ModProg 15-16, Vorl. 11

Richard Grzibovski

Jan. 13, 2016

Übersicht

Speicherkomplexität

Der effziente Umgang mit dem vorhandenen Speicher ist ein sehr wichtiger Aspekt bei der Entwicklung von Algorithmen, dem beim Design von Datenstrukturen ein besondere Rolle zukommt. Nicht selten ist dies entscheidend dafür, ob die Aufgabe uberhaupt rechnergestützt gelöst werden kann. Zunächst einige einfache Beispiele:

- Um die Komponenten eines Vektors $x \in \mathbb{R}^n$ abzuspeichern, benötigt man offensichtlich n Speicherplätze.
- Ebenso evident ist, dass zur Speicherung einer Matrix $A \in \mathbb{R}^{m \times n}$ mn Speicherplätze erforderlich sind. Speziell folgt für quadratische Matrizen $A \in \mathbb{R}^{n \times n}$ ein Speicheraufwand von

$$Mem(n) = n^2.$$

Symmetrische Matrizen

In Anwendungen treten häufig symmetrische Matrizen auf. Das sind quadratische Matrizen $A = (a_{ij})_{i,j=1}^{n} \in \mathbb{R}^{n \times n}$ für die gilt:

$$A = A^{\top}, \text{d.h.} \forall i, j = 1, \ldots, n : a_{ij} = a_{ji}.$$

Ein Beispiel für eine solche Matrix ist etwa

$$A = \begin{pmatrix} 1 & -2 & 3 \\ -2 & 4 & 5 \\ 3 & 5 & -6 \end{pmatrix} \in \mathbb{R}^{3 \times 3}.$$

Symmetrische Matrizen

Bezeichnet

$$D = (d_{ij})_{i,j=1}^{n} = \operatorname{diag}(a_{11}, a_{22}, \ldots, a_{nn}) = \begin{cases} a_{ij}, & i = j \\ 0, & sonst \end{cases},$$

die Diagonale der Matrix A und ist

$$L = (l_{ij})_{i,j=1}^{n} = \begin{cases} a_{ij}, & i > j \\ 0, & sonst \end{cases},$$

so folgt für eine symmetrische Matrix A, dass gilt

$$L^{\top} = (l_{ji})_{i,j=1}^{n} = \begin{cases} a_{ij}, & i < j \\ 0, & sonst \end{cases},$$

so dass man die Matrix A wie folgt additiv zerlegen kann:

$$A = D + L + L^{\top}.$$

Die Matrix A ist also vollständig bekannt, wenn man D und L kennt. Statt eines Speicheraufwands von n^2 genügt im Fall von symmetrischen $(n \times n)$-Matrizen

$$Mem(n) = \sum_{k=1}^{n} k = n(n+1)/2.$$

Um eine symmetrische Matrix effizient im Speicher abzulegen, bietet sich u.a. folgende Vorgehensweise an, die ein eindimensionales Feld in Verbindung mit einer Indextransformation verwendet:

- Die ersten n Einträge belegen wir mit den Einträgen der Diagonalmatrix D, d.h. wir speichern zuerst a_{11} bis a_{nn} .
- Als nächstes legen wir die Matrix L Zeile für Zeile hintereinander im Feld ab, wobei wir natürlich nur die von 0 verschiedenen Einträge l_{ij} mit $i > j$ berücksichtigen. Der Index idx(i, j) des Eintrags l_{ij} lautet dann

$$idx(i,j) = n - 1 + \sum_{k=0}^{i-2} k + j = n - 1 + (i-1)(i-2)/2 + j,$$

wobei die Feldindizierung wie in C ublich mit 0 beginnt.

Symmetrische Matrizen

$d_{11} = a_{11}$ $\text{idx} = 0$			
$l_{21} = a_{21}$ $\text{idx} = 4$	$d_{22} = a_{22}$ $\text{idx} = 1$		
$l_{31} = a_{31}$ $\text{idx} = 5$	$l_{32} = a_{32}$ $\text{idx} = 6$	$d_{33} = a_{33}$ $\text{idx} = 2$	
$l_{41} = a_{41}$ $\text{idx} = 7$	$l_{42} = a_{42}$ $\text{idx} = 8$	$l_{43} = a_{43}$ $\text{idx} = 9$	$d_{44} = a_{44}$ $\text{idx} = 3$

