Numerische Mathematik (i282)

Jens Flemming (HTW Dresden)

19.06.2026

Inhaltsverzeichnis

0 Vorwort

Dieses Vorlesungsskript ist die Arbeitsgrundlage für die Veranstaltung “Numerische Mathematik” (Modul i282) an der HTW Dresden ab Sommersemester 2026. Hinweise zum Ablauf der Veranstaltung, insbesondere zum Umgang mit diesem Skript, finden Sie unter Organisatorisches.

Technischer Unterbau ist MyST Markdown mit einem eigenen, auf dem Standard-Book-Theme aufbauenden Theme. Das MyST-Projekt ist noch recht jung und etliche Features sind im Beta-Stadium. Hier und da ist also mit Ecken und Kanten zu rechnen. Fragen und Anregungen zur Technik gern an Jens Flemming.

1 Was? Warum?

Die numerische Mathematik („Numerik“) befasst sich mit **Entwicklung und Analyse von Algorithmen** für das **zahlenmäßige Lösen** von in der Sprache der Mathematik formulierten Aufgabenstellungen.

Früher: Algorithmen per Hand/Zettel/Stift ausführen (oder in Lehm geritzt).

Heute: Computer.

1.1 Analytisches vs. numerisches Lösen

analytisches Lösen	numerisches Lösen
exakte Ausdrücke	Kommazahlen
exakte Zahlen	endlich viele Ziffern (Nachkommastellen!)
Grenzübergänge	endlich viele Rechenschritte
komplexe Operation	Grundrechenarten

Nicht Thema hier: **Computeralgebrasysteme (CAS)**. Diese können automatisiert mit exakten Ausdrücken rechnen, teils auch Grenzübergänge ausführen usw. Jedoch liefern sie nur Lösungen, wenn die Aufgabenstellung überhaupt analytisch bestimmbare Lösungen besitzt, exakte Eingangsdaten vorliegen und die Aufgabe nicht zu kompliziert ist! Rechenzeiten sind deutlich höher als beim numerischen Lösen.

1.2 Themen in der Numerik

Besondere Aufmerksamkeit gilt

- den technischen Gegebenheiten, z.B.
  - Genauigkeit der Zahlendarstellung,
  - verfügbarer Speicherplatz,
  - Möglichkeit der parallelen Ausführung,
  - verfügbare Rechenzeit,
- dem Unterschied zwischen exakter Lösung und algorithmisch berechneter Lösung,
- der Auswirkung von Datenfehlern, insbesondere des Rundens von Eingangswerten und Zwischenergebnissen, auf das Endergebnis,
- der Anzahl der benötigten Rechenschritte.

Typische Aufgabenstellungen sind zum Beispiel:

- lineare Gleichungssysteme lösen,
- nichtlineare Gleichungen Lösen,
- Funktionsauswertung (Interpolation, Ausgleichsprobleme),
- Integrale berechnen,
- Eigenwerte und -vektoren von Matrizen berechnen,
- Differentialgleichungen lösen
- Optimierungsprobleme lösen

Je nach Komplexität der konkreten Aufgabenstellung führen insbesondere die letzen beiden Punkte zum **wissenschaftlichen Rechnen**. Ein klare Abgrenzung zur Numerik ist nicht möglich. Die Numerik liefert die Werkzeuge. Das wissenschaftliche Rechnen kombiniert sie zur Lösung komplexer Aufgaben aus der Praxis.

Merke!

Grundkenntnisse der numerischen Mathematik sind essentiell für solide Software-Entwicklung!

1.3 Beispiele

Was kann da schon schiefgehen?

Groß plus Klein

Klar:

$$1 + x^{20} - x^{20} = 1,$$

z.B. $x = 10.5$.

Taschenrechner, Python, C, JavaScript,...: 0.

```
print(1 + 10.5 ** 20 - 10.5 ** 20)

# Ausgabe: "0.0"
```

Abbildung 1.1: *
Python-Code zum Testen

Bei $1 + (x^{20} - x^{20})$ liefern alle 1.

Wir sehen hier **Absorption** und **fehlende Assoziativität** der Computeraddition.

```c
#include <stdio.h>
#include <math.h>

int main() {
    printf("%f\n", 1 + pow(10.5, 20) - pow(10.5, 20));
    return 0;
}

// Ausgabe: "0.000000"
```

Abbildung 1.2: *
C-Code zum Testen

```javascript
console.log(1 + 10.5 ** 20 - 10.5 ** 20);

# Ausgabe: "0"
```

Abbildung 1.3: *
JavaScript-Code zum Testen

Integral

Wollen für großes $n \in \mathbb{N}$ (z.B. $n = 20$) das Integral

$$I_n := \int_0^1 x^n \, \mathrm{e}^x \, \mathrm{d}x$$

berechnen. Das geht z.B. mittels partieller Integration, welche auf die Rekursionsformel

$$I_n = \mathrm{e} - n \, I_{n-1}, \quad I_0 = \mathrm{e} - 1,$$

führt (IDVID 140). Alternativ: numerische Integration, welche später im Semester behandelt wird.

```python
from math import exp
from sympy import N, E

n_max = 20

num_e = exp(1)
cas_e = E
```

```
num_i = num_e - 1
cas_i = cas_e - 1
for n in range(0, n_max + 1):
    if n > 0:
        num_i = num_e - n * num_i
        cas_i = cas_e - n * cas_i
    print(f'n = {n}, exakt: {cas_i}')
    print(f'  {num_i:.20f} (numerisch)')
    print(f'  {N(cas_i, 20):.20f} (CAS)')
```

```
n = 0, exakt: -1 + E
  1.71828182845904509080 (numerisch)
  1.71828182845904523540 (CAS)
n = 1, exakt: 1
  1.00000000000000000000 (numerisch)
  1.00000000000000000000 (CAS)
n = 2, exakt: -2 + E
  0.71828182845904509080 (numerisch)
  0.71828182845904523536 (CAS)
n = 3, exakt: 6 - 2*E
  0.56343634308190981841 (numerisch)
  0.56343634308190952928 (CAS)
n = 4, exakt: -24 + 9*E
  0.46453645613140581716 (numerisch)
  0.46453645613140711824 (CAS)
n = 5, exakt: 120 - 44*E
  0.39559954780201600499 (numerisch)
  0.39559954780200964415 (CAS)
n = 6, exakt: -720 + 265*E
  0.34468454164694906083 (numerisch)
  0.34468454164698737048 (CAS)
n = 7, exakt: 5040 - 1854*E
  0.30549003693040166496 (numerisch)
  0.30549003693013364203 (CAS)
n = 8, exakt: -40320 + 14833*E
  0.27436153301583177111 (numerisch)
  0.27436153301797609914 (CAS)
n = 9, exakt: 362880 - 133496*E
  0.24902803131655915081 (numerisch)
  0.24902803129726034306 (CAS)
n = 10, exakt: -3628800 + 1334961*E
  0.22800151529345358270 (numerisch)
  0.22800151548644180472 (CAS)
n = 11, exakt: 39916800 - 14684570*E
```

```
  0.21026516023105568109 (numerisch)
  0.2102651510818538341 (CAS)
n = 12, exakt: -479001600 + 176214841*E
  0.19509990568637691766 (numerisch)
  0.1950999311608206344 8 (CAS)
n = 13, exakt: 6227020800 - 2290792932*E
  0.18198305453614516125 (numerisch)
  0.18198272336837698714 (CAS)
n = 14, exakt: -87178291200 + 32071101049*E
  0.17051906495301283329 (numerisch)
  0.17052370130176741544 (CAS)
n = 15, exakt: 1307674368000 - 481066515734*E
  0.16049585416385259151 (numerisch)
  0.16042630893253400376 (CAS)
n = 16, exakt: -20922789888000 + 7697064251745*E
  0.15034816183740362661 (numerisch)
  0.15146088553850117518 (CAS)
n = 17, exakt: 355687428096000 - 130850092279664*E
  0.16236307722318343849 (numerisch)
  0.14344677430452525731 (CAS)
n = 18, exakt: -6402373705728000 + 2355301661033953*E
  -0.20425356155825680204 (numerisch)
  0.13623989097759060374 (CAS)
n = 19, exakt: 121645100408832000 - 44750731559645106*E
  6.59909949806592432964 (numerisch)
  0.12972389988482376433 (CAS)
n = 20, exakt: -2432902008176640000 + 895014631192902121*E
  -129.26370813285942062976 (numerisch)
  0.12380383076256994869 (CAS)
```

Sehen hier ein **instabiles Verfahren** zur Berechnung des Integrals.

Pentium-FDIV-Bug

Siehe Pentium FDIV bug.

Kurzfassung:

1. Ein Fehler im Produktionsprozess von Pentium-Prozessoren führt zu nur selten auftretenden und **sehr kleinen Rechenfehlern** bei gewissen Grundrechenoperationen.
2. Ein iteratives Verfahren zur Division von Zahlen **verstärkt den Fehler** soweit, dass er relevant wird (fünfte Kommastelle falsch).
3. Intel merkt es, sagt aber nichts.

4. Jemand anderes merkt es und macht das Problem öffentlich.
5. PR-Disaster, Rückrufaktion, einige 100 Million Dollar Unkosten für Intel.

Patriot-Raketenabwehr

Siehe GAO/IMTEC-92-26 Patriot Missile Software Problem.

Kurzfassung:

1. Zweiter Golfkrieg ("Operation Desert Storm"). US-Amerikanische Militärbasis in Dharan (Saudi-Arabien) wird durch Patriot-Raketenabwehrsystem vor Scud-Raketen aus dem Irak geschützt. Vorher für dieses Szenario nicht wirklich getestet (70er-Jahre Legacy-Code).
2. Gelegentlich werden Raketen aus ungeklärten Gründen nicht abgefangen.
3. Jemand schaut mal nach, wo es klemmt.
4. Fehler gefunden: **Rundungsfehler verstärkt sich** im Laufe der Zeit durch ungeschickte Implementierung einer Multiplikation mit 0.1, sodass nach hinreichend langer Up-Time das System nicht mehr präzise genug arbeitet.
5. Anweisung an die Nutzer: täglich rebooten.
6. Software-Update wird am 16. Februar 1991 verfügbar, impliziert aber 2 Stunden Down-Time. Probleme beim Verteilen usw.
7. Am 25. Februar 1991 sterben 28 US-Soldaten durch eine Scud-Rakete.
8. Jemand spielt das Update ein am 26. Februar 1991.

Details zum Rundungsfehler: IDVID 170.

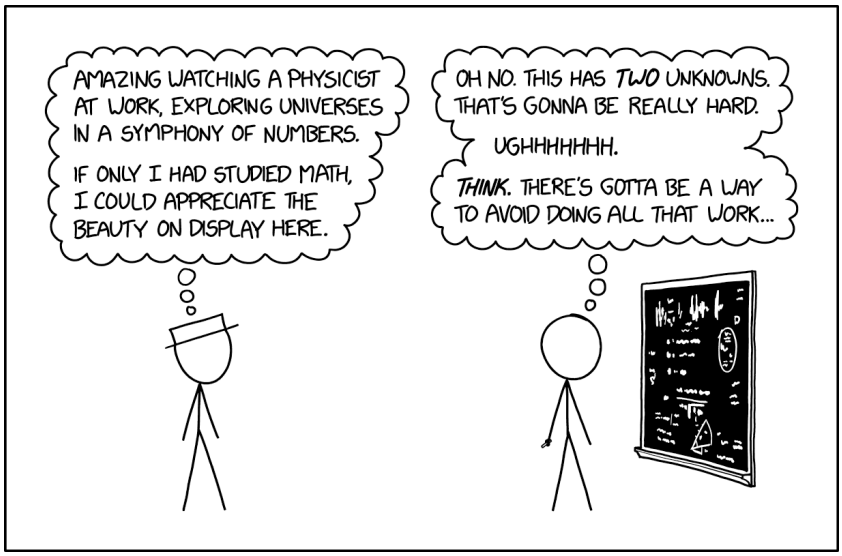


Abbildung 1.4: *
I could type this into a solver, which MIGHT help, but would also mean I have to get a lot of parentheses right... Quelle: Randall Munroe, xkcd.com/2207

2 Zahlen

Numerische Berechnung verwenden praktisch immer gebrochene Zahlen ("Kommazahlen"). Zwei Varianten für die Darstellung von gebrochenen Zahlen durch Bitfolgen:

- **Festkommazahlen**
  - begrenzte Anzahl Ziffern vor dem Komma,
  - begrenzte Anzahl Ziffern hinter dem Komma,
- **Gleitkommazahlen**
  - begrenzte Gesamtanzahl Ziffern,
  - flexible Position des Kommas (auch weit vor oder hinter der Ziffernfolge).

Festkommazahlen spielen heute nur in Spezialanwendungen eine Rolle, siehe z.B. GNU-Cash und STM32G4 CORDIC co-processor. Auch werden Festkommazahlen von gängigen Programmiersprachen nur in Ausnahmefällen unterstützt (ein Beispiel ist GNU C). Beschäftigen uns hier deshalb nur mit Gleitkommazahlen genauer.

2.1 Darstellung von Gleitkommazahlen

Merke!

Gleitkommazahlen (engl.: *floating-point numbers*, kurz: *floats*) können durch zwei Ganzzahlen beschrieben werden:

- **Mantisse** $m \in \mathbb{Z}$,
- **Exponent** $e \in \mathbb{Z}$.

Zusammen mit der gewählten Basis $b \in \{2, 10\}$ (Binär- oder Dezimalsystem) ergeben diese die Zahl

$$m\, b^e, \tag{2.1}$$

wobei e die Position des Kommas angibt. Bei $e = 0$ steht das Komma direkt hinter der Ziffernfolge. Positive e ergeben größere Zahlen (Komma weiter rechts), negative e kleinere (Komma weiter links).

Ist die Höchstanzahl der Mantissenziffern (ohne Vorzeichen) festgelegt auf $t \in \mathbb{N}$, so gibt es mehrere Paare (m, e), die die selbe Zahl darstellen. Beispiel für $b = 10$, $t = 4$:

$$123 \cdot 10^1 = 1230 \cdot 10^0. \tag{2.2}$$

Man wählt üblicherweise die Darstellung mit der längsten möglichen Mantisse, d.h. die Mantisse hat stets genau t Ziffern und die erste Ziffer ist nicht Null. Eine so dargestellte Gleitkommazahl heißt **normalisierte Gleitkommazahl**. Diese Darstellung ist

eindeutig, für Null aber nicht möglich. Wenn nicht anders angegeben, meinen wir mit "Gleitkommazahl" im Folgenden stets eine normalisierte Gleitkommazahl.

Die Ziffernanzahl im Exponent spielt in der numerischen Mathematik nur selten eine Rolle. Die Mantissenlänge ist hingegen ausschlaggebend für die Genauigkeit von Berechnungen. Wenn wir im Folgenden Aussagen der Form "für alle reellen Zahlen gilt..." formulieren, meinen wir damit stets nur solche reellen Zahlen, die betragsmäßig kleiner als die größte darstellbare Gleitkommazahl sind, also keinen **Exponentenüberlauf** verursachen.

2.2 Datenformat für binäre Gleitkommazahlen

Für die Umwandlung von Mantisse und Exponent einer binären Gleitkommazahl zu einer Bitfolge spielt heute hauptsächlich der Standard IEEE 754 eine Rolle. Siehe Floating-point arithmetic für Ausnahmen.

Benötigte Bitfolgen für eine normalisierte Gleitkommazahl:

- Vorzeichen S der Mantisse:
  - 1 Bit,
  - 0 oder 1 für positive bzw. negative Mantisse.
- Betrag M der Mantisse ohne erste Ziffer:
  - $t-1$ Bit,
  - die erste Ziffer ist immer 1, muss also nicht gespeichert werden ("hidden bit").
- transformierter Exponent E:
  - p Bit,
  - $E = e + t - 1 + 2^{p-1} - 1$,
  - $E = 0$ und $E = 2^p - 1$ markieren spezielle Gleitkommazahlen (siehe unten).

Der Zusammenhang zwischen E und e entsteht dabei aus der Forderung, dass links und rechts der zweiten Mantissenziffer gleich viele Kommapositionen darstellbar sein sollen:

- Der Summand $t-1$ verschiebt das Komma vom Ende der Mantisse zwischen die ersten beiden Ziffern.
- Der Summand $2^{p-1} - 1$ sorgt für die gleiche Anzahl Kommapositionen links und rechts.

Die durch Vorzeichenbit S, Mantissenbetrag M (ohne erste Ziffer) und transformierten Exponent E beschriebene Zahl ist also

$$(-1)^S \left(2^{t-1} + M\right) \cdot 2^{E-(2^{p-1}-1)-(t-1)} \tag{2.3}$$

Tabelle 1: *
Übliche Werte für die Bitanzahl von Mantisse und Exponent bei binären Gleitkommazahlen

Bezeichnung	$t-1$	p	Wertebereich von $e+t-1$	dezimale Ziffern von m
16-Bit-Geleitkommazahl	10	5	$-14\ldots15$	$3\ldots4$
32-Bit-Geleitkommazahl	23	8	$-126\ldots127$	$7\ldots8$
64-Bit-Geleitkommazahl	52	11	$-1022\ldots1023$	$15\ldots16$

oder auch

$$(-1)^S \cdot 1.M \cdot 2^{E-(2^{p-1}-1)}, \tag{2.4}$$

wobei $1.M$ als $1 + 2^{1-t}\,M$ zu verstehen ist.

Spezielle Werte:

- $E = 0$ markiert die Zahl 0, wenn $M = 0$; sonst sogenannte denormalisierte Zahlen (behandeln wir hier nicht).
- $E = 2^p - 1$ markiert ∞, wenn $M = 0$; sonst "NaN", also ungültige Zahl.

Ungültige Zahlen entstehen z.B. als Ergebnis der Division $0/0$. Unendliche Werte entstehen zum z.B. bei $1/0$ (dann $+\infty$) oder -1/0 (dann $-\infty$). Die Zahl 0 hat dabei ebenfalls ein Vorzeichen. Betragskleine negative Zahlen werden auf -0 gerundet, positive auf +0. Somit kann $1/0$ auch $-\infty$ ergeben, wenn die 0 durch Runden einer negativen Zahl entstanden ist.

Nebenbemerkung

In der Literatur wird der Begriff Gleitkommazahl nicht einheitlich verwendet. Gelegentlich sind damit alle in der Form $m\,b^e$ darstellbaren Zahlen gemeint und die mit dem Computer darstellbaren Zahlen (endlicher Bereich für m und e!) werden Maschinenzahlen genannt. Wir verwenden den Begriff Gleitkommazahl hier ausschließlich für mit dem Computer darstellbare Zahlen bei vorgegebener Ziffernanzahl und vorgegebenem Bereich von Kommapositionen.

2.3 Runden

Das übliche Rundungsverfahren für Gleitkommazahlen entspricht nicht exakt dem mathematischen Runden.

Merke!

Regeln für das Runden von Gleitkommazahlen:

- Runde zur nächst gelegenen darstellbaren Zahl.
- Wenn die zu rundende Zahl genau mittig zwischen zwei darstellbaren Zahlen liegt, runde so, dass die letzte Ziffer der gerundeten Zahl gerade ist.

Beim mathematischen Runden wird in der zweiten Regel immer die größere der beiden möglichen Zahlen gewählt. Dies führt jedoch zur **statistischen Drift**.

Beispiel: Rechnen mit 2 Kommastellen. Mittelwert der 10 Werte 0.105, 0.115, 0.125, 0.135, 0.145, 0.155, 0.165, 0.175, 0.185, 0.195 ist 0.15 bzw. 0.16 je nach verwendeter Rundungsregel (IDVID 230).

Merke!

Sei $x \in \mathbb{R}$ eine reelle Zahl, sei $m\,b^e$ die betragsmäßig nächst kleinere (normalisierte) Gleitkommazahl und bezeichne $\mathrm{round}(x)$ die durch Runden aus x erhaltene (normalisierte) Gleitkommazahl. Dann gilt für den **absoluten Rundungsfehler**

$$|x - \mathrm{round}(x)| \leq \tfrac{1}{2}\,|m\,b^e - (m+1)\,b^e| = \tfrac{1}{2}\,b^e \tag{2.5}$$

und für den **relativen Rundungsfehler**

$$\frac{|x - \mathrm{round}(x)|}{|x|} \leq \frac{b^e}{2\,|x|} \leq \frac{b^e}{2\,|m|\,b^e} = \frac{1}{2\,|m|} \leq \frac{1}{2}\,b^{1-t}, \tag{2.6}$$

wobei t die Mantissenlänge ist und die letzte Ungleichheit auf der Tatsache beruht, dass m genau t Ziffern hat und die erste Ziffer nicht Null ist.

Die Schranke $\varepsilon := \frac{1}{2}\,b^{1-t}$ für den relativen Rundungsfehler heißt **Maschinenepsilon**. Aus der Dreiecksungleichung folgen

$$(1-\varepsilon)\,|x| \leq |\mathrm{round}(x)| \leq (1+\varepsilon)\,|x| \tag{2.7}$$

und

$$\frac{1}{1+\varepsilon}\,|\mathrm{round}(x)| \leq |x| \leq \frac{1}{1-\varepsilon}\,|\mathrm{round}(x)| \tag{2.8}$$

(IDVID 260). Diese Ungleichungen spielen eine wichtige Rolle bei der Fehleranalyse von Algorithmen. Insbesondere existiert zu jedem reellen x ein ϱ mit

$$\mathrm{round}(x) = (1+\varrho)\,x, \qquad |\varrho| \leq \varepsilon. \tag{2.9}$$

2.4 Grundoperationen

Einige grundlegende Operationen, die nach IEEE 754 mit korrektem Runden implementiert sein müssen (meist direkt in der Hardware):

- Addition, Subtraktion,
- Multiplikation, Division,
- Quadratwurzel,
- Umwandlung in Ganzzahl,
- Umwandlung zwischen den verschiedenen Bitanzahlen.

Praktisch alle Prozessoren bieten viele weitere Operationen an. Siehe z.B. Intel®64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture (ab Abschnitt 8.3.7). Mathematische Bibliotheken liegen üblicherweise in an die verschiedenen Prozessoren angepassten Versionen vor, sodass möglichst viele Operationen direkt auf der Hardware ausgeführt werden können.

Als Maß für die Rechengeschwindigkeit eines Prozessors werden gern **FLOPS** (floating-point operations per second, auch: flop/s) angegeben. Eine "floating-point operation" ist dabei meist als kombinierte Addition und Multiplikation zu verstehen. Es gibt jedoch keine allgemein anerkannte feste Definition dieser Einheit! Die Umrechnungen von FLOPS in Prozessortakte ist prozessorabhängig; siehe Floating point operations per second für die Umrechnungsfaktoren bei üblichen Prozessoren. Die Prozessortaktanzahlen für verschiedene Operationen findet man zum Beispiel in 4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs (Spalte "Latency", z.B. Seite 369 für Intel-Ice-Lake-Prozessoren).

2.5 Hinweise für die Praxis

Der IEEE-754-Standard überlässt einige Details der konkreten Implementierung, sodass Programmcode sich auf unterschiedlichen Prozessorarchitekturen und bezüglich unterschiedlicher Compiler und Bibliotheken eventuell unterschiedlich verhält, siehe z.B. Floating-Point Determinism.

Auch setzen die meisten Compiler Codeoptimierung ein, wobei zu Gunsten von Geschwindigkeit auf Genauigkeit verzichtet wird. Die Optimierung lässt sich über zahlreiche Optionen steuern/abschalten. Für GNU C siehe Options That Control Optimization, für Microsoft C++ siehe `/fp` (Specify floating-point behavior).

Nicht nur mit Blick auf Abweichungen zwischen erwarteter und tatsächlich erzielter Genauigkeit sind diese Aspekte interessant, sondern auch mit Blick auf Reproducible Builds.

Für weitere "Randthemen" und Probleme bei der Verwendung von Gleitkommazahlen siehe auch Random ASCII – tech blog of Bruce Dawson (Category Archives: Floating

Point).

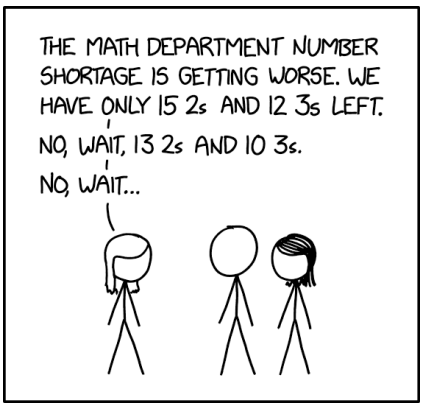


Abbildung 2.1: *
"10 minutes ago we were down to only 2 0s!" "How many do we have now?" "I ... don't know!!" Quelle: Randall Munroe, xkcd.com/3009

3 Fehler

3.1 Fehlerarten

Bei numerischen Berechnungen spielen verschiedene Fehlerarten eine Rolle:

- **Rundungsfehler** (Zwischenergebnisse werden auf die darstellbare Genauigkeit gerundet).
- **Datenfehler** (die Eingabewerte entstehen aus (ungenauen) Messungen),
- **Verfahrensfehler** (die eingesetzten Algorithmen liefern auch bei exakter Rechnung ohne Rundungsfehler nur Näherungslösungen).
- **Modellfehler** (vereinfachte/idealisierte Darstellung der tatsächlichen physikalischen oder anderweitigen Gegebenheiten),

Dabei enthält der Datenfehler praktisch immer auch Rundungsfehler, da die Daten als Gleitkommazahlen an den Algorithmus übergeben werden müssen.

Obwohl die numerische Mathematik sich auf die (Vermeidung der) Auswirkungen von Rundungsfehlern konzentriert, spielen auch die anderen Fehlerarten und -quellen eine Rolle. Insbesondere **zielt man gar nicht auf maximale Genauigkeit** (was mit hohem Rechenaufwand verbunden ist), sondern nur darauf, dass die Auswirkungen von Verfahrens- und Rundungsfehlern nicht größer sind als zwangsläufige Abweichungen aus Daten- und Modellfehlern.

3.2 Beispiel: Volumen der Erde

Wollen das Volumen der Erde aus gegebenem Radius r (in Meter) mittels Kugelvolumen

$$V = \frac{4}{3}\,\pi\, r^3 \tag{3.1}$$

berechnen.

Algorithmus

1. Teile 4 durch 3.
2. Multipliziere mit π.
3. Multipliziere dreimal mit r.

Auftretende Fehler:

- Modellfehler: Die Erde ist keine exakte Kugel, sondern ein Ellipsoid (auch nicht exakt).

- Verfahrensfehler: Hier nicht vorhanden, sofern (bei exakten Daten) mit exaktem π gerechnet wird.
- Datenfehler: Exakte Bestimmung von r scheitert schon an der Struktur der Erdoberfläche (Berge, Täler). Selbst mit sehr präziser Messtechnik wird man nur ein Intervall $r \in [a, b]$ angeben können. Wählt man $r = \frac{a+b}{2}$, beträgt der Datenfehler $\frac{b-a}{2}$.
- Rundungsfehler: r und π müssen für die Darstellung mittels Gleitkommazahlen gerundet werden. Die Zahlen 4 und 3 sind exakt darstellbar. Die Auswirkungen der Multiplikationen und der Division auf die Genauigkeit des Ergebnisses werden weiter unten untersucht.

Aufgrund der unvermeidlichen großen Modell- und Datenfehler gibt es keinen Grund die Berechnungen mit sehr großer Genauigkeit auszuführen. Solange die Rundungsfehler und deren Auswirkungen auf die einzelnen Rechenoperationen kleiner als die Modell- und Datenfehler bleiben, ist die Genauigkeit groß genug.

3.3 Bewertung von Algorithmen

Gegeben ist ein Algorithmus, der "ein Problem löst". Genauer: Es gibt eine Menge $X \subseteq \mathbb{R}^d$ von möglichen Eingaben, eine Menge $Y \subseteq \mathbb{R}$ von möglichen Ausgaben und eine (exakte) Lösungsabbildung $f : X \to Y$.

Zentrale Frage: **Wie hängt der Ausgabefehler vom Eingabefehler ab?**

Dabei nehmen wir an, dass es eine unbekannte exakte Eingabe $x \in X$ gibt und eine bekannte mess- und rundungsfehlerbehaftete Eingabe $\tilde{x}$. Der **relative Eingabefehler** ist dann

$$\left|\frac{x_1 - \tilde{x}_1}{x_1}\right| + \cdots + \left|\frac{x_d - \tilde{x}_d}{x_d}\right|. \tag{3.2}$$

Die numerische Auswertung von $f(x)$ erfolgt durch einen (potentiell fehlerhaften) Algorithmus $\tilde{f} : X \to Y$, der auf die fehlerbehafteten Daten $\tilde{x}$ angewendet wird. Für den **relativen Ausgabefehler** erhalten wir somit

$$\frac{|f(x) - \tilde{f}(\tilde{x})|}{|f(x)|}. \tag{3.3}$$

Nebenbemerkung (Warum relative Fehler und nicht absolute Fehler?)

Eine Aussage der Form "Der Algorithmus verursacht einen absoluten Fehler von 0.001" hat für sich allein keinen relevanten Informationsgehalt. Verarbeitet der Algorithmus sehr große Zahlen, so spricht diese Aussage für den Algorithmus. Verarbeitet er jedoch beispielsweise Zahlen unter 0.000001, so ist der Algorithmus völlig unbrauchbar.

Relative Fehler liefern uns Informationen über die Anzahl der korrekt berechneten Ziffern unabhängig davon, in welchem Zahlenbereich der Algorithmus zum Einsatz kommt.

Nachteil relativer Fehler: Wir müssen $x = 0$ und $f(x) = 0$ von den Bertrachtungen ausschließen, was wir stets ohne explizite Erwähnung dieser Tatsache tun werden. Wenn dies sachlich nicht gerechtfertigt ist, bleiben nur absolute Fehler als Ausweg.

Ziel einer **Fehleranalyse** sind Abschätzungen der Form

$$\frac{|f(x) - \tilde{f}(\tilde{x})|}{|f(x)|} \leq g(\delta), \tag{3.4}$$

wobei

$$\sum_{i=1}^{d} \left| \frac{x_i - \tilde{x}_i}{x_i} \right| \leq \delta \tag{3.5}$$

gelten soll. Die Zahl δ ist also eine obere Schranke an den zu erwartenden Datenfehler. Praktisch wird δ mindestens so groß sein wie das Maschinenepsilon.

