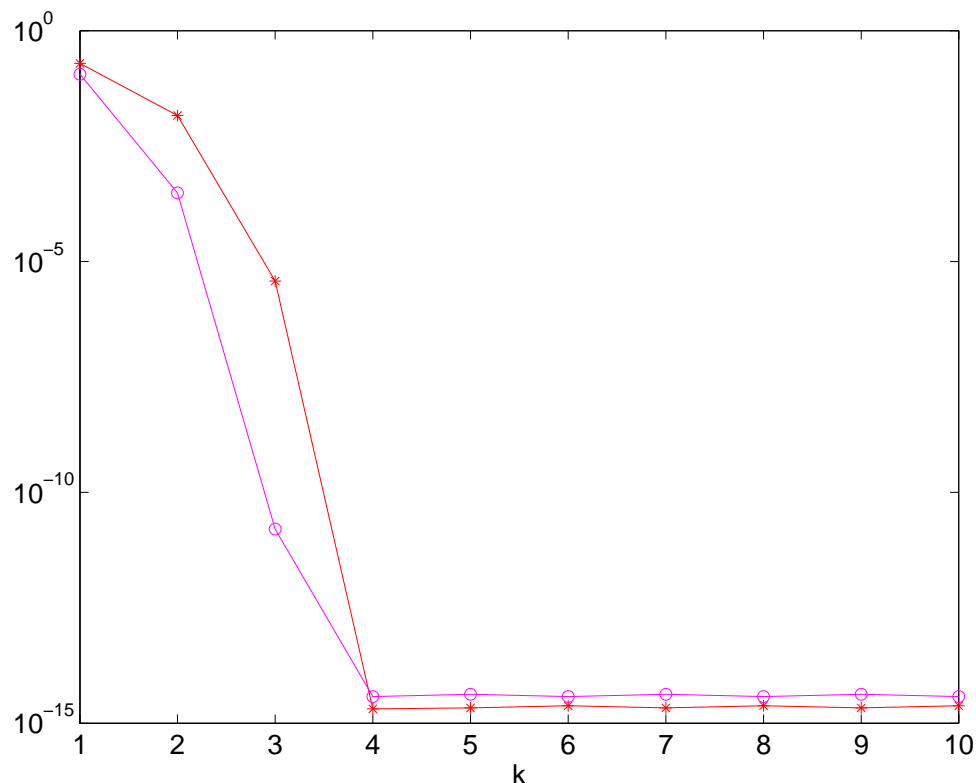
```

```
18     z /= np.linalg.norm(z)
19     alpha = np.dot(np.dot(A,z),z)
20     ea = abs(alpha-ew)
21     eb = np.min(np.linalg.norm(z-ev),np.linalg.norm(z+ev))
22     print 'ea, _eb_=', ea, eb
23
24 def runit(d, tol, func):
25     n = len(d) # size of the matrix
26     Z = np.diag(np.sqrt(np.r_[n:0:-1])) + np.ones((n,n))
27     Q,R = np.linalg.qr(Z)
28     A = np.dot(Q, np.dot(np.diag(d), np.linalg.inv(Q)) )
29     res = func(A,tol)
30     print res
31
32 # -----
33 if __name__=='__main__':
34     n = 10 # size of the matrix
35     d = 1.+np.r_[0:n] # eigenvalues
36     print 'd_=', d, 'd[-2]/d[-1]_=', d[-2]/d[-1]
37     runit(d, 10, rqi)
```

- Drawback compared with Code 4.3.3: 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. 4.3.5.

Example 4.3.8 (Rayleigh quotient iteration).

Monitored: iterates of Rayleigh quotient iteration (4.3.6) for s.p.d. $\mathbf{A} \in \mathbb{R}^{n,n}$



$$\begin{aligned}
 \circ & : |\lambda_{\min} - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})| \\
 * & : \left\| \mathbf{z}^{(k)} - \mathbf{x}_j \right\|, \lambda_{\min} = \lambda_j, \mathbf{x}_j \in \text{Eig}_{\mathbf{A}}(\lambda_j), \\
 & : \left\| \mathbf{x}_j \right\|_2 = 1
 \end{aligned}$$

k	$ \lambda_{\min} - \rho_{\mathbf{A}}(\mathbf{z}^{(k)}) $	$\ \mathbf{z}^{(k)} - \mathbf{x}_j\ $
1	0.09381702342056	0.20748822490698
2	0.00029035607981	0.01530829569530
3	0.00000000001783	0.00000411928759
4	0.00000000000000	0.00000000000000
5	0.00000000000000	0.00000000000000

Theorem 4.3.3. If $\mathbf{A} = \mathbf{A}^H$, then $\rho_{\mathbf{A}}(\mathbf{z}^{(k)})$ converges locally of order 3 (\rightarrow Def. 1.1.13) to an eigenvalue, when $\mathbf{z}^{(k)}$ are generated by the Rayleigh quotient iteration (4.3.6).



The inverse iteration with shift depends very much on the initial value: shifting near one eigenvalue forces the method to produce that value; in other words, the iteration is cached into the subspace corresponding to the nearest eigenvalue.

4.3.3 Preconditioned inverse iteration (PINVIT)

Task: given $\mathbf{A} \in \mathbb{K}^{n,n}$, find **smallest** (in modulus) eigenvalue of regular $\mathbf{A} \in \mathbb{K}^{n,n}$ and (an) associated eigenvector.



Options: inverse iteration (\rightarrow Code 4.3.3) and Rayleigh quotient iteration (4.3.6).

What if direct solution of $\mathbf{Ax} = \mathbf{b}$ not feasible ?

This can happen, in case

- for large sparse \mathbf{A} the amount of fill-in exhausts memory, despite sparse elimination techniques
- \mathbf{A} is available only through a routine `evalA(x)` providing $\mathbf{A} \times \text{vector}$.

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 $\mathbf{Aw} = \mathbf{z}^{(k-1)}$ may be considered. 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* instead.

Gradinaru
D-MATH

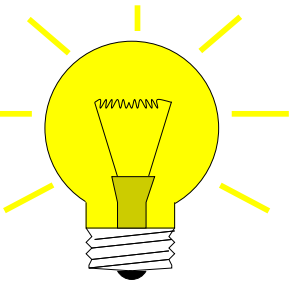
Idea: (for inverse iteration without shift, $\mathbf{A} = \mathbf{A}^H$ s.p.d.)

Instead of solving $\mathbf{Aw} = \mathbf{z}^{(k-1)}$ compute $\mathbf{w} = \mathbf{B}^{-1}\mathbf{z}^{(k-1)}$ with
“inexpensive” s.p.d. **approximate inverse** $\mathbf{B}^{-1} \approx \mathbf{A}^{-1}$

➤ $\mathbf{B} \hat{=}$ **Preconditioner** for \mathbf{A}

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 $\mathbf{Ax} = \lambda \mathbf{x}$:

$$\mathbf{r} := \mathbf{Az} - \rho_{\mathbf{A}}(\mathbf{z})\mathbf{z} \quad , \quad \rho_{\mathbf{A}}(\mathbf{z}) = \text{Rayleigh quotient} \rightarrow \text{Def. 4.3.1} \quad .$$

Note: only *direction* of $\mathbf{A}^{-1}\mathbf{z}$ matters in inverse iteration (4.3.4)

$$(\mathbf{A}^{-1}\mathbf{z}) \parallel (\mathbf{z} - \mathbf{A}^{-1}(\mathbf{Az} - \rho_{\mathbf{A}}(\mathbf{z})\mathbf{z})) \Rightarrow \text{defines same next iterate!}$$



[**Preconditioned inverse iteration** (PINVIT) for s.p.d. \mathbf{A}]

$$\mathbf{z}^{(0)} \text{ arbitrary, } \quad \mathbf{w} = \mathbf{z}^{(k-1)} - \mathbf{B}^{-1}(\mathbf{A}\mathbf{z}^{(k-1)} - \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{z}^{(k-1)}), \quad k = 1, 2, \dots \quad (4.3.5)$$

$$\mathbf{z}^{(k)} = \frac{\mathbf{w}}{\|\mathbf{w}\|_2},$$

Code 4.3.9: preconditioned inverse iteration (4.3.5)

```

1 from numpy import dot, ones, tile, array, linspace
2 from numpy.linalg import norm
3 from scipy.sparse import spdiags
4 from scipy.sparse.linalg.eigen.arpack import arpack
5 from scipy.sparse.linalg import spsolve
6
7 def pinvit(evA, invB, tol, maxit):
8     n = A.shape[0]
9     #z = ones(n)/n
10    z = linspace(1, n, n)
11    z = z/norm(z)
12
13    res = []; rho = 0
14    for k in xrange(maxit):
15        v = evA(z)
16        rhon = dot(v, z) # Rayleigh quotient
17        r = v - rhon*z # residual
18        z = z - invB(r) # iteration (4.3.5)

```

```
19     z /= norm(z) # normalization
20     res += [rhon] # tracking iteration
21     if abs(rho-rhon) < tol*abs(rhon): break
22     else: rho = rhon
23 lmin = dot((evA(z)),z)
24 res += [lmin]
25 return lmin, z, res
26
27 def getdiag(A,k):
28     # A in diagonal matrix format
29     ind = abs(k+A.offsets).argmin()
30     return A.data[ind,:]
31
32 # -----
33 if __name__=='__main__':
34     k = 1; tol = 1e-13; maxit = 50
35     errC = []; errB = []
36     import matplotlib.pyplot as plt
37     for n in [20, 50, 100, 200]:
38         a = tile([1./n, -1., 2*(1+1./n), -1., 1./n], (n,1))
39         A = spdiags(a.T,[-n/2,-1,0,1,n/2],n,n)
40         C = A.tocsr() # more efficient sparse format
41         lam = min(abs(arpack.eigen_symmetric(C, which='SM')[0]))
42         evC = lambda x: C*x
```

```
43 # inverse iteration
44 invB = lambda x: spsolve(C,x)
45 lmin, z, rn = pinvit(evC, invB, tol, maxit)
46 err = abs(rn - lam)/lam
47 errC += [err]
48 txt = 'n='+str(n)
49 plt.semilogy(err, '+', label='A-1, '+ txt)
50 print lmin, lam, err
51 # preconditioned inverse iteration
52 b = array([getdiag(A,-1), A.diagonal(), getdiag(A,1)])
53 B = spdiags(b, [-1,0,1], n,n)
54 B = B.tocsr()
55 invB = lambda x: spsolve(B,x)
56 lmin, z, rn = pinvit(evC, invB, tol, maxit)
57 err = abs(rn - lam)/lam
58 errB += [err]
59 plt.semilogy(err, 'o', label='B-1, '+ txt)
60 print lmin, lam, err
61
62 plt.xlabel('#_iteration_step')
63 plt.ylabel('error_in_approximation_for_λmin')
64 plt.legend()
65 plt.savefig('pinvitD.eps')
66 plt.show()
```

Computational effort:

1 matrix \times vector

1 evaluation of preconditioner

A few AXPY-operations

Example 4.3.10 (Convergence of PINVIT).

S.p.d. matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, tridiagonal preconditioner

Monitored: error decay during iteration of Code 4.3.8: $|\rho_{\mathbf{A}}(\mathbf{z}^{(k)}) - \lambda_{\min}(\mathbf{A})|$

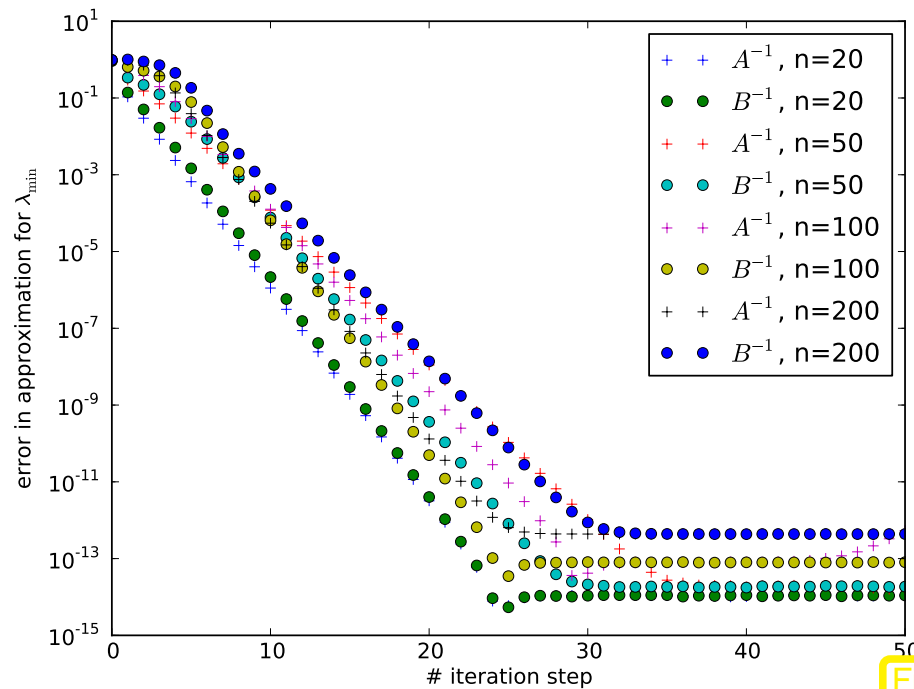


Fig. 38

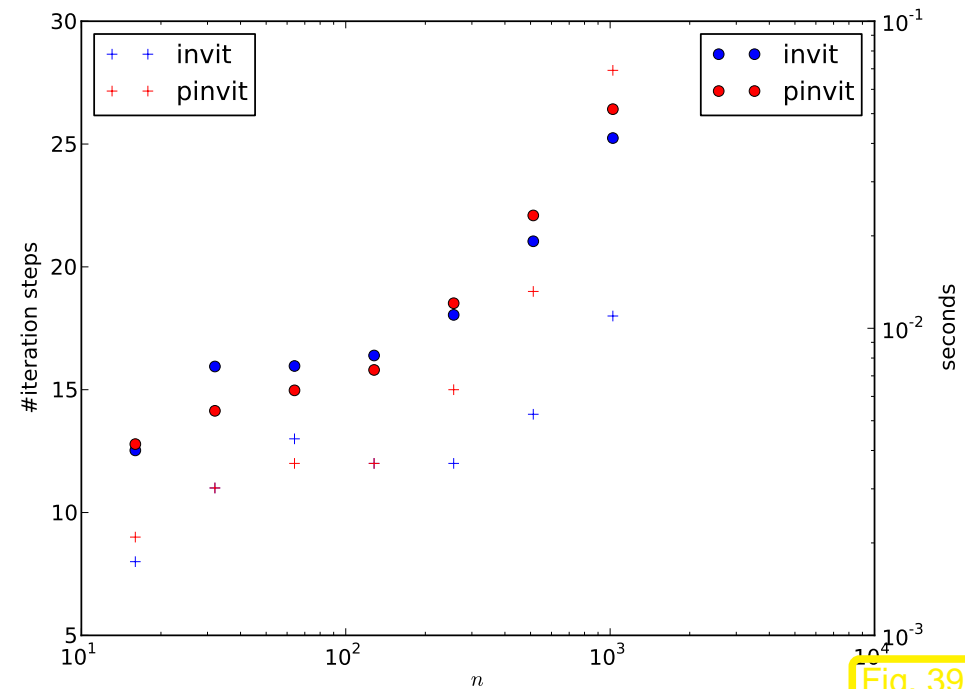


Fig. 39

Observation: linear convergence of eigenvectors also for PINVIT.



Theory:

- linear convergence of (4.3.5)
- fast convergence, if spectral condition number $\kappa(\mathbf{B}^{-1}\mathbf{A})$ small

The theory of PINVIT is based on the identity

$$\mathbf{w} = \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{A}^{-1}\mathbf{z}^{(k-1)} + (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})(\mathbf{z}^{(k-1)} - \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{A}^{-1}\mathbf{z}^{(k-1)}) . \quad (4.3.6)$$

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.

4.3.4 Subspace iterations

Task: Compute m , $m \ll n$, of the largest/smallest (in modulus) eigenvalues of $\mathbf{A} = \mathbf{A}^H \in \mathbb{C}^{n,n}$ and associated eigenvectors.

Recall that this task has to be tackled in step ② of the image segmentation algorithm Alg. ??.

Preliminary considerations:

According to Cor. 4.1.7: 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 $\mathbf{D} = \text{diag}(\lambda_1, \dots, \lambda_n)$, $\lambda_j \in \mathbb{R}$, $\lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$ \mathbf{U} orthogonal. Thus, $\mathbf{u}_j := (\mathbf{U})_{:,j}$, $j = 1, \dots, n$, are (mutually orthogonal) eigenvectors of \mathbf{A} .

Gradinaru
D-MATH

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 (4.3.2) 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 (??):


```

v = A*v; w = A*w
v = v/norm(v); w = w - dot(v,w)*v; w = w/norm(w)

```

Analysis through eigenvector expansions ($\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$, $\|\mathbf{v}\|_2 = \|\mathbf{w}\|_2 = 1$)

$$\mathbf{v} = \sum_{i=1}^n \alpha_j \mathbf{u}_j \quad , \quad \mathbf{w} = \sum_{i=1}^n \beta_j \mathbf{u}_j \quad ,$$

$$\Rightarrow \mathbf{A}\mathbf{v} = \sum_{i=1}^n \lambda_j \alpha_j \mathbf{u}_j \quad , \quad \mathbf{A}\mathbf{w} = \sum_{i=1}^n \lambda_j \beta_j \mathbf{u}_j \quad ,$$

$$\mathbf{v}_0 := \frac{\mathbf{v}}{\|\mathbf{v}\|_2} = \left(\sum_{i=1}^n \lambda_j^2 \alpha_j^2 \right)^{-1/2} \sum_{i=1}^n \lambda_j \alpha_j \mathbf{u}_j \quad ,$$

$$\mathbf{A}\mathbf{w} - (\mathbf{v}_0^T \mathbf{A}\mathbf{w}) \mathbf{v}_0 = \sum_{i=1}^n \left(\beta_j - \left(\sum_{i=1}^n \lambda_j^2 \alpha_j \beta_j / \sum_{i=1}^n \lambda_j^2 \alpha_j^2 \right) \alpha_j \right) \lambda_j \mathbf{u}_j \quad .$$

We notice that \mathbf{v} is just mapped to the next iterate in the regular direct power iteration (4.3.2). 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).

$$\mathbf{z} := \mathbf{A}\mathbf{w} - (\mathbf{v}_0^T \mathbf{A}\mathbf{w}) \mathbf{v}_0 = 0 \cdot \mathbf{u}_n + \sum_{i=1}^{n-1} \lambda_j \beta_j \mathbf{u}_j \quad ,$$

$$\mathbf{w}^{(\text{new})} := \frac{\mathbf{z}}{\|\mathbf{z}\|_2} = \left(\sum_{i=1}^{n-1} \lambda_j^2 \beta_j^2 \right)^{-1/2} \sum_{i=1}^{n-1} \lambda_j \beta_j \mathbf{u}_j .$$

The sequence $\mathbf{w}^{(k)}$ produced by repeated application of the mapping given by Code 4.3.10 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 $\lambda_{n-1}/\lambda_{n-2}$, see Thm. 4.3.2.

Remark 4.3.12 (Generalized normalization).

The following two code snippets perform the same function, cf. Code 4.3.10:



```
v = v/norm(v)
w = w - dot(v,w)*v; w = w/norm(w)
```

```
1 Q, R = qr(mat([v,w]))
2 v = Q[:,0]; w = Q[:,1]
```

Explanation ➤ Rem. 2.1.5

We revisit the above setting, Code 4.3.10. Is it possible to use the “ \mathbf{w} -sequence” to accelerate the convergence of the “ \mathbf{v} -sequence”?

Recall that by the min-max theorem Thm. ??

$$\mathbf{u}_n = \operatorname{argmax}_{\mathbf{x} \in \mathbb{R}^n} \rho_{\mathbf{A}}(\mathbf{x}) \quad , \quad \mathbf{u}_{n-1} = \operatorname{argmax}_{\mathbf{x} \in \mathbb{R}^n, \mathbf{x} \perp \mathbf{u}_n} \rho_{\mathbf{A}}(\mathbf{x}) . \quad (4.3.7)$$

Idea: maximize Rayleigh quotient over $\operatorname{Span}\{\mathbf{v}, \mathbf{w}\}$, where \mathbf{v}, \mathbf{w} are output by Code 4.3.10. This leads to the optimization problem

$$(\alpha^*, \beta^*) := \operatorname{argmax}_{\alpha, \beta \in \mathbb{R}, \alpha^2 + \beta^2 = 1} \rho_{\mathbf{A}}(\alpha \mathbf{v} + \beta \mathbf{w}) = \operatorname{argmax}_{\alpha, \beta \in \mathbb{R}, \alpha^2 + \beta^2 = 1} \rho_{(\mathbf{v}, \mathbf{w})^T \mathbf{A}(\mathbf{v}, \mathbf{w})} \left(\begin{pmatrix} \alpha \\ \beta \end{pmatrix} \right) . \quad (4.3.8)$$

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 4.3.10.

Then, orthogonalizing \mathbf{w} w.r.t \mathbf{v}^* will produce a new iterate \mathbf{w}^* .

Again the min-max theorem Thm. ?? tells us that we can find $(\alpha^*, \beta^*)^T$ as eigenvector to the largest eigenvalue of

$$(\mathbf{v}, \mathbf{w})^T \mathbf{A}(\mathbf{v}, \mathbf{w}) \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \lambda \begin{pmatrix} \alpha \\ \beta \end{pmatrix} . \quad (4.3.9)$$

Since eigenvectors of symmetric matrices are mutually orthogonal, we find $\mathbf{w}^* = \alpha_2 \mathbf{v} + \beta_2 \mathbf{w}$, where $(\alpha_2, \beta_2)^T$ is the eigenvector of (4.3.9) belonging to the smallest eigenvalue. This assumes orthonormal vectors \mathbf{v}, \mathbf{w} .

Summing up :

Code 4.3.13: one step of subspace power iteration, $m = 2$, with Ritz projection

```
1 v = A*v; w = A*w; Q, R = qr(mat(v,w)); [U,D] = eig(Q.H*A*Q)
2 w = Q*U(:,1); v = Q*U(:,2)
```

General technique:

Ritz projection

= “projection of a (symmetric) eigenvalue problem onto a subspace”

Example: Ritz projection of $\mathbf{Ax} = \lambda \mathbf{x}$ onto $\text{Span}\{\mathbf{v}, \mathbf{w}\}$:

$$(\mathbf{v}, \mathbf{w})^T \mathbf{A}(\mathbf{v}, \mathbf{w}) \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \lambda(\mathbf{v}, \mathbf{w})^T (\mathbf{v}, \mathbf{w}) \begin{pmatrix} \alpha \\ \beta \end{pmatrix} .$$

More general: Ritz projection of $\mathbf{Ax} = \lambda \mathbf{x}$ onto $\text{Im}(\mathbf{V})$ (subspace spanned by columns of \mathbf{V})

$$\mathbf{V}^H \mathbf{A} \mathbf{V} \mathbf{w} = \lambda \mathbf{V}^H \mathbf{V} \mathbf{w} . \quad (4.3.10)$$

If \mathbf{V} is unitary, then this generalized eigenvalue problem will become a standard linear eigenvalue problem.

Note that the orthogonalization step in Code 4.3.12 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. ??), cf. the discussion in Sect. 2.1.

In implementations the vectors \mathbf{v} , \mathbf{w} can be collected in a matrix $\mathbf{V} \in \mathbb{R}^{n,2}$:

Code 4.3.14: one step of subspace power iteration with Ritz projection, matrix version

```
V = A*V; Q,R = qr(V); U,D = eig(Q.H*A*Q); V = Q*U
```

Gradinaru
D-MATH

Algorithm 4.3.15 (Subspace variant of direct power method with Ritz projection).

Assumption: $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$, $k \ll n$

Subspace variant of direct power method for s.p.d. A

```

n = A.shape[0]; V = mat(rand(n,m)); d = mat(zeros()(k,1))
for i in xrange(maxit):
    V=A*V;
    Q, R = qr(V); Q = mat(Q)
    T=Q.T*A*Q;
    dn, S = eigh(T); S = mat(S)
    dn, S = sortEwEV(dn, S)
    V = Q*S
    res = norm(dn - d)
    if(res<tol): break
    d = dn
return d[:k], V[:, :k]

```

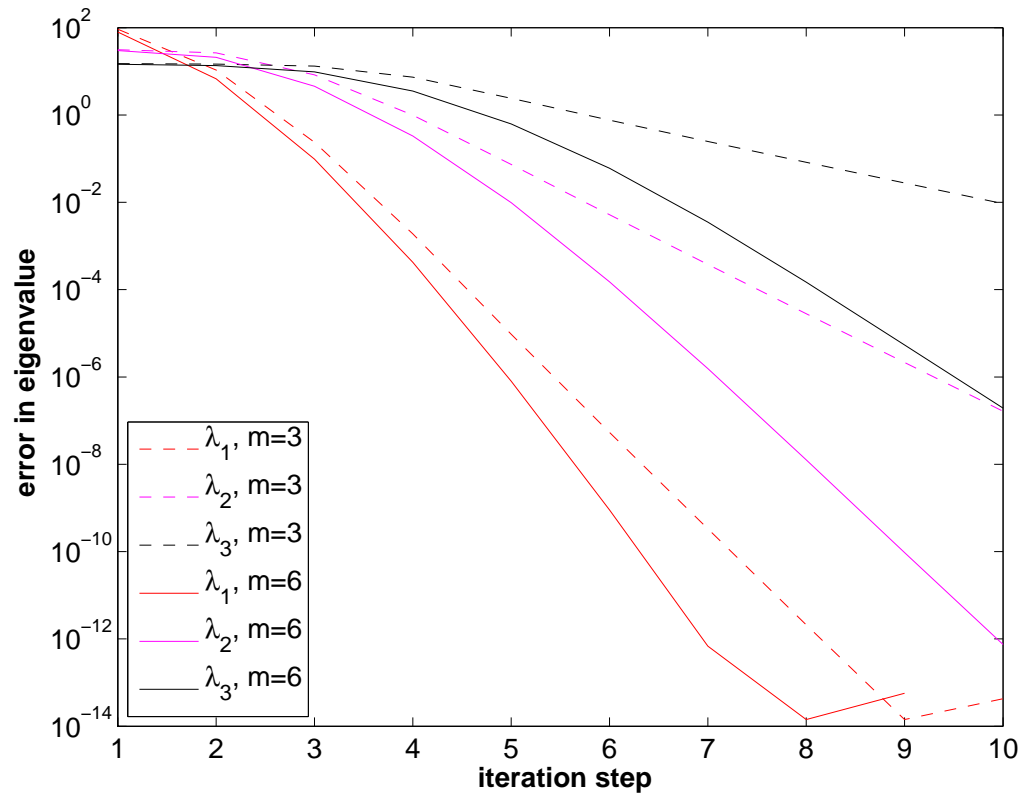
(4.3.11)

Ritz projection

Generalized normalization to $\|z\|_2 = 1$

Gradinaru
D-MATH

Example 4.3.16 (Convergence of subspace variant of direct power method).



S.p.d. test matrix: $a_{ij} := \min\{\frac{i}{j}, \frac{j}{i}\}$
 $n=200$

“Initial eigenvector guesses”:

$V = \text{eye}(n, m);$

- Observation:
linear convergence of eigenvalues
- choice $m > k$ boosts convergence
of eigenvalues

◇ Gradinaru
D-MATH

Remark 4.3.17 (Subspace power methods).

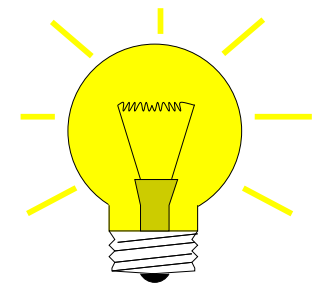
Analogous to Alg. 4.3.15: construction of subspace variants of inverse iteration (\rightarrow Code 4.3.3), PINVIT (4.3.5), and Rayleigh quotient iteration (4.3.6).

△

4.4 Krylov Subspace Methods

All power methods (\rightarrow Sect. 4.3) for the eigenvalue problem (EVP) $\mathbf{Ax} = \lambda\mathbf{x}$ only rely on the last iterate to determine the next one (1-point methods, cf. (1.1.1))

- NO MEMORY, “Memory for power iterations”: use information from previous iterates
- a direct method is fine for a small sparse matrix



Idea:

Better $\mathbf{v}^{(k)}$ from Ritz projection onto $V := \text{Span} \left\{ \mathbf{v}^{(0)}, \dots, \mathbf{v}^{(k)} \right\}$
(= space spanned by previous iterates)

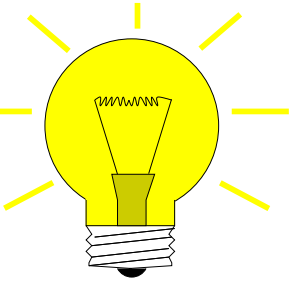
Definition 4.4.1 (Krylov space).

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

$$\mathcal{K}_l(\mathbf{A}, \mathbf{z}) := \text{Span} \left\{ \mathbf{z}, \mathbf{Az}, \dots, \mathbf{A}^{l-1}\mathbf{z} \right\} .$$

Equivalent definition:

$$\mathcal{K}_l(\mathbf{A}, \mathbf{z}) = \{p(\mathbf{A})\mathbf{z} : p \text{ polynomial of degree } \leq l\}$$



- Idea:
- Orthonormal basis of $\mathcal{K}_l(\mathbf{A}, \mathbf{z})$?
 - use explicit **Gram-Schmidt orthogonalization**

Details: inductive approach: given $\{\mathbf{v}_1, \dots, \mathbf{v}_m\}$ ONB of $\mathcal{K}_m(\mathbf{A}, \mathbf{z})$

$$\blacktriangleright \quad \tilde{\mathbf{v}}_{m+1} := \mathbf{A}\mathbf{v}_m - \sum_{j=1}^m (\mathbf{v}_j^H \mathbf{A}\mathbf{v}_m) \mathbf{v}_j, \quad \mathbf{v}_{m+1} := \frac{\tilde{\mathbf{v}}_{m+1}}{\|\tilde{\mathbf{v}}_{m+1}\|_2} \Rightarrow \mathbf{v}_{m+1} \perp \mathcal{K}_m(\mathbf{A}, \mathbf{z}). \quad (4.4.1)$$

orthogonal

(Gram-Schmidt)

Code 4.4.1: Arnoldi process

In step l :

$1 \times \mathbf{A} \times \text{vector}$
 $m+1$ dot products
 m AXPY-operations
 n divisions

➤ Computational cost
 for m steps, if at
 most k non-zero en-
 tries in each row of
 \mathbf{A} : $O(nkm^2)$

$H(m+1, m) = 0 \rightarrow$

STOP !

```

1 def arnoldi(A, v0, k):
2     V = mat(zeros((v0.shape[0], k+1), dtype=complex))
3     V[:, 0] = v0.copy() / norm(v0)
4     H = mat(zeros((k+1, k), dtype=complex))
5     for m in xrange(k):
6         vt = multMv(A, V[:, m])
7         for j in xrange(m+1):
8             H[j, m] = (V[:, j].H * vt)[0, 0]
9             vt -= H[j, m] * V[:, j]
10        H[m+1, m] = norm(vt);
11        V[:, m+1] = vt.copy() / H[m+1, m]
12    return V, H
  
```

If it does not stop prematurely, the Arnoldi process of Code 4.4.0 will yield an *orthonormal basis* (OBN) 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 4.4.0, meaning of output H:

$$\mathbf{V}_l = [\mathbf{v}_1, \dots, \mathbf{v}_l] : \mathbf{A}\mathbf{V}_l = \mathbf{V}_{l+1}\tilde{\mathbf{H}}_l \quad , \quad \tilde{\mathbf{H}}_l \in \mathbb{K}^{l+1,l} \text{ mit } \tilde{h}_{ij} = \begin{cases} \mathbf{v}_i^H \mathbf{A} \mathbf{v}_j & , \text{ if } i \leq j \text{ ,} \\ \|\tilde{\mathbf{v}}_i\|_2 & , \text{ if } i = j + 1 \text{ ,} \\ 0 & \text{else.} \end{cases}$$

➔ $\tilde{\mathbf{H}}_l$ = non-square upper Hessenberg matrices

$$\left(\begin{array}{c} \text{A} \end{array} \right) \left(\begin{array}{c} \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_l \end{array} \right) = \left(\begin{array}{c} \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_l \\ \mathbf{v}_{l+1} \end{array} \right) \left(\begin{array}{c} \tilde{\mathbf{H}}_l \end{array} \right)$$

Translate Code 4.4.0 to matrix calculus:

Lemma 4.4.2 (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 (\triangleright Lanczos process)

Proof. Direct from Gram-Schmidt orthogonalization and inspection of Code 4.4.0. □

Remark 4.4.2 (Arnoldi process and Ritz projection).

Interpretation of Lemma 4.4.2 (iii) & (i):

$\mathbf{H}_l \mathbf{x} = \lambda \mathbf{x}$ is a (generalized) Ritz projection of EVP $\mathbf{A} \mathbf{x} = \lambda \mathbf{x}$

$$\text{In } l\text{-th step: } \lambda_n \approx \mu_l^{(l)}, \lambda_{n-1} \approx \mu_{l-1}^{(l)}, \dots, \lambda_1 \approx \mu_1^{(l)},$$

$$\sigma(\mathbf{H}_l) = \{\mu_1^{(l)}, \dots, \mu_l^{(l)}\}, \quad |\mu_1^{(l)}| \leq |\mu_2^{(l)}| \leq \dots \leq |\mu_l^{(l)}|.$$

Code 4.4.3: Arnoldi eigenvalue approximation

```

1 import scipy.linalg
2 import scipy.io
3 from scipy.linalg import norm
4 from scipy import mat, eye, zeros, array, arange, sqrt, real, imag
5 from arnoldi import multMv, arnoldi, randomvector
6 from scipy.sparse.linalg.eigen.arnoldi import speigs
7 from scipy.sparse.linalg.eigen.arnoldi import arpack
8 import time
9
10 def delta(i, j):
11     if i==j: return 1
12     else: return 0
13
14 def ConstructMatrix(N, case='minij'):
15     H = mat(zeros([N,N]))
16     for i in xrange(N):
17         for j in xrange(N):

```

```
18         if case=='sqrts':
19             H[i,j] = 1L * sqrt((i+1)**2+(j+1)**2)+(i+1)*delta(i,j)
20         else:
21             H[i,j] = float( 1+min(i,j) )
22     return H
23
24 def sortEwEV(d,v, focus=abs):
25     u = v.copy()
26     #l = abs(d).argsort()
27     l = focus(d).argsort()
28     a = d[l]
29     u = u[:,l]
30     return a,u
31
32 if __name__ == "__main__":
33     # construct/import matrix
34     #sparseformat = True
35     sparseformat = False
36     if sparseformat:
37         # look at http://math.nist.gov/MatrixMarket/
38         L = scipy.io.mmread('SomeMatrices/qc324.mtx.gz')
39     else:
40         #L = scipy.io.loadmat('Lmat.mat')['L']
41         L = ConstructMatrix(1000)
```

```
42     #L = ConstructMatrix(400,case='sqrts')
43 #print L
44 #-----
45 N = L.shape[0]
46 nev = 10      # compute the largest nev eigenvalues and vectors
47 print 'full_matrix_approach'
48 t0 = time.clock()
49 if sparseformat:
50     d0,V0 = scipy.linalg.eig(L.todense())
51 else:
52     d0,V0 = scipy.linalg.eig(L)
53 print time.clock() - t0 , 'sec'
54 V0 = mat(V0)
55 # let us check if the eigenvectors are orthogonal
56 print abs(V0.H*V0 - eye(V0.shape[1])).max()
57 # consider only the largest in abs.value
58 a0,U0 = sortEwEV(d0,V0)
59 na = U0.shape[0]
60 a0 = a0[na-nev:]
61 U0 = U0[:,na-nev:]
62 print 'a0', a0
63 #-----
64 print 'my_arnoldi_procedure'
65 # compute the largest na eigenvalues and vectors
```

```
# but only the first eigenvectors are relevant
v = randomvector(N)
na = 3*nev+1 # take 3 times more steps
if sparseformat:
    Lt = L.todense()
    t0 = time.clock()
    A,H = arnoldi(Lt, v, na)
else:
    t0 = time.clock()
    A,H = arnoldi(L, v, na)
print time.clock() - t0, 'sec'
d1,V1 = scipy.linalg.eig(H[: -1 ,:])
Z = A[:, : -1]*V1
a1,U1 = sortEwEV(d1, Z)
a1 = a1[na-nev:]
U1 = U1[:, na-nev:]
print 'a1 ', a1
#-----
print 'same_with_ARPACK'
if sparseformat:
    L = L.tocsr()
    t0 = time.clock()
    d2,V2 = speigs.ARPACK_eigs(L.matvec, N, nev, which='LM')
else:
```



```
00     t0 = time.clock()
01     d2,V2 = arpack.eigen(L, k=nev, ncv=na, which='LM')
02     print time.clock() - t0 , 'sec'
03     a2,U2 = sortEwEV(d2,V2)
04     U2 = mat(U2)
05     print 'a2', a2
06     #-----
07     e1 = abs(a0-a1)/abs(a0)
08     e2 = abs(a0-a2)/abs(a0)
09     print 'error_in_eigenvalues:\n' , e1, '\n', e2
10     print 'residuals_in_eigenvectors:'
11     z0 = [norm(L*U0[:,k] - a0[k]*U0[:,k]) for k in xrange(nev) ]
12     print 'eig(L)', z0
13     z1 = [norm(L*U1[:,k] - a1[k]*U1[:,k]) for k in xrange(nev) ]
14     print 'arnoldi(L)', z1
15     z2 = [norm(L*U2[:,k] - a2[k]*U2[:,k]) for k in xrange(nev) ]
16     print 'arpack(L)', z2
17     #-----
18     # plot residuals in eigenvectors
19     from pylab import loglog, semilogy, plot, show, legend, xlabel,
20         ylabel, savefig, rcParams#, xlim,
21         ylim
22     plotf = loglog
23     #plotf = semilogy
```

```
12 params = { 'backend': 'png',                               # or 'ps'
13            'axes.labelsize': 20,
14            'text.fontsize': 20,
15            'legend.fontsize': 20,
16            'xtick.labelsize': 16,
17            'lines.markersize' : 12,
18            'ytick.labelsize': 16#,
19            #'text.usetex': True
20        }
21 rcParams.update(params)
22 #-----
23 plotf(abs(a0), z0, 'g—+')
24 plotf(abs(a0), z1, 'r—o')
25 plotf(abs(a0), z2, 'b—v')
26 legend(( 'eig', 'arnoldi', 'arpack' ),2)
27 xlabel(' | Eigenvalue | ')
28 ylabel(' Residual_of_Eigenvector ')
29 savefig('resEVarpack.eps')
30 show()
31 #-----
32 plotf(abs(a0), e1, 'ro')
33 plotf(abs(a0), e2, 'bv')
34 legend(( 'arnoldi', 'arpack' ),2)
```

```
36 xlabel( '| Eigenvalue | ' )  
37 ylabel( ' Error ' )  
38 savefig( 'errEWarpack.eps' )  
39 show()
```

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

If $\mathbf{A}^H = \mathbf{A}$, then $\mathbf{V}_m^T \mathbf{A} \mathbf{V}_m$ is a tridiagonal matrix.

$$\mathbf{V}_l^H \mathbf{A} \mathbf{V}_l = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \\ & & & \ddots & \ddots & \\ & & & & \ddots & \ddots & \\ & & & & & \ddots & \ddots & \beta_{k-1} \\ & & & & & \beta_{k-1} & \alpha_k \end{pmatrix} =: \mathbf{T}_l \in \mathbb{K}^{k,k} \quad [\text{tridiagonal matrix}]$$

Code 4.4.4: Lanczos process

```

1 def lanczos(A, v0, k):
2     V = mat(zeros((v0.shape[0], k+1)))
3     alpha = zeros(k)
4     bet = zeros(k+1)
5     for m in xrange(k):
6         vt = multMv(A, V[:, m])
7         if m > 0: vt -= bet[m] * V[:, m-1]
8         alpha[m] = (V[:, m].H * vt)[0, 0]
9         vt -= alpha[m] * V[:, m]
10        bet[m+1] = norm(vt)
11        V[:, m+1] = vt.copy() / bet[m+1]
12        rbet = bet[1:-1]
13        T = diag(alpha) + diag(rbet, 1) +
            diag(rbet, -1)
14    return V, T

```

Algorithm for computing V_l and T_l :

Lanczos process

Computational effort/step:

- 1 \times $A \times$ vector
- 2 dot products
- 2 AXPY-operations
- 1 division

Total computational effort for l steps of Lanczos process, if A has at most k non-zero entries per row:
 $O(nkl)$

Note: Code 4.4.3 assumes that no residual vanishes. This could happen, if 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 A , cf. Rem. 4.3.2.

Convergence (what we expect from the above considerations) \rightarrow [14, Sect. 8.5]

In l -th step: $\lambda_n \approx \mu_l^{(l)}$, $\lambda_{n-1} \approx \mu_{l-1}^{(l)}$, \dots , $\lambda_1 \approx \mu_1^{(l)}$,
 $\sigma(\mathbf{T}_l) = \{\mu_1^{(l)}, \dots, \mu_l^{(l)}\}$, $\mu_1^{(l)} \leq \mu_2^{(l)} \leq \dots \leq \mu_l^{(l)}$.

Example 4.4.5 (Lanczos process for eigenvalue computation).

\mathbf{A} with eigenvalues $1, 2, \dots, 100$

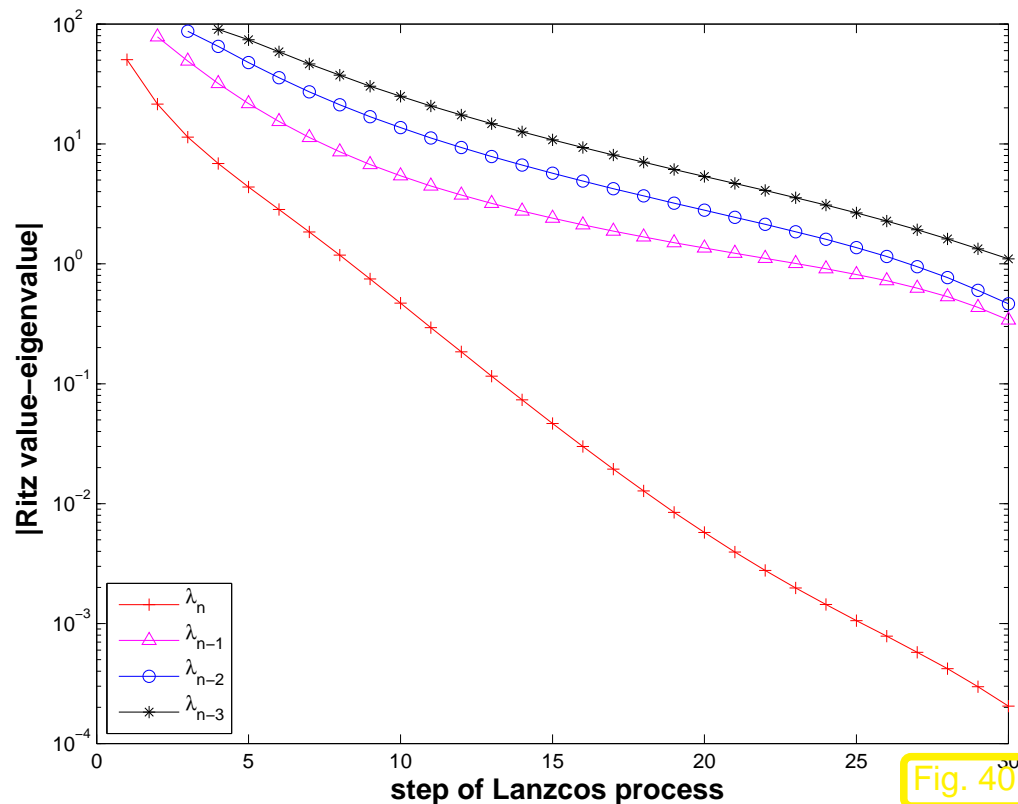


Fig. 40

$\mathbf{A} = [\min(i, j)]$, $n=100$

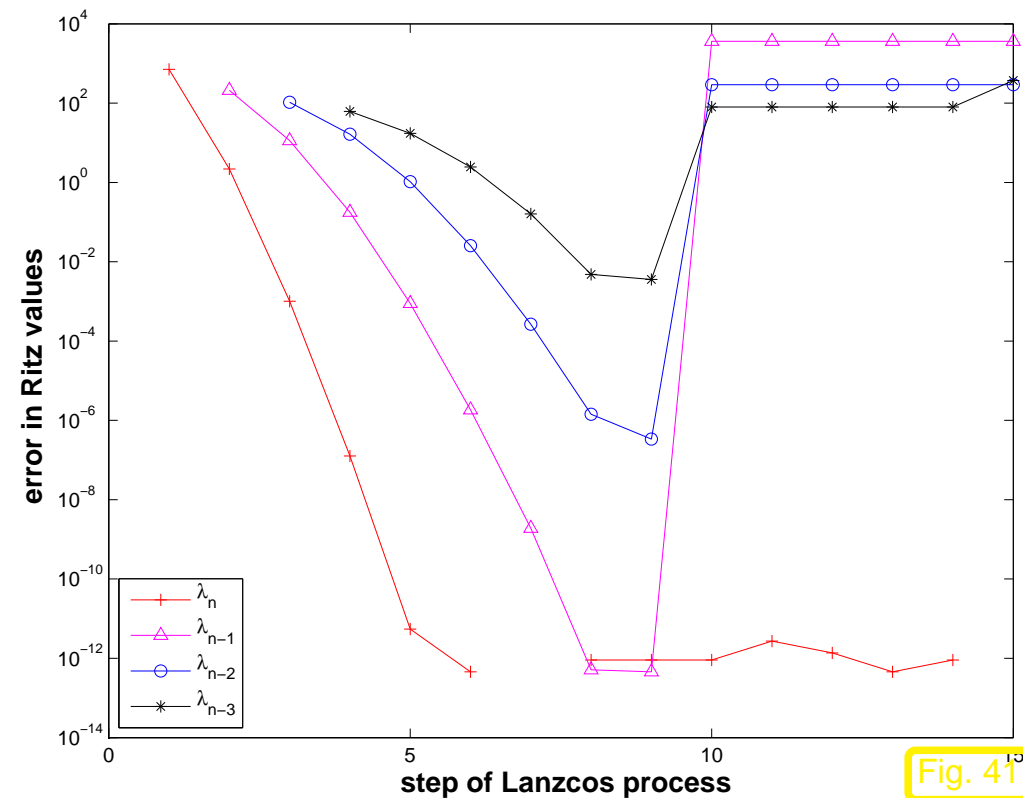


Fig. 41

Observation: 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



Example 4.4.6 (Impact of roundoff on Lanczos process).

$$\mathbf{A} \in \mathbb{R}^{10,10}, \quad a_{ij} = \min\{i, j\}.$$

$$\mathbf{A} = [\min(i, j)], \quad n=10$$

$$\mathbf{T} = \begin{pmatrix} 38.500000 & 14.813845 & & & & & & & & \\ 14.813845 & 9.642857 & 2.062955 & & & & & & & \\ & 2.062955 & 2.720779 & 0.776284 & & & & & & \\ & & 0.776284 & 1.336364 & 0.385013 & & & & & \\ & & & 0.385013 & 0.826316 & 0.215431 & & & & \\ & & & & 0.215431 & 0.582380 & 0.126781 & & & \\ & & & & & 0.126781 & 0.446860 & 0.074650 & & \\ & & & & & & 0.074650 & 0.363803 & 0.043121 & \\ & & & & & & & 0.043121 & 3.820888 & 11.991094 \\ & & & & & & & & 11.991094 & 41.254286 \end{pmatrix}$$

$$\sigma(\mathbf{A}) = \{0.255680, 0.273787, 0.307979, 0.366209, 0.465233, 0.643104, 1.000000, 1.873023, 5.048917, 44.766069\}$$

$$\sigma(\mathbf{T}) = \{0.263867, 0.303001, 0.365376, 0.465199, 0.643104, 1.000000, 1.873023, 5.048917, 44.765976, 44.766069\}$$

► Uncanny cluster of computed eigenvalues of **T** (“ghost eigenvalues”, [20, Sect. 9.2.5])

$$V^H V = \begin{pmatrix} 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000251 & 0.258801 & 0.883711 \\ 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000106 & 0.109470 & 0.373799 \\ 0.000000 & -0.000000 & 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000005 & 0.005373 & 0.018347 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000096 & 0.000328 \\ 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000001 & 0.000003 \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 \\ 0.000251 & 0.000106 & 0.000005 & 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 \\ 0.258801 & 0.109470 & 0.005373 & 0.000096 & 0.000001 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & 0.000000 \\ 0.883711 & 0.373799 & 0.018347 & 0.000328 & 0.000003 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{pmatrix},$$

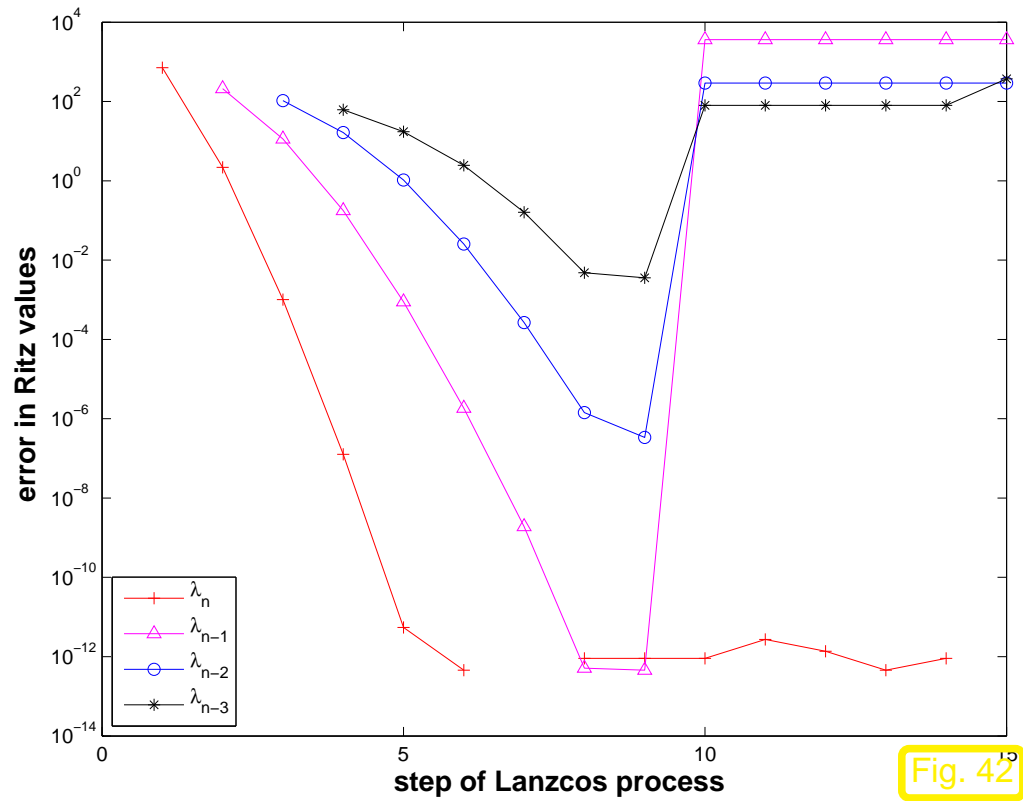
► Loss of orthogonality of residual vectors due to roundoff
(compare: impact of roundoff on CG iteration)

l	$\sigma(\mathbf{T}_l)$										Num. Meth. Phys.
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	◇

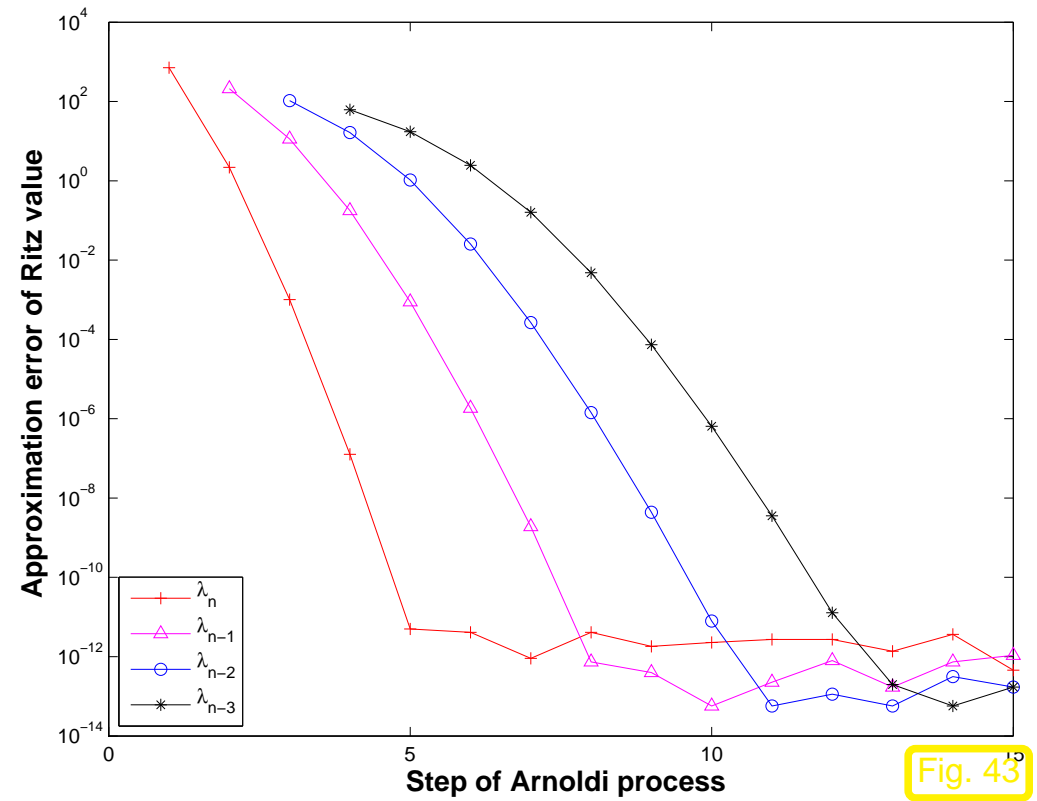
Example 4.4.7 (Stabilty of Arnoldi process).

$$\mathbf{A} \in \mathbb{R}^{100,100} \quad , \quad a_{ij} = \min\{i,j\} \quad .$$

$$\mathbf{A} = [\min(\mathbf{i},\mathbf{j})] , \quad \mathbf{n}=100$$



Lanczos process: Ritz values



Arnoldi process: Ritz values

Ritz values during Arnoldi process for $A = [\min(i, j)], n = 10$; \leftrightarrow Ex. 4.4.5

l	$\sigma(\mathbf{H}_l)$										Num. Meth. Phys.
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	0.276159 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})

Gradinaru
D-MATH

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 compared to the Lanczos process.

◇
 4.4
p. 256

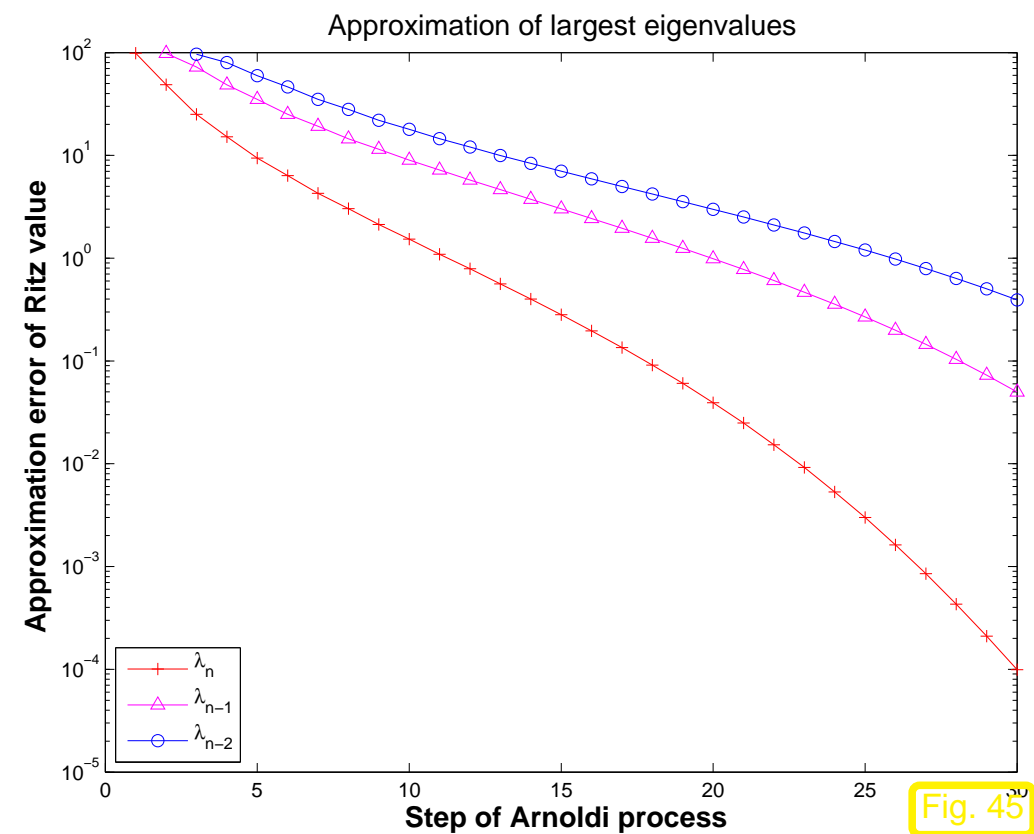
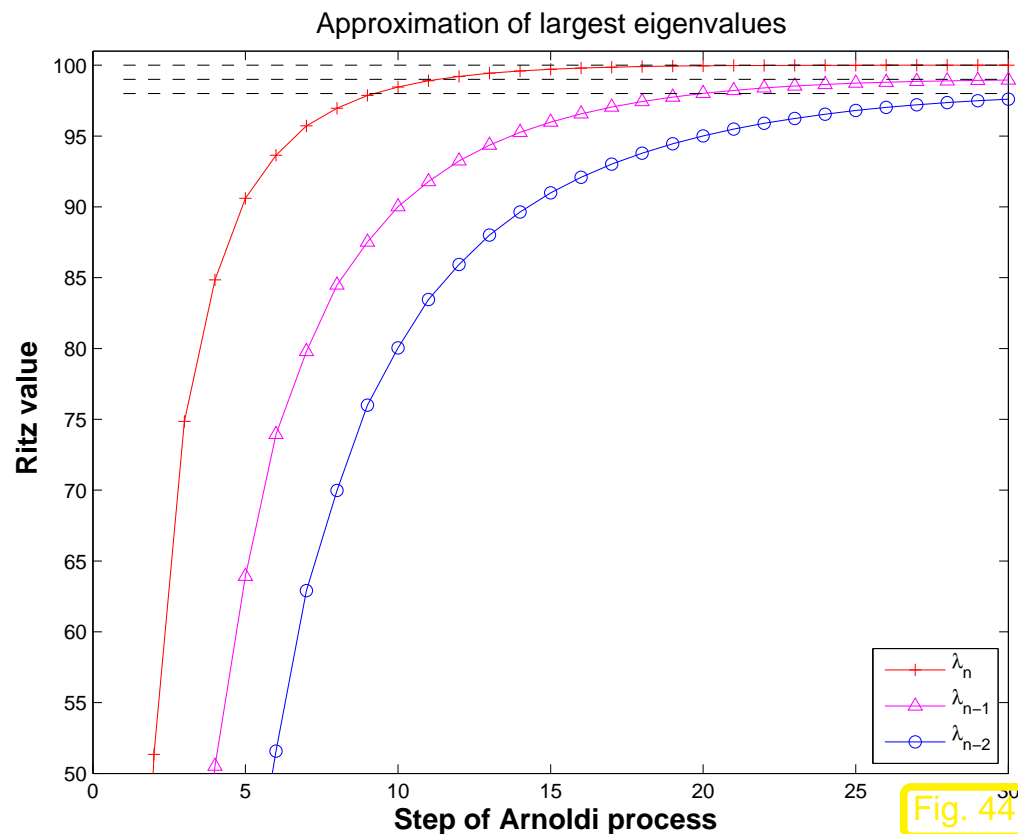
Example 4.4.8 (Eigenvalue computation with Arnoldi process).

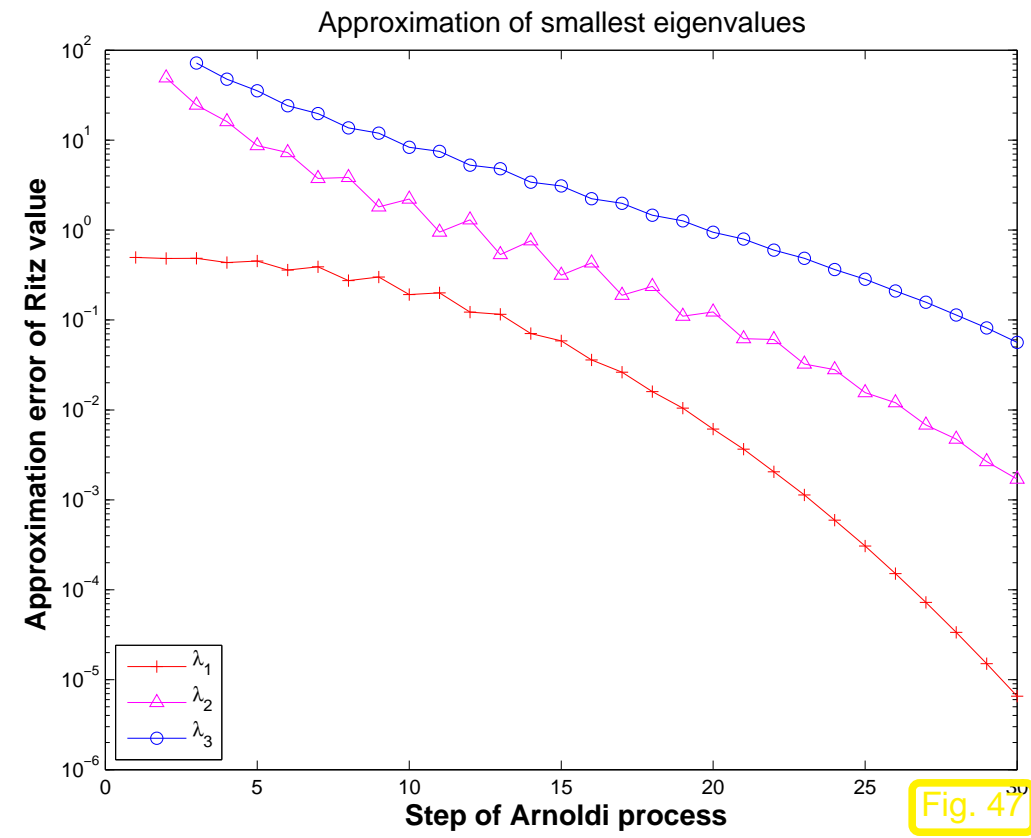
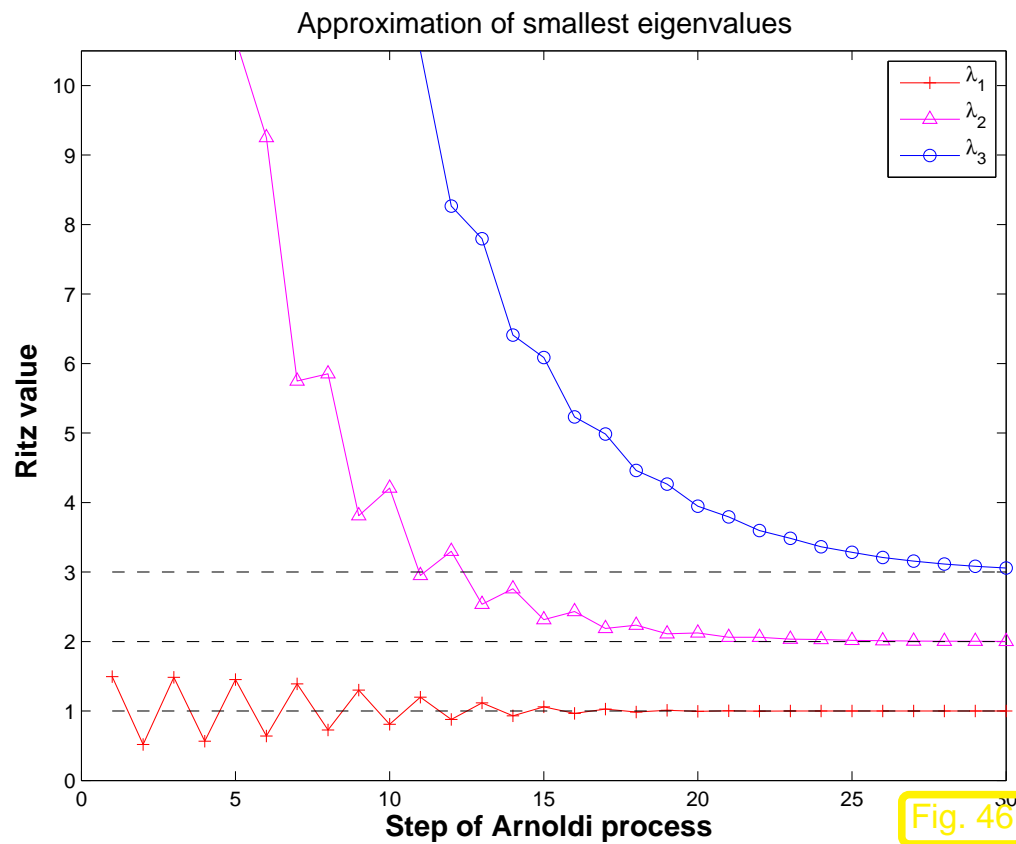
Eigenvalue approximation from Arnoldi process for *non-symmetric* A , initial vector `ones(100)` ;

```

1 n=100
2 M = 2.*eye(n) + -0.5*eye(n,k=-1) + 1.5*eye(n, k=1); M = mat(M)
3 A = M*(diag(r_[1:n+1]))*M.'

```





Observation: “vaguely linear” convergence of largest and smallest eigenvalues



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 4.4.9 (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 “**B-inner product**” $(\mathbf{x}, \mathbf{y}) \mapsto \mathbf{x}^H \mathbf{B} \mathbf{y}$.



Python-functions:

`scipy.sparse.linalg.eigen.lobpcg`: solve sparse symmetric generalized eigenproblems with Locally Optimal Block Preconditioned Conjugate Gradient Method

`scipy.linalg.eig`: solve generalized eigenvalue problem of a square full matrix

Gradinaru
D-MATH

MATLAB-functions:

`d = eigs(A,k,sigma)` : k largest/smallest eigenvalues of \mathbf{A}

`d = eigs(A,B,k,sigma)`: k largest/smallest eigenvalues for generalized EVP $\mathbf{Ax} = \lambda \mathbf{Bx}$, \mathbf{B} s.p.d.

`d = eigs(Afun,n,k)` : \mathbf{Afun} = handle to function providing matrix \times vector for $\mathbf{A}/\mathbf{A}^{-1}/\mathbf{A} - \alpha \mathbf{I}/(\mathbf{A} - \alpha \mathbf{B})^{-1}$. (Use flags to tell `eigs` about special properties of matrix behind \mathbf{Afun} .)

`eigs` just calls routines of the open source ARPACK numerical library.

4.5 Essential Skills Learned in Chapter 4

You should know:

- complexity of the direct eigensolver *eig* of Matlab
- how the direct power method works and its convergence
- the idea behind the inverse power iteration, Rayleigh quotient iteration and preconditioning
- what are Krylov methods, when and how to use them

Part II

Interpolation and Approximation

Distinguish two fundamental concepts:

(I) **data interpolation** (point interpolation, also includes CAD applications):

Given: data points $(\mathbf{x}_i, \mathbf{y}_i)$, $i = 1, \dots, m$, $\mathbf{x}_i \in D \subset \mathbb{R}^n$, $\mathbf{y}_i \in \mathbb{R}^d$

Goal: *reconstruction of a (continuous) function* $\mathbf{f} : D \mapsto \mathbb{R}^d$ satisfying **interpolation conditions**

$$f(\mathbf{x}_i) = \mathbf{y}_i, \quad i = 1, \dots, m$$

Additional requirements:

- smoothness of \mathbf{f} , e.g. $\mathbf{f} \in C^1$, etc.
- shape of \mathbf{f} (positivity, monotonicity, convexity)

Example 4.0.1 (Constitutive relations (*ger.* Kennlinien) from measurements).

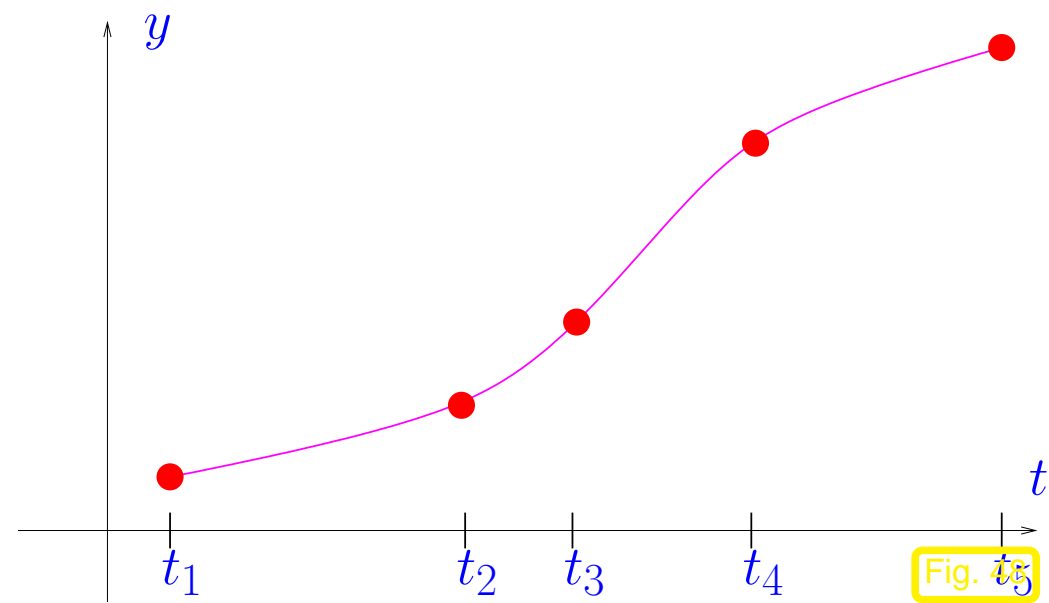
In this context: $t, y \hat{=}$ two state variables of a physical system, a functional dependence $y = y(t)$ is assumed.

Known: several *accurate* measurements

$$(t_i, y_i), \quad i = 1, \dots, m$$

Examples:

t	y
voltage U	current I
pressure p	density ρ
magnetic field H	magnetic flux B
...	...



Meaning of attribute “accurate”: justification for interpolation. If measured values y_i were affected by considerable errors, one would not impose the interpolation conditions but opt for **data fitting**.



(II) **function approximation**:

Given: function $\mathbf{f} : D \subset \mathbb{R}^n \mapsto \mathbb{R}^d$ (often in procedural form $y = \text{eval}(x)$)

Goal: Find a “simple”^(*) function $\tilde{\mathbf{f}} : D \mapsto \mathbb{R}^d$ such that the difference $\mathbf{f} - \tilde{\mathbf{f}}$ is “small”^(♣)

(*) : “simple” \sim described by small amount of information, easy to evaluate (e.g, polynomial or piecewise polynomial $\tilde{\mathbf{f}}$)

(♣) “small” \sim $\|\mathbf{f} - \tilde{\mathbf{f}}\|$ small for some norm $\|\cdot\|$ on space $C(D)$ of continuous functions, e.g. L^2 -norm
 $\|\mathbf{g}\|_2^2 := \int_D |\mathbf{g}(x)|^2 dx$, maximum norm $\|\mathbf{g}\|_\infty := \max_{x \in D} |\mathbf{g}(x)|$

Example 4.0.2 (Taylor approximation).

$$f \in C^k(I), \quad I \text{ interval}, \quad k \in \mathbb{N}, \quad T_k(t) := \frac{f^{(k)}(t_0)}{k!} (t - t_0)^k, \quad t_0 \in I.$$


The Taylor polynomial T_k of degree k approximates f in a neighbourhood $J \subset I$ of t_0 (J can be small!). The Taylor approximation is easy and direct but inefficient: a polynomial of lower degree gives the same accuracy.



Another technique:

Approximation by interpolation

$$\mathbf{f} \xrightarrow{\text{sampling}} (\mathbf{x}_i, \mathbf{y}_i := f(\mathbf{x}_i))_{i=1}^m \xrightarrow{\text{interpolation}} \tilde{\mathbf{f}}: \tilde{\mathbf{f}}(\mathbf{x}_i) = \mathbf{y}_i.$$


 free choice of nodes \mathbf{x}_i

Remark 4.0.3 (Interpolation and approximation: enabling technologies).

Approximation and interpolation are useful for several numerical tasks, like integration, differentiation and computation of the solutions of differential equations.



this is a “foundations” part of the course



Remark 4.0.4 (Function representation).

! General function $f : D \subset \mathbb{R} \mapsto \mathbb{K}$, D interval, contains an “infinite amount of information”.

? How to represent f on a computer?

➔ Idea: **parametrization**, a finite number of parameters $\alpha_1, \dots, \alpha_n$, $n \in \mathbb{N}$, characterizes f .

Special case: Representation with finite linear combination of **basis functions**

$b_j : D \subset \mathbb{R} \mapsto \mathbb{K}, j = 1, \dots, n$:

$$f = \sum_{j=1}^n \alpha_j b_j \quad , \quad \alpha_j \in \mathbb{K} .$$

→ $f \in$ finite dimensional **function space** $V_n := \text{Span} \{b_1, \dots, b_n\}$.

$b_j(t) = t^{j-1}$ leads to polynomial interpolation

$b_j(t) = \cos((j-1) \arccos t)$ leads to Chebychev interpolation

$b_j(t) = e^{2\pi i j t}$ leads to trigonometrical interpolation



5

Polynomial Interpolation

5.1 Polynomials

Notation: Vector space of the polynomials of degree $\leq k$, $k \in \mathbb{N}$:

$$\mathcal{P}_k := \{t \mapsto \alpha_k t^k + \alpha_{k-1} t^{k-1} + \cdots + \alpha_1 t + \alpha_0, \alpha_j \in \mathbb{K}\} . \quad (5.1.1)$$

Gradinaru
D-MATH

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.

Obvious: \mathcal{P}_k is a vector space. What is its dimension?

Theorem 5.1.1 (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.

Why are polynomials important in computational mathematics ?

- Easy to compute, integrate and differentiate
- Vector space & algebra
- Analysis: Taylor polynomials & power series

Remark 5.1.1 (Polynomials in python).

python: $\alpha_k t^k + \alpha_{k-1} t^{k-1} + \dots + \alpha_0 \rightarrow$ Vector $(\alpha_k, \alpha_{k-1}, \dots, \alpha_0)$ (ordered!).



Remark 5.1.2 (Horner scheme).

Evaluation of a polynomial in monomial representation:

Horner scheme

$$p(t) = (t \cdots t(t(\alpha_n t + \alpha_{n-1}) + \alpha_{n-2}) + \cdots + \alpha_1) + \alpha_0 .$$

(5.1.2)

Code 5.1.3: Horner scheme, polynomial in python format

```
def horner(p, x):  
    y = p[0]  
    for i in xrange(1, len(p)):  
        y = x * y + p[i]  
    return y
```

Asymptotic complexity: $O(n)$

Use: `numpy` “built-in”-function `polyval(p,x);`.



5.2 Newton basis and divided differences [10, Sect. 8.2.4]

Want: adding another data point should not affect *all* basis polynomials!

Tool: “update friendly” representation: **Newton basis** for \mathcal{P}_n

$$N_0(t) := 1, \quad N_1(t) := (t - t_0), \quad \dots, \quad N_n(t) := \prod_{i=0}^{n-1} (t - t_i). \quad (5.2.1)$$

Note: $N_n \in \mathcal{P}_n$ with *leading coefficient* 1.

➤ LSE for polynomial interpolation problem in Newton basis:

$$a_j \in \mathbb{R}: \quad a_0 N_0(t_j) + a_1 N_1(t_j) + \dots + a_n N_n(t_j) = y_j, \quad j = 0, \dots, n.$$

⇔ triangular linear system

$$\begin{pmatrix} 1 & 0 & \dots & 0 \\ 1 & (t_1 - t_0) & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 1 & (t_n - t_0) & \dots & \prod_{i=0}^{n-1} (t_n - t_i) \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}.$$

Solution of the system with forward substitution:

$$a_0 = y_0,$$

$$a_1 = \frac{y_1 - a_0}{t_1 - t_0} = \frac{y_1 - y_0}{t_1 - t_0},$$

$$a_2 = \frac{y_2 - a_0 - (t_2 - t_0)a_1}{(t_2 - t_0)(t_2 - t_1)} = \frac{y_2 - y_0 - (t_2 - t_0)\frac{y_1 - y_0}{t_1 - t_0}}{(t_2 - t_0)(t_2 - t_1)} = \frac{\frac{y_2 - y_0}{t_2 - t_0} - \frac{y_1 - y_0}{t_1 - t_0}}{t_2 - t_1},$$

$$\vdots$$

Observation: same quantities computed again and again !

In order to find a better algorithm, 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 $p_{0,\dots,j}$

➤ Recursion for leading coefficients $a_{\ell,\dots,m}$ of interpolating polynomials $p_{\ell,\dots,m}$, $0 \leq \ell \leq m \leq n$:

$$a_{\ell,\dots,m} = \frac{a_{\ell+1,\dots,m} - a_{\ell,\dots,m-1}}{t_m - t_\ell}.$$

Simpler and more efficient algorithm using **divided differences**:

$$\begin{aligned} y[t_i] &= y_i \\ y[t_i, \dots, t_{i+k}] &= \frac{y[t_{i+1}, \dots, t_{i+k}] - y[t_i, \dots, t_{i+k-1}]}{t_{i+k} - t_i} \end{aligned} \quad \text{(recursion)} \quad (5.2.2)$$

Recursive calculation by **divided differences scheme**

$$\begin{array}{l|l} t_0 & y[t_0] \\ & > y[t_0, t_1] \\ t_1 & y[t_1] > y[t_0, t_1, t_2] \\ & > y[t_1, t_2] > y[t_0, t_1, t_2, t_3], \\ t_2 & y[t_2] > y[t_1, t_2, t_3] \\ & > y[t_2, t_3] \\ t_3 & y[t_3] \end{array} \quad (5.2.3)$$

the elements are computed from left to right, every “>” means recursion (5.2.2).

If a new datum (t_{n+1}, y_{n+1}) is added, it is enough to compute $n + 2$ new terms

$$y[t_{n+1}], y[t_n, t_{n+1}], \dots, y[t_0, \dots, t_{n+1}].$$

Code 5.2.1: Divided differences, recursive implementation, in situ computation

```
1 def divdiff(t, y):
2     n = y.shape[0] - 1
3     if n > 0:
4         y[0:n] = divdiff(t[0:n], y[0:n])
5         for j in xrange(0,n):
6             y[n] = (y[n] - y[j]) / (t[n] - t[j])
7     return y
```

Code 5.2.2: Divided differences, non-recursive implementation, in situ computation

```
1 def divdiff(t, y):
2     n = y.shape[0] - 1
3     for l in xrange(0,n):
4         for j in xrange(l+1,n+1):
5             y[j] = (y[j] - y[l]) / (t[j] - t[l])
6     return y
```

By derivation: computed finite differences are the coefficients of interpolating polynomials in Newton basis:

$$p(t) = a_0 + a_1(t - t_0) + a_2(t - t_0)(t - t_1) + \cdots + a_n \prod_{j=0}^{n-1} (t - t_j) \quad (5.2.4)$$

$$a_0 = y[t_0], \quad a_1 = y[t_0, t_1], \quad a_2 = y[t_0, t_1, t_2], \quad \dots$$

“Backward evaluation” of $p(t)$ in the spirit of Horner’s scheme (\rightarrow Rem. 5.1.2, [10, Alg. 8.20]):

$$p \leftarrow a_n, \quad p \leftarrow (t - t_{n-1})p + a_{n-1}, \quad p \leftarrow (t - t_{n-2})p + a_{n-2}, \quad \dots$$

Computational effort:

- $O(n^2)$ for computation of divided differences,
- $O(n)$ for every single evaluation of $p(t)$.

Remark 5.2.3 (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 [10, Thm. 8.21].



5.3 Error estimates for polynomial interpolation

Focus: approximation of a function by global polynomial interpolation (\rightarrow Sect. ??)

Remark 5.3.1 (Approximation by polynomials).

? Is it always possible to approximate a continuous function by polynomials?

✓ Yes! Recall the **Weierstrass theorem**:

A continuous function f on the interval $[a, b] \subset \mathbb{R}$ can be uniformly approximated by polynomials.

! But not by the interpolation on a fixed mesh [42, pag. 331]:

Given a sequence of meshes of increasing size $\{\mathcal{T}_j\}_{j=1}^{\infty}$, $\mathcal{T}_j = \{x_1^{(j)}, \dots, x_j^{(j)}\} \subset [a, b]$,
 $a \leq x_1^{(j)} < x_2^{(j)} < \dots < x_j^{(j)} \leq b$, there exists a continuous function f such that
the sequence interpolating polynomials of f on \mathcal{T}_j does not converge uniformly to f as
 $j \rightarrow \infty$.

We consider Lagrangian polynomial interpolation on node set

$$\mathcal{T} := \{t_0, \dots, t_n\} \subset I, \quad I \subset \mathbb{R}, \text{ interval of length } |I|.$$

Notation: For a continuous function $f : I \mapsto \mathbb{K}$ we define the polynomial interpolation operator

$$l_{\mathcal{T}}(f) := l_{\mathcal{T}}(\mathbf{y}) \in \mathcal{P}_n \quad \text{with} \quad \mathbf{y} := (f(t_0), \dots, f(t_n))^T \in \mathbb{K}^{n+1}.$$

Goal: estimate of the **interpolation error** norm $\|f - l_{\mathcal{T}}f\|$ (for some norm on $C(I)$).

Focus: **asymptotic** behavior of interpolation error for $n \rightarrow \infty$

Example 5.3.2 (Asymptotic behavior of polynomial interpolation error).

Interpolation of $f(t) = \sin t$ on equispaced nodes in $I = [0, \pi]$: $\mathcal{T} = \{j\pi/n\}_{j=0}^n$.

Interpolating polynomial $p := l_{\mathcal{T}}f \in \mathcal{P}_n$.

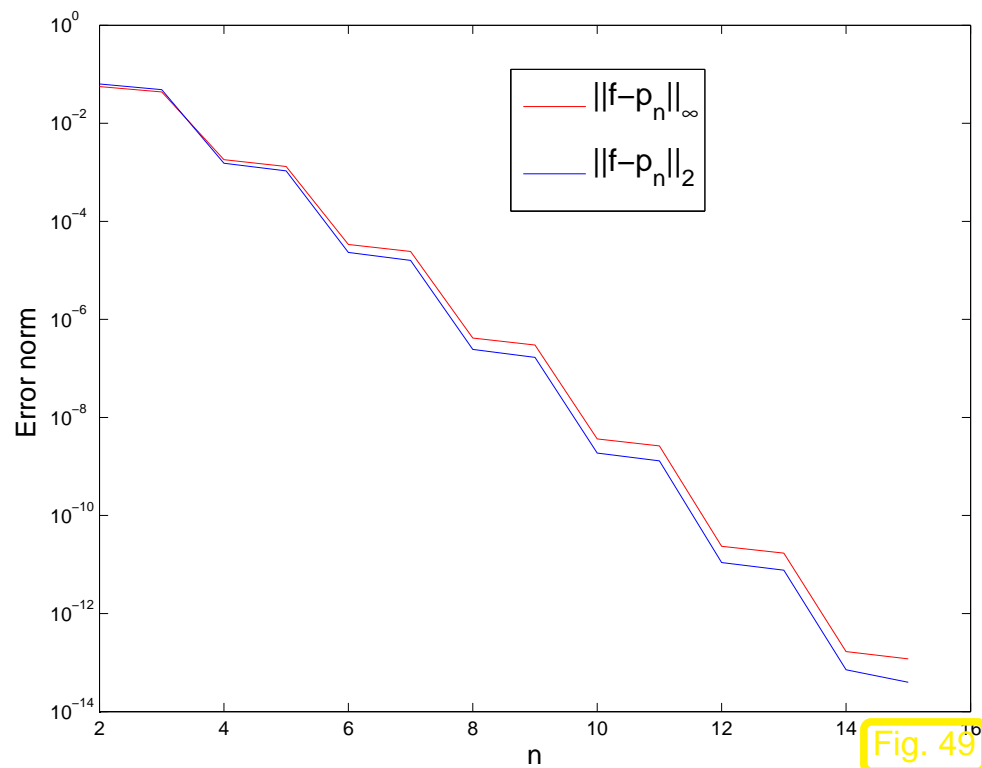


Fig. 49

computer-experiment: computation of the norms.

- L^∞ -norm: sampling on a grid of meshsize $\pi/1000$.
- L^2 -norm: numerical quadrature (\rightarrow Chapter 7) with trapezoidal rule (7.2.2) on a grid of mesh-size $\pi/1000$.



In the previous experiment we observed a clearly visible qualitative behavior of $\|f - I_{\mathcal{T}}f\|$ as we increased the polynomial degree n . The prediction of the decay law for $\|f - I_{\mathcal{T}}f\|$ 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.

Important terminology for the qualitative description of $\|f - \mathcal{I}_{\mathcal{T}} f\|$ as a function of the polynomial degree n :

$$\exists C \neq C(n) > 0: \quad \|f - \mathcal{I}_{\mathcal{T}} f\| \leq C T(n) \quad \text{for } n \rightarrow \infty. \quad (5.3.1)$$

Classification (best bound for $T(n)$):

$$\begin{aligned} \exists p > 0: \quad T(n) \leq n^{-p} & : \text{ algebraic convergence, with rate } p > 0, \quad \forall n \in \mathbb{N}. \\ \exists 0 < q < 1: \quad T(n) \leq q^n & : \text{ exponential convergence,} \end{aligned}$$

Convergence behavior of interpolation error is often expressed by means of the Landau-O-notation:

$$\begin{aligned} \text{Algebraic convergence:} \quad & \|f - \mathcal{I}_{\mathcal{T}} f\| = O(n^{-p}) \\ \text{Exponential convergence:} \quad & \|f - \mathcal{I}_{\mathcal{T}} f\| = O(q^n) \end{aligned} \quad \text{for } n \rightarrow \infty \text{ ("asymptotic!")}$$

Remark 5.3.3 (Exploring convergence).

Given: pairs (n_i, ϵ_i) , $i = 1, 2, 3, \dots$, $n_i \hat{=}$ polynomial degrees, $\epsilon_i \hat{=}$ norms of interpolation error

❶ Conjectured: **algebraic convergence**: $\epsilon_i \approx C n^{-p}$

$$\log(\epsilon_i) \approx \log(C) - p \log n_i \quad (\text{affine linear in log-log scale}).$$

Apply linear regression (`numpy.polyfit`) to points $(\log n_i, \log \epsilon_i)$ \triangleright estimate for rate p .

❶ 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. 3.0.1, `numpy.polyfit`) to points $(n_i, \log \epsilon_i)$ \triangleright estimate for $q := \exp(-\beta)$.

👉 Fig. 49: we suspect exponential convergence in Ex. 5.3.2.



Beware: same concept \leftrightarrow different meanings:

- **convergence** of a sequence (e.g. of iterates $x^{(k)} \rightarrow$ Sect. 1.1)
- **convergence** of an approximation (dependent on an approximation parameter, e.g. n)

Example 5.3.4 (Runge's example). \rightarrow Ex. ??

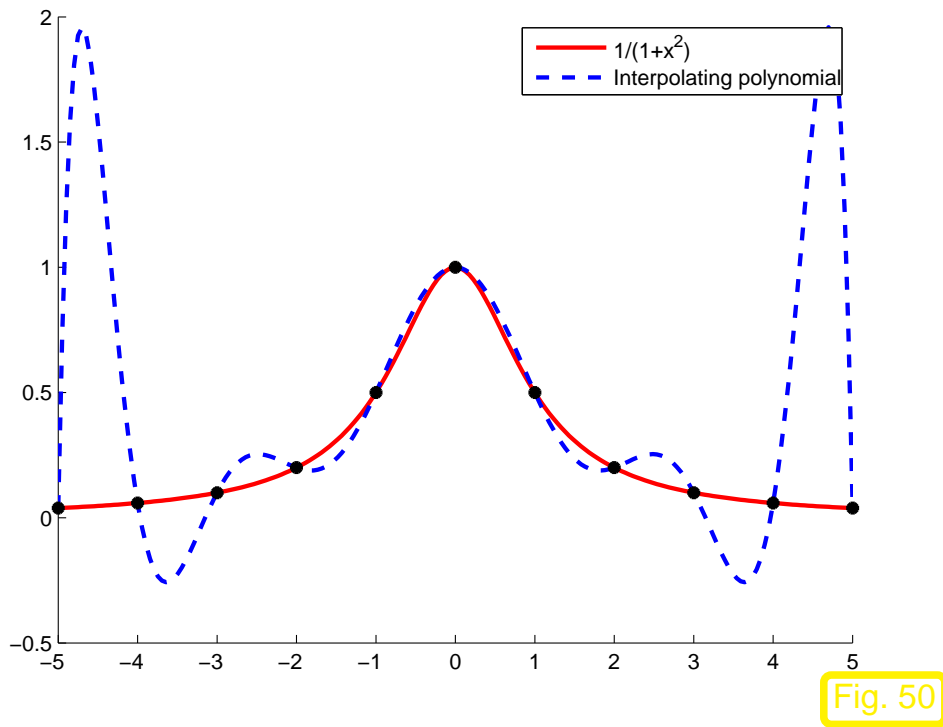
Polynomial interpolation of $f(t) = \frac{1}{1+t^2}$ with equispaced nodes:

$$\mathcal{T} := \left\{ t_j := -5 + \frac{10}{n} j \right\}_{j=0}^n, \quad y_j = \frac{1}{1+t_j^2}, j = 0, \dots, n.$$

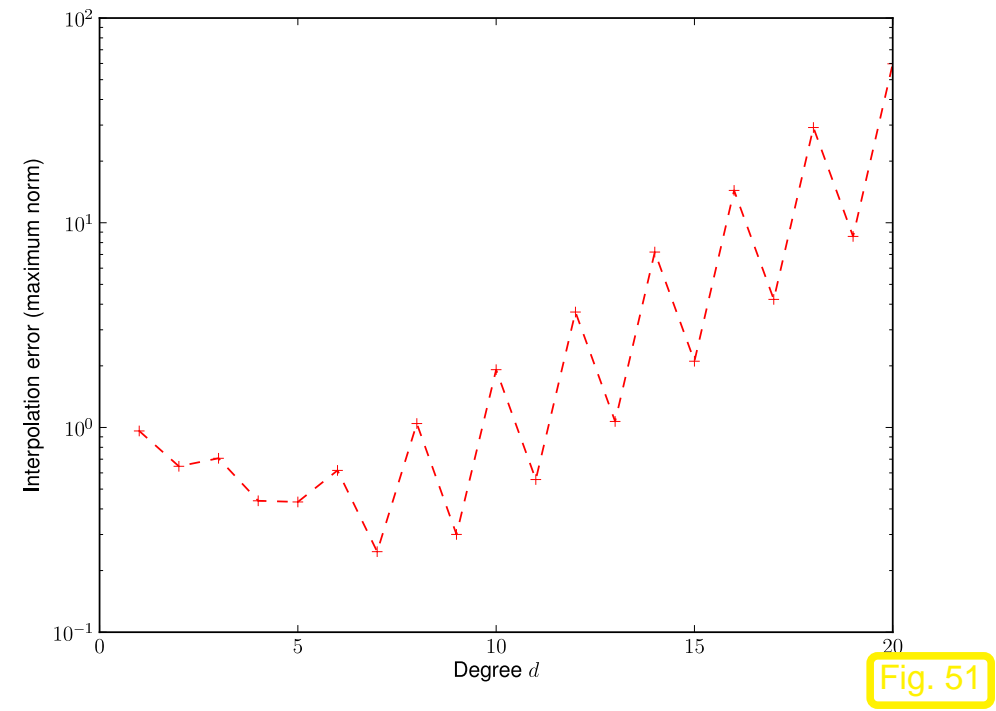
Code 5.3.5: Computing the interpolation error for Runge's example

```
1 # Interpolation error plot for Runge's example
2
3 from numpy import linspace, array, polyfit, polyval, max, abs, hstack,
   vstack
4 from matplotlib.pyplot import *
5
6 # Exact values
7 x = linspace(-5, 5, 1001)
8 f = lambda x: 1.0 / (1.0 + x**2)
9
10 fv = f(x)
11
12 # Compute approximation of increasing degree
13 err = []
14
15 for d in xrange(1, 21):
16     t = linspace(-5, 5, d+1)
```

