Algorithmen und Datenstrukturen

Praktische Einführung und Programmierung

Stefan Bosse

Universität Koblenz - FB Informatik

Numerische Algorithemen und Mathematik

Mathematik ist Algorithmik!

Numerische Algorithemen und Mathematik

Mathematik ist Algorithmik!

Das Lösen von mathematischen Problemen ist Algorithmik

Numerische Algorithemen und Mathematik

Mathematik ist Algorithmik!

Das Lösen von mathematischen Problemen ist Algorithmik

Das Lösen von mathematischen Problemen mit Computer Algorithmen ist seinerseits ein Problem!

Mathematik

Algorithmen

Funktionen

Datenstrukturen

Funktionen (!), Vektoren, Matrizen, Tensoren usw.

Künstliche Intelligenz

Algorithmen

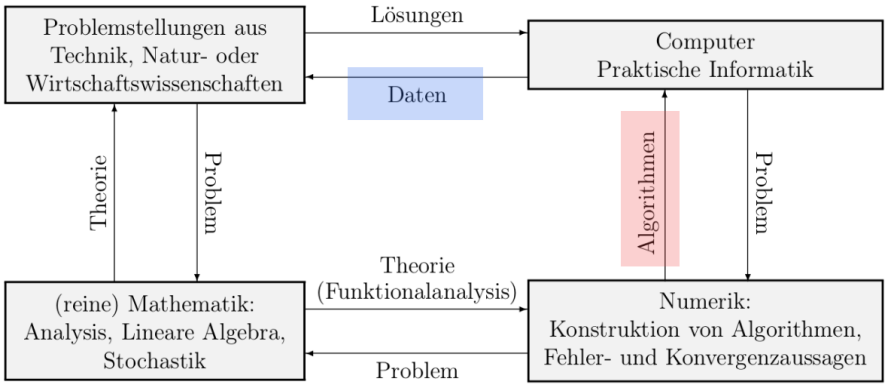
Iterative Trainingsalgorithmen, Vereinfachung

Datenstrukturen

Funktionen, Datengraphen und Bäume, Funktionsgraphen

Numerik

Man spricht von numerischen Algorithmen wenn diese basierend auf mathematischen Prinzipien und optional auf naturwissenschaftlichen Modellen "natürliche" Daten verarbeiten, also i.A. gemessene Daten.



[numalg]

Abb. 1. Beziehung der Numerik zu anderen Bereichen:

Numerik

Das **Ziel der Numerik** ist die Konstruktion ökonomischer (effizienter) und stabiler Algorithmen. Speziell gilt es, mögliche Fehlerquellen zu berücksichtigen. Diese ergeben sich durch Modellierungsfehler, durch Fehler in den Eingangsdaten, durch Fehler im Algorithmus, und durch Diskretisierungsfehler in der Numerik.

Numerik: Anwendungen und Probleme

1. Arithmetik mit diskreten Zahlensystemen (Ganzzahl-, Festpunkt-, Fliesskommarithmetik)
2. Vektor- und Matrixalgebra
3. Verfahren zur Lösung linearer Gleichungssysteme
   - Gauß–Elimination, LR–Zerlegung und QR–Zerlegung von Matrizen, Cholesky–Verfahren
4. Ausgleichsrechnung (Least–Squares–Approximation)
   - Ausgleichsprobleme über Normalengleichungen
   - QR–Zerlegung von Matrizen
5. Berechnung von Eigenwerten und Eigenvektoren
   - Principle Component Analysis ⇒ ML
   - QR–Zerlegung von Matrizen

Numerik: Anwendungen und Probleme

6. Iterative Verfahren zur Lösung nichtlinearer Gleichungen
7. Nichtlineare Ausgleichsprobleme
8. Interpolation mit Polynomen
   - Support Vector Machines ⇒ ML
9. Splinefunktionen:
10. Numerische Integration (Quadratur): Selbst bei gegebenem Integranden kann die In-tegration einer Funktion theoretisch häufig nicht durchgeführt werden. Zur numerischen Berechnung greift man daher entweder auf Punktauswertungen des Integranden zu oder approximiert den Integranden durch einfacher zu integrierende Funktionen wie Splines
    - Newton-Cotes-Formel
    - Gauss Formel
11. Approximierte Berechnung von Gradienten und Gradientengleichungen
12. Training von Künstlichen Neuronalen Netzwerken (Gradientverfahren) ⇒ ML

Numerik: Anwendungen und Probleme

Da sich viele konstruktive Verfahren aus der Theorie nicht zur praktischen Durchführung auf Computern eignen (Numerik), erfordert für schwierige Probleme die Entwicklung guter numerischer Verfahren umfangreiche Kenntnisse und große Erfahrung.

- Wir werden einige überraschende Ergebnisse sehen und dass Numerik (und die Ergebnisse daraus) ähnlich verwirrend und unerwartet sein können wie bei der Betriebssystemprogrammierung!

Numerik: Fehleranalyse

Wir haben bei numerischen Problemen folgende Randbedingungen zu berücksichtigen:

1. Kondition
2. Rundungsfehler
3. Stabilität

[algomat]

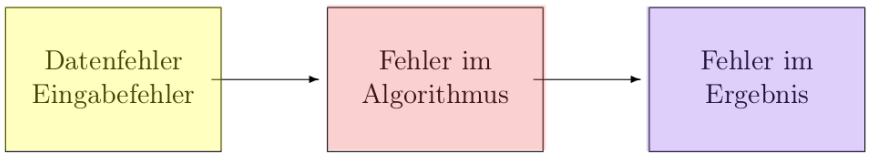


Abb. 2. Beim Ausführen von Rechnungen gibt es verschiedene Fehlerquellen

Numerik: Fehleranalyse

Kondition eines Problems

Die Kondition eines Problems gibt an, welche Genauigkeit man bei exakter Lösung bestenfalls erreichen kann: Ein Problem heißt gut konditioniert, wenn kleine Störungen der Eingangsdaten kleine Änderungen im Ergebnis bewirken, sonst schlecht konditioniert.

Rundungsfehler

In der Mathematik werden kontinuierliche und gar unendlich große Zahlenmenge betrachtet (reele Zahlen). **In der Numerik haben wir immer diskerete und begrenzte Wertemengen!**. Daher sind numerische Verfahren immer ungenauer als mathematische.

Stabilität

Iterative Algorithmen können eine "richtige" Lösung liefern (Konvergenz), können aber auch falsche Ergebnisse liefern (Divergenz). Man versucht Algorithmen stabil gegen Divergenz zu machen.

Numerik und Zahlensysteme

Mathematisch können wir zwei Basismengen von Zahlen unterscheiden:

Ganze Zahlen $\mathbb{N}$

Eine unendlich große Menge ganzer positiver und negativer Zahlen inklusive 0. Aber die Menge ist abzählbar! Es gibt keine weitere ganze Zahl in jedem Intervall $[x,x+1]$.

Reelle Zahlen $\mathbb{R}$

Eine unendlich große Menge kontinuierlicher positiver und negativer Zahlen inklusive 0. Aber die Menge ist nicht abzählbar! Es gibt unendlich viele weitere Zahlen in jedem Intervall $[x,x+\delta]$, egal wie klein man das Intervall macht.

Computer Zahlensysteme

Maschinenzahlen und Rundungsfehler

Maschinenzahlen werden durch Datenbits repräsentiert (kodiert).

Ganze Zahlen (Integer Datentyp)

Eine endlich große Menge ganzer positiver und negativer Zahlen inklusive 0. Es gibt keine weitere ganze Zahl in jedem Intervall $[x,x+1]$.

Festpunktzahlen (Fixed Point Datentyp)

Eine endlich große Menge diskreter positiver und negativer Zahlen inklusive 0. Aber es gibt **keine weiter Zahl** in einem kleinen Intervall $[x,x+\delta_{min}]$, δ_{min} ist absolut und hängt von der Anzahl der Datenbits und der absoluten Größe des Zahlenintervalls ab!
Eigentlich ganze Zahlen mit einem festen verschobenen Dezimalpunkt.

Gleitkommazahlen (Floating Point Datentyp)

Eine endlich große Menge diskreter positiver und negativer Zahlen inklusive 0. Aber es gibt **keine weiter Zahl** in einem kleinen Intervall $[x,x+\delta_{min}]$, δ_{min} ist relativ und hängt von der Anzahl der Datenbits ab!

Computer Zahlensysteme

Integer Zahlen

Die digitalen Zahlen (Kodierung) *d* kann man in dezimale mit der gewichteten Summe umrechnen (Binärzahlensystem mit Basis *b*=2):

$$a = (-1)^s \sum_{i=0}^{m-1} d_i \cdot b^i$$

[algomat]

m	Speicher	Zahlbereich	42 (Big Endian)	42 (Little Endian)
8	1 Byte	$0, \ldots, 2^8 - 1 (= 255)$	0010 1010	0101 0100
16	2 Bytes	$0, \ldots, 2^{16} - 1 (= 65535)$	0000 0000 0010 1010	0101 0100 0000 0000
32	4 Bytes	$0, \ldots, 2^{32} - 1 (= 4294967295)$	...	
64	8 Bytes	$0, \ldots, 2^{64} - 1 (\approx 1.84 \cdot 10^{19})$	...	

