# Numerical Methods

## for Physics

(401-1662-10 L)

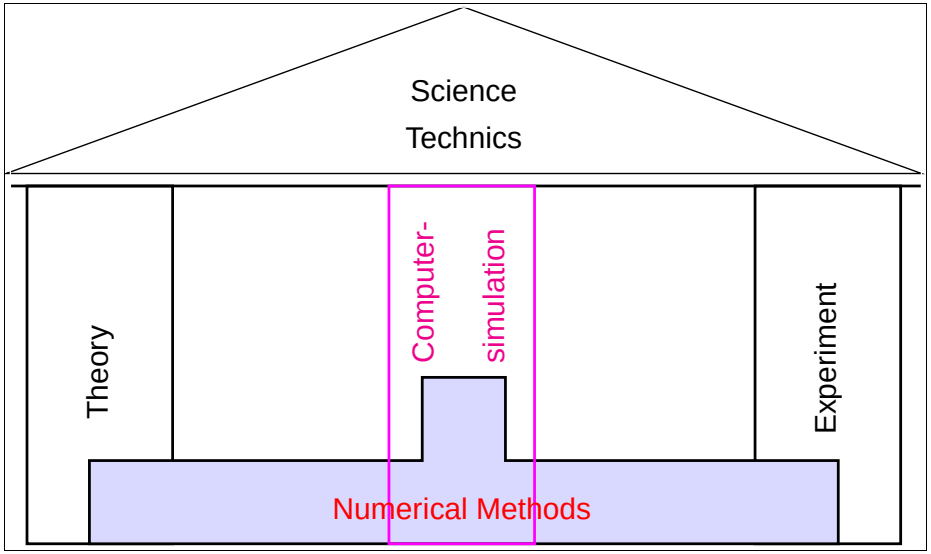
Dr. Vasile Gradinaru and Prof. Ralf Hiptmair





# Contents

## 0.1 About this course






### Focus

▷ 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.

## Reading instructions

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 examnination 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, *Einfuhrung 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.

## 0.2 Appetizer

*Example* 0.2.1 (Euler method for pendulum equation).

Hamiltonian form of equations of motion for pendulum

angular velocity $p := \dot{\alpha} \quad \Rightarrow \quad \frac{d}{dt}\begin{pmatrix}\alpha\\ p\end{pmatrix} = \begin{pmatrix} p \\ -\frac{g}{l}\sin\alpha\end{pmatrix}, \quad g = 9.8, l = 1 \,.$ (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: $\alpha(0) = \pi/4$, $p(0) = 0$.

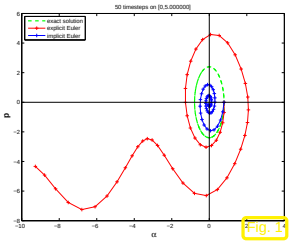


Fig. 2

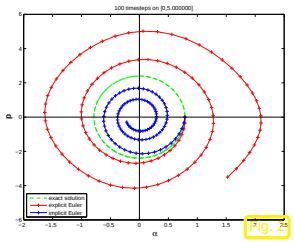


Fig. 3

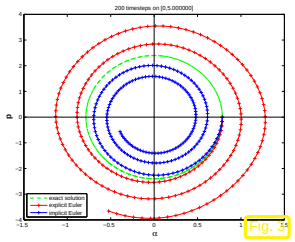


Fig. 5



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

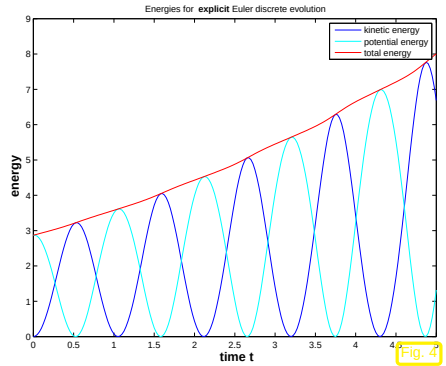


Fig. 4

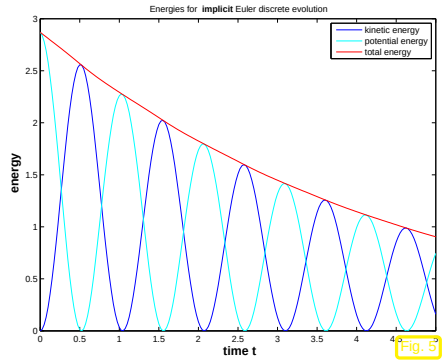


Fig. 5

☞ 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 , $D = \mathbb{R}^2$:

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

▶ Note that $I(\mathbf{y}) = \|\mathbf{y}\|$ is constant.
(movement with constant velocity on the circle)

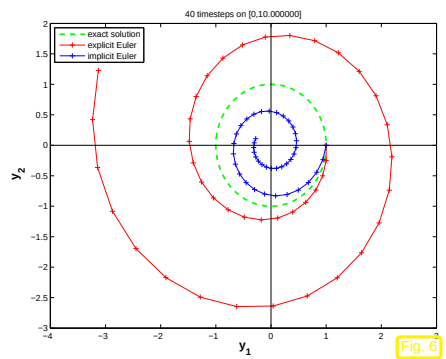


Fig. 6

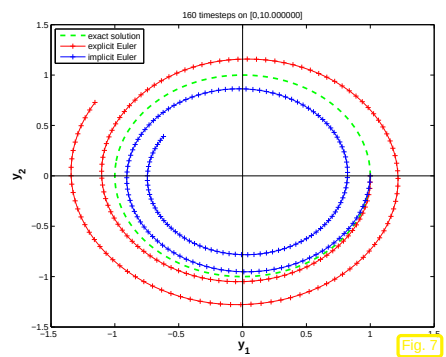


Fig. 7

☞ explicit Euler: numerical solution flyes 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).

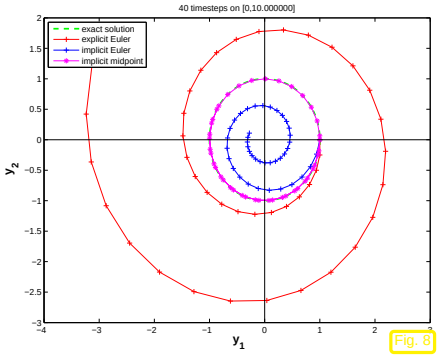


Fig. 8

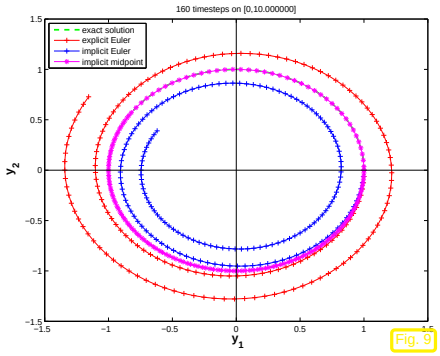


Fig. 9

☞ 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

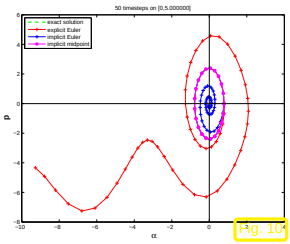


Fig. 10

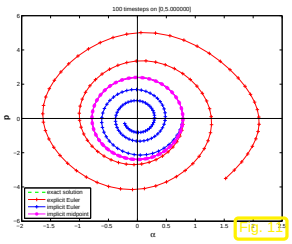


Fig. 11

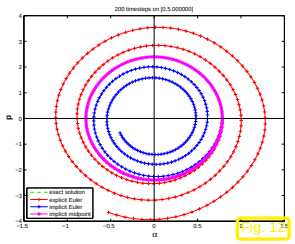


Fig. 12

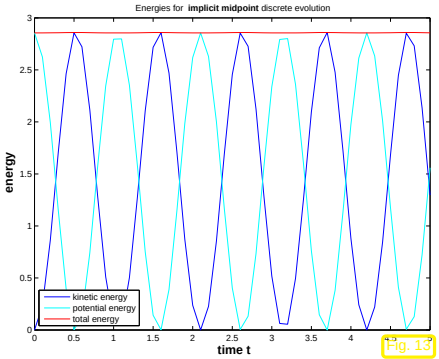


Fig. 13

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

$\Updownarrow$

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

Sought: solution of non-linear equation $F(\mathbf{x}) = 0$

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

In general no existence & uniqueness of solutions

## 1.1 Iterative methods

*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.

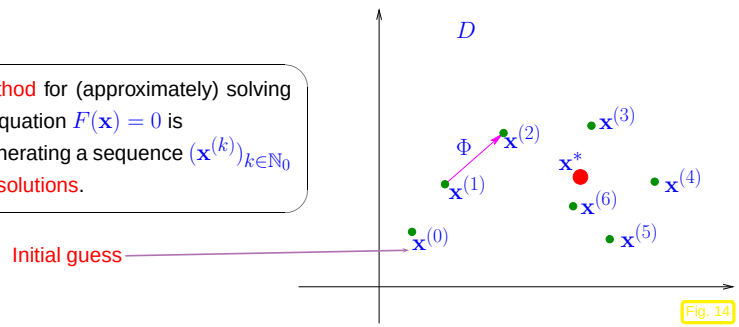



Fig. 14

Fundamental concepts: convergence ➡ speed of convergence

consistency

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

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



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

**Definition 1.1.1** (Convergence of iterative methods)**.**
*An iterative method converges (for fixed initial guess(es))* $:\Leftrightarrow$ $\mathbf{x}^{(k)} \to \mathbf{x}^{*}$ *and* $F(\mathbf{x}^{*}) = 0$.

**Definition 1.1.2** (Consistency of iterative methods)**.**
*An iterative method is consistent with* $F(\mathbf{x}) = 0$

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

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*

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

*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.)

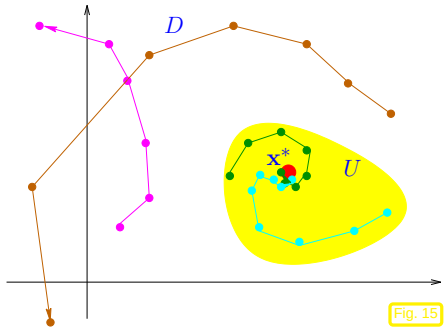



Fig. 15

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 indiuced 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* $\mathbf{x}^{(k)}$, $k = 0, 1, 2, \ldots$, *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).

| | | |
|---|---|---|
| *Fact of convergence* of iteration is | independent | of choice of norm |
| *Fact of linear convergence* | depends | on choice of norm |
| *Rate* of linear convergence | depends | 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)* $\forall \mathbf{x} \in X: \quad \mathbf{x} \neq 0 \quad \Leftrightarrow \quad \|\mathbf{x}\| > 0$ *(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 $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T \in \mathbb{K}^n$)

| name | : | definition | `numpy.linalg` function |
|---|---|---|---|
| Euclidean norm | : | $\|\mathbf{x}\|_2 := \sqrt{\lvert x_1\rvert^2 + \cdots + \lvert x_n\rvert^2}$ | `norm(x)` |
| 1-norm | : | $\|\mathbf{x}\|_1 := \lvert x_1\rvert + \cdots + \lvert x_n\rvert$ | `norm(x,1)` |
| $\infty$-norm, max norm | : | $\|\mathbf{x}\|_\infty := \max\{\lvert x_1\rvert, \ldots, \lvert x_n\rvert\}$ | `norm(x,inf)` |



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*

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





*Theorem* 1.1.11 (Equivalence of all norms on finite dimensional vector spaces).
*If* $\dim V < \infty$ *all norms (→ Def. 1.1.9) on* $V$ *are equivalent (→ Def. 1.1.10).*

△

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

$$\|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1 \leq \sqrt{n} \|\mathbf{x}\|_2 \ , \tag{1.1.7}$$
$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \sqrt{n} \|\mathbf{x}\|_\infty \ , \tag{1.1.8}$$
$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_1 \leq n \|\mathbf{x}\|_\infty \ . \tag{1.1.9}$$



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

notation: $\|\mathbf{x}\|_2 \to \|\mathbf{M}\|_2$, $\|\mathbf{x}\|_1 \to \|\mathbf{M}\|_1$, $\|\mathbf{x}\|_\infty \to \|\mathbf{M}\|_\infty$

*Example* 1.1.4 (Matrix norm associated with $\infty$-norm and 1-norm).

e.g. for $\mathbf{M} \in \mathbb{K}^{2,2}$:

$$\begin{aligned} \|\mathbf{Mx}\|_\infty &= \max\{|m_{11}x_1 + m_{12}x_2|, |m_{21}x_1 + m_{22}x_2|\} \\ &\leq \max\{|m_{11}| + |m_{12}|, |m_{21}| + |m_{22}|\} \; \|x\|_\infty \;, \\ \|\mathbf{Mx}\|_1 &= |m_{11}x_1 + m_{12}x_2| + |m_{21}x_1 + m_{22}x_2| \\ &\leq \max\{|m_{11}| + |m_{21}|, |m_{12}| + |m_{22}|\}(|x_1| + |x_2|) \;. \end{aligned}$$

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

- matrix norm $\leftrightarrow \|\cdot\|_\infty$ = row sum norm $\|\mathbf{M}\|_\infty := \max\limits_{i=1,\dots,m} \sum\limits_{j=1}^{n} |m_{ij}|$ , (1.1.10)
- matrix norm $\leftrightarrow \|\cdot\|_1$ = column sum norm $\|\mathbf{M}\|_1 := \max\limits_{j=1,\dots,n} \sum\limits_{i=1}^{m} |m_{ij}|$ . (1.1.11)



*Remark* 1.1.5 (Seeing linear convergence).

norms of iteration errors

$\updownarrow$

$\sim$ straight line in lin-log plot

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

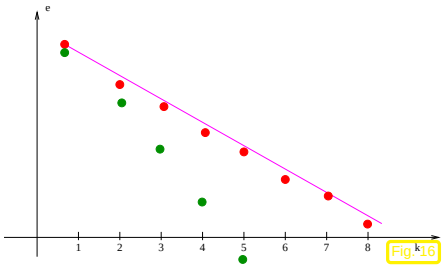


Fig. 16

(●: 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 $0 < L < 1$)

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

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

```
def lincvg(x):
    y = []
    for k in xrange(15):
        x = x +(cos(x)+1)/sin(x)
        y += [x]
    err = array(y) - x
    rate = err[1:]/err[:-1]
    return err, rate
```

Note: $x^{(15)}$ replaces the exact solution $x^*$ in the computation of the rate of convergence.

| $k$ | $x^{(0)} = 0.4$ | | $x^{(0)} = 0.6$ | | $x^{(0)} = 1$ | |
|---|---|---|---|---|---|---|
| | $x^{(k)}$ | $\frac{\lvert x^{(k)}-x^{(15)}\rvert}{\lvert x^{(k-1)}-x^{(15)}\rvert}$ | $x^{(k)}$ | $\frac{\lvert x^{(k)}-x^{(15)}\rvert}{\lvert x^{(k-1)}-x^{(15)}\rvert}$ | $x^{(k)}$ | $\frac{\lvert x^{(k)}-x^{(15)}\rvert}{\lvert x^{(k-1)}-x^{(15)}\rvert}$ |
| 2 | 3.3887 | 0.1128 | 3.4727 | 0.4791 | 2.9873 | 0.4959 |
| 3 | 3.2645 | 0.4974 | 3.3056 | 0.4953 | 3.0646 | 0.4989 |
| 4 | 3.2030 | 0.4992 | 3.2234 | 0.4988 | 3.1031 | 0.4996 |
| 5 | 3.1723 | 0.4996 | 3.1825 | 0.4995 | 3.1224 | 0.4997 |
| 6 | 3.1569 | 0.4995 | 3.1620 | 0.4994 | 3.1320 | 0.4995 |
| 7 | 3.1493 | 0.4990 | 3.1518 | 0.4990 | 3.1368 | 0.4990 |
| 8 | 3.1454 | 0.4980 | 3.1467 | 0.4980 | 3.1392 | 0.4980 |

Rate of convergence $\approx 0.5$

Linear convergence as in Def. 1.1.4

$\Updownarrow$

error graphs = straight lines in lin-log scale

$\longrightarrow$ Rem. 1.1.5

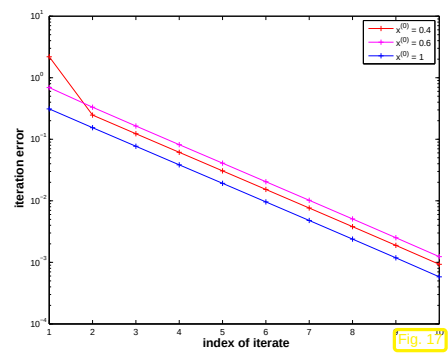


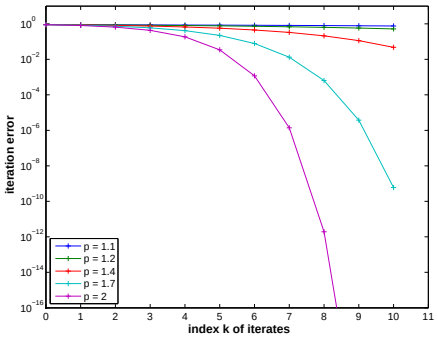
Fig. 17

◇

**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 $\longrightarrow$ 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):

$$\epsilon_{k+1} \approx C\epsilon_k^p \quad \Rightarrow \quad \log \epsilon_{k+1} = \log C + p \log \epsilon_k \quad \Rightarrow \quad \log \epsilon_{k+1} = \log C \sum_{l=0}^{k} p^l + p^{k+1} \log \epsilon_0$$

$$\Rightarrow \quad \log \epsilon_{k+1} = -\frac{\log C}{p-1} + \left(\frac{\log C}{p-1} + \log \epsilon_0\right) p^{k+1} .$$

In this case, the error graph is a concave power curve (for sufficiently small $\epsilon_0$ !)

*Example* 1.1.8 (quadratic convergence). (= convergence of order 2)
Iteration for computing $\sqrt{a}$, $a > 0$:

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

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 $k \geq 2$ ➢ $(x^{(k)})_{k\in\mathbb{N}_0}$ converges as a decreasing sequence that is bounded from below (→ analysis course)

How to guess the order of convergence in a numerical experiment?
Abbreviate $\epsilon_k := \left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\|$ and then

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

Numerical experiment: iterates for $a = 2$:



| $k$ | $x^{(k)}$ | $e^{(k)} := x^{(k)} - \sqrt{2}$ | $\log \frac{\lvert e^{(k)}\rvert}{\lvert e^{(k-1)}\rvert} : \log \frac{\lvert e^{(k-1)}\rvert}{\lvert e^{(k-2)}\rvert}$ |
|---|---|---|---|
| 0 | 2.00000000000000000 | 0.58578643762690485 | |
| 1 | 1.50000000000000000 | 0.08578643762690485 | |
| 2 | 1.41666666666666652 | 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 $\delta_k$, we have:

$$x^{(k)} = x^*(1+\delta_k) \quad \Rightarrow \quad x^{(k)} - x^* = \delta_k x^* .$$
$$\Rightarrow |x^*\delta_{k+1}| = |x^{(k+1)} - x^*| \leq C|x^{(k)} - x^*|^2 = C|x^*\delta_k|^2$$
$$\Rightarrow \quad |\delta_{k+1}| \leq C|x^*|\delta_k^2 . \qquad (1.1.14)$$

Note: $\delta_k \approx 10^{-\ell}$ means that $\mathbf{x}^{(k)}$ has $\ell$ 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: ▷ stop iteration when iteration error is just "small enough"

"small enough" depends on concrete setting:

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

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

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

$\mathbf{x}^*$ 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.

③ use that $\mathbb{M}$ is finite! (→ Sect. **??**)

➢ possible to wait until (convergent) iteration becomes stationary

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

Code 1.1.9: stationary iteration

```
from numpy import sqrt, array
def sqrtit(a,x):
    exact = sqrt(a)
    e = [x]
    x_old = -1.
    while x_old != x:
        x_old = x
        x = 0.5*(x+a/x)
        e += [x]
    e = array(e)
    e = abs(e-exact)
    return e

e = sqrtit(2.,1.)
print e
```

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

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

no guaranteed accuracy





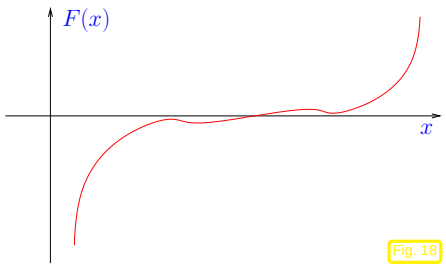



Fig. 18

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

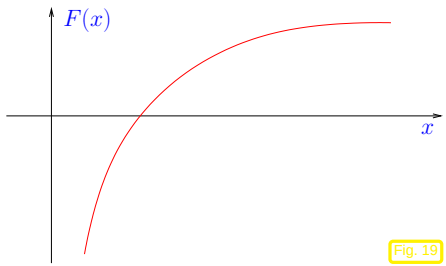



Fig. 19

$\left\|F(\mathbf{x}^{(k)})\right\|$ small $\Rightarrow$ $|x-x^*|$ 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 $0 < L < 1$:

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

Iterates satisfy: $$\boxed{\left\|\mathbf{x}^{(k+1)}-\mathbf{x}^*\right\| \leq \frac{L}{1-L}\left\|\mathbf{x}^{(k+1)}-\mathbf{x}^{(k)}\right\|}\,. \tag{1.1.17}$$

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:

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

Observed rate of convergence: $L = 1/2$

Error and error bound for $x^{(0)} = 0.4$:

| $k$ | $\lvert x^{(k)} - \pi \rvert$ | $\frac{L}{1-L}\lvert x^{(k)} - x^{(k-1)} \rvert$ | slack of bound |
|---|---|---|---|
| 1 | 2.191562221997101 | 4.933154875586894 | 2.741592653589793 |
| 2 | 0.247139097781070 | 1.944423124216031 | 1.697284026434961 |
| 3 | 0.122936737876834 | 0.124202359904236 | 0.001265622027401 |
| 4 | 0.061390835206217 | 0.061545902670618 | 0.000155067464401 |
| 5 | 0.030685773472263 | 0.030705061733954 | 0.000019288261691 |
| 6 | 0.015341682696235 | 0.015344090776028 | 0.000002408079792 |
| 7 | 0.007670690889185 | 0.007670991807050 | 0.000000300917864 |
| 8 | 0.003835326638666 | 0.003835364250520 | 0.000000037611854 |
| 9 | 0.001917660968637 | 0.001917665670029 | 0.000000004701392 |
| 10 | 0.000958830190489 | 0.000958830778147 | 0.000000000587658 |
| 11 | 0.000479415058549 | 0.000479415131941 | 0.000000000073392 |
| 12 | 0.000239707524646 | 0.000239707533903 | 0.000000000009257 |
| 13 | 0.000119853761949 | 0.000119853762696 | 0.000000000000747 |
| 14 | 0.000059926881308 | 0.000059926880641 | 0.000000000000667 |
| 15 | 0.000029963440745 | 0.000029963440563 | 0.000000000000181 |

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

Note: If $L$ not known then using $\widetilde{L} > L$ 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 $\mathbf{x}^{(0)} \in U$

$$\rightarrow \underbrace{\boxed{\text{iterates } (\mathbf{x}^{(k)})_{k\in\mathbb{N}_0}: \quad \mathbf{x}^{(k+1)} := \Phi(\mathbf{x}^{(k)})}}_{\rightarrow \text{ 1-point method, } cf. \text{ (1.1.1)}} .$$

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*

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

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

General construction of fixed point iterations that is consistent with $F(\mathbf{x}) = 0$:
rewrite $F(\mathbf{x}) = 0 \quad \Leftrightarrow \quad \Phi(\mathbf{x}) = \mathbf{x}$ and then

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

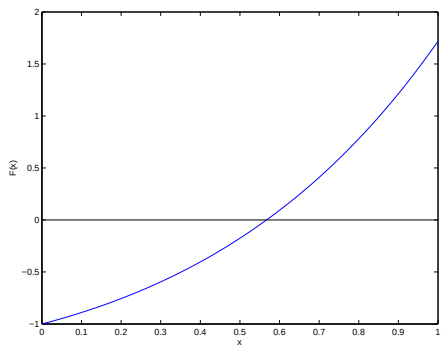
Note: there are *many ways* to transform $F(\mathbf{x}) = 0$ into a fixed point form **!**

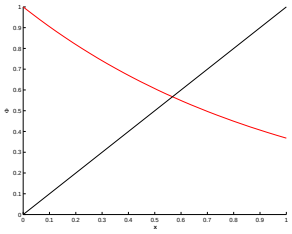
*Example* 1.2.1 (Options for fixed point iterations).

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

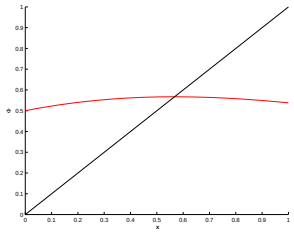
Different fixed point forms:

$$\begin{aligned} \Phi_1(x) &= e^{-x} , \\ \Phi_2(x) &= \frac{1+x}{1+e^x} , \\ \Phi_3(x) &= x + 1 - xe^x . \end{aligned}$$

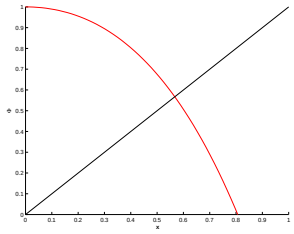




function $\Phi_1$



function $\Phi_2$



function $\Phi_3$

| $k$ | $x^{(k+1)} := \Phi_1(x^{(k)})$ | $x^{(k+1)} := \Phi_2(x^{(k)})$ | $x^{(k+1)} := \Phi_3(x^{(k)})$ |
|---|---|---|---|
| 0 | 0.500000000000000 | 0.500000000000000 | 0.500000000000000 |
| 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 |

| $k$ | $\lvert x_1^{(k+1)} - x^*\rvert$ | $\lvert x_2^{(k+1)} - x^*\rvert$ | $\lvert x_3^{(k+1)} - x^*\rvert$ |
|---|---|---|---|
| 0 | 0.067143290409784 | 0.067143290409784 | 0.067143290409784 |
| 1 | 0.039387369302849 | 0.000832287212566 | 0.108496074240152 |
| 2 | 0.021904078517179 | 0.000000125374922 | 0.219330611898582 |
| 3 | 0.012559804468284 | 0.000000000000003 | 0.288178118764323 |
| 4 | 0.007078662470882 | 0.000000000000000 | 0.723649245792953 |
| 5 | 0.004028858567431 | 0.000000000000000 | 0.410183132337935 |
| 6 | 0.002280343429460 | 0.000000000000000 | 1.186907542305364 |
| 7 | 0.001294757160282 | 0.000000000000000 | 0.146569797006362 |
| 8 | 0.000733837662863 | 0.000000000000000 | 0.310516641279937 |
| 9 | 0.000416343852458 | 0.000000000000000 | 0.357777386500765 |
| 10 | 0.000236077474313 | 0.000000000000000 | 0.974565695952037 |

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

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*

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



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

$$\left\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\right\| = \left\|\Phi(\mathbf{x}^{(k)}) - \Phi(\mathbf{x}^*)\right\| \overset{(1.2.3)}{\le} L\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\|\ ,$$

that is, the iteration converges (at least) linearly (→ Def. 1.1.4).

*Remark* 1.2.2 (Banach's fixed point theorem). → [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*

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

*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* $\mathbf{x}^{(k+1)} := \Phi(x^{(k)})$ *for any* $\mathbf{x}^{(0)} \in D$.

*Proof.* Proof based on 1-point iteration $\mathbf{x}^{(k)} = \Phi(\mathbf{x}^{(k-1)})$, $\mathbf{x}^{(0)} \in D$:



$$\left\|\mathbf{x}^{(k+N)} - \mathbf{x}^{(k)}\right\| \le \sum_{j=k}^{k+N-1} \left\|\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)}\right\| \le \sum_{j=k}^{k+N-1} L^j \left\|\mathbf{x}^{(1)} - \mathbf{x}^{(0)}\right\|$$
$$\le \frac{L^k}{1-L}\left\|\mathbf{x}^{(1)} - \mathbf{x}^{(0)}\right\| \xrightarrow{k\to\infty} 0\ .$$

$(\mathbf{x}^{(k)})_{k\in\mathbb{N}_0}$ Cauchy sequence ➤ convergent $\mathbf{x}^{(k)} \xrightarrow{k\to\infty} \mathbf{x}^*$ .
Continuity of $\Phi$ ➤ $\Phi(\mathbf{x}^*) = \mathbf{x}^*$. Uniqueness of fixed point is evident. □

△



A simple criterion for a differentiable $\Phi$ to be contractive:

**Lemma 1.2.4** (Sufficient condition for local linear convergence of fixed point iteration)**.**
*If $\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$, $\Phi(\mathbf{x}^*) = \mathbf{x}^*$, $\Phi$ 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 !*

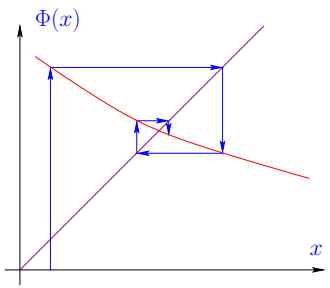
✎ notation: $D\Phi(\mathbf{x}) \hat{=}$ Jacobian (*ger.:* Jacobi-Matrix) of $\Phi$ at $\mathbf{x} \in D$

*Example* 1.2.3 (Fixed point iteration in 1D).

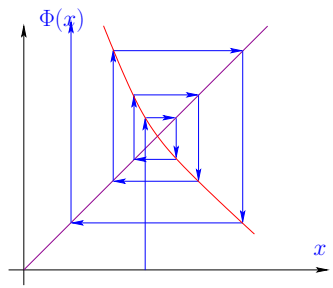
1D setting ($n = 1$): $\Phi : \mathbb{R} \mapsto \mathbb{R}$ continuously differentiable, $\Phi(x^*) = x^*$

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

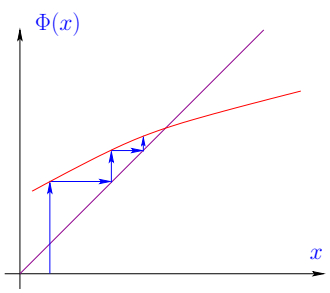
"Visualization" of the statement of Lemma 1.2.4: The iteration converges *locally*, if $\Phi$ is flat in a neighborhood of $x^*$, it will diverge, if $\Phi$ is steep there.



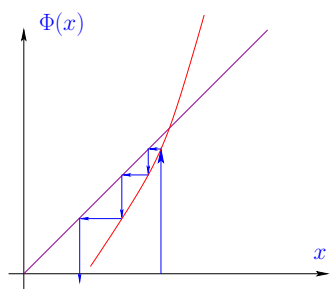

$-1 < \Phi'(x^*) \leq 0$ ➢ convergence




$\Phi'(x^*) < -1$ ➢ divergence




$0 \leq \Phi'(x^*) < 1$ ➢ convergence




$1 < \Phi'(x^*)$ ➢ 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 $\lim_{t \to 0} \psi(t) = 0$.

Choose $\delta > 0$ such that

$$L := \psi(t) + \|D\Phi(\mathbf{x}^*)\| \leq \tfrac{1}{2}(1 + \|D\Phi(\mathbf{x}^*)\|) < 1 \quad \forall 0 \leq t < \delta \ .$$

By inverse triangle inequality we obtain for fixed point iteration

$$\|\Phi(\mathbf{x}) - \mathbf{x}^*\| - \|D\Phi(\mathbf{x}^*)(\mathbf{x} - \mathbf{x}^*)\| \leq \psi(\|\mathbf{x} - \mathbf{x}^*\|)\,\|\mathbf{x} - \mathbf{x}^*\|$$

$$\text{➤} \quad \left\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\right\| \leq (\psi(t) + \|D\Phi(\mathbf{x}^*)\|)\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| \leq L\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| \ ,$$

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 $L := \sup_{\mathbf{x} \in U} \|D\Phi(\mathbf{x})\| < 1$. 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

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

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

There is $\delta > 0$: $B := \{\mathbf{x}\colon \|\mathbf{x} - \mathbf{x}^*\| \leq \delta\} \subset \mathrm{dom}(\Phi)$. $\Phi$ 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 $L = \|D\Phi(x^*)\|$

△

*Example* 1.2.5 (Multidimensional fixed point iteration).

System of equations in fixed point form:

$$\begin{cases} x_1 - c(\cos x_1 - \sin x_2) = 0 \\ (x_1 - x_2) - c\sin x_2 = 0 \end{cases} \Rightarrow \begin{cases} c(\cos x_1 - \sin x_2) = x_1 \\ c(\cos x_1 - 2\sin x_2) = x_2 \end{cases} .$$

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: $\|\cdot\|$ = $\infty$-norm $\|\cdot\|_\infty$ ($\rightarrow$ Example 1.1.4) ;

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

$\succ$ (at least) linear convergence of the fixed point iteration.

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

$\diamond$

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

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

**Theorem 1.2.6** (Taylor's formula). $\rightarrow$ *[52, Sect. 5.5]*
*If $\Phi : U \subset \mathbb{R} \mapsto \mathbb{R}$, $U$ interval, is $m+1$ times continuously differentiable, $x \in U$*

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

Apply Taylor expansion (1.2.3) to iteration function $\Phi$:

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

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

**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 $x^*$

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

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

Then appeal to (1.2.4) $\square$

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 $x^* e^{x^*} - 1 = 0$

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

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

*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 $0 \leq L < 1$:

$$\left\|\mathbf{x}^{(k+m)} - \mathbf{x}^{(k)}\right\| \overset{\triangle\text{-ineq.}}{\leq} \sum_{j=k}^{k+m-1} \left\|\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)}\right\| \leq \sum_{j=k}^{k+m-1} L^{j-k} \left\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\right\|$$

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

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

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

| Set $l = 0$ in (1.2.5) | Set $l = k-1$ in (1.2.5) |
|---|---|
| a priori termination criterion | a posteriori termination criterion |
| $\left\|\mathbf{x}^* - \mathbf{x}^{(k)}\right\| \leq \frac{L^k}{1-L}\left\|\mathbf{x}^{(1)} - \mathbf{x}^{(0)}\right\|$ (1.2.6) | $\left\|\mathbf{x}^* - \mathbf{x}^{(k)}\right\| \leq \frac{L}{1-L}\left\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right\|$ (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: $x^* \in I$: $F(x^*) = 0$

### 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 !)

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

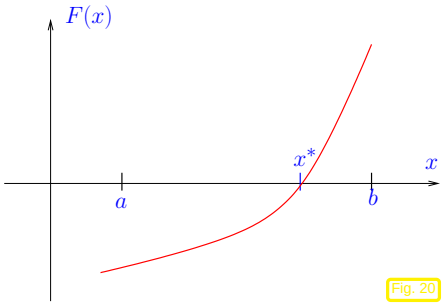


Fig. 20

*Algorithm* 1.3.1 (Bisection method).

Code 1.3.2: Bisection method

```
def mybisect(f,a,b,tol=1e-12):
    if a>b:
        t = b; a = b; b = t
    fa = f(a); fb = f(b)
    if fa*fb > 0: raise ValueError
    v = 1
    if fa>0: v = -1
    x = 0.5*(a+b)
    k = 1
    while (b-a>tol) and (a<x) and (x<b):
        if v*f(x)>0: b = x
        else: a = x
        x = 0.5*(a+b)
        k += 1
    return x, k

if __name__=='__main__':
    f = lambda x: x**3 - 2*x**2 + 4.*x/3. - 8./27.
    x, k = mybisect(f,0,1)
    print 'x_bisect = ', x, ' after k=', k, 'iterations'
    from scipy.optimize import fsolve, bisection
    x = fsolve(f,0,full_output=True)
    print x
    print bisection(f,0,1)
```

Advantages:
- "foolproof"
- requires only $F$ evaluations

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

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

### 1.3.2 Model function methods

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

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

❶ replace $F$ with model function $\widetilde{F}$
(using function values/derivative values in $x^{(k)}, x^{(k-1)}, \ldots, x^{(k-m)}$)

❷ $x^{(k+1)} :=$ zero of $\widetilde{F}$
(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 $\longrightarrow$ Sect. 1.2)

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

### 1.3.2.1 Newton method in scalar case

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

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

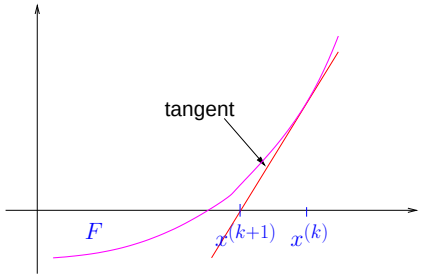
$$\widetilde{F}(x) := F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)})$$

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

We obtain Newton iteration

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

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




*Example* 1.3.3 (Newton method in 1D). (→ Ex. 1.2.1)

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

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

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

1.2.1 1.1.13

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

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

From Lemma 1.2.7:

Newton method locally quadratically convergent (→ 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

$$h(x) := \frac{a}{x+b} + c \quad \text{(rational function) such that } F^{(j)}(x^{(k)}) = h^{(j)}(x^{(k)}) \,, \quad j = 0, 1, 2 \,.$$

$$\frac{a}{x^{(k)} + b} + c = F(x^{(k)}) \,, \quad -\frac{a}{(x^{(k)} + b)^2} = F'(x^{(k)}) \,, \quad \frac{2a}{(x^{(k)} + b)^3} = F''(x^{(k)}) \,.$$

$$x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})} \cdot \frac{1}{1 - \frac{1}{2}\frac{F(x^{(k)})F''(x^{(k)})}{F'(x^{(k)})^2}} \,.$$

Halley's iteration for $F(x) = \dfrac{1}{(x+1)^2} + \dfrac{1}{(x+0.1)^2} - 1 \,, \quad x > 0 :$ 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 (→ Sect. **??**) ◇

In the previous example Newton's method performed rather poorly. Often its convergence can be boosted by converting the non-linear equation to an equivalent one (that is, one with the same solutions) for another function $g$, which is "closer to a linear function":

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

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

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

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

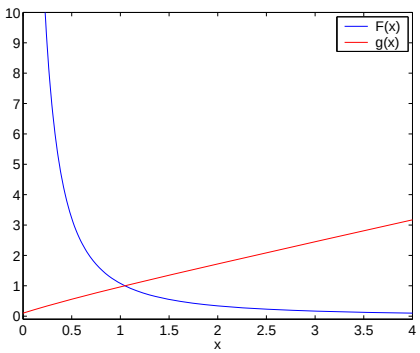
*Example* 1.3.5 (Adapted Newton method).

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

Observation:

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

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




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

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

$$= x^{(k)} + \frac{2(F(x^{(k)})+1)(1 - \sqrt{F(x^{(k)})+1})}{F'(x^{(k)})} \ .$$

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:

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

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

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

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

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





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

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

Lemma 1.2.7 ➢ **Necessary** conditions for local convergencd of order $p$:

$p = 2$ (quadratical convergence): $H(x^*) = 1$ ,

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

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

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

**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).

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

Numerical experiment:

$$F(x) = xe^x - 1\,,\quad x^{(0)} = 5$$

| $k$ | $e^{(k)} := x^{(k)} - x^*$ | | |
|---|---|---|---|
| | 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 | |

◇



### 1.3.2.3 Multi-point methods

Idea: Replace $F$ with interpolating polynomial

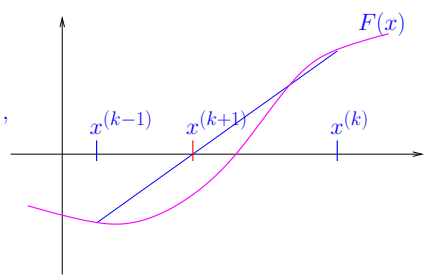
producing interpolatory model function methods

Simplest representative: secant method

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

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

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






secant method
(python implementation)

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

Code 1.3.7: secant method

```
def secant(f,x0,x1,maxit=50,tol=1e-12):
    fo = f(x0)
    for k in xrange(maxit):
        fn = f(x1)
        print 'x1=',x1, 'f(x1)=', fn
        s = fn*(x1-x0)/(fn-fo)
        x0 = x1; x1 -= s
        if abs(s)<tol:
            x = x1
            return x, k
        fo = fn
    x = NaN
    return x, maxit
```

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). $F(x) = xe^x - 1\,,\quad x^{(0)} = 0\,,\, x^{(1)} = 5\,.$

| $k$ | $x^{(k)}$ | $F(x^{(k)})$ | $e^{(k)} := x^{(k)} - x^*$ | $\frac{\log\lvert e^{(k+1)}\rvert - \log\lvert e^{(k)}\rvert}{\log\lvert e^{(k)}\rvert - \log\lvert e^{(k-1)}\rvert}$ |
|---|---|---|---|---|
| 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 | -0.00000000000001 | -0.00000000000000 | |

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 $e^{(k)} := x^{(k)} - x^*$

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

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:

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

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

$$\Rightarrow \quad e^{(k+1)} = K^{p+1}(e^{(k-1)})^{p^2}$$

$$\Rightarrow \quad (e^{(k-1)})^{p^2-p-1} = K^{-p}C \quad \Rightarrow \quad p^2 - p - 1 = 0 \quad \Rightarrow \quad p = \tfrac{1}{2}(1 \pm \sqrt{5}) \,.$$

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





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

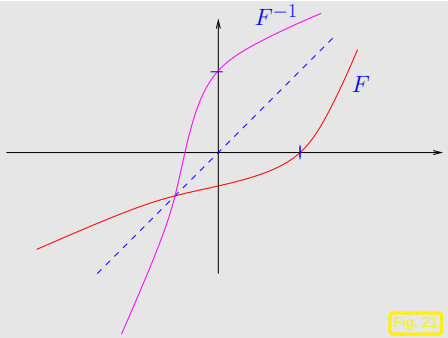
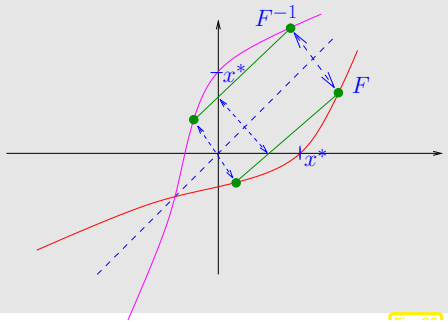



Fig. 21

Case $m = 1$ ➢ secant method



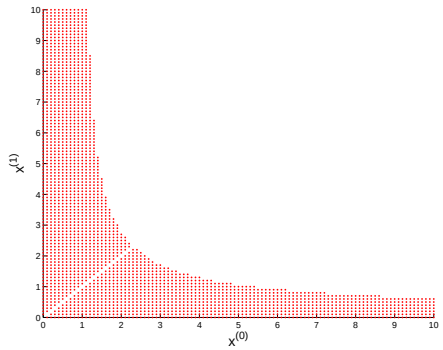



*Example* 1.3.10 (local convergence of secant method).

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

· ≙ secant method converges for a pair $(x^{(0)}, x^{(1)})$ of initial guesses. ▷

= local convergence → Def. 1.1.3 ◇






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);
```

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

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

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

| $k$ | $x^{(k)}$ | $F(x^{(k)})$ | $e^{(k)} := x^{(k)} - x^*$ | $\frac{\log\lvert e^{(k+1)}\rvert - \log\lvert e^{(k)}\rvert}{\log\lvert e^{(k)}\rvert - \log\lvert e^{(k-1)}\rvert}$ |
|---|---|---|---|---|
| 3 | 0.08520390058175 | -0.90721814294134 | -0.48193938982803 | |
| 4 | 0.16009252622586 | -0.81211229637354 | -0.40705076418392 | 3.33791154378839 |
| 5 | 0.79879381816390 | 0.77560534067946 | 0.23165052775411 | 2.28740488912208 |
| 6 | 0.63094636752843 | 0.18579323999999 | 0.06380307711864 | 1.82494667289715 |
| 7 | 0.56107750991028 | -0.01667806436181 | -0.00606578049951 | 1.87323264214217 |
| 8 | 0.56706941033107 | -0.00020413476766 | -0.00007388007872 | 1.79832936980454 |
| 9 | 0.56714331707092 | 0.00000007367067 | 0.00000002666114 | 1.84841261527097 |
| 10 | 0.56714329040980 | 0.00000000000003 | 0.00000000000001 | |





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

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

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

- 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 $x^{(k+1)} := p(0)$

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



### 1.4.1 The Newton iteration

Idea (→ Sect. 1.3.2.1): local linearization:

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

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

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

▶ Newton iteration: (↔ (1.3.1) for $n = 1$)

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

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



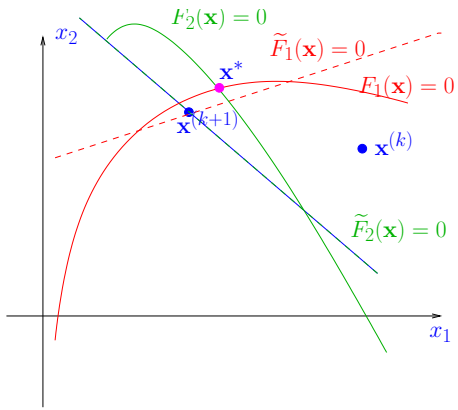


Illustration of idea of Newton's method for $n = 2$:

▷

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








template for Newton method:

Solve linear system:

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

`F`, `DF`: functions

A posteriori termination criterion

MATLAB-CODE: Newtonverfahren

```
def newton(x,F,DF,tol,maxit):
  for i in xrange(maxit):
    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 | $(0.87850000000000,\ 1.064285714285714)^T$ | 1.37e-01 |
| 2 | $(1.01815943274188,\ 1.00914882463936)^T$ | 2.03e-02 |
| 3 | $(1.00023355916300,\ 1.00015913936075)^T$ | 2.83e-04 |
| 4 | $(1.00000000583852,\ 1.00000002726552)^T$ | 2.79e-08 |
| 5 | $(0.999999999999998,\ 1.000000000000000)^T$ | 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 → [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:

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

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). → 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*

linear mapping $DF(\mathbf{x}) : V \mapsto W$ ,

such that $\|F(\mathbf{x}+\mathbf{h}) - F(\mathbf{x}) - DF(\mathbf{h})\mathbf{h}\| = o(\|\mathbf{h}\|) \quad \forall \mathbf{h}\ ,\ \|\mathbf{h}\| < \delta$ .

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

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

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

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

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

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

△



*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: $f'(x) \approx \dfrac{f(x+h) - f(x)}{h}$ .

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



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, $f(x) = e^x$, $x = 0$ ▷

| $\log_{10}(h)$ | 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 |



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 $h \to 0$ (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

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

| $\log_{10}(h)$ | 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 $f(x) = \exp(x)$:

$$\begin{aligned} \texttt{df} &= \frac{e^{x+h}\boxed{(1+\delta_1)} - e^x\boxed{(1+\delta_2)}}{h} \\ &= e^x\left(\frac{e^h - 1}{h} + \frac{\delta_1 e^h - \delta_2}{h}\right) \end{aligned}$$

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

$$|\delta_1|, |\delta_2| \le \texttt{eps} \ .$$

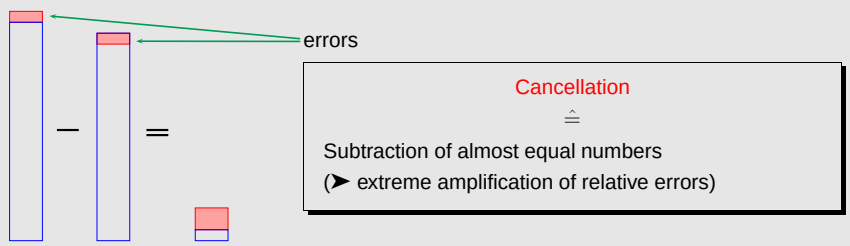
$1 + O(h) \qquad O(h^{-1})$ für $h \to 0$

➤ relative error: $\left|\frac{e^x - \mathtt{df}}{e^x}\right| \approx h + \frac{2\mathtt{eps}}{h} \to \min$ for $h = \sqrt{2\,\mathtt{eps}}$ .

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

◇

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






*Example* 1.4.8 (cancellation in decimal floating point arithmetic).

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

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

padded zeroes

◇

## 1.4.2 Convergence of Newton's method

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

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

$$\Big[\text{“product rule”}: \quad D\Phi(\mathbf{x}) = \mathbf{I} - D(DF(\mathbf{x})^{-1})F(\mathbf{x}) - DF(\mathbf{x})^{-1}DF(\mathbf{x}) \quad \Big]$$



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

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:

| $k$ | $\mathbf{x}^{(k)}$ | $\epsilon_k := \lVert\mathbf{x}^* - \mathbf{x}^{(k)}\rVert_2$ | $\frac{\log\epsilon_{k+1} - \log\epsilon_k}{\log\epsilon_k - \log\epsilon_{k-1}}$ |
|---|---|---|---|
| 0 | $(0.7,\ 0.7)^T$ | 4.24e-01 | |
| 1 | $(0.87850000000000,\ 1.064285714285714)^T$ | 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 | $(1.00000000583852,\ 1.00000002726552)^T$ | 2.79e-08 | 1.77 |
| 5 | $(0.999999999999998,\ 1.000000000000000)^T$ | 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)* $\exists L \geq 0$: $\left\lVert DF(\mathbf{x})^{-1}(DF(\mathbf{x}+\mathbf{v}) - DF(\mathbf{x}))\right\rVert_2 \leq L\,\lVert\mathbf{v}\rVert_2 \quad \begin{array}{l}\forall \mathbf{v} \in \mathbb{R}^n, \mathbf{v}+\mathbf{x} \in D, \\ \forall \mathbf{x} \in D\end{array}$,

*(E)* $\exists \mathbf{x}^*$: $F(\mathbf{x}^*) = 0$ (*existence of solution in* $D$)

*(F) initial guess* $\mathbf{x}^{(0)} \in D$ *satisfies* $\rho := \left\lVert \mathbf{x}^* - \mathbf{x}^{(0)}\right\rVert_2 < \frac{2}{L} \ \wedge \ B_\rho(\mathbf{x}^*) \subset D$ .

**then** *the Newton iteration* (1.4.1) *satisfies:*

*(i)* $\mathbf{x}^{(k)} \in B_\rho(\mathbf{x}^*) := \{\mathbf{y} \in \mathbb{R}^n,\ \lVert\mathbf{y} - \mathbf{x}^*\rVert < \rho\}$ *for all* $k \in \mathbb{N}$,

*(ii)* $\lim\limits_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}^*$,

*(iii)* $\left\lVert \mathbf{x}^{(k+1)} - \mathbf{x}^*\right\rVert_2 \leq \frac{L}{2}\left\lVert \mathbf{x}^{(k)} - \mathbf{x}^*\right\rVert_2^2$ (*local quadratic convergence*) .

✎ notation: ball $B_\rho(\mathbf{z}) := \{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x}-\mathbf{z}\|_2 \le \rho\}$

Terminology: (D) $\hat{=}$ affine invariant Lipschitz condition

Problem: Usually neither $\omega$ 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:

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

➢ 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 $\tau$ = 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:

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

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

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

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 $DF(\mathbf{x}^{(k-1)})$ ➤ $\Delta\bar{\mathbf{x}}^{(k)}$ 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).

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

$$x^{(0)} < -1 \ \Rightarrow \ x^{(k)} \to -\infty$$
$$x^{(0)} > -1 \ \Rightarrow \ x^{(k)} \to x^*$$

◇

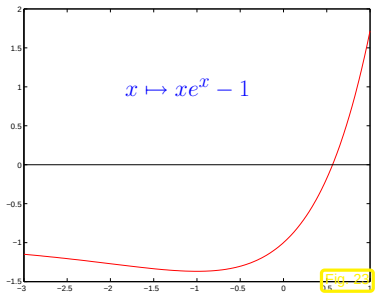



Fig. 24

*Example* 1.4.12 (Region of convergence of Newton method).

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

with zero $x^* = 0$ .

◇

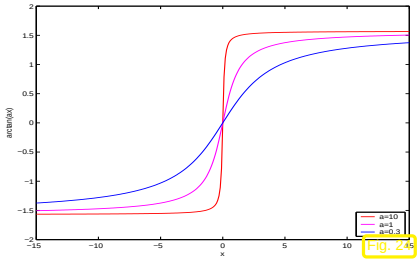
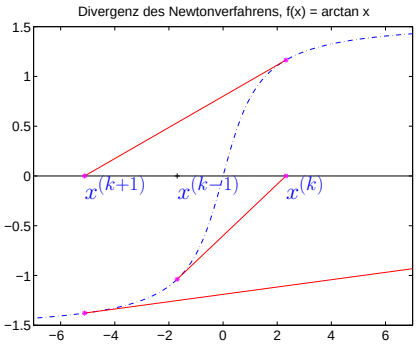
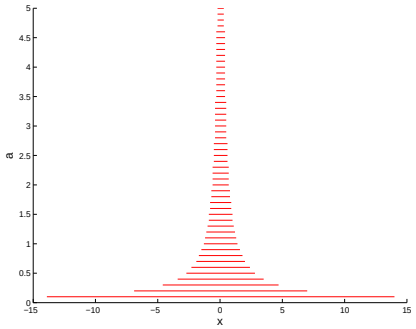



Fig. 25




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

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

"maximal" $\lambda^{(k)} > 0$: $\left\|\Delta\overline{\mathbf{x}}(\lambda^{(k)})\right\| \leq (1 - \frac{\lambda^{(k)}}{2}) \left\|\Delta\mathbf{x}^{(k)}\right\|_2$ (1.4.6)

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

Heuristics: convergence ⇔ 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 posterori 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). (→ 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 | -0.00000000000001 | -0.00000000000001 |

◇

*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 < LMIN`! | |

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

◇

### 1.4.5 Quasi-Newton Method

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

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

already computed ! → 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:

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

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.:

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

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

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

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

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

In conlcusion,
(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).