```
7 p = polyfit(t, f(t), d)
8 y = polyval(p, x)
9
0 err.append( hstack([d, max(abs(y-fv))]) )
1
2 err = array(err)
3
4 # Plot
5 figure()
6 semilogy(err[:,0], err[:,1], "r—+")
7 xlabel(r "Degree_ $d$ ")
8 ylabel(r "Interpolation_error_(maximum_norm)")
9 savefig("../PICTURES/rungeerrmax.eps")
```



Interpolating polynomial, $n = 10$



Approximate $\|f - \mathcal{I}_{\mathcal{T}}f\|_{\infty}$ on $[-5, 5]$

Note: approximation of $\|f - \mathcal{I}_{\mathcal{T}}f\|_{\infty}$ by *sampling* in 1000 equidistant points.

Observation: Strong oscillations of $\mathcal{I}_{\mathcal{T}}f$ near the endpoints of the interval:

$$\|f - \mathcal{I}_{\mathcal{T}}f\|_{L^{\infty}([-5, 5])} \xrightarrow{n \rightarrow \infty} \infty.$$



Theorem 5.3.1 (Representation of interpolation error). [10, Thm. 8.22], [29, Thm. 37.4]

$f \in C^{n+1}(I): \forall t \in I: \exists \tau_t \in]\min\{t, t_0, \dots, t_n\}, \max\{t, t_0, \dots, t_n\}[$:

$$f(t) - l_{\mathcal{T}}(f)(t) = \frac{f^{(n+1)}(\tau_t)}{(n+1)!} \cdot \prod_{j=0}^n (t - t_j) . \quad (5.3.2)$$

Proof. Write $q(t) := \prod_{j=0}^n (t - t_j) \in \mathcal{P}_{n+1}$ and fix $t \in I$.

$$t \neq t_j \Rightarrow q(t) \neq 0 \Rightarrow \exists c(t) \in \mathbb{R}: f(t) - l_{\mathcal{T}}(f)(t) = cq(t)$$

$\varphi(x) := f(x) - l_{\mathcal{T}}(f)(x) - cq(x)$ has $n+2$ *distinct* zeros t_0, \dots, t_n, t . By iterated application of the **mean value theorem** [52, Thm .5.2.1], we conclude

$\varphi^{(m)}$ has $n+2-m$ distinct zeros in I .

$$\Rightarrow \exists \tau_t \in I: \varphi^{(n+1)}(x) = f^{(n+1)}(\tau_t) - c(n+1)! = 0 .$$

This fixes the value of $c = \frac{f^{(n+1)}(\tau_t)}{(n+1)!}$.

□

The theorem can also be proved using the following lemma.

Lemma 5.3.2 (Error of the polynomial interpolation). For $f \in C^{n+1}(I)$: $\forall t \in I$:

$$f(t) - l_{\mathcal{T}}(f)(t) = \int_0^1 \int_0^{\tau_1} \cdots \int_0^{\tau_{n-1}} \int_0^{\tau_n} f^{(n+1)}(t_0 + \tau_1(t_1 - t_0) + \cdots + \tau_n(t_n - t_{n-1}) + \tau(t - t_n)) d\tau d\tau_n \cdots d\tau_1 \cdot \prod_{j=0}^n (t - t_j) .$$

Proof. By induction on n , use (??) and the fundamental theorem of calculus [44, Sect. 3.1]:

Remark 5.3.6. Lemma 5.3.2 holds also for general polynomial interpolation with multiple nodes, see (??).

Gradinaru
D-MATH

In the *equation* (5.3.2) we can

- first bound the right hand side via $f^{(n+1)}(\tau_t) \leq \left\| f^{(n+1)} \right\|_{L^\infty(I)}$,
- 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 !),

- and, finally, take the maximum w.r.t. t on the left of \leq .

This yields the following **interpolation error estimate**:

$$\text{Thm. 5.3.1} \quad \Rightarrow \quad \|f - \mathcal{I}_T f\|_{L^\infty(I)} \leq \frac{\|f^{(n+1)}\|_{L^\infty(I)}}{(n+1)!} \max_{t \in I} |(t - t_0) \cdots (t - t_n)|. \quad (5.3.3)$$

Interpolation error estimate requires smoothness!

Example 5.3.7 (Error of polynomial interpolation). Ex. 5.3.2 cnt'd

Theoretical explanation for exponential convergence observed for polynomial interpolation of $f(t) = \sin(t)$ on equidistant nodes: by Thm. 5.3.2 and (5.3.3)

$$\begin{aligned} \|f^{(k)}\|_{L^\infty(I)} \leq 1, \quad \forall k \in \mathbb{N}_0 \quad \Rightarrow \quad \|f - p\|_{L^\infty(I)} &\leq \frac{1}{(1+n)!} \max_{t \in I} \left| (t - 0)(t - \frac{\pi}{n})(t - \frac{2\pi}{n}) \cdots (t - \pi) \right| \\ &\leq \frac{1}{n+1} \left(\frac{\pi}{n} \right)^{n+1}. \end{aligned}$$

→ **Uniform asymptotic 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 5.3.8 (Runge's example). Ex. 5.3.4 cnt'd

How can the blow-up of the interpolation error observed in Ex. 5.3.4 be reconciled with Thm. 5.3.2 ?

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. 5.3.1 $\rightarrow \infty$ for $n \rightarrow \infty$.

◇ Gradinaru
D-MATH

Remark 5.3.9 (L^2 -error estimates for polynomial interpolation).

Thm. 5.3.1 gives error estimates for the L^∞ -Norm. And the other norms?

From Lemma. 5.3.2 using Cauchy-Schwarz inequality:

$$\|f - \mathbf{l}_{\mathcal{T}}(f)\|_{L^2(I)}^2 = \int_I \left| \int_0^1 \int_0^{\tau_1} \cdots \int_0^{\tau_{n-1}} \int_0^{\tau_n} f^{(n+1)}(\dots) d\tau d\tau_n \cdots d\tau_1 \cdot \underbrace{\prod_{j=0}^n (t - t_j)}_{|t-t_j| \leq |I|} \right|^2 dt$$

$$\begin{aligned}
 &\leq \int_I |I|^{2n+2} \underbrace{\text{vol}_{(n+1)}(S_{n+1})}_{=1/(n+1)!} \int_{S_{n+1}} |f^{(n+1)}(\dots)|^2 d\boldsymbol{\tau} dt \\
 &= \int_I \frac{|I|^{2n+2}}{(n+1)!} \int_I \underbrace{\text{vol}_{(n)}(C_{t,\tau})}_{\leq 2^{(n-1)/2}/n!} |f^{(n+1)}(\tau)|^2 d\tau dt,
 \end{aligned}$$

$S_{n+1} := \{\mathbf{x} \in \mathbb{R}^{n+1} : 0 \leq x_n \leq x_{n-1} \leq \dots \leq x_1 \leq 1\}$ (unit simplex) ,

$C_{t,\tau} := \{\mathbf{x} \in S_{n+1} : t_0 + x_1(t_1 - t_0) + \dots + x_n(t_n - t_{n-1}) + x_{n+1}(t - t_n) = \tau\}$.

This gives the bound for the L^2 -norm of the error:

$$\Rightarrow \|f - \mathbf{l}_{\mathcal{T}}(f)\|_{L^2(I)} \leq \frac{2^{(n-1)/4} |I|^{n+1}}{\sqrt{(n+1)!n!}} \left(\int_I |f^{(n+1)}(\tau)|^2 d\tau \right)^{1/2}. \quad (5.3.4)$$

Notice: $f \mapsto \left\| f^{(n)} \right\|_{L^2(I)}$ defines a **seminorm** on $C^{n+1}(I)$
 (**Sobolev-seminorm**, measure of the smoothness of a function).

Estimates like (5.3.4) play a key role in the analysis of numerical methods for solving partial differential equations (\rightarrow course “Numerical methods for partial differential equations”).

5.4 Chebychev Interpolation

Perspective: function **approximation** by polynomial interpolation



Freedom to choose interpolation nodes judiciously

5.4.1 Motivation and definition

Mesh of nodes: $\mathcal{T} := \{t_0 < t_1 < \dots < t_{n-1} < t_n\}$, $n \in \mathbb{N}$,
function $f : I \rightarrow \mathbb{R}$ continuous; without loss of generality $I = [-1, 1]$.

Thm. 5.3.1:

$$\|f - p\|_{L^\infty(I)} \leq \frac{1}{(n+1)!} \|f^{(n+1)}\|_{L^\infty(I)} \|w\|_{L^\infty(I)} ,$$
$$w(t) := (t - t_0) \cdot \dots \cdot (t - t_n) .$$

Idea: choose nodes t_0, \dots, t_n such that $\|w\|_{L^\infty(I)}$ is minimal!

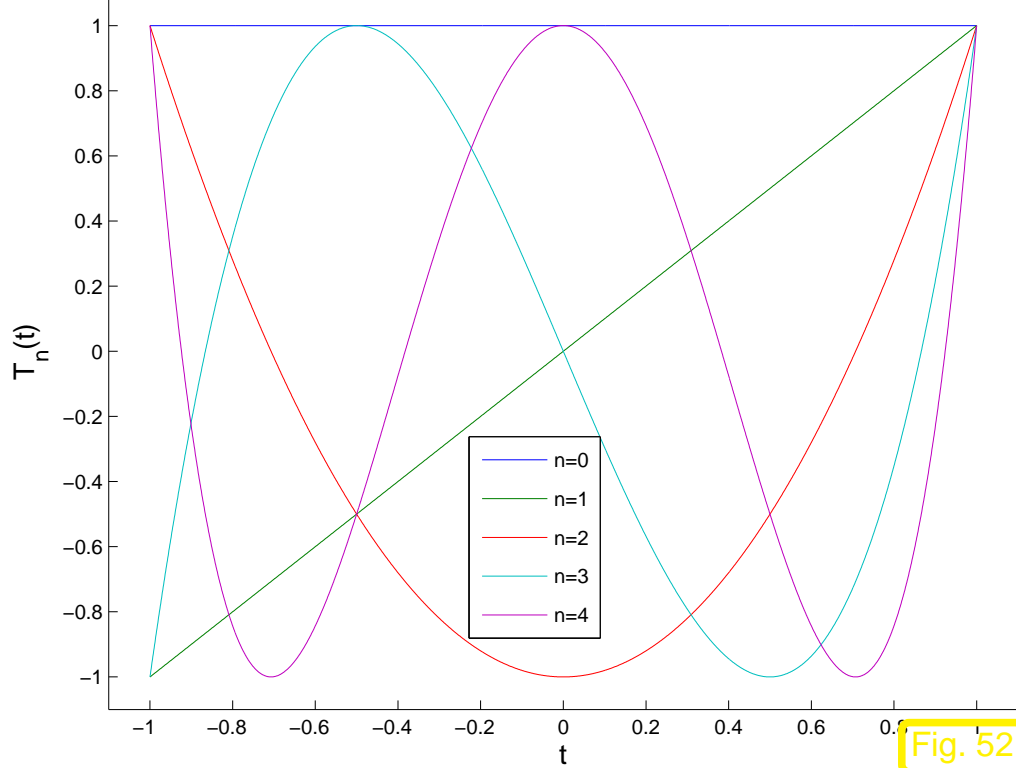
Equivalent to finding $q \in \mathcal{P}_{n+1}$, with leading coefficient = 1, such that $\|q\|_{L^\infty(I)}$ is minimal.

Choice of t_0, \dots, t_n = zeros of q (caution: t_j must belong to I).

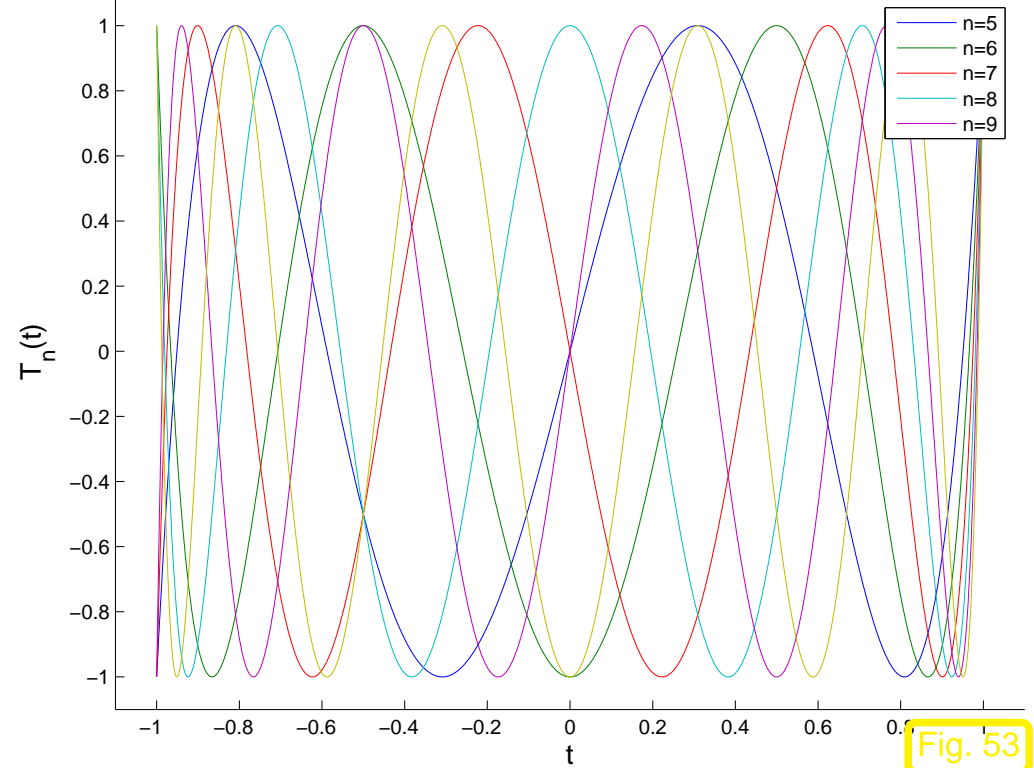
- Heuristic:
- t^* extremal point of $q \rightarrow |q(t^*)| = \|q\|_{L^\infty(I)}$,
 - q has $n + 1$ zeros in I ,
 - $|q(-1)| = |q(1)| = \|q\|_{L^\infty(I)}$.

Definition 5.4.1 (Chebychev polynomial).

The n^{th} *Chebychev polynomial* is $T_n(t) := \cos(n \arccos t)$, $-1 \leq t \leq 1$.



Chebyshev polynomials T_0, \dots, T_4



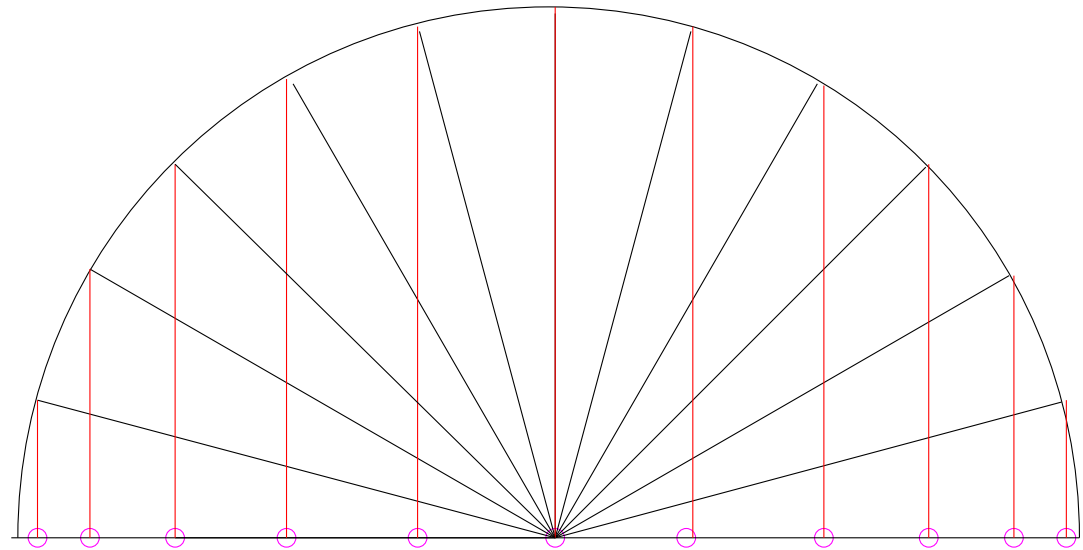
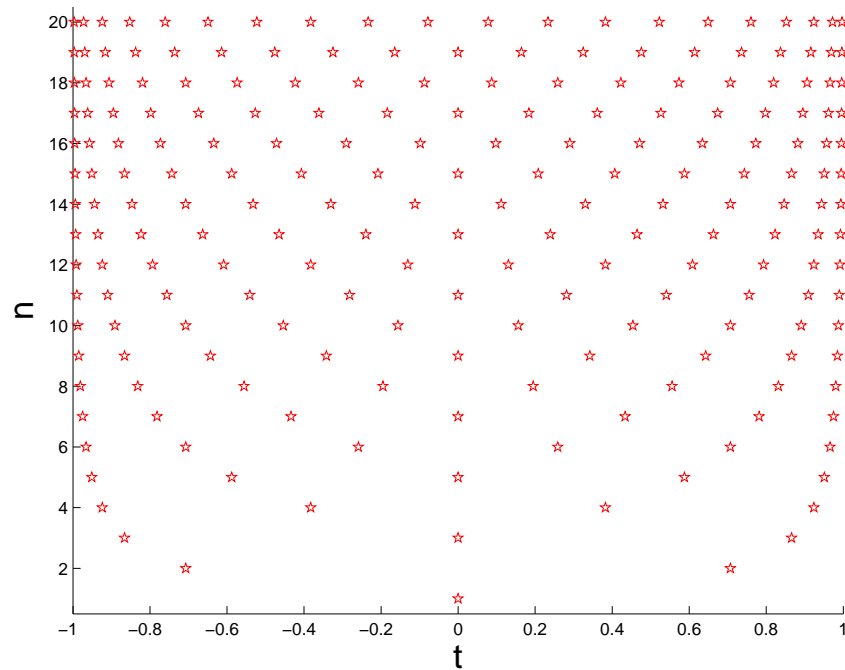
Chebyshev polynomials T_5, \dots, T_9

Zeros of T_n :
$$t_k = \cos \left(\frac{2k-1}{2n} \pi \right), \quad k = 1, \dots, n. \quad (5.4.1)$$

Extrema (alternating signs) of T_n :

$$|T_n(\bar{t}_k)| = 1 \Leftrightarrow \exists k = 0, \dots, n: \quad \bar{t}_k = \cos \frac{k\pi}{n}, \quad \|T_n\|_{L^\infty([-1,1])} = 1.$$

Chebyshev nodes t_k from (5.4.1):



Remark 5.4.1 (3-term recursion for Chebychev polynomial).

3-term recursion by $\cos(n+1)x = 2\cos nx \cos x - \cos(n-1)x$ with $\cos x = t$:

$$T_{n+1}(t) = 2tT_n(t) - T_{n-1}(t) \quad , \quad T_0 \equiv 1 \quad , \quad T_1(t) = t \quad , \quad n \in \mathbb{N} . \quad (5.4.2)$$

This implies:

- $T_n \in \mathcal{P}_n$,
- leading coefficients equal to 2^{n-1} ,
- T_n linearly independent,
- T_n basis of $\mathcal{P}_n = \text{Span} \{T_0, \dots, T_n\}$, $n \in \mathbb{N}_0$.

Theorem 5.4.2 (Minimax property of the Chebychev polynomials).

$$\|T_n\|_{L^\infty([-1,1])} = \inf\{\|p\|_{L^\infty([-1,1])} : p \in \mathcal{P}_n, p(t) = 2^{n-1}t^n + \dots\}, \quad \forall n \in \mathbb{N}.$$

Proof. See [14, Section 7.1.4.] (indirect) Assume

$$\exists q \in \mathcal{P}_n, \text{ leading coefficient } = 2^{n-1}: \quad \|q\|_{L^\infty([-1,1])} < \|T_n\|_{L^\infty([-1,1])}.$$

► $(T_n - q)(x) > 0$ 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 (??):

$T_n - q$ changes sign at least $n + 1$ times

$\Rightarrow T_n - q$ has at least n zeros

$T_n - q \equiv 0$, because $T_n - q \in \mathcal{P}_{n-1}$ (same leading coefficient!)

□

Application to approximation by polynomial interpolation:

- For $I = [-1, 1]$
- “optimal” interpolation nodes $\mathcal{T} = \left\{ \cos \left(\frac{2k+1}{2(n+1)} \pi \right) , k = 0, \dots, n \right\}$,
 - $w(t) = (t - t_0) \cdots (t - t_{n+1}) = 2^{-n} T_{n+1}(t)$, $\|w\|_{L^\infty(I)} = 2^{-n}$,
with leading coefficient 1.

Then, by Thm. 5.3.1,

$$\|f - \mathbf{l}_{\mathcal{T}}(f)\|_{L^\infty([-1,1])} \leq \frac{2^{-n}}{(n+1)!} \left\| f^{(n+1)} \right\|_{L^\infty([-1,1])} . \quad (5.4.3)$$

Remark 5.4.2 (Chebychev polynomials on arbitrary interval).

How to use Chebychev polynomial interpolation on an arbitrary interval?

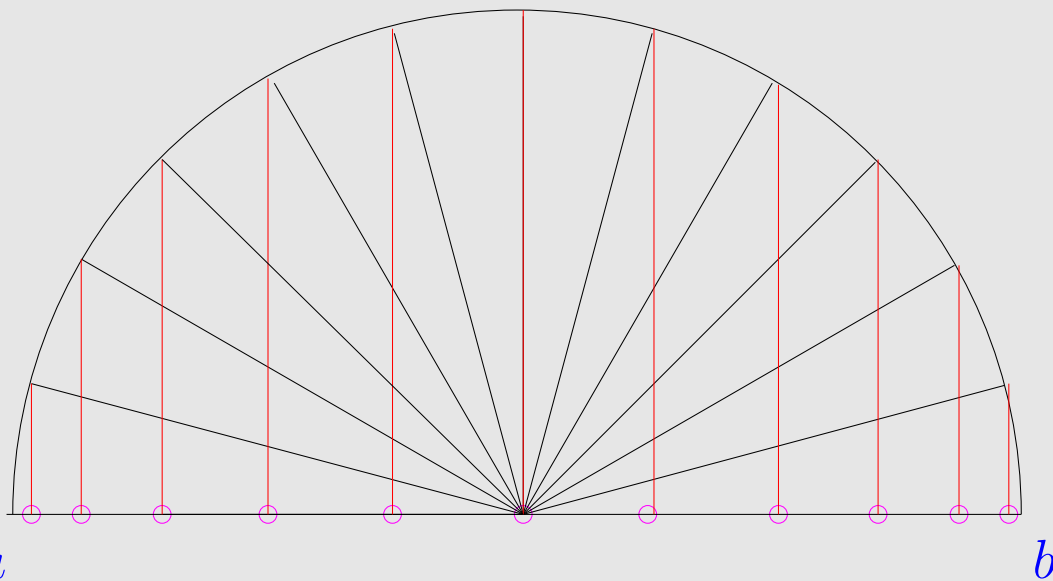
Scaling argument: interval transformation requires the transport of the functions

$$[-1, 1] \xrightarrow{\hat{t} \mapsto t := a + \frac{1}{2}(\hat{t} + 1)(b - a)} [a, b] \quad \Leftrightarrow \quad \hat{f}(\hat{t}) := f(t) .$$

$$p \in \mathcal{P}_n \quad \wedge \quad p(t_j) = f(t_j) \quad \Leftrightarrow \quad \hat{p} \in \mathcal{P}_n \quad \wedge \quad \hat{p}(\hat{t}_j) = \hat{f}(\hat{t}_j) .$$

$$\frac{d^n \hat{f}}{d\hat{t}^n}(\hat{t}) = \left(\frac{1}{2}|I|\right)^n \frac{d^n f}{dt^n}(t)$$

$$\begin{aligned} \|f - \mathbf{l}_{\mathcal{T}}(f)\|_{L^\infty(I)} &= \left\| \hat{f} - \mathbf{l}_{\hat{\mathcal{T}}}(\hat{f}) \right\|_{L^\infty([-1,1])} \leq \frac{2^{-n}}{(n+1)!} \left\| \frac{d^{n+1} \hat{f}}{d\hat{t}^{n+1}} \right\|_{L^\infty([-1,1])} \\ &\leq \frac{2^{-2n-1}}{(n+1)!} |I|^{n+1} \left\| f^{(n+1)} \right\|_{L^\infty(I)} . \end{aligned} \quad (5.4.4)$$



The **Chebyshev nodes** in the interval $I = [a, b]$ are

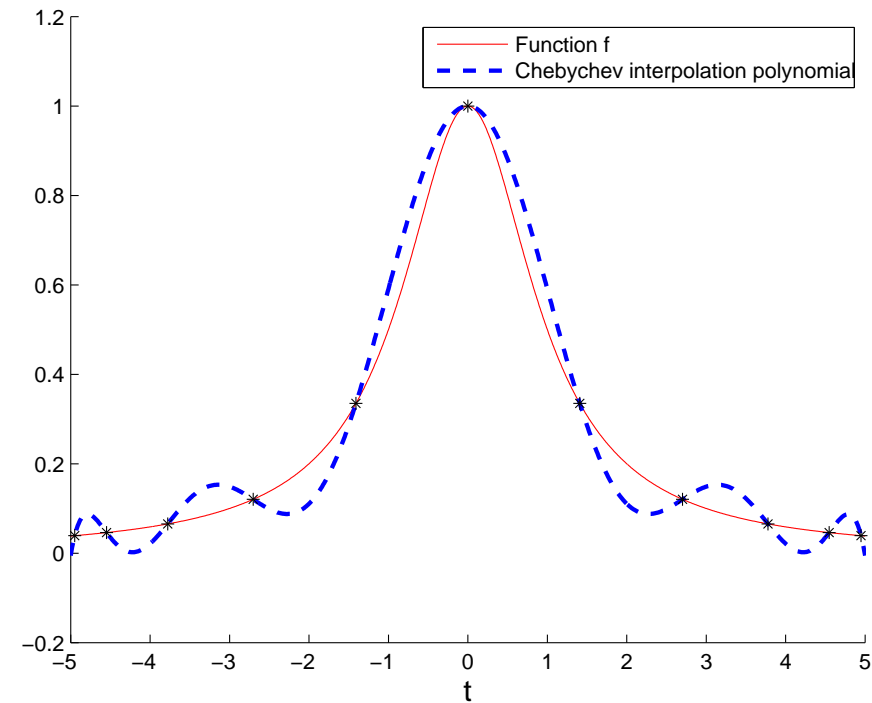
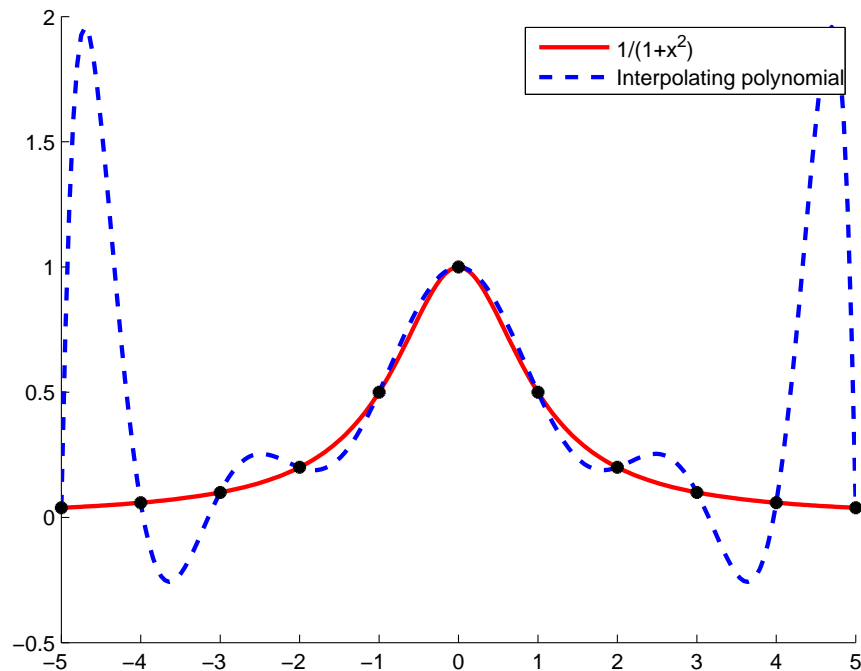
$$t_k := a + \frac{1}{2}(b - a) \left(\cos\left(\frac{2k+1}{2(n+1)}\pi\right) + 1 \right) , \quad (5.4.5)$$

$$k = 0, \dots, n .$$

5.4.2 Chebychev interpolation error estimates

Example 5.4.3 (Polynomial interpolation: Chebychev nodes versus equidistant nodes).

Runge's function $f(t) = \frac{1}{1+t^2}$, see Ex. 5.3.4, polynomial interpolation based on uniformly spaced nodes and Chebychev nodes:

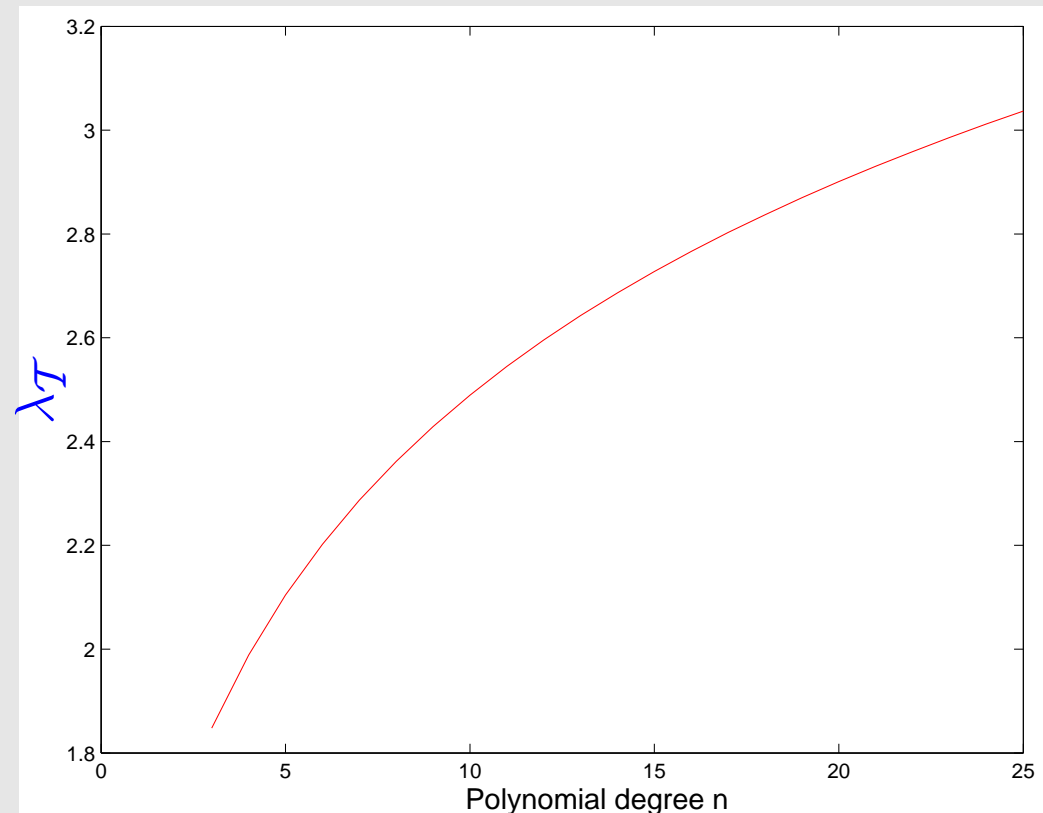


Remark 5.4.4 (Lebesgue Constant for Chebychev nodes).

Theory [7, 56, 55]:

$$\lambda_{\mathcal{T}} \sim \frac{2}{\pi} \log(1+n) + o(1) ,$$

$$\lambda_{\mathcal{T}} \leq \frac{2}{\pi} \log(1+n) + 1 . \quad (5.4.6)$$



Example 5.4.5 (Chebychev interpolation error).

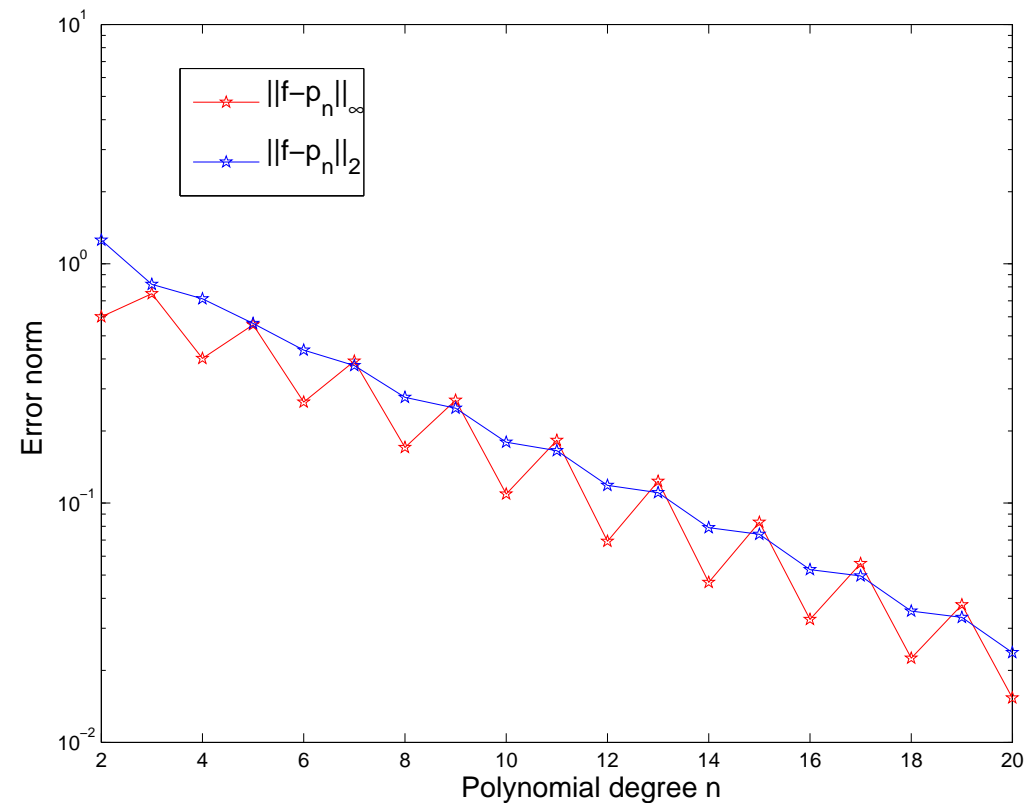
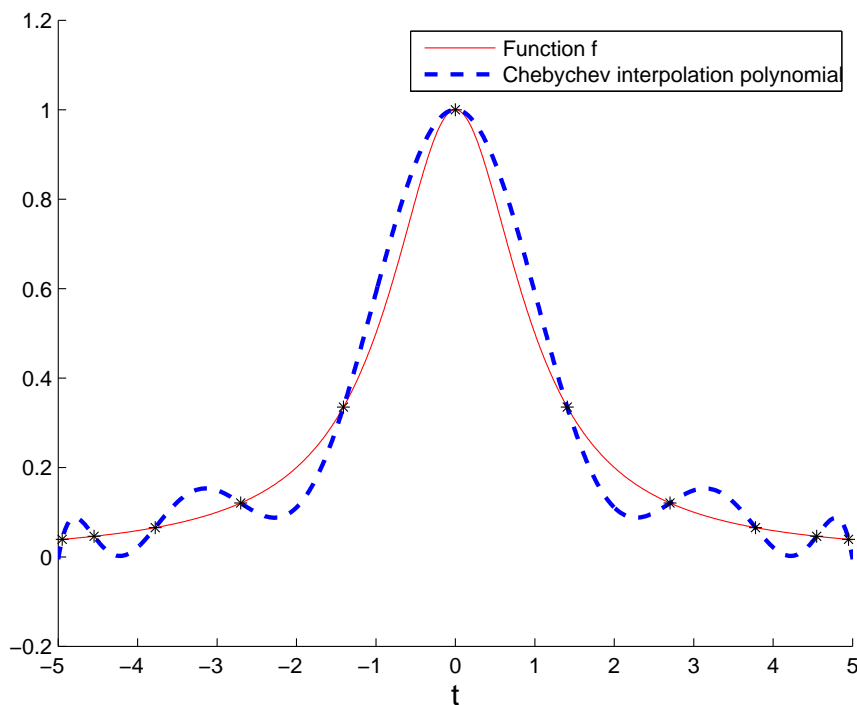
For $I = [a, b]$ let $x_l := a + \frac{b-a}{N}l$, $l = 0, \dots, N$, $N = 1000$ we approximate the norms of the error

$$\|f - p\|_{\infty} \approx \max_{0 \leq l \leq N} |f(x_l) - p(x_l)|$$

$$\|f - p\|_2^2 \approx \frac{b-a}{2N} \sum_{0 \leq l < N} \left(|f(x_l) - p(x_l)|^2 + |f(x_{l+1}) - p(x_{l+1})|^2 \right)$$

① $f(t) = (1 + t^2)^{-1}$, $I = [-5, 5]$ (see Ex. 5.3.4)

Interpolation with $n = 10$ Chebychev nodes (plot on the left).



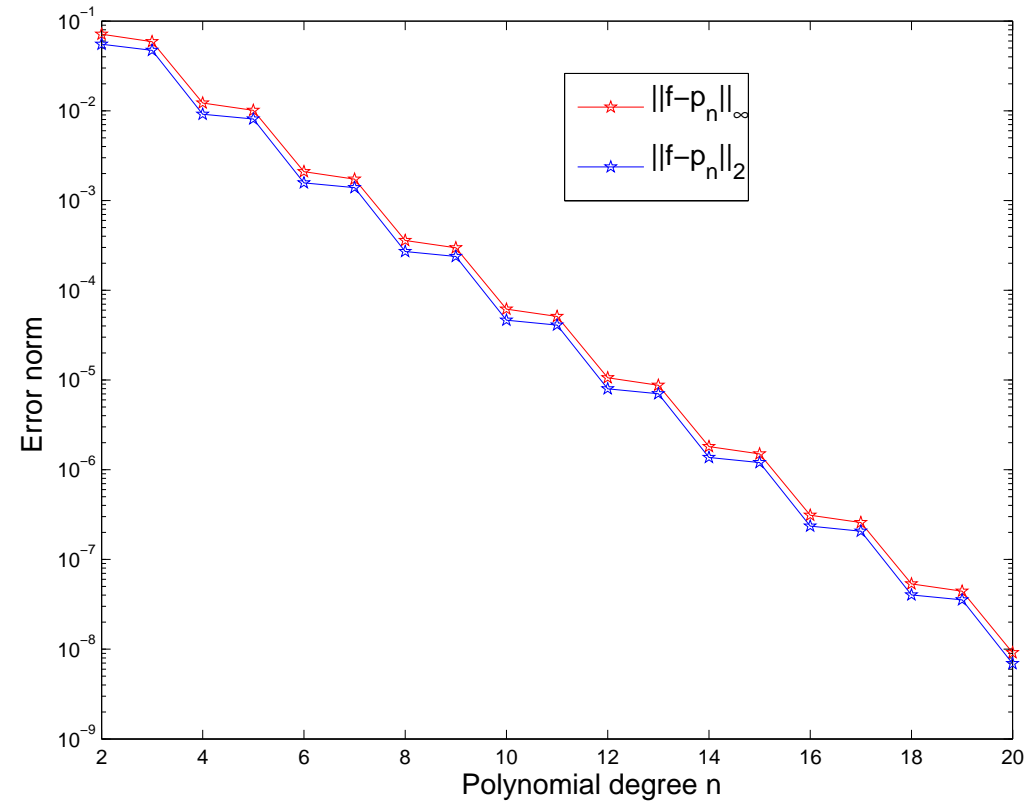
Notice: exponential convergence of the Chebychev interpolation:

$$p_n \rightarrow f, \quad \|f - \mathbf{l}_n f\|_{L^2([-5,5])} \approx 0.8^n$$

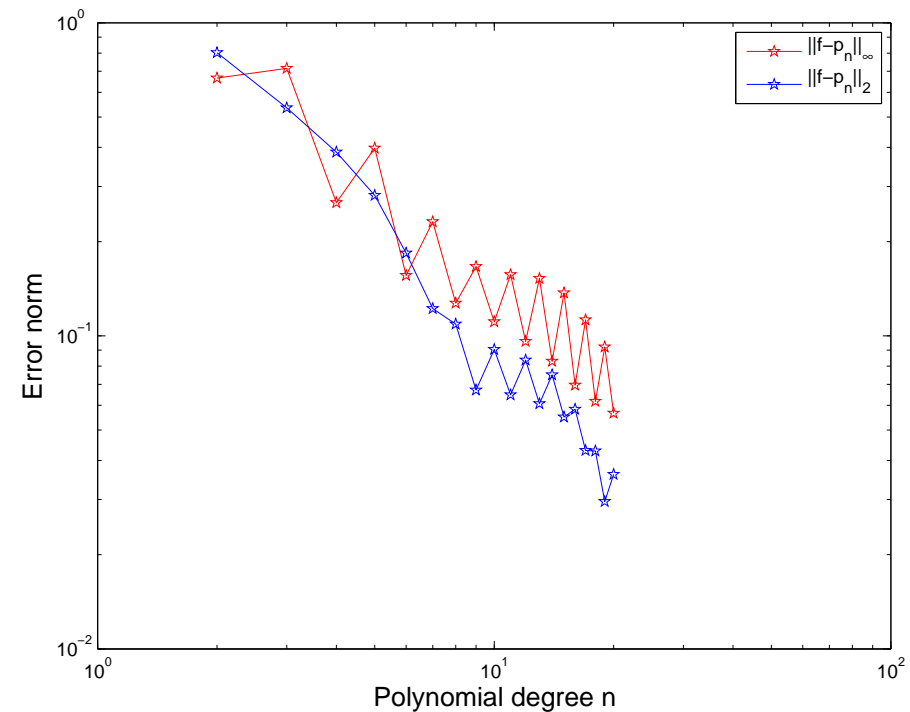
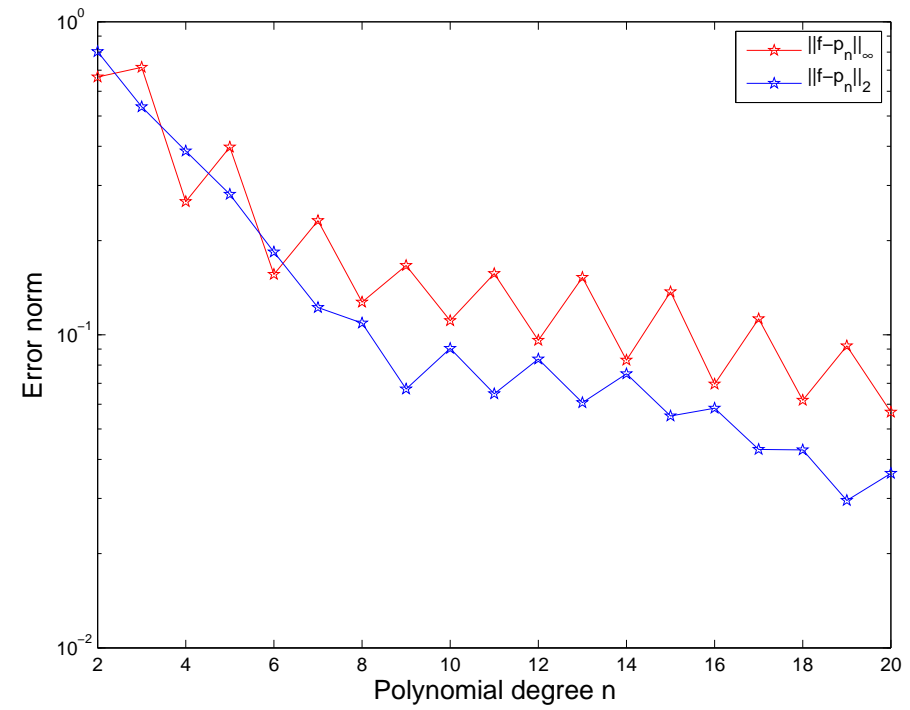
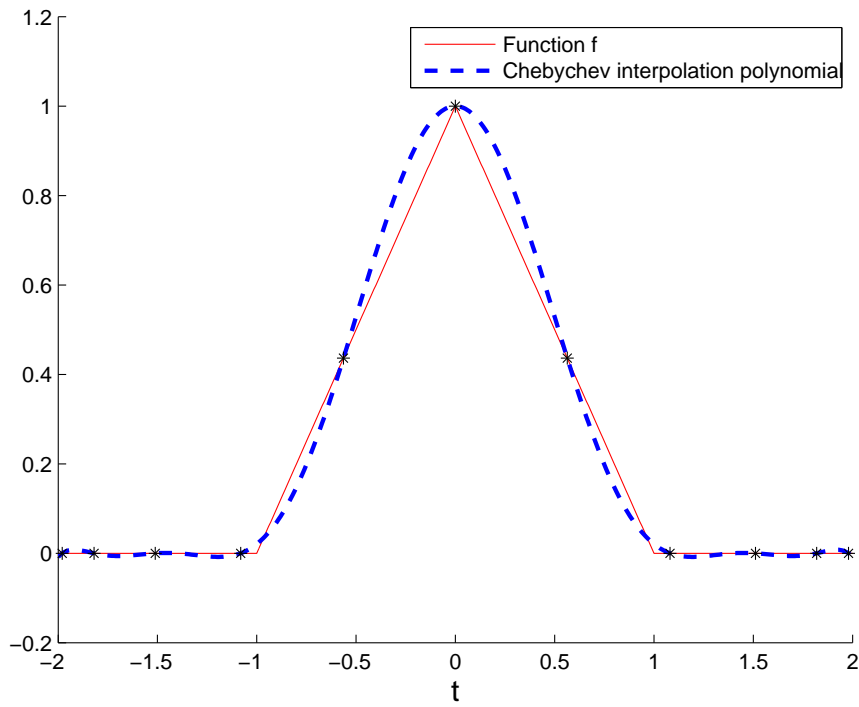
Now: the same function $f(t) = (1 + t^2)^{-1}$
on a smaller interval $I = [-1, 1]$.

(Faster) exponential convergence:

$$\|f - \mathbf{l}_n f\|_{L^2([-1,1])} \approx 0.42^n.$$



② $f(t) = \max\{1 - |t|, 0\}$, $I = [-2, 2]$, $n = 10$ nodes (plot on the left).
 $f \in C^0(I)$ but $f \notin C^1(I)$.

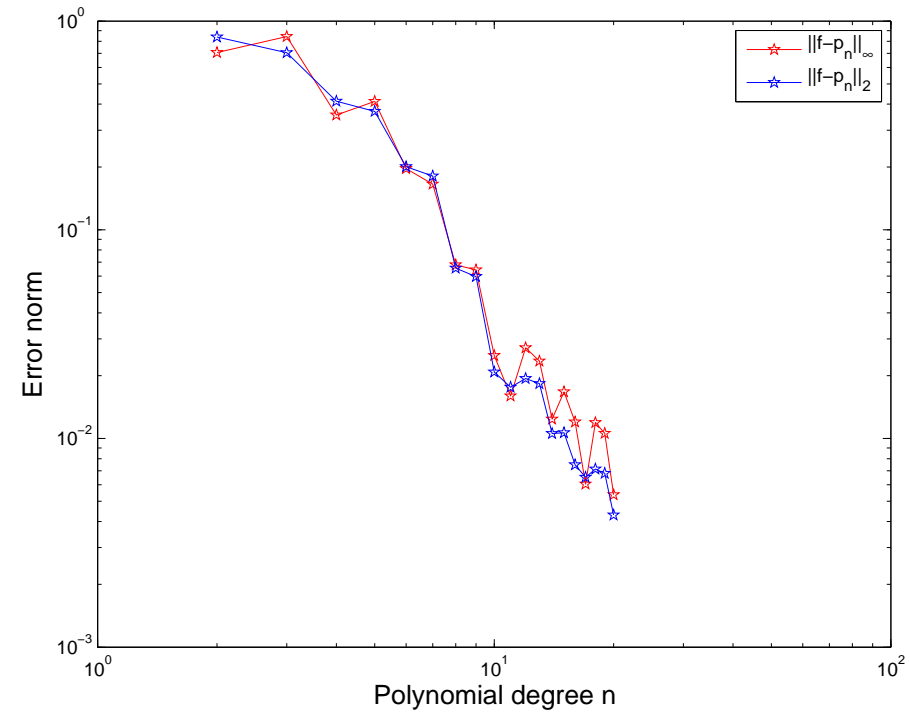
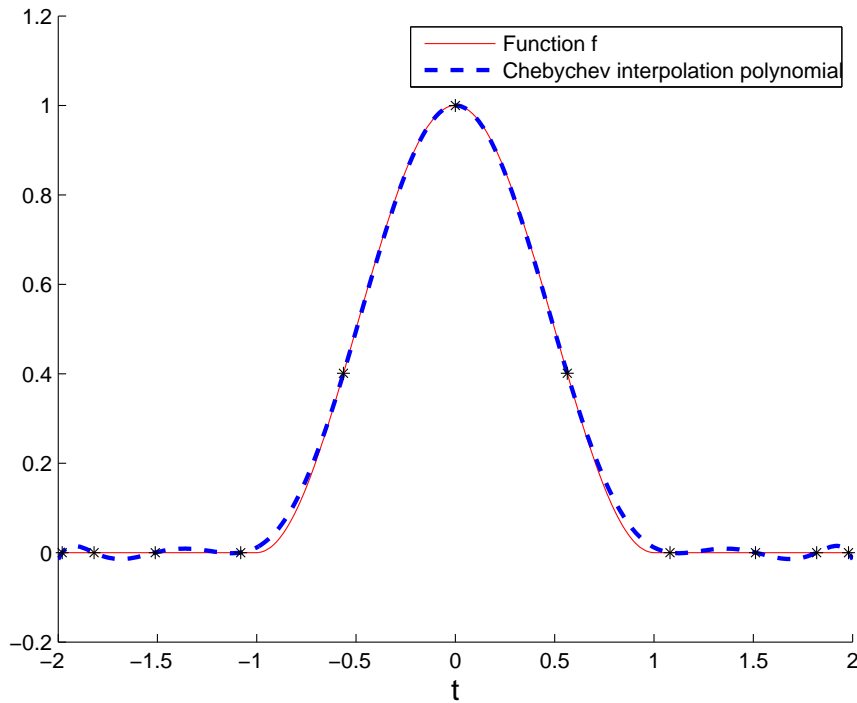


From the double logarithmic plot, notice

- no exponential convergence
- algebraic convergence (?)



③ $f(t) = \begin{cases} \frac{1}{2}(1 + \cos \pi t) & |t| < 1 \\ 0 & 1 \leq |t| \leq 2 \end{cases} \quad I = [-2, 2], \quad n = 10 \quad (\text{plot on the left}).$



Notice: only algebraic convergence.



Summary of observations, cf. Rem. 5.3.3:

- Essential role of smoothness of f : slow convergence of approximation error of the Cheychev interpolant if f enjoys little smoothness, cf. also (5.3.3),
- for smooth $f \in C^\infty$ approximation error of the Cheychev interpolant seems to decay to zero **exponentially** in the polynomial degree n .

5.4.3 Chebychev interpolation: computational aspects

Theorem 5.4.3 (Orthogonality of Chebychev polynomials).

The Chebychev polynomials are orthogonal with respect to the scalar product

$$\langle f, g \rangle = \int_{-1}^1 f(x)g(x) \frac{1}{\sqrt{1-x^2}} dx . \quad (5.4.7)$$

Theorem 5.4.4 (Discrete orthogonality of Chebychev polynomials).

The Chebychev polynomials T_0, \dots, T_n are orthogonal in the space \mathcal{P}_n with respect to the scalar product:

$$(f, g) = \sum_{k=0}^n f(x_k)g(x_k) , \quad (5.4.8)$$

where x_0, \dots, x_n are the zeros of T_{n+1} .

① Computation of the coefficients of the interpolation polynomial in Chebychev form:

Theorem 5.4.5 (Representation formula).

The interpolation polynomial p of f in the Chebychev nodes x_0, \dots, x_n (the zeros of T_{n+1}) is given by:

$$p(x) = \frac{1}{2}c_0 + c_1T_1(x) + \dots + c_nT_n(x) , \quad (5.4.9)$$

with

$$c_k = \frac{2}{n+1} \sum_{l=0}^n f \left(\cos \left(\frac{2l+1}{n+1} \cdot \frac{\pi}{2} \right) \right) \cdot \cos \left(k \frac{2l+1}{n+1} \cdot \frac{\pi}{2} \right) . \quad (5.4.10)$$

For sufficiently large n ($n \geq 15$) it is convenient to compute the c_k with the FFT; the direct computation of the coefficients needs $(n+1)^2$ multiplications, while FFT needs only $O(n \log n)$.

② Evaluation of polynomials in Chebychev form:

Theorem 5.4.6 (Clenshaw algorithm).

Let $p \in \mathcal{P}_n$ be an arbitrary polynomial,

$$p(x) = \frac{1}{2}c_0 + c_1T_1(x) + \dots + c_nT_n(x) .$$

Set

$$d_{n+2} = d_{n+1} = 0$$

$$d_k = c_k + (2x) \cdot d_{k+1} - d_{k+2} \quad \text{for } k = n, n-1, \dots, 0. \quad (5.4.11)$$

Then $p(x) = \frac{1}{2}(d_0 - d_2)$.

```

1 def clenshaw(a,x):
2     # Degree of polynomial
3     n = a.shape[0] - 1
4     d = tile( reshape(a,n+1,1), (x.shape[0], 1) )
5     d = d.T
6     for j in xrange(n, 1, -1):
7         d[j-1,:] = d[j-1,:] + 2.0*x*d[j,:]
8         d[j-2,:] = d[j-2,:] - d[j,:]
9     y = d[0,:] + x*d[1,:]
10    return y

```

While using recursion it is important how the error (e.g. rounding error) propagates.

Simple example:

$$\begin{aligned}
 x_{n+1} &= 10x_n - 9, \\
 x_0 = 1 &\Rightarrow x_n = 1 \quad \forall n \\
 x_0 = 1 + \epsilon &\Rightarrow \tilde{x}_n = 1 + 10^n \epsilon.
 \end{aligned}$$

This is not a problem here: Clenshaw algorithm is stable.

Theorem 5.4.7 (Stability of Clenshaw algorithm).

Consider the perturbed Clenshaw algorithm recursion:

$$\tilde{d}_k = c_k + 2x \cdot \tilde{d}_{k+1} - \tilde{d}_{k+2} + \epsilon_k, \quad k = n, n-1, \dots, 0$$

with $\tilde{d}_{n+2} = \tilde{d}_{n+1} = 0$. Set $\tilde{p}(x) = \frac{1}{2}(\tilde{d}_0 - \tilde{d}_2)$. Then $|\tilde{p}(x) - p(x)| \leq \sum_{j=0}^n |\epsilon_j|$ for $|x| \leq 1$.

Remark 5.4.7 (Chebychev representation of built-in functions).

Computers use approximation by sums of Chebychev polynomials in the computation of functions like $\log, \exp, \sin, \cos, \dots$. The evaluation through Clenshaw algorithm is much more efficient than with Taylor approximation.



5.5 Essential Skills Learned in Chapter 5

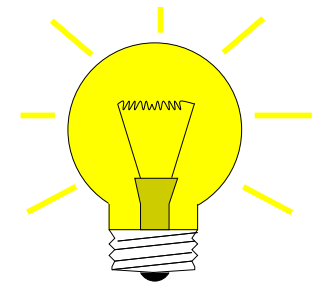
You should know:

- what are the divided differences and their use for polynomial interpolation
- what algebraic/exponential convergence means
- error behavior for polynomial interpolation on equally spaced points
- reasons for using rather Chebyshev interpolation
- definition and properties of the Chebyshev polynomials
- error behavior for Chebyshev interpolation
- how to compute with Chebyshev polynomials

6

Trigonometric Interpolation

Is there something else than polynomials and Taylor approximation in the world?



Idea (J. Fourier 1822): Approximation of a function not by usual polynomials but by trigonometrical polynomials = partial sum of a Fourier series

Gradinaru
D-MATH

We call **trigonometrical polynomial** of degree $\leq 2m$ the function

$$T_{2m}(t) := t \mapsto \sum_{j=-m}^m \gamma_j e^{2\pi i j t}, \quad \gamma_j \in \mathbb{C}, t \in \mathbb{R}.$$

Remark 6.0.1. $T_{2m} : \mathbb{R} \rightarrow \mathbb{C}$ is periodic of period 1. Moreover, if $\gamma_{-j} = \overline{\gamma_j}$ for all $j = 0, \dots, 2m$, then $T_{2m}(t)$ takes only **real** values and may be written as

$$T_{2m}(t) = \frac{a_0}{2} + \sum_{j=1}^{2m} (a_j \cos(2\pi j t) + b_j \sin(2\pi j t))$$

with $a_0 = 2\gamma_0$ and $a_j = 2 \operatorname{Re} \gamma_j$, $b_j = -2 \operatorname{Im} \gamma_j$ for all $j = 1, \dots, 2m$.

Remark 6.0.2.

The functions $w_j(t) = e^{2\pi i j t}$ are orthogonal with respect to the $L^2([0, 1])$ scalar product.

Let us take as granted (or known from lectures in Analysis, Mathematical Methods of Physics, etc.):

Theorem 6.0.1 (L^2 -convergence of the Fourier series). *Every squared integrable function $f \in L^2([0, 1]) := \{f : [0, 1[\mapsto \mathbb{C} : \|f\|_{L^2([0, 1])} < \infty\}$ is the $L^2([0, 1])$ -limit of its **Fourier series***

$$f(t) = \sum_{k=-\infty}^{\infty} \widehat{f}(k) e^{2\pi i k t} \quad \text{in } L^2([0, 1]) ,$$

with **Fourier coefficients** defined by

$$\widehat{f}(k) = \int_0^1 f(t) e^{-2\pi i k t} dt , \quad k \in \mathbb{Z} .$$

Remark 6.0.3. In view of this theorem, we may think of *one* function in *two* ways: once in the *time* (or *space*) domain $t \rightarrow f(t)$ and once in the *frequency* domain $k \rightarrow \widehat{f}(k)$.

Isometry property (Parseval):
$$\sum_{k=-\infty}^{\infty} |\widehat{f}(k)|^2 = \|f\|_{L^2([0,1])}^2 \quad (6.0.1)$$

Remark 6.0.4. A real function f may be equally represented as

$$\frac{a_0}{2} + \sum_{j=1}^{\infty} (a_j \cos(2\pi j t) + b_j \sin(2\pi j t)) \quad \text{in } L^2([0, 1]) ,$$

with

$$a_j = 2 \int_0^1 f(t) \cos(2\pi j t) dt , \quad j \geq 0 ,$$

$$b_j = 2 \int_0^1 f(t) \sin(2\pi j t) dt , \quad j \geq 1 .$$

Lemma 6.0.2 (Derivative and Fourier coefficients).

$$f \in L^1([0, 1]) \ \& \ f' \in L^1([0, 1]) \quad \Rightarrow \quad \widehat{f'}(k) = 2\pi i k \widehat{f}(k), \quad k \in \mathbb{Z}.$$

Remark 6.0.5.

$$\left\| f^{(n)} \right\|_{L^2([0,1])}^2 = (2\pi)^{2n} \sum_{k=-\infty}^{\infty} k^{2n} |\widehat{f}(k)|^2 . \quad (6.0.2)$$

$$\text{If } |\widehat{f^{(n)}}(k)| \leq \int_0^1 |f^{(n)}(t)| dt < \infty \Rightarrow \widehat{f}(k) = O(k^{-n}) \text{ for } |k| \rightarrow \infty .$$

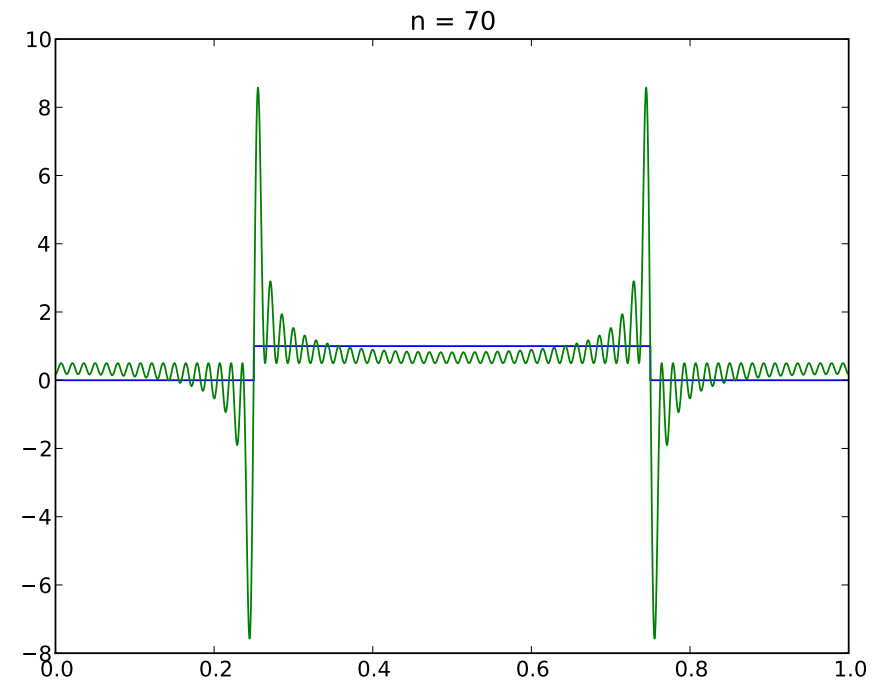
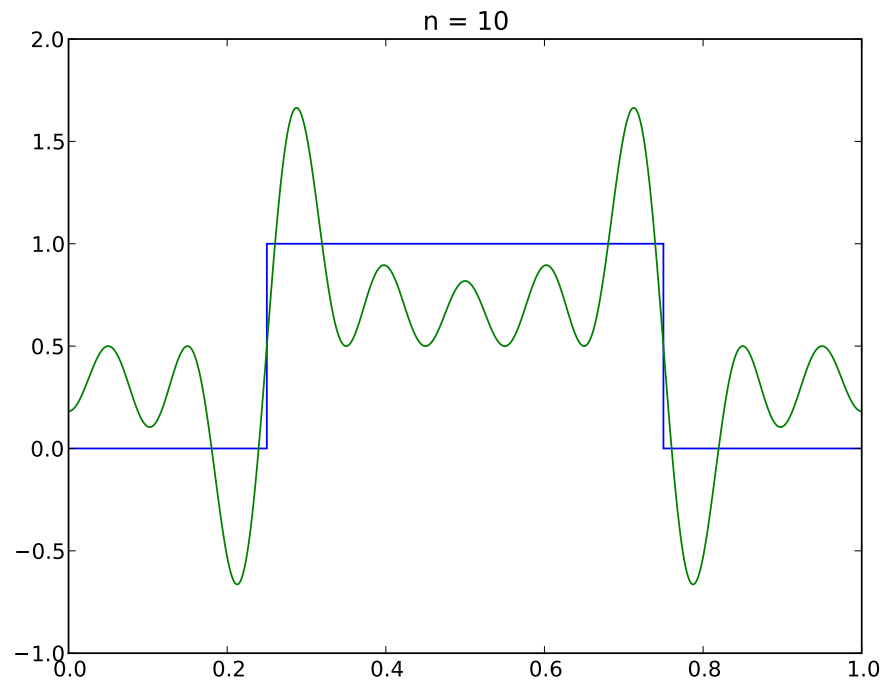
The smoothness of a function directly reflects in the quick decay of its Fourier coefficients.

Example 6.0.6. The Fourier series associated with the characteristic function of an interval $[a, b] \subset]0, 1[$ may be computed analytically as

$$b - a + \frac{1}{\pi} \sum_{|k|=1}^{\infty} e^{-ikc} \frac{\sin kd}{k} e^{i2\pi kt}, \quad t \in [0, 1],$$

with $c = \pi(a + b)$ and $d = \pi(b - a)$.

Note the slow decay of the Fourier coefficients, and hence expect slow convergence of the series. Moreover, observe in the pictures below the Gibbs phenomenon: the ripples move closer to the discontinuities and increase with larger n . Explanation: we have L^2 -convergence but no uniform convergence of the series!



Remark 6.0.7. Usually one cannot compute analytically $\hat{f}(k)$ or one has to rely only on discrete values at nodes $x_\ell = \frac{\ell}{N}$ for $\ell = 0, 1, \dots, N$; the trapezoidal rule gives

$$\hat{f}(k) \approx \frac{1}{N} \sum_{\ell=0}^{N-1} f(x_\ell) e^{-2\pi i k x_\ell} =: \hat{f}_N(k) \quad (6.0.3)$$

6.1 Discrete Fourier Transform (DFT)

Let $n \in \mathbb{N}$ fixed and denote the n th **root of unity** $\omega_n := \exp(-2\pi i/n) = \cos(2\pi/n) - i \sin(2\pi/n)$

$$\text{hence } \omega_n^k = \omega_n^{k+n} \quad \forall k \in \mathbb{Z} \quad , \quad \omega_n^n = 1 \quad , \quad \omega_n^{n/2} = -1 \quad , \quad (6.1.1)$$

$$\sum_{k=0}^{n-1} \omega_n^{kj} = \begin{cases} n & , \text{ if } j = 0 \quad , \\ 0 & , \text{ if } j \neq 0 \quad . \end{cases} \quad (6.1.2)$$

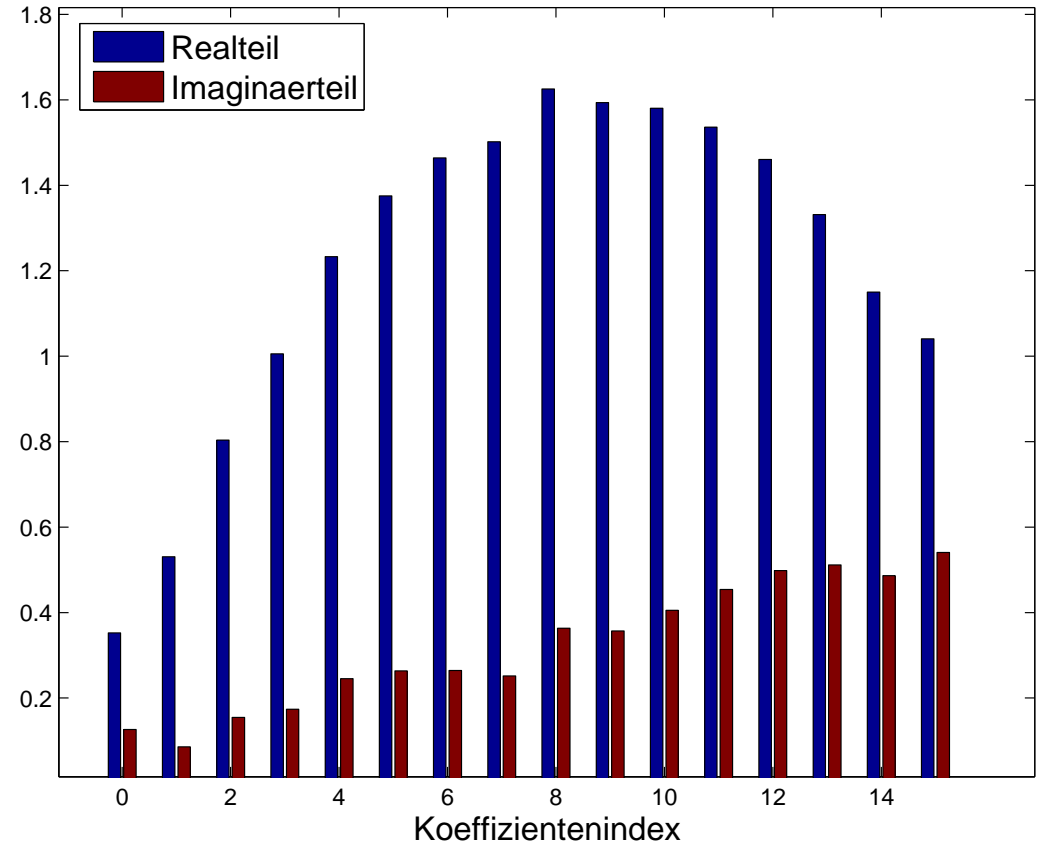
A change of basis in \mathbb{C}^n :

standard basis of \mathbb{K}^n

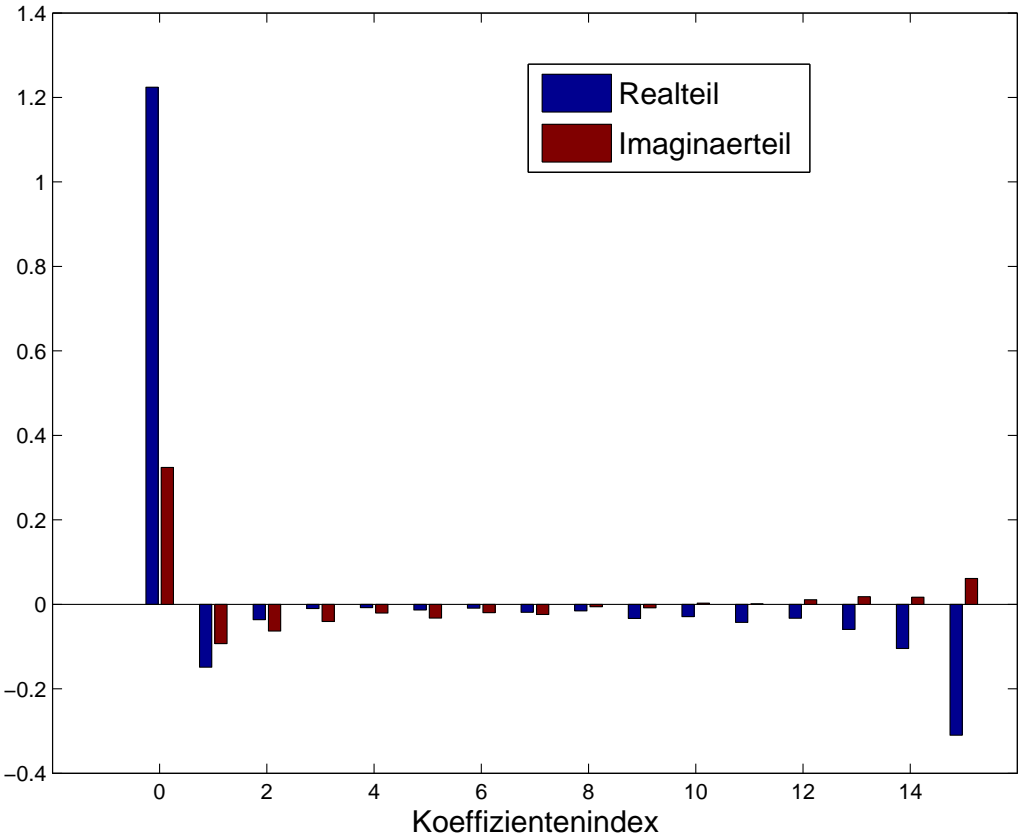
Trigonometrical Basis

$$\left\{ \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \dots \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{pmatrix} \right\} \leftarrow \left\{ \begin{pmatrix} \omega_n^0 \\ \vdots \\ \omega_n^0 \end{pmatrix} \begin{pmatrix} \omega_n^0 \\ \omega_n^1 \\ \vdots \\ \omega_n^{n-1} \end{pmatrix} \dots \begin{pmatrix} \omega_n^0 \\ \omega_n^{n-2} \\ \omega_n^{2(n-2)} \\ \vdots \\ \omega_n^{(n-1)(n-2)} \end{pmatrix} \begin{pmatrix} \omega_n^0 \\ \omega_n^{n-1} \\ \omega_n^{2(n-1)} \\ \vdots \\ \omega_n^{(n-1)^2} \end{pmatrix} \right\}$$

$$\mathbf{F}_n = \begin{pmatrix} \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \cdots & \omega_n^{2n-2} \\ \vdots & \vdots & & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)^2} \end{pmatrix} = \left(\omega_n^{ij} \right)_{i,j=0}^{n-1} \in \mathbb{C}^{n,n} . \tag{6.1.3}$$



Vector in standardbasis



Vector in trigonometrical basis

Definition 6.1.1 (Diskrete Fourier transform). We call linear map $\mathcal{F}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$, $\mathcal{F}_n(\mathbf{y}) := \mathbf{F}_n \mathbf{y}$, $\mathbf{y} \in \mathbb{C}^n$, *discrete Fourier transform* (DFT), i.e. for $\mathbf{c} := \mathcal{F}_n(\mathbf{y})$

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj} \quad , \quad k = 0, \dots, n-1 . \quad (6.1.4)$$

Convention: in the discussion of the DFT: vektor indexes run from 0 to $n-1$.

Lemma 6.1.2.

The scaled Fourier-matrix $\frac{1}{\sqrt{n}} \mathbf{F}_n$ is unitary: $\mathbf{F}_n^{-1} = \frac{1}{n} \mathbf{F}_n^H = \frac{1}{n} \overline{\mathbf{F}_n}$

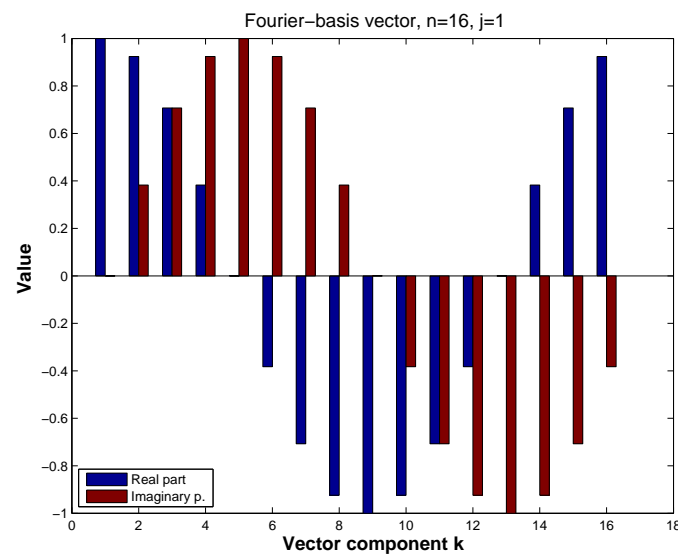
Remark 6.1.1. $\frac{1}{n} \mathbf{F}_n^2 = -I$ and $\frac{1}{n^2} \mathbf{F}_n^4 = I$, hence the eigenvalues of \mathbf{F}_n are in the set $\{1, -1, i, -i\}$.

numpy-functions:

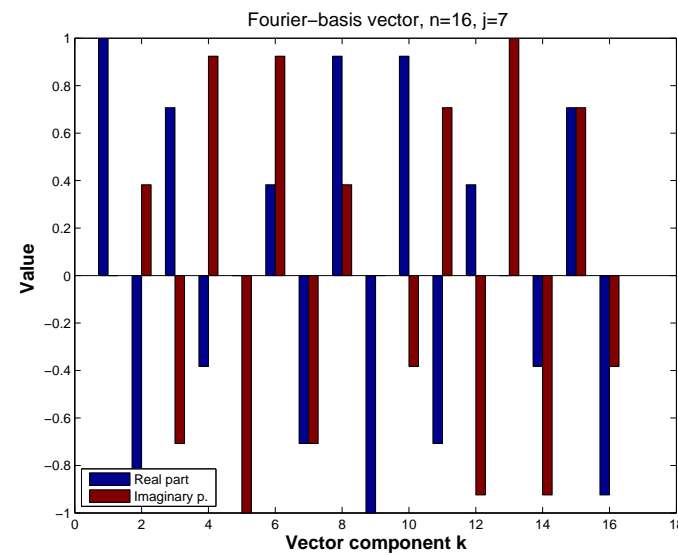
$$\mathbf{c} = \text{fft}(\mathbf{y}) \leftrightarrow \mathbf{y} = \text{ifft}(\mathbf{c}) ;$$

Example 6.1.2 (Frequency analysis with DFT).

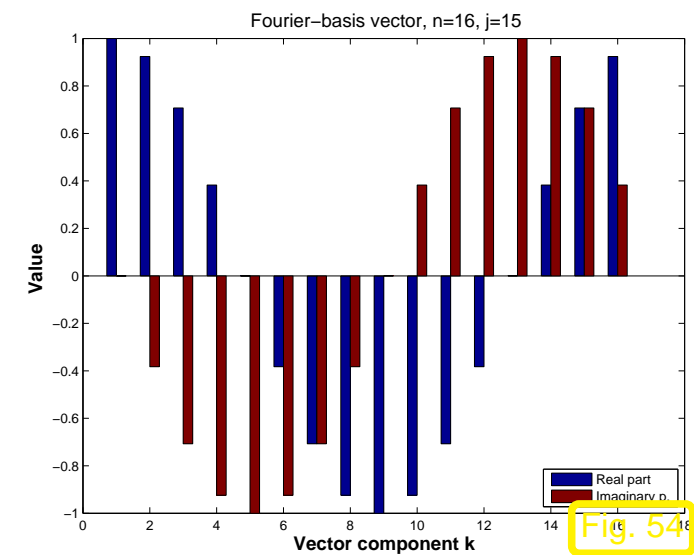
Some vectors of the Fourier basis ($n = 16$):



"low frequency"



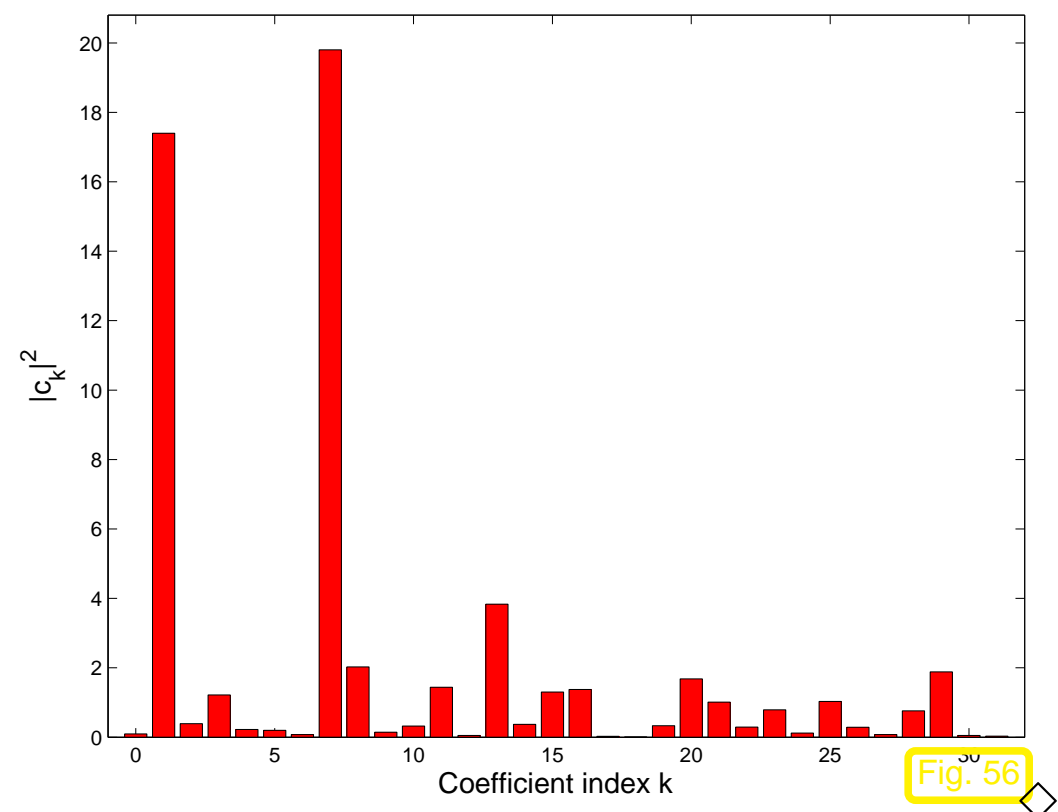
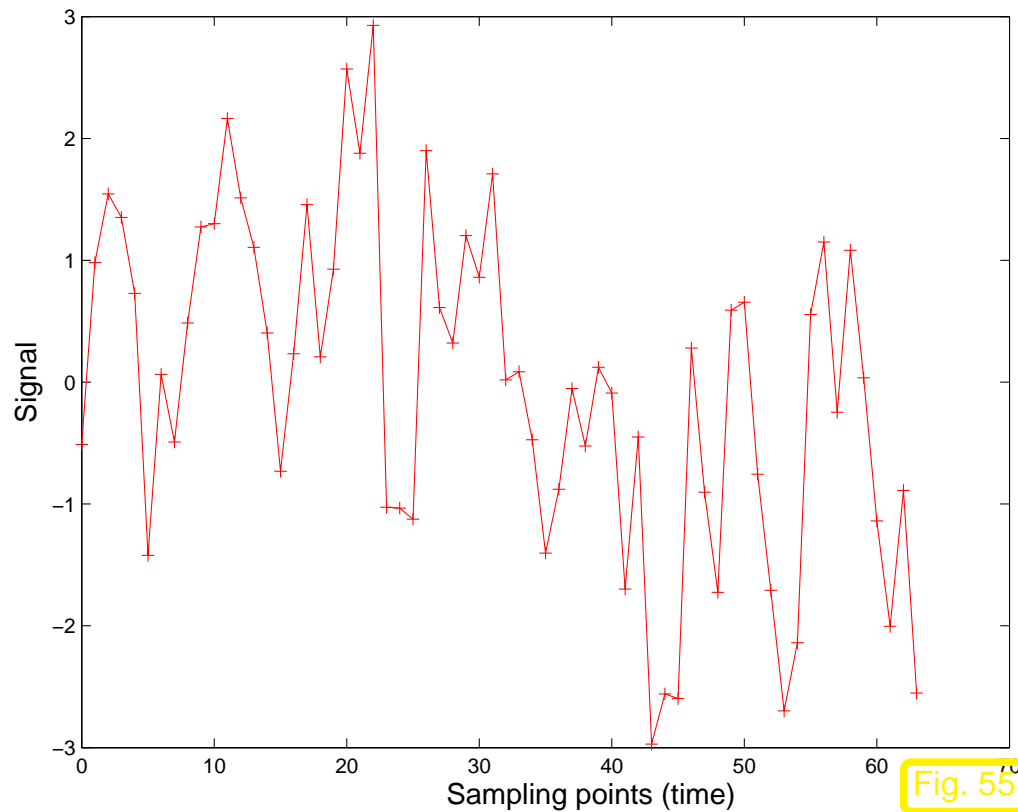
"high frequency"



"low frequency"

Extraction of characteristical frequencies from a distorted discrete periodical signal:

```
from numpy import sin, pi, linspace, random, fft
from pylab import plot, bar, show
t = linspace(0,63,64); x = sin(2*pi*t/64)+sin(7*2*pi*t/64)
y = x + random.randn(len(t)) %distortion
c = fft.fft(y); p = abs(c)**2/64
plot(t,y,'-+'); show()
bar(t[:32],p[:32]); show()
```



6.2 Fast Fourier Transform (FFT)

At first glance (at (6.1.4)): DFT in \mathbb{C}^n seems to require asymptotic computational effort of $O(n^2)$ (matrix \times vector multiplication with dense matrix).

Example 6.2.1 (Efficiency of `fft`).

`tic-toc`-timing: compare `fft`, loop based implementation, and direct matrix multiplication

Code 6.2.2: timing of different implementations of DFT

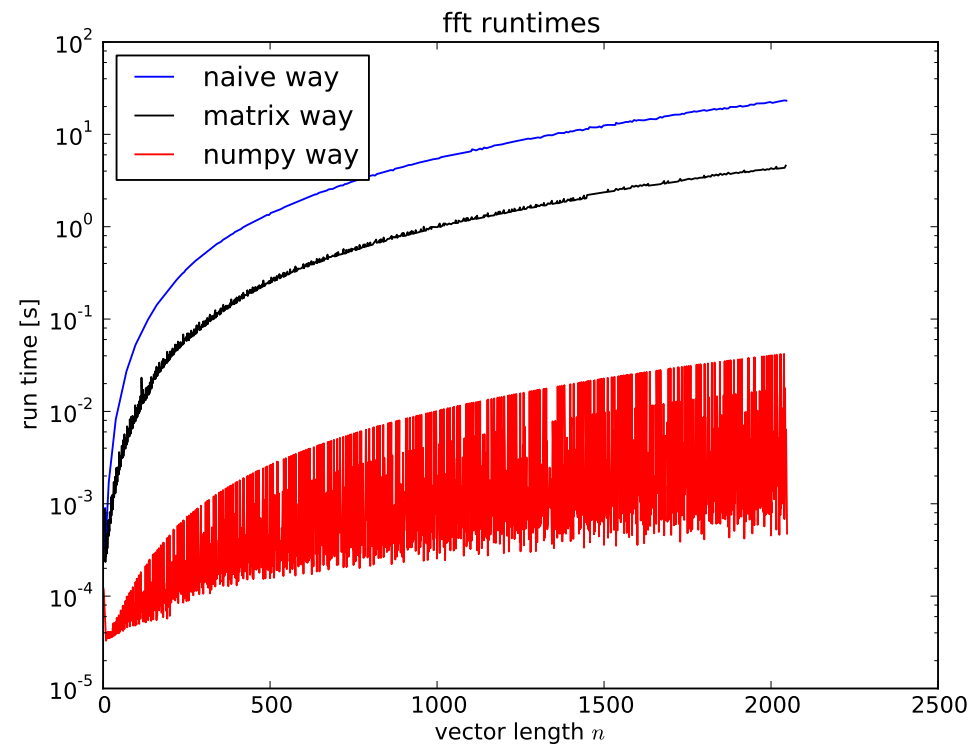
```
1 from numpy import zeros, random, exp, pi, meshgrid, r_, dot,  fft
2 import timeit
3
4 def naiveFT():
5     global y, n
6
7     c = 0.*1j*y
8     # naive
9     omega = exp(-2*pi*1j/n)
10    c[0] = y.sum(); s = omega
11    for jj in xrange(1,n):
12        c[jj] = y[n-1]
13        for kk in xrange(n-2,-1,-1): c[jj] = c[jj]*s+y[kk]
14        s *= omega
15    #return c
16
17 def matrixFT():
```

```
18 global y, n
19
20 # matrix based
21 I, J = meshgrid(r_[:n], r_[:n])
22 F = exp(-2*pi*1j*I*J/n)
23 tm = 10**3
24 c = dot(F,y)
25 #return c
26
27 def fftFT():
28     global y,n
29     c = fft.fft(y)
30     #return c
31
32 nrexp = 5
33 N = 2**11 # how large the vector will be
34 res = zeros((N,4))
35 for n in xrange(1,N+1):
36     y = random.rand(n)
37
38     t = timeit.Timer('naiveFT()', 'from __main__ import _naiveFT')
39     tn = t.timeit(number=nrexp)
40     t = timeit.Timer('matrixFT()', 'from __main__ import _matrixFT')
```

```
41 tm = t.timeit(number=nexp)
42 t = timeit.Timer('fftFT()', 'from __main__ import _fftFT')
43 tf = t.timeit(number=nexp)
44 #print n, tn, tm, tf
45 res[n-1] = [n, tn, tm, tf]
46
47 print res[:,3]
48 from pylab import semilogy, show, plot, savefig
49 semilogy(res[:,0], res[:,1], 'b-')
50 semilogy(res[:,0], res[:,2], 'k-')
51 semilogy(res[:,0], res[:,3], 'r-')
52 savefig('ffftime.eps')
53 show()
```

naive DFT-implementation

```
c = 0.*1j*y
omega = exp(-2*pi*1j/n)
c[0] = y.sum(); s = omega
for jj in xrange(1,n):
    c[jj] = y[n-1]
    for kk in xrange(n-2,-1,-1):
        s *= omega
```



Incredible! The `fft()`-function clearly beats the $O(n^2)$ asymptotic complexity of the other implementations. Note the logarithmic scale!



The secret of `fft()`:

the **Fast Fourier Transform** algorithm [15]

(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 (6.1.4) for $n = 2m$, $m \in \mathbb{N}$:

$$\begin{aligned}
 c_k &= \sum_{j=0}^{n-1} y_j e^{-\frac{2\pi i}{n} jk} \\
 &= \sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{n} 2jk} + \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{n} (2j+1)k} \\
 &= \underbrace{\sum_{j=0}^{m-1} y_{2j} \underbrace{e^{-\frac{2\pi i}{m} jk}}_{=\omega_m^{jk}}}_{=:\tilde{c}_k^{\text{even}}} + e^{-\frac{2\pi i}{n} k} \cdot \underbrace{\sum_{j=0}^{m-1} y_{2j+1} \underbrace{e^{-\frac{2\pi i}{m} jk}}_{=\omega_m^{jk}}}_{=:\tilde{c}_k^{\text{odd}}} .
 \end{aligned} \tag{6.2.1}$$

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^{odd} from DFTs of length m !

with $\mathbf{y}_{\text{even}} := (y_0, y_2, \dots, y_{n-2})^T \in \mathbb{C}^m$: $(\tilde{c}_k^{\text{even}})_{k=0}^{m-1} = \mathbf{F}_m \mathbf{y}_{\text{even}}$,

with $\mathbf{y}_{\text{odd}} := (y_1, y_3, \dots, y_{n-1})^T \in \mathbb{C}^m$: $(\tilde{c}_k^{\text{odd}})_{k=0}^{m-1} = \mathbf{F}_m \mathbf{y}_{\text{odd}}$.

(6.2.1): DFT of length $2m = 2 \times$ DFT of length m + $2m$ additions & multiplications

Code 6.2.3: Recursive FFT

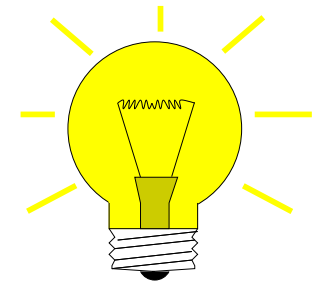
```
1 from numpy import linspace, random,  
    hstack, pi, exp  
2 def fftrec(y):  
3     n = len(y)  
4     if n == 1:  
5         c = y  
6         return c  
7     else:  
8         c0 = fftrec(y[::2])  
9         c1 = fftrec(y[1::2])  
10        r = (-2.j*pi/n) *  
            linspace(0,n-1,n)  
11        c = hstack((c0,c0)) +  
            exp(r) * hstack((c1,c1))  
12    return c
```

Idea:

divide & conquer recursion

(for DFT of length $n = 2^L$)

FFT-algorithm

Computational cost of `fftrec`:

$1 \times$ DFT of length 2^L

$2 \times$ DFT of length 2^{L-1}

$4 \times$ DFT of length 2^{L-2}



$2^L \times$ DFT of length 1

Code 6.2.2: each level of the recursion requires $O(2^L)$ elementary operations.

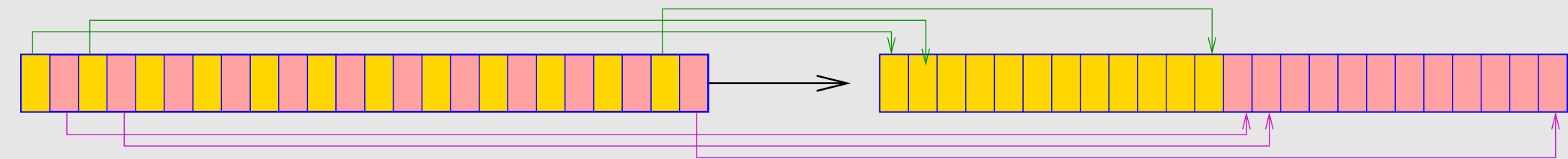
Asymptotic complexity of FFT algorithm, $n = 2^L$: $O(L2^L) = O(n \log_2 n)$

(fft-function: cost $\approx 5n \log_2 n$).

Remark 6.2.4 (FFT algorithm by matrix factorization).

For $n = 2m$, $m \in \mathbb{N}$,

permutation $P_m^{\text{OE}}(1, \dots, n) = (1, 3, \dots, n-1, 2, 4, \dots, n)$.



As $\omega_n^{2j} = \omega_m^j$:

permutation of rows $P_m^{\text{OE}} \mathbf{F}_n =$

$$\begin{pmatrix} & \mathbf{F}_m & & \mathbf{F}_m \\ \hline \mathbf{F}_m \begin{pmatrix} \omega_n^0 & & & \\ & \omega_n^1 & & \\ & & \dots & \\ & & & \omega_n^{n/2-1} \end{pmatrix} & \mathbf{F}_m \begin{pmatrix} \omega_n^{n/2} & & & \\ & \omega_n^{n/2+1} & & \\ & & \dots & \\ & & & \omega_n^{n-1} \end{pmatrix} \\ \hline \begin{pmatrix} & & & \\ \hline & & & \\ & & & \\ \hline & & & \end{pmatrix} \begin{pmatrix} \mathbf{I} & \mathbf{I} \\ \hline \omega_n^0 & -\omega_n^0 \\ & \omega_n^1 & -\omega_n^1 \\ & & \dots & \\ & & & \omega_n^{n/2-1} & -\omega_n^{n/2-1} \end{pmatrix} \end{pmatrix}$$

Example: factorization of Fourier matrix for $n = 10$

$$P_5^{\text{OE}} \mathbf{F}_{10} = \left(\begin{array}{ccccc|ccccc} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 \\ \omega^0 & \omega^4 & \omega^8 & \omega^2 & \omega^6 & \omega^0 & \omega^4 & \omega^8 & \omega^2 & \omega^6 \\ \omega^0 & \omega^6 & \omega^2 & \omega^8 & \omega^4 & \omega^0 & \omega^6 & \omega^2 & \omega^8 & \omega^4 \\ \omega^0 & \omega^8 & \omega^6 & \omega^4 & \omega^2 & \omega^0 & \omega^8 & \omega^6 & \omega^4 & \omega^2 \\ \hline \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 & \omega^8 & \omega^9 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 & \omega^2 & \omega^5 & \omega^8 & \omega^1 & \omega^4 & \omega^7 \\ \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 \\ \omega^0 & \omega^7 & \omega^4 & \omega^1 & \omega^8 & \omega^5 & \omega^2 & \omega^9 & \omega^6 & \omega^3 \\ \omega^0 & \omega^9 & \omega^8 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{array} \right), \quad \omega := \omega_{10}.$$



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. 6.2.1.

Remark 6.2.5 (FFT based on general factorization).

Fast Fourier transform algorithm for DFT of length $n = pq$, $p, q \in \mathbb{N}$ (Cooley-Tuckey-Algorithm)

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{jk} \stackrel{[j=:lp+m]}{=} \sum_{m=0}^{p-1} \sum_{l=0}^{q-1} y_{lp+m} e^{-\frac{2\pi i}{pq}(lp+m)k} = \sum_{m=0}^{p-1} \omega_n^{mk} \sum_{l=0}^{q-1} y_{lp+m} \omega_q^{l(k \bmod q)}.$$

(6.2.2)

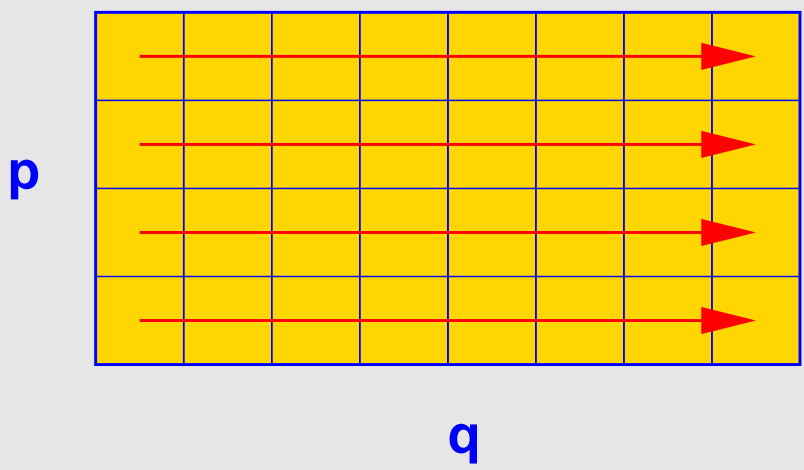
Step I: perform p DFTs of length q $z_{m,k} := \sum_{l=0}^{q-1} y_{lp+m} \omega_q^{lk}, \quad 0 \leq m < p, 0 \leq k < q.$

Step II: for $k =: rq + s, \quad 0 \leq r < p, 0 \leq s < q$

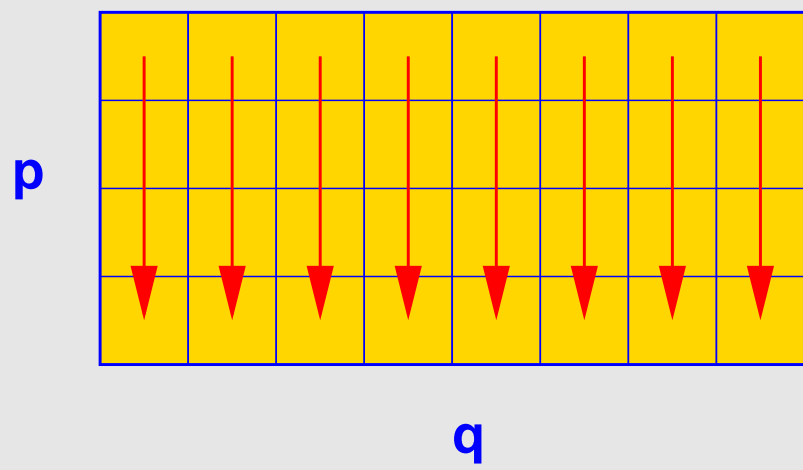
$$c_{rq+s} = \sum_{m=0}^{p-1} e^{-\frac{2\pi i}{pq}(rq+s)m} z_{m,s} = \sum_{m=0}^{p-1} (\omega_n^{ms} z_{m,s}) \omega_p^{mr}$$

and hence q DFTs of length p give all c_k .

Step I



Step II



Remark 6.2.6 (FFT for prime n).

When $n \neq 2^L$, even the Cooley-Tuckey algorithm of Rem. 6.2.5 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 [43]. 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:

$$\forall p \in \mathbb{N} \text{ prime} \quad \exists g \in \{1, \dots, p-1\}: \{g^k \bmod p: k = 1, \dots, p-1\} = \{1, \dots, p-1\},$$

► permutation $P_{p,g} : \{1, \dots, p-1\} \mapsto \{1, \dots, p-1\}$, $P_{p,g}(k) = g^k \bmod p$,

reversing permutation $P_k : \{1, \dots, k\} \mapsto \{1, \dots, k\}$, $P_k(i) = k - i + 1$.

For Fourier matrix $\mathbf{F} = (f_{ij})_{i,j=1}^p$: $P_{p-1}P_{p,g}(f_{ij})_{i,j=2}^p P_{p,g}^T$ is circulant.

Example for $p = 13$:

$g = 2$, permutation: $(2\ 4\ 8\ 3\ 6\ 12\ 11\ 9\ 5\ 10\ 7\ 1)$.

$$\mathbf{F}_{13} \longrightarrow \begin{array}{c|cccccccccccc} \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 \\ \hline \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 \\ \omega^0 & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} \\ \omega^0 & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} \\ \omega^0 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 \\ \omega^0 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 \\ \omega^0 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 \\ \omega^0 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 \\ \omega^0 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 \end{array}$$

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 $\text{c=fft}(\mathbf{y})$ for $\mathbf{y} \in \mathbb{C}^n = O(n \log n)$.

Theorem 6.3.1 (Trigonometric interpolant). Let N even and $z = \mathcal{F}_N y \in \mathbb{C}^N$. The trigonometric interpolant

$$p_N(x) := \sum_{k=-N/2}^{N/2} z_k e^{2\pi i k x} := \frac{1}{2} \left(z_{-N/2} e^{-2\pi i x N/2} + z_{N/2} e^{2\pi i x N/2} \right) + \sum_{|k| \leq N/2} z_k e^{2\pi i k x}$$

has the property that $p_N(x_\ell) = y_\ell$, where $x_\ell = \ell/N$.

► Fast interpolation by means of FFT: $O(n \log n)$ asymptotic complexity, see Sect. 6.2, provide uniformly distributed nodes $t_k = \frac{k}{2n+1}$, $k = 0, \dots, 2n$

► $(2n+1) \times (2n+1)$ linear system of equations:

$$\sum_{j=0}^{2n} \gamma_j \exp\left(2\pi i \frac{jk}{2n+1}\right) = z_k := \exp\left(2\pi i \frac{nk}{2n+1}\right) y_k, \quad k = 0, \dots, 2n.$$



$$\overline{\mathbf{F}}_{2n+1} \mathbf{c} = \mathbf{z}, \quad \mathbf{c} = (\gamma_0, \dots, \gamma_{2n})^T \xRightarrow{\text{Lemma 6.1.2}} \mathbf{c} = \frac{1}{2n+1} \mathbf{F}_{2n} \mathbf{z}. \quad (6.3.1)$$

$(2n+1) \times (2n+1)$ (conjugate) **Fourier matrix**, see (6.1.3)

Code 6.3.2: Efficient computation of coefficient of trigonometric interpolation polynomial (*equidistant nodes*)

