Programmierkurs in C

Universität Augsburg

Wintersemester 2017 / 2018

Prof. Dr. Robert Lorenz

Lehrprofessur für Informatik

Intervallschachtelung

Inhalt

<u>Anwendung des Prinzips der Intervallschachtelung:</u>

Implementierung von C-Funktionen, die per Konstruktion korrekt sind

Implementierung mathematischer Näherungsverfahren

Fehlerbehandlung in C (Eingabefehler vs Logikfehler)

Lernziele

Das Invarianten-Prinzip verstehen und anwenden können

Das Prinzip der Intervallschachtelung verstehen und anwenden können

Mathematische (geometrische) Beschreibungen von Algorithmen in C-Code umsetzen können

Per Konstruktion korrekte und effiziente Algorithmen entwerfen können

Konstruktion korrekter Algorithmen

Berechnung von Werten ganzzahliger Funktionen:

Ganzzahliger Quotient
Ganzzahlige Quadratwurzel
Ganzzahliger Logarithmus
...

Hier gibt es eine Lösungsstrategie, die per Konstruktion zu (partiell) korrekten Algorithmen führt.

Sie beruht auf der Idee, den gesuchten Funktionswert durch ein immer kleiner werdendes Intervall einzugrenzen.

Wiederholung: Schleifeninvarianten

Allgemein:

Eine Schleifeninvariante **I** ist vor und nach jeder Ausführung des Schleifenrumpfs gültig.

```
A
{I}
WHILE B DO
      {I ∧ B}
      S
      {I}
END(*WHILE*)
{I ∧ ¬B}
```

Wiederholung: Schleifeninvarianten

Beispiel: Summation über **`int a[M]`** mit **`M > 0`**

Problemspezifikation: Ausgabe **`s = a[0] + … + a[M-1]`**

`s` Akkumulatorvariable

```
s := 0; i := 0;
{}
WHILE B DO
      {}
      s := s + a[i];
      i := i + 1 ;
      {}
END(*WHILE*)
{}
```

Wiederholung: Schleifeninvarianten

Beispiel: Summation über **`int a[M]`** mit **`M > 0`**

Problemspezifikation: Ausgabe **`s = a[0] + … + a[M-1]`**

`s` Akkumulatorvariable

```
s := 0; i := 0;
{i ≤ M}
WHILE (i < M) DO
      {i ≤ M ∧ i < M}
      s := s + a[i];
      i := i + 1 ;
      {i ≤ M}
END(*WHILE*)
{i ≤ M ∧ i ≥ M}
```

Wiederholung:Schleifeninvarianten

Beispiel: Summation über **`int a[M]`** mit **`M > 0`**

Problemspezifikation: Ausgabe **`s = a[0] + … + a[M-1]`**

`tsum(a;i) := a[0] + … + a[i-1]`

Schleifeninvariante: **(i ≤ M) ∧ (s = tsum(a;i))**

```
s := 0; i := 0;
{(i ≤ M) ∧ (s = tsum(a;i))}
WHILE (i < M) DO
      {((i ≤ M) ∧ (i < M)) ∧ (s = tsum(a;i))}
      s := s + a[i];
      i := i + 1 ;
      {(i ≤ M) ∧ (s = tsum(a;i))}
END(*WHILE*)
{((i ≤ M) ∧ (i ≥ M)) ∧ (s = tsum(a;i))}
```

Theorem

Gegeben: Programm mit Spezifikation **N:**

```
A;
{V}
WHILE B DO S END(*WHILE*)
{N}
```

Es gelte:

Nach **A** gilt **V**

$V \Rightarrow I$ (In-Kraft-Setzen der Invariante, Induktionsanfang)

I Schleifeninvariante (Erhalte Invariante, Induktionsschritt)

$(I \wedge \neg B) \Rightarrow N$ (Schlussbetrachtung)

Dann folgt:

Das Programm ist bezüglich seiner Zusicherungen korrekt.
(**N** ist nach Programmende wahr)