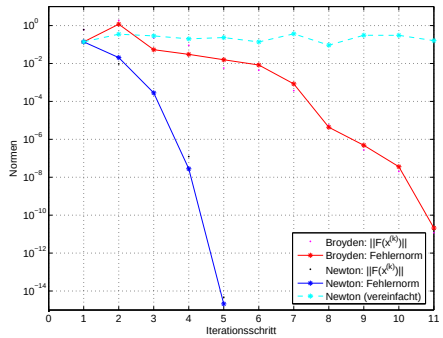
- In the non-linear system of the example 1.4.1, $n = 2$ take $\mathbf{x}^{(0)} = (0.7.0.7)^T$ and $\mathbf{J}_0 = DF(\mathbf{x}^{(0)})$

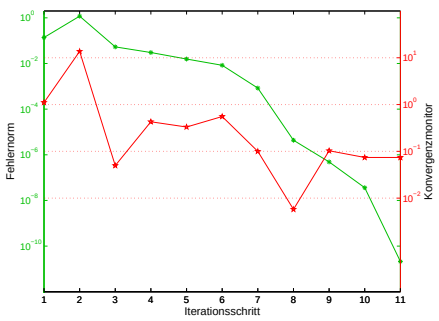
The numerical example shows that the method is:

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 $\mu > 1$

◇

*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

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

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 : $N$ steps

- $O(N^2 \cdot n)$ operations with vectors, (Level I)
- 1 LU-decomposition of J, $N\times$ solutions of SLEs, see section **??**
- $N$ evalutations of $F$ **!**

Memory cost : $N$ steps

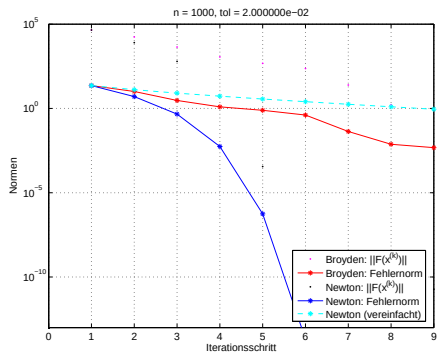
- LU-factors of J **+** auxiliary vectors $\in \mathbb{R}^n$
- $N$ vectors $\mathbf{x}^{(k)} \in \mathbb{R}^n$

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

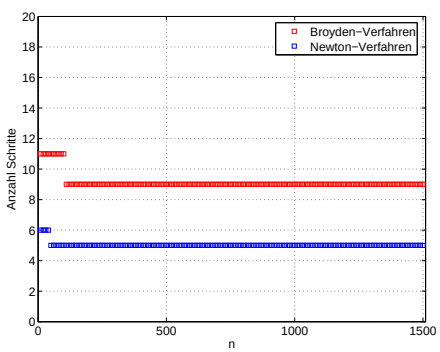
$$F(\mathbf{x}) = \begin{cases} \mathbb{R}^n & \mapsto \mathbb{R}^n \\ \mathbf{x} & \mapsto \operatorname{diag}(\mathbf{x})\mathbf{A}\mathbf{x} - \mathbf{b} \end{cases},$$
$$\mathbf{b} = (1, 2, \ldots, n) \in \mathbb{R}^n ,$$
$$\mathbf{A} = \mathbf{I} + \mathbf{a}\mathbf{a}^T \in \mathbb{R}^{n,n} ,$$
$$\mathbf{a} = \frac{1}{\sqrt{\mathbf{1} \cdot \mathbf{b} - \mathbf{1}}}(\mathbf{b} - \mathbf{1}) .$$

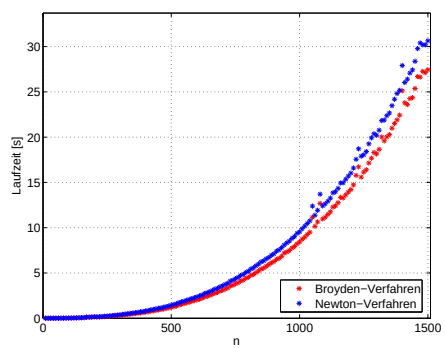
The interpretation of the results resemble the example 1.4.17 ▷

```
h = 2/n; x0 = (2:h:4-h)';
```




Efficiency comparison: Broyden method $\longleftrightarrow$ Newton method:
(in case of dimension $n$ use tolerance $\texttt{tol} = 2n \cdot 10^{-5}$, `h = 2/n; x0 = (2:h:4-h)';` )




In conclusion,
the Broyden method is worthwile 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 infinetly many |
| $A$ has full rank | $A$ has rank $r < n$ |
| $A$ has $n$ nonzero pivots | $A$ has $r < n$ pivots |
| span$\{A_{:,1}, \dots, A_{:,n}\}$ has dimension $n$ | span$\{A_{:,1}, \dots, A_{:,n}\}$ has dimension $r < n$ |
| span$\{A_{1,:}, \dots, A_{n,:}\}$ has dimension $n$ | span$\{A_{1,:}, \dots, A_{n,:}\}$ has dimension $r < n$ |
| All eigenvalues of $A$ are nonzero | $0$ is eigenvalue of $A$ |
| $0 \notin \sigma(A) =$ 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:

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

$$\begin{pmatrix} 1 & & \\ & 1 & \\ & & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{pmatrix} \begin{pmatrix} 4 \\ 1 \\ -3 \end{pmatrix} \Longrightarrow \begin{pmatrix} 1 & & \\ 2 & 1 & \\ & 0 & 1 \end{pmatrix} \begin{pmatrix} \boxed{1 \quad 1 \quad 0} \\ 0 \quad -1 \quad -1 \\ 3 \quad -1 \quad -1 \end{pmatrix} \begin{pmatrix} 4 \\ -7 \\ -3 \end{pmatrix}$$

$$\Longrightarrow \begin{pmatrix} 1 & & \\ 2 & 1 & \\ 3 & 0 & 1 \end{pmatrix} \begin{pmatrix} \boxed{1 \quad 1 \quad 0} \\ 0 \quad -1 \quad -1 \\ 0 \quad -4 \quad -1 \end{pmatrix} \begin{pmatrix} 4 \\ -7 \\ -15 \end{pmatrix} \Longrightarrow \underbrace{\begin{pmatrix} 1 & & \\ 2 & 1 & \\ 3 & 4 & 1 \end{pmatrix}}_{=L} \underbrace{\begin{pmatrix} 1 & 1 & 0 \\ 0 & \boxed{-1 \quad -1} \\ 0 & 0 & 3 \end{pmatrix}}_{=U} \begin{pmatrix} 4 \\ -7 \\ 13 \end{pmatrix}$$

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

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

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





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

$$\mathbf{A} = \mathbf{L}_1 \cdot \dots \cdot \mathbf{L}_{n-1} \mathbf{U} \quad \text{with} \quad \begin{array}{l} \text{elimination matrices } \mathbf{L}_i,\ i = 1, \dots, n-1 , \\ \text{upper triangular matrix } \mathbf{U} \in \mathbb{R}^{n,n} . \end{array}$$

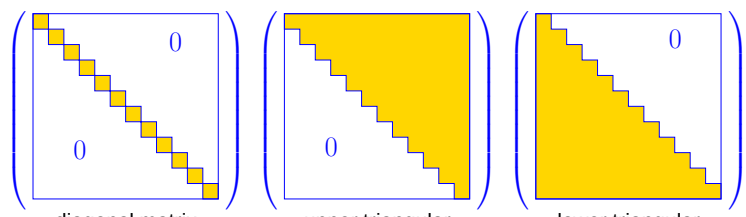
$$\begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ l_2 & 1 & & & 0 \\ l_3 & & \ddots & & \\ \vdots & & & \ddots & \\ l_n & 0 & & & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & & & 0 \\ 0 & h_3 & 1 & & \\ \vdots & \vdots & & \ddots & \\ 0 & h_n & 0 & & 1 \end{pmatrix} = \begin{pmatrix}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

$\blacktriangleright$ $\mathbf{L}_1 \cdot \dots \cdot \mathbf{L}_{n-1}$ 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).

$$\mathbf{Ax} = \mathbf{b} \quad : \quad \begin{array}{l} ① \ LU\text{-decomposition } \mathbf{A} = \mathbf{LU}, \text{ \#elementary operations } \frac{1}{3}n(n-1)(n+1) \\ ② \text{ forward substitution, solve } \mathbf{Lz} = \mathbf{b}, \text{ \#elementary operations } \frac{1}{2}n(n-1) \\ ③ \text{ backward substitution, solve } \mathbf{Ux} = \mathbf{z}, \text{ \#elementary operations } \frac{1}{2}n(n+1) \end{array}$$

Stability needs pivoting:

*Example* 2.0.3.

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 2 \\ 2 & -7 & 2 \\ 1 & 24 & 0 \end{pmatrix} \overset{❶}{\rightarrow} \begin{pmatrix} 2 & -7 & 2 \\ 1 & 2 & 2 \\ 1 & 24 & 0 \end{pmatrix} \overset{❷}{\rightarrow} \begin{pmatrix} 2 & -7 & 2 \\ 0 & 5.5 & 1 \\ 0 & 27.5 & -1 \end{pmatrix} \overset{❸}{\rightarrow} \begin{pmatrix} 2 & -7 & 2 \\ 0 & 27.5 & -1 \\ 0 & 5.5 & 1 \end{pmatrix} \overset{❹}{\rightarrow} \begin{pmatrix} 2 & -7 & 2 \\ 0 & 27.5 & -1 \\ 0 & 0 & 1.2 \end{pmatrix}$$

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

◇

**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 $\mathbf{A} \in \mathbb{R}^{n,n}$:* $\operatorname{cond}(\mathbf{A}) := \left\|\mathbf{A}^{-1}\right\| \|\mathbf{A}\|$

Note:

$$\operatorname{cond}(\mathbf{A}) \text{ depends on } \|\cdot\| \text{ !}$$

- If $\operatorname{cond}(\mathbf{A}) \gg 1$, *small perturbations* in $\mathbf{A}$ can lead to *large relative errors* in the solution of the LSE.
- If $\operatorname{cond}(\mathbf{A}) \gg 1$, an algorithm can produce solutions with large relative error !

**(1')** If $A$ is symmetric, then $A = LDL^H = R^H R$ with diagonal matrix $D$ containing the pivots (Choleski-decomposition) and $R = \sqrt{D}L^H$. No pivoting is necessary.
python-function: `scipy.linalg.cho_factor(A)`



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

**(3)** Singular value decomposition (SVD):

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

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

**(4)** Schur-decomposition:

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

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

**Theorem 2.1.2** (Criteria for Unitarity)**.**

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

► $\boxed{\mathbf{Q} \text{ unitary} \quad \Rightarrow \quad \operatorname{cond}(\mathbf{Q}) = 1}$ $\overset{(??)}{\rightarrow}$ unitary transformations enhance (numerical) stability

If $\mathbf{Q} \in \mathbb{K}^{n,n}$ 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 $\in \{z \in \mathbb{Z}: |z| = 1\}$.
- $\|\mathbf{QA}\|_2 = \|\mathbf{A}\|_2$ for any matrix $\mathbf{A} \in \mathbb{K}^{n,m}$

Drawbacks of $LU$-factorization:

☹ often pivoting required (➞ destroys structure, Ex. **??**, leads to fill-in)

☹ Possible (theoretical) instability of partial pivoting $\longrightarrow$ 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. **??**

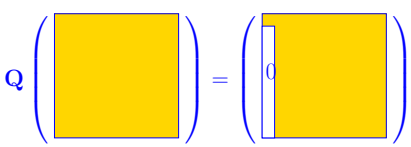
Which bijective row transformations preserve the Euclidean condition number of a matrix ?



➢ transformations hat 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.



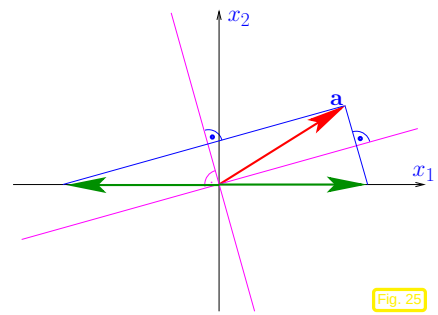

with $\mathbf{Q}^H = \mathbf{Q}^{-1}$ .

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.

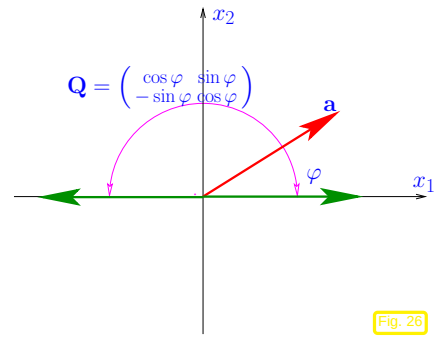
In 2D: two possible orthogonal transformations make 2nd component of $\mathbf{a} \in \mathbb{R}^2$ vanish:








*reflection* at angle bisector,




*rotation* turning $\mathbf{a}$ 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 $\mathbf{Qa} = \|\mathbf{a}\|_2 \mathbf{e}_1$, $\mathbf{e}_1 \hat{=}$ 1st unit vector.

Choice 1: Householder reflections

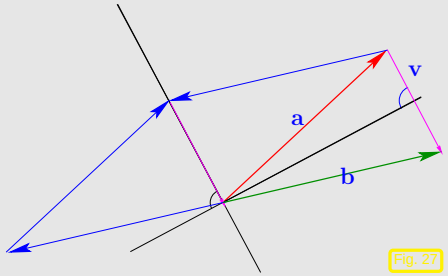
$$\mathbf{Q} = \mathbf{H}(\mathbf{v}) := \mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^H}{\mathbf{v}^H\mathbf{v}} \quad \text{with} \quad \mathbf{v} = \tfrac{1}{2}(\mathbf{a} \pm \|\mathbf{a}\|_2 \mathbf{e}_1) \; . \tag{2.1.1}$$





"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.




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

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

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

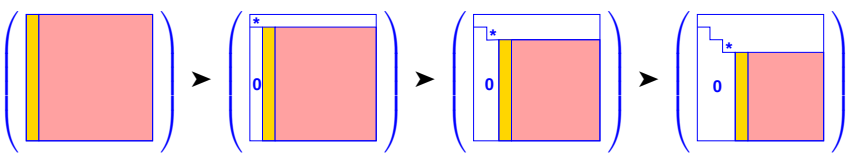
- efficient storage of Householder matrices ⟶ [3]

△

Choice 2: successive Givens rotations (⟶ 2D case)

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

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




= "target column $\mathbf{a}$" (determines unitary transformation),

= modified in course of transformations.

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

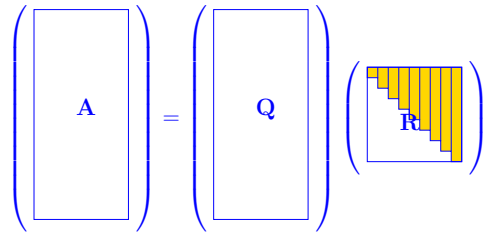




QR-factorization (QR-decomposition) of $\mathbf{A} \in \mathbb{C}^{n,n}$: $\mathbf{A} = \mathbf{QR}$, $\mathbf{Q} := \mathbf{Q}_1^H \cdot \cdots \cdot \mathbf{Q}_{n-1}^H$ unitary matrix, $\mathbf{R}$ upper triangular matrix.

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

$m > n$:




$$, \mathbf{A} = \mathbf{QR}\,, \quad \begin{array}{l} \mathbf{Q} \in \mathbb{K}^{m,n}\,, \\ \mathbf{R} \in \mathbb{K}^{n,n}\,, \end{array} \tag{2.1.3}$$

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

**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* $\operatorname{rank}(\mathbf{A}) = n$ *is unique, if we demand* $r_{ii} > 0$.

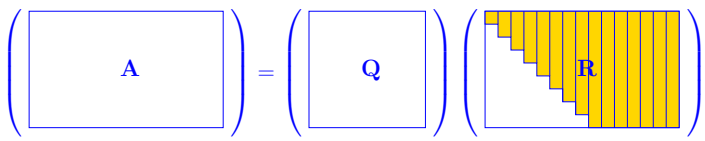
*Proof.* we observe that $\mathbf{R}$ is regular, if $\mathbf{A}$ has full rank $n$. Since the regular upper triangular matrices form a group under multiplication:

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

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

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

$m < n$:




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

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 (← more selective), if target column sparsely populated. △

functions:

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

`Q,R = qr(A,econ=True)` $\mathbf{Q} \in \mathbb{K}^{m,n}$, $\mathbf{R} \in \mathbb{K}^{n,n}$ for $\mathbf{A} \in \mathbb{K}^{m,n}$, $m > n$ (*economical* QR-factorization)

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

`Q,R = qr(A)` ➔ Costs: $O(m^2 n)$

`Q,R = qr(A,econ=True)` ➔ Costs: $O(mn^2)$

*Example* 2.1.3 (Complexity of Householder QR-factorization).

Code 2.1.4: timing QR-factorizations

```
from numpy import r_, mat, vstack, eye, ones, zeros
from scipy.linalg import qr
import timeit

def qr_full():
    global A
    Q, R = qr(A)

def qr_econ():
    global A
    Q, R = qr(A, econ=True)

def qr_ovecon():
    global A
    Q, R = qr(A, econ=True, overwrite_a=True)

def qr_r():
    global A
    R = qr(A, mode='r')

def qr_recon():
    global A
    R = qr(A, mode='r', econ=True)
```





```

nrEXP = 4
sizes = 2**r_[2:7]
qrtimes = zeros((5,sizes.shape[0]))
k = 0
for n in sizes:
    print 'n=', n
    m = n**2#4*n
    A = mat(1.*r_[1:m+1]).T*mat(1.*r_[1:n+1])
    A += vstack((eye(n), ones((m-n,n)) ))

    t = timeit.Timer('qr_full()','from __main__ import qr_full')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[0,k] = avqr

    t = timeit.Timer('qr_econ()','from __main__ import qr_econ')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[1,k] = avqr

    t = timeit.Timer('qr_ovecon()','from __main__ import qr_ovecon')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
```





```
    qrtimes[2,k] = avqr

    t = timeit.Timer('qr_r()','from __main__ import qr_r')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[3,k] = avqr

    t = timeit.Timer('qr_recon()','from __main__ import qr_recon')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[4,k] = avqr

    k += 1

#print qrtimes[3]
import matplotlib.pyplot as plt
plt.loglog(sizes, qrtimes[0], 's', label='qr')
plt.loglog(sizes, qrtimes[1], '*', label="qr(econ=True)")
plt.loglog(sizes, qrtimes[2], '.', label="qr(econ=True, overwrite_a=True)")
plt.loglog(sizes, qrtimes[3], 'o', label="qr(mode='r')")
plt.loglog(sizes, qrtimes[4], '+', label="qr(mode='r', econ=True)")
v4 =  qrtimes[1,1]* (sizes/sizes[1])**4
v6 =  qrtimes[0,1]* (sizes/sizes[1])**6
```

```
plt.loglog(sizes, v4, label='O(n^4)')
plt.loglog(sizes, v6, '--',label='O(n^6)')
plt.legend(loc=2)
plt.xlabel('n')
plt.ylabel('time [s]')
plt.savefig('qrtiming.eps')
plt.show()
```

timing of different variants of QR-factorization

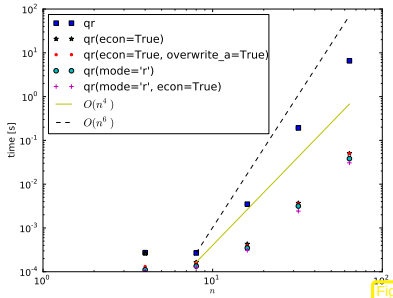
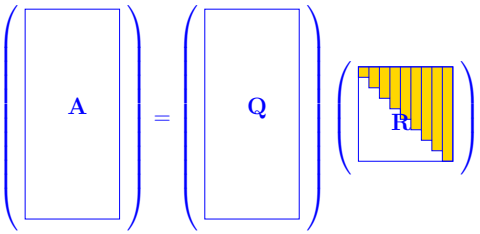



Fig. 28 ◇

*Remark* 2.1.5 (QR-orthogonalization).




, $\mathbf{A},\mathbf{Q}\in\mathbb{K}^{m,n}$,$\mathbf{R}\in\mathbb{K}^{n,n}$ .

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

➤ $\{\mathbf{q}_{\cdot,1},\dots,\mathbf{q}_{\cdot,n}\}$ is orthonormal basis of $\mathrm{Im}(\mathbf{A})$ with $\mathrm{Span}\left\{\mathbf{q}_{\cdot,1},\dots,\mathbf{q}_{\cdot,k}\right\}=\mathrm{Span}\left\{\mathbf{a}_{\cdot,1},\dots,\mathbf{a}_{\cdot,k}\right\}$ ,$1\le k\le n$ .

△

*Remark* 2.1.6 (Keeping track of unitary transformations).

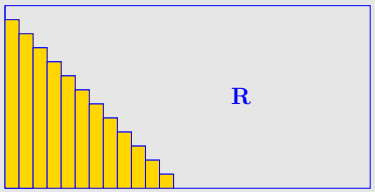
How to store $\begin{array}{l}\mathbf{G}_{i_1j_1}(a_1,b_1)\cdot\dots\cdot\mathbf{G}_{i_kj_k}(a_k,b_k)\,,\\ \mathbf{H}(\mathbf{v}_1)\cdot\dots\cdot\mathbf{H}(\mathbf{v}_k)\end{array}$ ?

☛ For Householder reflections

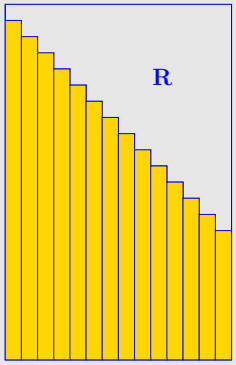
$\mathbf{H}(\mathbf{v}_1)\cdot\dots\cdot\mathbf{H}(\mathbf{v}_k)$: store $\mathbf{v}_1,\dots,\mathbf{v}_k$

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 $m>n$ →

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

$$\mathbf{G}=\begin{pmatrix}\bar\gamma & \bar\sigma\\ -\sigma & \gamma\end{pmatrix}\quad\Rightarrow\quad\text{store}\quad \rho:=\begin{cases}1 & \text{, if } \gamma=0\,,\\ \frac12\,\mathrm{sign}(\gamma)\sigma & \text{, if } |\sigma|<|\gamma|\,,\\ 2\,\mathrm{sign}(\sigma)/\gamma & \text{, if } |\sigma|\ge|\gamma|\,.\end{cases}$$

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



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

**Storing orthogonal transformation matrices is usually inefficient !**

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

$\mathbf{A}\mathbf{x}=\mathbf{b}$ :

① QR-decomposition $\mathbf{A}=\mathbf{Q}\mathbf{R}$, computational costs $\frac23 n^3+O(n^2)$ (about twice as expensive as $LU$-decomposition without pivoting)

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 $\mathbf{R}\mathbf{x}=\mathbf{z}$, computational costs $\frac12 n(n+1)$

☞ Computing the generalized QR-decomposition $\mathbf{A}=\mathbf{Q}\mathbf{R}$ 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. ↔ Gaussian elimination

```
from numpy import tril, vstack, hstack, eye, zeros, ones, dot, r_
from numpy.linalg import norm, solve, qr
sizes = r_[10:1001:10]
errlu = zeros(sizes.shape)
errqr = zeros(sizes.shape)
k = 0
for n in sizes:
    A = -tril(ones((n,n-1))) + 2*vstack((eye(n-1),zeros(n-1)))
    A = hstack((A, ones((n,1))))
    x = (-1.)**r_[1:n+1]
    b = dot(A,x)
    Q, R = qr(A)
    errlu[k] = norm(solve(A,b)-x)/norm(x)
    errqr[k] = norm(solve(R,dot(Q.T.conj(),b))-x)/norm(x)
    k += 1

import matplotlib.pyplot as plt
plt.semilogy(sizes, errlu, '*', label='LU')
plt.semilogy(sizes, errqr, 'ro', label='QR')
plt.legend(loc='center right')
plt.savefig('wilksolerr.eps')
plt.show()
```

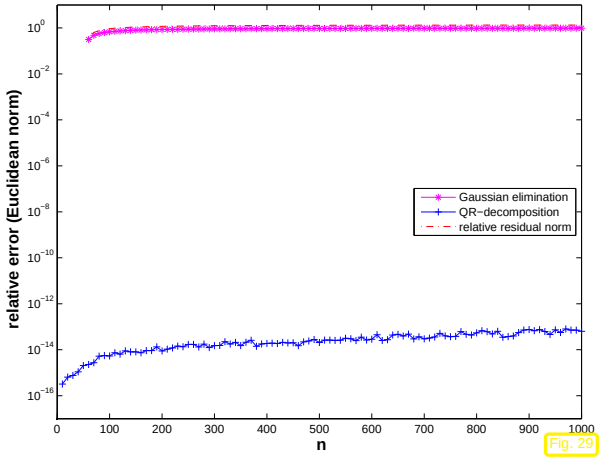



Fig. 29

◁ superior stability of QR-decomposition ! ◇

Fill-in for QR-decomposition **?**

bandwidth

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



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

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

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

◇

## 2.2 Singular Value Decomposition



*Remark* 2.2.1 (Principal component analysis (PCA)).

Given: $n$ data points $\mathbf{a}_j \in \mathbb{R}^m$, $j = 1, \ldots, n$, 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*

(➢ possibility of dimensional reduction)

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

Perspective of linear algebra:

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

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

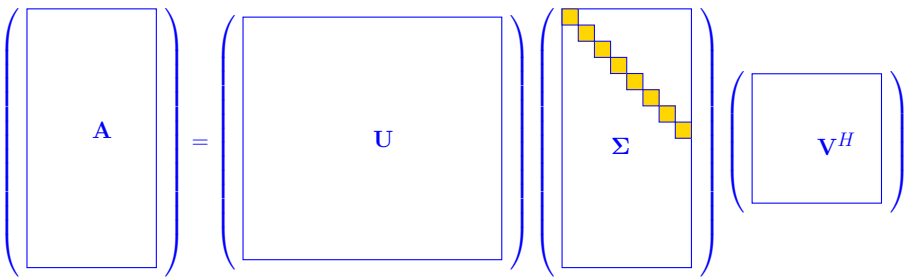
Application: ▶Chemometrics (multivariate calibration methods for the analysis of chemical mixtures)

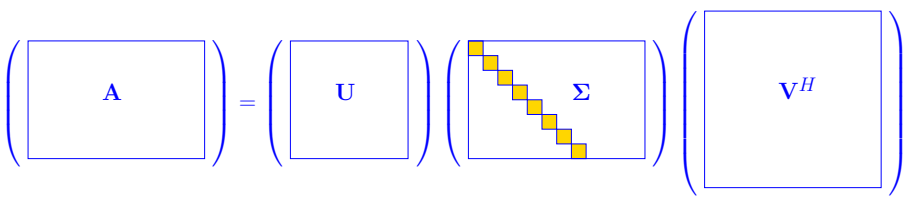
△

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

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

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




*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 $\{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x}\|_2 \leq 1\}$)

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

where we used the definition of the matrix 2-norm, see Def. 1.1.12. By Gram-Schmidt orthogonalization: $\exists \widetilde{\mathbf{V}} \in \mathbb{K}^{n,n-1}$, $\widetilde{\mathbf{U}} \in \mathbb{K}^{m,m-1}$ such that

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

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





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

$$\|\mathbf{A}_1\|_2^2 = \sup_{0 \neq \mathbf{x} \in \mathbb{K}^n} \frac{\|\mathbf{A}_1\mathbf{x}\|_2^2}{\|\mathbf{x}\|_2^2} \geq \frac{\left\|\mathbf{A}_1 \binom{\sigma}{\mathbf{w}}\right\|_2^2}{\left\|\binom{\sigma}{\mathbf{w}}\right\|_2^2} \geq \frac{(\sigma^2 + \mathbf{w}^H\mathbf{w})^2}{\sigma^2 + \mathbf{w}^H\mathbf{w}} = \sigma^2 + \mathbf{w}^H\mathbf{w} \ . \tag{2.2.1}$$

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

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

Then apply induction argument to $\mathbf{B}$ $\square$.

**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 $\sigma_i$ of $\mathbf{\Sigma}$ are the singular values of $\mathbf{A}$.*

**Lemma 2.2.3.** *The squares $\sigma_i^2$ of the non-zero singular values of $\mathbf{A}$ are the non-zero eigenvalues of $\mathbf{A}^H\mathbf{A}$, $\mathbf{A}\mathbf{A}^H$ with associated eigenvectors $(\mathbf{V})_{:,1}, \dots, (\mathbf{V})_{:,p}$, $(\mathbf{U})_{:,1}, \dots, (\mathbf{U})_{:,p}$, respectively.*

*Proof.* $\mathbf{A}\mathbf{A}^H$ and $\mathbf{A}^H\mathbf{A}$ are similar ($\rightarrow$ Lemma 4.1.4) to diagonal matrices with non-zero diagonal entries $\sigma_i^2$ ($\sigma_i \neq 0$), e.g.,

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

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

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

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

because $\operatorname{rank}(\mathbf{A}) = 1$ 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 $\mu(\mathbf{x}) = \mathbf{v}^H\mathbf{x}$.

$\blacktriangleright$ Singular value decomposition provides additive decomposition into rank-1 matrices:

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

*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

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

Two similar diagonal matrices are equal ! □

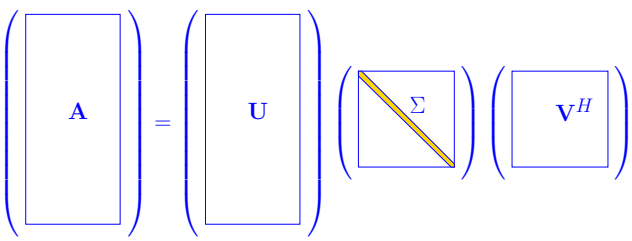
△

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

Explanation: "economical" singular value decomposition:






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

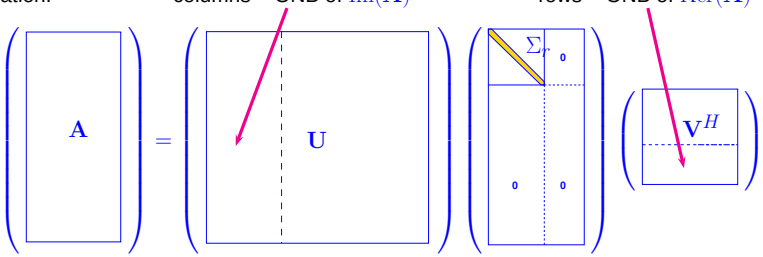
Complexity:
$2mn^2 + 2n^3 + O(n^2) + O(mn)$ for `s = svd(A)`,
$4m^2n + 22n^3 + O(mn) + O(n^2)$ for `[U,S,V] = svd(A)`,
$O(mn^2) + O(n^3)$ for `[U,S,V]=svd(A,0)`, $m \gg n$.

- 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* $\sigma_1 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots \sigma_p = 0$*, then*

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

Illustration: columns = ONB of $\mathrm{Im}(\mathbf{A})$ rows = ONB of $\mathrm{Ker}(\mathbf{A})$




(2.2.4)

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

Lemma 2.2.4 ► PCA by SVD

❶ no perturbations:

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

❷ with perturbations:

SVD: $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$ satisfies $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_p \gg \sigma_{p+1} \approx \cdots \approx \sigma_{\min\{m,n\}} \approx 0$ ,

$$V = \mathrm{Span}\underbrace{\{(\mathbf{U})_{:,1}, \ldots, (\mathbf{U})_{:,p}\}}_{\text{ONB of } V} .$$

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 $\mathrm{Im}(\mathbf{A})$ 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).

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

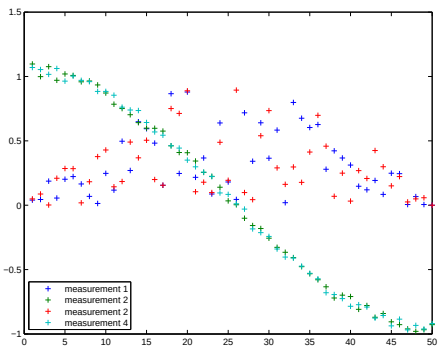
Columns $\mathbf{A}$ $\longrightarrow$ series of measurements at different times/locations etc.
Rows of $\mathbf{A}$ $\longrightarrow$ measured values corresponding to one time/location etc.

Goal: detect linear correlations

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

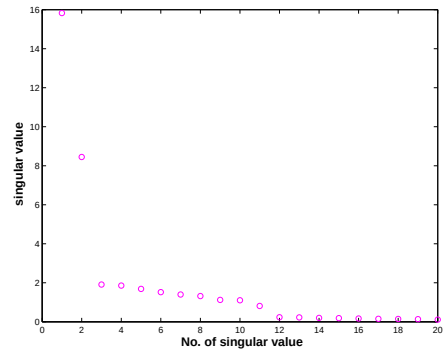
(Of course, measurements affected by errors/fluctuations)

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

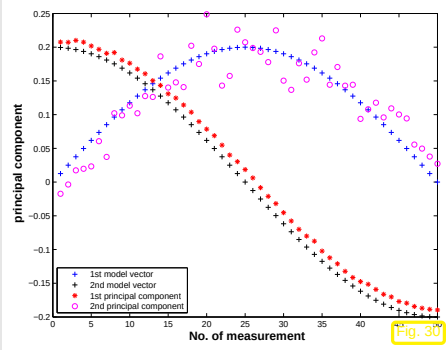



Fig. 30

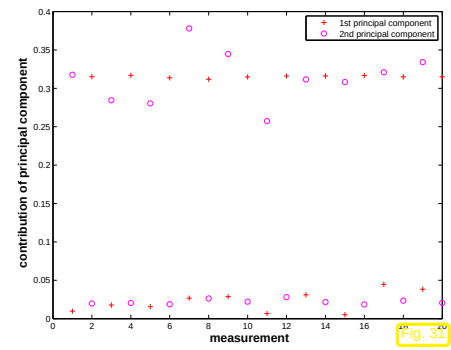



Fig. 31 ◇



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

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

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

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

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

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

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

By similar arguments:

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

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: Enchanced 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 $y_i, \mathbf{x}_i$, $y_i \in \mathbb{R}$, $\mathbf{x}_i \in \mathbb{R}^n$, $i = 1, \ldots, m$, $m \geq n+1$
($y_i$, $\mathbf{x}_i$ 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

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

linear regression for $n = 2, m = 8$ $\quad \triangleright$.

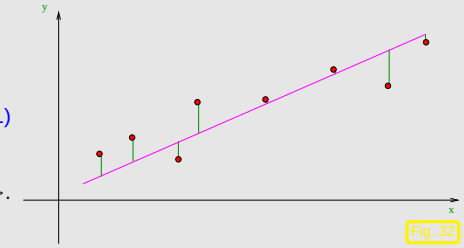


Fig. 32

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). ($\longrightarrow$ Ex. 3.3.1 for a related problem)

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

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

Special case: polynomial fit: $b_j(t) = t^{j-1}$.
MATLAB-function: `p = polyfit(t,y,n);` $\quad n$ = polynomial degree.

$\diamond$

*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) $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 = \inf\{\|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2 : \mathbf{y} \in \mathbb{K}^n\}$, (3.0.3)

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

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

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

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

In both cases the residual norm $\|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2$ 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.lstsqr(A,b)` Reassuring: stable (→ 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) $\|\mathbf{A}\mathbf{x} - \mathbf{b}\| \to \min$, $\|\mathbf{x}\| \to \min$.

`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, → 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 (→ Def. 2.2.2) of $\mathbf{A} \in \mathbb{K}^{m,n}$. Then*

$\sigma_1$

*Theorem* 3.0.4. *For $m \geq n$, $\mathbf{A} \in \mathbb{K}^{m,n}$, $\operatorname{rank}(\mathbf{A}) = n$, let $\mathbf{x} \in \mathbb{K}^n$ be the solution of the least squares problem $\|\mathbf{A}\mathbf{x} - \mathbf{b}\| \to \min$ and $\widehat{\mathbf{x}}$ the solution of the perturbed least squares problem $\|(\mathbf{A} + \Delta\mathbf{A})\widehat{\mathbf{x}} - \mathbf{b}\| \to \min$. Then*

$$\frac{\|\mathbf{x} - \widehat{\mathbf{x}}\|_2}{\|\mathbf{x}\|_2} \overset{\cdot}{\leq} \left( 2\operatorname{cond}_2(\mathbf{A}) + \operatorname{cond}_2^2(\mathbf{A}) \frac{\|\mathbf{r}\|_2}{\|\mathbf{A}\|_2 \|\mathbf{x}\|_2} \right) \frac{\|\Delta\mathbf{A}\|_2}{\|\mathbf{A}\|_2}$$

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

This means: if $\|\mathbf{r}\|_2 \ll 1$ ➤ condition of the least squares problem $\approx \operatorname{cond}_2(\mathbf{A})$
if $\|\mathbf{r}\|_2$ "large" ➤ condition of the least squares problem $\approx \operatorname{cond}_2^2(\mathbf{A})$

△

## 3.1 Normal Equations

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

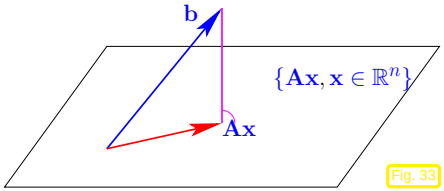



Fig. 33

Geometric interpretation of linear least squares problem (3.0.3):

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

Geometric interpretation: the least squares problem (3.0.3) amounts to searching the point $\mathbf{p} \in \operatorname{Im}(\mathbf{A})$ 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 $\operatorname{Im}(\mathbf{A})$, that is $\mathbf{b} - \mathbf{p} \perp \operatorname{Im}(\mathbf{A})$. Note the equivalence

$$\mathbf{b} - \mathbf{p} \perp \operatorname{Im}(\mathbf{A}) \quad \Leftrightarrow \quad \mathbf{b} - \mathbf{p} \perp (\mathbf{A})_{:,j}\,, \quad j = 1, \dots, n \quad \Leftrightarrow \quad \mathbf{A}^H(\mathbf{b} - \mathbf{p}) = 0\,,$$

Representation $\mathbf{p} = \mathbf{A}\mathbf{x}$ leads to normal equations (3.1.2).

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

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

A quadratic functional, *cf.* (**??**)

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

Minimization problem for $f$ ➢ study gradient, *cf.* (**??**)

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

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

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

*Remark* 3.1.1 (Conditioning of normal equations).

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

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



➢ 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 (→ 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.

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

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

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

◇

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

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

- 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:

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

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

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

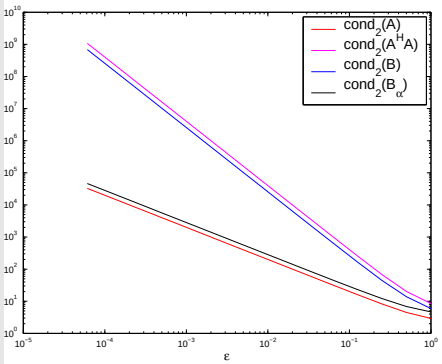
For $m, n \gg 1$, $\mathbf{A}$ sparse, both (3.1.3) and (3.1.4) lead to large sparse linear systems of equations, amenable to sparse direct elimination techniques, see Sect. **??**

*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 $\epsilon$ ($\alpha = \epsilon \|\mathbf{A}\|_2 / \sqrt{2}$)






## 3.2 Orthogonal Transformation Methods

Consider the linear least squares problem (3.0.3)

given $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$ 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 (→ Def. 2.1.1) leave 2-norm invariant.

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

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

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

As in the case of LSE (→ Sect. 2.1): "simpler form" = triangular form.

Concrete realization of this idea by means of QR-decomposition (→ 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.

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 = \left\|\mathbf{Q}(\mathbf{R}\mathbf{x} - \mathbf{Q}^H\mathbf{b})\right\|_2 = \left\|\mathbf{R}\mathbf{x} - \widetilde{\mathbf{b}}\right\|_2 \quad , \quad \widetilde{\mathbf{b}} := \mathbf{Q}^H\mathbf{b} .$$

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 \to \min \quad \Leftrightarrow \quad \left\| \begin{pmatrix} \mathbf{R} \\ \mathbf{0} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} - \begin{pmatrix} \widetilde{b}_1 \\ \vdots \\ \\ \vdots \\ \widetilde{b}_m \end{pmatrix} \right\|_2 \to \min .$$

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



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

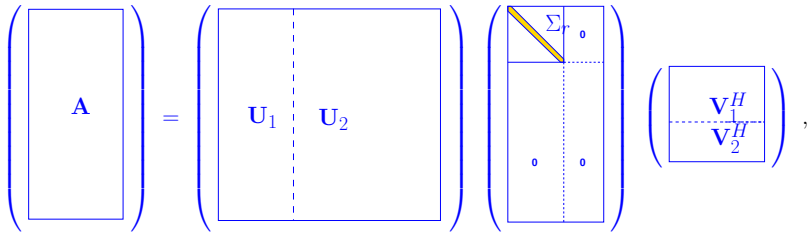
Implementation: successive orthogonal row transformations (by means of Householder reflections (2.1.1) for general matrices, and Givens rotations (2.1.2) for banded matrices, see Sect. 2.1 for details) of augmented matrix $(\mathbf{A}, \mathbf{b}) \in \mathbb{R}^{m,n+1}$, which is transformed into $(\mathbf{R},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

$\mathbf{Q}$ need not be stored !

Alternative: Solving linear least squares problem by SVD

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

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)

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

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

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:

➢ *underdetermined* linear system $\Sigma_r \mathbf{V}_1^H \mathbf{x} = \mathbf{U}_1^H \mathbf{b}_1$ . (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 $\operatorname{Ker}(\mathbf{V}_1) = \operatorname{Im}(\mathbf{V}_2)$. Since $\mathbf{V}$ is orthogonal, the minimal norm solution is obtained by setting contributions from $\operatorname{Im}(\mathbf{V}_2)$ to zero, which amounts to choosing $\mathbf{x} \in \operatorname{Im}(\mathbf{V}_1)$.

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

Practical implementation:

"numerical rank" test:

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

Code 3.2.1: Solving LSQ problem via SVD

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



*Remark* 3.2.2 (Pseudoinverse and SVD). → Rem. 3.0.4

The solution formula (3.2.4) directly yields a representation of the pseudoinverse $\mathbf{A}^+$ (→ 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 (→ 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 (→ 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: $f(t) = x_1 + x_2 \exp(-x_3 t)$, $n = 3$.

Determine parameters by non-linear least squares data fitting:

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

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

**Non-linear least squares problem**

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

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

As in the case of linear least squares problems (→ 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 (→ 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":

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

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

Simple idea: use Newton's method (→ 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 $\mathbf{grad}\,\Phi(\mathbf{x}) = 0$

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

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

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

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

$\Updownarrow$

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



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

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

*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 $\Phi$:

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

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

➢ 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).

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

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 $\Phi$ underlying Newton's method for (3.3.2), see Sect. 3.3.1, Rem. 3.3.2.





Gauss-Newton iteration

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

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

we solve a linear least squares problem in each step:

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

`x = numpy.linalg.lstsq(A,b)[0]` $\updownarrow$

$\mathbf{x}$ minimizer of $\|\mathbf{A}\mathbf{x}-\mathbf{b}\|_2$ with minimal 2-norm

Code 3.3.4: template for Gauss-Newton method

```
def gn(x,F,J,tol):
    s = lstsq(J(x),F(x))[0] #
    x = x-s
    while norm(s) > tol*norm(x): #
        s = lstsq(J(x),F(x))[0] #
        x = x-s
    return x
```

Comments on Code 3.3.2:

☞ Argument x passes initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$, argument F must be a *handle* to a function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$, argument J provides the Jacobian of $F$, namely $DF : \mathbb{R}^n \mapsto \mathbb{R}^{m,n}$, argument `tol` specifies the tolerance for termination





☞ Line 4: iteration terminates if relative norm of correction is below threshold specified in `tol`.

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)). $\longrightarrow$ Ex. 3.3.1

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

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

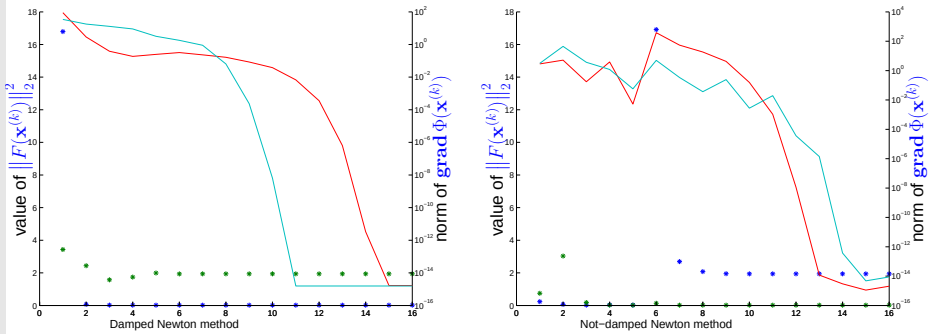
Numerical experiment:

convergence of the Newton method, damped Newton method ($\longrightarrow$ 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:

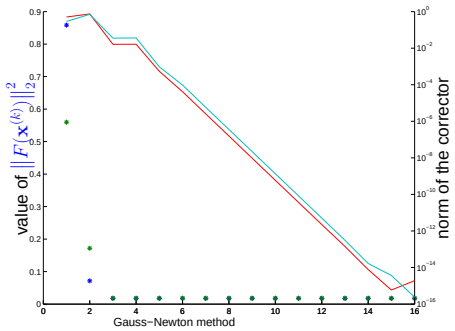
initial value $(1.8, 1.8, 0.1)^T$ (red curve) ➤ Newton method caught in local minimum,
initial value $(1.5, 1.5, 0.1)^T$ (cyan curve) ➤ fast (locally quadratic) convergence.

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

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

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

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

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

`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:

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

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 $m$ ► kinetic energy $K(\mathbf{x}, \dot{\mathbf{x}}) = \frac{m}{2} \left\|\dot{\mathbf{x}}\right\|_2^2$

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 conveninent 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 $\lambda_j$: $\mathbf{H}\mathbf{w}^j = \lambda_j\mathbf{w}^j$.

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):

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} \quad , \quad \mathbf{A} \in \mathbb{C}^{n,n} \ . \tag{4.0.1}$$

$$\mathbf{A} = \mathbf{S}\underbrace{\begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix}}_{=:\mathbf{D}}\mathbf{S}^{-1} \ , \quad \mathbf{S} \in \mathbb{C}^{n,n} \text{ regular} \implies \left(\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} \overset{\mathbf{z}=\mathbf{S}^{-1}\mathbf{y}}{\longleftrightarrow} \dot{\mathbf{z}} = \mathbf{D}\mathbf{z}\right) \ .$$



➢ solution of initial value problem:

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} \ , \quad \mathbf{y}(0) = \mathbf{y}_0 \in \mathbb{C}^n \quad \Rightarrow \quad \mathbf{y}(t) = \mathbf{S}\mathbf{z}(t) \ , \quad \dot{\mathbf{z}} = \mathbf{D}\mathbf{z} \ , \quad \mathbf{z}(0) = \mathbf{S}^{-1}\mathbf{y}_0 \ .$$

The initial value problem for the *decoupled* homogeneous linear ODE $\dot{\mathbf{z}} = \mathbf{D}\mathbf{z}$ has a simple analytic solution

$$\mathbf{z}_i(t) = \exp(\lambda_i t)(\mathbf{z}_0)_i = \exp(\lambda_i t)\left((\mathbf{S}^{-1})^T_{i,:}\mathbf{y}_0\right) \ .$$

In light of Rem. **??**:

$$\mathbf{A} = \mathbf{S}\begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix}\mathbf{S}^{-1} \quad \Leftrightarrow \quad \mathbf{A}\left((\mathbf{S})_{:,i}\right) = \lambda_i\left((\mathbf{S})_{:,i}\right) \quad i = 1, \dots, n \ . \tag{4.0.2}$$

In order to find the transformation matrix $\mathbf{S}$ all non-zero solution vectors (= eigenvectors) $\mathbf{x} \in \mathbb{C}^n$ of the linear eigenvalue problem

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

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 $\omega$ to be determined from the boundary conditions. In case of the string (one dimensional problem), we can obtain $\omega$ analyticaly, but in several diemensions not anymore.
A possibility is to use finite differences for the last differential equation ($x_i = x_1 + ih$, $y_i \approx y(x_i)$):

$$\frac{T}{m_i}\frac{y_{i-1} - 2y_i + y_{i+1}}{2h} + \omega^2 y_i = 0 \ , i = 1, 2, \ldots, N-1 \ bcom$$

which boilds down to the eigenvalue problem

$$\mathbf{A}\mathbf{y} = \omega^2 \mathbf{y} ,$$

with tridiagonal matrix $\mathbf{A}$ and $\mathbf{y} = (y_1, \ldots, y_{N-1})^T$.

◇

## 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* $\mathbf{A} \in \mathbb{K}^{n,n}$: $\sigma(\mathbf{A}) := \{\lambda \in \mathbb{C} \colon \lambda \text{ eigenvalue of } \mathbf{A}\}$
- *eigenspace* (ger.: *Eigenraum*) *associated with eigenvalue* $\lambda \in \sigma(\mathbf{A})$:
  $$\mathrm{Eig}_{\mathbf{A}}(\lambda) := \mathrm{Ker}(\lambda\mathbf{I} - \mathbf{A})$$
- $\mathbf{x} \in \mathrm{Eig}_{\mathbf{A}}(\lambda) \setminus \{0\}$ $\Rightarrow$ $\mathbf{x}$ *is eigenvector*
- *Geometric multiplicity (ger.: Vielfachheit) of an eigenvalue* $\lambda \in \sigma(\mathbf{A})$:



Two simple facts:

$$\lambda \in \sigma(\mathbf{A}) \quad \Rightarrow \quad \dim \mathrm{Eig}_{\mathbf{A}}(\lambda) > 0 , \tag{4.1.1}$$

$$\det(\mathbf{A}) = \det(\mathbf{A}^T) \quad \forall \mathbf{A} \in \mathbb{K}^{n,n} \quad \Rightarrow \quad \sigma(\mathbf{A}) = \sigma(\mathbf{A}^T) . \tag{4.1.2}$$

✎ notation: $\rho(\mathbf{A}) := \max\{|\lambda| \colon \lambda \in \sigma(\mathbf{A})\} \mathrel{\hat{=}}$ 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 (→ 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*

$$\sigma(\mathbf{A}) \subset \bigcup_{j=1}^{n} \{z \in \mathbb{C} \colon |z - a_{jj}| \leq \sum\nolimits_{i \neq j} |a_{ji}|\} .$$

**Lemma 4.1.4** (Similarity and spectrum)**.**
*The spectrum of a matrix is invariant with respect to similarity transformations:*

$$\forall \mathbf{A} \in \mathbb{K}^{n,n} \colon \quad \sigma(\mathbf{S}^{-1}\mathbf{A}\mathbf{S}) = \sigma(\mathbf{A}) \quad \forall \text{ regular } \mathbf{S} \in \mathbb{K}^{n,n} .$$

**Lemma 4.1.5.** *Existence of a one-dimensional invariant subspace*

$$\forall \mathbf{C} \in \mathbb{C}^{n,n} \colon \quad \exists \mathbf{u} \in \mathbb{C}^n \colon \quad \mathbf{C}\,(\mathrm{Span}\,\{\mathbf{u}\}) \subset \mathrm{Span}\,\{\mathbf{u}\} .$$

**Theorem 4.1.6** (Schur normal form)**.**

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

**Corollary 4.1.7** (Principal axis transformation).

$\mathbf{A} \in \mathbb{K}^{n,n}$, $\mathbf{A}\mathbf{A}^H = \mathbf{A}^H\mathbf{A}$: $\exists \mathbf{U} \in \mathbb{C}^{n,n}$ *unitary*: $\mathbf{U}^H\mathbf{A}\mathbf{U} = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$, $\lambda_i \in \mathbb{C}$.