Tab. 1. Zahlenbereich hängt von der Anzahl Bits ab (*m*). Für das Vorzeichen kommt noch ein Bit hinzu (oder der Zahlenbereich verringert sich). Es gibt keinen Rundungsfehler!

Arithmetik

Es gibt die grundlegenden numerischen arithmetischen Operationen:

1. Addition (Subtraktion)
2. Multiplikation (Division)
3. Negierung

Die Reihenfolge von arithmetischen Operationen kann in der Numerik von elementarer Bedeutung sein. Top oder Flop! Vor allem bei Integer Zahlenarithmetik

```
a=1; b=10; c=100;
x1 = (a/b)*c
x2 = (a*c)/b
print(x1,x2)
```

Das Experiment (Hinweis: JavaScript verarbeitet alle Zahlen als Gleitkommazahlen, daher ist Umwandlung in Integer erforderlich)

Gleitkommazahlen

$$a = (-1)^s \sum_{i=0}^{m-1} d_i \cdot b^{-i+e}$$

- Es gibt eine Mantisse $\Sigma d_i b^{-i}$ und einen Exponenten e.
- Anders als bei der Festpunktdarstellung (die einfach nur reelen Zahlen in einen Ganzzahlbereich durch Skalierung verschiebt) ist hier die Position des Dezimalpunktes durch die Zahl e festgelegt.

Eigenschaften	single precision/float	double precision/double
Speichergröße	32 bit = 4 byte	64 bit = 8 byte
Bits der Mantisse	23	52
Stellen der Mantisse	24	53
Bits des Exponenten	8	11
Zahlbereich des Exponenten	$-126, \ldots, 127$	$-1022, \ldots, 1023$
Zahlbereich normalisiert	$1.2 \cdot 10^{-38}, \ldots, 3.4 \cdot 10^{38}$	$2.2 \cdot 10^{-308}, \ldots, 1.8 \cdot 10^{308}$
kleinste positive denorm. Zahl	$1.4 \cdot 10^{-45}$	$4.9 \cdot 10^{-324}$
Genauigkeit für Dezimalzahlen	$7-8$ Stellen	$15-16$ Stellen

Tab. 2. Eigenschaften verschiedener Datenformate bei Gleitkommazahlen

Rundungsfehler

- oder richtig ausgedrückt Diskretisierungsfehler bezüglich des Zahlensystems und deren Kodierung (da niemand explizit rundet)
- Rundungsfehler, Über- und Unterlauf und "Not a Number NaN" können bei arithmetischen Operationen auftreten.

Diskretisierung

- Die Dikretisierung von Integer Zahlen ist ohne Genauigkeitsverlust möglich, nur die Wertemenge ist begrenzt
- Die Diskretisierung von reelen Zahlen ist i.A. immer mit Genauigkeitsverlust und einer Einschränkung der Wertemenge verbunden (also schon hier RUndungsfehler möglich)

Rundungsfehler

Arithmetik

- Arithmetische Operationen wie Division können bei ganzen Zahlen zu erheblichen Rundungsfehlern führen (bis zu 100%), Addition und Multiplikation führen keine weiteren Rundungsfehler ein, können aber zum Überlauf bei Integer Zahlen führen!
- Arithmetische Operationen können bei Gleitkommazahlen (aber nicht reellen) zu Rundungsfehlern führen (bis zu 1%), und es kann zum Überlauf kommen!

Der Klassiker in der Numerik: Ein Quotient wird Null obwohl es mathematisch eine Zahl $\neq 0$ ergeben müsste.

- Im Bereich der ML Algorithmen spricht man vom verschwindenden Gradienten (auch ein Quotient mit Divisionsoperation)
- Bei Gleitpunktarithmetik gibt es eine Fehlerverstärkung bei den elementaren Rechenoperationen!

Funktionen und Schleifen

- Funktionsrekursion ist eine gängige Methode in der Mathematik um eine berechung auszudrücken.
- Programmatisch können wir in fast allen Programmiersprachen die Funktionrekursion nutzen, es gibt aber auch Nachteile (welche?)
- Man kann rekursive Funktionen in Schleifen transformieren und umgekehrt!

Rekursion

```
function fac(n) {
  if (n<2) return 1
  else return n*fac(n-1)
}
```

Schleife

```
y=1
for(i=2;i<=n;i++) {
  y=y*i
}
```

Algorithmus 1: Berechnung von e

[numalg pp 6]

Es gibt viele verschiedene Möglichkeiten, um die Eulersche Zahl *e* numerisch zu approximieren. Nachfolgendend führen wir vier verschiedene Varianten ein.

1. Grenzwert

$$e = \lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n$$

für x=1 berechnen.

⇒ Grenzwerte können nicht direkt prozedural (iterativ) algorithmisch gelöst werden (deklarative und symbolische Methodik erforderlich). Nur endliche Approximation möglich!

Algorithmus 1: Berechnung von e

```
n=100
e=pow(1+1/n,n)
print(e)
```

Algorithmus 1: Berechnung von e

2. Summe: Eine andere Möglichkeit ist, die Funktion

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

für x=1 zu berechnen.

```
e=0;N=100
function fac(n) { return n<2?1:n*fac(n-1) }
for(n=0;n<=N;n++) {
  e=e+1/fac(n)
}
print(e)
```

Was ändert sich algorithmisch und bei der Laufzeit im Vergelich zu Methode 1?

Algorithmus 1: Berechnung von e

Algorithmus 1: Berechnung von e

[numalg]

m	n	$e_m^{(1)}$	$\Delta_m^{(1)}$	$e_m^{(2)}$	$\Delta_m^{(2)}$
$m=1$	$n=10^1$	2.593742	$1.24 \cdot 10^{-1}$	2.000000	$7.18 \cdot 10^{-1}$
$m=2$	$n=10^2$	2.704814	$1.35 \cdot 10^{-2}$	2.500000	$2.18 \cdot 10^{-1}$
$m=3$	$n=10^3$	2.716924	$1.36 \cdot 10^{-3}$	2.666667	$5.16 \cdot 10^{-2}$
$m=4$	$n=10^4$	2.718146	$1.36 \cdot 10^{-4}$	2.708333	$9.95 \cdot 10^{-3}$
$m=5$	$n=10^5$	2.718268	$1.36 \cdot 10^{-5}$	2.716667	$1.62 \cdot 10^{-3}$
$m=6$	$n=10^6$	2.718280	$1.36 \cdot 10^{-6}$	2.718056	$2.26 \cdot 10^{-4}$
$m=7$	$n=10^7$	2.718282	$1.34 \cdot 10^{-7}$	2.718254	$2.79 \cdot 10^{-5}$
$m=8$	$n=10^8$	2.718282	$3.01 \cdot 10^{-8}$	2.718279	$3.06 \cdot 10^{-6}$
$m=9$	$n=10^9$	2.718282	$2.24 \cdot 10^{-7}$	2.718282	$3.03 \cdot 10^{-7}$
$m=10$	$n=10^{10}$	2.718282	$2.25 \cdot 10^{-7}$	2.718282	$2.73 \cdot 10^{-8}$
$m=11$	$n=10^{11}$	2.718282	$2.25 \cdot 10^{-7}$	2.718282	$2.26 \cdot 10^{-9}$
$m=12$	$n=10^{12}$	2.718523	$2.42 \cdot 10^{-4}$	2.718282	$1.73 \cdot 10^{-10}$
$m=13$	$n=10^{13}$	2.716110	$2.17 \cdot 10^{-3}$	2.718282	$1.23 \cdot 10^{-11}$
$m=14$	$n=10^{14}$	2.716110	$2.17 \cdot 10^{-3}$	2.718282	$8.15 \cdot 10^{-13}$
$m=15$	$n=10^{15}$	3.035035	$3.17 \cdot 10^{-1}$	2.718282	$5.06 \cdot 10^{-14}$
$m=16$	$n=10^{16}$	1.000000	$1.72 \cdot 10^{0}$	2.718282	$2.66 \cdot 10^{-15}$

Abb. 3. Vergleich der Ergebnisse von den beiden Varianten 1 und 2 zur Berechnung der Eulerschen Zahle *e* für verschiedene *n* (*n*=10 [m]). Es gibt eine Überraschung!

Algorithmus 1: Berechnung von e

Warum scheitert Verfahren 1 (numerisch!) für große n, aber nicht Verfahren 2?

Algorithmus 1: Berechnung von e

Warum scheitert Verfahren 1 (numerisch!) für große n, aber nicht Verfahren 2?

Hinweis: Wir haben diskrete und intervallbegrenzte Numerik!

