Ivo F. Sbalzarini

Basic Numerical Methods

Lecture Notes

TU Dresden, Faculty of Computer Science
Chair of Scientific Computing for Systems Biology

Prof. Dr. sc. techn. Dipl. Masch.-Ing. ETH Ivo F. Sbalzarini
Center for Systems Biology Dresden, TUD & MPI-CBG

Winter 2021/22

This page is intentionally left blank

Contents

Contents **v**

Foreword **ix**

1 Foundations of Numerical Computation **1**
1.1 Floating-Point Number Representation 1
1.2 Integer Number Representation 4
1.3 Rounding 4
1.4 Pseudo Arithmetic 6
1.5 Error Propagation 7
1.5.1 Extinction 8
1.5.2 Condition number of a function 9
1.5.3 Condition number of a matrix 11
1.5.4 Condition number of an algorithm 14
1.6 Backward Error Analysis 15

2 Linear System Solvers **17**
2.1 Gaussian Elimination – LU Decomposition 17
2.1.1 Pivoting strategies 19
2.2 Linear Fixed-Point Iterations 21
2.2.1 Convergence criteria 22
2.2.2 Jacobi method 24
2.2.3 Gauss-Seidel method 25
2.2.4 Successive Over-Relaxation (SOR) method 26
2.3 Gradient Descent 27
2.3.1 Geometric interpretation 28
2.3.2 Algorithm 30
2.4 Conjugate Gradient Method 31
2.5 Pre-Conditioning 36

3 Linear Least-Squares Problems **37**
3.1 Error Equations and Normal Equations 37
3.2 Solution by QR Decomposition 39
3.3 Singular Value Decomposition 42

3.3.1 Properties of the singular value decomposition 42
3.4 Solution by Singular Value Decomposition 44
3.5 Nonlinear Least-Squares Problems 45

4 Solving Nonlinear Equations 47
4.1 Condition Number of the Problem 47
4.2 Newton's Method 49
4.2.1 The fixed-point theorem 50
4.2.2 Convergence of Newton's method 53
4.2.3 Algorithm 53
4.3 Secant Method 54
4.4 Bisection Method 55

5 Nonlinear System Solvers 59
5.1 Newton's Method in Arbitrary Dimensions 59
5.1.1 Quasi-Newton method 61
5.2 Broyden Method 61

6 Scalar Polynomial Interpolation 65
6.1 Uniqueness 66
6.2 Existence: The Lagrange Basis 67
6.3 Barycentric Representation 68
6.4 Aitken-Neville Algorithm 70
6.5 Approximation Error 71
6.6 Spline Interpolation 73
6.6.1 Hermite interpolation 73
6.6.2 Cubic splines 75

7 Trigonometric Interpolation 79
7.1 The Trigonometric Polynomial 79
7.2 Discrete Fourier Transform (DFT) 80
7.3 Efficient Computation of DFT by FFT 84
7.4 Practical Considerations 86

8 Numerical Integration (Quadrature) 89
8.1 Rectangular Method 90
8.2 Trapezoidal Method 91
8.3 Simpson's Rule 94
8.4 Romberg's Method 96
8.5 Gauss Quadrature 99

9 Numerical Differentiation 105
9.1 Finite Differences 106
9.2 Romberg Scheme 110

10 Initial Value Problems of Ordinary Differential Equations 113
10.1 Numerical Problem Formulation 114
10.2 Explicit One-Step Methods 115
10.2.1 Explicit single-stage one-step methods: the explicit Euler method 117
10.2.2 Higher-order single-stage one-step methods 119
10.2.3 Heun's multi-stage one-step method 119
10.2.4 Higher-order multi-stage one-step methods: Runge-Kutta methods 121
10.3 Dynamic Step-Size Adaptation in One-Step Methods 122
10.3.1 Richardson extrapolation 124
10.3.2 Embedded Runge-Kutta methods 126
10.3.3 Practical implementation 127
10.4 Implicit Methods 128
10.4.1 Implicit single-stage methods: Trapezoidal method 129
10.4.2 Implicit multi-stage methods: implicit Runge-Kutta . . . 130
10.5 Multistep Methods 132
10.5.1 An explicit multistep method: Adams-Bashforth 133
10.5.2 An implicit multistep method: Adams-Moulton 135
10.5.3 Predictor-Corrector method 136
10.6 Systems of ODEs and Higher-Order ODEs 137
10.6.1 Multidimensional systems of ODEs 137
10.6.2 ODEs of higher order 138
10.7 Numerical Stability 139
10.7.1 Stability of the explicit Euler method for real $\boldsymbol{\lambda}$ 140
10.7.2 Stability of Heun's method for real $\boldsymbol{\lambda}$ 142
10.7.3 Stability of explicit Euler for complex $\boldsymbol{\lambda}$ 142
10.7.4 Stability of Heun's method for complex $\boldsymbol{\lambda}$ 143
10.7.5 Stability of the implicit trapezoidal method 145
10.8 Stiff Initial Value Problems 147
10.9 The Lax Equivalence Theorem 149

11 Numerical Solution of Partial Differential Equations 151
11.1 Parabolic Problems: The Method of Lines 155
11.1.1 The explicit Richardson method 157
11.1.2 The implicit Crank-Nicolson method 158
11.2 Elliptic Problems: Stencil Methods 159
11.3 Hyperbolic Problems: The Method of Characteristics 164
11.3.1 Numerical approximation 167
11.3.2 Numerical domain of dependence 168
11.3.3 Courant-Friedrichs-Lewy condition 169

Bibliography 171

Foreword

These lecture notes were created for the course "Basic Numerical Methods", taught as part of the mandatory electives module "CMS-COR-NUM" in the Masters Program "Computational Modeling and Simulation" at TU Dresden, Germany. The notes are based on handwritten notes by Prof. Sbalzarini, which in turn are adapted from the lecture "Numerische Mathematik" taught by Prof. K. Nipp at ETH Zürich during winter 1999/2000, which in turn was based on the German-language textbook "Numerische Mathematik" by H. R. Schwarz, Teubner Publishing Company, 4. Edition, 1997. Prof. Sbalzarini's handwritten notes have been translated to English and typeset in LaTeX by Harish Jain as a paid student teaching assistantship during his first term of studies in the Masters Program "Computational Modeling and Simulation" at TU Dresden, Germany, and subsequently edited, extended, and proofread by Prof. Sbalzarini.

Notation

We follow the standard mathematical notation. Vectors are always understood as column vectors and are typeset with an over-arrow: $\vec{v}$. Matrices are typeset in bold as $\mathbf{A}$ and contain elements $\mathbf{A} = (a_{ij})$ where the first index (here: i) runs over rows from top (= row 1) to bottom and the second index (here: j) runs over columns from left (= column 1) to right. Scalar (inner) products between two vectors are written as: $\langle \vec{p}, \vec{q} \rangle = \vec{p}^{\mathsf{T}} \vec{q} = \vec{p} \cdot \vec{q}$. Single vertical bars $|\cdot|$ denote absolute values or vector norms, whereas double vertical bars $\|\cdot\|$ mean matrix norms or function norms. Open intervals are given by their boundaries a, b as (a, b), whereas closed intervals are $[a, b]$.

> Sections in gray-background boxes are parenthetic remarks that provide additional background or additional examples that are not part of the primary lecture contents.

Particularly important key equations are boxed. Historic context for the learning material is provided by margin notes with portraits and minimal biographies. The reader will notice the particularly poor diversity and gender balance amongst the portrayed. It is the more important that many female students and students from underrepresented backgrounds study the material and change this in the future by providing the next innovations in numerical methods.

Chapter 1

Foundations of Numerical Computation

The objective of a numerical method is to solve a continuous[1] mathematical problem with the help of a computer. To this end, a numerical algorithm is created, consisting of a finite sequence of well-defined calculation steps. The result of such algorithmic computation is always *approximate*, i.e., the numerical algorithm computes an approximation to the true solution of the original mathematical problem. The field of numerical mathematics hence contains two areas of study: A) Theory of Algorithms, B) Approximation Theory. The area (A) studies how mathematical solutions can be formulated as numerical algorithms, and what complexity (upper or lower bounds) these algorithms must have. Area (B) studies how computers can approximate results over the set of real numbers $\mathbb{R}$ and how well a given algorithm is able to approximate the exact solution of a problem.

Example 1.1. *Frequently considered mathematical problems and their formulation as numerical problems:*

- *Linear system of equations* $\mathbf{A}\vec{x} = \vec{b}$ $\longrightarrow$ *find approximate solution* $\hat{\vec{x}} \approx \vec{x}$
- *Ordinary differential equation* $\frac{\mathrm{d}\vec{x}}{\mathrm{d}t} = f(\vec{x}, t)$, $x(0) = x_0$ $\longrightarrow$ *find approximate sequence* $\hat{\vec{x}}(t_k) \approx \vec{x}(t = t_k)$ *for time points* t_k, $k = 0, 1, 2, \ldots$.

1.1 Floating-Point Number Representation

Only finitely many numbers can be represented on a digital computer. This is true for both integer numbers, of which a digital computer can only represent a finite range, and for fractional numbers. Arbitrary real numbers (e.g., irrational

[1] Discrete mathematical problems can usually directly be implemented on a computer and require no numerical methods, as computers operate discretely.

numbers) cannot be numerically represented on a computer at all. Therefore, all known computer architectures approximate real numbers by floating-point numbers:

$$\hat{x} = \sigma \cdot M \cdot B^E \tag{1.1}$$

with:

- σ: the sign of the number (either + or −),
- B: the basis of number representation (an integer constant > 1 ,e.g., $B = 2$ for binary number representation, $B = 10$ for decimal representation),
- M: the mantissa, and
- E: the exponent.

In the following, we denote by $\hat{x}$ the floating-point approximation of a real number x. The mantissa is composed of l digits according to the convention:

$$M = \sum_{j=1}^{l} m_{-j} B^{-j} \tag{1.2}$$

with all digits $0 \leq m_{-j} \leq B - 1$. By convention, the mantissa is always *normalized* so that for any number $\hat{x} \neq 0$, the first digit of the mantissa is non-zero, i.e., $m_{-1} \neq 0$ if $\hat{x} \neq 0$ (see box below for why this is so).
The exponent is represented by the convention:

$$E = \tau \sum_{j=1}^{k} e_j B^{k-j} \tag{1.3}$$

with sign τ, digits $0 \leq e_j \leq B - 1$, and total length (i.e., number of digits) k. This defines the representation of floating-point machine numbers.
Often, machine numbers are given by their digits, as:

$$\hat{x} = \sigma\, m_{-1}\, m_{-2} \ldots m_{-l}\ (\tau e_1 \ldots e_k). \tag{1.4}$$

The digits of the mantissa are indexed with negative indices (from -1 to $-l$) by convention, whereas the digits of the exponent are indexed with positive indices. This on the one hand makes it easier to recognize immediately whether a given digit is part of the mantissa or the exponent, and it also reminds that the digits of the mantissa represent the number in units of B^{-j}, see Eq. 1.2.
This floating-point representation of machine numbers is standard today. It was originally proposed by Leonardo Torres y Quevedo in 1914 for decimal base $B = 10$, while Konrad Zuse was the first to use binary floating-point numbers with $B = 2$ in 1938. While floating-point representations are nowadays standard in numerical computing, this was not clear for a long time. Still in 1951, John von Neumann argued strongly that fixed-point arithmetics should be preferred. Also standard today in digital computers is the binary representation with base

$B = 2$. But this is only standardized since 1985 as IEEE Standard 754. Most early computers used decimal arithmetics, including the ENIAC, IBM NORC, and IBM 1400 series of machines, and the famous IBM S/360 architecture introduced hexadecimal floating-point arithmetics with base $B = 16$, which is still optionally available in today's IBM Mainframes of the z/Architecture line of products.

Image: researchgate
Leonardo Torres y Quevedo
∗ 28 December 1852
Santa Cruz de Iguña, Spain
† 18 December 1936
Madrid, Spain

Image: Uni Hamburg
Konrad Zuse
∗ 22 June 1910
Berlin, German Empire
† 18 December 1995
Hünfeld, Germany

Example 1.2. *Let $B = 10$, $l = 5$, and $k = 2$ (i.e., maximum exponent is 99). Then, the following are numbers with their associated floating-point representation:*

- $x = -14.315 \quad \longrightarrow \hat{x} = -14315(+02)$,
- $x = 0.00937 \quad \longrightarrow \hat{x} = +93700(-02)$.

Definition 1.1 (Machine numbers)**.** *We define $\mathbb{M}(B, l, k)$ the set of* machine numbers *with base B, mantissa length l, and exponent length k.*

We then observe that:

- $\mathbb{M}$ is finite.
- $\mathbb{M}$ possesses a largest element $\hat{x}_{\max} = \max_{\mathbb{M}} |\hat{x}|$ and a smallest element $\hat{x}_{\min} = \min_{\mathbb{M}} |\hat{x}|$ corresponding to the largest and smallest absolute values of numbers that can be represented. Numbers with absolute value larger than $\hat{x}_{\max}$ cause an arithmetic *overflow*, numbers with absolute value smaller than $\hat{x}_{\min}$ cause an arithmetic *underflow* (see Fig. 1.1).
- The numbers in $\mathbb{M}$ are not equidistant on the real axis (see Fig. 1.2).

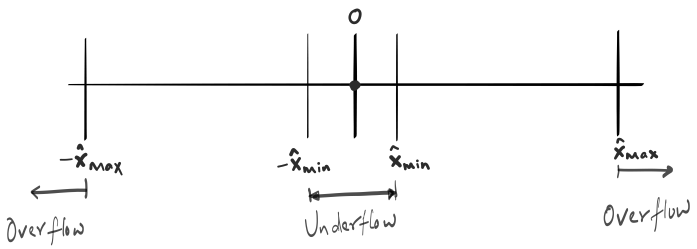


Figure 1.1: Overflow and underflow of machine numbers. 0 is part of the set of machine number, but the interval $(-\hat{x}_{\min}, +\hat{x}_{\min}) \setminus 0$ cannot be represented; it is an underflow. The intervals $(-\infty, -\hat{x}_{\max})$ and $(+\hat{x}_{\max}, \infty)$ cannot be represented either; they are an overflow.

Example 1.3. *Assume the machine numbers $\mathbb{M}(2, 3, 2)$. Then, the following is an exhaustive list of all mantissas and exponents that can be represented (remember the mantissa has to be normalized):*

Mantissa	*Exponent*

$$\begin{array}{cccl} 0 & 0 & 0 & = 0/0 \\ 1 & 0 & 0 & = 1/2 \\ 1 & 0 & 1 & = 5/8 \\ 1 & 1 & 0 & = 3/4 \\ 1 & 1 & 1 & = 7/8 \end{array} \qquad \begin{array}{ccl} \pm 0 & 0 & = \pm 0 \\ \pm 0 & 1 & = \pm 1 \\ \pm 1 & 0 & = \pm 2 \\ \pm 1 & 1 & = \pm 3 \end{array}$$

Therefore, this set of machine numbers has the following largest and smallest members by absolute value: $\hat{x}_{max} = \frac{7}{8} \cdot 2^3 = 7$, $\hat{x}_{min} = \frac{1}{2} \cdot 2^{-3} = \frac{1}{16}$. *Furthermore, the machine numbers in the range between* $\frac{1}{16}$ *and* 7 *are not equidistant on the real axis (see Fig. 1.2).*

Figure 1.2: Distribution of machine numbers in $\mathbb{M}(2, 3, 2)$

1.2 Integer Number Representation

Representing (signed) integers is straightforward and follows the representation of the exponent of a floating-point number, as:

$$i = \sigma N \quad \text{with} \quad N = \sum_{j=1}^{r} n_j B^{r-j} \tag{1.5}$$

with digits $0 \leq n_j \leq B - 1$. On a machine with mantissa length l and exponent length k, a total of $r = l + k + 1$ (i.e., all digits of the mantissa and the exponent, plus the sign of the exponent) digits are available for representing integers. The set of machine integers is denoted $\mathbb{I}(B, r) = \mathbb{I}(B, l + k + 1)$. Integers within this set are represented exactly.

1.3 Rounding

Since the set of machine numbers is finite, any given real number must be mapped into this set before it can be represented and used in a computer. This operation is a *reduction map* called *rounding*:

$$\begin{aligned} \rho : \mathbb{R} &\to \mathbb{M} \\ x &\mapsto \hat{x} = \rho(x). \end{aligned} \tag{1.6}$$

Definition 1.2 (Rounding)**.** *Given* $x \in \mathbb{R}$, $\hat{x} = \rho(x)$ *is the number in* $\mathbb{M}$ *that is closest to* x, *i.e.,* $\rho(x) \in \mathbb{M}$ *such that* $|x - \rho(x)|$ *is minimal. The unique operation* ρ *fulfilling this is called* rounding.

Definition 1.3 (Machine accuracy, machine epsilon)**.** *The relative rounding error is bounded by:*

$$\left|\frac{x-\rho(x)}{x}\right| < \frac{\frac{1}{2}B^{E-l}}{B^{E-1}} = \frac{1}{2}B^{1-l} =: \varepsilon. \tag{1.7}$$

The number ε is called machine epsilon *or* machine accuracy*.*

The numerator of this bound can be understood as follows: The rounding error comes from neglecting the digits of the mantissa that are not explicitly represented any more, i.e., $m_{-(l+1)}B^{-(l+1)}$, $m_{-(l+2)}B^{-(l+2)}$, $m_{-(l+3)}B^{-(l+3)}\ldots$. By definition, we have $0 \leq m_{-(l+i)} \leq B-1$, for all $i = 1, 2, 3, \ldots$, so the neglected part of the mantissa is $< B^{-l}$. Since rounding is either up or down, the maximum error in the last represented digit of the mantissa is $\frac{1}{2}B^{-l}$.
The denominator is $|x| \geq |m_{-1}||B^{-1}|$, i.e., the number to be represented is certainly larger than or equal to the first digit of the mantissa alone. Because of the normalization convention of the mantissa, we know that $m_{-1} \geq 1$ and hence:

$$\frac{1}{|x|} \leq |B| = \frac{1}{|B^{-1}|}.$$

Putting these together, and inserting again the exponents, we find for the complete relative error bound:

$$\left|\frac{x-\rho(x)}{x}\right| < \frac{\frac{1}{2}B^{-l}B^{E}}{|B^{-1}|B^{E}} = \frac{1}{2}B^{1-l}, \tag{1.8}$$

as stated in the definition.
As a corollary, we have that $\forall\alpha \in \mathbb{R}$ with $0 \leq \alpha \leq \varepsilon$, we have $\rho(1+\alpha) = 1$. Machine epsilon can hence be understood (and empirically determined!) as the largest positive number that can be added to 1 so that the result remains 1 in the machine's floating-point number representation.

Why is the mantissa normalized?

By convention, as stated above, the mantissa is normalized such that for any $\hat{x} \neq 0$, $m_{-1} \neq 0$. At first sight, the reasons for this may not be clear, as it leads to a smaller range of representable numbers $[\hat{x}_{\min}, \hat{x}_{\max}]$ than what would be possible with an un-normalized mantissa. There are, however, two good reasons for normalization:

1. For binary numbers, where $B = 2$, this allows "gaining" an additional bit. Whenever a number is non-zero, we know that the first digit of the mantissa is a 1. Therefore, the first digit of the mantissa does not need to be stored explicitly. Since one then only has to store $m_{-2}\ldots m_{-l}$, this effectively extends the mantissa length by one bit (see also the IEEE 754 standard below).

2. The above upper bound on the rounding error would not be possible for an un-normalized mantissa, where machine epsilon could potentially be much larger. Normalizing the mantissa leads to a theoretically bounded machine epsilon.

Therefore, by normalizing the mantissa, one gains numerical guarantees (bounded epsilon) and storage (an additional bit), at the expense of a smaller representable range of numbers.

Example 1.4. *Following are the characteristic figures of the IEEE 32-bit floating-point number standard (IEEE 754):*

B	l	k	$\hat{x}_{min}$	$\hat{x}_{max}$	E_{min}	E_{max}	ε
2	24	8	$1.175 \cdot 10^{-38}$	$3.402 \cdot 10^{38}$	-126	127	$\frac{1}{2} \cdot 2^{-23} \approx 6 \cdot 10^{-8}$

A few comments, as there are some additional tweaks in the standard that we have not discussed above:

- *The IEEE 754 binary32 standard uses unsigned exponents. Above, we have used an explicit sign bit in the exponent, followed by k digits. In the IEEE 754 standard, $k = 8$ with no sign bit. Instead, the exponents are represented in a shifted fashion with an offset of 127. To convert the binary number stored in the exponent to the actual exponent, one has to subtract 127 from it.*
- *The smallest exponent allowed is 1, the largest is 254. The exponents 0 and 255 are used to represent special conditions: An exponent of 0 indicates and underflow ("subnormal number") and an exponent of 255 an overflow ("NaN/Inf"). Therefore, we have $E_{min} = -126$ and $E_{max} = 127$.*
- *The IEEE 754 standard achieves an effective $l = 24$ by explicitly storing only 23 bits of the mantissa by using the normalization trick explained in the box above. Therefore, we gain an additional bit without having to store it.*

1.4 Pseudo Arithmetic

Since $\mathbb{M}$ is finite, it is not closed with respect to the elementary arithmetic operations $+, -, \times, /$, while the set of real numbers $\mathbb{R}$ **is** closed.

Example 1.5. *Assume $\mathbb{M}(10, 3, 2)$, then the result $9.22 + 1.22 = 10.44 \notin \mathbb{M}$. Instead, it will be rounded to: $+922(+01) + 122(+01) = +104(+02) \in \mathbb{M}$.*

In order to close the set $\mathbb{M}$, the arithmetic in $\mathbb{R}$ is replaced with the following pseudo-arithmetic in $\mathbb{M}$:

$$\begin{aligned} \hat{x} \oplus \hat{y} &= \rho(\hat{x} + \hat{y}) \\ \hat{x} \ominus \hat{y} &= \rho(\hat{x} - \hat{y}) \\ \hat{x} \otimes \hat{y} &= \rho(\hat{x} \times \hat{y}) \\ \hat{x} \oslash \hat{y} &= \rho(\hat{x} / \hat{y}) \end{aligned} \tag{1.9}$$

with the exception of overflows and underflows, when an error condition is returned instead of the result.
The compute rules in pseudo arithmetics are changed. In, $\mathbb{R}$, addition and multiplication are associative, distributive, and commutative. In $\mathbb{M}$, they are only commutative, but not associative and not distributive.

Example 1.6. *Associativity of addition in* $\mathbb{M}(10, 3, 1)$. *Let* $x = 9.22$, $y = 1.22$, $z = 0.242$. *Their exact sum is* $x + y + z = 10.682$. *In pseudo arithmetics:*

$$\hat{x} \oplus \hat{y} = \rho(10.44) = 10.4$$

$$(\hat{x} \oplus \hat{y}) \oplus \hat{z} = \rho(10.4 + 0.242 = 10.642) = 10.6.$$

However, if we add them in a different order:

$$\hat{y} \oplus \hat{z} = \rho(1.462) = 1.46$$

$$\hat{x} \oplus (\hat{y} \oplus \hat{z}) = \rho(9.22 + 1.46 = 10.68) = 10.7.$$

The second result is closer to the truth than the first.

The observation from the above example is in fact general. It is always best in floating-point pseudo-arithmetic to sum numbers from small to large. This is a simple algorithm to improve the numerical accuracy of computing long sums. Kahan summation, Neumaier summation, the Shewchuk algorithm, and Bresenham's line algorithm are fancier algorithms to reduce the accumulation or rounding errors in floating-point addition. They are implemented by default in numerical computing languages, such as the BLAS library, Python, and Julia.

1.5 Error Propagation

Rounding errors from pseudo arithmetics propagate through downstream operations in an algorithm and may be amplified during further processing. The goal of error propagation analysis is to see how and to which extent such error amplification may happen.

Definition 1.4 (Absolute and Relative Error)**.** *Let* $\hat{x} = \rho(x)$ *be the floating-point approximation to a true value* x. *Then, we have:*

1. Absolute error*:* $\Delta x = \hat{x} - x$,
2. Relative error*:* $\delta x = \frac{\Delta x}{x}$ *for* $x \neq 0$.

Let's see how absolute errors are propagated by the four basic arithmetic operations:

$$\Delta(x \pm y) = (\hat{x} \pm \hat{y}) - (x \pm y) = \Delta x \pm \Delta y \tag{1.10}$$

$$\Delta(x \times y) = \hat{x}\hat{y} - xy = (x + \Delta x)(y + \Delta y) - xy \tag{1.11}$$

$$= x\Delta y + y\Delta x + \Delta x\Delta y \approx x\Delta y + y\Delta x = xy(\delta x + \delta y) \tag{1.12}$$

$$\Delta(x/y) = \frac{\hat{x}}{\hat{y}} - \frac{x}{y} = \frac{x + \Delta x}{y + \Delta y} - \frac{x}{y} \approx \ldots \approx \frac{x}{y}\delta x - \frac{x}{y}\delta y = \frac{x}{y}(\delta x - \delta y) \tag{1.13}$$

where the approximations are to first order in the absolute error.
Next, consider how relative errors are propagated by the four basic arithmetic operations:

$$\delta(x \pm y) = \frac{\Delta(x \pm y)}{x \pm y} = \frac{x}{x \pm y}\frac{\Delta x}{x} \pm \frac{y}{x \pm y}\frac{\Delta y}{y} = \frac{x}{x \pm y}\delta x \pm \frac{y}{x \pm y}\delta y \quad (1.14)$$

$$\delta(x \times y) = \frac{\Delta(xy)}{xy} \approx \delta x + \delta y \quad (1.15)$$

$$\delta(x/y) = \frac{\Delta(x/y)}{x/y} \approx \delta x - \delta y \quad (1.16)$$

where we have used Eqs. 1.10-1.13 in the derivations.
To summarize, absolute errors get added (subtracted) during addition (subtraction) and relative errors get added (subtracted) during multiplication (division).

1.5.1 Extinction

If $|x \pm y| \ll |x|$ or $|x \pm y| \ll |y|$, it is possible that $\delta(x \pm y) \gg |\delta x| + |\delta y|$, as the prefactors to the relative error components in Eq. 1.14 are $\gg 1$. This phenomenon of potentially massive error amplification is called *numerical extinction.*

Example 1.7. *To illustrate extinction, consider the following two examples:*

1. *Assume* $\mathbb{M}(10, 4, \cdot)$ *and two numbers* $x = \pi$ *and* $y = \frac{22}{7}$. *Then:*

$$\hat{x} \ominus \hat{y} = 3.142 - 3.143 = -1.000 \cdot 10^{-3}.$$

Only one correct digit of the result remains. All other digits are lost due to extinction.

2. *Assume* $\mathbb{M}(10, 3, \cdot)$ *and two numbers* $x = 701$ *and* $y = 700$. *Then:*

$$x^2 - y^2 = 1401$$

$$\rho(x^2) \ominus \rho(y^2) = 4.91 \cdot 10^5 - 4.90 \cdot 10^5 = 1.00 \cdot 10^3.$$

Therefore, the absolute error is $\Delta = 401$ *and the relative error* $\delta = 0.3$, *which is huge. The result of the calculation is wrong by almost one third due to extinciton.*

A better way to compute the same function is to use the fact that $x^2 - y^2 = (x + y)(x - y)$ *and compute:*

$$\hat{x} \oplus \hat{y} = 1.40 \cdot 10^3$$

$$\hat{x} \ominus \hat{y} = 1.00,$$

leading to a result of 1400 and hence $\Delta = 1$ *and* $\delta = 10^{-3}$, *which is a good approximation given the short mantissa used.*

In general, one should always avoid subtracting two almost equally large numbers. What "almost equally large" means depends on the absolute magnitude of the numbers. Since machine numbers are not evenly distributed (see Fig.1.2), closeness of numbers has to be measured by how many machine numbers lie between them. Note, however, that subtraction is not the only way for extinction to occur, and more thorough error propagation analysis is usually required when developing numerical algorithms. If extinction cannot be avoided, a higher numerical precision (i.e., larger bit width) should be used locally. This is the case whenever the function to be computed is badly conditioned, as captured by the concept of condition numbers.

1.5.2 Condition number of a function

The effect of error amplification in a computation is formalized in the concept of *condition numbers*. Consider a function H with exact result $y = H(x)$. Assume that the function can be evaluated exactly, but that the input argument x contains rounding errors. The absolute error in the function return value then is:

$$\Delta H(x) = H(\hat{x}) - H(x) = H(x + \Delta x) - H(x) \approx H'(x)\Delta x. \tag{1.17}$$

Here, we have used the differential quotient $H'(x) = \frac{\mathrm{d}H}{\mathrm{d}x} \approx \frac{H(x+\Delta x)-H(x)}{\Delta x}$ to approximate the derivative. Then, the relative error is:

$$\delta H(x) = \frac{\Delta H(x)}{H(x)} \approx \frac{xH'(x)\Delta x}{xH(x)}. \tag{1.18}$$

Therefore:

$$|\delta H(x)| \approx \underbrace{\left|\frac{xH'(x)}{H(x)}\right|}_{=:\kappa_H(x)} \cdot |\delta x|. \tag{1.19}$$

The scalar pre-factor $\kappa_H(x)$, for $x \neq 0$, is called the *condition number* of the function H. It is the amplification factor of the relative error in x through H. Functions with large condition numbers are called "badly conditioned" or "ill-conditioned". Functions with small condition number are called "well conditioned".

Ill-conditioned and ill-posed problems

There are two kinds of mathematical problems that are hard or impossible to solve numerically: ill-posed problems and ill-conditioned problems.
Ill-conditioned problems, as introduced above, are characterized by a large condition number and hence large amplification of the numerical rounding errors. For an ill-conditioned problem, there exists no accurate numerical algorithm. Rounding error amplification and extinction are necessarily a

problem. Only larger bit widths or different kinds of computer arithmetics (e.g., arbitrary-precision arithmetics) may help.
Ill-posed problems cannot be solved numerically at all. A problem is called "ill-posed" if no solution exists, the solution is not unique, or the solution is unstable. In all of these cases, the problem cannot be computed numerically. This is obviously the case if no solution exists. But also if the solution is not unique, it cannot be computed numerically, as you would never know which one is computed. Finally, solutions are unstable if a small change to the problem parameters leads to an unbounded change in the solution. Small changes in problem parameters (e.g., initial conditions) can always occur from rounding errors. If this has the potential to completely change the solution, reliable numerical computation is also not possible.
Problems for which a solution exists, is unique, and is stable are called *well posed* or *well formed*. They can be approximated numerically. If in addition the problem is also *well conditioned*, then this approximation can be made accurate (i.e., an accurate numerical approximation algorithm can exist).

Example 1.8. *Consider the quadratic equation $x^2 - 2ax + 1 = 0$ for $a > 1$. By the standard formula, it possesses the following two roots:*

$$x_1 = a + \sqrt{a^2 - 1},$$

$$x_2 = a - \sqrt{a^2 - 1}.$$

Now, define the algorithm that takes as an input the value of a and computes as an output x_2, the smaller of the two roots. The function of the algorithm then is:

$$x_2 = H(a) = a - \sqrt{a^2 - 1}$$

and its condition number is defined as:

$$\kappa_H(a) = \left| \frac{aH'(a)}{H(a)} \right|.$$

We find:

$$H'(a) = 1 - \frac{2a}{2\sqrt{a^2 - 1}}$$

and therefore

$$\frac{aH'(a)}{H(a)} = \frac{a - \frac{a^2}{\sqrt{a^2-1}}}{a - \sqrt{a^2 - 1}} = \frac{-a\left(\frac{a}{\sqrt{a^2-1}} - 1\right)}{\sqrt{a^2 - 1}\left(\frac{a}{\sqrt{a^2-1}} - 1\right)} = \frac{-a}{\sqrt{a^2 - 1}}.$$

Thus:

$$\kappa_H(a) = \left| \frac{-a}{\sqrt{a^2 - 1}} \right|.$$

We observe two cases:

i) for $a \gg 1$, we have $\kappa_H \approx 1$ and the problem is well conditioned. But in this case, $H(a) = a - \sqrt{a^2-1}$ is a bad algorithm, because $\sqrt{a^2-1} \approx a$, which leads to extinction (subtraction of two similarly large machine numbers). However, a well-conditioned problem possesses a good algorithm, which just needs to be found. In the present case, we can use the formula of Vieta to find

$$H(a) = \frac{1}{a + \sqrt{a^2-1}},$$

which has a small condition number for large a and does not suffer from extinction. This would thus be a good way of numerically evaluating the root.

ii) for $a \approx 1$, we have $\kappa_H \gg 1$ and the problem is ill-conditioned. In this case, there exists no good algorithm. Regardless of how we compute the result, the error is always going to be amplified by a large condition number. We could only use higher-precision arithmetics (i.e., more bits).

1.5.3 Condition number of a matrix

An important special case are linear functions, i.e., H is linear and can thus be written as a matrix multiplication. This allows us to define the concept of a condition number for matrices, which will be important for determining the accuracy with which a linear system of equations $\mathbf{A}\vec{x} = \vec{b}$ with regular $\mathbf{A} \in \mathbb{R}^{n\times n}$, $b \neq 0$, can be solved for $\vec{x} = \mathbf{A}^{-1}\vec{b} \neq 0$.

Assume that b has rounding errors, i.e., that instead of $\vec{b}$ we have $\hat{\vec{b}}$ with small magnitude of $\Delta\vec{b} = \hat{\vec{b}} - \vec{b}$. Further assume that we can solve $\mathbf{A}\hat{\vec{x}} = \hat{\vec{b}}$ exactly and that the entries of the matrix $\mathbf{A}$ are free of rounding errors. What is then the connection between $\delta\vec{b}$ and $\delta\vec{x} = \frac{1}{|\vec{x}|}(\hat{\vec{x}} - \vec{x})$? We find:

$$\Delta\vec{x} = \hat{\vec{x}} - \vec{x} = \mathbf{A}^{-1}\hat{\vec{b}} - \mathbf{A}^{-1}\vec{b} = \mathbf{A}^{-1}(\hat{\vec{b}} - \vec{b}) = \mathbf{A}^{-1}\Delta\vec{b} \tag{1.20}$$

and therefore:

$$|\delta\vec{x}| = \frac{|\mathbf{A}^{-1}\Delta\vec{b}|}{|\vec{x}|}\frac{|\vec{b}||\Delta\vec{b}|}{|\vec{b}||\Delta\vec{b}|} = \frac{|\overbrace{\mathbf{A}\vec{x}}^{\vec{b}}|}{|\vec{x}|}\frac{|\mathbf{A}^{-1}\Delta\vec{b}|}{|\Delta\vec{b}|}|\delta\vec{b}|. \tag{1.21}$$

Due to the triangular inequality, we have:

$$\frac{|\mathbf{A}\vec{x}|}{|\vec{x}|} \leq \|\mathbf{A}\|$$

$$\frac{|\mathbf{A}^{-1}\Delta\vec{b}|}{|\Delta\vec{b}|} \leq \left\|\mathbf{A}^{-1}\right\|$$

and can therefore bound

$$|\delta\vec{x}| \leq \|\mathbf{A}\|\left\|\mathbf{A}^{-1}\right\| \cdot |\delta\vec{b}|.$$

The maximum amplification factor of the relative error therefore is:

$$\kappa_{\mathbf{A}} := \|\mathbf{A}\| \left\|\mathbf{A}^{-1}\right\|, \tag{1.22}$$

which is called the *condition number of the matrix* $\mathbf{A}$. The smallest condition number any matrix can have is 1, because:

$$1 = \|\mathbf{1}\| = \left\|\mathbf{A}\mathbf{A}^{-1}\right\| \leq \|\mathbf{A}\| \left\|\mathbf{A}^{-1}\right\| = \kappa_{\mathbf{A}}.$$

For the case that also the entries of the matrix $\mathbf{A}$ contain rounding errors, the linear system of equations becomes $\hat{\mathbf{A}}\hat{\vec{x}} = \hat{\vec{b}}$, and one can show that:

$$|\delta\vec{x}| \leq \frac{\kappa_{\mathbf{A}}}{1 - \kappa_{\mathbf{A}} \|\delta\mathbf{A}\|}(\|\delta\mathbf{A}\| + |\delta\vec{b}|) \tag{1.23}$$

with the above definition of the matrix condition number, $\kappa_{\mathbf{A}} \|\delta\mathbf{A}\| < 1$, and the definition of the relative error $\delta\mathbf{A}$ understood element-wise.
Obviously, the value of the condition number of a matrix depends on the matrix norm one chooses. A frequent choice is the 2-norm, as revisited in the box below. Readers familiar with the concept can safely skip the box.

The 2-norm $\|\mathbf{A}\|_2$ of a real $n \times n$ matrix $\mathbf{A}$

An induced matrix norm is defined from the corresponding vector norm by:

$$\|\mathbf{A}\|_* = \sup_{x \neq 0} \left\{ \frac{|\mathbf{A}\vec{x}|_*}{|\vec{x}|_*} \right\}. \tag{1.24}$$

So, specifically, the 2-norm for vectors induces the 2-norm of a real matrix as:

$$\|\mathbf{A}\|_2 = \sup_{x \neq 0} \left\{ \frac{|\mathbf{A}\vec{x}|_2}{|\vec{x}|_2} \right\} = \sup_{|\vec{x}|_2 = 1} \{|\mathbf{A}\vec{x}|_2\}. \tag{1.25}$$

It can thus be understood as the Euclidean length of the longest vector one can obtain by mapping a unit vector through $\mathbf{A}$. For any real $n \times n$ matrix $\mathbf{A}$, $\mathbf{A}^\mathsf{T}\mathbf{A}$ is symmetric. Therefore, an orthogonal matrix $\mathbf{T}$ exists, such that

$$\mathbf{T}^\mathsf{T}(\mathbf{A}^\mathsf{T}\mathbf{A})\mathbf{T} = \mathbf{D} = \mathrm{diag}(\mu_1, \ldots, \mu_n),$$

where the μ_i are the eigenvalues of $\mathbf{A}^\mathsf{T}\mathbf{A}$.
Using the orthogonal transformation $\vec{x} = \mathbf{T}\vec{y}$, we have:

$$0 \leq \|\mathbf{A}\|_2^2 = \sup_{|\vec{x}|_2 = 1} \left\{|\mathbf{A}\vec{x}|_2^2\right\} = \sup_{|\vec{y}|_2 = 1} \left\{|\mathbf{A}\mathbf{T}\vec{y}|_2^2\right\},$$

where in the last step we have used the fact that for any orthogonal transformation $|\vec{x}|_2 = |\vec{y}|_2$. From the definition of the vector 2-norm $|\vec{x}|_2^2 = \vec{x}^\mathsf{T}\vec{x}$,

we further find:

$$= \sup_{|\vec{y}|_2=1} \left\{ \vec{y}^{\mathsf{T}} \underbrace{\mathbf{T}^{\mathsf{T}}(\mathbf{A}^{\mathsf{T}}\mathbf{A})\mathbf{T}}_{\mathbf{D}} \vec{y} \right\} = \sup_{|\vec{y}|_2=1} \left\{ \sum_{i=1}^{n} \mu_i y_i^2 \right\} = \max_i \{\mu_i\}.$$

Therefore, the 2-norm of a matrix is:

$$\boxed{\|\mathbf{A}\|_2 = \sqrt{\mu_{\max}}\,; \quad \mu_{\max}: \text{ largest eigenvalue of } \mathbf{A}^{\mathsf{T}}\mathbf{A}.} \tag{1.26}$$

Some facts and observations:

- If $\mathbf{A}$ is orthogonal, then $\|\mathbf{A}\|_2 = 1$, because $\mathbf{A}^{\mathsf{T}}\mathbf{A} = \mathbf{1}$.
- If $\mathbf{A}^{\mathsf{T}} = \mathbf{A}$ is symmetric, then $\mathbf{A}^{\mathsf{T}}\mathbf{A} = \mathbf{A}^2$ and therefore $\|\mathbf{A}\|_2 = |\lambda_{\max}|$ where $\lambda_{\max}$ is the largest eigenvalue of $\mathbf{A}$.
- If $\mathbf{A}$ is regular, then $\left\|\mathbf{A}^{-1}\right\|_2 = \frac{1}{\sqrt{\mu_{\min}}}$ where $\mu_{\min}$ is the smallest eigenvalue of $\mathbf{A}^{\mathsf{T}}\mathbf{A}$. Note that $\mu_{\min} > 0$ always because $\mathbf{A}^{\mathsf{T}}\mathbf{A}$ is positive definite.
- If $\mathbf{A}^{\mathsf{T}} = \mathbf{A}$ is regular and symmetric, then $\left\|\mathbf{A}^{-1}\right\|_2 = \frac{1}{|\lambda_{\min}|}$ where $\lambda_{\min}$ is the smallest eigenvalue of $\mathbf{A}$.

In the 2-norm, we have:

$$\kappa_{\mathbf{A}} = \frac{\sqrt{\mu_{\max}}}{\sqrt{\mu_{\min}}} \qquad \mu: \text{ eigenvalues of } \mathbf{A}^{\mathsf{T}}\mathbf{A} \tag{1.27}$$

$$\text{and for symmetric } \mathbf{A}: \kappa_{\mathbf{A}} = \frac{|\lambda_{\max}|}{|\lambda_{\min}|} \qquad \lambda: \text{ eigenvalues of } \mathbf{A}. \tag{1.28}$$

If one wants to measure the error in the 2-norm, this can be used as an alternative to Eq. 1.22 if appropriate for the purpose (e.g., the size of the matrix).

Example 1.9. *We determine the condition numbers of the following matrices in the 2-norm:*

1. $$\mathbf{A} = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

   The eigenvalues of $\mathbf{A}$ *are:* $(1, 1, 4)$ *and therefore* $\kappa_{\mathbf{A}} = 4$*. This is a small condition number and, therefore, solving a linear system of equations with this matrix is a well-conditioned mathematical problem that can be solved accurately using numerical methods.*

2. $$\mathbf{A} = \begin{bmatrix} 168 & 113 \\ 113 & 76 \end{bmatrix}$$

The eigenvalues of $\mathbf{A}$ *are:* $(244.004\ldots, -0.00409829\ldots)$ *and therefore* $\kappa_{\mathbf{A}} \approx 5.95 \cdot 10^4$*, which is a large condition number. Solving a linear system of equations with this matrix will amplify the relative error of the right-hand side up to 60 000 fold, regardless of which algorithm is used to actually perform the computations.*

The following rule of thumb is useful in practice: On a computer with d-digit arithmetics (i.e., floating point numbers are represented to d decimal digits after the dot), the solution of a linear system with condition number $\kappa_{\mathbf{A}} \approx 10^k$ is going to have approximately $d - k$ correct digits.

1.5.4 Condition number of an algorithm

In the ideal case, a function H is encoded in an algorithm f that computes the exact result $f(x)$ for any input x. In reality, however, the input $\hat{x}$ is only approximate as it contains rounding errors, measurement errors, or other input uncertainties. Also, exact algorithms for many mathematical problems do not exist (or are not known or are too computationally expensive), so that we have to use an approximate algorithm $\hat{f}$ for computing the result $\hat{f}(\hat{x})$. We then define the:

$$\begin{aligned} &\text{input error:} \quad \hat{x} - x \\ &\text{total error:} \quad \hat{f}(\hat{x}) - f(x) \quad = \underbrace{\hat{f}(\hat{x}) - f(\hat{x})}_{\text{algorithm approximation error}} + \underbrace{f(\hat{x}) - f(x)}_{\text{propagated input error}} . \end{aligned}$$

Therefore, the total error is the sum of the algorithmic approximation error (i.e., the error from using an approximate algorithm) plus the rounding error propagated by the exact algorithm.

Similar to how the condition number of a function (see Section 1.5.2) tells us whether a good algorithm for evaluating that function exists, the condition number of an algorithm should tell us whether a good numerical approximation algorithm can possibly exist. This can only be the case if the algorithm f itself is stable with respect to input errors. This is quantified by the *sensitivity* of the exact algorithm f:

$$\kappa = \left| \frac{\text{relative change in result}}{\text{relative change in input}} \right| = \left| \frac{(f(\hat{x}) - f(x))/f(x)}{(\hat{x} - x)/x} \right| \approx \left| \frac{x f'(x)}{f(x)} \right|, \quad (1.29)$$

because (relative error) = (absolute error) / (true value). This is the same expression for the condition number as we found in Eq. 1.19 for functions. Algorithms with small κ are well-conditioned, and good approximations $\hat{f}$ exist and can be found. Algorithms with large κ are ill-conditioned. In an ill-conditioned algorithm, a small error in the input leads to a large error in the result and, thus, no good approximations $\hat{f}$ exist.

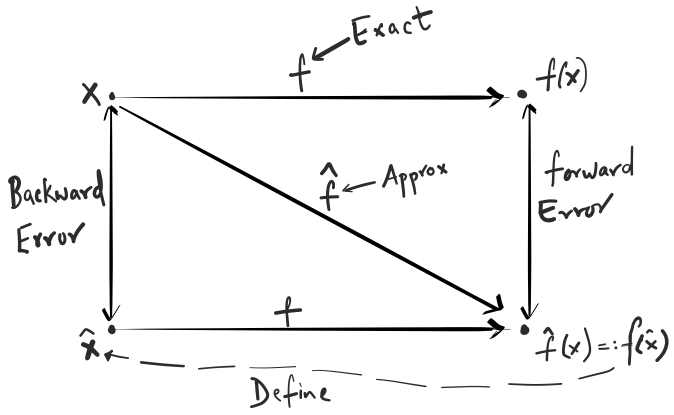


Figure 1.3: Schematic of the concept of backward error analysis

1.6 Backward Error Analysis

We can now define the concept of forward and backward errors. The error between the exact solution $f(x)$ and the approximate solution $\hat{f}(x)$ is called the *forward error* (see Fig. 1.3). Alternatively, we can ask the question: "how much would we have to change the input such that the exact algorithm would yield the same result as the approximate algorithm did for the original input?" That is, what $\hat{x}$ do we have to use such that $f(\hat{x}) = \hat{f}(x)$? The difference between this so-defined $\hat{x}$ (which is **not** a rounding of x) and the original x is the *backward error*.

Example 1.10. *Consider the example of computing the exponential function $F(x) = \mathrm{e}^x$ for an input x. One way of performing the computation is to use the series expansion*

$$f(x) = \mathrm{e}^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

There is no algorithm for computing this exactly, as it would require infinitely many terms to be added. Therefore, we truncate the series at some point, defining an approximate algorithm, e.g.:

$$\hat{f}(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} \approx \mathrm{e}^x.$$

The forward error then is: $\hat{f}(x) - f(x) = -\frac{x^4}{4!} - \frac{5^3}{5!} - \dots$. In order to determine the backward error, we have to ask which $\hat{x}$ we would have to use such that:

$$\mathrm{e}^{\hat{x}} = \hat{f}(x).$$

Solving this for $\hat{x}$, we find:

$$\hat{x} = \log(\hat{f}(x))$$

and therefore the backward error $\hat{x} - x = \log(\hat{f}(x)) - x$.
Evaluating this, e.g., for $x = 1$, *we find:*

$$\begin{aligned} f(x) &= 2.718282\ldots \\ \hat{f}(x) &= 2.6666667\ldots \\ \hat{x} &= \log(2.6666667\ldots) = 0.980829\ldots \end{aligned}$$

In this case, the forward error therefore is: $-0.051615\ldots$ *and the backward error is:* $\hat{x} - x = -0.019171\ldots$.

Chapter 2

Linear System Solvers

One of the fundamental tasks in numerical computation is to solve a linear system of n equations

$$\mathbf{A}\vec{x} = \vec{b} \tag{2.1}$$

with a regular (i.e., invertible) matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and right-hand side $\vec{b} \in \mathbb{R}^n$ for the unknown $\vec{x} \in \mathbb{R}^n$. This can be done using either direct solvers or iterative solvers. The classic direct solver algorithm is Gaussian elimination.

2.1 Gaussian Elimination – LU Decomposition

Image: wikipedia
Johann Carl Friedrich Gauss
∗ 30 April 1777
Brunswick, Principality of Brunswick-Wolfenbüttel
† 23 February 1855
Göttingen, Kingdom of Hanover

Gaussian elimination computes the inverse of a matrix and hence the solution $\vec{x} = \mathbf{A}^{-1}\vec{b}$ of a linear system of equations by subtracting multiples of rows from the other rows in order to make entries zero. It proceeds by selecting a *pivot* element of the matrix and then subtracting multiples of the row in which the pivot lies from the other rows, so as to make all entries in the column of the pivot become zero. The right-hand side is irrelevant for this, as Gaussian elimination only operates on the matrix $\mathbf{A}$.

Example 2.1. *Consider the matrix*

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & -3 \\ 6 & 1 & -10 \\ -2 & -7 & 8 \end{bmatrix}.$$

We perform Gaussian elimination of $\mathbf{A}$*. In the following, the pivots are circled, whereas the multiples of the pivotal row that are* subtracted *from the other rows are written to the left of the matrix (i.e., the first 3 means* $row_2 = row_2 - 3row_1$*, etc.). All entries that thus became zero are replaced by these factors (boxed numbers). The three steps of Gaussian elimination for the above matrix then*

are:

$$\begin{matrix} \\ 3 \\ -1 \end{matrix} \begin{bmatrix} \textcircled{2} & -1 & -3 \\ 6 & 1 & -10 \\ -2 & -7 & 8 \end{bmatrix} \longrightarrow \begin{matrix} \\ \\ -2 \end{matrix} \begin{bmatrix} \textcircled{2} & -1 & -3 \\ \boxed{3} & \textcircled{4} & -1 \\ \boxed{-1} & -8 & 5 \end{bmatrix} \longrightarrow \begin{bmatrix} \textcircled{2} & -1 & -3 \\ \boxed{3} & \textcircled{4} & -1 \\ \boxed{-1} & \boxed{-2} & \textcircled{3} \end{bmatrix}.$$

Splitting the result along the diagonal, we can put all entries below the diagonal into a matrix $\mathbf{L}$ *with unit diagonal and all entries above and including the diagonal into a matrix* $\mathbf{U}$*, thus:*

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ -1 & -2 & 1 \end{bmatrix},$$
$$\mathbf{U} = \begin{bmatrix} 2 & -1 & -3 \\ 0 & 4 & -1 \\ 0 & 0 & 3 \end{bmatrix}.$$

Due to its shape, $\mathbf{L}$ *is called a* lower triangular matrix *and* $\mathbf{U}$ *an* upper triangular matrix*.*

In general, we have:

Theorem 2.1. *Gaussian elimination on* $\mathbf{A} \in \mathbb{R}^{n \times n}$ *without row permutations yields a lower triangular matrix* $\mathbf{L}$ *with ones in the diagonal and an upper triangular matrix* $\mathbf{U}$*.* $\mathbf{L}$ *and* $\mathbf{U}$ *are regular, and we have* $\mathbf{A} = \mathbf{LU}$*. This is called the* LU-decomposition *of the matrix* $\mathbf{A}$*.*

We refer to the literature for proofs of this theorem. Using the LU-decomposition, the linear system can be reformulated as:

$$\begin{aligned} \mathbf{A}\vec{x} &= \vec{b} \\ \mathbf{L}\underbrace{\mathbf{U}\vec{x}}_{\vec{c}} &= \vec{b} \\ &\Rightarrow \begin{cases} \mathbf{L}\vec{c} = \vec{b} \\ \mathbf{U}\vec{x} = \vec{c}, \end{cases} \end{aligned}$$

which decomposes the linear system into two new linear systems. These new systems, however, can be solved straightforwardly due to the triangular shapes of the matrices $\mathbf{L}$ and $\mathbf{U}$. The first system can directly be solved for $\vec{c}$ by forward substitution. Then, the second system is solved for $\vec{x}$ by backward substitution.

Example 2.2. *Consider again the example from above, now with the right-hand side*

$$\vec{b} = \begin{bmatrix} 4 \\ -1 \\ 25 \end{bmatrix}.$$

The $\mathbf{L}$*-equation reads:*

$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ -1 & -2 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 4 \\ -1 \\ 25 \end{bmatrix}.$$

The first row simply says $c_1 = 4$*. Using this knowledge in the second row, we find:* $3c_1 + c_2 = -1 \Rightarrow 12 + c_2 = -1$ *and thus* $c_2 = -13$*. And from the third row we find* $c_3 = 3$*. The* $\mathbf{U}$*-equation then reads:*

$$\begin{bmatrix} 2 & -1 & -3 \\ 0 & 4 & -1 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ -13 \\ 3 \end{bmatrix}.$$

This time we start from the bottom (i.e., backward substitution), where the last row reads: $3x_3 = 3$ *and thus* $x_3 = 1$*. The second row says:* $4x_2 - 1 = -13 \Rightarrow x_2 = -3$*. And finally, the first row tells us:* $2x_1 + 3 - 3 = 4 \Rightarrow x_1 = 2$*. So the solution of this linear system of equations is:*

$$\vec{x} = \begin{bmatrix} 2 \\ -3 \\ 1 \end{bmatrix}.$$

The final algorithm in its simplest form is given in Algorithm 1. The first step of this algorithms requires $O(n^3)$ compute operations, whereas the second and third steps each require $O(n^2)$ operations (up to n operations per row, n rows). If the same system is to be solved for multiple right-hand sides, the computational cost for each additional right-hand side is only $O(n^2)$, since the LU-decomposition only needs to be done once and the factors can be stored and re-used.

Algorithm 1 Direct linear solver without pivoting

1: $(\mathbf{L}, \mathbf{U})$ = LU-decomposition of $\mathbf{A}$ ▷ $O(n^3)$
2: solve $\mathbf{L}\vec{c} = \vec{b}$ for $\vec{c}$ by forward substitution ▷ $O(n^2)$
3: solve $\mathbf{U}\vec{x} = \vec{c}$ for $\vec{x}$ by backward substitution ▷ $O(n^2)$

2.1.1 Pivoting strategies

In the example above, we have simply chosen the first element along the diagonal as the first pivot, followed by the second element, third, and so on. This is not always good, as it may lead to amplification of rounding errors and numerical extinction (see Sec. 1.5), in particular if a pivot is chosen that is small in comparison with the other entries in that column. Such a pivot has to be multiplied by a large number, thus losing the least-significant digits. Pivoting strategies aim to select the pivots in a smarter way, such that the process of Gaussian elimination is numerically more accurate. They are based on re-arranging the

rows of $\mathbf{A}$ before starting Gaussian elimination. Since the ordering of the equations in a linear system of equations in inconsequential, rows may be arbitrarily re-arranged before solving.