Theorem

Gegeben: Programm mit Spezifikation **N:**

```
A;
{V}
WHILE B DO S END(*WHILE*)
{N}
```

Es gelte:

- `Nach` **A** `gilt` **V**
- **V** $\Rightarrow$ **I** (In-Kraft-Setzen der Invariante, Induktionsanfang)
- **I** Schleifeninvariante (Erhalte Invariante, Induktionsschritt)
- **(I** $\wedge$ $\neg$**B)** $\Rightarrow$ **N** (Schlussbetrachtung)

Dann folgt:

Das Programm ist bezüglich seiner Zusicherungen korrekt.
(**N** ist nach Programmende wahr))

Übliches Vorgehen: Konstruiere **A,B,S** aus **N** und finde dazu **V,I**

Theorem

Gegeben: Programm mit Spezifikation **N**:

```
A;
{V}
WHILE B DO S END(*WHILE*)
{N}
```

Es gelte:

Nach **A** gilt **V**

$V \Rightarrow I$ (In-Kraft-Setzen der Invariante, Induktionsanfang)

I Schleifeninvariante (Erhalte Invariante, Induktionsschritt)

$(I \land \neg B) \Rightarrow N$ (Schlussbetrachtung)

Dann folgt:

Das Programm ist bezüglich seiner Zusicherungen korrekt.
(**N** ist nach Programmende wahr))

Neues Vorgehen: Konstruiere **V**,**I** aus **N** und finde dazu **A**,**B**,**S**

Intervallschachtelung

Konstruiere zuerst korrekte Zusicherung, dann das Programm

1. Wir geben als Spezifikation jeweils die gewünschte Nachbedingung **N** an

2. Dann konstruieren wir unter Verwendung des Theorems eine Schleife, die **N** herstellt

Intervallschachtelung

Gesucht (zu berechnen):

Ein ganzzahliger Wert **k**

Ziel:

Finde endliche Folge von ganzzahligen Intervallen

$[u_1, o_1] \supset \ldots \supset [u_j, o_j] \ni k$ mit $k = u_j$ oder $k = o_j$

Intervallschachtelung

<u>Gesucht (zu berechnen):</u>

Ein ganzzahliger Wert **k**

<u>Ziel:</u>

Finde endliche Folge von ganzzahligen Intervallen

$[u_1, o_1] \supset \ldots \supset [u_j, o_j] \ni k$ mit $k = u_j$ oder $k = o_j$

Intervallschachtelung

Gesucht (zu berechnen):

Ein ganzzahliger Wert **k**

Ziel:

Finde endliche Folge von ganzzahligen Intervallen

$[u_1, o_1] \supset \ldots \supset [u_j, o_j] \ni k$ mit $k = u_j$ oder $k = o_j$

Beispiel 1: Berechnung des ganzzahligen Quotienten

$\mathbf{iquot} : \mathrm{IN} \times \mathrm{IN} \setminus \{0\} \rightarrow \mathrm{IN}$

$k = \mathrm{iquot}(n,m) :\Leftrightarrow k * m <= n \wedge (k + 1) * m > n$

Intervallschachtelung

Beispiel 1: Berechnung des ganzzahligen Quotienten

$$\mathbf{iquot} : \mathbf{IN} \times \mathbf{IN}\backslash\{0\} \rightarrow \mathbf{IN}$$

$$\mathbf{k = iquot(n,m)} :\Leftrightarrow \mathbf{k * m <= n} \wedge \mathbf{(k + 1) * m > n}$$

Mögliches Startintervall:

$$[u_1,o_1] = [0,n]$$

Mögliche Strategie:

Einengung des Intervalls auf der linken Seite bis **k** gefunden:

Finde endliche Folge von ganzzahligen Intervallen

$$[u_1,o_1] \supset \ldots \supset [u_j,o_j] \ni k \text{ mit } k = u_j$$

Intervallschachtelung

Beispiel 1: Berechnung des ganzzahligen Quotienten

$$\mathbf{iquot} : \mathrm{IN} \times \mathrm{IN}\setminus\{0\} \rightarrow \mathrm{IN}$$

$$\mathbf{k = iquot(n,m)} \;:\Leftrightarrow\; \mathbf{k * m <= n} \wedge \mathbf{(k + 1) * m > n}$$