Algorithmus 2: Approximierte Berechnung des Integrals

Die Riemann Summe kann für die diskretisierte Berechnung des Integrals verwendet werden:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{N} \Delta x_i f\left(x_i^\star\right)$$

$$De < yx_i = x_i - x_{i-1}$$

$$x_i^\star \in [x_{i-1}, x_i]$$

- Jetzt wird es spannend: Das Berechnungsproblem ist nicht eindeutig definiert da es auf die Auswahl eines x-Wertes innerhalb des Intervalls $[x_{i-1},x_i]$ ankommt:
  - $x^*=x_{i-1}$ ⇒ linke Riemann Summe
  - $x^*=x_i$ ⇒ rechte Riemann Summe
  - $x^*=(x_i+x_{i-1})/2$ ⇒ mittlere Riemann Summe

Algorithmus 2: Approximierte Berechnung des Integrals

[Wikipedia]

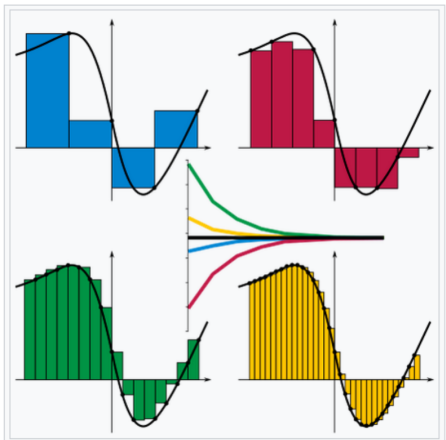


Abb. 4. Die Genauigkeit der approximierten Berechnung vom Integral hängt von der Partitionierung und der diskretisierten *x*-Auswahl ab

Algorithmus

```
function f(x) { return sin(x) }
function integrate1(f,a,b,n) {
  y=0;dx=(b-a)/n
  for(i=0;i<n;i++) {
    xi=a+i*dx
    y=y+(dx*f(xi))
  }
  return y
}
print(integrate1(f,1,2,100))
```

Alg. 1. Einfache Integralberechnung mit einer einzigen Stützstelle pro Diskretisierungsintervall

Algorithmus 2: Approximierte Berechnung des Integrals

Algorithmus 2: Approximierte Berechnung des Integrals

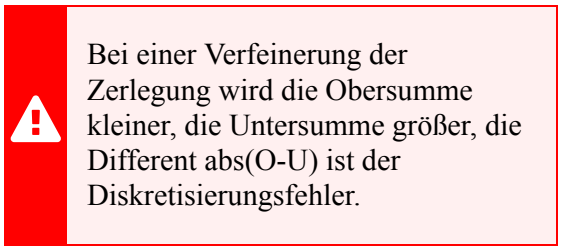
Einige Fragen:

1. Wie hängt die Rechenzeit (Einheitsoperationen) von n und $[a,b]$ ab?
2. Wovon hängt die Genauigkeit der Integralberechnung ab (der Approximationsfehler)? Nur von n?
3. Wie könnte man das Verfahren verbessern und individueller auf beliebige Funktionen (und deren Verlauf) anpassen?

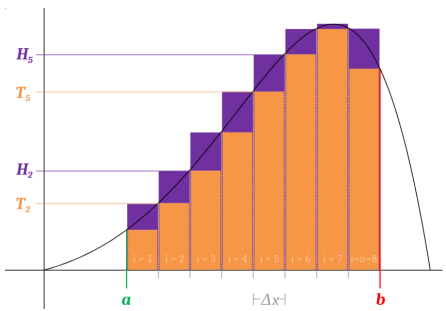
Algorithmus 2B: Approximierte Berechnung des Integrals

- Bisher haben wir nur einen Funktionswert pro Stützpunkt evaluiert hatten ist die Berechnung schwierig vorherzusagen, je nachdem welchen x Wert wir aus dem Intervall $[x_i, x_{i+1}]$ gewählt haben.

- Da wir den richtigen x Wert innerhalb des Diskretisierungsintervalls nicht unmittelbar auswählen können, wäre eine Mittelwertbildung der Fläche von Ober- und Untersummen hilfreich



Bei einer Verfeinerung der Zerlegung wird die Obersumme kleiner, die Untersumme größer, die Different abs(O-U) ist der Diskretisierungsfehler.



[Wikipedia]

Trapezregel

- Es werden Unter- und Obersumme gemittelt (Trapezregel)
  - Zwischen zwei Stützpunkten wird eine lineare Verbindung geschaffen, die Fläche des entstehenden Trapezes wird als Näherung des Integrals benutzt
  - Auch möglich mit weiteren Termen/Stützpunkten

[Wikipedia]

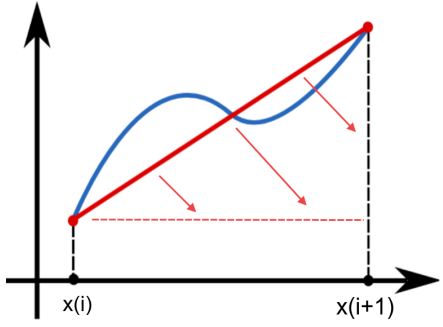


Abb. 5. Annäherung eines nichtlinearen Kurvenverlaufs durch eine lineare Funkion durch zwei Stützpunkte x_i und x_{i+1}

```
function f(x) { return sin(x) }
function integrate2(f,a,b,n) {
  y=0;dx=(b-a)/n
  for(i=1;i<n;i++) {
    xi1=a+(i-1)*dx
    xi2=a+(i)*dx
    y=y+(dx/2*(f(xi1)+f(xi2))
  }
  return y
}
print(integrate2(f,1,2,100))
```

Alg. 2. Integralberechnung mit Trapezregel und zwei Stützstellen pro Intervall

- Es können mehr als zwei (Sub)Stützpunkte zwischen zwei Schritten gewählt und berechnet werden, d.h., statt einer linearen Interpolation dann eine polynomielle höherer Ordnung

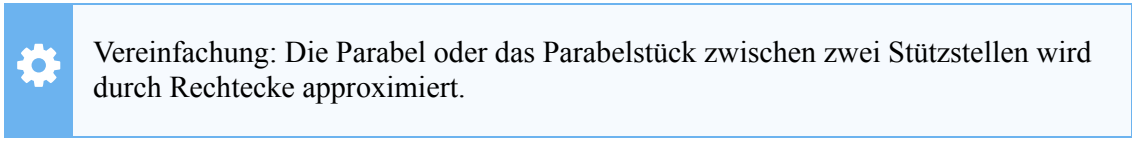
- Aber letztlich ist das eine Erhöhung der gesamten Anzahl der Stützpunkte *N* und bringt noch nicht die Balance zwischen Genauigkeit und Effizienz.

n	$\lambda_{0n}, \ldots, \lambda_{nn}$	Geläufige Namen
1	$\frac{1}{2}\ \frac{1}{2}$	Trapezregel
2	$\frac{1}{6}\ \frac{4}{6}\ \frac{1}{6}$	Simpson-Formel, Keplersche Faßregel
3	$\frac{1}{8}\ \frac{3}{8}\ \frac{3}{8}\ \frac{1}{8}$	Newtonsche 3/8-Regel, pulcherrima
4	$\frac{7}{90}\ \frac{32}{90}\ \frac{12}{90}\ \frac{32}{90}\ \frac{7}{90}$	Milne-Regel

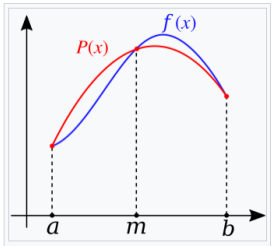
Abb. 6. Stützstellen zwischen [*x*,*x*+δ] und die Termgewichte

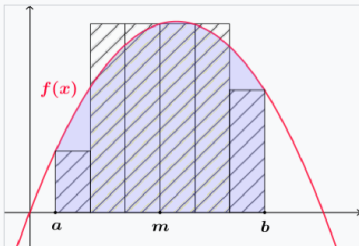
Simpson Regel

- Die Simpsonregel oder simpsonsche Formel (nach Thomas Simpson) ist ein Verfahren der numerischen Integration, bei dem eine Näherung zum Integral einer in einem Intervall schwer zu integrierenden Funktion berechnet wird, indem man die Funktion durch eine exakt integrierbare **Parabel** annähert.



Vereinfachung: Die Parabel oder das Parabelstück zwischen zwei Stützstellen wird durch Rechtecke approximiert.