```
from numpy import transpose, hstack, arange
from scipy import pi, exp, fft

def trigipequid(y):
    """ Efficient computation of coefficients in expansion
        \eqref{eq:trigpreal} for a trigonometric
        interpolation polynomial in equidistant points \Blue{(\frac{j}{2n+1}, y_j)},
        \Blue{j = 0, \dots, 2n}
        \texttt{y} has to be a row vector of odd length, return values are
        column vectors
    """

    N = y.shape[0]
```



```

if N%2 != 1:
    raise ValueError("Odd_number_of_points_required!")

```

```

n = (N-1.0)/2.0
z = arange(N)

```

```

# See (6.3.1)
c = fft(exp(2.0j*pi*(n/N)*z)*y) / (1.0*N)

```

```

# From (??):  $\alpha_j = \frac{1}{2}(\gamma_{n-j} + \gamma_{n+j})$  and
#  $\beta_j = \frac{1}{2i}(\gamma_{n-j} - \gamma_{n+j})$ ,  $j = 1, \dots, n$ ,  $\alpha_0 = \gamma_n$ 

```

```

a = hstack([ c[n], c[n-1::-1]+c[n+1:N] ])
b = hstack([ -1.0j*(c[n-1::-1]-c[n+1:N]) ])

```

```

return (a,b)

```

```

if __name__ == "__main__":
    from numpy import linspace

```

```

y = linspace(0,1,9)

```

```

# Expected Result:

```

```
# a =  
#  
# 0.50000 - 0.00000i  
# -0.12500 + 0.00000i  
# -0.12500 + 0.00000i  
# -0.12500 - 0.00000i  
# -0.12500 + 0.00000i  
#  
# b =  
#  
# -0.343435 + 0.000000i  
# -0.148969 + 0.000000i  
# -0.072169 - 0.000000i  
# -0.022041 + 0.000000i  
  
w = trigipequid(y)  
  
print(w)
```

Code 6.3.3: Computation of coefficients of trigonometric interpolation polynomial, general nodes

```
from numpy import hstack, arange, sin, cos, pi, outer  
from numpy.linalg import solve  
  
def trigpolycoeff(t, y):
```

```
"""Computes_expansion_coefficients_of_trigonometric_polyonomials_  
    \eqref{eq:trigpreal}  
_____\texttt{t}:_row_vector_of_nodes_\Blue{t_0,\dots,t_n \in [0,1[}  
\texttt{y}: row vector of data \Blue{y_0,\dots,y_n}  
return values are column vectors of expansion coefficients \Blue{\alpha_j},  
    \Blue{\beta_j}  
"""
```

```
N = y.shape[0]
```

```
if N%2 != 1:  
    raise ValueError("Odd_number_of_points_required!")
```

```
n = (N-1.0)/2.0
```

```
M = hstack([ cos(2*pi*outer(t,arange(0,n+1))),  
            sin(2*pi*outer(t,arange(1,n+1))) ])  
c = solve(M, y)
```

```
a = c[0:n+1]  
b = c[n+1:]
```

```
return (a, b)
```

```
if __name__ == "__main__":  
    from numpy import linspace  
  
    t = linspace(0, 1, 5)  
    y = linspace(0, 5, 5)  
  
    # Expected values:  
    # a =  
    #  
    # 2.5000e+00  
    # -1.1150e-16  
    # -2.5000e+00  
    #  
    # b =  
    #  
    # -2.5000e+00  
    # -1.0207e+16  
  
    z = trigpolycoeff(t, y)  
  
    print(z)
```

Example 6.3.4 (Runtime comparison for computation of coefficient of trigonometric interpolation polynomials).

tic-toc-timings

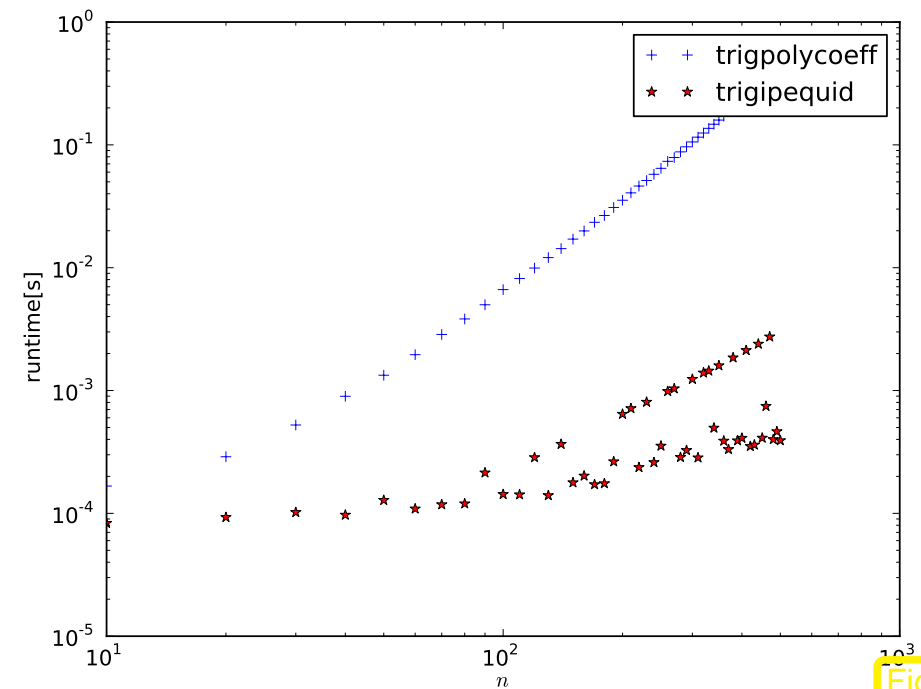


Fig. 57

Code 6.3.5: Runtime comparison

```

from numpy import linspace, pi, exp, cos, array
from matplotlib.pyplot import *
import time

from trigpolycoeff import trigpolycoeff
from trigipequid import trigipequid

def trigipequidtiming():
    """Runtime_comparison_between_efficient_(→ Code~\ref{trigipequid}) and direct
    computation
    (→ Code~\ref{trigpolycoeff} of coefficients of trigonoetric interpolation
    polynomial in
    equidistant points.

```

```
"""
```

```
Nruns = 3
```

```
times = []
```

```
for n in xrange(10, 501, 10):  
    print(n)
```

```
    N = 2*n+1
```

```
    t = linspace(0, 1 - 1.0/N, N)  
    y = exp(cos(2*pi*t))
```

```
    # Infinity for all practical purposes
```

```
    t1 = 10.0**10
```

```
    t2 = 10.0**10
```

```
    for k in xrange(Nruns):
```

```
        tic = time.time()
```

```
        a, b = trigpolycoeff(t, y)
```

```
        toc = time.time() - tic
```

```
        t1 = min(t1, toc)
```

```
        tic = time.time()
```

```
        a, b = trigipequid(y)
```

```
        toc = time.time() - tic
```

```
        t2 = min(t2, toc)
```

```
    times.append( (n, t1, t2) )
```

```
times = array(times)

fig = figure()
ax = fig.gca()

ax.loglog(times[:,0], times[:,1], "b+", label="trigpolycoeff")
ax.loglog(times[:,0], times[:,2], "r*", label="trigipequid")

ax.set_xlabel(r" $n$ ")
ax.set_ylabel(r"runtime[s]")
ax.legend()

fig.savefig("../PICTURES/trigipequidtiming.eps")

if __name__ == "__main__":
    trigipequidtiming()
    show()
```

Same observation as in Ex. 6.2.1: massive gain in efficiency through relying on FFT.



Remark 6.3.6 (Efficient evaluation of trigonometric interpolation polynomials).

Task: evaluation of trigonometric polynomial (??) at *equidistant* points $\frac{k}{N}$, $N > 2n$. $k = 0, \dots, N-1$.

$$(??) \quad \blacktriangleright \quad q(k/N) = e^{-2\pi i k/N} \sum_{j=0}^{2n} \gamma_j \exp(2\pi i \frac{kj}{N}), \quad k = 0, \dots, N-1.$$

$$\blacktriangleright \quad q(k/N) = e^{-2\pi i kn/N} v_j \quad \text{with} \quad \mathbf{v} = \bar{\mathbf{F}}_N \tilde{\mathbf{c}}, \quad (6.3.2)$$

Fourier matrix, see (6.1.3).

where $\tilde{\mathbf{c}} \in \mathbb{C}^N$ is obtained by **zero padding** of $\mathbf{c} := (\gamma_0, \dots, \gamma_{2n})^T$:

$$(\tilde{\mathbf{c}})_k = \begin{cases} \gamma_j & , \text{ for } k = 0, \dots, 2n, \\ 0 & , \text{ for } k = 2n+1, \dots, N-1. \end{cases}$$

Code 6.3.7: Fast evaluation of trigonometric polynomial at *equidistant* points

```
from numpy import transpose, vstack, hstack, zeros, conj, exp, pi
from scipy import fft

def trigipequidcomp(a, b, N):
    """ Efficient_evaluation_of_trigonometric_polynomial_at_equidistant_
        points
    """
    column_vectors = [texttt{a} and [texttt{b} pass_coefficients [Blue{\alpha_j},
```



```
\Blue{\beta_j} in
representation \eqref{eq:trigpreal}
"""
```

```
n = a.shape[0] - 1
```

```
if N < 2*n-1:
    raise ValueError("N_too_small")
```

```
gamma = 0.5*hstack([ a[-1:0:-1] + 1.0j*b[-1::-1], 2*a[0], a[1:] -
    1.0j*b[0:] ])
```

```
# Zero padding
ch = hstack([ gamma, zeros((N-(2*n+1),)) ])
```

```
# Multiplication with conjugate Fourier matrix
v = conj(fft(conj(ch)))
```

```
# Undo rescaling
q = exp(-2.0j*pi*n*arange(N)/(1.0*N)) * v
```

```
return q
```

```
if __name__ == "__main__":  
    from numpy import arange  
  
    a = arange(1,6)  
    b = arange(6,10)  
    N = 10  
  
    # Expected values:  
    #  
    # 15.00000 - 0.00000i 21.34656 + 0.00000i -5.94095 - 0.00000i  
    # 8.18514 + 0.00000i -3.31230 - 0.00000i 3.00000 + 0.00000i  
    # -1.68770 - 0.00000i -2.71300 - 0.00000i 0.94095 + 0.00000i  
    # -24.81869 - 0.00000i  
  
    print(a)  
    print(b)  
    print(N)  
  
    w = trigipequidcomp(a, b, N)  
  
    print(w)
```

```
from numpy import zeros, fft, sqrt, pi, real, linalg, linspace, sin
```

```
def evaliptrig(y,N):  
    n = len(y)  
    if (n%2) == 0:  
        c = fft.ifft(y)  
        a = zeros(N, dtype=complex)  
        a[:n/2] = c[:n/2]  
        a[N-n/2:] = c[n/2:]  
        v = fft.fft(a);  
        return v  
    else: raise TypeError, 'odd_length'
```

```
f = lambda t: 1./sqrt(1.+0.5*sin(2*pi*t))
```

```
vlinf = []; vl2 = []; vn = []
```

```
N = 4096; n = 2
```

```
while n < 1+2**7:  
    t = linspace(0,1,n+1)  
    y = f(t[:-1])  
    v = real(evaliptrig(y,N))  
    t = linspace(0,1,N+1)  
    fv = f(t)  
    d = abs(v-fv[:-1]); linf = d.max()
```

```
l2 = linalg.norm(d)/sqrt(N)
vlinf += [linf]; vl2 += [l2]; vn += [n]
n *= 2
```

```
from pylab import semilogy, show
semilogy(vn, vl2, '—+'); show()
```

compare with

Code 6.3.9: *Equidistant* points: fast on the fly evaluation of trigonometric interpolation polynomial

```
from numpy import zeros, fft, sqrt, pi, real, linalg, linspace, sin
```

```
def evaliptrig(y,N):
    n = len(y)
    if (n%2) == 0:
        c = fft.fft(y)*1./n
        a = zeros(N, dtype=complex)
        a[:n/2] = c[:n/2]
        a[N-n/2:] = c[n/2:]
        v = fft.ifft(a)*N;
        return v
    else: raise TypeError, 'odd_length '
```

```
f = lambda t: 1./sqrt(1.+0.5*sin(2*pi*t))
```

```
vlinf = []; vl2 = []; vn = []
N = 4096; n = 2
while n < 1+2**7:
    t = linspace(0,1,n+1)
    y = f(t[:-1])
    v = real(evaliptrig(y,N))
    t = linspace(0,1,N+1)
    fv = f(t)
    d = abs(v-fv[:-1]); linf = d.max()
    l2 = linalg.norm(d)/sqrt(N)
    vlinf += [linf]; vl2 += [l2]; vn += [n]
    n *= 2

from pylab import semilogy, show
semilogy(vn, vl2, '-+'); show()
```



6.4 Trigonometric Interpolation: Error Estimates

Linear trigonometric interpolation operator: for 1-periodic $f : \mathbb{R} \mapsto \mathbb{C}$

$$\mathsf{T}_n(f) := p \in \mathcal{P}_{n-1}^T \quad \text{with} \quad p\left(\frac{j}{n}\right) = y_j, \quad j = 0, \dots, n-1.$$

Example 6.4.1 (Trigonometric interpolation).

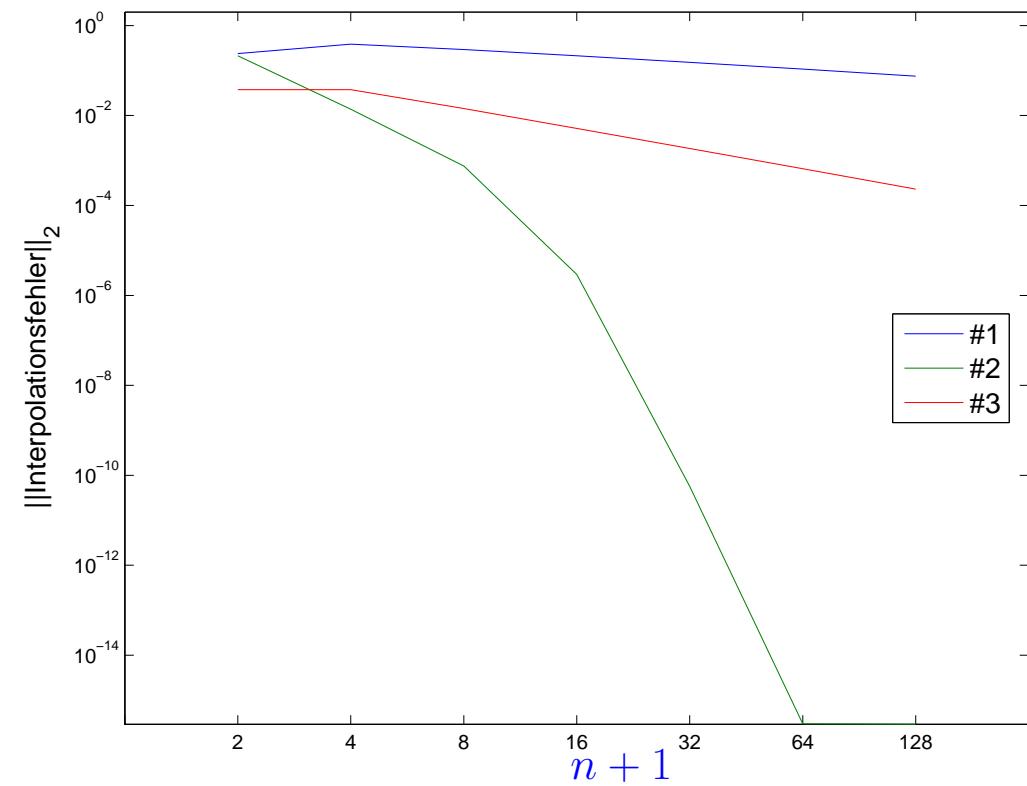
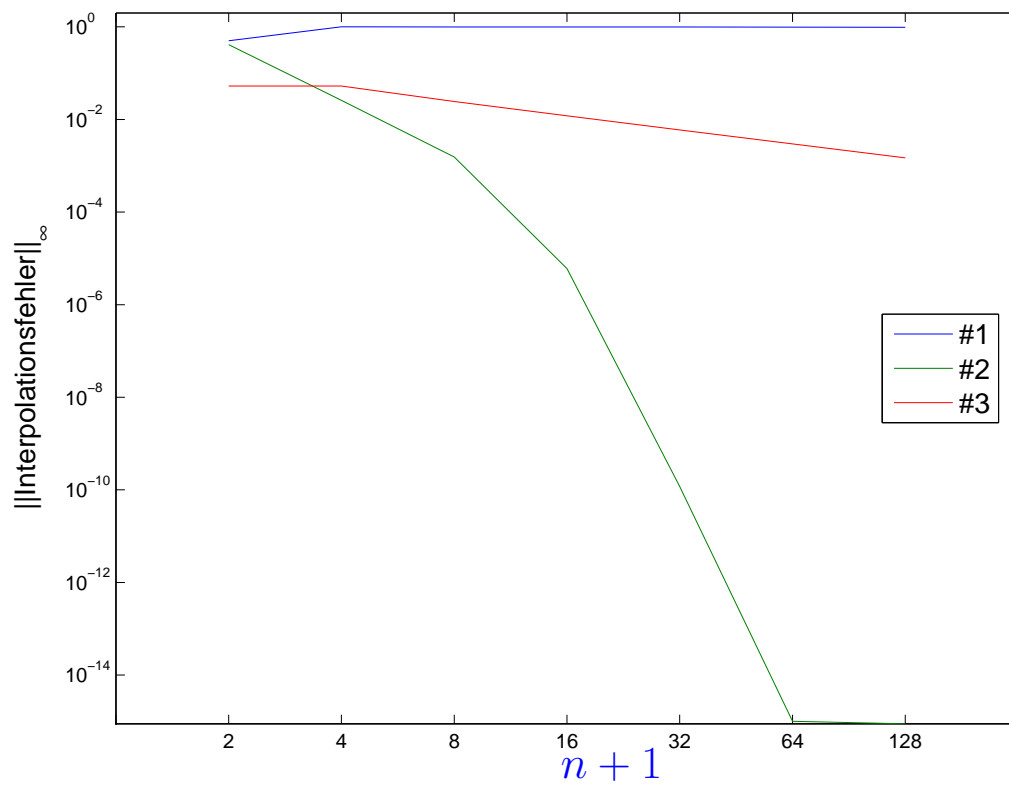
#1 step function: $f(t) = 0$ for $|t - \frac{1}{2}| > \frac{1}{4}$, $f(t) = 1$ for $|t - \frac{1}{2}| \leq \frac{1}{4}$

#2 smooth periodic function: $f(t) = \frac{1}{\sqrt{1 + \frac{1}{2} \sin(2\pi t)}}$.

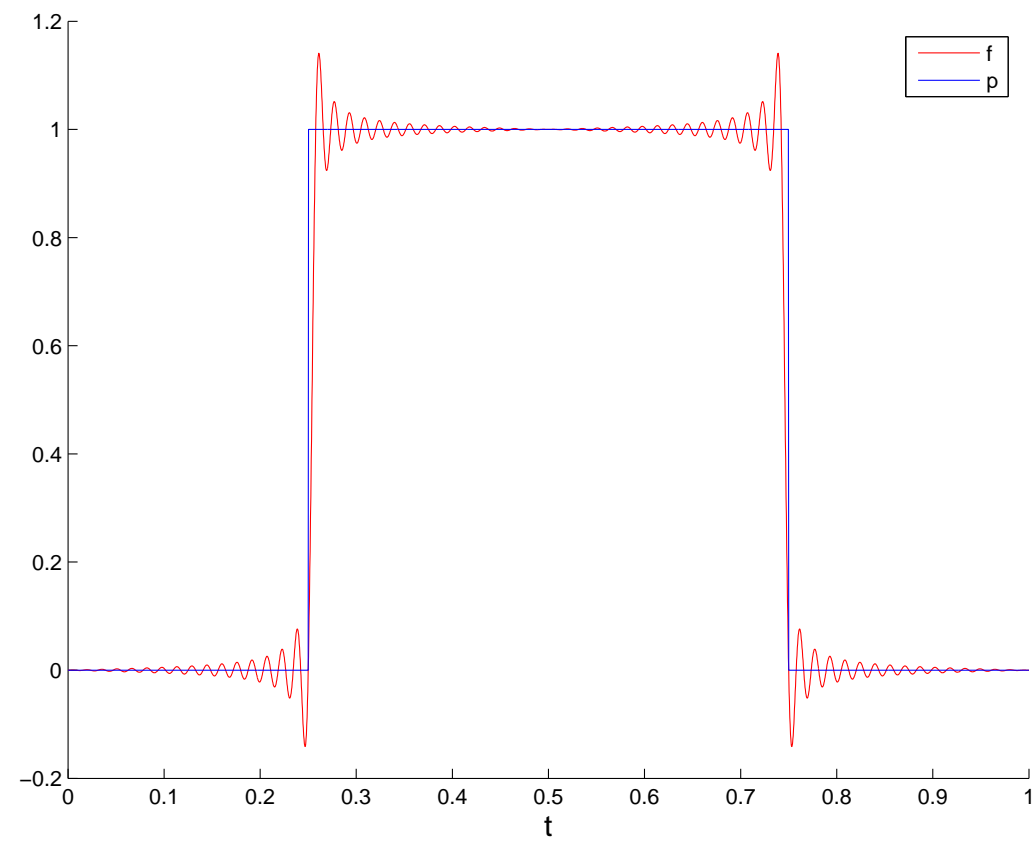
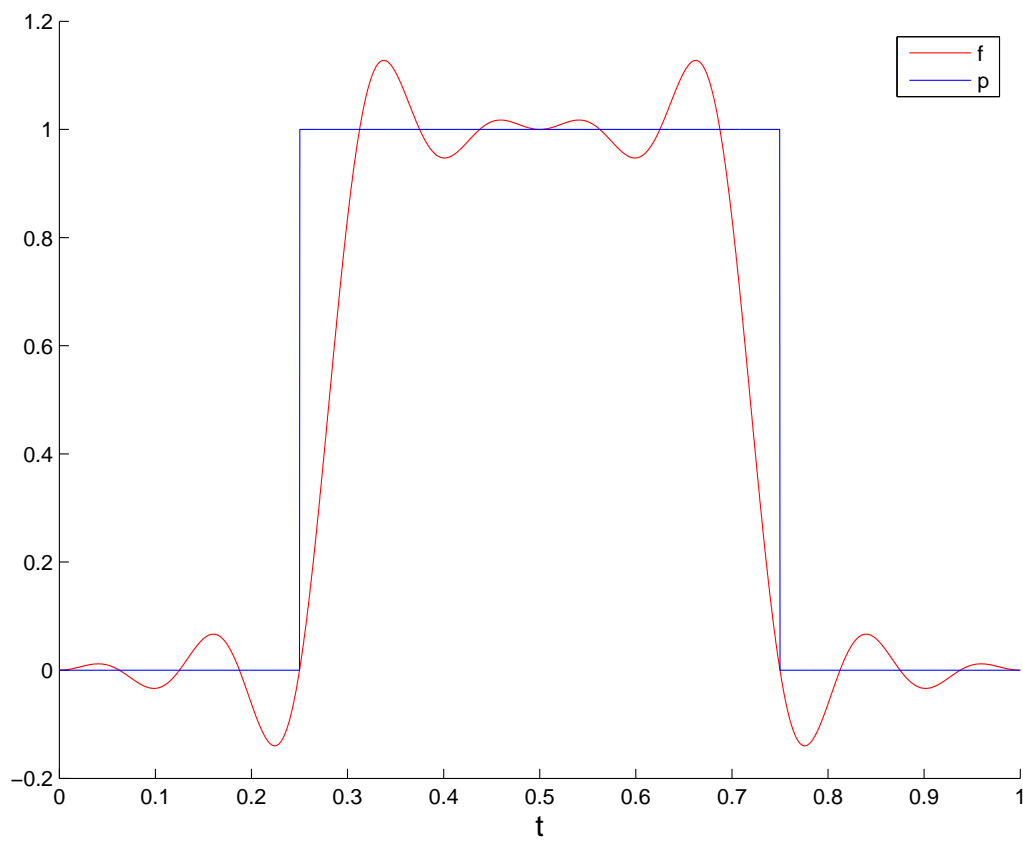
#3 hat function: $f(t) = |t - \frac{1}{2}|$

Note: computation of the norms of the interpolation error:

```
t = linspace(0,1,n+1);  y = f(t[:-1])
v = real(evaliptrig(y,N))
t = linspace(0,1,N+1);  fv = f(t)
d = abs(v-fv[:-1]);  linf = d.max()
l2 = linalg.norm(d)/sqrt(N)
```



Note: Cases #1, #3: algebraic convergence
Case #2: exponential convergence



$n = 16$

Gibbs' phenomenon: ripples near discontinuities

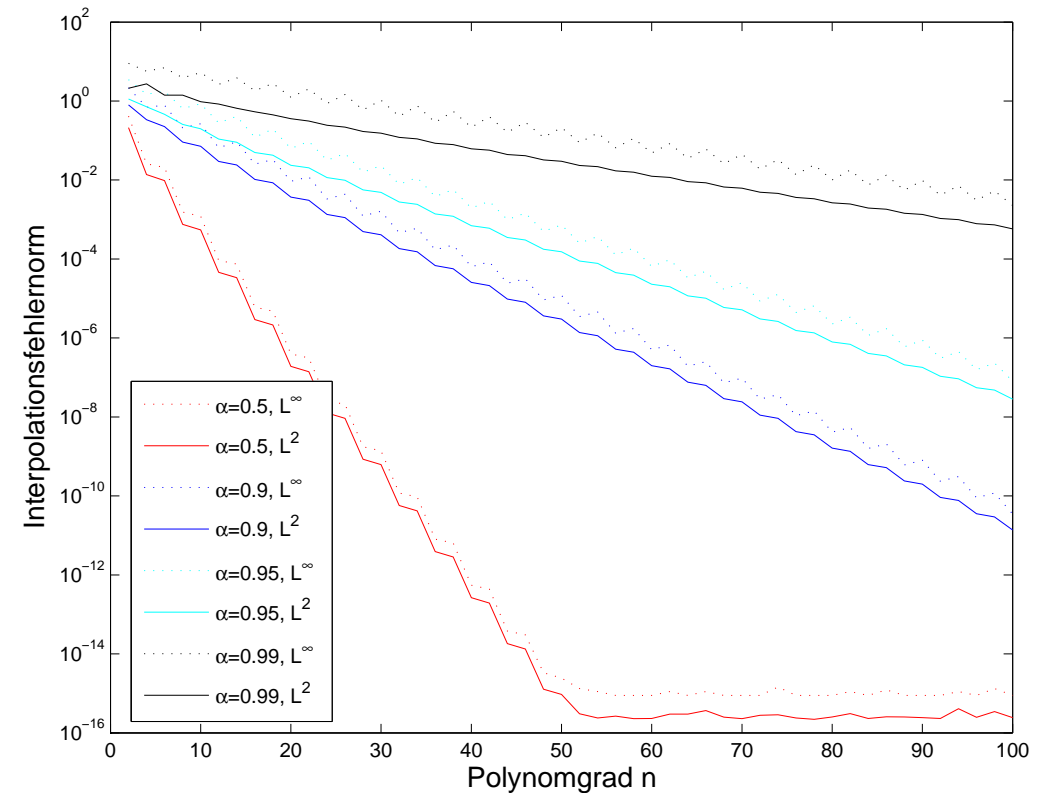
$n = 128$



Example 6.4.2 (Trigonometric interpolation of analytic functions).

$$f(t) = \frac{1}{\sqrt{1 - \alpha \sin(2\pi t)}} \quad \text{auf } I = [0, 1] .$$

approximation of the norms: “oversampling” in 4096 points



Note: exponential convergence in degree n , quicker for smaller α



Analysis uses: trigonometric polynomials = partial sums of **Fourier series**

Theorem 6.4.1 (Error of DFT; aliasing formula). *If $\sum_{k \in \mathbb{Z}} \hat{f}(k)$ absolut converges, then*

$$\hat{f}_N(k) - \hat{f}(k) = \sum_{\substack{j \in \mathbb{Z} \\ j \neq 0}} \hat{f}(k + jN)$$

Corollary 6.4.2. Let $f \in C^p$ with $p \geq 2$ and 1-periodic. Then it holds:

$$\hat{f}_N(k) - \hat{f}(k) = O(N^{-p}) \quad \text{for } |k| \leq \frac{N}{2}$$

$$\text{for: } h = \frac{1}{N}, \quad h \sum_{j=0}^{N-1} f(x_j) - \int_0^1 f(x) dx = O(h^p).$$

Remark 6.4.3. $\hat{f}_N(k)$ is N -periodic, while $\hat{f}(k) \rightarrow 0$ quickly

$\hat{f}_N(k)$ is a bad approximation of $\hat{f}(k)$ for k large ($k \approx N$), but for $|k| \leq \frac{N}{2}$ it is good enough.

Theorem 6.4.3 (Error of trigonometric interpolation). If f is 1-periodic and $\sum_{k \in \mathbb{Z}} \hat{f}(k)$ absolutely converges, then

$$|p_N(x) - f(x)| \leq 2 \sum_{|k| \geq N/2} |\hat{f}(k)| \quad \forall x \in \mathbb{R}$$

Corollary 6.4.4 (Sampling-Theorem). Let f 1-periodic with maximum frequency M : $\hat{f}(k) = 0$ for all $|k| > M$. Then $p_N(x) = f(x)$ for **all** x , if $N > 2M$

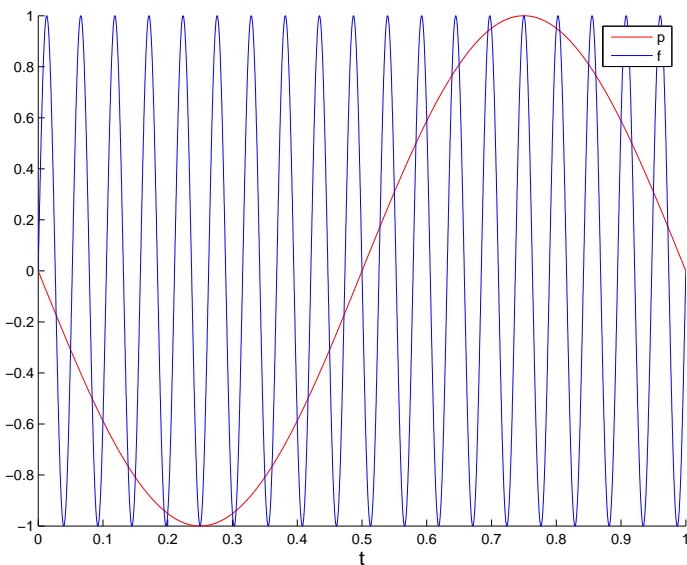
data compression

Remark 6.4.4 (Aliasing).

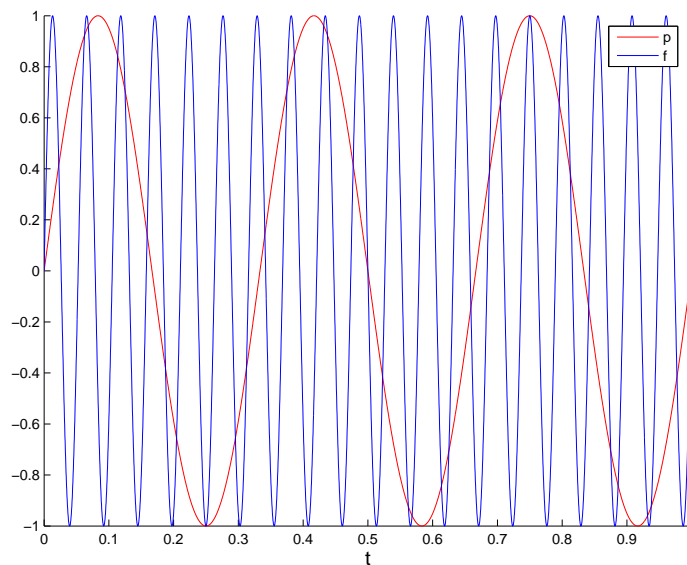
$$\mathcal{T} = \left\{ \frac{j}{n} \right\}_{j=0}^{n-1} \blacktriangleright e_{|\mathcal{T}}^{2\pi i N t} = e_{|\mathcal{T}}^{2\pi i (N-n)t} \blacktriangleright \mathsf{T}_n(e^{2\pi i N \cdot}) = \mathsf{T}_n(e^{2\pi i (N \bmod n) \cdot}) .$$

hence: The trigonometric interpolation of $t \rightarrow e^{2\pi i N t}$ and $t \rightarrow e^{2\pi i (N \bmod n)t}$ of degree n give the same trigonometric interpolation polynomial ! \rightarrow **Aliasing**

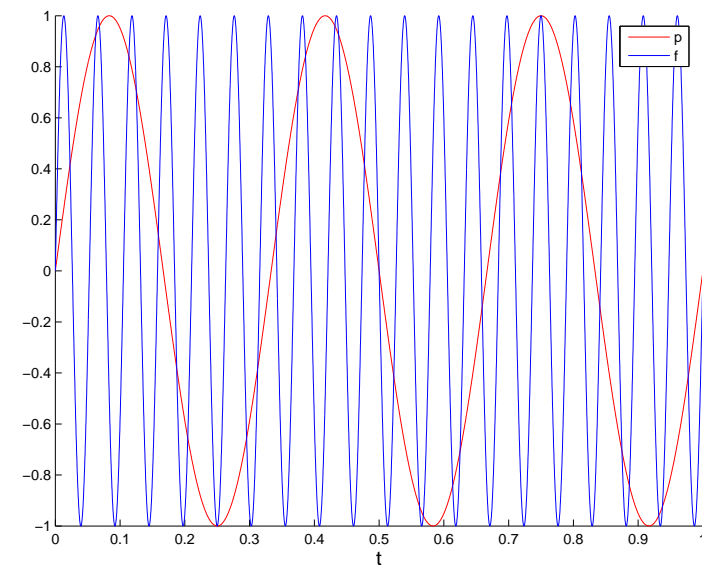
Example for $f(t) = \sin(2\pi \cdot 19t)$ $\succ N = 38$:

 Gradinaru
D-MATH


$n = 4$



$n = 8$



$n = 16$



For $f \in C([0, 1])$ 1-periodic, $n = 2m$:

$$f(t) = \sum_{j=-\infty}^{\infty} \hat{f}(j) e^{2\pi i j t} \Rightarrow T_n(f)(t) = \sum_{j=-m+1}^m \gamma_j e^{2\pi i j t}, \quad \gamma_j = \sum_{l=-\infty}^{\infty} \hat{f}(j + ln).$$

Fourier coefficients of the error function $e = f - T_n(f)$:

$$\hat{e}(j) = \begin{cases} - \sum_{|l|=1}^{\infty} \hat{f}(j + ln) & , \text{ if } -m+1 \leq j \leq m, \\ \hat{f}(j) & , \text{ if } j \leq -m \vee j > m. \end{cases}$$

$$\stackrel{(6.0.1)}{\Rightarrow} \|f - T_n(f)\|_{L^2([0,1])}^2 \leq \left| \sum_{|l|=1}^{\infty} \hat{f}(j + ln) \right|^2 + \sum_{|j| \geq m} |\hat{f}(j)|^2. \quad (6.4.1)$$

may be estimated, if we know the decay of the Fourier coefficients $\hat{f}(j) \Leftrightarrow$ smoothness of f , see (6.0.2)

Theorem 6.4.5 (Error estimate for trigonometric interpolation). For $k \in \mathbb{N}$:

$$f^{(k)} \in L^2(]0, 1[) \Rightarrow \|f - \mathsf{T}_n f\|_{L^2(]0, 1[)} \leq \sqrt{1 + c_k} n^{-k} \|f^{(k)}\|_{L^2(]0, 1[)} ,$$

with $c_k = 2 \sum_{l=1}^{\infty} (2l - 1)^{-2k}$.

6.5 DFT and Chebychev Interpolation

Let $p \in \mathcal{P}_n$ be the Chebychev interpolation polynom on $[-1, 1]$ of $f : [-1, 1] \mapsto \mathbb{C}$ (\rightarrow section 5.4)

$$p(t_k) = f(t_k) \quad \text{for Chebychev nodes, see (5.4.5),} \quad t_k := \cos \left(\frac{2k + 1}{2(n + 1)} \pi \right), \quad k = 0, \dots, n .$$

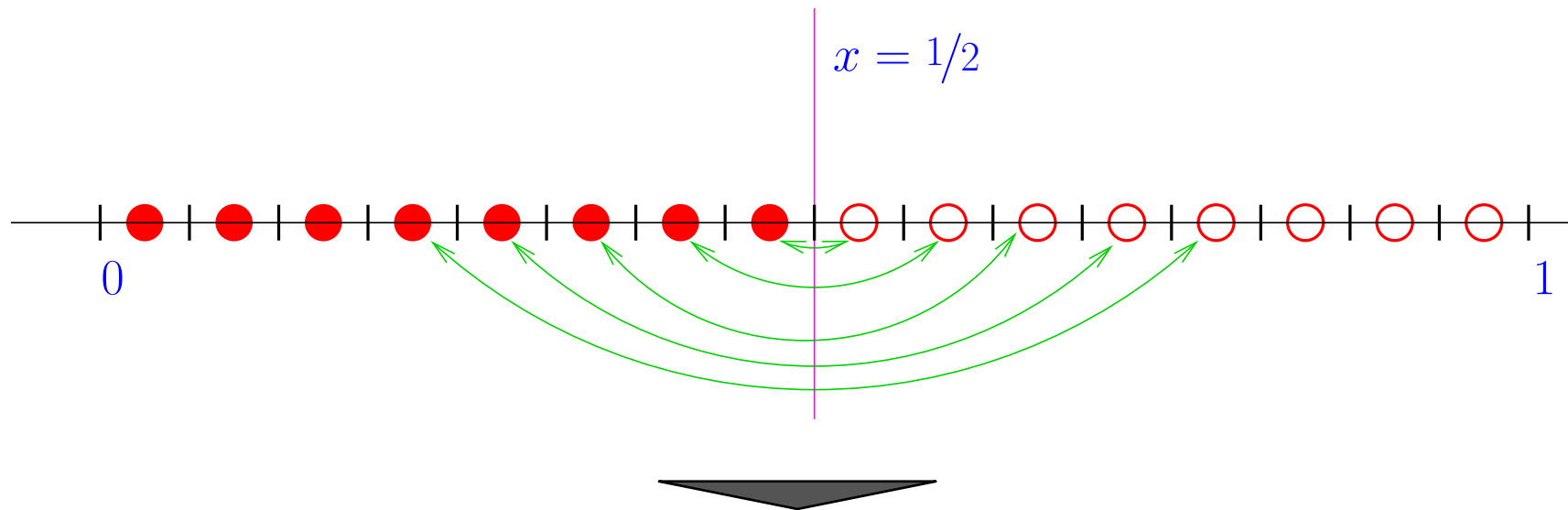
Define now the corresponding functions g, q :

$$\left. \begin{array}{ll} f : [-1, 1] \mapsto \mathbb{C} & \Leftrightarrow g(s) := f(\cos 2\pi s) , \\ p : [-1, 1] \mapsto \mathbb{C} & \Leftrightarrow q(s) := p(\cos 2\pi s) , \end{array} \right\} \quad \text{1-periodic, symmetric wrt. 0 and } \frac{1}{2}$$

Hence:

$$p(t_k) = f(t_k) \Leftrightarrow q \left(\frac{2k + 1}{4(n + 1)} \right) = g \left(\frac{2k + 1}{4(n + 1)} \right) . \quad (6.5.1)$$

and with a translation $\tilde{s} = s + \frac{1}{4(n+1)}$ we have:



$$\begin{aligned} \tilde{g}(s) &:= g\left(s + \frac{1}{4(n+1)}\right), \\ \tilde{q}(s) &:= q\left(s + \frac{1}{4(n+1)}\right) \end{aligned} \quad \Rightarrow \quad \tilde{q}\left(\frac{k}{2(n+1)}\right) = \tilde{g}\left(\frac{k}{2n+2}\right), \quad k = 0, \dots, 2n.$$

! We show: $q(s)$ is trigonometric polynomial: $q \in \mathcal{P}_{2n}^T$, namely the trigonometric interpolation polynomial of \tilde{g} . Indeed, since p is the Chebychev interpolation polynomial:

$$\begin{aligned}
 q(s) &= p(\cos 2\pi s) = \sum_{j=0}^n \gamma_j T_j(\cos 2\pi s) = \sum_{j=0}^n \gamma_j \cos 2\pi s = \gamma_0 + \sum_{j=-n, j \neq 0}^n \frac{1}{2} \gamma_{|j|} e^{-2\pi i j s} = \\
 &\gamma_0 + \sum_{j=-n, j \neq 0}^n \frac{1}{2} \gamma_{|j|} e^{2\pi i \frac{j}{4(n+1)}} \cdot e^{-2\pi i j \left(s + \frac{1}{4(n+1)}\right)} = \sum_{j=-n}^n \tilde{\gamma}_j e^{-2\pi i j \tilde{s}} \\
 &\text{with } \tilde{\gamma}_j := \begin{cases} \frac{1}{2} \exp\left(2\pi i \frac{j}{4(n+1)}\right) \gamma_{|j|} & , \text{ if } j \neq 0 , \\ \gamma_0 & , \text{ for } j = 0 . \end{cases}
 \end{aligned}$$

Hence

$$\begin{aligned}
 \tilde{q}(s) &= \sum_{j=-n}^n \tilde{\gamma}_j e^{-2\pi i j s} \\
 \text{and } \tilde{q}\left(\frac{k}{2(n+1)}\right) &= \tilde{g}\left(\frac{k}{2(n+1)}\right) , \quad \text{for all } k = 0, \dots, 2n .
 \end{aligned}$$

that means that \tilde{q} is the trigonometric interpolation polynomial of \tilde{g} .

Hence $\|f - p\|_{L^\infty([-1,1])} = \|\tilde{g} - \tilde{q}\|_{L^\infty([0,1])}$

and the error estimates from the trigonometric interpolation directly transfers here.

```

1 from numpy import exp, pi, real, hstack, arange
2 from scipy import fft
3
4 def chebexp(y):
5     """ Efficiently compute coefficients  $\alpha_j$  in the Chebychev
6         expansion
7          $p = \sum_{j=0}^n \alpha_j T_j$  of  $p \in \mathcal{P}_n$  based on values  $y_k$ ,
8          $k = 0, \dots, n$ , in Chebychev nodes  $t_k$ ,  $k = 0, \dots, n$ .
9         These values are
10        passed in the row vector y.
11        """
12
13    # degree of polynomial
14    n = y.shape[0] - 1
15
16    # create vector  $z$  by wrapping and componentwise scaling
17
18    # r.h.s. vector
19    t = arange(0, 2*n+2)
20    z = exp(-pi*1.0j*n/(n+1.0)*t) * hstack([y, y[::-1]])
21
22    # Solve linear system (??) with effort  $O(n \log n)$ 

```



```
21 c = ifft(z)
22
23 # recover  $\beta_j$ , see (??)
24 t = arange(-n, n+2)
25 b = real(exp(0.5j*pi/(n+1.0)*t) * c)
26
27 # recover  $\alpha_j$ , see (??)
28 a = hstack([ b[n], 2*b[n+1:2*n+1] ])
29
30 return a
31
32 if __name__ == "__main__":
33     from numpy import array
34
35     # Test with arbitrary values
36     y = array([1, 2, 3, 3.5, 4, 6.5, 6.7, 8, 9])
37
38     # Expected values:
39     # 4.85556 -3.66200 0.23380 -0.25019 -0.15958 -0.36335 0.18889 0.16546 -0.27329
40
41     w = chebexp(y)
42
43     print(w)
```

6.6 Essential Skills Learned in Chapter 6

You should know:

- the idea behind the trigonometric approximation;
- what is the Gibbs phenomenon;
- the discrete Fourier transform and its use for the trigonometric interpolation;
- the idea and the importance of the fast Fourier transform;
- how to use the fast Fourier transform for the efficient trigonometrical interpolation;
- the error behavior for the trigonometric interpolation;
- the aliasing formula and the Sampling-Theorem;
- how to use the fast Fourier transform for the efficient Chebychev interpolation.

7

Numerical Quadrature

Numerical quadrature

= Approximate evaluation of $\int_{\Omega} f(\mathbf{x}) \, d\mathbf{x}$, integration domain $\Omega \subset \mathbb{R}^d$


Continuous function $f : \Omega \subset \mathbb{R}^d \mapsto \mathbb{R}$ only available as `function y = f(x)` (point evaluation)

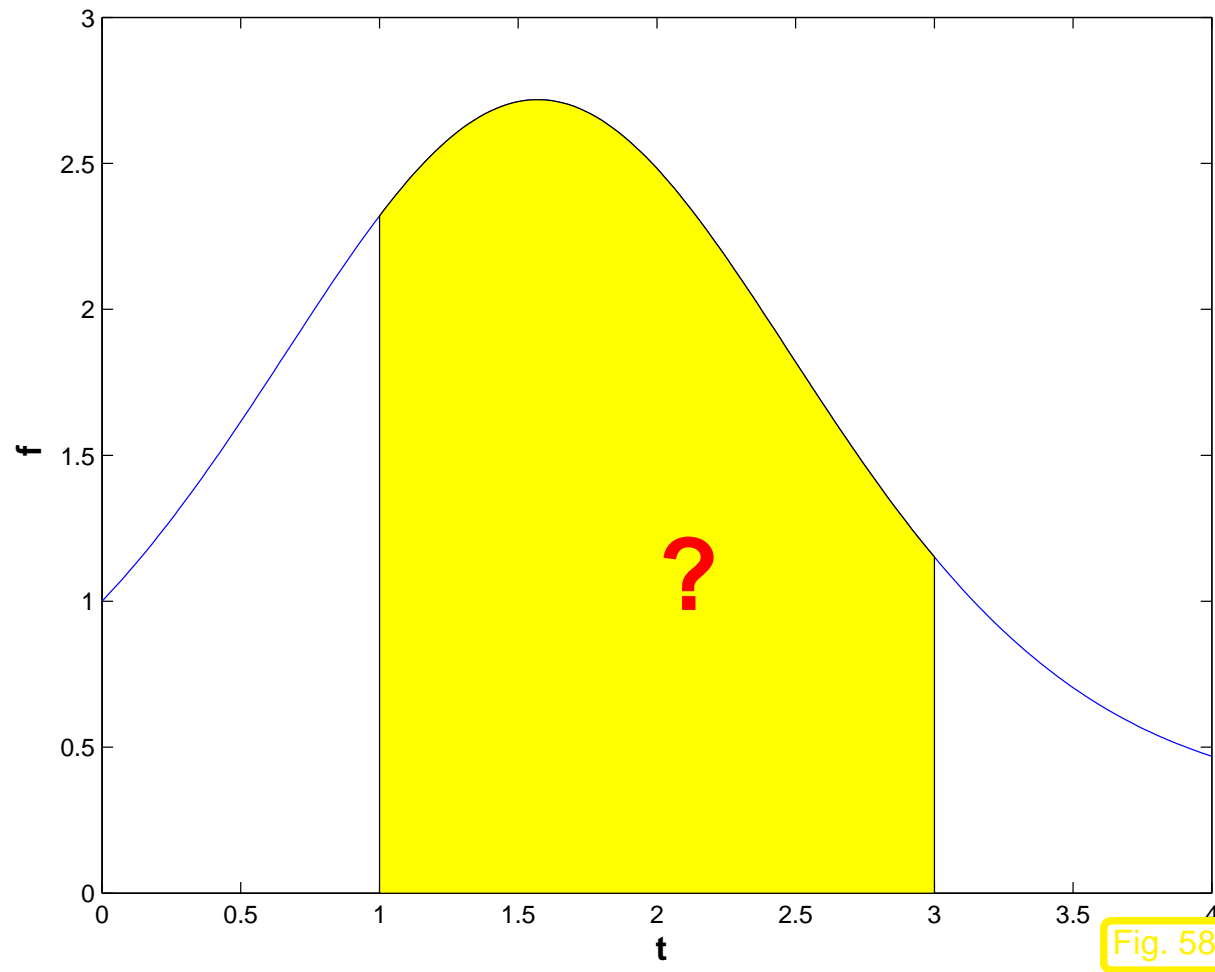
Special case $d = 1$: $\Omega = [a, b]$ (interval)

☞ Numerical quadrature methods are key building blocks for methods for the numerical treatment of differential equations.

Gradinaru
D-MATH

Remark 7.0.1 (Importance of numerical quadrature).

☞ Numerical quadrature methods are key building blocks for methods for the numerical treatment of partial differential equations (➤ Course “Numerical treatment of partial differential equations”) 



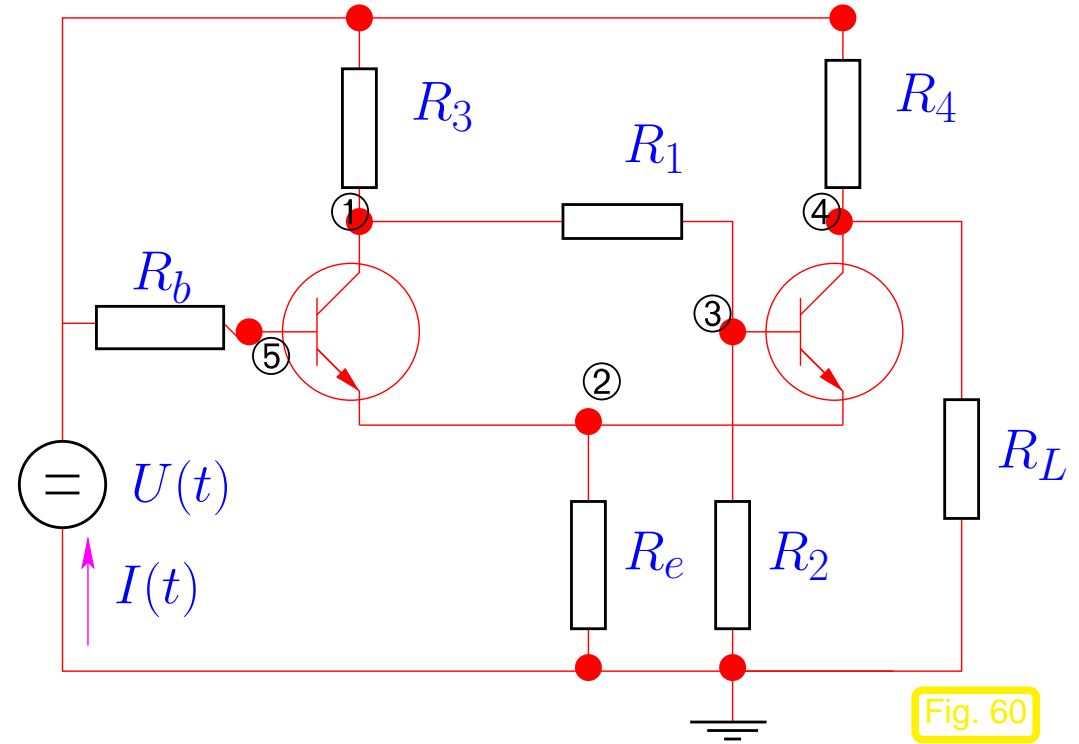
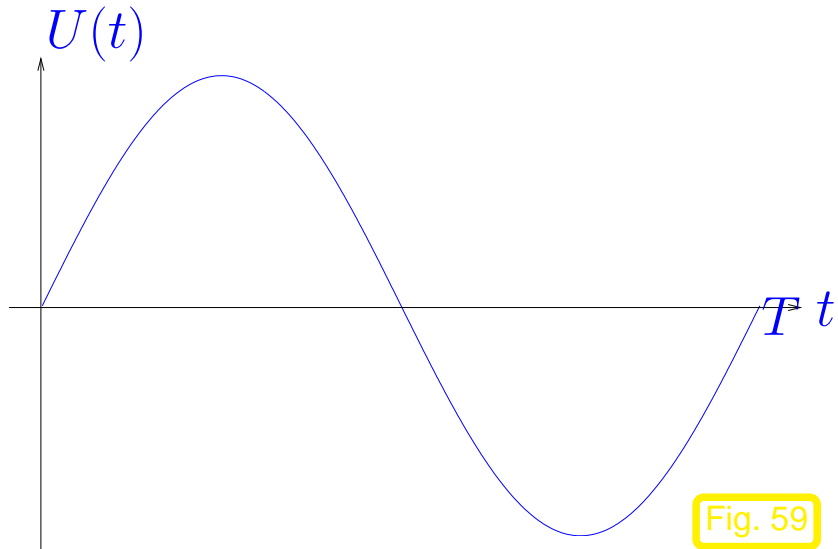
Numerical quadrature methods

approximate

$$\int_a^b f(t) dt$$

Example 7.0.2 (Heating production in electrical circuits).

Time-harmonic excitation:



Integrating power $P = UI$ over period $[0, T]$ yields heat production per period:

$$W_{\text{therm}} = \int_0^T U(t)I(t) dt, \quad \text{where } I = I(U).$$

function $I = \text{current}(U)$ involves solving non-linear system of equations!



7.1 Quadrature Formulas

n -point **quadrature formula** on $[a, b]$:
(n -point quadrature rule)

$$\int_a^b f(t) \, dt \approx Q_n(f) := \sum_{j=1}^n w_j^n f(c_j^n) . \quad (7.1.1)$$

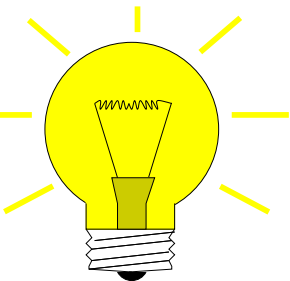
$$\begin{aligned} w_j^n &: \text{quadrature weights} \in \mathbb{R} & (\text{ger.: Quadraturgewichte}) \\ c_j^n &: \text{quadrature nodes} \in [a, b] & (\text{ger.: Quadraturknoten}) \end{aligned}$$

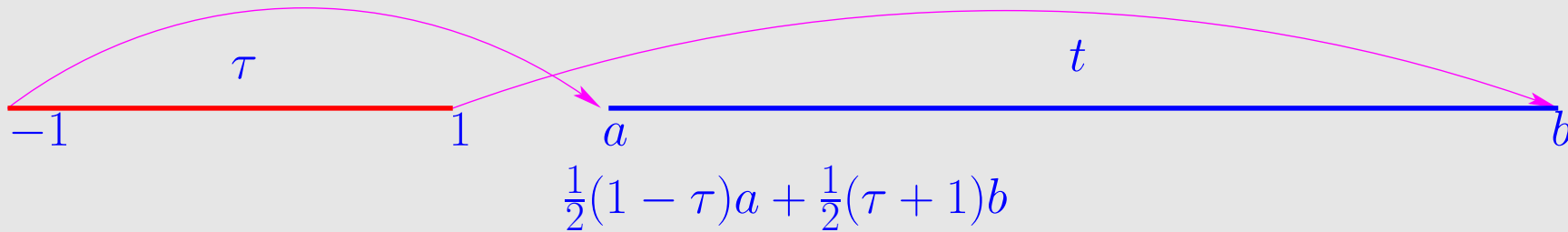
Remark 7.1.1 (Transformation of quadrature rules).

Given: quadrature formula $(\hat{c}_j, \hat{w}_j)_{j=1}^n$ on **reference interval** $[-1, 1]$

Idea: transformation formula for integrals

$$\int_a^b f(t) \, dt = \frac{1}{2}(b-a) \int_{-1}^1 \hat{f}(\tau) \, d\tau , \quad \hat{f}(\tau) := f\left(\frac{1}{2}(1-\tau)a + \frac{1}{2}(\tau+1)b\right) . \quad (7.1.2)$$





► quadrature formula for general interval $[a, b]$, $a, b \in \mathbb{R}$:

$$\int_a^b f(t) dt \approx \frac{1}{2}(b-a) \sum_{j=1}^n \hat{w}_j \hat{f}(\hat{c}_j) = \sum_{j=1}^n w_j f(c_j) \quad \text{with} \quad \begin{aligned} c_j &= \frac{1}{2}(1 - \hat{c}_j)a + \frac{1}{2}(1 + \hat{c}_j)b, \\ w_j &= \frac{1}{2}(b-a)\hat{w}_j. \end{aligned}$$

► A 1D quadrature formula on arbitrary intervals can be specified by providing its weights \hat{w}_j /nodes \hat{c}_j for integration domain $[-1, 1]$. Then the above transformation is assumed.

Other common choice of reference interval: $[0, 1]$



quadrature error

$$E(n) := \left| \int_a^b f(t) \, dt - Q_n(f) \right|$$

Given families of quadrature rules $\{Q_n\}_n$ with 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

should be aware of the *asymptotic* behavior of quadrature error $E(n)$ for $n \rightarrow \infty$

▷ algebraic convergence $E(n) = O(n^{-p}), p > 0$

Qualitative distinction:

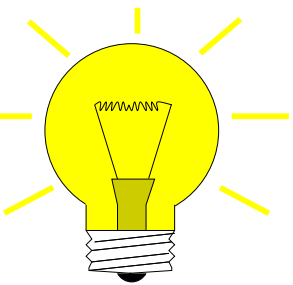
▷ exponential convergence $E(n) = O(q^n), 0 \leq q < 1$

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 $Q_n(f)$.

Gradinaru
D-MATH

Therefore we consider the quadrature error as a function of n .

Idea: **Equidistant** quadrature nodes $t_j := a + hj, h := \frac{b-a}{n}, j = 0, \dots, n$: choose the n weights such that the error $E(n) = 0$ for all polynomials f of degree $n - 1$.

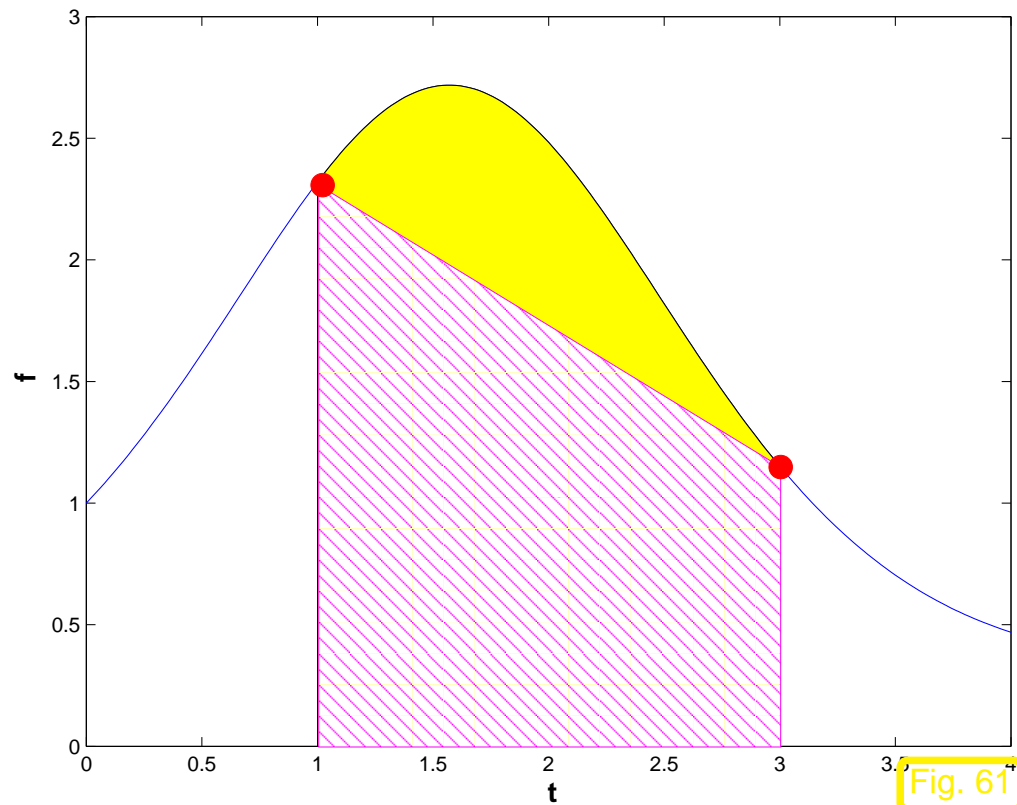


Example 7.1.2 (Newton-Cotes formulas).

- $n = 1$: Trapezoidal rule

$$\hat{Q}_{\text{trp}}(f) := \frac{1}{2} (f(0) + f(1)) \quad (7.1.3)$$

$$\left(\int_a^b f(t) \, dt \approx \frac{b-a}{2} (f(a) + f(b)) \right)$$



- $n = 2$: **Simpson rule**

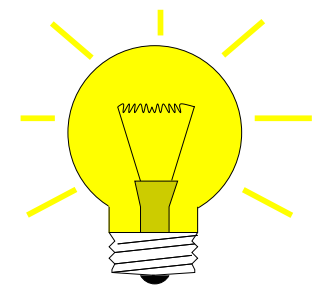
$$\frac{h}{6} \left(f(0) + 4 f\left(\frac{1}{2}\right) + f(1) \right) \quad \left(\int_a^b f(t) dt \approx \frac{b-a}{6} \left(f(a) + 4 f\left(\frac{a+b}{2}\right) + f(b) \right) \right) \quad (7.1.4)$$



Remark 7.1.3 (Error estimates for polynomial quadrature).

Quadrature error estimates directly from L^∞ -interpolation error estimates for Lagrangian interpolation with polynomial of degree $n - 1$:

$$f \in C^n([a, b]) \Rightarrow \left| \int_a^b f(t) dt - Q_n(f) \right| \leq \frac{1}{n!} (b - a)^{n+1} \|f^{(n)}\|_{L^\infty([a, b])} . \quad (7.1.5)$$



Idea: **Gaussian** quadrature: Choose the n weights and the n points such that the error $E(n) = 0$ for all polynomials f of degree $2n - 1$.

Example 7.1.4 (2-point quadrature rule of order 4).

Necessary & sufficient conditions for order 4 (first wrong integral is $\int_a^b x^4 dx$):

$$Q_n(p) = \int_a^b p(t) dt \quad \forall p \in \mathcal{P}_3 \quad \Leftrightarrow \quad Q_n(t^q) = \frac{1}{q+1}(b^{q+1} - a^{q+1}), \quad q = 0, 1, 2, 3.$$



4 equations for weights w_j and nodes c_j , $j = 1, 2$ ($a = -1, b = 1$), cf. Rem. ??

$$\begin{aligned} \int_{-1}^1 1 dt = 2 &= 1w_1 + 1w_2, & \int_{-1}^1 t dt = 0 &= c_1w_1 + c_2w_2 \\ \int_{-1}^1 t^2 dt = \frac{2}{3} &= c_1^2w_1 + c_2^2w_2, & \int_{-1}^1 t^3 dt = 0 &= c_1^3w_1 + c_2^3w_2. \end{aligned}$$

(7.1.6)

Gradinaru
D-MATH

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: $\left\{ w_2 = 1, w_1 = 1, c_1 = 1/3 \sqrt{3}, c_2 = -1/3 \sqrt{3} \right\}$

► quadrature formula: $\int_{-1}^1 f(x) dx \approx f\left(\frac{1}{\sqrt{3}}\right) + f\left(-\frac{1}{\sqrt{3}}\right)$ (7.1.7)



Remark 7.1.5 (Computing Gauss nodes and weights).

Code 7.1.6: Golub-Welsch algorithm

```

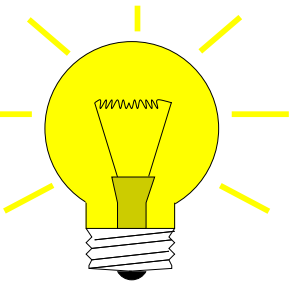
1 from numpy import zeros, diag, sqrt, size
2 from numpy.linalg import eigh
3
4 def gaussquad(n):
5     b = zeros(n-1);
6     for i in xrange(size(b)):
7         b[i] = (i+1)/sqrt(4*(i+1)*(i+1)-1)
8     J = diag(b, -1) + diag(b, 1)
9     x, ev = eigh(J); w = 2*ev[0]*ev[0]
10    return (x, w)

```

Compute nodes/weights of
Gaussian quadrature by solving
an eigenvalue problem!
(**Golub-Welsch algorithm** [18,
Sect. 3.5.4])

In codes: c_j, w_j from tables!





$$\int_{-1}^1 f(x) dx = \int_0^\pi f(\cos \theta) \sin \theta d\theta = \sum_{\text{even } k} \frac{2a_k}{1-k^2} \quad (7.1.8)$$

with a_k the Fourier coefficients of $F(\theta) = f(\cos \theta) = \sum_{k=0}^{\infty} a_k \cos(k\theta)$.

Advantage for the Clenshaw-Curtis is the speed and stability of the fast Fourier transform.

Code 7.1.7: Clenshaw-Curtis: direct implementation

```

1 def cc2(func, a, b, N):
2     """
3     Clenshaw-Curtis quadrature rule
4     by FFT with the function values
5     """
6     bma = 0.5*(b-a)
7     x = np.cos(np.pi * np.linspace(0, N, N+1)/N) # Chebyshev points
8     x *= bma
9     x += 0.5*(a+b)
10    fx = func(x)*0.5/N
11    vx = np.hstack((fx, fx[-2:0:-1]))
12    g = np.real(np.fft.fft(vx))
13    A = np.zeros(N+1)
14    A[0] = g[0]; A[N] = g[N]
15    A[1:N] = g[1:N] + np.flipud(g[N+1:])

```

```
16 w = 0.*x
17 w[::2] = 2./(1.-np.r_[ :N+1:2]**2)
18 return np.dot(w,A)*bma
```

Code 7.1.8: Clenshaw-Curtis: weights and points

```
1 def cc1(func, a, b, N):
2     """
3     Clenshaw-Curtis_quadrature_rule
4     by_constructing_the_points_and_the_weights
5     """
6     bma = b-a
7     c = np.zeros([2,2*N-2])
8     c[0][0] = 2.0
9     c[1][1] = 1
10    c[1][-1]= 1
11    for i in np.arange(2.,N,2):
12        val = 2.0/(1-pow(i,2))
13        c[0][i] = val
14        c[0][2*N-2-i] = val
15
16    f = np.real(np.fft.ifft(c))
17    w = f[0][:N]; w[0] *= 0.5; w[-1] *= 0.5 # weights
```

```

8 x = 0.5*((b+a)+(N-1)*bma*f[1][:N]) # points
9 return np.dot(w,func(x))*bma

```

Example 7.1.9 (Error of (non-composite) quadratures).

Code 7.1.10: important polynomial quadrature rules

```

1 from gaussquad import gaussquad
2 from numpy import *
3
4 def numquad(f,a,b,N,mode='equidistant'):
5     """ Numerical_quadrature_on_[a,b]_by_polynomial_quadrature_
6         formula
7         _____f_>_function_to_be_integrated_(handle)
8         _____a,b_>_integration_interval_[a,b]_(endpoints_included)
9         _____N_>_Maximal_degree_of_polynomial
10        _____mode_(equidistant,_Chebychev,_Clenshaw-Curtis,_Gauss)_selects_
11        quadrature_rule
12        """
13        # use a dictionary as "switch" statement:
14        quadrule = { 'gauss':quad_gauss, 'equidistant':quad_equidistant,
15                    'chebychev':quad_chebychev}
16        nvals = range(1,N+1); res = []
17        try:

```

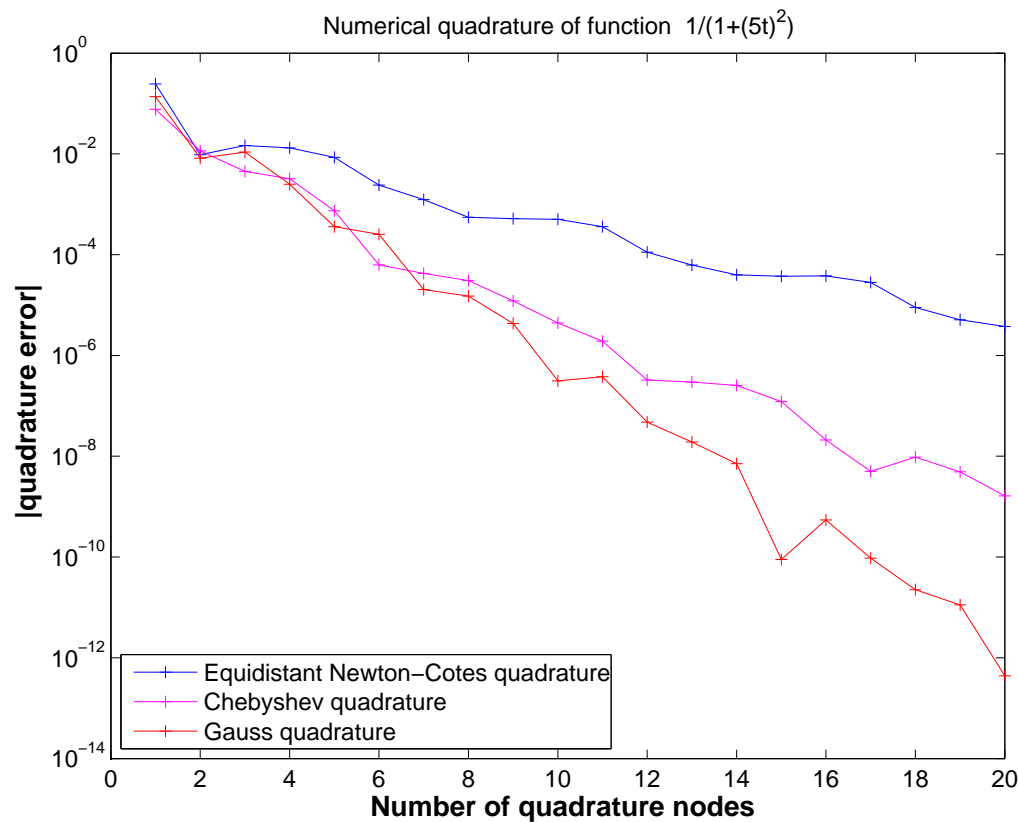
```
15         for n in nvals:
16             res.append(quadrule[mode.lower()](f,a,b,n))
17     except KeyError:
18         print "invalid_quadrature_type!"
19     else:
20         return (nvals,res)
```

```
21
22 def quad_gauss(f,a,b,deg):
23     # get Gauss points for [-1,1]
24     [gx,w] = gaussquad(deg);
25     # transform to [a,b]
26     x = 0.5*(b-a)*gx+0.5*(a+b)
27     y = f(x)
28     return 0.5*(b-a)*dot(w,y)
```

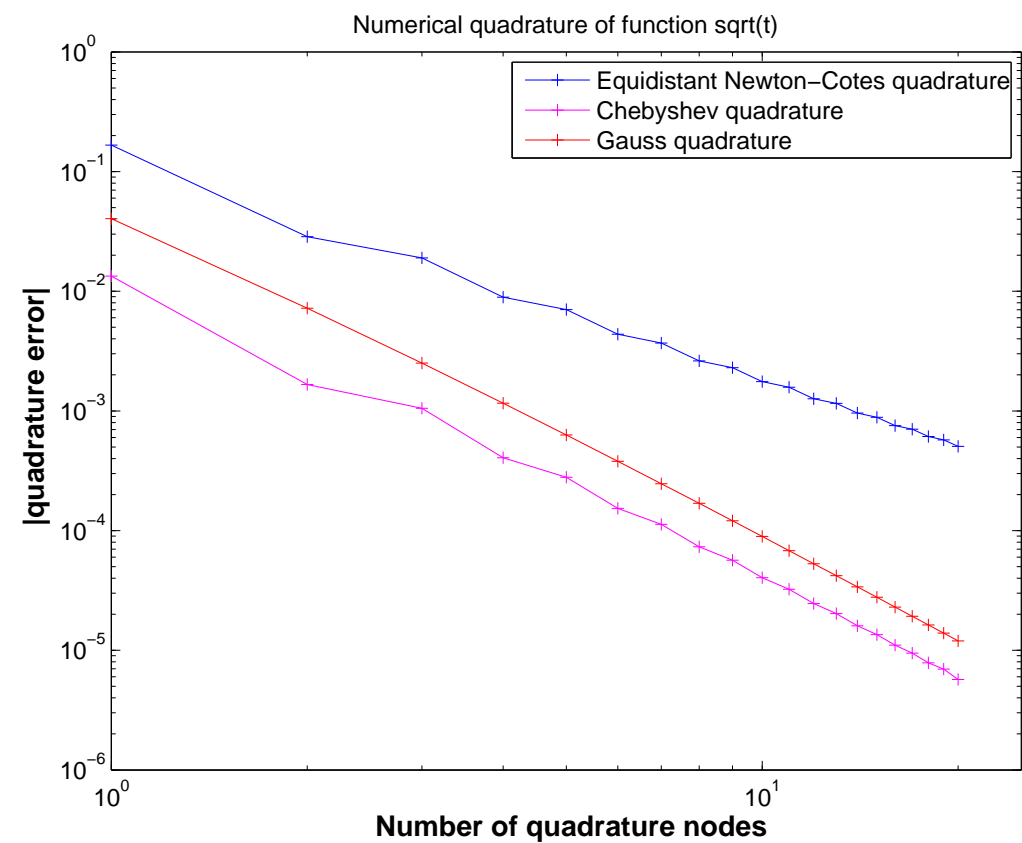
```
29
30 def quad_equidistant(f,a,b,deg):
31     p = arange(deg+1.0,0.0,-1.0)
32     w = (power(b,p) - power(a,p))/p
33     x = linspace(a,b,deg+1)
34     y = f(x)
35     # "Quick and dirty" implementation through polynomial interpolation
36     poly = polyfit(x,y,deg)
37     return dot(w,poly)
```



```
38 def quad_chebychev(f , a , b , deg) :  
39     p = arange(deg+1.0, 0.0, -1.0)  
40     w = (power(b,p) - power(a,p)) / p  
41     x = 0.5*(b-a)*cos((arange(0, deg+1)+0.5)/(deg+1)*pi) + 0.5*(a+b) ;  
42     y = f(x)  
43     # "Quick and dirty" implementation through polynomial interpolation  
44     poly = polyfit(x,y,deg)  
45     return dot(w,poly)  
46
```



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^∞ -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 : $\alpha \approx 2.7$

Code 7.1.11: tracking errors on quadrature rules

```
1 from numquad import numquad
2 import matplotlib.pyplot as plt
3 from numpy import *
4
5 def numquaderrs():
6     """Numerical_quadrature_on_[0,1]"""
7     N = 20;
8
9     plt.figure()
10    exact = arctan(5)/5;
11    f = lambda x: 1./(1+power(5.0*x,2))
12    nvals,eqdres = numquad(f,0,1,N,'equidistant')
13    nvals, chbres = numquad(f,0,1,N,'Chebychev')
14    nvals, gaures = numquad(f,0,1,N,'Gauss')
15    plt.semilogy(nvals,abs(eqdres-exact),'b+-',label='Equidistant_\n\nNewton-Cotes_quadrature')
16    plt.semilogy(nvals,abs(chbres-exact),'m+-',label='Clenshaw-Curtis\n\n_quadrature')
```

```
plt.semilogy(nvals,abs(gaures-exact),'r+-',label='Gauss_
quadrature')
plt.title('Numerical_quadrature_of_function_1/(1+(5t)^2)')
plt.xlabel('f_Number_of_quadrature_nodes')
plt.ylabel('f_|quadrature_error|')
plt.legend(loc="lower_left")
plt.show()
```

```
# eqdp1 = polyfit(nvals,log(abs(eqdres-exact)),1)
# chbp1 = polyfit(nvals,log(abs(chbres-exact)),1)
# gaup1 = polyfit(nvals,log(abs(gaures-exact)),1)
# plt.savefig("../PICTURES/numquaderr1.eps")
```

```
plt.figure()
exact = array(2./3.);
f = lambda x: sqrt(x)
nvals,eqdres = numquad(f,0,1,N,'equidistant')
nvals,chbres = numquad(f,0,1,N,'Chebychev')
nvals,gaures = numquad(f,0,1,N,'Gauss')
plt.loglog(nvals,abs(eqdres-exact),'b+-',label='Equidistant_
Newton-Cotes_quadrature')
plt.loglog(nvals,abs(chbres-exact),'m+-',label='Clenshaw-Curtis_
quadrature')
plt.loglog(nvals,abs(gaures-exact),'r+-',label='Gauss_
```

```
37     quadrature ' )
38     plt.axis([1, 25, 0.000001, 1]);
39     plt.title('Numerical_quadrature_of_function_sqrt(t)')
40     plt.xlabel('f_Number_of_quadrature_nodes')
41     plt.ylabel('f_|quadrature_error|')
42     plt.legend(loc="lower_left")
43     plt.show()
44 # eqdp1 = polyfit(nvals,log(abs(eqdpres-exact)),1)
45 # chbp1 = polyfit(nvals,log(abs(chbpres-exact)),1)
46 # gaup1 = polyfit(nvals,log(abs(gaupres-exact)),1)
47 # plt.savefig("../PICTURES/numquaderr2.eps")
48 if "__name__" == "__main__":
49     numquaderrs()
```

Equal spacing is a disaster for high-order interpolation and integration !

► Divide the integration domain in small pieces and use low-order rule on each piece (composite quadrature)

► Take into account the eventual non-smoothness of f when dividing the integration domain

7.2 Composite Quadrature

With $a = x_0 < x_1 < \cdots < x_{m-1} < x_m = b$

$$\int_a^b f(t) \, dt = \sum_{j=1}^m \int_{x_{j-1}}^{x_j} f(t) \, dt . \quad (7.2.1)$$

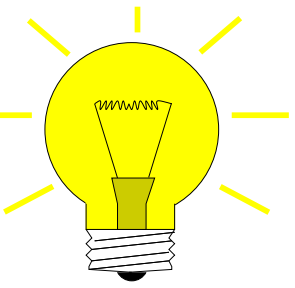
Recall (7.1.5): for polynomial quadrature rule and $f \in C^n([a, b])$ quadrature error shrinks with $n + 1$ st power of length of integration interval.



Reduction of quadrature error can be achieved by

- splitting of the integration interval according to (7.2.1),
- using the intended quadrature formula on each sub-interval $[x_{j-1}, x_j]$.

Note: Increase in total no. of f -evaluations incurred, which has to be balanced with the gain in accuracy to achieve optimal efficiency,

- Idea:  Partition integration domain $[a, b]$ by **mesh** (grid) $\mathcal{M} := \{a = x_0 < x_1 < \dots < x_m = b\}$
- Apply quadrature formulas on sub-intervals $I_j := [x_{j-1}, x_j]$, $j = 1, \dots, m$, and sum up.



composite quadrature rule

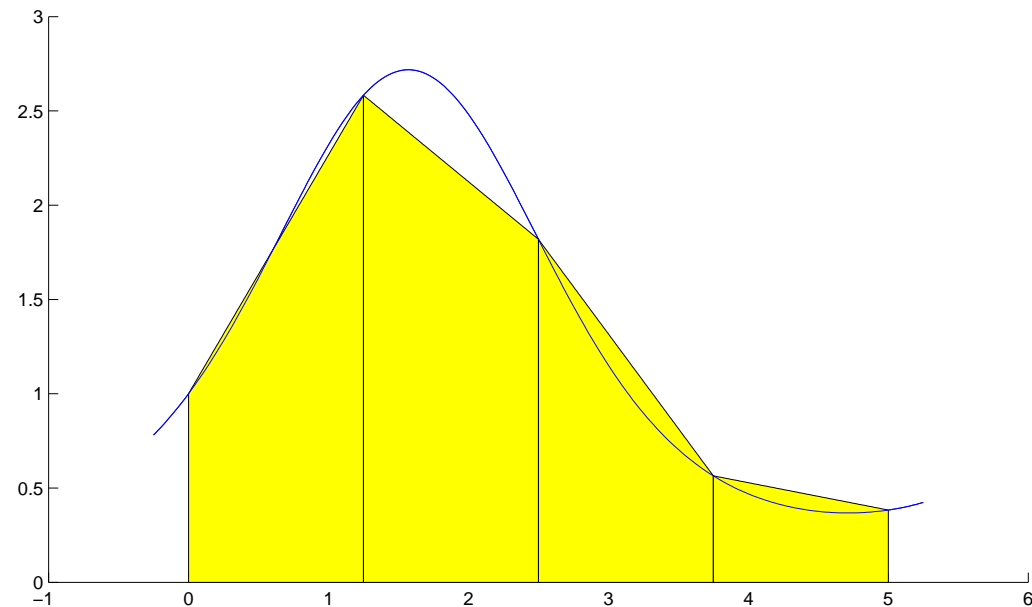
Note: Here we only consider one and the same quadrature formula (local quadrature formula) applied on all sub-intervals.

Example 7.2.1 (Simple composite polynomial quadrature rules).

Composite trapezoidal rule, *cf.* (8.6.2)

$$\int_a^b f(t) dt = \frac{1}{2}(x_1 - x_0)f(a) + \sum_{j=1}^{m-1} \frac{1}{2}(x_{j+1} - x_{j-1})f(x_j) + \frac{1}{2}(x_m - x_{m-1})f(b) .$$

(7.2.2)



Composite Simpson rule, cf. (7.1.4)

$$\int_a^b f(t) dt =$$

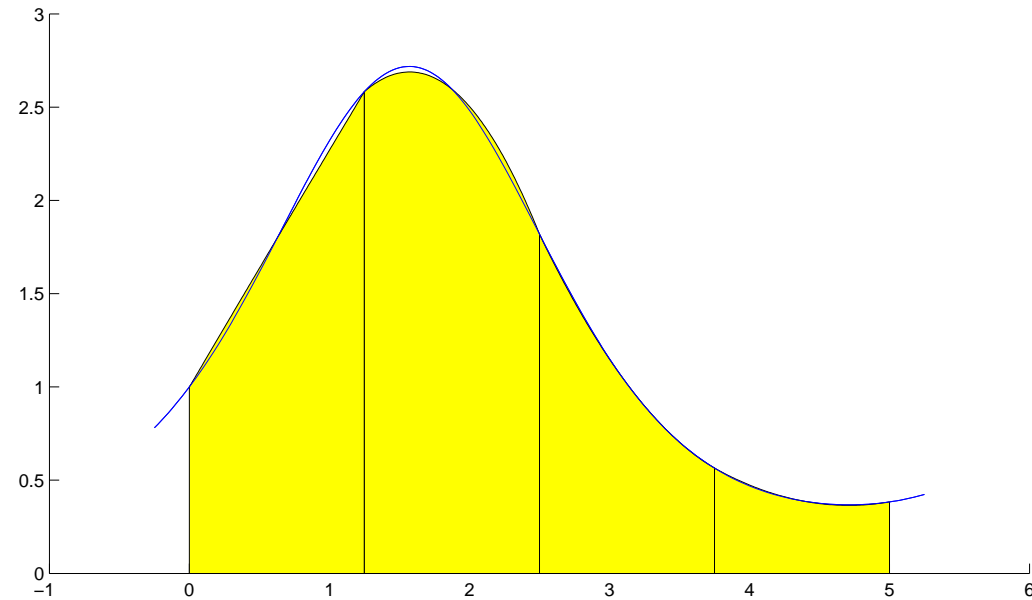
$$\frac{1}{6}(x_1 - x_0)f(a) +$$

$$\sum_{j=1}^{m-1} \frac{1}{6}(x_{j+1} - x_{j-1})f(x_j) +$$

$$\sum_{j=1}^m \frac{2}{3}(x_j - x_{j-1})f\left(\frac{1}{2}(x_j + x_{j-1})\right) +$$

$$\frac{1}{6}(x_m - x_{m-1})f(b) .$$

(7.2.3)



Formulas (7.2.2), (7.2.3) directly suggest efficient implementation with minimal number of f -evaluations.



Focus: *asymptotic* behavior of quadrature error for

$$\text{mesh width} \quad h := \max_{j=1, \dots, m} |x_j - x_{j-1}| \rightarrow 0$$

For *fixed* local n -point quadrature rule: $O(mn)$ f -evaluations for composite quadrature (“total cost”)

➤ If mesh equidistant ($|x_j - x_{j-1}| = h$ for all j), then total cost for composite numerical quadrature $= O(h^{-1})$.

Theorem 7.2.1 (Convergence of composite quadrature formulas).

For a composite quadrature formula Q based on a local quadrature formula of order $p \in \mathbb{N}$ holds

$$\exists C > 0: \left| \int_I f(t) dt - Q(f) \right| \leq Ch^p \|f^{(p)}\|_{L^\infty(I)} \quad \forall f \in C^p(I), \forall \mathcal{M}.$$

Proof. Apply interpolation error estimate .



Example 7.2.2 (Quadrature errors for composite quadrature rules).

Composite quadrature rules based on

- trapezoidal rule (8.6.2) \supset local order 2 (exact for linear functions),
- Simpson rule (7.1.4) \supset local order 3 (exact for quadratic polynomials)

on equidistant mesh $\mathcal{M} := \{jh\}_{j=0}^n$, $h = 1/n$, $n \in \mathbb{N}$.

Code 7.2.3: composite trapezoidal rule (7.2.2)

```

1 def trapezoidal(func,a,b,N):
2     """
3     Numerical_quadrature_based_on_trapezoidal_rule
4     func:_handle_to_y_=f(x)
5     a,b:_bounds_of_integration_interval
6     N+1:_number_of_equidistant_quadrature_points
7     """
8
9     from numpy import linspace , sum
10    # quadrature nodes
11    x = linspace(a,b,N+1); h = x[1]-x[0]
12    # quadrature weights: internal nodes: w=1, boundary nodes: w=0.5
13    I = sum(func(x[1:-1])) + 0.5*(func(x[0])+func(x[-1]))
14    return I*h
15
16 if __name__ == "__main__":
17     import matplotlib.pyplot as plt

```

```
18 from scipy import integrate
19 from numpy import array, linspace, size, log, polyfit
20
21 # define a function and an interval:
22 f = lambda x: x**2
23 left = 0.0; right = 1.0
24
25 # exact integration with scipy.integrate.quad:
26 exact, e = integrate.quad(f, left, right)
27 # trapezoid rule for different number of quadrature points
28 N = linspace(2, 101, 100)
29 res = array(N) # preallocate same amount of space as N uses
30 for i in xrange(size(N)):
31     res[i] = trapezoidal(f, left, right, N[i])
32 err = abs(res - exact)
33 #plt.loglog(N, err, 'o')
34 #plt.show()
35
36 # linear fit to determine convergence order
37 p = polyfit(log(N), log(err), 1)
38 # output the convergence order
39 print "convergence_order: ", -p[0]
```

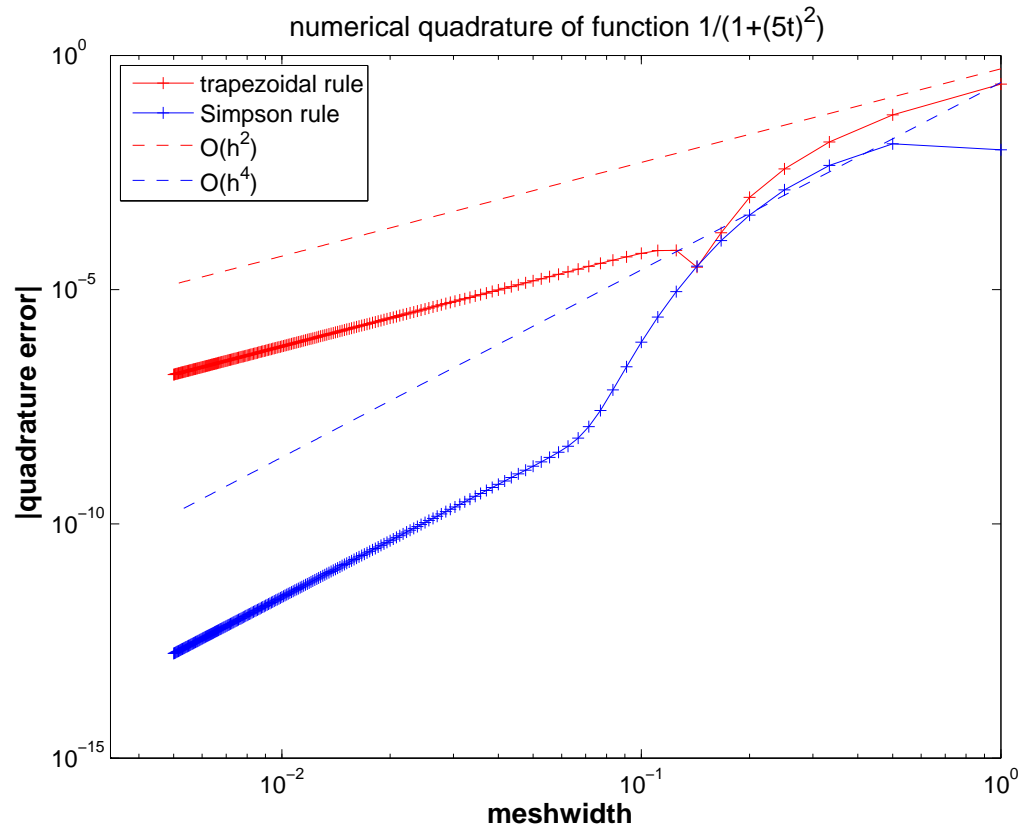
```

1 from numpy import linspace , sum, size
2
3 def simpson(func ,a ,b ,N):
4     """
5     Numerical_quadrature_based_on_Simpson_rule
6     func: _handle_to_y_=f(x)
7     a,b:_bounds_of_integration_interval
8     N+1:_number_of_equidistant_quadrature_points
9     """
10
11     # ensure that we have an even number of subintervals
12     if N%2 == 1: N = N+1
13     # quadrature nodes
14     x = linspace(a,b,N+1); h = x[1]-x[0]
15     # quadrature weights:
16     # internal nodes: even: w=2/3, odd: w=4/3
17     # boundary nodes: w=1/6
18     I = h*sum(func(x[0:-2:2]) + 4*func(x[1:-1:2]) +
19               func(x[2::2]))/3.0
20     return I
21
22 if __name__ == "__main__":
23     import matplotlib.pyplot as plt

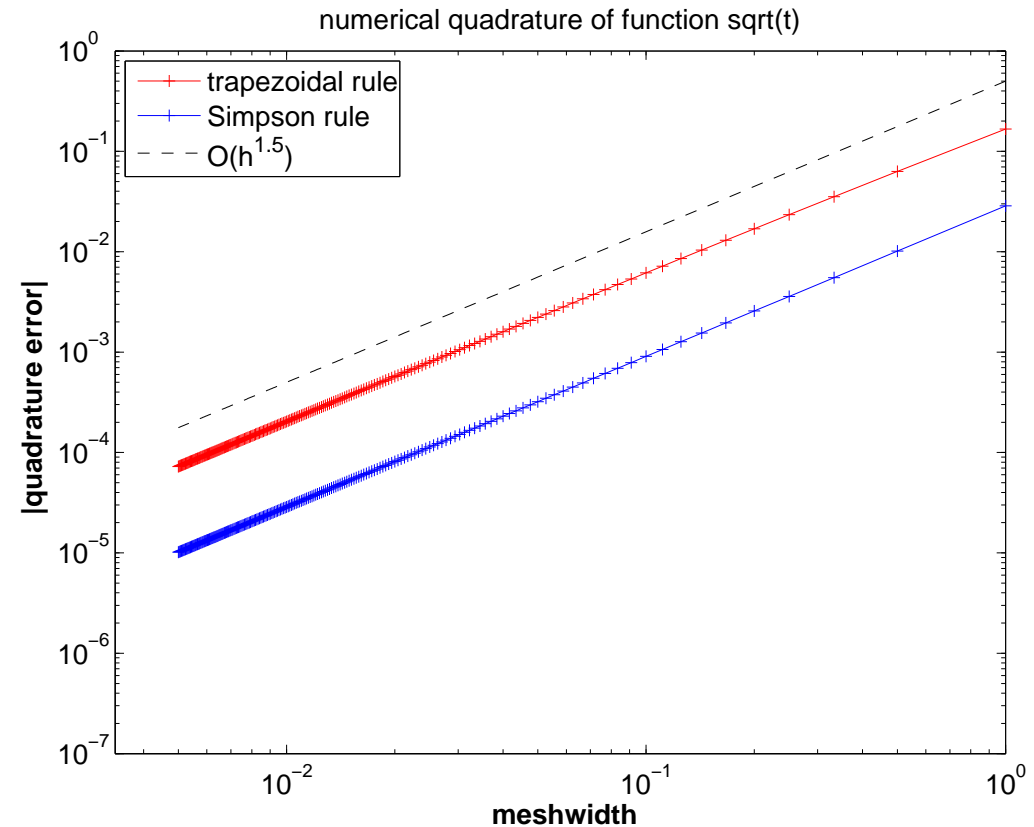
```

```
23 from scipy import integrate
24 from numpy import array, linspace, size, log, polyfit
25
26 # define a function and an interval:
27 f = lambda x: 1./(1+(5*x)**2)
28 left = 0.0; right = 1.0
29
30 # exact integration with scipy.integrate.quad:
31 exact,e = integrate.quad(f, left, right)
32 # trapezoid rule for different number of quadrature points
33 N = linspace(2,101,100)
34 res = array(N) # preallocate same ammount of space as N uses
35 for i in xrange(size(N)):
36     res[i] = simpson(f, left, right, N[i])
37 err = abs(res - exact)
38 #plt.loglog(N,err,'o')
39 #plt.show()
40
41 # linear fit to determine convergence order
42 p = polyfit(log(N[-20:]), log(err[-20:]), 1)
43 # output the convergence order
44 print "convergence_order: ", -p[0]
```

Note: `fnct` is supposed to accept vector arguments and return the function value for each vector component!



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 $E(n) := \left| \int_0^1 f(t) dt - Q_n(f) \right|$ 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 !

Simpson rule: order = 4 ? investigate with 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);
```

$$err := \left(\frac{1}{90} \left(D^{(4)} \right) (f) (0) h^5 + O \left(h^6 \right) \right), h, 6$$

➤ Simpson rule is of order 4, indeed !