Mögliches Startintervall:

$$[\mathbf{u}_1,\mathbf{o}_1] = [0,\mathbf{n}]$$

Mögliche Strategie:

Einengung des Intervalls auf der linken Seite bis **k** gefunden:
Finde endliche Folge von ganzzahligen Intervallen

$$[\mathbf{u}_1,\mathbf{o}_1] \supset \ldots \supset [\mathbf{u}_j,\mathbf{o}_j] \ni \mathbf{k} \text{ mit } \mathbf{k} = \mathbf{u}_j$$

Idee: Erhöhe, wenn möglich, linke Grenze **u** um **1** : $\mathbf{u}_{i+1} = \mathbf{u}_i + 1$

Intervallschachtelung

Beispiel 1: Berechnung des ganzzahligen Quotienten

$$\texttt{iquot} : \mathbb{IN} \times \mathbb{IN}\setminus\{0\} \rightarrow \mathbb{IN}$$

$$k = \texttt{iquot}(n,m) :\Leftrightarrow k * m <= n \wedge (k + 1) * m > n$$

```
u := 0;
{n >= 0 ∧ m > 0 ∧ u = 0} /* V */
{u * m <= n} /* I */
WHILE (u + 1) * m <= n DO
      {u * m <= n ∧ (u + 1) * m <= n} /* I && B */
      u := u + 1;
      {u * m <= n} /* I */
END(*WHILE*)
{u * m <= n ∧ (u + 1) * m > n} /* (I && !B) == N */
```

Nach Ende der Schleife stimmen **u** und **k** also überein!

Intervallschachtelung

Beispiel 1: Berechnung des ganzzahligen Quotienten

$$\mathbf{iquot} : \mathrm{IN} \times \mathrm{IN}\backslash\{0\} \rightarrow \mathrm{IN}$$

$$k = \mathrm{iquot}(n,m) :\Leftrightarrow k * m <= n \wedge (k + 1) * m > n$$

```
u := 0;
WHILE (u + 1) * m <= n DO
      u := u + 1;
END(*WHILE*)
```

Verbesserung:
Vermeide die Berechnung von **`(u + 1) * m`** in jedem Iterationsschritt. (Multiplikationen sind aufwendig)

Intervallschachtelung

Beispiel 1: Berechnung des ganzzahligen Quotienten

$$\mathtt{iquot} : \mathrm{IN} \times \mathrm{IN}\backslash\{0\} \rightarrow \mathrm{IN}$$

$$k = \mathtt{iquot}(n,m) :\Leftrightarrow k * m <= n \wedge (k + 1) * m > n$$

```
u := 0;
WHILE (u + 1) * m <= n DO
      u := u + 1;
END(*WHILE*)
```

Prinzip der Fortschaltung:

Setze **(u + 1) * m =: h** für eine Hilfsvariable **h**

Erweitere Invariante um **(u + 1) * m = h**

Schalte **h** durch *effizientere* Operationen in der Schleife fort:

((u + 1) + 1) * m = (u + 1) * m + m = h + m

Intervallschachtelung

Beispiel 1: Berechnung des ganzzahligen Quotienten

```
   iquot : IN × IN\{0} → IN
k = iquot(n,m) :⇔ k * m <= n ∧ (k + 1) * m > n

u := 0; h := m;
{n >= 0 ∧ m > 0 ∧ u = 0 ∧ h = m} /* V */
{u * m <= n ∧ h = (u + 1) * m} /* I */
WHILE h <= n DO
     {u * m <= n ∧ h <= n ∧ h = (u + 1) * m} /* I && B */
      u := u + 1;
     {u * m <= n ∧ h = u * m}
      h := h + m;
     {u * m <= n ∧ h = (u + 1) * m} /* I */
END(*WHILE*)
{u * m <= n ∧ h > n ∧ h = (u + 1) * m} /* I && !B */
{u * m <= n ∧ (u + 1) * m > n} /* N */
```