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: $\mathbf{A}^H = \mathbf{A}$ ➤ $\sigma(\mathbf{A}) \subset \mathbb{R}$
- unitary matrices: $\mathbf{A}^H = \mathbf{A}^{-1}$ ➤ $|\sigma(\mathbf{A})| = 1$
- skew-Hermitian matrices: $\mathbf{A} = -\mathbf{A}^H$ ➤ $\sigma(\mathbf{A}) \subset i\mathbb{R}$

➤

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 $\mathbf{x} \neq 0$, $\lambda \in \mathbb{C}$

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x} \quad \Leftrightarrow \quad \mathbf{B}^{-1}\mathbf{A}\mathbf{x} = \lambda\mathbf{x}\,. \tag{4.1.3}$$

$\mathbf{x} \hat{=}$ generalized eigenvector, $\lambda \hat{=}$ generalized eigenvalue

Obviously every generalized eigenvalue problem is equivalent to a standard eigenvalue problem

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x} \quad \Leftrightarrow \quad \mathbf{B}^{-1}\mathbf{A} = \lambda\mathbf{x}\,.$$

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. (→ Def. **??**) with Cholesky factorization $\mathbf{B} = \mathbf{R}^H\mathbf{R}$

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x} \quad \Leftrightarrow \quad \widetilde{\mathbf{A}}\mathbf{y} = \lambda\mathbf{y} \quad \text{where } \widetilde{\mathbf{A}} := \mathbf{R}^{-H}\mathbf{A}\mathbf{R}^{-1},\ \mathbf{y} := \mathbf{R}\mathbf{x}\,.$$



➞ 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`

| | | |
|---|---|---|
| `w = eigvals(A)` | : | computes spectrum $\sigma(\mathbf{A}) = \{w_1, \dots, w_n\}$ of $\mathbf{A} \in \mathbb{C}^{n,n}$ |
| `w, V = eig(A)` | : | computes spectrum $w$ and corresponding normed eigenvectors |
| `eigvalsh(A)` and `eigh(A)` | : | specialized algorithms for Hermitial matrices |
| | ⇒ | wrappers to `lapack`-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

$$\mathbf{A} = \mathbf{A}^{(0)} \longrightarrow \mathbf{A}^{(1)} \longrightarrow \dots \longrightarrow \begin{cases} \text{diagonal matrix} & \text{, if } \mathbf{A} = \mathbf{A}^H\,, \\ \text{upper triangular matrix} & \text{, else.} \\ (\longrightarrow \text{Thm. 4.1.6}) \end{cases}$$

(superior stability of unitary transformations, see Rem. **??**)

Code 4.2.2: QR-algorithm with shift

```
'''
QR-algorithm with shift
'''
from numpy import mat, argmin, eye, tril
from numpy.linalg import norm, eig, qr, eigvals

def eigqr(A,tol):
    n = A.shape[0]
    while (norm(tril(A,-1), ord=2) > tol*norm(A,
      ord=2)):
        # shift by ew of lower right 2×2 block closest to (A)_{n,n}
        sc, dummy = eig(A[n-2:n,n-2:n])
        k = argmin(abs(sc - A[n-1,n-1]))
        shift = sc[k]
        Q, R = qr( A - shift * eye(n))
        A = mat(Q).H*mat(A)*mat(Q);
    d = A.diagonal()
    return d

if __name__ == "__main__":
    A = mat('1,2,3; 4,5,6; 7,8,9')
    print 'numpy.linalg.eigvals:', eigvals(A)
    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])

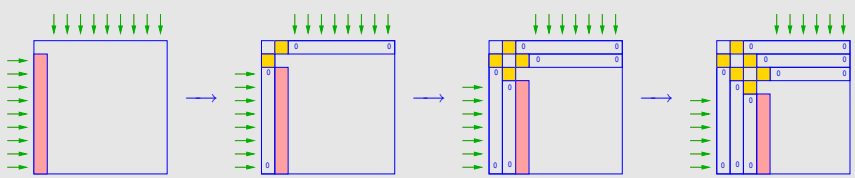
Computational cost: $O(n^3)$ operations per step of the QR-algorithm

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 $\mathbf{A} = \mathbf{A}^H$:
(→ ≙ affected rows/columns, ▭ ≙ targeted vector)



transformation to tridiagonal form ! (for general matrices a similar strategy can achieve a similarity transformation to upper Hessenberg form)





this transformation is used as a preprocessing step for QR-algorithm ➢ `eig`.

Similar functionality for generalized EVP $\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x}$, $\mathbf{A},\mathbf{B} \in \mathbb{C}^{n,n}$

`scipy.linalg.eig(a, b=None, left=False, right=True)`: also for generalized eigenvalue problems

Note: (Generalized) eigenvectors can be recovered as columns of $\mathbf{V}$:

$$\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{D} \quad \Leftrightarrow \quad \mathbf{A}(\mathbf{V})_{:,i} = (\mathbf{D})_{i,i}\mathbf{V}_{:,i} \,,$$

if $\mathbf{D} = \operatorname{diag}(d_1,\dots,d_n)$.

*Remark* 4.2.4 (Computational effort for eigenvalue computations).

Computational effort (#elementary operations) for `eig()`:

| | | |
|---|---|---|
| eigenvalues & eigenvectors of $\mathbf{A} \in \mathbb{K}^{n,n}$ | $\sim 25n^3 + O(n^2)$ | |
| only eigenvalues of $\mathbf{A} \in \mathbb{K}^{n,n}$ | $\sim 10n^3 + O(n^2)$ | $O(n^3)$! |
| 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
```

```python
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
```



```python
    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)
```



```python
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()
```

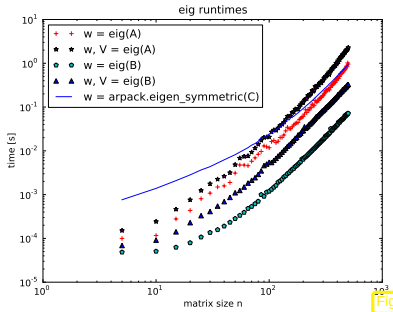


Fig. 34

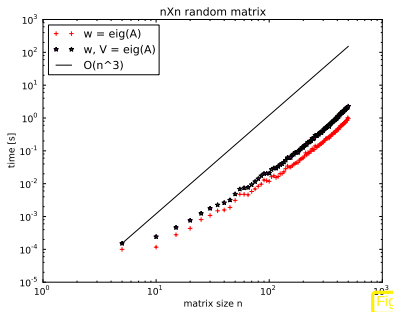


Fig. 35



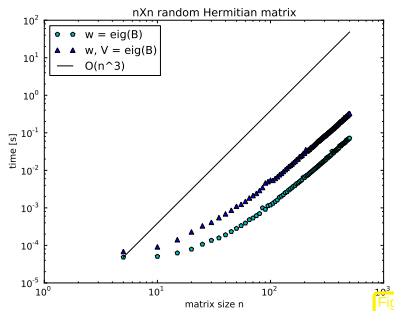


Fig. 36

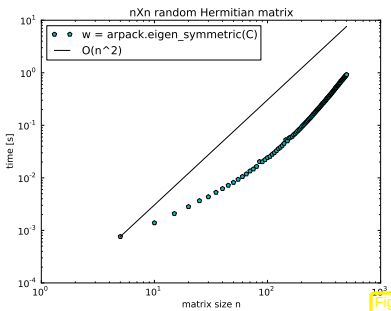


Fig. 37

☛ For the sake of efficiency: think which information you really need when computing eigenvalues/eigenvectors of dense matrices

Potentially more efficient methods for *sparse matrices* will be introduced below in Sects. 4.3, 4.4.

◇

## 4.3 Power Methods

### 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: $\mathbf{S}^{-1}\mathbf{A}\mathbf{S} = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$

$$\mathbf{z} = \sum_{j=1}^{n} \zeta_j (\mathbf{S})_{:,j} \quad \Rightarrow \quad \mathbf{A}^k \mathbf{z} = \sum_{j=1}^{n} \zeta_j \lambda_j^k (\mathbf{S})_{:,j} \,.$$

If $|\lambda_1| \leq |\lambda_2| \leq \dots \leq |\lambda_{n-1}| < |\lambda_n|$, $\left\|(\mathbf{S})_{:,j}\right\|_2 = 1$, $j = 1, \dots, n$, $\zeta_n \neq 0$

$$\blacktriangleright \quad \frac{\mathbf{A}^k \mathbf{z}}{\|\mathbf{A}^k \mathbf{z}\|} \to \pm(\mathbf{S})_{:,n} \quad = \text{eigenvector for } \lambda_n \quad \text{for } k \to \infty \,. \tag{4.3.1}$$

Suggests direct power method (*ger.:* Potenzmethode): iterative method (→ Sect. 1.1)

$$\text{initial guess:}\quad \mathbf{z}^{(0)} \text{ "arbitrary"}\,,$$
$$\text{next iterate:}\quad \mathbf{w} := \mathbf{A}\mathbf{z}^{(k-1)}\,,\quad \mathbf{z}^{(k)} := \frac{\mathbf{w}}{\|\mathbf{w}\|_2}\,,\quad k = 1,2,\dots\,. \tag{4.3.2}$$

Computational effort: $1\times$ matrix$\times$vector per step ➢ inexpensive for sparse matrices

$\mathbf{z}^{(k)} \longrightarrow$ eigenvector, but how do we get the associated eigenvalue $\lambda_n$ ?

❶ upon convergence from (4.3.1) ➢ $\mathbf{A}\mathbf{z}^{(k)} \approx \lambda_n \mathbf{z}^{(k)}$ ➢ $|\lambda_n| \approx \dfrac{\left\|\mathbf{A}\mathbf{z}^{(k)}\right\|}{\left\|\mathbf{z}^{(k)}\right\|}$

❷ for $\mathbf{A} = \mathbf{A}^T \in \mathbb{R}^{n,n}$: $\lambda_n \approx \operatorname*{argmin}_{\theta\in\mathbb{R}} \left\|\mathbf{A}\mathbf{z}^{(k)} - \theta\mathbf{z}^{(k)}\right\|_2^2$ ➢ $\lambda_n \approx \dfrac{(\mathbf{z}^{(k)})^T\mathbf{A}\mathbf{z}^{(k)}}{\left\|\mathbf{z}^{(k)}\right\|^2}$ .

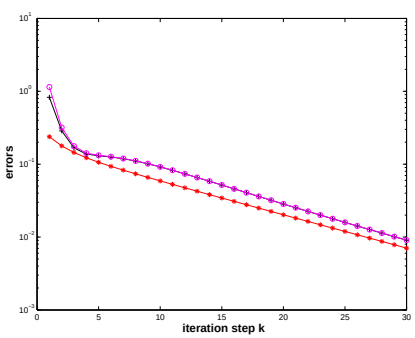
**Definition 4.3.1.** *For $\mathbf{A} \in \mathbb{K}^{n,n}$, $\mathbf{u} \in \mathbb{K}^n$ the Rayleigh quotient is defined by*

$$\rho_{\mathbf{A}}(\mathbf{u}) := \frac{\mathbf{u}^H\mathbf{A}\mathbf{u}}{\mathbf{u}^H\mathbf{u}}\,.$$

An immediate consequence of the definitions:

$$\lambda \in \sigma(\mathbf{A})\quad,\quad \mathbf{z} \in \mathrm{Eig}_\lambda(\mathbf{A}) \quad\Rightarrow\quad \rho_{\mathbf{A}}(\mathbf{z}) = \lambda\,. \tag{4.3.3}$$

*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)) )
```

◁ o : error $|\lambda_n - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})|$

$*$ : error norm $\left\|\mathbf{z}^{(k)} - \mathbf{s}_{\cdot,n}\right\|$

$+$ : $\left|\lambda_n - \dfrac{\left\|\mathbf{A}\mathbf{z}^{(k-1)}\right\|_2}{\left\|\mathbf{z}^{(k-1)}\right\|_2}\right|$

$\mathbf{z}^{(0)}$ = random vector





Test matrices:

① `d = 1.+r_[0:n]` ➢ $|\lambda_{n-1}| : |\lambda_n| = 0.9$

② `d = ones(n); d[-1] = 2.` ➢ $|\lambda_{n-1}| : |\lambda_n| = 0.5$

③ `d = 1.-0.5**r_[1:5.1:0.5]` ➢ $|\lambda_{n-1}| : |\lambda_n| = 0.9866$

$$\rho_{\mathrm{EV}}^{(k)} := \frac{\left\|\mathbf{z}^{(k)} - \mathbf{s}_{\cdot,n}\right\|}{\left\|\mathbf{z}^{(k-1)} - \mathbf{s}_{\cdot,n}\right\|}\,,$$
$$\rho_{\mathrm{EW}}^{(k)} := \frac{|\rho_{\mathbf{A}}(\mathbf{z}^{(k)}) - \lambda_n|}{|\rho_{\mathbf{A}}(\mathbf{z}^{(k-1)}) - \lambda_n|}\,.$$

| | ① | | ② | | ③ | |
|---|---|---|---|---|---|---|
| $k$ | $\rho_{\mathrm{EV}}^{(k)}$ | $\rho_{\mathrm{EW}}^{(k)}$ | $\rho_{\mathrm{EV}}^{(k)}$ | $\rho_{\mathrm{EW}}^{(k)}$ | $\rho_{\mathrm{EV}}^{(k)}$ | $\rho_{\mathrm{EW}}^{(k)}$ |
| 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 $\lambda_n$ normalized according to $\|\mathbf{y}\|_2 = \|\mathbf{v}\|_2 = 1$. Then there is convergence*

$$\left\|\mathbf{A}\mathbf{z}^{(k)}\right\|_2 \to \lambda_n \quad,\quad \mathbf{z}^{(k)} \to \pm\mathbf{v} \quad \textit{linearly with rate} \quad \frac{|\lambda_{n-1}|}{|\lambda_n|}\,,$$

*where $\mathbf{z}^{(k)}$ are the iterates of the direct power iteration and $\mathbf{y}^H\mathbf{z}^{(0)} \neq 0$ is assumed.*

*Remark* 4.3.2 (Initial guess for power iteration).

roundoff errors ➢ $\mathbf{y}^H\mathbf{z}^{(0)} \neq 0$ always satisfied in practical computations

Usual (not the best!) choice for $\mathbf{x}^{(0)}$ = random vector

△

*Remark* 4.3.3 (Termination criterion for direct power iteration). (→ Sect. 1.1.2)

Adaptation of a posteriori termination criterion (1.2.7)

"relative change" $\leq$ tol:

$$\begin{cases} \left\| \mathbf{z}^{(k)} - \mathbf{z}^{(k-1)} \right\| \leq (1/L - 1)\texttt{tol} \ , \\ \left| \frac{\left\| \mathbf{A}\mathbf{z}^{(k)} \right\|}{\left\| \mathbf{z}^{(k)} \right\|} - \frac{\left\| \mathbf{A}\mathbf{z}^{(k-1)} \right\|}{\left\| \mathbf{z}^{(k-1)} \right\|} \right| \leq (1/L - 1)\texttt{tol} \quad \text{see (1.1.17)} \ . \end{cases}$$

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:

Smallest (in modulus) EV of $\mathbf{A}$ = $\left(\text{Largest (in modulus) EV of } \mathbf{A}^{-1}\right)^{-1}$

Direct power method (→ Sect. 4.3.1) for $\mathbf{A}^{-1}$ = inverse iteration

Code 4.3.4: inverse iteration for computing $\lambda_{\min}(\mathbf{A})$ and associated eigenvector

```
import numpy as np
import scipy.linalg as splalg

def invit(A,tol):
    LUP = splalg.lu_factor(A, overwrite_a=True)
    n = A.shape[0]
    x = np.random.rand(n)
    x /= np.linalg.norm(x)
    splalg.lu_solve(LUP, x, overwrite_b=True)
    lold = 0
    lmin = 1./np.linalg.norm(x)
    x *= lmin
    while(abs(lmin-lold) > tol*lmin):
        lold = lmin
        splalg.lu_solve(LUP, x, overwrite_b=True)
        lmin = 1./np.linalg.norm(x)
        x *= lmin
    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:

$$\mathbf{z}^{(0)} \text{ arbitrary} \ , \quad \mathbf{w} = (\mathbf{A} - \alpha\mathbf{I})^{-1}\mathbf{z}^{(k-1)} \quad , \quad \mathbf{z}^{(k)} := \frac{\mathbf{w}}{\|\mathbf{w}\|_2} \ , \quad k = 1, 2, \dots \ , \tag{4.3.4}$$

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 (↔ a single LU-factorization of $\mathbf{A} - \alpha\mathbf{I}$ as in Code 4.3.3).

What if "by accident" $\alpha \in \sigma(\mathbf{A})$ (⇔ $\mathbf{A} - \alpha\mathbf{I}$ singular) ?

| |
|---|
| Stability of Gaussian elimination/LU-factorization will ensure that "$\mathbf{w}$ 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 $\longrightarrow \lambda_j$ with rate

$$\frac{|\lambda_j - \alpha|}{\min\{|\lambda_i - \alpha|, i \neq j\}} \quad \text{with} \quad \lambda_j \in \sigma(\mathbf{A}) \ , \quad |\alpha - \lambda_j| \leq |\alpha - \lambda| \quad \forall \lambda \in \sigma(\mathbf{A}) \ .$$

Extremely fast for $\alpha \approx \lambda_j$ !

△

Idea: A posteriori adaptation of shift

Use $\alpha := \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})$ in $k$-th step of inverse iteration.

*Algorithm* 4.3.6 (Rayleigh quotient iteration).

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

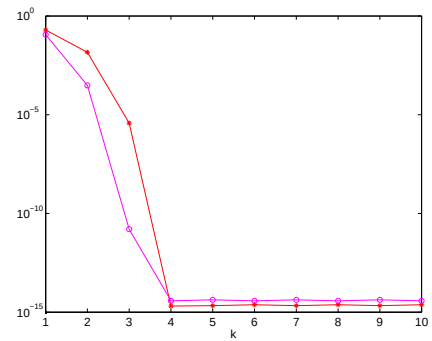
```
import numpy as np
def rqi(A,maxit):
    # For calculating the errors, the eigenvalues are calculated here with eig
    w, V = np.linalg.eig(A)
    t = np.where(w == abs(w).min())
    k = t[0];
    if len(k) > 1:
        print 'Error: no single smallest EV'
        raise ValueError
    ev = V[:,k[0]];  ev /= np.linalg.norm(ev)
    ew = w[k[0]]
    #
    n = A.shape[0]
    alpha = 0.
    z = np.random.rand(n); z /= np.linalg.norm(z)
    for k in xrange(maxit):
        z = np.linalg.solve(A-alpha*np.eye(n), z)
        z /= np.linalg.norm(z)
        alpha = np.dot(np.dot(A,z),z)
        ea = abs(alpha-ew)
        eb = np.min(np.linalg.norm(z-ev),np.linalg.norm(z+ev))
        print 'ea, eb =', ea, eb

def runit(d, tol, func):
    n = len(d) # size of the matrix
    Z = np.diag(np.sqrt(np.r_[n:0:-1])) + np.ones((n,n))
    Q,R = np.linalg.qr(Z)
    A = np.dot(Q, np.dot(np.diag(d), np.linalg.inv(Q)) )
    res = func(A,tol)
    print res

# ----------------------------
if __name__=='__main__':
    n = 10 # size of the matrix
    d = 1.+np.r_[0:n] # eigenvalues
    print 'd =', d, 'd[-2]/d[-1] =', d[-2]/d[-1]
    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. $\mathbf{A} \in \mathbb{R}^{n,n}$




o : $|\lambda_{\min} - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})|$

$*$ : $\left\|\mathbf{z}^{(k)} - \mathbf{x}_j\right\|$, $\lambda_{\min} = \lambda_j$, $\mathbf{x}_j \in \mathrm{Eig}_{\mathbf{A}}(\lambda_j)$,

: $\left\|\mathbf{x}_j\right\|_2 = 1$





| $k$ | $\lvert\lambda_{\min} - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})\rvert$ | $\left\lVert\mathbf{z}^{(k)} - \mathbf{x}_j\right\rVert$ |
|---|---|---|
| 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 (→ *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 (→ 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 $\mathbf{A}\times$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.

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 $\mathbf{B}^{-1} \approx \mathbf{A}^{-1}$

$\rightarrow$ $\mathbf{B} \hat{=}$ 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 $\mathbf{A}^{-1} \to \mathbf{B}^{-1}$ possible only when applied to residual quantity

residual quantity **=** quantity that $\to 0$ in the case of convergence to exact solution

Natural residual quantity for eigenvalue problem $\mathbf{Ax} = \lambda\mathbf{x}$:

$$\mathbf{r} := \mathbf{Az} - \rho_{\mathbf{A}}(\mathbf{z})\mathbf{z} \quad , \quad \rho_{\mathbf{A}}(\mathbf{z}) = \text{Rayleigh quotient} \to \text{Def. 4.3.1 .}$$

Note: only *direction* of $\mathbf{A}^{-1}\mathbf{z}$ matters in inverse iteration (4.3.4)

$$(\mathbf{A}^{-1}\mathbf{z}) \parallel (\mathbf{z} - \mathbf{A}^{-1}(\mathbf{Az} - \rho_{\mathbf{A}}(\mathbf{z})\mathbf{z})) \quad \Rightarrow \quad \text{defines same next iterate!}$$

[Preconditioned inverse iteration (PINVIT) for s.p.d. $\mathbf{A}$]





$$\mathbf{z}^{(0)} \text{ arbitrary,} \quad \begin{aligned} \mathbf{w} &= \mathbf{z}^{(k-1)} - \mathbf{B}^{-1}(\mathbf{Az}^{(k-1)} - \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{z}^{(k-1)}) \,, \\ \mathbf{z}^{(k)} &= \frac{\mathbf{w}}{\|\mathbf{w}\|_2} \,, \end{aligned} \quad k = 1, 2, \dots \,. \tag{4.3.5}$$

Code 4.3.9: preconditioned inverse iteration (4.3.5)

```python
from numpy import dot, ones, tile, array, linspace
from numpy.linalg import norm
from scipy.sparse import spdiags
from scipy.sparse.linalg.eigen.arpack import arpack
from scipy.sparse.linalg import spsolve

def pinvit(evA, invB, tol, maxit):
    n = A.shape[0]
    #z = ones(n)/n
    z = linspace(1,n,n)
    z = z/norm(z)

    res = []; rho = 0
    for k in xrange(maxit):
        v = evA(z)
        rhon = dot(v,z) # Rayleigh quotient
        r = v - rhon*z # residual
        z = z - invB(r) # iteration (4.3.5)
        z /= norm(z) # normalization
        res += [rhon] # tracking iteration
        if abs(rho-rhon) < tol*abs(rhon): break
        else: rho = rhon
    lmin = dot((evA(z)),z)
    res += [lmin]
    return lmin, z, res

def getdiag(A,k):
    # A in diagonal matrix format
    ind = abs(k+A.offsets).argmin()
    return A.data[ind,:]

# ----------------------------
if __name__=='__main__':
    k = 1; tol = 1e-13; maxit = 50
    errC = []; errB = []
    import matplotlib.pyplot as plt
    for n in [20, 50, 100, 200]:
        a = tile([1./n, -1., 2*(1+1./n), -1., 1./n], (n,1))
        A = spdiags(a.T,[-n/2,-1,0,1,n/2],n,n)
        C = A.tocsr() # more efficient sparse format
        lam = min(abs(arpack.eigen_symmetric(C,which='SM')[0]))
        evC = lambda x: C*x
```

```
        # inverse iteration
        invB = lambda x: spsolve(C,x)
        lmin, z, rn = pinvit(evC, invB, tol, maxit)
        err = abs(rn - lam)/lam
        errC += [err]
        txt = 'n='+str(n)
        plt.semilogy(err,'+', label='A^{-1}, '+ txt)
        print lmin, lam, err
        # preconditioned inverse iteration
        b = array([getdiag(A,-1), A.diagonal(), getdiag(A,1)])
        B = spdiags(b, [-1,0,1], n,n)
        B = B.tocsr()
        invB = lambda x: spsolve(B,x)
        lmin, z, rn = pinvit(evC, invB, tol, maxit)
        err = abs(rn - lam)/lam
        errB += [err]
        plt.semilogy(err,'o', label='B^{-1}, '+ txt)
        print lmin, lam, err

    plt.xlabel('# iteration step')
    plt.ylabel('error in approximation for λ_min')
    plt.legend()
    plt.savefig('pinvitD.eps')
    plt.show()
```

Computational effort:

1 matrix×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: $|\rho_{\mathbf{A}}(\mathbf{z}^{(k)}) - \lambda_{\min}(\mathbf{A})|$

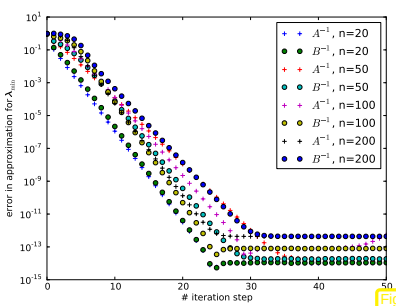


Fig. 38

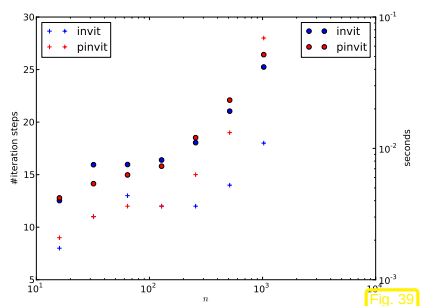


Fig. 39



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

$$\mathbf{w} = \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{A}^{-1}\mathbf{z}^{(k-1)} + (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})(\mathbf{z}^{(k-1)} - \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{A}^{-1}\mathbf{z}^{(k-1)}) \,. \tag{4.3.6}$$

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 ❷ 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 $\mathbf{D} = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$, $\lambda_j \in \mathbb{R}$, $\lambda_1 \le \lambda_2 \le \dots \le \lambda_n$ $\mathbf{U}$ orthogonal. Thus, $\mathbf{u}_j := (\mathbf{U})_{:,j}$, $j = 1, \dots, n$, are (mutually orthogonal) eigenvectors of $\mathbf{A}$.

Assume $0 \le \lambda_1 \le \dots \le \lambda_{n-2} < \lambda_{n-1} < \lambda_n$ (largest eigenvalues are simple).

If we just carry out the direct power iteration (4.3.2) for two vectors both sequencies 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 (**??**):

Num. Meth. Phys.
Gradinaru D-MATH
4.3
p. 226

Code 4.3.11: one step of subspace power iteration, $m = 2$

```
v = A*v; w = A*w
v = v/norm(v); w = w - dot(v,w)*v; w = w/norm(w)
```

Analysis through eigenvector expansions ($\mathbf{v},\mathbf{w} \in \mathbb{R}^n$, $\|\mathbf{v}\|_2 = \|\mathbf{w}\|_2 = 1$)

$$\mathbf{v} = \sum_{i=1}^{n} \alpha_j \mathbf{u}_j \quad , \quad \mathbf{w} = \sum_{i=1}^{n} \beta_j \mathbf{u}_j \ ,$$

$$\Rightarrow \quad \mathbf{A}\mathbf{v} = \sum_{i=1}^{n} \lambda_j \alpha_j \mathbf{u}_j \quad , \quad \mathbf{A}\mathbf{w} = \sum_{i=1}^{n} \lambda_j \beta_j \mathbf{u}_j \ ,$$

$$\mathbf{v}_0 := \frac{\mathbf{v}}{\|\mathbf{v}\|_2} = \Big(\sum_{i=1}^{n} \lambda_j^2 \alpha_j^2\Big)^{-1/2} \sum_{i=1}^{n} \lambda_j \alpha_j \mathbf{u}_j \ ,$$

$$\mathbf{A}\mathbf{w} - (\mathbf{v}_0^T \mathbf{A}\mathbf{w})\mathbf{v}_0 = \sum_{i=1}^{n} \left( \beta_j - \Big( \sum_{i=1}^{n} \lambda_j^2 \alpha_j \beta_j / \sum_{i=1}^{n} \lambda_j^2 \alpha_j^2 \Big) \alpha_j \right) \lambda_j \mathbf{u}_j \ .$$

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).

$$\mathbf{z} := \mathbf{A}\mathbf{w} - (\mathbf{v}_0^T \mathbf{A}\mathbf{w})\mathbf{v}_0 = 0 \cdot \mathbf{u}_n + \sum_{i=1}^{n-1} \lambda_j \beta_j \mathbf{u}_j \ ,$$

$$\mathbf{w}^{(\mathrm{new})} := \frac{\mathbf{z}}{\|\mathbf{z}\|_2} = \Big(\sum_{i=1}^{n-1} \lambda_j^2 \beta_j^2\Big)^{-1/2} \sum_{i=1}^{n-1} \lambda_j \beta_j \mathbf{u}_j \ .$$

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 $\widetilde{\mathbf{A}} := \mathbf{U}\,\mathrm{diag}(\lambda_1, \dots, \lambda_{n-1}, 0)$. 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)
```

```
Q, R = qr(mat([v,w]))
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. **??**

$$\mathbf{u}_n = \mathrm{argmax}_{\mathbf{x}\in\mathbb{R}^n} \rho_{\mathbf{A}}(\mathbf{x}) \quad , \quad \mathbf{u}_{n-1} = \mathrm{argmax}_{\mathbf{x}\in\mathbb{R}^n, \mathbf{x}\perp\mathbf{u}_n} \rho_{\mathbf{A}}(\mathbf{x}) \ . \tag{4.3.7}$$

Idea: maximize Rayleigh quotient over $\mathrm{Span}\{\mathbf{v},\mathbf{w}\}$, where $\mathbf{v}$, $\mathbf{w}$ are output by Code 4.3.10. This leads to the optimization problem

$$(\alpha^*, \beta^*) := \underset{\alpha,\beta\in\mathbb{R},\, \alpha^2+\beta^2=1}{\mathrm{argmax}} \rho_{\mathbf{A}}(\alpha\mathbf{v} + \beta\mathbf{w}) = \underset{\alpha,\beta\in\mathbb{R},\, \alpha^2+\beta^2=1}{\mathrm{argmax}} \rho_{(\mathbf{v},\mathbf{w})^T\mathbf{A}(\mathbf{v},\mathbf{w})}\left(\begin{pmatrix}\alpha\\ \beta\end{pmatrix}\right) \ . \tag{4.3.8}$$

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

$$(\mathbf{v},\mathbf{w})^T\mathbf{A}(\mathbf{v},\mathbf{w})\begin{pmatrix}\alpha\\ \beta\end{pmatrix} = \lambda\begin{pmatrix}\alpha\\ \beta\end{pmatrix} \ . \tag{4.3.9}$$

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

```
v = A*v; w = A*w; Q, R = qr(mat(v,w)); [U,D] = eig(Q.H*A*Q)
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 $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ onto $\mathrm{Span}\{\mathbf{v},\mathbf{w}\}$:

$$(\mathbf{v},\mathbf{w})^T\mathbf{A}(\mathbf{v},\mathbf{w})\begin{pmatrix}\alpha\\ \beta\end{pmatrix} = \lambda(\mathbf{v},\mathbf{w})^T(\mathbf{v},\mathbf{w})\begin{pmatrix}\alpha\\ \beta\end{pmatrix} \ .$$

More general: Ritz projection of $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ onto $\mathrm{Im}(\mathbf{V})$ (subspace spanned by columns of $\mathbf{V}$)

$$\mathbf{V}^H\mathbf{A}\mathbf{V}\mathbf{w} = \lambda\mathbf{V}^H\mathbf{V}\mathbf{w} \ . \tag{4.3.10}$$

If $\mathbf{V}$ is unitary, then this generalized eigenvalue problem will become a standard linear eigenvalue problem.

Note that he 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 (→ 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
```



*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. $\mathbf{A}$

```
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]
```

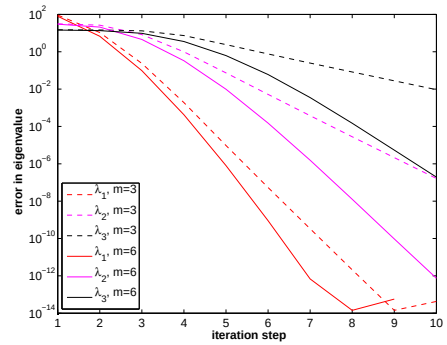
(4.3.11)

Ritz projection

Generalized normalization to $\|\mathbf{z}\|_2 = 1$

*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 = eye(n,m);
```

- Observation:
  linear convergence of eigenvalues
- choice $m > k$ boosts convergence of eigenvalues

◇

*Remark* 4.3.17 (Subspace power methods).

Analoguous to Alg. 4.3.15: construction of subspace variants of inverse iteration (→ Code 4.3.3), PINVIT (4.3.5), and Rayleigh quotient iteration (4.3.6).

△



## 4.4 Krylov Subspace Methods

All power methods (→ 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 $V := \operatorname{Span}\left\{\mathbf{v}^{(0)}, \dots, \mathbf{v}^{(k)}\right\}$
(= 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*

$$\mathcal{K}_l(\mathbf{A}, \mathbf{z}) := \operatorname{Span}\left\{\mathbf{z}, \mathbf{Az}, \dots, \mathbf{A}^{l-1}\mathbf{z}\right\} .$$

Equivalent definition: $\mathcal{K}_l(\mathbf{A},\mathbf{z}) = \{p(\mathbf{A})\mathbf{z}\colon p \text{ polynomial of degree } \leq l\}$

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 $\mathcal{K}_m(\mathbf{A},\mathbf{z})$

$$\widetilde{\mathbf{v}}_{m+1} := \mathbf{A}\mathbf{v}_m - \sum_{j=1}^{m}(\mathbf{v}_j^H\mathbf{A}\mathbf{v}_m)\,\mathbf{v}_j\,, \quad \mathbf{v}_{m+1} := \frac{\widetilde{\mathbf{v}}_{m+1}}{\|\widetilde{\mathbf{v}}_{m+1}\|_2} \quad \Rightarrow \quad \mathbf{v}_{m+1} \perp \mathcal{K}_m(\mathbf{A},\mathbf{z})\,. \tag{4.4.1}$$

(Gram-Schmidt ) orthogonal



Arnoldi process

In step $l$:

| | |
|---|---|
| $1\times$ | $\mathbf{A}\times$vector |
| $m+1$ | dot products |
| $m$ | AXPY-operations |
| $n$ | divisions |

➢ Computational cost for $m$ steps, if at most $k$ non-zero entries in each row of $\mathbf{A}$: $O(nkm^2)$

`H(m+1,m) = 0` → STOP !

Code 4.4.1: Arnoldi process

```
def arnoldi(A, v0, k):
    V = mat(zeros((v0.shape[0],k+1), dtype=complex))
    V[:,0] = v0.copy()/norm(v0)
    H = mat(zeros((k+1,k), dtype=complex))
    for m in xrange(k):
        vt = multMv(A, V[:, m])
        for j in xrange( m+1):
            H[ j, m] = (V[:, j].H * vt)[0,0]
            vt -= H[j, m] * V[:, j]
        H[m+1, m] = norm(vt);
        V[:,m+1] = vt.copy()/H[m+1, m]
    return V, H
```

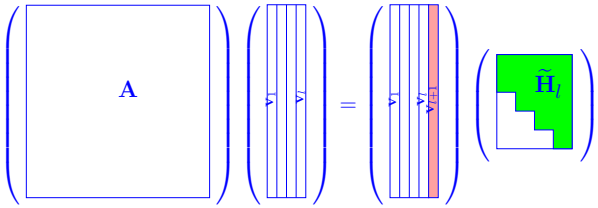
If it does not stop prematurely, the Arnoldi process of Code 4.4.0 will yield an *orthonormal basis* (OBN) of $\mathcal{K}_{k+1}(\mathbf{A},\mathbf{v}_0)$ for a **general** $\mathbf{A}\in\mathbb{C}^{n,n}$.



Algebraic view of the Arnoldi process of Code 4.4.0, meaning of output H:

$$\mathbf{V}_l = [\mathbf{v}_1,\dots,\mathbf{v}_l] \;:\; \mathbf{A}\mathbf{V}_l = \mathbf{V}_{l+1}\widetilde{\mathbf{H}}_l \;,\quad \widetilde{\mathbf{H}}_l \in \mathbb{K}^{l+1,l} \text{ mit } \widetilde{h}_{ij} = \begin{cases} \mathbf{v}_i^H\mathbf{A}\mathbf{v}_j & \text{, if } i\leq j\,, \\ \|\widetilde{\mathbf{v}}_i\|_2 & \text{, if } i=j+1\,, \\ 0 & \text{else.} \end{cases}$$

➡ $\widetilde{\mathbf{H}}_l$ = non-square upper Hessenberg matrices




Translate Code 4.4.0 to matrix calculus:

**Lemma 4.4.2** (Theory of Arnoldi process)**.**
*For the matrices $\mathbf{V}_l\in\mathbb{K}^{n,l}$, $\widetilde{\mathbf{H}}_l\in\mathbb{K}^{l+1,l}$ 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)* $\mathbf{A}\mathbf{V}_l = \mathbf{V}_{l+1}\widetilde{\mathbf{H}}_l$, $\widetilde{\mathbf{H}}_l$ *is non-square upper Hessenberg matrix,*

*(iii)* $\mathbf{V}_l^H\mathbf{A}\mathbf{V}_l = \mathbf{H}_l\in\mathbb{K}^{l,l}$, $h_{ij}=\widetilde{h}_{ij}$ *for* $1\leq i,j\leq l$,

*(iv) If* $\mathbf{A}=\mathbf{A}^H$ *then* $\mathbf{H}_l$ *is tridiagonal (➢ 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 $\mathbf{A}\mathbf{x}=\lambda\mathbf{x}$

△

Eigenvalue approximation for general EVP $\mathbf{Ax} = \lambda\mathbf{x}$ by Arnoldi process:

In $l$-th step: $\lambda_n \approx \mu_l^{(l)}$, $\lambda_{n-1} \approx \mu_{l-1}^{(l)}, \ldots, \lambda_1 \approx \mu_1^{(l)}$,

$$\sigma(\mathbf{H}_l) = \{\mu_1^{(l)}, \ldots, \mu_l^{(l)}\}, \quad |\mu_1^{(l)}| \le |\mu_2^{(l)}| \le \cdots \le |\mu_l^{(l)}| .$$

Code 4.4.3: Arnoldi eigenvalue approximation

```
import scipy.linalg
import scipy.io
from scipy.linalg import norm
from scipy import mat, eye, zeros, array, arange, sqrt, real, imag
from arnoldi import multMv, arnoldi, randomvector
from scipy.sparse.linalg.eigen.arpack import speigs
from scipy.sparse.linalg.eigen.arpack import arpack
import time

def delta(i,j):
    if i==j: return 1
    else: return 0

def ConstructMatrix(N, case='minij'):
    H = mat(zeros([N,N]))
    for i in xrange(N):
        for j in xrange(N):
            if case=='sqrts':
                H[i,j] = 1L * sqrt((i+1)**2+(j+1)**2)+(i+1)*delta(i,j)
            else:
                H[i, j] = float( 1+min(i, j) )
    return H

def sortEwEV(d,v, focus=abs):
    u = v.copy()
    #I = abs(d).argsort()
    I = focus(d).argsort()
    a = d[I]
    u = u[:,I]
    return a,u

if __name__ == "__main__":
    # construct/import matrix
    #sparseformat = True
    sparseformat = False
    if sparseformat:
        # look at http://math.nist.gov/MatrixMarket/
        L = scipy.io.mmread('SomeMatrices/qc324.mtx.gz')
    else:
        #L = scipy.io.loadmat('Lmat.mat')['L']
        L = ConstructMatrix(1000)
```



```
        #L = ConstructMatrix(400,case='sqrts')
    #print L
    #————————————————
    N = L.shape[0]
    nev = 10      # compute the largest nev eigenvalues and vectors
    print 'full matrix approach'
    t0 = time.clock()
    if sparseformat:
        d0,V0 = scipy.linalg.eig(L.todense())
    else:
        d0,V0 = scipy.linalg.eig(L)
    print time.clock() - t0 , 'sec'
    V0 = mat(V0)
    # let us check if the eigenvectors are orthogonal
    print abs(V0.H*V0 - eye(V0.shape[1])).max()
    # consider only the largest in abs.value
    a0,U0 = sortEwEV(d0,V0)
    na = U0.shape[0]
    a0 = a0[na-nev:]
    U0 = U0[:,na-nev:]
    print 'a0', a0
    #————————————————
    print 'my arnoldi procedure'
    # 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:
```

```
    t0 = time.clock()
    d2,V2 = arpack.eigen(L, k=nev, ncv=na, which='LM')
print time.clock() - t0 , 'sec'
a2,U2 = sortEwEV(d2,V2)
U2 = mat(U2)
print 'a2', a2
#-----------------
e1 = abs(a0-a1)#/abs(a0)
e2 = abs(a0-a2)#/abs(a0)
print 'error in eigenvalues:\n' , e1, '\n', e2
print 'residuals in eigenvectors:'
z0 = [norm(L*U0[:,k] - a0[k]*U0[:,k]) for k in xrange(nev) ]
print 'eig(L)', z0
z1 = [norm(L*U1[:,k] - a1[k]*U1[:,k]) for k in xrange(nev) ]
print 'arnoldi(L)', z1
z2 = [norm(L*U2[:,k] - a2[k]*U2[:,k]) for k in xrange(nev) ]
print 'arpack(L)', z2
#-----------------
# plot residuals in eigenvectors
from pylab import loglog, semilogy, plot, show, legend, xlabel,
  ylabel, savefig, rcParams#, xlim,
  ylim
plotf = loglog
#plotf = semilogy
```



```
params = {'backend': 'png',              # or 'ps'
          'axes.labelsize': 20,
          'text.fontsize': 20,
          'legend.fontsize': 20,
          'xtick.labelsize': 16,
          'lines.markersize'  : 12,
          'ytick.labelsize': 16#,
          #'text.usetex': True
          }
rcParams.update(params)
#-----------------
plotf(abs(a0), z0,'g-+')
plotf(abs(a0), z1,'r-o')
plotf(abs(a0), z2,'b-v')
legend(('eig', 'arnoldi', 'arpack'),2)
xlabel('|Eigenvalue|')
ylabel('Residual of Eigenvector')
savefig('resEVarpack.eps')
show()
#-----------------
plotf(abs(a0),e1,'ro')
plotf(abs(a0),e2,'bv')
legend(('arnoldi', 'arpack'),2)
```



```
xlabel('|Eigenvalue|')
ylabel('Error')
savefig('errEWarpack.eps')
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
($\succ$ 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.

$$\mathbf{V}_l^H\mathbf{A}\mathbf{V}_l = \begin{pmatrix} \alpha_1 & \beta_1 & & & & \\ \beta_1 & \alpha_2 & \beta_2 & & & \\ & \beta_2 & \alpha_3 & \ddots & & \\ & & \ddots & \ddots & & \\ & & & & \ddots & \\ & & & \ddots & \ddots & \beta_{k-1} \\ & & & & \beta_{k-1} & \alpha_k \end{pmatrix} =: \mathbf{T}_l \in \mathbb{K}^{k,k} \quad \text{[tridiagonal matrix]}$$

Algorithm for computing $\mathbf{V}_l$ and $\mathbf{T}_l$:

Lanczos process

Computational effort/step:

- $1\times$ $\mathbf{A}\times$vector
- $2$ dot products
- $2$ AXPY-operations
- $1$ division

Code 4.4.4: Lanczos process

```
def lanczos(A, v0, k):
    V = mat(zeros((v0.shape[0],k+1)))
    alpha = zeros(k)
    bet = zeros(k+1)
    for m in xrange(k):
        vt = multMv(A , V[ :, m])
        if m > 0: vt -= bet[m] * V[:, m-1]
        alpha[m] = (V[:, m].H * vt)[0, 0]
        vt -= alpha[m] * V[:, m]
        bet[m+1] = norm(vt)
        V[:,m+1] = vt.copy()/bet[m+1]
        rbet = bet[1:-1]
        T = diag(alpha) + diag(rbet, 1) +
          diag(rbet, -1)
    return V, T
```

Total computational effort for $l$ steps of Lanczos process, if $\mathbf{A}$ has at most $k$ non-zero entries per row: $O(nkl)$

Note: Code 4.4.3 assumes that no residual vanishes. This could happen, if $\mathbf{z}_0$ exactly belonged to the span of a few eigenvectors. However, in practical computations inevitable round-off errors will always ensure that the iterates do not stay in an invariant subspace of $\mathbf{A}$, *cf.* Rem. 4.3.2.



Convergence (what we expect from the above considerations) $\longrightarrow$ [14, Sect. 8.5])

In $l$-th step: $\lambda_n \approx \mu_l^{(l)}$, $\lambda_{n-1} \approx \mu_{l-1}^{(l)}, \dots, \lambda_1 \approx \mu_1^{(l)}$,

$$\sigma(\mathbf{T}_l) = \{\mu_1^{(l)}, \dots, \mu_l^{(l)}\}, \quad \mu_1^{(l)} \le \mu_2^{(l)} \le \dots \le \mu_l^{(l)} .$$

*Example* 4.4.5 (Lanczos process for eigenvalue computation).

$\mathbf{A}$ with eigenvalues $1, 2, \dots, 100$

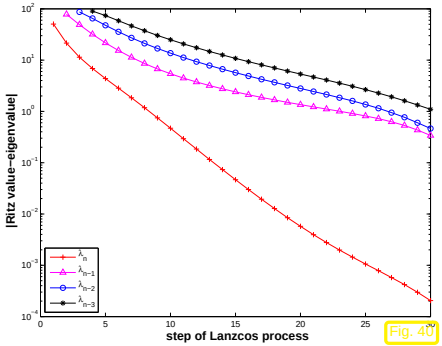



Fig. 40

`A = [min(i,j)], n=100`

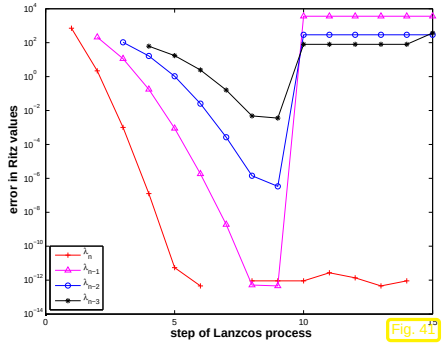



Fig. 41





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).

$\mathbf{A} \in \mathbb{R}^{10,10}$ , $a_{ij} = \min\{i,j\}$ . `A = [min(i,j)], n=10`

Computed by `V, alpha, beta = lanczos(A,n,ones(n))`, see Code 4.4.3:

$$\mathbf{T} = \begin{pmatrix}
38.500000 & 14.813845 & & & & & & & & \\
14.813845 & 9.642857 & 2.062955 & & & & & & & \\
 & 2.062955 & 2.720779 & 0.776284 & & & & & & \\
 & & 0.776284 & 1.336364 & 0.385013 & & & & & \\
 & & & 0.385013 & 0.826316 & 0.215431 & & & & \\
 & & & & 0.215431 & 0.582380 & 0.126781 & & & \\
 & & & & & 0.126781 & 0.446860 & 0.074650 & & \\
 & & & & & & 0.074650 & 0.363803 & 0.043121 & \\
 & & & & & & & 0.043121 & 3.820888 & 11.991094 \\
 & & & & & & & & 11.991094 & 41.254286
\end{pmatrix}$$

$$\sigma(\mathbf{A}) = \{0.255680, 0.273787, 0.307979, 0.366209, 0.465233, 0.643104, 1.000000, 1.873023, 5.048917, 44.766069\}$$
$$\sigma(\mathbf{T}) = \{0.263867, 0.303001, 0.365376, 0.465199, 0.643104, 1.000000, 1.873023, 5.048917, 44.765976, 44.766069\}$$

Uncanny cluster of computed eigenvalues of $\mathbf{T}$ ("ghost eigenvalues", [20, Sect. 9.2.5])

$$\mathbf{V}^H\mathbf{V} = \begin{pmatrix}
1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000251 & 0.258801 & 0.883711 \\
0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000106 & 0.109470 & 0.373799 \\
0.000000 & -0.000000 & 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000005 & 0.005373 & 0.018347 \\
0.000000 & 0.000000 & 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000096 & 0.000328 \\
0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000001 & 0.000003 \\
0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 \\
0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 \\
0.000251 & 0.000106 & 0.000005 & 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 \\
0.258801 & 0.109470 & 0.005373 & 0.000096 & 0.000001 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & 0.000000 \\
0.883711 & 0.373799 & 0.018347 & 0.000328 & 0.000003 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000
\end{pmatrix},$$

Loss of orthogonality of residual vectors due to roundoff
(compare: impact of roundoff on CG iteration



| $l$ | $\sigma(\mathbf{T}_l)$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

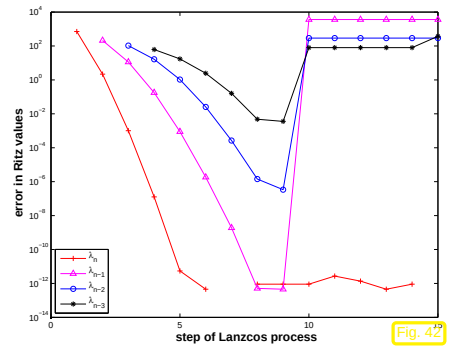
◇

*Example* 4.4.7 (Stabilty of Arnoldi process).

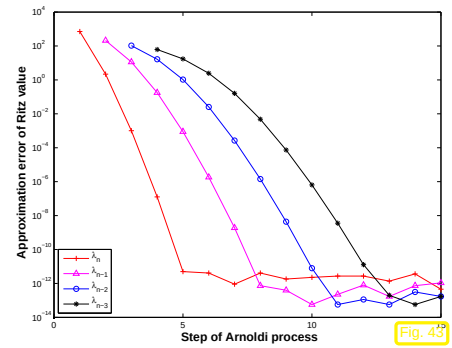
$\mathbf{A} \in \mathbb{R}^{100,100}$ , $a_{ij} = \min\{i,j\}$ . `A = [min(i,j)], n=100`








Lanczos process: Ritz values




Arnoldi process: Ritz values

Ritz values during Arnoldi process for `A = [min(i,j)], n =10);` ↔ Ex. 4.4.5

| $l$ | $\sigma(\mathbf{H}_l)$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | | | | 38.500000 |
| 2 | | | | | | | | | 3.392123 | 44.750734 |
| 3 | | | | | | | | 1.117692 | 4.979881 | 44.766064 |
| 4 | | | | | | | 0.597664 | 1.788008 | 5.048259 | 44.766069 |
| 5 | | | | | | 0.415715 | 0.925441 | 1.870175 | 5.048916 | 44.766069 |
| 6 | | | | | 0.336507 | 0.588906 | 0.995299 | 1.872997 | 5.048917 | 44.766069 |
| 7 | | | | 0.297303 | 0.431779 | 0.638542 | 0.999922 | 1.873023 | 5.048917 | 44.766069 |
| 8 | | | 0.276159 | 0.349722 | 0.462449 | 0.643016 | 1.000000 | 1.873023 | 5.048917 | 44.766069 |
| 9 | | 0.263872 | 0.303009 | 0.365379 | 0.465199 | 0.643104 | 1.000000 | 1.873023 | 5.048917 | 44.766069 |
| 10 | 0.255680 | 0.273787 | 0.307979 | 0.366209 | 0.465233 | 0.643104 | 1.000000 | 1.873023 | 5.048917 | 44.766069 |

Observation: (almost perfect approximation of spectrum of $\mathbf{A}$)

For the above examples both the Arnoldi process and the Lanczos process are *algebraically equivalent*, because they are applied to a symmetric matrix $\mathbf{A} = \mathbf{A}^T$. However, they behave strikingly differently, which indicates that they are *not numerically equivalent*.

The Arnoldi process is much less affected by roundoff than the Lanczos process, because it does not take for granted orthogonality of the "residual vector sequence". Hence, the Arnoldi process enjoys superior numerical stability compared to the Lanczos process.

◇

*Example* 4.4.8 (Eigenvalue computation with Arnoldi process).

Eigenvalue approximation from Arnoldi process for *non-symmetric* $\mathbf{A}$, initial vector `ones(100);`