Die Funktion g ist meistens ein Monom $g(\delta) = c\,\delta^p$ mit $p > 0$ und einer (hoffentlich kleinen) Konstante $c > 0$. Je größer p, desto unempfindlicher reagiert der untersuchte Algorithmus auf Datenfehler. In jedem Fall muss $\lim_{\delta \to 0} g(\delta) = 0$ gelten, damit die Fehlerabschätzung eine sinnvolle Aussage liefert ("Wenn wir den Datenfehler kleiner machen, wird auch der Lösungsfehler kleiner").

Zentrale Werkzeuge für die Herleitung solcher Abschätzungen sind die **Kondition** der Abbildung f und die **Stabilität** des Algorithmus $\tilde{f}$, welche im Folgenden eingeführt werden.

3.4 Kondition

Merke!

Die **Kondition** einer Abbildung $f : X \to Y$ beschreibt den Zusammenhang zwischen Eingabefehler und Ausgabefehler bei exakter Auswertung von f. Sie ist eine Eigenschaft des zu lösenden Problems und hängt in keiner Weise vom gewählten Auswertealgorithmus $\tilde{f}$ ab.

Verstärkt f die Eingabefehler, so ist f **schlecht konditioniert** oder hat schlechte Kondition. Haben Eingabe- und Ausgabefehler die gleiche Größenordnung oder sind die Ausgabefehler sogar kleiner als die Eingabefehler, so ist f **gut konditioniert** oder hat gute Kondition.

Einfaches Beispiel: Berechnung des Schnittpunktes zweier Geraden (IDVID 320).

- Verlaufen die Geraden nahezu senkrecht zueinander, dann gute Kondition.
- Verlaufen die Geraden nahezu parallel, dann schlechte Kondition.

Die Kondition hängt nicht nur von f selbst, sondern auch von der konkreten Eingabe x ab!

Die Kondition eines Problems bzw. der entsprechenden Abbildung f kann man in Zahlen fassen. Für die konkrete Definition dieser **Konditionszahl** gibt es verschiedene Ansätze. Wir wählen zunächst den folgenden:

Merke!

Sei $f : X \to Y$ zweimal stetig differenzierbar an der exakten Eingabe $x \in X$ und sei $\tilde{x} \in X$ eine entsprechende fehlerbehaftete Eingabe. Die Zahl

$$\kappa(x) := \max_{i \in \{1,\dots,d\}} \left| \frac{\partial f}{\partial x_i}(x) \right| \left| \frac{x_i}{f(x)} \right| \tag{3.6}$$

heißt **Konditionszahl** von f an der Stelle x. Mit ihr gilt

$$\left| \frac{f(\tilde{x}) - f(x)}{f(x)} \right| \lessapprox \kappa(x) \sum_{i=1}^{d} \left| \frac{\tilde{x}_i - x_i}{x_i} \right| \tag{3.7}$$

(siehe IDVID 330). Das Symbol $\lessapprox$ ist als “im Wesentlichen kleiner als” zu lesen.

Der hier vorgestellte Konditionsbegriff führt immer zu linearen Fehlerzusammenhängen g in (3.4). Dieser Ansatz ist in der Literatur auch als *differentielle Fehleranalyse* oder *Störungstheorie 1. Ordnung* zu finden.

Schlecht koniditionierte Probleme sind in der Praxis (leider) häufig anzutreffen, z.B. das bei der Computertomografie zu lösende mathematische Problem (siehe Veranstaltung zum wissenschaftlichen Rechnen).

3.5 Beispiele zur Kondition

Multiplikation

Betrachten

$$f : \mathbb{R}^2 \to \mathbb{R}, \qquad f(x_1, x_2) := x_1 \, x_2. \tag{3.8}$$

Für $x_1 \neq 0$ und $x_2 \neq 0$ erhalten wir $\kappa(x) = 1$. Der Ausgabefehler ist als höchstens so groß wie der Eingabefehler.

Für $x_1 = 0$ oder $x_2 = 0$ können wir nur absolute Fehler betrachten:

$$|f(x) - f(\tilde{x})| = |\tilde{x}_1| \, |\tilde{x}_2|. \tag{3.9}$$

Ist z.B. $x_1 = 0$ und x_2 ist sehr groß, also auch $\tilde{x}_2$ sehr groß, so kann das Produkt der gestörten Eingaben erheblich vom tatsächlichen Produkt 0 abweichen. Wähle z.B. $x_1 = 10^{-5}$ und $x_2 = \tilde{x}_2 = 10^6$. Dann ist das berechnete Produkt 10 statt 0. Praktisch ist dieser Fall aber selten von Interesse, da die Null exakt als Gleitkommazahl darstellbar ist. Dennoch ist an dieser Stelle Vorsicht geboten!

```
a = 0.1 * 0.1 - 0.01
b = 10 ** 20
print(a * b)
```

```
173.4723475976807
```

Division

Betrachten

$$f : \mathbb{R} \times (\mathbb{R} \setminus \{0\}) \to \mathbb{R}, \qquad f(x_1, x_2) := \frac{x_1}{x_2}. \tag{3.10}$$

Für $x_1 \neq 0$ erhalten wir $\kappa(x) = 1$. Der Ausgabefehler ist als höchstens so groß wie der Eingabefehler.

Für $x_1 = 0$ können wir nur absolute Fehler betrachten:

$$|f(x) - f(\tilde{x})| = \frac{|\tilde{x}_1|}{|\tilde{x}_2|}. \tag{3.11}$$

Die daraus zu ziehenden Schlüsse entsprechen denen bei der Multiplikation.

```
a = 0.1 * 0.1 - 0.01
b = 10 ** ( -20)
print(a / b)
```

```
173.4723475976807
```

Quadratwurzel

Betrachten

$$f : [0, \infty) \to \mathbb{R}, \qquad f(x) := \sqrt{x}. \tag{3.12}$$

Für $x \neq 0$ erhalten wir $\kappa(x) = \frac{1}{2}$. Der Ausgabefehler ist also kleiner als der Eingabefehler.

Für $x = 0$ erhalten wir den absoluten Fehler

$$|f(x) - f(\tilde{x})| = \sqrt{\tilde{x}} = \sqrt{|0 - \tilde{x}|}, \tag{3.13}$$

der bei $\tilde{x} < 1$ stets größer ist als der Eingabefehler $\tilde{x}$. Beachte hier, dass Eingabefehler $\tilde{x} < 0$ zu undefiniertem Ergebnis bzw. komplexen Zahlen führen!

```
a = 0.01 - 0.1 * 0.1
print(a ** 0.5)
```

```
(8.064844237971058e -26+1.3170890159654386e -09j)
```

```
from numpy import sqrt

a = 0.01 - 0.1 * 0.1
print(sqrt(a))
```

```
nan
```

```
/tmp/ipykernel_25646/1240944823.py:4: RuntimeWarning: invalid value
↪  encountered in sqrt
  print(sqrt(a))
```

Addition und Subtraktion

Betrachten

$$f : \mathbb{R}^2 \to \mathbb{R}, \qquad f(x_1, x_2) := x_1 + x_2. \tag{3.14}$$

Für $x_1 \neq 0$ und $x_2 \neq 0$ erhalten wir

$$\kappa(x) = \frac{\max\{|x_1|, |x_2|\}}{|x_1 + x_2|}. \tag{3.15}$$

Haben x_1 und x_2 gleiches Vorzeichen, dann ist $\kappa(x) \leq 1$. Haben x_1 und x_2 jedoch unterschiedliches Vorzeichen, so kann $|x_1 + x_2|$ klein sein, obwohl weder $|x_1|$ noch $|x_2|$ klein ist. In diesem Fall ist κ sehr groß, die Addition also schlecht konditioniert! Diese Situation wird als **Auslöschung** bezeichnet.

Beispiel: Rechnen mit 5 Dezimalstellen, $x_1 = 0.120587$, $x_2 = -0.120942$. Durch Runden der Eingabewerte entsteht ein Eingabefehler der Größenordnung 10^{-4}. Der Fehler im Ergebnis liegt jedoch bei 10^{-2}, ist also deutlich größer (Details: IDVID 340).

Weiteres Beispiel: Berechne die Differenz $a - b$, wobei statt a und b die auf 64-Bit-Gleitkommazahlen gerundeten Werte verwendet werden.

```
a = 1 + 11e -15
b = 1 + 10e -15
print(f'a = {a:.20f}')
print(f'b = {b:.20f}')
```

```
a = 1.00000000000001110223
b = 1.00000000000000999201
```

Mit exakten Werten erhalten wir $a - b = 10^{-15}$. Der Eingabefehler ist kleiner als 10^{-14}, während der Ausgabefehler größer als 0.11 ist (IDVID 345). Somit weicht die tatsächliche Differenz um **mehr als 11 Prozent vom berechneten Wert** ab. Beachte hierbei: Wir haben exakt gerechnet, das Ergebnis also nicht gerundet. Ursache für den großen Fehler ist somit nicht ein konkreter Algorithmus (z.B. addiere exakt, runde dann auf 64-Bit-Gleitkommazahl), sondern das Problem selbst in Kombination mit leicht gestörten Eingabewerten. Berechnen wir die Differenz algorithmisch (also mit zusätzlichem Runden des Ergebnisses), so erhalten wir

```
c = a - b
print(f'a - b = {c}')
print( 'exakt = 1.0000000000000000e -15')
```

```
a - b = 1.1102230246251565e -15
exakt = 1.0000000000000000e -15
```

Nur die Ziffer vor dem Komma ist korrekt!

Nebenbemerkung (Ist Auslöschung in der Praxis ein relevantes Problem?)

Ob Auslöschung wirklich ein Problem ist, hängt vom Kontext ab. Im obigen Beispiel wurden zwei Zahlen subtrahiert, die beide ungefähr 1 waren. Der absolute Fehler im Ergebnis liegt bei ca. 10^{-16}, was im Vergleich zu 1 verschwindend gering ist. Die Rechnung an sich ist also hinreichend genau. Problematisch ist nur der Fakt, das von den ca. 16 Dezimalziffern des Ergebnisses nur die erste etwas mit dem tatsächlichen Wert zu tun hat. Alle weiteren sind falsch und unbrauchbar. Wird der berechnete Wert als Eingabe für weitere Rechnungen verwendet, ist auch bei gut konditionierten Operationen ein Ausgabefehler von der Größenordnung 10 Prozent zu erwarten!

```
print(1 + c)          # Fehler in c nicht relevant
print(c * 10 ** 15)   # Fehler in c relevant
```

```
1.000000000000001
1.1102230246251565
```

Sinus

Betrachten

$$f : \mathbb{R} \to [-1, 1], \qquad f(x) := \sin x. \tag{3.16}$$

Für $x \neq k\,\pi$, $k \in \mathbb{Z}$, erhalten wir

$$\kappa(x) = \frac{|x|\,|\cos x|}{|\sin x|}. \tag{3.17}$$

Für x in der Nähe von Vielfachen von π wird die Kondition also beliebig groß, während f z.B. in der Nähe von $\frac{\pi}{2}$ sehr gut konditioniert ist.

```
from math import sin

def print_sin(a, b):

    sina = sin(a)
    sinb = sin(b)

    print(sina)
    print(sinb)

    print(f'Eingabefehler: {abs((a - b) / a)}')
    print(f'Ausgabefehler: {abs((sina - sinb) / sina)}')

print_sin(
    3.141593,
    3.141593123456789
)
```

```
-3.464102066193935e -07
 -4.6986699585547026e -07
Eingabefehler: 3.929751219718377e -08
Ausgabefehler: 0.3563890060887287
```

Alle Mantissenziffern des für 3.141593 aus dem fehlerbehafteten Wert 3.141593123456789 berechneten Ergebnisses sind falsch, nur der Exponent stimmt. Der relative Fehler liegt bei 34 Prozent. Die Konditionszahl ist $\kappa(3.141593) \approx 9068997$. Beachte, dass die absoluten Fehler bei Eingabe und Ausgabe dennoch etwa gleich groß sind, das Ergebnis je nach Kontext also durchaus brauchbar sein kann.

```
print_sin(
    1.570796,
    1.570796123456789
)
```

```
0.9999999999999466
0.9999999999999793
Eingabefehler: 7.8595049270588e -08
Ausgabefehler: 3.2751579226443866e -14
```

Hier stimmen fast alle Mantissenziffern trotz gleichem absoluten Fehler in der Eingabe wie zuvor (der relative Eingabefehler ist hier sogar doppelt so groß). Der relative Aus-

gabefehler liegt im Bereich 10^{-13}, ist also um Größenordnungen kleiner als der relative Eingabefehler.

Quadratische Gleichung

Betrachten das Lösen quadratischer Gleichungen $x^2 + p\,x + q = 0$, wobei wir nur die kleinere der beiden möglichen Lösungen berechnen möchten. Haben mit $D(f) := \{(p,q) \in \mathbb{R}^2 : p^2 > 4\,q\}$ also

$$f : D(f) \to \mathbb{R}, \qquad f(x) := -\frac{p}{2} - \sqrt{\frac{p^2}{4} - q}. \tag{3.18}$$

Für $p \neq 0$, $q \neq 0$ und $f(p,q) \neq 0$ (also $p > 0$ oder $q \neq 0$) erhalten wir

$$\kappa(p,q) = \max\left\{ \frac{\frac{|p|}{2}}{\sqrt{\frac{p^2}{4} - q}}, \frac{\left|-\frac{p}{2} + \sqrt{\frac{p^2}{4} - q}\right|}{2\sqrt{\frac{p^2}{4} - q}} \right\}. \tag{3.19}$$

Für $q \leq 0$ gilt somit $\kappa(p,q) \leq 1$, also gute Kondition. Für $q > 0$ wächst die Kondition mit q und wird für $q \to \frac{p^2}{4}$ beliebig groß. Das Problem ist für große q also sehr schlecht konditioniert (Details und grafische Interpretation: IDVID 350).

3.6 Stabilität

Merke!

Ein Algorithmus $\tilde{f}$ zur Auswertung einer Abbildung f heißt **stabil**, wenn $\tilde{f}(x) \approx f(x)$ gilt für alle möglichen Eingaben x.

Ein Algorithmus heißt also stabil, wenn er das tut, was man von ihm erwartet. Für die Formulierung präziserer Aussagen über den erwarteten Zusammenhang zwischen $\tilde{f}(x)$ und $f(x)$ gibt es verschiedene Ansätze. Wir gehen hier nur auf die beiden am weitesten verbreitetenen ein: **Vorwärtsstabilität** und **Rückwärtsstabilität**.

Merke!

Sei $\delta > 0$ eine obere Schranke an den erwarteten relativen Eingabefehler für das betrachtete Problem f. Ein Algorithmus $\tilde{f}$ zur Auswertung von f heißt **vorwärtsstabil**, wenn

$$\frac{|f(x) - \tilde{f}(x)|}{|f(x)|} \leq c\,\kappa(x)\,\delta \tag{3.20}$$

für alle möglichen Eingaben x und ein nicht zu großes $c \geq 0$ gilt.

Vorwärtsstabilität sichert also, dass der durch den verwendeten Algorithmus verursachte Fehler in der Größenordnung des auch bei exakter Rechnung nicht zu vermeidenden Fehlers liegt.

Für konkrete Algorithmen lassen sich Aussagen zur Vorwärtsstabilität oft nur schwer herleiten. Deshalb verwendet man fast immer den folgenden alternativen Stabilitätsbegriff.

Merke!

Sei $\delta > 0$ eine obere Schranke an den erwarteten relativen Eingabefehler für das betrachtete Problem f. Ein Algorithmus $\tilde{f}$ zur Auswertung von f heißt **rückwärtsstabil**, wenn es zu jeder möglichen Eingabe x eine Eingabe $\hat{x}$ gibt, sodass

$$\tilde{f}(x) = f(\hat{x}) \quad \text{und} \quad \sum_{i=1}^{d} \left| \frac{x_i - \hat{x}_i}{x_i} \right| \leq c\,\delta, \tag{3.21}$$

wobei $c \geq 0$ unabhängig von x und nicht zu groß sein soll.

Rückwärtsstabilität sichert also, dass die Ausgabe des Algorithmus der exakten Rechnung mit einer Eingabe entspricht, die im Rahmen der Größenordnung des ohnehin vorhandenen Eingabefehlers ebenfalls in Frage kommt.

Um Rückwärtsstabilität zu zeigen müssen zwei Fragen beantwortet werden:

1. Gibt es zu jeder Eingabe x ein passendes $\hat{x}$?
2. Wie groß ist der Abstand zwischen x und zugehörigem $\hat{x}$ höchstens?

Merke!

Aus der Rückwärtsstabilität folgt stets die Vorwärtsstabilität.

Beweis: IDVID 360.

3.7 Beispiele für Stabilitätsanalysen

Addition/Subtraktion

Nach IEEE 754 erfolgt die Addition zweier Gleitkommazahlen nach folgendem Algorithmus:

Algorithmus

1. Berechne das exakte Ergebnis.
2. Runde entsprechend Rundungsregel auf eine Gleitkommazahl.

Sind x_1, x_2 die beiden Summanden und ist ε das Maschinenepsilon, so liefert der Algorithmus das Ergebnis

$$(1+\varrho)\,(x_1+x_2) \tag{3.22}$$

für ein ϱ mit $|\varrho| \leq \varepsilon$.

Der einfache Algorithmus erlaubt eine Vorwärtsanalyse:

$$\left|\frac{f(x)-\tilde{f}(x)}{f(x)}\right| = \left|\frac{x_1+x_2-(1+\varrho)\,(x_1+x_2)}{x_1+x_2}\right| = |\rho| \leq \varepsilon. \tag{3.23}$$

Die Kondition der Addition erfüllt $\kappa(x) \geq \frac{1}{2}$ (IDVID 370). Wir erhalten somit

$$\left|\frac{f(x)-\tilde{f}(x)}{f(x)}\right| \leq 2\,\kappa(x)\,\varepsilon. \tag{3.24}$$

Da der zu erwartende relative Eingabefehler praktisch immer größer als $2\,\varepsilon$ sein wird (Rundung jedes Summanden auf Gleitkommazahl), ist der untersuchte Algorithmus zur Addition als vorwärtsstabil anzusehen.

Zu Demonstrationszwecken untersuchen wir auch die Rückwärtsstabilität. Haben

$$\begin{aligned}\tilde{f}(x_1,x_2) &= (1+\varrho)\,(x_1+x_2)\\ &= (1+\varrho)\,x_1 + (1+\varrho)\,x_2\\ &= f\big((1+\varrho)\,x_1, (1+\varrho)\,x_2\big),\end{aligned} \tag{3.25}$$

mit $\hat{x}_1 := (1+\varrho)\,x_1$, $\hat{x}_2 := (1+\varrho)\,x_2$ also $\tilde{f}(x) = f(\hat{x})$. Der relative Fehler beim Ersetzen von x durch $\hat{x}$ ist

$$\begin{aligned}\left|\frac{x_1-\hat{x}_1}{x_1}\right| + \left|\frac{x_2-\hat{x}_2}{x_2}\right| &= \left|\frac{x_1-(1+\varrho)\,x_1}{x_1}\right| + \left|\frac{x_2-(1+\varrho)\,x_2}{x_2}\right|\\ &= 2\,|\varrho| \leq 2\,\varepsilon.\end{aligned} \tag{3.26}$$

Da die zu erwartenden Eingabefehler größer als $2\,\varepsilon$ sein werden, ist der Algorithmus als rückwärtsstabil anzusehen.

Lange Summen

Die Summe von n Zahlen soll berechnet werden:

$$f : \mathbb{R}^n \to \mathbb{R}, \qquad f(x) := \sum_{i=1}^{n} x_i. \tag{3.27}$$

Untersuchen dazu zunächst den einfachsten Algorithmus:

Algorithmus

1. Addiere x_1 und x_2 exakt.
2. Runde auf Gleitkommazahl.
3. Für $i = 3, \ldots, n$:
   a) Addiere x_i exakt.
   b) Runde auf Gleitkommazahl.

Mit Maschinenepsilon ε und den aus den Rundungsschritten entstehenden Abweichungen $\varrho_2, \ldots, \varrho_n$ wird also

$$\Big(\cdots\Big(\big((x_1 + x_2)\,(1 + \varrho_2) + x_3\big)\,(1 + \varrho_3) + \cdots\Big)\cdots + x_n\Big)\,(1 + \varrho_n) \tag{3.28}$$

berechnet. Der Algorithmus ist für große n nicht rückwärtsstabil (und auch nicht vorwärtstabil). Für die Konstante in der Definition der Rückwärtsstabilität erhalten wir $c \sim n$ (IDVID 375).

Betrachten nun folgenden alternativen Algorithmus für Summen mit $n = 2^m$ Summanden:

Algorithmus

1. Für $k = 0, \ldots, m - 1$:
   a) Berechne aus den 2^{m-k} vorhandenen Werten die 2^{m-k-1} paarweisen Summen.
   b) Runde jede dieser Summen auf Gleitkommazahlen.

In der letzten Iteration ist nur noch eine Summe mit zwei Summanden zu berechnen, welche das Endergebnis darstellt. Bezeichnen wir die bei den Rundungsschritten entstehenden Abweichungen mit $\varrho_{k,l}$, $k = 0, \ldots, m-1$, $l = 1, \ldots, 2^{m-k-1}$, so entspricht der Algorithmus folgender Formel:

$$\Big(\cdots\Big(\big((x_1+x_2)\,(1+\varrho_{0,1})+(x_3+x_4)\,(1+\varrho_{0,2})\big)\,(1+\varrho_{1,1})+\cdots\Big)\cdots\Big)\,(1+\varrho_{m-1,1}) \tag{3.29}$$

Für die Konstante in der Definition der Rückwärtsstabilität erhalten wir $c \sim m$ (IDVID 380). Der Algorithmus kann also auch für große n noch als rückwärtsstabil (und somit auch vorwärtsstabil) angesehen werden, da $m = \log_2 n$ nur sehr langsam mit n wächst.

Quadratische Gleichung

Wir untersuchen zwei Algorithmen zur Berechnung der kleineren der beiden Lösungen der quadratischen Gleichung $x^2 + p\,x + q = 0$ auf Stabilität.

Algorithmus

1. Multipliziere p mit p, dann runden.
2. Teile durch 4, dann runden.
3. Subtrahiere q, dann runden.
4. Ziehe die Wurzel, dann runden.
5. Teile $-p$ durch 2, dann runden.
6. Subtrahiere das Ergebnis aus Schritt 4, dann runden.

Direktes Auswerten der Lösungsformel Vorwärts- und auch Rückwärtsanalyse sind hier recht mühsam. Mindestens die Rückwärtsanalyse ist mit unseren Mitteln aber machbar. Wählen einen anderen Weg: Betrachten jeden Schritt separat und schließen aus der Kondition der Einzelschritte auf die (Nicht-)Stabilität des Gesamtalgorithmus.

- Schritt 1 ist stabil, liefert also bei fehlerfreier Eingabe p einen Ausgabefehler unterhalb des Maschinenepsilons ε.
- Schritt 2 erhält eine fehlerbehaftete Eingabe, ist aber gut konditioniert. Die Division ohne Runden wird also wieder einen Ausgabefehler in der Größenordnung von ε liefern. Anschließendes Runden erhöht den relativen Fehler höchstens um den Faktor $1 + \varepsilon$, sodass der Gesamtfehler nach Schritt 2 in der Größenordnung von ε bleibt.
- Die Subtraktion in Schritt 3 ist schlecht konditioniert, wenn $q \approx \frac{p^2}{4}$, insbesondere für $q \leq 0$ aber gut konditioniert. Je nach Wert von q kann der Ausgabefehler in Schritt 3 beliebig groß sein. Für $q \leq 0$ wird er bei ε bleiben.
- Schritte 4 und 5 verhalten sich analog zu Schritt 2, also keine relevante Fehlerverstärkung.
- Schritt 6 ist schlecht konditioniert, wenn $p < 0$ und $q \approx 0$; sonst gut konditioniert. Fehlerverhalten analog zu Schritt 3.

Der Algorithmus ist also nur stabil, wenn gilt:

$$q \ll \frac{p^2}{4} \quad \text{und} \quad |q| \gg 0 \text{ wenn } p < 0. \tag{3.30}$$

Umweg über Satz von Vieta Wollen einen Algorithmus, der für $p \leq 0$ bei betragskleinem q stabil ist. Der Satz von Vieta liefert $x_1\, x_2 = q$ für die beiden Lösungen der quadratischen Gleichung. Wir können also zunächst die größere Lösung berechnen und dann daraus die kleinere. Die ersten 5 Schritte bleiben unverändert:

Algorithmus

1. Multipliziere p mit p, dann runden.
2. Teile durch 4, dann runden.
3. Subtrahiere q, dann runden.
4. Ziehe die Wurzel, dann runden.
5. Teile $-p$ durch 2, dann runden.
6. Addiere das Ergebnis aus Schritt 4, dann runden.
7. Teile q durch das Ergebnis aus Schritt 6, dann runden.

Für $p \leq 0$ ist Schritt 6 nun gut Konditioniert (unabhängig vom Wert von q). Auch Schritt 7 ist gut konditioniert solange das Ergebnis aus Schritt 6 nicht zu nah bei 0 liegt. Allerdings ist Schritt 6 für $p > 0$ und $q \approx 0$ nun schlecht konditioniert. Der Algorithmus ist also nur stabil, wenn gilt:

$$q \ll \frac{p^2}{4} \quad \text{und} \quad |q| \gg 0 \text{ wenn } p > 0. \tag{3.31}$$

Kombinierter Algorithmus Um die kleinere Lösung einer quadratischen Gleichung stabil zu erhalten, oder auch um beide Lösungen zu erhalten, sollten also erst die obigen Schritte 1 bis 5 ausgeführt werden. Anschließend ist zu unterscheiden:

- Wenn $p > 0$, dann subtrahiere die Wurzel von $-\frac{p}{2}$.
- Wenn $p < 0$, dann addiere die Wurzel zu $-\frac{p}{2}$.

Anschließend kann über die Beziehung $x_1\, x_2 = q$ (Vieta!) die andere Lösung berechnet werden.

Dieser Algorithmus ist stabil, wenn $q \ll \frac{p^2}{4}$.

Die bei $q \approx \frac{p^2}{4}$ auftretende Instabilität aufgrund von Auslöschungseffekten spielt praktisch kaum eine Rolle. In diesem Fall sind beide Lösungen sehr nah beieinander. Sind beide Lösungen hinreichend weit von der Null entfernt, werden diese trotzdem korrekt berechnet, da dann nur der durch Auslöschung entstandene absolute Fehler relevant ist für den relativen Fehler in den Lösungen. Und dieser ist sehr klein. Liegen jedoch beide Lösungen nah bei Null, so überträgt sich der durch Auslöschung entstandene große relative Fehler auf den relativen Fehler der Lösungen.

3.8 Gesamtfehler

Faustregel: Gute Kondition des Problems f und Stabilität des Algorithmus $\tilde{f}$ garantieren einen hinnehmbar kleinen Gesamtfehler. Schlechte Kondition oder fehlende Stabilität lassen keine brauchbaren Ergebnisse erwarten.

Merke!

Sei x die exakte Eingabe zum Problem f mit Kondition κ. Sei $\tilde{x}$ eine gestörte Eingabe, wobei der relative Eingabefehler durch $\delta > 0$ beschränkt sei. Ist der Algorithmus $\tilde{f}$ zur Auswertung von f vorwärts- oder rückwärtsstabil, so gilt für den relativen **Gesamtfehler**

$$\frac{|f(x) - \tilde{f}(\tilde{x})|}{|f(x)|} \leq \big(\kappa(x) + c\,(1 + \kappa(x)\,\delta)\,\kappa(\tilde{x})\big)\,\delta. \tag{3.32}$$

Dabei ist c die Konstante aus der Definition der Vorwärts- bzw. Rückwärtsstabilität.

Beweis: IDVID 390.

Die Abschätzung zeigt, dass der Ausgabefehler um so kleiner wird, je kleiner der Eingabefehler ist (wenn $\kappa(\tilde{x})$ bei $\tilde{x} \to x$ beschränkt bleibt).

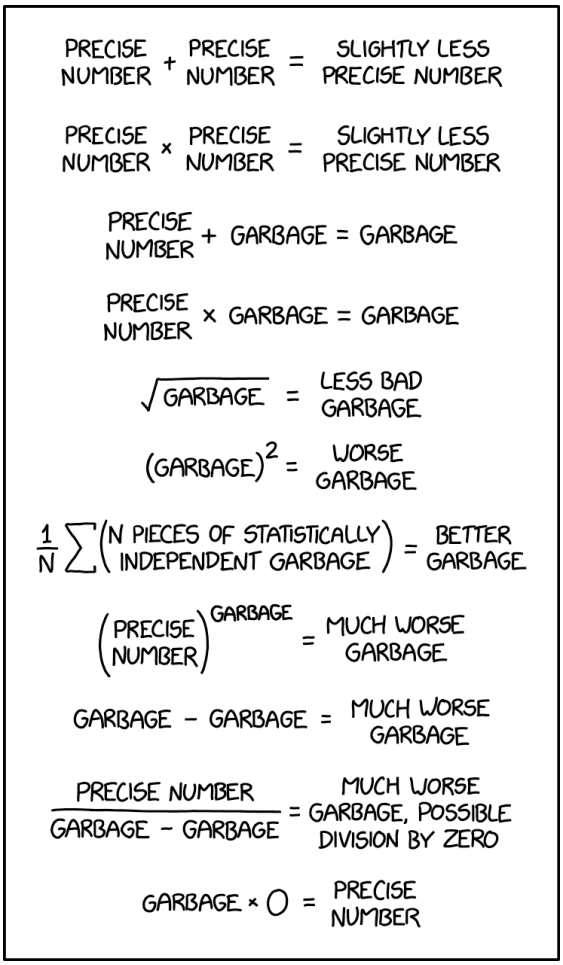


Abbildung 3.1: *
"Garbage In, Garbage Out" should not be taken to imply any sort of conservation law limiting the amount of garbage produced. Quelle: Randall Munroe, xkcd.com/2295

4 Nichtlineare Gleichungen

Beschäftigen uns mit dem numerischen Lösen nichtlinearer Gleichungen

$$f(x) = 0, \tag{4.1}$$

wobei der Definitionsbereich $[a, b]$ von $f : [a, b] \to \mathbb{R}$ so eingeschränkt sei, dass es genau eine Lösung gibt. Nichtlineare Gleichungen können immer auf diese Form gebracht werden, sodass das Lösen nichtlinearer Gleichungen äquivalent zum Finden von Nullstellen ist.