Intervallschachtelung

Beispiel 2: Berechnung der ganzzahligen Wurzel

$\texttt{isqrt : IN} \rightarrow \texttt{IN}$

$\texttt{k = isqrt(n)} :\Leftrightarrow k^2 \texttt{ <= n} \wedge (k + 1)^2 > n$

Ziel:

Finde endliche Folge von ganzzahligen Intervallen

$[u_1, o_1] \supset \ldots \supset [u_j, o_j] \ni k$ mit $k = u_j$

Startintervall: `[0,n]`

Programm: Übungsaufgabe (zuerst ohne, dann mit Fortschaltungstechnik)

Intervallschachtelung

Beispiel 3: Berechnung des ganzzahligen Logarithmus

ilog : IN\{0} → IN

k = ilog(n) $:\Leftrightarrow$ **2^k <= n** $\wedge$ **2^{k+1} > n**

Ziel:

Finde endliche Folge von ganzzahligen Intervallen

$[u_1, o_1] \supset \ldots \supset [u_j, o_j] \ni k$ mit $k = u_j$

Startintervall: **[0,n]**

Programm: Übungsaufgabe (zuerst ohne, dann mit Fortschaltungstechnik)

Intervallschachtelung

Nachteil der bisherigen Verfahren:

Zeitkomplexität `O(n)` (Maximale Anzahl Schleifendurchläufe)

Grund: Intervall wird pro Schleifendurchlauf nur `1` kürzer

Verbesserung:

Halbiere pro Schleifendurchlauf die Intervalllänge (ähnlich wie beim binären Suchen)

Zeitkomplexität `O(log(n))`

Intervallschachtelung

Beispiel 1: Berechnung des ganzzahligen Quotienten NEU

$$\mathbf{iquot} : \mathrm{IN} \times \mathrm{IN}\setminus\{0\} \rightarrow \mathrm{IN}$$

$$k = \mathrm{iquot}(n,m) :\Leftrightarrow k * m <= n \wedge (k + 1) * m > n$$

Mögliches Startintervall:

$$[u_1, o_1) = [0, n + 1)$$

Mögliche Strategie:

Halbiere Intervall bis **k** gefunden:

Finde endliche Folge von ganzzahligen Intervallen

$[u_1, o_1) \supset \ldots \supset [u_j, o_j) \ni k$ mit $k = u_j = (o_j - 1)$

und $o_{i+1} - u_{i+1} = (o_i - u_i) / 2$

Intervallschachtelung

Beispiel 1: Berechnung des ganzzahligen Quotienten NEU

$$\mathtt{iquot} : \mathrm{IN} \times \mathrm{IN}\backslash\{0\} \to \mathrm{IN}$$

$$\mathtt{k = iquot(n,m)} :\Leftrightarrow \mathtt{k * m <= n} \wedge \mathtt{(k + 1) * m > n}$$

```
u := 0; o := n + 1;
{n >= 0 ∧ m > 0 ∧ u = 0 ∧ o = n + 1} /* V */
{u * m <= n ∧ n < o * m ∧ u <= (o – 1)} /* I */
WHILE u < (o – 1) DO
      {u*m <= n ∧ n < o*m ∧ u <= (o–1) ∧ u < (o–1)} /* I && B*/
      a := (u + o) div 2;
      if (a * m <= n){ u := a; }
      else { o := a; }
      {u * m <= n ∧ n < o * m ∧ u <= (o – 1)} /* I */
END(*WHILE*)
{ u*m <= n ∧ n < o*m ∧ u <= (o – 1) ∧ u >= (o – 1)} /*I && !B*/
{ u*m <= n ∧ (u + 1) * m > n} /* N */
```

Fehlerbehandlung in C

Falsche Benutzereingaben (Systemzustände):

- Müssen *abgefangen* …
- … und *behandelt* werden, ohne dass das Programm unerwartetes Verhalten aufweist (z.B. crash).