```
n=100
M = 2.*eye(n) + -0.5*eye(n,k=-1) + 1.5*eye(n, k=1); M = mat(M)
A = M*(diag(r_[1:n+1]))*M.I
```

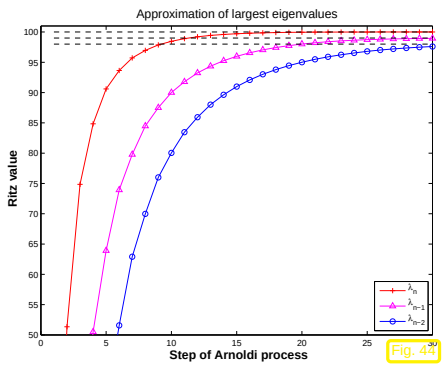


Fig. 44

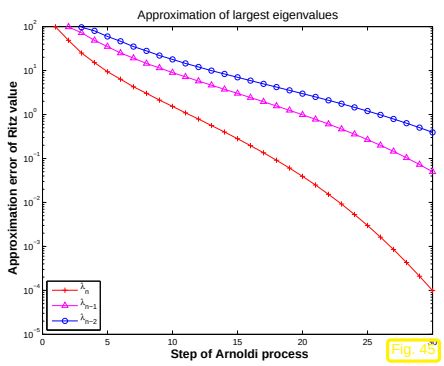


Fig. 45

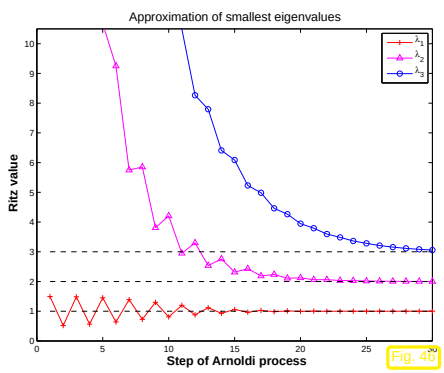


Fig. 46

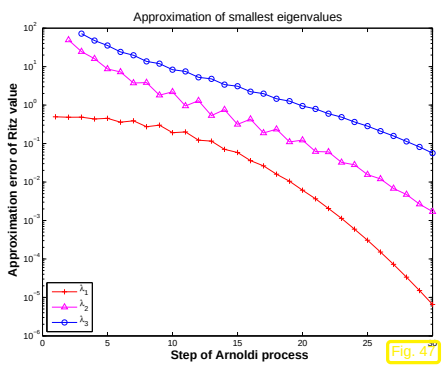


Fig. 47

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 "$\mathbf{B}$-inner product" $(\mathbf{x},\mathbf{y}) \mapsto \mathbf{x}^H\mathbf{By}$.

△

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

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)` : `Afun` = handle to function providing matrix×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 `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, Rayleight quotient iteration and preconditioning
- what are Krylov methods, when and how to use them

# Part II

# Interpolation and Approximation

## Introduction

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

$$\boxed{f(\mathbf{x}_i) = \mathbf{y}_i, \quad i = 1, \dots, m}$$

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: $t$, $y \mathrel{\hat{=}}$ 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 $\rho$ |
| magnetic field $H$ | magnetic flux $B$ |
| ... | ... |

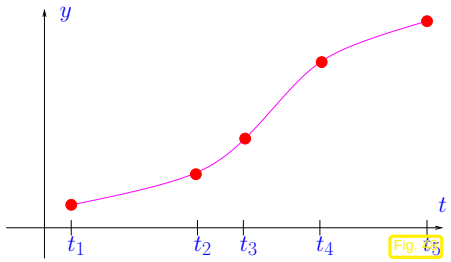


Fig. 5

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=feval(x)`)

Goal: Find a "simple"$^{(*)}$ function $\widetilde{\mathbf{f}} : D \mapsto \mathbb{R}^d$ such that the difference $\mathbf{f} - \widetilde{\mathbf{f}}$ is "small"$^{(\clubsuit)}$





$^{(*)}$: "simple" $\sim$ described by small amount of information, easy to evaluate (e.g, polynomial or piecewise polynomial $\widetilde{\mathbf{f}}$)

$^{(\clubsuit)}$ "small" $\sim$ $\left\|\mathbf{f} - \widetilde{\mathbf{f}}\right\|$ small for some norm $\|\cdot\|$ on space $C(D)$ of continous 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).

$$f \in C^k(I), \quad I \text{ interval}, \quad k \in \mathbb{N}, \qquad T_k(t) := \frac{f^{(k)}(t_0)}{k!}\,(t - t_0)^k, \quad t_0 \in I\,.$$

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

$$\mathbf{f} \xrightarrow{\text{sampling}} (\mathbf{x}_i, \mathbf{y}_i := f(\mathbf{x}_i))_{i=1}^m \xrightarrow{\text{interpolation}} \widetilde{\mathbf{f}}: \quad \widetilde{\mathbf{f}}(\mathbf{x}_i) = \mathbf{y}_i\,.$$

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 $V_n := \operatorname{Span}\{b_1, \dots, b_n\}$.

$b_j(t) = t^{j-1}$ 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 $\leq k$, $k \in \mathbb{N}$:

$$\mathcal{P}_k := \{t \mapsto \alpha_k t^k + \alpha_{k-1} t^{k-1} + \dots + \alpha_1 t + \alpha_0 \, , \, \alpha_j \in \mathbb{K}\} \, . \tag{5.1.1}$$

Terminology: the functions $t \mapsto t^k$, $k \in \mathbb{N}_0$, are called monomials

$t \mapsto \alpha_k t^k + \alpha_{k-1} t^{k-1} + \dots + \alpha_0$ = 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)**.**

$$\dim \mathcal{P}_k = k + 1 \quad \textit{and} \quad \mathcal{P}_k \subset C^\infty(\mathbb{R}).$$

*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: $\alpha_k t^k + \alpha_{k-1} t^{k-1} + \dots + \alpha_0$ → 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

$$p(t) = (t \cdots t(t(\alpha_n t + \alpha_{n-1}) + \alpha_{n-2}) + \dots + \alpha_1) + \alpha_0 \, . \tag{5.1.2}$$

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 $\mathcal{P}_n$

$$N_0(t) := 1\,, \quad N_1(t) := (t-t_0)\,, \quad \dots \quad , \quad N_n(t) := \prod_{i=0}^{n-1}(t-t_i)\,. \tag{5.2.1}$$

Note: $N_n \in \mathcal{P}_n$ with *leading coefficient* $1$.

➢ LSE for polynomial interpolation problem in Newton basis:

$$a_j \in \mathbb{R}: \quad a_0N_0(t_j) + a_1N_1(t_j) + \cdots + a_nN_n(t_j) = y_j, \quad j = 0,\dots,n\,.$$

⇔ triangular linear system

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 1 & (t_1-t_0) & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 1 & (t_n-t_0) & \cdots & \prod_{i=0}^{n-1}(t_n-t_i) \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}\,.$$

Solution of the system with forward substitution:

$$a_0 = y_0\,,$$



$$a_1 = \frac{y_1-a_0}{t_1-t_0} = \frac{y_1-y_0}{t_1-t_0}\,,$$

$$a_2 = \frac{y_2-a_0-(t_2-t_0)a_1}{(t_2-t_0)(t_2-t_1)} = \frac{y_2-y_0-(t_2-t_0)\frac{y_1-y_0}{t_1-t_0}}{(t_2-t_0)(t_2-t_1)} = \frac{\dfrac{y_2-y_0}{t_2-t_0}-\dfrac{y_1-y_0}{t_1-t_0}}{t_2-t_1}\,,$$

$$\vdots$$

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

⇒ $a_j$ is the leading coefficient of the interpolating polynomial $p_{0,\dots,j}$

➢ Recursion for leading coefficients $a_{\ell,\dots,m}$ of interpolating polynomials $p_{\ell,\dots,m}$, $0 \le \ell \le m \le n$:

$$a_{\ell,\dots,m} = \frac{a_{\ell+1,\dots,m} - a_{\ell,\dots,m-1}}{t_m - t_\ell}\,.$$



Simpler and more efficient algorithm using divided differences:

$$\begin{aligned} y[t_i] &= y_i \\ y[t_i,\dots,t_{i+k}] &= \frac{y[t_{i+1},\dots,t_{i+k}] - y[t_i,\dots,t_{i+k-1}]}{t_{i+k}-t_i} \quad \text{(recursion)} \end{aligned} \tag{5.2.2}$$

Recursive calculation by divided differences scheme

$$\begin{array}{c|ccccc} t_0 & y[t_0] & & & & \\ & & > & y[t_0,t_1] & & \\ t_1 & y[t_1] & & & > & y[t_0,t_1,t_2] \\ & & > & y[t_1,t_2] & & \\ t_2 & y[t_2] & & & > & y[t_1,t_2,t_3] \\ & & > & y[t_2,t_3] & & \\ t_3 & y[t_3] & & & & \end{array} \quad > \; y[t_0,t_1,t_2,t_3], \tag{5.2.3}$$

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

```
def divdiff(t, y):
    n = y.shape[0] - 1
    if n > 0:
        y[0:n] = divdiff(t[0:n], y[0:n])
        for j in xrange(0,n):
            y[n] = (y[n] - y[j]) / (t[n] - t[j])
    return y
```

Code 5.2.2: Divided differences, non-recursive implementation, in situ computation

```
def divdiff(t, y):
    n = y.shape[0] - 1
    for l in xrange(0,n):
        for j in xrange(l+1,n+1):
            y[j] = (y[j] - y[l]) / (t[j] - t[l])
    return y
```

By derivation: computed finite differences are the coefficients of interpolating polynomials in Newton basis:

$$p(t) = a_0 + a_1(t-t_0) + a_2(t-t_0)(t-t_1) + \cdots + a_n \prod_{j=0}^{n-1}(t-t_j) \tag{5.2.4}$$

$$a_0 = y[t_0],\ a_1 = y[t_0,t_1],\ a_2 = y[t_0,t_1,t_2],\ \ldots.$$

"Backward evaluation" of $p(t)$ in the spirit of Horner's scheme (→ Rem. 5.1.2, [10, Alg. 8.20]):

$$p \leftarrow a_n, \quad p \leftarrow (t-t_{n-1})p + a_{n-1}, \quad p \leftarrow (t-t_{n-2})p + a_{n-2}, \quad \ldots.$$

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 $y_0,\ldots,y_n$ are the values of a smooth function $f$ in the points $t_0,\ldots,t_n$, that is, $y_j := f(t_j)$, then

$$y[t_i,\ldots,t_{i+k}] = \frac{f^{(k)}(\xi)}{k!}$$

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 (→ 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 $\{\mathcal{T}_j\}_{j=1}^{\infty}$, $\mathcal{T}_j = \{x_1^{(j)},\ldots,x_j^{(j)}\} \subset [a,b]$, $a \le x_1^{(j)} < x_2^{(j)} < \cdots < x_j^{(j)} \le b$, 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 $j \to \infty$. △

We consider Lagrangian polynomial interpolation on node set

$$\mathcal{T} := \{t_0,\ldots,t_n\} \subset I,\ I \subset \mathbb{R}, \text{ interval of length } |I|.$$

Notation: For a continuous function $f : I \mapsto \mathbb{K}$ we define the polynomial interpolation operator

$$\mathsf{I}_{\mathcal{T}}(f) := \mathsf{I}_{\mathcal{T}}(\mathbf{y}) \in \mathcal{P}_n \quad \text{with} \quad \mathbf{y} := (f(t_0),\ldots,f(t_n))^T \in \mathbb{K}^{n+1}.$$

Goal: estimate of the interpolation error norm $\|f - \mathsf{I}_{\mathcal{T}} f\|$ (for some norm on $C(I)$).

Focus: asymptotic behavior of interpolation error for $n \to \infty$

*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 $p := \mathsf{I}_{\mathcal{T}} f \in \mathcal{P}_n$.

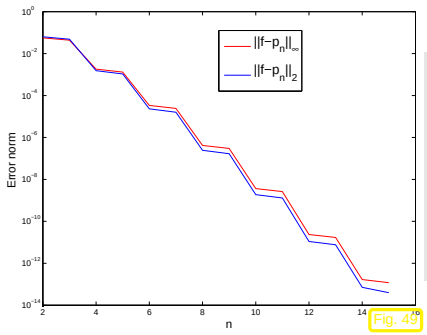



Fig. 49

computer-experiment: computation of the norms.

- $L^\infty$-norm: sampling on a grid of meshsize $\pi/1000$.
- $L^2$-norm: numerical quadrature (→ Chapter 7) with trapezoidal rule (7.2.2) on a grid of meshsize $\pi/1000$.

◇

In the previous experiment we observed a clearly visible qualitative behavior of $\|f - \mathsf{I}_{\mathcal{T}} f\|$ as we increased the polynomial degree $n$. The prediction of the decay law for $\|f - \mathsf{I}_{\mathcal{T}} f\|$ 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 $\|f - \mathsf{I}_{\mathcal{T}} f\|$ as a function of the polynomial degree $n$:

$$\exists\, C \neq C(n) > 0: \quad \|f - \mathsf{I}_{\mathcal{T}} f\| \leq C\, T(n) \quad \text{for } n \to \infty\,. \tag{5.3.1}$$

Classification (best bound for $T(n)$):

$\exists\, p > 0$: $T(n) \leq n^{-p}$ : algebraic convergence, with rate $p > 0$ ,
$\exists\, 0 < q < 1$: $T(n) \leq q^n$ : exponential convergence , $\forall n \in \mathbb{N}$ .

Convergence behavior of interpolation error is often expressed by means of the Landau-O-notation:

Algebraic convergence: $\|f - \mathsf{I}_{\mathcal{T}} f\| = O(n^{-p})$
Exponential convergence: $\|f - \mathsf{I}_{\mathcal{T}} f\| = O(q^n)$
for $n \to \infty$ ("asymptotic!")

*Remark* 5.3.3 (Exploring convergence).

Given: pairs $(n_i, \epsilon_i)$, $i = 1, 2, 3, \ldots$, $n_i \mathrel{\hat{=}}$ polynomial degrees, $\epsilon_i \mathrel{\hat{=}}$ norms of interpolation error



❶ Conjectured: algebraic convergence: $\epsilon_i \approx C n^{-p}$

$$\log(\epsilon_i) \approx \log(C) - p \log n_i \qquad \text{(affine linear in log-log scale).}$$

Apply linear regression (`numpy.polyfit`) to points $(\log n_i, \log \epsilon_i)$ ➢ estimate for rate $p$.

❶ Conjectured: exponential convergence: $\epsilon_i \approx C \exp(-\beta n_i)$

$$\log \epsilon_i \approx \log(C) - \beta n_i \qquad \text{(affine linear in lin-log scale). .}$$

Apply linear regression (Ex. 3.0.1, `numpy.polyfit`) to points $(n_i, \log \epsilon_i)$ ➢ estimate for $q := \exp(-\beta)$.

☞ Fig. 49: we suspect exponential convergence in Ex. 5.3.2.

△

Beware: same concept ↔ different meanings:

- convergence of a sequence (e.g. of iterates $\boldsymbol{x}^{(k)}$ → Sect. 1.1 )
- convergence of an approximation (dependent on an approximation parameter, e.g. $n$)

*Example* 5.3.4 (Runge's example). → Ex. **??**

Polynomial interpolation of $f(t) = \frac{1}{1+t^2}$ with equispaced nodes:

$$\mathcal{T} := \left\{ t_j := -5 + \tfrac{10}{n} j \right\}_{j=0}^{n}\,, \quad y_j = \frac{1}{1+t_j^2}\,.j = 0, \ldots, n\,.$$

Code 5.3.5: Computing the interpolation error for Runge's example

```
# Interpolation error plot for Runge's example

from numpy import linspace, array, polyfit, polyval, max, abs, hstack,
  vstack
from matplotlib.pyplot import *

# Exact values
x = linspace(-5, 5, 1001)
f = lambda x: 1.0 / (1.0 + x**2)

fv = f(x)

# Compute approximation of increasing degree
err = []

for d in xrange(1, 21):
    t = linspace(-5, 5, d+1)
```

```
    p = polyfit(t, f(t), d)
    y = polyval(p, x)

    err.append( hstack([d, max(abs(y-fv))]) )

err = array(err)

# Plot
figure()
semilogy(err[:,0], err[:,1], "r--+")
xlabel(r"Degree $d$")
ylabel(r"Interpolation error (maximum norm)")
savefig("../PICTURES/rungeerrmax.eps")
```

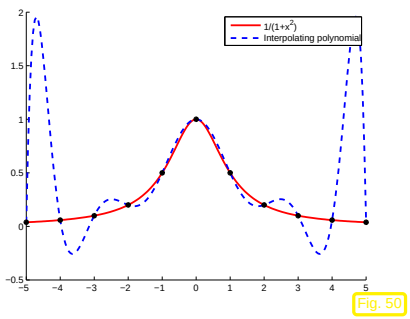



Fig. 50

Interpolating polynomial, $n = 10$

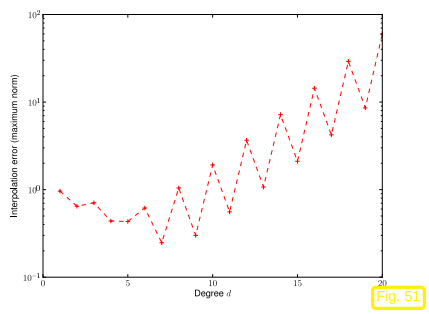



Fig. 51

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 $\mathsf{I}_{\mathcal{T}} f$ near the endpoints of the interval:

$$\|f - \mathsf{I}_{\mathcal{T}} f\|_{L^\infty(]-5,5[)} \xrightarrow{n\to\infty} \infty \,.$$

◇



**Theorem 5.3.1** (Representation of interpolation error). *[10, Thm. 8.22], [29, Thm. 37.4]*
$f \in C^{n+1}(I)$: $\forall\, t \in I$: $\exists\, \tau_t \in ]\min\{t, t_0, \dots, t_n\}, \max\{t, t_0, \dots, t_n\}[$:

$$f(t) - \mathsf{I}_{\mathcal{T}}(f)(t) = \frac{f^{(n+1)}(\tau_t)}{(n+1)!} \cdot \prod_{j=0}^{n} (t - t_j) \,. \tag{5.3.2}$$

*Proof.* Write $q(t) := \prod_{j=0}^{n} (t - t_j) \in \mathcal{P}_{n+1}$ and fix $t \in I$.

$$t \neq t_j \;\Rightarrow\; q(t) \neq 0 \;\Rightarrow\; \exists c(t) \in \mathbb{R}\colon\; f(t) - \mathsf{I}_{\mathcal{T}}(f)(t) = cq(t)$$

$\varphi(x) := f(x) - \mathsf{I}_{\mathcal{T}}(f)(x) - cq(x)$ 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

$$\varphi^{(m)} \text{ has } n + 2 - m \text{ distinct zeros in } I.$$

$$\Rightarrow \quad \exists \tau_t \in I\colon\; \varphi^{(n+1)}(x) = f^{(n+1)}(\tau_t) - c(n+1)! = 0 \,.$$

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* $f \in C^{n+1}(I)$: $\forall\, t \in I$:

$$f(t) - \mathsf{I}_{\mathcal{T}}(f)(t) = \int_0^1 \int_0^{\tau_1} \cdots \int_0^{\tau_{n-1}} \int_0^{\tau_n} f^{(n+1)}(t_0 + \tau_1(t_1 - t_0) + \cdots$$
$$+ \tau_n(t_n - t_{n-1}) + \tau(t - t_n))\, \mathrm{d}\tau \mathrm{d}\tau_n \cdots \mathrm{d}\tau_1 \cdot \prod_{j=0}^{n} (t - t_j) \,.$$

*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 (**??**).

△



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:

Thm. 5.3.1 $\Rightarrow$ $$\|f - \mathsf{I}_{\mathcal{T}} f\|_{L^\infty(I)} \leq \frac{\left\|f^{(n+1)}\right\|_{L^\infty(I)}}{(n+1)!} \max_{t\in I} |(t-t_0)\cdot\cdots\cdot(t-t_n)| \ . \qquad (5.3.3)$$

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)

$$\left\|f^{(k)}\right\|_{L^\infty(I)} \leq 1 \ , \quad \forall k \in \mathbb{N}_0 \quad \Rightarrow \quad \begin{aligned} \|f-p\|_{L^\infty(I)} &\leq \frac{1}{(1+n)!} \max_{t\in I} \left|(t-0)(t-\tfrac{\pi}{n})(t-\tfrac{2\pi}{n})\cdot\cdots\cdot(t-\pi)\right| \\ &\leq \frac{1}{n+1}\left(\frac{\pi}{n}\right)^{n+1} \ . \end{aligned}$$



➜ 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 $n \to \infty$.

➜ Possible blow-up of error bound from Thm. 5.3.1 $\to \infty$ for $n \to \infty$. ◇



*Remark* 5.3.9 ($L^2$-error estimates for polynomial interpolation).

Thm. 5.3.1 gives error estimates for the $L^\infty$-Norm. And the other norms?

From Lemma. 5.3.2 using Cauchy-Schwarz inequality:

$$\|f - \mathsf{I}_{\mathcal{T}}(f)\|^2_{L^2(I)} = \int_I \left| \int_0^1 \int_0^{\tau_1} \cdots \int_0^{\tau_{n-1}} \int_0^{\tau_n} f^{(n+1)}(\ldots)\,\mathrm{d}\tau\mathrm{d}\tau_n \cdots \mathrm{d}\tau_1 \cdot \underbrace{\prod_{j=0}^{n}(t-t_j)}_{|t-t_j|<|I|} \right|^2 \mathrm{d}t$$





$$\leq \int_I |I|^{2n+2} \underbrace{\mathrm{vol}_{(n+1)}(S_{n+1})}_{=1/(n+1)!} \int_{S_{n+1}} |f^{(n+1)}(\ldots)|^2 \,\mathrm{d}\boldsymbol{\tau}\,\mathrm{d}t$$

$$= \int_I \frac{|I|^{2n+2}}{(n+1)!} \int_I \underbrace{\mathrm{vol}_{(n)}(C_{t,\tau})}_{\leq 2^{(n-1)/2}/n!} |f^{(n+1)}(\tau)|^2 \,\mathrm{d}\tau\mathrm{d}t \ ,$$

$$S_{n+1} := \{\mathbf{x} \in \mathbb{R}^{n+1} \colon 0 \leq x_n \leq x_{n-1} \leq \cdots \leq x_1 \leq 1\} \quad \text{(unit simplex)} \ ,$$
$$C_{t,\tau} := \{\mathbf{x} \in S_{n+1} \colon t_0 + x_1(t_1-t_0) + \cdots + x_n(t_n - t_{n-1}) + x_{n+1}(t-t_n) = \tau\} \ .$$

This gives the bound for the $L^2$-norm of the error:

$$\Rightarrow \quad \|f - \mathsf{I}_{\mathcal{T}}(f)\|_{L^2(I)} \leq \frac{2^{(n-1)/4}|I|^{n+1}}{\sqrt{(n+1)!n!}} \left(\int_I |f^{(n+1)}(\tau)|^2 \,\mathrm{d}\tau\right)^{1/2} . \qquad (5.3.4)$$

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 ($\longrightarrow$ 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: $\mathcal{T} := \{t_0 < t_1 < \cdots < t_{n-1} < t_n\}$, $n \in \mathbb{N}$,
function $f : I \to \mathbb{R}$ continuous; without loss of generality $I = [-1,1]$.

Thm. 5.3.1: $$\|f-p\|_{L^\infty(I)} \leq \frac{1}{(n+1)!} \left\|f^{(n+1)}\right\|_{L^\infty(I)} \|w\|_{L^\infty(I)} \ ,$$
$$w(t) := (t-t_0)\cdot\cdots\cdot(t-t_n) \ .$$

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 $q$ ➔ $|q(t^*)| = \|q\|_{L^\infty(I)}$,
- $q$ has $n+1$ zeros in $I$,
- $|q(-1)| = |q(1)| = \|q\|_{L^\infty(I)}$.

**Definition 5.4.1** (Chebychev polynomial).
*The* $n^{\text{th}}$ *Chebychev polynomial is* $T_n(t) := \cos(n \arccos t)$, $\quad -1 \le t \le 1$.

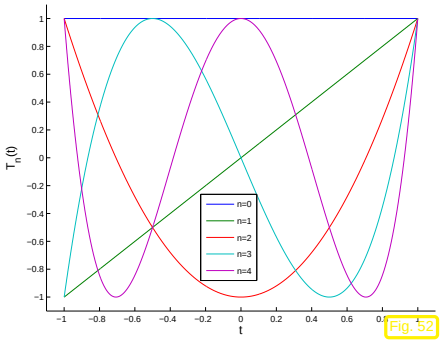


Fig. 52

Chebychev polynomials $T_0, \dots, T_4$

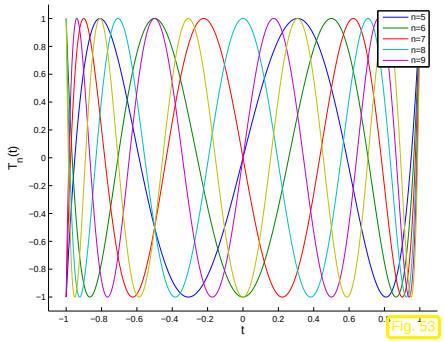


Fig. 53

Chebychev polynomials $T_5, \dots, T_9$

Zeros of $T_n$: $\quad t_k = \cos\left(\frac{2k-1}{2n}\pi\right), \quad k = 1, \dots, n$. (5.4.1)

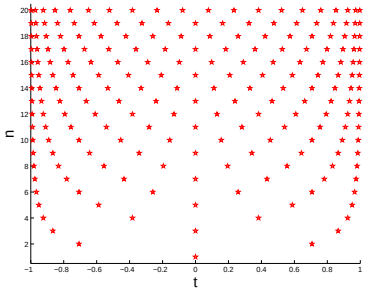
Extrema (alternating signs) of $T_n$:

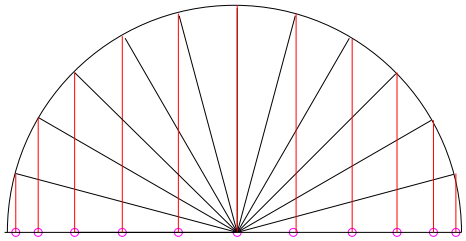
$$|T_n(\bar{t}_k)| = 1 \Leftrightarrow \exists\, k = 0, \dots, n: \quad \bar{t}_k = \cos\frac{k\pi}{n}, \qquad \|T_n\|_{L^\infty([-1,1])} = 1 .$$

Chebychev 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 $\cos x = t$:

$$T_{n+1}(t) = 2t\,T_n(t) - T_{n-1}(t) \quad , \quad T_0 \equiv 1 \, , \quad T_1(t) = t \, , \quad n \in \mathbb{N} \, . \tag{5.4.2}$$

This implies:
- $T_n \in \mathcal{P}_n$,
- leading coefficients equal to $2^{n-1}$,
- $T_n$ linearly independent,
- $T_n$ basis of $\mathcal{P}_n = \operatorname{Span}\{T_0, \dots, T_n\}$, $n \in \mathbb{N}_0$.

△





**Theorem 5.4.2** (Minimax property of the Chebychev polynomials).

$$\|T_n\|_{L^\infty([-1,1])} = \inf\{\|p\|_{L^\infty([-1,1])} : p \in \mathcal{P}_n, p(t) = 2^{n-1}t^n + \cdots\} \, , \quad \forall n \in \mathbb{N} \, .$$

*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

$\Rightarrow \quad T_n - q$ 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 $\mathcal{T} = \left\{ \cos\left(\frac{2k+1}{2(n+1)}\pi\right) \;,\; k = 0,\dots,n \right\}$,
- $w(t) = (t-t_0)\cdots(t-t_{n+1}) = 2^{-n}T_{n+1}(t) \quad , \quad \|w\|_{L^\infty(I)} = 2^{-n}$,

  with leading coefficient $1$.

Then, by Thm. 5.3.1,

$$\|f - \mathsf{I}_{\mathcal{T}}(f)\|_{L^\infty([-1,1])} \leq \frac{2^{-n}}{(n+1)!} \left\| f^{(n+1)} \right\|_{L^\infty([-1,1])} \;. \tag{5.4.3}$$

*Remark* 5.4.2 (Chebychev polynomials on arbitrary interval).

How to use Chebychev polynomial interpolation on an arbitrary interval?

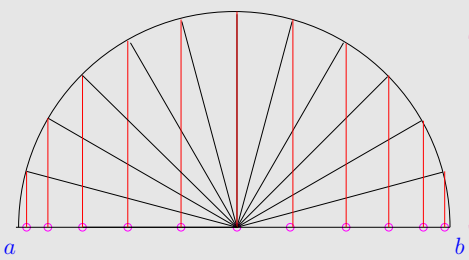
Scaling argument: interval transformation requires the transport of the functions

$$[-1,1] \xrightarrow{\widehat{t} \mapsto t := a + \frac{1}{2}(\widehat{t}+1)(b-a)} [a,b] \quad \leftrightarrow \quad \widehat{f}(\widehat{t}) := f(t) \;.$$

$$p \in \mathcal{P}_n \;\wedge\; p(t_j) = f(t_j) \quad \Leftrightarrow \quad \widehat{p} \in \mathcal{P}_n \;\wedge\; \widehat{p}(\widehat{t}_j) = \widehat{f}(\widehat{t}_j) \;.$$

$$\frac{\mathrm{d}^n \widehat{f}}{\mathrm{d}\widehat{t}^n}(\widehat{t}) = (\tfrac{1}{2}|I|)^n \frac{\mathrm{d}^n f}{\mathrm{d}t^n}(t)$$

$$\begin{aligned} \|f - \mathsf{I}_{\mathcal{T}}(f)\|_{L^\infty(I)} &= \left\| \widehat{f} - \mathsf{I}_{\widehat{\mathcal{T}}}(\widehat{f}) \right\|_{L^\infty([-1,1])} \leq \frac{2^{-n}}{(n+1)!} \left\| \frac{\mathrm{d}^{n+1}\widehat{f}}{\mathrm{d}\widehat{t}^{n+1}} \right\|_{L^\infty([-1,1])} \\ &\leq \frac{2^{-2n-1}}{(n+1)!} |I|^{n+1} \left\| f^{(n+1)} \right\|_{L^\infty(I)} \;. \end{aligned} \tag{5.4.4}$$




The Chebychev nodes in the interval $I = [a,b]$ are

$$t_k := a + \tfrac{1}{2}(b-a)\left(\cos\left(\frac{2k+1}{2(n+1)}\pi\right) + 1\right) \;, \tag{5.4.5}$$

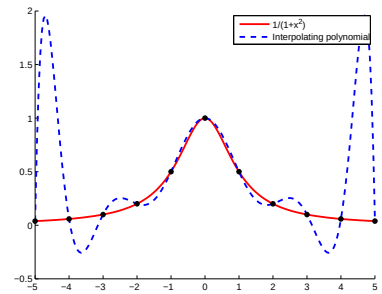
$$k = 0,\dots,n \;.$$

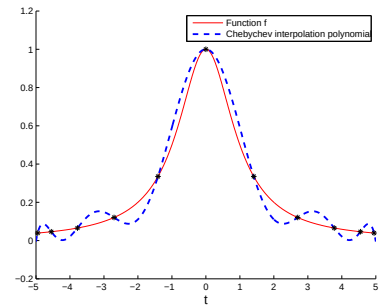


### 5.4.2 Chebychev interpolation error estimates

*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:



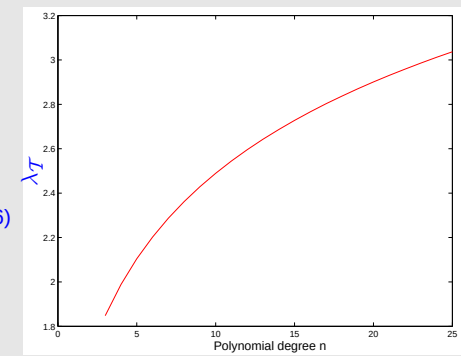

◇

*Remark* 5.4.4 (Lebesgue Constant for Chebychev nodes).

Theory [7, 56, 55]:

$$\lambda_{\mathcal{T}} \sim \frac{2}{\pi}\log(1+n) + o(1) \;,$$

$$\boxed{\lambda_{\mathcal{T}} \leq \tfrac{2}{\pi}\log(1+n) + 1} \;. \tag{5.4.6}$$




△

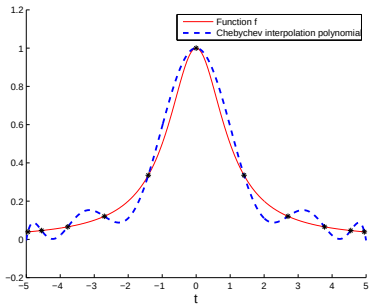
*Example* 5.4.5 (Chebychev interpolation error).

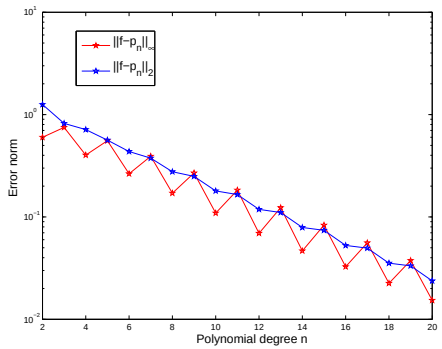
For $I = [a,b]$ let $x_l := a + \frac{b-a}{N}l,\; l = 0,\dots,N,\; N = 1000$ we approximate the norms of the error

$$\|f - p\|_\infty \approx \max_{0 \leq l \leq N} |f(x_l) - p(x_l)|$$

$$\|f-p\|_2^2 \approx \frac{b-a}{2N} \sum_{0\le l<N} \left( |f(x_l)-p(x_l)|^2 + |f(x_{l+1})-p(x_{l+1})|^2 \right)$$

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).



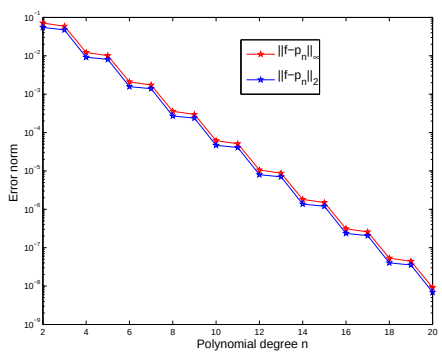

Notice: exponential convergence of the Chebychev interpolation:

$$p_n \to f, \quad \|f - \mathsf{I}_n f\|_{L^2([-5,5])} \approx 0.8^n$$

.

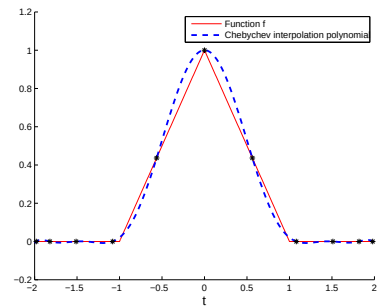
Now: the same function $f(t) = (1+t^2)^{-1}$
on a smaller interval $I = [-1,1]$.
(Faster) exponential convergence:

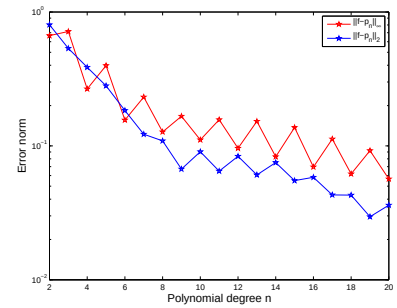
$$\|f - \mathsf{I}_n f\|_{L^2([-1,1])} \approx 0.42^n.$$




② $f(t) = \max\{1-|t|, 0\}$, $I = [-2,2]$, $n = 10$ nodes (plot on the left).
$f \in C^0(I)$ but $f \notin C^1(I)$.

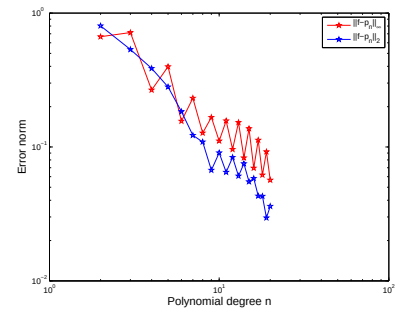




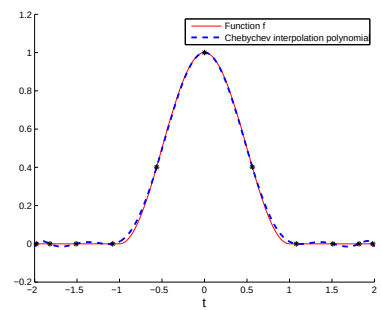



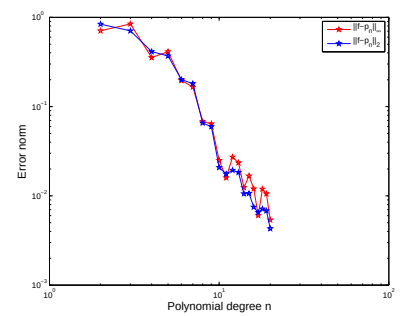

From the double logarithmic plot, notice ➜

- no exponential convergence
- algebraic convergence (**?**)




③ $f(t) = \begin{cases} \frac{1}{2}(1+\cos \pi t) & |t| < 1 \\ 0 & 1 \le |t| \le 2 \end{cases}$ $I = [-2,2]$, $n = 10$ (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*

$$\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)\frac{1}{\sqrt{1-x^2}}dx \ . \tag{5.4.7}$$

**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:*

$$(f, g) = \sum_{k=0}^{n} f(x_k)g(x_k) \ , \tag{5.4.8}$$

*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:*

$$p(x) = \frac{1}{2}c_0 + c_1T_1(x) + \dots + c_nT_n(x) \ , \tag{5.4.9}$$

*with*

$$c_k = \frac{2}{n+1}\sum_{l=0}^{n} f\left(\cos\left(\frac{2l+1}{n+1}\cdot\frac{\pi}{2}\right)\right)\cdot\cos\left(k\frac{2l+1}{n+1}\cdot\frac{\pi}{2}\right) \ . \tag{5.4.10}$$



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*

$$\begin{aligned} d_{n+2} &= d_{n+1} = 0 \\ d_k &= c_k + (2x)\cdot d_{k+1} - d_{k+2} \qquad \text{for } k = n, n-1, \dots, 0. \end{aligned} \tag{5.4.11}$$

*Then* $\quad p(x) = \frac{1}{2}(d_0 - d_2)$.

Code 5.4.6: Clenshaw algorithm

```
def clenshaw(a,x):
    # Degree of polynomial
    n = a.shape[0] - 1
    d = tile( reshape(a,n+1,1), (x.shape[0], 1) )
    d = d.T
    for j in xrange(n, 1, -1):
        d[j-1,:] = d[j-1,:] + 2.0*x*d[j,:]
        d[j-2,:] = d[j-2,:] - d[j,:]
    y = d[0,:] + x*d[1,:]
    return y
```

While using recursion it is important how the error (e.g. rounding error) propagates.

Simple example:

$$\begin{aligned} x_{n+1} &= 10x_n - 9, \\ x_0 = 1 &\Rightarrow x_n = 1 \quad \forall n \\ x_0 = 1 + \epsilon &\Rightarrow \widetilde{x}_n = 1 + 10^n\epsilon. \end{aligned}$$

This is not a problem here: Clenshaw algorithm is stable.

**Theorem 5.4.7** (Stability of Clenshaw algorithm)**.**
*Consider the perturbed Clenshaw algorithm recursion:*

$$\widetilde{d}_k = c_k + 2x \cdot \widetilde{d}_{k+1} - \widetilde{d}_{k+2} + \epsilon_k\,, \quad k = n, n-1, \dots, 0$$

*with* $\widetilde{d}_{n+2} = \widetilde{d}_{n+1} = 0$. *Set* $\widetilde{p}(x) = \frac{1}{2}(\widetilde{d}_0 - \widetilde{d}_2)$. *Then* $|\widetilde{p}(x) - p(x)| \leq \sum_{j=0}^{n} |\epsilon_j|$ *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 thier use for polynomial interpolation
- what algebraic/exponential convergence means
- error behavior for polynomial interpolation on equally spaced points
- reasons for uing rather Chebychev interpolation
- definition and properties of the Chebychev polynomials
- error behavior for Chebychev interpolation
- how to compute with Chebychev 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

We call trigonometrical polynomial of degree $\leq 2m$ the function

$$T_{2m}(t) := t \mapsto \sum_{j=-m}^{m} \gamma_j e^{2\pi i j t}\,, \qquad \gamma_j \in \mathbb{C}, t \in \mathbb{R}\,.$$





*Remark* 6.0.1. $T_{2m} : \mathbb{R} \to \mathbb{C}$ 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

$$T_{2m}(t) = \frac{a_0}{2} + \sum_{j=1}^{2m} \left(a_j \cos(2\pi j t) + b_j \sin(2\pi j t)\right)$$

with $a_0 = 2\gamma_0$ and $a_j = 2\,\mathrm{Re}\,\gamma_j$, $b_j = -2\,\mathrm{Im}\,\gamma_j$ 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, Mathematcial Methods of Physics, etc.):



**Theorem 6.0.1** ($L^2$-convergence of the Fourier series)**.** *Every squared integrable function* $f \in L^2(]0,1[) := \{f : ]0,1[ \mapsto \mathbb{C} \colon \|f\|_{L^2(]0,1[)} < \infty\}$ *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*

$$\widehat{f}(k) = \int_0^1 f(t)\, e^{-2\pi i k t}\, \mathrm{d}t\,, \quad k \in \mathbb{Z}\,.$$



*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* $t \to f(t)$ and once in the *frequency domain* $k \to \widehat{f}(k)$.

Isometry property (Parseval): $$\sum_{k=-\infty}^{\infty} |\widehat{f}(k)|^2 = \|f\|^2_{L^2(]0,1[)} \qquad (6.0.1)$$

*Remark* 6.0.4. A real function $f$ may be equally represented as

$$\frac{a_0}{2} + \sum_{j=1}^{\infty} \big(a_j \cos(2\pi jt) + b_j \sin(2\pi jt)\big) \quad \text{in } L^2(]0,1[) \ ,$$

with

$$a_j = 2\int_0^1 f(t)\,\cos(2\pi jt)\,\mathrm{d}t \ , \quad j \geq 0 \ ,$$
$$b_j = 2\int_0^1 f(t)\,\sin(2\pi jt)\,\mathrm{d}t \ , \quad j \geq 1 \ .$$

**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.

$$\left\|f^{(n)}\right\|^2_{L^2([0,1])} = (2\pi)^{2n} \sum_{k=-\infty}^{\infty} k^{2n} |\widehat{f}(k)|^2 \ . \qquad (6.0.2)$$

If $|\widehat{f^{(n)}}(k)| \leq \int_0^1 |f^{(n)}(t)|\,\mathrm{d}t < \infty \quad \Rightarrow \quad \widehat{f}(k) = O(k^{-n})$ for $|k| \to \infty$ .

The smoothness of a function directly reflects in the quick decay of its Fourier coefficients.

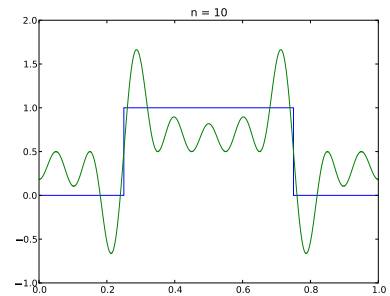
*Example* 6.0.6. The Fourier series associated with the characteristic function of an interval $[a,b] \subset ]0.1[$ may be computed analytically as

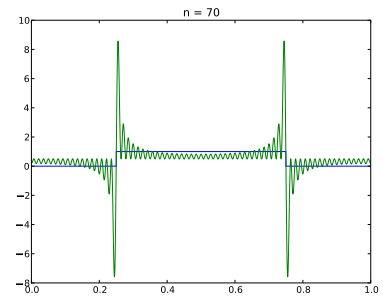
$$b - a + \frac{1}{\pi} \sum_{|k|=1}^{\infty} e^{-ikc} \frac{\sin kd}{k} e^{i2\pi kt} \ , \quad t \in [0,1] \ ,$$

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 uniforme convergence of the series!








*Remark* 6.0.7. Usually one cannot compute analytically $\widehat{f}(k)$ or one has to rely only on discrete values at nodes $x_\ell = \frac{\ell}{N}$ for $\ell = 0, 1, \ldots, N$; the trapezoidal rule gives

$$\hat{f}(k) \approx \frac{1}{N} \sum_{\ell=0}^{N-1} f(x_\ell) e^{-2\pi i k x_\ell} =: \hat{f}_N(k) \qquad (6.0.3)$$

## 6.1 Discrete Fourier Transform (DFT)

Let $n \in \mathbb{N}$ fixed and denote the $n$th root of unity $\omega_n := \exp(-2\pi i/n) = \cos(2\pi/n) - i\sin(2\pi/n)$

hence $$\omega_n^k = \omega_n^{k+n} \ \forall k \in \mathbb{Z} \quad , \quad \omega_n^n = 1 \quad , \quad \omega_n^{n/2} = -1 \ , \qquad (6.1.1)$$

$$\sum_{k=0}^{n-1} \omega_n^{kj} = \begin{cases} n & \text{, if } j = 0 \ , \\ 0 & \text{, if } j \neq 0 \ . \end{cases} \qquad (6.1.2)$$

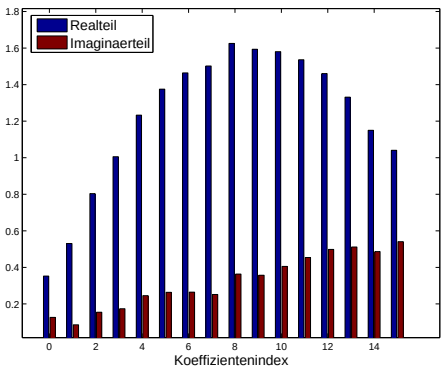
A change of basis in $\mathbb{C}^n$:

standard basis of $\mathbb{K}^n$ — Trigonometrical Basis

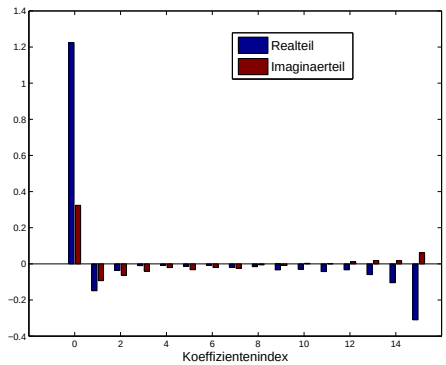
$$\left\{ \begin{pmatrix} 1 \\ 0 \\ \vdots \\ \\ \vdots \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ \vdots \\ 0 \end{pmatrix} \cdots \begin{pmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ \vdots \\ \\ \vdots \\ 0 \\ 1 \end{pmatrix} \right\} \longleftarrow \left\{ \begin{pmatrix} \omega_n^0 \\ \vdots \\ \\ \vdots \\ \omega_n^0 \end{pmatrix} \begin{pmatrix} \omega_n^0 \\ \omega_n^1 \\ \vdots \\ \\ \vdots \\ \omega_n^{n-1} \end{pmatrix} \cdots \begin{pmatrix} \omega_n^0 \\ \omega_n^{n-2} \\ \omega_n^{2(n-2)} \\ \vdots \\ \vdots \\ \omega_n^{(n-1)(n-2)} \end{pmatrix} \begin{pmatrix} \omega_n^0 \\ \omega_n^{n-1} \\ \omega_n^{2(n-1)} \\ \vdots \\ \vdots \\ \omega_n^{(n-1)^2} \end{pmatrix} \right\}$$

Matrix of change of basis  trigonometrical basis ⟶ standard basis:  Fourier-matrix

$$\mathbf{F}_n = \begin{pmatrix} \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \cdots & \omega_n^{2n-2} \\ \vdots & \vdots & & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)^2} \end{pmatrix} = \left(\omega_n^{ij}\right)_{i,j=0}^{n-1} \in \mathbb{C}^{n,n} . \tag{6.1.3}$$




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* $\mathbf{c} := \mathcal{F}_n(\mathbf{y})$

$$c_k = \sum_{j=0}^{n-1} y_j\, \omega_n^{kj} \quad , \quad k = 0, \dots, n-1 . \tag{6.1.4}$$

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:*  $\mathbf{F}_n^{-1} = \frac{1}{n}\mathbf{F}_n^H = \frac{1}{n}\overline{\mathbf{F}}_n$

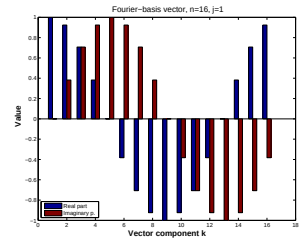
*Remark* 6.1.1. $\frac{1}{n}\mathbf{F}_n^2 = -I$ and $\frac{1}{n^2}\mathbf{F}_n^4 = I$, hence the eigenvalues of $\mathbf{F}_n$ are in the set $\{1, -1, i, -i\}$.

numpy-functions:  `c=fft(y)` ↔ `y=ifft(c);`

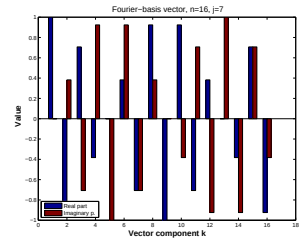
*Example* 6.1.2 (Frequency analysis with DFT).

Some vectors of the Fourier basis ($n = 16$):






"'low frequence"




"'high frequence"

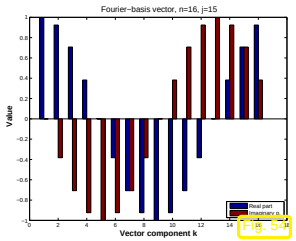



Fig. 54

"'low frequence"

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()
```

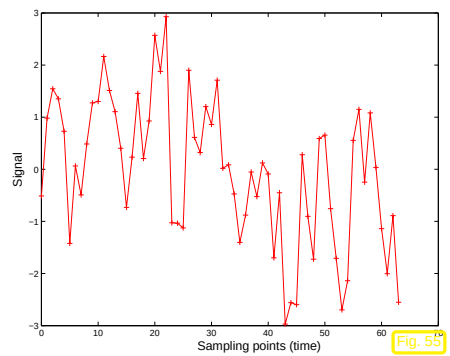



Fig. 55

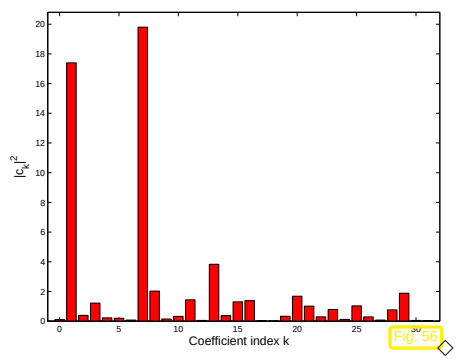



Fig. 56

## 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×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

```
from numpy import zeros, random, exp, pi, meshgrid, r_, dot,  fft
import timeit

def naiveFT():
    global y, n

    c = 0.*1j*y
    # naive
    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): c[jj] = c[jj]*s+y[kk]
        s *= omega
    #return c

def matrixFT():
```



```
    global y, n

    # matrix based
    I, J = meshgrid(r_[:n], r_[:n])
    F = exp(-2*pi*1j*I*J/n)
    tm = 10**3
    c = dot(F,y)
    #return c

def fftFT():
    global y,n
    c = fft.fft(y)
    #return c

nrexp = 5
N = 2**11 # how large the vector will be
res = zeros((N,4))
for n in xrange(1,N+1):
    y = random.rand(n)

    t = timeit.Timer('naiveFT()', 'from __main__ import naiveFT')
    tn = t.timeit(number=nrexp)
    t = timeit.Timer('matrixFT()', 'from __main__ import matrixFT')
```

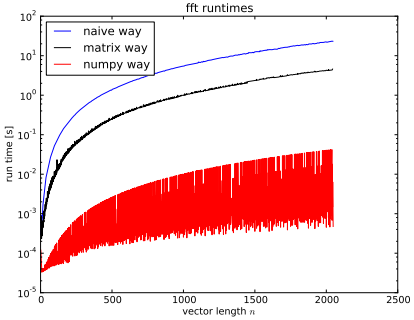


```
    tm = t.timeit(number=nrexp)
    t = timeit.Timer('fftFT()', 'from __main__ import fftFT')
    tf = t.timeit(number=nrexp)
    #print n, tn, tm, tf
    res[n-1] = [n, tn, tm, tf]

print res[:,3]
from pylab import semilogy, show, plot, savefig
semilogy(res[:,0], res[:,1], 'b-')
semilogy(res[:,0], res[:,2], 'k-')
semilogy(res[:,0], res[:,3], 'r-')
sevefig('ffttime.eps')
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,-
        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 $n = 2m$, $m \in \mathbb{N}$:

$$\begin{aligned} c_k &= \sum_{j=0}^{n-1} y_j e^{-\frac{2\pi i}{n} jk} \\ &= \sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{n} 2jk} + \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{n}(2j+1)k} \\ &= \underbrace{\sum_{j=0}^{m-1} y_{2j} \underbrace{e^{-\frac{2\pi i}{m} jk}}_{=\omega_m^{jk}}}_{=:\widetilde{c}_k^{\mathtt{even}}} + e^{-\frac{2\pi i}{n} k} \cdot \underbrace{\sum_{j=0}^{m-1} y_{2k+1} \underbrace{e^{-\frac{2\pi i}{m} jk}}_{=\omega_m^{jk}}}_{=:\widetilde{c}_k^{\mathtt{odd}}} . \end{aligned} \tag{6.2.1}$$

Note $m$-periodicity: $\widetilde{c}_k^{\mathtt{even}} = \widetilde{c}_{k+m}^{\mathtt{even}}$, $\widetilde{c}_k^{\mathtt{odd}} = \widetilde{c}_{k+m}^{\mathtt{odd}}$.



Note: $\widetilde{c}_k^{\mathtt{even}}, \widetilde{c}_k^{\mathtt{odd}}$ from DFTs of length $m$!

with $\mathbf{y}_{\mathtt{even}} := (y_0, y_2, \dots, y_{n-2})^T \in \mathbb{C}^m$: $\left(\widetilde{c}_k^{\mathtt{even}}\right)_{k=0}^{m-1} = \mathbf{F}_m \mathbf{y}_{\mathtt{even}}$ ,

with $\mathbf{y}_{\mathtt{odd}} := (y_1, y_3, \dots, y_{n-1})^T \in \mathbb{C}^m$: $\left(\widetilde{c}_k^{\mathtt{odd}}\right)_{k=0}^{m-1} = \mathbf{F}_m \mathbf{y}_{\mathtt{odd}}$ .

(6.2.1): DFT of length $2m$ = 2× DFT of length $m$ + 2m additions & multiplications



Idea:
divide & conquer recursion

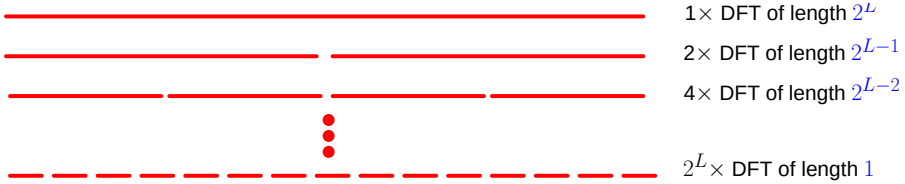
(for DFT of length $n = 2^L$)

FFT-algorithm

Code 6.2.3: Recursive FFT