4.1 Kondition

Formale Untersuchungen zur Kondition des Problems sind mit unseren Mitteln nicht möglich. Die Eingabe ist hier eine Funktion. Die Ausgabe ist eine Nullstelle der Funktion. Wir müssten uns also mit Abständen in Funktionenräume beschäftigen um überhaupt über Eingabefehler reden zu können. Grundsätzlich gilt aber, dass die Kondition des Problems vor allem von der konkreten Gestalt der Funktion f festgelegt wird.

Für quadratische Gleichungen haben wir die Kondition des Nullstellenfindens genauer untersucht. Allerdings haben wir dort nicht die gesamte quadratische Funktion als Eingabe betrachtet, sondern nur die in der Funktionsdefinition auftretenden Parameter p und q. Somit konnten wir Eingabefehler mit unseren Mittel ausdrücken.

Eine andere Frage ist die nach der Kondition beim Auswerten der Funktion f an einer Stelle x. Diese Frage können wir mit unseren Mitteln beantworten, sofern eine konkrete Funktion betrachtet wird. Für eine allgemeine Funktion f sind keine Aussagen möglich. Da alle numerische Lösungsverfahren die Funktion f an mehreren Stellen auswerten müssen, ist die Frage nach der Kondition dieser Auswertungen sehr relevant für die Stabilität von Lösungsalgorithmen.

4.2 Iterative Verfahren

Idee

Alle Lösungsverfahren für nichtlineare Gleichungen arbeiten **iterativ**, d.h. sie arbeiten nach folgendem Muster:

1. Wähle einen Startwert $x_0 \in [a, b]$.
2. Für $k = 1, 2 \ldots$ wiederhole:
    a) Berechne eine neue Näherungslösung x_k aus x_{k-1}.

Im Prinzip kann x_k nicht nur von x_{k-1} sondern auch von mehreren Vorgängern abhängen, ist aber hier nicht Thema.

Die Folge $x_0, x_1, x_2, \ldots$ liefert immer bessere Näherungen an die tatsächliche Lösung, sofern das Verfahren sinnvoll konstruiert ist.

Merke!

Ein iteratives Verfahren heißt **konvergent** (auch: "es konvergiert"), wenn für die exakte Lösung x^* gilt:

$$x^* = \lim_{k \to \infty} x_k. \tag{4.2}$$

Um aus der obigen Verfahrensidee konkrete Algorithmen zu erhalten, müssen wir folgende Fragen beantworten:

- Wie wählen wir x_0?
- Wie berechnen wir x_k aus x_{k-1}?
- Wann brechen wir die Iteration ab?

Abbruchkriterium

Die Frage nach dem Abbruch der Iteration ist relevant, weil ein Computer nur endlich viele Rechenschritte durchführen kann. Die Antwort ist eine Abwägung zwischen verfügbarer Rechenzeit und akzeptiertem Lösungsfehler. Folgende Varianten sind üblich:

- Falls vor allem die Rechenzeit relevant ist, bricht man nach einer vorgegebenen Anzahl an Iterationen ab.
- Falls vor allem die Genauigkeit der Lösung relevant ist, bricht man erst ab, wenn die gewünschte Genauigkeit erreicht ist. Da man die exakte Lösung nicht kennt, ist die Machbarkeit dieses Ansatzes vom konkreten Verfahren abhängig.
- Falls vor allem die Genauigkeit der Lösung relevant ist, aber diese bei dem gewählten Verfahren nicht zuverlässig bestimmt werden kann, bricht man ab, wenn sich die Näherungslösungen im Rahmen der gewünschten Genauigkeit nicht mehr ändern.
- Praktisch wird man abbrechen, wenn die Genauigkeit hoch genug ist, und zusätzlich eine (hohe) maximale Iterationsanzahl vorgeben um auch in ungünstigen Fällen (langsame Konvergenz) die Rechenzeit in akzeptablem Rahmen zu halten.

Konvergenzordnung

Aus der Beziehung $x^* = \lim_{k \to \infty} x_k$ lassen sich keine Erkenntnisse zur Konvergenzgeschwindigkeit gewinnen. Deshalb ist man an konkreten Abschätzungen des Lösungsfehlers interessiert:

Merke!

Gilt

$$|x_k - x^*| \leq c\,\gamma^k, \qquad k = 1, 2, \ldots, \tag{4.3}$$

für ein $\gamma \in (0, 1)$ und ein $c > 0$, so heißt das iterative Verfahren **linear konvergent** oder **konvergent mit der Konvergenzordnung 1**.

Gilt sogar

$$|x_k - x^*| \leq c\,\gamma^{(p^k)}, \qquad k = 1, 2, \ldots, \tag{4.4}$$

für ein $p > 1$, ein $\gamma \in (0, 1)$ and ein $c > 0$, so heißt das iterative Verfahren **konvergent mit Konvergenzordnung** p.

Aus beiden Abschätzungen folgt insbesondere $x^* = \lim_{k\to\infty} x_k$, also Konvergenz.

Je höher die Konvergenzordnung, desto schneller die Konvergenz. Entsprechend sind bei hoher Konvergenzordnung weniger Iterationen nötig um einen vorgegebenen maximalen Lösungsfehler zu erreichen als bei niedriger Konvergenzordnung.

Abschätzungen der obigen Form sind jedoch selten direkt zu bekommen. Meist fügt man einen Zwischenschritt folgender Form ein:

$$|x_k - x^*| \leq C\,|x_{k-1} - x^*|^p, \qquad k = 1, 2, \ldots, \tag{4.5}$$

wobei $p \geq 1$ und $C > 0$. Daraus folgt dann Konvergenz mit Konvergenzordnung p, sofern folgende Bedingungen an die Konstante C und den Startfehler $|x_0 - x^*|$ erfüllt sind (IDVID 420):

- $p = 1$: Gilt $C < 1$, so folgt lineare Konvergenz mit $c = |x_0 - x^*|$ und $\gamma = C$.
- $p > 1$: Gilt

$$|x_0 - x^*| < \begin{cases} C^{-\frac{1}{p}}, & \text{falls } C < 1, \\ C^{-\frac{1}{p-1}}, & \text{falls } C \geq 1, \end{cases} \tag{4.6}$$

  so folgt Konvergenz mit Konvergenzordnung p, wobei $c = 1$ und

$$\gamma = \begin{cases} C^{\frac{1}{p}}\,|x_0 - x^*|, & \text{falls } C < 1, \\ C^{\frac{1}{p-1}}\,|x_0 - x^*|, & \text{falls } C \geq 1, \end{cases} \tag{4.7}$$

  gilt.

Beispiel: Gilt für ein festes k, dass $|x_{k-1} - x^*| \approx 0.1$, so sinkt der Fehler in den nächsten Schritten bei linear Konvergenz ($p = 1$) stets nur um den Faktor $C < 1$. Bei quadratischer Konvergenz ($p = 2$) hingegen beträgt der Fehler in den nächsten Schritte etwa 0.01, 0.0001, 0.00000001.

Nebenbemerkung

Während bei linear konvergenten Verfahren allein $C < 1$ die Konvergenz sichert, muss bei superlinear konvergenten Verfahren ($p > 1$) der Startfehler $|x_0 - x^*|$ klein genug sein um Konvergenz zu sichern.

4.3 Bisektionsverfahren

Verfahren

Das Bisektionsverfahren (auch: Intervallschachtelung) ist ein einfaches Verfahren zur Nullstellenbestimmung.

Algorithmus

1. Wähle a_0 and b_0 so, dass $f(a_0)\, f(b_0) \leq 0$ gilt.
2. Setze $x_0 := \frac{a_0+b_0}{2}$.
3. Für $k = 1, 2, \ldots$ wiederhole:
   a) Setze

$$a_k := \begin{cases} a_{k-1}, & \text{falls } f(a_{k-1})\, f(x_{k-1}) \leq 0, \\ x_{k-1}, & \text{sonst} \end{cases} \tag{4.8}$$

   und

$$b_k := \begin{cases} x_{k-1}, & \text{falls } f(a_{k-1})\, f(x_{k-1}) \leq 0, \\ b_{k-1}, & \text{sonst.} \end{cases} \tag{4.9}$$

   b) Setze $x_k := \frac{a_k+b_k}{2}$.

Die Bedingung der Form $f(a)\, f(b) \leq 0$ sichert, dass entweder im Intervall (a, b) eine Nullstelle liegt (bei strenger Ungleichheit) oder ein Intervallende eine Nullstelle ist (bei Gleichheit). Liegen mehrere Nullstellen im Startintervall $[a_0, b_0]$, so liefert das Verfahren nur eine dieser Nullstellen, wobei keine allgemeine Aussagen darüber möglich sind, welche Nullstelle geliefert wird.

Konvergenz und Abbruchkriterium

Merke!

Für stetiges f konvergiert das Bisektionsverfahren linear mit $\gamma = \frac{1}{2}$ und $c = \frac{b_0 - a_0}{2}$ (IDVID 440).

Ist der gewünschte maximale (absolute) Lösungsfehler $\Delta > 0$ vorgegeben, so kann wegen

$$|x_k - x^*| \leq \frac{b_k - a_k}{2} \tag{4.10}$$

die Iteration gestoppt werden, sobald $b_k - a_k \leq 2\,\Delta$. Wegen

$$b_k - a_k = \frac{b_0 - a_0}{2^k} \tag{4.11}$$

ist die Iterationsanzahl dann

$$n = \left\lceil \log_2 \frac{b_0 - a_0}{2\,\Delta} \right\rceil ; \tag{4.12}$$

insbesondere ist sie im Vorfeld exakt bekannt. Ein zusätzliche Begrenzung der Iterationsanzahl als weiteres Abbruchkriterium ist nicht nötig.

Algorithmus

Ein konkreter Algorithmus zur Umsetzung des Bisektionsverfahrens kann wie folgt aussehen (ohne explizites Erwähnen des Rundens nach jeder Rechenoperation):

Algorithmus

1. Zu gegebenen a_0, b_0 mit $a_0 < b_0$ und $f(a_0)\,f(b_0) \leq 0$ setze $a := a_0$, $b := b_0$.
2. Setze $A := f(a)$ und $B := f(b)$.
3. Wiederhole solange $b - a > 2\,\Delta$:
   a) Berechne $x := \frac{a+b}{2}$.
   b) Setze $X := f(x)$.
   c) Falls $A\,X \leq 0$,
   setze $b := x$ und $B := X$,
   sonst
   setze $a := x$ und $A := X$.
4. Berechne das Ergebnis $x := \frac{a+b}{2}$.

Eine vollständige Stabilitätsanalyse ist trotz des relativ einfachen Algorithmus recht mühsam. Das Verfahren gilt als sehr stabil. Wenn die Funktion f also fehlerfrei auswertbar ist (fehlerfreie Eingabe), dann liegt der Ausgabefehler in der Größenordnung des durch übliche fehlerbehaftete Eingaben auch bei exakter Rechnung nicht zu vermeidenden Ausgabefehlers (Kondition!), sofern Δ im Abbruchkriterium entsprechend klein gewählt wird. Einige Hinweise zur Einschätzung der Stabilität:

- Die Summen $a+b$ beim Berechnen der Intervallmitten sind anfällig für Auslöschung falls $0 \in (a, b)$. Der relative Fehler der Intervallmitten kann dann recht groß sein. Die Werte x_k werden also nicht exakt in der Mitte liegen, was aber aufgrund der Arbeitsweise des Bisektionsverfahrens keinen Einfluss auf das Endergebnis hat.
- Rundungsfehler bei der Berechnung der Intervallmitten haben ebenfalls keinen Einfluss auf das Endergebnis. Insbesondere werden sie nicht von Schritt zu Schritt verstärkt, da stets eine neue (im Rahmen der Stabilitätsanalyse exakte) Auswertung von f stattfindet.
- Auslöschungseffekte bei $b-a$ im Abbruchkriterium können zu leichten Abweichungen bei der theoretisch ermittelten Iterationsanzahl führen. Da die Fehlerschranke Δ in der Praxis ohnehin nur eine grobe Vorgabe/Schätzung ist, spielt dieses Problem in der Praxis keine Rolle.

4.4 Newton-Verfahren

Verfahren

Das Newton-Verfahren nutzt zusätzlich zu den Funktionswerten von f auch die erste Ableitung f' um Informationen über die Lage einer Nullstelle zu erhalten.

Algorithmus

1. Wähle einen Startpunkt x_0.
2. Für $k = 1, 2, \ldots$ wiederhole:

   a) Setze

$$x_k := x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}. \tag{4.13}$$

Dieses Verfahren wählt also x_k als Nullstelle der Tangente an f an der Stelle x_{k-1} (IDVID 450).

Konvergenz und Abbruchkriterium

Merke!

Seien a und b so gewählt, dass $x^* \in (a, b)$, und sei f zweimal stetig differenzierbar in (a, b) mit

$$0 < c_1 \leq |f'(x)| \quad \text{und} \quad |f''(x)| \leq c_2 \quad \text{für alle } x \in (a, b) \tag{4.14}$$

für Konstanten c_1 und c_2. Gilt $x_{k-1} \in (a, b)$, so folgt

$$|x_k - x^*| \leq \frac{c_2}{2\,c_1} |x_{k-1} - x^*|^2 \tag{4.15}$$

(IDVID 460).

Sofern alle Iterierten im Intervall (a, b) liegen, folgt aus dieser Abschätzung quadratische Konvergenz $|x_k - x^*| \leq c\,\gamma^{(2^k)}$ mit $c = 1$ und

$$\gamma = \begin{cases} \sqrt{\frac{c_2}{2\,c_1}}\,|x_0 - x^*|, & \text{falls } c_2 < 2\,c_1, \\ \frac{c_2}{2\,c_1}\,|x_0 - x^*|, & \text{sonst,} \end{cases} \tag{4.16}$$

wobei

$$|x_0 - x^*| < \begin{cases} \sqrt{\frac{2\,c_1}{c_2}}, & \text{falls } c_2 < 2\,c_1, \\ \frac{2\,c_1}{c_2}, & \text{sonst} \end{cases} \tag{4.17}$$

gelten muss. Damit das Newton-Verfahren konvergiert müssen also insbesondere zwei Bedingungen erfüllt sein:

- Der Startwert x_0 liegt nah genug bei der (unbekannten) Lösung x^*.
- Die Iterierten x_k müssen in der Nähe von x^* bleiben.

Sind diese Bedingungen erfüllt, so konvergiert das Newton-Verfahren quadratisch. Man spricht auch von **lokaler quadratischer Konvergenz**. Im Gegensatz dazu ist das Bisektionsverfahren global konvergent, allerdings nur linear.

Da die Konstanten c_1 und c_2 praktisch nicht zugänglich sind, kann aus der Abschätzung für die Konvergenzordnung kein in der Praxis einsetzbares Abbruchkriterium gewonnen werden. Man behilft sich mit der Tatsache, dass bei Konvergenz der Abstand $|x_k - x_{k-1}|$ zwischen zwei Iterierten gegen Null geht. Abgebrochen wird also, wenn dieser Abstand unter eine vorgegebene Schranke fällt.

Algorithmus

Eine konkrete Umsetzung des Newton-Verfahrens kann wie folgt aussehen (ohne explizites Erwähnen des Rundens nach jeder Rechenoperation):

Algorithmus

1. Zu gegebenem Startwert x_0 setzte $x_{\text{alt}} := x_0$.
2. Berechne $x := x_0 - \frac{f(x_0)}{f'(x_0)}$.
3. Wiederhole solange $|x - x_{\text{alt}}| > \Delta$:
   a) Setze $x_{\text{alt}} := x$.
   b) Berechne $x := x - \frac{f(x)}{f'(x)}$.

Dieser Algorithmus ist stabil. Insbesondere werden Rundungsfehler im Laufe der Iterationen nicht verstärkt. Jede neue Iteration kann als Neustart des Algorithmus mit anderem Startwert betrachtet werden. Da die Wahl des Startwertes (innerhalb gewisser Grenzen) keinen Einfluss auf die Lösung hat, spielen Rundungsfehler des Startwertes keine Rolle.

Beispiele

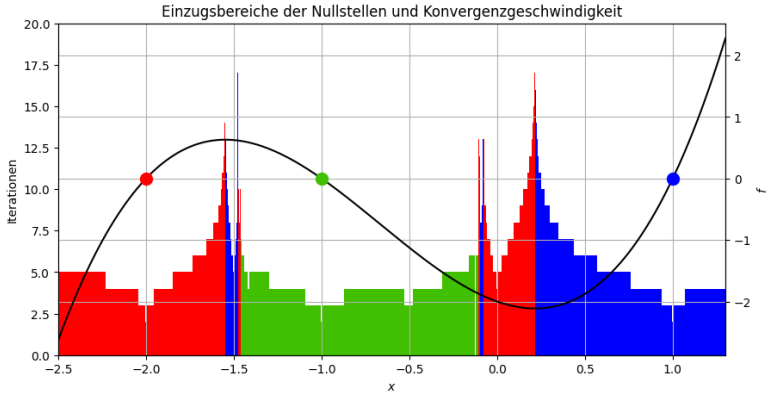
Um das Problem der nicht globalen Konvergenz zu verdeutlichen führen wir das Newton-Verfahren für viele verschiedene Startwerte x_0 aus und stellen die Konvergenz bzw. Nichtkonvergenz grafisch dar. Dabei sehen wir eine Näherungsfolge $x_0, x_1, x_2, \ldots$ als konvergent an, wenn innerhalb von 1000 Iterationen der Abstand $|x_k - x_{k-1}|$ unter 10^{-10} sinkt. Zusätzlich stellen wir dar, gegen welche von ggf. mehreren Nullstellen die Folge konvergiert.

In den folgenden Diagrammen sind neben der betrachteten Funktion f die Iterationsanzahlen bis zum erstmaligen Unterschreiten der Fehlerschranke und (farblich) die angenäherten Nullstellen ausgehend von verschiedenen Startwerten x_0 dargestellt.

Polynom 3. Grades Konvergenzverhalten für das Polynom

$$f(x) = x^3 + 2\,x^2 - x - 2, \quad x \in [-2.5, 1.3]. \tag{4.18}$$

```
visualize_newton(
    lambda x: x ** 3 + 2 * x ** 2 - x - 2,
    lambda x: 3 * x ** 2 + 4 * x - 1,
    [ -2, -1, 1],
    -2.5, 1.3
)
```

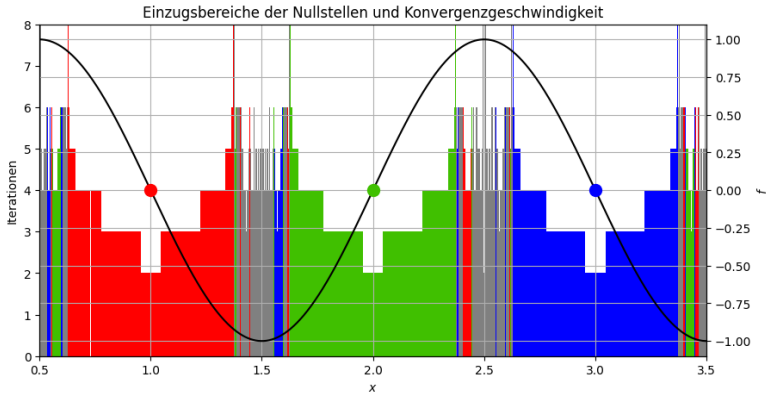


An den Extremstellen gilt $f'(x) = 0$, sodass das Newton-Verfahren dort nicht anwendbar ist (alle Berechnungen liefern `nan`, sodass die Abbruchbedingung nie erfüllt ist). Für x_0, für die weitere Iterierte auf eine Extremstelle fallen, ist das Newton-Verfahren ebenfalls nicht anwendbar. Ist x_0 in der Nähe einer Extremstelle, so führen schon sehr kleine Änderungen an x_0 zum Wechsel der angestrebten Nullstelle.

Sinus Konvergenzverhalten für die Sinusfunktion

$$f(x) = \sin(\pi\, x), \quad x \in [0.5, 3.5]. \tag{4.19}$$

```
visualize_newton(
    lambda x: np.sin(np.pi * x),
    lambda x: np.pi * np.cos(np.pi * x),
    [1, 2, 3],
    0.5, 3.5
)
```



Liegt x_0 in der Nähe einer Extremstelle, so verläuft die Tangente sehr flach. Entsprechend weit entfernt ist x_1, sodass auch Startwerte innerhalb des dargestellten Intervalls zur Annäherung an weit entfernt liegende Nullstellen führen können. Liegt die angestrebte

Nullstelle nicht im dargestellten Intervall, so wird der entsprechende Startwert grau eingefärbt (gleiche Farbe wie bei Nicht-Konvergenz).

5 Nichtlineare Gleichungssysteme

Wir werfen einen kurzen Blick auf das numerische Lösen nichtlinearer Gleichungssysteme

$$F(x) = 0, \quad F = (f_1, \ldots, f_d) : D \to \mathbb{R}^d, \tag{5.1}$$

wobei $D \subseteq \mathbb{R}^d$ der Definitionsbereich der vektorwertigen Funktion F ist. Beschränken uns auf den Fall, dass die Anzahl der Unbekannten mit der Anzahl der Gleichungen übereinstimmt.

Gelegentlich treten nichtlineare Gleichungssysteme mit nur kleinem d auf. Meist entstehen die Systeme jedoch als endliche Näherung kontinuierlicher Probleme (Strömungsmechanik usw.) und weisen dann sehr viele Unbekannte auf.

Ziel dieses Kapitels ist nicht die detaillierte Behandlung konkreter Verfahren, sondern das Aufzeigen von zusätzlichen Fragen und Problemen, die im eindimensionalen Fall nicht auftraten.

5.1 Bisektion vs. Dimension

Prinzipiell lässt sich das Bisektionsverfahren auf höhere Dimensionen übertragen, allerdings nicht so einfach wie es zunächst erscheinen mag; siehe z.B. [VI86]. Hauptproblem ist die mit der Dimension d exponentiell wachsende Anzahl zu untersuchender Punkte. Während im eindimensionalen Fall der Suchbereich durch zwei Punkte begrenzt ist, werden in d Dimensionen 2^d Punkte benötigt. Dies entspricht gerade der Anzahl der Ecken eines d-dimensionalen Quaders.

Beispiel: Für ein nichtlineares Gleichungssystem mit nur $d = 30$ Unbekannten und Gleichungen wäre das Suchgebiet durch ca. 1 Milliarde Punkte begrenzt!

5.2 Newton-Verfahren

Das Newton-Verfahren lässt sich leicht auf Gleichungssysteme übertragen. Das Newton-Verfahren und daraus abgeleitete Verfahren (siehe unten) sind die wichtigsten Verfahren für das Lösen auch großer nichtlinearer Gleichungssysteme.

Verfahren

Wie bei $d = 1$ lösen wir eine linearisierte Version des nichtlinearen Gleichungsystems, wobei wir die Linearisierungspunkte sukzessive ausgehend von einem Startpunkt x_0 als Lösung des vorhergehenden linearisierten Systems berechnen.

Für gegebenen Punkt x_{k-1} liefert der Satz von Taylor bei zweimal stetig differenzierbarem F die Näherung

$$F(x) \approx F(x_{k-1}) + J_F(x_{k-1})\,(x - x_{k-1}), \tag{5.2}$$

wobei

$$J_F(x_{k-1}) := \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x_{k-1}) & \cdots & \frac{\partial f_1}{\partial x_d}(x_{k-1}) \\ \vdots & & \vdots \\ \frac{\partial f_d}{\partial x_1}(x_{k-1}) & \cdots & \frac{\partial f_d}{\partial x_d}(x_{k-1}) \end{bmatrix} \tag{5.3}$$

die Matrix der ersten partiellen Ableitungen der Komponenten $f_1, \ldots, f_d$ von F ist ("Jacobi-Matrix"). Das nichtlineare Gleichungssystem $F(x) = 0$ wird also in einer (kleinen) Umgebung von x_{k-1} näherungsweise durch das lineare Gleichungssystem

$$J_F(x_{k-1})\, x = J_F(x_{k-1})\, x_{k-1} - F(x_{k-1}) \tag{5.4}$$

beschrieben. Ist $J_F(x_{k-1})$ invertierbar, so besitzt dieses die eindeutige Lösung

$$x_k := x_{k-1} - J_F(x_{k-1})^{-1}\, F(x_{k-1}). \tag{5.5}$$

Dies ist die Iterationsvorschrift für das mehrdimensionale Newton-Verfahren.

Probleme

Analog zum eindimensionalen Fall (aber technisch aufwendiger) kann man zeigen, dass das mehrdimensionale Newton-Verfahren quadratisch konvergiert, also

$$|x_k - x^*| \leq C\, |x_{k-1} - x^*|^2 \tag{5.6}$$

gilt, sofern die Funktion F gewisse, recht strenge Bedingungen erfüllt und x_0 nah genug an der Lösung x^* liegt. Beachte, dass x^* und alle x_k hier Vektoren aus $\mathbb{R}^d$ sind, in der Ungleichung nun also Längen von Vektoren verglichen werden.

Wie man x_0 in der Praxis wählen muss um quadratische Konvergenz oder überhaupt Konvergenz zu erhalten, ist nicht so klar. Der Satz von Kantorovich liefert hier praktisch verwertbare Anhaltspunkte. Letztlich wird es aber meist auf "Versuch und Irrtum" hinauslaufen.

Neben dem auch im eindimensionalen Fall vorhandenen Konvergenzproblem macht zusätzlich die Jacobi-Matrix $J_F(x_{k-1})$ Probleme. Selbst wenn diese als auswertbare Formel vorliegt, müssen in jedem Interationsschritt die d^2 Einträge neu berechnet werden (eigentlich nur $\frac{d\,(d+1)}{2}$, da symmetrisch). Oft ist F so beschaffen, dass man J_F gar nicht explizit in Formeln ausdrücken kann. Dann muss $J_F(x_{k-1})$ numerisch bestimmt werden. Die Probleme um J_F haben vielfältige Lösungen hervorgeracht, die zu unterschiedlichen, so genannten Newton-ähnlichen Verfahren geführt haben.

5.3 Newton-ähnliche Verfahren

Bezüglich der Jacobi-Matrix J_F im mehrdimensionalen Newton-Verfahren sind zwei Probleme zu lösen:

- Ersetze J_F durch eine Matrix mit möglichst vielen Nullen (weniger Rechenaufwand) aber annähernd gleicher Wirkung.
- Bestimme J_F numerisch stabil (!) mit möglichst geringem Aufwand.

Der zweite Punkt ist besonders schwierig, weil numerische Differentiation ein schlecht konditioniertes Problem ist (siehe Veranstaltung zum wissenschaftlichen Rechnen).

Lösungsansätze:

- Beim **vereinfachten Newton-Verfahren** wird $J_F(x_{k-1})$ nicht bei jedem Schritt neu berechnet, sondern über einige Schritte konstant gehalten. Hintergrund ist, dass J_F sich von Schritt zu Schritt ohnehin meist nur wenig ändert.
- **Quasi-Newton-Verfahren** bestimmen eine Nährung für J_F aus den zurückliegenden Iterationsschritten. Sie "sammeln" also im Laufe der Zeit Informationen über die Ableitungen von F. Dies ist deutlich effizienter (aber technisch schwieriger) als numerische Differentiation ohne Nutzung der ohnehin bereits bekannten Funktionswerte.
- Bei manchen Anwendungen, z.B. beim Lösen von partiellen Differentialgleichungen (vgl. Veranstaltung zum wissenschaftlichen Rechnen), hat J_F eine gut verstandene Struktur, die den Einsatz **schneller Löser** für das lineare Gleichungssystem in jedem Iterationsschritt zulässt. Dies ist z.B. möglich, wenn man weiß, dass J_F nur in der Nähe der Diagonale von Null verschieden ist.

5.4 Ausflug: Newton-Fraktal

Um die Konvergenz bzw. Nichtkonvergenz des Newton-Verfahrens in Abhängigkeit vom Startpunkt zu visualisieren betrachten wir verschiedene Beispiele mit $d = 2$.

Wir generieren ein regelmäßiges Rechteckgitter aus Startpunkten und führen das Newton-Verfahren mit jedem dieser Punkte als Startpunkt aus. Jeder Lösung der Gleichung $F(x, y) = (0, 0)$ wird eine Farbe zugeordnet. Die Startpunkte werden in der Farbe der Lösung gefärbt, zu der die Iteration führt. Je langsamer die Konvergenz (hohe Iterationsanzahl), desto dunkler die Farbe.

Startpunkte, die nicht zur Konvergenz führen, werden schwarz erscheinen. Zur leichteren Wahrnehmung dieser Punkte erstellen wir eine zweite Grafik, die diese Startpunkte in weiß auf schwarzem Grund darstellt. Grautöne entsprechen langsamer Konvergenz.