<u>Vorgehensweise:</u>

- Eingaben, die zur Laufzeit (z.B. durch einen Benutzer) erfolgen, müssen auf Korrektheit überprüft werden.
- Im Fall einer ungültigen Eingabe, wird das System wieder in einen gültigen Zustand überführt und der Nutzer ggf. auf den Fehler aufmerksam gemacht.

Fehlerbehandlung in C

Falsche Benutzereingaben (Systemzustände):

<u>Beispiel:</u>

- **int read_`semester`(void)**: Liest Benutzereingabe mit **`scanf`** ein.
- Die Eingabe wird auf Korrektheit überprüft.
- Im Erfolgsfall gibt die Funktion das eingelesene Semester zurück.
- Im Fehlerfall gibt die Funktion einen Fehlerrückgabewert (Wert, der nicht im Bereich der möglichen Eingaben liegt) zurück.
- Aufrufer der Funktion überprüft Rückgabewert auf Fehler.

Fehlerbehandlung in C

Logikfehler (Programmierfehler):

- Werden im Allgemeinen nicht abgefangen, sondern entdeckt und behoben.

Beim Kunden (*release version*):
- Durch Logging von z.B. stack traces.

In der Entwicklung (development version):
- Durch Testverfahren (Unit-Tests, System-Tests,…)
- Durch Zusicherungen (**`assert`**-Funktion)

Fehlerbehandlung in C

Logikfehler (Programmierfehler):

Beispiel:

`double sqrt(double x)`:Berechnet die Quadratwurzel von x.

- Es gibt *keinen Fehlerrückgabewert* für ungültige Parameterwerte. Es liegt in der Verantwortung des Aufrufers, die Parameter vor Funktionsaufruf auf Korrektheit zu überprüfen (Programmierfehler).
- Um solche ungültigen Eingaben in der Entwicklung schnell zu finden, wird zu Beginn der Funktion eine Zusicherung verwendet:

```
assert(x >= 0)
```

Fehlerbehandlung in C

Logikfehler (Programmierfehler):

`void assert(int expression)` aus **`assert.h`**:
- Falls der Wahrheitswert des übergebenen Ausdrucks 0 ist, gibt die Funktion eine *Fehlermeldung* aus und *bricht das Programm ab*.
- Solche Programmabbrüche sind nur in der Entwicklung gewünscht, nicht jedoch in der release version!
- **`asserts`** können leicht deaktiviert werden, indem das Makro **`NDEBUG`** gesetzt wird.
- Zum Beispiel als Compiler-Schalter: **`-DNDEBUG`**
- *Achtung*: Aus diesem Grund keinen Code in asserts schreiben, der relevant für den Programmablauf ist! (z.B. **`assert(++x)`**)

Näherungslösungen

Probleme beim Rechnen mit Maschinen:

Die meisten Werte können nur näherungsweise dargestellt werden

- Rundungsfehler
- Mathematische Forschung: Begrenzung solcher Rundungsfehler

Wir kennen einige Grundregeln zur Auswertungsreihenfolge

Für Berechnungen haben wir nur Grundoperationen

- Genügen nicht für irrationale Zahlen; **`e,π,...`**
- Genügen nicht zur Auswertung von Funktionen: **`sqrt(x),log(x),..`**
- Genügen nicht für Nullstellenberechnungen, Integration von Funktionen

Näherungslösungen

Auswertung von Funktionen:

Lösung 1: Wertetabellen speichern

Zu großer Speicheraufwand
Wie werden die Werte zum 1. Mal berechnet?

Lösung 2: Mathematische Forschung zu effizienten Näherungsverfahren

Wir wollen einige einfache Ideen besprechen, die sich unmittelbar einsehen und umsetzen lassen.