Definition 2.1 (Permutation matrix)**.** *A $n \times n$ matrix $\mathbf{P}$ with exactly one 1 in each column and each row, and all other elements equal to 0, is called a* permutation matrix*. $\mathbf{P}$ is regular, as it can be obtained from the identity matrix $\mathbf{1}$ by row permutations. Any matrix $\mathbf{PA}$ has the same row permutations with respect to $\mathbf{A}$ as $\mathbf{P}$ has with respect to $\mathbf{1}$.*

Pivoting strategies amount to first permutating the rows of $\mathbf{A}$ and then performing LU-decomposition as above, successively selecting pivots down the diagonal. Therefore, $\mathbf{LU} = \mathbf{PA}$.

Theorem 2.2. *Gaussian elimination on $\mathbf{PA} \in \mathbb{R}^{n\times n}$, where $\mathbf{P}$ is a permutation matrix, yields a lower triangular matrix $\mathbf{L}$ with ones in the diagonal and an upper triangular matrix $\mathbf{U}$. $\mathbf{L}$ and $\mathbf{U}$ are regular, and we have $\mathbf{PA} = \mathbf{LU}$.*

Due to this theorem, which we do not prove here, we find for the situation with pivoting:

$$\mathbf{A}\vec{x} = \vec{b} \iff \mathbf{PA}\vec{x} = \mathbf{P}\vec{b} \iff \mathbf{LU}\vec{x} = \mathbf{P}\vec{b} \implies \mathbf{L}\vec{c} = \mathbf{P}\vec{b};\ \ \mathbf{U}\vec{x} = \vec{c}$$

and the algorithm is given in Algorithm 2.

Algorithm 2 Direct linear solver with pivoting

1: $(\mathbf{L}, \mathbf{U})$ = LU-decomposition of $\mathbf{PA}$ with permutation matrix $\mathbf{P}$ ▷ $O(n^3)$
2: solve $\mathbf{L}\vec{c} = \mathbf{P}\vec{b}$ for $\vec{c}$ by forward substitution ▷ $O(n^2)$
3: solve $\mathbf{U}\vec{x} = \vec{c}$ for $\vec{x}$ by backward substitution ▷ $O(n^2)$

In practical implementations, it is more memory-efficient to store the permutations in a vector instead of a matrix. Storing the permutation matrix would require $O(n^2)$ memory, whereas storing a permutation vector only requires $O(n)$ memory. A permutation vector is obtained from the vector $\vec{v}^\mathsf{T} = [1, 2, 3, \ldots, n]$ by applying the permutation matrix, thus $\mathbf{P}\vec{v}$. This permutation vector can then directly be used to translate row indices when looping over the rows of $\mathbf{PA}$.

Now that we know how pivot selection can be implemented by permutations, the question of course is *how* to select the pivots, i.e., what permutation $\mathbf{P}$ to use. When calculating the LU-decomposition on a computer with finite-precision floating-point arithmetic (see Chapter 1), pivot selection plays an important role in ensuring numerical accuracy of the result. The pivoting strategy used in Example 2.1 is called **diagonal strategy** and it simply consists in selecting the pivots in order along the diagonal of the matrix from top-left to bottom-right. However, the diagonal strategy without permutation may be arbitrarily bad and lead to significant extinction if a small pivot is chosen. Conversely, the diagonal strategy is good if the matrix is diagonally dominant, i.e., if the absolute value

of the diagonal element in each row is equal to or larger than the sum of the absolute values of all other elements in that row.
This observation inspires another pivoting strategy, the **column maximum strategy**, also known as *partial pivoting*. In this strategy, the rows of the matrix are rearranged such that the row containing the largest entry (by absolute value) in the first column is first, and so on. This avoids extinction because the pivot never needs to be multiplied by a large number. However, it can still lead to inaccurate results if the equations are badly scaled, i.e., if the absolute values of the entries within a row range over a wide spectrum. Also, it may not always be possible to choose the largest remaining pivot, if the corresponding row has already been used before, leading to sub-optimal pivots.
In such situations, the **relative column maximum strategy**, also called *complete pivoting* can help. In this strategy, one first scales each equation (i.e., row of $\mathbf{A}$ and corresponding entry in $\vec{b}$) such that $\max_k |a_{ik}| = 1$, i.e., the largest entry of row i has absolute value 1. After this scaling has been applied to all rows, choose the largest possible pivots, ideally the 1s if possible. The scaling can efficiently be done by multiplying $\mathbf{A}$ from the left with $\mathrm{diag}(1/\max_k |a_{ik}| \,\forall i)$. In some cases, e.g., when the goal is to actually invert the matrix $\mathbf{A}$ rather than solve a linear system of equations, LU-decomposition (Gaussian elimination) can still be done on the original, unscaled matrix. The scaling is then only used to decide the pivots.

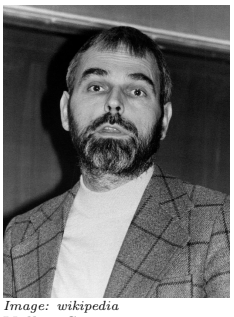


Image: wikipedia
Volker Strassen
∗ 29 April 1936
Düsseldorf, Germany


Image: ethw wiki
Don Coppersmith
∗ c. 1950, USA


Image: ithistory.org
Shmuel Winograd
∗ 4 January 1936
Tel Aviv, Israel
† 2019, New York, USA

2.2 Linear Fixed-Point Iterations

Direct solvers, like LU-decomposition or Gaussian elimination, can be computationally expensive. Gaussian elimination needs $O(n^3)$ compute operations. While more efficient direct solvers are known, such as the Strassen algorithm ($O(n^{2.807355})$) or the Coppersmith-Winograd algorithm ($O(n^{2.3728639})$), direct solvers become inefficient and potentially numerically inaccurate for large linear systems, such as those often encountered when numerically solving partial differential equations. In that case, iterative methods may be preferred. They do not directly compute the solution to machine precision, like direct solvers do, but iteratively improve an approximation to the solution.
Consider again the problem from Eq. 2.1 for a regular, real $n \times n$ matrix $\mathbf{A}$. An iterative solver computes a solution $\vec{x}_*$ that approximates the true and unique solution $\vec{x}$. It does so by starting from an initial guess $\vec{x}_0$ and iterating

$$\vec{x}_{k+1} = f(\vec{x}_k)\,, \qquad k = 0, 1, 2 \ldots \tag{2.2}$$

The simplest class of iteration functions are linear functions. Then:

$$f(\vec{x}) = \mathbf{T}\vec{x} + \vec{c}$$

for some matrix $\mathbf{T} \in \mathbb{R}^{n\times n}$ and vector $\vec{c} \in \mathbb{R}^n$. The resulting iteration is:

$$\vec{x}_{k+1} = \mathbf{T}\vec{x}_k + \vec{c}, \qquad k = 0, 1, 2 \ldots \tag{2.3}$$

If $\{\vec{x}_k\}$ converges, i.e. $\lim_{k\to\infty} \vec{x}_k = \vec{x}_*$, then obviously:

$$\vec{x}_* = \mathbf{T}\vec{x}_* + \vec{c} \tag{2.4}$$

and $\vec{x}_*$ is the uniquely determined *fixed point* of the iteration. Such a fixed point exists if and only if $[\mathbf{1} - \mathbf{T}]$ is regular, because solving Eq. 2.4 for $\vec{x}_*$ yields: $\vec{x}_* = [\mathbf{1} - \mathbf{T}]^{-1}\vec{c}$. In order for this fixed point $\vec{x}_*$ to be the correct solution of the linear system of equations in Eq. 2.1, we have to require:

$$\mathbf{A}\vec{x}_* = \mathbf{A}[\mathbf{1} - \mathbf{T}]^{-1}\vec{c} = \vec{b}. \tag{2.5}$$

Iterations that fulfill this condition are called *consistent*. Consistency implies that a fixed point exists, and that this fixed point is the correct solution of Eq. 2.1, i.e., $\vec{x}_* = \vec{x}$.

2.2.1 Convergence criteria

A natural question to ask at this point is under which conditions the iteration in Eq. 2.3 converges to its fixed point if a fixed point exists. In order to analyze this, consider the absolute error (i.e., distance from the fixed point) at iteration k, $\vec{e}_k \coloneqq \vec{x}_k - \vec{x}_*$. Subtracting 2.4 from 2.3, we find that

$$\vec{e}_{k+1} = \mathbf{T}\vec{e}_k\,, \qquad k = 0, 1, 2 \ldots. \tag{2.6}$$

and therefore, after k iterations, $\vec{e}_k = \mathbf{T}^k\vec{e}_0$.

Theorem 2.3. *Let $[\mathbf{1} - \mathbf{T}]$ be regular. Then, the iteration defined in Eq. 2.3 converges for any start vector $\vec{x}_0$ if $\|\mathbf{T}\| < 1$.*

Proof. $|\vec{e}_k| = |\mathbf{T}^k\vec{e}_0| \leq \left\|\mathbf{T}^k\right\| \cdot |\vec{e}_0| \leq \|\mathbf{T}\|^k \cdot |\vec{e}_0| \implies |\vec{e}_k| \to 0$ as $k \to \infty$ if $\|\mathbf{T}\| < 1$. □

It is evident from the proof that convergence is linear with decay factor $\|\mathbf{T}\|$, i.e., $|\vec{e}_{k+1}| \approx \|\mathbf{T}\| \cdot |\vec{e}_k|$. Moreover, it is clear that requiring this condition for all possible matrix norms is sufficient for convergence, but it is not necessary. It is only necessary that there exists *a* norm for which it holds. Therefore, the question arises, whether there exists a necessary *and* sufficient condition for a certain choice of matrix norm that is easy to check numerically.
For this, we consider the following quantity:

Definition 2.2 (Spectral radius)**.** *Let λ_i, $i = 1, \ldots, n$, be the eigenvalues of a matrix $\mathbf{T} \in \mathbb{R}^{n\times n}$. Then,*

$$\rho(\mathbf{T}) \coloneqq \max_i |\lambda_i|$$

is called the spectral radius *of* $\mathbf{T}$.

The spectral radius provides a lower bound on all matrix norms, as stated in the following theorem.

Theorem 2.4. *For any induced matrix norm $\|\cdot\|$ and any matrix $\mathbf{T} \in \mathbb{R}^{n\times n}$, it is $\rho(\mathbf{T}) \leq \|\mathbf{T}\|$.*

Proof. $\rho(\mathbf{T}) = |\lambda_j|$ with corresponding eigenvector $\vec{x}_j$. Then:

$$\|\mathbf{T}\| = \sup_{x\neq 0} \frac{|\mathbf{T}\vec{x}|}{|\vec{x}|} \geq \frac{|\mathbf{T}\vec{x}_j|}{|\vec{x}_j|} = \frac{|\lambda_j \vec{x}_j|}{|\vec{x}_j|} = |\lambda_j| = \rho(\mathbf{T}),$$

where we have used the definition of an induced matrix norm from Eq. 1.24 and the defining property of eigenvectors. □

The bound provided by the spectral radius is tight in the sense that for any matrix $\mathbf{T} \in \mathbb{R}^{n\times n}$, one can always construct a certain matrix norm whose value is arbitrarily close to the value of the spectral radius. This is stated in the following theorem:

Theorem 2.5. *For every matrix $\mathbf{T} \in \mathbb{R}^{n\times n}$ and every $\epsilon > 0$, there exists an induced matrix norm $\|\cdot\|_\epsilon$ for which*

$$\|\mathbf{T}\|_\epsilon \leq \rho(\mathbf{T}) + \epsilon.$$

The proof of this theorem is more involved and therefore omitted here. But letting $\epsilon \to 0$, we find that there always exists a matrix norm whose value is identical with the value of the spectral radius (i.e., is $\leq$ and at the same time $\geq$).
This leads to the following result, which is what we were looking for:

Theorem 2.6. *If $\rho(\mathbf{T}) < 1$, then there exists an $\epsilon > 0$ and an induced matrix norm $\|\cdot\|_\epsilon$ such that $\|\mathbf{T}\|_\epsilon < 1$.*

Proof. Choosing $\epsilon = (1 - \rho(\mathbf{T}))/2 > 0$ proves the claim due to Theorem 2.5, where the right-hand side then becomes $\frac{1}{2} + \frac{\rho}{2} < 1$ if $\rho < 1$. □

The above theorem states that if the spectral radius of a matrix is less than one, then there always exists a norm whose value is also less than one, and therefore the linear fixed-point iteration converges. If the spectral radius is larger than one, no norm less than one exists, because the spectral radius is a lower bound on all norms, and the linear fixed-point iteration does not converge. Therefore, we find that the following condition, which is moreover easily checked numerically, is both necessary and sufficient:

> If and only if $\rho(\mathbf{T}) < 1$, then Eq. 2.3 converges for any $\vec{x}_0$.
> If also Eq. 2.5 is fulfilled, convergence is to the correct solution $\vec{x}$.

The last remaining question is when to stop the iteration. For this, heuristic termination criteria are used that are usually of the type:

$$|\vec{x}_k - \vec{x}_{k-1}| \leq \text{RTOL}|\vec{x}_k| + \text{ATOL}. \tag{2.7}$$

for some user-defined relative tolerance RTOL and absolute tolerance ATOL, chosen according to how accurate the application at hand requires the results to be. This stops the iterations as soon as the change in the solution is below tolerance. Using both a relative and an absolute tolerance is good practice, because for solutions of small or large absolute value, one or the other could be more relevant. However, it is important to keep in mind that while these termination criteria are necessary indicators of convergence, they are not sufficient.

2.2.2 Jacobi method

Now that we know the basics of iterative methods, we look at a few classical examples of specific iterations used to solve linear systems of equations. They are usually based on decomposing the matrix into additive summands, e.g.,:

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$$

where here:

- $\mathbf{D}$ is a diagonal matrix containing all diagonal entries from $\mathbf{A}$, and zeros everywhere else.
- $\mathbf{L}$ is a lower triangular matrix containing all entries from $\mathbf{A}$ below (but without) the diagonal, and zeros everywhere else.
- $\mathbf{U}$ is an upper triangular matrix containing all entries from $\mathbf{A}$ above (but without) the diagonal, and zeros everywhere else.

Clearly then, the problem in Eq. 2.1 can be written as:

$$\begin{aligned} \mathbf{A}\vec{x} &= \mathbf{b} \\ \mathbf{D}\vec{x} + (\mathbf{L} + \mathbf{U})\vec{x} &= \vec{b} \\ \mathbf{D}\vec{x} &= \underbrace{-(\mathbf{L} + \mathbf{U})}_{\mathbf{S}}\vec{x} + \vec{b}, \end{aligned} \tag{2.8}$$

suggesting the Iteration:

$$\mathbf{D}\vec{x}_{k+1} = \mathbf{S}\vec{x}_k + \vec{b}. \tag{2.9}$$

This is the classic *Jacobi iteration* with $\mathbf{S} = -\mathbf{L} - \mathbf{U}$. Comparing this to Eq. 2.3, we can identify the iteration matrix and vector of the Jacobi method as: $\mathbf{T}_J = \mathbf{D}^{-1}\mathbf{S}$ and $\vec{c}_J = \mathbf{D}^{-1}\vec{b}$, respectively. If $\mathbf{D}$ is regular, then this iteration is consistent, because Eq. 2.5 becomes:

$$\mathbf{A}[\mathbf{1} - \mathbf{T}_J]^{-1}\vec{c}_J = \mathbf{A}[\mathbf{1} - \mathbf{D}^{-1}\mathbf{S}]^{-1}\mathbf{D}^{-1}\vec{b} = \mathbf{A}\vec{x} = \vec{b},$$

where in the last step we used that solving Eq. 2.8 for $\vec{x}$ yields:

$$\begin{aligned}\mathbf{D}\vec{x} &= \mathbf{S}\vec{x} + \vec{b}\\ \mathbf{D}\vec{x} - \mathbf{S}\vec{x} &= \vec{b}\\ [\mathbf{1} - \mathbf{D}^{-1}\mathbf{S}]\vec{x} &= \mathbf{D}^{-1}\vec{b}\\ \vec{x} &= [\mathbf{1} - \mathbf{D}^{-1}\mathbf{S}]^{-1}\mathbf{D}^{-1}\vec{b}.\end{aligned}$$

According to Theorems 2.3 and 2.4, the Jacobi method converges to the correct solution $\vec{x}$ for any start vector $\vec{x}_0$ if $\mathbf{D}$ is regular and $\left\|\mathbf{D}^{-1}\mathbf{S}\right\| < 1$, for which it is necessary and sufficient that $\rho(\mathbf{D}^{-1}\mathbf{S}) < 1$ (see Theorem 2.6).

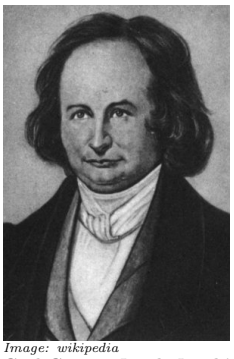


Image: wikipedia
Carl Gustav Jacob Jacobi
∗ 10 December 1804
Potsdam, Prussia
† 18 February 1851
Berlin, Prussia

Definition 2.3 (Diagonally dominant)**.** *A matrix* $\mathbf{A} = (a_{ij})$ *is* diagonally dominant *if and only if*

$$|a_{ii}| \geq \sum_{\substack{j=1\\ j\neq i}}^{n} |a_{ij}|, \qquad i = 1, 2, \ldots, n$$

and strictly diagonally dominant *if and only if*

$$|a_{ii}| > \sum_{\substack{j=1\\ j\neq i}}^{n} |a_{ij}|, \qquad i = 1, 2, \ldots, n.$$

Theorem 2.7. *The Jacobi iteration is consistent and converges if* $\mathbf{A}$ *is strictly diagonally dominant.*

Proof. Take $\|\mathbf{T}\|_\infty = \max_i \sum_{j=1}^{n} |t_{ij}|$, the maximum norm of the matrix $\mathbf{T}$, i.e., the largest row sum of absolute values. Then:

$$\|\mathbf{T}\|_\infty = \left\|\mathbf{D}^{-1}\mathbf{S}\right\|_\infty \Longrightarrow \max_i \sum_{j=1}^{n} |t_{ij}| = \max_i \sum_{\substack{j=1\\ j\neq i}}^{n} \frac{|a_{ij}|}{|a_{ii}|} < 1,$$

where in the last expression we have used the fact that $s_{ii} = 0$ because $\mathbf{S} = -\mathbf{L} - \mathbf{U}$ only has zeros on the diagonal. Therefore, $\|\mathbf{T}\|_\infty < 1$, which implies that $\rho(\mathbf{T}) < 1$, because the spectral radius is a lower bound on all matrix norms, and the iteration converges. □

Note that the condition is sufficient for convergence, but not necessary. Indeed, the Jacobi method is also found to converge for diagonally dominant matrices as long as they are irreducible (not discussed here).

2.2.3 Gauss-Seidel method

While the Jacobi method is simple and elegant, it typically converges slowly, i.e. $\|\mathbf{T}_J\| < 1$ but close to 1, requiring many iterations to determine a good

Image: wikipedia
Philipp Ludwig von Seidel
∗ 24 October 1821
Zweibrücken, Germany
† 13 August 1896
Munich, German Empire

approximation to the solution. Moreover, convergence is guaranteed (i.e., sufficient condition) only for strictly diagonally dominant matrices, which is a rather limited set of problems. Several related methods are available to improve convergence and/or enlarge the feasible problem set. the classic Gauss-Seidel method does so by considering a part of the already calculated new approximation in each iteration:

$$\mathbf{D}\vec{x}_{k+1} = -\mathbf{L}\vec{x}_{k+1} - \mathbf{U}\vec{x}_k + \vec{b}.$$

Therefore, the two summands of the Jacobi matrix $\mathbf{S}$ are considered separately, and the new solution $\vec{x}_{k+1}$ is used with one of them, whereas the old approximation $\vec{x}_k$ is used with the other. Solving this for $\vec{x}_{k+1}$ yields the iteration:

$$\vec{x}_{k+1} = -(\mathbf{D}+\mathbf{L})^{-1}\mathbf{U}\vec{x}_k + (\mathbf{D}+\mathbf{L})^{-1}\vec{b}. \tag{2.10}$$

The iteration matrix of the Gauss-Seidel method therefore is: $\mathbf{T}_{GS} = -(\mathbf{D}+\mathbf{L})^{-1}\mathbf{U}$ and the iteration vector is: $\vec{c}_{GS} = (\mathbf{D}+\mathbf{L})^{-1}\vec{b}$. The Gauss-Seidel method is consistent and converges for any start vector $\vec{x}_0$ if $\mathbf{D}$ is regular and $\left\|(\mathbf{D}+\mathbf{L})^{-1}\mathbf{U}\right\| < 1$. This turns out (not proven here) to be the case whenever $\mathbf{A}$ is:

1. either strictly diagonally dominant,
2. or symmetric and positive definite.

Definition 2.4 (Positive definite)**.** *A matrix* $\mathbf{A} \in \mathbb{R}^{n\times n}$ *is* positive definite *if and only if* $\vec{u}^{\mathsf{T}}\mathbf{A}\vec{u} > 0$ *for all* $\vec{u} \in \mathbb{R}^n \setminus \{0\}$.

Gauss-Seidel therefore converges for a larger class of matrices than Jacobi. In addition, the Gauss-Seidel method converges at about twice the speed of the Jacobi method, as, for a certain important class of matrices,

$$\|\mathbf{T}_{GS}\| \approx \|\mathbf{T}_J\|^2.$$

2.2.4 Successive Over-Relaxation (SOR) method

The Successive Over-Relaxation (SOR) method to iteratively solve linear systems of equations on a digital computer was independently developed by David M. Young Jr. and by Stanley P. Frankel, both in 1950. It further improves convergence speed over the Gauss-Seidel method by weighting between the old and new Gauss-Seidel vectors:

$$\mathbf{D}\vec{x}_{k+1} = \omega(-\mathbf{L}\vec{x}_{k+1} - \mathbf{U}\vec{x}_k + \vec{b}) + (1-\omega)\mathbf{D}\vec{x}_k$$

with weight $\omega \in \mathbb{R}^+$. Solving this for $\vec{x}_{k+1}$, one obtains the iteration:

$$\vec{x}_{k+1} = \underbrace{(\mathbf{D}+\omega\mathbf{L})^{-1}(-\omega\mathbf{U}+(1-\omega)\mathbf{D})}_{\mathbf{T}_{SOR}}\vec{x}_k + \underbrace{(\mathbf{D}+\omega\mathbf{L})^{-1}\omega}_{\vec{c}_{SOR}}\vec{b}. \tag{2.11}$$

The SOR method is consistent because $[\mathbf{1} - \mathbf{T}_{SOR}] = (\mathbf{D} + \omega\mathbf{L})^{-1}(\mathbf{D} + \omega\mathbf{L} + \omega\mathbf{U} - (1-\omega)\mathbf{D}) = (\mathbf{D} + \omega\mathbf{L})^{-1}\omega\mathbf{A}$ is regular (since $\mathbf{A}$ is regular by definition) and $[\mathbf{1} - \mathbf{T}_{SOR}]^{-1}\vec{c}_{SOR} = \mathbf{A}^{-1}\vec{b}$, fulfilling Eq. 2.5.
SOR converges for:

1. all $0 < \omega \leq 1$ for $\mathbf{A}$ for which the Jacobi method also converges,
2. all $0 < \omega < 2$ for symmetric and positive definite $\mathbf{A}$.

For $\omega \geq 2$, the SOR method never converges.
The art in using SOR effectively is to choose good values for the *relaxation parameter* ω. A classic good choice is

$$\omega = \frac{2}{1 + \sqrt{1 - \rho(\mathbf{T}_J)^2}},$$

for which the SOR method converges much faster than the Gauss-Seidel method. For certain matrices $\mathbf{A}$ with a special block-diagonal form (so-called T-matrices), this choice of ω is provably optimal. This is particularly relevant in applications of numerically solving partial differential equations (see Sec. 11.2).

Image: computerhope.com
David M. Young Jr.
* 20 October 1923
Quincy, Mass., USA
† 21 December 2008
Austin, Texas, USA

Image: wikipedia
Stanley Phillips Frankel
* 1919
Los Angeles, CA, USA
† May 1978
Los Angeles, CA, USA

2.3 Gradient Descent

If the matrix $\mathbf{A}$ is symmetric and positive definite, the linear system of equations can be formulated as an equivalent convex optimization problem, which can then be solved using gradient descent approaches. The problem in Eq. 2.1 can equivalently be written as:

$$\mathbf{A}\vec{x} + \vec{b} = 0 \tag{2.12}$$

when replacing $\vec{b}$ with $-\vec{b}$. For symmetric and positive definite $\mathbf{A}$, we have:

1. all eigenvalues $\lambda_i > 0,\ i = 1, 2, 3, \ldots, n \implies \mathbf{A}$ is regular since $\det(\mathbf{A}) = \prod_i \lambda_i > 0$,
2. $\exists \mathbf{T}$ orthogonal, such that $\mathbf{T}^\mathsf{T}\mathbf{A}\mathbf{T} = \mathbf{D} = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$.

We do not prove these statements here, as they are well-known facts for symmetric, positive definite matrices.
Now consider the functional

$$F(u) := \frac{1}{2}\vec{u}^\mathsf{T}\mathbf{A}\vec{u} + \vec{u}^\mathsf{T}\vec{b} \tag{2.13}$$

with gradient

$$\nabla F = \begin{bmatrix} \frac{\partial F}{\partial u_1} \\ \vdots \\ \frac{\partial F}{\partial u_n} \end{bmatrix} = \mathbf{A}\vec{u} + \vec{b} =: \vec{r}(\vec{u}). \tag{2.14}$$

We call $\vec{r}(\vec{u})$ the *residual*. Now we claim:

Theorem 2.8. *If and only if $\vec{x}$ is a solution of the linear system of equations $\mathbf{A}\vec{x}+\vec{b}=0$ for symmetric, positive definite $\mathbf{A}$, then $\vec{x}$ is also a minimum of the functional $F(u)=\frac{1}{2}\vec{u}^\mathsf{T}\mathbf{A}\vec{u}+\vec{u}^\mathsf{T}\vec{b}$, and vice versa. Hence:*

$$\vec{x}=-\mathbf{A}^{-1}\vec{b}=\arg\min_{\vec{u}\in\mathbb{R}^n} F(\vec{u}).$$

Proof. We first prove the inverse direction, i.e., that $\vec{x}$ being a minimum of F implies that it solves the linear system. This is easy to see, because if $\vec{u}=\vec{x}$ is a minimum of F, then $\nabla F(\vec{x})=\mathbf{A}\vec{x}+\vec{b}=0$, which solves the linear system. Now we prove the forward direction, i.e., that if $\vec{x}$ solves the linear system then this implies that it is also a minimum of F. Certainly $\mathbf{A}\vec{x}+\vec{b}=0\implies\nabla F(\vec{x})=0$, implying that $\vec{x}$ is an extremal point of F (i.e., a minimum, maximum, or saddle point). To show that it actually is a minimum, we perturb the point by any $\vec{v}\in\mathbb{R}^n\neq\vec{0}$ and observe that:

$$\begin{aligned}
F(\vec{x}+\vec{v}) &= \frac{1}{2}(\vec{x}+\vec{v})^\mathsf{T}\mathbf{A}(\vec{x}+\vec{v})+(\vec{x}+\vec{v})^\mathsf{T}\vec{b}\\
&= F(\vec{x})+\frac{1}{2}\left(\underbrace{\vec{x}^\mathsf{T}\mathbf{A}}_{(\mathbf{A}\vec{x})^\mathsf{T}=-\vec{b}^\mathsf{T}}\vec{v}+\vec{v}^\mathsf{T}\underbrace{\mathbf{A}\vec{x}}_{-\vec{b}}+\vec{v}^\mathsf{T}\mathbf{A}\vec{v}\right)+\vec{v}^\mathsf{T}\vec{b}\\
&= F(\vec{x})+\underbrace{\frac{1}{2}\vec{v}^\mathsf{T}\mathbf{A}\vec{v}}_{>0}>F(\vec{x}).
\end{aligned}$$

In the second step, we used the fact that $\mathbf{A}$ is symmetric, and in the last step we used the fact that $\mathbf{A}$ is positive definite. Sine the value of F is hence increasing in every direction, $\vec{x}$ is a minimum. □

The theorem provides us with an alternative way of solving the linear system in Eq. 2.12, namely by finding the minimum of the functional in Eq. 2.13. Since the solution of the linear system is unique for regular $\mathbf{A}$, there exists only one minimum, implying that the functional is convex. Any minimum we find therefore is *the* minimum.

2.3.1 Geometric interpretation

The question of course is *how* the minimum of F can be numerically computed. For this, it is instructive to look at an example in $n=2$ dimensions and to interpret $y=F(\vec{u})$ as a function over the 2D plane. The minimum of the function is at location $\vec{x}$ and has value $m\coloneqq F(\vec{x})$. Applying the transformation $\vec{v}:=\vec{u}-\vec{x}$ and $w:=y-m$ shifts the minimum to be located at $\vec{0}$ and have value 0. For the function value, we find:

$$\begin{aligned}
w=y-m=F(\vec{u})-F(\vec{x}) &= \frac{1}{2}\vec{u}^\mathsf{T}\mathbf{A}\vec{u}+\vec{u}^\mathsf{T}\vec{b}-\frac{1}{2}\vec{x}^\mathsf{T}\mathbf{A}\vec{x}-\underbrace{\vec{x}^\mathsf{T}\vec{b}}_{-\vec{x}^\mathsf{T}\mathbf{A}\vec{x}}\\
&= \frac{1}{2}\vec{u}^\mathsf{T}\mathbf{A}\vec{u}+\vec{u}^\mathsf{T}\vec{b}+\frac{1}{2}\vec{x}^\mathsf{T}\mathbf{A}\vec{x}.
\end{aligned}$$

We also find that:

$$\vec{v}^\mathsf{T}\mathbf{A}\vec{v} = (\vec{u}-\vec{x})^\mathsf{T}\mathbf{A}(\vec{u}-\vec{x}) = \vec{u}^\mathsf{T}\mathbf{A}\vec{u} - \underbrace{\vec{u}^\mathsf{T}\mathbf{A}\vec{x}}_{-\vec{u}^\mathsf{T}\vec{b}} - \underbrace{\vec{x}^\mathsf{T}\mathbf{A}}_{(\mathbf{A}\vec{x})^\mathsf{T}=-\vec{b}^\mathsf{T}}\vec{u} + \vec{x}^\mathsf{T}\mathbf{A}\vec{x}$$
$$= \vec{u}^\mathsf{T}\mathbf{A}\vec{u} + 2\vec{u}^\mathsf{T}\vec{b} + \vec{x}^\mathsf{T}\mathbf{A}\vec{x}$$

and therefore:

$$w = \frac{1}{2}\vec{v}^\mathsf{T}\mathbf{A}\vec{v} = G(\vec{v}).$$

The shifted function $G(\vec{v}) := F(\vec{u}-\vec{x}) - m$ has the gradient $\nabla G(\vec{v}) = \mathbf{A}\vec{v}$. This function in two dimensions, $w = G(v_1, v_2)$, is drawn in Fig. 2.1 in both side and top views.

Image: wikipedia
Augustin-Louis Cauchy
∗ 21 August 1789
Paris, France
† 23 May 1857
Sceaux, France

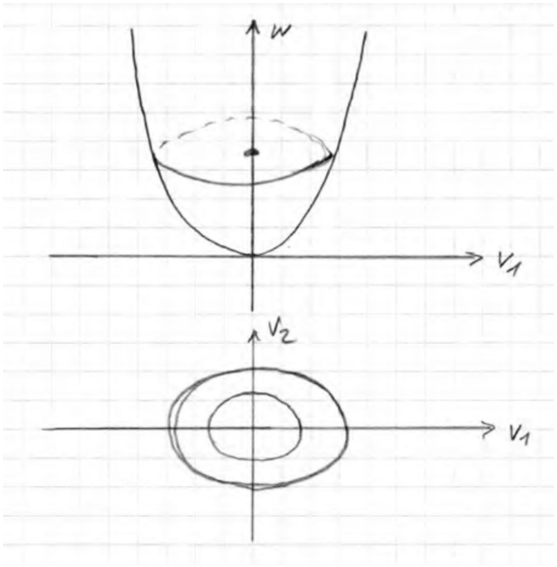


Figure 2.1: Side and top views of $w = G(v_1, v_2)$ with contour lines.

The contour lines (iso-lines) $G(\vec{v}) = \text{const}$ are ellipses, and the residual $\vec{r} = \nabla G$ is always perpendicular to these ellipses. The gradient of any function is everywhere perpendicular to the function's contour line through that point and points in the direction of steepest increase of the function value. Since we aim to minimize G, and the gradient can be computed at every point by a simple matrix-vector multiplication $\nabla G(\vec{v}) = \mathbf{A}\vec{v}$, the idea is to iteratively take steps in direction of the negative gradient in order to approach the minimum of the function. This method is called *gradient descent*, and it constitutes a generic recipe for finding minima in functions.

2.3.2 Algorithm

The gradient descent algorithm was suggested by Augustin-Louis Cauch in 1847. It starts from an initial point $\vec{u}_0$ and looks for the minimum in direction $-\nabla F(\vec{u}) = -(\mathbf{A}\vec{u} + \vec{b})$ (cf. Eq. 2.14). The algorithm is given in Algorithm 3. It requires setting a step size α that governs what fraction of the gradient magnitude the method walks down in one step. This is a critical parameter. If it set too large, the minimum can be overshot. If it is set too small, the algorithm converges very slowly. For gradient descent over quadratic forms, such as Eq. 2.13, the step size α must be chosen such that the (real) eigenvalues of $[\mathbf{1} - \alpha\mathbf{A}]$ are in the open interval $(-1, 1)$. Then the algorithm converges.

Algorithm 3 Gradient descent

```
1: procedure GradientDescent(u⃗_0, RTOL, ATOL)
2:         ▷ starting value u⃗_0, relative and absolute tolerances RTOL, ATOL
3:     k = −1
4:     repeat
5:         k = k + 1
6:         u⃗_{k+1} = u⃗_k − α(A u⃗_k + b⃗)                ▷ step down the gradient
7:     until |u⃗_{k+1} − u⃗_k| ≤ |u⃗_{k+1}|RTOL + TOL
8: end procedure
```

Instead of using a fixed step size α, one can also perform a line search (e.g., bisection search) along the negative gradient, which always jumps to the next point where the direction of the gradient is tangential to a contour ellipse. For the shifted problem, this point can be computed analytically, as:

$$\begin{aligned}
[\mathbf{A}(\vec{v} - \alpha\mathbf{A}\vec{v})]^\mathsf{T}\mathbf{A}\vec{v} &= 0 \\
(\vec{v} - \alpha\mathbf{A}\vec{v})^\mathsf{T}\mathbf{A}\mathbf{A}\vec{v} &= 0 \\
(\mathbf{A}\vec{v})^\mathsf{T}(\mathbf{A}\vec{v}) - \alpha(\mathbf{A}\vec{v})^\mathsf{T}\mathbf{A}(\mathbf{A}\vec{v}) &= 0 \\
\alpha = \frac{|\mathbf{A}\vec{v}|_2^2}{(\mathbf{A}\vec{v})^\mathsf{T}\mathbf{A}(\mathbf{A}\vec{v})} &= \frac{|\mathbf{A}\vec{v}|_2^2}{|\mathbf{A}\vec{v}|_\mathbf{A}^2},
\end{aligned} \tag{2.15}$$

where in the second step we used the fact that $\mathbf{A}$ is symmetric. The quantity $|p|_\mathbf{A}^2 = \langle\vec{p}, \mathbf{A}\vec{p}\rangle = \vec{p}^\mathsf{T}\mathbf{A}\vec{p} > 0$ is the square of the $\mathbf{A}$-norm of a vector $\vec{p} \in \mathbb{R}^n$ for a positive definite matrix $\mathbf{A}$. This, however, is not useful on the original problem, since computing $\vec{v} = \vec{u} - \vec{x}$ requires knowing the solution $\vec{x}$. A fixed step size, or line search, are therefore used in practice.

A general disadvantage of gradient descent is that the convergence toward the minimum is slow if the condition number of $\mathbf{A}$ is large. According to Eq. 1.28 for symmetric matrices, the condition number is:

$$\kappa_\mathbf{A} = \frac{|\lambda_\text{max}|}{|\lambda_\text{min}|}.$$

The largest and smallest eigenvalues of $\mathbf{A}$, however, correspond to the lengths of the longest and shortest half-axes of the ellipsoidal contours of F. Therefore,

a large condition number implies a large aspect ratio for the ellipses (i.e., very elongated ellipses). You can easily convince yourself with a little drawing that in this case, taking steps in directions perpendicular to the ellipses takes a very long time to reach the center.

2.4 Conjugate Gradient Method

Image: wikipedia
Eduard Stiefel
∗ 21 April 1909
Zürich, Switzerland
† 25 November 1978
Zürich, Switzerland

Image: wikipedia
Magnus Hestenes
∗ 13 February 1906
Bricelyn, Minnesota, USA
† 31 May 1991
Los Angeles, CA, USA

The convergence for problems for large $\kappa_{\mathbf{A}}$ can be significantly improved by introducing the concept of *conjugate directions*.

Definition 2.5 (Conjugate direction)**.** *Let $F(\vec{u}) \in \mathbb{R}$ be a continuously differentiable function with gradient $\nabla F = \vec{r}$. Let $\vec{x}_1$ be the point where the direction $-\vec{r}(\vec{x}_0)$ is tangential to a contour of the function. The* conjugate direction *to $-\vec{r}(\vec{x}_0)$ then points from $\vec{x}_1$ to the origin.*

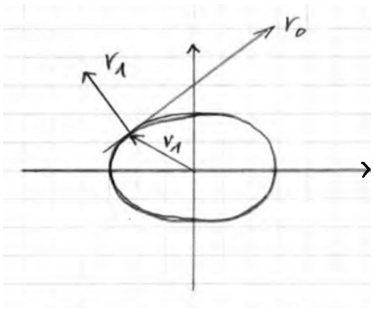


Figure 2.2: $\vec{v}_1$ is the conjugate direction of $-\vec{r}_0$.

For the shifted problem $G(\vec{v})$, which has the minimum at 0, the conjugate direction to the gradient points to the minimum in 2D. Looking for the minimum along this direction is going to solve the problem. This is illustrated in Fig. 2.2, where $\vec{r}_1 = \nabla G(\vec{v}_1) = \mathbf{A}\vec{v}_1$. The tangent to the contour ellipse at $\vec{v_1}$ is $\vec{r}_0$ and therefore

$$-\vec{r}_0 \perp \vec{r}_1 \Longleftrightarrow -\vec{r_0}^{\mathsf{T}}\vec{r}_1 = 0.$$

Substituting the expression for $\vec{r}_1$: $-\vec{r_0}^{\mathsf{T}}\mathbf{A}\vec{v}_1 = 0$, which implies that $-\vec{r}_0$ and $\vec{v}_1$ are conjugate directions.
In general, for the functional from Eq. 2.13, two directions $\vec{p}$ and $\vec{q}$ are conjugate if

$$\vec{p}^{\mathsf{T}}\mathbf{A}\vec{q} = 0.$$

In 3D (i.e., $n = 3$), the contours of $G(\vec{v})$ are ellipsoids and the conjugate gradient method consists of the steps:

1. Gradient descent (direction $-\vec{r}_0$) until the next tangential ellipsoid is touched in point $\vec{r}_1$.

2. Search for the minimum in the plane spanned by $-\vec{r}_0$ and $\vec{r}_1$. This plane intersects the contour ellipsoid in a concentric ellipse. The minimum in this plane is located at the center of this ellipse. The direction $\vec{p}_2$ that is conjugate to $\vec{r}_0$ points to the minimum (see Fig. 2.3).

3. Compute $\vec{r}_2 = \nabla G(\vec{v}_2)$ and search for the minimum in the plane spanned by $\vec{r}_2$ and $\vec{p}_2$. The point $\vec{v}_3$ thus found is the origin and hence the minimum is found.

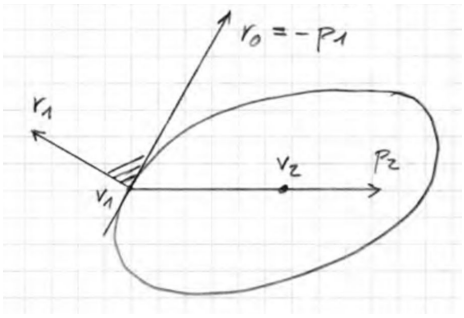


Figure 2.3: Illustration of one step of the conjugate gradient method in 3D.

Therefore, the conjugate gradient method is guaranteed to find the minimum in 3D in 3 steps. The conjugate gradient method was developed by Magnus Hestenes and Eduard Stiefel in 1952 when programming the Zuse Z4 computer to numerically solve differential equations at ETH Zurich, Switzerland.

Derivation in arbitrary dimensions

The general procedure in n dimensions and for the original, unshifted problem can be derived analogously. Starting from an initial point $\vec{u}_0$ with gradient $\vec{r}_0 = \nabla F(\vec{u}_0)$, the first descent is along $\vec{p}_1 = -\vec{r}_0$. We then find the point where the first contour is tangential to the direction of descent, i.e., where F is minimal along the line of descent, as:

$$\vec{u}_1 = \vec{u}_0 - \rho_1 \vec{r}_0$$

$$\vec{r}_1 = \nabla F(\vec{u}_1)$$

with the distance (i.e., step size)

$$\rho_1 = \frac{-\langle \vec{r}_0, \vec{p}_1 \rangle}{\langle \mathbf{A}\vec{p}_1, \vec{p}_1 \rangle}.$$

Note that this is the same expression we also found in Eq. 2.15. The proof that also in the general, unshifted case, the minimum of F in direction $\vec{r}_0$ is at $\vec{u}_1$ is:

Proof.

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\rho}F(\vec{u}+\rho\vec{p}) &= \frac{\mathrm{d}}{\mathrm{d}\rho}\left[\frac{1}{2}(\vec{u}+\rho\vec{p})^\mathsf{T}\mathbf{A}(\vec{u}+\rho\vec{p}) + (\vec{u}+\rho\vec{p})^\mathsf{T}\vec{b}\right] \\
&= \frac{\mathrm{d}}{\mathrm{d}\rho}\frac{1}{2}\left[\vec{u}^\mathsf{T}\mathbf{A}\vec{u} + \rho\vec{p}^\mathsf{T}\mathbf{A}\vec{u} + \rho \underbrace{\vec{u}^\mathsf{T}\mathbf{A}\vec{p}}_{(\vec{u}^\mathsf{T}\mathbf{A}\vec{p})^\mathsf{T}=\vec{p}^\mathsf{T}\mathbf{A}^\mathsf{T}\vec{u}=\vec{p}^\mathsf{T}\mathbf{A}\vec{u}} + \rho^2\vec{p}^\mathsf{T}\mathbf{A}\vec{p}\right] + \vec{p}^\mathsf{T}\vec{b} \\
&= \vec{p}^\mathsf{T}\mathbf{A}\vec{u} + \rho\vec{p}^\mathsf{T}\mathbf{A}\vec{p} + \vec{p}^\mathsf{T}\vec{b} \stackrel{!}{=} 0 \\
&\Longrightarrow \rho = -\frac{\vec{p}^\mathsf{T}\mathbf{A}\vec{u}+\vec{p}^\mathsf{T}\vec{b}}{\vec{p}^\mathsf{T}\mathbf{A}\vec{p}} = -\frac{\vec{p}^\mathsf{T}(\mathbf{A}\vec{u}+\vec{b})}{\vec{p}^\mathsf{T}\mathbf{A}\vec{p}} = -\frac{\langle\vec{p},\vec{r}\rangle}{\langle\mathbf{A}\vec{p},\vec{p}\rangle},
\end{aligned}$$

where in the second step we have used the fact that for symmetric matrices $\mathbf{A} = \mathbf{A}^\mathsf{T}$: $\langle\mathbf{A}\vec{p},\vec{q}\rangle = \langle\vec{p},\mathbf{A}\vec{q}\rangle$. □

Then, the algorithm enters the iteration $k = 2, 3, 4, \ldots$ and in each iteration computes the search plane as:

$$\vec{p}_k = -\vec{r}_{k-1} + e_{k-1}\vec{p}_{k-1}$$

and the direction conjugate to $\vec{p}_{k-1}$ as:

$$\langle\vec{p}_k, \mathbf{A}\vec{p}_{k-1}\rangle = 0.$$

Substituting the first expression into the second, we can solve for the unknown e_{k-1}:

$$\begin{aligned}
\langle\vec{p}_k, \mathbf{A}\vec{p}_{k-1}\rangle &= -\langle\vec{r}_{k-1}, \mathbf{A}\vec{p}_{k-1}\rangle + e_{k-1}\langle\vec{p}_{k-1}, \mathbf{A}\vec{p}_{k-1}\rangle \\
\Longrightarrow e_{k-1} &= \frac{\langle\vec{r}_{k-1}, \mathbf{A}\vec{p}_{k-1}\rangle}{\langle\vec{p}_{k-1}, \mathbf{A}\vec{p}_{k-1}\rangle}.
\end{aligned}$$

The distance until the minimum in direction $\vec{p}_k$ is found is (see above):

$$\begin{aligned}
\vec{u}_k &= \vec{u}_{k-1} + \rho_k\vec{p}_k \\
\Longrightarrow \rho_k &= -\frac{\langle\vec{r}_{k-1}, \vec{p}_k\rangle}{\langle\mathbf{A}\vec{p}_k, \vec{p}_k\rangle}.
\end{aligned}$$

Finally, we compute the gradient at $\vec{u}_k$ as:

$$\vec{r}_k = \nabla F(\vec{u}_k) = \mathbf{A}\vec{u}_k + \vec{b}$$

and iterate.
It is easily verified that in each of these iterations, we have the orthogonality relations:

1. $\langle \vec{r}_k, \vec{r}_{k-1} \rangle = 0$,
2. $\langle \vec{r}_k, \vec{p}_k \rangle = 0$,
3. $\langle \vec{r}_k, \vec{p}_{k-1} \rangle = 0$.

The geometric interpretation of these is that in each iteration of the algorithm, the plane spanned by $\vec{r}_{k-1}$ and $\vec{p}_{k-1}$ is tangential to the contour (i.e., perpendicular to the gradient $\vec{r}_k$) at $\vec{u}_k$. Therefore, $\vec{u}_k$ is a minimum of F in this plane. Also $\vec{p}_k$ lies within this plane, and $\vec{r}_k$ is perpendicular to $\vec{r}_{k-1}$.
From these orthogonality relations, we can simplify some of the above expressions to:

$$\rho_k = \frac{\langle \vec{r}_{k-1}, \vec{r}_{k-1} \rangle}{\langle \mathbf{A}\vec{p}_k, \vec{p}_k \rangle} > 0$$
$$\vec{r}_k = \vec{r}_{k-1} + \rho_k \mathbf{A}\vec{p}_k$$
$$e_{k-1} = \frac{\langle \vec{r}_{k-1}, \vec{r}_{k-1} \rangle}{\langle \vec{r}_{k-2}, \vec{r}_{k-2} \rangle} > 0.$$

The general algorithm for conjugate gradient solvers in n dimensions is given in Algorithm 4, according to the derivation in the box above.

Algorithm 4 Conjugate gradient descent

procedure ConjugateGradient($\vec{u}_0$) ▷ start point $\vec{u}_0$
  $\vec{r}_0 = \mathbf{A}\vec{u}_0 + \vec{b}$
  $\vec{p}_1 = -\vec{r}_0$
  for $k = 1, 2, \ldots, n$ **do**
    if $k \geq 2$ **then**
      $e_{k-1} = \frac{\langle \vec{r}_{k-1}, \vec{r}_{k-1} \rangle}{\langle \vec{r}_{k-2}, \vec{r}_{k-2} \rangle}$
      $\vec{p}_k = -\vec{r}_{k-1} + e_{k-1}\vec{p}_{k-1}$
    end if
    $\rho_k = \frac{\langle \vec{r}_{k-1}, \vec{r}_{k-1} \rangle}{\langle \mathbf{A}\vec{p}_k, \vec{p}_k \rangle}$
    $\vec{u}_k = \vec{u}_{k-1} + \rho_k \vec{p}_k$
    $\vec{r}_k = \vec{r}_{k-1} + \rho_k \mathbf{A}\vec{p}_k$
  end for
end procedure

Each iteration of Algorithm 4 requires one matrix-vector multiplication ($O(n^2)$) to compute $\mathbf{A}\vec{p}_k$ and two scalar products ($O(n)$ each) to compute $\langle \vec{r}_{k-1}, \vec{r}_{k-1} \rangle$ and $\langle \mathbf{A}\vec{p}_k, \vec{p}_k \rangle$. The algorithm is therefore very efficient.

Theorem 2.9. *At each iteration k of Algorithm 4, the $\vec{p}_j$ ($0 \leq j \leq k$) are*

pairwise conjugate, and the $\vec{r}_k$ are pairwise orthogonal. Therefore, the conjugate gradient Algorithm 4 computes the solution to $\mathbf{A}\vec{x}+\vec{b}=0$ for symmetric, positive definite $\mathbf{A}\in\mathbb{R}^{n\times n}$ in at most n iterations.

Proof. The $\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_{n-1}$ are pairwise orthogonal. Therefore, $\vec{r}_n$ has to be orthogonal to all of $\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_{n-1}$, which is only possible in an n-dimensional space if $\vec{r}_n = 0$. This means that $\vec{r}_n = \mathbf{A}\vec{u}_n + \vec{b} = 0$ and thus $\vec{u}_n = \vec{x}$. □

This theorem, however, assumes exact arithmetics. Using finite-precision pseudo-arithmetics, more than n steps may be necessary. This is because rounding errors lead to the $\vec{r}_j$ not being exactly orthogonal to one another, which means that the in-plane minima might be missed by a small margin so that more than n iterations may be necessary in practice. A tolerance-based termination criterion, like the one in Algorithm 3, is therefore mostly used in practice for conjugate gradient methods.

A typical use case for conjugate gradient solvers is to solve the sparse, but large linear systems of equations obtained from spatial discretization of partial differential equations (PDEs). In this case, one often deliberately performs less than n steps in order to save time, since only an approximate solution is possible anyway. For finite-precision pseudo-arithmetics, the conjugate gradient method converges as:

$$|\vec{u}_k - \vec{x}|_{\mathbf{A}} \leq 2\alpha^k |\vec{u}_0 - \vec{x}|_{\mathbf{A}}$$

with linear convergence factor

$$\alpha = \frac{\sqrt{\kappa_{\mathbf{A}}} - 1}{\sqrt{\kappa_{\mathbf{A}}} + 1}.$$

This means that conjugate gradients do not converge well if the condition number of the matrix $\mathbf{A}$, $\kappa_{\mathbf{A}}$, is large. Conjugate gradient methods should, therefore, not be used for solving linear systems that result from discretizing stiff differential equations (see Sec. 10.8).

Image: wikipedia
Aleksey Krylov
∗ 15 August 1863
Alatyrsky uezd, Simbirsk Gubernia, Russian Empire
† 26 October 1945
Leningrad, USSR
(now St. Petersburg, Russia)

Example 2.3. *We illustrate this in an example for two assumed condition numbers:*

- $\kappa_{\mathbf{A}} = 9 \Longrightarrow \alpha = \frac{1}{2}$,
- $\kappa_{\mathbf{A}} = 100 \Longrightarrow \alpha = \frac{9}{11}$.

Therefore, already for a moderate condition number of 100, convergence is extremely slow (but still much better than that of normal gradient descent!). In practical applications, conditions numbers can be in the tens of thousands, limiting the use of direct conjugate gradient solvers.

Image: wikipedia
André-Louis Cholesky
∗ 15 October 1875
Montguyon, France
† 31 August 1918
Bagneux, France

While the conjugate gradient method as given in Algorithm 4 requires the matrix $\mathbf{A}$ to be symmetric and positive definite, it is a representative of a larger class of linear systems solvers, collectively called *Krylov subspace methods*, where methods for asymmetric and not positive definite matrices also exist (e.g., BICG or GMRES). These however, while more general, are typically less numerically robust than conjugate gradient descent.

2.5 Pre-Conditioning

Pre-conditioning methods are available to reduce the condition number of the matrix before entering the solver. The idea is to multiply the entire system of equations from the left with a matrix $\mathbf{H}^{-1}$. Let $\mathbf{C} = \mathbf{C}^\mathsf{T}$ be a symmetric, positive definite matrix with Cholesky factors $\mathbf{C} = \mathbf{H}\mathbf{H}^\mathsf{T}$ (determined using Cholesky decomposition), then $\mathbf{A}\vec{x} + \vec{b} = 0$ is equivalent to

$$\begin{aligned} \mathbf{H}^{-1}\mathbf{A}\vec{x} + \mathbf{H}^{-1}\vec{b} &= 0, \\ \mathbf{H}^{-1}\mathbf{A}\underbrace{\mathbf{H}^{-\mathsf{T}}\mathbf{H}^\mathsf{T}}_{\mathbf{1}}\vec{x} + \mathbf{H}^{-1}\vec{b} &= 0. \end{aligned}$$

This means that we can equivalently solve the system by using:

$$\begin{aligned} \widetilde{\mathbf{A}} &\coloneqq \mathbf{H}^{-1}\mathbf{A}\mathbf{H}^{-\mathsf{T}}, \\ \widetilde{\vec{x}} &\coloneqq \mathbf{H}^\mathsf{T}\vec{x}, \\ \widetilde{\vec{b}} &\coloneqq \mathbf{H}^{-1}\vec{b}. \end{aligned}$$

We observe that $\widetilde{\mathbf{A}}$ is related to $\mathbf{C}^{-1}\mathbf{A}$ in the sense that

$$\mathbf{H}^{-\mathsf{T}}\widetilde{\mathbf{A}}\mathbf{H}^\mathsf{T} = \mathbf{H}^{-\mathsf{T}}\mathbf{H}^{-1}\mathbf{A} = (\mathbf{H}\mathbf{H}^\mathsf{T})^{-1}\mathbf{A} = \mathbf{C}^{-1}\mathbf{A}.$$

Obviously, a small condition number of the transformed problem is obtained by choosing $\mathbf{C} = \mathbf{A}$ because then $\widetilde{\mathbf{A}}$ is similar to $\mathbf{1}$ and therefore $\kappa_{\widetilde{\mathbf{A}}} \approx 1$. However, this is not meaningful, since finding $\widetilde{\mathbf{A}}$ in this case is equivalent to solving the original problem. Rather, $\mathbf{C}$ should be an *approximation* of $\mathbf{A}$.
One frequent choice is to find $\mathbf{H}$ by incomplete Cholesky decomposition of $\mathbf{A}$, i.e., by not computing an iterative Cholesky decomposition algorithm all the way to the end. This is particularly beneficial for sparse $\mathbf{A}$, because the fill-in is limited when stopping the decomposition prematurely, also leading to sparse pre-conditioners. We then have $\mathbf{A} \approx \mathbf{H}\mathbf{H}^\mathsf{T} = \mathbf{C}$. While incomplete Cholesky decomposition is possible for almost all matrixes (in particular for the so-called M-matrices), there also exist cases where it is not possible.
The preconditioning can directly be integrated into the conjugate gradient algorithm. In this case the changes are:

1. Compute the incomplete Cholesky decomposition of $\mathbf{A}$ once, before starting the first iteration.

2. In each iteration, additionally solve the linear system $\mathbf{H}\mathbf{H}^\mathsf{T}\vec{v}_k = \vec{r}_k$ by efficient forward and backward substitution ($O(n)$) and use the $\vec{v}_k$ as corrections to the directions $\vec{u}_k$ (see literature for details).