[Wikipedia]

Abb. 7. Simpson Regel für die Integration von Funktionen

```
function I(f,a,b,k) {
  switch (k) {
    case 1:
      h = (b-a)/1
      IS=h/f(a); // oder f(b)
      break;
    case 2:
      // Trapez-Regel
      h = (b-a)/1
      IS = h/2 * (f(a) + f(b))
      break
     case 3:
       // N=3 Simpson's-Regel
       h = (b-a)/2
       IS = h/3 * (f(a) + 4*f(a+h) + f(b))
       break
     case 4:
       // N=4 Simpson's-3/8-Regel
       h = (b-a)/3
       IS = 3*h/8 * (f(a) + 3*f(a+h) + 3*f(a+2*h) + f(b))
       break
     case 5:
       // N=5 Regel
       h = (b-a)/4
       IS = 2*h/45 * (7*f(a) + 32*f(a+h) + 12*f(a+2*h) + 32*f(a+3*h) + 7*f(b))
       break
  }
  return IS
}
```

Alg. 3. Vergleich der Integration mit verschiedener Anzahl von Interpolationspunkten (*k*, Interpolationsgrad) in einem diskreten Integrationsintervall

```
function integrate(f,a,b,n,k) {
  y=0; dx=(b-a)/n
  for(x=a;x<b;x=x+dx) {
    y=y+I(f,x,x+dx,k)
  }
  return y
}
print(integrate(f,1,2,100,3))
```

Alg. 4. Wie zuvor muss jetzt noch eine iterative Summation für das gesamte Intervall [*a*,*b*] erfolgen. *k* ist der Interpolationsgrad.

[numbook]

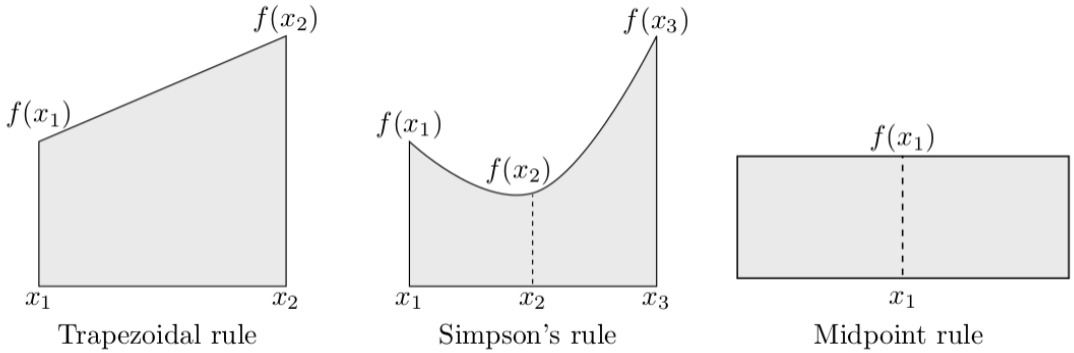


Abb. 8. Zusammenfassung der wichtigsten Integralapproximationen (ein Intervallschritt Δx ∈ [a,b])

Bisher haben wir nur mit äquidistanten Knoten gearbeitet und so ein einfaches und klares Formelrepertoire erhalten. Wollten wir höhere Genauigkeiten, so wurden einfach der Grad oder die Anzahl der Zusammensetzungen erhöht, um eine geringere Maschen-breite und einen als geringer abschätzbaren Fehler zu erhalten. Die Verfahren sind zwar recht einfach, doch rechnen diese bei schon längst vorhandener hoher Genauigkeit der Iteration immer noch weiter.

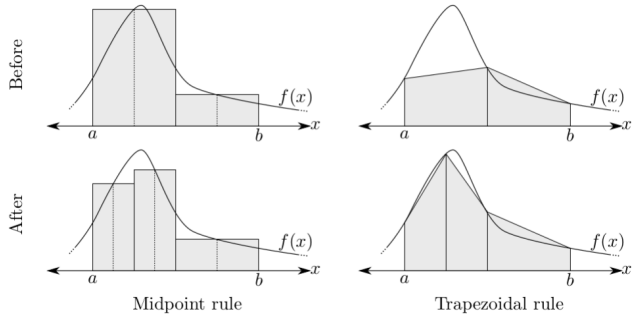


Abb. 9. Bessere Approximation (höhere Genauigkeit) bei dynamisch adaptiven Intervallen

Algorithmus 2C: Approximierte Berechnung des Integrals

Die Genauigkeit hängt von der Intervallbreite Δx und somit *n*, aber auch von dem Kurvenverlauf der zu integrierenden Funktion sab.

- Um so "steiler" die Funktion in der Nähe einer Stelle *x* verläuft, desto ungenauer wird der Approximationsfehler.
- Eine Lösung wäre die Wahl eines dynamischen Intervalls in Abhängigkeit vom *Gradienten* der Funktion an der Stelle *x*

Adaptive Algorithmen

Iterative numerische Algorithmen haben immer eine Schrittweite. Diese Schrittweite kann statisch oder adaptive veränderlich sein.

Wie schon bei der diskreten und approximierten Integration von Funktionen gezeigt kann eine adaptive Berechnung die auf Fehlertoleranzen und Fehlerschwellen mit der Anpassung der Schrittweite basiert das Endergebnis deutlich verbessern.

- Beispiele für numerische Software die adaptive Schrittweiten nutzt:
  - Elektroniksimulator SPICE3
  - Lösung von Diffentialgleicheungen (Physik, Atmosphäre, Wetter)
  - Maschinelles Lernen (ADAM Optimierer für KNN)

Adaptive Schrittweite

[Alexander Schwanecke]

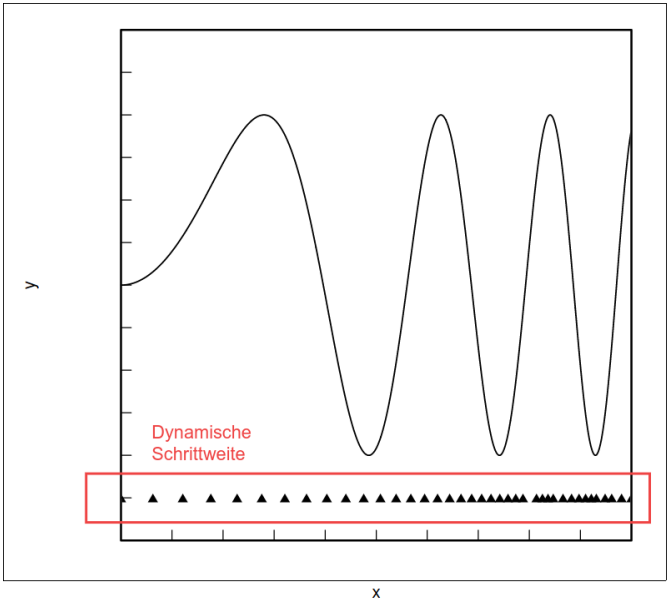


Abb. 10. Beispiel einer oszillierenden Funktion

Adaptive Schrittweite

Wie soll ohne Fehlerbetrachtung ein günstiges kleineres Intervall *dx* gewählt werden?

Adaptive Schrittweite

Wie soll ohne Fehlerbetrachtung ein günstiges kleineres Intervall *dx* gewählt werden?

Durch dynamische Verkleinerung des Intervalls wird jetzt die Rechenzeit (also das tatsächliche *N*) abhängig von der zu integrierenden Funktion (also den Daten)

- Die Schrittweite kann durch relativen Fehlervergleich schrittweise angepasst werden
- Die Schrittweite kann durch Analyse der zu integrierenden Funktion (Gradient) absolut angepasst werden (mit analytischen Fehlermodell)
- Die Schrittweite wird bei Überschreitung von Schwellwerten reduziert.

Gradientenbasierte Anpassung

```
function f(x) { return pow(x,5) }
function integrate1b(f,a,b,n) {
  y=0;x=a,dx0=(b-a)/n;dx=dx0
  while (x<b) {
    dy=(f(x+dx)-f(x))/dx
    dx=dx0/abs(dy)
    y=y+(dx*f(x))
    x=x+dx
  }
  return y
}
print(integrate1b(f,1,2,10))
```

Alg. 5. Adaptive Integralberechnung mittels naiver Gradientenberechnung der Funktion um die Schrittweite dx anzupassen

Aber eigentlich interessiert uns nicht der Gradient der Funktion (Fehler immer noch vom Startwert *n* abhängig), sonder der Fehler selbst. Den Fehler wollen wir unter eine Schwelle ε bringen (auch Toleranz genannt).

Aber eigentlich interessiert uns nicht der Gradient der Funktion (Fehler immer noch vom Startwert *n* abhängig), sonder der Fehler selbst. Den Fehler wollen wir unter eine Schwelle ε bringen (auch Toleranz genannt).

- Bisher: Die Besonderheiten der Funktion, sei es ihre Linearität oder auch ihr großer Veŕänderungsgrad werden nur global berücksichtigt und rufen, wegen kleinen Bereichen starker Veränderung evtl. eine sehr enge Maschenbreite bei vorgegebenem Fehler hervor.
- Wir müssen jetzt den aktuellen Diskretisierungsfehler abschätzen. Aber wie? Wir kennen das "richtige" Ergebnis nicht.
- Aber wir können die Simpsonregel mit verschiedener Anzahl von Stützpunkten vergleichen. Beide haben einen Fehler relativ zum "richtigen" Wert, aber unterschiedlich. Dieser Unterschied definiert die Entscheidung ob verfeinert wird oder nicht.