Code 7.2.5: errors of composite trapezoidal and Simpson rule

```
1 #!/usr/bin/env python
2
3 import numpy as np
4 import matplotlib.pyplot as plt
```

```
5 from scipy import integrate
6 # own integrators:
7 from trapezoidal import trapezoidal
8 from simpson import simpson
9
10 # integration functions:
11 integrators = [trapezoidal, simpson]
12 intNames = ('trapezoidal', 'simpson')
13
14 # define a few different Ns
15 N = np.linspace(2,201,200)
16
17 # define a function...
18 f = lambda x: 1./(1+(5*x)**2)
19 #f = lambda x: x**2
20 # ...and an interval
21 left=0.0; right=1.0
22
23 # "exact" integration with scipy function:
24 exact,e = integrate.quad(f, left, right, epsabs=1e-12)
25
26 # our versions
27 err = []; res = []
```



```
28 for int in integrators:
29     for i in xrange(np.size(N)):
30         res.append(int(f, left, right, N[i]))
31     err.append(np.abs(np.array(res)-exact))
32     res = []
33
34 plt.figure()
35 # evaluation
36 logN=np.log(N)
37 for i in xrange(np.size(integrators)):
38     # linear fit to determine convergence orders
39     p = np.polyfit(logN[-20:], np.log(err[i][-20:]), 1) # only look at last
40     # 20 entries- asymptotic!
41     # plot errors
42     plt.loglog(N, err[i], 'o', label=intNames[i])
43     # plot linear fitting
44     x = np.linspace(min(logN), max(logN), 10)
45     y = np.polyval(p, x)
46     plt.loglog(np.exp(x), np.exp(y), label="linear_fit:_
47     m="+str(-p[0])[:4])
48     # output the convergence order
49     print "convergence_order_of_"+intNames[i]+":",-p[0]

plt.xlabel("log(N)"); plt.ylabel("log(err)")
```

```
50 plt.legend(loc="lower_left")  
51 plt.show()
```



Remark 7.2.6 (Removing a singularity by transformation).

Ex. 7.2.2 ➤ lack of smoothness of integrand limits rate of algebraic convergence of composite quadrature rule for meshwidth $h \rightarrow 0$.

Idea: recover integral with smooth integrand by “analytic preprocessing”

Here is an example:

For $f \in C^\infty([0, b])$ compute $\int_0^b \sqrt{t} f(t) dt$ via quadrature rule (\rightarrow Ex. 7.2.2)

substitution $s = \sqrt{t}$:
$$\int_0^b \sqrt{t} f(t) dt = \int_0^{\sqrt{b}} \boxed{2s^2 f(s^2)} ds . \quad (7.2.4)$$

Apply quadrature rule to smooth integrand



Then:

Example 7.2.7 (Convergence of equidistant trapezoidal rule).

Sometimes there are surprises: convergence of a composite quadrature rule is much better than predicted by the order of the local quadrature formula:

Equidistant trapezoidal rule (order 2), see (7.2.2)

$$\int_a^b f(t) dt \approx T_m(f) := h \left(\frac{1}{2} f(a) + \sum_{k=1}^{m-1} f(kh) + \frac{1}{2} f(b) \right), \quad h := \frac{b-a}{m}. \quad (7.2.5)$$

Code 7.2.8: equidistant trapezoidal quadrature formula

```
1 def trapezoidal(func, a, b, N):
2     """
3     Numerical quadrature based on trapezoidal rule
4     func: handle to y = f(x)
5     a, b: bounds of integration interval
```

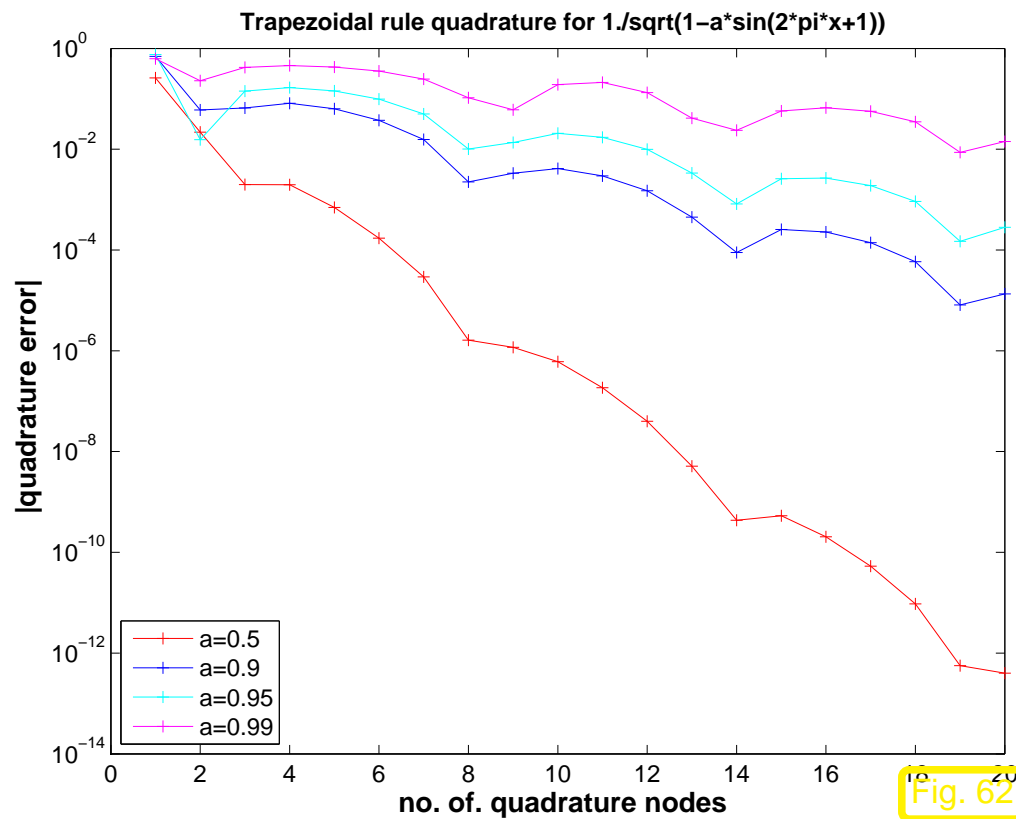
```
6  _____N+1:_number_of_equidistant_quadrature_points
7  _____" " "
8  _____
9      from numpy import linspace , sum
10     # quadrature nodes
11     x = linspace(a,b,N+1); h = x[1]-x[0]
12     # quadrature weights: internal nodes: w=1, boundary nodes: w=0.5
13     I = sum(func(x[1:-1])) + 0.5*(func(x[0])+func(x[-1]))
14     return I*h
15
16 if __name__ == "__main__":
17     import matplotlib.pyplot as plt
18     from scipy import integrate
19     from numpy import array, linspace , size , log , polyfit
20
21     # define a function and an interval:
22     f = lambda x: x**2
23     left = 0.0; right = 1.0
24
25     # exact integration with scipy.integrate.quad:
26     exact,e = integrate.quad(f , left , right )
27     # trapezoid rule for different number of quadrature points
28     N = linspace(2,101,100)
29     res = array(N) # preallocate same amount of space as N uses
```

```
30 for i in xrange(size(N)):
31     res[i] = trapezoidal(f, left, right, N[i])
32 err = abs(res - exact)
33 #plt.loglog(N, err, 'o')
34 #plt.show()
35
36 # linear fit to determine convergence order
37 p = polyfit(log(N), log(err), 1)
38 # output the convergence order
39 print "convergence_order: ", -p[0]
```

1-periodic smooth (**analytic**) integrand

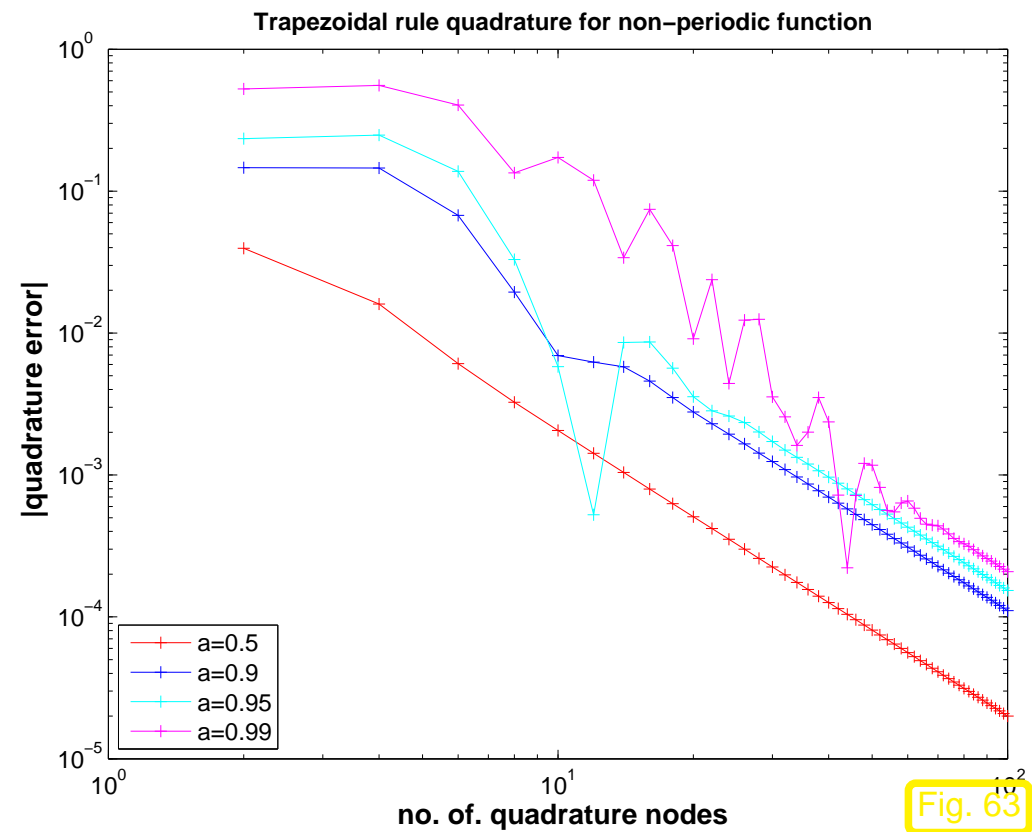
$$f(t) = \frac{1}{\sqrt{1 - a \sin(2\pi t - 1)}}, \quad 0 < a < 1.$$

("exact value of integral": use T_{500})



quadrature error for $T_n(f)$ on $[0, 1]$

exponential convergence !!



quadrature error for $T_n(f)$ on $[0, \frac{1}{2}]$

merely algebraic convergence

Code 7.2.9: tracking error of equidistant trapezoidal quadrature formula

```

1
2 import matplotlib.pyplot as plt
3 from scipy import integrate
4 from trapezoidal import trapezoidal
5 from numpy import *
```

```
6
7 # define the function: (0 < a < 1)
8 f = lambda x: 1./sqrt(1-a*sin(2*pi*x+1))
9 avals = [0.5, 0.9, 0.95, 0.99];
10
11 left = 0
12 # first interval: [0, 0.5]
13 err = []; right = 0.5
14 # loop over different values of a:
15 for a in avals:
16     # exact integration with scipy.integrate.quad:
17     exact,e = integrate.quad(f, left, right)
18     # trapezoid rule for different number of quadrature points
19     N = linspace(2,50,49)
20     res = array(N) # preallocate same amount of space as N uses
21     for i in xrange(size(N)):
22         res[i] = trapezoidal(f, left, right, N[i])
23     err.append( abs(res - exact) )
24 plt.figure()
25 plt.xlabel('N')
26 plt.ylabel('error')
27 plt.title(u'Trapezoidal_rule_quadrature_for_non-periodic_function')
28 for i in xrange(size(avals)): plt.loglog(N, err[i], '-o', label='a=_'
```

```
29 '+str( avals[ i ]))
30 plt.legend( loc="lower_left" )
31 plt.show()
32 #plt.savefig("../PICTURES/traperr2.eps")
33 # second interval: [0, 1]
34 err = []; right = 1
35 # loop over different values of a:
36 for a in avals:
37     # exact integration with scipy.integrate.quad:
38     exact, e = integrate.quad( f, left, right )
39     # trapezoid rule for different number of quadrature points
40     N = linspace( 2, 50, 49 )
41     res = array( N ) # preallocate same amount of space as N uses
42     for i in xrange( size( N ) ):
43         res[ i ] = trapezoidal( f, left, right, N[ i ] )
44     err.append( abs( res - exact ) )
45 plt.figure()
46 plt.xlabel( 'N' )
47 plt.ylabel( 'error' )
48 plt.title( r" Trapezoidal_rule_quadrature_for_  $(1 - a \cdot \sin(2\pi x + 1))^{\frac{-1}{2}}$  " )
49 for i in xrange( size( avals ) ): plt.loglog( N, err[ i ], '—o', label= 'a=_'
50 '+str( avals[ i ]))
```



```

50 plt.legend(loc="lower_left")
51 plt.show()
52 #plt.savefig("../PICTURES/traperr1.eps")

```

Explanation:

$$f(t) = e^{2\pi i k t} \quad \blacktriangleright \quad \begin{cases} \int_0^1 f(t) dt = \begin{cases} 0 & , \text{ if } k \neq 0 , \\ 1 & , \text{ if } k = 0 . \end{cases} \\ T_m(f) = \frac{1}{m} \sum_{l=0}^{m-1} e^{\frac{2\pi i}{m} l k} \stackrel{(6.1.2)}{=} \begin{cases} 0 & , \text{ if } k \notin m\mathbb{Z} , \\ 1 & , \text{ if } k \in m\mathbb{Z} . \end{cases} \end{cases}$$

Equidistant trapezoidal rule T_m is exact for trigonometric polynomials of degree $< 2m$!

It takes sophisticated tools from complex analysis to conclude exponential convergence for analytic integrands from the above observation.

7.3 Adaptive Quadrature

Example 7.3.1 (Rationale for adaptive quadrature).

Consider composite trapezoidal rule (7.2.2) on mesh $\mathcal{M} := \{a = x_0 < x_1 < \cdots < x_m = b\}$:

Local quadrature error (for $f \in C^2([a, b])$):

$$\int_{x_{k-1}}^{x_k} f(t) dt - \frac{1}{2}(f(x_{k-1}) + f(x_k)) \leq (x_k - x_{k-1})^3 \|f''\|_{L^\infty([x_{k-1}, x_k])}.$$

➤ Do not use equidistant mesh !

Refine \mathcal{M} , where $|f''|$ large !

Makes sense, e.g., for “spike function”

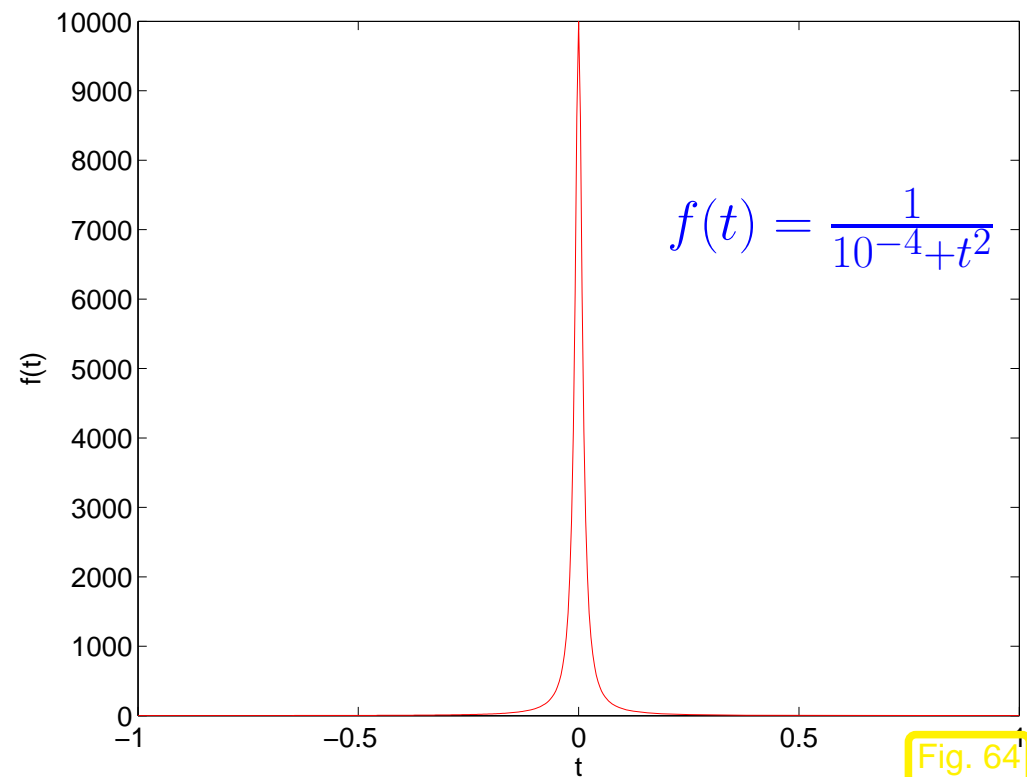


Fig. 64

Goal: *Equilibrate error contributions of all mesh intervals*

Tool: Local **a posteriori error estimation**
(Estimate contributions of mesh intervals from intermediate results)

Policy: **Local mesh refinement**

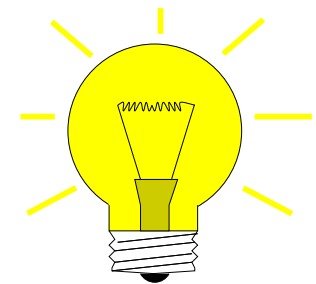
► *Adaptive multigrid quadrature* → [14, Sect. 9.7]

Idea: local error estimation by comparing local results of two quadrature formulas

Q_1, Q_2 of *different* order → local error estimates

heuristics: $\text{error}(Q_2) \ll \text{error}(Q_1) \Rightarrow \text{error}(Q_1) \approx Q_2(f) - Q_1(f)$.

Now: $Q_1 = \text{trapezoidal rule (order 2)} \leftrightarrow Q_2 = \text{Simpson rule (order 4)}$



Given: mesh $\mathcal{M} := \{a = x_0 < x_1 < \dots < x_m = b\}$

❶ (error estimation)

For $I_k = [x_{k-1}, x_k]$, $k = 1, \dots, m$ (midpoints $p_k := \frac{1}{2}(x_{k-1} + x_k)$)

$$\text{EST}_k := \underbrace{\left| \frac{h_k}{6} (f(x_{k-1}) + 4f(p_k) + f(x_k)) \right|}_{\text{Simpson rule}} - \underbrace{\left| \frac{h_k}{4} (f(x_{k-1}) + 2f(p_k) + f(x_k)) \right|}_{\text{trapezoidal rule on split mesh interval}}. \quad (7.3.1)$$

❷ (Termination)

Simpson rule on \mathcal{M} \Rightarrow preliminary result I

$$\text{If } \sum_{k=1}^m \text{EST}_k \leq \text{RTOL} \cdot I \quad (\text{RTOL} := \text{prescribed tolerance}) \Rightarrow \mathbf{STOP} \quad (7.3.2)$$

❸ (local mesh refinement)

$$\mathcal{S} := \{k \in \{1, \dots, m\} : \text{EST}_k \geq \eta \cdot \frac{1}{m} \sum_{j=1}^m \text{EST}_j\}, \quad \eta \approx 0.9. \quad (7.3.3)$$

► new mesh: $\mathcal{M}^* := \mathcal{M} \cup \{p_k : k \in \mathcal{S}\}.$

Then continue with step ❶ and mesh $\mathcal{M} \leftarrow \mathcal{M}^*.$

Non-optimal recursive implementation:

Code 7.3.2: h -adaptive numerical quadrature

```
1 from numpy import *
```

```
2 from scipy import integrate
3
4 def adaptquad(f,M,rtol , abstol):
5     """
6     adaptive_quadrature_using_trapezoid_and_simpson_rules
7     Arguments:
8     f handle_to_function_f
9     M initial_mesh
10    rtol relative_tolerance_for_termination
11    abstol absolute_tolerance_for_termination ,_necessary_in_case_
    the_exact_integral_value_=0,_which_renders_a_relative_tolerance_
    meaningless.
12    """
13    h = diff(M)
14        # compute lengths of mesh
15    intervals
16    mp = 0.5*( M[:-1]+M[1:] ) # compute
17    midpoint positions
18    fx = f(M); fm = f(mp) # evaluate
19    function at positions and midpoints
20    trp_loc = h*( fx[:-1]+2*fm+fx[1:] )/4 # local trapezoid rule
21    simp_loc= h*( fx[:-1]+4*fm+fx[1:] )/6 # local simpson rule
22    l = sum(simp_loc)
23    # use simpson rule value as intermediate approximation for integral
24    value
25    est_loc = abs(simp_loc - trp_loc) # difference of values
```

obtained from local composite trapezoidal rule and local simpson rule is used as an estimate for the local quadrature error.

```

20 err_tot = sum(est_loc) # estimate
    for global error (sum moduli of local error contributions)
21 # if estimated total error not below relative or absolute threshold, refine mesh
22 if err_tot > rtol*abs(l) and err_tot > abstol:
23     refcells = nonzero( est_loc >
24         0.9*sum(est_loc)/size(est_loc) )[0]
        l =
            adaptquad(f, sort(append(M, mp[refcells])), rtol, abstol)
            # add midpoints of intervalls with large error contributions,
            recurse.
25 return l
26
27 if __name__ == '__main__':
28     f = lambda x: exp(6*sin(2*pi*x))
29     #f = lambda x: 1.0/(1e-4+x*x)
30     M = arange(11.)/10 # 0, 0.1, ... 0.9, 1
31     rtol = 1e-6; abstol = 1e-10
32     l = adaptquad(f, M, rtol, abstol)
33     exact, e = integrate.quad(f, M[0], M[-1])
34     print 'adaptquad: ', l, '"exact": ', exact
35     print 'error: ', abs(l-exact)

```

Comments on Code 7.3.1:

- Arguments: $f \hat{=}$ *handle* to function f , $M \hat{=}$ initial mesh, $\text{rtol} \hat{=}$ relative tolerance for termination, $\text{abstol} \hat{=}$ absolute tolerance for termination, necessary in case the exact integral value $= 0$, which renders a relative tolerance meaningless.
- line 13: compute lengths of mesh-intervals $[x_{j-1}, x_j]$,
- line 14: store positions of midpoints p_j ,
- line 15: evaluate function (vector arguments!),
- line 16: local composite trapezoidal rule (7.2.2),
- line 17: local simpson rule (7.1.4),
- line 18: value obtained from composite simpson rule is used as intermediate approximation for integral value,
- line 19: 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 20: estimate for global error by summing up **moduli** of local error contributions,
- line 21: terminate, once the estimated total error is below the relative or absolute error threshold,
- line 24 otherwise, add midpoints of mesh intervals with large error contributions according to (7.3.3) to the mesh and continue.

Example 7.3.3 (h -adaptive numerical quadrature).

• 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 7.3.1

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.

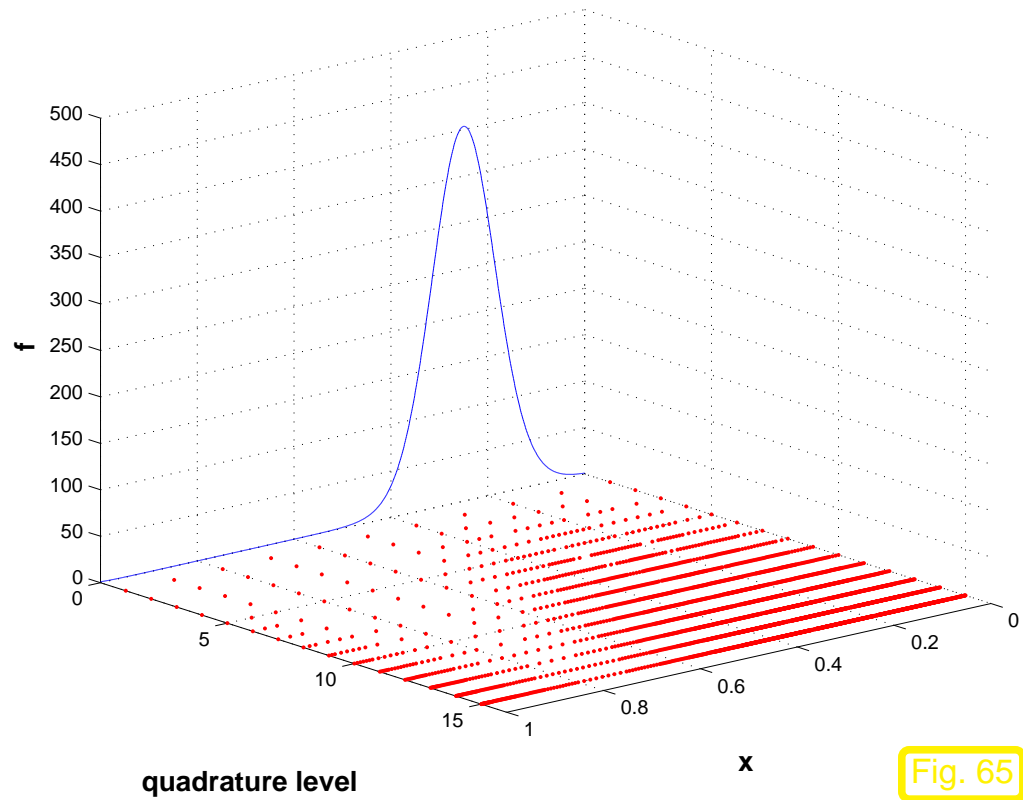


Fig. 65

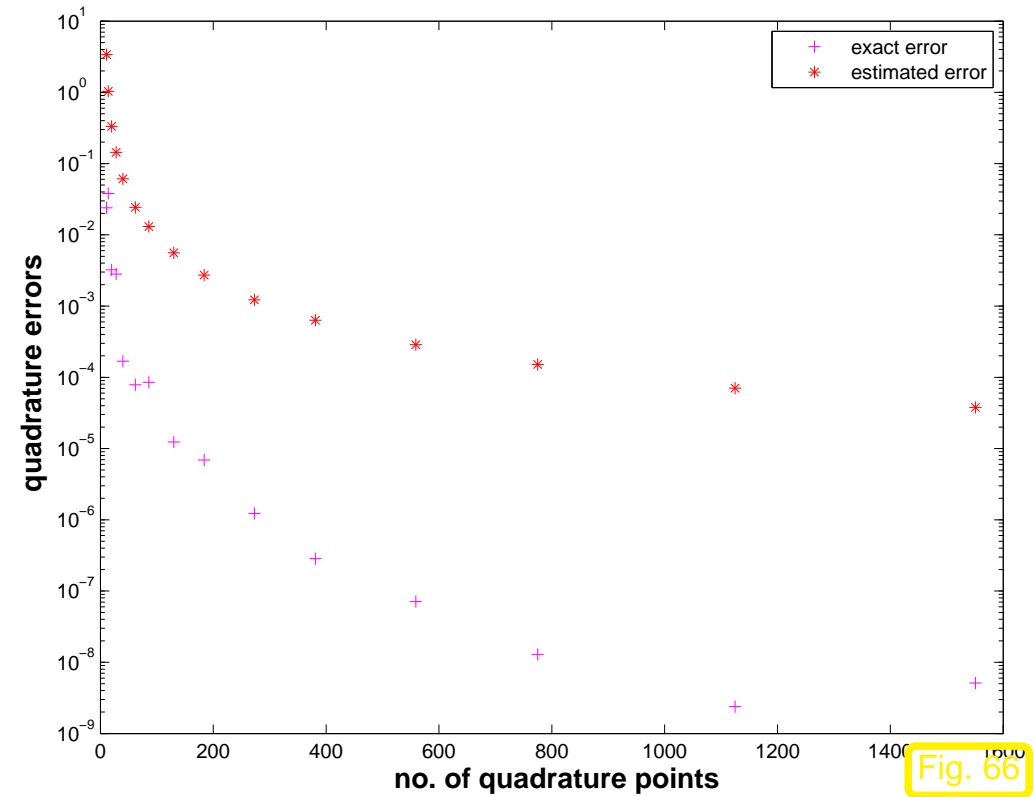
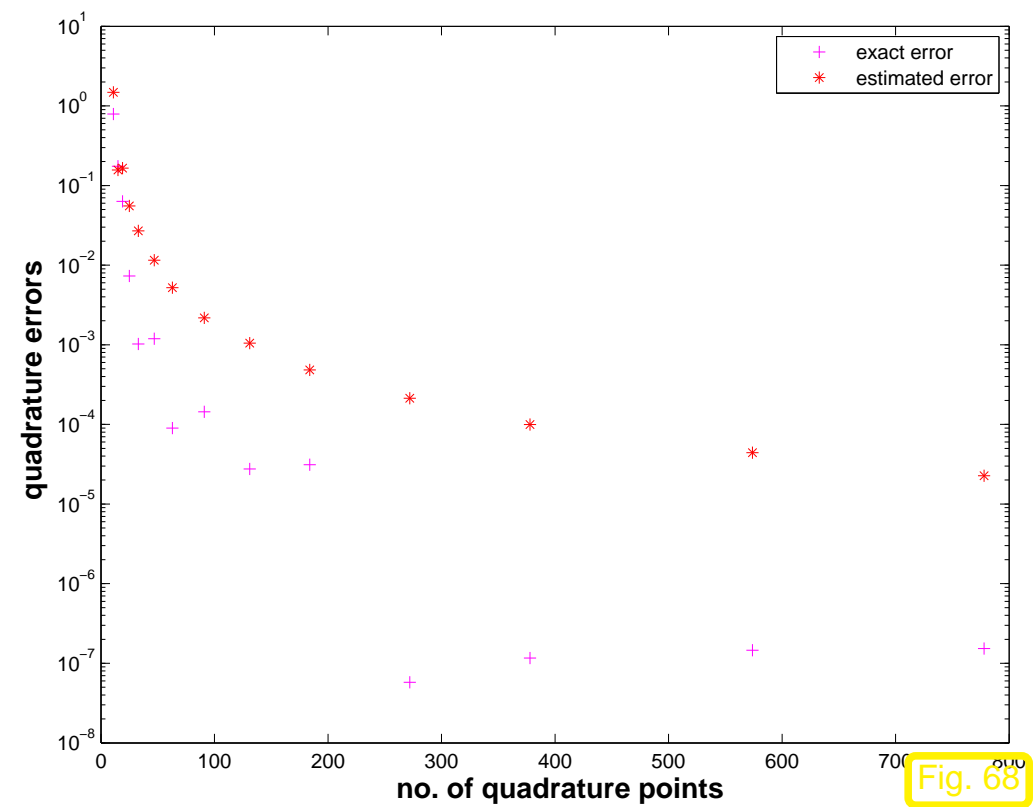
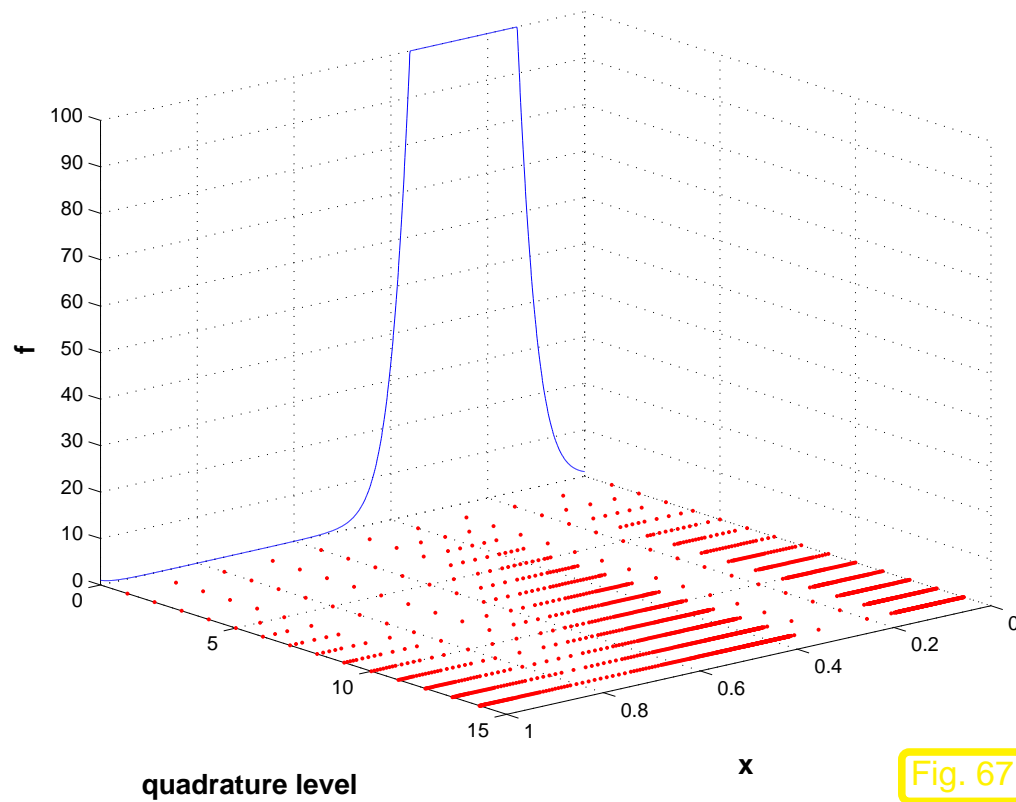


Fig. 66

• 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 (7.3.1)

However, the important information we want to glean from EST_k is about the *distribution* of the quadrature error.

Remark 7.3.4 (Adaptive quadrature in Python).

`scipy.integrate.quad`:

adaptive multigrid quadrature
wrapper to the Fortran library C

see help and `scipy.integrate.explain_quad()` for details

`scipy.integrate.quadrature`:

adaptive Gaussian quadrature



7.4 Multidimensional Quadrature

The cases of dimension $d = 2$ or $d = 3$ are easily treated by the iteration of the previous quadrature rules, e.g:

$$I = \int_a^b \int_c^d f(x, y) dx dy = \int_a^b F(y) dy,$$

where

$$F(y) = \int_c^d f(x, y) dx \approx \sum_{j_1=1}^{n_1} w_{j_1}^1 f(c_{j_1}^1, y),$$

with the corresponding quadrature rule in the x -direction.

The integration of $F(y)$ requires then the quadrature rule in the y -direction:

$$I \approx \sum_{j_1=1}^{n_1} \sum_{j_2=1}^{n_2} w_{j_1}^1 w_{j_2}^2 f(c_{j_1}^1, c_{j_2}^2).$$

We obtain hence the **tensor product quadrature**:
given the 1-dimensional quadrature rules

$$\left(w_{j_k}^k, c_{j_k}^k \right)_{1 \leq j_k \leq n_k}, \quad k = 1, \dots, d,$$

the d -dimensional integral is approximated by

$$I \approx \sum_{j_1=1}^{n_1} \dots \sum_{j_d=1}^{n_d} w_{j_1}^1 \dots w_{j_d}^d f(c_{j_1}^1, \dots, c_{j_d}^d).$$



Drawback: the number of d -dimensional quadrature points N increases exponentially with dimension d : with n quadrature points in each direction, we have $N = n^d$ quadrature points!

➤ Note: the convergence speed depends essentially on the dimension and smoothness of the function to integrate: $O(N^{-r/d})$ for a function $f \in C^r$.

7.4.1 Quadrature on Classical Sparse Grids

Consider first a quadrature formula in 1-dimension:

$$I = \int_a^b f(x)dx \approx Q_n^1(f).$$

Increasing the number of points used by the quadrature is expected to improve the result: $Q_{n+1}^1(f) - Q_n^1(f)$ is expected to decrease. Here a simple example using the trapezoidal rule:

$$T_1(f, a, b) = \frac{f(a) + f(b)}{2}(b - a),$$

and let us denote the error by

$$S_1(f, a, b) = \int_a^b f(x)dx - T_1(f, a, b),$$

which describe in Figure 7.4.1 the area uncovered by the first trapez T_1 . Here we used only the values $f(a)$ and $f(b)$. Clearly, the approximation is quite bad. We may improve it by using two shorter trapezes, i.e. using $f((a+b)/2)$ as supplementary information:

$$T_2(f, a, b) = \frac{f(a) + f((a+b)/2)}{2} + \frac{f((a+b)/2) + f(b)}{2}.$$

Clearly, both terms in the last sum are large.

However, we may improve the quadrature just by adding the information we gain by using the new function value $f(a+b)/2$. We only add to T_1 the value of the area of the triangle D_1 sitting on the top of the trapez T_1 :

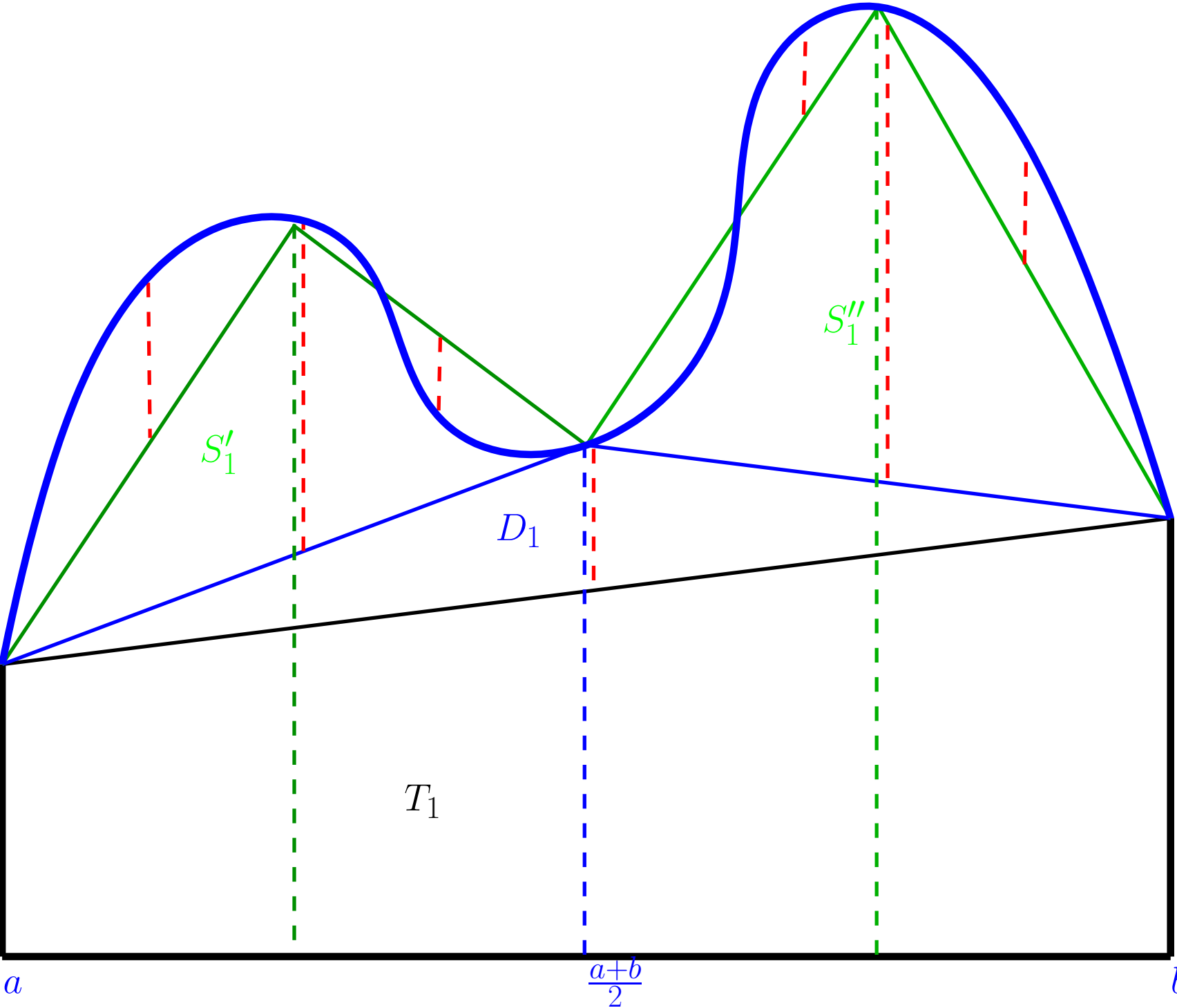
$$T_2(f, a, b) = T_1(f, a, b) + D_1(f, a, b),$$

with

$$D_1(f, a, b) = \left(f\left(\frac{a+b}{2}\right) - \frac{f(a) + f(b)}{2} \right) \frac{b-a}{2} = g_1(f, a, b) \frac{b-a}{2}.$$

We call here $g_1(f, a, b) = f\left(\frac{a+b}{2}\right) - \frac{f(a)+f(b)}{2}$ the **hierarchical surplus** of the function f on the interval $[a, b]$. The error is then the sum of the area of the triangle D_1 and the two smaller parts S'_1 and S''_1 :

$$S_1(f, a, b) = D_1(f, a, b) + S_1\left(f, a, \frac{a+b}{2}\right) + S_1\left(f, \frac{a+b}{2}, b\right).$$



We expect to have smaller errors S'_1 and S''_2 and hence smaller hierarchical surpluses when we repeat the procedure.

Let us focus on the interval $[0, 1]$, which we divide in 2^ℓ sub-intervals $\frac{k}{2^\ell}, \frac{k+1}{2^\ell}$ for $k = 0, 1, \dots, 2^\ell - 1$. We call ℓ level and consider Q_ℓ^1 a simple quadrature formula on each interval. Typical examples are the trapezoidal rule and the midpoint rule. The Clenshaw-Curtis rule and Gaussian rules are of the same flavour, but they spread the quadrature points non-uniformly.

The procedure described previously may be formulated as a simple telescopic sum of the details at each level:

$$\begin{aligned} Q_\ell^1 f &= Q_0^1 f + (Q_1^1 f - Q_0^1 f) + (Q_2^1 f - Q_1^1 f) + \dots + (Q_\ell^1 f - Q_{\ell-1}^1 f) \\ &= Q_0^1 f + \Delta_1^1 f + \Delta_2^2 f + \dots + \Delta_\ell^1 f. \end{aligned}$$

As long as we remain in 1-dimension, there is no gain in this reformulation. Things change fundamentally when going to d dimensions.

The tensor-product quadrature for the levels $\ell = (\ell_1, \dots, \ell_d)$ is

$$Q_\ell^d = Q_{\ell_1}^1 \otimes \dots \otimes Q_{\ell_d}^1$$

$$\begin{aligned}
 &= \sum_{j_1=1}^{N_1} \cdots \sum_{j_d=1}^{N_d} w_{j_1}^1 \cdots w_{j_d}^d f(c_{j_1}^1, \dots, c_{j_d}^d) \\
 &= \sum_{j=1}^d \sum_{1 \leq k_j \leq \ell_j} \left(\Delta_{k_1}^1 \otimes \cdots \otimes \Delta_{k_d}^1 \right) f.
 \end{aligned}$$

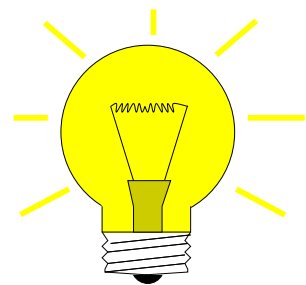
In the case of an isotropic grid $\ell = (\ell, \dots, \ell)$ we denote

$$Q_\ell^d = \sum_{j=1}^d \sum_{1 \leq k_j \leq \ell} \left(\Delta_{k_1}^1 \otimes \cdots \otimes \Delta_{k_d}^1 \right) f = \sum_{|\mathbf{k}|_\infty \leq \ell} \left(\Delta_{k_1}^1 \otimes \cdots \otimes \Delta_{k_d}^1 \right) f.$$

Idea: In the case that f is a smooth function, many of the details $\left(\Delta_{k_1}^1 \otimes \cdots \otimes \Delta_{k_d}^1 \right) f$ are so small that they may be neglected.

The **classical sparse grid quadrature (Smolyak) rule** is defined by

$$S_\ell^d f := \sum_{|\mathbf{k}|_1 \leq \ell + d - 1} \left(\Delta_{k_1}^1 \otimes \cdots \otimes \Delta_{k_d}^1 \right) f.$$



One can prove the **combinations formula**:

$$S_\ell^d f = \sum_{\ell \leq |\mathbf{k}|_1 \leq \ell + d - 1} (-1)^{\ell + d - |\mathbf{k}|_1 - 1} \binom{d-1}{|\mathbf{k}|_1 - \ell} \left(Q_{k_1}^1 \otimes \cdots \otimes Q_{k_d}^1 \right) f,$$

which is used in the practical parallel implementations.

The sparse grid is then the grid formed by the reunion of the anisotropic full grids used in the combinations formula. Its cardinality is $N = O(2^\ell \ell^{d-1})$ which is much less than $O(2^{d\ell})$ of the full grid. The error of the classical sparse grid quadrature is $O\left(N^{-r} \log^{(d-1)(r+1)}(N)\right)$ for f of a certain smoothness r .

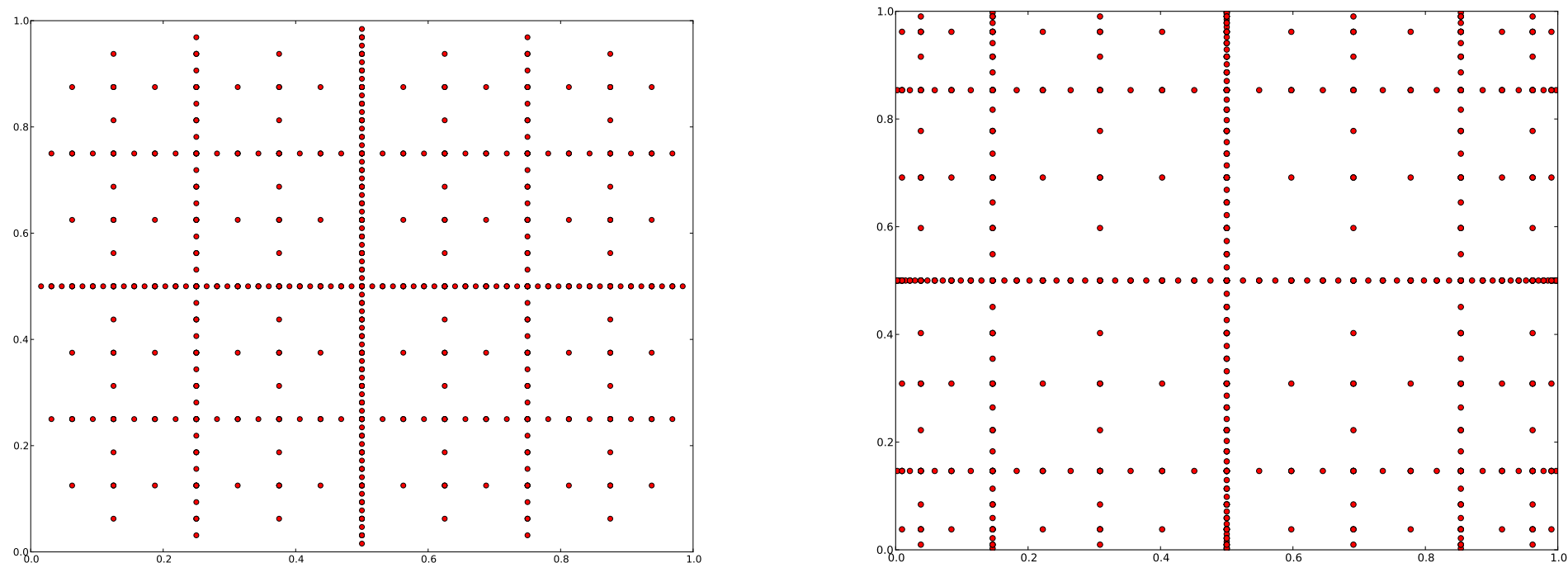


Figure 7.1: Sparse Grids based on midpoint rule and on open Clenshaw-Curtis rule

The Clenshaw-Curtis rule on a full grid in $d = 3$ dimensions at level $n = 6$ needs 274625 function evaluations and executes in about 6.6 seconds on my laptop to run. The same quadrature rule on

the sparse grid of the same level $n = 6$ needs only 3120 function evaluations and executes in about 0.1 seconds. The error in the full grid case is $1.5 \cdot 10^{-5}$ and in the sparse grid case $1.5 \cdot 10^{-6}$ for the function $f(\mathbf{x}) = (1 + 1/d)^d (x_1 \cdot \dots \cdot x_d)^{1/2}$.

The same example in $d = 4$ dimensions requires in the full grid case about 17.8 millions function evaluations with error $3.3 \cdot 10^{-5}$ and runs in 426 seconds, while the sparse grid algorithms needs 9065 function evaluations and runs in 0.3 seconds with error $8 \cdot 10^{-6}$.

The parallel code needs for $d = 10$ dimensions at level $n = 6$ only 33 seconds and produce an error about $3 \cdot 10^{-5}$. The combinations formula contains a sum with 3003 terms in this case.

7.5 Monte-Carlo Quadrature

Monte-Carlo integration: instead of step-functions as quadrature,

$$I = \int_0^1 f(t) dt \approx h \sum_{i=1}^N f(t_i)$$

where $t_i = (i - \frac{1}{2})h$, $h = \frac{1}{N}$, the $\{f(t_i)\}$ may be reordered in any way. In particular, we can order them randomly:

$$I = \int_0^1 f(t)dt \approx \frac{1}{N} \sum_{i=1}^N f(t_i)$$

where $t_i \in (0, 1)$ are uniformly distributed and sampled from a **random number generator**. A little more generally,

$$I = \int_a^b f(t)dt = |b - a| \langle f \rangle \approx |b - a| \frac{1}{N} \sum_{i=1}^N f(t_i)$$

where $t_i = a + (b - a) \cdot RNG$ (rand e.g.).

Each Monte-Carlo Method needs:

- a domain for the “experiment”: here $[0, 1]^d$
- generated random numbers : here t_i
- a deterministic computation: here $|b - a| \frac{1}{N} \sum_{i=1}^N f(t_i)$
- a representation of the result: here $P(I \in [I_N - \sigma_N, I_N + \sigma_N]) = 0.683$

Random variables and statistics are not subject of this lecture.

Essential: good RNG: fullfill statistical tests and is deterministic (reproducible).

Uniform RNG: Mersene-Twister (actual in python and Matlab), better is Marsaglia (C-MWC), best is **WELL** (2006); **SPRNG** (Masagni) is especially suited to large-scale **parallel** Monte Carlo applications.

Normal RNG: Box-Muller improved by Marsaglia, better is Ziggurat-method of Marsaglia (1998)

Note: methods based on the inversion of the distribution function resides often on badly-conditioned zero solvers and hence have to be avoided.

What is really important is the statistical error, in d -dimensions,

$$\text{error} = \frac{k_d}{\sqrt{N}}.$$

The constant $k_d = \sqrt{\text{variance}}$.

Note the **different meaning** of this error: it is now of a probabilistic nature: **68.3%** of the time, the estimate is within one standard deviation of the correct answer.

The method is very general. For example, $A \subset \mathbb{R}^d$,

$$\int_A f(\mathbf{x}) dx_1 dx_2 \cdots d\mathbf{x}_d \approx |A| \langle f \rangle$$

where $|A|$ is the volume of region A .

The error is always $\propto N^{-1/2}$ independently on the dimension d !

But k_d can be reduced significantly. Two such methods: antithetic variates & control variates. An example,

$$I_0(x) = \frac{1}{\pi} \int_0^\pi e^{-x \cos t} dt .$$

$I_0(x)$ is a modified Bessel function of order zero.

Code 7.5.1: Plain Monte-Carlo

```

1  """
2  _Computes_ integral
3  _I0(1) = (1/pi) * int(z=0..pi) exp(-cos(z)) dz by raw MC.
4  _Abramowitz_and_Stegun_give_I0(1) = 1.266066
5  """
6
7  import numpy as np

```

```
8 import time
9
10 t1 = time.time()
11
12 M = 100 # number of times we run our MC inetgration
13 asval = 1.266065878
14 ex = np.zeros(M)
15 print 'A_and_S_tables: I0(1)=', asval
16 print 'sample variance MC_I0_val',
17 print '_____',
18 k = 5 # how many experiments
19 N = 10**np.arange(1, k+1)
20 v = []; e = []
21 for n in N:
22     for m in xrange(M):
23         x = np.random.rand(n) # sample
24         x = np.exp(np.cos(-np.pi*x))
25         ex[m] = sum(x)/n # quadrature
26     ev = sum(ex)/M
27     vex = np.dot(ex, ex)/M
28     vex -= ev**2
29     v += [vex]; e += [ev]
30     print n, vex, ev
31
```

```

32 t2 = time.time()
33 t = t2-t1
34
35 print "Serial_calculation_completed, _time_=_%s_s" % t

```

- General Principle of Monte Carlo: If, at any point of a Monte Carlo calculation, we can replace an estimate by an exact value, we shall reduce the sampling error in the final result.
- Mark Kac: "You use Monte Carlo until you understand the problem."

Antithetic variates: usually only 1-D. Estimator for

$$I = I_a + I_b$$

where

$$I_a \approx \theta_a = \frac{1}{N} \sum_{i=1}^N f^{[a]}(x_i) \quad \text{and} \quad I_b \approx \theta_b = \frac{1}{N} \sum_{i=1}^N f^{[b]}(x_i)$$

so the variance is

$$\begin{aligned}
 \text{Var}_{ab} &= \langle (\theta_a + \theta_b - I)^2 \rangle \\
 &= \langle (\theta_a - I_a)^2 \rangle + \langle (\theta_b - I_b)^2 \rangle + 2 \langle (\theta_a - I_a)(\theta_b - I_b) \rangle
 \end{aligned}$$

$$= \text{Var}_a + \text{Var}_b + 2\text{Cov}_{ab}$$

If $\text{Cov}_{ab} = \langle (\theta_a - I_a)(\theta_b - I_b) \rangle < 0$ (negatively correlated), Var_{ab} is reduced.

Our example: break the integral into two pieces $0 < x < \pi/2$ and $x + \pi/2$. The new integrand is $e^{\sin(x)} + e^{-\cos(x)}$, for $0 < x < \pi/2$, and strictly **monotone**.

$$\begin{aligned} I_0(1) &\approx I_+ + I_- \\ &= \frac{1}{4N} \sum_{i=1}^N e^{\sin \pi u_i / 2} + e^{\sin \pi (1-u_i) / 2} \\ &\quad + e^{-\cos \pi u_i / 2} + e^{-\cos \pi (1-u_i) / 2}. \end{aligned}$$

Code 7.5.2: Antithetic Variates Monte-Carlo

```

1  " " "
2  _Computes_integral
3  _I0(1) = (1/pi) _int(z=0..pi) _exp(-cos(z)) _dz
4  _by _antithetic_variates.
5  _We_split_the_range_into_0_<_x_<_pi/2_and_pi/2<x<pi.
6  _The_resulting_integrand_is
7  _exp(sin(x))_+_exp(-cos(x)),_which_is_monotone
8  _increasing_in_0_<_x_<_pi/2,_so_antithetic_variates
9  _can_be_used.

```

7.5
p. 422

```

34 vex -= ev**2
35 v += [vex]; e += [ev]
36 print n, vex, ev

```

Control variates Integral

$$I = \int_0^1 f(t) dt$$

can be re-written

$$\begin{aligned}
 I &= \int_0^1 (f(t) - \phi(t)) dt + \int_0^1 \phi(t) dt \\
 &\approx \frac{1}{N} \sum_{i=1}^N [f(t_i) - \phi(t_i)] + I_\phi
 \end{aligned}$$

Gradinaru
D-MATH

Pick ϕ :

- $\phi(u) \approx g(u)$ is *nearby*, and
- $I_\phi = \int \phi(u) du$ is known.

Variance is reduced if

$$\text{var}(f - \phi) \ll \text{var}(f)$$

To see how it works, our problem

$$f(t) = e^{-\cos(\pi t)}$$

$$= 1 - \cos(\pi t) + \frac{1}{2}(\cos(\pi t))^2 + \dots$$

$$\phi(t) = \text{first three terms}$$

Code 7.5.3: Control Variates Monte-Carlo

```

1  """
2  Computes integral
3  I0(1) = (1/pi) * int(z=0..pi) exp(-cos(z)) dz
4  by control variate. The splitting is
5  <exp(-cos)> = <exp(-cos) - phi> + int(0..pi) phi dz
6  where phi = 1 + cos - (1/2)*cos*cos is the control.
7  The exact integral of the control
8  int(z=0..pi) (1 + cos - (1/2)*cos*cos) dz = 1.25.
9  Abramowitz and Stegun give I0(1) = 1.266066
10 """
11
12 import numpy as np
13
14 M = 100 # number of times we run our MC inetgration
15 asval = 1.266065878
16 ex = np.zeros(M)

```

```
17 print 'A_and_S_tables:{}_I0(1)_{}=_{}', asval
18 print 'sample{}_variance{}_MC_I0_val',
19 print '_____
20 k = 5 # how many experiments
21 N = 10**np.arange(1,k+1)
22 v = []; e = []
23 for n in N:
24     for m in xrange(M):
25         x = np.pi*np.random.rand(n) # sample
26         ctv = np.exp(-np.cos(x)) - 1. + np.cos(x) -
            0.5*np.cos(x)*np.cos(x)
27         ex[m] = 1.25 + sum(ctv)/n # quadrature
28     ev = sum(ex)/M
29     vex = np.dot(ex,ex)/M
30     vex -= ev**2
31     v += [vex]; e += [ev]
32     print n, vex, ev
```

Importance Sampling

Idea: Concentrate the distribution of the sample points in the parts of the interval that are of most importance instead of spreading them out evenly.

Importance Sampling:

$$\theta = \int_0^1 f(x) dx = \int_0^1 \frac{f(x)}{g(x)} g(x) dx = \int_0^1 \frac{f(x)}{g(x)} dG(x),$$

where g and G satisfy

$$G(x) = \int_0^x g(y) dy, \quad G(1) = \int_0^1 g(y) dy = 1,$$

and $G(x)$ is a distribution function. Variance

$$\sigma_{f/g}^2 = \int_0^1 (f(x)/g(x) - \theta)^2 dG(x)$$

How to select a good sampling function?

How about $g = cf$? g must be simple enough for us to know its integral theoretically.

Example 7.5.4. $\int_0^1 f(x) dx$ with $f(x) = 1/\sqrt{x(1-x)}$ has singularities at $x = 0, 1$. General trick: isolate them!

$$g(x) = \frac{1}{4\sqrt{x}} + \frac{1}{4\sqrt{1-x}} \Rightarrow \int_0^1 h(x) dG(x)$$

with

$$h(x) = \frac{4}{\sqrt{x} + \sqrt{1-x}}$$

and $dG(x)$ will be sampled as

```
u = rand(N)
v = rand(N)
x = u*u
w = where(v>0.5)
x[w] = 1 - x[w]
```

Code 7.5.5: Importance Sampling Monte-Carlo

```
1 from scipy import where, sqrt, arange, array
2 from numpy.random import rand
3 from time import time
4
5 from scipy.integrate import quad
6 f = lambda x: 1/sqrt(x*(1-x))
7 print 'quad(f,0,1) = ', quad(f,0,1)
8
9 func = lambda x: 4./(sqrt(x)+sqrt(1-x))
10
11 def exotic(N):
12     u = rand(N)
13     v = rand(N)
14     x = u*u
```

```
15 w = where(v>0.5)
16 x[w] = 1 - x[w]
17 return x
18
19 def ismcquad():
20     k = 5 # how many experiments
21     N = 10**arange(1,k+1)
22     ex = []
23     for n in N:
24         x = exotic(n) # sample
25         x = func(x)
26         ex += [x.sum()/n] # quadrature
27     return ex
28
29 def mcquad():
30     k = 5 # how many experiments
31     N = 10**arange(1,k+1)
32     ex = []
33     for n in N:
34         x = rand(n) # sample
35         x = f(x)
36         ex += [x.sum()/n] # quadrature
37     return ex
38
```



```
39 M = 100 # number of times we run our MC inetgration
40
41 t1 = time()
42 results = []
43 for m in xrange(M):
44     results += [ismcquad()]
45
46 t2 = time()
47 t = t2-t1
48
49 print "ISMC_Serial_calculation_completed ,_time_=_%s_s" % t
50
51 ex = array(results)
52 ev = ex.sum(axis=0)/M
53 vex = (ex**2).sum(axis=0)/M
54 vex = vex - ev**2
55 print ev[-1]
56 print vex
57
58 t1 = time()
59 results = []
60 for m in xrange(M):
61     results += [mcquad()]
62
```

```
63 t2 = time()
64 t = t2-t1
65
66 print "MC_Serial_calculation_completed ,_time_=_%s_s" % t
67
68 ex = array(results)
69 ev = ex.sum(axis=0)/M
70 vex = (ex**2).sum(axis=0)/M
71 vex = vex - ev**2
72 print ev[-1]
73 print vex
```

. Compare with the analytical computed value.

7.6 Essential Skills Learned in Chapter 7

You should know:

- several (composite) polynomial quadrature formulas with their convergence order

- what is special about the trapezoidal rule
- Gaussian quadrature rules
- how to compute a high-dimensional integral
- particularities of Monte-Carlo integration
- how to reduce the variance in Monte-Carlo integration

x

Part III

Integration of Ordinary Differential Equations

8

Single Step Methods

8.1 Initial value problems (IVP) for ODEs

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 [52, Sect. 5.6, 5.7, 6.5].

Gradinaru
D-MATH

Example 8.1.1 (Growth with limited resources). [1, Sect. 1.1]

$y : [0, T] \mapsto \mathbb{R}$: bacterial population density as a function of time

Model: autonomous **logistic differential equations**

$$\dot{y} = f(y) := (\alpha - \beta y) y \quad (8.1.1)$$

8.1

✎ Notation (Newton): $\dot{} \hat{=}$ (total) derivative with respect to time t

• $y \hat{=}$ 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.

Note: we can only compute a solution of (8.1.3), when provided with an **initial value** $y(0)$.

The logistic differential equation arises in autocatalytic reactions (as in haloform reaction, tin pest, binding of oxygen by hemoglobin or the spontaneous degradation of aspirin into salicylic acid and acetic acid, causing very old aspirin in sealed containers to smell mildly of vinegar):



As $\dot{c}_A = -r$ and $\dot{c}_B = -r + 2r = r$ we have that $c_A + c_B = c_A(0) + c_B(0) = D$ is constant and we get two decoupled equations

$$\dot{c}_A = -k(D - c_A)c_A \quad (8.1.3)$$

$$\dot{c}_B = k(D - c_B)c_B \quad (8.1.4)$$

which are of the form of (8.1.3) with $\beta = k$, $\alpha = kD$ for the element B and $\beta = -k$, $\alpha = -kD$ for the element A .

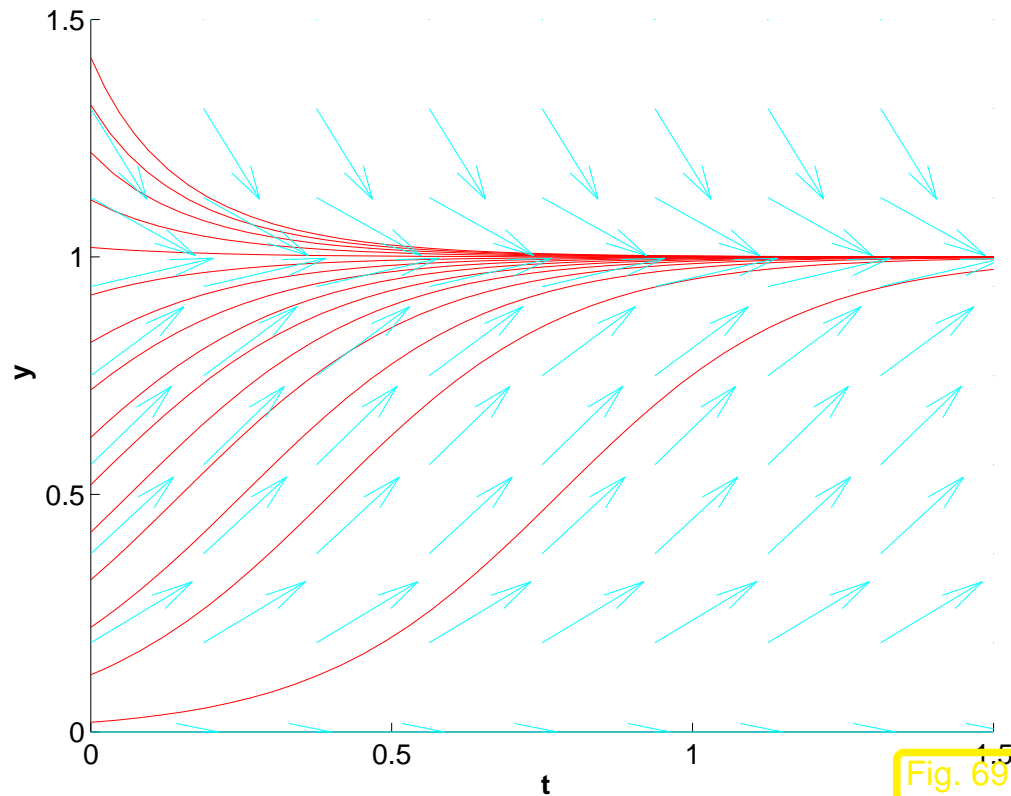


Fig. 69

Solution for different $y(0)$ ($\alpha, \beta = 5$)

By separation of variables

→ solution of (8.1.3)

for $y(0) = y_0 > 0$

$$y(t) = \frac{\alpha y_0}{\beta y_0 + (\alpha - \beta y_0) \exp(-\alpha t)}, \quad (8.1.5)$$

for all $t \in \mathbb{R}$

$f'(y^*) = 0$ for $y^* \in \{0, \alpha/\beta\}$, which are the **stationary points** for the ODE (8.1.3). If $y(0) = y^*$ the solution will be constant in time.



Example 8.1.2 (Predator-prey model). [1, Sect. 1.1] & [27, Sect. 1.1.1] initially proposed by Alfred J. Lotka in “The theory of autocatalytic chemical reactions” in 1910

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.

Model: autonomous **Lotka-Volterra ODE**:

$$\begin{aligned} \dot{u} &= (\alpha - \beta v)u \\ \dot{v} &= (\delta u - \gamma)v \end{aligned} \quad \leftrightarrow \quad \dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad \text{with} \quad \mathbf{y} = \begin{pmatrix} u \\ v \end{pmatrix}, \quad \mathbf{f}(\mathbf{y}) = \begin{pmatrix} (\alpha - \beta v)u \\ (\delta u - \gamma)v \end{pmatrix}. \quad (8.1.6)$$

population sizes:

$u(t) \rightarrow$ no. of prey at time t ,

$v(t) \rightarrow$ no. of predators at time t

vector field \mathbf{f} for Lotka-Volterra ODE

Solution curves are trajectories of particles carried along by velocity field \mathbf{f} .

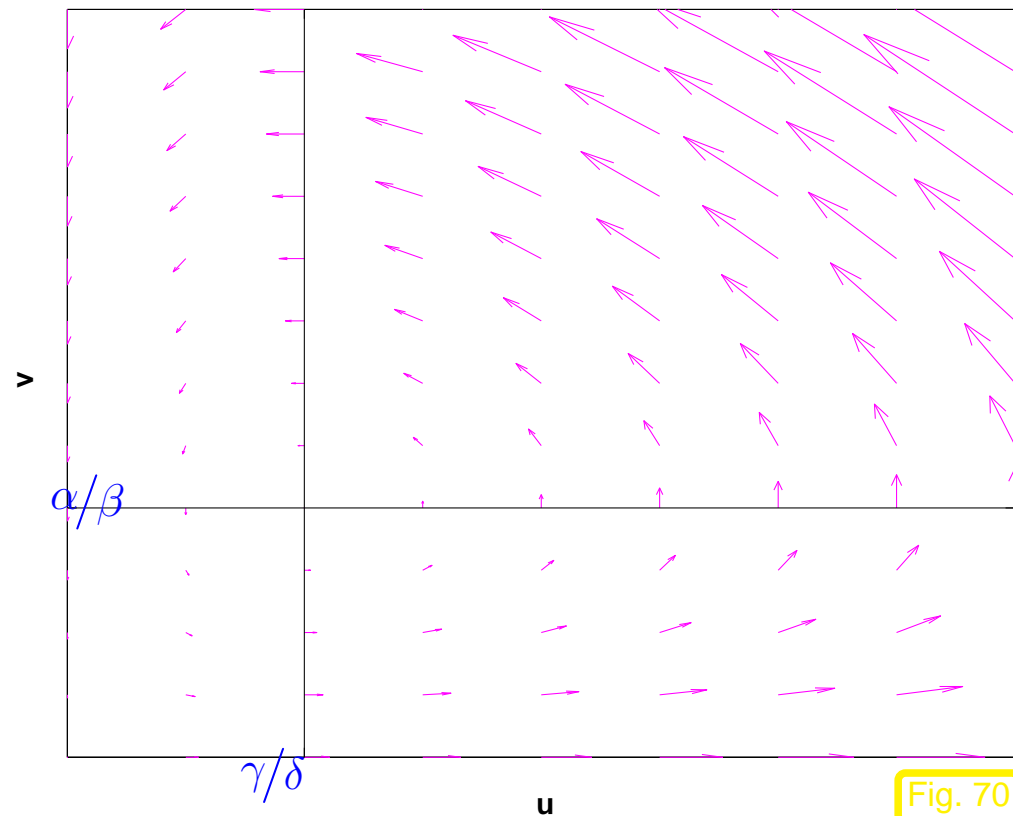
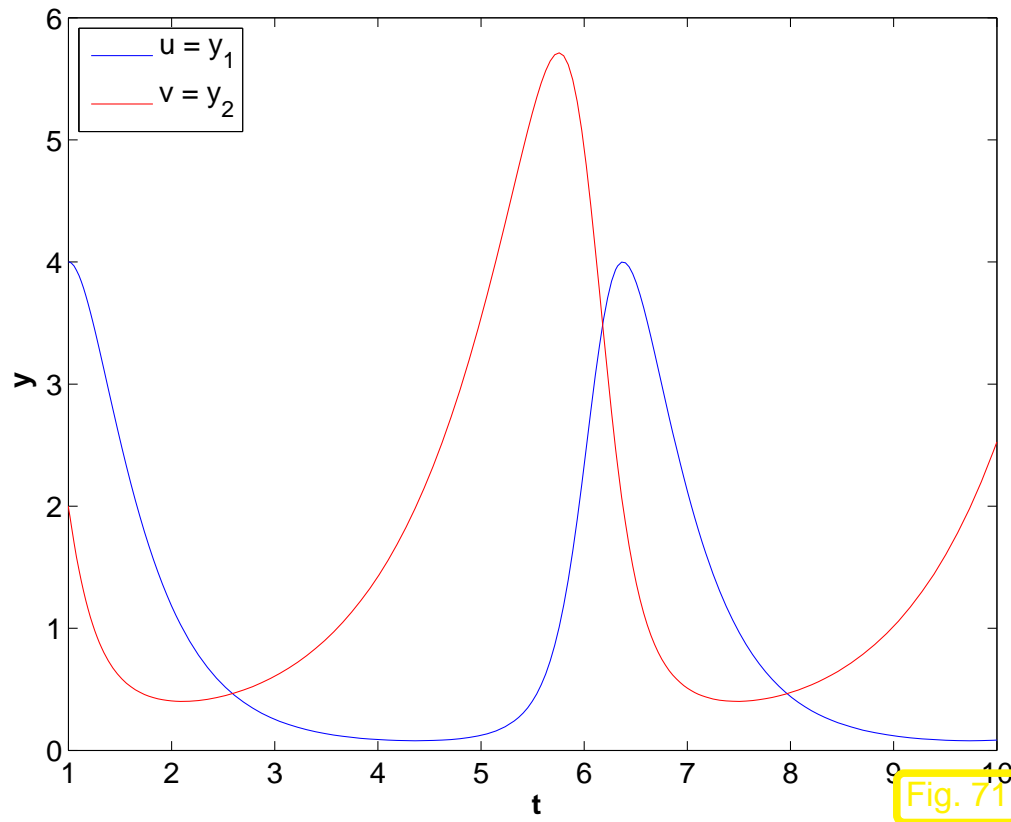
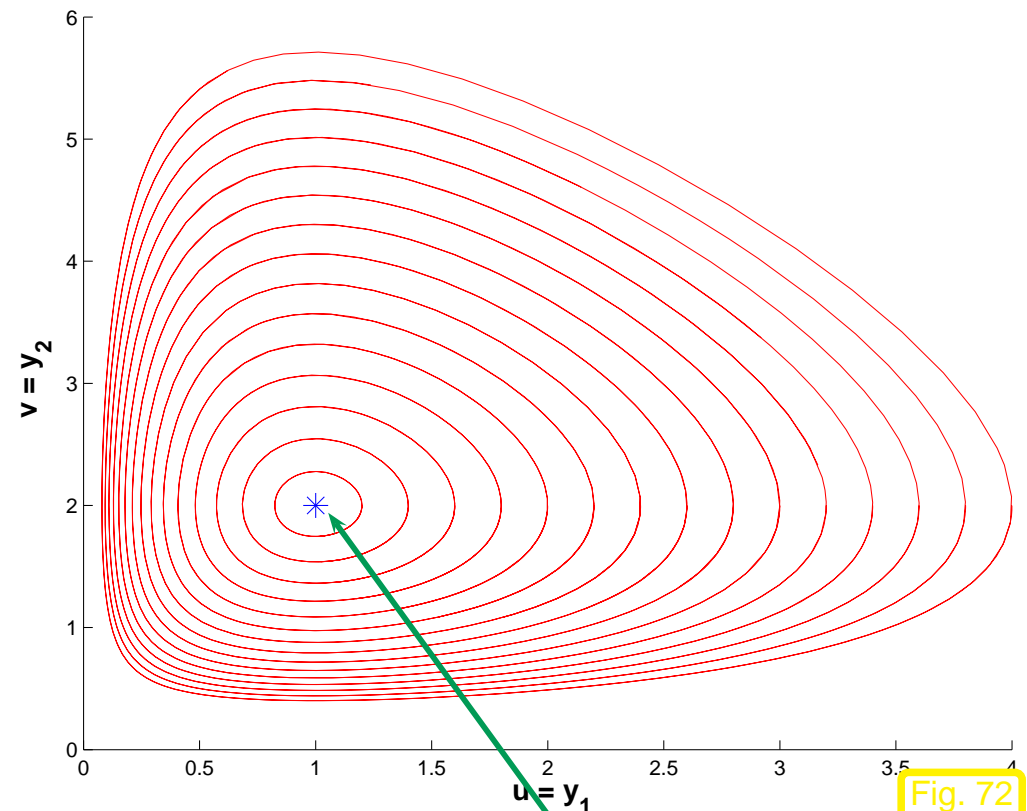


Fig. 70

Parameter values for Fig. 70: $\alpha = 2, \beta = 1, \delta = 1, \gamma = 1$



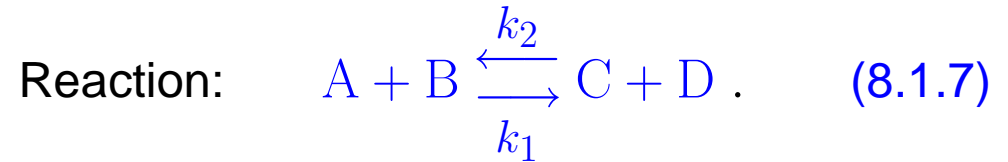
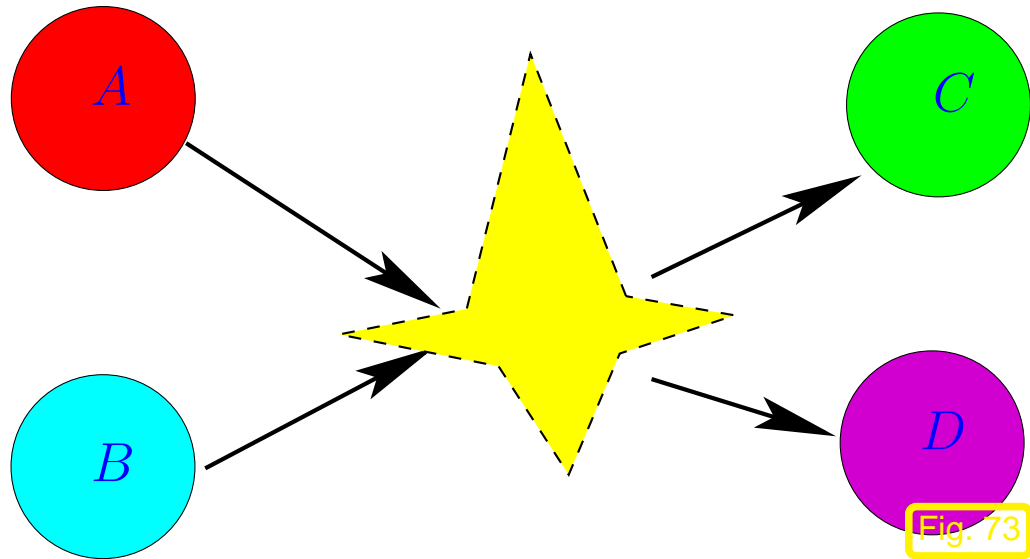
Solution $\begin{pmatrix} u(t) \\ v(t) \end{pmatrix}$ for $\mathbf{y}_0 := \begin{pmatrix} u(0) \\ v(0) \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \end{pmatrix}$



Solution curves for (8.1.6)

stationary point

Parameter values for Figs. 72, 71: $\alpha = 1, \beta = 1, \delta = 1, \gamma = 2$



with reaction constants k_1 (“forwards”), k_2 (“backwards”), $[k_1] = [k_2] = \frac{\text{cm}^3}{\text{mol s}}$.

Rule of thumb: Speed of a bimolecular reaction is proportional to the product of the concentrations of each component:

► for (8.1.7): $\dot{c}_A = \dot{c}_B = -\dot{c}_C = -\dot{c}_D = -k_1 c_A c_B + k_2 c_C c_D$. (8.1.8)

$c_A, c_B, c_C, c_D \hat{=}$ (time dependent) concentrations of components, $[c_X] = \frac{\text{mol}}{\text{cm}^3} \rightarrow c_X(t) > 0; \forall t$

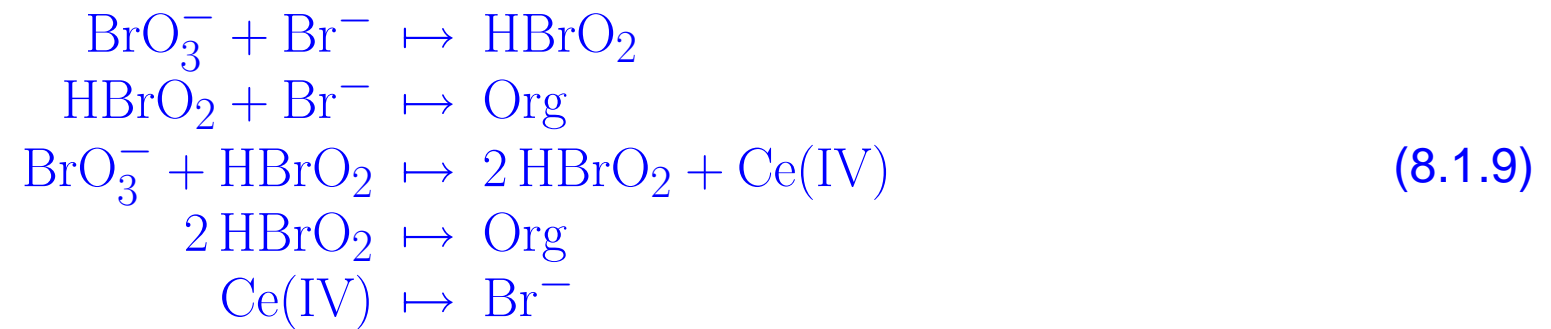
(8.1.8) = autonom ordinary differential equation (??) with

$$\mathbf{y}(t) = \begin{pmatrix} c_A(t) \\ c_B(t) \\ c_C(t) \\ c_D(t) \end{pmatrix}, \quad \mathbf{f}(t, \mathbf{y}) = (-k_1 y_1 y_2 + k_2 y_3 y_4) \begin{pmatrix} 1 \\ 1 \\ -1 \\ -1 \end{pmatrix}.$$

Conservation of mass: $\frac{d}{dt} (c_A(t) + c_B(t) + c_C(t) + c_D(t)) = 0$

Example 8.1.4 (Oregonator-Reaction).

Special case of a time-dependent oscillation Zhabotinski-Belousov-Reaction [21]:



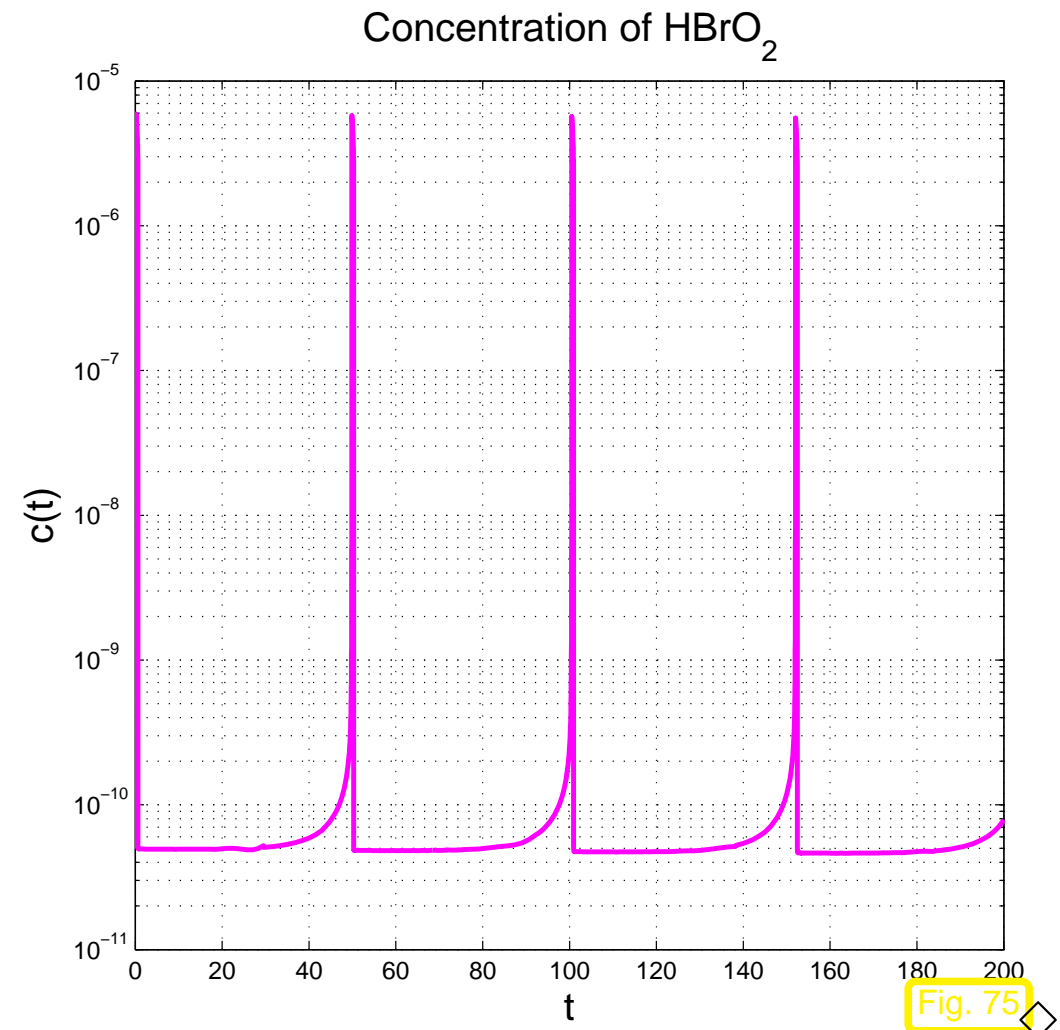
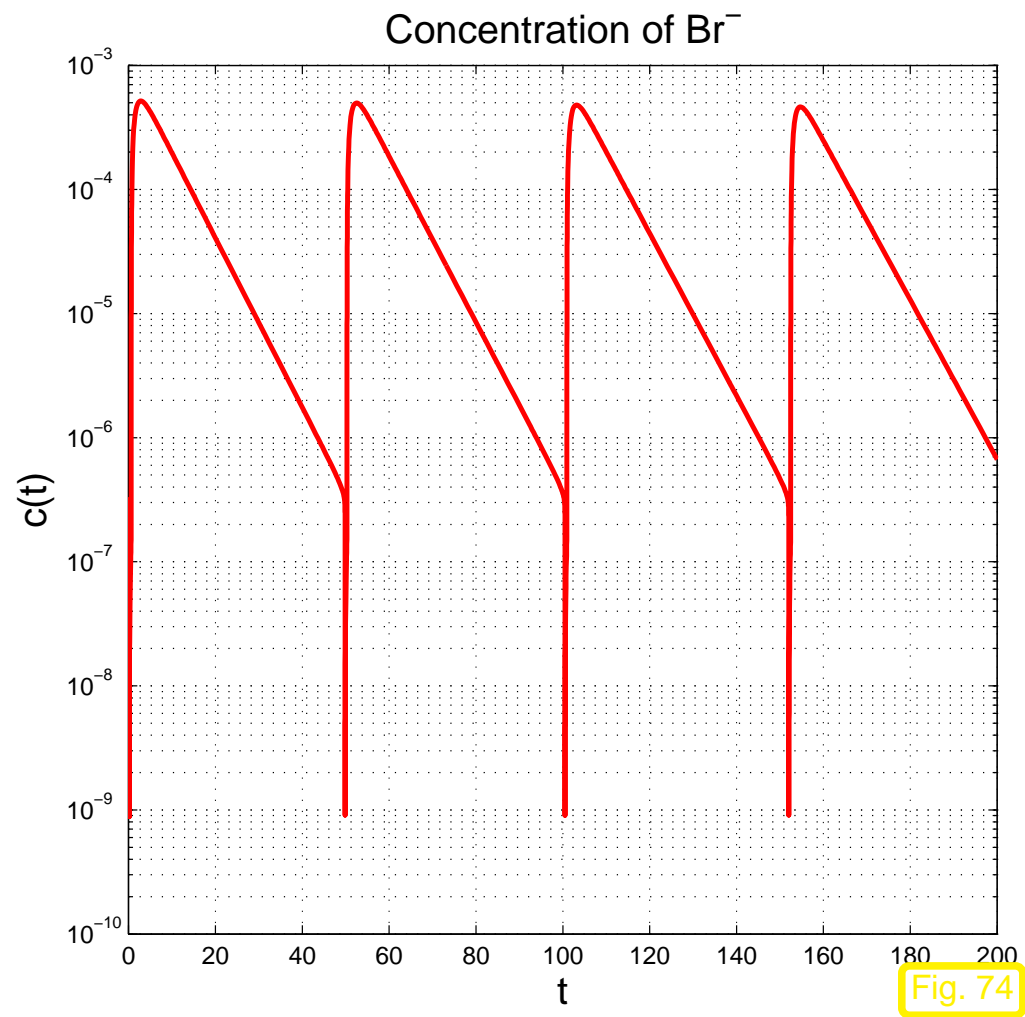
$$\begin{aligned}
 y_1 &:= c(\text{BrO}_3^-): & \dot{y}_1 &= -k_1 y_1 y_2 - k_3 y_1 y_3, \\
 y_2 &:= c(\text{Br}^-): & \dot{y}_2 &= -k_1 y_1 y_2 - k_2 y_2 y_3 + k_5 y_5, \\
 y_3 &:= c(\text{HBrO}_2): & \dot{y}_3 &= k_1 y_1 y_2 - k_2 y_2 y_3 + k_3 y_1 y_3 - 2k_4 y_3^2, \\
 y_4 &:= c(\text{Org}): & \dot{y}_4 &= k_2 y_2 y_3 + k_4 y_3^2, \\
 y_5 &:= c(\text{Ce(IV)}): & \dot{y}_5 &= k_3 y_1 y_3 - k_5 y_5,
 \end{aligned}
 \tag{8.1.10}$$

with (dimensionless) reaction constants:

$$k_1 = 1.34, \quad k_2 = 1.6 \cdot 10^9, \quad k_3 = 8.0 \cdot 10^3, \quad k_4 = 4.0 \cdot 10^7, \quad k_5 = 1.0.$$

Periodical chemical reaction  Video 1, Video 2

Simulation with initial values $y_1(0) = 0.06$, $y_2(0) = 0.33 \cdot 10^{-6}$, $y_3(0) = 0.501 \cdot 10^{-10}$, $y_4(0) = 0.03$, $y_5(0) = 0.24 \cdot 10^{-7}$:



Abstract mathematical description:

Initial value problem (IVP) for first-order ordinary differential equation (ODE): (\rightarrow [52, Sect. 5.6])

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0 . \quad (8.1.11)$$

- $\mathbf{f} : I \times D \mapsto \mathbb{R}^d \hat{=}$ right hand side (r.h.s.) ($d \in \mathbb{N}$), given in procedural form

function $v = f(t, y)$.