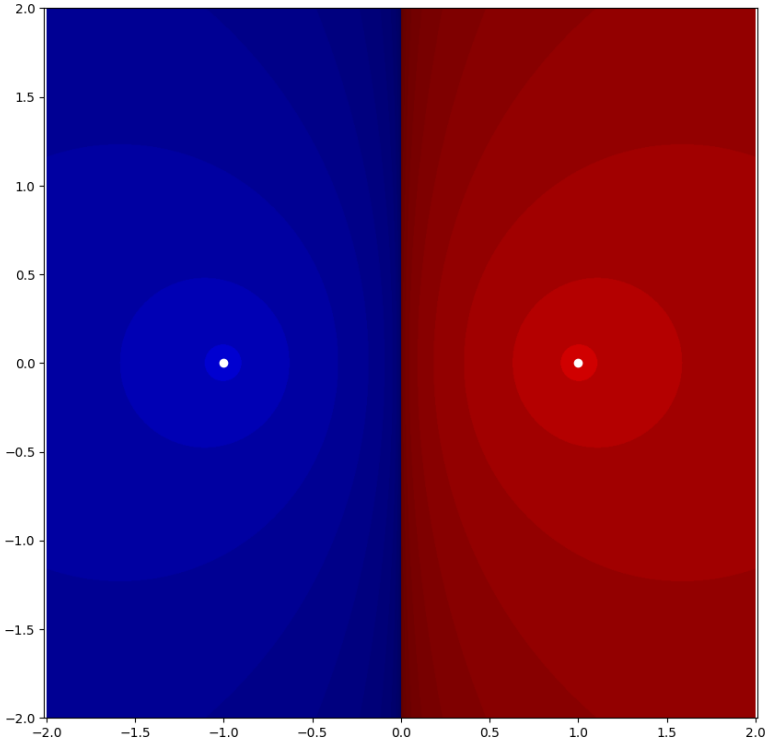
Schauen uns zunächst die Ergebnisse für die Funktion

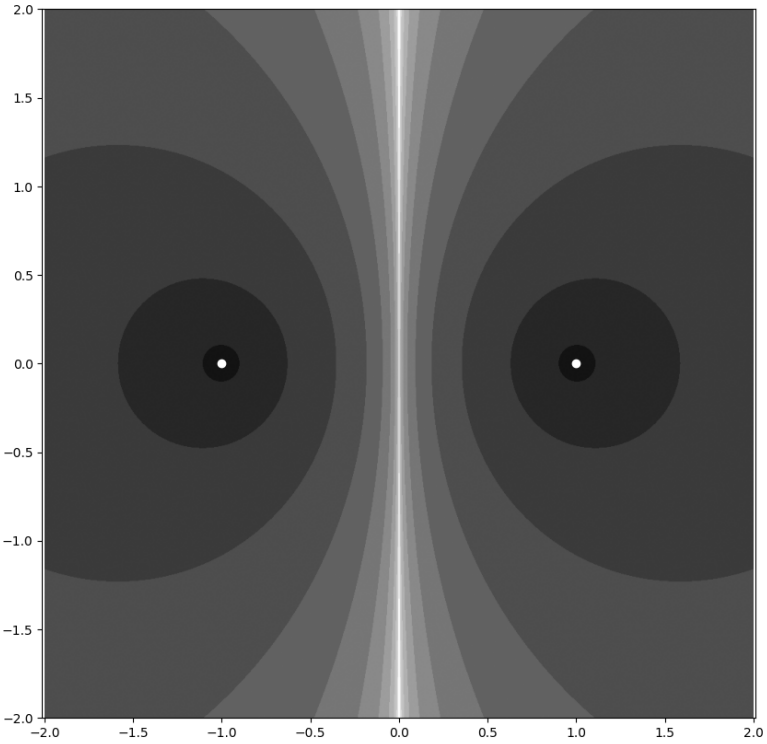
$$F(x, y) := \begin{bmatrix} x^2 - y^2 - 1 \\ 2\,x\,y \end{bmatrix} \tag{5.7}$$

an. Diese Funktion hat zwei Nullstellen bei $(-1, 0)$ und $(1, 0)$ (weiße Punkte).

```
x, y = sympy.symbols('x y', real=True)
```

```
#z = x + y * sympy.I
#f = z ** 2 - 1
#f1 = sympy.re(f)
#f2 = sympy.im(f)

f1 = x ** 2 - y ** 2 - 1
f2 = 2 * x * y

roots = [[1, 0], [ -1, 0]]

newton_fractal(f1, f2, x, y, roots)
```





Das Newton-Verfahren konvergiert für alle Startpunkte, die nicht auf der vertikalen Koordinatenachse liegen. Je näher der Startpunkt an einer Nullstelle liegt, desto schneller die Konvergenz.

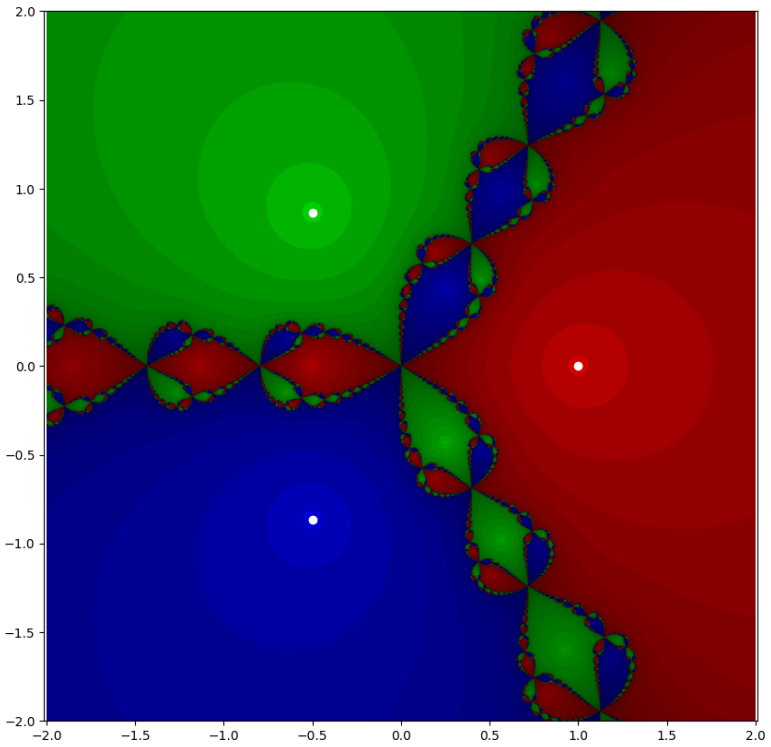
Erhöhen wir den Polynomgrad auf 3, z.B.

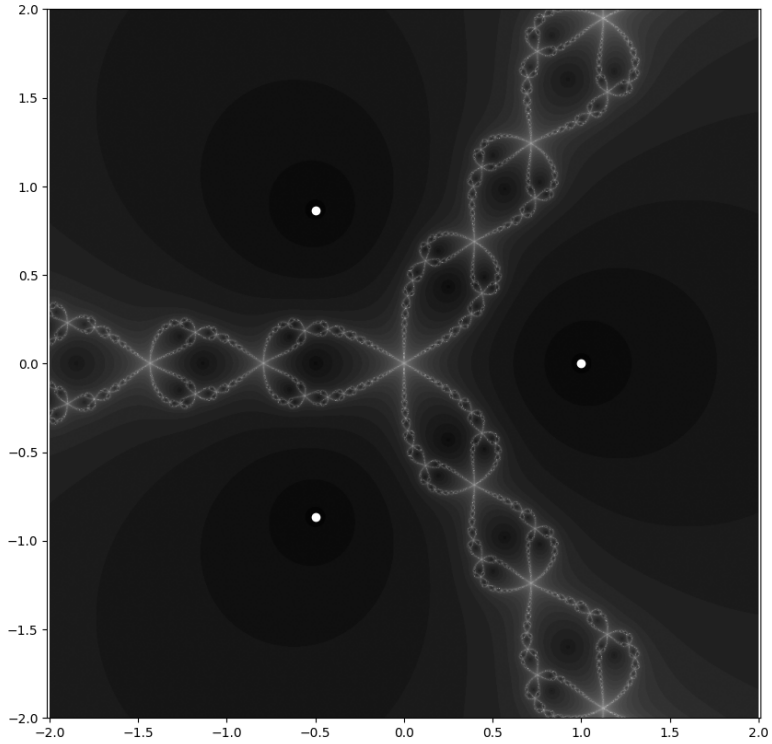
$$F(x,y) := \begin{bmatrix} x^3 - 3\,x\,y^2 - 1 \\ 3\,x^2\,y - y^3 \end{bmatrix}, \tag{5.8}$$

so wird das Konvergenzverhalten deutlich komplexer. Die Funktion hat 3 Nullstellen (weiße Punkte).

```
x, y = sympy.symbols('x y', real=True)

#z = x + y * sympy.I
#f = z ** 3 - 1
#f1 = sympy.re(f)
#f2 = sympy.im(f)

f1 = x ** 3 - 3 * x * y ** 2 - 1
f2 = 3 * x ** 2 * y - y ** 3

roots = [[1, 0], [ -0.5, 0.5 * np.sqrt(3)], [ -0.5, - 0.5 *
↪  np.sqrt(3)]]
```

```
newton_fractal(f1, f2, x, y, roots)
```





Bei Start in der Nähe einer Nullstelle konvergiert das Newton-Verfahren zu dieser Nullstelle. Liegt der Startpunkt jedoch etwa gleich weit entfernt von zwei Nullstellen, so führen schon sehr kleine Änderungen am Startwert zu einem Wechsel der angenäherten Nullstelle.

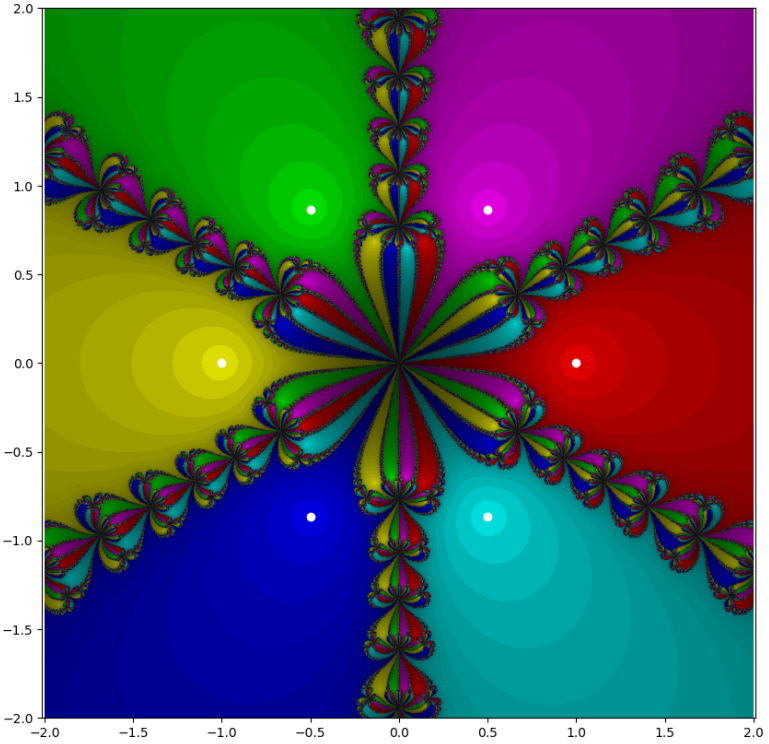
Die Menge der Startpunkte von denen aus das Newton-Verfahren nicht konvergiert heißt **Newton-Fraktal**. Diese Menge hat eine äußerst komplexe sich sowohl in der Ausdehnung als auch im Detail immer wieder wiederholende Struktur. Das Newton-Fraktal ist weder Linie, noch Fläche, sondern eine Menge mit gebrochener Dimension zwischen 1 und 2 bei ca. 1.42. Siehe z.B. Fractal Basins of Attraction Associated with a Damped Newton's Method für Details.

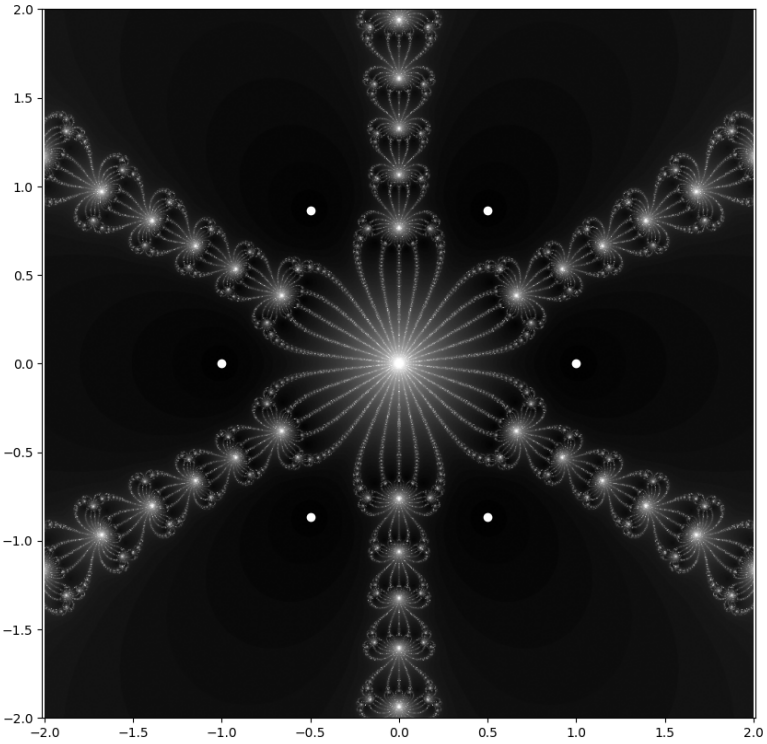
Zur besseren Visualisierung des Newton-Fraktals (Zoom usw.) existieren verschiedene Computerprogramme, z.B. XaoS (open-source für alle gängigen Plattformen).

Polynom 6. Grades:

```
x, y = sympy.symbols('x y', real=True)

#z = x + y * sympy.I
#f = z ** 6 - 1
#f1 = sympy.re(f)
#f2 = sympy.im(f)
```

```
f1 = x ** 6 - 15 * x ** 4 * y ** 2 + 15 * x ** 2 * y ** 4 - y ** 6 - 1
f2 = 6 * x ** 5 * y - 20 * x ** 3 * y ** 3 + 6 * x * y ** 5

roots = [
    [1, 0], [ -0.5, 0.5 * np.sqrt(3)], [ -0.5, - 0.5 * np.sqrt(3)],
    [ -1, 0], [0.5, 0.5 * np.sqrt(3)], [0.5, - 0.5 * np.sqrt(3)]
]

newton_fractal(f1, f2, x, y, roots)
```

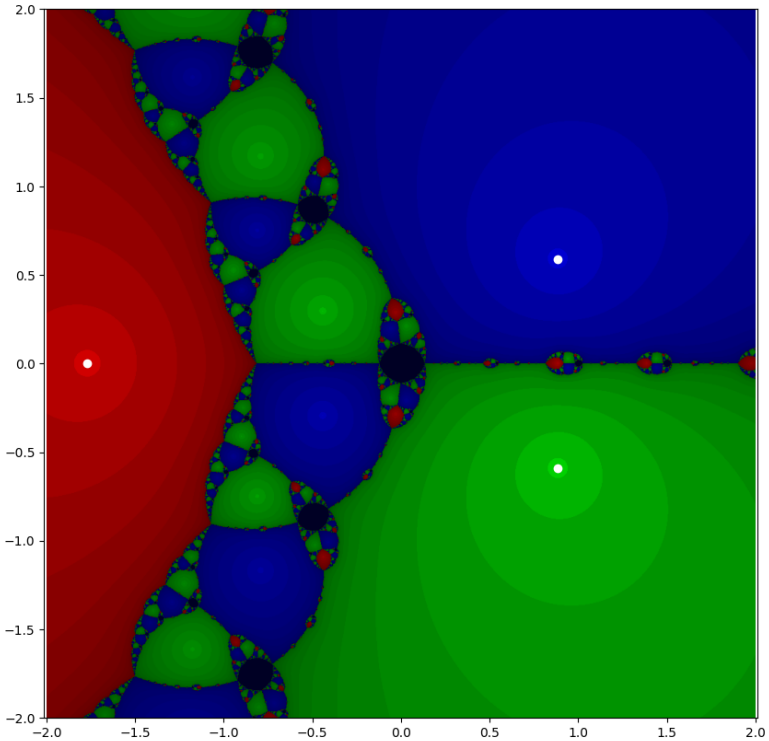


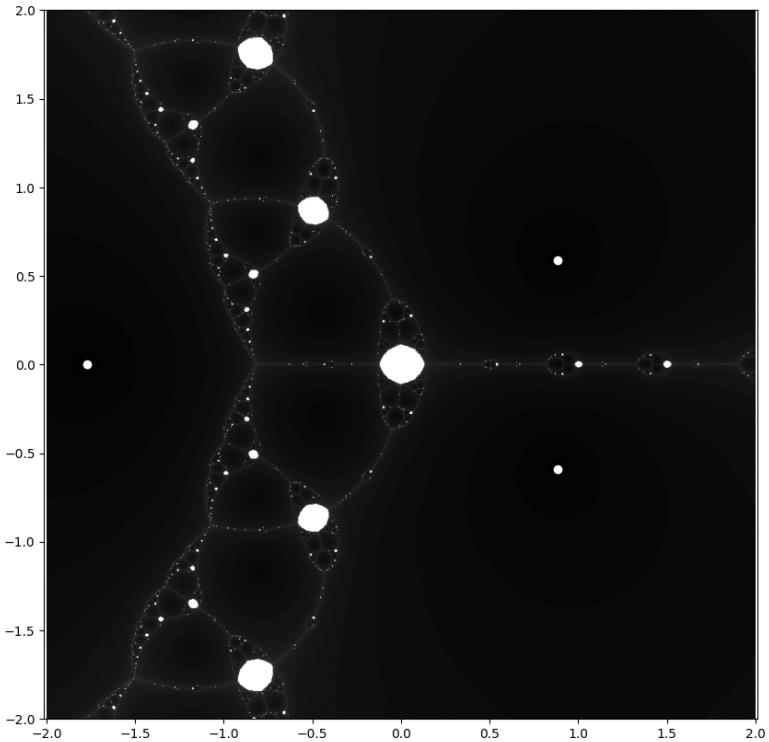


Für manche Funktionen gibt es ganze Flächen an Startpunkten, bei denen das Newton-Verfahren nicht konvergiert. Beispiel:

$$F(x, y) := \begin{bmatrix} x^3 - 3\,x\,y^2 - 2\,x + 2 \\ 3\,x^2\,y - y^3 - 2\,y \end{bmatrix}. \tag{5.9}$$

```
x, y = sympy.symbols('x y', real=True)

#z = x + y * sympy.I
#f = z ** 3 - 2 * z + 2
#f1 = sympy.re(f)
#f2 = sympy.im(f)

f1 = x ** 3 - 3 * x * y ** 2 - 2 * x + 2
f2 = 3 * x ** 2 * y - y ** 3 - 2 * y

roots = [[ -1.7693, 0], [0.88465, -0.58974], [0.88465, 0.58974]]

newton_fractal(f1, f2, x, y, roots)
```


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

6 Lineare Gleichungssysteme

Lineare Gleichungssysteme

$$\begin{aligned} a_{1,1}\,x_1 + \cdots + a_{1,n}\,x_n &= b_1 \\ &\vdots \\ a_{m,1}\,x_1 + \cdots + a_{m,n}\,x_n &= b_m \end{aligned} \tag{6.1}$$

entstehen in allen Bereichen der Wissenschaft, entweder direkt zur Formulierung eines praktischen Problems oder als Teilschritt innerhalb komplexerer Lösungsstrategien (vgl. auch Veranstaltung zum wissenschaftlichen Rechnen). Dabei sind die Zahlen $a_{i,j}$ und b_i gegeben und die Zahlen $x_1, \ldots, x_n$ gesucht.

Zur Vereinfachung der Darstellung setzt man üblicherweise die **Systemmatrix**

$$A := \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{6.2}$$

und den **Vektor der rechten Seiten**

$$b := \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}. \tag{6.3}$$

Das Gleichungssystem bekommt damit die Form

$$A\,x = b \tag{6.4}$$

mit dem **Lösungsvektor**

$$x := \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}. \tag{6.5}$$

Es existieren vielfältige numerische Lösungsverfahren mit unterschiedlichen Eigenschaften bzgl. Stabilität, Rechenaufwand, Speicherbedarf und Anwendungsbreite. Schauen uns hier zwei relativ breit anwendbare Verfahren näher an und geben einen kurzen Überblick über weitere Verfahrensideen.

Beschränken uns hier auf invertierbare quadratische Systemmatrizen, also auf den Fall $m = n$. Insbesondere folgt aus der Forderung der Invertierbarkeit, dass das Gleichungssystem genau eine Lösung besitzen soll. Diese kann dann als

$$x = A^{-1}\,b \tag{6.6}$$

geschrieben werden. Erst im nächsten Kapitel gehen wir auf den allgemeinen Fall ein.

6.1 Kondition

Zur Beurteilung der Kondition beim Lösen eines linearen Gleichungssystems muss zunächst geklärt werden, welche Größen als (potentiell fehlerbehaftete) Eingabe zu betrachten sind. Hier gibt es zwei Ansätze:

- Nur b wird als Eingabe betrachtet und es ist die Kondition der Abbildung $b \mapsto A^{-1}\,b$ gesucht. Dieser Ansatz ist leichter zu handhaben und wird immer verwendet, wenn A sehr genau oder sogar exakt bekannt ist.
- A und b werden beide als Eingaben betrachtet und es ist die Kondition der Abbildung $(A, b) \mapsto A^{-1}\,b$ gesucht.

Weiterhin bestehen gewisse Freiheiten bei der Beantwortung der Frage, wie wir Abweichungen zwischen exakten und fehlerbehafteten Ein- bzw. Ausgaben ausdrücken. Bisher haben wir Konditionsbetrachtungen nur für Abbildungen $f : \mathbb{R}^d \to \mathbb{R}$ angestellt. Auf der Eingabeseite hatten wir uns für die Summe der komponentenweisen relativen Fehler entschieden. Auf der Ausgabeseite bestand gar keine Wahl. Im Kontext linearer Gleichungssysteme muss nun auch auf der Ausgabeseite entschieden werden, wie die relativen Fehler in den einzelnen Komponenten zu einem Gesamtausgabefehler zusammengeführt werden.

Vektornormen

Für einen Vektor $v \in \mathbb{R}^n$, z.B. den Vektor der komponentenweisen relativen Ein- oder Ausgabefehler, können wir verschiedene **Normen** einführen, also Zahlen $\|v\|$, die die Größe der einzelnen Einträge des Vektors sinnvoll zusammenfassen. Unter "sinnvoll" verstehen wir, dass folgende Eigenschaften gelten sollen:

- $\|v\| \geq 0 \quad$ für alle $v \in \mathbb{R}^n$,
- $\|v\| = 0 \;\Rightarrow\; v = 0$,
- $\|\alpha\, v\| = |\alpha|\,\|v\| \quad$ für alle $\alpha \in \mathbb{R}$ und alle $v \in \mathbb{R}^n$,
- $\|v + w\| \leq \|v\| + \|w\| \quad$ für alle $v, w \in \mathbb{R}^n$.

Beispiele:

- **1-Norm**:

$$\|v\|_1 := \sum_{i=1}^{n} |v_i|, \tag{6.7}$$

- **∞-Norm**:

$$\|v\|_\infty := \max_{i \in \{1,\ldots,n\}} |v_i|, \tag{6.8}$$

- **euklidische Norm**:

$$\|v\|_2 := \sqrt{\sum_{i=1}^{n} v_i^2}. \tag{6.9}$$

Alle Normen in $\mathbb{R}^n$ sind **äquivalent**, d.h. für zwei Normen $\|\cdot\|_\alpha$ und $\|\cdot\|_\beta$ gibt es stets Konstanten $c, C > 0$, sodass

$$c\,\|v\|_\alpha \leq \|v\|_\beta \leq C\,\|v\|_\alpha \quad \text{für alle } v \in \mathbb{R}^n \tag{6.10}$$

gilt. Mit Blick auf Fehlerabschätzungen beeinflussen die eingesetzten Normen also höchstens die auftretenden konstanten Faktoren, die ohnehin kaum relevant für die Kernaussagen sind.

Matrixnormen

Bei Konditionsuntersuchungen werden wir stets auf Vektoren der Form $A\,x$ und $A^{-1}\,b$ stoßen, die wir in Beziehung zu x bzw. b setzen möchten. Zu diesem Zweck haben sich **von einer Vektornorm induzierte Matrixnormen** etabliert.

Merke!

Für $A \in \mathbb{R}^{m\times n}$ und eine gegebene Vektornormen $\|\cdot\|_\alpha$ und $\|\cdot\|_\beta$ in $\mathbb{R}^m$ bzw. $\mathbb{R}^n$ ist durch

$$\|A\|_{\alpha,\beta} := \max_{v\in\mathbb{R}^n\setminus\{0\}} \frac{\|A\,v\|_\alpha}{\|v\|_\beta} \tag{6.11}$$

eine **Matrixnorm** gegeben, d.h. es sind die gleichen vier Eigenschaften wie bei Vektornormen erfüllt. Zusätzlich gilt

$$\|A\,v\|_\alpha \leq \|A\|_{\alpha,\beta}\,\|v\|_\beta \quad \text{für alle } v \in \mathbb{R}^m. \tag{6.12}$$

Am häufigsten anzutreffen ist die **Spektralnorm**, die entsteht, wenn für beide Vektornormen die euklidische Norm genutzt wird:

$$\|A\|_2 := \max_{v\in\mathbb{R}^n\setminus\{0\}} \frac{\|A\,v\|_2}{\|v\|_2}. \tag{6.13}$$

Merke!

Es gilt

$$\|A\|_2 := \sqrt{\lambda_{\max}}, \tag{6.14}$$

wobei $\lambda_{\max}$ der größte Eigenwert von $A^{\mathrm{T}}\,A$ ist (IDVID 610).

Die Spektralnorm ist von der gelegentlich auch verwendeten **Frobenius-Norm**

$$\|A\|_{\mathrm{F}} = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{i,j}^2 \tag{6.15}$$

zu unterscheiden, welche nicht durch Vektornormen induziert wird, aber dennoch die vier Normbedingungen erfüllt.

Die von der 1-Norm oder der ∞-Norm induzierten Matrixnormen spielen nur selten eine Rolle, sind aber deutlich einfacher zu berechnen als die Spektralnorm. Für die 1-Norm erhalten wir

$$\|A\|_1 = \max_{j \in \{1,\ldots,n\}} \sum_{i=1}^{m} |a_{i,j}| \tag{6.16}$$

(IDVID 613). Die ∞-Norm liefert

$$\|A\|_\infty = \max_{i \in \{1,\ldots,m\}} \sum_{j=1}^{n} |a_{i,j}| \tag{6.17}$$

(IDVID 617).

Analog zu den Vektornormen sind auch alle Matrixnormen zu einander **äquivalent**.

Fehlerhafte rechte Seite (Konditionszahl)

Betrachten wir nur b als Eingabe, so erhalten wir für den relativen Ausgabefehler bei fehlerhafteteter Eingabe $\tilde{b}$ die Abschätzung

$$\frac{\|A^{-1}\, b - A^{-1}\, \tilde{b}\|}{\|A^{-1}\, b\|} \leq \|A\|\, \|A^{-1}\|\, \frac{\|b - \tilde{b}\|}{\|b\|}, \tag{6.18}$$

wobei die verwendeten Matrix- und Vektornormen kompatibel zu einander sein sollen, ansonsten aber beliebig gewählt werden können (IDVID 620).

Beachte, dass beim bisherigen Konditionsbegriff für Abbildungen nach $\mathbb{R}$ (statt $\mathbb{R}^n$) der Eingabefehler gerade die 1-Norm des Vektors der komponentenweisen relativen Fehler war. Im Kontext linearer Gleichungssysteme ist jedoch die Norm des Vektors der absoluten Fehler geteilt durch die Norm des exakten Eingabevektors als Ausdruck des relativen Eingabefehlers besser handhabbar. Auf der Ausgabeseite wird die gleiche Variante zur Beschreibung des relativen Ausgabefehlers verwendet.

Merke!

Die Zahl

$$\kappa(A) := \|A\|\, \|A^{-1}\| \tag{6.19}$$

heißt **Konditionszahl** von A. Diese beschreibt die Auswirkungen des relativen Eingabefehlers auf den relativen Ausgabefehler sowohl bei der Multiplikation mit A als auch bei der Multiplikation mit A^{-1} (Gleichungssystem lösen).

Je nach Wahl der Matrixnorm entstehen unterschiedliche Zahlen $\kappa_1(A)$, $\kappa_2(A)$, $\kappa_\infty(A)$. Wenn im Folgenden kein Index angegeben ist, gelten die Resulte für alle Matrixnormen.

Es gilt stets

$$\kappa(A) = \frac{\max\limits_{\|x\|=1} \|A\,x\|}{\min\limits_{\|x\|=1} \|A\,x\|} \geq 1 \tag{6.20}$$

(IDVID 625).

Für die euklidische Matrixnorm kann man noch

$$\kappa_2(A) = \sqrt{\frac{\lambda_{\max}}{\lambda_{\min}}} \tag{6.21}$$

zeigen (tun wir aber nicht), wobei $\lambda_{\max}$ und $\lambda_{\min}$ der größte und der kleinste Eigenwert von A sind.

Fehlerhafte Matrix und rechte Seite

Untersuchen noch die Kondition der Abbildung $(A, b) \mapsto A^{-1}\,b$ bei fehlerbehafteter Matrix $\tilde{A}$ und fehlerbehafteter rechter Seite $\tilde{b}$. Hier muss zunächst geklärt werden, unter welchen Bedingungen $\tilde{A}$ überhaupt noch invertierbar ist.

Merke!

Ist $A \in \mathbb{R}^{n\times n}$ invertierbar und gilt

$$\|A - \tilde{A}\| < \frac{1}{\|A^{-1}\|}, \tag{6.22}$$

so ist auch $\tilde{A}$ invertierbar (IDVID 630).

Für den Fehlerzusammenhang zwischen Eingabe und Ausgabe erhält man dann

$$\frac{\|A^{-1}\,b - \tilde{A}^{-1}\,\tilde{b}\|}{\|A^{-1}\,b\|} \leq \frac{\kappa(A)}{1 - \frac{\|A-\tilde{A}\|}{\|A\|}\,\kappa(A)} \left(\frac{\|A - \tilde{A}\|}{\|A\|} + \frac{\|b - \tilde{b}\|}{\|b\|} \right) \tag{6.23}$$

(IDVID 635), wobei Matrix- und Vektornormen wieder kompatibel zu einander sein sollen. Bei kleinem Fehler in A und nicht zu großer Kondition $\kappa(A)$ beschreibt also auch hier $\kappa(A)$ die Fehlerverstärkung.