Adaptiver Simpson Algorithmus

[Alexander Schwanecke]

```
a = 0.; // Linke Grenze.
b = 1.; // Rechte Grenze.
e = 1e-7; // Erlaubter Gesamtfehler.

function SV(f,a,b,e,t,h) {
  g = 15*e/(b-a)
  if (t >= b) return 0// Erforderlicher Bereich integriert?
  // Die Simpsonregel.
  IS = h/3*(f(t)+ 4*f(t+h)+f(t+2*h))
  // Die zusammengesetzte Simpsonregel.
  IZS = h/6*(f(t)+4*f(t+h/2)+2*f(t+h)+4*f(t+3*h/2)+f(t+2*h))
  if (abs(IZS-IS) <= (g*h)) { // Die Fehlerabschaetzung.
    I = IZS; // Aufsummierung.
    I += SV(f,a,b,e,t+2*h,(b-(t+2*h))/2) // Restlicher Intervall.
    return I
  } else
    return SV(f,a,b,e,t,h/2) // Verfeinerung.
}
function integrate4(f,a,b,eps) {
  y=SV(f,a,b,eps,a(b-a)/2)
  return y
}
```

Alg. 6. In Abhängigkeit vom Fehler zweier unterschiedlicher Teilberechnungen wird entweder das Teilergebnis für die Summation verwendet oder verfeinert.

Vergleich der verschiedenen Algorithmen

Universelle Berechnungsfunktionen

- Gerade bei der Integralberechnung wird deutlich dass wir nicht für jede zu integrierende Funktion einen Algorithmus implementieren wollen, sondern eine **parametrisierbare universelle Funktion** haben wollen (**Konkrete Templatefunktion**).

In dynamisch typisierten Programmiersprachen und funktionalen Sprachen kein Problem. Man benötigt dazu nur einfache Lambda Ausdrücke.

Lambda Ausdruck

Die unverselle Funktion $F(f,p)$ soll eine parametrisierte Berechnung (Parametersatz p, wie z.B. Integralgrenzen, Stützpunkte usw.) einer beliebigen Funktion f durchführen.

Die Funktion f wird als namenlose Funktion in Form eines Lambda Ausdrucks $x \rightarrow \varepsilon$ beschrieben und an F übergeben.

Universelle Berechnungsfunktionen

```
function mean(f,p) {
  dx=(p.b-p.a)/p.n
  x=p.a; sum=0
  for(i=0;i<p.n;i++) {
    sum+=f(x)
    x+dx
  }
  return sum/p.n
}
print(mean(x => (1/x),{
  a:1,
  b:10,
  n:5
}))
```

Alg. 7. Berechnung des Mittelwerts einer beliebigen Funktion

Universelle Berechnungsfunktionen

Lambda Ausdrücke in Java

- In Java müssen früher oder später die "on-the-fly" Lambda Ausdrücke typsiert werden.
- Hier über das Functioninterface: `Function <T,R>`

```
import java.util.function.Function;
class MyMath {
  public float mean(Function <Float,Float> f,float a, float b,float delta) {
    float y=0; int n=0;
    for(float x=a;x<=b;x+=delta) { n++; y+=(f.apply(x)) };
    return y/n;
  }
}
class Test {
  public static void main(String[] args) {
    Function<Float,Float> foo = (x) -> { return x+1; };
    MyMath m = new MyMath();
    System.out.println(m.mean(foo,1,2,(float)0.1));
    System.out.println(m.mean((x) -> { return 1/x; },1,2,(float)0.1));
  }
}
```

Alg. 8. Lambda Ausdrücke in Java

Lambda Ausdrücke in Java

Algorithmus 3: Iterative Berechnung der Quadratwurzel

- Die Berechnung der Quadratwurzel steht meist als mathematische Funktion in der Standard Mathematikbibliothek (Math) zur Verfügung und kann im numerischen Koprozessor des Rechners erfolgen
- Aber wie wird sie berechnet? Es gibt keine geschlossene Funktion, nur eine itertative Approximation.

Gegeben: eine positive reelle Zahl a > 0. [matalg]
Gesucht: eine numerische Näherung für die Quadratwurzel von a.

Die Berechnung ist iterativ nach dem Heron Verfahren:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

Algorithmus 3: Iterative Berechnung der Quadratwurzel

Algorithmus 3: Iterative Berechnung der Quadratwurzel

Fragen:

1. Gibt es immer eine Konvergenz, d.h. ist das Stabilitätskriterium gewährleistet? Kann man als Theorembeweis bestätigen!
2. Wie sieht es mit der Genauigkeit aus? Könnten Rundungsfehler eine Rolle spielen?
3. Wie hoch ist der Rechenaufwand und wovon hängt er ab? Ist ein statisches N sinnvoll?
4. Wie könnte man den Algorithmus verbessern (Laufzeit und Genauigkeit)?

Algorithmus 4: Vektorprodukt

- Vektoren sind integraler Bestandteil der Numerik. Vektoren (Arrays) sind Datenstrukuren!
- Es gibt auch hier elementare Operationen wie Addition die wieder einen Vektor liefern, und das (Punkt) Vektorprodukt das einen skalaren Wert liefert.

```
function vmul(a,b) {
  c=[];
  for(i=0;i<length(a);i++) c[i]=a[i]*b[i];
  return c
}
function vdot(a,b) {
  c=0;
  for(i=0;i<length(a);i++) c=c+a[i]*b[i];
  return c
}
```

Alg. 9. vmul: Elementweise Multiplikation vdot:Skalarprodukt

Wie sieht es mit dem Rechenaufwand bezüglich *N=length*(*vec*) aus?

Algorithmus 4: Vektorprodukt

Welchen Rechenaufwand hat das Vektorprodukt (als Größe der Vektorelemente N)?

Algorithmus 5: Matrixprodukt

- Jetzt geht es in zwei Dimensionen. Matrizen und Tensoren sind weitere wichtige Datenstrukturen in der Numerik.
- Matrixoperationen kommen häufig vor, auch im ML

```
function mmul(a,b) {
  c=matrix(nrow(a),ncol(a));
  for(i=0;i<nrow(a);i++)
    for(j=0;j<ncol(a);j++)
      c[i][j]=a[i][j]*b[i][j];
  return c
}
```

```
function mdot(a,b) {
  c=matrix(nrow(a),ncol(b));
  for(i=0;i<nrow(a);i++)
    for(j=0;j<ncol(b);j++)
      for(k=0;k<ncol(a);k++)
        c[i][j]=c[i][j]+a[i][k]*b[k][j];
  return c
}
```

Alg. 10. mmul: Elementweise Multiplikation mdot:Matrixprodukt

Wie sieht es mit dem Rechenaufwand bezüglich $N=max(nrow(mat),ncol(mat))$ aus?

Algorithmus 4: Matrixprodukt

Welchen Rechenaufwand hat das Matrixprodukt im Vergleich zum Vektorprodukt (mit Größe von N als die Anzahl Zeilen oder Spalten)?

Algorithmus 6: Polynomberechnung

- Ein Polynom n-ten Grades ist berechenbar (liefert ein Skalar y) durch einen Parametervektor $\boldsymbol{p}$ und einem Eingabevektor $\boldsymbol{x}$:

$$y = \sum_{i=0}^{n-1} p_i \cdot x_i^i$$

- Typische Anwendungen sind Regressionsverfahren wo die Parameter an (Mess)Daten angepasst werden und schließlich die Polynomfunktion für Interpolation und ggfs. Extrapolation verwendet wird.

⇒ ML Anwendung (Messwertvorhersage)

Algorithmus 6: Polynomberechnung

Algorithmus 7: Lösung eines Linearen Gleichungssystems

Ein LGS ist gegeben als:

$$\hat{A}\vec{x} = \vec{b}$$

$$\hat{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & .. & .. & a_{1,n} \\ a_{2,1} & a_{2,2} & .. & .. & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & .. & .. & a_{n,n} \end{pmatrix}$$

$$\vec{b} = (b_1, b_2, .., b_n)$$

$$\vec{x} = (x_1, x_2, .., x_n)$$

Der Vector *x* wird gesucht, mit einem gegebene Satz an Parametern $a_{\mathrm{i,j}}$ und b_{i} , die man aus einem gegebenen zu lösenden Problem erhält (also gemessene Werte).

Bei der Lösung des Gleichungssystems Ax=b spielen Dreiecksformen der A-Matrize eine große Rolle. Man spricht von der linken (L) und rechten (R, oder U für Upper) Dreiecksmatrix.