Chapter 3

Linear Least-Squares Problems

Often in modeling, we face the task of finding values of model parameters such that the model optimally fits some given data. For this to work, there of course have to be more data points than model parameters. Let the vector of unknown model parameters be $\vec{x} \in \mathbb{R}^n$ and the vector of data points or measurements $\vec{c} \in \mathbb{R}^m$ with $m > n$. For a linear model, we have that $\mathbf{A}\vec{x}$, with $\text{rank}(\mathbf{A}) = n$, should somehow represent $\vec{c}$. Every output of a linear model is a linear combination (i.e., weighted sum) of the model parameters. The weights $\mathbf{A}$ are the model (or model matrix). Obviously, simply setting $\mathbf{A}\vec{x} = \vec{c}$ cannot be solved, because there are more equations than unknowns. Instead, we want to determine the model parameter values $\vec{x}$ such that the data $\vec{c}$ is approximated as well as possible for every measurement, in the sense that the fitting error $\mathbf{A}\vec{x} - \vec{c}$ is minimized in a suitable norm.

3.1 Error Equations and Normal Equations

We start by considering a concrete and small example and then generalize to arbitrary n and m from there. For now, consider a problem with $n = 2$ parameters and $m = 3$ data points. We then specifically have:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 - c_1 &= r_1 \\ a_{21}x_1 + a_{22}x_2 - c_2 &= r_2 \\ a_{31}x_1 + a_{32}x_2 - c_3 &= r_3, \end{aligned}$$

where $\mathbf{A} = (a_{ij}) \in \mathbb{R}^{3\times 2}$, $\vec{x} = (x_1, x_2)^\mathsf{T}$, and $\vec{c} = (c_1, c_2, c_3)^\mathsf{T}$. This set of equations is called the *error equations* because the values $\vec{r} = (r_1, r_2, r_3)^\mathsf{T}$ measure the fitting errors, i.e., the differences between the model output and the data values. These errors $\vec{r}$ are called *residuals*. If all residuals are zero, then the

model perfectly fits all data points. In vector notation, the error equation is:

$$x_1\vec{a}^{(1)} + x_2\vec{a}^{(2)} - \vec{c} = \vec{r}, \tag{3.1}$$

where $\vec{a}^{(1)}$ is the first column vector of $\mathbf{A}$ and $\vec{a}^{(2)}$ the second. The geometric interpretation is depicted in Fig. 3.1. The two column vectors $\vec{a}^{(1)}$, $\vec{a}^{(2)}$ span a plane α, as any two vectors do if they are not collinear (in which case the model effectively only has one parameter, which we avoid by requiring $\text{rank}(\mathbf{A}) = n$). The point $x_1\vec{a}^{(1)} + x_2\vec{a}^{(2)}$ lies within this plane, as x_1, x_2 can simply be understood as the coordinates of this point in the (not necessarily orthogonal) coordinate system spanned by the axes $\vec{a}^{(1)}$, $\vec{a}^{(2)}$. Unless the model is able to perfectly reproduce all data, the vector $\vec{c}$ is not contained in the plane α, but points out of it. The residual vector is then simply the difference between the two (see Fig. 3.1 left). The Euclidean length of the residual is minimized if and only if $\vec{r}$ is perpendicular to α. The coordinates (x_1, x_2) of the point where the perpendicular $\vec{r}$ intersects the plane α are therefore the optimal model parameters under the 2-norm, i.e., those for which the sum of the squares of the residuals for the given data is minimal. Hence, the problem of finding the point (x_1, x_2) can easily be formulated in the Euclidean 2-norm as the *least-squares problem*.

Least-squares problems are classic and were first studied by Carl Friedrich Gauss in 1795. Since then, they have been one of the workhorses of numerical computing, modeling, engineering, statistics, machine learning, and data science. In statistics and machine learning, the residual is often called the *loss* or *loss function*, and using the 2-norm is often referred to as using a *quadratic loss*.

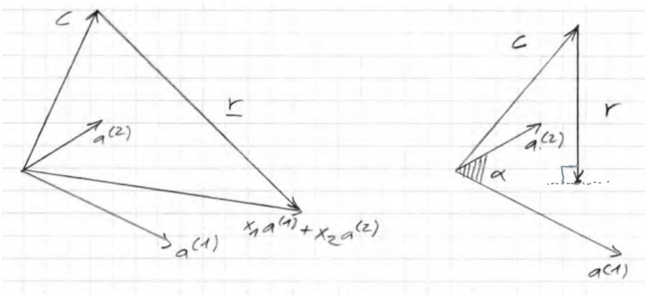


Figure 3.1: The residual $\vec{r}$ is minimal if and only if it is perpendicular to the plane spanned by the column vectors of the model matrix.

From the requirement that the optimal $\vec{r}$ is perpendicular to the plane α, and therefore orthogonal to both $\vec{a}^{(1)}$ and $\vec{a}^{(2)}$, we get the following equations:

$$\langle\vec{a}^{(1)}, \vec{r}\rangle = 0,$$
$$\langle\vec{a}^{(2)}, \vec{r}\rangle = 0.$$

By substituting the left-hand side of Eq. 3.1, we find:

$$\langle \vec{a}^{(1)}, x_1\vec{a}^{(1)} + x_2\vec{a}^{(2)} - \vec{c}\rangle = 0,$$
$$\langle \vec{a}^{(2)}, x_1\vec{a}^{(1)} + x_2\vec{a}^{(2)} - \vec{c}\rangle = 0.$$

Multiplying out the scalar products, this is:

$$\langle \vec{a}^{(1)}, \vec{a}^{(1)}\rangle x_1 + \langle \vec{a}^{(1)}, \vec{a}^{(2)}\rangle x_2 - \langle \vec{a}^{(1)}, \vec{c}\rangle = 0,$$
$$\langle \vec{a}^{(2)}, \vec{a}^{(1)}\rangle x_1 + \langle \vec{a}^{(2)}, \vec{a}^{(2)}\rangle x_2 - \langle \vec{a}^{(2)}, \vec{c}\rangle = 0.$$

This set of equations is called the *normal equations*, for obvious reasons. Notice that the system matrix of the normal equations is symmetric, due to the commutative property of the scalar product. While the error equations in Eq. 3.1 are overdetermined and cannot be solved for $\vec{x}$, the normal equations are regular by design (here: 2 linearly independent equations for 2 unknowns, since $\text{rank}(\mathbf{A}) = 2$).
If we determine (x_1, x_2) such that the normal equations are fulfilled, then $|\vec{r}|_2 = \sqrt{r_1^2 + r_2^2 + r_3^2}$ is minimal. Therefore, if $|\vec{r}|_2$ is minimal, then also $|\vec{r}|_2^2 = r_1^2 + r_2^2 + r_3^2$ is minimal, hence solving the least-squares problem.

Now that we have a good understanding of the problem and its geometric interpretation, we can generalize to arbitrary n, m. Then, the error equation reads:

$$x_1\vec{a}^{(1)} + x_2\vec{a}^{(2)} + \cdots + x_n\vec{a}^{(n)} - \vec{c} = \vec{r},$$

where all vectors $\vec{a}^{(1)}$, $\vec{a}^{(2)}$, $\ldots$, $\vec{a}^{(n)}$, $\vec{c}$, and $\vec{r}$ are in $\mathbb{R}^m$ with $m > n$. In matrix notation, this is:

$$\mathbf{A}\vec{x} - \vec{c} = \vec{r}, \tag{3.2}$$

where $\mathbf{A} \in \mathbb{R}^{m\times n}$. The normal equations then are:

$$\mathbf{A}_*\vec{x} = \vec{b}, \tag{3.3}$$

with $\mathbf{A}_* \in \mathbb{R}^{n\times n}$ and $\vec{b} \in \mathbb{R}^n$ as follows:

$$\mathbf{A}_* = \mathbf{A}^\mathsf{T}\mathbf{A} = \begin{bmatrix} \langle \vec{a}^{(1)}, \vec{a}^{(1)}\rangle & \ldots & \langle \vec{a}^{(1)}, \vec{a}^{(n)}\rangle \\ \vdots & \ddots & \vdots \\ \langle \vec{a}^{(n)}, \vec{a}^{(1)}\rangle & \ldots & \langle \vec{a}^{(n)}, \vec{a}^{(n)}\rangle \end{bmatrix}, \qquad \vec{b} = \mathbf{A}^\mathsf{T}\vec{c} = \begin{bmatrix} \langle \vec{a}^{(1)}, \vec{c}\rangle \\ \vdots \\ \langle \vec{a}^{(n)}, \vec{c}\rangle \end{bmatrix}.$$

3.2 Solution by QR Decomposition

While the normal equations can in principle be directly solved, the symmetric coefficient matrix $\mathbf{A}_*$ of the normal equations is unfortunately badly conditioned. This makes solving the normal equations numerically inaccurate, and a better algorithm is needed. While the normal equations are great for theoretical considerations, we go back to the error equations for deriving a stable numerical algorithm.
We start from the fact that the length of any vector remains unchanged upon multiplication with an orthogonal matrix. An orthogonal matrix is a matrix for which $\mathbf{Q}\mathbf{Q}^\mathsf{T} = \mathbf{1} = \mathbf{Q}^\mathsf{T}\mathbf{Q}$.

Proof. $|\mathbf{Q}\vec{v}|^2 = \langle \mathbf{Q}\vec{v}, \mathbf{Q}\vec{v}\rangle = (\mathbf{Q}\vec{v})^\mathsf{T}\mathbf{Q}\vec{v} = \vec{v}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{Q}\vec{v} = \vec{v}^\mathsf{T}\vec{v} = \langle \vec{v}, \vec{v}\rangle = |\vec{v}|^2$. □

Multiplying the error equation with an orthogonal matrix $\mathbf{Q}^\mathsf{T} \in \mathbb{R}^{m\times m}$ from the left, we obtain the equivalent system:

$$\mathbf{Q}^\mathsf{T}\mathbf{A}\vec{x} - \mathbf{Q}^\mathsf{T}\vec{c} = \mathbf{Q}^\mathsf{T}\vec{r} =: \vec{s}. \tag{3.4}$$

This system of equations is equivalent to the original error equation in Eq. 3.2 in the sense that $|\vec{r}|_2 = |\vec{s}|_2$ due to the above-mentioned property of orthogonal matrices. Therefore, any solution that minimizes $|\vec{s}|_2$ also minimizes the residual of the original problem. But which $\mathbf{Q}$ to use? The following theorem helps:

Theorem 3.1. *If the column vectors $\vec{a}^{(i)}$ of a real $m \times n$ matrix $\mathbf{A}$, $m \geq n$, are linearly independent, then there exists an orthogonal $m \times m$ matrix $\mathbf{Q}$ and a regular $n \times n$ upper-triangular matrix $\mathbf{R_0}$, such that*

$$\mathbf{A} = \mathbf{QR} \quad \textit{with} \quad \mathbf{R} = \begin{bmatrix} \mathbf{R_0} \\ --- \\ \mathbf{0} \end{bmatrix} \in \mathbb{R}^{m\times n},$$

where $\mathbf{0}$ is a $(m-n)\times n$ zero matrix. This defines the QR decomposition *of $\mathbf{A}$.*

Using the matrix $\mathbf{Q}$ of the QR decomposition of $\mathbf{A}$, the transformed error equations become:

$$\mathbf{R}\vec{x} - \vec{d} = \vec{s} \qquad \text{with } \vec{d} \coloneqq \mathbf{Q}^\mathsf{T}\vec{c},$$

because $\mathbf{Q}^\mathsf{T}\mathbf{A} = \mathbf{Q}^\mathsf{T}\mathbf{QR} = \mathbf{R}$ due to the orthogonality of $\mathbf{Q}$. The structure of this system is depicted in Fig. 3.2.

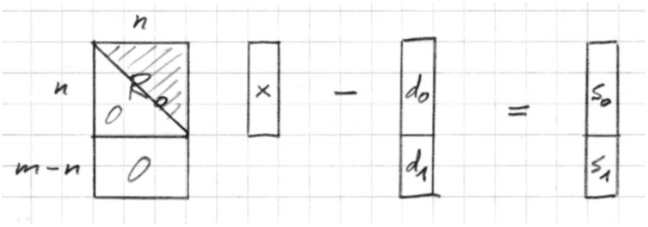


Figure 3.2: Structure of the transformed error equations after QR decomposition of $\mathbf{A}$.

According to this structure, the system of equations splits into the two parts:

$$\begin{aligned} \mathbf{R_0}\vec{x} - \vec{d_0} &= \vec{s_0} \\ -\vec{d_1} &= \vec{s_1}\,. \end{aligned} \tag{3.5}$$

The second equation is independent of $\vec{x}$. Therefore, minimizing the residual is solely achieved by determining $\vec{x}$ such that $\vec{s_0}$ is minimized. The first $n \times n$

system is regular (due to theorem 3.1 asserting that $\mathbf{R}_0$ is regular) and can therefore be uniquely solved. Since $\vec{s}_1$ is independent of $\vec{x}$, $|\vec{s}|_2$ is minimal if $\vec{s_0} = \vec{0}$. Therefore, we determine $\vec{x}$ such that $\mathbf{R_0}\vec{x} = \vec{d_0}$, which is easy to solve by backward substitution, since $\mathbf{R_0}$ is an upper-triangular matrix. This provides a numerically robust and computationally efficient ($O(n^2)$) algorithm for solving the linear least-squares problem. This way of solving the problem is numerically much more accurate than solving the normal equations, because $\kappa_{\mathbf{R_0}} = \kappa_{\mathbf{A}}$ (since $\mathbf{Q}$ is orthogonal), but $\kappa_{\mathbf{A}^\intercal\mathbf{A}} = \kappa_{\mathbf{A}}^2$.
The question of how to compute the QR decomposition of the matrix $\mathbf{A}$ is not treated here. There are established standard algorithms for doing this, available in any scientific computing software package, that can simply be used. Most implementations of QR decomposition are based on Givens rotations, which is a numerically robust and stable procedure. We refer to the literature for details.

Image: SIAM
James Wallace Givens Jr.
∗ 14 December 1910
Alberene, Virginia, USA
† 5 March 1993
USA

Algorithm 5 Direct Least-Squares Method

1: **procedure** LEASTSQUARES($\mathbf{A}$,$\vec{c}$) ▷ model $\mathbf{A}$, data $\vec{c}$
2: $\mathbf{R} = \mathbf{Q}^\intercal\mathbf{A}$ ▷ QR decomposition of $\mathbf{A}$
3: $\vec{d} = \mathbf{Q}^\intercal\vec{c}$ ▷ Simultaneously constructed
4: Solve $\mathbf{R_0}\vec{x} = \vec{d_0}$ for $\vec{x}$ by backward substitution ▷ see Fig. 3.2
5: **end procedure**

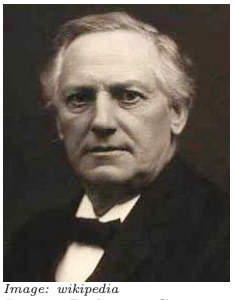


Image: wikipedia
Jørgen Pedersen Gram
∗ 27 June 1850
Nustrup, Denmark
† 29 April 1916
Copenhagen, Denmark

Note that the matrix $\mathbf{Q}$ never needs to be stored, as $\vec{d_0}$ can be constructed simultaneously with QR decomposition. Therefore, only the upper-triangular block $\mathbf{R}_0$ of $\mathbf{R}$ needs to be stored, resulting in a memory-efficient algorithm. $\mathbf{Q}$ is only explicitly required if the values of the final residuals $\vec{r} = \mathbf{Q}\vec{s}$ (because $\vec{s} = \mathbf{Q}^\intercal\vec{r}$) need to be computed.

Image: wikipedia
Erhard Schmidt
∗ 13 January 1876
Tartu, Livonia, Russian Empire (now: Estonia)
† 6 December 1959
Berlin, Germany

Gram-Schmidt Transformation using QR Decomposition

The Gram-Schmidt process (published by Gram and Schmidt in 1883) is an algorithm for converting a set of vectors into an orthonormal basis that spans the same space as the original vectors. It, too, can be efficiently solved by QR decomposition. In this case, the column vectors

$$\vec{a}^{(1)}, \ldots, \vec{a}^{(n)}$$

of a matrix $\mathbf{A} = (\vec{a}^{(1)}, \ldots, \vec{a}^{(n)}) \in \mathbb{R}^{n\times n}$ span the n-dimensional space $\mathbb{R}^n$, but may not be an orthonormal basis of this space (i.e., they may have lengths other than 1 and may not be pairwise orthogonal). Using the Gram-Schmidt process, the vectors $\vec{a}^{(i)}$, $i = 1, \ldots, n$, can be converted into an equivalent orthonormal basis, which can for example be used to define a Cartesian coordinate system. Since the matrix is square in this case, the QR decomposition

$$\mathbf{A} = \mathbf{QR}$$

has $\mathbf{Q}^\mathsf{T}\mathbf{Q} = \mathbf{1}_n$ and $\mathbf{R}$ is a regular upper-triangular matrix. Then, the column vectors of $\mathbf{Q} = (\vec{q}^{(1)}, \ldots, \vec{q}^{(n)})$ are an orthonormal basis of $\mathbb{R}^n$, and any subspace spanned by a partial set of this orthonormal basis is identical to the subspace spanned by the corresponding partial set of original vectors, i.e., $\text{span}(\vec{q}^{(1)}, \ldots, \vec{q}^{(j)}) = \text{span}(\vec{a}^{(1)}, \ldots, \vec{a}^{(j)})\ \forall j = 1, 2, \ldots, n$. This provides another example of the usefulness of QR decomposition.

3.3 Singular Value Decomposition

Matrix decompositions are a central concept in numerical computing, as exemplified by the LU and QR decompositions so far. Another important decomposition is the singular value decomposition (SVD). SVDs are central to control theory, signal processing, image processing, machine learning, coding theory, and many other applications.

Theorem 3.2. *Every real $m \times n$ matrix $\mathbf{A}$ of rank $k \leq n$ can be decomposed into an orthogonal $m \times m$ matrix $\mathbf{U}$ and an orthogonal $n \times n$ matrix $\mathbf{V}$, such that:*

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^\mathsf{T}$$

with

$$\mathbf{S} = \begin{cases} \begin{bmatrix} \hat{\mathbf{S}} \\ - - - \\ \mathbf{0} \end{bmatrix} & \text{if } m \geq n \\ \\ \begin{bmatrix} \hat{\mathbf{S}} & | & \mathbf{0} \end{bmatrix} & \text{if } m < n\,, \end{cases}$$

where $\hat{\mathbf{S}}$ is a diagonal matrix of dimension $p = \min(m, n)$ with diagonal elements $s_1 \geq s_2 \geq \ldots \geq s_k > s_{k+1} = \ldots = s_p = 0$. The s_i are called the singular values *of $\mathbf{A}$, the column vectors $\vec{u}^{(i)}$ of $\mathbf{U}$ are the* left singular vectors*, and the column vectors $\vec{v}^{(i)}$ of $\mathbf{V}$ are the* right singular vectors *of $\mathbf{A}$. The s_i^2 are the eigenvalues of $\mathbf{A}^\mathsf{T}\mathbf{A}$ if $m \geq n$, or of $\mathbf{A}\mathbf{A}^\mathsf{T}$ if $m < n$. Therefore, they are uniquely determined and*

$$\begin{aligned} \mathbf{A}\vec{v}^{(i)} &= s_i \vec{u}^{(i)} \qquad i = 1, 2, \ldots, p \\ \mathbf{A}^\mathsf{T}\vec{u}^{(i)} &= s_i \vec{v}^{(i)} \qquad i = 1, 2, \ldots, p\,. \end{aligned}$$

Computing the singular value decomposition is iterative and infinite. Finite-precision approximations, however, can be computed in finitely many iterations.

3.3.1 Properties of the singular value decomposition

The singular values and singular vectors provide an analogous concept for rectangular matrices as eigenvalues and eigenvectors do for square matrices. The following important properties of the singular value decomposition are frequently exploited in applications:

1. The kernel (also called *null space*) of the matrix $\mathbf{A}$, $\ker(\mathbf{A})$, which is the space of all vectors mapped to $\vec{0}$ by $\mathbf{A}$, i.e., $\ker(\mathbf{A}) = \{\vec{b} : \mathbf{A}\vec{b} = \vec{0}\}$, is spanned by the columns $k+1$ to n of $\mathbf{V}$:

$$\ker(\mathbf{A}) = \text{span}(\vec{v}^{(k+1)}, \ldots, \vec{v}^{(n)}).$$

   Likewise, the image $\text{im}(\mathbf{A})$ is spanned by:

$$\text{im}(\mathbf{A}) = \text{span}(\vec{u}^{(1)}, \ldots, \vec{u}^{(k)}).$$

   (proofs not given.)

2. $\|\mathbf{A}\|_2 = s_1$.

   Proof. $s_1^2 = \mu_{\max}(\mathbf{A}^\mathsf{T}\mathbf{A}) = \|\mathbf{A}\|_2^2$. □

3. for $\mathbf{A} \in \mathbb{R}^{n\times n}$ regular (i.e., $k = n$): $\|\mathbf{A}\|_2 \cdot \left\|\mathbf{A}^{-1}\right\|_2 = \frac{s_1}{s_n} = \kappa_\mathbf{A}$.

4. for $\mathbf{A} \in \mathbb{R}^{n\times n}$ symmetric: $s_i = |\lambda_i|$, $i = 1, 2, \ldots, n$, i.e., the singular values of the matrix are equal to the absolut values of the eigenvalues of the matrix.

   Proof. $s_i^2 = \mu_i(\mathbf{A}^\mathsf{T}\mathbf{A}) = \mu_i(\mathbf{A}^2) = \lambda_i^2$. □

5. $\mathbf{A} \in \mathbb{R}^{m\times n}$ defines a linear map from $V^n = \mathbb{R}^n$ to $V^m = \mathbb{R}^m$: $\vec{x} \longmapsto \vec{x'}$. Upon the coordinate transforms:

$$V^n : \vec{x} = \mathbf{V}\vec{y}$$
$$V^m : \vec{x'} = \mathbf{U}\vec{y'},$$

   the linear map is diagonal in the new coordinates $\vec{y}, \vec{y'}$, because: $\vec{x'} = \mathbf{A}\vec{x} = \mathbf{U}\mathbf{S}\mathbf{V}^\mathsf{T}\vec{x} = \mathbf{U}\vec{y'}$ and since $\vec{x} = \mathbf{V}\vec{y}$, we have $\vec{y'} = \mathbf{S}\vec{y}$. This diagonal map is defined by the singular value matrix $\mathbf{S} = \mathbf{U}^\mathsf{T}\mathbf{A}\mathbf{V} = \mathbf{B}$ (see Fig. 3.3). Therefore, the column vectors $\vec{v}^{(i)}$ of $\mathbf{V}$ and $\vec{u}^{(i)}$ of $\mathbf{U}$, together with the singular values s_i, completely describe the geometry of the linear map $\mathbf{A}$, in analogy to eigenvalues and eigenvectors for linear maps $\mathbb{R}^n \longrightarrow \mathbb{R}^n$.

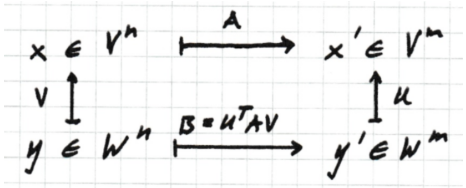


Figure 3.3: Schematic of the relation between a linear map $\mathbf{A}$ and its SVD.

3.4 Solution by Singular Value Decomposition

Singular value decomposition can be used to elegantly and efficiently solve linear least-squares problems. Consider again the error equation

$$\mathbf{A}\vec{x} - \vec{c} = \vec{r},$$

with $\mathbf{A} \in \mathbb{R}^{m \times n}$, $m > n$, and $\text{rank}(\mathbf{A}) = n$. Then, perform a singular value decomposition of $\mathbf{A}$ as $\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^\mathsf{T}$. Using the orthogonal matrices from the singular value decomposition, we can again, following the same argumentation as in Section 3.2, apply an orthogonal transformation to the error equations:

$$\underbrace{\mathbf{U}^\mathsf{T}\mathbf{A}}_{\mathbf{S}\mathbf{V}^\mathsf{T}}\vec{x} - \underbrace{\mathbf{U}^\mathsf{T}\vec{c}}_{\hat{\vec{c}}} = \underbrace{\mathbf{U}^\mathsf{T}\vec{r}}_{\hat{\vec{r}}},$$

with $|\hat{\vec{r}}|_2 = |\vec{r}|_2$ due to the orthogonality of $\mathbf{U}$. Using the coordinate transform $\vec{x} = \mathbf{V}\vec{y}$, the transformed error equation becomes

$$\mathbf{S}\vec{y} - \hat{\vec{c}} = \hat{\vec{r}}.$$

Due to the structure of $\mathbf{S}$ (see Theorem 3.2 for the case where $m > n$) this can again, similar to the QR decomposition, be split into two equations:

$$\begin{aligned} \hat{\mathbf{S}}\vec{y} - \hat{\vec{c}}_0 &= \hat{\vec{r}}_0 \\ -\hat{\vec{c}}_1 &= \hat{\vec{r}}_1\,, \end{aligned} \tag{3.6}$$

where $\hat{\mathbf{S}}$ is a regular (because $\text{rank}(\mathbf{A}) = n$) diagonal matrix. Again, like in the QR decomposition case, the length of the residual $|\vec{r}|_2$ is minimal when $\hat{\vec{r}}_0 = \vec{0}$, which is possible because the first linear system is regular. Because $\hat{\mathbf{S}}$ is a diagonal matrix, we immediately find the solution $\vec{y} = \hat{\mathbf{S}}^{-1}\hat{\vec{c}}_0$ as:

$$y_i = \frac{1}{s_i}\langle \vec{u}^{(i)}, \vec{c}\rangle, \quad i = 1, \ldots, n\,, \tag{3.7}$$

$$\vec{x} = \mathbf{V}\vec{y}. \tag{3.8}$$

This solution to the linear least-squares problem is very elegant and computationally efficient ($O(n^2)$ for the matrix-vector product in Eq. 3.8). It only requires n scalar divisions, n scalar products, and one matrix-vector multiplication.

This solution by singular value decomposition also suggests a natural recipe for how to deal with cases where $\mathbf{A}$ does not have full rank. In such cases, QR decomposition fails. If $\text{rank}(\mathbf{A}) = k < n$ in SVD, it simply means that $s_i = 0$ for $i > k$. Then, only those $y_i = \frac{1}{s_i}\langle \vec{u}^{(i)}, \vec{c}\rangle$ with $i = 1, \ldots, k$ are defined, and $\vec{x} = \mathbf{V}\vec{y}$ is no longer uniquely determined because $y_{k+1}, \ldots, y_n$ can be chosen arbitrarily. One possible choice from these infinitely many solutions is to choose the $\vec{y}$ of minimal length, i.e., $y_{k+1} = \ldots = y_n = 0$. Then, because $\mathbf{V}$ is orthogonal, also $\vec{x}$ has minimal length $\min_{\mathbf{A}\vec{x} - \vec{c} = \vec{r}} |\vec{x}|_2$, corresponding to an

ℓ^2-regularization of the problem, which is also called *Tikhonov regularization* in statistics or *ridge regression* in machine learning. This is a classic example of the more general concept of *regularization*, which makes undetermined problems solvable by assuming additional constraints on the solution.

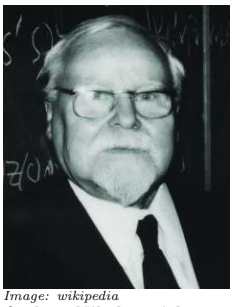


Image: wikipedia
Andrey Nikolayevich Tikhonov
∗ 17 October 1906
Gzhatsk, Russian Empire
† 7 October 1993
Moscow, Russia

3.5 Nonlinear Least-Squares Problems

So far, we have considered least-squares problems for linear models, where elegant and numerically robust algorithms exist. If the model is non-linear, i.e., if the model output is a nonlinear function $f(\cdot)$ of the model parameters $\vec{x}$, then the error equation reads:

$$\vec{f}(\vec{x}) - \vec{c} = \vec{r} \tag{3.9}$$

with $f_i(x_1, \ldots, x_n) - c_i = r_i$ for each $i = 1, \ldots, m \geq n$. Again, we want to find $\vec{x} \in \mathbb{R}^n$ such that $|\vec{r}|_2$ is minimized. This is the case if the scalar-valued function

$$S(\vec{x}) \coloneqq |\vec{r}|_2^2 = \vec{r}^{\mathsf{T}}\vec{r} = \sum_{i=1}^{m} \left(f_i(x_1, \ldots, x_n) - c_i\right)^2 \tag{3.10}$$

is minimized. A necessary condition for this is that the gradient of S vanishes, hence defining an extremal point (minimum, maximum, or saddle point) of S. Therefore, we require that

$$\frac{\partial S(\vec{x})}{\partial x_j} = 2\sum_{i=1}^{m} \left(f_i(x_1, \ldots, x_n) - c_i\right) \frac{\partial f_i(x_1, \ldots, x_n)}{\partial x_j} = 0\,, \quad \forall j = 1, \ldots, n. \tag{3.11}$$

This defines a system of n non-linear equations for the n unknowns $x_1, \ldots, x_n$, analogous to the normal equations of the linear case. While this system can in principle be solved, practical solutions may be difficult, in particular if the derivatives of the f_i are not known in analytical form.

One approach is the so-called Gauss-Newton method, which is based on expanding the non-linear system in Eq. 3.11 into a Taylor series and solving the linear system resulting from only using the first two terms of the expansion (i.e., constant and linear). The general topic of numerically approximating the solution of a non-linear equation is discussed in the next chapter.

Since the solutions of Eq. 3.11 include all extremal points of S, of which there are potentially many, one also requires a starting value close enough to the *global* minimum, and one needs to verify after the fact that the extremal point found is indeed a minimum (e.g., by computing the Hessian matrix, due to Ludwig Otto Hesse, around that point).

Image: wikipedia
Ludwig Otto Hesse
∗ 22 April 1811
Königsberg, Prussian Empire
(now: Kaliningrad, Russia)
† 4 August 1874
Munich, Kingdom Bavaria

Chapter 4

Solving Nonlinear Equations

Now that we know how to solve linear equations, both regular systems and least-squares fits, we consider the problem of solving a nonlinear equation, i.e., of finding a root x^* such that $y = f(x^*) = 0$ for some given nonlinear continuous function $f : \mathbb{R} \longrightarrow \mathbb{R}$. We first consider the scalar case and then generalize to systems of nonlinear equations in the next chapter.

While linear equations can often be solved analytically, and a complete and closed theory exists for linear systems of equations, this is not the case for nonlinear equations. Only few special cases of nonlinear equations can be solved analytically (e.g., quadratic and cubic equations). Numerical methods are therefore often the only approach to the problem in nonlinear cases. Most of these numerical methods require the function to be at least once continuously differentiable, and convergence proofs often assume that it is twice continuously differentiable. Some methods, in particular search-based ones, also work for more general continuous functions.

4.1 Condition Number of the Problem

If the nonlinear equation $f(x) = 0$ has at least one solution x^*, we can distinguish two cases, as illustrated in Fig. 4.1: (a) the function $y = f(x)$ intersects $y = 0$ at x^*; (b) the function $y = f(x)$ is tangential to $y = 0$ at x^*. If none of the two is the case for any x, then the equation has no solutions at all. Case (a) is characterized by the derivative of f being $f'(x^*) \neq 0$, whereas for case (b) we have $f'(x^*) = 0$. The two cases have different condition numbers.

Under the additional assumption that f is uniquely invertible in some interval around a given x^*, we can explicitly compute the condition number for case (a) by considering

$$x^* = f^{-1}(0) =: H(0).$$

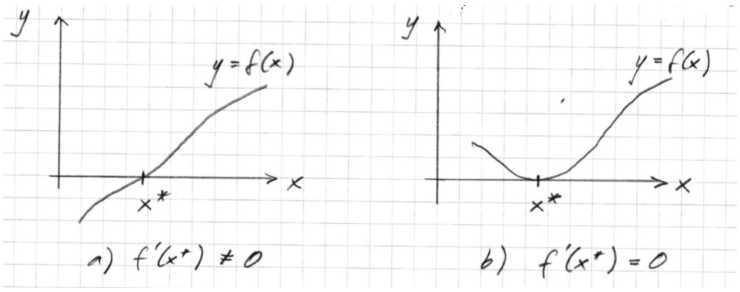


Figure 4.1: The two possibilities for $f(x^*) = 0$.

For computing the condition number, considering the relative error as we did in Section 1.5.2 makes no sense, because the value at which we evaluate H is 0, so the relative error is undefined. We instead consider the absolute error, therefore looking at the condition number as the amplification factor of the absolute error. For this, we find:

$$\kappa_H = \left|\frac{H'(0)}{H(0)}\right| = \left|\frac{1}{x^* f'(x^*)}\right| \tag{4.1}$$

because

$$\frac{\mathrm{d}}{\mathrm{d}z}\left[f^{-1}(f(z)) = z\right] \Longrightarrow (f^{-1})'(f(z))f'(z) = 1 \xRightarrow{z=x^*} (f^{-1})'(0) = \frac{1}{f'(x^*)}.$$

and $H(0) = x^*$ by construction. Note that the chain rule can be applied here in the interval around x^* where f is uniquely invertible, because H is a proper function there. From this, we see that for $|f'(x^*)| \ll 1$, the problem is ill-conditioned. In fact, for either $f'(x^*) = 0$ or $x^* = 0$, the condition number is infinite and the numerical error may be arbitrarily large.

This can be intuitively understood and is true also for functions that are not uniquely invertible around x^*. The case (b) where $f'(x^*) = 0$ corresponds to $f(x)$ having a minimum, maximum, or saddle point at the root. This means that the x-axis is touched rather than intersected. The closer one gets to the root, the more $y = f(x)$ looks similar to $y = 0$ and, eventually, there are infinitely many points where $|f(x)| < \hat{x}_{\min}$ and that are therefore numerically indistinguishable from zero. The actual root cannot be identified among them. In the worst case, the function $f(x) = 0$ for some $x \in [a, \infty)$, and the root could be anywhere. Therefore, the only upper bound one can give on the absolute error is ∞, which is accurately reflected in the above condition number. The case where $x^* = 0$ is similar, as one eventually hits $|x| < \hat{x}_{\min}$, again leaving infinitely many solutions that are numerically indistinguishable.

As pointed out in Example 1.8, it is impossible to find a good numerical algorithm for an ill-conditioned problem. Therefore, we only consider the well-conditioned case in the following, where accurate algorithms can be derived. From now on, we thus assume that $f'(x^*) \neq 0$ and $x^* \neq 0$. Note that the latter

assumption is not limiting, as any problem not meeting it can always be shifted by adding a constant in x such that the assumption is fulfilled.

4.2 Newton's Method

A classic method for solving nonlinear equations with analytically known derivatives is given by Newton's method. Let $f(x)$ be (at least once) continuously differentiable, and let x_0 be an initial point "close enough" (we will get back to what this means) to x^*.

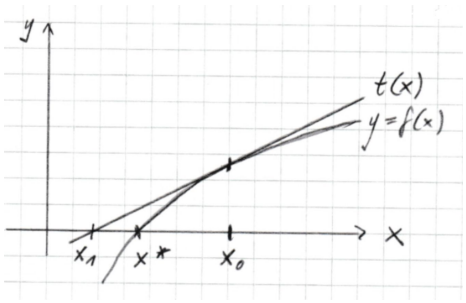


Figure 4.2: Illustration of the first iteration of Newton's method, advancing the solution from x_0 to x_1.

The idea behind Newton's method is to locally linearize the function $f(x)$, i.e., to compute the Taylor expansion (introduced by Brook Taylor in 1715) of $f(x)$ around the starting point x_0 and only keep terms up to and including the linear order, yielding:

$$t(x) = f(x_0) + f'(x_0)(x - x_0). \tag{4.2}$$

Geometrically, the linear approximation $t(x)$ is the equation of the tangent line to $f(x_0)$, see Fig. 4.2. This scalar linear equation can be solved analytically, yielding an approximation $x_1 \approx x_0$ where $t(x_1) = 0$:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} =: F(x_0). \tag{4.3}$$

Iterating this procedure should yield successively better approximations x_k to x^* by computing:

$$x_{k+1} = F(x_k) = x_k - \frac{f(x_k)}{f'(x_k)}, \qquad k = 0, 1, 2, \ldots. \tag{4.4}$$

Notice that the $f'(x)$ occurring in the denominator is a direct manifestation of the ill-conditioned nature of the case where $f'(x^*) = 0$, as discussed in Section 4.1. The Newton method diverges and becomes numerically unstable in this

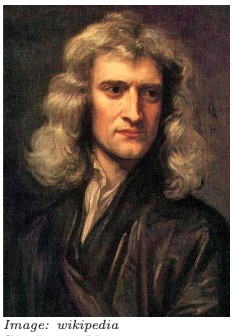


Image: wikipedia
Sir Isaac Newton
∗ 4 January 1643
Woolsthorpe-by-Colsterworth, England
† 31 March 1727
Kensington
Middlesex, England

Image: wikipedia
Brook Taylor
∗ 18 August 1685
Edmonton, Middlesex
England
† 29 December 1731
London, England

case. If the values $\{x_k\}$ of the above iteration converge for $k \to \infty$ to a value $\bar{x}$, then $\bar{x}$ is a solution of the *fixed-point equation*

$$\bar{x} = F(\bar{x})$$

and, due to Eq. 4.4, we have $f(\bar{x}) = 0$. Therefore, if a fixed point exists for this iteration, then this fixed point necessarily is a solution of the nonlinear equation in question. The iteration is therefore *always consistent*, and there is no need for an additional consistency condition like the one from Eq. 2.5 for the linear case. The iteration defined by Eq. 4.4 is called *Newton's method*, named after Sir Isaac Newton who wrote it down in 1669 in his book "De analysi per aequationes numero terminorum infinitas" and applied it to finding roots of polynomials.

4.2.1 The fixed-point theorem

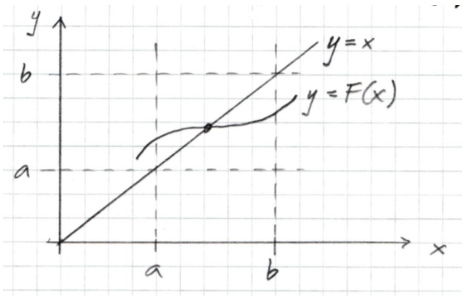


Figure 4.3: Geometric illustration of the solution of the fixed-point equation $x = F(x)$.

The only question that remains to be addressed is whether (or when) Eq. 4.4 converges to a fixed point, which is then automatically guaranteed to be a solution of the original equation. If $F(x)$ is continuous in an interval $I = [a, b]$ and F maps that interval onto itself, i.e. $F : I \to I$, then the solution $\bar{x} = F(\bar{x})$ is given by the point where the line $y = x$ intersects the graph of $y = F(x)$, as illustrated in Fig. 4.3. The answer under which conditions the iteration defined in Eq. 4.4 converges to this point is given by the *fixed-point theorem*:

Theorem 4.1 (Fixed-Point Theorem)**.** *If a continuous function $F : \mathbb{R} \to \mathbb{R}$ satisfies:*

i. There exists an interval $I = [a, b] \subset \mathbb{R}$ that gets mapped onto itself by F, i.e., $F : I \to I$ and,

ii. F is a contraction on this interval, i.e., $|F(x') - F(x)| \leq L|x' - x|$ $\forall x', x \in I$ with Lipschitz constant $0 < L < 1$*,*

then:

1. *the fixed-point equation* $F(x) = x$ *has a unique solution* $\bar{x} \in I$,
2. *the fixed-point iteration* $x_{k+1} = F(x_k)$ *converges to* $\bar{x}$ *for* $k \to \infty$ *starting from any* $x_0 \in I$, *and*
3. $|\bar{x} - x_k| \leq \frac{L^{k-j}}{1-L}|x_{j+1} - x_j|$ *for all* $0 \leq j \leq k$.

This theorem provides a nonlinear generalization of the fixed-point iteration from the linear case discussed in Section 2.2, where $\rho(\mathbf{T}) < 1$ ensured that the linear map is a contraction (remember that the spectral radius is the largest stretching factor applies to any vector by the map). Note that precondition (ii) is always fulfilled if $|F'(x)| \leq L < 1 \ \forall x \in I$. Indeed, in the scalar case, the Lipschitz constant L is an *upper bound* on the absolute value of the first derivative. This is illustrated in Fig. 4.4. In both panels of the figure, the function $y = F(x)$ maps the interval $[a, b]$ onto itself, i.e., all y are between a and b for any x between a and b. However, in the left panel $|F'(x)| < 1$ everywhere in $[a, b]$, whereas in the right panel there exist points for which $|F'(x)| > 1$. As is easily visualized geometrically, the left case amounts to the fixed-point iteration being "attracted" to $\bar{x}$, whereas in the right case it spirals away from the solution, as claimed by the theorem.

Image: wikipedia
Rudolf Lipschitz
∗ 4 May 1832
Königsberg, Prussia
(now: Kaliningrad, Russia)
† 7 October 1903
Bonn, German Empire

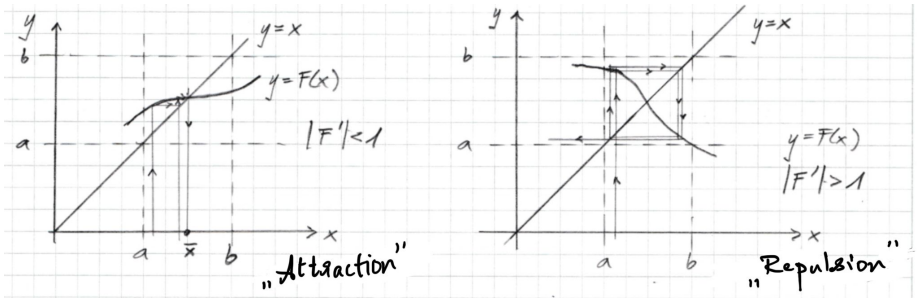


Figure 4.4: Illustration of the fixed-point theorem. The left panel illustrates the situation where $F(x)$ is a contraction and the fixed-point iteration converges, whereas the right panel illustrates the case where both are not the case.

Proof. We prove points (1) and (2) of the theorem, but omit the proof for point (3).

1. By assumption, $F(x)$ is Lipschitz continuous (Rudolf Lipschitz) and therefore $g(x) = F(x) - x$ is also continuous. The requirement that F maps the interval $[a, b]$ onto itself implies that $F(a) \geq a$ and $F(b) \leq b$ and therefore $g(a) \geq 0$, $g(b) \leq 0$.
   $\Longrightarrow \exists x^* \in [a, b]$ with $g(x^*) = 0$, because g is continuous.
   $\Longrightarrow$ there exists at least one fixed point.

We show by contradiction that for $L < 1$ there is exactly one fixed point. Assume there is another $x^{**} \neq x^*$ with $F(x^{**}) = x^{**}$. Then:

$$0 < |x^* - x^{**}| = |F(x^*) - F(x^{**})| \leq L|x^* - x^{**}| < |x^* - x^{**}|,$$

which is a contradiction, since a number > 0 cannot be strictly smaller than itself. Therefore, there exists exactly one, unique fixed point.

2. Assume an arbitrary starting point $x_0 \in [a, b]$. Then, all $x_k = F(x_{k-1})$ for $k = 1, 2, 3, \ldots$ are also $\in [a, b]$, and:

$$\begin{aligned} |x^* - x_k| = |F(x^*) - F(x_{k-1})| \leq L|x^* - x_{k-1}| \leq L^2|x^* - x_{k-2}| \leq \ldots \\ \leq L^k|x^* - x_0|. \end{aligned}$$

Since $L < 1$, $L^k \to 0$ for $k \to \infty$. Therefore, $x_k \to x^*$ for $k \to \infty$, for all $x_0 \in [a, b]$.

□

From Eq. 4.4, the iteration function for the Newton method is

$$F(x) = x - \frac{f(x)}{f'(x)}.$$

Note that this is not the only possible fixed-point iteration for solving nonlinear equations, as illustrated in the following example.

Example 4.1. *It is easy to see that the fixed-point iteration $F(x) = f(x) + x$ also converges to the solution of the equation $f(x^*) = 0$ if the prerequisites of the fixed-point theorem are fulfilled, because*

$$f(x^*) = 0 \Longrightarrow \underbrace{f(x^*) + x^*}_{F(x^*)} = x^*.$$

The question is under which conditions the prerequisites of Theorem 4.1 are fulfilled, i.e., when $|F'(x)| < 1$ for this iteration function. We find: $F'(x) = f'(x) + 1$. So, this is only a contraction for $f'(x) \in (-2, 0)$. This iteration thus only converges for monotonically decreasing f with slope less than 2, which is a very limited set of functions. Moreover, this iteration converges slowly (linearly in the Lipschitz constant L, as guaranteed by Theorem 4.1), whereas Newton is much faster, as we will see below.

Knowing the fixed-point theorem, we can come back to the question of how close x_0 needs to be to x^* in order for the Newton iteration to converge to $\bar{x} = x^*$. Assuming again that $f'(x^*) \neq 0$ (see Section 4.1), we have:

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

This means that the prerequisites of Theorem 4.1 are fulfilled for a, b such that $x^* \in [a, b]$ and $|b - a|$ small enough such that $|F'(x)| < 1$ everywhere in $[a, b]$. Therefore, the more curved the function is (i.e., the larger $|f''(x)|$), the closer x_0 has to be to x^*. For $|b-a| \to 0$, the function is locally flat (i.e., $|f''(x)| \to 0$) so that $|F'(x)| \to 0$. Therefore, the Newton method always converges for a close-enough starting point. Moreover, because $F'(x^*) = 0$, Newton's method converges fast. If $F'(x) = 0$ everywhere in $[a, b]$, the iteration converges in one step. Since $F(x)$ is continuous and $F'(x) < 1$, the Newton iteration continually speeds up as it approaches its fixed point.

4.2.2 Convergence of Newton's method

How fast does Newton's method converge precisely? In order to analyze this, we look at the absolute error $e_k := x_k - x^*$ at iteration k. For Newton's method, under the assumption that f is (at least) twice continuously differentiable, one finds (proof not given here):

$$e_{k+1} \propto \frac{f''(x^*)}{2f'(x^*)} e_k^2. \tag{4.5}$$

This means that for $f'(x^*) \neq 0$, Newton converges (at least) quadratically if $f'' < 2f'$ (see above "close enough" discussion), i.e., the number of correct digits doubles in each iteration. This fast convergence means that few iterations of the method are sufficient in practice to compute the result to machine precision. In general, we define:

Definition 4.1 (Order of convergence)**.** *If $e_{k+1} \propto C e_k^p$ for a constant $|C| < 1$, then p is called the* order of convergence *and C is the convergence factor.*

For the general fixed-point iteration, we find:

$$\begin{aligned} x_k &= F(x_{k-1}) \\ x^* &= F(x^*) \\ \implies |\underbrace{x_k - x^*}_{e_k}| &= |F(x_{k-1}) - F(x^*)| \leq \underbrace{L}_{<1} |\underbrace{x_{k-1} - x^*}_{e_{k-1}}|, \end{aligned}$$

implying linear convergence (exponent $p = 1$) with pre-factor $0 < L < 1$. Therefore, while every fixed-point iteration converges at least (note the $\leq$ above!) linearly, the specific choice of $F(x)$ in the Newton method is responsible for the fact that Newton converges faster than any arbitrary fixed-point iteration. In fact, using Definition 4.1, we find that for the Newton method $L = \frac{f''(x^*)}{2f'(x^*)} e_{k-1}$, such that the fast convergence can be explained by the pre-factor depending on the error, which is a direct consequence of $F'(x^*) = 0$ for Newton's method.

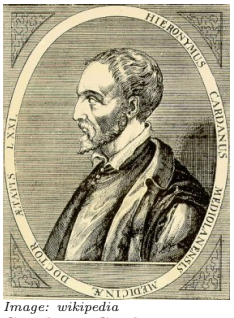


Image: wikipedia
Gerolamo Cardano
(Hieronymus Cardanus)
∗ 24 September 1501
Pavia (now in Italy)
† 21 September 1576
Rome (now in Italy)

4.2.3 Algorithm

In a practical implementation of Newton's method, the user needs to specify the starting point x_0 as well as the relative tolerance RTOL and the absolute tolerance ATOL. Then, the algorithm is given by Algorithm 6.

Algorithm 6 Newton Method

```
1: procedure NewtonMethod(x_0, RTOL, ATOL)  ▷ start point, tolerance
2:     k = −1
3:     repeat
4:         k = k + 1
5:         x_{k+1} = x_k − f(x_k)/f'(x_k)                    ▷ Eq. 4.4
6:     until |x_{k+1} − x_k| ≤ |x_{k+1}|RTOL + ATOL
7: end procedure
```

A key advantage of the Newton method is its fast convergence. Disadvantages are that it is a local method (i.e., x_0 must be sufficiently close to x^*) and that the derivative $f'(x)$ must exist and be analytically known at arbitrary points. In applications where this is the case, Newton is the best choice known. However, the second disadvantage is often limiting in practice, since for many numerical problems, particularly when f is given by data, the derivative may not be known everywhere.

4.3 Secant Method

The secant method addresses the second disadvantage of Newton's method, as it does not require analytically knowing $f'(x)$ (but it must still exist). It can be interpreted as a version of Newton's method that uses a finite difference to numerically approximate $f'(x)$. However, the secant method was discovered over 3000 years before Newton's method, in 18$^{\text{th}}$ century B.C. Babylon (now: Hillah, Babil, Iraq). It was first used to solve nonlinear equations by Gerolamo Cardano in 1545 in his book "Artis Magnae" (source Papakonstantinou, J. (2009), Historical development of the secant method and its characterization properties). The idea behind the secant method is to start from two approximations x_0, x_1 and iterate toward the solution x^* where $f(x^*) = 0$ by locally replacing $f(x)$ with the secant through $(x_0, f(x_0))$ and $(x_1, f(x_1))$, instead of the tangent as in the Newton method. The root of the secant defines the next point x_2 of the iteration (see Fig. 4.5). This method does not require the derivative of f.
The secant is the line defined by the function:

$$y = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1) + f(x_1),$$

which has the unique root

$$x_2 = \underbrace{x_1}_{\to x^* \in O(1)} - \frac{x_1 - x_0}{f(x_1) - f(x_0)} \underbrace{f(x_1)}_{\to 0}.$$

The problem with this formula, however, is that it suffers from increasing numerical extinction the closer x_k gets to x^*. This is because the last factor $f(x_k) \to 0$

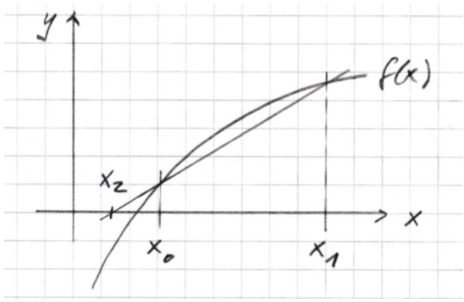


Figure 4.5: Illustration of the first iteration of the secant method.

as $x_k \to x^*$. Therefore, the formula subtracts increasingly small numbers from the first summand $x_k \in O(1)$ (since by assumption $x^* \neq 0$), losing significant digits. A numerically better (but algebraically identical) way of computing the new root is:

$$x_2 = \frac{x_0 f(x_1) - x_1 f(x_0)}{f(x_1) - f(x_0)}, \tag{4.6}$$

which has a much lower condition number as all terms go to zero together for $x_k \to x^*$. The resulting secant method is given in Algorithm 7.

Algorithm 7 Secant Method

```
1: procedure SecantMethod(x_0, x_1, RTOL, ATOL)
2:     k = 0
3:     repeat
4:         k = k + 1
5:         x_{k+1} = (x_{k-1} f(x_k) − x_k f(x_{k-1})) / (f(x_k) − f(x_{k-1}))   ▷ Eq. 4.6
6:     until |x_{k+1} − x_k| ≤ |x_{k+1}|RTOL + ATOL
7: end procedure
```

For x_0, x_1 close enough to x^*, the secant method converges with convergence order $p \approx 1.6$. The main advantage of the secant method is that it does not require derivatives of the function f and therefore is computationally cheap and also applicable to cases where derivatives are not available. The disadvantage is that it does not converge quite as fast as Newton's method.