```
from numpy import linspace, random, hstack, pi, exp
def fftrec(y):
    n = len(y)
    if n == 1:
        c = y
        return c
    else:
        c0 = fftrec(y[::2])
        c1 = fftrec(y[1::2])
        r = (-2.j*pi/n) * linspace(0,n-1,n)
        c = hstack((c0,c0)) + exp(r) * hstack((c1,c1))
    return c
```

Computational cost of `fftrec`:




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 $\approx 5n \log_2 n$).

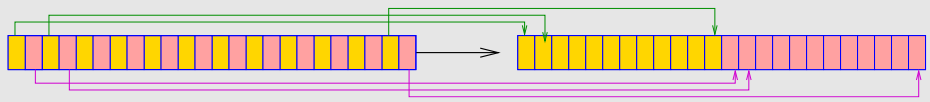
➢



*Remark* 6.2.4 (FFT algorithm by matrix factorization).

For $n = 2m$, $m \in \mathbb{N}$,

permutation $P_m^{\mathrm{OE}}(1, \dots, n) = (1, 3, \dots, n-1, 2, 4, \dots, n)$ .



As $\omega_n^{2j} = \omega_m^j$:

permutation of rows
$$P_m^{\mathrm{OE}}\mathbf{F}_n = \left(\begin{array}{c|c} \mathbf{F}_m & \mathbf{F}_m \\ \hline \mathbf{F}_m \begin{pmatrix} \omega_n^0 & & & \\ & \omega_n^1 & & \\ & & \ddots & \\ & & & \omega_n^{n/2-1} \end{pmatrix} & \mathbf{F}_m \begin{pmatrix} \omega_n^{n/2} & & & \\ & \omega_n^{n/2+1} & & \\ & & \ddots & \\ & & & \omega_n^{n-1} \end{pmatrix} \end{array}\right)$$

$$\left(\begin{array}{c|c} \mathbf{F}_m & \\ \hline & \mathbf{F}_m \end{array}\right) \left(\begin{array}{c|c} \mathbf{I} & \mathbf{I} \\ \hline \begin{pmatrix} \omega_n^0 & & & \\ & \omega_n^1 & & \\ & & \ddots & \\ & & & \omega_n^{n/2-1} \end{pmatrix} & \begin{pmatrix} -\omega_n^0 & & & \\ & -\omega_n^1 & & \\ & & \ddots & \\ & & & -\omega_n^{n/2-1} \end{pmatrix} \end{array}\right)$$

Example: factorization of Fourier matrix for $n = 10$





$$P_5^{\mathrm{OE}}\mathbf{F}_{10} = \left(\begin{array}{ccccc|ccccc} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 \\ \omega^0 & \omega^4 & \omega^8 & \omega^2 & \omega^6 & \omega^0 & \omega^4 & \omega^8 & \omega^2 & \omega^6 \\ \omega^0 & \omega^6 & \omega^2 & \omega^8 & \omega^4 & \omega^0 & \omega^6 & \omega^2 & \omega^8 & \omega^4 \\ \omega^0 & \omega^8 & \omega^6 & \omega^4 & \omega^2 & \omega^0 & \omega^8 & \omega^6 & \omega^4 & \omega^2 \\ \hline \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 & \omega^8 & \omega^9 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 & \omega^2 & \omega^5 & \omega^8 & \omega^1 & \omega^4 & \omega^7 \\ \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 \\ \omega^0 & \omega^7 & \omega^4 & \omega^1 & \omega^8 & \omega^5 & \omega^2 & \omega^9 & \omega^6 & \omega^3 \\ \omega^0 & \omega^9 & \omega^8 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{array}\right), \quad \omega := \omega_{10}\,.$$

△



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)

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{jk} \overset{[j=:lp+m]}{=} \sum_{m=0}^{p-1}\sum_{l=0}^{q-1} y_{lp+m} e^{-\frac{2\pi i}{pq}(lp+m)k} = \sum_{m=0}^{p-1} \omega_n^{mk} \sum_{l=0}^{q-1} y_{lp+m}\, \omega_q^{l(k \mod q)}\,. \tag{6.2.2}$$

Step I: perform $p$ DFTs of length $q$ $\quad z_{m,k} := \sum_{l=0}^{q-1} y_{lp+m}\, \omega_q^{lk}, \quad 0 \le m < p, 0 \le k < q.$

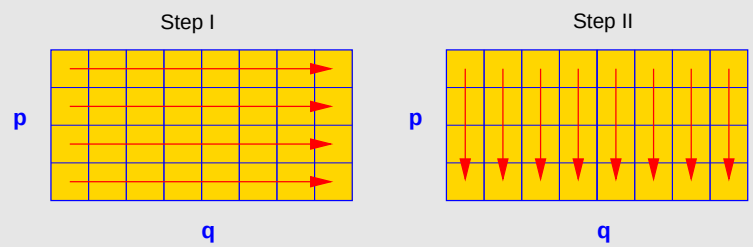
Step II: for $k =: rq + s, \quad 0 \le r < p, 0 \le s < q$

$$c_{rq+s} = \sum_{m=0}^{p-1} e^{-\frac{2\pi i}{pq}(rq+s)m} z_{m,s} = \sum_{m=0}^{p-1} \left(\omega_n^{ms} z_{m,s}\right) \omega_p^{mr}$$





and hence $q$ DFTs of length $p$ give all $c_k$.






*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:

$$\forall p \in \mathbb{N} \text{ prime} \quad \exists g \in \{1,\dots,p-1\}: \quad \{g^k \mod p : k = 1,\dots,p-1\} = \{1,\dots,p-1\} \,,$$

► permutation $P_{p,g} : \{1,\dots,p-1\} \mapsto \{1,\dots,p-1\}$ , $P_{p,g}(k) = g^k \mod p$ ,
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|ccccccccccccc}
\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 `c=fft(y)` for $\mathbf{y} \in \mathbb{C}^n$ = $O(n \log n)$.





## 6.3 Trigonometric Interpolation: Computational Aspects

**Theorem 6.3.1** (Trigonometric interpolant). *Let $N$ even and $z = \mathcal{F}_N y \in \mathbb{C}^N$. The trigonometric interpolant*

$$p_N(x) := \sum_{k=-N/2}^{N/2}{}' z_k e^{2\pi i k x} := \frac{1}{2}\left(z_{-N/2}e^{-2\pi i x N/2} + z_{N/2}e^{2\pi i x N/2}\right) + \sum_{|k|\le N/2} z_k e^{2\pi i k x}$$

*has the property that $p_N(x_\ell) = y_\ell$, where $x_\ell = \ell/N$.*

► Fast intepolation by means of FFT: $O(n \log n)$ asymptotic complexity, see Sect. 6.2, provide uniformly distributed nodes $t_k = \dfrac{k}{2n+1}$, $k = 0,\dots,2n$

► $(2n+1) \times (2n+1)$ linear system of equations:

$$\sum_{j=0}^{2n} \gamma_j \exp\left(2\pi i \frac{jk}{2n+1}\right) = z_k := \exp\left(2\pi i \frac{nk}{2n+1}\right) y_k \,, \quad k = 0,\dots,2n \,.$$

$$\Updownarrow$$

$$\overline{\mathbf{F}}_{2n+1}\mathbf{c} = \mathbf{z} \,, \quad \mathbf{c} = (\gamma_0,\dots,\gamma_{2n})^T \quad \overset{\text{Lemma 6.1.2}}{\Longrightarrow} \quad \mathbf{c} = \frac{1}{2n+1}\mathbf{F}_{2n}\mathbf{z} \,. \tag{6.3.1}$$

$(2n+1) \times (2n+1)$ (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{(j/(2n+1), y_j)},
  \Blue{j = 0,...,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 (??): α_j = 1/2(γ_{n-j} + γ_{n+j}) and
    # β_j = 1/(2i)(γ_{n-j} - γ_{n+j}), j = 1,...,n, α_0 = γ_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,...,t_n ∈ [0,1[}
    \texttt{y}: row vector of data \Blue{y_0,...,y_n}
    return values are column vectors of expansion coefficients \Blue{α_j},
      \Blue{β_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 ▷

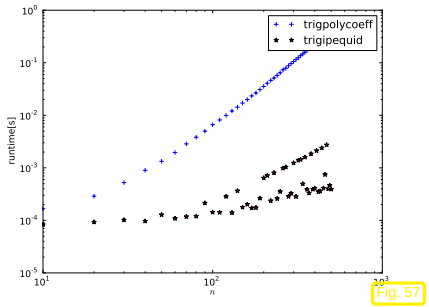



Fig. 57

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.

◇

6.3 p. 337

*Remark* 6.3.6 (Efficient evaluation of trigonometric interpolation polynomials).

Task: evaluation of trigonometric polynomial (**??**) at *equidistant* points $\frac{k}{N}$, $N > 2n$. $k = 0,\dots,N-1$.

$$(??) \quad \blacktriangleright \quad q(k/N) = e^{-2\pi i k/N} \sum_{j=0}^{2n} \gamma_j \exp(2\pi i \frac{kj}{N}) \,, \quad k = 0,\dots,N-1 \,.$$

$$\blacktriangleright \quad q(k/N) = e^{-2\pi i k n/N} v_j \quad \text{with} \quad \mathbf{v} = \overline{\mathbf{F}}_N \widetilde{\mathbf{c}} \,, \tag{6.3.2}$$

Fourier matrix, see (6.1.3).

where $\widetilde{\mathbf{c}} \in \mathbb{C}^N$ is obtained by zero padding of $\mathbf{c} := (\gamma_0,\dots,\gamma_{2n})^T$:

$$(\widetilde{\mathbf{c}})_k = \begin{cases} \gamma_j & \text{, for } k = 0,\dots,2n \,, \\ 0 & \text{, for } k = 2n+1,\dots,N-1 \,. \end{cases}$$

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{α_j},
```

6.3 p. 338

```
  \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)
```

Code 6.3.8: *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.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 $f : \mathbb{R} \mapsto \mathbb{C}$

$$\mathsf{T}_n(f) := p \in \mathcal{P}_{n-1}^T \quad \text{with} \quad p(\tfrac{j}{n}) = y_j\,, \quad j = 0, \dots, n-1\,.$$

*Example* 6.4.1 (Trigonometric interpolation).

#1 step function: $f(t) = 0$ for $|t - \frac{1}{2}| > \frac{1}{4}$, $f(t) = 1$ for $|t - \frac{1}{2}| \leq \frac{1}{4}$

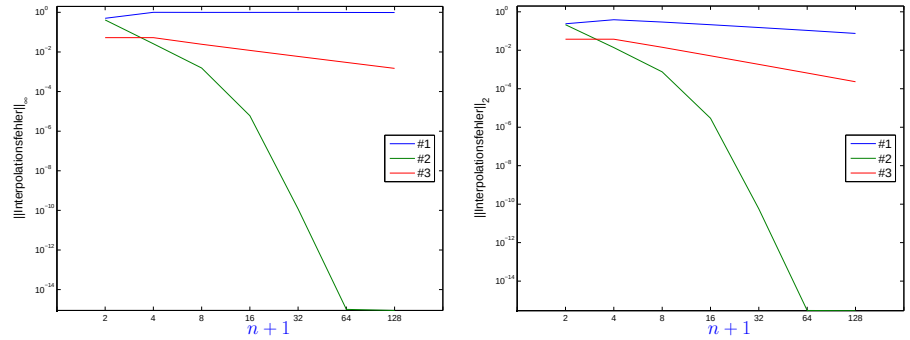
#2 smooth periodic function: $f(t) = \dfrac{1}{\sqrt{1 + \frac{1}{2}\sin(2\pi t)}}$.

#3 hat function: $f(t) = |t - \frac{1}{2}|$

Note: computation of the norms of the interpolation error:

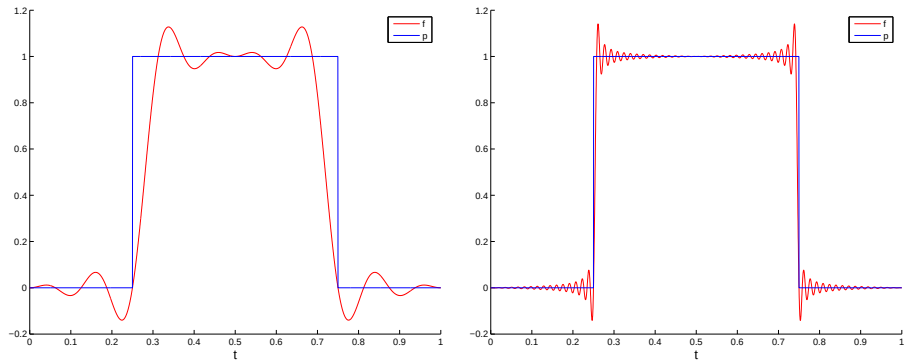
```
    t = linspace(0,1,n+1);  y = f(t[:-1])
    v = real(evaliptrig(y,N))
    t = linspace(0,1,N+1);  fv = f(t)
    d = abs(v-fv[:-1]); linf = d.max()
    l2 = linalg.norm(d)/sqrt(N)
```






Note: Cases #1, #3: algebraic convergence
Case #2: exponential convergence





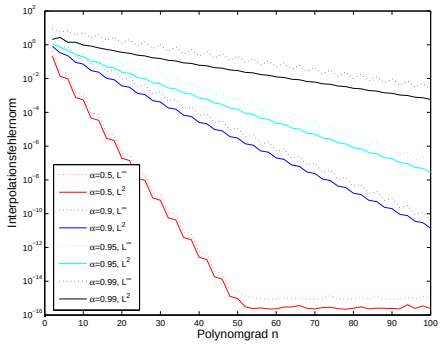

$n = 16$ $n = 128$

➢ Gibbs'phenomenon: ripples near discontinuities ◇

*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 $\alpha$ ◇

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:*

$$\hat{f}_N(k) - \hat{f}(k) = O(N^{-p}) \quad \text{for} \quad |k| \leq \frac{N}{2}$$

$$\text{for:} \quad h=\frac{1}{N}, \quad h\sum_{j=0}^{N-1} f(x_j) - \int_0^1 f(x)dx = O(h^p).$$

*Remark* 6.4.3. $\hat{f}_N(k)$ is $N$-periodic, while $\hat{f}(k)\to 0$ 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 interpolantion)**.** *If* $f$ *is* $1$*-periodic and* $\sum_{k\in\mathbb{Z}} \hat{f}(k)$ *absolut 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* $N>2M$



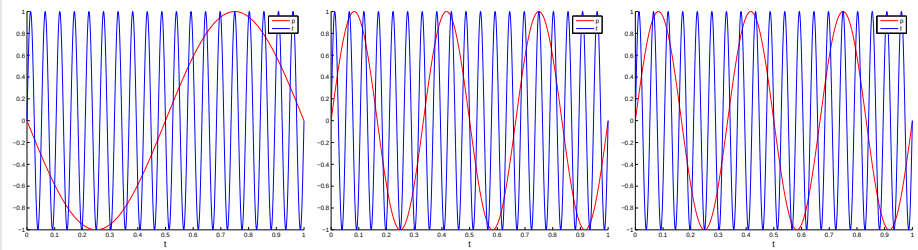
data compression

*Remark* 6.4.4 (Aliasing).

$$\mathcal{T} = \{\tfrac{j}{n}\}_{j=0}^{n-1} \quad ▶ \quad e^{2\pi iNt}_{|\mathcal{T}} = e^{2\pi i(N-n)t}_{|\mathcal{T}} \quad ▶ \quad \mathsf{T}_n(e^{2\pi iN\cdot}) = \mathsf{T}_n(e^{2\pi i(N \bmod n)\cdot})\,.$$

hence: The trigonometric interpolation of $t\to e^{2\pi iNt}$ and $t\to e^{2\pi i(N \bmod n)t}$ of degree $n$ give the same trigonometric interpolation polynomial ! $\longrightarrow$ Aliasing

Example for $f(t)=\sin(2\pi\cdot 19t)$ ➢ $N=38$:



$n=4$ $n=8$ $n=16$ △

[Linearity of trigonometric interpolation]

For $f\in C([0,1])$ 1-periodic, $n=2m$:

$$f(t) = \sum_{j=-\infty}^{\infty} \widehat{f}(j)e^{2\pi ijt} \quad \Rightarrow \quad \mathsf{T}_n(f)(t) = \sum_{j=-m+1}^{m} \gamma_j e^{2\pi ijt}\,, \quad \gamma_j = \sum_{l=-\infty}^{\infty} \widehat{f}(j+ln)\,.$$

▶ Fourier coefficients of the error function $e=f-\mathsf{T}_n(f)$:

$$\widehat{e}(j) = \begin{cases} -\sum_{|l|=1}^{\infty} \widehat{f}(j+ln) & \text{, if } -m+1\leq j\leq m\,,\\ \widehat{f}(j) & \text{, if } j\leq -m \vee j>m\,. \end{cases}$$

$$\overset{(6.0.1)}{\Rightarrow} \quad \|f-\mathsf{T}_n(f)\|^2_{L^2(]0,1[)} \leq \Big|\sum_{|l|=1}^{\infty} \widehat{f}(j+ln)\Big|^2 + \sum_{|j|\geq m} |\widehat{f}(j)|^2\,. \qquad (6.4.1)$$

▶ may be estimated, if we know the decay of the Fourier coefficients $\widehat{f}(j)$ $\Leftrightarrow$ smoothness of $f$, see (6.0.2)

**Theorem 6.4.5** (Error estimate for trigonometric interpolation). *For $k \in \mathbb{N}$:*

$$f^{(k)} \in L^2(]0,1[) \quad \Rightarrow \quad \|f - \mathsf{T}_n f\|_{L^2(]0,1[)} \leq \sqrt{1+c_k}\, n^{-k} \left\| f^{(k)} \right\|_{L^2(]0,1[)} ,$$

*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 $f : [-1,1] \mapsto \mathbb{C}$ (→ section 5.4)

$$p(t_k) = f(t_k) \quad \text{for Chebychev nodes, see (5.4.5),} \quad t_k := \cos\left(\frac{2k+1}{2(n+1)}\pi\right), \; k = 0,\dots,n .$$

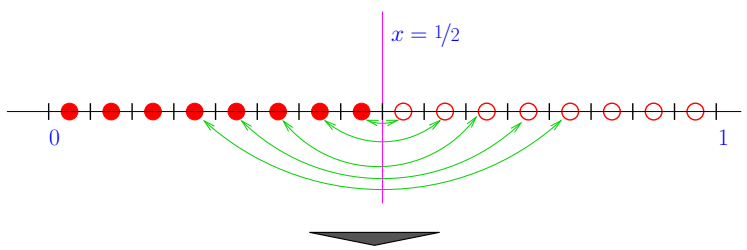
Define now the corresponding functions $g, q$:

$$\left.\begin{array}{lcl} f : [-1,1] \mapsto \mathbb{C} & \leftrightarrow & g(s) := f(\cos 2\pi s) , \\ p : [-1,1] \mapsto \mathbb{C} & \leftrightarrow & q(s) := p(\cos 2\pi s) , \end{array}\right\} \quad \text{1-periodic, symmetric wrt. } 0 \text{ and } \frac{1}{2}$$

Hence:

$$p(t_k) = f(t_k) \quad \Leftrightarrow \quad q\left(\frac{2k+1}{4(n+1)}\right) = g\left(\frac{2k+1}{4(n+1)}\right) . \tag{6.5.1}$$

and with a translation $\tilde{s} = s + \frac{1}{4(n+1)}$ we have:




$$\begin{array}{l} \widetilde{g}(s) := g(s + \dfrac{1}{4(n+1)}) , \\ \widetilde{q}(s) := q(s + \dfrac{1}{4(n+1)}) \end{array} \quad \Rightarrow \quad \widetilde{q}\left(\frac{k}{2(n+1)}\right) = \widetilde{g}\left(\frac{k}{2n+2}\right) , \quad k = 0,\dots,2n .$$

! We show: $q(s)$ is trigonometric polynomial: $\boxed{q \in \mathcal{P}_{2n}^T}$, namely the trigonometric interpolation polynomial of $\widetilde{g}$. Indeed, since $p$ is the Chebychev interpolation polynomial:





$$q(s) = p(\cos 2\pi s) = \sum_{j=0}^{n} \gamma_j T_j(\cos 2\pi s) = \sum_{j=0}^{n} \gamma_j \cos 2\pi s = \gamma_0 + \sum_{j=-n, j\neq 0}^{n} \tfrac{1}{2}\gamma_{|j|} e^{-2\pi i j s} =$$

$$\gamma_0 + \sum_{j=-n, j\neq 0}^{n} \tfrac{1}{2}\gamma_{|j|} e^{2\pi i \frac{j}{4(n+1)}} \cdot e^{-2\pi i j \left(s + \frac{1}{4(n+1)}\right)} = \sum_{j=-n}^{n} \tilde{\gamma}_j e^{-2\pi i j \tilde{s}}$$

$$\text{with} \quad \widetilde{\gamma}_j := \begin{cases} \frac{1}{2}\exp\left(2\pi i \frac{j}{4(n+1)}\right)\gamma_{|j|} & \text{, if } j \neq 0 , \\ \gamma_0 & \text{, for } j = 0 . \end{cases}$$

Hence

$$\tilde{q}(s) = \sum_{j=-n}^{n} \tilde{\gamma}_j e^{-2\pi i j s}$$

$$\text{and } \tilde{q}\left(\frac{k}{2(n+1)}\right) = \tilde{g}\left(\frac{k}{2(n+1)}\right) , \quad \text{for all } k = 0,\dots,2n .$$

that means that $\tilde{q}$ is the trigonometric interpolation polynomial of $\tilde{g}$.

Hence $\|f - p\|_{L^\infty([-1,1])} = \|\widetilde{g} - \widetilde{q}\|_{L^\infty([0,1])}$

and the error estimates from the trigonometric interpolation directly transfers here.

Code 6.5.1: Efficient Chebyshev interpolation

```
from numpy import exp, pi, real, hstack, arange
from scipy import ifft

def chebexp(y):
    """Efficiently compute coefficients \Blue{α_j} in the Chebychev
      expansion
    \Blue{p = Σ_{j=0}^{n} α_j T_j} of \Blue{p ∈ P_n} based on values \Blue{y_k},
    \Blue{k = 0,...,n}, in Chebychev nodes \Blue{t_k}, \Blue{k = 0,...,n}.
      These values are
    passed in the row vector \texttt{y}.
    """

    # degree of polynomial
    n = y.shape[0] -1

    # create vector z by wrapping and componentwise scaling

    # r.h.s. vector
    t = arange(0, 2*n+2)
    z = exp(-pi*1.0j*n/(n+1.0)*t) * hstack([y,y[::-1]])

    # Solve linear system (??) with effort O(n log n)
```

```
    c = ifft(z)

    # recover β_j, see (??)
    t = arange(-n, n+2)
    b = real(exp(0.5j*pi/(n+1.0)*t) * c)

    # recover α_j, see (??)
    a = hstack([ b[n], 2*b[n+1:2*n+1] ])

    return a

if __name__ == "__main__":
    from numpy import array

    # Test with arbitrary values
    y = array([1, 2, 3, 3.5, 4, 6.5, 6.7, 8, 9])

    # Expected values:
    # 4.85556 -3.66200 0.23380 -0.25019 -0.15958 -0.36335 0.18889 0.16546 -0.27329

    w = chebexp(y)

    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 dicrete 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 trionomteric 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 $\int_\Omega f(\mathbf{x})\,\mathrm{d}\mathbf{x}$, 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.

*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") ⚠



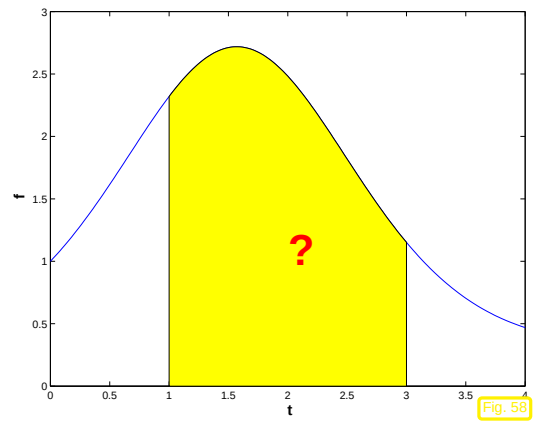



Fig. 58

Numerical quadrature methods

*approximate*

$$\int_a^b f(t)\,\mathrm{d}t$$

*Example* 7.0.2 (Heating production in electrical circuits).

Time-harmonic excitation:

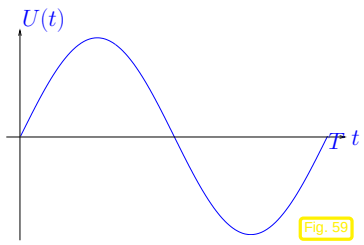



Fig. 59

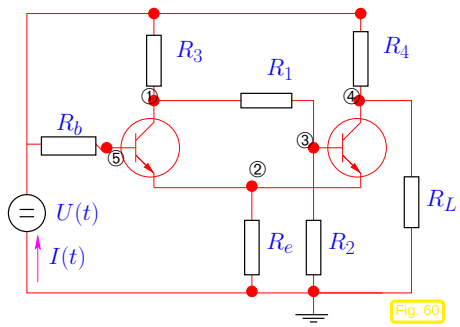



Fig. 60

Integrating power $P = UI$ over period $[0,T]$ yields heat production per period:

$$W_{\text{therm}} = \int_0^T U(t)I(t)\,\mathrm{d}t\,, \quad \text{where} \quad I = I(U)\,.$$

`function I = current(U)` involves solving non-linear system of equations!

◇

## 7.1 Quadrature Formulas

$n$-point quadrature formula on $[a,b]$: ($n$-point quadrature rule)

$$\int_a^b f(t)\,\mathrm{d}t \approx Q_n(f) := \sum_{j=1}^n w_j^n\, f(c_j^n)\,. \qquad (7.1.1)$$

$w_j^n$ : quadrature weights $\in \mathbb{R}$ (*ger.:* Quadraturgewichte)
$c_j^n$ : quadrature nodes $\in [a,b]$ (*ger.:* Quadraturknoten)

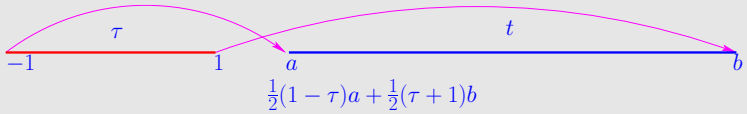
*Remark* 7.1.1 (Transformation of quadrature rules).

Given: quadrature formula $(\widehat{c}_j, \widehat{w}_j)_{j=1}^n$ on reference interval $[-1,1]$

Idea: transformation formula for integrals

$$\int_a^b f(t)\,\mathrm{d}t = \tfrac{1}{2}(b-a)\int_{-1}^1 \widehat{f}(\tau)\,\mathrm{d}\tau\,, \quad \widehat{f}(\tau) := f(\tfrac{1}{2}(1-\tau)a + \tfrac{1}{2}(\tau+1)b)\,. \qquad (7.1.2)$$






► quadrature formula for general interval $[a,b]$, $a,b \in \mathbb{R}$:

$$\int_a^b f(t)\,\mathrm{d}t \approx \tfrac{1}{2}(b-a)\sum_{j=1}^n \widehat{w}_j \widehat{f}(\widehat{c}_j) = \sum_{j=1}^n w_j f(c_j) \quad \text{with} \quad \begin{aligned} c_j &= \tfrac{1}{2}(1-\widehat{c}_j)a + \tfrac{1}{2}(1+\widehat{c}_j)b\,, \\ w_j &= \tfrac{1}{2}(b-a)\widehat{w}_j\,. \end{aligned}$$



► A 1D quadrature formula on arbitrary intervals can be specified by providing its weights $\widehat{w}_j$ /nodes $\widehat{c}_j$ for integration domain $[-1,1]$. Then the above transformation is assumed.

Other common choice of reference interval: $[0,1]$

△





Inevitable for generic integrand:

$$\text{quadrature error} \qquad E(n) := \left| \int_a^b f(t)\,\mathrm{d}t - Q_n(f) \right|$$

Given families of quadrature rules $\{Q_n\}_n$ with quadrature weights $\left\{w_j^n,\ j=1,\dots,n\right\}_{n\in\mathbb{N}}$ and quadrature nodes $\left\{c_j^n,\ j=1,\dots,n\right\}_{n\in\mathbb{N}}$ we

should be aware of the *asymptotic* behavior of quadrature error $E(n)$ for $n \to \infty$

Qualitative distinction:
- ▷ algebraic convergence $E(n) = O(n^{-p}), p > 0$
- ▷ exponential convergence $E(n) = O(q^n), 0 \le q < 1$



Note that the number $n$ of nodes agrees with the number of $f$-evaluations required for evaluation of the quadrature formula. This is usually used as a *measure for the cost* of computing $Q_n(f)$.

Therefore 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

$$\widehat{Q}_{\mathrm{trp}}(f) := \frac{1}{2}\left(f(0) + f(1)\right) \tag{7.1.3}$$

$$\left( \int_a^b f(t)\,\mathrm{d}t \approx \frac{b-a}{2}(f(a)+f(b)) \right)$$

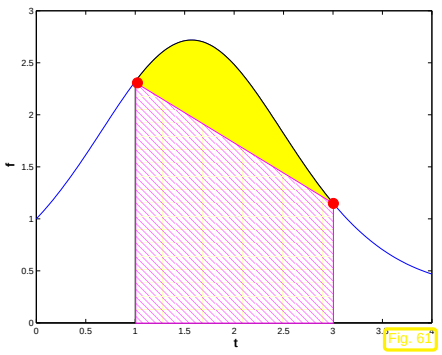



Fig. 61

- $n = 2$: Simpson rule

$$\frac{h}{6}\left(f(0) + 4\,f(\tfrac{1}{2}) + f(1)\right) \quad \left( \int_a^b f(t)\,\mathrm{d}t \approx \frac{b-a}{6}\left(f(a) + 4\,f\left(\frac{a+b}{2}\right) + f(b)\right) \right) \tag{7.1.4}$$

◇

*Remark* 7.1.3 (Error estimates for polynomial quadrature).

Quadrature error estimates directly from $L^\infty$-interpolation error estimates for Lagrangian interpolation with polynomial of degree $n-1$:

$$f \in C^n([a,b]) \quad \Rightarrow \quad \left| \int_a^b f(t)\,\mathrm{d}t - Q_n(f) \right| \le \frac{1}{n!}(b-a)^{n+1} \left\| f^{(n)} \right\|_{L^\infty([a,b])} . \tag{7.1.5}$$

△

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 $\int_a^b x^4 dx$):

$$Q_n(p) = \int_a^b p(t)\,\mathrm{d}t \;\forall p \in \mathcal{P}_3 \quad \Leftrightarrow \quad Q_n(t^q) = \frac{1}{q+1}(b^{q+1} - a^{q+1}) \;, \quad q = 0,1,2,3 \;.$$

4 equations for weights $w_j$ and nodes $c_j$, $j = 1,2$ $(a = -1, b = 1)$, *cf.* Rem. **??**

$$\begin{aligned} \int_{-1}^1 1\,\mathrm{d}t = 2 = 1w_1 + 1w_2 \quad &, \quad \int_{-1}^1 t\,\mathrm{d}t = 0 = c_1 w_1 + c_2 w_2 \\ \int_{-1}^1 t^2\,\mathrm{d}t = \frac{2}{3} = c_1^2 w_1 + c_2^2 w_2 \quad &, \quad \int_{-1}^1 t^3\,\mathrm{d}t = 0 = c_1^3 w_1 + c_2^3 w_2 \;. \end{aligned} \tag{7.1.6}$$

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: $$\int_{-1}^1 f(x)\,dx \approx f\left(\frac{1}{\sqrt{3}}\right) + f\left(-\frac{1}{\sqrt{3}}\right) \tag{7.1.7}$$

◇

*Remark* 7.1.5 (Computing Gauss nodes and weights).

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!

Code 7.1.6: Golub-Welsch algorithm

```
from numpy import zeros, diag, sqrt, size
from numpy.linalg import eigh

def gaussquad(n):
        b = zeros(n-1);
        for i in xrange(size(b)):
          b[i]=(i+1)/sqrt(4*(i+1)*(i+1)-1)
        J=diag(b,-1)+diag(b,1)
        x,ev=eigh(J); w=2*ev[0]*ev[0]
        return (x,w)
```

△

Idea: Clenshaw-Curtis quadrature:

$$\int_{-1}^{1} f(x)\,dx = \int_{0}^{\pi} f(\cos\theta)\,\sin\theta d\theta = \sum_{\text{even } k} \frac{2a_k}{1-k^2} \tag{7.1.8}$$

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: Clensahw-Curtis: direct implementation

```
def cc2(func,a,b,N):
    """
    Clenshaw-Curtis quadrature rule
    by FFT with the function values
    """
    bma = 0.5*(b-a)
    x = np.cos(np.pi * np.linspace(0,N,N+1)/N) # Chebyshev points
    x *= bma
    x += 0.5*(a+b)
    fx = func(x)*0.5/N
    vx = np.hstack((fx,fx[-2:0:-1]))
    g = np.real(np.fft.fft(vx))
    A = np.zeros(N+1)
    A[0] = g[0]; A[N] = g[N]
    A[1:N] = g[1:N] + np.flipud(g[N+1:])
    w = 0.*x
    w[::2] = 2./(1.-np.r_[:N+1:2]**2)
    return np.dot(w,A)*bma
```

Code 7.1.8: Clensahw-Curtis: weights and points

```
def cc1(func, a, b, N):
    """
    Clenshaw-Curtis quadrature rule
    by constructing the points and the weights
    """
    bma = b-a
    c = np.zeros([2,2*N-2])
    c[0][0] = 2.0
    c[1][1] = 1
    c[1][-1]= 1
    for i in np.arange(2.,N,2):
        val = 2.0/(1-pow(i,2))
        c[0][i] = val
        c[0][2*N-2-i] = val

    f = np.real(np.fft.ifft(c))
    w = f[0][:N]; w[0] *= 0.5; w[-1] *= 0.5 # weights
    x = 0.5*((b+a)+(N-1)*bma*f[1][:N]) # points
    return np.dot(w,func(x))*bma
```

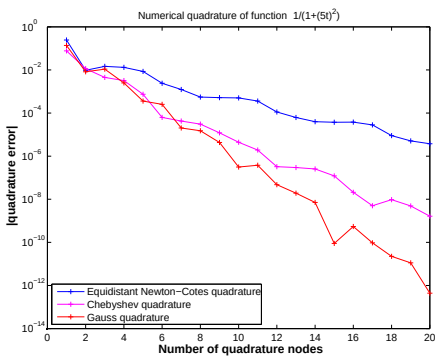


*Example* 7.1.9 (Error of (non-composite) quadratures).

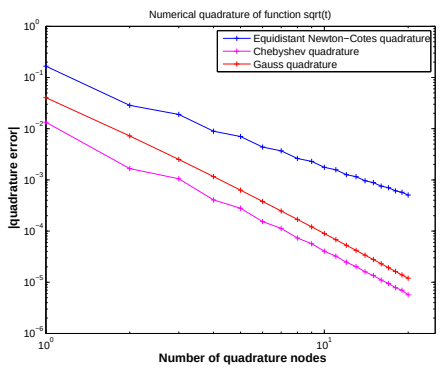
Code 7.1.10: important polynomial quadrature rules

```
from gaussquad import gaussquad
from numpy import *

def numquad(f,a,b,N,mode='equidistant'):
        """Numerical quadrature on [a,b] by polynomial quadrature
          formula
        f -> function to be integrated (handle)
        a,b -> integration interval [a,b] (endpoints included)
        N -> Maximal degree of polynomial
        mode (equidistant, Chebychev = Clenshaw-Curtis, Gauss) selects
  quadrature rule
        """
        # use a dictionary as "switch" statement:
        quadrule = {'gauss':quad_gauss, 'equidistant':quad_equidistant,
          'chebychev':quad_chebychev}
        nvals = range(1,N+1); res = []
        try:
                for n in nvals:
                        res.append(quadrule[mode.lower()](f,a,b,n))
        except KeyError:
                print "invalid quadrature type!"
        else:
                return (nvals,res)

def quad_gauss(f,a,b,deg):
        # get Gauss points for [-1,1]
        [gx,w] = gaussquad(deg);
        # transform to [a,b]
        x = 0.5*(b-a)*gx+0.5*(a+b)
        y = f(x)
        return 0.5*(b-a)*dot(w,y)

def quad_equidistant(f,a,b,deg):
        p = arange(deg+1.0,0.0,-1.0)
        w = (power(b,p) - power(a,p))/p
        x = linspace(a,b,deg+1)
        y = f(x)
        # "Quick and dirty" implementation through polynomial interpolation
        poly = polyfit(x,y,deg)
        return dot(w,poly)
```

```
def quad_chebychev(f,a,b,deg):
        p = arange(deg+1.0,0.0,-1.0)
        w = (power(b,p) - power(a,p))/p
        x = 0.5*(b-a)*cos((arange(0,deg+1)+0.5)/(deg+1)*pi)+0.5*(a+b);
        y = f(x)
        # "Quick and dirty" implementation through polynomial interpolation
        poly = polyfit(x,y,deg)
        return dot(w,poly)
```




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 $\epsilon_n := \left|\int_0^1 f(t)\,\mathrm{d}t - Q_n(f)\right|$ for "$n \to \infty$":

➢ exponential convergence $\epsilon_n \approx O(q^n)$, $0 < q < 1$, for $C^\infty$-integrand $f_1$ $\rightsquigarrow$ : Newton-Cotes quadrature : $q \approx 0.61$, Clenshaw-Curtis quadrature : $q \approx 0.40$, Gauss-Legendre quadrature : $q \approx 0.27$





➢ algebraic convergence $\epsilon_n \approx O(n^{-\alpha})$, $\alpha > 0$, for integrand $f_2$ with singularity at $t = 0$ $\rightsquigarrow$ Newton-Cotes quadrature : $\alpha \approx 1.8$, Clenshaw-Curtis quadrature : $\alpha \approx 2.5$, Gauss-Legendre quadrature : $\alpha \approx 2.7$

Code 7.1.11: tracking errors on quadrature rules

```
from numquad import numquad
import matplotlib.pyplot as plt
from numpy import *

def numquaderrs():
        """Numerical quadrature on [0,1]"""
        N = 20;

        plt.figure()
        exact = arctan(5)/5;
        f = lambda x: 1./(1+power(5.0*x,2))
        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.semilogy(nvals,abs(eqdres-exact),'b+-',label='Equidistant Newton-Cotes quadrature')
        plt.semilogy(nvals,abs(chbres-exact),'m+-',label='Clenshaw-Curtis 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
```

```
            quadrature')
        plt.axis([1, 25, 0.000001, 1]);
        plt.title('Numerical quadrature of function sqrt(t)')
        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/numquaderr2.eps")

if "__name__" == "__main__":
        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 $a = x_0 < x_1 < \cdots < x_{m-1} < x_m = b$

$$\int_a^b f(t)\,\mathrm{d}t = \sum_{j=1}^{m} \int_{x_{j-1}}^{x_j} f(t)\,\mathrm{d}t\,. \tag{7.2.1}$$

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: Increasse 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) $\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_m = b\}$
- Apply quadrature formulas on sub-intervals $I_j := [x_{j-1}, x_j]$, $j = 1, \ldots, m$, and sum up.

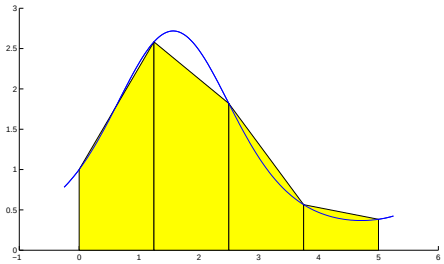
▶ composite quadrature rule

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).

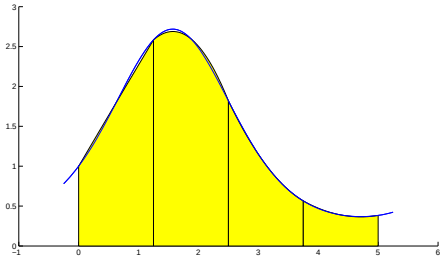
Composite trapezoidal rule, *cf.* (8.6.2)

$$\int_a^b f(t)\mathrm{d}t = \tfrac{1}{2}(x_1 - x_0)f(a) + \sum_{j=1}^{m-1} \tfrac{1}{2}(x_{j+1} - x_{j-1})f(x_j) + \tfrac{1}{2}(x_m - x_{m-1})f(b)\,. \tag{7.2.2}$$



Composite Simpson rule, *cf.* (7.1.4)

$$\int_a^b f(t)\mathrm{d}t = \tfrac{1}{6}(x_1 - x_0)f(a) + \sum_{j=1}^{m-1} \tfrac{1}{6}(x_{j+1} - x_{j-1})f(x_j) + \sum_{j=1}^{m} \tfrac{2}{3}(x_j - x_{j-1})f(\tfrac{1}{2}(x_j + x_{j-1})) + \tfrac{1}{6}(x_m - x_{m-1})f(b)\,. \tag{7.2.3}$$



Formulas (7.2.2), (7.2.3) directly suggest efficient implementation with minimal number of $f$-evaluations.

◇

Focus: *asymptotic* behavior of quadrature error for

mesh width $h := \max_{j=1,\ldots,m} |x_j - x_{j-1}| \to 0$

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*

$$\exists C > 0: \quad \left| \int_I f(t)\,\mathrm{d}t - Q(f) \right| \leq C h^p \left\| f^{(p)} \right\|_{L^\infty(I)} \qquad \forall f \in C^p(I), \forall \mathcal{M}\,.$$

*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) ➢ local order 2 (exact for linear functions),
- Simpson rule (7.1.4) ➢ 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)

```
def trapezoidal(func,a,b,N):
        """
        Numerical quadrature based on trapezoidal rule
        func: handle to y = f(x)
        a,b: bounds of integration interval
        N+1: number of equidistant quadrature points
        """

        from numpy import linspace, sum
        # quadrature nodes
        x = linspace(a,b,N+1); h = x[1]-x[0]
        # quadrature weights: internal nodes: w=1, boundary nodes: w=0.5
        I = sum(func(x[1:-1])) + 0.5*(func(x[0])+func(x[-1]))
        return I*h

if __name__ == "__main__":
        import matplotlib.pyplot as plt
        from scipy import integrate
        from numpy import array, linspace, size, log, polyfit

        # define a function and an interval:
        f = lambda x: x**2
        left = 0.0; right = 1.0

        # exact integration with scipy.integrate.quad:
        exact,e = integrate.quad(f,left,right)
        # trapezoid rule for different number of quadrature points
        N = linspace(2,101,100)
        res = array(N) # preallocate same amount of space as N uses
        for i in xrange(size(N)):
                res[i] = trapezoidal(f,left,right,N[i])
        err = abs(res - exact)
        #plt.loglog(N,err,'o')
        #plt.show()

        # linear fit to determine convergence order
        p = polyfit(log(N),log(err),1)
        # output the convergence order
        print "convergence order:",-p[0]
```

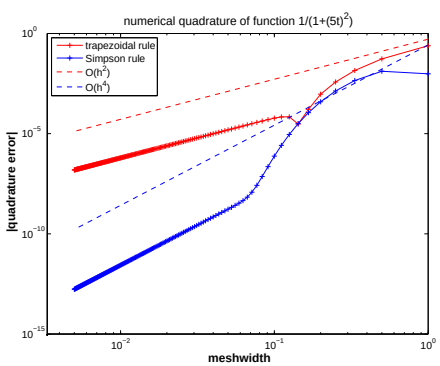


Code 7.2.4: composite Simpson rule (7.2.3)

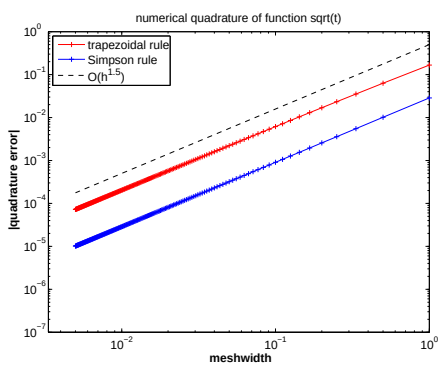
```
from numpy import linspace, sum, size

def simpson(func,a,b,N):
        """
        Numerical quadrature based on Simpson rule
        func: handle to y = f(x)
        a,b: bounds of integration interval
        N+1: number of equidistant quadrature points
        """

        # ensure that we have an even number of subintervals
        if N%2 == 1: N = N+1
        # quadrature nodes
        x = linspace(a,b,N+1); h = x[1]-x[0]
        # quadrature weights:
        # internal nodes: even: w=2/3, odd: w=4/3
        # boundary nodes: w=1/6
        I = h*sum(func(x[0:-2:2]) + 4*func(x[1:-1:2]) +
          func(x[2::2]))/3.0
        return I

if __name__ == "__main__":
        import matplotlib.pyplot as plt
```

```
from scipy import integrate
from numpy import array, linspace, size, log, polyfit

# define a function and an interval:
f = lambda x: 1./(1+(5*x)**2)
left = 0.0; right = 1.0

# exact integration with scipy.integrate.quad:
exact,e = integrate.quad(f,left,right)
# trapezoid rule for different number of quadrature points
N = linspace(2,101,100)
res = array(N) # preallocate same ammount of space as N uses
for i in xrange(size(N)):
        res[i] = simpson(f,left,right,N[i])
err = abs(res - exact)
#plt.loglog(N,err,'o')
#plt.show()

# linear fit to determine convergence order
p = polyfit(log(N[-20:]),log(err[-20:]),1)
# output the convergence order
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 $E(n) := \left|\int_0^1 f(t)\,\mathrm{d}t - Q_n(f)\right|$ for meshwidth "$h \to 0$"

☛ algebraic convergence $E(n) = O(h^\alpha)$ of order $\alpha > 0$, $n = h^{-1}$



➢ Sufficiently smooth integrand $f_1$: trapezoidal rule $\to \alpha = 2$, Simpson rule $\to \alpha = 4$ !?

➢ 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);
```

$$err := \left(\frac{1}{90}\left(D^{(4)}\right)(f)(0)\,h^5 + O\left(h^6\right), h, 6\right)$$

➢ Simpson rule is of order 4, indeed **!**



Code 7.2.5: errors of composite trapezoidal and Simpson rule

```
#!/usr/bin/env python

import numpy as np
import matplotlib.pyplot as plt
```



```
from scipy import integrate
# own integrators:
from trapezoidal import trapezoidal
from simpson import simpson

# integration functions:
integrators = [trapezoidal, simpson]
intNames = ('trapezoidal','simpson')

# define a few different Ns
N = np.linspace(2,201,200)

# define a function...
f = lambda x: 1./(1+(5*x)**2)
#f = lambda x: x**2
# ...and an interval
left=0.0; right=1.0

# "exact" integration with scipy function:
exact,e = integrate.quad(f,left,right,epsabs=1e-12)

# our versions
err = []; res = []
```

```
for int in integrators:
        for i in xrange(np.size(N)):
                res.append(int(f,left,right,N[i]))
        err.append(np.abs(np.array(res)-exact))
        res = []

plt.figure()
# evaluation
logN=np.log(N)
for i in xrange(np.size(integrators)):
        # linear fit to determine convergence orders
        p = np.polyfit(logN[-20:],np.log(err[i][-20:]),1) # only look at last 20 entries- asymptotic!
        # plot errors
        plt.loglog(N,err[i],'o', label=intNames[i])
        # plot linear fitting
        x = np.linspace(min(logN),max(logN),10)
        y = np.polyval(p,x)
        plt.loglog(np.exp(x),np.exp(y),label="linear fit: m="+str(-p[0])[:4])
        # output the convergence order
        print "convergence order of "+intNames[i]+":",-p[0]

plt.xlabel("log(N)"); plt.ylabel("log(err)")
```



```
plt.legend(loc="lower left")
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 $h \to 0$.

Idea: recover integral with smooth integrand by "analytic preprocessing"

Here is an example:

For $f \in C^\infty([0,b])$ compute $\int_0^b \sqrt{t} f(t)\,\mathrm{d}t$ via quadrature rule (→ Ex. 7.2.2)



substitution $s = \sqrt{t}$: $$\int_0^b \sqrt{t} f(t)\,\mathrm{d}t = \int_0^{\sqrt{b}} \boxed{2s^2 f(s^2)}\,\mathrm{d}s\,. \tag{7.2.4}$$

Then: Apply quadrature rule to smooth integrand

△

*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)

$$\int_a^b f(t)\,\mathrm{d}t \approx T_m(f) := h\left(\tfrac{1}{2}f(a) + \sum_{k=1}^{m-1} f(kh) + \tfrac{1}{2}f(b)\right)\,, \quad h := \frac{b-a}{m}\,. \tag{7.2.5}$$

Code 7.2.8: equidistant trapezoidal quadrature formula

```
def trapezoidal(func,a,b,N):
        """
        Numerical quadrature based on trapezoidal rule
        func: handle to y = f(x)
        a,b: bounds of integration interval
        N+1: number of equidistant quadrature points
        """

        from numpy import linspace, sum
        # quadrature nodes
        x = linspace(a,b,N+1); h = x[1]-x[0]
        # quadrature weights: internal nodes: w=1, boundary nodes: w=0.5
        I = sum(func(x[1:-1])) + 0.5*(func(x[0])+func(x[-1]))
        return I*h

if __name__ == "__main__":
        import matplotlib.pyplot as plt
        from scipy import integrate
        from numpy import array, linspace, size, log, polyfit

        # define a function and an interval:
        f = lambda x: x**2
        left = 0.0; right = 1.0

        # exact integration with scipy.integrate.quad:
        exact,e = integrate.quad(f,left,right)
        # trapezoid rule for different number of quadrature points
        N = linspace(2,101,100)
        res = array(N) # preallocate same amount of space as N uses
```

```
# define the function: (0 < a < 1)
f = lambda x: 1./sqrt(1-a*sin(2*pi*x+1))
avals = [0.5, 0.9, 0.95, 0.99];

left = 0
# first interval: [0, 0.5]
err = []; right = 0.5
# loop over different values of a:
for a in avals:
        # exact integration with scipy.integrate.quad:
        exact,e = integrate.quad(f,left,right)
        # trapezoid rule for different number of quadrature points
        N = linspace(2,50,49)
        res = array(N) # preallocate same amount of space as N uses
        for i in xrange(size(N)):
                res[i] = trapezoidal(f,left,right,N[i])
        err.append( abs(res - exact) )
plt.figure()
plt.xlabel('N')
plt.ylabel('error')
plt.title(u'Trapezoidal rule quadrature for non-periodic function')
for i in xrange(size(avals)): plt.loglog(N,err[i],'-o',label='a= '+str(avals[i]))
plt.legend(loc="lower left")
plt.show()
#plt.savefig("../PICTURES/traperr2.eps")

# second interval: [0, 1]
err = []; right = 1
# loop over different values of a:
for a in avals:
        # exact integration with scipy.integrate.quad:
        exact,e = integrate.quad(f,left,right)
        # trapezoid rule for different number of quadrature points
        N = linspace(2,50,49)
        res = array(N) # preallocate same amount of space as N uses
        for i in xrange(size(N)):
                res[i] = trapezoidal(f,left,right,N[i])
        err.append( abs(res - exact) )
plt.figure()
plt.xlabel('N')
plt.ylabel('error')
plt.title(r"Trapezoidal rule quadrature for $(1-a\cdot\sin(2\pi x+1))^{\frac{-1}{2}}$")
for i in xrange(size(avals)): plt.loglog(N,err[i],'-o',label='a= '+str(avals[i]))
```





```
        for i in xrange(size(N)):
                res[i] = trapezoidal(f,left,right,N[i])
        err = abs(res - exact)
        #plt.loglog(N,err,'o')
        #plt.show()

        # linear fit to determine convergence order
        p = polyfit(log(N),log(err),1)
        # output the convergence order
        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 $T_{500}$)

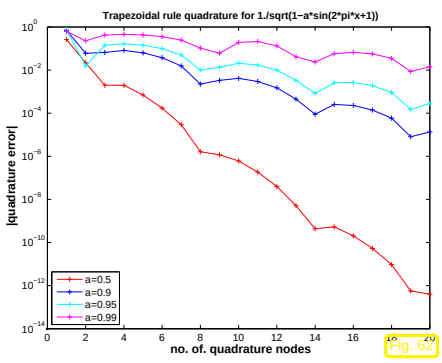



Fig. 62

quadrature error for $T_n(f)$ on $[0,1]$

exponential convergence !!

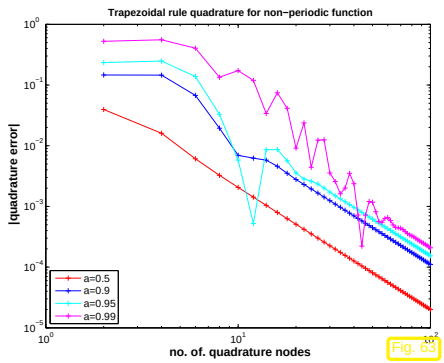



Fig. 63

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

```
import matplotlib.pyplot as plt
from scipy import integrate
from trapezoidal import trapezoidal
from numpy import *
```

```
plt.legend(loc="lower left")
plt.show()
#plt.savefig("../PICTURES/traperr1.eps")
```

Explanation:

$$f(t) = e^{2\pi i k t} \quad \blacktriangleright \quad \begin{cases} \int_0^1 f(t)\,\mathrm{d}t = \begin{cases} 0 & \text{, if } k \neq 0\,, \\ 1 & \text{, if } k = 0\,. \end{cases} \\ T_m(f) = \frac{1}{m}\sum_{l=0}^{m-1} e^{\frac{2\pi i}{m} l k} \overset{(6.1.2)}{=} \begin{cases} 0 & \text{, if } k \notin m\mathbb{Z}\,, \\ 1 & \text{, if } k \in m\mathbb{Z}\,. \end{cases} \end{cases}$$

$\blacktriangleright$

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 $f \in C^2([a,b])$):

$$\int_{x_{k-1}}^{x_k} f(t)\,\mathrm{d}t - \frac{1}{2}(f(x_{k-1}) + f(x_k)) \leq (x_k - x_{k-1})^3 \left\| f'' \right\|_{L^\infty([x_{k-1},x_k])} .$$

➢ Do not use equidistant mesh !