Beispiele

Einheitsmatrix Für

$$A = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \tag{6.24}$$

gilt $A = A^{-1}$ und $\|A\| = 1 = \|A^{-1}\|$, also $\kappa(A) = 1$.

Hilbert-Matrix Für

$$A = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n+1} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots & \frac{1}{n+2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \cdots & \frac{1}{2\,n-1} \end{bmatrix} \tag{6.25}$$

wächst $\kappa(A)$ exponentiell mit n, siehe The Condition of the Finite Segments of the Hilbert Matrix und Praktikum.

6.2 Wie es nicht geht

Für $A \in \mathbb{R}^{n \times n}$ kann die Lösung x des Gleichungssystems mit der Cramer'schen Regel ermittelt werden: Zur Berechnung von x_i sei A_i wie A, aber mit Spalte i ersetzt durch die rechte Seite b. Dann gilt

$$x_i = \frac{\det A_i}{\det A}, \tag{6.26}$$

wobei

$$\det A = \sum_{\sigma} (\operatorname{sgn} \sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}. \tag{6.27}$$

die Determinate von A ist (analog für A_i). Die Summe läuft hier über alle Permutationen $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$, also alle möglichen Anordnungen der Zahlen $1, \ldots, n$. Der Wert $\operatorname{sgn} \sigma$ ist das Vorzeichen der Permutation und ist entweder 1 oder -1.

Der Aufwand für das Berechnen von Determinanten ist enorm! Er liegt bei $n!$ Additionen und $(n-1)\,n!$ Multiplikationen pro Determinante. Für das Lösen eines Gleichungssystems werden $n + 1$ Determinanten benötigt. Für $n = 20$ würde ein 1-Peta-FLOPS-Rechner mehr als 11 Tage für das Lösen mittels Cramer'scher Regel benötigen.

Zusätzlich sind auf der Cramer'schen Regel beruhende Algorithmen durch die vielen Differenzen als instabil anzusehen.

Auch das explizite Berechnen der Inverse A^{-1}, um dann $x = A^{-1}\,b$ zu bekommen, ist im Allgemeinen aufwendiger als das Lösen des Gleichungssystems ohne Verwendung der

Inverse. Der Rechenaufwand für den unten behandelten Gauß-Algorithmus entspricht zwar dem des Invertierens (etwa n^3 Grundoperationen), der Gauß-Algorithmus ist aber stabiler. Praktisch auftretende Gleichungssysteme haben meist beim Lösen vorteilhaft nutzbare Zusatzeigenschaften (symmetrisch, dünn besetzt, Bandstruktur,...), die den Einsatz schnellerer und speichersparenderer Algorithmen zulassen. So ist beispielsweise die im Allgemeinen vollbesetzte Inverse von großen Tridiagonalmatrizen, wie sie häufig in der Praxis auftreten (vgl. Veranstaltung zum wissenschaftlichen Rechnen), viel zu groß um sie überhaupt im Arbeitsspeicher ablegen zu können.

Grundsätzlich gilt: Inverse verweiden. Ausnahmen sind sehr kleine, gut konditionierte Gleichungssysteme, die für viele verschiedene rechte Seiten zu lösen sind (z.B. Koordinatentransformationen in der Computergrafik).

6.3 Gauß-Algorithmus

Idee

Merke!

Sind alle Diagonaleinträge von $A \in \mathbb{R}^{n\times n}$ von Null verschieden, so existieren eine untere Dreiecksmatrix L mit Einsen auf der Diagonale und eine obere Dreiecksmatrix R, sodass

$$A = L\,R. \tag{6.28}$$

Die Zerlegung von A in solche Faktoren L und R ist eindeutig (IDVID 640).

Die LR-Zerlegung von A kann mittels Gauß-Algorithmus (siehe unten) bestimmt werden. Liegt diese vor, so erfolgt das Lösen des Gleichungssystems durch **Vorwärtssubstitution** und anschließende **Rückwärtssubstitution**:

Algorithmus

1. Löse $L\,y = b$; dazu setze $y_1 := b_1$ und wiederhole für $i = 2, \ldots, n$:

   a) $y_i := b_i - \sum_{j=1}^{i-1} l_{i,j}\,y_j$.

2. Löse $R\,x = y$; dazu setze $x_n := \frac{y_n}{r_{n,n}}$ und wiederhole für $i = n-1, n-2, \ldots, 1$:

   a) $x_i := \frac{1}{r_{i,i}} \left(y_i - \sum_{j=i+1}^{n} r_{i,j}\,x_j \right)$.

LR-Zerlegung

Sei $E_{i,j} \in \mathbb{R}^{n\times n}$ die Matrix, die in Zeile i und Spalte j eine 1 hat und sonst nur Nullen, und sei $I \in \mathbb{R}^{n\times n}$ die Einheitsmatrix. Für $c \in \mathbb{R}$ setzen wir

$$M_{i,j}(c) := I + c\,E_{i,j}. \tag{6.29}$$

Dann entsteht das Produkt $M_{i,j}(c)\,A$ aus A indem Zeile j von A mit c multipliziert und zu Zeile i von A addiert wird. Umgekehrt wird bei $A\,M_{i,j}(c)$ gerade Spalte i mit c multipliziert und zu Spalte j addiert. Offensichtlich gilt $M_{i,j}(c)^{-1} = M_{i,j}(-c)$.

Mit diesen Matrizen beschreiben wir nun den Algorithmus zum Erhalt der LR-Zerlegung. Dieser erzeugt von der Matrix A ausgehend spaltenweise Nullen unterhalb der Diagonale, indem geeignete Vielfache einer Zeile von den darunter liegenden Zeilen abgezogen werden:

Algorithmus

1. Setze $L := I$ und $R := A$.
2. Wiederhole für $j = 1, \ldots, n-1$:
   a) Wiederhole für $i = j+1, \ldots, n$:
      i. Setze $c := \frac{r_{i,j}}{r_{j,j}}$.
      ii. Setze $L := L\,M_{i,j}(c)$.
      iii. Setze $R := M_{i,j}(-c)\,R$.

Das Produkt $L\,R$ verändert sich im Laufe der Iteration nicht, bleibt also beim initialen $L\,R = A$. Die Zahlen c sind gerade so gewählt, dass R zu einer oberen Dreiecksmatrix wird. Wegen $i > j$ bleiben die Diagonaleinträge und das obere Dreieck von L unverändert.

Stabilität

Ist $\tilde{x}$ die mittels Gauß-Algorithmus aus exakten Eingaben A und b numerisch ermittelte und somit (rundungs-)fehlerbehaftete Lösung, so haben wir im Sinne der Rückwärtsanalyse

$$\tilde{x} = A^{-1}\,\hat{b} \quad \text{mit } \hat{b} = A\,\tilde{x}. \tag{6.30}$$

Liegt der relative Fehler zwischen b und $A\,\tilde{x}$ im Bereich des erwarteten Eingabefehlers, so ist der Algorithmus stabil. Die Differenz $b - A\tilde{x}$ heißt auch **Residuum**.

Wie groß das Residuum ist, ist allgemein kaum zu bestimmen. Bei schlecht konditioniertem A wird es jedenfalls groß sein, da die Rundungsfehler aus den ersten Rechenschritten

analog zu Eingabefehlern verstärkt werden. Der Gauss-Algorithmus ist somit nicht als stabil anzusehen.

Ein alternative Analyse, die eine Darstellung $\tilde{x} = \hat{A}^{-1}\, b$ nutzt, findet man in [Sau78] (Satz 4.5). Auch die dort erzielte Abschätzung für den Unterschied zwischen A und $\hat{A}$ liefert nicht in jedem Fall Stabilität.

Folgende Punkte fassen die Sachlage zusammen:

- Theoretisch ist der Gauss-Algorithmus instabil.
- Für gewisse Problemklassen (z.B. Tridiagonalmatrizen) ist er garantiert stabil.
- Praktisch tritt Instabilität nur sehr selten auf, sofern Pivotsuche eingesetzt wird (siehe unten).
- Für nicht zu große Gleichungssysteme ohne spezielle Struktur ist der Gauß-Algorithmus heute das Standardlösungsverfahren.

Zeit- und Speicherbedarf

Rechenaufwand für die LR-Zerlegung: ca. n^3 Grundoperationen (IDVID 645).

Rechenaufwand für Vorwärts- und Rückwärtssubstitution: ca. n^2 Grundoperationen.

Der Rechenaufwand ist für große Gleichungssysteme also relativ hoch. Ist ein System für mehrere rechte Seiten zu lösen, so muss die LR-Zerlegung allerdings nur einmal berechnet werden. Das Lösen der weiteren Systeme benötigt dann nur noch n^2 Operationen.

Beachte: Die LR-Zerlegung kann in einer Matrix der Größe $n \times n$ gespeichert werden, da von L nur das untere Dreieck ohne Diagonalelemente (alle 1) benötigt wird. Diese Matrix kann den Speicherbereich von A nutzen, da R schrittweise aus A entsteht. Die Nicht-Null-Einträge von L sind in jedem Schritt gerade dort, wo R Nullen enthält. Sofern die Matrix A nicht mehr benötigt wird, also überschrieben werden kann, benötigt der Gauß-Algorithmus also keinen zusätzlichen Speicher.

Pivotsuche

Der Gauß-Algorithmus in der bisher vorgestellten Form hat zwei Schwächen:

- Alle Diagonalelemente von A müssen von Null verschieden sein.
- Die Division durch $r_{j,j}$ kann bei kleinem $r_{j,j}$ zu Instabilität führen.

Beide Probleme kann man relativ leicht beheben. Zunächst ein Beispiel für die Instabilität: Rechen mit dezimalen Gleitkommazahlen mit zweistelliger Mantisse und lösen

$$\begin{bmatrix} 1.0 \cdot 10^{-3} & 1.0 \\ 1.0 & 1.0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix}. \tag{6.31}$$

Bei exaktem Rechnen erhält man $x_1 = 1.\overline{001}$ und $x_2 = 0.\overline{998}$, gerundet auf das genutzte Zahlenformat also $x_1 = 1.0$ und $x_2 = 1.0$. Führt man den Gauß-Algorithmus mit Runden nach jeder Operations aus, so liefert dieser $x_1 = 0.0$ und $x_2 = 1.0$ (IDVID 650), also eine grob falsche Lösung. Führt man den Algorithmus jedoch mit vertauschten Zeilen aus, so stimmt die numerische Lösung mit der gerundeten exakten Lösung überein.

Bei Gleichungssystemen ist die Reihenfolge der Gleichungen egal. Man kann also die Zeilen der Systemmatrix A beliebig anordnen, solange man die Reihenfolge der Einträge in der rechten Seite b entsprechend anpasst. Diese Freiheit nutzen wir zur Lösung der beiden genannten Probleme. Vor jeder neuen Iteration der äußeren Schleife bei der LR-Zerlegung sortieren wir die verbleibenden Zeilen so, dass $r_{j,j}$ (das sogenannte **Pivotelement**) so groß wie möglich wird. Insbesondere ist das Pivotelement dann nie Null (sonst wäre das Gleichungssystem nicht lösbar). Aus Sicht der Stabilität ist dieses Vorgehen vorteilhaft, weil dann die Faktoren $\frac{r_{i,j}}{r_{j,j}}$ klein sind, also Rundungsfehler aus den vorhergehenden Schritten so wenig wie möglich verstärkt werden.

Dieses Vorgehen wird **Spaltenpivotsuche** genannt, da nur innerhalb einer Spalte gesucht wird. Der zusätzliche Rechenaufwand liegt in der Größenordnung n^2 (n-mal suchen in höchstens n Einträgen). Zur weiteren Erhöhung der Stabilität kann auch spaltenübergreifend gesucht werden. Dann müssen ggf. Spalten von A vertauscht werden, was einem Verändern der Reihenfolge der Einträge in x entspricht. Der zusätzliche Rechenaufwand steigt durch den größeren Suchbereich jedoch auf ca. n^3, weshalb diese **vollständige Pivotsuche** kaum eingesetzt wird.

Implementierung

Den Gauß-Algorithmus sollte man (heute) nicht selbst implementieren, da hochwertige Implementierungen in allen gängigen Software-Biliotheken verfügbar sind. Meistens basieren diese auf LAPACK, einer sehr ausgereiften Bibliothek für lineare Algebra, die in für verschiedene Prozessorarchitekturen optimierten Versionen vorliegt.

6.4 Cholesky-Verfahren

Für positiv definite symmetrische Matrizen $A \in \mathbb{R}^{n\times n}$ kann die LR-Zerlegung einfacher ausgedrückt und effizienter berechnet werden.

Positiv definite Matrizen

Eine symmetrische Matrix heißt **positiv definit**, wenn gilt:

$$v^{\mathrm{T}} A\, v > 0 \quad \text{für alle } v \in \mathbb{R}^n \setminus \{0\}. \tag{6.32}$$

Positiv definite symmetrische Matrizen haben viele vorteilhafte Eigenschaften (siehe Grundlagenliteratur zur linearen Algebra):

- A ist stets invertierbar und A^{-1} ist wieder symmetrisch und positiv definit.
- Alle Eigenwerte von A sind reell und positiv.
- $\det A > 0$.
- Alle Diagonaleinträge sind positiv.
- Der betragsgrößte Eintrag von A liegt auf der Diagonale.

Positiv definite symmetrische Matrizen treten in zahlreichen Anwendungen auf (vgl. Veranstaltung zum wissenschaftlichen Rechnen). Beispielsweise sind Matrizen der Form $A = B^{\mathrm{T}} B$ für invertierbares $B \in \mathbb{R}^{n,n}$ stets symmetrisch und positiv definit (IDVID 655). Solche Matrizen treten unter anderem beim Berechnen von Skalarprodukten in transformierten Koordinaten auf. B ist dabei die Koordinatentransformation (vgl. Hauptkomponentenanalyse ("PCA") in den Datenwissenschaften).

Cholesky-Zerlegung

Merke!

Für positiv definites symmetrisches $A \in \mathbb{R}^{n\times n}$ existieren stets eine untere Dreiecksmatrix $L \in \mathbb{R}^{n\times n}$ mit Einsen auf der Diagonale und eine Diagonalmatrix $D \in \mathbb{R}^{n\times n}$ mit positiven Einträgen auf der Diagonale, sodass

$$A = L\,D\,L^{\mathrm{T}}. \tag{6.33}$$

L und D sind eindeutig bestimmt.

Diese sogenannte **Cholesky-Zerlegung** erhält man direkt aus der LR-Zerlegung, wenn man die Symmetrie $A = A^{\mathrm{T}}$ ausnutzt. Insbesondere gilt $R = D\,L^{\mathrm{T}}$ (IDVID 660).

Allerdings geht es auf direktem Wege etwas schneller. Betrachten dazu die Einträge von

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ l_{2,1} & 1 & 0 & \cdots & 0 & 0 \\ l_{3,1} & l_{3,2} & 1 & & 0 & 0 \\ \vdots & \vdots & & \ddots & & \vdots \\ l_{n-1,1} & l_{n-1,2} & l_{n-1,3} & & 1 & 0 \\ l_{n,1} & l_{n,2} & l_{n,3} & \cdots & l_{n,n-1} & 1 \end{bmatrix} \tag{6.34}$$

und von

$$D = \begin{bmatrix} d_{1,1} & 0 & \cdots & 0 & 0 \\ 0 & d_{2,2} & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & d_{n-1,n-1} & 0 \\ 0 & 0 & \cdots & 0 & d_{n,n} \end{bmatrix} \tag{6.35}$$

als Unbekannte und lösen das nichtlineare Gleichungssystem

$$A = L\,D\,L^{\mathrm{T}}. \tag{6.36}$$

Dieses kann explizit gelöst werden (IDVID 665). Man erhält den folgenden Lösungsalgorithmus:

Algorithmus

1. Setze $d_{1,1} := a_{1,1}$ und $l_{2,1} := \frac{a_{2,1}}{d_{1,1}}$.
2. Für $i = 2, \ldots, n$ wiederhole:
   a) Setze $l_{i,1} := \frac{a_{i,1}}{d_{1,1}}$.
3. Für $j = 2, \ldots, n$ wiederhole:
   a) Setze

$$d_{j,j} := a_{j,j} - \sum_{k=1}^{j-1} l_{j,k}^2\, d_{k,k}. \tag{6.37}$$

   b) Für $i = j+1, \ldots, n$ wiederhole:
      i. Setze

$$l_{i,j} := \frac{1}{d_{j,j}} \left(a_{i,j} - \sum_{k=1}^{j-1} l_{i,k}\, l_{j,k}\, d_{k,k} \right). \tag{6.38}$$

Zeit- und Speicherbedarf

Der Rechenaufwand liegt wie bei der LR-Zerlegung in der Größenordnung n^3, ist aber trotzdem nur etwa halb so groß oder, bei Zwischenspeichern der Produkte $l_{j,k}\,d_{k,k}$, sogar nur ein Viertel so groß (IDVID 670).

Der Speicherplatz des unteren Dreiecks von A kann beim Berechnen der Zerlegung mit den Einträgen von L überschrieben werden, da alle Einträge von A nochmal im oberen Dreieck vorhanden sind. Die Einträge von D können auf die entsprechenden Diagonaleinträge von A geschrieben werden, da die Diagonalelemente von A im weiteren Verlauf nicht mehr benötigt werden. Somit benötigt das Berechnen der Cholesky-Zerlegung keinen zusätzlichen Speicherplatz. Lediglich für das eventuelle Zwischenspeichern der Produkte $l_{j,k}\,d_{k,k}$ wird Speicherplatz in der Größenordnung n^2 benötigt.

Gesamtalgorithmus

Ist die Cholesky-Zerlegung bekannt, kann das Gleichungssystem $A\,x = b$ analog zur LR-Zerlegung mittels Vorwärts- und anschließender Rückwärtssubstitution gelöst werden:

Algorithmus

1. Löse $L\,y = b$; dazu setze $y_1 := b_1$ und wiederhole für $i = 2, \ldots, n$:

   a) $y_i := b_i - \sum_{j=1}^{i-1} l_{i,j}\, y_j$.

2. Löse $D\,z = y$; dazu wiederhole für $i = 1, \ldots, n$:

   a) $z_i := \frac{y_i}{d_{i,i}}$.

3. Löse $L^{\mathrm{T}}\,x = z$; dazu setze $x_n := z_n$ und wiederhole für $i = n-1, n-2, \ldots, 1$:

   a) $x_i := z_i - \sum_{j=i+1}^{n} l_{j,i}\, x_j$.

Stabilität

Das Lösen von linearen Gleichungssystemen mittels Cholesky-Zerlegung ist rückwärtsstabil. Ist $\tilde{x}$ die aus dem Cholesky-Verfahren resultierende (rundungsfehlerbehaftete) Lösung, so findet man eine Matrix $\hat{A}$ mit $\tilde{x} = \hat{A}^{-1}\,b$. Für diese Matrix kann man

$$\frac{\|A - \hat{A}\|_2}{\|A\|_2} \leq c_n\,\varepsilon \tag{6.39}$$

zeigen, wobei ε das Maschinenepsilon ist. Die Konstante c_n ist meist klein und liegt nur in Ausnahmefällen in der Nähe der garantierten oberen Schranke $c_n \leq 4\,n\,(3\,n+1)$ (siehe [Hig02] für Herleitung und Details).

Insbesondere gilt das Cholesky-Verfahren als stabiler als der Gauss-Algorithmus. Ein günstiges Vertauschen von Zeilen im Vorfeld zur Verbesserung der Stabilität wie beim Gauss-Algorithmus ist beim Cholesky-Verfahren nicht zweckmäßig, da dadurch die Symmetrie der Matrix verloren gehen würde.

6.5 Weitere Verfahren

Wir erwähnen kurz weitere Verfahrensideen um bei Bedarf mit den Begrifflichkeiten vertraut zu sein.

QR-Zerlegung

Die LR-Zerlegung steht nur für quadratische Matrizen zur Verfügung. Für allgemeine rechteckige Matrizen $A \in \mathbb{R}^{m \times n}$, $m \geq n$ (überbestimmtes Gleichungssystem) kann man jedoch stets die sogenannte QR-Zerlegung $A = Q\,R$ finden. Dabei ist $R \in \mathbb{R}^{m \times n}$ wieder eine obere Dreiecksmatrix, wobei die unteren $m - n$ Zeilen mit Nullen aufgefüllt werden, und $Q \in \mathbb{R}^{m \times m}$ eine orthogonale Matrix. Die Spalten von Q sind also orthogonal zu einander und haben Norm 1. Für orthogonale Matrizen gilt stets $Q^{-1} = Q^{\mathrm{T}}$, sodass sich das Lösen des Gleichungssystems $A\,x = b$ auf Rückwärtseinsetzen und Multiplikation mit Q^{T} reduziert.

Für quadratische Matrizen ($m = n$) kann man die QR-Zerlegung mit etwa n^3 Grundoperationen berechnen.

Iterative Verfahren

Iterative Verfahren liefern im Gegensatz zum Gauß-Algorithmus und zum Cholesky-Verfahren nicht die (bis auf Rundungsfehler) exakte Lösung des Gleichungssystems $A\,x = b$, sondern nur Näherungslösungen. Die Verfahren berechnen zunächst eine sehr grobe Näherung, die dann Schritt für Schritt verfeinert wird. Dieser Ansatz erlaubt deutlich geringere Rechenzeiten bei sehr großen Gleichungssystemen.

Richardson-Iteration Für die Lösung x des Gleichungssystems und eine beliebige reelle Zahle $\tau > 0$ gilt

$$x = x - \tau\,(A\,x - b) = (I - \tau\,A)\,x + \tau\,b. \tag{6.40}$$

Dies ist eine sogenannte **Fixpunktgleichung**, da das Auswerten der rechten Seite wieder auf x führt. Daraus kann man die Iterationsvorschrift

$$x_k := (I - \tau\,A)\,x_{k-1} + \tau\,b \tag{6.41}$$

ableiten. Man kann zeigen, dass für hinreichend kleines τ die Folge $x_0 := 0, x_1, x_2, \ldots$ gegen die Lösung des Gleichungssystems konvergiert. Je nach gewünschter Genauigkeit kann die Iteration früher oder später abgebrochen werden.

Die Iterationsvorschrift kann zu

$$x_k := M\,x_{k-1} + c \tag{6.42}$$

verallgemeinert werden, wobei $M \in \mathbb{R}^{n \times n}$ und $c \in \mathbb{R}^n$ von A und b abhängen. So erhält man eine Vielzahl verschiedener Lösungsverfahren, die je nach Eigenschaften von A vorteilhafte Eigenschaften wie beispielsweise besonders schnelle Konvergenz haben können. Die Konvergenz der Iterierten zur exakten Lösung ist dabei keinesfalls garantiert, sondern kann nur unter recht engen Voraussetzungen an M und c abgesichert werden.

CG-Verfahren Für das Verfahren der konjugierten Gradienten (kurz: CG-Verfahren, CG = conjugate gradients) wird das Gleichungssystem durch das Minimierungsproblem

$$J(x) := \frac{1}{2}\, x^{\mathrm{T}}\, A\, x - x^{\mathrm{T}}\, b \to \min_{x \in \mathbb{R}^n} \tag{6.43}$$

ersetzt. Man kann zeigen, dass die Lösung des Minimierungsproblems mit der Lösung des Gleichungssystems übereinstimmt.

Zur Lösung des Minimierungsproblems stehen eine Vielzahl numerischer Optimierungsverfahren zur Verfügung. Das einfachste ist das so genannte Gradienten-Verfahren, welches, beginnend an einem Punkt x_0, den Gradient ∇J der Zielfunktion J berechnet und dann die aktuelle Position in Richtung des negativen Gradienten (Richtung des steilsten Abstiegs!) verändert:

$$x_k := x_{k-1} - s_k\, \nabla J(x_{k-1}). \tag{6.44}$$

Dabei ist s_k die Schrittweite, welche auf verschiedene Arten gewählt werden kann, z.B. konstant. Für den Gradient erhält man

$$\nabla J(x) = A\, x - b. \tag{6.45}$$

Wählt man s_k stets so, dass $J(x_{k-1} - s\, \nabla J(x_{k-1}))$ für $s = s_k$ minimal wird, so erhält man das CG-Verfahren, welches besonders schnell konvergiert. Für konstantes s_k erhält man hingegen das Verfahren des steilsten Abstiegs, welches verhältnismäßig langsam konvergiert.

Verfahren für dünn besetzte Matrizen In zahlreichen Anwendungen treten dünn besetzte (englisch: sparse) sehr große Matrizen $A \in \mathbb{R}^{n \times n}$ auf, also Matrizen, die nur wenige von Null verschiedene Einträge haben. Die Anzahl der Nicht-Null-Einträge ist üblicherweise ein kleines Vielfaches von n. Diese Matrizen speichert man nicht als Block aus n^2 Zahlen, sondern als Liste der Nicht-Null-Einträge und der zugehörigen Zeilen- und Spalten-Indizes.

Beispielsweise sind Adjazenzmatrizen von Graphen oft dünnbesetzt (und groß). Auch beim numerischen Lösen vieler naturwissenschaftlicher Probleme treten große dünnbesetzte Matrizen auf (vgl. Veranstaltung zum wissenschaftlichen Rechnen).

Der Zugriff auf einen durch Zeilen- und Spaltenindex gegebenen Matrixeintrag ist bei dünn besetzten Matrizen mit hohem Aufwand verbunden (Liste durchsuchen!). Matrix-Vektor-Produkte können jedoch schnell berechnet werden (IDVID 680). Somit sind iterative Lösungsverfahren gegenüber Gauß-Algorithmus und Cholesky-Verfahren hier klar im Vorteil, da iterative Verfahren meist nur Matrix-Vektor-Produkte auswerten und keine anderweitigen Zugriffe auf die Matrixeinträge tätigen.

Heute übliche Verfahren für große dünn besetzte Gleichungssysteme sind Krylow-Unterraum-Verfahren (Verallgemeinerung des CG-Verfahrens) und Mehrgitterverfahren.

Vorkonditionierung

Die Konvergenz iterativer Lösungsverfahren kann deutlich beschleunigt werden, wenn das Gleichungssystem im Vorfeld geeignet äquivalent umgeformt wird. Auch für nicht iterative Verfahren kann eine geeignete Umformung zur Verbesserung der Matrixkondition führen. Eine übliche Umformung ist die Multiplikation mit einer geeignet gewählten Matrix $M \in \mathbb{R}^{n \times n}$:

$$M\,A\,x = M\,b. \tag{6.46}$$

Diese Matrix soll leicht zu beschaffen sein und dabei die Inverse A^{-1} möglichst gut annähern. Wie M konkret zu wählen ist, hängt auch vom eingesetzten Lösungsverfahren ab.

Eine einfache Technik zur Vorkonditionierung ist die **Äquilibrierung**: Jede Zeile des Gleichungssystems wird durch die Norm der entsprechenden Zeile der Systemmatrix als Vektor geteilt. Bei Verwendung der euklidischen Norm also

$$M = \begin{bmatrix} \left(\sum_{j=1}^{n} a_{1,j}^2\right)^{-\frac{1}{2}} & & 0 \\ & \ddots & \\ 0 & & \left(\sum_{j=1}^{n} a_{n,j}^2\right)^{-\frac{1}{2}} \end{bmatrix}. \tag{6.47}$$

7 Ausgleichsverfahren

Ausgleichsverfahren erzeugen ein Modell für den Zusammenhang zwischen gemessenen Größen. Obwohl nur endlich viele punktuelle Messungen vorliegen, beschreibt das Modell den Zusammenhang lückenlos. Messfehler und Widersprüche in den Messwerten werden bei der Erzeugung des Modells ausgeglichen.

Das wichtigeste Ausgleichsverfahren ist die Methode der kleinsten Quadrate (siehe unten). Ausgleichsverfahren kommen immer dann zum Einsatz, wenn die darstellung eines funktionalen Zusammenhangs durch eine endliche Anzahl von Punkte nicht genügt, z.B. bei

- der Auswertung des funktionalen Zusammenhangs an beliebigen Zwischenstellen,
- der Berechnung von Flächeninhalten unter dem Funktionsgraph (numerische Integration, siehe späteres Kapitel),
- der Visualisierung von funktionalen Zusammenhängen.

Im Kontext des maschinellen Lernens werden Ausgleichsverfahren als Regressionsverfahren bezeichnet.

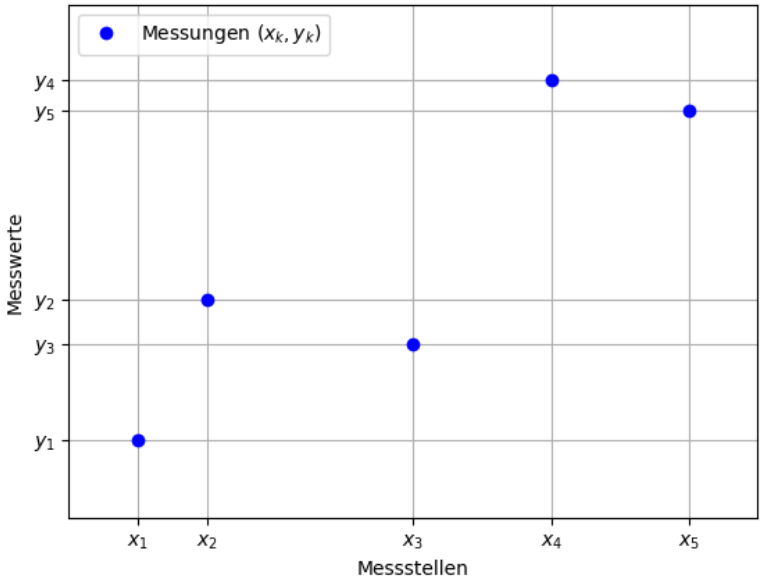
7.1 Zu lösendes Problem

Der Zusammenhang zwischen zwei Größen x und y soll aus n Messungen $(x_1, y_1), \ldots, (x_n, y_n)$ ermittelt werden. Dabei sind die Messstellen x_k vorgegeben oder praktisch fehlerfrei gemessen. Die von den x_k abhängenden Werte y_k können Messfehler aufweisen.