- Hat man L und R bestimmt, kann man direkt das LNG lösen (also x berechnen).

$$\hat{L} = \begin{pmatrix} l_{1,1} & 0 & .. & .. & 0 \\ l_{2,1} & l_{2,2} & .. & .. & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & 0 \\ l_{n,1} & l_{n,2} & .. & .. & a_{n,n} \end{pmatrix}, \hat{R} = \begin{pmatrix} r_{1,1} & r_{1,2} & .. & .. & r_{1,n} \\ 0 & r_{2,2} & .. & .. & r_{2,n} \\ 0 & \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & .. & .. & r_{n,n} \end{pmatrix}$$

Hat die Matrix A linke oder rechte Dreiecksgestalt, dann ist das Lösen des Gleichungssystems besonders einfach. Beim Lösen von

$$Lx = b$$

spricht man von Vorwärtssubstitution und beim Lösen von

$$Rx = b$$

spricht man von Rückwärtssubstitution.

- Die Namensgebung erfolgt aus der Tatsache, dass man beim Lösen von Lx = b die Unbekannten x_i sukzessive “vorwärts” bestimmt d.h. zuerst $x_1 = b_1 / a_{1,1}$, mit dessen Hilfe man x_2 bestimmt, dann x_3 usw.
- Bei Lösen von Rx = b werden die Unbekannten x_i sukzessive “rückwärts” bestimmt, d.h. zuerst $x_n = b_n / a_{n,n}$, dann damit x_{n-1}, dann x_{n-2} usw.
- Dieses Vorwärts- und Rückwärtseinsetzen formalisieren wir in den folgenden zwei Algorithmen

Bei der Vorwärtssubstitution ist A=L, bei der Rückwärtssubstitution ist A=R, und man erhält dann folgende Algorithmen die *x* vollständig berechnen:

$$\text{for}(i=l;i\leq n;i++)\text{do}$$
$$x_i = \frac{1}{a_{i,i}}\left(b_i - \sum_{k=1}^{i-1} a_{i,k}x_k\right)$$
$$\text{for}(i=n;i\geq 1;i--)\text{do}$$
$$x_i = \frac{1}{a_{i,i}}\left(x_i - \sum_{k=i+1}^{n} a_{i,k}x_k\right)$$

Gauss’scher Algorithmus und LR-Zerlegung

- Wir haben gesehen, dass gestaffelte Gleichungssysteme (d.h. solche, bei denen die Matrix A linke oder rechte Dreiecksgestalt hat) besonders einfach aufzulösen sind.
- Der Gauss'sche Algorithmus führt nun den allgemeinen Fall auf diese beiden Fälle zurück, indem er eine Matrix A in ein Produkt aus einer Linksdreiecksmatrix und einer Rechtsdreiecks-matrix zerlegt:

$$\hat{A} = \hat{L}\hat{R}$$

- Im Grunde ist die LR Zerlegung das Verfahren was wir aus der Schule kennen um händisch ein LGS zu lösen ↠ **Treppeniteration**.
- Der Ansatz: Die LR-Zerlegung einer Matrix A geschieht in n − 1 Schritten.
  - Es wird nun von der zweiten, dritten, etc. Zeile ein geeignetes Vielfaches der ersten Zeile subtrahiert, um die Variable x_1 in Zeilen 2 bis n zu eliminieren.

$$l_{i,1} = \frac{a_{i,l}}{a_{1,1}}$$

$$a_{i,j}^{(1)} = a_{i,j} - a_{1,j} l_{i,1}$$

- Schließlich erhält man:

$$\boldsymbol{L} := \begin{pmatrix} 1 & & & & & \\ l_{21} & \ddots & & & & \\ l_{31} & \ddots & 1 & & & \\ \vdots & & l_{k+1,k} & 1 & & \\ \vdots & & \vdots & \ddots & \ddots & \\ l_{n1} & l_{n2}\cdots & l_{n\,k} & \cdots & l_{n,n-1} & 1 \end{pmatrix}, \quad \boldsymbol{R} := \begin{pmatrix} a_{11} & a_{12} & \cdots & & & \\ 0 & a_{22}^{(1)} & a_{21}^{(1)} & \cdots & & \\ \vdots & 0 & a_{33}^{(2)} & a_{34}^{(2)} & & \\ \vdots & & \ddots & \ddots & & \\ \vdots & & & \ddots & \ddots & \\ 0 & \cdots & \cdots & \cdots & 0 & a_{nn}^{(n-1)} \end{pmatrix}$$

- Dabei sind die $a^{(1)}$, $a^{(2)}$, usw. die aus der i-ten Iteration erhaltenen modifizierten A Koeffizienten.

```
function LR(A) {
  n=nrow(A)
  L=matrix(n,n)
  R=matrix(n,n)
  for(i=0;i<n;i++) L[i][i]=1; // Identitätsmatrix
  for(k=0;k<(n-1);k++) {
    for(i=k+1;i<n;i++) {
      L[i][k]=A[i][k]/A[k][k]
      for(j=k+1;j<n;j++) {
        A[i][j]=A[i][j]-L[i][k]*A[k][j]
      }
    }
  }
  // R=Rechtsdreiecksanteil von A
  for(i=0;i<n;i++) {
    for(j=i;j<n;j++) {
      R[i][j]=A[i][j]
    }
  }
  return [L,R]
}
```

Alg. 11. Inplace (L) Gauss'sche LR Zerlegung ohne Pivotsuche

Gesamte Vorgehensweise:

1. Bestimme LR-Zerlegung von A
2. Löse Ly = b mithilfe der Vorwärtssubstitution. Dabei beachtet man, dass für die Diagonalelemente von L gilt: $L_{i,i} = 1$.
3. Löse Rx = y mithilfe der Rückwärtssubstitution Algorithmus

Achtung: LR-Zerlegung nur möglich wenn Diagonalelemente (Pivotelemente) von $A(a_{k,k})$ nicht Null sind. Ein Pivot wäre in einen der Zwischenschritte dass $A_{k,k} = 0$ ist.

- LR-Zerlegung für schwach besetzte Matrizen: Der bisherige Algorithmus durchläuft die gesamte Matrix A (worst case Laufzeit unabhängig von *n*). Da gibt es Verbesserungsbedarf!

Beispiel

Laufzeit

Teilberechnung	Laufzeit
LR-Zerlegung	$1/3\ n(n\text{-}1)(n+1)$
Vorwärtssubstitution	$1/2\ n(n\text{-}1)$
Rückwärtssubstitution	$1/2\ n(n\text{-}1)$
Gesamt	$1/3\ n^3+n^2\text{-}1/3n = O(n^3)$

Tab. 3. Kosten für das Lösen eines linearen Gleichungssystems nach Guass

Laufzeit

In der Praxis (z.B. in der Strukturmechanik und bei der Diskretiersierung von partiellen Differen-tialgleichungen) sind die auftretenden Matrizen oft schwach besetzt (engl. sparse), d.h. viele Einträge von A sind gleich Null. Dies kann in zweierlei Hinsicht ausgenutzt werden:

1. Speicherersparnis: Man speichert nicht die gesamte Matrix A ab, sondern nur die wesentliche Information, d.h. welche Einträge von Null verschieden sind und was ihre Werte sind.

2. Die Matrizen L, R der LR-Zerlegung von A sind ebenfalls schwach besetzt. Auch hier kann Speicher und Rechenzeit eingespart werden, indem nur die nicht-trivialen Einträge von L und R berechnet werden.

Spezielle Matrizen sind:

A. Bandmatrizen

B. Skylinematrizen

Big O bleibt auch bei der Rechnung mit reduzierten Matrizen $O(n^3)$, die Komplxitätsklasse bleibt unverändert. Aber dennoch sinkt die Rechenzeit signifikant, was schon hilfreich sein kann (i.A. $n \bullet p \bullet q$, wobei $p,q < n$ sind.

Pivotisierung

Was machen wir wenn Diagonalelemente Null sind oder bei der Iteration Null werden?

- Ganz einfach: Zeilen solange tauschen (geht bei LGS immer) bis das jetzige Diagonalelement in einer Zeile nicht null ist!
- Dazu kann z.B. man Permutationsmatrizen benutzen

Die Zeilenvertauschungen müssen notiert werden, da später bei der Resubstituierung die richigen b Werte und x Zuordnungen ausgweählt werden müssen.

Algorithmus 8: Differenzieren und Gradientenberechnung

Neben der Integration von Funktionen die wir schon kennen gelernt haben ist das Differenzieren von Funktion eine weitgere häufig vorkommende und wichtige Methode in der Mathematik und Numerik.

- Bei der Integration wurden Kuvrenabschnitte durch Rechtecke approximiert.
- Bei der Differenzierung führt ähnlich der Trapezregel eine Linearisierung des i.A. nicht-linearen Kurvenverlaufs zwischen zwei Intervallpunkten $[x_1,x_2]$ durch:

[numbook]

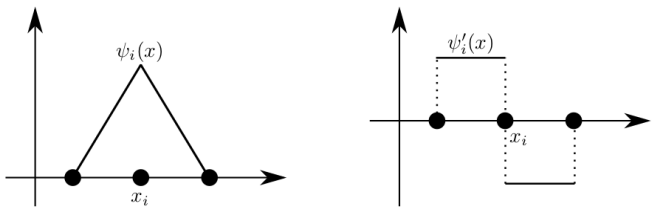


Abb. 11. Kann man eine Funktion f(x) in abschnittsweise lineare Abschnitte (lineare Teilfunktionen) zerlegn ("Hüte"), dann ist die Ableitung dieser linearisierten Funktion f'(x) dann eine Folge von konstanten Funktionen. (Pice-wise linear Function)

Ansatz ist die Approximation durch endliche Differenzen...