- $I \subset \mathbb{R} \hat{=}$ (time)interval \leftrightarrow “time variable” t
- $D \subset \mathbb{R}^d \hat{=}$ state space/phase space \leftrightarrow “state variable” \mathbf{y} (ger.: Zustandsraum)
- $\Omega := I \times D \hat{=}$ extended state space (of tuples (t, \mathbf{y}))
- $t_0 \hat{=}$ initial time, $\mathbf{y}_0 \hat{=}$ initial state \triangleright initial conditions

Terminology: $\mathbf{f} = \mathbf{f}(\mathbf{y})$, r.h.s. does not depend on time $\rightarrow \dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ is autonomous ODE

For autonomous ODEs:

- $I = \mathbb{R}$ and r.h.s. $\mathbf{y} \mapsto \mathbf{f}(\mathbf{y})$ can be regarded as stationary vector field (velocity field)
- if $t \mapsto \mathbf{y}(t)$ is solution \Rightarrow for any $\tau \in \mathbb{R}$ $t \mapsto \mathbf{y}(t + \tau)$ is solution, too.
- initial time irrelevant: canonical choice $t_0 = 0$

Note: autonomous ODEs naturally arise when modeling time-invariant systems/phenomena. All examples above led to autonomous ODEs.

Remark 8.1.5 (Conversion into autonomous ODE).

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

$$\mathbf{z}(t) := \begin{pmatrix} \mathbf{y}(t) \\ t \end{pmatrix} = \begin{pmatrix} \mathbf{z}' \\ z_{d+1} \end{pmatrix} : \dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \Leftrightarrow \dot{\mathbf{z}} = \mathbf{g}(\mathbf{z}), \quad \mathbf{g}(\mathbf{z}) := \begin{pmatrix} \mathbf{f}(z_{d+1}, \mathbf{z}') \\ 1 \end{pmatrix}.$$



Remark 8.1.6 (From higher order ODEs to first order systems).

Ordinary differential equation of order $n \in \mathbb{N}$:

$$\boxed{\mathbf{y}^{(n)} = \mathbf{f}(t, \mathbf{y}, \dot{\mathbf{y}}, \dots, \mathbf{y}^{(n-1)})} \quad (8.1.12)$$

Notation: superscript $^{(n)} \hat{=}$ n -th temporal derivative t

➤ Conversion into 1st-order ODE (system of size nd)

$$\mathbf{z}(t) := \begin{pmatrix} \mathbf{y}(t) \\ \mathbf{y}^{(1)}(t) \\ \vdots \\ \mathbf{y}^{(n-1)}(t) \end{pmatrix} = \begin{pmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \\ \vdots \\ \mathbf{z}_n \end{pmatrix} \in \mathbb{R}^{dn}: \quad (8.1.12) \quad \Leftrightarrow \quad \dot{\mathbf{z}} = \mathbf{g}(\mathbf{z}), \quad \mathbf{g}(\mathbf{z}) := \begin{pmatrix} \mathbf{z}_2 \\ \mathbf{z}_3 \\ \vdots \\ \mathbf{z}_n \\ \mathbf{f}(t, \mathbf{z}_1, \dots, \mathbf{z}_n) \end{pmatrix}. \quad (8.1.13)$$

Note: n initial values $\mathbf{y}(t_0), \dot{\mathbf{y}}(t_0), \dots, \mathbf{y}^{(n-1)}(t_0)$ required!

△

Basic assumption: right hand side $\mathbf{f} : I \times D \mapsto \mathbb{R}^d$ locally Lipschitz continuous in \mathbf{y}

Definition 8.1.1 (Lipschitz continuous function). $(\rightarrow [52, \text{Def. 4.1.4}])$

$\mathbf{f} : \Omega \mapsto \mathbb{R}^d$ is **Lipschitz continuous** (in the second argument), if

$$\exists L > 0: \quad \|\mathbf{f}(t, \mathbf{w}) - \mathbf{f}(t, \mathbf{z})\| \leq L \|\mathbf{w} - \mathbf{z}\| \quad \forall (t, \mathbf{w}), (y, \mathbf{z}) \in \Omega.$$

Definition 8.1.2 (Local Lipschitz continuity). $(\rightarrow [52, \text{Def. 4.1.5}])$

Notation: $D_{\mathbf{y}}\mathbf{f} \hat{=}$ derivative of \mathbf{f} w.r.t. state variable (= Jacobian $\in \mathbb{R}^{d,d}$!)

A simple criterion for local Lipschitz continuity:

Lemma 8.1.3 (Criterion for local Lipschitz continuity).

If \mathbf{f} and $D_{\mathbf{y}}\mathbf{f}$ are continuous on the extended state space Ω , then \mathbf{f} is locally Lipschitz continuous (\rightarrow Def. 8.1.2).

Theorem 8.1.4 (Theorem of Peano & Picard-Lindelöf). [1, Satz II(7.6)], [52, Satz 6.5.1]

*If $\mathbf{f} : \hat{\Omega} \mapsto \mathbb{R}^d$ is locally Lipschitz continuous (\rightarrow Def. 8.1.2) then for all initial conditions $(t_0, \mathbf{y}_0) \in \hat{\Omega}$ the IVP (8.1.11) has a solution $\mathbf{y} \in C^1(J(t_0, \mathbf{y}_0), \mathbb{R}^d)$ with **maximal** (temporal) domain of definition $J(t_0, \mathbf{y}_0) \subset \mathbb{R}$.*

Remark 8.1.7 (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 $J(t_0, \mathbf{y}_0) = I \iff$ solution $\mathbf{y} : I \mapsto \mathbb{R}^d$ is **global**.



Notation: for autonomous ODE we always have $t_0 = 0$, therefore write $J(\mathbf{y}_0) := J(0, \mathbf{y}_0)$.

In light of Rem. 8.1.5 and Thm. 8.1.4: we consider only

autonomous IVP: $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$, (8.1.14)

with locally Lipschitz continuous (\rightarrow Def. 8.1.2) right hand side \mathbf{f} .

Assumption 8.1.5 (Global solutions).

All solutions of (8.1.14) are global: $J(\mathbf{y}_0) = \mathbb{R}$ for all $\mathbf{y}_0 \in D$.

Change of perspective: fix “time of interest” $t \in \mathbb{R} \setminus \{0\}$

➤ mapping $\Phi^t : \begin{cases} D \mapsto D \\ \mathbf{y}_0 \mapsto \mathbf{y}(t) \end{cases}$, $t \mapsto \mathbf{y}(t)$ solution of IVP (8.1.14) ,

is well-defined mapping of the state space into itself, by Thm. 8.1.4 and Ass. 8.1.5

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 time t and an initial state value \mathbf{y}_0 !

Definition 8.1.6 (Evolution operator).

Under Assumption 8.1.5 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* for the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$.

Note: $t \mapsto \Phi^t \mathbf{y}_0$ describes the solution of $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ for $\mathbf{y}(0) = \mathbf{y}_0$ (a trajectory)

Remark 8.1.8 (Group property of autonomous evolutions).

Under Assumption 8.1.5 the evolution operator gives rise to a **group** of mappings $D \mapsto D$:

$$\Phi^s \circ \Phi^t = \Phi^{s+t} \quad , \quad \Phi^{-t} \circ \Phi^t = Id \quad \forall t \in \mathbb{R} . \quad (8.1.15)$$

This is a consequence of the uniqueness theorem Thm. 8.1.4. 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 $s + t$.



8.2 Euler methods

Targeted: initial value problem (8.1.11)

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0 . \quad (8.1.11)$$

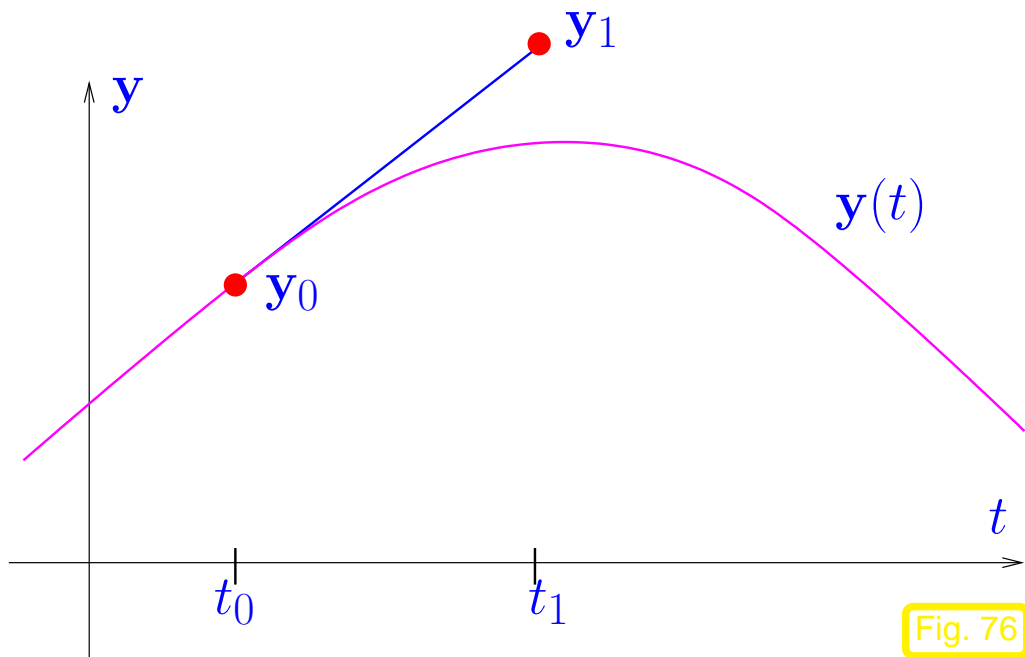
Sought: *approximate* solution of (8.1.11) on $[t_0, T]$ up to **final time** $T \neq t_0$

However, the solution of an initial value problem is a *function* $J(t_0, \mathbf{y}_0) \mapsto \mathbb{R}^d$ and requires a suitable approximate representation. We postpone this issue here and first study a geometric approach to numerical integration.

numerical integration = approximate solution of initial value problems for ODEs

(Please distinguish from “numerical quadrature”, see Ch. 7.)

- Idea: ❶ **timestepping**: successive approximation of evolution on *small* intervals $[t_{k-1}, t_k]$, $k = 1, \dots, N$, $t_N := T$,
- ❷ approximation of solution on $[t_{k-1}, t_k]$ by **tangent** curve to current initial condition.



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!

Example 8.2.1 (Visualization of explicit Euler method).

IVP for Riccati differential equation, see Ex. ??

$$\dot{y} = y^2 + t^2. \quad (??)$$

Here: $y_0 = \frac{1}{2}, t_0 = 0, T = 1,$
 — $\hat{=}$ “Euler polygon” for uniform timestep $h = 0.2$

$\rightarrow \hat{=}$ tangent field of Riccati ODE

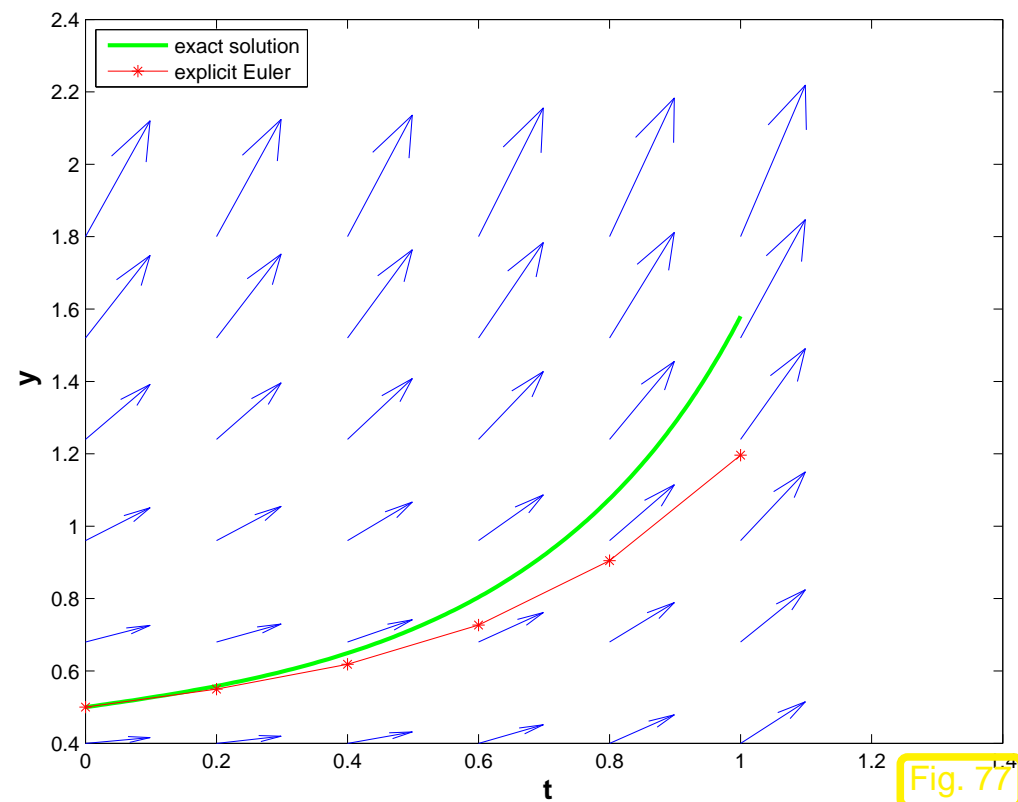


Fig. 77



Gradinaru
D-MATH

Formula: explicit Euler method generates a sequence $(\mathbf{y}_k)_{k=0}^N$ by the recursion

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_k, \mathbf{y}_k), \quad k = 0, \dots, N-1, \quad (8.2.1)$$

with local (size of) **timestep** (stepsize) $h_k := t_{k+1} - t_k$.

Remark 8.2.2 (Explicit Euler method as **difference scheme**).

(8.2.1) by approximating derivative $\frac{d}{dt}$ by **forward difference quotient** on a (temporal) **mesh** $\mathcal{M} := \{t_0, t_1, \dots, t_N\}$:

$$\dot{\mathbf{y}} = f(t, \mathbf{y}) \quad \longleftrightarrow \quad \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} = f(t_k, \mathbf{y}_h(t_k)) , \quad k = 0, \dots, N-1 . \quad (8.2.2)$$

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).

△

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

$$\dot{\mathbf{y}} = f(t, \mathbf{y}) \quad \longleftrightarrow \quad \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} = f(t_{k+1}, \mathbf{y}_h(t_{k+1})) , \quad k = 0, \dots, N-1 . \quad (8.2.3)$$

Backward difference quotient

This leads to another simple timestepping scheme analogous to (8.2.1):

$$\boxed{\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1}) , \quad k = 0, \dots, N-1} , \quad (8.2.4)$$

with local **timestep** (stepsize) $h_k := t_{k+1} - t_k$.

(8.2.4) = **implicit Euler method**

Note: (8.2.4) requires solving of a (possibly non-linear) system of equations to obtain \mathbf{y}_{k+1} !
(► Terminology “implicit”)

Remark 8.2.3 (Feasibility of implicit Euler timestepping).

Consider autonomous ODE and assume continuously differentiable right hand side: $\mathbf{f} \in C^1(D, \mathbb{R}^d)$.

(8.2.4) \Leftrightarrow h -dependent non-linear system of equations:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1}) \Leftrightarrow G(h, \mathbf{y}_{k+1}) = 0 \quad \text{with} \quad G(h, \mathbf{z}) := \mathbf{z} - h \mathbf{f}(\mathbf{z}) - \mathbf{y}_k.$$

Partial derivative:

$$\frac{dG}{d\mathbf{z}}(0, \mathbf{z}) = \mathbf{I}$$

Implicit function theorem: for **sufficiently small** $|h|$ the equation $G(h, \mathbf{z}) = 0$ defines a continuous function $\mathbf{z} = \mathbf{z}(h)$.

How to interpret the sequence $(\mathbf{y}_k)_{k=0}^N$ from (8.2.1)?

By “geometric insight” we expect:

$$\mathbf{y}_k \approx \mathbf{y}(t_k)$$

(Throughout, we use the notation $\mathbf{y}(t)$ for the exact solution of an IVP.)

If we are merely interested in the final state $\mathbf{y}(T)$, then the explicit Euler method will give us the answer \mathbf{y}_N .

If we are interested in an approximate solution $\mathbf{y}_h(t) \approx \mathbf{y}(t)$ as a function $[t_0, T] \mapsto \mathbb{R}^d$, we have to do

post-processing = reconstruction of a function from $\mathbf{y}_k, k = 0, \dots, N$

Technique: *interpolation*, see Ch. 5

Simplest option: piecewise linear interpolation (\rightarrow Sect. ??) \rightarrow **Euler polygon**, see Fig. 77.

Recall Euler methods for autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$:

explicit Euler: $\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_k)$,

implicit Euler: \mathbf{y}_{k+1} : $\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_{k+1})$.

Both formulas provide a mapping

$$(\mathbf{y}_k, h_k) \mapsto \Psi(h, \mathbf{y}_k) := \mathbf{y}_{k+1} . \quad (8.2.5)$$

Recall the interpretation of the \mathbf{y}_k as approximations of $\mathbf{y}(t_k)$:

$$\Psi(h, \mathbf{y}) \approx \Phi^h \mathbf{y} , \quad (8.2.6)$$

where Φ is the evolution operator (\rightarrow Def. 8.1.6) for $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$.

The Euler methods provide approximations for evolution operator for ODEs

This is what every single step method does: it tries to approximate the evolution operator Φ for an ODE by a mapping of the type (8.2.5).

\rightarrow mapping Ψ from (8.2.5) is called **discrete evolution**.

Vice versa: a mapping Ψ as in (8.2.5) defines a single step method.

☞ In a sense, a single step method defined through its associated discrete evolution does not approximate an initial value problem, but tries to approximate an ODE.

Definition 8.2.1 (Single step method (for autonomous ODE)).

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 $\mathcal{M} := \{t_0 < t_1 < \dots < t_N = T\}$ the recursion

$$\mathbf{y}_{k+1} := \Psi(t_{k+1} - t_k, \mathbf{y}_k), \quad k = 0, \dots, N-1, \quad (8.2.7)$$

defines a **single step method** (SSM, ger.: *Einschrittverfahren*) for the autonomous IVP $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$.

Procedural view of discrete evolutions:

$$\Psi^h_{\mathbf{y}} \longleftrightarrow \begin{array}{l} \text{function } \mathbf{y}_1 = \text{esvstep}(h, \mathbf{y}_0) . \\ \quad (\text{function } \mathbf{y}_1 = \text{esvstep}(\text{ rhs}, h, \mathbf{y}_0)) \end{array}$$

 Notation: $\Psi^h \mathbf{y} := \Psi(h, \mathbf{y})$

Concept of single step method according to Def. 8.2.1 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 discrete evolution defined on $I \times I \times D$.

Remark 8.2.4 (Notation for single step methods).

Many authors specify a single step method by writing down the first step:

$$\mathbf{y}_1 = \text{expression in } \mathbf{y}_0 \text{ and } \mathbf{f} .$$

Also this course will sometimes adopt this practice.



8.3 Convergence of single step methods

Important issue: accuracy of approximation $\mathbf{y}_k \approx \mathbf{y}(t_k)$?

As in the case of composite numerical quadrature, see Sect. 7.2: in general impossible to predict error $\|\mathbf{y}_N - \mathbf{y}(T)\|$ for particular choice of timesteps.

Tractable: **asymptotic behavior** of error for timestep $h := \max_k h_k \rightarrow 0$



Will tell us asymptotic gain in accuracy for extra computational effort.
(computational effort = no. of **f**-evaluations)

Example 8.3.1 (Speed of convergence of Euler methods).

- IVP for Riccati ODE (??) on $[0, 1]$
- explicit Euler method (8.2.1) with uniform timestep $h = 1/N$,
 $N \in \{5, 10, 20, 40, 80, 160, 320, 640\}$.

• **Error** $\text{err}_h := |y(1) - y_N|$

Observation:

algebraic convergence $\text{err}_h = O(h)$

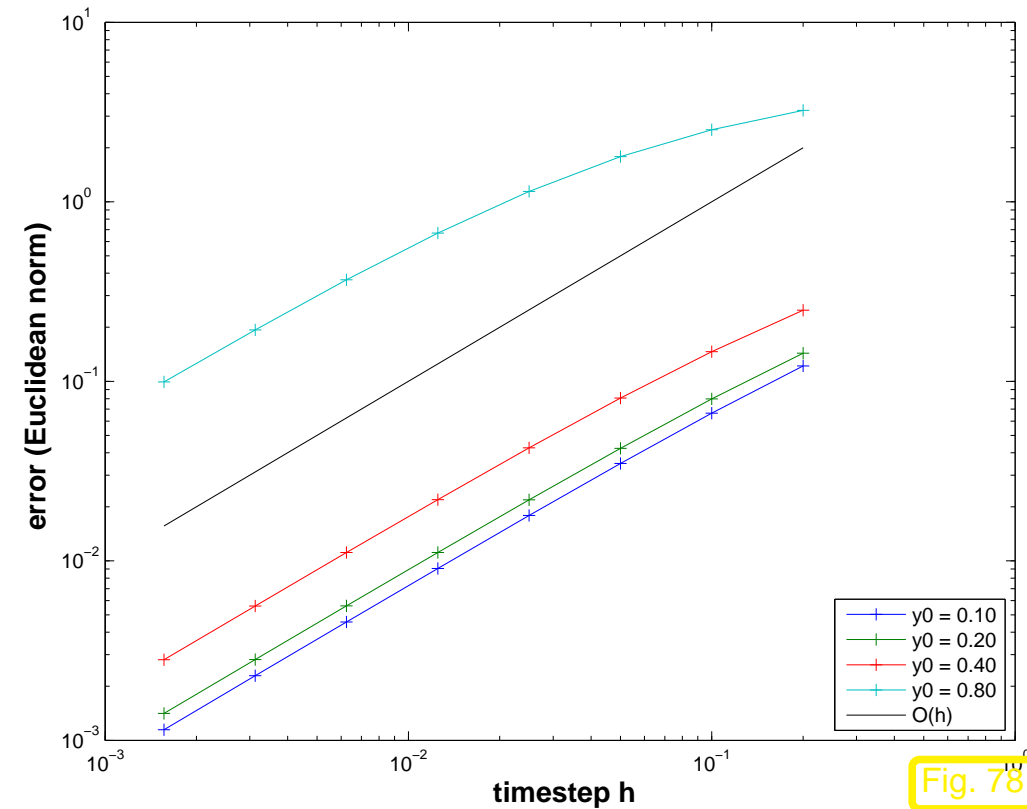


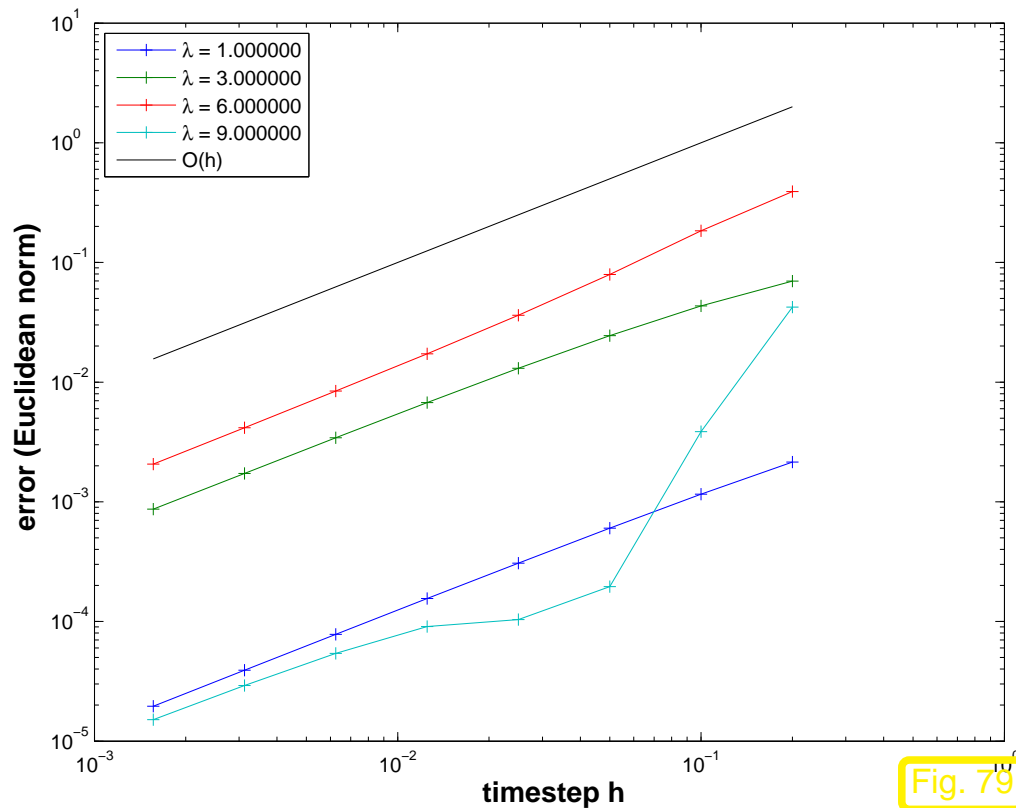
Fig. 78

• IVP for logistic ODE, see Ex. 8.1.1

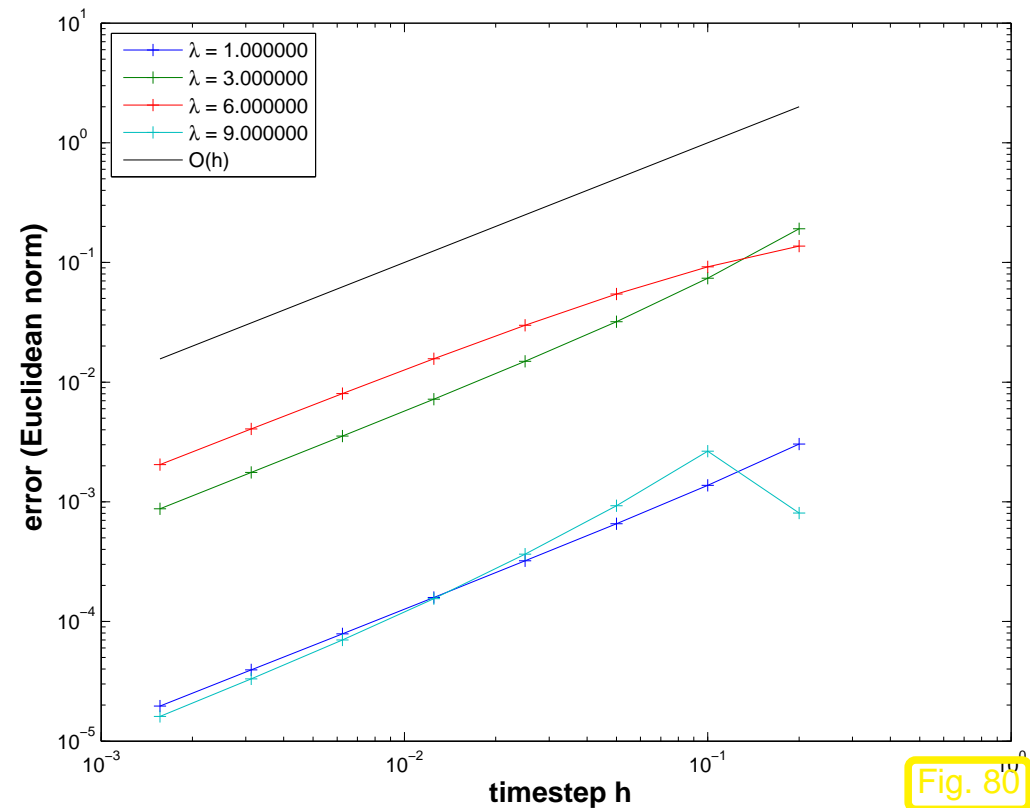
$$\dot{y} = \lambda y(1 - y) \quad , \quad y(0) = 0.01 \quad .$$

• Explicit and implicit Euler methods (8.2.1)/(8.2.4) 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

$O(h)$ algebraic convergence in both cases

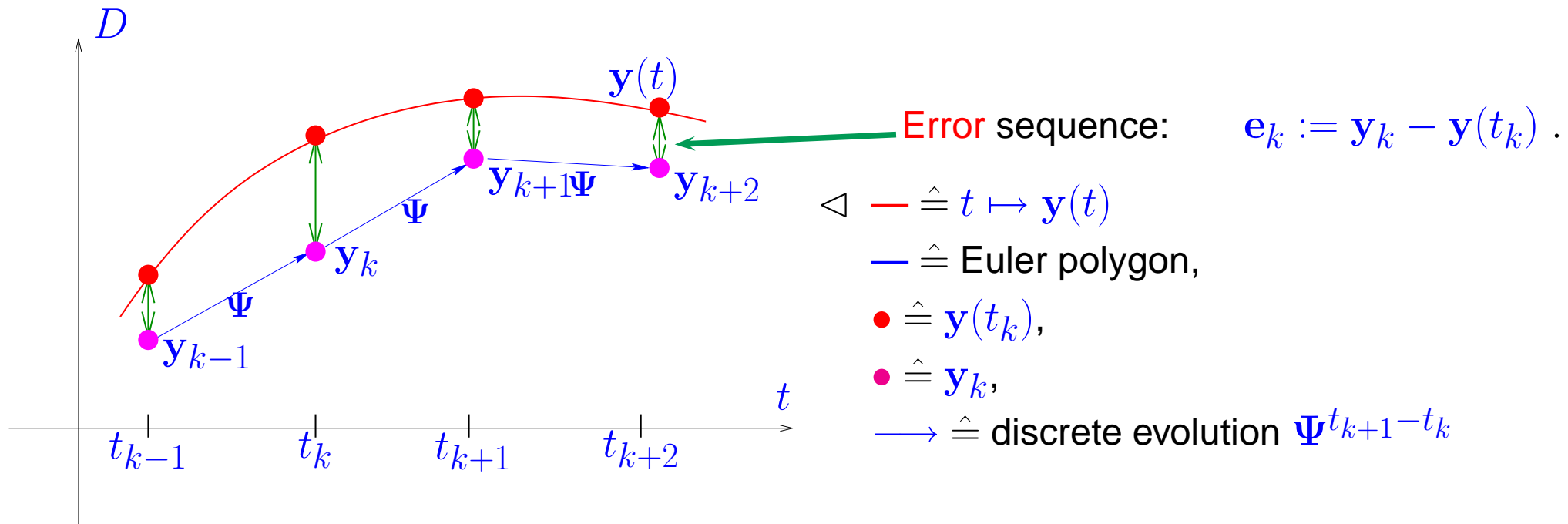


Convergence analysis for explicit Euler method (8.2.1) for autonomous IVP (8.1.11) with sufficiently smooth and (*globally*) Lipschitz continuous \mathbf{f} , that is,

$$\exists L > 0: \quad \|\mathbf{f}(t, \mathbf{y}) - \mathbf{f}(t, \mathbf{z})\| \leq L \|\mathbf{y} - \mathbf{z}\| \quad \forall \mathbf{y}, \mathbf{z} \in D. \quad (8.3.1)$$

Recall: recursion for explicit Euler method

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_k), \quad k = 1, \dots, N-1. \quad (8.2.1)$$



① Abstract splitting of error:

Here and in what follows we rely on the abstract concepts of the evolution operator Φ associated with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ (\rightarrow Def. 8.1.6) and discrete evolution operator Ψ defining the explicit Euler single step method, see Def. 8.2.1:

$$(8.2.1) \Rightarrow \Psi^h \mathbf{y} = \mathbf{y} + h\mathbf{f}(\mathbf{y}) . \quad (8.3.2)$$

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

$$\begin{aligned} \mathbf{e}_{k+1} &= \Psi^{h_k} \mathbf{y}_k - \Phi^{h_k} \mathbf{y}(t_k) \\ &= \underbrace{\Psi^{h_k} \mathbf{y}_k - \Psi^{h_k} \mathbf{y}(t_k)}_{\text{propagated error}} \\ &\quad + \underbrace{\Psi^{h_k} \mathbf{y}(t_k) - \Phi^{h_k} \mathbf{y}(t_k)}_{\text{one-step error}} . \end{aligned} \quad (8.3.3)$$

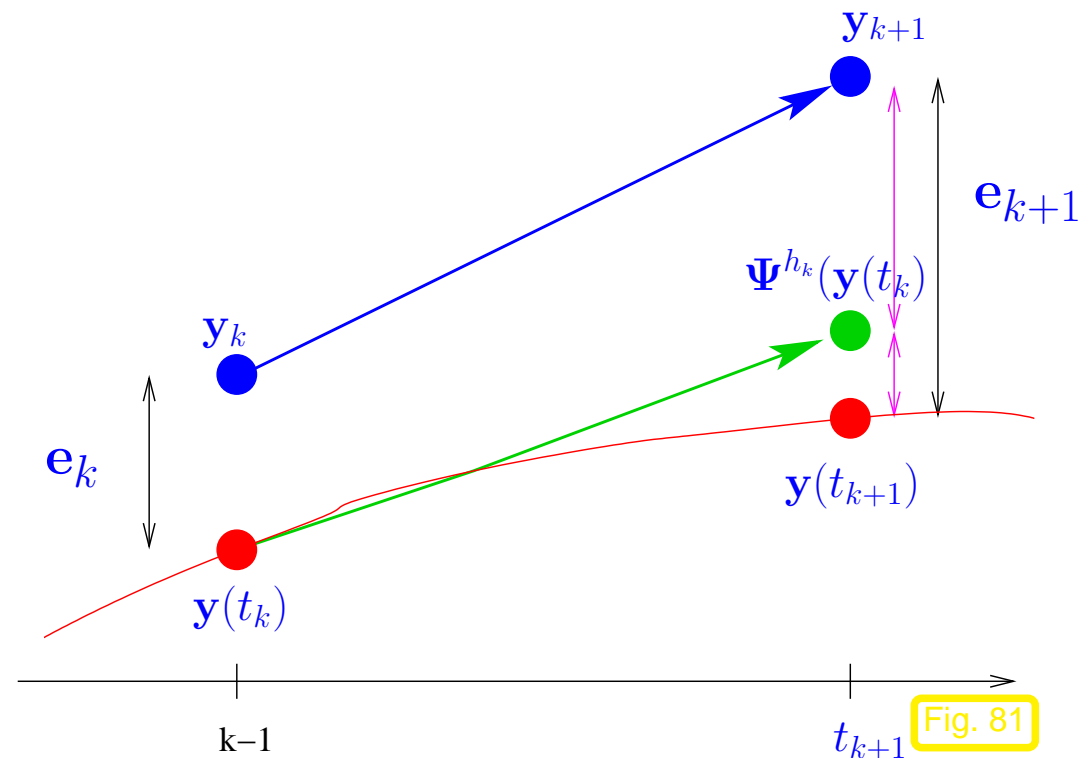
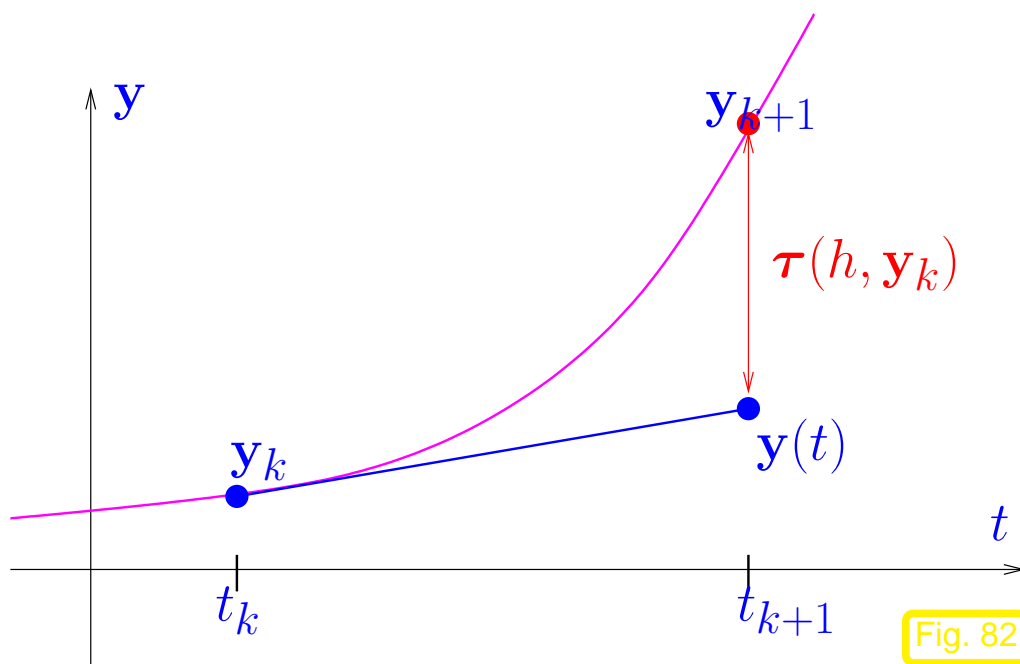


Fig. 81



one-step error:

$$\tau(h, \mathbf{y}) := \Psi^h \mathbf{y} - \Phi^h \mathbf{y} . \quad (8.3.4)$$

◁ geometric visualisation of one-step error for explicit Euler method (8.2.1), *cf.* Fig. 76.

Fig. 82

notation: $t \mapsto \mathbf{y}(t) \hat{=}$ (unique) solution of IVP, *cf.* Thm. 8.1.4.

② Estimate for one-step error:

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. 84).

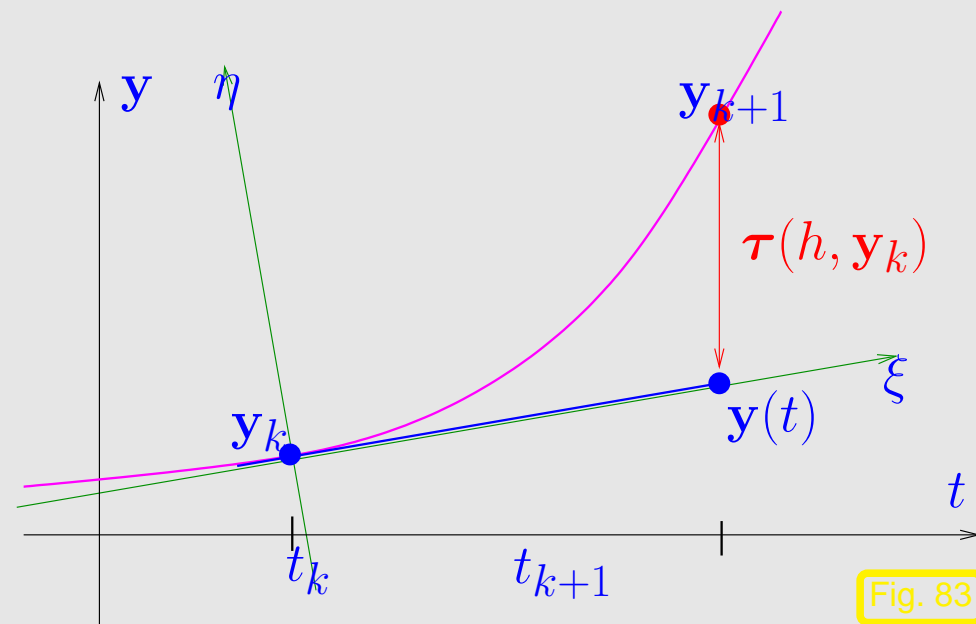


Fig. 83

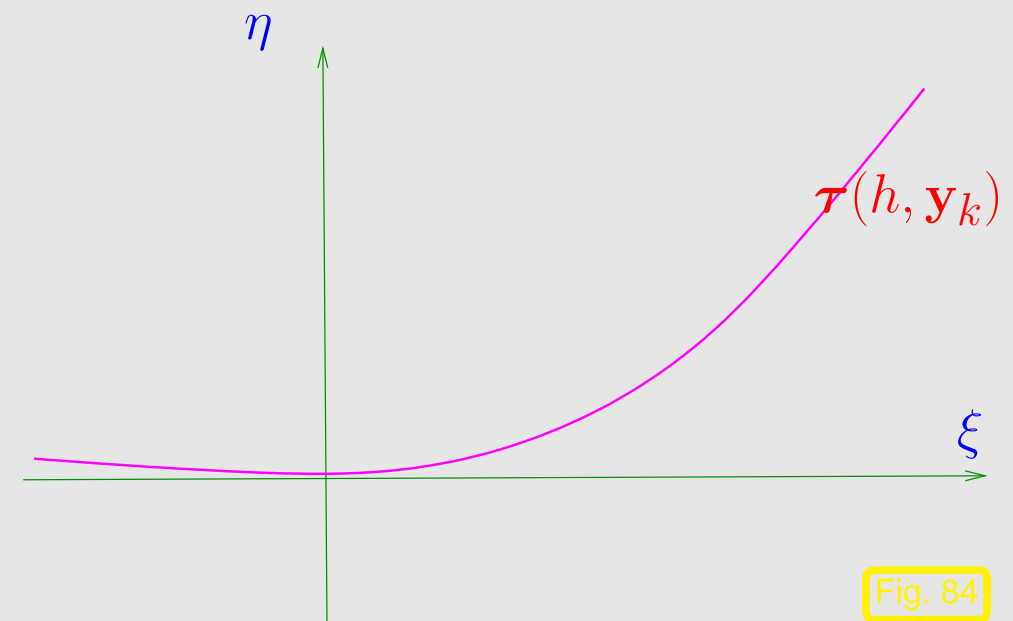


Fig. 84

Analytic considerations: recall Taylor's formula for function $\mathbf{y} \in C^{K+1}$

$$\mathbf{y}(t+h) - \mathbf{y}(t) = \sum_{j=0}^K \mathbf{y}^{(j)}(t) \frac{h^j}{j!} + \underbrace{\int_t^{t+h} f^{(K+1)}(\tau) \frac{(t+h-\tau)^K}{K!} d\tau}_{= \frac{f^{(K+1)}(\xi)}{K!} h^{K+1}}, \quad (8.3.5)$$

for some $\xi \in [t, t+h]$

\Rightarrow if $\mathbf{y} \in C^2([0, T])$, then

$$\blacktriangleright \quad \mathbf{y}(t_{k+1}) - \mathbf{y}(t_k) = \dot{\mathbf{y}}(t_k) h_k + \frac{1}{2} \ddot{\mathbf{y}}(\xi_k) h_k^2 \quad \text{for some } t_k \leq \xi_k \leq t_{k+1}$$

$$= \mathbf{f}(\mathbf{y}(t_k))h_k + \frac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2 ,$$

since $t \mapsto \mathbf{y}(t)$ solves the ODE, which implies $\dot{\mathbf{y}}(t_k) = \mathbf{f}(\mathbf{y}(t_k))$. This leads to an expression for the one-step error from (8.3.4)

$$\begin{aligned} \tau(h_k, \mathbf{y}(t_k)) &= \Psi^{h_k} \mathbf{y}(t_k) - \mathbf{y}(t_k + h_k) \\ &\stackrel{(8.3.2)}{=} \mathbf{y}(t_k) + h_k \mathbf{f}(\mathbf{y}(t_k)) - \mathbf{y}(t_k) - \mathbf{f}(\mathbf{y}(t_k))h_k + \frac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2 \\ &= \frac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2 . \end{aligned} \tag{8.3.6}$$

Sloppily speaking, we observe $\tau(h_k, \mathbf{y}(t_k)) = O(h_k^2)$ uniformly for $h_k \rightarrow 0$.

③ Estimate for the propagated error from (8.3.3)

$$\begin{aligned} \left\| \Psi^{h_k} \mathbf{y}_k - \Psi^{h_k} \mathbf{y}(t_k) \right\| &= \left\| \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_k) - \mathbf{y}(t_k) - h_k \mathbf{f}(\mathbf{y}(t_k)) \right\| \\ &\stackrel{(8.3.1)}{\leq} (1 + Lh_k) \underbrace{\left\| \mathbf{y}_k - \mathbf{y}(t_k) \right\|} . \end{aligned} \tag{8.3.7}$$

③ *Recursion* for error norms $\epsilon_k := \|\mathbf{e}_k\|$ by \triangle -inequality:

$$\epsilon_{k+1} \leq (1 + h_k L) \epsilon_k + \rho_k , \quad \rho_k := \frac{1}{2} h_k^2 \max_{t_k \leq \tau \leq t_{k+1}} \|\ddot{\mathbf{y}}(\tau)\| . \tag{8.3.8}$$

Taking into account $\epsilon_0 = 0$ this leads to

$$\epsilon_k \leq \sum_{l=1}^k \prod_{j=1}^{l-1} (1 + Lh_j) \rho_l, \quad k = 1, \dots, N. \quad (8.3.9)$$

Use the elementary estimate $(1 + Lh_j) \leq \exp(Lh_j)$ (by convexity of exponential function):

$$(8.3.9) \Rightarrow \epsilon_k \leq \sum_{l=1}^k \prod_{j=1}^{l-1} \exp(Lh_j) \cdot \rho_l = \sum_{l=1}^k \exp\left(L \sum_{j=1}^{l-1} h_j\right) \rho_l.$$

Note: $\sum_{j=1}^{l-1} h_j \leq T$ for final time T

$$\begin{aligned} \blacktriangleright \quad \epsilon_k &\leq \exp(LT) \sum_{l=1}^k \rho_l \leq \exp(LT) \max_k \frac{\rho_k}{h_k} \sum_{l=1}^k h_l \\ &\leq T \exp(LT) \max_{l=1, \dots, k} h_l \cdot \max_{t_0 \leq \tau \leq t_k} \|\ddot{\mathbf{y}}(\tau)\|. \end{aligned}$$

$$\blacktriangleright \quad \|\mathbf{y}_k - \mathbf{y}(t_k)\| \leq T \exp(LT) \max_{l=1, \dots, k} h_l \cdot \max_{t_0 \leq \tau \leq t_k} \|\ddot{\mathbf{y}}(\tau)\|.$$

Total error arises from accumulation of one-step errors!

- error bound = $O(h)$, $h := \max_l h_l$ (► 1st-order **algebraic convergence**)
- Error bound grows *exponentially* with the length T of the integration interval.

Most commonly used single step methods display algebraic convergence of integer order with respect to the meshwidth $h := \max_k h_k$. This offers a criterion for gauging their quality.

The sequence $(\mathbf{y}_k)_k$ generated by a

single step method (→ Def. 8.2.1) of **order** (of consistency) $p \in \mathbb{N}$

for $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ on a mesh $\mathcal{M} := \{t_0 < t_1 < \dots < t_N = T\}$ satisfies

$$\max_k \|\mathbf{y}_k - \mathbf{y}(t_k)\| \leq Ch^p \quad \text{for} \quad h := \max_{k=1, \dots, N} |t_k - t_{k-1}| \rightarrow 0,$$

with $C > 0$ independent of \mathcal{M} , provided that \mathbf{f} is *sufficiently smooth*.

8.4 Structure Preservation

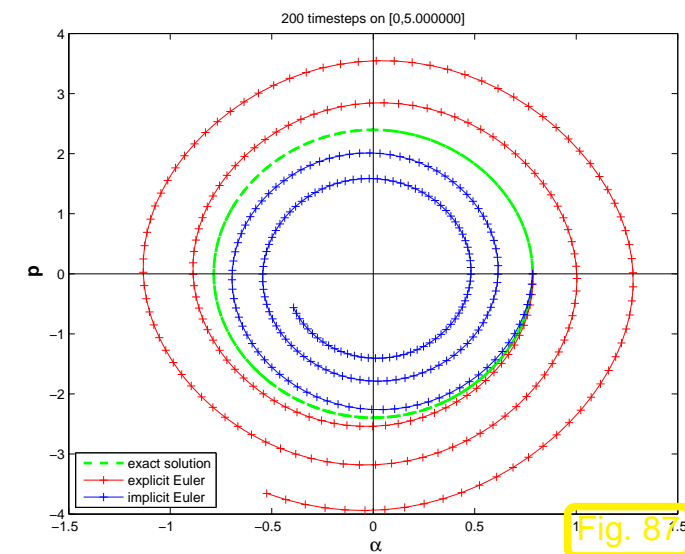
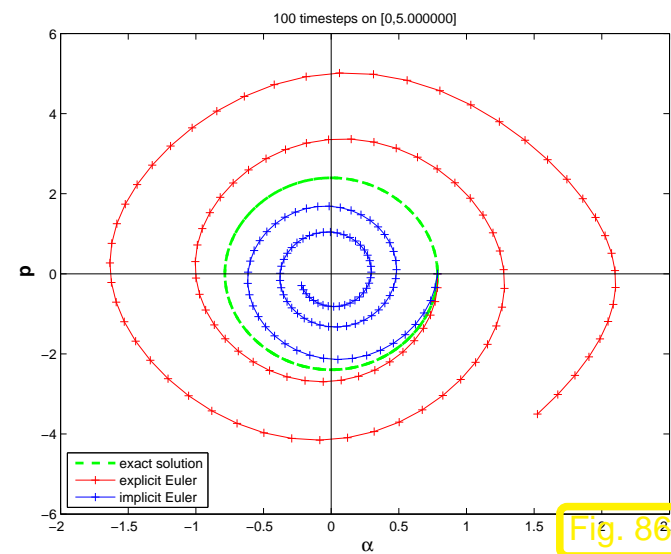
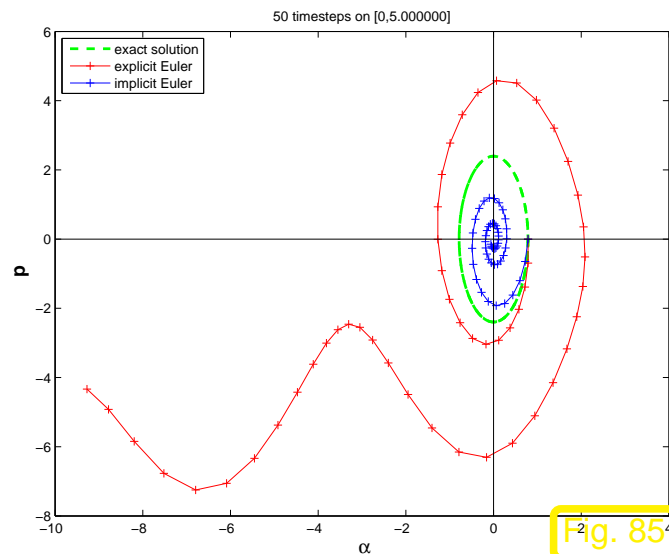
Example 8.4.1 (Euler method for pendulum equation).

Hamiltonian form of equations of motion for pendulum

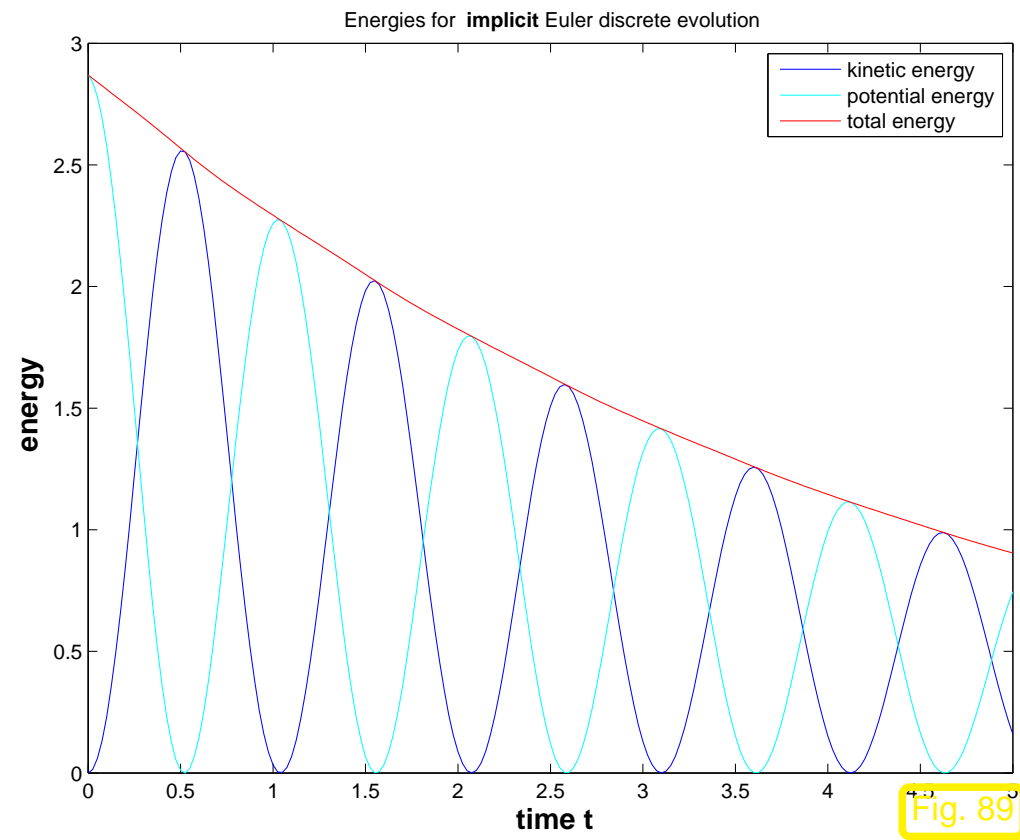
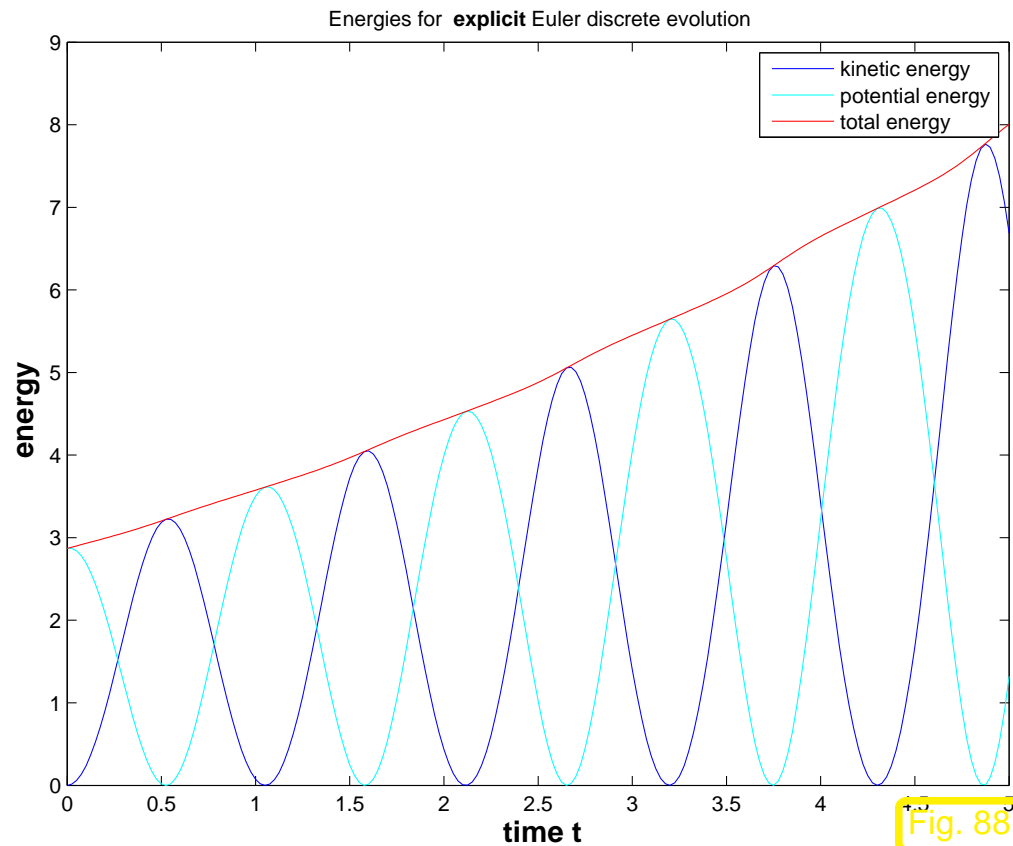
$$\text{angular velocity } p := \dot{\alpha} \Rightarrow \frac{d}{dt} \begin{pmatrix} \alpha \\ p \end{pmatrix} = \begin{pmatrix} p \\ -\frac{g}{l} \sin \alpha \end{pmatrix}, \quad g = 9.8, l = 1. \quad (8.4.1)$$

- numerical solution with explicit/implicit Euler method (8.2.1)/(8.2.4),
- constant time-step $h = T/N$, end time $T = 5$ fixed, $N \in \{50, 100, 200\}$,
- initial value: $\alpha(0) = \pi/4, p(0) = 0$.

Gradinaru
D-MATH



Behavior of the computed energy: kinetic energy : $E_{\text{kin}}(t) = \frac{1}{2}p(t)^2$
 potential energy : $E_{\text{pot}}(t) = -\frac{g}{l} \cos \alpha(t)$



explicit Euler: increase of total energy



implicit Euler: decrease of total energy ("numerical friction")



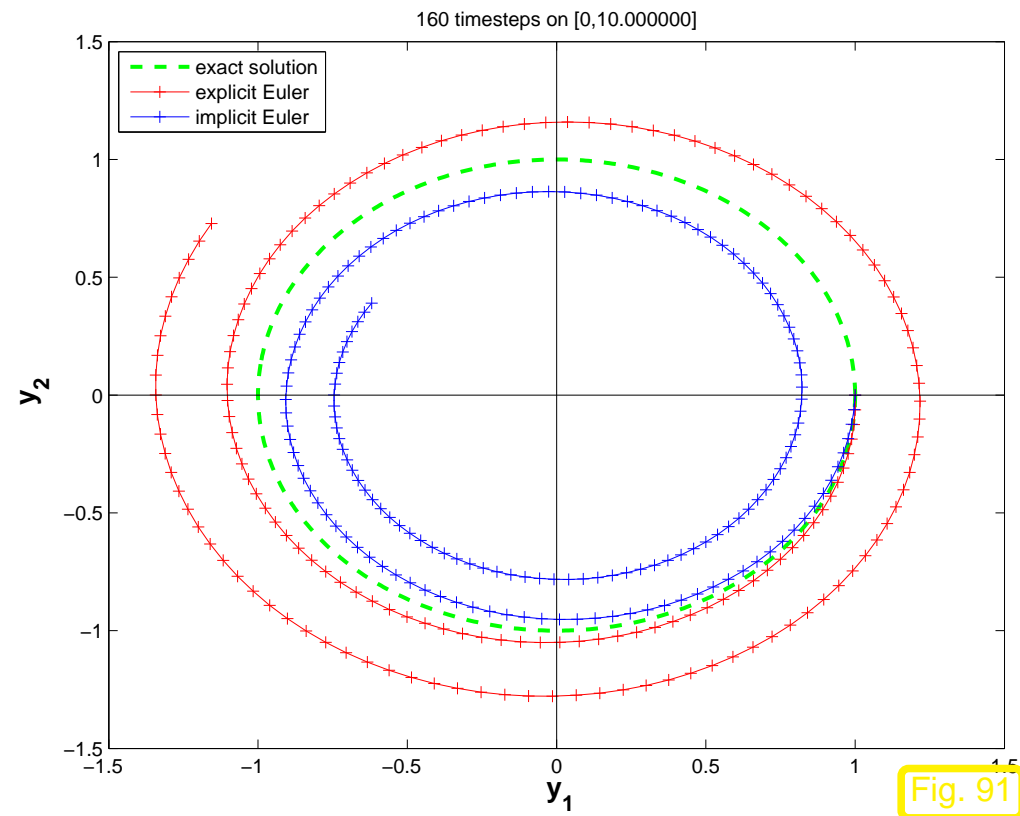
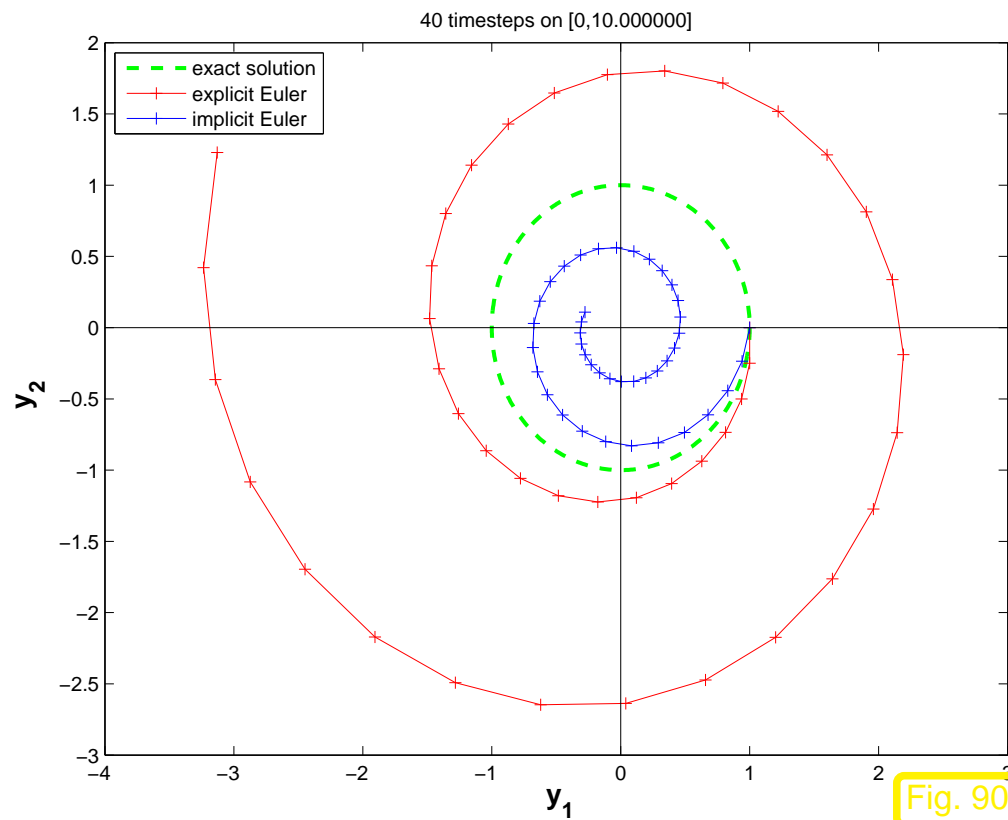
Example 8.4.2 (Euler method for long-time evolution).

Initial value problem for , $D = \mathbb{R}^2$:

$$\dot{\mathbf{y}} = \begin{pmatrix} y_2 \\ -y_1 \end{pmatrix}, \quad \mathbf{y}(0) = \mathbf{y}_0 \quad \blacktriangleright \quad \mathbf{y}(t) = \begin{pmatrix} \cos t & \sin t \\ -\sin t & \cos t \end{pmatrix} \mathbf{y}_0.$$

Note that $I(\mathbf{y}) = \|\mathbf{y}\|$ is constant.

(movement with constant velocity on the circle)



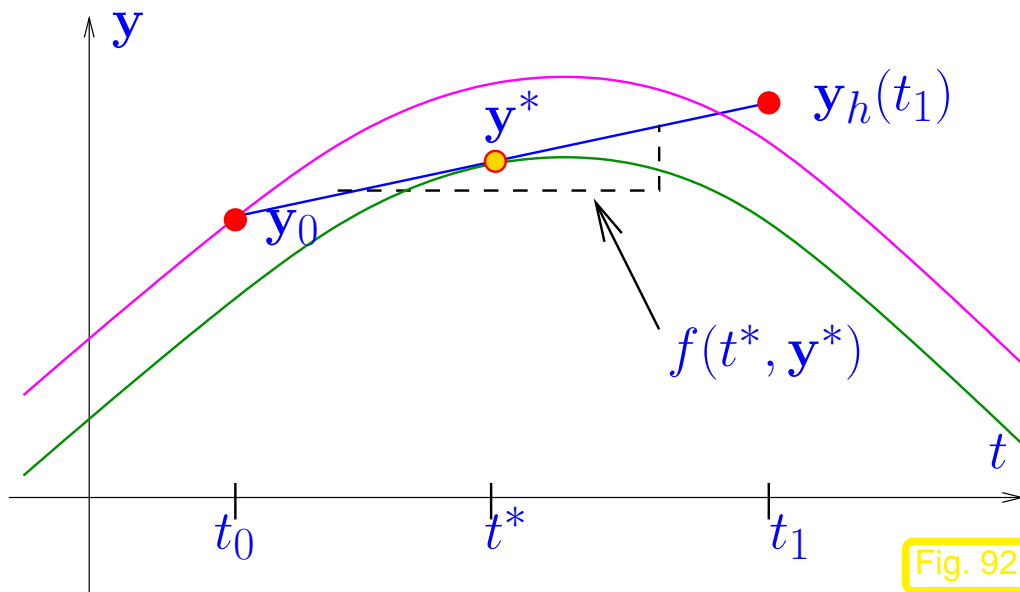
explicit Euler: numerical solution flies away

implicit Euler: numerical solution falls off into the center



8.4.1 Implicit Midpoint Rule

Can we avoid the energy drift ?



Idea: Approximate solution through (t_0, \mathbf{y}_0) on $[t_0, t_1]$ via

- linear polynomial through (t_0, \mathbf{y}_0)
- with slope $f(t^*, \mathbf{y}^*)$,
 $t^* := \frac{1}{2}(t_0 + t_1)$, $\mathbf{y}^* = \frac{1}{2}(\mathbf{y}_0 + \mathbf{y}_1)$

◁ — $\hat{=}$ solution through (t_0, y_0) ,
 — $\hat{=}$ solution through (t^*, y^*) ,
 — $\hat{=}$ tangent at — in (t^*, y^*) .

Apply on small time intervals $[t_0, t_1], [t_1, t_2], \dots, [t_{N-1}, t_N]$ ➤ **implicit midpoint rule**

► via implicit midpoint rule generated approximation \mathbf{y}_{k+1} for $\mathbf{y}(t_k)$ fulfils

$$\mathbf{y}_{k+1} := \mathbf{y}_h(t_{k+1}) = \mathbf{y}_k + h_k \mathbf{f}\left(\frac{1}{2}(t_k + t_{k+1}), \frac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k+1})\right), \quad k = 0, \dots, N-1, \quad (8.4.2)$$

with local **(Time)step** $h_k := t_{k+1} - t_k$.

Note: (8.4.2) requires solution of a (evtl. non-linear) equation for \mathbf{y}_{k+1} !
 (► "implicit")

Remark 8.4.3 (Implicit midpoint rule as difference method).

(8.4.2) from the approximation of time-derivative $\frac{d}{dt}$ via **centred difference quotients** on **time-grid** $\mathcal{G} := \{t_0, t_1, \dots, t_N\}$:

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad \longleftrightarrow \quad \frac{\mathbf{y}_h(t_{k+1}) - \mathbf{y}_h(t_k)}{h_k} = \mathbf{f}\left(\frac{1}{2}(t_k + t_{k+1}), \frac{1}{2}(\mathbf{y}_h(t_k) + \mathbf{y}_h(t_{k+1}))\right), \quad k = 0, \dots, N-1.$$

△

Example 8.4.4 (Implicit midpoint rule for logistic equation).

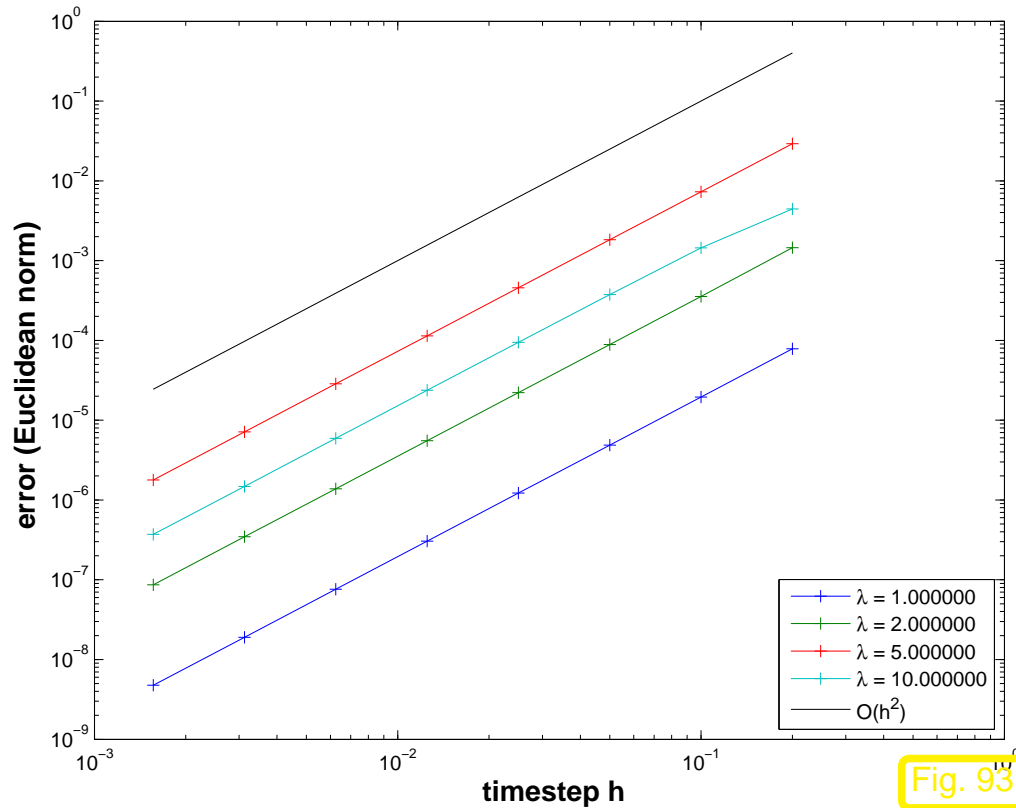


Fig. 93

λ small: $O(h^2)$ -convergence (asymptotical)

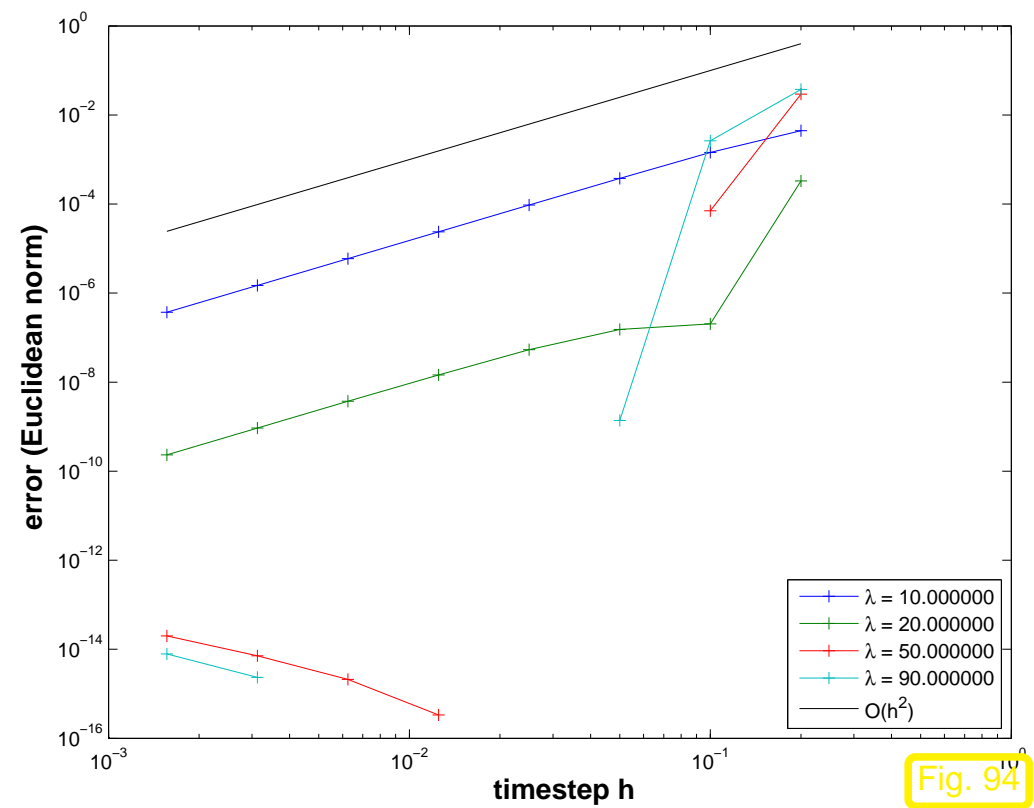
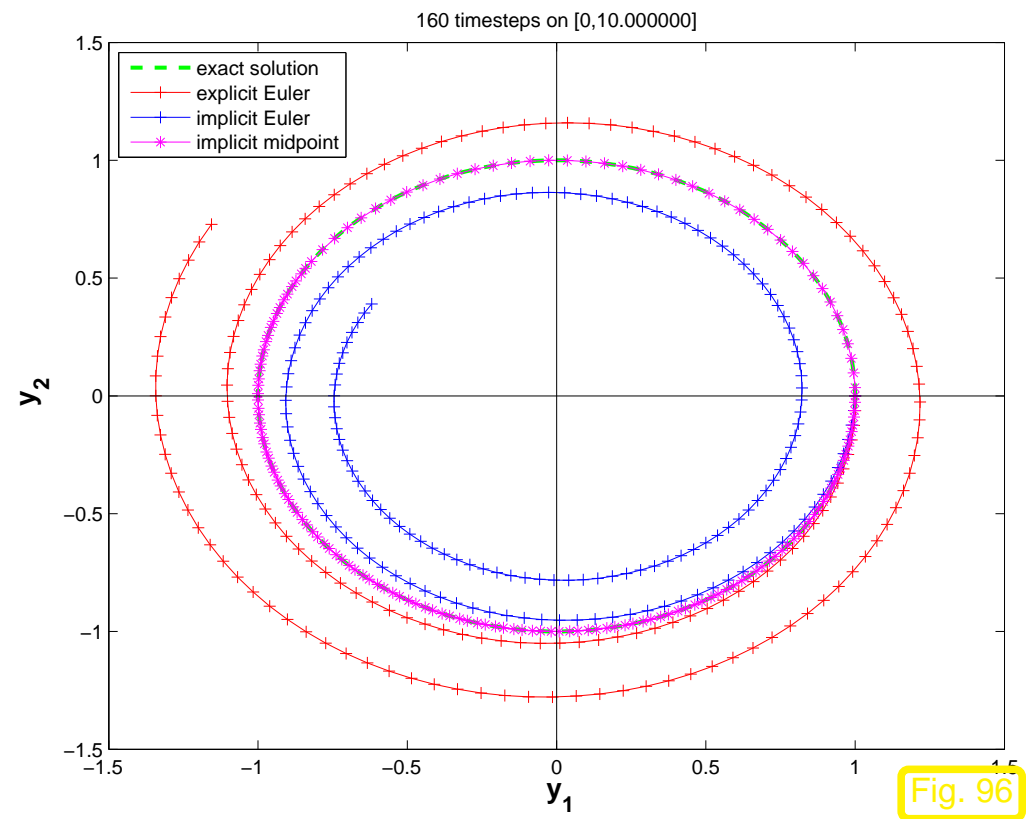
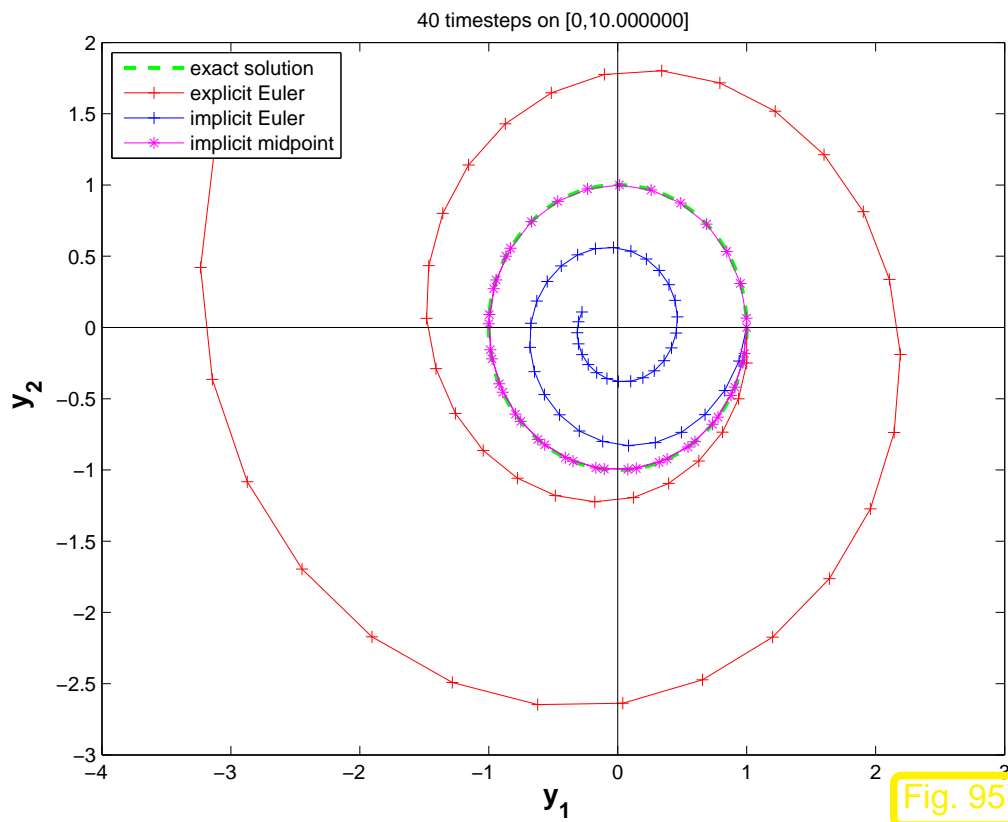


Fig. 94

λ large: stable for all time steps h !



Example 8.4.5 (Implicit midpoint rule for circular motion).



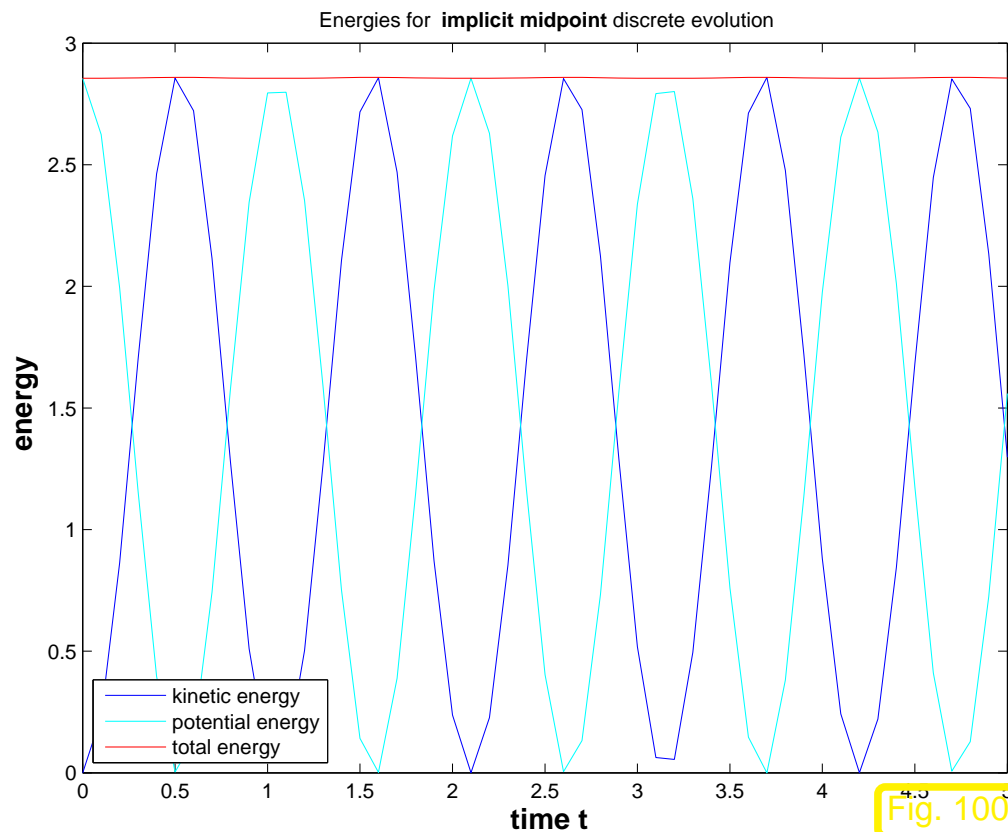
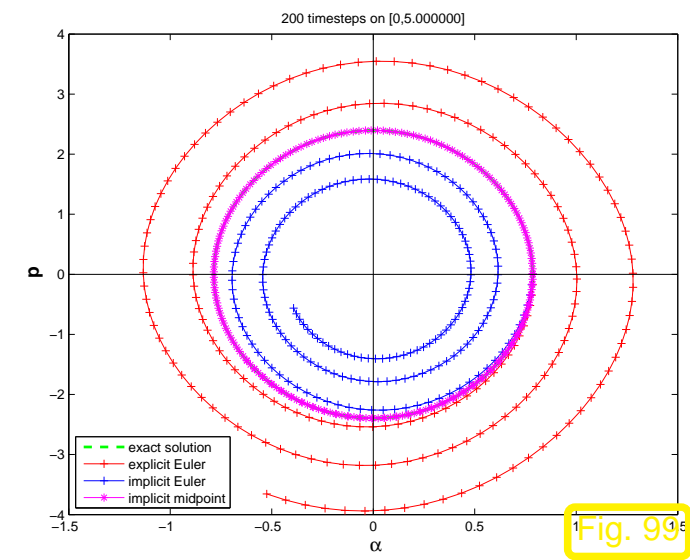
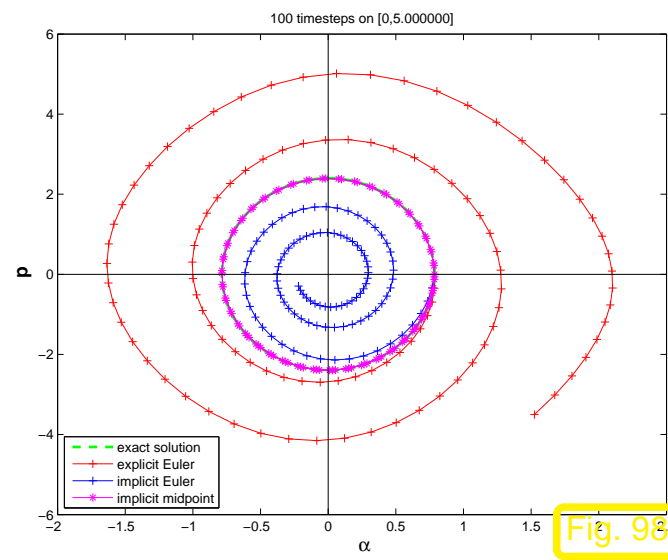
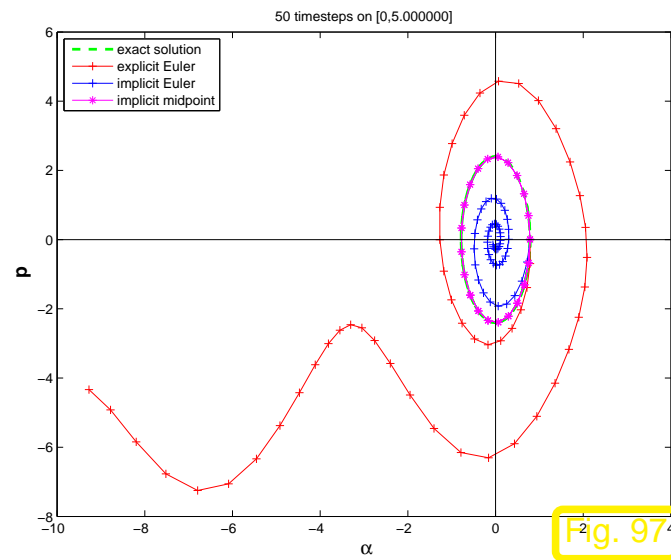
Implicit midpoint rule: perfect conservation of length !

Gradinaru
D-MATH



Example 8.4.6 (Implicit midpoint rule for pendulum).

Initial values and problem as in Bsp. 8.4.1



◁ Behavior of the energy of the numerical solution computed with the midpoint rule (8.4.2), $N = 50$.
No energy drift although large time step)

8.4.2 Störmer-Verlet Method [?]

use the idea of Euler method for an equation of 2nd order

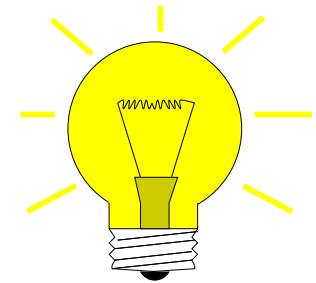
?

$$\ddot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) . \quad (8.4.3)$$

Given $\mathbf{y}_{k-1} \approx \mathbf{y}(t_{k-1})$, $\mathbf{y}_k \approx \mathbf{y}(t_k)$ approximate $\mathbf{y}(t)$ on $[t_{k-1}, t_{k+1}]$ via

- parabola $\mathbf{p}(t)$ through $(t_{k-1}, \mathbf{y}_{k-1})$, (t_k, \mathbf{y}_k) (*),
- with $\ddot{\mathbf{p}}(t_k) = \mathbf{f}(\mathbf{y}_k)$ (*).

(*) \rightarrow Parabola is uniquely determined.



$$\mathbf{y}_{k+1} := \mathbf{p}(t_{k+1}) \approx \mathbf{y}(t_{k+1})$$

Störmer-Verlet method for (8.4.3) (time-grid $\mathcal{G} := \{t_0, t_1, \dots, t_N\}$):

$$\mathbf{y}_{k+1} = -\frac{h_k}{h_{k-1}}\mathbf{y}_{k-1} + \left(1 + \frac{h_k}{h_{k-1}}\right)\mathbf{y}_k + \frac{1}{2}(h_k^2 + h_k h_{k-1})\mathbf{f}(t_k, \mathbf{y}_k) , \quad k = 1, \dots, N-1 . \quad (8.4.4)$$

In case of a fixed time step h :

$$\mathbf{y}_{k+1} = -\mathbf{y}_{k-1} + 2\mathbf{y}_k + h^2\mathbf{f}(t_k, \mathbf{y}_k) , \quad k = 1, \dots, N-1 . \quad (8.4.5)$$

Note: (8.4.4) does not require the solution of an equation (➤ explicit method)

We say: $\mathbf{y}_{k+1} = \mathbf{y}_{k+1}(\mathbf{y}_k, \mathbf{y}_{k-1})$ ➤ (8.4.4) is a **two-step method**
(explicit/implicit Euler method, midpoint rule = **one-step method**)

Remark 8.4.7 (Störmer-Verlet method as difference method).

(8.4.5) from the approximation of the second time-derivative by a **second centered difference quotients** on time-grid $\mathcal{G} := \{t_0, t_1, \dots, t_N\}$: for constant time-step $h > 0$

$$\ddot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad \longleftrightarrow \quad \frac{\frac{\mathbf{y}_h(t_{k+1}) - \mathbf{y}_h(t_k)}{h} - \frac{\mathbf{y}_h(t_k) - \mathbf{y}_h(t_{k-1}))}{h}}{h} = \frac{\mathbf{y}_h(t_{k+1}) - 2\mathbf{y}_h(t_k) + \mathbf{y}_h(t_{k-1}))}{h^2} = \mathbf{f}(\mathbf{y}_h(t_k)) .$$

Gradinaru
D-MATH

△

Remark 8.4.8 (Initialisation of the Störmer-Verlet method).