Image: wikipedia
Bernardus Placidus Johann Nepomuk Gonzal Bolzano
∗ 5 October 1781
Prague, Bohemia
† 18 December 1848
Prague, Bohemia
(now: Czech Republic)

4.4 Bisection Method

The other disadvantage of the Newton method, its locality, is addressed by the bisection method, also called *bisection search*. The bisection method was developed by Bernardus Bolzano in 1817 (source: Edwards, C. H. (1979). Bolzano, Cauchy, and Continuity. The Historical Development of the Calculus, pp. 308). The idea behind the bisection method is to iteratively search for a root in the

entire interval. This removes the requirement for f to be (at least once) differentiable, and the problem of choosing suitable starting points does not exist. However, the bisections method is only guaranteed to exactly hit a root when used with finite-precision arithmetics. Also, the search interval needs to be defined beforehand and must contain (at least one) root.

Let $f(x)$ be continuous in the interval $I = [a, b]$ with $f(a)f(b) < 0$. Then, $f(x)$ has at least one root (in fact, any odd number of roots) in I (see Fig. 4.6).

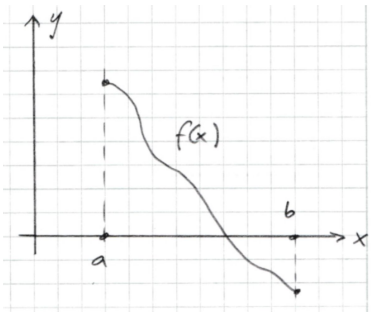


Figure 4.6: Illustration of the idea behind the bisection method.

Algorithm 8 Bisection Method

```
1: procedure BisectionMethod(a, b, ATOL)        ▷ Start interval I = [a, b]
2:     if f(a)f(b) > 0 then
3:         return error
4:     end if
5:     loop
6:         m = 1/2(a + b)
7:         if |f(m)| < ATOL then
8:             return m                                  ▷ m is a root
9:         else if f(a)f(m) < 0 then
10:            b = m
11:        else
12:            a = m
13:        end if
14:    end loop
15: end procedure
```

The bisection method starts with the entire interval and recursively cuts it in the middle with the goal of homing in on a root. The algorithm is given in Algorithm 8.

This results in a sequence of Intervals $I > I_1 > I_2 > \ldots$, for which we have:

- $\text{Length}(I_{j+1}) = \frac{1}{2}\text{Length}(I_j)$,

- There is exactly one point that belongs to all intervals, i.e., $x^* \in I \cap I_1 \cap I_2 \cap \ldots$,
- x^* is a root of f, i.e., $f(x^*) = 0$.

Obviously, the bisection method finds *one* root globally in I. Recursing the bisection method as a tree finds multiple roots. There is no guarantee, however, that all roots are found in either case. This is because while $f(a)f(b) < 0$ is sufficient for the existence of at least one root, it is not necessary. If f has an even number of roots in I, then $f(a)f(b) > 0$. No guarantees can therefore be given for the bisection method to find all roots.
Studying the convergence of the bisection method is not straightforward, because it generates a sequence of intervals rather than a sequence of points. One approach is to define the center point x_k of interval I_k as the k-th approximation, thereby creating a sequence of points for which it can be shown that

$$|x_k - x^*| \leq \frac{1}{2^{k+1}}(b-a),$$

therefore,

$$e_{k+1} \propto qe_k \text{ with } |q| < \frac{1}{2}.$$

This means that the bisection method converges linearly with pre-factor at most $1/2$, because the intervals are halved successively. The big advantage over Newton's method and over the secant method is that the bisection method *always* converges, because it is a global method that requires no "close enough" starting point.
It is also popular to combine the bisection method with the Newton (if derivatives are available) or the secant (if derivatives are not available) method. The goal is to benefit from the global nature of bisection *and* from the faster convergence of Newton or bisection. The idea is to first apply bisection to find intervals that contain (ideally one) root(s) and then start independent Newton or secant iterations in each of these intervals. For this combined method, however, convergence can not be guaranteed, as the preconditions of the fixed-point theorem need not be fulfilled in all intervals. Also, due to the properties of the bisection method, there is no guarantee that all roots are found.

Chapter 5

Nonlinear System Solvers

Now that we know how to solve scalar nonlinear equations, we generalize to the case of systems of nonlinear equations, like the one encountered in Eq. 3.11. While scalar nonlinear equations can be analytically solved in some special cases (notably quadratic and cubic equations), analytical solutions of systems of nonlinear equations are almost never possible, not even in the quadratic case. Numerical methods therefore enjoy great importance in this field.
The problem is stated as finding a vector $\vec{x} \in \mathbb{R}^n$ for for which

$$\vec{f}(\vec{x}) = \vec{0}\,, \quad \text{with } \vec{f}(\vec{x}) = \begin{bmatrix} f_1(x_1, \ldots, x_n) \\ \vdots \\ f_n(x_1, \ldots, x_n) \end{bmatrix}. \tag{5.1}$$

5.1 Newton's Method in Arbitrary Dimensions

Since a system of n nonlinear equations can be interpreted as a nonlinear equation in n dimensions (i.e., over n-dimensional vectors), we first generalize Newton's method from 1D (see Sec. 4.2) to nD. For this, we need to find a n-dimensional generalization of the first derivative of f. For this to exist, we again assume, like in Sec. 4.2, that the functions f_i, $i = 1, \ldots, n$, are differentiable. Then, we can define the n-dimensional generalization of the first derivative, as:

Definition 5.1 (Jacobian)**.** *The* Jacobian *or* Jacobi matrix *of* $\vec{f}(\vec{x}) : \mathbb{R}^n \to \mathbb{R}^n$ *is the matrix of all possible partial derivatives:*

$$\frac{\partial \vec{f}}{\partial \vec{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} = \mathbf{J}(\vec{x}).$$

In 1D, the Jacobian corresponds to the first derivative of the function. There, with $f'(x_0) = J(x_0)$, linearizing the function around a given point x_0 leads to

(see Eq. 4.2):

$$f(x) \approx f(x_0) + J(x_0)(x - x_0),$$

which is the Taylor expansion of f around x_0 keeping only terms up to and including the linear order. It was the basis of the Newton method that the root of this linear equation can easily be found as:

$$x_1 = x_0 - J^{-1}(x_0)f(x_0).$$

By the analogy $f' \Longleftrightarrow \mathbf{J}$, $\frac{1}{f'} \Longleftrightarrow \mathbf{J}^{-1}$, this is the Newton method.
This immediately suggests a generalization to arbitrary dimensions by defining the fixed-point iteration

$$\begin{aligned} \vec{x}_{k+1} &= \vec{F}(\vec{x}_k), \quad k = 0, 1, 2, \dots \\ \text{with } \vec{F}(\vec{x}) &= \vec{x} - \mathbf{J}^{-1}(\vec{x})\vec{f}(\vec{x}). \end{aligned} \tag{5.2}$$

At the fixed point $\vec{x}^* = \vec{F}(\vec{x}^*)$, we have by construction that $\vec{f}(\vec{x}^*) = 0$ if $\det(\mathbf{J}(\vec{x}^*)) \neq 0$, i.e., if the Jacobian is regular in therefore its inverse exists.
The question when the iteration defined in Eq. 5.2 converges to this fixed point can be answered by generalizing Theorem 4.1 to arbitrary dimensions (Stefan Banach, 1922):

Theorem 5.1 (Banach Fixed-Point Theorem)**.** *Let a continuous function $\vec{F} : D \to D$, $D \subset \mathbb{R}^n$, be a contraction, i.e., there exist $0 < L < 1$ and $|\cdot|$, so that $\forall(\vec{x}_a, \vec{x}_b) \in D$, $|\vec{F}(\vec{x}_a) - \vec{F}(\vec{x}_b)| \leq L|\vec{x}_a - \vec{x}_b|$. Then:*

1. *there exists exactly one fixed point $\vec{x}^* \in D$ with $\vec{x}^* = \vec{F}(\vec{x}^*)$,*
2. *the iteration $\vec{x}_{k+1} = \vec{F}(\vec{x}_k)$ converges for $k \to \infty$ to $\vec{x}^*$ $\forall \vec{x}_0 \in D$, and*
3. $|\vec{x}^* - \vec{x}_k| \leq \frac{L^k}{1-L}|\vec{x}_1 - \vec{x}_0|$

   $|\vec{x}^* - \vec{x}_k| \leq \frac{L}{1-L}|\vec{x}_k - \vec{x}_{k-1}|$.

The overall algorithm for Newton's method in n dimensions is then given in Algorithm 9. It requires solving a linear system of equations in each iteration, typically using LU decomposition because convergence of iterative methods cannot be guaranteed for arbitrary Jacobians. Therefore, a linear system solver is a prerequisite as a building block for nonlinear system solvers.
The update step in lines 5 and 6 of Algorithm 9 directly follows from Eq. 5.2 as:

$$\begin{aligned} \vec{x}_{k+1} &= \vec{x}_k - \mathbf{J}^{-1}(\vec{x}_k)\vec{f}(\vec{x}_k) \\ \mathbf{J}(\vec{x}_k)\underbrace{(\vec{x}_{k+1} - \vec{x}_k)}_{\vec{\Delta}_k} &= -\vec{f}(\vec{x}_k) \\ &\Longrightarrow \vec{x}_{k+1} = \vec{x}_k + \vec{\Delta}_k. \end{aligned}$$

Algorithm 9 n-Dimensional Newton Method

1: **procedure** NEWTON-ND($\vec{x}_0$, ATOL, RTOL) ▷ Start vector $\vec{x}_0$
2: $k = -1$
3: **repeat**
4: $k = k + 1$
5: Solve $\mathbf{J}(\vec{x}_k)\vec{\Delta}_k = -\vec{f}(\vec{x}_k)$ for $\vec{\Delta}_k$
6: $\vec{x}_{k+1} = \vec{x}_k + \vec{\Delta}_k$
7: **until** $|\vec{\Delta}_k| \leq |\vec{x}_{k+1}|\text{RTOL} + \text{ATOL}$
8: **end procedure**

While the Newton method in n dimensions inherits the fast convergence from its 1D counterpart (it converges quadratically if $\vec{x}_0$ is close enough to $\vec{x}^*$ and $\mathbf{J}$ is regular everywhere along the path), it also has three important drawbacks: First, finding a starting point $\vec{x}_0$ that is sufficiently close to the solution is much harder in n dimensions than it is in 1D. Moreover, the question what "close enough" means is much more difficult to answer in n dimensions. Second, the Jacobian $\mathbf{J}(\vec{x}_k)$ needs to be completely recomputed in each iteration, requiring n^2 derivatives. Third, the method is computationally expensive requiring $O(n^3)$ operations *in each iteration*.

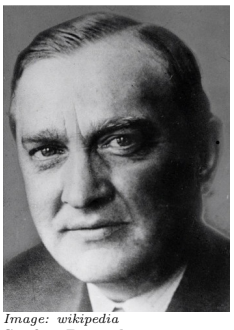


Image: wikipedia
Stefan Banach
∗ 30 March 1892
Kraków, Austria-Hungary
(now: Poland)
† 31 August 1945
Lviv, Soviet Union USSR
(now: Ukraine)

5.1.1 Quasi-Newton method

The Quasi-Newton method improves the computational cost by computing the Jacobian only once, at the beginning of the algorithm, as $\mathbf{J} = \mathbf{J}(\vec{x}_0)$ and then never updating it. This requires only one calculation of $\mathbf{J}$ and only one LU decomposition, which can then be re-used in each iteration for different right-hand sides, thus reducing the computational cost to $O(n^2)$ *per iteration*. The disadvantage is that the method then only has linear convergence order and is even more sensitive to finding a "good" choice of x_0.

5.2 Broyden Method

The Broyden method (Charles George Broyden, 1965) seeks a middle ground between completely re-computing the Jacobian in each iteration and never doing so. The idea is to *update* the Jacobian from the previous iteration instead of computing a new one from scratch. The method uses a rank-1 update, meaning that the Jacobian is updated by one linearly independent vector direction in each iteration. This yields an approximation of $\mathbf{J}_{k+1}$ as a modification of $\mathbf{J}_k$.
Let $\vec{\Delta}$ again denote the step taken by the method in one iteration, i.e., $\vec{x}_{k+1} = \vec{x}_k + \vec{\Delta}_k$. Then, we find an approximation for the Jacobian $\mathbf{J}_{k+1} = \mathbf{J}(\vec{x}_k + \vec{\Delta}_k)$ by Expanding $\vec{f}$ into a Taylor series. For any arbitrary index k, hence omitted, this yields:

$$\vec{f}(\vec{x}) = \vec{f}(\vec{x} + \vec{\Delta}) - \mathbf{J}(\vec{x} + \vec{\Delta})\vec{\Delta} + O(|\vec{\Delta}|^2)$$

Image: Documenta Mathematica
Charles George Broyden
∗ 3 February 1933
Essex, UK
† 20 May 2011
Bologna, Italy

and therefore the linear approximation:

$$\mathbf{J}(\vec{x}+\vec{\Delta})\vec{\Delta} \approx \vec{f}(\vec{x}+\vec{\Delta}) - \vec{f}(\vec{x})\,,$$

leading to a linear system of equations for the linear approximation of the Jacobian $\bar{\mathbf{J}} \approx \mathbf{J}(\vec{x}+\vec{\Delta})$:

$$\bar{\mathbf{J}}\vec{\Delta} = \vec{f}(\vec{x}+\vec{\Delta}) - \vec{f}(\vec{x}). \tag{5.3}$$

However, this system of equations cannot be solved, because there are only n equations for n^2 unknowns (i.e., the n^2 entries of $\bar{\mathbf{J}}$).
Therefore, Broyden made the (arbitrary, but reasonable) assumption that $\bar{\mathbf{J}} - \mathbf{J} = 0$ in directions perpendicular to $\vec{\Delta}$, i.e., that $\vec{\Delta}$ is a normal vector onto the null space of $\bar{\mathbf{J}} - \mathbf{J}$. This assumption is reasonable because in the Newton method the solution $\vec{x}$ only changes in direction $\vec{\Delta}$, to $\vec{x}+\vec{\Delta}$, making changes in the Jacobian perpendicular to $\vec{\Delta}$ less important. Therefore, the Broyden approximation is to just set them to zero and neglect them. This assumption then implies that:

$$(\bar{\mathbf{J}} - \mathbf{J})\vec{y} = \vec{0} \quad \forall \vec{y} \text{ with } \vec{y}^{\mathsf{T}}\vec{\Delta} = 0. \tag{5.4}$$

This means:

- $\vec{y}$ is a vector in an $(n-1)$-dimensional subspace of $\mathbb{R}^n$
- $\mathbf{B} = \bar{\mathbf{J}} - \mathbf{J}$ has rank 1. This is because $\vec{y} \in \ker(\mathbf{B})$ and therefore $\dim(\ker(\mathbf{B})) = n-1$. Remember that for any matrix it always holds that $\dim(\ker(\mathbf{B})) + \operatorname{rank}(\mathbf{B}) = n$.
- because $\bar{\mathbf{J}} - \mathbf{J}$ has rank 1, it can be written as

  $$\bar{\mathbf{J}} - \mathbf{J} = \vec{u}\vec{\Delta}^{\mathsf{T}} \tag{5.5}$$

  for some $\vec{u} \in \mathbb{R}^n$.

Outer product

The outer product between two n-vectors forms an $n \times n$ matrix

$$\vec{u}\vec{\Delta}^{\mathsf{T}} = \begin{bmatrix} u_1\vec{\Delta}^{\mathsf{T}} \\ \vdots \\ u_n\vec{\Delta}^{\mathsf{T}} \end{bmatrix} = \begin{bmatrix} u_1\Delta_1 & \dots & u_1\Delta_n \\ \vdots & & \vdots \\ u_n\Delta_1 & \dots & u_n\Delta_n \end{bmatrix}$$

of rank 1, because all rows are scalar multiples of $\vec{\Delta}^{\mathsf{T}}$, and thus linearly dependent, with only one of them linearly independent.

It is trivial to see that the rank-1 ansatz in Eq. 5.5 satisfies Eq. 5.4, because: $(\vec{u}\vec{\Delta}^{\mathsf{T}})\vec{y} = \vec{u}(\vec{\Delta}^{\mathsf{T}}\vec{y}) = \vec{0}$ for any $\vec{y}$ with $\vec{y}^{\mathsf{T}}\vec{\Delta} = \vec{\Delta}^{\mathsf{T}}\vec{y} = 0$. Using this rank-1

ansatz, there are only n unknowns left, namely $(u_1, \ldots, u_n)$. From Eq. 5.3, we get:

$$(\bar{\mathbf{J}} - \mathbf{J})\vec{\Delta} = \vec{f}(\vec{x} + \vec{\Delta}) - \vec{f}(\vec{x}) - \mathbf{J}\vec{\Delta}.$$

In the Newton method, the last term $\mathbf{J}\vec{\Delta} = -\vec{f}(\vec{x})$ (see Algorithm 9), and therefore:

$$(\bar{\mathbf{J}} - \mathbf{J})\vec{\Delta} = \vec{f}(\vec{x} + \vec{\Delta}).$$

Using the rank-1 ansatz from Eq. 5.5, this is:

$$\vec{u}\vec{\Delta}^\mathsf{T}\vec{\Delta} = \vec{f}(\vec{x} + \vec{\Delta}),$$

which can be solved for

$$\vec{u} = \frac{1}{\vec{\Delta}^\mathsf{T}\vec{\Delta}}\vec{f}(\vec{x} + \vec{\Delta}).$$

Note that $\vec{\Delta}^\mathsf{T}\vec{\Delta}$ is a scalar, so that this expression is easy to compute. Finally, from Eq. 5.5. we find the update formula

$$\mathbf{J}(\vec{x} + \vec{\Delta}) \approx \bar{\mathbf{J}} = \mathbf{J} + \vec{u}\vec{\Delta}^\mathsf{T} = \mathbf{J} + \frac{1}{\vec{\Delta}^\mathsf{T}\vec{\Delta}}\vec{f}(\vec{x} + \vec{\Delta})\vec{\Delta}^\mathsf{T}. \tag{5.6}$$

Using this formula, the Jacobian can be approximately updated from one iteration to the next without having to recompute it from scratch. This then yields the Broyden method as given in Algorithm 10.

Algorithm 10 n-Dimensional Broyden Method

```
1:  procedure BROYDEN(x⃗_0, ATOL, RTOL)                ▷ Start vector x⃗_0
2:      x⃗ = x⃗_0
3:      J = J(x⃗)
4:      v⃗ = f⃗(x⃗)
5:      repeat
6:          Solve JΔ⃗ = −v⃗ for Δ⃗
7:          x⃗ = x⃗ + Δ⃗
8:          v⃗ = f⃗(x⃗)
9:          J = J + (1/(Δ⃗ᵀΔ⃗)) v⃗Δ⃗ᵀ                  ▷ Eq. 5.6
10:     until |Δ⃗| ≤ |x⃗|RTOL + ATOL
11: end procedure
```

The Algorithm requires $O(n^2)$ operations in the update step in line 9, but line 6 would still require $O(n^3)$ operations if done naively, in which case the full Newton method would be preferable. Fortunately, however, the LU decomposition of $\mathbf{J}$ only needs to be done once during initialization and the update can be done directly on the factors $\mathbf{L}$ and $\mathbf{U}$ by forward/backward substitution. This then brings the algorithm overall to $O(n^2)$, which is significantly faster than Netwon. The convergence oder of the Broyden method is between 1 and 2, depending on the function $\vec{f}(\vec{x})$. The Broyden method can also be easily adapted to the case

where derivatives of $\vec{f}$ are not available. The only place where the derivatives are required is in the initial computation of the Jacobian in line 3. It turns out that the method also works when simply initializing $\mathbf{J} = \mathbf{1}$ and then iteratively adapting from there. This yields a derivative-free method, which, however, converges slower.

A further optimization of the method is given by the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, which uses the Hessian matrix (i.e., the matrix of second derivatives of $\vec{f}$) to accelerate convergence.

Chapter 6

Scalar Polynomial Interpolation

Very often in computational modeling and simulation, data are given by a finite set of points in 1D. At these locations, some function value is measured or modeled. The problem then frequently arises to reconstruct a continuous function that goes through these data points, i.e., to find a function $g(x) : \mathbb{R} \to \mathbb{R}$ such that $g(x_i) = f_i$ for a finite set of *collocation points* x_i, $i = 0, 1, \ldots, n$ with values $f_i = f(x_i)$ given at these points. Situations where problems of this sort arise are plentiful, from evaluating differential operators in numerical solutions of differential equations, to data filtering and extrapolation, to computer graphics, cryptography, and numerical integration.
Of particular interest is the special case where $g(x)$ is a polynomial. This is due to the pleasant structure of polynomials, which, for example, easily allows analytically computing their integrals and derivatives and provides an orthogonal basis. Moreover, polynomials are related to arbitrary nonlinear functions through Taylor series expansion. The *interpolant* $g(x)$ is then a polynomial of degree $\leq n$, thus:

$$g(x) = P_n(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n \tag{6.1}$$

so that $P_n(x_i) = f_i$ for all $i = 0, 1, \ldots, n$. The problem of finding this polynomial is called *polynomial interpolation*. This is illustrated in Fig. 6.1 for the case of three collocation points and a true underlying function $f(x)$, which is to be approximated by a polynomial $P(x)$ from the values f_0, f_1, f_2 at the collocation points x_0, x_1, x_2.
Trivially, the polynomial interpolation problem can be written as a linear system of equations

$$\mathbf{V}_{n+1} \vec{a} = \vec{f}$$

with the vector of unknown coefficients $\vec{a}^{\mathsf{T}} = [a_n, a_{n-1}, \ldots, a_1, a_0]$, the vector of given values $\vec{f}^{\mathsf{T}} = [f_0, f_1, \ldots, f_{n-1}, f_n]$, and the *Vandermonde matrix* of

Image: TU Berlin
Alexandre-Théophile Vandermonde
∗ 28 February 1735
Paris, Kingdom of France
† 1 January 1796
Paris, France

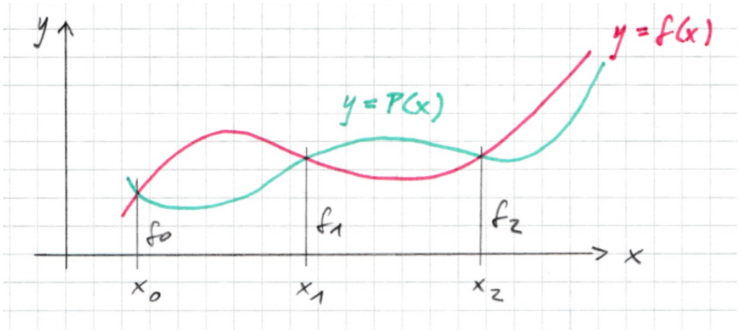


Figure 6.1: Illustration of the polynomial interpolation problem.

order $n+1$ of the collocation points:

$$\mathbf{V}_{n+1} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \vdots & \vdots & & \vdots & \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{bmatrix}.$$

Solving this linear system yields the interpolation polynomial. However, this way of solving the problem has two important drawbacks: (1) it is expensive ($O(n^3)$ for solving the linear system), and (2) it is numerically inaccurate because the condition number of the Vandermonde matrix tends to be high as it contains a large spectrum of values (from 1 to x_i^n).
As we will see below, there are much better algorithms for solving the scalar polynomial interpolation problem, which do not suffer from error amplification and are as efficient as $O(n^2)$ or even $O(n)$. The polynomial interpolation problem hence provides an example of a mathematical problem for which the straightforward numerical approach is not the best. In higher-dimensional spaces, i.e. when $\vec{x}_i \in \mathbb{R}^m$ for some $m > 1$, however, solving the above linear system is often the only way, and more efficient and robust algorithms are hard to come by (even though they are actively being researched, e.g., `https://arxiv.org/abs/1812.04256`, `https://arxiv.org/abs/1710.10846`, `https://arxiv.org/abs/2010.10824`).
Before looking into algorithms for scalar polynomial interpolation, however, we consider uniqueness and existence of the solutions. These considerations will then naturally lead to the formulation of better algorithms.

6.1 Uniqueness

The solution to the scalar polynomial interpolation problem is unique, i.e., there is only one such polynomial. To prove this, assume that P_n and Q_n are two solution polynomials, each of degree $\leq n$. Because both polynomials solve the same polynomial interpolation problem, their difference $P_n(x) - Q_n(x) =: D_n(x)$

has $n+1$ distinct roots. These roots are the collocation points x_i, $i = 0, 1, \ldots, n$, because

$$P_n(x_i) = Q_n(x_i) = f_i \Longrightarrow D_n(x_i) = 0, \qquad \forall i = 0, 1, \ldots, n.$$

However, since D_n is a polynomial of degree $\leq n$, it cannot have $n+1$ distinct roots. This would contradict the Fundamental Theorem of Algebra, according to which a polynomial of degree n can have at most n distinct roots. Therefore, two different solutions cannot co-exist, and we must have $P_n = Q_n$ so that the solution to the polynomial interpolation problem is unique.

6.2 Existence: The Lagrange Basis

Now that we know that the solution is unique, we ask the question whether it always exists, i.e., whether the polynomial interpolation problem can always be (uniquely) solved, for *any* given set of collocation points. For that, it is instructive to first consider the special case of $n = 3$ with given collocation points and function values

$$(x_0, f_0),\ (x_1, f_1),\ (x_2, f_2),\ (x_3, f_3).$$

The idea is to find four polynomials of degree $n \leq 3$ such that each of them has a value of 1 at exactly one collocation point, and 0 at all others. That is, we would like to find:

$$\begin{aligned}
&l_0(x) \text{ such that } l_0(x_1) = l_0(x_2) = l_0(x_3) = 0 \text{ and } l_0(x_0) = 1,\\
&l_1(x) \text{ such that } l_1(x_0) = l_1(x_2) = l_1(x_3) = 0 \text{ and } l_1(x_1) = 1,\\
&l_2(x) \text{ such that } l_2(x_0) = l_2(x_1) = l_2(x_3) = 0 \text{ and } l_2(x_2) = 1,\\
&l_3(x) \text{ such that } l_3(x_0) = l_3(x_1) = l_3(x_2) = 0 \text{ and } l_3(x_3) = 1.
\end{aligned}$$

These basis functions are the *Lagrange Polynomials* of degree 3, named after Joseph-Louis Lagrange who published them in 1795, although they were earlier discovered by Edward Waring in 1779. If such polynomials exist then, by definition,

$$P_3(x) = f_0 l_0(x) + f_1 l_1(x) + f_2 l_2(x) + f_3 l_3(x)$$

solves the interpolation problem because $P_3(x_i) = f_i$ for all $i = 0, 1, 2, 3$. It therefore suffices to show that Lagrange polynomials l_i can always be constructed for given data. Take, for example, l_0. The ansatz

$$l_0(x) = c_0(x - x_1)(x - x_2)(x - x_3)$$

with $c_0 \neq 0$ clearly fulfills the first three conditions $l_0(x_1) = l_0(x_2) = l_0(x_3) = 0$. For disjoint collocation points (i.e., $x_0 \neq x_1 \neq x_2 \neq x_3$), we can always determine c_0 from the remaining condition, namely we find c_0 such that $l_0(x_0) = 1$, which is the case for

$$c_0 = \frac{1}{(x_0 - x_1)(x_0 - x_2)(x_0 - x_3)}.$$

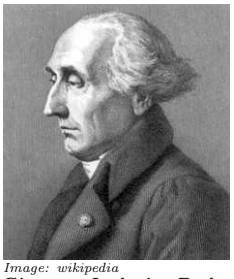


Image: wikipedia
Giuseppe Ludovico De la Grange Tournier (later: Joseph-Louis Lagrange)
∗ 25 January 1736
Turin, Piedmont-Sardinia
† 10 April 1813
Paris, France

Image: wikipedia
Edward Waring
∗ c. 1736
Old Heath, England
† 15 August 1798
Plealey, England

Therefore,

$$l_0(x) = \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)}$$

fulfills all four conditions. Similarly, we find the other Lagrange polynomials:

$$l_1(x) = \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)},$$
$$l_2(x) = \frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)},$$
$$l_3(x) = \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)},$$

all of which satisfy

$$l_i(x_j) = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j. \end{cases}$$

In this Lagrange basis, we can immediately write down the solution of the polynomial interpolation problem in *Lagrange form* as:

$$P_3(x) = L_3(x) = f_0 l_0(x) + f_1 l_1(x) + f_2 l_2(x) + f_3 l_3(x).$$

Clearly, this construction can be repeated for arbitrary n, where we find:

$$l_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{(x-x_j)}{(x_i-x_j)} \qquad i = 0, 1, \ldots, n$$
$$L_n(x) = \sum_{i=0}^{n} f_i l_i(x), \tag{6.2}$$

the general Langrange basis and Lagrange form of degree $\leq n$. Since these polynomials always exists, the polynomial interpolation problem always possesses a solution (and the solution is unique, as shown above).

6.3 Barycentric Representation

The solution to the polynomial interpolation problem is unique and, therefore, there is only one polynomial interpolant. Nevertheless, this one polynomial can be represented in different ways. The above Lagrange form is one possibility, for which the construction from the Lagrange basis polynomials is particularly simple. However, the Lagrange form is costly to evaluate, because for every query point x, all n products for each of the n Lagrange polynomials in Eq. 6.2 need to be recomputed from scratch, requiring $O(n^2)$ operations per query point. The barycentric form of the same polynomial provides a computationally more efficient form. The idea is to reduce any evaluation of the interpolant to computing a weighted sum, and to segregate all multiplicative terms into pre-factors

that only depend on the locations of the collocation points x_i, but not on the query point x.

Definition 6.1 (Barycentric weights)**.** *For an interpolation problem with collocation points $x_0, x_1, \ldots, x_n$, the* barycentric weights *are*

$$\lambda_i = \frac{1}{\prod\limits_{\substack{j=0 \\ j\neq i}}^{n} (x_i - x_j)}, \qquad i = 0, \ldots, n.$$

The barycentric weights only depend on the locations of the collocation points, but not on the query point. In order to evaluate the interpolation polynomial for any given query point x, one then computes:

$$\mu_i = \frac{\lambda_i}{x - x_i}, \qquad i = 0, \ldots, n$$
$$P_n(x) = \sum_{i=0}^{n} \mu_i f_i \Bigg/ \sum_{i=0}^{n} \mu_i. \tag{6.3}$$

In this representation, the λ_i can be precomputed once in the beginning and reused for each subsequent evaluation of the polynomial. This is beneficial, because the λ_i contain all the expensive products. In fact, computing the barycentric weights is $O(n^2)$, but only needs to be done once, independent of the query point. Later evaluation of the polynomial according to Eq. 6.3 only contains the sums that can be computed in $O(n)$ operations per query point. This renders each evaluation of $P_n(x)$ cheaper than in the Lagrange form.

Derivation of the barycentric formula

In the Lagrange form, as given in Eq. 6.2, we can factorize the numerator

$$\hat{\ell}(x) = \prod_{j=0}^{n} (x - x_j)$$

from the denominator

$$\lambda_i = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{1}{x_i - x_j} = \frac{1}{\prod\limits_{\substack{j=0 \\ j\neq i}}^{n} (x_i - x_j)},$$

which are the Barycentric weights that are independent of the query point x. Using the definition of μ_i from Eq. 6.3, the interpolation polynomial

can hence be written as:

$$P_n(x) = \hat{\ell}(x) \sum_{i=0}^{n} \mu_i f_i,$$

because $\hat{\ell}(x)$ is independent of i. Since by definition

$$\sum_{i=0}^{n} l_i(x) = 1,$$

this trivially (i.e., by dividing with 1) yields:

$$P_n(x) = \frac{\hat{\ell}(x) \sum_{i=0}^{n} \mu_i f_i}{\sum_{i=0}^{n} l_i(x)}.$$

This is:

$$P_n(x) = \frac{\hat{\ell}(x) \sum_{i=0}^{n} \mu_i f_i}{\hat{\ell}(x) \sum_{i=0}^{n} \mu_i}.$$

After crossing out $\hat{\ell}$, we obtain the Barycentric formula as given in Eq. 6.3.

6.4 Aitken-Neville Algorithm

A particularly efficient and numerically accurate method for determining the interpolation polynomial and directly also evaluating it at a given query point x is the algorithm by Alexander Aitken and Eric Neville (1934). The Aitken-Neville algorithm (also known as *divided difference scheme*) determines *and* evaluates the polynomial in $O(n^2)$ computation steps. We introduce this algorithm here for the special case $n = 3$.

In this case, we are given collocation points (x_i, f_i) for $i = 0, 1, 2, 3$ and want to find the value of the interpolant $P_3(x)$ for a given x. The computational scheme for this case is illustrated in Fig. 6.2. It starts in the first column with the four collocation values f_0, f_1, f_2, f_3, which we call P_0, P_1, P_2, P_3 (here the subscript is an index and not the degree of the polynomial; all of these are constants and hence polynomials of degree 0). From these, the second column computes values P_{01}, P_{12}, P_{23} according to the following rules:

$$P_{01} = P_1 + \frac{x - x_1}{x_0 - x_1}(P_0 - P_1),$$
$$P_{12} = P_2 + \frac{x - x_2}{x_1 - x_2}(P_1 - P_2),$$
$$P_{23} = P_3 + \frac{x - x_3}{x_2 - x_3}(P_2 - P_3).$$

In the third column, these values are further combined to:

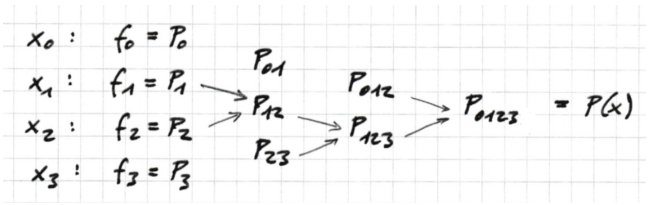


Figure 6.2: The Aitken-Neville scheme for $n = 3$ with one exemplary path highlighted by arrows.

$$P_{012} = P_{12} + \frac{x - x_2}{x_0 - x_2}(P_{01} - P_{12}),$$
$$P_{123} = P_{23} + \frac{x - x_3}{x_1 - x_3}(P_{12} - P_{23}),$$

and, finally, in the last column to:

$$P(x) = P_{0123} = P_{123} + \frac{x - x_3}{x_0 - x_3}(P_{012} - P_{123}).$$

The Aitken-Neville algorithm does not only produce the final result in $O(n^2)$ steps (n columns with $O(n)$ entries each), but also provides all intermediate results along the way. Indeed, for any $k \geq i$, $P_{i\ldots k}$ is the value of the interpolation polynomial through $(x_i, f_i), \ldots, (x_k, f_k)$, evaluated at x.
The generalization of the algorithm to general n should be intuitive from the example above, and we refer to the literature for actual formulations. The Aitken-Neville algorithm is a good choice if the polynomial is to be determined and evaluated at only one query point. If the same polynomial is to be evaluated at multiple query points, the Barycentric form is preferable.

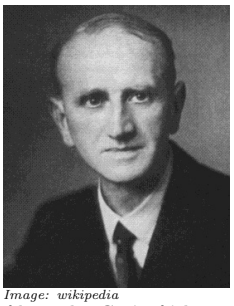


Image: wikipedia
Alexander Craig Aitken
∗ 1 April 1895
Dunedin, New Zealand
† 3 November 1967
Edinburgh, Scotland, UK


Image: JSTOR.org
Eric Harold Neville
∗ 1 January 1889
London, England, UK
† 22 August 1961
Reading, England, UK

6.5 Approximation Error

We know that the interpolation polynomial always exists and is unique, and we have efficient algorithms for computing it. The remaining question is how accurately $P_n(x)$ approximates the unknown $f(x)$ from which the collocation data were sampled. Therefore, we are interested in the approximation error $f(x) - P_n(x)$.
For this, the following theorem from function approximation theory is useful.

Theorem 6.1. *Let $f(x) : [a, b] \to \mathbb{R}$ be a scalar, real-valued function over the closed interval $[a, b] \subset \mathbb{R}$. Assume $f(x)$ is at least $(n + 1)$ times continuously differentiable. For the interpolation polynomial $P_n(x)$ of the $(n + 1)$ collocation points $x_0, x_1, \ldots, x_n$ with $a = \min_i x_i$ and $b = \max_i x_i$, and collocation values*

$f_i = f(x_i)$, we have that for each $x \in [a, b]$, there exists $\xi \in [a, b]$ such that

$$f(x) - P_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i) = \frac{f^{(n+1)}(\xi)}{(n+1)!} l(x), \tag{6.4}$$

where $f^{(n+1)}(\xi)$ is the $(n+1)$-st derivative of the function f evaluated at position ξ.

We omit the proof of this theorem here; it is classic. This theorem tells us that finding an upper bound on the approximation error of a polynomial interpolant requires bounding the absolute value of derivative $|f^{(n+1)}|$ over $[a, b]$. If f is continuous, such a bound always exists and can often be found from prior knowledge.

Example 6.1. *Consider linear interpolation as an example, illustrated in Fig. 6.3. In this case, the interpolation polynomial $P_1(x)$ is a linear function, whereas $f(x)$ could be anything.*

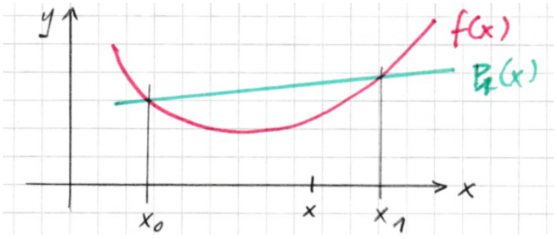


Figure 6.3: Approximating a nonlinear function $f(x)$ by a linear interpolation polynomial $P_1(x)$.

In this case, Eq. 6.4 becomes:

$$f(x) - P_1(x) = \frac{f''(\xi)}{2}(x - x_0)(x - x_1).$$

The function $l(x) = (x - x_0)(x - x_1)$ has its maximum over $[a, b] = [x_0, x_1]$ at the center of the interval, thus for $x = \frac{1}{2}(x_0 + x_1)$ the error is largest and amounts to:

$$\frac{f''(\xi)}{2} \frac{1}{2}(x_1 - x_0) \frac{1}{2}(x_0 - x_1)$$

and, if we call $h = x_1 - x_0$,

$$\frac{f''(\xi)}{2} \frac{h}{2} \frac{-h}{2} = -\frac{1}{8} h^2 f''(\xi).$$

If additionally $|f''(\xi)| \leq M_2$ for $\xi \in [x_0, x_1]$, then the absolute value of the error is bounded by

$$|f(x) - P_1(x)| \leq \frac{1}{8} h^2 M_2.$$

Indeed it is true in general that polynomial interpolation with $P_n(x)$ of degree $\leq n$ has an approximation error that scales as $O(h^{n+1})$ for equidistant collocation points $x_i = x_0 + ih$, $i = 1, 2, 3, \ldots$, with spacing h. The function $l(x) = \prod_{i=0}^{n}(x - x_i)$ has at least one extremum between any two collocation points, where it has roots. For large n, however, the amplitude of these extrema is rapidly increases toward the boundaries of the interval $[a, b]$. This *ringing phenomenon* (also called *Runge's phenomenon*) means that increasing the degree of the interpolation polynomial does not necessarily decrease the error, as the theorem only makes a statement about the *scaling* of the error.
A common solution to this problem is to interpolate piecewise, as illustrated in Fig. 6.4 for piecewise linear interpolation. Instead of approximating $f(x)$ by *one* polynomial of degree n, we approximate $f(x)$ by multiple polynomials, possibly of lower degree. This allows reducing h (and hence the approximation error) without having to increase n, effectively avoiding the ringing phenomenon.

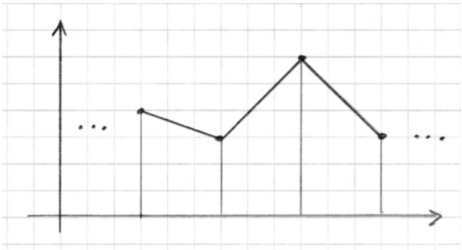


Figure 6.4: Illustration of piecewise linear interpolation.

6.6 Spline Interpolation

The problem with piecewise interpolation is that the interpolant is no longer continuously differentiable. While this may not be a problem for some applications (e.g., numerical integration), it is undesirable for others (e.g., numerical differentiation). In order to obtain a piecewise interpolant that is continuously differentiable to some order, we need to impose additional smoothness conditions at the transition points. This is the basic idea of Spline interpolation.

Image: wikipedia
Charles Hermite
∗ 24 December 1822
Dieuze, France
† 14 January 1901
Paris, France

6.6.1 Hermite interpolation

The concept of Splines is best understood when starting from the classic formulation of Hermite interpolation (Charles Hermite, 1878). There, we are given n collocation points, ordered[1] such that

$$x_1 < x_2 < \cdots < x_i < x_{i+1} < \cdots < x_n.$$

[1] Notice that so far the ordering of the collocation points did not matter and could have been arbitrary. This is no longer the case for Splines.

(N.B.: the numbering of the collocation points now starts from 1, as is usual in the Spline literature) Assume that at these points, we known both the value and the first derivative of an unknown function $f(x)$, hence:

$$f_1 = f(x_1), \ldots, f_i = f(x_i), \ldots, f_n = f(x_n),$$
$$f'_1 = f'(x_1), \ldots, f'_i = f'(x_i), \ldots, f'_n = f'(x_n).$$

In each interval $[x_i, x_{i+1}]$, we want to determine a polynomial $P_i(x)$ (here the subscript i is the interval index and not the polynomial degree), so that

$$\begin{aligned} P_i(x_i) &= f_i, \quad P_i(x_{i+1}) = f_{i+1}, \\ P'_i(x_i) &= f'_i, \quad P'_i(x_{i+1}) = f'_{i+1}. \end{aligned} \tag{6.5}$$

These are four constraints, and therefore the $P_i(x)$ are of degree ≤ 3. Let $h_i = x_{i+1} - x_i$ and apply the transformation of variables

$$t = \frac{x - x_i}{h_i} \quad \Longleftrightarrow \quad x = x_i + h_i t, \tag{6.6}$$

such that in each interval, $t \in [0, 1]$ measures normalized position relative to the interval. Then, the problem is the same in each interval, and we can solve it for one representative interval. This defines the scaled polynomials

$$Q_i(t) = P_i(x_i + h_i t) \quad \Longleftrightarrow \quad P_i(x) = Q_i\left(\frac{x - x_i}{h_i}\right).$$

For these scaled polynomials, the conditions from Eq. 6.5 become:

$$\begin{aligned} Q_i(0) &= f_i, \quad Q_i(1) = f_{i+1} \\ \dot{Q}_i(0) &= h_i f'_i, \quad \dot{Q}_i(1) = h_i f'_{i+1}. \end{aligned} \tag{6.7}$$

where $\dot{Q}_i = \frac{\mathrm{d}Q_i(t)}{\mathrm{d}t}$ and thus the additional pre-factors h_i come from the inner derivative (chain rule of differentiation). Solving this linear system of equations for the unknown coefficients of the cubic polynomial $Q_i(t) = a_0 t^3 + a_1 t^2 + a_2 t + a_3$, one finds:

$$Q_i(t) = f_i(1 - 3t^2 + 2t^3) + f_{i+1}(3t^2 - 2t^3) + h_i f'_i(t - 2t^2 + t^3) + h_i f'_{i+1}(-t^2 + t^3). \tag{6.8}$$

Transforming this back by substituting the definition of t from Eq. 6.6 yields the final *Hermite interpolation polynomial*

$$g(x) \coloneqq P_i(x) \text{ for } x \in [x_i, x_{i+1}].$$

It is a piecewise cubic polynomial that is continuously differentiable everywhere. Similar to the Aitken-Neville algorithm for global polynomial interpolation, an efficient algorithm also exists for determining $g(x)$ and directly evaluating it at a given query point x. It starts by first determining the index i for which $x_i \leq x < x_{i+1}$. Then, compute

$$h_i = x_{i+1} - x_i, \qquad t = \frac{x - x_i}{h_i}.$$

Finally, the result is computed as

$$g(x) = Q_i\left(t = \frac{x - x_i}{h_i}\right) = a_0 + (b_0 + (c_0 + d_0 t)(t-1))t.$$

The coefficients a_0, b_0, c_0, d_0 are computed according to the scheme shown in Fig. 6.5.

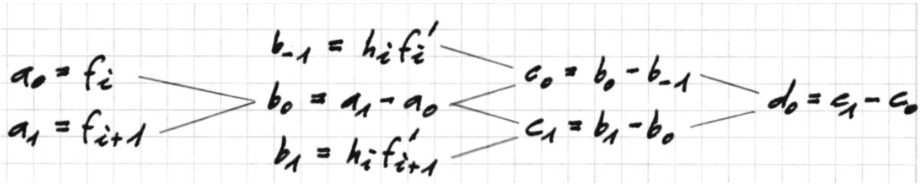


Figure 6.5: Calculation of the coefficients of a Hermite interpolation polynomial.

As elegant and efficient as Hermite interpolation is, it suffers from the problem that the values f_i' are often not available in practice. A way out is to determine the f_i' from additional conditions, leading to cubic Splines.

6.6.2 Cubic splines

Cubic Splines introduce additional conditions into the Hermite framework in order to do away with the need of having derivative data available. The goal is still the same: interpolate $f(x)$ by a piecewise cubic polynomial that is continuously differentiable everywhere.
Instead of imposing known derivative values at the transition points, however, cubic Splines demand that the second derivatives of the polynomials are continuous at interior collocation points, thus:

$$P_i''(x_{i+1}) \stackrel{!}{=} P_{i+1}''(x_{i+1}), \qquad i = 1, \ldots, n-2.$$

This is trivially the case for the function $f(x)$ itself. Imposing this also for the interpolant provides $n-2$ conditions from which the unknown $f_1', \ldots, f_n'$ can be determined using additionally two boundary conditions. Applying again the Hermite transformation from Eq. 6.6, this condition becomes:

$$\frac{\ddot{Q}_i(1)}{h_i^2} \stackrel{!}{=} \frac{\ddot{Q}_{i+1}(0)}{h_{i+1}^2}.$$

Substituting the second derivative of the expression for $Q(t)$ from Eq. 6.8, suitably evaluated at $t = 0$ or $t = 1$, this condition reads:

$$\frac{6}{h_i^2}(f_i - f_{i+1}) + \frac{2}{h_i}f_i' + \frac{4}{h_i}f_{i+1}' = \frac{6}{h_{i+1}^2}(f_{i+2} - f_{i+1}) - \frac{4}{h_{i+1}}f_{i+1}' - \frac{2}{h_{i+1}}f_{i+2}'.$$

Image: wikipedia
Carl-Wilhelm Reinhold de Boor
* 3 December 1937
Stolp, Germany
(now: Słupsk, Poland)

Listing these conditions for all interior nodes (N.B., the collocation points are usually called "nodes" in Spline interpolation) leads to the linear system of equations:

$$\begin{bmatrix} b_1 & a_1 & b_2 & & & & \\ & b_2 & a_2 & b_3 & & & \\ & & \cdots & \cdots & \cdots & & \\ & & & & b_{n-2} & a_{n-2} & b_{n-1} \end{bmatrix} \begin{bmatrix} f'_1 \\ f'_2 \\ \vdots \\ f'_n \end{bmatrix} = \begin{bmatrix} d_2 \\ d_3 \\ \vdots \\ d_{n-1} \end{bmatrix}, \tag{6.9}$$

with

$$\begin{aligned} b_i &= \frac{1}{h_i}, \\ a_i &= \frac{2}{h_i} + \frac{2}{h_{i+1}}, \\ c_i &= \frac{f_{i+1} - f_i}{h_i^2}, \\ d_{i+1} &= 3(c_i + c_{i+1}) = 3\left(\frac{f_{i+1} - f_i}{h_i^2} + \frac{f_{i+2} - f_{i+1}}{h_{i+1}^2}\right). \end{aligned}$$

These are $n-2$ equations for n unknowns. We thus need two additional conditions to render the system determined. These are conditions at the boundary nodes. There are many possible choices, leading to different flavors of cubic Splines (n-Splines, B-Splines, etc.). A classic choice is de Boor's "not a knot" condition (1978):

$$\begin{aligned} P_1(x) &\equiv P_2(x), \\ P_{n-2}(x) &\equiv P_{n-1}(x), \end{aligned} \tag{6.10}$$

which can be motivated from approximation error considerations. If the polynomials in the first two and last two intervals are identical, then in particular also their third derivatives at the collocation points are the same:

$$P_1 \equiv P_2 \quad \Longrightarrow \quad P_1''' \equiv P_2''' \quad \Longrightarrow \quad \frac{\dddot{Q}_1(1)}{h_1^3} = \frac{\dddot{Q}_2(0)}{h_2^3}.$$

Substituting again the expression for $Q(t)$ from Eq. 6.8, this becomes:

$$\begin{aligned} \frac{1}{h_1^2} f'_1 + \left(\frac{1}{h_1^2} - \frac{1}{h_2^2}\right) f'_2 - \frac{1}{h_2^2} f'_3 &= 2\left(\frac{c_1}{h_1} - \frac{c_2}{h_2}\right) \\ \frac{1}{h_1} f'_1 + \left(\frac{1}{h_1} + \frac{1}{h_2}\right) f'_2 &= 2c_1 + \frac{h_1}{h_1 + h_2}(c_1 + c_2). \end{aligned}$$

Analogously, the condition obtained from $P_{n-2} \equiv P_{n-1}$ is:

$$\frac{1}{h_{n-1}} f'_n + \left(\frac{1}{h_{n-2}} + \frac{1}{h_{n-1}}\right) f'_{n-1} = 2c_{n-1} + \frac{h_{n-1}}{h_{n-1} + h_{n-2}}(c_{n-1} + c_{n-2}).$$

This provides the remaining two equations for the linear system in Eq. 6.9, which can then be solved for the unknown $f'_1, \ldots, f'_n$. This linear system has tridiagonal structure and is strictly diagonally dominant, as is easily verified. The system matrix therefore is regular and the $f'_1, \ldots, f'_n$ uniquely determined. Because of its tridiagonal structure, the system can be efficiently solved, e.g., using Gaussian elimination without pivoting (i.e., with direct diagonal strategy). After the system has been solved, the $f'_1, \ldots, f'_n$ can be used in the standard Hermite algorithm as given in Fig. 6.5 to determine the final piecewise cubic interpolant.

While the $f'_1, \ldots, f'_n$ were the exact values of the derivative in the classic Hermite approach, the values determined by solving the Spline system are approximations of the derivative of $f(x)$ at the collocation points. For Splines, they are, in fact, the derivatives of the interpolant instead.

While we have discussed cubic Splines here, as they are directly related to Hermite interpolation and are the most frequently used in practice, higher-order Splines can be derived analogously.

Finally, as already outlined, the "not a knot" condition from Eq. 6.10 is not the only possible choice for closing the system. Other boundary conditions at the first and last interval can be chosen, leading to different flavors of Splines. Particularly popular are:

- Natural Splines (n-Splines): $P''_1(x_1) = 0$, $P''_{n-1}(x_n) = 0$.
- Periodic Splines (p-Splines): $f_1 = f_n$, $f'_1 = f'_n$. In this case there are only $n - 1$ unknowns $f'_2, \ldots, f'_n$ such that one additional condition is enough: $P''_1(x_1) = P''_{n-1}(x_n)$.

Splines can be efficiently evaluated, for example using the Cox-de Boor algorithm (not covered here), they afford an elegant decomposition in high-dimensional spaces, and iterative calculations are possible as well. It is therefore no surprise that Spline interpolation enjoys widespread use, from signal processing to computer graphics (in the form of NURBS and Bézier curves).

Chapter 7

Trigonometric Interpolation

In cases where the interpolant $g(x) : \mathbb{R} \to \mathbb{R}$ is not a polynomial, other types of interpolation arise. Indeed, interpolation can be formulated in any function space given by some basis functions. For polynomial interpolation, as considered in the previous chapter, the function space in which $g(x)$ lives is the space of polynomials with degree $\leq n$, given by the monomial basis $(1, x, x^2, \ldots, x^n)$. Another space of common interest is the space of harmonic functions with angular frequency $\leq n$, which, in 1D, is spanned by the basis $(1, \sin x, \cos x, \sin 2x, \cos 2x, \ldots, \sin nx, \cos nx)$. In this case, $g(x)$ is 2π-periodic, so we also have to require that the collocation data come from a 2π-periodic function, i.e.,

$$f(x + 2\pi) = f(x) \quad \forall x.$$

Further, we assume that the N collocation points are regularly spaced

$$x_k = \frac{2\pi k}{N} \in [0, 2\pi), \quad k = 0, \ldots, N-1$$

with collocation values at these points given by

$$f_k = f(x_k), \quad k = 0, \ldots, N-1.$$

This leads to trigonometric interpolation, which is connected to polynomial interpolation through the trigonometric polynomial. Note that we index the collocation points by k now in order to make explicit that they are evenly spaced, whereas in the previous chapter they were index by i and could be arbitrarily spaced.

7.1 The Trigonometric Polynomial

Note that due to the periodicity, $f_0 = f_N$. According to the Nyquist-Shannon sampling theorem, harmonic functions with angular frequency up to $n \leq N/2$

can be represented with these N collocation points. Therefore, we can determine the interpolant:

$$g_n(x) = \frac{a_0}{2} + \sum_{j=1}^{\frac{N}{2}-1} (a_j \cos(jx) + b_j \sin(jx)) + \frac{1}{2} a_{\frac{N}{2}} \cos\left(\frac{N}{2}x\right), \tag{7.1}$$

so that $g_n(x_k) = f_k$ for all $k = 0, \ldots, N-1$. It will become clear later on, why the term of order $N/2$ only contains a cosine, and no sine. For now, the argument shall suffice that the number of unknown coefficients of the above trigonometric polynomial is N, which is all we can determine from the N collocation points.