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

Entfällt der Grenzwert dann kann auch schreiben:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

O(h) ist ein Fehlerterm, und schließlich gilt für die Vorwärtsdifferenzapproximation:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Weiterhin kann man auch die Rückwärtsdifferenzapproximation verwenden:

$$\begin{aligned} f'(x) &= \frac{f(x+h) - f(x-h)}{2h} + O\left(h^2\right) \\ &\approx \frac{f(x+h) - f(x-h)}{2h} \end{aligned}$$

Höhere Ableitungen lassen sich entsprechend einfach berechnen:

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

Divide & Conquer

[numbook]

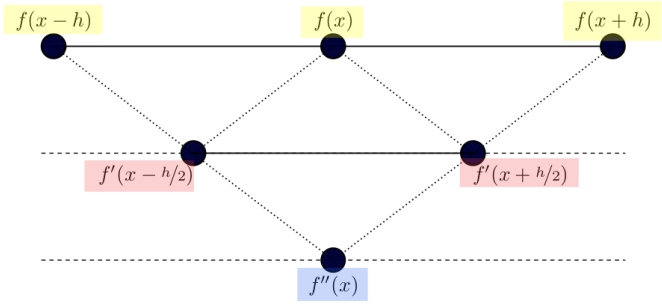


Abb. 12. Die Berechnung der zweiten Ableitung f''(x) durch geteilte Differenzen kann als die mehrfache Anwendung der geteilten Differenzregel angesehen werden, einmal angewendet um f' zu approximieren und ein zweites Mal, um f'' zu approximieren.

Auch hier haben wir wieder das Problem steigender Ungenauigkeit bei kleinen Differenzen, wie das folgende Beispiel zeigt.

[numbook]

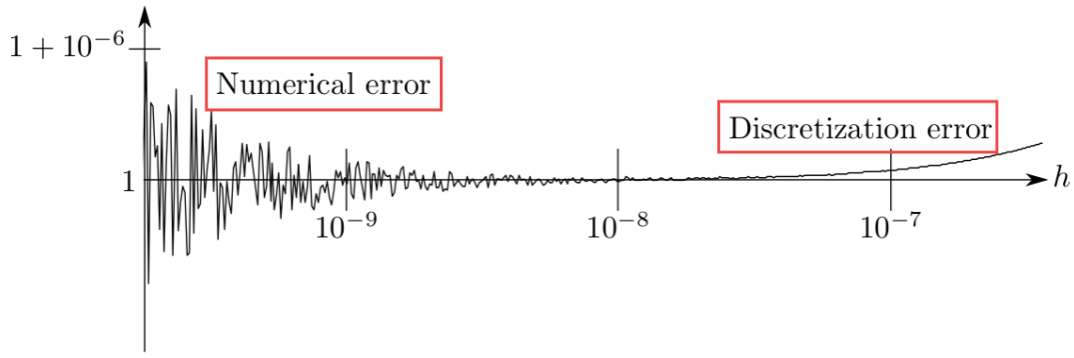


Abb. 13. Die Differenz (f(x+h)−f(x)/h als eine Funktion von h für die Funktion $f(x)=x^2/2$ mit IEEE Gleitkommaarithmetik. Wichtig: Der numerische (Rundungs)fehler ist bei kleinen h dominierend, bei großen h der Diskretisierungsfehler der Differenz

Algorithmus

```
function diff(f,x,h) {
  return (f(x+h)-f(x-h))/(2*h)
}
y=diff(sin,1,0.01)
```

Alg. 12. Numerische Differenzierung einer Funktion. Es wird für einen bestimmten x-Wert berechnet, anders als beim Integral wo eine Akkmulation von x-Werten in einem Intervall stattfindet.

Komplexität

- O(1) oder O(n) bei n Differenzwerten.

Algorithmus 9: KNN / Perzeptron

Die Berechnung von künstlichen neuronalen Netzwerken (KNN) basiert starak auf Matrixakgebra und Grdaientberechnung (also Differentiation)

Ein KNN ist mathematisch eine i.A. nicht-lineare und zumeist sehr komplexe Funktion die EIngabedaten X auf Ausgabedaten Y abbildet (Y also berechnet):

$$ML : X \times P \rightarrow Y$$
$$f_P(X) : X \rightarrow Y$$

- Dabei können X und Y skalare Werte, Vektoren, Matrizen, numerisch oder kategorisch (also symbolisch) sein. P sind Modellparameter.

Man unterscheidet bei KNN zwei Phasen der Berechnung:

1. Vorwärtsberechnung f(X) : $X \rightarrow Y$
2. Rückwärtsberechnung f(Y): $Y \rightarrow \partial E(Y)/\partial P$, mit E: Fehlerfunktion, P: Modellparameter (Rückpropagation == Training durch Anpassung von P)

Das Perzeptron

- Ein Perzeptron ist einem biologischen Neuron mathematisch sehr grob nachempfunden.
- Das Perzeptron hat als Eingabe einen Vektor $\boldsymbol{x}$, als Ausgabe ein skalaren Wert y
- Die Berechnungsfunktion lautet:

$$a(\vec{x}) = f(u(\vec{x}) + b)$$

$$u(\vec{x}, \vec{w}) = \sum_{i=1}^{n} x_i w_i$$

$$f(u) = \frac{1}{1 + e^{-u}}$$

$$\vec{P} = \vec{W} = (w_1, .., w_n)$$

- Dabei ist $a(\boldsymbol{x})$ die Ausgabefunktion des Perzeptrons und setzt sich aus zwei verketteten Funktionen zusammen:
  - $u(\boldsymbol{x})$: Gewichte Summation der Elemente von $\boldsymbol{x}$
  - $f(u)$: Die sogenannte Aktivierungsfunktion, hier z.B. die sigmoid Funktion

Das Perzeptron

- Der Eingabevektor $\boldsymbol{x}$ kann verschiedene sensorische Variablen zusammenfassen:
  - Temperatur
  - Luftfeuchtigkeit
  - Länge eines Blattes
  - Zeitaufgelöste Signale (Schall)
  - Bilder
  - usw.
- Der Ausgabewerte eines Perzeptrons y kann genutzt werden für:
  - Klassifikation (hier eine Klasse oder zwei mutual exklusive Klassen 0/1)
  - Regression

Das Perzeptron

[NNbook]

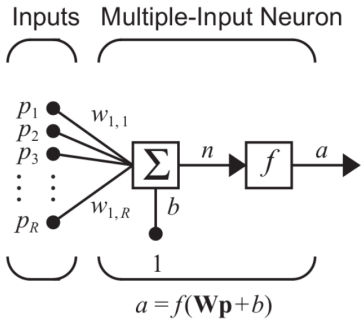


Abb. 14. Perzeptron (künstliches Neuron) mit mehreren Eingängen

Vektoralgebra: $\boldsymbol{W}$ ist ein Vektor, b ist ein Skalar

Neuronale Netzwerke (eine Schicht)

[NNbook]

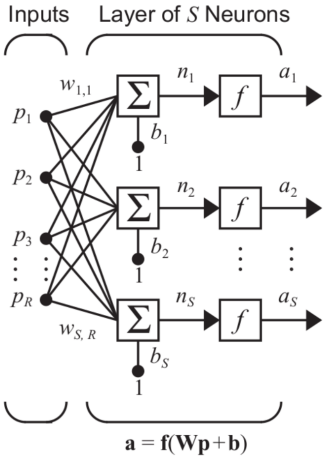


Abb. 15. Eine Schicht aus mehreren Neuronen

Matrixalgebra

Neuronale Netzwerke

- Bei einer Schicht sind jetzt $\boldsymbol{W}$ eine Matrize und $\boldsymbol{b}$ ein Vektor
- Jetzt wird ein Matrix-Vektor Produkt benötigt, ähnlich dem Matrixprodukt, aber als Ergebnis gibt es einen Vektor.