Symmetrische Matrizen

Diese Speichermethode lässt sich in Form einer Struktur realisieren, die die Matrixdimension enthält:

```
1  typedef struct {
2      int dim;
3      double *val;
4  } SymMatrix;
```

Bei der dynamischen Erzeugung wird die Komponente `dim` mit der jeweiligen Matrixdimension belegt und für das Feld `val` wird Speicher durch

```
val=malloc(n*(n-1)*sizeof(double)/2);
```

angefordert. Um das Produkt einer derart abgespeicherten Matrix mit einem Vektor $x \in \mathbb{R}^n$ zu implementieren, verwendet man die Zerlegung

$$Ax = Dx + Lx + L^\top x.$$

Symmetrische Matrizen

$$Ax = Dx + Lx + L^\top x.$$

$$(Dx)_i = d_{ii} x_i$$

$$(Lx)_i = \sum_{j=1}^{i-1} l_{ij} x_j$$

$$(L^\top x)_i = \sum_{j=i+1}^{n} l_{ji} x_j$$

wobei
d_{ii} =`val[i-1]` und
l_{ij} =`val[dim-1+(i-1)(i-2)/2+j]`.

Das dyadische Produkt

Die allgemeine Definition des dyadischen Produkts zweier Vektoren

$$u = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{pmatrix} \in \mathbb{R}^m, \quad v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \in \mathbb{R}^n$$

lautet

$$uv^\top = \begin{pmatrix} u_1 v_1 & u_1 v_2 & \dots & u_1 v_n \\ u_2 v_1 & u_2 v_2 & \dots & u_2 v_n \\ \dots & \dots & \dots & \dots \\ u_m v_1 & u_m v_2 & \dots & u_m v_n \end{pmatrix} \in \mathbb{R}^{m \times n}.$$

Das dyadische Produkt

Diese Matrix lässt sich offensichtlich besonders effizient abspeichern, denn statt der im allgemeinen Fall erforderlichen mn Speicherplätze benötigt man für das dyadische Produkt lediglich Speicher für die Vektorkomponenten, d.h. es ist in diesem Fall

$$Mem(uv^\top) = m + n.$$

Speziell gilt im (quadratischen) Fall $m = n$:

Ist eine quadratische Matrix $A \in \mathbb{R}^{n \times n}$ als dyadisches Produkt darstellbar, so liegt statt eines (in n) quadratischen Speicheraufwands

$$Mem(A) = n^2$$

lediglich eine lineare Speicherkomplexität von

$$Mem(uv^\top) = 2n$$

vor.

Das dyadische Produkt

Diese Tatsache motiviert die Untersuchung der so genannten *Niedrigrangapproximation*, bei der eine Matrix $A \in \mathbb{R}^{m \times n}$ durch eine Linearkombination von dyadischen Produkten approximiert wird:

$$A \approx \sum_{i=1}^{r} \alpha_i u^{(i)} v^{(i)^\top}, \; u^{(i)} \in \mathbb{R}^m, \; v^{(i)} \in \mathbb{R}^n, \; \alpha_i \in \mathbb{R}. i = 1$$

Das dyadische Produkt ist nicht nur sehr vorteilhaft in Bezug auf den Speicherbedarf, sondern ermöglicht auch schnelle Berechnungen.

Laufzeitkomplexität

Eine weitere Herausforderung bei der Entwicklung von Algorithmen besteht darin, möglichst wenig Operationen zur Lösung der Aufgabe zu benötigen. Es ist in vielen Fällen alles andere als trivial, zu einem gegebenen Problem die zu seiner Lösung mindestens erforderliche Anzahl an Operationen zu bestimmen.

Beispiel: zwei Methoden zur Polynomauswertung

Um ein Polynom vom Grad $n \in \mathbb{N}$

$$P(x) = \sum_{k=0}^{n} a_k x^k,$$

an einer Stelle $x_0 \in \mathbb{R}$ auszuwerten, kann man zunächst folgende "naive" Methode verwenden:

1. Setze $p_0 := a_0$.
2. Für $k = 1, 2, \ldots, n$:
   - setze Hilfsgröße $M := a_k$,
   - Für $m = 1, \ldots, k$ berechne $M := M \cdot x_0$.
   - $p_0 := p_0 + M$.
3. Liefere p_0 zurück.