7.2 Discrete Fourier Transform (DFT)

The *trigonometric polynomial* in Eq. 7.1 has N unknown coefficients

$$(a_0;\, a_1, b_1, \ldots a_{\frac{N}{2}-1}, b_{\frac{N}{2}-1};\, a_{\frac{N}{2}}),$$

which can be uniquely determined from the N collocation points. The solution of this trigonometric interpolation problem is called *Discrete Fourier Transform* (Jean-Baptiste Joseph Fourier, 1822). An efficient algorithm for computing it can be derived from the infinite Fourier series of the unknown function f from which the data have been sampled, which is defined as:

$$f(x) = \sum_{j=-\infty}^{\infty} c_j \mathrm{e}^{ijx}, \quad c_j \in \mathbb{C}, \tag{7.2}$$

where the coefficients are complex numbers and the Euler notation is used for compactness: $\mathrm{e}^{ijx} = \cos(jx) + i \sin(jx)$. According to standard Fourier theory, the Fourier coefficients of f are determined by:

$$c_j = \frac{1}{2\pi} \int_0^{2\pi} f(x) \mathrm{e}^{-ijx} \,\mathrm{d}x. \tag{7.3}$$

This continuous integral can be approximated in the discrete case. A standard method for integral approximation is illustrated in Fig. 7.1. If the integral over a continuous function $F(x)$ is interpreted as the area under the graph of this function, then the integration interval $[a, b]$ can be split into many smaller sub-intervals, each of width $h = (b-a)/N$, and the area under the curve in one of these sub-intervals $[a+kh, a+(k+1)h]$ can be approximated by the area of the shaded rectangle as:

$$\frac{h}{2} \left(F(a+kh) + F(a+(k+1)h)\right).$$

Summing over all sub-intervals $l = 0, 1, \ldots, N$ then yields:

$$\int_a^b F(x) \,\mathrm{d}x \approx \frac{h}{2}(F(a) + F(b)) + h \sum_{k=1}^{N-1} F(a+kh).$$

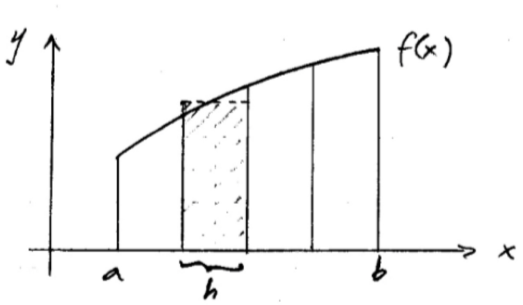


Figure 7.1: Approximation of an integral by trapezoidal blocks.

Applying this to Eq. 7.3 with $F(x) = \frac{1}{2\pi} f(x) \mathrm{e}^{-ijx}$ and collocation points $x_k = \frac{2\pi k}{N}$, we find:

$$
\begin{aligned}
c_j \approx \tilde{c}_j &= \frac{h}{4\pi} \left(f(0) + f(2\pi) \right) + \frac{h}{2\pi} \sum_{k=1}^{N-1} f\left(\frac{2\pi k}{N} \right) \mathrm{e}^{-ij\frac{2\pi k}{N}} \\
&= \frac{1}{N} \sum_{k=0}^{N-1} f\left(\frac{2\pi k}{N} \right) \mathrm{e}^{-ij\frac{2\pi k}{N}}, \quad j = 0, 1, \ldots, N-1 \qquad (7.4)
\end{aligned}
$$

Image: wikipedia
Jean-Baptiste Joseph Fourier
∗ 21 March 1768
Auxerre, Kingdom of France
† 16 May 1830
Paris, Kingdom of France

because $f(0) = f(2\pi)$ due to the 2π-periodicity of the function and $h = 2\pi/N$. This can be written more compactly by defining the following vectors in $\mathbb{C}^N$:

$$
\vec{f}_N \coloneqq \begin{bmatrix} f_0 \\ \vdots \\ f_{N-1} \end{bmatrix}, \qquad \vec{\tilde{c}}_N \coloneqq \begin{bmatrix} \tilde{c}_0 \\ \vdots \\ \tilde{c}_{N-1} \end{bmatrix}.
$$

Then, Eq. 7.4 reads:

$$
\boxed{\vec{\tilde{c}}_N = \frac{1}{N} \mathbf{W} \vec{f}_N, \qquad (7.5)}
$$

where the matrix $\mathbf{W}$ has entries $(w)_{jk} = w^{jk} = \mathrm{e}^{-ij\frac{2\pi k}{N}}$ for $j, k = 0, \ldots, N-1$ with $w \coloneqq \mathrm{e}^{-i\frac{2\pi}{N}}$, the N-th complex root of unity. In the usual notational convention where the first subscript of a matrix denotes the row index and the second subscript the column index, this is:

$$
\mathbf{W} = \begin{bmatrix}
1 & 1 & 1 & \ldots & 1 \\
1 & w & w^2 & \ldots & w^{N-1} \\
1 & w^2 & w^4 & \ldots & w^{2(N-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & w^{N-1} & w^{2(N-1)} & \ldots & w^{(N-1)^2}
\end{bmatrix}.
$$

The mapping $\vec{f}_N \in \mathbb{C}^N \mapsto \frac{1}{N}\mathbf{W}\vec{f}_N \in \mathbb{C}^N$ is called *Discrete Fourier Transform* (i.e., discrete Fourier analysis) and it requires $O(N^2)$ operations to be computed, because that is the number of elements in $\mathbf{W}$.
Since $\mathbf{W}^{-1} = \frac{1}{N}\overline{\mathbf{W}}$, where $\overline{\mathbf{W}}$ is the element-wise complex conjugate of $\mathbf{W}$ (which is in this case identical to the conjugate transpose $\mathbf{W}^H$ because $\mathbf{W}$ is a symmetric matrix), the inverse discrete Fourier transform (i.e., discrete Fourier synthesis) is simply:

$$\vec{f}_N = \overline{\mathbf{W}}\vec{\tilde{c}}_N \tag{7.6}$$

and can also be computed in $O(N^2)$ operations.
Since the sequence of f_k is periodic, i.e. $f_{k\pm N} = f_k$, the sequence of $\tilde{c}_k$ is also periodic, i.e. $\tilde{c}_{k\pm N} = \tilde{c}_k$, because

$$\mathrm{e}^{-ij\frac{2\pi}{N}(k\pm N)} = \mathrm{e}^{-ij\frac{2\pi k}{N}} \underbrace{\mathrm{e}^{\pm ij2\pi}}_{=1} = \mathrm{e}^{-ij\frac{2\pi k}{N}}. \tag{7.7}$$

The fact that the Fourier coefficients are periodic is only true for the DFT, but not for the continuous Fourier transform. It is the reason for *aliasing* phenomena in digital electronics and digital signal processing, as "out of range" frequencies get mirrored/shadowed into the signal.
The inverse discrete Fourier transform can be used to find the interpolant in Eq. 7.1. Indeed, from Eq. 7.6 we have that:

$$\begin{aligned} f_k = f(x_k) = f\left(\frac{2\pi k}{N}\right) &= \sum_{j=0}^{N-1} \tilde{c}_j \mathrm{e}^{ij\frac{2k\pi}{N}} \\ &= \sum_{j=0}^{\frac{N}{2}-1} \tilde{c}_j \mathrm{e}^{ij\frac{2\pi k}{N}} + \sum_{j=\frac{N}{2}+1}^{N-1} \tilde{c}_j \mathrm{e}^{ij\frac{2\pi k}{N}} + \tilde{c}_{\frac{N}{2}} \underbrace{\mathrm{e}^{ik\pi}}_{\cos(k\pi)}, \end{aligned} \tag{7.8}$$

where we simply wrote the matrix-vector product in sum notation and then split the sum into three parts: one up to $N/2-1$, one for $N/2$, and one from $N/2+1$ onward, in order to match the index range of the trigonometric polynomial we seek to determine. Now we perform the index transformation $j = l + N$, such that $l = -1$ for $j = N-1$ and $l = -N/2+1$ for $j = N/2+1$. The second partial sum then becomes:

$$\sum_{j=\frac{N}{2}+1}^{N-1} \tilde{c}_j \mathrm{e}^{ij\frac{2\pi k}{N}} = \sum_{l=-\frac{N}{2}+1}^{-1} \tilde{c}_l \mathrm{e}^{il\frac{2\pi k}{N}}.$$

Notice that due to Eq. 7.7 and the periodicity of f, the expression for the summand did not change. Renaming $l = j$, we can therefore concatenate this sum with the first partial sum from Eq. 7.8, as they sum over the identical

summands, leading to:

$$f_k = \sum_{j=-\frac{N}{2}+1}^{\frac{N}{2}-1} \tilde{c}_j \mathrm{e}^{ijx_k} + \tilde{c}_{\frac{N}{2}} \cos\left(\frac{N}{2}x_k\right).$$

This function has the same form as the interpolant in Eq. 7.1 and fulfills the interpolation property by construction. Therefore, setting

$$g_n(x) = \sum_{j=-\frac{N}{2}+1}^{\frac{N}{2}-1} \tilde{c}_j e^{ijx} + \tilde{c}_{\frac{N}{2}} \cos\left(\frac{N}{2}x\right) \tag{7.9}$$

trivially fulfills $g(x_k) = f(x_k)$ for all $k = 0, 1, \ldots, N-1$. Indeed, this is identical to Eq. 7.1, because

$$\tilde{c}_j \mathrm{e}^{ijx} + \tilde{c}_{-j} \mathrm{e}^{-ijx} = \underbrace{(\tilde{c}_j + \tilde{c}_{-j})}_{a_j} \cos(jx) + \underbrace{(i\tilde{c}_j - i\tilde{c}_{-j})}_{b_j} \sin(jx) \tag{7.10}$$

for $j = 0, 1, \ldots, \frac{N}{2} - 1$, since $\cos(-jx) = \cos(jx)$ and $\sin(-jx) = -\sin(jx)$. Therefore,

$$g_n(x) = \frac{a_0}{2} + \sum_{j=1}^{\frac{N}{2}-1} (a_j \cos(jx) + b_j \sin(jx)) + \frac{1}{2} a_{\frac{N}{2}} \cos\left(\frac{N}{2}x\right).$$

The first term, $a_0/2$, follows from the fact that $\cos(0) = 1$ and $\tilde{c}_0 + \tilde{c}_{-0} = 2c_0$. For real-valued f, the coefficients a_j and b_j are also real, because according to Eq. 7.4, $\overline{\tilde{c}}_j = \tilde{c}_{-j}$ and thus:

$$\begin{aligned} a_j &= \tilde{c}_j + \tilde{c}_{-j} = \Re(\tilde{c}_j) + i\Im(\tilde{c}_j) + \Re(\tilde{c}_j) - i\Im(\tilde{c}_j) = 2\Re(\tilde{c}_j), \\ b_j &= i\tilde{c}_j - i\tilde{c}_{-j} = i\Re(\tilde{c}_j) - \Im(\tilde{c}_j) - i\Re(\tilde{c}_j) - \Im(\tilde{c}_j) = -2\Im(\tilde{c}_j). \end{aligned}$$

The discrete Fourier transform from Eq. 7.5 can be used to directly compute the coefficients of the trigonometric interpolation polynomial for given data (x_k, f_k), $k = 0, \ldots, N-1$ in $O(N^2)$ operations. This is the same computational cost as polynomial interpolation (see previous chapter), which is not surprising because both are interpolation problems, albeit in different basis functions.
An important result, which we give here without proof, states that the trigonometric interpolation polynomial $g_n(x)$ above is the optimal approximation of the function $f(x)$ in the sense that:

$$g_n(x) = \arg\min_{g_n'(x)} \|g_n'(x) - f(x)\|_2$$

with the 2π-periodic function 2-norm defined as:

$$\|g_n(x) - f(x)\|_2 := \left[\int_0^{2\pi} (g_n(x) - f(x))^2 \,\mathrm{d}x\right]^{\frac{1}{2}}.$$

This means that the trigonometric interpolation polynomial approximates the Fourier Series for $|j| < \frac{N}{2}$ in the least-squares sense. It is the polynomial for which the sum of the squared errors everywhere is minimal (and is zero at the collocation points).

7.3 Efficient Computation of DFT by FFT

As we have just seen, the discrete Fourier transform (DFT) can be used to compute a trigonometric polynomial to interpolate an unknown function f from a finite set of evenly spaced discrete samples (x_k, f_k). This has a multitude of important applications, ranging from signal filtering (e.g., edge detectors in computer vision) over data compression methods (e.g., JPEG'92) to numerical solvers for partial differential equations (e.g., spectral methods). It is therefore not surprising that a lot of research has gone into finding efficient ways of computing DFTs. Directly evaluating the matrix-vector products in Eqs. 7.5 and 7.6 requires a computational effort of $O(N^2)$ complex multiplications, which can become prohibitive for large N. This is the generic computational cost of any interpolation problem. For the special case of evenly spaced trigonometric interpolation, however, the computational cost can be further reduced by exploiting the special structure of the matrix $\mathbf{W}$.

Prominently, this reduction of the computational cost works best if the number of collocation points is a power of 2, i.e., $N = 2^m$ for some $m \in \mathbb{Z}^+$, leading to the *Fast Fourier Transform* (FFT) algorithm. The FFT was already used by Carl Friedrich Gauss in unpublished work in 1805 to interpolate the orbit of asteroids Pallas and Juno from observations. It was published in 1965 by James Cooley and John Tukey, who are credited for the invention of the modern FFT. The FFT is on the Millennial "Computing in Science and Engineering" (Jack Dongarra, 2000) list of the 10 most important and influential algorithms of all times (along with: Newton method, LU/QR decomposition, Singular Value Decomposition, Metropolis-Hastings algorithm, QuickSort, PageRank, Krylov subspace iteration, simplex algorithm, Kalman filter).

The idea of FFT is to recursively decompose the DFT computation into smaller problems, so that in the end the complete problem can be solved in only $O(N \log_2 N)$. Indeed, the computation of the Fourier coefficients according to Eq. 7.5

$$\vec{\tilde{c}}_N = \frac{1}{N}\mathbf{W}\vec{f}_N$$

can be split into two parts: one for all coefficients with even index, and one for all coefficients with odd index. For the even coefficients, we find:

$$N\tilde{c}_{2k} = \sum_{j=0}^{N-1} w^{2kj} f_j = \sum_{j=0}^{n-1} w^{2kj} f_j + \sum_{j=n}^{2n-1} w^{2kj} f_j \tag{7.11}$$

with $n = \frac{N}{2}$. For the second partial sum on the right-hand side, we apply the

index transformation $j = j' + n$, leading to:

$$\sum_{j=n}^{2n-1} w^{2kj} f_j = \sum_{j'=0}^{n-1} w^{2k(j'+n)} f_{j'+n} = \sum_{j'=0}^{n-1} w^{2kj'} f_{j'+n}.$$

In the second step, we have used the periodicity property of the roots of unity $w^{2k(j'+n)} = w^{2kj'} w^{2kn} = w^{2kj'} \cdot 1$ (see also Eq. 7.7). Therefore, the first and second sum in Eq. 7.11 are over the same range and can be combined to:

$$2n\tilde{c}_{2k} = \sum_{j=0}^{n-1} (w^2)^{kj} (f_j + f_{j+n}). \tag{7.12}$$

Note that $f_{j+n} \neq f_j$ because $n = N/2$ is only half the period of the function. Comparing Eq. 7.12 above with Eq. 7.5, we see that Eq. 7.12 is also a DFT, but one of size $n = \frac{N}{2}$ on the data points $(f_j + f_{j+n})$. For the odd coefficients, a similar derivation (omitted here) yields:

$$2n\tilde{c}_{2k-1} = \sum_{j=0}^{n-1} (w^2)^{kj} (f_j - f_{j+n}) w^j, \tag{7.13}$$

which is a DFT of size $n = \frac{N}{2}$ on the data points $(f_j - f_{j+n})w^j$.
Computing the Fourier coefficients via Eqs. 7.12 and 7.13 instead of Eq. 7.5 yields the same result in $O(2 \cdot N^2/4) = O(N^2/2)$, i.e., in half the operations. This can now be iterated. The vectors $\vec{\tilde{c}}_{2k}$ and $\vec{\tilde{c}}_{2k-1}$ can again be split into odd and even parts each, leading to four DFTs of size $N/4$ that compute the result in a quarter of the time.
For $N = 2^m$, i.e., if the number of collocation points is a power of two, this binary decomposition can be iterated $m = \log_2(N)$ times. Then, instead of computing one DFT of size N, we compute N DFTs of size 1. Computing a DFT of size 1 requires $\mu(1) = 0$ complex multiplications, where $\mu(n)$ denotes the number of complex multiplications required for a DFT of size n. In each step of the recursive decomposition, we require $n = \frac{N}{2}$ complex multiplications to compute the $\cdot w^j$ terms in the odd coefficients (see Eq. 7.13). Moreover, the problem of size $2n$ is computed by solving two problems of size n. Therefore, $\mu(2n) = 2\mu(n) + n$. For $N = 2^m$, this yields

$$\mu(N) = nm = n \log_2(N) = \frac{N}{2} \log_2(N) \in O(N \log_2 N)$$

as the number of complex multiplications required by the FFT algorithm. Already for small N, this is much less than the direct $O(N^2)$ algorithm requires, therefore enabling very efficient trigonometric interpolation. Note, however, that this classical formulation of the FFT is only possible if the collocation points are evenly spaced and their number is a power of two.

Image: IEEE
James William Cooley
* 18 September 1926
New York, NY, USA
† 29 June 2016
Huntington Beach, CA, USA

Image: wikipedia
John Tukey
* 16 June 1915
New Bedford, MA, USA
† 26 July 2000
New Brunswick, NJ, USA

7.4 Practical Considerations

When using FFT implementations in practice, a couple of things need to be kept in mind. The first is that the normalization of the problem can be done in different ways. Here, we have defined:

$$\vec{\tilde{c}}_N = \frac{1}{N}\mathbf{W}\vec{f}_N$$
$$\vec{f}_N = \overline{\mathbf{W}}\vec{\tilde{c}}_N,$$

that is, we chose to normalize the forward transform (the Fourier analysis) by multiplying with $\frac{1}{N}$. Alternatively, of course, it is also possible to compute unnormalized Fourier coefficients γ and normalize the inverse transform as:

$$\vec{\gamma}_N = N\vec{\tilde{c}}_N = \mathbf{W}\vec{f}_N$$
$$\vec{f}_N = \frac{1}{N}\overline{\mathbf{W}}\vec{\gamma}_N.$$

This second way is, for example, how MATLAB does it. Therefore, the MATLAB function `fft([f])` does not compute the Fourier coefficients $\tilde{c}_j$, but their unnormalized versions $\gamma_j = N\tilde{c}_j$. Likewise, one has to be careful to use unnormalized coefficients as an input to the MATLAB function `ifft()`, as otherwise wrong function values are synthesized.
A third choice is to equally distribute the normalization to both the forward and backward transform, thus

$$\vec{\beta}_N = = \frac{1}{\sqrt{N}}\mathbf{W}\vec{f}_N$$
$$\vec{f}_N = \frac{1}{\sqrt{N}}\overline{\mathbf{W}}\vec{\beta}_N.$$

This is often seen in theoretical works because with this choice the DFT fulfills the mathematical properties of a *unitary transformation*.
Another practical choice is whether the Fourier coefficients are indexed $-\frac{N}{2}, \ldots, \frac{N}{2}$ or $0, \ldots, N-1$. Both choices are found in practice, with computer codes having an obvious tendency to use the latter, whereas Eq. 7.9 used the former. The non-negative formulation is obtained from the symmetric one by setting $\frac{N}{2}+1, \ldots, N-1 = -\frac{N}{2}, \ldots, -1$, as illustrated in Fig. 7.2.
An important property of Fourier coefficients, which forms the basis of applications such as data compression, is that for smooth periodic functions f, only few Fourier coefficients are significantly bigger than zero. Therefore, data can be represented in a more compact form by computing the DFT and only storing the $M < N$ largest coefficients. When reconstructing the data from this compressed form, the missing coefficients are simply set to 0 in the inverse transform, using

$$\underbrace{\tilde{c}_0, \tilde{c}_1, \ldots,}_{\frac{M}{2}} \underbrace{0, 0, \ldots, 0, 0,}_{N-M} \underbrace{\ldots, \tilde{c}_M}_{\frac{M}{2}}$$

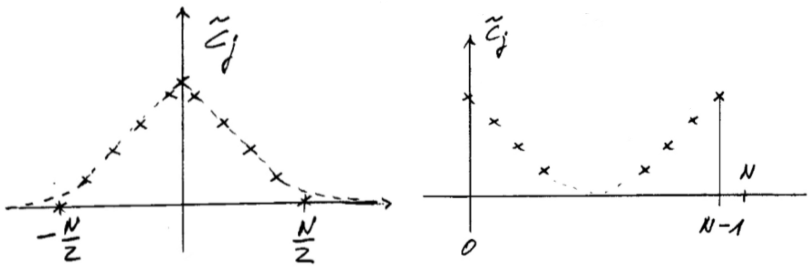


Figure 7.2: Two common indexing schemes used for Fourier coefficients: symmetric (left panel) and non-negative (right panel).

in the $0, \ldots, N-1$ indexing scheme (flipped in the other indexing scheme). A similar construction can also be used to approximately evaluate the trigonometric polynomial at intermediate positions $x \neq x_k$.

Chapter 8

Numerical Integration (Quadrature)

An important application of numerical methods is to approximate the value of an integral that cannot be solved analytically, or where the integrand is not given in closed form. The mathematical problem is to find the value

$$I = \int_a^b f(x)\,\mathrm{d}x \tag{8.1}$$

for a function $f(x) : [a, b] \to \mathbb{R}$. The basic idea of numerical integration is to approximate the integral by approximating the area under the graph of $y = f(x)$ between a and b. This idea goes back to Ancient Greece when Pythagoras and Hippocrates were concerned with finding a square that has the same area as a given circle, a problem known as the "quadrature of the circle". Because this can be computed as the integral (albeit the Ancient Greeks did not know calculus, they had a notion of geometric area) over a half-circle, the historic name for numerical integration is "quadrature".

We again only consider the scalar case. The numerical problem is to compute an approximation $\tilde{I} \approx I$ if the function f is either computable, i.e., can be evaluated for any given x (but need not be known in closed form), or its value is given at fixed collocation points x_i as f_i.

Unlike many of the problems considered so far, integration does not require $f(x)$ to be smooth. Indeed, the integral in Eq. 8.1 may exist and be well-defined even if $f(x)$ has discontinuities or singularities. For the numerical treatment of the problem, however, we still need to require that $f(x)$ is continuous. This can always be guaranteed by splitting an integral over a discontinuous function into multiple sub-integrals over continuous functions. For example, if the integrand $f(x)$ has a singularity or discontinuity at $x = s$, the problem can be decomposed as:

$$\int_a^b f(x)\,\mathrm{d}x = \int_a^{s^-} f(x)\,\mathrm{d}x + \int_{s^+}^b f(x)\,\mathrm{d}x,$$

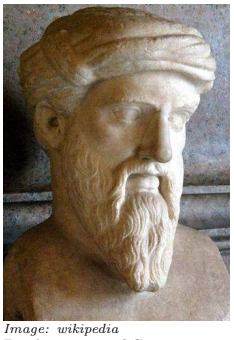


Image: wikipedia
Pythagoras of Samos
∗ ca. 570 B.C.
Samos, Ancient Greece
† ca. 495 B.C.
Croton, Ancient Greece
(now: Crotone, Italy)

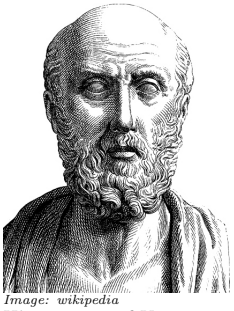


Image: wikipedia
Hippocrates of Kos
∗ ca. 460 B.C.
Kos, Ancient Greece
† ca. 370 B.C.
Larissa, Ancient Greece

where $s^- = s - \epsilon$ and $s^+ = s + \epsilon$ for some numerically negligible ϵ. The individual sub-integrals over continuous functions can then be approximated separately and the results added.

8.1 Rectangular Method

The simplest quadrature method approximates the area under the graph of $y = f(x)$ by a sum of rectangular boxes, as already realized by Pythagoras and illustrated in Fig. 8.1 for the case where all rectangles have the same width h. Then, the area of the k-th rectangle is given by

$$f(a + kh)h$$

for $k = 0, 1, 2, \ldots, N - 1$. Alternatively, one can also index the rectangles from 1 to N and use the value of the function at the right border of each rectangle to compute its area. The method is the same. Summing over all rectangles then approximates the integral as:

$$I \approx \tilde{I} = B(h) = f(a)h + f(a + h)h + f(a + 2h)h + \ldots + f(b - h)h$$

$$\tilde{I} = B(h) = \sum_{k=0}^{N-1} f(a + kh)h = h \sum_{k=0}^{N-1} f(a + kh). \qquad (8.2)$$

The value $B(h)$ is called the *rectangular approximation* (B for "block" or "box") of the integral with resolution h.

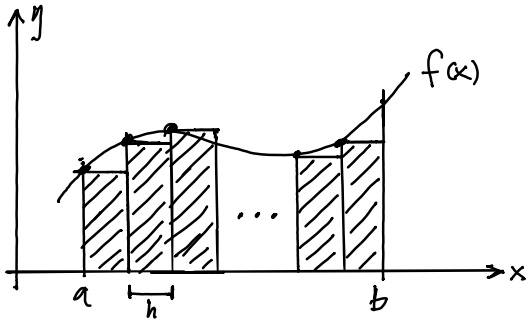


Figure 8.1: Approximating the area under the graph of $y = f(x)$ by a sum of rectangles of width h.

Considering again Fig. 8.1, we realize that another interpretation is that the rectangular method replaces the function $f(x)$ over the interval $[a, b]$ with a piecewise constant approximation of which the integral can trivially be computed. A piecewise constant approximation is identical to a polynomial interpolation

with polynomial degree $n = 0$, i.e., with constants. This immediately suggests more accurate approximations of the integral could be obtained by replacing $f(x)$ with its interpolation polynomial of higher degree. For polynomials, integrals can always be computed analytically, such that any such approximation would provide a viable algorithm. Therefore, numerical integration can be interpreted as an application of polynomial interpolation, followed by the analytical evaluation of the integral over the polynomial. This is also the reason why for the numerical treatment of the problem, it is convenient to require $f(x)$ to be continuous, so that polynomial approximation has a bounded error (see Section 6.5).

8.2 Trapezoidal Method

The next higher polynomial degree, $n = 1$, leads to a piecewise linear approximation of the integrand $f(x)$. This is illustrated in Fig. 8.2 for the case when $f(x)$ is approximated by a single linear function connecting the points $(a, f(a))$ and $(b, f(b))$ by a straight line. The integral over the linear approximation can then be analytically computed as the area under the line:

$$I \approx \tilde{I} = T := \frac{b-a}{2}(f(a) + f(b)).$$

This method is called *trapezoidal method* because the area under the graph of $f(x)$ is approximated by the area of the trapezoid $\tilde{I}$. It is a variant of *midpoint quadrature*. It was probably already discovered in ancient Babylon around 50 B.C. by an unspecified scientist who used it for integrating the velocity of the planet Jupiter (source: Mathieu Ossendrijver, Science 351(6272): 482–484, 2016).

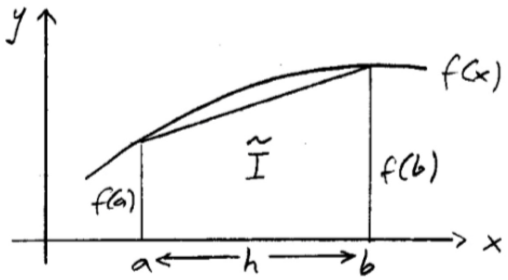


Figure 8.2: Illustration of quadrature where f is approximated by a linear function.

It is intuitive that the approximation quality improves if this strategy is applied piecewise to a finer subdivision of the interval $[a, b]$. Assume that the domain of integration is subdivided into N equally sized sub-intervals of width

$$h = \frac{b-a}{N},$$

leading to collocation points

$$x_k = a + kh, \quad f_k = f(x_k), \qquad k = 0, 1, \ldots, N,$$

as illustrated in Fig. 8.3. Clearly then, $a = x_0$ and $b = x_n$.

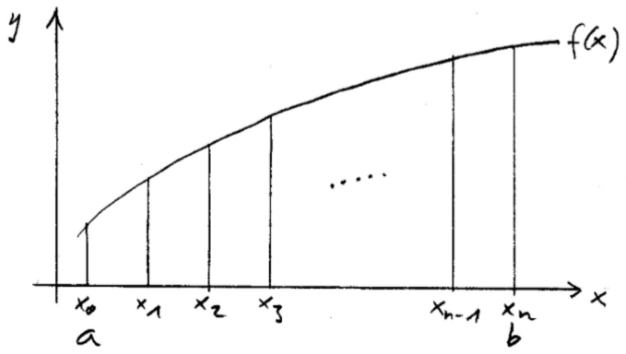


Figure 8.3: Subdivision of the domain of integration into N sub-intervals.

Summing the areas of the trapezoids in all sub-intervals, the *trapezoidal approximation* of the integral becomes:

$$T(h) = \frac{h}{2}(f_0 + f_1) + \frac{h}{2}(f_1 + f_2) + \ldots + \frac{h}{2}(f_{N-1} + f_N),$$

which can be simplified to:

$$\tilde{I} = T(h) := h\left[\frac{1}{2}f_0 + f_1 + f_2 + \cdots + f_{N-2} + f_{N-1} + \frac{1}{2}f_N\right]. \qquad (8.3)$$

Because this approximation is computed by replacing $f(x)$ with its linear interpolation polynomial, the error is bounded by:

$$|I - T(h)| \leq \frac{M}{8}(b-a)h^2$$

if $|f''(x)| \leq M$ for all $x \in [a, b]$. This directly follows from the error bound for polynomial approximation as given in Section 6.5. The polynomial approximation error bounds the error in the heights of the bars. The additional factor $(b - a)$ accounts for the cumulative width of the bars, yielding an error bound for the area.
In practice, M is usually not known and this formula is therefore not very useful to determine the h needed to achieve a desired approximation accuracy. A practical algorithm can be constructed by recursive interval refinement. For example, by successively halving the interval length,

$$h_0 = b - a, \quad h_1 = \frac{h_0}{2}, \quad h_2 = \frac{h_1}{2}, \quad \ldots,$$

we can compute gradually better approximations

$$T_0 = T(h_0), \quad T_1 = T(h_1), \quad T_2 = T(h_2), \quad \dots$$

This can be done without evaluating the function f more than once at any given location, by computing iteratively:

$$\begin{aligned}
s_0 &\coloneqq \frac{1}{2}(f(a) + f(b)) && T_0 = h_0 s_0 \\
s_1 &= s_0 + f(a + h_1) && T_1 = h_1 s_1 \\
s_2 &= s_1 + f(a + h_2) + f(a + 3h_2) && T_2 = h_2 s_2 \\
s_3 &= s_2 + f(a + h_3) + f(a + 3h_3) + f(a + 5h_3) + f(a + 7h_3) && T_3 = h_3 s_3 \\
&\vdots && \vdots
\end{aligned}$$

We can terminate this sequence as soon as the approximation does not significantly (as given by some tolerance) change any more. This provides a practical way of computing the trapezoidal approximation to a desired accuracy without priorly needing to know the h required to do so, and it computationally costs only marginally (for the intermediate multiplications with h_i) more than a direct evaluation of the trapezoidal sum from Eq. 8.3 for that h. The resulting algorithm for general N is given in Algorithm 11.

Algorithm 11 Trapezoidal Quadrature

procedure TRAPEZOIDAL(a, b, RTOL, ATOL) ▷ Domain interval $[a, b]$
  $h_0 = b - a$
  $s_0 = \frac{1}{2}(f(a) + f(b))$
  $T_0 = h_0 s_0$
  $N_0 = 0$
  $i = -1$
  repeat
    $i = i + 1$
    $h_{i+1} = \frac{h_i}{2}$
    $s_{i+1} = s_i + \sum_{j=0}^{N_i} f(a + (2j+1)h_{i+1})$
    $T_{i+1} = h_{i+1} s_{i+1}$
    $N_{i+1} = 2N_i + 1$
  until $|T_{i+1} - T_i| \leq |T_{i+1}|\text{RTOL} + \text{ATOL}$
end procedure

The main disadvantage of the trapezoidal method is its slow convergence, i.e., the large number of iterations of the algorithm (equivalently: the small h) required to achieve high accuracy.

8.3 Simpson's Rule

Further increasing the polynomial degree to $n > 1$ leads to quadrature schemes of successively higher orders of convergence. Since interpolation is done piecewise (like Hermite interpolation) with sub-interval widths scaling inversely proportional with polynomial degree, the ringing artifact discussed in Section 6.5 is not an issue.

The next-higher order scheme is obtained when approximating $f(x)$ with its quadratic interpolation polynomial $P_2(x)$. Determining $P_2(x)$ requires three collocation points, as illustrated in Fig. 8.4. The easiest is to place the additional collocation point c at the center of the integration interval, thus $c = \frac{1}{2}(a+b)$.

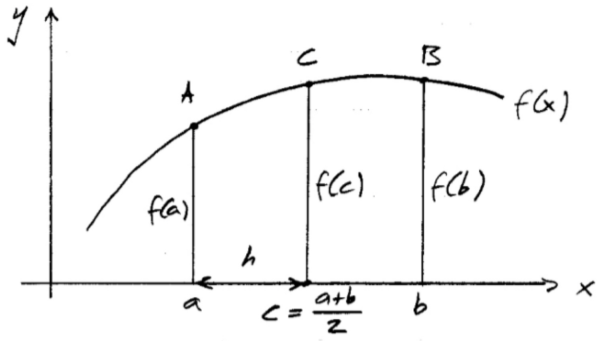


Figure 8.4: Using three collocation points to approximate $f(x)$ by an interpolation polynomial of degree 2.

Because $P_2(x)$ solves the polynomial interpolation problem for $f(x)$, we have:

$$P_2(a) = f(a), \;\; P_2(b) = f(b), \;\; P_2(c) = f(c).$$

The integral over the interpolation polynomial can then be computed as:

$$S \coloneqq \int_a^b P_2(x)\,\mathrm{d}x = h \int_{-1}^{1} \underbrace{P_2(a + h(t+1))}_{Q_2(t)}\,\mathrm{d}t,$$

with $h = \frac{b-a}{2}$, $x = a + h(t+1)$, and therefore $\mathrm{d}x = h\mathrm{d}t$. This transformation of variables rescales the domain of integration to $[-1, 1]$, regardless of the original domain of the integral. To determine this integral, we thus need to find the rescaled polynomial $Q_2(t)$, which can be done by solving the three-point interpolation problem. Therefore, $Q_2(t)$ is a polynomial of degree ≤ 2,

$$Q_2(t) = \alpha t^2 + \beta t + \gamma,$$

satisfying

$$\begin{aligned} Q_2(-1) = P_2(a) = f(a) &= \alpha - \beta + \gamma, \\ Q_2(0) = P_2(c) = f(c) &= \gamma, \\ Q_2(1) = P_2(b) = f(b) &= \alpha + \beta + \gamma. \end{aligned}$$

Solving this linear system of equations analytically, we find

$$\alpha = \frac{f(b) - 2f(c) + f(a)}{2}, \quad \beta = \frac{f(b) - f(a)}{2}, \quad \gamma = f(c).$$

Now we can solve the integral:

$$\begin{aligned} \int_{-1}^{1} Q_2(t)\,\mathrm{d}t &= \alpha \int_{-1}^{1} t^2\,\mathrm{d}t + \beta \int_{-1}^{1} t\,\mathrm{d}t + \gamma \int_{-1}^{1} \mathrm{d}t = \frac{2}{3}\alpha + 2\gamma \\ &= \frac{1}{3}f(b) - \frac{2}{3}f(c) + \frac{1}{3}f(a) + 2f(c), \end{aligned}$$

leading to the result

$$S = \frac{h}{3}(f(a) + 4f(c) + f(b)). \tag{8.4}$$

Image: JSTOR.org
Thomas Simpson
∗ 20 August 1710
Sutton Cheney, England
† 14 May 1761
Market Bosworth, England

Like before, we can improve the approximation by not only subdividing the integration domain into two sub-intervals, but into $2N$. We write $2N$ instead of N in order to highlight that the number of sub-intervals for the Simpson method always has to be even. Then,

$$h = \frac{b-a}{2N}$$

and the collocation points are

$$x_k = a + kh, \quad f_k = f(x_k), \qquad k = 0, 1, \ldots, 2N.$$

The Simpson approximation for equidistant[1] collocation points then is:

$$S(h) = \frac{h}{3}(f_0 + 4f_1 + f_2) + \frac{h}{3}(f_2 + 4f_3 + f_4) + \cdots + \frac{h}{3}(f_{2N-2} + 4f_{2N-1} + f_{2N}),$$

which simplifies to:

$$\tilde{I} = S(h) \coloneqq \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \cdots + 2f_{2N-2} + 4f_{2N-1} + f_{2N}). \tag{8.5}$$

Image: wikipedia
Johannes Kepler
∗ 27 December 1571
Weil der Stadt, Holy Roman Empire (now: Germany)
† 15 November 1630
Regensburg, Holy Roman Empire (now: Germany)

[1] There is also a version of Simpson's rule that uses sub-intervals of different length, concentrating on areas where the integrand varies more quickly (*adaptive Simpson rule*) and a version for completely irregular collocation points.

For the Simpson method, the approximation error is

$$|I - S(h)| \leq \frac{M}{180}(b-a)h^4$$

if $|f^{(4)}(x)| \leq M$ for $x \in [a,b]$. Therefore, Simpson's method is of convergence order 4, whereas the trapezoidal method is of order 2 and the rectangular method of order 1. Simpson's method gains an additional order over the order 3 that is guaranteed by the use of quadratic interpolation, because the collocation points are symmetric in the interval. This is not a contradiction, because the polynomial approximation error is an upper bound and special cases (like here symmetric point distributions) can be below.
In practice, the h required to achieve a certain desired accuracy is again not priorly known, but successively refined values can be computed by iteratively halving the sub-intervals until the change falls below a given tolerance. The algorithm is based on the observation that the trapezoidal values and the Simpson values are related through:

$$S_i = \frac{4T_i - T_{i-1}}{3}, \qquad i = 1, 2, 3, \ldots \tag{8.6}$$

Therefore, Algorithm 11 can be used to compute the sequence of trapezoidal approximations $T_0, T_1, T_2, \ldots$ from which the Simpson approximations are computed using the above formula. The resulting Simpson algorithm is given in Algorithm 12. It is generally attributed to Thomas Simpson (not to be confused with the arctic explorer and convicted murderer by the same name), although the algorithm was already described and used by Johannes Kepler 100 years before Simpson, which is why it is also sometimes called *Kepler's Rule* in the literature (in German: "Kepler'sche Fassregel").

8.4 Romberg's Method

As we have seen in the previous section, the Simpson values can be expressed as linear combinations of trapezoidal values. This suggests that using some other linear combination of Simpson values might yield an even higher-order quadrature. This is indeed the case, as was realized by Werner Romberg in 1955, who applied Richardson extrapolation (Lewis Fry Richardson, 1911) recursively to trapezoidal values in order to systematically derive higher-order quadrature formulas.
The Romberg method is based on the observation that the trapezoidal values can be expanded into a series

$$T(h) = I + c_1 h^2 + c_2 h^4 + c_3 h^6 + \ldots, \tag{8.7}$$

where I is the exact value of the integral. Due to the symmetric distribution of collocation points, all odd orders vanish. Accounting for terms up to and including order h^{2n}, the approximation error is of order h^{2n+2}, provided that

Algorithm 12 Simpson Quadrature

```
1: procedure SIMPSON(a, b, RTOL, ATOL)          ▷ Domain interval [a, b]
2:     h_0 = b − a
3:     h_1 = h_0/2
4:     s_0 = ½(f(a) + f(b))
5:     s_1 = s_0 + f(a + h_1)
6:     T_1 = h_1 s_1
7:     S_1 = ⅓(4T_1 − h_0 s_0)
8:     N_1 = 1
9:     i = 0
10:    repeat
11:        i = i + 1
12:        h_{i+1} = h_i/2
13:        s_{i+1} = s_i + Σ_{j=0}^{N_i} f(a + (2j + 1)h_{i+1})
14:        T_{i+1} = h_{i+1} s_{i+1}
15:        S_{i+1} = ⅓(4T_{i+1} − T_i)
16:        N_{i+1} = 2N_i + 1
17:    until |S_{i+1} − S_i| ≤ |S_{i+1}|RTOL + ATOL
18: end procedure
```

f is sufficiently smooth, i.e., sufficiently many times continuously differentiable everywhere, such that a finite upper bound on $|f^{(2n+2)}|$ exists.
The idea of Richardson extrapolation is to exploit the existence of Eq. 8.7 to eliminate (unknown) error terms and thus construct higher-order schemes by forming linear combinations of series expansions for different values of h.
For example, consider the expansions from Eq. 8.7 for h and $h/2$:

$$T(h) = I + c_1 h^2 + c_2 h^4 + c_3 h^6 + \dots,$$
$$T\left(\frac{h}{2}\right) = I + \frac{1}{4} c_1 h^2 + \frac{1}{16} c_2 h^4 + \frac{1}{64} c_3 h^6 + \dots.$$

Multiplying the second equation by 4 and subtracting the first equation from it yields:

$$4T\left(\frac{h}{2}\right) - T(h) = 3I - \frac{3}{4} c_2 h^4 - \frac{15}{16} c_3 h^6 - \dots$$

and thus:

$$\frac{4T(\frac{h}{2}) - T(h)}{3} = I - \frac{1}{4} c_2 h^4 + O(h^6), \tag{8.8}$$

which according to Eq. 8.6 is the Simpson value for resolution $h/2$:

$$S\left(\frac{h}{2}\right) = \frac{4T\left(\frac{h}{2}\right) - T(h)}{3} = \frac{T\left(\frac{h}{2}\right) - 4^{-1} T(h)}{1 - 4^{-1}}. \tag{8.9}$$

The second way of writing the linear combination will become clear when we look at the next-higher order.

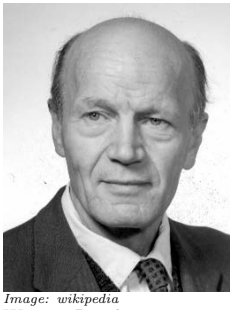


Image: wikipedia
Werner Romberg
∗ 16 May 1909
Berlin, German Empire
† 5 February 2003
Heidelberg, Germany

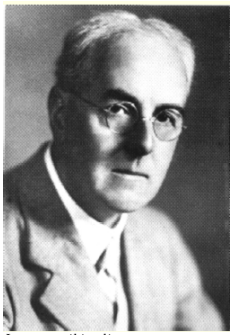


Image: wikipedia
Lewis Fry Richardson
∗ 11 October 1881
Newcastle upon Tyne, UK
† 30 September 1953
Kilmun, Scotland, UK

From Eq. 8.8 it is clear that the Simpson values have the asymptotic series expansion (confirming their fourth-order convergence):

$$S\left(\frac{h}{2}\right) = I - \frac{1}{4}c_2h^4 - \frac{5}{16}c_3h^6 + \dots$$

The next Richardson extrapolation step thus aims to eliminate the term of order h^4 by forming a suitable linear combination of:

$$S\left(\frac{h}{2}\right) = I - \frac{1}{4}c_2h^4 - \dots$$
$$S\left(\frac{h}{4}\right) = I - \frac{1}{64}c_2h^4 - \dots$$

Multiplying the second equation by 16 and subtracting the first from it eliminates the fourth-order term and yields:

$$16S\left(\frac{h}{4}\right) - S\left(\frac{h}{2}\right) = 15I + O(h^6)$$

and thus:

$$\frac{16S\left(\frac{h}{4}\right) - S\left(\frac{h}{2}\right)}{15} = \frac{S\left(\frac{h}{4}\right) - 4^{-2}S\left(\frac{h}{2}\right)}{1 - 4^{-2}} = I + O(h^6), \tag{8.10}$$

which is a sixth-order accurate quadrature formula. Comparing the expressions in Eqs. 8.9 and 8.10 immediately suggests a regularity.

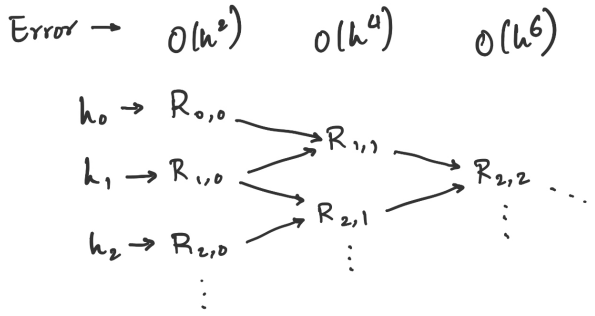


Figure 8.5: Illustration of the sequence of calculations in the Romberg method.

Indeed, the scheme can be continued as illustrated in Fig. 8.5. It starts by computing the sequence of successively halved interval sizes

$$h_0 = b - a, \quad h_1 = \frac{h_0}{2}, \quad h_2 = \frac{h_1}{2}, \quad \dots,$$

each giving rise to a row in the scheme of Fig. 8.5. For each row, we then compute the corresponding trapezoidal value using Algorithm 11. This defines the first column of the scheme with values

$$R_{i,0} = T_i, \qquad i = 0, 1, 2, \ldots \tag{8.11}$$

According to the regularity suggested by the above two steps of Richardson extrapolation, the subsequent columns of the scheme are then computed as:

$$R_{i,k} = \frac{R_{i,k-1} - 4^{-k} R_{i-1,k-1}}{1 - 4^{-k}}, \qquad k = 1, 2, \ldots, i. \tag{8.12}$$

This is the *Romberg method* for computing successively higher-order approximations to the integral value. If all coefficients in the series expansion of Eq. 8.7 $c_k \neq 0$, then each column of this scheme converges to the correct value I faster (i.e., at higher convergence rate) than all previous columns. Furthermore, the sequence of values along each diagonal (from top-left to bottom-right) converges faster than the sequence of values along any column (not proven here).
The calculation is terminated as soon as $|R_{i,i} - R_{i-1,i-1}| \leq |R_{i,i}|\text{RTOL} + \text{ATOL}$, which is the usual stopping condition. Then $\tilde{I} = R_{i,i}$ is an approximation of I to accuracy $O(h^{2i+2})$.
However, one has to be careful not to proceed too far in the Romberg scheme, i.e., not to use tolerances that are too small. This is because for large i, it may happen that the approximations get worse again, which could prevent the algorithm from ever reaching the tolerance. The reason for this can be either that f is not sufficiently many times continuously differentiable (in which case the terms in the Taylor expansion for Richardson extrapolation don't exist) or that the degree of the interpolation polynomial becomes too high with respect to the interval size h_i (note that here the two are decoupled), leading to the ringing phenomenon discussed in Section 6.5. In practice, the Romberg scheme is usually limited to $i_{\max} \approx 6$ in order to prevent these problems.
The Romberg schemes also only work for equidistant collocation points. Finally, while the Romberg scheme is easy to implement, it is computationally not particularly efficient, because many evaluations of the function f are required to compute the trapezoidal values to start the scheme, and a small h may be required to resolve strongly varying functions.

8.5 Gauss Quadrature

The Romberg scheme and its computational inefficiency raises the question what the *optimal* quadrature formula would be, i.e., the formula that achieves the highest possible accuracy with a given number of collocation points or, equivalently, that needs the smallest number of collocation points to achieve a given accuracy. It is intuitive that placing the collocation points at regular distances may not always be the best. If the integrand varies little in some areas of the

integration domain, but strongly in others, one may better place collocation points more densely in areas of strong variation. Using the same overall number of collocation points N, the accuracy could then be higher than in equidistant methods. This is the idea behind *adaptive* quadrature methods, of which there exist many, including the *adaptive Simpson rule*. Therefore, using the additional degree of freedom of how to place the collocation points can improve the accuracy of quadrature.

To address the question how accurate any adaptive quadrature can be at best, Carl Friedrich Gauss realized that any quadrature formula is a weighted sum of the N collocation values and can hence be written in general as:

$$Q_N = \sum_{j=1}^{N} w_j f_j \tag{8.13}$$

with ordered collocation points

$$a = x_1 < x_2 < \cdots < x_N = b,$$

collocation values

$$f_1 = f(x_1), \ldots, f_N = f(x_N),$$

and quadrature weights

$$w_1, w_2, \ldots, w_N.$$

So far, we considered fixed, equidistant $x_j = x_k = a + kh$, $k = 0, \ldots, N-1$, and have determined the weights w_j such that $Q_N \approx I$ to some order of accuracy. This gave rise to the Romberg family of methods with the trapezoidal and Simpson methods as its first two members. Now, we want to consider the case where *both* the x_j and the w_j can be freely chosen in order to achieve higher orders of accuracy with the same *number* of collocation points. This allows for collocations points to be arbitrarily spaced, which is why we index them by j now in order to differentiate from the evenly spaced x_k of above.

We start by noting that any problem as defined in Eq. 8.1 and be formulated as an integral over the inverval $[-1, 1]$, hence

$$I = \int_a^b g(t)\,\mathrm{d}t = \int_{-1}^{1} f(x)\,\mathrm{d}x$$

by the change of variables

$$t = \frac{b-a}{2}x + \frac{a+b}{2}, \quad x = \frac{2}{b-a}t - \frac{a+b}{b-a},$$

as

$$I = \int_{-1}^{1} \underbrace{g\left(\frac{b-a}{2}x + \frac{a+b}{2}\right)\frac{b-a}{2}}_{f(x)}\,\mathrm{d}x.$$

Therefore, for simplicity of notation, we restrict the discussion to integrals on the interval $[-1, 1]$ without loss of generality.

Before considering the question of what the *optimal* quadrature formula is, we need to sharpen a bit the concept of how we quantify accuracy of quadrature.

Accuracy of quadrature schemes

There are two ways of defining the accuracy of a quadrature scheme: (1) by the asymptotic decay of the truncation error, or (2) by the complexity of functions that are still exactly integrated. So far, in the Romberg view, we have considered the asymptotic decay of the truncation error from the series expansion of the quadrature values. This has led to statements like, e.g. for the trapezoidal method, "the quadrature error decreases as $O(h^2)$ for $h \to 0$." While this tells us how the error *scales* with h, and that it *eventually* converges for *sufficiently* small h, it makes no statement about the accuracy for any given finite h. Therefore, Gauss introduced an alternative way of defining accuracy of quadrature as the largest degree of a polynomial that is still integrated *exactly*. We therefore define:

Definition 8.1 (Degree of accuracy)**.** *The* degree of accuracy m *of a quadrature scheme* Q_N *is the largest degree of a polynomial* $p(x)$ *for which the integral* $I = \int_{-1}^{1} p(x)\,\mathrm{d}x$ *is computed exactly, i.e.,* $|Q_N - I| = 0$ *for all* $p(x)$ *with* $\deg p(x) \leq m$.

The scaling of the error with h is referred to as the *order of accuracy*. The degree of accuracy and the order of accuracy are not the same.

For quadrature methods from the Romberg family, i.e., methods based on replacing the integrand with its interpolation polynomial, the order of accuracy is always one more than the degree of accuracy. Moreover, the degree of accuracy m of a Romberg method is related to the degree n of the approximating interpolation polynomial. Consider the following two examples:

Example 8.1. *The trapezoidal method with* $h = 2$ *on the interval* $[-1, 1]$ *is given by:*

$$T(h = 2) = Q_2 = f(-1) + f(1).$$

Now, consider as integrand a polynomial of degree $n = 1$*:*

$$f(x) = p_1(x) = a_0 x + a_1$$

with its exact integral:

$$\int_{-1}^{1} p_1(x)\,\mathrm{d}x = 2a_1.$$

For the trapezoidal approximation, we find:

$$T(2) = Q_2 = p_1(-1) + p_1(1) = -a_0 + a_1 + a_0 + a_1 = 2a_1,$$

which is exact, as expected from the fact that a linear function has $f'' = 0$ *and hence the error of the trapezoidal method vanishes. Since this is not the case for any more for* $n = 2$*, the degree of accuracy of the trapezoidal method is* $m = 1$*. The degree of the approximating interpolation polynomial is* $n = 1 = m$.

Example 8.2. *For Simpson's method on the interval* $[-1, 1]$*, we have:*

$$S(h = 1) = Q_3 = \frac{1}{3}(f(-1) + 4f(0) + f(1)).$$

Consider as integrand a cubic polynomial

$$f(x) = p_3(x) = a_0x^3 + a_1x^2 + a_2x + a_3$$

with exact integral:

$$\int_{-1}^{1} p_3(x)\,\mathrm{d}x = \frac{2}{3}a_1 + 2a_3.$$

For the Simpson approximation, we find:

$$\begin{aligned} p_3(-1) &= -a_0 + a_1 - a_2 + a_3 \\ p_3(0) &= a_3 \\ p_3(1) &= a_0 + a_1 + a_2 + a_3 \\ &\Longrightarrow S(1) = Q_3 = \frac{1}{3}(2a_1 + 6a_3), \end{aligned}$$

which is exact, as expected from the error bound of Simpson's method when $f^{(4)} = 0$*. Therefore, Simpson's method integrates polynomials of degree* $n = 3$ *exactly. Since this is not the case any more for* $n = 4$*, the Simpson method has a degree of accuracy of* $m = 3$*. The degree of the approximating interpolation polynomial is* $n = 2 = m - 1$*.*

The following interesting facts have been proven (elsewhere):

- Romberg methods based on an interpolation polynomial of even degree n have a degree of accuracy of $m = n + 1$. Romberg methods based on an interpolation polynomial of odd degree n have a degree of accuracy of $m = n$.

- The degree of accuracy of any quadrature scheme with N collocation points is at most $m \leq 2N - 1$.

- There exists exactly one quadrature scheme with N collocation points $x_j \in [-1, 1]$ that reaches the maximum degree of accuracy $m = 2N - 1$.