Refine $\mathcal{M}$, where $|f''|$ large !

Makes sense, e.g., for "spike function" ▷

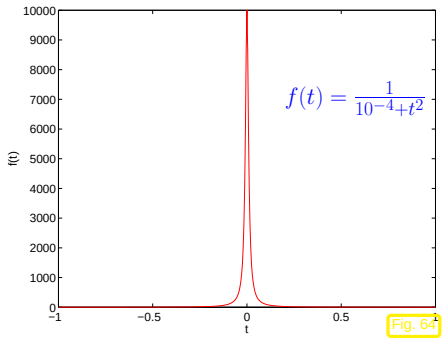



Fig. 64

◇



| | |
|---|---|
| *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 |

$\blacktriangleright$ *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: error$(Q_2) \ll$ error$(Q_1)$ ⇒ error$(Q_1) \approx Q_2(f) - Q_1(f)$ .

Now: $Q_1$ = trapezoidal rule (order 2) ↔ $Q_2$ = Simpson rule (order 4)

Given: mesh $\mathcal{M} := \{a = x_0 < x_1 < \cdots < x_m = b\}$

❶ (error estimation)

For $I_k = [x_{k-1}, x_k]$, $k = 1, \ldots, m$ (midpoints $p_k := \frac{1}{2}(x_{k-1} + x_k)$ )

$$\mathtt{EST}_k := \Bigg| \underbrace{\frac{h_k}{6}(f(x_{k-1}) + 4f(p_k) + f(x_k))}_{\text{Simpson rule}} - \underbrace{\frac{h_k}{4}(f(x_{k-1}) + 2f(p_k) + f(x_k))}_{\text{trapezoidal rule on split mesh interval}} \Bigg| . \tag{7.3.1}$$

❷ (Termination)

Simpson rule on $\mathcal{M}$ ⇒ preliminary result $I$

If $\displaystyle\sum_{k=1}^{m} \mathtt{EST}_k \leq \mathtt{RTOL} \cdot I$ ($RTOL$ := prescribed tolerance) ⇒ **STOP** (7.3.2)

❸ (local mesh refinement)

$$\mathcal{S} := \{k \in \{1,\ldots,m\} \colon \mathtt{EST}_k \geq \eta \cdot \frac{1}{m}\sum_{j=1}^{m} \mathtt{EST}_j\}\,, \quad \eta \approx 0.9\,. \tag{7.3.3}$$

$\blacktriangleright$ new mesh: $\mathcal{M}^* := \mathcal{M} \cup \{p_k \colon k \in \mathcal{S}\}$ .

Then continue with step ❶ and mesh $\mathcal{M} \leftarrow \mathcal{M}^*$.

Non-optimal recursive implementation:

Code 7.3.2: $h$-adaptive numerical quadrature

```
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)                                  # compute lengths of mesh intervals
14         mp = 0.5*( M[:-1]+M[1:] )                    # compute midpoint positions
15         fx = f(M); fm = f(mp)                        # evaluate function at positions and midpoints
16         trp_loc = h*( fx[:-1]+2*fm+fx[1:] )/4        # local trapezoid rule
17         simp_loc= h*( fx[:-1]+4*fm+fx[1:] )/6        # local simpson rule
18         I = sum(simp_loc)                            # use simpson rule value as intermediate approximation for integral value
19         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(I) and err_tot > abstol:
23                 refcells = nonzero( est_loc > 0.9*sum(est_loc)/size(est_loc) )[0]
24                 I = adaptquad(f,sort(append(M,mp[refcells])),rtol,abstol)   # add midpoints of intervalls with large error contributions, recurse.
25         return I
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         I = adaptquad(f,M,rtol,abstol)
33         exact,e = integrate.quad(f,M[0],M[-1])
34         print 'adaptquad:',I, '"exact":',exact
35         print 'error:',abs(I-exact)
```



Comments on Code 7.3.1:

- Arguments: `f` $\hat{=}$ *handle* to function $f$, `M` $\hat{=}$ initial mesh, `rtol` $\hat{=}$ relative tolerance for termination, `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 ocal 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 $\displaystyle\int_0^1 \exp(6\sin(2\pi t))\,\mathrm{d}t$, initial mesh $\mathcal{M}_0 = \{j/10\}_{j=0}^{10}$

Algorithm: adaptive quadrature, Code 7.3.1

Tolerances: `rtol` $= 10^{-6}$, `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.

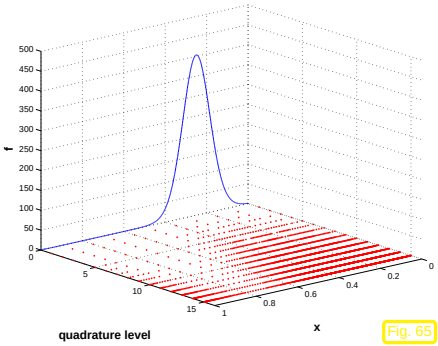



Fig. 65

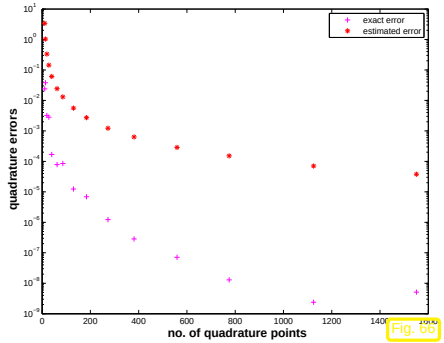



Fig. 66

- approximate $\int_0^1 \min\{\exp(6\sin(2\pi t)), 100\}\,\mathrm{d}t$, initial mesh as above

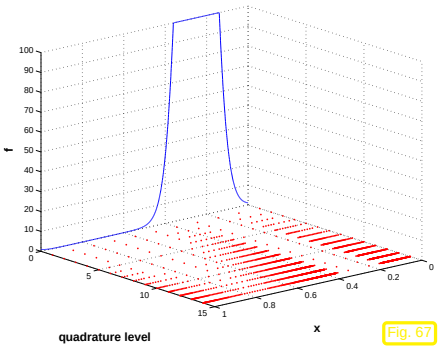



Fig. 67

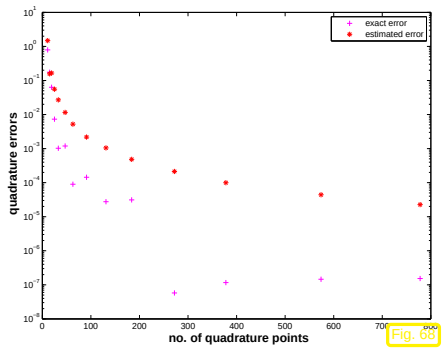



Fig. 68

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 $\mathrm{EST}_k$ 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 Q

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 easely treated by the iteration of the previous quadrature rules, e.g:

$$I = \int_a^b \int_c^d f(x,y)dxdy = \int_a^b F(y)dy,$$

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

$$\left(w_{j_k}^k, c_{j_k}^k\right)_{1 \le j_k \le n_k}, \qquad k = 1, \dots, d,$$

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

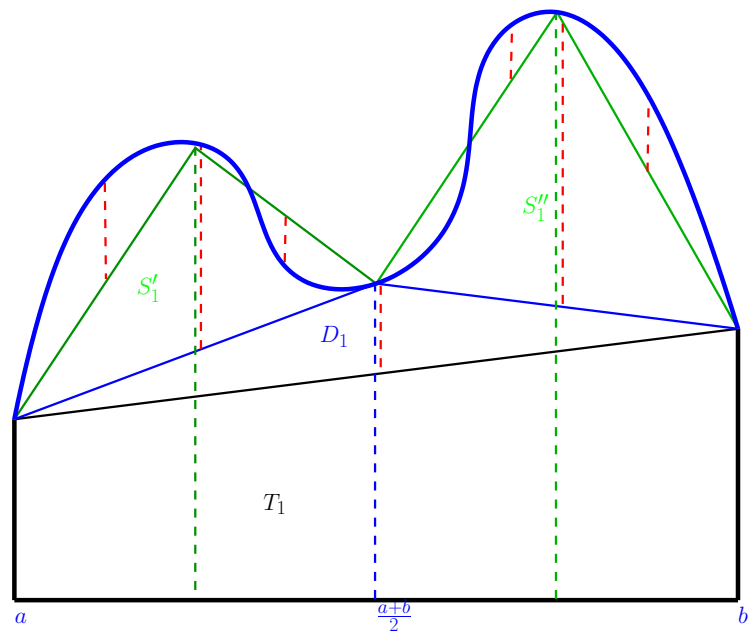
$$D_1(f,a,b) = \left(f(\frac{a+b}{2}) - \frac{f(a)+f(b)}{2}\right)\frac{b-a}{2} = g_1(f,a,b)\frac{b-a}{2}.$$

We call here $g_1(f,a,b) = f(\frac{a+b}{2}) - \frac{f(a)+f(b)}{2}$ 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 $S_1''$:

$$S_1(f,a,b) = D_1(f,a,b) + S_1(f,a,\frac{a+b}{2}) + S_1(f,\frac{a+b}{2},b).$$








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 $2^\ell$ sub-intervals $\frac{k}{2^\ell}, \frac{k+1}{2^\ell}$ for $k = 0,1,\ldots,2^\ell-1$. We call $\ell$ level and consider $Q_\ell^1$ a simple quadrature formula on each interval. Typical examples are the tapezoidal 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:

$$\begin{aligned} Q_\ell^1 f &= Q_0^1 f + (Q_1^1 f - Q_0^1 f) + (Q_2^1 f - Q_1^1 f) + \ldots + (Q_\ell^1 f - Q_{\ell-1}^1 f) \\ &= Q_0^1 f + \Delta_1^1 f + \Delta_2^2 f + \ldots + \Delta_\ell^1 f. \end{aligned}$$

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 $\ell = (\ell_1,\ldots,\ell_d)$ is

$$Q_\ell^d = Q_{\ell_1}^1 \otimes \ldots \otimes Q_{\ell_d}^1$$

$$= \sum_{j_1=1}^{N_1} \dots \sum_{j_d=1}^{N_d} w_{j_1}^1 \dots w_{j_d}^d f(c_{j_1}^1, \dots, c_{j_d}^d)$$

$$= \sum_{j=1}^{d} \sum_{1 \le k_j \le \ell_j} \left( \Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1 \right) f.$$

In the case of an isotropic grid $\ell = (\ell, \dots, \ell)$ we denote

$$Q_\ell^d = \sum_{j=1}^{d} \sum_{1 \le k_j \le \ell} \left( \Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1 \right) f = \sum_{|\mathbf{k}|_\infty \le \ell} \left( \Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1 \right) f.$$

Idea: In the case that $f$ is a smooth function, many of the details $\left( \Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1 \right) f$ are so small that they may be neglected.
The classical sparse grid quadrature (Smolyak) rule is defined by

$$S_\ell^d f := \sum_{|\mathbf{k}|_1 \le \ell + d - 1} \left( \Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1 \right) f.$$

One can prove the combinations formula:

$$S_\ell^d f = \sum_{\ell \le |\mathbf{k}|_1 \le \ell + d - 1} (-1)^{\ell + d - |\mathbf{k}|_1 - 1} \binom{d-1}{|\mathbf{k}|_1 - \ell} \left( Q_{k_1}^1 \otimes \dots \otimes Q_{k_d}^1 \right) f,$$

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 $r$.

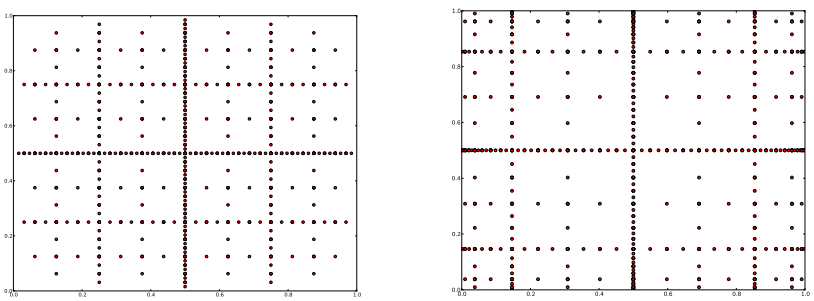


Figure 7.1: Sparse Grids based on midpoint rule and on open Clenshaw-Curtis rule

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 gird 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 $P(I \in [I_N - \sigma_N, I_N + \sigma_N]]) = 0.683$



Num. Meth. Phys. Gradinaru D-MATH 7.5 p. 41

Random variables and statistics are not subject of this lecture.

Essential: good RNG: fullfill statistical tests and is deterministic (reproductible).

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 offen 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, $A \subset R^d$,

$$\int_A f(\mathbf{x}) dx_1 dx_2 \cdots d\mathbf{x}_d \approx |A| \langle f \rangle$$

where $|A|$ is the volume of region $A$.

The error is always $\propto N^{-1/2}$ independetly 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

```
"""
 Computes integral
 I0(1) = (1/pi) int(z=0..pi)  exp(-cos(z)) dz by raw MC.
 Abramowitz and Stegun give I0(1) = 1.266066
"""

import numpy as np
import time

t1 = time.time()

M = 100 # number of times we run our MC inetgration
asval = 1.266065878
ex = np.zeros(M)
print 'A and S tables:  I0(1) = ',asval
print 'sample        variance        MC I0 val'
print '------        --------        ----------'
k = 5 # how many experiments
N = 10**np.arange(1,k+1)
v = []; e = []
for n in N:
    for m in xrange(M):
        x = np.random.rand(n) # sample
        x = np.exp(np.cos(-np.pi*x))
        ex[m] = sum(x)/n # quadrature
    ev = sum(ex)/M
    vex = np.dot(ex,ex)/M
    vex -=  ev**2
    v += [vex]; e += [ev]
    print n, vex, ev

t2 = time.time()
t = t2-t1

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 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: Antitetic Variates Monte-Carlo

```
"""
 Computes integral
 I0(1) = (1/pi) int(z=0..pi)  exp(-cos(z)) dz
 by  antithetic variates.
 We split the range into 0 < x < pi/2 and pi/2<x<pi.
 The resulting integrand is
 exp(sin(x)) + exp(-cos(x)), which is monotone
 increasing in 0 < x < pi/2, so antithetic variates
 can be used.
 Abramowitz and Stegun give I0(1) = 1.266066
"""

import numpy as np

M = 100  # number of times we run our MC inetgration
asval = 1.266065878
ex = np.zeros(M)
print 'A and S tables:  I0(1) = ',asval
print 'sample       variance       MC I0 val'
print '______       ________       _________'
k = 5  # how many experiments
N = 10**np.arange(1,k+1)
v = []; e = []
pi2 = 0.5*np.pi
for n in N:
    for m in xrange(M):
        up = pi2*np.random.rand(n)  # sample
        dn = pi2-up
        x = np.exp(np.sin(up))  + np.exp(np.sin(dn))
        x += np.exp(np.cos(-up))  + np.exp(-np.cos(dn))
        ex[m] = 0.25*sum(x)/n  # quadrature
    ev = sum(ex)/M
    vex = np.dot(ex,ex)/M
    vex -=  ev**2
    v += [vex]; e += [ev]
    print n, vex, ev
```



**Control variates** Integral

$$I = \int_0^1 f(t)dt$$

can be re-written

$$\begin{aligned} I &= \int_0^1 (f(t) - \phi(t))dt + \int_0^1 \phi(t)dt \\ &\approx \frac{1}{N}\sum_{i=1}^{N}[f(t_i) - \phi(t_i)] + I_\phi \end{aligned}$$

Pick $\phi$:

- $\phi(u) \approx g(u)$ 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

$$\begin{aligned} f(t) &= e^{-cos(\pi t)} \\ &= 1 - cos(\pi t) + \frac{1}{2}(cos(\pi t))^2 + \ldots \\ \phi(t) &= \text{first three terms} \end{aligned}$$

Code 7.5.3: Control Variates Monte-Carlo

```
"""
 Computes integral
 I0(1) = (1/pi) int(z=0..pi)  exp(-cos(z)) dz
 by  control variate. The splitting is
 < exp(-cos) >  = < exp(-cos) - phi > + int(0..pi) phi dz
 where phi = 1 + cos - (1/2)*cos*cos is the control.
 The exact integral of the control
 int(z=0..pi) (1 + cos - (1/2)*cos*cos) dz = 1.25.
 Abramowitz and Stegun give I0(1) = 1.266066
"""

import numpy as np

M = 100  # number of times we run our MC inetgration
asval = 1.266065878
ex = np.zeros(M)
```

```
print 'A and S tables:  I0(1) = ',asval
print 'sample       variance        MC I0 val'
print '------       --------        ---------'
k = 5 # how many experiments
N = 10**np.arange(1,k+1)
v = []; e = []
for n in N:
    for m in xrange(M):
        x = np.pi*np.random.rand(n) # sample
        ctv = np.exp(-np.cos(x)) - 1. + np.cos(x) -
          0.5*np.cos(x)*np.cos(x)
        ex[m] = 1.25 + sum(ctv)/n # quadrature
    ev = sum(ex)/M
    vex = np.dot(ex,ex)/M
    vex -=  ev**2
    v += [vex]; e += [ev]
    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

$$\sigma^2_{f/g} = \int_0^1 (f(x)/g(x) - \theta)^2 dG(x)$$

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!

$$g(x) = \frac{1}{4\sqrt{x}} + \frac{1}{4\sqrt{1-x}} \quad \Rightarrow \quad \int_0^1 h(x) dG(x)$$

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

```
from scipy import where, sqrt, arange, array
from numpy.random import rand
from time import time

from scipy.integrate import quad
f = lambda x: 1/sqrt(x*(1-x))
print 'quad(f,0,1) = ', quad(f,0,1)

func = lambda x: 4./(sqrt(x)+sqrt(1-x))

def exotic(N):
    u = rand(N)
    v = rand(N)
    x = u*u
    w = where(v>0.5)
    x[w] = 1 - x[w]
    return x

def ismcquad():
    k = 5 # how many experiments
    N = 10**arange(1,k+1)
    ex = []
    for n in N:
        x = exotic(n) # sample
        x = func(x)
        ex += [x.sum()/n] # quadrature
    return ex

def mcquad():
    k = 5 # how many experiments
    N = 10**arange(1,k+1)
    ex = []
    for n in N:
        x = rand(n) # sample
        x = f(x)
        ex += [x.sum()/n] # quadrature
    return ex
```

```
M = 100  # number of times we run our MC inetgration

t1 = time()
results = []
for m in xrange(M):
    results += [ismcquad()]

t2 = time()
t = t2-t1

print "ISMC Serial calculation completed, time = %s s" % t

ex = array(results)
ev = ex.sum(axis=0)/M
vex = (ex**2).sum(axis=0)/M
vex = vex - ev**2
print ev[-1]
print vex

t1 = time()
results = []
for m in xrange(M):
    results += [mcquad()]

```





```
t2 = time()
t = t2-t1

print "MC Serial calculation completed, time = %s s" % t

ex = array(results)
ev = ex.sum(axis=0)/M
vex = (ex**2).sum(axis=0)/M
vex = vex - ev**2
print ev[-1]
print vex
```

. Compare with the analytical computed value.



## 7.6 Essential Skills Learned in Chapter 7

You should know:

- several (compozite) 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].

*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

$$\dot{y} = f(y) := (\alpha - \beta y)\, y \tag{8.1.1}$$



Notation (Newton): dot $\dot{}$ $\hat{=}$ (total) derivative with respect to time $t$

- $y \hat{=}$ population density, $[y] = \frac{1}{\mathrm{m}^2}$
- growth rate $\alpha - \beta y$ with growth coefficients $\alpha, \beta > 0$, $[\alpha] = \frac{1}{\mathrm{s}}$, $[\beta] = \frac{\mathrm{m}^2}{\mathrm{s}}$: 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 logisitc 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):

$$A + B \longrightarrow 2B \quad \text{with rate } r = k c_A c_B \tag{8.1.2}$$

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

$$\dot{c}_A = -k(D - c_A)c_A \tag{8.1.3}$$



$$\dot{c}_B = k(D - c_B)c_B \tag{8.1.4}$$

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 $A$.

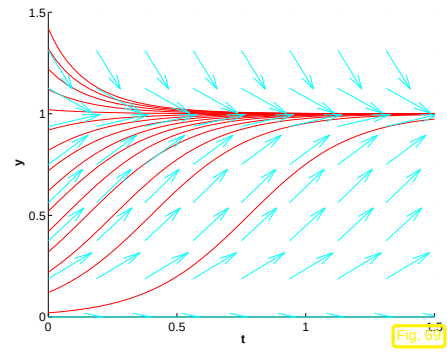



Fig. 69

Solution for different $y(0)$ ($\alpha, \beta = 5$)

By separation of variables
➥ solution of (8.1.3)
for $y(0) = y_0 > 0$

$$y(t) = \frac{\alpha y_0}{\beta y_0 + (\alpha - \beta y_0)\exp(-\alpha t)}\,, \tag{8.1.5}$$

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:

$$\begin{aligned} \dot{u} &= (\alpha - \beta v)u \\ \dot{v} &= (\delta u - \gamma)v \end{aligned} \quad \leftrightarrow \quad \dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad \text{with} \quad \mathbf{y} = \begin{pmatrix} u \\ v \end{pmatrix}, \quad \mathbf{f}(\mathbf{y}) = \begin{pmatrix} (\alpha - \beta v)u \\ (\delta u - \gamma)v \end{pmatrix}. \tag{8.1.6}$$

population sizes:

$u(t) \longrightarrow$ no. of prey at time $t$,
$v(t) \longrightarrow$ no. of predators at time $t$

vector field $f$ for Lotka-Volterra ODE ▷

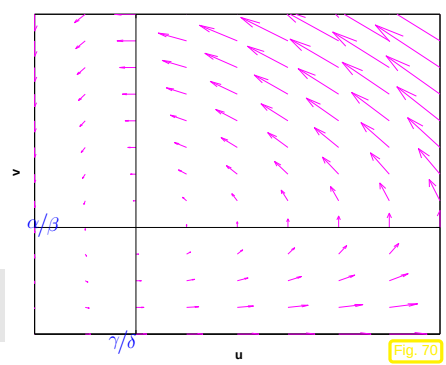
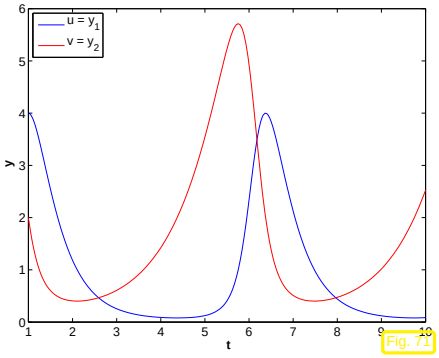


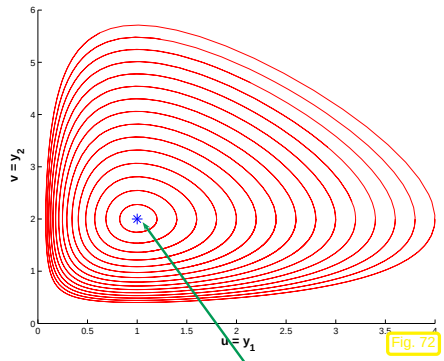

Fig. 70

Solution curves are trajectories of particles carried along by velocity field $\mathbf{f}$.

Parameter values for Fig. 70: $\alpha = 2, \beta = 1, \delta = 1, \gamma = 1$




Solution $\binom{u(t)}{v(t)}$ for $\mathbf{y}_0 := \binom{u(0)}{v(0)} = \binom{4}{2}$




Solution curves for (8.1.6)

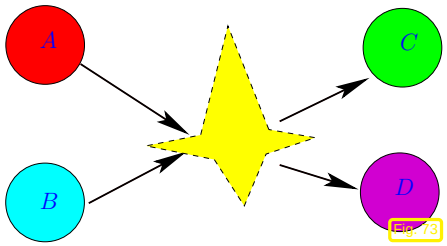
stationary point

Parameter values for Figs. 72, 71: $\alpha = 1, \beta = 1, \delta = 1, \gamma = 2$

◇



*Example* 8.1.3 (Bimolecular Reaction).




Reaction: $\mathrm{A} + \mathrm{B} \underset{k_1}{\overset{k_2}{\rightleftharpoons}} \mathrm{C} + \mathrm{D}$ . (8.1.7)

with reaction constants $k_1$ ("forwards"), $k_2$ ("backwards"), $[k_1] = [k_2] = \frac{\mathrm{cm}^3}{\mathrm{mol\,s}}$.

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)

$c_A, c_B, c_C, c_D \mathrel{\hat{=}}$ (time dependent) concentrations of components, $[c_X] = \frac{\mathrm{mol}}{\mathrm{cm}^3}$ ➥ $c_X(t) > 0; \forall t$

(8.1.8) = autonom ordinary differential equation (**??**) with

$$\mathbf{y}(t) = \begin{pmatrix} c_A(t) \\ c_B(t) \\ c_C(t) \\ c_D(t) \end{pmatrix} \quad , \quad \mathbf{f}(t, \mathbf{y}) = (-k_1 y_1 y_2 + k_2 y_3 y_4) \begin{pmatrix} 1 \\ 1 \\ -1 \\ -1 \end{pmatrix} .$$

▶ 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]:

$$\begin{aligned} \mathrm{BrO_3^-} + \mathrm{Br^-} &\mapsto \mathrm{HBrO_2} \\ \mathrm{HBrO_2} + \mathrm{Br^-} &\mapsto \mathrm{Org} \\ \mathrm{BrO_3^-} + \mathrm{HBrO_2} &\mapsto 2\,\mathrm{HBrO_2} + \mathrm{Ce(IV)} \\ 2\,\mathrm{HBrO_2} &\mapsto \mathrm{Org} \\ \mathrm{Ce(IV)} &\mapsto \mathrm{Br^-} \end{aligned} \tag{8.1.9}$$

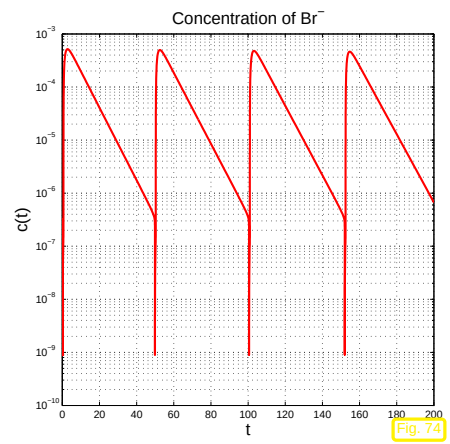
$$\begin{aligned} y_1 &:= c(\mathrm{BrO_3^-}): & \dot{y}_1 &= -k_1 y_1 y_2 - k_3 y_1 y_3 \,, \\ y_2 &:= c(\mathrm{Br^-}): & \dot{y}_2 &= -k_1 y_1 y_2 - k_2 y_2 y_3 + k_5 y_5 \,, \\ y_3 &:= c(\mathrm{HBrO_2}): & \dot{y}_3 &= k_1 y_1 y_2 - k_2 y_2 y_3 + k_3 y_1 y_3 - 2k_4 y_3^2 \,, \\ y_4 &:= c(\mathrm{Org}): & \dot{y}_4 &= k_2 y_2 y_3 + k_4 y_3^2 \,, \\ y_5 &:= c(\mathrm{Ce(IV)}): & \dot{y}_5 &= k_3 y_1 y_3 - k_5 y_5 \,, \end{aligned} \tag{8.1.10}$$

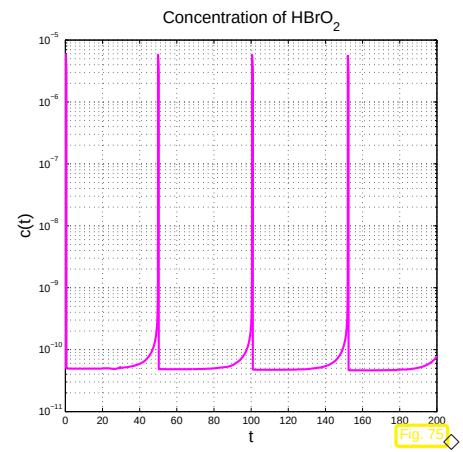
with (dimensionless) reaction constants:

$$k_1 = 1.34 \,, \quad k_2 = 1.6 \cdot 10^9 \,, \quad k_3 = 8.0 \cdot 10^3 \,, \quad k_4 = 4.0 \cdot 10^7 \,, \quad k_5 = 1.0 \,.$$

▶ 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): (→ [52, Sect. 5.6])

$$\dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0 \, . \tag{8.1.11}$$

- $\mathbf{f} : I \times D \mapsto \mathbb{R}^d \mathrel{\hat{=}}$ right hand side (r.h.s.) ($d \in \mathbb{N}$), given in procedural form

```
function v = f(t,y).
```

- $I \subset \mathbb{R} \mathrel{\hat{=}}$ (time)interval $\leftrightarrow$ "time variable" $t$
- $D \subset \mathbb{R}^d \mathrel{\hat{=}}$ state space/phase space $\leftrightarrow$ "state variable" $\mathbf{y}$ (*ger.:* Zustandsraum)
- $\Omega := I \times D \mathrel{\hat{=}}$ extended state space (of tupels $(t,\mathbf{y})$)
- $t_0 \mathrel{\hat{=}}$ initial time, $\mathbf{y}_0 \mathrel{\hat{=}}$ initial state ➢ initial conditions

Terminology: $\mathbf{f} = \mathbf{f}(\mathbf{y})$, r.h.s. does not depend on time ➔ $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ 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 $= 1$

$$\mathbf{z}(t) := \begin{pmatrix} \mathbf{y}(t) \\ t \end{pmatrix} = \begin{pmatrix} \mathbf{z}' \\ z_{d+1} \end{pmatrix} : \quad \dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y}) \quad \leftrightarrow \quad \dot{\mathbf{z}} = \mathbf{g}(\mathbf{z}) \, , \quad \mathbf{g}(\mathbf{z}) := \begin{pmatrix} \mathbf{f}(z_{d+1},\mathbf{z}') \\ 1 \end{pmatrix} \, .$$

△

*Remark* 8.1.6 (From higher order ODEs to first order systems).

Ordinary differential equation of order $n \in \mathbb{N}$:

$$\boxed{\mathbf{y}^{(n)} = \mathbf{f}(t,\mathbf{y},\dot{\mathbf{y}},\dots,\mathbf{y}^{(n-1)})} \, . \tag{8.1.12}$$

✎ Notation: superscript $^{(n)} \mathrel{\hat{=}} n$-th temporal derivative $t$



➢ Conversion into 1st-order ODE (system of size $nd$)

$$\mathbf{z}(t) := \begin{pmatrix} \mathbf{y}(t) \\ \mathbf{y}^{(1)}(t) \\ \vdots \\ \mathbf{y}^{(n-1)}(t) \end{pmatrix} = \begin{pmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \\ \vdots \\ \mathbf{z}_n \end{pmatrix} \in \mathbb{R}^{dn} : \quad (8.1.12) \quad \leftrightarrow \quad \dot{\mathbf{z}} = \mathbf{g}(\mathbf{z}) \, , \quad \mathbf{g}(\mathbf{z}) := \begin{pmatrix} \mathbf{z}_2 \\ \mathbf{z}_3 \\ \vdots \\ \mathbf{z}_n \\ \mathbf{f}(t,\mathbf{z}_1,\dots,\mathbf{z}_n) \end{pmatrix} \, . \tag{8.1.13}$$

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). *(→ [52, Def. 4.1.4])*
$\mathbf{f} : \Omega \mapsto \mathbb{R}^d$ *is Lipschitz continuous (in the second argument), if*

$$\exists L > 0 : \quad \|\mathbf{f}(t,\mathbf{w}) - \mathbf{f}(t,\mathbf{z})\| \le L \, \|\mathbf{w} - \mathbf{z}\| \quad \forall (t,\mathbf{w}),(y,\mathbf{z}) \in \Omega \, .$$



**Definition 8.1.2** (Local Lipschitz continuity). *(→ [52, Def. 4.1.5])*



✎ Notation: $D_{\mathbf{y}}\mathbf{f} \mathrel{\hat{=}}$ 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 Liptschitz continuity).
*If $\mathbf{f}$ and $D_{\mathbf{y}}\mathbf{f}$ are continuous on the extended state space $\Omega$, then $\mathbf{f}$ is locally Lipschitz continuous(→ 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 (→ 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 $J(t_0, \mathbf{y}_0) = I$ ➥ 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

$$\text{autonomous IVP:} \quad \boxed{\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad , \quad \mathbf{y}(0) = \mathbf{y}_0} \; , \tag{8.1.14}$$

with locally Lipschitz continuous (→ 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 $\mathbf{\Phi}^t : \begin{cases} D & \mapsto & D \\ \mathbf{y}_0 & \mapsto & \mathbf{y}(t) \end{cases}$ , $t \mapsto \mathbf{y}(t)$ 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 $\{\mathbf{\Phi}^t\}$ 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*

$$\mathbf{\Phi} : \begin{cases} \mathbb{R} \times D & \mapsto & D \\ (t, \mathbf{y}_0) & \mapsto & \mathbf{\Phi}^t \mathbf{y}_0 := \mathbf{y}(t) \end{cases} ,$$

*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: $t \mapsto \mathbf{\Phi}^t \mathbf{y}_0$ 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 $D \mapsto D$:

$$\mathbf{\Phi}^s \circ \mathbf{\Phi}^t = \mathbf{\Phi}^{s+t} \quad , \quad \mathbf{\Phi}^{-t} \circ \mathbf{\Phi}^t = Id \quad \forall t \in \mathbb{R} \; . \tag{8.1.15}$$

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)

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0 \; . \tag{8.1.11}$$



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.

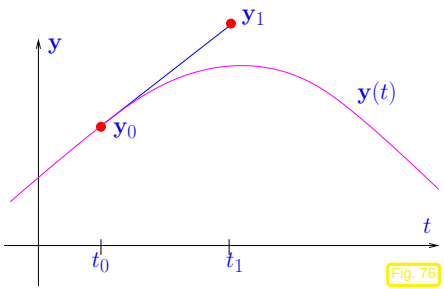
numerical integration = approximate solution of initial value problems for ODEs

(Please distinguish from "numerical quadrature", see Ch. 7.)



Idea: ❶ timestepping: successive approximation of evolution on *small* intervals $[t_{k-1}, t_k]$, $k = 1, \ldots, N$, $t_N := T$,

❷ approximation of solution on $[t_{k-1}, t_k]$ by tangent curve to current initial condition.




*Example* 8.2.1 (Visualization of explicit Euler method).

explicit Euler method (Euler 1768)

◁ First step of explicit Euler method ($d = 1$):

Slope of tangent $= f(t_0, \mathbf{y}_0)$

$\mathbf{y}_1$ serves as initial value for next step!

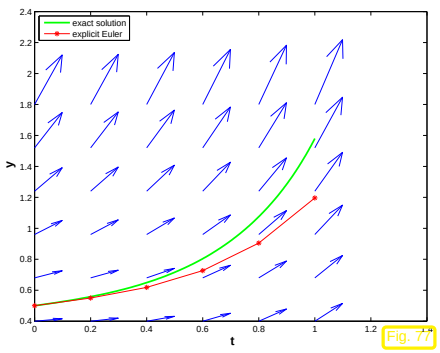
IVP for Riccati differential equation, see Ex. **??**

$$\dot{y} = y^2 + t^2 \,. \qquad \textbf{(??)}$$

Here: $y_0 = \frac{1}{2}$, $t_0 = 0$, $T = 1$, ▷

— $\hat{=}$ "Euler polygon" for uniform timestep $h = 0.2$

$\mapsto$ $\hat{=}$ tangent field of Riccati ODE




◇

Formula: explicit Euler method generates a *sequence* $(\mathbf{y}_k)_{k=0}^N$ by the recursion

$$\boxed{\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_k, \mathbf{y}_k) \,, \quad k = 0, \dots, N-1 \quad ,} \qquad (8.2.1)$$

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 $\mathcal{M} := \{t_0, t_1, \dots, t_N\}$:

$$\dot{\mathbf{y}} = f(t, \mathbf{y}) \quad \longleftrightarrow \quad \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} = f(t_k, \mathbf{y}_h(t_k)) \,, \quad k = 0, \dots, N-1 \,. \qquad (8.2.2)$$

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

$$\dot{\mathbf{y}} = f(t, \mathbf{y}) \quad \longleftrightarrow \quad \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} = f(t_{k+1}, \mathbf{y}_h(t_{k+1})) \,, \quad k = 0, \dots, N-1 \,. \qquad (8.2.3)$$

Backward difference quotient

This leads to another simple timestepping scheme analoguous to (8.2.1):

$$\boxed{\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1}) \,, \quad k = 0, \dots, N-1} \,, \qquad (8.2.4)$$

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:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1}) \quad \Leftrightarrow \quad G(h, \mathbf{y}_{k+1}) = 0 \quad \text{with} \quad G(h, \mathbf{z}) := \mathbf{z} - h\mathbf{f}(\mathbf{z}) - \mathbf{y}_k \,.$$

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: $\boxed{\mathbf{y}_k \approx \mathbf{y}(t_k)}$

(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 (→ Sect. **??**) ➔ Euler polygon, see Fig. 77.

**Abstract single step methods**

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

$$(\mathbf{y}_k, h_k) \mapsto \mathbf{\Psi}(h, \mathbf{y}_k) := \mathbf{y}_{k+1} \ . \tag{8.2.5}$$

Recall the interpretation of the $\mathbf{y}_k$ as approximations of $\mathbf{y}(t_k)$:

$$\mathbf{\Psi}(h, \mathbf{y}) \approx \mathbf{\Phi}^h \mathbf{y} \ , \tag{8.2.6}$$

where $\mathbf{\Phi}$ is the evolution operator (→ 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 $\mathbf{\Phi}$ for an ODE by a mapping of the type (8.2.5).

➔ mapping $\mathbf{\Psi}$ from (8.2.5) is called discrete evolution.

Vice versa: a mapping $\mathbf{\Psi}$ 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 and initial value problem, but tries to approximate an ODE.

**Definition 8.2.1** (Single step method (for autonomous ODE))**.**
*Given a discrete evolution* $\mathbf{\Psi} : \Omega \subset \mathbb{R} \times D \mapsto \mathbb{R}^d$*, an initial state* $\mathbf{y}_0$*, and a temporal mesh* $\mathcal{M} := \{t_0 < t_1 < \cdots < t_N = T\}$ *the recursion*

$$\mathbf{y}_{k+1} := \mathbf{\Psi}(t_{k+1} - t_k, \mathbf{y}_k) \ , \quad k = 0, \dots, N-1 \ , \tag{8.2.7}$$

*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:

$\mathbf{\Psi}^h \mathbf{y}$ ⟷ `function y1 = esvstep(h,y0)` .
( `function y1 = esvstep( rhs,h,y0)` )





✎ Notation: $\mathbf{\Psi}^h \mathbf{y} := \mathbf{\Psi}(h, \mathbf{y})$

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:

$$\mathbf{y}_{k+1} := \mathbf{\Psi}(t_k, t_{k+1}, \mathbf{y}_k) \ , \quad k = 0, \dots, N-1 \ ,$$

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:

$\mathbf{y}_1 =$ expression in $\mathbf{y}_0$ 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 $h := \max_k h_k \to 0$

Will tell us asymptotic gain in accuracy for extra computational effort.
(computational effort = no. of $\mathbf{f}$-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 $\mathrm{err}_h := |y(1) - y_N|$

Observation:

algebraic convergence $\mathrm{err}_h = O(h)$

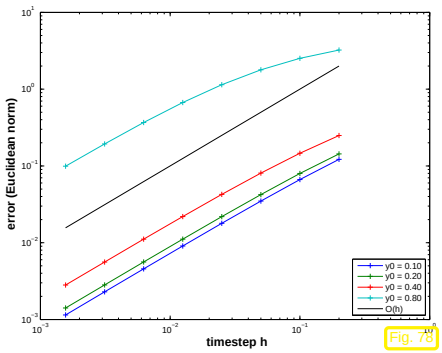


Fig. 78



- IVP for logistic ODE, see Ex. 8.1.1

$$\dot{y} = \lambda y(1-y) \quad , \quad y(0) = 0.01 \; .$$

- 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 $E(h) := |y(1) - y_N|$

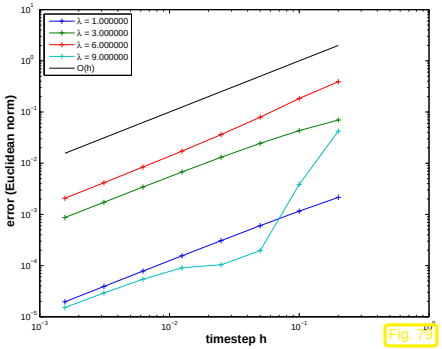


Fig. 79

explicit Euler method

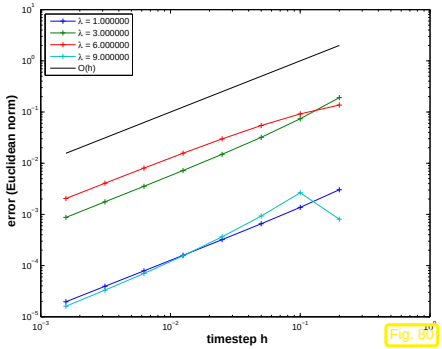


Fig. 80

implicit Euler method

$O(h)$ algebraic convergence in both cases ◇

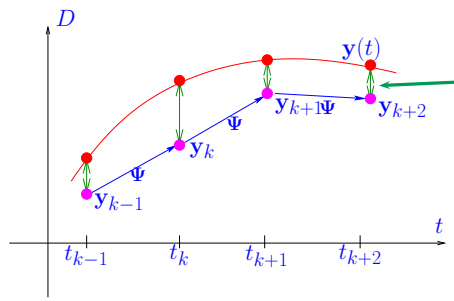


**Convergence analysis** for explicit Euler method (8.2.1) for autonomous IVP (8.1.11) with sufficiently smooth and (*globally*) Lipschitz continuous $\mathbf{f}$, that is,

$$\exists L > 0: \quad \|\mathbf{f}(t,\mathbf{y}) - \mathbf{f}(t,\mathbf{z})\| \leq L \, \|\mathbf{y} - \mathbf{z}\| \quad \forall \mathbf{y}, \mathbf{z} \in D \; . \tag{8.3.1}$$

Recall: recursion for explicit Euler method

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_k) \; , \quad k = 1, \ldots, N-1 \; . \tag{8.2.1}$$




Error sequence: $\mathbf{e}_k := \mathbf{y}_k - \mathbf{y}(t_k)$ .

◁ — $\hat{=}$ $t \mapsto \mathbf{y}(t)$
— $\hat{=}$ Euler polygon,
• $\hat{=}$ $\mathbf{y}(t_k)$,
• $\hat{=}$ $\mathbf{y}_k$,
⟶ $\hat{=}$ discrete evolution $\Psi^{t_{k+1}-t_k}$

① Abstract splitting of error:

Here and in what follows we rely on the abstract concepts of the evolution operator $\mathbf{\Phi}$ associated with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ (→ Def. 8.1.6) and discrete evolution operator $\mathbf{\Psi}$ defining the explicit Euler single step method, see Def. 8.2.1:

$$(8.2.1) \quad \Rightarrow \quad \mathbf{\Psi}^h\mathbf{y} = \mathbf{y} + h\mathbf{f}(\mathbf{y}) \; . \tag{8.3.2}$$

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

$$\begin{aligned}\mathbf{e}_{k+1} &= \mathbf{\Psi}^{h_k}\mathbf{y}_k - \mathbf{\Phi}^{h_k}\mathbf{y}(t_k) \\ &= \underbrace{\mathbf{\Psi}^{h_k}\mathbf{y}_k - \mathbf{\Psi}^{h_k}\mathbf{y}(t_k)}_{\text{propagated error}} \\ &\quad + \underbrace{\mathbf{\Psi}^{h_k}\mathbf{y}(t_k) - \mathbf{\Phi}^{h_k}\mathbf{y}(t_k)}_{\text{one-step error}} \; .\end{aligned} \tag{8.3.3}$$

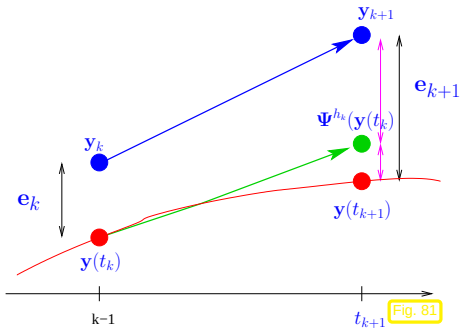


Fig. 81

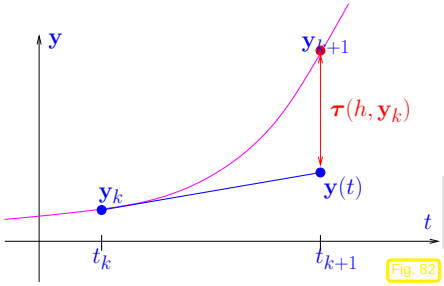


Fig. 82

one-step error:

$$\boldsymbol{\tau}(h,\mathbf{y}) := \mathbf{\Psi}^h\mathbf{y} - \mathbf{\Phi}^h\mathbf{y} \; . \tag{8.3.4}$$

◁ geometric visualisation of one-step error for explicit Euler method (8.2.1), *cf.* Fig. 76.

✎ notation: $t \mapsto \mathbf{y}(t) \hat{=}$ (unique) solution of IVP, *cf.* Thm. 8.1.4.

② **Estimate for one-step error**:

Geomtric 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).



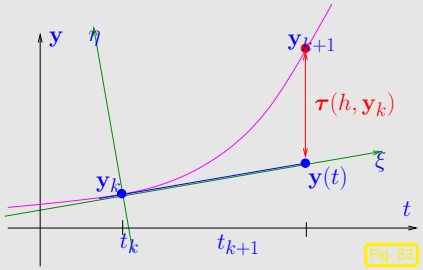


Fig. 83

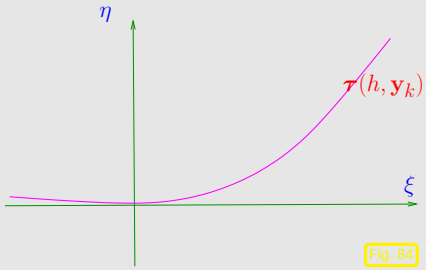


Fig. 84



Analytic considerations: recall Taylor's formula for function $\mathbf{y} \in C^{K+1}$

$$\mathbf{y}(t+h) - \mathbf{y}(t) = \sum_{j=0}^{K} \mathbf{y}^{(j)}(t)\frac{h^j}{j!} + \underbrace{\int_t^{t+h} f^{(K+1)}(\tau)\frac{(t+h-\tau)^K}{K!}\,\mathrm{d}\tau}_{=\dfrac{f^{(K+1)}(\xi)}{K!}h^{K+1}} \; , \tag{8.3.5}$$

for some $\xi \in [t, t+h]$

$\Rightarrow$ if $\mathbf{y} \in C^2([0,T])$, then

$$\mathbf{y}(t_{k+1}) - \mathbf{y}(t_k) = \dot{\mathbf{y}}(t_k)h_k + \tfrac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2 \quad \text{for some } t_k \le \xi_k \le t_{k+1}$$





$$= \mathbf{f}(\mathbf{y}(t_k))h_k + \tfrac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2 \; ,$$

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)

$$\begin{aligned}\boldsymbol{\tau}(h_k,\mathbf{y}(t_k)) &= \mathbf{\Psi}^{h_k}\mathbf{y}(t_k) - \mathbf{y}(t_k + h_k) \\ &\overset{(8.3.2)}{=} \mathbf{y}(t_k) + h_k\mathbf{f}(\mathbf{y}(t_k)) - \mathbf{y}(t_k) - \mathbf{f}(\mathbf{y}(t_k))h_k + \tfrac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2 \\ &= \tfrac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2 \; .\end{aligned} \tag{8.3.6}$$

Sloppily speaking, we observe $\boxed{\boldsymbol{\tau}(h_k,\mathbf{y}(t_k)) = O(h_k^2)}$ uniformly for $h_k \to 0$.



③ **Estimate for the propageted error** from (8.3.3)

$$\begin{aligned}\left\|\mathbf{\Psi}^{h_k}\mathbf{y}_k - \mathbf{\Psi}^{h_k}\mathbf{y}(t_k)\right\| &= \|\mathbf{y}_k + h_k\mathbf{f}(\mathbf{y}_k) - \mathbf{y}(t_k) - h_k\mathbf{f}(\mathbf{y}(t_k))\| \\ &\overset{(8.3.1)}{\le} (1 + Lh_k)\,\|\mathbf{y}_k - \mathbf{y}(t_k)\| \; .\end{aligned} \tag{8.3.7}$$

③ *Recursion* for error norms $\epsilon_k := \|\mathbf{e}_k\|$ by △-inequality:

$$\epsilon_{k+1} \le (1 + h_k L)\epsilon_k + \rho_k \; , \quad \rho_k := \tfrac{1}{2}h_k^2 \max_{t_k \le \tau \le t_{k+1}} \|\ddot{\mathbf{y}}(\tau)\| \; . \tag{8.3.8}$$

Taking into account $\epsilon_0 = 0$ this leads to

$$\epsilon_k \le \sum_{l=1}^{k} \prod_{j=1}^{l-1} (1 + Lh_j)\, \rho_l \;, \quad k = 1, \dots, N \;. \tag{8.3.9}$$

Use the elementary estimate $(1 + Lh_j) \le \exp(Lh_j)$ (by convexity of exponential function):

$$(8.3.9) \;\Rightarrow\; \epsilon_k \le \sum_{l=1}^{k} \prod_{j=1}^{l-1} \exp(Lh_j) \cdot \rho_l = \sum_{l=1}^{k} \exp(L \sum_{j=1}^{l-1} h_j) \rho_l \;.$$

Note: $\sum_{j=1}^{l-1} h_j \le T$ for final time $T$

$$\blacktriangleright \quad \epsilon_k \le \exp(LT) \sum_{l=1}^{k} \rho_l \le \exp(LT) \max_k \frac{\rho_k}{h_k} \sum_{l=1}^{k} h_l$$
$$\le T \exp(LT) \max_{l=1,\dots,k} h_l \cdot \max_{t_0 \le \tau \le t_k} \|\ddot{\mathbf{y}}(\tau)\| \;.$$

$$\blacktriangleright \quad \|\mathbf{y}_k - \mathbf{y}(t_k)\| \le T \exp(LT) \max_{l=1,\dots,k} h_l \cdot \max_{t_0 \le \tau \le t_k} \|\ddot{\mathbf{y}}(\tau)\| \;.$$

Total error arises from accumulation of one-step errors!

- error bound $= O(h)$, $h := \max_l h_l$ ($\blacktriangleright$ 1st-order algebraic convergence)
- Error bound grows *exponentially* with the length $T$ of the integration interval.

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 ($\rightarrow$ 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

$$\max_k \|\mathbf{y}_k - \mathbf{y}(t_k)\| \le Ch^p \quad \text{for} \quad h := \max_{k=1,\dots,N} |t_k - t_{k-1}| \to 0 \;,$$

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

angular velocity $p := \dot{\alpha} \;\Rightarrow\; \frac{d}{dt}\begin{pmatrix} \alpha \\ p \end{pmatrix} = \begin{pmatrix} p \\ -\frac{g}{l}\sin\alpha \end{pmatrix} \;, \quad g = 9.8, l = 1 \;.$ (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: $\alpha(0) = \pi/4$, $p(0) = 0$.

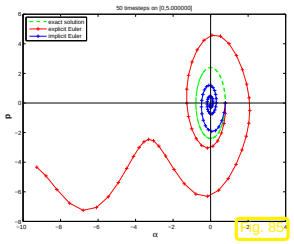


Fig. 85

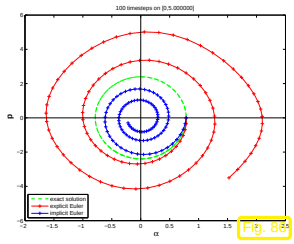


Fig. 86

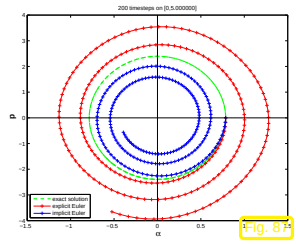


Fig. 87

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

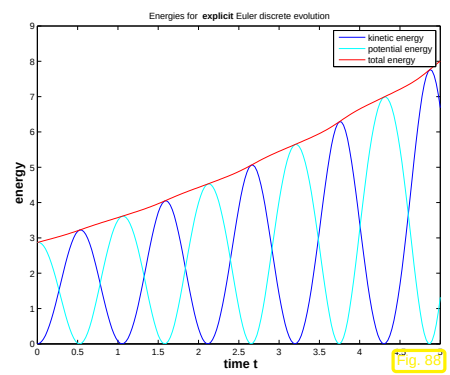


Fig. 88

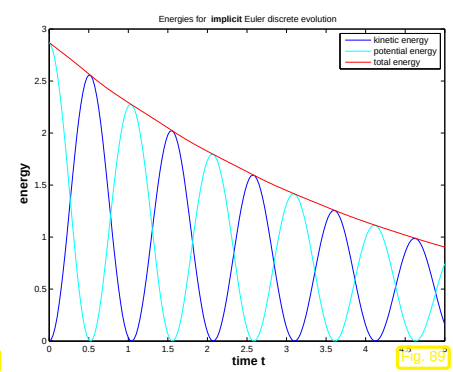


Fig. 89

☞ 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 , $D = \mathbb{R}^2$:

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

▶ Note that $I(\mathbf{y}) = \|\mathbf{y}\|$ is constant.
(movement with constant velocity on the circle)

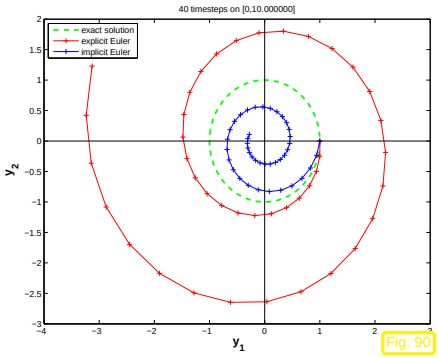


Fig. 90

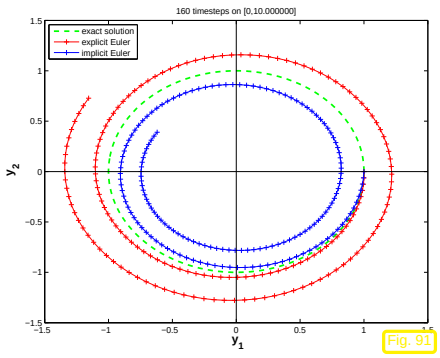


Fig. 91

☞ explicit Euler: numerical solution flyes away

☞ implicit Euler: numerical solution falls off into the center

◇



### 8.4.1 Implicit Midpoint Rule

Can we avoid the energy drift ?