Zwei Methoden zur Polynomauswertung

In jedem der n Schritte werden jeweils eine Addition und k Multiplikationen durchgeführt. Da wir den Zeitaufwand für die Zuweisungen gegenüber den arithmetischen Operationen vernachlüssigen künnen, ist die Gesamtanzahl der Operationen

$$Op(n) = \sum_{k=1}^{n}(k+1) = n + \sum_{k=1}^{n} k = n + \frac{1}{2}n(n+1) = \frac{1}{2}n^2 + \frac{3}{2}n.$$

Man erkennt sofort, wo die Schwäche dieser Methode liegt: Für jeden Summanden im Polynom wird x^k komplett neu berechnet!

Zwei Methoden zur Polynomauswertung

Eine wesentlich effzientere Vorgehensweise erhält man, indem man das Polynom in Form ineinander geschachtelter Linearfaktoren schreibt:

$$P(x) = (\ldots(((a_n x + a_{n-1})\,x + a_{n-2})\,x + a_{n-3})\ldots)\,x + a_0$$

Man erhält so die folgende Methode, die auch Horner-Schema genannt wird:

1. Setze Hilfsgröße $p_0 := a_n$.
2. Für $k = n-1, \ldots, 0$ berechne

$$p_0 := p_0 \cdot x_0 + a_k.$$

3. Liefere p_0 zurück.

Diese Vorgehensweise ist nicht nur kürzer in der Formulierung: In jedem der n Schritte werden lediglich eine Addition und nur eine Multiplikation vorgenommen, so dass für diese Methode $Op(n) = 2n$ folgt.

Beispiele aus der linearen Algebra

Die Berechnung des euklidischen Skalarprodukts zweier Vektoren $x, y \in \mathbb{R}^n$

$$(x, y) = \sum_{i=1}^{n} x_i y_i$$

durch

1. Setze $s := x_1 y_1$.
2. Für $i = 2, \ldots, n$ berechne $s := s + x_i y_i$

erfordert offensichtlich n Multiplikationen und $n - 1$ Additionen, d.h.

$$Op(n) = 2n - 1.$$

Daraus folgt für das Matrix-Vektor-Produkt Ax mit $A \in \mathbb{R}^{m \times n}$ sofort

$$Op(m, n) = m(2n - 1),$$

bzw. im Speziallfall einer quadratischen Matrix $A \in \mathbb{R}^{n \times n}$

$$Op(n, n) = 2n^2 - n.$$

Beispiele aus der linearen Algebra

Das Matrizenprodukt aus $A \in \mathbb{R}^{k \times m}$ und $B \in \mathbb{R}^{m \times n}$ erfordert entsprechend einen Aufwand von

$$Op(k, m, n) = kn(2m - 1),$$

woraus für $A, B \in \mathbb{R}^{n \times n}$ folgt

$$Op(n, n, n) = 2n^3 - n^2.$$

Man spricht von einem (in n) kubischen Aufwand.

Liegt eine Matrix in Form eines dyadischen Produkts vor,

$$A = uv^\top \in \mathbb{R}^{m\times n}, u \in \mathbb{R}^m, v \in \mathbb{R}^n,$$

so lässt sie sich nicht nur effizient abspeichern. Auch die beiden letztgenannten Operationen lassen sich mit wenig Aufwand durchführen. Um das einzusehen, beachten wir, dass das Matrizenprodukt assoziativ ist. Für das Produkt mit einem Vektor $x \in \mathbb{R}^n$ folgt

$$Ax = uv^\top x = uv^\top x = (v, x)u,$$

und das Matrix-Vektor-Produkt reduziert sich auf die Berechnung eines euklidischen Skalarprodukts dessen Ergebnis anschließend mit allen Komponenten von u multipliziert wird. Der Gesamtaufwand beträgt also

$$Op_{dyad}(m, n) = 2n - 1 + m,$$

und für $m = n$

$$Op_{dyad}(n, n) = 3n - 1.$$

Statt eines in n quadratischen Aufwands ist die Komplexität nur noch linear.