These facts are interesting because they tell us that: (1) one can do better than the Romberg methods, since $2N - 1 > N + 1$ for all $N > 2$, and (2) there exists one *unique* quadrature formula that achieves the maximum degree of accuracy. This quadrature method of optimal degree of accuracy is called *Gauss quadrature*. We describe it here without deriving or proving it. In N-point Gauss quadrature, the collocation points x_j are given by the roots of the N-th Legendre polynomial $L_N(x)$, and the quadrature weights w_j are given by:

$$w_j = \int_{-1}^{1} \prod_{\substack{k=1 \\ k \neq j}}^{N} \left(\frac{x - x_k}{x_j - x_k} \right)^2 \mathrm{d}x \quad > 0, \qquad j = 1, 2, \ldots, N. \tag{8.14}$$

The Legendre polynomials can be computed from the recursion:

$$L_0(x) = 1, \tag{8.15}$$

$$L_1(x) = x, \tag{8.16}$$

$$L_{k+1}(x) = \frac{2k+1}{k+1} x L_k(x) - \frac{k}{k+1} L_{k-1}(x), \quad k \geq 2. \tag{8.17}$$

Image: wikipedia
Adrien-Marie Legendre
∗ 18 September 1752
Paris, Kingdom of France
† 9 January 1833
Paris, France

The roots of these polynomials are known analytically up to $k \leq 5$ and have been computed numerically for higher degrees. The values can be found tabulated. All non-zero roots of Legendre polynomials are pairwise symmetric around $x = 0$. The quadrature weights for both roots of a symmetric pair are identical. Therefore, it is in practice enough to store the non-negative x_j and compute their weights w_j from Eq. 8.14. The symmetric negative ones can be mirrored at runtime. Because Gauss quadrature normalizes everything into the standard interval $[-1, 1]$, the (x_j, w_j) are always the same, regardless of the integral to be approximated. They can therefore be pre-computed and stored in tables once and for all, rendering Gauss quadrature computationally very efficient and fast. While Gauss quadrature is much more accurate than Romberg schemes, it requires that the integrand $f(x)$ can be evaluated at arbitrary locations. This may not be the case in applications where $f(x)$ is only given at discrete data points. Clearly, interpolating from these points to the Gauss points x_j would again amount to Romberg quadrature and would therefore be meaningless. In such cases, the classic Romberg schemes are preferred.
Interestingly, while Gauss quadrature is *the* scheme of maximum degree of accuracy, its order of accuracy is not known. An error estimation is therefore not available. If this is required, Romberg methods are again the way to go.
Romberg and Gauss methods are related in the sense that the Gauss quadrature formula with N collocation points x_j corresponds to the integral from -1 to 1 over the Legendre interpolation polynomial $L_{N-1}(x)$ approximated on the collocation points $(x_j, f(x_j))$. Because Legendre polynomials do not suffer from ringing artifacts, however, the maximum degree N is no longer limited, and Gauss quadrature also works for large N without problems.

Chapter 9

Numerical Differentiation

Now that we know how to numerically approximate integrals, we consider the opposite operation, namely the numerical approximation of derivatives. This is a central topic in many applications of numerical computing, as it allows computing numerical solutions to differential equations, giving rise to the entire sub-field of numerical analysis.

Given a function $f(x) : \mathbb{R} \to \mathbb{R}$, which is sufficiently many times continuously differentiable, the goal is to approximate $f^{(\nu)}(z)$, the ν-th ($\nu > 0$) derivative of f evaluated at a given location z. This is a well-defined real number that can be numerically computed.

Numerical differentiation is required whenever *(i)* $f(x)$ is only known at discrete collocation points or given as a data table, *(ii)* $f(x)$ is not available in closed or symbolic form but can be evaluated for any x, *(iii)* analytically computing the derivative would be too complicated, or *(iv)* differential operators in differential equations are to be discretized.

Numerical differentiation is tricky. While replacing the function $f(x)$ with its interpolation polynomial $p(x)$ was a good strategy for quadrature, it does not provide a viable approach to differentiation. Indeed, a small $|p(x) - f(x)|$ does not imply small $|p'(x) - f'(x)|$. This is illustrated in Fig. 9.1, where a wavy function $f(x)$ is well approximated (in value) by a polynomial $p(x)$. Clearly, the areas under the two curves are also almost the same, making $p(x)$ a good choice for integration. The slopes of $f(x)$ and $p(x)$ at any location z, however, can be very different. Therefore, simply using the derivative of the interpolation polynomial $p(x)$ as an approximation of the derivative of $f(x)$ is not a good idea.

Numerical differentiation therefore is fundamentally different from numerical integration, and it is tricky because it amplifies noise. If the data points though which $f(x)$ is given are noisy, the approximation of the derivative is even more noisy. Due to its practical importance, however, numerical differentiation has attracted a lot of attention and there exists a wealth of different approaches, including:

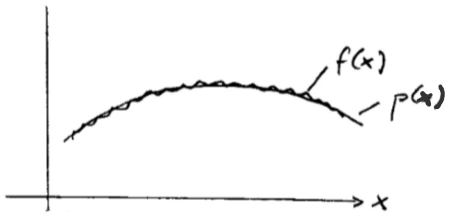


Figure 9.1: A function $f(x)$ is well approximated by its interpolation polynomial $p(x)$, but its derivative is not.

1. Spectral methods that use discrete Fourier transforms to approximate derivatives, because derivatives in real space amount to multiplications with the wave number in Fourier space. Since this can use the high frequencies, too, and Fourier interpolation is optimal in the least-squares sense (see Chapter 7), the problem in Fig. 9.1 does not occur. In fact, when using all frequencies up to the Nyquist limit of the discretization, spectral methods are exact to machine precision. They are, however, computationally more costly ($N \log N$) than the methods below and more difficult to efficiently implement on parallel computer architectures.

2. Galerkin methods that approximate derivatives in the weak sense by finding the coefficients of an expansion into a finite series of basis functions.

3. Collocation methods that compute derivative approximations by forming linear combinations of function values at given collocation points.

4. Automatic Differentiation (AD) methods that generate code to numerically evaluate the symbolic derivative.

Examples of (1) include spectral solvers and Ewald methods. Examples of (2) include finite-element methods, (weak) particle methods (e.g., particle strength exchange (PSE), reproducing kernel particle methods (RKPM)), and partition-of-unity methods. Examples of (3) include finite-difference methods, finite-volume methods, and (strong) particle methods (e.g., smoothed particle hydrodynamics (SPH), moving least squares (MLS), discretization-corrected particle strength exchange (DC-PSE)). Here, we only consider the most basic class of finite-difference methods, as its derivation and analysis are easily accessible and allow introducing the basic concepts without much notational overhead. However, we keep in mind that this is only the tip of the iceberg, and more elaborate finite-difference methods (e.g., less sensitive to noise or with solution-adaptive stencils) as well as a wealth of fundamentally different approaches also exist.

9.1 Finite Differences

Finite differences as approximations of derivatives go back to Sir Isaac Netwon, who used limits over divided differences (also called *difference quotients*) to

derive calculus. The basic tool for deriving any finite-difference method is the Taylor expansion (Brook Taylor, 1715) of $f(z)$ around a slightly shifted location $z+h$, $h>0$:

$$f(z+h) = f(z)+\frac{h}{1!}f'(z)+\frac{h^2}{2!}f''(z)+\cdots+\frac{h^k}{k!}f^{(k)}(z)+\frac{h^{k+1}}{(k+1)!}f^{(k+1)}(\xi_+) \quad (9.1)$$

for $k = 1, 2, 3, \ldots$ and an unknown $\xi_+ \in (z, z+h)$ in the remainder term. Notice that the Taylor expansion contains all derivatives of f at the desired location z. The trick now is to form a linear combination of multiple such Taylor expansions for different shifted locations in order to isolate the term that contains the desired derivative.
For example, we can also shift by h in the negative direction and find:

$$f(z-h) = f(z) - \frac{h}{1!}f'(z) + \frac{h^2}{2!}f''(z) - \ldots \quad (9.2)$$

Subtracting this series from the previous one, we get:

$$f(z+h) - f(z-h) = 2hf'(x) + 2c_1h^3 + 2c_2h^5 + \cdots + 2c_{m-1}h^{2m-1} + 2hR_m(h), \quad (9.3)$$

with

$$c_k = \frac{1}{(2k+1)!}f^{(2k+1)}(z), \qquad k = 1, 2, \ldots, m-1$$

and the remainder term

$$R_m(h) = \frac{h^{2m+1}}{(2m+1)!}f^{(2m+1)}(\xi)$$

for some unknown ξ. The defines the *symmetric finite difference for the first derivative* with resolution h, also sometimes called *central finite difference* or *centered finite difference*:

$$\Delta_h^{(1)}(z) = \frac{f(z+h) - f(z-h)}{2h} \quad (9.4)$$

for which Eq. 9.3 becomes:

$$\Delta_h^{(1)}(z) = f'(z) + c_1h^2 + c_2h^4 + \cdots + c_{m-1}h^{2m-2} + R_m(h)$$

for $m = 2, 3, 4, \ldots$. The finite difference in Eq. 9.4 thus has an approximation error of order $O(h^2)$, i.e., it is asymptotically second-order accurate. Notationally, we denote finite differences by a capital Greek letter Delta (for "difference"), superscripted with the derivative it approximates and subscripted with the resolution parameter.
Clearly, the above central finite difference is not the only possibility of isolating the first derivative in the Taylor expansion. From Eq. 9.1, we can also see that

$$\frac{f(z+h) - f(z)}{h} = f'(z) + \frac{1}{2}hf''(z) + O(h^2)$$

or from Eq. 9.2 that:

$$\frac{f(z) - f(z-h)}{h} = f'(z) - O(h).$$

These are the *forward* and *backward finite differences* for the first derivative, respectively. As we can see from their expansions, they are approximations with an error of order $O(h)$, thus only first-order accurate.
In the same way, we can also see that the linear combination of Taylor expansions

$$f(z+h) - 2f(z) + f(z-h) = h^2 f''(z) + \tilde{c}_1 h^4 + \tilde{c}_2 h^6 + \dots$$

defines the *symmetric finite difference for the second derivative* with resolution h, as:

$$\Delta_h^{(2)}(z) = \frac{f(z+h) - 2f(z) + f(z-h)}{h^2} \tag{9.5}$$

for which

$$\Delta_h^{(2)}(z) = f''(z) + \tilde{c}_1 h^2 + \tilde{c}_2 h^4 + \dots + \tilde{c}_{m-1} h^{2m-2} + O(h^{2m}).$$

Therefore, this finite difference is again a second-order accurate approximation. Equivalently, we can rewrite this as:

$$\Delta_h^{(2)}(z) = \frac{f(z+2h) - 2f(z) + f(z-2h)}{(2h)^2}$$

using a shift of $2h$. Then, the denominator is the square of the denominator of Eq. 9.4, which allows us to see a regularity.
Indeed, a general symmetric finite difference approximation for the ν-th derivative is:

$$\begin{aligned} \Delta_h^{(\nu)}(z) &= \frac{1}{(2h)^\nu} \sum_{j=0}^{\nu} (-1)^j a_j f(z + b_j h) \\ a_j &= \binom{\nu}{j} \\ b_j &= \nu - 2j. \end{aligned} \tag{9.6}$$

The absolute values of the *stencil weights* a_j are given by the binomial coefficients and can hence be determined as the ν-th row of the Pascal triangle (Blaise Pascal, *Traité du triangle arithmétique*, 1654), calling the tip of the triangle the 0-th row (see Fig. 9.2).
From the general formula in Eq. 9.6, we can for example directly find:

$$\Delta_h^{(3)}(z) = \frac{1}{(2h)^3}[f(z+3h) - 3f(z+h) + 3f(z-h) - f(z-3h)],$$

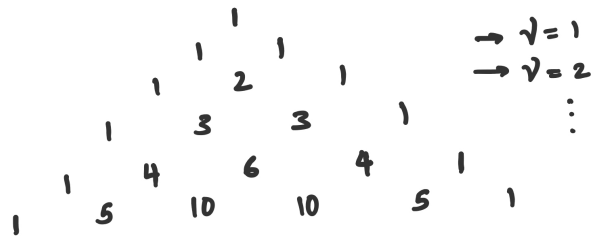


Figure 9.2: The absolute values of the stencil weights for second-order symmetric finite differences are given by the rows of the Pascal triangle.

$$\Delta_h^{(4)}(z) = \frac{1}{(2h)^4}[f(z+4h) - 4f(z+2h) + 6f(z) - 4f(z-2h) + f(z-4h)].$$

Image: wikipedia
Blaise Pascal
* 19 June 1623
Clermont-Ferrand,
Kingdom of France
† 19 August 1662
Paris, Kingdom of France

For all symmetric finite differences from this family, we have

$$\Delta_h^{(\nu)}(z) = f^{(\nu)}(z) + c_1^{(\nu)}h^2 + c_2^{(\nu)}h^4 + \dots.$$

Therefore, they are all second-order accurate.
While this way of computing finite differences for higher derivatives is simple and intuitive, it is numerically problematic because for small h the resulting formulae are prone to numerical extinction. Consider for example a case where $f^{(4)}(z) \approx 1$. The above symmetric finite difference for $h = 10^{-3}$ then yields:

$$1 \approx \Delta_h^{(4)}(z) = \frac{10^{12}}{16}[\underbrace{f(z+4h) + 6f(z) + f(z-4h)}_{\approx 8f(z)} - \underbrace{(4f(z+2h) + 4f(z-2h))}_{\approx 8f(z)}],$$

which can only be if the expression contained in the square brackets is $O(10^{-12})$, which is very small. Thus, we subtract two numbers of about equal magnitude, losing all digits down to machine epsilon before multiplying this almost empty bit string by a large number (10^{12}). While using a larger h reduces the numerical extinction, it increases the approximation error of the finite difference – quadratically! This means that when choosing h there is a trade-off between the approximation error and the numerical rounding error. One can show that for first-order forward finite differences, the optimal h is:

$$h_{\text{opt}} \approx \sqrt{2 \cdot 10^{-n} \frac{|f(z)|}{|f''(z)|}} \tag{9.7}$$

for n-digit floating-point arithmetics (for MATLAB: $n = 16$). For this choice of h, the approximation error is about equal to the rounding error. Smaller h decrease the overall accuracy due to larger rounding errors, while larger h decrease the accuracy due to larger approximation errors.

9.2 Romberg Scheme

The trade-off between approximation error and rounding error can be relaxed using the principle of "extrapolation toward zero". Because the approximation error terms are known from the Taylor expansion, one can design an extrapolation scheme that successively reduces them. This reduces the absolute value of the approximation error, but not its asymptotic scaling with h, which allows us to choose larger h for the same error, thereby avoiding numerical extinction problems.

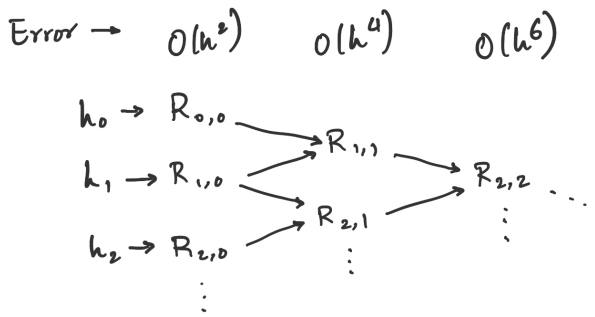


Figure 9.3: Illustration of the Romberg scheme

This is achieved by the Romberg scheme, due to Werner Romberg, as illustrated in Fig. 9.3. In this scheme, each column corresponds to a certain approximation order, whereas each row corresponds to a certain choice of h. The first column is given by the symmetric finite differences according to Eq. 9.6, thus:

$$R_{s,0} = \Delta_{h_s}^{(\nu)} \tag{9.8}$$

for increasingly finer resolutions

$$h_s = q^s h, \qquad 0 < q < 1, \quad s = 0, 1, 2 \ldots. \tag{9.9}$$

The subsequent columns are then computed by:

$$R_{s,t} = R_{s,t-1} + \frac{1}{(1/q^2)^t - 1}(R_{s,t-1} - R_{s-1,t-1}) \tag{9.10}$$

for

$$t = 1, 2, 3, \ldots, s, \quad s = 1, 2, 3 \ldots.$$

Interestingly, it turns out that the intuitive *refinement factor* $q = \frac{1}{2}$ leads to numerical extinction across the columns (i.e., when computing the $R_{s,t}$ for larger s and t) and is therefore a bad choice. A good choice turns out to be $q = \frac{1}{\sqrt{2}}$.

Still, the first column is eventually going to suffer from the extinction problems of Eq. 9.6, which is why h should not be too small in practice. Typically, the Romberg scheme is therefore limited to $s \lesssim 6$. It is also numerically better to start the computation of the scheme from the "bottom", i.e., from the highest-resolution level, and to successively increase h from there (remember: sum numbers from small to large, see Example 1.6).
A frequent choice in practice is $h_0 = 0.11$ and $q = \frac{1}{\sqrt{2}}$ or $q = \frac{2}{3}$, and to stop the scheme as soon as $|R_{s,s} - R_{s-1,s-1}| \leq \text{ATOL} + |R_{s,s}|\text{RTOL}$ or when $s = s_{\max}$. The tolerance has to be chosen larger for higher derivatives, as they cannot be computed as accurately as lower derivatives, because of noise amplification.

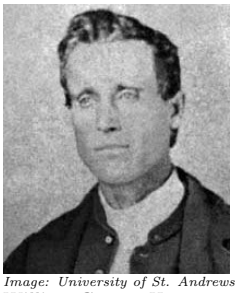


Image: University of St. Andrews
William George Horner
* 9 June 1786
Bristol, UK
† 22 September 1837
Bath, UK

Automatic Differentiation (AD)

A special case is when $f(x)$ is given by a computer program or subroutine that is available in source code. This program reads z as an input and returns $f(z)$. The idea of automatic differentiation (AD) is to automatically (i.e., by the compiler) change or rewrite the program such that is computes $f^{(\nu)}(z)$ for any given input z. AD is an active field of research and has been rather successful recently with the advent of machine learning and artificial intelligence. It is frequently used to compute gradients of deep neural nets during training.

Example 9.1. *As an example, consider the Horner Scheme (attributed to William George Horner, who published it in 1819, albeit the method was already used by Joseph-Louis Lagrange and can be traced back many hundreds of years to Chinese—Qin Jiushao 600 yeas earlier—and Persian—Sharaf al-Din al-Tusi 700 years earlier—mathematicians) for efficient evaluation of a polynomial. The input to the scheme is a vector of coefficients $\vec{a} = (a_0, \ldots, a_m)$ of a polynomial*

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m,$$

as well as the location z at which the polynomial is to be evaluated. The Horner scheme evaluates the polynomial as:

$$p(z) = (\ldots((a_m z + a_{m-1})z + a_{m-2})z + \cdots + a_1)z + a_0.$$

This can be implemented in MATLAB, for example, as the black code that follows:

```
function horner(a,z)
    f1 = 0;
    f = 0;
    m = length(a) - 1;
    for j = m:-1:0;
```

```
        f1 = f1*z + f;
        f = f*z + a(j+1);
    end
    return f1;
    return f;
```

If one now also wants to compute the first derivative $p'(z)$, the compiler can automatically insert the statements printed in red, which compute the first derivative alongside.

The general concept in AD is to write the formal derivative of each statement containing f *before* that statement.

Chapter 10

Initial Value Problems of Ordinary Differential Equations

Ordinary differential equations (ODEs), i.e., equations relating a function to its derivatives in one variable, are a common mathematical tool to model systems with continuous dynamics. They occur in a multitude of applications from many-body mechanics to chemistry to electronics engineering and population genetics. The simplest case are linear ODEs of first order, such as the one in the following example.

Example 10.1. *Radioactive decay is a physical process characterized by uniformly random spontaneous events (more precisely: by a Bernoulli process). Let $m(t)$ be the mass of radioactive material at time t. Further let λ be the decay rate, i.e., the fraction of mass that decays per unit time on average. Then, the time evolution of $m(t)$ is given by the following ODE:*

$$\frac{\mathrm{d}m}{\mathrm{d}t} = -\lambda m(t).$$

This linear ODE of first order can be solved analytically by

$$m(t) = C\mathrm{e}^{-\lambda t},$$

which is the general solution for any $C \in \mathbb{R}$. From this infinite number of general solutions, the well-defined particular solution is selected by the initial condition $m(t = t_0) = m_0$. Then, $m(t) = m_0\mathrm{e}^{-\lambda(t-t_0)}$.

Image: wikipedia
Jacob Bernoulli
∗ 6 January 1655
Basel, Switzerland
† 16 August 1705
Basel, Switzerland

In general, the solution of an ODE $\dot{x} := \frac{\mathrm{d}x}{\mathrm{d}t} = f(t, x)$ for some given *right-hand side* $f(t, x)$ is a function $x(t) : \mathbb{R} \to \mathbb{R}$, such that $\dot{x}(t) = f(t, x(t))$.
As we have also seen in the example above, ODEs have (infinitely) many solutions, typically parameterized by an integration constant. To determine one

particular solution, an initial condition for $x(t = t_0)$ is required. While the analytical solution of an ODE can be general, a numerical solution necessarily needs to be particular in order for the problem to be well-posed. Therefore, numerically solving an ODE is called an *initial value problem* (IVP).

Definition 10.1 (Initial value problem)**.** *Given t_0 and x_0, solving $\dot{x} = f(t, x)$ such that $\dot{x}(t) = f(t, x(t))$ and $x(t = t_0) = x_0$ is called an* initial value problem *(IVP).*

If $f(t, x)$ is Lipschitz-continuous (see Section 4.2.1), the initial value problem has a unique solution.

Example 10.2. *Consider the right-hand side $f(t, x) = \frac{x^2}{t}$ and $t_0 = 1$, $x_0 = 1$. The associated initial value problem is:*

$$\dot{x} = \frac{x^2}{t}, \qquad x(1) = 1,$$

which possesses the unique solution $x(t) = \frac{1}{1-\ln t}$. This is an example of a nonlinear IVP that can still be solved analytically.

10.1 Numerical Problem Formulation

In cases where the IVP cannot be solved analytically, for example for general nonlinear right-hand sides f, we aim to determine a numerical approximation to the solution. Given an IVP

$$\dot{x} = f(t, x(t)), \qquad x(t_0) = x_0, \tag{10.1}$$

and some final time t_F, the goal is to compute an approximation to $x(t_F)$, see illustration in Fig. 10.1. This is a well-posed numerical problem. Since t_F can be chosen arbitrarily, computing this problem allows approximating the solution of the IVP at any time point of interest.

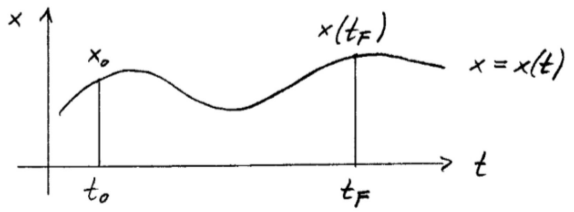


Figure 10.1: Illustration of the numerical initial value problem.

Since t (henceforth called "time" although it could represent any other quantity) is a continuous variable, we need to discretize it in order to enable a computational treatment. This is generally done by subdividing the interval $[t_0, t_F]$ by

a finite number of collocation points $t_0 < t_1 < \cdots < t_n = t_F$. Then, we can successively compute the approximations $x(t_1), x(t_2), \ldots, x(t_n)$. The original global problem is hence broken down into a series of local problems, which is why this procedure is also referred to as *time stepping*. Time stepping algorithms are classified into *one-step methods* (which are further sub-classified into *single-stage* and *multi-stage* methods), *multistep methods*, and *implicit methods*. These classes of methods differ with respect to their numerical properties, accuracy, and computational cost. The field of time stepping algorithms is still an active area of research with new ideas and approaches regularly invented.

10.2 Explicit One-Step Methods

Explicit one-step methods compute an approximation to the solution at the next time step from only the solution at the current time step. They require one step to compute the next, hence the name "one-step methods". Their derivation is generally based on the Taylor expansion

$$x(t+h) = x(t) + \frac{h}{1!}\frac{\mathrm{d}x}{\mathrm{d}t}(t) + \frac{h^2}{2!}\frac{\mathrm{d}^2x}{\mathrm{d}t^2}(t) + \cdots + \frac{h^p}{p!}\frac{\mathrm{d}^px}{\mathrm{d}t^p}(t) + \frac{h^{p+1}}{(p+1)!}\frac{\mathrm{d}^{p+1}x}{\mathrm{d}t^{p+1}}(\xi) \quad (10.2)$$

for some $\xi \in (t, t+h)$. For sufficiently small $|h|$, terms up to and including order p approximate $x(t+h)$ with error $O(h^{p+1})$.
One-step methods are *explicit* if the solution x at a later time $t+h$ is approximated using only the already known solution at time t and the right-hand side function f. The approximation is hence computable as an explicit formula, which is defined by a *propagator function* $F(t, x, h)$ such that $x(t+h) \approx F(t, x(t), h)$. This can then be applied over the time steps

$$t_0 < t_1 < \cdots < t_n = t_F$$

to compute a sequence of numbers

$$\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n$$

that are approximations to the true solutions

$$\tilde{x}_1 \approx x_1 = x(t_1),\ \tilde{x}_2 \approx x_2 = x(t_2), \ldots, \tilde{x}_n \approx x_n = x(t_n)$$

of the IVP in Eq. 10.1 at these time points. The solution $\tilde{x}_0 = x_0$ at time t_0 does not need to be computed, as it is given by the initial condition. The time steps taken are

$$h_0 = t_1 - t_0,\ h_1 = t_2 - t_1, \ldots, h_{n-1} = t_n - t_{n-1}.$$

The iteration proceeds by:

$$\tilde{x}_{j+1} = \tilde{x}(t_j + h_j) = F(t_j, \tilde{x}_j, h_j), \qquad j = 0, 1, 2, \ldots, n-1.$$

In practice, the time points are often chosen equidistant for simplicity, as:

$$h = \frac{t_F - t_0}{n}, \quad t_j = t_0 + jh,$$

but this is not necessary.
One-step explicit time stepping incurs two approximation errors:

$$x(t_j + h_j) \approx F(t_j, x(t_j), h_j) \approx F(t_j, \tilde{x}_j, h_j) = \tilde{x}_{j+1}.$$

The first error is due to approximating $x(t_j + h_j)$ by $F(t_j, x(t_j), h_j)$, i.e., by its truncated Taylor series. This error is called the *local discretization error* of the time stepping algorithm and, as seen above, it is of order $O(h^{p+1})$ independently for each step. The second error is due to approximating $F(t_j, x(t_j), h_j)$ by $F(t_j, \tilde{x}_j, h_j)$, i.e., due to using the previous numerical approximation $\tilde{x}_j$ in the propagator function and not the (unknown) true solution. This error is called the *global discretization error* of the time stepping algorithm, because it accumulates over time steps. Formally, we define:

Definition 10.2 (Local discretization error)**.** *The error of an explicit time-stepping method in a single step j is called the* local discretization error d_j.

Therefore, the local error measures the deviation between the exact solution $x(t_j + h_j)$ and the numerical solution starting from the exact solution at the previous time point, i.e., $d_j = x(t_j + h_j) - F(t_j, x(t_j), h_j)$. Equivalently, the local error is the difference between the numerical solution $\tilde{x}_{j+1}$ and the exact solution $z(t_j + h_j)$ starting from the previous numerical solution as its initial condition $z(t_j) = \tilde{x}_j$. Both constructions fulfill the above definition and can be used interchangeably. While the exact value of d_j may differ in the two constructions, the convergence behavior is the same since it is independent of x.

Definition 10.3 (Global discretization error)**.** *The difference $D_j := x(t_j) - \tilde{x}_j$ is called the* global discretization error *of an explicit time-stepping method at time t_j.*

The global error measures the absolute deviation between the numerical solution and the exact solution through (t_0, x_0) at any given time.
Since the total number of time steps required to reach the final time t_F is $O(1/h)$, the global error at time t_F is of order $O(h^{p+1}/h) = O(h^p)$. This proves the following theorem:

Theorem 10.1. *Any explicit one-step method to numerically solve an IVP with local discretization error $d_j \in O(h^{p+1})$ has global discretization error $D_j \in O(h^p)$, i.e., the global order of convergence is one less than the local order of convergence.*

Finally, we define:

Definition 10.4 (Consistency)**.** *A numerical method for IVPs is called* consistent *if and only if it has a positive non-zero global order of convergence.*

For consistent methods, the numerical solution $\tilde{x}_n$ converges to the exact solution $x(t_n)$ when $h_j \to 0 \, \forall j$ if the method is stable (cf. Section 10.9).

10.2.1 Explicit single-stage one-step methods: the explicit Euler method

A single-stage method only requires evaluating the right-hand side f at *one* location (t^*, x^*) in each time step. The simplest of such algorithms approximates $x(t+h)$ by $x(t) + h\dot{x}(t)$, i.e., by using the Taylor series up to and including terms of order $p = 1$. Because $\dot{x}(t) = f(t, x(t))$ from the problem definition in Eq. 10.1, this yields the propagator function $F(t, x, h) = x + hf(t, x)$. The value $F(t, x(t), h)$ approximates the true solution $x(t+h)$ with local error $O(h^2)$, and the value $F(t, \tilde{x}_j, h)$ does the same with global error $O(h)$. Therefore, the method is consistent. This is the famous *explicit Euler scheme* for numerically solving an IVP, which Leonhard Euler published in 1768 in his book *Institutionum calculi integralis*:

$$\tilde{x}_{j+1} = F(t_j, \tilde{x}_j, h_j) = \tilde{x}_j + h_j f(t_j, \tilde{x}_j), \qquad j = 0, 1, 2, \ldots, n-1. \quad (10.3)$$

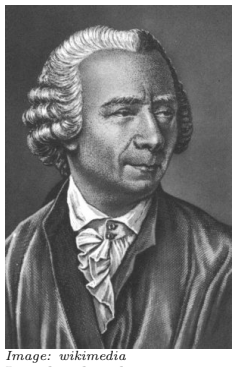


Image: wikimedia
Leonhard Euler
∗ 15 April 1707
Basel, Switzerland
† 18 September 1783
St.Petersburg, Russia

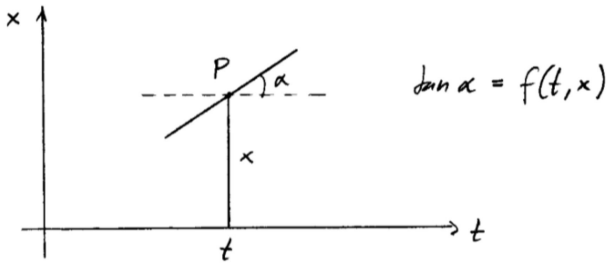


Figure 10.2: An ODE defines a slope at each point in the (t, x)-plane.

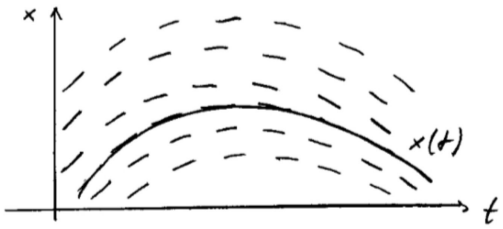


Figure 10.3: Any solution $x(t)$ is tangential to the director field $(t, x) \mapsto f(t, x) = \tan\alpha$ everywhere, i.e., for all (t, x).

The explicit Euler method can be interpreted geometrically, which leads to a better understanding of its functioning and suggests how to construct more accurate, higher-order schemes. The right-hand side f of Eq. 10.1 defines at each

point P in the (t, x)-plane a slope $\dot{x} = f(t, x) = \tan\alpha$, as illustrated in Fig. 10.2. Applying this over the entire plane defines a *director field* (an undirected vector field). The general solution of the ODE are all curves that are tangential to this director field at every point, as illustrated in Fig. 10.3. There are infinitely many curves $x(t)$ that fulfill this. The initial condition in Eq. 10.1 serves to select *one* of them as the *particular solution* of the IVP, namely the director field line that starts at point (t_0, x_0). This also illustrates why the initial condition is required to render the problem well posed, and why the solution is unique for Lipschitz-continuous f, where director field lines cannot cross nor jump.

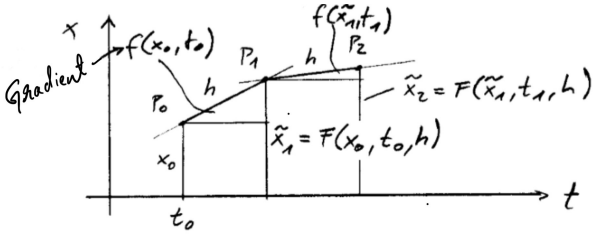


Figure 10.4: Illustration of two explicit Euler steps, $P_0 \to P_1$ and $P_1 \to P_2$.

The explicit Euler method can now be understood as making steps along straight line segments in the direction given by the director field. Starting from the initial point $P_0 = (t_0, x_0)$, a step of length h and slope $f(t_0, x_0)$ is taken in order to reach point $P_1 = (t_1, \tilde{x}_1)$. From there, another step of length h and slope $f(t_1, \tilde{x}_1)$ is taken to reach point $P_2 = (t_2, \tilde{x}_2)$, and so on. This is illustrated in Fig. 10.4 and defines a piecewise linear polynomial.

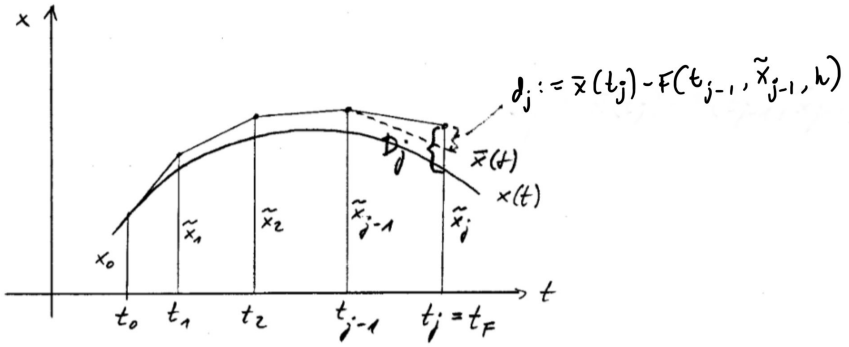


Figure 10.5: Local and global errors in the explicit Euler method. The dashed line is the exact solution $\bar{x}(t)$ starting from the previous numerical solutions as initial condition.

As illustrated in Fig. 10.5, this polynomial may increasingly deviate from the exact solution $x(t)$ as more steps are taken. In each step, a local error of size $d_j = x(t_{j+1}) - F(t_j, x(t_j), h_j) \in O(h^2)$ is made, which is the same order as the

deviation between the numerical solution and the analytical solution through the previous point (dashed line in the figure). Overall, until final time t_F, the accumulated global error is $D_j = \tilde{x}(t_F) - x(t_F) \in O(h)$.

10.2.2 Higher-order single-stage one-step methods

Clearly, the construction of the explicit Euler scheme can be extended to higher orders of convergence by truncating the Taylor series from Eq. 10.2 at $p > 1$. For example, truncating after $p = 2$, we would approximate $x(t + h)$ by $x(t) + h\dot{x}(t) + \frac{h^2}{2}\ddot{x}(t)$. We know from the problem formulation in Eq. 10.1 that

$$\dot{x}(t) = f(t, x(t)),$$

but what is $\ddot{x}(t)$? Applying the chain rule of differentiation to the above expression, we find:

$$\ddot{x}(t) = \frac{\mathrm{d}f}{\mathrm{d}t}(t, x(t)) + \frac{\mathrm{d}f}{\mathrm{d}x}(t, x(t))\dot{x}(t)$$

and thus:

$$\ddot{x}(t) = \frac{\mathrm{d}f}{\mathrm{d}t}(t, x(t)) + \frac{\mathrm{d}f}{\mathrm{d}x}(t, x(t))f(t, x(t)).$$

Defining the propagator function

$$F(t, x, h) = x + hf(t, x) + \frac{h^2}{2}\left[\frac{\mathrm{d}f}{\mathrm{d}t}(t, x(t)) + \frac{\mathrm{d}f}{\mathrm{d}x}(t, x(t))f(t, x(t))\right],$$

we see that $F(t, x, h)$ approximates $x(t + h)$ with a local discretization error of order $O(h^3)$. The usual iteration $\tilde{x}_{j+1} = F(t_j, \tilde{x}_j, h_j)$ for $j = 0, 1, 2, \ldots$ then has a global discretization error of order $O(h^2)$, and the method is consistent.
A clear disadvantage of this method is that the partial derivatives of f with respect to both t and x must be known analytically. While this may be the case in some applications where the right-hand side f is given analytically, it may be limiting. Pursuing this route further, schemes of even higher order can be derived. For example, a scheme of global order p would be obtained by approximating $x(t+h)$ by $x(t)+h\dot{x}(t)+\cdots+\frac{h^p}{p!}x^{(p)}(t)$. For $p > 2$, however, an increasingly large number of partial derivatives and mixed partial derivatives of f must be analytically know. This route is therefore not practical in general, while it may have valuable applications in specific cases.

10.2.3 Heun's multi-stage one-step method

Analytical partial derivatives of the right-hand side can be avoided by numerically approximating them as nested finite differences of sufficiently high order of convergence. This leads to *multi-stage* methods where the right-hand side f is evaluated at multiple points in each time step. Each such point is called a *stage* of the method. This classic idea by Karl Heun (1886) leads to higher-order one-step schemes that do not require analytical derivatives to be available.
The classic Heun method uses the propagator function:

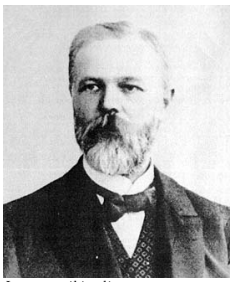


Image: wikipedia
Karl Heun
∗ 3 April 1859
Wiesbaden, Duchy of Nassau
† 10 January 1929
Karlsruhe, Weimar Republic

$$F(t,x,h) = x + \frac{h}{2}[f(t,x) + f(t+h,\, x+hf(t,x))] \tag{10.4}$$

and the usual one-step iteration

$$\tilde{x}_{j+1} = F(t_j, \tilde{x}_j, h_j), \quad j = 0, 1, \ldots, n-1.$$

This propagator is computable if and only if the right-hand side f of Eq. 10.1 can be evaluated at arbitrary points (t^*, x^*). We do not require analytical derivatives, but a computable function, which is a weaker requirement. In each step of the Heun method, two evaluations of the right-hand side are required: $f(t,x)$ and $f(t+h, x+hf(t,x))$. It is therefore a two-stage method.
We claim that the Heun method has a global order of convergence of 2, i.e., that the above propagator function exactly reproduces terms up to and including order $p = 2$ in the Taylor series of Eq. 10.2. To prove this, we expand

$$f(t+\delta, x+\Delta) = f(t,x) + f_t(t,x)\delta + f_x(t,x)\Delta + O(\delta^2) + O(\delta\Delta) + O(\Delta^2),$$

where subscripts mean partial derivatives with respect to the subscripted variable. Therefore:

$$f(t+h, x+hf(t,x)) = f(t,x) + f_t(t,x)h + f_x(t,x)hf(t,x) + O(h^2),$$

and

$$\frac{h}{2}\left[f(t+h, x+hf(t,x))\right] = \frac{h}{2}f(t,x) + \frac{h^2}{2}[f_t(t,x) + f_x(t,x)f(t,x)] + O(h^3).$$

Substituting into the expression for the Heun propagator function from Eq. 10.4, we find:

$$F(t,x,h) = x(t) + h\underbrace{f(t,x)}_{\dot{x}(t)} + \underbrace{[f_t(t,x) + f_x(t,x)f(t,x)]}_{\ddot{x}(t)}\frac{h^2}{2} + O(h^3).$$

Therefore, indeed, the Heun method has local error order 3 and global error order 2, rendering it more accurate than the Euler method at the price of requiring twice the number of function evaluations. This is beneficial in many applications because we pay an additive price (one additional function evaluation in each time step) for a multiplicative gain (error order goes from h to h^2).
From Eq. 10.4, we see that Heun's method also has a nice geometric interpretation: Instead of simply following the tangent of $x(t)$ at t_j, it uses two slopes, $f(t,x)$ and $f(t+h, x+hf(t,x))$, and averages them. The first is the tangent at the current time point, as in the explicit Euler method. The second is the tangent at the point one would land at when doing an Euler step (notice that the shift in the function arguments of f exactly corresponds to one Euler step into the future). These two slopes are then averaged in order to form a more accurate prediction of the new time point.

10.2.4 Higher-order multi-stage one-step methods: Runge-Kutta methods

Image: wikidata
Carl David Tolmé Runge
∗ 30 August 1856
Bremen, State of Bremen
(now: Germany)
† 3 January 1927
Göttingen, Weimar Republic
(now: Germany)

The idea behind the Heun method, namely to average the slopes of $x(t)$ at suitably shifted locations in order to form a more accurate prediction, can be generalized from averaging two slopes to averaging more than two slopes. This gives rise to the large family of Runge-Kutta methods, due to Carl Runge and Wilhelm Kutta who proposed these methods in the year 1900. All Runge-Kutta methods substitute partial derivatives by nested function evaluations (amounting to embedded finite differences) at suitably shifted locations in order to reproduce the Taylor expansion from Eq. 10.2 to some order p.
In general, the s stages of a Runge-Kutta method are given by s shifted evaluations of the right-hand side, i.e., the slopes of $x(t)$ at s different locations in space-time. These are then averaged with suitable weights. The propagator function therefore is a linear combination of shifted function evaluations, hence in the most general explicit case:

$$k_i = f\left(t + c_i h,\, x + h\sum_{j=1}^{i-1} a_{ij} k_j\right), \qquad i = 1, \ldots, s \tag{10.5}$$

$$F(t, x, h) = x + h\sum_{i=1}^{s} b_i k_i. \tag{10.6}$$

Image: wikipedia
Wilhelm Martin Kutta
∗ 3 November 1867
Pitschen, Prussia
(now: Byczyna, Poland)
† 25 December 1944
Fürstenfeldbruck, Germany

In order to define such a scheme, one must specify the recursive[1] spatial shift matrix $\mathbf{A} = (a_{ij})$, the temporal shift vector $\vec{c} = (c_i)$, and the averaging weights vector $\vec{b} = (b_i)$.

Example 10.3. *Defining the $s = 4$ recursive stages*

$$\begin{aligned} k_1 &= f(t, x), \\ k_2 &= f(t + \frac{h}{2}, x + \frac{h}{2}k_1), \\ k_3 &= f(t + \frac{h}{2}, x + \frac{h}{2}k_2), \\ k_4 &= f(t + h, x + hk_3), \end{aligned}$$

and the propagator function

$$F(t, x, h) = x + \frac{h}{6}[k_1 + 2k_2 + 2k_3 + k_4]$$

yields a Runge-Kutta scheme with a global order of convergence of 4. It has the following shifts and weights:

Image: wikipedia
John C. Butcher
∗ 31 March 1933
Auckland, New Zealand

[1] The spatial shifts are recursive because the shift for stage i depends on the values of the previous stages $j = 1, \ldots, i-1$.

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad \vec{c}^{\mathsf{T}} = \begin{bmatrix} 0, \frac{1}{2}, \frac{1}{2}, 1 \end{bmatrix}, \quad \vec{b}^{\mathsf{T}} = \begin{bmatrix} \frac{1}{6}, \frac{1}{3}, \frac{1}{3}, \frac{1}{6} \end{bmatrix}.$$

This scheme is frequently used in computational mechanics and is usually referred to by the abbreviation "RK4".

When defining a Runge-Kutta method, it is custom to arrange the matrix $\mathbf{A}$ and the two vectors $\vec{b}$ and $\vec{c}$ in a *Butcher tableau* (John C. Butcher, 1975), as shown in Fig. 10.6 for explicit Runge-Kutta methods. In an explicit Runge-Kutta method, the matrix $\mathbf{A}$ is lower triangular, so each stage can be evaluated using only the previous stages. Also, it has been shown that explicit Runge-Kutta methods are only consistent (i.e., have global order of convergence > 0) if $\sum_{i=1}^{s} b_i = 1$ and $c_i = \sum_{j=1}^{i-1} a_{ij}$ (verify that these hold for Example 10.3). These are necessary conditions, but they are not sufficient. Another interesting fact is that for global convergence orders $p > 4$ there exists no Runge-Kutta method with $s \leq p$ stages. Runge-Kutta methods with $s = p$ stages are called *optimal*. Therefore, RK4 is the optimal Runge-Kutta method of highest possible convergence order, justifying its general importance. Explicit Runge-Kutta methods with $p > s$ do not exist.

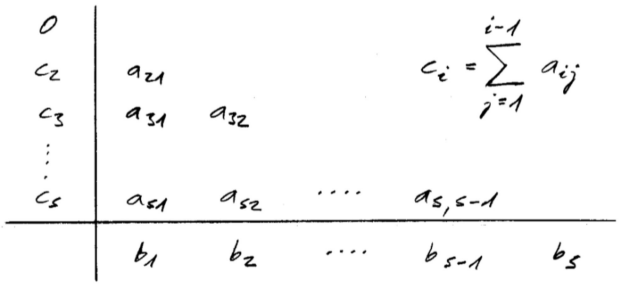


Figure 10.6: Butcher tableau for an explicit Runge-Kutta method.

Example 10.4. *The explicit Euler method is a Runge-Kutta method of global order $p = 1$ with $s = 1$ stage, thus Euler=RK1. The Heun method is a Runge-Kutta method with $p = s = 2$, thus Heun=RK2. The RK4 method from example 10.3 has $p = s = 4$. All are optimal, and their Butcher tableaus are given in Fig. 10.7.*

10.3 Dynamic Step-Size Adaptation in One-Step Methods

The time-step size in explicit methods needs to be chosen small enough to resolve variations in the solution $x(t)$. This is intuitive from Fig. 10.5 where

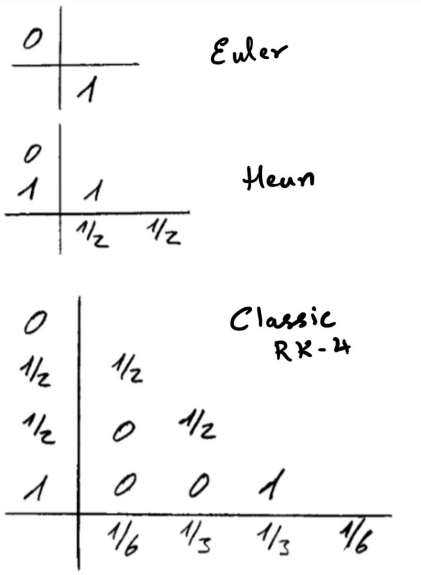


Figure 10.7: Butcher tableaus for the explicit Euler, Heun, and RK4 methods.

taking one large step instead of j small ones would miss the inflection point and lead to a completely wrong prediction. This requirement of well-sampling the solution also follows from the Nyquist-Shannon sampling theorem. In practice, this means that the time-step size must be limited by the fastest dynamics to be resolved. When using the same time-step size throughout a simulation, this may be wasteful, as the fastest transients anywhere in the simulation govern the time-step size everywhere. In regions where the solution varies more slowly, which by definition is almost everywhere else, larger steps could in principle be taken, requiring less computational time.

The goal of dynamic step-size adaptation is to automatically adjust the time-step size in an explicit one-step IVP method, such that at every step of the algorithm the largest possible step size is used for maximum computational efficiency. This means that the time-step size is no longer a constant h, but depends on the step $j = 0, 1, 2, \ldots, n-1$. Ideally, one chooses at each step j a step size h_j that is just small enough for the local error of the IVP solver to remain below a certain, given tolerance TOL, hence:

$$|d_j(t_j, \tilde{x}_j, h_j)| = |x(t_j + h_j) - F(t_j, x(t_j), h_j)| \leq \text{TOL}. \tag{10.7}$$

The problem is that the function $d_j(t_j, \tilde{x}_j, h_j)$ is not known, but only its asymptotic scaling for $h \to 0$, i.e., the order of convergence of the local error is known. Therefore, dynamic step-size adaptation always hinges on finding an estimator $\hat{\ell}(t_j, \tilde{x}_j, h_j)$ for the magnitude of the local discretization error. A popular idea for constructing such an estimator is to perform each time step twice, with two different methods, as:

1. calculate $\tilde{x}_{j+1} = F(t_j, \tilde{x}_j, h_j)$ with a first method F,
2. calculate $\hat{\tilde{x}}_{j+1} = \hat{F}(t_j, \tilde{x}_j, h_j)$ with a reference method $\hat{F}$, and
3. set $\hat{\ell}(t_j, \tilde{x}_j, h_j) \coloneqq \hat{F}(t_j, \tilde{x}_j, h_j) - F(t_j, \tilde{x}_j, h_j)$.

Theorem 10.2 (Convergence of the error estimator)**.** *When using an explicit one-step method F of global convergence order p, and an explicit one-step reference method $\hat{F}$ of global convergence order $\hat{p} \geq p$, then the error estimator $\hat{\ell}(t_j, \tilde{x}_j, h_j) \coloneqq \hat{F}(t_j, \tilde{x}_j, h_j) - F(t_j, \tilde{x}_j, h_j)$ has an error of at most $O(h_j^{p+1})$.*

Proof. We start by adding and subtracting the locally exact solution z_{j+1} to the error estimator:

$$\hat{\ell}(t_j, \tilde{x}_j, h_j) = \hat{F}(t_j, \tilde{x}_j, h_j) - F(t_j, \tilde{x}_j, h_j) - z_{j+1} + z_{j+1},$$

where $z_{j+1} = z(t_j + h_j)$ is the exact solution with initial condition $z(t_j) = \tilde{x}_j$, i.e., starting from the current numerical solution. Then,

$$z_{j+1} - F(t_j, \tilde{x}_j, h_j) - (z_{j+1} - \hat{F}(t_j, \tilde{x}_j, h_j)) = d_j(t_j, \tilde{x}_j, h_j) - \hat{d}_j(t_j, \tilde{x}_j, h_j)$$

is the difference between the unknown local errors of the two schemes. By the assumption on the global convergence orders of the two schemes, this difference is asymptotically:

$$O(h_j^{p+1}) - O(h_j^{\hat{p}+1}) \leq O(h_j^{p+1})$$

because $\hat{p} \geq p$ by assumption. □

The question that remains to be answered is how to choose the reference method $\hat{F}$. There are two classic ways: Richardson extrapolation and embedded Runge-Kutta methods.

10.3.1 Richardson extrapolation

Richardson extrapolation is a classical principle in numerical computing, and we have already used it in Section 8.4 to derive higher-order quadrature schemes. For the sake of example, and without loss of generality, we consider the case $p = 1$ with F the propagator of the explicit Euler method. We choose the reference scheme $\hat{F}$ that consists of taking two half-steps with the original scheme F. In the example of the Euler method, we therefore simply perform two explicit Euler steps of length $\hat{h}_j = h_j/2$ each, thus $\hat{p} = p$. Then, we have for the reference method:

$$\begin{aligned}\hat{\tilde{x}}_{j+1} &= \tilde{x}_j + \frac{h_j}{2} f(t_j, \tilde{x}_j) + \frac{h_j}{2} f\left(t_j + \frac{h_j}{2}, \tilde{x}_j + \frac{h_j}{2} f(t_j, \tilde{x}_j)\right) \\ &= \tilde{x}_j + h_j f(t_j, \tilde{x}_j) + \frac{h_j^2}{4}[f_t(\dots) + f_x(\dots) f(\dots)] + O(h_j^3), \qquad (10.8)\end{aligned}$$

because the second step can be Taylor-expanded as:

$$f\left(t_j + \frac{h_j}{2}, \tilde{x}_j + \frac{h_j}{2} f(t_j, \tilde{x}_j)\right) =$$
$$f(t_j, \tilde{x}_j) + \frac{h_j}{2} f_t(t_j, \tilde{x}_j) + \frac{h_j}{2} f_x(t_j, \tilde{x}_j) f(t_j, \tilde{x}_j) + O(h_j^2),$$

just as we did in Section 10.2.3 when deriving the Heun method. Subscripts to functions again indicate partial derivatives of the function with respect to the subscripted variable.
Let again $z(t)$ for $t \geq t_j$ be the locally exact solution to the IVP with initial condition $z(t_j) = \tilde{x}_j$. Its Taylor expansion for one time step is:

$$z(t_j + h_j) = \tilde{x}_j + h_j f(t_j, \tilde{x}_j) + \frac{h_j^2}{2} z_{tt}(t_j) + O(h_j^3). \tag{10.9}$$

Therefore, we find for the local error of the scheme F:

$$z(t_j + h_j) - \tilde{x}_{j+1} = \frac{h_j^2}{2} z_{tt}(t_j) + O(h_j^3). \tag{10.10}$$

In order to derive the local error of the scheme $\hat{F}$, we first notice that the term within the angle brackets in Eq. 10.8 is equal to $z_{tt}(t_j)$. This is because from the definition $z_t(t) = f(t, z)$, the chain rule of differentiation gives: $z_{tt}(t) = f_t(t, z) + f_z(t, z) f(t, z)$. Subtracting then Eq. 10.8 from 10.9, we find the local error of the scheme $\hat{F}$:

$$z(t_j + h_j) - \hat{\tilde{x}}_{j+1} = \frac{h_j^2}{4} z_{tt}(t_j) + O(h_j^3). \tag{10.11}$$

Now comes the Richardson extrapolation step: adding or subtracting multiples of the two Taylor expansions in Eqs. 10.10 and 10.11 from one another in order to cancel the leading-order error term. In this case, we compute 2·Eq. 10.11 − Eq. 10.10, and find:

$$z(t_j + h_j) - 2\hat{\tilde{x}}_{j+1} + \tilde{x}_{j+1} = O(h_j^3)$$
$$z(t_j + h_j) - \hat{\tilde{x}}_{j+1} = \hat{\tilde{x}}_{j+1} - \tilde{x}_{j+1} + O(h_j^3)$$
$$\hat{d}_j(t_j, \hat{\tilde{x}}_j, h_j) = \hat{\ell}(t_j, \tilde{x}_j, h_j) + O(h_j^3).$$

Therefore, $\hat{\ell} = \hat{F} - F$ provides an estimator for the local error $\hat{d}_j$ at $\hat{\tilde{x}}_{j+1}$ with accuracy of order $O(h_j^3)$. Also, $2\hat{\ell}(t_j, \tilde{x}_j, h_j)$ provides an estimator for the local error d_j at $\tilde{x}_{j+1}$ with accuracy of order $O(h_j^3)$, because $d_j(\tilde{x}_j) = 2\hat{d}_j(\hat{\tilde{x}}_j)$ since $h_j = 2\hat{h}_j$.
Although we have used the Euler method as an example here, Richardson extrapolation is generally applicable to explicit one-step methods. In general, one finds for an explicit one-step method of global convergence order p:

$$z(t_j + h_j) - \hat{\tilde{x}}_{j+1} = \frac{\hat{\tilde{x}}_{j+1} - \tilde{x}_{j+1}}{2^p - 1} + O(h_j^{p+2}), \tag{10.12}$$

when using the same scheme as a reference scheme with $\hat{h}_j = h_j/2$ for all j. Comparing this with Theorem 10.2, we see that the accuracy of a Richardson error estimator with $\hat{h}_j = h_j/2$ for all j is one order better than what is guaranteed in the general case.