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

Neuronale Netzwerke (mehrere Schichten)

[NNbook]

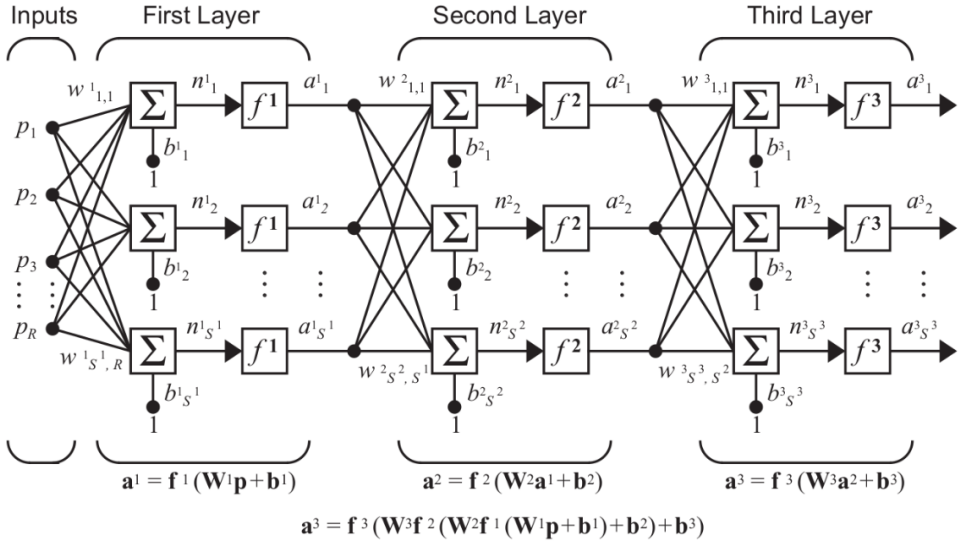


Abb. 16. Komplexes Neurnales Netzwerk (hier mit drei Schichten und S_i Neuronen pro Schicht)

Neuronale Netzwerke (mehrere Schichten)

[NNbook]

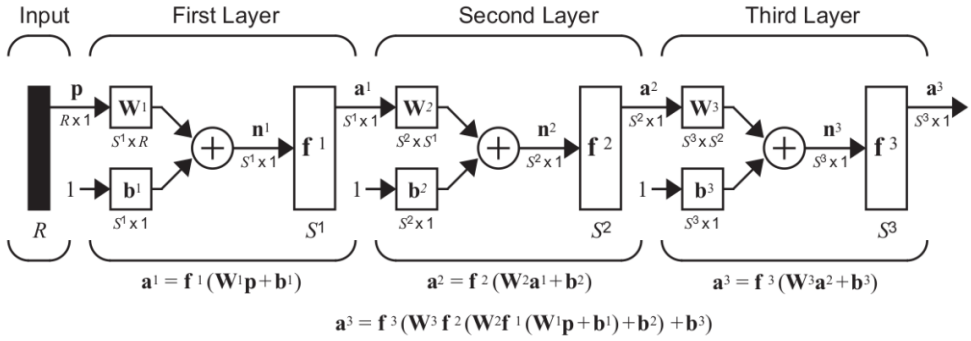


Abb. 17. Kompakte Darstellung des komplexen Neurnalen Netzwerks (hier mit drei Schichten und S_i Neuronen pro Schicht)

Aktiverungsfunktionen

- Sie spielen eine wichtige Rolle, es gibt eine Vielzahl verschiedener Funktionen
- Bekannt sind teillineare (ReLU) für Regression und sigmoid sowie Schrittfunktionen für Klassifikation.

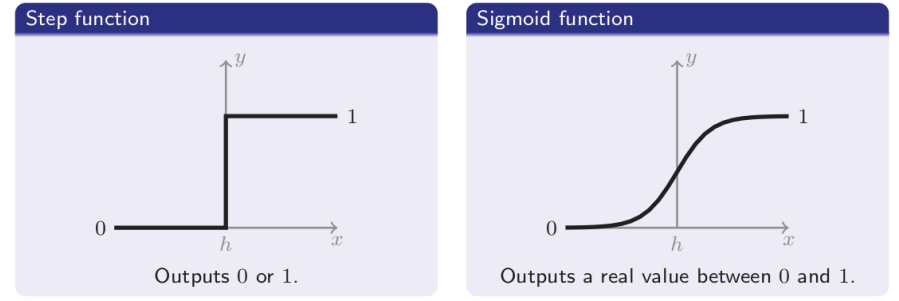


Abb. 18. Vergleich zweier bekannter Aktivierungsfunktionen deren Ausgabe limitiert ist (Abschneeidung, Clipping)

Training eines Perzeptrons

Die Modellparameter $\boldsymbol{P}$ sind unbekannt, vielleicht mit zufälligen Werten initialisiert. Wie sollen die richtigen Modellparameter $\boldsymbol{P}$ (hier $\boldsymbol{W}$ und b) bestimmt werden damit das Modell x auf y korrekt abbildet?

Fehlerminimierung mit absteigenden Gradienten

- Es gibt Trainingsdaten, die Beispiele für $\boldsymbol{x}$ und y liefern. Z.B.

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

- Mit einem gegeben Parametersatz können wird jetzt für alle Beispiele einen Fehler berechnen, wobei *y* das Ziel und *a* die momentane Ausgabe des Modells ist:

$$E(\vec{x}, y, a) = y - a(\vec{x})$$

- Jetzt müssen die Parameter angepasst werden, bei einem Perzeptron sind es die Gewichte $\boldsymbol{W}$ und der Biaswert *b*.

- Hierfür kann man vereinfacht (gilt eigentlich nur bei linearer Aktivierungsfunktion) den Fehler verwenden, d.h. für jedes Beispiel werden die Parameter nacheinander angepasst:

$$w_i \leftarrow w_i + \alpha E x_i$$
$$b \leftarrow b + \alpha E$$
$$E = y - a$$

- α ist dabei die "Lernrate" $\in$ (0,1] die bestimmt wie große der Fehleranteil bei der Anpassung der Parameter ist.
  - Zu kleines α: Langsame Anpassung des Modells
  - Zu großes α: Sprünge und Divergenz (keine Konvergenz in Richtung Fehler 0)

Bisher wurde eine lineare Aktivierungsfunktion angenommen. Stückweise ist die sigmoid Funktion auch linear, daher "geht so".

- Verbesserte Anpassung der Parameter mit dem Ziel der Fehlerminimierung (absteigender Gradient) mit Differenzierung der Perzeptron Funktion (bzw. deren Fehler) nach den einzelnen Parametern:

$$w_i \leftarrow w_i + \Delta w_i$$
$$\Delta w_i = -\alpha \frac{\partial E}{\partial w_i}$$
$$E(\vec{x}_j) = y_j - a(\vec{x}_j)$$

- In der Neuronfunktion *a* stecken ja zwei Funktionen drin. Die Ableitung der Summenfunktion *u* ist konstant, bleibt nur noch die Aktivierungsfunktion *f*, so dass dann gilt:

$$\Delta w_i = -\alpha E f'(u) x_i$$

Ableitung von einer Funktion? Geschlossen analytisch oder approximativ numerisch?

- Es gibt verschiedene gängige Aktivierungsfunktionen wie z.B. die sigmoid Funktion.
- U.A. die sigmoid Funktion hat eine Eigenschaft die es erlaubt die Ableitung f' wieder mit f auszudrücken ("Selbstähnlichkeit")!

$$f(u) = \frac{1}{1+e^{-u}}$$
$$f'(u) = f(u)(1-f(u))$$

- Numerisch: Auch angenehm da die sigmoid Funktion einen begrenzter Gradienten besitzt, nämlich [0,0.25] da der Wertebereich von der sigmoid Funktion auf (0,1) begrenzt ist!
  - Apprximationsfehler sind auch bei größeren h Intervall (aber $h < 1$) klein!
  - Numerische Fehler sind nicht zu erwarten (also Divergenz)

+ ∅ Ⓟ ▶ perzeptron1.js perzeptron2.js perzeptron4.js

```
    // bias
    P[length(P)-1]=P[length(P)-1]+err*alpha
  }
  for(e=0;e<epochs;e++) {
    r=randint(0,length(data)-1)
    x=slice(data[r],0,1)
    y=data[r][2]
    update(x,y)
  }
}
train(m,P,data,1000,0.05)
```

```
[ 0.37785642044597 71,
  0.9022372500821144,
  0.9021504141970335,
  0.9929127141635602 ]
[ 0.36269012262817507,
  0.9075035899451477,
  0.9081995346244482,
  0.9941711034437114 ]
[ 0.3361271861474265,
  0.9089569518289452,
  0.9089449892483454,
  0.9949453891692261 ]
```

+

Zusammenfassung

1. Numerik ist Algorithmik.
2. Datenstrukturen sind hier:
   - Funktionen (Werte erster Ordnung, Lambda Ausdrücke!)
   - Vektoren
   - Matrizen
3. Operationen (Funktionen) auf den Daten sind hier:
   - Iterative approximative Berechnungen von Zahlen und Funktionen mit Summen
   - Integration von Funktionen
   - Differenzierung von Funktionen (Gradienten)
   - Regression
   - Vektor- und Matrixalgebra
   - Lösung linearer Gleichungssysteme (V/M Algebra)
   - Maschinelles Lernen (V/M Algebra, Gradient)
4. Korrektheit == Konvergenz + Genauigkeit abhängig von Verfahren und Daten!
5. Effizienz == Abhängig von Verfahren und Daten, i.A. polynomielle Laufzeitklasse $O(n|n^2|n^3)$!