Beispiele aus der linearen Algebra

Betrachten wir weiter das Produkt aus $A = uv^\top$ mit $u \in \mathbb{R}^l$, $v \in \mathbb{R}^m$ und $B = yz^\top$ mit $y \in \mathbb{R}^m$, $z \in \mathbb{R}^n$:

$$C = AB = (uv^\top)(yz^\top) = u(v^\top y)z^\top = (v, y)uz^\top.$$

Wir sehen, dass das Ergebnis C bis auf einen Vorfaktor wieder als dyadisches Produkt vorliegt. Da u und z bekannt sind, muss nur dieser Vorfaktor - dessen Berechnung $2m - 1$ Operationen erfordert - abgespeichert werden.

Die O-Notation (Landau-Symbole)

In den bisherigen Beispielen haben wir jeweils die exakte Anzahl der benötigten arithmetischen Operationen berechnet. Aus den folgenden Gründen kann man hierauf durchaus verzichten:

- Für kompliziertere und umfangreicher formulierte Algorithmen wird die exakte Berechnung der Laufzeitkomplexität sehr mühselig.
- Der Unterschied in den Laufzeiten wird mit wachsendem n im Wesentlichen durch die höchste Potenz von n bestimmt. Die Terme niedrigerer Ordnung in der Formel für die Komplexität wirken sich immer weniger aus.

Die O-Notation (Landau-Symbole)

Für einen theoretischen Aufwandsvergleich von Algorithmen genügt meist bereits die Kenntnis der Größenordnungen von Laufzeit- und Speicherkomplexität. Der folgende Begriff ist eine mathematische Präzisierung hiervon:

Definition (O-Notation, Landau-Symbole)

Es sei $I \subseteq \mathbb{R}$ ein Intervall und

$$f, g : I \mapsto R$$

seien zwei Funktionen. Sei $x_0 \in \mathbb{R}$. Die Funktion f heißt von der Ordnung $O(g(x))$ für $x \in x_0$, wenn es eine Konstante $C > 0$ und ein $\delta > 0$ gibt, so dass die folgende Ungleichung gilt:

$$|f(x)| \leq C|g(x)| \text{ für alle } x \in I \text{ mit } |x - x_0| < \delta.$$

Die O-Notation (Landau-Symbole)

Definition (O-Notation, Landau-Symbole) Teil 2

Die Funktion f heißt von der Ordnung $O(g(x))$ für $x \to \infty$ (bzw. $x \to -\infty$), wenn es eine Konstante $C > 0$ und ein $M \in R$ gibt, so dass die folgende Ungleichung gilt:

$$|f(x)| \leq C|g(x)| \text{ für alle } x \in I \text{ mit } x \geq M(bzw. x \leq M).$$

Bemerkung. Man beachte, dass in der Definition nicht gefordert wird, dass $x_0 \in I$ gilt. Die Funktionen f und g müssen an dieser Stelle gar nicht definiert sein, sondern es genügt, dass man der Stelle $x_0 \in \mathbb{R}$ mit Punkten aus dem Intervall I beliebig nahe kommen kann.

Ausdrücke in O-Notation

- Mit Hilfe des Mittelwertsatzes der Differentialrechnung zeigt man leicht:
$$\sin(x) = O(x),\ x \to 0.$$
- Es gilt auch:
$$\sinh(x) = O(x),\ x \to 0.$$
- Ebenfalls mit Hilfe des Mittelwertsatzes erhält man
$$e^x = O\left(\frac{1}{1-x}\right),\ x \to 0.$$

Ausdrücke in O-Notation

Im Zusammenhang mit dem Aufwand für Algorithmen werden wir die O-Notation hauptsächlich für Zahlenfolgen verwenden. Wir betrachten also die asymptotische Entwicklung der Komplexität für wachsendes n.

Definition (O-Notation für Folgen)

Seien $(f_n)_{n=1}^{\infty}$ und $(g_n)_{n=1}^{\infty}$ zwei reelle Zahlenfolgen, so heißt f_n von der Ordnung $O(g_n)$, wenn es eine Konstante $C > 0$ und ein $n_0 \in \mathbb{N}$ gibt mit

$$\forall n > n0 : |f_n| \leq C|g_n|.$$

Man schreibt: $f_n = O(g_n)$ (für $n \to \infty$).