10.3.2 Embedded Runge-Kutta methods

Another way of constructing a local error estimator is to consider two solution approximations $(\tilde{x}_{j+1}, \hat{\tilde{x}}_{j+1})$ computed using two members of the Runge-Kutta family of methods with identical step sizes h_j and identical function evaluations, but with different global convergence orders p and $\hat{p} > p$. Typically, one chooses $\hat{p} = p + 1$.
Since the function evaluations in both Runge-Kutta methods are the same (for computational efficiency), the coefficients a_{ij} and c_i in the Butcher tableaus of the two schemes are identical. The only difference is in the weights b_i, with the b_i of the original scheme and the $\hat{b}_i$ of the reference scheme being different. The resulting joint Butcher tableau of such an *embedded explicit Runge-Kutta method* is illustrated in Fig. 10.8.

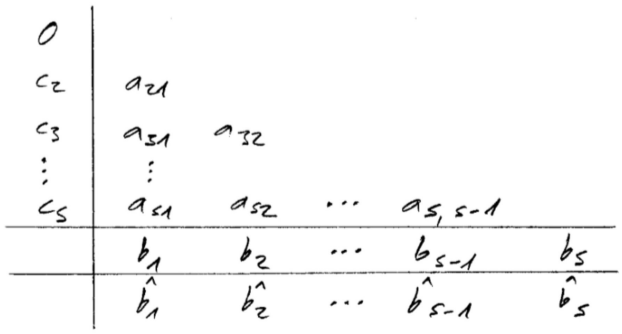


Figure 10.8: Joint Butcher tableau of an embedded explicit Runge-Kutta method.

We then have the two Runge-Kutta methods

$$\tilde{x}_{j+1} = \tilde{x}_j + h \sum_{i=1}^{s} b_i k_i,$$
$$\hat{\tilde{x}}_{j+1} = \tilde{x}_j + h \sum_{i=1}^{s} \hat{b}_i k_i,$$

with identical stages k_i, but different global convergence orders p and $\hat{p}$, respectively. The number of stages needs to be at least $s \geq \max(p, \hat{p})$ so different linear combinations can yield approximations of different orders. Optimal embedded Runge-Kutta method with respect to p are therefore not possible. In embedded

Runge-Kutta methods, $\hat{\tilde{x}}_{j+1} - \tilde{x}_{j+1} \in O(h^{p+1})$, which is the lower bound from Theorem 10.2. Therefore, in this case, the bound is tight (not proven here), and the accuracy of the error estimator in an embedded Runge-Kutta method is therefore worse than when using Richardson extrapolation. However, Richardson extrapolation requires additional function evaluations and therefore doubles the computational cost of the simulation, whereas embedded Runge-Kutta has almost no added cost (only a second weighted average). Richardson extrapolation is therefore only worth it if the time required for the additional function evaluations is amortized by the runtime gain due to better adaptive time stepping. Otherwise, one typically chooses an embedded Runge-Kutta method.
There are infinitely many possibilities for embedded Runge-Kutta methods. One of the most frequently used flavors in practice is the method *DOPRI5*, due to J. R. Dormand and P. J. Prince (1980), which achieves $p = 4$ and $\hat{p} = 5$ using $s = 7$ Runge-Kutta stages. Other famous examples of embedded Runge-Kutta methods include the original Fehlberg method (Erwin Fehlberg, 1969), which first introduced the idea of embedded error estimators, and the Cash-Karp method (Jeff Cash and Alan Karp, 1990).

10.3.3 Practical implementation

Using any of the above methods, we can determine the optimal size of the next time step from the user-provided tolerance TOL. Given a time-stepping method F of global convergence order p, and a reference method $\hat{F}$ of global order $\hat{p} \geq p$, we have the local error estimator

$$\hat{\ell}(t_j, \tilde{x}_j, h_j) = \hat{F}(t_j, \tilde{x}_j, h_j) - F(t_j, \tilde{x}_j, h_j) \in O(h_j^{p+1}),$$

as guaranteed by Theorem 10.2. This implies that

$$|\hat{\ell}| \cong C h_j^{p+1} \quad \Longrightarrow \quad C \cong \frac{|\hat{\ell}|}{h_j^{p+1}},$$

where the symbol $\cong$ means "asymptotically equal to". The result of step-size adaptation should be that

$$C h_{\text{opt}}^{p+1} \cong \text{TOL}.$$

Solving this for the optimal time-step size h_{opt} using the above estimate of the pre-factor C yields:

$$h_{\text{opt}} \cong \left(\frac{\text{TOL}}{C}\right)^{\frac{1}{p+1}} = h_j \left(\frac{\text{TOL}}{|\hat{\ell}|}\right)^{\frac{1}{p+1}}. \tag{10.13}$$

When using Richardson extrapolation, the exponent $p + 1$ can alternatively be replaced by $p + 2$ to get sharper adaptation, as the accuracy of the error estimation in this case exceeds the lower bound guaranteed by Theorem 10.2.

This formula is used in Algorithm 13, along with the two approximations of the result after the current time step ($\hat{\tilde{x}}_{j+1}$, $\tilde{x}_{j+1}$), in order to decide whether the step is accepted or has to be repeated, and what the new step size should be.

Algorithm 13 Dynamic step-size adaptation for one-step methods

1: **procedure** AdaptStep($\hat{\tilde{x}}_{j+1}$, $\tilde{x}_{j+1}$, TOL, F_s, p) ▷ Convergence order p
2: $\hat{\ell} = \hat{\tilde{x}}_{j+1} - \tilde{x}_{j+1}$
3: **if** $|\hat{\ell}| >$ TOL **then**
4: reject $\tilde{x}_{j+1}$
5: reduce step size $h_j \leftarrow h_j(\text{TOL}/|\hat{\ell}|)^{1/(p+1)} \cdot F_s$ ▷ safety factor F_s
6: **return** (j, h_j) ▷ repeat step j with h_j
7: **else if** $|\hat{\ell}| \leq$ TOL **then**
8: accept $\tilde{x}_{j+1}$
9: propose size for next step $h_{j+1} \leftarrow h_j(\text{TOL}/|\hat{\ell}|)^{1/(p+1)} \cdot F_s$
10: **return** $(j+1, h_{j+1})$ ▷ proceed to next time step
11: **end if**
12: **end procedure**

This algorithm is called at the end of each simulation time step, i.e., after the step has been computed with both the method F and the reference method $\hat{F}$ to compute $\tilde{x}_{j+1}$ and $\hat{\tilde{x}}_{j+1}$, respectively. If returns the index of the next time step to be computed, along with the corresponding step size. If the current step meets the tolerance TOL, then the time step index is advanced by 1 and the size of the next step is tentatively increased. Otherwise, the step index is not advanced and the calling procedure repeats step j with a reduced step size. The safety/fudge factor F_s is usually chosen around 0.8 and prevents over-adaptation, which otherwise could lead to unwanted oscillations in h_j. With $F_s < 1$, Algorithm 13 is guaranteed to converge, but may require arbitrarily many re-evaluations of any given time step if the solution suddenly becomes infinitely steep. Most software implementations therefore also include an upper limit for the number of retries, or equivalently a lower limit for h_j, $h_{\min}$, beyond which an error is raised and the adaptation terminated.

10.4 Implicit Methods

Implicit methods compute an approximation to the solution at the next time step using knowledge of the solution at both the current and the next step. Because they require information about the future in order to approximate the future, they are not explicitly computable. Instead, implicit methods lead to implicit equations that need to be numerically solved for the value of the solution at the next time step. While explicit methods are derived from a numerical approximation of the time derivative, implicit methods are derived from numerical quadrature schemes. Because of this, implicit methods are also

sometimes referred to as *time-integration* methods, or *time integrators*, instead of as *time-stepping* methods.

10.4.1 Implicit single-stage methods: Trapezoidal method

As a first example of an implicit time-integration method, we consider the trapezoidal method, which is derived from the quadrature scheme of the same name, as introduced in Section 8.2. We again consider the IVP

$$\dot{x}(t) = f(t, x(t)), \quad x(t_0) = x_0$$

at discrete time points $t_0 < t_1 < t_2 < \cdots < t_n = t_F$. Instead of numerically approximating the derivative $\dot{x}$, however, we integrate both sides of the equation from time t_0 until time t_1, yielding:

$$x(t_1) - x(t_0) = \int_{t_0}^{t_1} f(t, x(t)) \, \mathrm{d}t. \tag{10.14}$$

This integral equation is equivalent to the original IVP over the interval $[t_0, t_1]$. Using quadrature, we can approximate the integral on the right-hand side. Here, we use the trapezoidal method from Section 8.2 and get:

$$\int_{t_0}^{t_1} f(t, x(t)) \, \mathrm{d}t \approx \frac{t_1 - t_0}{2} [f(t_0, x(t_0)) + f(t_1, x(t_1))].$$

Clearly, we can not only do this for the first time step, but for every time step j of size $h_j = t_{j+1} - t_j$. Using the above trapezoidal quadrature approximation in Eq. 10.14 and solving for the value at the new time point, $\tilde{x}_{j+1} \approx x(t_{j+1})$, results in the *trapezoidal method*:

$$\tilde{x}_{j+1} = \tilde{x}_j + \frac{h_j}{2} [f(t_j, \tilde{x}_j) + f(t_{j+1}, \tilde{x}_{j+1})], \quad j = 0, 1, \ldots, n-1. \tag{10.15}$$

This is an implicit equation for $\tilde{x}_{j+1}$, since both the left and the right-hand side depend on the unknown $\tilde{x}_{j+1}$. The propagator function hence depends on the new time point:

$$\tilde{x}_{j+1} = F(t_j, \tilde{x}_j, t_{j+1}, \tilde{x}_{j+1}, h_j)$$

and cannot be explicitly evaluated, which is the hallmark of implicit methods. This implicit equation needs to be solved in each time step. If $f(t, x)$ is analytically known, it might be possible to solve the implicit equation analytically and thus obtain a closed-form propagator. For general $f(t, x)$, however, the equation usually needs to be solved numerically in each time step, e.g., using Newton methods or fixed-point iteration (provided that $|F'| < 1$) as introduced in Chapter 4. The starting value for the nonlinear solver is usually generated by performing one step of an explicit one-step method, e.g., explicit Euler:

$$\tilde{x}_{j+1}^{(0)} = \tilde{x}_j + h_j f(t_j, \tilde{x}_j).$$

Using this starting value for the example of the Trapezoidal method from Eq. 10.15, the nonlinear system solver then iterates

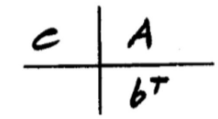
$$\tilde{x}_{j+1}^{(k+1)} = \tilde{x}_j + \frac{h_j}{2}[f(t_j, \tilde{x}_j) + f(t_{j+1}, \tilde{x}_{j+1}^{(k)})], \quad k = 0, 1, 2, \ldots$$

until convergence, yielding $\tilde{x}_{j+1}$ for the next time step. Convergence in guaranteed if h_j is small enough and f is Lipschitz-continuous (cf. Section 4.2.1). In the important special case of a linear function $f(t, x)$, the implicit equation is solved in each time step using a linear system solver, as introduced in Chapter 2.

The trapezoidal method of time integration is consistent with global order of convergence 2. While implicit methods incur a higher implementation complexity than their explicit counterparts, they are numerically superior, e.g., in terms of stability, as we will see in Section 10.7 below. Due to their favorable numerical properties, they may be computationally more efficient than explicit methods, as the overhead of solving the implicit equation may be amortized by, e.g., the ability to take larger time steps.

10.4.2 Implicit multi-stage methods: implicit Runge-Kutta

An important class of implicit time-integration methods in practical applications are implicit Runge-Kutta methods. Just like their explicit counterparts, implicit Runge-Kutta methods can be described by a Butcher tableau, as shown in Fig. 10.9, with two coefficient vectors $\vec{b}, \vec{c} \in \mathbb{R}^s$ and a matrix $\mathbf{A} \in \mathbb{R}^{s \times s}$ for a scheme with s stages.



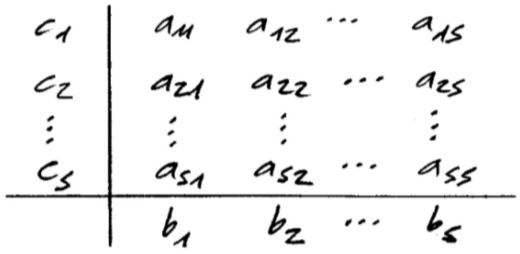


Figure 10.9: Butcher tableau for a general Runge-Kutta method with $\vec{b}, \vec{c} \in \mathbb{R}^s$ and $\mathbf{A} \in \mathbb{R}^{s \times s}$.

In an explicit Runge-Kutta method, the matrix $\mathbf{A}$ is lower-triangular, i.e. $a_{ij} = 0 \, \forall i \leq j$ (see Section 10.2.4), since entries on and above the diagonal correspond to function evaluations shifted into the future. In an implicit Runge-Kutta method, the matrix $\mathbf{A}$ may therefore be fully populated and the scheme reads:

$$k_i = f\left(t + c_i h,\, x + h \sum_{j=1}^{s} a_{ij} k_j\right), \quad i = 1, \ldots, s \tag{10.16}$$

$$F(t, x, h) = \bar{x} = x + h \sum_{i=1}^{s} b_i k_i \,. \tag{10.17}$$

Example 10.5. *The simplest example of an implicit Runge-Kutta method is the scheme with $s = p = 1$, whose Butcher tableau is shown in Fig. 10.10. Just like in the explicit case, this first member of the Runge-Kutta family is the Euler scheme, now implicit Euler, defined by:*

$$\bar{x} = x + h \underbrace{f(t + h, \bar{x})}_{k_1} .$$

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$$

Figure 10.10: Butcher tableau of the implicit Euler method.

Example 10.6. *Also the Trapezoidal method, as derived from quadrature in Section 10.4.1, is a member of the Runge-Kutta family with the Butcher tableau shown in Fig. 10.11. This scheme reads*

$$\bar{x} = x + \frac{h}{2}[\underbrace{f(t, x)}_{k_1} + \underbrace{f(t + h, \bar{x})}_{k_2}]$$

and has $s = p = 2$.

Examples 10.5 and 10.6 show the first two members of this family of methods, which are the *implicit Euler* and the Trapezoidal method. Both are optimal Runge-Kutta methods, as they have $p = s$, i.e., their orders of convergence are equal to their numbers of stages. However, implicit Euler is not the only implicit Runge-Kutta scheme with $s = 1$. Single-stage Runge-Kutta methods can be written as a parametric family with meta-parameter $\vartheta \in [0, 1]$ and a Butcher tableau as shown in Fig. 10.12.

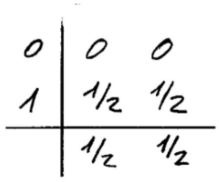


Figure 10.11: Butcher tableau of the Trapezoidal method.

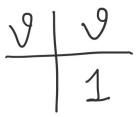


Figure 10.12: Butcher tableau of the parametric ϑ-method.

The resulting single-stage Runge-Kutta scheme is called the *ϑ-method*:

$$\begin{aligned} k_1 &= f(t + \vartheta h,\, x + h\vartheta k_1), \quad 0 \leq \vartheta \leq 1, \\ \bar{x} &= x + hk_1. \end{aligned}$$

The ϑ-method includes the following special cases:

- for $\vartheta = 0$: explicit Euler
- for $\vartheta = 1$: implicit Euler
- for $\vartheta = \frac{1}{2}$: implicit rectangular method

$$\bar{x} = x + hf\left(t + \frac{h}{2}, \frac{x + \bar{x}}{2}\right),$$

  obtained by using rectangular quadrature (see Section 8.1) in the derivation from Section 10.4.1.

All of these methods are optimal with $s = p = 1$, except for the implicit rectangular method, which has $p = 2$, but $s = 1$. Therefore, this is an example of a Runge-Kutta method with $p > s$, which is only possible in implicit methods. Explicit methods can reach at most $p = s$ (see Section 10.2.4). This is another example of how implicit methods can be numerically superior to explicit ones. In the rectangular method, the price paid for solving the implicit equation in each time step is rewarded by one additional order of convergence.

10.5 Multistep Methods

So far, all explicit and implicit methods we considered were one-step methods, for which a single propagator function F can be written. One-step methods

go from the present time step to the next without considering the past. Multistep methods, in contrast, also use past time points $\tilde{x}_{j-r+1}, \tilde{x}_{j-r+2}, \ldots, \tilde{x}_{j-1}$ together with the present time point $\tilde{x}_j$ to determine the next time point $\tilde{x}_{j+1}$. Such a method, using the present point plus $r-1$ past points, is called an *r-step method*. It is illustrated in Fig. 10.13. Intuitively, this uses more information about the past evolution of the solution, which should lead to a better, more accurate prediction of the next time point.

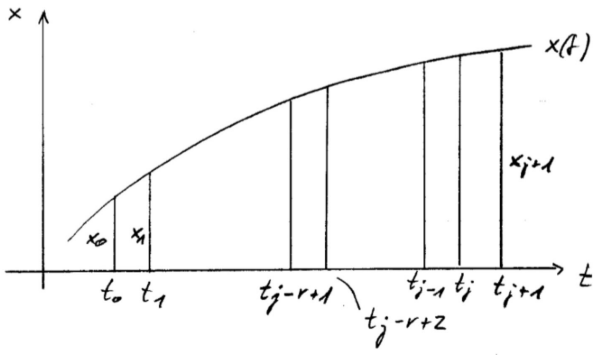


Figure 10.13: Illustration of an r-step method using the present plus $r-1$ past time points in order to compute the next point $j+1$.

While explicit one-step methods are usually derived from a numerical approximation of the time derivative, and implicit one-step methods from numerical quadrature, multistep methods are derived from polynomial interpolation.

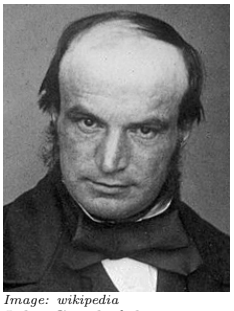


Image: wikipedia
John Couch Adams
∗ 5 June 1819
Laneast, UK
† 21 January 1892
Cambridge, UK

Image: Oxford University
Francis Bashforth
∗ 8 January 1819
Thurnscoe, UK
† 12 February 1912
Woodhall Spa, UK

10.5.1 An explicit multistep method: Adams-Bashforth

A classic explicit multistep method is the 3-step Adams-Bashforth method, invented by John Couch Adams, but published as part of a fluid dynamics application by Francis Bashforth (1883). For simplicity, we derive it for equidistant time steps $t_j = t_0 + jh$. Similar to Section 10.4.1, we start by integrating the IVP

$$\dot{x}(t) = f(t, x(t)), \quad x(t_0) = x_0$$

over one time step from t_j to t_{j+1}, yielding:

$$\underbrace{x(t_{j+1})}_{x_{j+1}} - \underbrace{x(t_j)}_{x_j} = \int_{t_j}^{t_{j+1}} f(t, x(t))\,\mathrm{d}t.$$

In this integral, we perform a change of variables from time t to normalized time s, as $t = t_j + sh$ (cf. Section 6.6.1). With the resulting $\mathrm{d}t = h\mathrm{d}s$, this gives:

$$x_{j+1} - x_j = h \int_0^1 \underbrace{f(t_j + sh, x(t_j + sh))}_{=:F(s)}\,\mathrm{d}s. \tag{10.18}$$

The function $F(s)$ in the integrand, evaluated at the present (j) and two past ($j-1$, $j-2$) time points, as illustrated in Fig. 10.14, corresponds to:

$$\begin{aligned} F(0) &= f(t_j, x_j) = f_j, \\ F(-1) &= f(t_{j-1}, x_{j-1}) = f_{j-1}, \\ F(-2) &= f(t_{j-2}, x_{j-2}) = f_{j-2}. \end{aligned}$$

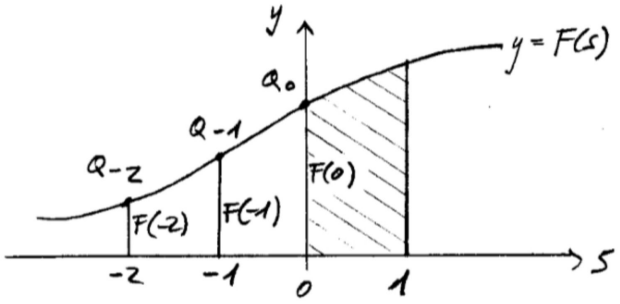


Figure 10.14: Illustration of the Adams-Bashforth method using three time points ($s = 0, -1, -2$) to predict the next time point $s = 1$).

This defines three points in the s–y plane: $Q_{-2} : (s = -2, y = F(-2))$, $Q_{-1} : (s = -1, y = F(-1))$, $Q_0 : (s = 0, y = F(0))$. The idea is now to use the interpolation polynomial $P_2(s)$ through Q_{-2}, Q_{-1}, Q_0 to approximate the function $F(s)$ for $s \in [0, 1]$. Note that the problems mentioned in the introduction to Chapter 9 do not occur here, since we are not using an interpolation polynomial to approximate a derivative, but to approximate the integral on the right-hand side of Eq. 10.18. Indeed, the shaded area under the curve $s \mapsto F(s)$ in Fig. 10.14) is the time-step increment. The interpolation polynomial can very well be used to approximate this area. Using Lagrange interpolation (see Section 6.2), we find:

$$P_2(s) = f_j \frac{(s+1)(s+2)}{2} - f_{j-1} s(s+2) + f_{j-2} \frac{s(s+1)}{2}.$$

The integral on the right-hand side of Eq. 10.18 is then approximated by analytically integrating the interpolation polynomial, as:

$$\int_0^1 F(s)\,\mathrm{d}s \approx \int_0^1 P_2(s)\,\mathrm{d}s = \frac{23}{12} f_j - \frac{16}{12} f_{j-1} + \frac{5}{12} f_{j-2}.$$

Substituting this in Eq. 10.18 yields the 3-step Adam-Bashforth method:

$$\begin{aligned} \tilde{x}_{j+1} &= \tilde{x}_j + \frac{h}{12}\left[23 f(t_j, \tilde{x}_j) - 16 f(t_{j-1}, \tilde{x}_{j-1}) + 5 f(t_{j-2}, \tilde{x}_{j-2})\right], \\ & j = 2, 3, 4, \ldots . \end{aligned} \tag{10.19}$$

This is an explicit (the right-hand side only uses the present and past time points of the solution), linear (the right-hand side is a linear combination of function evaluations), 3-step method. It is consistent with global error order $p = 3$ and local discretization error of $O(h^4)$. The third-order convergence is a direct consequence of the fact that the point-wise approximation error of an interpolation polynomial of degree q is of order $p = q + 1$ (see Theorem 6.1). Therefore, the local error per time step is bounded by this, times h.

As is obvious from Eq. 10.19, multistep methods are not self-initializing. In order to perform the first step of an r-step scheme, $x_0, x_1, \ldots, x_{r-1}$ need to be known, and the initial condition of the IVP is not sufficient to get the method initialized. The usual remedy is to first use a one-step scheme to compute the $r - 1$ initial steps, before switching to the r-step scheme. Of course, the initial one-step scheme must be of equal or higher order of convergence as the multistep scheme used thereafter in order to maintain solution quality.

An important benefit of multistep methods is their high computational efficiency. They achieve high convergence orders using few evaluations of the right-hand-side function. In the Adams-Bashforth scheme, for example, only one function evaluation is required at each time step, $F(0)$, in order to achieve global convergence order 3. The past time points needed by the formula are simply read from memory. An explicit one-step method would require at least 3 function evaluations to achieve the same order of convergence.

The main drawback of multistep methods is that dynamic step-size adaptation is virtually impossible without sacrificing computational efficiency[2].

10.5.2 An implicit multistep method: Adams-Moulton

Multistep methods become implicit if the future time point is included in the polynomial interpolation. A classic example is the Adams-Moulton method, developed by John Couch Adams. The name of Forst Ray Moulton became later associated with the method because he realized its use in the Predictor-Corrector scheme (see below). The Adams-Moulton method is based on the Adams-Bashforth method, but additionally uses the point $Q_1 : (s = 1, y = F(1))$ (see Fig. 10.14). Now using 4 points, we can determine the interpolation polynomial of degree 3, $P_3(s)$ and follow the same derivation to find the Adams-Moulton scheme:

Image: Physics Today
Forest Ray Moulton
* 29 April 1872
Le Roy, MI, USA
† 7 December 1952
Wilmette, IL, USA

$$\tilde{x}_{j+1} = \tilde{x}_j + \frac{h}{24}[9f(t_{j+1}, \tilde{x}_{j+1}) + 19f(t_j, \tilde{x}_j) - 5f(t_{j-1}, \tilde{x}_{j-1}) + f(t_{j-2}, \tilde{x}_{j-2})], \qquad j = 2, 3, \ldots. \tag{10.20}$$

This is also a 3-step method, despite the fact that the prediction is based on 4 points. The naming of multistep methods is only based on how many points of the present and past they use. But Adams-Moulton is an *implicit* 3-step method, as it also uses a future point. Adams-Moulton therefore is an implicit,

[2]Think about why this might be the case.

linear, 3-step method. It is consistent with global convergence order $p = 4$, since it is based on an interpolation polynomial of degree 3, and local error of $O(h^5)$. Like Adams-Bashforth, also Adams-Moulton is not self-initializing and an initial one-step method needs to be used to get it started. The implicit equation of Adams-Moulton can be solved well using fixed-point iteration with the value of the present time step as a starting point, i.e., $\tilde{x}_{j+1}^{(0)} = \tilde{x}_j$. If f is linear, a linear system solver can be used.

10.5.3 Predictor-Corrector method

A neat trick, realized by Forest Ray Moulton in 1926, is to use the explicit Adams-Bashforth method to predict the next time point and then use the Adams-Bashforth prediction $\hat{\tilde{x}}_{j+1}$ to evaluate the right-hand side of Eq. 10.20 and determine the final, corrected value of $\tilde{x}_{j+1}$:

$$\begin{aligned}
\hat{\tilde{x}}_{j+1} &= \tilde{x}_j + \frac{h}{12}\left[23f(t_j, \tilde{x}_j) - 16f(t_{j-1}, \tilde{x}_{j-1}) + 5f(t_{j-2}, \tilde{x}_{j-2})\right], \\
\tilde{x}_{j+1} &= \tilde{x}_j + \frac{h}{24}[9f(t_{j+1}, \hat{\tilde{x}}_{j+1}) + 19f(t_j, \tilde{x}_j) - 5f(t_{j-1}, \tilde{x}_{j-1}) \\
&\quad + f(t_{j-2}, \tilde{x}_{j-2})].
\end{aligned}$$

This is called the *Predictor-Corrector* method. It can be interpreted as using Adams-Bashforth to predict the starting value for the implicit solver of Adams Moulton (cf. Section 10.4.1), but perform only one fixed-point iteration instead of solving the implicit problem until convergence. This yields an overall explicit method with global order $p = 4$, which is the same convergence order as the implicit Adams-Moulton method. The absolute magnitude of the error and the numerical stability, however, are not as good as those of Adams-Moulton, because the implicit problem is not solved to convergence. The predictor-corrector method requires two function evaluations per time step ($f(t_j, \tilde{x}_j)$ and $f(t_{j+1}, \hat{\tilde{x}}_{j+1})$). It is therefore one of the most efficient (in terms of convergence order per function evaluation) explicit methods, making it a popular choice in practical applications.

Was this all there is?

Of course, the IVP solvers we discussed here are just examples and barely scratch the surface of the field. Time stepping and time integration are active areas of research, and a rich landscape of methods exists. We gave the general principles and classification of methods, so that other schemes should fall into place. All methods are based on one of the three principles discussed here: numerical approximation of the time derivative, numerical integration of the IVP, or numerical interpolation of the solution. For example, using Chebyshev interpolation instead of Lagrange interpolation when deriving a multistep scheme gives rise to the family of Chebyshev

integrators with their famous stability properties. Using trigonometric interpolation leads to time-harmonic methods.
In addition, special energy-preserving time-reversible integrators are available for equilibrium simulations. These are called *symplectic integrators* and they are important in applications of discrete particle methods, including molecular dynamics simulations and celestial or many-body mechanics. Famous examples of symplectic time integrators include the Størmer-Verlet method, the velocity-Verlet method, and Ruth's third- and fourth-order methods.
For stiff systems (see Section 10.8), robust time integration is a field of active and ongoing research. Methods include the Rosenbrock methods, Patankar schemes, and exponential integrators.
In addition to the rich landscape of time stepping methods, meta-methods also exist, which can be used in conjunction with any method from a certain family. This for example includes the super-time-stepping methods of Alexiades [Comm. Numer. Meth. Eng. 12(1), 1996], which can be used with any explicit one-step scheme in order to enable time-step sizes that are larger than what the scheme could normally tolerate. An elegant trick worth reading up on!

10.6 Systems of ODEs and Higher-Order ODEs

So far, we have considered the problem of numerically solving a scalar IVP with a single, first-order Ordinary Differential Equation (ODE). However, all methods we have presented so far generalize to systems of multiple ODEs and to ODEs of higher order, as we describe below.

10.6.1 Multidimensional systems of ODEs

An IVP that consists of multiple ODEs can be written in vector form as:

$$\dot{\vec{x}}(t) = \vec{f}(t, \vec{x}(t)), \quad \vec{x}(t = t_0) = \vec{x}_0$$

with solution:

$$\vec{x}(t) = \begin{bmatrix} x_1(t) \\ \vdots \\ x_n(t) \end{bmatrix} \in \mathbb{R}^n$$

and n right-hand-side functions

$$\vec{f}(t, \vec{x}) = \begin{bmatrix} f_1(t, x_1, \ldots, x_n) \\ \vdots \\ f_n(t, x_1, \ldots, x_n) \end{bmatrix}.$$

All numerical methods for solving IVPs apply component-wise.

Example 10.7. *Solving a system of ODEs using the implicit Euler method, we have the scheme*

$$\tilde{\vec{x}}^{(j+1)} = \tilde{\vec{x}}^{(j)} + h\vec{f}(t+h, \tilde{\vec{x}}^{(j+1)}).$$

Applied separately to each component of a system of $n = 2$ ODEs, this is:

$$\begin{aligned}\tilde{x}_1^{(j+1)} &= \tilde{x}_1^{(j)} + hf_1(t+h, \tilde{x}_1^{(j+1)}, \tilde{x}_2^{(j+1)}),\\ \tilde{x}_2^{(j+1)} &= \tilde{x}_2^{(j)} + hf_2(t+h, \tilde{x}_1^{(j+1)}, \tilde{x}_2^{(j+1)}),\end{aligned}$$

where time step indices are denoted as superscripts in parentheses in order to distinguish them from the vector components given as subscripts.

Also Runge-Kutta schemes generalize component-wise with $\vec{f}$, $\vec{x}$, and all $\vec{k}_i$ becoming n-vectors. The Butcher tableau remains exactly the same with the same number of entries as in the scalar case.

10.6.2 ODEs of higher order

The second generalization is to ODEs of higher order, i.e., to ODEs where the left-hand side is not a first derivative, but a second, third, or higher-order derivative. Such higher-order ODEs can always be written as systems of first-order ODEs by introducing auxiliary variables for the higher derivatives. This is illustrated in the following example.

Example 10.8. *Consider the third-order ODE*

$$\dddot{x}(t) = g(t, x, \dot{x}, \ddot{x})$$

with some right-hand side g. If we define the new variables $x_1 = x$, $x_2 = \dot{x}$, $x_3 = \ddot{x}$, this becomes:

$$\begin{aligned}\dot{x}_1 &= x_2,\\ \dot{x}_2 &= x_3,\\ \dot{x}_3 &= g(t, x_1, x_2, x_3).\end{aligned}$$

This is a system of $n = 3$ first-order ODEs, which can be numerically solved component-wise as presented above.

Since this can be done for arbitrary orders, and also for systems of higher-order ODEs, it is sufficient to be able to solve scalar first-order ODEs. Numerical methods, and their software implementations, therefore focus on the first-order scalar case without loss of generality.

Example 10.9. *As a practical example, consider the equation of motion of a pendulum with mass $m = 1$, length $l = 1$, and gravitational acceleration magnitude $g = 1$.The dynamics of the deflection angle x of the pendulum from the axis of gravity is governed by the second-order ODE*

$$\ddot{x} + \sin(x) = 0.$$

Using $x_1 = x$ and $x_2 = \dot{x}$, this reduces to a system of two first-order ODEs:

$$\dot{x}_1 = x_2,$$
$$\dot{x}_2 = -\sin(x_1),$$

which can be solved numerically component-wise. In this example, a symplectic time integrator would be a good choice for physical correctness of the numerical result, because the pendulum as modeled here is frictionless and energy should hence be conserved exactly.

10.7 Numerical Stability

So far, we have mainly discussed the accuracy of time-stepping and time-integration schemes, that is, their local and global errors and how these errors converge with decreasing time-step size. There is, however, another very important property that we are going to consider now: numerical stability. In practice, the requirement of numerical stability can limit the time-step size one is allowed to use for a certain scheme, and it may even limit the choice of scheme for a given problem.
Numerical stability is defined as follows:

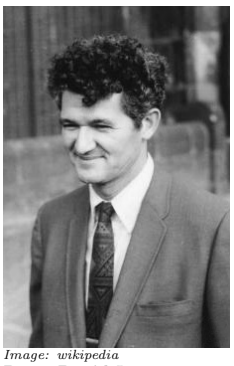


Image: wikipedia
Peter David Lax
* 1 May 1926
Budapest, Hungary

Image: alchetron
Germund Dahlquist
* 16 January 1925
Uppsala, Sweden
† 8 February 2005
Stockholm, Sweden

Definition 10.5 (Numerical stability)**.** *If the true solution $x(t)$ of an IVP is bounded for $t \longrightarrow \infty$, i.e., $\exists$ a constant C such that $|x(t)| < C \ \forall t$, then a numerical solution $\tilde{x}(t_j)$ is* stable *if and only if it is also bounded for $j \longrightarrow \infty$. For a given IVP $\dot{x} = f(t, x)$, $x(t_0) = x_0$, a numerical method is called* stable *if and only if it produces stable numerical solutions for* any *right-hand side f.*

Clearly, numerical stability is an elementary requirement for the numerical solution to make any sense. If time steps are chosen too big, it is possible for the numerical solution to diverge to infinity even if the true solution remains bounded. As a simple example, consider a sine wave. Clearly, it is bounded by $|\sin t| \leq 1$ for all t. If we compute a numerical solution using the explicit Euler scheme, and we use a time step that is identical to the period of the sine wave, i.e., $h = 2\pi$, then the numerical solution will always go up in every time step. This is because the sampling with resolution $h = 2\pi$ always hits the ascending flank of the wave. The numerical solution is therefore not bounded and will diverge to infinity for $t \to \infty$. The sine wave is therefore an example of a function for which the explicit Euler scheme is not stable for $h = 2\pi$.
It is of practical importance to know beforehand whether a given scheme is going to be stable for a given IVP and time-step size. This is determined by stability analysis of the numerical scheme, as we introduce below. Since in order to be called stable, a numerical method needs to be stable for *any* right-hand side f, we would have to check and prove stability for all possible right-hand sides f, which is clearly infeasible. Luckily, the following theorem holds:

Theorem 10.3 (Lax-Dahlquist Theorem)**.** *If a numerical IVP solver with given time-step size h produces stable solutions according to Definition 10.5 for the*

linear problem $\dot{x}(t) = \lambda x(t)$ for some set of constants $\lambda \in \Lambda$, then it also produces stable solutions for any nonlinear problem $\dot{x}(t) = f(t, x(t))$, as long as $\partial f/\partial x \in \Lambda$ and f is smooth enough. The set Λ is called the stability region *of the method.*

This theorem follows from the Hartman-Grobman theorem, and it states that "linear stability implies nonlinear stability". The converse, however, is not true. A method can be stable for a nonlinear right-hand side f even if it is unstable for the corresponding linearized function. The theorem thus states a sufficient condition for nonlinear stability, but not a necessary one. But this is good enough for us, because it means that we only need to carry out stability analysis for the linear problem

$$\dot{x} = \lambda x(t), \quad x(0) = x_0. \tag{10.21}$$

If a method produces stable solutions for this problem, we know that it also produces stable solutions for *any* right-hand side with the same slope and sufficient smoothness.
The above linear, first-order IVP can be solved analytically. Its solution is:

$$x(t) = x_0 \mathrm{e}^{\lambda t}.$$

This is bounded over $t \in [0, \infty)$ for $\lambda < 0$, as then in fact $x(t) \to 0$ for $t \to \infty$. Hence, we define:

Definition 10.6 (Linear stability)**.** *A numerical method is called* linearly stable *if and only if the numerical solution $\tilde{x}_j \to 0$ for $j \to \infty$ on the IVP $\dot{x} = \lambda x(t)$, $x(0) = x_0$, with $\lambda < 0$.*

Proving stability for linear f with negative λ is therefore sufficient. In general, the constant λ can be a complex number. For simplicity, however, we first consider the case of real λ. For systems of linear ODEs, λ is the largest eigenvalue of the system matrix.

10.7.1 Stability of the explicit Euler method for real λ

We start with a simple example that illustrates the general procedure of linear stability analysis. Consider the explicit Euler method from Section 10.2.1 with fixed time-step size h:

$$\tilde{x}_{j+1} = \tilde{x}_j + hf(t_j, \tilde{x}_j).$$

When applied to the linear problem from Definition 10.6, this becomes:

$$\begin{aligned} \tilde{x}_{j+1} &= \tilde{x}_j + h\lambda\tilde{x}_j \\ \tilde{x}_{j+1} &= \tilde{x}_j(1 + h\lambda). \end{aligned}$$

This means that for the linear problem, the explicit Euler scheme simply consists of multiplying the solution at each time step with a constant $d = 1 + h\lambda$. In order for the solution to go to zero for $j \to \infty$, we obviously require that $|d| < 1$, so that at each time step the solution is multiplied with a number less than one

in magnitude. Otherwise, the solution $\tilde{x}_{j+1} = d\tilde{x}_j$ would blow up to infinity for $j \to \infty$. Therefore, we find the condition

$$|1 + h\lambda| < 1$$

as a necessary and sufficient condition for linear stability of the explicit Euler method. Depending on the sign of d, there are two cases:

- If $d > 0$:
$$\begin{aligned} 1 + h\lambda &< 1 \\ h\lambda &< 0 \\ h &> 0 \end{aligned}$$
because $\lambda < 0$.

- If $d < 0$:
$$\begin{aligned} -1 - h\lambda &< 1 \\ -h\lambda &< 2 \\ -h &> \frac{2}{\lambda} \\ h &< \frac{2}{|\lambda|}. \end{aligned}$$

These are two conditions on the time-step size h. The first condition requires that we move forward in time, i.e., $h > 0$. This is obviously required for stability, because the solution of the linear problem grows to infinity when moving backward in time for $t \to -\infty$. The second condition requires that the time-step size is less than a positive constant $2/|\lambda|$. This also makes sense in light of the Nyquist-Shannon sampling theorem, because it states that we need to resolve every slope λ with at least 2 time points. Therefore, the stability region of the explicit Euler method is $0 < h < \frac{2}{|\lambda|}$. For time steps in this range, the explicit Euler method is stable for any IVP.
This is the general procedure of linear stability analysis. In order to extend this to more complicated time-stepping schemes, it is convenient to define $h\lambda = \mu$ and an amplification factor:

Definition 10.7 (Amplification factor)**.** *The factor $d(\mu)$ by which the numerical solution of a linear IVP gets multiplied in each time step of a numerical method, $\tilde{x}_{j+1} = d(\mu)\tilde{x}_j = d^{j+1}x_0$, $j = 0, 1, 2, \ldots$, is called the* amplification factor *of the numerical method.*

The amplification factor for the explicit Euler method therefore is: $d(\mu) = 1+\mu$. Note that this is the series expansion of the true solution of the linear IVP over one time step, $\mathrm{e}^{\mu} = 1 + \mu + \frac{\mu^2}{2!} + \frac{\mu^3}{3!} + \ldots$, up to order $O(\mu^2)$, which is the local error of the Euler method.

10.7.2 Stability of Heun's method for real λ

We analyze the stability of Heun's method from Section 10.2.3,

$$\tilde{x}_0 = x_0, \;\; \tilde{x}_{j+1} = \tilde{x}_j + \frac{h}{2}[f(t_j, \tilde{x}_j) + f(t_{j+1}, \tilde{x}_j + hf(t_j, \tilde{x}_j))],$$

which for the linear IVP becomes:

$$\begin{aligned}\tilde{x}_{j+1} &= \tilde{x}_j + \frac{h}{2}[\lambda\tilde{x}_j + \lambda(\tilde{x}_j + h\lambda\tilde{x}_j)] \\ &= \tilde{x}_j \left[1 + h\lambda + \frac{(h\lambda)^2}{2}\right].\end{aligned}$$

Therefore, with $\mu = h\lambda$, the amplification factor of Heun's method is:

$$d(\mu) = 1 + \mu + \frac{\mu^2}{2}.$$

Again, this is the series expansion of the true solution over one time step up to order $O(\mu^3)$, which is the local error of Heun's method. Indeed, the Heun method approximates the solution at the discrete time $t = jh$

$$x_0 \mathrm{e}^{\lambda t} = x_0 \mathrm{e}^{\lambda jh} = x_0 (\mathrm{e}^{\lambda h})^j$$

by the discrete approximation

$$x_0 (d(\lambda h))^j = x_0 \left(1 + \mu + \frac{\mu^2}{2}\right)^j = x_0 \left(\mathrm{e}^{\lambda h} + O(\mu^3)\right)^j.$$

This discrete solution asymptotically decreases to zero if and only if $|d(\mu)| < 1$ for $\lambda < 0$. The set of μ for which this is the case defines the stability region of Heun's method:

$$B = \{\mu : |d(\mu)| < 1\}.$$

We find the stability region by plotting the function $d(\mu)$, as shown in Fig. 10.15 We observe that the function $d(\mu)$ is always positive and has a value < 1 for $-2 < \mu < 0$, which implies $0 < h < \frac{2}{|\lambda|}$. The corresponding real stability region is:

$$B_{\text{Heun}} = (-2, 0).$$

For real λ, the Heun method therefore has the same stability properties as the explicit Euler method, but it is one order more accurate.

10.7.3 Stability of explicit Euler for complex λ

The linear IVP in Eq. 10.21 makes no assumptions about the constant λ. It can also be a complex number, $\lambda \in \mathbb{C}$. In this case, the amplification factor becomes a *stability function* $R(\mu) \in \mathbb{C}$ and we define in accordance with Theorem 10.3:

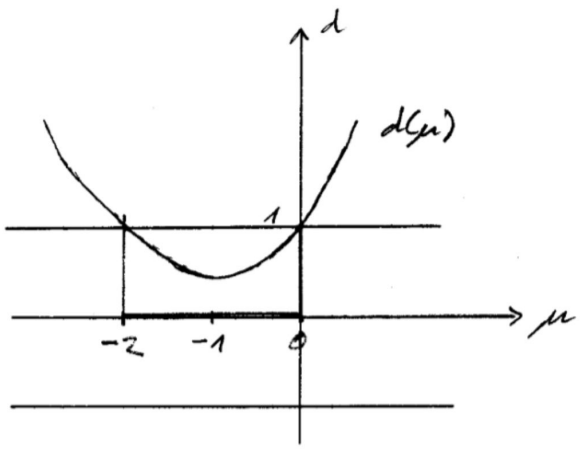


Figure 10.15: Plot of the amplification factor of Heun's method as a function of $\mu = h\lambda \in \mathbb{R}$.

Definition 10.8 (Stability region)**.** *The* stability region *of a numerical IVP solver is the set* $A = \{\mu \in \mathbb{C} : |R(\mu)| < 1\} \subset \mathbb{C}$*, where the magnitude of the stability function over the complex plane is less than 1.*

Again, we first illustrate the idea for the simplest case, the explicit Euler method, where we have, from above, the stability function:

$$R(\mu) = 1 + \mu,$$

but now for complex $\mu = u + iv$. For stability, we require:

$$\begin{aligned} |R(\mu)| &< 1 \\ |R(\mu)|^2 &< 1 \\ R\bar{R} &< 1 \\ (1+u+iv)(1+u-iv) = 1 + u - iv + u + u^2 - iuv + iv + iuv + v^2 &< 1 \\ 1 + 2u + u^2 + v^2 &< 1 \\ (u+1)^2 + v^2 &< 1, \end{aligned}$$

which defines the interior of a circle in the complex plane, with center $(-1, 0)$ and radius 1. This open disk is the stability region A of the explicit Euler method.

10.7.4 Stability of Heun's method for complex λ

For Heun's method in the complex plane, we have:

$$R(\mu) = 1 + \mu + \frac{\mu^2}{2}.$$

We now want to determine the boundary ∂A of the stability region

$$\partial A = \{\mu \in \mathbb{C} : |R(\mu)| = 1\}.$$

From the real case, we know that the points $-2 + 0i$ and $0 + 0i$ are on ∂A. We thus try the following ansatz for an egg-shaped outline (see Fig. 10.16) in polar complex coordinates (r, φ):

$$\begin{aligned} \mu &= -1 + re^{i\varphi}, \\ \mu^2 &= 1 - 2re^{i\varphi} + r^2 e^{2i\varphi}, \end{aligned}$$

for which the stability function becomes:

$$\begin{aligned} R(\mu) &= 1 + \mu + \frac{\mu^2}{2} = re^{i\varphi} + \frac{1}{2} - re^{i\varphi} + \frac{r^2}{2} e^{2i\varphi} \\ &= \frac{1}{2}(1 + r^2 e^{2i\varphi}). \end{aligned}$$

For the boundary ∂A, we have

$$1 \stackrel{!}{=} |R(\mu)|^2 = R\bar{R} = \frac{1}{4}(1 + 2r^2 \cos(2\varphi) + r^4),$$

which is a quadratic equation for r^2 with solution:

$$r^2 = -\cos(2\varphi) + \sqrt{\cos^2(2\varphi) + 3}.$$

Only the positive solution makes sense, as polar radii cannot be negative. This defines the egg-shaped stability region shown in Fig. 10.16. For $\varphi = 0, \pi$ we have $r = 1$, and for $\varphi = \frac{\pi}{2}, \frac{3\pi}{2}$ we have $r = \sqrt[4]{3}$.

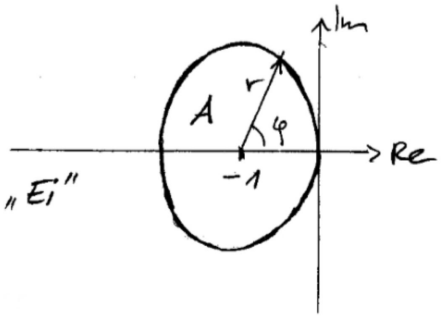


Figure 10.16: Stability region of Heun's method in the complex plane.

10.7.5 Stability of the implicit trapezoidal method

Next, we analyze the stability of an implicit method, the trapezoidal method from Section 10.4.1:

$$\tilde{x}_0 = x_0, \;\; \tilde{x}_{j+1} = \tilde{x}_j + \frac{h}{2}[f(t_j, \tilde{x}_j) + f(t_{j+1}, \tilde{x}_{j+1})],$$

which for the linear IVP with $f(t,x) = \lambda x$ becomes:

$$\begin{aligned} \tilde{x}_{j+1} = \tilde{x}_j + \frac{h}{2}[\lambda\tilde{x}_j + \lambda\tilde{x}_{j+1}] = \frac{h\lambda}{2}\tilde{x}_{j+1} + \left(1 + \frac{h\lambda}{2}\right)\tilde{x}_j \\ \left(1 - \frac{h\lambda}{2}\right)\tilde{x}_{j+1} = \left(1 + \frac{h\lambda}{2}\right)\tilde{x}_j \\ \tilde{x}_{j+1} = \frac{1 + \frac{h\lambda}{2}}{1 - \frac{h\lambda}{2}}\tilde{x}_j, \quad j = 0, 1, 2, \ldots \end{aligned}$$

Therefore, the amplification factor of the implicit trapezoidal method is:

$$d(\mu) = \frac{1 + \frac{\mu}{2}}{1 - \frac{\mu}{2}},$$

which is identical with the series expansion of the exact solution over one time step, e^{μ}, up to an error of order $O(\mu^3)$, which is the local error of the trapezoidal method. To see this, note that

$$\frac{1}{1 - \frac{\mu}{2}} = 1 + \frac{\mu}{2} + \frac{\mu^2}{4} + \ldots$$

and therefore:

$$d(\mu) = \left(1 + \frac{\mu}{2}\right)\left(1 + \frac{\mu}{2} + \frac{\mu^2}{4} + \ldots\right) = 1 + \mu + \frac{\mu^2}{2} + \cdots = \mathrm{e}^{\mu} + O(\mu^3).$$

For $\mu \in \mathbb{R}$, the function $d(\mu)$ is a hyperbola, as plotted in Fig. 10.17. Its absolute value is below one in the stability region $B_{\mathrm{Trapez}} = (-\infty, 0)$. This means that for $\lambda < 0$, the time step h can be arbitrarily large > 0.

For complex $\mu \in \mathbb{C}$, the stability function of the implicit trapezoidal method is:

$$R(\mu) = \frac{2 + \mu}{2 - \mu},$$

which is identical to the amplification factor in the real case. The boundary of

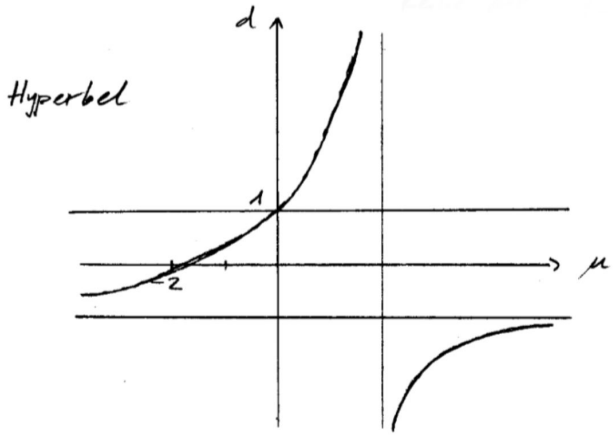


Figure 10.17: Plot of the amplification factor of the implicit trapezoidal method as a function of $\mu = h\lambda \in \mathbb{R}$.

the stability region is $\partial A = \{\mu : R\bar{R} = 1\}$. Let $\mu = u + iv$, then:

$$\begin{aligned} R\bar{R} &= \frac{(2+u+iv)(2+u-iv)}{(2-u-iv)(2-u+iv)} \\ &= \frac{(2+u)^2+v^2}{(2-u)^2+v^2} \overset{!}{=} 1 \\ (2+u)^2 &= (2-u)^2 \\ 4u &= -4u \\ 8u &= 0 \\ u &= 0. \end{aligned}$$

Therefore, ∂A is the imaginary axis. The stability region A is the entire half-plane to the left of the imaginary axis:

$$A = \{\mu \in \mathbb{C} : \mathrm{Re}(\mu) < 0\}.$$

This means that for any λ with $\mathrm{Re}(\lambda) < 0$, for which the true solution is bounded, the numerical solution is stable for arbitrary time-step sizes $h > 0$. Methods that have this property are called *A-stable* (for: "absolutely stable"). They are stable for any time step, on any IVP.

Definition 10.9 (A-stable)**.** *A numerical IVP solver is called* A-stable *if and only if it is numerically stable for any time-step size* $h > 0$.

This is the optimal stability an IVP solver can achieve and illustrates another important numerical advantage of implicit methods over explicit methods. Indeed, all implicit methods are A-stable.

Have you noticed it?

In each of the above stability analyses, we have observed that the amplification factor of a numerical method approximates the true solution of the IVP over one time step with the same order of accuracy as the local error of the scheme itself. Indeed, it is generally true for any explicit or implicit method from the Runge-Kutta family with global order of convergence p that

$$R(\mu) = 1 + \mu + \frac{\mu^2}{2!} + \cdots + \frac{\mu^p}{p!} + O(\mu^{p+1})$$

so that

$$R(\mu) - \mathrm{e}^{\mu} = O(\mu^{p+1}) = O(h^{p+1}).$$

This makes is very easy to write down the formula for the amplification factor of any explicit Runge-Kutta method. For implicit Runge-Kutta methods, $R(\mu)$ is not a polynomial, but a rational function. This is the source of the absolute stability property of implicit schemes.

10.8 Stiff Initial Value Problems

As numerical stability limits the time-step size of explicit methods by the local slope of the right-hand side, λ, problems that incur a wide range of λ values with greatly differing magnitudes are particularly difficult to solve. Such problems are called *stiff*. For systems of ODEs, the same is true when λ is defined as the largest eigenvalue of the system matrix of the linear(ized) ODEs.