Im Allgemeinen wird man deutlich mehr Messwerte zur Verfügung haben als (bei fehlerfreier Messung) eigentlich benötigt.

Beispiele:

- x_k sind Zeitpunkte zu denen Temperaturen y_k gemessen werden.
- x_k sind Positionen entlang einer Strecke an denen Höhen y_k gemessen werden.



Aus dem Kontext der Messaufgabe ist der grundlegende funktionale Zusammenhang $f : \mathbb{R} \to \mathbb{R}$ zwischen den Größen x und y bekannt. Es müssen nur noch einige Parameter bestimmt werden, sodass f die Messwerte gut repräsentiert. Beachte: Die Funktion f kann auch nur auf einem Intervall von Interesse sein, was am Verfahren aber nichts ändert.

Beispiele:

- Die Temperatur steigt linear mit der Zeit (Gerade, 2 Parameter).
- Das Höhenprofil hat keine starken Gefälle oder Steigungen und kann durch eine stückweise lineare Funktion mit 20 Stützstellen hinreichend genau beschrieben werden (20 Parameter).

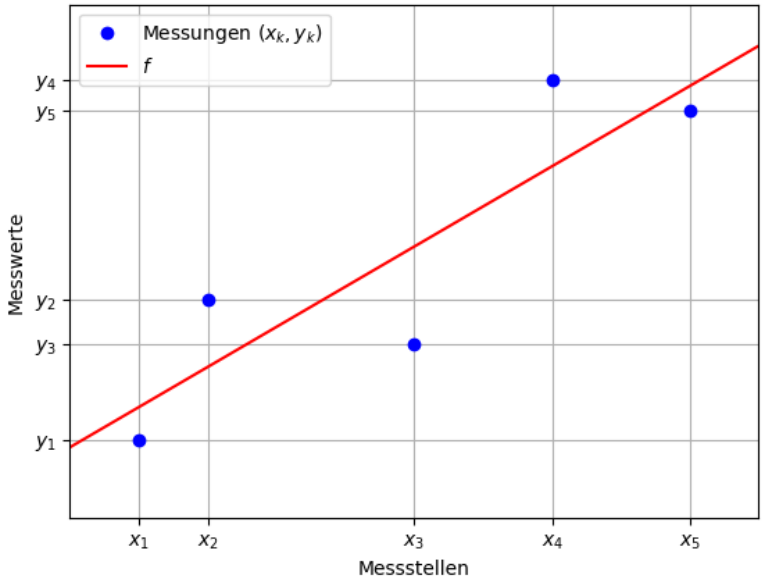
Merke!

Gegeben:

- Messstellen $x_1, \ldots, x_n$
- Messwerte $y_1, \ldots, y_n$
- Ansatzfunktion f, die noch zu bestimmende Parameter enthält

Gesucht:

- Werte für die Parameter in f, sodass f gut zu den Messergebnissen passt



7.2 Wie Ansatzfunktion wählen?

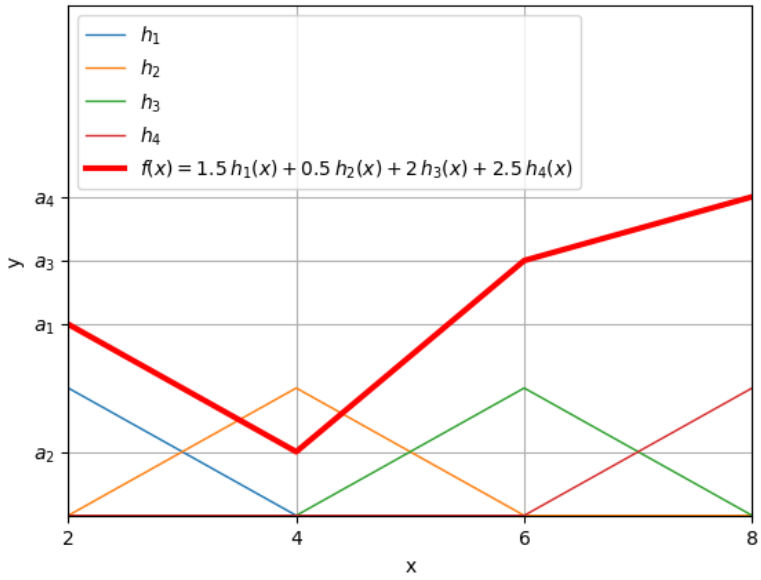
Das hängt vom Kontext ab! Oft ergibt sich die Ansatzfunktion aus physikalischen Gesetzen oder aus der verfolgten Zielstellung heraus.

Übliche Ansätze für f:

- Gerade $f(x) = a\,x + b$
- Parabel $f(x) = a\,x^2 + b\,x + c$
- Polynom höheren Grades
- stückweise linear Funktion mit m gleichmäßig verteilten Stützstellen und Werten $a_1, \ldots, a_m$ an diesen Stellen
- Summen von verschobenen Hütchenfunktionen:

$$f(x) = \sum_{l=1}^{m} a_l\, h_l(x),$$

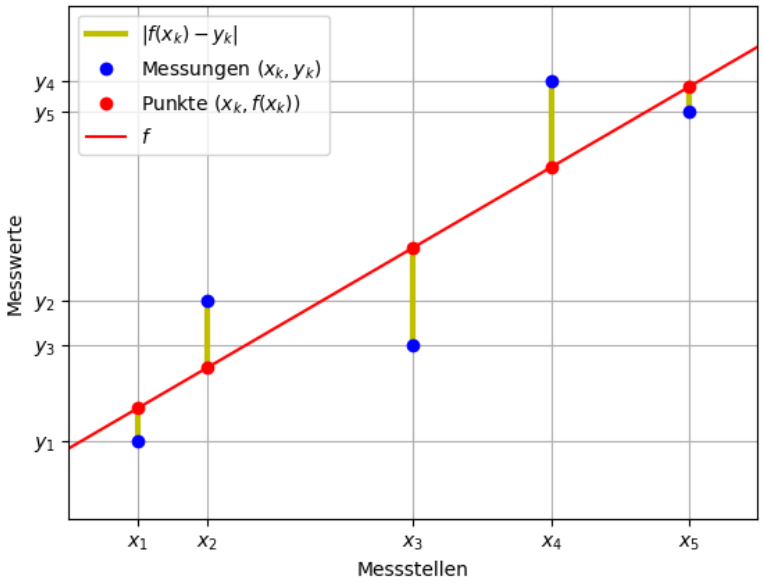
wobei $h_1, \ldots, h_m$ verschobene Kopien einer Funktion h sind



7.3 Was heißt „gut“?

Es gibt viele verschiedene richtige Antworten auf diese Frage!

Grundsätzlich möchte man die Differenzen $f(x_k) - y_k$ zwischen f an den Messstellen und den zugehörigen Messwerten für alle k klein bekommen.



Variante 1 (Summe aller Fehler minimieren)

Minimiere

$$\sum_{k=1}^{n} |f(x_k) - y_k|$$

bezüglich der Parameter in f.

Ist f beispielsweise eine Gerade, so müssen wir

$$\sum_{k=1}^{n} |a\,x_k + b - y_k|$$

bezüglich a und b minimieren.

Probleme:

- Funktion nicht differenzierbar (aber es gibt numerische Verfahren dafür, siehe Subgradient method)
- Lösung oft nicht eindeutig
- kleine Änderungen in den Messpunkten können zu großen Änderungen in der Lösung führen (schlechte Kondition)

Für Details siehe Least absolute deviations.

Variante 2 (maximalen Fehler minimieren)

Minimiere

$$\max_{k=1,\ldots,n} |f(x_k) - y_k|$$

bezüglich der Parameter in f.

Probleme:

- Funktion nicht differenzierbar (und es gibt keine wirklich guten numerischen Verfahren dafür)

Variante 3 (Summe aller Fehlerquadrate minimieren)

„Rund-Lutschen“ von Variante 1 liefert Minimierung von

$$\sum_{k=1}^{n} \big(f(x_k) - y_k\big)^2$$

bezüglich der Parameter in f.

Das ist differenzierbar und hat praktisch immer einen eindeutigen Minimierer!

Für die Eindeutigkeit werden mehr Messstellen benötigt als f Parameter hat.

Diese Variante zum Lösen des Ausgleichsproblems ist als **Methode der kleinesten Quadrate** bekannt. Das Verfahren wurde von Carl Friedrich Gauß für Anwendungen in der Astronomie entwickelt und von ihm auch bei der Vermessung des Königreichs Hannover angewendet. Heute ist die Methode der kleinsten Quadrate in praktisch allen Wissenschaftszweigen im Einsatz, in denen Erkenntnisse aus Messungen gewonnen werden.

7.4 Warum Quadrate?

Warum nicht dritte oder vierte Potenzen?

Einfache Gründe:

- Die Ableitungen werden schön einfach.
- $p = 2$ ist die kleinste Potenz, sodass $t \mapsto t^p$ zweimal differenzierbar ist (hinreichende Optimalitätsbedingung!).
- Je höher die Potenz, desto weniger Einfluss haben kleine Messfehler auf die Zielfunktion. Die gesuchte Funktion f orientiert sich bei großen Potenzen im wesentlichen an Ausreißern in den Messungen.

Tiefer liegende Gründe:

- Betrachtet man die Abweichungen nicht einzeln an jeder Stelle x_k, sondern interpretiert sie als Abweichung zwischen den beiden Vektoren $[f(x_1), \dots, f(x_n)]^{\mathrm{T}}$ und $[y_1, \dots, y_n]^{\mathrm{T}}$, so ist die Zielfunktion mit Potenz 2 gerade der übliche Abstand zwischen Vektoren im $\mathbb{R}^n$.
- Man kann zeigen, dass die Kleineste-Quadrate-Lösung in gewissem Sinn die bestmögliche Lösung ist, wenn die Messfehler gewisse statistische Eigenschaften haben (siehe Satz von Gauß-Markow für Details).

7.5 Lösung des Minimierungsproblems

Lösen das Kleinste-Quadrate-Problem hier nur beispielhaft für eine Gerade $f(x) = a\,x + b$. Für andere Ansätze f erfolgen die Rechenschritte völlig analog, solange f linear von den Parametern abhängt (siehe unten für den nichtlinearen Fall).

Die allgemeine Lösung wird anschließend ohne Rechenweg angeben.

Ausgleichsgerade

Gegeben sind Punkte (x_k, y_k) für $k = 1, \dots, n$. Gesucht ist der Minimierer von

$$g(a,b) := \sum_{k=1}^{n} \big(a\,x_k + b - y_k\big)^2$$

bezüglich $a, b \in \mathbb{R}$.

Gradient von g berechnen:

$$\begin{aligned}
\nabla g(a,b) &= \begin{bmatrix} 2\sum_{k=1}^{n} x_k \left(a\,x_k + b - y_k\right) \\ 2\sum_{k=1}^{n} \big(a\,x_k + b - y_k\big) \end{bmatrix} \\
&= \begin{bmatrix} 2\,a\sum_{k=1}^{n} x_k^2 + 2\,b\sum_{k=1}^{n} x_k - 2\sum_{k=1}^{n} x_k\,y_k \\ 2\,a\sum_{k=1}^{n} x_k + 2\,b\sum_{k=1}^{n} 1 - 2\sum_{k=1}^{n} y_k \end{bmatrix} \\
&= \begin{bmatrix} 2\,c_1\,a + 2\,c_2\,b - 2\,c_3 \\ 2\,c_2\,a + 2\,n\,b - 2\,c_4 \end{bmatrix},
\end{aligned} \tag{7.1}$$

wobei

$$c_1 := \sum_{k=1}^{n} x_k^2, \tag{7.2}$$

$$c_2 := \sum_{k=1}^{n} x_k, \tag{7.3}$$

$$c_3 := \sum_{k=1}^{n} x_k\,y_k, \tag{7.4}$$

$$c_4 := \sum_{k=1}^{n} y_k \tag{7.5}$$

konkrete Zahlen sind.

Der Gradient wird genau dann zum Nullvektor, wenn das lineare Gleichungssystem

$$\begin{bmatrix} c_1 & c_2 \\ c_2 & n \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} c_3 \\ c_4 \end{bmatrix}$$

erfüllt ist.

Man kann zeigen, dass die Lösung des Gleichungssystems nicht nur ein stationärer Punkt, sondern ein globales Minimum von g ist.

Beispiel

gegeben: Messpunkte $(1, 2.3)$, $(2, 6.4)$, $(5, 5.1)$, $(7, 12.8)$, $(9, 11.9)$

gesucht: Parameter a, b der Gerade

Das Gleichungssystem

$$\begin{bmatrix} 160 & 24 \\ 24 & 5 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 237.3 \\ 38.5 \end{bmatrix}$$

hat die Lösung

$$a = 1.171875, \qquad b = 2.075$$

Die gesuchte Ausgleichsgerade ist also

$$f(x) = 1.171875\, x + 2.075.$$

Allgemeine Lösung

Hat die gesuchte Ausgleichsfunktion die Form

$$f(x) = \sum_{l=1}^{m} a_l\, h_l(x),$$

wobei $h_1, \ldots, h_m$ beliebige Funktionen sind (nicht notwendig verschobene Hütchen), so soll die Funktion

$$g(a_1, \ldots, a_m) := \sum_{k=1}^{n} \left(\sum_{l=1}^{m} a_l\, h_l(x_k) - y_k \right)^2$$

minimiert werden. Mit

$$\underline{a} := \begin{bmatrix} a_1 \\ \vdots \\ a_m \end{bmatrix}, \tag{7.6}$$

$$H := \begin{bmatrix} h_1(x_1) & \cdots & h_m(x_1) \\ \vdots & \ddots & \vdots \\ h_1(x_n) & \cdots & h_m(x_n) \end{bmatrix}, \tag{7.7}$$

$$\underline{y} := \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \tag{7.8}$$

kann dies als

$$g(\underline{a}) = \left\| H\, \underline{a} - \underline{y} \right\|_2^2$$

geschrieben werden.

Für den Gradient erhalten wir

$$\nabla g(\underline{a}) = 2\,H^{\mathrm{T}}\left(H\,\underline{a} - \underline{y}\right),$$

sodass die stationären Punkte Lösungen des linearen Gleichungssystems

$$\boxed{H^{\mathrm{T}}\,H\,\underline{a} = H^{\mathrm{T}}\,\underline{y}}$$

sind. Dieses $(m \times m)$-Gleichungssystem hat immer eine Lösung, die in allen praktisch relevanten Fällen $(n \geq m)$ auch eindeutig ist. Die Systemmatrix ist immer symmetrisch und in allen praktisch relevanten Fällen auch positiv definit. Über die Kondition des Gleichungssystems und damit über die Kondition des Kleinste-Quadrate-Verfahrens sind keine allgemeinen Aussagen möglich. Die Kondition wird maßgeblich vom Zusammenspiel der Messstellen mit den Ansatzfunktionen bestimmt. Bei üblichen Ansatzfunktionen und gleichmäßig verteilten Messstellen kann die Kondition als gut angenommen werden.

Beispiel

Wenden die allgemeine Formel für das Finden einer Ausgleichsgerade

$$f(x) = a_1\,x + a_2$$

an.

Haben $\quad m = 2, \quad h_1(x) = x, \quad h_2(x) = 1 \quad$ und

$$\underline{a} := \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}, \qquad H = \begin{bmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix}, \qquad \underline{y} := \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Systemmatrix und rechte Seite sind also

$$H^{\mathrm{T}}\,H = \begin{bmatrix} \sum_{k=1}^{n} x_k^2 & \sum_{k=1}^{n} x_k \\ \sum_{k=1}^{n} x_k & n \end{bmatrix}, \qquad H^{\mathrm{T}}\,\underline{y} = \begin{bmatrix} \sum_{k=1}^{n} x_k\,y_k \\ \sum_{k=1}^{n} y_k \end{bmatrix}.$$

7.6 Hürden im praktischen Einsatz

Verteilung der Messstellen

Die Messstellen sollten möglichst gleichmäßig im Definitionsbereich der gesuchten Funktion f verteilt sein. In Bereichen mit überdurchschnittlich vielen Messstellen wird sich

die Funktion stärker den Messpunkten anpassen als in Bereichen mit nur wenigen Messstellen.

Umgekehrt kann man diesen Effekt nutzen, um durch Vervielfachung von Messpunkten diesen Messpunkten mehr Einfluss auf die Lösung zu geben. Das kann sinnvoll sein, wenn gleichzeitig Messpunkte mit kleinem Messfehler und Messpunkte mit großen Messfehler vorliegen. Eine Vervielfachung der genaueren Punkte gibt diesen mehr Einfluss auf die Kleinste-Quadrate-Lösung.

Nichtlineare Ansatzfunktionen

Das Lösen eines Kleinste-Quadrate-Problems als lineares Gleichungssystem funktioniert nur, wenn die Ansatzfunktion f linear bezüglich ihrer Parameter ist.

Manche nichtlineare Ansatzfunktionen, wie z.B.

$$f(x) = a\,\mathrm{e}^{b\,x},$$

können durch eine geeignete Transformation der Messwerte zu linearen Ansatzfunktionen umgebaut werden (IDVID 790).

Meistens, wie z.B. bei

$$f(x) = \sin(a\,x + b),$$

müssen Lösungsverfahren für nichtlineare Optimierungsprobleme eingesetzt werden. Beispielsweise liefert $\nabla g(a_1, \ldots, a_m) = 0$ für die Zielfunktion g des Kleinste-Quadrate-Problems ein nichtlineares Gleichungssystem, welches mittels Newton-Verfahren gelöst werden kann.

8 Interpolation

Für die Umwandlung von Punktfolgen in Funktionen haben wir bisher nur die Ausgleichsrechnung kennengelernt. Liegen nur wenige und kaum fehlerbehaftete Punkte vor, so ist neben der Approximation der Punkte durch eine Funktion auch die **Interpolation** möglich, also das Finden einer Funktion, die exakt durch alle gegebenen Punkte verläuft.

Um Funktionen mit dem Computer beschreiben zu können, müssen diese durch endlich viele Werte vollständig festlebar sein. Bei der Ausgleichsrechnung hatten wir deshalb auf gewisse Funktionenklassen beschränkt, in denen jede Funktion durch endlich viele Parameter beschrieben war. Bei der Interpolation beschränkt man sich praktisch immer auf **Polynome**

$$a_0 + a_1\,x + a_2\,x^2 + \cdots + a_{n-1}\,x^{n-1} + a_n\,x^n \tag{8.1}$$

oder sogenannte trigonometrische Polynome (werden hier nicht behandelt). Gründe:

- Auswertung eines Polynoms an einer Stelle x erfordert nur wenige Grundoperationen (Multiplikationen, Additionen).
- Polynome sind bei theoretischen Betrachtungen sehr gut handhabbar.
- Polynome sind beliebig oft differenzierbar, also in vielfältigen Szenarien einsetzbar (Optimierung!).
- Polynome sind leicht integrierbar (Flächenberechnungen!).

Beschäftigen uns zunächst mit der klassischen Polynominterpolation (ein Polynom auf dem gesamte Definitionsbereich) und geben dann einen Überblick über Splines (stückweise polynomiale Funktionen).

8.1 Formale Problemstellung

Gegen ist eine Folge von Punkten $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{R}^2$. Finde ein Polynom $p : [a, b] \to \mathbb{R}$ (oder einen Spline), sodass

$$p(x_k) = y_k \quad \text{für } k = 1, \ldots, n \tag{8.2}$$

gilt, wobei die Intervallgrenzen a und b so sein sollen, dass $x_k \in [a, b]$ für $k = 1, \ldots, n$.

Die Zahlen $x_1, \ldots, x_n$ werden **Stützstellen** genannt. Im Folgenden nehmen wir stets an, dass **alle Stützstellen verschieden** sind, also $x_k \neq x_l$ für $k \neq l$ gilt.

8.2 Polynominterpolation

Direkter Ansatz

Sind n zu interpolierende Punkte gegeben, so erscheint ein Polynom $(n-1)$-ten Grades als Interpolante zweckmäßig, da dieses genau n frei wählbare Parameter enthält. Setzt man die einzelnen Punkte in die Interpolationsbedingung (8.1) ein, so erhält man das lineare Gleichungssystem

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \tag{8.3}$$

Die Systemmatrix heißt Vandermonde-Matrix und ist stets invertierbar (wenn alle x_k verschieden sind). Das Lösen des Gleichungssystems benötigt etwa n^3 Rechenoperationen und ist damit deutlich aufwendiger als weiter unten behandelte Verfahren zur Polynominterpolation.

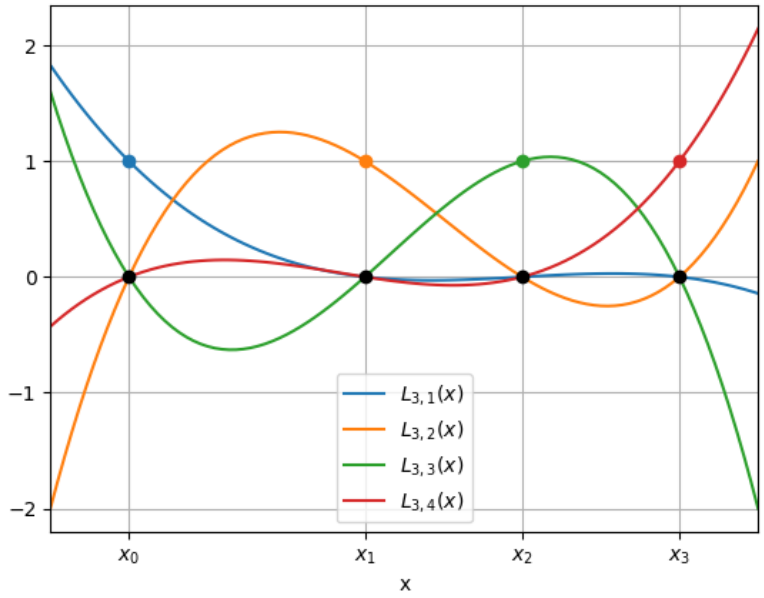
Lagrange-Interpolation

Statt das obige Gleichungssystemzu lösen, können wir für das gesuchte Interpolationspolynom auch direkt eine Formel angeben. Benötigen dazu die sogenannten Langrange'schen Basispolynome vom Grad $n-1$ zu den Stützstellen $x_1, \ldots, x_n$:

$$L_{n-1,k}(x) := \prod_{l \neq k} \frac{x - x_l}{x_k - x_l}, \quad k = 1, \ldots, n. \tag{8.4}$$

Offensichtlich gilt

$$L_{n-1,k}(x_l) = \begin{cases} 1, & \text{wenn } l = k, \\ 0, & \text{wenn } l \neq k. \end{cases} \tag{8.5}$$



Somit ist

$$y_1 \, L_{n-1,1}(x) + \cdots + y_n \, L_{n-1,n}(x) \tag{8.6}$$

das gesuchte Interpolationspolynom. In dieser Darstellung wird das Interpolationspolynom als **Langrange'sches Interpolationspolynom** bezeichnet.

Damit ist die **Existenz** eines interpolierenden Polynoms vom Grad $n-1$ gezeigt. Die **Eindeutigkeit** folgt leicht aus der Tatsache, dass ein Polynom vom Grad $n-1$ höchstens n Nullstellen hat (IDVID 810).

Beachte: Durch die zu interpolierenden Punkte ist das Interpolationspolynom eindeutig bestimmt. Ja nach Darstellung dieses Polynoms als Formel gibt man ihm aber verschiedenen Namen, z.B. Langrange'sches Interpolationspolynom und weiter unten dann Newton'sches Interpolationspolynom. Das durch die jeweilige Formel beschriebene Polynom ist stets das selbe.

Das Lagrange'sche Interpolationspolynom ist für den praktischen Einsatz kaum von Bedeutung, leistet aber bei theoretischen Überlegungen (Existenz, Eindeutigkeit, Kondition) sehr gute Dienste.

Anwendungsfälle

Polynominterpolation ist Baustein in zahlreichen numerischen Verfahren. Grundsätzlich sind zwei Anwendungsfälle zu unterscheiden:

- Das Interpolationspolynom soll nur an wenigen Stellen ausgewertet werden (z.B. Interpolation von 6-stündlichen Temperaturmessungen um 0 Uhr, 6 Uhr, 12 Uhr, 18 Uhr zum Erhalt einer Temperatur um 8 Uhr).

- Das Interpolationspolynom selbst ist von Interesse oder soll an sehr vielen Stellen ausgewertet werden (z.B. Ersatz einer komplizierten Funktion wie sin durch ein Polynom).

Im ersteren Fall lohnt sich das Aufstellen des Polynoms nicht. Man setzt hier üblicherweise das Aitken-Neville-Verfahren (siehe unten) zur effizienten Auswertung des (nicht explizit aufgestellten) Interpolationspolynoms an einer Stelle ein.

Im zweiten Fall ist man bestrebt, das Interpolationspolynom in einer Form aufzustellen, die einerseits effizient zu bekommen ist und andererseits die effiziente Auswertung des Polynoms ermöglicht (siehe unten).

Kondition

Üblicherweise sind die Stützstellen $x_1, \ldots, x_n$ nicht oder nur geringfügig mit Fehlern behaftet. Das die zugehörigen $y_1, \ldots, y_n$ jedoch meist aus Messungen entstehen, ist hier mit relevanten Fehlern zu rechnen. Für die praktische Arbeit stehen uns als statt der Punkte $(x_1, y_1), \ldots, (x_n, y_n)$ nur die fehlerbehafteten Punkte $(x_1, \tilde{y_1}), \ldots, (x_n, \tilde{y_n})$ zur Verfügung. Daraus ergeben sich das exakte Interpolationspolynom p und das auf fehlerhaften Eingaben beruhende Interpolationspolynom $\tilde{p}$.

Fehlermaße Als Fehlermaß für die Eingaben wählen wir den relativen Fehler bzgl. der ∞-Norm der Vektoren y bzw $\tilde{y}$, also

$$\frac{\|y - \tilde{y}\|_\infty}{\|y\|_\infty}, \tag{8.7}$$

da wir für den Abstand zweier Polynome mangels Kenntnis anderer Abstandsmaße zwischen Funktionen ebenfalls den maximalen punktweisen Abstand wählen müssen. Für eine stetige Funktion $f : [a, b] \to \mathbb{R}$ setzen wir

$$\|f\|_\infty := \max_{x \in [a,b]} |f(x)| \tag{8.8}$$

und drücken den Ausgabefehler mittels

$$\frac{\|p - \tilde{p}\|_\infty}{\|p\|_\infty}. \tag{8.9}$$

aus.

Fehlerabschätzung Stellen wir p und $\tilde{p}$ als Lagrange'sches Interpolationspolynom dar, also

$$p(x) = y_1 \, L_{n-1,1}(x) + \cdots + y_n \, L_{n-1,n}(x) \tag{8.10}$$

und

$$\tilde{p}(x) = \tilde{y}_1 \, L_{n-1,1}(x) + \cdots + \tilde{y}_n \, L_{n-1,n}(x), \tag{8.11}$$

so erhalten wir

$$\frac{\|p-\tilde{p}\|_\infty}{\|p\|_\infty} \leq \kappa \, \frac{\|y-\tilde{y}\|_\infty}{\|y\|_\infty} \tag{8.12}$$

mit

$$\kappa := \max_{x\in[a,b]} \sum_{k=1}^{n} |L_{n-1,k}(x)| \tag{8.13}$$

(IDVID 620).

Aus der Definition der Langrange'schen Basispolynome $L_{n-1,k}$ sieht man leicht (schauen wir uns hier aber nicht an), dass κ bei festem n nicht von der Lage des Intervalls $[a,b]$ abhängt, sondern nur von der Verteilung der Stützstellen innerhalb des Intervalls. Auch die Werte $y_1,\ldots,y_n$ gehen nicht in κ ein. Die Frage nach der Kondition ist also ausschließlich eine Frage der Stützstellenverteilung.

Merke!

Für **äquidistante Stützstellen**, also $x_k = a + (b-a)\,\frac{k-1}{n-1}$, kann man zeigen, dass

$$\kappa \approx \frac{2^n}{(n-1)\,\log(n-1)} \tag{8.14}$$

für große n gilt, also die **Kondition schlecht** ist (für große n).

Wählt man hingegen beispielsweise die sogenannten **Tschebyscheff-Stützstellen**

$$x_k = a + (b-a)\,\left(\frac{1}{2} - \frac{1}{2}\,\cos\left(\frac{2\,k-1}{2\,n}\,\pi\right)\right), \tag{8.15}$$

so kann man

$$\kappa \approx \frac{2}{\pi}\,\log(n-1) \tag{8.16}$$

für große n zeigen. Man erhält also eine **gute Kondition**. Die Tschebyscheff-Stützstellen nicht äquidistant, sondern verdichten sich mit zunehmendem Abstand von der Intervallmitte.

Algorithmen

Direkter Ansatz und Horner-Schema Das Aufstellen des Interpolationspolynoms mittels Vandermonde-Matrix benötigt ca. n^3 Rechenopertationen (Gauss-Algorithmus). Auch hat die Vandermonde-Matrix oft schon bei niedrigem zweistelligem n eine zu hohe Konditionszahl, sodass dieser Ansatz in der Praxis keine Rolle spielt.

Das anschließende Auswerten des Polynoms an einer Stelle erfordert \frac{n,(n+1)}{2} Multiplikationen und n Additionen, liegt also in der Größenordnung von n^2 Grundoperationen.

Liegt das Polynom in der Form

$$p(x) = a_0 + a_1\,x + a_2\,x^2 + \cdots + a_{n-2}\,x^{n-2} + a_{n-1}\,x^{n-1} \tag{8.17}$$

vor, so kann man die Auswertung an einer Stelle noch beschleunigen, denn offensichtlich gilt

$$p(x) = a_0 + x\,(a_1 + x\,(a_2 + \cdots + x\,(a_{n-2} + x\,a_{n-1})\cdots)) \tag{8.18}$$

Der daraus abgeleitete Algorithmus zur Polynomauswertung heißt **Horner-Schema** und benötigt nur $n-1$ Multiplikationen und $n-1$ Additionen, liegt also in der Größenordnung n:

1. Setze $y := a_{n-1}$.
2. Wiederhole für $k = n-2, \ldots, 0$:
   a) Setze $y := a_k + x\,y$.

Bei gegebenen $a_0, \ldots, a_{n-1}$ und x liefert dieser Algorithmus $y = p(x)$.