Ausdrücke in O-Notation

- Vektoren und euklidisches Skalarprodukt auf R^n :

$$Mem(n) = n = O(n), Op(n) = 2n - 1 = O(n).$$

- Matrix-Vektor-Produkt im R^n mit quadratischer Matrix:

$$Mem(n, n) = n^2 + n = O(n^2), Op(n, n) = 2n^2 - n = O(n^2).$$

- Matrizenprodukt $C = AB$ für $A, B \in R^{n \times n}$:

$$Mem(n, n) = 3n^2 = O(n^2), Op(n, n) = 2n^3 - n2 = O(n^3).$$

- Dyadisches Produkt auf R^n :
  Speicheraufwand für $A = uv^\top$:

$$Mem(n, n) = 2n = O(n).$$

  Matrix-Vektor-Produkt $uv^\top x$:

$$Op(n, n) = 3n - 1 = O(n)$$

Die Kondition eines Problems

Folgende Fehlerquellen treten u.a. auf:

- Messfehler in den Eingabedaten,
- Rundungsfehler bei der Gleitpunktarithmetik,
- Verfahrensfehler im Algorithmus (Approximationen, Vereinfachungen etc.).

Fragestellung

Wie pflanzen sich Fehler in den Eingabedaten entlang des Lösungsweges fort und wie stark wirken sie sich auf die Ergebnisse aus?

Dabei spielt nicht nur die konkrete Wahl der Lösungstrategie eine Rolle, sondern bereits die Aufgabenstellung gibt vor, wie stark sich die Eingabefehler in den Resultaten niederschlagen.

Die entscheidende Größe bei diesen Betrachtungen ist

Definition (Relativer Fehler (relative Abweichung) in x)

$$e_{rel}(x) = \frac{|x - \tilde{x}|}{|x|}$$

$\tilde{x}$ bezeichnet hierbei die fehlerbehaftete Größe.

Definition (Konditionierung)

Ein Problem heißt

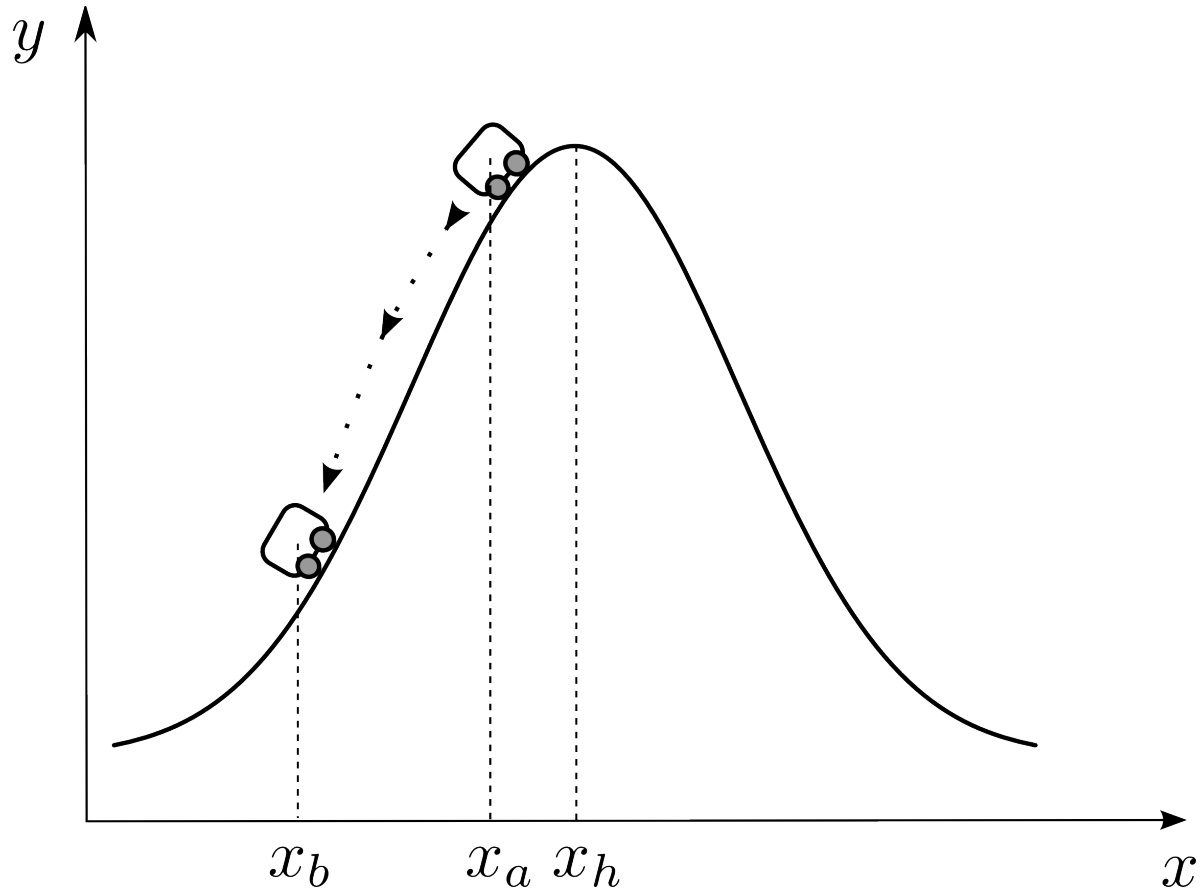
gut konditioniert, wenn sich die relativen Fehler in den Eingabedaten nur gering bzw. mäßig auf das Ergebnis auswirken,