Näherungslösungen

Gesucht (zu berechnen):

Ein reeller Wert $\mathbf{k}$

Ziel:

Finde endliche Folge von reellen Intervallen

$[u_1, o_1] \supset [u_2, o_2] \supset \ldots \supset [u_n, o_n] \ni k$

mit $(o_n - u_n) \to 0$ für $n \to \infty$

Idee:

Gib (sehr) kleine Zahl ε vor und rechne bis $(o_j - u_j) < \varepsilon$

$(o_j - u_j)/2$ ist Näherungslösung für k mit Fehler unter $\varepsilon/2$

Näherungslösungen

<u>Beispiel 5: Berechnung der Eulerschen Zahl $e = \exp(1)$</u>

Manchmal genügt es, die untere Folge $u_1, u_2, ...$ zu berechnen

Maß für die Genauigkeit der Näherung: Abstand zwischen den beiden zuletzt berechneten Elementen der Folge $|u_{n+1}-u_n|$
Abbruch, falls $|u_{n+1}-u_n| < \varepsilon/2$

Notwendig: Mathematischer Beweis, um von diesem Abstand auf den Näherungsfehler zu schließen

Unbewiesen glauben wir:

$$e = 1/0! + 1/1! + 1/2! + 1/3! + \ldots$$

Näherungslösungen

Beispiel 5: Berechnung der Eulerschen Zahl **`e = exp(1)`**

Unbewiesen glauben wir:

$$e = 1/0! + 1/1! + 1/2! + 1/3! + \dots$$

Wir setzen:

$$u_0 = 1$$

$$u_{n+1} = u_n + 1 / (n + 1)!$$

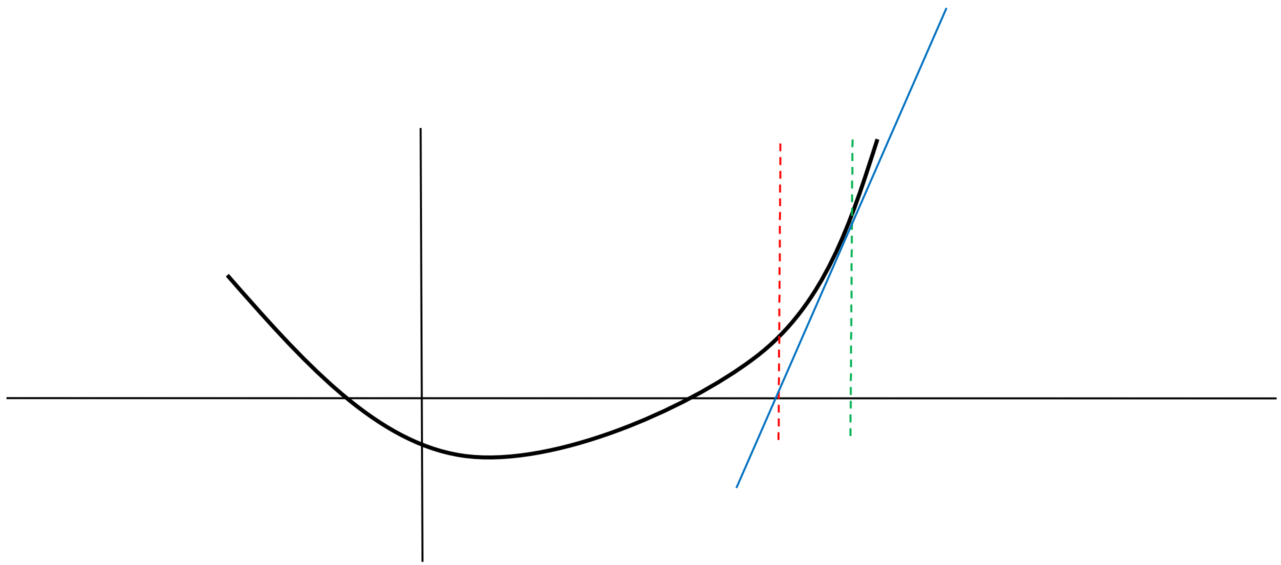
$$(u_{n+1} - u_n) \rightarrow 0$$

Für den Näherungsfehler **`r`** gilt:

$$|r| <= 2 * 1/(n+1)!$$

Näherungslösungen

Beispiel 6: Nullstellenberechnung (Newton-Verfahren)