Das Horner-Schema ist für nicht zu große n stabil wie man mittels Rückwärtsanalyse zeigen kann (tun wir hier nicht). Insbesondere sind sie Zwischenergebnisse kleiner als beim direkten Auswerten (Potenzen von x!), sodass die Gefahr von Exponentenüberläufen geringer ist.

Aitken-Neville-Schema Soll das Interpolationspolynom nur an wenigen Stellen ausgewertet werden, so muss das Polynom gar nicht explizit aufgestellt werden. Um auch ohne Kenntnis des Polynoms p den Wert $p(x)$ für ein Stelle x zu bekommen, verwenden wir den folgenden Zusammenhang zwischen den Interpolationspolynom vom Grad $n-1$ und zwei Interpolationspolynomen vom Grad $n-2$ zu unterschiedlichen Stützpunkten:

Merke!

Zu gegebenen Punkten $(x_1, y_1), \ldots, (x_n, y_n)$ seien p das Interpolationspolynom vom Grad $n-1$ und p_1 und p_2 die Interpolationspolynome vom Grad $n-2$ zu den Punkten $(x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$ bzw. $(x_2, y_2), \ldots, (x_n, y_n)$. Für jede Stelle x gilt dann

$$p(x) = \frac{x_n - x}{x_n - x_1}\,p_1(x) + \frac{x - x_1}{x_n - x_1}\,p_2(x) \tag{8.19}$$

(IDVID 830).

Bezeichnen wir mit $P_{l,k}$ den Wert des Interpolationspolynoms vom Grad $l-1$ zu den Punkten $x_k, \ldots, x_{k+l-1}$ an einer festen Stelle x, so erhalten wir den als **Aitken-Neville-Schema** bekannten Zusammenhang

$$P_{l,k} = \frac{x_{k+l-1} - x}{x_{k+l-1} - x_k}\,P_{l-1,k} + \frac{x - x_k}{x_{k+l-1} - x_k}\,P_{l-1,k+1}. \tag{8.20}$$

Ausgehend von

$$P_{1,k} = y_k, \quad k = 1, \ldots, n \tag{8.21}$$

erhalten wir daraus Schritt für Schritt den gesuchten Wert

$$P_{n,1} = p(x) \tag{8.22}$$

(IDVID 635). Das Aitken-Neville-Schema benötigt etwa n^2 Rechenoperationen und lässt sich mit n Speicherplätzen umsetzen, wenn man nich mehr benötigte Zwischenergebnisse wieder überschreibt:

1. Setze $p_1 := y_1$.
2. Wiederhole für $i = 2, \ldots, n$:
   a) Setze $p_i := y_i$.
   b) Wiederhole für $j = i - 1, \ldots, 1$:
      i. Setze

$$p_j := \frac{(x_i - x)\, p_j + (x - x_j)\, p_{j+1}}{x_i - x_j}. \tag{8.23}$$

3. Liefere p_1 als Ergebnis.

Kann man die Eingaben $y_1, \ldots, y_n$ überschreiben, so wird kein zusätzlicher Speicher benötigt.

Die Rekursionsvorschrift in Schritt 2.2.1 des Algorithmus ist als stabil anzusehen. Auslöschungseffekte sind im Allgemeinen nicht zu erwarten.

Newton-Interpolation Ist das Interpolationspolynom explizit gesucht, z.B. um es an vielen verschiedenen Stellen auswerten zu können, kann man ähnlich zum Aitken-Neville-Schema ebenfalls eine rekursive Vorschrift finden, die Schritt für Schritt Interpolationspolynome der Ordnungen $0, 1, \ldots, n-1$ zu stets um einen Punkt erweiterten Punktmengen liefert.

Merke!

Mit den **Newton'schen Basispolynomen**

$$N_0(x) := 1 \tag{8.24}$$

und

$$N_l(x) := (x - x_1) \cdots (x - x_l) \tag{8.25}$$

für $l = 1, \ldots, n-1$ kann das Interpolationspolynom p vom Grad $n-1$ zu den Punkten $(x_1, y_1), \ldots, (x_n, y_n)$ als **Newton'sches Interpolationspolynom**

$$p(x) = \sum_{l=1}^{n} b_{l,1}\, N_{l-1}(x) \tag{8.26}$$

dargestellt werden. Dabei sind die Koeffizienten $b_{l,1}$ durch die rekursive Berechnungsvorschrift

$$b_{1,k} := y_k \tag{8.27}$$

für $k = 1, \ldots, n$ und

$$b_{l,k} := \frac{b_{l-1,k+1} - b_{l-1,k}}{x_{k+l-1} - x_k} \tag{8.28}$$

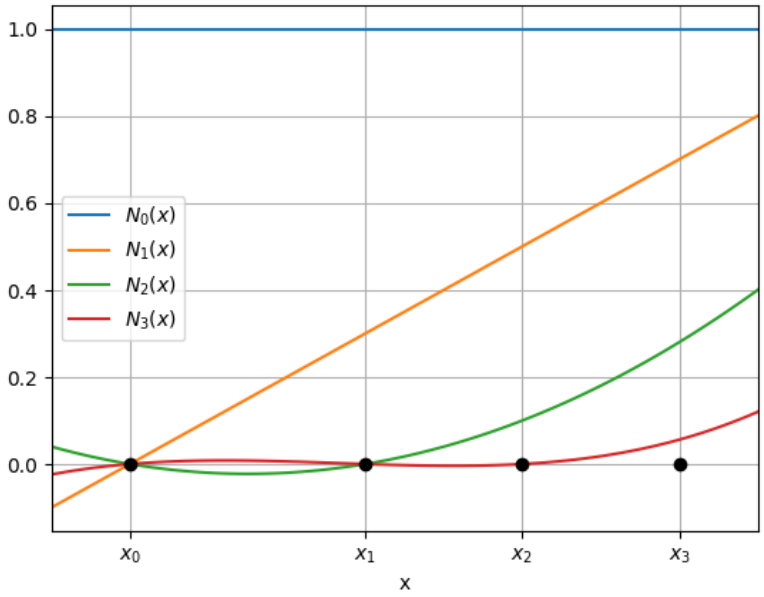
für $l = 2, \ldots, n$ und $k = 1, \ldots, n - l + 1$ gegeben.

Alternativ können die Newton'schen Interpolationspolynome $p_0, p_1, \ldots, p_{n-1} = p$ zu den von (x_1, y_1) ausgehenden und stets um einen Punkt wachsenden Punktmengen über die rekursive Vorschrift

$$p_0(x) = y_0, \quad p_{l-1}(x) = p_{l-2}(x) + b_{l,1}\, N_{l-1}(x) \tag{8.29}$$

erhalten werden.

Herleitung der obigen Formeln: IDVID 650.



Die Berechnung der Koeffizienten $b_{1,1}, \ldots, b_{n,1}$ benötigt ca. n^2 Operationen, ist also deutlich schneller möglich als über das Lösen eines Gleichungssystems. Der Algorithmus dafür

ist völlig analog zum Aitken-Neville-Schema.

Das Auswerten des des Newton'schen Interpolationspolynoms an einer Stelle x kann ähnlich zum Horner-Schema in ca. n Operationen erfolgen, denn es gilt

$$p(x) = b_{1,1} + (x - x_1)\,(b_{2,1} + \cdots + (x - x_{n-2})\,(b_{n-1,1} + (x - x_{n-1})\,b_{n,1})\cdots). \tag{8.30}$$

Algorithmus zum Auswerten:

1. Setze $P := b_{n,1}$.
2. Wiederhole für $l = n-1, \ldots, 1$:
   a) Setze $P := b_{l,1} + (x - x_l)\,P$.

Am Ende des Algorithmus gilt $P = p(x)$.

Approximationsfehler

Nutzt man durch Interpolation erhaltene Polynome als einfach handhabbaren Ersatz für kompliziertere Funktionen, so stellt sich die Frage nach dem Abstand $\|f - p\|_\infty$ des Interpolationspolynoms p zur ursprünglichen Funktion f zwischen den Stützstellen. Diese Frage ist eine andere als die nach der Kondition, da hier Fehler in den Eingaben gar keine Rolle spielen.

Man kann folgendes Resultat zeigen (tun wir hier aber nicht):

Merke!

Sei $[a, b]$ ein Intervall, welches die Stützstellen $x_1, \ldots, x_n$ enthält, und sei $f : [a, b] \to \mathbb{R}$ dort n-mal stetig differenzierbar. Dann gilt

$$\|f - p\|_\infty \leq \frac{1}{n!}\,\|N_n\|_\infty\,\|f^{(n)}\|_\infty, \tag{8.31}$$

wobei p das Interpolationspolynom zu den Punkten $(x_1, f(x_1)), \ldots, (x_n, f(x_n))$ und

$$N_n(x) := \prod_{k=1}^{n} (x - x_k) \tag{8.32}$$

das Newton'sche Basispolynom vom Grad n ist.

Einige Bemerkungen zur Interpretation:

- Der Faktor $\frac{1}{n!}$ wird umso kleiner je mehr Stützstellen genutzt werden.
- Der Faktor $\|f^{(n)}\|_\infty$ ist unabhängig von der Lage der Stützstellen und wird meistens kleiner je größer n ist. Nur Polynome erfüllen $\|f^{(n)}\|_\infty = 0$ für alle hinreichend

großen n. Grob formuliert: Je schneller $\|f^{(n)}\|_\infty$ gegen Null geht für große n, desto polynomähnlicher ist die Funktion f.

- Der Faktor $\|N_n\|_\infty$ hängt nicht von f ab und auch nicht vom Intervall $[a, b]$ (Verschiebung des Intervall ändert den Wert nicht), sondern nur von der Verteilung der Stützstellen innerhalb des Intervalls. Für äquidistante Stützstellen wird dieser Faktor groß sein, für die Tschebyscheff-Stützstellen hingegen klein (vgl. Betrachtungen zur Kondition).
- Bei festem n wird $\|N_n\|_\infty$ umso kleiner je kleiner das Intervall $[a, b]$ ist.

Daraus lassen sich folgende **Grundsätze für die Polynominterpolation** ableiten:

- Nur auf kleinen Intervallen interpolieren. Falls nötig, mehrer Interpolationspolynome "zusammenstückeln" (siehe Splines unten).
- Falls kleine Intervalle nicht möglich, dann Stützstellen günstig wählen, z.B. Tschebyscheff-Stützstellen.

Das folgende Beispiel zeigt, dass eine Erhöhung der Stützstellenzahl nicht automatisch den Approximationsfehler verbessert.

Beispiel

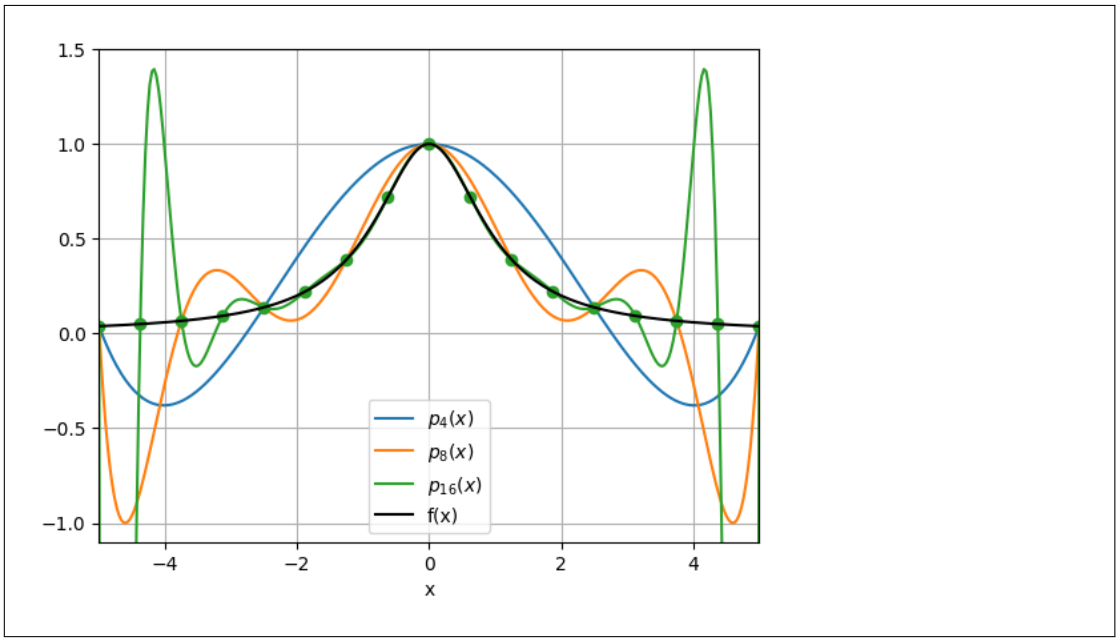
Wird die Funktion

$$f(x) = \frac{1}{1 + x^2} \tag{8.33}$$

auf dem Intervall $[-5, 5]$ an n äquidistanten Stützstellen durch ein Polynom p_{n-1} interpoliert, so kann man

$$\lim_{n\to\infty} \|f - p_{n-1}\|_\infty = \infty \tag{8.34}$$

zeigen (tun wir hier aber nicht).



8.3 Splines

Idee

Hatten festgestellt, dass der Approximationsfehler bei Interpolation auf kleinen Intervallen geringer ist als auf großen Intervallen. Auch führt ein hoher Polynomgrad (also viele Stützstellen) nicht automatisch zu besserer Ergebnissen. Aus diesen Beobachtungen sind die so genannten Splines (Englisch für Straklatte) als Ersatz für Interpolationspolynome entstanden:

- Zu gegebenen Stützstellen $x_1, \ldots, x_n$ finde für jedes Intervall $[x_k, x_{k+1}]$ ein separates Interpolationspolynom vom Grad m (meist klein, z.B. $m = 3$).
- An den Intervallenden sollen die Ableitungen der benachbarten Polynome bis zur Ordnung $m - 1$ übereinstimmen, damit die Verbindungsstellen “glatt” sind.
- Bei x_1 und x_n können in Summe $m - 1$ Ableitungen frei gewählt werden.

Aus diesem Ansatz ergeben sich

- $2\,(n-1)$ Bedingungen für die intervallweise Interpolation,
- $(m-1)\,(n-2)$ Bedingungen für die Ableitungen an den inneren Intervallgrenzen,
- $m - 1$ Bedingungen für die Ableitungen an den äußeren Intervallgrenzen,

also insgesamt $(m+1)\,(n-1)$ Bedingungen. Die Anzahl der zu bestimmenden Koeffizienten bei $n - 1$ Polynomen vom Grad m ist ebenfalls $(m+1)\,(n-1)$. Alle Bedingungen

liefern lineare Gleichungen, sodass die Spline-Koeffizienten als Lösung eines linearen Gleichungssystems berechnet werden können.

Für Splines existiert analog zur Polynominterpolation eine vollständige, ausgereifte Theorie und auch eine umfangreich Software-Unterstützung.

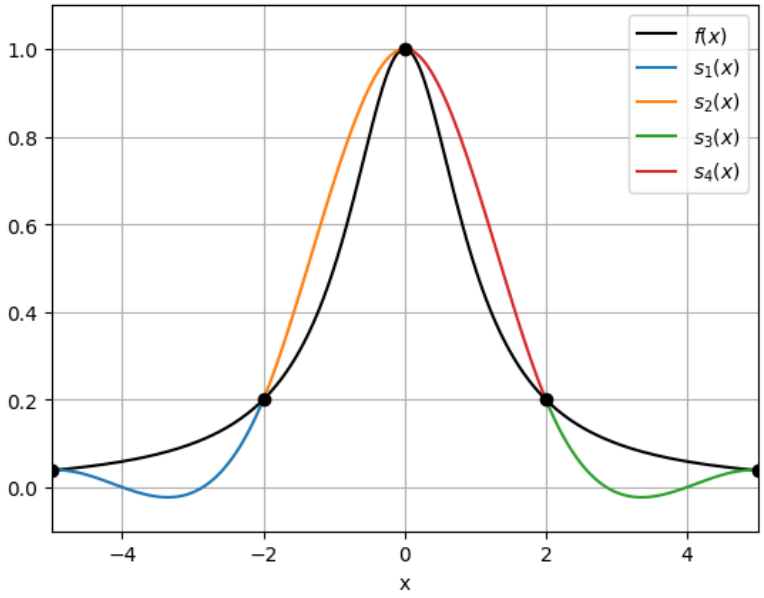
Beispiel

Wollen die Funktion

$$f(x) = \frac{1}{1+x^2} \tag{8.35}$$

auf dem Intervall $[-5, 5]$ durch einen kubischen Spline approximieren. Wählen die Stützstellen $\{-5, -2, 0, 2, 5\}$, also 4 Teilintervalle. An den äußeren Intervallgrenzen soll die erste Ableitung des Splines der ersten Ableitung der Funktion gleichen.

Die vier Polynome s_1, s_2, s_3, s_4, die zusammen den Spline ergeben, sind in der Abbildung dargestellt (IDVID 680):



Kondition und Algorithmen

Die Kondition für das Berechnen der Spline-Koeffizienten ist sehr gut (ohne Beweis).

Für die Berechnung der Koeffizienten muss analog zu den Interpolationspolynomen kein Gleichungssystem gelöst werden. Stattdessen kann man spezielle Basis-Splines einführen, die dann das effiziente und stabile Aufstellen des gesuchten Splines ermöglichen (vgl. auch Langrange'sche und Newton'sche Basispolynome). Mit dieser speziellen Basis dargestellte Splines heißen auch **B-Splines** (kurz für Basis-Splines).

Kurven und Flächen

Kurven in der Ebene oder im Raum sind Funktionen, die ein Intervall $[a, b]$ nach $\mathbb{R}^2$ oder $\mathbb{R}^3$ abbilden. Ist jede Koordinatenfunktion ein Spline, so erhält man leicht zu handhabende Kurven für die grafische Modellierung. Die Koeffizienten bzgl. der B-Spline-Basis können in Grafiksoftware üblicherweise über sogenannte Kontrollpunkte verändert werden.

Ein alternative Basis zur B-Spline-Basis sind die Bernstein-Polynome, die dann sogenannte **Bézier-Kurven** liefern. Diese können ebenfalls über Kontrollpunkte leicht und intuitiv angepasst werden.

NURBS (Non-Uniform Rational B-Splines) sind eine Verallgemeinerung der Splines, welche auch Quotienten von Polynomen zulassen. Neben Kurven können NURBS auch nahezu beliebige Flächen beschreiben, sodass sie heute das Standardwerkzeug für die Freiformmodellierung sind.

9 Numerische Integration

Ziel der numerischen Integration (auch: numerische Quadratur) ist das Berechnen von bestimmten Integralen

$$\int_a^b f(x)\,\mathrm{d}x \tag{9.1}$$

für gegebene Funktionen $f : [a, b] \to \mathbb{R}$. Dabei soll nicht die Stammfunktion von f genutzt werden, sondern nur die Funktionswerte an endlichen vielen Stellen $x_1, \ldots, x_n \in [a, b]$.

9.1 Idee

Verschiedene Ansätze führen zum Ziel. Die wichtigste Gruppe von numerischen Integrationsverfahren sind die sogenannten **interpolatorischen Verfahren**. Hier wird die Funktion f an zweckmäßig (siehe unten) gewählten Stützstellen interpoliert, um dann das Integral der Interpolante zu berechnen. Sofern der Interpolationsfehler klein genug ist, wird der so berechnete Wert in der Nähe des tatsächlichen Wertes des Integrals liegen.

Neben einfacher Polynominterpolation kommen vorallem stückweise Polynome und Splines zum Einsatz. Integrale von Polynomen lassen sich mit den Computer exakt berechnen. Neben der Frage der Kondition ist hier, analog zu den Lösungsverfahren für nichtlineare Gleichungen, insbesondere die Frage der **Konvergenz** zu klären, da auch beim exakten Rechnen ohne Rundungsfehler nur eine Näherungslösung zu erwarten ist.

Numerische Integrationsverfahren können auch danach beurteilt werden, für welche Funktionsklassen sie exakte Lösungen liefern. Meist sind diese Polynome bis zu einem gewissen Grad.

9.2 Kondition

Merke!

Sind f und $\tilde{f}$ exakte und fehlerbehaftete Eingaben mit Ausgaben

$$I := \int_a^b f(x)\,\mathrm{d}x \quad \text{und} \quad \tilde{I} := \int_a^b \tilde{f}(x)\,\mathrm{d}x, \tag{9.2}$$

so gilt

$$\frac{I - \tilde{x}}{I} \leq \kappa(f)\,\frac{\|f - \tilde{f}\|_\infty}{\|f\|_\infty} \tag{9.3}$$

mit

$$\kappa(f) := \frac{(b-a)\,\|f\|_\infty}{\left|\int_a^b f(x)\,\mathrm{d}x\right|} \tag{9.4}$$

(IDVID 910).

Die Kondition ist somit gut, außer für Funktion f mit $\int_a^b f(x)\,\mathrm{d}x \approx 0$. In diesem Fall führen auch kleine absolute Ausgabefehler zu großen relativen Fehlern aufgrund von Auslöschungseffekten.

9.3 Konvergenz

Bezeichnen wir mit I_n den auf der Grundlage von Funktionsauswertungen an n Stützstellen $x_1, \ldots, x_n$ erhaltenen Näherungswert für das exkate Integral I, so sollte ein Verfahren für die numerische Integration die Bedingung

$$\lim_{n\to\infty} |I - I_n| = 0 \tag{9.5}$$

erfüllen. Man sagt dann: Das verfahren **konvergiert**.

Im Gegensatz zu Lösungsverfahren für nichtlineare Gleichungen spielt der Begriff der Konvergenzordnung bei der numerischen Integration nur eine untergeordnete Rolle.

9.4 Exaktheit

Merke!

Ein numerisches Integrationsverfahren heißt **exakt vom Grad m**, wenn es Polynome bis zum Grad m exakt integriert.

Werden n Stützstellen verwendet, so ist klar, dass man stets ein Verfahren findet, welches exakt vom Grad $n-1$ ist (ersetze f durch das entsprechende Interpolationspolynom). Weiter unten werden wir sehen, dass auch Verfahren mit Exaktheitsgrad $2\,n-1$ möglich sind.

9.5 Allgemeiner Ansatz

Merke!

Wir beschränken uns auf Integrationsverfahren der Form

$$I_n := \sum_{k=1}^{n} w_k\, f(x_k) \tag{9.6}$$

mit Stützstellen $x_1, \ldots, x_n \in [a, b]$ und **Gewichten** $w_1, \ldots, w_n \in [0, \infty)$. Je nach Wahl der Stützstellen und der Gewichte entstehen unterschiedlich Verfahren.

Da sinnvollerweise $I_n \geq 0$ für alle nichtnegativen Funktionen f gelten soll, ist klar, dass nur positive Gewichte w_k in Frage kommen.

Soll das Verfahren wenigstens exakt vom Grad 0 sein (also konstante Funktionen exakt integrieren), so muss

$$\sum_{k=1}^{n} w_k = b - a \tag{9.7}$$

gelten (IDVID 920).

9.6 Newton-Cotes-Formeln

Idee

Wählt man die Stützstellen äquidistant, also

$$x_k := a + (b-a)\,\frac{k-1}{n-1}, \quad k = 1, \ldots, n, \tag{9.8}$$

und ersetzt die zu integrierende Funktion f durch das entsprechende Interpolationspolynom vom Grad $n-1$, so erhält man die so genannten Newton-Cotes-Quadraturformeln. Sind $L_{n-1,1}, \ldots, L_{n-1,n}$ die Langrange'schen Basispolynome, so folgt

$$w_k = \int_a^b L_{n-1,k}(x)\,\mathrm{d}x \tag{9.9}$$

(IDVID 930).

Newton-Cotes-Formeln liefern Verfahren mit Exaktheitsgrad $n-1$. Man kann sogar Exaktheitsgrad n zeigen bei ungeradem n (tun wir hier nicht), wenn man Symmetrieeigenschaften der Lagrange-Basispolynome ausnutzt. Allerdings ist auch bekannt, dass ab $n = 9$ negative Gewichte auftreten. Somit ist der sinnvolle Einsatz nur bei sehr geringer Stützstellenanzahl möglich.

Ein Ausweg sind sogenannte **summierte Formeln**: Für je m benachbarte Stützstellen wende die Newton-Cotes-Formel an und addiere alle so erhaltenen Teilresultate. Ist $n = s\,m$, so wird die Newton-Cotes-Formel also s-mal angewendet. Dieses Verfahren ist dann exakt vom Grad $m-1$.

Approximationsfehler

Ausgehend von der Abschätzung des Approximationsfehlers bei der Polynominterpolation kann man folgendes Resultat zeigen:

Merke!

Sei $f : [a,b] \to \mathbb{R}$ eine n-mal stetig differenzierbare Funktion mit $n = s\,m$. Ist p das stückweise Interpolationspolynom für f vom Grad $m-1$ zu den äquidistanten

Stützstellen $x_1, \ldots, x_n$ und ist h der Abstand der Stützstellen, so gilt

$$|I - I_n| \leq \frac{|b-a|}{m!}\, h^m\, \|f^{(m)}\|_\infty \tag{9.10}$$

(IDVID 940).

Je mehr Stützstellen genutzt werden, desto kleiner ist h und desto kleiner wird der Approximationsfehler werden. Je größer m, desto schneller sinkt der Fehler bei Erhöhung der Stützstellenanzahl.

Beispiele

Rechteckregel Für $m = 1$ (Interpolationspolynome vom Grad 0) wird pro Interpolationspolynom nur eine Stützstelle benötigt, sodass nicht klar ist, auf welchem Intervall die einzelnen Interpolationspolynome zum Einsatz kommen. Zwei Varianten sind üblich ($h := \frac{1}{n-1}$):

- $[x_1, x_1 + \frac{h}{2}]$, $[x_k - \frac{h}{2}, x_k + \frac{h}{2}]$, $[x_n - \frac{h}{2}, x_n]$, $k = 2, \ldots, n-1$,
- $[x_k, x_{k+1}]$, $k = 1, \ldots, n-1$.

Bei der ersten Variante ist die Quadraturformel etwas komplizierter:

$$I_n = \frac{1}{n-1}\left(\frac{1}{2}\, f(x_1) + \sum_{k=2}^{n-1} f(x_k) + \frac{1}{2}\, f(x_n)\right). \tag{9.11}$$

Bei der zweiten Variante ist die Formel einfacher, aber die letzte Stützstelle (Intervallende) hat keinerlei Einfluss:

$$I_n = \frac{1}{n-1} \sum_{k=1}^{n-1} f(x_k). \tag{9.12}$$

Verdoppelt man die Stützstellenanzahl, so wird sich der Approximationsfehler etwa halbieren.

Trapezregel Für $m = 2$ werden pro Interpolationspolynom zwei Stützstellen benötigt. Jedes Interpolationspolynom deckt also den Bereich zwischen zwei benachbarten Stützstellen ab. Statt der Funktion f wird somit das Integral einer stückweise linearen

Approximation berechnet.

$$\begin{aligned} I_n &= \sum_{k=1}^{n-1} \frac{x_{k+1} - x_k}{2} \big(f(x_k) + f(x_{k+1})\big) \\ &= \frac{h}{2} f(x_1) + h \sum_{k=2}^{n-1} f(x_k) + \frac{h}{2} f(x_n). \end{aligned} \tag{9.13}$$

Bei Verdoppelung der Stützstellenanzahl wird der Approximationsfehler etwa auf ein Viertel reduziert.

Simpson-Regel Für $m = 3$ werden pro Interpolationspolynom drei Stützstellen benötigt. Jedes Interpolationspolynom deckt also den Bereich zwischen drei benachbarten Stützstellen ab. Statt der Funktion f wird somit das Integral einer stückweise quadratischen Approximation berechnet. Achtung: Die Anzahl der Stützstellen muss hier ungerade sein.

$$I_n = \frac{h}{3} f(x_1) + \frac{h}{3} f(x_n) + \frac{2\,h}{3} \sum_{k=1}^{\frac{n-3}{2}} f(x_{2k+1}) + \frac{4\,h}{3} \sum_{k=1}^{\frac{n-1}{2}} f(x_{2k}) \tag{9.14}$$

9.7 Gauss-Quadratur

Wählt man die Stützstellen für die Polynominterpolation nicht äquidistant, kann man unter Umständen numerische Integrationsverfahren finden, deren Exaktheitsgrad bei n Stützstellen größer als $n - 1$ ist. Der maximal mögliche Exaktheitsgrad liegt bei $2\,n - 1$ (IDVID 970).

Man kann zeigen (zun wir hier nicht), dass der maximale Exaktheitsgrad erreicht wird, wenn die Stützstellen gerade die Nullstellen des Legendre-Polynoms vom Grad n sind und die Gewichte wie bei den Newton-Cotes-Formeln als Integrale der Lagrange-Basispolynome gewählt werden. Das so konstruierte, aus Sicht des Exaktheitsgrades optimale numerische Integrationsverfahren wird als **Gauß-Quadratur** bezeichnet (manchmal auch Gauß-Legendre-Quadratur).

Auch kann man zeigen, dass die Gewichte auch bei größerem n im Gegensatz zu den Newton-Cotes-Formeln stets positiv sind, sodass die Gauss-Quadratur weniger anfällig für Auslöschungseffekte ist als Newton-Cotes-Formeln.

Beispiel

Bei $n = 2$ Stützstellen liefert die Quadraturformel

$$I_2 = f\big(-\sqrt{1/3}\big) + f\big(\sqrt{1/3}\big) \tag{9.15}$$

exakte Ergebnisse für Polynome bis Grad 3 auf dem Intervall $[-1, 1]$. Die Newton-Cotes-Formel mit Exaktheitsgrad 3 (Simpson-Regel) benötigt hingegen drei Stützstellen bei -1, 0, 1.