Initial values for (8.4.3) $\mathbf{y}(0) = \mathbf{y}_0, \dot{\mathbf{y}}(0) = \mathbf{v}_0$

- use virtual moment $t_{-1} := t_0 - h_0$
- apply (8.4.5) to $[t_{-1}, t_1]$:

$$\mathbf{y}_1 = -\mathbf{y}_{-1} + 2\mathbf{y}_0 + h_0^2 \mathbf{f}(t_0, \mathbf{y}_0) . \quad (8.4.6)$$

- centered difference quotient on $[t_{-1}, t_1]$:

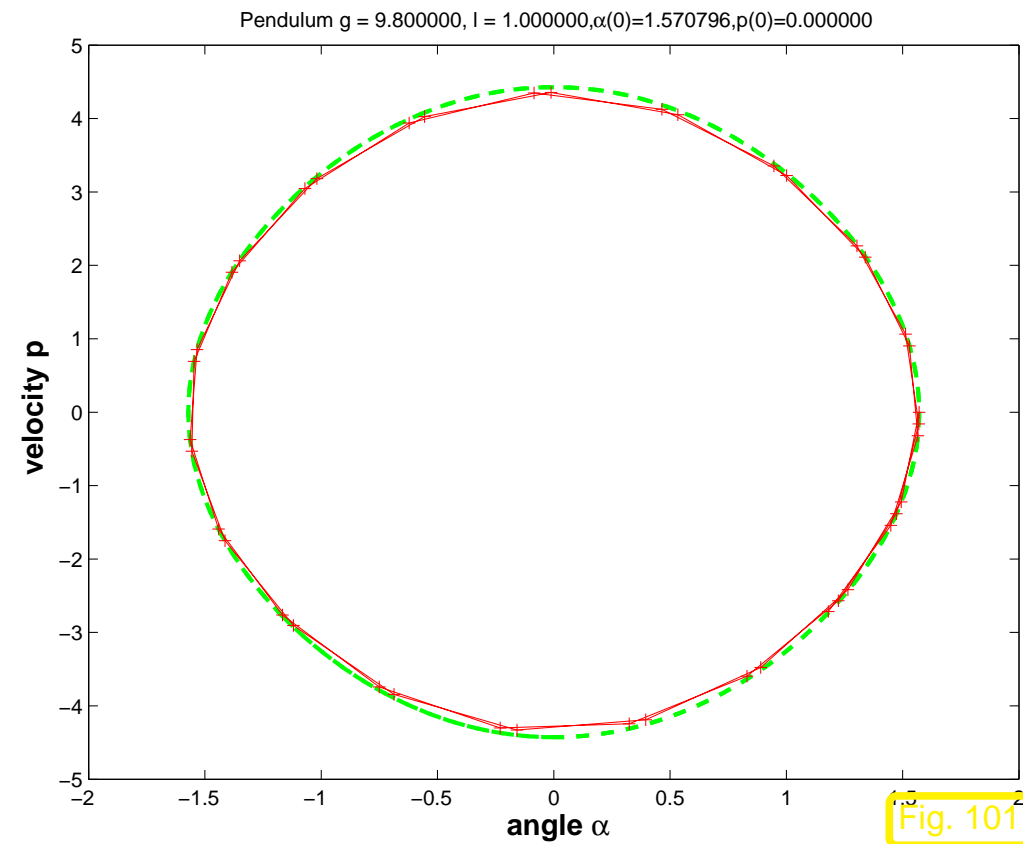
$$\frac{\mathbf{y}_1 - \mathbf{y}_{-1}}{2h_0} = \mathbf{v}_0 . \quad (8.4.7)$$

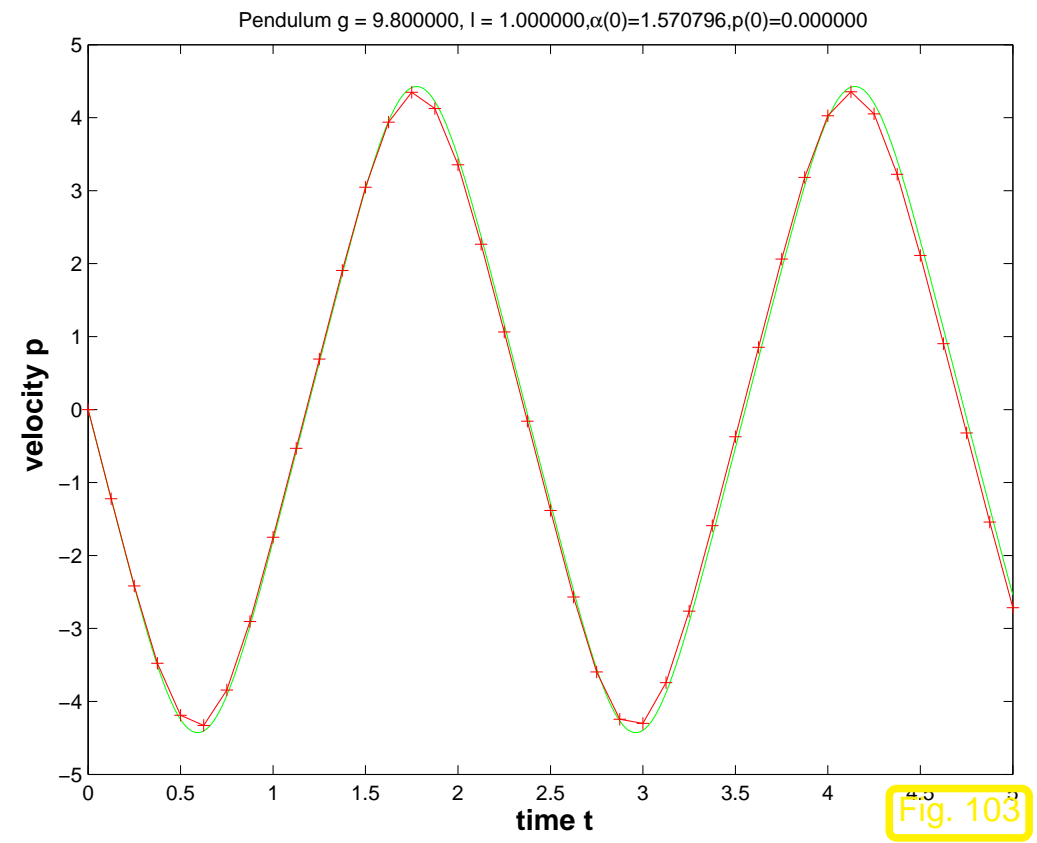
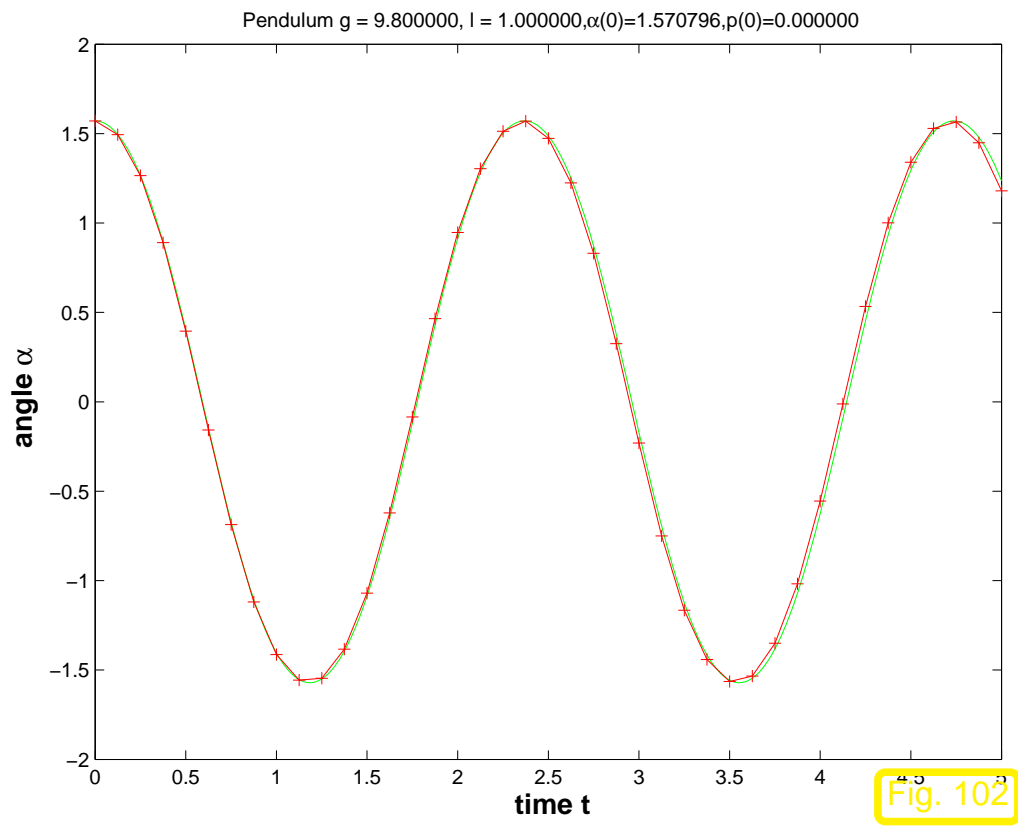
compute \mathbf{y}_1 from (8.4.6) & (8.4.7)

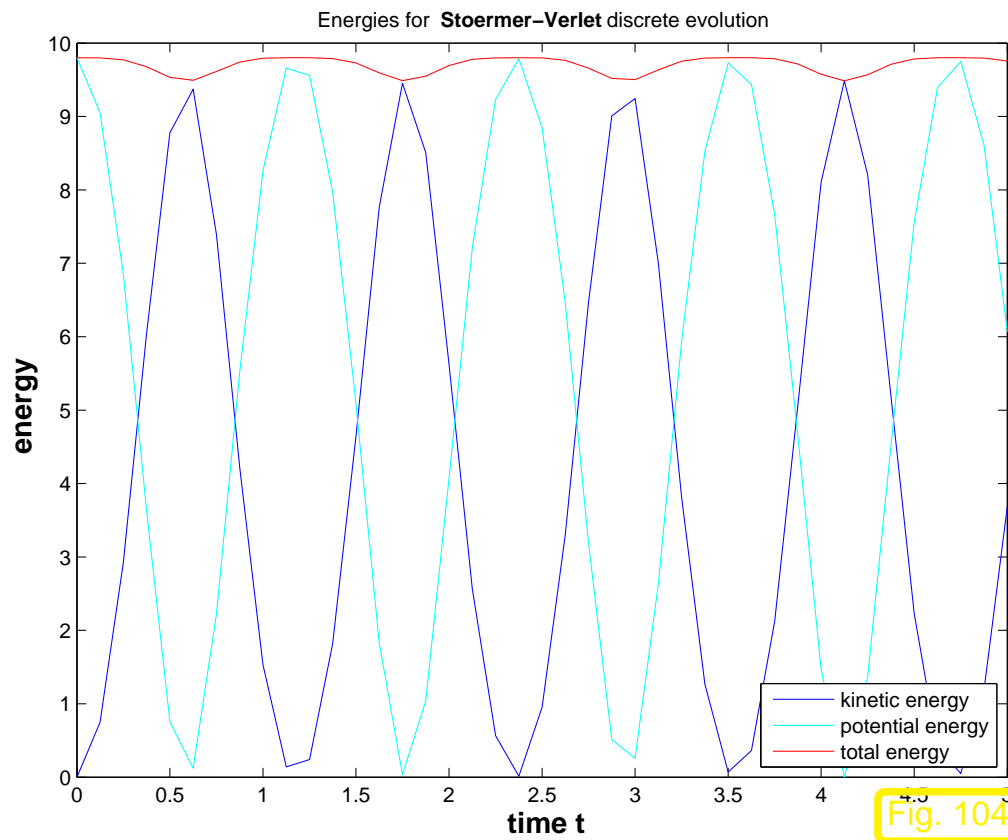


Example 8.4.9 (Störmer-Verlet method for pendulum).

- (8.4.5)
- initialisation via. 8.4.8
- constant time step $h := T/N$, $N \in \mathbb{N}$ time steps
- reference solution via very precise integration `ode45()`
- $\alpha_0 = \pi/2$, $p_0 = 0$, $T = 5$, vgl. Bsp. 8.4.1
- Number of time steps: $N = 40$







➡ No energy drift though large time steps
perfect periodical orbits !

On the contrary: Ex. 8.4.1



Remark 8.4.10 (One-step formulation of the Störmer-Verlet method).

In case of constant time step, see (8.4.5), analogously to the reformulation of a 2nd order equation to

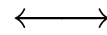
a first order equation, see (8.1.13): with $\mathbf{v}_{k+\frac{1}{2}} := \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h}$

$$\ddot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$$



$$\mathbf{y}_{k+1} - 2\mathbf{y}_k + \mathbf{y}_{k-1} = h_k^2 \mathbf{f}(\mathbf{y}_k)$$

two-steps method



$$\begin{aligned} \dot{\mathbf{y}} &= \mathbf{v} , \\ \dot{\mathbf{v}} &= \mathbf{f}(\mathbf{y}) . \end{aligned}$$



$$\begin{aligned} \mathbf{v}_{k+\frac{1}{2}} &= \mathbf{v}_k + \frac{h}{2} \mathbf{f}(\mathbf{y}_k) , \\ \mathbf{y}_{k+1} &= \mathbf{y}_k + h \mathbf{v}_{k+\frac{1}{2}} , \\ \mathbf{v}_{k+1} &= \mathbf{v}_{k+\frac{1}{2}} + \frac{h}{2} \mathbf{f}(\mathbf{y}_{k+1}) . \end{aligned}$$

one-step method

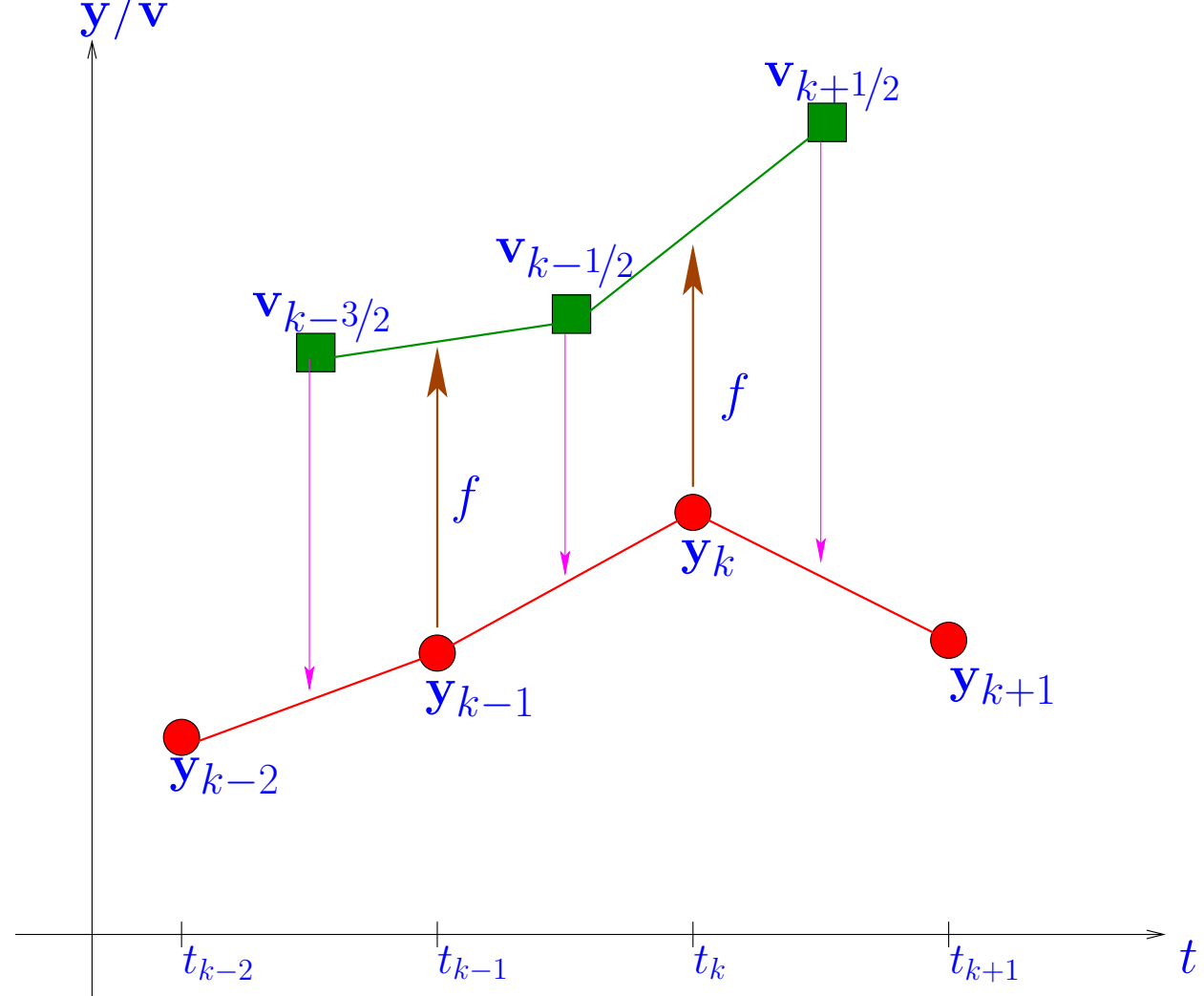
Initialisation (\rightarrow Rem. 8.4.8) is built in already in the formulation.



Remark 8.4.11 (Störmer-Verlet method as polygonal line method).

Perspective: Störmer-Verlet method as
polygonal line method
(see Ren. 8.4.10)

$$\begin{aligned} \mathbf{v}_{k+\frac{1}{2}} &= \mathbf{v}_{k-\frac{1}{2}} + h\mathbf{f}(\mathbf{y}_k) , \\ \mathbf{y}_{k+1} &= \mathbf{y}_k + h\mathbf{v}_{k+\frac{1}{2}} . \end{aligned}$$



"Why so many different numerical methods for ODEs?"

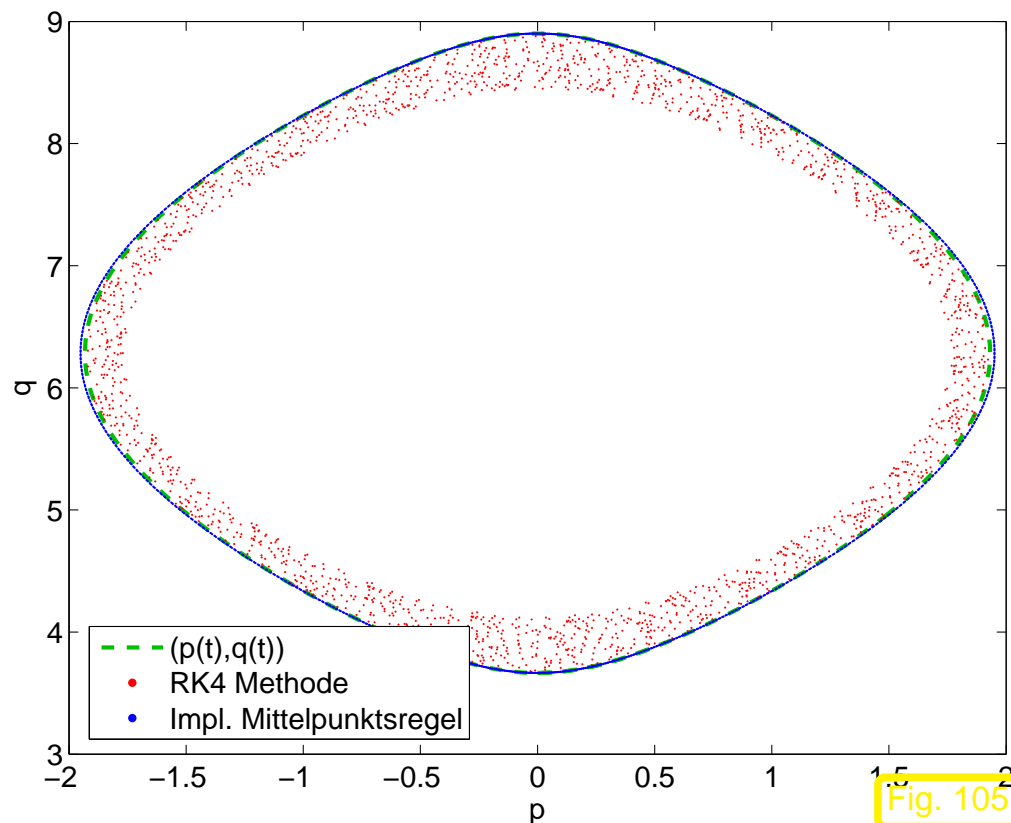
Answer: each numerical integrator has its specific properties
 ➡ best (not) suited for certain classes of initial value problems

8.4.3 Examples

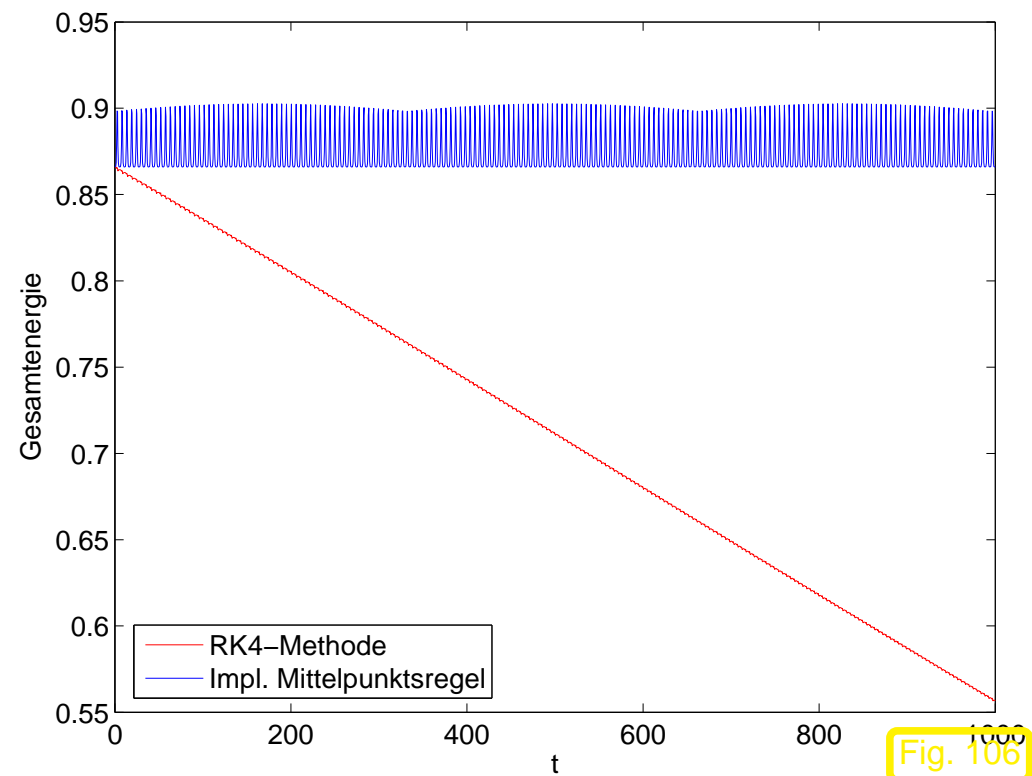
Example 8.4.12 (Energy conservation). \leftrightarrow Bsp. 8.4.6

Pendulum IVP (8.4.1) on $[0, 1000]$, $p(0) = 0$, $q(0) = 7\pi/6$.

Compare classical Runge-Kutta method (8.6.6) (order 4) with implicit midpoint rule 8.4.2), constant time-step $h = \frac{1}{2}$:



Trajectories of "exact" /discrete evolutions



Energy conservation of discrete evolutions



➤ No drift in energy in case of midpoint rule

Fascinating observation

Some (*) numerical time-integrators:

approximative long-time energy conservation (no energy drift)

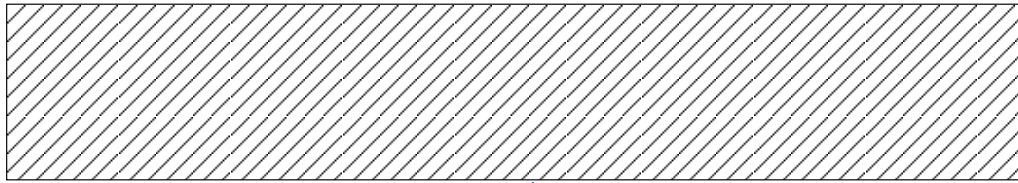
(*) Implicit midpoint rule (8.4.2) → Ex. 8.4.12, 8.4.6,
Störmer-Verlet method (??) → Ex. 8.4.9

Gradinaru
D-MATH

Example 8.4.13 (Spring-pendulum).

Friction-free spring-pendulum: Hamilton function (energy) $H(\mathbf{p}, \mathbf{q}) = \frac{1}{2} \|\mathbf{p}\|^2 + \frac{1}{2}(\|\mathbf{q}\| - 1)^2 + q_2$
($\mathbf{q} \hat{=}$ position, $\mathbf{p} \hat{=}$ momentum)

$$\blacktriangleright \quad \dot{\mathbf{p}} = -(\|\mathbf{q}\| - 1) \frac{\mathbf{q}}{\|\mathbf{q}\|} - \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad \dot{\mathbf{q}} = \mathbf{p}. \quad (8.4.8)$$



Trajectories for long-time evolution
(chaotical mechanical system)

Fig. 107

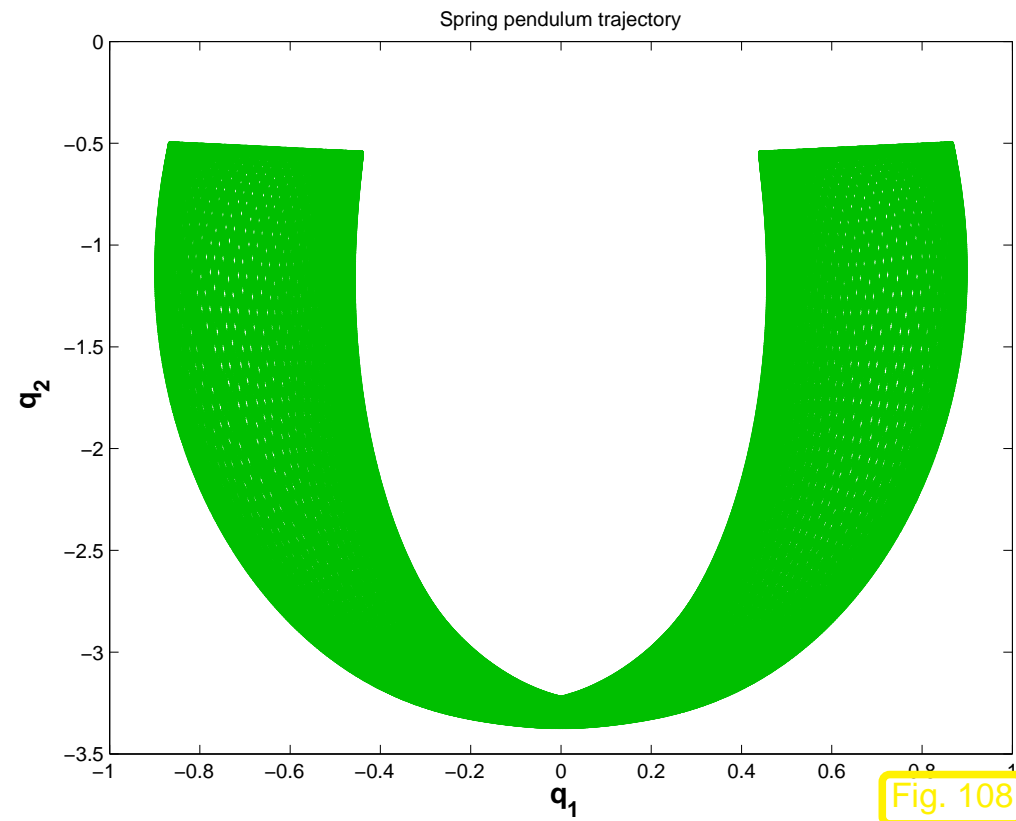
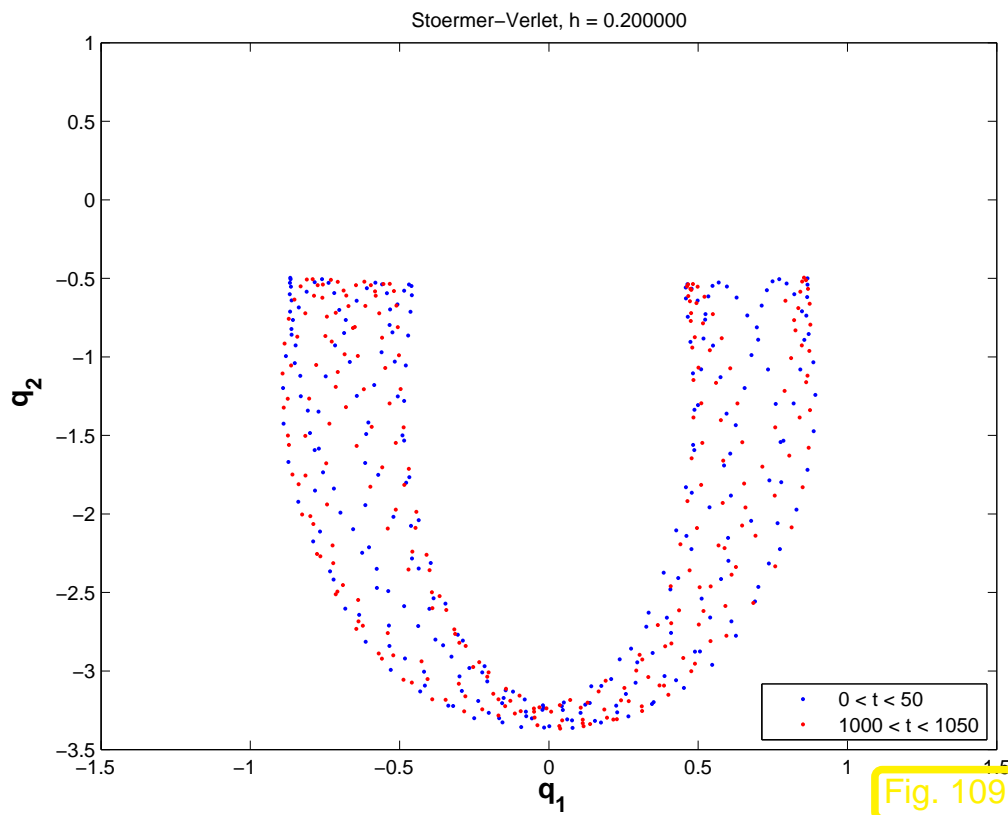
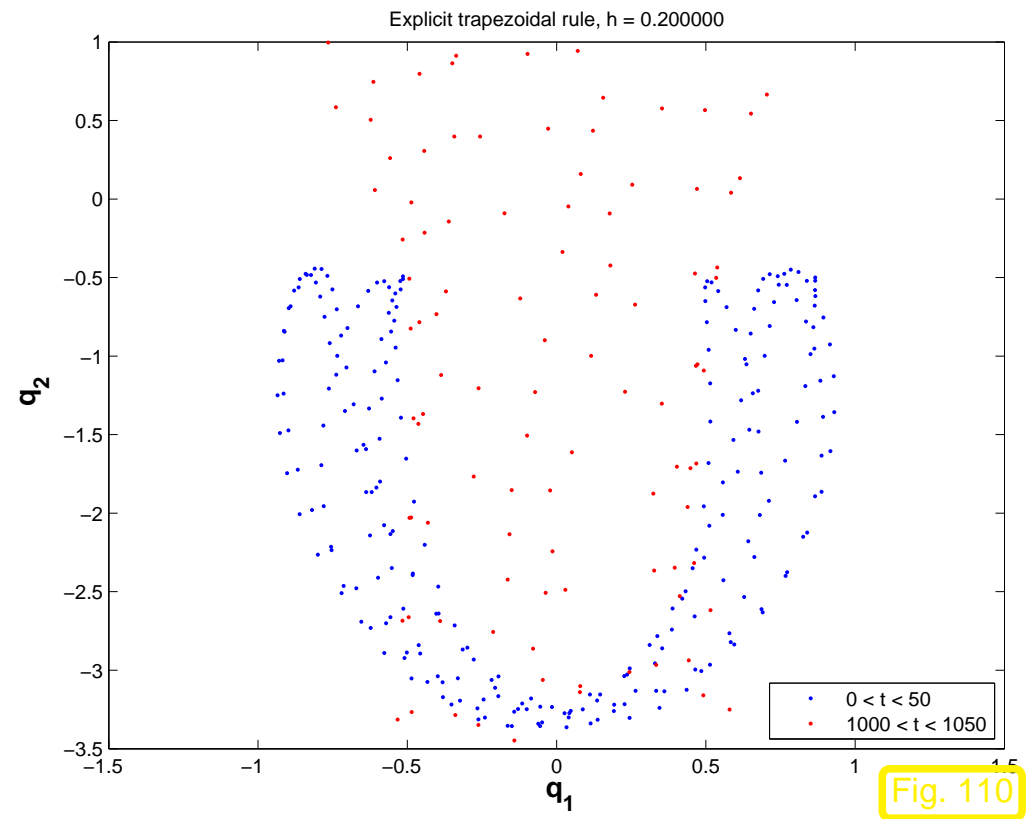


Fig. 108

ESV: • Störmer-Verlet method (8.4.5) (order 2),
• explicit trapezoidal rule (8.6.3) (order 2).



Störmer-Verlet



Explicit trapezoidal rule

Störmer-Verlet: positions in “allowed domain” even for long times

Explicit trapezoidal rule: trajectories leave the “allowed domain” at long times (energy drift !)



Example 8.4.14 (Molecular dynamics). \rightarrow [13, Sect. 1.2]

- Space of states for $n \in \mathbb{N}$ atoms in $d \in \mathbb{N}$ dimensions: $D = \mathbb{R}^{2dn}$
(positions $\mathbf{q} = [\mathbf{q}^1; \dots; \mathbf{q}^n]^T \in \mathbb{R}^{dn}$, Momenta $\mathbf{p} = [\mathbf{p}^1, \dots, \mathbf{p}^n]^T \in \mathbb{R}^{dn}$)

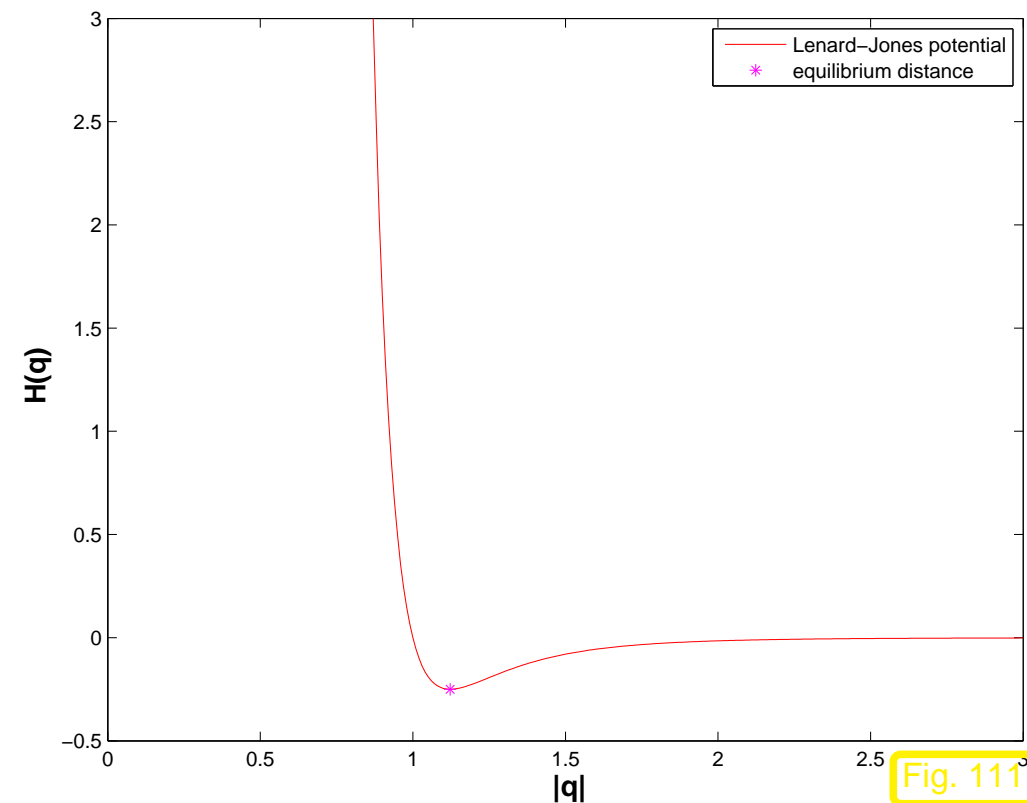
Energy (Hamilton-function):

$$H(\mathbf{p}, \mathbf{q}) = \frac{1}{2} \|\mathbf{p}\|_2^2 + V(\mathbf{q}) .$$

Lenard-Jones-potential:

$$V(\mathbf{q}) = \sum_{j=1}^n \sum_{i \neq j} \mathcal{V}(\|\mathbf{q}^i - \mathbf{q}^j\|_2) ,$$

$$\mathcal{V}(\xi) = \xi^{-12} - \xi^{-6} . \quad (8.4.9)$$



➡ Hamiltonian ODE

$$\dot{\mathbf{p}}^j = - \sum_{i \neq j} \mathcal{V}'(\|\mathbf{q}^j - \mathbf{q}^i\|_2) \frac{\mathbf{q}^j - \mathbf{q}^i}{\|\mathbf{q}^j - \mathbf{q}^i\|_2} , \quad \dot{\mathbf{q}}^j = \mathbf{p}^j , j = 1, \dots, n .$$

► Störmer-Verlet method (8.4.5):

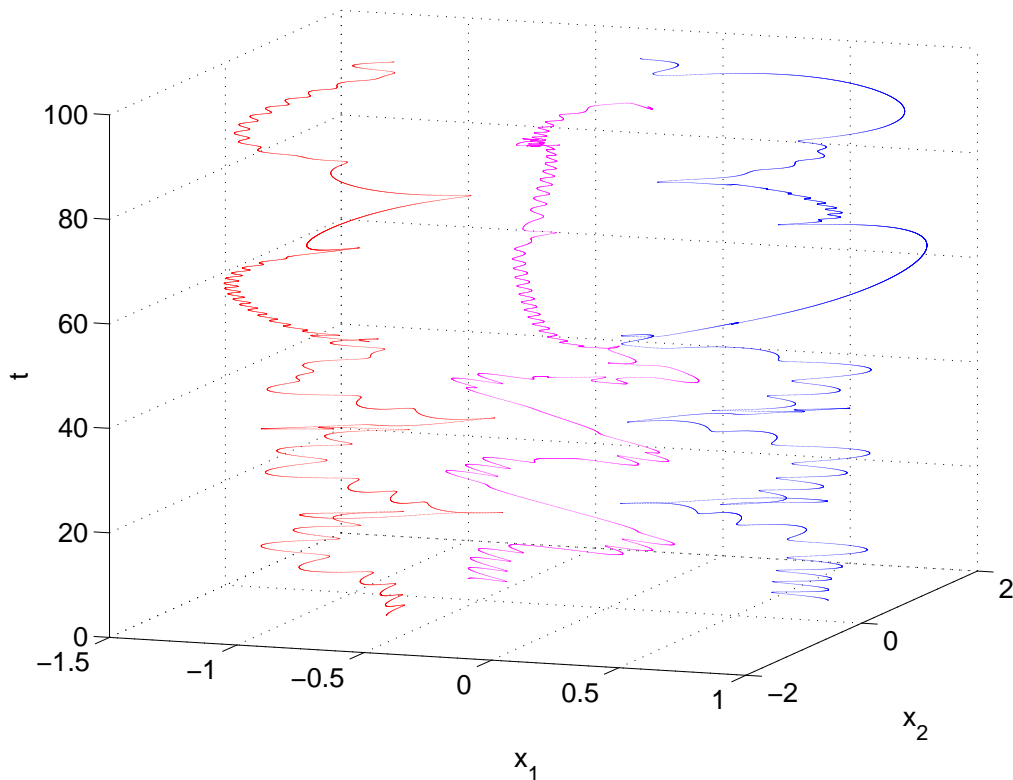
$$\mathbf{q}_h(t + \frac{1}{2}h) = \mathbf{q}_h(t) + \frac{h}{2}\mathbf{p}_h(t) ,$$

$$\mathbf{p}_h^j(t+h) = \mathbf{p}_h^j(t) - h \sum_{i \neq j} \mathcal{V}'(\|\mathbf{q}_h^j(t + \frac{1}{2}h) - \mathbf{q}_h^i(t + \frac{1}{2}h)\|_2) \frac{\mathbf{q}_h^j(t + \frac{1}{2}h) - \mathbf{q}_h^i(t + \frac{1}{2}h)}{\|\mathbf{q}_h^j(t + \frac{1}{2}h) - \mathbf{q}_h^i(t + \frac{1}{2}h)\|_2},$$

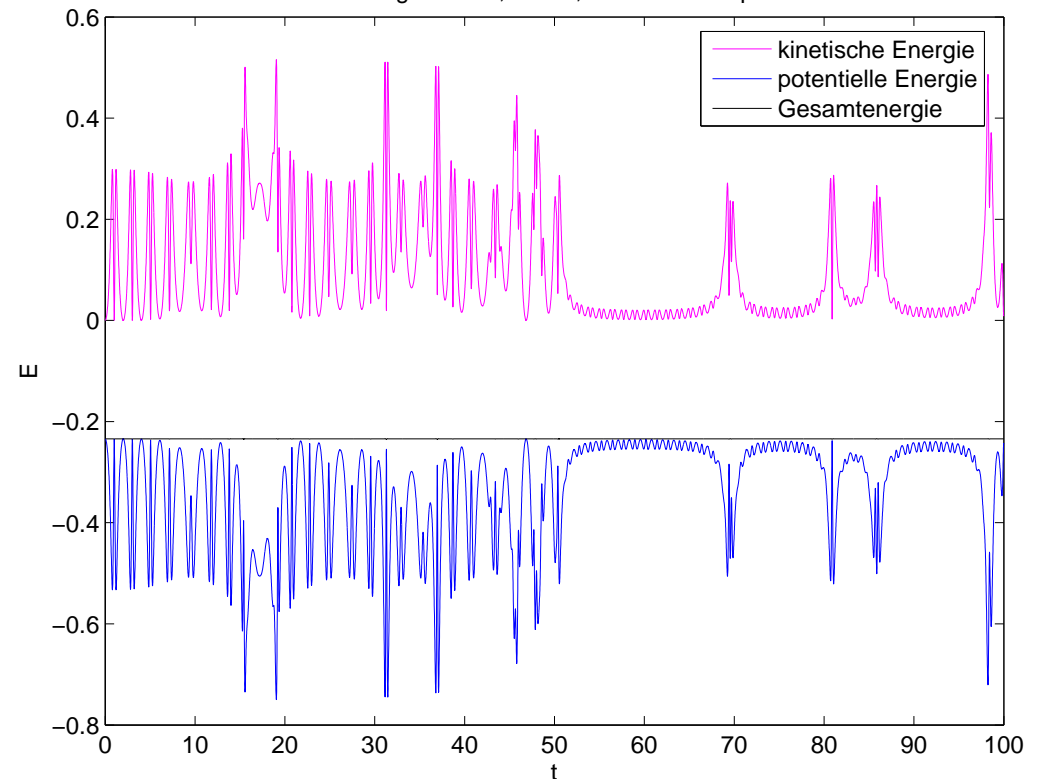
$$\mathbf{q}_h(t+h) = \mathbf{q}_h(t + \frac{1}{2}h) + \frac{h}{2} \mathbf{p}_h(t+h).$$

Simulation with $d = 2$, $n = 3$, $\mathbf{q}^1(0) = \frac{1}{2}\sqrt{2}\begin{pmatrix} -1 \\ 1 \end{pmatrix}$, $\mathbf{q}^2(0) = \frac{1}{2}\sqrt{2}\begin{pmatrix} 1 \\ 1 \end{pmatrix}$, $\mathbf{q}^3(0) = \frac{1}{2}\sqrt{2}\begin{pmatrix} -1 \\ -1 \end{pmatrix}$, $\mathbf{p}(0) = 0$, end-time $T = 100$

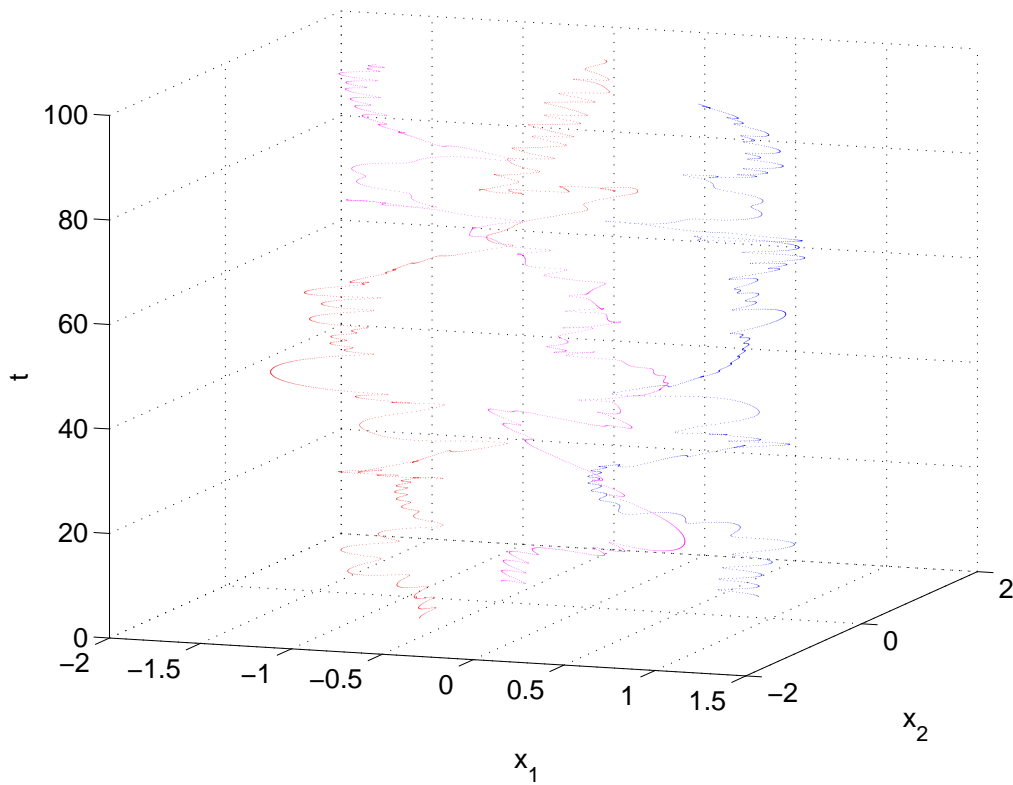
Trajektorien der Atome, Verlet, 10000 timesteps



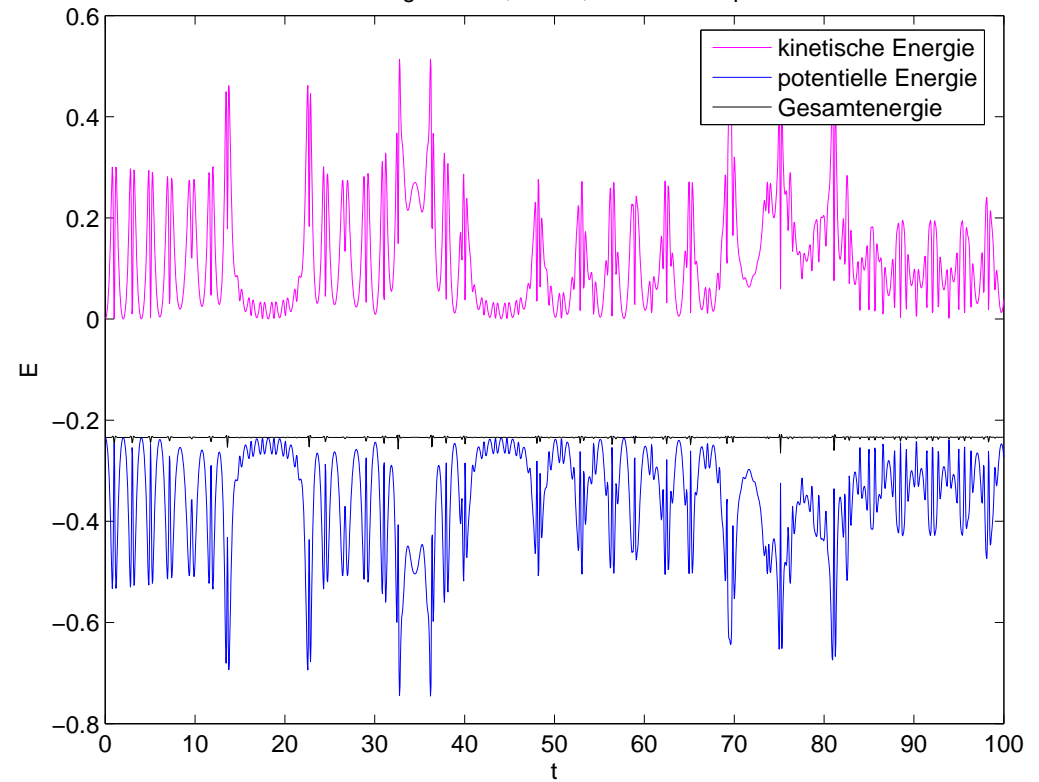
Energieanteile, Verlet, 10000 timesteps



Trajektorien der Atome, Verlet, 2000 timesteps

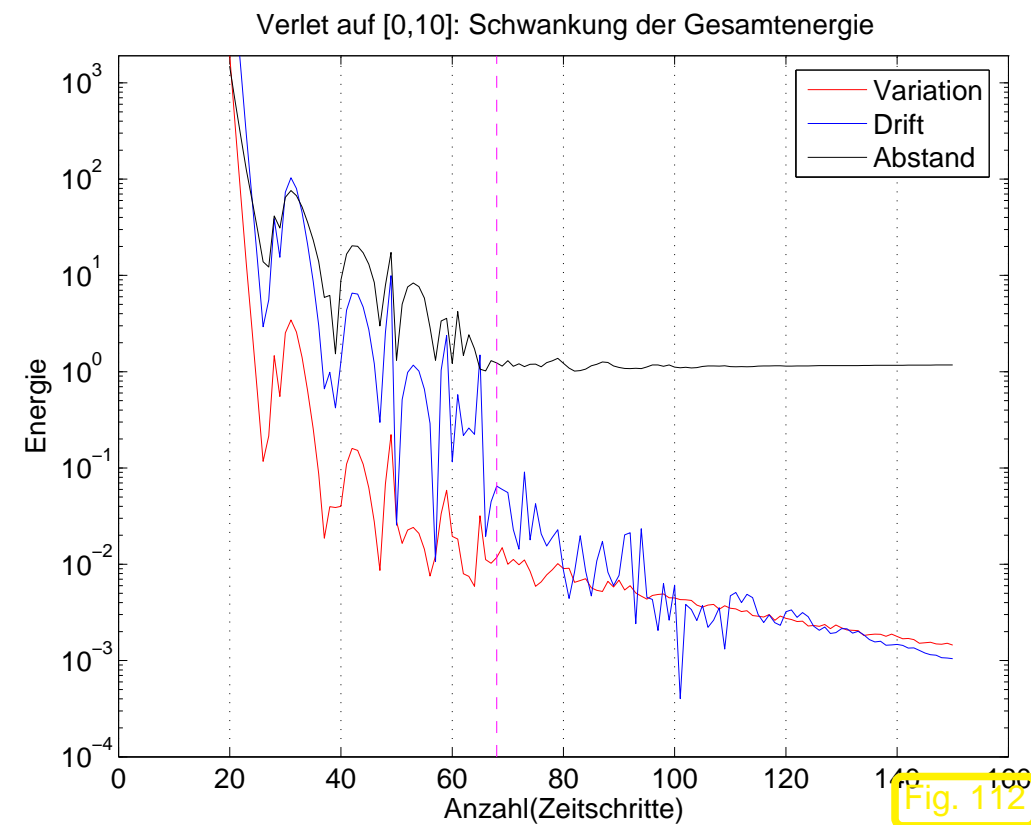


Energieanteile, Verlet, 2000 timesteps



Observations:

- very different trajectories for long-times with different time steps h .
- Qualitatively correct trajectories in any case.



$$T = 10, d = 2, n = 3, \mathbf{q}^1(0) = \frac{1}{2}\sqrt{2}\begin{pmatrix} -1 \\ -1 \end{pmatrix}, \\ \mathbf{q}^2(0) = \frac{1}{2}\sqrt{2}\begin{pmatrix} 1 \\ 1 \end{pmatrix}, \mathbf{q}^3(0) = \frac{1}{2}\sqrt{2}\begin{pmatrix} -1 \\ 1 \end{pmatrix}, \mathbf{p}(0) = 0.$$

$$\text{Variation} = \sum_{i=1}^{N-1} |E_{\text{tot}}((i+1)h) - E_{\text{tot}}(ih)|,$$

$$\text{Drift} = |E_{\text{tot}}(T) - E_{\text{tot}}(0)|,$$

$$\text{Abstand} = \max\{\|\mathbf{q}_h^j(T)\|_2, j = 1, 2, 3\}.$$



Example 8.4.15 (Many particles molecular dynamics). \rightarrow [?, Sect. 4.5.1]

2D conservative system of many particles with
Lennard-Jones-potential \rightarrow Bsp. 8.4.14

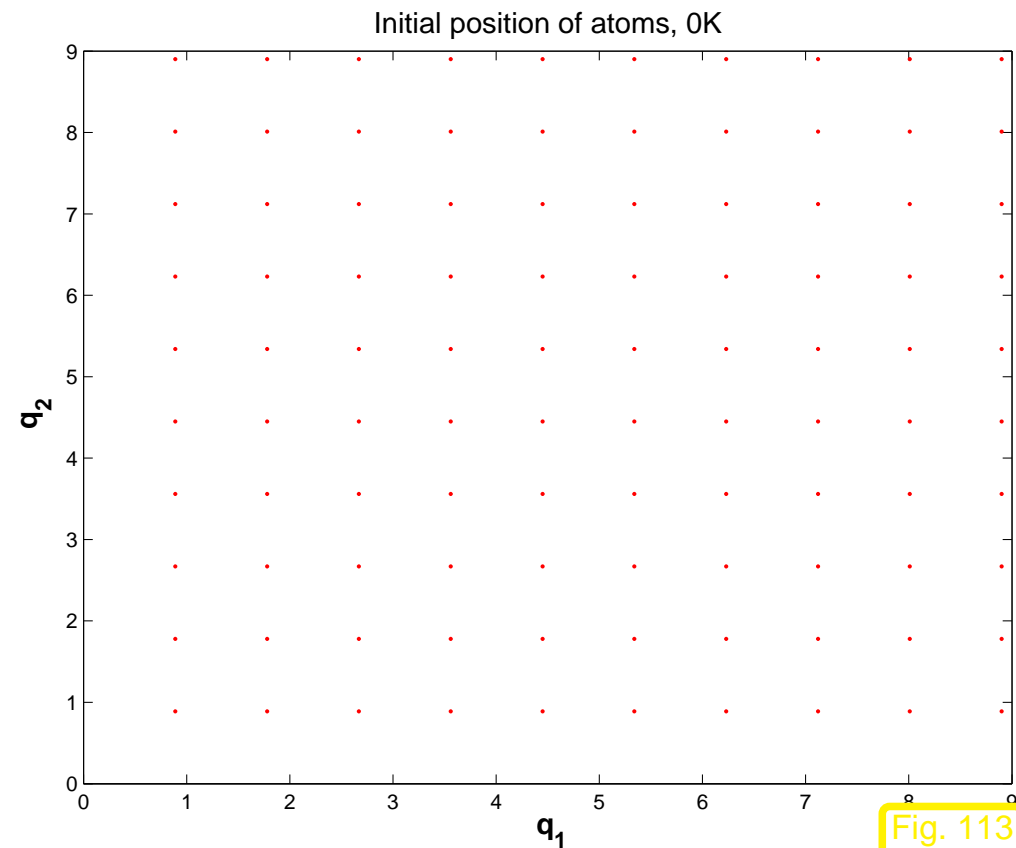
initial positions \triangleright

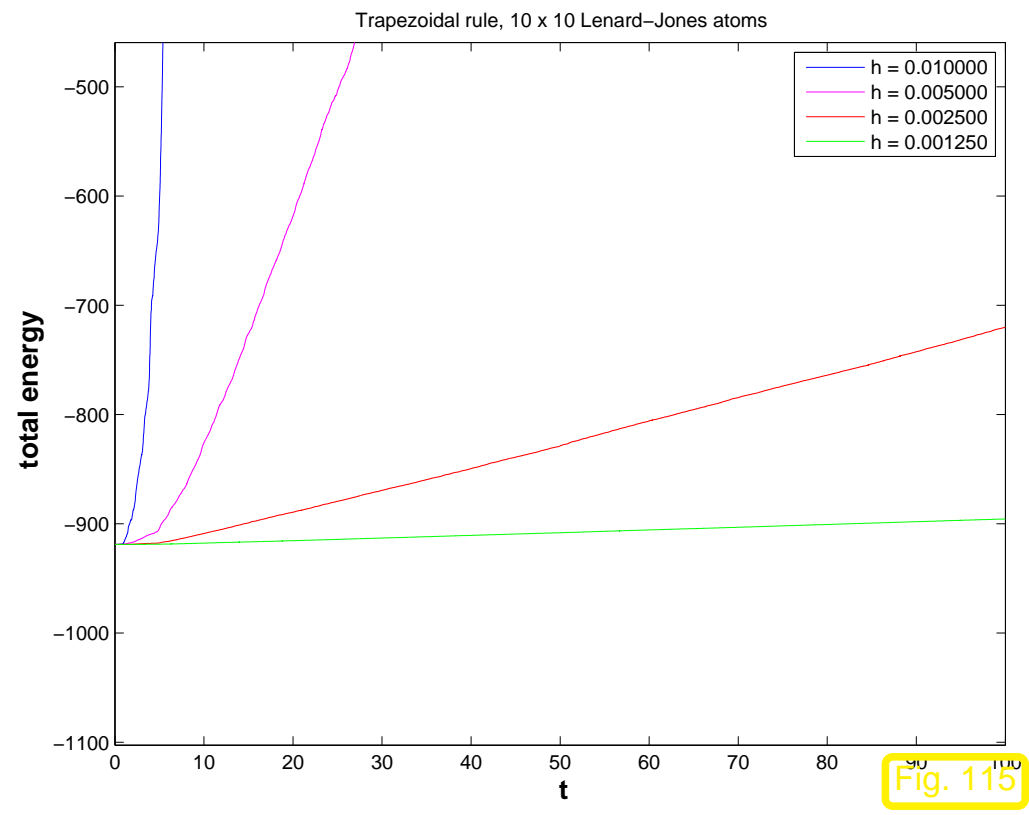
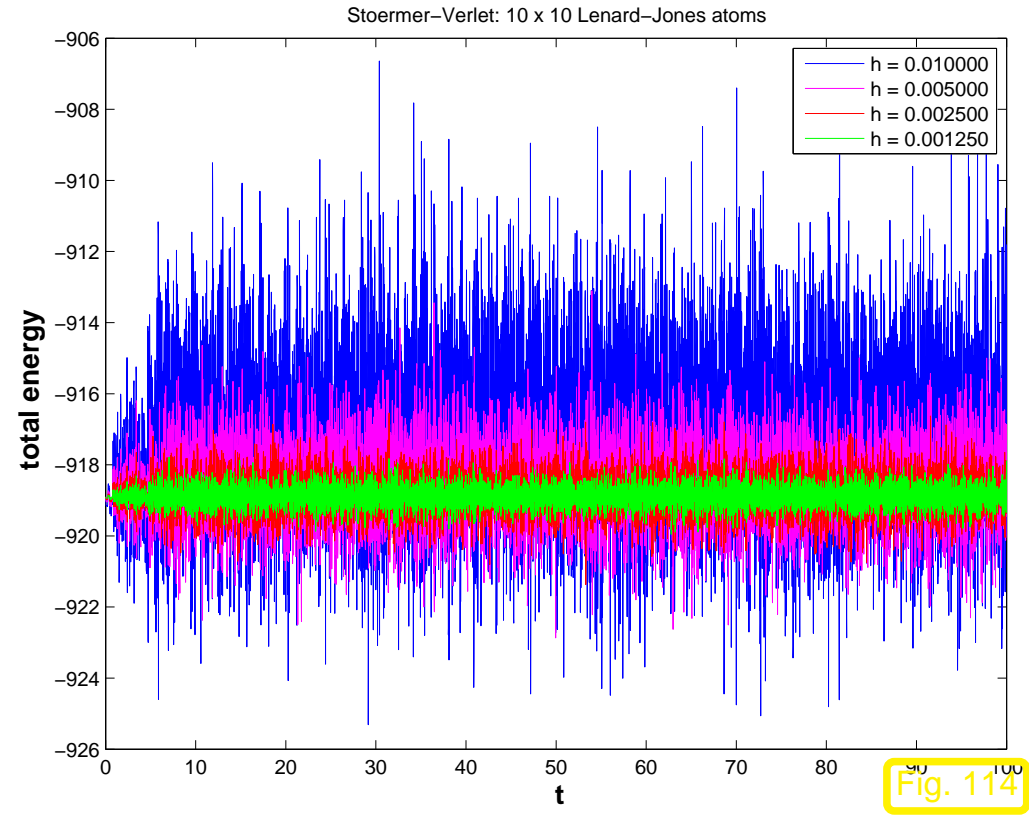
(initial momenta $= 0 \leftrightarrow 0K$)

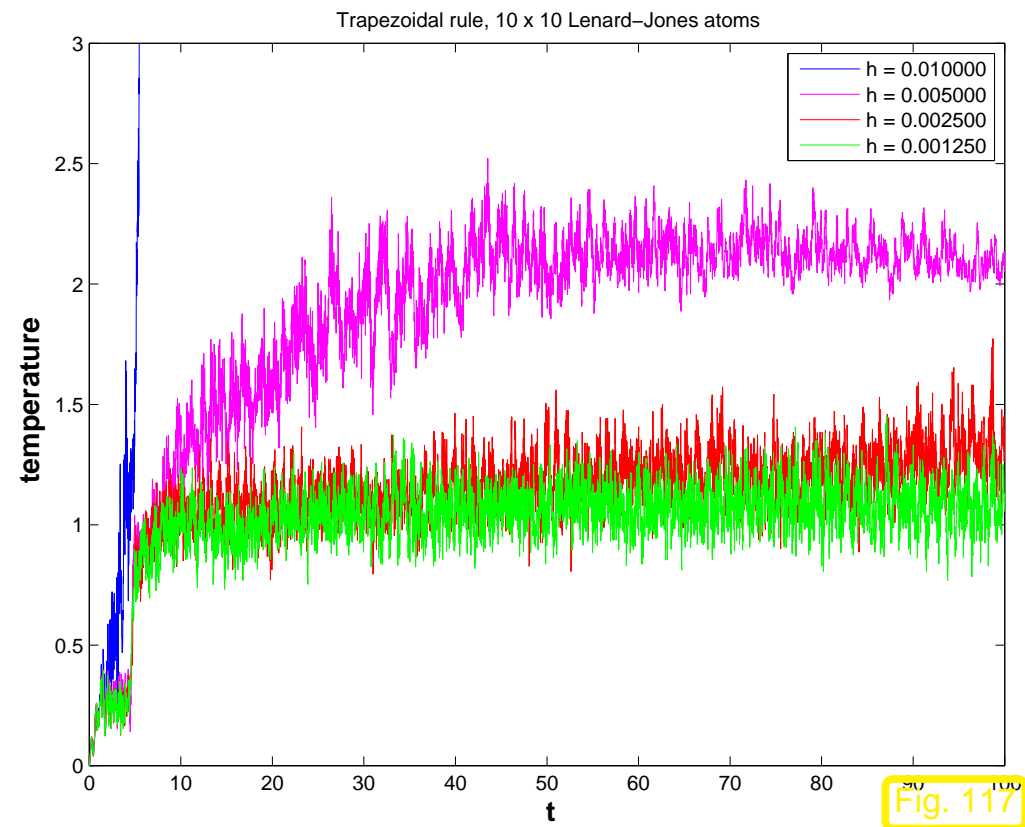
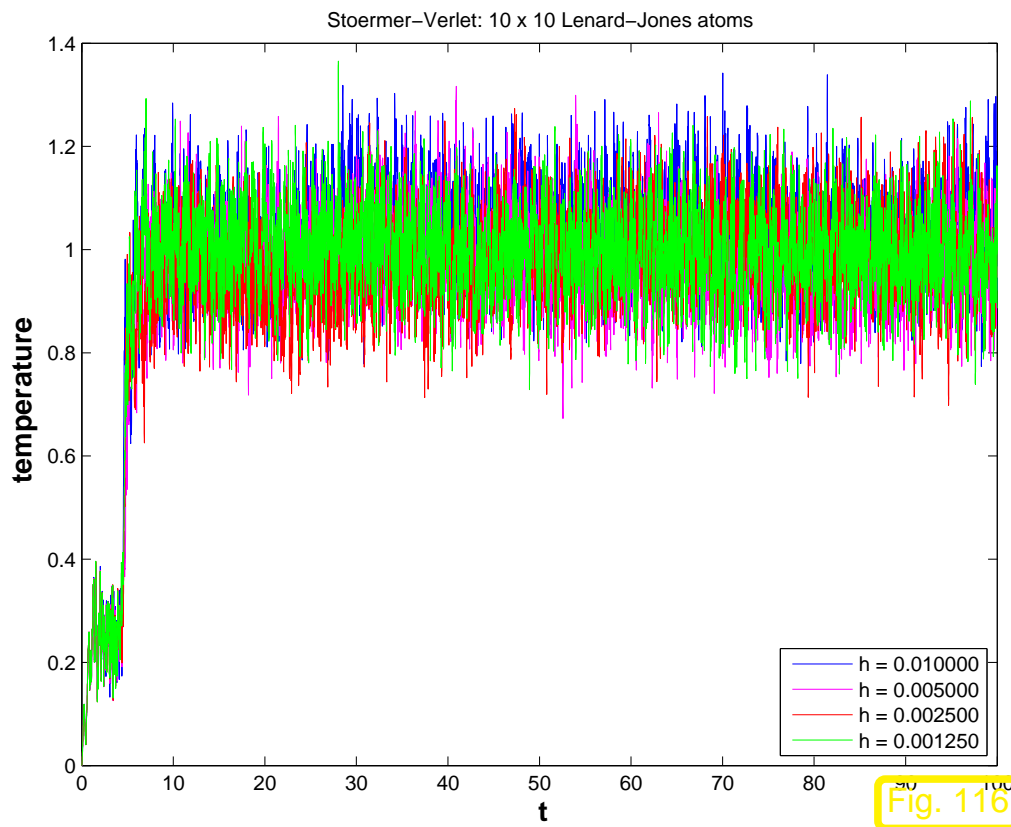
Observations for explicit trapezoidal rule (8.6.3),
Störmer-Verlet (8.4.5)

- Approximation of energy $H(\mathbf{p}, \mathbf{q})$
- middle kinetic energy ("temperature")

Animation \triangleright







Symplectic time-integrator (Verlet): qualitatively correct behaviour of the temperature



8.5 Splitting methods [27, Sect. 2.5]

Autonomous IVP with right hand side of the form:

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) + \mathbf{g}(\mathbf{y}) \quad , \quad \mathbf{y}(0) = \mathbf{y}_0 \quad , \quad (8.5.1)$$

with $\mathbf{f} : D \subset \mathbb{R}^d \mapsto \mathbb{R}^d$, $\mathbf{g} : D \subset \mathbb{R}^d \mapsto \mathbb{R}^d$ “suff. smooth”, locally Lipschitz continuous (\rightarrow Def. 8.1.2)

(Continuous) Evolution:

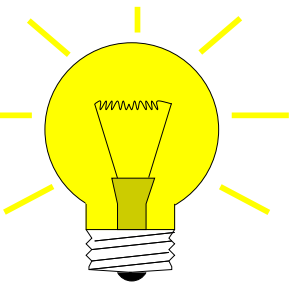
$$\begin{aligned} \Phi_f^t &\leftrightarrow \text{eq. } \dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad , \\ \Phi_g^t &\leftrightarrow \text{eq. } \dot{\mathbf{y}} = \mathbf{g}(\mathbf{y}) \quad . \end{aligned}$$

Assume: Φ_f^t, Φ_g^t (analytically) known

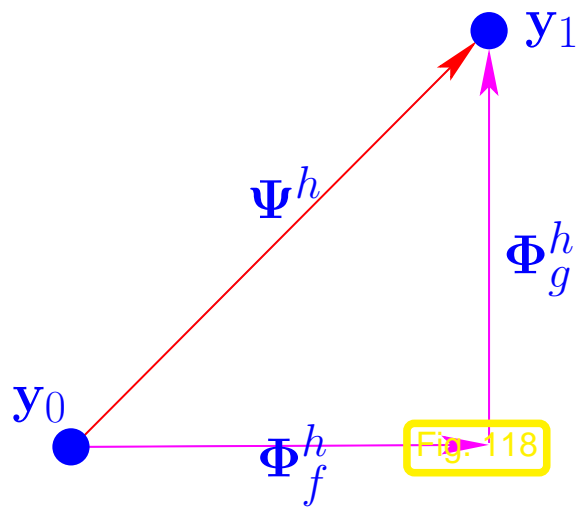
Idea: Construct Single-step method with discrete evolution

Lie-Trotter-Splitting: $\Psi^h = \Phi_g^h \circ \Phi_f^h \quad , \quad (8.5.2)$

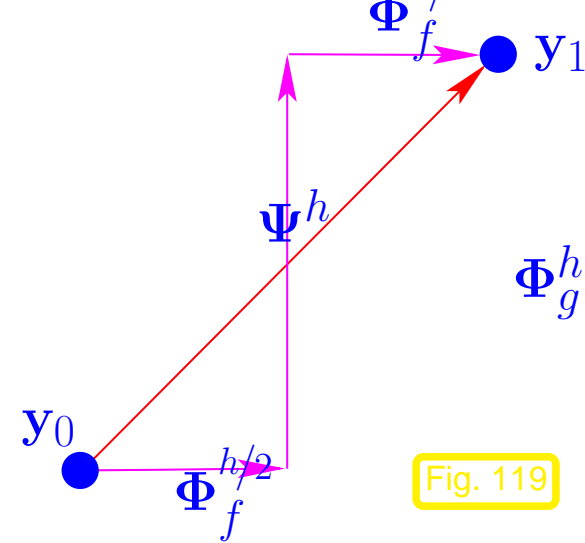
Strang-Splitting: $\Psi^h = \Phi_f^{h/2} \circ \Phi_g^h \circ \Phi_f^{h/2} \quad . \quad (8.5.3)$



(8.5.2) \leftrightarrow



(8.5.3) \leftrightarrow

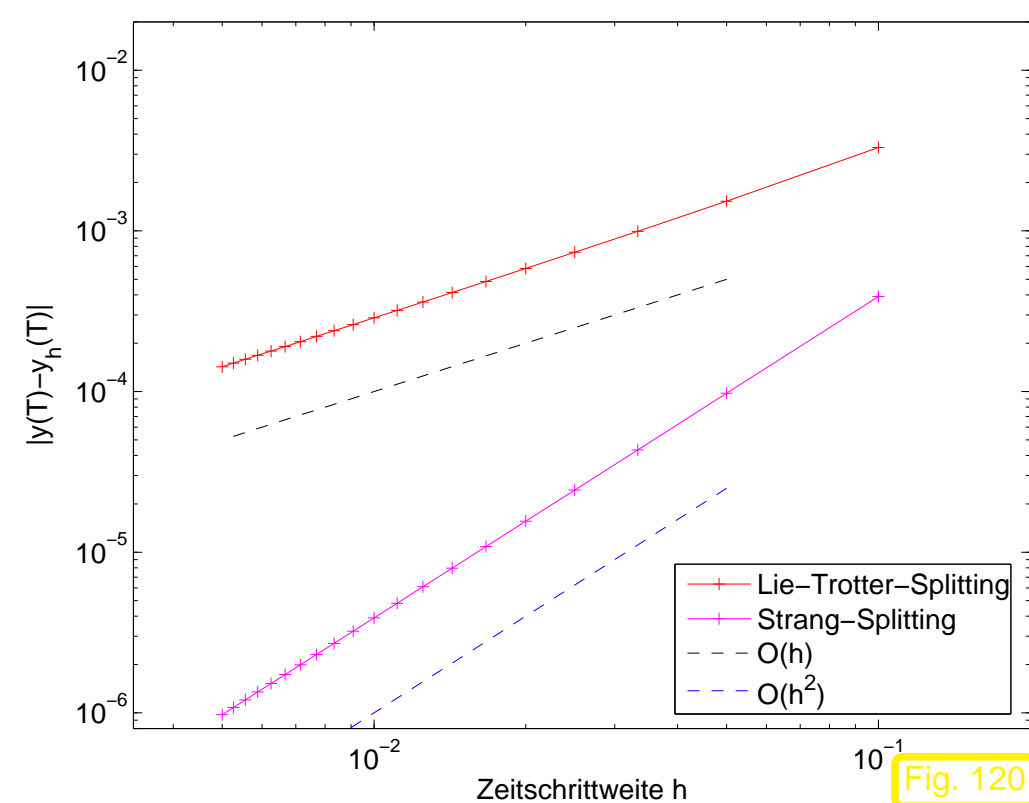


Example 8.5.1 (Convergence of simple splitting methods).

$$\dot{y} = \underbrace{\lambda y(1 - y)}_{=:f(y)} + \underbrace{\sqrt{1 - y^2}}_{=:g(y)}, \quad y(0) = 0.$$

► $\Phi_f^t y = \frac{1}{1 + (y^{-1} - 1)e^{-\lambda t}}, \quad t > 0, y \in]0, 1]$ (logistic differential equation (8.1.3))

► $\Phi_g^t y = \begin{cases} \sin(t + \arcsin(y)) & , \text{ if } t + \arcsin(y) < \frac{\pi}{2} , \\ 1 & , \text{ else,} \end{cases} \quad t > 0, y \in [0, 1] .$



Numerical experiment:

$T = 1$, $\lambda = 1$, compare splitting method (constant time-step) with very precise numerical solution

◁ Error at end-time $T = 1$



Example 8.5.2 (Splitting methods for mechanical systems).

Newton eq. of motion $\ddot{\mathbf{r}} = a(\mathbf{r}) \quad \stackrel{(8.1.13)}{\Longleftrightarrow} \quad \dot{\mathbf{y}} := \begin{pmatrix} \dot{\mathbf{r}} \\ \dot{\mathbf{v}} \end{pmatrix} = \begin{pmatrix} \mathbf{v} \\ a(\mathbf{r}) \end{pmatrix} =: \mathbf{F}(\mathbf{y}) .$

Splitting:
$$\mathbf{F}(\mathbf{y}) = \underbrace{\begin{pmatrix} 0 \\ a(\mathbf{r}) \end{pmatrix}}_{=: \mathbf{f}(\mathbf{y})} + \underbrace{\begin{pmatrix} \mathbf{v} \\ 0 \end{pmatrix}}_{=: \mathbf{g}(\mathbf{y})} .$$

►
$$\Phi_f^t \begin{pmatrix} \mathbf{r}_0 \\ \mathbf{v}_0 \end{pmatrix} = \begin{pmatrix} \mathbf{r}_0 \\ \mathbf{v}_0 + ta(\mathbf{r}_0) \end{pmatrix} , \quad \Phi_g^t \begin{pmatrix} \mathbf{r}_0 \\ \mathbf{v}_0 \end{pmatrix} = \begin{pmatrix} \mathbf{r}_0 + t\mathbf{v}_0 \\ \mathbf{v}_0 \end{pmatrix} .$$

► Lie-Trotter-Splitting (8.5.2): ➤ **Symplectic Euler method**

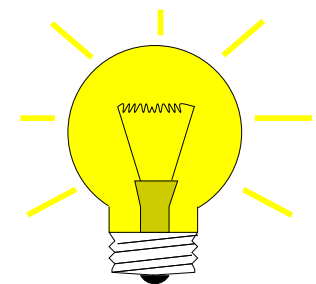
$$\Psi^h \begin{pmatrix} \mathbf{r} \\ \mathbf{v} \end{pmatrix} = \left(\Phi_g^h \circ \Phi_f^h \right) \begin{pmatrix} \mathbf{r} \\ \mathbf{v} \end{pmatrix} = \begin{pmatrix} \mathbf{r} + h(\mathbf{v} + ha(\mathbf{r})) \\ \mathbf{v} + ha(\mathbf{r}) \end{pmatrix}. \quad (8.5.4)$$

► Strang-Splitting (8.5.3):

$$\Psi^h \begin{pmatrix} \mathbf{r} \\ \mathbf{v} \end{pmatrix} = \left(\Phi_g^{h/2} \circ \Phi_f^h \circ \Phi_g^{h/2} \right) \begin{pmatrix} \mathbf{r} \\ \mathbf{v} \end{pmatrix} = \begin{pmatrix} \mathbf{r} + h\mathbf{v} + \frac{1}{2}h^2a(\mathbf{r} + \frac{1}{2}h\mathbf{v}) \\ \mathbf{v} + ha(\mathbf{r} + \frac{1}{2}h\mathbf{v}) \end{pmatrix}. \quad (8.5.5)$$

= single-step formulation of Störmer-Verlet methods (8.4.5), see Rem. 8.4.10 !

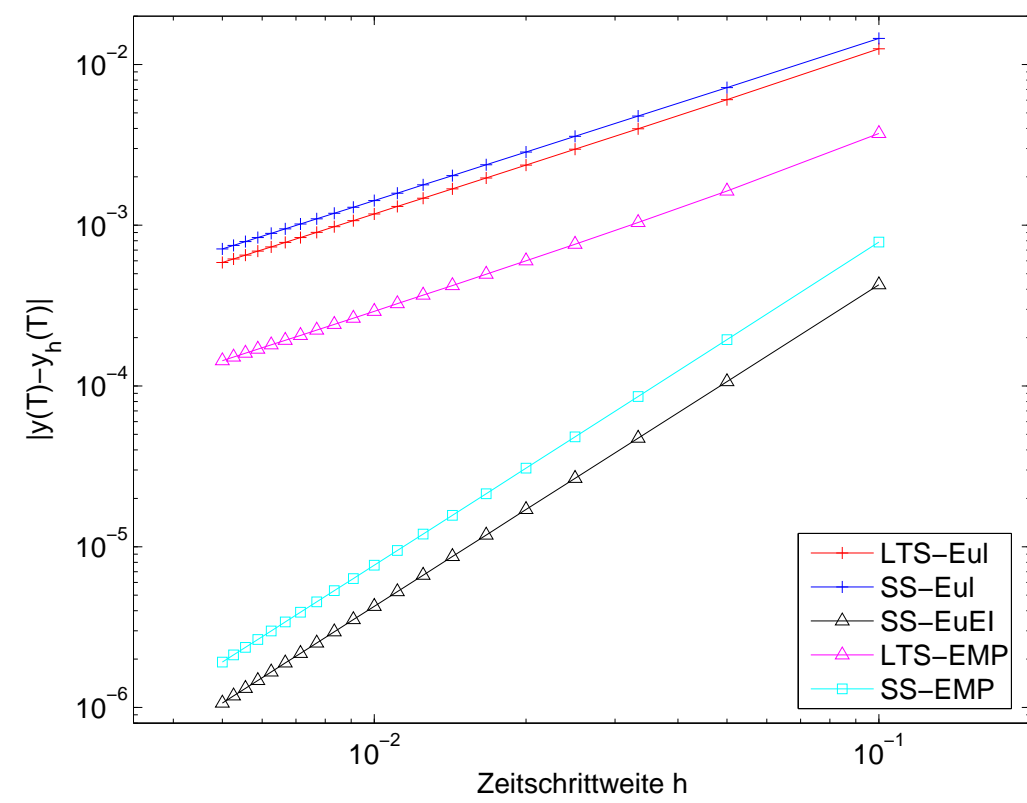
$$\begin{aligned} (8.5.5) \quad & \longleftrightarrow \begin{aligned} \mathbf{r}_{k+\frac{1}{2}} &= \mathbf{r}_k + \frac{1}{2}h\mathbf{v}_k, \\ \mathbf{v}_{k+1} &= \mathbf{v}_k + ha(\mathbf{r}_{k+\frac{1}{2}}), \\ \mathbf{r}_{k+1} &= \mathbf{r}_{k+\frac{1}{2}} + \frac{1}{2}h\mathbf{v}_{k+1}. \end{aligned} \end{aligned} \quad (8.5.6)$$



Idea: Replace

$$\begin{array}{ccc} \text{exact evolution} & \longrightarrow & \text{discrete evolution} \\ \Phi_g^h, \Phi_f^h & \longrightarrow & \Psi_g^h, \Psi_f^h \end{array}$$

Example 8.5.3 (Inexact splitting method). Cont. Ex. 8.5.1



- LTS-Eul explicit Euler method (??) $\rightarrow \Psi_{h,g}^h$,
 $\Psi_{h,f}^h + \text{Lie-Trotter-Splitting (8.5.2)}$
- SS-Eul explicit Euler method (??) $\rightarrow \Psi_{h,g}^h$,
 $\Psi_{h,f}^h + \text{Strang-Splitting (8.5.3)}$
- SS-EuEI Strang-Splitting (8.5.3): explicit Euler method (??) \circ exact evolution $\Phi_g^h \circ$ implicate Euler method (??)
- LTS-EMP explicit midpoint rule (8.6.4) $\rightarrow \Psi_{h,g}^h$,
 $\Psi_{h,f}^h + \text{Lie-Trotter-Splitting (8.5.2)}$
- SS-EMP explicit midpoint rule (8.6.4) $\rightarrow \Psi_{h,g}^h$,
 $\Psi_{h,f}^h + \text{Strang-Splitting (8.5.3)}$

Order of splitting methods is determined by the quality of Φ_f^h, Φ_g^h .



Remark 8.5.4. Note that the convergence order of a reversible method (i.e., if we exchange $h \leftrightarrow -h$ and $y_k \leftrightarrow y_{k+1}$ we get the same method) is always even.

8.6 Runge-Kutta methods

So far we only know first order methods, the explicit and implicit Euler method (8.2.1) and (8.2.4), respectively.

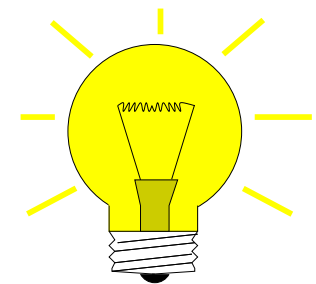
Now we will build a class of methods that achieve orders > 1 . The starting point is a simple *integral equation* satisfied by solutions of initial value problems:

$$\text{IVP: } \begin{array}{l} \dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t)) , \\ \mathbf{y}(t_0) = \mathbf{y}_0 \end{array} \Rightarrow \mathbf{y}(t_1) = \mathbf{y}_0 + \int_{t_0}^{t_1} \mathbf{f}(\tau, \mathbf{y}(t_0 + \tau)) \, d\tau$$

Idea: approximate integral by means of s -point quadrature formula (\rightarrow Sect. 7.1, defined on reference interval $[0, 1]$) with nodes c_1, \dots, c_s , weights b_1, \dots, b_s .

$$\mathbf{y}(t_1) \approx \mathbf{y}_1 = \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{f}(t_0 + c_i h, \mathbf{y}(t_0 + c_i h)) , \quad h := t_1 - t_0 . \quad (8.6.1)$$

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)?

Goal: one-step error $\mathbf{y}(t_1) - \mathbf{y}_1 = O(h^{p+1})$

This goal can already be achieved, if only

$\mathbf{y}(t_0 + c_i h)$ is approximated up to an error $O(h^p)$,

because in (8.6.1) 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 bidding for. Thus, we can construct discrete evolutions of higher and higher order, successively.

Example 8.6.1 (Construction of simple Runge-Kutta methods).

Quadrature formula = trapezoidal rule (8.6.2):

$$Q(f) = \frac{1}{2}(f(0) + f(1)) \quad \leftrightarrow \quad s = 2: \quad c_1 = 0, c_2 = 1, \quad b_1 = b_2 = \frac{1}{2}, \quad (8.6.2)$$

and $\mathbf{y}(T)$ approximated by explicit Euler step (8.2.1)

$$\mathbf{k}_1 = \mathbf{f}(t_0, \mathbf{y}_0), \quad \mathbf{k}_2 = \mathbf{f}(t_0 + h, \mathbf{y}_0 + h\mathbf{k}_1), \quad \mathbf{y}_1 = \mathbf{y}_0 + \frac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2). \quad (8.6.3)$$

(8.6.3) = **explicit trapezoidal rule** (for numerical integration of ODEs)

Quadrature formula \rightarrow simplest Gauss quadrature formula = midpoint rule & $\mathbf{y}(\frac{1}{2}(t_1 + t_0))$ approximated by explicit Euler step (8.2.1)

$$\mathbf{k}_1 = \mathbf{f}(t_0, \mathbf{y}_0), \quad \mathbf{k}_2 = \mathbf{f}(t_0 + \frac{h}{2}, \mathbf{y}_0 + \frac{h}{2}\mathbf{k}_1), \quad \mathbf{y}_1 = \mathbf{y}_0 + h\mathbf{k}_2. \quad (8.6.4)$$

(8.6.4) = **explicit midpoint rule** (for numerical integration of ODEs)



Example 8.6.2 (Convergence of simple Runge-Kutta methods).

- IVP: $\dot{y} = 10y(1 - y)$ (logistic ODE (8.1.3)), $y(0) = 0.01$, $T = 1$,
- Explicit single step methods, uniform timestep h .

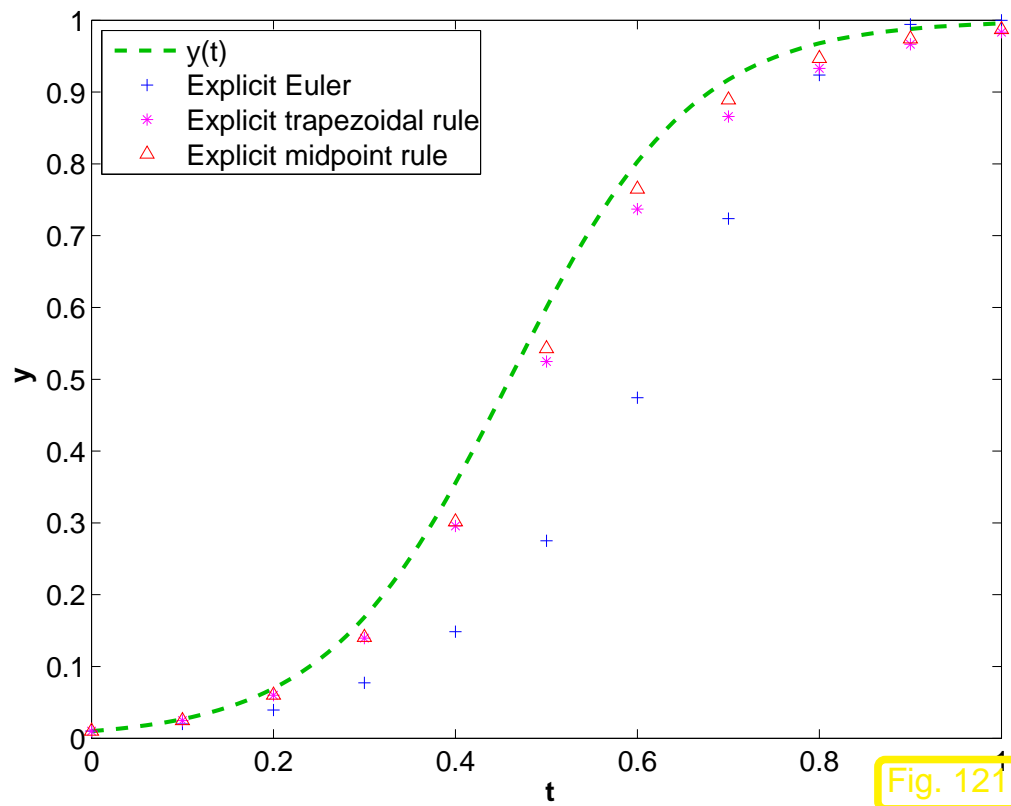


Fig. 121

$y_h(j/10)$, $j = 1, \dots, 10$ for explicit RK-methods

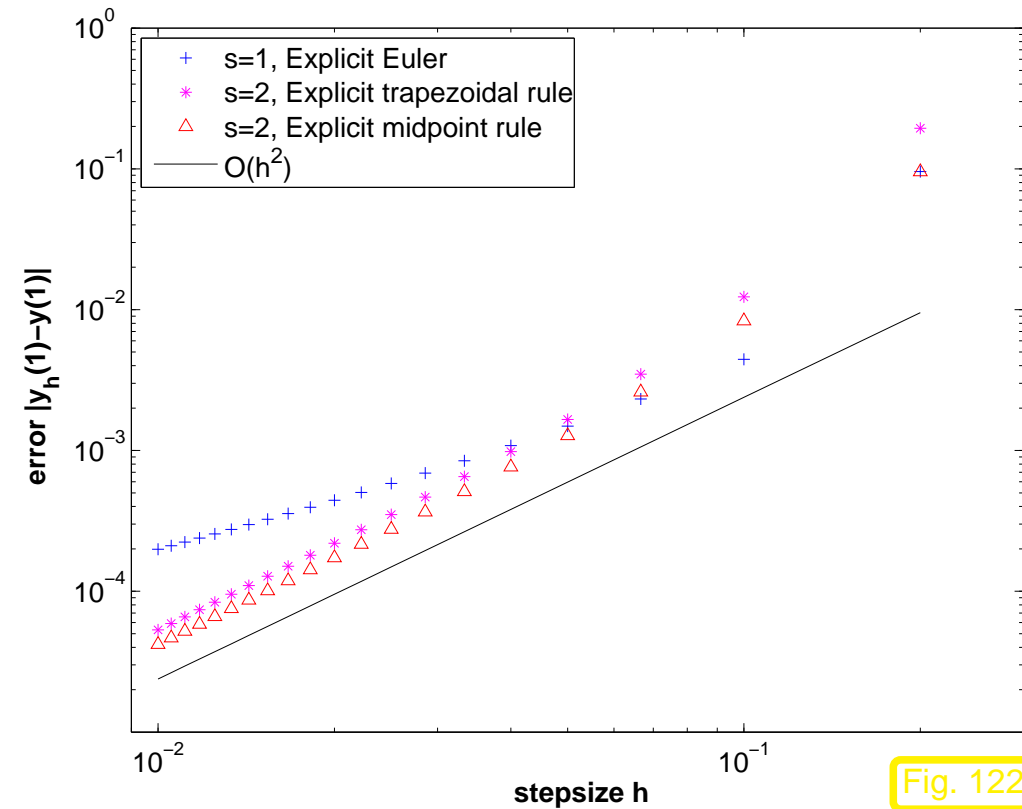


Fig. 122

Errors at final time $y_h(1) - y(1)$

Observation: obvious algebraic convergence with integer rates/orders

explicit trapezoidal rule (8.6.3) order 2

explicit midpoint rule (8.6.4) order 2



The formulas that we have obtained follow a general pattern:

Definition 8.6.1 (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 IVP (8.1.11) is defined by

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + h \sum_{j=1}^{i-1} a_{ij} \mathbf{k}_j), \quad i = 1, \dots, s, \quad \mathbf{y}_1 := \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{k}_i.$$

The $\mathbf{k}_i \in \mathbb{R}^d$ are called *increments*.

The implementation of an *s*-stage explicit Runge-Kutta single step method according to Def. 8.6.1 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* **f**-evaluations and AXPY operations are required.

Shorthand notation for (explicit) Runge-Kutta methods

Butcher scheme

(Note: \mathcal{A} is strictly lower triangular $s \times s$ -matrix)



$$\begin{array}{c|c} \mathbf{c} & \mathcal{A} \\ \hline & \mathbf{b}^T \end{array} :=$$

$$\begin{array}{c|cccc} c_1 & 0 & & \cdots & 0 \\ c_2 & a_{21} & \ddots & & \vdots \\ \vdots & \vdots & & \ddots & \vdots \\ c_s & a_{s1} & \cdots & a_{s,s-1} & 0 \\ \hline & b_1 & \cdots & b_s & \end{array} . \quad (8.6.5)$$

Note that in Def. 8.6.1 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” $\dot{y} = f(t)$.



Necessarily
$$\sum_{i=1}^s b_i = 1$$

Example 8.6.3 (Butcher scheme for some explicit RK-SSM).

- Explicit Euler method (8.2.1):

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$$



order = 1

- explicit trapezoidal rule (8.6.3):

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$



order = 2

- explicit midpoint rule (8.6.4):

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}$$



order = 2

- Classical 4th-order RK-SSM:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array}$$



order = 4

- Kutta's 3/8-rule:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{2}{3} & -\frac{1}{3} & 1 & 0 & 0 \\ \frac{2}{3} & 1 & -1 & 1 & 0 \\ 1 & 1 & -1 & 1 & 0 \\ \hline & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} \end{array}$$



order = 4

Example 8.6.4 (explicit Runge-Kutta method for Riccati equation).

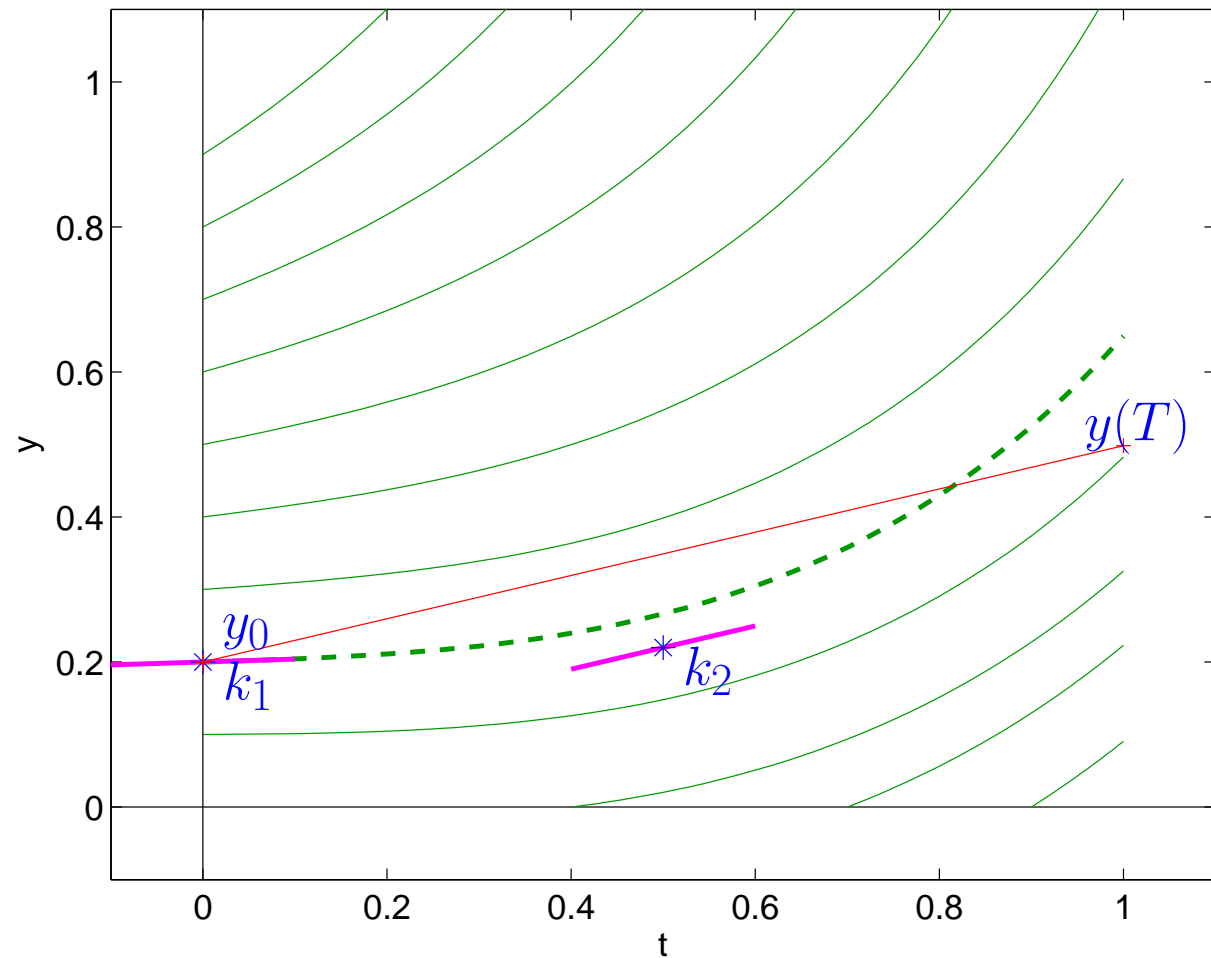
Initial value problem: $\dot{y} = t^2 + y^2$, $y(0) = 0.2$.