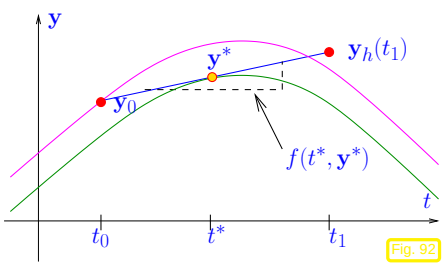


Fig. 92

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 $[t_0, t_1], [t_1, t_2], \ldots, [t_{N-1}, t_N]$ ➤ implicit midpoint rule

▶ via implicit midpoint rule generated approximation $\mathbf{y}_{k+1}$ for $\mathbf{y}(t_k)$ fulfils

$$\boxed{\mathbf{y}_{k+1} := \mathbf{y}_h(t_{k+1}) = \mathbf{y}_k + h_k \mathbf{f}(\tfrac{1}{2}(t_k + t_{k+1}), \tfrac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k+1})) \, , \quad k = 0, \ldots, N-1} \quad , \qquad (8.4.2)$$

with local (Time)step $h_k := t_{k+1} - t_k$ .





Note: (8.4.2) requires soltion 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 $\mathcal{G} := \{t_0, t_1, \ldots, t_N\}$:

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad \longleftrightarrow \quad \frac{\mathbf{y}_h(t_{k+1}) - \mathbf{y}_h(t_k)}{h_k} = \mathbf{f}(\tfrac{1}{2}(t_k + t_{k+1}), \tfrac{1}{2}(\mathbf{y}_h(t_k) + \mathbf{y}(t_{k+1})), \; k = 0, \ldots, N-1 \, .$$

△

*Example* 8.4.4 (Implicit midpoint rule for logistic equation).

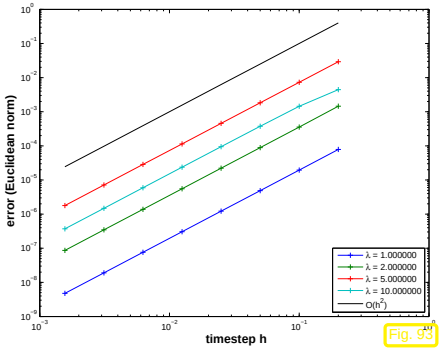


Fig. 93

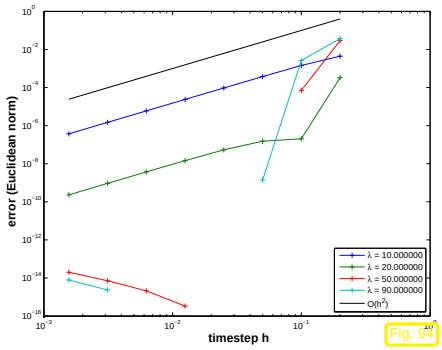


Fig. 94

$\lambda$ small: $O(h^2)$-convergence (asymptotical) $\qquad$ $\lambda$ large: stable for all time steps $h$ ! ◇

*Example* 8.4.5 (Implicit midpoint rule for circular motion).

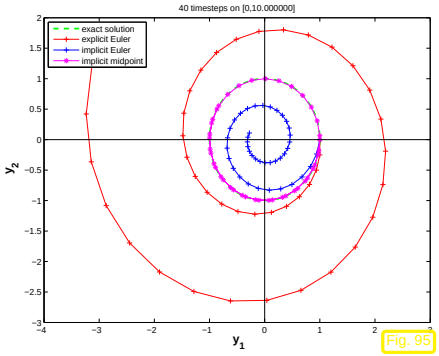


Fig. 95

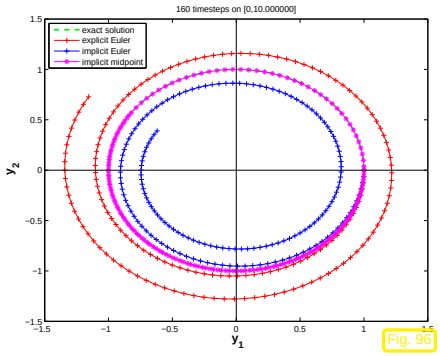


Fig. 96

☞ Implicit midpoint rule: perfect conservation of length !

◇

*Example* 8.4.6 (Implicit midpoint rule for pendulum).

Initial values and problem as in Bsp. 8.4.1

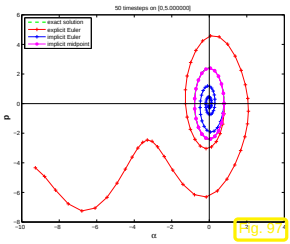


Fig. 97

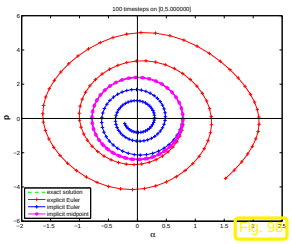


Fig. 98

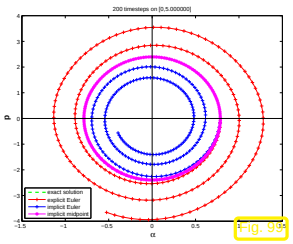


Fig. 99

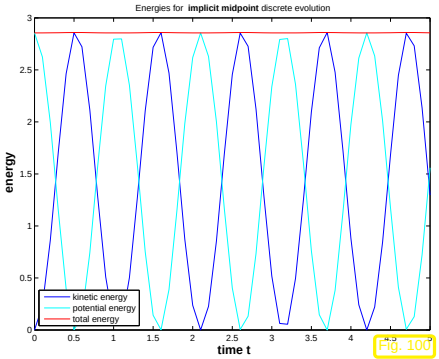


Fig. 100

◁ 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

?

$$\ddot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \ . \tag{8.4.3}$$

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 $\ddot{\mathbf{p}}(t_k) = \mathbf{f}(\mathbf{y}_k)$ $(*)$.

$(*)$ ➜ 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 $\mathcal{G} := \{t_0, t_1, \dots, t_N\}$):

$$\mathbf{y}_{k+1} = -\frac{h_k}{h_{k-1}}\mathbf{y}_{k-1} + \left(1 + \frac{h_k}{h_{k-1}}\right)\mathbf{y}_k + \tfrac{1}{2}(h_k^2 + h_k h_{k-1})\mathbf{f}(t_k, \mathbf{y}_k) \ , \quad k = 1, \dots, N-1 \ . \tag{8.4.4}$$

In case of a fixed time step $h$:

$$\mathbf{y}_{k+1} = -\mathbf{y}_{k-1} + 2\mathbf{y}_k + h^2\mathbf{f}(t_k, \mathbf{y}_k) \ , \quad k = 1, \dots, N-1 \ . \tag{8.4.5}$$

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 cosntant time-step $h > 0$

$$\ddot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad \longleftrightarrow \quad \frac{\frac{\mathbf{y}_h(t_{k+1}) - \mathbf{y}_h(t_k)}{h} - \frac{\mathbf{y}_h(t_k) - \mathbf{y}_h(t_{k-1})}{h}}{h} = \frac{\mathbf{y}_h(t_{k+1}) - 2\mathbf{y}_h(t_k) + \mathbf{y}_h(t_{k-1})}{h^2} = \mathbf{f}(\mathbf{y}_h(t_k)) \ .$$

△

*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 $[t_{-1}, t_1]$:

$$\mathbf{y}_1 = -\mathbf{y}_{-1} + 2\mathbf{y}_0 + h_0^2 \mathbf{f}(t_0, \mathbf{y}_0) \ . \tag{8.4.6}$$

- centered difference quotient on $[t_{-1}, t_1]$:

$$\frac{\mathbf{y}_1 - \mathbf{y}_{-1}}{2h_0} = \mathbf{v}_0 \ . \tag{8.4.7}$$

compute $\mathbf{y}_1$ from (8.4.6) **&** (8.4.7) △

*Example* 8.4.9 (Störmer-Verlet method for pendulum).



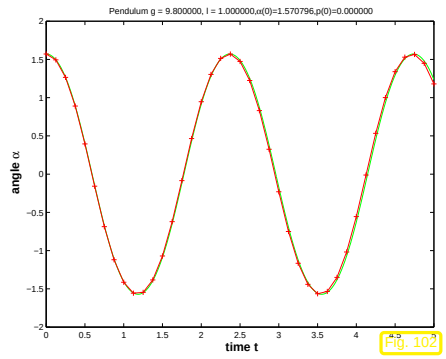


Fig. 102

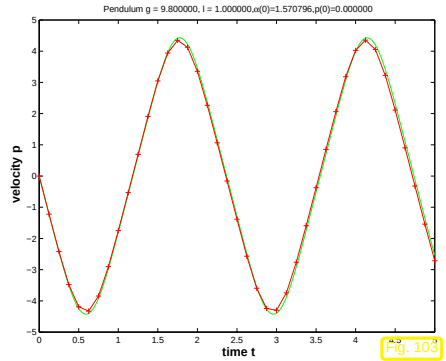


Fig. 103



- (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: $N = 40$

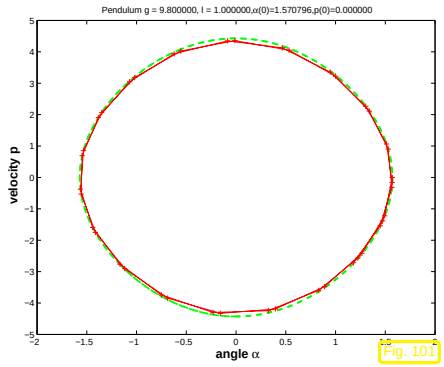


Fig. 101



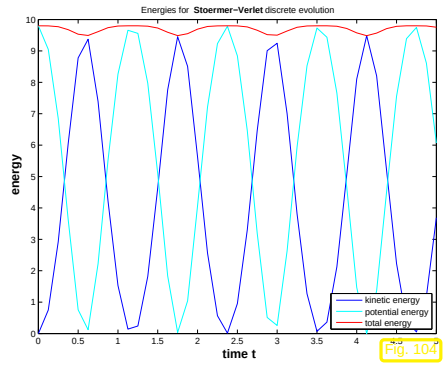


Fig. 104

☞ No energy drift thought 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 refromulation of a 2nd order equation to

a fisrt order equation, see (8.1.13): with $\mathbf{v}_{k+\frac{1}{2}} := \frac{\mathbf{y}_{k+1}-\mathbf{y}_k}{h}$

$$\ddot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad \longleftrightarrow \quad \begin{aligned} \dot{\mathbf{y}} &= \mathbf{v} \,, \\ \dot{\mathbf{v}} &= \mathbf{f}(\mathbf{y}) \,. \end{aligned}$$

$$\updownarrow \qquad\qquad\qquad \updownarrow$$

$$\mathbf{y}_{k+1} - 2\mathbf{y}_k + \mathbf{y}_{k-1} = h_k^2 \mathbf{f}(\mathbf{y}_k) \quad \longleftrightarrow \quad \boxed{\begin{aligned} \mathbf{v}_{k+\frac{1}{2}} &= \mathbf{v}_k + \tfrac{h}{2}\mathbf{f}(\mathbf{y}_k) \,, \\ \mathbf{y}_{k+1} &= \mathbf{y}_k + h\mathbf{v}_{k+\frac{1}{2}} \,, \\ \mathbf{v}_{k+1} &= \mathbf{v}_{k+\frac{1}{2}} + \tfrac{h}{2}\mathbf{f}(\mathbf{y}_{k+1}) \,. \end{aligned}}$$

two-steps method — one-step method

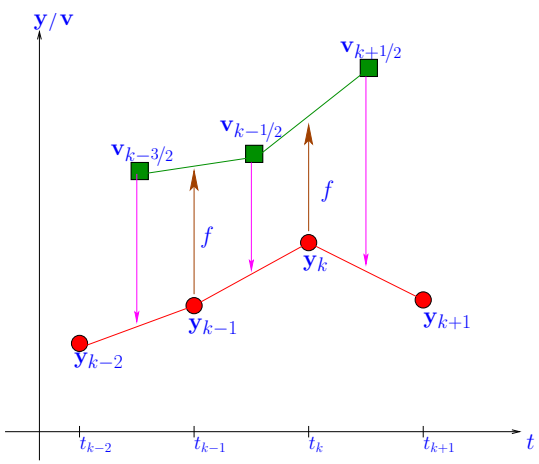
Initialisation (→ Rem. 8.4.8) is built in already in the formulation.

△

*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

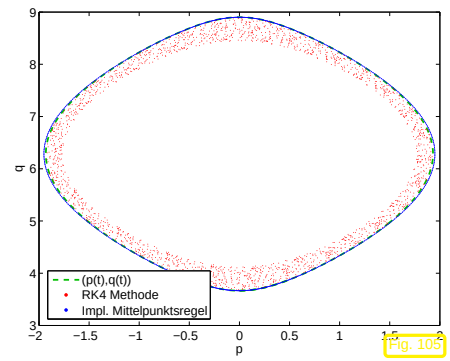


### 8.4.3 Examples

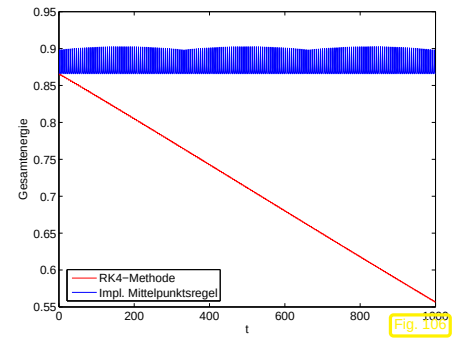
*Example* 8.4.12 (Energy conservation). ↔ Bsp. 8.4.6

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 enrgy 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 mehtod (**??**) → Ex. 8.4.9

*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)

$$\dot{\mathbf{p}} = -(\|\mathbf{q}\|-1)\frac{\mathbf{q}}{\|\mathbf{q}\|} - \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad , \quad \dot{\mathbf{q}} = \mathbf{p} \,. \tag{8.4.8}$$

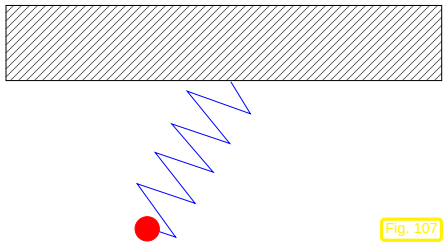


Fig. 107

Trajectories for long-time evolution ▷
(chaotical mechanical system)

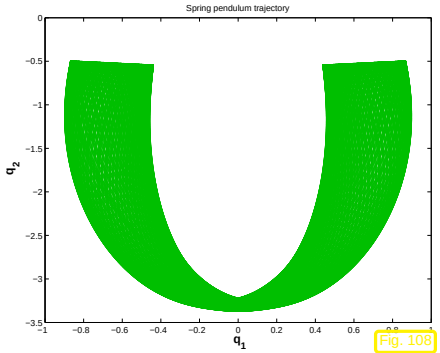



Fig. 108

ESV:
- Störmer-Verlet method (8.4.5) (order 2),
- explicit trapezoidal rule (8.6.3) (order 2).

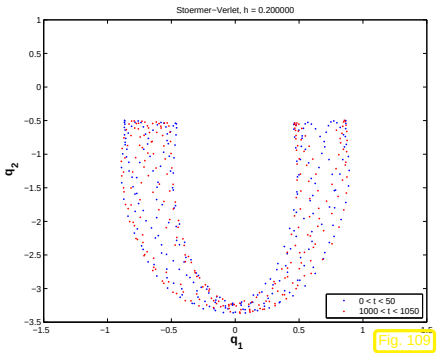



Fig. 109

Störmer-Verlet

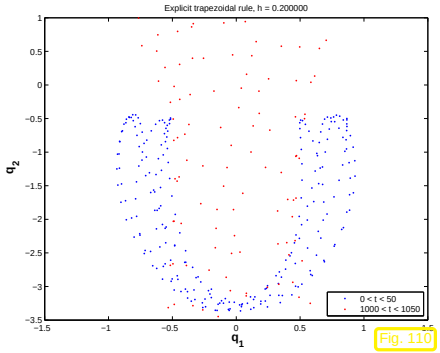



Fig. 110

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). → [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):

$$H(\mathbf{p}, \mathbf{q}) = \tfrac{1}{2}\|\mathbf{p}\|_2^2 + V(\mathbf{q}) \ .$$

Lenard-Jones-potential: ▷

$$V(\mathbf{q}) = \sum_{j=1}^{n} \sum_{i \neq j} \mathcal{V}(\left\|\mathbf{q}^i - \mathbf{q}^j\right\|_2) \ ,$$

$$\mathcal{V}(\xi) = \xi^{-12} - \xi^{-6} \ . \qquad (8.4.9)$$

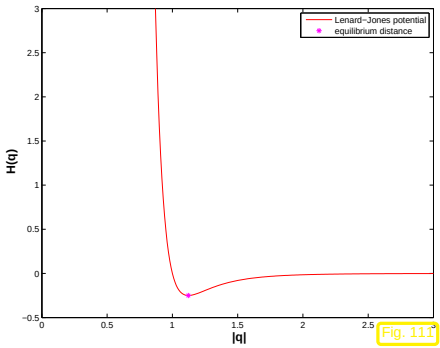



Fig. 111

➥ Hamiltonian ODE

$$\dot{\mathbf{p}}^j = -\sum_{i \neq j} \mathcal{V}'(\left\|\mathbf{q}^j - \mathbf{q}^i\right\|_2) \frac{\mathbf{q}^j - \mathbf{q}^i}{\left\|\mathbf{q}^j - \mathbf{q}^i\right\|_2} \quad , \quad \dot{\mathbf{q}}^j = \mathbf{p}^j \ , \ j = 1, \dots, n \ .$$

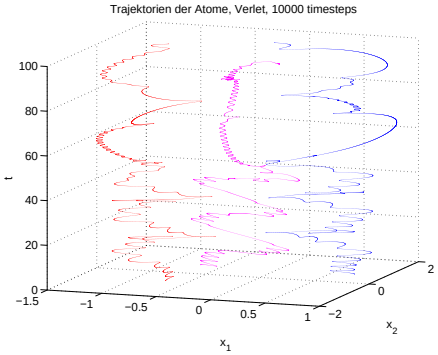
▶ Störmer-Verlet method (8.4.5):

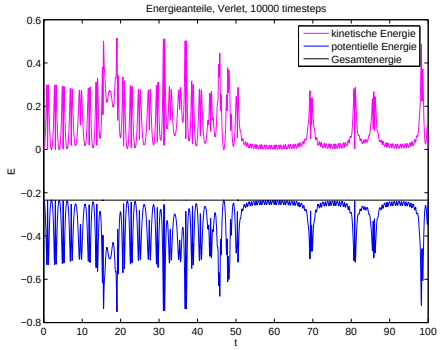
$$\mathbf{q}_h(t + \tfrac{1}{2}h) = \mathbf{q}_h(t) + \tfrac{h}{2}\mathbf{p}_h(t) \ ,$$

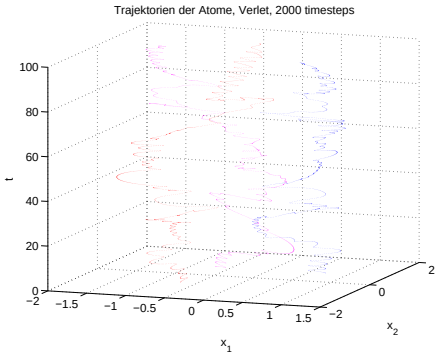
$$\mathbf{p}_h^j(t+h) = \mathbf{p}_h^j(t) - h \sum_{i \neq j} \mathcal{V}'(\left\|\mathbf{q}_h^j(t+\tfrac{1}{2}h) - \mathbf{q}_h^i(t+\tfrac{1}{2}h)\right\|_2) \frac{\mathbf{q}_h^j(t+\frac{1}{2}h) - \mathbf{q}_h^i(t+\frac{1}{2}h)}{\left\|\mathbf{q}_h^j(t+\frac{1}{2}h) - \mathbf{q}_h^i(t+\frac{1}{2}h)\right\|_2} \ ,$$

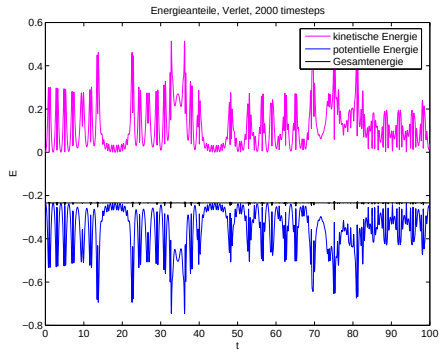
$$\mathbf{q}_h(t+h) = \mathbf{q}_h(t+\tfrac{1}{2}h) + \tfrac{h}{2}\mathbf{p}_h(t+h) \ .$$

Simulation with $d = 2$, $n = 3$, $\mathbf{q}^1(0) = \frac{1}{2}\sqrt{2}\binom{-1}{-1}$, $\mathbf{q}^2(0) = \frac{1}{2}\sqrt{2}\binom{1}{1}$, $\mathbf{q}^3(0) = \frac{1}{2}\sqrt{2}\binom{-1}{1}$, $\mathbf{p}(0) = 0$, end-time $T = 100$






Observations:

- very different trajektories for long-times with different tim=steps $h$.
- Qualitatively correct trajektories in any case.



2D conservative system of many particles with Lennard-Jones-potential $\longrightarrow$ Bsp. 8.4.14

initial positions ▷

(initial momenta $= 0 \leftrightarrow$ 0K)

Obseravtions for explicite trapezoiudal rule (8.6.3), Störmer-Verlet (8.4.5)

- Approximation of energy $H(\mathbf{p},\mathbf{q})$
- middle kinetic energy ("temperature")

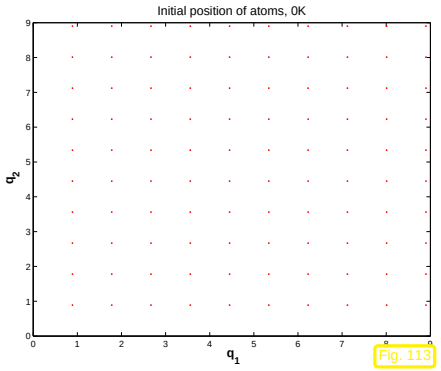



Fig. 113

Animation ▷



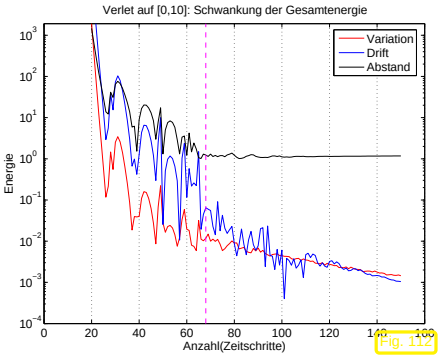



Fig. 112

$T = 10$, $d = 2$, $n = 3$, $\mathbf{q}^1(0) = \frac{1}{2}\sqrt{2}\binom{-1}{-1}$, $\mathbf{q}^2(0) = \frac{1}{2}\sqrt{2}\binom{1}{1}$, $\mathbf{q}^3(0) = \frac{1}{2}\sqrt{2}\binom{-1}{1}$, $\mathbf{p}(0) = 0$.

Variation $= \sum_{i=1}^{N-1} |E_{\text{tot}}((i+1)h) - E_{\text{tot}}(ih)|$ ,

Drift $= |E_{\text{tot}}(T) - E_{\text{tot}}(0)|$ ,

Abstand $= \max\{\left\|\mathbf{q}_h^j(T)\right\|_2, j = 1,2,3\}$ .

◇

*Example* 8.4.15 (Many particles molecular dynamics). $\longrightarrow$ [**?**, Sect. 4.5.1]



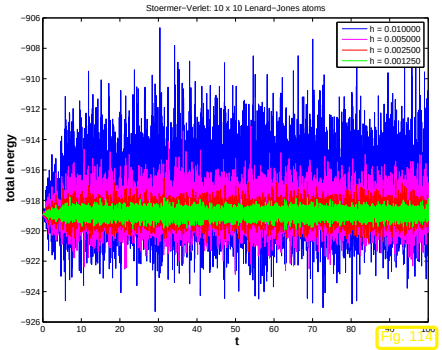



Fig. 114

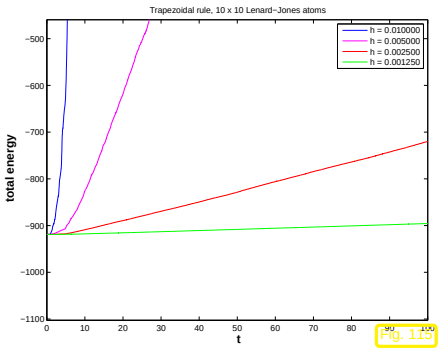



Fig. 115

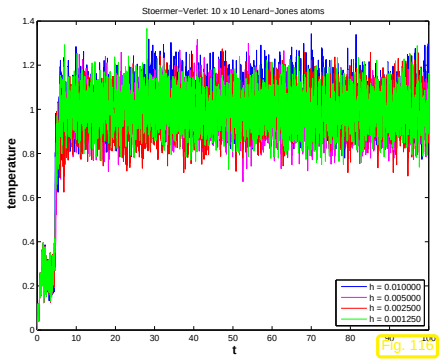


Fig. 116

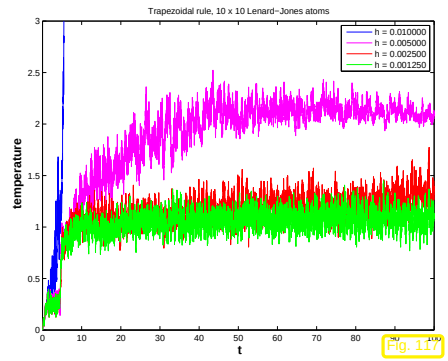


Fig. 117

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:

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) + \mathbf{g}(\mathbf{y}) \quad , \quad \mathbf{y}(0) = \mathbf{y}_0 \, , \tag{8.5.1}$$

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 (→ Def. 8.1.2)

(Continuous) Evolution:
$$\begin{aligned} \mathbf{\Phi}_f^t &\leftrightarrow \text{eq.} \quad \dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \, , \\ \mathbf{\Phi}_g^t &\leftrightarrow \text{eq.} \quad \dot{\mathbf{y}} = \mathbf{g}(\mathbf{y}) \, . \end{aligned}$$

Assume: $\mathbf{\Phi}_f^t$, $\mathbf{\Phi}_g^t$ (analytically) known

Idea: Construct Single-step method with discrete evolution

Lie-Trotter-Splitting: $\mathbf{\Psi}^h = \mathbf{\Phi}_g^h \circ \mathbf{\Phi}_f^h \, ,$ (8.5.2)

Strang-Splitting: $\mathbf{\Psi}^h = \mathbf{\Phi}_f^{h/2} \circ \mathbf{\Phi}_g^h \circ \mathbf{\Phi}_f^{h/2} \, .$ (8.5.3)





(8.5.2) ↔

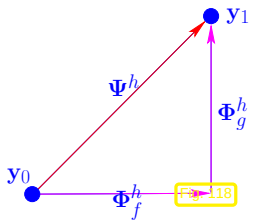


Fig. 118

(8.5.3) ↔

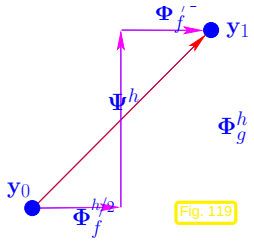


Fig. 119

*Example* 8.5.1 (Convergence of simple splitting methods).

$$\dot{y} = \underbrace{\lambda y(1-y)}_{=:f(y)} + \underbrace{\sqrt{1-y^2}}_{=:g(y)} \quad , \quad y(0) = 0 \, .$$

- $\mathbf{\Phi}_f^t y = \dfrac{1}{1 + (y^{-1} - 1)e^{-\lambda t}} \, , \; t > 0, y \in ]0,1]$ (logistic differential equation (8.1.3))
- $\mathbf{\Phi}_g^t y = \begin{cases} \sin(t + \arcsin(y)) & \text{, if } t + \arcsin(y) < \frac{\pi}{2} \, , \\ 1 & \text{, else,} \end{cases} \quad t > 0, y \in [0,1] \, .$





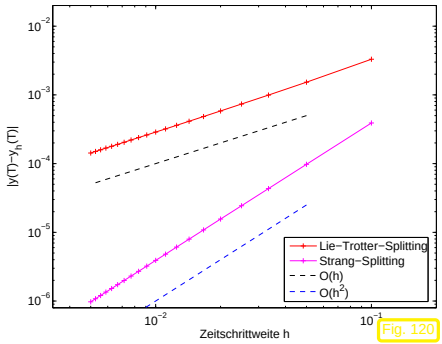


Fig. 120

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 $\ddot{\mathbf{r}} = a(\mathbf{r}) \overset{(8.1.13)}{\Longleftrightarrow} \dot{\mathbf{y}} := \begin{pmatrix} \dot{\mathbf{r}} \\ \mathbf{v} \end{pmatrix} = \begin{pmatrix} \mathbf{v} \\ a(\mathbf{r}) \end{pmatrix} =: \mathbf{F}(\mathbf{y}) \, .$

Splitting: $F(\mathbf{y}) = \underbrace{\begin{pmatrix} 0 \\ a(\mathbf{r}) \end{pmatrix}}_{=:\mathbf{f}(\mathbf{y})} + \underbrace{\begin{pmatrix} \mathbf{v} \\ 0 \end{pmatrix}}_{=:\mathbf{g}(\mathbf{y})} \, .$

- $\mathbf{\Phi}_f^t \begin{pmatrix} \mathbf{r}_0 \\ \mathbf{v}_0 \end{pmatrix} = \begin{pmatrix} \mathbf{r}_0 \\ \mathbf{v}_0 + ta(\mathbf{r}_0) \end{pmatrix} \quad , \quad \mathbf{\Phi}_g^t \begin{pmatrix} \mathbf{r}_0 \\ \mathbf{v}_0 \end{pmatrix} = \begin{pmatrix} \mathbf{r}_0 + t\mathbf{v}_0 \\ \mathbf{v}_0 \end{pmatrix} \, .$

Lie-Trotter-Splitting (8.5.2): ➢ Symplectic Euler method

$$\Psi^h\begin{pmatrix}\mathbf{r}\\ \mathbf{v}\end{pmatrix} = \left(\Phi_g^h \circ \Phi_f^h\right)\begin{pmatrix}\mathbf{r}\\ \mathbf{v}\end{pmatrix} = \begin{pmatrix}\mathbf{r}+h(\mathbf{v}+ha(\mathbf{r}))\\ \mathbf{v}+ha(\mathbf{r})\end{pmatrix} . \tag{8.5.4}$$

Strang-Splitting (8.5.3):

$$\Psi^h\begin{pmatrix}\mathbf{r}\\ \mathbf{v}\end{pmatrix} = \left(\Phi_g^{h/2} \circ \Phi_f^h \circ \Phi_g^{h/2}\right)\begin{pmatrix}\mathbf{r}\\ \mathbf{v}\end{pmatrix} = \begin{pmatrix}\mathbf{r}+h\mathbf{v}+\frac{1}{2}h^2a(\mathbf{r}+\frac{1}{2}h\mathbf{v})\\ \mathbf{v}+ha(\mathbf{r}+\frac{1}{2}h\mathbf{v})\end{pmatrix} . \tag{8.5.5}$$

= single-step formulation of Störmer-Verlet methods (8.4.5), see Rem. 8.4.10 **!**

$$(8.5.5) \quad \longleftrightarrow \quad \begin{aligned} \mathbf{r}_{k+\frac{1}{2}} &= \mathbf{r}_k + \tfrac{1}{2}h\mathbf{v}_k ,\\ \mathbf{v}_{k+1} &= \mathbf{v}_k + ha(\mathbf{r}_{k+\frac{1}{2}}) ,\\ \mathbf{r}_{k+1} &= \mathbf{r}_{k+\frac{1}{2}} + \tfrac{1}{2}h\mathbf{v}_{k+1} . \end{aligned} \tag{8.5.6}$$

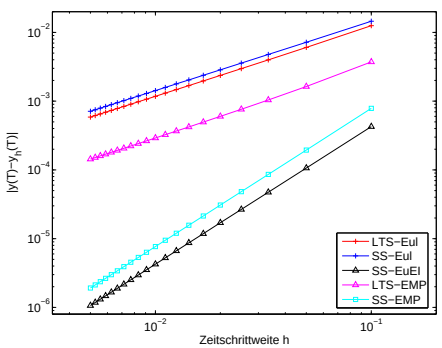
◇

Idea: Replace

| exact evolution | $\longrightarrow$ | discrete evolution |
|---|---|---|
| $\Phi_g^h$, $\Phi_f^h$ | $\longrightarrow$ | $\Psi_g^h$, $\Psi_f^h$ |

*Example* 8.5.3 (Inexact splitting method). Cont. Ex. 8.5.1

IVP of Eq. 8.5.1, inexact splitting method when we rely on (several) inexact underlying methods:




| | |
|---|---|
| LTS-Eul | explicit Euler method (**??**) $\longrightarrow$ $\Psi_{h,g}^h$, $\Psi_{h,f}^h$ + Lie-Trotter-Splitting (8.5.2) |
| SS-Eul | explicit Euler method (**??**) $\longrightarrow$ $\Psi_{h,g}^h$, $\Psi_{h,f}^h$ + Strang-Splitting (8.5.3) |
| SS-EuEl | Strang-Splitting (8.5.3): explicit Euler method (**??**) $\circ$ exact evolution $\Phi_g^h$ $\circ$ implicite Euler method (**??**) |
| LTS-EMP | explicit midpoint rule (8.6.4) $\longrightarrow$ $\Psi_{h,g}^h$, $\Psi_{h,f}^h$ + Lie-Trotter-Splitting (8.5.2) |
| SS-EMP | explicit midpoint rule (8.6.4) $\longrightarrow$ $\Psi_{h,g}^h$, $\Psi_{h,f}^h$ + Strang-Splitting (8.5.3) |

☞ Order of splitting methods is determined by the quality of $\Phi_f^h$, $\Phi_g^h$.

◇

*Remark* 8.5.4. Note that the converegence 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 buld a class of methods that achieve orders $> 1$. The starting point is a simple *integral equation* satisfied by solutions of initial value problems:

IVP: $\begin{aligned}\dot{\mathbf{y}}(t) &= \mathbf{f}(t,\mathbf{y}(t)) ,\\ \mathbf{y}(t_0) &= \mathbf{y}_0\end{aligned} \quad \Rightarrow \quad \mathbf{y}(t_1) = \mathbf{y}_0 + \int_{t_0}^{t_1} \mathbf{f}(\tau,\mathbf{y}(t_0+\tau))\,\mathrm{d}\tau$

Idea: approximate integral by means of $s$-point quadrature formula (→ Sect. 7.1, defined on reference interval $[0,1]$) with nodes $c_1,\dots,c_s$, weights $b_1,\dots,b_s$.

$$\mathbf{y}(t_1) \approx \mathbf{y}_1 = \mathbf{y}_0 + h\sum_{i=1}^{s} b_i\mathbf{f}(t_0+c_ih, \boxed{\mathbf{y}(t_0+c_ih)}) , \quad h := t_1 - t_0 . \tag{8.6.1}$$

Obtain these values by bootstrapping

bootstrapping = use the same idea in a simpler version to get $\mathbf{y}(t_0+c_ih)$, 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 $\mathbf{y}(t_0+c_ih)$ (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

$\mathbf{y}(t_0+c_ih)$ is approximated up to an error $O(h^p)$,

because in (8.6.1) a factor of size $h$ multiplies $\mathbf{f}(t_0+c_i,\mathbf{y}(t_0+c_ih))$.

This is accmomplished 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):

$$Q(f) = \tfrac{1}{2}(f(0)+f(1)) \quad \leftrightarrow \quad s=2: \quad c_1 = 0, c_2 = 1\,, \quad b_1 = b_2 = \frac{1}{2}\,, \tag{8.6.2}$$

and $\mathbf{y}(T)$ approximated by explicit Euler step (8.2.1)

$$\mathbf{k}_1 = \mathbf{f}(t_0,\mathbf{y}_0)\,, \quad \mathbf{k}_2 = \mathbf{f}(t_0+h,\mathbf{y}_0+h\mathbf{k}_1)\,, \quad \mathbf{y}_1 = \mathbf{y}_0 + \tfrac{h}{2}(\mathbf{k}_1+\mathbf{k}_2)\,. \tag{8.6.3}$$

(8.6.3) = explicit trapezoidal rule (for numerical integration of ODEs)

Quadrature formula $\longrightarrow$ simplest Gauss quadrature formula = midpoint rule & $\mathbf{y}(\frac{1}{2}(t_1+t_0))$ approximated by explicit Euler step (8.2.1)

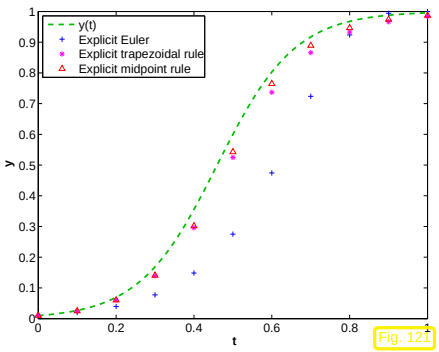
$$\mathbf{k}_1 = \mathbf{f}(t_0,\mathbf{y}_0)\,, \quad \mathbf{k}_2 = \mathbf{f}(t_0+\tfrac{h}{2},\mathbf{y}_0+\tfrac{h}{2}\mathbf{k}_1)\,, \quad \mathbf{y}_1 = \mathbf{y}_0 + h\mathbf{k}_2\,. \tag{8.6.4}$$

(8.6.4) = explicit midpoint rule (for numerical integration of ODEs)

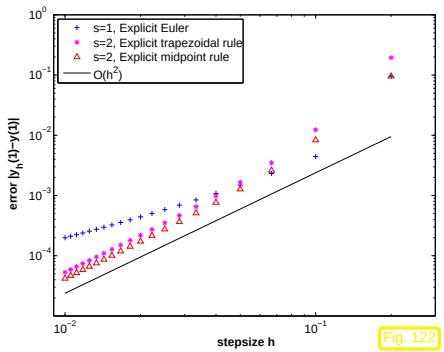
◇

*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 $h$.




$y_h(j/10)$, $j = 1,\dots,10$ for explicit RK-methods




Errors at final time $y_h(1) - y(1)$

Observation: obvious algebraic convergence 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*

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + h\sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j)\,, \quad i = 1,\dots,s \quad , \quad \mathbf{y}_1 := \mathbf{y}_0 + h\sum_{i=1}^{s} b_i\mathbf{k}_i\,.$$

*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 $s$ $\mathbf{f}$-evaluations and AXPY operations are required.

Shorthand notation for (explicit) Runge-Kutta methods

Butcher scheme ▷

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & 0 & \cdots & 0 \\ c_2 & a_{21} & \ddots & \vdots \\ \vdots & \vdots & & \ddots \;\; \vdots \\ c_s & a_{s1} & \cdots & a_{s,s-1} \;\; 0 \\ \hline & b_1 & \cdots & b_s \end{array}\,. \tag{8.6.5}$$

(Note: $\mathfrak{A}$ is strictly lower triangular $s \times s$-matrix)

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 $\displaystyle\sum_{i=1}^{s} b_i = 1$

*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: $\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array}$ ➢ order = 4

- Kutta's 3/8-rule: $\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{2}{3} & -\frac{1}{3} & 1 & 0 & 0 \\ 1 & 1 & -1 & 1 & 0 \\ \hline & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} \end{array}$ ➢ order = 4

◇

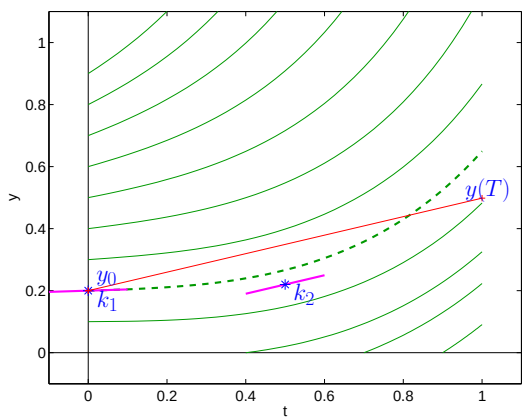
*Example* 8.6.4 (explicit Runge-Kutta method for Ricatti equation).
Initial value problem: $\dot{y} = t^2 + y^2$, $y(0) = 0.2$.

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 slops $\mathbf{k}_i$
$*$ points $f$-evaluation
red: polygonal line



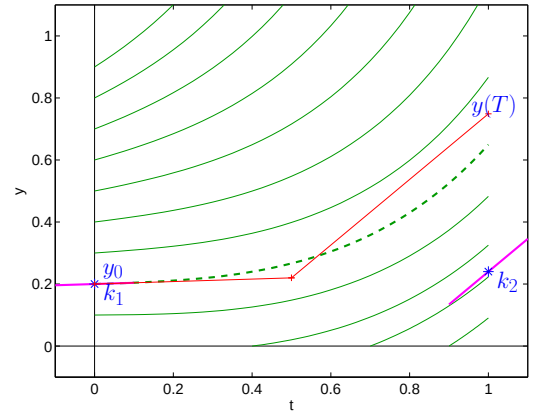





explicit trapezoidal rule

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

green: solution
**magenta**: local slops $\mathbf{k}_i$
$*$: points $f$-evaluation
red: polygonal line



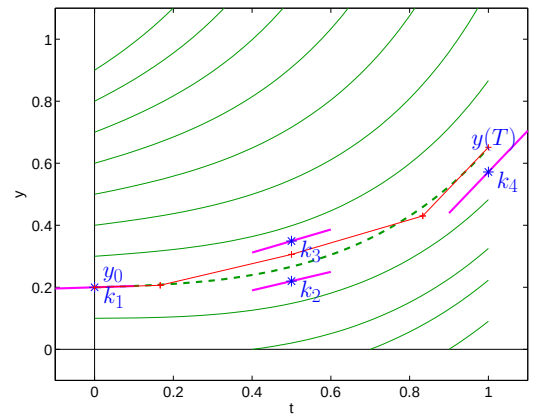





classical Runge-Kutta method (RK4)

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array} \qquad (8.6.6)$$

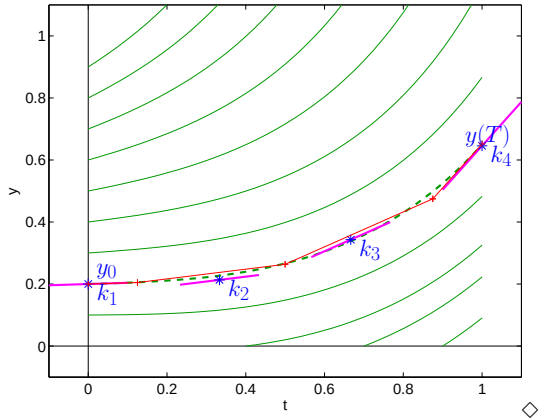
green solution
**magenta** local slopes $\mathbf{k}_i$
$*$ point $f$-evaluation
red: polygonal line



Kuttas 3/8-rule

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{2}{3} & -\frac{1}{3} & 1 & 0 & 0 \\ 1 & 1 & -1 & 1 & 0 \\ \hline & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} \end{array} \qquad (8.6.7)$$

| | |
|---|---|
| green | solution |
| **magenta** | local slopes $\mathbf{k}_i$ |
| $*$ | 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 | $\geq 9$ |
|---|---|---|---|---|---|---|---|---|---|
| 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)` $\leftrightarrow$ 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 $\{t_0 < t_1 < t_2 < \cdots < t_{N-1} = t_N = T\}$ |
| `y` | : | sequence $(\mathbf{y}_k)_{k=0}^N$ (column vectors) |

△



*Example* 8.6.7 (Numerical integration of logistic ODE ).

integrator: `ode45()`:

usage of ode45

```
from ode45 import ode45
fn = lambda t, y:  5.*y*(1.-y)
t,y = ode45(fn,(0, 1.5),[1.5])
```

- Handle passing r.h.s.
- initial and final time
- initial state $\mathbf{y}_0$

◇

## 8.7 Stepsize control

*Example* 8.7.1 (Oregonator reaction).

Special case of oscillating Zhabotinski-Belousov reaction [21]:

$$\begin{array}{rcl} BrO_3^- + Br^- & \mapsto & HBrO_2 \\ HBrO_2 + Br^- & \mapsto & Org \\ BrO_3^- + HBrO_2 & \mapsto & 2\,HBrO_2 + Ce(IV) \\ 2\,HBrO_2 & \mapsto & Org \\ Ce(IV) & \mapsto & Br^- \end{array} \qquad (8.7.1)$$

$$\begin{array}{lrcl} y_1 := c(BrO_3^-): & \dot{y}_1 & = & -k_1y_1y_2 - k_3y_1y_3\,, \\ y_2 := c(Br^-): & \dot{y}_2 & = & -k_1y_1y_2 - k_2y_2y_3 + k_5y_5\,, \\ y_3 := c(HBrO_2): & \dot{y}_3 & = & k_1y_1y_2 - k_2y_2y_3 + k_3y_1y_3 - 2k_4y_3^2\,, \\ y_4 := c(Org): & \dot{y}_4 & = & k_2y_2y_3 + k_4y_3^2\,, \\ y_5 := c(Ce(IV)): & \dot{y}_5 & = & k_3y_1y_3 - k_5y_5\,, \end{array} \qquad (8.7.2)$$

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 inital state $y_1(0) = 0.06$, $y_2(0) = 0.33\cdot 10^{-6}$, $y_3(0) = 0.501\cdot 10^{-10}$, $y_4(0) = 0.03$, $y_5(0) = 0.24\cdot 10^{-7}$:

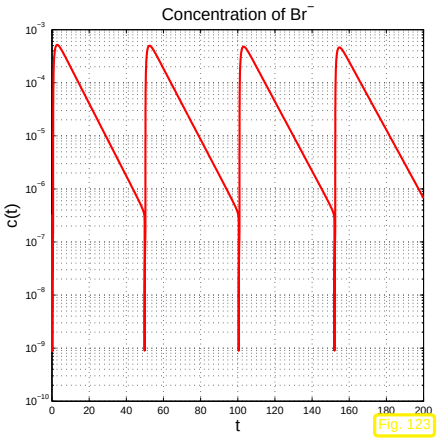


Fig. 123

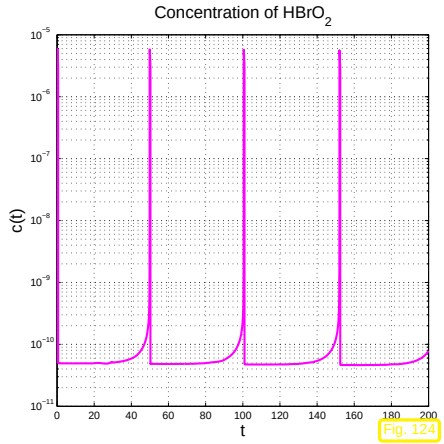


Fig. 124

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:

$$\dot{y} = y^2 \;, \quad y(0) = y_0 > 0 \;.$$

$$\blacktriangleright \quad y(t) = \frac{y_0}{1 - y_0 t} \;, \quad t < 1/y_0 \;.$$

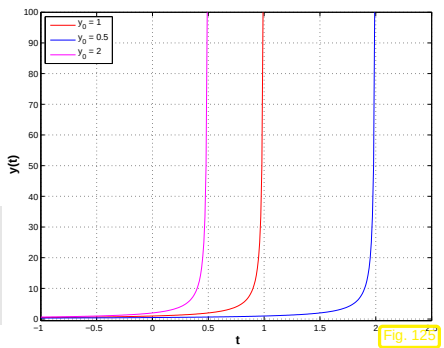


Fig. 125

Solution exists only for finite time and then suffers a Blow-up, that is, $\lim_{t \to 1/y_0} y(t) = \infty$ : $J(y_0) = ]-\infty, 1/y_0]$!

How to choose temporal mesh $\{t_0 < t_1 < \cdots < t_{N-1} < t_N\}$ 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?



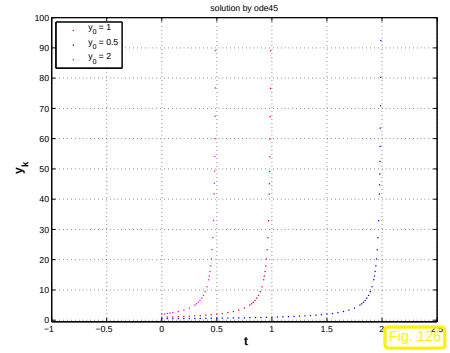


Fig. 126

```
odefun = lambda t,y: y**2.0
print 'y0 = ',0.5
t2, y2 = ode45(odefun, (0,2), 0.5, stats=True)
print '\n','y0 = ',1
t1, y1 = ode45(odefun, (0,2), 1, stats=True)
print '\n','y0 = ',2
t3, y3 = ode45(odefun, (0,2), 2, 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* $\{0 = t_0 < t_1 < \cdots < t_{N-1} < t_N\}$
for a given single step method applied to an IVP

What does "good" mean ?

be efficient — be accurate

*Objective:* $N$ as small as possible & $\max_{k=1,\dots,N} \|\mathbf{y}(t_k) - \mathbf{y}_k\| < \mathtt{TOL}$ or $\|\mathbf{y}(T) - \mathbf{y}_N\| < \mathtt{TOL}$ , $\mathtt{TOL}$ = tolerance

*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,

☞ because it is inexpensive and it works for many practical problems,

☞ because there is not reliable method that can deliver guaranteed accuracy for general IVP.

"Recycle" heuristics already employed for adaptive quadrature, see Sect. 7.3:

Idea: *Estimation of one-step error*, *cf.* Sect. 7.3

Compare two discrete evolutions $\mathbf{\Psi}^h$, $\widetilde{\mathbf{\Psi}}^h$ of different order for *current timestep* $h$:

If Order$(\widetilde{\mathbf{\Psi}}) >$ Order$(\mathbf{\Psi})$

$$\Rightarrow \quad \underbrace{\mathbf{\Phi}^h\mathbf{y}(t_k) - \mathbf{\Psi}^h\mathbf{y}(t_k)}_{\text{one-step error}} \approx \mathtt{EST}_k := \widetilde{\mathbf{\Psi}}^h\mathbf{y}(t_k) - \mathbf{\Psi}^h\mathbf{y}(t_k) \,. \tag{8.7.3}$$

**Heuristics** for concrete $h$

absolute tolerance

▶ Compare $\mathtt{EST}_k \leftrightarrow \mathtt{ATOL}$, $\mathtt{EST}_k \leftrightarrow \mathtt{RTOL}\,\|\mathbf{y}_k\|$ ➢ Reject/accept current step (8.7.4)

relative tolerance



▶ Simple algorithm:

$\mathtt{EST}_k < \max\{\mathtt{ATOL}, \|\mathbf{y}_k\|\,\mathtt{RTOL}\}$: Carry out next timestep (stepsize $h$)
Use larger stepsize (e.g., $\alpha h$ with some $\alpha > 1$) for following step $(*)$

$\mathtt{EST}_k > \max\{\mathtt{ATOL}, \|\mathbf{y}_k\|\,\mathtt{RTOL}\}$: *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

```
def odeintadapt(Psilow, Psihigh, T, y0, h0, reltol, abstol, hmin):
        t=[0]; y=[y0]; h=h0 #
        while t[-1] < T and h > hmin:
                yh = Psihigh(h,y0)
                yH = Psilow(h,y0)
                est = norm(yH-yh)
                if est < max(reltol*norm(y0), abstol):
                        y0 = yh; y.append(y0);
                          t.append(t[-1]+min(T-t[-1],h))
                        h = 1.1*h
                else:
                        h = h/2.0
        return (array(t),array(y))

def odeintadapt_ext(Psilow, Psihigh, T, y0, h0, reltol, abstol, hmin):
        """extended version of odeintadapt. also returns vector of
          rejected points and vector of estimated errors"""
        t=[0]; y=[y0]; h=h0; rej = []; ee = [0]
        while t[-1] < T and h > hmin:
                yh = Psihigh(h,y0)
                yH = Psilow(h,y0)
                est = norm(yH-yh)
                if est < max(reltol*norm(y0), abstol):
# print 'accept step, h = y0 = yh; y.append(y0); t.append(t[-1]+min(T-t[-1],h))
                        h = 1.1*h; ee.append(est)
                else:
# print 'reject step, h = rej = hstack([rej,t]); h = h/2.0
        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 $\mathbf{y}_0$,
- `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:





- `t`: temporal mesh $t_0 < t_1 < t_2 < \ldots < t_N < T$, where $t_N < T$ indicated premature termination (collapse, blow-up),
- `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 $\boldsymbol{\Psi}$ and that we should use $\mathbf{y}_{k+1} = \boldsymbol{\Psi}^{h_k}\boldsymbol{y}_k$, if the timestep is accepted.