Startwert: x_1

Tangente an $(x_1, f(x_1))$

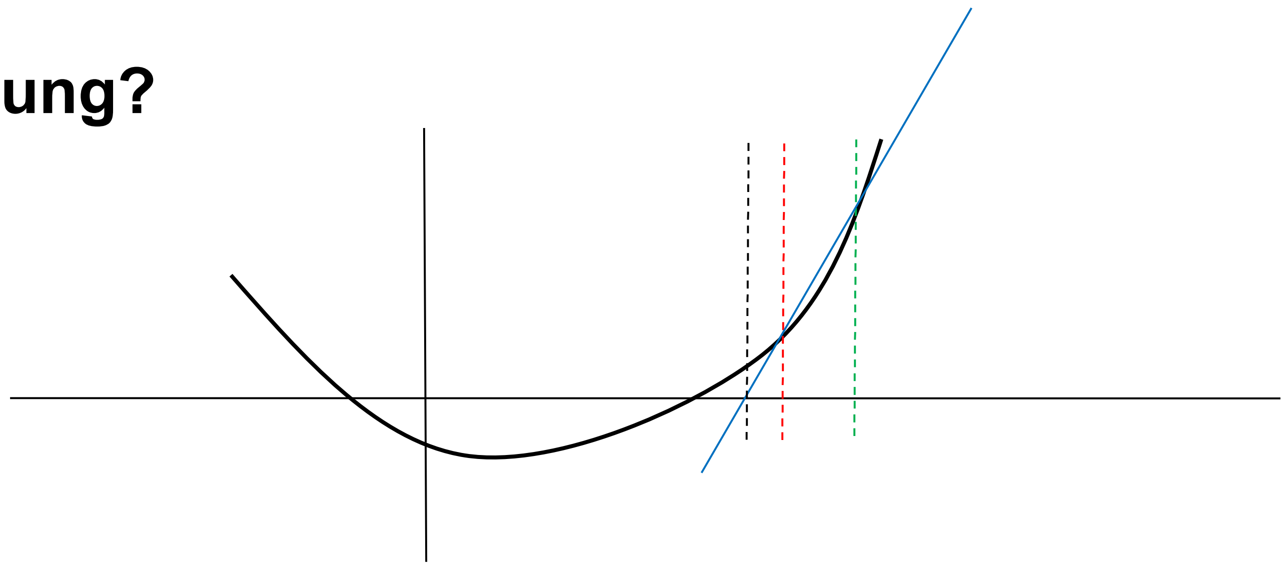
Schnittpunkt Tangente mit x-Achse: x_2

$x_{n+1} = x_n - f(x_n) / f`(x_n)$ (Funktion: $f$; Ableitung: $f`$)

Näherungslösungen

Beispiel 6: Nullstellenberechnung (Newton-Verfahren)

Unbekannte Ableitung?