Geometrical interpretation of explicit RK-methods as polygonal lines

explicit midpoint rule:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}$$

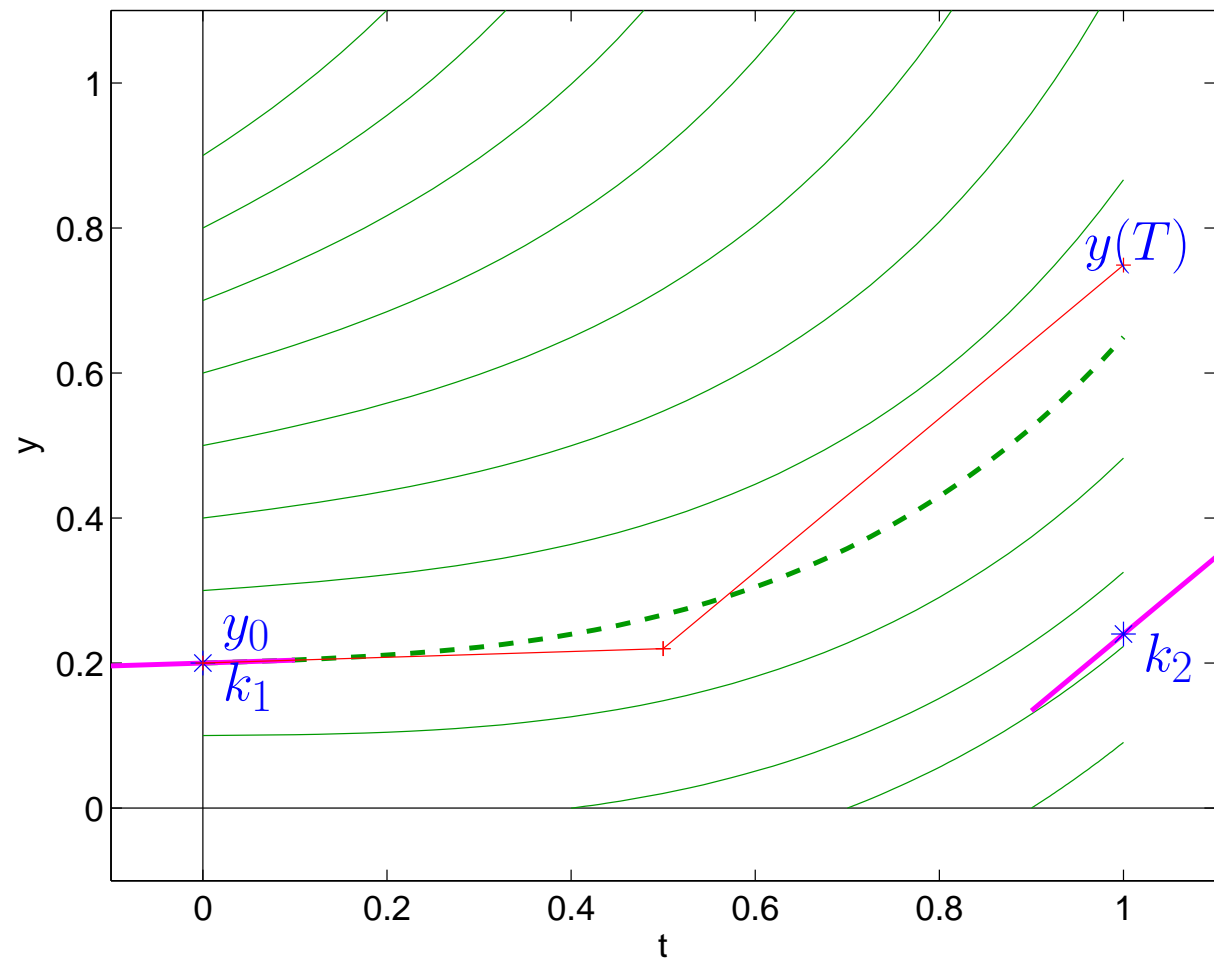
green solution
magenta local slopes k_i
* points f -evaluation
red: polygonal line



explicit trapezoidal rule

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

- green: solution
- magenta: local slops k_i
- *: points f -evaluation
- red: polygonal line

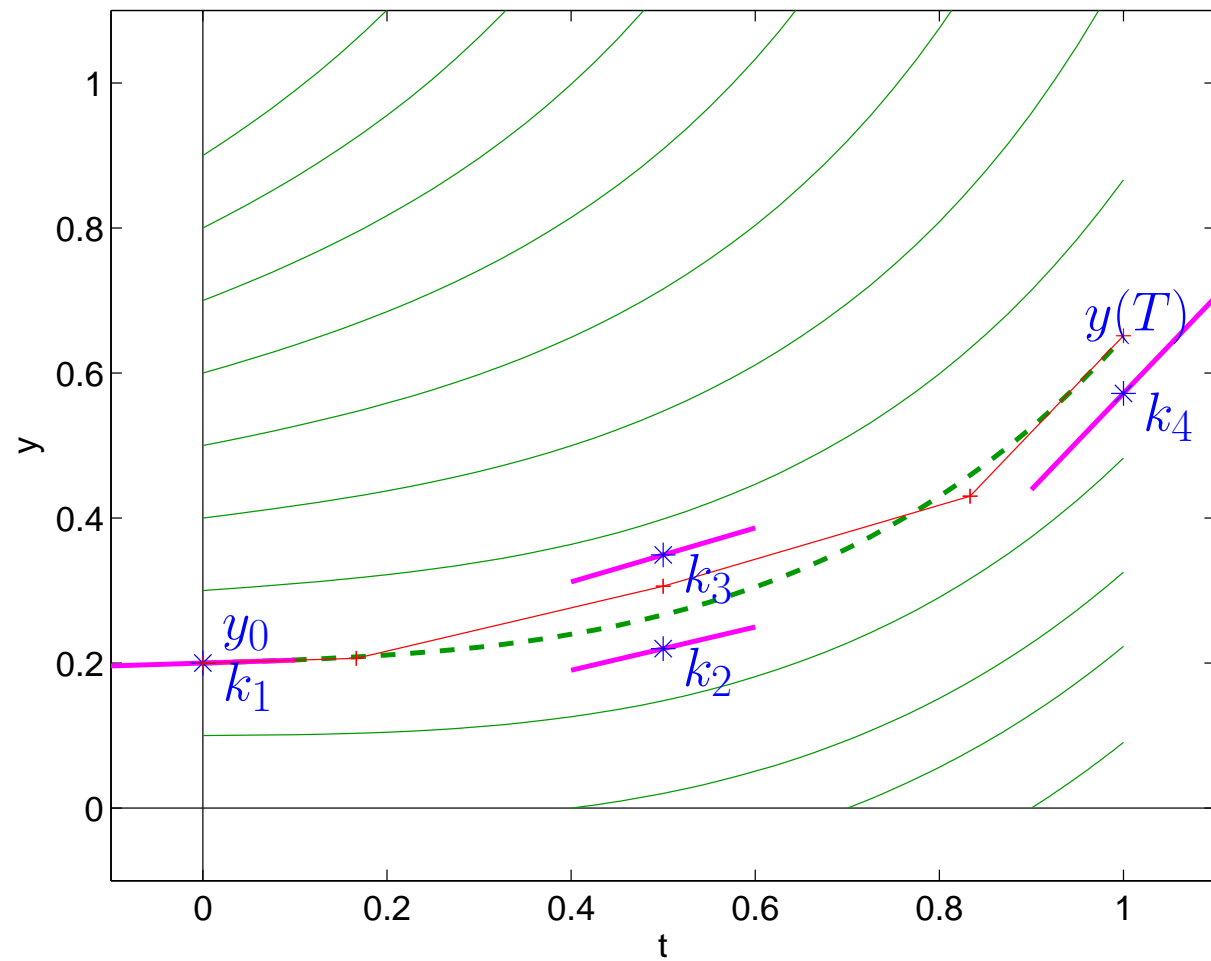


classical Runge-Kutta method (RK4)

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
<hr/>				
	$\frac{1}{6}$	$\frac{2}{6}$	$\frac{2}{6}$	$\frac{1}{6}$

(8.6.6)

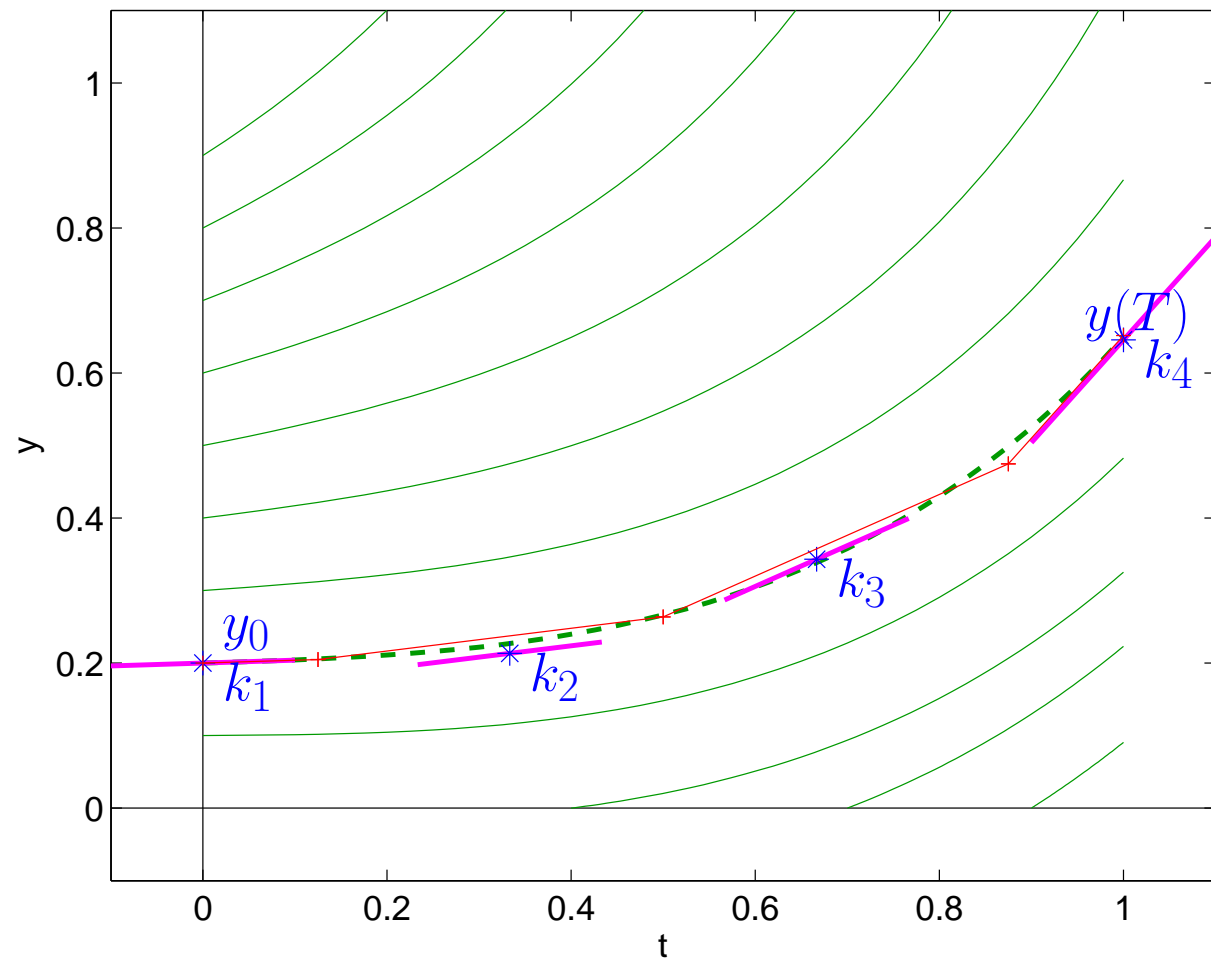
- green solution
- magenta local slopes k_i
- * point f -evaluation
- red: polygonal line



Kuttas 3/8-rule

0	0	0	0	0
1	$\frac{1}{3}$	0	0	0
3	$-\frac{1}{3}$	1	0	0
1	1	-1	1	0
<hr/>				
	$\frac{1}{8}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{1}{8}$

(8.6.7)



- green solution
- magenta local slopes k_i
- * point f -evaluation
- red: polygonal line

Remark 8.6.5 (“Butcher barriers” for explicit RK-SSM).

order p	1	2	3	4	5	6	7	8	≥ 9
minimal no. s of stages	1	2	3	4	6	7	9	11	$\geq p + 3$

No general formula available so far

Known: order $p <$ number s of stages of RK-SSM

Remark 8.6.6 (Explicit ODE integrator a la MATLAB).

Syntax:

```
t,y = ode45(odefun,tspan,y0);
```

odefun : **Handle** to a function of type $(t, y) \leftrightarrow \text{r.h.s. } \mathbf{f}(t, \mathbf{y})$
 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:

t : temporal mesh $\{t_0 < t_1 < t_2 < \dots < t_{N-1} = t_N = T\}$
 y : sequence $(\mathbf{y}_k)_{k=0}^N$ (column vectors)

Example 8.6.7 (Numerical integration of logistic ODE).

usage of ode45

```
from ode45 import ode45
fn = lambda t, y: 5.*y*(1.-y)
t,y = ode45(fn,(0, 1.5),[1.5])
```

integrator: `ode45()`:

→ Handle passing r.h.s.

→ initial and final time

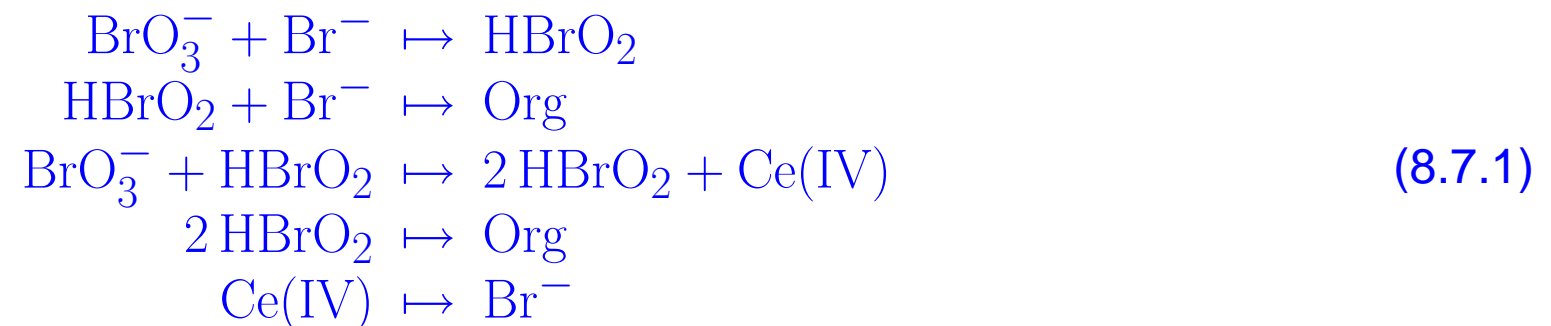
→ initial state y_0



8.7 Stepsize control

Example 8.7.1 (Oregonator reaction).

Special case of oscillating Zhabotinski-Belousov reaction [21]:



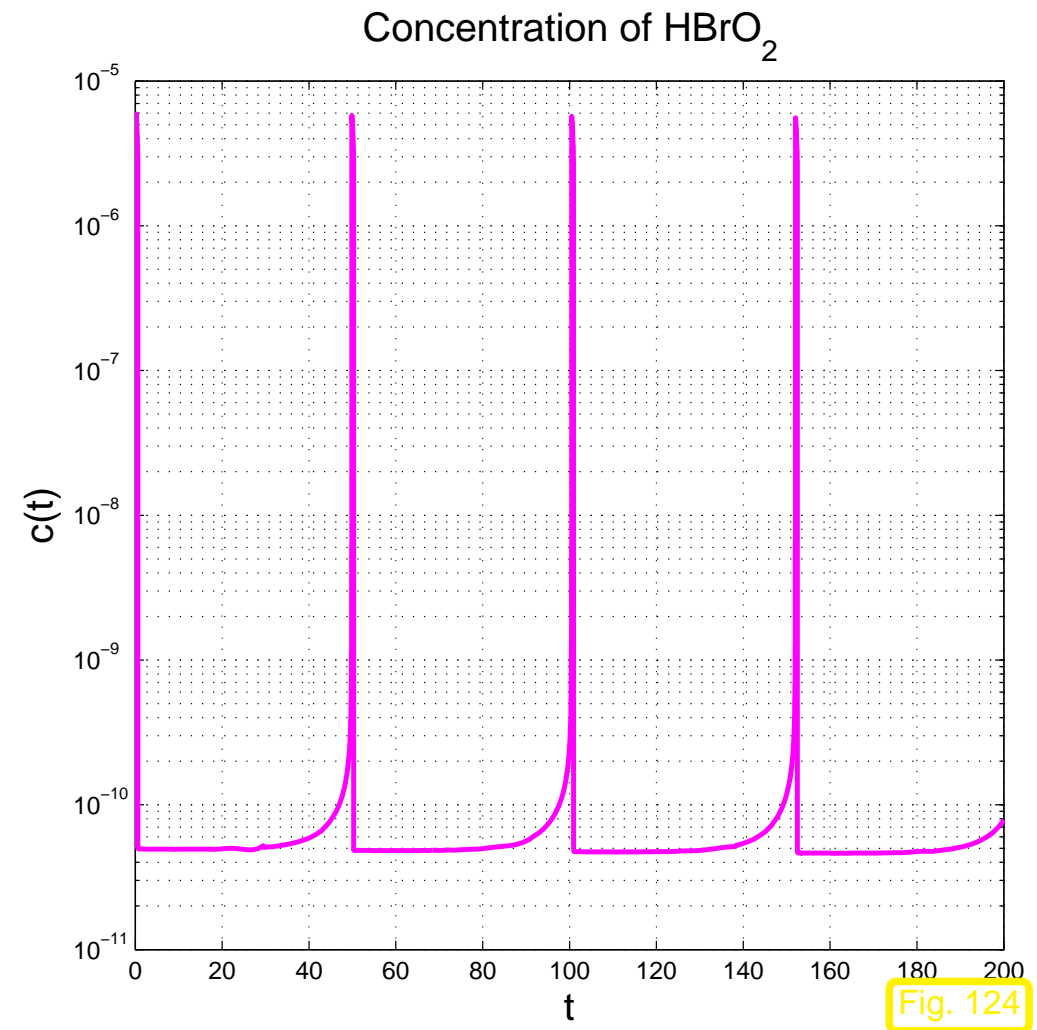
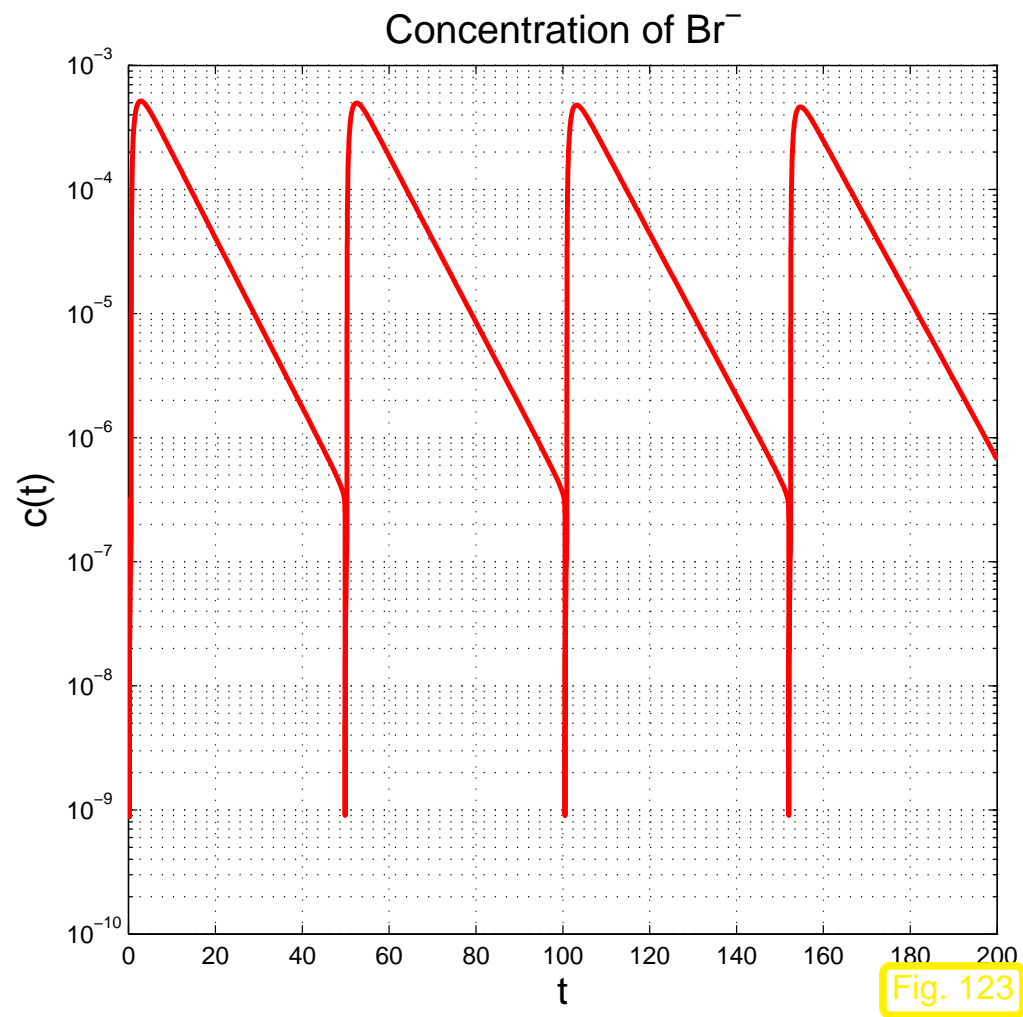
$$\begin{aligned}
y_1 &:= c(\text{BrO}_3^-): & \dot{y}_1 &= -k_1 y_1 y_2 - k_3 y_1 y_3, \\
y_2 &:= c(\text{Br}^-): & \dot{y}_2 &= -k_1 y_1 y_2 - k_2 y_2 y_3 + k_5 y_5, \\
y_3 &:= c(\text{HBrO}_2): & \dot{y}_3 &= k_1 y_1 y_2 - k_2 y_2 y_3 + k_3 y_1 y_3 - 2k_4 y_3^2, \\
y_4 &:= c(\text{Org}): & \dot{y}_4 &= k_2 y_2 y_3 + k_4 y_3^2, \\
y_5 &:= c(\text{Ce(IV)}): & \dot{y}_5 &= k_3 y_1 y_3 - k_5 y_5,
\end{aligned} \tag{8.7.2}$$

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

simulation with initial state $y_1(0) = 0.06$, $y_2(0) = 0.33 \cdot 10^{-6}$, $y_3(0) = 0.501 \cdot 10^{-10}$, $y_4(0) = 0.03$, $y_5(0) = 0.24 \cdot 10^{-7}$:



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 8.7.2 (Blow-up).

Scalar autonomous IVP:

$$\dot{y} = y^2, \quad y(0) = y_0 > 0.$$

$$\blacktriangleright \quad y(t) = \frac{y_0}{1 - y_0 t}, \quad t < 1/y_0.$$

Solution exists only for finite time and then suffers a **Blow-up**, that is, $\lim_{t \rightarrow 1/y_0} y(t) = \infty$: $J(y_0) =] - \infty, 1/y_0]$!

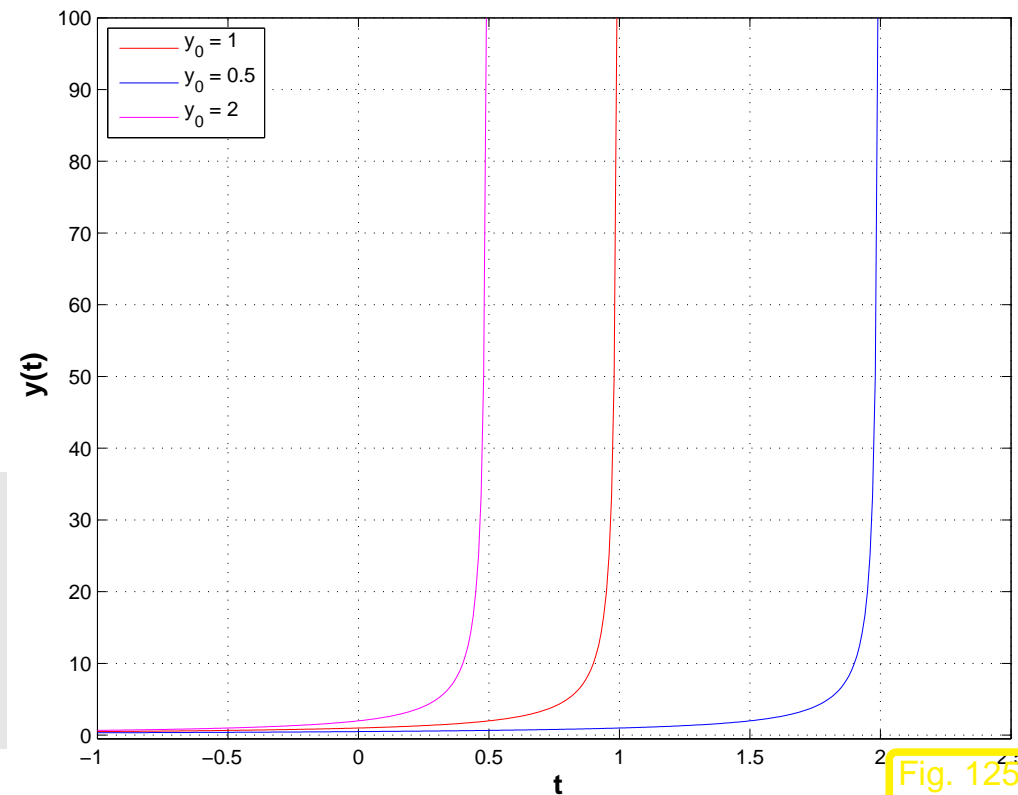
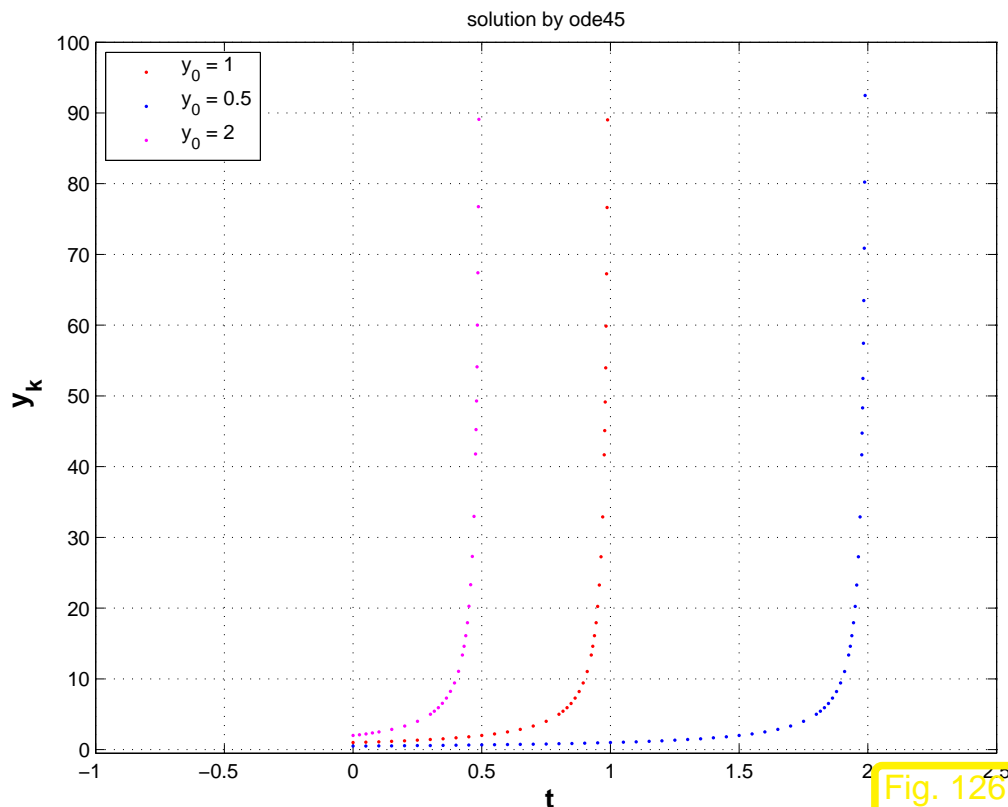


Fig. 125

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 (8.2.1) applied to the above IVP?



```

1 odefun = lambda t,y: y**2.0
2 print 'y0_=',0.5
3 t2, y2 = ode45(odefun, (0,2), 0.5,
4               stats=True)
5 print '\n', 'y0_=',1
6 t1, y1 = ode45(odefun, (0,2), 1,
7               stats=True)
8 print '\n', 'y0_=',2
9 t3, y3 = ode45(odefun, (0,2), 2,
10              stats=True)

```

warning messages:

```

y0 = 0.5
error: OdePkg:InvalidArgument
Solving has not been successful. The iterative integration loop exited at time t =

```

```

Number of successful steps: 151
Number of failed attempts: 74
Number of function calls: 1344

```

```

y0 = 1
error: OdePkg:InvalidArgument
Solving has not been successful. The iterative integration loop exited at time t =

```

```
Number of successful steps: 146  
Number of failed attempts: 74  
Number of function calls: 1314
```

```
y0 = 2  
error: OdePkg:InvalidArgument  
Solving has not been successful. The iterative integration loop exited at time t =
```

```
Number of successful steps: 144  
Number of failed attempts: 72  
Number of function calls: 1290
```

We observe: ode45 manages to reduce stepsize more and more as it approaches the singularity of the solution!



Key issue (discussed for autonomous ODEs below):

Choice of *good temporal mesh* $\{0 = t_0 < t_1 < \dots < t_{N-1} < t_N\}$
for a given single step method applied to an IVP

What does “good” mean ?

be efficient

be accurate

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}

} **local-in-time**
stepsize control

Tool: **Local-in-time** one-step error estimator (*a posteriori*, based on \mathbf{y}_k, h_{k-1})

Why local-in-time timestep control (based on estimating 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,

Simple algorithm:

$EST_k < \max\{ATOL, \|y_k\| RTOL\}$: Carry out next timestep (stepsize h)
 Use larger stepsize (e.g., αh with some $\alpha > 1$) for following step
 (*)

$EST_k > \max\{ATOL, \|y_k\| RTOL\}$: 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.

Code 8.7.3: simple local stepsize control for single step methods

```

1 def odeintadapt(Psilow, Psihigh, T, y0, h0, reltol, abstol, hmin):
2     t=[0]; y=[y0]; h=h0 #
3     while t[-1] < T and h > hmin:
4         yh = Psihigh(h,y0)
5         yH = Psilow(h,y0)
6         est = norm(yH-yh)
7         if est < max(reltol*norm(y0), abstol):
8             y0 = yh; y.append(y0);
9             t.append(t[-1]+min(T-t[-1],h))
            h = 1.1*h

```

```
10         else:
11             h = h/2.0
12         return (array(t), array(y))
13
14 def odeintadapt_ext(Psilow, Psihigh, T, y0, h0, reltol, abstol, hmin):
15     """extended_version_of_odeintadapt_also_returns_vector_of_
16     rejected_points_and_vector_of_estimated_errors"""
17     t=[0]; y=[y0]; h=h0; rej = []; ee = [0]
18     while t[-1] < T and h > hmin:
19         yh = Psihigh(h, y0)
20         yH = Psilow(h, y0)
21         est = norm(yH-yh)
22         if est < max(reltol*norm(y0), abstol):
23             # print 'accept step, h = y0 = yh; y.append(y0); t.append(t[-1]+min(T-t[-1],h))
24             h = 1.1*h; ee.append(est)
25         else:
26             # print 'reject step, h = rej = hstack([rej,t]); h = h/2.0
27             return (array(t), array(y), rej, ee)
```

Comments on Code 9.2.2:

- Input arguments:

- `Psilow`, `Psihigh`: function handles to discrete evolution operators for autonomous ODE of different order, type `@(y,h)`, expecting a state (column) vector as first argument, and a stepsize as second,
 - `T`: final time $T > 0$,
 - `y0`: initial state y_0 ,
 - `h0`: stepsize h_0 for the first timestep
 - `reltol`, `abstol`: relative and absolute tolerances, see (8.7.4),
 - `hmin`: minimal stepsize, timestepping terminates when stepsize control $h_k < h_{\min}$, which is relevant for detecting blow-ups or collapse of the solution.
-
- line `??`: check whether final time is reached or timestepping has ground to a halt ($h_k < h_{\min}$).
 - line `??`, `??`: advance state by low and high order integrator.
 - line `??`: compute norm of estimated error, see (`??`).
 - line `??`: make comparison (8.7.4) to decide whether to accept or reject local step.
 - line `??`, `??`: step accepted, update state and current time and suggest 1.1 times the current stepsize for next step.
 - line `??` step rejected, try again with half the stepsize.
 - Return values:

- \mathbf{t} : temporal mesh $t_0 < t_1 < t_2 < \dots < t_N < T$, where $t_N < T$ indicated premature termination (collapse, blow-up),
- \mathbf{y} : sequence $(\mathbf{y}_k)_{k=0}^N$.

! By the heuristic considerations, see (8.7.3) it seems that EST_k measures the one-step error for the low-order method Ψ and that we should use $\mathbf{y}_{k+1} = \Psi^{h_k} \mathbf{y}_k$, if the timestep is accepted.

However, it would be foolish not to use the better value $\mathbf{y}_{k+1} = \tilde{\Psi}^{h_k} \mathbf{y}_k$, since it is available for free. This is what is done in every implementation of adaptive methods, also in Code 9.2.2, and this choice can be justified by control theoretic arguments [13, Sect. 5.2].

Example 8.7.4 (Simple adaptive stepsize control).

- 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 (8.2.1) and explicit trapezoidal rule (8.6.3)

Code 8.7.5: function for Ex. 8.7.4

```
1 def odeintadaptdriver(T,a, reltol=1e-2, abstol=1e-4):
```

```

2      " " " Simple_adaptive_timestepping_strategy_of_
          Code~\ref{mc:odeintadapt}
3  based_on_explicit_Euler\eqref{eq:eeul}_and_explicit_
    trapezoidal
4  rule\eqref{eq:exTrap}
5  " " "
6
7      # autonomous ODE  $\dot{y} = \cos(ay)$  and its general solution
8      f = lambda y: (cos(a*y)**2)
9      sol = lambda t: arctan(a*(t-1))/a
10     # Initial state  $y_0$ 
11     y0 = sol(0)
12
13     # Discrete evolution operators, see Def. 8.2.1
14     # Explicit Euler (8.2.1)
15     Psilow = lambda h,y: y + h*f(y)
16     # Explicit trapzoidal rule (8.6.3)
17     Psihigh = lambda h,y: y + 0.5*h*( f(y) + f(y+h*f(y)) )
18
19     # Heuristic choice of initial timestep and  $h_{\min}$ 
20     h0 = T/(100.0*(norm(f(y0))+0.1)); hmin = h0/10000.0;
21     # Main adaptive timestepping loop, see Code 9.2.2
22     t,y, rej, ee =

```

```
odeintadapt_ext(Psilow , Psihigh , T, y0 , h0 , reltol , abstol , hmin)
```

```
# Plotting the exact the approximate solutions and rejected timesteps
```

```
fig = plt.figure()
```

```
tp = r_[0:T:T/1000.0]
```

```
plt.plot(tp , sol(tp) , 'g--' , linewidth=2, label=r' $y(t)$ ')
```

```
plt.plot(t , y , 'r.' , label=r' $y_k$ ')
```

```
plt.plot(rej , zeros(len(rej)) , 'm+' , label='rejection')
```

```
plt.title('Adaptive_timestepping , _rtol_=%1.2f , _atol_=%1.4f , _a  
_=%d' % (reltol , abstol , a))
```

```
plt.xlabel(r't' , fontsize=14)
```

```
plt.ylabel(r'y' , fontsize=14)
```

```
plt.legend(loc='upper_left')
```

```
plt.show()
```

```
# plt.savefig('../PICTURES/odeintadaptsol.eps')
```

```
print '%d_timesteps , %d_rejected_timesteps' %  
(size(t)-1, len(rej))
```

```
# Plotting estimated and true errors
```

```
fig = plt.figure()
```

```
plt.plot(t , abs(sol(t) - y) , 'r+' , label=r'true_error_ $|y(t_k) - y_k|$ ')
```

```
plt.plot(t , ee , 'm*' , label='estimated_error_EST_k')
```



```
43     plt.xlabel(r't', fontsize=14)
44     plt.ylabel(r'error', fontsize=14)
45     plt.legend(loc='upper_left')
46     plt.title('Adaptive_timestepping, _rtol_=%1.2f, _atol_=%1.4f, _a
         _=%d' % (reitol, abstol, a))
47     plt.show()
48 # plt.savefig('../PICTURES/odeintadapterr.eps')
49
50 if __name__ == '__main__':
51     odeintadaptdriver(T=2, a=20, reitol=1e-2, abstol=1e-4)
```

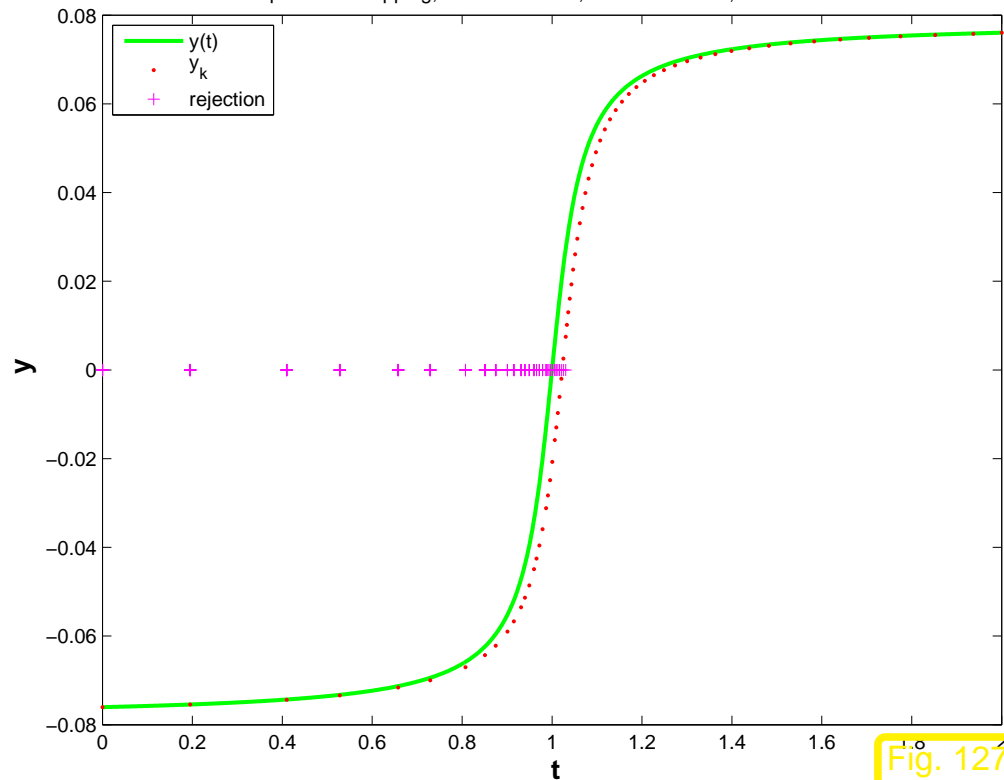
Adaptive timestepping, $\text{rtol} = 0.010000$, $\text{atol} = 0.000100$, $a = 20.000000$ 

Fig. 127

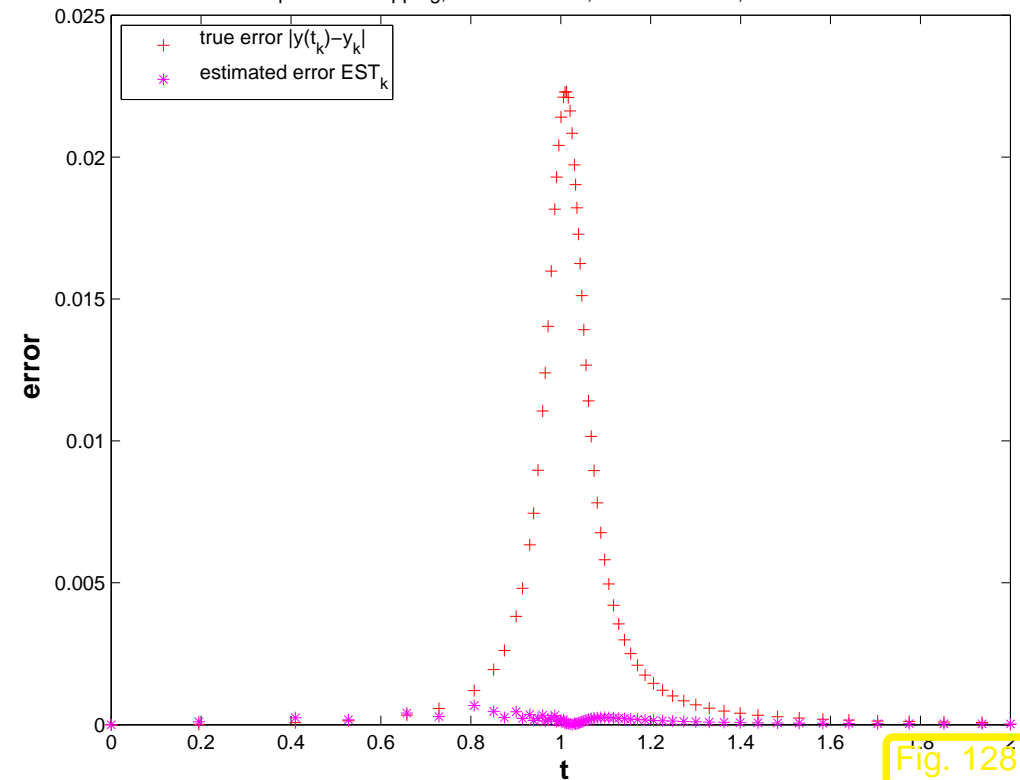
Adaptive timestepping, $\text{rtol} = 0.010000$, $\text{atol} = 0.000100$, $a = 20.000000$ 

Fig. 128

Gradinaru
D-MATH

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!

Example 8.7.6 (Gain through adaptivity). → Ex. 8.7.4

Simple adaptive timestepping from previous experiment Ex. 8.7.4.

New: initial state $y(0) = 0!$

Now we study the dependence of the maximal point error on the computational effort, which is proportional to the number of timesteps.

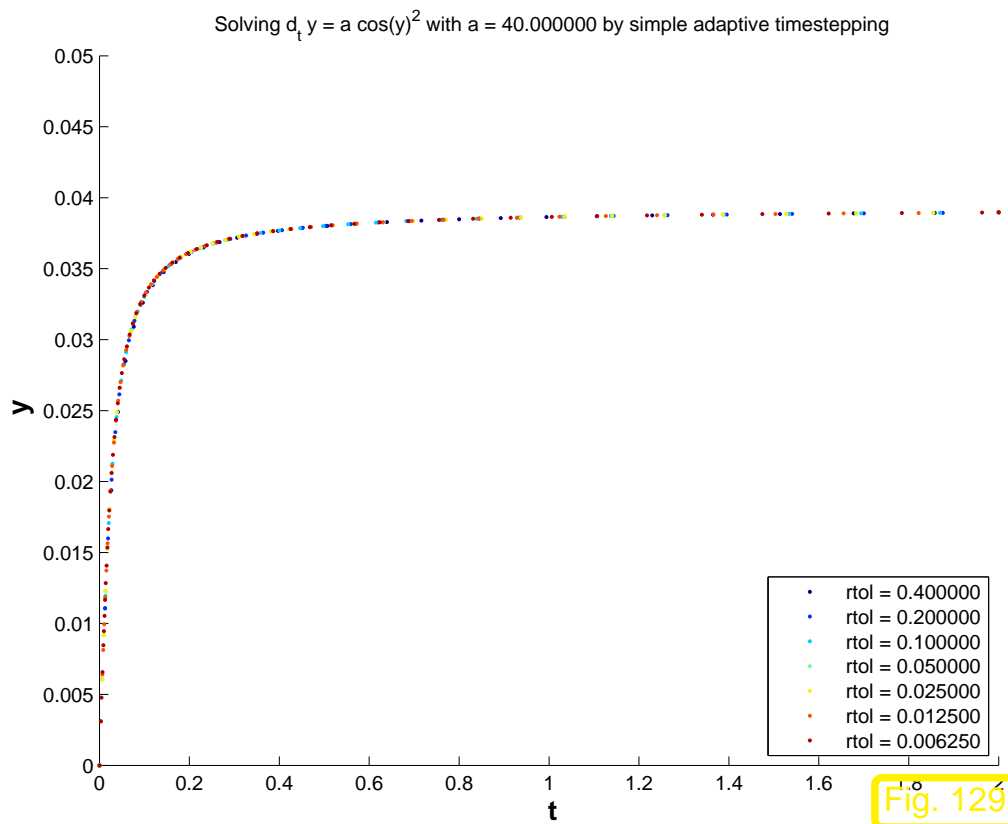
Code 8.7.7: function for Ex. 8.7.6

```
1 from numpy import *
2 import matplotlib.pyplot as plt
3 from odeintadapt import odeintadapt_ext
4
5 def adaptgain(T=2.0, a=40.0, reltol=1e-1, abstol=1e-3):
6     """ Experimental_study_of_gas_in_through_simple_adaptive_
7         timestepping
8         strategy_of_Code~\ref{mc:odeintadapt}_based_on_explicit_Euler
9         \eqref{eq:eeul}_and_explicit_trapezoidal
10        rule_\eqref{eq:exTrap}
11        """
```

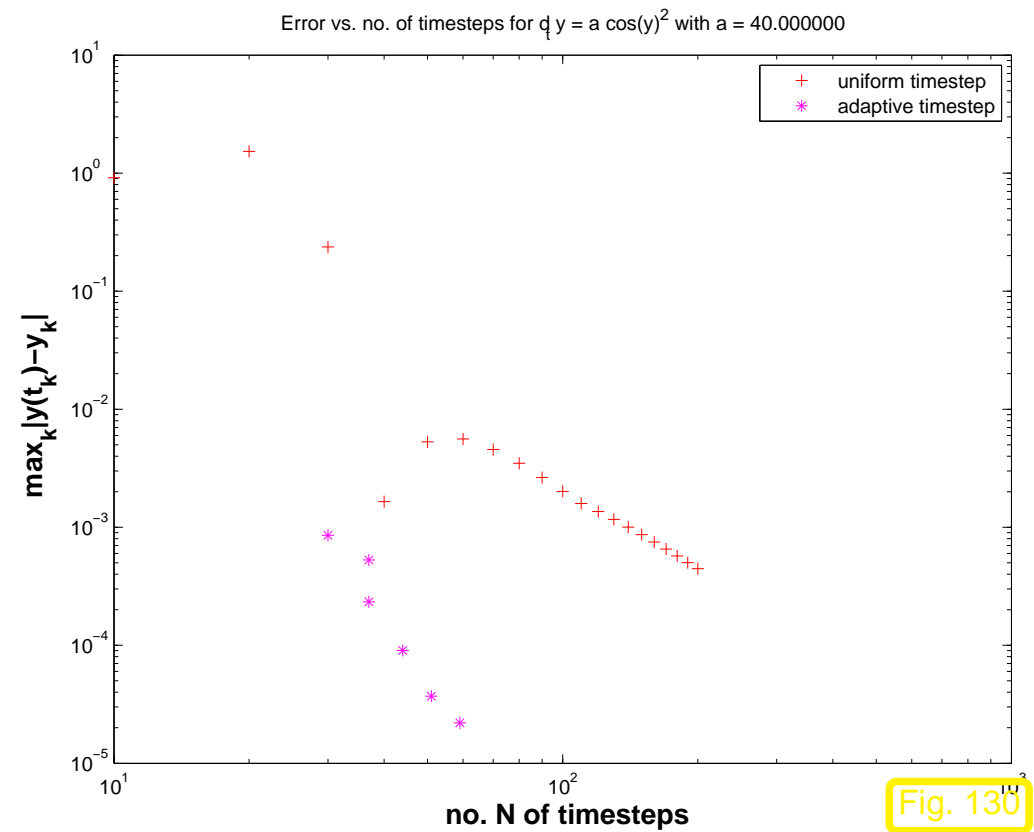
```
12 # autonomous ODE  $\dot{y} = \cos(ay)$  and its general solution
13 f = lambda y: (cos(a*y)**2.0)
14 sol = lambda t: arctan(a*(t))/a # the constant c is 0: arctan(a*(t-c))/a
15 # Initial state  $y_0$ 
16 y0 = sol(0)
17
18 # Discrete evolution operators, see Def. 8.2.1
19 # Explicit Euler (8.2.1)
20 Psi_low = lambda h, y: y + h*f(y)
21 # Explicit trapezoidal rule (8.6.3)
22 Psi_high = lambda h, y: y + 0.5*h*(f(y)+f(y+h*f(y)))
23
24 # Loop over uniform timesteps of varying length and integrate ODE by explicit
   trapezoidal
25 # rule (8.6.3)
26 nvals = r_[10:210:10]
27 err_unif = []
28 for N in nvals:
29     h = T/N; t = 0; y = y0; err = 0
30     for k in arange(1, N+1):
31         y = Psi_high(h, y); t = t+h
32         err = max(err, abs(sol(t) - y));
33     err_unif.append(err)
34 # Run adaptive timestepping with various tolerances, which is the only way
```

```
35 # to make it use a different total number of timesteps.
36 # Plot the solution sequences for different values of the relative tolerance.
37 plt.figure()
38 # axis([0 2 0 0.05]); hold on; col = colormap;
39 ntimes = []
40 err_ad = []
41 tols = []
42 rejs = []
43 l = 1
44 for rtol in reltol*2.0**r_[2:-5:-1]:
45     # Crude choice of initial timestep and  $h_{\min}$ 
46     h0 = T/10.0; hmin = h0/10000.0
47     # Main adaptive timestepping loop, see Code 9.2.2
48     t, y, rej, ee =
49         odeintadapt_ext(Psilow, Psihigh, T, y0, h0, rtol, 0.01*rtol, h
50         ntimes.append(len(t)-1)
51         err_ad.append(max(abs(sol(t) - y)))
52         tols.append(rtol)
53         rejs.append(len(rej))
54         print 'rtol=%1.3f: %d timesteps, %d rejected, %d
55             timesteps' % (rtol, len(t)-1, len(rej))
56         plt.plot(t, y, '.', label='rtol=%1.3f'%rtol, #, 'color', col(10*(l-
57             1)+1,:));
58         l = l+1
```

```
56 plt.xlabel(r't', fontsize=14)
57 plt.ylabel(r'y', fontsize=14)
58 plt.legend(loc='lower_right')
59 plt.title(r'Solving  $d_t y = a \cdot \cos(y)^2$  with  $a = ' + str(a) + '$  by simple
    adaptive timestepping')
60 # plt.savefig('../PICTURES/adaptgainsol.eps')
61
62 # Plotting the errors vs. the number of timesteps
63 plt.figure()
64 plt.loglog(nvals, err_unif, 'r+', label='uniform_timestep')
65 plt.loglog(ntimes, err_ad, 'm*', label='adaptive_timestep')
66 plt.xlabel('no. of timesteps', fontsize=14)
67 plt.ylabel(r' $\max_k |y(t_k) - y_k|$ ', fontsize=14)
68 plt.title(r'Error vs. no. of timesteps for  $d_t y = a \cdot \cos(y)^2$  with
     $a = ' + str(a) + '$ ')
69 plt.legend(loc='upper_right')
70 plt.show()
71 # plt.savefig('../PICTURES/adaptgain.eps')
72
73 if __name__ == '__main__':
74     adaptgain()
```



Solutions $(y_k)_k$ for different values of $rtol$



Error vs. computational effort

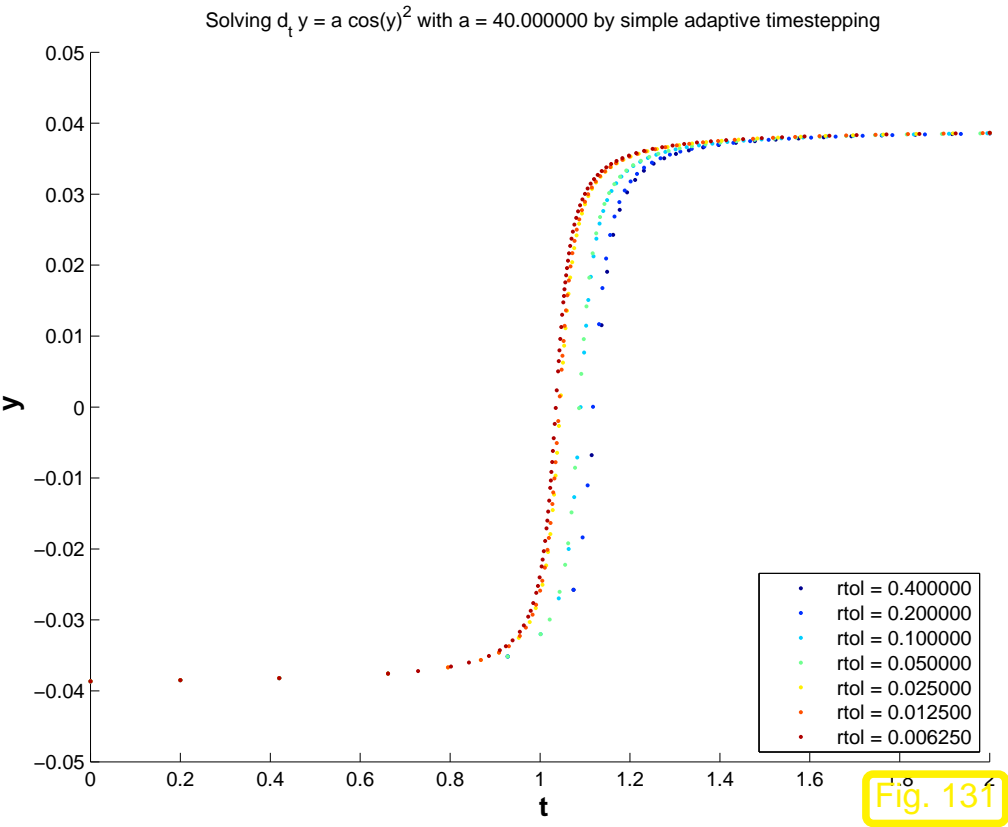
Observations:

👉 Adaptive timestepping achieves much better accuracy for a fixed computational effort.

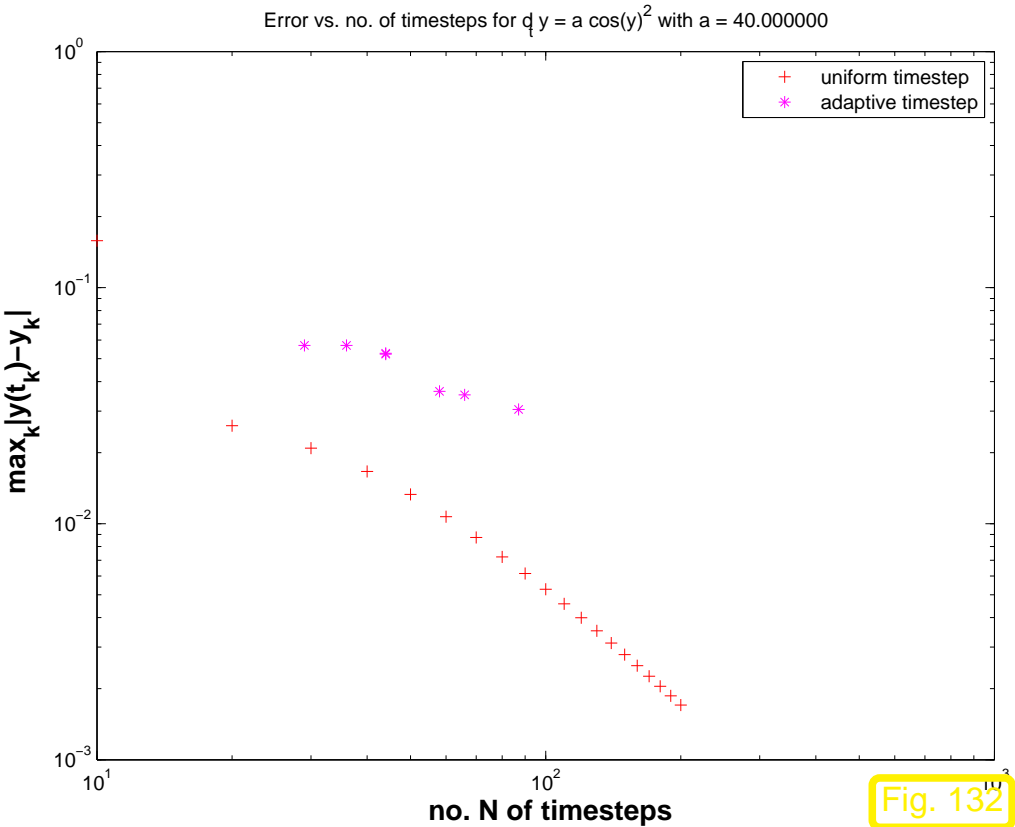
Example 8.7.8 (“Failure” of adaptive timestepping). → Ex. 8.7.6

Same ODE and simple adaptive timestepping as in previous experiment Ex. 8.7.6. Same evaluations.

Now: initial state $y(0) = -0.0386$ as in Ex. 8.7.4



Solutions $(y_k)_k$ for different values of $rtol$



Error vs. computational effort

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 $y(0) \approx -\pi/2$. 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.



Remark 8.7.9 (Refined local stepsize control).

The above algorithm (Code 9.2.2) is simple, but the rule for increasing/shrinking of timestep arbitrary and “wastes” information contained in EST_k : TOL:

More ambitious goal ! When $EST_k > TOL$: stepsize adjustment better $h_k = ?$
 When $EST_k < TOL$: stepsize prediction good $h_{k+1} = ?$

Assumption: At our disposal are two discrete evolutions:

- Ψ with $order(\Psi) = p$ (\rightarrow “low order” single step method)
- $\tilde{\Psi}$ with $order(\tilde{\Psi}) > p$ (\rightarrow “higher order” single step method)

These are the same building blocks as for the simple adaptive strategy employed in Code 9.2.2 (, passed as arguments `Psilow`, `Psihigh` there).

Asymptotic expressions for one-step error for $h \rightarrow 0$:

$$\begin{aligned}\Psi^{h_k} \mathbf{y}(t_k) - \Phi^{h_k} \mathbf{y}(t_k) &= ch^{p+1} + O(h_k^{p+2}) , \\ \tilde{\Psi}^{h_k} \mathbf{y}(t_k) - \Phi^{h_k} \mathbf{y}(t_k) &= O(h^{p+2}) ,\end{aligned}\tag{8.7.5}$$

with some $c > 0$.

Why h^{p+1} ? Remember estimate (??) from the error analysis if the explicit Euler method: we also found $O(h_k^2)$ there for the one-step error of a single step method of order 1.

Heuristics: the timestep h is small \Rightarrow “higher order terms” $O(h^{p+2})$ can be ignored.

$$\blacktriangleright \quad \begin{aligned} \Psi^{h_k} \mathbf{y}(t_k) - \Phi^{h_k} \mathbf{y}(t_k) &\doteq ch_k^{p+1} + O(h_k^{p+2}), \\ \tilde{\Psi}^{h_k} \mathbf{y}(t_k) - \Phi^{h_k} \mathbf{y}(t_k) &\doteq O(h_k^{p+2}). \end{aligned} \quad \Rightarrow \quad \boxed{\text{EST}_k \doteq ch_k^{p+1}}. \quad (8.7.6)$$

\pencil notation: \doteq equality up to higher order terms in h_k

$$\text{EST}_k \doteq ch_k^{p+1} \Rightarrow c \doteq \frac{\text{EST}_k}{h_k^{p+1}}. \quad (8.7.7)$$

Available in algorithm, see (8.7.3)

For the sake of *accuracy* (stipulates “ $\text{EST}_k < \text{TOL}$ ”) & *efficiency* (favors “ $>$ ”) we aim for

$$\text{EST}_k \stackrel{!}{=} \text{TOL} := \max\{\text{ATOL}, \|\mathbf{y}_k\| \text{ RTOL}\}. \quad (8.7.8)$$

What timestep h_* can actually achieve (8.7.8), if we “believe” in (8.7.6) (and, therefore, in (8.7.7))?

$$(8.7.7) \ \& \ (8.7.8) \quad \Rightarrow \quad \text{TOL} = \frac{\text{EST}_k}{h_k^{p+1}} h_*^{p+1}.$$



“Optimal timestep”:
(stepsize prediction)

$$\boxed{h_* = h^{p+1} \sqrt{\frac{\text{TOL}}{\text{EST}_k}}}.$$

(8.7.9)

adjusted stepsize (**A**)

suggested stepsize
(**B**)

(A): In case $EST_k > TOL \Rightarrow$ repeat step with stepsize h_* .

(B): If $EST_k \leq TOL \Rightarrow$ use h_* as stepsize for next step.

Code 8.7.10: refined local stepsize control for single step methods

```

1 from numpy import *
2 from numpy.linalg import norm
3 import matplotlib.pyplot as plt
4
5 def odeintssctrl(Psilow,p,Psihigh,T,y0,h0,reltol,abstol,hmin):
6     t = [0]; y = [y0]; h = h0          #
7     while t[-1] < T and h > hmin:      #
8         yh = Psihigh(h,y0)             #
9         yH = Psilow(h,y0)               #
10        est = norm(yH-yh)                #
11        tol = max(reltol*norm(y[-1]),abstol) #
12        h = h*max(0.5, min(2,(tol/est)**(1.0/(p+1)))) #
13        if est < tol:                    #
14            y0 = yh; y.append(y0);
15            t.append(t[-1]+min(T-t[-1],h)) #
16
17        return (array(t),array(y)) # convert to numpy array
18
19 if __name__ == '__main__':
20     """ this example demonstrates that a slight reduction in the

```

```
20 tolerance_can_have_a_large_effect_on_the_quality_of_the_
    solution
21 """
22 # use two different tolerances
23 reltol1=1e-2; abstol1=1e-4
24 reltol2=1e-3; abstol2=1e-5
25 T=2; a=20
26
27 # autonomous ODE  $\dot{y} = \cos(ay)$  and its general solution
28 f = lambda y: (cos(a*y)**2)
29 sol = lambda t: arctan(a*(t-1))/a
30 # Initial state  $y_0$ 
31 y0 = sol(0)
32
33 # Discrete evolution operators, see Def. 8.2.1
34 Psilow = lambda h,y: y + h*f(y) # Explicit Euler (8.2.1)
35 o = 1 # global order of lower-order evolution
36 Psihigh = lambda h,y: y + 0.5*h*( f(y) + f(y+h*f(y)) ) # Explicit
    trapezoidal rule (8.6.3)
37
38 # Heuristic choice of initial timestep and  $h_{\min}$ 
39 h0 = T/(100.0*(norm(f(y0))+0.1)); hmin = h0/10000.0;
40 # Main adaptive timestepping loop, see Code 9.2.2
41 t1 , y1 =
```

```

odeintssctrl(Psilow,o,Psihigh,T,y0,h0,reltol1,abstol1,hmin)
42 t2,y2 =
odeintssctrl(Psilow,o,Psihigh,T,y0,h0,reltol2,abstol2,hmin)
43
44 # Plotting the exact the approximate solutions and rejected timesteps
45 fig = plt.figure()
46 tp = r_[0:T:T/1000.0]
47 plt.plot(tp,sol(tp),'g-',linewidth=2, label=r'y(t)')
48 plt.plot(t1,y1,'r.',label='reltol:_%1.3f,_%abstol:_%
    %1.5f'%(reltol1, abstol1))
49 plt.plot(t2,y2,'b.',label='reltol:_%1.3f,_%abstol:_%
    %1.5f'%(reltol2, abstol2))
50 plt.title('Adaptive_timestepping,_%a_%d'%a)
51 plt.xlabel(r't',fontsize=14)
52 plt.ylabel(r'y',fontsize=14)
53 plt.legend(loc='upper_left')
54 plt.show()
55 # plt.savefig('../PICTURES/odeintssctrl.eps')

```

Comments on Code 8.7.9 (see comments on Code 9.2.2 for more explanations):

- Input arguments as for Code 9.2.2, except for $p \hat{=}$ order of lower order discrete evolution.

Example 8.7.12 (Adaptive timestepping for mechanical problem).

Movement of a point mass in a conservative force field: $t \mapsto \mathbf{y}(t) \in \mathbb{R}^2 \hat{=}$ trajectory

Newton's law: $\ddot{\mathbf{y}} = F(\mathbf{y}) := -\frac{2\mathbf{y}}{\|\mathbf{y}\|_2^2}.$ (8.7.10)

acceleration \rightarrow $\ddot{\mathbf{y}}$ \leftarrow force $F(\mathbf{y})$

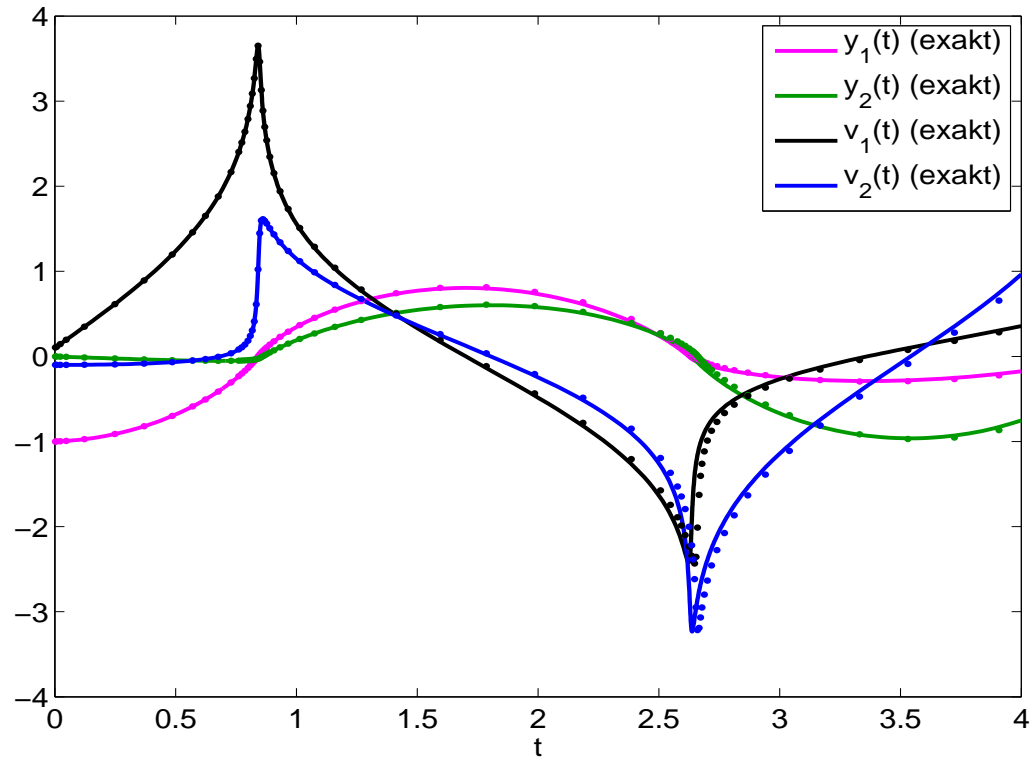
Equivalent 1st-order ODE, see Rem. 8.1.6: with velocity $\mathbf{v} := \dot{\mathbf{y}}$

$$\begin{pmatrix} \dot{\mathbf{y}} \\ \dot{\mathbf{v}} \end{pmatrix} = \begin{pmatrix} \mathbf{v} \\ -\frac{2\mathbf{y}}{\|\mathbf{y}\|_2^2} \end{pmatrix}. \quad (8.7.11)$$

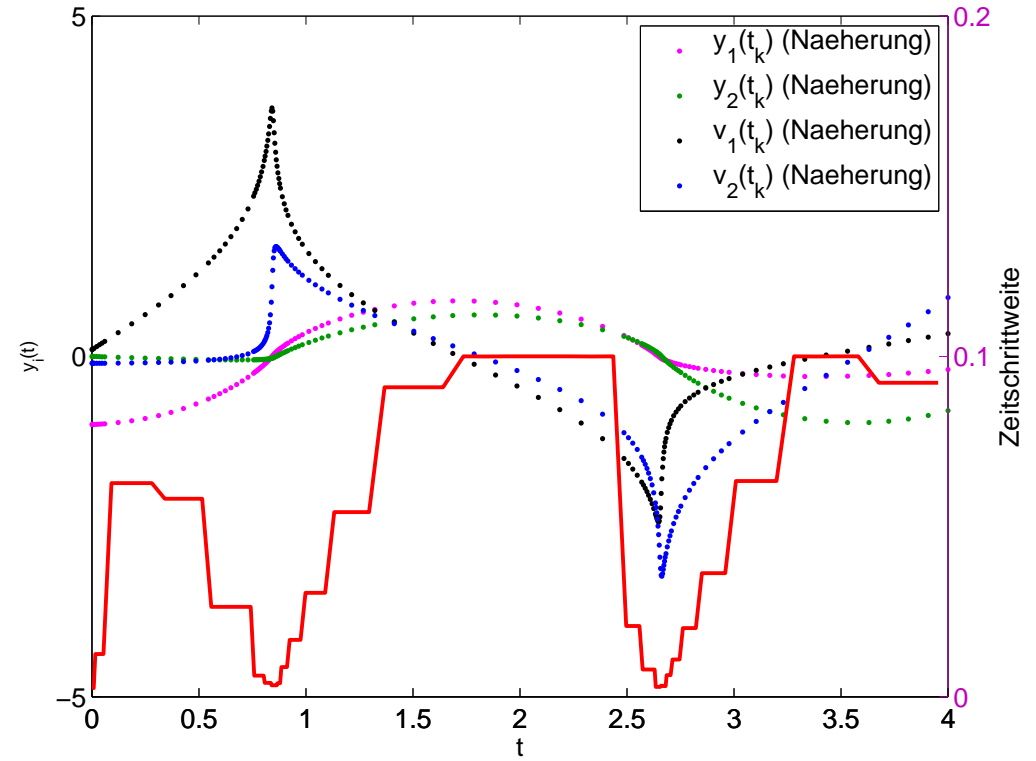
Initial values used in the experiment:

$$\mathbf{y}(0) := \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad \mathbf{v}(0) := \begin{pmatrix} 0.1 \\ -0.1 \end{pmatrix}$$

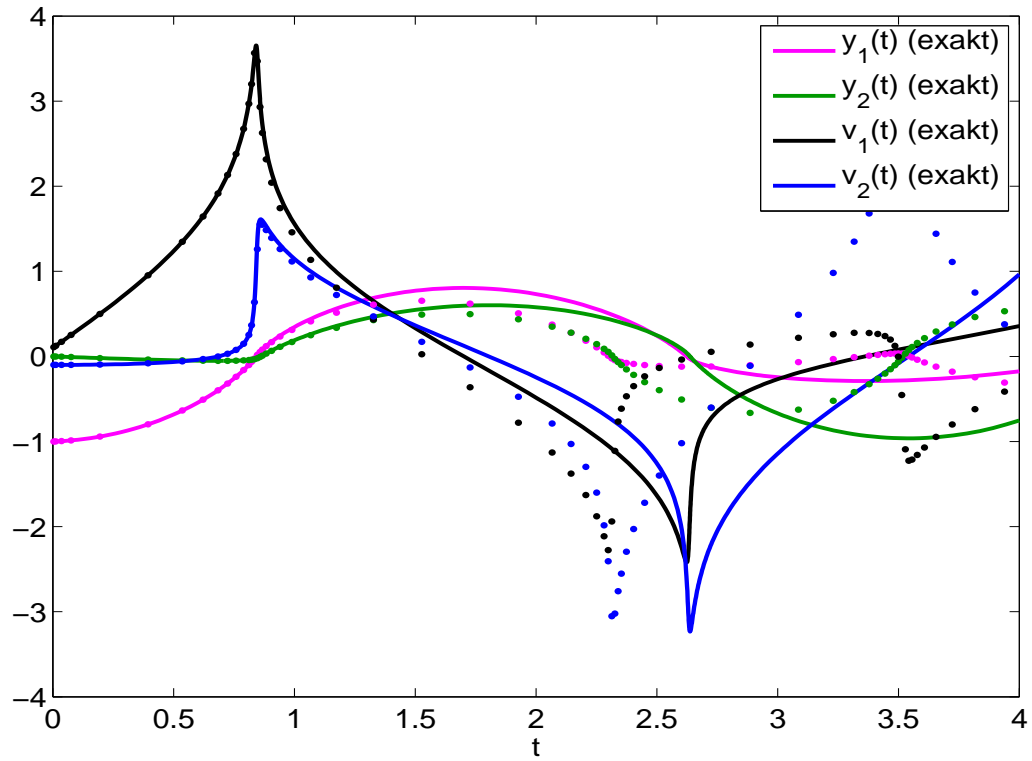
abstol = 0.000010, reltol = 0.001000



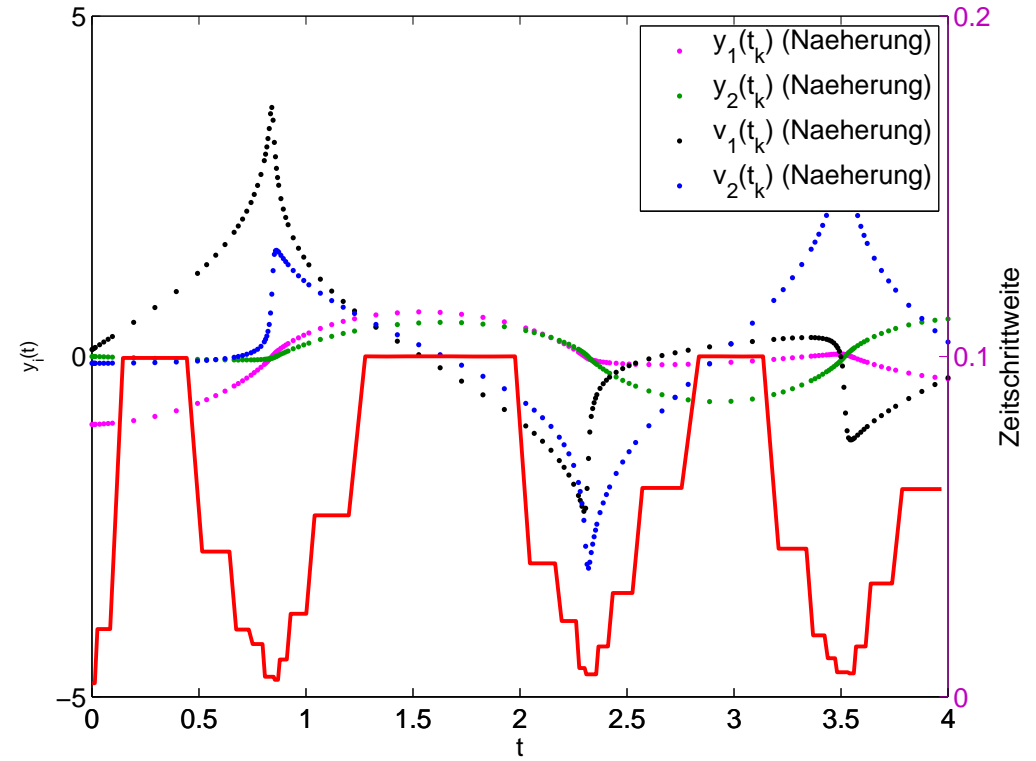
abstol = 0.000010, reltol = 0.001000

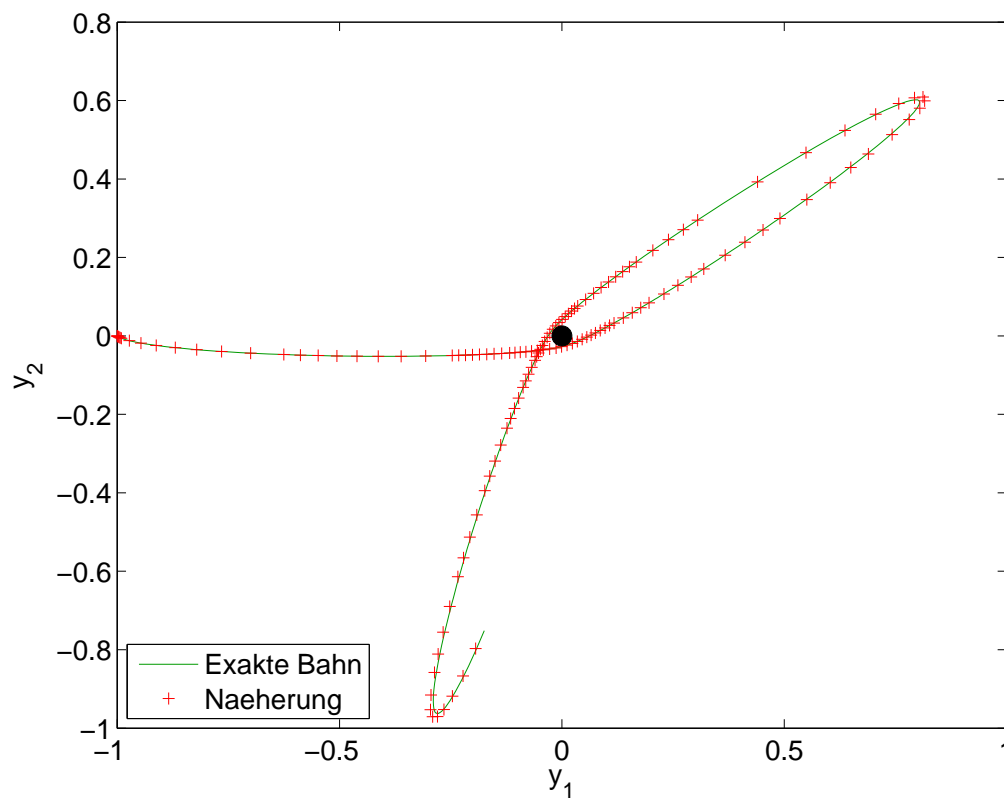


abstol = 0.001000, reltol = 0.010000

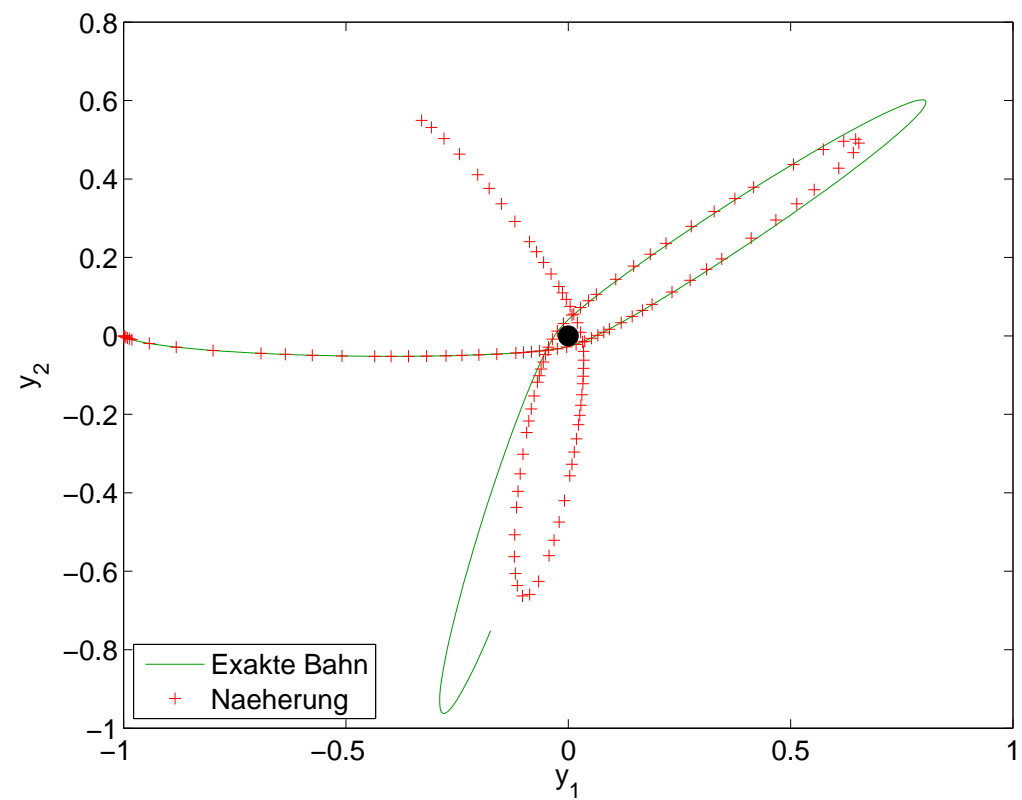


abstol = 0.001000, reltol = 0.010000





$\text{reltol}=0.001, \text{abstol}=1\text{e-}5$



$\text{reltol}=0.01, \text{abstol}=1\text{e-}3$

Gradinaru
D-MATH

Observations:

- 👉 Fast changes in solution components captured by adaptive approach through very small timesteps.
- 👉 Completely wrong solution, if tolerance reduced slightly.

An inevitable consequence of time-local error estimation:

Absolute/relative tolerances do *not* allow to predict accuracy of solution!

8.8 Essential Skills Learned in Chapter 8

Gradinaru
D-MATH

You should know:

- what is an autonomous ODE
- how to reduce an ODE to an autonomous first order system of ODEs
- the meaning of evolution operator
- the Euler methods with pros and contras
- the derivation and advantages of the implicit midpoint rule

- the derivation and advantages of the Störmer-Verlet method
- the idea behind splitting methods and the Strang-splitting
- the idea behind Runge-Kutta methods
- how to use `ode45` method
- the importance and the idea behind adaptivity

9

Stiff Integrators

Explicit Runge-Kutta methods with stepsize control (\rightarrow Sect. 8.7) 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 9.0.1 (ode45 for stiff problem).

$$\text{IVP: } \dot{y} = \lambda y^2(1 - y), \quad \lambda := 500, \quad y(0) = \frac{1}{100}.$$

Code 9.0.2: ode45 for stiff problem

```
1 import matplotlib.pyplot as plt
2 from numpy import diff, finfo, double
3 from ode45 import ode45
4
5 def ode45stiff():
```

```
6      """ tries_to_solve_the_logistic_differential_equation_using_
      ode45_with
7      adaptive_stepsize_control_and_draws_the_results :
8      resulting_data_points -> logode451.eps
9      timestep_size -> logode452.eps
10     both_plots_also_contain_the_'exact'_solution
11     """
12
13     # define the differential equation
14     fun = lambda t, x: 500*x**2*(1-x)
15     # define the time span
16     tspan = (0.0, 1.0)
17     # get total time span
18     L = diff(tspan)
19     # starting value
20     y0 = 0.01
21
22     # make the figure
23     fig = plt.figure()
24
25     # exact solution
26     # set options
27     eps = finfo(double).eps
28     options = {'reltol': 10*eps, 'abstol': eps, 'stats': 'on'}
```

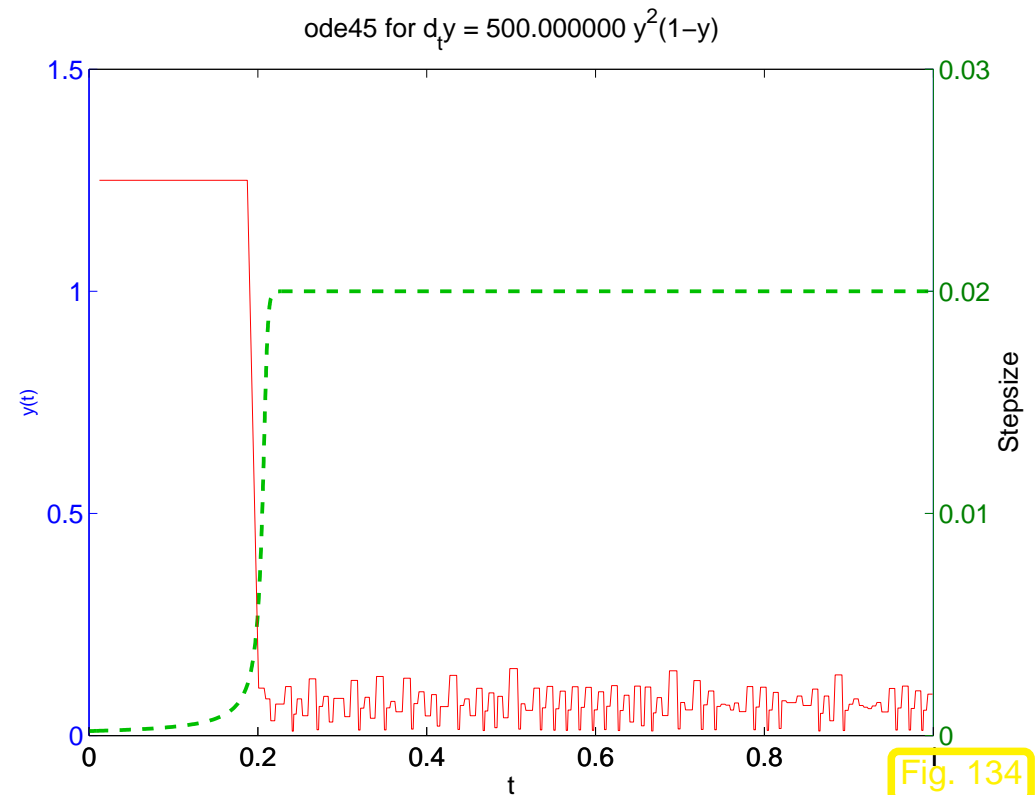
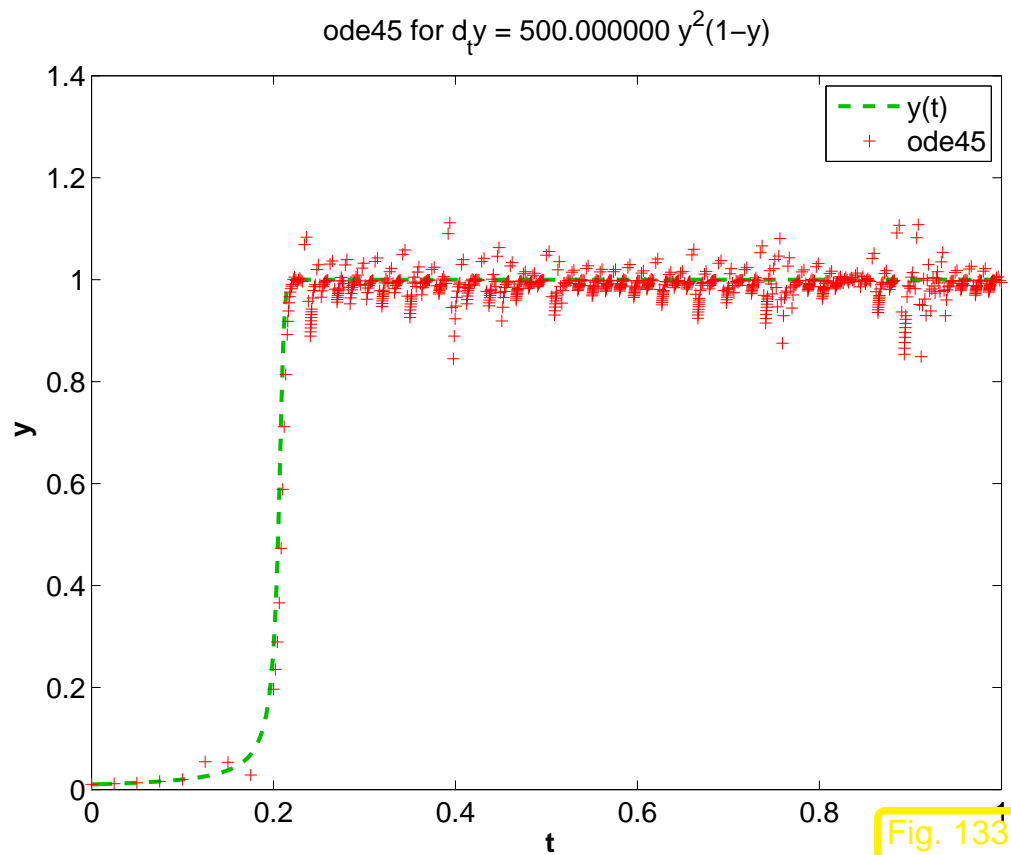
```
29 tex , yex = ode45( fun , (0 , 1) , y0 , ** options )
30 # plot 'exact' results
31 plt . plot ( tex , yex , '—' , color = [ 0 , 0.75 , 0 ] , linewidth = 2 , label = 'y(t)' )
32 plt . hold ( True )
33
34 # ODE45
35 # set options
36 options = { 'reltol' : 0.01 , 'abstol' : 0.001 , 'stats' : 'on' }
37 # get the solution using ode45
38 [ t , y ] = ode45( fun , (0 , 1) , y0 , ** options )
39 # plot the solution
40 plt . plot ( t , y , 'r+' , label = 'ode45' )
41
42 # set label, legend, ....
43 plt . xlabel ( 't' , fontsize = 14 )
44 plt . ylabel ( 'y' )
45 plt . title ( 'ode45_for_d_ty_=_500_y^2(1-y)' )
46 plt . show ( )
47 # write to file
48 # plt.savefig('../PICTURES/logode451.eps')
49
50 # now plot stepsizes
51 fig = plt . figure ( )
52 plt . title ( 'ode45_for_d_ty_=_500_y^2(1-y)' )
```



```
53 ax1 = fig.add_subplot(111)
54 ax1.plot(tex,yex[:,2], 'r—', linewidth=2)
55 ax1.set_xlabel('t')
56 ax1.set_ylabel('y(t)')
57
58 ax2 = ax1.twinx()
59 ax2.plot(0.5*(t[:-1]+t[1:]), diff(t), 'r—')
60 ax2.set_ylabel('stepsize')
61
62 plt.show()
63
64 # write to file
65 #plt.savefig('../PICTURES/logode452.eps')
66
67 if __name__ == '__main__':
68     print 'warning: this function takes a long time to complete!_'
69         (not_really_worth_it)\n'
        ode45stiff()
```

The option 'stats' : 'on' makes the program print statistics about the run of the integrators.

```
Number of successful steps:
2119
Number of failed attempts: 3
Number of function calls:
12726
```



Stepsize control of ode45 running amok!

?

The solution is virtually constant from $t > 0.2$ and, nevertheless, the integrator uses tiny timesteps until the end of the integration interval.



9.1 Model problem analysis

Example 9.1.1 (Blow-up of explicit Euler method).

As in part II of Ex. 8.3.1:

- IVP for logistic ODE, see Ex. 8.1.1

$$\dot{y} = f(y) := \lambda y(1 - y) \quad , \quad y(0) = 0.01 \quad .$$

- Explicit Euler method (8.2.1) with uniform timestep $h = 1/N$, $N \in \{5, 10, 20, 40, 80, 160, 320, 640\}$.

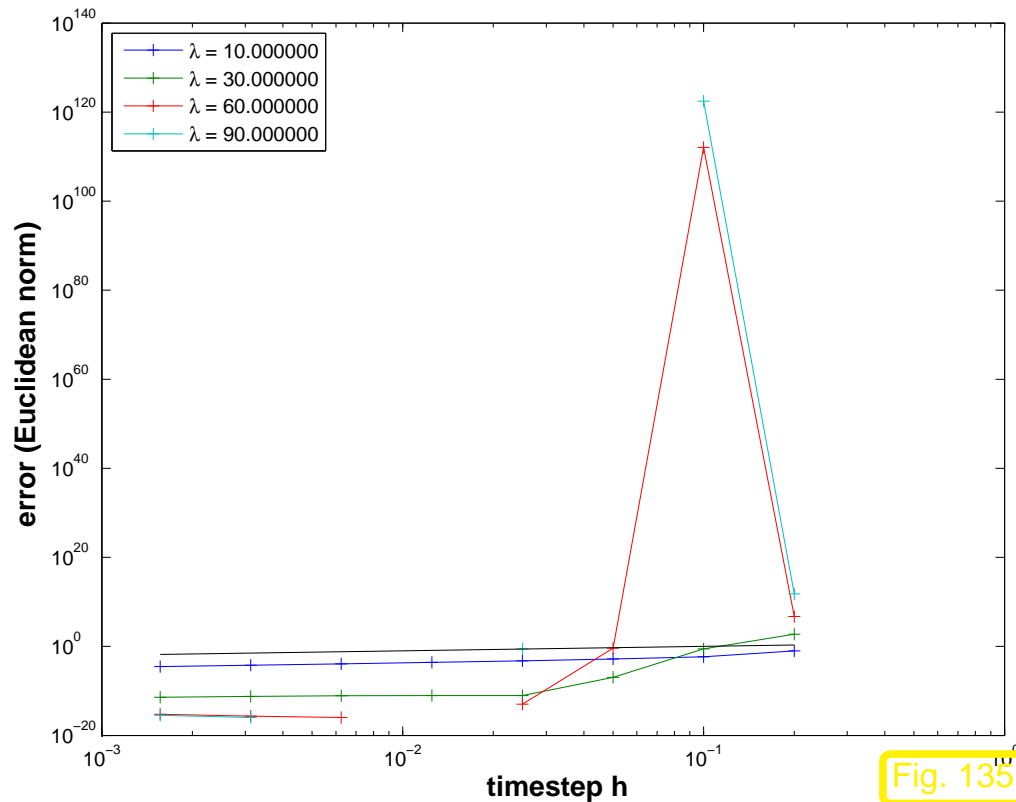


Fig. 135

λ large: blow-up of y_k for large timestep h

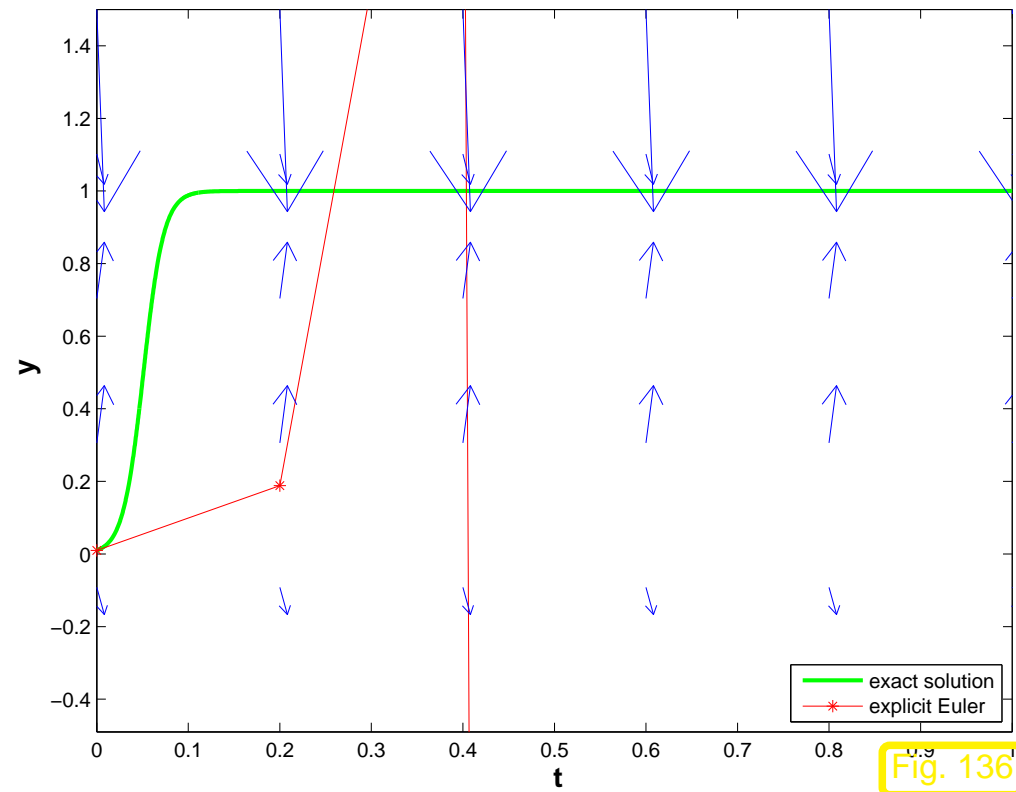


Fig. 136

$\lambda = 90$: — $\hat{=}$ $y(t)$, — $\hat{=}$ Euler polygon

Explanation: y_k may miss the stationary point $y = 1$ (overshooting).