schlecht konditioniert, wenn sich die relativen Fehler in den Eingabedaten erheblich auf das Ergebnis auswirken können.

Man mache sich folgenden Sachverhalt ganz klar:

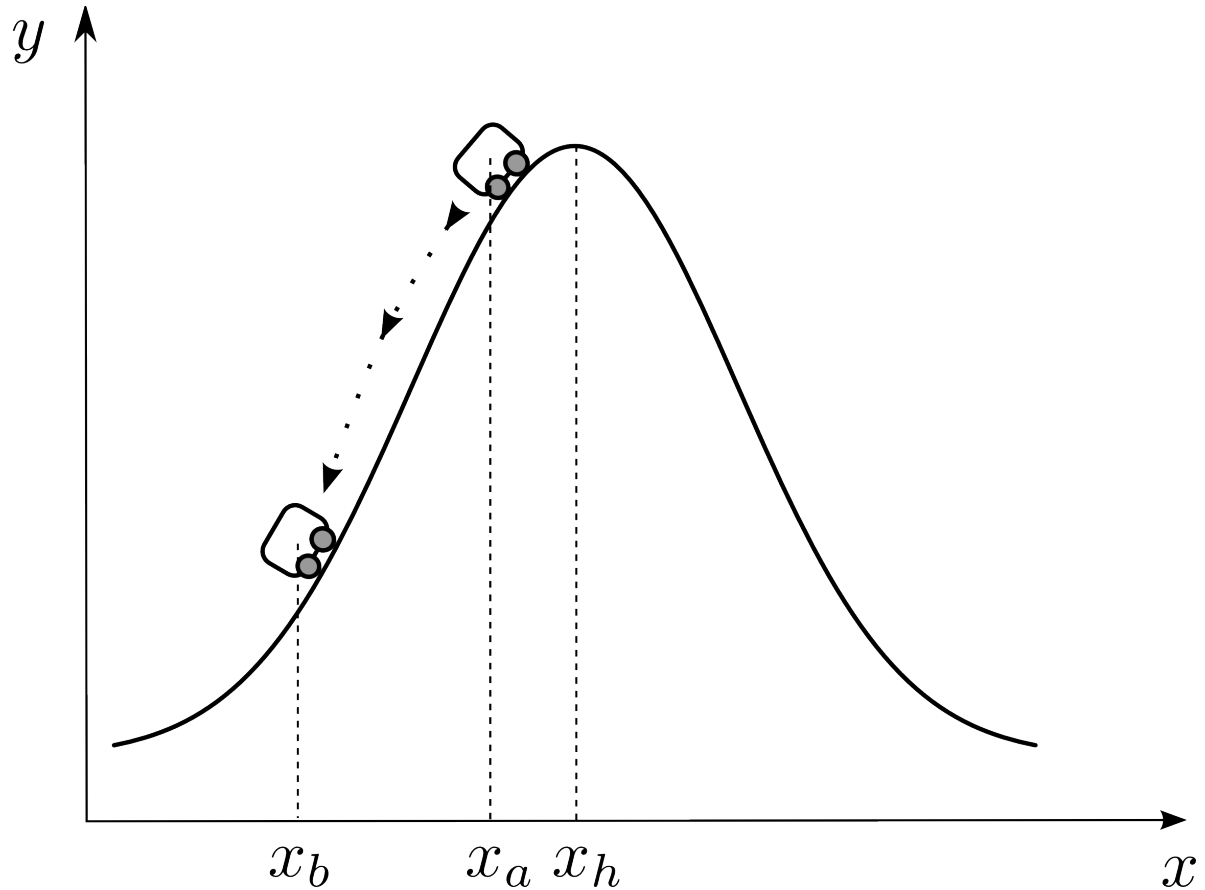
Die Kondition ist eine Eigenschaft des Problems selbst und unabhängig von der Lösungsmethode.