Beachte: Die Gauß-Quadraturformeln werden meist nur für das Intervall $[-1, 1]$ angegeben. Soll über ein anderes Intervall $[a, b]$ integriert werden, so sind als Stützstellen und Gewichte

$$x_k = \frac{b-a}{2}\,\tilde{x}_k + \frac{a+b}{2} \quad \text{und} \quad w_k = \frac{b-a}{2}\,\tilde{w}_k \tag{9.16}$$

zu verwenden, wobei $\tilde{x}_k$ und $\tilde{w}_k$ die Stützstellen und Gewichte für das Intervall $[-1, 1]$ sind.

9.8 Monte-Carlo-Verfahren

Das Integral

$$I = \int_a^b f(x)\,\mathrm{d}x \tag{9.17}$$

kann als Erwartungswert einer Zufallsgröße interpretiert werden. Sei dazu X eine auf $[a, b]$ gleichverteilte Zufallsgröße und sei $Y := f(X)$ die durch Anwenden von f daraus entstehende transformierte Zufallsgröße. Mit der Dichtefunktion $p_X(x) = \frac{1}{b-a}$ zu X gilt

$$\mathbb{E}Y = \int_a^b f(x)\,p_X(x)\,\mathrm{d}x. \tag{9.18}$$

Somit

$$I = (b-a)\int_a^b f(x)\,p_X(x)\,\mathrm{d}x = (b-a)\,\mathbb{E}X. \tag{9.19}$$

Den Erwartungswert $\mathbb{E}X$ der Zufallsgröße X kann man über den Mittelwert aus n Realisierungen $x_1, \ldots, x_n$ annähern. Setzen wir

$$I_n := (b-a)\,\frac{1}{n}\sum_{k=1}^{n} f(x_k) \tag{9.20}$$

so gilt nach dem Gesetz der großen Zahlen in gewissem Sinne also $\lim\limits_{n\to\infty} I_n = I$.

Die Konvergenzgeschwindigkeit ist jedoch relativ gering. Der Fehler $|I - I_n|$ verhält sich wie $\frac{1}{\sqrt{n}}$ für große n (siehe Konvergenzgeschwindigkeit im Gesetz der großen Zahlen. Das Gesetz vom iterierten Logarithmus (Josef Steinebach, Uni Köln) und Gesetz des iterierten Logarithmus für Details). Die Trapezregel liefert hingegen beispielsweise einen Fehler in der Größenordnung $\frac{1}{n^2}$. Aber: Möchte man mehrdimensionale Integrale berechnen, so bleibt beim Monte-Carlo-Verfahren der Fehler in der Größenordnung $\frac{1}{\sqrt{n}}$, während er bei der Trapezregel auf $n^{-\frac{2}{d}}$ steigt (d ist die Dimension des Integrationsbereichs).

Neben der niedrigen Konvergenzgeschwindigkeit für eindimensionale Integrale ist ein weiter Nachteil des Monte-Carlo-Verfahrens, dass bei jeder Auswertung des Integrals ein etwas anderer Wert ermittelt wird.

10 Praktikum 1

10.1 JupyterLab

Aufgabe 1.1

Starten Sie JupyterLite, indem Sie auf den Link klicken, und machen Sie sich mit der Umgebung vertraut.

Aufgabe 1.2

Starten Sie ein JupyterLab auf mybinder.org, indem Sie auf den Link klicken, und machen Sie sich mit der Umgebung vertraut.

Aufgabe 1.3 (optional)

Installieren Sie JupyterLab lokal auf Ihrem Computer. Hinweise dazu finden Sie auch im Kapitel Install Jupyter Locally aus Data Science and Artificial Intelligence for Undergraduates.

> **Nebenbemerkung**
>
> Das E-Book Data Science and Artificial Intelligence for Undergraduates wurde für Studienanfänger im Fach Data-Science geschrieben, welche über keinerlei Programmierkenntnisse verfügen. Deshalb sind einige Erklärungen etwas ausführlicher gehalten als für unsere Zwecke nötig. Im Folgenden wird diese Quelle mit **DSAI** abgekürzt.

10.2 Python

Aufgabe 1.4

Machen Sie sich mit den Grundlagen von Python vertraut, z.B. indem Sie das Kapitel Crash Course aus DSAI überfliegen.

Aufgabe 1.5

Machen Sie sich mit den Grundlagen von NumPy vertraut, z.B. indem Sie das Kapitel Efficient Computations with NumPy aus DSAI überfliegen.

Aufgabe 1.6

Machen Sie sich mit den Grundlagen von Matplotlib vertraut, z.B. indem Sie das Kapitel Matplotlib Basics aus DSAI überfliegen.

10.3 Let's code!

Aufgabe 1.7

Zeichnen Sie mit `matplotlib.pyplot.plot` einen Kreis und positionieren Sie 12 Punkte gleichmäßig auf diesem wie bei einer Uhr. Lösen Sie diese Aufgabe einmal ohne NumPy (nur `math`-Paket) und einmal mit NumPy.

```
# Lösung ohne NumPy
```

```
# Lösung mit NumPy
```

Aufgabe 1.8 (optional)

Nutzen Sie die IPython-Magic `%%timeit` (ohne Argumente) um die Laufzeit beider Varianten zu vergleichen. Verzichten Sie dabei auf das Plotten des Kreises, berechnen Sie jeweils nur die notwendigen Koordinaten.

```
%%timeit

# Berechnungen ohne NumPy
```

```
%%timeit

# Berechnungen mit NumPy
```

11 Praktikum 2

11.1 Gleitkommazahlen

Bezeichnen im Folgenden mit “float8” binäre Gleitkommazahlen mit Mantissenlänge 5 Bit (einschließlich Hidden-Bit) und Exponentenlänge 3 Bit. Das erste Bit codiert das Vorzeichen, dann folgt der Exponent, dann die Mantisse.

Aufgabe 2.1 (KTA)

Welcher Dezimalzahl entsprechen die folgenden binär dargestellten float8-Zahlen?

- **(a)** 00110000
- **(b)** 00111000
- **(d)** 00010000

Aufgabe 2.2 (KTA)

Finden Sie die Dezimaldarstellung für folgende normalisierte float8-Zahlen:

- **(a)** die kleinste positive,
- **(b)** die größte positive.

Aufgabe 2.3 (KTA)

Welche float8-Zahl entspricht am besten den folgenden Dezimalzahlen (im Sinne der nach IEEE 754 üblichen Rundungsregel)?

- **(a)** 2
- **(b)** 2.0625
- **(c)** 2.1
- **(d)** 2.1875

11.2 Patriot-Beispiel

Aufgabe 2.4 (KTA)

Erklären Sie die Unterschiede in den sechs Ausgaben des folgenden Codes (vgl. Patriot-Beispiel).

```
import numpy as np
```

```
from sympy import N

def print_float(x):
    print(f'{x:.40f}')

a = 3600000
b = 0.1

print_float(  np.float16(a) * np.float16(b)              )
print_float(  np.float64(a) * np.float64(np.float16(b))  )
print_float(  np.float32(a) * np.float32(b)              )
print_float(  np.float64(a) * np.float64(np.float32(b))  )
print_float(  np.float64(a) * np.float64(b)              )
print(N(f'{a:.70f} * {b:.70f}', 70))
```

```
inf
359912.1093750000000000000000000000000000000000
360000.0000000000000000000000000000000000000000
360000.0053644180297851562500000000000000000000
360000.0000000000000000000000000000000000000000
360000.0000000000199840144432528177276253700256347656250000000000000000
```

```
/tmp/ipykernel_90757/3638290620.py:10: RuntimeWarning: overflow
↪  encountered in cast
  print_float(  np.float16(a) * np.float16(b)              )
```

11.3 Daten einlesen

Die folgenden Aufgaben dienen der Vorbereitung späterer Praktikumsaufgaben (Code aufheben!).

Aufgabe 2.4

Die Textdatei test.txt enthält pro Zeile eine Gleitkommazahl. Lesen Sie diese Zahlen in eine Python-Liste ein und geben Sie Minimum und Maximum aus.

```
# Ihre Lösung
```

Aufgabe 2.5

Schreiben Sie eine rekursive Funktion, die das Minimum in der List der vorhergehenden Aufgabe findet, indem Sie die Liste halbiert, dann das Minimum in jeder Hälfte bestimmt

(rekursiver Aufruf!) und dann das Minimum der beiden Minima der Hälften bestimmt. Testen Sie diese Funktion mit der Liste der Voraufgabe und auch mit anderen Listen (Randfälle!). Auch Listen mit ungerader Elementanzahl sollen geeignet verarbeitet werden.

```
# Ihre Lösung
```

12 Praktikum 3

12.1 Warm-Up

Aufgabe 3.1 (KTA)

Wir rechnen mit dezimalen Gleitkommazahlen $m\,10^e$, wobei m höchstens 3 Stellen hat und $e \geq 0$ gelten soll. Gerundet wird stets mathematisch. Berechnen Sie die folgenden Ausdrücke indem Sie vor und nach jeder Rechenoperation runden. Vergleichen Sie mit dem exakten Ergebnis und begründen Sie den Unterschied.

- **(a)** $100.5 - 0.4$
- **(b)** $210.51 - 209.49$
- **(c)** $(1000 + 4) + 4) + 4$
- **(d)** $1000 + ((4 + 4) + 4)$

Aufgabe 3.2 (KTA)

Bestimmen Sie die Konditionszahl für das Addieren von n Summanden. In welchen Situationen ist die Kondition gut?

12.2 Stabilität bei langen Summen

Aufgabe 3.3

Implementieren Sie das sequentielle Summieren aller Zahlen in einer Liste sowie das paarweise rekursive Summieren jeweils als Funktion. Verwenden Sie nur den `+`-Operator, nicht die `sum`-Funktion von Python und auch keine anderen Summationsfunktionen. Auch Listen mit ungerader Elementanzahl sollen verarbeitet werden können.

```
# Ihre Lösung
```

Aufgabe 3.4

Erzeugen Sie mit dem folgenden Code eine Liste von Zufallszahlen. Diese wird in verschiedenen Varianten erstellt:

- unsortierte Liste `numbers`,
- aufsteigend sortierte Liste `numbers_sorted`,
- nach Betrag aufsteigende sortierte Liste `numbers_abs_sorted`,

- geteilt in zwei aufsteigend sortierte Listen `neg` und `pos`, die nur negative bzw. nur positive Werte enthalten,
- geteilt in zwei nach Betrag aufsteigend sortierte Listen `neg_inv` und `pos`, die nur negative bzw. nur positive Werte enthalten.

```
import numpy as np

# make list of random numbers
rng = np.random.default_rng(42)  # random number generator
numbers = [float(x) for x in rng.uniform( -1, 1, 2 ** 20)]

# sort and sort by absolute value
numbers_sorted = sorted(numbers)
numbers_abs_sorted = sorted(numbers, key=abs)

# split by sign
if numbers_sorted[0] > 0:
    split_at = 0
else:
    for i, x in enumerate(numbers_sorted):
        if x > 0:
            split_at = i
            break
neg = numbers_sorted[:split_at]
pos = numbers_sorted[split_at:]

# sort negative numbers by absolute value
neg_inv = neg[:: -1]
```

Berechnen Sie für diese Listen bzw. Listenpaare die Summe aller Zahlen einmal sequentiell und einmal rekursiv. Geben Sie alle Ergebnisse so aus, dass Sie die Werte gut visuell vergleichen können. Geben Sie mindestens 17 Dezimalziffern aus.

```
# Ihre Lösung
```

Aufgabe 3.5 (KTA)

Welchen Wert halten Sie für den genauesten? Begründen Sie Ihre Wahl indem Sie insbesondere Erklärungen für die auftretenden Abweichungen bei folgenden Lösungspaaren angeben:

1. sequentiell, unsortiert vs. rekursiv, unsortiert
2. sequentiell, unsortiert vs. sequentiell, sortiert,

3. rekursiv, unsortiert vs. rekursiv, sortiert,
4. sequentiell, unsortiert vs. sequentiell, betragssortiert,
5. rekursiv, unsortiert vs. rekursiv, betragssortiert,
6. sequentiell, sortiert vs. sequentiell, negativ/positiv getrennt und sortiert,
7. sequentiell, sortiert vs. sequentiell, negativ/positiv getrennt und betragssortiert,
8. rekursiv, sortiert vs. rekursiv, negativ/positiv getrennt und betragssortiert,
9. rekursiv, negativ/positiv getrennt und sortiert vs. rekursiv, negativ/positiv getrennt und betragssortiert.

Aufgabe 3.6

Vergleich Sie den vertrauenswürdigsten Wert aus der vorhergehenden Aufgabe mit dem Wert, welchen `sum(numbers)` liefert. Gibt es einen Unterschied? Wenn ja: Wie groß ist dieser im Vergleich zum Fehler in den einzelnen Summanden (welche durch Runden von reellen Zahlen auf 64-Bit-Gleitkommazahlen entstanden sind)?

12.3 Approximation von π

Aufgabe 3.7 (KTA)

Mit jeder der beiden rekursiv definierten Zahlenfolgen

$$a_0 := \frac{1}{\sqrt{3}}, \qquad a_{i+1} := \frac{\sqrt{a_i^2 + 1} - 1}{a_i} \quad \text{für } i = 1, 2, \ldots \tag{12.1}$$

und

$$b_0 := \frac{1}{\sqrt{3}}, \qquad b_{i+1} := \frac{b_i}{\sqrt{b_i^2 + 1} + 1} \quad \text{für } i = 1, 2, \ldots \tag{12.2}$$

kann π aufgrund

$$\pi = 6 \lim_{i \to \infty} 2^i \, a_i \qquad \text{bzw.} \qquad \pi = 6 \lim_{i \to \infty} 2^i \, b_i \tag{12.3}$$

beliebig genau approximiert werden.

Überzeugen Sie sich zunächst, dass $a_i = b_i$ für alle i gilt.

Aufgabe 3.8 (KTA)

Untersuchen Sie die Kondition für die Berechnung von a_{i+1} aus a_i. Ist für die Kondition bei der alternativen Berechnung ein anderes Ergebnis zu erwarten?

Aufgabe 3.9 (KTA)

Untersuchen Sie die Stabilität beim Berechnen von a_{i+1} und b_{i+1} aus a_i bzw. b_i mit dem jeweils direkt aus der Formel ableitbaren Algorithmus.

Aufgabe 3.10

Implementieren Sie beide Algorithmen aus der vorhergehenden Aufgabe. Vergleichen Sie die erzielten Approximationen für π mit dem genauen Wert

$$\pi = 3.14159265358979323846264338\ldots. \tag{12.4}$$

Passen diese Ergebnisse zu den Stabilitätsaussagen aus der vorhergehenden Aufgabe?

13 Praktikum 4

13.1 Kovergenz des Newton-Verfahrens

Aufgabe 4.1 (KTA)

Führen Sie zwei Schritte des Newton-Verfahrens für die Funktion

$$f(x) = x^3 - 2\,x + 2 \tag{13.1}$$

mit dem Startwert $x_0 = 1$ manuell aus. Ist bei Fortführung der Iteration mit Konvergenz zu rechnen?

Aufgabe 4.2 (KTA)

Führen Sie zwei Schritte des Newton-Verfahrens für die Funktion

$$f(x) = \frac{x}{x^2 + 2} \tag{13.2}$$

mit dem Startwert $x_0 = 2$ manuell aus. Ist bei Fortführung der Iteration mit Konvergenz zu rechnen? Was passiert bei $x_0 \approx \sqrt{2}$?

Aufgabe 4.3 (KTA)

Führen Sie einen Schritt des Newton-Verfahrens für die Funktion

$$f(x) = \sqrt{25 - x^2} - 1 \tag{13.3}$$

mit dem Startwert $x_0 = 3$ manuell aus. Sind weitere Schritte möglich?

13.2 Nichtlineare Gleichungen mit SciPy

Aufgabe 4.4

Lösen Sie die Gleichung

$$x + \sin x = -1 \tag{13.4}$$

mittels `scipy.optimize.bisect` und `scipy.optimize.newton`. Welche Ursachen können zu Abweichungen zwischen den beiden Lösungen führen?

```
# Ihre Lösung
```

Aufgabe 4.5

Implementieren Sie das Bisektionsverfahren selbst und lösen Sie damit die Gleichung aus der vorhergehenden Aufgabe. Vergleichen Sie die Iterationsanzahlen bei Ihrer Implementierung mit der Anzahl der SciPy-Implementierungen von Bisektion und Newton-Verfahren.

```
# Ihre Lösung
```

Aufgabe 4.6

Lösen Sie das Gleichungssystem

$$x^3 - 3\,x\,y^2 = 1$$
$$x^2\,y - y^3 = 0$$

mittels `scipy.optimize.root(method='broyden1')`. Wählen Sie mehrere verschiedene Startpunkte.

Zusatz: Finden Sie heraus, was diese Funktion tut.

```
# Ihre Lösung
```

14 Praktikum 5

14.1 Kondition einer Matrix

Aufgabe 5.1

Berechnen Sie die Normen $\|A\|_1$, $\|A\|_2$ und $\|A\|_\infty$ sowie die Konditionszahlen $\kappa_1(A)$, $\kappa_2(A)$ und $\kappa_\infty(A)$ mit `numpy.linalg.norm` bzw. `numpy.linalg.cond` für die Matrizen

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}, \tag{14.1}$$

$$A = \begin{bmatrix} 0.99 & 1 \\ 1 & 1 \end{bmatrix} \tag{14.2}$$

und

$$A = \begin{bmatrix} 1 & 0.99 & 0 \\ 1 & 0.01 & 2 \\ 0.5 & 0 & 1 \end{bmatrix}. \tag{14.3}$$

Ist die Konditionszahl immer groß, wenn die Matrixspalten fast linear abhängig sind?

```
# Ihre Lösung
```

Aufgabe 5.2

Berechnen Sie die Konditionszahl der Hilbert-Matrix für $n = 1, \ldots, 30$ mit `numpy.linalg.cond` und stellen Sie die Abhängigkeit von n grafisch dar (`matplotlib.pyplot.semilogy` könnte nützlich sein).

Stellen Sie zum Vergleich auch die Punkte $(n, \mathrm{e}^{3.4\,(n-1)})$ dar. Warum wachsen die berechneten Konditionszahlen nicht weiter? Finden Sie dazu heraus, wie NumPy die Konditionszahl berechnet (Doku!). Tipps:

- Die Singulärwerte von A sind gerade die Wurzeln der Eigenwerte von $A^{\mathrm{T}}\,A$.
- Der größte Singulärwert der Hilbert-Matrizen bleibt bei $n \to \infty$ beschränkt unter 4.
- Irgendwann sollten Sie bei der Dokumentation von LAPACK ankommen.

```
# Ihre Lösung
```

Aufgabe 5.3 (KTA)

Die Konditionzahl der Matrix A sei $\kappa_2(A) = 20$. Wie groß darf der relative Fehler in b beim exakten Lösen von $A\,x = b$ sein, damit der relative Lösungsfehler höchstens 1 Prozent beträgt?

Ist diese Fehlerschranke erfüllt, wenn statt der exakten rechten Seite $b = [1, 2, 1]^{\mathrm{T}}$ die fehlerbehaftete rechte Seite $\tilde{b} = [1.001, 1.999, 1]^{\mathrm{T}}$ verwendet wird?

14.2 Lineare Gleichungssysteme

Aufgabe 5.4

Lösen Sie das Gleichungssystem

$$\begin{aligned} 50\,x_1 + 20\,x_3 &= 2 \\ 2\,x_1 + x_2 + x_3 &= 0 \\ x_1 + 2\,x_2 + 3\,x_3 &= 1 \end{aligned} \tag{14.4}$$

mit `numpy.linalg.solve`.

Welches Lösungsverfahren kommt hier zum Einsatz?

```
# Ihre Lösung
```

Aufgabe 5.5

Führen Sie für das Gleichungssystem in der vorhergehenden Aufgabe eine Äquilibrierung (Vorkonditionierung) durch. Lösen Sie das System dann nochmals und vergleichen Sie die Lösung mit der zuvor erhaltenen.

Vergleichen Sie auch die Konditionszahlen vor und nach der Äquilibrierung.

```
# Ihre Lösung
```

15 Praktikum 6

15.1 Methode der kleinsten Quadrate

Aufgabe 6.1 (KTA)

Die drei Punkte $(1, 1)$, $(2, 3)$, $(3, 8)$ sollen bestmöglich (im Sinne der Methode der kleinsten Quadrate) durch eine Parabel mit Scheitelpunkt im Koordinatenursprung angenähert werden. Stellen Sie zunächst die Zielfunktion für das Minimierungsproblem auf. Berechnen Sie anschließend die Lösung und geben Sie die gefundene Funktion an.

Aufgabe 6.2

Die Daten aus Höhenmessungen entlang einer Strecke liegen als Textdateien `positions.txt` (jede Zahl beschreibt die Enfernung von einem Startpunkt entlang einer vorgegebenen Strecke in Meter) und `altitudes.txt` (jede Zahl beschreibt die Höhe zur entsprechenden Positionsangabe) vor. Die Daten wurden aus mehreren GPS-Tracks gewonnen.

Der Höhenverlauf soll als stückweise lineare Funktion, also als Summe

$$x \mapsto \sum_{l=1}^{m} a_l \, h_l(x) \tag{15.1}$$

von Hütchenfunktionen h_l auf einem äquidistanten Gitter dargestellt werden.

Vervollständigen Sie dazu den folgenden Quellcode, fügen Sie also Code zum Einlesen der Daten ein (vgl. Praktikum 2) und implementieren Sie das Kleinste-Quadrate-Verfahren unter Nutzung von `numpy.linalg.lstsq`.

```
import numpy as np
import matplotlib.pyplot as plt

def piecewise_linear(grid, coeffs, x):
    # grid und coeffs beschreiben eine stückweise lineare Funktion
    ↪ (zwei NumPy-Arrays gleich Länge).
    # x ist ein NumPy-Array, das die Stellen enthält, an denen die
    ↪ Funktion ausgewertet werden soll.
    return np.interp(x, grid, coeffs, left=0, right=0)

def hat(grid, l, x):
    # grid muss NumPy-Array mit äquidistanten Stützstellen enthalten
    ↪ (z.B. np.linspace(a, b, m)).
    # l ist die Position des Hütchens im Gitter (also Werte von 0 bis m
    ↪ -1)
    # x ist ein NumPy-Array, das die Stellen enthält, an denen die
    ↪ Hütchenfunktion ausgewertet werden soll.
```

```
    h = np.zeros(len(grid))
    h[l] = 1
    return piecewise_linear(grid, h, x)

def load_data():
    # TODO: Werte aus positions.txt und altitudes.txt einlesen und als
    ↪  zwei NumPy -Arrays zurückgeben.

def least_squares_pwl(grid, x, y):
    # grid ist Gitter der Hütchenfunktionen (NumPy -Array).
    # x und y sind die Datenpunkte (zwei NumPy -Arrays gleicher Länge).
    # TODO: Koeffizienten der bestmöglichen stückweise linearen
    ↪  Funktion zurückgeben.

# Daten einlesen
positions, altitudes = load_data()

# Gitter für die Hütchenfunktionen (ein Hütchen pro Gitterpunkt)
grid = np.linspace(positions.min(), positions.max(), 20)

# Kleinste -Quadrate -Lösung
coeffs = least_squares_pwl(grid, positions, altitudes)

# Plot
fig, ax = plt.subplots()
ax.plot(positions, altitudes, 'o', markersize=3)
ax.plot(grid, piecewise_linear(grid, coeffs, grid), ' -', linewidth=2)
ax.set_xlabel('Position')
ax.set_ylabel('Höhe')
plt.show()
```

15.2 Polynominterpolation

Aufgabe 6.3

Finden Sie mit `scipy.interpolate.lagrange` ein Interpolationspolynom 4-ten Grades für die Funktion $f(x) = \sin 3x$ auf dem Intervall $[0, 1]$. Lösen Sie diese Aufgabe sowohl mit äquidistanten Stützstellen als auch mit den Tschebyscheff-Stützstellen.

```
# Ihre Lösung
```

Aufgabe 6.4 (KTA)

Geben Sie das Interpolationspolynom zu den Punkte $(0, 0)$, $(1, 1)$, $(2, 32)$, $(3, 243)$ in Langrange-Form und in Newton-Form an.

Aufgabe 6.5 (KTA)

Das in der vorhergehenden Aufgabe gefundene Polynom kann als Approximation der Funktion $f(x) = x^5$ auf dem Intervall $[0, 3]$ durch Interpolation mit äquidistanten Stützstellen angesehen werden.

Welche Schranke für den Approximationsfehler liefert die Abschätzung

$$\|f - p\|_\infty \leq \frac{1}{n!} \|N_n\|_\infty \|f^{(n)}\|_\infty \tag{15.2}$$

hier (N_n ist das Newton'sche Basispolynom der Ordnung n)?

16 Praktikum 7

16.1 Splines

Aufgabe 7.1 (KTA)

Die durch $f(x) = \sin(\pi\, x)$ soll durch einen quadratischen Spline zu den Stützstellen 0, 1, 2 approximiert werden. Geben Sie die Formel(n) für diesen Spline allgemein an und stellen Sie die Systemmatrix sowie die rechte Seite des Gleichungssystems zur Berechnung der darin auftretenden Koeffizienten auf. Wählen dabei geeignete Bedingungen an den Enden des Intervalls $[0, 2]$.

16.2 Numerische Integration

Aufgabe 7.2 (KTA)

Berechnen Sie

$$\int_0^1 \sin(\pi\, x)\, dx \tag{16.1}$$

näherungsweise mit der (nicht summierten) Simpson-Regel. Wie groß ist die Abweichung zum exakten Wert? Welche Fehlerschranke liefert die Abschätzung

$$|I - I_n| \leq \frac{|b-a|}{m!}\, h^m\, \|f^{(m)}\|_\infty \tag{16.2}$$

wobei, h der Abstand der Stützstellen und $m = 3$ ist?

Aufgabe 7.3

Implementieren Sie die numerische Integration mittels summierter Trapezregel. Berechnen Sie damit das Integral aus der vorhergehenden Aufgabe für verschiedene Stützstellenanzahlen und vergleichen Sie die Ergebnisse mit denen, die `numpy.trapezoid` liefert.

Berechnen Sie auch das Integral über dem Intervall $[0, 2]$. Erklären Sie eventuelle Unterschiede zwischen Ihren Ergebnissen und den von NumPy.

```
# Ihre Lösung
```

Aufgabe 7.4

Berechnen Sie das Integral

$$\int_0^1 x^m\, \mathrm{e}^x\, \mathrm{d}x \tag{16.3}$$

numerisch mittels Trapezregel für s$m = 1, \ldots, 20$. Vergleichen Sie die erhaltenen Werte mit denen im Section 1.3.

```
# Ihre Lösung
```

17 Download

Zum Offline-Lesen steht das Vorlesungsskript als PDF-Datei zur Verfügung. Aus naheliegenden Gründen sind Animationen dort nicht enthalten. Generall ist die Konvertierung nach PDF noch etwas experimentell, sodass Darstellungsfehler nicht auszuschließen sind.

Download PDF-Datei

Aufgabenblätter für das Praktikum können Sie auch einzeln als PDF-Datei oder als Jupyter-Notebook auf der entsprechenden Seite runterladen (Button oben rechts).

18 Organisatorisches

Modulbeschreibung in Modulux

18.1 Ablauf der Veranstaltung

- wöchentlich eine Einheit Vorlesung oder Praktikum (im Wesentlichen im Wechsel)
- **Vorbereitung:** Skript vor Vorlesung und Praktikum mindestens überfliegen!
- **Nachbereitung:** Alles verstanden? Rückstände aus dem Praktikum nachholen!

Datum	
20.03.2026	Vorlesung
27.03.2026	Praktikum
03.04.2026	-
10.04.2026	~~Praktikum~~ Vorlesung
17.04.2026	Vorlesung
24.04.2026	Praktikum
01.05.2026	-
08.05.2026	~~Praktikum~~ Vorlesung
15.05.2026	-
22.05.2026	Praktikum
29.05.2026	Vorlesung
05.06.2026	Praktikum
12.06.2026	Vorlesung
19.06.2026	Praktikum
26.06.2026	Vorlesung
03.07.2026	Praktikum

18.2 Praktika

- Praktika im Computer-Pool (oder eigener Computer)
- benötigte Software: Webbrowser (je nach Vorlieben eine lokale Python/Jupyter-Installation)

18.3 Prüfung

- schriftliche Klausur, 120 Minuten
- Beispielaufgaben sind bei den Praktikumsaufgaben gekennzeichnet

- keine Hilfsmittel außer Papier, Stifte, einfacher Taschenrechner (kein CAS, keine Grafik)
- Klausur auch ohne Taschenrechner gut lösbar
- Beispielklausur (dient nur zur groben Orientierung; grundsätzlich können alle behandelten Themen und Aufgabentypen Teil der Klausur sein)

18.4 Hinweise zum Skript

- Download als PDF-Datei möglich
- Aufgabenzettel für das Praktikum auch einzeln als PDF-Datei
- Aufgabenzettel für das Praktikum als Jupyter-Notebook (Lösen der Aufgaben direkt im Dokument)

Viele Rechnungen, Beweise, Herleitungen, Zeichnungen fehlen im Skript und werden an der Tafel geliefert. Deshalb:

- mitschreiben oder
- Fotos von der Tafel (nicht vom Vorlesenden) machen
- (optional) Fotos zum Verlinken im Skript abliefern (Upload im OPAL-Kurs)
- bitte keine Videos (entweder professionell oder gar nicht; ersteres aktuell nicht leistbar)

Die Technik hinter dem Skript ist noch in Entwicklung und teils etwas “beta”. Hinweise zu Problemen und Verbesserungsvorschläge gern an den Vorlesenden!

Literaturverzeichnis

[Hig02] Nicholas J. Higham. Accuracy and Stability of Numerical Algorithms. Society for Industrial, 1 2002.

[Sau78] Werner Sautter. Fehleranalyse für die Gauß-Elimination zur Berechnung der Lösung minimaler Länge. Numerische Mathematik, 30(2):165–184, 6 1978.

[VI86] M. N. Vrahatis and K. I. Iordanidis. A rapid Generalized Method of Bisection for solving Systems of Non-linear Equations. Numerische Mathematik, 49(2–3):123–138, 3 1986.