This leads to a sequence $(y_k)_k$ with exponentially increasing oscillations.

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. Similarly, $z(t) := 1 - y(t)$ will behave like the solution of the “decay equation” $\dot{z} = -\lambda z$.

Motivated by the considerations in Ex. 9.1.1 we study the explicit Euler method (8.2.1) for the

linear model problem: $\dot{y} = \lambda y$, $y(0) = y_0$, with $\lambda \ll 0$, (9.1.1)

and *exponentially decaying* exact solution

$$y(t) = y_0 \exp(\lambda t) \rightarrow 0 \quad \text{for } t \rightarrow \infty .$$

Recursion of explicit Euler method for (9.1.1):

(8.2.1) for $f(y) = \lambda y$: $y_{k+1} = y_k(1 + \lambda h)$. (9.1.2)

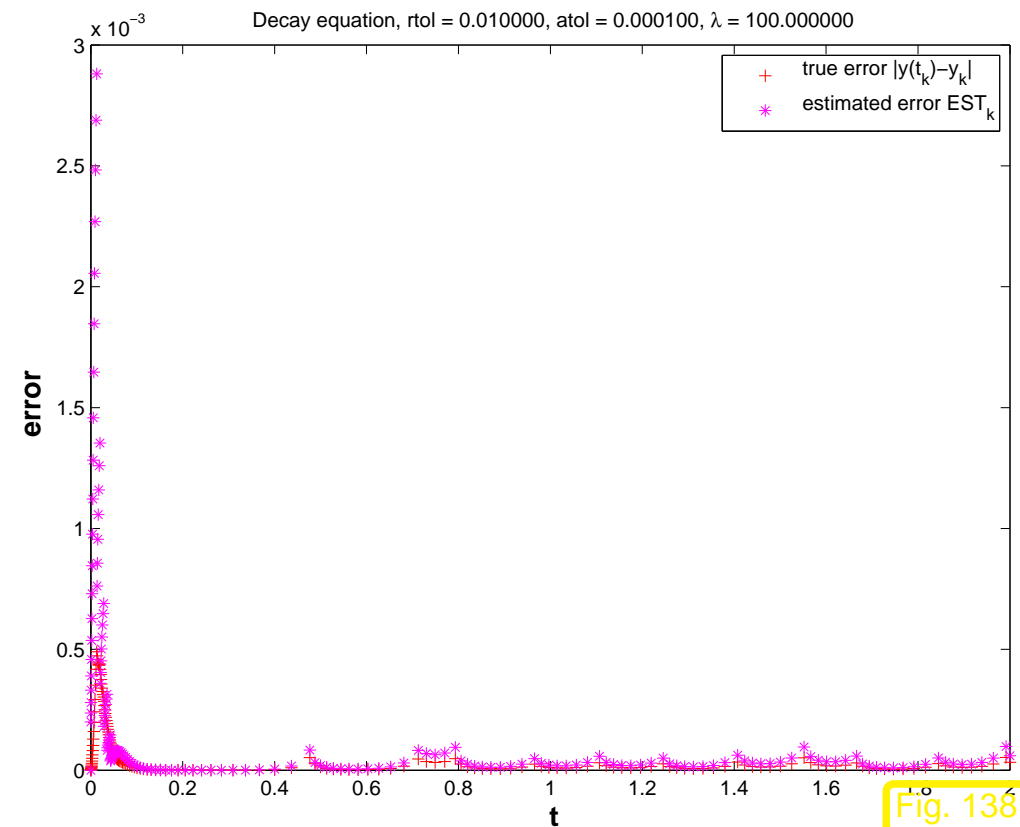
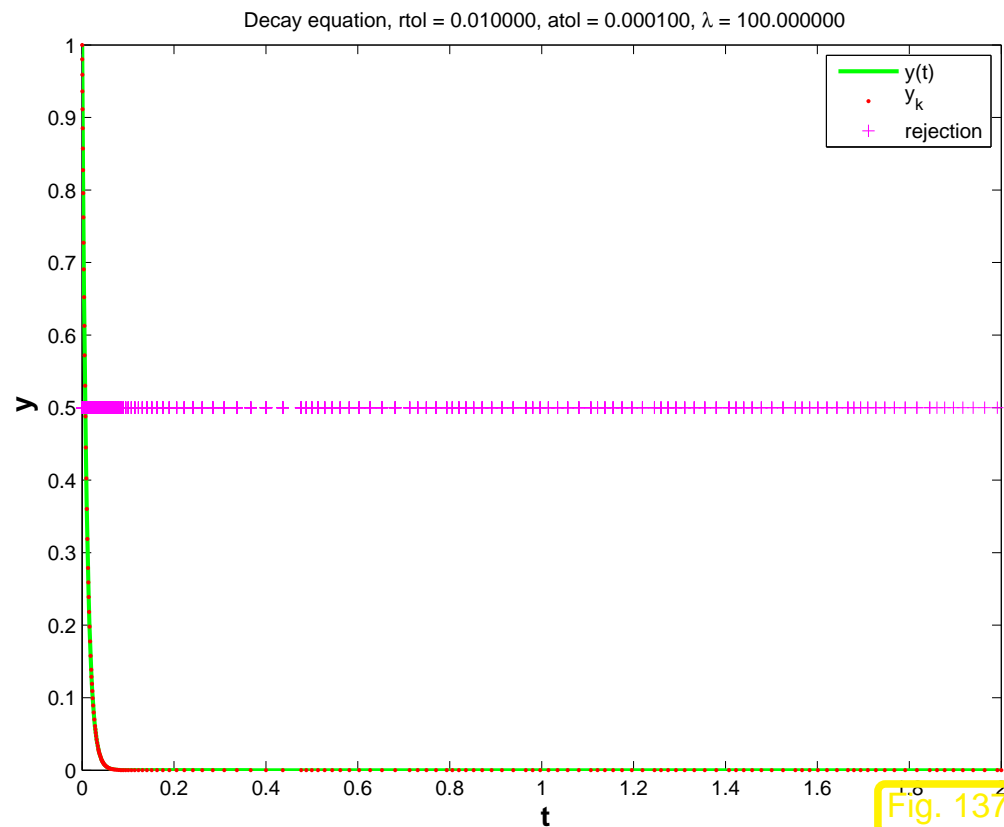
► $y_k = y_0(1 + \lambda h)^k \Rightarrow |y_k| \rightarrow \begin{cases} 0 & , \text{ if } \lambda h > -2 \quad (\text{qualitatively correct}) , \\ \infty & , \text{ if } \lambda h < -2 \quad (\text{qualitatively wrong}) . \end{cases}$

Timestep constraint: only if $|\lambda|h < 2$ we obtain decaying solution by 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. 9.1.1? Let us examine how the simple stepsize control of Code 9.2.2 fares for model problem (9.1.1):

Example 9.1.2 (Simple adaptive timestepping for fast decay).

- “Linear model problem IVP”: $\dot{y} = \lambda y$, $y(0) = 1$, $\lambda = -100$
- Simple adaptive timestepping method as in Ex. 8.7.4, see Code 9.2.2



Observation: in fact, stepsize control enforces small timesteps even if $y(t) \approx 0$ 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.

Is this 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 9.1.3 (Explicit trapezoidal rule for decay equation).

Recall recursion for explicit trapezoidal rule:

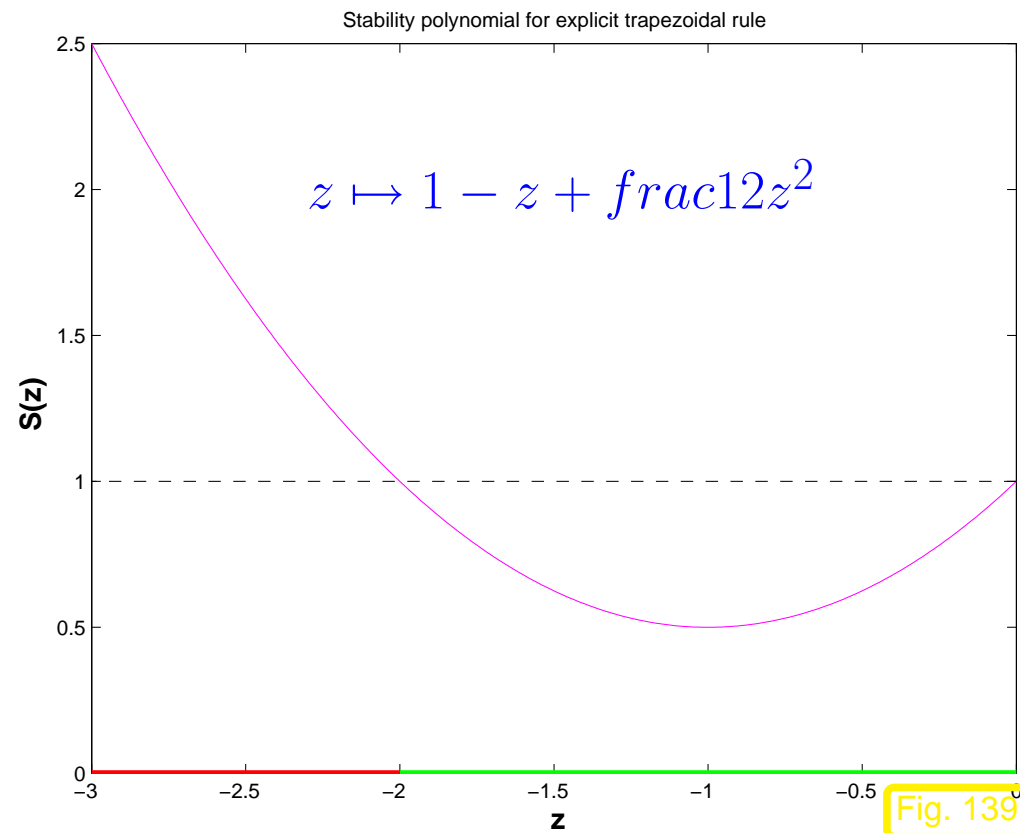
$$\mathbf{k}_1 = \mathbf{f}(t_0, \mathbf{y}_0) , \quad \mathbf{k}_2 = \mathbf{f}(t_0 + h, \mathbf{y}_0 + h\mathbf{k}_1) , \quad \mathbf{y}_1 = \mathbf{y}_0 + \frac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2) . \quad (8.6.3)$$

Apply this to the model problem (9.1.1), that is $\mathbf{f}(y) = f(y) = \lambda y$, $\lambda < 0$:

$$\blacktriangleright \quad k_1 = \lambda y_0 , \quad k_2 = \lambda(y_0 + hk_1) \quad \Rightarrow \quad y_1 = \underbrace{\left(1 + \lambda h + \frac{1}{2}(\lambda h)^2\right)}_{=:S(h\lambda)} y_0 . \quad (9.1.3)$$

► sequence generated by explicit trapezoidal rule:

$$y_k = S(h\lambda)^k y_0 , \quad k = 0, \dots, N . \quad (9.1.4)$$



$$|S(h\lambda)| < 1 \Leftrightarrow -2 < h\lambda < 0 .$$

Qualitatively correct decay behavior of $(y_k)_k$ only under **timestep constraint**

$$h \leq |2/\lambda| . \quad (9.1.5)$$

◇ Gradinaru
D-MATH

Mode problem analysis for general explicit Runge-Kutta method (\rightarrow Def. 8.6.1): apply

Runge-Kutta method $\frac{\mathbf{c}}{\mathbf{b}^T} \mathcal{A}$ to (9.1.1)

$$\begin{aligned} k_i &= \lambda(y_0 + h \sum_{j=1}^{i-1} a_{ij} k_j) , \\ y_1 &= y_0 + h \sum_{i=1}^s b_i k_i \end{aligned} \Rightarrow \begin{pmatrix} \mathbf{I} - z\mathcal{A} & 0 \\ -z\mathbf{b}^T & 1 \end{pmatrix} \begin{pmatrix} \mathbf{k} \\ y_1 \end{pmatrix} = y_0 \begin{pmatrix} \mathbf{1} \\ 1 \end{pmatrix} , \quad (9.1.6)$$

where $\mathbf{k} \in \mathbb{R}^s \hat{=}$ denotes the vector $(k_1, \dots, k_s)^T / \lambda$ of increments, and $z := \lambda h$.

$$\blacktriangleright \quad y_1 = S(z)y_0 \quad \text{with} \quad S(z) := 1 + z\mathbf{b}^T (\mathbf{I} - z\mathfrak{A})^{-1} \mathbf{1} = \det(\mathbf{I} - z\mathfrak{A} + z\mathbf{1}\mathbf{b}^T) . \quad (9.1.7)$$

The first formula for $S(z)$ immediately follows from (9.1.6), the second is a consequence of Cramer's rule.

Thus we have proved the following theorem.

Theorem 9.1.1 (Stability function of explicit Runge-Kutta methods).

The discrete evolution Ψ_λ^h of an explicit s -stage Runge-Kutta single step method (\rightarrow Def. 8.6.1) with Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$ (see (8.6.5)) for the ODE $\dot{y} = \lambda y$ is a multiplication operator according to

$$\Psi_\lambda^h = \underbrace{1 + z\mathbf{b}^T (\mathbf{I} - z\mathfrak{A})^{-1} \mathbf{1}}_{\text{stability function } S(z)} = \det(\mathbf{I} - z\mathfrak{A} + z\mathbf{1}\mathbf{b}^T) , \quad z := \lambda h , \quad \mathbf{1} = (1, \dots, 1)^T \in \mathbb{R}^s .$$

$$\text{Thm. 9.1.1} \quad \Rightarrow \quad S \in \mathcal{P}_s$$

Remember from Ex. 9.1.3: for sequence $(|y_k|)_{k=0}^{\infty}$ produced by explicit Runge-Kutta method applied to IVP (9.1.1) holds $y_k = S(\lambda h)^k y_0$.



$$(|y_k|)_{k=0}^{\infty} \text{ non-increasing} \Leftrightarrow |S(\lambda h)| \leq 1, \quad (9.1.8)$$

$$(|y_k|)_{k=0}^{\infty} \text{ exponentially increasing} \Leftrightarrow |S(\lambda h)| > 1.$$

On the other hand:

$$\forall S \in \mathcal{P}_s: \lim_{|z| \rightarrow \infty} |S(z)| = \infty$$

► **timestep constraint:** In order to avoid exponentially increasing (qualitatively wrong for $\lambda < 0$) sequences $(y_k)_{k=0}^{\infty}$ we must have $|\lambda h|$ *sufficiently small*.

Small timesteps may have to be used for stability reasons, though accuracy may not require them!



Inefficient numerical integration

Remark 9.1.4 (Stepsize control detects instability).

Always look at the bright side of life:

Ex. 9.0.1, 9.1.2: Stepsize control guarantees acceptable solutions, with a hefty price tag however.



9.2 Stiff problems

Objection: The IVP (9.1.1) may be an oddity rather than a model problem: the weakness of explicit Runge-Kutta methods discussed in the previous section may be just a peculiar response to an unusual situation.

This section will reveal that the behavior observed in Ex. 9.0.1 and Ex. 9.1.1 is typical of a large class of problems and that the model problem (9.1.1) really represents a “generic case”.

We examine linear homogeneous IVP of the form

$$\dot{\mathbf{y}} = \underbrace{\begin{pmatrix} 0 & 1 \\ -\beta & -\alpha \end{pmatrix}}_{=: \mathbf{M}} \mathbf{y} \quad , \quad \mathbf{y}(0) = \mathbf{y}_0 \in \mathbb{R}^2 . \quad (9.2.1)$$

In Ex .??: $\beta \gg \frac{1}{4}\alpha^2 \gg 1$.

[52, Sect. 5.6]: general solution of $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, $\mathbf{M} \in \mathbb{R}^{2,2}$, by diagonalization of \mathbf{M} (if possible):

$$\mathbf{M}\mathbf{V} = \mathbf{M}(\mathbf{v}_1, \mathbf{v}_2) = (\mathbf{v}_1, \mathbf{v}_2) \begin{pmatrix} \lambda_1 & \\ & \lambda_2 \end{pmatrix} . \quad (9.2.2)$$

► $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^2 \setminus \{0\} \hat{=}$ eigenvectors of \mathbf{M} , $\lambda_1, \lambda_2 \hat{=}$ eigenvalues of \mathbf{M} , see Def. 4.1.1.

Idea: transform $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ into *decoupled* scalar linear ODEs!

$$\dot{\mathbf{y}} = \mathbf{M}\mathbf{y} \quad \Leftrightarrow \quad \mathbf{V}^{-1}\dot{\mathbf{y}} = \mathbf{V}^{-1}\mathbf{M}\mathbf{V}(\mathbf{V}^{-1}\mathbf{y}) \quad \overset{\mathbf{z}(t):=\mathbf{V}^{-1}\mathbf{y}(t)}{\Leftrightarrow} \quad \dot{\mathbf{z}} = \begin{pmatrix} \lambda_1 & \\ & \lambda_2 \end{pmatrix} \mathbf{z} . \quad (9.2.3)$$

This yields the general solution of the ODE $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$

$$\mathbf{y}(t) = A\mathbf{v}_1 \exp(\lambda_1 t) + B\mathbf{v}_2 \exp(\lambda_2 t) , \quad A, B \in \mathbb{R} . \quad (9.2.4)$$

Note: $t \mapsto \exp(\lambda_i t)$ is general solution of the ODE $\dot{z}_i = \lambda_i z_i$.

Consider discrete evolution of explicit Euler method (8.2.1) for ODE $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$

$$\Psi^h \mathbf{y} = \mathbf{y} + h\mathbf{M}\mathbf{y} \quad \leftrightarrow \quad \mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{M}\mathbf{y}_k .$$

Perform the same transformation as above on the discrete evolution:

$$\mathbf{V}^{-1}\mathbf{y}_{k+1} = \mathbf{V}^{-1}\mathbf{y}_k + h\mathbf{V}^{-1}\mathbf{M}\mathbf{V}(\mathbf{V}^{-1}\mathbf{y}_k) \quad \mathbf{z}_k := \mathbf{V}^{-1}\mathbf{y}_k \quad \Leftrightarrow \quad \underbrace{(\mathbf{z}_{k+1})_i = (\mathbf{z}_k)_i + h\lambda_i (\mathbf{z}_k)_i}_{\hat{=} \text{explicit Euler step for } \dot{z}_i = \lambda_i z_i} . \quad (9.2.5)$$

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 $\dot{z} = \lambda_i z$, $i = 1, \dots, d$.

An analogous statement is true for all Runge-Kutta methods!

(This is revealed by simple algebraic manipulations of the increment equations.)

So far we conducted the model problem analysis under the premises $\lambda < 0$.

However: in Ex. ?? we have $\lambda_{1/2} = \frac{1}{2}\alpha \pm i\sqrt{\beta - \frac{1}{4}\alpha^2}$ (complex eigenvalues!). How will explicit Euler/explicit RK-methods respond to them.

Example 9.2.1 (Explicit Euler method for damped oscillations).

Consider linear model IVP (9.1.1) for $\lambda \in \mathbb{C}$:


$$\operatorname{Re} \lambda < 0 \Rightarrow \text{exponentially decaying solution } y(t) = y_0 \exp(\lambda t) ,$$

because $|\exp(\lambda t)| = \exp(\operatorname{Re} \lambda t)$.

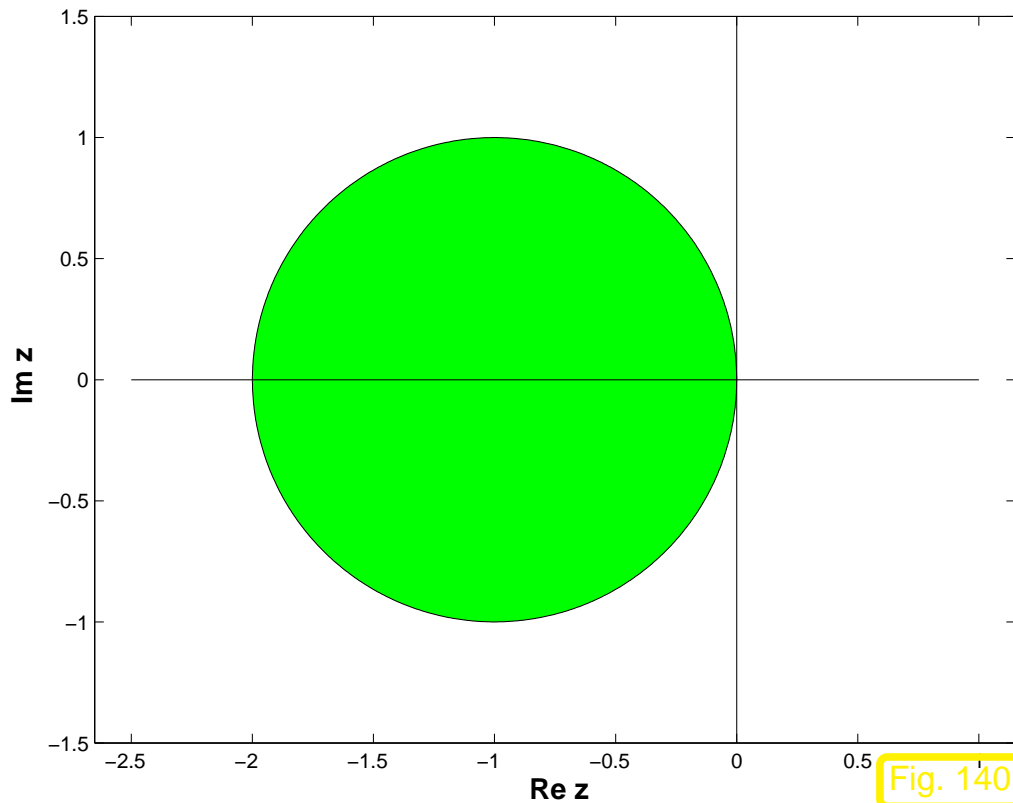
Model problem analysis (\rightarrow Ex. 9.1.1, Ex. 9.1.3) for explicit Euler method and $\lambda \in \mathbb{C}$:

Sequence generated by explicit Euler method (8.2.1) for model problem (9.1.1):

$$y_{k+1} = y_k(1 + h\lambda) . \tag{9.1.2}$$

 $\lim_{k \rightarrow \infty} y_k = 0 \Leftrightarrow |1 + h\lambda| < 1 .$

timestep constraint to get decaying (discrete) solution !

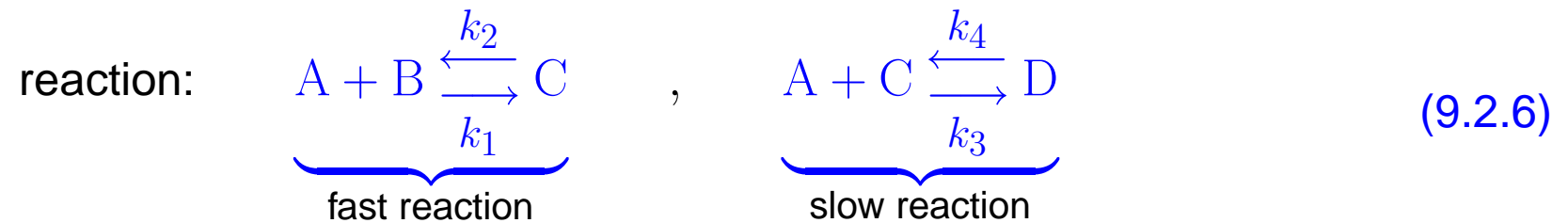


$$\triangleleft \{z \in \mathbb{C}: |1 + z| < 1\}$$

Now we can conjecture what happens in Ex. ??: the eigenvalue $\lambda_2 = \frac{1}{2}\alpha - i\sqrt{\beta - \frac{1}{4}\alpha^2}$ of \mathbf{M} has 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. 9.1.1. Thus stepsize control must resort to tiny timesteps.



Can we predict this kind of difficulty ?

Example 9.2.2 (Chemical reaction kinetics).

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: ODE involving **concentrations** $\mathbf{y}(t) = (c_A(t), c_B(t), c_C(t), c_D(t))^T$

$$\dot{\mathbf{y}} := \frac{d}{dt} \begin{pmatrix} c_A \\ c_B \\ c_C \\ c_D \end{pmatrix} = \mathbf{f}(\mathbf{y}) := \begin{pmatrix} -k_1 c_A c_B + k_2 c_C - k_3 c_A c_C + k_4 c_D \\ -k_1 c_A c_B + k_2 c_C \\ k_1 c_A c_B - k_2 c_C - k_3 c_A c_C + k_4 c_D \\ k_3 c_A c_C - k_4 c_D \end{pmatrix}.$$

MATLAB computation: $t_0 = 0$, $T = 1$, $k_1 = 10^4$, $k_2 = 10^3$, $k_3 = 10$, $k_4 = 1$

Code 9.2.3: Explicit Integration of Stiff Equations of Chemical reactions

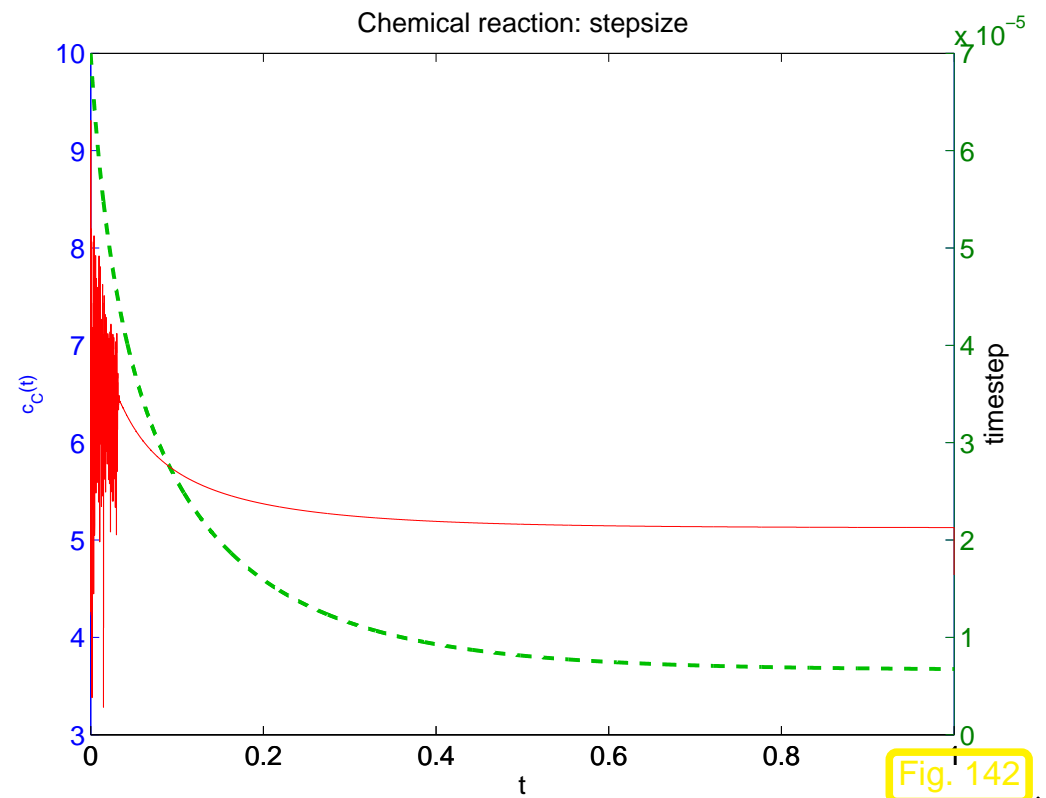
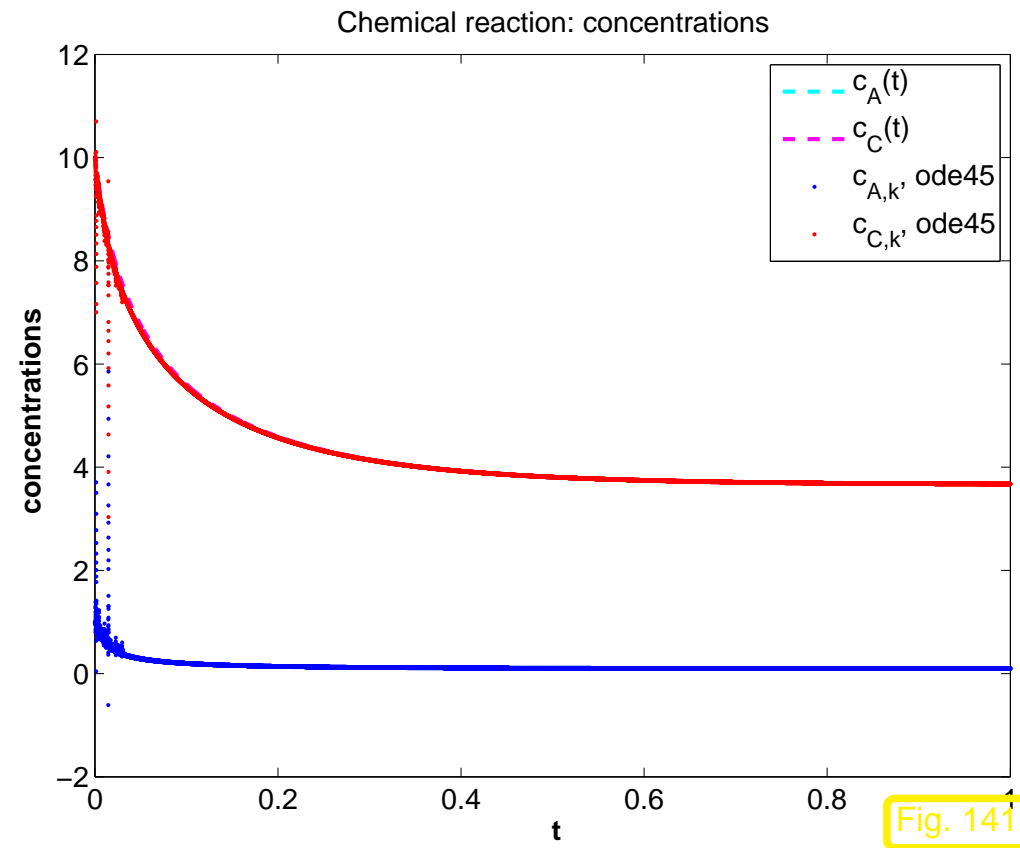
```
1 from numpy import array, linspace, double, diff
2 from ode45 import ode45
```



```
3 import matplotlib.pyplot as plt
4 from odewrapper import odewrapper
5
6 def chemstiff():
7     """ Simulation of kinetics of coupled chemical reactions with
8         vastly different reaction
9         rates, see \eqref{eq:chemstiff} for the ODE model.
10        reaction rates \Blue{k_1, k_2, k_3, k_4}, \Magenta{k_1, k_2 \gg k_3, k_4}.
11        """
12
13     k1 = 1e4; k2 = 1e3; k3 = 10; k4 = 1
14     # definition of right hand side function for ODE solver
15     fun = lambda t, y: array([-k1*y[0]*y[1] + k2*y[2] - k3*y[0]*y[2]
16                             + k4*y[3],
17                             -k1*y[0]*y[1] + k2*y[2],
18                             k1*y[0]*y[1] - k2*y[2] - k3*y[0]*y[2] + k4*y[3],
19                             k3*y[0]*y[2] - k4*y[3]])
20
21     tspan = [0, 1] # Integration time interval
22     L = tspan[1]-tspan[0] # Duration of simulation
23     y0 = array([1, 1, 10, 0]) # Initial value  $y_0$ 
24
25     # compute "exact" solution, using fortran routines (using wrapper for
26         scipy.integrate.ode class)
27     tex = linspace(0, 1, 201)
```

```
24 yex = odewrapper(fun , y0 , 0 , tex )
25
26 # Compute solution with ode45 and moderate tolerances
27 options = { 'RelTol':0.01 , 'AbsTol':0.001 , 'Stats': 'on' }
28 t , y = ode45(fun , [0 , 1] , y0 , ** options )
29 # plot the solution
30 plt.figure()
31 plt.plot(t , y[:,0] , 'b.' , label=r' $c_{A,k}$  , low tolerance')
32 plt.plot(t , y[:,2] , 'r.' , label=r' $c_{C,k}$  , low tolerance')
33 # plot 'exact' results
34 plt.plot(tex , yex[:,0] , 'c—' , linewidth=3 , label=r' $c_A(t)$ ')
35 plt.plot(tex , yex[:,2] , 'm—' , linewidth=3 , label=r' $c_C(t)$ ')
36 plt.axis([0 , 1 , 0 , 12])
37 plt.xlabel('t')
38 plt.ylabel('concentrations')
39 plt.title('Chemical_reaction:_concentrations')
40 plt.legend()
41 # plt.savefig('.../PICTURES/chemstiff.eps')
42
43 # Plot stepsizes together with "exact" solution
44 fig = plt.figure()
45 plt.title('Chemical_reaction:_stepsize')
46 ax1 = fig.add_subplot(111)
47 ax1.plot(tex , yex[:,2] , 'r—' , linewidth=2)
```

```
48     ax1.set_xlabel( 't' )
49     ax1.set_ylabel( 'c_C(t)' )
50
51     ax2 = ax1.twinx()
52     ax2.plot(0.5*(t[:-1]+t[1:]), diff(t), 'r-')
53     ax2.set_ylabel( 'timestep' )
54
55     plt.show()
56     #plt.savefig('chemstiffss.eps')
57
58 if __name__ == '__main__':
59     chemstiff()
```



Example 9.2.4 (Strongly attractive limit cycle).

Autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$

$$\mathbf{f}(\mathbf{y}) := \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \mathbf{y} + \lambda(1 - \|\mathbf{y}\|^2) \mathbf{y},$$

on state space $D = \mathbb{R}^2 \setminus \{0\}$.

Solution trajectories ($\lambda = 10$) ▷

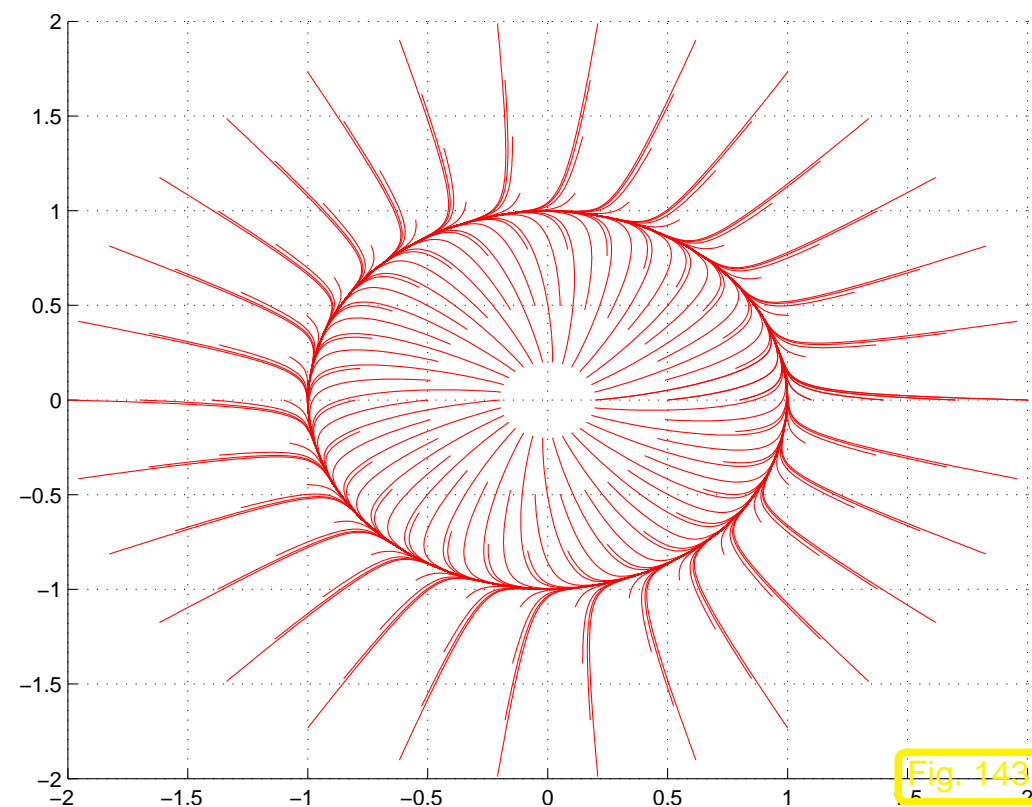


Fig. 143

Code 9.2.5: Integration of IVP with limit cycle

```

1 from ode45 import ode45
2 from numpy import *
3 import matplotlib.pyplot as plt
4 def limitcycle(lam, y0=array((1,0))):
5     # PYTHON script for solving limit cycle ODE (??)
6     # define right hand side vectorfield
7     fun = lambda t,y: array([-y[1],y[0]]) +
8         lam*(1-y[0]**2-y[1]**2)*y
9     # standard invocation of MATLAB integrator, see Ex. 8.6.7
10    tspan = (0,2*pi)
11    opts = {'stats':False, 'reltol':1e-4, 'abstol':1e-4}

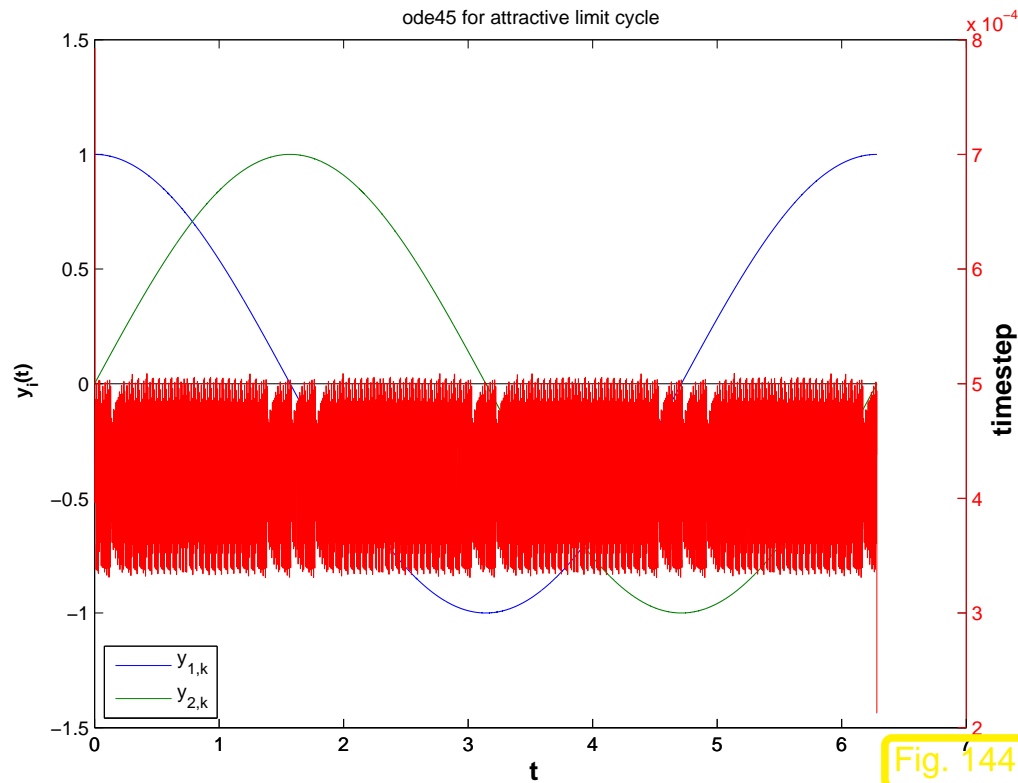
```

```
11     t45 , y45 = ode45(fun , tspan , y0 , ** opts )
12     return (t45 , y45)
13
14 if __name__ == '__main__':
15     for lam in [0,1000]:
16         t,y = limitcycle(lam)
17         fig = plt.figure()
18         plt.title(r' $\lambda =$ ' + str(lam) + ')
19         ax1 = fig.add_subplot(111)
20         ax1.plot(t,y)
21         ax1.set_ylabel(r'y_i(t)')
22         ax1.set_xlabel('t')
23         ax2 = ax1.twinx()
24         ax2.plot(0.5*(t[:-1]+t[1:]), diff(t), 'r-')
25         ax2.set_ylabel('timestep')
26         plt.show()
27
28 # now make some nice solution trajectories
29 lam = 10
30 plt.figure()
31 plt.title(r'Solution_Trajectories_( $\lambda = 10$ )')
32 for rad in r_[0.25:2.25:0.25]:
33     for phi in r_[0:2*pi:31j][:-1]:
34         x,y = rad*cos(phi), rad*sin(phi)
```

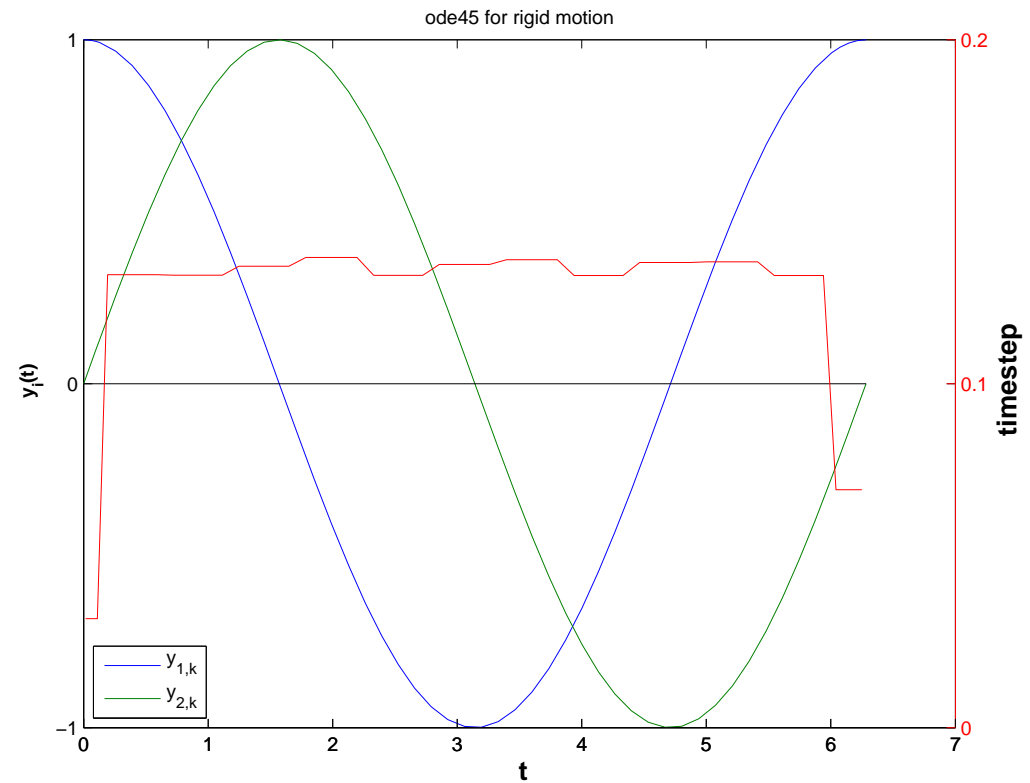
```

35 t,y = limitcycle(lam,array([x,y]))
36 plt.plot(y[:,0],y[:,1], 'r')
37 plt.grid(True)
38 plt.show()

```



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 λ 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.



Notion 9.2.1 (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.*

Typical features of stiff IVPs:

- Presence of **fast transients** in the solution, see Ex. 9.1.1, ??,
- Occurrence of **strongly attractive** fixed points/limit cycles, see Ex. 9.2.4

Code 9.2.6: An ODE wrapper

```
1 import scipy.integrate
2 from numpy import size, zeros
3
```



```
4 def odewrapper(f , y0 , t0 , t , integrator='dopri5') :
5     """ integrates the function f with initial values y0 , t0
6     returns the value of the integral at times specified in t
7     'dopri5' is the scipy.integrate analog of matlab's ode45
8     'ode15s' is the scipy.integrate analog of matlab's ode15s
9     """
10    # initialize ode class:
11    # constructor takes function to integrate
12    # second function sets initial values (for y and for t)
13    # third function sets integrator (default:'dopri5' is equivalent to ode45 in MATLAB)
14    if integrator=='dopri5':
15        r =
16            scipy.integrate.ode(f).set_initial_value(y0,t0).set_int
17    if integrator == 'ode15s':
18        print integrator+'\n'
19        r =
20            scipy.integrate.ode(f).set_initial_value(y0,t0).set_int
21            method='bdf', order=15)
22    # do the integration, read values at points in array t.
23    i=1; n=size(t)
24    y = zeros([n, size(y0)])
25    y[0]=y0
26    while r.successful() and i < n:
27        r.integrate(t[i])
```

```

25         y[i] = r.y
26         i += 1
27     return y

```

9.3 (Semi-)implicit Runge-Kutta methods

Example 9.3.1 (Implicit Euler timestepping for decay equation).

Again, **model problem analysis**: study implicit Euler method (8.2.4) for IVP (9.1.1)

$$\blacktriangleright \text{ sequence } y_k := \left(\frac{1}{1 - \lambda h} \right)^k y_0 . \quad (9.3.1)$$

$$\Rightarrow \boxed{\operatorname{Re} \lambda < 0 \Rightarrow \lim_{k \rightarrow \infty} y_k = 0 !} \quad (9.3.2)$$

No timestep constraint: qualitatively correct behavior of $(y_k)_k$ for $\operatorname{Re} \lambda < 0$ and **any** $h > 0$!



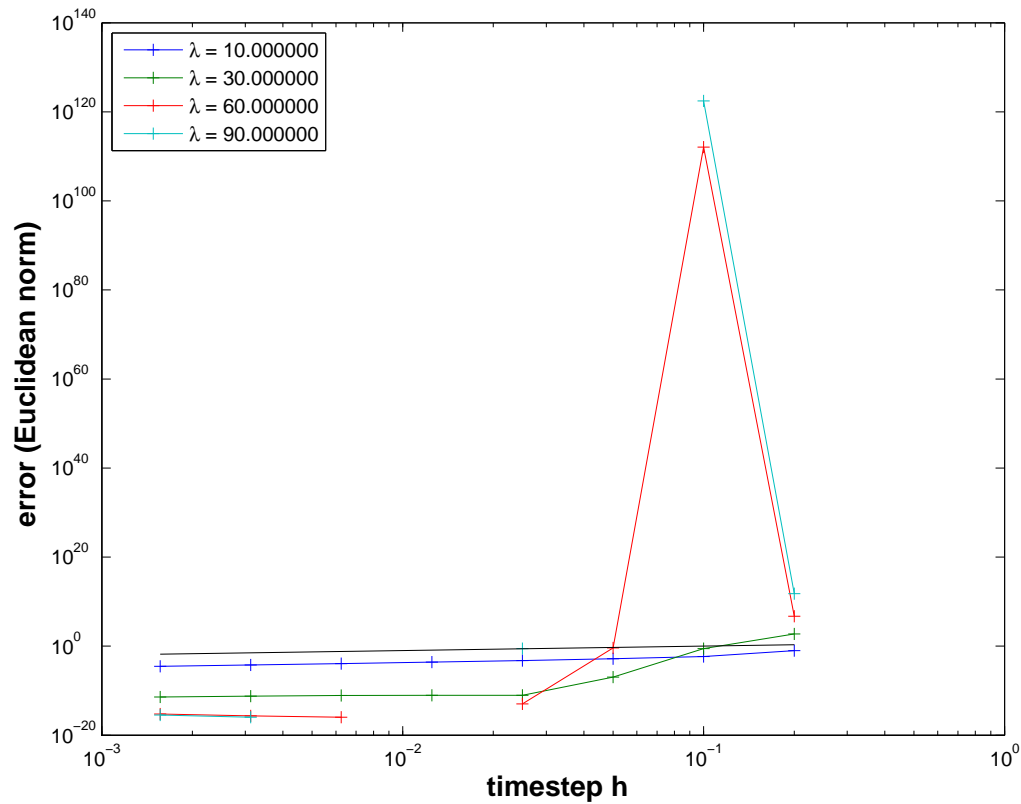
Observe: transformation idea, see (9.2.3), (9.2.5), applies to explicit *and implicit* Euler method alike.

Conjecture: implicit Euler method will not face timestep constraint for stiff problems (\rightarrow Notion 9.2.1).

Example 9.3.2 (Euler methods for stiff logistic IVP).

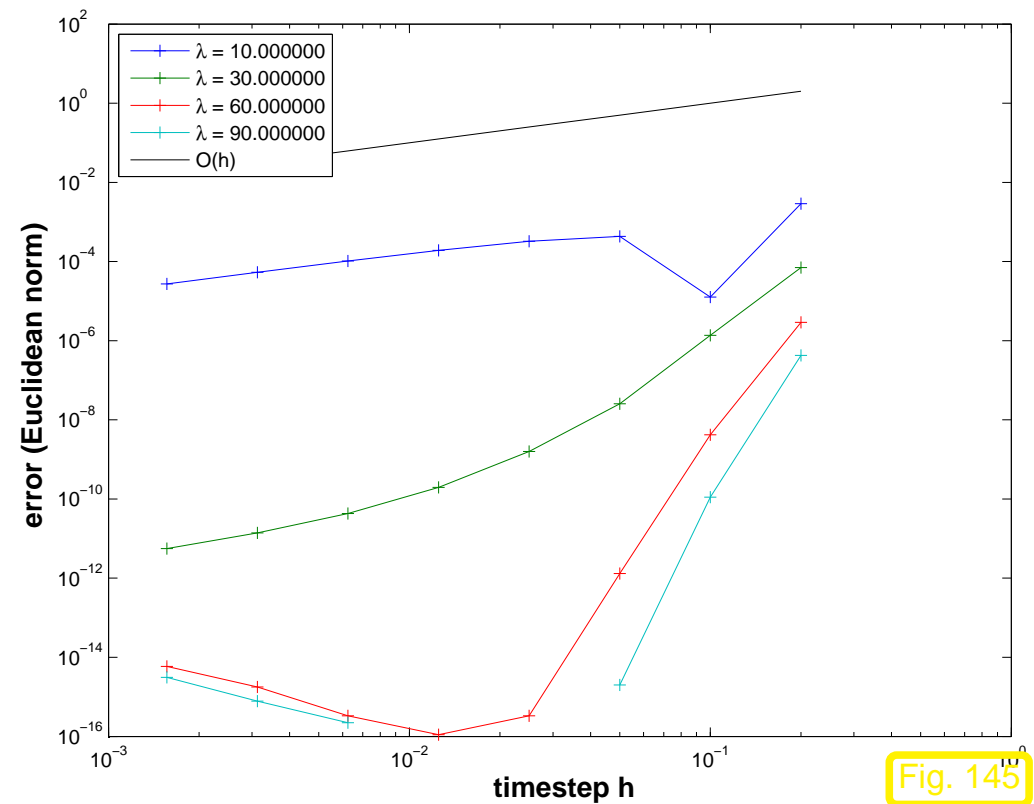
 Redo Ex. 9.1.1 for implicit Euler method:

Explicit Euler method (8.2.1)



λ large: blow-up of y_k for large timestep h

Implicit Euler method (8.2.4)



λ large: stable for all timesteps h !

Fig. 145

Well, we see what we expected!



Unfortunately the implicit Euler method is of first order only, see Ex. 8.3.1. Can the Runge-Kutta design principle for integrators also yield higher order methods, which can cope with stiff problems.

YES !

Definition 9.3.1 (General Runge-Kutta method). (cf. Def. 8.6.1)

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 (8.1.11) 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)

Shorthand notation for Runge-Kutta methods

Butcher scheme



$$\frac{\mathbf{c} \mid \mathfrak{A}}{\mathbf{b}^T} := \frac{\begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \end{array}}{\begin{array}{c|ccc} b_1 & \cdots & b_s \end{array}} . \quad (9.3.3)$$

Note: now \mathfrak{A} can be a general $s \times s$ -matrix.

- \mathfrak{A} strict lower triangular matrix ➤ explicit Runge-Kutta method, Def. 8.6.1
- \mathfrak{A} lower triangular matrix ➤ diagonally-implicit Runge-Kutta method (DIRK)

Model problem analysis for general Runge-Kutta single step methods (\rightarrow Def. 9.3.1): exactly the same as for explicit RK-methods, see (9.1.6), (9.1.7)!

Theorem 9.3.2 (Stability function of Runge-Kutta methods).

The discrete evolution Ψ_λ^h of an s -stage Runge-Kutta single step method (\rightarrow Def. 9.3.1) with Butcher scheme $\frac{\mathbf{c} \mid \mathfrak{A}}{\mathbf{b}^T}$ (see (9.3.3)) for the ODE $\dot{y} = \lambda y$ is a multiplication operator according to

$$\Psi_\lambda^h = \underbrace{1 + z \mathbf{b}^T (\mathbf{I} - z \mathfrak{A})^{-1} \mathbf{1}}_{\text{stability function } S(z)} = \frac{\det(\mathbf{I} - z \mathfrak{A} + z \mathbf{1} \mathbf{b}^T)}{\det(\mathbf{I} - z \mathfrak{A})}, \quad z := \lambda h, \quad \mathbf{1} = (1, \dots, 1)^T \in \mathbb{R}^s.$$

Note: from the determinant representation of $S(z)$ we infer that the stability function of an s -stage Runge-Kutta method is a **rational function** of the form $S(z) = \frac{P(z)}{Q(z)}$ with $P \in \mathcal{P}_s$, $Q \in \mathcal{P}_s$.

Of course, such rational functions can satisfy $|S(z)| < 1$ for all $z < 0$. For example, the stability function of the implicit Euler method (8.2.4) is

$$\frac{1|1}{1} \xRightarrow{\text{Thm. 9.3.2}} S(z) = \frac{1}{1-z}. \quad (9.3.4)$$

In light of the previous detailed analysis we can now state what we expect from the stability function of a Runge-Kutta method that is suitable for stiff IVP (\rightarrow Notion 9.2.1):

Definition 9.3.3 (L-stable Runge-Kutta method).

A Runge-Kutta method (\rightarrow Def. 9.3.1) is **L-stable/asymptotically stable**, if its stability function (\rightarrow Def. 9.3.2) satisfies

$$(i) \quad \operatorname{Re} z < 0 \Rightarrow |S(z)| < 1, \quad (9.3.5)$$

$$(ii) \quad \lim_{\operatorname{Re} z \rightarrow -\infty} S(z) = 0. \quad (9.3.6)$$

Remark 9.3.3 (Necessary condition for L-stability of Runge-Kutta methods).

Consider: Runge-Kutta method (\rightarrow Def. 9.3.1) with Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$

Assume: $\mathfrak{A} \in \mathbb{R}^{s,s}$ is regular

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 :

$$\text{Thm. 9.3.2} \Rightarrow S(-\infty) = 1 - \mathbf{b}^T \mathfrak{A}^{-1} \mathbf{1}. \quad (9.3.7)$$

► If $\mathbf{b}^T = (\mathfrak{A})_{:,j}^T$ (row of \mathfrak{A}) $\Rightarrow S(-\infty) = 0$. (9.3.8)

Butcher scheme (9.3.3) for L-stable RK-methods, see Def. 9.3.3

$$\triangleright \begin{array}{c|c} \mathbf{c} & \mathcal{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_{s-1} & a_{s-1,1} & \cdots & a_{s-1,s} \\ \hline 1 & b_1 & \cdots & b_s \\ \hline & b_1 & \cdots & b_s \end{array} .$$

Example 9.3.4 (L-stable implicit Runge-Kutta methods).

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$$

Implicit Euler method

$$\begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ \hline 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}$$

Radau RK-SSM, order 3

$$\begin{array}{c|ccc} \frac{4-\sqrt{6}}{10} & \frac{88-7\sqrt{6}}{360} & \frac{296-169\sqrt{6}}{1800} & \frac{-2+3\sqrt{6}}{225} \\ \hline \frac{4+\sqrt{6}}{10} & \frac{296+169\sqrt{6}}{1800} & \frac{88+7\sqrt{6}}{360} & \frac{-2-3\sqrt{6}}{225} \\ \hline 1 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \\ \hline & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \end{array}$$

Radau RK-SSM, order 5



Equations fixing increments $\mathbf{k}_i \in \mathbb{R}^d$, $i = 1, \dots, s$, for s -stage implicit RK-method

=

(Non-)linear system of equations with $s \cdot d$ unknowns



Example 9.3.5 (Linearization of increment equations).

- Initial value problem for logistic ODE, see Ex. 8.1.1

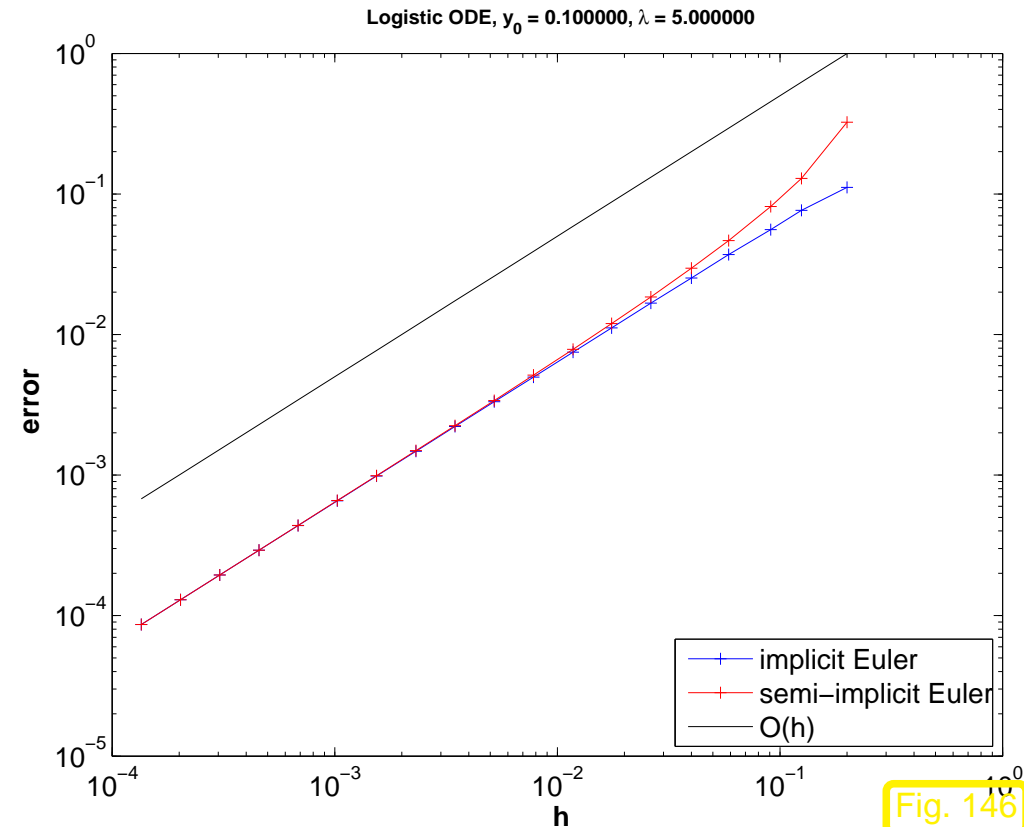
$$\dot{y} = \lambda y(1 - y) \quad , \quad y(0) = 0.1 \quad , \quad \lambda = 5 \quad .$$

- Implicit Euler method (8.2.4) 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 (8.2.4) with timestep $h > 0$

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}(\mathbf{y}_{k+1}) \Leftrightarrow F(\mathbf{y}_{k+1}) := \mathbf{y}_{k+1} - h\mathbf{f}(\mathbf{y}_{k+1}) - \mathbf{y}_k = 0 \quad .$$

One Newton step applied to $F(\mathbf{y}) = 0$ with initial guess \mathbf{y}_k yields

$$\mathbf{y}_{k+1} = \mathbf{y}_k - D\mathbf{f}(\mathbf{y}_k)^{-1}F(\mathbf{y}_k) = \mathbf{y}_k + (\mathbf{I} - hD\mathbf{f}(\mathbf{y}_k))^{-1}h\mathbf{f}(\mathbf{y}_k) \quad .$$

$$\mathbf{f}(\mathbf{y}) = \mathbf{A}\mathbf{y} \quad \mathbf{A} \in \mathbb{R}^{d,d}$$

$$y_{k+1}$$



Idea: Use **linearized increment equations** for implicit RK-SSM

$$\mathbf{k}_i = \mathbf{f}(\mathbf{y}_0) + hD\mathbf{f}(\mathbf{y}_0) \left(\sum_{j=1}^s a_{ij} \mathbf{k}_j \right), \quad i = 1, \dots, s. \quad (9.3.9)$$

Linearization does nothing for linear ODEs ➤ stability function (→ Thm. 9.3.2) not affected!

► Class of **semi-implicit (linearly implicit) Runge-Kutta methods** (Rosenbrock-Wanner (ROW) methods):

$$(\mathbf{I} - ha_{ii}\mathbf{J})\mathbf{k}_i = \mathbf{f}(\mathbf{y}_0 + h \sum_{j=1}^{i-1} (a_{ij} + d_{ij})\mathbf{k}_j) - h\mathbf{J} \sum_{j=1}^{i-1} d_{ij}\mathbf{k}_j, \quad (9.3.10)$$


$$\mathbf{J} := D\mathbf{f}\left(\mathbf{y}_0 + h \sum_{j=1}^{i-1} (a_{ij} + d_{ij})\mathbf{k}_j\right), \quad (9.3.11)$$

$$\mathbf{y}_1 := \mathbf{y}_0 + \sum_{j=1}^s b_j \mathbf{k}_j. \quad (9.3.12)$$

Remark 9.3.6 (Adaptive integrator for stiff problems in MATLAB).

Handle of type $@(\mathbf{t}, \mathbf{y}) \rightarrow \mathbf{J}(\mathbf{t}, \mathbf{y})$ to Jacobian $D\mathbf{f} : I \times D \mapsto \mathbb{R}^{d,d}$

```
opts = odeset('abstol',atol,'reltol',rtol,'Jacobian',J)
[t,y] = ode23s(odefun,tspan,y0,opts);
```



Stepsize control according to policy of Sect. 8.7:

$\Psi \hat{=}$ RK-method of order 2 $\tilde{\Psi} \hat{=}$ RK-method of order 3

ode23s

integrator for stiff IVP



9.4 Essential Skills Learned in Chapter 9

You should know:

- what is a stiff problem with concrete examples
- the typical situations when stiff problems occur
- the meaning of the stability function associated with a Runge-Kutta method
- the importance of the implicit Runge-Kutta methods
- example of semi-implicit methods
- an adaptive integrator for stiff-problems

Bibliography

- [1] H. AMANN, *Gewöhnliche Differentialgleichungen*, Walter de Gruyter, Berlin, 1st ed., 1983.
- [2] S. AMAT, C. BERMUDEZ, S. BUSQUIER, AND S. PLAZA, *On a third-order Newton-type method free of bilinear operators*, Numerical Lin. Alg., (2010). DOI: 10.1002/nla.654.
- [3] C. BISCHOF AND C. VAN LOAN, *The WY representation of Householder matrices*, SIAM J. Sci. Stat. Comput., 8 (1987).
- [4] S. BÖRM, L. GRASEDYCK, AND W. HACKBUSCH, *Introduction to hierarchical matrices with applications*, Engineering Analysis with Boundary Elements, 27 (2003), pp. 405–422.
- [5] F. BORNEMANN, *A model for understanding numerical stability*, IMA J. Numer. Anal., 27 (2006), pp. 219–231.
- [6] E. BRIGHAM, *The Fast Fourier Transform and Its Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

- [7] 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 (1995), pp. 405–417.
- [8] D. COPPERSMITH AND T. RIVLIN, *The growth of polynomials bounded at equally spaced points*, SIAM J. Math. Anal., 23 (1992), pp. 970–983.
- [9] D. COPPERSMITH AND S. WINOGRAD, *Matrix multiplication via arithmetic progression*, J. Symbolic Computing, 9 (1990), pp. 251–280.
- [10] W. DAHMEN AND A. REUSKEN, *Numerik für Ingenieure und Naturwissenschaftler*, Springer, Heidelberg, 2006.
- [11] M. DEAKIN, *Applied catastrophe theory in the social and biological sciences*, Bulletin of Mathematical Biology, 42 (1980), pp. 647–679.
- [12] P. DEUFLHARD, *Newton Methods for Nonlinear Problems*, vol. 35 of Springer Series in Computational Mathematics, Springer, Berlin, 2004.
- [13] P. DEUFLHARD AND F. BORNEMANN, *Numerische Mathematik II*, DeGruyter, Berlin, 2 ed., 2002.
- [14] P. DEUFLHARD AND A. HOHMANN, *Numerische Mathematik I*, DeGruyter, Berlin, 3 ed., 2002.
- [15] P. DUHAMEL AND M. VETTERLI, *Fast fourier transforms: a tutorial review and a state of the art*, Signal Processing, 19 (1990), pp. 259–299.
- [16] A. DUTT AND V. ROKHLIN, *Fast Fourier transforms for non-equispaced data II*, Appl. Comput. Harmon. Anal., 2 (1995), pp. 85–100.

- [17] F. FRITSCH AND R. CARLSON, *Monotone piecewise cubic interpolation*, SIAM J. Numer. Anal., 17 (1980), pp. 238–246.
- [18] M. GANDER, W. GANDER, G. GOLUB, AND D. GRUNTZ, *Scientific Computing: An introduction using MATLAB*, Springer, 2005. In Vorbereitung.
- [19] J. GILBERT, C. MOLER, AND R. SCHREIBER, *Sparse matrices in MATLAB: Design and implementation*, SIAM Journal on Matrix Analysis and Applications, 13 (1992), pp. 333–356.
- [20] G. GOLUB AND C. VAN LOAN, *Matrix computations*, John Hopkins University Press, Baltimore, London, 2nd ed., 1989.
- [21] C. GRAY, *An analysis of the Belousov-Zhabotinski reaction*, Rose-Hulman Undergraduate Math Journal, 3 (2002). <http://www.rose-hulman.edu/mathjournal/archives/2002/vol3-n1/paper1/v3n1-1pd.pdf>.
- [22] L. GREENGARD AND V. ROKHLIN, *A new version of the fast multipole method for the Laplace equation in three dimensions*, Acta Numerica, (1997), pp. 229–269.
- [23] M. GUTKNECHT, *Lineare algebra*, lecture notes, SAM, ETH Zürich, 2009. <http://www.sam.math.ethz.ch/~mhg/unt/LA/HS07/>.
- [24] W. HACKBUSCH, *Iterative Lösung großer linearer Gleichungssysteme*, B.G. Teubner–Verlag, Stuttgart, 1991.
- [25] —, *Iterative Solution of Large Sparse Systems of Equations*, vol. 95 of Applied Mathematical Sciences, Springer-Verlag, New York, 1993.

- [26] W. HACKBUSCH AND S. BÖRM, *Data-sparse approximation by adaptive \mathcal{H}^2 -matrices*, Computing, 69 (2002), pp. 1–35.
- [27] E. HAIRER, C. LUBICH, AND G. WANNER, *Geometric numerical integration*, vol. 31 of Springer Series in Computational Mathematics, Springer, Heidelberg, 2002.
- [28] C. HALL AND W. MEYER, *Optimal error bounds for cubic spline interpolation*, J. Approx. Theory, 16 (1976), pp. 105–122.
- [29] M. HANKE-BOURGEOIS, *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens*, Mathematische Leitfäden, B.G. Teubner, Stuttgart, 2002.
- [30] N. HIGHAM, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, PA, 2 ed., 2002.
- [31] S. JOHNSON, *Notes on the convergence of trapezoidal-rule quadrature*. MIT online course notes, <http://math.mit.edu/~stevenj/trapezoidal.pdf>, 2008.
- [32] M. KOWARSCHIK AND W. C, *An overview of cache optimization techniques and cache-aware numerical algorithms*, in Algorithms for Memory Hierarchies, vol. 2625 of Lecture Notes in Computer Science, Springer, Heidelberg, 2003, pp. 213–232.
- [33] A. LALIENA AND F.-J. SAYAS, *Theoretical aspects of the application of convolution quadrature to scattering of acoustic waves*, Numer. Math., 112 (2009), pp. 637–678.
- [34] A. LENGVILLE AND C. MEYER, *Google’s PageRank and Beyond: The Science of Search Engine Rankings*, Princeton University Press, Princeton, NJ, 2006.

- [35] D. McALLISTER AND J. ROULIER, *An algorithm for computing a shape-preserving osculatory quadratic spline*, ACM Trans. Math. Software, 7 (1981), pp. 331–347.
- [36] C. MOLER, *Numerical Computing with MATLAB*, SIAM, Philadelphia, PA, 2004.
- [37] K. NEYMEYR, *A geometric theory for preconditioned inverse iteration applied to a subspace*, Tech. Rep. 130, SFB 382, Universität Tübingen, Tübingen, Germany, November 1999. Submitted to Math. Comp.
- [38] —, *A geometric theory for preconditioned inverse iteration: III. Sharp convergence estimates*, Tech. Rep. 130, SFB 382, Universität Tübingen, Tübingen, Germany, November 1999.
- [39] K. NIPP AND D. STOFFER, *Lineare Algebra*, vdf Hochschulverlag, Zürich, 5 ed., 2002.
- [40] M. OVERTON, *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM, Philadelphia, PA, 2001.
- [41] A. D. H.-D. QI, L.-Q. QI, AND H.-X. YIN, *Convergence of Newton's method for convex best interpolation*, Numer. Math., 87 (2001), pp. 435–456.
- [42] A. QUARTERONI, R. SACCO, AND F. SALERI, *Numerical mathematics*, vol. 37 of Texts in Applied Mathematics, Springer, New York, 2000.
- [43] C. RADER, *Discrete Fourier transforms when the number of data samples is prime*, Proceedings of the IEEE, 56 (1968), pp. 1107–1108.
- [44] R. RANNACHER, *Einführung in die numerische mathematik*. Vorlesungsskriptum Universität Heidelberg, 2000. <http://gaia.iwr.uni-heidelberg.de/>.

- [45] V. ROKHLIN, *Rapid solution of integral equations of classical potential theory*, J. Comp. Phys., 60 (1985), pp. 187–207.
- [46] A. SANKAR, D. SPIELMAN, AND S.-H. TENG, *Smoothed analysis of the condition numbers and growth factors of matrices*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 446–476.
- [47] T. SAUER, *Numerical analysis*, Addison Wesley, Boston, 2006.
- [48] J.-B. SHI AND J. MALIK, *Normalized cuts and image segmentation*, IEEE Trans. Pattern Analysis and Machine Intelligence, 22 (2000), pp. 888–905.
- [49] M. STEWART, *A superfast toeplitz solver with improved numerical stability*, SIAM J. Matrix Analysis Appl., 25 (2003), pp. 669–693.
- [50] J. STOER, *Einführung in die Numerische Mathematik*, Heidelberger Taschenbücher, Springer, 4 ed., 1983.
- [51] V. STRASSEN, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.
- [52] M. STRUWE, *Analysis für informatiker*. Lecture notes, ETH Zürich, 2009. <https://moodle-app1.net.ethz.ch/lms/mod/resource/index.php?id=145>.
- [53] F. TISSEUR AND K. MEERBERGEN, *The quadratic eigenvalue problem*, SIAM Review, 43 (2001), pp. 235–286.
- [54] L. TREFETHEN AND D. BAU, *Numerical Linear Algebra*, SIAM, Philadelphia, PA, 1997.
- [55] P. VERTESI, *On the optimal lebesgue constants for polynomial interpolation*, Acta Math. Hungaria, 47 (1986), pp. 165–178.
- [56] —, *Optimal lebesgue constant for lagrange interpolation*, SIAM J. Numer. Aanal., 27 (1990), pp. 1322–1331.

Index

- LU -decomposition
 - existence, 117
- 3-term recursion
 - for Chebychev polynomials, 291
- a posteriori error bound, 40
- absolute tolerance, 518
- adaptive multigrid quadrature, 399
- adaptive quadrature, 398
- AGM, 34
- algebraic convergence, 278
- algorithm
 - Clenshaw, 303
- asymptotic error behavior, 276
- asymptotic rate of linear convergence, 52
- backward substitution, 117
- Banach's fixed point theorem, 48
- basis
 - orthonormal, 134, 193
 - trigonometrical, 312
- Belousov-Zhabotinsky reaction, 511
- bimolecular reaction, 437
- bisection, 58
- blow-up, 514
- Broyden
 - Quasi-Newton Method, 103
- Broyden-Verfahren
 - convergence monitor, 106
- Butcher scheme, 504, 578
- cancellation, 88, 89

for Householder transformation, 125	of iterative methods, 23
chain rule, 84	fixed point iteration, 43
characteristic polynomial, 190	constant
Chebyshev nodes, 294, 296	Lebesgue, 296
Chebyshev polynomials	constitutive relations, 262
3-term recursion, 291	convergence
Chebyshev-interpolation, 288, 353	algebraic, 278
chemical reaction kinetics, 564	algebraic for trigonometric interpolation, 347
Classical Runge-Kutta method	asymptotic, 73
Butcher scheme, 505	exponential, 278, 286, 298
Clenshaw algorithm, 303	exponential bei trigonometric interpolation, 347
column sum norm, 29	global, 24
complexity	iterative method, 23
of SVD, 147	linear, 26
composite quadrature, 378	linear in Gauss-Newton method, 181
computational costs	local, 24
QR-decomposition, 136	local quadratic in damped Newton method, 180
computational effort	quadratic, 34
eigenvalue computation, 199	rate, 26
condition number	convergence monitor
of a matrix, 118	of Broyden method, 106
consistency	

damped Newton method, 96	error behavior
damping factor, 97	asymptotic, 276
definite, 27	error estimator
DFT, 314	a posteriori, 40
diagonal matrix, 116	Euler method
diagonalization	explicit, 449
of a matrix, 193	implicit, 452
difference method, 471	semi implicit, 582
difference quotient, 86	Euler polygon, 450
difference scheme, 450	Euler's iteration, 69
differential in non-linear least squares, 84	explicit Euler method, 449
direct power method, 209	Butcher scheme, 504
discrete Fourier transform, 312	explicit midpoint rule
divided differences, 272	Butcher scheme, 505
	for ODEs, 501
economical singular value decomposition, 146	explicit Runge-Kutta method, 503
eigenspace, 190	explicit trapezoidal rule
eigenvalue, 190	Butcher scheme, 505
eigenvector, 190	exponential convergence, 298
energy drift, 18, 474, 480, 484	extended state space
Equation	of an ODE, 441
non-linear, 20	

fast Fourier transform, 320	Givens rotation, 126
FFT, 320	Givens-Rotation, 139
fit	global solution
polynomial, 156	of an IVP, 445
fixed point, 43	Golub-Welsch algorithm, 368
fixed point form, 43	gradient, 85
fixed point iteration	Gram-Schmidt
consistency, 43	Orthonormalisierung, 237
forward substitution, 117	Gram-Schmidt orthogonalization, 237
Fourier	Halley's iteration, 63, 69
Koeffizienten, 308	Hamilton-function
matrix, 313	molekular dynamics, 487
Reihe, 308	Hessian matrix, 85
Fourier transform	homogeneous, 27
discrete, 312	Horner scheme, 268
fractional order of convergence, 72	Householder reflection, 123
Funktion	implicit Euler method, 452
shandles, 80	implicit midpoint rule, 470
Gauss-Newton method, 176	increment equations
Gerschgorin circle theorem, 191	linearized, 583
Gibbsches Phänomen, 348	increments

Runge-Kutta, 503, 577	Jacobian, 49, 78
ine-step method, 476	kinetics
initial guess, 23, 42	of chemical reaction, 564
initial value problem	Konvergenz
stiff, 572	Algebraische, Quadratur, 375
initial value problem (IVP), 440	Krylov space, 236
Initialisation, 476	L-stable, 580
intermediate value theorem, 58	Landau-O, 278
interpolation	least squares, 157
Chebychev, 288, 353	least squares problem
trigonometric, 353	conditioning, 159
inverse interpolation, 74	Lebesgue
inverse iteration, 214	constant, 296
preconditioned, 220	Lenard-Jones-Potential, 487
iteration	Lie-Trotter-Splitting, 494
Halley's, 69	limit cycle, 568
Euler's, 69	linear correlation, 149
quadratical inverse interpolation, 69	linear ordinary differential equation, 188
iteration function, 23, 42	local a posteriori error estimation
iterative method	for adaptive quadrature, 399
convergence, 23	local convergence
IVP, 440	

Newton method, 96	upper triangular, 116
local linearization, 78	matrix faktORIZATION, 114
local mesh refinement	matrix norm, 28
for adaptive quadrature, 399	column sums, 29
Logistic differential equation, 495	row sums, 29
Lotka-Volterra ODE, 435	Matrixnorm, 28
lower triangular matrix, 116	Submultiplikativität, 29
Matrix	mesh
Hermitian, 193	in time, 455
normal, 193	Method
skew-Hermitian, 193	Quasi-Newton, 102
unitary, 193	method
matrix	Runge-Kutta, classical, 508
condition number, 118	midpoint rule, 501
diagonal, 116	implicit, 470
Fourier, 313	Mittelpunktsregel
Hessian, 85	explizit, 506
lower triangular, 116	model function, 60
normalized, 116	Modellfunktionsverfahren, 60
orthogonal, 120	molecular dynamics, 490
unitary, 120	molekular dnamics, 486
	monomial representation

of a polynomial, 267	local quadratic convergence, 91
monomials, 267	region of convergence, 96
Monte-Carlo quadrature, 415	node
multi-point methods, 60, 70	quadrature, 362
multiplicity	nodes
geometric, 190	Chebyshev, 294
Newton	Chebyshev nodes, 296
basis, 270	non-linear data fitting, 172
damping, 97	norm, 27
damping factor, 97	∞ -, 27
monotonicity test, 98	1-, 27
simplified method, 85	Euclidean, 27
Newton correction, 78	of matrix, 28
simplified, 95	Sobolev semi-, 287
Newton iteration, 78	normal equations, 162
numerical Differentiation, 86	normal mode analysis, 184
termination criterion, 93	normalized lower triangular matrix, 115
Newton method	normalized triangular matrix, 116
1D, 61	Nullstellenbestimmung
damped, 96	Modellfunktionsverfahren, 60
local convergence, 96	Numerical differentiation
	roundoff, 88

numerical Differentiation	phase space
Newton iteration, 86	of an ODE, 441
numerical quadrature, 359	Picard-Lindelöf
numerical rank, 170	Theorem of, 444
numerischer Integrator, 482	PINVIT, 220
ODE, 440	polynomial
one-point methods, 60	characteristic, 190
order of convergence, 32	polynomial space, 267
fractional, 72	polynomiales fit, 156
ordinary differential equation	preconditioned inverse iteration, 220
linear, 188	predator-prey model, 435
ordinary differential equation (ODE), 440	principal axis transformation, 193
Oregonator, 439	principal component, 151
oregonator, 511	principal component analysis, 139
orthogonal matrix, 120	product rule, 85
orthonormal basis, 134, 193	pwer method
	direct, 209
PCA, 139	QR algorithm, 195
Peano	QR-algorithm with shift, 197
Theorem of, 444	QR-decomposition
Phänomen	computational costs, 136
Gibbsches, 348	

QR-factorization, QR-decomposition, 127

quadratic convergence, 55

quadratic inverse interpolation, 76

quadratical inverse interpolation, 69

quadrature

adaptive, 398

quadrature formula, 362

quadrature node, 362

quadrature numerical, 359

quadrature weight, 362

Quasi-Newton Method, 103

Quasi-Newton method, 102

Radau RK-method

order 3, 581

order 5, 581

rank

computation, 147

numerical, 170

rate

of algebraic convergence, 278

of convergence, 26

Rayleigh quotient, 209

Rayleigh quotient iteration, 217

recursion

3-term, 291

Reihe

Fourier, 308

relative tolerance, 518

residual quantity, 222

Riccati differential equation, 450

richt hand side

of an ODE, 441

Ritz projection, 234

RK4, 508

roundoff

for numerical differentiation, 88

row sum norm, 29

rule

Kuttas 3/8, 509

trapezoidal, explicit, 507

Runge-Kutta

3/8-rule, 509

classical, 508

increments, 503, 577	spectral radius, 191
Runge-Kutta method, 503, 577	spectrum, 190
L-stable, 580	Splitting
Runge-Kutta methods	Lie-Trotter, 494
stability function, 557, 578	Strang, 494
scheme	splitting method
Horner, 268	inexact, 497
Schur's lemma, 192	splitting methods, 494
secant condition, 102	Störmer-Verlet method, 475
secant method, 70, 102	Molecular dynamics, 487
semi-implicit Euler method, 582	stability function
seminorm, 287	of explicit Runge-Kutta methods, 557
shifted inverse iteration, 215	of Runge-Kutta methods, 578
similarity	state space
of matrices, 192	of an ODE, 441
similarity transformations, 192	stationary point, 437
similarity transformation	stiff IVP, 572
unitary, 196	Strang-Splitting, 494
Simpson rule, 366	sub-multiplicative, 29
single step method, 455	subspace iteration
singular value decomposition, 139, 143	for direct power method, 233
	SVD, 139, 143

Taylor expansion, 54	unitary matrix, 120
Taylor's formula, 54	unitary similiary transformation, 196
termination criterion, 36	upper Hessenberg matrix, 239
Newton iteration, 93	upper triangular matrix, 115, 116
reliable, 37	weight
residual based, 38	quadrature, 362
timestep constraint, 558	
tolerance, 38	Zerlegung
absolute, 518	LU, 117
for adaptive timestepping for ODEs, 517	zero padding, 340
realtive, 518	
trajectory, 436	
transform	
fast Fourier, 320	
trapezoidal rule, 365, 501	
explicit, 507	
for ODEs, 501	
triangle inequality, 27	
trigonimetrical basis, 312	
trigonometric interpolation, 353	
trust region method, 181	
two-step method, 476	

List of Symbols

$D\Phi \hat{=}$ **Jacobian** of $\Phi : D \mapsto \mathbb{R}^n$ at $\mathbf{x} \in D$, 49
 $D_{\mathbf{y}}\mathbf{f} \hat{=}$ Derivative of \mathbf{f} w.r.t.. \mathbf{y} (Jacobian), 444
 $J(t_0, \mathbf{y}_0) \hat{=}$ maximal domain of definition of a solution of an IVP, 444
 $\text{Eig}_{\mathbf{A}}(\lambda) \hat{=}$ eigenspace of \mathbf{A} for eigenvalue λ , 190
 $\text{Im}(\mathbf{A}) \hat{=}$ range/column space of matrix \mathbf{A} , 147
 $\text{Ker}(\mathbf{A}) \hat{=}$ nullspace of matrix \mathbf{A} , 147
 $\mathcal{K}_l(\mathbf{A}, \mathbf{z}) \hat{=}$ Krylov subspace, 236
 \mathcal{P}_k , 267
 $\Psi^h_{\mathbf{y}} \hat{=}$ discrete evolution for autonomous ODE, 456
 \mathbf{A}^+ , 159
 $\mathbf{1} = (1, \dots, 1)^T$, 557, 578

$\text{cond}(\mathbf{A})$, 118
 $\rho(\mathbf{A}) \hat{=}$ spectral radius of $\mathbf{A} \in \mathbb{K}^{n,n}$, 191
 $\rho_{\mathbf{A}}(\mathbf{u}) \hat{=}$ Rayleigh quotient, 209
 $\mathbf{f} \hat{=}$ right hand side of an ODE, 441
 $y[t_i, \dots, t_{i+k}] \hat{=}$ divided difference, 272
 $\|\mathbf{x}\|_1$, 27
 $\|\mathbf{x}\|_2$, 27
 $\|\mathbf{x}\|_{\infty}$, 27
 $\dot{} \hat{=}$ Derivative w.r.t. time t , 434
 TOL tolerance, 517

List of Definitions

- Chebyshev polynomial, 289
- Condition (number) of a matrix, 118
- Consistency of fixed point iterations, 43
- Consistency of iterative methods, 23
- Contractive mapping, 47
- Convergence, 23
 - global, 24
 - local, 24
- diskrete
 - Fourier transform, 314
- eigenvalues and eigenvectors, 190
- equivalence of norms, 27
- Evolution operator, 446
- Explicit Runge-Kutta method, 503
- Fourier transform, 314
- Hessian matrix, 85
- Krylov space, 236
- L-stable Runge-Kutta method, 580
- Linear convergence, 26
- Lipschitz continuous function, 443
- matrix
 - generalized condition number, 159
- matrix norm, 28
- norm, 27
- Order of convergence, 32
- orthogonal matrix, 120

polynomial

Chebyshev, 289

pseudoinverse, 159

Rayleigh quotient, 209

Runge-Kutta method, 577

Single step method, 455

singular value decomposition (SVD), 143

transform

Fourier, 314

Types of matrices, 116

unitary matrix, 120

Examples and Remarks

L^2 -error estimates for polynomial interpolation, 286	Efficiency, 317
h -adaptive numerical quadrature, 404	3-term recursion for Chebychev polynomials, 291
[From higher order ODEs to first order systems, 442	A posteriori error bound for linearly convergent iteration, 40
[Many particles molecular dynamics, 490	A posteriori termination criterion for linearly con- vergent iterations, 39
[Stable solution of LSE by means of QR-decomposition, 137	Adapted Newton method, 65
ode45 for stiff problem, 546	Adaptive integrators for stiff problems in MAT- LAB, 584
“Butcher barriers” for explicit RK-SSM, 509	Adaptive quadrature in Python, 407
“Failure” of adaptive timestepping, 532	Adaptive timestepping for mechanical problem, 540
$\mathbf{B} = \mathbf{B}^H$ s.p.d. mit Cholesky-Zerlegung, 194	Affine invariance of Newton method, 83
L-stable implicit Runge-Kutta methods, 581	Aliasing, 351
fft	

<p>Analytic solution of homogeneous linear ordinary differential equations, 188</p> <p>approximation</p> <p> uses of, 265</p> <p>Approximation by polynomials, 275</p> <p>Arnoldi process Ritz projection, 240</p> <p>Banach's fixed point theorem, 47</p> <p>bimolecular reaction, 437</p> <p>Blow-up, 514</p> <p>Blow-up of explicit Euler method, 551</p> <p>Broyden method for a large non-linear system, 108</p> <p>Broydens Quasi-Newton method: convergence, 105</p> <p>Butcher scheme for some explicit RK-SSM, 504</p> <p>Cancellation in decimal floating point arithmetic, 90</p> <p>Chebyshev interpolation error, 296</p> <p>Chebyshev polynomials on arbitrary interval, 293</p> <p>Chebyshev representation of built-in functions, 305</p>	<p>Chebyshev vs uniform nodes, 295</p> <p>Chemical reaction kinetics, 564</p> <p>Choice of unitary/orthogonal transformation, 128</p> <p>Complexity of Householder QR-factorization, 129</p> <p>Computational effort for eigenvalue computations, 199</p> <p>Computing Gauss nodes and weights, 368</p> <p>condition</p> <p> extended system, 165</p> <p>Conditioning of normal equations, 162</p> <p>Conditioning of the least squares problem, 159</p> <p>Constitutive relations from measurements, 262</p> <p>Construction of simple Runge-Kutta methods, 501</p> <p>Convergence of equidistant trapezoidal rule, 391</p> <p>Convergence of Newton's method, 91</p> <p>Convergence of PINVIT, 226</p> <p>Convergence of simple Runge-Kutta methods, 501</p> <p>Convergence of simple splitting methods, 495</p> <p>Convergence of subspace variant of direct power method, 234</p> <p>Conversion into autonomous ODE, 442</p>	<p>Num. Meth. Phys.</p> <p>Gradinaru D-MATH</p> <p>9.4</p> <p>p. 607</p>
--	---	--

Damped Newton method, 100	Explicit integrator, 510
Details of Householder reflections, 124	explicit Runge-Kutta steps for Ricatti equation, 506
DFT	Explicit trapezoidal rule for decay equation, 555
Frequency analysis, 314	Exploring convergence, 278
Differentiation repetition, 84	Failure of damped Newton method, 101
Direct power method, 210	Feasibility of implicit Euler timestepping, 452
Divided differences and derivatives, 274	FFT algorithm by matrix factorization, 324
Domain of definition of solutions of IVPs, 444	FFT based on general factorization, 327
Efficiency of FFT, 317	FFT for prime vector length, 328
Efficient evaluation of trigonometric interpolation polynomials, 339	fit
Eigenvalue computation with Arnoldi process, 257	linear, 156
Energy conservation, 483	Fixed points in 1D, 49
Error estimates for polynomial quadrature, 366	Fractional order of convergence of secant method, 72
Error of Gauss quadrature, 371	Frequency analysis with DFT, 314
Error of polynomial interpolation, 285	Function representation, 265
Euler method for long-time evolution, 15, 468	Gain through adaptivity, 527
Euler method for pendulum equation, 14, 467	Gaussian elimination and LU-factorization, 113
Euler methods for stiff logistic IVP, 575	Generalized eigenvalue problems and Cholesky factorization, 194
Explicit Euler method as difference scheme, 450	
Explicit Euler method for damped oscillations, 562	

Generalized normalization, 230	trigonometric, 346
Group property of autonomous evolutions, 447	trigonometric analytic functions, 348
Growth with limited resources, 433	interpolation error, 276
Halley's iteration, 63	Keeping track of unitary transformations, 134
Heating generation in electrical circuits, 360	Krylov subspace methods for generalized EVP, 259
Hermite interpolation theorem, 284	Lanczos process for eigenvalue computation, 250
Horner scheme, 268	linear regression, 155
Impact of choice of norm, 26	Linearization of increment equations, 582
Impact of roundoff on Lanczos process, 251	Linearly convergent iteration, 30
Implicit Euler timestepping for decay equation, 574	Local convergence of Newton's method, 96
Implicit midpoint rule as difference method, 471	local convergence of secant method, 74
Implicit midpoint rule for circular motion, 17, 472	molecular dynamics, 486
Implicit midpoint rule for logistic equation, 471	Multidimensional fixed point iteration, 53
Implicit midpoint rule for pendulum, 17, 473	Necessary condition for L-stability, 580
Importance of numerical quadrature, 359	Necessity of iterative approximation, 21
Inexact splitting method, 497	Newton simplified method, 85
Initial guess for power iteration, 212	Newton method and minimization of quadratic functional, 176
Instability of normal equations, 163	
interpolation	

Newton method in 1D, 61	Principal component analysis for data analysis, 149
Newton method in 2D, 80	Pseudoinverse, 159
Newton method, modified , 69	Pseudoinverse and SVD, 171
Newton-Cotes formulas, 365	QR
Non-linear data fitting, 172	Orthogonalisierung, 133
Non-linear data fitting (II), 179	QR-Algorithm, 195
Normal equations vs. orthogonal transformations method, 171	QR-based solution of tridiagonal LSE, 139
Normal mode analysis, 184	Quadratic convergence], 33
Notation for single step methods, 456	quadratic inverse interpolation, 76
Numerical integration of logistic ODE, 511	Quadratur
One-step formulation of the Störmer-Verlet method, 480	Gauss-Legendre Ordnung 4, 367
Options for fixed point iterations, 44	Quadrature errors for composite quadrature rules, 381
Oregonator reaction, 511	Rationale for adaptive quadrature, 398
Oregonator-Reaction, 439	Rayleigh quotient iteration, 219
Overdetermined linear systems, 156	Refined local stepsize control, 533
Polynomials in Python, 268	Region of convergence of Newton method, 96
Predator-prey model, 435	regression
Principal component analysis, 139	linear (fit), 156
	Removing a singularity by transformation, 390

- Roundoff errors and difference quotients, 86
- Runge's example, 279, 286
- Runtime comparison for computation of coefficient of trigonometric interpolation polynomials, 336
- Runtimes of `eig`, 200
- secant method, 71
- Seeing linear convergence, 29
- Shifted inverse iteration, 215
- Simple adaptive stepsize control, 522
- Simple adaptive timestepping for fast decay, 554
- Simple composite polynomial quadrature rules, 379
- Simplified Newton method, 85
- Speed of convergence of explicit Euler method, 457
- Splitting methods for mechanical systems, 496
- Spring-pendulum, 484
- Störmel-Verlet method for pendulum, 477
- Störmer-Verlet method as difference method, 476
- Störmer-Verlet method as polygonal line method, 481
- Stability of Arnoldi process, 254
- Stepsize control, 539
- Stepsize control detects instability, 559
- Strongly attractive limit cycle, 568
- Subspace power methods, 235
- SVD and additive rank-1 decomposition, 144
- Taylor approximation, 264
- Termination criterion for contractive fixed point iteration, 55
- Termination criterion for direct power iteration, 212
- Transformation of quadrature rules, 362
- trigonometric interpolation, 346
 - analytic functions, 348
- Uniqueness of SVD, 145
- Unitary similarity transformation to tridiagonal form 198
- Visualization of explicit Euler method, 449