Example 10.10. *As an illustrative example, consider the second-order IVP*

$$\ddot{x} + 101\dot{x} + 100x = 0, \quad x(0) = 0,\, \dot{x}(0) = 0.99.$$

This is equivalent to the system of first-order ODEs

$$\begin{aligned} \dot{x}_1 &= x_2, \\ \dot{x}_2 &= -100x_1 - 101x_2. \end{aligned}$$

The system matrix of this linear problem $\dot{\vec{x}} = \mathbf{A}\vec{x}$ *is* $\mathbf{A} = \begin{bmatrix} 0 & 1 \\ -100 & -101 \end{bmatrix}$, *and it has the eigenvalues* $\lambda_1 = -1$ *and* $\lambda_2 = -100$. *The analytical solution of this problem therefore is* $x(t) = 0.01\mathrm{e}^{-t} - 0.01\mathrm{e}^{-100t}$, *which is plotted in Fig. 10.18. The solution consists of a boundary layer where the fast exponential dominates, and an outer layer where the slow exponential dominates.*
To numerically approximate this solution, the time-step size must be small enough to properly resolve the initial, fast dynamics. This small time step must be used even in the slower tail, so dynamic step-size adaptation does not help. To see this, consider a time point outside the boundary layer, say $t = 0.2$, *where*

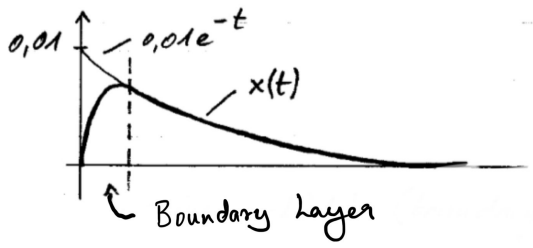


Figure 10.18: Analytical solution of the stiff problem in Example 10.10.

the fast exponential plays almost no role any more:

$$t = 0.2: \quad \mathrm{e}^{-100t} = \mathrm{e}^{-20} \approx 2 \cdot 10^{-9}.$$

Starting from here and taking time steps $j = 0, 1, 2, \ldots$ of size h, we have:

$$x(0.2 + jh) = 0.01\mathrm{e}^{-0.2}(\mathrm{e}^{-h})^j - 0.01\mathrm{e}^{-20}(\mathrm{e}^{-100h})^j.$$

Using a numerical scheme with amplification factor d, this becomes:

$$\tilde{x}(0.2 + jh) = 0.01\mathrm{e}^{-0.2}(d(-h))^j - 0.01\mathrm{e}^{-20}(d(-100h))^j$$

for $\mu = h\lambda$ of the two eigenvalues $\lambda_1 = -1$ and $\lambda_2 = -100$. For a time-step size of $h = 0.05$, we find for example:

$$d_{Heun}(-h) \approx 0.95, \quad d_{Trapez}(-h) \approx 0.95$$

$$d_{Heun}(-100h) \approx \mathbf{8.5}, \quad d_{Trapez}(-100h) \approx -0.43.$$

This means that even outside of the boundary layer, Heun's method is unstable for this step size, because $-100h = -5$ is outside the stability region of the Heun method. This is despite the fact that the fast exponential plays almost no role any more (but is still there!) outside the boundary layer. Dynamically adaptive step sizes would therefore not help. The implicit trapezoidal method, however, is stable, as the amplification factors associated with both eigenvalues are below one in absolute value.

Definition 10.10 (Stiff IVP)**.** *A linear system of ordinary differential equations $\dot{\vec{x}} = \mathbf{A}\vec{x} + \vec{b}$ with $\mathbf{A} \in \mathbb{R}^{n \times n}$ is* stiff *if and only if the eigenvalues of $\mathbf{A}$ have strongly different negative real parts. For nonlinear IVPs, $\mathbf{A}$ is the Jacobian of the linearized right-hand side.*

For systems of nonlinear ordinary differential equations, the qualitative behavior of the solution is described locally by the eigenvalues of the Jacobi matrix (see Definition 5.1) of the linearized system. This extends the definition of stiffness to nonlinear problems.
Using explicit methods for stiff problems requires extremely small time steps. This eventually means that h may drop below machine epsilon, in which case

the simulation does not advance at all and the problem becomes numerically unsolvable with explicit methods. Special "stiff explicit integrators" are available to address this problem, for example Rosenbrock schemes (Howard Rosenbrock, 1963) or exponential integrators. The safest solution for stiff problems, however, is to use implicit methods, as they are A-stable.

Image: wikipedia
Howard Harry Rosenbrock
∗ 16 December 1920
Ilford, UK
† 21 October 2010
London, UK

10.9 The Lax Equivalence Theorem

Now that we are acquainted with the notions of stability, consistency, and convergence of numerical IVP solvers, we can state one of the fundamental theorems of numerical computation, the Lax Equivalence Theorem due to Peter David Lax and Robert D. Richtmyer, who published it in 1956.

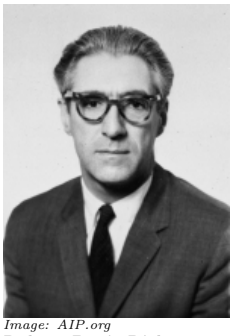


Image: AIP.org
Robert Davis Richtmyer
∗ 10 October 1910
Ithaca, NY, USA
† 24 September 2003
Gardner, CO, USA

Theorem 10.4 (Lax Equivalence Theorem)**.** *A consistent numerical approximation to a well-posed IVP is convergent if and only if it is stable.*

Consistency states that the numerical operators, i.e., the numerically approximated derivatives or integrals, correspond to the original continuous operators up to some truncation error. However, it does not imply that the numerical solution computed by any algorithm using these operators converges to the true solution of the IVP. For this to happen, the algorithm also needs to be stable. Only the combination of consistency and stability guarantees convergence, and is also necessary for it. This means that any statement about error orders is only valid if the method is stable. Then, the methods are asymptotically exact for $h \to 0$.

Chapter 11

Numerical Solution of Partial Differential Equations

Numerically solving partial differential equations (PDEs) is probably the most important application of numerical methods in practice and is the crown jewel of scientific computing. It relates to the field of numerical analysis, which is vast in and of itself. We can only scratch the surface here, and there are many lectures and books that continue the topic. The goal here is to provide examples and intuition rather than a concise treatise on the subject.
A PDE is an equation for an unknown function in multiple variables. This function is then to be determined by solving the PDE subject to certain boundary and/or initial conditions. Formally, we define:

Definition 11.1 (Partial Differential Equation)**.** *A partial differential equation (PDE) is an equation that relates partial derivatives of an unknown function $u : \Omega \to \mathbb{R}^n$ in multiple variables, i.e., $\Omega \subset \mathbb{R}^d$ with $d > 1$.*

Broadly, numerical methods for PDEs fall into three categories:

1. Collocation methods are based on sampling the unknown function u at a finite number of discrete points (called *collocation points*) and approximating partial derivatives using the methods from the previous chapters. A famous example are *finite-difference methods*.

2. Galerkin methods are based on expanding the unknown function in terms of the basis functions ζ of a function space $u(\cdot) = \sum_i w_i \zeta_i(\cdot)$ and numerically solving for the known coefficients w_i of this expansion, i.e., for the "coordinates" of the solution in this function space. A famous example are *finite-element methods*.

Image: wikipedia
Boris Grigoryevich Galerkin
∗ 4 March 1871
Polotsk, Russian Empire
† 12 July 1945
Moscow, USSR

3. Spectral methods are based on frequency-space transformations, followed by numerical solution of the equation in the transformed space and back-transformation of the solution. A famous example are *Fourier* and *Laplace transformation* methods.

Each of these categories of methods has many members and comes with its own advantages and disadvantages. Galerkin methods, for example, often solve the weak form of the PDE, which copes well with discontinuities in the solution, but may find spurious solutions. Spectral methods are often accurate to machine precision with exponentially fast convergence, but they are computationally costly and do not parallelize well. Collocation methods are easy to implement and parallelize, but often have inferior stability and accuracy. As always, there is no free lunch.
Here, we only consider methods of collocation type. Rather than delving into a general theory of collocation methods for arbitrary PDEs, however, we illustrate the main concepts in three examples that we use as model problems:

Example 11.1. ***The 2D Poisson equation.***

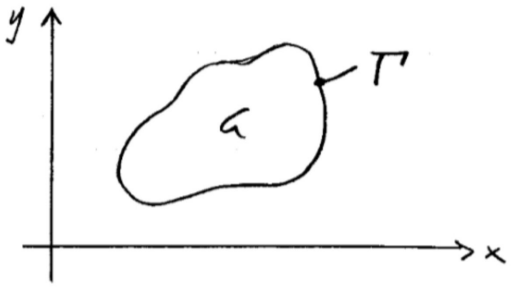


Figure 11.1: Domain and boundary for the 2D Poisson problem.

Given are:

- *a domain* $G \subset \mathbb{R}^2$ *with boundary* Γ*, see Fig. 11.1,*
- *a sufficiently smooth continuous function* $f(x, y) : G \to \mathbb{R}$, $(x, y) \in G \mapsto f(x, y)$*,*
- *a function* $\varphi(x, y) : \Gamma \to \mathbb{R}$, $(x, y) \in \Gamma \mapsto \varphi(x, y)$*.*

We want to find a continuous function $u(x, y)$ *such that:*

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y) \qquad \text{in } G, \tag{11.1}$$

$$u(x,y) = \varphi(x,y) \qquad \text{on } \Gamma. \tag{11.2}$$

This is the Poisson equation with Dirichlet boundary conditions. Because solving this equation is a problem with only boundary conditions, this type of problem is called a boundary value problem *(BVP).*

Example 11.2. ***The 1D heat equation.***

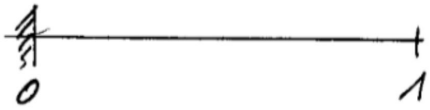


Figure 11.2: The unit interval for the 1D heat equation.

Image: wikipedia
Siméon Denis Poisson
∗ 21 June 1781
Pithiviers, Kingdom France
† 25 April 1840
Sceaux, July Monarchy

Given are:

- *the one-dimensional interval* $(0,1)$ *as shown in Fig. 11.2,*
- *a sufficiently smooth continuous function* $f(x)$ *for* $x \in (0,1)$*,*
- *two functions* $\alpha(t)$ *and* $\beta(t)$ *for* $t \geq 0$*.*

We want to find a continuous function $u(t,x)$ *such that:*

$$\frac{\partial u(t,x)}{\partial t} = \frac{\partial^2 u(t,x)}{\partial x^2}, \tag{11.3}$$

$$u(0,x) = f(x), \quad x \in (0,1), \tag{11.4}$$

$$u(t,0) = \alpha(t), \quad t \geq 0, \tag{11.5}$$

$$u(t,1) = \beta(t), \quad t \geq 0. \tag{11.6}$$

This is the heat (or diffusion) equation with boundary and initial conditions. Because solving this equation is a problem with both boundary and initial conditions, this type of problem is called an initial boundary value problem *(IBVP).*

Example 11.3. ***The 1D wave equation.***
Given are:

- *the set of real numbers* $x \in \mathbb{R}$ *as domain,*
- *two functions* $\varphi(x)$*,* $\psi(x)$ *for* $x \in \mathbb{R}$*.*

We want to find a continuous function $u(t,x)$ *such that:*

$$\frac{\partial^2 u(t,x)}{\partial t^2} = c^2 \frac{\partial^2 u(t,x)}{\partial x^2}, \tag{11.7}$$

$$u(0,x) = \varphi(x), \tag{11.8}$$

$$\frac{\partial u(0,x)}{\partial t} = \psi(x), \tag{11.9}$$

for some constant $c \neq 0$ *and* $t \geq 0$*. This is the wave equation with initial conditions for* u *and for its first derivative. Because solving this equation is a problem with only initial conditions, this type of problem is called an* initial value problem *(IVP) of a partial differential equation.*

Classification of PDEs

Partial differential equations can be classified into families in two ways: (1) according to their form and (2) according to their solvability. Classification according to form includes:

- **Order**: The *order* of a PDE is defined by the highest occurring derivative in the equation. PDEs of order >4 hardly occur in practice. The most important (physical) PDEs are second order. There are, therefore, special sub-classes defined for second-order PDEs (see below).
- **Linearity**: PDEs that only contain linear combinations of the unknown function and its derivatives are called *linear*. *Nonlinear* PDEs also contain nonlinear combinations (e.g., products) of the unknown function or its derivatives. In the class of nonlinear PDEs, we further distinguish PDEs that are linear in the highest occurring derivative as *quasi-linear* PDEs. Nonlinear PDEs are a difficult topic, since no closed theory exists for this class of equations. They are notoriously difficult to solve and numerical simulations frequently suffer from instabilities and ill-posedness.
- **Homogeneity**: PDEs where all terms, i.e. summands, contain the unknown function or one of its derivatives are called *homogeneous*. A PDE is *inhomogeneous* if at least one term is independent of the unknown function and its derivatives.
- **Coefficients**: PDEs where all pre-factors of the unknown function and its derivatives in all terms are independent of the independent variables (e.g., space or time) are called *PDEs with constant coefficients*. If any coefficient explicitly depends on an independent variable, the PDE has *varying coefficients*.

Depending on order, linearity, homogeneity, and coefficients of a PDE, one can know which numerical solver to choose and how many side conditions, i.e., boundary and initial conditions, are required. All of the above classifications are easy to determine and to check by inspecting the equation. Classification according to solvability (due to Jacques Hadamard) is not obvious, but very useful if known. It classifies PDEs according to:

1. **Solution existence**: PDEs for which a solution *exists*. Solution existence is mostly proven by showing that the assumption that no solution exists leads to a contradiction.

2. **Solution uniqueness**: PDEs that have only one, *unique* solution. If a solution is found, then one knows that it is the only possible solution.

3. **Solution stability**: A solution is *stable* if and only if small perturbations in the initial and/or boundary conditions of the PDE only lead to small (i.e., bounded proportional to the magnitude of the perturbation) variations in the solution.

Proving any of these points for a given PDE is usually a lot of work, but for many well-known PDEs, the results are known. PDEs that do not fulfill *all three* of the above conditions are called *ill-posed* and are hard or impossible to solve numerically. PDEs that fulfill all three conditions are called *well-posed*.

Image: wikipedia
Jacques Hadamard
∗ 8 December 1865
Versailles, France
† 17 October 1963
Paris, France

All of Examples 11.1–11.3 are linear PDEs of second order with constant coefficients. The Poisson equation is inhomogeneous, the other two are homogeneous. Linear second-order PDEs with constant coefficients constitute the most common class of PDEs in practical applications from physics and engineering. Any linear PDE of second order with constant coefficients in two variables can be written in the following general form:

$$A\frac{\partial^2 u(x,y)}{\partial x^2} + 2B\frac{\partial^2 u(x,y)}{\partial x \partial y} + C\frac{\partial^2 u(x,y)}{\partial y^2} + D\frac{\partial u(x,y)}{\partial x} + E\frac{\partial u(x,y)}{\partial y} + Fu(x,y) = f(x,y),$$

with coefficients $(A, B, C) \neq (0, 0, 0)$. Based on this generic form, such equations are classified by computing the number $\Delta := AC - B^2$, and defining three subclasses of linear second-order PDEs:

1. $\Delta > 0$: elliptic PDE (e.g., Poisson equation),
2. $\Delta = 0$: parabolic PDE (e.g., heat equation),
3. $\Delta < 0$: hyperbolic PDE (e.g., wave equation).

Depending on the sub-class, different numbers of side conditions (i.e., initial and boundary conditions) are required to render the problem well-posed. Elliptic equations require boundary conditions, parabolic equations require boundary and initial conditions, and hyperbolic equations require two initial conditions. In addition, the type of methods appropriate for numerically solving an equation depends on the sub-class. In the following, we give examples of such methods for each sub-class for the model problems defined above.

11.1 Parabolic Problems: The Method of Lines

We first consider the numerical solution of parabolic PDEs using the 1D heat equation from Example 11.2 as a model problem with specific boundary conditions $u(t, 0) = u(t, 1) = 0$, $t \geq 0$. The method of lines is frequently used to

discretize parabolic PDEs. It is based on first discretizing the space coordinate x and then use numerical methods for initial-value problems of ordinary differential equations (see Chapter 10) to solve over time at each spatial discretization point.

In our model problem, we therefore start by discretizing $\frac{\partial^2 u}{\partial x^2}$. We choose a symmetric finite difference of second order to do so, hence:

$$\frac{\partial^2 u(t,x)}{\partial x^2} = \frac{u(t,x+h) - 2u(t,x) + u(t,x-h)}{h^2} + O(h^2)$$

on a regular grid with discretization points $x_l = lh$, $l = 0, 1, 2, \ldots, N$. At each of these spatial points, we then have an ODE over time, which can be visualized as a line at location x_l over all $t \geq 0$, as shown in Fig. 11.3, hence the name of the method.

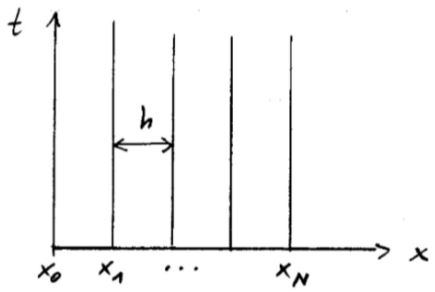


Figure 11.3: Illustration of the method of lines, obtained by a semi-discretization of the PDE in space followed by solving N initial-value problems over time along the lines $(x_l, t \geq 0)$.

Along each line, we have $u_l(t) := u(t, x_l)$. Therefore, we have N unknown functions in one variable (here: t) governed by the following ODEs:

$$\begin{aligned} u_0(t) &= 0, \quad u_N(t) = 0 \quad \text{from the boundary conditions} \\ \frac{\mathrm{d}u_l(t)}{\mathrm{d}t} &= \frac{u_{l+1}(t) - 2u_l(t) + u_{l-1}(t)}{h^2}, \quad l = 1, \ldots, N-1 \\ u_l(0) &= f(x_l) \quad \text{from the initial condition in Eq. 11.4.} \end{aligned}$$

Using the short-hand notation $\dot{u}_l = \frac{\mathrm{d}u_l(t)}{\mathrm{d}t}$, this can be written in matrix-vector notation as:

$$\begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \\ \dot{u}_3 \\ \vdots \\ \dot{u}_{N-1} \end{bmatrix} = \frac{1}{h^2} \underbrace{\begin{bmatrix} -2 & 1 & 0 & 0 & \ldots & \\ 1 & -2 & 1 & 0 & \ldots & \\ 0 & 1 & -2 & 1 & \ldots & \\ \vdots & \vdots & \vdots & \ddots & \ddots & \\ & & & & 1 & -2 \end{bmatrix}}_{\hat{\mathbf{A}}} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \end{bmatrix},$$

which makes use of the fact that $u_0 = u_N = 0$. In general, there can also be non-zero entries for the boundary conditions on the right-hand side. In any case, this defines a system of linear ordinary differential equations

$$\dot{\vec{u}} = \mathbf{A}\vec{u} \quad \text{with} \quad \mathbf{A} = \frac{1}{h^2}\hat{\mathbf{A}} \in \mathbb{R}^{(N-1)\times(N-1)}$$

a tri-diagonal matrix, and initial condition

$$\vec{u}(0) = \vec{f} = \begin{bmatrix} f(x_1) \\ \vdots \\ f(x_{N-1}) \end{bmatrix}.$$

We can numerically solve these ODEs using any of the methods from Chapter 10.
Analyzing this system of linear ODEs, we find that it becomes increasingly stiff for $N \gg 1$ (cf. also section 10.8). Indeed, $\hat{\mathbf{A}}$ has eigenvalues

$$\hat{\lambda}_l = -4\sin^2\left(\frac{l\pi}{2N}\right)$$

and, therefore, $\mathbf{A}$ has eigenvalues

$$\lambda_l = -\frac{4}{h^2}\sin^2\left(\frac{l\pi}{2N}\right), \quad l = 1, \ldots, N-1. \tag{11.10}$$

For $N = \frac{1}{h} \gg 1$, we find for the largest and smallest eigenvalues:

$$\begin{aligned} l = 1: &\quad \sin\left(\frac{\pi}{2N}\right) \approx \frac{\pi}{2N} \Longrightarrow \lambda_1 \approx -\pi^2 \in O(1), \\ l = N-1: &\quad \sin\left(\frac{(N-1)\pi}{2N}\right) \approx 1 \Longrightarrow \lambda_{N-1} \approx \frac{-4}{h^2} \in O(h^{-2}). \end{aligned}$$

Therefore, for $h \ll 1$ (i.e., $N \gg 1$), λ_{N-1} is much larger than λ_1 and in fact the stiffness grows quadratically with N. This stiffness also becomes obvious by writing the system of ODEs in diagonalized form as $\dot{\vec{v}}_l = \lambda_l \vec{v}_l$, where $\vec{v}_l$ is the eigenvector corresponding to the eigenvalue λ_l.

11.1.1 The explicit Richardson method

As a first example, we use the explicit Euler method along the lines, hence discretizing time as $t_j = j\bar{h}$ for $j = 0, 1, 2, \ldots$ with time-step size $\bar{h}$. This yields:

$$\vec{u}(t_{j+1}) \approx \vec{u}(t_j) + \bar{h}\mathbf{A}\vec{u}(t_j).$$

For the numerical approximation of the solution $\tilde{\vec{u}}_j \approx \vec{u}(t_j)$, we then find:

$$\tilde{\vec{u}}_{j+1} = (\mathbf{1}_{N-1} + \bar{h}\mathbf{A})\tilde{\vec{u}}_j = \left[\mathbf{1}_{N-1} + \frac{\bar{h}}{h^2}\hat{\mathbf{A}}\right]\tilde{\vec{u}}_j, \quad j = 0, 1, 2, \ldots \tag{11.11}$$

This solver, obtained by applying explicit Euler time-stepping in a method of lines, is called *Richardson Method* for parabolic PDEs (Lewis Fry Richardson, 1910). Here, $\mathbf{1}_{N-1}$ is the identity matrix of size $N-1$. the above formula is second-order accurate in space with h and first-order accurate in time with $\bar{h}$. The method amounts to a single matrix-vector multiplication per time step with a computational cost of $O(N^2)$ per time step. Since the matrix $\mathbf{A}$ is typically very large, schemes of order higher than 2 in time are rarely used.
In order to analyze the numerical stability of the Richardson method, we start from the stability region of the explicit Euler method for real eigenvalues, as all λ_l are real numbers in this example (see above). This stability region is: $\bar{h}\lambda_l \in (-2, 0)$ (see Sec. 10.7.1). Using Eq. 11.10, this produces the following condition for the time step size $\bar{h}$:

$$-\frac{4\bar{h}}{h^2}\sin^2\left(\frac{l\pi}{2N}\right) \in (-2, 0)$$

and therefore:

$$0 < \frac{\bar{h}}{h^2} < \frac{1}{2\sin^2\left(\frac{l\pi}{2N}\right)}$$

for $l = 1, \ldots, N-1$, where the sine is never equal to 0, nor 1. Because $\sin^2\left(\frac{l\pi}{2N}\right) \in (0, 1)$, stability is guaranteed for arbitrary N as long as

$$0 < \bar{h} \leq \frac{1}{2}h^2. \tag{11.12}$$

This implies very small time steps $\bar{h} > 0$ for any discretization with high spatial resolution $0 < h \ll 1$. In fact, the time step size has to shrink proportional to the square of the spatial resolution. This is a direct result of the stiffness of the equation system and often motivates the use of implicit methods in practice.

11.1.2 The implicit Crank-Nicolson method

A popular choice of an implicit time-integration method for the IVPs arising in method of lines is the implicit trapezoidal method from Section 10.4.1. In the present example, this means approximating the integrals

$$\int_{t_j}^{t_{j+1}} \dot{\vec{u}}(t)\,\mathrm{d}t = \int_{t_j}^{t_{j+1}} \mathbf{A}\vec{u}(t)\,\mathrm{d}t$$

using trapezoidal quadrature with spacing $\bar{h}$ in time, leading to:

$$\vec{u}(t_{j+1}) - \vec{u}(t_j) \approx \frac{\bar{h}}{2}[\mathbf{A}\vec{u}(t_j) + \mathbf{A}\vec{u}(t_{j+1})].$$

In this case, because it is a linear problem, we can separate the two time points as:

$$\left(\mathbf{1} - \frac{\bar{h}}{2}\mathbf{A}\right)\tilde{\vec{u}}_{j+1} = \left(\mathbf{1} + \frac{\bar{h}}{2}\mathbf{A}\right)\tilde{\vec{u}}_j,$$

producing the final scheme:

$$\tilde{\vec{u}}_{j+1} = \left[\mathbf{1} - \frac{\bar{h}}{2h^2}\hat{\mathbf{A}}\right]^{-1} \left[\mathbf{1} + \frac{\bar{h}}{2h^2}\hat{\mathbf{A}}\right] \tilde{\vec{u}}_j, \quad j = 0, 1, 2, \ldots \qquad (11.13)$$

This is the *Crank-Nicolson* method for parabolic PDEs, named after John Crank and Phyllis Nicolson who developed and published it in 1947. It uses an implicit time-integration method, which means that a linear system of equations needs to be solved at each time step, inverting the matrix $\left[\mathbf{1} - \frac{\bar{h}}{2h^2}\hat{\mathbf{A}}\right]$. Since this matrix is the same for all j, this can efficiently be done by performing LU-decomposition of $\hat{\mathbf{A}}$ once, and then simply using forward and backward substitution in each time step. The method then also has a computational cost of $O(N^2)$ per time step, plus an $O(N^3)$ initialization effort. It therefore has practically the same computational cost as the Richardson method.
Since the trapezoidal method is A-stable (see Sec. 10.7.5), the Crank-Nicolson method does not impose any limit on the time step size $\bar{h} > 0$. This is its main advantage over the Richardson method. Also, Crank-Nicolson is 2nd-order accurate in both space and time.

Image: wikipedia
John Crank
∗ 6 February 1916
Hindley, Lancashire, UK
† 3 October 2006
United Kingdom

Image: wikipidia
Phyllis Nicolson
∗ 21 September 1917
Macclesfield, Cheshire, UK
† 6 October 1968
Sheffield, Yorkshire, UK

11.2 Elliptic Problems: Stencil Methods

As a model problem for the elliptic case, we consider the 2D Poisson equation from Example 11.1, using the unit square as domain G, i.e., $0 \leq x \leq 1$; $0 \leq y \leq 1$. In a uniform stencil method, the domain is discretized in all independent variables (here: x, y) using a regular Cartesian (named after René Descartes, who published this coordinate system in 1637) lattice with spacing $h = 1/N$, as illustrated in Fig. 11.4.

Image: wikipedia
René Descartes
∗ 31 March 1596
La Haye en Touraine, France
† 11 February 1650
Stockholm, Swedish Empire

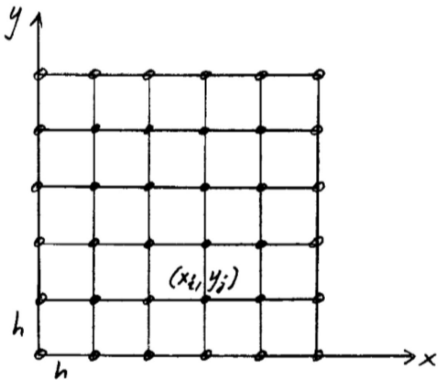


Figure 11.4: The 2D unit square discretized by a regular Cartesian lattice of spacing h.

This lattice has nodes $(x_i, y_j) = (ih, jh)$ for $i, j = 0, 1, \ldots, N$. At the boundary nodes of this lattice, i.e., for $i = 0$, $i = N$, $j = 0$, or $j = N$, the solution is known from the boundary condition $\varphi(x_i, y_j)$. At the interior points, we compute a numerical approximation to the solution $u(x_i, y_j) \approx \tilde{u}(x_i, y_j) = \tilde{u}_{ij}$. In the present example, we choose second-order symmetric finite differences to approximate the derivatives in the PDE (see also Chapter 9), hence:

$$\frac{\partial^2 u(x,y)}{\partial x^2} \approx \frac{u(x+h,y) - 2u(x,y) + u(x-h,y)}{h^2},$$
$$\frac{\partial^2 u(x,y)}{\partial y^2} \approx \frac{u(x,y+h) - 2u(x,y) + u(x,y-h)}{h^2}.$$

Summing these two, and evaluating only at interior grid points (x_i, y_j), yields:

$$\left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right]_{x=x_i, y=y_i} \approx$$
$$\frac{1}{h^2}[\tilde{u}_{i+1,j} + \tilde{u}_{i-1,j} + \tilde{u}_{i,j+1} + \tilde{u}_{i,j-1} - 4\tilde{u}_{i,j}], \quad i, j = 1, 2, \ldots, N-1.$$

For the numerical approximation $\tilde{u}_{ij} = \tilde{u}(x_i, y_j)$. Using the notation $f_{ij} = f(x_i, y_j)$, this yields the following discretized form for Eq. 11.1:

$$\frac{1}{h^2}[\tilde{u}_{i+1,j} + \tilde{u}_{i-1,j} + \tilde{u}_{i,j+1} + \tilde{u}_{i,j-1} - 4\tilde{u}_{ij}] = f_{ij}, \quad i, j = 1, 2, \ldots, N-1. \tag{11.14}$$

This is called a *stencil method*, as the left-hand side of the discretized equation can be seen as a stencil that iterates over the mesh from Fig. 11.4. The particular stencil in this example is visualized in Fig. 11.5. In general, *stencil operators* are linear combinations of grid point values in a certain, local neighborhood called the *stencil support*.

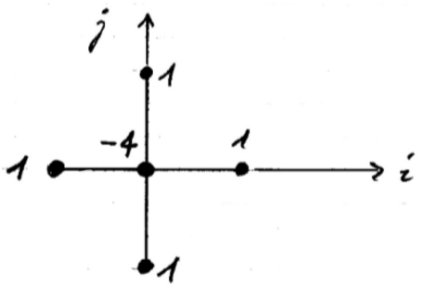


Figure 11.5: The stencil operator corresponding to Eq. 11.14.

At the boundaries, we directly use the given boundary condition $\varphi_{ij} = \varphi(x_i, y_j)$,

leading, e.g., for $i = 0$ to the discretized equation (see Fig. 11.6, left):

$$\frac{1}{h^2}[\tilde{u}_{2,j} + \tilde{u}_{1,j+1} + \tilde{u}_{1,j-1} - 4\tilde{u}_{1,j}] = f_{1,j} - \frac{1}{h^2}\varphi_{0,j},$$

with analogous expressions also at the other three boundaries. It is a convention that all known quantities are taken to the right-hand side of the discretized equation. For corners, e.g. $i = 0$, $j = N$ (see Fig. 11.6, right), we find:

$$\frac{1}{h^2}[\tilde{u}_{2,N-1} + \tilde{u}_{1,N-2} - 4\tilde{u}_{1,N-1}] = f_{1,N-1} - \frac{1}{h^2}[\varphi_{0,N-1} + \varphi_{1,N}],$$

again with analogous expressions also for the other three corners.

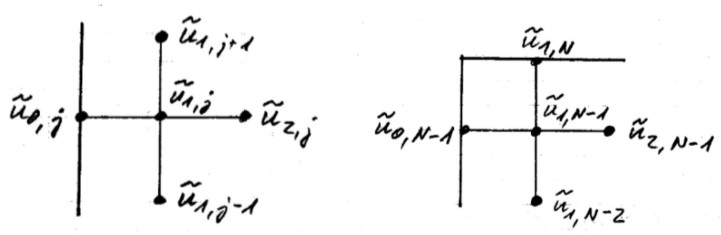


Figure 11.6: Examples of stencils at a boundary (left) and in a corner (right) of the domain.

Analyzing the order of convergence of the stencil

The order of convergence of the above stencil for the Laplace operator is 2. This can easily be seen from the Taylor expansion of u:

$$u(x \pm h, y) = (1 \pm hD_1 + \frac{h^2}{2}D_1^2 \pm \frac{h^3}{6}D_1^3 + \dots)u(x, y),$$

$$u(x, y \pm h) = (1 \pm hD_2 + \frac{h^2}{2}D_2^2 \pm \frac{h^3}{6}D_2^3 + \dots)u(x, y),$$

where we use the short-hand notation for the differential operators

$$D_1 = \frac{\partial}{\partial x},\ D_1^2 = \frac{\partial^2}{\partial x^2}, \dots \qquad D_2 = \frac{\partial}{\partial y},\ D_2^2 = \frac{\partial^2}{\partial y^2}, \dots$$

Substituting these Taylor expansions into the stencil, we find:

$$\frac{1}{h^2}[u(x_i + h, y_j) + u(x_i - h, y_j) + u(x_i, y_j + h) + u(x_i, y_j - h) - 4u(x_i, y_j)] =$$
$$\underbrace{(D_1^2 + D_2^2)u(x_i, y_j)}_{\Delta u(x_i, y_j)} + \frac{h^2}{12}(D_1^4 + D_2^4)u(x_i, y_j) + \frac{h^4}{60}(D_1^6 + D_2^6)u(x_i, y_j) + \dots$$

Because $\Delta u(x_i, y_j) = f_{i,j}$ due to the governing equation, we find that

$$\frac{1}{h^2}[u(x_i + h, y_j) + u(x_i - h, y_j) + u(x_i, y_j + h) + u(x_i, y_j - h) - 4u(x_i, y_j)] - f_{i,j} = \frac{h^2}{12}(D_1^4 + D_2^4)u(x_i, y_j) + \frac{h^4}{60}(D_1^6 + D_2^6)u(x_i, y_j) + \cdots \approx 0.$$

Therefore, the approximation error is $O(h^2)$ overall, as claimed. The recipe we followed here is generally applicable to convergence analysis of all stencil methods: substituting the appropriately shifted Taylor expansions of the exact solution into the stencil yields the discretization error.
Note that this procedure directly yields the global convergence order. Consider for example the explicit Euler method in space:

$$\frac{1}{h}(\tilde{x}_{j+1} - \tilde{x}_j) - f_j = 0.$$

Substituting the Taylor expansion of the exact solution yields:

$$\frac{1}{h}[(\cancel{x_j} + \cancel{h f_j} + \frac{h^2}{2}\ddot{x}(t_j) + \dots) - \cancel{x_j}] - \cancel{f_j} \in O(h),$$

which is the global error order of the Euler method.

Numerically approximating the 2D Poisson equation on the unit square using the above stencils results in a system of $(N-1)^2$ linear equations, one for each interior mesh node. This system of equations can be very large in practice. Already using only 1000 mesh cells in each direction leads to a system of 10^6 equations. Efficient linear system solvers (see Chapter 2) are therefore key to using stencil methods.

For illustration purposes, we explicitly construct the resulting system of linear equations for the small case $N = 4$ with $f \equiv -1$ and $\varphi \equiv 0$. This yields a Cartesian mesh of 5×5 nodes and a system of $(N-1)^2 = 9$ linear equations.

For simplicity of notation, we renumber the unknown interior points using linear indices as:

$$\begin{aligned} \tilde{u}_1 &= \tilde{u}_{1,1} & \tilde{u}_2 &= \tilde{u}_{2,1} & \tilde{u}_3 &= \tilde{u}_{3,1} \\ \tilde{u}_4 &= \tilde{u}_{1,2} & \tilde{u}_5 &= \tilde{u}_{2,2} & \tilde{u}_6 &= \tilde{u}_{3,2} \\ \tilde{u}_7 &= \tilde{u}_{1,3} & \tilde{u}_8 &= \tilde{u}_{2,3} & \tilde{u}_9 &= \tilde{u}_{3,3}. \end{aligned}$$

This numbering of the mesh nodes is illustrated in Fig. 11.7. Further, we multiply Eq. 11.14 with $h^2 = \frac{1}{16}$. This leads to the following system of equations for the unknown interior points:

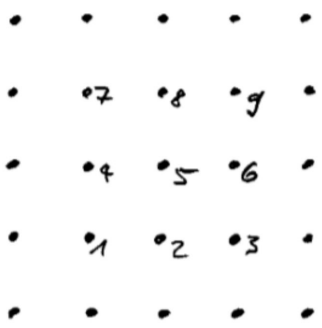


Figure 11.7: Linear indexing of the interior mesh nodes.

$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} \tilde{u}_1 \\ \tilde{u}_2 \\ \tilde{u}_3 \\ \tilde{u}_4 \\ \tilde{u}_5 \\ \tilde{u}_6 \\ \tilde{u}_7 \\ \tilde{u}_8 \\ \tilde{u}_9 \end{bmatrix} = \begin{bmatrix} -\frac{1}{16} \\ -\frac{1}{16} \\ -\frac{1}{16} \\ -\frac{1}{16} \\ -\frac{1}{16} \\ -\frac{1}{16} \\ -\frac{1}{16} \\ -\frac{1}{16} \\ -\frac{1}{16} \end{bmatrix}.$$

Because in this example $\varphi = 0$, there are no right-hand-side terms for the boundary nodes. For non-zero boundary conditions, the right-hand-side vector would contain additional contributions from the boundary conditions according to Fig. 11.6. As is typical of stencil methods, the matrix of the resulting system of linear equations is *sparse*, but does not have a simple, e.g., tri-diagonal structure, like it had for the method of lines. Since the matrix can also be very large, iterative linear system solvers are a good choice for stencil methods. In particular, there is a whole class of iterative linear system solvers that are designed to work well with sparse matrices. Famous examples of such sparse linear system solvers include *Krylov subspace methods*, the *Lanczos algorithm*, and *Multigrid Methods*. In fact, the conjugate gradient method discussed in Section 2.4 is a Krylov subspace method. For symmetric, positive definite matrices, it hence is a good choice to be used in stencil codes. For asymmetric or not positive definite matrices, GMRES is a famous Krylov subspace method often used in stencil methods. In any case, since the matrix is of size $(N-1)^2 \times (N-1)^2$, the computational cost is $O(N^2)$ per time step, because only a constant number of entries in each row are non-zero.

11.3 Hyperbolic Problems: The Method of Characteristics

As a model problem for hyperbolic PDEs, we consider the 1D wave equation from Example 11.3. A wave is characterized by a function moving at constant speed c, called the *phase velocity* of the wave. The lines/rays along which the wave propagates are called the *characteristics* of the wave equation. They are given by the equations:

$$\xi = x + ct, \;\; \eta = x - ct$$
$$\Longrightarrow x = \frac{\xi + \eta}{2}, \;\; t = \frac{\xi - \eta}{2c}.$$

The method of characteristics is based on writing the governing equation in these new coordinates, i.e., looking at how the function u changes along a characteristic. This is somewhat similar to the method of lines for parabolic PDEs, but the lines are particularly selected to be the characteristics. Every hyperbolic PDE possesses characteristics, not only the wave equation, rendering this a general numerical method for hyperbolic PDEs.
In the present example, substituting the above coordinate transform into the governing equation, we find:

$$\begin{aligned}
u(t,x) &= u\left(\frac{\xi-\eta}{2c}, \frac{\xi+\eta}{2}\right) =: \bar{u}(\xi,\eta) \\
u_x &= \bar{u}_\xi \frac{\partial \xi}{\partial x} + \bar{u}_\eta \frac{\partial \eta}{\partial x} = \bar{u}_\xi + \bar{u}_\eta \\
u_t &= \bar{u}_\xi \frac{\partial \xi}{\partial t} + \bar{u}_\eta \frac{\partial \eta}{\partial t} = c\bar{u}_\xi - c\bar{u}_\eta \\
u_{xx} &= (\bar{u}_\xi + \bar{u}_\eta)_\xi + (\bar{u}_\xi + \bar{u}_\eta)_\eta = \bar{u}_{\xi\xi} + 2\bar{u}_{\xi\eta} + \bar{u}_{\eta\eta} \\
u_{tt} &= c(c\bar{u}_\xi - c\bar{u}_\eta)_\xi - c(c\bar{u}_\xi - c\bar{u}_\eta)_\eta = c^2(\bar{u}_{\xi\xi} - 2\bar{u}_{\xi\eta} + \bar{u}_{\eta\eta}).
\end{aligned}$$

Therefore, Eq. 11.7 in the new coordinates reads:

$$4c^2\bar{u}_{\xi\eta} = 0 \qquad \Longrightarrow \qquad \bar{u}_{\xi\eta} = 0,$$

since $c^2 > 0$. Therefore, any function of the form

$$\bar{u}(\xi,\eta) = P(\xi) + Q(\eta)$$

is a solution to the equation, because its mixed derivative in (ξ,η) vanishes. In the original coordinates, this means that any function $u(t,x) = P(x+ct) + Q(x-ct)$ is a solution of the wave Eq. 11.7 for arbitrary $P(\cdot)$ and $Q(\cdot)$. Along the lines $x \pm ct = \text{const}$ in the (x,t)-plane, $P(x+ct)$ or $Q(x-ct)$ are constant, respectively. This means that P describes a function that travels left in space with constant speed c, whereas the function Q travels to the right with the same speed. This is illustrated in Fig. 11.8 for some example $Q(\cdot)$. As can be seen, the value of Q (and also of P in the opposite direction of travel) is constant along

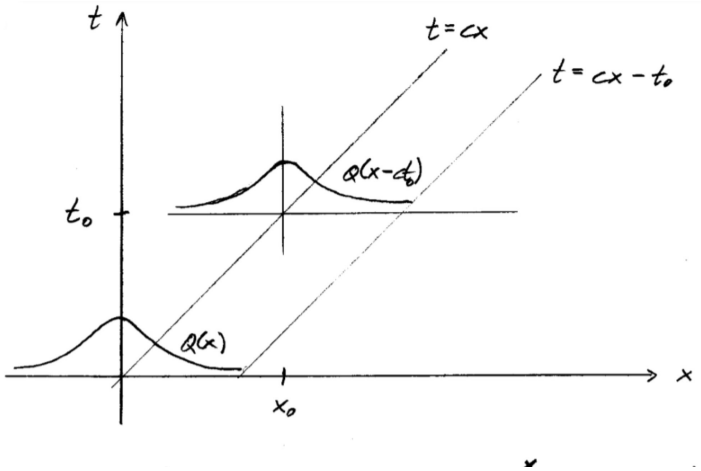


Figure 11.8: Illustration of a function $Q(x)$ traveling forward in time and right in space with constant speed c.

the characteristics $x - ct = \text{const}$. This is actually the general definition of *characteristics* as lines along with the solution of a hyperbolic PDE is constant. They are the key to solving hyperbolic PDEs.
The initial conditions of the problem in Example 11.3 are then used to determine P and Q. For $t = 0$, we have $\xi = x$ and $\eta = x$. Therefore:

$$\begin{aligned} u(0,x) &= \bar{u}(x,x) = P(x) + Q(x) \stackrel{!}{=} \varphi(x) \\ u_t(0,x) &= c[\bar{u}_\xi(x,x) - \bar{u}_\eta(x,x)] \\ &= c[P'(x) - Q'(x)] \stackrel{!}{=} \psi(x), \end{aligned}$$

where the prime means a derivative with respect to the only argument. Integrating both sides of the second equation $\int_0^x \mathrm{d}x$, we find:

$$P(x) - Q(x) = \frac{1}{c}\int_0^x \psi(s)\,\mathrm{d}s + \underbrace{P(0) - Q(0)}_{=:k}.$$

Together with the first equation, $P(x) + Q(x) \stackrel{!}{=} \varphi(x)$, we thus obtain two equations for the two unknowns P and Q. Solving this system of two equations for P and Q yields:

$$\begin{aligned} P(x) &= \frac{1}{2}\varphi(x) + \frac{1}{2c}\int_0^x \psi(s)\,\mathrm{d}s + \frac{k}{2} \\ Q(x) &= \frac{1}{2}\varphi(x) - \frac{1}{2c}\int_0^x \psi(s)\,\mathrm{d}s - \frac{k}{2}. \end{aligned}$$

Therefore, we find the final analytical solution of Eqs. 11.7–11.9 as:

$$u(t,x) = \frac{1}{2}[\varphi(x+ct) + \varphi(x-ct)] + \frac{1}{2c}\int_{x-ct}^{x+ct} \psi(s)\,\mathrm{d}s. \tag{11.15}$$

The above solution is exact and unique. The 1D wave equation therefore is an example of a PDE that can analytically be solved for arbitrary initial conditions, using the method of characteristics. However, the integral in Eq. 11.15 may not have a closed-form solution in general. There are therefore two ways to numerically approximate the solution of the wave equation: (1) Numerically evaluate the analytical solution in Eq. 11.15, using quadrature (see Chapter 8) to approximate the integral. If this is done along characteristics, i.e., along propagation rays of the wave, then the method is called *ray tracing*, which is frequently used in computer graphics to numerically approximate the *rendering equations*, or in radiative energy/heat transfer simulations. (2) Numerically approximate the derivatives in the original problem of Eq. 11.7.

An important observation we gain from the above exact solution is that only the initial conditions in a certain spatial interval influence the solution at a given point. To see this, consider Fig. 11.9. Through every point (x^*, t^*) in the solution space, there are exactly two characteristics: $x-ct = x_1$ and $x+ct = x_2$, which intersect the x-axis in two points, $x_1 = x^* - ct^*$ and $x_2 = x^* + ct^*$. The interval $[x_1, x_2]$ for $t = 0$ is called the *analytical domain of dependence* (or: *analytical region of influence*) for the solution at (x^*, t^*). Only the initial conditions $\varphi(x)$, $\psi(x)$ for $x \in [x_1, x_2]$ influence the solution at (x^*, t^*). This concept exists for every hyperbolic PDE.

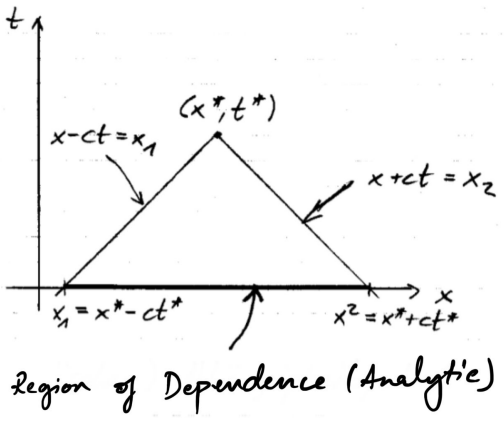


Figure 11.9: Illustration of the analytical domain of dependence of a hyperbolic PDE.

11.3.1 Numerical approximation

We do not discuss ray-tracing methods here, as they are an entirely different class of algorithms usually treated in computer graphics courses. Instead, we illustrate possibility (2) above in the example of the 1D wave equation. Using second-order central finite differences (see Chapter 9) to approximate the second derivatives in Eq. 11.7 in both space and time, we get:

$$\frac{1}{\bar{h}^2}(\tilde{u}_j^{k+1} - 2\tilde{u}_j^k + \tilde{u}_j^{k-1}) = \frac{c^2}{h^2}(\tilde{u}_{j+1}^k - 2\tilde{u}_j^k + \tilde{u}_{j-1}^k),$$

where subscripts denote space points $x_j = jh$ with resolution $\Delta x = h$ and superscripts denote time steps $t_k = k\bar{h}$ of size $\Delta t = \bar{h}$. The corresponding stencil is visualized in Fig. 11.10.

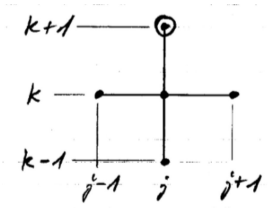


Figure 11.10: Visualization of the finite-difference stencil for the wave equation.

This stencil allows computing the circled value at time t_{k+1} from the known values at two previous times t_k and t_{k-1} with a local discretization error $O(\bar{h}^2)+O(h^2)$. Defining the constant

$$\lambda := \frac{c\bar{h}}{h}$$

and solving for the value at the new time point, we find:

$$\tilde{u}_j^{k+1} = 2(1-\lambda^2)\tilde{u}_j^k + \lambda^2(\tilde{u}_{j+1}^k + \tilde{u}_{j-1}^k) - \tilde{u}_j^{k-1}, \quad k = 1, 2, 3, \ldots \tag{11.16}$$

In order to start the algorithm, we need to know $\tilde{u}_j^0$ and $\tilde{u}_j^1$ for all j. The former is trivially obtained from the initial condition:

$$\tilde{u}_j^0 = \varphi(x_j), \;\; j \in \mathbb{Z}.$$

The latter can be approximated to second order from the Taylor series

$$\tilde{u}_j^1 = \tilde{u}(\bar{h}, x_j) = \underbrace{\tilde{u}(0, x_j)}_{\tilde{u}_j^0 = \varphi(x_j)} + \underbrace{\tilde{u}_t(0, x_j)}_{\psi(x_j)}\bar{h} + \tilde{u}_{tt}(0, x_j)\frac{\bar{h}^2}{2} + \ldots$$

as:

$$\tilde{u}_j^1 = \tilde{u}_j^0 + \bar{h}\psi(x_j), \;\; j \in \mathbb{Z}.$$

The algorithm then proceeds row-wise, building up the solution space by iterating over all x for each t.
For finite domains, e.g., $0 \leq x \leq L$, boundary conditions for $u(t,0)$ and $u(t,L)$ are required in addition. Frequently used boundary conditions for waves are *absorbing boundaries*, where $u_{\text{boundary}} = 0$, and *reflective boundaries*, where $\frac{\partial u}{\partial x}\big|_{\text{boundary}} = 0$. In the former case, the wave is absorbed at the boundary, in the latter case it is reflected with a phase shift of π. In a bounded domain, we can introduce the finite vector

$$\tilde{\vec{u}}_k = (\tilde{u}_1^k, \ldots, \tilde{u}_{N-1}^k)^\mathsf{T} \in \mathbb{R}^{N-1}.$$

The iteration in Eq. 11.16 can then be written as:

$$\tilde{\vec{u}}_{k+1} = 2\tilde{\vec{u}}_k - \lambda^2 \mathbf{A}\tilde{\vec{u}}_k - \tilde{\vec{u}}_{k-1}, \quad k = 1, 2, 3, \ldots \tag{11.17}$$

with the tri-diagonal matrix

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 & \ldots & \\ -1 & 2 & -1 & 0 & \ldots & \\ 0 & -1 & 2 & -1 & \ldots & \\ \vdots & \vdots & \vdots & \ddots & \ddots & \\ & & & & -1 & 2 \end{bmatrix} \in \mathbb{R}^{(N-1)\times(N-1)}$$

and initial conditions

$$\tilde{\vec{u}}_0 = (\varphi(x_1), \ldots, \varphi(x_{N-1}))^\mathsf{T}$$

$$\tilde{\vec{u}}_1 + \tilde{\vec{u}}_0 + \bar{h}(\psi(x_1), \ldots, \psi(x_{N-1}))^\mathsf{T}.$$

Evolving this iteration in time requires a matrix-vector multiplication in each time step and therefore has a computational cost of $O(N^2)$ per time step.

11.3.2 Numerical domain of dependence

Using the above scheme to numerically approximate the solution at point (x_j^*, t_k^*) requires three points from the row below: (x_{j-1}^*, t_{k-1}^*), (x_j^*, t_{k-1}^*), (x_{j+1}^*, t_{k-1}^*). Computing these three points in turn requires five points at t_{k-2}^*, and so on. In the end, all mesh nodes in a pyramid shown in Fig. 11.11 are required in order to compute the numerical approximation to the solution at (x_j^*, t_k^*). At $t = 0$, it includes the initial conditions for $x \in \left[x_j^* - (h/\bar{h})t_k^*,\, x_j^* + (h/\bar{h})t_k^*\right]$. This interval is called the *numerical domain of dependence* (or: *numerical region of influence*) of the method.
In order for the method to be consistent, **the numerical domain of dependence must include (i.e., be a superset of) the analytical domain of dependence**. This means that the characteristics through (x_j^*, t_k^*) must be entirely contained in the mesh pyramid used to compute the numerical approximation at (x_j^*, t_k^*). This is a necessary condition for a numerical solution of a hyperbolic PDE to be consistent.

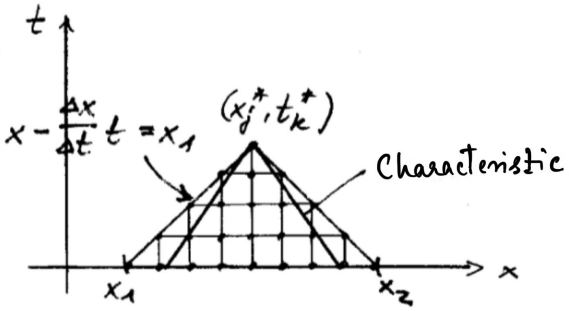


Figure 11.11: Illustration of the numerical domain of dependence of a hyperbolic PDE.

Image: geni.com
Richard Courant
∗ 8 January 1888
Lublinitz, German Empire
(now: Lubliniec, Poland)
† 27 January 1972
New Rochelle, NY, USA

Image: wikipidia
Kurt Otto Friedrichs
∗ 28 September 1901
Kiel, German Empire
† 31 December 1982
New Rochelle, NY, USA

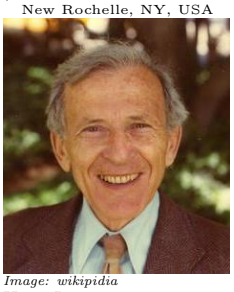


Image: wikipidia
Hans Lewy
∗ 20 October 1904
Breslau, German Empire
(now: Wrocław, Poland)
† 23 August 1988
Berkeley, CA, USA

In order to see this, assume that this condition is not fulfilled. Then, there exists a part of the initial condition that is within the analytical domain of dependence, but outside the numerical domain of dependence. Changing the initial condition in that region changes the analytical solution at (x_j^*, t_k^*), but not the numerical solution. The numerical method is then not able to ever converge to the correct solution, even if it is stable, which implies (by the Lax Equivalence Theorem) that it is inconsistent.
In the present example of the 1D wave equation, the condition that the numerical domain of dependence must include the analytical domain of dependence is not only necessary, but also sufficient for convergence. In general, for arbitrary hyperbolic PDEs, it is only necessary, and stability may not be implied (cf. the Lax Equivalence Theorem in Section 10.9).

11.3.3 Courant-Friedrichs-Lewy condition

The above condition that the numerical domain of dependence must include the analytical domain of dependence can be formally stated as:

$$\frac{\bar{h}}{h} \leq \frac{1}{c},$$

which means that the slope of the characteristics must be larger than the slope of the boundaries of the numerical domain of dependence. Using the above-defined constant

$$\lambda := \frac{c\bar{h}}{h},$$

this implies:

$$\lambda \leq 1.$$

The constant λ is called the *Courant number*, and the condition $\lambda \leq 1$ is known as the *Courant-Friedrichs-Lewy condition* (CFL condition) of this problem, named after Richard Courant, Kurt Friedrichs, and Hans Lewy, who described it for the first time in 1928. Every hyperbolic PDE has a CFL condition

of this sort, which must be fulfilled in order for a numerical method to be converge.
While we naturally found the CFL condition here in the example of the 1D wave equation, it also exist for other PDEs with hyperbolic parts, e.g., in fluid mechanics whenever there is an advection term with a certain velocity c.
In d-dimensional spaces, the Courant numbers add, hence:

$$\lambda = \bar{h} \left(\sum_{i=1}^{d} \frac{c_i}{h_i} \right),$$

where c_i is the velocity component in coordinate direction i, and h_i the spatial grid resolution in coordinate direction i.
In general, the CFL condition reads:

$$\lambda \leq C$$

for some constant C that depends on the numerical method used and on the PDE considered. For the central 2nd-order finite differences used in the above example, $C = 1$. More robust numerical methods can have $C > 1$, hence allowing larger time steps. For reasons of computational efficiency, one can tune the time and space resolution of a simulation to get as close as possible to the CFL limit. This is the most efficient simulation possible without becoming inconsistent.

Bibliography