Startwert 1: $\mathbf{x_1}$

Startwert 2: $\mathbf{x_2}$

Sekante durch $(x_1, f(x_1))$ und $(x_2, f(x_2))$

Schnittpunkt Sekante mit $\mathbf{x}$-Achse: $\mathbf{x_3}$

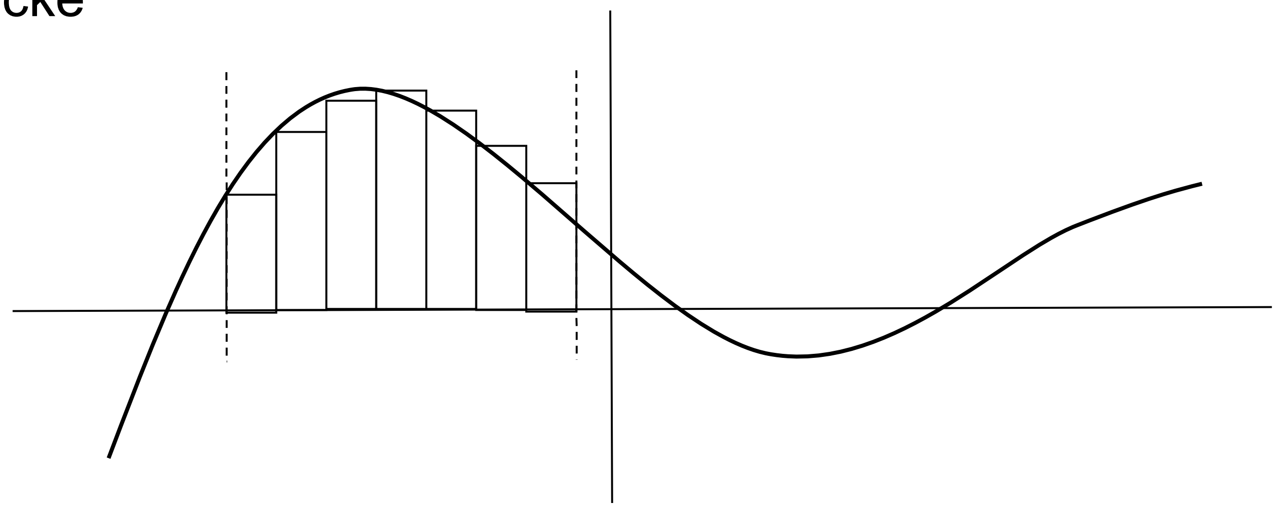
$$x_{n+1} = x_n - (f(x_n) * (x_{n-1} - x_n)) / (f(x_{n-1}) - f(x_n))$$

Näherungslösungen

<u>Beispiel 7: Flächenintegrale (naive Methode)</u>

Nähere Fläche unter einer Funktionskurve durch eine Summe von Rechtecksflächen an

Verkleinere (z.B. halbiere) sukzessive die Rechtecksbreite und vergrößere so die Anzahl der Rechtecke



Als Höhe kann man einen beliebígen Funktionswert im Intervall nehmen.

Näherungslösungen

Beispiel 7: Flächenintegrale (Allgemein)

Interpoliere die Funktion durch ein Polynom und berechne die Fläche unter der Polynomkurve (exakt bestimmbar).

Man speichert Funktionswerte zu einer kleinen Anzahl von Stützstellen (10 – 12), die man einmal ausrechnen muss.

Man **interpoliert die Funktion stückweise durch Polynome** (von Grad 10-12), die an den Stützstellen die gleichen Werte haben, und integriert die Polynomstücke.

Mathematische Forschung: Minimierung des maximalen Fehlers.

Sie wären jetzt grundsätzlich in der Lage, auch diese Methoden in C zu implementieren (Zusatzbetrachtungen notwendig: Auswertung von Polynomen, Polynomdivision, …).

Zeiger auf Funktionen

Variablendefinition

Klammern wichtig!

<return_type> (*<name>) (<arg_list>);

Beispiele

```
void (*ptr)(void);
```

Zeiger auf Funktion ohne Rückgabewerte und Argumente

```
double (*gptr)(double);
```

Zeiger auf Funktion mit Rückgabetyp double und Argument vom Typ double

```
int (*fptr)(int, double);
```

Zeiger auf Funktion mit Rückgabetyp int und zwei Argumenten vom Typ int und double

Zeiger auf Funktionen

Beispiel

```
#include <stdio.h>

double square(double x) {
    return x*x;
}

double lin(double x) {
    return 2*x;
}

void printFct( double(*fct)(double), double x) {
     printf("Funktion(%2.f) = %2.f \n", x, fct(x));
}

int main(void)
{
    double (*p) (double);

    p = square;

    printFct(p, 3);
    printFct(lin,2);

    return 0;
}
```

//Aufruf auch mit (*fct)(x) möglich

//Zuweisung p = &square auch möglich