However, it would be foolish not to use the better value $\mathbf{y}_{k+1} = \widetilde{\boldsymbol{\Psi}}^{h_k}\boldsymbol{y}_k$, 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

```
def odeintadaptdriver(T,a,reltol=1e-2,abstol=1e-4):
        """Simple adaptive timestepping strategy of
          Code~\ref{mc:odeintadapt}
        based on explicit Euler \eqref{eq:eeul} and explicit trapezoidal
        rule \eqref{eq:exTrap}
        """

        # autonomous ODE ẏ = cos(ay) and its general solution
        f = lambda y: (cos(a*y)**2)
        sol = lambda t: arctan(a*(t-1))/a
        # Initial state y_0
        y0 = sol(0)

        # Discrete evolution operators, see Def. 8.2.1
        # Explicit Euler (8.2.1)
        Psilow = lambda h,y: y + h*f(y)
        # Explicit trapzoidal rule (8.6.3)
        Psihigh = lambda h,y: y + 0.5*h*( f(y) + f(y+h*f(y)) )

        # Heuristic choice of initial timestep and h_min
        h0 = T/(100.0*(norm(f(y0))+0.1)); hmin = h0/10000.0;
        # Main adaptive timestepping loop, see Code 9.2.2
        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')
```

```
        plt.xlabel(r't',fontsize=14)
        plt.ylabel(r'error',fontsize=14)
        plt.legend(loc='upper left')
        plt.title('Adaptive timestepping, rtol = %1.2f, atol = %1.4f, a = %d' % (reltol,abstol,a))
        plt.show()
# plt.savefig('../PICTURES/odeintadapterr.eps')

if __name__ == '__main__':
        odeintadaptdriver(T=2,a=20,reltol=1e-2,abstol=1e-4)
```

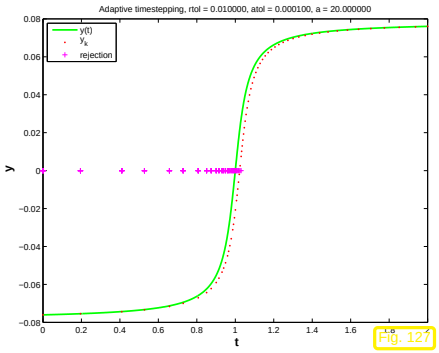


Fig. 127

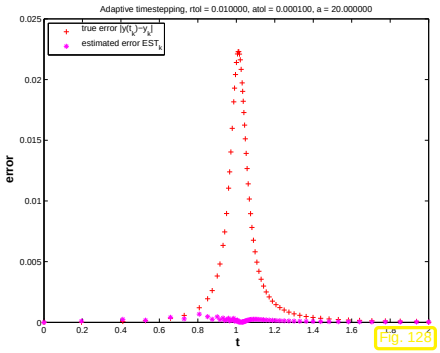


Fig. 128

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

```
from numpy import *
import matplotlib.pyplot as plt
from odeintadapt import odeintadapt_ext

def adaptgain(T=2.0, a=40.0, reltol=1e-1, abstol=1e-3):
        """Experimental study of gasin through simple adaptive timestepping
        strategy of Code~\ref{mc:odeintadapt} based on explicit Euler
        \eqref{eq:eeul} and explicit trapezoidal
        rule \eqref{eq:exTrap}
        """

        # autonomous ODE ẏ = cos(ay) and its general solution
        f = lambda y: (cos(a*y)**2.0)
        sol = lambda t: arctan(a*(t))/a # the constant c is 0: arctan(a*(t-c))/a
        # Initial state y0
        y0 = sol(0)

        # Discrete evolution operators, see Def. 8.2.1
        # Explicit Euler (8.2.1)
        Psilow = lambda h,y: y + h*f(y)
        # Explicit trapzoidal rule (8.6.3)
        Psihigh = lambda h,y: y + 0.5*h*(f(y)+f(y+h*f(y)))

        # Loop over uniform timesteps of varying length and integrate ODE by explicit trapezoidal
        # rule (8.6.3)
        nvals = r_[10:210:10]
        err_unif = []
        for N in nvals:
                h = T/N; t = 0; y = y0; err = 0
                for k in arange(1,N+1):
                        y = Psihigh(h,y); t = t+h
                        err = max(err,abs(sol(t) - y));
                err_unif.append(err)
        # Run adaptive timestepping with various tolerances, which is the only way
```

```
        # to make it use a different total number of timesteps.
        # Plot the solution sequences for different values of the relative tolerance.
        plt.figure()
        # axis([0 2 0 0.05]); hold on; col = colormap;
        ntimes = []
        err_ad = []
        tols = []
        rejs = []
        l = 1
        for rtol in reltol*2.0**r_[2:-5:-1]:
                # Crude choice of initial timestep and h_min
                h0 = T/10.0; hmin = h0/10000.0
                # Main adaptive timestepping loop, see Code 9.2.2
                t,y,rej,ee = odeintadapt_ext(Psilow,Psihigh,T,y0,h0,rtol,0.01*rtol,h
                ntimes.append(len(t)-1)
                err_ad.append(max(abs(sol(t) - y)))
                tols.append(rtol)
                rejs.append(len(rej))
                print 'rtol = %1.3f: %d timesteps, %d rejected timesteps' % (rtol,len(t)-1,len(rej))
                plt.plot(t,y,'.',label='rtol = %f'%rtol)#,'color',col(10*(l-1)+1,:));
                l = l+1
```



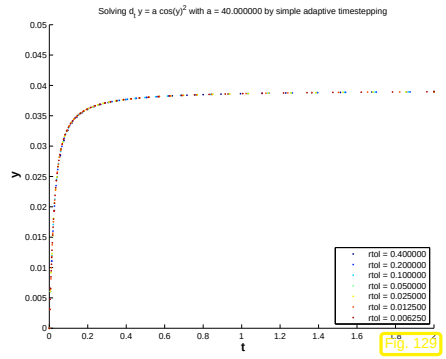



Fig. 129

Solutions $(\mathbf{y}_k)_k$ for different values of `rtol`

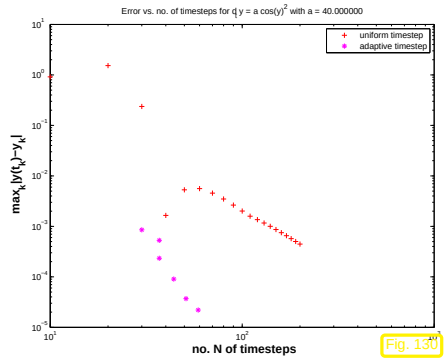



Fig. 130

Error vs. no. of timesteps

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

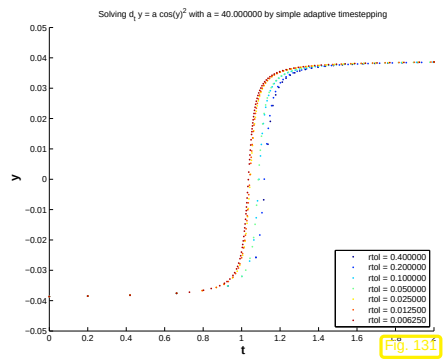



Fig. 131

Solutions $(\mathbf{y}_k)_k$ for different values of `rtol`

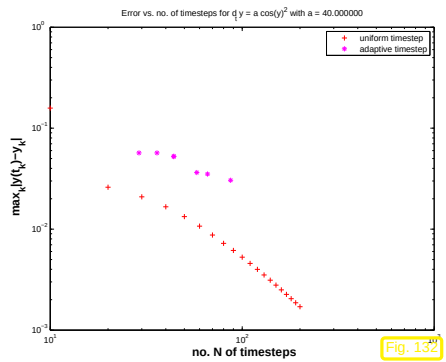



Fig. 132

Error vs. no. of timesteps

```
        plt.xlabel(r't', fontsize=14)
        plt.ylabel(r'y', fontsize=14)
        plt.legend(loc='lower right')
        plt.title(r'Solving $d_t y = a \cdot \cos(y)^2$ with $a =' +str(a)+'$ by simple adaptive timestepping')
        # plt.savefig('../PICTURES/adaptgainsol.eps')

        # Plotting the errors vs. the number of timesteps
        plt.figure()
        plt.loglog(nvals, err_unif,'r+',label='uniform timestep')
        plt.loglog(ntimes, err_ad,'m*',label='adaptive timestep')
        plt.xlabel('no. N of timesteps',fontsize=14)
        plt.ylabel(r'$\max_k |y(t_k) - y_k|$',fontsize=14)
        plt.title(r'Error vs. no. of timesteps for $d_t y = a \cdot \cos(y)^2$ with $a =' +str(a)+'$')
        plt.legend(loc='upper right')
        plt.show()
        # plt.savefig('../PICTURES/adaptgain.eps')

if __name__ == '__main__':
        adaptgain()
```

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" informatiom contained in $\mathrm{EST}_k$ : TOL:

More ambitious goal **!** When $\mathrm{EST}_k > \mathrm{TOL}$ : stepsize adjustment better $h_k =$ **?**
When $\mathrm{EST}_k < \mathrm{TOL}$ : stepsize prediction good $h_{k+1} =$ **?**

Assumption: At our disposal are two discrete evolutions:

- $\boldsymbol{\Psi}$ with order$(\boldsymbol{\Psi}) = p$ (➔ "low order" single step method)
- $\widetilde{\boldsymbol{\Psi}}$ with order$(\widetilde{\boldsymbol{\Psi}}) > p$ (➔ "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 $h \to 0$:

$$\begin{aligned} \boldsymbol{\Psi}^{h_k}\mathbf{y}(t_k) - \boldsymbol{\Phi}^{h_k}\mathbf{y}(t_k) &= ch^{p+1} + O(h_k^{p+2}) \,, \\ \widetilde{\boldsymbol{\Psi}}^{h_k}\mathbf{y}(t_k) - \boldsymbol{\Phi}^{h_k}\mathbf{y}(t_k) &= O(h^{p+2}) \,, \end{aligned} \tag{8.7.5}$$

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 ➥ "higher order terms" $O(h^{p+2})$ can be ignored.

$$\begin{aligned} \boldsymbol{\Psi}^{h_k}\mathbf{y}(t_k) - \boldsymbol{\Phi}^{h_k}\mathbf{y}(t_k) &\doteq ch_k^{p+1} + O(h_k^{p+2}) \,, \\ \widetilde{\boldsymbol{\Psi}}^{h_k}\mathbf{y}(t_k) - \boldsymbol{\Phi}^{h_k}\mathbf{y}(t_k) &\doteq O(h_k^{p+2}) \,. \end{aligned} \quad \Rightarrow \quad \boxed{\mathrm{EST}_k \doteq ch_k^{p+1}} \,. \tag{8.7.6}$$

notation: $\doteq$ equality up to higher order terms in $h_k$

$$\mathrm{EST}_k \doteq ch_k^{p+1} \quad \Rightarrow \quad c \doteq \frac{\mathrm{EST}_k}{h_k^{p+1}} \,. \tag{8.7.7}$$

Available in algorithm, see (8.7.3)



For the sake of *accuracy* (stipulates "$\mathrm{EST}_k < \mathrm{TOL}$") & *efficiency* (favors "$>$") we aim for

$$\mathrm{EST}_k \overset{!}{=} \mathrm{TOL} := \max\{\mathrm{ATOL}, \|\mathbf{y}_k\| \mathrm{RTOL}\} \,. \tag{8.7.8}$$

What timestep $h_*$ can actually achieve (8.7.8), if we "believe" in (8.7.6) (and, therefore, in (8.7.7))?

$$(8.7.7) \ \& \ (8.7.8) \quad \Rightarrow \quad \mathrm{TOL} = \frac{\mathrm{EST}_k}{h_k^{p+1}} h_*^{p+1} \,.$$

"'Optimal timestep": (stepsize prediction)

$$\boxed{h_* = h \sqrt[p+1]{\frac{\mathrm{TOL}}{\mathrm{EST}_k}}} \,. \tag{8.7.9}$$

adjusted stepsize (**A**)

suggested stepsize (**B**)





(**A**): In case $\mathrm{EST}_k > \mathrm{TOL}$ ➢ *repeat step* with stepsize $h_*$.

(**B**): If $\mathrm{EST}_k \leq \mathrm{TOL}$ ➢ use $h_*$ as stepsize for *next step*.

Code 8.7.10: refined local stepsize control for single step methods

```
from numpy import *
from numpy.linalg import norm
import matplotlib.pyplot as plt

def odeintssctrl(Psilow,p,Psihigh,T,y0,h0,reltol,abstol,hmin):
        t = [0]; y = [y0]; h = h0        #
        while t[-1] < T and h > hmin:    #
                yh = Psihigh(h,y0)       #
                yH = Psilow(h,y0)        #
                est = norm(yH-yh)        #
                tol = max(reltol*norm(y[-1]),abstol)     #
                h = h*max(0.5, min(2,(tol/est)**(1.0/(p+1))))    #
                if est < tol:    #
                        y0 = yh; y.append(y0);
                          t.append(t[-1]+min(T-t[-1],h))    #

        return (array(t),array(y)) # convert to numpy array

if __name__ == '__main__':
        """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 ẏ = 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 trapzoidal 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.





- line 12: compute presumably better local stepsize according to (8.7.9),
- line 13: decide whether to repeat the step or advance,
- line 14: extend output arrays if current step has not been rejected.

*Remark* 8.7.11 (Stepsize control in ode45).

$\Psi \hat{=}$ RK-method of order 4 $\qquad$ $\widetilde{\Psi} \hat{=}$ RK-method of order 5

ode45

Specifying tolerances for integrators: `options = {'reltol':1.0e-10, 'abstol':1.0e-1 'stats':'on'}`

`[t,y] = ode45(f,tspan,y0,**options)`

(`f` = function handle, `tspan` $\hat{=} [t_0, T]$, `y0` $\hat{=} \mathbf{y}_0$, `t` $\hat{=} t_k$, `y` $\hat{=} \mathbf{y}_k$)

△

*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: $$\ddot{\mathbf{y}} = F(\mathbf{y}) := -\frac{2\mathbf{y}}{\|\mathbf{y}\|_2^2} \,. \tag{8.7.10}$$

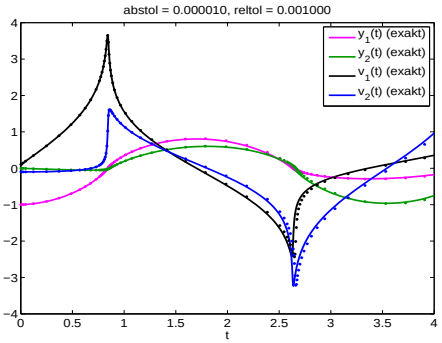
acceleration $\qquad$ force

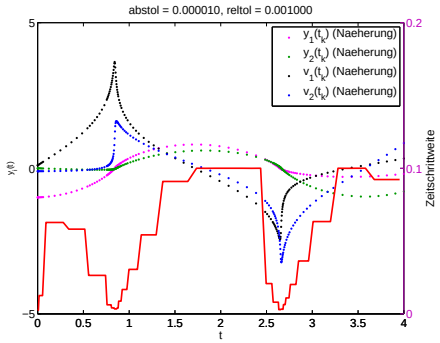
Equivalent 1st-order ODE, see Rem. 8.1.6: with velocity $\mathbf{v} := \dot{\mathbf{y}}$

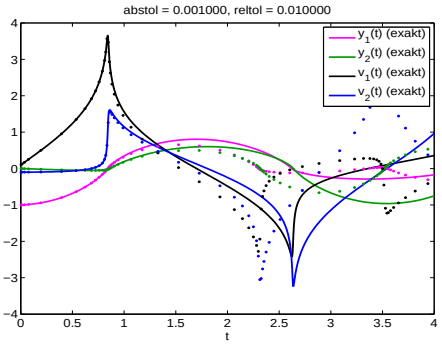
$$\begin{pmatrix} \dot{\mathbf{y}} \\ \dot{\mathbf{v}} \end{pmatrix} = \begin{pmatrix} \mathbf{v} \\ -\frac{2\mathbf{y}}{\|\mathbf{y}\|_2^2} \end{pmatrix} \,. \tag{8.7.11}$$

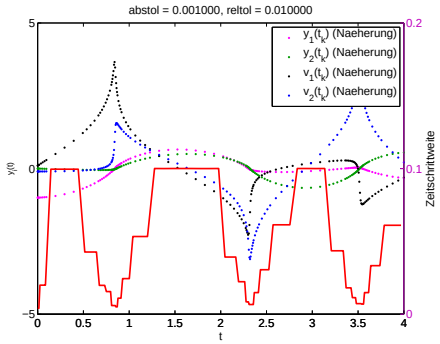
Initial values used in the experiment:

$$\mathbf{y}(0) := \begin{pmatrix} -1 \\ 0 \end{pmatrix} \quad , \quad \mathbf{v}(0) := \begin{pmatrix} 0.1 \\ -0.1 \end{pmatrix}$$

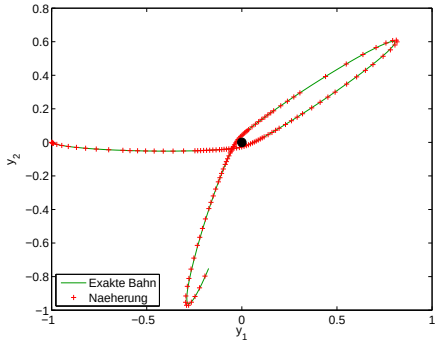


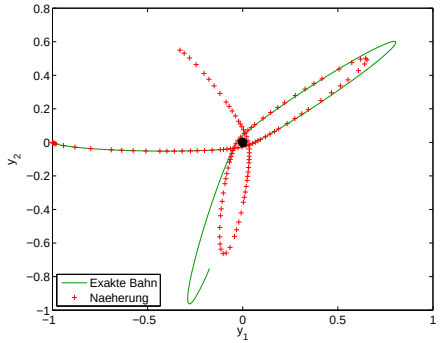






reltol=0.001, abstol=1e-5




reltol=0.01, abstol=1e-3

Observations:

☞ Fast changes in solution components captured by adaptive approach through very small timesteps.

☞ Completely wrong solution, if tolerance reduced slightly.





◇

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



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 (→ 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).

IVP: $\dot{y} = \lambda y^2(1-y)$ , $\lambda := 500$ , $y(0) = \frac{1}{100}$ .

Code 9.0.2: ode45 for stiff problem

```
import matplotlib.pyplot as plt
from numpy import diff, finfo, double
from ode45 import ode45

def ode45stiff():
```

Num. Meth. Phys. Gradinaru D-MATH 8.8 p. 545

```
    """tries to solve the logistic differential equation using ode45 with
        adaptive stepsize control and draws the results:
        resulting data points -> logode451.eps
        timestep size         -> logode452.eps
        both plots also contain the 'exact' solution
    """

    # define the differential equation
    fun = lambda t,x: 500*x**2*(1-x)
    # define the time span
    tspan = (0.0,1.0)
    # get total time span
    L = diff(tspan)
    # starting value
    y0 = 0.01

    # make the figure
    fig = plt.figure()

    # exact solution
    # set options
    eps = finfo(double).eps
    options = {'reltol':10*eps,'abstol':eps,'stats':'on'}
```

Num. Meth. Phys. Gradinaru D-MATH 9.0 p. 546

```
    tex,yex = ode45(fun,(0,1),y0,**options)
    # plot 'exact' results
    plt.plot(tex,yex,'--',color=[0,0.75,0],linewidth=2,label='y(t)')
    plt.hold(True)

    # ODE45
    # set options
    options = {'reltol':0.01,'abstol':0.001,'stats':'on'}
    # get the solution using ode45
    [t,y] = ode45(fun,(0,1),y0,**options)
    # plot the solution
    plt.plot(t,y,'r+',label='ode45')

    # set label, legend, ....
    plt.xlabel('t',fontsize=14)
    plt.ylabel('y')
    plt.title('ode45 for d_ty = 500 y^2(1-y)')
    plt.show()
    # write to file
    # plt.savefig('../PICTURES/logode451.eps')

    # now plot stepsizes
    fig = plt.figure()
    plt.title('ode45 for d_ty = 500 y^2(1-y)')
```

Num. Meth. Phys. Gradinaru D-MATH 9.0 p. 54

```
        ax1 = fig.add_subplot(111)
        ax1.plot(tex,yex[:,2], 'r--', linewidth=2)
        ax1.set_xlabel('t')
        ax1.set_ylabel('y(t)')

        ax2 = ax1.twinx()
        ax2.plot(0.5*(t[:-1]+t[1:]),diff(t), 'r-')
        ax2.set_ylabel('stepsize')

        plt.show()

        # write to file
        #plt.savefig('../PICTURES/logode452.eps')

if __name__ == '__main__':
        print 'warning: this function takes a long time to complete! (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
```

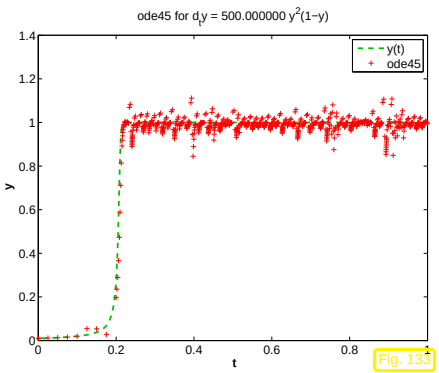


Fig. 133

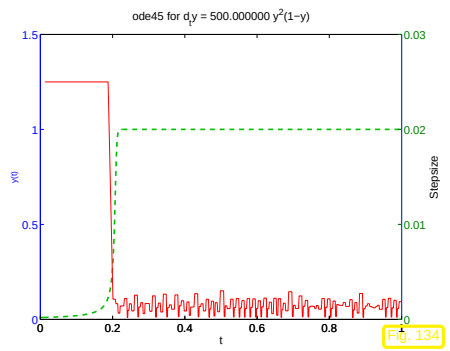


Fig. 134

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

$$\dot{y} = f(y) := \lambda y(1-y) \quad , \quad y(0) = 0.01 \; .$$

- Explicit Euler method (8.2.1) with uniform timestep $h = 1/N$, $N \in \{5, 10, 20, 40, 80, 160, 320, 640\}$.





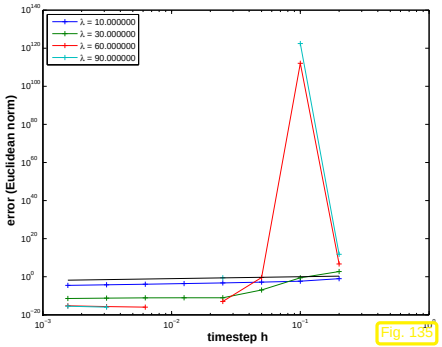


Fig. 135

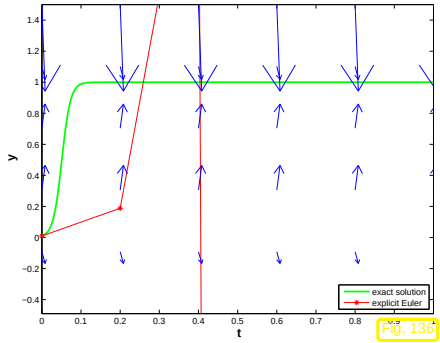


Fig. 136

$\lambda$ large: blow-up of $y_k$ for large timestep $h$

$\lambda = 90$: — $\hat{=}$ $y(t)$, — $\hat{=}$ Euler polygon



Explanation: $y_k$ way miss the stationary point $y = 1$ (overshooting).

This leads to a sequence $(y_k)_k$ with exponentially increasing oscillations.

**Deeper analysis:**

For $y \approx 1$: $f(y) \approx \lambda(1-y)$ ➢ If $y(t_0) \approx 1$, then the solution of the IVP will behave like the solution of $\dot{y} = \lambda(1-y)$, which is a linear ODE. Similary, $z(t) := 1 - y(t)$ will bahave 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

$$\text{linear model problem:} \quad \dot{y} = \lambda y \ , \quad y(0) = y_0 \ , \quad \text{with} \quad \lambda \ll 0 \ , \tag{9.1.1}$$

and *exponentially decaying* exact solution

$$y(t) = y_0 \exp(\lambda t) \to 0 \quad \text{for } t \to \infty \ .$$

Recursion of explicit Euler method for (9.1.1):

$$\text{(8.2.1) for } f(y) = \lambda y: \qquad y_{k+1} = y_k(1 + \lambda h) \ . \tag{9.1.2}$$

$$\blacktriangleright \quad y_k = y_0(1+\lambda h)^k \quad \Rightarrow \quad |y_k| \to \begin{cases} 0 & \text{, if } \lambda h > -2 \quad \text{(qualitativly correct)} \ , \\ \infty & \text{, if } \lambda h < -2 \quad \text{(qualitativly wrong)} \ . \end{cases}$$

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

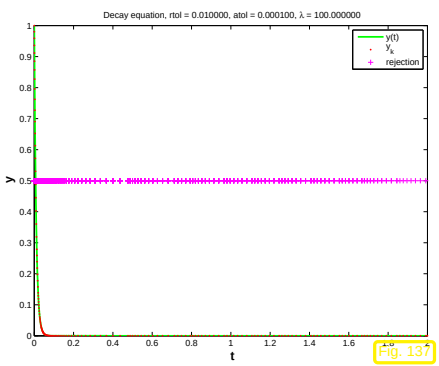


Fig. 137

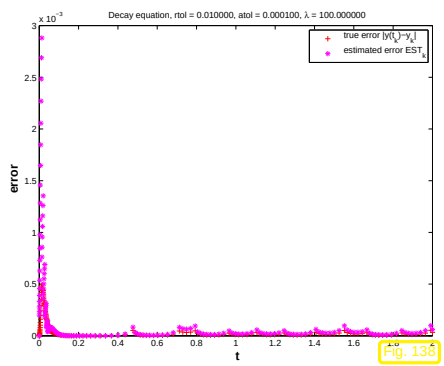


Fig. 138

Observation: in fact, stepsize control enforces small timesteps even if $\mathbf{y}(t) \approx 0$ and persistently triggers rejections of timesteps. This is necessary to prevent overshooting in the Euler method, which contributes to the estimate of the one-step error.





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 trapzoidal rule for decay equation).

Recall recursion for explicit trapezoidal rule:

$$\mathbf{k}_1 = \mathbf{f}(t_0, \mathbf{y}_0) \ , \quad \mathbf{k}_2 = \mathbf{f}(t_0 + h, \mathbf{y}_0 + h\mathbf{k}_1) \ , \quad \mathbf{y}_1 = \mathbf{y}_0 + \tfrac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2) \ . \tag{8.6.3}$$

Apply this to the model problem (9.1.1), that is $\mathbf{f}(y) = f(y) = \lambda y$, $\lambda < 0$:

$$\blacktriangleright \quad k_1 = \lambda y_0 \ , \quad k_2 = \lambda(y_0 + h k_1) \quad \Rightarrow \quad y_1 = \underbrace{(1 + \lambda h + \tfrac{1}{2}(\lambda h)^2)}_{=:S(h\lambda)} y_0 \ . \tag{9.1.3}$$

$\blacktriangleright$ sequence generated by explicit trapezoidal rule:

$$y_k = S(h\lambda)^k y_0 \ , \quad k = 0, \dots, N \ . \tag{9.1.4}$$

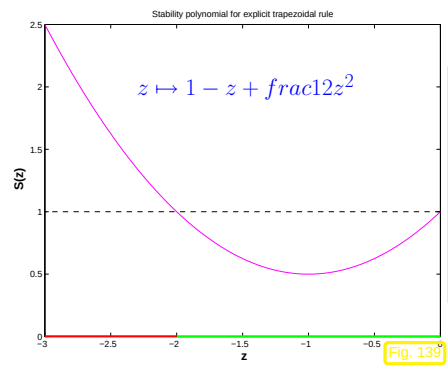


Fig. 139

$$|S(h\lambda)| < 1 \quad \Leftrightarrow \quad -2 < h\lambda < 0 \ .$$

Qualitatively correct decay behavior of $(y_k)_k$ only under timestep constraint

$$h \le |2/\lambda| \ . \tag{9.1.5}$$

Mode problem analysis for general explicit Runge-Kutta method ($\to$ Def. 8.6.1): apply Runge-Kutta method $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$ to (9.1.1)

$$\blacktriangleright \quad \begin{aligned} k_i &= \lambda(y_0 + h \sum_{j=1}^{i-1} a_{ij} k_j) \ , \\ y_1 &= y_0 + h \sum_{i=1}^{s} b_i k_i \end{aligned} \quad \Rightarrow \quad \begin{pmatrix} \mathbf{I} - z\mathfrak{A} & 0 \\ -z\mathbf{b}^T & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{k} \\ y_1 \end{pmatrix} = y_0 \begin{pmatrix} \mathbf{1} \\ 1 \end{pmatrix} \ , \tag{9.1.6}$$

where $\boldsymbol{k} \in \mathbb{R}^s \hat{=}$ denotes the vector $(k_1, \dots, k_s)^T/\lambda$ of increments, and $z := \lambda h$.

$$y_1 = S(z)y_0 \quad \text{with} \quad S(z) := 1 + z\mathbf{b}^T(\mathbf{I} - z\mathfrak{A})^{-1}\mathbf{1} = \det(\mathbf{I} - z\mathfrak{A} + z\mathbf{1}\mathbf{b}^T) \,. \tag{9.1.7}$$

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 $\Psi_\lambda^h$ of an explicit $s$-stage Runge-Kutta single step method (→ 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*

$$\Psi_\lambda^h = \underbrace{1 + z\mathbf{b}^T(\mathbf{I} - z\mathfrak{A})^{-1}\mathbf{1}}_{\text{stability function } S(z)} = \det(\mathbf{I} - z\mathfrak{A} + z\mathbf{1}\mathbf{b}^T) \,, \quad z := \lambda h \,, \quad \mathbf{1} = (1, \dots, 1)^T \in \mathbb{R}^s \,.$$

Thm. 9.1.1 $\Rightarrow$ $S \in \mathcal{P}_s$

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 $y_k = S(\lambda h)^k y_0$.

$$\boxed{(|y_k|)_{k=0}^\infty \text{ non-increasing} \quad \Leftrightarrow \quad |S(\lambda h)| \le 1} \,, \tag{9.1.8}$$
$$(|y_k|)_{k=0}^\infty \text{ exponentially increasing} \quad \Leftrightarrow \quad |S(\lambda h)| > 1 \,.$$

On the other hand: $\forall S \in \mathcal{P}_s$: $\lim\limits_{|z| \to \infty} |S(z)| = \infty$

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

$$\dot{\mathbf{y}} = \underbrace{\begin{pmatrix} 0 & 1 \\ -\beta & -\alpha \end{pmatrix}}_{=:\mathbf{M}} \mathbf{y} \quad , \quad \mathbf{y}(0) = \mathbf{y}_0 \in \mathbb{R}^2 \,. \tag{9.2.1}$$

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):

$$\mathbf{M}\mathbf{V} = \mathbf{M}\,(\mathbf{v}_1, \mathbf{v}_2) = (\mathbf{v}_1, \mathbf{v}_2) \begin{pmatrix} \lambda_1 & \\ & \lambda_2 \end{pmatrix} \,. \tag{9.2.2}$$

$\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^2 \setminus \{0\} \hat{=}$ 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!

$$\dot{\mathbf{y}} = \mathbf{M}\mathbf{y} \quad \Leftrightarrow \quad \mathbf{V}^{-1}\dot{\mathbf{y}} = \mathbf{V}^{-1}\mathbf{M}\mathbf{V}(\mathbf{V}^{-1}\mathbf{y}) \quad \overset{\mathbf{z}(t) := \mathbf{V}^{-1}\mathbf{y}(t)}{\Leftrightarrow} \quad \dot{\mathbf{z}} = \begin{pmatrix} \lambda_1 & \\ & \lambda_2 \end{pmatrix} \mathbf{z} \,. \tag{9.2.3}$$

This yields the general solution of the ODE $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$

$$\mathbf{y}(t) = A\mathbf{v}_1 \exp(\lambda_1 t) + B\mathbf{v}_2 \exp(\lambda_2 t) \,, \quad A, B \in \mathbb{R} \,. \tag{9.2.4}$$

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 $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$

$$\mathbf{\Psi}^h\mathbf{y} = \mathbf{y} + h\mathbf{M}\mathbf{y} \quad \leftrightarrow \quad \mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{M}\mathbf{y}_k \ .$$

Perform the same transformation as above on the discrete evolution:

$$\mathbf{V}^{-1}\mathbf{y}_{k+1} = \mathbf{V}^{-1}\mathbf{y}_k + h\mathbf{V}^{-1}\mathbf{M}\mathbf{V}(\mathbf{V}^{-1}\mathbf{y}_k) \overset{\mathbf{z}_k := \mathbf{V}^{-1}\mathbf{y}_k}{\Leftrightarrow} \underbrace{(\mathbf{z}_{k+1})_i = (\mathbf{z}_k)_i + h\lambda_i(\mathbf{z}_k)_i}_{\hat{=}\text{ explicit Euler step for } \dot{z}_i = \lambda_i z_i} \ . \quad (9.2.5)$$

Crucial insight:

> The explicit Euler method generates uniformly bounded solution sequencies $(\mathbf{y}_k)_{k=0}^{\infty}$ 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 sequencies for **all** the scalar ODEs $\dot{z} = \lambda_i \dot{z}$, $i = 1, \dots, d$.

> An analoguous 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/explicity RK-methods respond to them.

*Example* 9.2.1 (Explicit Euler method for damped oscillations).

Consider linear model IVP (9.1.1) for $\lambda \in \mathbb{C}$:

$$\operatorname{Re}\lambda < 0 \quad \Rightarrow \quad \text{exponentially decaying solution} \quad y(t) = y_0 \exp(\lambda t) \ ,$$

because $|\exp(\lambda t)| = \exp(\operatorname{Re}\lambda t)$.

Model problem analysis (→ 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):

$$y_{k+1} = y_k(1 + h\lambda) \ . \quad (9.1.2)$$

$$\lim_{k\to\infty} y_k = 0 \quad \Leftrightarrow \quad |1 + h\lambda| < 1 \ .$$

timestep constraint to get decaying (discrete) solution !

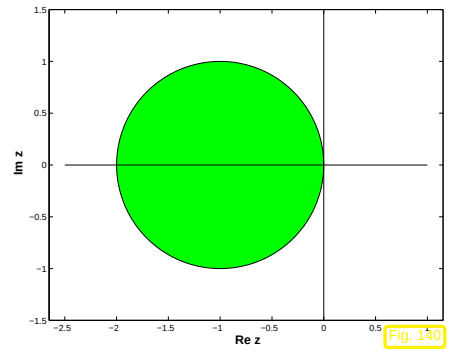



Fig. 140

◁ $\{z \in \mathbb{C}: \ |1 + z| < 1\}$

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).

$$\text{reaction:} \quad \underbrace{\mathrm{A} + \mathrm{B} \underset{k_1}{\overset{k_2}{\rightleftarrows}} \mathrm{C}}_{\text{fast reaction}} \quad , \quad \underbrace{\mathrm{A} + \mathrm{C} \underset{k_3}{\overset{k_4}{\rightleftarrows}} \mathrm{D}}_{\text{slow reaction}} \quad (9.2.6)$$

Vastly different reaction constants: $\boxed{k_1, k_2 \gg k_3, k_4}$

► If $c_A(0) > c_B(0)$ ➢ 2nd reaction determines overall long-term reaction dynamics

Mathematical model: ODE involving concentrations $\mathbf{y}(t) = (c_A(t), c_B(t), c_C(t), c_D(t))^T$

$$\dot{\mathbf{y}} := \frac{d}{dt}\begin{pmatrix} c_A \\ c_B \\ c_C \\ c_D \end{pmatrix} = \mathbf{f}(\mathbf{y}) := \begin{pmatrix} -k_1c_Ac_B + k_2c_C - k_3c_Ac_C + k_4c_D \\ -k_1c_Ac_B + k_2c_C \\ k_1c_Ac_B - k_2c_C - k_3c_Ac_C + k_4c_D \\ k_3c_Ac_C - k_4c_D \end{pmatrix} \ .$$

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



```
from numpy import array, linspace, double, diff
from ode45 import ode45
```

```python
import matplotlib.pyplot as plt
from odewrapper import odewrapper

def chemstiff():
        """Simulation of kinetics of coupled chemical reactions with vastly different reaction
        rates, see \eqref{eq:chemstiff} for the ODE model.
        reaction rates \Blue{k_1, k_2, k_3, k_4}, \Magenta{k_1, k_2 ≫ k_3, k_4}.
        """

        k1 = 1e4; k2 = 1e3; k3 = 10; k4 = 1
        # definition of right hand side function for ODE solver
        fun = lambda t,y: array([-k1*y[0]*y[1] + k2*y[2] - k3*y[0]*y[2] + k4*y[3],
                                 -k1*y[0]*y[1] + k2*y[2],
                                 k1*y[0]*y[1] - k2*y[2] - k3*y[0]*y[2] + k4*y[3],
                                 k3*y[0]*y[2] - k4*y[3]])
        tspan = [0, 1] # Integration time interval
        L = tspan[1]-tspan[0] # Duration of simulation
        y0 = array([1,1,10,0]) # Initial value y_0

        # compute "exact" solution, using fortran routines (using wrapper for scipy.integrate.ode class)
        tex = linspace(0,1,201)
```



```python
        yex = odewrapper(fun,y0,0,tex)

        # Compute solution with ode45 and moderate tolerances
        options = {'RelTol':0.01, 'AbsTol':0.001, 'Stats':'on'}
        t,y = ode45(fun,[0, 1],y0,**options)
        # plot the solution
        plt.figure()
        plt.plot(t,y[:,0],'b.', label=r'c_{A,k}, low tolerance')
        plt.plot(t,y[:,2],'r.', label=r'c_{C,k}, low tolerance')
        # plot 'exact' results
        plt.plot(tex,yex[:,0],'c--',linewidth=3, label=r'c_A(t)')
        plt.plot(tex,yex[:,2],'m--',linewidth=3, label=r'c_C(t)')
        plt.axis([0,1,0,12])
        plt.xlabel('t')
        plt.ylabel('concentrations')
        plt.title('Chemical reaction: concentrations')
        plt.legend()
# plt.savefig('.../PICTURES/chemstiff.eps')

        # Plot stepsizes together with "exact" solution
        fig = plt.figure()
        plt.title('Chemical reaction: stepsize')
        ax1 = fig.add_subplot(111)
        ax1.plot(tex,yex[:,2], 'r--', linewidth=2)
```



```python
        ax1.set_xlabel('t')
        ax1.set_ylabel('c_C(t)')

        ax2 = ax1.twinx()
        ax2.plot(0.5*(t[:-1]+t[1:]),diff(t), 'r-')
        ax2.set_ylabel('timestep')

        plt.show()
        #plt.savefig('chemstiffss.eps')

if __name__ == '__main__':
        chemstiff()
```



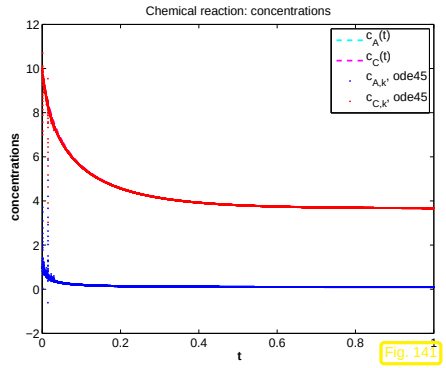



Fig. 141

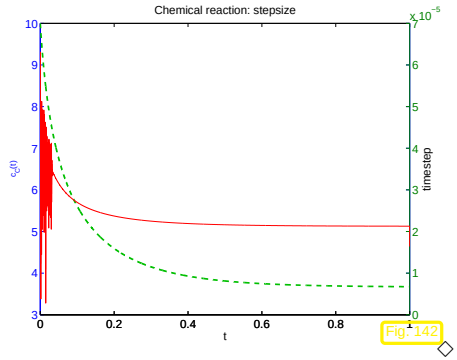



Fig. 142

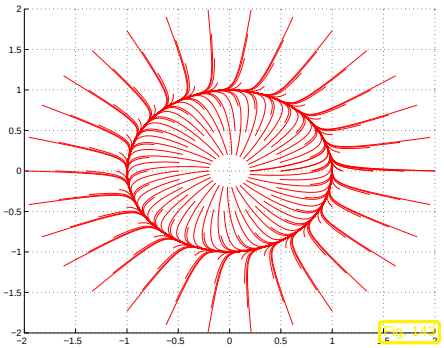
*Example* 9.2.4 (Strongly attractive limit cycle).

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 ($\lambda = 10$) ▷

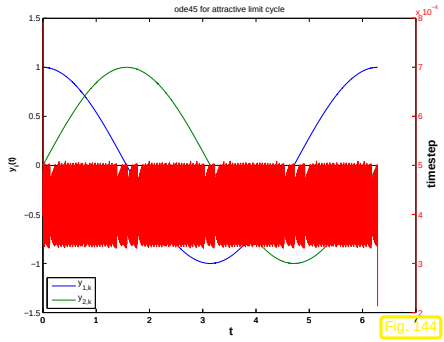



Code 9.2.5: Integration of IVP with limit cycle

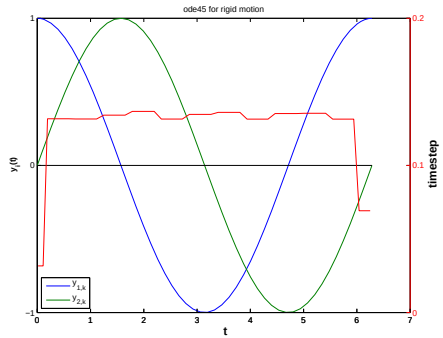
```
from ode45 import ode45
from numpy import *
import matplotlib.pyplot as plt
def limitcycle(lam,y0=array((1,0))):
        # PYTHON script for solving limit cycle ODE (??)
        # define right hand side vectorfield
        fun = lambda t,y: array([-y[1],y[0]]) +
          lam*(1-y[0]**2-y[1]**2)*y
        # standard invocation of MATLAB integrator, see Ex. 8.6.7
        tspan = (0,2*pi)
        opts = {'stats':False,'reltol':1e-4,'abstol':1e-4}
        t45,y45 = ode45(fun,tspan,y0,**opts)
        return (t45,y45)

if __name__ == '__main__':
        for lam in [0,1000]:
                t,y = limitcycle(lam)
                fig = plt.figure()
                plt.title(r'λ =' +str(lam)+'')
                ax1 = fig.add_subplot(111)
                ax1.plot(t,y)
                ax1.set_ylabel(r'y_i(t)')
                ax1.set_xlabel('t')
                ax2 = ax1.twinx()
                ax2.plot(0.5*(t[:-1]+t[1:]),diff(t), 'r-')
                ax2.set_ylabel('timestep')
                plt.show()

        # now make some nice solution trajectories
        lam = 10
        plt.figure()
        plt.title(r'Solution Trajectories (λ = 10)')
        for rad in r_[0.25:2.25:0.25]:
                for phi in r_[0:2*pi:31j][:-1]:
                        x,y = rad*cos(phi),rad*sin(phi)
                        t,y = limitcycle(lam,array([x,y]))
                        plt.plot(y[:,0],y[:,1],'r')
        plt.grid(True)
        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 $\lambda$ will always vanish on the exact solution trajectory, which stays on the unit circle.

Nevertheless, `ode45` is forced to use tiny timesteps by the mere presence of this term.

◇

**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 stronly attractive fixed points/limit cycles, see Ex. 9.2.4

Code 9.2.6: An ODE wrapper

```
import scipy.integrate
from numpy import size, zeros
```

```
def odewrapper(f,y0,t0,t,integrator='dopri5'):
        """integrates the function f with initial values y0, t0
        returns the value of the integral at times specified in t
        'dopri5' is the scipy.integrate analog of matlab's ode45
        'ode15s' is the scipy.integrate analog of matlab's ode15s
        """
        # initialize ode class:
        # constructor takes function to integrate
        # second function sets initial values (for y and for t)
        # third function sets integrator (default:'dopri5' is equivalent to ode45 in MATLAB)
        if integrator=='dopri5':
                r =
                  scipy.integrate.ode(f).set_initial_value(y0,t0).set_int
        if integrator == 'ode15s':
                print integrator+'\n'
                r =
                  scipy.integrate.ode(f).set_initial_value(y0,t0).set_int
                   method='bdf', order=15)
        # do the integration, read values at points in array t.
        i=1; n=size(t)
        y = zeros([n,size(y0)])
        y[0]=y0
        while r.successful() and i < n:
                r.integrate(t[i])
                y[i] = r.y
                i += 1
        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)

$$\blacktriangleright \quad \text{sequence} \qquad y_k := \left(\frac{1}{1-\lambda h}\right)^k y_0 \,. \tag{9.3.1}$$

$$\Rightarrow \quad \boxed{\operatorname{Re}\lambda < 0 \quad \Rightarrow \quad \lim_{k\to\infty} y_k = 0\,!} \tag{9.3.2}$$

**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 imlicit* Euler method alike.

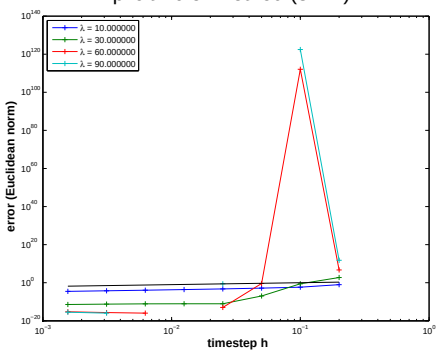
Conjecture: implicit Euler method will not face timestep constraint for stiff problems (→ Notion 9.2.1).

*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)




$\lambda$ large: blow-up of $y_k$ for large timestep $h$

Implicit Euler method (8.2.4)

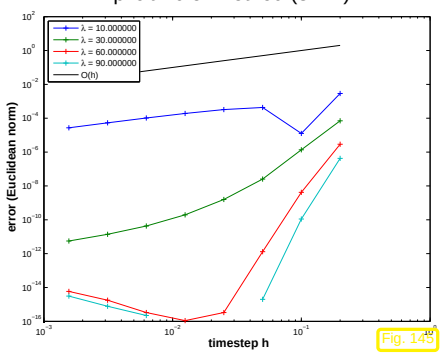



Fig. 145

$\lambda$ large: stable for all timesteps $h$ !

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*

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + h\sum_{j=1}^{s} a_{ij}\mathbf{k}_j) \;, \quad i = 1,\dots,s \quad , \quad \mathbf{y}_1 := \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{k}_i \;.$$

*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 $s \cdot d$ ($\longrightarrow$ "implicit" method)

Shorthand notation for Runge-Kutta methods

Butcher scheme $\triangleright$

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array} \;. \tag{9.3.3}$$

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 ($\longrightarrow$ 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 $\Psi_\lambda^h$ of an $s$-stage Runge-Kutta single step method ($\longrightarrow$ Def. 9.3.1) with Butcher scheme* $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$ *(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 represenation of $S(z)$ we infer that the stability function of an $s$-stage Runge-Kutta method is a rational function of the form $S(z) = \dfrac{P(z)}{Q(z)}$ 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

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \overset{\text{Thm. 9.3.2}}{\Longrightarrow} S(z) = \frac{1}{1-z} \;. \tag{9.3.4}$$

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 ($\longrightarrow$ Notion9.2.1):

**Definition 9.3.3** (L-stable Runge-Kutta method).
*A Runge-Kutta method ($\longrightarrow$ Def. 9.3.1) is L-stable/asymptotically stable, if its stability function ($\longrightarrow$ Def. 9.3.2) satisfies*

$$(i) \qquad \operatorname{Re} z < 0 \;\Rightarrow\; |S(z)| < 1 \;, \tag{9.3.5}$$

$$(ii) \qquad \lim_{\operatorname{Re} z \to -\infty} S(z) = 0 \;. \tag{9.3.6}$$

*Remark* 9.3.3 (Necessary condition for L-stability of Runge-Kutta methods).

Consider: Runge-Kutta method ($\longrightarrow$ 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 $|z| \to \infty$ exists and can easily be expressed by the leading coefficients of the polynomials $P$ and $Q$:

$$\text{Thm. 9.3.2} \;\Rightarrow\; S(-\infty) = 1 - \mathbf{b}^T\mathfrak{A}^{-1}\mathbf{1} \;. \tag{9.3.7}$$

$$\blacktriangleright \quad \boxed{\text{If } \mathbf{b}^T = (\mathfrak{A})^T_{:,j} \text{ (row of } \mathfrak{A}) \;\Rightarrow\; S(-\infty) = 0} \;. \tag{9.3.8}$$

Butcher scheme (9.3.3) for L-stable RK-methods, see Def. 9.3.3

$$\triangleright \quad \begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_{s-1} & a_{s-1,1} & \cdots & a_{s-1,s} \\ 1 & b_1 & \cdots & b_s \\ \hline & b_1 & \cdots & b_s \end{array} .$$

*Example* 9.3.4 (L-stable implicit Runge-Kutta methods).

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \qquad \begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array} \qquad \begin{array}{c|ccc} \frac{4-\sqrt{6}}{10} & \frac{88-7\sqrt{6}}{360} & \frac{296-169\sqrt{6}}{1800} & \frac{-2+3\sqrt{6}}{225} \\ \frac{4+\sqrt{6}}{10} & \frac{296+169\sqrt{6}}{1800} & \frac{88+7\sqrt{6}}{360} & \frac{-2-3\sqrt{6}}{225} \\ 1 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \\ \hline & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \end{array}$$

Implicit Euler method | Radau RK-SSM, order 3 | 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

$$\dot{y} = \lambda y(1-y) \quad , \quad y(0) = 0.1 \quad , \quad \lambda = 5 .$$

- 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 $\mathrm{err} = \max_{j=1,\dots,n} |y_j - y(t_j)|$

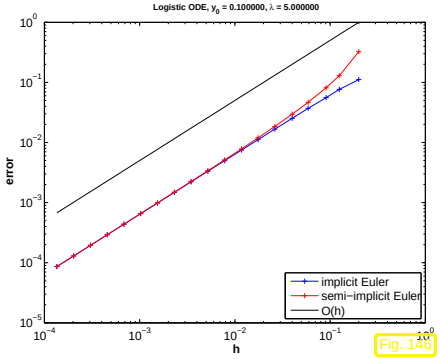



Fig. 140

From (8.2.4) with timestep $h > 0$

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}(\mathbf{y}_{k+1}) \quad \Leftrightarrow \quad F(\mathbf{y}_{k+1}) := \mathbf{y}_{k+1} - h\mathbf{f}(\mathbf{y}_{k+1}) - \mathbf{y}_k = 0 .$$

One Newton step applied to $F(\mathbf{y}) = 0$ with initial guess $\mathbf{y}_k$ yields

$$\mathbf{y}_{k+1} = \mathbf{y}_k - D\mathbf{f}(\mathbf{y}_k)^{-1} F(\mathbf{y}_k) = \mathbf{y}_k + (\mathbf{I} - hD\mathbf{f}(\mathbf{y}_k))^{-1} h\mathbf{f}(\mathbf{y}_k) .$$





$$\mathbf{f}(\mathbf{y}) = \mathbf{A}\mathbf{y} \quad \mathbf{A} \in \mathbb{R}^{d,d}$$

$$y_{k+1}$$

◇

Idea: Use linearized increment equations for implicit RK-SSM

$$\mathbf{k}_i = \mathbf{f}(\mathbf{y}_0) + hD\mathbf{f}(\mathbf{y}_0)\left(\sum_{j=1}^{s} a_{ij}\mathbf{k}_j\right) \quad , \quad i = 1, \dots, s . \tag{9.3.9}$$

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):

$$(\mathbf{I} - ha_{ii}\mathbf{J})\mathbf{k}_i = \mathbf{f}(\mathbf{y}_0 + h\sum_{j=1}^{i-1}(a_{ij} + d_{ij})\mathbf{k}_j) - h\mathbf{J}\sum_{j=1}^{i-1} d_{ij}\mathbf{k}_j , \tag{9.3.10}$$

$$\mathbf{J} := D\mathbf{f}\big(\mathbf{y}_0 + h\sum_{j=1}^{i-1}(a_{ij} + d_{ij})\mathbf{k}_j\big) , \tag{9.3.11}$$

$$\mathbf{y}_1 := \mathbf{y}_0 + \sum_{j=1}^{s} b_j\mathbf{k}_j . \tag{9.3.12}$$

*Remark* 9.3.6 (Adaptive integrator for stiff problems in MATLAB).

Handle of type `@(t,y) J(t,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:

$\Psi \mathrel{\hat{=}}$ RK-method of order 2 | $\widetilde{\Psi} \mathrel{\hat{=}}$ 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. Symbgolic 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/m̃hg/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.



Num. Meth. Phys.
Gradina D-MAT
9.4 p. 58
Num. Meth. Phys.
Gradina D-MAT
9.4 p. 58

[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

# List of Symbols









# List of Definitions

# Examples and Remarks