Beispiel 1



x_b = Position des Fahrzeugs nach 10 Sekunden nach dem Start.

Beispiel 1



Unabhängig von Lösungsmethoden ist das Problem schlecht konditioniert wenn, $x_a \approx x_h$.

Aufgabe F1

Gegeben: $x \in \mathbb{R}$
Zu finden: $y = f(x)$, wobei $f(x)$ eine glatte Funktion.

Ist die Aufgabe F1 gut oder schlecht konditioniert?
Beispiel: $y = kx$.

$$\Delta y = f(x + \Delta x) - f(x) = k\Delta x$$

$$e_{rel}(y) = |\frac{\Delta y}{y}| = |\frac{k\Delta x}{kx}| = e_{rel}(x)$$

Die Berechnung von $y = kx$ ist eine gut konditionierte Aufgabe!

Satz

Wenn f differenzierbar ist, dann

$$\Delta y = f(x + \Delta x) - f(x) = f'(x)\Delta x + \varphi(\Delta x)\Delta x,$$

wobei $\lim_{\Delta x \to 0} \varphi(\Delta x) = 0$.

Somit $\boxed{\Delta y \approx f'(x)\Delta x,}$ für kleine Δx.

Allgemein gilt für $y = f(x_1, x_2 \ldots, x_k)$

$$\Delta y \approx \sum_{i=1}^{k} \frac{\partial f}{\partial x_i} \Delta x_i,$$

solange f differenzierbar ist. *Fehler addieren sich!*

Auswertungen: Beispiele

Multiplikation: $y = x_1 x_2$

$$\Delta y = x_2 \Delta x_1 + x_1 \Delta x_2$$

$$\frac{\Delta y}{y} = \frac{\Delta x_1}{x_1} + \frac{\Delta x_2}{x_2}$$

$$e_{rel}(y) \leq e_{rel}(x_1) + e_{rel}(x_2)$$

Addition: $y = x_1 + x_2$

$$\Delta y = \Delta x_1 + \Delta x_2$$

$$\frac{\Delta y}{y} = \frac{x_1}{x_1 + x_2} \frac{\Delta x_1}{x_1} + \frac{x_2}{x_1 + x_2} \frac{\Delta x_2}{x_2}$$

$$e_{rel}(y) \leq \frac{|x_1|}{|x_1 + x_2|} e_{rel}(x_1) + \frac{|x_2|}{|x_1 + x_2|} e_{rel}(x_2)$$

Die Addition ist schlecht konditioniert wenn $x_1 \approx -x_2$!

Stabilität

Definition

Ein Algorithmus heißt stabil, wenn sich Rundungs- und Verfahrensfehler nur mäßig auf die vom ihm gelieferten Ergebnissen niederschlagen.

- Im Gegensatz zur Kondition ist die Stabilität also eine Eigenschaft der konkret zum Einsatz kommmenden Lösungsmethode.
- Instabiles Verhalten ist bei einem Algorithmus u.a. dann zu erwarten, wenn er die Lösung von schlecht konditionierten (Teil-)Problemen beinhaltet.

Beispiel

Gegeben: $p, q \in \mathbb{R}$.
Zu finden: Alle $x \in \mathbb{R}$, für welche $x^2 + px + q = 0$.
Lösungsmethode:

$$x_{1,2} = \frac{-p \pm \sqrt{p^2 - 4q}}{2}$$

Sei $p^2 \gg 4q$ und $p < 0$, dann $-p \approx \sqrt{p^2 - 4q}$, und die Formel für x_1 ist schlecht konditioniert. Deswegen, bekommt man in dieser Situation den stabilen Algorithmus durch die Berechnung von x_1 mit $x_1 = q/x_2$.